Projet – Transformation de grammaires

CHERQI Hajar

19 Janvier 2025

Table des matières

1 Introduction

Ce projet a été réalisé en binome.
Dans ce projet, nous avons programmé différents algorithmes permettant de transformer une grammaire algébrique donnée dans les deux formes normales : *forme normale de Chomsky* et *forme normale de Greibach*, tout en respectant les propriétés de chaque transformation. Une fois les transformations effectuées, les grammaires obtenues sont utilisées pour générer des mots de leurs langages respectifs.

2 Structure de données

Pour représenter notre grammaire, nous avons choisi de créer une classe `Grammaire` qui comporte un constructeur initialisant notre grammaire ainsi que les fonctions nécessaires à nos transformations.

2.1 Constructeur

La fonction représentant le constructeur est la suivante :
Notre grammaire comporte un axiome, un alphabet de non-terminaux, un alphabet de terminaux et nos règles.

- L'axiome est un non-terminal, il correspond au membre gauche de la première règle du fichier.
- L'alphabet des non-terminaux contient 25 lettres majuscules (toutes sauf E), suivies d'un des 10 chiffres. Il y a donc 250 non-terminaux possibles.
- L'ensemble des terminaux est l'ensemble des 26 lettres minuscules.
- La lettre E majuscule représente epsilon.
- Voici le constructeur de notre classe :

```
def __init__(self,axiome : str,regles : dict):
    self.non_terminaux_generaux = [lettre+str(chiffre) for chiffre in [0,1,2,3,4,5,6,7,8,9] for lettre in string.ascii_uppercase if lettre != "E"]
    self.terminaux_generaux = list(string.ascii_lowercase)
    self.nos_terminaux = set([lettre for values in regles.values() for regle in values for lettre in regle if lettre in self.terminaux_generaux])
    self.nos_non_terminaux = (regles.keys())
    self.axiome = axiome
    self.regles = regles
```

2.2 Méthodes

Nous avons créé des fonctions nécessaires à la transformation en formes normales de Greibach et de Chomsky. D'autres fonctions servent d'outils supplémentaires pour faciliter ces transitions.

2.2.1 Fonctions pour Greibach et Chomsky

- *supression_axiome* : Prend en entrée le nouvel axiome. Elle remplace l'axiome précédent par le nouveau et crée une règle de la forme : `nouvel_axiome` → `ancien_axiome`.

```
def suppression_axiome(self,nv_axiome : str):
    est_a_droite = False
    for regle in self.liste_regle():
        if self.axiome in regle[1]:
            est_a_droite = True
    if est_a_droite:
        prc_nt = self.prochain_non_term_dispo()
        self.regles[prc_nt] = [self.axiome]
        self.axiome = prc_nt
    self.regles = verif_doublon(self.regles)
```

— *supression_terminale* : Supprime les lettres terminales (voir section 4.2).

```
def suppression_terminales(self):
    modification={}
    nouvels_regles={}
    for elem in self.nos_terminaux:
        for non_ter in self.non_terminaux_generaux :
            if non_ter not in self.nos_non_terminaux and non_ter not in list(modification.values()):
                val=non_ter
                break
        modification[elem]=val
    for key,values in self.regles.items():
        for i,val in enumerate(values):
            valeur=self.decomposer_regle(val)
            temp=[]
            for elem in valeur:
                for lettre in elem:
                    if lettre in self.nos_terminaux:
                        temp.append(lettre)
            if len(valeur)>1:
                for l in set(temp):
                    values[i]=values[i].replace(l,modification[l])
                    nouvels_regles[modification[l]]=l
    for key,value in nouvels_regles.items():
        self.regles[key]=value
```

— *suppression_membres_droits* : Supprime les membres droits contenant plus de deux symboles (voir section 4.3).

```
def suppression_membres_droits(self):
    modification=[]
    non_term_selectionne=[]
    for key,values in self.regles.items():
        for i,val in enumerate(values):
            decomp_val=self.decomposer_regle(val)
            if len(decomp_val)>2 :
                for non_ter in self.non_terminaux_generaux:
                    if non_ter not in self.nos_non_terminaux and non_ter not in non_term_selectionne:
                        new_key=non_ter
                        non_term_selectionne.append(non_ter)
                        break
                new_val="".join(val[2::])
                modification.append((key, i, new_key, new_val))
    for key, i, new_key, new_val in modification:
        self.regles[new_key] = [new_val]
        self.regles[key][i] = self.regles[key][i][0:2] + new_key
```

— *suppression_epsilon* : Supprime les règles contenant des epsilon (ε) (voir section 4.4).

```
def suppression_epsilon(self):
    def donne(regle, element : str):
        nb_elem = regle.replace(element, "E").count("E")
        liste = []
        i = 0
        while i < len(regle):
            if regle[i:i+len(element)] == element:
                liste.append(element)
                i += len(element)
            else:
                liste.append(regle[i])
                i += 1
        combinaisons = [['' if (i >> j) & 1 else element for j
                         in range(nb_elem)] for i in range(2**nb_elem)]
        liste_final = [''.join(temp if temp != element
                               else combinaison.pop(0) for temp in liste)
                               for combinaison in combinaisons]
        for i in range(len(liste_final)):
            if liste_final[i] == '':
                liste_final[i] = 'E'
        return liste_final
```

```
liste_epsilon = [1]
while len(liste_epsilon) != 0:
    liste_epsilon = []
    for regle in self.liste_regle():
        if regle[1] == "E":
            if regle[0] != self.axiome:
                liste_epsilon.append(regle[0])
    for elem in liste_epsilon:
        self.regles[elem].remove("E")
    for regle in self.liste_regle():
        for elem in liste_epsilon:
            if elem in regle[1]:
                self.regles[regle[0]] += (donne(regle[1],elem))
self.regles = verif_doublon(self.regles)
```

— *suppression_unitaire* : Supprime les règles unitaires (voir section 4.5).

```python
def suppression_unitaire(self):
    for key,values in self.regles.items():
        for val in values:
            if len(val)==2 and val in self.nos_non_terminaux:
                values.remove(val)
                for elem in self.regles[val]:
                    if elem not in self.regles[key]:
                        self.regles[key].append(elem)
```

— *suppr_regle_1_1* : Supprime les règles tel que $X \to Y$.

```python
def supr_regle_1_1(self):
    for i in range(2):
        dico_des_modif = {}
        for NT, production in self.regles.items():
            for i in range(len(production)) :
                if len(production[i]) == 2 and production[i] in self.nos_non_terminaux:
                    dico_des_modif[NT] = self.regles[NT][:i] + self.regles[production[i]] + self.regles[NT][i+1:]
        for elem in dico_des_modif.keys():
            self.regles[elem] = dico_des_modif[elem]
        self.regles = verif_doublon(self.regles)
```

— *supr_non_terminaux_tete* : Élimine la récursivité gauche immédiate (voir section 3.3).

```python
def supr_non_terminaux_tete(self):
    liste = []
    for NT, prod in self.regles.items():
        for i in range(len(prod)):
            if prod[i][0] not in self.terminaux_generaux and prod[i][0] != 'E':
                if prod[i][:2] == NT:
                    liste.append((NT,prod[i][2:],prod[:i]+prod[i+1:]))
    for rec in liste:
        prc_nt = self.prochain_non_term_dispo()
        self.regles[rec[0]] = [regle + prc_nt for regle in rec[2]]
        self.regles[prc_nt] = [rec[1]+ rec[0],'E']
    self.supr_epsilon()
    for NT, prod in self.regles.items():
        for i in range(len(prod)):
            if prod[i][:2] in self.non_terminaux_generaux:
                self.regles[NT] = prod[:i] + [regle + prod[i][2:] for regle in self.regles[prod[i][:2]]]+ prod[i+1:]
```

— *supr_terminaux_pas_tête* : Élimine les terminaux qui ne sont pas en position initiale.

```python
def suppression_terminales(self):
    modification={}
    nouvels_regles={}
    for elem in self.nos_terminaux:
        for non_ter in self.non_terminaux_generaux :
            if non_ter not in self.nos_non_terminaux and non_ter not in list(modification.values()):
                val=non_ter
                break
        modification[elem]=val
    for key,values in self.regles.items():
        for i,val in enumerate(values):
            valeur=self.decomposer_regle(val)
            temp=[]
            for elem in valeur:
                for lettre in elem:
                    if lettre in self.nos_terminaux:
                        temp.append(lettre)
            if len(valeur)>1:
                for l in set(temp):
                    values[i]=values[i].replace(l,modification[l])
                    nouvels_regles[modification[l]]=l
    for key,value in nouvels_regles.items():
        self.regles[key]=value
```

2.2.2 Fonctions intermédiaires

— *decomposer_regle* : Décompose les éléments d'une règle en sous-parties.

```python
def decomposer_regle(self,regle : dict):
    pattern = r'[a-z]|[A-Z][0-9]'
    return re.findall(pattern, regle)
```

— *prochain_non_term_dispo* : Génère un nouveau non-terminal disponible pour l'utiliser dans les transformations.

```python
def prochain_non_term_dispo(self):
    nb_nt = len(self.nos_non_terminaux)
    prc_nt = "S0"
    for _ in range(nb_nt):
        if prc_nt in self.nos_non_terminaux :
            if "9" in prc_nt :
                asc = ord(prc_nt[0])+1
                if asc > 90:
                    asc = 65
                elif asc == 69:
                    asc = 70
                prc_nt = chr(asc) + "0"
            else :
                prc_nt = prc_nt[0] + str(int(prc_nt[1])+1)
    return prc_nt
```

3 Lecture du fichier

La fonction `lire` permet de lire un fichier contenant une grammaire et de la stocker dans la structure interne. La fonction est la suivante :

```python
def lire(file):
    dico={}
    with open(file,'r') as f:
        lignes=f.readlines()
        for ligne in lignes:
            if ":" in ligne:
                ligne = ligne.replace(" ", "").strip().split(":")
                dico[ligne[0]]=[]
                regles=ligne[1].split('|')
                for elem in regles:
                    dico[ligne[0]].append(elem)
    return dico
```

FIGURE 1 – Fonction lire

La fonction prend en paramètre un fichier de type `.general` et renvoie un dictionnaire sous la forme {membre gauche=[membre droit]}. Elle lit chaque ligne présente dans le fichier tout en ignorant les espaces et considère ce qui est avant " :" comme le membre gauche et le reste comme le membre droit.

4 Forme normale de Greibach

Une grammaire algébrique est en forme normale de Greibach si toutes ses règles respectent l'une des formes suivantes :

— $A \to aA_1A_2 \cdots A_n$ avec a un terminal et $n \geq 0$

— $S \to \epsilon$ (avec ϵ correspondant à epsilon, et S l'axiome)

Pour transformer une grammaire algébrique en forme normale de Greibach, il faut appliquer cinq algorithmes différents, soit l'implémentation de cinq fonctions dans notre programme.

4.1 Suppression des règles epsilon

Une production $A \to \epsilon$ est une règle de la forme $A \to \epsilon$ où ϵ représente le mot vide. Il faut identifier tous les non-terminaux qui produisent ϵ, remplacer les autres productions par toutes les combinaisons possibles (avec ou sans le non-terminal), puis supprimer les règles $A \to \epsilon$ une fois les substitutions effectuées (sauf pour l'axiome si la règle $S \to \epsilon$ existe).

4.2 Suppression des règles unitaires

Les règles unitaires sont celles où un non-terminal A produit directement un autre non-terminal B. Il faut alors supprimer ces règles et remplacer B par toutes les règles où le membre gauche est B.

4.3 Suppression de la récursivité gauche immédiate

Pour une règle $A \to A\alpha \mid \beta$, où α et β sont des chaînes de symboles, on la transforme en $A \to \beta A'$ et on ajoute la règle $A' \to \alpha A' \mid \epsilon$.

4.4 Suppression de la récursivité gauche

La récursivité gauche normale se produit lorsqu'un non-terminal apparaît dans le côté droit de sa propre production, mais qu'il est précédé d'un ou plusieurs symboles terminaux avant d'apparaître. Exemple : $A \to aA\alpha$.

4.5 Fonction Greibach

On crée une fonction `greibach` qui fait appel à toutes les fonctions citées précédemment. La fonction est la suivante :

```
def greibach():
    donnees=lire('Test.general')
    axiome = list(donnees.keys())[0]
    gram=Grammaire(axiome,donnees)
    gram.retirer_axiome_de_droite()
    gram.suppression_epsilon()
    gram.supr_regle_1_1()
    gram.supr_non_terminaux_tete()
    gram.supr_terminaux_pas_tete()
    return gram
```

FIGURE 2 – Fonction Greibach

5 Forme normale de Chomsky

Une grammaire algébrique est en forme normale de Chomsky si toutes les règles de production sont sous l'une des formes suivantes :

— $A \to BC$ avec A, B, C des non-terminaux

— $A \to a$ où a est un terminal et A un non-terminal

Pour transformer une grammaire algébrique en forme normale de Chomsky, il faut appliquer cinq algorithmes différents, soit l'implémentation de cinq fonctions dans notre programme.

5.1 Suppression de l'axiome

On ajoute une nouvelle règle $S_0 \to S$, avec S_0 le nouvel axiome et S l'axiome de départ.

5.2 Suppression des lettres terminales

On doit supprimer les lettres terminales dans les membres droits de règles de longueur au moins 2. Si une lettre terminale figure dans un membre droit de règle de longueur au moins 2, on la remplace par une nouvelle règle $Y \to a$ et on remplace a par Y dans les règles où a apparaît.

5.3 Suppression des membres droits

On supprime les membres droits avec plus de deux symboles. On crée une nouvelle règle avec les non-terminaux en trop dans la règle de départ, et on remplace dans la règle de départ ces éléments par le membre gauche de la nouvelle règle.

5.4 Suppression des règles epsilon

Cette fonction est similaire à celle présentée à l'étape 4.1.

5.5 Suppression des règles unitaires

Cette fonction est similaire à celle présentée à l'étape 4.2.

5.6 Fonction de Chomsky

Lors de la fonction, on fait appel aux fonctions citées précédemment. La fonction est la suivante :

```
def chomsky():
    donnees=lire('Test.general')
    axiome = list(donnees.keys())[0]
    gram=Grammaire(axiome,donnees)
    gram.suppression_axiome('S0')
    gram.suppression_terminales()
    gram.suppression_membres_droits()
    gram.suppression_epsilon()
    gram.suppression_unitaire()
    return gram
```

FIGURE 3 – Fonction de Chomsky

6 Écriture de la grammaire dans un fichier

La fonction `ecrire` prend en entrée un dictionnaire et écrit dans un fichier. On crée un fichier qui va contenir nos règles sous la forme $A \to \alpha$. La fonction parcourt le dictionnaire, chaque clé étant le membre gauche. On écrit ensuite une flèche, puis on écrit les éléments correspondant à la clé, séparés par un symbole |.

```python
def ecrire(gram : Grammaire,forme : str):
    with open(str('Grammaire.'+forme),'w') as file:
        liste_deja_parcouru = []
        liste_NT = [gram.axiome]
        while liste_NT != []:
            file.write(liste_NT[0] + ' -> ')
            i = 0
            for val in gram.regles[liste_NT[0]]:
                if i< len(gram.regles[liste_NT[0]])-1:
                    file.write(val + ' | ')
                else:
                    file.write(val+'\n')
                for elem in gram.nos_non_terminaux:
                    if elem in val and elem not in liste_deja_parcouru and elem not in liste_NT:
                        liste_NT.append(elem)
                i += 1
            liste_deja_parcouru.append(liste_NT.pop(0))
```

FIGURE 4 – Fonction d'écriture

7 Génération de mots de longueur n

Nous devons créer une fonction qui prend en paramètre un entier n et qui va retourner un ensemble de mots de longueur inférieure ou égale à n. On débute par l'axiome et on parcourt chaque règle donnée par l'axiome ainsi que celle donnée par celle-ci, et ainsi de suite (en récursivité), en ne gardant que les mots de la longueur souhaitée.

```python
def generation_mots(grammar, start_symbol, max_length):
    def parcours(sequence):
        if len(sequence) > max_length:
            return set()
        if all(symbol.islower() or symbol == "E" for symbol in sequence):
            return {"".join(symbol for symbol in sequence if symbol != "E")}

        words = set()
        for i, symbol in enumerate(sequence):
            if symbol.isupper() and symbol[-1].isdigit() and symbol in grammar:
                for production in grammar[symbol]:
                    new_sequence = sequence[:i] + production + sequence[i+1:]
                    words.update(parcours(new_sequence))
                break
        return list(words)

    return parcours([start_symbol])
```

FIGURE 5 – Fonction de génération de mots