Εθνικο Μετσοβιο Πολυτεχνειο

Τεχνητή Νοημοσύνη

Αναφορά 1ης Προγραμματιστικής Άσκησης

Γεωργίου Δημήτριος (03115106)
<el15106@central.ntua.gr>

Σαρμάς Ελισσαίος (03115111)
<el15111@central.ntua.gr>

Δεκέμβριος 2018

Γενικός Σχεδιασμός

Περιγραφή Προβλήματος

Στην παρούσα εργασία καλούμαστε να υλοποιήσουμε μια ευφυή υπηρεσία εξυπηρέτησης πελατών ταξί. Πιο συγκεκριμένα, θεωρούμε ότι υπάρχει ένας πελάτης που βρίσκεται σε μια ορισμένη τοποθεσία, ο οποίος επιθυμεί να καλέσει ένα ταξί δίνοντας στην υπηρεσία της συντεταγμένες του μέσω GPS κινητού τηλεφώνου. Η υπηρεσία διαθέτει μια βάση δεδομένων με έντεκα διαθέσιμα ταξί και τη γεωγραφική θέση στην οποία βρίσκονται (θεωρητικά η θέση των ταξί θα πρέπει να ανανεώνεται συνεχώς κάθε χρονική στιγμή). Η ευφυής υπηρεσία θα πρέπει να βρίσκει και να ειδοποιεί το ταξί που βρίσκεται πιό κοντά στον πελάτη και συνεπώς μπορεί να μεταβεί πιο γρήγορα κοντά στη θέση του πελάτη για να τον εξυπηρετήσει.

Δεδομένα Εισόδου

Λαμβάνουμε ως δεδομένα τις συντεταγμένες X,Υ (γεωγραφικό μήκος και γεωγραφικό πλάτος αντίστοιχα) για τον πελάτη, για τα ελεύθερα ταξί και για διάφορα σημεία των οδών. Κατά το στάδιο του σχεδιασμού λαμβάνουμε υπ' όψιν ότι τα σημεία κάθε οδού δίνονται ακολουθιακά και πως κάποια σημεία αντιστοιχούν σε σημεία τομής με άλλες οδούς. Επομένως, ένα σημείο μπορεί να ανήκει σε περισσότερες από μια οδούς. Τα δεδομένα δίνονται σε τρία αρχεία τύπου .csv, στα αρχεία client, taxis, nodes τα οποία περιέχουν τις συντεταγμένες της τοποθεσίας του πελάτη, τις συντεταγμένες και το id του κάθε ταξί και τις συντεταγμένες και το όνομα της οδού του κάθε κόμβου, αντίστοιχα.
Τα δεδομένα εντός των αρχείων .csv δεν υπέστησαν κάποια προεπεξεργασία. Ωστόσο, οι συντεταγμένες X,Υ ήταν εισαγμένες σε ανάποδη σειρά. Δηλαδή αν εισαχθούν στο Google Maps αντιστοιχούν σε πολυ μακρινές τοποθεσίες, ενώ αν εισαχθούν ανάποδα αντισοτιχούν σε κόμβους εντός της Αθήνας το οποίο είναι και το επιθυμητό.

Σκιαγράφηση Λύσης

Για την επίλυση του προβλήματος και την υλοποίηση της ευφυούς υπηρεσίας υλοποιήσαμε μια παραλλαγή του γνωστού αλγορίθμου A^*. Η υλοποίηση του αλγορίθμου A* έγινε στη γλώσσα προγραμματισμού Java (αναλυτικά οι κλάσσεις και οι δομές δεδομένων που χρησιμοποιήθηκαν θα παρουσιαστούν παρακάτω). Ως ευριστική συνάρτηση χρησιμοποιούμε την ευκλείδια απόσταση μεταξύ των σημείων. Για την υλοποίηση του αλγορίθμου χρησιμοποιούμε τις έννοιες του κλειστού συνόλου και του μετώπου αναζήτησης. Στο κλειστό σύνολο εισάγονται οι κόμβοι που ήδη έχουν εξερευνηθεί, ενώ στο μέτωπο αναζήτησης εισάγονται οι υπό εξερεύνηση κόμβοι. Κάθε κόμβος του προβλήματος έχει τα εξής χαρακτηριστικά μεγέθη: Την ευριστική συνάρτηση H που περιέχει την εκτιμώμενη απόσταση (ευκλείδια απόσταση) του κάθε κόμβου από τον κόμβο στόχο. Την πραγματική απόσταση G που έχει διανυθεί απο τον κόμβο αφετηρίας μέχρι κάθε κόμβο. Τέλος την συνάρτηση F=G+H που περιγράφει το άθροισμα της ευρυστικής συνάρτησης και της κανονικής απόστασης.
Η παρούσα υλοποίηση του αλγορίθμου A^* διαφέρει από τον κλασσικό αλγόριθμο A^* ως προς το γεγονός πως αν υπάρχουν περισσότερες από μία ισοδύναμες εναλλακτικές διαδρομές από πλευράς κόστους καταφέρνει να τις βρίσκει και να τις επιστρέφει. Για να είναι εφικτό να επιστρέφει ο αλγόριθμος τις ισοδύναμες διαδρομές (αν υπάρχουν), έχει προσαρμοστεί έτσι ώστε κατά την αναζήτηση, αν βρίσκεται σε κάποιον κόμβο από τον οποίο μπορεί να μεταβεί σε κάποιον επόμενο που βρίσκεται ήδη στο μέτωπο αναζήτησης με το ίδιο κόστος, να διατηρεί όλους τους προγόνους από τους οποίους μπορεί να φτάσει στον συγκεκριμένο κόμβο. Επομένως, κάθε κόμβος πλέον δεν έχει απαραίτητα έναν πατέρα -όπως στην κλασσική υλοποίηση- αλλά περισσότερους από έναν οι οποίοι αποθηκεύονται σε μια λίστα.
Παραδοχές:
• Όλες οι οδοί είναι διπλής κατεύθυνσης.
• Όλες οι οδοί επιτρέπουν την ίδια ταχύτητα κίνησης.
• Όλα τα σημεία τομής είναι σημεία διασταύρωσης.
• Όλα τα δεδομένα που σας δίνονται είναι οδοί όπου μπορούν να κινηθούν τα ταξί παρά το γεγονός ότι λόγω του απλουστευμένου τρόπου λήψης των δεδομένων, σε αυτά μπορεί να περιέχονται και άλλες γεωγραφικές γραμμές που δεν αντιστοιχούν σε οδούς (πχ. μονοπάτια, υδάτινες οδοί, όρια περιοχών, κλπ).

Περιγραφή Υλοποίησης

Περιγραφή Παραλλαγής του Αλγορίθμου A*

Σε αυτό το σημείο παρουσιάζεται η τροποποιημένη εκδοχή του αλγορίθμου A* ώστε να υπολογίζει εναλλακτικές ισοδύναμες συντομότερες διαδρομές (frontier ονομάζουμε το μέτωπο αναζήτησης και close το κλειστό σύνολο):

(1) Όσο το *frontier* δεν είναι κενό, αποθηκεύουμε το πρώτο του στοιχείο ως *current*, το προσθέτουμε στο *close* και το αφαιρούμε από το *frontier*.

(2) Αν το στοιχείο είναι το node του πελάτη, τότε ο αλγόριθμος τερματίζει.

(3) Διαφορετικά με τη βοήθεια ενός HashMap εξετάζουμε ξεχωριστά όλα τα παιδιά του current. Για κάθε παιδί:

- Αν εμπεριέχεται ήδη στο *close*, συνεχίζουμε στο επόμενο παιδί-γείτονα,
- Διαφορετικά, αν δεν εμπεριέχεται ήδη στο *close*, τότε:
  - Αν δεν εμπεριέχεται στο *frontier*, το προσθέτουμε, εξετάζοντας αν η προσθήκη οδήγησε στην επέκταση της λίστας και αποθηκεύουμε την νέα τιμή του μεγέθους της. Επίσης, ανανεώνουμε τα πεδία των nodes των παιδιών ως εξής:
    RealValue παιδιού = *RealValue πατέρα* + Πραγματική Απόσταση παιδιού-πατέρα (το βάρος της ακμής στον γράφο που τους συνδέει θεωρητικά)

    Αθροισμα παιδιού = Νέο *RealValue* παιδιού + HeuristicValue παιδιού.

    Τέλος, ανανεώνουμε την λίστα *previousPoint*, προσθέτοντας τον current.
  - Αν εμπεριέχεται ήδη στο *frontier* ελέγχουμε:
    * Αν το άθροισμα *RealValue πατέρα* + Απόσταση Πατέρα-Παιδιού είναι **μικρότερο** από το *RealValue παιδιού* και στην περίπτωση αυτή γίνεται *clear()* της λίστας *previousPoint* του παιδιού και μετά προσθήκη του current σε αυτήν, αφού υπάρχει μονοπάτι που φτάνει με λιγότερο κόστος πλέον στο παιδί.
    * Αν το textitRealValue πατέρα + Απόσταση Πατέρα-Παιδιού είναι **ίσο** με το *RealValue παιδιού*, έχουμε περίπτωση ισοβαθμίας και ανανεώνουμε την λίστα *previousPoint*, προσθέτοντας τον current.
    * Αν είναι **μεγαλύτερο** δεν έγινε καμία ανανέωση πεδίου και συνεχίζουμε στην εξέταση του επόμενου παιδιού-γείτονα.

Υλοποίηση Συναρτήσεων Εκτίμησης Απόστασης

Ως Ευριστική Συνάρτηση στην υλοποίηση μας επιλέξαμε την Ευκλίδεια απόσταση στο δισδιάστατο χώρο. Επομένως για κάθε σημείο του γράφου επιλέγουμε ως ευριστική συνάρτηση την απόσταση μεταξύ αυτού του σημείου και του σημείου στόχου που βρίσκεται ο πελάτης. Η επιλογή έγινε λόγω της απλότητας της στην υλοποίηση και του υψηλού ποσοστού επιτυχίας που έχει, όπως φαίνεται και στα αποτελέσματα παρακάτω.

Δομές Δεδομένων

Ο γράφος-χάρτης λοιπόν υλοποιείται με HashMap λόγω της ιδιότητας του να επιτελεί την αναζήτηση για τυχαίο σημείο σε **χρόνική πολυπλοκότητα Θ(1)**. Αυτή η ιδιότητα είναι πολύ χρήσιμη διότι κατά το διάβασμα των συντεταγμένων των σημείων πρέπει να γίνει γρήγορος έλεγχος για το αν υπάρχει ένα σημείο-node, αφού με τον τρόπο αυτό δηλώνονται οι διασταυρώσεις.
Εκτός του HashMap, όλες οι λίστες που απαιτούνται για την ανάπτυξη της εφαρμογής υλοποιούνται με την επιλογή ArrayList της Java, χάριν απλότητας και αποτελεσματικότητας της υλοποίησης της.

Προεπεξεργασία δεδομένων

Δεδομένου πως οι συντεταγμένες δίνονται σε μεταβλητές τύπου double πρέπει έμμεσα ή άμεσα να γίνει προεπεξεργασία των X,Υ για να υπάρχει η δυνατότητα δημιουργίας ισοδύναμων διαδρομών. Αυτό επιτυγχάνεται με τη στρογγυλοποίηση των αποστάσεων όπως αναλύεται παρακάτω. Οι μέθοδοι *distance()* και *heuristic()*, επιστρέφουν μία double τιμή στρογγυλοποιημένη στο 1ο δεκαδικό ψηφίο με χρήση της customed μεθόδου *round()*, με απώτερο σκοπό την αύξηση των πιθανοτήτων για τυχόν ισοβαθμίες μεταξύ αποστάσεων κατά την διάρκεια της αναζήτησης του αντίστοιχου μονοπατιού, και κατ'επέκταση της επίτευξης και προβολής shortest paths με ίδιο optimal κόστος, ως εναλλακτικές διαδρομές.
Επομένως, παρα το γεγονός πως δεν γίνεται κάποια άμεση προεπεξεργασία στα δεδομένα, η στρογγυλοποίηση των αποστάσεων τα τροποποιεί έμμεσα.

Περιγραφή Κλάσεων που χρησιμοποιήθηκαν

Για την υλοποίηση του ευφυούς συστήματος υλοποιήθηκαν συνολικά 9 κλάσεις με σκοπό την σωστή ανάλυση του προβλήματος και την απλούστευση της κατανόησης του κώδικα. Η λειτουργία της κάθε κλάσης παρουσιάζεται αναλυτικά παρακάτω (έχουν συμπεριληφθεί λεπτομέριες υλοποίησης ώστε ο κώδικας να γίνει πλήρως αντιληπτός σε οποιονδήποτε προσπαθήσει να τον διαβάσει):

(1) **Point:** Η κλάση Point αντιπροσωπεύει ένα node του χάρτη και περιλαμβάνει:

- Δύο μεταβλητές τύπου *double (x,y)* για την αποθήκευση των συντεταγμένων κάθε σημείου (node). Μια μεταβλητή τύπου *double (distanceCovered)*, η οποία συμβολίζει την απόσταση σε μέτρα που έχει διανυθεί μέχρι τη δεδομένη στιγμή κατά την αναζήτησης του shortest path και αρχικοποιείται με άπειρο. Μια μεταβλητή τύπου *double (distanceNeeded)*, η οποία συμβολίζει την εκτιμώμενη απόσταση σε μέτρα που απομένουν μέχρι τον πελάτη από το συγκεκριμένο node και αρχικοποιείται με άπειρο.
- Μία λίστα τύπου *ArrayList (previousPoint)*, η οποία περιέχει αντικείμενα τύπου *Point*. Σημειώνεται ότι παραπάνω από ένας πατέρας μπορεί να προκύψει σε περίπτωση ισοβαθμίας την οποία αντιμετωπίζει ο τροποποιημένος αλγορίθμος A*. Η σπουδαιότητα χρήσης της λίστας έγκειται στην ανακατασκευή του μονοπατιού διασχίζοντας όλους τους δυνατούς πατεράδες από το node-στόχο σε ανάποδη σειρά. Περισσότερη επεξήγηση στην **AstarSearch** κλάση.
- Τη μέθοδο *distance()*, η οποία επιστρέφει σε double τιμή την απόσταση σε μέτρα (ακτινωτή προσέγγιση) μεταξύ δύο οποιονδήποτε αντικειμένων τύπου *Point*.
- Τη μέθοδο *heuristic()*, η οποία δέχεται ως όρισμα τις συντεταγμένες του πελάτη και επιστρέφει την απόσταση σε μέτρα (ακτινωτή προσέγγιση) μεταξύ ενός αντικειμένου τύπου *Point* και του πελάτη.
- Τη μέθοδο *reset()* η οποία κάνει clear το *previousPoint* και θέτει τις τιμές *distanceCovered* και *distanceNeeed* ξανά ίσες με άπειρο. Ο λόγος που χρειάζεται η reset είναι πως μετά την εκτέλεση του αλγορίθμου για ένα ταξί, πρέπει τα πεδία όλων των points του χάρτη που συμμετείχαν στο μονοπάτι να γίνουν clear() πριν εκτελεστεί ο αλγόριθμος για το επόμενο ταξί. Διαφορετικά ο υπολογισμός των αποστάσεων θα ήταν λανθασμένος για κάποια ταξί (διότι κοντά στο σημείο-στόχο του πελάτη θεωρητικά θα μπορούσαν να υπάρχουν το μέγιστο 5 διαφορετικοί δρόμοι-κόμβοι που να το περιέχουν και άρα μόνο 5 διαφορετικές διαδρομές χωρίς overlap, πράγμα αδύνατο έφοσον γίνεται μελέτη για 11 ταξί).
- Έγινε override των μεθόδων *equals()* και *hashCode()* προκειμένου η σύγκριση αντικειμένων objects να γίνεται μέσω των συντεταγμένων τους.

(2) **ClientLocation:** Περιλαμβάνει:

- Δύο μεταβλητές τύπου *double (x,y)* για την αποθήκευση των συντεταγμένων X,Y του πελάτη και μία μεταβλητή τύπου *Point (ending_node)* για το κοντινότερο node στον πελάτη
- Τη μέθοδο *parser()* τύπου void, η οποία θα κληθεί με όρισμα το αρχείο 'client.csv' για το διάβασμα των συντεταγμένων του πελάτη.

- Τη μέθοδο *findEndingNode()* η οποία δέχεται ως όρισμα ένα αντικείμενο τύπου *NodeLocation*, δηλαδή το χάρτη-γράφο με τα προεπεξεργασμένα nodes και επιστρέφει ένα object τύπου *Point* με το κοντινότερο node στον πελάτη. Ο λόγος που χρειάζεται αυτή η μέθοδος είναι πως ο πελάτης δε βρίσκεται απαραίτητα σε κάποιο node του γράφου, συνεπώς πρέπει να βρεθεί η απόσταση μεταξύ του πελάτη και του κοντινότερου σημείου απο το οποίο θα μπορούσε να τον παραλάβει το ταξί με την συντομότερη διαδρομή.
- Τη μέθοδο *distance()*, η οποία επιστρέφειτην απόσταση σε μέτρα (ακτινωτή προσέγγιση) μεταξύ του αντικειμένου τύπου *ClientLocation* του πελάτη και ενός τύπου *Point*
- Τον constructor *ClientLocation()* που δέχεται ως όρισμα το αρχείο 'client.csv' και τον "χάρτη" με τα nodes και καλεί τις προαναφερθείσες μεθόδους για τους παραπάνω υπολογισμούς.

(3) **TaxiLocation:** Περιλαμβάνει:

- Δύο μεταβλητές τύπου *double (x,y)* για την αποθήκευση των συντεταγμένων X,Y του ταξί και μία μεταβλητή τύπου *int (id)* για την αποθήκευση του id του ταξί. Μία μεταβλητή τύπου *Point (starting_node)* για το κοντινότερο node στο προς εξέταση ταξί
- Τη μέθοδο *distance()*, η οποία επιστρέφει σε double τιμή την απόσταση σε μέτρα (ακτινωτή προσέγγιση) μεταξύ του αντικειμένου τύπου *TaxiLocation* του ταξί και ενός τύπου *Point*
- Τη μέθοδο *findEndingNode()* η οποία δέχεται ως όρισμα ένα αντικείμενο τύπου *NodeLocation*, δηλαδή το "χάρτη" με τα προεπεξεργασμένα nodes και επιστρέφει ένα object τύπου *Point* με το κοντινότερο node στο προς εξέταση ταξί. Ο λόγος που χρειάζεται αυτή η μέθοδος είναι πως τα ταξί δε βρίσκονται απαραίτητα σε κάποιο node του γράφου, συνεπώς πρέπει να βρεθεί η απόσταση μεταξύ του ταξί και του κοντινότερου σημείου απο το οποίο μπορεί το ταξί να εκκινήσει στον αλγόριθμο.
- Τον constructor *TaxiLocaiton()* που δέχεται ως όρισμα τις πληροφορίες του προς εξέταση ταξί και τον "χάρτη" με τα nodes και καλεί τις προαναφερθείσες μεθόδους για τους παραπάνω υπολογισμούς.

(4) **Taxi:** Αποπειρόμαστε στην διάσπαση του γενικού task του project σε substasks, και με γνώμονα την ευελιξία εκ προθέσεως δεν ενοποιούμε τις λειτουργίες των **Taxi** και **TaxiLocation** σε μία ομαδοποιημένη κλάση.

- Μία λίστα τύπου *ArrayList (taxis)* που περιέχει αντικείμενα τύπου *TaxiLocation*, ένα για κάθε ταξί.
- Τη μέθοδο *parser()* τύπου void, η οποία θα κληθεί με όρισμα το αρχείο 'taxis.csv' με σκοπό το διάβασμα των συντεταγμένων και του id όλων ταξί. Για κάθε ταξί δημιουργεί ένα αντικείμενο τύπου *TaxiLocation* και το προσθέτει στην λίστα

(5) **NodeLocation:** Η κλάση NodeLocation αντιπροσωπεύει τον χάρτη-γράφο των σημείων-nodes και περιλαμβάνει:

- Μία μεταβλητή τύπου `HashMap<Point,ArrayList<Point>>(hashMap))` που περιέχει ως κλειδιά όλα τα σημεία του χάρτη και ως value όλους τους γείτονες του (αν ανήκουν στον ίδιο δρόμο ή έχουν ίδιο id).
- Τη μέθοδο *reset()* η οποία λειτουργεί σε συνάφεια με την μέθοδο *reset()* της κλάσης **Point** με την έννοια ότι η πρώτη δέχεται ως όρισμα το HashMap του χάρτη και καλεί την δεύτερη για κάθε κλειδί του HashMap αυτού, δηλαδή συνολικά για όλα τα points του χάρτη-γράφου.
- Τη μέθοδο *parser()* τύπου void, η οποία καλείται με όρισμα το αρχείο 'nodes.csv' και διαβάζει τις συνταγμένες και το id όλων των διαθέσιμων σημείων. Η συνάρτηση είναι επίσης υπεύθυνση για την δημιουργία του HashMap, που δομείται βάσει της εξής ιδιότητας: Εάν κάποιο σημείο-node έχει το ίδιο id με το προηγούμενο του, θεωρούνται γείτονες: Διαβάζουμε τις συντεταγμένες κάθε γραμμής του αρχείου 'nodes.csv' και κρατάμε το id της κάθε οδού. Ελέγχουμε αν το σημείο υπάρχει στο γράφο-χάρτη, Στην περίπτωση που υπάρχει, αποτελεί σημείο διασταύρωσης οπότε δεν χρειάζεται να τοποθετηθεί εκ νέου. Αν το hashMap περιέχει το τρέχον σημείο και το id της τρέχουσας οδού είναι ίδιο με το προηγούμενο προσθέτουμε στην `ArrayList<Point>` του συγκεκριμένου Point τον κόμβο.

(6) **AstarResult:** Η κλάση AstarResult ομαδοποιεί των στοιχείων που αποτελούν την λύση του προβλήματος, δηλαδή το shortest path για κάθε ένα από τα προς εξέταση ταξί. Σημειώνεται ότι για τις εναλλακτικές διαδρομές τα μόνα στοιχεία που αλλάζουν είναι τα nodes που το shortest path περιλαμβάνει, έτσι προκύπτουν διαδρομές με ίδια optimal απόσταση από το προς εξέταση ταξί μέχρι τον πελάτη.

- Δύο μεταβλητές τύπου *double (x,y)* για την αποθήκευση των συντεταγμένων κάποιου σημείου που θα εμπεριέχεται στο shortest path. Μία μεταβλητή τύπου *int (id)* για την αποθήκευση του id τους προς εξέταση ταξί. Μια μεταβλητή τύπου *double (dist)* για την ελάχιστη απόσταση σε μέτρα που διήνυσε το ταξί μέχρι τον πελάτη.
- Μία λίστα `ArrayList<double []>` με όνομα route για την αποθήκευση των συνεταγμένων όλων των σημειών που το shortest path ακολούθησε.

(7) **AstarSearch:** Υλοποιεί τον τροποποιημένο αλγόριθμο αναζήτησης A* όπως περιγράφεται παραπάνω (δηλαδή με την διαφορά ότι για σημεία που πληρούν την ιδιότητας της ισότητας, προσθέτουμε περισσότερους από έναν γονέα, γεγονός που μπορεί να οδηγήσει σε εναλλακτική διαδρομή). Χρήσιμοποιούνται τρείς λίστες `ArrayList<Point>`

- Μία λίστα με όνομα *frontier* η οποία είναι ταξινομημένη ως προς το άθροισμα των πεδιών *distanceCovered* και *distanceNeeded*,
- Μία λίστα με όνομα *close* η οποία αποθηκεύθει το κλειστό σύνολο,
- Μία λίστα με όνομα *children* η οποία αποθηκεύει τους γείτονες-παιδιά κάθε σημείου-node που εξετάζουμε

(8) **Astar:** Για κάθε ένα από τα ταξί καλεί την μέθοδο *Astar()* της **AstarSearch**, η οποία επιστρέφει το σημείο του πελάτη και αναδρομικά μπορούμε να οδηγηθούμε στην αρχή του ταξί με έναν ή περισσότερους τρόπους, ενώ σχηματίζουμε το ταυτόχρονα σχηματίζουμε το μονοπάτι. Συγκεκριμένα:

- Ορίζουμε την μέθοδο *findPaths()* η οποία θεωρητικά θα μπορούσε να είναι τύπου *void* αν επιδιώκαμε απλώς την εύρεση των διαφορετικών βέλτιστων μονοπατιών, αντ'αυτού είναι τύπου *int* και επιστρέφει τον αριθμό αυτών για λόγους στατιστικούς. Ξεκινάμε την αναζήτηση μας από το τελικό σημείο-node του πελάτη και μέσω του πεδίου λίστας *previousPoint* [1] Μάλιστα:
  (I) Δέχεται ορίσματα το σημείο-node εκκίνησης δηλαδή (του πελάτη κατά την αρχική κλήση) - εδώ το ονομάζουμε *end* -, το σημείο-node του ταξί (δεν αλλάζει κατά τις κλήσεις), τον εγγραφέα τύπου *PrintWriter (writer)*, μία λίστα `ArrayList<Point>(nextPoint)` για την απόθηκευση των όσων σημείων-κόμβων έχουμε συναντήσει μέχρι ένα ορισμένο σημείο εξερεύνησης και μία μεταβλητή τύπου *int (differentPath)* για τον αριθμό των διαφορετικών μονοπατιών
  (II) Για το σημείο-node *end* ορίζουμε την λίστα των γονέων, και όσο αυτή δεν είναι άδεια, βγάλε από την λίστα των προς εξέταση γονέα, τοποθέτησε τον στην λίστα *nextPoint*, κάλεσε αναδρομικά την *findPaths()* με *end* τον γονέα και την νέα *nextPoint* (τα υπόλοιπα ίδια), και τέλος αφαίρεσε από την λίστα *nextPoint* τον γονέα.
  (III) Η βάση της αναδρομής είναι η εξής: Εάν εξερευνώντας προς τα πίσω φτάσεις στο σημείο-node του ταξί, σταμάτησε, άρχισε να γράφεις στο αρχείο `"AllPossiblePaths_id_taxi.txt"` τις συντεταγμένες που περιέχει η λίστα *nextPoint* - άρα όλα τα σημεία μιας νέας διαδρομής - καθώς επίσης και το **"100000000"** που διαδραματίζει τον ρόλο διαχωριστικού ενός shortest path από το επόμενο για το προς εξέταση ταξί, ενώ τέλος αύξησε τον αριθμό των διαφορετικών μονοπατιών κατά 1.
- Ορίζουμε την μέθοδο *AstarTaxis()* που επιστρέφει ένα αντικείμενο τύπου `ArrayList<AstarResult>`, δηλαδή μίας λίστας από αντικείμενα τύπου *AstarResult*, καθένα από τα οποία αντιπροσωπεύει ένα shortest path για το συγκεκριμένο ταξί.

[1] Σημείωνεται ότι κάποια από τα παιδιά όντως βρέθηκαν στην if αυτή λόγω μικρότερης τάξης στρογγυλοποίσησης που έγινε στις αποστάσεις και απέκτησαν παραπάνω από έναν γονέα. Μικρότερο κομμάτι αυτών αποτέλεσε μέρος των σημείων-κόμβων που θα ανακληθούν recursively για την εύρεση του shortest path, ενώ κάποιο ακόμα μικρότερο αυτών απότελεσε όντως τμήμα του shortest path και λόγω ύπαρξης παραπάνω από ενός γονέα οδήγησε σε optimal shortest paths με ίδια απόσταση (με διαφορά μόνο στο μήκος του path)

(I) Γίνεται κλήση της μεθόδου *Astar()* το αποτέλεσμα της οποίας μεταβιβάζεται ως όρισμα στην κλήση της μεθόδου *findPaths* -αφού προηγηθεί ο ορισμός ενός writer τύπου *PrintWriter* που ανοίγει ένα αρχείο '.txt'. για την εγγραφή συντεταγμένων - η οποία θα επιστρέψει τον αριθμό των διαφορετικών μονοπατιών για το προς εξέταση ταξί, ενώ ταυτόχρονα εγγράφει σε **ορθή σειρά** [2] στο αρχείο '.txt' τις συντεταγμένες αυτές με διαχωριστικό μεταξύ των διαφορετικών μονοπατιών.

(II) Ορίζουμε έναν reader τύπου *BufferedReader* με σκοπό το κατά γραμμή διάβασμα του '.txt' αρχείου που δημιοργήσαμε προηγουμένως, και όπως και στην περίπτωση της προεπερξεργασίας δεδομένων (θα επεξηγηθεί παρακάτω) έτσι και εδώ, γίνεται η κατάλληλη προεπεξεργασία και οι συνεταγμένες κάθε μονοπατιού τοποθετούνται με την μορφή αντικειμένου τύπου *double []* στην λίστα τύπου `ArrayList <double[] >(path)`, η οποία αφού περιέχει τις συνεταγμένες όλων των σημείων για κάποιο μονοπάτι, μεταβιβάζεται ως όρισμα στον constructor *AstarResult()* αντικείμενο του οποιού προστίθεται στην λίστα τύπου `ArrayList <AstarResult >(results)`

(III) Αφού επεκταθεί κατάλληλα η λίστα *results* για όλα τα μονοπάτια, επιστρέφεται

(9) **Main:** Η κλάση Main αποτελεί την κεντρική κλάση της υλοποίησης μας. Η λειτουργία της περιγράφεται παρακάτω:

- Δημιουργεί από ένα αντικείμενο τύπου NodeLocation, ClientLocation και Taxi και χρησιμοποιώντας τις μεθόδους Parser της κάθε κλάσσης εισάγει τα δεδομένα απο τα αρχεία τύπου .csv.
- Εν συνεχεία, τυπώνει στο Standard Output τις συντεταγμένες της τοποθεσίας που βρίσκεται αρχικά ο πελάτης, αλλά και τις συντεταγμένες του κόμβου στις οποίες πρέπει να μετακινηθεί ο πελάτης για να τον παραλάβει το ταξί.
- Έπειτα, καλεί τον αλγόριθμο A^* για κάθε ένα από τα ταξί και τυπώνει τις εξής πληροφορίες στο Standard Output:
  - Μέγιστο Μέγεθος Μετώπου Αναζήτησης
  - Πλήθος διαφορετικών ισοδύναμων μονοπατιών
  - Μέγεθος Βέλτιστου Μονοπατιού
  - Συνολική Απόσταση (εκφρασμένη σε μέτρα)
- Τέλος κατασκευάζει το αρχείο .kml σύμφωνα με τα πρότυπα του παραδείγματος που δώθηκε στην εκφώνηση, προκειμένου να γίνει η αναπαράσταση της λύσης στο Google Maps.

Σημαντικές Παρατηρήσεις

(1) Ακολουθήθηκαν οι εξής κανόνες για τα ονόματα μεθόδων και κλάσεων:

- Για τις κλάσεις: Κεφαλαίο το πρώτο γράμμα και κάθε επόμενης λέξης, ενωμένες χωρίς κενό ή underscore
- Για τις μεθόδους: Μικρό το πρώτο γράμμα και κεφαλαίο κάθε επόμενης λέξης, ενωμένες χωρίς κενό ή underscore

(2) Για την πλήρη κατανόηση της παραπάνω υλοποίησης, οι προδιαγραφές των κλάσεων και κατ'επέκταση των μεταβλητών και των μεθόδων που ενθυλακώνουν, πρέπει διαβαστούν συνδυαστικά.

(3) Παρά την ύπαρη επικάλυψης του παρόντος section με κάποια σημεία των παρακάτω sections, είναι σημαντικός ο διαχωρισμός αυτών για να δοθεί έμφαση στις λεπτομέρειες που απαιτούνται απο την εκφώνηση της αναφοράς

(4) Κατά την διάρκεια οποιουδήποτε open από αρχείου ή read/write σε αρχείου τοποθετούμε try-catch για την καταπολέμηση πιθανών Exceptions που μπορεί να προκύψουν (**IOException** ή **FileNotFound** αντίστοιχα)

[2] Η εγγραφή των σημείων-nodes σε σωστή σειρά ήταν κριτικό ζήτημα για την σωστή δημιουργία του μονοπατιού και κατ'επέκταση του kml αρχείου στην Main - Δεν αρκεί να διαθέτουμε τις συντεταγμένες του μονοπατιού απαιτείται να βρίσκονται και σε σειρά που αυτά ακολουθήθηκαν στο μονοπάτι

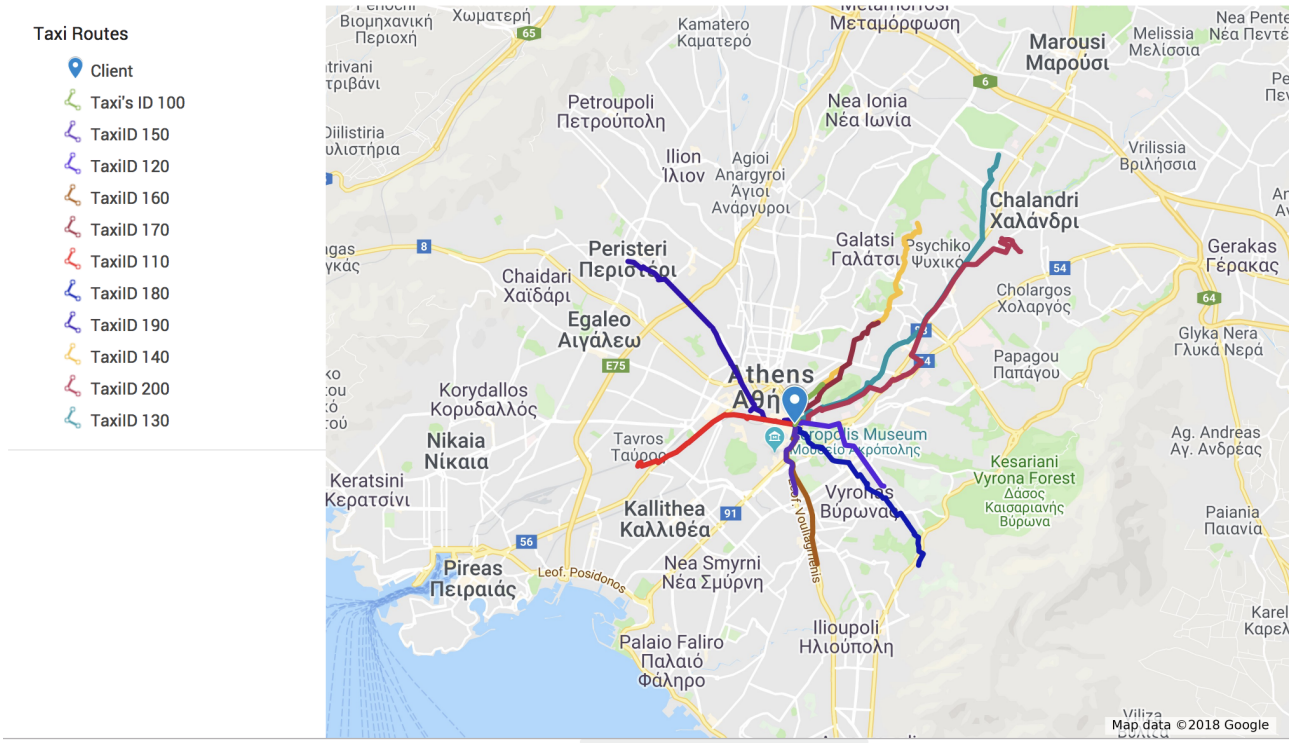
Αποτελέσματα - Οπτικές Αναπαραστάσεις

Παρακάτω επισυνάπτονται τα αποτελέσμα της εφαρμογής τόσο για το δοθέν Input της εκφώνησης όσο και για ένα δοκιμαστικό Input που δημιουργήθηκε από εμάς και επισυνάπτεται στο παραδοτέο:

Για το Input που δίνεται στην εκφώνηση έχουμε τα εξής αποτελέσματα:

Taxi ID	Μήκος Βέλτιστης Διαδρομής (m)	Μέγεθος Μονοπατιού	Μέγιστο Μέγεθος Μετώπου Αναζήτησης
100	47	1533.7	112
110	142	5189.8	328
120	91	3393.4	221
130	224	9517.6	347
140	215	7125.7	414
150	122	1822.7	140
160	166	3526.6	224
170	111	3847.1	273
180	218	5754.1	233
190	216	6614.4	450
200	277	8507.0	411

ProjectInput1

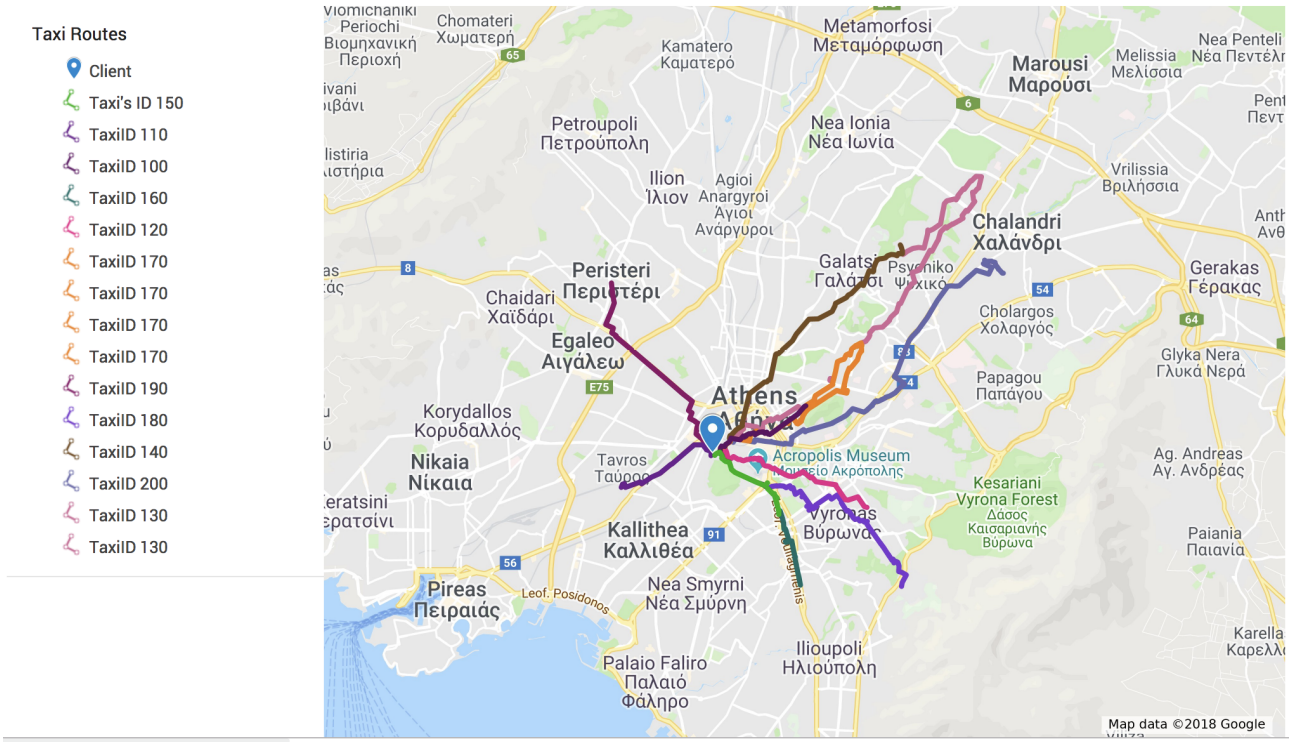


Παρατηρούμε πως η βέλτιστη διαδρομή είναι αυτή του Ταξί με TaxiID 100 (ζωγραφισμένη στο χάρτι με πράσινο χρώμα). Το ταξί απέχει 1.53 km από τον πελάτη και η διαδρομή περιλαμβάνει 47 μόνο κόμβους. Με τα υπόλοιπα χρώματα παριστάνονται οι άλλες διαδρομές.

Παρακάτω φαίνεται το αποτέλεσμα για ένα customized input με σκοπό να αναδείξουμε την λειτουργία του αλγορίθμου για εναλλακτικές διαδρομές:

Taxi ID	Μήκος Βέλτιστης Διαδρομής (m)	Μέγεθος Μονοπατιού	Μέγιστο Μέγεθος Μετώπου Αναζήτησης
100	129	3351.3m	258
110	87	3258.9m	179
120	254	5365.7m	345
130	369 377	11505.7m	402
140	308	8352.1m	514
150	118	2698.5m	209
160	164	4482.7m	249
170	214 209 241	5680.4m	362
180	260	7629.6m	315
190	157	6046.5m	385
200	383	10714.4m	439

ProjectInput2



- Παρατηρούμε πως η βέλτιστη διαδρομή είναι αυτή του Ταξί με TaxiID 150 (ζωγραφισμένη στο χάρτι με πράσινο χρώμα). Το ταξί απέχει 3,35 km από τον πελάτη και η διαδρομή περιλαμβάνει 129 μόνο κόμβους. Με τα υπόλοιπα χρώματα παριστάνονται οι άλλες διαδρομές.
- Παρατηρούμε ότι για τα ταξί με TaxiID 130 και 170, ο αλγόριθμος **έχει την ικανότητα** και βρίσκει συνολικά 2 και 3 βέλτιστες διαδρομές αντίστοιχα, άρα 1 και 2 εναλλακτικές αντίστοιχα, οι οποίες είναι ίδιας απόστασης, διαφέρουν σε αριθμό κόμβων και έχουν προφανώς overlapping για κάποια υποδιαδρομές.

Φαίνονταιμε το ίδιο χρώμα. Προτείνουμε την διαπίστωση τους με κατάλληλο άνοιγμα του αντίστοιχου **".kml"** αρχείου.

- Εάν οι μέθοδοι *distance()* και *heuristic()*, επέστρεφαν τιμή στρογγυλοποιημένη με χρήση της μεθόδου *ceil()* ή *floor()*, οι πιθανότητες για περισσότερες από μία εναλλακτικές διαδρομές αυξάνονται, καθώς ο αλγόριθμος αναζήτησης μπαίνει περισσότερες φορές στην **If** της ισότητας, άρα κάποια nodes έχουν περισσότερους πατεράδες στο field λίστα **previousPoint**, με αποτέλεσμα να είναι περισσότεροι οι δρόμοι που μπορούν να οδηγήσουν στο node του ταξί κατά την εκτέλεση της *findPaths()*. Άρα για την εύρεση εναλλακτικών διαδρομών σημαντική συνθήκη που πρέπει να πληρείται είναι η **στρογγυλοποίηση αποστάσεων**. Η πυκνότητα του χάρτη κόμβων για την πόλη της Αθήνας, ήταν καθοριστική για την εύρεση nodes που πληρούν το κριτήριο της ισότητας, άρα για το πρόβλημα της εκφώνησης η παραπάνω συνθήκη είχε νόημα.

- Επίσης αξίζει να παρατηρηθεί ότι για άλλα inputs που δώσαμε, ο αλγόριθμος έβρισκε εναλλακτικές οι οποίες διαφέρανε σε αριθμό κόμβων, αλλα πρακτικά είχαν πολύ μεγάλο ποσοστό overlapping, οπότε δεν τα ενσωμάτωσαμε στην αναφορά καθώς δεν θα αναδείκνυαν την ικανότητα του αλγορίθμου μας για εύρεση εναλλακτικών. Αντιθέτως, παρουσιάσαμε ένα πάραδειγμα input με μικρό ποσοστό overlapping μεταξύ των εναλλακτικών, το οποίο γίνεται επίσης αντιληπτό και σχηματικά μόνο μέσω της παραπάνω εικόνας.

- Ο αλγόριθμος δεν θα προτείνει όλες τις εναλλακτικές, καθώς ανάλογα με τον βαθμό στρογγυλοποίησης που έχει εφαρμοστεί μπορούν να βρεθούν περισσότερες ή λιγότερες, πράγμα που στην πράξη δεν έχει τόση μεγάλη σημασία. Αν και η ακρίβεια είναι σημαντικός παράγοντας, και έχει ληφθεί υπόψιν στον αλγόριθμο μας, μια απόκλιση της τάξης των 5 m, θεωρούμε ότι είναι επίσης αποδεκτή για τον πραγματικό κόσμο. Πάντως, για τις εναλλακτικές διαδρομές που βρίσκει ο αλγόριθμος, αυτές είναι βέλτιστες, καθώς ο A* βρίσκει πάντα τις βέλτιστες, εφόσον οι ευριστικές συναρτήσεις είναι αποδεκτές.