Université de Lille 1 SCIENCES ET TECHNOLOGIES

Crash Banducket

Un outil de rangement de StackTrace dans des Buckets

Lucas Delvallet et Mickaël Alvarez
OPL - Projet 2

Master 2 IAGL - Université de Lille 1
Encadrant : Martin MONPERRUS

Le 08/11/2016

Table des matières

Introduction

Dans le cadre du cours d'OPL[1], de la formation de Master 2 IAGL[2], nous avons réalisé un projet qui est un système de rangement de StackTrace[3] dans des Buckets[4].

Contexte

De nos jours, la plupart des logiciels[5] possèdent des systèmes de report de crash. Ceux-ci envoient généralement des données concernant le moment du crash ainsi que l'environnement sur lequel tourne l'application, on appellera ces reports StackTrace. Cependant, sur certain logiciel, ces crash sont légions et arrivent par milliers, pour un humain, classer ces crash par catégories ou similitude demande beaucoup de temps. Une catégorie de StackTrace est rangé dans un Bucket, chaque bug de nature différente sera donc rangé dans un Bucket différent.

Il nous faudrait donc un système automatique de rangement de StackTrace dans des Bucket. La solution devra être capable d'évaluer le Bucket dans lequel un StackTrace devra être rangé en analysant les StackTrace déjà catégorisé. Afin d'évaluer le système, deux jeu de donnée contenant des Bucket et des StackTrace rangé et non rangé nous sont fournit, ces StackTrace possèdent déjà une solution afin de vérifier la validité de notre solution. Notre programme devra être capable de retrouver ces solutions.

[1] http://portail.fil.univ-lille1.fr/portail/index.php?dipl=MInfo&sem=IAGL&ue=OPL&label=Pr%C3%A9sentation
[2] http://portail.fil.univ-lille1.fr/portail/index.php?dipl=MInfo&sem=IAGL&ue=ACCUEIL&label=Pr%C3%A9sentation
[3] https://fr.wikipedia.org/wiki/Trace_d%27appels
[4] https://en.wikipedia.org/wiki/Bucket_(computing)
[5] https://fr.wikipedia.org/wiki/Logiciel

Aspect technique

Le projet fut abordé en Java[6] en raison de la familiarité du binôme travaillant dessus. Nous avons temporairement hébergé le projet en privé sur BitBucket[7] dans un esprit de compétition, afin d'éviter les yeux baladeurs. A la fin du projet, nous avons importé le projet sous Github[8]. Cette partie présente les approches utilisées et présente l'architecture du projet.

Approches utilisée

Nous avons étudié plusieurs algorithmes afin de déterminer le meilleur moyen de réaliser une solution. Voici plusieurs d'entre elles présentée en détail :

Distance de Levenshtein[9]

Pour un premier algorithme, nous avons utilisé une méthode nommée « Distance de Levenshtein », celle-ci permet de comparer deux chaines de caractère et d'en sortir un pourcentage de ressemblance.

Elle consiste en la comparaison de chacun des caractères d'une chaîne de caractères et établit une distance entre les positions de deux caractères similaires. Cette étape est répétée pour chacun des caractères. A la fin, un pourcentage est établi en fonction de la longueur des chaînes de caractères qui sont comparée.

Exemple :

Chaine de caractère 1	Chaine de caractère 2	Pourcentage de ressemblance
OPL est une bonne matière.	OPL est une très bonne matière.	83.9%
Fantastique	Moustique	54.5%
Blanc	Noir	0%

[6] https://www.java.com/fr/about/whatis_java.jsp
[7] https://www.atlassian.com/software/bitbucket
[8] https://github.com/
[9] https://fr.wikipedia.org/wiki/Distance_de_Levenshtein

Version naïve[10]

Nous avons choisi, pour une première version, de récupérer chaque Frame d'un StackTrace en une chaîne de caractère et de comparer chacun des StackFrame[11] du StackTrace à ranger, avec chacun des StackTrace de chacun des Buckets. Avec la méthode de la distance de Levenshtein, nous obtenons un pourcentage de ressemblance pour chaque comparaison avec lesquelles nous calculons une moyenne de ressemblance pour StackTrace et enfin pour chaque Bucket. Nous rangeons alors le StackTrace dans le Bucket ayant le pourcentage de ressemblance le plus élevé. Cette méthode est trop complexe d'un point de vue algorithmique pour être utilisé sur des jeux de données conséquent.

Version évoluée

La première version n'étant pas assez performante, nous avons choisi d'améliorer notre approche. Nous avons donc perfectionné plusieurs points important :

Plutôt que de comparer une chaîne représentant un StackFrame complet, nous utilisons un parser[12] pour récupérer les valeurs importantes. Ainsi, nous obtenons une adresse mémoire[13], un nom de méthode[14], les arguments de la méthode[15], un nom de fichier avec son chemin et les variables[16] utilisée dans la méthode en elle-même. Chacun de ces champs peut-être vide ou nul. Après récupération de ces informations, nous utilisons la méthode de Levenshtein sur chacun de ces champs et calculons une moyenne de pourcentage de ressemblance.

Aussi, nous avons choisi de trouver de comparer uniquement les StackFrame d'index zéro car ceux-ci représente le dernier appel de la pile et nous paraît être plus pertinent.

[10] http://wcipeg.com/wiki/Naive_algorithm
[11] https://en.wikipedia.org/wiki/Call_stack#Structure
[12] https://en.wikipedia.org/wiki/Parsing
[13] https://en.wikipedia.org/wiki/Memory_address
[14] https://fr.wikipedia.org/wiki/M%C3%A9thode_(informatique)
[15] https://fr.wikipedia.org/wiki/Param%C3%A8tre
[16] https://fr.wikipedia.org/wiki/Variable_(informatique)

Comparaisons de méthodes.

Une seconde approche est de nous concentrer uniquement sur les méthodes de chaque StackFrame. Nous créons deux tableaux[17] contenant les méthodes de chaque StackFrame d'une StackTrace, ensuite nous comparons chacune des méthodes du premier tableau avec la méthode d'index correspondante de l'autre tableau. Ensuite nous établissons un pourcentage de ressemblance avec un simple calcul afin de pouvoir conserver nos méthodes de détermination du meilleur bucket.

Exemple :

StackFrame	StackTrace 1 : Méthode	StackTrace 2 : Méthode	Même méthode ?
#1	`getFile`	`getFile`	oui
#2	`listFileInFolder`	`loadSave`	non
#3	`??`	`??`	oui
#4	`??`	`main`	non
#5	`main`	`(aucune)`	non

Nous somme parti du principe que les méthodes qui n'ont pas pu être déterminée, sont identiques. Ici le résultat est de 40% de ressemblance entre les deux StackTrace. Enfin, il nous suffit de sélectionner le Bucket dans lequel notre StackTrace est le plus similaire aux StackTrace qu'il contient.

Attributions de points

Cette méthode part du principe que certains éléments sont plus importants que d'autre, mais aussi que leur positionnement dans la pile est significatif.

Notre procédure est telle que nous comparons les StackFrame de même position entre eux et attribuons des points en fonction des éléments identiques. Ainsi le nom de la méthode, le chemin du fichier, l'adresse mémoire, les arguments de la méthode et les variables donnent respectivement de 5 à 1 point par palier de 1. De plus ces points sont divisés par la position plus un du StackFrame dans le StackTrace. Cela à pour effet de favoriser les StackFrame en haut de pile car ils sont toujours plus précis car plus proche de l'erreur.

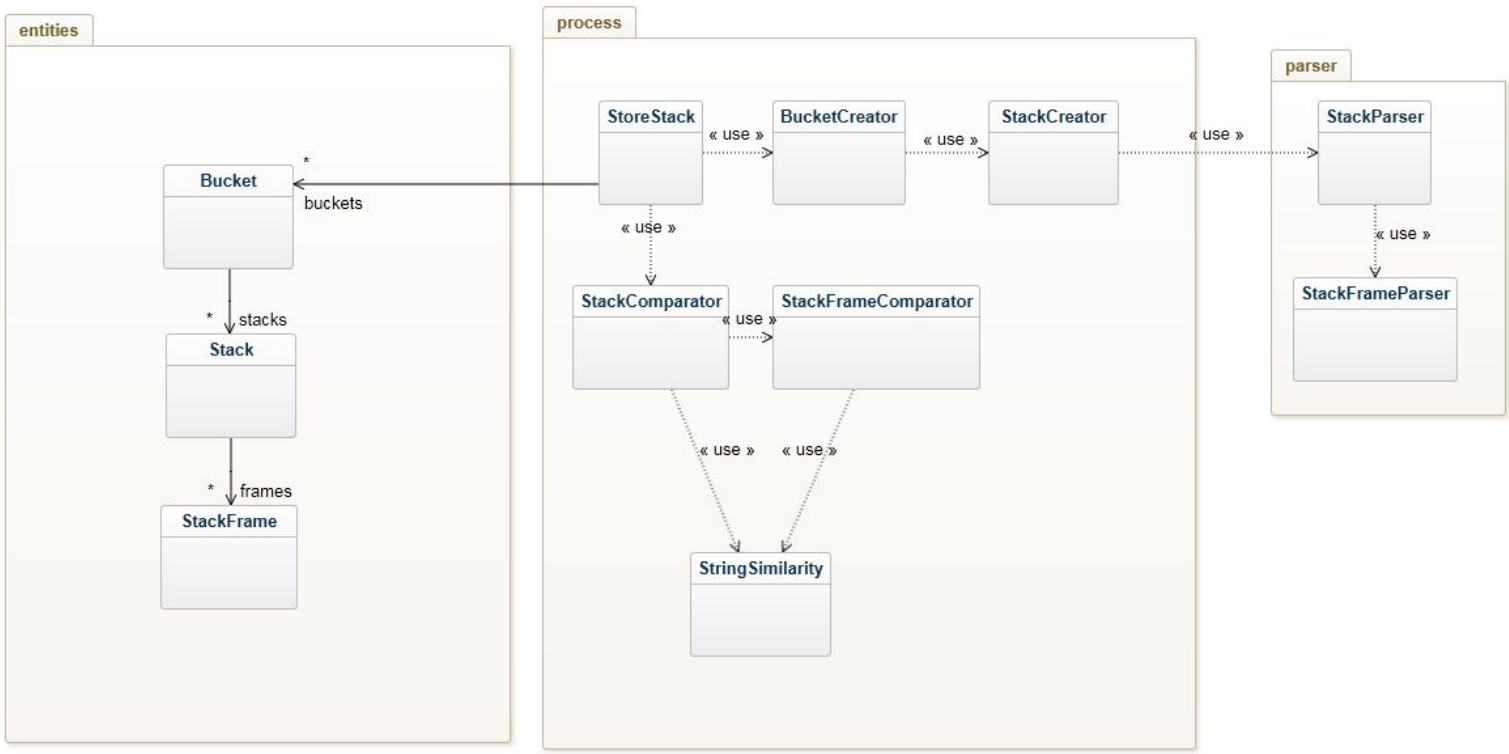
[17] https://fr.wikipedia.org/wiki/Tableau_(structure_de_donn%C3%A9es)

Version finale

Afin d'éviter les erreurs de jugement de nos algorithmes, nous avons décidé de faire une moyenne entre deux approches. Relativement complémentaire, il nous paraissait logique de minimiser les erreurs d'un algorithme en le compensant avec les résultats de l'autre. Nous avons donc multiplié le pourcentage obtenu avec la Distance de Levenshtein avec le score de point obtenu avec la méthode d'Attribution de points.

Architecture

L'architecture de notre programme est séparée de cette façon :



Nous avons trois packages[18] différents. Le premier « entities » contient les classe métier du projet. Ce sont ceux-ci qui correspondent aux objets utilisé par notre programme. Le deuxième, « process », correspond au cœur du projet. Ce sont ces classes qui vont créer, comparer et ranger les StackTrace dans les bon Bucket. Enfin, le troisième, « parser » correspond aux classes de parsage du projet. Ce sont eux qui vont transformer les fichiers en objet java afin de pouvoir les utiliser.

[18] https://fr.wikipedia.org/wiki/Paquet_(logiciel)

Evaluation

Jeux de données

Afin d'évaluer le projet, deux jeu de données nous a été fourni. Ceux-ci contiennent un certain nombre de Bucket qui contiennent eux aussi un certain nombre de StackTrace. Ensuite, il nous est fourni plusieurs StackTrace non classé. Le but est de ranger chacun de ces StackTrace dans le Bucket auquel il appartient. Les résultats donnée par notre solution est testable sur un site web qui permet de nous donner le nombre de bonne réponses.

Voici les scores et performance de chacune de nos approches :

Jeu de données 1 : 209 Buckets et 108 StackTrace à ranger :

Nom de la méthode	Score	Temps d'exécution
Distance de Levenshtein - Naïve	37/108	~ 3 heures
Distance de Levenshtein - Évoluée	48/108	~ 1 seconde
Comparaisons de méthodes	37/108	~ 3 secondes
Attributions de points	50/108	~ 1 seconde
Version finale	59/108	~ 2 secondes

Jeu de données 2 : 272 Buckets et 121 StackTrace à ranger :

Nom de la méthode	Score	Temps d'exécution
Distance de Levenshtein - Naïve	-	-
Distance de Levenshtein - Évoluée	50/121	~ 1 seconde
Comparaisons de méthodes	46/121	~ 3 secondes
Attributions de points	71/121	~ 1 seconde
Version finale	71/121	~ 2 secondes

Pour des raisons de temps, nous n'avons pas exécuté la méthode de Distance de Levenshtein naïve, car celle-ci prend beaucoup trop de temps, nous avons estimé son temps d'exécution à plus de quatre heures. En condition réelle, cela ne serait pas acceptable car il faut un système qui soit capable de trier jusqu'à plusieurs milliers de StackTrace par jour.

De plus, il est intéressant de noter, que la méthode d'attributions de points donne le même score que la version finale, qui combine cette méthode avec la Distance de Levenshtein évoluée, mais uniquement pour le second jeu de données. Dans le premier, le score est amélioré. La raison précise n'a pas été déterminée.

Test contrôlés

Exécution des tests

Pour lancer les tests il suffit d'ouvrir la classe Main et de mettre le boolean static TEST à true, puis de choisir le numéro de test à exécuter en l'affectant à la variable static TESTNUMBER.

Chaque dossier de test se trouve dans le dossier "eval" et contient le résultat attendu. Les numéro de tests sont indiqués avant chaques tests dans la suite.

Test 1

Nous avons créé trois Buckets composés chacun d'un StackTrace chacuns différent des autres, le but est de prendre les trois StackTraces et d'essayer de les ranger dans leur Bucket respectif (qui contient le même StackTrace).
Cette fois-ci dans un Bucket seul un stack est identique au stack testé :

Nom de la méthode	Score
Distance de Levenshtein - Évoluée	3/3
Comparaisons de méthodes	3/3
Attributions de points	3/3
Version finale	3/3

Test 2

Cette fois-ci on prend un StackTrace et on essaye de le ranger dans des Buckets qui contiennent le même StackTrace avec des méthodes supprimées (remplacés par "??"), le but sera de ranger le StackTrace dans le Bucket contenant le StackTrace avec le moins de méthodes supprimé:

Nom de la méthode\Nombre de méthodes décalage	Bien rangé
Distance de Levenshtein - Évoluée	Non
Comparaisons de méthodes	Oui
Attributions de points	Oui
Version finale	Oui

Test 3

Cette fois-ci nous allons tester l'impact de l'ordre des StackFrame dans un stack, pour cela on va partir d'un stack test, et on va essayer de le ranger dans des Buckets qui contiennent ce stack avec un décalage allant de 1 à 4 inclus.

Nom de la méthode\Nombre de méthodes décalage	Bien rangé
Distance de Levenshtein - Évoluée	Non
Comparaisons de méthodes	Oui
Attributions de points	Oui
Version finale	Oui

Nous voyons qu'avec cette méthode d'évaluation, la méthode de la distance de Levenshtein ne range pas correctement le StackTrace, cela est dû au fait qu'il regarde uniquement les StackFrame d'index zéro.

Test 4 à 13

Pour ce dernier test, nous avons choisi de retirer des StackTrace de Bucket contenant plus d'un StackTrace. Ensuite, nous avons demandé à nos algorithmes de classer ces StackTrace dans les bons Buckets en analysant tous les Buckets disponible.

Méthode \ test	4	5	6	7	8	9	10	11	12	13	Total
Distance de Levenshtein - Évoluée	Non	Oui	Non	Non	Oui	Oui	Oui	Oui	Non	Non	5 / 10
Comparaisons de méthodes	Oui	Oui	Non	Non	Non	Oui	Non	Oui	Non	Oui	5 / 10
Attributions de points	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Non	Oui	8 / 10
Version finale	Oui	Oui	Non	Oui	Oui	Oui	Oui	Oui	Non	Oui	8 / 10

Nous nous somme rendu compte lors de l'analyse, que les StackTrace provenant de Bucket possèdent beaucoup de StackTrace ont plus de chance d'être mieux classé, contrairement aux StackTrace provenant d'un Bucket en possèdent uniquement deux.

De plus, nous nous apercevons que les deux dernières méthodes sortent du lot. Comme avec le second jeu de données, la version finale ne fait pas mieux que la méthode d'attribution de points toute seule. Même si certains rangements sont différents entre elles.

Discussion / Limitations

Dans notre cas, faire un système qui atteint un score parfait n'est pas possible. En effet les StackTrace qu'il faut classer, l'ont été par des humains et très certainement pour des raisons très spécifiques. Nos approches n'étant pas capables de déterminer des cas spécifique ou des exceptions cela explique alors la difficulté de s'approche d'un score parfait.

De plus, il fut difficile de vérifier dans quel Bucket un StackFrame doit être rangé. Car la quantité de données fait qu'il est difficile de vérifier à la main les données. C'est pourquoi nous avons dû trouver des solutions afin d'estimer la validité de nos approches. L'une d'entre elle fût de faire notre propre jeu de données, en retirant des StackFrame déjà rangé dans un Bucket et demander à nos algorithmes de ranger ces données.

Aussi, nous avons étudié les méthodes et librairies de machine learning, cependant ceux-ci n'était pas adapté à la comparaison de chaine de caractère, qui est à la base de nos approches. Des solutions était présente afin de rendre les librairies compatible, cependant elles demandent un investissement trop important au vu du temps prévu pour le projet.

Conclusion

L'idée du projet était de partir sur des comparaisons des StackFrame comme de simple chaîne de caractère. Petit à petit, nous avons fait évoluer cette méthode vers une comparaison des éléments d'une StackFrame un à un tout en prenant en compte leur position. L'évaluation fut faite avec plusieurs jeux de données différente.

Ranger des StackFrame dans des Bucket n'est pas une tâche aisée. La pertinence des données n'est pas facile à estimer ni même à comparer. Beaucoup de cas particulier existe mais ils sont compliqué à prendre en compte ou même à identifier.