Algorithmique avancée

Analyse de programme en C
Le 8 Decembre 2016
Hobean BAE N'14500464

1 Introduction

Le but de ce projet est d'anaylser le fichier en C, sous forme de graphe avec la structure de matrice et de Brin.
Aussi il faut comparer l'efficacité de deux structures differentes.

2 Structure de donnée

La matrice de graphe est une forme de matrice comme structure suivante

```
struct graph_mat{
  int nbNode; // nombre de noeud
  int lineMax; // max noeud
  char * listf[MAT_MAX]; // le nom de fonction
  int matrice[MAT_MAX * MAT_MAX]; // matrice
  int declaration[MAT_MAX]; //si le noued 0 est declaré, declaration[0]=1;
  int appel[MAT_MAX];  // le nombre de fois de l'appel de la fonction
};
```

L'avantage est simple à implementer.
L'inconvenient est qu'elle réserve le memoire de N^2
Mais elle n'utilise pas tous les cases.

Et ceci est la deuxieme structure qui s'appelle brin

```
struct strand{
  Shu node; // l'identifiant du noued
  node_t next; // l'indice de successeur suivant
  int poid; // l'orientation et la distance
};
struct strandgraph {
  Shu nbs; // max noeud
  Shu nbstr; // max tableau de brin (nbs*2+1)
  Shu nbnd; // nombre de noeud
  Shu nbbrin; // nombre de noeud associe
  char * listf[GMAXND]; // liste de fonction dans le graphe
  node_t * node; /* first strand*/
  strand * nxt;/* node and next strand*/
  node_t * declaration; //si le noued 0 est declaré, declaration[0]=1;
  node_t * recursive; //si le noued 0 est une fonction recursive, recursive[0]=1;
  node_t * appel; // le nombre de fois de l'appel de la fonction
};
```

Il y a une tableau de structure *strand* qui contient trois variables différentes.
Le variable $node_t * node$ contient l'indice de chaque noeud qui est dans le tableau

de $strand * nxt$.
Si tous les successeurs de graphe pointent réciproquement vers leur parent, on peut supprimer le variable *poid* dans $strand * nxt$.
Le variable *poid* a été ajouté pour determiner l'orientation.

Les variables :
$node_t * declaration$;
$node_t * recursive$;
$node_t * appel$;
permet d'annalyser le fichier '.c'.
$char * listf[GMAXND]$;
est le tableau des fonctions
cela permet de trouver facilement le nom de fonction et de ne pas comparer tous les chaine de caracters d'une fonction pour comaparer les noeuds.

3 Algorithme

J'ai commencé d'abord par créer un parseur qui cherche des fonctions dans le fichier '*.c' La condition pour trouver une fonction est que si la fonction trouve une parenthese ouvrante, on considere comme une fonction, sauf la parenthese est utilisée pour le cast et le mot clé de C

```
if()
while()
return(int)val;
x/(float)y
sizeof(int)*
```

Dans ce cas, le parseur ne les ajoute pas dans le graphe.
Tout d'abord, le parseur doit savoir si le curseur de fichier est stiué dans la declaration de la fonction soit au debut de declaration de la fonction.
ex) foo() est une fonction qui appelle la fonction bar()

```
void foo(){
bar();
}
```

Si on trouve une accolade ouvrante,
on incremente le variable $accolade++$.
si on trouve une accolade fermente,
on decremente le variable $accolade--$.
Donc le variable *accolade* est à 0 et puis le parseur detecte une fonction,
Cela veut dire que le curseur est au debut de la declaration de la fonction.
$PARENT = foo()$

Si le variable accolade est superieur à 0, la fonction que le parseur trouve sera ajoutée au successeur de $PARENT$.
Il y aussi a des variable boolean pour voir si le curseur se situe dans le commentaire ou dans la chaine de caractere, Dans ce cas, il ignore tous les fonctions qu'il detecte.

```
/* fonc()*/
// fonc2()
char * str = {"fonc3()"}
```

La liste des fonctions suivante est l'outil pour anaylser le code C.

```
extern void Brin_printRoot(strgr g);
```

Cette fonction permet de trouver le maximum profondeur parmi tous les fonction, le maximum profondeur est important pour identifier la fonction principale *main*, parce que la fonction principale est la racine de tous les sous fonctions, la racine a toujour le profondeur maximum.

```
extern void Brin_analyseError1(strgr g);
```

Le noeud dans le graphe en brin peut être une fonction qui a été déclarée, pourtant elle n'a pas été appelée. Elle indique les fonctions qui n'ont pas étés appelée.

```
extern void Brin_analyseError2(strgr g);
```

Le noeud dans le graphe en brin peut être une fonction qui a été appelée, pourtant elle n'a pas été déclarée.La fonction verifie si la fonction n'est pas dans la liste de la fonction de la bibliothèque standard.
Elle indique les fonctions qui n'ont pas étés déclarées.

```
extern void Brin_analyseError3(strgr g);
```

La fonction visite tous les fonctions, elle detecte la fonction s'il y a une fonction infinie.

```
void a(){
b();
}
void b(){
c();
}
void c(){
 a();
}
```

```
extern void Brin_analyseError4(strgr g);
```

Cette fonction permet d'indiquer la fonction qui appelle elle-même.

```
int fibo(int n){
if(n<=1)return 1;
return fibo(n-1)+fibo(n-2);
}
```

```
extern void Mat_profondeur(graph_mat gm);
```

Cette fonction trouve la fonction principale, et elle affiche tous les successeurs de cette racine.

```
extern void Mat_connexePoid(graph_mat g);
```

Cette fonction analyse la liste de fonction de chaque connexe.
Et trouver une connexe forte et une connexe faible.
Cette methode est inspirée par le reseaux de neuron.
Elle parcour aleatoirement tous les sens dans son connexe,
quand elle passe un noeud, la fonction additionne une petite valeur à son noeud.
Si on passe souvent un noeud, ca veut dire que ce noeud est relie à plusieurs noeuds, ce noeud se situe au centre connexe,autrement dit c'est une connexe forte. Si on ne passe pas souvent un noeud, c'est un noeud qui est un peu eloigné de centre connexe. Pourtant le graphe avec lequel on travaille est plutôt une forme de l'arbre, le resultat que j'ai obtenu n'est pas très significatif.

4 Conclusion

Quand j'ai testé le programme avec la structure de matrice.
Elle considerait une structure qui utilise beaucoup de memoire.
Espace utilisé : N^2
N est le nombre de noeud.

Par contre, la structure de brin,
Espace utilisé : $6A + N$
6 est (poid + successeur + id noeud) * 2
A est le nombre d'arret
N est nombre de noeud

Si on prends un example.
S'il y a 100 noeud et 300 arrets (10000 arrets possibles)
MAT : 10000 case nécessaire
BRIN : 2100 case nécessaire
Espace utilisé : MAT $>$ BRIN

S'il y a 100 noeud et 5000 arrets (10000 arrets possibles)
MAT : 10000 case nécessaire
BRIN : 30100 case nécessaire
Espace utilisé : MAT $<$ BRIN

S'il y a 100 noeud et 1650 arrets (10000 arrets possibles)
MAT : 10000 case necessaire
BRIN : 10000 case necessaire
Espace utilisé : MAT $==$ BRIN

Par ce resultat,
si le taux de graphe(nbre d'arret associé / nbre d'arret possible * 100) est superieur à 16.5%, il vaut mieux utiliser la structure matrice.

Au point vue de l'efficacité,
Ce programme construie la graphe qui est plutôt une forme d'arbre, sauf la fonction recursive, puisque la fonction principale appele les autres fonctions.

Un noeud qui pointe vers son parent est très rarement apparaît sauf une fonction recursive.
Donc on peut quasiment supprimer le moitie arret possible.

Pour conclure,
La structure de graphe en brin n'est pas toujours efficase par rapport au matrice.
Cela doit attentivement appliquer au problematique différent.
Cepandant, en pratique, le taux de graphe est inferieur à 16.5% (avec un seul fichier 4.8%).
En plus, s'il y a beaucoup de noued, la structure de MAT n'arrive pas réserver son memoire.
Donc il est encore nécessaire de servir la structure Brin pour ce projet.