Projet machine a inventer des mots
LaTeX

William Kaczmarek Théo Julien Violette Mugnier

20 janvier 2019

Table des matières

I Introduction

Pour ce premier projet d'algorithmique, nous devions créer un algorithme pouvant générer des mots. Tout comme le fait David, dans sa vidéo extraite de son blog ScienceEtonnante. Nous avons cherché à créer un programme qui serait capable de créer des mots inexistants, mais tout de même français. En effet, nous voulions qu'à la lecture des mots générés un Français puisse douter de l'existence de ceux-ci. Nous avons donc à travers plusieurs algorithmes tenté de générer des mots les plus réalistes possible, les plus Français possible, respectant certaines règles de grammaire typiquement françaises. Dans ce rapport, vous trouverez notre raisonnement, et les différents algorithmes composant notre programme : *Aléatoire, Enchaînement de digrammes, Enchaînement de trigrammes* et enfin notre dernier algorithme permettant de *générer des simples phrases* en Français évidemment.

II Développement

II.1 Première génération, Aléatoire

Notre premier algorithme est le moins concluant, en effet, il génère bien des mots inventés qui n'existent pas. Mais ils n'ont aucune logique et sont souvent imprononçables comme par exemple : "flsfiq". En effet, il repose sur la fonction "randomize". Nous avons une constante *"alphabet"* et nous allons aléatoirement sortir une lettre de celle-ci afin de générer notre mot de la taille souhaitée. À la vue des mots générés n'importe qui serait capable d'affirmer que ces mots n'existent pas. Nous avons donc cherché un moyen de donner à notre algorithme les spécificités de la langue française pour qu'il puisse inventer des mots plus similaires à ceux qui existent. Nous n'avons rencontré aucune difficulté pour cette première partie, mais le résultat est loin d'être convaincant.

II.2 Deuxième génération, Enchaînement de digrammes

Continuons, pour rendre notre algorithme bien plus performant, nous devions lui fournir une base de données dont il devait "s'inspirer" pour générer des mots "plus français" (qui respectent les règles usuelles de formation de mots). Nous sommes donc allés chercher sur internet un fichier texte comportant une vaste liste de mots en français pour pouvoir l'analyser. Cette liste à été réalisée par Christophe Pallier à partir du dictionnaire Français de Gutenberg, elle comporte 336 531 mots, ce qui va nous permettre de faire des probabilités sur la formation des mots. Nous avons enlevé de la liste tout les caractères qui ne sont pas dans notre constante *alphabet*.

Construction de la table de Fréquence

Nous savions ce qu'il nous restait à faire, mais une question se posait : "Comment analyser ce dictionnaire de plus de 330 000 mots ???" Afin de résoudre ce mystère, nous avons essayé de visualiser le problème, nous avons donc pris un feutre Veleda et commencé à poser nos idées et simuler différents algorithmes, pour trouver une méthode viable. Nous devions extraire les probabilités qu'une lettre de l'alphabet soit suivie d'une autre dans notre dictionnaire, sachant que notre alphabet comprend tous les caractères, c'est-à-dire :

```
CONST   alphabet : WideString  = 'abcdefghijklmnopqrstuvwxyzàâéèêëîïôùûüÿæœç-';
```

Figure 1 – Image de notre constante alphabet

Voici notre raisonnement/méthode, pour un dictionnaire composé d'un seul mot : "chien"

-	c	h	i	e	n
c	x	y	0	0	0
h	x	0	y	0	0
i	x	0	0	y	0
e	x	0	0	0	y
n	x	0	0	0	0

→

Variables	Type	Explications
x	-FIN	Proba que cette lettre soit à la fin
	-DEBUT	Proba que cette lettre soit au début
	-Compt1	Nombre de fois que la lettre apparait
	-Compt2	Nombre de fois qu'elle soit suivie d'une autre lettre
	-Proba	Compt1 diviser par Compt2
y	-FIN	Proba 0
	-DEBUT	Proba 0
	-Compt1	Nombre de fois que la lettre apparaît
	-Compt2	Nombre de fois qu'elle soit suivie d'une autre lettre
	-Proba	Compt1 divisé par Compt2 (0 si compt2=0)
Quand la variable est égale à 0 cela signifie que tous les types sont égaux à 0. La lecture du tableau est : la probabilité que la lettre de gauche soit suivie de la lettre en haut.		

C'est ce raisonnement que nous avons traduit en pascal et appliqué au dictionnaire français de Gutenberg. Nous avons donc un énorme tableau de probabilité pour toutes les lettres de l'alphabet.
Ci-dessous une représentation graphique de ce tableau, plus la couleur est claire plus la probabilité que la lettre soit suivie de celle-ci est forte. On peut voir que dans la langue française la lettre *"Q"* est très souvent suivie de la lettre *"U"*. Par contre dans cette représentation graphique les probabilités de début et de fin n'apparaissent pas.

Génération des mots

Enfin, nous avons notre table de fréquence, c'est bien beau, mais notre algorithme ne nous donne toujours aucun mot. Pour réussir à utiliser notre table de fréquence, il nous faut plusieurs fonctions chargées de choisir les lettres en fonction de la précédente. Pour cela, nous commençons par choisir 2 premières lettres pouvant s'enchaîner correctement, puis on récupère la ligne de probabilité du tableau de la dernière lettre. On les additionne en enlevant les lettres de probabilité égale à 0, puis on choisit un nombre aléatoire entre $]0\,;\,1]$. Et on choisit la lettre qui correspond au chiffre : Voici un exemple si notre dictionnaire ne contient qu'un seul mot : *"génération"*, on génère les 2 premières lettres *"gé"* et on cherche ensuite les lettres suivant le *"é"*. on somme

g	é	n	é	r	a	t	i	o	n
0	0	0.5	0	0.5	0	0	0	0	0

les probabilités et on retire les inutiles : Puis on choisit un nombre aléatoire entre 0 et 1 et on prend la lettre

n	n	r
0	0.5	1

inférieure ou égale a la probabilité dans le tableau ci-dessus. Si notre nombre est 0.7, ce sera inférieur à "r" mais supérieur à "n", donc on prend la lettre "r". Et on continue ainsi de suite. Nous allons faire la même chose pour la lettre après "r" qui va dans ce cas être "a" vu que notre dictionnaire est composé d'un seul mot.

Le résultat est déjà impressionnant, mais nous pouvons aller encore plus loin en étendant cette méthode à une lettre de plus, nous avons donc commencé la plus grosse partie de ce projet, la méthode *trigramme*.

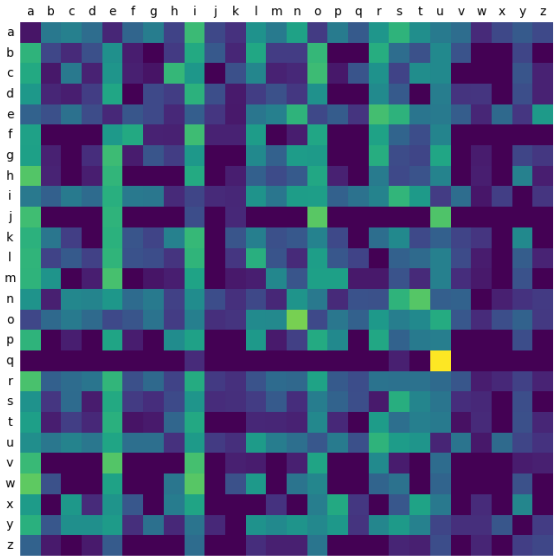

Figure 2 – Image matriciel du suivi des lettres dans le dictionnaire que nous utilisons, réalisé à partir d'un programme de David de ScienceEtonnante

II.3 Troisième génération, Enchaînement de trigrammes

Afin d'effectuer la méthode trigramme, nous utilisons deux constantes : **Alphabet** (la même que précédemment) et la constante **Voyelle**.

```
CONST       voyelle : WideString =  'aeyuioàâéèêëîïôùûüÿæœ'
```

Figure 3 – Image de notre constante voyelle

Lorsque le nombre de caractères est compris entre 1 et 3, la constante va nous permettre de rendre notre algorithme plus précis. En effet les mots courts sont souvent composés de voyelles comme par ***exemple**:"est"/"et"/"a"/"ou"*. Si jamais la probabilité qu'une lettre suivie de deux autres est nulle, avec la méthode trigramme, nous utilisons le digramme pour générer les caractères. Si il y a encore un problème alors on génère un mot aléatoire avec au moins une voyelle. Cela va nous permettre de rendre plus précis notre algorithme et que les petits mots de 1 à 3 lettres sonnent français.

Lecture du Dictionnaire

Pour cette étape, il y a peu de différences avec la méthode digramme. Nous allons ouvrir le fichier texte que nous voulons définir comme dictionnaire, nous avons eu de nombreux problèmes sur cette partie. Dans un premier temps notre lecture de dictionnaire respectait les consignes imposées, mais celle-ci était très lente, environ 1000 mots par seconde et sachant que notre dictionnaire fait 330 000, l'exécution du programme était infernale. Nous avons donc opté pour une fonction plus longue et hors-consignes de plus de 40 lignes de codes pour la lecture, nous étions donc à une lecture de 60 000 mots par seconde, ce qui prenait environ 5 secondes pour lire notre dictionnaire complet. Après plusieurs modifications, en utilisant des procédures au lieu de fonctions, nous avons réussi à conserver cette rapidité de lecture tout en réduisant notre code à 25 lignes, nous permettant ainsi de respecter les consignes.

Construction de la table de Fréquence

Suite à la lecture du dictionnaire, nous allons construire notre table de fréquences. Nous créons un Type ***DebutFin*** qui sera dans un tableau à une dimension. Il s'occupera de la lettre qui commence et celle qui termine un mot. Nous créons aussi un Type ***TableauTrigramme*** qui est un tableau en 3 dimensions avec la probabilité où 3 lettres commencent (Et 2 lettres pour la couche 0 pour le digramme). Ainsi que le nombres de fois où 3 lettres finissent un mot et la probabilité que la lettre est suivie des deux précédentes. Dans ce type la variable *compteur* est le nombre de fois où, une lettre est derrière les deux autres qui la précèdes.

```
Type Debut_Fin = record //Tableau 1 à 43 qui gère les premières et dernières lettres
            DebutEtFin: integer;
            Debut : integer;
            Fin : integer;
            End;
    TableauTrigramme = record //Tableau utiliser pour le digramme et trigramme
            Debut: integer;
            Fin: integer;
            Compteur: integer;
            End;
    Trigramme = array [1..43] of array [1..43] of array [0..43] of TableauTrigramme;
    Tableau2D = array [1..43] of array [1..43] of integer;
    Tableau3D = array [1..43] of array [1..43] of array [1..43] of integer;
    DebFin = array [1..43] of Debut_Fin;
    LesTableaux = record
            Tri : Trigramme;
            DF : DebFin;
            End;
```

Figure 4 – Image de notre table de fréquence

Génération des lettres et des mots

Selon ce que l'utilisateur demande, la méthode trigramme ne générera pas les mots de la même façon.

-Dans le **premier** cas possible, l'utilisateur demande un nombre de lettres fini. Par exemple, pour générer des mots à 6 lettres, le programme va, d'après la table de fréquence, choisir aléatoirement les 3 premières lettres en vérifiant qu'elles soient bien différentes les unes des autres. Ensuite, le programme va choisir les 2 prochaines lettres en fonction des précédentes. Puis la dernière lettre va être choisie en fonction des 2 précédentes, afin que ce soit le plus français possible.

-Dans le **deuxième** cas, l'utilisateur ne précise pas le nombre de lettres voulues, le programme va générer les 2 premières lettres aléatoirement jusqu'à ce qu'elles soient différentes l'une de l'autre. Puis générer une lettre en fonction des deux précédentes, puis on génère un nombre aléatoire si ce nombre est dans la probabilité que la lettre générée finisse alors le mot s'arrête sinon, le programme génère une lettre en fonction des 2 précédentes et test si le mot se termine. Et ainsi de suite. Les mots générés sont plus français, mais de taille totalement aléatoire.

II.4 Génération de phrases

Pour générer une phrase, notre algorithme va tout d'abord analyser le fichier *article.txt* ainsi que *nomCommun.txt* ou uniquement *nomPropre.txt*, afin d'appliquer la méthode trigramme pour créer le sujet de notre phrase. Une fois généré, nous allons construire notre phrase autour de ce sujet. Ensuite, notre algorithme va lire le fichier *verbePremierEtDeuxiemeGroupe.txt* et en générer un avec la méthode trigramme, puis l'accorder. Il continue avec la lecture du fichier *adjectif.txt* et cette fois en, choisit simplement un aléatoirement et l'accorde en fonction du sujet. Pour finir s'il y en a un, notre algorithme lis le fichier *adverbe.txt* et en choisit un aléatoirement. On aura : 1/3 chance d'avoir la forme **'sujet+verbe'** et 1/3 chance d'avoir **'sujet+verbe+adjectif'** et 1/3 chance d'avoir **'sujet+verbe+adverbe+adjectif'**.

Conjugaison du verbe

Quand le verbe est généré, il est à l'infinitif se terminant soit en **-er**, ou en **-ir** mais dans certains cas il peut ne pas se terminer ainsi. Alors il sera alors considéré comme verbe du second groupe. On aura donc un verbe à l'infinitif, une fois que l'algorithme a identifiée le groupe du verbe on enlève les deux dernières lettres pour n'avoir que le radical. Puis nous pouvons distinguer quatre cas :

-Sujet au singulier et verbe du premier groupe -> terminaison en *'e'*

-Sujet au pluriel et verbe du premier groupe-> terminaison en *'ent'*

-Sujet au singulier et verbe du second groupe->terminaison en *'it'*

-Sujet au pluriel et verbe du second groupe->terminaison en *'issent'*

La terminaison adéquate sera ensuite ajouté au radical. L'accord de l'adjectif est similaire, selon le sujet de notre phrase (singulier ou pluriel), on y ajoute un *'s'* ou non, etc.

II.5 Optimisation de notre code

Au tout début du projet, à la vue du sujet et de la quantité de travail, nous avions pour but principal de rendre un programme fonctionnel et qui puisse être compilé. Mais au fur et à mesure de l'avancement, nous avons voulu optimiser notre code pour respecter les consignes attendues et surtout réduire sa taille. Ainsi, nous avons revu la méthode du digramme. En effet pour réduire notre code, nous utilisons seulement une partie du trigramme pour réaliser le digramme. Le système est donc à quelque détails près le même qu'expliqué ci-dessus. Simplement pour optimiser notre code nous avons plus que 3 variables pour le digramme, il existe une couche 0 dans notre trigramme qui correspond au digramme. Mais nous ne pouvions pas faire cela avant la création et la réflexion sur le trigramme. Si nous avions manqué de temps, nous aurions rendu un code bien plus long avec le digramme séparé du trigramme et le raisonnement expliqué dans le paragraphe : Deuxième génération, Enchaînement de digrammes

III Excuter notre programme

Pour exécuter notre programme, il faut tout d'abord ouvrir un terminal et le compiler :
fpc projet.pas puis l'exécuter avec la commande ***./projet.pas***.
Devant nous apparaissent donc les instructions pour exécuter notre programme, il ne reste plus qu'à choisir la méthode que l'on souhaite utiliser pour générer des mots. On va ensuite retaper la commande en y ajoutant les instructions que l'on souhaite, par exemple pour lancer notre programme avec la méthode trigramme et avoir des mots de 6 lettres, on écrit :
./projet -n -s 6 -t dico où "dico" correspond au dictionnaire avec lequel on va faire notre trigramme.

Il n'y a aucun ordre particulier à respecter, on doit simplement choisir une des 3 méthodes ***-a ou -d ou -t*** avec ***-n*** plus un nombre de mots que l'on souhaite générer. Par défaut il y en a 100. Si on le souhaite, on peut ajouter l'argument ***-s*** plus ESPACE un nombre pour préciser le nombre de lettres/caractères de nos mots(Si il n'y a aucun nombre après ***-s*** le programme générera respectivement 3 et 4 lettres pour la méthode digramme et trigramme). Si l'on veut générer une phrase, on met seulement l'argument ***-p*** ce qui va nous générer une phrase aléatoire pouvant contenir :

-sujet+verbe

-sujet+verbe+adjectif

-sujet+verbe+adverbe+adjectif

Le sujet est aléatoirement article+nomCommun ou juste nomPropre.

IV Bonus

Pour rendre notre programme plus agréable à exécuter, nous avons fait en sorte que le terminal supprime les lignes inutiles qu'il affiche habituellement, afin qu'il ne reste plus que le résultat. De plus, nous avons fait en sorte d'afficher en temps réel l'analyse du dictionnaire, pour occuper l'utilisateur pendant la lecture de celui-ci. En effet notre terminale n'est pas vide durant les calculs de l'ordinateur. De plus, nous avons ajouté des indicateurs sur chacune des étapes que doit faire le programme afin de savoir si elles sont correctement exécutées.
Exemple : *Probabilités Début/Fin...Fait*
Afin d'améliorer l'expérience de l'utilisateur nous avons aussi ajouté des couleurs aux différentes parties qui seront affichées dans le terminal afin de facilement différencier *l'analyse du dico/les mots créés* et *l'analyse du dico/la phrase générée.*
Enfin au lieu de simplement piocher dans les listes (adverbe, adjectif etc) mis à notre disposition nous avons choisi d'analyser les fichiers et d'utiliser la méthode trigramme afin de générer de nouveaux noms propres ou communs de nouveaux articles et verbes.

V Conclusion

Ce projet fut intéressant, nous aurions aimé un sujet avec une interface graphique, mais finalement le résultat est amusant et nous a permis d'apprendre sur la langue française et le pascal. Mais ce projet ne fut pas de tout repos, en effet cela n'a rien à voir avec ce que nous avons fait depuis le début de l'année. Il a fallu être organisé autant dans la structure que dans la communication afin de pouvoir avancer le projet. De plus, nous avons dû repenser et changer de nombreuses fois le raisonnement et la façon dont notre programme fonctionne. Heureusement, notre groupe n'a pas eu de soucis d'entente et de partage du travail, nous avons su nous organiser correctement et avancer malgré la difficulté du sujet et les obstacles posés par le langage de programmation : pascal.

Bibliographie

[1] Dictionnaire Français *https://sciencetonnante.wordpress.com/2015/10/16/la-machine-a-inventer-des-mots-video/* David, le 16 octobre 2015

[2] Dictionnaire Français *http://www.pallier.org/liste-de-mots-francais.html* Christophe Pallier, 12/02/1999

[3] Pour les commandes que nous ne connaissions pas, nous les avons trouvé sur *https://www.freepascal.org/*

[4] forums.futura-sciences *https://forums.futura-sciences.com/*

[5] Wikibook *LatexWikibook* 2018