Représentation efficace des grammaires DOP

Patrick Roux, 7^e semestre de mathématiques

13 février 2003

Table des matières

1 Introduction

Ce projet se situe dans le cadre du traitement automatique du langage naturel.
Il s'agit en quelque sorte de la suite d'un travail effectué pas Romain Vinot dans [1]. Le projet a été réalisé en collaboration avec Jean-Cédric Chappelier, enseignant à l'EPFL.

Plus précisément, on s'intéresse à l'analyse syntaxique de phrases via des grammaires d'arbres. Dans ce contexte, l'objectif principal est d'améliorer sensiblement la représentation des règles DOP.
En effet, ces dernières sont pour l'instant représentées de manière relativement basique. Le problème est que pour des corpus de taille élevée, ces représentations prennent très rapidement des proportions importantes et provoquent des problèmes de représentation mémoire.
Dans un deuxième temps, le but sera de supprimer quelques imperfections, telle que la surreprésentation de certaines parties de la grammaire, par exemple.

Les buts concrets de cette étude sont donc les suivants :

– diminuer la taille de la représentation interne d'arbres d'analyse
– supprimer certaines règles grammaticales présentes en plusieurs exemplaires dans les fichiers

– améliorer l'algorithme de comparaison d'arbres

Ces trois points sont développés plus en détails en section 5.

2 Notions de base

Présentons brièvement les aspects théoriques fondamentaux nécessaires à une bonne compréhension du rapport.

2.1 Langage

Soit un alphabet Σ (ensemble de symboles, de caractères). On peut définir un langage $\mathcal{L} \subset \Sigma^*$ sur cet alphabet.

On note :

$$\begin{array}{lcl} \Sigma^* & = & \Sigma \times \Sigma \times \dots \\ & = & \text{ensemble des chaînes de caractères (fines ou infinies) de } \Sigma \\ a^* & = & \{a\}^* \\ & = & \{a^n \mid n = 0, 1, 2, \dots, \infty\} \\ & = & \{\epsilon, a, aa, aaa, \dots\} \\ \epsilon & = & \text{chaîne vide} \end{array}$$

Il existe plusieurs catégories de langage :

– fini
– régulier
– hors contexte (cadre de ce rapport)
– dépendant du contexte
– non décidable

Chomsky a démontré que chaque catégorie de langage était équivalente à une grammaire particulière.

2.2 Grammaire

Formellement, une grammaire G est un quadruplet (T, NT, S, R) où :

– $T \subset \mathcal{L}$ est l'ensemble des symboles terminaux
– $NT \subset \mathcal{L}$ est l'ensemble des symboles non-terminaux ($V = T \cup NT$ est appelé vocabulaire)
– $S \in NT$ est le symbole de plus haut niveau
– $R \subset V^* \times V^*$ est l'ensemble des règles de réécriture

Une grammaire contient donc des mots ou caractères (les terminaux), des entités servant à construire les phrases (les non-terminaux), ainsi que des règles transformant une séquence de caractères en une autre.

2.2.1 Exemple

Posons :

T = {chat, Le, mange}
NT = {P, GN, GV, N, V, Det}
(P = phrase (symbole de plus haut niveau), GN = groupe nominal, GV = groupe verbal, N =

nom, V = verbe, Det = déterminant)
$R = \{\text{GN} \longrightarrow \text{Det} \quad \text{N}\}$

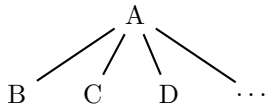
Alors, (T, NT, P, R) est une grammaire.

2.3 Arbre d'analyse

Toute règle de réécriture du type

$$\text{R} : \quad \text{A} \longrightarrow \text{B} \quad \text{C} \quad \text{D} \quad \ldots$$

où A, B, C, D, ... sont des éléments du vocabulaire, peut toujours être représentée sous la forme d'un arbre d'analyse :



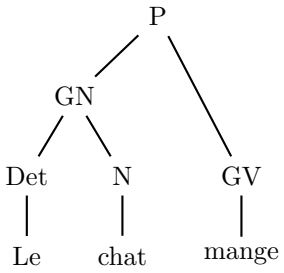
A est la racine de l'arbre, et B, C, D, ... sont ses feuilles.

2.3.1 Exemple

Considérons la même grammaire que précédemment munies des règles de réécriture suivantes :

(i) $\text{P} \longrightarrow \text{GN} \quad \text{GV}$
(ii) $\text{GN} \longrightarrow \text{Det} \quad \text{N}$
(iii) $\text{Det} \longrightarrow \text{Le}$
(iv) $\text{N} \longrightarrow \text{chat}$
(v) $\text{GV} \longrightarrow \text{mange}$

En combinant ces cinq règles, on obtient l'arbre suivant :



2.4 Dérivation

Si l'on reprend l'exemple précédent, on constate que l'on peut exprimer l'arbre d'analyse à l'aide de la suite de règles (i), (ii), (iii), (iv), (v). Toutefois, il faut veiller à ce que cette suite soit cohérente. Pour que ce soit le cas, le membre de gauche de chaque règle doit être égal au non-terminal le plus à gauche de l'arbre en construction. Dans ce cas-là, la suite des règles est appellée dérivation.
Formellement, une dérivation est donc une suite de règles issues d'une grammaire qui a la propriété que l'on réécrit toujours le non-terminal le plus à gauche de l'arbre d'analyse.

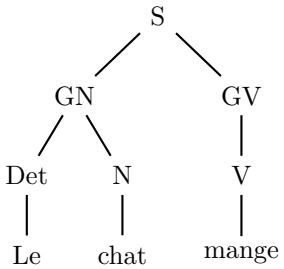
2.4.1 Exemple

R_1	:	S	$\longrightarrow$	GN GV
R_2	:	GN	$\longrightarrow$	Det N
R_3	:	Det	$\longrightarrow$	Le
R_4	:	N	$\longrightarrow$	chat
R_5	:	GV	$\longrightarrow$	V
R_6	:	V	$\longrightarrow$	mange

On a alors par exemple la dérivation :

S	$\mid R_1$
GN GV	$\mid R_2$
Det N GV	$\mid R_3$
Le N GV	$\mid R_4$
Le chat GV	$\mid R_5$
Le chat V	$\mid R_6$
Le chat mange	

Arbre d'analyse :



Par convention, l'arbre se construit toujours de gauche à droite.

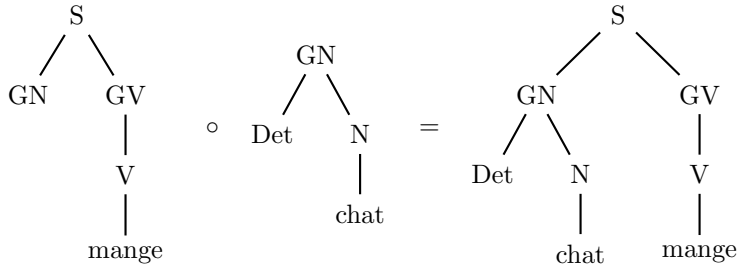
2.5 Substitution d'arbres

Introduisons un opérateur $\circ$ de substitution d'arbres :

$$T_3 = T_1 \circ T_2,$$

où les T_i sont des arbres tels que la racine de T_2 (notée x) est le non-terminal le plus à gauche de T_1. T_3 est alors l'arbre obtenu par substitution de x par T_2.

2.5.1 Exemple



3 Grammaires particulières

Voyons maintenant quelques exemples de grammaires utilisées en pratique dans l'analyse syntaxique de phrases.

3.1 Grammaires hors contexte

3.1.1 Grammaires CFG

Une grammaire CFG (*context free grammar*) est caractérisée par le fait que toutes les règles qui la composent sont du type $R \subset NT \times V^*$, c'est-à-dire de la forme :

$$\mathrm{NT} \longrightarrow \mathrm{A} \quad \mathrm{B} \quad \mathrm{C} \quad \ldots$$

où NT est un non-terminal et où A, B, C, ... sont des éléments du vocabulaire de la grammaire considérée.

3.1.2 Grammaires SCFG

Si, de plus, on probabilise les règles d'une grammaire CFG de sorte que

$$\sum_{\alpha : X \to \alpha \in R} P(X \to \alpha) = 1 \quad , \quad \forall\, X \in NT,$$

on obtient une grammaire SCFG (*stochastic context free grammar*). Si l'on considère alors une dérivation d, on aura :

$$P(\mathrm{d}) = \prod_{\mathrm{r}_i \in \mathrm{d}} P(\mathrm{r}_i)$$

où r_i est une règle particulière de la dérivation d.

Cette probabilisation est utile pour déterminer quelle analyse d'une phrase donnée est la plus probable.

3.2 Grammaires STSG

3.2.1 Définition

Une grammaire STSG (*stochastic tree substitution grammar*) n'est rien d'autre qu'une grammaire SCFG possédant des règles supplémentaires. Ces règles correspondent en fait à des arbres de profondeur supérieure à un.
Comme pour les grammaires SCFG, toutes les règles sont probabilisées de sorte que :

$$\sum_{\alpha : X \to \alpha \in R} P(X \to \alpha) = 1 \quad , \quad \forall\, X \in NT$$

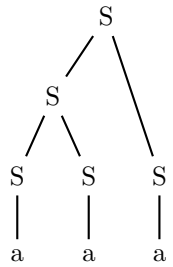
3.2.2 Différence avec les grammaires SCFG

La grande différence entre les grammaires SCFG et STSG est la suivante :

En SCFG, on a une stricte équivalence entre arbre d'analyse et dérivation, c'est-à-dire qu'à un arbre d'analyse correspond exactement une dérivation, et vice-versa.
Par exemple, considérons la grammaire suivante :

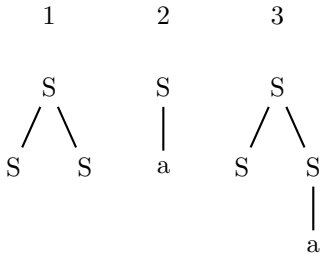
1 : S $\longrightarrow$ S S
2 : S $\longrightarrow$ a

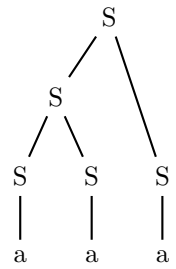
On a alors :



$\longleftrightarrow$ 1 1 2 2 2

Par contre, on n'a plus cette équivalence avec des STSG. Un arbre d'analyse va pouvoir correspondre à plusieurs dérivations différentes, comme c'est le cas dans l'exemple suivant :





$\longrightarrow$

Plusieurs dérivations possibles :

1 1 2 2 2
3 1 2 2
1 3 2 2
3 3 2

Étant donné une phrase, on va chercher l'arbre d'analyse de probabilité maximale, qui ne correspondra pas forcément à la dérivation la plus probable, puisqu'il n'y a plus d'équivalence entre les deux.

On distingue alors deux méthodes de résolution :

1. la méthode MPP (*most probable parse*), qui cherche l'arbre d'analyse le plus probable. L'inconvénient de cette méthode est sa complexité, qui est en général exponentielle.
2. la méthode MPD (*most probable derivation*), qui détermine la dérivation la plus probable. Sa complexité étant polynomiale par rapport à la taille de la phrase considérée, elle est asymptotiquement plus rapide que la précédente.

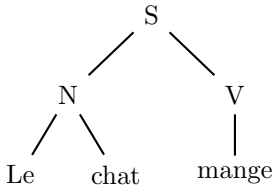
3.3 Grammaires DOP

3.3.1 Définition

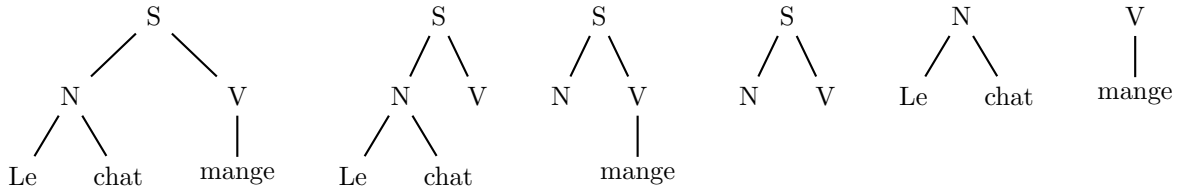
Une grammaire DOP (*data oriented parsing*) est une grammaire STSG particulière.
Considérons un corpus donné, c'est-à-dire dans notre cas un ensemble d'arbres syntaxiques. Les règles de la grammaire DOP correspondent alors à tous les sous-arbres de profondeur non nulle que l'on peut extraire de ce corpus.

3.3.2 Exemple

Si le corpus est composé de l'arbre



l'ensemble des règles de la grammaire DOP est le suivant :



3.4 Lexique

Un lexique est une liste de terminaux d'une grammaire.

Du point de vue informatique, ils sont représentés par différents champs (graphie, part of speech, probabilité, ...). Dans l'implémentation SlpToolkit (voir [4]), le champ graphie peut contenir soit des chaînes de caractères, soit des chaînes d'entiers.

Les grammaires CFG contiennent :

- un lexique de conversion $NT \leftrightarrow n$, où NT est un non-terminal et n un entier
- un lexique de chaînes d'entiers qui explicite toutes les règles de la grammaire

Les grammaires DOP sont quant à elles composées :

- d'une grammaire CFG standard
- d'un lexique d'entiers qui explicite le contenu des arbres de la grammaire
- d'un lexique exprimant les arbres DOP sous la forme racine - feuilles

4 Représentation informatique des grammaires DOP

La librairie SlpToolkit [4] est spécialement adaptée à l'implémentation de grammaires. En voici brièvement les bases.

4.1 Conventions de codage informatique

4.1.1 Non-terminaux

Les non-terminaux sont séparés en deux catégories :

1) les "pré-terminaux", c'est-à-dire les non-terminaux qui sont le membre de gauche d'une règle du type $NT \longrightarrow w$, où w est un terminal
2) les "non-pré-terminaux", autrement dit les non-terminaux n'appartenant pas à la première catégorie

Les non-terminaux sont codés par des entiers positifs de la manière suivante :

catégorie	codes
1	$1, 2, 3, 4, \ldots, n_1$
2	$n_2, n_2 + 1, n_2 + 2, \ldots$

Le premier entier de la seconde catégorie, n_2, est fixé par défaut à 512, mais peut être modifié. La seule contrainte est qu'il soit supérieur à n_1, puisque les catégories 1 et 2 sont bien distinctes.
De plus, n_2 est attribué par convention au symbole de plus haut niveau.

4.1.2 Terminaux

Les terminaux sont codés par des entiers positifs dans le lexique qui les convertit.
Toutefois, lorsqu'ils sont présents en tant que membre de droite d'une règle grammaticale (ils ne peuvent jamais être membre de gauche), on les représente par l'opposé de leur code lexical, afin de ne pas les confondre avec les pré-terminaux.

4.1.3 Arbre vide

Dans le cadre de l'analyse, il est indispensable de pouvoir utiliser l'arbre vide lorsque l'on ne veut pas réécrire le non-terminal le plus à gauche.
En effet, si l'on considère le cas

alors l'arbre t, qui est un sous-arbre de T, fait partie de la grammaire DOP constituée sur la base de l'arbre T. Ainsi, pour représenter t à l'aide d'arbres de profondeur un, on va forcément avoir besoin de l'arbre vide pour indiquer que l'on ne réécrit pas le non-terminal A.

Le code de l'arbre vide est par convention -1, le caractère 0 étant déjà réservé pour indiquer la fin d'une chaîne.

4.1.4 Règles

Il existe deux types de règles grammaticales :

Tout d'abord, il y a les règles lexicales, c'est-à-dire les règles du type $PT \longrightarrow w$, où PT est un pré-terminal et w un terminal.
Ces règles sont codées par l'opposé du code lexical de w auquel on soustrait une unité. Plus clairement, si w est codé par l'entier $n > 0$ dans le lexique de conversion des terminaux, alors la règle $PT \longrightarrow w$ est codée par -n-1, le -1 provenant du fait que la règle -1 est déjà utilisée pour désigner l'arbre vide.

Toutes les autres règles, qui ne contiennent donc aucun terminal, sont codées par des entiers strictement positifs.

4.1.5 Résumé du codage

Notons au passage qu'un niveau de l'implémentation, tous les entiers sont du type *longint*, c'est-à-dire que l'on travaille modulo N (N = 4'294'967'296). Ainsi, tout entier négatif m est considéré égal à N-m.

Voici donc la manière dont sont codés les caractères dans les règles de la grammaire :

	codes
Pré-terminaux	1, 2, 3, ..., n_1
Symbole de plus haut niveau	n_2
Non-pré-terminaux	n_2, $n_2 + 1$, ..., n_3
Terminaux	n_4, $n_4 + 1$, ..., N-2, N-1

Remarques :

1/ $n_2 > n_1$ ($n_2 = 512$ par défaut)

2/ si $n_3 \geq n_4$, on est vraiment embêté ... (en général, on a tout de même assez de marge)
ordres de grandeur : $n_3 \leq 10'000$, $n_4 \geq 4'294'300'000$)

Le codage des règles est quant à lui le suivant :

	codes
Règles non lexicales	1, 2, 3, ..., m_1
Règles lexicales	n_4-1, n_4, $n_4 + 1$, ..., N-2
Arbre vide	N-1

4.2 Éléments de la grammaire

Adoptons dès à présent les notations suivantes, qui sont celles en vigueur dans les programmes existants :

S	:	symbole de plus haut niveau
:n	:	pré-terminal quelconque ($n \in \mathbb{N}$)
a	:	terminal quelconque
"a"	:	pré-terminal fictif du terminal a (utilisé pour représenter les règles lexicalisées - i.e du type $X \longrightarrow w$, avec X pré-terminal et w terminal - sans mélanger non-terminaux et terminaux dans les règles de la grammaire)

Avant la réalisation du projet, la grammaire DOP était concrètement composée des éléments suivants :

4.2.1 Grammaire CFG

(1) **grammaire***gramCFG.slpgram :**
Fichier de configuration synthétisant (2) et (3) de manière à les rendre plus lisibles. En fait, c'est la liste des règles CF de la grammaire dont le membre de droite ne contient pas de terminal.
notation : NT $\longrightarrow$ NT* (proba)
exemple : S $\longrightarrow$ A :1 (0.5)
signification : S est réécrit par A et :1

(2) **grammaire***gramCFG-conv.slplex :**
Table de conversion des non-terminaux en nombres entiers
notation : NT | $\mathbb{N}$
exemple : S | 512
signification : S est codé par l'entier 512

(3) **grammaire***gramCFG-gram.slplex :**
Contient tous les sous-arbres de profondeur un du corpus (sous la forme feuilles - racine) munis de leur probabilité, qui est en fait leur fonction de fréquence
notation : $\mathbb{N}^*$ | $\mathbb{N}$ | proba
exemple : 513 1 | 512 | 0.5
signification : 513 et 1 (autrement dit A et :1) réécrivent 512 (S)

4.2.2 Grammaire DOP

(4) **grammaire***gramDOP.slpgram :**
Fichier de configuration contenant (6), c'est-à-dire tous les sous-arbres du corpus, représentés en omettant tous les noeuds internes. Il sont donc de la forme racine - feuilles. Leur probabilté correspond à leur fonction de fréquence
notation : NT $\longrightarrow$ NT* (proba)
exemple : S $\longrightarrow$ C D (0.02)
signification : S est réécrit par C et D

(5) **grammaire***lexCFG_DOP.slplex :**
Contient les dérivations CF de chaque arbre de (6), c'est-à-dire la suite des règles de la grammaire CFG (3) aboutissant à l'arbre en question, en notation codée
notation : $\mathbb{N}^*$ | nombre d'occurences
exemple : 2 4 9 | 3
signification : la suite de règles CF 2, 4 et 9 aboutissent à la règle "S va sur C et D"

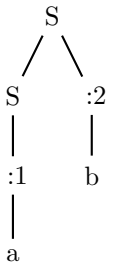
(6) **grammaire***lexDOP.slplex :**
Contient tous les sous-arbres du corpus sous la forme feuilles - racine
notation : $\mathbb{N}^*$ | $\mathbb{N}$ | proba
exemple : 515 516 | 512 | 0.02
signification : 515 et 516 (autrement dit C et D) réécrivent 512 (S)

4.2.3 Lexique

(7) **lexique***.slplex :**
lexique de conversion des terminaux en nombres entiers, sous la forme graphie (terminal) - part of speech - proba
notation : T | $\mathbb{N}$ | proba
exemple : a | 1 | 1
signification : le terminal a est codé par l'entier 1

4.3 Exemple

Considérons le corpus composé du seul arbre suivant :



Les éléments de la grammaire sont alors :

(1) S ⟶ S :2 (0.5)
S ⟶ :1 (0.5)

(2) S | 512

(3) 512 2 | 512 | 0.5
1 | 512 | 0.5

(4) S ⟶ “a” (0.125)
S ⟶ :1 (0.125)
S ⟶ “a” “b” (0.125)
S ⟶ “a” :2 (0.125)
S ⟶ :1 “b” (0.125)
S ⟶ :1 :2 (0.125)
S ⟶ S “b” (0.125)
S ⟶ S :2 (0.125)

(5)

2	4’294’967’294			1
2	4’294’967’295			1
1	2	4’294’967’294	4’294’967’293	1
1	2	4’294’967’294	4’294’967’295	1
1	2	4’294’967’295	4’294’967’293	1
1	2	4’294’967’295	4’294’967’295	1
1	4’294’967’295	4’294’967’293		1
1	4’294’967’295	4’294’967’295		1

n.b : 1 = règle CF nř1 dans (1)
2 = regle CF nř2 dans (1)
-1 = arbre vide (codé en *longint* par 4’294’967’295)
-2 = règle lexicale :1 ⟶ “a” (codée en *longint* par 4’294’967’294)
-3 = règle lexicale :2 ⟶ “b” (codée en *longint* par 4’294’967’293)

(6)

4'294'967'295			512	0.125
1			512	0.125
4'294'967'295	4'294'967'294		512	0.125
4'294'967'295	2		512	0.125
1	4'294'967'294		512	0.125
1	2		512	0.125
512	4'294'967'294		512	0.125
512	2		512	0.125

n.b :

1	=	pré-terminal :1
2	=	pré-terminal :2
-1	=	terminal "a" (codé en *longint* par 4'294'967'295)
-2	=	terminal "b" (codé en *longint* par 4'294'967'294)
512	=	noeud interne S

(7)

a	1	1
b	2	1

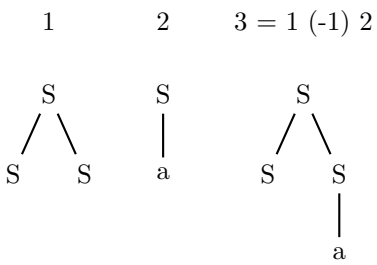
5 Objectifs

Intéressons-nous aux objectifs du projet énoncés en introduction.

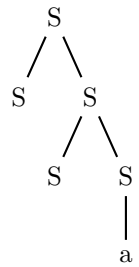
5.1 Représentation des règles DOP

La première tâche à réaliser est d'améliorer le lexique (5), en incluant des arbres de profondeur supérieure à un dans la représentation des arbres élémentaires, ceci afin de diminuer la taille de certaines représentations.

Par exemple, si l'on considère la grammaire

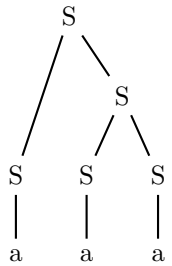


on préférerait avoir la représentation 1 (-1) 3 plutôt que 1 (-1) 1 (-1) 2 (qui est la représentation actuelle) pour l'arbre

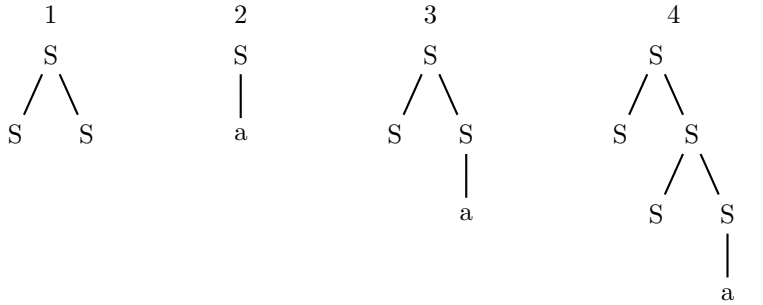


Soit d une dérivation décrivant l'arbre $T(d)$. d peut être codée de plusieurs façons différentes. On appelle code i de d la i^e expansion du code d'un arbre, où chaque expansion consiste à remplacer le numéro du premier arbre de profondeur supérieure à un par son code.

Par exemple, si T est l'arbre d'analyse



et si les arbres suivants font partie de la grammaire :



alors d = 4 2 2 (qui est une dérivation de T) peut être codée par $c_0(d)$ = 4 2 2, $c_1(d)$ = 1 2 3 2 ou $c_2(d)$ = 1 2 1 2 2. Par convention, $c_0(d)$ est le code de d (en fait, $d = c_0(d)$) alors que $c_1(d)$ est un code de d.

5.2 Représentation multiple des règles CF

Considérons l'exemple exposé au paragraphe 4.3. On constate alors que les règles CF y sont surreprésentées. En effet, elles apparaissent :

I) dans la grammaire CFG (3) :

512	2	|	512	| 0.5
1		|	512	| 0.5

II) dans la grammaire DOP (6) :

512	2	|	512	| 0.125
1		|	512	| 0.125

III) dans les règles de dérivation (5) :

1	4'294'967'295	4'294'967'295
2	4'294'967'295	

On pourrait donc gagner de l'espace mémoire en supprimant entièrement (3), ainsi que l'équivalent des règles CFG dans (5). Elles seraient ainsi uniquement présentes dans (6).

5.3 Comparaison d'arbres d'analyse

Considérons deux dérivations d_1 et d_2, avec leur arbre respectif $T(d_1)$ et $T(d_2)$. Étant donné que nous sommes dans le contexte de grammaires DOP, deux dérivations différentes peuvent engendrer

le même arbre. On aimerait donc pouvoir facilement décider si $T(d_1) = T(d_2)$.

Il est évident que si $d_1 = d_2$, alors $T(d_1) = T(d_2)$.
Par contre, si $d_1 \neq d_2$, on doit s'intéresser aux codes de d_1 et d_2.

Démontrons la propriété suivante :

$$\begin{array}{rrcl} & T(d_1) = T(d_2) & \iff & \exists\, k, l \quad t.q \quad c_k(d_1) = c_l(d_2) \\ \text{ou :} & T(d_1) \neq T(d_2) & \iff & c_k(d_1) \neq c_l(d_2) \quad , \quad \forall\, k, l \end{array}$$

L'implication $\Rightarrow$ de la première relation est triviale : un arbre étant décomposable de manière unique en règles CF, il suffit de prendre k et l de façon à ce que c_k et c_l soient les codes CF de d_1 et d_2 respectivement (qui existent toujours puisque la grammaire est atomique).
Réciproquement, si deux dérivations peuvent être codées exactement de la même manière, il est évident qu'elles vont générer le même arbre.

Considérons le second exemple de la section 5.1.
$d_1 = 4\ 2\ 2$ et $d_2 = 1\ 2\ 3\ 2$ sont deux dérivations de T. Donc, $T(d_1) = T(d_2)$.
Toutefois, si on veut le voir grâce aux codes, on a :

code 0 de d_1 : 4 2 2
or : 4 = 1 -1 3
$\Rightarrow$ code 1 de d_1 : (1 -1 3) 2 2 $\equiv$ 1 2 3 2 (les -1 sont remplacés par les règles suivantes)
On constate alors que $c_1(d_1) = 1\ 2\ 3\ 2 = c_0(d_2)$. Donc $T(d_1) = T(d_2)$.

On aurait aussi pu énumérer tous les codes de d_1 et d_2 aussi loin que possible et comparer les derniers (qui sont les codes CF) :

$c_0(d_1) = 4\ 2\ 2$
$4 = 1\ \text{-}1\ 3 \quad \Rightarrow \quad c_1(d_1) = (1\ \text{-}1\ 3)\ 2\ 2 = 1\ 2\ 3\ 2$
$3 = 1\ \text{-}1\ 2 \quad \Rightarrow \quad c_2(d_1) = 1\ 2\ (1\ \text{-}1\ 2)\ 2 = 1\ 2\ 1\ 2\ 2 = c_{CF}(d_1)$
$c_0(d_2) = 1\ 2\ 3\ 2$
$3 = 1\ \text{-}1\ 2 \quad \Rightarrow \quad c_1(d_2) = 1\ 2\ (1\ \text{-}1\ 2)\ 2 = 1\ 2\ 1\ 2\ 2 = c_{CF}(d_2)$
$c_{CF}(d_1) = c_{CF}(d_2) \quad \Longrightarrow \quad T(d_1) = T(d_2)$

À l'heure actuelle, pour comparer deux arbres, on remonte jusqu'au code CF de leur dérivation. Toutefois, il serait plus judicieux, si cela est possible, de s'arrêter plus tôt, le problème étant de savoir à quel moment le faire ...

5.4 Adaptation au cas non-DOP

Dans ces situations, la grammaire ne comporte pas forcément tous les sous-arbres issus du corpus, contrairement au cas de DOP.

6 Représentation des règles DOP

6.1 Principe

L'algorithme d'extraction des arbres élémentaires DOP dans [1] extrait chaque arbre l'un après l'autre, et calcule directement sa représentation à l'aide des règles CF.

Le nouvel algorithme, implémenté dans la fonction *obtient_sous_arbres_racine_fixe* (6.2.1), va créer une pile avec tous les arbres extraits ayant une racine fixée. Cette pile aura l'allure suivante :

$$
\begin{array}{l}
\mathrm{T}_k^{n_k} \\
\vdots \\
\mathrm{T}_2^{n_2} \\
\vdots \\
\mathrm{T}_2^2 \\
\\
\mathrm{T}_2^1 \\
\\
\mathrm{T}_1^{n_1} \\
\vdots \\
\mathrm{T}_1^2 \\
\\
\mathrm{T}_1^1
\end{array}
$$

Tous les T_i^1 sont appelés des bases et chaque T_i^j (pour tout $\mathrm{j} \neq \mathrm{n}_i$) va être représenté à l'aide de T_i^{j+1}.
En fait, l'arbre T_i^j sera représenté par la suite

$$n \quad -1 \ldots -1 \quad m \quad -1 \ldots -1$$

où n est le numéro de l'arbre T_i^{j+1} et m celui d'une règle CF.

Pour simplifier encore plus, on va donc coder chaque arbre dans le lexique CFG-DOP par quatre entiers

$$(\ n \ , \ k_1 \ , \ m \ , \ k_2 \ )$$

où n et m sont les mêmes que précédemment, et où k_1 et k_2 sont respectivement le nombre d'arbres vides entre n et m et après m.

Ensuite, il faut bien sûr modifier la fonction qui, à partir du lexique CFG-DOP, trouve les règles DOP. Ceci est l'objet de la fonction *remplit_grammaire_DOP_modifie* (6.2.9).

Enfin, l'algorithme de comparaison d'arbres d'analyse doit lui aussi être adapté.

6.2 Algorithme

La classe *arbre_extrait* est composée des attributs suivants :

racine est un pointeur sur la racine de l'arbre

base est un booléen indiquant si l'arbre est à la base d'une pile interne, autrement dit s'il n'est pas utile à la représentation d'un autre arbre

couvert est un booléen indiquant si l'arbre peut être exprimé à l'aide d'un autre arbre

regle_CF est le numéro m

nb_zeros_avant est k_1

nb_zeros_apres est k_2

numero_lexical est le numéro de l'arbre dans le lexique

identificateur est l'indice d'extraction de l'arbre pour une racine donnée

Variables utilisées :

stack	:	pile stockant les noeuds internes de l'arbre
finale	:	pile stockant les arbres extraits
provisoire	:	pile stockant les arbres dépilés de finale
arbre_courant	:	arbre extrait (de type arbre_extrait)
nb_faux	:	entier utile pour définir la hiérarchie des arbres
nb_faux_restants	:	entier utile à la représentation finale de l'arbre extrait
premier	:	booléen indiquant si l'arbre extrait est le plus petit, c'est-à-dire le dernier extrait ou le premier au-dessus de la pile
Regles_utilisees	:	vecteur dynamique contenant la représentation d'un arbre
continu	:	booléen indiquant s'il reste des arbres à extraire (pour une racine fixée)

6.2.1 Fonction obtient_sous_arbres_racine_fixe

Cette fonction extrait, pour une racine donnée, tous les sous-arbres DOP, tel que le fait la fonction obtient_sous_arbre_racine_fixe originale dans [1]. Dans un premier temps, elle extrait les arbres et les empile de façon convenable, et procède ensuite à la représentation de chaque arbre de la pile.

Arguments : la racine R des arbres élémentaires DOP qui vont être extraits

Répéter

On crée arbre_courant de racine R
On empile tous ses noeuds non-terminaux dans stack (via la fonction descend_et_empile ([1]))
continu est FAUX

Si stack n'est pas vide
On initialise nb_faux et nb_faux_restants à 0
Répéter
On dépile next de stack
Si next est fermé
On l'ouvre
On incrémente nb_faux
Sinon
On le ferme
continu est VRAI
jusqu'à ce que stack est vide ou que next est fermé

Tant que stack est non vide
On dépile stack
Si l'élément dépilé est fermé
On incrémente nb_faux_restants

On attribue nb_faux_restants et nb_faux à nb_zeros_avant et nb_zeros_apres de arbre_courant
On lui attribue la règle CF correspondant au noeud next ouvert (via la fonction numero_regle_CF_a_appliquer (6.2.2))
Si nb_faux est nul
arbre_courant est une base
On l'empile sur finale
Sinon
On appelle la fonction empile_et_ajoute (6.2.3)
On vide provisoire dans finale

jusqu'à ce que continu soit FAUX

On dépile arbre_courant de finale
On appelle la fonction cree_code_pour_lexique_CFGDOP (6.2.6) avec premier à VRAI
On met à jour le numero_lexical des copies de arbre_courant dans finale (via la fonction met_a_jour_numero_lexical (6.2.5))
predecesseur = arbre_courant

Tant que finale est non vide
  On dépile arbre_courant de finale
  On appelle la fonction cree_code_pour_lexique_CFGDOP (6.2.6) avec premier à FAUX
  On met à jour le numero_lexical des copies de arbre_courant dans finale (via la fonction met_a_jour_numero_lexical (6.2.5))
  Si arbre_courant est une base et finale est non vide
    On dépile predecesseur de finale
  Sinon
    predecesseur = arbre_courant

On ouvre tous les noeuds internes de l'arbre (via la fonction ouvre_noeuds_internes (6.2.8))

6.2.2 Fonction numero_regle_CF_a_appliquer

Cette fonction renvoie le numéro de la règle CF associée à un noeud et à ses fils.

Arguments : un noeud

On ouvre le noeud
On cherche la règle CF associée à ce noeud et à ses fils (algorithme issu de la fonction sous_arbre_DOP_vers_liste_regles_CF de [1])
On rétablit l'ouverture initiale du noeud
On retourne le numero de la règle CF

6.2.3 Fonction empile_et_ajoute

L'arbre passé en paramètre est placé convenablement dans la pile, c'est-à-dire juste au-dessus de tous les arbres qui seront représentés grâce à lui.

Arguments : arbre_courant, nb_faux, finale et provisoire

On empile arbre_courant sur provisoire
Si l'arbre supérieur de finale n'est pas une base
  on le couvre dans toute la pile finale (via la fonction couvre_dans_toute_la_pile (6.2.4))
Si nb_faux-1 est non nul
  On transvase un arbre de finale dans provisoire
  Tant que l'arbre supérieur de finale est couvert ou est une base
    On le dépile dans provisoire
  On appelle récursivement la fonction empile_et_ajoute avec nb_faux-1

6.2.4 Fonction couvre_dans_toute_la_pile

Toutes les copies d'un même arbre dans la pile vont être rendues couvertes.

Arguments : finale, indice, nb_de_fois

On dépile finale et à chaque fois que l'élément dépilé a son identificateur égal à indice, on le rend couvert
On s'arrête lorsque l'on a couvert nb_de_fois arbres
On réempile les arbres dépilés (dans le même ordre qu'initialement)

6.2.5 Fonction met_a_jour_numero_lexical

Attribue à toutes les copies d'un arbre son numero_lexical.

Arguments : finale, indice, nb_de_fois, no_lexical

On dépile finale et à chaque fois que l'élément dépilé à son identificateur égal à indice, on rend son numero_lexical égal à no_lexical
On s'arrête lorsque l'on a traité nb_de_fois arbres
On réempile les arbres dépilés (dans le même ordre qu'initialement)

6.2.6 Fonction cree_code_pour_lexique_CFGDOP

Créé la représentation d'un arbre dans Regles_utilisees.

Arguments : predecesseur, arbre_courant, premier

Si premier est VRAI
  On augmente Regles_utilisees avec 0
Sinon
  On l'augmente avec le numero_lexical de predecesseur
On ajoute dans Regles_utilisees les valeurs k_1, m et k_2 de arbre_courant, puis le symbole de fin de chaîne
On appelle la fonction ajout_code_dans_lexique_CFGDOP (6.2.7)

6.2.7 Fonction ajout_code_dans_lexique_CFGDOP

Arguments : arbre_courant, Regles_utilisees

L'algorithme est le même que dans la fonction ajout_regle_DOP_dans_lexique_CFG_DOP ([1]), excepté que l'on rajoute deux lignes définissant l'attribut numero_lexical de arbre_courant

6.2.8 Fonction ouvre_noeuds_internes

La fonction ouvre tous les noeuds (qui ne sont pas des feuilles) de l'arbre de racine noeud_courant, qui avaient été fermés lors de l'extraction des arbres élémentaires DOP (racine est un booléen indiquant si noeud_courant est la racine de l'arbre dont on ouvre les noeuds).

Arguments : noeud_courant, racine

Si noeud_courant a un fils
  On ouvre noeud_courant
  On appelle récursivement la fonction avec le fils de noeud_courant et racine à FAUX
Si noeud_courant a un frère et racine est FAUX
  On appelle récursivement la fonction avec le frère de noeud_courant et racine à FAUX

6.2.9 Fonction remplit_grammaire_DOP_modifie

Cette fonction crée la grammire DOP, c'est-à-dire exprime chaque arbre DOP sous la forme racine - feuilles, à l'aide du lexique CFG-DOP. Chaque ligne du lexique est du type (n,k_1,m,k_2), et la fonction traduit cette écriture en règle DOP, notée $A \longrightarrow B$.

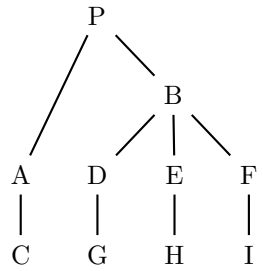
Arguments : la grammaire CFG, le lexique CFG-DOP, la grammaire DOP (vide au départ)

Pour toutes les lignes du lexique CFG-DOP
  Si $n = 0$

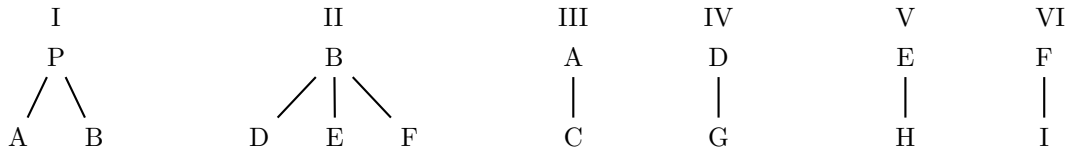
La règle considérée est la règle m
Sinon
A est égale à la partie gauche de n (déjà présent dans la grammaire DOP)
On initialise i et j à 0
Tant que i < k_1 ou que le j^e élément de la partie droite de n est terminal
On augmente B avec le j^e élément de la partie droite de n
Si le j^e élément de la partie droite de n est non terminal
On incrémente i
On incrémente j
On augmente B avec toute la partie droite de m
On incrémente j
Tant que j est inférieur au nombre d'éléments de la partie droite de n
On augmente B avec le j^e élément de la partie droite de n
On incrémente j
On ajoute la règle A ⟶ B à la grammaire DOP

6.3 Exemple

Considérons l'arbre suivant, et procédons à l'extraction et à la représentation de tous ses sous-arbres ayant P pour racine.



La grammaire CFG est la suivante :



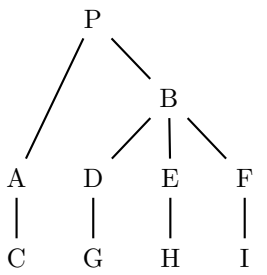
Voici la phase d'extraction (les noeuds ouverts sont marqués d'un x) :

id.	ouvertures					feuilles	stack	next	nb_faux	nb_faux_restants
	A	B	D	E	F					
1	x	x	x	x	x	CGHI	ABDEF	F	0	0
2	x	x	x	x		CGHF	ABDEF	E	1	0
3	x	x	x		x	CGEI	ABDEF	F	0	1
4	x	x	x			CGEF	ABDEF	D	2	0
5	x	x		x	x	CDHI	ABDEF	F	0	1
6	x	x		x		CDHF	ABDEF	E	1	1
7	x	x			x	CDEI	ABDEF	F	0	2
8	x	x				CDEF	ABDEF	B	3	0
9	x		x	x	x	CB	AB	A	1	0
10		x	x	x	x	AGHI	ABDEF	F	0	1
11		x	x	x		AGHF	ABDEF	E	1	1
12		x	x		x	AGEI	ABDEF	F	0	2
13		x	x			AGEF	ABDEF	D	2	1
14		x		x	x	ADHI	ABDEF	F	0	2
15		x		x		ADHF	ABDEF	E	1	2
16		x			x	ADEI	ABDEF	F	0	3
17		x				ADEF	ABDEF	B	3	1
18						AB	AB		2	0

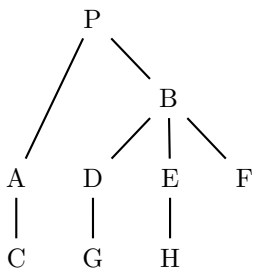
Explications :

- nb_faux correspond au nombre de noeuds dans stack que l'on ouvre avant de fermer le premier noeud ouvert que l'on rencontre (next)
- nb_faux_restants est le nombre de noeuds dans stack (après next) qui sont fermés

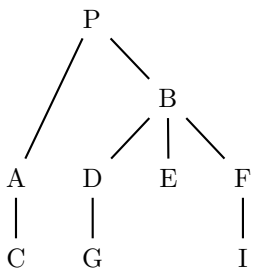
1)



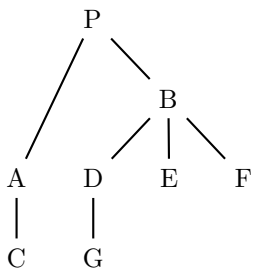
2)



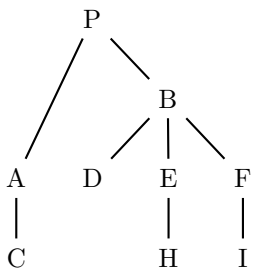
3)



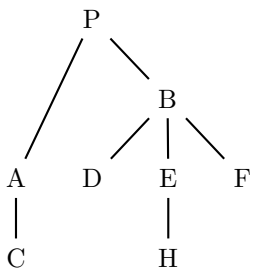
4)



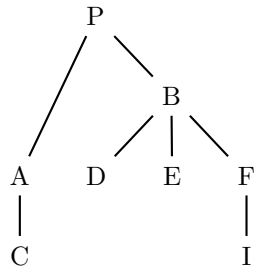
5)



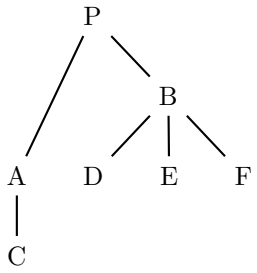
6)



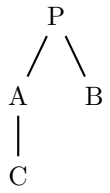
7)



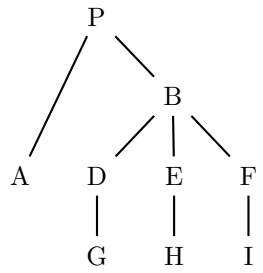
8)



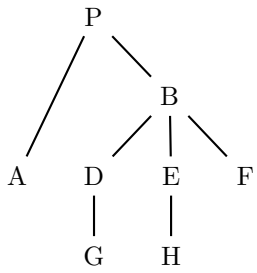
9)



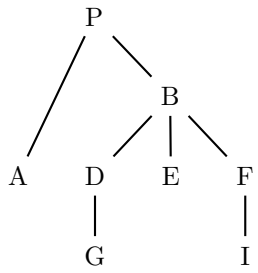
10)



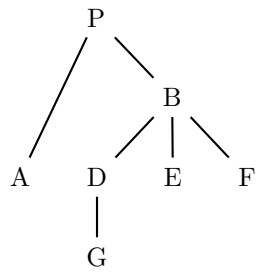
11)



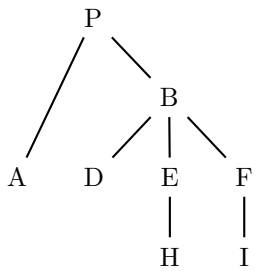
12)



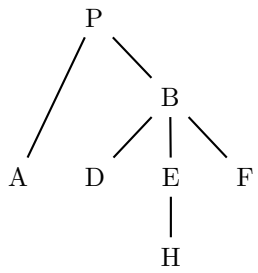
13)



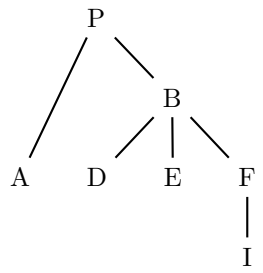
14)



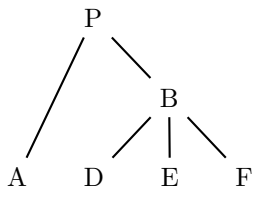
15)



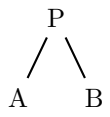
16)



17)



18)



Voici maintenant la pile finale (représentée à l'envers), avec les attributs de chaque arbre :

id.	base	couvert	nb_zeros _avant	nb_zeros _apres	regle_CF	code dans le lexique CFG-DOP
1	x		0	0	VI	(2,0,VI,0)
2		x	0	1	V	(4,0,V,1)
4		x	0	2	IV	
3	x		1	0	VI	(4,1,VI,0)
4		x	0	2	IV	(8,0,IV,2)
8		x	0	3	II	
5	x		1	0	VI	(6,1,VI,0)
6		x	1	1	V	(8,1,V,1)
8		x	0	3	II	
7	x		2	0	VI	(8,2,VI,0)
8		x	0	3	II	(9,0,II,3)
9		x	0	1	III	(18,0,III,1)
18			0	2	I	
10	x		1	0	VI	(11,1,VI,0)
11		x	1	1	V	(13,1,V,1)
13		x	1	2	IV	
12	x		2	0	VI	(13,2,VI,0)
13		x	1	2	V	(17,1,IV,2)
17		x	1	3	II	
14	x		2	0	VI	(15,2,VI,0)
15		x	2	1	V	(17,2,V,1)
17		x	1	3	II	
16	x		3	0	VI	(17,3,VI,0)
17		x	1	3	II	(18,1,II,3)
18			0	2	I	(0,0,I,2)

Remarques :

- Tout arbre que l'on rencontre juste en-dessous d'une base (17, 13, 18, 8 et 4) a déjà été représenté auparavant
- Le code est constitué de l'arbre qui précède, de nb_zeros_avant, de la règle CF et de nb_zeros_apres

- Les arbres sont injectés dans le lexique dans l'ordre inverse de leur extraction, c'est-à-dire du 18 au 1. Par conséquent, l'indice de l'arbre 18 dans le lexique est 1, celui du 17 est 2, et ainsi de suite ...
- Les règles CF I à VI sont codées par des entiers négatifs pour les règles lexicales (c'est-à-dire du type pré-terminal $\longrightarrow$ terminal) et par des entiers positifs pour les autres (voir 4.1.5). On aura donc le codage suivant :

  I : 1
  II : 2
  III : -2
  IV : -3
  V : -4
  VI : -5

  (rappel : le 0 est réservé pour la fin d'une chaîne et le -1 pour l'arbre vide)

Voici donc le lexique CFG-DOP tel qu'il est réellement :

indice	n	k_1	m	k_2
1	0	0	1	2
2	1	1	2	3
3	2	3	-5	0
4	2	2	-4	1
5	4	2	-5	0
6	2	1	-3	2
7	6	2	-5	0
8	6	1	-4	1
9	8	1	-5	0
10	1	0	-2	1
11	10	0	2	3
12	11	2	-5	0
13	11	1	-4	1
14	13	1	-5	0
15	11	0	-3	2
16	15	1	-5	0
17	15	0	-4	1
18	17	0	-5	0

Remarques :

- Le prédécesseur (n) de l'arbre 1 est 0 car c'est un arbre purement CF.
- Le problème, lors de l'implémentation, est que le zéro représente le symbole de fin de chaîne, et on ne peut donc pas l'injecter dans le lexique CFG-DOP pour désigner n, k_1 ou k_2. L'astuce utilisée est donc d'augmenter k_1 et k_2 d'une constante strictement positive, et de poser n égal à l'opposé cette même constante pour le cas où l'arbre est CF.
  Ensuite, lorsqu'on lit le lexique CFG-DOP pour construire la grammaire, il suffit d'exécuter l'opération inverse.

7 Représentation multiple des règles CF

7.1 Principe

Pour palier à la surreprésentation des règles CF, considérons la nouvelle structure *Grammaire_etendue*, composée d'une Grammaire *gram* et d'un entier *nb_regles_CF*.

Les modifications apportées au programme sont les suivantes :

Au lieu d'extraire la grammaire CFG ((3) dans 4.2), toutes les règles CF du corpus sont directement inserées au début de la grammaire DOP (6).
nb_regles_CF est alors égalisé au nombre de règles CF extraites du corpus, c'est-à-dire à la taille de la grammaire DOP à cet instant.

Ensuite, on cherche tous les arbres élémentaires DOP comme avant (la grammaire DOP jouant le rôle de la grammaire CFG), à la seule exception près que pour chaque racine, la représentation du dernier arbre extrait (qui correspond forcément à une règle CF), n'est pas injectée dans le lexique CFG-DOP (5), puisque l'arbre se trouve déjà dans la grammaire DOP.

En ce qui concerne les autres arbres, il faut faire attention lorsque l'on définit leur prédécesseur : si ce dernier est un arbre correspondant à une règle CF, il se trouve dans la grammaire DOP et son indice est celui qu'il possède dans celle-ci.
Si par contre, le prédécesseur n'est pas une règle CF, il faut aller chercher son indice dans le lexique

CFG-DOP, et l'augmenter de *nb_regles_CF*. En effet, un arbre ayant l'indice i dans le lexique CFG-DOP aura l'indice i + *nb_regles_CF* dans la grammaire DOP, puisqu'au début de cette dernière se trouvent tous les arbres CF, qui sont absents du lexique CFG-DOP.

Le lexique CFG-DOP constitué, on peut construire la suite de la grammaire DOP normalement.

7.2 Exemple

Pour le même exemple que précédemment, voici le lexique CFG-DOP :

indice	n	k_1	m	k_2
1	1	1	2	3
2	3	3	-5	0
3	3	2	-4	1
4	5	2	-5	0
5	3	1	-3	2
6	7	2	-5	0
7	7	1	-4	1
8	9	1	-5	0
9	1	0	-2	1
10	11	0	2	3
11	12	2	-5	0
12	12	1	-4	1
13	14	1	-5	0
14	12	0	-3	2
15	16	1	-5	0
16	16	0	-4	1
17	18	0	-5	0

On remarque que la première ligne a disparu, puisqu'elle représentait un arbre CF.

De plus, tous les n (excepté le premier) ont été augmenté d'une unité à cause de la règle CF B $\longrightarrow$ D E F, qui est présente au début de la grammaire DOP. Pour être exact, n a d'abord été diminué de un suite à la disparition de la première ligne du lexique CFG-DOP, puis a été augmenté de deux, qui est le nombre de règles CF de la grammaire.

Voici maintenant la grammaire DOP, constituée de tous les arbres élémentaires DOP (les arbres CF en premier) sous la forme racine - feuilles.

indice	graphie	morpho
1	1 513	512
2	2 3 4	513
3	1 2 3 4	512
4	1 2 3 -4	512
5	1 2 -3 4	512
6	1 2 -3 -4	512
7	1 -2 3 4	512
8	1 -2 3 -4	512
9	1 -2 -3 4	512
10	1 -2 -3 -4	512
11	-1 513	512
12	-1 2 3 4	512
13	-1 2 3 -4	512
14	-1 2 -3 4	512
15	-1 2 -3 -4	512
16	-1 -2 3 4	512
17	-1 -2 3 -4	512
18	-1 -2 -3 4	512
19	-1 -2 -3 -4	512

codes	:	terminaux	:	C	$\longleftrightarrow$	-1
				G	$\longleftrightarrow$	-2
				H	$\longleftrightarrow$	-3
				I	$\longleftrightarrow$	-4
		pré-terminaux	:	A	$\longleftrightarrow$	1
				D	$\longleftrightarrow$	2
				E	$\longleftrightarrow$	3
				F	$\longleftrightarrow$	4
		non-pré-terminaux	:	P	$\longleftrightarrow$	512
				B	$\longleftrightarrow$	513

8 Autres petites modifications

8.1 Phase d'analyse

Lors de la phase d'analyse, notamment lorsqu'il s'agit de comparer deux arbres d'analyse, l'algorithme existant utilise la représentation CF de l'arbre. Or cette représentation n'existe plus en tant que telle dans le lexique CFG-DOP.
Il faut donc créer un algorithme qui, à partir du code d'un arbre dans le lexique CFG-DOP, reconstruit la suite de règles CF équivalente à ce dernier. C'est justement l'objet de la fonction DOP2CF_modifie.

8.1.1 Fonction DOP2CF_modifie

Soit (n, k_1, m, k_2) le code de l'arbre dans le lexique CFG-DOP et code_CF le vecteur dynamique qui contiendra finalement la représentation CF de l'arbre.

Arguments : le lexique CFG-DOP et le nombre de règles CF de la grammaire (nb_regles_CF)

On remplit code_CF avec k_2 arbres vides, m, k_1 arbres vides et n
Tant que le dernier élément de code_CF est supérieur à nb_regles_CF (c'est-à-dire tant qu'il n'est pas une règle CF)

On récupère la ligne ($\widetilde{n}$, $\widetilde{k_1}$, $\widetilde{m}$, $\widetilde{k_2}$) de n dans le lexique CFG-DOP
On supprime n dans code_CF
On y insère $\widetilde{m}$, $\widetilde{k_1}$ positions avant la fin de code_CF
On ajoute $\widetilde{n}$ à la fin de code_CF

8.2 Extraction d'arbres de profondeur limitée

Dans le cas où l'on restreint la profondeur des arbres à extraire, un petit problème se pose lors de l'algorithme *obtient_sous_arbres_racine_fixe* (6.2.1). En effet, lorsque la fonction *descend_et_empile* est appelée, cette dernière ferme tous les noeuds qui sont à la profondeur maximale autorisée. Si bien que ces noeuds sont ensuite réouverts par l'algorithme et participent à l'incrémentation de *nb_faux*, ce qui fausse tout.
Pour résoudre ce problème, la structure *noeud* a été augmentée de l'attribut *max_profond*. Ce dernier est un booléen qui indique si le noeud en question se trouve à la profondeur maximale autorisée dans l'arbre considéré. Si tel est le cas, *nb_faux* n'est tout simplement pas incrémenté.
Ensuite, lors du passage dans la fonction *ouvre_noeuds_internes*, il suffit de repositionner l'attribut *max_profond* de tous les noeuds à FAUX.

9 Comparaison d'arbres d'analyse

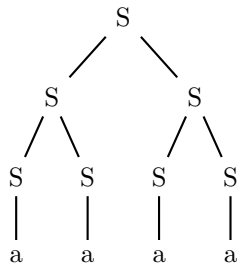
9.1 Principe

Soit T un arbre d'analyse et D_T l'ensemble des dérivations de T.
On peut munir D_T d'une relation d'ordre, notée $\leq$, que l'on définit comme suit :
$\forall\ d_1, d_2\ \in\ D_T\ :\ d_1 \leq d_2\ \Longleftrightarrow$ tout arbre élémentaire apparaissant dans d_1 est sous-arbre d'un arbre élémentaire apparaissant dans d_2 (la notion de sous-arbre comprend l'arbre lui-même).

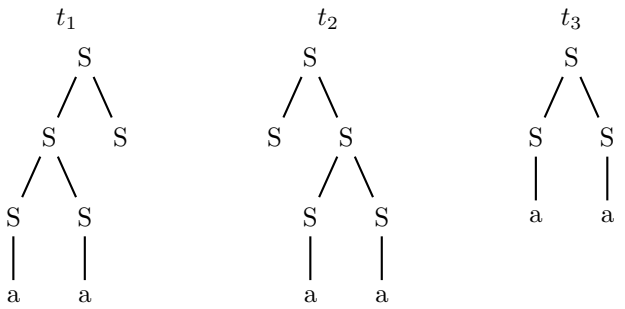
On dit alors que deux dérivations d_1 et d_2 sont comparables si :

(i) $T(d_1) = T(d_2)$
(ii) $d_1 \leq d_2$ ou $d_2 \leq d_1$

La relation d'ordre $\leq$ n'étant pas totale, (ii) n'est pas toujours vérifiée.
Par exemple, si T est l'arbre

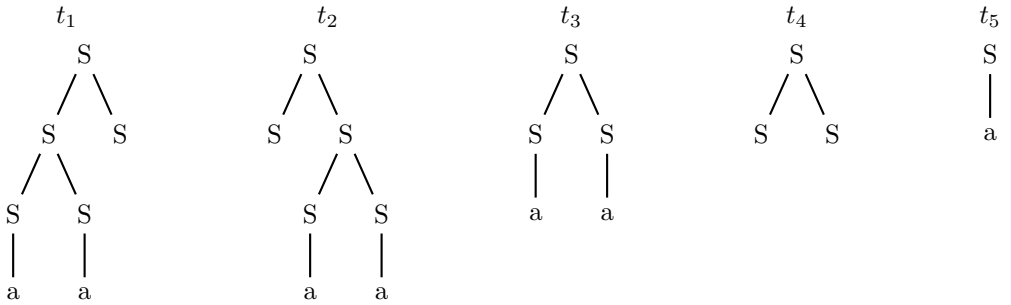


et si $d_1 = t_1 \circ t_3$, $d_2 = t_2 \circ t_3$, où :



On a bien $T(d_1) = T = T(d_2)$, mais par contre $d_1 \not\leq d_2$ (car t_1 n'est pas sous-arbre de t_2 ou t_3) et $d_2 \not\leq d_1$ (car t_2 n'est pas sous-arbre de t_1 ou t_3).
d_1 et d_2 ne sont dans ce cas pas comparables.

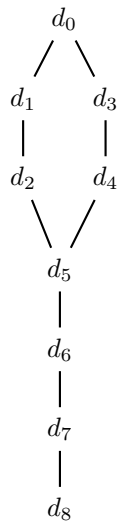
En fait, on peut représenter toutes les dérivations d'un arbre T par un treillis, où chaque noeud correspond à une dérivation et chaque arc à une relation du type $\leq$.
Par exemple, si T est le même arbre que précédemment et que les arbres suivants sont présents dans la grammaire



les dérivations possibles de T sont :

$d_0 = T$
$d_1 = t_1 \circ t_3$
$d_2 = t_1 \circ t_4 \circ t_5 \circ t_5$
$d_3 = t_2 \circ t_3$
$d_4 = t_2 \circ t_4 \circ t_5 \circ t_5$
$d_5 = t_4 \circ t_3 \circ t_3$
$d_6 = t_4 \circ t_3 \circ t_4 \circ t_5 \circ t_5$
$d_7 = t_4 \circ t_4 \circ t_5 \circ t_5 \circ t_3$
$d_8 = t_4 \circ t_4 \circ t_5 \circ t_5 \circ t_4 \circ t_5 \circ t_5$

On a alors le treillis suivant :



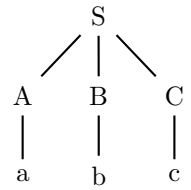
Si l'on considère maintenant les codes de ces dérivations, on constate qu'on a par exemple :

$c_1(d_0) \in \{c_0(d_1), c_0(d_3)\}$
$c_2(d_0) \in \{c_0(d_2), c_0(d_4)\}$
$c_3(d_0) = c_0(d_5)$
...
$c_1(d_1) = c_0(d_2) \neq c_0(d_4) = c_1(d_3)$
etc ...

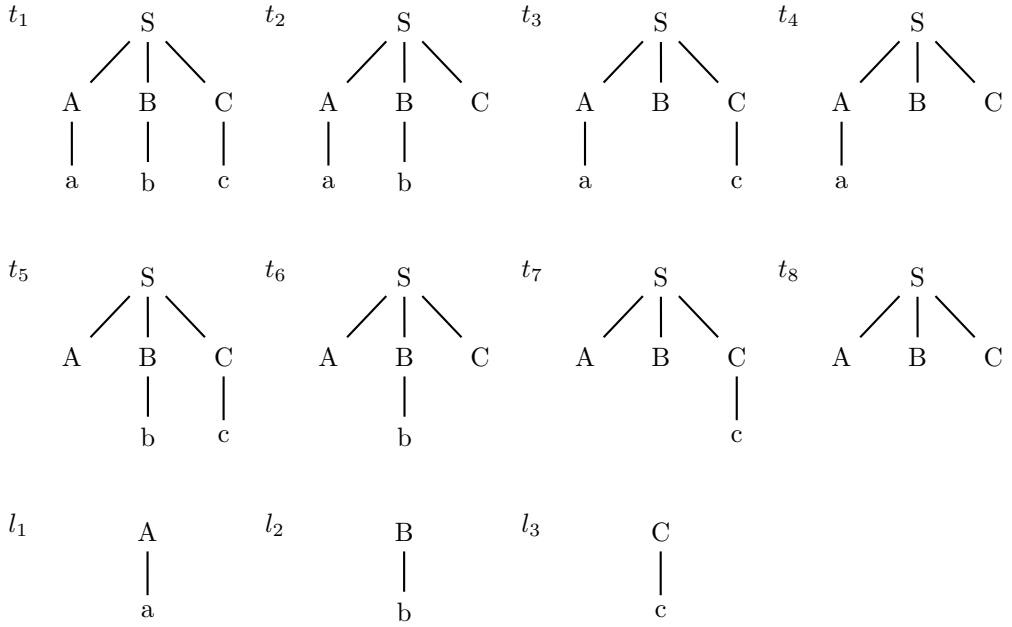
9.2 Cas d'une grammaire DOP

Dans le cas où tous les sous-arbres de T sont présents dans la grammaire, alors l'ensemble D_T de toutes les dérivations de l'arbre T est un treillis très régulier. Voyons cela dans le petit exemple suivant :

Soit T l'arbre



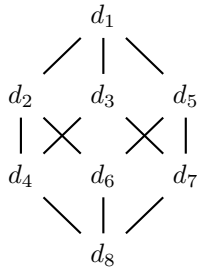
Les sous-arbres de la grammaire DOP sont :



Voici toutes les dérivations possibles pour T :

$d_1 = t_1$
$d_2 = t_2 \circ l_3$
$d_3 = t_3 \circ l_2$
$d_4 = t_4 \circ l_2 \circ l_3$
$d_5 = t_5 \circ l_1$
$d_6 = t_6 \circ l_1 \circ l_3$
$d_7 = t_7 \circ l_1 \circ l_2$
$d_8 = t_8 \circ l_1 \circ l_2 \circ l_3$

Ce qui nous donne le treillis suivant :



Dans le cas général, si l'on considère un arbre contenant n noeuds internes, on peut remarquer les propriétés suivantes :

1) Le treillis possède $N = 2^n$ noeuds.
2) Chaque niveau i du treillis (pour i de allant de 0 à n), est composé de $C_n^i = \frac{n!}{i!(n-i)!}$ noeuds.
3) Pour chaque noeud situé au niveau i, il y a i arcs qui partent contre le haut et n-i contre le bas.

9.3 Noeud optimal de comparaison

En fait, on a une équivalence code-dérivation du fait que tout code d'une dérivation d est une autre dérivation d' telle que $d' \leq d$.
Ainsi, pour vérifier que $T(d_1) = T(d_2)$, il suffit en fait de "descendre" dans le treillis depuis chacune des deux dérivations jusqu'à tomber sur un même noeud. Si l'on ne trouve pas un tel noeud, alors on peut conclure que $T(d_1) \neq T(d_2)$.

Il faut maintenant trouver un algorithme cherchant le noeud qui est le premier à satisfaire $c_k(d_1) = c_l(d_2)$, pour des k et l particuliers.

Soient d_1 et d_2 deux dérivations d'un même arbre T. Définissons l'ensemble

$$Min(d_1, d_2) = \{d \mid d \leq d_1 \text{ et } d \leq d_2\}$$

Cet ensemble est non vide car en particulier, la dérivation CF de T s'y trouve.
Le noeud optimal de comparaison se trouve alors dans cet ensemble, le but étant de le choisir sur le plus haut niveau du treillis possible.

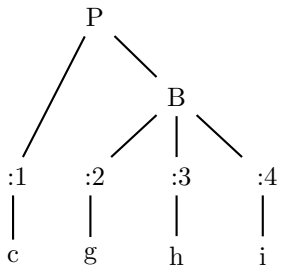
10 Tests

10.1 Fichiers de sortie

Lançons l'extraction de tous les sous-arbres DOP par la commande ./extractdopgram, avec L et G comme lexique et grammaire. On peut alors récupérer les fichiers suivants :

- le lexique CFG-DOP (listlexique -ul G-lexCFG_DOP.slplex)
- la grammaire DOP (listlexique -ul G-lexDOP.slplex)
- le lexique de conversion des non-terminaux (listlexique G-gramCFG-conv.slplex)
- le lexique de conversion des terminaux (listlexique L.slplex)

Voyons l'allure de ces fichiers pour la treebank SSbank9 composée du seul arbre suivant :



Le lexique CFG-DOP :

2)	1	1001	2	1003	1
3)	2	1003	4294967291	1000	1
4)	2	1002	4294967292	1001	1
5)	4	1002	4294967291	1000	1
6)	2	1001	4294967293	1002	1
7)	6	1002	4294967291	1000	1
8)	6	1001	4294967292	1001	1
9)	8	1001	4294967291	1000	1
11)	10	1002	4294967291	1000	1
12)	10	1001	4294967292	1001	1
14)	10	1000	4294967293	1002	1
13)	12	1001	4294967291	1000	1
15)	14	1001	4294967291	1000	1
16)	14	1000	4294967292	1001	1
17)	16	1000	4294967291	1000	1
18)	1	1000	4294967294	1001	1
19)	18	1000	2	1003	1
20)	19	1002	4294967291	1000	1
21)	19	1001	4294967292	1001	1
22)	21	1001	4294967291	1000	1
23)	19	1000	4294967293	1002	1
24)	23	1001	4294967291	1000	1
25)	23	1000	4294967292	1001	1
26)	25	1000	4294967291	1000	1

La première colonne correspond à l'indice du prédécesseur dans la grammaire DOP. Les deuxièmes et quatrièmes colonnes sont les valeurs de k_1 et k_2, qui ont été augmentée de 1'000 pour qu'elles soient toujours non nulles (voir la remarque à la fin de l'exemple 6.3). Quant à la troisième colonne, elle représente la règle CF qu'il faut ajouter au prédécesseur pour obtenir l'arbre que l'on considère. L'entier situé après le "|" est le nombre d'occurrences de la représentation.

Rappel pour la troisième colonne :

...4	=	-2	=	:1	$\longrightarrow$	c		
...3	=	-3	=	:2	$\longrightarrow$	g		
...2	=	-4	=	:3	$\longrightarrow$	h		
...1	=	-5	=	:4	$\longrightarrow$	i		
2			=	B	$\longrightarrow$	:2	:3	:4

La grammaire DOP :

1)	1	513			512	0.0555556
2)	1	2	3	4	512	0.0555556
3)	1	2	3	4294967292	512	0.0555556
4)	1	2	4294967293	4	512	0.0555556
5)	1	2	4294967293	4294967292	512	0.0555556
6)	1	4294967294	3	4	512	0.0555556
7)	1	4294967294	3	4294967292	512	0.0555556
8)	1	4294967294	4294967293	4	512	0.0555556
9)	1	4294967294	4294967293	4294967292	512	0.0555556
10)	2	3	4		513	0.125
11)	2	3	4294967292		513	0.125
12)	2	4294967293	4		513	0.125
13)	2	4294967293	4294967292		513	0.125
14)	4294967294	3	4		513	0.125
15)	4294967294	3	4294967292		513	0.125
16)	4294967294	4294967293	4		513	0.125
17)	4294967294	4294967293	4294967292		513	0.125
18)	4294967295	513			512	0.0555556
19)	4294967295	2	3	4	512	0.0555556
20)	4294967295	2	3	4294967292	512	0.0555556
21)	4294967295	2	4294967293	4	512	0.0555556
22)	4294967295	2	4294967293	4294967292	512	0.0555556
23)	4294967295	4294967294	3	4	512	0.0555556
24)	4294967295	4294967294	3	4294967292	512	0.0555556
25)	4294967295	4294967294	4294967293	4	512	0.0555556
26)	4294967295	4294967294	4294967293	4294967292	512	0.0555556

Les feuilles sont listées en premier, suivies de la partie gauche de la règle et de sa fréquence d'apparition.

Rappel :

...2	=	-4	=	i
...3	=	-3	=	h
...4	=	-2	=	g
...5	=	-1	=	c
1			=	:1
2			=	:2
3			=	:3
4			=	:4
512			=	P
513			=	B

Les lexiques de conversion :

P	512	
B	513	

c	1	1
g	2	1
h	3	1
i	4	1

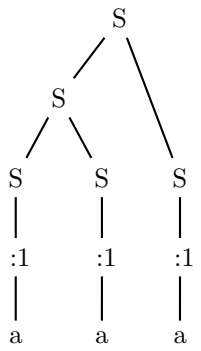
10.2 Comparaison avec l'ancien algorithme d'extraction

Comparons maintenant l'ancien algorithme d'extraction des sous-arbres DOP avec le nouveau.

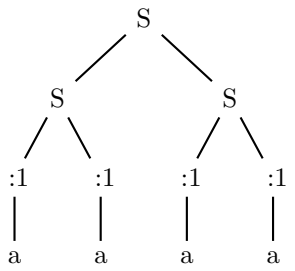
10.2.1 Petits corpus

Considérons dans un premier temps quelques corpus de taille restreinte :

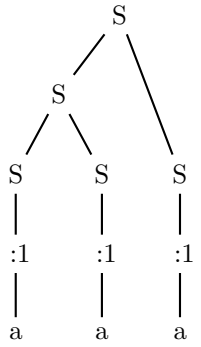
SSbank1

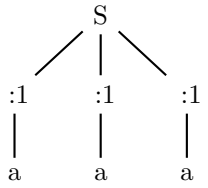


SSbank3

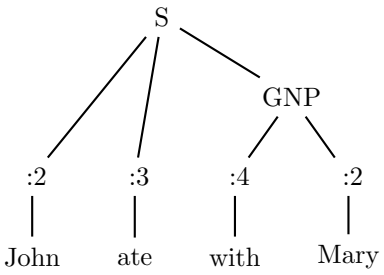


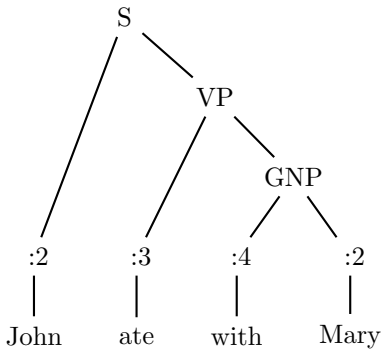
SSbank2





testbank





SSbank9 (comme en section 10.1)

Afin de comparer les algorithmes, prenons comme critère la taille des sous-arbres extraits, définie comme le nombres de sous-arbres apparaissant dans le lexique CFG-DOP. En fait, avec le nouvel algorithme, cette taille est égale à 5 fois le nombre d'arbres non CF extraits du corpus (car le lexique CFG-DOP contient les quatre entiers (n, k_1, m, k_2) et le symbole de fin de chaîne).

			taille du lexique CFG-DOP	
treebank	n° d'arbre	nbre de ss-arbres	ancien algo	nouvel algo
SSbank1	1	45	296	225
SSbank2	1	45	296	225
	2	8	40	40
SSbank3	1	33	212	165
SSbank9	1	26	160	130
testbank	1	24	148	120
	2	36	232	180

On constate que la différence est déjà très significative, même pour des corpus de taille relativement restreinte.

10.2.2 Corpus Atis-Bod

Comparons maintenant les algorithmes sur un corpus de taille plus conséquente, nommé corpus-atis-Bod. Il est constitué de 750 arbres.
Voici les performances de l'ancien algorithme en fonction de la profondeur maximale des arbres que l'on extrait :

- En profondeur inférieure ou égale à 3, l'algorithme extrait tous les sous-arbres
- En profondeur 4, il plante lors de l'extraction du 325^e arbre
- En profondeur 5, il plante au 11^e
- En profondeur infinie, il plante au 8^e arbre

En ce qui concerne le nouvel algorithme, il n'arrive malheureusement pas à terminer l'extraction en profondeur 3 (il stoppe au 187^e arbre). Quant à la profondeur infinie, il arrive uniquement à extraire les sous-arbres des deux premiers arbres du corpus. Ces résultats sont étonnants, surtout si l'on s'intéresse comme avant à la taille des sous-arbres extraits :

			taille du lexique CFG-DOP	
profondeur	nř d'arbre	nbre de ss-arbres	ancien algo	nouvel algo
3	1	153	1'234	765
	2	153	1'234	765
	3	473	5'430	2'365
	4	217	1'834	1'085
	5	250	2'544	1'250
	6	148	1'196	740
	7	1'077	14'626	5'385
	8	1'395	21'046	6'975
	9	221	1'856	1'105
	10	287	2'670	1'435
	...	...	...	...
	186	41	248	205
	187	894	12'258	"plante"
4	1	509	5'502	2'545
	2	509	5'502	2'545
	3	3'169	48'168	15'845
	4	1'067	13'340	5'335
	5	1'326	17'660	6'630
	6	412	4'252	2'060
	7	11'357	204'386	56'785
	8	38'247	791'804	"plante"
5	1	1'589	21'886	7'945
	2	1'589	21'886	7'945
	3	15'633	283'960	78'165
	4	3'455	51'528	"plante"
∞	1	10'373	188'926	51'865
	2	10'373	188'926	51'865
	3	83'121	1'784'024	"plante"

De manière similaire à ce qui a été observé sur de petits corpus, le nouvel algorithme diminue grandement la taille du lexique CFG-DOP. Toutefois, le nouvel algorithme s'arrête avant l'ancien.

11 Perspectives

Il est évident qu'il subsiste plusieurs améliorations à apporter au programme modifié dans ce projet.

En premier lieu, le nouvel algorithme d'extraction des sous-arbres DOP fonctionnant moins bien que l'original, il nécessite d'être amélioré.

Ensuite, les objectifs 5.3 et 5.4 n'ont pas été atteints.
En ce qui concerne la comparaison d'arbres d'analyse, il existe certainement un moyen de tirer profit de la nouvelle représentation interne des arbres.

Enfin, l'algorithme actuel crée dans un premier temps le lexique CFG-DOP dans son intégralité, avant de le reprendre par la suite afin de construire la grammaire DOP. Toutefois, il serait moins coûteux de la générer simultanément à l'extraction des arbres.

12 Conclusion

Le but principal de ce projet était de diminuer la taille des représentations des sous-arbres DOP. Le nouvel algorithme créé constitue de ce point de vue-là une réussite, puisque dorénavant chaque

arbre est représenté à l'aide de quatre entiers uniquement.
Toutefois, les nombreuses opérations d'empilement et de dépilement d'arbres dans la fonction *obtient_sous_arbre_racine_fixe* (6.2.1) pourraient bien pénaliser lourdement l'extraction. En effet, il s'avère que l'ancien algorithme d'extraction fonctionne en pratique mieux que le nouveau.

Références

[1] Vinot, Romain [1999] : *Validation de techniques d'échantillonnage dans le cadre d'une analyse syntaxique de type DOP.*

[2] Chappelier, Jean-Cédric ; Rajman, Martin ; Rozenknop, Antoine [2002] : *Proceedings of Formal Grammar 2002*, chapitre 1.

[3] Chappelier, Jean-Cédric ; Rajman, Martin ; Pallotta, Vincenzo : *Cours de Traitement Informatique des Données Textuelles*, cours 14 et 16, limites des SCFG.

[4] Chappelier, Jean-Cédric : *http ://liawww.epfl.ch/∼chaps/SlpTk/IntroSlpToolkit.html*

[5] Rens Bod ; Remko Scha ; Khalil Sima'an [2003] : *Data Oriented Parsing.*