Algorithme de détection de tous les circuits d'une composante fortement connexe

Résumé

Ce document décrit un algorithme qui permet de lister l'ensemble des circuits d'une composante fortement connexe.

1 Contexte

Le projet F-Safe a pour objectif de détecter la terminaison d'un programme écrit dans un langage spécifique. Cette détection nécessite l'analyse du graphe d'appels de fonction pour en détecter les circuits.

2 Motivation

L'algorithme de Tarjan [1] permet de décomposer un graphe orienté en ses composantes fortement connexes en temps linéaire. Aussi l'utilisera-t-on pour extraire du graphe que les composantes fortement connexes, c'est-à-dire les sous-ensembles de nœuds qui peuvent joindre un autre nœud du même sous-ensemble auquell il appartient par un chemin.

Lorsqu'un chemin revient à son point de départ, on parle alors de circuit. Lorsqu'un circuit ne passe qu'une seule fois par un même nœud, il est alors dit *élémentaire* et dans le cas contraire *composé*.

Nous avons besoin de ces deux types de circuits or nous n'avons pas trouvé d'algorithme adapté dans la littérature d'où la proposition de celui qui suit.

3 Algorithme

L'algorithme proposé se décompose en quatre phases :
- Initialisation
- Recherche des circuits
- Normalisation des circuits
- Élimination des doublons

Il est de type *branch and bound* c'est-à-dire une exploration de toutes les solutions de taille maximale.

Il prend en entrée une composante connexe avec pour chaque nœud, la liste des successeurs directs. Il retourne la liste de tous les circuits possibles, c'est-à-dire des chemins qui passent par plusieurs arcs une seule fois et retourne au nœud de départ. Un même nœud peut être visité plusieurs fois.

1. http ://fr.wikipedia.org/wiki/Algorithme_de_Tarjan

3.1 Initialisation

L'initialisation consiste au comptage des degrés sortant d^- et entrant d^+ de chaque nœud afin d'établir d'une part la disponibilité de chaque nœud et d'autre part le la racine de l'exploration. Cette dernière est importante puisqu'elle conditionne le nombre de listes dupliquées lors de la deuxième phase.

La disponibilité est le minimum des degrés entrant et sortant. En effet un nœud qui a un seul prédécesseur mais plusieurs successeurs ne peut être utilisé qu'une seule fois puisque le circuit ne doit comporter les arcs qu'une seule fois (cf. 4.2). La situation réciproque est équivalente (cf. 4.3).

3.2 Recherche des circuits

La recherche des circuits se base sur exploration exhaustive de tous les chemins et enregistre les éléments intermédiaires.

3.3 Normalisation

Les nœuds avec une disponibilité multiple peuvent engendrer des duplications dans les résultats. Il est donc nécessaire de normaliser ceux-ci, c'est-à-dire opérer une permutation circulaire sur chaque liste afin que le début de la séquence soit minimal dans l'ordre lexicographique.

3.4 Élimination des doublons

Cette dernière phase permet de garantir que la réponse de l'algorithme est minimale.

3.5 Pseudo-code

Entrée : liste des nœuds avec la liste de leurs successeurs
Sortie : liste de tous les circuits

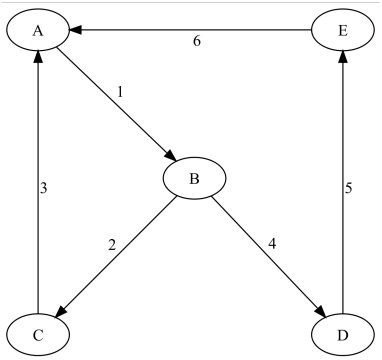
```
1.  pour chaque nœud faire
2.      n.d ← min(d+, d−)
3.      n.u ← 0
4.  fin pour
5.  r ← 1er nœud avec d minimal
6.  n ← r
7.  j ← 0

8.  parcours(n)
9.      k ← j + 1
10.     si n.d <> 0 alors
11.         si n.u <> 0 alors
12.             k ← k + 1
13.             j ← j + 1
14.             borne.j ← n
15.             rang.j ← n.u
16.         fin si
17.         ajouter n à la pile
18.         n.d ← n.d − 1
19.         n.u ← n.u + 1
20.         pour chaque successeur s de n faire
21.             parcours(s)
22.         fin pour
23.         dépiler
24.         n.d ← n.d + 1
25.         n.u ← n.u − 1
26.         pour i allant de k à j faire
27.             si rang.i <> 0 alors
28.                 ajouter n au début de liste.i
29.                 si t = borne.i alors
30.                     rang.i ← rang.i − 1
31.                 fin si
32.             fin si
33.         fin pour
34.     sinon
35.         j ← j + 1
36.         borne.j ← n
37.         rang.j ← n.u
38.     fin si
39. fin parcours

40. pour i allant de 1 à j faire
41.     normaliser(liste.i)
42. fin pour

43. supprimer doublons

44. pour i allant de 1 à j faire
45.     liste ← liste ∪ liste.i
46. fin pour

47. retourner liste
```

4 Exemples

4.1 Graphe à un seul circuit

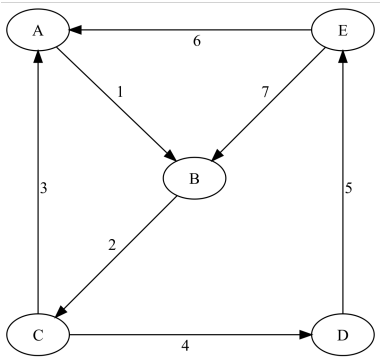
Les graphes à un seul circuit ne posent aucun problème, ils sont donc gérer correctement. Cependant, il est dommage de ne pas pouvoir les détecter sans faire toutes les itérations.

4.2 Graphe dont un nœud a un seul prédécesseur mais deux successeurs



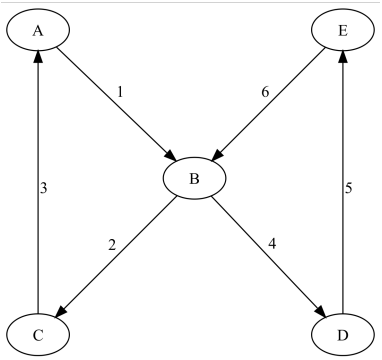
Les circuits de ce graphe sont :
{ A B C } et { A B D E }.
L'algorithme à la fin de la ligne 39 retourne exactement les mêmes circuits.

4.3 Graphe dont un nœud a un seul prédécesseur mais deux successeurs



Les circuits de ce graphe sont :
{ A B C }, { B C D E } et { A B C D E }.
L'algorithme à la fin de la ligne 39 retourne exactement les mêmes circuits.

4.4 Graphe dont un nœud a une *disponibilité* de 2

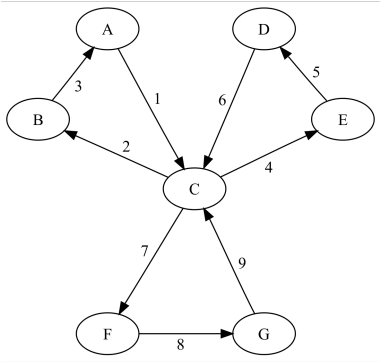


Les circuits de ce graphe sont :
{ A B C } ; { B D E } ; { A B D E B C }.
L'algorithme à la fin de la ligne 39 retourne un circuit de trop :

1. { A B C }
2. { B D E }
3. { A B D E B C }
4. { D E B }

On voit facilement que les 2^{e} et 4^{e} sont identiques à permutation circulaire près. C'est là qu'interviennent d'une part la normalisation qui effectue la permutation circulaire idoine et d'autre part l'élimination des doublons.

4.5 Graphe dont un nœud a une *disponibilité* de 3



Les circuits de ce graphe sont :

1. { A C B }
2. { A C E D C B }
3. { A C E D C F G C B }
4. { A C F G C B }
5. { A C F G C E D C B }
6. { C E D }
7. { C E D C F G }
8. { C F G }

L'algorithme à la fin de la ligne 39 retourne beaucoup plus de circuits.

1. { A C B }
2. { C E D }
3. { A C E D C B }
4. { E D C }
5. { C E D C F G }
6. { A C E D C F G C B }
7. { E D C F G C }
8. { F G C }
9. { C F G }
10. { A C F G C B }
11. { C F G C E D }
12. { A C F G C E D C B }
13. { E D C }
14. { F G C E D C }
15. { F G C }

Une fois normalisés, ils deviennent :

1. { A C B }
2. { C E D }
3. { A C E D C B }
4. { C E D }
5. { C E D C F G }
6. { A C E D C F G C B }
7. { C E D C F G }
8. { C F G }
9. { C F G }
10. { A C F G C B }
11. { C E D C F G }
12. { A C F G C E D C B }
13. { C E D }
14. { C E D C F G }
15. { C F G }

Et donc, après l'élimination des éléments identiques, on a bien la liste complète de tous les circuits.

5 Analyse

L'analyse de cet algorithme est complexe et ne sera pas faite dans les détails.

5.1 Terminaison

L'algorithme termine toujours. En effet, à chaque entrée dans la fonction récursive *parcours*, sa valeur de disponibilité décroît jusqu'à atteindre zéro (ligne 18). L'appel récursif (ligne 21) est dans une boucle qui termine aussi.

5.2 Exactitude

Du fait de l'utilisation d'une méthode *branch and bound*, l'ensemble des solutions est exploré.

5.3 Complexité

La phase 1 est linéaire en la taille des entrées, c'est-à-dire du nombre de nœuds.

La phase 2 est plus complexe à analyser notamment pour la partie de l'exploration. En revanche, la constitution des listes est linéaire en la taille de sortie.

La phase 3 est linéaire en la taille de sortie à condition de choisir une structure de données adaptée.

La phase 4 est quasi-linéaire avec un bon algorithme de tri.