Université Gustave Eiffel

Algorithmique des arbres

NAVARRO Antony **TP-5**
MUNAITPASOV Maksat **TP-5**

Greffes des arbres binaires

L2 - Informatique, semestre 2

Année universitaire : 2023-2024

Table Des Matières

Guide De l’Utilisateur

Compilation du programme : **make**

Lancement du programme : **./saage**

Nettoyage du programme: **make mrproper**

Usage :

Pour lancer les tests automatique, après avoir compilé tout, il faut écrire la commande:

$ clang -std=c17 -pedantic exemples/tests_prof.o build/option.o build/arbres_binaires.o build/saage.o build/greffe.o -o tests_prof

$ valgrind ./tests_prof

Le programme peut être utilisé avec différentes options pour effectuer des opérations spécifiques sur les arbres et les greffes.

Pour simplifier la vie, vous pouvez écrire **fichier.saage** ou **exemples/fichier.saage**.

Voici les options disponibles :

- **-G** : Affiche la greffe de l'arbre **chemin_arbre** dans l'arbre **chemin_greffe**.
- **-E** : L'entrée standard de l'utilisateur (d'après le clavier ou un fichier non saage) est ajoutée à **chemin_fichier**.
- **-DOT** : Prends un chemin valide et affiche l'arbre grâce à la fonction DOT.
- **-BIG** : Prends un chemin valide (**grand.saage** ou **immense.saage**) et renvoie la greffe d'un arbre et affiche avec DOT.

Exemples d'utilisation:

N'hésitez pas à utiliser Valgrind pour analyser l'utilisation de la mémoire.

Soit i = {1, 2, 3} et X = {B, C, D},

$ ./saage -G A_i.saage X.saage

$ ./saage -E resultat_usr.saage < exemples/usr_A_i.txt

$ ./saage -E resultat_usr.saage

$ ./saage -DOT fichier.saage

$ ./saage -BIG immense.saage

Présentation Technique Du Projet

Pendant ce projet, nous avons utilisé le C, Makefile, Dot, ainsi que les fichiers ".saage", ensuite, nous avons établi nos conventions, comme par exemple l'utilisation du nom *fptr pour "file pointer" (pointeur de fichier)*.

En d'autres termes, nous avons écrit la plupart des fonctions utiles telles que strstr(), strdup(), strcmp(), strchr(), strlen(). Nous les avons nommées comparer_chaines(), recherche_substring(), recherche_lettre(), dupliquer_string(), et len_string().

Nous pensons que ces noms de fonctions sont plus explicites. Ensuite, nous avons la fonction ***arbre_de_fichier_aux(FILE * __restrict__ fptr, Arbre * __restrict__ arbre)***, qui lit le fichier ".saage" et construit l'arbre associé. Bien que cette fonction puisse sembler complexe, les commentaires et les noms appropriés des fonctions la rendent compréhensible.

Ensuite, nous avons la fonction ***arbre_de_fichier(char * __restrict__ path)***, qui prend le chemin du fichier en entrée, l'ouvre en mode lecture, et renvoie l'arbre construit par la fonction précédente.

La fonction ***expansion(Arbre *dest, Arbre source)*** peut susciter des questions. D'abord, sa valeur de retour est un entier non signé (unsigned int) car elle renvoie 1 si tout se déroule bien et 0 sinon. Il s'agit d'une fonction récursive qui modifie *dest de telle manière qu'après son exécution, celui-ci contienne la greffe de l'arbre source sur *dest. Des commentaires ont été ajoutés dans la fonction pour faciliter la compréhension du code.

En ce qui concerne l'écriture des fichiers ".saage", nous avons besoin d'un flux (en mode écriture) ainsi que d'un arbre. La fonction correspondante est nommée ***ecrire_fichier_saage(FILE * __restrict__ fptr, Arbre __restrict__ arbre, uint count_tab)***. Comme son nom l'indique, cette fonction est chargée d'écrire sur un flux.

Enfin, pour créer un fichier ".saage", nous utilisons la fonction ***int serialise(char *path_create, Arbre __restrict__ arbre)***. Cette fonction nécessite un chemin d'accès ainsi qu'un arbre en entrée pour réaliser cette opération.

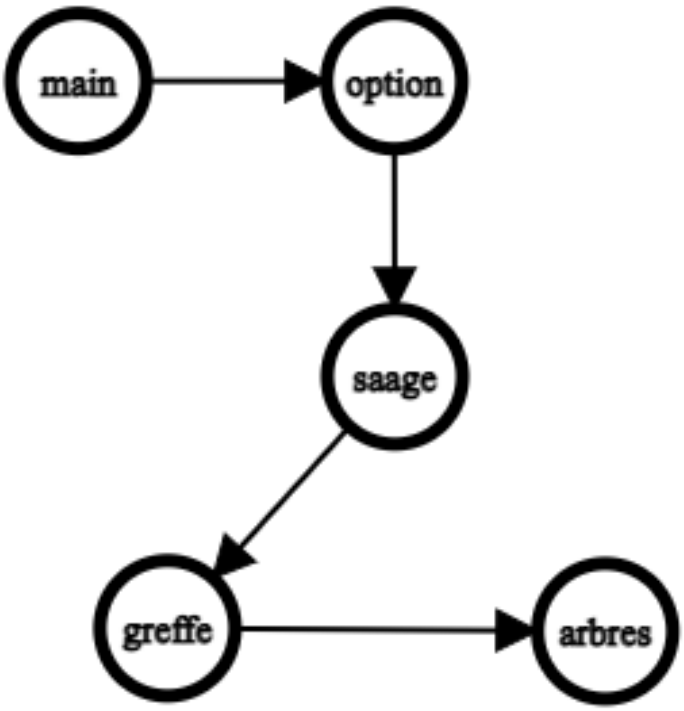
Graph d’inclusion des modules :

Afin de visualiser le graphe d'inclusion, nous avons ajouté une image PNG.

Vous remarquerez qu'aucun cycle n'est présent, chaque module dépend d'un autre module mais cette dépendance n'est pas réciproque.

De plus, nous avons créé un nouveau module appelé 'option' (les fichiers option.h et option.c respectivement).

En regroupant chaque option dans un seul fichier, cela simplifie la lecture des fichiers nécessaires pour comprendre l'origine de chaque fonction et sa fonctionnalité.



Fonctions Intermédiaires

Main et option d’execution :

Notre fonction main parcourt les arguments de la ligne de commande afin d'identifier les options spécifiques :

- Elle débute en vérifiant la présence d'un nombre suffisant d'arguments.
- Ensuite, elle itère à travers chaque argument pour repérer les options **-G, -E, -DOT, -BIG.**
- Une fois qu'une option est identifiée, la fonction correspondante est appelée avec les arguments associés.

Fonctionnement de l'option -G :

- Lecture des chemins des fichiers **s.saage** et **g.saage**.
- Chargement de l'arbre source à partir du fichier **s.saage**.
- Chargement du greffon à partir du fichier **g.saage**.
- Application du greffon à l'arbre source.
- Sérialisation de l'arbre résultant dans un fichier temporaire.
- Affichage du contenu du fichier temporaire sur la sortie standard.
- Suppression du fichier temporaire.
- Libération de la mémoire utilisée par les arbres.

Fonctionnement de l'option -E :

- La fonction **creer_arbre_stdin** lit un entier à partir de stdin.
- Si cet entier est 0, cela signifie la fin de l'arbre, donc l'arbre est défini comme nul (**NULL**).
- Sinon, la fonction lit une chaîne de caractères correspondant au nom du nœud de l'arbre à partir de stdin.
- Elle alloue ensuite un nœud pour l'arbre avec ce nom.
- Enfin, elle crée récursivement les sous-arbres gauche et droit en appelant **creer_arbre_stdin** pour chacun d'eux.

Fonctionnement de l’option -DOT :

- La fonction **ecrire_debut** initialise le fichier DOT avec les paramètres nécessaires pour définir le style des nœuds et des arêtes.
- La fonction **ecrire_arbre** écrit les nœuds et les arêtes de l'arbre dans le fichier DOT en parcourant l'arbre de manière récursive.
- La fonction **ecrire_fin** termine le fichier DOT.
- La fonction **dessine** appelle les fonctions précédentes pour écrire l'arbre dans le fichier DOT.
- La fonction **visualisation_dot** crée le fichier DOT, génère un fichier PDF à partir du DOT à l'aide de la commande **dot**, puis ouvre le fichier PDF avec le visualiseur PDF par défaut.
- Enfin, la fonction **option_DOT_main** est appelée par le **main** pour gérer l'option **-DOT**. Elle charge l'arbre à partir du fichier spécifié, génère la visualisation et libère la mémoire.

Fonctionnement de l'option -BIG :

L'option **-BIG** permet de greffer un arbre existant avec un autre arbre prédéfini (**greffe_grand.saage**) et d'afficher le résultat en format .dot.

- La fonction **greffe_dun_arbre** charge l'arbre initial à partir du chemin spécifié.
- Ensuite, elle charge l'arbre de greffe à partir du chemin prédéfini **exemples/greffe_grand.saage**.
- Elle utilise ensuite la fonction **expansion** pour greffer l'arbre de greffe sur l'arbre initial.
- Si la greffe réussit, l'arbre résultant est sérialisé dans un fichier spécifié par **path_create**.
- Ensuite, la fonction **visualisation_dot** est appelée pour afficher l'arbre résultant en format .dot.
- Enfin, la mémoire allouée pour les deux arbres est libérée.

Améliorations et Optimisations Du Code

- Tout d'abord, il est important de noter que nous avons employé plusieurs options de compilation en C, notamment : ***-ansi -std=c17 -pedantic -Wall -Wfatal-errors -Werror -Wextra -finline-functions -funroll-loops***. Ces choix sont faits dans le souci de respecter les normes et de maintenir un code C cohérent et de haute qualité. Comme optimisation flag, nous avons utilisé ***-O3*.**

- Nous avons créé un Makefile efficace pour simplifier la compilation, la suppression et l'organisation de notre répertoire de projet.

- D'après https://stackoverflow.com/questions/3375697/what-are-the-useful-gcc-flags-for-c, Nous avons ajouté les flags de GCC suivants dans makefile (**CFLAGS_OPTIONALS**): ***-Wfloat-equal -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-align -Wstrict-overflow=5 -Wundef -Waggregate-return -Wcast-qual -Wswitch-default -Wswitch-enum -Wconversion -Wunreachable-code***.

- Nous avons inclus des exemples supplémentaires, notamment les fichiers "grand.saage", "immense.saage" et "colossal.saage" pour illustrer les arbres de grande taille.

- Nous avons enrichi notre code de nombreux commentaires pertinents qui facilitent la compréhension et la maintenance du projet.

- La documentation que nous avons utilisé pour ***restrict*** et ***inline*** est suivant: **https://gcc.gnu.org/onlinedocs/gcc-8.5.0/gcc/Restricted-Pointers.html, https://gcc.gnu.org/onlinedocs/gcc/Inline.html**

- Enfin, tel qu'on suivit standard **-ansi** et le standard de **C99**, nous avons ajouté les mots clés comme ***restrict*** et ***inline***. Nous avons fait **__restrict__** et **__inline__**, car le standard de gcc nous oblige à souligner les mots que le standard **-ansi** ne connaît pas. D'après: **https://gcc.gnu.org/onlinedocs/gcc/Alternate-Keywords.html**.

- Le code est optimisé et évite les calculs superflus pour une performance accrue, de plus il est sécurisé et ne présente aucun risque de fuite mémoire, garantissant ainsi une exécution sans danger.

Répartition Des Tâches

Tout d’abord nous avons travaillé chacun de notre côté sur les fonctions importantes comme expansion(), serialise(), deserialise().

Lorsque ces fonctions ont été terminé et qu’il n’y avait plus de problème de fuite de mémoire, Antony s’est chargé de la création du main ainsi que l’option -G et Maksat s’est chargé de l’option -E et les fonctions supplémentaire pour -DOT et -BIG.

Ensuite Maksat à également amélioré le code en codant certaines fonctions de la bibliothèque string.h.

Antony a été responsable de toute la documentation en rendant le code plus lisible et compréhensible.

En partageant équitablement la charge de travail, nous avons bien travaillé sur ce DM, la répartition des tâches finales est équitablement répartie, avec 50% des responsabilités attribuées à Antony et 50% à Maksat.

Pendant le dm, la communication entre nous était efficace et solide et nous n'avons pas rencontré de difficultés pour nous comprendre et proposer des idées clés afin d'améliorer le code. Nous nous sommes mutuellement aidés et avons apporté notre soutien tout au long du projet dans le but d'apprendre davantage.