

Analyse et insertion de stacktraces dans des buckets pour la correction d'erreurs

Antonin DUREY, Jihad EL FAKAWY

15 Novembre 2016



UFR IEEA
Formations en
Informatique de
Lille 1

Table des matières

| | |
|-------------------------------|-----------|
| Introduction | 3 |
| 1 Travail technique | 4 |
| 1.1 But | 4 |
| 1.2 Overview | 4 |
| 1.3 Algorithmes | 5 |
| 1.4 Architecture | 8 |
| 1.5 Implémentation | 9 |
| 1.6 Utilisation | 10 |
| 1.7 Documentation | 10 |
| 2 Évaluation | 10 |
| 2.1 Tests unitaires | 10 |
| 2.2 Efficacité | 10 |
| 2.3 Complexité | 11 |
| 2.4 Performance | 12 |
| Conclusion | 14 |

Introduction

De nos jours, les crashes de logiciels ou de programmes sont quotidiens et les grosses applications doivent faire face à des milliers de rapport de crashes. L'envoi de ces rapports aux développeurs est néanmoins essentiel pour permettre à ceux-ci de corriger les problèmes et failles. Pour simplifier l'analyse de ces rapports, ceux-ci sont triés par bucket. Un bucket rassemble ainsi une série de rapports dont la source du problème est la même. Grâce à cela, lorsque les développeurs analyseront les rapports contenus dans chacun des bucket, ils disposeront d'une grande quantité d'information pour corriger le problème.

Dans le cadre du module Outils pour la Programmation des Logiciels de notre Master 2 IAGL, nous avons été amenés à réaliser un projet qui porte sur l'analyse de crash. Sur ce projet, nous avons implémenté le tri de rapports d'erreurs dans des buckets.

Pour trier ces rapports, nous avons implémenté quatre algorithmes. Dans un premier temps, nous les avons triés de manière aléatoire. Dans un second temps, nous avons attribué un certain nombre de points aux buckets en fonction des points communs entre le rapport à trier et les rapports contenus dans les buckets. Le bucket choisi est alors celui dont le score est le plus élevé. Le troisième algorithme est une variante du second. Il calcule la moyenne des points par nombre de lignes d'erreurs et par nombre de rapports dans un bucket. Enfin, le dernier algorithme stocke les données sous une autre forme avant d'attribuer les stacktraces selon la similitude avec les données nouvellement stockées.

L'évaluation de nos résultats a été facilitée par l'infrastructure mise en place. En effet, nous pouvions soumettre nos résultats, c'est-à-dire l'attribution des rapports d'erreurs à un bucket précis. Grâce à cela, nous avons constaté que notre quatrième algorithme est plus performant que le troisième et le second, eux-même plus performants que le premier. De plus, ce dernier algorithme est beaucoup plus efficace que les précédents. Nous avons également testé nos algorithmes en vérifiant le set d'apprentissage, c'est-à-dire les buckets qui nous avaient été fournis.

Ce rapport présente ces travaux en 2 parties. La première est consacré au travail technique qui a été effectué. La seconde se concentre sur l'évaluation des résultats et des algorithmes. La conclusion présente quant à elle des possibilités d'amélioration.

1 Travail technique

1.1 But

Une **stacktrace** - ou trace d'appels en français - représente la pile d'exécution photographiée à un moment t lors de l'exécution d'un programme. Lorsqu'une erreur survient lors de l'exécution d'un programme, il est désormais assez courant que la stacktrace soit envoyée automatiquement aux développeurs.

Pour les développeurs, le traitement des stacktraces ainsi reçues est très important. Il permet de mettre à jour de nouveaux bugs pas forcément connus auparavant et d'emmagasiner des informations sur ceux-ci pour pouvoir les résoudre le plus rapidement possible. Néanmoins, la première tâche lors de la réception d'une stacktrace est de comprendre à quel bug celle-ci fait référence. Cela permettra par la suite de créer des groupes de stacktraces - nommés **buckets** - représentant la même erreur, toujours dans le but d'obtenir un maximum d'informations sur une erreur.

C'est sur cette opération de rassemblement de buckets que nous intervenons. Notre but est, à partir de buckets déjà existant, d'intégrer des stacktraces à ces buckets. Pour cela, nous avons implémenté 4 algorithmes différents (*voir plus loin*) qui permettent, pour une liste de stacktraces, d'inclure chacune de ces stacktraces au bucket qui lui correspond le mieux.

Tout d'abord, nous commençons par parser des fichiers correspondant à des stacktraces, dont voici un exemple :

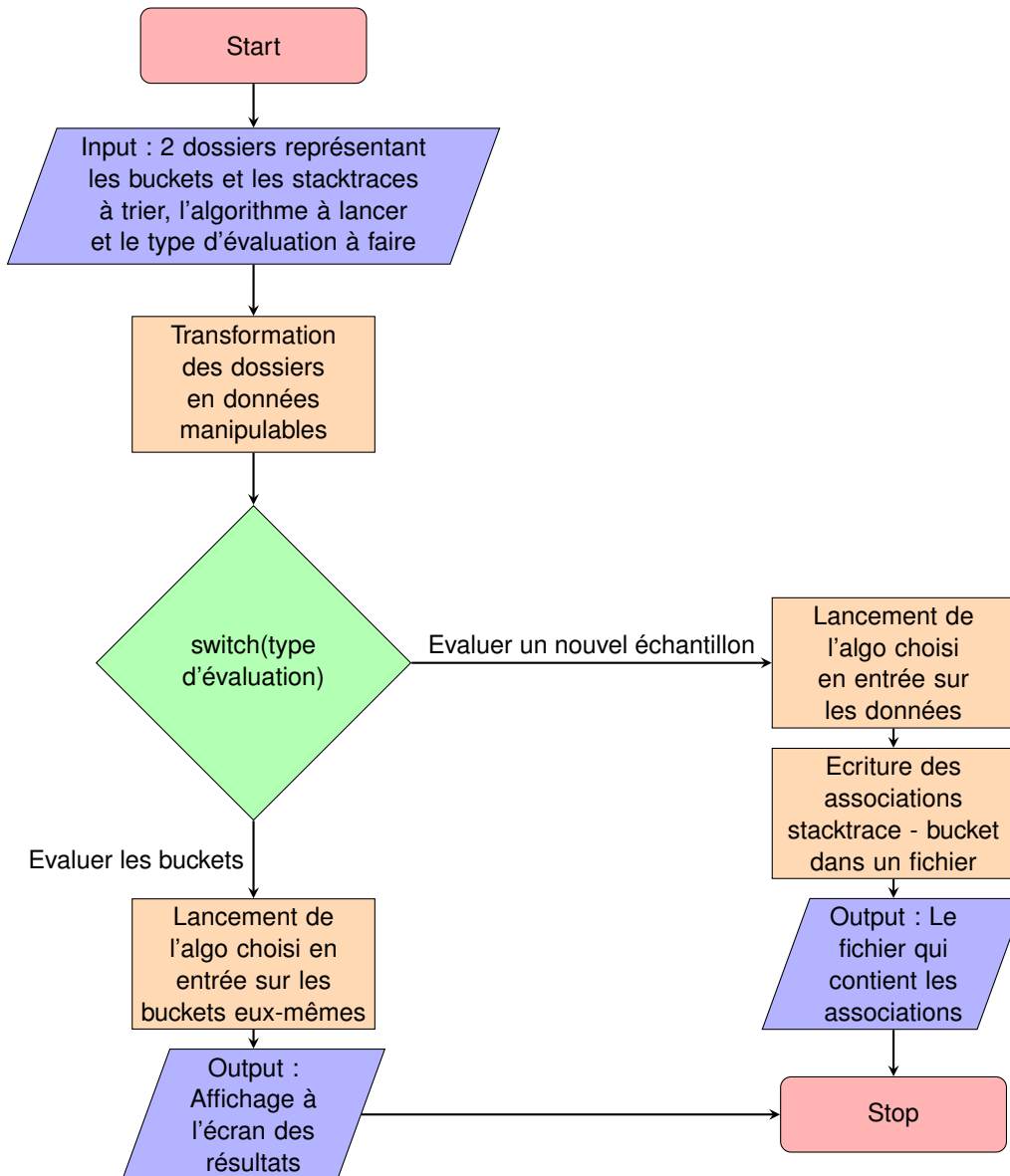
```
#0 0x00c482ac in dconf_engine_refresh_user (engine=0x84aedc8) at ../engine/dconf-engine.c:146
    PRETTY_FUNCTION = "dconf_engine_refresh_user"
#1 0x00c48723 in dconf_engine_refresh (engine=0x84aedc8) at ../engine/dconf-engine.c:197
No locals.
#2 dconf_engine_read_internal (engine=0x84aedc8, key=0x857fdc8 "/org/gnome/settings-daemon/plugins/power/idle-dim-ac",
user=1, system=1) at ../engine/dconf-engine.c:382
    value = 0x0
    lowest = <optimized out>
    limit = <optimized out>
    i = <optimized out>
#3 0x00c4b410 in dconf_settings_backend_read (key=0x857fdc8 "/org/gnome/settings-daemon/plugins/power/idle-dim-ac",
backend=0x84dc870, expected_type=<optimized out>, default_value=<optimized out>) at dconfsettingsbackend.c:452
No locals.
#4 dconf_settings_backend_read (backend=0x84dc870, key=0x857fdc8 "/org/gnome/settings-daemon/plugins/power/idle-dim-ac",
expected_type=0x37f560, default_value=0) at dconfsettingsbackend.c:438
    dcsb = 0x84dc870
#5 0x001a8bce in g_settings_backend_read (backend=0x84dc870, key=0x857fdc8 "/org/gnome/settings-
daemon/plugins/power/idle-dim-ac", expected_type=0x37f560, default_value=0) at /build/build/glib2.0-
2.30.0/.gio/gsettingsbackend.c:732
    value = <optimized out>
```

FIGURE 1 – Exemple de stacktrace avec ses informations utiles

Les données surlignées sont les données qui nous intéressent dans notre travail. Elles représentent le nom de la fonction, ses paramètres, ainsi que le chemin complet et la ligne où a eu lieu l'erreur. Ces données sont généralement présentes pour chaque couche de la pile d'exécution. Une fois ces données analysées, nous pouvons alors trier les stacktraces.

1.2 Overview

Le workflow suivant présente l'ensemble de notre travail pendant ce projet. Il reprend non seulement la partie sur les algorithmes (*voir partie suivante*) permettant de déterminer le meilleur bucket pour une stacktrace, mais il traite également des étapes permettant de traiter les données.



Comme le workflow le montre, il existe 2 manières de travailler pour l'algorithme choisi :

- Il travaille avec un dossier en entrée représentant les buckets et un autre dossier représentant les stacktraces à trier.
- Il travaille avec un dossier en entrée représentant les buckets et analyse ces mêmes buckets.

1.3 Algorithmes

Nous avons implémenté 4 algorithmes différents permettant d'analyser des stacktraces et de les incorporer à des buckets déjà existants.

Le premier est un algorithme aléatoire qui à chaque stacktrace à trier, associe un numéro de bucket aléatoire parmi ceux existants.

Algorithm 1: Algorithme d'attribution de buckets aléatoire

Data: les buckets existants nommés *buckets* et la liste des stacktraces à trier nommée *stacktracesToBeAttributed*
Result: une liste avec pour chaque stacktrace à trier le numéro du bucket auquel la stacktrace est associée
initialization
for *stacktrace* **in** *stacktracesToBeAttributed* **do**
 | `addToResults(stacktrace, getRandomBucketId(buckets))`
end

Le second algorithme parse les buckets et stacktraces existants. C'est à dire qu'il stocke les informations ligne par ligne pour chaque stacktrace. Cela permet d'obtenir la librairie ou le fichier, la méthode et la ligne où l'erreur s'est produite.

Dans un second temps, il analyse, pour chaque stacktrace à trier, toutes ses lignes ainsi que toutes les lignes des stacktraces appartenant aux buckets, et attribue des points en fonction de la similitude entre les lignes. Ainsi, l'algorithme accumule des points de similitude par bucket. Pour chaque stacktrace, le bucket choisi est celui qui a le plus de points. C'est cette seconde partie qui est présentée page suivante :

Algorithm 2: Algorithme d'attribution de buckets basé sur le calcul de points par rapport à la similitude des lignes des stacktraces

Data: les buckets existants nommés *buckets* et la liste des stacktraces à trier nommée *stacktracesToBeAttributed*

Result: une liste avec pour chaque stacktrace à trier le numéro du bucket auquel la stacktrace est associée

initialization

```

for stacktraceToBeAttributed in stacktracesToBeAttributed do
  initialize(bucketsWithValue)
  for bucket in buckets do
    points ← 0
    for stacktraceFromBucket in bucket do
      for lineFromToBeAttributed in stacktraceToBeAttributed do
        for lineFromBucket in stacktraceFromBucket do
          points ← points + getPointsFromLibSimilitude(lineFromBucket,
            lineFromToBeAttributed)
          points ← points + getPointsFromFileSimilitude(lineFromBucket,
            lineFromToBeAttributed)
          points ← points +
            getPointsFromMethodSimilitude(lineFromBucket,
            lineFromToBeAttributed)
          points ← points + getPointsFromLineSimilitude(lineFromBucket,
            lineFromToBeAttributed)
        end
      end
    end
    putIntoMap(bucketsWithValue, bucket, points)
  end
  chosenBucket ← getBucketWithHighestValue(bucketsWithValue)
  addToResults(map, chosenBucket)
end

```

Le troisième algorithme est une variante du second. Au lieu de calculer la somme des points, on calcule la moyenne des points en fonction du nombre de lignes de la stacktrace et du nombre de stacktraces dans un bucket. Cela donne donc un nombre de points moyen par ligne et par bucket. La fin de l'algorithme est la même : le bucket avec le plus de points est désigné.

Le quatrième et dernier algorithme est très différent. Il suppose davantage de stockage d'éléments. En effet, lors de la création des stacktraces, l'algorithme va stocker les bibliothèques, fichiers et méthodes dans une structure sous forme de maps imbriquées. Ainsi, lors de l'attribution d'une liste, il n'y a plus besoin de parcourir chacune des lignes, mais seulement de regarder cette structure pour voir quels sont les éléments en commun.

Algorithm 3: Algorithme d'attribution de buckets basé sur le calcul de points par rapport au nombre de librairies, fichiers et fonctions en commun

Data: les buckets existants nommés *buckets* et la liste des stacktraces à trier nommée *stacktracesToBeAttributed*

Result: une liste avec pour chaque stacktrace à trier le numéro du bucket auquel la stacktrace est associée

initialization

```

for stacktraceToBeAttributed in stacktracesToBeAttributed do
  initialize(bucketsWithValue)
  for bucket in buckets do
    points ← 0
    for stacktraceFromBucket in bucket do
      /* La comparaison des stacktraces s'effectue maintenant grâce à la structure
      contenant les librairies, fichiers et fonctions présentes dans chaque
      stacktrace. Le parse de ces données a été effectué précédemment lors de la
      création de la stacktrace */
      for lib in getLibsUsed(stacktracesToBeAttributed) do
        if stacktraceUsedLib(stacktraceFromBucket, lib) then
          points ← points + 1
          for file in getFilesUsed(stacktracesToBeAttributed, lib) do
            if stacktraceUsedFileFromLib(stacktraceFromBucket, lib, file)
            then
              points ← points + 5
              for function in
                getFunctionsUsed(stacktracesToBeAttributed, lib, file) do
                if
                  stacktraceUsedFunctionFromLibAndFile(stacktraceFromBucket,
                  lib, file, function) then
                    points ← points + 25
                end
              end
            end
          end
        end
      end
    end
    putIntoMap(bucketsWithValue, bucket, points)
  end
  chosenBucket ← getBucketWithHighestValue(bucketsWithValue)
  addToResults(map, chosenBucket)
end

```

Le premier algorithme basé sur l'aléatoire est, de manière assez logique, non déterministe. Les trois autres algorithmes sont déterministes.

1.4 Architecture

Pour ces travaux, nous avons choisi le langage Java. Il a permis de gagner du temps sur la structure des données manipulables car nous étions très compétents dans ce langage, Ainsi, nous avons pu nous concentrer sur le coeur du problème, à savoir l'analyse des données.

Nous n'avons utilisé aucune librairie dans notre architecture pour développer notre solution. La seule librairie utilisée est JUnit pour tester l'analyse syntaxique des fichiers (*voir plus loin*).

Notre architecture se subdivise en 7 packages dont voici la liste :

- *action* : Contient différentes actions nécessaires au fonctionnement de l'application comme la Factory ou le Writer pour l'écriture dans un fichier
- *action.decideur* : Contient les différents algorithmes de décision qui permettent d'affecter un bucket à une stacktrace.
- *model.analysis* : Contient les données créées pendant l'attribution d'un bucket à une stacktrace.
- *model.parsing* : Contient les données créées pendant l'analyse syntaxique.
- *read* : Contient le code permettant de transformer les fichiers en données manipulables.
- *read.exception* : Contient les exceptions renvoyées lors de la transformation des fichiers en données manipulables.
- *start* : Contient les programmes pouvant être lancés : l'analyse sur les buckets et l'analyse d'un nouvel ensemble de stacktrace.

Pour résumer, notre architecture est divisée en 2 parties :

- Une première partie transformation des fichiers en données manipulables.
- Une seconde partie qui attribue à des stacktraces un bucket.

1.5 Implémentation

Dans notre implémentation, nous avons mis en place un système de pattern matcher pour récupérer les informations qui nous serviront par la suite. Le tableau ci dessous récapitule les patterns et les informations que nous en retirons.

| Pattern | Données |
|--------------------------------|---|
| #[0-9]* | Le numéro de la couche de la pile d'exécution |
| in [a-z A-Z 0-9]* | Le nom de la méthode |
| at [a-z A-Z 0-9]* | Le chemin du fichier |
| from [a-z A-Z 0-9]* | Le chemin de la librairie |
| [: .][0-9]* | Le numéro de ligne de l'erreur |

Notre implémentation comprend aussi une classe abstraite nommée *Decideur* montrée ci-dessous.

```
public Decideur(Map<String, Bucket> buckets){
    this.buckets = buckets;
    this.valuesDecided = new ValuesDecided();
}

public abstract ValuesDecided decide(Map<Integer, Stacktrace> toBeAttributed);
public abstract String decideStacktrace(Stacktrace stacktrace);
```

FIGURE 2 – Decideur.java

L'utilisation d'une classe abstraite nous a permis de laisser la main à l'utilisateur sur quel algorithme il désirait tester. Chacune des classes représentant un algorithme doit implémenter cette classe abstraite. Dans notre implémentation, nous disposons donc de 4 classes (*RandomDecideur*, *SumPointsDecideur*, *AveragePointsDecideur* et *LibFileAndFunctionsDecideur*) qui représentent chacun des algorithmes présentés précédemment.

1.6 Utilisation

Notre application dispose de 2 classes principales pouvant être lancées :

- *StartDecideurOnStacktraces* est l'algorithme qui permet d'attribuer un bucket à de nouvelles stacktraces.
- *StartEvaluatingBuckets* permet, comme son nom l'indique, d'évaluer les buckets déjà existants.

1.7 Documentation

Nous avons documenté nos projet avec de la Javadoc, qui se trouve [ici](#).

2 Évaluation

Comme vu précédemment, notre projet se décompose en 2 parties : la transformation des fichiers en données manipulables et l'attribution d'un bucket aux stacktraces. Pour évaluer cette première partie, nous avons effectué des tests unitaires. Quant à la seconde, nous disposons de 2 manières : le serveur de validation mis en place par notre enseignant, ou l'analyse de l'ensemble d'apprentissage, c'est-à-dire des buckets. Pour évaluer ces éléments, nous parlerons ci-dessous de leur efficacité, de leur complexité et de leur performance.

2.1 Tests unitaires

Nous avons réalisé de nombreux tests unitaires portant sur l'analyse syntaxique des fichiers. Voici les statistiques de coverage concernant ces tests unitaires.





| | | | |
|---|---|--|-----|
| > |  <i>AnalyzeStacktrace.java</i> |  79,7 % | 408 |
| > |  <i>StacktracesReader.java</i> |  97,7 % | 86 |
| > |  <i>BucketsReader.java</i> |  90,1 % | 73 |

FIGURE 3 – Couverture des tests unitaires

Ces tests couvrent de très nombreux cas pour récupérer la ligne, la fonction, le fichier d'une erreur, ou d'autres éléments importants. Deux tests unitaires sont présentés ci-dessous :

2.2 Efficacité

Le premier critère de nos algorithmes est l'efficacité, c'est à dire le temps d'exécution. Avant toute chose, il importe de dire que, dans la présentation de nos résultats, nous n'avons pas inclut les temps nécessaire à l'analyse syntaxique des fichiers pour les transformer en données manipulables. Nous avons fait cela pour plusieurs raisons : tout d'abord, ce temps ne fait pas clairement partie du temps d'exécution des algorithmes. De plus, le temps d'analyse syntaxique de fichier est constant selon l'algorithme de décision, environ 2 secondes.

Nous avons tout d'abord testé l'efficacité des algorithmes sur le premier set de stacktraces - de taille 208. Voici les temps obtenu à pour l'exécution de chacun des algorithmes :

```

@Test
public void testGetLibFrom() {
    String path=" /usr/lib/libglib-2.0.so";
    String path2= new AnalyzeStacktrace().getLibFrom(LINE2);
    assertEquals(path,path2);
    assertEquals(0,new AnalyzeStacktrace().getLigne(LINE2));
}

@Test
public void testGetLigne() {
    assertEquals(81, new AnalyzeStacktrace().getLigne(LINE));
}

```

FIGURE 4 – Quelques tests unitaires

- Algorithme aléatoire : **9 ms**
- Algorithme de somme des points : > 2 minutes
- Algorithme de moyenne des points : > 2 minutes
- Algorithme de comparaison des librairies, fichiers et fonctions utilisés : **550 ms**

En terme de rapidité, l'algorithme aléatoire et l'algorithme de comparaison des librairies, fichiers et fonctions utilisés sont donc bien plus efficaces que les 2 autres algorithmes. Ces derniers sont en l'état inutilisables en conditions réelles. En effet, le nombre de stacktrace étant plus important, ils seraient très peu efficaces et donc inutiles. Par ailleurs, il est tout à fait normal que les second et troisième algorithmes aient le même temps d'exécution, la manière d'attribution ne variant que de manière infime.

Nous avons également testé l'efficacité de ces algorithmes avec le second système d'évaluation, c'est à dire l'évaluation des buckets déjà créés. Le set de buckets que nous avons dès le début de projet était constitué de 1 251 stacktraces. Voici les temps obtenus :

- *Algorithme aléatoire* : **55 ms**
- *Algorithme de somme des points* : entre 1h30 et 2h
- *Algorithme de moyenne des points* : entre 1h30 et 2h
- *Algorithme de comparaison des librairies, fichiers et fonctions utilisés* : **2.5 s**

Avec ces résultats, le premier et dernier algorithme se trouvent toujours dans des temps d'exécution raisonnables et même intéressants. Les deux autres algorithmes sont par contre clairement improductifs si aucune amélioration n'est faite.

2.3 Complexité

La complexité des algorithmes est un élément très important à prendre en compte. Comme nous ne pouvons pas tester sur des échantillons immenses, faute de données, c'est cela qui déterminera si nos algorithmes seront efficaces dans des cas d'utilisation réels.

Pour présenter la complexité de nos différents algorithmes, nous estimerons que les comparaisons de chaîne de caractères, recherche de clef dans une table de hachage, calcul d'un nombre aléatoire et autres opérations élémentaires s'effectuent en temps constant noté t . Ci-dessous sont présentés les complexités des algorithmes :

- *Algorithme aléatoire* : nombre de stacktrace à attribuer * t

- *Algorithme de somme des points* : nombre de stacktrace à attribuer * nombre de buckets * nombre de stacktraces dans chacun des buckets * nombre de lignes de la stacktrace du bucket * nombre de lignes de la stacktrace à attribuer * t
- *Algorithme de moyenne des points* : idem que précédemment
- *Algorithme de comparaison des librairies, fichiers et fonctions utilisés* : nombre de stacktrace à attribuer * nombre de buckets * nombre de stacktraces dans chacun des buckets * nombre de librairies en commun au 2 stacktraces * nombre de fichiers en commun à la librairie * nombre de fonction en commun au fichier * t

Les complexités sont assez logiquement polynomiales. Comme les calculs ci-dessus nous y faisaient penser, les 2 algorithmes les moins efficaces ont une complexité très grande, c'est-à-dire que dès que l'on modifie un paramètre ou la taille d'un paramètre, le temps d'exécution augmente.

Le dernier algorithme est à première vu aussi complexe voire plus complexe que les autres car il possède plus de variables dans la formule de complexité. Néanmoins, il faut comprendre que si les deux stacktraces comparées n'ont aucune librairie en commun, l'algorithme ne calculera rien et sera donc très court pour cette itération. L'algorithme ne va réellement prendre plus de temps que lorsque les deux stacktraces comparées auront des librairies, des fichiers, et pourquoi pas des fonctions en commun. Ainsi, dans de très nombreux cas, l'algorithme sera (très) rapide.

L'algorithme aléatoire a une complexité très simple. Il suffit en effet de calculer un id de bucket aléatoire pour chaque stacktrace à attribuer.

2.4 Performance

Dans un dernier temps, nous avons estimé la performance de nos algorithmes, c'est à dire combien de stacktraces à attribuer sont correctement placées. Pour cela, nous nous sommes servis du validateur mis en ligne par notre enseignant dans un premier temps. Voici les résultats :

- *Algorithme aléatoire* : 2 stacktraces convenablement attribuées sur 108
- *Algorithme de somme des points* : 3/108
- *Algorithme de moyenne des points* : **22/108**
- *Algorithme de comparaison des librairies, fichiers et fonctions utilisés* : **23/108**

De manière générale, ces résultats sont insuffisants, voir décevants, c'est-à-dire que les algorithmes ne pourraient pas être utilisés dans de vrais cas. Cela étant dit, il y a une différence énorme entre les deux premiers algorithmes et les 2 suivants.

Le premier algorithme étant aléatoire, les résultats varient légèrement mais restent toujours inférieurs à 10. En effet, comme il existe plus de 200 buckets, il est assez rare que le bucket choisi aléatoirement soit le bon. Cet algorithme n'est donc pas performant.

Le second algorithme fait un calcul de somme de points pour trouver le bucket le plus adéquat. Néanmoins, ne jamais diviser par la taille d'un bucket ou la taille d'une stacktrace tend à favoriser les buckets les plus gros ayant des stacktraces les plus grosses. Ainsi, en analysant les résultats de cet algorithme à la main, nous nous sommes aperçus que certains buckets revenaient de manière récurrente dans l'attribution. Après vérification, il s'avère que ce sont bien les buckets les plus gros ayant les stacktraces les plus grosses. Cette vérification nous amène à dire que cet algorithme n'est pas non plus performant.

Le troisième algorithme possède 22 bonnes réponses sur 108, ce qui est de l'ordre de 20%. Bien que cela soit mauvais, comme dit précédemment, on constate une réelle amélioration par le simple fait de diviser le nombre de points obtenus par la taille des buckets et par la taille de chaque stacktrace des buckets.

Le quatrième algorithme, bien que très différent, propose un résultat similaire - 23 bonnes réponses contre 22.

Pour essayer de différencier ces 2 derniers algorithmes, nous avons analysé les buckets avec 3 algorithmes, voici les résultats :

- *Algorithme aléatoire* : 8 stacktraces convenablement attribuées sur 1251
- *Algorithme de somme des points* : Non effectué
- *Algorithme de moyenne des points* : 334/1251
- *Algorithme de comparaison des bibliothèques, fichiers et fonctions utilisés* : 536/1251

Ces résultats sont très intéressants. Ils démontrent que malgré la proximité des résultats dans le cas précédent, le dernier algorithme est ici plus d'une fois et demi plus performant, ce qui est très important. A noter que nous n'avons pas lancé cette évaluation avec le second algorithme. En effet, la complexité étant identique, nous aurions mis probablement le même temps que le troisième algorithme pour avoir des résultats, tout en sachant pertinemment que ceux-ci auraient été moins bons.

Enfin, nous avons fait le choix de ne tester que la validité des buckets de taille supérieure ou égal à 3. Ainsi, lorsque chacune des stacktraces seraient enlevées du bucket pour pouvoir la tester, il resterait au moins 2 stacktraces dans le bucket pour permettre à l'algorithme d'attribuer le bucket correspondant. Le résultat ainsi obtenu est de **513/1062**, ce qui fait monter le pourcentage de stacktrace correctement attribuées **à près de 50%**.

Si nous rassemblons toutes ces analyses, il est clair que **le quatrième algorithme** (*Algorithme de comparaison des bibliothèques, fichiers et fonctions utilisés*) **est le plus intéressant**. Il est en effet le plus performant et l'un des plus efficaces. Les autres algorithmes sont moins intéressants voir inutilisables.

Conclusion

Durant ce projet, nous avons travaillé sur les stacktraces créées lors de l'apparition d'un bug ou d'une erreur dans un programme. Ces travaux nous ont permis de mettre en évidence l'importance du tri des stacktraces pour permettre aux développeurs d'obtenir un maximum d'informations sur les erreurs rencontrées par les utilisateurs.

Notre travail permet ainsi de trier des stacktraces et de les attribuer à un bucket existant. Cela implique de devoir analyser syntaxiquement les fichiers pour en ressortir les données importantes, mais également de pouvoir lancer un algorithme qui fera l'attribution. À ce titre, nous avons développé 4 algorithmes répondant à ce besoin.

Nous avons ensuite évalué ces différents algorithmes. Bien que les résultats globaux soient quelque peu décevants, il est clair que notre quatrième algorithme (*Algorithme de comparaison des librairies, fichiers et fonctions utilisés*) est le plus intéressant. En effet, il répond au besoin posé, est efficace et assez performant.

Les axes d'améliorations de ce projet sont multiples. Il peut être intéressant de regarder les données dont nous ne nous servons pas actuellement pour voir si elles peuvent avoir une importance quelconque. Dans un second temps, il est nécessaire d'améliorer les algorithmes de décision en étant plus précis sur les points à attribuer selon les éléments que deux stacktraces ont en commun. Enfin, il est probable que d'autres solutions existent, des solutions auxquelles nous n'aurions pas pensé et qui seraient encore plus performantes et plus efficaces.