Intro : présentation de l'intitulé et de la composition du groupe

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

I. Qu'est ce que l'algorithme CYK ?

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 |

b) Présentation de la version probabiliste du CYK

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

- Quels intérêts ?

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.

* Quels problemes éventuels avec cette version de CYK ?

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.

II. Notre implémentation

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).

a) Extraction de la grammaire en probabilisant chaque règle

- Présentation du corpus Sequoia fourni

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.

* Pourquoi un copus avec des thèmes varié est mieux ?
* Exemple de ce qu’on trouve dans le corpus afin de comprendre le format

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

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

* Préparation des données avant extraction (ce qu’on ignore, ce qu’on garde et pourquoi)

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 terminaux seront DET, NC, ADV, V, ADJ, PONCT, … Et nos non-terminaux seront SENT, NP, VN, AP, … Sachant cela, nous pouvons dès à présent ignorer totalement les mots de lexique dans le corpus. Nous avons également ignoré 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 la grammaire).

- Expliquer comment fonctionne notre programme d’extraction

- la partie qui extrait des structures « propres » en parcourant le copus

Nous avons écrit un programme en Python qui parcourt le corpus à l’aide du parseur PLY.

EXPLIQUER LES ETAPES D’EXTRACTION EN DETAIL

Illustrer avec des exemples si possible

- Présentation des fonctions qui vont ensuite parcourir cette structure « propre » pour en extraite à chaque fois les règles et attribuer une probas

Après l’obtention de ces structures, nous les parcourons afin d’extraire des règles de réécriture.

## COMMENT ON S’Y PREND POUR EXTRAIRE LES REGLES ? DANS QUELLE GENRE DE STRUCTURE DE DONNEES LES STOCKE T-ON ? ##

PARTIE A REVOIR ENTIEREMENT CAR CHANGEMENT DE STRATEGIE

##################################################################################

Probabiliser :

Probabiliser la grammaire pendant l’extraction ou probabiliser la grammaire après ?

Probabiliser la grammaire après extraction :

On parcours la grammaire extraite on compte le nombre de règle (par exemple le nombre de regle VP => ) on divise 1 par le nombre de règle ce qui donne la probabilité de chaque règle. Puis on cherche les règle identique et on les supprime en ajoutant leur probabilité entre-elles.

Par exemple :

Quatre règles VP donc 1/4 = 0.25

VP => v = 0.25

VP => v NP = 0.25

VP => v P = 0.25

VP => v = 0.25

Puis il y a deux règles identiques donc on en supprime une en ajoutant sa proba à l’autre :

VP => v = 0.50

VP => v NP = 0.25

VP => v P = 0.25

Cette méthode nous semblait plus rapide car on ne modifie qu’une seule fois les probas. Mais en réalité elle nous oblige à stocker toutes les règles qui ont la même partie gauche dans des structure (par exemple une liste avec toutes les règles VP, une autre avec toutes les règles NP, …)

Donc, on a décidé de gérer les probabilités au même temps que l’extraction. 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.
* 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 probas des deux règles identique et on en supprime une.

##################################################################################

b) Mettre la grammaire obtenue en forme normale de Chomsky

- Expliquer l'algorithme de mise en forme quadratique

- Expliquer comment nous avons implémenté ça en prenant en compte le recalcule des probas

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.

III. L'évaluation du programme

- Généralités sur l’évaluation d’un parseur, formules (précision, rappel, f-score)

- Notre évaluation (les entrées, les sorties, que fait l’algo en détail)

- Nos résultats commentés

IV. Manuel d'utilisation

- Comment lancer l'extraction de la grammaire et comment parser une phrase

- listing des commandes dispo (un « help » ?)

=> Ne pas oublier de bien illustrer avec des screenshoot genre tuto.

V. Annexes

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