David Shorten
#20233475

IFT3913 - Tache #3
Tests dans le module ‘core’ de Graphhopper

2025-11-21

Workflow

Tache #1: ecrire la sortie du resultat pitest avec | tee pitest_out.txt

```
      - name: Mutation Testing
        run: mvn -B -pl core
org.pitest:pitest-maven:mutationCoverage | tee pitest_out.txt
```

Tache #2: Chercher le score de mutation en utilisant la regex approprié, puis le sauvegarder dans $GITHUB_OUTPUT avec id=pit

```
      - name: Extract mutation score
        id: pit
        run: |
          SCORE=$(grep -oP 'Killed\s+\d+\s+\(\K\d+(?=%\))' <<<
"$(cat pitest_out.txt)")
          echo "score=$SCORE" >> $GITHUB_OUTPUT
```

Tache #3: Comparer le score à celle du dernier run, qu'on garde dans un fichier mutation_score.txt. Échouer au build s'il est plus petit.

```
      - name: Compare mutation scores
        id: compare
        run: |
          BASELINE=$(cat mutation_score.txt)
          NEW=${{ steps.pit.outputs.score }}

          echo "baseline=$BASELINE" >> $GITHUB_OUTPUT

          if [ "$NEW" -lt "$BASELINE" ]; then
            echo "Mutation score decreased! Failing build."
            exit 1
          fi
```

Tache #4: Mettre à jour le contenu de mutation_score.txt avec le score de ce run s'il est plus grand.

```
      - name: Write new mutation score if higher
        if: ${{ steps.pit.outputs.score >
steps.compare.outputs.baseline }}
```

```
        run: |
          git config user.name "github-actions"
          git config user.email "actions@github.com"
          echo "${{ steps.pit.outputs.score }}" >
mutation_score.txt
          git add mutation_score.txt
          git commit -m "Update mutation score"
          git push
```

Tests

Classe: QueryOverlayBuilder

Test:

`noVirtualNodeIfConsideredEqual()`

Objectif:
Assurer qu'un node virtuel n'est pas généré au cas ou le query point est extrêmement proche d'un pillar node. Dans le code suivant,

```
// no new virtual nodes if very close ("snap" together)
if (Snap.considerEqual(prevPoint.lat, prevPoint.lon,
currSnapped.lat, currSnapped.lon)) {
    res.setClosestNode(prevNodeId);
    res.setSnappedPoint(prevPoint);
    res.setWayIndex(i == 0 ? 0 : results.get(i -
1).getWayIndex());
    res.setSnappedPosition(i == 0 ? Snap.Position.TOWER :
results.get(i - 1).getSnappedPosition());
    res.setQueryDistance(DIST_PLANE.calcDist(prevPoint.lat,
prevPoint.lon, res.getQueryPoint().lat, res.getQueryPoint().lon));
    continue;
}
```

on voit une exemple d'un cas exceptionnel pour le comportement "snapping" qui évite la création redondante d'une node virtuel s'il est trop proche d'une autre.

Classes simulees:

On a besoin d'un mock de EdgeIteratorState pour renvoyer les coordonnées comme étant trop proche pour être distingués significativement. Parmi les 7 cas thenReturn(), seulement le suivant est important:

```
when(edge.fetchWayGeometry(FetchMode.ALL)).thenReturn(pl)
```

puisque c'est les points dans PointList pl définis au début du test qui vont être comparées. Les autres cas thenReturn() sont choisis au hasard pour que le test fonctionne.

Declarations:

```
assertEquals(firstVirtNode, s1.getClosestNode());
assertEquals(firstVirtNode, s2.getClosestNode());
```

Vu que les deux points vont joindre au meme point virtuel, on déclare que getClosestNode() pour s1 et s2 sont égaux.

```
assertEquals(1, qo.getVirtualNodes().size());
```

Naturellement, on déclare qu'il y aura seulement 1 node, puisque le deuxième était redondant.

```
assertEquals(4, qo.getNumVirtualEdges());
```

Et finalement, puisque le node a comme node adjacent lui-même, et que chaque node a un edge sortant et entrant, on déclare qu'il y aura 4 edges virtuels pour ce node.

Test:

```
testDistanceBasedWayIndexOrder()
```

Objectif:
Assurer que les valeurs de wayIndex des Snap sont mises en ordre en fonction de leur distance du node PILLAR.

Classes simulees:
Il suffit de mocker un EdgeIteratorState comme dans l'autre test, qui renvoie les coordonnées des points. On donne au Snap les mêmes wayIndex pour forcer le cas où il faut les mettre en ordre selon leur distance.

On déclare que l'ordre des points virtuels est l'ordre croissant de la distance des points qu'on a défini au début du test.

Classe: GHUtility.java

Test:

```
testCount()
```

Objectif:
Assurer que count() compte bien les voisins des classes EdgeIterator et RoutingCHEdgeIterator

Classes simulees:
On mock EdgeIterator et RoutingCHEdgeIterator, on simule plusieurs .next(), et on compte les itérations. On assertEquals() que le nombre d'itérations est le même.

Tests:

```
testGetNeighbors()
testGetEdgeIds()
```

Objectif:
Assurer qu'un séquence de .next() et .getAdjacentNode()/getEdges() renvoie les listes correctes de ids.

Classes simulees:
On mock EdgeIterator qui va renvoyer une séquence de id, on crée une liste avec les mêmes valeurs, et on déclare que les listes sont égales.

Test:

```
testGetDistance()
```

Objectif:
Assurer que la fonction getDistance() de GHUtility renvoie la bonne valeur de distance

Classes simulees:
On génère 2 coordonnées et on mocke la classe NodeAccess pour renvoyer ces coordonnées. On compare le résultat de la fonction avec le résultat de calcDist() avec nos coordonnées.

Tests:

```
testGetEdge_OneEdge()
testGetEdge_NullEdge()
testGetEdge_NoEdge()
testGetEdge_MultipleEdges()
```

Objectif:
Assurer que les cas divers de getEdge() renvoient les valeurs attendues. Les cas sont les suivant:

1. Il existe un edge -> on confirme qu'un objet de la classe EdgeIteratorState est renvoyé
2. Il existe plusieurs edge -> on confirme que IllegalArgumentException est lancé
3. Il n'existe pas de edge -> on confirme que null est renvoyé
4. Le edge n'est pas celle qu'on cherchait -> on confirme que IllegalStateException est lancé

Classes simulees:

Il faut simuler un Graph, puisque la fonction en prend comme argument. Il faut aussi simuler la classe EdgeExplorer, qui est renvoyé par la graphe. Finalement, il faut simuler EdgeIterator, qui est renvoyé par EdgeExplorer. Dans chacun des 4 tests, il est important que l'objet EdgeIterator renvoyé par le mock du EdgeExplorer est un nouvel instance a chaque appel, sinon quand la fonction getEdge appel la fonction pour la deuxième fois, il renvoie une EdgeIterator dans le mauvais état.

Dans le cas #1 (il existe un edge), on mock un EdgeIterator tel que le résultat de count() est 1, et que le id qu'il renvoie est 'adj', qu'on fournit aussi à la fonction getEdge(). Il suffit de déclarer que l'objet retourné est un EdgeIterator, puisqu'il y a seulement un cas où cela est vrai, et parce qu'il est très difficile et impratique de comparer deux mocks.

Dans le cas #2 (il existe plusieurs edge), on mock un EdgeIterator tel que le résultat de count() est plus qu'un. Cela lance l'exception IllegalArgumentException est on assertThrows() que cela est le cas.

Dans le cas #3 (il n'existe pas de edge), on mock un EdgeIterator tel que le résultat de count() est 0, et on déclare que la fonction renvoie la valeur nulle.

Dans le cas #4 (le edge n'est pas celle qu'on cherchait), il faut que, lors de l'appel a count(), le résultat est 1, mais que lors de la deuxième appel a EdgeExplorer.getBaseNode(), le EdgeIterator renvoie n'a plus le même node

adjacent. Pour le mock de EdgeIterator alors, on a deux .thenAnswer() différents, un qui renvoie un EdgeIterator avec getAdjNode() = adj et l'autre getAdjNode() = adj2. Le EdgeIterator n'est jamais renvoyé, et alors on déclare que l'exception IllegalStateException est lancée.