

5

Les tests structurels

- Rappels sur les graphes
 - Graphes de contrôle
 - Analyse statique
 - Analyse dynamique
 - Flots de contrôle
 - Flot de données

Rappels sur les graphes orientés

Définition

Un graphe orienté est fortement connexe si de tout nœud on peut aller à tout autre nœud par au moins un chemin.

Définition

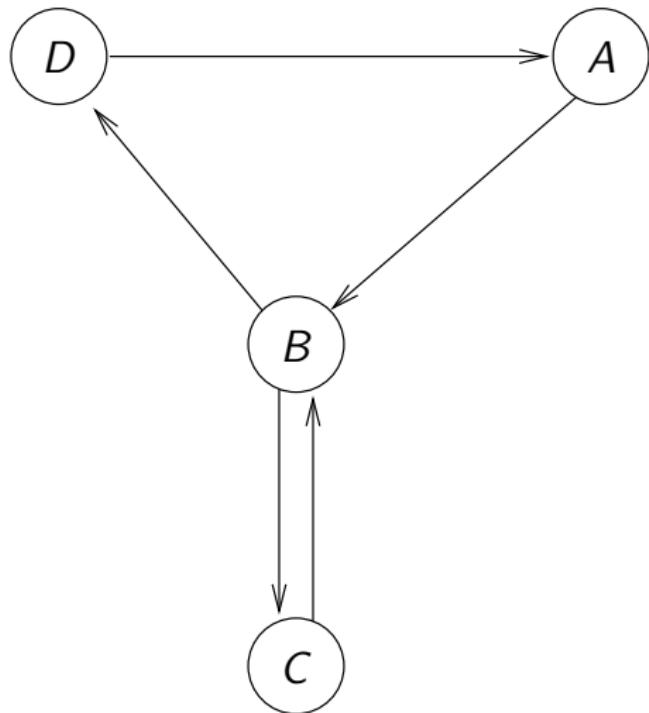
Tout graphe orienté fortement connexe possède au moins un ensemble, dit base de circuits indépendants, dont tout circuit peut être composé (au sens combinaison linéaire).

Exemple

Soit le graphe :

- 4 nœuds : A, B, C, D ;
- 5 arcs : AB, BC, CB, BD, DA .

Exemple



Exemple

circuits	AB	BC	CB	BD	DA
$A - B - D - A$	1	0	0	1	1
$A - B - C - B - D - A$	1	1	1	1	1
$A - B - (C - B)^4 - D - A$	1	4	4	1	1

On a

$$circuit_3 = 4 \times circuit_2 - 3 \times circuit_1.$$

Nombre cyclomatique

Problème

On ne dispose pas d'algorithme pour construire la base.

Proposition

On a

$$\text{Card}(\text{Base}(\mathfrak{G})) = \nu(\mathfrak{G})$$

où $\nu(\mathfrak{G})$ est le nombre cyclomatique du graphe.

Nombre cyclomatique

Soit \mathfrak{G} un graphe orienté fortement connexe de n noeuds et a arcs, on a

$$\nu(\mathfrak{G}) = a - n + 1.$$

Exemple

Le graphe \mathfrak{G} d'arcs AB, BC, CB, BD, DA est fortement connexe.
Son nombre cyclomatique est

$$\nu(\mathfrak{G}) = 5 - 4 + 1 = 2.$$

Exemple

Base

L'ensemble $\{circuit_1, circuit_2\}$ est une base de circuits indépendants.

Preuve

Pour montrer que les circuits sont indépendants, on construit la matrice comptabilisant le nombre de passages par les arcs pour chaque chemin et on exhibe une sous-matrice inversible.

Exemple

circuits	AB	BC	CB	BD	DA
$A - B - D - A$	1	0	0	1	1
$A - B - C - B - D - A$	1	1	1	1	1

Exemple

circuits	AB	BC	CB	BD	DA
$A - B - D - A$	1	0	0	1	1
$A - B - C - B - D - A$	1	1	1	1	1

Exemple

circuits	AB	BC	CB	BD	DA
$A - B - D - A$	1	0	0	1	1
$A - B - C - B - D - A$	1	1	1	1	1

La matrice $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ est inversible car de déterminant non nul :

$$\begin{vmatrix} 1 & 0 \\ 1 & 1 \end{vmatrix} = 1.$$

5

Les tests structurels

- Rappels sur les graphes
- **Graphes de contrôle**
- Analyse statique
- Analyse dynamique
- Flots de contrôle
- Flot de données

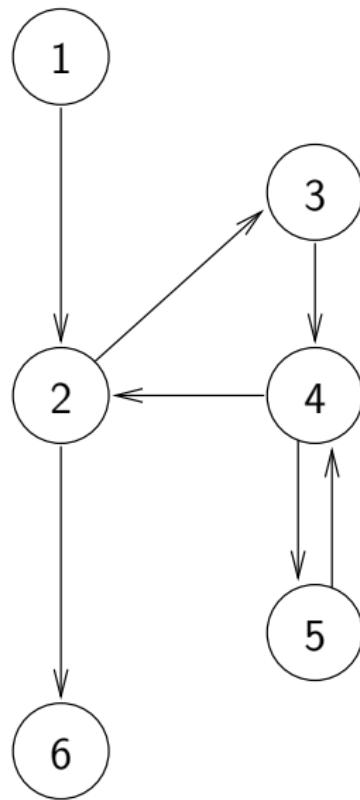
Graphes de contrôle

Définitions

Soit un graphe orienté quelconque.

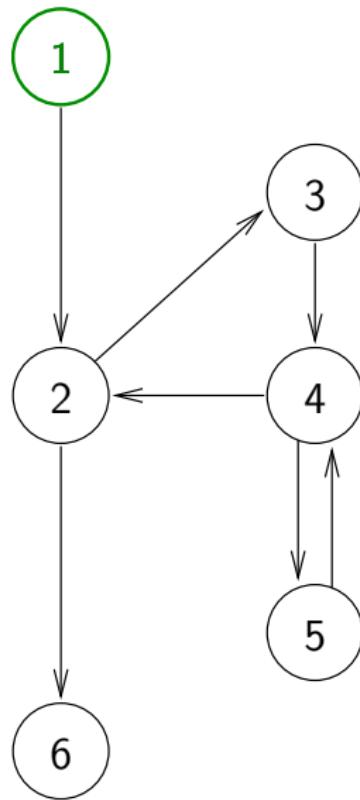
- Un nœud A est une **entrée** si pour tout autre nœud B il existe au moins un chemin de A vers B ;
- De façon similaire, un nœud A est une **sortie** si pour tout autre nœud B il existe au moins un chemin de B vers A ;
- un graphe est dit **de contrôle** s'il a exactement une entrée et une sortie.

Exemple

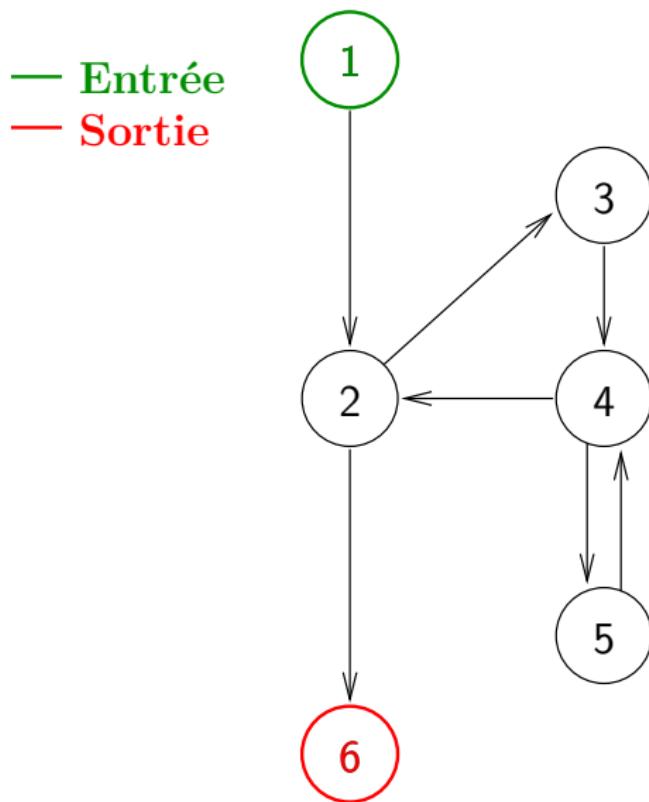


Exemple

— Entrée



Exemple



Graphes de contrôle et code source

Les graphes de contrôles sont liés au code source :

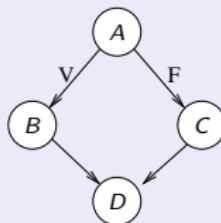
- à chaque instruction correspond à un nœud ;
- lorsqu'il existe la possibilité d'une exécution séquentielle directe entre deux instructions, nous relierons les deux nœuds correspondants par un arc ;
- on ajoute éventuellement un nœud d'entrée fictif et/ou un nœud de sortie fictif (afin d'assurer l'unicité de ceux-ci).

Graphes de contrôle et code source

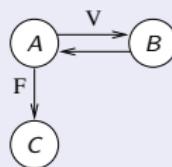
- Séquence : $A ; B$



- Sélection : si A alors B sinon C



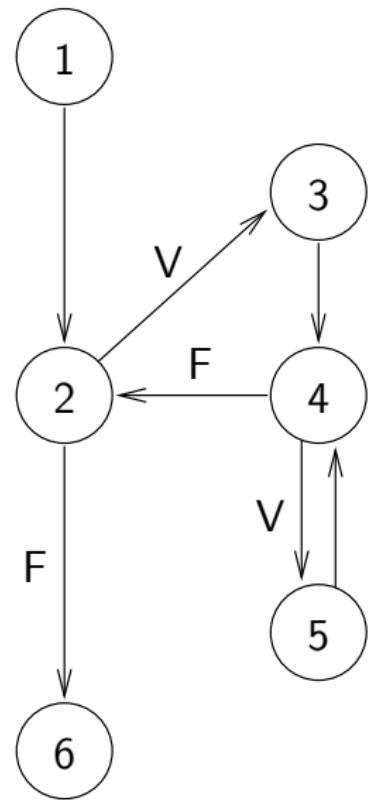
- Boucle : tant que A faire B fin



Exemple

```
i=0; j=0;  
while (i<n) {  
    i++;  
    while (j<m)  
        j++, t[i,j]=0;  
}  
return 0;
```

Exemple



Simplification

Simplification

- on peut remplacer tout chemin linéaire par un nœud sans altérer le nombre cyclomatique ;
- on peut remplacer chaque sous-graphe de contrôle par un seul sommet.

Nombre de McCabe

Graphe de contrôle et graphe fortement connexe

Si on ajoute à un graphe de contrôle un arc de la sortie à l'entrée alors on obtient un graphe fortement connexe.

Définition

On appelle **nombre de McCabe** du graphe de contrôle, le nombre cyclomatique du graphe résultant.

Mesure

Le nombre de McCabe d'un graphe de contrôle donne une estimation de la complexité du code associé.

Tests simples

On appelle décision multiple une sélection du type :

```
if a<2 and a=b then ...
```

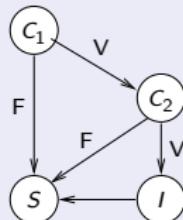
Elle peut-être transformée en deux sélections simples :

```
if a<2 then  
  if a=b then ...
```

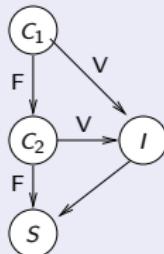
Tests simples

Plus généralement, nous ne ferons apparaître dans les graphes de contrôle que des tests simples. Aussi, lorsque les expressions seront évaluées de façon paresseuse, on aura :

- **if C_1 and C_2 then /**



- **if C_1 or C_2 then /**



Tests simples

Soit un code ne possédant que des tests simples. Alors le graphe de contrôle a un nombre de McCabe égal au nombre de tests simples +1.

Flots de contrôles

Proposition

A tout chemin d'exécution correspond un cycle dans le graphe fortement connexe déduit.

Corollaire

Le nombre de McCabe compte le nombre de chemins d'exécution indépendants. Tout autre chemin d'exécution est une « combinaison » de ces chemins de base.

Remarques

- Un code sans boucle comporte un nombre fini de chemins d'exécution ;
- un code sans boucle ni test comporte un seul chemin d'exécution.

5

Les tests structurels

- Rappels sur les graphes
- Graphes de contrôle
- **Analyse statique**
- Analyse dynamique
- Flots de contrôle
- Flot de données

Analyse statique

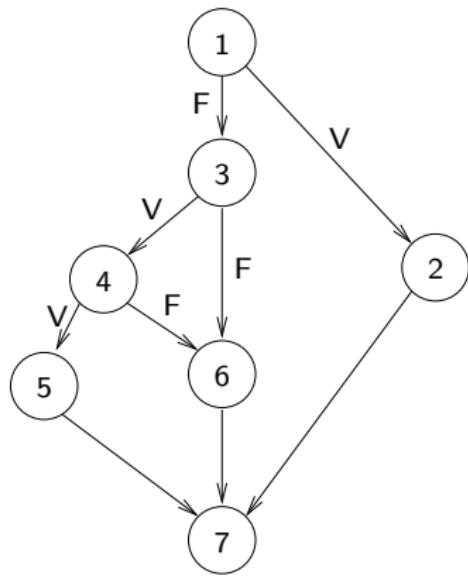
Principe

Le graphe de contrôle peut permettre de trouver des chemins non exécutables. Il peut ainsi être utilisé lors des tests statiques.

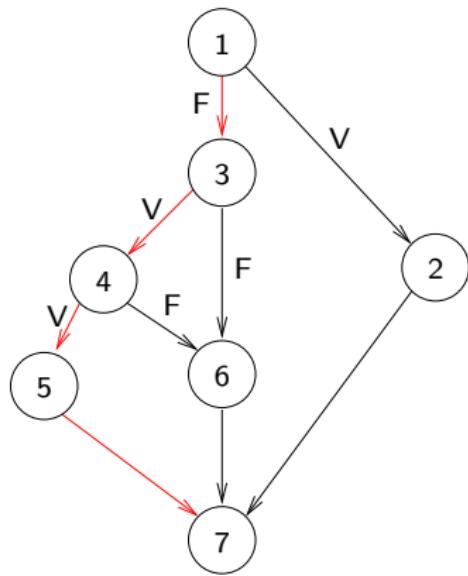
Exemple

```
if (x>0)
    instruction1;
else if (y>0 && x>1)
    instruction2;
else
    instruction3;
```

Exemple



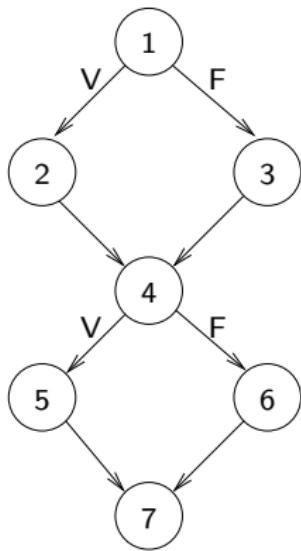
Exemple



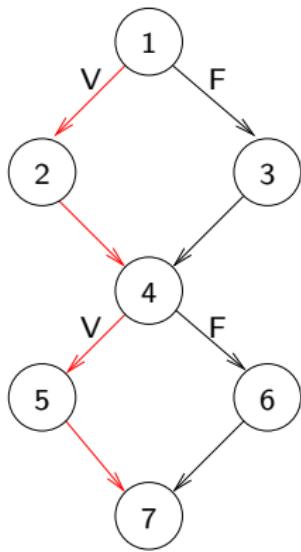
Exemple

```
if (x<=0)
    x=-x;
else
    x=x-1;
if (x== -1)
    y=0;
else
    y=x-1;
```

Exemple



Exemple



5

Les tests structurels

- Rappels sur les graphes
- Graphes de contrôle
- Analyse statique
- Analyse dynamique**
- Flots de contrôle
- Flot de données

Analyse dynamique

Principe

L'analyse structurelle et dynamique sélectionne selon tel ou tel critère un ensemble de chemins exécutables dans le graphe de contrôle.

Test structurel

Le test structurel dynamique associé fixe un ensemble de données de tests permettant de couvrir tous les chemins sélectionnés et l'oracle énonce le résultat attendu pour chaque donnée.

Test tous les chemins

Principe

Dans certains cas (pas de boucles) le graphe ne comporte qu'un nombre fini de chemins d'exécution. On cherche alors à les couvrir tous.

Exemple

```
int max(int x, int y) {  
    if (x<y)  
        return x;  
    return y;  
}
```

Il n'y a que deux chemins qui sont couverts, par exemple, par $(0, 1)$ et $(1, 0)$ (le résultat attendu est alors 1).

Limite du test structurel

Dans le cas particulier où il n'y a qu'un seul chemin d'exécutable, le test structurel dynamique est sans objet puisqu'alors toutes les données se valent.

Que penser alors du code suivant :

```
/* échange les valeurs de x et y */  
x = x * y;  
y = x / y;  
x = x / y;
```

Limite du test structurel

- Une couverture de 100% du code ne veut pas dire test exhaustif !
- N'implique donc pas le zéro défaut.
- Pas très adapté au formalisme objet.

Déterminer les données de test

Trouver des données de test qui sensibilisent un chemin exécutable est une tâche qui peut être difficile. Elle est complexifiée par l'existence de chemins non exécutables.

Exemple

```
while ( f( a , b ) )
    if ( g( a , b ) )
        h( a , b );
    else
        k( a , b );
```

On veut passer trois fois dans la boucle avec $g(a, b)$ alternativement vrai et faux. Pour y parvenir - si c'est possible - il faudra juger de l'impact des différentes valeurs des paramètres a et b sur f , g , h et k qui en même temps modifient probablement ces paramètres.

On cherche à automatiser la génération de données par des méthodes heuristiques : on peut en effet montrer qu'aucun algorithme général de génération ne peut exister.

Chemins non exécutables

L'existence de chemins non exécutables peut être un signe de mauvaise programmation mais pas toujours :

```
s:=0;  
for i:=0 to 1000 do  
    s:=s+a[ i ];
```

possède virtuellement une infinité de chemins dont probablement 1 seul est exécutable, à moins de provoquer la levée d'une exception dans l'instruction de boucle.

Relation d'ordre

Définition

On dit qu'un test T_1 est **plus fort** que T_2 et on écrit $T_1 \geq T_2$ si lorsque T_1 est satisfait, T_2 l'est aussi (les chemins de T_1 couvrent les chemins de T_2).

La couverture des chemins d'exécution peut être plus ou moins importante. Pour évaluer la couverture, on se base sur le **flux de contrôle** ou sur le **flux des données**.

5

Les tests structurels

- Rappels sur les graphes
- Graphes de contrôle
- Analyse statique
- Analyse dynamique
- **Flots de contrôle**
- Flot de données

Couvertures basées sur le flot de contrôle

Flot de contrôle

On se base sur le séquencement des instructions pour la sélection des D.T.

- Couverture de tous les nœuds
- Couverture de tous les arcs
- Couverture de tous les chemins indépendants
- Couverture de tous les PLCS
- ...

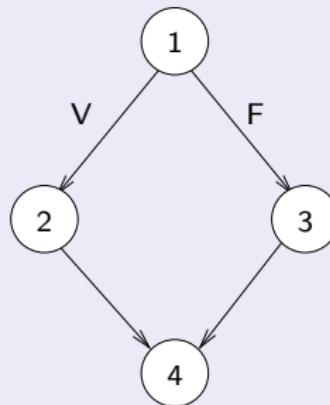
Couverture de tous les nœuds

Exemple

```
function somme(x, y : integer) : integer;
begin
  if x = -11 then somme := -3
  else somme := x + y;
end;
```

Le code n'est faux que pour $x = -11$ et $y \neq 8$. Un test fonctionnel a peu de chances de révéler une telle erreur.

Graphe de contrôle



Critère

La plupart des D.T. ne couvrent que le chemin $1 - 3 - 4$ et n'exécutent pas l'instruction 2.

TER1

Critère

Un critère tous les nœuds cherche à sensibiliser assez de chemins pour passer par tous les nœuds du graphe. Dans le cas de tests fonctionnels il faut s'attendre à ce que

$$\frac{\text{nb. nœuds couverts}}{\text{nb. nœuds total}} = \frac{3}{4}.$$

TER1

Nous noterons *TER1* (Test Effectiveness Ratio) le quotient

$$TER1 = \frac{\text{nb. nœuds couverts}}{\text{nb. nœuds total}}.$$

Nous dirons qu'un test satisfait *TER1* si $TER1 = 1$.

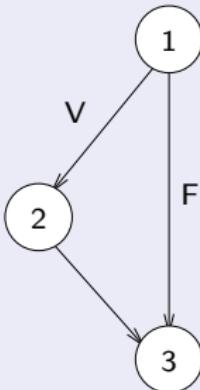
TER1

Un test qui satisfait *TER1* exécute au moins une fois chaque partie (instruction, test) exécutable du programme.

Couverture de tous les arcs

```
function inverse(x : real);  
begin  
  if x <> 0 then x := 1;  
  y := 1 / X;  
end;
```

Graphe de contrôle



TER2

Le test avec $DT = \{x = 1\}$ satisfait TER1. Pourtant il ne décèle pas l'erreur.

Ceci est du au fait que bien que tous les nœuds soient couverts, tous les arcs ne le sont pas.

$$\frac{\text{nb. arcs couverts}}{\text{nb. arcs total}} = \frac{3}{4}.$$

TER2

TER2

Nous noterons *TER2* le quotient

$$TER2 = \frac{\text{nb. arcs couverts}}{\text{nb. arcs total}}.$$

Nous dirons qu'un test satisfait *TER2* si $TER2 = 1$.

Nous dirons qu'un test satisfait *TER2* si $TER2 = 1$. Cela signifie que tous les arcs sont couverts et donc que

- toutes les décisions (tests simples) ont été couvertes
- toutes les branches ont été couvertes

TER2

TER2

Ce critère est plus fort que *TER1*.

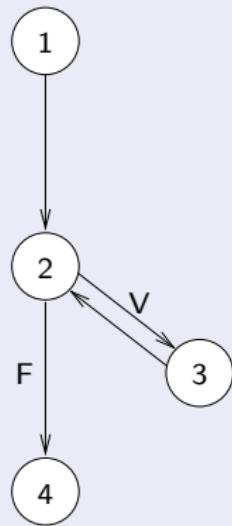
Le problème des boucles

```
function inverse_somme(a : Tableau ;
    inf , sup : integer );
var
    i , somme : integer ;

begin
    i := inf ;
    somme := 0 ;
    while i <= sup do begin
        somme := somme + a[ i ] ;
        i := i + 1 ;
    end ;
    inverse_somme := 1 / somme ;
end ;
```

Le problème des boucles

Graphe de contrôle



Le problème des boucles

D.T.

La donnée $DT1 = \{a = \{1, 2, 3\}, inf = 1, sup = 3\}$ couvre tous les arcs. Cependant, elle ne révèle pas l'erreur qui se manifeste lorsque $inf > sup$. Il s'agit d'un problème typique de boucle : une boucle peut être parcourue 0, 1 jusqu'à virtuellement un nombre infini de fois. L'erreur aurait été trouvée avec $DT2 = \{inf = 2, sup = 1\}$.

Le problème des boucles

Appliquer le principe aux limites

Le principe des tests aux limites nous incite à essayer 0, 1 et « plusieurs fois ». Le cas échéant plusieurs peut être remplacé par $\max - 1$ et \max fois. Si cela a un sens, on essaiera aussi -1 et $\max + 1$ fois (données non valides).

Dans notre cas l'erreur se révèle avec 0 fois. Il s'agit d'une erreur fréquente du type « liste vide ».

Couverture tous les chemins indépendants

Le principe “aux limites” est efficace dans beaucoup de cas, mais dans certains cas, on ne sait comment le combiner avec les autres, car en lui même il n'est pas comparable à “tous les sommets” “ni tous les arcs”.

Par exemple, que penser de :

```
if A then ...
else ...
i := 1;
while B do begin
  if C then ...
  ...
end;
...
```

Tous les chemins indépendants

Définition

On dit que le critère “tous les chemins indépendants” est satisfait si les données de test permet de couvrir tous les chemins d’une base du graphe de contrôle.

On montre que ce critère est plus fort que “tous les arcs”.

Tous les chemins indépendants

Ce critère est bien un critère qui exerce les boucles de “diverses façons”. Sur l'exemple précédent :

- $\{DT1\}$ ne forme pas une base mais $\{DT1, DT2\}$ oui
- Par contre $\{DT1, DT3\}$ avec
 $DT3 = \{a = \{1, 2\}, inf = 1, sup = 2\}$ est également une base mais ne révèle pas l'erreur.

Tous les chemins indépendants

Tout se passe comme si ce critère prétendait que “puisque tout chemin est une combinaison linéaire des chemins $DT1$ et $DT3$, tout ce qui peut se passer dans la boucle peut-être détecté avec $DT1$ et $DT3$.”

Même si

$$1 - 2 - 4 = 3 * DT3 - 2 * DT1,$$

utiliser $DT1$ et $DT3$ séparément n'a pas le même effet que d'utiliser $DT2$ seul.

Couverture des PLCS

On cherche donc d'autres critères permettant d'exercer les boucles.

Les critères PLCS partent de la constatation suivante :

- « tous les sommets » permet d'exercer toutes les expressions de décision et tous les enchaînements formés d'une instruction ;
- « tous les arcs » permet d'exercer toutes les expressions de décision et tous les enchaînements séquentiels formés de deux instructions.

On généralise ce principe.

Couverture des PLCS

Catégorie

On divise les nœuds en deux catégories :

- l'entrée, la sortie et les extrémités de branchement forment la première catégorie (se voit plus clairement en remplaçant le code structuré par des GOTO) ;
- les autres nœuds forment la seconde catégorie.

Définition

On appelle **PLCS** (portion linéaire de code avec saut) un chemin partant d'un nœud du premier type, arrivant à un nœud du premier type et où l'avant-dernier et le dernier nœud constituent le saut du chemin.

Couverture des PLCS

Définition

On pose

$$TER3 = \frac{\text{PLCS couvertes}}{\text{PLCS totales}}.$$

Résultat

On a

$$TER3 = 1 \implies TER2 = 1 \implies TER1 = 1.$$

Couverture des PLCS

Généralisation

Pour tout $n \geq 4$, on pose

$$TERn = \frac{\text{chemins couverts composés d'au plus } n-2 \text{ PLCS}}{\text{total des chemins composés d'au plus } n-2 \text{ PLCS}}.$$

Résultat

On a

$$\forall n \geq 1, \quad TER_{n+1} = 1 \implies TER_n = 1.$$

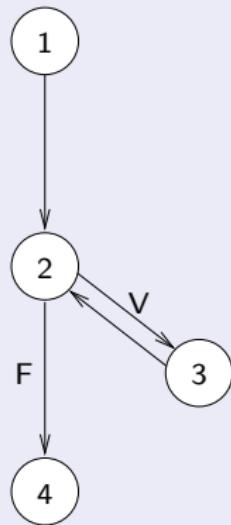
Exemple

```
function inverse_somme(a : Tableau ;
    inf , sup : integer );
var
    i , somme : integer ;

begin
    i := inf ;
    somme := 0 ;
    while i <= sup do begin
        somme := somme + a[ i ] ;
        i := i + 1 ;
    end ;
    inverse_somme := 1 / somme ;
end ;
```

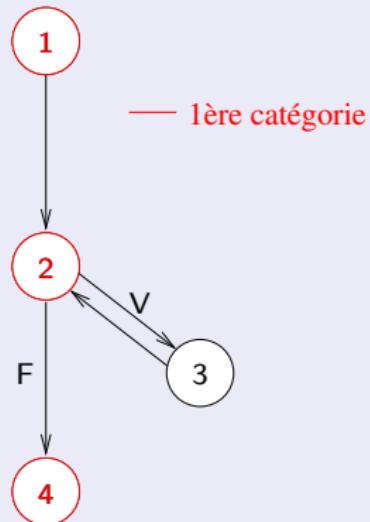
Le graphe de contrôle

Graphe de contrôle



Le graphe de contrôle

Graphe de contrôle



Exemple

PLCS

Les PLCS sont : 124, 1232, 232, 24.

PLCS

Les enchaînements d'au plus 1 PLCS (resp. 2, 3) sont :

- 124, 1232, 232, 24
- en plus : 123232, 12324, 23232, 2324
- en plus : 12323232, 1232324, 2323232, 232324

Exemple

Chemins

Les chemins exécutables sont donc :

- 124 et 12324
- en plus : 1232324
- en plus : 123232324

D.T.

Les données de test sont ...

5

Les tests structurels

- Rappels sur les graphes
- Graphes de contrôle
- Analyse statique
- Analyse dynamique
- Flots de contrôle
- **Flot de données**

Flots de données

Couvertures basées sur le flot de données

Ces critères se fondent sur le séquencement des différentes manières d'utiliser les données pour sélectionner les D.T.

Les données

Utilisation des données

On définit trois catégories d'utilisation des données :

- Définition (ou écriture sur la variable)

`x := ... ;`

`read(x);`

- Référence (ou utilisation : lecture - inspection - de sa valeur)

- P-utilisation (P=prédicat) dans un test

`if ... x ... then ...`

`while ... x ... do ...`

- C-utilisation (C=calcul) autrement

`... := x;`

`write(x);`

`... T[x] ...`

`...`

Définition

On dit qu'une instruction J est utilisatrice d'une instruction I (par rapport à une variable x) s'il existe un chemin C de I à J où la variable x est définie en I et utilisée en J sans qu'il ait eu redéfinition de x sur le chemin C.

Utilisatrice

On distingue les instructions C-utilisatrices des instructions P-utilisatrices. On parlera de nœud C-utilisateur, d'arc P-utilisateur et de chemin C ou P-utilisateur.

Exemple

```
lire(x); {1}
a := x + 2; {2}
if a = 3 {3} then a := a + 1; {4}
else y := x + a; {5}
```

- x : déf 1, C-ut 2, C-ut 5
- a : déf 2, P-ut 3, C-ut 4, def 4, C-ut 5
- y : def 5

Critères

Tous les P-utilisateurs

Nécessite que, pour chaque définition, tous les arcs P-utilisateurs de celle-ci soient couverts par au moins un chemin de P-utilisation. Ce critère implique que tous les arcs sont couverts.

Tous les utilisateurs

Nécessite que, pour chaque définition, tous les arcs P-utilisateurs et les nœuds C-utilisateurs de celle-ci soient couverts par au moins un chemin d'utilisation. Ce critère est plus fort que « tous les P-utilisateurs ».

Critères

Tous les P-utilisateurs, quelques C-utilisateurs

Critère intermédiaire, impose que pour chaque définition ayant un arc P-utilisateur, tous les arcs P-utilisateurs de celle-ci soient couverts par au moins un chemin de P-utilisation et que pour chaque définition ayant un nœud C-utilisateur, au moins une telle C-utilisation soit couverte par au moins un chemin.

Les définitions ci-dessus exigent qu'une définition et une utilisation soient reliées par au moins un chemin. Un critère plus fort encore (donc plus fort que « tous les utilisateurs ») est :

Tous les DU-chemins

Impose que pour chaque définition soit réalisé la couverture de **tous** les chemins P et C-utilisateurs.

Critères

On peut montrer que “tous les DU-chemins” est plus fort que “tous les chemins d’une base”. Cependant ce critère peut exiger la couverture d’une infinité de chemins lors de la présence de boucles.

Exemple

```
read(x, y, z); {1}
if pair(x) then {2}
  read(y, z, u) {3}
else
  read(u); {4}
if x < 0 then {5}
  write(z) {6}
else
  write(y, z); {7}
write(u); {8}
```

Exemple

Données

- x : déf. 1, P-ut. 2, P-ut. 5
- y : déf. 1, déf. 3, C-ut. 7
- z : déf. 1, déf. 3, C-ut. 6, C-ut. 7
- u : déf. 3, déf. 4, C-ut. 8

Exemple

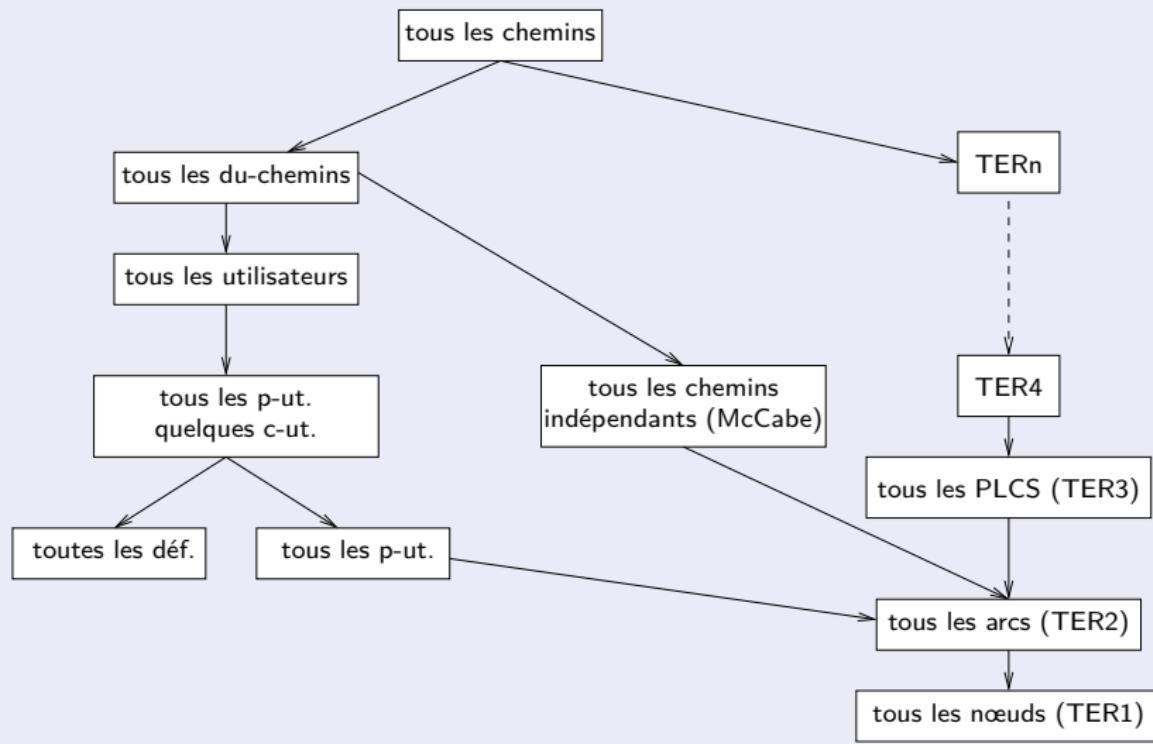
- 123568 et 124578 : tous les P-utilisateurs, mais pas tous les utilisateurs car pas d'utilisation de la déf. de y en 3.
- 123578 et 124568 : tous les P-utilisateurs et quelques C-utilisateurs, mais pas tous les utilisateurs car pas de du-chemin pour z allant de 3 à 6.
- 123578, 124568 et 123568 : tous les utilisateurs, mais pas tous DU-chemins car pour la du-utilisation de z $1 \rightarrow 7$, le chemin 12457 n'est pas parcouru.
- 124578, 123578, 124568 et 123568 : tous les du-chemins.

Exemple

On peut constater que les deux derniers critères contiennent des bases de chemins indépendants. Ce n'était pas obligatoire pour « tous les utilisateurs ». En effet, si on retire la définition de z en 3 alors 123578 et 124568 couvrent tous les utilisateurs sans couvrir une base.

Évaluation

Graphe de force



Analyse des domaines finis

On regroupe les données selon des classes d'équivalences.

Deux données appartiennent à la même classe si elles sensibilisent le même chemin dans le graphe de contrôle.

Les classes sont représentées comme des polygones délimités par les droites ou plans correspondants aux égalités ou inégalités entre les variables.

Analyse des domaines finis

Données de test

On sélectionne des données dans chaque domaine, mais aussi précisément sur les frontières et près de celles-ci (conditions limites).

Cette technique rappelle donc celles de l'analyse partitionnelle et l'analyse fonctionnelle aux limites et les tests associés. Seulement, ici, au lieu de se baser sur la spécification, on se base sur le code source pour trouver les classes et les conditions limites.

Exemple

```
read(x, y);
if x > y then
  if y < a then ...
  ...
```

Donne lieu à trois zones :

- $x < y$
- $x < y$ et $y < a$
- $x < y$ et $y > a$

Exemple

