Université Lille 1

Rapport OPL

Crash Analysis

Analyse de rapports de crash

Auteurs :
Franck Warlouzet
David Fitoussi

Responsable :
Martin Monperrus

14 novembre 2016

Table des matières

Introduction **2**

1 Travail technique **4**
1.1 Principes 4
1.2 Réalisation 4
1.2.1 Version aléatoire 4
1.2.2 Version exhaustive 5
1.2.3 Version affinée 6
1.3 Architecture et design 7

2 Evaluation **8**
2.1 Pertinence et temps d'exécution 8
2.1.1 Version aléatoire 8
2.1.2 Version exhaustive 9
2.1.3 Version affinée 10
2.2 Tests 10
2.3 Limitations 11

Conclusion **12**

Introduction

Nous avons tous vécu un jour la douloureuse expérience d'un plantage lors de l'utilisation d'un logiciel, et il n'y a rien de plus frustrant que de tomber sur ces mêmes crashs encore et encore. Pourtant, croyez nous, les éditeurs de logiciels mettent leurs cerveaux à contribution pour essayer de traiter ces erreurs. Que ce soit de la simple analyse manuelle ou des systèmes complexes traitant ces crashs, il existe aujourd'hui une multitude de travaux [1] [2] qui essayent de répondre à ces problèmes.

Cependant il reste aujourd'hui énormément de place à l'innovation et à l'amélioration des systèmes existants. La plupart des logiciels en production fonctionnent en mode fail-safe [3], ce qui évite, du moins essaye au maximum d'éviter, que l'utilisateur se retrouve face à un crash bloquant qui l'empêche de continuer à utiliser le logiciel de manière optimale. Lorsque ce n'est pas le cas, ou que les développeurs du logiciel décident tout simplement d'utiliser les crashs pour l'améliorer, un rapport de crash est alors envoyé au serveur ou stocké dans un log afin qu'il soit analysé. Vous imaginez bien qu'à l'echelle d'un logiciel comme Mozilla Firefox par exemple, ces rapports d'erreur représentent une quantité de données astronomique. Il est donc nécessaire de posséder des outils qui traitent et exploitent toutes ces données afin de les rendre utilisables et leur donner de la valeur. La première étape de ce traitement consiste en trier tous les rapports d'erreur pour constituer des catégories de crashs. Chaque catégorie représente un comportement similaire, un bug identique ou encore une configuration particulière pour laquelle un bug se produit.

C'est sur cet aspect de l'analyse de crash que nous avons travaillé. Nous avons développé une application qui permet de classifier les différents rapports d'erreur. Pour ce faire, nous disposions d'un ensemble de rapports de crash, matérialisés par des traces d'exécutions de l'endroit où le bug s'est produit, et d'un ensemble de catégories de bugs, elles aussi représentées par des ensembles de traces d'exécution,

1. Exemple de travail sur l'analyse de bug logiciel. Lien
2. Autre exemple, Microsoft !exploitable qui repère si un crash est exploitable de manière malveillante. Lien
3. Le fail-safe est une façon de gérer les erreurs de son logiciel afin qu'il ne cause aucun désagrément pour l'utilisateur, celui-ci ne se rend ainsi pas compte des erreurs qui se produisent

que nous appellerons par la suite bucket [4]. Notre application permet d'associer un rapport d'erreur à un bucket. Une fois mis en lien avec une application complète de gestion d'un bucket pour corriger le bug qui lui est associé, notre outil faciliterait le tri des données qui arriveraient en grande quantité chaque jour.

Durant notre analyse, nous avons exploré plusieurs méthodes pour arriver à nos fins. Du plus efficace mais extrêmement complexe, au plus simple mais moins rentable, nous allons présenter les différents algorithmes pour obtenir un résultat que nous avons étudié et réalisé. Ensuite nous ferons une critique de chacune des méthodologies abordées.

4. Le terme bucket vient de Microsoft qui a introduit cette notion avec Windows Error Reporting, depuis Windows XP, qui est un outil de rapport de crash. Lien

Chapitre 1

Travail technique

1.1 Principes

Tous les algorithmes que nous avons implémentés, à l'exception de celui basé sur l'aléatoire, reposent sur le même principe, qui est la comparaison textuelle d'un fichier de test, représentant le rapport d'erreur, avec un fichier représentant un rapport d'erreur déjà catégorisé dans un bucket. Là où nos méthodes d'analyse diffèrent, c'est sur le calcul de la différence entre deux textes. Nous avons ainsi pu comparer plusieurs procédés et algorithmes de calcul de similarité entre deux chaînes de caractères. Nous détaillerons dans la suite en quoi consistent les différents algorithmes utilisés.

1.2 Réalisation

De manière instinctive, nous avons décidé d'utiliser l'analyse syntaxique (ou parsing) pour réaliser nos associations rapports d'erreurs -> bucket. Nous souhaitions rester simple dans la façon de faire, car la complexité peut vite exploser lorsque l'on parse des centaines de fichiers et que l'on effectue des comparaisons textuelles sur chacun d'entre eux. Nous avons séparé notre travail algorithmique en 3 parties, une pour chaque type d'algorithme abordé.

1.2.1 Version aléatoire

Une première version de notre solution est très basique, elle se base sur l'aléatoire. Ayant des rapports à placer dans des buckets, nous avions naïvement pensé que l'aléatoire nous donnerait un résultat correct.

L'algorithme est simpliste, mais nous donnait une première estimation de la complexité à laquelle nous allons faire face dans les implémentations futures. Le principe est le suivant :

Result: Association de fichiers test -> bucket
for *tous les fichiers test* **do**
  for *tous les buckets* **do**
    index = index aléatoire entre 0 et buckets size - 1;
    on stocke le couple (fichier test -> buckets[index];
  end
end

Algorithm 1: Répartition aléatoire

1.2.2 Version exhaustive

La méthode aléatoire restant trop peu robuste, nous proposons donc une autre façon radicalement différente de réaliser nos associations. Les traces d'exécution contiennent un bon nombre d'informations exploitables afin de réaliser ces associations. Il est clair que nous devons utiliser ces informations pour trouver dans quel bucket mettre le rapport d'erreur traité. Nous traitons ces informations sous forme de mots clés qui caractérisent un rapport d'erreur. Pour réaliser une association, nous cherchons le bucket qui contient la trace d'exécution se rapprochant le plus de ce rapport d'erreur, c'est à dire qui contient le plus de mots clés similaires à notre rapport d'erreur. Pour cet algorithme, le principe est simple, nous calculons la similarité entre les fichiers de test et les traces d'exécution des buckets sans prendre en compte des éléments précis du contenu de ces fichiers. Cette approche naïve était rapide à implémenter et nous donnait un aperçu plus réaliste du résultat à obtenir que la version aléatoire. Le principe de cet algorithme est le suivant :

Result: Association de fichiers test -> bucket
for *tous les fichiers test f* **do**
  for *tous les buckets b* **do**
    score = similarité entre f et b;
    on stocke le couple (bucket -> score);
  end
  on récupère le bucket avec le score max;
  on affecte ce bucket au fichier test;
end

Algorithm 2: Répartition basée sur un score de similarité entre fichiers

Nous remarquons qu'il existe un élément qui peut varier et être affiné dans cet algorithme. Il s'agit du calcul de similarité entre les fichiers. Nous avons décidé de considérer 3 variantes de ce calcul. Une première évalue entre 0 et 1 la similarité de deux chaînes de caractères (ici le contenu du fichier complet). Une deuxième utilise la distance de Levenshtein [1]. Et enfin, nous avons utilisé la distance de Hamming [2]pour calculer cette similarité. Nous verrons dans la suite l'impact qu'a eu cette variation sur le résultat obtenu et le temps d'exécution.

1.2.3 Version affinée

Finalement, nous proposons une 3ème alternative qui est plus raffinée dans la manière de traiter les fichiers. Nous avons vu ci-dessus que les rapports d'erreur et les traces d'exécution des buckets contiennent bon nombre d'informations exploitables afin de les traiter et les analyser. Cette 3ème version utilise donc ces informations au maximum afin de réaliser des associations les plus précises possibles tout en restant générique afin de garder la même efficacité peu importe le jeu de données dont on dispose. Parmi toutes les informations présentes, nous avons choisi d'exploiter, lorsque celle-ci était disponible, les noms de fonction dans laquelle l'erreur s'est produite, la ligne à laquelle c'est arrivé, le fichier concerné avec son path [3] complet et enfin l'indice de la ligne dans la trace d'exécution. Ainsi, nous matérialisons une trace d'exécution comme un ensemble de lignes comme ceci :

```
#indice nomDeFonction fichier ligne
```

Après avoir construit notre modèle de données (dans lequel une ligne représente une ligne comme décrite ci-dessus) pour chacun des fichiers de test et chaque bucket, nous appliquons l'algorithme suivant :

1. La distance de Levenshtein entre 2 chaïnes est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.
2. La distance de Hamming compte le nombre de caractères différents entre deux chaînes, e.g 'toto' et 'tato' ont une distance de 1, tandis que 'toto' et 'barr' ont une distance de 4.
3. Le path est le chemin d'accès d'un fichier ou d'un répertoire

```
Result: Association de fichiers test -> bucket
for tous les fichiers test f do
    for tous les buckets b do
        for chaque ligne de f do
            for chaque ligne de b do
                score = calcul du match entre les 2 lignes;
                on stocke le score;
            end
        end
        on somme tous les scores pour ce bucket;
        on stocke le couple (bucket -> somme des scores);
    end
    on récupère le bucket avec le score max;
    on affecte ce bucket au fichier test;
end
```

Algorithm 3: Répartition basée sur un score de similarité entre fichiers

Là encore, il existe un point dans l'algorithme où l'on peut effectuer des variances pour affiner et obtenir des résultats différents selon les critères que l'on juge les plus importants, comme par exemple la rapidité d'exécution ou la précision des résultats, ou encore un compromis entre les 2. Nous avons là aussi utilisé les 3 variantes décrites dans la partie précédente, mais également ajouté une méthode propre au traitement de granularité ligne (contre granularité fichier dans le cas de l'algorithme précédent). Cette méthode teste l'égalité entre chaque mot clé de deux lignes et affecte un nombre de points en conséquence. On l'appelera par la suite la méthode de scoring. Nous verrons par la suite les avantages et inconvénients de chaque méthode de calcul.

1.3 Architecture et design

Pour implémenter nos différents algorithmes, nous utilisons le langage Java. Ce choix se justifie par la bonne gestion du traitement des fichiers de ce langage par rapport aux autres langages que nous maîtrisons (comme le JavaScript) ainsi qu'une bonne efficacité lorsqu'il est nécessaire de travailler en multithreading [4]. En effet, nos algorithmes utilisent les threads afin d'accélérer considérablement le temps d'exécution (un thread par fichier test traité, ce qui a divisé le temps d'exécution par 3 voire 4 selon les algorithmes et le jeu de données).

4. Une application multithreadée partage le travail sur différents cœurs du processeur, ce qui accélère l'exécution

Chapitre 2

Evaluation

2.1 Pertinence et temps d'exécution

Pour réaliser nos mesures, nous avons lancé tous nos algorithmes sur une machine ayant les caractéristiques suivantes :

- *Windows 10 64bits ;*
- *RAM 8GB ;*
- *Disque dur HDD 5400 tr/min*
- *Processeur Intel Core i7-4700MQ 2.40GHz 4 coeurs.*

Nos algorithmes ont été testés sur 2 jeux de données (data set, DS dans les tableaux qui suivent) afin d'avoir une idée de l'universalité de nos réflexions. Le premier jeu de données comporte 108 fichiers à associer à 217 buckets (contenant un total de 1251 traces d'exécutions). Le deuxième jeu de données comporte 121 fichiers à associer à 275 buckets (contenant un total de 1232 traces d'exécutions).

2.1.1 Version aléatoire

Si le hasard fait bien les choses dans la vie quotidienne, ce n'est certainement pas le cas dans le monde de l'algorithmique. Preuve en est, notre implémentation aléatoire de la répartition des rapports d'erreur nous donne un résultat de 0 association correcte pour 20 tentatives sur le jeu de données 1.

Ceci n'est pas illogique, selon un rapide calcul de probabilité nous avons pu extraire des données par rapport à cet algorithme naïf de répartition. En prenant en compte plusieurs données dont :

- Le nombre de fichiers traiter : 108
- Le nombre de buckets à analyser : 217

Nous avons pu émettre un calcul de probabilité utilisant la loi binomiale avec pour paramètres :

— n (nombre de jets)
— k (nombre attendu de réussites)
— p (probabilité de réussite pour un jet)

n est facile à trouver, il correspond au nombre total de fichiers à traiter (108). k correspond au nombre attendu de fichiers placés dans le bon bucket, celui-ci serait dans l'idéal 108. p correspond à la probabilité qu'un fichier choisi au hasard se retrouve dans le bon bucket($1/217 => 0{,}0046$)

Via ce site permettant facilement de disposer d'un calculateur binomial, nous avons pu être en mesure d'exposer mathématiquement les "performances" de notre algorithme.

Tout d'abord, nous avons voulu savoir combien de bons buckets nous pouvions espérer au maximum. Ce fut assez rapide, en effet, à partir d'un nombre de 3 bons buckets, la probabilité passe déjà à 0.0085 soit beaucoup moins d'1%. A l'inverse, nous avons également cherché la probabilité que notre algorithme nous sorte la totalité des bonnes associations, et comme attendu le résultat est plutôt décevant, $1.0531.10^{-259}$.

Sans surprise, ce calcul nous a permis de valider la non fiabilité de notre algorithme.

2.1.2 Version exhaustive

Concernant la version exhaustive, les résultats obtenus sont plus que corrects, dans la mesure où nous estimons grâce à une analyse empirique que même si l'on intervient manuellement, il serait extrêmement difficile, voire impossible de correctement associer plus de 70 fichiers au bon bucket. Nous listons dans le tableau ci-dessous les différents résultats ainsi que les temps d'exécution pour chacune des méthodes de calcul implémentée.

Méthode	résultat DS 1	temps DS 1	résultat DS 2	temps DS 2
Similarité	46	3h11	56	3h02
Levenshtein	47	3h03	57	2h50
Hamming	28	1s	26	1s

Les temps de calcul très élevés obtenus sont dus aux fichiers volumineux. En effet, lorsque nous retirons les 3 fichiers ayant une taille supérieure à 100ko dans le data set 1, on tombe à un temps d'exécution de 47min pour un score de 45. Ceci s'explique par les méthodes de calcul qui comparent tous les caractères des 2 chaînes entre eux, le nombre d'opérations explose avec l'augmentation de la taille de ces chaînes.

2.1.3 Version affinée

Le constat qui nous a amenés à réfléchir à une troisième solution est que comme vu dans le tableau de résultats ci-dessus, nous avons du mal à être rapides et efficaces à la fois. Après une analyse manuelle des fichiers à traiter, nous avons décidé de ne considérer qu'une partie des fichiers qui nous paraît plus importante que le reste. Nous avons décrit cette partie dans le chapitre précédent, nous allons donc ici uniquement nous intéresser aux résultats que cette méthode donne.

Le fait de ne traiter qu'une partie des fichiers nous fait gagner un temps considérable sur la partie calcul. En effet, la base de l'algorithme reste la même, en revanche le contenu qui est traité dans les calculs est beaucoup plus petit et fractionné et c'est donc moins complexe de calculer les similarités ou encore les distances de Levenshtein et Hamming. Pour rappel, ces trois méthodes parcourent chaque caractère des 2 chaînes considérées les uns avec les autres, ce qui explose donc niveau complexité en même temps que la taille de ces chaînes grandit.

De plus, nous avons pu procéder à la méthode de scoring décrite dans le chapitre précédent. Là encore, nous avons misé sur la rapidité d'exécution afin d'offrir une alternative intéressante à la version exhaustive qui apporte de bons résultats mais en temps très importants.

Nous listons dans le tableau ci-dessous les différents résultats ainsi que les temps d'exécution pour chacune des méthodes de calcul implémentée.

Méthode	résultat DS 1	temps DS 1	résultat DS 2	temps DS 2
Similarité	16	3min35	21	3min40
Levenshtein	17	3min30	23	3min20
Hamming	9	1s	13	1s
Scoring	24	16s	16	9s

Nous retenons de ces résultats qu'on a une forte perte de précision par rapport à la version exhaustive. Ceci s'explique par le nombre de données que nous ne considérons pas importantes et une implémentation qui ne s'adapte pas aussi bien à un traitement partiel des données qu'à un traitement global du fichier.

En revanche, nous observons un très grand gain en vitesse d'exécution. Ceci confirme les motivations qui nous ont poussés à étudier cette solution.

2.2 Tests

Tous les algorithmes que nous avons implémentés ont été pensés dans l'optique d'être testés. Ce type d'application est parfaitement adapté pour être appuyé d'une

large série de tests, qui garantissent la fiabilité des résultats obtenus. C'est ainsi que nous disposons de tests unitaires qui couvrent 94.7% de notre code. Ces tests nous ont en outre permis de corriger et affiner nos algorithmes, et ainsi augmenter le nombre d'associations correctement effectuées.

2.3 Limitations

Ces multiples implémentations offrent différents choix pour une utilisation future de notre application. Il n'existe pas de solution parfaite qui associe correctement tous les fichiers au bon bucket, en revanche, il est possible de s'approcher d'une bonne moyenne qui rendent exploitables les informations reçues lors de récolte de rapports d'erreur durant l'utilisation d'une application.

Il faudra faire un choix lors de l'utilisation de ces rapports d'erreur entre rapidité et précision. Pour une exploitation précise des données, la version exhaustive est préférable, mais si ce flux de données doit être exploité rapidement en continu, la version affinée est plus adaptée. La perte de précision est largement compensée par un temps de calcul et un coût en ressources moindres.

Conclusion

Nous avons vu au travers de raisonnement mathématique, de réflexions empiriques et de mesures temporelles et qualitatives de nos algorithmes qu'il existe plusieurs façons de traiter le problème d'analyse de crash d'application.

Malheureusement, il n'est pas possible de réaliser une application qui répartit sans marge erreur les rapports de crash dans des buckets représentant les catégories de bugs que l'on veut corriger.

Toutefois, notre solution apporte une alternative utilisable, avec une marge d'erreur non négligeable avec au mieux un peu moins d'un rapport bien placé sur 2. Cette ébauche de solution montre qu'il est possible de concevoir un mécanisme qui traite les données d'un rapport de crash pour en tirer profit et ainsi le répartir dans une catégorie de bug similaire. Ce tri permettra ensuite aux développeurs de connaître plus facilement les causes d'un problème et ensuite travailler à la résolution de celui-ci.

Notre solution est ouverte aux améliorations, le code est open-source et est trouvable sur GitHub [1].

1. https://github.com/FranckW/projet2$_opl$