

# Τεχνικές Εξόρυξης Δεδομένων

## Άσκηση 2

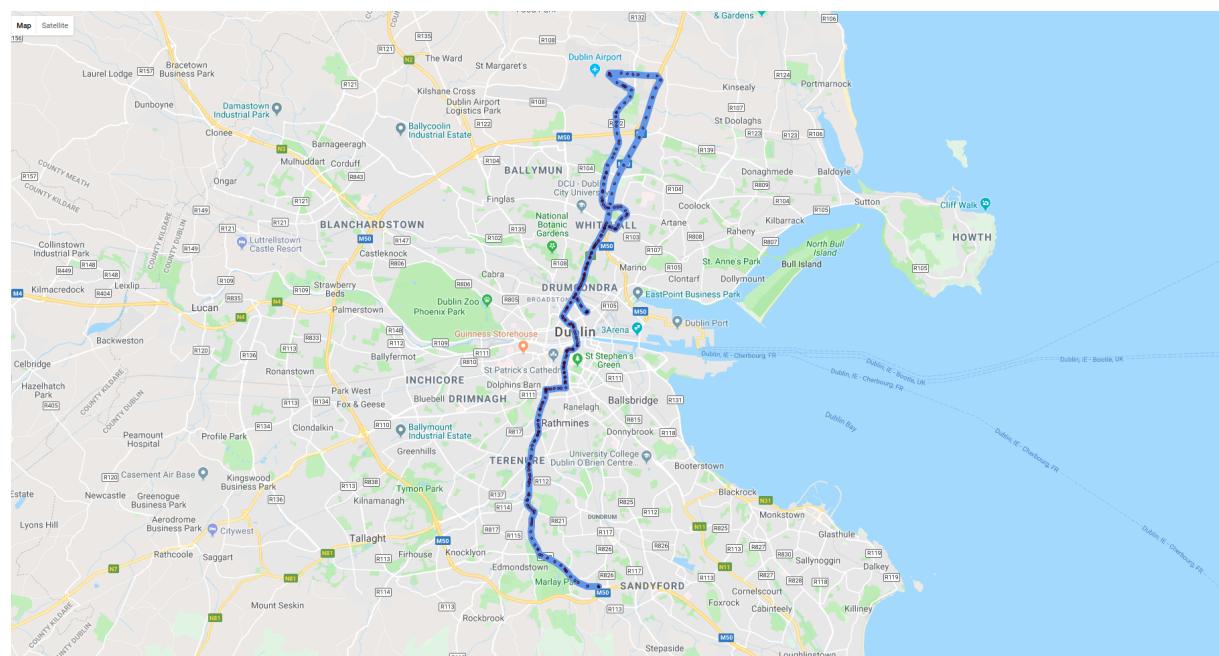
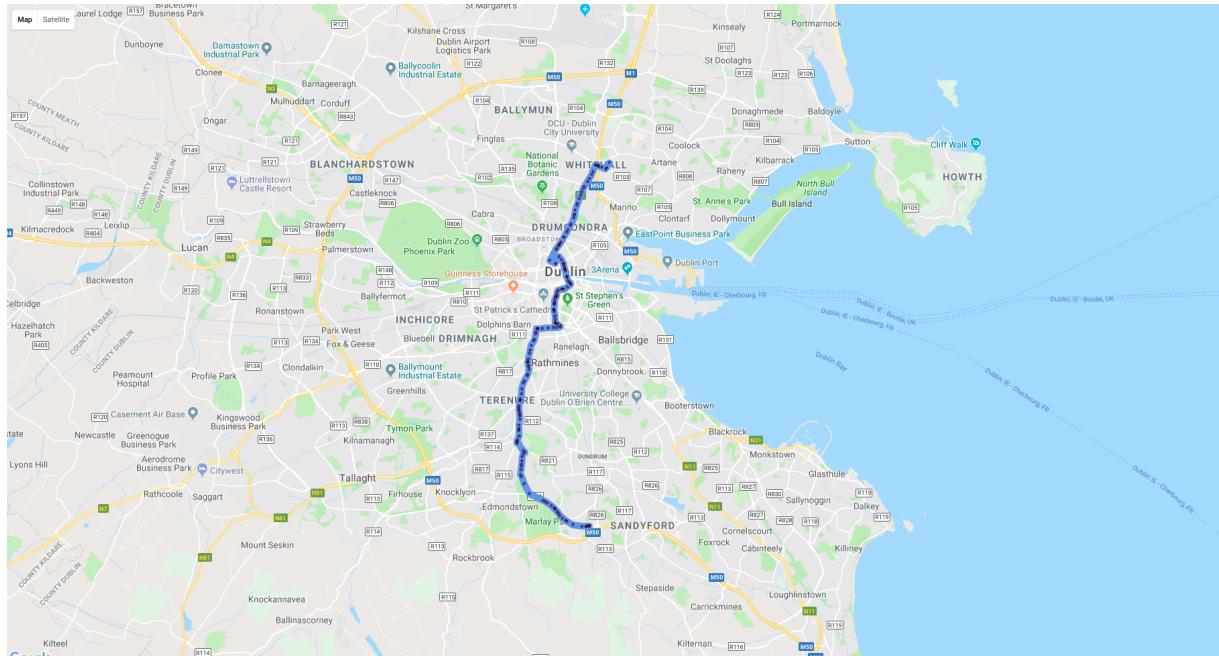
Τσεσμελής Δημήτρης – 1115201400208  
Φλωράκης Απόστολος – 1115201400217

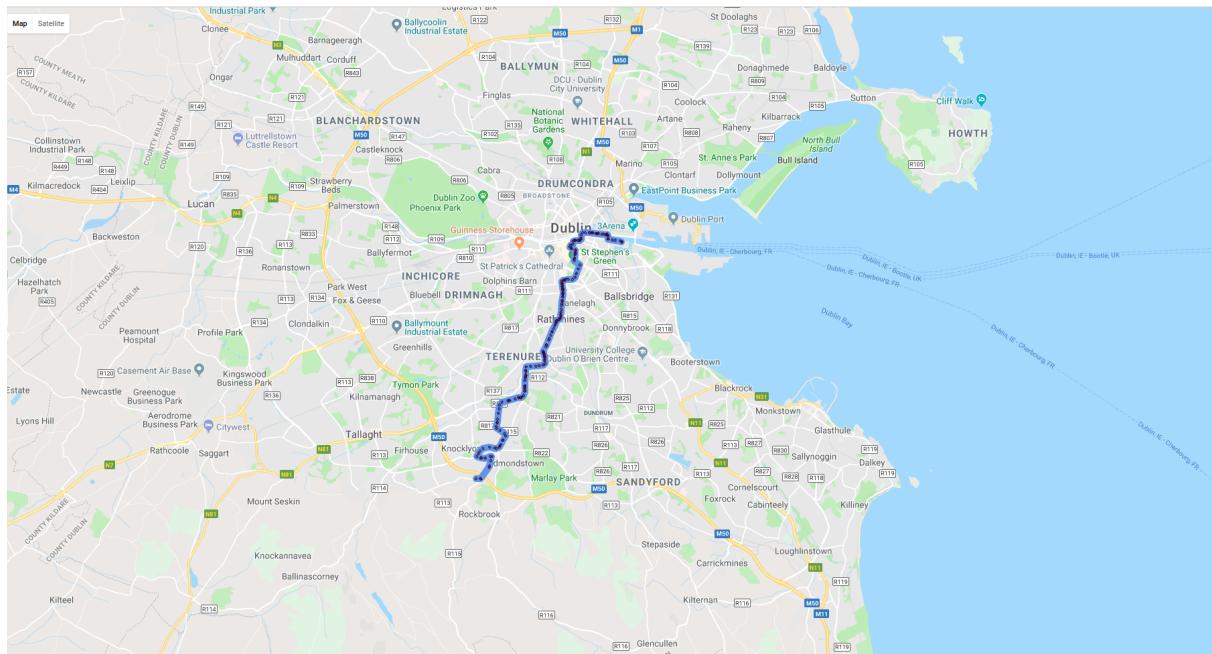
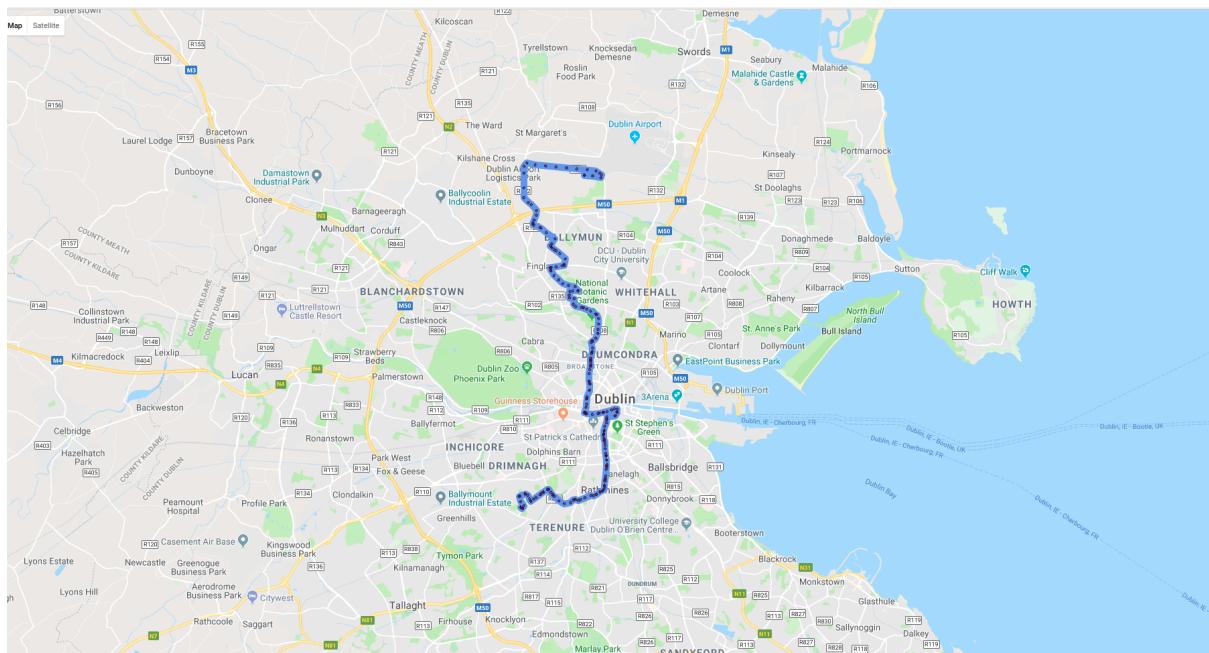
### Παραδοτέα αρχεία με κώδικα:

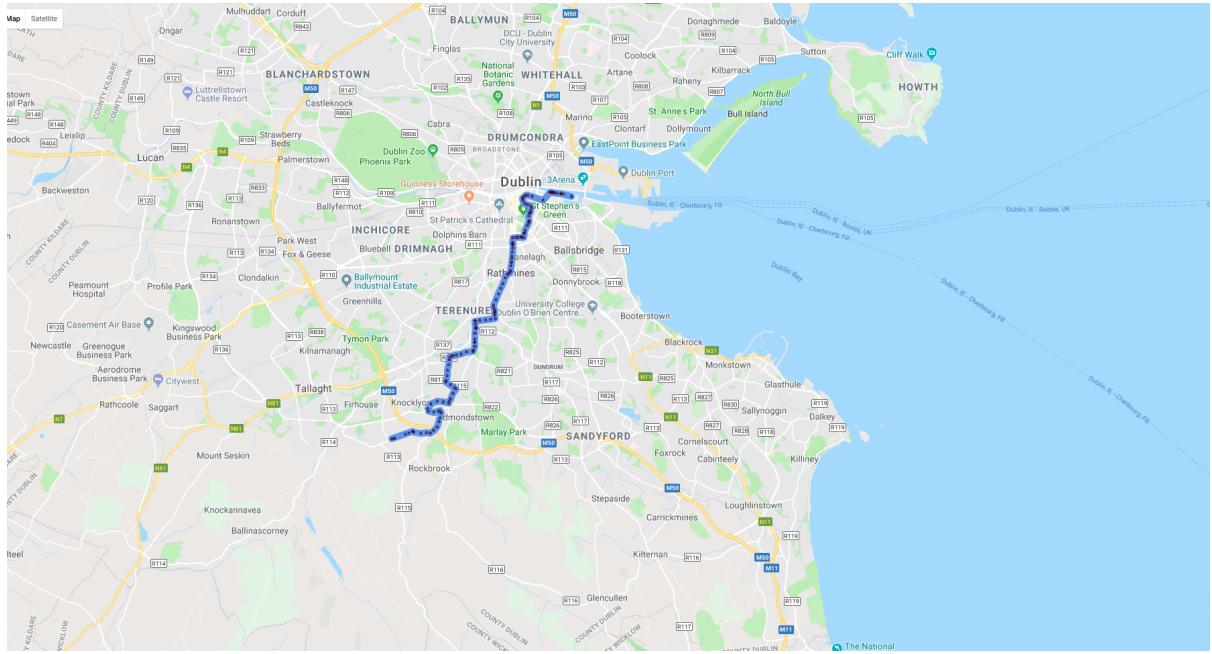
- **visualizer.py:** Το αρχείο αυτό οπτικοποιεί 5 διαφορετικές γραμμές λεωφορείων (journeyPatternID) από το αρχείο train\_set.csv. (Ερώτημα 1)
- **DTW.py:** Το αρχείο αυτό περιέχει την υλοποίηση του αλγορίθμου DTW. (Ερώτημα 2 A-1)
- **LCSS.py:** Το αρχείο αυτό περιέχει την υλοποίηση του αλγορίθμου LCSS (Ερώτημα 2 A-2)
- **KNN\_classifier.py:** Το αρχείο αυτό περιέχει την υλοποίηση του αλγορίθμου KNN (ίδια με αυτή της άσκησης 1, με μικρές τροποποιήσεις)
- **classification.py:** Στο αρχείο αυτό καλούνται οι συναρτήσεις που βρίσκονται στο αρχείο KNN\_classifier.py και πραγματοποιούν την cross\_validation και το prediction του άγνωστου dataset.
- **plotter.py:** Το αρχείο αυτό περιέχει κάποιες βοηθητικές συναρτήσεις που φτιάζαμε για την οπτικοποίηση των δεδομένων.

# 1. Οπτικοποίηση των Δεδομένων

Παρακάτω φαίνονται οι 5 διαφορετικές γραμμές λεωφορείων που προέκυψαν από τη βιβλιοθήκη gmplot:





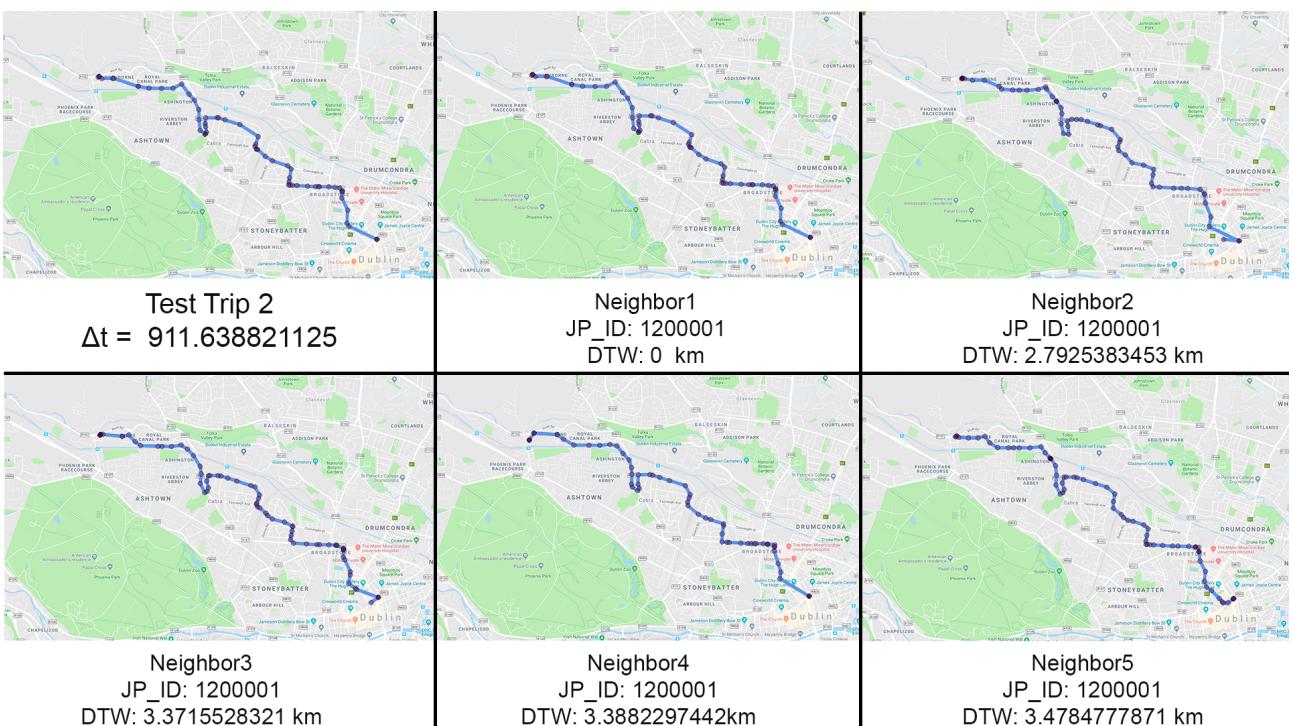
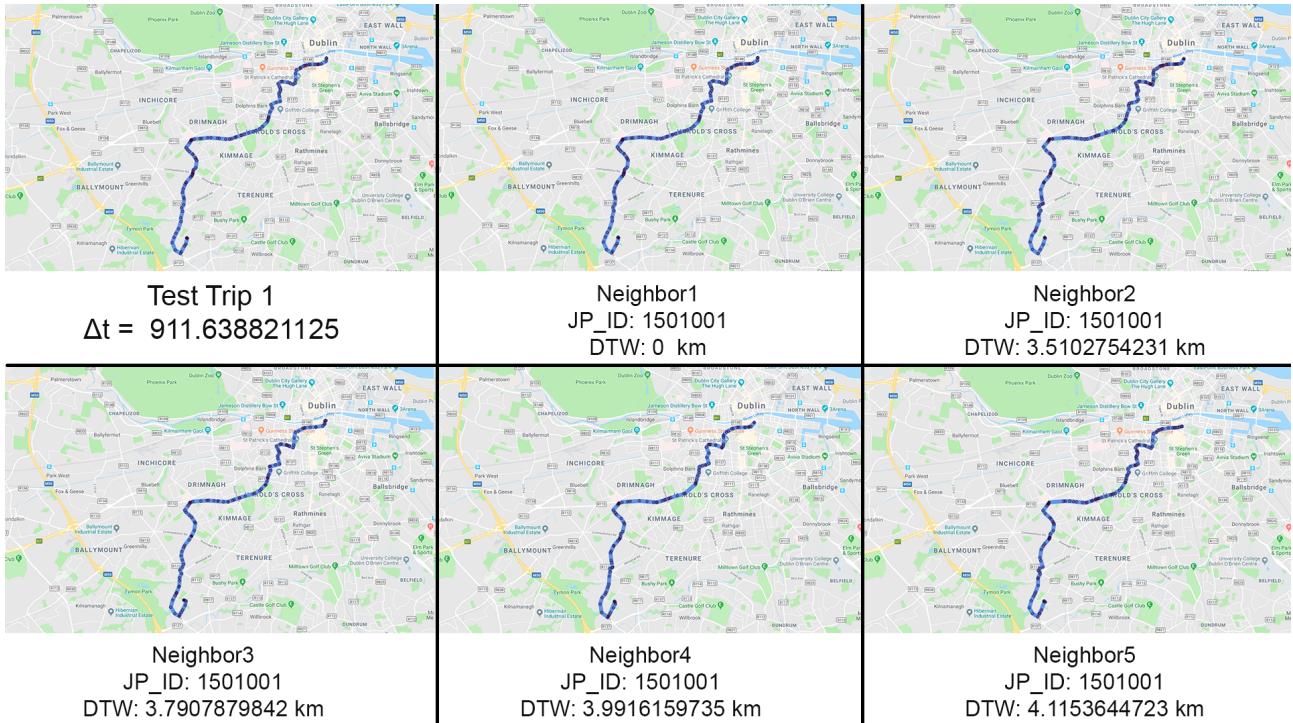


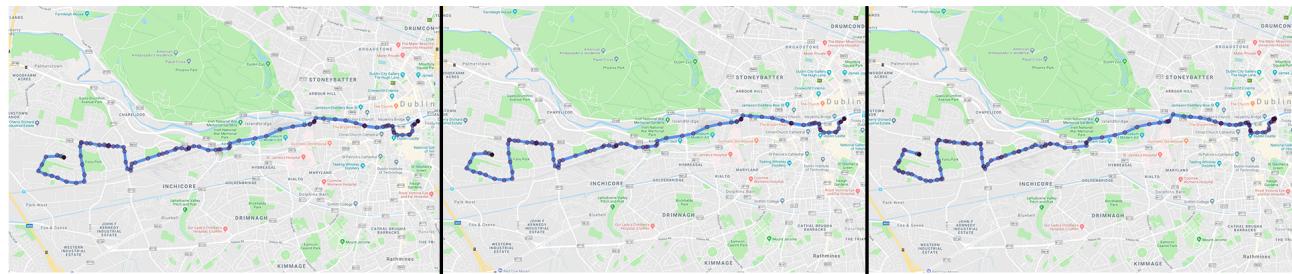
## 2 (Α-1). Εύρεση κοντινότερων γειτόνων

Έχουμε υλοποιήσει τον αλγόριθμο DTW για τον υπολογισμό της απόστασης μεταξύ δυο χρονοσειρών ορίζοντας τη συνάρτηση `DTW_distance(list1, list2)` βασιζόμενοι στον ψευδοκώδικα που υπάρχει στο παρακάτω σύνδεσμο: [https://en.wikipedia.org/wiki/Dynamic\\_time\\_warping](https://en.wikipedia.org/wiki/Dynamic_time_warping). Η συνάρτηση αυτή δέχεται ως ορίσματα δυο χρονοσειρές και υπολογίζει για κάθε μία από τις 6 διαδρομές του αρχείου `test_set_a1.csv` τους 5 κοντινότερους γείτονες, τις 5 δηλαδή πλησιέστερες διαδρομές σε αυτή, βάσει της DTW μετρικής, που βρίσκονται στο αρχείο `train_data.csv`. Αρχικά ορίζουμε ένα πίνακα διαστάσεων (`length(list1) + 1`) x (`length(list2) + 1`) κάθε κελί του οποίου (εκτός αυτών της πρώτης γραμμής και στήλης) αναπαριστά την απόσταση δυο στοιχείων των χρονοσειρών, και αρχικοποιούμε το στοιχείο της πρώτης γραμμής και στήλης (0,0) με 0 και όλα τα υπόλοιπα στοιχεία της πρώτης γραμμής και στήλης με `maxint` (το μέγιστο θετικό integer της python). Η αρχικοποίηση αυτή είναι απαραίτητη για ορισμένες συγκρίσεις που ο αλγόριθμος πραγματοποιεί στη συνέχεια. Έπειτα “γεμίζουμε” τον πίνακα αυτόν διατρέχοντας τον ανά γραμμή και ξεκινώντας από το στοιχείο (1,1). Για κάθε ζεύγος σημείων υπολογίζεται η απόσταση τους βάσει της μετρικής **haversine** (έχει χρησιμοποιηθεί η έτοιμη συνάρτηση `haversine()` της python) και ως τιμή του κελιού του πίνακα τίθεται η απόσταση αυτή αυξημένη με την ελάχιστη τιμή του αριστερού, του πάνω ή του πάνω-αριστερού κελιού του. Η διαδικασία επαναλαμβάνεται για όλα τα κελιά του πίνακα και στο τέλος η DTW απόσταση είναι αποθηκευμένη στο κάτω δεξιά κελί οπότε και μπορεί να επιστραφεί από τη συνάρτηση ως DTW απόσταση των `list1, list2`. Η απόσταση αυτή μαζί με τη διαδρομή του `train_data` που αντιστοιχεί αποθηκεύονται σε μια λίστα, η οποία

ταξινομείται κατά αύξουσα σειρά μετά τον υπολογισμό όλων των αποστάσεων και επιλέγονται οι 5 πιο μικρές. Η διαδικασία επαναλαμβάνεται για όλα τα στοιχεία του test\_set.

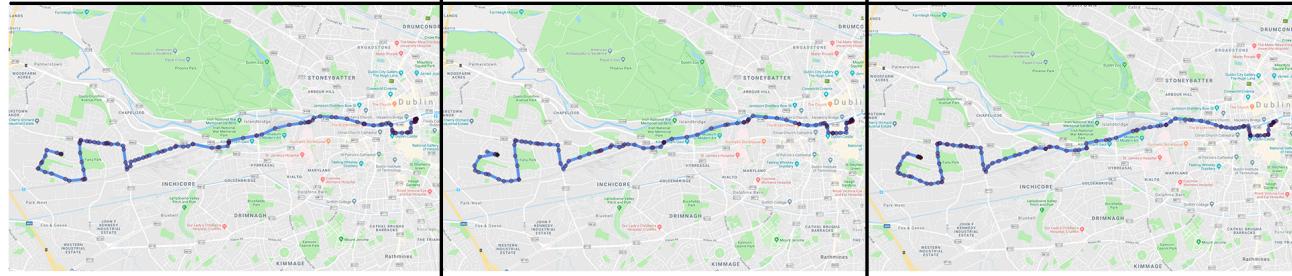
Τα αποτελέσματα που προέκυψαν για τις 5 test διαδρομές είναι τα παρακάτω:





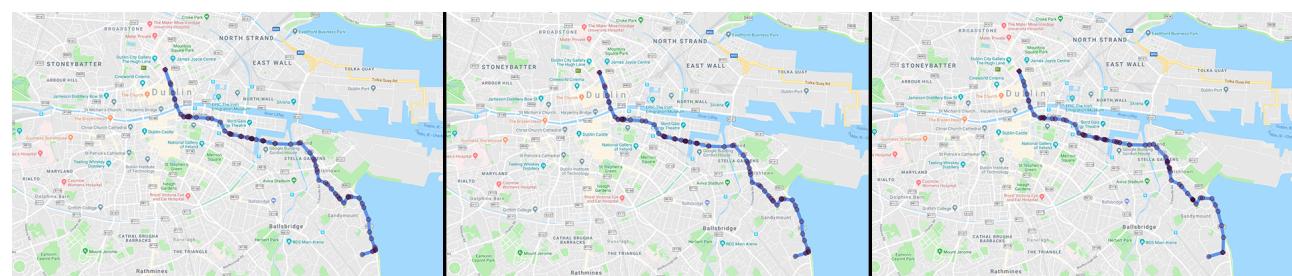
Neighbor1  
JP\_ID: 791001  
DTW: 0 km

Neighbor2  
JP\_ID: 791001  
DTW: 4.6923576557 km



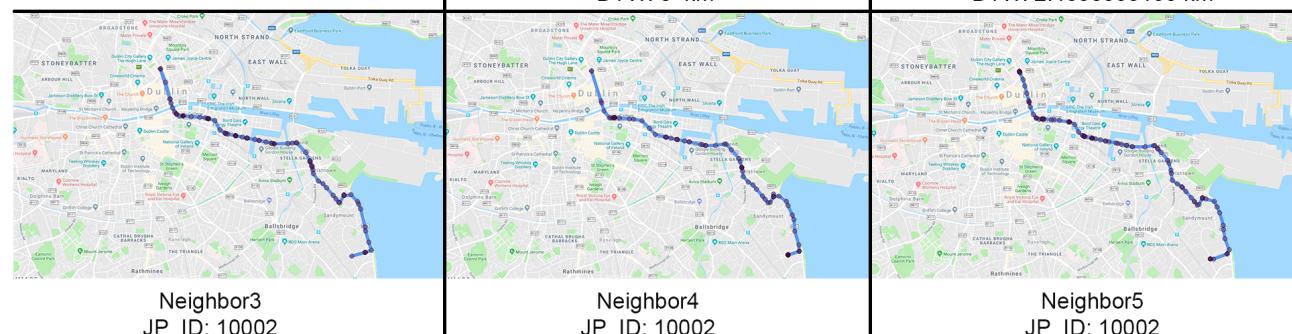
Neighbor4  
JP\_ID: 791001  
DTW: 4.7883232784 km

Neighbor5  
JP\_ID: 791001  
DTW: 4.7884998512 km



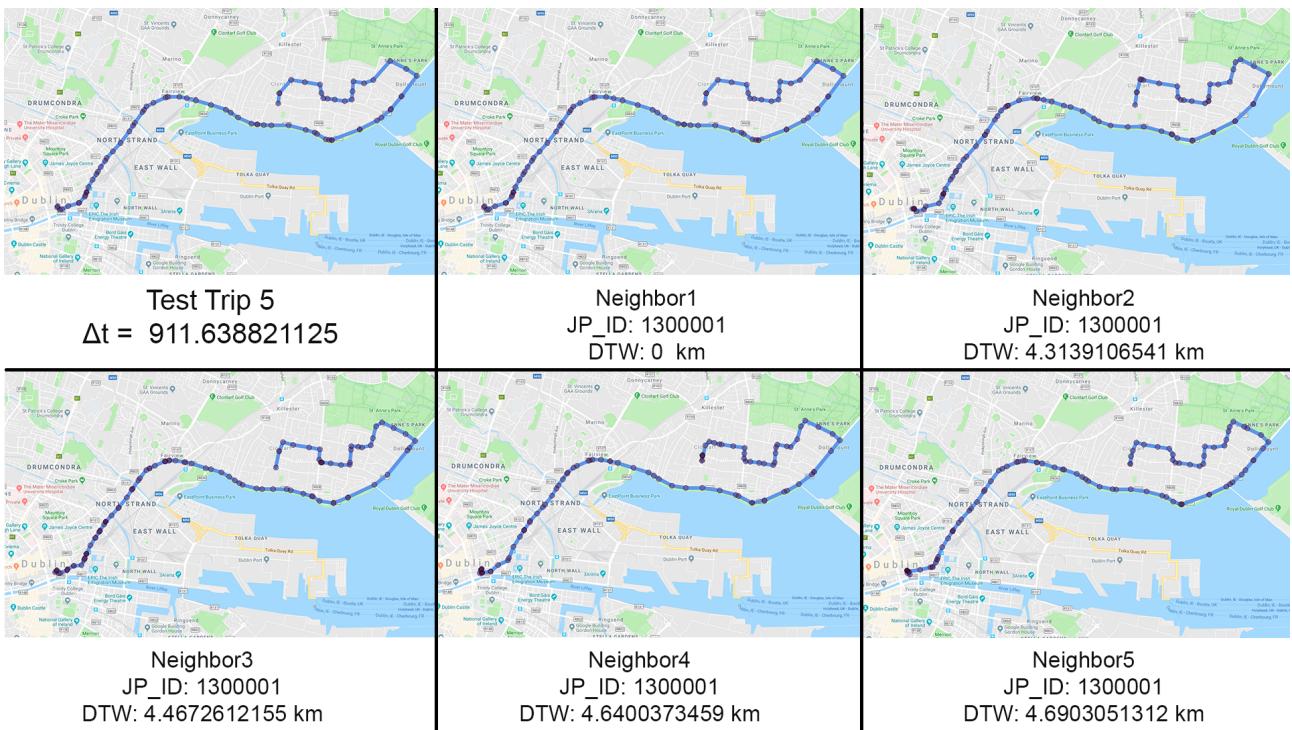
Neighbor1  
JP\_ID: 10002  
DTW: 0 km

Neighbor2  
JP\_ID: 10002  
DTW: 2.4555590198 km



Neighbor4  
JP\_ID: 10002  
DTW: 3.2106032984 km

Neighbor5  
JP\_ID: 10002  
DTW: 3.4581308973 km

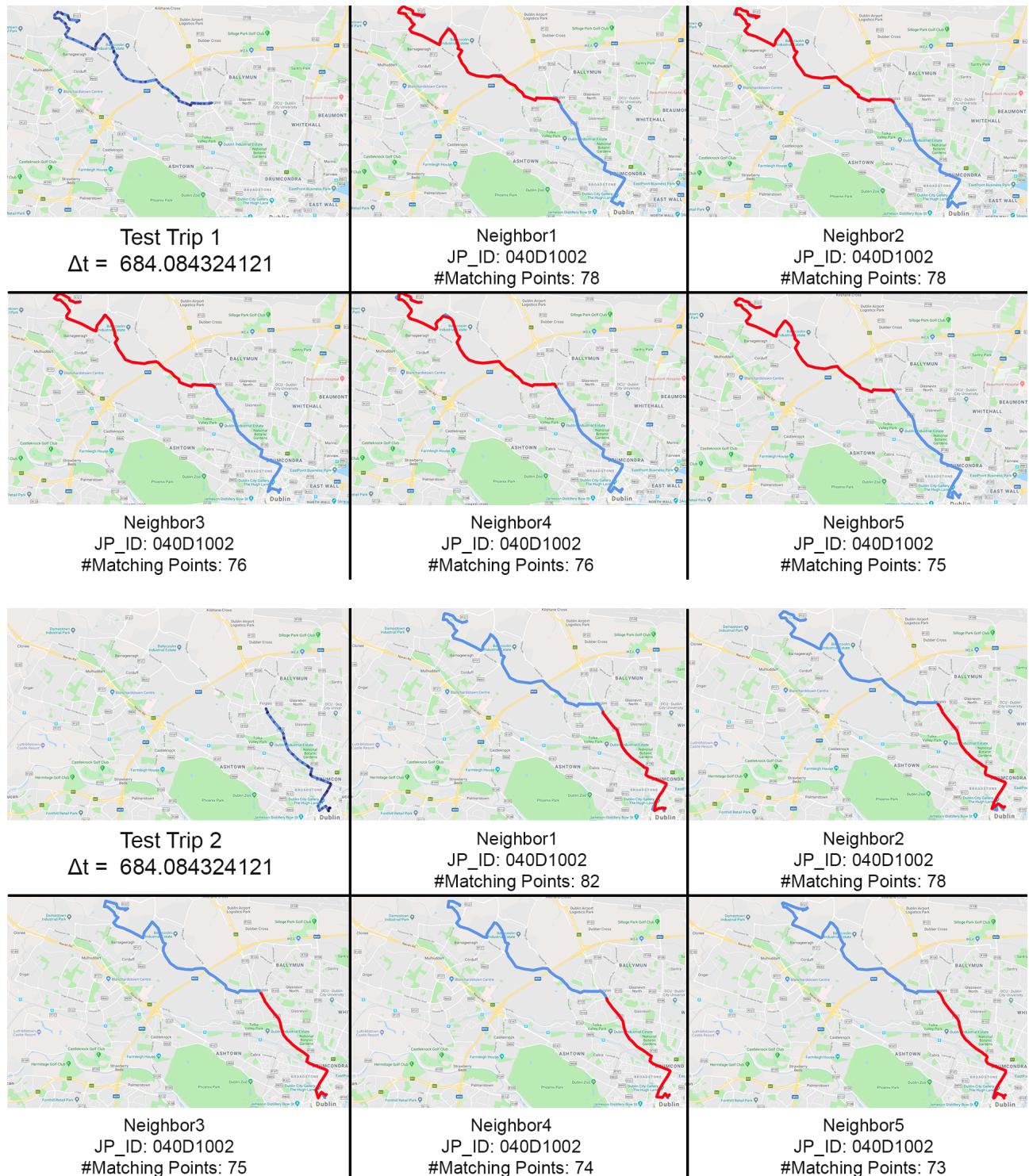


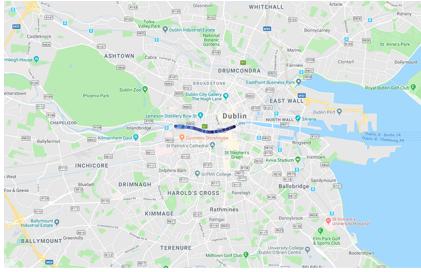
## 2 (A-2). Εύρεση κοντινότερων υποδιαδρομών

Για τον υπολογισμό της LCSS έχουμε υλοποιήσει συνάρτηση `find_LCSS(list1,list2)` η οποία υπολογίζει την Longest Common Subsequence μεταξύ δυο χρονοσειρών. Έχουμε βασιστεί σε ψευδοκώδικα που υπάρχει στον παρακάτω σύνδεσμο: [https://en.wikipedia.org/wiki/Longest\\_common\\_subsequence\\_problem](https://en.wikipedia.org/wiki/Longest_common_subsequence_problem). Αρχικά ορίζουμε πίνακα διαστάσεων  $(\text{length}(\text{list1}) + 1) \times (\text{length}(\text{list2}) + 1)$  αρχικοποιώντας την πρώτη γραμμή και στήλη του με τιμή 0. Διατρέχουμε τον πίνακα ξεκινώντας από το κελί (1,1) και εφόσον τα σημεία των λιστών `list1` `list2` που αντιπροσωπεύει πληρούν το κριτήριο `distance_matching` ως τιμή του κελιού ανατίθεται η τιμή του πάνω και αριστερά κελιού αυξημένη κατά 1. Σε διαφορετική περίπτωση ως τιμή του κελιού τίθεται η τιμή του πάνω αριστερά κελιού. Το κριτήριο `distance_matching` υλοποιείται μέσω της συνάρτησης `distance_matching(x, y)` η οποία επιστρέφει τιμή `True` όταν η απόσταση των σημείων είναι μικρότερη ή ίση των 0.2 km. Η προσέγγιση αυτή χρησιμοποιείται γιατί δυο σημεία διαδρομών είναι δύσκολο να συμπέσουν οπότε και θεωρούμε πως ταυτίζονται ακόμα και αν απέχουν απόσταση μικρότερη ή ίση των 0.2 km. Η διαδικασία επαναλαμβάνεται για όλα τα κελιά του πίνακα και τελικά στο κάτω δεξιά κελί υπολογίζεται το μέγεθος της μέγιστης κοινής επακολουθίας. Εφόσον αυτό δεν είναι μηδενικό, άμα δηλαδή υπάρχει κοινή υποκολουθία μεταξύ των δυο χρονοσειρών καλείται συνάρτηση `backtrack(matrix, list1, list2, i, j, result)`. Η συνάρτηση αυτή ξεκινάει από το κάτω αριστερά στοιχείο και καταλήγει στο πάνω δεξιά επιστρέφοντας τη μέγιστη κοινή επακολουθία στο `result`. Σε κάθε βήμα εφόσον υπάρχει `distance_matching` στο κελί που βρισκόμαστε, η τιμή ( οποιαδήποτε θέλουμε από τις δυο χρονοσειρές ) προστίθεται στο `result` και

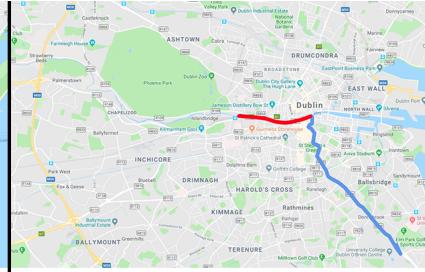
ως επόμενο κελί επιλέγεται το πάνω αριστερά, ενώ σε περίπτωσή που το κριτήριο δεν ισχύει επιλέγεται ως επόμενο κελί αυτό με τη μέγιστη τιμή μεταξύ του πάνω και του αριστερά. Για κάθε λοιπόν διαδρομή του αρχείου test\_set\_a2.csv υπολογίζονται όλες οι LCSS οι οποίες και ταξινομούνται κατά φθίνουσα σειρά οπότε και τελικά προκύπτουν οι 5 μεγαλύτερες σε μήκος κοινές επακολουθίες από όλες τις διαδρομές του train\_set.

Τα αποτελέσματα που προέκυψαν για τις 5 test διαδρομές είναι τα παρακάτω:

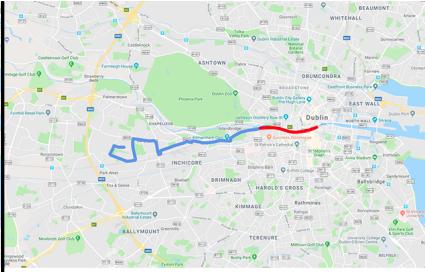




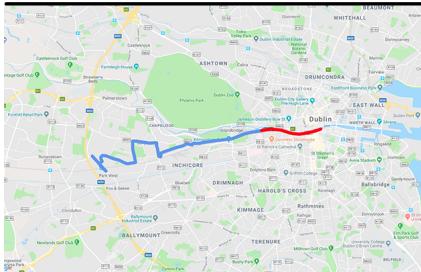
Test Trip 3  
Δt = 684.084324121



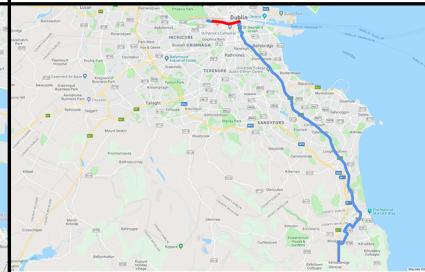
Neighbor1  
JP\_ID: 1451008  
#Matching Points: 40



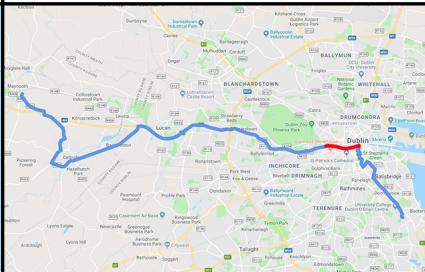
Neighbor2  
JP\_ID: 790001  
#Matching Points: 40



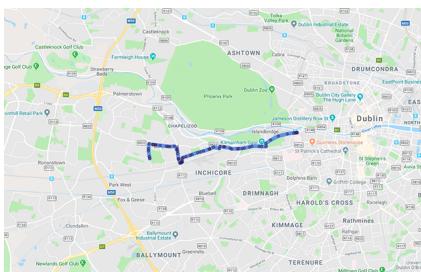
Neighbor3  
JP\_ID: 079A0001  
#Matching Points: 40



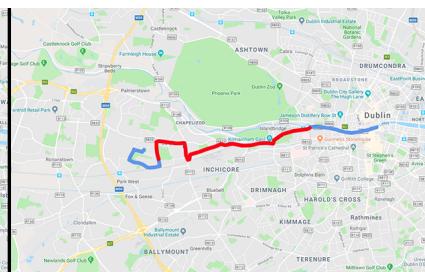
Neighbor4  
JP\_ID: 1451001  
#Matching Points: 40



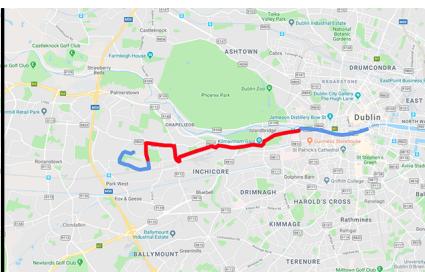
Neighbor5  
JP\_ID: 067X0001  
#Matching Points: 40



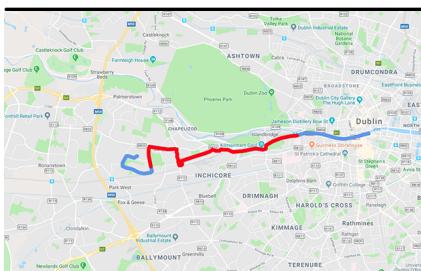
Test Trip 4  
Δt = 684.084324121



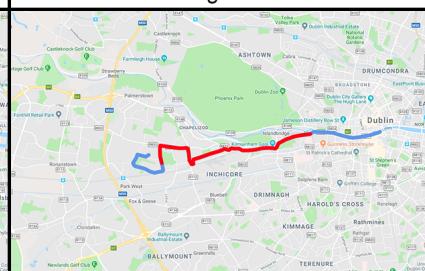
Neighbor1  
JP\_ID: 790001  
#Matching Points: 59



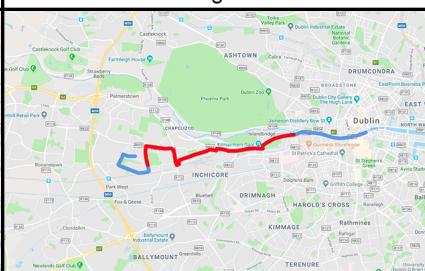
Neighbor2  
JP\_ID: 790001  
#Matching Points: 59



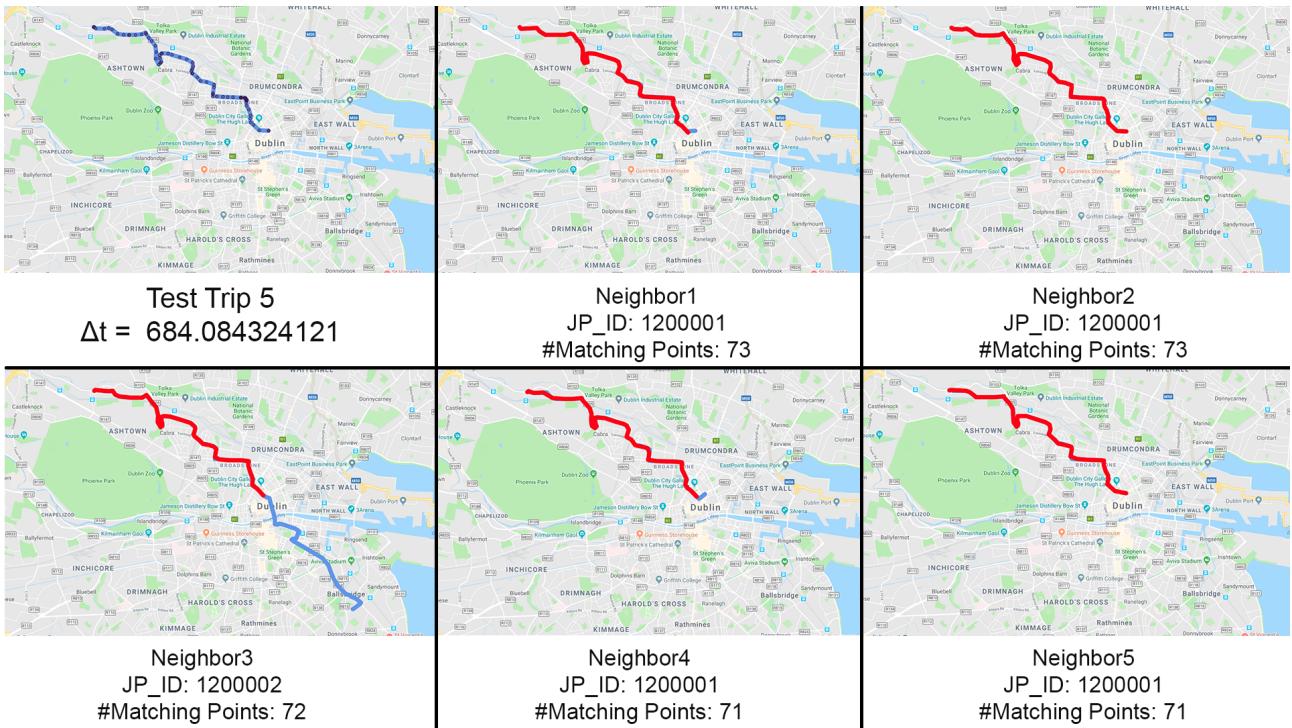
Neighbor3  
JP\_ID: 790001  
#Matching Points: 59



Neighbor4  
JP\_ID: 790001  
#Matching Points: 59



Neighbor5  
JP\_ID: 790001  
#Matching Points: 59



### 3. Κατηγοριοποίηση

Όσον αφορά τον KNN αλγόριθμο έχουμε χρησιμοποιήσει δική μας υλοποίηση παρόμοια με αυτή της Εργασίας 1 με μόνες διαφορές το πλήθος των γειτόνων που στη συγκεκριμένη περίπτωση είναι 5 και τη μετρική υπολογισμού της απόστασης η οποία είναι η DTW. Για κάθε στοιχείο του test\_set υπολογίζονται οι DTW αποστάσεις από όλα τα στοιχεία του train\_set οι οποίες και ταξινομούνται κατά αύξουσα σειρά και τελικά επιστρέφονται οι 5 μικρότερες (συνάρτηση calculate\_distances()). Η κατηγορία, δηλαδή το JourneyPatternId, στην οποία αντιστοιχεί η πλειοψηφία από τους 5 αυτούς υπολογισμούς αποστάσεων που έχουμε κάνει είναι και το τελικό αποτέλεσμα της ταξινόμησης για κάθε στοιχείο του test\_set. Η διαδικασία επαναλαμβάνεται για όλα τα στοιχεία και η ακρίβεια της κατηγοριοποίησης αξιολογείται μέσω της KNN\_CrossValidation(folds, data) συνάρτησης. Όπως προτάθηκε και από τους βοηθούς κατά την παρουσίαση της εργασίας, έχουμε τρέξει την cross validation με αρκετά μικρό dataset και συγκεκριμένα 500 data, διότι με μεγαλύτερο αριθμό δεδομένων η διαδικασία ήταν πολύ χρονοβόρα. Η ακρίβεια που πετύχαμε με αυτό το dataset και με 10 folds cross validation ήταν **89%**.