Analyse syntaxique par CKY probabiliste

Table des matières

[I. Qu'est-ce que l'algorithme CYK ? 4](#_Toc454309795)

[CYK classique 4](#_Toc454309796)

[CYK probabiliste 5](#_Toc454309797)

[II. Réalisation du projet 5](#_Toc454309798)

[Obtenir une grammaire pour le CYK 6](#_Toc454309799)

[Le corpus SEQUOIA 6](#_Toc454309800)

[Programme d’extraction 7](#_Toc454309801)

[Mettre la grammaire en forme normale de Chomsky 8](#_Toc454309802)

[Notre implémentation du CYK 8](#_Toc454309803)

[Remettre la grammaire en forme n-aire 9](#_Toc454309804)

[III. L'évaluation 9](#_Toc454309805)

[Généralités 9](#_Toc454309806)

[Notre programme d’évaluation 9](#_Toc454309807)

[Les résultats obtenus 10](#_Toc454309808)

[IV. Manuel d'utilisation 10](#_Toc454309809)

[Prérequis 10](#_Toc454309810)

[Les commandes du CYK 10](#_Toc454309811)

[Commandes principales 10](#_Toc454309812)

[Les options 11](#_Toc454309813)

[Le script extrateur.py 11](#_Toc454309814)

[Commandes principales 11](#_Toc454309815)

[Le script evaluation.py 12](#_Toc454309816)

[Commandes principales 12](#_Toc454309817)

[Options 12](#_Toc454309818)

[Quelques exemples de commandes 13](#_Toc454309819)

[V. Annexes 14](#_Toc454309820)

[Listing des algorithmes 14](#_Toc454309821)

[ALGO INIT\_CYK 14](#_Toc454309822)

[ALGO CYK 15](#_Toc454309823)

[Bibliographie 15](#_Toc454309824)

# Qu'est-ce que l'algorithme CYK ?

## CYK classique

Notre projet consiste en l’implémentation de l’algorithme CYK probabilisé.

L’algorithme CYK tire son nom des trois individus qui l’on trouvé, Cocke, Younger et Kasami. Il s’agit d’un algorithme tabulaire d’analyse syntaxique ascendante pour les grammaires non-contextuelles. Celui-ci permet de déterminer si un mot est reconnu ou non par une grammaire, si oui, il fournit également un arbre syntaxique.

Cet algorithme demande à ce que la grammaire soit en forme normale de Chomsky. Le temps de calcul de l’algorithme est proportionnel à la taille de la grammaire et à la taille du mot à analyser.

Voici un exemple. Nous disposons d’une grammaire en forme normale de Chomsky :

S -> SN SV

SN -> D N

SV -> V SN

V -> mange

D -> la | une

N -> fille | pomme

{S, SN, SV, V, N, D} est l’ensemble des non-terminaux et {mange, la, une, fille, pomme} est l’ensemble des terminaux, c’est-à-dire nos lettres.

Nous disposons également d’un mot à parser : « la fille mange une pomme ».

Avec l’algorithme CYK, on obtiendra :

A REMPLIR AVEC LA MEME STRATEGIE QUE L’ALGO DE KORANTIN

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| La | Fille | Mange | Une | Pomme |

## CYK probabiliste

Notre objectif était d’implémenter la version probabiliste de l’algorithme présentée précédemment.

Cette version probabiliste est intéressante car elle permet de choisir le ou les meilleures analyses syntaxique (et arbres syntaxiques). En effet, lorsqu’on remplit la demi-matrice avec une grammaire un peu plus complexe que le petit exemple vu précédemment, on a souvent des cases que l’on peut remplir avec plusieurs règles différentes. Or chaque règles finira par engendrer des analyses et des arbres parfois très différents les uns des autres. Avec la version probabiliste du parseur, on peut choisir de ne garder que la (ou les) analyse les plus probables, en choisissant de remplir les cases de la demi-matrice avec la (ou les) règles qui ont une meilleure probabilité.

=> METTRE ICI un exemple avec une grammaire probabilisée et une demi matrice remplie pour comprendre le fonctionnement de la version probabiliste.

Dans notre implémentation, nous devons choisir si nous gardons seulement une règle par case (la règle avec la plus forte probabilité), ou si nous en gardons plusieurs. En décidant de garder plusieurs règles, on autorise le parseur à produire potentiellement plusieurs analyses possibles pour une même phrase. En envisageant de couvrir plusieurs (voir toutes) les analyses possibles, nous sommes plus efficaces. Mais en contrepartie, la multiplication des règles dans les cases de la demi-matrice augmente de manière exponentielle la complexité de l’algorithme.

Nous verrons dans la suite de ce documents quels choix d’implémentation ont été retenus mais également ceux qui ont été abandonnés et pourquoi.

# Réalisation du projet

Maintenant que nous avons vu ce qu’est l’algorithme CYK et sa version probabiliste, nous allons nous pencher sur notre implémentation de cet algorithme. Comme nous l’avons vu précédemment, CYK nécessite pour son fonctionnement une grammaire en forme quadratique (ou forme normale de Chomsky). Ici, comme nous souhaitons implémenter la version probabiliste, il faut en plus que chaque règle de notre grammaire ait une probabilité, c’est-à-dire qu’il nous faut une PCFG (Probabilistic Context-Free Grammar).

## Obtenir une grammaire pour le CYK

### Le corpus SEQUOIA

Pour obtenir cette grammaire, nous disposons d’un corpus contenant plus de 3 000 phrases en français. Il s’agit du corpus Sequoia composé de diverses sources (Europarl, Le Monde, Wikipédia et EMA), c’est donc un corpus aux thèmes assez variés.

Un corpus à thèmes variés est meilleur qu’un corpus sur un seul thème ou domaine. Par exemple si l’on extrait une grammaire uniquement à partir d’un corpus médical, il y aura beaucoup de productions de la langue française qui nous échapperont totalement. En effet dans ce genre de corpus scientifique, il n’y a pas de dialogues, il n’y a pas ou peu de phrases à la première ou à la seconde personne et le vocabulaire est assez particulier et restreint. Des corpus très varié permettent lors de l’apprentissage de couvrir un maximum de phénomènes de la langue dans toute sa variété.

Le corpus Séquoia nous fournit un grand ensemble de phrases au format MRG. Ce format parenthésé est très pratique pour être parsé.

Voici un exemple de phrase que l’on peut trouver dans le corpus :

((SENT (NP-SUJ (DET La) (NC tâche)) (VN (ADV ne) (V sera)) (ADV pas) (AP-ATS (ADJ aisée)) (PONCT .)))

Pour extraire une grammaire à partir de ce corpus, on doit d’abord définir quels seront nos terminaux et nos non-terminaux. Nous avons décidé que notre programme ne parsera pas des phrases de lexique mais uniquement des suites d’étiquettes syntaxiques.

Donc nos non-terminaux seront SENT, NP, VN, AP, ADV, DET, NC, V, ADJ… C’est-à-dire, toutes les étiquettes en majuscules que l’on trouve derrière une parenthèse.

Nos terminaux seront, non pas les mots de la phrase (en minuscules), mais nous transformerons ce lexique par la dernière étiquette juste avant lui. Par exemple pour « (V sera) » nous n’aurons pas de règle « V -> sera » mais plutôt une règle « V -> v ». Nos terminaux sont donc des catégories morphosyntaxiques notées en minuscules.

Sachant cela, nous pouvons dès à présent ignorer totalement les mots de lexique dans le corpus. Nous avons également décidé d’ignorer les fonctions grammaticales car elles ne sont pas primordiales pour un parseur syntaxique mais surtout parce qu’elles viendraient augmenter la taille de la grammaire et donc augmenter la complexité en temps de l’algorithme (rappelez-vous, nous avons vu dans la présentation du CYK que la complexité en temps de ce dernier est proportionnelle à la taille de sa grammaire).

### Programme d’extraction

Notre programme d’extraction de grammaire prend en entré le fichier MRG.

L’objectif étant d’obtenir en sortie une grammaire hors contexte probabilisé en forme normale de Chomsky.

La première étape du programme d’extraction consiste transformer chaque phrase en tokens et les stocker dans une liste de listes. A partir de cette liste de listes, on peut ensuite récupérer les éléments terminaux, les éléments non-terminaux ainsi que les productions.

### Probabiliser la grammaire

Comment s’y prendre pour probabiliser nos règles de réécriture ? C’est une question sur laquelle nous avons passé du temps. Nous avions deux stratégies et nous avons dû faire un choix. Tout d’abord, nous avions pensé à probabilisé chaque règle au fur et à mesure qu’elles soient extraites. Cela nous semblait plus simple au départ. Notre idée était la suivante, chaque fois qu’on rencontre une règle :

* Si la partie gauche n’a jamais été trouvé avant, la règle vaut 1.
* Si la partie gauche a déjà été trouvé avant mais que la partie droite est nouvelle, on divise 1 par le nombre de règles vu auparavant (+1 car on vient d’en trouver une autre) et on obtient la probabilité de chaque règle partageant cette même partie gauche. Il faut alors mettre à jour toutes ces règles-là.
* Si la partie gauche a déjà été trouvée et que la partie droite aussi (c’est-à-dire qu’on a déjà rencontré cette même règle) alors on divise 1 par le nombre de règles (comme précédemment) sauf qu’ensuite on additionne les probabilités des deux règles identiques et on en supprime une. Ainsi la règle qui a été vu deux fois (ou plus) a une probabilité plus forte que les autres qui n’ont été vues qu’une seule fois.

Finalement, nous avons décidé de probabiliser chaque règles après qu’elles soient toutes extraites. En effet, cela nous évite de faire sans cesse des mises à jour des règles vues auparavant. Donc la méthode est la suivante. Pendant l’extraction des règles, on dispose de compteurs. Certains nous permettent de compter +1 chaque fois qu’on rencontre une règle parfaitement identique à une ou plusieurs règles déjà extraites. D’autres nous permettent de compter lorsque nous rencontrons une partie gauche de règle qui a déjà été extraite. Après l’extraction, nous reprenons nos compteurs afin d’attribuer les probabilités à nos règles de productions. Pour mieux comprendre ce processus, voici un exemple avec une petite grammaire :

VP -> v

VP -> v NP

VP -> v ADJ

VP -> v

On dispose de 4 règles qui ont la même partie gauche « VP » donc notre compteur de VP est à 4. Nous disposons également de 2 règles VP identiques, donc notre compteur de règles « VP -> v » est à 2.

Nous attribuons une probabilité à chacune des règles en divisant 1 par le nombre total de règles, nous obtenons donc une probabilité de 1/4 pour chaque règle.

VP -> v = 1/4

VP -> v NP = 1/4

VP -> v ADJ = 1/4

VP -> v = 1/4

Puis on additionne les probabilités de toutes les règles en double et supprime l’une des deux règles identiques pour ne plus avoir de doublons :

VP -> v = 1/4 + 1/4 = 1/2

VP -> v NP = 1/4

VP -> v ADJ = 1/4

Dans un souci de précision, nous avons choisi d’utiliser le module « fractions » pour Python pour les probabilités, et non pas des float.

### Mettre la grammaire en forme normale de Chomsky

Pour la mise en forme quadratique, nous nous sommes inspirés du papier de Leiss et Lange (voir Annexes).

Nous nous assurons d’abord qu’il n’y a aucune production epsilon (ou production vide) dans la grammaire précédemment extraite et probabilisée.

## Notre implémentation du CYK

c) On a enfin une PCFG, on peut faire fonctionner l'algorithme CYK probabiliste. Expliquer son fonctionnement. Expliquer nos choix d'implémentation (par exemple la forme de notre demi matrice, combien de règles par cases et pourquoi, qu’obtient-on en sortie ?, si on a essayé plusieurs versions il faut toutes les mettre et expliquer pourquoi on les a finalement abandonnées.

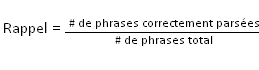
## Remettre la grammaire en forme n-aire

Pour comparer nos arbres de sortie et les arbres gold, il faut remettre nos arbres en n-aire. Car les arbres gold ne sont pas binaires.

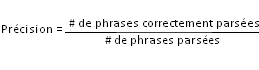
# L'évaluation

## Généralités

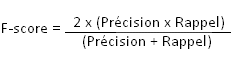
* **Rappel**: Le rappel définie le nombre d’éléments corrects parmi le nombre d’éléments total. Ce calcul permet de mesurer à quel point le programme est capable de donner toutes les solutions pertinentes.



* **Précision :** La précision définie le nombre d’éléments corrects parmi le nombre d’éléments prédits. Ce calcul permet de mesurer à quel point le programme est capable d’éclipser toutes les solutions non-pertinentes.



* **F-score :** Le f-score, aussi appelé f-mesure, est la moyenne harmonique de la précision et du rappel.



## Notre programme d’évaluation

Le script prend en input des arbres prédits par le CYK et des arbres gold.

La première étape consiste à la suppression des feuilles des deux ensembles d’arbres. On obtient ainsi des nouveaux arbres, nous conservons ensuite les feuilles de ces nouveaux arbres dans des listes et nous comparons les listes de feuilles des arbres gold et des arbres prédits afin de s’assurer que nous sommes bien en train de tester deux arbres de la même phrase.

On comptabilise le nombre d’éléments communs aux deux phrases et les éléments spécifiques à la gold et à la prédiction dans des variables utilisées ensuite pour calculer la précision, le rappel et le f-score de chaque phrase (appelé précision, rappel et f-score moyen).

Pour finir, la précision, rappel et f-score global est calculé, il s’agit là de la comparaison des éléments de l’ensemble des phrases.

Tous les résultats sont affichés dans le terminal.

## Les résultats obtenus

- Nos résultats commentés

- une partie aussi sur le temps d’exécution

# IV. Manuel d'utilisation

## Prérequis

Tous les scripts ont été écrits en Python 3, de ce fait, nous ne garantissons pas qu’ils fonctionnent en Python 2.

## Les commandes du CYK

### Commandes principales

Le script ckys.py se lance avec deux arguments.

Le premier est un pickle contenant une PCFG sous forme normale de Chomsky produite par le script extracteur.py.

Le deuxième argument est un corpus, un fichier mrg contenant les phrases que le script devra analyser.

ckys.py grammaire\_train.pickle test.mrg

Par défaut, le script est lancé de façon non-interactive : il traite toutes les phrases du corpus en commençant par la première et imprime l'analyse sur stdin et des messages d'informations ou d'erreur sur stderr.

### Les options

* L'option –i

L'option -i permet de lancer le script en mode interactif. Le mode interactif affiche chaque phrase une par une avec leur longueur et leur numéro, at attend des commandes de l'utilisateur. Les commandes acceptées par le script sont les suivantes:

* + - exit
    - quit
    - Ctrl+D : quitte le script.
    - y : lance l'analyse de la phrase courante et l’affiche.
    - goto NOMBRE : va à la phrase numéro NOMBRE si NOMBRE est plus grand que le numéro de la phrase actuelle.

* L'option -p

L'option -p donne la position de départ dans le corpus du script. Par défaut le script commence au début du fichier. Spécifier un numéro avec l'option -p permet de le faire commencer à la phrase correspondant à ce numéro.

## Le script extrateur.py

### Commandes principales

Extracteur.py est un script qui permet d’extraire une grammaire probabilisée sous forme normale de Chomsky, à partir d’un fichier au format mrg (mrg strict ou id mrg).

Le script extracteur.py se lance avec deux arguments :

* Le nom du fichier mrg à partir duquel construire la grammaire.
* Le nom du fichier où sauvegarder cette grammaire.

La grammaire ainsi crée est un fichier pickle (objet python sérialisé) contenant un tuple de trois éléments :

* L’ensemble (set) des terminaux de la grammaire.
* L’ensemble des non-terminaux
* Un dictionnaire contenant les règles de productions et leur probabilité.

## Le script evaluation.py

### Commandes principales

Evaluation.py est le script qui calcule la précision, le rappel et la f-mesure étiquetés ou non pour les constituants d’un ensemble d’analyses lues depuis un fichier, depuis un stdin, ce qui permet de l’utiliser avec un pipe unix.

Les analyses gold doivent être obligatoirement lue depuis un fichier spécifié avec l’option –g ou –gold.

Pour l’utiliser avec un fichier, il faut le lui passer en argument :

Evaluation.py –g fichierGold.mrg candidats.mrg

Ou

Evaluation.py candidats.mrg –g fichierGold.mrg

Ou

Cat candidats.mrg | evaluation.py –g fichierGold.mrg

Ou même

Ckys.py grammaire.pickle fichierGold.mrg | evaluation.py –g fichierGold.mrg

### Options

* L’option –l

Si l’option –l ou –labeled est activée, les constituants seront considérés comme correct uniquement s’ils ont e même span ET la même étiquette. Sinon, seul le span sera considéré.

* L’option –v

L’option –v ou –verbose affiche la précision, rappel et f-mesure pour chaque analyse, ainsi que les span manquant du gold ou les span suplémentaire.

|  |
| --- |
|  |

## Quelques exemples de commandes

Cette section présente de façon rapide l'ensemble des scripts du projet CYK probabiliste.

En lançant les commandes présentées dans l'ordre, on obtient successivement un corpus d'entraînement et de test, une grammaire tirée du corpus d'entraînement un ensemble de parse candidats et enfin leur évaluation vis-à-vis du corpus de test.

Étant donné un fichier mrg fi.mrg :

La commande :

dispatch.py fi.mrg train.mrg test.mrg

crée deux fichiers,

* train.mrg, qui doit servir de corpus d'entraînement (90 % de fi.mrg)
* test.mrg qui doit servir de corpus de test (10% de fi.mrg )

(La répartition entre les deux fichiers est faite au hasard).

La commande

extracteur.py fi.mrg grammaire.pickle

extrait une PCFG (sous forme normale de Chomsky) à partir de fi.mrg et l'enregistre dans le fichier grammaire.pickle.

extracteur.py train.mrg grammaire\_train.pickle

extrait une PCFG à partir de train.mrg et l'enregistre dans le fichier grammaire\_train.pickle.

updategrammar.py grammaire.pickle grammaire\_train.pickle

modifie le fichier grammaire\_train.pickle : elle sert à transférer dans la grammaire d'entraînement l'ensemble des règles lexicales du corpus afin d'éviter d'avoir des erreurs dues à des mots inconnus.

La commande

ckys.py grammaire\_train.pickle test.mrg > sortie\_test.mrg

lance l'analyse des phrases de test.mrg par CYK probabiliste et les rassemble dans sortie\_test.mrg.

ATTENTION : cette étape peut durer extrêmement longtemps (plusieurs heures). Pour voir rapidement le fonctionnement de cky, utiliser le "mode interactif" (cf. ci-dessous).

La commande

evaluation.py --gold test.mrg sortie\_test.mrg

calcule précision, rappel et f-mesure non-étiquetés sur sortie\_test.mrg en prenant pour référence les parses originaux dans test.mrg.

# V. Annexes

- Listing des algorithmes des programmes en pseudo code (dans leur totalité) pour l'extraction, pour le CYK et pour l’évaluation.

## Listing des algorithmes

### ALGO INIT\_CYK

| span = 1

| for (I, letter) in enumerate (len(mot)) :

| | for nt in nonTerminals :

| | | for lexical in productions(nt) :

| | | | chart[spam][i][nt] = (lexical.proba, (span, i))

| | [ Fin for

| | Fin for

| Fin for

Fin ALGO

### ALGO CYK

| Chart = defaultdict(lambda : defaultdict(int))

| for max in range (2, len(mot) +1) :

| | for min in range (max – 2, 0, -1) :

| | | for nt in nonTerminals :

| | | | best = 0

| | | | for binaire in prod-binaire-nonTerminals :

| | | | | for mid in range (min + 1, max -1) :

| | | | | | t1 = chart[min][mid][nt]

| | | | | | t2 = chart[min][max][nt]

| | | | | | candidat = t1 \* t2 \* binaire.proba

| | | | | | if candidat > best :

| | | | | | | best = (candidat, (t1.ind,t2.ind))

| | | | | | Fin if

| | | | | Fin for

| | | | | Chart[min][max][nt] = best

| | | | Fin for

| | | Fin for

| | Fin for

| Fin for

Fin ALGO

## Bibliographie

Martin LANGE & Hans LEIß, 2009, <https://www.informaticadidactica.de/index.php?page=LangeLeiss2009>

Brian E. ROARK & Richard SPROAT, *Computational Approaches to Morphology and Syntax*, Oxford University Press, 2007.

Mariana ROMANYSHYN & Vsevolod DYOMKIN, *The Dirty Little Secret of Constituency Parser Evaluation*, 2014, <http://tech.grammarly.com/blog/posts/The-Dirty-Little-Secret-of-Constituency-Parser-Evaluation.html>.

Ivan TITOV, 2000, <http://ivan-titov.org/teaching/nlmi/>

Page Wikipédia, *Précision et rappel*, <https://fr.wikipedia.org/wiki/Pr%C3%A9cision_et_rappel>

Alain BACCINI, Sébastien DEJEAN, Désiré KOMPAORE, Josiane MOTHE, *Analyse des critères d’évaluation des systèmes de recherche d’information*, <https://www.irit.fr/publis/SIG/2010_ISI_BADSKNDMJ.pdf>