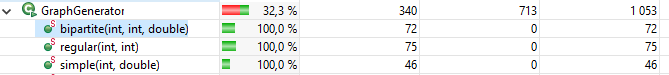
Rapport TP3

Vincent Chassé 1795836 Gabriel Bourgault 1794069

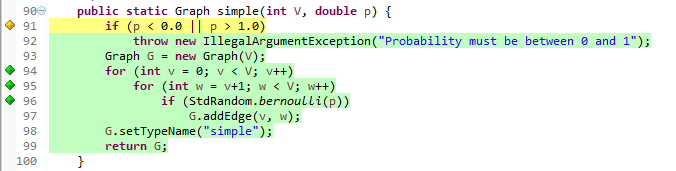
Lors des TP1 et TP2, nous nous sommes familiarisés avec les tests unitaires ainsi que la division des paramètres d’entrée en différentes classes afin de respecter les principes Each Choice (EC) et All Combinations (AC). Cela nous a permis de découvrir certains bogues dans le code de la classe GraphGenerator par rapport à la spécification. Lors du TP3, nous avons pu mettre à l’épreuve nos tests écrits lors du TP2 en analysant la couverture de code. Comme nous allons l’expliquer plus bas, nos tests boîte noire couvraient déjà toutes les lignes, mais pas nécessairement toutes les branches.

# Partie 1 – Couverture des tests AC

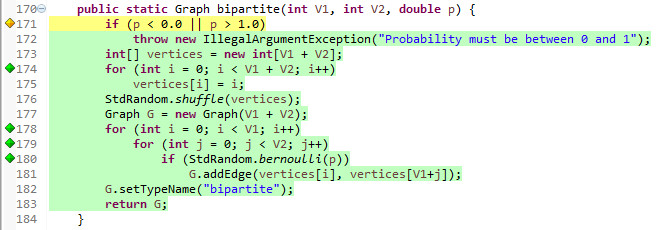
Premièrement, voici les résultats de la couverture de code par Jacoco sur nos tests boîte noire :



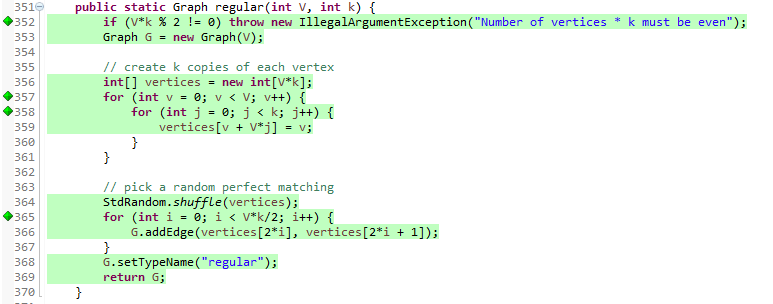
On remarque que 100% des lignes sont couvertes, mais en regardant les fonctions, on s’aperçoit que certaines branches ne sont pas couvertes. En effet, pour la fonction « simple », nos tests n’envoyaient pas un p plus petit que 0, alors p < 0.0 n’était jamais vrai et cette branche n’était jamais traversée.



De même, pour la fonction « bipartite », nous n’envoyions jamais de p plus petit que 0.



Cependant, nos tests pour la fonction « regular » permettaient d’atteindre 100% de couverture.

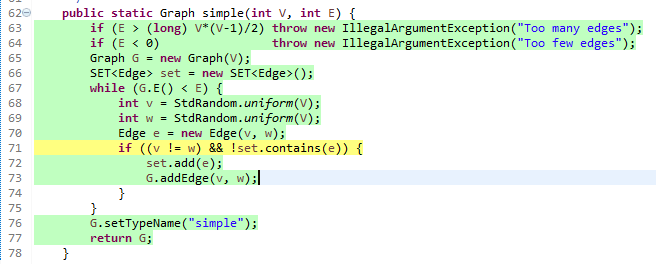


Pour assurer la couverture complète, nous avons donc ajouté un test pour « simple » et un test pour « bipartite » où nous envoyions un p négatif. Ces tests sont testSimplePNegatif et testBipartitePNegatif. Après avoir ajouté ces 2 tests, Jacoco nous a confirmé que notre couverture était bien de 100% des branches.

# Partie 2 – Couverture 100% des branches

Après avoir garanti la couverture complète des fonctions testés dans le TP2, nous avons écrit d’autres tests pour couvrir l’ensemble des branches de toutes les fonctions de GraphGenerator qui retourne un graphe. Notre méthode pour concevoir les tests boîtes blanche était simplement de regarder les parcours possibles à l’intérieur d’une fonction et de trouver les paramètres devant être envoyé à la fonction pour que ce parcours soit emprunté. On répète tant qu’il reste une branche à couvrir.

*simple(int V, int E) :* Dans cette fonction, il y a deux conditions au début qui vérifient la validité de E. Pour passer à travers les branches, nous avons envoyé un E > V\*(V-1)/2 dans testSimpleCouverture1 et un E négatif dans testSimpleCouverture2. La boucle while est toujours parcourue lorsque le E est valide, mais la ligne 71 contient une condition avec deux prédicats qu’on doit manipuler pour assurer la couverture. Cependant, il est impossible de garantir la couverture complète la ligne 71, car la fonction utilise des valeurs générées aléatoirement. Il est possible d’atteindre toutes les branches, mais cela dépend beaucoup de la chance. En effet, il faudrait que la valeur aléatoire de v et w soient égales une fois et qu’une même paire de v et w soit générée deux fois. Une solution possible serait d’encapsuler le générateur aléatoire afin de pouvoir le remplacer (mock) pour les tests unitaires.



*complete(int V) :* Un seul test est nécessaire pour couvrir cette fonction. Le test testCompleteCouverture1 envoie un V = 3 et cela couvre bien toute la fonction.

*completeBipartite (int V) :* Un seul test est nécessaire pour couvrir cette fonction. Le test testCompleteBipartiteCouverture1 envoie un V1 = 3 et un V2 = 5, ce qui couvre bien toute la fonction. D’ailleurs, ce test couvre aussi la partie de la fonction bipartite(int V1, int V2, int E) lorsque les paramètres sont valides.

*bipartite(int V1, int V2, int E) :* Dans cette fonction, il y a deux conditions au début qui vérifient la validité de E. Pour passer à travers les branches, nous avons envoyé un E > V\*(V-1)/2 dans testBipartiteCouverture1 et un E négatif dans testBipartiteCouverture2. La boucle while est toujours parcourue lorsque le E est valide, toute la fonction est alors couverte.

*path(int V) :* Un seul test est nécessaire pour couvrir cette fonction. Le test testPathCouverture1 envoie un V = 3 et cela couvre bien toute la fonction.

*binaryTree(int V) :* Un seul test est nécessaire pour couvrir cette fonction. Le test testBinaryTreeCouverture1 envoie un V = 3 et cela couvre bien toute la fonction.

*cycle(int V) :* Un seul test est nécessaire pour couvrir cette fonction. Le test testCycleCouverture1 envoie un V = 3 et cela couvre bien toute la fonction.

*eulerianCycle(int V, int E):* Dans cette fonction, il y a deux conditions au début qui vérifient la validité de E et de V. Pour passer à travers les branches, nous avons envoyé un E <= 0 dans testEulerianCycleCouverture2 et un V <= 0 dans testEulerianCycleCouverture3. Dans le test testEulerianCycleCouverture1, nous envoyons un E et V valides, ce qui couvre le reste de la fonction.

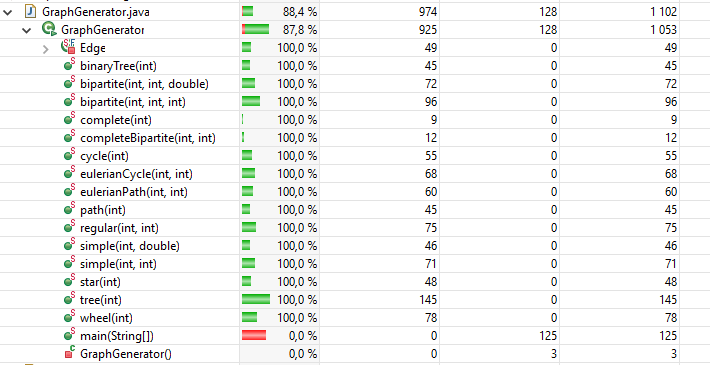
*eulerianPath(int V, int E):* Dans cette fonction, il y a deux conditions au début qui vérifient la validité de E et de V. Pour passer à travers les branches, nous avons envoyé un E <= 0 dans testEulerianPathCouverture2 et un V <= 0 dans testEulerianPathCouverture3. Dans le test testEulerianPathCouverture1, nous envoyons un E et V valides, ce qui couvre le reste de la fonction.

*wheel(int V):* Dans cette fonction, il y a une condition au début qui vérifie la validité de V. Pour passer à travers les branches, nous avons envoyé un V <= 1 dans testWheelCouverture1. Dans le test testWheelCouverture1, nous envoyons un V valide, ce qui couvre le reste de la fonction.

*star(int V):* Dans cette fonction, il y a une condition au début qui vérifie la validité de V. Pour passer à travers les branches, nous avons envoyé un V <= 1 dans testStarCouverture2. Dans le test testStarCouverture1, nous envoyons un V valide, ce qui couvre le reste de la fonction.

*tree(int V):* Dans cette fonction, il y a une condition au début qui vérifie un cas spécial de V == 1. Pour passer à travers les branches, nous avons envoyé un V == 1 dans testTreeCouverture2. Dans le test testTreeCouverture1, nous envoyons un V différent de 1 mais toujours valide, ce qui couvre le reste de la fonction.

On voit ici la couverture des lignes après l’ajout des tests boîte blanche :



On voit bien que toutes les lignes sont maintenant couvertes pour chaque fonction de la classe GraphGenerator retournant un graphe. En observant chaque fonction, on voit qu’il n’y a aucune branche non testée, sauf pour le cas de *simple(int V, int E)*précédemment mentionné. Cela peut aussi être constaté dans le rapport Jacoco effectué après avoir créé tous ces tests (rapport\_jacoco\_final).