03/02 2015

Algorithme du texte : exemples et applications

907

On note Σ un alphabet fini

I Recherche de motif [Cor]

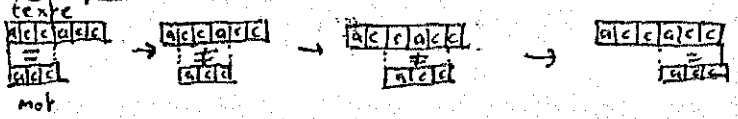
La recherche d'un mot dans un texte est un problème fréquent. On propose deux algorithme.

1. Algorithme naïf

```
Recherche_Naif(mot, texte)
    m := mot.longueur
    n := texte.longueur
    pour i=0 à n-m
        si mot = texte[i+1...i+m]
            afficher "position i"
    afficher "fin de parcours"
```

On peut facilement interpréter graphiquement l'algorithme

exemple 1 :



Prop 2 : La complexité en temps de Recherche_Naif est $O(m(n-m))$ (pire des cas). En moyenne : $O(n-m)$

2. Automate fini

a) Pré-traitement

Avant de rechercher le mot dans le texte, on "étudie" le mot pour mieux connaitre sa structure, qu'on code dans un automate fini.

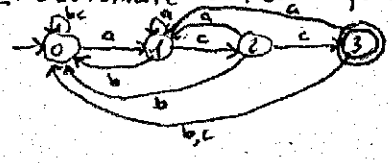
Fixons 'mot' et définissons σ, la fonction suffixe associée à mot. σ prend un élément w de Σ^* et renvoi la taille du plus long suffixe de w qui est un préfixe de "mot".

exemple 3 : avec mot = acc. $\sigma(accb) = 0$, $\sigma(a) = 1$
$\sigma(abac) = 2$, $\sigma(acccacc) = 3$

On définit ensuite l'automate par :

les états sont $Q = [0 \dots m]$ 0 état initial m état final	La fonction de transition est $\forall a \in \Sigma, q \in Q \quad \delta(q,a) = \sigma(mot[0..q-1]+a)$

exemple 4 : L'automate obtenue pour le mot acc est




```
Calcul_δ(mot, Σ)
    m := mot.longueur
    pour q=0 à m
        pour tout a∈Σ
            k := min(m, q+1)
            tant que mot[1...k] ≠ (mot[-q...-1]+a)[-k...-1]
                k := k-1
            δ(q,a) := k
    Renvoyer δ
```

prop 5 : La complexité de Calcul_δ en temps est $O(m \cdot |\Sigma|)$

b) Parcours

Maintenant il suffit de parcourir le texte en suivant les transitions. Arriver dans l'état m signifie qu'on a lu le mot recherché.

```
Recherche_Automate(mot, δ, texte)
    m := mot.longueur
    n := texte.longueur
    q := 0
    pour i = 1 à n
        q := δ(q, texte[i])
        si q=m
            afficher "position i-m"
    afficher "fin de parcours"
```

Prop 6 : La complexité en temps de Recherche_Automate est $O(n)$

c) Amélioration

L'algorithme Knuth-Morris-Pratt propose un pré-traitement plus astucieux en $O(m)$, en calculant un tableau π qui contient dans la case i la taille du plus long préfixe de "mot" qui est un suffixe propre de mot[1...i]. π permet de calculer δ à la volée lors du parcours.

↓ y a-t-il mieux ? Etre linéaire en la taille du mot ? => arbre des suffixes !

Grands paradigmes de l'algorithmique ?
→ programmation dynamique.

exemple 7: pour mot = ababc

i / π	a	b	a	b	c
	1	2	3	4	5
	0	0	1	2	0

II Comparaison de mots

Comparer des chaînes de caractères est parfois intéressant, des fragments d'ADN par exemple.

1) Distance d'édition [Cor]

Cette distance compte le coût minimal pour modifier un mot x en un autre mot y. On utilise pour cela un mot tampon z initialement vide, où grâce à un indice j on écrit sur z et grâce à un indice i on lit sur x. On s'autorise les opérations suivantes, chacune ayant son coût :

(c1) copier : $z[j] := x[i]$; incr i ; incr j
(c2) remplacer : $z[j] := c$; incr i ; incr j
(c3) supprimer : incr i
(c4) insérer : $z[j] := c$; incr j
(c5) permuter : $z[j] := x[i+1]$; $z[j+1] := x[i]$; incr i ; incr j
(c6) égaliser : $i := m+1$

Le coût minimal pour être dans l'état où il est écrit z sur le tampon et l'indice de x est i est donné par

$$f(z,i,j) = \min \begin{cases} c_1 + f(z[0..-2], i-1, j-1) & \text{si } i \geq 1, j \geq 1 \quad z[j-1] = x[i-1] \\ c_2 + f(z[0...-2], i-1, j-1) & \text{si } i \geq 1, j \geq 1 \\ c_3 + f(z, i-1, j) & \text{si } i \geq 1 \\ c_4 + f(z[0...-2], i, j-1) & \text{si } j \geq 1 \\ c_5 + f(z[0...-3], i-2, j-2) & \text{si } i \geq 2, j \geq 2, x[i-2] = z[j-1] \text{ et } x[i-1] = z[j-2] \\ c_6 + f(z, k) \text{ avec } k \in [0, m] & \text{si } i = m+1 \end{cases}$$

le coût cherché est donc $\min f(y, m+1, j)$ Cette fonction de récurrence permet de construire un algorithme en programmation dynamique.

exemple 8 : Pour $c_1 = 1$, $c_2 = 1$, $c_3 = 10$, $c_4 = 10$, $c_5 = 10$, $c_6 = 10$
La distance d'édition entre "ab" et "aa" est **2**

2) Plus longue sous-séquence commune [Cor]

En génomique, le calcul de plus longue sous-séquence commune permet de reconstruire un génome à partir de fragment.
Soit x et y deux mots sur Σ, on cherche le plus long mot w tel que les lettres de w apparaissent dans l'ordre dans x et dans y.

exemple 9 : la plus longue sous-séquence commune de "abcde" et "ceij" est "ce"

[Ce problème se résout par programmation dynamique, avec un algorithme de complexité exponentielle en la taille de x et y.] DVP

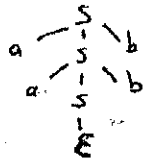
III Analyse syntaxique

Lorsqu'un programmeur écrit un programme, il le fait dans un éditeur de texte en concaténant des caractères. Il est donc nécessaire de savoir retrouver la structure du programme à partir du texte tapé.

Def 10 : Une grammaire non-contextuelle est un quadruplet (V_N, V_T, P, S), où :
V_N est l'ensemble des non-terminaux
V_T est l'ensemble des terminaux
$P \subset V_N \times (V_T \cup V_N)^*$ est l'ensemble des règles
$S \in V_N$ est l'axiome.

exemple 11 : $(\{S\}, \{a,b\}, \{S \to aSb, S \to \varepsilon\}, S)$ est une grammaire non-contextuelle, on notera plus simplement : $S \to aSb \mid \varepsilon$ la grammaire.

Cette grammaire permet de générer le langage $\{a^n b^n \mid n \in \mathbb{N}\}$. a^2b^2 est donc un mot généré par la grammaire et son arbre syntaxique est :



Prop 12 : Pour toute grammaire non-contextuelle G, on peut construire un automate à pile qui reconnait le langage de G.

Application 13 : Reconnaissance et construction de l'arbre syntaxique d'un texte généré par une grammaire LR(0) — DVP

revoir la formulation ! DVP

IV Codage de Huffman

vidéo + son + format d'images avec pertes

Le codage de Huffman est un algorithme qui permet de compresser des textes sans perte. Cherchons à coder le texte "l'agregation c'est chouette"

1) Calcul des occurrences.

O(n)

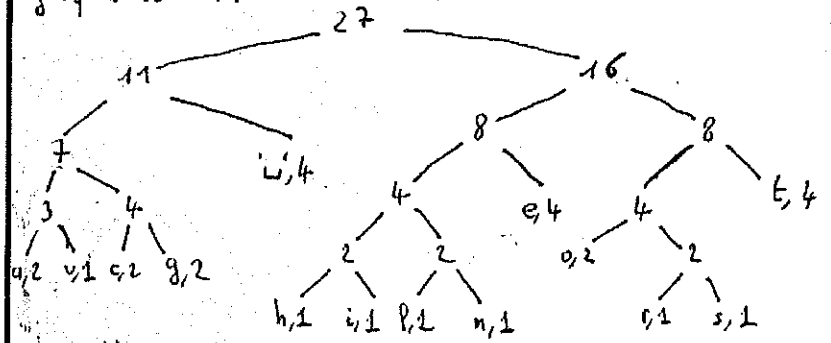
a,2 c,2 e,4 g,2 h,1 i,1 l,1 n,1 o,2 r,1 s,1 t,4 u,1 'u',4

2) Construction de l'arbre de Huffman.

On considère les paires précédentes comme des noeuds dont le poids est l'occurence de la lettre et on les fusionnent deux par deux en commençant par les noeuds de poids minimal et on crée ainsi des arbres dont les poids sont la sommes des poids des noeuds.

a,2 c,2 e,4 g,2 (2 : h,1 i,1) l,1 n,1 (2 : o,2) (2 : r,1 s,1) t,4 u,1 'u',4

jusqu'à obtenir :



Pour obtenir le code binaire de chaque lettre on descend dans l'arbre en lisant un 0 si on descend à gauche et un 1 sinon. "l'agreg" se code donc

1001001000000111101010100111.

ce qui représente 27 bits au lieu de 49 en ascii.

3) Décodage

Le décodage est similaire au codage : en lisant un code, on descend dans l'arbre jusqu'à un noeud. Puis on recommence depuis la racine.

complexité ? $O(n^2)$

structure de donnée adaptée ? type abstrait ? recherche du min + fusion ⇓ remplacer par file de priorité

Algorithmique du texte

→ dire qu'il y a de la programmation dynamique (à l'oral).

Plus longue sous-séquence commune

Référence : T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST, C. STEIN
Introduction to algorithms

2011-2012

Étant donnés deux mots $X = x_1 \cdots x_m$ et $Y = y_1 \cdots y_n$ sur un alphabet Σ, on cherche à savoir quel est le plus long sous-mot commun à X et Y.

La solution naïve est d'énumérer tous les sous-mots de X, et de regarder ensuite s'ils sont aussi sous-mots de Y, mais cette solution est inefficace : il y a 2^m sous-mots de X.

NOTATION – Si U est un mot de longueur p, on note U_{p-j} le préfixe de U de longueur $p - j$.

Un algorithme plus efficace pourra être déduit du théorème de structure suivant :

Théorème 1 : de structure des plus longues sous-séquences communes
Supposons que $Z = z_1 \cdots z_k$ est une plus longue sous-séquence commune à X et Y. Alors :

(i) Si $x_m = y_n$, alors $z_k = x_m (= y_n)$ et Z_{k-1} est une plus longue sous-séquence commune à X_{m-1} et Y_{n-1}.

(ii) Si $x_m \neq y_n$ alors

$$(z_k \neq x_m) \Rightarrow Z \text{ est une plus longue sous-séquence commune à } X_{m-1} \text{ et } Y.$$

(iii) Si $x_m \neq y_n$ alors

$$(z_k \neq y_n) \Rightarrow Z \text{ est une plus longue sous-séquence commune à } X \text{ et } Y_{n-1}.$$

Démonstration. C'est plutôt facile :

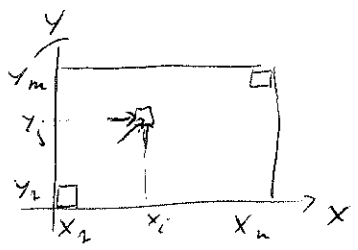
(i) $x_m = y_n$. Si $z_k \neq x_m$, alors Zx_m est un sous-mot commun à X et Y, ce qui contredit l'hypothèse de maximalité. D'où $z_k = x_m = y_n$.
De plus, Z_{k-1} est un sous-mot commun à X_{n-1} et Y_{m-1}. Supposons qu'il existe un sous-mot commun W de longueur plus grande que $k - 1$.
Alors, Wx_m est de longueur plus grande que k, et Wx_m est un sous-mot commun à X et Y, d'où une contradiction.
Donc Z_{k-1} est une plus longue sous-séquence commune à X_{m-1} et Y_{n-1}.

(ii) $x_m \neq y_n$. Supposons $z_k \neq x_m$. Alors Z est un sous-mot de X_{m-1}, et donc *a fortiori un sous-mot commun à X_{m-1} et Y.*
Supposons qu'il existe un sous-mot W commun à X_{m-1} et Y, de longueur plus grande. Alors W est a fortiori un sous-mot commun à X et Y, ce qui contredit l'hypothèse de maximalité de Z.

(iii) *cf* (ii)

□

Exo 1



Écrire la relation de récurrence.

Connaissant la plus longue sous-séquence commune au mot vide et n'importe quel mot, on peut en déduire formule de récurrence sur la longueur de la plus longue sous-séquence commune de X et Y. Si on note $c[i,$ longueur de la plus longue sous-séquence commune à X_i et Y_j, on a :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ c[i-1,j-1]+1 & \text{si } i,j > 0 \text{ et } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{si } i,j > 0 \text{ et } x_i \neq y_j \end{cases}$$

Cette formule pourrait nous donner un algorithme récursif pour calculer $c[m,n]$, mais sa complexité ser nouveau exponentielle. On va plutôt utiliser un algorithme de *programmation dynamique*.

Cet algorithme va calculer les uns après les autres les valeurs des $c[i,j]$, en utilisant la formule de récurre en remplissant le tableau par lignes.

Algorithm 1 Longueur-PLSC

Préconditions: X, Y mots sur Σ

```
m ← Longueur(X)
n ← Longueur(Y)
pour i = 1 to m faire
        c[i,0] ← 0
ruop
pour j = 0 to n faire
        c[0,j] ← 0
ruop
pour i = 1 to m faire
        pour j = 1 to n faire
                si x_i = y_j alors
                        c[i,j] ← c[i-1,j-1]+1
                        b[i,j] ← " ↖ "
                sinon si c[i-1,j] ⩾ c[i,j-1] alors
                        c[i,j] ← c[i-1,j]
                        b[i,j] ← " ↑ "
                sinon
                        c[i,j] ← c[i,j-1]
                        b[i,j] ← " ← "
                is
        ruop
ruop

retourner c et b
```

doit-on faire les flèches dans (i.e, ne pas faire deuxième tabl mais juste pour le chemin, comparer voisins de diag, gauche).

La matrice b sert à noter "d'où on vient" pour calculer la valeur $c[i,j]$ d'une case. Grâce à ce tableau, on p donc retrouver la plus longue sous-séquence commune à X et Y par l'algorithme :

Algorithm 2 PLSC

Préconditions: X mot sur Σ, b matrice calculée par Longueur-PLSC, i, j indices.

si $i = 0$ ou $j = 0$ **alors**

    afficher ()

is

si $b[i,j] =$ " $\nwarrow$ " **alors**

    PLSC$(b, X, i-1, j-1)$

    afficher x_i

sinon si $b[i,j] =$ " $\uparrow$ " **alors**

    PLSC$(b, X, i-1, j)$

sinon

    PLSC$(b, X, i, j-1)$

is

Théorème 2

La procédure Longueur-PLSC a une complexité en $\mathcal{O}(mn)$, et la procédure PLSC a une complexité en $\mathcal{O}(m+n)$.

Démonstration. Dans Longueur-PLSC, on a deux boucles "pour" imbriquées, et dans ces boucles des opérations en $\mathcal{O}(1)$. D'où la complexité.

Dans PLSC, on réalise un "chemin direct" dans notre matrice $(i+1) \times (j+1)$, et donc ce chemin ne peut avoir plus de $i+j$ étapes. □

EXEMPLE – On cherche la plus longue sous-séquence commune aux chaînes BDCABA et ABCBDAB. Le tableau construit par Longueur-PLSC est :

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
2	B	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖	2 ←
3	C	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑	2 ↑
4	B	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖	3 ←
5	D	0	1 ↑	2 ↖	2 ↑	2 ↑	3 ↑	3 ↑
6	A	0	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	4 ↖
7	B	0	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑

On suit les flèches (chemin grisé) et on note les lettres correspondant aux flèches obliques : BCBA.

↳ utile pour mesurer la PLSC dans des brins d'ADN (c'est une mesure).

Exemple d'analyse LR(0)

Mathias Millet

2014

Introduction

Analyse syntaxique : à partir d'un mot, on souhaite :

- savoir s'il appartient au language d'une grammaire donnée.
- reconstruire l'arbre syntaxique qui l'a engendré.

Définitions préliminaires et conventions : Soit $G = (V_N, V_T, P, S)$ une grammaire non-contextuelle, c'est à dire que l'on a :

- V_N est l'ensemble des symboles non-terminaux.
- V_T est l'ensemble des symboles terminaux.
- P est l'ensemble des productions de la grammaire, avec $P \subset V_N \times (V_N + V_T)^*$.
- $S \in V_N$ est l'axiome de la grammaire.

On utilisera les conventions suivantes :

- Les éléments de V_N seront représentés par les lettres $A, B, C, D, ...$ et S
- Les éléments de V_T seront représentés par les lettres $\mathrm{a}, \mathrm{b}, \mathrm{c}, ...$; les éléments de V_T^* par les lettres $\mathrm{u}, \mathrm{v}, \mathrm{w}, ...$
- Les éléments de $V_N \cup V_T$ seront représentés par les lettres $X, Y, ...$; les éléments de $(V_N \cup V_T)^*$ par les lettres $\alpha, \beta, ...$ Le mot vide sera représenté par ϵ.
- Un élément (A, α) de P sera noté $A \to \alpha$.

Analyse syntaxique ascendante

L'analyse ascendante (avec lecture de gauche à droite) fonctionne de la manière suivante :

- On garde en mémoire un mot α sur $V_N \cup V_T$, qui correspondra à la partie déjà reconnue.
- On peut alors
  - Soit lire un caractère supplémentaire du mot d'entrée, auquel cas on ajoute ce caractère à α
  - Soit identifier le suffixe de α à la partie droite δ d'une production de la grammaire $D \to \delta$, auquel cas on remplace δ par la partie gauche D associée.

Cela correspond à l'algorithme non-déterministe ci-dessous :

Algorithme 1 : Analyse ascendante()

Entrées : Grammaire G, mot w

$\alpha \longleftarrow$ ""

$i \longleftarrow 0$

Tant que $\alpha \neq S$ *et* $i \leq |w|$ **:**

  Choisir :

    soit *Lecture*

      $\alpha \longleftarrow \alpha \mathrm{w}_i$

      $i \longleftarrow i+1$

    soit *Réduction*

      Choisir $(\delta_0, D_0, \beta_0) \in \{(\delta, D, \beta)$ tels que $\alpha = \beta\delta$ et $(D \to \delta) \in P\}$

      $\alpha \longleftarrow \beta_0 D_0$

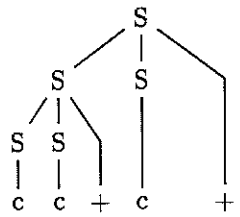
Exemple

Nous illustrerons ce développement à l'aide de la grammaire suivante :

$$G_0 = (\{S\}, \{c, +\}, \{S \to SS+, S \to c\}, S)$$

qui représente les expressions arithmétiques de l'opérateur + en notation post-fixée.

Figure 1: Arbre syntaxique correspondant au mot $cc+c+$



Automate des items

Définition (Item) Les items de la grammaire G sont les triplets (A, α, β) tels que $A \to \alpha\beta$ est une production de P. On notera $[A \to \alpha \bullet \beta]$ un tel item.

Un item $[A \to \alpha \bullet]$ sera dit complet.

À partir des items d'une grammaire, on considèrera l'automate non déterministe $AFN_G = (Q, V_N \cup V_T, \Delta, Q_0, F)$ suivant, appellé automate des items de la grammaire :

- Chaque état de Q correspond à un item de la grammaire.
- L'alphabet d'entrée est constitué des terminaux et des non-terminaux.
- La fonction de transition Δ est telle que
  - $\Delta([A \to \alpha \bullet B\beta], \epsilon) = \{[B \to \bullet\gamma]$ où $B \to \gamma$ est une production$\}$
  - $\Delta([A \to \alpha \bullet X\beta], X) = \{[A \to \alpha X \bullet \beta]\}$

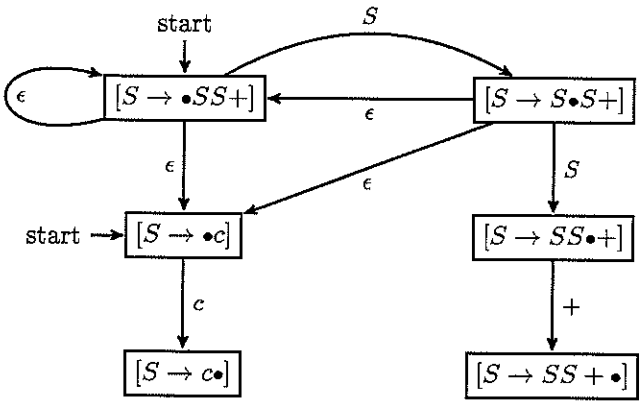


Figure 2: Automate des items de la grammaire G_0

- L'ensemble des états initiaux est $Q_0 = \{[S \to \bullet\alpha], S \to \alpha \in P\}$.
- Tous les états sont finaux : $F = Q$.

Théorème Soit $\gamma \in (V_N \cup V_T)^*$, on a, pour tout item $[A \to \alpha\bullet\beta]$ tel que α est un suffixe de γ (avec δ tel que $\gamma = \delta\alpha$):

$$[A \to \alpha\bullet\beta] \in \Delta(q_0, \delta\alpha) \quad \text{ssi} \quad \exists \text{w tels que } S \overset{*}{\Rightarrow}_d \delta A\text{w} \Rightarrow \delta\alpha\beta\text{w}$$

Autrement dit, la lecture d'un mot γ par l'automate nous amène dans les états correspondants aux items $[A \to \alpha\bullet\beta]$ où α est un suffixe de γ. Cela correspond au fait que l'on est en train d'essayer d'associer un suffixe de γ avec la partie droite de $(A \to \alpha\beta)$, et que α a déjà été reconnu. Si tout $\beta = \epsilon$, toute la partie droite de la règle est reconnue, et on pourra alors réduire en A, comme on le verra dans la partie suivante.

Idée de la démonstration

$\boxed{\Rightarrow}$ Induction sur la longueur du chemin.

$\boxed{\Leftarrow}$ On montre que $[A \to \bullet\alpha\beta] \in \Delta(q_0, \delta)$ par induction sur la longueur de la dérivation $S \overset{*}{\Rightarrow}_d \delta A\text{w} \Rightarrow_d \delta\alpha\beta\text{w}$

Automate de la grammaire

On a construit dans la section précédente un automate fini (non déterministe) reconnaissant les préfixes viables des proto-phrases de la grammaire (ce qui montre au passage que le language des préfixes viables est régulier). On va ici construire un automate à pile qui reconnaîtra exactement le langage de la grammaire.

Première étape On commence par déterminiser l'automate précédent. On obtient ainsi $AFD_G = (\bar{Q}, V_N \cup V_T, \bar{\Delta}, q_0)$. On notera $q_0, ..., q_n$ les éléments de $\bar{Q}$.

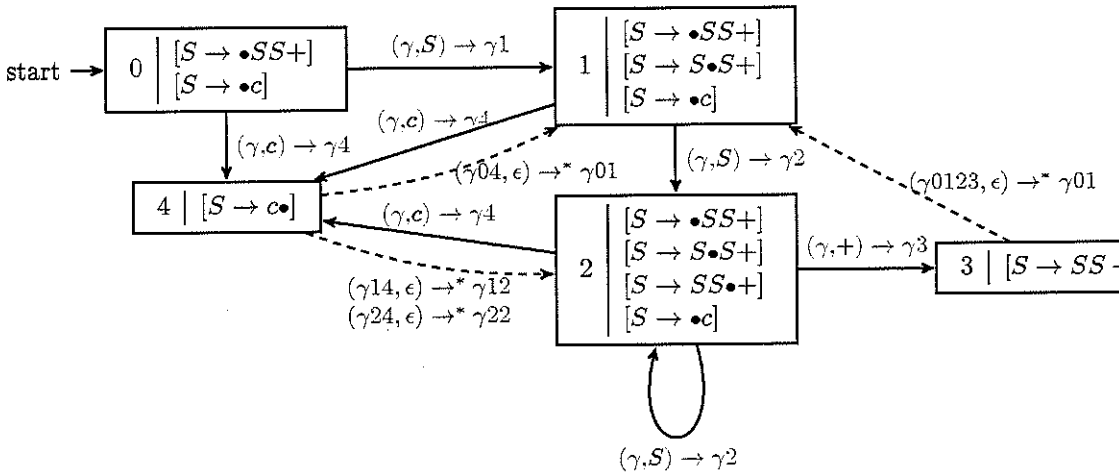


Figure 3: Automate à pile $\mathcal{A}_{G_0}$ correspondant à la grammaire G_0

Deuxième étape On ajoute les opérations de pile Λ suivantes:

- Pour toute transition $q_i \xrightarrow{a} q_j \in \bar{\Delta}$, $\quad (\gamma, q_i) \xrightarrow{a} (\gamma q_i, q_j) \in \Lambda$.

  Ces transitions, dites de lecture, servent à garder la trace des lectures effectuées, de manière à pouvoir associer au mot de pile un mot de $(V_N \cup V_T)^*$.

- Aux états contenant des items complets, on ajoute les transitions de pile suivantes :

  Si $[A \to \alpha\bullet] \in q_i$, alors il existe un préfixe viable γ tel que $S \overset{*}{\Rightarrow}_d \gamma A\mathrm{w} \Rightarrow_d \gamma\alpha\mathrm{w}$. Soit alors $q_j = \bar{\Delta}(\gamma A, q_0)$, on ajoute une série de transitions telles que$(\gamma\alpha, q_i) \xrightarrow{\epsilon}{}^{*} (\gamma, q_j)$ à Λ (on ne peut pas le faire en une seule transition car un automate à pile ne peut dépiler qu'un symbole par transition).

  Ces transitions, dites de réduction, correspondent à la reconnaissance de la phrase α comme étant la dérivation du symbole A.

Ainsi, A_G va lire son mot d'entrée jusqu'à se trouver dans un état contenant un item complet, auquel cas il pourra réduire. Si $\mathcal{A}_G$ a reconnu xy, sa pile contient exactement xy; s'il se trouve dans un état contenant $[A \to \mathrm{y}\bullet]$, on pourra réduire y en A, auquel cas la pile contiendra maintenant xA. On pourra alors continuer les lecture, ou réduire, jusqu'à épuiser l'entrée. Si alors le mot correspondant à l'état de la pile est exactement l'axiome, c'est à dire si l'on a réussi à réduire le mot d'entrée en l'axiome de la grammaire, alors il sera reconnu comme un élément de $L(G)$.

LR(0) Enfin, la grammaire G sera dite LR(O) si l'automate à pile obtenu est déterministe, c'est à dire qu'il n'y aura aucun conflit entre deux transitions de l'automate.

Les conflits peuvent être de deux ordres :

- *Lecture-réduction* Lorsque l'automate se trouve dans un état contenant un item complet $[A \to \alpha\bullet]$ et une transition par lecture d'un terminal $[B \to \beta\bullet b\beta'] \xrightarrow{b} [B \to \beta b\bullet\beta']$, si le caractère suivant du mot d'entrée est un b, alors l'automate devra faire un choix non-déterministe.

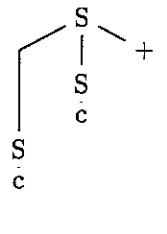
Mot d'entrée	arbre syntaxique	état de la pile
		01
+		0123
+		012
c+		014
c+		01
cc+		04
		0

Figure 4: Reconnaissance du mot $cc+$ par l'automate $\mathcal{A}_{G_0}$.

- *Réduction-réduction* Lorsque l'automate se trouve dans un état contenant deux items complets, l'automate devra choisir de manière non déterministe entre les deux réductions.

Si aucun de ces cas ne se produit, ce qui peut se vérifier facilement, alors l'automate aura un fonctionnement déterministe. Dans ce cas, la construction de $\mathcal{A}_G$ nous donne une procédure effective pour savoir si G est $LR(0)$, et, le cas échéant, nous permet de reconnaître les mots de $L(G)$ de manière efficace.

Références

- J. E. Hopcroft, J. D. Ullman *Introduction to automata theory, languages and computation* Seule source (presque) sérieuse et (presque) claire trouvée pour traiter LR.
- Willhelm, Maurer *Compilers* Trop technique, trop peu clair.
- Aho, Lam, Sethi, Ullman *Compilers* Pas assez formel, mais fournit une partie de l'intuition.