Τεχνητή Νοημοσύνη

1η Προγραμματιστική Άσκηση

Δεκέμβριος 2017

Γαβαλάς Νίκος 03113121

Μουσελίνος Σπυρίδων 03113072

1. Περιγραφή του συστήματος

1.1. Φόρτωση των δεδομένων εισόδου

Στην υλοποίηση μας αρχικά διαβάζουμε καθένα από τα αρχεία taxis.csv, client.csv και nodes.csv (τα οποία πρέπει να είναι στο ίδιο directory με το .jar) και δημιουργούμε τα αντίστοιχα objects και κατασκευάζουμε τον γράφο, στον οποίο θα τρέξουμε τον αλγόριθμο Astar.

Αυτή η διαδικασία αρχικοποίησης διαρκεί περίπου 20 δευτερόλεπτα στο runtime για το μέγεθος των δοθέντων αρχείων.

Για την σωστή επεξεργασία των δεδομένων αρχικά μετατρέψαμε το αρχείο σε κωδικοποίηση UTF-8 από ISO-8859-1 και στη συνέχεια για κάθε γραμμή του αρχείου τη σπάμε σε λίστα με delimiter το κόμμα “,” και παίρνουμε κάθε token που προκύπτει μετά από τις αντίστοιχες μετατροπές τύπων σε Integers, Strings ή Doubles όπου χρειάζεται (X(double), Y(double) ID(integer), Name(String)).

Συγκεκριμένα, για το client.csv, δημιουργούμε ένα instance της κλάσης Position, η οποία περιέχει τις συντεταγμένες X,Y του πελάτη (longitude latitude για την ακρίβεια).

Για το taxis.csv, για κάθε ταξί δημιουργούμε ένα instance της κλάσης Taxi (που κάνει inherit την Position) και αποθηκεύουμε εκεί τις αντίστοιχες πληροφορίες (X, Y, id) για το κάθε ταξί.

1.2. Κατάσκευη του γράφου

Ύστερα για κάθε γραμμή του nodes.csv με αντίστοιχο τρόπο δημιουργούμε ένα instance της κλάσης Vertex (που επίσης κάνει inherit την Position), και προσθέτουμε τον Vertex στην δομή ArrayList της κλάσης Graph, αλλά και σε μία δομή HashSet, επίσης της κλάσης Graph.

Ο λόγος που χρησιμοποιούμε το HashSet, είναι επειδή διαβάζοντας συντεταγμένες, θέλουμε να ξέρουμε άμεσα αν έχουμε ήδη κόμβο στον γράφο με τις ίδιες συντεταγμένες, ώστε σε αυτή τη περίπτωση αντί να φτιάξουμε έναν ίδιο, απλά να πάρουμε ένα reference στον ήδη υπάρχοντα. Η δομή HashSet μας επιτρέπει να το μάθουμε αυτό σε χρόνο O(1).

Επίσης κατά την ανάγνωση του αρχείου και της κατασκευής του γράφου, ελέγχουμε αν το Id του κόμβου που διαβάσαμε είναι ίδιο με του προηγουμένου. Σε τέτοια περίπτωση, καταλαβαίνουμε ότι οι κόμβοι είναι στον ίδιο δρόμο, οπότε βάζουμε τον έναν γείτονα του άλλου (προσθέτοντας τον καθένα σε μία δομή ArrayList του άλλου - η adjacency list (λίστα γειτνίασης)).

1.3. Ο Αλγόριθμος A* και η δομή Priority Queue

Στη συνέχεια τρέχουμε τον αλγόριθμο του Astar στον γράφο που κατασκευάσαμε για κάθε ένα από τα ταξί προς τα εμάς.

Στον Astar έχουμε μεταβλητό μέγεθος μετώπου αναζήτησης το οποίο δέχεται ως όρισμα ο constructor της κλάσης Graph για να αρχικοποιήσει τη δομή που κατασκευάσαμε για το μέτωπο του Astar. Στο ίδιο το πρόγραμμα, η παράμετρος αυτή *τίθεται ως όρισμα στο ίδιο το εκτελέσιμο* (το πρώτο commandline argument), και αν μείνει κενό έχει default τιμή 150000.

Αυτή η δομή είναι στην ουσία μία PriorityQueue που έχει όμως τις εξής ιδιότητες: πρόσβαση στο πρώτο (το μικρότερο) αλλά και το τελευταίο στοιχείο σε O(1), εισαγωγή σε O(logn), ενώ όταν είναι γεμάτη και πάει να εισαχθεί στοιχείο, τότε το στοιχείο αυτό συγκρίνεται με το υπάρχον τελευταίο (το χειρότερο της λίστας) και αν είναι μικρότερο από αυτό, τότε αφαιρείται

το τελευταίο και εισάγεται το νέο, ενώ αν δεν είναι, απλά αγνοείται η εισαγωγή του καινούργιου στοιχείου και η λίστα μένει ως έχει.

Συνοπτικά δηλαδή για την μέθοδο αντικατάστασης κάνουμε το εξής:

```
Αν η λίστα είναι γεμάτη:
   Συγκρινε το στοιχείο προς εισαγωγή με το τελευταίο στοιχείο
   Αν ειναι μεγαλυτερο τότε
      μην κανεις τιποτα
   Αλλιως
      διάγραψε το τελευταιο στοιχείο της λίστας-μετώπου και κάνε εισαγωγή το νέο
στοιχείο.
```

Η μέθοδος findPath() της κλάσης Graph είναι αυτή που υλοποιεί τον αλγόριθμο Astar. Η findPath() λειτουργεί υπολογίζοντας ως βάρη ακμών την απόσταση του τρέχοντος κόμβου από καθέναν από τους γείτονές του, με την μέθοδο της κλάσης Position, distanceFrom(), η οποία υπολογίζει σε μέτρα το μήκος τόξου μίας Position(longitude x, latitude y) από μία άλλη. Επίσης χρησιμοποιεί την εν λόγω συνάρτηση για τον υπολογισμό της ευκλείδειας απόστασης του τρέχοντος κόμβου από τον κόμβο-στόχο, την τιμή της οποίας χρησιμοποιούμε ως ευριστική.

Η ευριστική (H) μαζί με το τρέχον κόστος (G) χρησιμοποιούνται για να ενημερώσουμε τα τελικά κόστη των κόμβων που καθορίζουν τη διάταξη τους στην PriorityQueue.

1.4. Δημιουργία Αρχείου Εξόδου

Η findPath() αφού βρει την ελάχιστη διαδρομή την ανακατασκευάζει και επιστρέφει ένα στιγμιότυπο της κλάσης Path, που κρατάει σε ArrayList όλους τους κόμβους με τη σειρά και έχει μεθόδους μετατροπής αυτών σε String για να τα γράψουμε ύστερα στο KML αρχείο.

Επίσης να αναφέρουμε ότι η findPath() δεν εκτελείται κατ' ευθείαν για τις συντεταγμένες που τους περάστηκαν σαν ορίσματα, γιατί πιθανότατα δεν υπάρχουν ήδη στον γράφο, όποτε βρίσκει πρώτα με την μέθοδο findClosestVertex() τους πλησιέστερους κόμβους-αντιπρόσωπους που ανήκουν όντως στον γράφο σε χρόνο O(n) και χρησιμοποιεί αυτούς.

Αφού ολοκληρωθεί η εκτέλεση της findPath() λοιπόν για κάθενα από τα ταξί και την τοποθεσία του πελάτη, δημιουργούμε το KML αρχείο με τα αποτελέσματα, χρησιμοποιώντας τη βοηθητική κλάση KMLGenerator.

2. Αποτελέσματα

2.1. Εφαρμογή για τα δοθέντα αρχεία

Μέγεθος Μετώπου	Μέγιστος αριθμός βημάτων	Πραγματικό μέγιστο μέγεθος μετώπου	Μήκος Διαδρομής
50.000	33216	1238	3,64km (της βέλτιστης)
10.000	33216	1238	3,64km (βέλτιστης)
5.000	33216	1238	3,64km (βέλτιστης)
1.000	32425	1000	3,64km (βέλτιστης)
500	30184	500	3,64km (με μικρά σφάλματα στις υπόλοιπες)
200	21650	200	3,64km (με μικρά σφάλματα στις υπόλοιπες)
100	15078	100	3,74km (όχι βέλτιστο)
50	8495	50	3,82km (όχι βέλτιστο)
10	4379	10	3,64km (βέλτιστη / ωστόσο δεν βρίσκει μερικές καθόλου)
6	1853	6	4,87km (όχι βέλτιστο / και δεν βρίσκει μερικές καθόλου)

Οι άνωθι τιμές προκύπτουν από αυτά που τυπώνονται στο Stdout του προγράμματος μας κατά την εκτέλεση του.

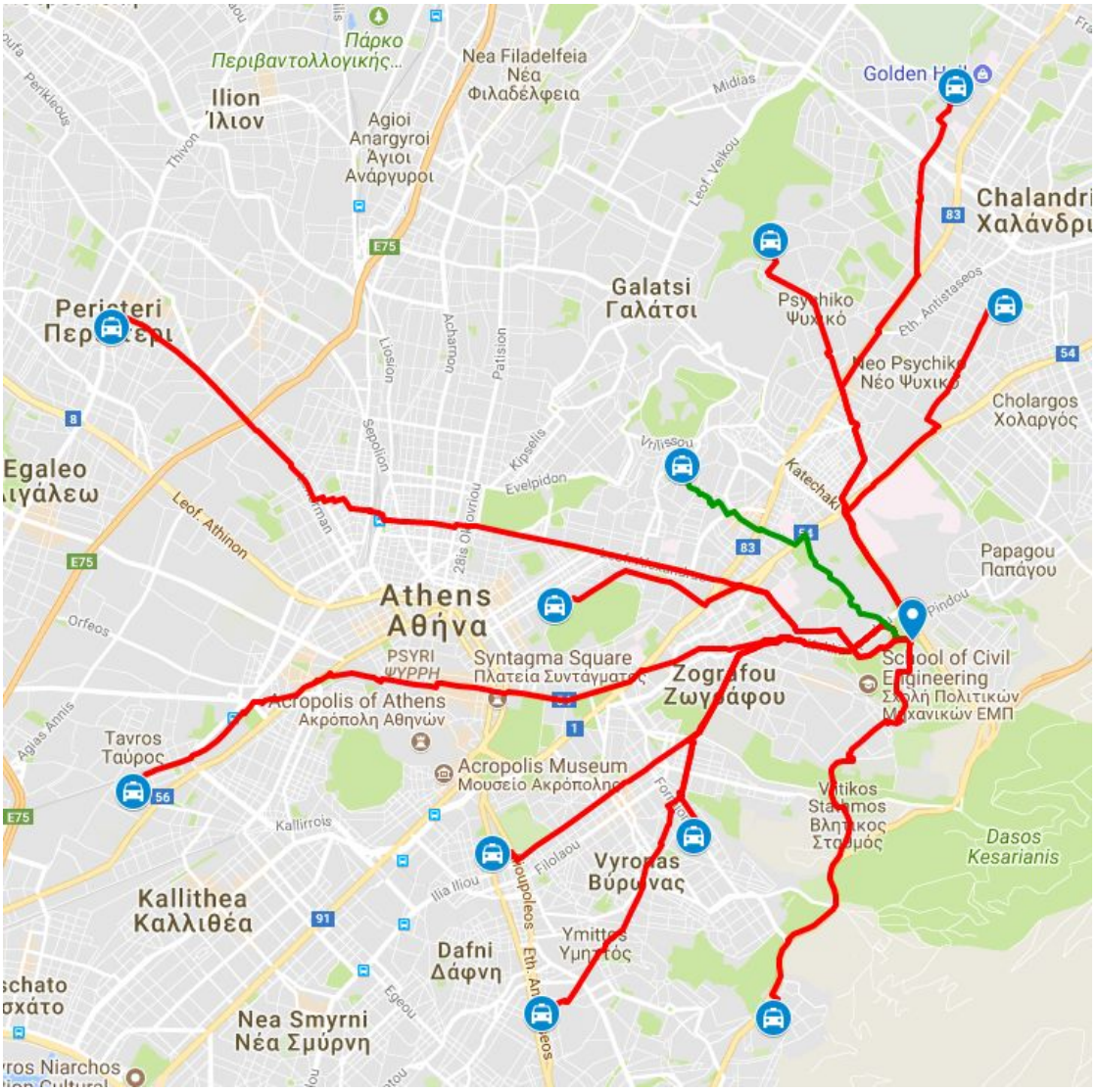
Από εκεί και κάτω δεν βγάζει αποτελέσματα ο Astar για τα συγκεκριμένα inputs.

Link για τον χάρτη που προέκυψε:

https://drive.google.com/open?id=1yu4H0GQiNPGiJIrwvyVyQZIlzEKY4P7A&usp=sharing

Από αριστερά επιλέγοντας τα αντίστοιχα checkboxes φαίνονται οι διαδρομές για τα αντίστοιχα μεγέθη του μετώπου. Με κλικ στην αντίστοιχη διαδρομή εμφανίζεται το μήκος της.

Ενδεικτικό screenshot και το αντίστοιχο KML αρχείο που παράγεται για μέγεθος 50000:



Το KML έχει την κάτωθι μορφή, με όσα placemark tags χρειάζονται (ένα για κάθε διαδρομή - απλώς εδώ εμφανίζουμε το ένα μόνο για λόγους συντομίας).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<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>100</name>
	<styleUrl>#red</styleUrl>
	<LineString>
		<altitudeMode/>
		<coordinates>
			23.741461,37.984285,0
			23.741852,37.983853,0
			.
			.
			.
			23.783841,37.980940,0
			23.783848,37.980871,0
		</coordinates>
	</LineString>
	</Placemark>
	</Document>
</kml>
```

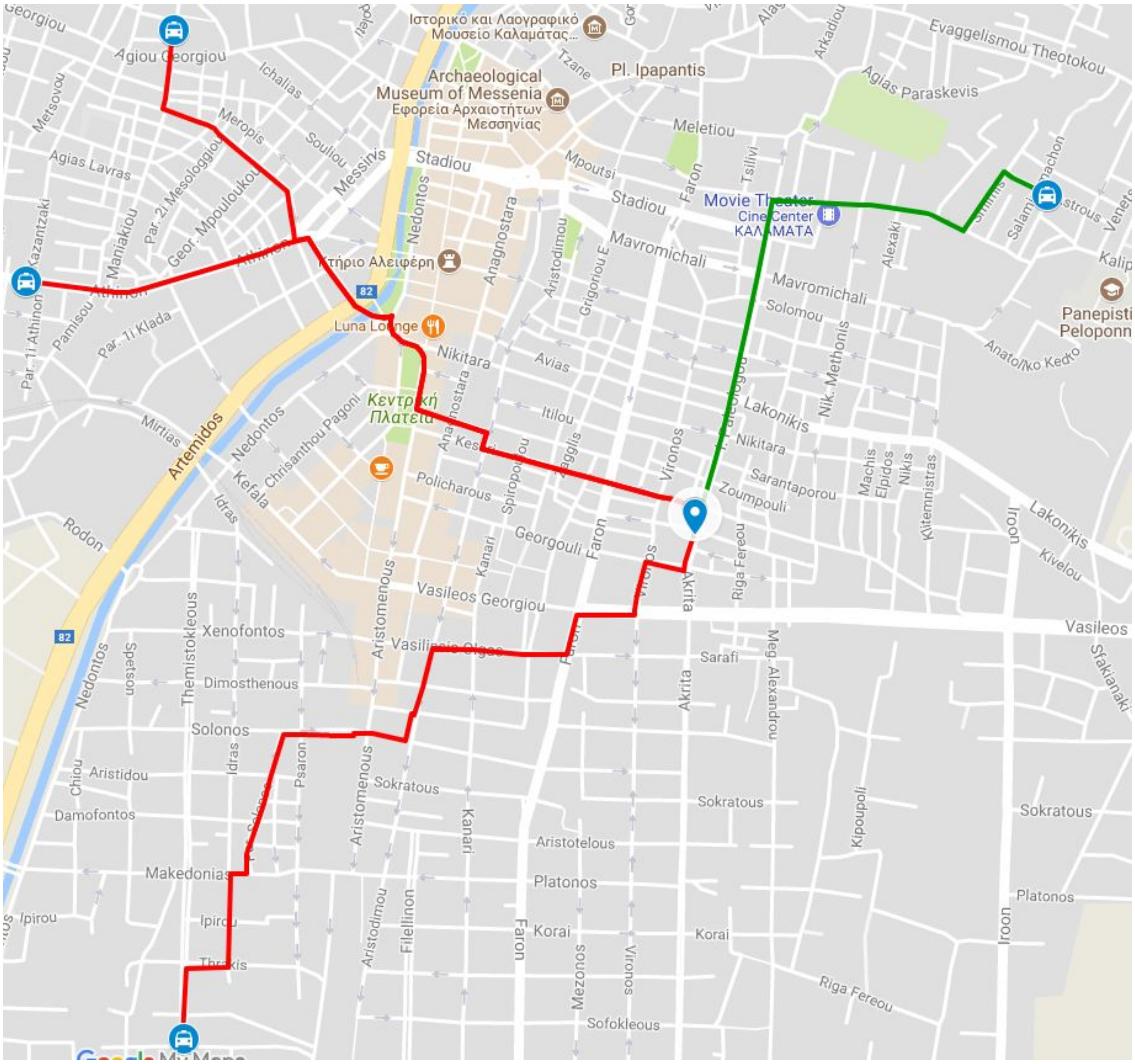
Όλα τα KML παραδίδονται μέσα στον συμπιεσμένο φάκελο.

2.2. 2η εφαρμογή για τον χάρτη της Καλαμάτας

Αφού με τον τρόπο που περιγράφεται στην εκφώνηση φτιάξαμε τα κατάλληλα csv αρχεία για τον χάρτη της Καλαμάτας, και τοποθετήσαμε μερικά ταξί και έναν πελάτη σε τυχαίες τοποθεσίες, πήραμε τα αντίστοιχα αποτελέσματα:

https://drive.google.com/open?id=1rsI1m7pC46qcqU0WCPVBdO7oZL1Umozr&usp=sharing

Ενδεικτικό screenshot για μέγεθος μετώπου 10000:



Ο πίνακας με τα ζητούμενα στοιχεία για τη βέλτιστη διαδρομή φαίνεται παρακάτω:

Μέγεθος Μετώπου	Μέγιστος αριθμός βημάτων	Πραγματικό μέγιστο μέγεθος μετώπου	Μήκος Επιλεγόμενης Διαδρομής
10.000	1128	182	1,02km (βέλτιστη)
1.000	1128	182	1,02km (βέλτιστη)
100	1074	100	1,04km (όχι βέλτιστο)
10	458	10	1,03km (όχι βέλτιστο)
5	189	5	1,06km (όχι βέλτιστο)
1	63	1	1,40km (όχι βέλτιστο / ωστόσο δεν βρίσκει μερικές καθόλου)

2.3. Σχολιασμός

Αρχικά κάθε εκτέλεση για διαφορετικό μέγεθος μετώπου αναζήτησης, είναι σίγουρο ότι θα βρει βέλτιστη διαδρομή όσο το μέτωπο αναζήτησης είναι μεγαλύτερο από το μέγιστο απαιτούμενο της κάθε διαδρομής.

Από κει και πέρα, μειώνοντας περαιτέρω το μέγεθος μετώπου αρχίζουν και χαλάνε πρώτα οι δευτερεύουσες διαδρομές και τελικά και η πράσινη (που αντιστοιχεί στον κοντινότερο στόχο), ενώ αν μειώσουμε το μέγεθος πάρα πολύ τότε μερικές διαδρομές δεν μπορούν καν να βρουν τον στόχο.

Αυτό συμβαίνει γιατί για μικρό μέγεθος ουράς, δεν κρατούνται μέσα αρκετοί κόμβοι (στους οποίους θα περιλαμβάνονταν και αυτοί που θα ανήκαν στη βέλτιστη διαδρομή), γιατί τους παίρνουν τη θέση εκείνοι που αρχικά έχουν μικρή ευριστική αλλά τελικά δεν καταλήγουν κάπου (επειδή για παράδειγμα η διαδρομή έχει ένα μεγάλο εμπόδιο στη μέση).

Πάντως η κοντινότερη διαδρομή βρίσκεται ακόμα και για σχετικά μικρές τιμές μεγέθους ουράς, πράγμα που σημαίνει ότι εφαρμόζοντας κατάλληλα μεγέθη σε αυτή τότε σώζουμε σε μνήμη και χρόνο εκτέλεσης του αλγορίθμου, χωρίς μεγάλα tradeoffs (δηλαδή η απόδοση δεν μειώνεται πολύ - ιδιαίτερα για την διαδρομή του κοντινότερου, που είναι αυτή που μας ενδιαφέρει κιόλας περισσότερο).