Projet 19 : Analyse d'un programme

PANCHALINGAMOORTHY Gajenthran

28 Novembre 2018

Table des matières

1 Introduction **2**

2 Organisation du code **2**

3 Détails du code **2**
3.1 Vérification des expressions : match 2
3.1.1 Reconnaître les fichiers sources dans un Makefile 2
3.1.2 Reconnaître une fonction déclarée 2
3.1.3 Reconnaître un appel de fonction 3
3.2 Matrice compacte 4
3.2.1 Créer la matrice compacte 4
3.2.2 Afficher la matrice compacte 4
3.2.3 Calculer la/les composante(s) connexe(s) de la mco . . 4
3.2.4 Calculer le/les cycles de la mco 5
3.3 Liste de successeurs 5
3.3.1 Créer la liste de successeurs 5
3.3.2 Afficher la liste de successeurs 6
3.3.3 Calculer la/les composante(s) connexe(s) de la lis . . . 6
3.3.4 Calculer le/les cycles de la lis 6
3.4 Lecture/Analyse du programme : adp 6
3.4.1 Lecture du/des fichier(s) 6
3.4.2 Analyse du/des fichier(s) 7

4 Améliorations possibles **7**

1 Introduction

Le projet 19 consiste à réaliser l'analyse d'un programme. Un programme peut être considéré comme un graphe orienté où chaque fonction est un noeud. Prendre un programme important (plusieurs dizaines de fichiers, plusieurs dizaines de milliers de lignes de code) et l'analyser sous la forme d'un graphe (Mco, Lis). On doit chercher les composantes connexes et les cycles (`http://www.ai.univ-paris8.fr/~jj/Cours/Algo/AA18.html`).

2 Organisation du code

Le code est organisé en 4 parties :
- match.c/.h qui s'occupent de reconnaître les différentes fonctions déclarées et les appels de fonctions, et de les placer dans un vecteur de fonctions.
- list.c/.h qui traitent le graphe Lis (liste de successeurs)
- mco.c/.h qui traitent le graphe Mco (matrice compacte)
- adp.c (adp : Analyse D'un Programme) qui est le fichier principale. Il s'occupe d'abord de lire et d'analyser le programme passé en paramètre de la fonction main

3 Détails du code

3.1 Vérification des expressions : match

3.1.1 Reconnaître les fichiers sources dans un Makefile

Pour reconnaître les fichiers sources, il suffit de vérifier si l'expression traitée est un ensemble de lettres, de chiffres et/ou de tirets et qu'il se termine par ".c". Au départ je ne comptais absolument pas traiter le fichier Makefile, je me contentais d'analyser les fichiers .c passés en argument de la fonction `main`. Finalement je me suis dit qu'il était préférable d'analyser le Makefile pour gérer les fichiers .c d'un programme qui ne sont pas dans le même repertoire.

3.1.2 Reconnaître une fonction déclarée

Pour reconnaitre les différentes fonctions déclarées du programme, il faut d'abord identifier la structure du nom des fonctions, puis la structure des listes d'arguments et enfin la structure du bloc des fonctions. Pour reconnaître le nom des fonctions, on appelle la fonction `match_fonction` qui va vérifier l'expression et qui appellera à son tour la fonction `match_args` si

l'expression est correcte. Puis `match_args` vérifiera si l'expression correspond bien à la structure des listes d'arguments pour ensuite appeler `match_bloc` qui va analyser le bloc de la fonction.
La structure du nom des fonctions est assez simple, elle utilisera la fonction `match_alphanum` qui se contentera de reconnaître une expression qui utilisent des lettres de l'alphabet ([a-zA-Z] donc des valeurs entre 65 et 122 dans la table ASCII) ou/et des chiffres ([0-9] donc des valeurs entre 48 et 57) ou/et également des tirets ([_] donc la valeur 95).
Pour vérifier la structure des listes d'arguments, on va donc voir si l'expression commence par une parenthèse ouvrante et se termine par une parenthèse fermante. Il n'y a pas besoin de gérer ce qu'il y a entre les deux parenthèses (les arguments) pour notre projet.
Une fois la structure des listes d'arguments validée, on doit savoir si la structure du bloc de la fonction est correcte. Pour cela, je vérifie si l'expression commence par une accolade ouvrante et se termine par une accolade fermante en appelant la fonction `match_acc`.
Remarque : Pour éviter de compter les accolades fermantes des structures de contrôles (comme la boucle `while` ou les conditions `if`, `else if` et `else`) comme l'accolade fermante du bloc de fonction, on va créer un compteur qui commencera à incrémenter dès qu'il y a une accolade ouvrante et qui décrémentera une fois qu'il verra une accolade fermante.
Enfin, si l'expression est correcte, on enregistre le nom de la fonction dans la structure `fprog_t` qui va contenir toutes les fonctions déclarées.
Pour le passage d'une structure (nom, arguments ou bloc) à l'autre, on doit ignorer certains caractères comme les espaces, les étoiles, les retours à la ligne (`match_whitespace`), les chaînes de caractères (`match_chaine`) et les commentaires (`match_comment`).

3.1.3 Reconnaître un appel de fonction

Pour reconnaître un appel de fonction, on va suivre le même itinéraire sauf que lors de l'analyse du bloc de fonction, on va appeler la fonction `match_callf` cette fois-ci qui va vérifier si l'expression répond au critère de `match_alphanum`. Sauf qu'en faisant cela, nous allons également récupérer les boucles `while` et les conditions `if` par exemple mais également les fonctions de la bibliothèque standard (`printf`, `malloc`...).
Donc, nous allons également vérifier si l'expression qu'on obtient est bien une fonction déclarée dans le programme (`cmp_nchaines`). Par conséquent, on devra d'abord analyser l'intégralité du programme une première fois pour enregistrer toutes les fonctions déclarées puis une seconde fois pour enregistrer les appels de fonctions.

Mais il y a un autre cas à traiter, celui des appels récursifs ou encore des appels multiples à la même fonction. Pour cela, on va vérifier si la fonction a déjà été appelé par la fonction en question (`cmp_callf`).
Comme je l'ai dit précédemment, on va stocker les fonctions déclarées et appelées dans la structure `fprog_t`. La structure `fprog_t` contient le nombre de fonctions déclarées `nfcts` et un vecteur de la structure `fonction_t` correspondant à l'ensemble des fonctions déclarées dans le programme. La structure `fonction_t` contient le nom de la fonction (`nom`) , un vecteur de la structure `fonction_t` qui correspondra ici à l'ensemble des fonctions appelées par la fonction en question et le nombre de fonctions appelées `nbs`.

3.2 Matrice compacte

La matrice compacte comporte le nombre de noeuds, le nombre d'arêtes et un ensemble de cellules de type `cell_t` de taille équivalente au nombre d'arêtes. Les cellules sont des structures comportant l'indice du noeud et l'indice du/des successeur(s) (noeud et successeur(s) liés par une arête).

3.2.1 Créer la matrice compacte

On va d'abord créer la matrice compacte (fonction `creer_mco`) à l'aide du vecteur de type `fprog_t` qui a une taille équivalente au nombre d'arêtes, en enregistrant les indices des noeuds reliant les arêtes. Pour chaque cellule de la matrice, on aura l'indice du noeud et l'indice du successeur.

3.2.2 Afficher la matrice compacte

On peut afficher ensuite le nom des fonctions et de ses successeurs à l'aide de `fprog_t` (fonction `afficher_mco`) . Pour le parcours des successeurs de chaque noeud, j'ai décidé d'utiliser une boucle `do...while` qui parcourt les cellules de la matrice et qui continuera tant que l'indice du noeud `i` de l'arête `k` est identique à l'indique du noeud `i` de l'arête `k+1`.

3.2.3 Calculer la/les composante(s) connexe(s) de la mco

Maintenant, on va pouvoir calculer les composantes connexes du graphe. Pour cela, j'ai utilisé un vecteur `dejavu` qui va vérifier si les noeuds ont déjà été traités en plaçant la valeur de `CC` (initialisée à 1) à l'indice du noeud traité. On parcourt ensuite tous les noeuds en commençant par `main`. Pour chaque noeud, on va également parcourir les successeurs de ce noeud si il n'a pas été visité. A la fin du parcours du noeud, on incrémente `CC`. `chercher_succ_mco` va parcourir les successeurs de chaque noeud de la même façon que lors de la

fonction d'affichage (utiliser une boucle `do...while` qui parcourt les cellules de la matrice et qui continuera tant que l'indice du noeud `i` de l'arête `k` est identique à l'indique du noeud `i` de l'arête `k+1`), tout en remplissant le vecteur `dejavu`. `calculer_cc_mco` va vérifier si il existe bien une fonction `main` dans le programme, si ce n'est pas le cas elle retournera le vecteur `dejavu` avec toutes les valeurs à 0. Sinon elle va appeler `chercher_succ_mco` en prenant l'indice du `main` pour calculer les composantes connexes. Si il existe encore des noeuds non visités dans le vecteur `dejavu`, rappeler la fonction `chercher_succ_mco` si le noeud possède un ou plusieurs successeurs sinon juste se contenter de remplir le vecteur `dejavu` et enfin retourner `dejavu` (cette fois-ci avec aucune valeur à 0 vu que tous les noeuds sont visités).
Pour connaître le nombre de composantes connexes (`compter_cc`), il suffit de parcourir le vecteur `dejavu` et de chercher la plus grande valeur.
Pour afficher les fonctions d'une même composante connexe, il faut récupérer les indices du vecteur `dejavu` ayant la même valeur.

3.2.4 Calculer le/les cycles de la mco

Pour calculer le/les cycles de la matrice compacte, je vais parcourir le graphe en profondeur (`dfs_cycle_mco`) en stockant les noeuds dans une pile. Lors du parcours, si je retrouve un noeud déjà visité (donc déjà présent dans la pile), cela est synonyme de cycle. J'appelle donc `ajouter_cycle_mco` pour pouvoir ajouter le cycle au vecteur `cycles`.

3.3 Liste de successeurs

Le graphe "liste de successeurs" comporte le nombre de noeuds du programme, un vecteur de type `noeud_t` de taille équivalente au nombre de noeuds. Un noeud `noeud_t` possède l'indice de la fonction de la structure `fprog_t` et une liste de successeurs `lis_t`. La structure `lis_t` contient l'adresse d'un noeud et un pointeur vers le successeur suivant.

3.3.1 Créer la liste de successeurs

Pour créer une liste de successeurs, on va d'abord remplir le vecteur de type `noeud_t` à l'aide d'une boucle `for` puis on va ajouter les successeurs de chaque élément du vecteur de type `noeud_t` à l'aide d'une deuxième boucle `for`.

3.3.2 Afficher la liste de successeurs

On peut afficher ensuite le nom des fonctions et de ses successeurs à l'aide de `fprog_t` (fonction `afficher_lis`) . Le parcours des successeurs de chaque noeud sera différent de celui de la matrice compacte. Ici, on va utiliser une boucle `while` qui continuera tant qu'il existe un successeur (tant que le successeur est différent de `NULL`). A chaque tour, on affiche le contenu du successeur et on pointe vers le successeur suivant.

3.3.3 Calculer la/les composante(s) connexe(s) de la lis

Pour calculer la/les composante(s) connexe(s), on va beaucoup se servir de l'algorithme de la matrice compacte. La seule différence concernera la fonction `chercher_succ_lis`, c'est-à-dire le parcours des successeurs. Pour parcourir les successeurs, je vais utiliser la même méthode que pour l'affichage de la liste de successeurs : utiliser une boucle `while` qui continuera tant qu'il existe un successeur (tant que le successeur est différent de `NULL`) et à chaque tour de boucle, on pointe vers le successeur suivant.
Pour connaître le nombre de composantes connexes (`compter_cc`), il suffit de parcourir le vecteur `dejavu` et de chercher la plus grande valeur. La même fonction que la matrice compacte.
Pour afficher les fonctions d'une même composantes connexes, il faut récupérer les indices du vecteur `dejavu` ayant la même valeur. La même fonction que la matrice compacte.

3.3.4 Calculer le/les cycles de la lis

Pour calculer le/les cycles de la liste de successeurs, je vais procéder de la même manière que pour la matrice compacte. Le parcours du graphe concerne la principale différence entre les deux fonctions.

3.4 Lecture/Analyse du programme : adp

3.4.1 Lecture du/des fichier(s)

Avant de lire les différents arguments du `main` qui correspondent à des fichiers, nous allons vérifier si un des arguments comporte un fichier `Makefile` (fonction `contenir_chaine`). Deux cas sont possibles :
- Si il comporte un fichier, on lit le fichier `Makefile` (fonction `lire_makefile`) en le mettant dans une chaîne de caractère puis on analyse le contenu de `Makefile`. L'analyse du `Makefile` (fonction `analyser_makefile`) permettra de récupérer les fichiers sources .c que l'on va stocker dans une liste de

chaînes de caractères. Enfin, on va lire les différents éléments de la liste des chaînes de caractères et donc on ouvre les différents fichiers sources .c qu'on va stocker dans une chaîne de caractère du `Makefile`
- Si il ne comporte pas de fichier, on lit la liste des chaînes de caractères (`argv`) et plus précisément on lit le contenu des fichiers et stocke dans une chaîne de caractère
Mieux vaut conserver le contenu des fichiers dans un tableau car c'est beaucoup plus rapide d'accéder à la mémoire qu'au disque.

3.4.2 Analyse du/des fichier(s)

Une fois que les fichiers sont lus (que ça soit via le `Makefile` ou sans) et stockés dans une chaîne de caractères, on va procéder à l'analyse. L'analyse du programme se divise en 2 parties :
- l'analyse des fonctions déclarées
- l'analyse des appels de fonctions
Comme je l'ai expliqué précédemment, les fonctions déclarées et leurs appels sont stockées dans un vecteur de type `fprog_t`.
Après les analyses, on peut maintenant s'occuper de créer les graphes (matrice compacte, liste de successeurs) et ensuite de libérer la mémoire pour les deux graphes.

4 Améliorations possibles

Beaucoup de points peuvent être ajoutés ou modifiés pour pouvoir améliorer le projet. Voici un tour général de ce qui peut être amélioré :
- Les vérifications des expressions peuvent être améliorés. Je trouve que certaines fonctions possèdent trop d'arguments. Et certains algorithmes pourraient être alléger (par exemple l'appel à de nombreuses reprises à `match_ignore`).
- Rajouter les shaders dans les fichiers sources du `Makefile`
- Trouver un moyen plus simple que de rechercher la chaîne de caractère `main` dans le calcul des composantes connexes
- Ajouter brin, matrice et vecteur de successeurs
- Améliorer l'analyse du fichier `Makefile`. Par exemple, pour l'analyse du programme, je suis contraint de mettre l'exécutable dans le même repertoire que le fichier `Makefile`.
- Des optimisations peuvent être apportés (notamment avec des fonctions qui se ressemblent beaucoup : `ajouter_cycle`)