UNIVERSITÉ DU QUÉBEC À MONTRÉAL

Propriétés combinatoires des f-palindromes

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

SÉBASTIEN LABBÉ

OCTOBRE 2008

REMERCIEMENTS

Je désire remercier avant tout le Pr Srečko Brlek, mon directeur de recherche, pour sa grande disponibilité malgré ses responsabilités de directeur du Laboratoire de combinatoire et d'informatique mathématique (LaCIM). Il m'a encouragé à présenter nos résultats depuis le début et m'a guidé dans la rédaction d'articles. Il m'a aussi permis de travailler avec d'éminents chercheurs français et italiens. J'ai pu ainsi rencontrer Valérie Berthé (LIRMM, Montpellier), Laurent Vuillon (LAMA, Chambéry où j'ai séjourné un mois), Simone Rinaldi et Andrea Frosini (Université de Sienne).

Merci à Christophe Reutenauer, professeur à l'UQAM et membre du LaCIM, qui m'a consacré du temps et qui a discuté avec moi sur un automate de chevauchement alors que mes idées n'étaient pas encore tout à fait limpides.

Merci à Alexandre Blondin Massé, étudiant à la maîtrise en mathématiques de l'UQAM. Nos discussions sur l'exposant critique et les lacunes palindromiques du mot de Thue-Morse étaient très plaisantes et fécondes.

Je veux dire merci aussi à Franco Saliola, postdoctorant au LaCIM, qui a répondu à mes nombreuses questions à propos de Linux, Ubuntu et LaTeX, entre autres, et surtout pour m'avoir fait connaître le logiciel libre Sage.

Merci à Amy Glen, postdoctorante au LaCIM, qui a toujours su répondre à mes questions sur la combinatoire de mots.

De plus, merci à Anne Bergeron, Yannick Gingras, André Lauzon, Pierre Leroux, François Bergeron, Laurent Vuillon, Arnaud Bergeron et Lise Tourigny pour leurs conseils, leur influence ou pour leur soutien au cours des deux dernières années.

Merci aux organismes subventionnaires canadien (Conseil de recherches en sciences

naturelles et en génie du Canada) et québécois (Fonds québécois de la recherche sur la nature et les technologies) qui m'ont permis au cours des deux dernières années de me consacrer pleinement à mes études et recherches.

Merci à mes parents Jean et Madeleine et à mon frère Jean-Philippe pour leur soutien indéfectible. Finalement, merci à Renée avec qui la vie est tellement plus passionnante et qui m'a accompagné pendant ces deux années bien occupées.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

Ce mémoire fait partie du domaine de la combinatoire des mots et plus particulièrement de l'étude de la complexité palindromique (le nombre de facteurs palindromes) des mots infinis. La conjecture de Hof, Knill et Simon, énoncée pour la première fois en 1995, donne une caractérisation des points fixes dont la complexité palindromique est infinie. Récemment, elle a été résolue pour les points fixes sur un alphabet binaire (Tan, 2007). Dans ce mémoire, nous la démontrons pour les points fixes de morphismes *uniformes* sur un alphabet binaire (ce n'est pas plus général que le résultat de Tan). De plus, notre approche permet d'obtenir une démonstration d'un résultat similaire pour les points fixes contenant une infinité d'antipalindromes.

Afin d'atteindre notre objectif, nous établissons un ensemble de résultats combinatoires sur les mots. En effet, nous faisons une étude des f-palindromes et de certaines équations qui en contiennent. Ensuite, nous introduisons les morphismes de classe $\mathcal{P}$, $\mathcal{P}'$ et f-$\mathcal{P}$ et nous démontrons notamment que l'ensemble des morphismes de classe $\mathcal{P}'$ est un monoïde. Nous rassemblons également les résultats d'un travail précédent sur les morphismes conjugués. Finalement, nous étudions les chevauchements de mots et nous construisons un graphe de chevauchements, assise de notre démonstration de la conjecture.

Toutes ces recherches ont contribué au développement d'un outil informatique voué à l'étude de questions soulevées en combinatoire des mots. Ce dernier est constitué d'un ensemble de classes et de fonctions écrites en langage Python annexées à ce mémoire. Elles seront bientôt incluses dans un paquetage sur la combinatoire des mots associé au logiciel libre Sage.

Mots-clés : combinatoire des mots ; f-palindrome ; complexité palindromique ; conjecture de Hof, Knill et Simon ; point fixe de morphisme ; chevauchement ; automates.

INTRODUCTION

Les premières références à la combinatoire des mots apparaissent au XVIIIe siècle, avec Bernouilli, et plus tard au XIXe s. avec Christoffel et Markov (le même qui a donné son nom aux processus de Markov) et au XXe siècle avec Morse qui a développé certaines idées de Birkhoff (et de manière indépendante certaines de Thue). À partir de la moitié du XXe siècle, la combinatoire des mots prend son véritable essor avec l'apparition et le développement des ordinateurs dont on connait aujourd'hui l'omniprésence dans tous les domaines de la vie. La théorie des langages formels en informatique théorique, les logiciels de traitement de textes, l'infographie et le traitement des images (en imagerie médicale par exemple), la télédétection, la télémétrie sont des domaines d'application où la combinatoire des mots a joué un rôle significatif.

Dans l'analyse de la structure des mots, suites de lettres, la recherche des régularités (ou motifs) s'impose, et le calcul de leurs occurrences est une statistique qu'on aime connaître. Par exemple, en 2007, nous avons étudié l'exposant critique des mots de Thue-Morse généralisés et nous avons calculé explicitement les occurrences de leurs facteurs critiques (Blondin Massé, Brlek, Glen, Labbé, 2007; Blondin Massé, Labbé, 2007).

Parmi les motifs intéressants, les palindromes jouent un rôle central pour plusieurs raisons (Allouche, 1997; Allouche, Shallit, 2000; Baake, 1999; de Luca, 1997). Entre autres, ils décrivent la structure des mots de Christoffel, approximations des droites sur un réseau carré, dont l'état actuel des connaissances est présenté de façon complète dans un livre bientôt disponible (Berstel, Lauve, Reutenauer, Saliola, 2008). De plus, démontrer l'existence d'une infinité de palindromes dans la classe des mots lisses résoudrait (Brlek, Ladouceur, 2003) une conjecture (Dekking, 1980) sur la récurrence du mot de Kolakoski. Ainsi, en 2008, nous avons étudié la complexité palindromique, c'est-à-dire le nombre

de facteurs palindromes de longueur n, du mot de Thue-Morse et nous avons déterminé explicitement ses lacunes palindromiques (Blondin Massé, Brlek, Labbé, 2008). Plus récemment, nous avons étudié la réciproque, c'est-à-dire déterminer les mots qui ont une suite donnée de lacunes palindromiques (Blondin Massé, Brlek, Frosini, Labbé, Rinaldi, 2008).

Le sujet de ce mémoire porte sur des mots qui généralisent la notion de palindrome. Au cours de la dernière année, nous avons concentré notre étude sur une conjecture de (Hof, Knill, Simon, 1995) qui donne une caractérisation des points fixes de morphismes ayant une complexité palindromique infinie. Récemment, la conjecture a été démontrée pour les points fixes de morphismes quelconques sur un alphabet binaire (Tan, 2007). Ici, nous avons démontré la conjecture pour les points fixes de morphismes uniformes sur un alphabet à deux lettres ayant une complexité f-palindromique infinie.

Parallèlement au travail théorique, l'étude des problèmes en combinatoire des mots s'accompagne d'expérimentations faites par ordinateur. Ainsi, au cours des deux dernières années, j'ai développé un ensemble de programmes codés dans le langage Python pour étudier les mots, les chevauchements et les morphismes. Cette contribution vient enrichir l'outil de calcul formel développé au Laboratoire de combinatoire et d'informatique mathématique (LaCIM) par mon directeur de recherche (Brlek et al., 2006). Depuis mai 2008, nous travaillons afin de rassembler tous ces algorithmes et, d'ici quelques mois, nous avons l'intention d'inclure dans le logiciel Sage (Sage, 2008) un nouveau paquetage sur la combinatoire des mots (Bergeron, Brlek, Glen, Labbé, Saliola, 2008).

Ce mémoire est divisé en cinq chapitres. D'abord, le chapitre 1 présente la terminologie de la combinatoire des mots et de la théorie des automates. Ensuite, au chapitre 2, on définit les f-palindromes et on présente plusieurs résultats combinatoires utiles pour la suite. Certains sont des généralisations simples de résultats connus sur les palindromes, d'autres sont inédits. Le chapitre 3 rassemble pour la première fois les connaissances sur les morphismes de classe $\mathcal{P}$. De plus, il définit les morphismes de classe f-$\mathcal{P}$. On y trouve aussi une section sur les morphismes conjugués. Enfin, le chapitre 4 introduit les

chevauchements de mots et présente le graphe des chevauchements qui permet d'aborder sous un nouvel angle la conjecture de Hof, Knill et Simon. Finalement, le chapitre 5 résoud la conjecture pour les morphismes uniformes sur un alphabet à deux lettres pour les f-palindromes en utilisant les résultats des trois chapitres précédents. Puis, en annexes, on y trouve d'une part plusieurs calculs qui illustrent les exemples et les différents résultats de ce mémoire. D'autre part, l'ensemble des outils informatiques que j'ai développés afin d'étudier certains problèmes de combinatoire des mots depuis 2006 y sont rassemblés. Ceux-ci ont été modifiés pour la dernière fois en avril 2008. Une version améliorée de ceux-ci sera disponible à la fin de l'été 2008 dans le paquetage associé au logiciel Sage.

CHAPITRE I

PRÉLIMINAIRES

Dans ce chapitre, nous introduisons toutes les définitions et les notations relatives aux mots, aux morphismes et à la théorie des langages et des automates.

1.1 Mots

Comme d'habitude, $\mathbb{N}$, $\mathbb{Z}$ et $\mathbb{Q}$ désignent respectivement l'ensemble des nombres naturels, des entiers relatifs et des nombres rationnels. On écrit $\mathbb{P} = \mathbb{N} \setminus \{0\}$ pour représenter l'ensemble des entiers strictement positifs.

Nous empruntons à (Lothaire, 2002) toute la terminologie de base sur les mots. Dans ce qui suit, Σ est un *alphabet* fini dont les éléments sont appelés des *lettres*. Un *mot* est une suite finie de lettres $w : [1, 2 \dots n] \to \Sigma$, où $n \in \mathbb{P}$. La longueur de w est $|w| = n$ et w_i désigne sa i-ème lettre. L'ensemble des mots de longueur n sur Σ est noté Σ^n. Par convention le mot *vide* est noté ε et sa longueur est 0. Le monoïde libre engendré par Σ est défini par $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$. L'ensemble des mots infinis à droite est noté Σ^ω et nous désignons $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Étant donné un mot $w \in \Sigma^\infty$, un *facteur* f de w est un mot $f \in \Sigma^*$ satisfaisant

$$\exists x \in \Sigma^*, y \in \Sigma^\infty, w = xfy.$$

Si $x = \varepsilon$ (resp. $y = \varepsilon$) alors f est appelé *préfixe* (resp. *suffixe*) de w. Le préfixe (resp. suffixe) de w de longueur d est noté $\mathrm{pref}_d(w)$ (resp. $\mathrm{suff}_d(w)$). L'ensemble de tous les facteurs de w est noté $\mathrm{Fact}(w)$, et ceux de longueur n est $\mathrm{Fact}_n(w) = \mathrm{Fact}(w) \cap \Sigma^n$.

Finalement, Pref(w) désigne l'ensemble de tous les préfixes de w. Le nombre d'occurrences d'un facteur $f \in \Sigma^*$ est $|w|_f$. Si $w = pu$, alors $p^{-1}w = u$ est le mot obtenu en effaçant p et wu^{-1} est le mot obtenu en effaçant u.

Exemple 1.1 Sur $\Sigma = \{a, b\}$, nous avons

$$\begin{aligned}
\text{Fact}(abbab) &= \{\varepsilon, a, b, ab, ba, bb, abb, bab, bba, abba, bbab, abbab\},\\
\text{Pref}(abbab) &= \{\varepsilon, a, ab, abb, abba, abbab\},\\
|abbab|_{ab} &= 2,\\
|abbab|_{ba} &= 1,\\
|abbab|_{aa} &= 0,\\
abbab \cdot (ab)^{-1} &= abb,\\
(ab)^{-1} \cdot abbab &= bab. \quad \diamondsuit
\end{aligned}$$

Une *période* d'un mot w est un entier $p < |w|$ tel que $w[i] = w[i+p]$, pour tout $i < |w|-p$.

Exemple 1.2 L'entier 3 est une période du mot `entente`. $\diamondsuit$

Un résultat très important sur les mots et les périodes est le suivant (Lothaire, 2002, Théorème 8.1.4).

Théorème 1.3 (Fine et Wilf) *Soit w un mot ayant des périodes p et q. Si $|w| \geq p + q - \text{pgcd}(p, q)$, alors* pgcd$(p, q)$ *est aussi une période de w.* ∎

Un mot infini w est *récurrent* s'il satisfait à la condition $u \in \text{Fact}(w) \implies |w|_u = \infty$. Soit $n \in \mathbb{P}$ et w un mot. Le plus petit nombre naturel m (s'il existe) tel que tout facteur de longueur m de w contient tout facteur de longueur n de w est appelé *le n-ème indice de récurrence $R_w(n)$* de w (Morse, Hedlund, 1938). Si $R_w(n)$ existe quel que soit $n \in \mathbb{P}$, alors w est dit *uniformément récurrent.*

Exemple 1.4 Pour le mot fini $w = abbab$ sur $\Sigma = \{a, b\}$, nous avons les indices de récurrence suivants.

n	1	2	3	4	5
$R_w(n)$	3	4	5	5	5

♢

Clairement, un mot périodique infini est uniformément récurrent. En effet, si p est sa période, alors tout facteur de longueur n est contenu dans tout facteur de longueur $n+p-1$.

On dit que r est la *puissance d'exposant rationnel* $\frac{|r|}{|w|}$ *de* w si $r = w^n u$ où u est un préfixe de w et on écrit $r = w^{|r|/|w|}$.

Exemple 1.5 Soit le mot $w = ababcd$ sur l'alphabet $\{a, b, c, d\}$. On a que

$$\begin{aligned} w^0 &= \varepsilon, \\ w^{1/6} &= a, \\ w^1 &= ababcd, \\ w^{3/2} &= ababcd \cdot aba, \\ w^2 &= ababcd \cdot ababcd, \\ w^{10/3} &= ababcd \cdot ababcd \cdot ababcd \cdot ab. \quad \diamondsuit \end{aligned}$$

On dit qu'un mot est *primitif* s'il n'est pas la puissance entière d'un *autre* mot. Deux mots u et v sont *conjugués* s'il existe des mots x, y tels que $u = xy$ et $v = yx$. De plus, le *miroir* de $u = u_1 u_2 \cdots u_n \in \Sigma^n$ est le mot $\widetilde{u} = u_n u_{n-1} \cdots u_1$. Un *palindrome* est un mot p tel que $p = \widetilde{p}$, et l'ensemble de tous les palindromes d'un langage $L \subseteq \Sigma^\infty$ est noté $\mathrm{Pal}(L)$. De plus, pour un mot $w \in \Sigma^*$, l'ensemble de ses facteurs palindromes est $\mathrm{Pal}(w) = \mathrm{Pal}(\Sigma^*) \cap \mathrm{Fact}(w)$ et l'ensemble de ceux de longueur n est $\mathrm{Pal}_n(w) = \mathrm{Pal}(\Sigma^*) \cap \mathrm{Fact}_n(w)$. La *complexité palindromique* d'un mot w est le nombre $|\mathrm{Pal}(w)|$.

Un mot *sturmien* est un mot infini $\mathbf{u}$ tel que $|\mathrm{Fact}_n(\mathbf{u})| = n+1$. Plusieurs caractérisations des mots sturmiens ont été établies dont celle-ci, reliée à leur nombre de palindromes de chaque longueur :

Lemme 1.6 (Droubay, Pirillo, 1999) *Un mot infini* **u** *est sturmien si et seulement si, pour tout* $n \in \mathbb{P}$*,*

$$|\mathrm{Pal}_n(\mathbf{u})| = \begin{cases} 1 & \textit{si } n \textit{ est pair,} \\ 2 & \textit{autrement.} \end{cases}$$

■

Notons qu'il a été démontré dans (Droubay, Justin, Pirillo, 2001) que la complexité palindromique d'un mot fini w est bornée par $|w| + 1$ et que les mots sturmiens atteignent cette borne. Il est alors naturel de se poser la question de l'existence de mots ne l'atteignant pas. Ainsi, (Brlek, Hamel, Nivat, Reutenauer, 2004) ont défini le *défaut palindromique* d'un mot w comme étant

$$D(w) = |w| + 1 - |\mathrm{Pal}(w)|. \tag{1.1}$$

De plus, le défaut d'un mot infini est le supremum des défauts de ses préfixes finis. Dans le cas des mots périodiques, on trouve dans (Brlek, Hamel, Nivat, Reutenauer, 2004) un résultat (algorithmique) optimal permettant de le calculer. Si $D(w) = 0$, c'est-à-dire lorsque w contient un nombre maximal de facteurs palindromes, on dit que w est *plein*[1].

Exemple 1.7 Le défaut du mot infini périodique $(aababbaabbabaa)^\omega$ est 1. Cet exemple est tiré de (Blondin Massé, Brlek, Labbé, 2008).

1.2 Morphismes

Un *morphisme* est une fonction $\varphi : \Sigma^* \to \Sigma^*$ compatible avec la concaténation, c'est-à-dire tel que $\varphi(uv) = \varphi(u)\varphi(v)$ pour tout $u, v \in \Sigma^*$.

Exemple 1.8 Le morphisme identité sur Σ est noté Id_Σ. Certains morphismes sont bien connus sur l'alphabet à deux lettres $\Sigma = \{a, b\}$: par exemple, le morphisme de

[1] Certains auteurs les appellent *riches*, mais nous préférons le mot plein, car cela évoque une limite.

Fibonacci (Séébold, 1991) $\Phi : a \mapsto ab, b \mapsto a$ et le morphisme de Thue-Morse (Morse, Hedlund, 1938) $\mu : a \mapsto ab, b \mapsto ba$ qui sont omniprésents dans ce mémoire. ♢

Exemple 1.9 Sur $\Sigma = \{a, b\}$, on a également le morphisme $E : a \mapsto b, b \mapsto a$ qui échange les lettres. Il sera souvent utilisé dans la suite, et en conséquence le symbole E lui est réservé. ♢

Un morphisme φ est dit *sturmien* si $\varphi(w)$ est un mot sturmien pour tout mot sturmien w.

Théorème 1.10 (Mignosi, Séébold, 1993) *L'ensemble des morphismes sturmiens sur $\Sigma = \{a, b\}$ forme un monoïde et il est engendré par les morphismes E,Φ et $\widetilde{\Phi}$, où Φ est le morphisme de Fibonacci.* ■

Un morphisme φ est *primitif* s'il existe un nombre naturel k tel que, pour tout $\alpha \in \Sigma, \varphi^k(\alpha)$ contient toutes les lettres de Σ. Pour $\alpha \in \Sigma$, on appelle φ-*bloc* (*bloc* s'il n'y a pas de confusion) un facteur de la forme $\varphi(\alpha)$. On définit la fonction $|\varphi| : \Sigma \to \mathbb{N}$ par $|\varphi|(\alpha) = |\varphi(\alpha)|$ pour tout $\alpha \in \Sigma$. Un morphisme est dit *uniforme* quand les blocs sont de longueurs égales, c'est-à-dire, il existe $k \in \mathbb{N}$ tel que $|\varphi| = k$.

Exemple 1.11 Par exemple, le morphisme de Thue-Morse μ est uniforme, mais celui de Fibonacci Φ ne l'est pas. ♢

Un morphisme φ est dit *effaçant* s'il existe $\alpha \in \Sigma$ tel que $\varphi(\alpha) = \varepsilon$. Un morphisme φ est *croissant* si $|\varphi(\alpha)| \geq 2$ pour tout $\alpha \in \Sigma$.

Lemme 1.12 *Soit $\varphi : \Sigma^* \to \Sigma^*$ un morphisme primitif où $|\Sigma| \geq 2$. Il existe un entier k tel que φ^k est un morphisme croissant.* ■

Un morphisme φ est une *involution* si $\varphi^2 = \mathrm{Id}$.

Exemple 1.13 Le morphisme E est la seule involution non triviale sur $\Sigma = \{a, b\}$. ♢

Le *miroir* d'un morphisme φ, noté $\widetilde{\varphi}$, est le morphisme tel que $\widetilde{\varphi}(\alpha) = \widetilde{\varphi(\alpha)}$ pour tout $\alpha \in \Sigma$. Il est facile de vérifier que

$$\widetilde{\varphi(w)} = \widetilde{\varphi}(\widetilde{w}) \quad \text{pour tout} \quad w \in \Sigma^*; \quad \widetilde{\varphi \circ \mu} = \widetilde{\varphi} \circ \widetilde{\mu}. \tag{1.2}$$

Un *antimorphisme* est une fonction $\varphi : \Sigma^* \to \Sigma^*$ telle que $\varphi(uv) = \varphi(v)\varphi(u)$ pour tout $u, v \in \Sigma^*$. Un antimorphisme φ est *involutif* si $\varphi^2 = \mathrm{Id}$. Pour référence ultérieure, on liste les propriétés suivantes faciles à établir.

Lemme 1.14 *Soit $\varphi : \Sigma^* \to \Sigma^*$, une fonction. On a que*

(i) $\widetilde{\ }$ *est un antimorphisme involutif ;*

(ii) φ *est un morphisme si et seulement si $\varphi \circ \widetilde{\ }$ est un antimorphisme ;*

(iii) *si φ est un morphisme et que $\varphi(\alpha)$ est un palindrome pour tout $\alpha \in \Sigma$, alors $\varphi \circ \widetilde{\ } = \widetilde{\ } \circ \varphi$.* ∎

On trouve, caché dans la section 2.3.4 de (Lothaire, 2002), que φ est *conjugué à droite* de φ', noté $\varphi \triangleleft \varphi'$, s'il existe $u \in \Sigma^*$ tel que

$$\varphi(\alpha)u = u\varphi'(\alpha), \quad \text{pour tout } \alpha \in \Sigma, \tag{1.3}$$

ou de façon équivalente que $\varphi(x)u = u\varphi'(x)$, pour tout mot $x \in \Sigma^*$. Il est clair que cette relation n'est pas symétrique de sorte que l'on dit alors que deux morphismes φ et φ' sont *conjugués* si $\varphi \triangleleft \varphi'$ ou $\varphi' \triangleleft \varphi$. La relation de conjugaison des morphismes est une relation d'équivalence.

Exemple 1.15 Soit $\varphi_1 : a \mapsto bbaba, b \mapsto bba$ et $\varphi_2 : a \mapsto abbab, b \mapsto abb$ deux morphismes. Ils sont conjugués, car

$$\begin{aligned} a \cdot \varphi_1(a) = a \cdot bbaba &= abbab \cdot a = \varphi_2(a) \cdot a, \\ a \cdot \varphi_1(b) = a \cdot bba &= abb \cdot a = \varphi_2(a) \cdot a. \quad \diamondsuit \end{aligned}$$

Un *point fixe* d'un morphisme φ est un mot x tel que $\varphi(x) = x$. Bien sûr ε est point fixe de tout morphisme étant donné que $\varphi(\varepsilon) = \varepsilon$.

Un morphisme φ sur Σ^* est dit *cyclique* s'il existe un mot non vide $w \in \Sigma^*$ et si pour tout $\alpha \in \Sigma$ il existe un entier $n_\alpha \in \mathbb{P}$ tel que $\varphi(\alpha) = w^{n_\alpha}$. Dans ce cas, le point fixe infini $w^\omega = www\cdots$ du morphisme φ est périodique.

Lemme 1.16 *Tout point fixe infini de morphisme primitif est uniformément récurrent.*

Démonstration. Si $|\Sigma| = 1$, alors un mot infini sur Σ est trivialement périodique et donc uniformément récurrent. Ainsi, on peut supposer que $|\Sigma| \geq 2$. Soit $\varphi : \Sigma^* \to \Sigma^*$ un morphisme primitif. Selon le Lemme 1.12, il existe $k \in \mathbb{N}$ tel que $\varphi(\alpha)$ contient toutes les lettres de Σ et donc tel que φ^k est croissant. Ainsi, $\lim_{i\to\infty} |\varphi^i(\alpha)| = \infty$ pour tout lettre $\alpha \in \Sigma$. Soit $\mathbf{u} = \varphi(\mathbf{u}) = u_1u_2u_3\cdots$, $u_i \in \Sigma$, un point fixe infini de φ. Comme il existe un préfixe fini de $\mathbf{u}$ qui contient tous les facteurs de longueur n de $\mathbf{u}$, soit

$$\ell_n = \min\{\ell \in \mathbb{N} : f \in \mathrm{Fact}_n(\mathbf{u}) \implies f \in \mathrm{Fact}_n(\varphi^\ell(u_1))\}.$$

Pour tout $\alpha \in \Sigma$, $\varphi^k(\alpha)$ contient au moins une fois la lettre u_1 et donc $\varphi^{\ell_n+k}(\alpha)$ contient tous les facteurs de longueur n de $\mathbf{u}$. En considérant

$$\mathbf{u} = \varphi^{\ell_n+k}(\mathbf{u}) = \varphi^{\ell_n+k}(u_1)\cdot\varphi^{\ell_n+k}(u_2)\cdot\varphi^{\ell_n+k}(u_3)\cdot\ldots,$$

et en posant $m = \max\{|\varphi^{\ell_n+k}(\alpha)| : \alpha \in \Sigma\}$, on conclut que tout facteur de longueur $2m$ de $\mathbf{u}$ contient au moins un facteur de la forme $\varphi^{\ell_n+k}(\alpha)$ qui lui-même contient tout facteur de longueur n de $\mathbf{u}$. Ainsi, $\mathbf{u}$ est uniformément récurrent. ∎

Il existe des mots récurrents non périodiques : le mot de Thue-Morse $\mathbf{t}$ (Morse, Hedlund, 1938) et les mots sturmiens en étant des exemples classiques.

Soit $\mathbf{u}$ un mot infini sur Σ. On dit que $\mathbf{u}$ est *rigide de base* φ si $\mathbf{u}$ est un point fixe d'un morphisme φ sur Σ et si, pour tout morphisme φ' sur Σ tel que $\mathbf{u} = \varphi'(\mathbf{u})$, il existe $n \in \mathbb{N}$ tel que $\varphi' = \varphi^n$ (Mignosi, Séébold, 1993). Nous utilisons cette notion dans une démonstration du Chapitre 3.

1.3 Théorie des langages et des automates

Dans cette section, on introduit les automates finis de même que les langages reconnaissables qui leurs sont associés. Nous utilisons la notation de (Autebert, 1994).

Un *automate fini* $\mathcal{A}$ est un quintuplet : $\langle \Sigma, Q, D, A, \delta \rangle$ où
– Σ est un alphabet, dit alphabet d'entrée,
– Q est un ensemble fini dont les éléments sont appelés les états de l'automate,
– $D \subset Q$ est l'ensemble des états de départ de l'automate,
– $A \subset Q$ est l'ensemble des états d'acceptation de l'automate,
– $\delta \subset Q \times \Sigma \times Q$ est l'ensemble des transitions de l'automate.
Sous-jacent à un tel automate, il y a un multigraphe valué $\langle \Sigma, Q, \delta \rangle$ où Q est l'ensemble des sommets du graphe, Σ l'ensemble des étiquettes des arcs et δ l'ensemble des arcs étiquetés.

Soit $\mathcal{A} = \langle \Sigma, Q, D, A, \delta \rangle$ un automate fini. Considérant l'ensemble fini δ comme un alphabet (dont les lettres sont les arcs du graphe sous-jacent), on dira qu'un mot w de δ^* est un *chemin dans* $\mathcal{A}$ qui mène de l'état q_0 à l'état q_n s'il s'écrit

$$w = (q_0, x_1, q_1)(q_1, x_2, q_2) \dots (q_{n-1}, x_n, q_n).$$

Par convention, on dira que le chemin vide mène de l'état q à lui-même pour tout $q \in Q$. On appelle *trace* l'homomorphisme

$$\begin{array}{rccc} t: & \delta^* & \to & \Sigma^* \\ & (q, x, q') & \mapsto & x. \end{array}$$

Un mot $u \in \Sigma^*$ est *accepté* (ou *reconnu*) par l'automate $\mathcal{A}$ s'il existe un chemin w qui mène d'un l'état $q_d \in D$ à un état $q_a \in A$, et qui est tel que $u = t(w)$. Remarquons que le mot vide est reconnu par l'automate fini $\mathcal{A}$ si et seulement si $D \cap A \neq \varnothing$. On appelle *langage accepté* (ou *langage reconnu*) par $\mathcal{A}$, et on note $L_{\mathcal{A}}$, l'ensemble des mots de Σ^* qui sont acceptés par $\mathcal{A}$. L'automate $\mathcal{A}$ est dit *déterministe* s'il n'y a qu'un état de départ, i.e. $|D| = 1$, et si une seule transition est possible partant d'un état par la

lecture d'une lettre :

$$\forall q \in Q, \forall x \in \Sigma, |\{q' | (q, x, q') \in \delta\}| \leq 1.$$

Il est dit *déterministe complet* si, de plus, une transition est toujours possible :

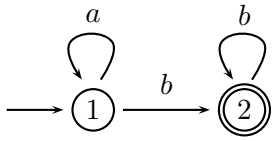
$$\forall q \in Q, \forall x \in \Sigma, |\{q' | (q, x, q') \in \delta\}| = 1.$$

Lorsque l'automate fini $\mathcal{A}$ est déterministe, δ peut être vu (et c'est le point de vue que nous avons adopté dans la suite) comme une application de $Q \times \Sigma$ dans Q définie par :

$$\delta(q, x) = q' \iff (q, x, q') \in \delta.$$

On dit alors souvent que δ est la *fonction de transition* de l'automate. On dit qu'un état p de l'automate $\mathcal{A}$ est *accessible* (resp. *coaccessible*) s'il existe un chemin d'un état initial à p (resp. de p à un état final). On dit que $\mathcal{A}$ est *accessible* (resp. *coaccessible*) si tout état de $\mathcal{A}$ est accessible (resp. coaccessible). On dit que l'automate est *émondé* s'il est à la fois accessible et coaccessible.

Exemple 1.17 Soit $\mathcal{A} = \langle \Sigma, Q, D, A, \delta \rangle$, où $\Sigma = \{a, b\}$, $Q = \{1, 2\}$, $D = \{1\}$, $A = \{2\}$. Sa fonction de transition est donnée par $\delta = \{(1, a, 1), (1, b, 2), (2, b, 2)\}$. La représentation sagittale de l'automate $\mathcal{A}$ est illustrée ci-dessous.



Dans ce dessin, l'état final est représenté par un double cercle et une flèche pointe vers l'état initial. ♢

CHAPITRE II

STRUCTURE DES f-PALINDROMES

Dans ce chapitre, nous introduisons les f-palindromes et nous obtenons plusieurs résultats sur les équations, les paires d'équations et les mots récurrents qui en contiennent.

2.1 Définition et exemples

Soit $f : \Sigma \to \Sigma$ une involution qui s'étend évidemment à un morphisme sur Σ^*. On dit que $w \in \Sigma^*$ est un *f-pseudo-palindrome* (Anne, Zamboni, Zorca, 2005; de Luca, De Luca, 2006; Halava, Harju, Kärki, Zamboni, 2007), ou plus simplement un *f-palindrome*, si $w = f(\widetilde{w})$. L'ensemble des f-palindromes d'un langage $L \subseteq \Sigma^\infty$ est noté $f\text{-Pal}(L)$.

Exemple 2.1 Un palindrome sur Σ est un Id_Σ-palindrome. ♢

Exemple 2.2 Soit $\Sigma = \{a, b\}$ et E l'involution définie à l'Exemple 1.9. Les mots

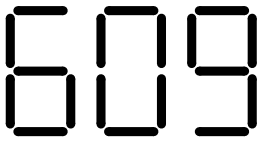
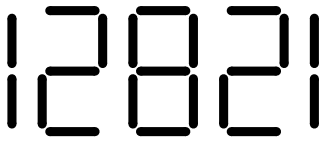
$$\varepsilon, ab, ba, abab, aabb, baba, bbaa, abbaab, bababa$$

sont des E-palindromes. ♢

Exemple 2.3 Soit l'alphabet $\Sigma = \{A, C, G, T\}$ et l'involution $\tau : A \mapsto T, C \mapsto G, T \mapsto A, G \mapsto C$. La présence de τ-palindromes dans les chaînes d'acide nucléique influence leur structure secondaire par des pliages du polynucléotide. Par exemple, un τ-palindrome apparaît (à quelques lettres près) dans le début de la chaîne d'ADN du virus d'immunodéficience humaine (VIH). Une fois plié, ce τ-palindrome est un point d'attache pour

la protéine Tat qui favorise la réplication et la prolifération du virus (Flint et al., 2003, p. 635). ♢

Exemple 2.4 Soit l'alphabet $\Sigma = \{0, 1, 2, 5, 6, 8, 9\}$ et l'involution $\rho : 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 5 \mapsto 5, 6 \mapsto 9, 8 \mapsto 8, 9 \mapsto 6$. Ainsi, 609, 98086 et 1221 sont des ρ-palindromes et correspondent aux nombres digitaux qui sont égaux à leur image par une rotation de 180 degrés (voir la figure ci-bas). ♢

(a)  (b) 

Lemme 2.5 *Soit $f : \Sigma \to \Sigma$ une involution. On a que $f \circ \tilde{\_} = \tilde{\_} \circ f$.*

Démonstration. Ce résultat est une conséquence du Lemme 1.14 (iii). ∎

Dans la suite, on travaille parfois avec deux involutions à la fois comme pour la Proposition 2.14. Lorsqu'il est clair qu'une seule involution f est utilisée, on écrit $\widehat{\_}$ pour représenter l'antimorphisme $f \circ \tilde{\_} = \tilde{\_} \circ f$ afin de simplifier la notation de sorte que w est un f-palindrome si et seulement si $w = \widehat{w}$.

Lemme 2.6 *Soit $f : \Sigma \to \Sigma$ une involution et $w \in \Sigma^*$. Les conditions suivantes sont équivalentes :*

(i) *w est un f-palindrome ;*

(ii) *$\widetilde{w}$ est un f-palindrome ;*

(iii) *$f(w)$ est un f-palindrome ;*

(iv) *$f(\widetilde{w})$ est un f-palindrome.*

Démonstration. (i) $\Longrightarrow$ (ii) Si w est un f-palindrome, alors

$$f(\widetilde{\widetilde{w}}) = f \circ \tilde{\_} \circ \tilde{\_}(w) = \tilde{\_} \circ f \circ \tilde{\_}(w) = \widetilde{f(\widetilde{w})} = \widetilde{w}.$$

(ii) $\Longrightarrow$ (iii) Si $\widetilde{w}$ est un f-palindrome, alors $\widetilde{w} = f(\widetilde{\widetilde{w}}) = f(w)$.

(iii) $\Longrightarrow$ (iv) Si $f(w)$ est un f-palindrome, alors

$$f(\widetilde{f(\widetilde{w})}) = f \circ \widetilde{\ } \circ f \circ \widetilde{\ }(w) = \widetilde{\ } \circ f \circ \widetilde{\ } \circ f(w) = \widetilde{f(\widetilde{f(w)})} = \widetilde{f(w)} = f(\widetilde{w}).$$

(iv) $\Longrightarrow$ (i) Si $f(\widetilde{w})$ est un f-palindrome, alors

$$f(\widetilde{w}) = f(\widetilde{f(\widetilde{w})}) = f \circ \widetilde{\ } \circ f \circ \widetilde{\ }(w) = \widetilde{\ } \circ f \circ f \circ \widetilde{\ }(w) = \widetilde{\ } \circ \widetilde{\ }(w) = w. \quad \blacksquare$$

Sur un alphabet à 2 lettres, la seule involution non triviale est l'échange de lettres. Dans ce cas particulier, c'est-à-dire quand $|\Sigma| = 2$ et $f \neq \mathrm{Id}_\Sigma$, un f-palindrome de Σ^* est appelé un *antipalindrome*. Remarquons que les antipalindromes sont toujours de longueur paire.

Exemple 2.7 Les E-palindromes de l'Exemple 2.2 sont des antipalindromes. $\diamondsuit$

Exemple 2.8 Considérons le morphisme de Thue-Morse μ sur $\{a, b\}$ dont les facteurs ont été étudiés dans (Brlek, 1989). Il est bien connu que $\mu^n(a)$ est un palindrome pour tous les entiers pairs n et il est facile de vérifier que $\mu^n(a)$ est un antipalindrome pour tous les entiers impairs n.

$$\begin{aligned}
\mu^1(a) &= ab \\
\mu^2(a) &= abba \\
\mu^3(a) &= abbabaab \\
\mu^4(a) &= abbabaabbaababba \\
\mu^5(a) &= abbabaabbaababbabaababbaabbabaab
\end{aligned}$$

En général, p est un palindrome si et seulement si $\mu(p)$ est un antipalindrome. $\diamondsuit$

Clairement un f-palindrome contient plusieurs facteurs qui sont aussi des f-palindromes. En particulier, on a l'énoncé suivant.

Lemme 2.9 *Soit $p, x \in \Sigma^*$ et $f : \Sigma \to \Sigma$ une involution. On a*

$$p \text{ est un } f\text{-palindrome} \iff xp\widehat{x} \text{ est un } f\text{-palindrome.} \quad \blacksquare$$

Dans ce cas, on dit que p est un f-palindrome *central* de $xp\widehat{x}$.

2.2 Équations sur les f-palindromes

De (Lothaire, 1983, Proposition 1.3.4) nous rappelons un résultat utile pour la suite : si $w = xy = yz$, alors il existe $u, v \in \Sigma^*$ et $i \in \mathbb{N}$ tels que

$$x = uv, y = (uv)^i u, z = vu; \tag{2.1}$$

Les palindromes satisfaisant ces hypothèses possèdent plusieurs caractérisations qu'on trouve énoncées dans les Lemmes 1 et 2 de (Blondin Massé, Brlek, Labbé, 2008). Voir aussi le Lemme 5 de (Brlek, Hamel, Nivat, Reutenauer, 2004) et le Lemme 2.3 de (de Luca, De Luca, 2006). Nous commençons par en donner une extension aux f-palindromes.

Lemme 2.10 *Supposons que $w = xy = yz$. Soient $u, v \in \Sigma^*$ et $i \in \mathbb{N}$ satisfaisant l'équation* (2.1)*. Soit f une involution sur Σ. Les conditions suivantes sont équivalentes :*

(i) $x = \widehat{z}$;

(ii) *u et v sont des f-palindromes ;*

(iii) *w est un f-palindrome ;*

(iv) *xyz est un f-palindrome.*

De plus, si l'un des conditions ci-dessus est vérifiée, alors

(v) *y est un f-palindrome.*

Démonstration. (i) $\implies$ (ii) Puisque $uv = x = \widehat{z} = \widehat{u}\,\widehat{v}$, on a $u = \widehat{u}$ et $v = \widehat{v}$. (ii) $\implies$ (iii) On a $w = xy = (uv)^{i+1}u$. Alors, $\widehat{w} = \widehat{u}(\widehat{v}\,\widehat{u})^{i+1} = u(vu)^{i+1} = (uv)^{i+1}u = w$.

(iii) $\Longrightarrow$ (iv) On a $xy = w = \widehat{w} = \widehat{z}\,\widehat{y}$. Alors, $x = \widehat{z}$, $y = \widehat{y}$, i.e. y est un f-palindrome et $\widehat{xyz} = \widehat{z}\,\widehat{y}\,\widehat{x} = xyz$. (iv) $\Longrightarrow$ (i) Puisque $|x| = |z|$, on a $x = \widehat{z}$. ∎

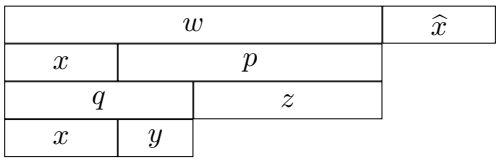
Notons que la condition (v) n'est pas équivalente aux conditions (i)-(iv) du Lemme 2.10. En effet, soit $f = \mathrm{Id}_\Sigma$, $y = aba$, $x = abaab$ and $z = ababa$. L'égalité $xy = yz$ est vérifiée, y est un palindrome mais $x \neq \widetilde{z}$. Cependant, si $|y| \geq |x|$, c'est-à-dire si $i > 0$ dans l'équation (2.1), alors (v) $\Longrightarrow$ (ii). En effet, si y est un f-palindrome, alors $(uv)^i u = y = \widehat{y} = \widehat{u}(\widehat{v}\,\widehat{u})^i = (\widehat{u}\,\widehat{v})^i \widehat{u}$. Donc, $u = \widehat{u}$ et $v = \widehat{v}$ puisque $i > 0$. On a ainsi démontré :

Lemme 2.11 *Soit $w = xy = yz \in \Sigma^*$ tel que $|y| \geq |x|$. Alors les conditions* (i)-(v) *du Lemme 2.10 sont équivalentes.* ∎

Maintenant, nous généralisons un résultat établi dans (Blondin Massé, Brlek, Frosini, Labbé, Rinaldi, 2008).

Proposition 2.12 *Soit f une involution sur Σ. Supposons que $w = xp = qz$ où p et q sont des f-palindromes tels que $|q| > |x|$. Alors, $|x| + |z|$ est une période de w, et $z\widehat{x}$ est un produit de deux f-palindromes.*

Démonstration. Puisque $|q| > |x|$, il existe un mot non vide y tel que $q = xy$ et $p = yz$.



Il s'ensuit que

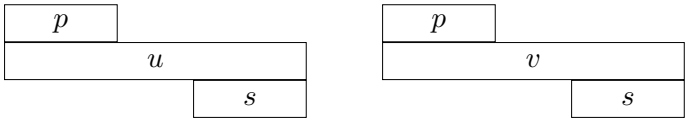
$$w\widehat{x} = qz\widehat{x} = xyz\widehat{x} = xp\widehat{x} = x\widehat{p}\widehat{x} = x\widehat{z}\widehat{y}\widehat{x} = x\widehat{z}\widehat{q} = x\widehat{z}q.$$

Puisque $q \cdot z\widehat{x} = x\widehat{z} \cdot q$, on obtient de l'équation (2.1) que $|z\widehat{x}|$ est une période de $w\widehat{x}$. De plus, en vertu du Lemme 2.10 (i) et (ii), il existe des f-palindromes u, v tels que $z\widehat{x} = uv$. ∎

2.3 Paire d'équations sur les f-palindromes

Le résultat suivant est donné sans démonstration, car elle est immédiate.

Lemme 2.13 *Soit $u, v \in \Sigma^*$ tels que $|u| = |v| = \ell$. Supposons que p est un préfixe et s, un suffixe d'à la fois u et v.*



(i) *Si $|p| + |s| \geq \ell$, alors $u = v$.*

(ii) *Si $|p| + |s| < \ell$, alors il existe $u', v' \in \Sigma^*$ tels que $u = pu's$, $v = pv's$ et $|u'| = |v'|$.*

∎

Proposition 2.14 *Soient $f, g : \Sigma \mapsto \Sigma$ deux involutions. Soient $x, y, p, s, v, w \in \Sigma^*$ où v et w sont des g-palindromes et où $|p| + |s| > 0$, c'est-à-dire p ou s est non vide.*

(i) *Si $xs = pv$ et $f(\widetilde{x})s = pw$, alors x est un f-palindrome.*

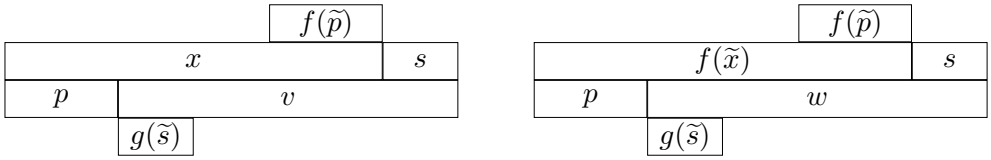
(ii) *Si $xs = py$ et $f(\widetilde{x})s = pg(\widetilde{y})$, alors x est un f-palindrome et y est un g-palindrome.*

(iii) *Si $sx = vp$ et $sf(\widetilde{x}) = wp$, alors x est un f-palindrome.*

Démonstration. En considérant le miroir des deux côtés des équations, on a que (iii) est une conséquence de (i) en vertu des Lemmes 2.6 et 1.14 (iii). En effet, en appliquant l'antimorphisme $\widetilde{\ }$ sur les équations, on obtient $\widetilde{x}\widetilde{s} = \widetilde{p}\widetilde{v}$ et $f(x)\widetilde{s} = \widetilde{f(\widetilde{x})}\widetilde{s} = \widetilde{p}\widetilde{w}$. On montre les deux premiers énoncés en procédant par récurrence sur $|xs|$.

(i) Base. Supposons que $0 < |xs| \leq |p| + |s|$. Dans ce cas, x et $f(\widetilde{x})$ sont préfixes de p, de sorte que $x = f(\widetilde{x})$.

INDUCTION. Supposons que $xs = pv$ et $f(\widetilde{x})s = pw$ implique $x = f(\widetilde{x})$ lorsque $|xs| < \ell$. Maintenant supposons que $|xs| = \ell > |p| + |s|$.



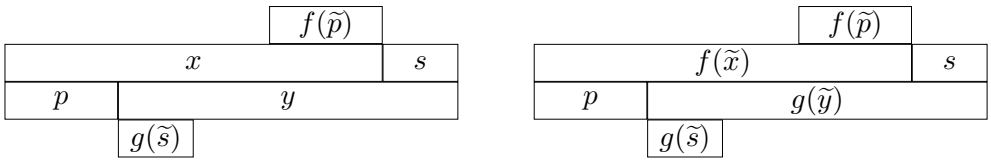
Puisque p est un préfixe de x et $f(\widetilde{x})$, on a que $f(\widetilde{p})$ est un suffixe de $f(\widetilde{x})$ et x. Par le Lemme 2.13, soit $x = f(\widetilde{x})$ ou il existe x' tel que $x = px'f(\widetilde{p})$ et $f(\widetilde{x}) = pf(\widetilde{x'})f(\widetilde{p})$. De plus, comme s est un suffixe de v et w, on a que $g(\widetilde{s})$ est préfixe de v et w. Par le Lemme 2.13, soit $v = w$, de sorte que $x = f(\widetilde{x})$, ou il existe v', w' tels que $v = g(\widetilde{s})v's$ et $w = g(\widetilde{s})w's$. De plus, v' et w' sont des g-palindromes. On obtient

$$px'f(\widetilde{p})s = xs = pv = pg(\widetilde{s})v's \quad \text{et} \quad pf(\widetilde{x'})f(\widetilde{p})s = f(\widetilde{x})s = pw = pg(\widetilde{s})w's,$$

d'où $x'f(\widetilde{p}) = g(\widetilde{s})v'$ et $f(\widetilde{x'})f(\widetilde{p}) = g(\widetilde{s})w'$. D'autre part, $|x'f(\widetilde{p})| = |xs| - |p| - |s| < \ell$ et $|f(\widetilde{p})| + |g(\widetilde{s})| = |p| + |s| > 0$. Donc, selon l'hypothèse de récurrence, on a que $x' = f(\widetilde{x'})$ et que $x = f(\widetilde{x})$.

(ii) BASE. Supposons que $0 < |xs| \leq |p| + |s|$. Dans ce cas, x et $f(\widetilde{x})$ sont préfixes de p, de sorte que $x = f(\widetilde{x})$.

INDUCTION. Supposons que $xs = py$ et $f(\widetilde{x})s = pg(\widetilde{y})$ implique $x = f(\widetilde{x})$ lorsque $|xs| < \ell$. Maintenant supposons que $|xs| = \ell > |p| + |s|$.



Puisque p est un préfixe de x et $f(\widetilde{x})$, on a que $f(\widetilde{p})$ est un suffixe de $f(\widetilde{x})$ et x. Par le Lemme 2.13, soit $x = f(\widetilde{x})$ ou il existe x' tel que $x = px'f(\widetilde{p})$ et $f(\widetilde{x}) = pf(\widetilde{x'})f(\widetilde{p})$. Comme s est un suffixe de y et $g(\widetilde{y})$, on a que $g(\widetilde{s})$ est préfixe de $g(\widetilde{y})$ et y. Par le Lemme 2.13, soit $g(\widetilde{y}) = y$, de sorte que $x = f(\widetilde{x})$, ou il existe y' tel que $y = g(\widetilde{s})y's$ et

$g(\widetilde{y}) = g(\widetilde{s})g(\widetilde{y'})s$. On obtient

$$px'f(\widetilde{p})s = xs = py = pg(\widetilde{s})y's \quad \text{et} \quad pf(\widetilde{x'})f(\widetilde{p})s = f(\widetilde{x})s = pg(\widetilde{y}) = pg(\widetilde{s})g(\widetilde{y'})s,$$

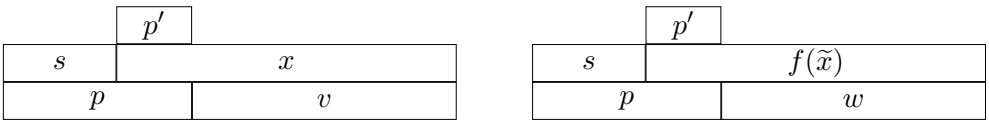
d'où $x'f(\widetilde{p}) = g(\widetilde{s})y'$ et $f(\widetilde{x'})f(\widetilde{p}) = g(\widetilde{s})g(\widetilde{y'})$. D'autre part, $|x'f(\widetilde{p})| = |xs| - |p| - |s| < \ell$ et $|f(\widetilde{p})| + |g(\widetilde{s})| = |p| + |s| > 0$. Donc, selon l'hypothèse de récurrence, on a que $x' = f(\widetilde{x'})$ et que $x = f(\widetilde{x})$. ∎

Corollaire 2.15 *Soient $f, g : \Sigma \mapsto \Sigma$ deux involutions. Soient $x, y, p, s, v, w \in \Sigma^*$ où v et w sont des g-palindromes et $|p| \neq |s|$.*

(i) *Si $sx = pv$ et $sf(\widetilde{x}) = pw$, alors x est un f-palindrome.*

(ii) *Si $sx = py$ et $sf(\widetilde{x}) = pg(\widetilde{y})$, alors x est un f-palindrome et y est un g-palindrome.*

Démonstration. (i) Si $|p| > |s|$, alors il existe un mot p' non vide tel que $x = p'v$ et $f(\widetilde{x}) = p'w$.



Par la Proposition 2.14 (i), on conclut que x est un f-palindrome. Si $|p| < |s|$, alors il existe un mot s' non vide tel que $s'x = v$ et $s'f(\widetilde{x}) = w$ et x est un f-palindrome en vertu de la Proposition 2.14 (iii). (ii) Si $|p| > |s|$, alors il existe un mot p' non vide tel que $x = p'y$ et $f(\widetilde{x}) = p'g(\widetilde{y})$. Le résultat suit de la Proposition 2.14 (ii). Si $|p| < |s|$, alors il existe un mot s' non vide tel que $s'x = y$ et $s'f(\widetilde{x}) = g(\widetilde{y})$, et de nouveau la Proposition 2.14 (ii) permet de conclure. ∎

2.4 Mots récurrents

Lemme 2.16 *Soient* **u** *un mot uniformément récurrent et w un mot non vide. Les conditions suivantes sont équivalentes :*

(i) **u** *contient des puissances arbitrairement longues de w ;*

(ii) *il existe un conjugué w' de w tel que $\mathbf{u} = w'^{\omega}$.*

DÉMONSTRATION. (ii) $\Longrightarrow$ (i) est évident. On montre la réciproque. Soient p un préfixe de $\mathbf{u}$ et $\ell = R_{\mathbf{u}}(|p|)$ le $|p|$-ième indice de récurrence de $\mathbf{u}$. Soit $n \in \mathbb{N}$ tel que $|w^n| > \ell$. Alors p est un facteur de w^n et il existe un conjugué w' de w tel que $p = w'^r$ avec $r \in \mathbb{Q}$. On conclut que $\mathbf{u} = w'^{\omega}$. ∎

Remarquons que si $\mathbf{u}$ n'est pas un mot uniformément récurrent alors le lemme précédent est faux. En effet, considérons l'exemple suivant.

Exemple 2.17 Soit le mot récurrent

$$\mathbf{u} = aba^2ba^3ba^4ba^5ba^6ba^7b\cdots .$$

Il n'est pas périodique malgré qu'il contient des puissances arbitrairement longues de a. ♢

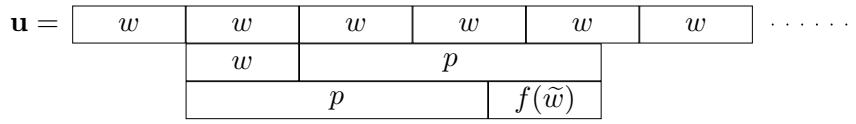
Dans l'esprit du Théorème 4 de (Brlek, Hamel, Nivat, Reutenauer, 2004) qui caractérise les mots périodiques infinis (et bi-infinis) ayant une infinité de palindromes, nous énonçons le lemme suivant pour les mots périodiques infinis à droite. Voir aussi la preuve du Théorème 13 de (Allouche, Baake, Cassaigne, Damanik, 2003). Notre preuve se base sur le Lemme 2.11.

Lemme 2.18 *Soit $\mathbf{u}$ un mot infini périodique. Les conditions suivantes sont équivalentes :*

(i) *$\mathbf{u}$ contient des f-palindromes arbitrairement longs ;*

(ii) *il existe deux f-palindromes $u, v \in \Sigma^*$ tel que $\mathbf{u} = (uv)^{\omega}$.*

DÉMONSTRATION. (ii) $\Longrightarrow$ (i) est évident. En effet, les préfixes de $\mathbf{u}$ de la forme $(uv)^n u$, $n \in \mathbb{N}$, sont des f-palindromes. On montre (i) $\Longrightarrow$ (ii). Soit w tel que $\mathbf{u} = w^{\omega}$.



Comme **u** contient des f-palindromes arbitrairement longs, il existe un f-palindrome p tel que $|p| > |w|$ et p est préfixe d'une puissance de w. Ainsi, wp est aussi préfixe de w^ω. De plus, $f(\widetilde{w})$ est suffixe de $f(\widetilde{p}) = p$, car w est préfixe de p. Alors, $wp = pf(\widetilde{w})$. Le résultat découle du Lemme 2.11. ∎

Remarquons que ce lemme est valide en toute généralité pour les mots bi-infinis, mais comme nous travaillons avec les mots infinis à droite, cette généralisation n'est pas utile.

CHAPITRE III

MORPHISMES DE CLASSE f-$\mathcal{P}$

Dans ce chapitre, nous introduisons les morphismes de classe $\mathcal{P}$ et de classe f-$\mathcal{P}$. Nous obtenons plusieurs résultats qui les caractérisent en exhibant plusieurs exemples. D'un travail précédent, nous rassemblons une section complète sur les morphismes conjugués. Finalement, on s'intéresse aux carrés de morphismes particuliers.

3.1 Morphismes de classe $\mathcal{P}$

Soit $\mathcal{P}$ l'ensemble des morphismes φ tel qu'il existe un palindrome p et pour chaque $\alpha \in \Sigma$ il existe un palindrome q_α tels que $\varphi(\alpha) = pq_\alpha$ (Hof, Knill, Simon, 1995). On dit d'un morphisme $\varphi \in \mathcal{P}$ qu'il est de *classe* $\mathcal{P}$.

Exemple 3.1 Soit l'alphabet $\Sigma = \{a, b\}$. Le morphisme

$$\begin{array}{rccc} \varphi: & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & bb \cdot aba \\ & b & \mapsto & bb \cdot a \end{array}$$

est de classe $\mathcal{P}$. Le morphisme φ possède un seul point fixe qui commence par la lettre b. En calculant le nombre de palindromes de chacunes des itérations successives du morphisme sur la lettre a, on obtient le tableau suivant (calculs à l'Appendice A) :

i	0	1	2	3	4	5	6	7
$\lvert\mathrm{Pal}(\varphi^i(a))\rvert$	2	6	20	72	266	990	3692	13776

Cela semble indiquer que le point fixe de φ contient une infinité de palindromes. $\diamondsuit$

Exemple 3.2 Le morphisme de Thue-Morse $\mu : a \mapsto ab, b \mapsto ba$ n'est pas de classe $\mathcal{P}$, mais son carré l'est :

$$\begin{array}{rccc} \mu^2 : & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & abba \\ & b & \mapsto & baab \end{array}$$

On sait que les deux points fixe de μ^2 possèdent une infinité de palindromes (voir l'Exemple 2.8). $\diamondsuit$

Exemple 3.3 Soit l'alphabet $\Sigma = \{a, b\}$. Le morphisme

$$\begin{array}{rccc} \varphi : & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & abb \\ & b & \mapsto & ba \end{array}$$

n'est pas de classe $\mathcal{P}$. Le morphisme φ possède deux points fixes. En calculant le nombre de palindromes de chacunes des itérations successives du morphisme sur la lettre a et b, on obtient le tableau suivant (calculs à l'Appendice A) :

i	0	1	2	3	4	5	6	7	8
$\lvert\mathrm{Pal}(\varphi^i(a))\rvert$	2	4	8	15	23	23	23	23	23
$\lvert\mathrm{Pal}(\varphi^i(b))\rvert$	2	3	6	13	18	23	23	23	23

Les résultats semblent indiquer que les deux points fixes de φ contiennent le *même* nombre fini de palindromes. On peut vérifier que ce sont les mêmes 23 palindromes. $\diamondsuit$

Dans leur article, Hof, Knill et Simon posent la question suivante :

> "Clearly, we could include into class $\mathcal{P}$ substitutions of the form $s(b) = q_b p$. We do not know whether all palindromic x_s arise from substitutions that are in this extended class $\mathcal{P}$."

que nous reformulons dans les termes de ce mémoire :

Conjecture 3.4 (Hof, Knill, Simon, 1995) *Soit* $\mathbf{u}$ *un point fixe de morphisme primitif. Alors,* $|\mathrm{Pal}(\mathbf{u})| = \infty$ *si et seulement s'il existe un morphisme* φ *tel que* $\varphi(\mathbf{u}) = \mathbf{u}$ *et tel que soit* φ *ou* $\widetilde{\varphi}$ *est de classe* $\mathcal{P}$.

3.2 Morphismes de classe $\mathcal{P}'$

Tel que suggéré par Hof, Knill et Simon, des auteurs ont étendu l'ensemble $\mathcal{P}$ en incluant le miroir des morphismes de classe $\mathcal{P}$ (Allouche, Baake, Cassaigne, Damanik, 2003). La proposition suivante montre que l'ensemble $\mathcal{P}$ doit aussi inclure les morphismes conjugués. Certains ont même déjà modifié la définition de l'ensemble $\mathcal{P}$ en les incluant (Glen, Justin, Widmer, Zamboni, 2008).

Proposition 3.5 (Blondin Massé, 2007) *Il existe un mot infini* $\mathbf{u}$ *point fixe de morphisme primitif et contenant une infinité de palindromes tel que* $\mathbf{u}$ *n'est ni le point fixe d'un morphisme de classe* $\mathcal{P}$ *ni le point fixe d'un morphisme dont le miroir est de classe* $\mathcal{P}$.

Démonstration. Nous construisons l'exemple à partir du morphisme sur $\{a, b\}$ suivant

$$\varphi : \begin{array}{ccc} a & \mapsto & abbab \\ b & \mapsto & abb \end{array} = \begin{pmatrix} a & \mapsto & ab \\ b & \mapsto & b \end{pmatrix} \circ \begin{pmatrix} a & \mapsto & ab \\ b & \mapsto & a \end{pmatrix} \circ \begin{pmatrix} a & \mapsto & ab \\ b & \mapsto & a \end{pmatrix}.$$

Soit $\mathbf{u} = \varphi(\mathbf{u})$ l'unique point fixe de φ. Le morphisme φ est sturmien, car il est le produit de trois morphismes sturmiens. Ainsi, $\mathbf{u}$ est un mot sturmien et contient une infinité de facteurs palindromes en vertu du Lemme 1.6. On vérifie que φ n'est puissance d'aucun autre morphisme. Ainsi, comme φ est sturmien, on a que $\mathbf{u}$ est rigide de base φ (Mignosi, Séébold, 1993). Or, pour tout $k \in \mathbb{P}$, ni φ^k ni $\widetilde{\varphi}^k$ ne sont de classe $\mathcal{P}$. ∎

Par conséquent, la Conjecture 3.4 est fausse et l'ensemble $\mathcal{P}$ devrait définitivement inclure ses morphismes conjugués. Dans ce travail, nous préférons conserver l'ensemble $\mathcal{P}$ tel que défini originellement par Hof, Knill et Simon et nous proposons la définition suivante.

Définition 3.6 *Soit $\mathcal{P}'$ l'ensemble des morphismes ayant un conjugué de classe $\mathcal{P}$. On dit qu'un morphisme φ est de* classe $\mathcal{P}'$ *si $\varphi \in \mathcal{P}'$.*

On a que $\widetilde{\mathcal{P}} \subset \mathcal{P}'$, c'est-à-dire que l'ensemble $\mathcal{P}'$ inclut le miroir des morphismes de classe $\mathcal{P}$. Avec cette définition, la Conjecture 3.4 corrigée s'écrit de la façon suivante.

Conjecture 3.7 *Soit $\mathbf{u}$ un point fixe de morphisme primitif. Alors, $|Pal(\mathbf{u})| = \infty$ si et seulement s'il existe un morphisme φ tel que $\varphi(\mathbf{u}) = \mathbf{u}$ et φ est de classe $\mathcal{P}'$.*

3.3 Morphismes conjugués

Dans cette section, nous rappelons des résultats utiles qui font tous partie d'un travail précédent (Blondin Massé, Brlek, Labbé, 2008).

Lemme 3.8 *Soient φ et φ' deux morphismes non effaçants sur $\Sigma = \{a, b\}$ tels que $\varphi \triangleleft \varphi'$, i.e. $\varphi(\alpha)u = u\varphi'(\alpha)$ pour tout $\alpha \in \Sigma$. Soient $p = |\varphi(a)|$ et $q = |\varphi(b)|$. Si $|u| \geq p + q - \text{pgcd}(p, q)$, alors φ et φ' sont cycliques.*

Démonstration. Selon les équations (2.1), il existe $x, z \in \Sigma^*$, $i \in \mathbb{N}$ tels que $\varphi(a) = xz$, $u = (xz)^i x$ et $\varphi'(a) = zx$. De plus, il existe $w, y \in \Sigma^*$, $j \in \mathbb{N}$ tels que $\varphi(b) = wy$, $u = (wy)^j w$ et $\varphi'(b) = yw$. Alors, $(xz)^i x = u = (wy)^j w$ et donc, $p = |\varphi(a)| = |xz|$ et $q = |\varphi(b)| = |wy|$ sont des périodes de u. Le Théorème 1.3 de Fine et Wilf dit que $\text{pgcd}(p, q)$ est une période de u. Il s'ensuit que $\varphi(a)$ et $\varphi(b)$ sont des puissances du même mot, étant préfixes de u ; $\varphi'(a)$ et $\varphi'(b)$ aussi, étant suffixes de u. ∎

Nous désirons maintenant améliorer un lemme de (Lothaire, 2002, Lemme 2.3.17) sur la composition de morphismes conjugués que nous rappelons d'abord.

Lemme 3.9 (Lothaire, 2002) *Soient φ, φ', μ, μ' des morphismes.*

(i) *Si $\varphi \triangleleft \varphi'$ et $\mu \triangleleft \varphi'$, alors $\varphi \triangleleft \mu$ ou $\mu \triangleleft \varphi$.*

(ii) *Si* $\varphi \lhd \varphi'$ *et* $\varphi \lhd \mu'$, *alors* $\varphi' \lhd \mu'$ *ou* $\mu' \lhd \varphi'$.

(iii) *Si* $\varphi \lhd \varphi'$ *et* $\mu \lhd \mu'$, *alors* $\varphi \circ \mu \lhd \varphi' \circ \mu'$. ∎

Lemme 3.10 *Soient* φ, φ', μ, μ' *des morphismes. On a que*

(i) *si* $\varphi \lhd \varphi'$ *et* $\mu \lhd \mu'$, *alors* $\varphi \circ \mu \lhd \varphi' \circ \mu'$.

(ii) *si* $\varphi \lhd \varphi'$ *et* $\mu \lhd \mu'$, *alors* $\varphi \circ \mu'$ *et* $\varphi' \circ \mu$ *sont conjugués.*

(iii) *si* φ, φ' *sont conjugués et si* μ, μ' *sont conjugués, alors* $\varphi \circ \mu$ *et* $\varphi' \circ \mu'$ *sont conjugués.*

Démonstration. (i) Soient $u, v \in \Sigma^*$ tels que $\varphi(\alpha)u = u\varphi'(\alpha)$ et $\mu(\alpha)v = v\mu'(\alpha)$, pour tout $\alpha \in \Sigma$. On calcule

$$\begin{aligned} \varphi \circ \mu(\alpha) \cdot u\varphi'(v) &= \varphi \circ \mu(\alpha) \cdot \varphi(v)u \\ &= \varphi(\mu(\alpha) \cdot v)u \\ &= u\varphi'(v \cdot \mu'(\alpha)) \\ &= u\varphi'(v) \cdot \varphi' \circ \mu'(\alpha). \end{aligned}$$

(ii) Soient $u, v \in \Sigma^*$ tels que $\varphi(\alpha)u = u\varphi'(\alpha)$ et $\mu(\alpha)v = v\mu'(\alpha)$, pour tout $\alpha \in \Sigma$. Si $|u| \leq |\varphi(v)|$, alors $u^{-1}\varphi(v) = \varphi'(v)u^{-1}$ et on obtient

$$\begin{aligned} u^{-1}\varphi(v) \cdot \varphi \circ \mu'(\alpha) &= u^{-1}\varphi(v \cdot \mu'(\alpha)) \\ &= \varphi'(\mu(\alpha) \cdot v)u^{-1} \\ &= \varphi' \circ \mu(\alpha) \cdot \varphi'(v)u^{-1} \\ &= \varphi' \circ \mu(\alpha) \cdot u^{-1}\varphi(v), \end{aligned}$$

c'est-à-dire que $\varphi' \circ \mu \lhd \varphi \circ \mu'$. Si $|u| \geq |\varphi(v)|$, alors

$$\begin{aligned}
\varphi \circ \mu'(\alpha) \cdot \varphi(v)^{-1}u &= \varphi(v)^{-1}\varphi(v) \cdot \varphi \circ \mu'(\alpha) \cdot uu^{-1} \cdot \varphi(v)^{-1}u \\
&= \varphi(v)^{-1} \cdot \varphi(v\mu'(\alpha))u \cdot u^{-1}\varphi(v)^{-1}u \\
&= \varphi(v)^{-1} \cdot u\varphi'(\mu(\alpha)v) \cdot (\varphi(v)u)^{-1}u \\
&= \varphi(v)^{-1}u \cdot \varphi' \circ \mu(\alpha) \cdot \varphi'(v)(u\varphi'(v))^{-1}u \\
&= \varphi(v)^{-1}u \cdot \varphi' \circ \mu(\alpha),
\end{aligned}$$

c'est-à-dire que $\varphi \circ \mu' \lhd \varphi' \circ \mu$. (iii) Le résultat est une conséquence de (i) et (ii). ∎

La conjugaison des morphismes uniformes peut être définie par une condition équivalente comme le montre le lemme suivant.

Lemme 3.11 *Soient φ et φ' deux morphismes uniformes. Alors $\varphi \lhd \varphi'$ si et seulement s'il existe x et pour tout $\alpha \in \Sigma$ il existe z_α tels que*

$$\varphi(\alpha) = xz_\alpha \text{ et } \varphi'(\alpha) = z_\alpha x. \tag{3.1}$$

Démonstration. ($\Rightarrow$) Supposons qu'il existe $u \in \Sigma^*$ tel que pour tout $\alpha \in \Sigma$, on a $\varphi(\alpha)u = u\varphi'(\alpha)$. D'après les équations (2.1), il existe $x_\alpha, z_\alpha \in \Sigma^*$, $i_\alpha \in \mathbb{N}$ tels que $\varphi(\alpha) = x_\alpha z_\alpha$, $u = (x_\alpha z_\alpha)^{i_\alpha} x_\alpha$ et $\varphi'(\alpha) = z_\alpha x_\alpha$. Comme les morphismes φ et φ' sont uniformes, soit l'entier $k = |\varphi(\alpha)| = |\varphi'(\alpha)|$. Soient deux lettres, disons α et β, telles que pour des entiers i et j, on a $\varphi(\alpha) = xz$, $\varphi'(\alpha) = zx$, $\varphi(\beta) = wy$, $\varphi'(\beta) = yw$ et

$$(xz)^i x = u = (wy)^j w.$$

En considérant les longueurs des mots de la dernière équation, on obtient

$$k \cdot i + |x| = k \cdot j + |w|,$$

où $0 \leq |x| \leq k$ et $0 \leq |w| \leq k$. Si $|x| \neq k$ et $|w| \neq k$, alors on obtient par le théorème de la division euclidienne que $i = j$ et $|x| = |w|$, et ainsi $x = w$ car x et w sont suffixes de u. Si $|x| = k$ et $|w| = k$, alors $x = w$ pour la même raison. Si $|x| = k$ et $|w| \neq k$,

alors $|w| = k \cdot (i - j + 1)$ et donc $|w| = 0$. Ainsi, $j = i + 1$, $x^{i+1} = u = y^j$ et $x = y$. Par conséquent, $\varphi(\alpha) = x = \varphi'(\alpha)$ pour tout $\alpha \in \Sigma$. Finalement, le cas $|x| \neq k$ et $|w| = k$ se règle comme le précédent.

($\Leftarrow$) On a $\varphi(\alpha)x = xz_\alpha x = x\varphi'(\alpha)$ quel que soit α. ∎

Le lemme précédent est faux pour certains morphismes non uniformes. En effet, considérons les morphismes conjugués

$$\varphi_1 : \begin{array}{lcl} a & \mapsto & abaab \\ b & \mapsto & ab \end{array}, \qquad \varphi_2 : \begin{array}{lcl} a & \mapsto & baaba \\ b & \mapsto & ba \end{array},$$

$$\varphi_3 : \begin{array}{lcl} a & \mapsto & aabab \\ b & \mapsto & ab \end{array}, \qquad \varphi_4 : \begin{array}{lcl} a & \mapsto & ababa \\ b & \mapsto & ba \end{array}.$$

Les paires (φ_1, φ_3) et (φ_3, φ_4) satisfont à l'équation (3.1), mais pas la paire (φ_1, φ_4). C'est pourquoi la Proposition 3.13 et plus particulièrement le Corollaire 3.14 sont plus généraux que le Lemme 3 de (Allouche, Baake, Cassaigne, Damanik, 2003) que l'on rappelle.

Lemme 3.12 (Allouche, Baake, Cassaigne, Damanik, 2003) *Let u be a fixed point of a primitive morphism σ on the alphabet A. Suppose that there exists a non-empty word x that is a prefix of $\sigma(a)$ for all $a \in A$ (resp. a suffix of $\sigma(a)$ for all $a \in A$). Write $\sigma(a) = xz_a$ (resp. $\sigma(a) = z_a x$). Let $\sigma_\#$ be the morphism defined by $\sigma_\#(a) = z_a x$ (resp. $\sigma_\#(a) = xz_a$). Then $\sigma_\#$ is primitive, and any fixed point v of a power of $\sigma_\#$ (there always exists at least one such fixed point) has the same factors as u.* ∎

Proposition 3.13 *Soient φ et φ' deux morphismes primitifs et soit $\mathbf{u} = \varphi(\mathbf{u})$, $\mathbf{v} = \varphi'(\mathbf{v})$ deux points fixes respectifs. Si $\varphi \triangleleft \varphi'$, alors* $\mathrm{Fact}(\mathbf{u}) = \mathrm{Fact}(\mathbf{v})$.

Démonstration. Par l'hypothèse et d'après le Lemme 3.10 (i), nous avons que pour tout k, il existe w tel que $\varphi^k(\alpha)w = w\varphi'^k(\alpha)$, et ce pour tout $\alpha \in \Sigma$. Montrons d'abord que $\varphi^k(\alpha)w$ est un facteur de $\mathbf{u}$ et $\mathbf{v}$. Puisque φ est primitif, il existe $x \in \Sigma^*, \mathbf{z} \in \Sigma^\omega$

tels que $\mathbf{u} = x\alpha\mathbf{z}$. On a

$$\mathbf{u} = \varphi^k(x) \cdot \varphi^k(\alpha) \cdot \varphi^k(\mathbf{z}) = \varphi^k(x) \cdot \varphi^k(\alpha) \cdot w\varphi'^k(\mathbf{z}).$$

Par conséquent, $\varphi^k(\alpha)w$ est facteur de $\mathbf{u}$. De façon similaire, il existe $x \in \Sigma^*, \mathbf{z} \in \Sigma^\omega$ tels que $\mathbf{v} = x\alpha\mathbf{z}$. Alors, on a

$$w\mathbf{v} = w\varphi'^k(x) \cdot \varphi'^k(\alpha) \cdot \varphi'^k(\mathbf{z}) = \varphi^k(x)w \cdot \varphi'^k(\alpha) \cdot \varphi'^k(\mathbf{z}).$$

On peut choisir x suffisamment long pour satisfaire $|\varphi^k(x)| \geq |w|$. Ainsi, $w\varphi'^k(\alpha)$ est facteur de $\mathbf{v}$. Finalement, soit $f \in \mathrm{Fact}(\mathbf{u})$. Alors, il existe n tel que $f \in \varphi^n(\mathbf{u}_0)$. Ainsi, $f \in \mathrm{Fact}(\mathbf{v})$. Par le même argument, les facteurs de $\mathbf{v}$ sont facteurs de $\mathbf{u}$. ∎

Certains morphismes, comme $\varphi : a \mapsto ba, b \mapsto ab$, n'ont pas de point fixe. Par contre, tout morphisme possède une puissance ayant un point fixe. C'est pourquoi le résultat suivant, qui est un fait accepté dans (Tan, 2007), est utile.

Corollaire 3.14 *Soient φ et φ' deux morphismes primitifs et supposons qu'il existe des entiers k, l tels que φ^k et φ'^l ont $\mathbf{u} = \varphi^k(\mathbf{u})$ et $\mathbf{v} = \varphi'^l(\mathbf{v})$ comme points fixes. Si φ et φ' sont conjugués, alors* $\mathrm{Fact}(\mathbf{u}) = \mathrm{Fact}(\mathbf{v})$.

Démonstration. Si φ et φ' sont conjugués, alors φ^{kl} et φ'^{kl} sont conjugués par le Lemme 3.10. Puisque $\mathbf{u} = \varphi^{kl}(\mathbf{u})$ et $\mathbf{v} = \varphi'^{kl}(\mathbf{v})$, le résultat est une conséquence de la Proposition 3.13. ∎

En posant $\varphi' = \varphi$ dans la Proposition 3.13, on obtient le corollaire qui suit.

Corollaire 3.15 *Soit φ morphisme primitif, et soit $\mathbf{u} = \varphi(\mathbf{u})$, $\mathbf{v} = \varphi(\mathbf{v})$ deux points fixes. Alors,* $\mathrm{Fact}(\mathbf{u}) = \mathrm{Fact}(\mathbf{v})$.

3.4 La classe $\mathcal{P}'$ est un monoïde

Dans cette section, nous démontrons que la classe $\mathcal{P}'$ est un monoïde, un fait donné sans preuve dans (Glen, Justin, Widmer, Zamboni, 2008).

Lemme 3.16 *Soient $\mu : \alpha \mapsto pq_\alpha$ et $\varphi : \alpha \mapsto uv_\alpha$ deux morphismes de classe $\mathcal{P}$ sur Σ^*, où p, q_α, u, v_α sont des palindromes.*

(i) *Si $u = \varepsilon$, alors $\mu \circ \varphi \in P$.*

(ii) *Si $u \neq \varepsilon$, alors $\widetilde{\mu} \circ \varphi \in P$.*

(iii) *$\mu \circ \varphi$ et $\widetilde{\mu} \circ \varphi$ sont conjugués.*

(iv) *$\mu \circ \varphi \in \mathcal{P}'$.*

Démonstration. On suppose que $u = u_1u_2 \cdots u_n$ et $v_\alpha = v_1v_2 \cdots v_m$. (i) Si $u = \varepsilon$, alors $\mu \circ \varphi(\alpha) = \mu(v_\alpha) = (p) \cdot (q_{v_1}p \cdot q_{v_2}p \cdots q_{v_m})$. (ii) Supposons que $u \neq \varepsilon$. On calcule

$$\widetilde{\mu} \circ \varphi(\alpha) = \widetilde{\mu}(uv_\alpha) \ = (q_{u_1}p \cdot q_{u_2}p \cdots q_{u_n}) \cdot (p \cdot q_{v_1}p \cdot q_{v_2}p \cdots q_{v_m}p).$$

(iii) En effet,

$$\begin{aligned} \mu \circ \varphi(\alpha) \cdot p &= \mu(uv_\alpha) \cdot p \\ &= pq_{u_1}p \cdot q_{u_2}p \cdots q_{u_n}p \cdot q_{v_1}p \cdot q_{v_2}p \cdots q_{v_m}p \\ &= p \cdot \widetilde{\mu}(uv_\alpha) \\ &= p \cdot \widetilde{\mu} \circ \varphi(\alpha). \end{aligned}$$

(iv) De (i), (ii) et (iii), on obtient $\mu \circ \varphi \in \mathcal{P}'$. ∎

L'ensemble des morphismes de classe $\mathcal{P}$ n'est pas un monoïde, car $\mathcal{P} \circ \mathcal{P} \not\subset \mathcal{P}$. Par exemple, considérons le morphisme de Fibonacci : $\Phi \in \mathcal{P}$, mais $\Phi^2 \notin \mathcal{P}$.

Proposition 3.17 *L'ensemble des morphismes de classe $\mathcal{P}'$ est un monoïde.*

Démonstration. Soient $\mu', \varphi' \in \mathcal{P}'$, i.e. il existe un morphisme $\mu \in \mathcal{P}$ conjugué de μ' et un morphisme $\varphi \in \mathcal{P}$ conjugué de φ'. En vertu du Lemme 3.10 (iii), $\varphi' \circ \mu'$ est conjugué à $\varphi \circ \mu$ qui est de classe $\mathcal{P}'$ par le Lemme 3.16. Alors, $\varphi' \circ \mu' \in \mathcal{P}'$. ∎

3.5 Sur la conjecture de Hof, Knill et Simon

Récemment, un article de (Tan, 2007) paru en décembre 2007 présente une démonstration de la Conjecture 3.7 pour un alphabet à deux lettres. Dans ce mémoire, nous étudions l'analogue de la Conjecture 3.7 pour les f-palindromes. Ainsi, nous proposons les définitions utiles suivantes.

Définition 3.18 *On dit qu'un morphisme φ est de* classe f-$\mathcal{P}$ *s'il existe $p \in f\text{-Pal}(\Sigma^*)$ et pour chaque $\alpha \in \Sigma$ il existe $q_\alpha \in f\text{-Pal}(\Sigma^*)$ tels que $\varphi(\alpha) = pq_\alpha$.*

Définition 3.19 *On dit qu'un morphisme φ' est de* classe f-$\mathcal{P}'$*, noté $\varphi' \in f$-$\mathcal{P}'$, s'il existe un morphisme φ conjugué de φ' tel que $\varphi \in f$-$\mathcal{P}$.*

Exemple 3.20 Soit l'alphabet $\Sigma = \{a, b\}$. Le morphisme

$$\begin{array}{rccc} \varphi: & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & ab \cdot baba \\ & b & \mapsto & ab \cdot ab \end{array}$$

est de classe E-$\mathcal{P}$ où E est l'involution sur $\Sigma = \{a, b\}$ définie à l'Exemple 1.9. ♢

La classe f-$\mathcal{P}'$ n'est pas un monoïde pour la composition des morphismes. En effet, on a que le morphisme de Thue-Morse μ est de classe E-$\mathcal{P}'$, mais son carré $\mu^2 \notin E$-$\mathcal{P}'$. Le lemme qui suit donne des équivalences utiles sur la forme des morphismes de classe E-$\mathcal{P}$.

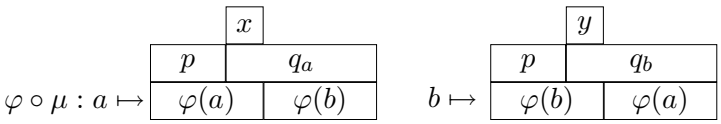
Lemme 3.21 *Soit φ un morphisme uniforme sur $\{a, b\}$ et soit μ le morphisme de Thue-Morse. On a que*

(i) *$\varphi \circ \mu$ est de classe E-$\mathcal{P}$ $\iff$ φ est de la forme $a \mapsto qx$, $b \mapsto qE(\widetilde{x})$ où q est un antipalindrome.*

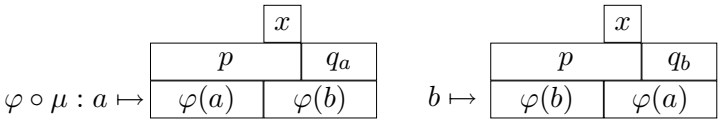
(ii) *φ est de classe E-$\mathcal{P}$ $\iff$ $\varphi \circ \mu$ est de la forme $a \mapsto qx$, $b \mapsto qE(\widetilde{x})$ où q est un antipalindrome.*

DÉMONSTRATION. Dans cette preuve, on utilise le symbole $\widehat{\ }$ pour représenter l'antimorphisme $E \circ \widetilde{\ }$ et on pose $k = |\varphi(a)| = |\varphi(b)|$.

(i) ($\Rightarrow$) Si $\varphi \circ \mu$ est de classe E-$\mathcal{P}$, alors il existe des antipalindromes p, q_a et q_b tels que $\varphi(ab) = pq_a$ et $\varphi(ba) = pq_b$. Supposons que $|p| \leq k$. Soient $x, y \in \Sigma^*$ tels que $\varphi(a) = px$ et $\varphi(b) = py$.



On obtient l'égalité $pxpy = pq_a$, d'où $x = \widehat{y}$ car $|x| = |y|$. Supposons que $|p| > k$. Soit $x \in \Sigma^*$ tel que $\varphi(a) = xq_b$ et $\varphi(b) = xq_a$.

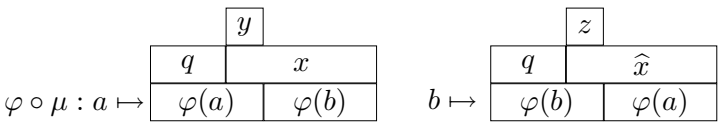


On obtient l'égalité $xq_bx = p = xq_ax$ de sorte que $q_a = q_b$ et $x = \widehat{x}$, car p est un antipalindrome.

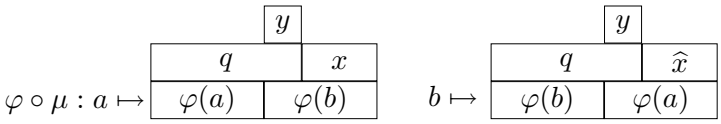
($\Leftarrow$) Si φ est de la forme $\varphi(a) = qx$, $\varphi(b) = q\widehat{x}$ où q est un antipalindrome, alors $\varphi \circ \mu(a) = q \cdot xq\widehat{x}$ et $\varphi \circ \mu(b) = q \cdot \widehat{x}qx$ de sorte que $\varphi \circ \mu$ est de classe E-$\mathcal{P}$.

(ii) ($\Rightarrow$) Si φ est de classe E-$\mathcal{P}$, alors il existe des antipalindromes p, q_a et q_b tels que $\varphi(a) = pq_a$ et $\varphi(b) = pq_b$. Donc, $\varphi \circ \mu(a) = p \cdot q_apq_b$ et $\varphi \circ \mu(b) = p \cdot q_bpq_a$.

($\Leftarrow$) Supposons que $\varphi \circ \mu$ est de la forme $a \mapsto qx$, $b \mapsto q\widehat{x}$ où q est un antipalindrome. On suppose d'abord que $|q| \leq k$. Soient $y, z \in \Sigma^*$ tels que $\varphi(a) = qy$ et $\varphi(b) = qz$.



On obtient les égalités $x = yqz$ et $\widehat{x} = zqy$. Donc, $yqz = x = \widehat{y}q\widehat{z}$ de sorte que $y = \widehat{y}$ et $z = \widehat{z}$. On conclut que φ est de classe E-$\mathcal{P}$. Supposons maintenant que $|q| > k$. Soit $y \in \Sigma^*$ tel que $\varphi(a) = y\widehat{x}$ et $\varphi(b) = yx$.



On obtient l'égalité $y\widehat{x}y = q = yxy$ de sorte que $\widehat{x} = x$ et $\widehat{y} = y$, car q est un antipalindrome. On conclut que φ est de classe E-$\mathcal{P}$. ∎

Lemme 3.22 *Supposons que $\varphi : \Sigma^* \to \Sigma^*$ est un morphisme primitif sur $\Sigma = \{a, b\}$ tel que $\mathbf{u} = \varphi(\mathbf{u})$ est un point fixe infini. Nous avons*

(i) (Hof, Knill, Simon, 1995) *si $\varphi \in \mathcal{P}$, alors $|\mathrm{Pal}(\mathbf{u})| = \infty$;*

(ii) *si $\varphi \in \mathcal{P}'$, alors $|\mathrm{Pal}(\mathbf{u})| = \infty$;*

(iii) *si φ est de la forme $\varphi(a) = qx$, $\varphi(b) = qE(\widetilde{x})$ où q est un antipalindrome, alors $|E\text{-Pal}(\mathbf{u})| = \infty$.*

Démonstration. Tout d'abord, comme φ est primitif, on a que $|\varphi(a)| \geq 1$, $|\varphi(b)| \geq 1$, $|\varphi(a)| > 1$ ou $|\varphi(b)| > 1$. (i) Pour la suite, on dit qu'un mot est *trivial* s'il est la puissance d'une lettre. Il suffit de montrer que si $\mathbf{u}$ contient un palindrome r non trivial alors il en contient un autre plus long.

Base. La primitivité du morphisme φ implique que chaque lettre apparaît une infinité de fois dans $\mathbf{u}$. Ainsi, $\mathbf{u}$ contient un palindrome r non trivial.

Induction. En supposant que le morphisme φ s'écrit $a \mapsto pq_a, b \mapsto pq_b$, on a que $s = \varphi(r)p \in \mathrm{Fact}(\mathbf{u})$ est un palindrome. Par la primitivité, on a aussi que l'image d'un mot non trivial est non triviale. Ainsi, s est non trivial et $|s| > |r|$.

(ii) C'est une conséquence de (i) et du Corollaire 3.14.

(iii) Il suffit de montrer par récurrence que si $\mathbf{u}$ contient un antipalindrome, alors il en contient un autre plus long.

Base. La primitivité du morphisme φ implique que $\mathbf{u}$ contient l'antipalindrome ab ou ba.

Induction. Si $\mathbf{u}$ contient l'antipalindrome r, alors $s = \varphi(r)q \in \mathrm{Fact}(\mathbf{u})$ est aussi un antipalindrome. Ainsi, comme tout antipalindrome non vide contient des a et des b et par la primitivité de φ, on a $|s| > |r|$. ∎

Il est à remarquer que le Lemme 3.22 (i) et (ii) ne se généralise pas pour les f-palindromes. En effet, voici deux contre-exemples. Les détails du calcul du nombre de leurs f-palindromes se trouvent à l'Appendice A.

Exemple 3.23 Soit le morphisme

$$\begin{aligned} \varphi: \quad a &\mapsto ab \cdot ab \\ b &\mapsto ab \cdot ba \end{aligned}$$

sur $\{a, b\}$ et $\mathbf{u} = \varphi(\mathbf{u})$ son point fixe. On a $\varphi \in E\text{-}\mathcal{P}$, mais $|E\text{-Pal}(\mathbf{u})| = 72$. ♢

Exemple 3.24 Soit le morphisme

$$\begin{aligned} \varphi: \quad a &\mapsto acb \cdot cc \\ b &\mapsto acb \cdot ab \\ c &\mapsto acb \cdot ba \end{aligned}$$

sur $\{a, b, c\}$ et $\mathbf{u} = \varphi(\mathbf{u})$ son point fixe. Soit l'involution $f : a \mapsto b, b \mapsto a, c \mapsto c$. On a $\varphi \in f\text{-}\mathcal{P}$, mais $|f\text{-Pal}(\mathbf{u})| = 50$. ♢

3.6 Le carré de certains morphismes

Les carrés de morphismes sur l'alphabet binaire possèdent de remarquables propriétés. On en regroupe certaines dans le lemme suivant.

Lemme 3.25 *Soient l'alphabet $\Sigma = \{a, b\}$, $\varphi : \Sigma^* \mapsto \Sigma^*$ un morphisme et $f : \Sigma \mapsto \Sigma$ une involution. Si $\varphi(a)$ est un antipalindrome et $f(\widetilde{\varphi(a)}) = \varphi(b)$, alors*

(i) $\widetilde{\varphi} = f \circ \varphi \circ E$,

(ii) $\widetilde{\varphi} = E \circ \varphi$,

(iii) $\varphi^2 = f(\widetilde{\varphi^2})$, *i.e.* $\varphi^2(a)$ *et* $\varphi^2(b)$ *sont des f-palindromes,*

(iv) φ^2 *est de classe f-$\mathcal{P}$.*

DÉMONSTRATION. (i) Selon l'hypothèse, $\widetilde{\varphi}(b) = \widetilde{\varphi(b)} = f \circ \varphi(a) = f \circ \varphi \circ E(b)$. De plus,

$$\widetilde{\varphi}(a) = f \circ f \circ \widetilde{\varphi}(a) = f \circ \varphi(b) = f \circ \varphi \circ E(a).$$

(ii) Puisque $\varphi(a)$ est un antipalindrome, on a $\widetilde{\varphi}(a) = \widetilde{\varphi(a)} = E(\varphi(a)) = E \circ \varphi(a)$. De plus,

$$\widetilde{\varphi}(b) = f \circ f \circ \widetilde{\varphi}(b) = f \circ \varphi(a) = f \circ E \circ E \circ \varphi(a) = f \circ E \circ \widetilde{\varphi}(a) = f \circ E \circ f \circ \varphi(b).$$

Comme $|\Sigma| = 2$, soit $f = E$ ou $f = \mathrm{Id}_\Sigma$. Alors, on conclut $\widetilde{\varphi}(b) = E \circ \varphi(b)$.
(iii) Soit $\alpha \in \Sigma$. On a

$$\begin{aligned} \varphi^2(\alpha) &= f^2 \circ \varphi \circ E^2 \circ \varphi(\alpha) \\ &= f \circ [f \circ \varphi \circ E] \circ [E \circ \varphi](\alpha) = f \circ \widetilde{\varphi} \circ \widetilde{\varphi}(\alpha) = f(\widetilde{\varphi^2(\alpha)}). \end{aligned}$$

(iv) Suit de (iii). ∎

On a alors immédiatement le corollaire suivant, illustré par un exemple.

Corollaire 3.26 *Soient $\Sigma = \{a, b\}$ et $\varphi : \Sigma^* \mapsto \Sigma^*$ un morphisme. Si $\varphi(a)$ est un antipalindrome et $\widetilde{\varphi(a)} = \varphi(b)$, alors $\varphi^2(a)$ et $\varphi^2(b)$ sont des palindromes.* ∎

Exemple 3.27 Soit le morphisme

$$\begin{aligned} \varphi : \quad a &\mapsto abbaab \\ b &\mapsto baabba \end{aligned}$$

sur $\{a, b\}$. On observe que

$$\begin{aligned} \varphi^2(a) &= abbaabbaabbabaabbaabbaababbaabbaabba, \\ \varphi^2(b) &= baabbaabbaababbaabbaabbabaabbaabbaab \end{aligned}$$

sont des palindromes. ♢

CHAPITRE IV

LE GRAPHE DES CHEVAUCHEMENTS

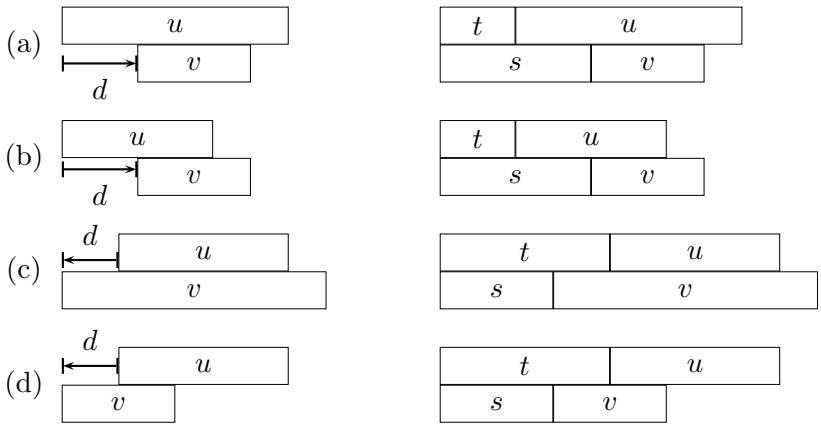
Dans ce chapitre, on définit les chevauchements de mots et on introduit un graphe de chevauchements. À partir de ce graphe, on construit des automates afin d'étudier les points fixes de morphismes contenant une infinité de f-palindromes.

4.1 Chevauchements

Soient $u, v \in \Sigma^*$. On dit que u *chevauche* v *avec un décalage* d si

$$-|v| < d < |u|$$

et s'il existe $s, t \in \Sigma^*$ tel que $d = |s| - |t|$ et tu et sv sont comparables pour l'ordre préfixe. Dans les diagrammes ci-dessous, le mot u chevauche v avec un décalage $d = |s| - |t|$ positif en (a) et (b) et négatif en (c) et (d).



Exemple 4.1 Le mot `cheval` chevauche le mot `chevalet` avec un décalage 0. Le mot `chevalet` chevauche le mot `valet` avec un décalage 3. Le mot `alphabet` chevauche le mot `beta` avec un décalage 5. Le mot `beta` chevauche le mot `alphabet` avec un décalage -5. Le mot `alphabet` chevauche le mot `hab` avec un décalage 3. ♢

Nous définissons la relation

$$\mathcal{R} = \{(u, v, d) \in \Sigma^* \times \Sigma^* \times \mathbb{Z} |\ u \text{ chevauche } v \text{ avec un décalage } d\ \}.$$

Lemme 4.2 *Soit $u, v \in \Sigma^*$ et $d \in \mathbb{Z}$. Les conditions suivantes sont équivalentes :*

(i) $(u, v, d) \in \mathcal{R}$,

(ii) $(\widetilde{v}, \widetilde{u}, |v| - |u| + d) \in \mathcal{R}$,

(iii) $(v, u, -d) \in \mathcal{R}$,

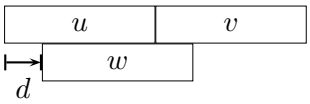
(iv) $(\widetilde{u}, \widetilde{v}, |u| - |v| - d) \in \mathcal{R}$. ■

Lemme 4.3 *Soient $u, v, w \in \Sigma^*$ et $d \in \mathbb{Z}$ tels que $(uv, w, d) \in \mathcal{R}$. Alors :*

(i) *si $d < |u|$, $(u, w, d) \in \mathcal{R}$;*

(ii) *si $-|w| < d - |u|$, $(v, w, d - |u|) \in \mathcal{R}$.*

Démonstration. Considérons le diagramme suivant.



Comme $(uv, w, d) \in \mathcal{R}$, il existe $s, t \in \Sigma^*$ tel que $d = |s| - |t|$, tuv et sw sont comparables pour l'ordre préfixe et $-|w| < d < |uv|$.

(i) On a $-|w| < d < |u|$ et tu et sw sont comparables pour l'ordre préfixe, de sorte que $(u, w, d) \in \mathcal{R}$.

(ii) On a $-|w| < d - |u| < |v|$, $tu \cdot v$ et sw sont comparables pour l'ordre préfixe et $d - |u| = |s| - |tu|$. Alors, $(v, w, d - |u|) \in \mathcal{R}$. ■

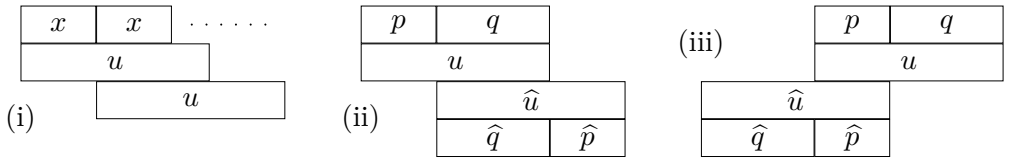
Lemme 4.4 *Soient $u = pq \in \Sigma^*$, $d \in \mathbb{Z}$ et $f : \Sigma \to \Sigma$ une involution. On a :*

(i) *si $(u, u, d) \in \mathcal{R}$, alors $|d|$ est une période de u,*

(ii) *si $(u, \widehat{u}, |p|) \in \mathcal{R}$, alors q est un f-palindrome,*

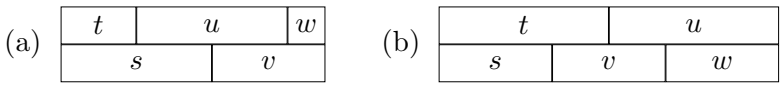
(iii) *si $(u, \widehat{u}, -|q|) \in \mathcal{R}$, alors p est un f-palindrome.*

Démonstration. Les cas sont illustrés dans la figure ci-dessous.



(i) Le résultat suit de l'équation (2.1). (ii) On a $pq\widehat{p} = p\widehat{q}\widehat{p}$, de sorte que $q = \widehat{q}$ est un f-palindrome. (iii) Le résultat suit de (ii) et du Lemma 4.2. ∎

Soit $(u, v, d) \in \mathcal{R}$. Par définition, il existe $s, t \in \Sigma^*$ tels que $d = |s| - |t|$ et tu est préfixe de sv ou vice versa. Alors, il existe un mot w tel que soit $tuw = sv$ ou $tu = svw$.



On définit les deux fonctions d'extensions suivantes :

$$\begin{array}{rccc} \mathrm{Ext}: & \mathcal{R} & \to & \Sigma^* \\ & (u,v,d) & \mapsto & w \end{array}$$

$$\begin{array}{rccc} \mathrm{ext}: & \mathcal{R} & \to & \mathbb{Z} \\ & (u,v,d) & \mapsto & |u| - |v| - d. \end{array}$$

Si $|u| - |v| - d \geq 0$, alors $\mathrm{ext}(u, v, d) = |w|$ et cela correspond au dessin (b) ci-haut. Si $|u| - |v| - d < 0$, alors $\mathrm{ext}(u, v, d) = -|w|$ et cela correspond au dessin (a) ci-haut. On dit que $r, r' \in \mathcal{R}$ *ont la même extension* si $\mathrm{Ext}(r) = \mathrm{Ext}(r')$ et $\mathrm{ext}(r) = \mathrm{ext}(r')$.

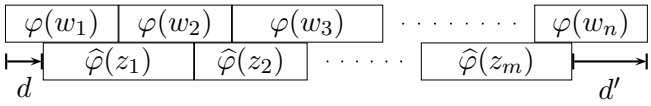
Exemple 4.5 On a les valeurs suivantes pour trois chevauchements distincts.

$$\begin{array}{rcll} \mathcal{R} & & \text{Ext} & \text{ext} \\ (\texttt{cheval}, \texttt{chevalet}, 0) & \mapsto & \texttt{et} & -2, \\ (\texttt{chevalet}, \texttt{cheval}, 0) & \mapsto & \texttt{et} & 2, \\ (\texttt{lebeaumuret}, \texttt{mur}, 6) & \mapsto & \texttt{et} & 2. \end{array}$$

Donc, $(\texttt{chevalet}, \texttt{cheval}, 0)$ et $(\texttt{lebeaumuret}, \texttt{mur}, 6)$ ont la même extension. ♢

4.2 Définition de l'automate

Soit $f : \Sigma \to \Sigma$ une involution. Soient $d \in \mathbb{Z}$, $\ell : \Sigma \to \mathbb{N}$ une fonction et $\varphi : \Sigma^* \to \Sigma^*$ un morphisme tel que $|\varphi| = \ell$ (notation définie à la section 1.2). Soit $w = w_1 w_2 w_3 \cdots w_n$ et $z = z_1 z_2 \cdots z_m \in \Sigma^*$, $w_i, z_j \in \Sigma$, tels que $(\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R}$.

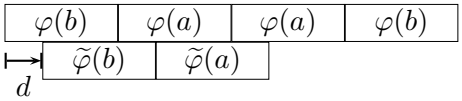


On dit que $(w, z) \in \Sigma^* \times \Sigma^*$ est *équilibré par rapport à d et ℓ* si

$$-\ell(z_1) < d < \ell(w_1) \quad \text{et} \quad -\ell(z_m) < d' < \ell(w_n),$$

où $d' = \text{ext}(\varphi(w), \widehat{\varphi}(z), d)$.

Exemple 4.6 Soient $\Sigma = \{a, b\}$ et $\ell : \Sigma \to \mathbb{N}$ tels que $\ell(a) = \ell(b) = k$. Soit $d \in \mathbb{Z}$ tel que $0 \leq d \leq k$. Supposons que $(\varphi(baab), \widetilde{\varphi}(ba), d) \in \mathcal{R}$.



On a que $(baab, ba)$ n'est pas équilibré, car un bloc dépasse à droite, et plus formellement parce que $\text{ext}(\varphi(baab), \widetilde{\varphi}(ba), d) > \ell(b)$. ♢

On définit l'ensemble des états

$$Q_{d,\ell} = \{(w,z) \in \Sigma^* \times \Sigma^* | (w,z) \text{ est équilibré}\},$$

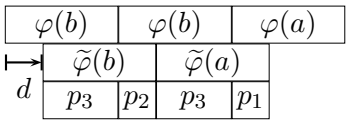
et on dit que $(w,z) \in Q_{d,\ell}$ est *de classe* f-$\mathcal{P}'$ si pour tout morphisme φ tel que $|\varphi| = \ell$ on a

$$(\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R} \implies \varphi \text{ est de classe } f\text{-}\mathcal{P}'.$$

Les deux prochains lemmes illustrent des éléments de $Q_{d,\ell}$ qui sont de classe f-$\mathcal{P}'$ (des exemples sont à l'Appendice A).

Lemme 4.7 *Soient $\Sigma = \{a,b\}$ et $\ell : \Sigma \to \mathbb{N}$ tels que $\ell(a) = \ell(b) = k$. Soit $d \in \mathbb{Z}$ tel que $0 \leq d \leq k$. Alors, $(bba, ba) \in Q_{d,\ell}$ est de classe $\mathcal{P}'$.*

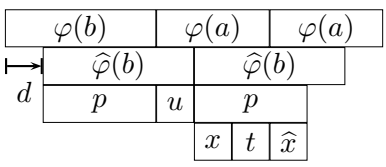
Démonstration. Soit $\varphi : \Sigma^* \to \Sigma^*$ un morphisme tel que $|\varphi| = \ell$ et supposons que $(\varphi(bba), \widetilde{\varphi}(ba), d) \in \mathcal{R}$.



En vertu des Lemmes 4.3 et 4.4, il existe des palindromes $p_1, p_2, p_3 \in \Sigma^*$ tels que $\varphi(a) = p_1p_3$ et $\varphi(b) = p_2p_3$ de sorte que φ est de classe $\mathcal{P}'$. Alors, $(bba, ba) \in Q_{d,\ell}$ est de classe $\mathcal{P}'$. ∎

Lemme 4.8 *Soient $\Sigma = \{a,b\}$ et $l : \Sigma \to \mathbb{N}$ tels que $\ell(a) < \ell(b)$. Soit $d \in \mathbb{Z}$ tel que et $0 \leq d \leq \ell(b)$ et $\ell(b) + d < 2 \cdot \ell(a)$. Soit $f : \Sigma \to \Sigma$ une involution. Alors, $(baa, bb) \in Q_{d,\ell}$ est de classe f-$\mathcal{P}'$.*

Démonstration. Soit $\varphi : \Sigma^* \to \Sigma^*$ un morphisme tel que $|\varphi| = \ell$ et supposons que $(\varphi(baa), \widehat{\varphi}(bb), d) \in \mathcal{R}$.



En vertu du Lemme 4.4, il existe un f-palindrome $p \in \Sigma^*$ de longueur $\ell(b) - d$ et $u \in \Sigma^*$ tels que $\widehat{\varphi}(b) = pu$. Soit $t \in \Sigma^*$ le plus long f-palindrome central de p à l'intérieur de $\varphi(a)$. On a

$$|t| = \left((\ell(b) + \ell(a)) - \left(\frac{\ell(b) + d}{2} + \ell(b)\right)\right) \cdot 2 = 2\ell(a) - \ell(b) - d > 0.$$

Mais $\text{ext}(\varphi(baa), \widehat{\varphi}(bb), d) = 2\ell(a) - \ell(b) - d$, alors $t = \text{Ext}(\varphi(baa), \widehat{\varphi}(bb), d)$. Soit $x \in \Sigma^*$ tel que $p = xt\widehat{x}$. On a $x \neq \varepsilon$ parce que $\ell(a) \neq \ell(b)$. Ainsi, $\varphi(a) = uxt = \widehat{x}ut$ de sorte que $ux = \widehat{x}u$. Par le Lemme 2.10, u et ux sont des f-palindromes. Alors, $\varphi(a) = uxt$, $\varphi(b) = \widehat{u}xt\widehat{x} = uxt\widehat{x}$, et φ est un conjugué de $\varphi' : a \mapsto tux, b \mapsto t\widehat{x}ux$ qui est de classe f-$\mathcal{P}$. Alors, $(baa, bb) \in Q_{d,\ell}$ est de classe f-$\mathcal{P}'$. ∎

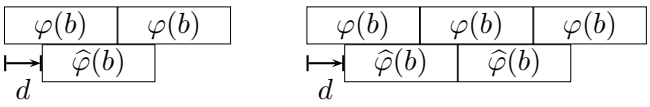
On définit une relation d'équivalence sur $Q_{d,\ell}$. Soient $(x, y), (w, z) \in Q_{d,\ell}$. On écrit $(x, y) \equiv (w, z)$ si pour tout morphisme φ tel que $|\varphi| = \ell$ on a :

$$(\varphi(x), \widehat{\varphi}(y), d) \in \mathcal{R} \iff (\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R},$$

et si $(\varphi(x), \widehat{\varphi}(y), d) \in \mathcal{R}$ et $(\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R}$ ont la même extension. Le prochain lemme illustre deux éléments de $Q_{d,\ell}$ qui sont équivalents.

Lemme 4.9 *Soient $\Sigma = \{a, b\}$ et $\ell : \Sigma \to \mathbb{N}$ tels que $\ell(a) = \ell(b) = k$. Soit $d \in \mathbb{Z}$ tel que $0 \leq d \leq k$. Soit $f : \Sigma \to \Sigma$ une involution. Alors, $(bb, b) \equiv (bbb, bb)$.*

Démonstration. Soit $\varphi : \Sigma^* \to \Sigma^*$ un morphisme tel que $|\varphi| = \ell$. Soient $r_1 = (\varphi(bb), \widehat{\varphi}(b), d)$ et $r_2 = (\varphi(bbb), \widehat{\varphi}(bb), d)$.



On a

$$\begin{aligned} r_1 \in \mathcal{R} &\iff (\varphi(b), \widehat{\varphi}(b), d) \in \mathcal{R} \text{ et } (\varphi(b), \widehat{\varphi}(b), d - k) \in \mathcal{R} \\ &\iff r_2 \in \mathcal{R}. \end{aligned}$$

De plus,

$$\mathrm{Ext}(r_1) = \mathrm{suff}_{k-d}(\varphi(b)) = \mathrm{Ext}(r_2)$$

et

$$\mathrm{ext}(r_1) = k - d = \mathrm{ext}(r_2),$$

de sorte que r_1 et r_2 ont la même extension. ∎

Notons $\dot{\Sigma} = \Sigma \cup \varepsilon$. Finalement, la fonction de transition $\sigma : Q_{d,\ell} \times \left(\dot{\Sigma} \times \dot{\Sigma}\right) \to Q_{d,\ell}$ est définie par

$$\sigma : (w,z), \alpha, \beta \mapsto \begin{cases} (\alpha, \beta) & \text{si } (w,z) = (\varepsilon, \varepsilon), \\ (w,z) & \text{sinon et si } (w,z) \text{ est de classe } f\text{-}\mathcal{P}', \\ (w\alpha, z) & \text{sinon et si } (w\alpha, z) \text{ est équilibré}, \\ (w, z\beta) & \text{autrement.} \end{cases}$$

Proposition 4.10 *Soit $\alpha \in \Sigma$ et $I = \{(\varepsilon, \varepsilon)\} \subset Q_{d,\ell}$. Soit $T \subset Q_{d,\ell}$. Les conditions suivantes sont satisfaites :*

(i) *l'ensemble $Q_{d,\ell}/_{\equiv}$ est fini ;*

(ii) *la relation $\sigma/_{\equiv} : Q_{d,\ell}/_{\equiv} \times \dot{\Sigma} \times \dot{\Sigma} \to Q_{d,\ell}/_{\equiv}$ définie par $[q]_{\equiv}, \alpha, \beta \mapsto [\sigma(q, \alpha, \beta)]_{\equiv}$ est une fonction ;*

(iii) *$\langle \dot{\Sigma} \times \dot{\Sigma}, Q_{d,\ell}/_{\equiv}, I/_{\equiv}, T/_{\equiv}, \sigma/_{\equiv} \rangle$ est un automate fini déterministe complet.*

Démonstration. (i) Soit $k = \max\{\ell(\alpha) | \alpha \in \Sigma\}$ et soit φ tel que $|\varphi| = \ell$. On a que

$$\begin{aligned} |\{(\varphi(\alpha), \widehat{\varphi}(\beta), d) \in \mathcal{R} | \alpha, \beta \in \Sigma, d \in \mathbb{Z}\}| &\leq |\Sigma| \cdot |\Sigma| \cdot |\{d \in \mathbb{Z}| - k \leq d \leq k\}| \\ &= (2k+1) \cdot |\Sigma|^2. \end{aligned}$$

De plus,

$$\begin{aligned} |\{Ext(r) &: r = (\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R}, (w,z) \in Q_{d,\ell}\}| \\ &\leq |\{s \in \Sigma^* | \varphi(\alpha) = ps, \alpha \in \Sigma\}| + |\{s \in \Sigma^* | \widehat{\varphi}(\alpha) = ps, \alpha \in \Sigma\}| \\ &= 2 \cdot |\{s \in \Sigma^* | \varphi(\alpha) = ps, \alpha \in \Sigma\}| \\ &\leq 2 \cdot |\Sigma| \cdot (k+1) \end{aligned}$$

et

$$\begin{aligned} |\{ext(r) \quad &: \quad r = (\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R}, (w, z) \in Q_{d,\ell}\}| \\ &\leq \quad |\{d' \in \mathbb{Z}| - k \leq d' \leq k\}| \\ &= \quad 2k + 1. \end{aligned}$$

Alors,

$$|Q_{d,\ell}/_{\equiv}| \leq 2^{(2k+1)\cdot|\Sigma|^2} \cdot 2|\Sigma|(k+1) \cdot (2k+1).$$

(ii) On veut montrer que

$$\sigma((x, y), \alpha, \beta) \equiv \sigma((w, z), \alpha, \beta)$$

quels que soient $\alpha, \beta \in \dot{\Sigma}$ et $(x, y), (w, z) \in Q_{d,\ell}$ tels que $(x, y) \equiv (w, z)$. Nous procédons selon les quatre cas de la fonction de transition σ. D'abord, supposons que $(w, z) = (\varepsilon, \varepsilon)$. Comme $|[(\varepsilon, \varepsilon)]_{\equiv}| = 1$, il est clair que $(x, y) = (w, z)$ de sorte que

$$\sigma((x, y), \alpha, \beta) = (\alpha, \beta) = \sigma((w, z), \alpha, \beta).$$

Maintenant, supposons que (w, z) est de classe f-$\mathcal{P}'$. Comme $(x, y) \equiv (w, z)$ on a aussi que (x, y) est de classe f-$\mathcal{P}'$. Alors,

$$\sigma((x, y), \alpha, \beta) = (x, y) \equiv (w, z) = \sigma((w, z), \alpha, \beta).$$

Ensuite, supposons que $(w\alpha, z)$ est équilibré et posons

$$v = \mathrm{Ext}((\varphi(w), \widehat{\varphi}(z), d)) = \mathrm{Ext}((\varphi(x), \widehat{\varphi}(y), d)).$$

On a

$$\begin{aligned} (\varphi(w\alpha), \widehat{\varphi}(z), d) \in \mathcal{R} \quad &\iff \quad (\varphi(w), \widehat{\varphi}(z), d) \in \mathcal{R} \text{ et } (\varphi(\alpha), v, 0) \in \mathcal{R} \\ &\iff \quad (\varphi(x), \widehat{\varphi}(y), d) \in \mathcal{R} \text{ et } (\varphi(\alpha), v, 0) \in \mathcal{R} \\ &\iff \quad (\varphi(x\alpha), \widehat{\varphi}(y), d) \in \mathcal{R} \end{aligned}$$

et

$$\begin{aligned} \mathrm{Ext}(\varphi(w\alpha), \widehat{\varphi}(z), d) = \quad \mathrm{Ext}(\varphi(\alpha), v, 0) \quad &= \mathrm{Ext}(\varphi(x\alpha), \widehat{\varphi}(y), d), \\ \mathrm{ext}(\varphi(w\alpha), \widehat{\varphi}(z), d) = \quad \mathrm{ext}(\varphi(\alpha), v, 0) \quad &= \mathrm{ext}(\varphi(x\alpha), \widehat{\varphi}(y), d). \end{aligned}$$

Alors, $\sigma((x,y),\alpha,\beta) = (x\alpha, y) \equiv (w\alpha, z) = \sigma((w,z),\alpha,\beta)$. Finalement, supposons que $(w\alpha, z)$ n'est pas équilibré, ce qui signifie que $(w, z\beta)$ est équilibré. Les arguments sont similaires à ceux du cas précédent.

(iii) D'abord, il n'y a qu'un seul état initial ($|I| = 1$). Le résultat suit de (i) et (ii). ∎

4.3 L'automate pour les morphismes uniformes sur un alphabet à 2 lettres

Dans cette section, on suppose que $\Sigma = \{a, b\}$, $\ell(a) = \ell(b)$ et $d \geq 0$. Le graphe

$$\mathcal{G}_{d,\ell} =< \dot{\Sigma} \times \dot{\Sigma}, Q_{d,\ell}/_{\equiv}, \sigma/_{\equiv} >,$$

illustré à la Figure 4.1, montre les sous-ensembles d'états suivants

$$\begin{aligned} T_1 &= \{(bba, bba), (baa, ba), (babb, bab)\}, \\ T_2 &= \{(bba, ba)\}, \\ T_3 &= \{(bbb, ba), (bba, bbb)\} \end{aligned}$$

en gris. Notons que pour alléger le dessin, on suppose que la seule transition possible à partir de l'état initial est la transition (b, b). En effet, la transition (a, a) mène à un graphe identique et les transitions (a, b) et (b, a) mènent à des états qui ne sont pas coaccessibles pour les six automates définis ci-bas. Pour la même raison, les étiquettes sur les boucles sont omises. Tout élément de $T_1 \cup T_2 \cup T_3$ est une composante fortement connexe de $\mathcal{G}_{d,\ell}$. En outre, le graphe a trois autres composantes fortement connexes à savoir

$$\begin{aligned} T_4 &= \{(bb, b), (bb, bb)\}, \\ T_5 &= \{(ba, ba), (bab, ba), (bab, bab), (baba, bab)\}, \\ T_6 &= \{(ba, bb), (baa, bb), (baa, bba)\}. \end{aligned}$$

Elles sont entourées de traits pointillés sur la Figure 4.1. Pour $i \in \{1, 2, 3, 4, 5, 6\}$, on définit l'automate

$$\mathcal{A}_i = \langle \dot{\Sigma} \times \dot{\Sigma}, Q_{d,\ell}/_{\equiv}, I/_{\equiv}, T_i/_{\equiv}, \sigma/_{\equiv} \rangle$$

et son langage associé $L_i = L_{\mathcal{A}_i}$.

Lemme 4.11 *Supposons que* $(\varphi(x), \widehat{\varphi}(y), d) \in \mathcal{R}$.

(i) *Si* $(x, y) \in L_1$, *alors* $\varphi(a) = uv$ *et* $\varphi(b) = uw$, *où* u, v *et* w *sont des* f*-palindromes.*

(ii) *Si* $(x, y) \in L_2$, *alors* $\varphi(a) = vu$ *et* $\varphi(b) = wu$, *où* u, v *et* w *sont des* f*-palindromes.*

(iii) *Si* $(x, y) \in L_3$, *alors* $\varphi(a) = uv$ *et* $\varphi(b) = uv$, *où* u *et* v *sont des* f*-palindromes.*

(iv) *Si* $(x, y) \in L_6$, *alors* $\varphi(a) = x'u$ *et* $\varphi(b) = \widehat{x'}u$, *où* u *est un* f*-palindrome.*

Démonstration. Les démonstrations sont similaires à celle du Lemme 4.7 et sont des conséquences des Lemmes 4.3 et 4.4. ∎

Le lemme suivant illustre l'émergence des antipalindromes. En effet, les deux composantes des couples appartenant au langage reconnu par $\mathcal{A}_6$ sont dépendantes l'une de l'autre. Elles sont reliées par le morphisme E définit à l'Exemple 1.9.

Lemme 4.12 *On a*

(i) $L_4 = \{(b^n, b^n) | n \geq 2\} \cup \{(b^n, b^{n-1}) | n \geq 2\}$.

(ii) $L_5 = \{((ba)^{n/2}, (ba)^{m/2}) | n, m \geq 2$ *et* $0 \leq n - m \leq 1\}$.

(iii) $L_6 = \{(ba \cdot v, b \cdot E(v) \cdot \gamma) | v \in \Sigma^*, \gamma \in \dot{\Sigma}\}$.

Démonstration. On déduit ces résultats de la Figure 4.1 où les états terminaux qui nous intéressent sont entourés de traits pointillés.
(i) Le langage reconnu par $\mathcal{A}_4$ est périodique dont la base du cycle est (b, b).
(ii) Le langage reconnu par $\mathcal{A}_5$ est périodique dont une base du cycle est (ba, ba).
(iii) Le langage reconnu par $\mathcal{A}_6$ est décrit par deux cycles connexes dont les bases sont (a, b) et (b, a). ∎

Lemme 4.13 *Soient* $\Sigma = \{a, b\}$ *et* $f : \Sigma \mapsto \Sigma$ *une involution. Soit* $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme uniforme et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Si* $\mathbf{u}$ *contient des* f*-palindromes arbitrairement longs, alors soit*

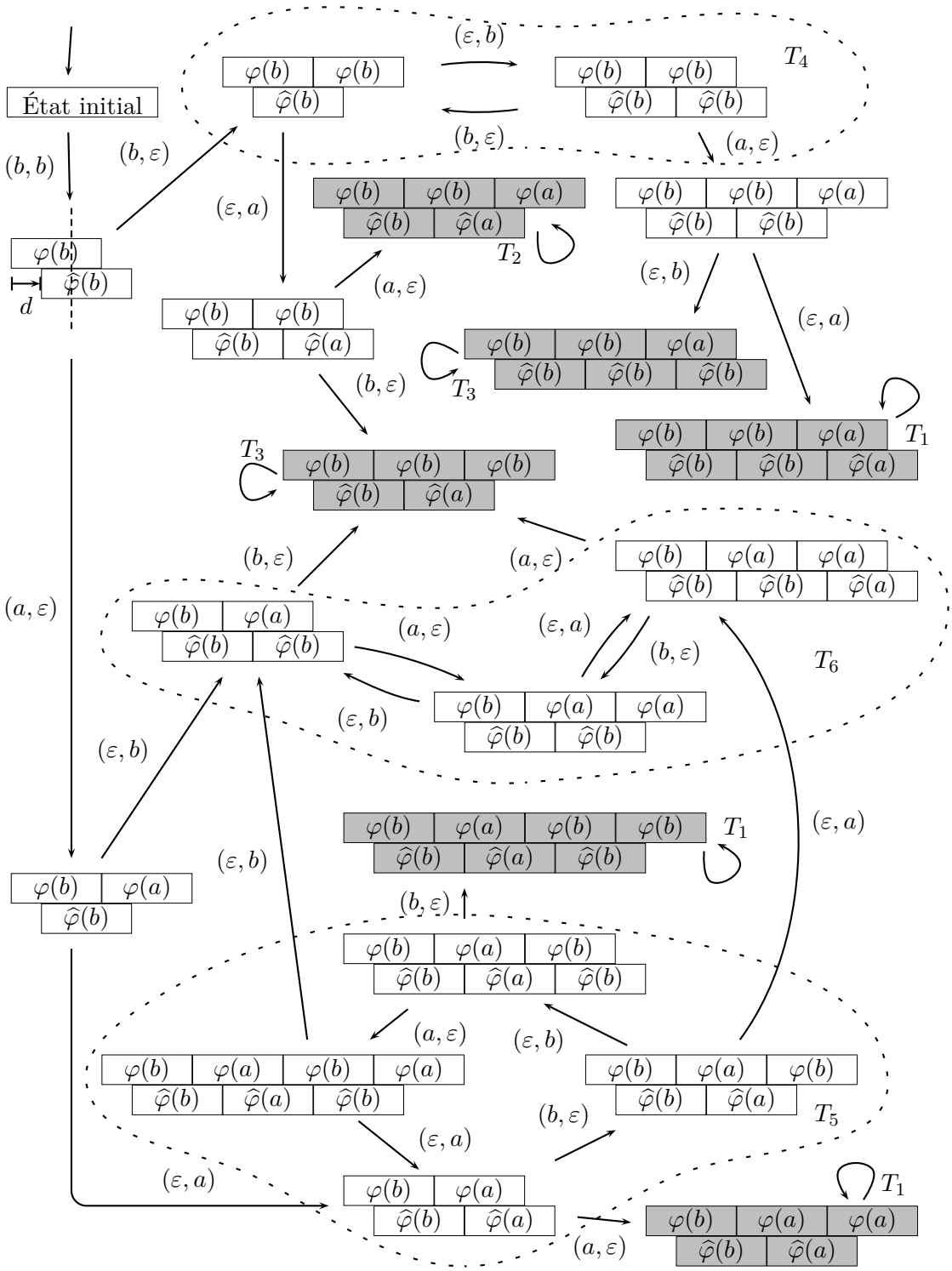


Figure 4.1 Le graphe $\mathcal{G}_{d,\ell} =< \dot{\Sigma} \times \dot{\Sigma}, Q_{d,\ell}/_{\equiv}, \sigma/_{\equiv} >$.

(i) $\varphi(a) = uv$ *et* $\varphi(b) = uw$*, où* u*,* v *et* w *sont des* f*-palindromes,*

(ii) $\varphi(a) = vu$ *et* $\varphi(b) = wu$*, où* u*,* v *et* w *sont des* f*-palindromes,*

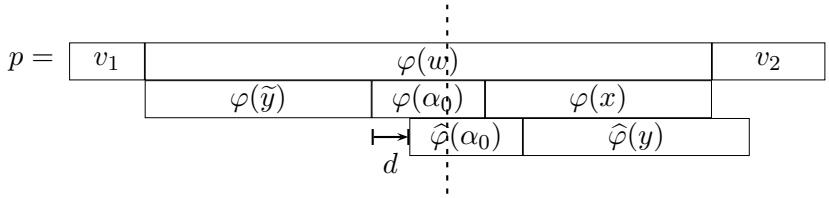
(iii) $\mathbf{u}$ *contient des puissances arbitrairement longues de* b *ou* a*,*

(iv) $\mathbf{u}$ *contient des puissances arbitrairement longues de* ab*,*

(v) $\varphi(a) = ux$ *et* $\varphi(b) = uf(\widetilde{x})$*, où* u *est un* f*-palindrome, et* $\mathbf{u}$ *contient des antipalindromes arbitrairement longs,*

(vi) $\varphi(a) = xu$ *et* $\varphi(b) = f(\widetilde{x})u$*, où* u *est un* f*-palindrome, et* $\mathbf{u}$ *contient des antipalindromes arbitrairement longs.*

Démonstration. Tout facteur de $\mathbf{u}$ et par conséquent tout f-palindrome $p \in f\text{-Pal}(\mathbf{u})$ peut s'écrire $p = v_1\varphi(w)v_2$ où $w \in \Sigma^*$, $v_1 \in \Sigma^* \setminus \Sigma^*\varphi(\Sigma)$ et $v_2 \in \Sigma^* \setminus \varphi(\Sigma)\Sigma^*$ tel qu'illustré dans le diagramme suivant.



Soient $\alpha_0 \in \Sigma$ et $x, y \in \Sigma^*$, tels que $w = \widetilde{y}\alpha_0 x$ et l'axe de symétrie de p est situé à l'intérieur du bloc $\varphi(\alpha_0)$, i.e. $|v_1\varphi(\widetilde{y}\alpha_0)| \geq \frac{1}{2}|p|$ et $|\varphi(\alpha_0 x)v_2| > \frac{1}{2}|p|$. On dit alors que $(\alpha_0 x, \alpha_0 y)$ est un *couple d'antécédents centrés (CAC)* associé au f-palindrome p. Puisque p est un f-palindrome, $(\varphi(\alpha_0 x), \widehat{\varphi}(\alpha_0 y), d) \in \mathcal{R}$ est un chevauchement où $d = |\varphi(x)v_2| - |v_1\varphi(\widetilde{y})|$. Comme φ est uniforme, les CAC $(\alpha_0 x, \alpha_0 y)$ sont clairement équilibrés par rapport à d et $|\varphi|$. On définit l'ensemble de tous les tels $(\alpha_0 x, \alpha_0 y)$:

$$K_{d,\ell} = \{c = (\alpha_0 x, \alpha_0 y) | c \text{ est un CAC associé à un } f\text{-palindrome de } \mathbf{u}\} \subset Q_{d,\ell},$$

où $\ell = |\varphi|$. Puisque l'ensemble f-Pal($\mathbf{u}$) est infini, il existe $d \in \mathbb{Z}$ tel que $|K_{d,\ell}| = \infty$. Tout d'abord, supposons que $d \geq 0$ et $\alpha_0 = b$.

Si $K_{d,\ell} \cap (L_1 \cup L_2 \cup L_3) \neq \emptyset$, alors en vertu du Lemme 4.11 (i-iii), soit

(a) $\varphi(a) = uv$ et $\varphi(b) = uw$, où u, v et w sont des f-palindromes,

(b) $\varphi(a) = vu$ et $\varphi(b) = wu$, où u, v et w sont des f-palindromes,

(c) $\varphi(a) = uv$ et $\varphi(b) = uv$, où u et v sont des f-palindromes.

Autrement, si $K_{d,\ell} \cap (L_1 \cup L_2 \cup L_3) = \emptyset$, alors

$$|K_{d,\ell} \cap (L_4 \cup L_5 \cup L_6)| = \infty,$$

car les couples de $K_{d,\ell}$ commencent tous par (b, b) et T_4, T_5 et T_6 sont les seules autres composantes fortement connexes de $\mathcal{G}_{d,\ell}$ mises à part celles de $T_1 \cup T_2 \cup T_3$. Il est à noter que puisque $\varphi(\mathbf{u}) = \mathbf{u}$, alors $\widetilde{y}bx \in \mathrm{Fact}(\mathbf{u})$. Alors, par le Lemme 4.11 (iv) et le Lemme 4.12, soit

(d) $\mathbf{u}$ contient des puissances arbitrairement longues de b,

(e) $\mathbf{u}$ contient des puissances arbitrairement longues de ba,

(f) $\varphi(a) = xu$ et $\varphi(b) = f(\widetilde{x})u$, où u est un f-palindrome, et $\mathbf{u}$ contient des antipalindromes arbitrairement longs.

Si $d < 0$, on considère $(\widehat{\varphi}(\alpha_0 y), \varphi(\alpha_0 x), -d) \in \mathcal{R}$ et on obtient les mêmes résultats (a-f) pour $\widehat{\varphi}$. Si $\alpha_0 = a$, on obtient les mêmes résultats sauf pour (d) où il faut ajouter que $\mathbf{u}$ peut contenir des puissances arbitrairement longues de a. ∎

CHAPITRE V

CONJECTURE DE HOF, KNILL ET SIMON

Dans ce chapitre, nous obtenons nos principaux résultats. D'abord, nous démontrons la conjecture de Hof, Knill et Simon pour les morphismes uniformes sur un alphabet binaire. Ensuite, nous caractérisons les points fixes de morphismes uniformes sur un alphabet binaire ayant une complexité antipalindromique infinie. Finalement, nous énonçons quelques conjectures immédiates.

5.1 Morphismes uniformes sur un alphabet binaire

Dans cette section, nous obtenons d'abord un sous-cas d'un résultat démontré récemment par (Tan, 2007). Notre approche est différente et utilise la notion d'antipalindromes qui émerge naturellement.

Proposition 5.1 *Soit* $\Sigma = \{a, b\}$. *Soit* $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme primitif uniforme et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Si* $\mathbf{u}$ *contient des palindromes arbitrairement longs, alors soit* φ, $\widetilde{\varphi}$ *ou* φ^2 *est de classe* $\mathcal{P}$.

Démonstration. Nous procédons selon les cas obtenus au Lemme 4.13. Dans le cas (i), φ est de classe $\mathcal{P}$. Dans le cas (ii), $\widetilde{\varphi}$ est de classe $\mathcal{P}$. Dans le cas (iii), on obtient du Lemme 2.16 que $\mathbf{u} = a^\omega$ ou $\mathbf{u} = b^\omega$ ce qui est une contradiction, car φ est primitif.

Dans le cas (iv), on obtient du Lemme 2.16 que $\mathbf{u} = (ab)^\omega$ ou $\mathbf{u} = (ba)^\omega$. Si $|\varphi(a)|$ est impair, alors $\varphi(a)$ et $\varphi(b)$ sont des palindromes, de sorte que φ est de classe $\mathcal{P}$. Si $|\varphi(a)|$

est pair, alors $\varphi(a) = \varphi(b) = pq$ où p et q sont des palindromes, de sorte que φ est de classe $\mathcal{P}$.

Dans le cas (v), on obtient que

$$\varphi(a) = ux \text{ et } \varphi(b) = u\widetilde{x},$$

où u est un palindrome. Alors, nous procédons à nouveau selon les cas obtenus au Lemme 4.13, car **u** contient des antipalindromes arbitrairement longs.

Dans le cas (v.i), on a $\varphi(a) = pv$ et $\varphi(b) = pw$, où p, v et w sont des antipalindromes. Alors, on a $ux = pv$ et $u\widetilde{x} = pw$. Si $|u| \neq |p|$, on obtient du Corollaire 2.15 (i) que $x = \widetilde{x}$, de sorte que φ est de classe $\mathcal{P}$. Si $|u| = |p|$, alors $u = p = \varepsilon$, car c'est le seul mot qui est à la fois un palindrome et un antipalindrome. Donc, on a que $\varphi(a)$ est un antipalindrome et $\widetilde{\varphi(a)} = \varphi(b)$ et selon le Lemme 3.25, φ^2 est de classe $\mathcal{P}$.

Dans le cas (v.ii), on a $\varphi(a) = vp$ et $\varphi(b) = wp$, où p, v et w sont des antipalindromes. Alors, $ux = vp$ et $u\widetilde{x} = wp$. Si $|u| + |p| > 0$, alors par la Proposition 2.14 (iii), x est un palindrome de sorte que φ est de classe $\mathcal{P}$. Si $|u| = |p| = 0$, alors, $\varphi(a)$ est un antipalindrome et $\varphi(b) = \widetilde{\varphi(a)}$ et selon le Lemme 3.25, φ^2 est de classe $\mathcal{P}$.

Dans le cas (v.iii), on obtient une contradiction comme dans le cas (iii), car φ est primitif.

Dans le cas (v.iv), on obtient comme précédemment que φ est de classe $\mathcal{P}$ (voir cas (iv)).

Dans le cas (v.v), on a $\varphi(a) = py$ et $\varphi(b) = pe(\widetilde{y})$ où p est un antipalindrome. Alors, $ux = py$ et $u\widetilde{x} = pe(\widetilde{y})$. Si $|u| \neq |p|$, on obtient du Corollaire 2.15 (ii) que $x = \widetilde{x}$, de sorte que φ est de classe $\mathcal{P}$. Si $|u| = |p|$, alors $u = p = \varepsilon$, car c'est le seul mot qui est à la fois un palindrome et un antipalindrome. Aussi, $x = e(x)$ ce qui implique que $x = \varepsilon$ et $\varphi(a) = \varepsilon$, une contradiction.

Dans le cas (v.vi), on a $\varphi(a) = yp$ et $\varphi(b) = e(\widetilde{y})p$, où p est un antipalindrome. Alors, $ux = yp$ et $u\widetilde{x} = e(\widetilde{y})p$. Si $|p| + |u| > 0$, alors selon la Proposition 2.14 (ii), x est un palindrome de sorte que φ est de classe $\mathcal{P}$. Si $|p| = |u| = 0$, alors $x = \varepsilon$, une contradiction.

Dans le cas (vi), on obtient que

$$\varphi(a) = xu \text{ et } \varphi(b) = \widetilde{x}u,$$

où u est un palindrome. Alors, nous procédons à nouveau selon les cas obtenus au Lemme 4.13, car $\mathbf{u}$ contient des antipalindromes arbitrairement longs. La suite se fait de façon similaire au cas (v). ∎

Nous pouvons maintenant énoncer un théorème qui découle de la proposition précédente et d'un lemme du Chapitre 3. Ce résulat est connu depuis peu pour les morphismes sur un alphabet binaire (Tan, 2007).

Théorème 5.2 *Soient* $\Sigma = \{a, b\}$, $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme primitif uniforme et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Alors,* $\mathbf{u}$ *contient des palindromes arbitrairement longs si et seulement si* φ, $\widetilde{\varphi}$ *ou* φ^2 *est de classe* $\mathcal{P}$.

Démonstration. Le résultat suit de la Proposition 5.1 et du Lemme 3.22 (ii). ∎

5.2 Théorème principal

Nous désirons maintenant généraliser la Proposition 5.1 pour les antipalindromes. L'énoncé doit être adapté comme le montre l'exemple qui suit.

Exemple 5.3 Soit le morphisme primitif uniforme sur l'alphabet $\Sigma = \{a, b\}$ suivant

$$\begin{array}{rccc} \varphi: & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & aba \\ & b & \mapsto & abb \end{array}$$

et $\mathbf{u} = \varphi(\mathbf{u}) = abaabbabaabaabbabbab\cdots$ son point fixe. En vertu du Lemme 3.22 (iii), on a que $\mathbf{u}$ contient une infinité d'antipalindromes (calculs à l'Appendice A). Néanmoins, ni φ, ni $\widetilde{\varphi}$, ni φ^2 ne sont de classe E-$\mathcal{P}$. ♢

Nous pouvons maintenant énoncer le théorème principal de ce mémoire qui donne d'une certaine façon une réciproque au Lemme 3.22 (iii).

Théorème 5.4 *Soient* $\Sigma = \{a, b\}$, $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme primitif uniforme et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Si* $\mathbf{u}$ *contient des antipalindromes arbitrairement longs, alors soit* φ, $\widetilde{\varphi}$, $\varphi \circ \mu$ *ou* $\widetilde{\varphi} \circ \mu$ *est de classe E-*$\mathcal{P}$*, où* μ *est le morphisme de Thue-Morse.*

Démonstration. Nous procédons selon les cas obtenus au Lemme 4.13. Dans le cas (i), on obtient que φ est de classe *E-*$\mathcal{P}$. Dans le cas (ii), on obtient que $\widetilde{\varphi}$ est de classe *E-*$\mathcal{P}$. Dans le cas (iii), on obtient du Lemme 2.16 que $\mathbf{u} = a^\omega$ ou $\mathbf{u} = b^\omega$ ce qui est une contradiction puisque φ est primitif.

Dans le cas (iv), on obtient du Lemme 2.16 que $\mathbf{u} = (ab)^\omega$ ou $\mathbf{u} = (ba)^\omega$. Si $|\varphi(a)|$ est pair, alors $\varphi(a)$ et $\varphi(b)$ sont des antipalindromes, de sorte que φ est de classe *E-*$\mathcal{P}$. Si $|\varphi(a)|$ est impair, alors $\varphi(ab)$ et $\varphi(ba)$ sont des antipalindromes, de sorte que $\varphi \circ \mu$ est de classe *E-*$\mathcal{P}$. Dans le cas (v), on obtient que

$$\varphi(a) = ux \text{ et } \varphi(b) = ue(\widetilde{x}),$$

où u est un antipalindrome. Selon le Lemme 3.21, $\varphi \circ \mu$ est de classe *E-*$\mathcal{P}$. Dans le cas (vi), on obtient que

$$\varphi(a) = xu \text{ et } \varphi(b) = e(\widetilde{x})u,$$

où u est un antipalindrome. Alors, selon le Lemme 3.21, $\widetilde{\varphi} \circ \mu$ est de classe *E-*$\mathcal{P}$. ∎

5.3 Nouvelles conjectures

En observant les Exemples 3.23 et 5.3, on voudrait élaguer l'énoncé du Théorème 5.4 en démontrant la conjecture suivante :

Conjecture 5.5 *Soient* $\Sigma = \{a, b\}$, $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme primitif uniforme et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Si* $\mathbf{u}$ *contient des antipalindromes arbitrairement longs, alors soit* $\varphi \circ \mu$ *ou* $\widetilde{\varphi} \circ \mu$ *est de classe E-*$\mathcal{P}$*, où* μ *est le morphisme de Thue-Morse.*

Ainsi, en vertu du Théorème 5.2 et des Lemmes 3.21 et 3.22, on aurait établi le résultat suivant (que nous formulons pour les morphismes non nécessairement uniformes) qui

donne une équivalence appréciable pour les points fixes de morphismes. Notez que la partie de l'énoncé pour les palindromes a déjà été démontrée par (Tan, 2007).

Conjecture 5.6 *Soient* $\Sigma = \{a, b\}$, $\varphi : \Sigma^* \mapsto \Sigma^*$ *un morphisme primitif et* $\mathbf{u} = \varphi(\mathbf{u})$ *un point fixe. Alors* $\mathbf{u}$ *contient des* f*-palindromes arbitrairement longs si et seulement si l'une des conditions suivantes est vérifiée :*

(i) $\varphi \circ \mu$ *ou* $\widetilde{\varphi} \circ \mu$ *est de classe E-*$\mathcal{P}$*, où* μ *est le morphisme de Thue-Morse.*

(ii) φ, $\widetilde{\varphi}$ *ou* φ^2 *est de classe* $\mathcal{P}$*.*

Exemple 5.7 Soit le morphisme primitif sur l'alphabet $\Sigma = \{a, b\}$ suivant

$$\begin{array}{rccc} \varphi : & \Sigma^* & \to & \Sigma^* \\ & a & \mapsto & ab \\ & b & \mapsto & baabba \end{array}$$

et $\mathbf{u} = \varphi(\mathbf{u}) = abbaabbabaabbaababbaab\cdots$ son point fixe. En vertu du Lemme 3.22 (i), on a que $\mathbf{u}$ contient une infinité de palindromes (calculs à l'Appendice A). Or, φ^2 est de classe $\mathcal{P}$. ♢

CONCLUSION

Le but de ce mémoire était de démontrer la conjecture de Hof, Knill et Simon énoncée pour la première fois en 1995. Celle-ci donne une caractérisation des mots infinis points fixes de morphismes dont la complexité palindromique est infinie. Nous l'avons résolue pour les points fixes de morphismes uniformes sur un alphabet à deux lettres. Toutefois, ce résultat est déjà connu. En effet, un article publié pendant la rédaction de ce mémoire donne la solution pour les points fixes de morphismes quelconques sur un alphabet à deux lettres (Tan, 2007). Nous sommes tout de même parvenus à un résultat original : par notre approche, nous avons démontré un résultat similaire pour les antipalindromes. Notre réalisation principale est de donner une caractérisation des mots infinis points fixes de morphismes uniformes sur un alphabet à deux lettres dont la complexité f-palindromique est infinie.

En étudiant la conjecture, nous avons démontré plusieurs propriétés combinatoires sur les mots. Plus précisément, nous avons établi plusieurs résultats sur la structure des f-palindromes et leur périodicité. Nous avons fait une étude de certains morphismes, ceux de classe f-$\mathcal{P}$. Finalement, nous avons introduit la définition de chevauchement. Cela nous a permis de construire un graphe de chevauchements, assise de notre démonstration de la conjecture.

Toutes ces recherches ont mené au développement d'un outil informatique voué à l'étude de problèmes de la combinatoire des mots constitué d'un ensemble de classes et de fonctions écrites en langage Python. Toutes ces fonctions sont jointes en appendice. Prochainement, elles seront jumelées à un paquetage ruby réalisé au LaCIM depuis 1995 sous la direction de Srečko Brlek (Brlek et al., 2006). Nous avons le projet (Bergeron, Brlek, Glen, Labbé, Saliola, 2008) d'en faire un nouveau paquetage qui s'intégrera à Sage, un logiciel libre de mathématiques (Sage, 2008). Déjà, des collaborateurs tels

qu'Amy Glen se sont joints au projet.

Défaut et complexité palindromique des points fixes

Les résultats de ce mémoire et ceux de (Tan, 2007) permettent de distinguer les points fixes de morphismes sur un alphabet binaire dont la complexité palindromique est finie versus infinie. Or, l'étude de la complexité palindromique peut être encore plus fine.

Rappelons que le défaut $D(w)$ d'un mot w a été défini à la fin de la section 1.1 comme étant la différence entre le nombre maximal de palindromes qu'il peut contenir et le nombre qu'il contient. Dans le cas des mots périodiques, on trouve dans (Brlek, Hamel, Nivat, Reutenauer, 2004) un résultat (algorithmique) optimal permettant de le calculer. Mais, le défaut n'a jamais été calculé explicitement pour les mots infinis en général. Un problème naturel est de classifier les mots infinis $\mathbf{u}$ points fixes de morphismes primitifs selon les classes suivantes :

$\lvert\mathrm{Pal}(\mathbf{u})\rvert$	$D(\mathbf{u})$	Exemples
fini	∞	Exemple 3.3.
∞	0	Mots sturmiens, mot de Fibonacci, Exemple 3.1.
∞	$0 < D(\mathbf{u}) < \infty$	Exemple 1.7.
∞	∞	Mot de Thue-Morse.

Évidemment, ceci demande une étude fine de la complexité palindromique, et pour le troisième cas de cette classification, nous avons énoncé la conjecture suivante :

Conjecture 5.8 (Blondin Massé, Brlek, Labbé, 2008) *Soit le point fixe* $\mathbf{u} = \varphi(\mathbf{u})$ *d'un morphisme primitif* φ*. Si le défaut de* $\mathbf{u}$ *est tel que* $0 < D(\mathbf{u}) < \infty$*, alors* $\mathbf{u}$ *est périodique.*

La démonstration de la Conjecture 5.8 permettrait de caractériser les points fixes ayant un défaut fini non nul. Ensuite, il resterait à différencier les points fixes pleins (comme les mots sturmiens et le mot de Fibonacci) de ceux dont la complexité palindromique et le défaut sont infinis (comme le mot de Thue-Morse).

Problèmes ouverts

Plusieurs pistes de recherches restent ouvertes :

Améliorer la Proposition 2.12. En faire une équivalence à l'image du Lemme 2.10.

Équations de géométrie discrètes. Certaines équations qui apparaissent dans des problèmes de géométrie discrète ressemblent aux équations de la Propostion 2.14 et des lemmes suivants. Elles sont toujours non résolues à ce jour. Nous croyons que des méthodes similaires et les outils informatiques développés pour l'étude des chevauchements pourront faire des avancées dans ce domaine.

Améliorer le Lemme 3.21. Vérifier les équivalences pour les morphismes non uniformes.

Étude des mots avec peu de f-palindromes. Construire les mots dont les facteurs f-palindromes sont contenus dans un ensemble fini donné de f-palindromes à l'image de la section 3 de (Brlek, Hamel, Nivat, Reutenauer, 2004).

Points fixes de morphismes uniformes sur un alphabet à deux lettres. Résoudre la Conjecture 5.5. En observant les Exemples 3.23 et 5.3, on voudrait élaguer l'énoncé du Théorème 5.4 en démontrant que sous les mêmes conditions soit $\varphi \circ \mu$ ou $\widetilde{\varphi} \circ \mu$ est de classe E-$\mathcal{P}$, où μ est le morphisme de Thue-Morse.

Points fixes de morphismes sur un alphabet à deux lettres. Résoudre la Conjecture 5.6. En supposant la Conjecture 5.5 vraie, il reste à résoudre la partie pour la complexité antipalindromique pour les morphismes non uniformes. Lorsque $\ell(a)$ et $\ell(b)$ sont copremiers, le graphe des chevauchement semble plus facile à résoudre que celui pour les morphismes uniformes qu'on a étudié (voir la Figure B.1). En effet, le graphe possède des boucles simples mais pas de doubles boucles qui apparaissent dans le cas uniforme (voir la Figure B.2). De plus, certains graphes de chevauchements pour les cas où $\ell(a)$ divise $\ell(b)$ se ressemblent entre eux (voir les Figures B.3 et B.4). Notre approche pourrait mener à de nouvelles équations en combinatoire des mots.

Points fixes de morphismes uniformes sur un alphabet à trois lettres. Résou-

dre la Conjecture 3.7 pour les morphismes uniformes sur un alphabet à trois lettres en étudiant les graphes de chevauchement.

Points fixes de morphismes primitifs quelconques. Résoudre la Conjecture 3.7 dans le cas général. Étendre ensuite ce résultat aux f-palindromes.

Caractérisation de points fixes et complexité palindromique. En premier lieu, résoudre la Conjecture 5.8 qui traite le cas des points fixes de morphismes dont le défaut est fini non nul. En deuxième lieu, parmi les mots infinis non périodiques points fixes de morphismes de classe $\mathcal{P}'$, déterminer quels sont ceux qui, comme le mot de Thue-Morse, ne sont pas pleins et lesquels le sont.

Étude des mots f-pleins. À l'image des palindromes, tout mot possède au plus un suffixe f-palindrome unioccurent. Ainsi, tout mot w possède au plus $|w| + 1$ f-palindromes. Toutefois, en considérant $\Sigma = \{a, b\}$ et $f = E$, seul le mot vide ε atteint cette limite. Quelle est la bonne notion de E-plein ? Est-ce que les puissances $(ab)^n$ et $(ba)^n$ sont les seuls mots qui contiennent $|w|$ E-palindromes ?

Étude des lacunes f-palindromiques. Récemment, nous avons étudié les lacunes palindromiques du mot de Thue-Morse (Blondin Massé, Brlek, Labbé, 2008; Blondin Massé, Brlek, Frosini, Labbé, Rinaldi, 2008). Qu'en est-il des lacunes antipalindromiques du mot de Thue-Morse ? Quelles sont les lacunes f-palindromiques des points fixes de morphismes en général ?

Étendre la notion de f-palindrome aux permutations. La condition $w = f(\widetilde{w})$ nous contraint aux involutions f de l'alphabet. Une notion plus générale de palindrome est d'utiliser la condition $w = x\gamma f(\widetilde{x})$ où $\gamma \in \dot{\Sigma}$ et f est une permutation de l'alphabet. Par exemple, en considérant $f : a \mapsto b, b \mapsto c, c \mapsto a$, on a que *abbcccaaaccb*, *bccaaabbbaac* et *cbbaaabbbcca* sont des f-palindromes. Supposer que $\gamma \in \Sigma^*$ ne serait pas non plus un non sens et aurait des applications en géométrie discrète et en analyse de certaines séquences biologiques telles que l'ADN des chloroplastes.

APPENDICE A

QUELQUES CALCULS

Dans cet Appendice, on explique comment utiliser certaines des fonctions et des classes définies à l'Appendice C. Nous supposons d'abord qu'une version récente de Python est utilisée et que les 5 fichiers `partition.py`, `mot.py`, `morphisme.py`, `algo_defaut.py` et `etat.py` sont dans le répertoire courant.

D'abord, on importe les classes des fichiers de l'Appendice C de la façon suivante :

```
>>> from partition import Partition
>>> from mot import Mot
>>> from morphisme import Morphisme
>>> from algo_defaut import algo_defaut
>>> from etat import Etat
```

Une partition est un ensemble d'ensembles disjoints.

```
>>> p=Partition(range(6));p
[[0], [1], [2], [3], [4], [5]]
```

On peut joindre des classes.

```
>>> p.joindre(1,2);p
[[0], [1, 2], [3], [4], [5]]
```

```
>>> p.joindre(0,5);p
[[0, 5], [1, 2], [3], [4]]
>>> p.joindre(5,1);p
[[0, 5, 1, 2], [3], [4]]
```

Chaque classe a un unique représentant.

```
>>> [p.representantDe(x) for x in range(6)]
[0, 0, 0, 3, 4, 0]
```

On les utilise pour créer des classes d'équivalence (une partition) de l'alphabet. Deux lettres sont dans la même classe si elles se chevauchent.

```
>>> cheval = Mot('cheval')
>>> propre = Mot('propre')
>>> #Illustration du chevauchement de décalage = 0 :
>>> #cheval
>>> #propre
>>> p = cheval.chevauche(propre,0);p
[['a', 'h', 'r'], ['l', 'e', 'o'], ['v', 'c', 'p']]
>>> #Illustration du chevauchement de décalage = 1 :
>>> #cheval
>>> # propre
>>> p = cheval.chevauche(propre,1);p
[['a', 'h', 'p'], ['c'], ['l', 'e', 'r'], ['v', 'o']]
>>> #Illustration du chevauchement de décalage = 3 :
>>> #cheval
>>> #   propre
>>> q = cheval.chevauche(propre, 3);q
[['a', 'r'], ['c'], ['e'], ['h'], ['l', 'o'], ['v', 'p']]
```

Ensuite, on peut réécrire chaque mot en remplaçant chaque lettre par son représentant dans la partition.

```
>>> print cheval.representation(q), propre.representation(q)
cheval valvae
```

Ainsi, on peut étudier des équations sur les mots. Par exemple, quel mot chevauche son propre miroir ?

```
>>> r = cheval.chevauche(cheval.miroir(), 0);r
[['c', 'l'], ['e', 'v'], ['h', 'a']]
```

Réponse : un palindrome.

```
>>> cheval.representation(r)
'cheehc'
```

Utilisons ce principe pour illustrer le Lemme 2.11.

```
>>> x = Mot('abcde')
>>> y = Mot('fghijklmnopq')
>>> z = Mot('rstuv')
>>> p = (x + y).chevauche(y + z, 0)
>>> p = y.chevauche(y.miroir(), 0, p)
>>> X = x.representation(p)
>>> Y = y.representation(p)
>>> Z = z.representation(p)
>>> print X,Y,Z
aacdc aacdcaacdcaa cdcaa
>>> (X+Y).estPalindrome()
True
```

La suite illustre des exemples et des Lemmes présentés dans ce mémoire.

L'exemple 2.8 :

```
>>> tm = Morphisme('a->ab,b->ba');tm
a->ab,b->ba
>>> E = {'a':'b','b':'a'}
>>> for i in range(6):
...       w = tm('a',i)
...       print w,w.estPalindrome(),w.estFPalindrome(E)
a True False
ab False True
abba True False
abbabaab False True
abbabaabbaababba True False
abbabaabbaababbabaabababbaabbabaab False True
```

Notez que la fonction `liste_nombre_pal` retourne la liste du nombre de f-palindromes dans les mots obtenus par l'itération du morphisme par défaut sur la lettre `a`.

L'exemple 3.1 :

```
>>> phi = Morphisme('a->bbaba,b->bba')
>>> phi.liste_nombre_pal(8)
[2, 6, 20, 72, 266, 990, 3692, 13776]
```

L'exemple 3.2 :

```
>>> tm = Morphisme('a->ab,b->ba')
>>> tm.liste_nombre_pal(14)
[2, 3, 5, 9, 15, 29, 53, 109, 205, 429, 813, 1709, 3245, 6829]
```

L'exemple 3.3 :

```
>>> phi = Morphisme('a->abb,b->ba')
>>> phi.liste_nombre_pal(15)
[2, 4, 8, 15, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23]
>>> phi.liste_nombre_pal(10,lettreInitiale='b')
[2, 3, 6, 13, 18, 23, 23, 23, 23, 23]
```

L'exemple 3.23 :

```
>>> phi = Morphisme('a->abab,b->abba')
>>> phi.liste_nombre_pal(9, involution=E)
[1, 4, 14, 40, 72, 72, 72, 72, 72]
```

L'exemple 3.24 :

```
>>> f = {'a':'b','b':'a','c':'c'}
>>> phi = Morphisme('a->acbcc,b->acbab,c->acbba')
>>> phi.liste_nombre_pal(7, involution=f)
[1, 4, 22, 50, 50, 50, 50]
```

Le Lemme 4.7 :

```
>>> from etat import Etat
>>> e = Etat(7,7,3,'bba','ba') ; e

hijklmnhijklmnabcdefg
[  B  ][  B  ][  A  ]
   [ B~  ][ A~  ]
   nmlkjihgfedcba
```

```
Morphisme     : a->abakllk,b->hihkllk
Est Classe P  : False
Est Classe P' : True
Extension     : Ext = kllk, ext = 4
```

Le Lemme 4.8 :

```
>>> from etat import Etat
>>> e = Etat(10,14,3,'baa','bb') ; e

klmnopqrstuvwxabcdefghijabcdefghij
[     B      ][   A    ][   A    ]
   [    B~      ][    B~      ]
   xwvutsrqponmlkxwvutsrqponmlk

Morphisme     : a->cbcdcbcjij,b->cbcdcbcjijcbcd
Est Classe P  : False
Est Classe P' : True
Extension     : Ext = jij, ext = 3
>>> e = Etat(9,13,3,'baa','bb') ; e

jklmnopqrstuvabcdefghiabcdefghi
[     B     ][   A   ][   A   ]
   [    B~     ][    B~     ]
   vutsrqponmlkjvutsrqponmlkj

Morphisme     : a->cbcdcbcii,b->cbcdcbciicbcd
Est Classe P  : False
Est Classe P' : True
Extension     : Ext = ii, ext = 2
```

L'exemple 5.3 :

```
>>> phi = Morphisme('a->aba,b->abb')
>>> print phi.estClasseP(E)
False
>>> phi.miroir().estClasseP(E)
False
>>> (phi^2).estClasseP(E)
False
>>> phi.liste_nombre_pal(9, involution=E)
[1, 3, 7, 19, 55, 163, 487, 1459, 4375]
```

L'exemple 5.7 :

```
>>> phi = Morphisme('a->ab,b->baabba');phi
a->ab,b->baabba
>>> print phi.estClasseP()
False
>>> phi.miroir().estClasseP()
False
>>> (phi^2).estClasseP()
True
>>> phi.liste_nombre_pal(8)
[2, 3, 9, 29, 117, 437, 1845, 6965]
>>> phi('a',3)
'abbaabbabaabbaabbabbaabbabaabbaab'
```

Finalement, voici comment on obtient les graphes de l'Appendice B.

```
>>> from etat import Etat
>>> Etat(6,11,4).creeArbreDirige()
```

```
Resultats dans Graphes/graphe6_11_4.dot
>>> Etat(7,7,3).creeArbreDirige()
Resultats dans Graphes/graphe7_7_3.dot
>>> Etat(4,12,2).creeArbreDirige()
Resultats dans Graphes/graphe4_12_2.dot
>>> Etat(5,15,3).creeArbreDirige()
Resultats dans Graphes/graphe5_15_3.dot
```

APPENDICE B

GRAPHES DE CHEVAUCHEMENT

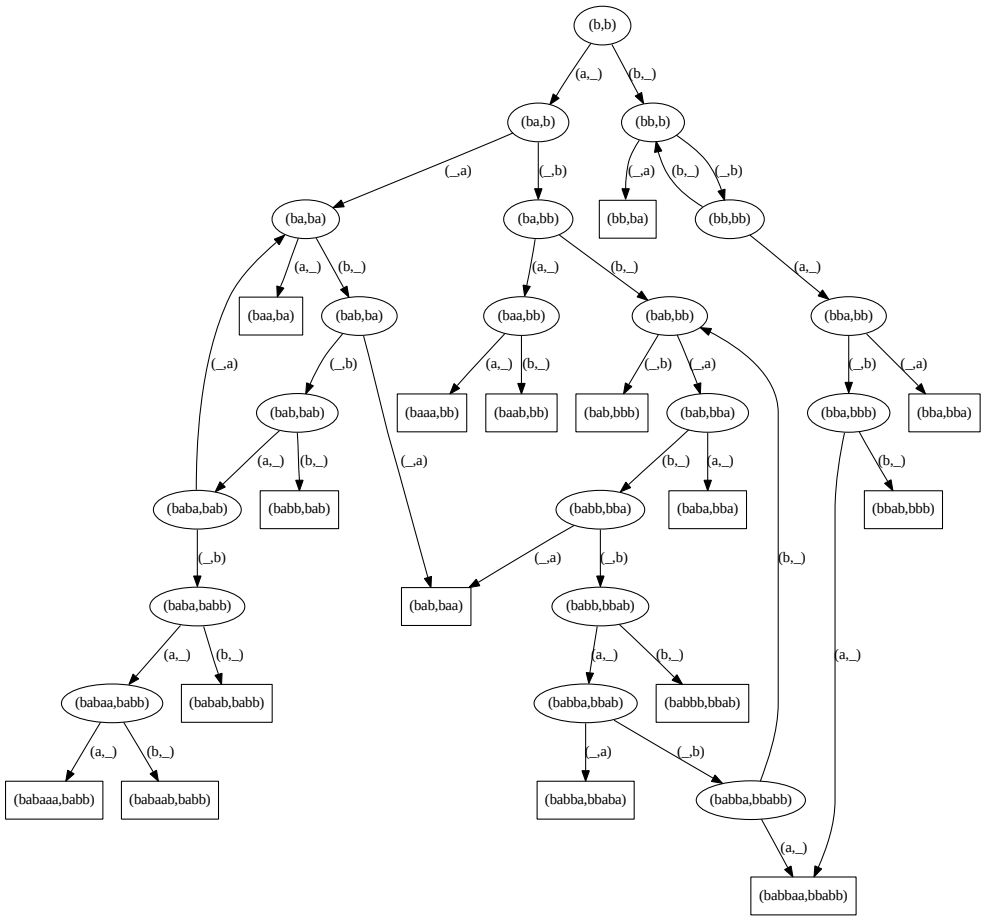


Figure B.1 Le graphe des chevauchements obtenu pour $|\varphi(a)| = 6$ et $|\varphi(b)| = 11$ et un décalage de 4. Les rectangles représentent les états de classe $\mathcal{P}$.

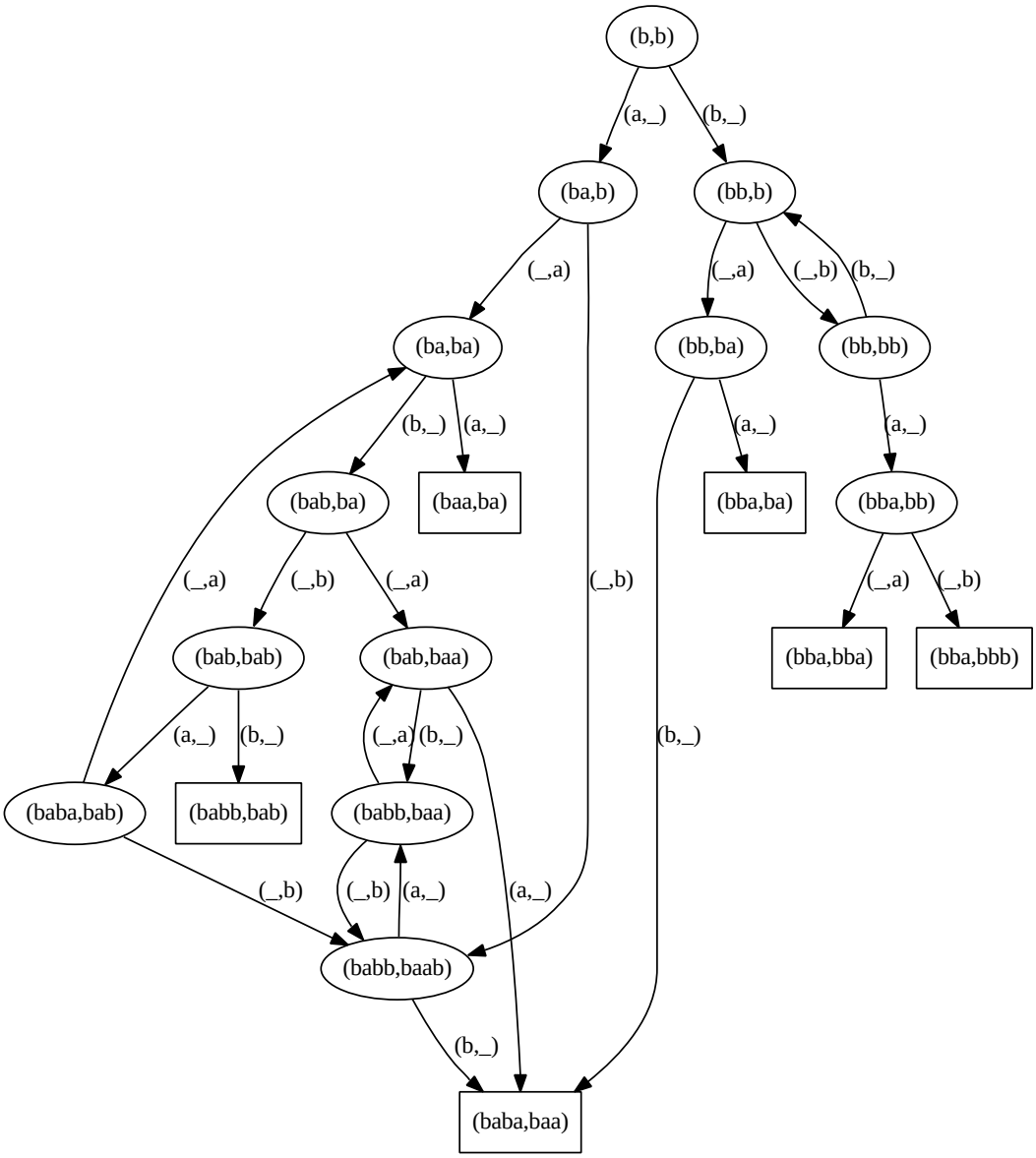


Figure B.2 Le graphe des chevauchements obtenu pour $|\varphi(a)| = 7$ et $|\varphi(b)| = 7$ et un décalage de 3. Remarquez que ce graphe est identique à celui de la Figure 4.1.

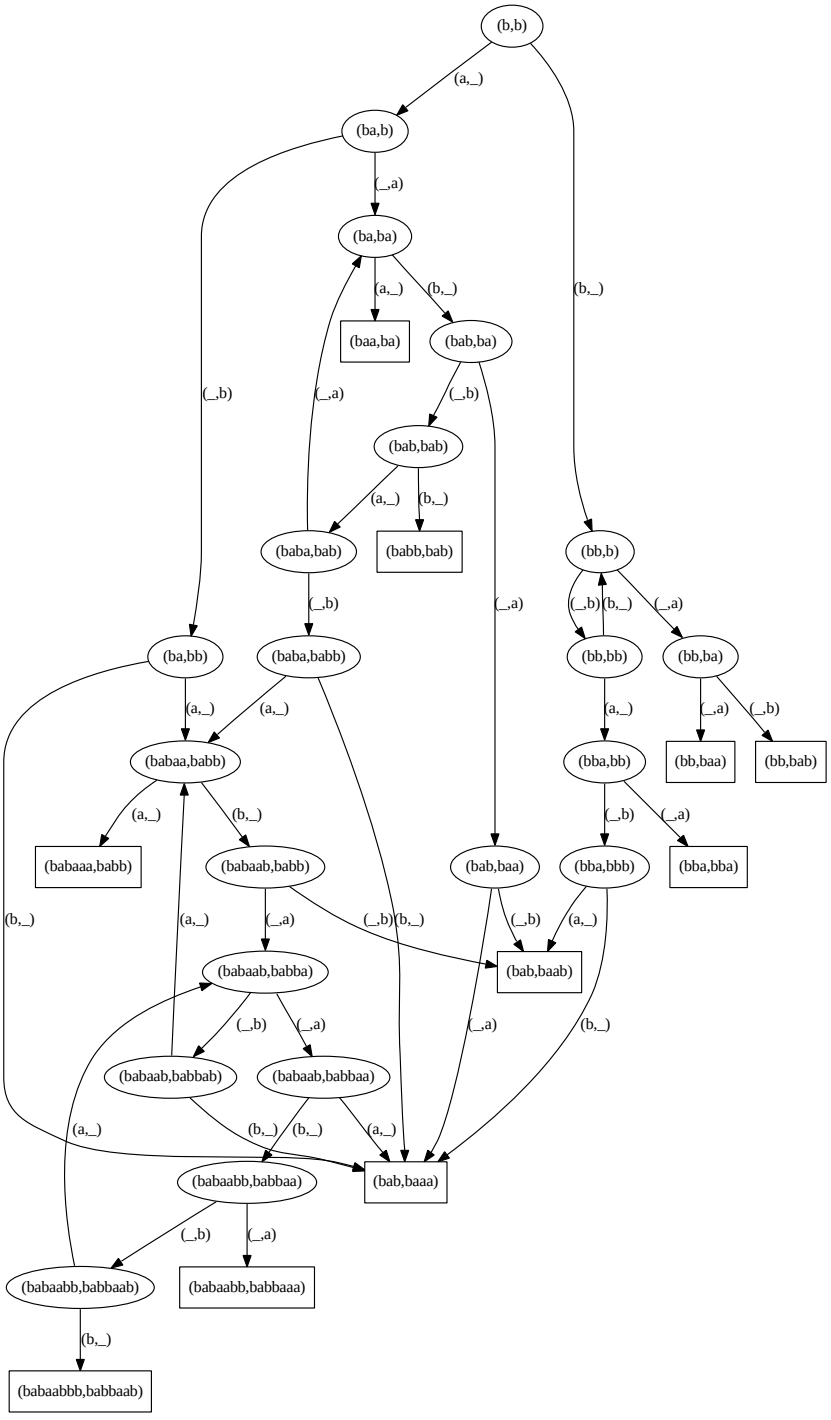


Figure B.3 Le graphe des chevauchements obtenu pour $|\varphi(a)| = 4$ et $|\varphi(b)| = 12$ et un décalage de 2. Remarquez que ce graphe est identique à celui de la Figure B.4.

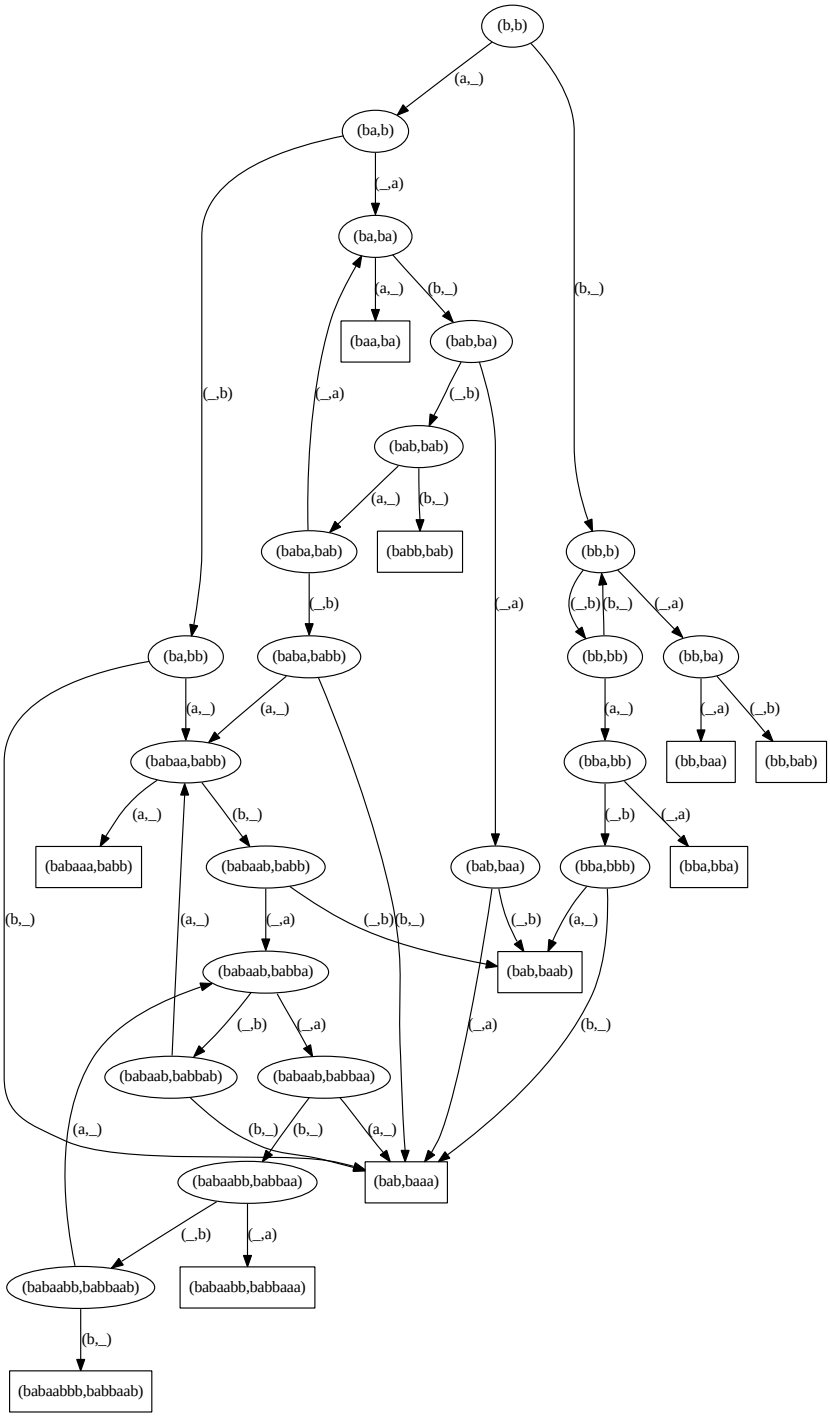


Figure B.4 Le graphe des chevauchements obtenu pour $|\varphi(a)| = 5$ et $|\varphi(b)| = 15$ et un décalage de 3. Remarquez que ce graphe est identique à celui de la Figure B.3.

APPENDICE C

PROGRAMMES POUR LA COMBINATOIRE DES MOTS

C.1 Classe Partition

```
class Partition:

    def __init__(self, ens):
        """
        Construction de la partition triviale ou chaque element
            est une classe
        """
        self.classeDe = {}

        for element in set(ens):
            self.classeDe[element] = [element]

    def __repr__(self):
        """
        p.__repr__() <==> repr(p)

        Retourne la partition en str (pour affichage)
        """
        #print "essait"
        return repr(self.classeDe.values())

    def representantDe(self, a):
        """
        p.representantDe(a) -> element

        Retourne le representant de a
        """
        for representant in self.classeDe:
            if a in self.classeDe[representant]:
                return representant

        raise TypeError, "L'element "+repr(a)+" n'est pas dans
           la partition "+repr(self)
```

```
    def possede(self, a):
        """
        p.possede(a) -> True of False

        Retourne VRAI si l'element a est dans la partition
        """
        for representant in self.classeDe:
            if a in self.classeDe[representant]:
                return True

        return False

    def ajoute(self, a):
        """
        p.ajoute(a) -> None

        Ajoute l'element a dans la partition
        """
        assert not self.possede(a)

        self.classeDe[a] = [a]

    def joindre(self, a, b):
        """
        p.joindre(a, b) -> None

        Joindre les classes des elements a et b
        """
        if not self.possede(a):
            self.ajoute(a)
        if not self.possede(b):
            self.ajoute(b)

        repA = self.representantDe(a)
        repB = self.representantDe(b)

        if repA == repB:
            return

        self.classeDe[repA] += self.classeDe[repB]
        del self.classeDe[repB]

    def nbClasse(self):
        """
        p.nbClasse() -> entier

        Retourne le nombre de classes
        """
        return len(self.classeDe)
```

C.2 Classe Mot

```
from partition import Partition

class Mot(str):

    def __init__(self,s):
        """
        s = "abcde"
        Mot(s) -> Mot

        Construit le Mot a partir de lobjet str s
        """
        str.__init__(s)

        self.strFactGD = None        #Une factorisation en str
            de gauche a droite

    def __add__(self, autre):
        """
        w.__add__(y) <==> w+y -> Mot

        Retourne la concatenation des mots w et y
        """
        return Mot(str.__add__(self, autre))

    def __radd__(self, autre):
        """
        w.__radd__(y) <==> y+w -> Mot

        Retourne la concatenation des mots y et w
        """
        return Mot(str.__add__(autre, self))

    def __getitem__(self, i):
        """
        w.__getslice__(i) <==> w[i] -> Mot

        Retourne la lettre a l'indice i.
        """
        return Mot(str.__getitem__(self,i))

    def __getslice__(self, i, j):
        """
        w.__getslice__(i, j) <==> w[i:j] -> Mot

        Retourne le facteur de longueur (j-i) commencant a l'
            indice i.
        """
        return Mot(str.__getslice__(self,i,j))

    def __pow__(self, exp):
```

```
        """
        w.__pow__(exp) <==> w**exp -> Mot

        Retourne la puissance w**exp.
        Attention, il faut que |w|exp soit entier.
        """
        nbCaracteres = len(self)*exp

        #Si exp*|self| n'est pas entier
        if nbCaracteres % 1 != 0 :
            raise ValueError, "Puissance non definie sur l'
               exposant "+str(exp)

        #Si exp n'est pas positif
        elif exp < 0 :
            raise ValueError, "L'exposant doit etre positif."

        #Si exp n'est pas entier
        elif int(exp) != exp :
            return ( self ** int(exp) ) + self[:int(
               nbCaracteres)%len(self)]

        #Base de recurence
        elif exp == 0 :
            return Mot('')

        else :
            return self + self ** (exp-1)

    def miroir(self):
        """
        w.miroir() -> Mot

        Retourne le miroir du mot (sans changer w)
        """
        nv = Mot('')
        for lettre in self:
            nv = lettre + nv
        return nv

    def conjugue(self, debut=1):
        """
        w.conjugue([debut]) -> Mot

        Retourne le conjugue du mot w.
        Le parametre debut est optionel : par defaut, debut=1.
        Ex : 'abcde'.conjugue(3) -> 'deabc'
        """
        return self[debut:]+self[:debut]

    def prefixeCommun(self, autre):
        """
```

```
        w.prefixeCommun(autre) -> Mot

        Retourne le plus grand prefixe commun de self et autre
        """
        i=0
        while(i < min(len(self),len(autre)) and self[i] ==
            autre[i]):
            i += 1
        return self[:i]

    def suffixeCommun(self, autre):
        """
        w.suffixeCommun(autre) -> Mot

        Retourne le plus grand suffixe commun de self et autre
        """
        i=0
        while(i < min(len(self),len(autre)) and self[-(i+1)] ==
            autre[-(i+1)]):
            i += 1
        if i == 0:
            return Mot('')
        return self[-i:]

    def estPrefixeDe(self, autre):
        """
        w.estPrefixeDe(autre) -> True or False

        vrai si w est prefixe de autre
        """
        return self == autre[:len(self)]

    def estSuffixeDe(self, autre):
        """
        w.estSuffixeDe(autre) -> True or False

        vrai si w est suffixe de autre
        """
        if len(self)==0:
            return True
        else:
            return self == autre[-len(self):]

    def estFacteurDe(self, autre):
        """
        w.estFacteurDe(autre) -> True or False

        vrai si w est facteur de autre
        """
        return autre.find(self) != -1

    def estPalindrome(self):
```

```
        """
        w.estPalindrome() -> True or False

        vrai si w est un palindrome
        """
        for i in range((len(self)+1)/2):
            if self[i] != self[-(i+1)]:
                return False
        return True

    def estFPalindrome(self,involution=None):
        """
        w.estFPalindrome([involution]) -> True or False

        Entree : involution, une involution sur l'alphabet
        Retourne vrai si w est un f-palindrome (f=involution)

        Exemples:
        sage: f={'a':'b','b':'a'}
        sage: Mot('abba').estFPalindrome(involution=None)
        True
        sage: Mot('abba').estFPalindrome(involution=f)
        False
        sage: Mot('abab').estFPalindrome(involution=f)
        True
        sage: Mot('ab').estFPalindrome(f)
        True
        sage: Mot('a').estFPalindrome(f)
        False
        sage: Mot('').estFPalindrome(involution=f)
        True
        sage: Mot('').estFPalindrome(involution=None)
        True
        """
        if involution is None:
            return self.estPalindrome()

        for i in range((len(self)+1)/2):
            if self[i] != involution[self[-(i+1)]]:
                return False
        return True

    def chevauche(self, autre, decalage, p=None):
        """
        w.chevauche(autre, decalage) -> Partition

        decalage = +3
        cheval
           abcdef

        Creation des classes d'equivalence (partition) des
           lettres pour que
```

```
        le chevauchement donne par le decalage ait lieu
        """
        #Si decalage negatif
        if decalage < 0:
            return Mot(autre).chevauche(self,-decalage)

        #Partition par defaut
        if p == None:
            p = Partition(list(self+autre))

        for i in range(decalage,min(len(autre)+decalage,len(
            self))):
            p.joindre(self[i], autre[i-decalage])

        #print ''.ljust(-decalage)+self
        #print ''.ljust(decalage)+autre

        return p

    def representation(self, partition):
        """
        w.representation(partition) -> Mot

        Retourne le mot obtenu en remplacant chaque caractere
        de w par son representant dans la partition ou par lui-
            meme
        s'il n'est pas dans la partition.
        """
        rep=Mot('')
        for c in self:
            if partition.possede(c):
                rep += partition.representantDe(c)
            else:
                rep += c
        return rep

    def permutationCanonique(self):
        """
        w.permutationCanonique() -> dictionnaire dun Morphisme

        Retourne la permutation permettant d'ecrire le mot w
            sous sa forme canonique.
        Ex : abc -> abc, xyz -> abc, wwyhgw -> aabcda
        """
        caracteres = 'abcdefghijklmnopqrstuvwxyz0123456789'
        caracteres += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        d={}
        i = 0

        for c in self:
            if c not in d:
                assert i<len(caracteres)
```

```
                d[c] = caracteres[i]
                i+=1

        return d

    def __factorise__(self, morphisme, pos=0):
        """
        w.__factorise__(morphisme, pos) -> Liste d'indices

        morphisme       dictionnaire dont les images sont des
            facteurs
        pos             position relative de self dans le mot

        Retourne un dictionnaire d'indices correspondant aux
            debuts
        de chacun des facteurs composant le mot w.
        Chaque indice pointe vers son etiquette dans le
            dictionnaire.
        La solution n'est pas necessairement unique.
        Retourne None si la factorisation de w en les facteurs
        est impossible.
        Note : La factorisation doit commencer par un facteur.
        """

        for cle in morphisme:
            bloc = morphisme[cle]

            if len(bloc) == 0:
                continue

            p = self.prefixeCommun(bloc)

            #Si le bloc termine exactement le mot self
            if len(p) == len(self) and len(p) == len(bloc):
                rep = {}
                rep[pos]=cle
                rep[pos+len(p)]=None
                return rep

            #Si un prefixe du bloc termine exactement le mot
                self
            elif len(p) == len(self):
                rep = {}
                rep[pos]=cle
                return rep

            #Si le bloc est un prefixe de self
            elif len(p) == len(bloc):
                rep = self[len(bloc):].__factorise__(morphisme,
                    pos+len(bloc))
                if rep == None :
                    continue
```

```
                else:
                    assert pos not in rep
                    rep[pos] = cle
                    return rep

        #S'il n'y a pas de solution
        return None

    def factorise(self, morphisme):
        """
        w.factorise(morphisme) -> Liste d'indices

        morphisme      dictionnaire dont les images sont des
            facteurs

        Retourne un dictionnaire d'indices correspondant aux
            debuts
        de chacun des facteurs composant le mot w.
        Chaque indice pointe vers son etiquette dans le
            dictionnaire.
        La solution n'est pas necessairement unique. ----> A
            AMELIORER
        Retourne None si la factorisation de w en les facteurs
        est impossible.
        Note : La factorisation peut commencer par un suffixe d
            'un facteur.
        """

        #Calculer la longueur du plus grand facteur
        longueurPGF = 0
        for f in morphisme.values():
            if len(f) > longueurPGF:
                longueurPGF = len(f)

        #Boucle sur les debuts possibles
        for debut in range(longueurPGF):

            #Chercher un facteur tronque au debut de self
            for cle in morphisme:
                f = morphisme[cle]
                if self[:debut].estSuffixeDe(f):

                    #Factoriser le reste
                    rep = self[debut:].__factorise__(morphisme,
                        debut)

                    #Retourner la factorisation si elle a
                        reussi
                    if rep != None:
                        return rep

        #S'il n'y a pas de solution
```

```
        return None

    def strFactorise(self, morphisme):
        """
        w.strFactorise(morphisme) -> Str

        morphisme      dictionnaire dont les images sont des
            facteurs

        Retourne la factorisation en str ou '[' est un debut de
            facteur
                                                et ']' est une fin
                                                   de facteur
        Ex: 'abbabaab' sur 'ab','ba' ->  '[][][][]'
        """

        #Calcule le dictionnaire de la factorisation
        #d[position du debut d'un bloc] = etiquette de ce bloc
        d = self.factorise(morphisme)

        #S'il n'y a pas de factorisation
        if d == None:
            print "Factorisation impossible"
            return
        elif len(d)==0:
            print "Factorisation vide"
            return

        l=d.keys()
        l.sort()

        s=''
        if l[0] > 0 :
            s += ']'.rjust(l[0])

        for i in range(len(l)-1):
            posG = l[i]
            posD = l[i+1]
            longueur = posD-posG
            assert longueur >= 0
            if longueur == 0:
                continue
            elif longueur == 1:
                s += '['
            elif longueur == 2:
                s += '[]'
            else:
                etiquette = d[posG].upper()
                assert len(etiquette) == 1
                s += '[' + etiquette.center(longueur-2) + ']'

        s += '['
```

```
        self.strFactGD = s.ljust(len(self))[:len(self)]

        return self.strFactGD

    def strFactoriseMiroir(self, morphisme):
        """
        w.strFactorise(morphisme) -> Str

        morphisme      dictionnaire dont les images sont des
            facteurs

        Retourne la factorisation miroir en str ou '[' est un
            debut de facteur
                                                et ']' est une fin
                                                    de facteur
        """
        if self.strFactGD == None:
            self.strFactorise(morphisme)

        rep = Mot(self.strFactGD).miroir()

        assert rep.find('*') == -1
        rep = rep.replace('[','*')
        rep = rep.replace(']','[')
        rep = rep.replace('*',']')
        rep = rep.replace(' B','B~')
        rep = rep.replace(' A','A~')

        return rep

    def pgPalSuf(self, palPrec = None, involution=None):
        """
        w.pgPalSuf([palPrec, involution]) -> Mot

        Entrees :
        palPrec = plus grand palindrome suffixe de w[:-1] (si
            connu).
        involution = une involution sur l'alphabet

        Retourne le plus grand palindrome suffixe de w.

        Exemples :

        sage: f={'a':'b','b':'a'}
        sage: Mot('abbb').pgPalSuf()
        'bbb'
        sage: Mot('abbbab').pgPalSuf()
        'bab'
        sage: Mot('abbbab').pgPalSuf(None,involution=f)
        'ab'
        sage: Mot('ababab').pgPalSuf(None,involution=f)
```

```
        'ababab'
        sage: Mot('babab').pgPalSuf(None,involution=f)
        'abab'
        sage: Mot('ab').pgPalSuf(None,involution=f)
        'ab'
        sage: Mot('ab').pgPalSuf('',involution=f)
        'ab'

        """
        f=involution
        #Si le plus grand palindrome suffixe precedent n'est
            pas donne
        if palPrec == None :
            for i in range(len(self)+1):
                fact = self[i:]
                if fact.estFPalindrome(involution=f):
                    return fact

        #Si le plus grand palindrome suffixe precedent est
            donne
        else:
            l=len(palPrec)
            #Verification que palPrec est bien le PGPS de w
                [:-1]
            #assert len(self)-1 >= l
            #assert palPrec == self[-l-1:-1]
            #assert palPrec.estPalindrome()

            #S'il n'y a pas de lettre a gauche de palPrec dans
                w, tout recalculer
            if len(self) < l+2:
                return self.pgPalSuf(palPrec=None, involution=f
                    )

            #Calculer la lettre a gauche (g) et a droite (d) de
                palPrec dans w
            g=self[-l-2]
            d=self[-1]
            gd=g+d

            #Si le mot g+d forme un palindrome, le resultat
                suit
            if gd.estFPalindrome(involution=f):
                return self[-l-2:]

            #Sinon, chercher parmi les suffixes plus petits que
                l + 2
            else:
                return self[len(self)-l-1:].pgPalSuf(palPrec=
                    None, involution=f)
```

```
    def delta(self):
        """
        w.delta() -> Mot

        Retourne l'image par l'operateur Delta du mot w. C'est-
            a-dire, le nombre de lettres egales
        consecutives. Exemple : si w = abbbaabba, alors w.delta
            () = 13221.
        """
        rep = Mot('')
        if len(self) == 0:
            return rep

        i = 0
        lettreBase = self[i]
        exposant = 0

        while i < len(self):

            #Si on change de lettre
            if self[i] != lettreBase :
                if exposant <=9:
                    rep += str(exposant)
                else:
                    rep += '.'+str(exposant)+'.'
                exposant = 0
                lettreBase = self[i]

            exposant += 1
            i += 1

        #Ajouter le mot de la fin
        if exposant <=9:
            rep += str(exposant)
        else:
            rep += '.'+str(exposant)+'.'

        return rep
```

C.3 Classe Algo Defaut

```
from mot import Mot

class algo_defaut:

    def __init__(self, mot, involution = None):
        """
        Initialisation de l'algorithme

        Exemples :
```

```
sage: a=algo_defaut('abcde')
sage: a

Mot                     : abcde
Longueur du mot         : 5
Involution f            : None
Nombre de palindromes   : 6
Defaut                  : 0
Lacunes                 : []
Tableau de Brlek        : [0, 1, 1, 1, 1, 1]
Tableau des LPS         : [0, 1, 1, 1, 1, 1]
sage: a.palindromes
set(['', 'a', 'c', 'b', 'e', 'd'])
sage: f={'a':'b','b':'a'}
sage: w=Mot('abbabaabbaab')
sage: algo_defaut(w)

Mot                     : abbabaabbaab
Longueur du mot         : 12
Involution f            : None
Nombre de palindromes   : 11
Defaut                  : 2
Lacunes                 : [9, 10]
Tableau de Brlek        : [0, 1, 1, 2, 4, 3, 3, 2, 4,
   None, None, 6, 8]
Tableau des LPS         : [0, 1, 1, 2, 4, 3, 3, 2, 4, 2,
   4, 6, 8]
Palindromes             : set(['', 'a', 'aa', 'b', '
   baabbaab', 'bb', 'abba', 'aba', 'bab', 'aabbaa', '
   baab'])
sage: algo_defaut(w,f)

Mot                     : abbabaabbaab
Longueur du mot         : 12
Involution f            : {'a': 'b', 'b': 'a'}
Nombre de palindromes   : 10
Defaut                  : 3
Lacunes                 : [1, 3, 5]
Tableau de Brlek        : [0, None, 2, None, 2, None, 4,
   6, 8, 4, 6, 4, 6]
Tableau des LPS         : [0, 0, 2, 0, 2, 2, 4, 6, 8, 4,
   6, 4, 6]
Palindromes             : set(['', 'baba', 'bbaa', 'ab',
   'ba', 'aabb', 'bbabaa', 'baabba', 'abbaab', '
   abbabaab'])

"""

self.mot = Mot(mot)
self.involution = involution
self.palindromes = set()
```

```
        self.nbPal = None
        self.defaut = None
        self.tabBrlek    = []
        self.tabLPS      = []
        self.lacunes     = []
        self.posDernPal = None #position de la fin du dernier
            palindrome

        self.trouvePalindromes()

    def __eq__(self, autre):
        """
        a.__eq__(autre) <==> a==autre -> True or False

        Retourne True si les deux algos ont les memes
            parametres
        """
        #Coder la robustesse : Verifier que autre est un
            algo_defaut
        return self.mot == autre.mot and \
               self.involution == autre.involution

    def __repr__(self):
        """
        a.__repr__() <==> a -> str

        Retourne les resultats de l'algorithme en str (pour
            affichage)
        """
        MAX=30

        rep = ''
        rep +=    'Mot                    : ' + self.mot[:MAX]
        rep += '\nLongueur du mot         : ' + str(len(self.mot)
            )
        rep += '\nInvolution f            : ' + str(self.
            involution)
        rep += '\nNombre de palindromes   : ' + str(self.nbPal)
        rep += '\nDefaut                  : ' + str(self.defaut)
        rep += '\nLacunes                 : ' + str(self.lacunes
            [:MAX])
        rep += '\nTableau de Brlek        : ' + str(self.tabBrlek
            [:MAX])
        rep += '\nTableau des LPS         : ' + str(self.tabLPS[:
            MAX])
        #rep += '\nPalindromes             : ' + str(self.
            palindromes)

        return rep


    def trouvePalindromes(self):
```

```
        """
        w.trouvePalindromes() -> None

        Calcule les palindromes distincts de w              ->
            w.palindromes
        Calcule le tableau des defauts de w                 ->
            w.tabDefaut
        Calcule la pos. de la fin du dernier palindrome de w ->
            w.posDernPal
        Calcule la position du premier defaut de w          ->
            w.posPremDef
        """

        #Initialise le palindrome precedent a None
        pal = None

        #Pour tous les prefixes de self.mot
        for i in range(len(self.mot)+1):
            prefixe = self.mot[:i]
            #Calculer le plus grand pal. suffixe (en utilisant
                le PGPS precedent)
            pal  = prefixe.pgPalSuf(pal, self.involution)
            self.tabLPS.append(len(pal))
            if pal in self.palindromes:
                #Si pal n'est pas un nouveau palindromes
                self.tabBrlek.append(None)
                self.lacunes.append(i)
            else :
                #S pal est un nouveau palindrome
                self.palindromes.add(pal)
                self.tabBrlek.append(len(pal))
                self.posDernPal = i

        self.nbPal = len(self.palindromes)
        self.defaut = len(self.mot)+1-self.nbPal
```

C.4 Classe Morphisme

```
from mot import Mot
from algo_defaut import algo_defaut

class Morphisme:
    def __init__(self, s):
        """
        s = "a->ab,b->ba"
        Morphisme(s) -> Morphisme

        Construit le morphisme represente par s
        """
        self.morph  = {}
```

```
        self.permCano = None #Sauvegarde de la Permutation
            canonique
        self.__construction__(s)
        #assert self.estEndomorphisme()

    def __construction__(self, s):
        """
        Construction du Morphisme a partir du str s
        Exemple : s = "a->ab,b->ba"
        Verifie que s est bien definie
        NE DOIT PAS ETRE UTILISEE PAR LUTILISATEUR
        """
        for fleche in s.split(','):
            if len(fleche) == 0:
                continue

            if len(fleche) < 3 or fleche[1:3] != '->':
                raise TypeError, "Le 2e et 3e caractere de la
                    fleche doit etre '->'."

            lettre = Mot(fleche[0])
            image  = Mot(fleche[3:])

            if self.morph.has_key(lettre):
                raise TypeError, "L'image de la lettre "+lettre
                    +"est definie deux fois."

            self.morph[lettre] = image

    def __repr__(self):
        """
        m.__repr__() <==> m -> str

        Retourne le morphisme en str (pour affichage)
        """
        s = ''
        for lettre,image in self.morph.items() :
            s += lettre + '->' + image + ','
        return s[:-1] #pour enlever la derniere virgule

    def __getitem__(self, w):
        """
        m.__getitem__(w) <==> m[w] -> Mot

        Retourne limage de w par le morphisme m
        """
        rep = Mot('')
        for lettre in w:
            if not self.morph.has_key(lettre):
                raise TypeError, "L'image de la lettre " +
                    lettre + " n'est pas definie."
            rep += self.morph[lettre]
```

```
        return rep

    def __mul__(self, autre):
        """
        m.__mul__(autre) <==> m*autre -> Morphisme

        Retourne le morphisme m*autre
        """
        nv = Morphisme('')
        for lettre in autre.morph.keys():
            nv.morph[lettre]=self[autre.morph[lettre]]
        return nv

    def __pow__(self, exp):
        """
        m.__pow__(exp) <==> m**exp -> Morphisme

        Retourne le morphisme puissance m**exp.
        """
        #Si exp n'est pas positif
        if exp <= 0 :
            raise ValueError, "L'exposant doit etre strictement
                positif."

        #Si exp n'est pas entier
        elif int(exp) != exp :
            raise ValueError, "L'exposant doit etre entier."

        #Base de recurence
        elif exp == 1 :
            return self

        else :
            return self * self ** (exp-1)


    def __eq__(self, autre):
        """
        m.__eq__(autre) <==> m==autre -> True or False

        Retourne True si les deux morphismes sont egaux
        """
        #Coder la robustesse : Verifier que autre est un
            morphisme
        if autre == None:
            return False
        else:
            return self.morph == autre.morph

    def permutationCanonique(self):
        """
        m.permutationCanonique(self) -> Morphisme
```

```
        Retourne la permutation permettant d'ecrire le
           morphisme sous sa forme canonique.
        Ex : 'a->lkkl,b->hjkllk' -> 'a->abba,b->cdbaab'
        """
        #Si n'a pas deja ete calcule
        if self.permCano == None :
            l = self.morph.keys()
            l.sort()

            #Mettre dans w la concatenation de toutes les
               images
            w=Mot('')
            for c in l:
                w += self[c]

            #Sauvegarder la permutation canonique
            self.permCano = Morphisme('')
            self.permCano.morph = w.permutationCanonique()


        return self.permCano

    def estIsomorphe(self,autre):
        """
        m.estIsomorphe(autre) -> True or False

        Retourne True si les deux morphismes sont isomorphes.
        Soient A, S, T et U des alphabets.
        Deux morphismes m1 : A* -> S*, m2 : A* -> T* sont
           isomorphes s'il existe deux
        permutations p1 : S -> U, p2 : T -> U telle que p1 * m1
            = p2 * m2.
        """
        return self.permutationCanonique()*self == autre.
           permutationCanonique()*autre

    def domaine(self):
        """
        m.domaine() -> set(Mots)

        Retourne l'ensemble de toutes les lettres pour
           lesquelles
        l'image par le morphisme est definie : le domaine.
        """
        return set(self.morph.keys())

    def codomaine(self):
        """
        m.codomaine() -> set(Mots)
```

```
        Retourne l'ensemble de toutes les lettres qui
           apparaissent
        dans les images du morphisme : le codomaine.
        """
        rep=set()
        for image in self.images():
            rep.update(list(image))
        return rep

    def estEndomorphisme(self):
        """
        m.estEndomorphisme() -> True or False

        Retourne vrai si l'alphabet du codomaine est
        un sous-ensemble de celui du domaine.
        """
        return self.codomaine() <= self.domaine()

    def images(self):
        """
        m.images() -> Liste de Mots

        Retourne la liste de toutes les images des lettres
        """
        return self.morph.values()

    def miroir(self):
        """
        m.miroir() -> Morphisme

        Retourne le morphisme dont l'image des lettres
        est lue de droite a gauche et ou m(uv) = m(u)m(v) quand
            meme.
        Rmq : m n'est pas change
        """
        nv = Morphisme('')
        for lettre in self.domaine():
            nv.morph[lettre]=self[lettre].miroir()
        return nv

    def classe(self):
        """
        m.classe() -> Liste de morphismes

        Retourne une liste des morphismes de sa classe
        En appliquant les operations [m.miroir(),e*m*e,e*m]
        et leurs compositions.
        """

        #e=Morphisme('a->b,b->a,c->c')
        e=Morphisme('a->b,b->a')
        miroir = self.miroir()
```

```
        SelfE = self*e
        miroirE = miroir*e

        #Construire la liste c des morphismes de la classes
        c = []
        c.append(self)
        c.append(miroir)
        c.append(e*SelfE)
        c.append(e*miroirE)
        c.append(SelfE)
        c.append(miroirE)
        c.append(e*self)
        c.append(e*miroir)

        #Construire la liste d des morphismes distincts
        d = []
        for m in c:
            if m not in d:
                d.append(m)
        return d

    def petiteClasse(self):
        """
        m.petiteClasse() -> Liste de morphismes

        Retourne une liste [m,m*e] des morphismes
        de sa classe.
        """

        #e=Morphisme('a->b,b->a,c->c')
        e=Morphisme('a->b,b->a')
        SelfE = self*e

        #Construire la liste c des morphismes de la classes
        c = []
        c.append(self)
        c.append(SelfE)

        #Construire la liste d des morphismes distincts
        d = []
        for m in c:
            if m not in d:
                d.append(m)
        return d

    def estEffacant(self):
        """
        m.estEffacant() -> True or False

        Retourne vrai si la longueur dune des images du
           morphisme est zero.
        En anglais : an erasing morphism.
```

```
        """
        for image in self.images():
            if len(image) == 0:
                return True
        return False

    def lettreCommuneGauche(self):
        """
        m.lettreCommuneGauche() -> True or False

        Retourne vrai si toutes les images commencent par la
           meme lettre.
        """
        #Si le morphisme est vide
        if len(self.images()) == 0:
            return False

        #Si le morphisme est effacant
        if self.estEffacant():
            return False

        #Comparer les premieres lettres de toutes les images
        lettre = self.images()[0][0]
        for image in self.images()[1:] :
            if image[0] != lettre :
                return False

        return True

    def lettreCommuneDroite(self):
        """
        m.lettreCommuneDroite() -> True or False

        Retourne vrai si toutes les images terminent par la
           meme lettre.
        """
        return self.miroir().lettreCommuneGauche()

    def conjugue(self, debut=1):
        """
        m.conjugue([debut]) -> Morphisme

        Retourne le morphisme dont toutes les images ont etes
           conjuguees.
        Le parametre debut est optionel : par defaut, debut=1.
        Rmq : m n'est pas change.
        Ex : 'a->abcde'.conjugue(3) -> 'a->deabc'
        """
        nv = Morphisme('')
        for lettre in self.domaine():
            nv.morph[lettre]=self[lettre].conjugue(debut)
        return nv
```

```
    def listeConjugues(self):
        """
        m.listeConjugues() -> Liste de morphismes

        Retourne une liste des morphismes conjugues de m
        obtenus en conjuguant les prefixes (et suffixes)
           communs
        de chacune des images.
        """
        #Si le morphisme est vide
        if len(self.images()) == 0:
            return [self]

        #Construire la liste c des morphismes conjugues
        c = []
        m = self
        c.append(m)
        while(m.lettreCommuneGauche()):
            m=m.conjugue(1)
            if m==self:
                break
            c.append(m)
        m = self
        while(m.lettreCommuneDroite()):
            m=m.conjugue(-1)
            if m==self:
                break
            c.insert(0,m)

        #Construire la liste d des morphismes distincts
        d = []
        for m in c:
            if m not in d:
                d.append(m)
        return d


    def estPtFixeEn(self, w):
        """
        m.estPtFixeEn(w) -> True or False

        Retourne True si w est un prefixe de son image sous le
           morphisme m.
        """
        return Mot(w).estPrefixeDe(self[w])

    def printEstPtFixeEn(self, w):
        """
        m.printEstPtFixeEn(w) -> str

        Retourne 'pFx' si w est un prefixe de m[w]
```

```
        Retourne '' sinon
        """
        rep = ''
        if self.estPtFixeEn(w):
            rep += 'pFx'
        #else:
        #    rep += 'non'
        return rep

    def __call__(self, w, ordre=1):
        """
        m.__call__(w, ordre) -> Mot

        Retourne la ordre-ieme iteration du morphisme sur le
           mot w
        """
        assert (ordre >= 0)
        if ordre == 0:
            return Mot(w)
        if ordre == 1:
            return self[w]
        else :
            return self.__call__(self[w], ordre-1)


    def calculePointFixe(self, w, longueurMax):
        """
        m.calculePointFixe(w,longueurMax) -> Mot

        Retourne le prefixe du point fixe en w de longueur =
           longueurMax
        """
        image = self[w]
        if Mot(w).estPrefixeDe(image):
            s = image
            i = len(w)
            while (i < len(s) and len(s) < longueurMax):
                s += self.morph[s[i]]
                i = i+1
            if i == len(s) :
                assert(self[s] == s) #Point fixe fini!
            return s
        else :
            raise TypeError, w + " n'est pas un point fixe."

    def liste_nombre_pal(self, iterations=4, lettreInitiale='a'
        , involution=None):
        """
        Retourne une liste dont le i-eme terme est le nombre de
            f-palindromes
        contenu dans la i-eme iteration du morphisme sur la
           lettre initiale donnee.
```

```
        """
        return [algo_defaut(self(lettreInitiale, i), involution
            =involution).nbPal for i in range(iterations)]

    def __etudePal__(self, ordre, involution=None,
        lettreInitiale = 'a'):
        """
        m.__etudePal__(ordre[, involution, lettreInitiale]) ->
            string

        Genere les palindromes obtenus par l'iteration du
            morphisme
        sur la lettreInitiale.
        Decris la factorisation en blocs des palindromes
            maximaux.
        """

        #Calculer l'ensemble des palindromes maximaux generes
            par le morphisme
        mot = self(lettreInitiale, ordre)
        a=algo_defaut(mot,involution=involution)
        ensPalMax=elementsMax(a.palindromes)

        #Ecrire les parametres
        rep = ''
        rep += ('Morphisme :        ' + str(self)).ljust(50)
        rep += 'Longueur du mot :        ' + str(len(mot)) + '\n
            '
        rep += ('Lettre initiale : ' + lettreInitiale).ljust
            (50)
        rep += 'Nombre de palindromes : ' + str(a.nbPal) + '\n'
        rep += ('Ordre :            ' + str(ordre)).ljust(50)
        rep += 'Defaut : ' + str(a.defaut) + '\n'
        rep += 'Nombre de palindromes maximaux : ' + str(len(
            ensPalMax)) + '\n'

        #Factoriser chacun des palindromes en bloc-images du
            morphisme
        for p in ensPalMax:
            rep += str(p) + '\n'
            rep += str(p.strFactorise(self.morph)) + '\n'
            rep += str(p.strFactoriseMiroir(self.morph)) + '\n'
                + '\n'

        return rep

    def etudePal(self, ordre, involution=None, lettreInitiale =
        'a'):
        """
        m.etudePal(ordre[, involution,lettreInitiale]) -> None
```

```
        Genere les palindromes obtenus par l'iteration du
           morphisme
        sur la lettreInitiale.
        Decris la factorisation en blocs des palindromes
           maximaux.
        (Le resultat est imprime.)

        Exemple :

        sage: m=Morphisme('a->ab,b->ba')
        sage: m.etudePal(4)
        Morphisme :         a->ab,b->ba
           Longueur du mot :        16
        Lettre initiale : a
           Nombre de palindromes : 15
        Ordre :             4
           Defaut : 2
        Nombre de palindromes maximaux : 1
        abbabaabbaababba
        [][][][][][][][][][]
        [][][][][][][][][][]

        sage: f={'a':'b','b':'a'}
        sage: m.etudePal(5,involution=f)
        Morphisme :         a->ab,b->ba
           Longueur du mot :        32
        Lettre initiale : a
           Nombre de palindromes : 25
        Ordre :             5
           Defaut : 8
        Nombre de palindromes maximaux : 1
        abbabaabbaababbabaabababbaabbabaab
        [][][][][][][][][][][][][][][][][][][][][][][][][][]
        [][][][][][][][][][][][][][][][][][][][][][][][][][]


        """
        print self.__etudePal__(ordre, involution=involution,\
                                        lettreInitiale=
                                           lettreInitiale)

    def etudeDef(self, ordre, involution=None, lettreInitiale =
        'a'):
        """
        m.etudeDef(ordre[,involution, lettreInitiale]) -> None

        Affiche le defaut et les lacunes du mot obtenu par l'
           iteration du morphisme
        sur la lettreInitiale.
        (Le resultat est imprime.)

        Exemple :
```

```
sage: m=Morphisme('a->ab,b->ba')
sage: m.etudeDef(3)
Morphisme :         a->ab,b->ba
Lettre initiale : a
Ordre :             3
Mot                         : abbabaab
Longueur du mot             : 8
Involution f                : None
Nombre de palindromes : 9
Defaut                      : 0
Lacunes                     : []
Tableau de Brlek            : [0, 1, 1, 2, 4, 3, 3, 2, 4]
Tableau des LPS             : [0, 1, 1, 2, 4, 3, 3, 2, 4]
Palindromes                 : set(['', 'a', 'aa', 'b', 'bb',
    'abba', 'aba', 'bab', 'baab'])

sage: f={'a':'b','b':'a'}
sage: m.etudeDef(3,involution=f)
Morphisme :         a->ab,b->ba
Lettre initiale : a
Ordre :             3

Mot                         : abbabaab
Longueur du mot             : 8
Involution f                : {'a': 'b', 'b': 'a'}
Nombre de palindromes : 6
Defaut                      : 3
Lacunes                     : [1, 3, 5]
Tableau de Brlek            : [0, None, 2, None, 2, None, 4,
    6, 8]
Tableau des LPS             : [0, 0, 2, 0, 2, 2, 4, 6, 8]
Palindromes                 : set(['', 'baba', 'ab', 'ba', '
    bbabaa', 'abbabaab'])

"""

#Calculer l'ensemble des palindromes maximaux generes
    par le morphisme
mot = self(lettreInitiale, ordre)

#Ecrire les parametres
rep = ''
rep += ('Morphisme :        ' + str(self)) +'\n'#.ljust
    (50)
rep += ('Lettre initiale : ' + lettreInitiale) +'\n'#.
    ljust(50)
rep += ('Ordre :            ' + str(ordre))# +'\n'.ljust
    (50)
print rep

return algo_defaut(mot,involution=involution)
```

```
    def estClassePGauche(self, involution=None):
        """
        m.estClassePGauche([involution]) -> True or False

        Entrees :
        involution, une involution sur l'alphabet

        Retourne VRAI si m est de classe P avec un palindrome
        commun a gauche

        Exemples :

        sage: m=Morphisme('a->ab,b->ba')
        sage: m.estClassePGauche()
        False
        sage: f={'a':'b','b':'a'}
        sage: m.estClassePGauche(involution=f)
        True
        """
        #Si le morphisme est vide
        if len(self.morph) == 0:
            return True

        #Calculer le plus grand prefixe commun des images
        prefixe = self.images()[0]
        for image in self.images():
            prefixe = prefixe.prefixeCommun(image)

        #Chercher un palindrome prefixe commun
        for i in range(len(prefixe)+1):
            if prefixe[:i].estFPalindrome(involution=involution
                ):

                #Verifier que les suffixes sont tous
                    palindromes
                for image in self.images():
                    if not image[i:].estFPalindrome(involution=
                        involution):
                        break
                else:
                    return True

        return False

    def estClasseP(self, involution=None):
        """
        m.estClasseP() -> True or False

        Entrees :
        involution, une involution sur l'alphabet
```

```
        Retourne VRAI si m est de classe P

        Exemples :

        sage: m=Morphisme('a->ab,b->ba')
        sage: m.estClasseP()
        False
        sage: f={'a':'b','b':'a'}
        sage: m.estClasseP(involution=f)
        True
        sage: (m^2).estClasseP()
        True
        sage: (m^2).estClasseP(involution=f)
        False
        sage: fibo=Morphisme('a->ab,b->a')
        sage: fibo.estClasseP()
        True
        sage: fibo.estClasseP(involution=f)
        False
        sage: (fibo^2).estClasseP()
        False
        sage: (fibo^2).estClasseP(involution=f)
        False
        sage: (fibo^4).estClasseP()
        False

        """
        rep = self.estClassePGauche(involution=involution)
                    # or self.miroir().estClassePGauche()

        return rep

    def aConjugueClasseP(self, involution=None):
        """
        m.aConjugueClasseP() -> True or False

        Entrees :
        involution, une involution sur l'alphabet

        Retourne VRAI si m a un conjugue de classe P
        """
        for k in self.listeConjugues():
            if k.estClassePGauche(involution=involution):
                #print "conjugue de Classe P:", k
                return True
        return False


#Fonction externe de POSET
def elementsMax(ensemble):
```

```
    """
    ENTREE : ensemble
             #relation dordre partiel des elements de l'
                ensemble
    SORTIE : ensemble des elements maximaux selon la relation
       estFacteurDe
             definie dans la classe Mot
    """
    rep = set()

    for f in ensemble:
        for m in rep:
            #Si m est plus grand que
            if f.estFacteurDe(m):
                break
        else:
            aEnlever = set()
            for m in rep:
                #Si m est plus petit que
                if m.estFacteurDe(f):
                    aEnlever.add(m)
            rep.difference_update(aEnlever)
            rep.add(f)

    return rep
```

C.5 Classe Etat

```
from mot import Mot
from morphisme import Morphisme
import os

class Etat:

    def __init__(self, longA,longB, d, dessus='b', dessous='b')
       :
        """
        Construction de l'etat
        """
        #La premiere lettre des deux mots doit etre la meme
        assert dessus[0]==dessous[0],'Les deux mots doivent
           commencer par la meme lettre.'

        #Attributs de base :
        self.alphabet = ['a','b']
        self.longA = longA
        self.longB = longB
        self.decalage = d
        self.dessus = Mot(dessus)
        self.dessous = Mot(dessous)
```

```
        #Initialisation des attributs calculables.
        self.initialisation()

        #Calculer les attributs calculables
        self.calcule()


    def initialisation(self):
        """
        Initialisation des attributs calculables.
        """
        self.dessin        = None #Dessin de chevauchement
        self.morphisme     = None
        self.excedent      = None #Petit mot qui depasse du
            chevauchement
        self.longExcedent = None  #Positif si l'excedent est en
             haut
        self.estClassePP   = None #estClasseP'

    def calcule(self):

        caracteres = 'abcdefghijklmnopqrstuvwxyz0123456789'
        caracteres += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        assert self.longA + self.longB <= len(caracteres)
        A = Mot(caracteres[:self.longA])
        B = Mot(caracteres[self.longA : self.longA + self.longB
            ])
        m = Morphisme('a->'+A+',b->'+B)

        imageDessus = m[self.dessus]
        imageDessous = m.miroir()[self.dessous]

        #Calcul du dessin de chevauchement
        D1 = ''.ljust(-self.decalage)
        D2 = ''.ljust(self.decalage)

        L1 = D1 + imageDessus
        L2 = D1 + str(imageDessus.strFactorise(m.morph))
        L3 = D2 + str(imageDessous.strFactorise(m.miroir().
            morph))
        L3nv = L3.replace(' B','B~').replace(' A','A~')
        L4 = D2 + imageDessous

        self.dessin         =  L2 + '\\n'+ L3
        self.dessinComplet =  '\n' + L1 + '\n' + L2 + '\n'+
            L3nv +'\n' + L4  + '\n\n'

        #Calcul du morphisme
        p = imageDessus.chevauche(imageDessous, self.decalage)
        Anv = A.representation(p)
        Bnv = B.representation(p)
```

```
        self.morphisme = Morphisme('a->'+Anv+',b->'+Bnv)

        #Calcul de l'excedent
        imageDessusNV = self.morphisme[self.dessus]
        imageDessousNV = self.morphisme.miroir()[self.dessous]
        lh = len(imageDessusNV)
        lb = len(imageDessousNV)+self.decalage
        if lh > lb :
            self.excedent = imageDessusNV[lb-lh:]
        elif lh == lb :
            self.excedent = ''
        else:
            self.excedent = imageDessousNV[lh-lb:]
        self.longExcedent = lh-lb

    def nom(self):
        return self.dessus+'_'+self.dessous

    def etiquette(self, sorte='simple'):
        if sorte == 'simple':
            return self.dessous.miroir()[:-1]+'|'+self.dessus
                [0]+'|'+self.dessus[1:]
        elif sorte == 'dessin':
            return self.dessin
        elif sorte == 'couple':
            return '(%s,%s)' % (self.dessus, self.dessous)
        else:
            raise ValueError, "sorte d'etiquette inconnue"


    def __repr__(self):
        s = self.dessinComplet
        s += 'Morphisme      : '+ str(self.morphisme)+'\n'
        s += 'Est Classe P   : '+ str(self.morphisme.estClasseP
            ())+'\n'
        s += "Est Classe P' : " + str(self.estTerminal()) +'\n'
        s += "Extension      : Ext = " + self.excedent + ", ext
            = " + str(self.longExcedent)
        return s

    def __eq__(self, autre):
        """
        e.__eq__(autre) <==> e==autre -> True or False

        Retourne True si les deux etats sont egaux
        """
        excedent1 = self.morphisme.permutationCanonique()[self.
            excedent]
        excedent2 = autre.morphisme.permutationCanonique()[
            autre.excedent]

        rep = self.longA == autre.longA \
```

```
        and self.longB == autre.longB \
        and self.decalage == autre.decalage \
        and self.morphisme.estIsomorphe(autre.morphisme) \
        and self.longExcedent == autre.longExcedent \
        and excedent1 == excedent2

        ################test##############a enlever plus tard
        #if rep and not self.excedent == autre.excedent :
        #    print "voila un exemple"
        #    print self
        #    print autre
        ################test##############a enlever plus tard

        return rep


    def estTerminal(self):
        """
        e.estTerminal() -> True or False

        Retourne True si le morphisme de l'etat a un conjugue
           de classe P.
        """
        if self.estClassePP == None :
            self.estClassePP = self.morphisme.aConjugueClasseP
               ()
        return self.estClassePP

    def ajouteBloc(self,lettre):
        """
        Changer l'etat en ajoutant un bloc (le  bloc est l'
           image de la lettre).

        Retourne l'etiquette correspondant a l'arete cree.
        """
        s=''

        if self.longExcedent >= 0:
            self.dessous += lettre
            s += '(_,%s)' %lettre
        else :
            self.dessus += lettre
            s += '(%s,_)' %lettre

        #Reinitialisation
        self.initialisation()
        self.calcule()

        return s

    def enfants(self):
        """
```

```
        Retourne l'ensemble des enfants de l'etat par l'ajout d
            'un bloc.

        Sortie : Liste d'etats.
        """
        enfants = []
        for lettre in self.alphabet:
            nvEtat=Etat(self.longA, self.longB, self.decalage,
                self.dessus, self.dessous)
            nvEtat.ajouteBloc(lettre)
            enfants.append(nvEtat)
        return enfants

    def graphe(self):
        """
        Retourne le graphe des etats obtenus a partir de letat
            self.

        Entree : Listes de sommets et aretes deja generes.
        Sortie : Listes de sommets et aretes generes.
        """
        return self.__graphe__([self],[])

    def __graphe__(self, sommets, aretes):
        """
        Retourne le graphe des etats obtenus a partir de letat
            self.

        Entree : Listes de sommets et aretes deja generes.
        Sortie : Listes de sommets et aretes generes.
        (Un arete est un triplet (sommet1, sommet2, etiquette).
            )
        """

        if self.estTerminal():
            return sommets,aretes

        #Pour tous les enfants de self
        for lettre in self.alphabet:
            enfant=Etat(self.longA, self.longB, self.decalage,
                self.dessus, self.dessous)
            etiquette_arete = enfant.ajouteBloc(lettre)

            ##Si l'etat-enfant est terminal
            #if enfant.estTerminal():
            #    #Ajouter le sommet enfant et l'arete meme si l
                'enfant existe peut-etre deja
            #    sommets.append(enfant)
            #    aretes.append( (self,enfant,lettre) )
            #
            ##Si l'etat-enfant n'est pas terminal
            #else:
```

```
            #Si l'etat-enfant est deja existant dans les
                sommets crees
            for s in sommets:
                if enfant == s:
                    #Ajouter seulement l'arete
                    aretes.append( (self,s,etiquette_arete) )
                    break
            #Si l'etat-enfant n'est pas existant
            else:
                #Ajouter le sommet enfant et l'arete
                sommets.append(enfant)
                aretes.append( (self,enfant,etiquette_arete) )

                #Recursivite sur l'enfant
                sommets,aretes = enfant.__graphe__(sommets,
                    aretes)

        return sommets, aretes

    def creeFichierDot(self, AS='simple'):
        """
        Cree le sous-graphe des enfants de l'etat en format .
            dot.
        AS signifie affichageSimple
        Retourne le nom du fichier.
        """
        sommets,aretes = self.graphe()

        nomFichier = "graphe"+str(self.longA)+'_'+str(self.
            longB)+'_'+str(self.decalage)

        rep='digraph '+ nomFichier +' {\n'
        #Sommets...
        for s in sommets :
            rep += '    '+ s.nom() + ' [label="' + s.etiquette(
                AS)+'"'
            if s.estTerminal():
                rep += ',shape=box'
            rep += '];\n'
        #Aretes...
        for s1,s2,etiquette in aretes :
            rep += '    ' + s1.nom() + ' -> ' +  s2.nom()
            rep += ' [label="'+etiquette+'"];\n'
        rep += '}\n'

        dossier = "Graphes/"
        cheminFichier = dossier+nomFichier+".dot"
        f = open(cheminFichier,'w')
        f.write(rep)
        f.close()
        print "Resultats dans "+cheminFichier
```

```
        return dossier+nomFichier

    def creeArbreDirige(self,AS='couple'):
        """
        Cree le sous-graphe dirige des enfants de l'etat en
            format .ps.
        AS signifie affichageSimple
        """
        nomFichier=self.creeFichierDot(AS)
        os.system("dot -Tps2 "+nomFichier+".dot -o "+nomFichier
            +".ps")
        os.system("evince "+nomFichier+".ps")

    def creeArbre(self,AS='simple'):
        """
        Cree le sous-graphe non dirige des enfants de l'etat en
            format .ps.
        AS signifie affichageSimple
        """
        nomFichier=self.creeFichierDot(AS)
        os.system("neato -Tps2 "+nomFichier+".dot -o "+
            nomFichier+".ps")
        os.system("evince "+nomFichier+".ps")
```

RÉFÉRENCES

Allouche, J.P. 1997. «Schrödinger operators with Rudin-Shapiro potentials are not palindromic», *Journal of Mathematical Physics*, vol. 38, p. 1843–1848.

Allouche, J.P., M. Baake, J. Cassaigne et D. Damanik. 2003. «Palindrome complexity», *Theoretical Computer Science*, vol. 292, p. 9–31.

Allouche, J.P., et J. Shallit. 2000. «Sums of digits, overlaps, and palindromes», *Discrete Mathematics and Theoretical Computer Science*, vol. 4, p. 1–10.

Anne, V., L. Q. Zamboni et I. Zorca. 2005. «Palindromes and Pseudo-Palindromes in Episturmian and Pseudo-Palindromic Infinite Words». In S. Brlek, C. Reutenauer (éd.), *Words 2005*, Publications du LaCIM, vol. 36, p. 91–100.

Autebert, J.-M. 1994. *Théorie des langages et des automates*, Paris : Masson, 179 p.

Baake, M. 1999. «A note on palindromicity», *Letters in Mathematical Physics*, vol. 49, p. 217–227.

Bergeron, A., S. Brlek, A. Glen, S. Labbé et F. Saliola. 2008. sage-words : Combinatorics on words package for Sage, `sage-words.googlecode.com`.

Berstel, J., A. Lauve, C. Reutenauer et F. Saliola. 2008. «Combinatorics on words», In *CRM Proceedings & Lecture Notes*, Providence (RI) : American Mathematical Society, à paraître.

Blondin Massé, A. Communications personnelles, 2007.

Blondin Massé, A., S. Brlek, A. Frosini, S. Labbé et S. Rinaldi. 2008. «Reconstructing words from a fixed palindromic length sequence», In *Proc. TCS 2008*, 5th IFIP International Conference on Theoretical Computer Science (8–10 sept. 2008, Milan, Italie), p. 101–114.

Blondin Massé, A., S. Brlek, A. Glen et S. Labbé, 2007. «On the Critical Exponent of Generalized Thue-Morse Words», *Discrete Mathematics and Theoretical Computer Science*, vol. 9, p. 293–304.

Blondin Massé, A., S. Brlek et S. Labbé, 2008. «Palindromic lacunas of the Thue-Morse word», *Proc. GASCOM 2008* (16–20 juin 2008, Bibbiena, Arezzo-Italie), p. 53–67.

Blondin Massé, A., et S. Labbé, 2007. «A note on the critical exponent of generalized Thue-Morse words», In P. Arnoux, N. Bédaride, J. Cassaigne (éd.), *Proceedings of WORDS 2007*, p. 57–62.

Brlek, S. 1989. «Enumeration of the factors in the Thue-Morse word», *Discrete Appl. Math.*, vol. 24, p. 83–96.

Brlek, S. (dir.), A. Bergeron, A. Ladouceur, P. Lamas et X. Provençal. 1995-2006. Paquetage ruby pour la combinatoire des mots (code source et documentations), LaCIM,

`www.lacim.uqam.ca/pages/gen.php?page=packages_RubyComb&lang=fra.`

Brlek, S., S. Hamel, M. Nivat et C. Reutenauer, 2004. «On the Palindromic Complexity of Infinite Words», In J. Berstel, J. Karhumaki, D. Perrin (éd.), Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, vol. 15, no 2, p. 293–306

Brlek, S., et A. Ladouceur. 2003. «A note on differentiable palindromes», *Theoretical Computer Science*, vol. 302, p. 167–178.

Dekking, F. M. 1980. «On the structure of self generating sequences», Séminaire de théorie des nombres de Bordeaux, exposé 31.

de Luca, A. 1997. «Sturmian words : structure, combinatorics, and their arithmetics», *Theoretical Computer Science*, vol. 183, p. 45–82.

de Luca, A., et A. De Luca. 2006. «Pseudopalindrome closure operators in free monoids», *Theoretical Computer Science*, vol. 362, p. 282–300.

Droubay, X., et G. Pirillo. 1999. «Palindromes and Sturmian words», *Theoretical Computer Science*, vol. 223, p. 73–85.

Droubay, X., J. Justin et G. Pirillo. 2001. «Episturmian words and some constructions of de Luca and Rauzy», *Theoretical Computer Science*, vol. 255, p. 539–553.

Flint, S.J. (éd.), L. W. Enquist, A. M. Skalka, V.R. Racaniello et S. Jane Flint. 2003. *Principles of virology : molecular biology, pathogenesis, and control of animal viruses 2nd Edition*, Washington DC : Amer. Society for Microbiology Press, 918 p.

Glen, A., J. Justin, S. Widmer et L. Q. Zamboni, «Palindromic Richness», *European Journal of Combinatorics*, à paraître.

Halava, V., T. Harju, T. Kärki et L. Q. Zamboni. 2007. «Relational Fine and Wilf words», In P. Arnoux, N. Bédaride, J. Cassaigne (éd.), *Proceedings of WORDS 2007*, p. 159–167.

Hof, A., O. Knill et B. Simon. 1995. «Singular continuous spectrum for palindromic

Schrödinger operators», *Communications in Mathematical Physics*, vol. 174, p. 149–159.

Lothaire, M. 1983. *Combinatorics on words*, Addison-Wesley.

Lothaire, M. 2002. *Algebraic Combinatorics on words*, Cambridge University Press, 504 p.

Mignosi, F., et P. Séébold. 1993. «Morphismes sturmiens et règles de Rauzy», *Journal de théorie des nombres de Bordeaux*, vol. 5, p. 221–233.

Morse, M., et G. A. Hedlund. 1938. «Symbolic dynamics», *American Journal of Mathematics*, vol. 60, p. 815–866.

Stein, W. 2008. *Sage : Open Source Mathematical Software (Version 3.0.3)*, The Sage Group, `www.sagemath.org`.

Séébold, P. 1991. «Fibonacci morphisms and Sturmian words», *Theoretical Computer Science*, vol. 88, p. 365–384.

Tan, B. 2007. «Mirror substitutions and palindromic sequences», *Theoretical Computer Science*, vol. 389, p. 118–124.