Dans le secret des tresses ?
Dossier ENS du TIPE 2013-2014

Jocelyn Beauchesne

Résumé

Après avoir rapidement présenté le groupe des tresses sous forme d'un monoïde généré par des tresses de bases, j'évoquerai les problèmes de mots et de conjuguaison dont j'ai été amené à en chercher des solutions et à les implémenter.

L'objectif de ce TIPE est d'implémenter en *Caml-light* les différents éléments nécessaires à la constitution d'un cryptosystème fondé sur le groupe des tresses et reposant sur la difficulté du problème de conjuguaison.

Le groupe des tresses permet un échange asymétrique d'une clé. Pour la partie chiffrement j'ai choisi un chiffrement dît par ≪ flot ≫avec XOR tout en essayant d'exploiter les propriétés de la suite logistique et de la structure géométrique des tresses.

J'ai réalisé ce TIPE en monôme.

Table des matières

Présentation du groupe des tresses et de quelques-uns de ses problèmes

1 Présentation du groupe des tresses

1.1 La tresse

On peut définir une tresse comme un ensemble fini de brins qui s'entrelacent sans rebrousser chemin, par exemple la figure 2 représente une tresse à trois brins.

Dans la suite, on adoptera une représentation en mots. On note $n \geq 2$ le nombre entiers de brins, pour $i \in 1, .., n-1$ on note a_i l'interversion des brins i et $i+1$ où le i-ième brin passe au dessus du $i+1$-ième. Ainsi, dans la 2, la tresse s'écrit $a_1 \cdot a_2$.

1.2 Structure de groupe

En introduisant l'interversion inverse, *i.e.* interversion des brins i et $i+1$ où cette fois-ci le brin $i+1$ passe au dessus du i, on obtient une structure de groupe . On note a_i^{-1} cette interversion.

La loi est simplement la concaténation et n'a de sens que sur des tresses au même nombre de brins, ce qui se traduit visuellement par un recollement des extrémités des brins ayant même numéro i. On donne un exemple de cette opération en figure 4.

Cet exemple est la concaténation de $a_1^{-1} \cdot a_2$ et $a_2^{-1} \cdot a_1$ qui s'écrit : $(a_1^{-1} \cdot a_2) \star (a_2^{-1} \cdot a_1) = (a_1^{-1} \cdot a_2 \cdot a_2^{-1} \cdot a_1)$.

Par soucis de simplicité on assimilera $\star$ à $\cdot$ qu'on omettera tant que cela n'affecte pas la clarté.

On note qu'une fois « tirée »la tresse de la figure 4 est sans entrecroisement, c'est la tresse triviale que l'on note e, voir figure 3.

On note alors B_{n+1} le groupe des tresses à $n+1$ brins, de neutre e et généré par les interversions $a_i^{\pm 1}, i \in \{1, \ldots, n\}$.

En Caml : on introduit notre propre type, voir figure 14, qui est grossièrement une liste de couple d'entiers.

La compléxité des programmes agissant sur une tresse T de longueur $|T|$, nombre de générateurs dans l'écriture de T, sera compté en nombre d'opération élémentaire : ajout ou suppression d'un noeud, elle est noté $C_{programme}(T)$

Par exemple l'obtention de l'inverse d'une tresse se fait avec une compléxité linéaire en $|T|$: $C_{inverse}(T) = O(|T|)$. De même, générer une tresse de longueur $l \in \mathbb{N}$ se fait avec une compléxité linéaire. Idem pour le produit. Voir 15 pour le code.

1.3 Relations de tresses

Géométriquement, le groupe des tresses B_{n+1} possèdent les deux propriétés intéressantes. On pourrait dire qu'il est « partiellement commutatif ».

$$\forall (i,j) \in [\![1,n]\!] \begin{cases} a_i a_j & = a_j a_i, \ |i-j| \geq 2 \\ a_i a_j a_i & = a_j a_i a_j, \ |i-j| = 1 \end{cases}$$

FIGURE 1 – Relations de tresse

Ce qui est illustré par la figure 5.

On dira que deux tresses T_1 et T_2 sont égales si leurs écritures sur B_n coïncident terme à terme après simplification immédiate des couples $a_i \cdot a_i^{-1}$ présent dans l'écriture, on réservera le symbole $=$ à ces seuls cas. Si après un nombre fini d'application des relations précédentes sur la tresse T_1 celle-ci est égale à la tresse T_2 alors on dira que T_1 est équivalente à T_2 et on notera $T_1 \equiv T_2$.

2 Problèmes du groupe des tresses

2.1 Problème de mots

De ces relations, certes intéressantes, né un problème : celui de savoir si deux tresses sont équivalentes ou non. C'est le problème de mot.

Je me suis intéressé en premier lieu à la forme normale de Garside, comme décrite par exemple par [5]. J'ai également implémenté une autre méthode, la « réduction des poignées »de Dehornoy, voir 16 pour le code et [6] pour l'algorithme. Si il n'est pas très complexe, en revanche, la preuve de sa convergence est compliqué et je n'ai pas eu le temps de m'y intéresser.

L'intérêt de la forme normale est qu'elle permet de transmettre des tresses sans que la façon dont elles ont été construite n'apparaisse clairement. Par exemple, si je calcule *conjugue a b*, le résultat est $a \cdot b \cdot a^{-1}$ et un attaquant n'aura que peu de mal à identifier a et b.

2.1.1 Restriction au cas positif

Il est une tresse introduite par Garside aux propriétés intéressantes, nous la noterons Δ_n, voir 7 pour une représentation géométrique, et elle vérifie la relation de récurrence suivante :

$$\Delta_n = (a_1 \dots a_{n-1}) \cdot \Delta_{n-1} \text{ et } \Delta_1 = e \tag{1}$$

L'article [4] assure alors :

$$\Delta^{-1} a_{n-i} = a_i \Delta^{-1} \text{ ainsi que l'existence de } X_i \in B_n^+ \text{ tel que } a_i X_i = \Delta \tag{2}$$

Dès lors, étant donné une tresse T, dès qu'on rencontre un générateur négatif dans son écriture on écrit : $a_i^{-1} = \Delta^{-1}\Delta a_i^{-1} = \Delta^{-1} X_i$. On fait alors remonter les Δ^{-1} en utilisant la relation précédente. Finalement : $T \equiv \Delta^{-p} T^+$. L'implémentation se fait en une compléxité linéaire si l'on connaît d'avance les X_i, voir 17.

2.1.2 Résolution dans le cas positif sur B_n^+

Soit $(T_1, T_2) \in B_n^+$, on dit que T_1 divise T_2 à gauche si et seulement si il existe $T_3 \in B_n^+$ tel que $T_2 \equiv T_1 \cdot T_3$. On note alors $T_1 \dashv T_2$.

PGCD Étant donné $(T_1, T_2) \in B_n^+$, il existe une unique tresse $G = T_1 \wedge T_2$ telle que pour tout diviseur à gauche T de T_1 et T_2 on ait $T \dashv G$. Voir [5] pour la démonstration.

Ceci permet de prouver l'existence d'une décomposition unique, la forme normale de Garside :

$$T \equiv \Delta^p A_1 \dots A_j \text{ où } A_i \equiv \Delta \wedge A_i \dots A_j \text{ et } p \in \mathbb{Z} \tag{3}$$

Les tresses A_i peuvent alors être plongée dans Σ_n, pour n'avoir plus qu'un représentant de la classe d'équivalence.

2.1.3 Obtention :

J'ai donc appliqué la méthode décrite dans l'article [3], voir 18 pour le code. Le programme *left-WeightedDecomposition* a grossièrement une compléxité en $O(n \cdot |T|^4)$. En effet, l'algorithme est de faire remonter à gauche, en les ajoutant à la fin des listes certains éléments. À chaque fois qu'il faut faire cela, on est amené à faire une opération de coût $O(|T|^2 \cdot n)$ (les calculs des *sets* est en $O(n \cdot |T|)$ et f est composée d'au plus $O(|T|)$ éléments). Puis, factoriser est de compléxité linéaire en $|T|$, et enfin, on peut être amené à répéter ces opérations en plus $|T|$ fois, donc une compléxité maximale en $O(n \cdot |T|^4)$ soit polynomiale avec un facteur n.

Hélas, pour une raison que je ne suis pas parvenu à déterminer, l'algorithme fonctionne pour des tresses simples comme :

```
canoniser 5 (Noeud((2,1),Noeud((4,1),Noeud((2,-1),Noeud((4,-1),Triviale)))));;
```

qui renvoi le résultat souhaité, mais pas pour :

```
canoniser 7 (Noeud((2,1),Noeud((4,1),Noeud((6,-1),Noeud((4,-1),Noeud((6,1),Triviale))))));;.
```

Tout ceci donne naissance au programme *canonique* et à *egal* qui compare la forme canonique de deux tresses.

2.2 Problème de conjuguaison

Étant donnée deux tresses A et B, comment savoir si il existe, et comment trouver en cas d'existence, P telle que $A \equiv P \cdot B \cdot P^{-1}$.

C'est un problème difficile, résolu entièrement en temps polynomiale dans les seuls cas où n est petit ($n \leq 5$). En revanche, les travaux récent [2] parlent d'une résolution en temps polynomiale de la décision du problème de conjuguaison. Je ne sais toutefois pas si on peut exhiber un conjuguant en temps polynomiale à l'heure actuelle.

J'ai moi-même implémenté une simple résolution en brute-force.

Le système cryptographique qui suit, fonde sa sécurité sur la difficulté de ce problème. Sa pertinence dépend donc de la résolution de ce problème.

Application à la cryptographie

3 Un protocole asymétrique

Alex et Camille veulent communiquer en secret. Le système est asymétrique, de type Diffie-Hellman, et repose sur la ≪ presque ≫commutativité de B_2n. Alex génère une clé privée $A \in B_n$ et Camille une clé $C \in B_{n+1,2n}$ de sorte que A et C commutent. En se donnant $Pub \in B_{2n}$ les protagonistes peuvent alors mettre en commun une clé secrète P_S conformément à 8.

L'attaquant n'ayant *a priori* que connaissance de Pub, il ne peut que difficilement trouver, conformément au problème de conjuguaison, A et C et ne connait donc pas la tresse secrète P_S.

4 Le chiffrement via XOR et la suite logistique

4.1 Principe de fonctionnenement

Étant donné un message M codé sur n bits à transmettre, les deux protagonistes sont en possession d'une clé C codée également sur n bits. L'émetteur effectue l'addition bit à bit : $T = M \oplus C$ et transmet T. Le destinataire reçoit T et effectue l'opération inverse : $M = T \oplus C$. Voir 9 pour un schéma.

L'intérêt de cette méthode est qu'elle est simple et peut-être très sûre. Shannon a, d'après [1], énoncé des conditions mathématiques précises pour qu'une telle séquence soit qualifiée de parfaitement aléatoire et que l'algorithme soit très sécurisé. On s'en remet à des logiciels tels *ent* pour tester cela.

4.2 La suite logistique

On note $(u_k)_{k\in\mathbb{N}}$ la suite logistique vérifiant la relation :

$$u_{k+1} = \mu u_k(1 - u_k), u_0 \in [0, 1], k \in \mathbb{N} \tag{4}$$

L'idée, insuffisante on le verra, est d'utiliser la sensibilité aux conditions initiales de la suite logistique pour réaliser un système de génération d'une séquence de bits infini, apparement aléatoire mais déterministe en réalité. De sorte que les deux protagonistes génèrent la même séquence infinie et puisse ainsi communiquer une quantité arbitraire de données.

On se donne un vecteur T de taille $2n$ comme dans le protocole. On initialise les valeurs de chaque cases i à $\frac{i}{2n}$ par exemple. On choisit $\mu = 3.6+p\cdot10^{log_{10}(p)-2}$ où p est l'exposant de Δ dans la décomposition canonique, pour obtenir un comportement chaotique. Soit $\tau_1 \dots \tau_k$ la décomposition en permutation de notre tresse secrète. On applique alors la permutation aux éléments de notre tableau, et lorsqu'il y a changement de position, on itère la relation donnée précédement pour obtenir le terme suivant de la suite considérée.

Arrivé à la k-ième permutation, on considère la parité de la première décimale de chacunes des cases du tableau : on obtient une suite de $2n$ bits a priori aléatoire. On repète le procédé sur le nouveau tableau, *etc.* D'où une suite infinie de bits. On a représenté le procédé en 13.

4.3 Sécurité

Le programme `ent` nous donne une entropie de 1.4 sur une échelle de 0 à 8, dans le meilleur des cas pour notre algorithme. Ce qui n'est *a priori* pas suffisant pour une utilisation cryptographique réelle.

Pour mettre tout ceci en image j'ai utilisé le programme `convert` sous *GNU Linux*, voir 19 pour la commande exacte.

Pour un fichier vérifiant une entropie de 7.99 avec `ent`, on obtient 10. Mais la suite logistique seule donne 11, et notre algorithme donne 12.

De façon tout à fait informelle, on remarque que notre algorithme semble conserver les ≪ motifs ≫de la suite utilisée, décourageant ainsi la recherche d'autre suite pour modifier notre procédé.

L'alternative consisterait à utiliser des générateurs pseudo-aléatoires certifiés viable cryptographiquement en utilisant notre tresse secrète comme graîne de l'algorithme.

Conclusion

Les tresses sont des objets mathématiques intéressants et qui pourraient être des candidates pertinentes pour un système cryptographique asymétrique. Toutefois, les recherches récentes comme [2] semblent mettre à mal le protocole d'échange de clé présenté ici, car avec le problème de conjuguaison résolu en temps polynomial la sécurité du protocole n'est plus assurée.

De plus, mon idée de suite logistique en utilisant la structure géométrique des tresses pour créer une suite binaire infinie n'est pas suffisante pour assurer la sécurité des échanges chiffrés.

Cependant, peut-être que d'autres problèmes du groupe des tresses peuvent être exploités pour l'élaboration d'un système cryptographique et que de meilleures fonction de hashage de B_n vers $\{0,1\}^{\mathbb{N}}$ existent.

Références

[1] Fonction ou exclusif — Wikipedia, the free encyclopedia, 2010. [Online ; accessed 8-June-2014].

[2] Sandrine Caruso. Algorithmes et généricité dans les groupes des tresses. pages 5 – 8, 2013.

[3] Hugh R. Morton Elsayed A.Elrifai. Algorithms for positive braids. pages 6 – 13, 1991.

[4] F.A.Garside. The braid group and other groups. pages 1–8, 1967.

[5] Cédric Milliet. Groupe des tresses et cryptographie. pages 8 – 9, 2003.

[6] L.H. Robert N. Curien. Groupes des tresses and algorithme de réduction des poignées. pages 21–23, 2006.

Annexe : Images et code en *Caml-light* et *bash*

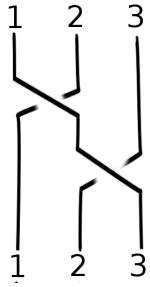

FIGURE 2 – Une tresse à trois brins

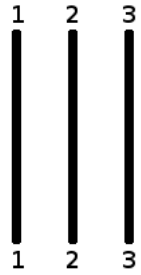

FIGURE 3 – La tresse triviale à trois brins.

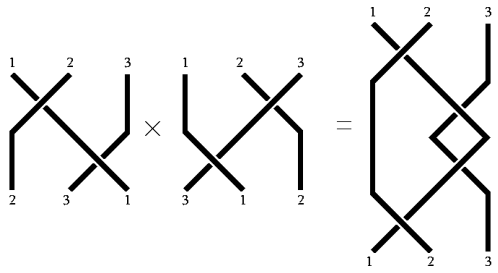

FIGURE 4 – La concaténation de deux tresses

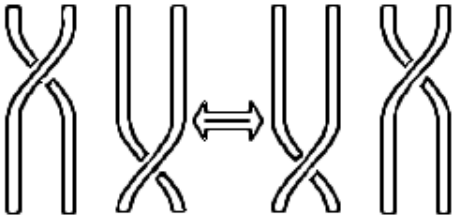

FIGURE 5 – Illustration de la relation $a_i a_j = a_j a_i$, $|i-j| \geq 2$

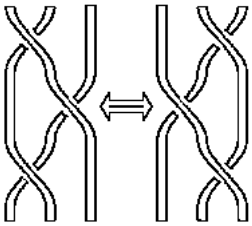

FIGURE 6 – Illustration de la relation $a_i a_j a_i = a_j a_i a_j$, $|i-j| = 1$

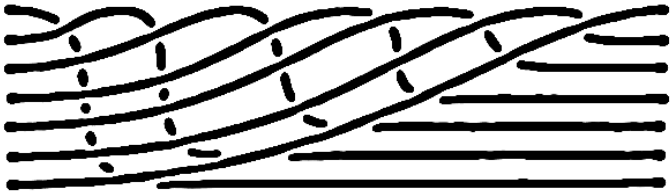

FIGURE 7 – Représentation de Δ dans B_6

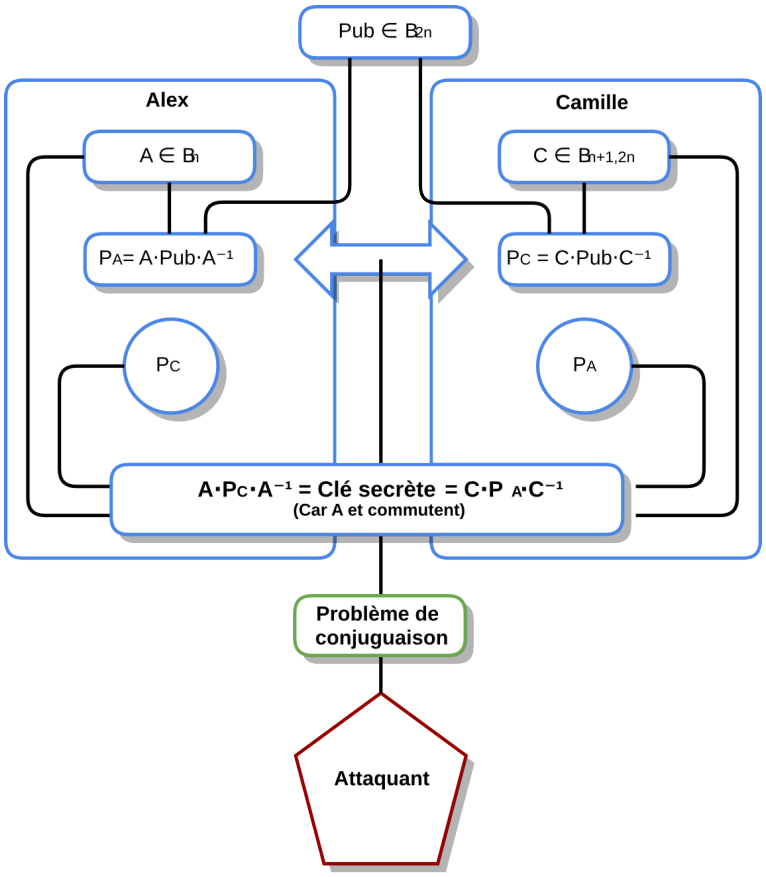

FIGURE 8 – Le protocole d'échange

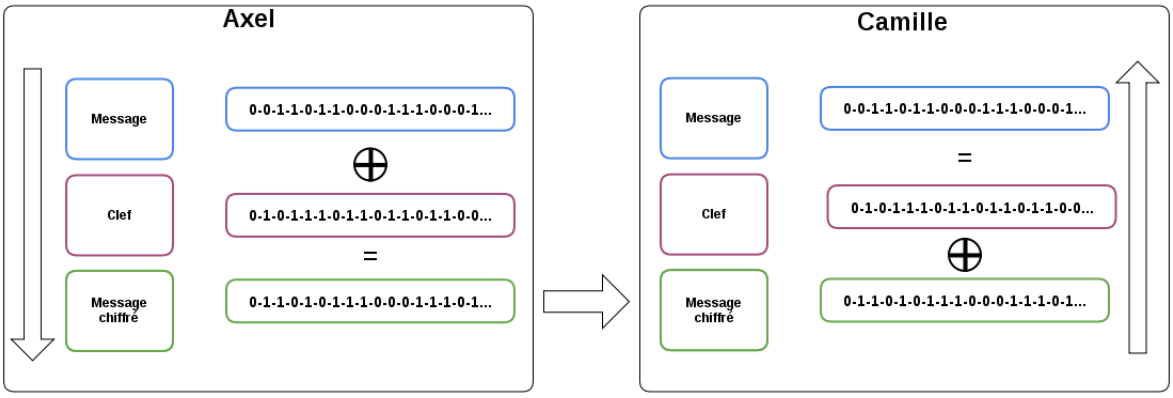

FIGURE 9 – Le chiffrement par « flot »avec XOR

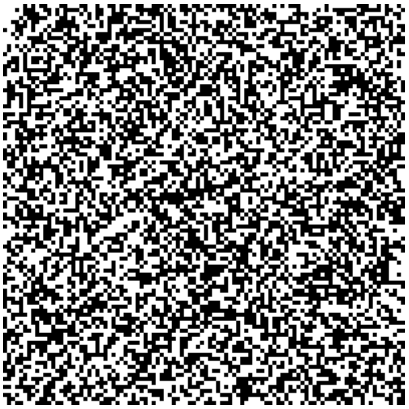

FIGURE 10 – Une séquence aléatoire de bits affichés

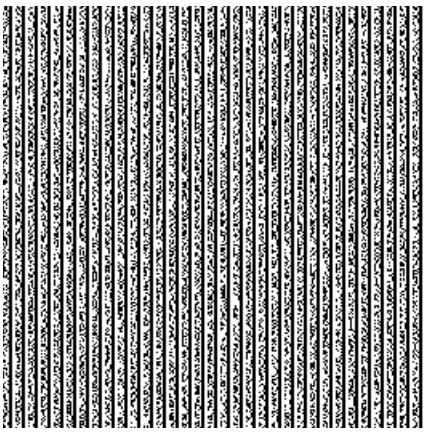

FIGURE 11 – La suite logistique affichée

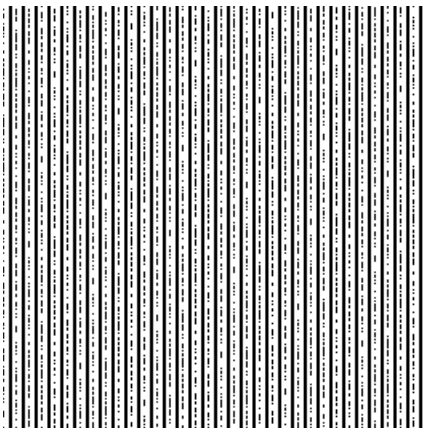

FIGURE 12 – Le résultat de notre algorithme affiché

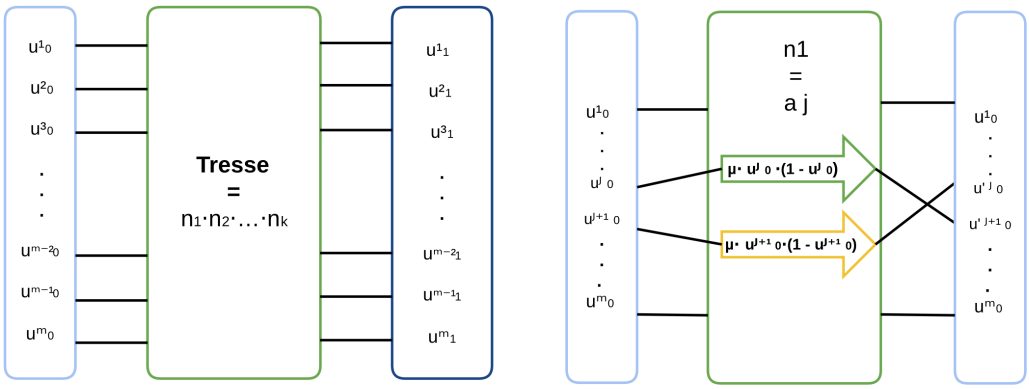

FIGURE 13 – La génération de la suite binaire pseudo-aléatoire

```
type tresse =
    |Triviale
    |Noeud of (int*int)*tresse
    |Delta of int*tresse;;
```

FIGURE 14 – Le type des tresses, $a_i^{\pm 1}$ donne Noeud((*i*,±1),Triviale)

```
let inverse t =
  let rec aux a b =
    match b with
      |Triviale -> a
      |Noeud((i,p),q) -> aux (Noeud((i,(-1)*p),a)) q in

    aux Triviale t;;

let rec generer m M l = (* m et M l'interval dans lequel doivent operer les permutations | ex : ge
    match l with
      |0 -> Triviale
      |l -> Noeud(((random__int (M-m))+m,plusmoinsun 2),(generer m M (l-1)));;

let produit x y =
  let rec concat a b =
    match a with
      |Triviale -> b
      |Delta(p,q) -> Delta( p, (concat q b) )
      |Noeud((i,p),q) -> Noeud((i,p),(concat q b)) in
    concat x y;;

let conjuguer x y = produit ( produit x y ) (inverse x) ;;
```

FIGURE 15 – Quelques programmes sur les tresses

```
let ch_poignee i l =
(*
 * Renvoi a,p,b, où p est la première i-poignée de l, c'est à dire que le premier élément de p
 * est a_i, le dernier est a_i^{-1} et les éléments entre sont d'indice strictement supérieur
 * à i (si il y en a).
 *)

let rec reduction i signe v =
  match v with
    | [] -> []
    | (t,s)::q when t = i+1 -> (i+1,-signe)::(i,s)::(i+1,signe)::(reduction i signe q)
    | (t,s)::q -> (t,s)::(reduction i signe q);;

let rec reduire v =
  let l = ref v in
  let booleen = ref true in

  while !booleen && !l <> [] do
    let m = ref (minimum (!l)) in
    let d,n1,p,n2,f = ch_poignee !m !l in
    let (t,s) = n1 in
      if n2 = (0,0) then booleen := false else l := d@(reduction t s (reduire p))@f;
  done;
!l;;
```

FIGURE 16 – La réduction des poignées de P.Dehornoy

```
let rec decompo_p n t = (*decomposition_positive à gauche*)
  match t with
    |Triviale ->       ( 0 , Triviale ) (*exposant de Delta, la queue*)
    |Delta(p,q) -> let ( e , r ) = decompo_p n q in
         ( e+p , r )
    |Noeud((i,p),q) when p = 1 -> let (e,r) = decompo_p n q in
 if pair e then ( e , Noeud( (i,1),r ) ) else ( e, Noeud( ((n-i),1),r ) )
    |Noeud((i,p),q) -> let ( e , r ) = decompo_p n q in
        if pair e then ( e-1 , (produit (quiestx i n) r) ) else ( e-1 , produit (R n (quiestx i n)
```

FIGURE 17 – La restriction au cas positif

```
let type tresse_canonique = {delta : int; eS : int vect list};;

let startingset l n =
  let tau = tresseEnListeVersPermutation (revL l) n in
  let sdeT = ref [] in
    for i = (n-1) downto 1 do
      if tau.(i+1) < tau.(i) then sdeT := i::!sdeT (* la liste est ordonnée *)
    done;
    !sdeT;;

let rec factoriser i t =
  match t with
    |x::q when x = i -> (i::q)
    |a::x::q when x= i && ( abs (a-i) ) = 1 -> let b::q2 = factoriser a q in
           (i::a::i::q2)
    |a::x::q when x = i -> (i::a::q)
    |x::q -> let q2 = factoriser i q in factoriser i (x::q2);;


let leftWeightedDecomposition n T =

  let rec aux d f = (* première moitiée - deuxième moitiée --
       * cette fonction coupe en deux la liste des éléments simple,
       * là où il n'y a pas maximalisation *)
    match f with
      |[x] -> true,x::d,[]
      |t1::t2::q -> let b,i = estSousEnsemble (startingset t2 n) (finishingset t1 n) in
             if b then (aux (t1::d) (t2::q))
             else
        let l = factoriser i t2 in
        match l with
   |[x] -> false,t1::d,[x]::q
                        |x::q2 -> false,t1::d,[x]::q2::q in

    let bol = ref false in let d = ref [] in let f = ref T in let t = ref 0 in let c = ref [] in
    let x,y,z = aux [] !f in bol := x ; d := y ; f := z;
    while non !bol do
      t := hd(hd(!f)); (* t est un générateur *)
      f := tl(!f);
      c := hd(!d) ;
      d := tl(!d);
      c := addEnd !c !t; (* On ajoute le générateur à la fin de c*)
      f := !c::!f;
    let x,y,z = aux [] ((!d)@(!f)) in bol := x ; d := y ; f := z;
    done;
    revL !d;; (*revL car les éléments sont stockés à l'envers lors du processus *)
```

FIGURE 18 – La canonisation

```
convert -size 2896x2896 mono:data -crop 100x100+0+0 -scale 300x300 data-mono.png
```

FIGURE 19 – La mise en image des binaires