Algorithme de Cocke-Kasami-Younger

Guillaume Wisniewski
guillaume.wisniewski@linguist.univ-paris-diderot.fr

décembre 2019

Ces notes décrivent l'algorithme Cocke-Kasami-Younger qui permet de vérifier si un mot $\mathbf{w}$ appartient au langage engendré par une grammaire G mise sous forme normale de Chomsky. Cet algorithme montre l'intérêt de la forme normale de Chomsky : en imposant certaines contraintes aux productions d'une grammaire, on arrive à une implémentation nettement plus simple que les autres algorithmes d'analyse. C'est également un bon exemple de programmation dynamique.

1 Grammaire hors-contexte sous forme normale de Chomsky

Notations Nous considérons, dans la suite de ce document, une grammaire sous forme normale de Chomsky $\mathcal{G} = \langle N, \Sigma, P, S \rangle$ [1]. Les productions peuvent donc avoir deux formes :

— $A \rightarrow a$ avec $A \in N$ et $a \in \Sigma$;

— $A \rightarrow B\ C$ avec $A, B, C \in N^3$.

Nous noterons $\mathbf{w} = w_0 w_1 ... w_{|\mathbf{w}|-1}$ les mots composés de $|\mathbf{w}|$ symboles de l'alphabet Σ et$\Rightarrow$ la relation de dérivation : $\alpha \Rightarrow \beta$ si et seulement si α peut être ré-écrit en β en appliquant les productions de la grammaire.

Représentation de la grammaire Dans un soucis de simplification, nous supposerons que les productions générant les terminaux et les productions ré-écrivant les non-terminaux sont stockées séparément. Une grammaire peut alors être représentée comme un triplet, contenant :

— l'axiome de la grammaire ;

— les productions générant les terminaux ;

— les productions ré-écrivant les non-terminaux.

1. Pour mémoire : Σ est l'ensemble des terminaux, N des non-terminaux, P l'ensemble des productions et $S \in N$ est l'axiome de la grammaire.

Les productions générant les terminaux, de la forme $A \to a$ où $A \in N$ et $a \in \Sigma$, peuvent être représentées par un tuple `("A", "a")`, les productions ré-écrivant les non-terminaux, de la forme $A \to B\ C$ où $A, B, C \in N^3$, par un triplet `("A", "B", "C")`.

Ainsi la grammaire de la figure 1 sera représentée par la structure définie au listing 1.

$$\begin{array}{ll} S \to X\ Y & Z \to T\ Z \\ T \to Z\ T & T \to a \\ X \to T\ Y & Y \to b \\ Y \to Y\ T & Z \to b \end{array}$$

FIGURE 1 – Exemple d'une grammaire simple

```
1  grammar = (
2      "S",
3      [("T", "a"), ("Y", "b"), ("Z", "b")],
4      [("S", "X", "Y"), ("T", "Z", "T"), ("X", "T", "Y"),
5       ("Y", "Y", "T"), ("Z", "T", "Z")])
```

Listing 1: Le code python représentant la grammaire de la figure 1.

Cette représentation rend la définition et l'utilisation de la grammaire extrêmement simple : comme seuls des types standards de python sont utilisés, il n'y a aucune « syntaxe » (au sens API) à découvrir. Toutefois son manque *robustesse* restreint son utilisation à un cadre purement pédagogique. En particulier, la distinction entre terminaux et non-terminaux ne repose que sur des conventions implicites (position des symboles dans les tuples et utilisation de la casse) et la distinction entre partie gauche et partie droite des règles uniquement sur la position des symboles dans le tuple. Le risque de faire une erreur en définissant ou en utilisant une grammaire est élevé et la détection des erreurs particulièrement difficile. Il serait préférable d'introduire des types permettant de distinguer ces deux catégories de symboles, par exemple en définissant des classes, mais cela complexifierait immédiatement le code.

Génération des mots engendrés par une grammaire Pour illustrer la manière dont une grammaire représentée selon les principes décrits dans le paragraphe précédent peut être utilisée, nous nous intéressons au problème de l'échantillonnage [2] d'un mot du langage engendré par une grammaire.

La fonction dont le code est repris dans le listing 2 permet de générer aléatoirement un mot du langage engendré par la grammaire passée en argument. L'implémentation

2. Formellement, le problème revient un choisir un mot du langage $\mathcal{L}(\mathcal{G})$, tous les mots ayant la même probabilité d'être choisi.

choisie repose sur deux étapes : une première étape (lignes 8–11) consiste à construire un dictionnaire associant chaque non terminal aux différentes manières de ré-écrire celui-ci (soit par deux non-terminaux, soit par un non terminal) ; puis, dans une seconde étape (lignes 13–17), le proto-mot courant est parcouru de gauche à droite, chaque symbole non terminal rencontré étant ré-écrit en choisissant une des valeurs associées à celui-ci dans le dictionnaire.

```
1  from operator import itemgetter
2  from itertools import groupby, chain
3  from random import choice
4
5
6  def generate(grammar):
7
8      rewriting_rules = groupby(sorted(grammar[1] + grammar[2]), #
9                                key=itemgetter(0))
10     rewriting_rules = {rhs: [g[1:] for g in group]
11                        for rhs, group in rewriting_rules}      #
12
13     proto_mot = [grammar[0]] #
14     while any(w.isupper() for w in proto_mot):
15         proto_mot = (choice(rewriting_rules[w]) if w.isupper() else w
16                      for w in proto_mot) #
17         proto_mot = list(chain.from_iterable(proto_mot)) #
18
19     return proto_mot
20
```

Listing 2: Fonction échantillonnant de manière aléatoire les mots du langage engendré par une grammaire.

Quelques remarques et explications sur la syntaxe python employée dans cette fonction :

— Les lignes 8–11 rassemblent toutes les règles dont la partie gauche est identique. Pour cela nous utilisons la fonction `groupby` qui permet de regrouper tous les éléments d'un *itérable* identiques et consécutifs (c'est pourquoi il est nécessaire de commencer par trier celui-ci). Pour déterminer si deux éléments sont identiques, `groupby` utilise la fonction passée dans l'argument `key` ; en fixant la valeur de cet argument à `itemgetter(0)` (la fonction `itemgetter(i)` renvoie un objet pouvant être utilisé comme une fonction qui retourne le i^{e} élément de l'opérande), nous nous assurons que seule la partie gauche de la règle (dont l'index dans les tuples décrivant les règles est toujours 0) sera utilisée pour déterminer si deux éléments

sont identiques.

— la fonction `choice` permet de choisir un élément de manière aléatoire dans une liste. Appliquée aux valeurs associées à un non-terminal elle retourne une des réécritures possibles de ce non-terminal sous forme d'un tuple.

— la ligne 17 permet de ré-écrire le proto-mot courant sous forme d'une liste de terminaux et de non-terminaux ; en utilisant la fonction `chain` il est possible de transformer la liste de tuples décrivant construite à la ligne 16 (chaque tuple correspondant à la partie droite de la règle sélectionnée) en une liste de symboles (c.-à-d. de « coller » les éléments des différents tuples les uns à la suite des autres).

2 Principe de l'algorithme CYK

L'algorithme CYK est algorithme d'analyse ascendante qui analyse un mot en regroupant de proche en proche les terminaux et les non-terminaux selon les productions de la grammaire en espérant retrouver l'axiome. Ce type d'analyse présente la propriété des sous-structures identiques : le fait qu'une partie du mot analysé puisse être engendrée par un non-terminal et être considérée comme un constituant peut être utilisé dans plusieurs dérivations ; il est naturel de chercher à factoriser les calculs pour réduire la complexité de l'analyse en évitant de recalculer en permanence les mêmes constituants. C'est le principe général de la *programmation dynamique*.

L'idée centrale de l'algorithme CYK, comme tous les algorithmes de programmation dynamique, est de définir une table, la table d'analyse [3] (un *chart* en anglais), permettant d'organiser les calculs et de stocker les résultats intermédiaires de manière à ne pas avoir à répéter la création de sous-structures identiques.

Soit T un tableau comportant $|\mathbf{w}|^2$ éléments définis de la manière suivante :

— $T[i..i]$ correspond à l'ensemble des non-terminaux A tel que $A \to w_i$ est une production de $\mathcal{G}$

— $T[i..j]$ avec $i, j \in [\![0, |\mathbf{w}| - 1]\!]^2$ et $i < j$ contient l'ensemble des non-terminaux A tel que $A \Rightarrow^* w_i...w_j$

Avec cette structure de données, il suffit, pour savoir si un mot peut être généré par une grammaire, de regarder si l'axiome de celle-ci est présent dans $T[0..|\mathbf{w}| - 1]$.

Comme la grammaire est sous forme normale de Chomsky, il est possible de remplir le tableau de proche en proche de manière très simple. Les seules règles introduisant des terminaux étant de la forme $A \to a$, il est facile de remplir la diagonale du tableau $T[i..i]$. Toutes les étapes de réductions ultérieures mettront alors en jeu uniquement deux non-terminaux (à nouveau parce que la grammaire est sous forme normale de Chomsky).

Les autres valeurs du tableau peuvent être déterminées en observant que, lorsque l'on cherche l'ensemble des non-terminaux pouvant engendrer le mot $w_i...w_j$ (c.-à-d. à déterminer la valeur de $T[i..j]$), il suffit, étant donné les contraintes sur la forme des productions, de regarder toutes les manières de diviser le mot $w_i...w_j$ en deux parties

3. L'algorithme CYK est généralement catégorisé comme un algorithme d'*analyse tabulaire*, *chart parsing* en anglais.

$w_i...w_k$ et $w_{k+1}...w_j$ et de chercher les productions de la grammaire dont la partie droite contient des non-terminaux B et C tel que $B \Rightarrow w_i...w_k$ et $C \Rightarrow w_{k+1}...w_j$. Trouver B et C peut se faire directement à l'aide de la table d'analyse : par définition de celle-ci, il suffit de regardes les éléments qui sont stockés aux indices $[i..k]$ et $[k+1..j]$. Il faut toutefois s'assurer que les valeurs pour ces deux indices de la table d'analyse ont déjà été calculées, par exemple en remplissant le tableau T avec les mots $w_i...w_j$ de taille croissante.

3 Implémentation de l'algorithme CYK

3.1 Analyse d'un mot

Les intuitions décrites dans le paragraphe précédent peuvent être formalisées dans l'algorithme 1 qui détermine si un mot w donné appartient au langage engendré par la grammaire $\mathcal{G}$.

Algorithme 1 : Algorithme d'analyse CYK pour une grammaire CNF $\mathcal{G} = \langle N, \Sigma, P, S \rangle$

```
   // Le mot à analyser est w = w_0 ... w_{|w|-1}.
   // La table d'analyse T est initialement vide: T[i..j] = ∅.
1  Function CYK is
       // Initialisations.
2      for i := 0 ... |w| − 1 do
3          foreach A → w_i ∈ P do
4              T[i..i] := T[i..i] ∪ {A}
       // Boucle principale: Cherche les constituants de longueur d
          croissante.
5      for d := 2 ... n do
6          for i := 0 ... |w| − d do
               // On cherche tous les non-terminaux pouvant générer T[i..j]
7              j := i + d // Parcourt toutes les divisions de w_i...w_j
8              for k := i + 1 ... j − 1 do
9                  if B ∈ T[i..k] ∧ C ∈ T[k..j] ∧ A → BC ∈ P then
10                     T[i..j] := T[i..j] ∪ {A}
11     if S ∈ T[0..|w| − 1] then return true else return false ;
```

Une implémentation de l'algorithme CYK en python est donnée dans le listing 3. Cette implémentation soulève plusieurs points intéressants :

— on choisit d'indexer la table d'analyse par un tuple d'indices (`chart[i, j]` formellement équivalent à `chart[(i, j)]`) afin d'avoir les notations les plus proches possibles de l'algorithme initial. Il aurait également été possible de définir la table d'analyse comme une liste de listes d'ensembles mais cette structure est plus com-

pliquée à initialiser [4] et la syntaxe pour accéder à un élément plus « lourde » ; son seul intérêt est qu'il est possible d'itérer facilement sur tous les éléments d'une ligne, une opération qui n'est pas utilisée dans l'algorithme.

— l'utilisation d'un `defaultdict` permet de créer la table d'analyse sans avoir à l'initialiser (dans la plupart des autres langages il serait nécessaire de commencer par faire une double boucle pour créer explicitement un ensemble pour chaque couples d'indice possible).

```python
from collections import defaultdict

def recognize(word, grammar):

    axiom, term_prod, non_term_prod = grammar

    chart = defaultdict(set)

    for i, w in enumerate(word):
        chart[i, i].update(nt for nt, t in term_prod if t == w)

    for d in range(1, len(word)):
        for i in range(0, len(word) - d):
            j = i + d
            for k in range(i, j):
                for lhs, rhs1, rhs2 in non_term_prod:
                    if rhs1 in chart[i, k] and rhs2 in chart[k + 1, j]:
                        chart[i, j].add(lhs)

    return axiom in chart[0, len(word) - 1]
```

Listing 3: Fonction python vérifiant si un mot est engendré par une grammaire à l'aide de l'algorithme CYK.

Un exemple de la table d'analyse générée par l'algorithme CYK est donnée à la figure 2.

4. Le code permettant de l'initialiser serait :

```python
chart = []
for i in range(len(w)):
    chart.append([set() for j in range(len(w))])
```

3	$\{Z, X, S\}_{10}$	$\{Z, X\}_{9}$	$\{Z, X\}_{7}$	$\{Z, Y\}_{4}$
2	$\{T, X\}_{8}$	$\{T, Y\}_{6}$	$\{T\}_{3}$	
1	$\{Z, X\}_{5}$	$\{Z, Y\}_{2}$		
0	$\{T\}_{1}$			
	0	1	2	3

FIGURE 2 – Table d'analyse générée lors de la reconnaissance du mot `abab` par la grammaire de la figure 1. Les indices en rouge indique l'ordre dans lequel les cases sont remplies.

3.2 Complexité de l'analyse

La boucle principale de l'algorithme CYK comporte trois boucles `for` imbriquées se répétant au pire n fois (rappelons que n correspond à la longueur du mot). La complexité de l'algorithme sera donc cubique en n. Chaque cellule de la table d'analyse contient au plus $|N|$ symboles. Il y a, par conséquent, dans le pire des cas, $|N|^2$ combinaison de non-terminaux à tester pour chaque cellule de la table T.

La complexité de l'algorithme CYK est donc, au final, en $\mathcal{O}\left(|N|^2 \times n^3\right)$. De manière très naturelle, la complexité dépend de la taille du mot à analyser et de la taille de la grammaire (à travers le nombre de non-terminaux *après* transformation de la grammaire en forme normale de Chomsky).

3.3 Génération de l'arbre d'analyse

L'algorithme CYK peut facilement être généralisé pour construire un arbre d'analyse d'un mot. Il suffit de stocker dans la table d'analyse les informations indiquant comment le terminal inséré à la position $[i..j]$ a été généré (c.-à-d. la production qui été utilisée lors de la création de celui-ci et les non-terminaux qui ont été combinés). Ces informations peuvent être utilisées, une fois le tableau rempli, pour reconstruire la dérivation lors d'une phase dite de « chainage arrière » *(backtracking)*. Cette approche en deux temps est commune à tous les algorithmes de programmation dynamique :

— une première phase permet de remplir la table et de répondre à la question motivant l'algorithme (dans le cas de l'algorithme CYK : est-ce que le mot appartient au langage engendré par la grammaire ? ; dans le cas du calcul de la distance d'édition, quelle est la valeur de la distance d'édition ?) ;

— une seconde phase permet de reconstruire, à partir de la table, la solution « optimale » (l'arbre d'analyse ou la séquence d'éditions).

Le code du listing 4 montre comment la table peut être enrichie pour stocker toutes les informations nécessaires pour reconstruire l'arbre de dérivation. Pour améliorer la lisibilité et la robustesse du code, toutes ces informations sont stockées dans un `namedtuple`, ce qui permet d'accéder aux différents « champs » en utilisant un nom (p. ex. `t.rhs`

plutôt qu'un indice si l'on avait utilisé un simple tuple). Afin de pouvoir accéder facilement (et efficacement) à l'ensemble des non-terminaux d'une cellule, la table d'analyse est désormais représenté par un dictionnaire de dictionnaires : comme dans le listing 3, le premier dictionnaire permet d'associer à une cellule donnée l'ensemble des non-terminaux qu'elle contient ; le second dictionnaire permet d'associer à un non-terminal donné les informations permettant de reconstruire l'arbre d'analyse.

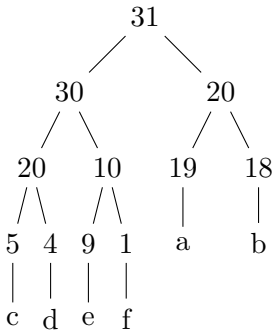
```python
from collections import namedtuple, defaultdict

Item = namedtuple("Item", "rhs lhs k")

def parse(word, grammar):

    axiom, term_prod, non_term_prod = grammar

    chart = defaultdict(dict)

    for i, w in enumerate(word):
        chart[i, i] = {non_term: Item(lhs=non_term, rhs=term, k=0)
                       for non_term, term in term_prod
                       if term == w}

    for d in range(1, len(word)):
        for i in range(0, len(word) - d):
            j = i + d
            for k in range(i, j):
                for lhs, rhs1, rhs2 in non_term_prod:
                    if rhs1 in chart[i, k] and rhs2 in chart[k + 1, j]:
                        chart[i, j][lhs] = Item(lhs=lhs,
                                                rhs=(rhs1, rhs2),
                                                k=k)

    return chart
```

Listing 4: Fonction python implémentant l'algorithme CYK en stockant les informations de chaînage arrière qui permettront de déterminer *un* arbre d'analyse d'un mot.

Dans la version « enrichie » de la table de la figure 2, la cellule $(0, 3)$ contiendra $\{Z : \langle Z, (T, Z), 0\rangle, S : \langle S, (X, Y), 2\rangle, Z : \langle Z, (T, Z), 2\rangle, X : \langle X, (T, Y), 2\rangle\}$ et la cellule $(3, 3)$, $\{Z : \langle Z, b, 0\rangle, Y : \langle Y, b, 0\rangle\}$. Grâce aux informations stockées dans cette table, il

est possible, une fois celle-ci construite, de reconstruire l'arbre d'analyse [5] : il suffit, en commençant par l'entrée décrivant l'axiome de la grammaire dans la cellule $(0, |\mathbf{w}| - 1)$, de retrouver les productions utilisées lors de l'analyse. Ainsi, on voit que la dérivation permettant de générer le mot `abab` commence par ré-écrire l'axiome la grammaire en combinant le non-terminal X de la cellule $(0, 2)$ (le premier non-terminal de la partie droite est stocké dans la cellule (i, k)) et le non-terminal Y de la cellule $(k + 1, j)$ soit $(3, 3)$. En continuant de suivre les liens, il est possible de générer l'arbre d'analyse complet.

La fonction décrite dans le listing 5 met en œuvre ce principe pour renvoyer l'arbre d'analyse d'un mot. Cet arbre est représenté par une structure très simple : chaque nœud interne est décrit par un triplet (un tuple de trois éléments) correspondant (dans cet ordre) à l'étiquette du nœud, le nœud décrivant le fils gauche et celui décrivant le nœud droit (comme nous ne considérons que des grammaires sous forme normale de Chomsky, tous les arbres d'analyse sont des arbres binaires). Les feuilles sont décrites par une paire `(étiquette, contenu)`. La figure 3 donne l'exemple d'un arbre et de sa description en python. Comme pour la représentation de la grammaire (section 1), cette manière de représenter des arbres a pour principal intérêt de n'utiliser que des types standards de python et de ne pas définir de nouvelle classe ou API. L'utilisation d'une bibliothèque permettant de représenter des arbres de manière plus robuste est néanmoins préférable dans un cadre non pédagogique.




```
("31",
  ("30",
     ("20",
        ("5", "c"),
        ("4", "d")),
     ("10",
        ("9", "e"),
        ("1", "f"))),
  ("20",
     ("19", "a"),
     ("18", "b")))
```

FIGURE 3 – Exemple d'un arbre d'analyse correspondant à une grammaire sous forme normale de Chomsky et de sa représentation en python.

4 Lien avec la déduction naturelle

(KÖNIG 1989) montre comment il est possible de formuler les algorithmes d'analyse syntaxique en utilisant le formalisme de la déduction naturelle. Ce lien est approfondi par (SHIEBER, SCHABES et PEREIRA 1995). Cette formulation a un intérêt double.

5. Le cas des grammaires ambiguës, pour lesquelles l'arbre d'analyse n'est pas unique sera traité à la section 5.

```python
def build_tree(chart, axiom, word):

    def build_tree_rec(chart, non_terminal, i, j):
        item = [it for it in chart[i, j].values()
                if it.lhs == non_terminal][0]

        if type(item.rhs) == type(""):
            return (non_terminal, item.rhs)

        return (non_terminal,
                build_tree_rec(chart, item.rhs[0], i, item.k),
                build_tree_rec(chart, item.rhs[1], item.k + 1, j))

    return build_tree_rec(chart, axiom, 0, len(word) - 1)
```

Listing 5: Fonction python permettant de construire un arbre d'analyse à partir du *chart* renvoyé par la fonction `parse`.

Premièrement elle permet de faire le lien entre deux domaines de l'informatique (la preuve automatique et les langages formelles). En plus d'un intérêt « théorique » (il est toujours intéressant de montrer que deux questions qui n'ont, à priori, rien à voir peuvent être résolues par des mécanismes similaires), ce lien permet de décrire dans un formalisme commun tous les algorithmes d'analyse.

Le formalisme de la déduction repose sur la définition de quatre éléments :

— des *items* qui définissent les objets qui seront manipulés ;

— un item particulier, l'*objectif* ;

— une série d'items « donnés » définissant les *axiomes* ;

— les règles d'inférences qui permettent de ré-écrire ces items.

Une preuve par déduction consiste alors trouver une combinaison des règles d'inférences permettant de retrouver l'objectif à partir des axiomes. Ce processus est similaire à celui d'une analyse ascendante dont l'objectif est de retrouver l'axiome de la grammaire en appliquant une suite de ré-écriture.

L'algorithme CYK peut s'exprimer dans le formalisme de la déduction naturelle de la manière suivante :

items $[A, i, j]$

axiomes $[A, i, i]$ pour toute production $A \to w_{i+1}$

objectif $[S, 0, n]$

règle d'inférence pour toute production $A \to B\ C$: $\dfrac{[B, i, j] \quad [C, j+1, k]}{[A, i, k]}$

La figure 4 donne un exemple d'analyse utilisant ce formalisme.

$$\dfrac{\dfrac{[T,0,0]\ \ (a) \qquad \dfrac{[Y,1,1]\ \ (b) \quad [T,2,2]\ \ (a)}{[Y,1,2]}}{[X,0,2]} \qquad [Y,3,3]\ \ (b)}{[S,0,3]}$$

FIGURE 4 – Exemple de l'analyse du mot `abab` dans le formalisme de la déduction naturelle.

5 Cas des grammaires ambiguës

Une grammaire est ambiguë s'il existe plusieurs dérivation possible d'un même mot. Ainsi, il y a deux analyses possibles du mot `1 + 1 + 1` avec la grammaire donnée à la figure 5 : XXX.

$$\begin{aligned} S &\to C\ E \\ E &\to PE|C \\ P &\to + \\ C &\to 1|2|3|4|5|6|7|8|9 \end{aligned}$$

FIGURE 5 – Une grammaire permettant de générer toutes les expressions décrivant un somme d'un chiffre.

L'algorithme CYK, tel qu'implementé dans le listing 3 est incapable d'identifier s'il existe plusieurs manières de dériver un mot avec une grammaire : il ne peut qu'identifier si un mot appartient ou non au langage engendré par une grammaire, quelque soit le nombre de dérivations possibles de celui-ci. En effet, cette implémentation ne stocke dans la cellule (i, j) de la table d'analyse que l'ensemble des non-terminaux qui peuvent générer le sous-mot $w_i..w_j$ et ne conserve aucune information sur les différentes manières dont un terminal peut générer un sous-mot donné. [6]

Il est possible de modifier l'algorithme CYK pour *compter* le nombre de manière dont un mot peut être généré par une grammaire : il suffit d'associer, dans la table d'analyse, chaque non-terminal au nombre de manières dont celui-ci peut-être généré. La valeur de ce compteur peut être déterminée de manière récursive :

— lors de l'initialisation de la table d'analyse à l'aide des productions générant les non-terminaux, le compteur associé au non-terminal est initialisé à 1 ;

6. La situation est similaire pour la méthode décrite dans le listing 4 : une seule manière de générer un non-terminal donné est conservée.

— lors d'une réduction à l'aide de la production $A \to B\ C$, le nombre de manières de générer A sera donné par le produit entre le nombre de manières de générer B et le nombre de manière de générer C. Ces deux valeurs sont directement accessibles dans la table d'analyse si celle-ci est remplie suivant la procédure décrite dans la section 3.

Ces principes sont mis en œuvre dans l'implémentation donnée au listing 6.

```python
def count_derivations(word, grammar):

    axiom, term_prod, non_term_prod = grammar

    chart = defaultdict(lambda: defaultdict(int))

    for i, w in enumerate(word):
        for nt in (nt for nt, t in term_prod if t == w):
            chart[i, i][nt] += 1

    for d in range(1, len(word)):
        for i in range(0, len(word) - d + 1):
            j = i + d
            for k in range(i, j):
                for lhs, rhs1, rhs2 in non_term_prod:
                    if rhs1 in chart[i, k] and rhs2 in chart[k + 1, j]:
                        cnt = chart[i, k][rhs1] * chart[k + 1, j][rhs2]
                        chart[i, j][lhs] += cnt

    return chart[0, len(word) - 1][axiom]
```

Listing 6: Modification de l'algorithme CYK pour compter le nombre de dérivation possible d'un mot donné.

6 CYK pour les WCFG

L'algorithme CYK peut également être utilisé pour déterminer la dérivation de plus grand poids lors de l'analyse d'un mot par une WCFG.

Rappelons que, dans une WCFG, chaque production est associée à un poids et que le poids d'une dérivation (et de l'arbre d'analyse qui lui est associé) est définie comme :

$$\mathbb{W}(\mathbf{d}) = \sum_{r \in \mathbf{d}} w(r)$$

où $\mathbf{d} = (r_i)_{i=1}^{n}$ est une dérivation constituée par l'application de n productions r_i et $w(r_i)$ est la probabilité d'une production donnée [7].

Pour déterminer la dérivation de plus grand poids, il suffit de stocker dans la cellule (i, j) de la table d'analyse le non terminal *et* le plus grand poids permettant de générer le sous-mot $w_i..w_j$ à partir de ce non terminal. Formellement, dans l'algorithme 1, la table d'analyse T doit être modifiée pour pouvoir associer pour chaque couple d'indices i, j un dictionnaire tel que $T[i..j][A]$ sera le plus grand poids permettant de dériver $w_i..w_j$ à partir de A ; la règle d'insertion (lignes 9–10) devient alors :

```
1 if w(A → B C) + T[i..k][B] + T[k..wj] > T[i..j][A] then
2 |   T[i..j][A] = w(A → BC) + T[i..k][B] + T[k..wj]
```

avec les conventions que $w(A \to B\ C) = 0$ si $A \to B\ C$ n'est pas une production de la grammaire et que $T[i..j][A] = -\infty$ si $A \notin T[i..j]$. Cette condition permet d'assurer que seule la dérivation de plus grand poids sera stockée dans la table (principe des sous-structures optimales).

Le listing 7 donne une implémentation de l'algorithme CYK capable de déterminer la dérivation de plus grand poids. Cette implémentation repose sur une généralisation simple de la représentation des CFG introduite à la section 1 dans laquelle un poids est ajouté, en dernière position, au tuple décrivant les productions. Un exemple de représentation de la PCFG donné à la figure 6 (et d'utilisation de celle-ci) est donné au listing 8.

S → NP VP	$[p = 1, 0]$	NP → NP PP	$[p = 0, 4]$
PP → P NP	$[p = 1, 0]$	NP → astronomers	$[p = 0, 1]$
VP → V NP	$[p = 0, 7]$	NP → ears	$[p = 0, 18]$
VP → VP PP	$[p = 0, 3]$	NP → saw	$[p = 0, 04]$
P → with	$[p = 1, 0]$	NP → stars	$[p = 0, 18]$
V → saw	$[p = 1, 0]$	NP → telescopes	$[p = 0, 1]$

FIGURE 6 – Exemple d'une PCFG.

7. Chaque production est souvent associé à la probabilité que celle-ci soit utilisée ; le poids est alors défini comme le logarithme de la probabilité.

Références

[1] Esther KÖNIG. "Parsing As Natural Deduction". In : *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*. ACL '89. Vancouver, British Columbia, Canada : Association for Computational Linguistics, 1989, p. 272–279. DOI : 10.3115/981623.981656. URL : https://doi.org/10.3115/981623.981656.

[2] Stuart M. SHIEBER, Yves SCHABES et Fernando C.N. PEREIRA. "Principles and implementation of deductive parsing". In : *The Journal of Logic Programming* 24.1 (1995). Computational Linguistics and Logic Programming, p. 3–36. ISSN : 0743-1066. DOI : https://doi.org/10.1016/0743-1066(95)00035-I. URL : http://www.sciencedirect.com/science/article/pii/074310669500035I.

On pourra également consulter les pages suivantes :

— Une implémentation avec des exemples d'utilisation (à noter en particulier le code permettant de transformer une grammaire sous forme normale de Chomsky) : https://nbviewer.jupyter.org/github/Naereen/notebooks/blob/master/agreg/Algorithme%20de%20Cocke-Kasami-Younger%20%28python3%29.ipynb

```python
from collections import defaultdict, namedtuple

WItem = namedtuple("WItem", "rhs lhs k weight")

def best_derivation(word, grammar):

    axiom, term_prod, non_term_prod = grammar

    chart = defaultdict(dict)

    for i, w in enumerate(word):
        chart[i, i] = {non_term: WItem(lhs=non_term, rhs=term, k=0, weight=weight)
                       for non_term, term, weight in term_prod
                       if term == w}
    print(chart)
    for d in range(1, len(word)):
        for i in range(0, len(word) - d):
            j = i + d
            for k in range(i, j):
                for lhs, rhs1, rhs2, weight in non_term_prod:
                    w = weight +\
                        chart[i, k][rhs1].weight if rhs1 in chart[i, k]\
                          else float("-infinity") +\
                        chart[k + 1, j][rhs2].weight if rhs2 in chart[k + 1, j]\
                          else float("-infinity")])

                    if lhs not in chart[i, j] or w > chart[i, j][lhs].weight:
                        chart[i, j][lhs] = WItem(lhs=lhs,
                                                 rhs=(rhs1, rhs2),
                                                 k=k,
                                                 weight=w)

    return chart
```

Listing 7: Version « probabilisée » de l'algorithme CYK capable de retrouver la dérivation de plus grand poids.

```python
from math import log10

wcfg = ["S",
        (("P", "with", log10(1)),
         ("V", "saw", log10(1)),
         ("NP", "astronomers", log10(.1)),
         ("NP", "ears", log10(0.18)),
         ("NP", "saw", log10(0.04)),
         ("NP", "stars", log10(0.18)),
         ("NP", "telescopes", log10(0.1))),
        (("S", "NP", "VP", log10(1)),
         ("PP", "P", "NP", log10(1)),
         ("VP", "V", "NP", log10(0.7)),
         ("VP", "VP", "PP", log10(0.3)),
         ("NP", "NP", "PP", log10(0.4)))]


word = "astronomers saw stars with ears".split()
chart = best_derivation(word, wcfg)

print(f"best proba: {10 ** chart[0, len(word) - 1]['S'].weight:.7}")
print(build_tree(chart,
                 "S",
```

Listing 8: Définition d'une WCFG en python et affichage du meilleur arbre d'analyse.