Nada Alem | Alessandra Mancas

Choix général des méthodes:

En se documentant sur le projet Graphhopper, nous avons compris que le projet porte sur le routage et sur le calcul d'itinéraires optimisés. Puisqu'un des algorithmes de génération de trajet optimisés est Dijkstra, il nous a semblé pertinent de tester les structures de données sur lesquelles les algorithmes de routage et d'extraction de trajets sont exécutés. Les points d'une carte sont représentés sous forme de graphe, dans ce projet. Les routes, intersections et autres éléments géographiques sont modélisés à l'aide de **nœuds** et **arêtes**, suivant la structure classique des graphes.
Pour cette raison, nous avons choisi des classes qui sont liées à la structure de données Graphe mais aussi les méthodes de celles-ci qui n'ont pas suffisamment de coverage.
Nous avons également choisi les méthodes à tester par nombre d'occurrences/usages dans le projet. En effet, nous utilisons IntelliJ IDEA pour travailler sur ce projet. Cet IDE nous permet de voir tous les usages d'une méthode ou d'une classe données, ce qui nous a beaucoup aiguillé dans le choix des méthodes, car plus le nombre d'usages est élevé, plus il nous semblait pertinent de tester le comportement de cette méthode. De plus, dû à la couverture déjà élevée de Graphhopper et de notre manque de familiarité avec le projet, nous avons choisi des méthodes qui ne travaillent pas avec trop d'objets autres que Graph. En effet, Graphhopper crée ses propres structures de données, mais fournit très peu de documentation pour les comprendre, ce qui aurait rendu les tests très complexes si on devait les implémenter ou même en faire des *mock*.
Par ailleurs, certaines des méthodes que nous testons dans ce devoir sont définies comme *private* ou *protected*. Malgré leur niveau de visibilité, nous argumentons que tant qu'elles sont appelées quelque part dans le programme, elles valent la peine d'être testées car leur comportement affecte le comportement général de l'application, même si elles ont moins de chances d'échouer dû à un *input* externe aberrant.

Classe: GHCUtility

En regardant le rapport de couverture généré par Jacoco, nous avons remarqué que cette classe contient non seulement beaucoup de méthodes qui sont utilisées mais pas

encore couvertes par les tests, mais également des méthodes qui ne travaillent pas sur des structures de données trop complexes. Ainsi, on a pu en tester une bonne partie.

Méthode testée #1 :***getCommonNode()*** :
*core/src/main/java/com/graphhopper/util/GHCUtility.java/****getCommonNode()***
Cette méthode retourne le nœud commun entre deux arêtes dans un graphe. Les arêtes doivent être connectées à exactement un nœud distinct, et la méthode retournera ce nœud.
Si les arêtes ne sont pas connectées, si l'une des arêtes forme une boucle (son nœud de base est le même que son nœud adjacent), ou si les deux arêtes forment un cercle (les deux arêtes partagent les mêmes nœuds), une IllegalArgumentException est levée.
Les paramètres de la méthode sont le graph (BaseGraph implémentation principale de l'interface graphe dans le projet), la première arête et la deuxième arête. Le retour attendu est soit le nœud commun ou les deux arêtes sont connectés ou une exception.
Le coverage de cette méthode ne couvrait pas tous les cas et ne testait pas un cas sans problème et un cas qui retournerait l'exception attendue.

** *La structure de données utilisée pour les méthodes qui s'exécutent sur un graphe est la classe BaseGraph puisque c'est dans le projet l'implémentation principale de l'interface Graph. La structure créée pour les tests est un 'mock' d'un graphe, aucune réelle données géographiques n'est ajoutée à la structure. L'instanciation est simple et comporte que les éléments nécessaires au comportement de la méthode.*

Tests écrit pour évaluer le comportement cette méthode :
*core/src/main/java/com/graphhopper/util/GHCUtilityTest.java/****getCommonNodeTest()***
:
*core/src/main/java/com/graphhopper/util/GHCUtilityTest.java/****testGetCommonNode ThrowsIllegalArgumentException()*****:**
Ces deux méthodes testent le comportement de la méthode getCommonNode().

getCommonNodeTest() :
Arrange:

- Construire un BaseGraph graph sans réelles données géographiques (Mock)

Act:

- Appeler la méthode avec comme paramètres le BaseGraph graph et le id de 2 arêtes

Assert:

- Vérifier que le retour de la méthode est l'identifiant du nœud attendu.

testGetCommonNodeThrowsIllegalArgumentException():

Arrange:

- Construire un BaseGraph graph sans réelles données géographiques (Mock)

Act:

- Appeler la méthode avec comme paramètres le BaseGraph graph et le id de 2 arêtes identiques (pour provoquer l'erreur attendu)

Assert:

- Vérifier que le message d'erreur est retourné 'IllegalArgumentException e'

Note:

Pour les méthodes de tests qui attendent un message d'exception comme ***testGetCommonNodeThrowsIllegalArgumentException()*** , la structure de la méthode a été prise de cette discussion sur le site stackoverflow : [forum]

La structure en question :

```
try {
   int result = appel de la methode testée
   fail("expected exception was not occured.");
} catch(IllegalArgumentException e) {
   //if execution reaches here,
   //it indicates this exception was occured.
   //so we need not handle it.
   System.out.println("Excepted exception is handled");
}
```

Méthode testée : getProblems() :

Cette méthode vérifie un graphe pour détecter des erreurs possibles liées aux nœuds et aux arêtes. Elle renvoie une liste de messages décrivant les problèmes trouvés. Si une

erreur se produit pendant l'exécution, une exception est levée, précisant quel nœud a causé l'erreur.

testGetProblems():

Arrange:

- Construire un BaseGraph graph sans réelles données géographiques (Mock)

Act:

- Appeler la méthode avec comme paramètres le BaseGraph graph

Assert:

- Vérifier que la liste de chaine de caractère avec tous les problèmes est retournée

Test 1: testComparePaths_normal()

Méthode testée: comparePaths()

*core/src/main/java/com/graphhopper/util/GHCUtility.java/**comparePaths()***

La méthode comparePaths est appelée dans le cadre de la méthode *RandomizedRoutingTest.randomGraph*(*FixtureSupplier fixtureSupplier*). Elle permet de comparer deux objets Path et elle s'assure que leurs attributs (poids, distance, temps et liste de nœuds) sont corrects et quasi-égaux (car deux chemins seront considérés égaux si la différence de leurs distances est à moins de 0.01 de différence). Si les chemins sont, d'une manière ou d'une autre, inégaux, la méthode soit lancera un AssertionError, soit elle ajoutera une phrase indiquant l'erreur à une liste nommée strictViolations, qu'elle retournera par la suite.

Le test appelant cette méthode compare deux algorithmes de routage sur un graphe créé aléatoirement (randomGraph) en vérifiant si strictViolations est vide.

Malgré le fait que cette méthode est appelée en contexte de test et non lors du roulement normal du programme, nous considérons quand même qu'elle est pertinente car si son retour est incorrect, le test sera incorrect.

Nous avons découpé les tests de la méthode en deux cas afin d'améliorer la lisibilité.

core/src/test/java/com/graphhopper/util/GHCUtilityTest.java

/ testComparePaths_normal()

Nous testerons d'abord le comportement normal de la méthode, c'est-à-dire quand elle retourne strictViolations vide.

Nous avons testé deux cas:

- Les Paths sont quasi-égaux (très peu de différence entre elles, pas assez pour déclencher les inégalités)
- Les Paths sont nuls (ils seront traités comme égaux)

Arrange:

- Construire le graphe et définir les paths avec les valeurs voulues selon le cas de test

Act:

- Appeler la méthode sur les paths voulus

Assert:

- Évaluer sa sortie (la liste vide)

Test 2: testComparePaths_errors()

core/src/test/java/com/graphhopper/util/GHCUtilityTest.java

/ testComparePaths_errors()

Ensuite, nous testons le comportement de comparePaths() lorsqu'elle lance une erreur ou elle retourne une liste non-vide.

Nous avons testé deux cas:

- Les Paths ont des poids drastiquement différents (pour voir si la méthode lance bien le AssertionError)
- Les Paths ont le même poids, mais des distances & temps drastiquement différents (pour voir si la méthode retourne bien la liste contenant les bons messages)

Arrange:

- Construire le graphe et définir les paths avec les valeurs voulues selon le cas de test

Act:

- Appeler la méthode sur les paths voulus

Assert:

- Évaluer sa sortie (la liste ou le lancement d'une AssertionError)

Méthode testée: updateDistancesFor(Graph g, int node, double ... latlonele)
*core/src/main/java/com/graphhopper/util/GHCUtility.java/**updateDistancesFor***

La méthode updateDistancesFor est appelée beaucoup de fois à travers le programme, notamment, encore une fois, dans le cadre de tests unitaires. Elle modifie les coordonnées d'un nœud donné dans un graphe passé en argument selon la latitude, la longitude et l'élévation qu'on passe également en argument. Après la mise à jour des coordonnées, elle met à jour la longueur de toutes les arêtes concernant le nœud donné. Elle ne retourne rien, car on travaille directement sur l'objet du graphe qu'on veut mettre à jour grâce à l'instance NodeAccess qui est associée à un graphe unique et au EdgeIterator qui modifie directemetn les arêtes du graphe.
Dans la couverture du code, cette fonction était déjà partiellement testée. Nous avons donc écrit deux tests pour couvrir les branches qui doivent lancer des erreurs, car aucun cas de test ne les couvrait encore.
Dans nos tests, nous voulons voir si elle lance les bonnes erreurs selon l'appel de la fonction

Test 1: testUpdateDistancesFor_invalidArgs()
core/src/test/java/com/graphhopper/util/GHCUtilityTest.java
/testUpdateDistancesFor_invalidArgs()
Dans ce cas de test, nous appelons la fonction avec un graphe 2D en argument, mais avec la longitude manquante, ce qui est incorrect.
Avec cette structure suivante nous avons testé deux cas:

- Une seule valeur passée pour latlonele
- Aucune valeur passée pour latlonele

Arrange:

- Construire un Graph g quelconque sur lequel travailler

Act:

- Appeler la méthode avec le mauvais nombre d'arguments

Assert:

- Vérifier qu'une IllegalArgumentException est lancée
- Vérifier que le message de l'erreur est correct

- “illegal number of arguments 1”
- “illegal number of arguments 0”

On a choisi de ne pas tester pour un *node* indéfini, car la méthode ne semble pas avoir de manière explicite de gérer ce cas, alors elle lancera probablement un simple IllegalArgumentError.
De même, la méthode ne gère pas encore le cas où on lui donne plus que le nombre attendu (>2 si 2D, >3 si 3D) pour définir un nœud, alors nous n’avons pas testé cette situation.

Test 2: testUpdateDistancesFor_3DInvalidInput()
core/src/test/java/com/graphhopper/util/GHCUtilityTest.java
/testUpdateDistancesFor_3D()
Dans ce deuxième cas de test, nous évaluons le comportment de l’autre branche qui n’était pas encore couverte. La fonction devrait lancer une erreur si on travaille sur un graphe 3D mais on lui donne le mauvais nombre d’arguments. En effet, si le graphe est en 3D, chaque nœud est également défini par son élévation et cette donnée est importante dans le calcul de la distance. Nous avons donc appelé la méthode sans préciser l’élévation en attendant qu’elle nous lance un IllegalArgumentException propre à cette situation, et non simplement l’erreur liée au mauvais nombre d’arguments.
Arrange:

- Construire un Graph g en 3D

Act:

- Appeler la méthode sans préciser l’élévation

Assert:

- Vérifier qu’une IllegalArgumentException est lancé
- Vérifier que le message d’erreur est correct
    - “graph requires elevation”

Classe AngleCalc.java :

Cette classe est utilisée pour faire des calculs d'angles et d'orientations géographiques. Plusieurs méthodes y sont définies pour manipuler, convertir des angles et pour analyser la relation angulaire entre plusieurs points géographiques. Les méthodes de cette classe permettent entre autres de communiquer avec l'utilisateur dans un langage plus naturel et intuitif.

Méthode : **azimuth2compassPoint()**
Cette méthode permet d'afficher les directions en termes familiers lors de la navigation. Par exemple, lorsque l'utilisateur doit tourner ou suivre une route, il est plus intuitif de dire "Tournez vers le nord-est" plutôt que de donner un angle exact comme "Tournez à 45°". Cela rend l'expérience utilisateur plus naturelle et accessible.
La méthode prend en entrée un double représentant l'angle et retourne une chaîne de caractères représentant le point cardinal.
Cette méthode n'a pas été testée alors qu'il y a beaucoup de conditions if à tester, ce qui représenterait un cas de branch testing.
Arrange:

- Passer en paramètres une mesure d'angle aléatoire.

Act:

- Appeler la méthode avec comme paramètres un angle pour chaque point cardinal possiblement attendu.

Assert:

- Vérifier que le bon point cardinal est retourné pour l'angle en paramètre.

Méthode : **ConvertAzimuth2xAxisAngle**
Cette méthode convertit un angle mesuré depuis le nord (un azimuth) en un angle par rapport à l'axe X, où l'est correspond à 0 radians et l'angle peut être positif ou négatif selon la direction. Cette méthode avait initialement une couverture de test mais pas suffisamment puisqu'elle permet de faire une opération assez importante dans le calcul des directions et de l'orientation. Nous avons donc ajouté des tests pour compléter la couverture de if conditions et pour augmenter et nous avons également ajouté un cas de test dans cette même méthode pour couvrir la gestion d'exception.
Arrange:

- Aucune préparation requise.

Act:

- Appeler la méthode avec comme paramètres une mesure d'angle mesuré depuis le nord (40, 315 et 400 degré).

Assert

- S'assurer que le message d'exception est retourné, que les bons cadrans sont utilisés dans la coordonnée en X retournée.

Classe BaseGraph.java :

Méthode #2 testée : **Edge():**

path : *core/src/main/java/com/graphhopper*/storage/BaseGraphTest

Explication de la méthode :

Cette méthode crée et retourne une nouvelle arête entre 2 noeuds spécifiés en paramètres (Node A, Node B) dans le graphe. Si le graphe est frozen, ou si les 2 noeuds spécifiés dans les paramètres sont identiques, la méthode devrait retourner un message d'exception. Sinon, la nouvelle arête devrait être renvoyée. Cette méthode est pertinente à tester car elle est utilisée très fréquemment, et c'est une des méthodes fondamentales pour la définition d'un graphe juste et valide pour l'exécution des algorithmes de routage. Le manque de test pourrait générer des erreurs très évitables et triviales mais qui pourraient causer plusieurs heures perdues de débogage. Il est important d'évaluer le comportement de la méthode qui crée les nœuds et les arêtes des graphes, pour s'assurer que le comportement en cas de conflit est bien généré comme par exemple en cas de conflit ou les coordonnées de 2 points sont identiques. Cela peut être un problème si ces deux points représentent des entités distinctes dans la vraie vie, comme deux routes ou intersections différentes qui se trouvent théoriquement au même endroit (ex : un pont au-dessus d'une route, ou des sorties empilées). Pour l'utilisation locale, ce genre de conflit peut être problématique.

(1) Arrange:

- Construire un BaseGraph graph sans réelles données géographiques (Mock)

Act:

- Appeler la méthode avec comme paramètres le BaseGraph graph et le id de 2 noeuds identiques (pour provoquer l'erreur attendu)

Assert:

- Vérifier que le message d'erreur est retourné 'IllegalArgumentException e'

(2)

Arrange:

- Construire un BaseGraph graph sans réelles données géographiques (Mock).

Act:

- Appeler la méthode avec comme paramètres le BaseGraph graph et le id de 2 nœuds différent (pour provoquer le comportement juste).

Assert:

- Vérifier que la nouvelle arête est bien retournée.