Travaux dirigés de test du logiciel

Christian Rinderknecht

4 février 2015

L'objectif de ce travail est de se familiariser avec les graphes de contrôle, leur construction à partir d'un programme, de comprendre leur propriétés (dont le nombre cyclomatique), et d'en déduire un jeu de test qui couvre tous les chemins exécutables. Cette notion de chemin est replacée dans le contexte des automates finis. Un aperçu de la preuve de programme est donné, ainsi que son rôle par rapport au test.

1 Graphes de contrôle et couverture de test

1.1 Définitions

Le *graphe de contrôle* d'un programme est un graphe qui permet de modéliser un sur-ensemble de toutes les exécutions possibles d'un programme donné. Un graphe est un ensemble fini de nœuds et un ensemble d'arcs les reliants, ces arcs pouvant être étiquetés. Chaque nœud correspond à un point du programme (par exemple une instruction), et les arcs représentent les changements possibles de l'état du programme (c'est-à-dire de l'ensemble des valeurs des variables). Les arcs sont donc orientés. Parmi les nœuds, on distingue aussi un nœud correspondant à l'état initial et un nœud correspondant à l'état terminal (ou *final*). Pour les distinguer aisément, nous ajoutons deux arcs, l'un entrant vers le nœud initial (arrivant de nulle-part) et l'autre sortant du nœud terminal (allant nulle-part). Le graphe capture un sur-ensemble des exécutions car il est construit en suivant la syntaxe du programme, donc en ignorant sa sémantique (c'est-à-dire son comportement lors de l'exécution).

Un *jeu de tests* est un ensemble de chemins, un chemin étant une suite d'arcs successifs dans le graphe de contrôle, le premier issu de l'état initial et le dernier parvenant à l'état terminal. Lorsque les étiquettes des arcs sont uniques, et caractérisent donc de façon unique un arc, les chemins seront notés par la juxtaposition des étiquettes successives des arcs formant le chemin.

Selon les objectifs de test, un graphe de contrôle sera plus ou moins détaillé. Ici, le but est de couvrir toutes les parties *a priori* accessibles d'un programme. Nous concentrerons donc notre attention sur les variables qui influent sur les chemins dans le programme.

Le *nombre cyclomatique* d'un graphe est un majorant du nombre minimal de chemins recouvrant intégralement le graphe, il vaut $\gamma = A - N + 2$, où A est le nombre d'arcs (sans compter les arcs spéciaux caractérisant les états initiaux terminaux), et N est le nombre de nœuds. Après le calcul du graphe de contrôle, la première étape consiste à calculer son nombre cyclomatique, car celui-ci est un majorant du nombre minimal de tests de couverture (un test est

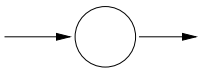
ici un chemin). Le nombre cyclomatique est une mesure de la *complexité statique* d'un programme, et une bonne habitude consiste à n'écrire que des fonctions dont le nombre cyclomatique (par abus de langage) est inférieur à dix. D'autre part, il faut se souvenir que certains chemins ne sont peut-être pas exécutables, par exemple si une condition n'est jamais vraie.

L'étape suivante consiste à annoter les arcs du graphe (par exemple par des lettres, s'il y a moins de vingt-six arcs), et rechercher un ensemble minimal de chemins recouvrants. Il est possible de considérer le graphe de contrôle, avec son état initial et terminal, ainsi que les annotations d'arcs, comme étant un *automate fini*. Dualement, les chemins dans le graphe de contrôle peuvent donc être considérés comme des *expressions régulières* sur l'alphabet des annotations d'arcs. Tous les tests devant être finis, les expressions régulières associées sont donc finies (syntaxiquement, cela est équivalent à l'absence d'opérateur de Kleene, ou « étoile »).

1.2 Construction du graphe de contrôle

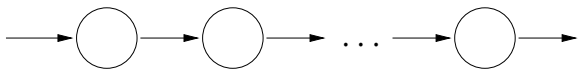
Le graphe de contrôle étant construit à partir du programme, cette construction dépend de la syntaxe et de la sémantique du langage de programmation, en particulier du paradigme qui le sous-tend, tel que impératif, fonctionnel, à objets, à logique etc. Nous ferons ici l'hypothèse que notre langage est impératif, avec une notion d'instruction et d'expression. Nous définissons la construction du graphe à partir d'un pseudo-langage (ce n'est pas un vrai langage de programmation existant) dont nous espérons que la syntaxe et la sémantique seront intuitifs. Le principe de construction est d'associer un nœud du graphe à chaque instruction et expression booléenne contrôlant une conditionnelle ou une boucle, et un arc les joignant lorsque le contrôle passe d'une instruction à l'autre. Il faut rajouter des nœuds de jonction correspondant à la fin d'une boucle ou d'une conditionnelle. Il faut aussi rajouter deux arcs spéciaux, un entrant et un sortant, dénotant respectivement l'état initial et l'état terminal. Voyons cela cas par cas :

— **L'instruction simple.** À toute instruction de simple (telle que l'affectation notée `<-` ou, plus joliment, ←) correspond le graphe de contrôle simple



Le nombre d'arcs de ce graphe est 0 (on ne compte pas les arcs spéciaux dénotant l'état initial et terminal). Le nombre de nœud est évidemment 1, donc le nombre cyclomatique de ce graphe est $\gamma = 0 - 1 + 2 = 1$. L'interprétation de ce nombre en termes de majorant du nombre minimal de chemins recouvrants (du nœud initial au nœud terminal) est trivialement vérifiée ici : il n'y a besoin que d'un seul chemin qui est le nœud 1 lui-même.

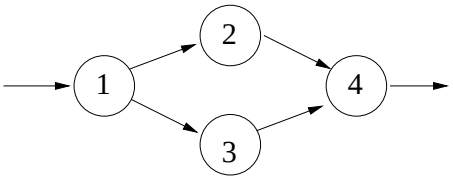
— **La séquence d'instructions.** Les instructions sont séparées des instructions suivantes par un point-virgule. Le graphe associé à une telle séquence est un chemin



S'il y a n instructions, il y a donc n nœuds. Étant donné qu'il n'y a pas

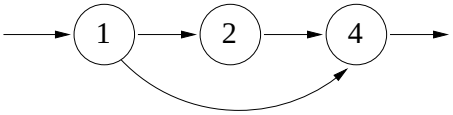
de circuit, il y a donc $n-1$ arcs. Par conséquent le nombre cyclomatique de ce graphe est : $\gamma = (n-1) - n + 2 = 1$, exactement comme dans le cas d'une seule instruction. On constate en effet qu'il n'y a besoin que d'un seul chemin pour couvrir tout le graphe du nœud initial au nœud terminal.

— **La conditionnelle.** Elle a pour syntaxe **if** <*expression booléenne*> **then** <*instructions*> [**else** <*instructions*>] **endif**. Les nœuds du graphe de contrôle correspondent à l'expression booléenne (nœud étiqueté 1), aux instructions (nœuds étiquetés 2 si la condition est vraie, et 3 sinon) et au mot-clé **endif** (nœud étiqueté 4). Nous avons le graphe de la conditionnelle complète



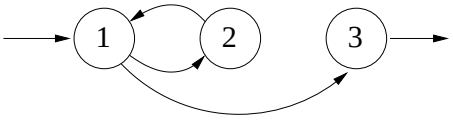
Le nombre d'arcs est 4, et le nombre de nœuds est 4 aussi. Donc le nombre cyclomatique de ce graphe est $\gamma = 2$. Il est évident qu'il suffit de deux chemins, $(1, 2, 4)$ et $(1, 3, 4)$, pour couvrir tout le graphe du nœud initial au nœud terminal.

Le graphe de la conditionnelle sans clause **else** est



Le nombre d'arcs est 3 et le nombre de nœuds est 3 aussi. Donc le nombre cyclomatique de ce graphe est $\gamma = 2$, comme dans le cas de la conditionnelle complète. Ici aussi, deux chemins, $(1, 2, 4)$ et $(1, 4)$, suffisent à couvrir le graphe.

— **L'itération non bornée.** L'itération non bornée a pour syntaxe **while** <*expression booléenne*> **do** <*instructions*> **done**. Les nœuds du graphe de contrôle correspondent à l'expression booléenne (nœud étiqueté 1), aux instructions (nœud étiqueté 2) et au mot-clé **done** (nœud étiqueté 3). Le graphe est alors



Le nombre d'arcs est 3 et le nombre de nœuds est 3 aussi. Donc le nombre cyclomatique de ce graphe est $\gamma = 2$. Ici, un chemin est suffisant pour couvrir le graphe : $(1, 2, 1, 3)$. Le fait que le nombre cyclomatique soit ici le même que dans le cas de la conditionnelle incomplète est dû au fait que l'itération bornée peut être obtenue en composant une conditionnelle et une instruction de saut, telle que le **goto** : la sémantique de

while e **do** c **done**

est

label : **if** e **then** c ; **goto** label **endif**

Cependant, une instruction de saut explicite ne favorisant pas la programmation structurée, nous ne pourrons pas utiliser explicitement cette sé-

mantique. Son intérêt est de justifier la construction du graphe de contrôle de l'itération non bornée, un saut correspondant simplement à un arc.

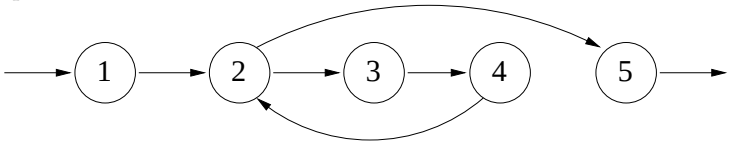
— **L'itération bornée.** L'itération bornée est un cas particulier de l'itération non bornée. Sa syntaxe est **for** <*variable*> ← <*expression entière*> **to** <*expression entière*> **do** <*instructions*> **done**. La sémantique de

for $i \leftarrow e_0$ **to** e_1 **do** c **done**

est

$i \leftarrow e_0$; **while** $i \leqslant e_1$ **do** c ; $i \leftarrow i + 1$ **done**

Donc, en étiquetant par 1 le nœud correspondant à $i \leftarrow e_0$, en étiquetant par 2 le nœud correspondant à $i \leqslant e_1$, en étiquetant par 3 le nœud correspondant à c, en étiquetant par 4 le nœud correspondant à $i \leftarrow i+1$ et en étiquetant par 5 le nœud correspondant au mot-clé **done**, on obtient le graphe de contrôle



Le nombre d'arcs est 5 et le nombre de nœuds est 5, donc le nombre cyclomatique de ce graphe est $\gamma = 2$. Il suffit néanmoins d'un seul chemin pour couvrir le graphe : $(1, 2, 3, 4, 2, 5)$.

1.3 Graphes réductibles

Il est possible de réduire les graphes de contrôle sans perdre d'information sur le flot de contrôle et en conservant le nombre cyclomatique. L'idée directrice est que la longueur des tests n'est pas importante. Pour réduire les graphes en accord avec ce principe, il faut considérer une règle simple :

où les deux nœuds sont des instructions simples. Ainsi une séquence d'instructions se réduit à un seul nœud, mais le graphe de l'itération bornée est n'est pas réductible à celui de l'itération non-bornée, car le nœud qui initialise l'index de la boucle ne doit pas être parcouru à chaque itération.

2 Preuves de programme et test

Avant d'aborder la thématique du test, faisons un détour par celle de la preuve de programme. Pour des programmes de petite taille il est possible d'effectuer des preuves mathématiques de certaines propriétés attendues, fondées sur le principe de l'induction structurelle (dans le cas des entiers naturels, c'est simplement le principe de récurrence que vous avez étudié dès le lycée). Il y a quelques outils d'aide à la preuve de programme, mais ils sont, en général, très difficiles à utiliser pour des ingénieurs. Les propriétés intéressantes sont presque toujours la terminaison, la correction et la complétude. La correction et la complétude sont des notions relatives : on les prouve par rapport à la spécification du programme. Une spécification peut être considérée comme étant

une abstraction du programme, une vue générale mais néanmoins rigoureuse. La correction dit que toutes les valeurs retournées par le programme sont bien prévues par la spécification, et la complétude dit que le programme n'en oublie aucune. Pour des raisons qui touchent aux fondements des mathématiques, toutes ces propriétés ne peuvent être démontrées en général de façon entièrement automatique (on parle d'indécidabilité), c'est-à-dire qu'il n'existe pas de programme universel qui prendrait en argument un programme et dirait (donc s'arrêterait) si celui-ci s'arrête.

La taille des programmes industriels rend impossible des preuves complètes. C'est pour ces raisons que le test du logiciel est absolument nécessaire. Dans certains cas très particuliers, comme certains protocoles de communication très simple, le test peut être équivalent à une preuve de correction (on parle alors de conformité), mais en pratique le test apporte donc simplement un degré de confiance lorsque la preuve logique ne peut se faire. Il se peut d'autre part qu'il existe plusieurs degrés de raffinement des spécifications (c'est-à-dire une suite de spécifications de plus en plus précises, dans le but de se rapprocher du programme final), et que les preuves ne se fassent qu'entre spécifications de degrés successifs. Dans ce cas, le test est toujours nécessaire pour dire quelque chose sur le comportement du programme. Un autre cas de figure est qu'il est possible de prouver quelque propriété sur une partie du programme (telle qu'une petite fonction) mais pas sur le reste. Là aussi, le test (du reste ou de l'ensemble) est fondamental.

Nous allons ici commencer par envisager des programmes si simples qu'il est possible d'effectuer à la fois des preuves et du test. Cela peut avoir du sens car nous ne considérons pas vraiment un programme exécutable, mais sa version épurée qu'est l'algorithme. En effet, un programme comportera presque toujours des interactions avec l'usager, avec le système de fichiers, avec des périphériques comme l'écran, et la preuve de programme ignore ces aspects-là, se concentrant sur l'algorithme. C'est pourquoi le test reste nécessaire sur le programme final, même si l'algorithme a été prouvé correct, car il peut y avoir des erreurs dans les interactions (dites *entrées-sorties*).

Un autre aspect très important de l'activité de programmation est la détermination de la complexité dynamique, par opposition à la complexité structurelle ou statique, dont le nombre cyclomatique peut être un indicateur. Cette complexité dynamique concerne le temps d'exécution du programme, ainsi que sa consommation de mémoire. Ces domaines ressortent plutôt de l'algorithmique, et, même si nous donnons parfois la complexité de certains algorithmes, nous n'élaborerons pas ici à leur propos.

3 Le tri par sélection

3.1 Principe

Le tri par sélection consiste à trouver l'emplacement de l'élément le plus petit du tableau $(a_1, \ldots, a_n)$, c'est-à-dire l'entier m tel que $a_i \geqslant a_m$ pour tout i. Ensuite on échange a_1 et a_m, puis on recommence avec le sous-tableau $(a_2, \ldots, a_n)$:

3.2 Algorithme

Nous utilisons notre pseudo-langage :

```
let tri_selection (a) =
  for i <- 1 to n-1 do
    m <- i;
    for j <- i+1 to n do
      if a[j] < a[m] then m <- j endif
    done;
    t <- a[i];
    a[i] <- a[m];
    a[m] <- t
  done
```

3.3 Terminaison

L'algorithme termine car il n'est constitué que d'itérations bornées (boucles `for`).

3.4 Correction

L'invariant de boucle à établir est qu'avant chaque itération les éléments $a_1, \ldots, a_{i-1}$ sont bien placés. En supposant que $a_0 = -\infty$, cette propriété est vraie pour $i = 1$ (i.e. avant la première itération). Supposons que la propriété soit vraie avant une itération quelconque. Alors a_m est l'élément minimal du sous-tableau restant $(a_i, \ldots, a_n)$. Par conséquent, après la permutation de a_i et a_m, le nouvel a_i est bien placé. Donc les éléments $a_1, \ldots, a_i$ sont bien placés avant l'itération suivante. Finalement, la boucle s'achève avec $i = n$ (i.e. dépassement de 1 de la borne supérieure). Donc les éléments $a_1, \ldots, a_{n-1}$ sont bien placés. Cela implique que a_n est bien placé, donc que le tableau $(a_1, \ldots, a_n)$ est trié.

3.5 Complexité temporelle

Le nombre de comparaisons est toujours :

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

La complexité temporelle du tri par sélection est quadratique par rapport à la taille du tableau.

3.6 Complexité spatiale

Le tri par sélection sur un tableau est un tri interne [1]. La complexité spatiale est donc proportionnelle à la taille du tableau : $\Theta(n)$.

1. Un tri interne est un tri qui n'autorise qu'une mémoire auxiliaire indépendante de la taille de la liste à trier.

3.7 Stabilité

Un tri est stable s'il conserve la position relative des éléments équivalents (ou égaux). Ici, s'il existe plusieurs éléments équivalents minimaux dans le sous-tableau $a_{i+1,n}$, c'est le premier qui sera sélectionné (a_m), et positionné. L'ordre relatif de deux éléments équivalents n'est donc pas modifié : le tri par sélection est stable.

4 Le tri par insertion

— **Principe.** Il consiste à insérer un élément dans un sous-tableau déjà trié à gauche (c'est le tri du joueur de cartes) :

— **Algorithme.**

```
let tri_insertion (a) =
  for j <- 2 to n do
    key <- a[j];
    i <- j - 1;
    while i > 0 and a[i] > key do
      a[i+1] <- a[i];
      i <- i - 1
    done;
    a[i+1] <- key
  done
```

— **Terminaison.** La terminaison de l'algorithme se ramène à celle de l'itération `while`. La fin de la boucle est assurée si $i = 0$ (dans le pire des cas). Or la variable i est positive et est décrémentée à chaque itération. Donc, l'algorithme termine.

— **Correction.** L'invariant de boucle à établir est qu'avant chaque itération (`for`), le sous-tableau $a_{1,j-1}$ est constitué des éléments originellement dans $a_{1,j-1}$ mais triés. Avant la première itération, $j = 2$, donc le sous-tableau en question est réduit à $a[1]$, qui est le $a[1]$ initial trivialement trié. Supposons la propriété vraie avant une itération quelconque. Avant l'itération suivante, on a $a_1 \leqslant \ldots \leqslant a_i \leqslant a_j \leqslant a_{i+1} \leqslant \ldots \leqslant a_{j-1}$. Donc, le nouveau sous-tableau $a_{1,j}$ est constitué des éléments initiaux mais triés. À la fin de la boucle, on a $j = n + 1$, donc le tableau $a_{1,n}$ est trié.

— **Complexité temporelle.** Lorsque le tableau est trié par ordre décroissant, nous sommes dans le pire des cas. Le nombre de comparaisons est alors :

$$\sum_{j=2}^{n} \sum_{i=1}^{j-1} 1 = \frac{n(n-1)}{2} = O(n^2)$$

Lorsque le tableau est déjà trié, nous sommes dans le meilleur des cas. Le nombre de comparaisons est alors :

$$\sum_{j=2}^{n} 1 = n - 1$$

— **Complexité spatiale**. Le tri par insertion sur un tableau est un tri interne. La complexité spatiale est donc proportionnelle à la taille du tableau : $\Theta(n)$.
— **Stabilité.** Un élément a_i n'est déplacé que si $a_i > key$ et $i < j$. Donc si $key = a_i$, alors a_i n'est pas déplacé, et key est inséré juste après a_i, conservant ainsi l'ordre relatif initial de ces deux éléments. Le tri par insertion est donc stable.

5 Questions

— Établissez les graphes de contrôle des programmes de tri. Calculez leur nombre cyclomatique.
— Réduisez-les graphes et recalculez leur nombre cyclomatique.
— Étiquetez les arcs par des lettres uniques.
— Trouvez un ensemble minimal de chemins recouvrants et faites-les correspondre à des données particulières.
— Assurez-vous que tous les chemins sont exécutables et que la couverture est complète.