Τεχνητή Νοημοσύνη

Γενικό Θέμα

Μητρόπουλος Κωνσταντίνος - 03113169
Μπαξόπουλος Γεώργιος - 03113021

Εισαγωγή

Στο πλαίσιο αυτής της άσκησης, φτιάχνουμε ένα πρόγραμμα το οποίο θα βρίσκει απο μία λίστα ταξί, αυτό που βρίσκεται πιο κοντά στον πελάτη. Ο αλγόριθμος που θα χρησιμοποιηθεί για την εύρεση της διαδρομής είναι ο A*.

Δεδομένα

- Ο χάρτης δίνεται σε ένα αρχείο **nodes.csv** όπως φαίνεται από κάτω:

  ```
  X,Y,id,name
  23.7140723, 37.950289, 23181765,
  23.714813, 37.9497098, 23181765,
  23.7188913, 37.9491957, 23181771, Εφέσσου
  23.7184186, 37.949578, 23181771, Εφέσσου
  …
  ```

  Κάθε ζευγάρι συντεταγμένων x-y αποτελεί ένα node ενός δρόμου (ή περισσότερων αν είναι σταυροδρόμι) με το ζευγάρι id-name ως ταυτότητα του δρόμου στον οποίο ανήκει. Καταλαβαίνουμε οτι οι κόμβοι που αποτελούν σταυροδρόμια θα εμφανίζονται πολλές φορές στο αρχείο με διαφορετικά ζευγάρια id-name. Στην άσκηση μας ο χάρτης που χρησιμοποιείται αποτελεί

- Τα ταξί δίνονται σε ένα αρχείο **taxis.csv** όπως φαίνεται από κάτω:

  ```
  X,Y,id
  23.733553,38.000959,100
  23.692615,37.945689,110
  23.757838,37.962517,120
  …
  ```

  Κάθε ταξί προφανώς θα έχει ξεχωριστό id, το οποίο ομως στο πρόγραμμα μας δεν το χρησιμοποιούμε, αλλα τα διακρίνουμε με βάση τη σειρά που μας έρχονται κατα το input. Δηλαδή στο παραπάνω παράδειγμα το ταξί με id = 100 θα είναι το Taxi 1, το ταξί με id = 110 θα είναι το Taxi 2, κτλ.
  ΠΡΟΣΟΧΗ: Οι συντεταγμένες των ταξί μπορεί να μην βρίσκονται ακριβώς πάνω σε κάποιο node του χάρτη. Για αυτό είναι απαραίτητο να υπάρξει μια συνάρτηση **findClosestToTaxi()** η οποία θα βρίσκει το κόντινοτερο node στο ταξί που να ανήκει στο χάρτη.

- Ο πελάτης δίνεται σε ένα αρχείο **client.csv** όπως φαίνεται από κάτω:

```
X,Y
23.74674,37.97852
```

ΠΡΟΣΟΧΗ: Όπως και στα ταξί, έτσι και με τον πελάτη απαιτείται μια συνάρτηση **findClosestToClient()** η οποία θα βρίσκει το κόντινοτερο node στο ταξί που να ανήκει στο χάρτη (στην ουσία το δρόμο που θα πρέπει να περπατήσει ο πελάτης για να μπει στο ταξι).

Ζητούμενα

Το πρόγραμμα μας καλείται να βρεί ποιο ταξί βρίσκεται πιο κοντά στον πελάτη, δηλαδή ποιο θα πρέπει να να διανύσει την μικρότερη δυνατή διαδρομή για να τον φτάσει. Συγκεκριμένα, εκτυπώνει κατάλληλα το περιεχόμενο ενός KML αρχείου στο stdout, έτοιμο για visualisation στο *https://www.google.com/maps/d/*, με χρώμα κόκκινο για κάθε ταξί και απο κάτω μια σειρά σε σχόλιο που να γράφει ποιο ταξί στο KML θα πρέπει να αλλάξουμε manually το χρώμα σε πράσινο στο αρχείο KMLI (το ταξί που επιλέγει δηλαδή το πρόγραμμα μας). Ένα σύντομο απλοποιημένο παράδειγμα output του προγράμματος μας είναι το απο κάτω:

```
<?xml version="1.0" encoding="UTF-8"?>
	<kml xmlns="http://earth.google.com/kml/2.1">
	<Document>
		<name>Taxi Routes</name>
		<Style id="green">
			<LineStyle>
				<color>ff009900</color>
				<width>4</width>
			</LineStyle>
		</Style>
		<Style id="red">
			<LineStyle>
				<color>ff0000ff</color>
				<width>4</width>
			</LineStyle>
		</Style>
		<Placemark>
			<name>Taxi 1</name>
			<styleUrl>#red</styleUrl>
			<LineString>
				<altitudeMode>relative</altitudeMode>
				<coordinates>
```

```
                                        23.7462886,37.9783345,299645496
                                        ...
                                        23.7464026,37.9780191,23184108
                                    </coordinates>
                                </LineString>
                            </Placemark>
                            <Placemark>
                                <name>Taxi 2</name>
                                <styleUrl>#red</styleUrl>
                                <LineString>
                                    <altitudeMode>relative</altitudeMode>
                                    <coordinates>
                                        23.746432,37.197345,299645431
                                        ...
                                        23.7464026,37.9780191,23184108
                                    </coordinates>
                                </LineString>
                            </Placemark>
                        </Document>
</kml>
<!--Manually change Taxi 2 with distance 0.022724206193895168 to green colour at the kml file→
```

Επομένως για να δούμε τωρα τον χάρτη με τις διαδρομές, θα γράψουμε το output σε ένα kml αρχείο, θα αλλάξουμε τη γραμμή του Taxi 2 (λόγω της τελευταίας σειράς του output) από <styleUrl>#red</styleUrl> σε <styleUrl>#green</styleUrl>, θα πάμε στο *https://www.google.com/maps/d/* και θα κάνουμε import το αρχείο kml για το visualisation.

Υποθέσεις

Στο πλαίσιο της άσκησης έχουμε κάνει μερικές υποθέσεις:

- Αγνοούμε εντελώς σύνθετες παραμέτρους όπως φανάρια, απαγορευτικά, κατευθύνσεις δρόμων, χωματόδρομους, δρόμοι για πεζους κτλ.
- Όλα τα σημεία τομής αποτελούν διασταυρώσεις
- Γεωγραφικές γραμμές που δεν αντιστοιχούν σε οδούς, όπως πχ μονοπάτια, υδάτινες οδοί, όρια περιοχών, κτλ θεωρούνται δρόμοι στο πλαίσιο της άσκησης. Αυτό οφείλεται στα nodes που μας δίνει ως έξοδο το QGIS

Μοντελοποίηση του προβλήματος

- Τις συντεταγμένες του client τις αποθηκεύουμε σε ένα πίνακα double με δύο στοιχεία.
- Τα ταξί τα αποθηκεύουμε με τη σειρά σε ένα ArrayList<Node>, όπου Node είναι μια κλάση που δημιουργήσαμε εμείς και περιέχει τις εξής πληροφορίες: {double x, double y, ind id, int arraylocation, boolean isMain, int indexMain, ArrayList<Integer> neighbours, private double gScore, private double fScore, Node cameFrom}. Τα x, y, id είναι προφανώς οι συντεταγμένες και τα υπόλοιπα είναι στην ουσία άχρηστα για τα ταξί και δεν μας ενδιαφέρουν. *Σημείωση*: Κόντρα στους άγραφους κανόνες της Java , κάναμε τις συντεταγμένες public, με τη λογική ότι επειδή χρειάζεται να τις επεξεργαστούμε και να τις ελέγξουμε πάρα πολλές φορές κατά το τρέξιμο τόσο του aStar όσο και της createHashMap, θέλουμε να αποφύγουμε πολλές κλήσεις getter συναρτήσεων.
- Τις συντεταγμένες του χάρτη, τις αποθηκεύουμε σε ένα ArrayList<Node> και ύστερα σε ένα HashMap<String, ArrayList<Node>> με το ίδιο Node που περιγράψαμε πιο πριν, απλά τώρα χρησιμοποιούμε όλες τις πληροφορίες του. Το HashMap χρησιμοποιήθηκε απλά για την πιο γρήγορη εύρεση και αποθήκευση των γειτόνων κάθε κόμβου στον ArrayList (από O(n^2) σε O(n•k) όπου το k προκύπτει από την υλοποίηση του HashMap από τη Java, αλλά είναι σίγουρα *πολύ* μικρότερο από n, επειδή χρησιμοποιήσαμε τον default load factor 0.75 για το Map, άρα θυσιάζουμε μνήμη για ταχύτητα). Για key χρησιμοποιήσαμε το “X,Y”. Σχετικα με τα ορίσματα:
    - Το arraylocation αποτελεί τη θέση του Node στον ArrayList (για όταν βρισκόμαστε στο Map να έχουμε τη θέση του).
    - Το isMain είναι true όταν έχουμε Node που εμφανίζεται πολλές φορές, αλλά το παρόν node είναι το πρώτο που εμφανίζεται, ενώ τα υπόλοιπα με τις ίδιες συντεταγμένες x, y, έχουν isMain = false.
    - Το neighbours είναι μια λίστα με τα indices όλων των γειτόνων του κόμβου στο ArrayList. Το neighbours περιέχει indices μόνο όταν το isMain είναι true, αφού στον A* μόνο αυτός ο κόμβος θα μας απασχολήσει (βλ. Ανάλυση της aStar() στην παρούσα αναφορά).
    - Το gScore είναι η απόσταση που έχει διανυθεί για να φτάσουμε μέχρι τον κόμβο (κατα το τρέξιμο του aStar())
    - Το fScore είναι το άθροισμα της απόστασης που έχει διανυθεί μέχρι τον τρέχοντα κόμβο και της ευριστικής συνάρτησης του κόμβου αυτού, δηλαδή η ευκλείδεια απόστασή του από τον κόμβο-στόχο.
    - Το cameFrom είναι reference στο Node από το οποίο φτάσαμε στο συγκεκριμένο κόμβο. Χρησιμέυει για να αποθηκεύουμε το βέλτιστο path που ακολουθήσαμε για να φτάσουμε στο στόχο.

aStar

Ο αλγόριθμος που χρησιμοποίησαμε για την εύρεση της βέλτιστης διαδρομής είναι ο Astar. Συγκεκριμένα, τροποποιήσαμε για τις προαναφερθείσες δομές δεδομένων τον ψευδοκώδικα από τη σελίδα https://en.wikipedia.org/wiki/A*_search_algorithm

Η συνάρτηση aStar() παίρνει ως παραμέτρους την ArrayList<Nodes> του χάρτη, τον αρχικό κόμβο start, και τον κόμβο προορισμού goal. Η λογική είναι η εξής:

- Ο αλγόριθμος πάντα έχει ένα Node current, το οποίο εξετάζει. Προσέχουμε να πάρουμε ως current *πάντα* ένα node που έχει isMain true, ώστε να μπορούμε να βρούμε τους γείτονες του άμεσα. Αν isMain == false τότε πηδάμε στο αντίστοιχο indexMain του πίνακα, όπου βρίσκεται το Main Node (αντιπρόσωπος κόμβος).
- Αφαιρούμε το current από το openSet, και το προσθέτουμε στο closedSet (λίστα από ήδη εξετασμένα nodes), ώστε να μην κάνει κύκλους ο αλγόριθμος.
- Στη συνέχεια, διατρέχει όλα τα παιδιά του current σε ένα for. Για κάθε παιδί, έχουμε έναν κόμβο neighbour, για τον οποίο πάλι προσέχουμε να έχουμε isMain = true, επειδή στη συνέχεια θα *εξετάσουμε* αυτόν τον κόμβο, δηλαδή θα γίνει current.
- Κάθε neighbour το προσθέτουμε στο ArrayList<Node> openSet, εφόσον *δεν* υπάρχει ήδη, ούτε στο openSet ούτε στο closedSet. Το openSet αποτελεί τη μοντελοποίηση του μετώπου αναζήτησής μας. Θέλουμε να είναι πάντα ταξινομημένο σε *αύξουσα* σειρά ως προς το fScore των κόμβων, ώστε σε O(1) χρόνο να παίρνουμε για εξέταση τον *φαινομενικά* πιο συμφέροντα κόμβο. Γι'αυτό κάνουμε το insertion στη σωστή θέση ώστε να διατηρηθεί η αύξουσα σειρά.
- Αν δεν υπάρχει ο neighbour σε καμία από τις δύο παραπάνω λίστες, τότε το fScore του γίνεται ίσο με tentative_gScore (δηλαδή η απόσταση που διανύεται μέχρι αυτόν) + την ευριστική από το στόχο, δηλαδή την ευκλείδεια απόσταση από αυτόν. Σημείωση: οι default τιμές των fScore και gScore τίθενται ίσες με Double.MAXVALUE κατά τη δημιουργία του node.
- Αν υπάρχει στο openSet, ελέγχουμε (αφού το έχουμε προσπελάσει ήδη) εάν μας συμφέρει να το ξαναπροσπελάσουμε, δηλαδή ελέγχουμε αν το μονοπάτι από το current στο neighbour είναι καλύτερο από αυτό που βρέθηκε κάποια στιγμή πριν και οδηγεί στο neighbour. Αυτό γίνεται με τον έλεγχο `if (tentative_gScore >= neighbour.getgScore()) continue;`
- Αν μας συμφέρει να πάρουμε το neighbour στο βέλτιστο μονοπάτι, τότε αποθηκεύουμε στην cameFrom το current (ότι δηλαδή προήλθε από το current) και θέτουμε το gScore ίσο με την απόσταση μέχρι το σημείο αυτό, και το fScore όπως είπαμε πριν.
- Στη συνέχεια παίρνουμε ως current το πρώτο στοιχείο της λίστας openSet, το οποίο έχουμε εξασφαλίσει ότι είναι αυτό με το μικρότερο fScore. Ο αλγόριθμος τότε επαναλαμβάνεται από το πρώτο βήμα.

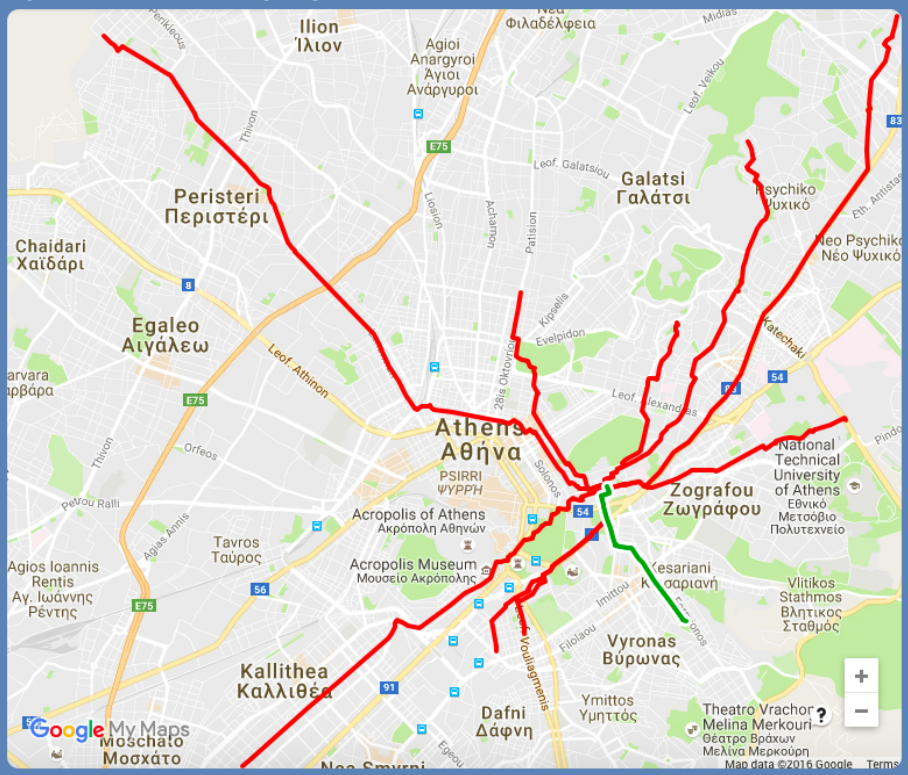
- Ο αλγόριθμος σταματά με *επιτυχία* αν στην πρώτη θέση του openSet βρει τον κόμβο-στόχο και πάει να τον αναπτύξει, ή με *αποτυχία* εάν αδειάσει το openSet, δηλαδή δεν υπάρχουν άλλοι κόμβοι προς εξέταση.

Παρατηρήσεις για την εργασία

Ο πηγαίος κώδικας Java συμπεριλαμβάνεται στο αρχείο που υποβάλαμε. Αποτελείται από τα *.java αρχεία με τις κλάσεις που χρησιμοποιήσαμε. Τα compiled *.class (εκτελέσιμα) αρχεία βρίσκονται στο out/Production/Taxis directory. Τα *.csv αρχεία εισόδου βρίσκονται τόσο στο root directory της εργασίας, όσο και στο out/Production/Taxis, ανάλογα με το πώς επιθυμεί ο χρήστης να τρέξει το πρόγραμμα, δηλαδή είτε από κάποιο IDE (αναπτύχθηκε σε Jetbrains intelliJ) είτε σε κάποιο terminal. Στην τελευταία περίπτωση αρκεί να δοθεί εντολή `java taxis`, αφού στο taxis.java -> taxis.class βρίσκεται η main().

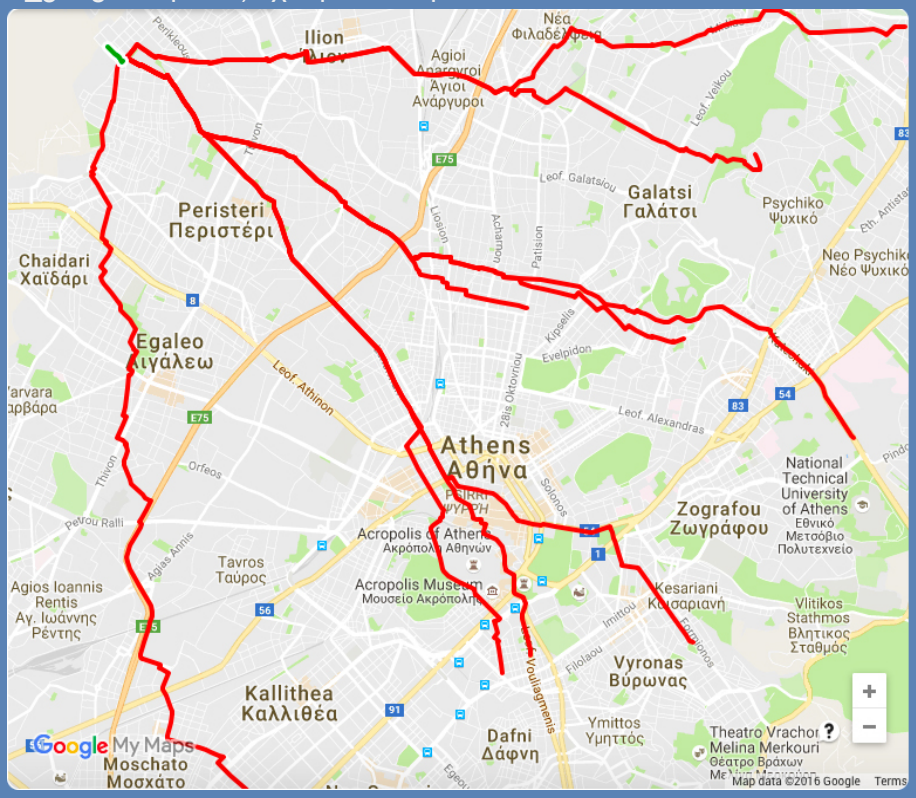
Visualisation

Για την αναπαράσταση του προβλήματος κάναμε import στο *https://www.google.com/maps/d/* τα KML που παράξαμε τρέχοντας το πρόγραμμα και αφού αλλάξαμε το χρώμα στο κατάλληλο ταξί. Για το taxi.csv, client.csv και nodes.csv που μας δίνονται (αρχείο googlemap.kml) έχουμε το παρακάτω **visualisation**:



Client's location: (23.74674,37.97852)

Για το taxi.csv, custom_client.csv και nodes.csv που μας δίνονται (αρχείο custom_googlemap.kml) έχουμε το παρακάτω **visualisation**:



Client's location: (23.6743782,38.0292029)