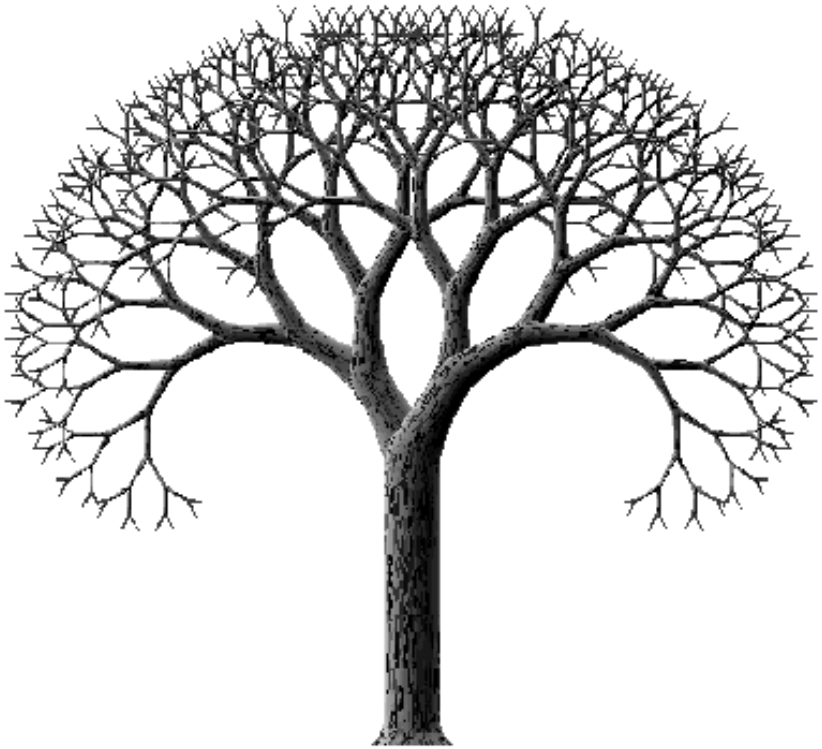
Rapport du projet : Greffes génétiquement modifiées d’un arbre binaire



Narmin Elkilani
Tatiana Desravines

1. Guide de l’utilisateur

- Objectif du programme
- Lancement du programme

2. Etat d'avancement du projet

- Explication des fonctionnalités réalisées

3. Organisation du travail

- Communication et synchronisation du groupe
- Répartition quantifiée des tâches
- Difficultés rencontrées

4. Conclusion

1. Guide de l'utilisateur

- ## Objectif du programme

Le but du programme est étant donné un arbre binaire A et un autre arbre binaire B, on fait la greffe de B sur A pour en produire un nouvel arbre.

- ## Lancement du programme

Pour lancer le programme suivez les différentes étapes :

Étape 1 : vous devez ouvrir le dossier DESRAVINES_ELKILANI .
Étape 2 : lancez un éditeur de code et votre terminal.
Étape 3 : utiliser la commande cd pour accéder au dossier cd DESRAVINES_ELKILANI sur le terminal.
Étape 4 : utiliser la commande : Make pour créer un exécutable.

Vous devez choisir entre deux options :

Étape 5 :
option -E : saage -E fichier.saage crée une sauvegarde dans le fichier .saage d'un arbre saisi par l'utilisateur au clavier.
Le code à saisir sera de ce type :

1 "mot_1\n" 1 "mot_2\n" 0 0 1 "mot_3\n" 0 0
Le programme sauvegarde le fichier dans le dossier data/.

Étape 6 :
option -G :
saage -G fichier1.saage fichier2.saage crée l'arbre du greffon fichier2.saage est appliqué à l'arbre source stockée dans fichier1.saage.
Le résultat de la greffe sera fourni sur la sortie standard au format.saage stocké dans le dossier data/.

2. Etat d'avancement du projet

- **Explication des fonctionnalités réalisées**

Module arbres_binaires:

- **Noeud * alloue_noeud(char * s) :**

  On alloue un nœud dans l'arbre contenant comme valeur le mot s, si l'allocation du noeud échoue on renvoie NULL, si l'allocation du mot s échoue on libère le nœud et renvoie NULL, sinon on renvoie l'adresse du nœud.

- **void liberer(Arbre * A) :**

  On libère l'arbre récursivement et on libère sa valeur.

- **Arbre cree_A_1(void) , Arbre cree_A_2(void), Arbre cree_A_3(void) :**

  On renvoie respectivement les arbres *A1* , *A2* et *A3* une fois créés en mémoire.

  On crée manuellement les 3 arbres fournis comme exemple, si l'allocation d'un nœud échoue on libère tout l'arbre et on renvoie NULL, sinon on renvoie l'adresse de la racine.

- **int construit_arbre(Arbre *a) :**

Saisir le code de l'arbre directement sur le terminal sous cette forme :
1 "arbre\n" 1 "binaire\n" 0 0 1 "ternaire\n" 0 0 **Nous avons gardé l'ancien code de saisie qui se trouve ci-dessous.**

3.1 Codage d'un arbre et création à la volée

Une manière de créer un arbre en mémoire est de le créer à la volée par une saisie de l'utilisateur suivant le parcours en profondeur préfixe de l'arbre que l'on souhaite créer. Un nœud vide sera représenté par l'entier 0, tandis qu'un nœud non vide sera représenté par l'entier 1, suivi de la chaine de caractère qu'il représente, placée entre des guillemets.

On notera que les chaines de caractères à l'intérieur d'un nœud peuvent contenir plusieurs mots, et même des nombres !

Par exemple, les séquences à saisir pour construire en mémoire les arbres A_1, A_2 et A_3 sont données dans Figure 2. Pour la saisie utilisateur d'une telle phrase, on pourra utiliser la fonction `fgets`.

Arbre A_1 :

```
1 "arbre\n" 1 "binaire\n" 0 0 1 "ternaire\n" 0 0
```

Pour cela , nous aurons besoin des fonctions suivantes:

- **Arbre construire_arbre_binaire(char * mot)**
  Création d'un arbre à partir du code de l'arbre . On renverra l'arbre en cas de succès sinon on renvoie NULL.

- **int creation_Memo(Memo *p)***
  Création d'une chaîne de caractère qui va contenir le code de l'arbre. On renverra 1 en cas de succès sinon on renvoie 0.

- **int creation_code_adapter_arbre(Memo * p)**
  Il crée un nouveau code de ce type :
  1 *arbre* 1 *binaire* 0* 0* 1 *ternaire*0 *0 **qui contient un unique séparateur pour chaque mot**
  Pour ce faire, on sépare chaque bloque à l'aide de parenthèse en utilisant *strtok*.
  On utilisera également :

int Copier_Chiffre(char *mot , Memo * p)

Pour traiter les blocs de zéro et d'un et il renvoie 1 si le bloc contient que des zéro ou des uns, sinon on renvoie 0 si le bloc contient un mot et on renvoie -1 si c'est ni l'un ni l'autre.

int verification_mot(char * mot)

On vérifie que le bloc de mot contient *"\n"*. Si c'est le cas, on renvoie 1 sinon on renvoie 0. Puis on termine de construire l'arbre à l'aide de la fonction *Arbre construire_arbre_binaire(char * mot)*.

-

int lire(char * chaine , int longeur , Memo * p)

La fonction permet de faire une saisie contrôlée, on renvoie 1 si la chaîne (le code de l'arbre) est correcte sinon 0.

Void vider_memo(char * code , int taille)

Si la saisie de la chaîne est incorrecte alors on supprime les éléments à l'intérieur de la chaîne.

Module greffe :

int copie(Arbre * dest, Arbre source):

On copie dans *dest l'arbre source,
Si *dest est vide on y alloue la mémoire, après on copie récursivement dans l'arbre gauche de *dest et dans l'arbre droite de *dest,
Si la longueur du mot de source est plus grande que celle de * dest, on réalloue de la mémoire au mot.
On retourne 0 si l'allocation échoue, sinon on retourne 1.

static int ajout_fg(Arbre *g, Arbre a) :

On ajoute le sous arbre gauche de a à tous les noeuds de *g sans fils gauche en utilisant la fonction copie(), on retourne 0 si copie échoue sinon on retourne 1.

static int ajout_fd(Arbre *g, Arbre a):

Idem que ajout_fg, on ajoute le sous arbre droit de a à tous les noeuds de *g sans fils droit en utilisant la fonction copie(), on retourne 0 si copie échoue sinon on retourne 1.

int expansion(Arbre * a, Arbre b) :

On greffe l'arbre b sur a, on adopte un parcours suffixe des feuilles vers la racine,
Soit n un noeud de a avec la même valeur que celle de la racine:

1. **on copie b dans g avec la fonction copie(),**
2. **on ajoute le sous arbre gauche de n à tous les noeuds de g sans fils gauche avec la fonction ajout_fg()**
3. **on ajoute le sous arbre droite de n à tous les noeuds de g sans fils droite avec la fonction ajout_fd()**
4. **n devient l'arbre g**

Après on remonte dans l'arbre a et on refait récursivement le même traitement si on on trouve un nœud avec la même valeur que celle de la racine.
La fonction renvoie 0 si copie() ou ajout_fg() ou ajout_fd() échoue, sinon on retourne 1

Module saage:

int serialise(char * nom_de_fichier, Arbre A)

La fonction écrit dans un fichier l'arbre a. En cas de succès, on renverra 1 sinon on renvoie 0 en cas d'échec.

Pour cela, on utilise les fonctions suivantes :

- **int test_format(char * nom_de_fichier) La fonction teste si le format du fichier est correct, il renvoie 1 sinon on renvoie 0.**
- **void affiche(Arbre A , FILE * out, int nombre_espace) écrit d'abord dans le fichier le sous-arbre gauche ensuite le sous-arbre droit en respectant les conventions données.**

int deserialise(char * nom_de_fichier, Arbre * A)

construit un arbre à partir d'un fichier en cas de succès, on renverra 1 sinon on renvoie 0.
Pour cela, on utilise les fonctions suivantes :

- **int creation_Memo(Memo *p) création d'une chaîne de caractère qui va contenir le code de l'arbre. On renverra 1 en cas de succès sinon on renvoie 0.**

- **Arbre construire_arbre_binaire(char * mot) création d'un arbre à partir du code de l'arbre . On renverra l'arbre en cas de succès sinon on renvoie NULL.**

Module main:

La ligne 53 à 55, l'initialisation des positions des fichiers pris en argument dans le main, un tableau de caractère qui contiendra le nom du fichier puis fichier sera utilisé dans la fonction *void nom(char * fichier , char mot[MAX_MOT])* retire le format du fichier.

Ligne 63 à 69, les fonctionnalités correspondant à l'option -E(avec une description plus détaillée dans la partie guide de l'utilisation) est qu'il prend en argument un fichier de format *".saage"* pour que le programme crée un arbre à partir de la saisie sur le terminal puis il sérialise l'arbre dans le fichier pris en argument.

Ligne 71 à 102,

Les fonctionnalités correspondant à l'option -G est qu'il prend en argument deux fichiers Puis on les déserialise le fichier pour obtenir deux arbres. Qu'on utilisera pour effectuer l'expansion de B dans A puis l'arbre sera sauvegarder dans un fichier *".saage"*

Bien entendu, à chaque erreur produite par le programme, on libéra les arbres produits dans le code.

3. Organisation du travail

- ### Communication et synchronisation du groupe

  On a communiqué sur discord et nous avons mis notre code sur github

- ### Répartition quantifiée des tâches

- **Narmin Elkilani a fait le module greffe, les fonctions alloue_noeud(), liberer() dans le module arbres_binaires et le makefile**

- **Tatiana Desravines a fait le module saage, les fonctions *construire_arbre()* dans le module arbres_binaires et le main.**

- **Difficultés rencontrées**

Narmin ELkilani **: faire le module greffe m'a pris deux semaines pour comprendre et trouver, et c'était challengeant et bénéficiant pour comprendre la récursivité et la structure des arbres.**

Tatiana Desravines **: gestion des cas d'erreurs et fonction *construire_arbre()* avec la fonction *fgets* m'a pris du temps pour comprendre à créer un arbre binaire en une seule saisie.**

4. Conclusion

Le projet était intéressant et challengeant et nous a aidé à bien comprendre la structure des arbres binaires et la récursivité.

Lien du github pour le projet:
https://github.com/nermine11/Genetically-modified-grafts-of-a-binary-tree

Merci pour votre attention