Sur la vérification de preuve et la preuve automatique d'appartenance d'un mot à une grammaire.

Ulysse Durand

Contents

1 Définitions

Les grammaires formelles

Une grammaire formelle est un quadruplet $G = (T, N_t, S, D)$ où :

- T est l'alphabet des terminaux
- N_t est l'alphabet des non terminaux
- $S \in N_t$ est l'axiome
- D, un ensemble d'éléments de $(N_t \cup T)^\star \times (N_t \cup T)^\star$, est l'ensemble des règles de dérivation.

Soit $\rightarrow$ la relation binaire définie sur $(N_t \cup T)^\star$ par

$$\forall m, m' \in (N_t \cup T)^\star, m \rightarrow m' \iff \exists (a,b) \in D, u, v \in (N_t \cup T)^\star / m = uav \text{ et } m' = ubv \quad (1)$$

On note $\overset{*}{\rightarrow}$ la cloture transitive et réflexive de $\rightarrow$.
Pour $x \in (N_t \cup T)^\star$, notons

$\delta(x) := \{y \in (N_t \cup T)^\star / x \overset{*}{\rightarrow} y\}$ et

$\eta(x) := \delta(x) \cap T^\star$.

Alors le langage de la grammaire formelle G est le suivant :

$$\mathcal{L}(G) := \eta(S)$$

Nous allons supposer que N_t est dénombrable et T est fini.

En OCaml :

```
type 'e caractere = T of 'e | Nt of int

type 'a regle = ('a array) * ('a array)

type 'e fg = {
  terminaux : ('e caractere) array ;
  nonterminaux : ('e caractere) array ;
  axiome : 'e caractere ;
  reglesf : ('e caractere) regle array
}
```

Les grammaires non contextuelle

Définition : Une grammaire non contextuelle est une grammaire $G = (T, N_t, S, D)$ telle que :

$$\forall (a,b) \in D, a \in N_t$$

```
type 'a reglecf = int * ('a array)

type 'e cfg = {
  terminaux : ('e caractere) array ;
  nonterminaux : ('e caractere) array ;
  axiome : 'e caractere ;
  reglescf : ('e caractere) reglecf array
}
```

Exemple : la grammaire non contextuelle des expressions arithmétiques suffixes.

$$\begin{aligned} D =& \{(\underline{S}, \underline{SS+})_1, (\underline{S}, \underline{SS*})_2, (\underline{S}, \underline{"N"})_3\} \\ & \cup \{(\underline{N}, \underline{C})_4 | (\underline{N}, \underline{NC})_5\} \\ & \cup \{(\underline{C}, \underline{0})_6 | (\underline{C}, \underline{1})_7\} \end{aligned}$$

(les indices ci-dessus correspondent à une énumération des règles de dérivation, par exemple, $D_5 = (\underline{N}, \underline{NC})$)

Fôret de dérivation

Ici, $G = (T, N_t, S, D)$ est une grammaire non contextuelle.

Une forêt est une liste d'arbres

```
type 'a arbre = 'a * 'a foret
and 'a foret = F of 'a arbre list
```

Une preuve qu'un mot appartient à $\eta(m)$ peut être donnée par une forêt de dérivation :

> Si m est le mot vide, il s'agit de la forêt vide.
>
> Sinon, soit $n = |m|$,
>
> > Si $n = 1$ et $m_0 \in T$,il s'agit d'un seul arbre où la racine est m et la sous forêt est la forêt de dérivation d'un b tel que $(m, b) \in D$.
> >
> > Si $n = 1$ et $m_0 \in N_t$, il s'agit de l'arbre de racine m et de sous forêt vide.
> >
> > Si $n > 1$, il s'agit de la liste des arbres $[f_1, \ldots, f_n]$ où $\forall i \in [1, n]$, f_i est un arbre de dérivation de la lettre m_i.

D'où la fonction OCaml suivante :

```
let rec test_foret_deriv foret mot regles =
  let n = List.length mot in
  if n = 0 then foret = F [] else
```

```
  match foret with
      |(F sousarbres) ->
          if (List.length sousarbres) <> n then false else
          List.for_all2
          (fun arb lettre ->
              test_arbre_deriv arb regles
          )
          sousarbres
          mot
and test_arbre_deriv arb regles =
  let (lettre,sousforet) = arb in
  match lettre with
      |T x -> arb = (T x,F [])
      |Nt i ->
          let nouveaumot = lisracines sousforet in
          (List.mem (i,nouveaumot) regles)&&
          (test_foret_deriv sousforet nouveaumot regles)
```

L'ensemble des forêts de dérivation de m est noté $\mathcal{F}(m)$.

Le parcours infixe des feuilles d'une forêt de dérivation de x donne un mot de $\eta(x)$. Un exemple sera donnée dans le cas d'une grammaire non contextuelle.

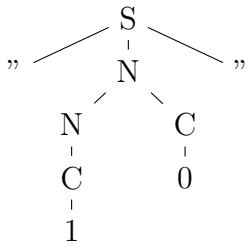
Pour $a \in \mathcal{F}(x)$, on note $\mathcal{I}(a)$ le parcours infixe des feuilles de a.

Exemple de preuve

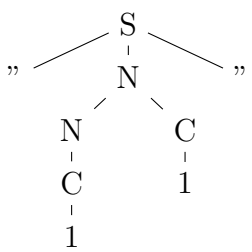
Une preuve que $\underline{\text{"10""11"+"0"*}} \in \mathcal{L}(G)$:
Lemme 1 : $\underline{\text{"10"}} \in \eta(S)$

En effet, $\underline{S} \to_3 \underline{\text{"}N\text{"}} \to_5 \underline{\text{"}NC\text{"}} \to_6 \underline{\text{"}N0\text{"}} \to_4 \underline{\text{"}C0\text{"}} \to_7 \underline{\text{"10"}}$

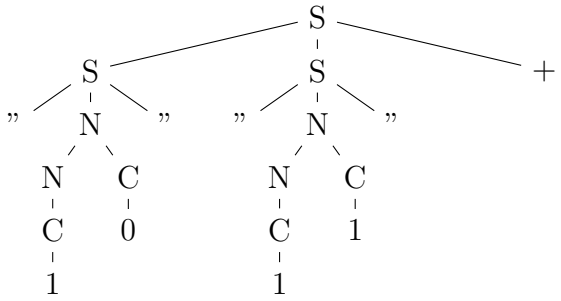
L'abre de dérivation fait aussi office de preuve, bien plus concise :



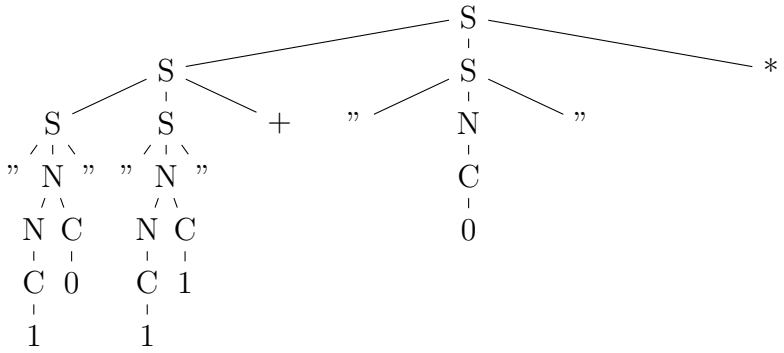
Lemme 2 : $\underline{\text{"11"}} \in \eta(S)$ Preuve avec l'arbre suivant



On peut alors, en combinant les deux arbres précédants, prouver que $\underline{\text{"10""11"+}} \in \eta(S)$:



Avec l'arbre suivant, on prouve enfin que $\underline{\text{"10""11"+"0"*}} \in \delta(S) \cap T^\star = \mathcal{L}(G)$:



On remarque bien qu'un parcours infixe des feuilles de l'arbre donne $\underline{\text{"10""11"+"0"*}}$

2 Vérification de preuve - grammaire non contextuelle

La forêt de dérivation nous donne une bonne manière de vérifier une preuve, car une preuve est caractérisé par une forêt de dérivation.

(en pratique, `'e` sera `string`).

Par induction on peut montrer $m \in \eta(x) \iff$ il existe p une forêt de dérivation dont le parcours infixe des feuilles est m et les racines sont les lettres de x.

NB : $A_{[p,q]}$ correspond à la forêt A restreinte à ses arbres d'indice $i/p \leq i \leq q$ et $A@B$ correspond à la concaténation des forêts A et B.

$m \in \eta(x)$
$\iff \exists n \in \mathbb{N} / \exists m_1, m_2, \ldots, m_n \in (N_t \cup T)^\star / x \to m_1 \to m_2 \to \cdots \to m_n$ et $m_n \in T^\star$ (2_n)
$\mathcal{P}(n) := ((2_n) \implies \exists A \in \mathcal{F}(x) / \mathcal{I}(x) = m)$

$\mathcal{P}(0) : x \in \eta(x) \implies x \in T^\star$ donc $[B(x_1); \ldots; B(x_{|x|})]$ convient.
$\forall n \in \mathbb{N}^*, \mathcal{P}(n-1) \implies (m \in \eta(m_1) \implies \exists A \in \mathcal{F}(m_1) / \mathcal{I}(A) = m)$

et comme $x \to m_1, \exists (a, b) \in D, u, v \in (N_t \cup T)^\star / x = uav$ et $m_1 = ubv$

Soit $p := |u|, q := |v|, r := |b|$

alors $A|_{[1,p]} \in \mathcal{F}(u), A|_{[p+1,p+r]} \in \mathcal{F}(b), A|_{[p+r+1,p+r+q]} \in \mathcal{F}(v)$

Donc $A|_{[1,p]} @ A' @ A|_{[p+r+1,p+r+q]} \in \mathcal{F}(x)$ où $A' = N(b, A|_{[p+1,p+r]})$

Soit $\mathcal{P}(n)$.

La réciproque se montre avec un algorithme qui à un arbre de dérivation associe une séquence $m_1, m_2, \ldots, m_n \in (N_t \cup T)^\star$ telle que $x \to m_1 \to m_2 \to \cdots \to m_n$

En entrée, nous avons f_0, une forêt de dérivation.

f est une copie de f_0.

Tant que f n'est pas constituée que de feuilles :

On prend a un arbre qui n'est pas une feuille dans f

Remplacer a, dans f, par ses fils.

Lire les racines de la forêt de gauche à droite.

En remplaçant a dans f par ses fils, on dérive notre mot une fois.

Alors $a \in \mathcal{F}(S)$ est une preuve que $\mathcal{I}(a) \in \mathcal{L}(G)$.

Alors, vérifier une preuve donnée par une forêt consiste juste à vérifier que la forêt est bien une forêt de dérivation.

3 Preuve automatique - grammaire non contextuelle

Toujours dans une grammaire non contextuelle

Avec une grammaire dont l'axiome est S, pour générer automatiquement une preuve, l'idée est qu'à chaque étape, on dispose d'une liste de mots prouvés, et à chaque étape, on fait toutes les dérivations possibles de notre liste de mots prouvés pour obtenir une liste de mots prouvés plus grande.

Deux approches coexistent alors, soit on fait toutes les dérivations possibles dans le sens direct, soit dans le sens indirect.

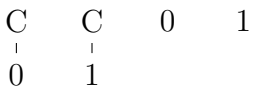
La méthode bottom-up

On part des $\{[(b_1, []); \ldots; (b_n, [])] \mid (a, b) \in D \text{ et } b \in T^\star, n = |b|\}$ et on construit d'autres arbres de dérivation à partir des règles de dérivation.

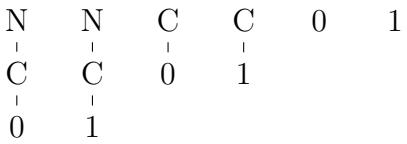
Par la méthode bottom-up, on générerait dans l'ordre :
à l'étape 0 :

0 1

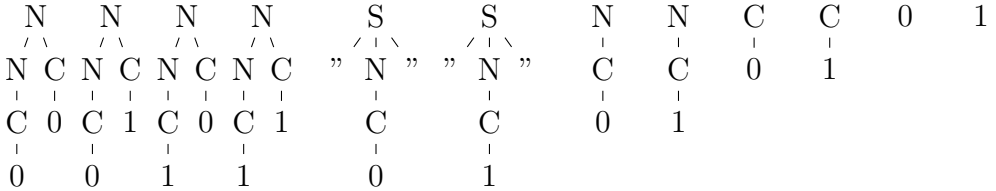
à l'étape 1 :



à l'étape 2 :



à l'étape 3 :



On prouve par induction simple qu'à l'étape k, on a généré tous les arbres de dérivations possibles de profondeur $\leq k$.

La méthode top-down

On part de l'axiome S de la grammaire non contextuelle pour construire des mots de $\delta(S)$ en remplaçant un nonterminal x d'un mot par un b d'une règle de dérivation (x, b).

Cette méthode suit d'assez près la définition des mots de la grammaire non contextuelle
(on peut écrire la clôture transitive et réflexive $\xrightarrow{*}$ comme : $\xrightarrow{*} = \bigcup_{n \in \mathbb{N}} \rightarrow^n$, avec le produit usuel de deux relation binaires $\forall \mathcal{R}, \mathcal{R}' \in X^2, \mathcal{R}\mathcal{R}' := \{(x, z) \in X^2 \mid \exists y \in X / (x, y) \in \mathcal{R} \text{ et } (y, z) \in \mathcal{R}'\}$).
On génère en effet, avec cette méthode, à la n-ième étape,

$A_n = \left\{ m \in (N \cup T)^\star \mid S \left(\bigcup_{k \in [1,n]} \rightarrow^k \right) m \right\}$. Il est alors simple de vérifier que tout mot du langage de la grammaire sera généré en temps fini par cette méthode.

On génère alors dans l'ordre :

à l'étape 0 : $\{S\}$,
à l'étape 1 : $\{S, SS+, SS*, "N"\}$,
et à l'étape 2 : $\{S, SS+, SS*, "N", SS*S+, SSS*+, SSS+*, SS+S*,$
$"N"S+, S"N"+, "N"S*, S"N"*, "C", "NC"\}$.

Cette méthode est bien plus proche de la définition et est directement applicable aux grammaire formelles.

4 Vérification de preuve - grammaire formelle

Une nouvelle structure de données pour les preuves

Nous ne disposons plus des forêts de dérivation. Alors nous pouvons nous restreindre à la définition d'un mot qui dérive d'un autre. On va alors fournir, comme preuve, une liste de mots $m_1, \ldots, m_n$ tels que $\forall i \in [1, n-1], m_i \to m_{i+1}$. Pour ceci nous allons renseigner de quelle manière le mot m_{i+1} dérive du mot m_i par la règle de dérivation (a, b), par exemple en fournissant les indices délimitant a dans m_i, et b le remplaçant. Nous aboutissons alors à :

```
type 'e preuveformelle = (int*int*('e caractere list)) list
```

Nous prendrons comme exemple la grammaire formelle $G = (T, N_t, S, D)$ où :
$T = \{\underline{a}, \underline{b}, \underline{c}\}$
$N_t = \{\underline{S}, \underline{B}\}$
$D = \{(\underline{S}, \underline{aBSc})_1, (\underline{S}, \underline{abc})_2, (\underline{Ba}, \underline{aB})_3, (\underline{Bb}, \underline{bb})_4\}$,
alors $\underline{\text{aabbcc}}$ est dans $\mathcal{L}(G)$ car $\underline{S} \to_1 \underline{aBSc} \to_2 \underline{aBabcc} \to_3 \underline{aaBbcc} \to_4 \underline{aabbcc}$.
En Ocaml :

```
let unepreuve = [(0,1,mot "aBSc"),(2,3,mot "abc"),(1,3,mot "aB"),(2,4,mot "bb")]
```

Vérifier une preuve dans une telle structure de données

```
let verif_deriv p grf mot =
    let i,j,m = p in
    let achanger = soustrait i j mot in
    List.mem (aechanger,m) grf.reglesf

let verif_preuve p grf =
    let rec aux p mot =
        match p with
            |[] -> true
            |t::q ->
                let i,j,m = t in
                (verif_deriv t grf) && (aux q (remplace i j mot))
    in
```

```
    aux p [grf.axiome]
```

5 Preuve automatique - grammaire formelle

Chercher automatiquement une preuve d'appartenance d'un mot à une grammaire non contextuelle se fait déjà très bien avec les automates à pile. Nous allons ici nous intéresser plus spécifiquement aux grammaires formelles, et réutiliser notre méthode top-down. La méthode s'applique alors directement de la même manière. Nous avons donc un algorithme qui permet théoriquement de prouver que n'importe quel mot de la grammaire appartient bien à cette dernière. Le problème étant la complexité de ce dernier, qui fonctionne comme une machine non déterministe où on ne coupe pas les instances arrivant à un état puit. La suite va donc consister à trouver, de manière heuristique ou non, des méthodes affirmant qu'à partir d'un mot m, on aura des difficultés à dériver en notre mot m' à prouver. Alors, nous allons arrêter les recherches de dérivations de S en m' qui passent par le mot m.

Pour ce faire, pour dériver un mot m en mot m', on calcule $S(x) := \{x \in (N \cup T)^\star \mid m \to x\} = \{ubv \in (N_t \cup T)^\star \mid \exists (a,b) \in D / \exists (u,v) \in (N_t \cup T)^\star / x = uav\}$, puis, pour chaque $y \in S(x)$, on cherche une dérivation de y en m'.

Pour calculer $S(x)$, il suffit de faire une recherche de facteur.

D'où une implémentation en OCaml :

NB : `remplace x i l b` donnera le mot `x` où $x_{[i,i+l[}$ est remplacé par `b` .

`ppregles` est l'ensemble des règles de dérivation de la grammaire qui sont prétraitées pour l'algorithme kmp. `ppregles` est donné par `preprocessgf grf`

```
let preprocessgf grf =
  Array.map
  (fun (a,b) ->
    (a,b,kmppreprocess a)
  )
  grf.reglesf

let succ ppregles x =
  let res =
  List.flatten
  (
    Array.to_list (
      Array.map
      (fun (a,b,pp) ->
        List.map
        (fun i ->
```

```
          remplace x i (Array.length a) b
        )
        (kmp x a pp)
      )
      ppregles
    )
  )
  in
  ajouteplein res []
```

`succ ppregles x` calcule $S(x)$.

Un parcours en largeur particulier

Nous allons utiliser un parcours qui nous permettra de trouver dans un graphe un chemin plus court entre un sommet donné et un sommet 'valide'. Ce parcours pourra aussi éviter les chemins passant par un sommet 'invalide'.

Alors on fournira à ce parcours une fonction `elimine` et une fonction `termine` toutes deux de type `sommet -> bool` .

Ce parcours est donc un parcours en largeur depuis un sommet x_0 qui garde en mémoire le chemin parcouru et qui s'arrête dès qu'un sommet s, tel que `termine s` , est parcouru. Il renvoit alors un chemin de x_0 à s. Lorsque le parcours passe par un sommet s tel que `elimine s` , il ne parcourt pas ensuite ses voisins.

En voici une implémentation OCaml :

```
let rec parcoursmagique delta elimine termine dejavu avoir =
  let navoir = ref [] in
  let ndejavu = ref dejavu in
  if avoir = [] then None else (
  let res =
    List.find_opt
    (function
      |[] -> failwith "mauvais chemin"
      |s::q ->
        ndejavu := ajoute s (!ndejavu);
        if (termine s) then true else
        if (elimine s || List.mem s dejavu) then false else
        (
        navoir :=
          ajouteplein
          (
            List.map
            (fun v -> v::s::q)
            (delta s)
          )
```

```
          (!navoir);
        false)
    )
    avoir
    in
  match res with
    |None -> parcoursmagique delta elimine termine (!ndejavu) (!navoir)
    |Some x -> Some x
  )
```

Enfin, un prouveur automatique d'appartenance d'un mot à une grammaire formelle

Il ne nous reste plus qu'à appliquer notre parcours sur un graphe où les sommets sont les mots de $(N_T \cup T)^\star$, et les arêtes sont $\{(a,b) \in (N_T \cup T)^\star \mid a \to b\} = \{(a,b) \in (N_T \cup T)^\star \mid b \in S(a)\}$.

En OCaml :

```
let chercherderivationnaif x m ppregles =
  parcoursmagique
  (succ ppregles)
  (fun x -> false)
  (fun x -> x = m)
  []
  [[x]]
```

Par la suite, nous allons apporter des améliorations, en fournissant des fonctions `elimine` .

6 Amélioration pour les grammaires croissantes

Une grammaire croissante est une grammaire telle que :

$$\forall (a,b) \in D, |a| \leq |b| \tag{2}$$

On a alors une première propriété très simple, $\forall m, m' \in \delta(S), m \xrightarrow{*} m' \implies |m| \leq |m'|$.

Alors, dans la recherche de dérivations de S vers m, on peut supprimer les "branches de recherche" qui partent d'un mot de longueur $> |m|$. Il ne reste qu'à faire le même parcours que précédement avec la fonction suivante comme fonction `elimine` .

```
let eliminecroiss x = Array.length x > Array.length m
```

7 Amélioration dans le cas général : déduction sur le nombre d'occurence de chaque lettre

Rappelons, le problème : étant donné un mot m d'une grammaire, comment le dériver en un mot m' ? Pour tout $l \in N \cup N_t$, on va chercher un ensemble $q(l)$ qui majore $|\delta(m)|_l$.

Ainsi, $m \xrightarrow{*} m' \implies |m'|_l \in q(l)$, la contraposée nous sera utile :

$$|m'|_l \notin q(l) \implies \neg(m \xrightarrow{*} m')$$

Ce qui pourra nous permettre de réduire notre champ de recherche. En effet,

si pour un mot $x \in \delta(m)$, $\forall l \in (T \cup N_t), |\delta(x)|_l \cap q(l) = \emptyset$, alors $m' \notin \delta(x)$.

Pour calculer $|\delta(x)|_l \cap q(l)$, nous allons utiliser un graphe semblable à un automate.

Les sommets sont des états de $Q \subset (\mathcal{P}(\mathbb{N}))^{T \cup N_t}$ tels que :

$$\forall q, q' \in Q, \exists l \in (T \cup N_t)/q(l) \cap q'(l) = \emptyset$$
$$\text{et } \forall m \in \delta(S), \exists q \in Q/\forall l \in (T \cup N_t), |m|_l \in q(l)$$

Ainsi à tout mot $x \in \delta(S)$, on peut associer un unique état q tel que $\forall l \in (N_t \cup T), |x|_l \in q(l)$

Les arêtes du graphe sont les dérivation possibles d'un famille majorante q à une autre q'.

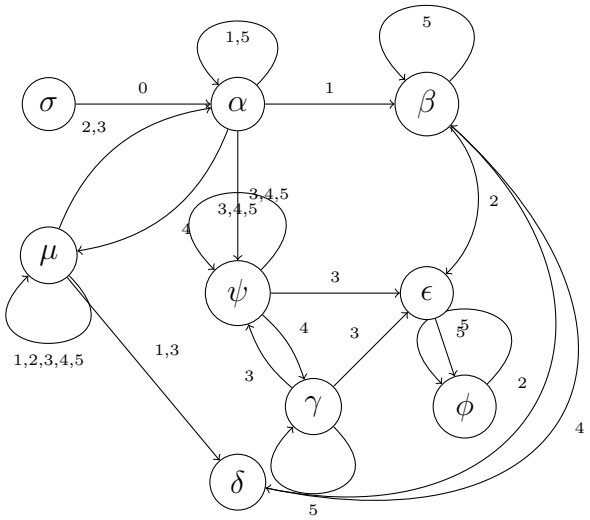
$$\forall (q, q') \in Q^2, (q, q') \in A \iff \exists x, x' \in (N_t \cup T)^\star / x \to x' \text{ et } \forall l \in N_t \cup T, |x|_l \in q(l) \text{ et } |x'|_l \in q(l)$$

Ainsi, si m correspond à un état q, et m' à un état q', alors $m \xrightarrow{*} m' \implies q'$ est accessible depuis q.

Les états q sous la forme $q \in \{\{0\}, \mathbb{N}^*\}^{(N_t \cup T)}$

Exemple avec la grammaire suivante : $\mathrm{D} = \{(S, \underline{abc})_0, (\underline{abc}, !ab!)_1, (\underline{b}, \underline{k})_2, (\underline{c}, \underline{ak})_3, (\underline{kak}, \underline{aa})_4, (\underline{a}, \underline{aaa})_5\}$

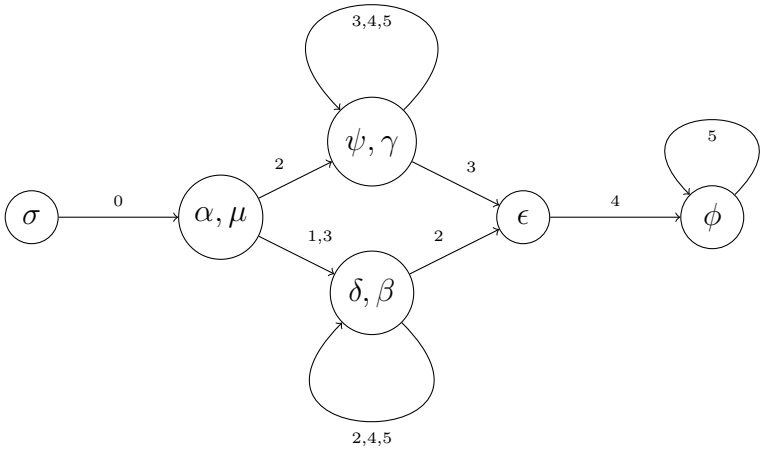
Le graphe est le suivant :



Et voici ce à quoi correspondent les différents états (sommets) du graphe :

q	α	β	γ	δ	ϵ	φ	ψ	μ	σ
$q(a)$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(b)$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$
$q(c)$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(k)$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$	$\mathbb{N}^*$	$\mathbb{N}^*$	$\{0\}$
$q(S)$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\mathbb{N}^*$

Le graphe réduit suivant sera utile :



Ainsi, nous pouvons tout de suite affirmer qu'il est impossible de dériver aaabakab en akkcckaaakck (en effet, ψ n'est pas accessible depuis δ)