TD N°2: Sur les techniques de test structurel

Christoph Samuel – Jankowiak Matthias

I-PROBLÉMATIQUE :

1- La série de TD n°2 a pour but de mettre en œuvre dans la pratique certaines techniques de test structurel introduites en cours, en abordant dans un premier temps l'aspect statique (méthode de McCabe) puis dynamique, mais également de nous entraîner à pratiquer et correctement utiliser ces techniques suivant des cas spécifiques. Dans le cadre de l'activité de test de logiciel, ce TD prend place dans la partie cruciale de l'ingénieur de test. Entre autre, le choix des domaines afin de générer les bons jeux de DT et, ainsi, tester le programme le plus efficacement et le plus pertinemment possible.

2- Afin de réaliser un test structurel efficace, il faut suivre une approche particulière, c'est-à-dire, réaliser un graphe de contrôle via généralement un code source, établir les expressions relatives à ce graphe pour calculer le nombre cyclomatique avec la méthode de McCabe dans le but de de prendre connaissance du nombre de chemins indépendants ainsi que le nombre de décisions simples ou principales. Et pour finir, générer les DT afin de tester les trois approches « Tous-les-nœuds, Tous-les-Arcs, Tous-les-Chemins ».

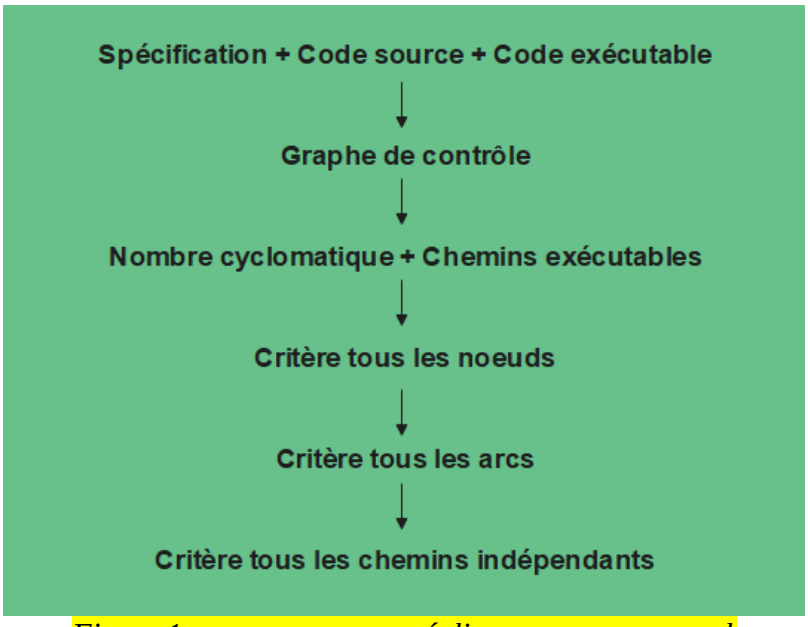


Figure 1 : processus pour réaliser un test structurel

Plus précisément, on distingue l'approche statique, qui consiste à de la revue de code (lecture et analyse du code source), de l'estimation de la complexité (le nombre de défauts est lié à la complexité du code) et de l'analyse du flot de données (des variables). Cette approche a une interprétation très abstraite. Enfin, l'approche dynamique, elle, consiste à produire des DT (jeux de tests) à partir du code source du logiciel, s'appuyer sur le code exécutable pour produire des résultats et finalement, analyser ces résultats en les comparant avec ceux prévus (exécution de programme sur un jeu de DT généré selon la technique de couverture).

II-RÉALISATION :

Pour cet exercice, nous allons nous baser sur un pseudo code qui est censé représenter un composant cherchant l'indice d'un élément (s'il existe) dans une liste triée.

Nous allons dans un premier temps décrire le graphe de contrôle correspondant au pseudo-code, puis calculer le nombre cyclomatique et donner sa signification. Ensuite nous allons établir l'expression des chemins de contrôle, établir le tableau des PI() expressions (ou dr-chaînes) correspondant aux différentes variables pour ensuite dresser le tableau qui met en évidence d'éventuelles anomalies dans le flot de données.

Enfin, en utilisant les résultats qui précèdent, nous proposerons une mise au point de la procédure sous la forme d'un code C, C++ ou Java pour obtenir le composant logiciel sous test et pour finir, on générera un jeu de DT afin de tester les trois approches « Tous-les-nœuds, Tous-les-Arcs, Tous-les-Chemins ».

Étape 1 : Décrire le graphe de contrôle correspondant :

Un graphe de contrôle comporte un seul nœud à partir duquel on peut atteindre tous les autres : c'est l'entrée (a). Il comporte également un seul nœud atteignable à partir de tous les autres : c'est la sortie (m ou n dans notre cas). Un nœud représente un bloc d'instructions. Un arc formalise un transfert de l'exécution d'un bloc d'instructions à un autre. Un chemin formalise une séquence d'exécution qui révèle un comportement du logiciel.

Voici tout d'abord le pseudo code de l'exercice, pour lequel nous avons attribué une lettre et une couleur pour chaque étape afin de le rendre plus lisible et plus compréhensible :

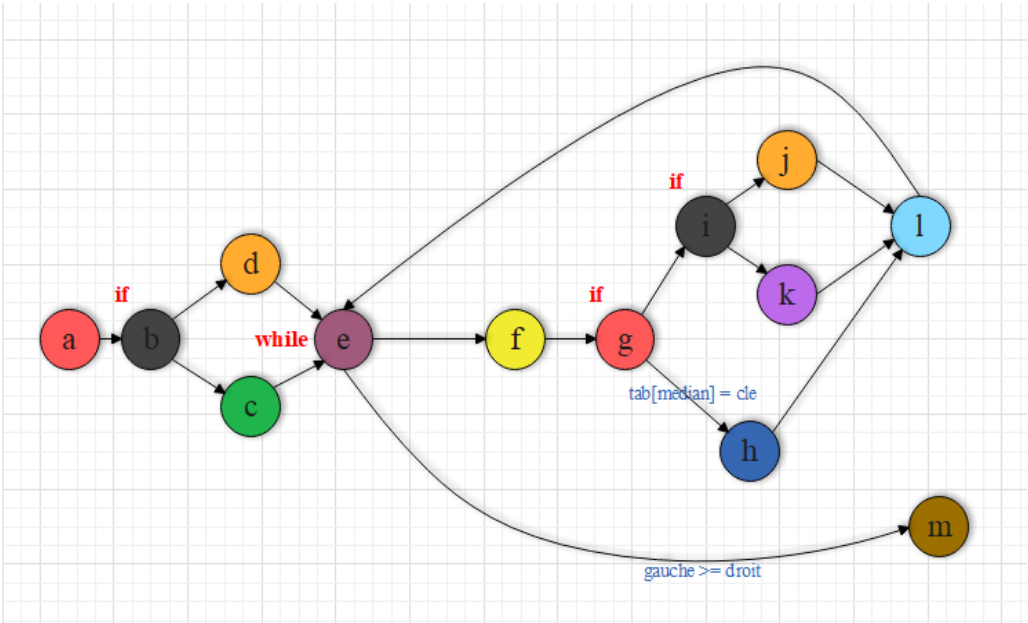
```
look (ELEMENT cle, ELEMENT tab[ ], integer taille, boolean trouve, integer A)
        begin
        integer droit, gauche, median, inf, sup ;
        boolean inc;
        gauche :=inf;
        droit :=sup;
        A :=(droit + gauche)/ 2 ;
        trouve:= inc;                                                    a

        if tab[A] = cle                                                  b
                then trouve :=true;                                      c
                else trouve :=false;                                     d

        while gauche <= droit and trouve = false                         e
                begin
                median:=(droit + gauche) / 2;                            f
                if tab[median] = cle                                     g
                        then
                                begin
                                trouve:=true;
                                A:=median
                                end                                      h
                        else
                                if tab[median]> cle                      i
                                        then gauche:=median+1;           j
                                        else droit :=median-1;           k
                end
end look
```

Figure 2 : pseudo-code boucle WHILE double

Nous allons ensuite en déduire un graphe n°1 de contrôle correspondant au pseudo code avec un graphe contrôle WHILE double.



Graphe 1 : graphe de contrôle WHILE double

Dans ce graphe nous pouvons remarquer la présence de 3 IF ainsi qu'un double WHILE. L'entrée du graphe n°1 est le nœud à et sa sortie est le nœud. La boucle double WHILE est une boucle très compliquée à gérer et peut inclure différentes erreurs, pour cela, on peut changer légèrement le pseudo code à partir de la condition en remplaçant ce double WHILE par un WHILE simple suivi d'une condition IF, le pseudo code devient alors :

```
look (ELEMENT cle, ELEMENT tab[ ], integer taille, boolean trouve, integer A)
        begin
        integer droit, gauche, median, inf, sup ;
        boolean inc;
        gauche :=inf;
        droit :=sup;
        A :=(droit + gauche)/ 2 ;
        trouve:= inc;                                                    a

        if tab[A] = cle                                                  b
                then trouve :=true;                                      c
                else trouve :=false;                                     d

        while gauche <= droit                                            e
                if trouve = false                                        f
                begin
                median:=(droit + gauche) / 2;                            g
                if tab[median] = cle                                     h
                        then
                                begin
                                trouve:=true;
                                A:=median
                                end                                      i
                        else
                                if tab[median]> cle                      j
                                        then gauche:=median+1;           k
                                        else droit :=median-1;           l
                end
end look
```

Figure 3 : pseudo-code boucle WHILE simple

```
while gauche <= droit and trouve = false
```

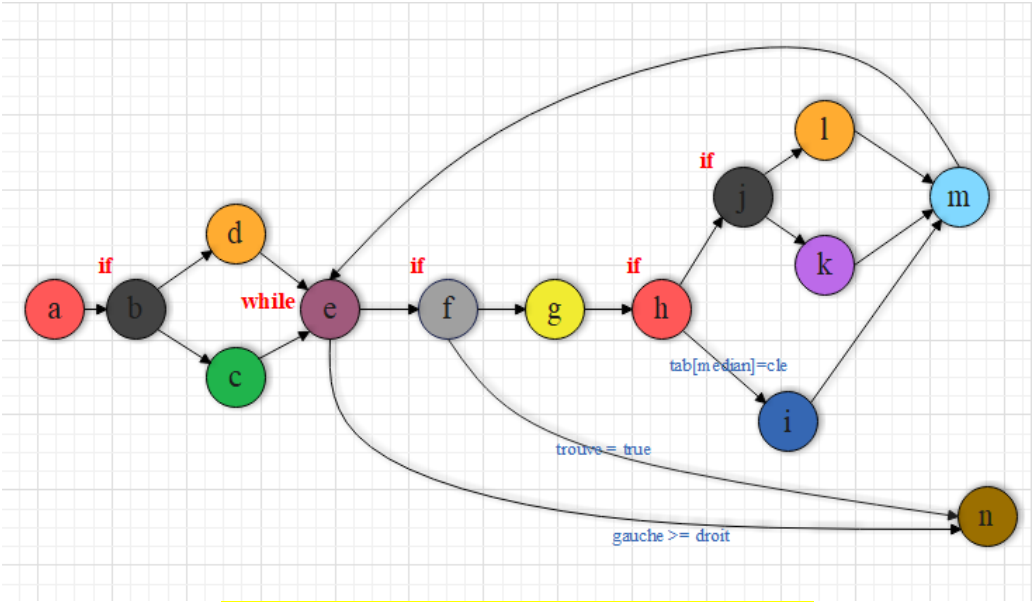
Figure 4 : WHILE double

>>>

```
while gauche <= droit
    if trouve = false
```

Figure 5 : WHILE simple

Déduisons alors de ce nouveau pseudo code un second graphe (graphe n°2) un graphe contrôle WHILE simple cette fois-ci. De plus, on peut remarquer que l'on a rajouté une condition IF juste après la condition WHILE. Ce qui nous fait au total 4 conditions IF.



Graphe 2 : graphe de contrôle WHILE simple

Étape 2 : Calculer le nombre cyclomatique et donner sa signification pour le code :

Ce modèle est fondé sur l'étude de la structure du graphe de contrôle d'un programme. La métrique de McCabe calcule le nombre cyclomatique (ou complexité structurelle) de ce graphe. Sachant d'après le cours que V(G) = nombre cyclomatique de McCabe. On peut alors calculer le nombre cyclomatique de nos graphes, de plus on sait qu'il y a 1 point d'entrée et 1 point de sortie, soit V(G) = m(nombre de nœuds) - n(nombre d'arcs) + 2.

Pour le graphe WHILE double :
Calculons le nombre cyclomatique : V(G) = m-n+1 = 16-13+2 = 5

Pour le code, on a V(G)-1 = 4 décisions :
- 1 WHILE double (nœud e),
- 3 IF (nœuds b, g et j),

Même raisonnement pour le graphe WHILE simple :
Calculons le nombre cyclomatique : V(G) = m-n+1 = 18-14+2 = 6

Pour ce code, on a V(G)-1 = 5 décisions simples :
- 1 WHILE (nœud e),
- 4 IF (nœuds b, f, h et j),

Étape 3 : Établir l'expression des chemins de contrôles :

Commençons pour le graphe WHILE double :

L'entrée du graphe (graphe 1) commence par le nœud a puis réalise une forme séquentielle avec b (premier IF) du type M = ab. On continue avec une forme alternative de c, d et e du type M = ab(c+d)e. On arrive au nœud e (représentant le double WHILE) et on constate une forme itérative passant par f puis g laissant apparaître un début de forme alternative en se séparant en un chemin i et h. Le chemin par le nœud i continue par une forme alternative de j et k du type M = i(j+k)l pour aller sur le nœud l. Le chemin h va directement vers l.

On résume donc le type : M = ab(c+d)e(fg(h+i(j+k)l. Sachant que le nœud part en direction du nœud e pour finir sur le nœud m (sa sortie).

On obtient donc : M = ab(c+d)e(fg(h+i(j+k)le))*m

Passons au graphe WHILE simple :

L'entrée du graphe (graphe 2) commence de la même façon que le graphe précédent. Avec le début d'expression M = ab(c+d)e. On arrive donc au nœud e (représentant cette fois-ci un simple WHILE) et on constate une forme itérative passant par f qui se sépare en une forme alternative simple vers g et n. Le chemin g continue vers le nœud h qui se sépare par la forme alternative en deux chemins vers i et j. Le chemin vers j s'avance dans une forme alternative a nouveau par k, l pour aller vers m.

On le résume par cette expression : M = j(k+l)m. Le chemin de i va directement vers le nœud m. A partir du nœud m, le chemin repart vers e pour finir vers n.

On a alors pour ce graphique : M = ab(c+d)e(f(1+gh(i+j(k+l))me))*n

Étape 4 : Établir le tableau des PI() expressions (ou dr-chaînes) correspondant aux différentes variables :

Variables	PI (variables)
median	(dr[r+r[r+r]])*
cle	d(r(r[1+r])*)*
trouve	d[d+d]d([d+1])*
droit	drr(r(1+(1+d)))*
gauche	drr(r(1+(d+1)))*
A	dr([d+1])*
tab[]	d(r(r[1+r])*)*
taille	d*
inf	r
inc	r
sup	r

Étape 5 : Dresser le tableau qui met en évidence d'éventuelles anomalies dans le flot de données.

Le flot de données est composé de plusieurs types d'anomalies parmi lesquelles on retrouve des variables utilisées sans être préalablement définies, des variables définie sans être utilisée et des variables définie plusieurs fois consécutivement. Dans notre cas, vois-ci les différentes anomalies que nous avons pu trouver :

Variables	Anomalies
taille	d*: Variable définie mais non utilisée
inf	r: Variable non initialisée
sup	r: Variable non initialisée
inc	r: Variable non initialisée
A	Variable définie dans la boucle WHILE mais aucune d'utilité
trouve	d[d+d] : Variable définie dans un chemin plusieurs fois

Étape 6 : En utilisant les résultats qui précèdent, proposer une mise au point de la procédure sous la forme d'un code C, C++ ou Java pour obtenir le composant logiciel sous test.

Dans cette étape, nous avons créé le logiciel de sous-test (joint avec le compte rendu Exercice1-Christoph-Jankowiak.c) permettant de retrouver une clé souhaitée par l'utilisateur dans un tableau triée par recherche dichotomique.

Selon les résultats précédents, on constate que inf, sup et inc sont inutiles. La première boucle est également inutile, car l'opération est déjà effectuée. De plus, median est devient inutilisable, on la remplace alors par A (le tab) et on initialise directement trouve à false. Enfin, du fait de ces changements, on obtient alors un nouveau code ainsi qu'une nouvelle fonction look() ressemblant a aux différentes captures ci-dessous.

```c
/*
 * Ent-tête nécessaire à la programmation.
 */
#include <stdio.h>
#include <stdlib.h>
/*
 * Définition des éléments de type booléen.
 */
typedef int bool;
#define FALSE 0
#define TRUE 1
/*
 * Fonction qui permet de remplir un tableau.
 */
void remplirTableau(int tableau[], int taille)
{
    for (int i = 0 ; i < taille ; i++)
    {
        printf("Entrez l'element %d : ", i + 1);
        scanf("%d", &tableau[i]);
    }
}
/*
 * Fonction qui permet d'affficher un tableau.
 */
void afficheTableau(int tableau[], int taille)
{
    printf("Le tableau est le suivant: ");
    for (int i = 0 ; i < taille; i++)
    {
        printf("%d ", tableau[i]);
    }
    printf("\n");
}
```

Dans notre logiciel test nous pouvons retrouver en premier lieu les en-têtes des éléments nécessaires a la programmation en c, le définition des élément de type booléens (car inexistant en

c), mais également la définition de deux fonctions remplirTableau() et afficheTableau() qui, comme leur noms l'indique, permettent de créer et afficher un tableau à x éléments définit par l'utilisateur

```c
/*
 * Fonction de recherche dichotomique d'un élément dans un tableau trié.
 */
int look(int cle, int *tab, int taille, int trouve, int A)
{
    int droit, gauche;
    trouve = FALSE;
    droit = taille;
    gauche = 0;

    while(gauche <= droit && trouve == FALSE)
    {
        A = ((droit+gauche)/2);
        if (tab[A] == cle)
        {
            trouve = TRUE;
        }
        else
        {
            if(tab[A] > cle)
            {
                gauche = A+1;
            }
            else
            {
                droit = A-1;
            }
        }
    }
    return trouve;
}
```

Ensuite une fonction look(), elle va permettre une analyse complète du tableau trié avec les variables droit, gauche, A, trouve et taille cette fonction suite à l'analyse peut donc nous renvoyer deux valeurs true (vrai) si l'élément cherché (clé) et son indice (A) on été trouvé, false (faux) sinon.

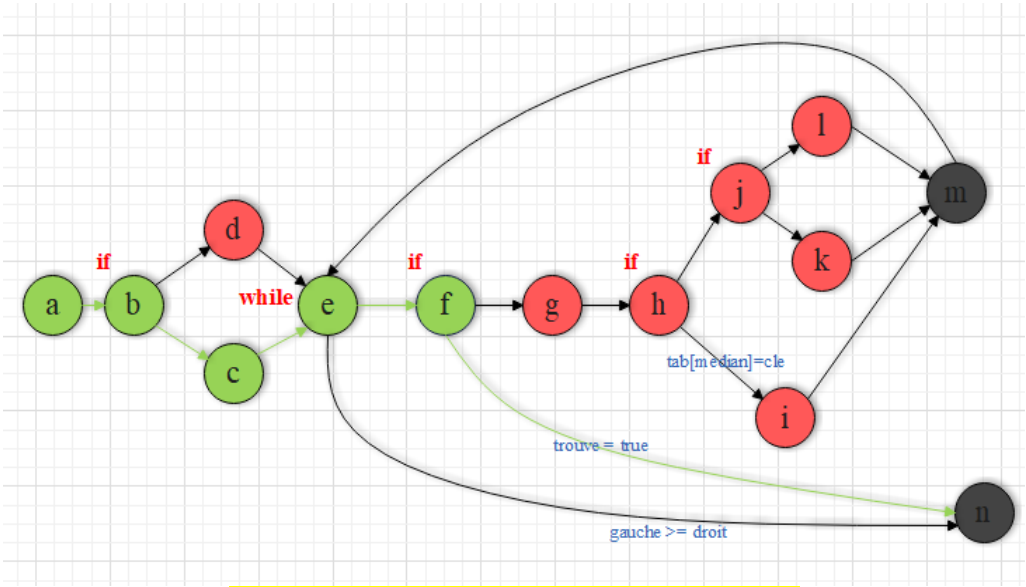
```c
/*
 * Programme principal.
 */
int main(int argc, char *argv[])
{
    int cle, taille, A;
    bool trouve;
    int tableau[taille];
    /*
     * On donne la taille du tableau.
     */
    printf("\nIndiquez la taille du tableau : ");
    scanf("%d", &taille);
    /*
     * Appel des fonctions pour remplir et afficher le tableau.
     */
    printf("\n");
    remplirTableau(tableau, taille);
    printf("\n");
    afficheTableau(tableau, taille);
    printf("\n");
    /*
     * On renseigne l'élément recherché et on appelle la fonction de recherche.
     */
    printf("Donner l'élément recherché : ");
    scanf("%d", &cle);
    if(look(cle, tableau, taille, trouve, A))
    {
        printf("\nL'élément est présent à l'indice %d dans la liste.\n\n", A);
    }
    else
    {
        printf("\nL'élément n'est pas présent dans la liste.\n\n");
    }
    return 0;
}
```

La fonction main() va seulement nous permettre d'informer et de communiquer avec l'utilisateur, en effet elle lui demande la taille du tableau, la clé recherchée, les différents éléments de celui-ci et affiche le tableau pour plus de compréhension. Enfin elle effectue la fonction look et renvoie si l'élément est présent dans là liste ou au contraire s'il ne s'y trouve pas.

Étape 7 : Générer un jeu de DT pour sensibiliser les chemins exécutables permettant la couverture de tous les nœuds du graphe de contrôle, déterminer le taux de couverture de chaque DT :

Déterminer le taux de couverture consiste à sensibiliser suffisamment de chemins de sorte à couvrir tous les nœuds. Le taux de couverture ou mesure de complétude exprime le degré de satisfaction de ce critère. Il est donné par le TER1 (Test Effectiveness Ratio 1), TER1 = Nc1/Nt1 (*avec : Nc1 = le nombre de nœuds couverts et Nt1 = le nombre total de nœuds).*

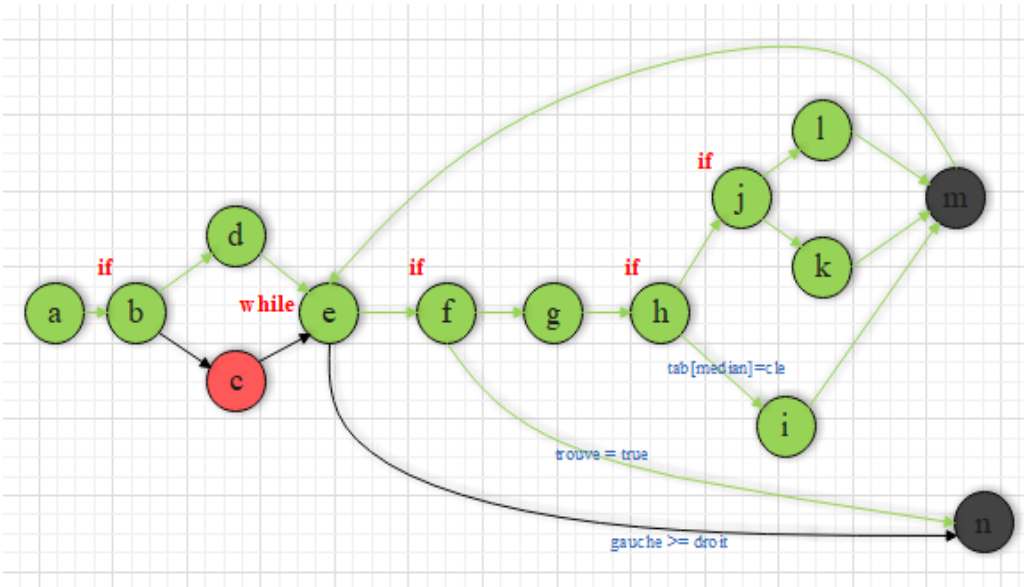
Prenons comme DT1 : {clé = 35, tab[] = [61,52,44,38,35,32,24,12,8], taille = 9, trouve = false},



Jeu DT1 pour l'élément « Tous-les-noeuds »

Cette DT sensibilise le chemin exécutable a/b/c/e/f soit 5/12. On peut en déduire un taux de couverture des nœuds TER1 = (5/12)x100 = 41,6%.

Prenons comme DT2 : {clé = 32, tab[] = [61,52,44,38,35,32,24,12,8], taille = 9, trouve = false},



Jeu DT2 pour l'élément « Tous-les-noeuds »

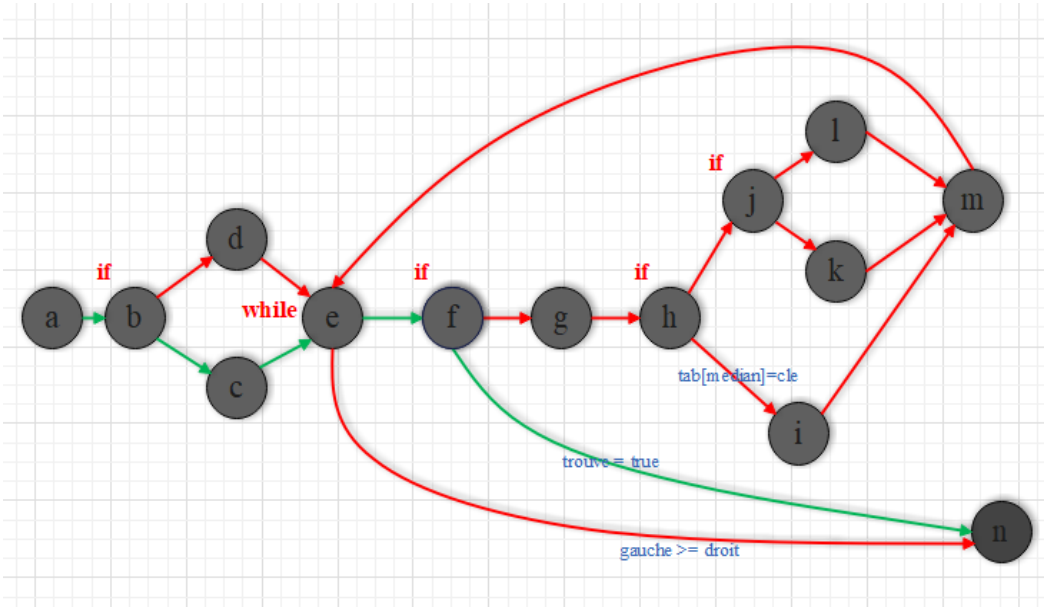
Cette DT sensibilise le chemin exécutable a/b/d(e/f/g/h/j/l)*(e/f/g/h/j/k)*(e/f/g/h/i) soit 11/12. On peut en déduire un taux de couverture des nœuds TER1 = (11/12)x100 = 91,6%.

L'ensemble de ses 2 DT nous permet de sensibiliser tous les nœuds du graphe de contrôle, effet la DT2 couvre 11 des 12 nœuds et la DT1 couvre parmi ces nœuds le nœud manquant (m et n ne comptant pas comme nœud, car ce sont des fin de boucle/programme). Cependant, le critère tous-les-nœuds est insuffisant pour détecter un grand nombre de défauts. Le problème provient du fait que nous avons couvert tous les nœuds du graphe, mais pas tous les arcs (par exemple l'arc {en} dans ce jeu de DT n'est pas couvert).

Étape 8 : Générer un jeu de DT pour sensibiliser les chemins exécutables permettant la couverture de tous les arcs du graphe de contrôle, déterminer le taux de couverture de chaque DT :

C'est pour cela qu'il faut définir donc un critère plus fort : le critère tous-les-arcs. Un tel critère impose de couvrir tous les arcs du graphe : {ab, bc, bd, ce, de, ef, fg, gh, hj, hi, jl, jk, lm, km, im, me, fn}. Il est donné par le TER2 (Test Effectiveness Ratio 2), TER2 = Nc2/Nt2 *(avec : Nc2 = nombre d'arcs couverts et Nt1 = nombre total d'arcs).*

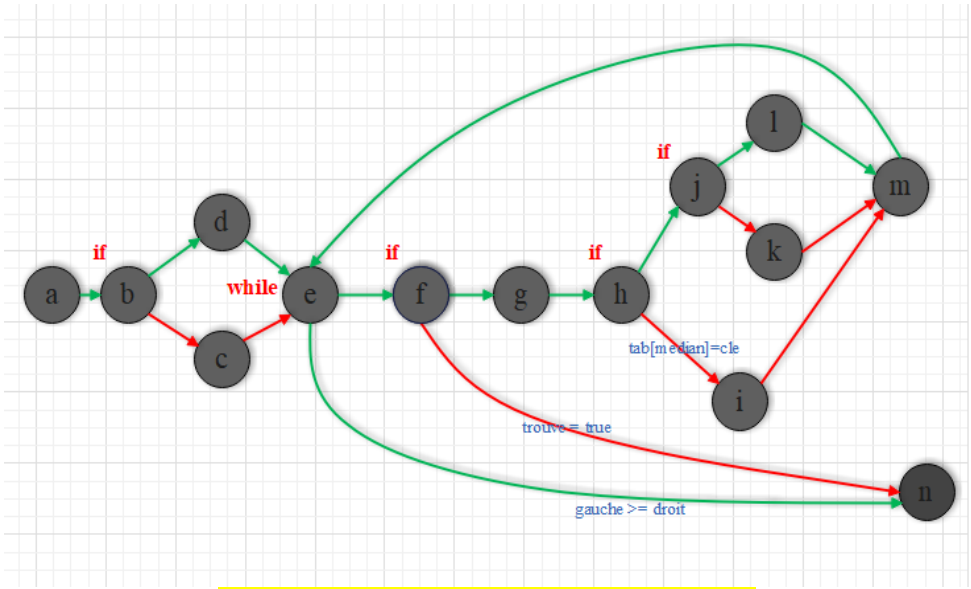
- <u>Prenons comme DT1 :</u>
{clé = 35, tab[] = [61,52,44,38,35,32,24,12,8], taille = 9, trouve = false},



Jeu DT1 pour l'élément « Tous-les-Arcs »

Cette DT permet de couvrir les arcs {ab, bc, ce, ef, fn}, on peut en déduire un taux de couverture des arcs TER2 = (5/18)x100 = 27,7%.

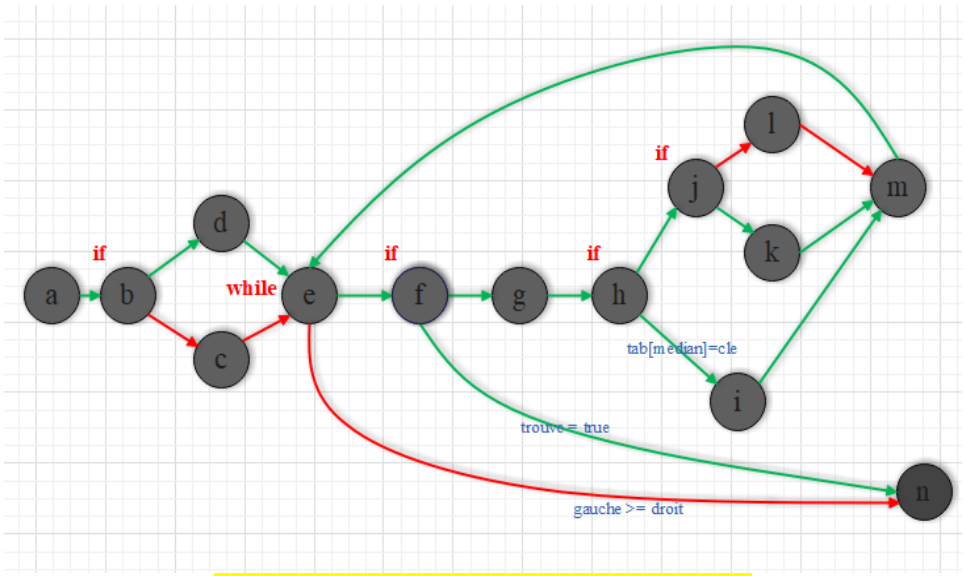
- <u>Prenons comme DT2 :</u>
{clé = 4, tab[] = [61,52,44,38,35,32,24,12,8], taille = 9, trouve = false},



Jeu DT2 pour l'élément « Tous-les-Arcs »

Cette DT permet de couvrir les arcs {ab, bd, de, ef, fg, gh, hj, jl, lm, me, en}, on peut en déduire un taux de couverture des arcs TER2 = (11/18)x100 = 61,1%.

- <u>Prenons comme DT3 :</u>
{clé = 32, tab[] = [61,52,44,38,35,32,24,12,8], taille = 9, trouve = false},



Jeu DT3 pour l'élément « Tous-les-Arcs »

Cette DT permet de couvrir les arcs {ab, bd, de, ef, fg, gh, hj, jl, lm, me, hi, im, jk, km, fn}, on peut en déduire un taux de couverture des arcs TER2 = (15/18)x100 = 83,3%.

L'ensemble de ses 3 DT nous permet de couvrir tous les arcs {ab, bc, bd, ce, de, ef, fg, gh, hj, hi, jl, jk, lm, km, im, me, fn} du graphe de contrôle. Cependant, le critère tous-les-arcs est incapable de détecter des défauts fréquents, pour cela on génère un troisième critère : le critère « Tous-les-Chemins ».

Étape 9 : Générer un jeu de DT pour sensibiliser tous les chemins indépendants du graphe et déterminer le taux de couverture de chaque DT :

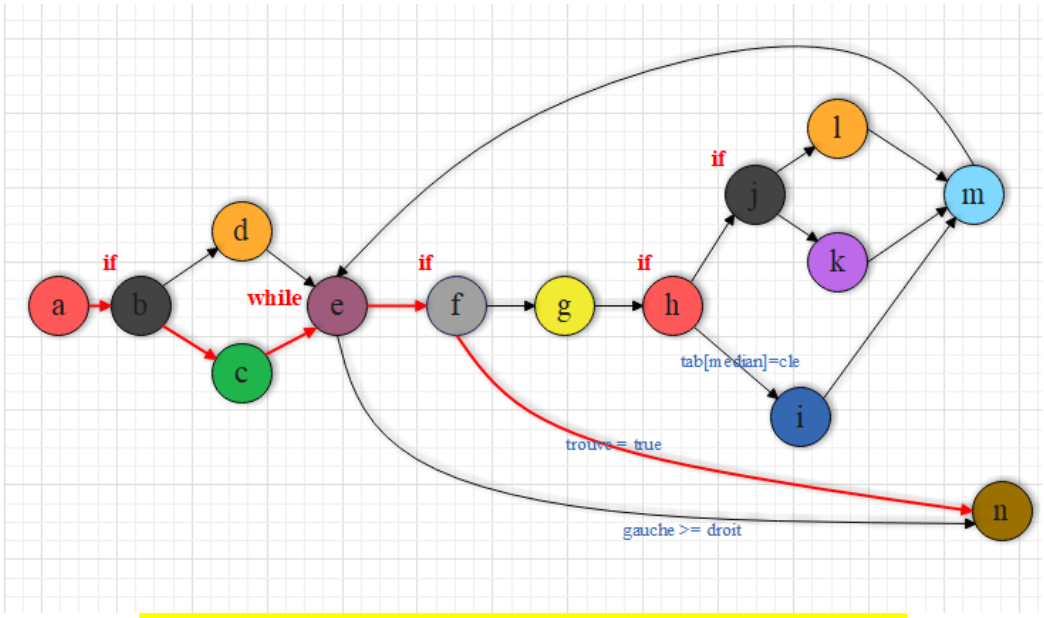
C'est pour cela qu'on introduit donc un troisième critère : le critère « Tous-les-chemins-indépendants ». Pour cela, nous devons générer un jeu de DT pour sensibiliser chaque chemin dépendants du graphe et déterminer le taux de couverture de chaque DT. Il est donné par le TER3 (Test Effectiveness Ratio 3), TER3 = Ncc / V(G) *(avec : Ncc = nombre de chemins couverts et V(G) = nombre cyclomatique de McCabe).*

Le nombre cyclomatique de notre graphe de contrôle est V(G) = 6, de ce fait, il possède 6 chemins indépendants. On choisit alors les 6 DT suivantes :

- **Premier jeu de DT :** On opte pour un premier chemin avec comme paramètres :
{clé = 35, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On a alors A = (droite + gauche)/2 = (1+9)/2 = 5 soit tab(5) = 35 et trouve = true ainsi le chemin suivant s'offre à nous :



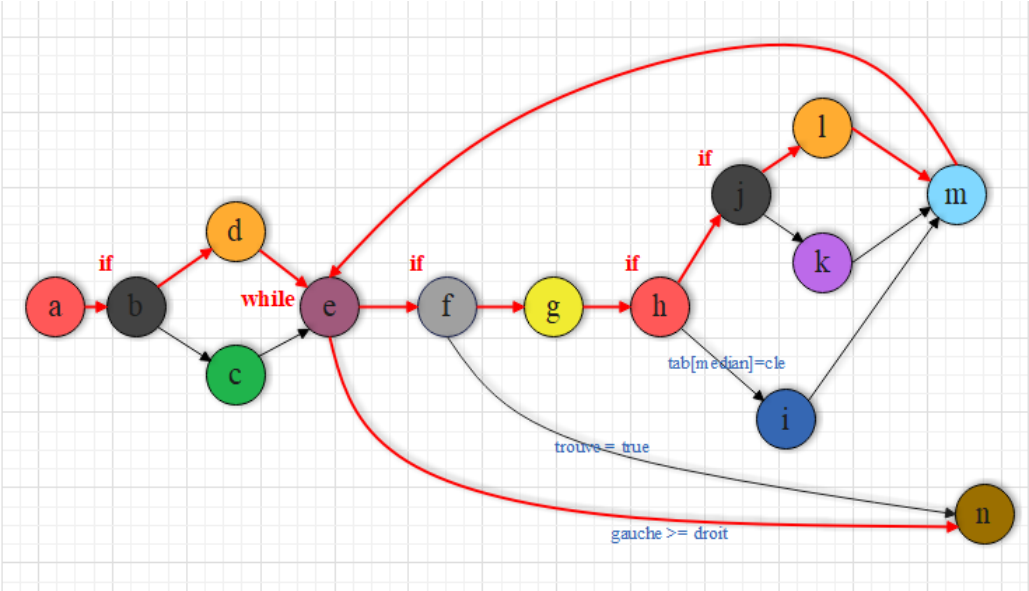
Jeu DT1 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abcefn]**, avec Ncc = 1 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 1/6 = 0,16667 soit 16,6%.

- **Deuxième jeu de DT :** On opte pour un second chemin avec comme paramètres :
{clé = 5, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On remarque que tab[median]>cle c'est-à-dire 8 > 5 on a donc 'then gauche:=median+1' soit gauche = 9+1 =10 > droit ainsi le chemin suivant s'offre à nous :



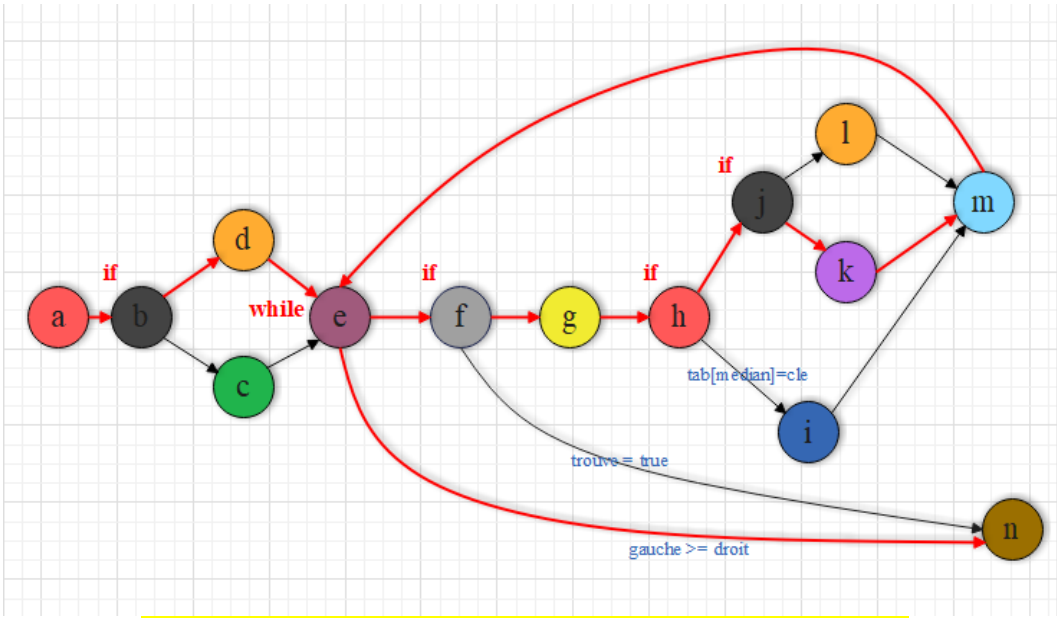
Jeu DT2 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abd(efghjlm)*en]**, avec Ncc = 1 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 1/6 = 0,16667 soit 16,6%.

- <u>Troisième jeu de DT :</u> On opte pour un troisième chemin avec comme paramètres : {clé = 72, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On remarque que tab[median]<cle c'est-à-dire 72 < 61 on a donc 'then droite:=median-1' soit droit = 1-1 = 0 < gauche ainsi le chemin suivant s'offre à nous :



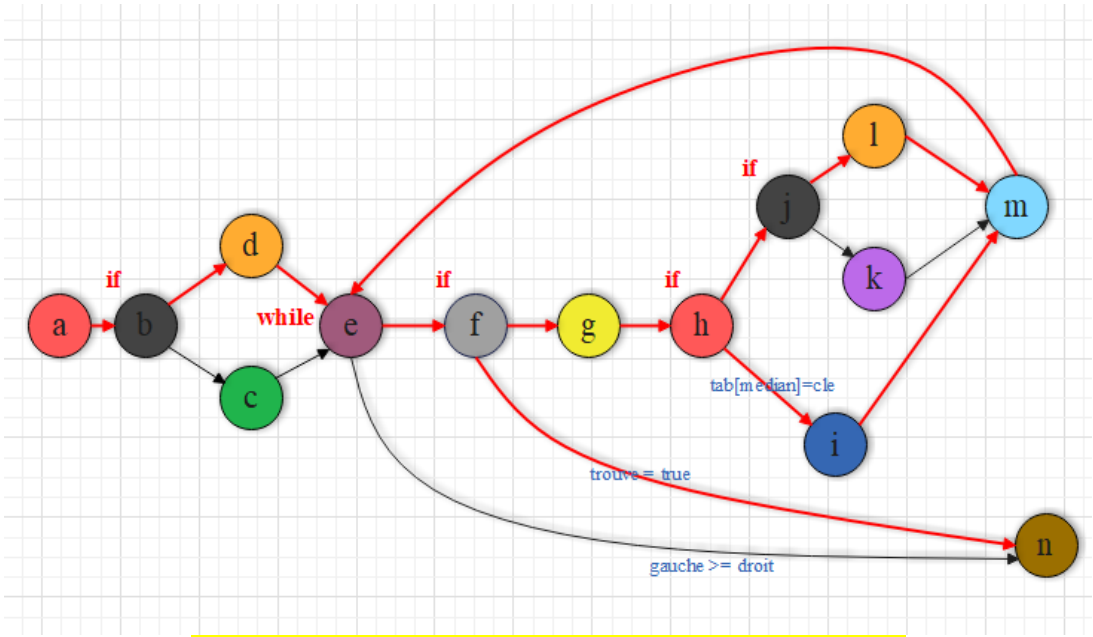
Jeu DT3 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abd(efghjkm)*en]**, avec Ncc = 1 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 1/6 = 0,16667 soit 16,6%.

- <u>Quatrième jeu de DT :</u> On opte pour un quatrième chemin avec comme paramètres : {clé = 24, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On a alors A = (droite + gauche)/2 = (1+9)/2 = 5 soit tab(5) = 35>24 et trouve = false ainsi le chemin suivant s'offre à nous :



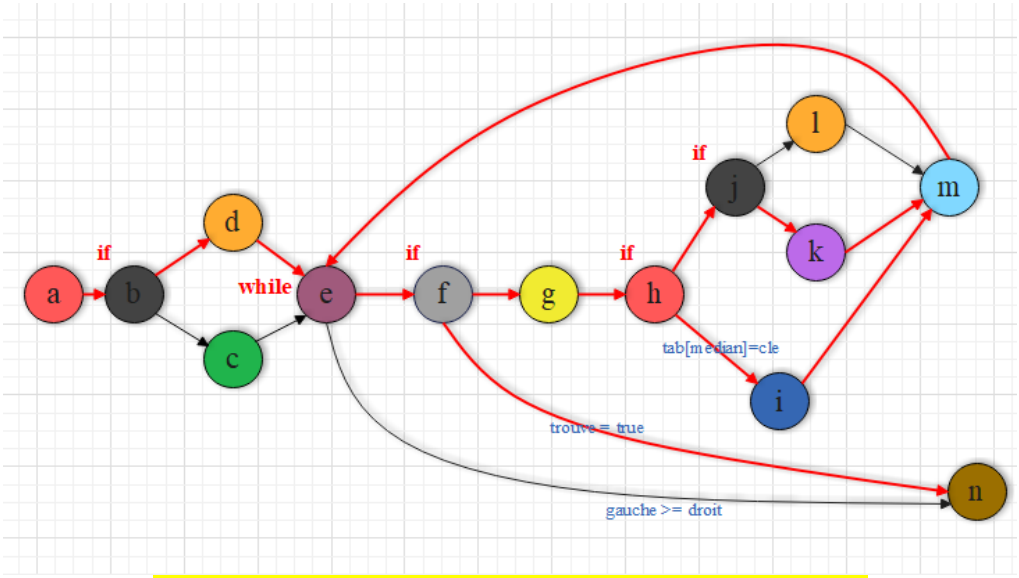
DT4 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abd(efghjlm)efghimefn]**, avec Ncc = 2 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 2/6 = 0,333 soit 33,3%.

- <u>Cinquième jeu de DT :</u> On opte pour un cinquième chemin avec comme paramètres : {clé = 52, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On a alors A = (droite + gauche)/2 = (1+9)/2 = 5 soit tab(5) = 35<52 et false = true ainsi le chemin suivant s'offre à nous :



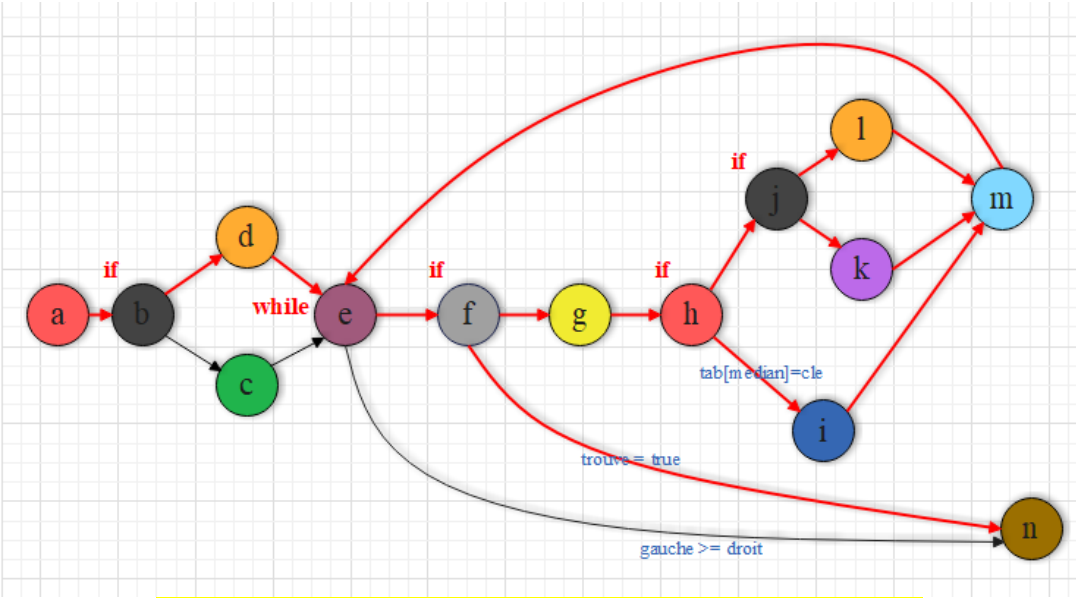
DT5 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abdefghjkmefghimefn]**, avec Ncc = 2 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 2/6 = 0,333 soit 33,3%.

- <u>Sixième jeu de DT :</u> On opte pour un sixième chemin avec comme paramètres : {clé = 38, taille = 9, trouve = false},

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

On a alors A = (droite + gauche)/2 = (1+9)/2 = 5 soit tab(5) = 35<38 et false = true ainsi le chemin suivant s'offre à nous :



Jeu DT6 pour l'élément « Tous-les-chemins-indépendants »

Cette DT permet d'exprimer le chemin **[abdefghjkmefghjlmefghimefn]**, avec Ncc = 3 et V(G) = 6, on peut en déduire un taux de couverture pour se chemin de TER3 = 3/6 = 0,5 soit 50%.

Étape 10 : Exécuter le composant logiciel sous test sur les données générés en 7), 8), et 9):

Pour cette étape, nous allons utiliser le code réalisé à l'étape 6. En exécutant le composant logiciel sous test sur les données générés en question 7, 8 et 9, on se rend compte que toutes les DT marche et on ne retrouve pas d'anomalies. Voici quelques exemples pour illustrer cette étape :

Dans ce premier exemple nous testerons le jeu de DT1 de l'étape 9 pour l'élément « Tous-les-chemins-indépendants » avec comme paramètres : {clé = 35, taille = 9, trouve = false} et le tableau suivant :

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

```
schristoph@scinfe142 ~/Bureau/stockage/TD2/TD2 - CHRISTOPH Samuel - JANKOWIAK Matthias
./Exercice1-Christoph-Jankowiak.out

Indiquez la taille du tableau : 9

Entrez l'element 1 : 61
Entrez l'element 2 : 52
Entrez l'element 3 : 44
Entrez l'element 4 : 38
Entrez l'element 5 : 35
Entrez l'element 6 : 32
Entrez l'element 7 : 24
Entrez l'element 8 : 12
Entrez l'element 9 : 8

Le tableau est le suivant: 61 52 44 38 35 32 24 12 8

Donner l'élément recherché : 35

L'élément est présent à l'indice 5 dans la liste.
```

Dans ce cas cette TD réalise un chemin indépendant et l'élément est trouvé dés le départ car il se trouve au milieu du tableau, ce que le logiciel de sous-test nous confirme : « L'élément est présent à l'indice 5 dans le tableau. »

Pour le second exemple nous testerons le jeu de DT5 de l'étape 9 pour l'élément « Tous-les-chemins-indépendants » avec comme paramètres : {clé = 52, taille = 9, trouve = false} et le tableau suivant :

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

```
schristoph@scinfe142 ~/Bureau/stockage/TD2/TD2 - CHRISTOPH Samuel - JANKOWIAK Matthias
./Exercice1-Christoph-Jankowiak.out

Indiquez la taille du tableau : 9

Entrez l'element 1 : 61
Entrez l'element 2 : 52
Entrez l'element 3 : 44
Entrez l'element 4 : 38
Entrez l'element 5 : 35
Entrez l'element 6 : 32
Entrez l'element 7 : 24
Entrez l'element 8 : 12
Entrez l'element 9 : 8

Le tableau est le suivant: 61 52 44 38 35 32 24 12 8

Donner l'élément recherché : 52

L'élément est présent à l'indice 2 dans la liste.
```

Dans ce cas cette TD réalise un chemin indépendant et l'élément est trouvé après plusieurs tour dans la boucle WHILE, ce que le logiciel de sous-test nous confirme : « L'élément est présent à l'indice 2 dans le tableau. »

Pour le dernier exemple nous testerons le jeu de DT2 de l'étape 8 pour l'élément « Tous-les-Arcs » avec comme paramètres : {clé = 4, taille = 9, trouve = false} et le tableau suivant :

1	2	3	4	5	6	7	8	9
61	52	44	38	35	32	24	12	8

```
schristoph@scinfe142 ~/Bureau/stockage/TD2/TD2 - CHRISTOPH Samuel - JANKOWIAK Matthias
./Exercice1-Christoph-Jankowiak.out

Indiquez la taille du tableau : 9

Entrez l'element 1 : 61
Entrez l'element 2 : 52
Entrez l'element 3 : 44
Entrez l'element 4 : 38
Entrez l'element 5 : 35
Entrez l'element 6 : 32
Entrez l'element 7 : 24
Entrez l'element 8 : 12
Entrez l'element 9 : 8

Le tableau est le suivant: 61 52 44 38 35 32 24 12 8

Donner l'élément recherché : 4

L'élément n'est pas présent dans la liste.
```

Dans ce cas cette TD réalise un chemin indépendant comme pour le jeu de DT2 de l'étape 9, l'élément ici n'est trouvé après plusieurs tour dans la boucle et la condition WHILE se termine, ce que le logiciel de sous-test nous confirme : « L'élément n'est pas présent dans le tableau. »

III-BILAN/CONCLUSION :

1- En conclusion, dans cet exercice nous avons donc pu étudier différentes approches pour réaliser un test structurel, les test structurels se déroulent en 2 phases, statique et dynamique, chaque phase est importante car ils vérifie des erreurs différentes. La partie statique qui se repose sur l'analyse du code, est indicateur des erreurs lié la spécification et permet de savoir si le code n'est pas trop lourd en décision et en complexité , de plus les pi expressions permettent de vérifié l'utilisation des variables. Le graphe de contrôle permet de vérifié les conditions et le bon déroulement du code.
La partie dynamique, elle, a pour vocation de vérifié les erreurs lié au codage, comme les nœuds du graphe de contrôle qui ne seraient pas utilisés, ou encore des arcs inutilisés.

2- Son application à un cas concret nous a donc permis de nous familiariser avec les graphes de contrôle qui permettent de traduire par des nœud et des arcs les blocs d'instructions et formalise un transfert de l'exécution d'un bloc d'instruction à un autre. Ce graphe nous permet également de visualiser l'algorithme d'une autre façon pour en trouver tous les chemins existants (formalise une séquence d'exécution) : il révèle un comportement du logiciel.Il est également important pour en déduire le nombre de cyclomatique et l'expression du graphe mais aussi de chaque chemins indépendants grâce aux jeux de DT. Cet exercice nous a permis aussi de comprendre et d'analyser des cas concret d'un algorithme et ce qu'il va en découdre. On peur savoir à l'avance tout les chemins possibles. Nous avons également réaliser un deuxième graphe (en WHILE simple) qui montre une réutilisation du premier algorithme et de montrer une version différente (mais également trouver une amélioration). Ce deuxième graphe nous a permit de déduire plus de chemins. On peut donc changer des algorithmes après avoir découvert leur jeux de DT pour une meilleure efficacité.

3- Réaliser cette série, nous a permis de constater que pour certains programmes, des données de tests peuvent couvrir l'ensemble de celui-ci. Permettant ainsi un test de qualité optimale. L'ensemble des étapes à suivre pour en arriver a pouvoir créer un jeu de DT nous a permis de mieux comprendre notre propre code, et aussi de mieux le structurer.Contrairement aux tests fonctionnels qui détectent plus facilement les erreurs de spécifications, les tests structurels détectent plus facilement les erreurs de codage. Les tests structurels sont d'une grande importance et prennent du sens dans des gros codes complexes et importants, car c'est un outil extrêmement puissant. Nous en arrivons donc à la conclusion que, connaître des techniques de tests permet de coder de manière plus structurée et plus efficace.