Priorisation des tests logiciels

Alexandre Ouellet et Mourad Badri {alexandre.ouellet | mourad.badri}@uqtr.ca

Laboratoire de recherche en génie logiciel - département de mathématiques et d'informatique

Université du Québec à Trois-Rivières

CRSNG NSERC

Introduction

Le processus de test logiciel est une étape cruciale dans le développement de logiciel. Cette étape consiste à rechercher des fautes dans le logiciel avant sa distribution. Il s'agit de l'étape la plus coûteuse, tant financièrement qu'au niveau des ressources humaines. Une des raisons de ces coûts élevés est l'absence de méthode automatisée capable de détecter les zones les plus critiques d'un logiciel.

Plusieurs approches ont été développées à partir des séquences de tests. Ces approches nécessitent de connaître a priori les tests qui devront être exécutés. Elles permettent de réduire et d'optimiser la séquence de tests. Cependant, connaître les tests préalablement est une condition très restrictive.

Nous avons exploré un nouveau type d'approche pour les systèmes orientés objet. Ces systèmes ont la particularité d'être découpés en composantes appelées classes. Une classe peut être vue comme une abstraction d'une entité du monde réel que l'on souhaite représenter dans le logiciel. Tout comme dans le monde réel, la collaboration entre plusieurs entités permet d'offrir des traitements. Les traitements qu'une classe peut faire sont offerts à travers les méthodes de la classe. Un type de test, appelé test unitaire, à pour but de tester les classes individuellement. Notre approche, au lieu de s'intéresser aux tests écrits, visera à déterminer les classes pour lesquelles il est le plus pertinent d'écrire les tests (optimiser la distribution de l'effort de test).

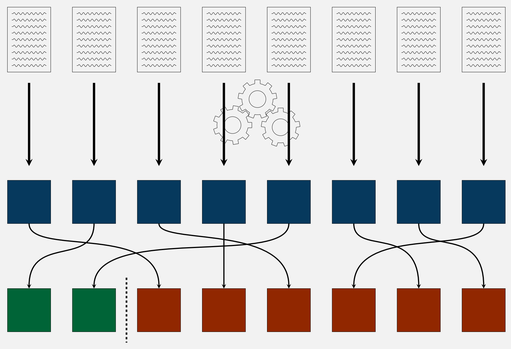
Problème de la priorisation

La détection des classes critiques est un sujet qui a été largement exploré dans la littérature. Plusieurs approches utilisant statistiques ou algorithmes d'apprentissage artificiels ont été développées et validées. Cependant, la priorisation des tests est un élément qui a été essentiellement exploré pour les tests et non pour les classes. Une approche de priorisation consiste à déterminer un ordre de priorité pour l'exécution des tests. Formellement, le problème de la priorisation se définit comme suit.

> *Soit C l'ensemble des classes candidates au test, PC l'ensemble des permutations de C et P_i la permutation i. Si l'on souhaite maximiser un critère dont la fonction objective associée est f, on recherche P_j tel que $f(P_j) \geq f(P_i)$ toutes les permutations telles que $i \neq j$.*

Le problème donné par la recherche d'une permutation optimale est un problème difficile (non résoluble en temps polynomial). Il est donc impossible de tester chacune des permutations, car pour un système contenant seulement 150 classes candidates, plus de $5,7 \times 10^{262}$ permutations sont à explorer. Nous procéderons plutôt avec une heuristique visant à construire une séquence dans l'ordre optimal.

Étapes de l'approche développée



La fonction objective utilisée pour évaluer la séquence est une fonction qui considère la couverture de la classe. Plus une classe est de grande taille ou requiert le service de classes de grande taille, meilleure est sa couverture. Comme notre approche proposée est naïve, nous ne tiendrons pas compte des répétitions dans la couverture.

Métriques logicielles

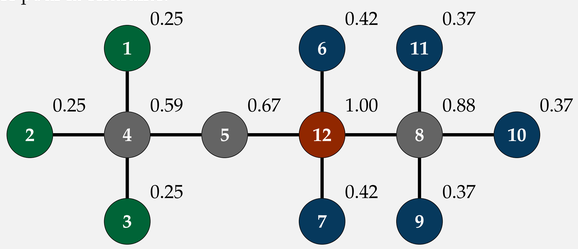
Afin de développer une approche automatique, il nous faut trouver des façons de quantifier les classes et leur complexité. Ces quantificateurs sont appelés métriques dans le domaine du génie logiciel. Trois métriques sont utilisées dans la suite. Les deux premières seront utilisées dans notre approche, tandis que la troisième servira aux fins d'évaluation.

1. Complexité cyclomatique moyenne (AVG_CC) : indique le nombre de chemins d'exécution moyen des méthodes ;
2. Nombre de méthodes dans la classe (WMC) : compte le nombre de méthodes dans la classe ;
3. Nombre de ligne de code (LOC) : compte le nombre de lignes de code total de la classe.

Mesures de centralité

Les mesures de centralité ont été développées en premier lieu dans l'étude des réseaux sociaux. Les systèmes orientés objet partagent certaines caractéristiques en lien avec ces réseaux. Il est possible de représenter un logiciel sous forme de graphe de dépendances. On assigne chaque classe à un nœud et si deux classes interagissent entre elles, on ajoute un arc entre les nœuds associés à ces classes.

La mesure de centralité utilisée est celle des vecteurs propres. Cette centralité est plus élevée pour les classes ayant de nombreux liens ou qui sont en lien avec des classes qui ont de nombreux liens. Toutefois, dans le deuxième cas, plus une classe est éloignée, moins elle a d'importance pour la centralité.



On peut remarquer nombre d'informations dans l'exemple qui nous renseigne sur le comportement de la mesure de centralité. D'abord, plusieurs valeurs sont identiques, conséquence de la symétrie du graphe (ex : nœuds 1, 2 et 3). Ensuite, les nœuds 4, 8 et 12 ont tous le même nombre de voisins, mais le nœuds 12 affiche une position plus centrale d'un point de vue visuel et cela se reflète dans la valeur associée à sa mesure de centralité. On remarquera également que même s'il possède moins de voisins, la centralité du nœud 5 est plus élevé que celle du nœud 4 en raison de sa proximité avec le nœud 12.

Approche de priorisation

Afin de sélectionner le sous-ensemble de classes candidates au test et de déterminer lesquelles sont plus prioritaires, nous avons proposé un règle de classification. Cette règle considère les métriques et les mesures à pondération égale et produit un score dans l'intervalle $[0, 1]$. Le score représente l'ordre de la classe parmi les classes à tester (objectif de priorisation). Ensuite, nous déterminons un seuil ϕ et toutes les classes qui ont un résultat égal ou supérieur à ϕ seront dans l'ensemble des classes candidates (objectif de sélection).

La règle de priorisation travaille sur les données transformées de deux façons. La première est par normalisation. La normalisation consiste à diviser une valeur par la plus grande valeur de la série de données. Cette technique ramène les données sur un intervalle commun tout en préservant la dispersion. Le second est par le passage au rang. Ce processus remplace la donnée par sa position dans l'ensemble des données. Le passage au rang préserve uniquement l'ordonnancement des individus et abolit la dispersion.

Nous notons la fonction de passage au rang par $\mathcal{R}$ et les données normalisées en leur ajoutant l'indice N. La règle utilisée pour déterminer le score est :

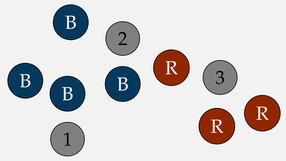
$$\phi = \frac{\mathcal{R}(WMC_N + AVG\_CC_N) + \mathcal{R}(EV)}{2}$$

Approche de priorisation (suite)

À la lumière de divers tests statistiques menés sur les données, nous avons choisi de retenir 0.82 comme seuil de notre approche. Donc, nous proposons de tester les 18% des classes ayant le score le plus élevé. Remarquons aussi que notre règle prend en compte les deux aspects que nous recherchions dans notre approche. Les métriques de code source sélectionnées sont réputées pour bien repérer la présence de fautes dans un logiciel. La mesure de centralité favorise les classes avec une forte couverture du code source.

Comparaison

Nous avons testé notre approche sur un logiciel, ANT, comportant 745 classes. Nous avons traité les mêmes données avec l'algorithme d'apprentissage artificiel k-NN qui a été utilisé dans de nombreux travaux connexes. Cet algorithme recherche les voisins de l'instance et lui assigne la même catégorie que celle qui est majoritaire. Les voisins sont calculées avec une variante de la distance euclidienne qui évite de biaiser les résultats dans le cas multidimensionnel.



Supposons que nous connaissons des exemples de deux catégories, points bleus et rouges et que nous souhaitons classifier les points gris. Les points 1 et 2 seront dans la catégorie bleue et le 3 dans la catégorie rouge

Nous avons ensuite comparé notre approche avec les résultats obtenus par l'algorithme k-NN, afin de valider notre approche. Le tableau ci-dessous présente les résultats des deux approches. Remarquons que k-NN teste davantage de classe ce qui induit une augmentation de la couverture, mais également de l'effort. Aussi, au niveau de la détection des fautes k-NN est évalué sur son ensemble d'apprentissage ce qui biaise les résultats positivement (plus efficace que la performance réelle).

	Approche développée	**k-NN**
Nombre de classes testées	139	159
Classes fautives	50/69	69/69
% de lignes de code couvertes	68%	72%

Des résultats préliminaires sur la performance réelle de k-NN présente un ratio de classes fautives plus près du 50% (la moitié des classes ayant des fautes sont testées).

Conclusion et limites

En conclusion, notre approche pour la priorisation des tests quoique simple est prometteuse. Les résultats que nous obtenons en comparant notre approche avec l'algorithme d'apprentissage artificiel k-NN sont très acceptables. Nos résultats ne se généralisent pas sur d'autres logiciels. Cependant, nous avons fait quelques expérimentations préliminaires avec des données provenant d'autres logiciels et la tendance semble se confirmer.

Nous voudrions raffiner notre approche afin d'obtenir de meilleurs résultats. Deux approches pourront être explorées dans des travaux futurs. La première est de développer un algorithme utilisant l'intelligence artificielle afin d'obtenir une meilleure séquence. Cette approche pourrait aussi explorer le graphe afin de limiter les redondances dans la couverture des tests. La seconde approche à explorer est d'utiliser les algorithmes génétiques. Cette famille d'algorithmes est réputée pour être très efficace dans la recherche de solutions optimales de problèmes d'optimisation complexes.

Références

Ma, W., Chen, L., Yang, Y., Zhou, Y., & Xu, B. (2016). Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology*, 69, 50-70.

Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization : A family of empirical studies. *IEEE transactions on software engineering*, *28(2)*, 159-182.

Zimmermann, T., & Nagappan, N. (2009, April). Predicting defects using network analysis on dependency graphs. *In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on (pp. 531-540)*. IEEE.