Algorithme de Cocke-Kasami-Younger

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

novembre 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. Nous noterons $\Rightarrow$ 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 $A \in V$ 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 $A, B, C \in V^3$, par un triplet `("A", "B", "C")`.

Ainsi la grammaire de la figure 1 sera représentée par le code de la figure 2.

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

FIGURE 1 – Exemple d'une grammaire simple

```
grammar = (
    "S",
    [("T", "a"), ("Y", "b"), ("Z", "b")],
    [("S", "X", "Y"), ("T", "Z", "T"), ("X", "T", "Y"),
     ("Y", "Y", "T"), ("Z", "T", "Z")])
```

FIGURE 2 – 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 de « 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 complexifierai immédiatement le du 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 à la figure 3 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
```

FIGURE 3 – 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 permettent de regrouper 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 itérables identiques et consécutifs (c'est pourquoi il est nécessaire de commencer par trier l'itérable). Pour déterminer si deux éléments sont identiques, la fonction `groupby` utilisent la fonction passée en dans l'argument `key` ; en fixant la valeur de cet argument à `itemgetter(0)` (la fonction `itemgetter(i)` renvoie un objet appelable qui retourne le i^{e} élément de l'opérande), nous nous assurons que seules 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 seront iden-

tiques.

— la fonction `choice` permet de choisir un élément de manière aléatoire dans une liste. Appliquée aux valeurs associés à une non-terminal elle retourne une des ré-écritures 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 tuples ; en utilisant la fonction `chain` il est possible de transformer cette liste de tuples en liste de chaines de caractères (c.-à-d. de « coller » les valeurs des différents tuples les unes à la suite des autres).

2 Principe de l'algorithme CYK

Les algorithmes d'analyse ascendante 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 : lors de l'analyse d'un mot, XXX

L'idée centrale de l'algorithme CYK, comme tous les algorithmes de programmation dynamique, est de définir un tableau permettant d'organiser les calculs et de stocker les résultats intermédiaires de manière à ne pas avoir à répéter le calcul 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 $i \in [\![0, |\mathbf{w}| - 1]\!]$ et $i < j$ contient l'ensemble des non-terminaux A tel que $A \Rightarrow^* w_i...w_j$

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 $A \prod a$, 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.

Ensuite, les étapes de réduction, dans lesquelles XXX : 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 $w_i...w_k$ et $w_{k+1}...w_j$ et de chercher les productions de la grammaire dont la partie droite contient les non-terminaux B et C tel que $B \Rightarrow w_i...w_k$ et $C \Rightarrow w_{k+1}...w_j$. Déterminer les terminaux B et C peut se faire directement à l'aide de la table T : 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 du tableau ont déjà été calculées, par exemple en considérant les mots $w_i...w_j$ de taille croissante.

3 Implémentation de l'algorithme CYK

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

```
// La phrase à analyser est u = u_1 ... u_n.
// La table d'analyse T est initialement vide: T[i, j] = ∅.
Function CYK is
    // Initialisations.
    for j := 1 ... n do
        foreach A → u_i ∈ P do
            T[i, i + 1] := T[i, i + 1] ∪ {A}
    // Boucle principale: Cherche les constituants de longueur
       croissante.
    for l := 2 ... n do
        for i := 1 ... n − l + 1 do
            for k := i + 1 ... i + l − 1 do
                if B ∈ T[i, k] ∧ C ∈ T[k, i + l] ∧ A → BC ∈ P then
                    T[i, k + l] := T[i, k + l] ∪ {A}
    if S ∈ T[1, n] then return true else return false ;
```

Algorithme 1 : Analyseur CYK pour une grammaire CNF $G = (\Sigma, N, S, P)$

Une implémentation de l'algorithme CYK en python est donnée à la figure 4. Cette implémentation soulève plusieurs points intéressant :

— la fonction `range(i, j)` permet de définir un générateur parcourant les entiers de $[\![i, j[\![$; pour inclure l'entier j, il est nécessaire d'appeler la fonction `range` avec i et $j + 1$ en argument.

— on choisit d'indexer le chart 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 le chart comme une liste de listes d'ensembles mais cette structure est plus compliquée à initialiser [3] 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 le *chart* sans avoir à l'initialiser (dans la plupart des autres langages il serait nécessaire de commencer par faire une

3. Le code permettant de l'initialiser serait :

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

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(non_term 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 + 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]:
                        chart[i, j].add(lhs)

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

FIGURE 4 – Fonction python vérifiant si un mot est engendré par une grammaire à l'aide de l'algorithme CYK.

4 Génération de l'arbre d'analyse

L'algorithme CYK peut facilement être généralisé pour construire un arbre d'analyse d'un mot suivant une grammaire donnée. Il suffit de stocker dans le *chart* 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 réduits).

Le code de la figure 5 montre comment le chart peut être enrichi 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). L'avantage d'utiliser un `namedtuple` plutôt qu'un dictionnaire est double : *i)* il n'est pas possible d'ajouter des dictionnaires

à des ensembles et *ii)* la syntaxe pour accéder à un élément est plus « légère » avec un `namedtuple` (`t.rhs` au lieu de `t["rhs"]`).

```python
from collections import namedtuple
from collections import defaultdict

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

def parse(word, grammar):

    axiom, term_prod, non_term_prod = grammar

    chart = defaultdict(set)

    for i, w in enumerate(word):
        chart[i, i].update(Item(lhs=non_term, rhs=term, k=0) for non_term, term in term

    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 {c.lhs for c in chart[i, k]} and rhs2 in {c.lhs for c in
                        chart[i, j].add(Item(lhs=lhs, rhs=(rhs1, rhs2), k=k))

    print(chart[0, len(word) - 1])
    # backtracking code is missing
```

FIGURE 5 – Fonction python déterminant *un* arbre d'analyse d'un mot à l'aide de l'algorithme CYK.

5 Références

— 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`