Futur d'une infrastructure de correction automatisée : CodeGradX

Christian Queinnec, Olivier Pons

Futur d'une infrastructure de correction automatisée : CodeGradX

Christian Queinnec[1] and Olivier Pons[2]

[1] Sorbonne Universités, UPMC Univ Paris 06, Laboratoire d'Informatique de Paris 6 (LIP6), 4 place Jussieu, 75252 Paris Cedex 05, France christian.queinnec@upmc.fr
[2] CNAM, Laboratoire CEDRIC, 292 rue Saint Martin, 75141 Paris Cedex 03, France olivier.pons@cnam.fr

Résumé Cet article évoque quelques expériences et nouvelles idées autour d'une infrastructure de correction automatisée de programmes.

CodeGradX est une infrastructure de correction automatisée de programmes. Elle est en service depuis 2008 (initialement sous le nom de FW4EX) et a depuis corrigé plus de 150 000 soumissions d'étudiants à des exercices principalement en Scheme, utilitaires d'Unix, JavaScript mais aussi en C, Octave, O'Caml, Python sans oublier les compétitions annuelles des Journées Franciliennes de Programmation.

L'infrastructure repose sur une constellation de serveurs indépendants afin d'assurer une bonne montée en charge. Depuis 2008, l'infrastructure a gagné en robustesse tant du côté de la résistance aux programmes à corriger, erronés voire malveillants que du côté des pannes de serveurs ou de réseau afin d'assurer le service sans perte et le plus continu possible.

Ce papier expose quelques-unes des idées pour la mise en œuvre de la correction automatisée vue des apprenants ou des enseignants, recense certaines expériences menées ces dernières années et les questionnements qui en découlent enfin, présente quelques nouvelles pistes de recherche que nous souhaitons entreprendre.

1 Idéologie

Dans cette section, nous nous intéressons à deux idées contraignantes : la facilité de déploiement et l'importance des tests.

1.1 Facilité de déploiement

Lorsque les apprenants sont novices, sont dotés de machines multiples, diversement configurées, alors l'usage d'un correcteur automatisé doit rester simpliste. Ce qui implique (au regard des techniques actuelles) que leur navigateur est l'environnement d'écriture des programmes qu'ils soumettront. Il faut donc disposer, au moins, dans le navigateur, d'un éditeur et d'un metteur au point ce qui est une limitation forte sur les langages de programmation disponibles.

Inversement, pour des apprenants déjà rompus à la programmation, les priver de leur environnement favori (leur IDE) est contre-productif. Une solution alors est de procurer un greffon permettant à cet environnement d'interagir avec l'infrastructure de correction automatisée.

Une nouveauté de CodeGradX, récemment apparue, est une bibliothèque d'interaction (une API). Écrite en JavaScript, elle peut s'exécuter dans un navigateur ou en ligne de commande et simplifie ainsi le couplage d'un navigateur ou d'un IDE avec CodeGradX. Cette API se repose sur les protocoles REST sous-jacents d'accès à l'infrastructure CodeGradX en place depuis le début.

1.2 Importance des tests

Même si aujourd'hui, le courant JUnit déclare à l'envi que les développeurs aiment tester, nous croyons que les exercices de programmation doivent exiger d'être accompagnés de tests et qu'inculquer cette habitude dès le début des études a des vertus morales indéniables.

Une première contrainte induite est que l'apprenant doit écrire ses tests en respectant les règles de l'architecture de tests retenue ce qui augmente le nombre de concepts à maîtriser, au moins partiellement, lors de l'écriture des premiers programmes. Proposer un canevas de solution, pour les premiers exercices, dont seules quelques parties sont à remplir, résoud ce problème.

La seconde contrainte s'en déduit : la correction automatisée doit aussi corriger les tests de l'apprenant. Pour cela, nous employons une technique de mesure de couverture de code. Si l'on nomme f_s et t_s le code de l'apprenant et ses tests, f_t et t_t le code et les tests de l'enseignant. La correction s'effectue par comparaison du code de l'apprenant avec celui de l'enseignant. Ainsi

- on vérifie la cohérence du code et des tests de l'apprenant avec $t_s(f_s)$. On rejette la soumission en cas d'incohérence.
- on vérifie que l'apprenant satisfait les tests de l'enseignant avec $t_t(f_s)$. on peut compter les tests réussis et ratés pour obtenir une note.
- on compare les couvertures de code obtenues (sur f_s) avec $t_s(f_s)$ et $t_t(f_s)$. Le but est que les tests de l'apprenant couvre au moins autant de son code que les tests de l'enseignant.

Il faut donc être capable de mesurer des taux de couverture. Une bonne solution que nous avons éprouvée pour Scheme et sommes en train de mettre en œuvre pour JavaScript est de disposer d'un interprète du langage (ou du sous-ensemble enseigné) ce qui permet d'en déduire, assez simplement, un metteur au point pas à pas et un mesureur de couverture.

Une troisième contrainte existe, plus subtile. Quelle peut être la valeur de $t_s(f_t)$? Si le code de l'enseignant ne satisfait pas les tests de l'apprenant, ce peut être pour plusieurs raisons : l'apprenant traite un problème différent que celui suggéré par l'énoncé, le code de l'enseignant est incorrect (les tests de l'apprenant sont meilleurs que ceux de l'enseignant) ou encore, les tests de l'étudiants couvre des cas hors épure c'est-à-dire non spécifiés par l'énoncé.

Les cas hors épure peuvent être détectés par des tests appropriés mais cela complexifie l'écriture des tests de l'enseignant et ne permet en général plus de montrer le code de l'enseignant comme exemple de solution.

2 Expériences

Aider les apprenants à progresser seuls est l'un des défis des MOOC.

2.1 Coup de pouce

Lors de la seconde édition du MOOC Programmation récursive, nous avons introduit la notion de « coup de pouce ». Lorsqu'un apprenant obtient une certaine note, le bouton « coup de pouce » lui permet de voir une autre proposition de solution avec une note légèrement supérieure. Étudier cet autre programme et comprendre pourquoi il est meilleur est un exercice de lecture de code (une activité peu enseignée). En retour, il est demandé à l'apprenant d'indiquer si ce programme lui a été bénéfique ou pas. L'idée est, peu à peu, d'identifier les solutions intéressantes à montrer.

Le mécanisme « coup de pouce » rebaptisé « autre solution » est aussi disponible pour les étudiants ayant réussi un exercice et leur permet de découvrir d'autres solutions (une demande formulée lors de la première édition du MOOC Programmation récursive). En retour, il leur est demandé de juger si l'autre solution leur semble meilleure que la leur. L'idée étant, là encore, d'identifier les meilleures solutions.

Pour le mécanisme « coup de pouce » on tombe sur le problème de la taxonomie des erreurs car la note actuelle d'un apprenant ne permet pas de déterminer quelle autre proposition de solution lui montrer parmi toutes celles qui existent.

2.2 Duo ou duels

Afin d'attiser une certaine émulation entre les apprenants, nous avons introduit une notion de classement inspiré par le classement ELO des échecs et les méthodes dérivées utilisées par les jeux vidéo. Le classement introduit considère chaque proposition de solution comme un jeu où l'apprenant bat tous les autres apprenants ayant une note inférieure et en cas d'égalité bat tous les apprenants qui ont soumis plus d'essais. L'apprenant parfait réussit tous les exercices du premier coup, le pire apprenant (virtuel) rate tous les exercices un nombre non borné de fois.

Outre la possibilité de classer un apprenant parmi un groupe d'apprenants (groupe décidé par un enseignant) et de lui montrer sa position, le classement peut aussi servir à proposer un partenaire (ou un adversaire) à la mesure d'un apprenant.

Le travail en groupe est intéressant lorsque ses membres sont de niveau comparable. Un serveur de binôme pourrait proposer un tel partenariat pour un travail en binôme. Inversement, dans le cas d'un duel où deux apprenants se

lancent le défi de résoudre un exercice dans le moindre temps, proposer un adversaire de force comparable est souhaitable pour l'intérêt du duel.

Malheureusement le travail en binôme ne peut fonctionner que s'il y a un certain synchronisme ce qui peut être le cas dans une classe, un MOOC synchrone ou encore dans un MOOC asynchrone pléthorique afin d'assurer une bonne probabilité de trouver un partenaire.

3 Futur

Pour certains exercices, nous disposons de milliers de propositions de solutions. De cette masse de données, l'on doit pouvoir extraire d'intéressantes choses.

La première serait d'arriver à une taxonomie des erreurs les plus fréquentes. Malheureusement, il n'existe pas de forme canonique pour les programmes ni même pour les traces d'exécution que pourrait récupérer un interprète dûment instrumenté. Disposer d'un interprète instrumenté permet néanmoins, souvent, de déterminer la complexité de l'algorithmique mise en œuvre et ainsi d'améliorer les retours vers l'apprenant.

Une autre serait d'identifier, au sein d'une séquence ordonnée d'exercices à faire, les irrégularités de progression ce qui permettrait d'identifier à la fois les exercices trop similaires aux précédents ou trop ardus. Inversement, une fois qu'une séquence a été expérimentée, tout écart d'un apprenant vis-à-vis de la moyenne des comportements observés peut être révélateur de difficultés.

Une autre direction que nous souhaitons investiguer, à une époque où les téléphones ou tablettes favorisent la lecture et le choix tactile plutôt que l'écriture de programmes, serait d'afficher des énoncés d'exercices et des propositions de solution et de demander si la solution est correcte, quelle note elle pourrait avoir, etc.

Bref exploiter cette masse de données ouvre de larges horizons de recherche.