SORBONNE UNIVERSITÉ

Rapport de Projet AAGA: Sujet 2

Wenzhuo ZHAO, Edwin ANSARI, Chengyu YANG

M2 STL

wenzhuo.zhao@etu.sorbonne-universite.fr,
edwin.ansari_tabrizi@etu.sorbonne-universite.fr,
cheng-yu.yang@etu.sorbonne-universite.fr

January 27, 2022

Contents

1 Mise en route: test de propriété

De nombreux travaux de recherche ont été menés sur le test logiciel, aussi bien afin de donner des fondements mathématiques que pour automatiser la procédure de test. Dans cette partie, nous nous intéressons à des outils permettant de faire du test de propriété et leurs principes sur le générateur de données à utiliser pour le test.

1.1 junit-quickcheck pour Java

junit-quickcheck [1] est une bibliothèque qui permet d'écrire et d'exécuter des tests basés sur les propriétés dans JUnit, inspirée de QuickCheck pour Haskell. En ajoutant l'annotation **@Property** sur l'en-tête d'une fonction Java, chaque propriété sera évaluée 100 fois avec des valeurs aléatoires générées par un générateur prédéfini sur des types primitives ou des générateurs définis par l'utilisateur.
Lorsque plusieurs générateurs peuvent satisfaire des contraintes sur un paramètre de propriété donné en fonction de son type (par exemple, **java.io.Serializable**), pour une génération, junit-quickcheck choisira l'un des multiples générateurs au hasard avec une probabilité égale.
Pour chaque paramètre de propriété, junit-quickcheck utilise une valeur unique comme graine pour la source d'aléatoire utilisée pour générer les valeurs du paramètre. Il est donc aussi possible de fixer la graine par l'utilisateur.
Par défaut, junit-quickcheck vérifie une propriété en mode échantillonnage - il génère 100 groupes de valeurs aléatoires pour la liste de paramètres d'une propriété, et vérifie la propriété avec les 100 groupes de valeurs. junit-quickcheck peut également vérifier une propriété en mode "exhaustif". En mode "exhaustif", junit-quickcheck génère des valeurs aléatoires pour chacun des paramètres de la propriété, et vérifie la propriété avec chaque membre du produit cartésien des ensembles de valeurs aléatoires pour chaque paramètre.

1.2 Coverage Guided, Property Based Testing: une extension de QuickCheck pour Coq

La plupart des entrées générées aléatoirement ne parviennent pas à satisfaire les propriétés avec des préconditions aussi peu nombreuses, et sont donc tout simplement rejetées. Ce problème est abordé avec une nouvelle technique appelée CGPT (coverage guided, property based testing) [2].Plutôt que de générer une nouvelle valeur aléatoire à chaque itération de test, CGPT peut également produire de nouvelles entrées en mutant les précédentes à l'aide d'opérateurs de mutation génériques et sensibles au type.
Le CGPT est implémenté à la base de QuickChick [3] qui est quasiment un clone de QuickCheck de Haskell, Crowbar et QcCrowbar qui servent à tester les propriétés dont les préconditions sont éparses.
Le générateur est soit un générateur "purement" aléatoire, soit on choisit une mutation à appliquer à une graine précédemment découverte à utiliser pour la génération.

2 Vérification d'algorithmes

2.1 Ternary Search Trie

Un arbre de recherche ternaire [4] est une structure de données trie spéciale où les nœuds enfants d'un trie standard sont ordonnés comme un arbre de recherche binaire.
Pour composer un mot, une représentation des arbres de recherche ternaires est comme: Contrairement à la structure de données trie(standard) où chaque nœud contient 26 pointeurs pour ses enfants, chaque nœud d'un arbre de recherche ternaire ne contient que 3 fils :
1. Le fils gauche possède la valeur inférieure à la valeur du noeud actuel.
2. Le fils droit possède la valeur est supérieure à la valeur du nœud actuel.
3. Le fils au milieu possède la valeur qui est la prochaine lettre du mot préfixé par les valeurs sur le chemin de noeuds jusqu'à où le noeud est un fils gauche ou droit de son parent.
En dehors des trois fils ci-dessus, chaque nœud possède un autre champ pour marquer la fin d'une chaîne.
Ainsi, il est plus ou moins similaire au BST(Binary Search Tree) qui stocke les données en fonction d'un certain ordre. Cependant, dans un arbre de recherche ternaire, les données sont réparties sur les nœuds.
L'arbre ternaire peut être utilisé pour effectuer des recherches rapides dans un dictionnaire [5], par exemple, une table de routage d'URL. Les adresses URL ont souvent un préfixe commun, puis elles se ramifient. Nous pouvons aussi nous inspirer son implémentation [6] en Java.

2.1.1 Construction d'un arbre ternaire par des mots

Prenons un fichier contenant N mots et un entier nb, nous choisissons uniformément nb mots pour construire un arbre ternaire.
Nous utilisons le générateur d'entier **rand_int** vu en cours pour générer les nb indices dans l'intervalle [0;N] du tableau de mots pour choisir des mots uniformément.

Algorithm 1: extrait *nb* mots pour construction d'un arbre ternaire

Data: Collection<String> *mots*, int *nb*
Result: ArbreTernaire *arbre*
ArbreTernaire *arbre* = gener_feuille();
while *nb != 0* **do**
    index = rand_int(N);
    mot = *mots*.get(*index*);
    arbre = insert(*arbre*, *mot*);
    mots.remove(*mot*);
    nb = *nb* - 1 ;
return *arbre* ;

2.1.2 Observer le bug de fonction fusion

Pour fusionner 2 arbres ternaires en un seul, le principe de la fonction **fusion(arbreA, arbreB)** [7] est de fusionner récursivement la clé et les clés de fils de chacun en respectant que l'arbre soit un arbre de recherche. Cependant, il est possible que ce nouvel arbre peut perdre des mots des 2 arbres anciens, ou créer de nouveaux mots n'ayant aucun sens. Pour observer un tel bug, notre stratégie est d'essayer de trouver tous les mots qui existent dans les 2 arbres anciens dans l'arbre fusionné. Nous pouvons observer que le bug est présent s'il existe un mot introuvable dans l'arbre fusionné.
Définissons la fonction **get_mots(arbre, prefixe)**(dans Annexe A) qui prend un préfixe de mots quelconques et renvoie tous les mots préfixé dans l'arbre, alors si le préfixe est vide, la fonction renvoie tous les mots dans l'arbre. Une autre fonction **search(arbre, mot)**(dans Annexe B) permet de donner l'existence d'un mot dans l'arbre. Le code en Python de ces 2 fonctions se trouvent dans l'annexe.
Maintenant nous nous intéressons à déterminer si le résultat de **fusion** est valide donc sans bug. Fusionner l'arbre a et b en un nouvel arbre X, prendre l'ensemble m de mots dans a et b. Pour chaque mot dans m, nous utilisons **search** pour déterminer son existence dans l'arbre X: si le mot n'est présent dans X, nous pouvons dire que la fonction **fusion** a un bug et arrêter l'exécution de l'algorithme.

Algorithm 2: déterminer si l'arbre fusionné par arbre a et b est valide

```
Data: ArbreTernaire a, ArbreTernaire b
Result: Boolean valide
ArbreTernaire X = fusion(a, b);
Collection<String> mots = get_mots(a, "").union(get_mots(b, "")) ;
foreach String mot in mots do
    if not search(X, mot) then
        return False ;
return True ;
```

2.2 Arbre binaire de Rémy

2.2.1 Implémentation d'une version de l'algorithme de Rémy

Nous implémentons le code fourni dans l'annexe en Python qui se trouve dans l'annexe C de ce rapport. Nous avons défini la classe **Node** pour enregistrer les informations sur les noeuds de fils, le noeud de parent et la valeur sur un noeud, ensuite nous implémentons les fonctions **change_leaves** et **growing_tree** de manière similaire au code fourni dans la classe **RemyTree**.

2.2.2 Rendre l'algorithme déterministe

Pour rendre l'algorithme déterministe, nous remplaçons les choix aléatoires d'indice de noeud **number = random(i)** par les valeurs générées au préalable. Pour les choix d'indice à chaque étape **i**, nous pouvons déduire que $index_i \in [0, i-1]$ et $i \in [2, n]$ du code donné dans l'annexe. Pour générer les listes de valeurs pour l'arbre rémy de taille **n** dont $n >= 2$, il existe $2 * 3 * 4 * ... * (n-1) = (n-1)!$ possible listes de valeurs.

2.2.3 Égalité de deux arbres

Nous pouvons savoir si deux arbres non étiquetés sont identiques en comparant leur compression en chaîne de caractères comme la manière décrite dans le sujet. Nous définissons donc l'algorithme **compresser(arbre)** comme ci-dessous qui renvoie la compression en chaîne de caractères.

Algorithm 3: Compresser un arbre

```
Data: ArbreBinaire arbre
Result: String compression
if arbre.est_noeud() then
    return ε;
else
    return "(" + compresser(arbre.fils_gauche) + ")" +
     compresser(arbre.fils_droit)
```

Ensuite il est tout simplement de comparer l'égalité des chaînes de caractères renvoyées par cet algorithmes de différents arbres.

2.2.4 Test de couverture de l'implémentation incorrecte

Nous générons des listes de valeurs de la manière décrite dans la partie 2.2.2 de taille **n** dont $n \in [0, 9]$. Nous fournissons les listes possibles de valeurs à chaque taille pour générer des arbres rémy, puis nous vérifions si les occurrences de différentes structures d'arbre est équivalentes pour chaque différente liste de valeurs à la même taille. Si toutes les occurrences sont équivalentes pour une même taille, nous pouvons déterminer que l'algorithme de génération d'arbres rémy est une génération uniforme. Le résultat de test s'accorde avec l'affirmation dans le sujet qui affirme que cet algorithme ne génère pas d'arbres rémy de manière uniforme.

2.2.5 Test de couverture de notre implémentation de l'algorithme de Rémy

Une implémentation correcte est donnée à la partie 4 dans l'article [8] mentionné dans le sujet. En suivant le principe de la présentation, nous avons intégré notre code de cette implémentation en version aléatoire avec des variables aléatoires et en version déterministe avec une liste de valeurs comme dans les questions précédentes.
Comme la question précédente, pour générer toutes les listes de valeurs comme entrée de ce test, il faut créer un pair de valeurs **(index, direction)** dont l'index est l'indice du noeud qui sera remplacé par un nouveau noeud et la direction est le choix de fils gauche ou droite de ce nouveau noeud avec une probabilité 1/2.

Dans la version aléatoire de cet algorithme de Rémy, à chaque étape **i** de boucle, l'index est généré entre [0, i-1] et l'étape **i** incrémente de 2 à chaque fois de boucle. Alors pour générer l'index dans une liste de valeurs, le maximum des valeurs de l'index à étape **i** est inférieur de 2 au maximum des valeurs de l'index à étape **i+1**, qui est $max(index_i) + 2 = max(index_{i+1})$.
Voici un exemple: nous souhaitons générer des listes de **(index, direction)** pour l'arbre rémy de taille 3 comme $[(index_0,\ direction),\ (index_1,\ direction),\ (index_2,\ direction)]$, dont $index_0$ est dans [0, 0], $index_1$ est dans [0, 2] et $index_2$ est [0, 4]. La direction est soit 0 soit 1. Il existe 1 * 2 * 3 * 2 * 5 * 2 = 120 possibilités de listes de valeurs. Nous pouvons en déduire qu'il existe 1 * 2 * 3 * 2 * 5 * 2 * ... * (2n-1) * 2 = $(2n)!/(n)!$ possibles listes de valeurs.
En faisant le test de couverture comme la question précédente, nous avons trouvé que notre algorithme génère des arbres rémy de manière uniforme.

3 Utilisation d'un outil

4 Conclusion

Appendices

Appendix A Fonction get_mots(arbre, prefixe)

```python
def get_mots(arbre, prefx):
    """
    A partir d'un prefixe, traverser tous les chemins de l'arbre puis
        composer le mot par le prefixe et les lettres
    Args:
        prefx: le prefixe du mot

    Returns:
        ensemble de mots prefixes trouves dans l'arbre
    """
    words = set()

    if arbre.cle == '':
        return words
    if arbre.val == 0:
        words.add(prefx + arbre.cle)

    words = words.union(arbre.fils[0].get_mots(prefx))
    words = words.union(arbre.fils[1].get_mots(prefx + arbre.cle))
```

```
        words = words.union(arbre.fils[2].get_mots(prefx))

        return words
```

Appendix B Fonction search(arbre, mot)

```
    def search(A, mot):
        """
        Chercher le mot donne dans l'arbre A et renvoie True si il y est
        present, False sinon
        A : arbre
        mot : mot a chercher
        """

        if mot == '':
            return False
        elif len(A.fils) == 0:
            return False
        elif mot[0] < A.cle:
            return search(A.fils[0], mot)
        elif mot[0] > A.cle:
            return search(A.fils[2], mot)
        elif len(mot) == 1:
            return True if mot[0] == A.cle and A.val == 0 else False
        return search(A.fils[1], mot[1:])
```

Appendix C Algorithme de Rémy présentée dans le livre de Alonso et Schott

```
class Node:

    def __init__(self, num):
        self.left_child = -1
        self.right_child = -1
        self.parent = -1
        self.num = num

    def __str__(self):
        return "num : " + str(self.num) + " parent: " + str(self.parent) + "
            left_child : " + str(self.left_child) + " right_child : " +
            str(self.right_child)
```

```
    def __repr__(self):
        return "num : " + str(self.num) + " parent: " + str(self.parent) + "
            left_child : " + str(self.left_child) + " right_child : " +
            str(self.right_child)

    def __eq__(self, other):
        if not isinstance(other, Node):
            return False
        return self.num == other.num and self.parent == other.parent and
            self.left_child == other.left_child and self.right_child ==
            other.right_child

    def is_leaf(self):
        return self.left_child == -1 and self.right_child == -1


class RemyTree:
    tree = None

    def __init__(self, N):
        random.seed()
        self.tree = [Node(i) for i in range(2 * N + 1)]

    def __str__(self):
        return str(self.tree)

    def __eq__(self, other):
        if not isinstance(other, RemyTree):
            return False
        return self.tree == other.tree

    def change_leaves(self, a, b):
        parentA = self.tree[a].parent
        parentB = self.tree[b].parent
        if self.tree[parentA].right_child == a:
            self.tree[parentA].right_child = b
        else:
            self.tree[parentA].left_child = b
        self.tree[a].parent = parentB
        if self.tree[parentB].right_child == b:
            self.tree[parentB].right_child = a
        else:
            self.tree[parentB].left_child = a
        self.tree[b].parent = parentA

    def growing_tree(self, n):
        if n == 0:
            return self
```

```
        self.tree[0].left_child = 1
        self.tree[0].right_child = 2
        self.tree[0].num = 0
        self.tree[1].parent = self.tree[2].parent = 0
        self.tree[1].right_child = self.tree[1].left_child = -1
        self.tree[2].right_child = self.tree[2].left_child = -1

        for i in range(2, n + 1):
            nb = random.randint(0, i - 1)
            self.change_leaves(i - 1, nb + i - 1)
            self.tree[i - 1].right_child = 2 * i - 1
            self.tree[i - 1].left_child = 2 * i
            self.tree[i - 1].num = i - 1
            self.tree[2 * i - 1].parent = self.tree[2 * i].parent = i - 1
            self.tree[2 * i - 1].right_child = self.tree[2 *
                                                  i - 1].left_child = -1
            self.tree[2 * i].right_child = self.tree[2 * i].left_child = -1

    def growing_tree_det(self, n, l):
        if n == 0:
            return self

        self.tree[0].left_child = 1
        self.tree[0].right_child = 2
        self.tree[0].num = 0
        self.tree[1].parent = self.tree[2].parent = 0
        self.tree[1].right_child = self.tree[1].left_child = -1
        self.tree[2].right_child = self.tree[2].left_child = -1

        for i, nb in zip(range(2, n + 1), l):
            self.change_leaves(i - 1, nb + i - 1)
            self.tree[i - 1].right_child = 2 * i - 1
            self.tree[i - 1].left_child = 2 * i
            self.tree[i - 1].num = i - 1
            self.tree[2 * i - 1].parent = self.tree[2 * i].parent = i - 1
            self.tree[2 * i - 1].right_child = self.tree[2 *
                                                  i - 1].left_child = -1
            self.tree[2 * i].right_child = self.tree[2 * i].left_child = -1

        return self
```

References

[1] P. Holser, "junit-quickcheck: Property-based testing, junit-style." https://github.com/pholser/junit-quickcheck.

[2] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proc. ACM Program. Lang.*, vol. 3, oct 2019.

[3] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L.-y. Xia, "Beginner's luck: A language for property-based generators," *SIGPLAN Not.*, vol. 52, p. 114–129, jan 2017.

[4] "Ternary search trees." `https://iq.opengenus.org/ternary-search-tree/`.

[5] "Ternary search tree." `https://lukaszwrobel.pl/blog/ternary-search-tree/`.

[6] "Ternary search tree: Core methods (java implementation)." `https://dev.to/mlarocca/ternary-search-tree-core-methods-java-implementation-2hlj`.

[7] "Fusion incorrecte de 2 arbres ternaire." `https://github.com/valeeraZ/Sorbonne_AAGA/blob/master/Project/ternary_trie.py`.

[8] E. Mäkinen and J. Siltaneva, "A Note on Rémy's Algorithm for Generating Random Binary Trees," *Missouri Journal of Mathematical Sciences*, vol. 15, no. 2, pp. 103 – 109, 2003.