Rapport projet C : Générateur automatique de phrases

PARIS PANTHÉON-ASSAS UNIVERSITÉ

Ce rapport vous est présenté par SIMOES Nathan, RILI Adam, DEBORD Timothée

L2 – Groupe A promotion 2026 12/11/2022

Table des matières :

Introduction

Au cours du semestre 3 à EFREI, il nous a été demandé de réaliser un projet de programmation implémenté en C. Celui-ci a eu pour objectif de nous familiariser avec l'utilisation des différentes structures de données enseignées ce semestre. C'est-à-dire, entre autres, les listes chaînées et les arbres.

Pour réaliser ce générateur automatique de phrases, des documents nous ont été fournis afin de respecter les attentes de nos professeurs. Ceux-ci nous ont permis de ne pas s'écarter du sujet et nous ont amenés sur les pistes intéressantes. Ainsi, nous avons pu entamer notre réflexion sur les bonnes structures de données dès le début du projet.

À la suite des cours de M. Chabchoub, nous sommes ressortis avec de nombreux outils utiles pour ce projet. Ayant abordé les arbres, leur fonctionnement, mais aussi les listes chaînées que nous avions déjà vu l'année précédente. Cependant, nous avons approfondi nos connaissances sur ces structures de données afin de les maîtriser et de pouvoir les utiliser sans problèmes.

Au cours du mois de travail qui nous a été donné pour travailler sur ce projet, nous avons eu de nombreuses idées quant à l'utilisation des différents outils que nous avions à notre disposition. Nous sommes passés parfois par des impasses, mais nous nous sommes corrigés afin de mener à bien ce projet que nous vous présentons aujourd'hui. Il y sera abordé chaque partie du cahier des charges ainsi que l'implémentation utilisée à chaque fonction.

Enfin, nous parlerons de notre expérience personnelle concernant ce projet. Nous exprimerons notre avis sur le travail effectué, les bénéfices de ce devoir et comment le travail en groupe s'est déroulé.

Partie 1 – Structures de données

Cette première partie est consacrée à l'explication détaillée des structures de données sélectionnées pour implémenter nos fonctions.

Dans un premier temps, nous avions réfléchi ensemble pour trouver la manière la plus optimisée et efficace d'implémenter une structure répondant à toutes les contraintes du sujet. C'est pourquoi nous somme partie au début sur un graph avec une matrice d'adjacence. Néanmoins, il a été décidé que comme la notion n'a pas été vu en cours elle devra être laissé de côté nonobstant son optimisation pour ce type de problème.

Nous avons donc réfléchi entre une implémentation en Premier-Fils Frère-Droit, ou alors celle que nous avons choisi, un arbre N-aire avec comme enfants un tableau dynamique contenant la suite de notre arbre. Après avoir longuement mis en opposition ces deux méthodes, nous pouvons notamment justifier notre choix par la compréhension du code qui ressemble le plus à ce que l'on fait en cours avec un arbre N-aire classique ce qui nous avait été reprocher avec notre première idée du graph.

Pour ce qui est des structures en elles-mêmes voici comment elles s'organisent.

```
================================================================
                        STRUCTURE POUR LES VERBES
================================================================
*/
//pour stocker les formes flechits dans une liste chainee
//ainsi que les parametres de cette forme
typedef struct FF_VB
{
    char *ff;           //string du du decalage entre fdb et ff
    int diff;           //decalage entre la fdb et la ff
    char *personne;
    char *conjugaison;
    char *nombre;
    char *genre;
    struct FF_VB *next;
}Fvb;
```

```c
//pour stocker le nombre de forme de base
//ainsi qu'un pointeur vers la liste chainee des formes de base
typedef struct VB
{
    char lettre;
    int nbenfant;     //nombre d'enfant donc de lettre suivante
    int nbflechit;      // !=0 quand end == 1
    int end;            // est == 1 quand fin de forme de base
    struct FF_VB *ff;   // si end==1 alors *ff != NULL
    struct VB **child; //toutes les enfants donc lettre suivant
}Vb;

typedef struct ROOT_VB{
    int nbenfant;
    struct VB **child;
}RVb;
```

- Une racine (ROOT_) qui permet de stocker notre premier niveau de l'arbre. Cette racine nous sert notamment pour le backtracking de nos fonctions qui doivent chercher un mot selon certaines conditions. Mais aussi dans la gestion de notre mémoire et de notre arbre afin de permettre une meilleure représentation globale de la structure.

- Un nœud (ici VB mais est aussi NOM, ADJ et ADV). Ce nœud est la structure globale de notre arbre qui va contenir toutes nos formes de base. Il contient le caractère "lettre" qui stock la lettre d'un mot, un entier "nbenfant" qui est un compteur de nombre d'enfant qui va nous permettre de nous déplacer dans nos enfants qui sont stoker dans un tableau dynamique "child" dans l'ordre de rencontre dans le dictionnaire ici alphabétique, un autre entier "nbflechit" qui contient, dans un nœud qui a "end" égal a 1 un indicateur si c'est la fin d'un mot, le nombre de mot dépendant de cette forme de base et enfin "ff" qui est un pointeur vers une autre structure permettant le stockage de nos formes fléchies.

- Une liste chaînée (FF_), qui contient diffèrent champs selon le type de mot. Mais ont en commun une chaîne de caractères “ff” qui est la terminaison de notre mot donc la différence avec notre forme de base, un entier “diff” qui va nous permettre de reconstruire nos chaînes jusqu’à la fin des caractères communs entre une forme de base et une forme fléchie. Des champs en chaîne de caractères qui nous permette de stocker les paramètres des formes fléchie puis un “next” qui permet de faire créer la linéarité de notre chaîne simplement chaînée.

Partie 2 : Générateur automatique de phrases

- ## La fonction de lecture

Afin de traiter le fichier texte fourni comportant environ 300 000 lignes, il a fallu réfléchir dans un premier temps à la manière optimale d'exploiter ce fichier afin de faire les bons appels et les traitements de manière efficace.

En effet, la syntaxe du fichier fourni est bien définie, cela a grandement facilité son traitement étant donné que nous pouvions détecter les différents éléments séparant forme de base, forme fléchie et paramètres.

1. Gestion du fichier texte fourni

Pour lire notre dictionnaire, nous avions besoin du chemin d'accès du fichier, cependant, celui-ci diffère entre CLion et linux étant donné que CLion créé un fichier exécutable en dehors du projet. Cette étape fut handicapante puisque nous devions à chaque fois changer la source du fichier pour effectuer nos tests.

Une fois ce problème diagnostiqué et résolu, nous avons utilisé la fonction fopen() en mode read ("r") qui renvoie un pointeur au début du fichier texte. Pour éviter toute erreur de lecture, une condition a été ajoutée afin de s'assurer que le fichier dictionnaire.txt est bien ouvert.

Par la suite, nous avons initialisé toutes les variables qui nous seront utiles au traitement de ce fichier. Concernant les quatre strings utilisées, on retrouve la forme fléchie, la forme de base, les paramètres et le type. Pour éviter une consommation de mémoire excessive lors de l'exécution du programme nous avons décidé d'utiliser les calloc() avec un nombre conséquent permettant le traitement de la plus grande string du dictionnaire. Nous avons choisi la fonction calloc(), car lors de l'exécution de notre programme avec malloc(), malgré plusieurs free, certaines strings étaient initialisées avec une valeur.

La fonction utilisée afin de traiter le dictionnaire est la fonction fgetc(). En effet, il paraissait trop complexe de traiter la ligne dans son entièreté en utilisant fgets(); alors nous avons décidé de récupérer les caractères un par un. La fonction fgetc() est particulière, elle récupère le code ASCII de la lettre scannée,

il a donc été nécessaire de transformer cette variable dans un second temps en type “char”.

Concernant les boucles de traitement des différentes formes et paramètres, nous avons choisi d’utiliser des boucles infinies while(1) pour traiter tous les cas non-désirés dans la boucle.

Dans un premier temps, la forme fléchie et la forme de base sont séparées par une tabulation (“\t”), notre condition d’arrêt pour la boucle infinie a donc été de vérifier si le caractère récupéré est une tabulation ou non. D’autre part, pour éviter toute erreur éventuelle de lecture, nous avons ajouté deux conditions d’arrêt supplémentaires : si le caractère récupéré est la fin du fichier (EOF = End Of File) et si le caractère récupéré est un espace (32 en code ASCII). Les conditions d’arrêt étant bien définies, il ne reste plus qu’à convertir notre valeur de lettre en caractère et de l’ajouter à notre string stockant la forme fléchie.

Pour cela, nous avons décidé d’utiliser une fonction codée à la main afin de contrôler son rôle : mystrcat. Il s’agit d’une fonction permettant de concaténer deux chaînes de caractères l’une à la suite de l’autre. Cette fonction récupère la longueur de la première chaîne à l’aide d’une autre fonction codée par nos soins : mystrlen. Ensuite, puisque nous n’ajoutons qu’un seul caractère à la suite de notre chaîne, nous avons utilisé la fonction realloc avec comme paramètres la première string et la longueur de celle-ci à laquelle nous ajoutons la taille d’un caractère (1 * sizeof(char)).

Suite à de nombreuses erreurs concernant les paramètres se situant en fin de ligne dans le fichier, nous avons décidé de récupérer la valeur du caractère à la fin de nos boucles, pour pallier le problème du premier caractère nous avons effectué un premier fgetc() avant le début des boucles.

Dans un second temps, nous récupérons la string correspondante à la forme de base. Pour cela, nous répétons une deuxième fois la boucle infinie vue pour la forme fléchie puisque la forme de base et les paramètres sont aussi séparés par une tabulation.

Une fois la forme fléchie et la forme de base récupérées, il ne nous reste plus qu’à traiter les paramètres et le type de notre mot. Cela nous permettra d’effectuer les bons appels de fonctions par la suite pour éviter des traitements supplémentaires dans une autre fonction.

2. Séparation des données

C'est la dernière section de chaque ligne du dictionnaire qui nous a posé un problème. En effet, lors de nos tests, nous nous sommes rendu compte grâce au débuggeur GDB que la string paramètres comptabilisait un "\r" or nous ne savions pas que cela pouvait se manifester en plus du "\n". Une fois ce problème surmonté, nous avons pu séparer le type (Verbe, nom, adjectif, adverbe) en détecter les premiers doubles points qui nous permettent d'adresser la bonne fonction de traitement pour l'arbre correspondant.

Ensuite, pour traiter toutes les exceptions, il était possible de rencontrer dans le dictionnaire des prépositions, des adverbes ou des verbes à l'infinitif. Alors, les doubles points n'apparaissaient pas à ce moment-là. Nous avons donc rajouté une boucle avec une variable temporaire qui est mise à 1 lorsque le "\n" et détecté.

La boucle suivante, qui récupère les paramètres, repère cette variable temporaire et s'arrête si celle-ci est à 1.

La fonction permettant l'adressage de la bonne fonction d'insertion dans l'arbre est strcmp(). Cette fonction retourne la valeur 0 si les deux strings comparées sont identiques. La dernière section d'une ligne du dictionnaire commençant toujours par Ver, Nom, Adj, Adv ou encore Pre etc... Il suffisait de comparer notre type récupéré à une string comportant les types initialisés à la main. Enfin, en fonction de la valeur retournée par strcmp(), nous appelons la fonction d'insertion correspondant à l'arbre Ver, Nom, Adj ou Adv. Etant donné que le sujet du projet ne nous demandait pas de traiter les prépositions, nous n'avons pas implémenté nos fonctions pour les insérer dans un nouvel arbre.

Grâce aux traitements effectués auparavant, les fonctions appelées peuvent directement entamer l'insertion puisqu'elles prennent en paramètres les strings nécessaires : forme fléchie, forme de base et paramètres, sauf pour les adverbes ou il n'y a pas de paramètres.

Enfin, nous appelons la fonction free() sur nos variables initialisées avec calloc() et nous remettons ces pointeurs à nul pour entamer une nouvelle ligne du fichier dictionnaire.

Lorsque le traitement du dictionnaire est complet, nous arrivons au “caractère” End of File, ce qui nous permet de fermer le fichier avec la fonction fclose().

Finalement, une seule fonction est appelée dans le main pour le traitement du fichier dictionnaire :

```
void dico_read(char *dico, RVb *v_tree, RNom *n_tree, RAdj *adj_tree, RAdv *adv_tree);
```

La génération des arbres est gérée dans cette fonction par la suite.

- **Génération des arbres**

La génération des arbres est une partie importante de ce projet. En effet, c'est dans ces arbres que sont stockés tous les mots du dictionnaire qui nous est fourni.

Etant donné le fait que le dictionnaire contient quatre types de mots, on a donc décidé de créer quatre arbres différents (un pour chaque type). Ce sont tous des arbres de types N-aires. Ils sont constitués d'une racine, de nœuds (contenant chaque lettre d'une forme de base) et de LSC à la fin de chaque forme de bases (contenant les formes fléchies).

Une fonction principale, que l'on a découpée en plusieurs fonctions, s'occupe de créer les arbres (par exemple pour les verbes : `void insertTreeVb(RVb* root, char* temp1, char* temp2, char* temp3)`)

1. Création des nœuds

La première étape de la création d'un arbre, passe par la création de la racine et de ses fils. En effet, si la racine, que l'on met en paramètre dans notre fonction principale, est vide, ou si la première lettre de la forme de base que l'on souhaite insérer ne se trouve pas dans la racine, cela veut donc dire que l'on doit créer de nouveaux nœuds et y insérer chaque lettre de la forme de base à partir de cette racine.

Et c'est cette fonction qui s'occupe de vérifier cela : `isValInTabVb(Vb** tab, int length, char flettre)`

Elle va vérifier dans un tableau dynamique de pointeurs, si la lettre que l'on cherche se trouve bien dans un des nœuds pointés par une des cases de ce tableau. Si elle la trouve, elle retourne alors l'indice de la case pointant vers le nœud la contenant. Sinon elle retourne –1.

Cette fonction est très utile. En effet, chaque adresse des fils d'un nœud est stockée dans le tableau dynamique de celui-ci. On a donc besoin de cette fonction à chaque fois que l'on veut vérifier si une lettre de la forme de base se trouve ou non dans un des fils de ce nœud (si l'on parcourt ou si l'on ajoute un fils).

Pour ce qui est de la création des nœuds, on utilise trois fonctions différentes (par exemple pour les verbes) :

- newNodeVb, qui créé un nouveau nœud en l'initialisant et en y insérant un caractère désiré, et qui le retourne.
- initStructVb, qui initialise un nœud insérer en paramètre.
- createNodeVb, qui créer un nouveau nœud en l'insérant dans le tableau de fils du nœud mis en paramètres.

On a dû séparer en quatre cas différents la création des nœuds :

- Le cas où la première lettre de la forme de base ne se trouve pas dans le tableau des fils de la racine.
- Le cas où la première lettre de la forme de base s'y trouve. Dans ce cas-là, on effectue le parcours de l'arbre en insérant des nœuds si besoin.
- Après le parcours, le cas où l'on se trouve à la fin de la forme de base (on ajoute les formes fléchies)
- Le cas où l'on ne se trouve pas à la fin de la forme de base mais dans un nœud (on insère et parcourt de nouveaux fils, jusqu'à la fin de la forme de base).

Une fois la création de nœuds pour chacun des quatre différents cas, on doit insérer les formes fléchies correspondant aux formes de bases.

2. Gestion des listes de formes fléchies

Dans un arbre que l'on a créé, les formes fléchies ainsi que leurs caractéristiques, sont stockées à la fin de chaque forme de bases dans des LSC. On utilise donc quatre fonctions pour la création des LSC et l'insertion des formes fléchies et des caractéristiques (par exemple pour le cas des verbes):

- La fonction Calculer_Diff_et_ffVb, qui va retourner les caractères de différences entre la forme fléchie et la forme de base ainsi que son nombre de caractères.
- La fonction insertCaracVb, qui récupère les caractéristiques d'une forme fléchie et qui les insère dans la cellule de la forme fléchie. (*)
- La fonction addFvb. Fonction récursive qui ajoute une ou plusieurs cellules remplie (en appelant Calculer_Diff_et_ffVb et insetCaracVb) dans une LSC.
- La fonction createFirstFvb, qui créer la première cellule et celles qui suivent si y en a, et qui les relies à la LSC du dernier nœud de la forme de base (en appelant notamment les trois fonctions ci-dessus).

(*) Pour les verbes, on a remarqué que les caractéristiques étaient sous quatre formes différentes :

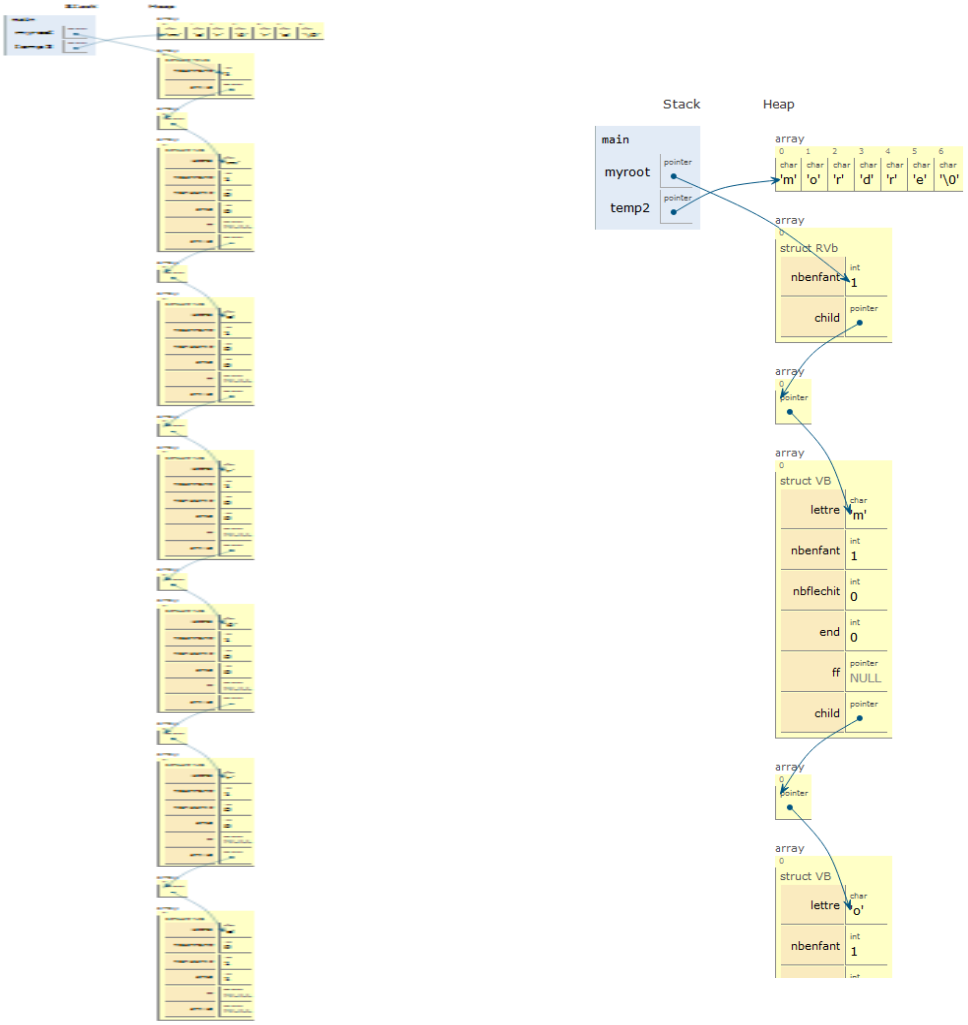
- “Ver:IImp+PL+P1:SPre+PL+P1:SImp+PL+P1”
- “Ver:PPre”
- “Ver:PPre+Mas+SG”
- “Ver:IImp+PL+P3”

On a pris en compte ses quatre formes pour la fonction “insertCaracVb”.

Dans la fonction principale, on utilise ces fonctions pour insérer les formes fléchies et leurs caractéristiques, en prenant en compte si on doit insérer une LSC, ou si on doit insérer à partir d’une cellule de la LSC.

L’arbre des adverbes fait exception pour les formes fléchies, vu que la forme fléchie d’un adverbe est égale à sa forme de base. Donc les fonctions ci-dessus ne sont pas utilisées pour l’arbre des adverbes.

3. <u>Représentation d’un arbre sur C Tutor</u>



Voilà à quoi peut ressembler un arbre en mémoire. Malheureusement, les fonctions des formes fléchies et la création d'autres fils, dépassant la limite de ce que C Tutor peut avoir en ligne de code, on n'a pu faire que la représentation de l'arbre avec une seule forme de base.

Une fois l'arbre créé et la fonction de lecture prête, on peut aborder la génération des phrases.

- **Génération des phrases**

Pour pouvoir générer nos phrases, nous avions besoin de parcourir nos arbres et les listes chaînées des formes fléchies. Certaines nécessitaient d'implémenter un backtracking. Pour monter la compréhension du cours et des algorithmes de parcours basique les fonctions implémenter sont aussi bien en récursif qu'en itératif.

1. Recherche des types dans les arbres

On traverse notre arbre de manière aléatoire jusqu'à arriver sur une feuille ou alors la fin d'un mot, mais qui n'est pas une feuille. Ensuite, on va chercher dans les formes fléchies de nœud s'il forme peux s'accorder avec le nom ou alors l'adjectif ou encore le verbe de la phrase. Si ce n'est pas sur une feuille la récursion continue, si c'est sur une feuille alors il sort de la fonction et repart de la racine grâce à une fonction chapeau avec un while qui nous permet de relancer tant qu'on ne trouve pas la bonne forme. On récupère à la fois une chaîne de caractère qui est la forme de base ou fléchie et un pointeur sur le nœud qui contient cette forme.

2. Accords des adjectifs, verbes avec le nom sélectionné

Comme précisé plus haut, nous récupérons dans un pointeur la bonne forme fléchie. Nous l'utilisons donc pour pouvoir conjuguer notre phrase. Encore une fois le principe de backtracking est utilisé en remontant à chaque fois dans la fonction chapeau jusqu'à ce qu'une forme correspondante soit trouvée.

- **Recherche de mots parmi les formes de bases**

Une autre tâche importante qui nous a été demandée de faire, était la recherche de mots parmi les formes fléchies. Or, nous manquions de temps sur le projet et nous n'avons pu faire la recherche de mots que parmi les formes de bases.

Les quatre types d'arbres ayant été créés, on peut faire la recherche de mots parmi les formes de bases. Pour ce faire, on a créé et utilisé 4 fonctions différentes (une pour chaque type) :

- Nom_Présent, pour vérifier si un nom est présent dans le dictionnaire.
- Verbe_Présent, pour vérifier si un verbe est présent dans le dictionnaire.
- Adj_Présent, pour vérifier si un adjectif est présent dans le dictionnaire.
- Adv_Présent, pour vérifier si un adverbe est présent dans le dictionnaire.

Ces quatre fonctions recherchent dans les enfants, à partir de la racine de chaque type, lettre par lettre dans les mots entrés en paramètres et retourne 1 s'il trouve le mot et 0 sinon.

Finalement, dans le main, à l'aide de différentes conditions, on utilise ces fonctions en fonction de ce qu'elles retournent pour faire la recherche de mots parmi les formes de bases, pour les quatre différents types.

- **Expérience personnelle**

Répartition des tâches :

<table>
<tr><td></td><td>Structures</td><td>Fonction de lecture</td><td>Génération des arbres</td><td>Génération des phrases</td><td>Recherche de mots parmi les formes de base</td><td>Main</td></tr>
<tr><td>Nathan</td><td rowspan="3">Travail de réflexion et d’implémentation à part égale pour les 3 membres du groupe</td><td colspan="2">Code de la génération des 4 arbres</td><td colspan="3"></td></tr>
<tr><td>Adam</td><td colspan="2"></td><td>Code de la recherche de forme de base aléatoire et chercher une forme fléchie conjuguée selon un nom choisit aléatoirement</td><td>Code de la fonction pour trouver une forme de base dans nos arbres</td><td></td></tr>
<tr><td>Timothée</td><td>Code la fonction de lecture et les appels pour la génération des arbres</td><td colspan="3"></td><td>Génération du menu pour une navigation claire, initialisation des variables pour la fonction de lecture</td></tr>
</table>

Pour ce projet, nous avons décidé de réfléchir dans un premier temps sur les structures que nous allions utiliser. Pour cela, nous avons tous les trois

effectué un travail de réflexion et s'en est suivi un brainstorming afin de finaliser notre choix quant aux structures.

Pour le reste du projet, les fonctions demandées sont variées et demandent chacune une réflexion particulière, nous avons pensé qu'il serait plus compliqué de travailler ensemble sur les différentes fonctionnalités. Alors nous nous sommes réparti les tâches pour une plus grande efficacité.

Cependant, chaque membre du groupe est resté attentif aux autres si de l'aide était requise.

Grâce à cette répartition, nous avons pu finaliser les fonctionnalités demandées à temps.

Adam RILI :

Malgré une petite déception de ne pas pouvoir utiliser de graph, le projet reste intéressant. En termes de connaissance personnelle, je ne pense pas avoir appris de nouvelles choses, mais ce projet m'a permis de me remémorer certains principes algorithmiques basiques. Le fait de ne pas avoir de test unitaire et seulement des scanf() et printf() dans un menu n'est pas forcément réaliste. Il a fallu aussi pallier à CLion et ces problèmes dans notre groupe que ce soit sur l'affichage la compilation ou l'exécutable. Globalement, c'était une expérience assez satisfaisante et plaisante.

Nathan SIMOES :

J'ai beaucoup appris durant ce projet. En effet, en ayant réfléchi à la création des arbres N-aire et à leur implémentation, j'ai appris et compris le fonctionnement d'une telle structure. J'ai aussi mieux compris comment marche l'allocation dynamique, notamment la fonction de realloc. Ce projet m'a aussi permis de m'améliorer aux découpages en plusieurs fonctions. Et finalement, cela m'a permis de mieux comprendre comment fonctionne une fonction récursive et son utilité. Donc ce projet m'a permis de développer mes compétences en programmation ainsi que mes compétences organisationnelles, et aura été une expérience très enrichissante.

Timothée DEBORD :

Ce projet aura été très enrichissant sur de nombreux aspects. En effet, tout d'abord, nous avons mêlé plusieurs types de structures de données pour optimiser notre code au maximum. Ma compréhension quant aux arbres N-aires et aux listes chaînées n'en a été qu'améliorée. D'autre part, j'ai pu apprendre de nouvelles notions concernant la lecture d'un fichier en programmation C. De nombreuses syntaxes sont à connaître afin de manipuler le fichier parfaitement. D'une manière plus générale, ce projet m'aura permis d'ancrer les nouvelles notions vu ce semestre qui seront très utiles pour la suite de mon parcours. Après avoir pris connaissance des arbres N-aires et du fait que l'on puisse mélanger plusieurs structures de données pour différentes fonctionnalités, je compte approfondir ces notions afin de progresser davantage en programmation C, car c'est un code qui me plaît, étant très polyvalent et à la base de beaucoup d'autres codes.