AAGA : Devoir de programmation

Yassine Herida et Willy Chiv

Janvier 2020

Table des matières

1 Introduction **2**

2 Vérification d'algorithmes **2**
2.1 Ternary search tree 2
2.2 Algorithme de Rémy 5

3 Outils de vérification **6**

4 Bibliographie **7**

1 Introduction

QuickCheck est un outil permettant de vérifier et tester les propriétés des programmes. En cas d'erreur retrouvée, l'outil va développer de nouveaux tests plus ciblés pour obtenir un grain plus fin sur l'origine de l'erreur. L'outil est capable de générer aléatoirement des valeurs ou des cas de test afin de couvrir un maximum de cas qui n'auraient pas pu être couverts par les développeurs. Il reste possible de créer ses propres cas de tests, à partir de l'outil, pour cibler certains cas particuliers par exemple.

QuickCheck a été à l'origine développé pour faire des tests de propriétés sur Haskell. Cependant des variantes de celui-ci sont apparues dans d'autres langages de programmation comme :

— QuickTheories (Java)
— Hypothesis (Pyhton)
— Theft (C)
— Test.check (Clojure)
— Jsverify (Javascript)

2 Vérification d'algorithmes

2.1 Ternary search tree

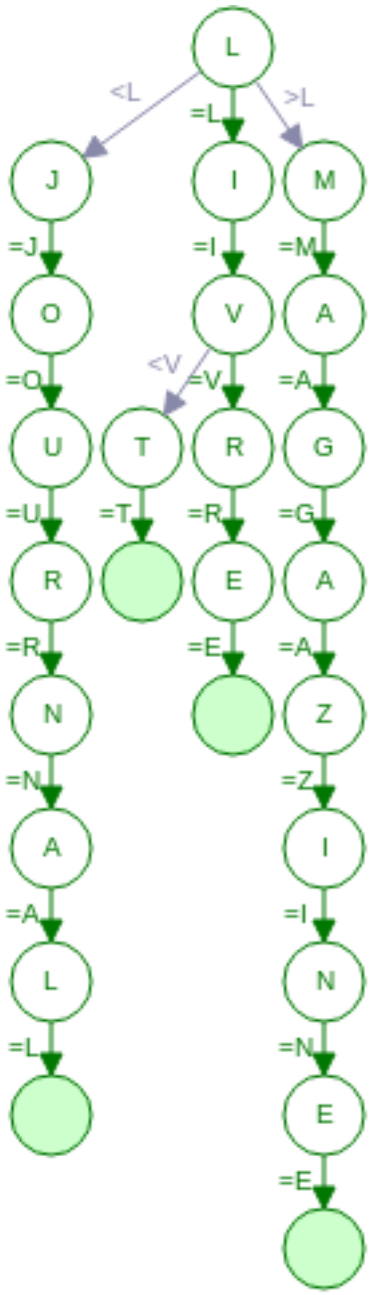
Les arbres ternaires de recherche sont une structure de données très semblables aux arbres binaires de recherche où les noeuds possèdent chacun jusqu'à 3 fils au lieu de 2 fils.
Cette structure d'arbre peut être décrite de cette façon :
T = Z + Z * T + Z * (T * T) + Z * (T * T * T)

Dans notre cas, chaque noeud de l'arbre contient comme valeur une lettre de l'alphabet et pointe vers 3 noeuds fils :

— middle child qui représente la lettre suivante dans le mot
— lower child qui possède une lettre avec une valeur plus faible
— higher child qui possède une lettre avec une valeur plus élevée

Les opérations de recherche, d'insertion et de suppression se font en moyenne en temps O(log n). Dans les pires cas, nous sommes dans une complexité O(n).

Exemple :



Arbre ternaire de recherche construit à partir des mots : “livre”, “lit”, “journal” et “magazine”

Pour construire un arbre ternaire de recherche à partir d’un fichier, nous allons récupérer tous les mots du texte dans un tableau (doublons inclus) puis tirer de façon uniforme dans ce tableau, le bon nombre de mots.

Algorithm 1 BuildTreeBook(fichier : nom du fichier , nb : nombre de mots)

```
contenu_fichier := lecture_fichier(fichier).split()
arbre := generer_feuille()
for i := 0 to nb do
  if len(contenu_fichier) == 0 then

    return arbre
  end if
  n := GenRandomInt(0, len(contenu_fichier) − 1)
  buff := MotToArbre(contenu_fichier[n])
  arbre := fusion(arbre, buff)
end for
return arbre
```

Remarque :
On peut remplacer la fusion par un ajout successif sur l'arbre, nous avons codé les deux versions, ici nous mettons la version avec fusion (même si le code de la fusion ne fonctionne pas) parce que nous en aurons besoin plus tard pour nos tests.
Afin de retrouver l'erreur, nous avons fait des recherches de mots. Pour cela :

- On part de l'arbre vide A et d'un tableau de mots
- On prend un mot, on construit l'arbre B correspondant à ce mot
- On fusionne A et B puis on recommence jusqu'à qu'on ait inséré tous les mots
- Enfin on cherche les mots un par un dans l'arbre ternaire de recherche

Algorithm 2 Detection(mots : tableau de mots , nb : taille de mots)

```
c := buildTreeBookFusion(mots, nb)
mots2 := []
for m in mots do
  if Get(m, c) then
    mots2.append(m)
  end if
end for
return len(mots) == len(mots2)
```

Après expérimentation, on a pu voir que certain mots disparaissaient. La fusion n'est pas totalement faite en profondeur, en effet quand on compare les clés, la fusion en profondeur est faite uniquement que d'un coté.
Par exemple, si on fusionne deux arbres A et B où la clé de la racine de B est plus grande que celle de A mais que le fils gauche de B est plus petit que la racine de A, celle-ci sera perdu dans la fusion.

Exemple :
Après expérimentation sur mots = ['o', 'brings', 'i'] , on ne trouve plus le mot 'o'.

2.2 Algorithme de Rémy

Afin de déterminiser l'algorithme d'arbre binaire de Rémy, nous avons passé en paramètre une liste de valeurs qui représente le tirage des différents noeuds à étendre. Pour déterminiser notre impémentation de Remy, nous avons dû rajouter le pile ou face qui choisit le noeud qu'on à tirer est soit un fils gauche soit un fils droit. Pour construire cette liste :

- On commence de la liste de listes res qui ne contient que [0], au début on ne peut tirer qu'un seul noeud et i=1
- Puis pour chaque liste l de res :
- Pour tous les j de 0 à i+1 : on rajoute à une copie de l buff l, la valeur j, buff.append(j) et on ajoute buff à buffres
- Puis on lance l'algorithme récursivement avec i=i+1 et res = buffres jusqu'à que i soit égal à n

Pour notre version de l'algorithme de Rémy, j va de 0 à 2i+1 et on rajoute entre chaque valeur de tirage du noeud, la valeur du pile ou face. On a donc pour chaque buff 2 versions, une version où on rajoute 0, une autre où on rajoute 1.

Pour vérifier si deux arbres sont identiques, nous les transformons en chaîne de caractères puis on les compare.

Pour construire nos tests de couverture, nous avons d'abord généré toutes les listes de valeurs possibles de tirage (cf. alllist) pour une taille donnée. Nous stockons ensuite dans un dictionnaire toutes les structures d'arbre, sous forme de chaîne de caractères, qui ont été générées à partir des différentes listes de valeurs avec l'occurrence d'apparitions de la structure.
Dans cette chaîne de caractères, qui est une représentation de l'arbre, nous n'affichons que les fils gauches pour obtenir une lecture simplifiée.

Le test d'uniformité sur le Rémy à implémenter :

```
taille 2 : {'(())': 6}
taille 3 : {'(())()': 12, '(()())': 12}
taille 4 : {'(()())()': 60, '(())()()': 20, '((()))()': 20, '(()()())': 20}
```

A partir des arbres de taille 4, on peut déjà observer une apparition non-uniforme des arbres.

Le test sur notre version de l'algorithme de Rémy :

```
2
taille 0 : {'': 2}
12
taille 1 : {'()': 12}
120
taille 2 : {'()()': 60, '(())': 60}
1680
taille 3 : {'()()()': 336, '(())()': 336, '()(())': 336, '(()())': 336, '((()))': 336}
30240
taille 4 : {'()()()()': 2160, '(())()()': 2160, '()(())()': 2160, '()()(())': 2160, '()(
)(())': 2160, '(())(())': 2160, '((()))()': 2160, '(())(())': 2160, '((()))()': 2160, '(()()())': 2160, '(
()((()))': 2160, '()((()))': 2160, '(((())))': 2160}
665280
taille 5 : {'()()()()()': 15840, '(())()()()': 15840, '()(())()()': 15840, '()()(())()': 15840, '()()()(
)': 15840, '()()(())()': 15840, '()()()(())': 15840, '(())(())()': 15840, '((()))()()
': 15840, '(())()(())': 15840, '(())()()()': 15840, '()(())()()': 15840, '((()))()()': 15840, '(()(())())'
: 15840, '(()()(()))': 15840, '(()()()())': 15840, '(()()(()))': 15840, '()(()())()': 15840, '()((()))()':
15840, '()(())(())': 15840, '()(()())()': 15840, '()((()))()': 15840, '()(()()())': 15840, '()(())()()': 1
5840, '()()((()))': 15840, '((()))()()': 15840, '(()())(())': 15840, '((()))()()': 15840, '((()()))()': 15
840, '(()(()))()': 15840, '((()))(())': 15840, '((()))(())': 15840, '((()))()()': 15840, '(())((()))': 158
40, '((()))(())': 15840, '(((())))()': 15840, '((()(())))': 15840, '(((()(())))': 15840, '(()((()))())': 1584
0, '()(((())))': 15840, '((((()))))': 15840}
```

A partir de la taille 6, il n'est plus possible d'effectuer des vérifications parce que la complexité de la génération des tirage est en temps exponentiel.

3 Outils de vérification

Les tas binaires min sont une structure de données qui possèdent la même structure que les arbres binaires parfaits : tous les noeuds du tas possèdent chacun exactement 2 fils, excepté ceux qui sont à hauteur h-1. Dans ce cas, le tas doit être rempli de la gauche vers la droite.

L'étiquette de chaque noeud doit être aussi inférieur ou égale à celles de leurs noeuds fils. Le parcours d'une branche depuis la racine doit donc donner des étiquettes de valeur croissante.

Les tas binaires peuvent être aussi représentés sous la forme d'un tableau de la manière suivante :

- La racine du tas est située à l'indice 0 du tableau
- Soit n un noeud situé à l'indice i, son fils gauche est lui à l'indice 2i+1 et le fils droit à l'indice 2i+2
- Soit n un noeud situé à l'indice i > 0, son père est situé à l'indice (i - 1)/2 (arrondi à l'entier inférieur)

Pour effectuer une insertion dans un tas binaire min, on ajoute l'élément à insérer à la prochaine position libre (qui est soit la position la plus à gauche possible du dernier niveau de l'arbre soit sur un nouveau niveau).

Après cela, on remonte ce nouveau noeud dans l'arbre en l'échangeant successivement avec son père ou son nouveau père en veillant à respecter que son étiquette reste supérieure à son père.

Dans le cas de la suppression, l'élément à la racine est supprimé et le noeud en dernière position est placé à la racine. Ensuite on effectue des échanges de la même

manière que dans l'insertion entre ce noeud et son fils ou ses nouveaux fils tout en conservant la propriété d'ordre des étiquettes.

Étant donné que seuls les noeuds d'une branche sont impliqués dans les opérations d'insertion et de suppression, la complexité de ces opération sont donc en O(log n).

Pour faire le test de notre fonction d'insertion et de suppression, nous allons construire à partir d'une liste d'entiers , un tas avec la fonction d'ajout, puis nous allons pop la plus petit valeur et la stocker dans un tableau.
Cela est sensé nous donner une version triée de la liste d'entrées.

L'outil de vérification automatique Hypothesis sur Phyton va nous permettre de tester en générant plusieurs listes d'entiers aléatoires afin de vérifier notre programme.

Pour vérifier l'algorithme de fusion des arbres ternaires de recherche, nous avons repris le même code et généré aléatoirement des entrées pour vérifier que la fusion ne se fait pas correctement.

4 Bibliographie

QuickCheck : A Lightweight Tool for Random Testing of Haskell Programs.
https ://www.cs.tufts.edu/ nr/cs257/archive/john-hughes/quick.pdf

QuickCheck in every language.
https ://hypothesis.works/articles/quickcheck-in-every-language/

Ternary Search Tree Visualization
https ://www.cs.usfca.edu/ galles/visualization/TST.html