

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ



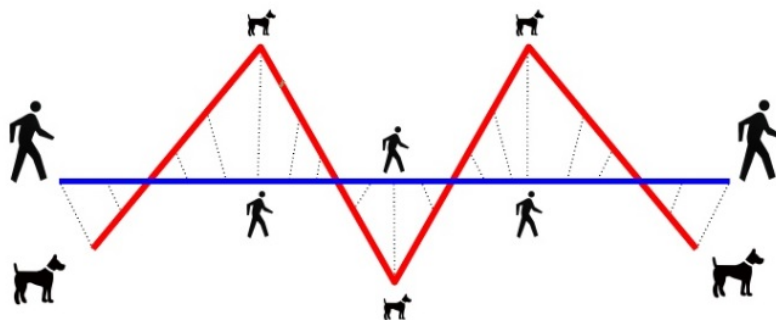
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
— ΙΔΡΥΘΕΝ ΤΟ 1837 —



ΥΠΟΛΟΓΙΣΤΙΚΗ ΓΕΩΜΕΤΡΙΑ

ΑΝΑΖΗΤΗΣΕΙΣ ΚΟΝΤΙΝΩΝ ΓΕΙΤΟΝΩΝ ΒΑΣΗ ΤΗΣ ΑΠΟΣΤΑΣΗΣ FRÉCHET

ΧΡΙΣΤΟΦΟΡΟΣ ΖΑΧΑΡΗΣ



ΑΚΑΔΗΜΑΪΚΟ ΈΤΟΣ 2024-2025

Περιεχόμενα

Εισαγωγή	7
1 Η απόσταση Fréchet	9
1.1 Βασικές έννοιες	9
1.2 Το διάγραμμα ελεύθερου χώρου	11
1.3 Ο αλγόριθμος υπολογισμού της απόστασης σε σχέση με το ε	17
1.4 Ο αλγόριθμος σε κώδικα python	21
1.4.1 Ο υπολογισμός του ελεύθερου χώρου στα σύνορα	21
1.4.2 Η απόφαση για το αν υπάρχει έγκυρο μονοπάτι	27
1.4.3 Ο υπολογισμός των κρίσιμων ε	34
1.4.4 Δυναδική αναζήτηση και εύρεση της απόστασης Fréchet	39
2 Αναζητήσεις γειτονικών καμπύλων	43
2.1 Τοποθέτηση του προβλήματος	43
2.2 Τα δεδομένα: καμπύλες από διαδρομές GPS	43
2.3 Επεξεργασία των δεδομένων	45
2.3.1 Κάτω φράγμα για την απόσταση Fréchet	45

2.3.2	Δοκιμή του κάτω φράγματος	55
2.3.3	Θετικό φίλτρο για την απόσταση Fréchet	59
2.3.4	Δοκιμή του θετικού φίλτρου	62
2.3.5	Αρνητικό φίλτρο για την απόσταση Fréchet	65
2.3.6	Δοκιμή του αρνητικού φίλτρου	69
2.3.7	Σύγκριση θετικού και αρνητικού φίλτρου	72
2.4	Αναδρομικός αλγόριθμος για την απόσταση Fréchet	73
2.4.1	Ο αλγόριθμος και ο κώδικας	73
2.4.2	Η έξοδος του κώδικα	83
3	Χρόνος εκτέλεσης των αλγορίθμων	89
3.1	Σύγκριση των δυο αλγορίθμων για την απόσταση Fréchet	89
3.2	Συμπεριφορά για διαφορετικό μέγεθος καμπυλών	91
3.3	Συμπεριφορά για διαφορετικό μέγεθος βάσης δεδομένων	95
3.4	Σύγκριση θετικού και αρνητικού φίλτρου	97
3.5	Τελική δοκιμασία: κλασσική μέθοδος vs κάτω φράγμα + φίλτρα + επαναληπτικός αλγόριθμος	99
3.6	Δοκιμή με πραγματικές καμπύλες	103
4	Βιβλιογραφία	107

Εισαγωγή

SOME INTRO

Κεφάλαιο 1

Η απόσταση Fréchet

1.1 Βασικές έννοιες

Γενικά, η απόσταση Fréchet αποτελεί ένα μέτρο για το πόσο παρόμοιες είναι δυο καμπύλες. Για την περιπτωσή μας, θα ασχοληθούμε με πολυγωνικές καμπύλες, που στην ουσία είναι ο τύπος των καμπυλών που συναντάμε σε μια διαδρομή GPS. Μια πολυγωνική καμπύλη P αποτελείται από συνδεδεμένα ευθύγραμμα τμήματα και χαρακτηρίζεται από τις κορυφές της $p_1, \dots, p_n$ με $p_i \in \mathbb{R}^2$. Οι κορυφές αυτές είναι προφανώς τα σημεία τομής των ευθύγραμμων τμημάτων, ή αλλιώς είναι τα άκρα κάθε τμήματος. Το συνολικό μήκος της ευθείας είναι $\sum_{i=1}^{n-1} \|p_i - p_{i+1}\|$.

Κάθε τέτοια καμπύλη μπορεί να παραμετροποιηθεί γραμμικά μέσω μιας συνάρτησης με μεταβλητή t έστω $\varphi(t)$. Η συνάρτηση αυτή πρέπει να είναι συνεχής, αφού η καμπύλη μας είναι συνεχόμενη και στην ουσία κάθε τιμή της συνάρτησης μας πηγαίνει σε κάποιο σημείο της καμπύλης. Αυτές τις συναρτήσεις μπορούμε να τις κανονικοποιήσουμε ώστε να παίρνουν τιμές στο $[0, 1]$ με $\varphi(0) = p_1$ και $\varphi(1) = p_n$ και άρα να έχει σύνολο τιμών το $[0, n-1]$.

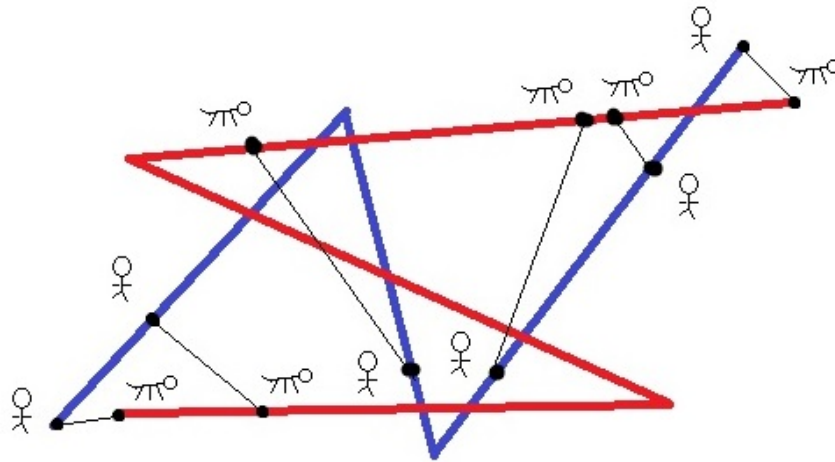
Δίνουμε τώρα ένα επίσημο ορισμό για την απόσταση Fréchet. Έστω δυο καμπύλες P_1, P_2 και Φ_n το σύνολο όλων των συνεχών, θετικά μονότονων συναρτήσεων φ από το $[0, 1] \rightarrow [0, n-1]$. Η απόσταση Fréchet των δυο καμπυλών είναι:

$$d_F(P_1, P) = \min_{\varphi_1, \varphi_2 \in \Phi_n} \max_{t \in [0, 1]} \|P_1(\varphi_1(t)) - P_2(\varphi_2(t))\| \quad (1.1)$$

Ο καλύτερος τρόπος να κατανοήσουμε τον παραπάνω ορισμό είναι με το παράδειγμα του ανθρώπου ο οποίος έχει βγάλει τον σκύλο του βόλτα. Άνθρωπος και σκύλος ακολουθούν από ένα διαφορετικό μονοπάτι και το ερώτημα είναι ποιο το μικρότερο μήκος που πρέπει να έχει το λουρί του σκύλου ώστε να φτάσουν στο τέρμα χωρίς πρόβλημα. Η κίνηση και των δυο μπορεί να γίνει με οποιαδήποτε ταχύτητα, αλλά πρώτον δεν μπορούν να γυρίσουν προς τα πίσω σε οποιαδήποτε στιγμή και δεύτερον υπάρχει η δυνατότητα να σταματήσει κάποιος αν χρειαστεί. Συνεπώς, μπορούμε να επιλέξουμε εμείς με ποιόν ρυθμό και με ποια στρατηγική θα κινηθούν πάνω στις καμπύλες τους, έτσι ώστε να πάρουμε το ελάχιστο μήκος λουριού.

Συνοψίζοντας, παραμετροποιούμε τις καμπύλες για τον άνθρωπο και τον σκύλο με κάποια εξίσωση $\varphi(t)$ η οποία θα μας δώσει τον τρόπο με τον οποίο θα κινηθούν πάνω σε αυτές. Υπάρχει άπειρες δυνατές παραμετροποιήσεις, ανάλογα με τον τρόπο κίνησης που επιλέγουμε κάθε φορά. Για κάθε μια στρατηγική υπάρχει ένα μέγιστο μήκος που πρέπει να έχει το λουρί του σκύλου ώστε να μην χάσουμε επαφή έως το τέρμα. Από όλα αυτά τα μήκη λουριού, επιλέγουμε τον τρόπο κίνησης στις καμπύλες που το ελαχιστοποιεί. Εξ' ου και ο ορισμός $\min \max$ της απόστασης Fréchet.

Ένα παράδειγμα μιας πιθανής διάχισης των δυο μονοπατιών είναι το παρακάτω:



Σχήμα 1.1: Παράδειγμα μιας πιθανής διάχισης των δυο καμπύλων

Βλέπουμε τον άνθρωπο και τον σκύλο σε διαφορετικές χρονικές στιγμές στις θέσεις που βρίσκονται πάνω στις καμπύλες. Το μέγιστο μήκος για το λουρί είναι προφανώς όσο το μήκος την στιγμή που οι δυο τους βρίσκονται στην μέγιστη δυνατή απόσταση, ανάλογα με τον ρυθμό που διατρέχουν τις καμπύλες. Το ελάχιστο μήκος του λουριού φράσσεται από την απόσταση των αρχικών και των τελικών σημείων της κάθε καμπύλης. Προφανώς, και οι δυο ξεκινούν από τα αρχικά σημεία

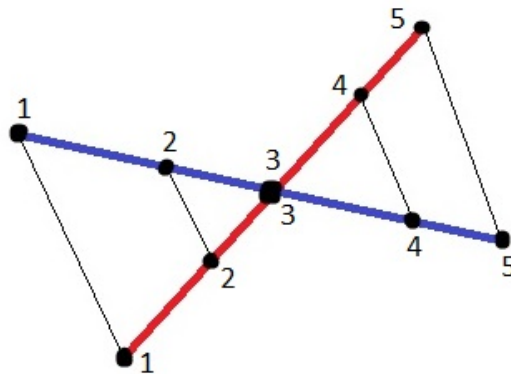
της καμπύλης τους και καταλήγουν στα τελικά σημεία και άρα, το λουρί πρέπει να φτάνει ώστε να καλύψει αυτές τις περιπτώσεις.

Η απόσταση Fréchet και οι παραλλαγές της βρίσκουν εφαρμογή σε διάφορα προβλήματα, από το morphing και την αναγνώριση χειρόγραφου κειμένου έως την ευθυγράμμιση δομών πρωτεϊνών. Οι Alt και Godau ήταν οι πρώτοι που περιέγραψαν έναν πολυωνυμικό αλγόριθμο για τον υπολογισμό της απόστασης Fréchet μεταξύ δύο πολυγωνικών καμπυλών σε Ευκλείδιο χώρο, βασισμένο στην αρχή της παραμετρικής αναζήτησης. Ο χρόνος εκτέλεσης του αλγορίθμου τους είναι $O(mn \log(mn))$ για δύο πολυγωνικές καμπύλες με m και n τμήματα.

1.2 Το διάγραμμα ελεύθερου χώρου

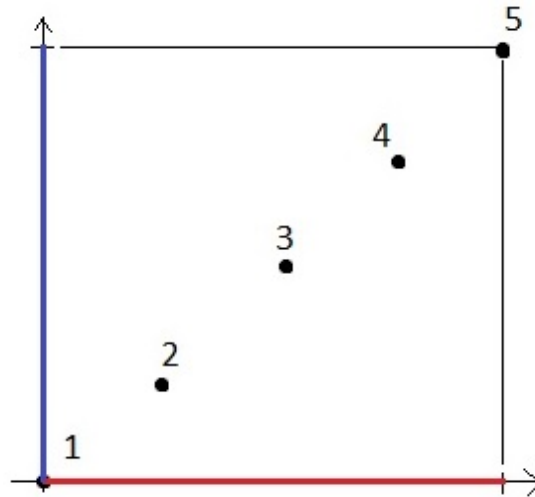
Ένα σημαντικό εργαλείο για την εύρεση της απόστασης Fréchet είναι το διάγραμμα ελεύθερου χώρου. Είδαμε, παραπάνω πως ψάχνουμε το ελάχιστο δυο πραγματικών παραμετρικών συναρτήσεων για τις καμπύλες κάτι το οποίο δεν είναι εύκολη υπόθεση. Ωστόσο, πολλές φορές είναι πιο εύκολο να αποφασίσουμε αν το ελάχιστο αυτό φράσσεται από κάποιο τιμή την οποία γνωρίζουμε. Αυτό θα κάνουμε και εδώ. Θεωρούμε μια τιμή ε για την οποία μπορούμε να απαντήσουμε αν η απόσταση Fréchet δυο καμπυλών είναι το πολύ όσο το ε .

Αυτό μπορεί να γίνει μέσω του διαγράμματος ελεύθερου χώρου. Προκειται στην ουσία για μια απεικόνιση όλων πιθανών συνδυασμών για το που βρισκόμαστε σε κάθε καμπύλη σε ένα καρτεσιανό διάγραμμα. Ας θεωρήσουμε δυο ευθύγραμμα τμήματα e_1, e_2 με παραμετρικές εξισώσεις $\varphi_1(t), \varphi_2(t)$, όπως στο παρακάτω σχήμα:



Σχήμα 1.2: Δυο ευθύγραμμα τμήματα με σημειωμένα κάποια τυχαία ζεύγη σημείων

Το διάγραμμα ελεύθερου χώρου είναι ένας μοναδιαίος δισδιάστατος χώρος, με το κάθε του σημείο να αντιστοιχεί σε μια δυάδα σημείων πάνω στις καμπύλες. Έτσι, σε κάθε σημείο αντιστοιχεί και μια Ευκλείδεια απόσταση, όση και η απόσταση των δυο σημείων. Έτσι, για μια επιλογή φράγματος απόστασης ε , μπορούμε να επιλέξουμε τον γεωμετρικό τόπο εκείνων των σημείων του διαγράμματος ελεύθερου χώρου τα οποία ικανοποιούν την ανισότητα με το ε . Ας δούμε το διάγραμμα χώρου για το παραπάνω παράδειγμα τοποθετώντας κατάλληλα τα σημεία:

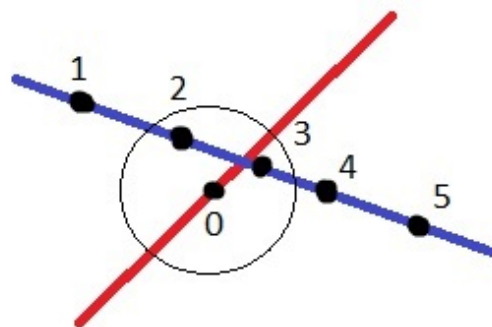


Σχήμα 1.3: Τα αντίστοιχα σημεία στο διάγραμμα ελεύθερου χώρου

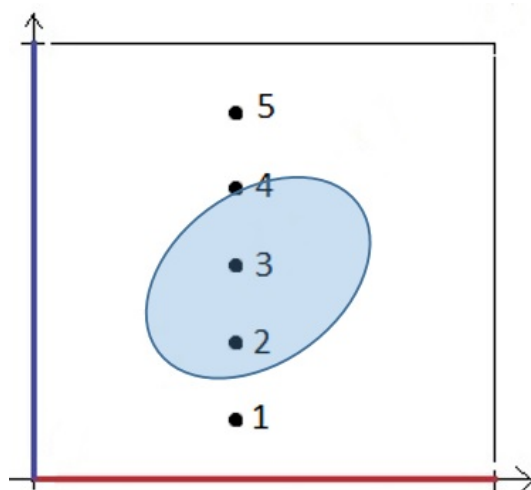
Βλέπουμε τα 5 σημεία στο διάγραμμα χώρου. Κάτω αριστερά έχουμε τις αρχικές θέσεις για τις δυο καμπύλες, δηλαδή άνθρωπος και σκύλος βρίσκονται στο αρχικό τους σημείο. Ομοίως, πάνω δεξιά έχουμε την τελική θέση όπου και οι δυο βρίσκονται στα τελικά σημεία. Οι άξονες είναι από το 0 ως το 1 και άρα μπορούμε επιπλέον να τους σκεφτούμε σαν να δίνουν το ποσοστό της απόστασης που έχουμε διανύσει πάνω σε κάθε καμπύλη. Για παράδειγμα στο σημείο 4 έχουμε διανύσει σχεδόν το 75% του μήκους κάθε καμπύλης.

Ας δώσουμε τώρα μια τιμή στο ε , η οποία είπαμε πως δίνει το μέγιστο όριο που θέλουμε να έχει το λουρί μας. Επιλέγουμε μια αυθαίρετη τιμή για το πόσο μακρύ είναι το λουρί που κρατά ο άνθρωπος και την σχεδιάζουμε (ο άνθρωπος βρίσκεται τώρα στην κόκκινη ευθεία):

Όσο ο άνθρωπος κινείται προς τα εμπρός στην καμπύλη του, μπορούμε να φανταστούμε τον κύκλο να τον ακολουθεί. Τα σημεία της μπλε καμπύλης του σκύλου στα οποία μπορούμε να κρατήσουμε το λουρί είναι αυτά τα οποία βρίσκονται εντός του κύκλου καθώς αυτός διατρέχει την κόκκινη καμπύλη. Όλα αυτά τα σημεία δημιουργούν ένα γεωμετρικό τόπο στο διάγραμμα χώρου τον οποίο μπορούμε να δούμε στο επόμενο σχήμα:



Σχήμα 1.4: Δίνουμε κάποια τιμή ε ως το μέγιστο μήκος για το λουρί

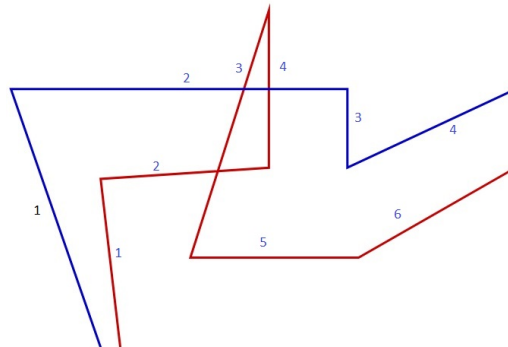


Σχήμα 1.5: Ο γεωμετρικός τόπος των σημείων που ικανοποιούν το μήκος ε

Για όλα τα σημεία εντός της έλλειψης, η αναπαράστασή τους ως διπλά σημεία στις καμπύλες έχει απόσταση μικρότερη του ε . Για όλα τα σημεία εκτός, η απόσταση είναι μεγαλύτερη του ε . Δείτε για παράδειγμα το σημείο 4. Αυτό βρίσκεται εκτός του γεωμετρικού τόπου, αφού η απόσταση του 0 με το 4 είναι μεγαλύτερη του ε . Αν όμως ο άνθρωπος μετακινηθεί πάνω στην κόκκινη καμπύλη λίγο ακόμη, το νέο σημείο 4 θα μετατοπιστεί δεξιά στο διάγραμμα χώρου και άρα θα πέσει μέσα στον γεωμετρικό τόπο. Πράγματι, ο κύκλος θα συμπεριλάβει το 4 όταν μετακινηθεί λίγο πιο πάνω στην καμπύλη. Αντίθετα, το σημείο 5 δεν θα πέσει ποτέ εντός του κύκλου, όπου και να βρίσκεται αυτός πάνω στην κόκκινη καμπύλη. Γενικά, για ευθύγραμμα τμήματα, P_1, P_2 ο ελεύθερος χώρος F_ε είναι η τομή του μοναδιαίου τετραγώνου με μια έλλειψη. Να σημειωθεί πως τα παραπάνω είναι συμμετρικά με την περίπτωση το λουρί να κρατάει ο σκύλος.

Συμπεραίνουμε λοιπόν πως η απόσταση Fréchet είναι σίγουρα μεγαλύτερη από ε στο παραπάνω σχήμα. Γιατί μεγαλύτερη; Γιατί δεν υπάρχει τρόπος να ξεκινήσουμε από την αρχή της κάθε καμπύλης και να φτάσουμε στο τέλος της, αλλά παράλληλα να κρατάμε το λουρί καθ' όλη την διάρκεια της διαδρομής. Αποτυγχάνουμε μάλιστα από την αρχή ακόμα αφού για λουρί ε , δεν μπορούμε να δέσουμε τον σκύλο όταν βρεθούμε και οι δυο στα αρχικά μας σημεία. Στο διάγραμμα ελεύθερου χώρου, αυτό μεταφράζεται στο ότι ο γεωμετρικός τόπος δεν περιλαμβάνει μονοπάτι από το κάτω αριστερά σημείο (αρχή των καμπυλών) προς το πάνω δεξιά (τέλος των καμπυλών). Για να μπορεί να ικανοποιηθεί η συνθήκη του να κρατάμε το λουρί σε όλη την διαδρομή, το μονοπάτι αυτό πρέπει να υπάρχει. Άρα, μέσω του διαγράμματος ελεύθερου χώρου, μπορούμε τελικά να βρούμε το βέλτιστο ε για το οποίο υπάρχει αυτό το μονοπάτι.

Η λογική για τα ευθύγραμμα τμήματα, μπορεί να επεκταθεί και σε πολυγωνικές καμπύλες. Αυτές αποτελούνται από ένα συνδυασμό ευθύγραμμων τμημάτων όπως είδαμε και στην πρώτη ενότητα. Κατά αυτόν τον τρόπο, στο διάγραμμα ελεύθερου χώρου θα έχουμε πολλά κελιά χώρου, ένα για κάθε συνδυασμό ευθύγραμμων τμημάτων, ο οποίος θα έχει τις ίδιες ιδιότητες με τον χώρο ευθύγραμμου τμήματος προς ευθύγραμμο τμήμα. Έτσι, σε κάθε κελί ο ελεύθερος χώρος θα έχει μορφή έλλειψης και θα είναι κυρτός (κάθε σημείο μπορεί να ενωθεί με κάθε άλλο σημείο μέσω μιας ευθείας). Ένα παράδειγμα θα μας πείσει. Ας δούμε τις παρακάτω καμπύλες:



Σχήμα 1.6: Δυο πολύγωνες καμπύλες. Έχουμε αριθμήσει τα ευθύγραμμα τμήματα

Η μπλε καμπύλη αποτελείται από 4 ευθύγραμμα τμήματα, ενώ η κόκκινη από 6 ευθύγραμμα τμήματα. Συνεπώς, στο διάγραμμα ελεύθερου χώρου, θα έχουμε 24 κελιά όπου σε κάθε κελί θα γίνεται σύγκριση δυο ευθύγραμμων τμημάτων σύμφωνα με τα όσα είδαμε στην απλή περίπτωση. Ας σχεδιάσουμε το διαγραμμα:

Στον κατακόρυφο άξονα έχουμε την μπλε καμπύλη και στον οριζόντιο την κόκκινη. Έχουμε χωρίσει τους δυο άξονες, ανάλογα με το ποσοστό μήκους κάθε ευθύγραμμου τμήματος σε σχέση με το συνολικό μήκος της καμπύλης. Στη συνέχεια,

4,1	4,2	4,3	4,4	4,5	4,6
3,1	3,2	3,3	3,4	3,5	3,6
2,1	2,2	2,3	2,4	2,5	2,6
1,1	1,2	1,3	1,4	1,5	1,6

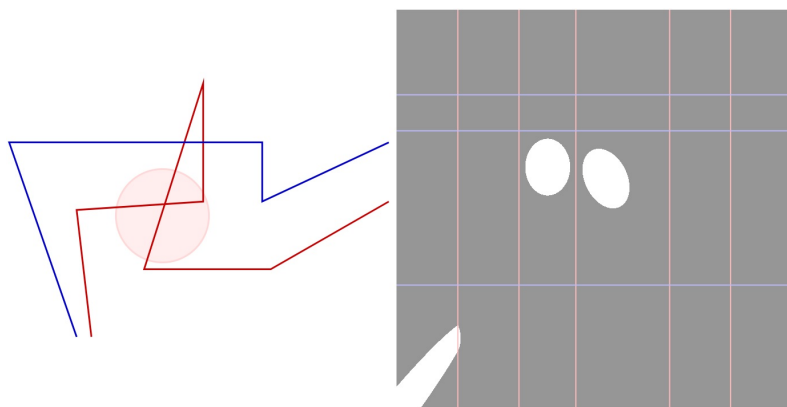
Σχήμα 1.7: Το διάγραμμα ελεύθερου χώρου για τις παραπάνω καμπύλες. Κάθε κελί αντιστοιχεί σε δυο ευθύγραμμα τμήματα των αρχικών καμπύλων

έχουμε σχεδιάσει τα κελιά που αντιστοιχούν στα ευθύγραμμα τμήματα που φαίνονται με τους αριθμούς. Συνεπώς, στο κελί 2,4 για παράδειγμα, έχουμε το διάγραμμα ελεύθερου χώρου για το 2 ευθύγραμμο τμήμα της μπλε καμπύλης και για το 4 ευθύγραμμο τμήμα της κόκκινης.

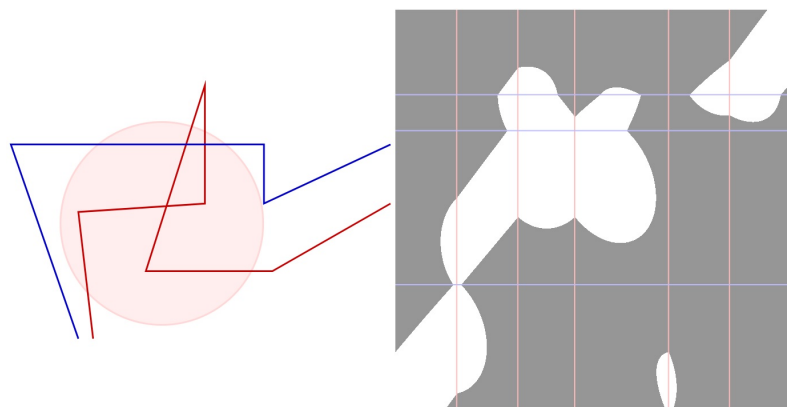
Στον ελεύθερο χώρο που δημιουργείται σε κάθε κελί για κάποιο ε μπορώ να κινηθώ μόνο πάνω και δεξιά. Αυτό προκύπτει από το γεγονός πως δεν μπορώ σε καμία περίπτωση να γυρίσω πίσω σε κάποια καμπύλη. Αν σε κάποιο κελί μπω από τη μέση του αριστερού συνόρου, μπορώ να βρεθώ μόνο στο πάνω μισό του. Συνεπώς, ακόμη και προκύψει μια πιθανή διαδρομή από κάτω αριστερά προς πάνω δεξιά, αυτή δεν θα είναι έγκυρη αν αναγκάζομαι να κινηθώ προς τα κάτω ή προς τα αριστερά σε οποιοδήποτε σημείο.

Μπορούμε τώρα να θέσουμε διάφορες τιμές του ε και να δούμε πως μεταφράζεται αυτό σε ελεύθερο χώρο στο διάγραμμα. Οι παρακάτω εικόνες θα μας διαφωτίσουν. Στα αριστερά βλέπουμε το μέγεθος του κύκλου, η ακτίνα του οποίου είναι το ε , δηλαδή το μέγιστο μήκος του λουριού, και δεξιά το διάγραμμα ελεύθερου χώρου όπου με άσπρο χρώμα είναι ο ελεύθερος χώρος.

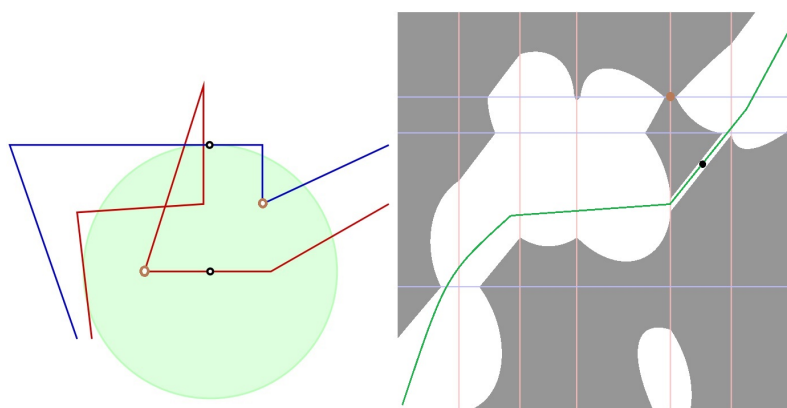
Ενδιαφέρον στο τελευταίο σχήμα παρουσιάζει η διαδρομή που έχει εμφανιστεί. Ενώ κανείς θα περίμενε πως όσο μεγαλώνουμε το ε το καφέ σημείο στο διάγραμμα (που αντιστοιχεί στα δυο καφέ σημεία στις καμπύλες) θα γίνει ελεύθερος χώρος και το έγκυρο μονοπάτι θα περάσει από εκεί, εντούτοις εμφανίστηκε ένα μονοπάτι πιο χαμηλά στο διάγραμμα. Όπως φαίνεται και από το μύαρο σημείο, η καλύτερη



Σχήμα 1.8: Το διάγραμμα ελεύθερου χώρου για κάποια τιμή του ε



Σχήμα 1.9: Το διάγραμμα ελεύθερου χώρου για μια μεγαλύτερη τιμή ε



Σχήμα 1.10: Το διάγραμμα ελεύθερου χώρου για κάποια μεγαλύτερη τιμή του ε για την οποία τώρα υπάρχει μονοπάτι από την αρχή ως το τέλος

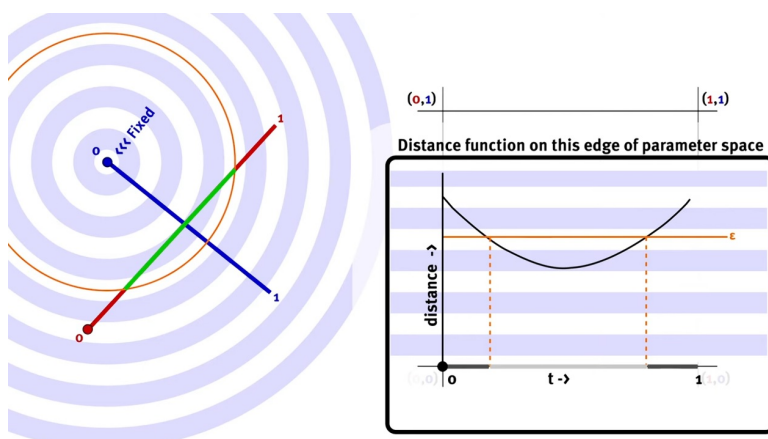
1.3. Ο ΑΛΓΟΡΙΘΜΟΣ ΥΠΟΛΟΓΙΣΜΟΥ ΤΗΣ ΑΠΟΣΤΑΣΗΣ ΣΕ ΣΧΕΣΗ ΜΕ ΤΟ ε 17

στρατηγική ώστε να κρατήσουμε χαμηλά το μήκος του λουριού δεν είναι να το μεγαλώσουμε τόσο ώστε να συνδέονται οι δυο καμπές που έχουμε σημειώσει με καφέ, αλλά να κινηθούμε παράλληλα στις δυο καμπύλες με την ίδια ταχύτητα έως ότου διασχίσουμε αυτά τα απόμακρα σημεία.

Συμπερασματικά λοιπόν, έχουμε απαντήσει στο πρόβλημα απόφασης για το αν η απόσταση Fréchet είναι μεγαλύτερη ή όχι του ε . Αν στο διάγραμμα ελεύθερου χώρου υπάρχει έγκυρο μονοπάτι, τότε η απόσταση Fréchet είναι μικρότερη του ε . Αν δεν υπάρχει έγκυρο μονοπάτι, η απόσταση Fréchet είναι μεγαλύτερη του ε . Το επόμενο τώρα πρόβλημα αποτελεί η βελτιστοποίηση του ε στην τιμή εκείνη για την οποία μόλις που υπάρχει ένα έγκυρο μονοπάτι. Η τιμή αυτή θα είναι και η απόσταση Fréchet.

1.3 Ο αλγόριθμος υπολογισμού της απόστασης σε σχέση με το ε

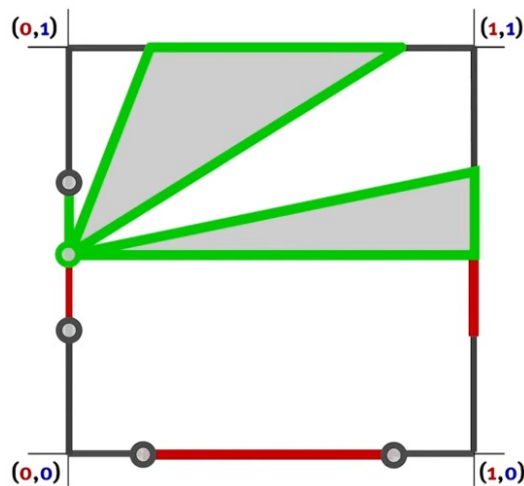
Μεγάλο ενδιαφέρον στο διάγραμμα ελεύθερου χώρου παρουσιάζουν τα σύνορα κάθε κελιού. Καθώς κινούμαστε στο σύνορο κάθε κελιού στην ουσία φιζά-ρουμε το ένα σημείο στο άκρο από το ένα ευθύγραμμο τμήμα, καθώς το άλλο σημείο κινείται στο μήκος του δεύτερου τμήματος. Δείτε για παράδειγμα τα σχήματα 1.4 και 1.5 και φανταστείτε τα σημεία στο σύνορο. Για κάποια τιμή του ε λοιπόν, μας ενδιαφέρει πόσο από το σύνορο βρίσκεται στον ελεύθερο χώρο και πόσο εκτός. Ας δούμε το παρακάτω γράφημα όπου επικεντρωνόμαστε στις δυο καμπύλες που περιγράφουν κάποιο κελί:



Σχήμα 1.11: Το διάγραμμα ελεύθερου χώρου για το σύνορο ενός κελιού. Υπάρχουν 0, 1 ή 2 σημεία ενδιαφέροντος πάνω στο σύνορο, ανάλογα με την τιμή του ε

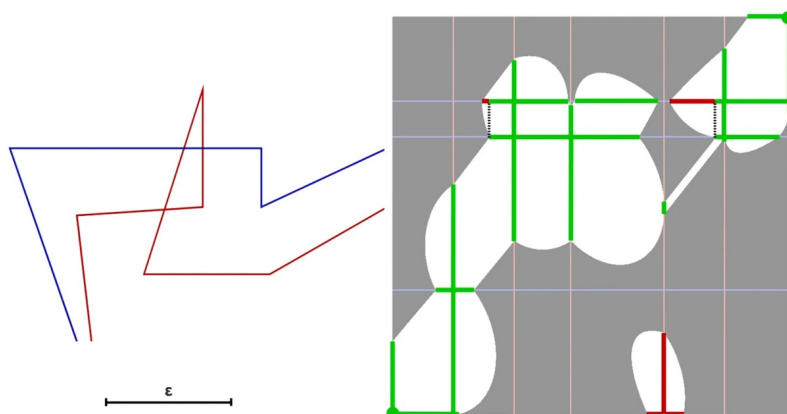
Η συνάρτηση απόστασης για το σύνορο ενός κελιού είναι παραβολή και μάλιστα μη αρνητική και το τμήμα της παραβολής που βρίσκεται κάτω από την τιμή του ε μας δίνει το ποσοστό του συνόρου που αποτελεί ελεύθερο χώρο. Τα σημεία αλλαγής της συμπεριφοράς του συνόρου μπορεί να είναι 0, αν όλη η παραβολή βρίσκεται πάνω από το ε , 1 αν εφάπτεται και 2 αν είναι όπως στο σχήμα. Η γνώση για το πόσο από το σύνορο αποτελεί ελεύθερο χώρο σε συνάρτηση με αυτό που έχουμε τονίσει παραπάνω, πως η περιοχή ελεύθερου χώρου εντός κάθε κελιού είναι κυρτή, μας απλοποιεί το πρόβλημα, αφού πλέον δεν μας ενδιαφέρει ποια περιοχή εντός του κελιού αποτελεί ελεύθερο χώρο. Αν μια περιοχή στο κάτω σύνορο είναι ελεύθερη και μια περιοχή στο πάνω σύνορο είναι ελεύθερη, τότε μπορώ να μεταβώ από το κάτω σύνορο στο πάνω γιατί η συνολική ελεύθερη περιοχή εντός του κελιού πρέπει να είναι κυρτή και άρα υπάρχει ευθύγραμμο τμήμα που συνδέει το κάτω με το πάνω σύνορο.

Σημαντικό είναι επίσης το γεγονός πως μπορώ να μετακινηθώ μόνο πάνω και δεξιά εντός κάθε κελιού λόγω της συνθήκης να μην γυρίζω προς τα πίσω στις καμπύλες. Συνεπώς, από το σημείο εισόδου ας πούμε στο αριστερό σύνορο, μπορώ να μετακινηθώ μόνο προς τα πάνω. Συνεπώς, αν υπάρχει ελεύθερο σύνορο προς τα κάτω, αυτό αυτομάτως απορρίπτεται αφού δεν μπορώ ποτέ να φτάσω σε αυτό. Το επόμενο γράφημα εξηγεί αυτό το γεγονός:



Σχήμα 1.12: Οι αποδεκτές περιοχές του συνόρου είναι αυτές που βρίσκονται πάνω και δεξιά από το σημείο εισόδου στο κελί

Συνεπώς, αν τώρα υπολογίσουμε όλες αυτές τις περιοχές για κάθε κελί του διαγράμματος ελεύθερου χώρου, τότε μπορώ να αποφασίσω την σχέση της απόστασης Fréchet με το ε . Στο παρακάτω παράδειγμα, βλέπουμε πως υπάρχουν διαδοχικές πράσινες περιοχές στα σύνορα των κελίων έτσι ώστε να φτάσουμε από κάτω αριστερά στο πάνω δεξιά σημείο:



Σχήμα 1.13: Ανάλογα με το τι συμβαίνει στα σύνορα των κελίων μπορώ να αποφασίσω τελικά για την απόσταση Fréchet σε σχέση με το ε

Για να αναλύσουμε τον αλγόριθμο, καλό θα ήταν να εισάγουμε έναν συμβολισμό. Ονομάζουμε τις δυο πολύγωνες καμπύλες P και Q και τις ακμές αυτών, δηλαδή τα ευθύγραμμα τμήματα, ως p_i και q_j με τα i, j από το 0 έως το πλήθος των ακμών πλην 1. Επιπλέον, κάθε p_i είναι τώρα η ευθεία μεταξύ των σημείων $P(i)$ και $P(i+1)$. Θυμηθείτε πως τα σημεία των πολύγωνων καμπύλων είναι γνωστά. Κάθε κελί λοιπόν είναι ένα ορθογώνιο της μορφής $[i, i+1] \times [j, j+1]$.

Τα σύνορα για κάθε κελί δίνονται τώρα ως εξής: για το αριστερό και το δεξί σύνορο φιξάρουμε το x στις τιμές i ή $i+1$ όσο το y μεταβάλλεται μεταξύ των $[j, j+1]$. Ομοίως για τα κάτω και πάνω σύνορα, φιξάρουμε το y στις τιμές j ή $j+1$ όσο το x μεταβάλλεται μεταξύ i και $i+1$. Για να υπολογίσουμε τον ελεύθερο χώρο στο σύνορο, είδαμε πως βρίσκουμε την τομή της τετραγωνικής συνάρτησης που χαρακτηρίζει την απόσταση και της ευθείας ε . Για παράδειγμα για το κάτω σύνορο, πρέπει να λύσουμε την ανισότητα:

$$\|P(x) - Q(j)\| \leq \varepsilon \quad (1.2)$$

Έτσι, ο αλγόριθμος για τον υπολογισμό των ελεύθερων συνόρων είναι ο εξής:

1. Σαρώνουμε όλα τα κελιά (i, j) , όπου κάθε κελί αντιστοιχεί σε ένα ζεύγος ακμών: μια από την καμπύλη P και μια από την καμπύλη Q .
2. Για κάθε σύνορο (αριστερά, δεξιά, κάτω, πάνω) του κελιού:
 - Σταθεροποιούμε την μια συντεταγμένη (είτε την x είτε την y), που αντιστοιχεί στο κάθε σύνορο.

- Λύνουμε την τεταγωνική ανισότητα

$$\|P(x) - Q(y)\| \leq \varepsilon$$

όπου $P(x)$ and $Q(y)$ τα παραμετροποιημένα σημεία σε κάθε ακμή.

- Αποθηκεύουμε τα αποτελέσματα ως τον ελεύθερο χώρο του συνόρου.

Από την στιγμή που έχουμε τον ελεύθερο χώρο στα σύνορα, είναι τώρα εύκολο να αποφασίσουμε για την απόσταση Fréchet σε σχέση με το ε .

1. Ξεκινάμε από το κελί κάτω αριστερά που περιλαμβάνει το σημείο εκκίνησης. Αρχικά ελέγχουμε αν το σημείο αυτό ανήκει στον ελεύθερο χώρο. Έπειτα, ελέγχουμε τον ελεύθερο χώρο στα σύνορα του κελιού. Αν υπάρχει ελεύθερος χώρος στο δεξί ή στο αριστερό σύνορο, τότε λόγω της κυρτότητας μπορούμε να φτάσουμε σε αυτό από το αρχικό σημείο.
2. Για κάθε επόμενο κελί, συγκρίνουμε τα σύνορα του με το προηγούμενο. Αν υπάρχει κοινός ελεύθερος χώρος τότε μπορούμε να εισέλθουμε σε αυτό.
3. Ελέγχουμε ξανά τον ελεύθερο χώρο στα σύνορα του νέου κελιού. Αν μπορούμε να μεταβούμε από αριστερά ή κάτω προς τα πάνω και δεξιά τότε συνεχίζουμε με το βήμα 2. Προσοχή, μπορούμε να κινηθούμε μόνο δεξιά και πάνω σε κάθε κελί οπότε κρατάμε τα ελεύθερα σύνορα που βρίσκονται πάνω και δεξιά από το ελάχιστο σημείο εισόδου.
4. Αν το πάνω δεξιά κελί περιλαμβάνεται σε αυτά που μπορούμε να εισέλθουμε και το τελικό σημείο ανήκει στον ελεύθερο χώρο τότε αποφασίζουμε πως $d_F(P, Q) \leq \varepsilon$ και απαντάμε ναι στην ερώτηση για το αν η απόσταση Fréchet είναι το πολύ ε . Αλλιώς η απάντηση είναι όχι.

Τέλος, αρκεί να βελτιστοποιήσουμε το ε έτσι ώστε μόλις που να υπάρχει μονοπάτι. Θα δούμε πως θα γίνει αυτό στην επόμενη ενότητα μέσω κάποιων κρίσιμων τιμών του ε , εκεί όπου αλλάζει συμπεριφορά το διάγραμμα ελεύθερου χώρου. Όλοι οι παραπάνω αλγόριθμοι προέρχονται από την εργασία των Helmut Alt και Michael Godau στην εργασία Computing the Fréchet distance between two polygonal curves.

1.4 Ο αλγόριθμος σε κώδικα python

1.4.1 Ο υπολογισμός του ελεύθερου χώρου στα σύνορα

Ο στόχος μας εδώ είναι να γράψουμε τον αλγόριθμο απόφασης που είδαμε παραπάνω σε γλώσσα python. Επιλέγουμε python λόγω των βιβλιοθηκών που περιλαμβάνονται σε αυτή και κάνουν τη ζωή μας πιο εύκολη σε θέματα όπως την δημιουργία καμπύλων και γραφημάτων. Προφανώς μιλάω για τα `numpy` και `matplotlib`. Θα αρχίσουμε με την πιο απλή περίπτωση, δυο ευθύγραμμα τμήματα. Όπως είδαμε στην θεωρία, κάθε πολυγωνική καμπύλη χαρακτηρίζεται από τα σημεία της που είναι τα σημεία τομής των διαφόρων ευθύγραμμων τμημάτων από τα οποία αποτελείται. Δημιουργούμε λοιπόν, δυο ευθύγραμμα τμήματα ορίζοντας τα άκρα τους και επιπλέον ορίζουμε μια τιμή για το ϵ .

```
1 # Input straight lines
2 P = [(0, 0), (1, 0)]
3 Q = [(0.5, 0), (0.5, 1)]
4 p_seg = (P[0], P[1])
5 q_seg = (Q[0], Q[1])
6 epsilon = 0.6
```

Listing 1.1: Create two lines as our curves

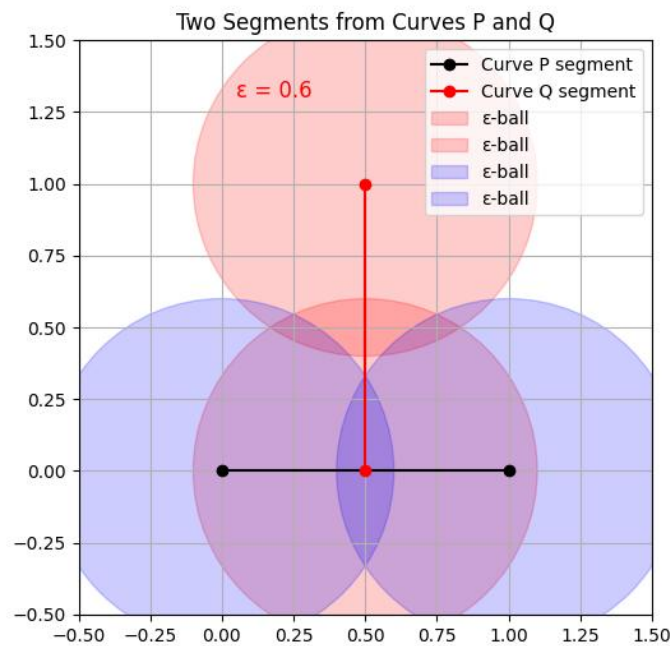
Ας φτιάξουμε και το γράφημα που δείχνει την τοπολογία τους:

```
1 ### FIRST PLOT: The two segments ###
2 fig1, ax1 = plt.subplots(figsize=(6, 6))
3 ax1.plot(*zip(*P), 'k-o', label='Curve P segment')
4 ax1.plot(*zip(*Q), 'r-o', label='Curve Q segment')
5
6 # -balls around points in Q
7 for qpt in Q:
8     ax1.add_patch(plt.Circle(qpt, epsilon, color='red',
9                             alpha=0.2, label='-ball'))
10 for qpt in P:
11     ax1.add_patch(plt.Circle(qpt, epsilon, color='blue',
12                             alpha=0.2, label='-ball'))
13
14 # Annotate epsilon
15 ax1.text(0.05, 1.3, f" = {epsilon}", fontsize=12, color='red')
16 ax1.set_aspect('equal')
17 ax1.set_title("Two Segments from Curves P and Q")
18 ax1.grid(True)
19 ax1.legend(loc='upper right')
20 ax1.set_xlim(-0.5, 1.5)
21 ax1.set_ylim(-0.5, 1.5)
```

Listing 1.2: Create the graph of the 2 line segments

Όπως γίνεται εύκολα αντιληπτό, χρησιμοποιείτε το πακέτο matplotlib για την σχεδίαση των γραφημάτων. Με τον ίδιο τρόπο ζωγραφίζουμε και τα γραφήματα στις υπόλοιπες περιπτώσεις, μέσω εντολών του matplotlib. Επειδή ο σκοπός εδώ δεν είναι η εκμάθηση του πακέτου, παρακάτω δεν θα γίνεται αναφορά στον κώδικα που δημιουργεί τις γραφικές παραστάσεις.

Σε κάθε περίπτωση το αποτέλεσμα του κώδικα φαίνεται παρακάτω. Ας δούμε τι βλέπουμε:



Σχήμα 1.14: Τα δυο ευθύγραμμα τμήματα τα οποία θα χρησιμοποιήσουμε στη συνέχεια. Φαίνονται οι κύκλοι ακτίνας ε γύρω από κάθε άκρο των ευθύγραμμων τμημάτων.

Έχουμε επιλέξει μια πολύ απλή τοπολογία όπου η μια ευθεία είναι στην ουσία η μεσοκάθετος της άλλης. Για την απλή αυτή τοπολογία ξέρουμε τι να περιμένουμε και άρα μπορούμε εύκολα να ελέγξουμε τα αποτελέσματα του κώδικα. Σε κάθε περίπτωση, θα μπορούσαν τα παραπάνω ευθύγραμμα τμήματα να αποτελούν μέρη δυο πολύγωνων καμπύλων.

Έχουμε επιπλέον σχεδιάσει τους κύκλους ακτίνας ε γύρω από κάθε άκρο των ευθύγραμμων τμημάτων. Είπαμε πως στο γράφημα ελεύθερου χώρου, η κίνηση πάνω στα σύνορα κάθε κελιού είναι ισοδύναμη με το να φιζάρουμε το ένα σημείο

σε κάποιο άκρο της μιας καμπύλης καθώς το άλλο σημείο κινείται ελεύθερα στην άλλη καμπύλη. Ανάλογα με το ε λοιπόν, μπορούμε να υπολογίσουμε τον ελεύθερο χώρο πάνω σε κάθε σύνορο μέσω της τομής του κύκλου αυτού με την άλλη ευθεία. Με κόκκινο είναι οι κύκλοι για την κόκκινη ευθεία, ενώ με μπλε οι κύκλοι για την μπλε ευθεία.

Τι περιμένουμε για τον ελεύθερο χώρο των ευθύγραμμων τμημάτων του παραδείγματος. Ας σκεφτούμε πως στον x άξονα έχουμε την μπλε καμπύλη και στον y άξονα την κόκκινη. Για το κάτω σύνορο του κελιού του ελεύθερου χώρου, το ένα σημείο βρίσκεται πάνω στο κάτω άκρο της κόκκινης καμπύλης καθώς το άλλο σημείο κινείται πάνω στη μπλε καμπύλη. Όπως φαίνεται στο σχήμα, ο κόκκινος κύκλος περιλαμβάνει το σύνολο της μπλε καμπύλης και άρα το κάτω σύνορο ανήκει εξ'ολοκλήρου στον ελεύθερο χώρο. Με την ίδια λογική παρατηρούμε τα σημεία τομής των κύκλων με τις καμπύλες.

Για τον κώδικα τώρα, πρέπει όπως είδαμε να λύσουμε μια ανισότητα η οποία θα μας δώσει τον ελεύθερο χώρο για το κάθε σύνορο. Αρχικά πρέπει να έχουμε κάποια παραμετροποίηση για τα δυο ευθύγραμμα τμήματα. Η πιο απλή παραμετροποίηση είναι η παρακάτω:

$$P(t) = p_0 + t(p_1 - p_0) \quad (1.3)$$

με $t \in [0, 1]$. Για τιμή $t = 0$ έχουμε το ένα άκρο ενώ για $t = 1$ το άλλο άκρο της ευθείας. Η παραπάνω σχέση εκφράζει στην ουσία την διάσχιση της ευθείας με σταθερή ταχύτητα. Θα μπορούσαμε να είχαμε επιλέξει t^2 όπου τότε η ταχύτητα θα ήταν αργή στην αρχή και πιο γρήγορη στη συνέχεια ή οτιδήποτε άλλο. Η επιλογή της παραμετροποίησης δεν επηρεάζει τον ελεύθερο χώρο.

Επιπλέον έχουμε και ένα σταθερό σημείο, το ένα άκρο κάποιας εκ των καμπυλών και ψάχνουμε να λύσουμε την ανισότητα (1.2). Αν ονομάσουμε q το σταθερό σημείο η ανισότητα γράφεται:

$$\|P(t) - q\| \leq \varepsilon$$

την οποία τώρα μπορούμε να αναπτύξουμε ως εξής:

$$\begin{aligned}
\|P(t) - q\| &\leq \varepsilon \\
\|p_0 + td - q\| &\leq \varepsilon \\
\|p_0 + td - q\|^2 &\leq \varepsilon^2 \\
(p_0 + td - q) \cdot (p_0 + td - q) &\leq \varepsilon^2 \\
(p_0 - q) \cdot (p_0 - q) + 2td \cdot (p_0 - q) + t^2 d \cdot d &\leq \varepsilon^2
\end{aligned}$$

όπου τώρα έχουμε μια τετραγωνική ανισότητα της μορφής:

$$at^2 + bt + c \leq 0 \quad (1.4)$$

με τους εξής συντελεστές:

- $a = d^2$
- $b = 2d(p_0 - q)$
- $c = (p_0 - q)(p_0 - q) - \varepsilon^2$

Αυτές οι ανισότητες λύνονται πρώτα ως ισότητες, δηλαδή βρίσκουμε την διακρίνουσα $\Delta = b^2 - 4ac$ και ελέγχουμε την τιμή της:

- Για $\Delta < 0$ δεν υπάρχουν πραγματικές ρίζες και άρα δεν υπάρχουν σημεία τομής
- Για $\Delta = 0$ υπάρχει μια διπλή ρίζα και ένα σημείο τομής
- Για $\Delta > 0$ υπάρχουν δυο ρίζες και άρα ένα διάστημα τομής. Αυτές οι δυο ρίζες δίνουν και τα σημεία ενδιαφέροντος στα οποία αλλάζει η συμπεριφορά του ελεύθερου χώρου πάνω στο σύνορο.

Ωραία, ας γράψουμε τώρα μια συνάρτηση να υπολογίζει τα παραπάνω. Δίνουμε 3 παραμέτρους, το ευθύγραμμο τμήμα (η ελεύθερη καμπύλη), το σταθερό σημείο (το άκρο της άλλης καμπύλης) και την τιμή του ε . Σε όλα τα παρακάτω είναι απαραίτητη η χρήση της βιβλιοθήκης `numpy` καθώς έχουμε να κάνουμε με πράξεις στον δισδιάστατο χώρο μεταξύ διανυσμάτων. Ας δούμε το πρώτο τμήμα της συνάρτησης:

```

1 def interval_within_epsilon(seg, point, epsilon):
2     p0, p1 = seg # The line segment
3     q = np.array(point) # The fixed point
4     d = np.array(p1) - np.array(p0) # The vector p1 - p0
5     a = np.dot(d, d) # Coefficient a
6     b = 2 * np.dot(d, np.array(p0) - q) # Coefficient b
7     c = np.dot(np.array(p0) - q, np.array(p0) - q) - epsilon**2 #
      Coefficient c

```

Listing 1.3: This function will calculate the free space of each border.

Υπολογίσουμε στην ουσία τους συντελεστές της ανισότητας που πρέπει να λύσουμε μέσω διανυσματικών πράξεων με χρήση του πακέτου numpy. Συνεχίζουμε με την διακρίνουσα και την εύρεση των ριζών:

```

1 disc = b**2 - 4*a*c # Discriminant of the equation
2 if disc < 0:
3     return None
4 sqrt_disc = np.sqrt(disc)
5 t1 = (-b - sqrt_disc) / (2*a) # Root 1
6 t2 = (-b + sqrt_disc) / (2*a) # Root 2
7 start = max(0.0, t1)
8 end = min(1.0, t2)
9 if start > end:
10     return None
11 return (start, end) # The endpoints of the free space

```

Listing 1.4: This function will calculate the free space of each border.

Υπολογίζουμε την ορίζουσα. Αν η ορίζουσα είναι μικρότερη του μηδενός δεν υπάρχουν πραγματικές ρίζες και σημεία τομής του κύκλου με την ευθεία. Αν η ορίζουσα δεν είναι μικρότερη του μηδέν, υπολογίζουμε τις ρίζες με την γνωστή από το σχολείο σχέση. Στη συνέχεια, ο ελεύθερος χώρος βρίσκεται μεταξύ των δυο ριζών. Για την περίπτωση που βγούμε έξω από το σύνορο, υπολογίζουμε τα μέγιστα των δυο ριζών με τα άκρα και κρατάμε αυτή την τιμή. Συνεπώς, η συνάρτηση έχει επιστρέψει το εύρος πάνω στο σύνορο το οποίο αποτελεί ελεύθερο χώρο.

Αρκεί τώρα, να τρέξουμε την παραπάνω συνάρτηση για όλα τα σύνορα του κελιού:

```

1 # Free-space intervals on borders
2 bottom = interval_within_epsilon(p_seg, Q[0], epsilon)
3 top = interval_within_epsilon(p_seg, Q[1], epsilon)
4 left = interval_within_epsilon(q_seg, P[0], epsilon)
5 right = interval_within_epsilon(q_seg, P[1], epsilon)

```

Listing 1.5: Run interval within epsilon on all borders.

και να ζωγραφίσουμε το αποτέλεσμα μέσω του παρακάτω κώδικα:

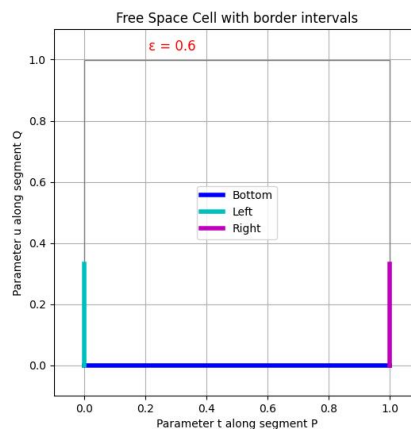
```

1 ### SECOND PLOT: Cell with free space intervals ###
2 fig2, ax2 = plt.subplots(figsize=(6, 6))
3
4 # Draw cell square (parameter space [0,1] x [0,1])
5 ax2.plot([0, 1, 1, 0, 0], [0, 0, 1, 1, 0], 'gray', lw=1)
6 ax2.set_aspect('equal')
7 ax2.set_xlim(-0.1, 1.1)
8 ax2.set_ylim(-0.1, 1.1)
9 ax2.grid(True)
10 ax2.set_title("Free Space Cell with border intervals")
11 ax2.set_xlabel("Parameter t along segment P")
12 ax2.set_ylabel("Parameter u along segment Q")
13
14 # Now, draw intervals only if present
15 if bottom:
16     ax2.plot([bottom[0], bottom[1]], [0, 0], 'b-', lw=4,
17             label='Bottom')
18     ....
19 if right:
20     ax2.plot([1, 1], [right[0], right[1]], 'm-', lw=4,
21             label='Right')
22 ax2.text(0.21, 1.03, f" = {epsilon}", fontsize=12, color='red')
23 ax2.legend()
24 plt.show()

```

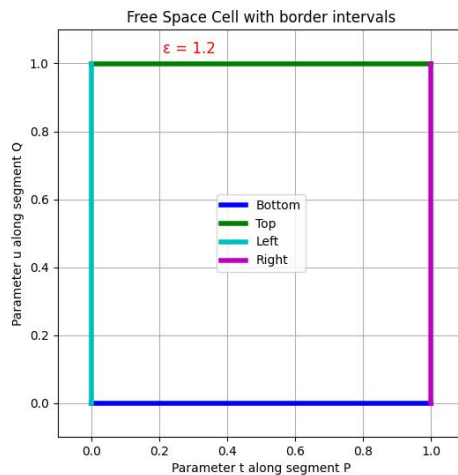
Listing 1.6: Draw the free space diagram for our example

Το αποτέλεσμα φαίνεται παρακάτω:



Σχήμα 1.15: Το διάγραμμα ελεύθερου χώρου για τα δυο ευθύγραμμα τμήματα. Έχουμε χρωματίσει στα σύνορα τις περιοχές που αποτελούν ελεύθερο χώρο.

Λόγω της απλής διάταξης των δυο ευθειών του παραδείγματος, ξέρουμε πως η απόσταση Fréchet των δυο ευθύγραμμων τμημάτων όση η απόσταση των δυο αρχικών σημείων δηλαδή γύρω στο 1.11. Ας θέσουμε το $\varepsilon = 1.2$ και ας δούμε αν τώρα στα χρωματιστά σύνορα περιλαμβάνεται το πάνω δεξιά σημείο:



Σχήμα 1.16: Το ε τώρα είναι ίσο με 1.2. Βλέπουμε πως τώρα υπάρχει μονοπάτι έως το πάνω δεξιά σημείο (όλος ο χώρος είναι ελεύθερος για το παράδειγμα αυτό).

Συνεπώς, για την πρώτη περίπτωση με $\varepsilon = 0.6$ η απόσταση Fréchet είναι μεγαλύτερη του ε ενώ για την δεύτερη περίπτωση με $\varepsilon = 1.2$ η απόσταση Fréchet είναι μικρότερη του με ε . Συνεχίζουμε με το γενικό πρόβλημα. Η εύρεση των ελεύθερων συνόρων για κάθε κελί αποτελεί την πρώτη φάση του ερωτήματος. Η δεύτερη φάση έγκειται στην αναζήτηση μονοπατιού από κάτω αριστερά προς τα πάνω δεξιά με τους περιορισμούς που έχουμε αναφέρει, ακολουθώντας την λογική που είδαμε στην προηγούμενη ενότητα.

1.4.2 Η απόφαση για το αν υπάρχει έγκυρο μονοπάτι

Για τον σκοπό αυτό θα θεωρήσουμε τώρα ένα παράδειγμα με καπώς πιο σύνθετες καμπύλες. Γράφουμε τον παρακάτω κώδικα:

```

1 # Define polygonal curves with 3 points (2 segments)
2 P = [(0, 0), (1, 0), (2, 1)] # Two segments: (0,0)-(1,0) and
   (1,0)-(2,1)
3 Q = [(0, 0), (0, 1), (1, 2)] # Two segments: (0,0)-(0,1) and
   (0,1)-(1,2)
4
5 # Store the segments of each curve

```

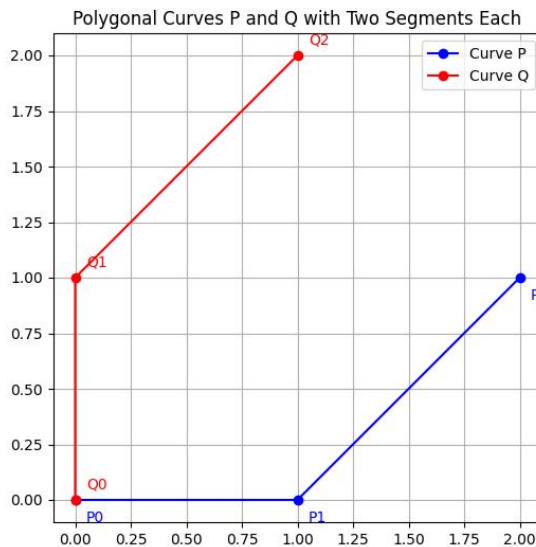
```

6 P_segments = [(P[i], P[i+1]) for i in range(len(P)-1)]
7 Q_segments = [(Q[i], Q[i+1]) for i in range(len(Q)-1)]
8
9 # Lets now plot the two curves
10 fig, ax = plt.subplots(figsize=(6, 6))
11 ax.plot(*zip(*P), 'bo-', label='Curve P')
12 ax.plot(*zip(*Q), 'ro-', label='Curve Q')
13
14 # Annotate points
15 for i, pt in enumerate(P):
16     ax.text(pt[0] + 0.05, pt[1] - 0.1, f'P{i}', color='blue')
17 for i, pt in enumerate(Q):
18     ax.text(pt[0] + 0.05, pt[1] + 0.05, f'Q{i}', color='red')
19
20 ax.set_title("Polygonal Curves P and Q with Two Segments Each")
21 ax.set_aspect('equal')
22 ax.grid(True)
23 ax.legend()
24 plt.show()

```

Listing 1.7: Lets use some more complex curves

όπου τώρα οι δυο καμπύλες είναι οι εξής:



Σχήμα 1.17: Πολυγωνικές καμπύλες με δυο τμήματα η κάθε μια.

Μπορούμε τώρα θέσουμε κάποιο ε και να τρέξουμε την συνάρτηση `interval with epsilon` για τα υπολογίσουμε τα σύνορα για κάθε κελί στο διάγραμμα ελεύθερου χώρου. Υπάρχουν τώρα φυσικά 4 κελιά, ένα για κάθε συνδυασμό τμημάτων.

Η απόσταση Fréchet για τις παραπάνω καμπύλες είναι βέβαια $\sqrt{2}$ όση και η απόσταση των δυο παράλληλων τμημάτων. Ας πάρουμε λοιπόν ένα ε μικρότερο και ας τρέξουμε τον κώδικα:

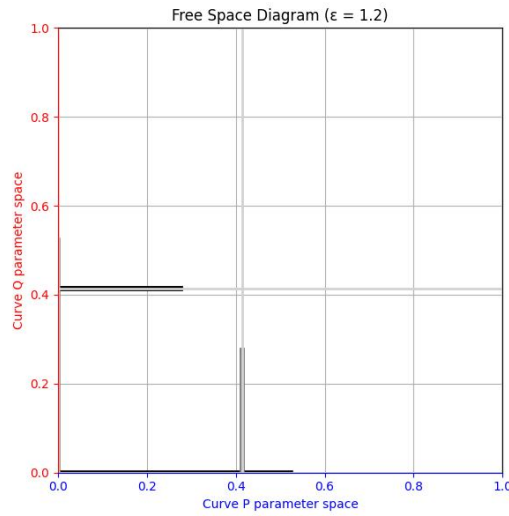
```

1 epsilon = 1.2
2 m, n = len(P_segments), len(Q_segments)
3 dx = 1 / m
4 dy = 1 / n
5
6 # Draw cell boundaries
7 for i in range(m+1):
8     ax.plot([i*dx, i*dx], [0, 1], 'gray', lw=1)
9 for j in range(n+1):
10    ax.plot([0, 1], [j*dy, j*dy], 'gray', lw=1)
11
12 # Draw free-space intervals
13 for i, p_seg in enumerate(P_segments):
14     for j, q_seg in enumerate(Q_segments):
15         x0 = i * dx
16         y0 = j * dy
17
18         # Bottom (P varies, Q fixed)
19         interval = interval_within_epsilon(p_seg, Q[j], epsilon)
20         if interval:
21             ax.plot([x0 + interval[0]*dx, x0 + interval[1]*dx],
22                     [y0, y0], 'b-', lw=4)
23
24         # Top
25         interval = interval_within_epsilon(p_seg, Q[j+1], epsilon)
26         if interval:
27             ax.plot([x0 + interval[0]*dx, x0 + interval[1]*dx],
28                     [y0+dy, y0+dy], 'g-', lw=4)
29
30         # Left (Q varies, P fixed)
31         interval = interval_within_epsilon(q_seg, P[i], epsilon)
32         if interval:
33             ax.plot([x0, x0], [y0 + interval[0]*dy, y0 +
34                         interval[1]*dy], 'c-', lw=4)
35
36         # Right
37         interval = interval_within_epsilon(q_seg, P[i+1], epsilon)
38         if interval:
39             ax.plot([x0+dx, x0+dx], [y0 + interval[0]*dy, y0 +
40                         interval[1]*dy], 'm-', lw=4)
41
42 plt.tight_layout()
43 plt.show()

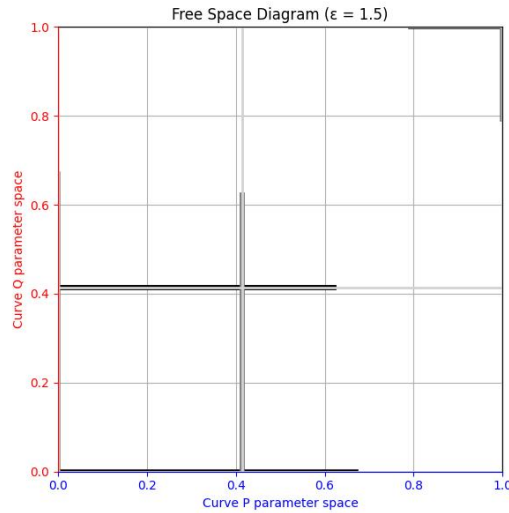
```

Listing 1.8: Find the free space border for each cell

με αποτέλεσμα τα γραφήματα που βλέπουμε στη συνέχεια για $\varepsilon = 1.2$ και για $\varepsilon = 1.5$.



Σχήμα 1.18: Ο ελεύθερος χώρος στα σύνορα κάθε κελιού με $\varepsilon = 1.2$



Σχήμα 1.19: Ο ελεύθερος χώρος στα σύνορα κάθε κελιού $\varepsilon = 1.5$

και άρα βλέπουμε οπτικά πως για $\varepsilon = 1.5$ υπάρχει μονοπάτι από κάτω δεξιά προς πάνω αριστερά. Καλό θα ήταν όμως να δίνει ο κώδικας την απάντηση αντί να κοιτάμε κάθε φορά την εικόνα. Για τον σκοπό αυτό πρέπει να αναβαθμίσουμε τον κώδικα με την λογική του αλγορίθμου της θεωρίας.

Αρχικά, δημιουργούμε μια συνάρτηση η οποία να αποθηκεύει τα σύνορα για κάθε κελί:

```
1 def cell_free_space(p_seg, q_seg, epsilon):
2     bottom = interval_within_epsilon(p_seg, q_seg[0], epsilon)
```

```

3     top = interval_within_epsilon(p_seg, q_seg[1], epsilon)
4     left = interval_within_epsilon(q_seg, p_seg[0], epsilon)
5     right = interval_within_epsilon(q_seg, p_seg[1], epsilon)
6     return {"bottom": bottom, "top": top, "left": left, "right":
        right}

```

Listing 1.9: Store the free borders of every cell

Πρέπει τώρα, να σκεφτούμε λίγο πιο σύνθετα. Έχουμε τώρα τα ελεύθερα σύνορα για κάθε κελί, ωστόσο δεν έχουμε καθαρή εικόνα για το ποιο κελί γειτονεύει με ποιο ώστε να συγκρίνουμε τα κατάλληλα κελιά. Για τον σκοπό αυτό θα φτιάξουμε μια δομή δεδομένων παρόμοια με το διάγραμμα ελεύθερου χώρου. Συγκεκριμένα θα φτιάξουμε ένα δισδιάστατο πίνακα με τόσα κελιά όσο και τα κελιά του διαγράμματος ελεύθερου χώρου. Σε κάθε κελί του νέου μας πίνακα, θα αποθηκεύσουμε τις τιμές για τα σύνορα των κελιών. Συνεπώς, με αυτή τη δομή μπορούμε άμεσα να συγκρίνουμε σύνορα και να ξέρουμε σε ποιο κελί βρισκόμαστε κάθε φορά. Ας δούμε πως θα γίνει αυτό:

```

1 free_space = [[None for _ in range(n)] for _ in range(m)]
2 for i in range(m):
3     for j in range(n):
4         p_seg = (P[i], P[i+1])
5         q_seg = (Q[j], Q[j+1])
6         free_space[i][j] = cell_free_space(p_seg, q_seg, epsilon)

```

Listing 1.10: Create a 2D data structure that holds the free borders of each cell

Δημιουργούμε τώρα ακόμη ένα δισδιάστατο πίνακα με την ίδια διάσταση με τον πίνακα free space όσα δηλαδή και τα κελιά ο οποίος θα περιέχει τιμές True και False στα στοιχεία του. Ο πίνακας αυτός θα μας δείχνει αν το κάθε κελί είναι προσβάσιμο από το προηγούμενό του. Αρχικά όλες οι τιμές θα είναι False και θα ξεκινήσουμε από το κάτω αριστερά κελί όπως είδαμε στον θεωρητικό αλγόριθμο. Καθώς σαρώνουμε όλα τα κελιά, αν το επόμενο κελί είναι προσβάσιμο από το προηγούμενο η τιμή αυτή θα αλλάζει σε True. Ισχύουν όλοι οι περιορισμοί που είδαμε στην θεωρία. Ας ξεκινήσουμε ορίζοντας τον πίνακα reachable:

```

1 # Reachability matrix: True if cell[i][j] is reachable, False
  otherwise
2 reachable = [[False for _ in range(n)] for _ in range(m)] # Same
  size as the initial cells

```

Listing 1.11: Another 2D structure that will hold the reachability of every cell

Στη συνέχεια ακολουθούμε τα βήματα του αλγορίθμου και ξεκινάμε από το πρώτο κελί κάτω αριστερά:

```

1 # Step 1: Initialize bottom-left corner
2 start_cell = free_space[0][0] # We start from bottom left cell
3 if start_cell["bottom"] and start_cell["left"]: # Check if bottom
   left cell has free space
4     if start_cell["bottom"][0] <= 0.0 and start_cell["left"][0]
       <= 0.0: # Check if bottom left point is included in free
         space
5         reachable[0][0] = True # If so, then the first cell is
           reachable

```

Listing 1.12: Initiallize starting from bottom left

Ελέγχουμε αν το αρχικό σημείο περιλαμβάνεται στον ελεύθερο χώρο. Αν όχι δεν έχει νόημα να συνεχίσουμε αφού έχουμε ήδη αποτύχει. Ο έλεγχος γίνεται σε δυο βήματα, πρώτα ελέγχουμε αν υπάρχουν τα κάτω και αριστερά σύνορα (αν δηλαδή υπάρχει έστω και ένα σημείο ελεύθερου χώρου σε αυτά) και έπειτα αν το ίδιο το αρχικό σημείο περιλαμβάνεται στον ελεύθερο χώρο. Αν πράγματι ισχύει αυτό, τότε θεωρούμε έγκυρο το πρώτο κελί και συνεχίζουμε:

```

1 # Step 2: Propagate reachability
2 for i in range(m): # Loop over all cells including the
   starting one
3     for j in range(n):
4         if not reachable[i][j]: # Ignore cell if it is not
           reachable
5             continue
6         cell = free_space[i][j] # The current cell we work on
7         # Right neighbor. This condition checks first if there is
           a cell to the right,
8         # second if there is free space in the right border
9         # and third if the cell to the right has any fre space on
           its left border
10        if j + 1 < n and cell["right"] and
           free_space[i][j+1]["left"]:
11            reachable[i][j+1] = True
12        # Top neighbor
13        if i + 1 < m and cell["top"] and
           free_space[i+1][j]["bottom"]:
14            reachable[i+1][j] = True
15        # Diagonal neighbor (top-right)
16        if i + 1 < m and j + 1 < n:
17            if cell["top"] and cell["right"]:
18                if cell["top"][1] == 1.0 and cell["right"][1] ==
                   1.0:
19                    reachable[i+1][j+1] = True

```

Listing 1.13: Propagate the reachability over all cells

Σαρώνουμε τώρα όλα τα κελιά ξεκινώντας από το πρώτο. Αν το κελί δεν είναι προσβάσιμο από τα γειτονικά του, το απορρίπτουμε και συνεχίζουμε. Στη συνέχεια ελέγχουμε τα εξής: 1) αν υπάρχει κελί στα δεξιά του, 2) αν το κελί έχει ελεύθερο δεξί σύνορο και 3) αν το κελί στα δεξιά του έχει αριστερό ελεύθερο σύνορο. Αν αυτά τα σύνορα υπάρχουν είναι προφανώς ίδια. Τότε το κελί στα δεξιά είναι προσβάσιμο και άρα ανανεώνουμε τη μεταβλήτη στο πίνακα *reachable*. Τα ίδια και για το πάνω σύνορο. Τέλος, ελέγχουμε και την ειδική περίπτωση το πάνω δεξί σημείο του κελιού να ανήκει στο ελεύθερο χώρο και άρα μέσω αυτού να μπορούμε στο πάνω και δεξιά διαγώνιο κελί.

Στο τέλος, πρέπει να ελέγχξουμε ειδικά το πάνω δεξιά κελί και συγκεκριμένα αν το πάνω δεξιά σημείο περιλαμβάνεται στον ελεύθερο χώρο:

```

1 # Final decision
2 final_cell = free_space[m-1][n-1]
3 if reachable[m-1][n-1] and final_cell["top"] and
   final_cell["right"]:
4     if final_cell["top"][1] >= 1.0 and final_cell["right"][1] >=
       1.0:
5         print("YES")
6     else:
7         print("NO")

```

Listing 1.14: Check whether the top right point is free

Τσεκάρουμε συγκεκριμένα το αν τα τελικά σημεία του ελεύθερου χώρου του πάνω και δεξιού συνόρου είναι ίσα ή μεγαλύτερα από το 1. Χρησιμοποιούμε το μεγαλύτερο ή ίσο για να αποφύγουμε τυχόν στρογγυλοποιήσεις που μπορεί να αλλοιώσουν την μονάδα. Αν τα σημεία αυτά είναι πράγματι μεγαλύτερα ή ίσα με το 1 τότε το πάνω δεξιά σημείο ανήκει στον ελεύθερο χώρο και έτσι έχουμε καταλήξει στο τέλος της διαδρομής. Απαντάμε τότε πως ναι, η απόσταση Fréchet είναι μικρότερη από ε , αλλιώς όχι η απόσταση Fréchet είναι μεγαλύτερη από ε .

Μπορούμε τώρα να τρέξουμε τον κώδικα με ένα for loop ώστε να δοκιμάσουμε διάφορα ε . Χρησιμοποιούμε το παράδειγμα με τις πιο σύνθετες καμπύλες του σχήματος 1.17. Θυμηθείτε πως η απόσταση Fréchet για το παράδειγμα αυτό είναι $\sqrt{2}$ δηλαδή γύρω στο 1.44. Παρατηρείστε την αλλαγή της απάντησης γύρω από αυτό το όριο:

```

Epsilon is: 0.60
NO
Epsilon is: 0.80
NO
Epsilon is: 1.00

```

```

NO
Epsilon is: 1.20
NO
Epsilon is: 1.40
NO
Epsilon is: 1.60
YES
Epsilon is: 1.80
YES

```

Έχουμε τελειώσει λοιπόν με τον αλγόριθμο απόφασης. Ποιά είναι όμως η χρονική του πολυπλοκότητα; Αρχικά, ο υπολογισμός των συνόρων γίνεται μέσω της επίλυσης μιας τετραγωνικής ανισότητας. Αυτό απαιτεί σταθερό χρόνο καθώς εκτελούμε μόνο κάποιους πολλαπλασιασμούς και διαιρέσεις. Υπάρχουν συνολικά $p \times q$ κελιά, όπου p, q τα συνολικά τμήματα κάθε πολυγωνικής καμπύλης. Επιπλέον, η διάδοση της προσβασιμότητας εκτελεί μόνο κάποιες συγκρίσεις πάλι σταθερού χρόνου. Τελικά λοιπόν, η χρονική πολυπλοκότητα είναι:

$$O(pq)$$

Συνεπώς, θέλουμε χρόνο $O(pq)$ ώστε να αποφασίσουμε αν $d_F \leq \varepsilon$.

1.4.3 Ο υπολογισμός των κρίσιμων ε

Η απάντηση βέβαια στο πρόβλημα της απόφασης δεν μας δίνει ποια τελικά είναι η απόσταση Fréchet. Ωστόσο, όπως γίνεται εύκολα κατανοητό, η απόσταση Fréchet είναι ίση με το ε για το οποίο μόλις που υπάρχει έγκυρο μονοπάτι από κάτω αριστερά προς πάνω δεξιά. Πρέπει λοιπόν να βρούμε ποιο είναι αυτό το ε για το οποίο η απόφαση του προηγούμενου αλγορίθμου αλλάζει από NO σε YES.

Ο πιο αφελής τρόπος να γίνει αυτό είναι βέβαια να εκτελέσουμε μια επανάληψη while αυξάνοντας το ε κατά κάποια τιμή και να κρατήσουμε την τιμή για την οποία θα αλλάξει η απόφαση. Η μέθοδος αυτή εμπεριέχει κάποια προβλήματα:

1. Δεν θα βρούμε την ακριβή τιμή της απόστασης Fréchet αλλά μια κοντινή σε αυτή τιμή ανάλογα με το βήμα που αλλάζουμε το ε .

2. Όσο λοιπόν μεγαλώνουμε το βήμα ώστε να πετύχουμε καλύτερη ακρίβεια, τόσο αυξάνει και ο χρόνος εκτέλεσης της μεθόδου.
3. Δεν μπορούμε να ξέρουμε ποιά θα ήταν το καλύτερο όριο από όπου να ξεκινήσει η επανάληψη.

Για τον λόγο αυτό, πρέπει να σκεφτούμε πιο έξυπνα. Όσο μεγαλώνει το ε τόσο μεγαλώνει και ο ελεύθερος χώρος στο αντίστοιχο γράφημα. Στην περίπτωση μας, αλλαγή στην συμπεριφορά του αλγορίθμου απόφασης μπορεί να προκύψει μόνο σε κάποιες συγκεκριμένες τιμές του ε , εκείνες για τις οποίες δυο μη γειτονικά κελιά μετατρέπονται σε γειτονικά. Κατά συνέπεια, μπορούμε να εισέλθουμε σε νέα κελιά και έτσι να βρούμε κάποια νέα διαδρομή προς τα πάνω δεξιά. Συνεπώς, δεν είναι απαραίτητο να ψάξουμε όλες τις του ε παρά μόνο εκείνες οι οποίες είναι κρίσιμες, δηλαδή υπάρχει αλλαγή στη συμπεριφορά του ελεύθερου χώρου.

Οι κρίσιμες τιμές μπορεί να είναι δυο κατηγοριών. Στην πρώτη κατηγορία έχουμε τις αποστάσεις $\|P_i - Q_j\|$ μεταξύ των κορυφών των δυο καμπύλων. Όταν το ε μεγαλώσει αρκετά ώστε να γίνει ίσο με την απόσταση δυο κορυφών, τότε τα κελιά που αντιστοιχούν στα ευθύγραμμα τμήματα που φέρονται από τις κορυφές (4 κελιά, ένα για κάθε συνδυασμό τμημάτων) γίνονται πλέον προσβάσιμα μέσω της γωνία τους. Στη δεύτερη κατηγορία έχουμε τις αποστάσεις μεταξύ σημείου και ευθύγραμμου τμήματος. Όταν το ε φτάσει σε μια τέτοια τιμή, τότε δυο κελιά γίνονται γειτονικά μέσω κάποιου σημείου των συόρων τους.

Ας δούμε πως θα γίνουν τα παραπάνω σε κώδικα. Πως βρίσκουμε την συντομότερη απόσταση από ένα σημείο προς μια ευθεία; Η μέθοδος που γνωρίζουμε είναι να φέρουμε μια κάθετη από το σημείο στην ευθεία. Το μήκος αυτής της κάθετου είναι η απόσταση σημείου και ευθείας. Αυτή τη λογική θα εφαρμόσουμε και εδώ. Έστω ένα σημείο q και τα άκρα της ευθείας p_1, p_0 . Εμείς εδώ έχουμε αυτά τα άκρα ως δεδομένα. Υπολογίζουμε αρχικά το διάνυσμα που χαρακτηρίζει το τμήμα:

$$d = p_1 - p_0 \quad (1.5)$$

Στη συνέχεια βρίσκουμε το διάνυσμα από το ένα άκρο της ευθείας έστω το p_0 προς το σημείο:

$$u = q - p_0 \quad (1.6)$$

Υπολογίζουμε τώρα την προβολή του διανύσματος αυτού πάνω στην ευθεία:

$$t = \frac{ud}{\|d\|^2} \quad (1.7)$$

Η προβολή αυτή μας δείχνει πάνω από πιο σημείο της ευθείας βρίσκεται το απομονωμένο σημείο. Θέλει βέβαια προσοχή γιατί μπορεί το σημείο να βρίσκεται εκτός των ορίων της ευθείας. Για τον λόγο αυτό παίρνουμε τον παρακάτω περιορισμό:

$$t_c = \min(\max(t, 0), 1) \quad (1.8)$$

Αν η προβολή είναι μικρότερη από κάποιο άκρο του ευθύγραμμου τμήματος, τότε υπολογίζουμε την απόσταση σε σχέση με το άκρο αυτό. Αυτός είναι ο σκοπός της παραπάνω σχέσης. Ξέρουμε τώρα πάνω από ποιο σημείο του τμήματος βρίσκεται το σημείο μας, ας το υπολογίσουμε:

$$p(t_c) = p_0 + t_c d \quad (1.9)$$

όπου βέβαια έχουμε την συνηθισμένη παραμετροποίηση. Απομένει τώρα να βρούμε το μήκος της καθέτου από το σημείο στην ευθεία το οποίο είναι:

$$\|q - p(t_c)\| \quad (1.10)$$

Η συνάρτηση που υπολογίζει τα παραπάνω είναι η εξής:

```

1 def point_segment_distance(p, seg):
2     #Compute the shortest distance from point p to segment (s0,
3       s1).
4     s0, s1 = np.array(seg[0]), np.array(seg[1]) # Segment
5     p = np.array(p) # Single Point
6     d = s1 - s0 # d = p0 - p1
7     l2 = np.dot(d, d) # ||d|| ^ 2
8     if l2 == 0: # Check if segment is just a point
9         return np.linalg.norm(p - s0)
10    t = np.dot(p - s0, d) / l2 # Find the projection
11    t = np.clip(t, 0, 1) # Clip it in [0,1]
12    projection = s0 + t * d # Find the point in the segment
    return np.linalg.norm(p - projection) # Return the distance

```

Listing 1.15: This function calculates the closest distance between a point and a segment. Can be used to find the distance between points as well.

Τώρα μπορούμε να υπολογίσουμε τα κρίσιμα ε , σαρώνοντας όλα τα σημεία των καμπύλων μεταξύ τους και σε σχέση με τα ευθύγραμμα τμήματα:

```

1 def compute_critical_epsilon(P, Q):
2     # Define this as set to avoid duplicates
3     epsilons = set()
4
5     # Vertex-vertex distances
6     for p in P:
7         for q in Q:
8             # Find the distances of all the points of the curves
9             eps = np.linalg.norm(np.array(p) - np.array(q))
10            epsilons.add(eps)
11
12    # Vertex-to-segment distances: P vertex to Q segment
13    for p in P:
14        for i in range(len(Q) - 1):
15            q_seg = (Q[i], Q[i + 1]) # Segment
16            eps = point_segment_distance(p, q_seg) # Use function
17            epsilons.add(eps)
18
19    # Vertex-to-segment distances: Q vertex to P segment
20    for q in Q:
21        for i in range(len(P) - 1):
22            p_seg = (P[i], P[i + 1]) # Segment
23            eps = point_segment_distance(q, p_seg) # Use function
24            epsilons.add(eps)
25
26    # Return sorted list of unique critical values
27    return sorted(epsilons)

```

Listing 1.16: Calculate the critical ε where the free space changes behaviour.

Τα σχόλια επεξηγούν κάθε σημείο στον κώδικα. Επιστρέφουμε τα ε ταξινομημένα για τον λόγο που θα δούμε στην συνέχεια. Πρώτα όμως ας δούμε το αποτέλεσμα του κώδικα για το γνωστό πλέον παράδειγμα του σχήματος 1.17 με τις δυο απλές πολύγωνες καμπύλες. Εμφανίζουμε τα αποτελέσματα με το παρακάτω κομμάτι κώδικα:

```

1 P = [(0, 0), (1, 0), (2, 1)]
2 Q = [(0, 0), (0, 1), (1, 2)]
3
4 critical_eps = compute_critical_epsilon(P, Q)
5 print("Critical epsilon values:")
6 for eps in critical_eps:
7     print(f"{eps:.6f}")

```

Listing 1.17: Print the results for the example curves

με αποτέλεσμα:

Critical epsilon values:

0.000000

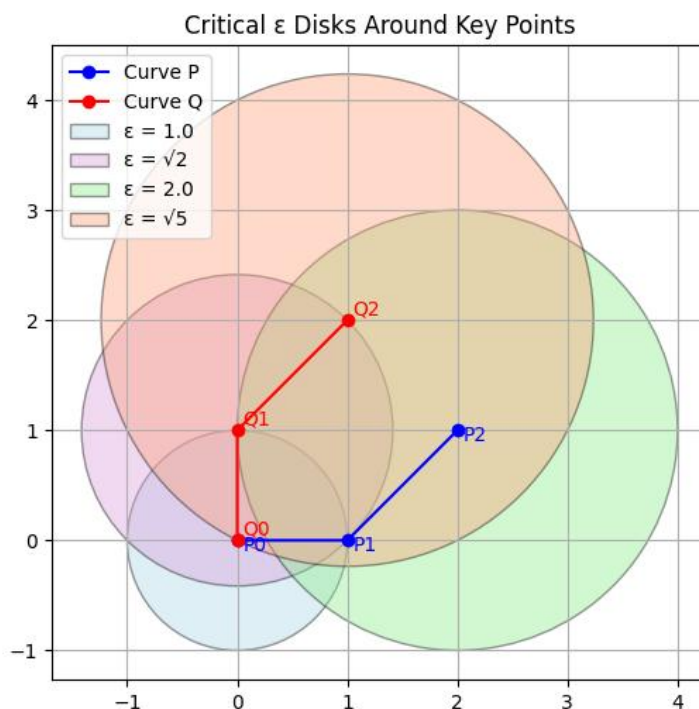
1.000000

1.414214

2.000000

2.236068

Αν σας θυμίζει κάτι το 1.414 είναι πράγματι το $\sqrt{2}$ που είναι και η σωστή απόσταση Fréchet για το παράδειγμα. Το 0 προκύπτει επειδή οι δυο πρώτες ευθείες ακουμπούν. Το 1 προκύπτει όταν η κάτω μπλε ευθεία ακουμπά την πάνω κόκκινη ή όταν η αριστερή κόκκινη ακουμπά την δεξιά μπλε. Ομοίως για τα υπόλοιπα. Μπορούμε να έχουμε μια εποπτεία αν σχεδιάσουμε τος αντίστοιχους κύκλους. Ας δούμε το παρακάτω σχήμα:



Σχήμα 1.20: Οι κύκλοι ακτίνας ίση με τις κρίσιμες τιμές. Κάθε κύκλος δείχνει το όριο που ένα νέο κελί γίνεται προσβάσιμο.

Δείτε πως σε κάθε κρίσιμη τιμή, δυο νέα ευθύγραμμα τμήματα έρχονται σε επαφή, δηλαδή αποκτούν κοινό σύνορο στο διάγραμμα ελεύθερου χώρου. Για πα-

ράδειγμα, στην κρίσιμη τιμή $\sqrt{2}$ έρχονται σε επαφή το σημείο q_1 με το τμήμα p_1, p_2 ενώ για κρίσιμη τιμή 2 έρχονται σε επαφή το p_2 με το τμήμα q_0, q_1 .

1.4.4 Δυαδική αναζήτηση και εύρεση της απόστασης Fréchet

Με τον ταξινομημένο πίνακα των κρίσιμων τιμών διαθέσιμο, μπορούμε τώρα να ψάξουμε την απόσταση Fréchet η οποία είδαμε πως είναι κάποια από αυτές τις τιμές. Για κάθε κρίσιμη τιμή έχουμε επιπλέον την απάντηση του κώδικα απόφασης, YES ή NO. Ταξινομήσαμε τον πίνακα ώστε να χρησιμοποιήσουμε δυαδική αναζήτηση. Η δυαδική αναζήτηση είναι χρόνου $O(\log n)$ και αρκετά χρήσιμη στην περίπτωση μας. Όλες οι κρίσιμες τιμές μικρότερες της απόστασης Fréchet δίνουν απάντηση NO ενώ όλες οι κρίσιμες τιμές μεγαλύτερες ή ίσες με την απόσταση Fréchet δίνουν απάντηση YES. Συνεπώς, εμείς ψάχνουμε την ελάχιστη κρίσιμη τιμή που δίνει απάντηση YES.

Η δυαδική αναζήτηση λειτουργεί ως εξής: Ξεκινάμε από το μεσαίο στοιχείο του πίνακα των κρίσιμων τιμών. Ανάλογα με το αν δώσει απάντηση YES ή NO, ξέρουμε να ψάξουμε το αριστερό ή το δεξί μισό του πίνακα. Σε κάθε βήμα λοιπόν υποδιπλασιάζονται τα υπό εξέταση στοιχεία για αυτό και ο λογαριθμικός χρόνος εκτέλεσης. Πρέπει ωστόσο ο πίνακας να είναι ταξινομημένος. Ας δούμε τον κώδικα για την διαδικασία αυτή. Αρχικά περικλείουμε τον κώδικα της απόφασης σε μια συνάρτηση:

```

1 def decision(P, Q, epsilon):
2     ....
3     if reachable[m-1][n-1] and final_cell["top"] and
        final_cell["right"]:
4         if final_cell["top"][1] >= 1.0 and final_cell["right"][1]
            >= 1.0:
5             return True
6     return False

```

Listing 1.18: Include the code in a function

Κατά αυτόν τον τρόπο δεν τυπώνουμε το αποτέλεσμα στην οθόνη αλλά το επιστρέφουμε μέσω της συνάρτησης. Στη συνέχεια, γράφουμε την συνάρτηση της δυαδικής αναζήτησης:

```

1     critical_epsilons = compute_critical_epsilons(P, Q) # The
        critical_epsilons
2     left = 0 # Start with the middle element of the array
3     right = len(critical_epsilons) - 1
4     result = critical_epsilons[-1]
5

```

```

6   while left <= right:
7       mid = (left + right) // 2
8       eps = critical_epsilon[mid]
9       if decision(P, Q, eps): # Call the decision function
10          result = eps        # Keep the lowest element that
11          answers YES
12          right = mid - 1
13       else:
14          left = mid + 1
15   return result
16
17 P = [(0, 0), (1, 0), (2, 1)] # Curve P
18 Q = [(0, 0), (0, 1), (1, 2)] # Curve Q
19 frechet = frechet_distance(P, Q)
20 print(f"Frechet distance: {frechet:.6f}")

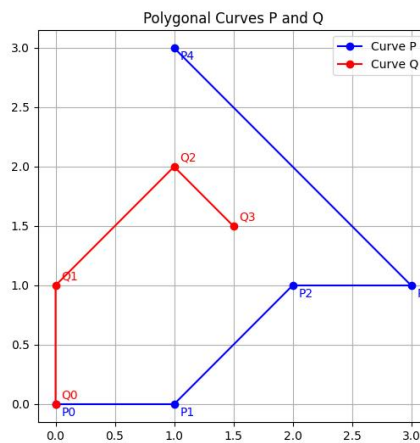
```

Listing 1.19: Binary search on the critical values array. The result is the Fréchet distance of the two curves

Το αποτέλεσμα είναι: *Fréchet distance: 1.414214*, που είναι βέβαια το σωστό. Λόγω της εύρεση των κρίσιμων τιμών και της δυαδικής αναζήτησης η πολυπλοκότητα χρόνου έχει αυξηθεί και τώρα δίνεται ως

$$O((p^2q + q^2p) \log(pq))$$

Ας δοκιμάσουμε ακόμη ένα παράδειγμα ώστε να δούμε αν θα δώσει αποτέλεσμα ο κώδικας. Έστω οι παρακάτω καμπύλες:



Σχήμα 1.21: Ένα πιο σύνθετο παράδειγμα.

Η απόσταση Fréchet εδώ δίνεται από την απόσταση q_3 και p_3 η οποία σύμφωνα με το σχήμα είναι $\sqrt{2.5} = 1.5811$. Θα δώσει ο κώδικας το σωστό αποτέλεσμα; Ας δούμε την έξοδο:

```
Critical epsilon values:
0.000000
0.707107
1.000000
1.414214
1.581139
2.000000
2.121320
2.236068
2.828427
3.000000
3.162278
Frechet distance: 1.581139
```

Πράγματι, ο κώδικας λειτουργεί σωστά. Προσοχή στο γεγονός πως όλα τα παραπάνω ισχύουν για ανοικτές πολύγωνα καμπύλες. Τα πράγματα είναι κάπως πιο διαφορετικά για κλειστές καμπύλες.

Κεφάλαιο 2

Αναζητήσεις γειτονικών καμπύλων

2.1 Τοποθέτηση του προβλήματος

Στο πρώτο κεφάλαιο είδαμε αναλυτικά τον ορισμό της απόστασης Fréchet, τον αλγόριθμο των Alt και Godau για τον υπολογισμό της και τέλος τον κώδικα σε python που την επιστρέφει. Χτίζοντας πάνω σε αυτή τη βάση, θα επιχειρήσουμε να βρούμε ένα γρήγορο και αποδοτικό αλγόριθμο ο οποίος για μια δεδομένη καμπύλη εισόδου να βρίσκει ποιές από τις καμπύλες μιας βάσης δεδομένων έχουν μορφή παραπλήσια της εισόδου, με κάποια ανοχή. Η μετρική για την αναζήτηση αυτή θα είναι βέβαια η απόσταση Fréchet.

Η σύγκριση γεωμετρικών σχημάτων είναι μια εργασία που εμφανίζεται σε ένα ευρύ πεδίο εφαρμογών. Η απόσταση Fréchet και οι παραλλαγές της έχουν χρησιμοποιηθεί για το σκοπό αυτό, για τη σύγκριση καμπυλών σε εφαρμογές όπως η δυναμική χρονική στρέβλωση, η αναγνώριση ομιλίας, η αναγνώριση υπογραφών και γραφής, η αντιστοίχιση χρονοσειρών σε βάσεις δεδομένων, καθώς και γεωγραφικές εφαρμογές, όπως η αντιστοίχιση χαρτών με δεδομένα εντοπισμού οχημάτων και η ανάλυση κινούμενων αντικειμένων.

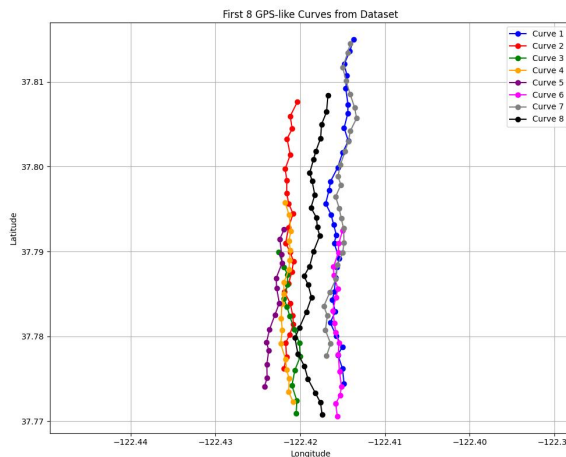
2.2 Τα δεδομένα: καμπύλες από διαδρομές GPS

Το πεδίο ενδιαφέροντος για εμάς θα είναι οι γεωγραφικές εφαρμογές. Πιο συγκεκριμένα, τα δεδομένα μας αποτελούν μια βάση δεδομένων από πολυγωνικές καμ-

πύλες οι οποίες προέρχονται από δεδομένα κίνησης GPS στο κέντρο μιας μεγάλης πόλης. Λόγω ακριβώς αυτής της ιδιότητας, οι καμπύλες ανήκουν στην κατηγορία των ρεαλιστικών καμπύλων και έχουν τα εξής δυο σημαντικά χαρακτηριστικά:

1. Υπάρχουν λίγες απότομες στροφές, δηλαδή λίγες έντονα οξείες γωνίες. Προφανώς οι οδηγοί στρίβουν το πολύ κατά 90 μοίρες καθώς ακολουθούν την διαδρομή τους.
2. Δυο καμπύλες με παραπλήσια σημεία αρχής και τέλους έχουν μικρή απόσταση Fréchet

Τα δεδομένα βρίσκονται σε ένα αρχείο `curvedataset.csv` και αφορούν 100 τον αριθμό πολυγωνικές καμπύλες με 10 έως 30 σημεία η καθεμία. (ΣΗΜΕΙΩΣΗ: Τα πραγματικά δεδομένα του διαγωνισμού ACM SIGSPATIAL Cup 2017 δεν φαίνεται να είναι διαθέσιμα και για τον λόγο αυτό τα δεδομένα που χρησιμοποιώ εδώ είναι τεχνικά κατασκευασμένα βάση των δυο παραπάνω χαρακτηριστικών). Κάθε γραμμή αφορά μια καμπύλη και τα σημεία της δίνονται σε διαδοχικές στήλες με πρώτα το x_1 και δίπλα του το y_1 , ενώ οι συντεταγμένες χωρίζονται μεταξύ τους με το σύμβολο `|`. Η πρώτη γραμμή αναγράφει το όνομα κάθε στήλης (x και y αντίστοιχα). Θα κρίνουμε τα παραπάνω χαρακτηριστικά ιδίως όμμασι σχεδιάζοντας μερικές από τις καμπύλες. Δείτε τις πρώτες 8 καμπύλες των δεδομένων:



Σχήμα 2.1: Οι πρώτες 8 καμπύλες από τη βάση δεδομένων

Μπορούμε επιπλέον να τρέξουμε τον αλγόριθμο για την εύρεση της απόστασης Fréchet που αναπτύξαμε στο προηγούμενο κεφάλαιο. Αρχικά εξάγουμε κάποιες από τις καμπύλες, έστω τις δυο από αυτές που φαίνονται και στο σχήμα, μέσω της παρακάτω συνάρτησης:

```

1 def extract_curves_as_lists(filename):
2     with open(filename, 'r') as f:
3         lines = f.readlines()[1:] # Skip header
4
5         # Process first two lines (first two curves)
6         curves = []
7         for line in lines[:2]:
8             line = line.strip()
9             if not line:
10                continue
11            points = [
12                tuple(map(float, pair.strip().split()))
13                for pair in line.split('|')
14            ]
15            curves.append(points)
16
17            # Unpack as P and Q
18            P, Q = curves[0], curves[1]
19            return P, Q
20
21 P, Q = extract_curves_as_lists("curve_dataset.csv")

```

Listing 2.1: Find the Fréchet distance for the 2 first curves of the database.

Στη συνέχεια εισάγουμε τα P και Q στον κώδικα του προηγούμενου κεφαλαίου και το αποτέλεσμα είναι:

```

Critical epsilon values:
0.004029
0.004353
.....
0.038228
0.039656
Frechet distance: 0.010398

```

2.3 Επεξεργασία των δεδομένων

2.3.1 Κάτω φράγμα για την απόσταση Fréchet

Ως είσοδο για το πρόβλημα έχουμε κάποια καμπύλη, διαφορετική από τις καμπύλες της βάσης δεδομένων και μια τιμή δ ως όριο ανοχής για το οποίο ψάχνουμε στη βάση δεδομένων για καμπύλες με μικρότερη απόσταση Fréchet σε σχέση με

την είσοδο. Ο πιο αφελής τρόπος θα ήταν να τρέξουμε τον κώδικά μας για όλες τις καμπύλες της βάσης δεδομένων και να βρούμε τελικά την μικρότερη τιμή. Σύμφωνα με τις πολυπλοκότητες που είδαμε παραπάνω, αν η καμπύλη εισόδου έχει n κορυφές, ενώ η βάση δεδομένων έχει k καμπύλες με m κορυφές η κάθε μια, τότε η χρονική πολυπλοκότητα είναι $O(knm \log(nm))$, δηλαδή τετραγωνικού χρόνου.

Ωστόσο, μπορούμε να μειώσουμε κατά πολύ τους υπολογισμούς αν χρησιμοποιήσουμε τα χαρακτηριστικά τα 2 χαρακτηριστικά των ρεαλιστικών καμπύλων που αναφέραμε προηγουμένως. Οι καμπύλες αυτές έχουν συγκεκριμένη συμπεριφορά η οποία αυτομάτως κάνει πιο πιθανή την εγγύτητα μεταξύ εκείνων των καμπυλών τα σημεία των οποίων βρίσκονται πιο κοντά μεταξύ τους.

Μια παρατήρηση για την απόσταση Fréchet μεταξύ δυο καμπύλων την οποία αναφέραμε και στην προηγούμενη ενότητα είναι πως η απόσταση αυτή είναι σίγουρα μεγαλύτερη ή ίση με την απόσταση των δυο αρχικών σημείων. Με άλλα λόγια, πρέπει το κάτω αριστερά σημείο του διαγράμματος ελεύθερου χώρου να ανήκει στον ελεύθερο χώρο αλλιώς έχουμε αποτύχει εξ'αρχής. Το ίδιο βέβαια πρέπει να συμβαίνει και με τα τελικά σημεία και το πάνω δεξιά σημείο του διαγράμματος ελεύθερου χώρου. Ονομάζουμε τις δυο καμπύλες P, Q με n, m σημεία αντίστοιχα. Τότε, σύμφωνα με τα παραπάνω έχουμε:

$$\|p_1 - q_1\| \leq d_F(P, Q)$$

$$\|p_n - q_m\| \leq d_F(P, Q)$$

Μπορούμε ωστόσο να φράξουμε την απόσταση Fréchet ακόμη περισσότερο χρησιμοποιώντας τα ελάχιστα σημεία κάθε καμπύλης ως προς τις δυο συντεταγμένες. Έστω οι ελάχιστες τιμές $\min x(P)$ και $\min x(Q)$, δηλαδή από όλες τις x συντεταγμένες των σημείων της καμπύλης εντοπίζουμε την ελάχιστη. Έστω τώρα πως βρισκόμαστε πάνω στο σημείο ελάχιστης x συντεταγμένης της καμπύλης P . Σίγουρα κάποια στιγμή της πορείας μας πάνω στην καμπύλη P θα περάσουμε από το σημείο αυτό. Την ίδια χρονική στιγμή, μπορεί να βρισκόμαστε οπουδήποτε πάνω στην Q καμπύλη, ανάλογα με την παραμετροποίηση που έχουμε επιλέξει. Η απόσταση από αυτό το αυθαίρετο σημείο πάνω στην Q είναι τουλάχιστον όσο η διαφορά των δυο ελάχιστων συντεταγμένων και άρα η απόσταση Fréchet φράσσεται από αυτήν:

$$|\min x(P) - \min x(Q)| \leq d_F \quad (2.1)$$

Διαισθητικά μπορούμε να δούμε την παραπάνω σχέση ως εξής: όταν βρισκό-

μαστε στο ελάχιστο σημείο x της P καμπύλης, το κοντινότερο σημείο της Q καμπύλης θα έχει x συντεταγμένη κατ'ελάχιστον $\min x(Q)$. Αν έχει x συντεταγμένη ίση με την ελάχιστη τότε η απόστασή τους είναι ακριβώς η σχέση (2.1) Αν έχει κάποια μεγαλύτερη συντεταγμένη x από την ελάχιστη τιμή της καμπύλης Q , τότε η απόστασή τους είναι σίγουρα όσο η σχέση (2.1) συν κάτι επιπλέον επειδή το σημείο βρίσκεται πιο δεξιά.

Μπορούμε τα παραπάνω να τα δούμε και μαθηματικά, κάτι που θα βοηθήσει και στην διαίσθησή μας για τον κώδικα. Ας ονομάσουμε τις ακραίες συντεταγμένες:

$$a = \min x(P)$$

$$b = \min x(Q)$$

για ευκολία και έστω $a < b$ χωρίς βλάβη της γενικότητας. Όπως είδαμε αναλυτικά, ισχύει πως:

$$|b - a| \leq d_F$$

Θεωρούμε τώρα δυο συνεχής μη μειούμενες παραμετροποιήσεις των δυο καμπύλων έστω $a(t)$, $b(t)$. Από την σχέση (1.1) του ορισμού της απόστασης Fréchet ισχύει πως για αυτές τις παραμετροποιήσεις:

$$\max_t \|P(a(t)) - Q(b(t))\| \leq d_F(P, Q)$$

Έστω την χρονική στιγμή t^* βρισκόμαστε πάνω στο σημείο της P με την ελάχιστη συντεταγμένη. Τότε στην Q βρισκόμαστε σε κάποιο άλλο τυχαίο σημείο και έχουμε πως:

$$\|P(a(t^*)) - Q(b(t^*))\| \geq |b - a|$$

και συνεπώς:

$$d_F(P, Q) \geq |b - a| = |\min x(P) - \min x(Q)| \quad (2.2)$$

Η παραπάνω ανάλυση ισχύει αυτούσια και για τις μέγιστες συντεταγμένες κατά x , αλλά και για τις ελάχιστες και μέγιστες συντεταγμένες κατά y . Συνολικά λοιπόν, έχουμε υπολογίσει 6 φράγματα για την απόσταση Fréchet. Από όλα αυτά τα φράγματα εκείνο που μας ενδιαφέρει είναι προφανώς το μεγαλύτερο:

$$d_F(P, Q) \geq \max \left\{ \begin{array}{l} \|p_1 - q_1\|, \\ \|p_n - q_m\|, \\ |\min x(P) - \min x(Q)|, \\ |\min y(P) - \min y(Q)|, \\ |\max x(P) - \max x(Q)|, \\ |\max y(P) - \max y(Q)| \end{array} \right\} = LB_F(P, Q)$$

Γιατί όμως η παραπάνω ανάλυση; Γιατί τώρα ξέρουμε το κάτω φράγμα της απόστασης Fréchet της καμπύλης εισόδου και των καμπυλών της βάσης δεδομένων. Συνεπώς, αν ψάχνουμε καμπύλες με απόσταση Fréchet μικρότερη κάποια τιμής δ , μπορούμε πολύ εύκολα να αγνοήσουμε καμπύλες με κάτω φράγμα μεγαλύτερο του δ και έτσι να περιορίσουμε την χρονική πολυπλοκότητα του προβλήματος. Η εύρεση του κάτω φράγματος είναι πιο αποδοτική σε χρόνο από τον ακριβή υπολογισμό της απόστασης Fréchet.

Θα γράψουμε τώρα μια συνάρτηση η οποία να υπολογίζει αυτό το φράγμα για δυο καμπύλες. Τα πράγματα είναι αρκετά απλά. Το μόνο που έχουμε να κάνουμε είναι να υπολογίσουμε δυο νόρμες και να βρούμε τις ελάχιστες και μέγιστες τιμές για κάθε συντεταγμένη:

```

1 def find_lower_bound(P, Q):
2     P = np.array(P)
3     Q = np.array(Q)
4
5     # Start and end point distances
6     d_start = np.linalg.norm(P[0] - Q[0])
7     d_end = np.linalg.norm(P[-1] - Q[-1])
8
9     # Min/max differences in x and y. Numpy makes things simple
10    min_x_diff = abs(np.min(P[:, 0]) - np.min(Q[:, 0]))
11    max_x_diff = abs(np.max(P[:, 0]) - np.max(Q[:, 0]))
12    min_y_diff = abs(np.min(P[:, 1]) - np.min(Q[:, 1]))
13    max_y_diff = abs(np.max(P[:, 1]) - np.max(Q[:, 1]))
14
15    # Return the max values of all the possible boundaries
16    return max(d_start, d_end, min_x_diff, max_x_diff,
               min_y_diff, max_y_diff)

```

Listing 2.2: This function calculates the lower bound for the Fréchet distance

Προσέξτε την οικονομία που προσφέρει το πακέτο Numpy. Επιπλέον, πίσω της κρύβεται διανυσματική C η οποία προσφέρει καλύτερη ανταπόκριση σε σχέση με μια επανάληψη for της python που ενδεχομένως να είχαμε χρησιμοποιήσει για την γραμμική σάρωση όλων των x και y συντεταγμένων. Ας δούμε το παράδειγμα με τις δυο πρώτες καμπύλες της βάσης δεδομένων:

Lower Bound is:

0.009909227

Frechet distance: 0.010398

Το αποτέλεσμα φαίνεται σωστό αφού το κάτω φράγμα είναι πράγματι μικρότερο της πραγματικής απόστασης Fréchet. Το επόμενο βήμα είναι δεδομένης μιας καμπύλης εισόδου να επιλέξουμε από το συνολικό πλήθος της βάσης δεδομένων τις καμπύλες εκείνες για τις οποίες το κάτω όριο της απόστασης Fréchet με την καμπύλη εισόδου είναι μικρότερο από την τιμή ανοχής, αφού μόνο αυτές οι καμπύλες έχουν πιθανότητα να είναι κοντινές μαζί της. Αν ονομάσουμε D το σύνολο της βάσης δεδομένων και P την καμπύλη εισόδου, τότε εμείς ψάχνουμε ένα σύνολο D' για το οποίο να ισχύει πως:

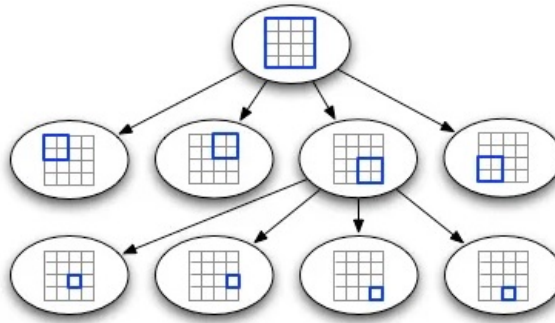
$$D' = \{k \in D | LB_F(P, k) \leq \delta\} \quad (2.3)$$

Για τις υπόλοιπες καμπύλες του συνόλου $D \setminus D'$ για τις οποίες δεν ισχύει η συνθήκη έχουμε πως $d_F(P, k) \geq LB_F(P, k) > \delta$ και άρα δεν υπάρχει περίπτωση να ανήκουν στο ζητούμενο χώρο.

Το επόμενο βήμα είναι υπολογίσουμε το σύνολο D' . Για τον σκοπό αυτό θα χρησιμοποιήσουμε ένα τετραγωνικό δέντρο σε 8 διαστάσεις ως δομή δεδομένων. Είναι το αντίστοιχο της δυαδικής αναζήτησης με την οποία υπολογίσαμε την απόσταση Fréchet βάση των κρίσιμων τιμών για το ε . Όπως είδαμε πριν λίγο, οι τιμές που χρειαζόμαστε για κάθε καμπύλη ώστε να υπολογίσουμε το κάτω φράγμα της απόστασης Fréchet είναι 8, οι 4 συντεταγμένες των αρχικών και τελικών σημείων (2 για κάθε σημείο αφού βρισκόμαστε σε διδιάστατο χώρο) καθώς και οι 4 ακραίες τιμές κατά τους δυο άξονες. Το κάτω φράγμα υπολογίζεται μέσω πράξεων σε αυτές τις τιμές. Αποθηκεύουμε τα παραπάνω σε ένα διάνυσμα 8 διαστάσεων και δημιουργούμε μια δομή δεδομένων ανάλογη του τετραγωνικού δέντρου σε 8 διαστάσεις.

Το τετραγωνικό δέντρο είναι μια δομή δεδομένων που χωρίζει αναδρομικά έναν διδιάστατο χώρο σε τέσσερα ίσα τεταρτημόρια ή περιοχές. Χρησιμοποιείται για την αναπαράσταση και αποθήκευση χωρικών δεδομένων όπως σημεία, γραμμές και πολύγωνα με πιο αποτελεσματικό τρόπο από τη χρήση μιας απλής λίστας

ή ενός πίνακα. Κάθε κόμβος του δέντρου έχει το πολύ τέσσερα παιδιά. Η ρίζα αναπαριστά ολόκληρο τον δισδιάστατο χώρο, ο οποίος χωρίζεται σε τέσσερα τεταρτημόρια. Κάθε τεταρτημόριο αντιπροσωπεύεται από έναν κόμβο-παιδί της ρίζας. Οι κόμβοι-παιδιά μπορούν επίσης να διαιρεθούν περαιτέρω σε τέσσερα τεταρτημόρια εάν περιέχουν περισσότερα από ένα σημεία. Η διαδικασία αυτή συνεχίζεται αναδρομικά μέχρι όλα τα σημεία να περιέχονται σε μεμονωμένα φύλλα του δέντρου. Μια εικόνα που δείχνει σχηματικά την διαδικασία είναι η παρακάτω:



Σχήμα 2.2: Η δομή ενός τετραγωνικού δέντρου

Αρκεί τώρα να εφαρμόσουμε την παραπάνω λογική για τα διανύσματα 8 διαστάσεων που μας ενδιαφέρουν εδώ. Η κατασκευή ενός τετραγωνικού δέντρου δεν είναι και η πιο απλή υπόθεση για αυτό θα αναλύσουμε προσεκτικά κάθε σημείο του κώδικα. Ξεκινάμε ορίζοντας μια κλάση:

```

1 class QuadTreeNode:
2     def __init__(self, min_coords, max_coords, max_elements=4):
3         self.min_coords = np.array(min_coords) # Bounding box
4             limits
5         self.max_coords = np.array(max_coords)
6         self.max_elements = max_elements # Max contained elements
7         self.elements = [] # list of tuples (point, data)
8         self.children = None

```

Listing 2.3: Create the Quadtree class and its constructor

Ορίζουμε την κλάση QuadTreeNode. Κάθε στιγμιότυπο της κλάσης αυτής αντιπροσωπεύει ένα κόμβο σε ένα δέντρο 8 διαστάσεων. Ο constructor της κλάσης, δέχεται ως παραμέτρους τις συντεταγμένες του ελάχιστου σημείου και του μέγιστου σημείου του συνόρου, καθώς και το πόσα σημεία μπορεί να περιέχει πριν διαχωριστεί εκ νέου. Για το τετραγωνικό δέντρο στις δυο διαστάσεις, τα ελάχιστα και μέγιστα σημεία είναι αυτά στο κάτω αριστερό άκρο των συνόρων του κουτιού και στο πάνω δεξί άκρο. Τα σημεία αυτά ορίζουν τα σύνορα του κόμβου στον χώρο και καθορίζουν το ποιά σημεία περιέχονται εντός τους. Αποθηκεύουμε λοιπόν, τα άκρα

αυτά σε δυο τοπικές μεταβλητές ως σημεία στο χώρο μέσω του numpy. Στη συνέχεια, ορίζουμε τις μεταβλητές `elements` και `children`. Στην `elements` θα αποθηκεύουμε τις καμπύλες που περιέχει ο κόμβος μαζί με το `id` τους στη βάση δεδομένων, ενώ στην `children` τα παιδιά του κόμβου εφόσον αυτός πρέπει να διαχωριστεί.

```

1 def insert(self, point, data):
2     point = np.array(point)
3     # If we have children, delegate insertion to appropriate
       child
4     if self.children is not None:
5         child = self._get_child(point)
6         child.insert(point, data)
7         return
8
9     # Otherwise, store element here
10    self.elements.append((point, data))
11
12    # Split if needed
13    if len(self.elements) > self.max_elements:
14        self._split()

```

Listing 2.4: Insert method used to add elements to the node

Αρχικά φροντίζουμε ώστε το σημείο να είναι σε μορφή πίνακα numpy ώστε να μπορούμε στη συνέχεια να εκτελέσουμε διανυσματικές πράξεις. Στη συνέχεια ελέγχουμε αν ο κόμβος έχει παιδιά. Αν πράγματι έχει παιδιά, βρίσκουμε ποιο παιδί εμπεριέχει το σημείο και στη συνέχεια περνάμε την μέθοδο στο αυτό το παιδί. Εδώ γίνεται φανερό η αναδρομή της όλης διαδικασίας. Αν κάποιος κόμβος περιέχει παιδιά, αυτός δεν κρατά δεδομένα για τα σημεία. Οι μόνοι κόμβοι που θα έχουν τα σημεία είναι οι ρίζες του δέντρου. Συνεπώς, ο κώδικας προχωρά παρακάτω όταν συναντήσει κάποιον κόμβο χωρίς παιδιά οπότε και θα αποθηκεύσει το ζεύγος 8 συντεταγμένων και του `id` τους. Ωστόσο, σε κάθε κόμβο επιτρέπουμε να έχει το πολύ 4 σημεία και έτσι αν ξεπεράσουμε τον αριθμό αυτό, ο κόμβος πρέπει να διαχωριστεί εκ νέου σε υποδιαίρεσεις.

Συνεχίζουμε στην επόμενη μέθοδο η οποία χειρίζεται τώρα τον διαχωρισμό ενός κόμβου σε υποσύνολα:

```

1 def _split(self):
2     mid = (self.min_coords + self.max_coords) / 2
3     dims = len(self.min_coords)
4
5     # Create children nodes (2^dims)
6     self.children = []
7     for i in range(2 ** dims):
8         # The starting coordinates of each child are the fathers
9         min_c = self.min_coords.copy()
10        max_c = self.max_coords.copy()

```

```

11         for d in range(dims):
12             if (i >> d) & 1: # Binary right shift and
13                 bitwise AND
14                 min_c[d] = mid[d]
15             else:
16                 max_c[d] = mid[d]
17             self.children.append(QuadtreeNode(min_c, max_c,
18                 self.max_elements))
19
20         # Move elements to children
21         for point, data in self.elements:
22             child = self._get_child(point)
23             child.insert(point, data)
24
25         # Clear current node's elements
26         self.elements = []

```

Listing 2.5: Split method is responsible for creating the children of a node

Πρώτη δουλειά η εύρεση του μέσου σημείου του χώρου που καταλαμβάνει ο κόμβος. Επίσης βρίσκουμε και τις διαστάσεις, 8 στην περίπτωσή μας. Στη συνέχεια χωρίζουμε το κουτί μας σε ίσα τμήματα. Επειδή σε χώρο 8 διαστάσεων δεν έχουμε εικόνα θα βασιστούμε στις 2 διαστάσεις. Στην περίπτωση αυτή, όπως βλέπουμε και στο σχήμα 2.2 ο χώρος χωρίζεται σε 4 ίσα τεταρτημόρια. Ο τρόπος να συμβεί αυτό είναι να χωρίσουμε για κάθε διάσταση το χώρο στη μέση, ακριβώς όπως στη δυαδική αναζήτηση χωρίζουμε κάθε φορά τον πίνακα στα 2. Συνεπώς, χωρίζουμε την x συντεταγμένη στα 2 και την y συντεταγμένη στα δυο και προκύπτουν έτσι 2^2 ίσα τεταρτημόρια. Με τον ίδιο τρόπο, χωρίζοντας κάθε μια από τις 8 συνιστώσες στα δυο, προκύπτουν 2^8 ίσα τμήματα τα οποία θα αποτελούν τα παιδιά του αρχικού κόμβου. Αυτός είναι και ο λόγος που βλέπουμε την εντολή `for` να κινείται από το 0 έως τις διαστάσεις στο τετράγωνο.

Στη συνέχεια, εκτελούμε μια νέα επανάληψη `for` στο πλήθος των διαστάσεων και ελέγχουμε τα εξής: Για κάθε διάσταση d ελέγχουμε το d -οστό ψηφίο του i στη δυαδική αναπαράσταση. Αυτό γίνεται μετατοπίζοντας όλο την δυαδική αναπαράσταση d ψηφία δεξιά και εξάγοντας το τέρμα δεξιά ψηφίο, το λιγότερο σημαντικό μέσω του bitwise AND. Αν το ψηφίο αυτό είναι το 1 τότε το παιδί βρίσκεται στο πάνω μισό της d διάστασης, αλλιώς στο κάτω μισό. Έτσι, για κάθε παιδί θα προκύψει ένας μοναδικός συνδυασμός από πάνω και κάτω θέσεις προς όλες τις διαστάσεις και άρα μια μοναδική θέση στο χώρο. Κατά αυτόν τον τρόπο ο διαμοιρασμός του όγκου του γονέα θα έχει γίνει σε ισόποσα ανεξάρτητα τμήματα.

Συνεπώς, ανάλογα με το αποτέλεσμα του `if`, αν στη συγκεκριμένη διάσταση βρισκόμαστε στο πάνω μέρος το μέσο σημείου του γονέα θα γίνει το κάτω άκρο και το αντίστροφο. Για κάθε ένα από τα παιδιά δημιουργούμε ένα νέο στιγμιότυπο

της κλάσης. Επιπλέον, περνάμε τα όποια σημεία κρατάει ο γονέας στα παιδιά τους. Κάθε σημείο θα διαμοιραστεί στο παιδί στο οποίο το περιέχει. Αυτό γίνεται με την παρακάτω μέθοδο:

```

1 def _get_child(self, point):
2     # Mid point in each direction
3     mid = (self.min_coords + self.max_coords) / 2
4     index = 0
5     for d in range(len(point)): # Check all dimensions
6         if point[d] >= mid[d]: # Whether the point is on top
7             # half or bottom half
8             index |= (1 << d) # If top half d bit is 1,
9                                 # otherwise 0
10    return self.children[index] # Index of the child it
11    belongs to

```

Listing 2.6: Get child method decides in which child node a point belongs to

Η παραπάνω μέθοδος δέχεται (πέρα από τις τοπικές μεταβλητές) ένα σημείο και μας λέει σε πιο παιδί ανήκει το σημείο που περιλαμβάνεται έως τώρα στο γονέα. Βρίσκει αρχικά το μέσο σημείο σε κάθε διάσταση. Στη συνέχεια, ψάχνει για κάθε διάσταση αν το σημείο βρίσκεται στο πάνω ή στο κάτω μισό. Ανάλογα με την απάντηση στο ερώτημα αυτό, το δυαδικό ψηφίο του `index` στην d θέση γίνεται 0 ή 1. Στην ουσία πρόκειται για την αντίστροφη διαδικασία με παραπάνω όπου εξάγαμε το ψηφίο στη θέση d πάλι με δυαδικές μετατοπίσεις. Η διαδικασία αυτή θα δώσει ένα μοναδικό παιδί το οποίο θα περιέχει το σημείο. Επιστρέφουμε το `index` αυτού του παιδιού.

Οι παραπάνω μέθοδοι έχουν ολοκληρώσει την κατασκευή της δομής δεδομένων και απομένει τώρα να κάνουμε αναζητήσεις σε αυτήν. Για τον σκοπό αυτό γράφουμε την παρακάτω μέθοδο:

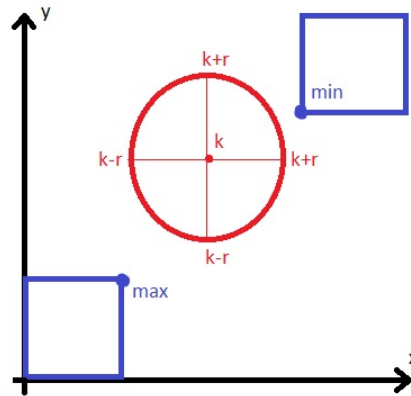
```

1 def _query_recursive(self, center, radius, results):
2     # Check if bounding box intersects the hypersphere
3     if np.any(self.min_coords > center + radius) or
4        np.any(self.max_coords < center - radius):
5         return
6     # Leaf node: check all elements
7     if self.children is None:
8         for point, data in self.elements:
9             if np.linalg.norm(point - center) <= radius:
10                results.append(data)
11        return
12    # Otherwise recurse into children
13    for child in self.children:
14        child._query_recursive(center, radius, results)

```

Listing 2.7: Method that finds all elements included in a hypersphere around a query

Έστω ότι έχουμε κάποιο σημείο γύρω από το οποίο θέλουμε να υπολογίσουμε πόσα σημεία υπάρχουν εντός μιας υπερσφαίρας κάποιας ακτίνας. Ελέγχουμε αρχικά τον όγκο του τρέχοντος κόμβου αν έχει τομή με την υπερσφαίρα. Αυτό το επιτυγχάνουμε συγκρίνοντας τις ελάχιστες συντεταγμένες προς όλες τις διαστάσεις με το κέντρο του σημείου εξέτασης συν την ακτίνα ή τις μέγιστες συντεταγμένες προς όλες τις διαστάσεις με το κέντρο του σημείου μείων την ακτίνα. Το επόμενο σχήμα θα μας δώσει μια διαίσθηση για τα παραπάνω στις δυο διαστάσεις:



Σχήμα 2.3: Τα σημεία που συγκρίνουμε ώστε να αποφασίσουμε σε ποιούς κόμβους πρέπει να ψάξουμε.

Αν το πρώτο `if` αποτύχει λοιπόν, τότε επιστρέφουμε αφού ο τρέχον κόμβος δεν έχει κοινό τόπο με την υπερσφαίρα και άρα δεν υπάρχει καμία περίπτωση σημεία του να περιλαμβάνονται στην επιθυμητή αναζήτηση. Η επόμενη `if` ελέγχει αν ο κόμβος έχει παιδιά. Αν όχι, τότε πρόκειται για φύλλο του δέντρου και άρα περιέχει κάποια σημεία. Ελέγχουμε ποιά από αυτά τα σημεία ανήκουν εντός της υπερσφαίρας μέσω της Ευκλείδειας απόστασης και αποθηκεύουμε το σημείο στα αποτελέσματα. Σε διαφορετική περίπτωση, περνάμε την αναζήτηση στα παιδιά του κόμβου. Τέλος, καλούμε την παραπάνω μέθοδο, η οποία ψάχνει σε κόμβους, εντός μιας συγκεκριμένης μεθόδου που επιστρέφει όλα τα σημεία για όλους τους κόμβους:

```

1 def query(self, center, radius):
2     results = []
3     self._query_recursive(np.array(center), radius, results)
4     return results

```

Listing 2.8: This method will return all nearby points

Η παραπάνω κλάση δημιουργεί τους κόμβους του δέντρου λοιπόν, αλλά λείπουν της λείπουν κάποια στοιχεία για να λειτουργήσει, όπως για παράδειγμα τα στοιχεία της ρίζας. Για τον σκοπό αυτό, φτιάχνουμε μια βοηθητική κλάση η οποία να υπολογίζει τα απαραίτητα. Η κλάση αυτή θα παίρνει όλα τα αρχικά διανύσματα

(ένα διάνυσμα 8 διαστάσεων από κάθε καμπύλη της βάσης δεδομένων) και θα εκκινεί την κατασκευή του δέντρου. Ας δούμε το πρώτο τμήμα:

```

1 class Quadtree:
2     def __init__(self, points, max_elements=4):
3
4         if not points:
5             raise ValueError("Points list is empty!")
6
7         # Calculate global min/max coords over all points
8         all_points = np.array([p for p, _ in points])
9         min_coords = np.min(all_points, axis=0)
10        max_coords = np.max(all_points, axis=0)

```

Listing 2.9: This class will act as a wrapper of the QuadtreeNode class

Αφού δώσουμε ως είσοδο τα διανύσματα και το μέγιστο πλήθος σημείων για τις ρίζες, υπολογίζουμε τις ελάχιστες και μέγιστες τιμές προς όλες τις διαστάσεις. Οι τιμές αυτές θα μας δώσουν τα σύνορα του αρχικού κουτιού στον χώρο των 8 διαστάσεων. Μένει μόνο να δημιουργήσουμε το αρχικό στιγμιότυπο της κλάσης QuadtreeNode που θα είναι και η ρίζα του δέντρου. Η λογική εντός της κλάσης θα αναλάβει την δημιουργία του δέντρου:

```

1         self.root = QuadtreeNode(min_coords, max_coords,
2                                   max_elements)
3         for point, data in points:
4             self.root.insert(point, data)
5
6     def insert(self, point, data):
7         self.root.insert(point, data)
8
9     def query(self, center, radius):
10        return self.root.query(center, radius)

```

Listing 2.10: Call QuadtreeNode to start creating the quadtree

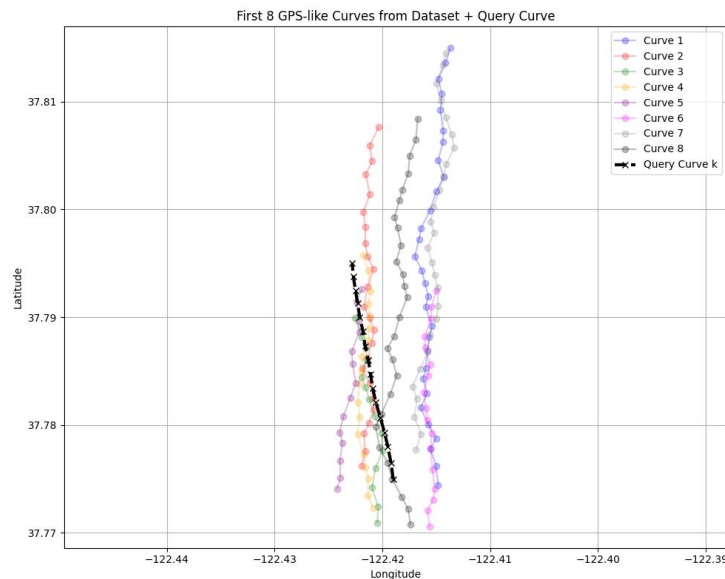
Οι δυο μέθοδοι που έχουμε δημιουργήσει εδώ τις δανειζόμαστε από την QuadtreeNode και χρησιμεύουν στο να εκινήσει η διαδικασία. Γιατί όμως είναι χρήσιμη η δημιουργία αυτής της δομής δεδομένων; Θα το δούμε αυτό ευθύς αμέσως.

2.3.2 Δοκιμή του κάτω φράγματος

Έως τώρα έχουμε δει αναλυτικά την βάση δεδομένων, καθώς και την μορφή που έχει, μια συλλογή από καμπύλες που προέχρονται από δεδομένα GPS. Ωστόσο, η αναζήτηση επί των δεδομένων γίνεται όπως έχουμε πει σε σχέση με μια καμπύλη

εισόδου και μια τιμή παραμέτρου δ . Συνεπώς, για την δοκιμή των παραπάνω κλάσεων αλλά και για τη συνέχεια του κώδικα, θα χρησιμοποιήσουμε μια τυχαία καμπύλη η οποία να συμπεριφέρεται ως είσοδος για την αναζήτηση. Η καμπύλη αυτή βέβαια θα έχει τα ίδια χαρακτηριστικά με τις προηγούμενες και θα ανταποκρίνεται σε κάποια διαδρομή στο κέντρο μιας πόλης. Μπορούμε είτε να επιλέξουμε μια καμπύλη από τα δεδομένα και να την χρησιμοποιήσουμε, είτε να πάρουμε μια άγνωστη καμπύλη μέσω τυχαίων τιμών εντός του εύρους των δεδομένων. Επιλέγουμε το δεύτερο σενάριο ώστε να μην αλλοιώσουμε τα δεδομένα.

Ο πιο εύκολος τρόπος να ελέγξουμε την μορφή που θα έχει η καμπύλη εισόδου είναι να την ζωγραφίσουμε. Επιλέγουμε να αποθηκεύσουμε την καμπύλη αυτή σε ένα csv αρχείο παρόμοιας μορφής με τα δεδομένα. Έτσι, η συνάρτηση εξαγωγής των στοιχείων της θα είναι η ίδια με αυτή που χρησιμοποιήσαμε για τα δεδομένα. Επιπλέον, μπορούμε στο αρχείο αυτό να προσθέσουμε όλες καμπύλες θέλουμε να ψάξουμε στα δεδομένα, εκτελώντας μάλιστα παράλληλα τους υπολογισμούς για κάθε καμπύλη εισόδου. Αυτά όμως στη συνέχεια. Εμείς τώρα εισάγουμε στο νέο αυτό αρχείο που το ονομάζουμε `query-database.csv`, την καμπύλη βάση της οποίας θα ψάξουμε. Εξάγουμε τα σημεία της και την ζωγραφίζουμε στο ίδιο διάγραμμα που είδαμε παραπάνω στο σχήμα 2.1:



Σχήμα 2.4: Βλέπουμε την καμπύλη βάση της οποίας θα τρέξουμε την αναζήτηση σε σχέση με τις 8 πρώτες καμπύλες της βάσης δεδομένων

Πρέπει επίσης να θεωρήσουμε μια τιμή για το δ . Θεωρούμε $\delta = 0.011$ ώστε να συμπεριλάβουμε αρκετές καμπύλες. Ωραία, τώρα ας χρησιμοποιήσουμε τις κλάσεις που είδαμε παραπάνω ώστε πρώτα να δημιουργήσουμε το τετραγωνικό δέντρο και

στη συνέχεια να ψάξουμε για ποιές από τις καμπύλες των δεδομένων έχουμε κάτω φράγμα για την απόσταση Fréchet μικρότερο του δ .

Χρειαζόμαστε μια επιπλέον συνάρτηση η οποία να εξάγει τα διανύσματα 8 διαστάσεων από τις καμπύλες των δεδομένων και την καμπύλη αναζήτησης, μέσω των οποίων υπολογίζουμε το κάτω άκρο σε κάθε περίπτωση. Η παρακάτω συνάρτηση κάνει την δουλειά αυτή:

```

1 def compute_8d_features(curve):
2     curve = np.array(curve)
3     return np.array([
4         curve[0][0],      # start x
5         curve[0][1],      # start y
6         curve[-1][0],     # end x
7         curve[-1][1],     # end y
8         np.min(curve[:, 0]), # min x
9         np.max(curve[:, 0]), # max x
10        np.min(curve[:, 1]), # min y
11        np.max(curve[:, 1])  # max y
12    ])

```

Listing 2.11: Extract the necessary data from each curve

Τα σχόλια εξηγούν ακριβώς τι βλέπουμε και άρα επιπλέον σχόλια είναι περιττά. Στη συνέχεια, δημιουργούμε τη δομή δεδομένων του τετραγωνικού δέντρου αλλά σε 8 διαστάσεις καλώντας τις κλάσεις που είδαμε:

```

1 dataset = []
2 for idx in range(100):
3     curve = extract_curve("curve_dataset.csv", idx)
4     features = compute_8d_features(curve)
5     dataset.append((features, idx))
6
7 tree = Quadtree(dataset, max_elements=4)

```

Listing 2.12: Create the quadtree structure

Σαρώνουμε όλες τις καμπύλες της βάσης δεδομένων και εξάγουμε τα 8 χαρακτηριστικά που μας ενδιαφέρουν. Αποθηκεύουμε αυτά τα χαρακτηριστικά σε μια λίστα την οποία συνέχεια εισάγουμε στην κλάση μαζί με τον μέγιστο αριθμό σημείων για τις ρίζες. Θα μπορούσε ο μέγιστος αριθμός να είναι η μονάδα, αλλά λόγω των 8 διαστάσεων αυτό θα εισάγει πάρα πολλούς επιπλέον κόμβους οι οποίοι δεν θα συνεισφέρουν αρκετά στην ταχύτητα ώστε να δικαιολογήσουν την υπαρξή τους.

Τρέχουμε τώρα την αναζήτηση με την τιμή του δ που ορίσαμε παραπάνω:

```

1 k = extract_curve("query_dataset.csv", 0)
2 k_features = compute_8d_features(k)

```

```

3 |
4 | delta = 0.011
5 |
6 | candidates = tree.query(k_features, delta)
7 | print("Candidates:", candidates)
8 | print("Total number of candidates:" , len(candidates))
9 |
10 | candidate_curves = [extract_curve("curve_dataset.csv", idx) for
    | idx in candidates]

```

Listing 2.13: Run the query to find candidate curves

Εξάγουμε τα χαρακτηριστικά της καμπύλης αναζήτησης και αυτό το διάνυσμα 8 διαστάσεων είναι το διάνυσμα που δίνουμε ως είσοδο στην μέθοδο query της κλάσης Quadtree. Παρατηρήστε πως εξάγονται οι υποψήφιες καμπύλες. Με την παραδοσιακή μέθοδο, θα έπρεπε να τρέξουμε ένα for σε όλες τις καμπύλες της βάσης δεδομένων και να βρούμε μέσω της συνάρτησης find-lower-bound το κάτω όριο της απόστασης Fréchet για κάθε ζεύγος καμπύλης εισόδου και καμπύλης δεδομένων προτού συγκρίνουμε με το δ .

Από την άλλη, με την χρήση της δομής δεδομένων, έχουμε τρέξει μια αναζήτηση βάση των γεωμετρικών χαρακτηριστικών των δεδομένων, σύμφωνα με την εικόνα του σχήματος 2.3. Έτσι, δημιουργήσαμε μια υπερσφαίρα γύρω από το σημείο αναζήτησης και ψάξαμε την δομή δεδομένων για το ποιοί κόμβοι έχουν επικάλυψη με την σφαίρα και στη συνέχεια τα σημεία μόνο αυτών των κόμβων. Γλιτώσαμε έτσι πολλές επιπλέον επαναλήψεις σε όλα τα δεδομένα. Έχουμε βέβαια πληρώσει ένα κόστος για την κατασκευή της δομής, αλλά αυτό πρέπει να γίνει μόνο μια φορά. Συνεπώς, για όλο και περισσότερα δεδομένα, η δομή δεδομένων προσφέρει ταχύτατες αναζητήσεις σε σχέση με μια επανάληψη for.

Η έξοδος τώρα του παραπάνω κώδικα είναι:

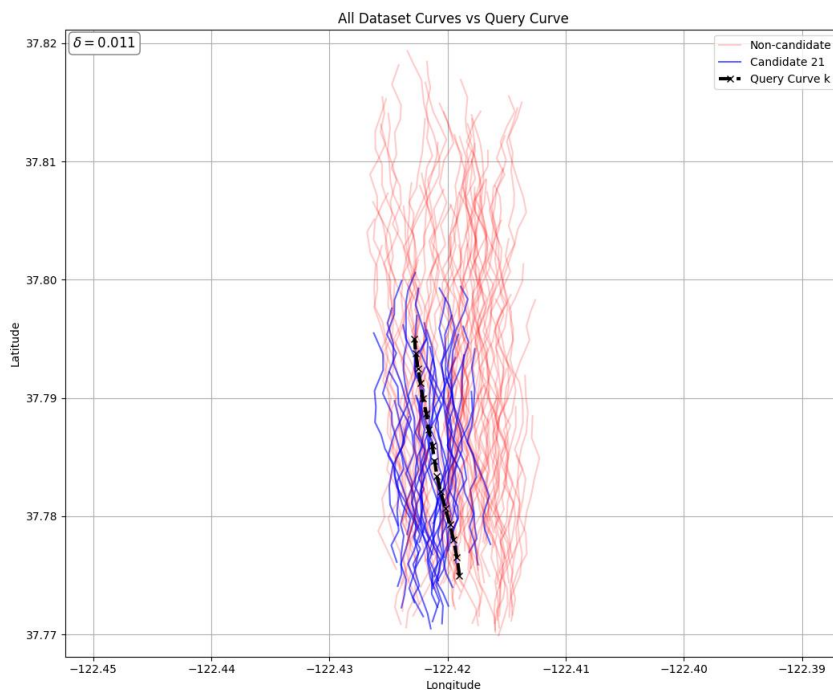
```

Candidates: [21, 4, 41, 13, 2, 75, 3,
37, 91, 46, 83, 90, 62, 66, 84, 35, 40,
57, 70, 48, 20, 15, 79, 80, 67, 98, 43,
14, 77]
Total number of candidates: 29

```

Πρόκειται για τα index κάθε καμπύλης, δηλαδή την γραμμή που εμφανίζεται στο αρχείο με τα δεδομένα. Αμέσως λοιπόν, έχουμε περιορίσει στο ένα τρίτο το πλήθος των δεδομένων που χρήζουν αντιμετώπισης. Μπορούμε να περιορίσουμε την αναζήτησή μας για τις κοντινές καμπύλες μόνο στις παραπάνω.

Είναι όμως τα παραπάνω αποτελέσματα κοντά στην αλήθεια. Ο μόνος τρόπος να το κρίνουμε αυτό είναι να σχεδιάσουμε τις καμπύλες σε ένα γράφημα και να ελέγξουμε αν πράγματι οι υποψήφιες καμπύλες βρίσκονται κοντά στην καμπύλη αναζήτησης. Το γράφημα που προκύπτει είναι το παρακάτω:



Σχήμα 2.5: Η καμπύλη αναζήτησης σε σύγκριση με τις καμπύλες της βάσης δεδομένων. Η καμπύλη αναζήτησης φαίνεται με το μαύρο, ενώ με μπλε χρώμα είναι οι υποψήφιες καμπύλες για $\delta = 0.011$ και με κόκκινο χρώμα οι καμπύλες που δεν περνούν τον έλεγχο.

Με μαύρο χρώμα η καμπύλη αναζήτησης, με μπλε οι υποψήφιες καμπύλες και με κόκκινο όλες οι υπόλοιπες καμπύλες της βάσης δεδομένων. Παρατηρούμε πως πράγματι, οι υποψήφιες καμπύλες είναι αυτές που βρίσκονται πιο κοντά σε σχέση με την καμπύλη αναζήτησης και έχουν μάλιστα παρόμοιο μήκος. Αυτό το χαρακτηριστικό επιβεβαιώνει και η διαίσθησή μας αφού καμπύλες με μεγαλύτερο μήκος έχουν πιο απομακρυσμένα σημεία από την καμπύλη αναζήτησης.

2.3.3 Θετικό φίλτρο για την απόσταση Fréchet

Για την συνέχεια θα εφαρμόσουμε ένα επιπλέον φίλτρο στα δεδομένα. Να σημειωθεί πως από εδώ και πέρα η εφαρμογή των μεθόδων θα γίνεται μόνο στις υποψήφιες καμπύλες που έχουν προκύψει από την προηγούμενη ενότητα, αφού μόνο

αυτές έχουν πιθανότητα να ανταποκρίνονται στην αναζήτησή μας. Μπορούμε λοιπόν, μέσω μιας προσέγγισης να υπολογίσουμε ένα άνω φράγμα για την απόσταση Fréchet μεταξύ δυο καμπύλων.

Ας σκεφτούμε δυο καμπύλες, την καμπύλη αναζήτησης και μια υποψήφια. Ο άπληστος αλγόριθμος που υπολογίζει ένα άνω φράγμα είναι ο εξής: Ξεκινούμε από τα αρχικά σημεία κάθε καμπύλης k_1, q_1 και υπολογίζουμε την απόσταση μεταξύ τους. Αυτή η απόσταση αποτελεί ένα κάτω φράγμα όπως έχουμε δει. Στη συνέχεια, θα κινηθούμε πάνω στα σημεία και μόνο της κάθε καμπύλης. Αυξάνουμε κατά 1 είτε τον δείκτη του k είτε τον δείκτη του q είτε και τους δυο δείκτες. Άρα το επόμενο ζεύγος σημείων θα είναι κάποιο εκ των k_2, q_1 ή k_1, q_2 ή k_2, q_2 . Πως όμως αποφασίζουμε ποιο από τα 3 θα πραγματοποιηθεί; Επιλέγουμε εκείνο το βήμα για το οποίο η απόσταση μεταξύ του ζεύγους είναι η μικρότερη δυνατή. Αυτός είναι και ο λόγος που ονομάζεται άπληστος ο αλγόριθμος. Συνεχίζουμε έως ότου φτάσουμε στα τελευταία σημεία. Εννοείται πως λαμβάνουμε περιορισμούς ώστε να μην ξεφύγουμε εκτός ορίων των καμπυλών.

Ο κώδικας που υλοποιεί τον παραπάνω αλγόριθμό είναι σχετικά απλός σε σχέση με ότι έχουμε δει έως τώρα. Ας ξεκινήσουμε όπως πάντα ορίζοντας μια συνάρτηση:

```

1 def euclidean(p1, p2):
2     return np.linalg.norm(np.array(p1) - np.array(p2))
3
4 def greedy_frechet_upper_bound(P, Q):
5     pos_a = 0
6     pos_b = 0
7     distance = euclidean(P[-1], Q[-1]) # Start from end-point
        distance

```

Listing 2.14: Greedy algorithm for upper Fréchet distance boundary

Αρχικά ορίζουμε μια συνάρτηση για την Ευκλείδεια απόσταση γιατί έχουμε βαρεθεί να την γράφουμε συνέχεια χειροκίνητα. Στη συνέχεια η συνάρτηση που θα υπολογίσει άπληστα κάποιο άνω όριο της απόστασης Fréchet. Ορίζουμε τις δυο μεταβλητές που θα δείχνουν pos-a και pos-b στις οποίες αποθηκεύουμε τα σημεία που βρισκόμαστε πάνω στις καμπύλες. Επιπλέον, ορίζουμε την distance η οποία εμπεριέχει την απόσταση μεταξύ a και b. Εκτελούμε τώρα μια επανάληψη. Ο αλγόριθμος είναι άπληστος και άρα θα ψάξουμε όλες τις περιπτώσεις που ελαχιστοποιούν την απόσταση μεταξύ των δυο σημείων, έως ότου φτάσουμε στο τέλος. Η κατάλληλη επιλογή εδώ είναι μια εντολή while:

```

1 # Loop until we reach the end points
2 while pos_a + pos_b < len(P) + len(Q) - 2:
3     distance = max(distance, euclidean(P[pos_a], Q[pos_b]))
4

```

```

5      # If pos_a reach the end, only increase pos_b and via
      versa
6      if pos_a == len(P) - 1:
7          pos_b += 1
8      elif pos_b == len(Q) - 1:
9          pos_a += 1
10
11     # If we are not at the end, find minimal distance
12     else:
13         # Compute possible future distances (squared for
            efficiency)
14         dist_a = np.sum((np.array(P[pos_a + 1]) -
            np.array(Q[pos_b])) ** 2)
15         dist_b = np.sum((np.array(P[pos_a]) -
            np.array(Q[pos_b + 1])) ** 2)
16         dist_both = np.sum((np.array(P[pos_a + 1]) -
            np.array(Q[pos_b + 1])) ** 2)
17
18         # Take the step that minimizes the distance
19         if dist_a < dist_b and dist_a < dist_both:
20             pos_a += 1
21         elif dist_b < dist_both:
22             pos_b += 1
23         else:
24             pos_a += 1
25             pos_b += 1
26
27     return distance

```

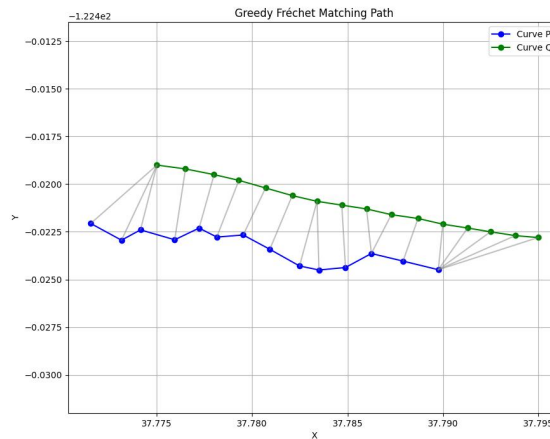
Listing 2.15: Main body of the function. Use a while to swipe all possible combos

Η συνθήκη της επανάληψης είναι το άθροισμα των δυο θέσεων να γίνει ίσο με το άθροισμα των δείκτων των τελικών σημείων. Τότε θα έχουμε φτάσει στο τέλος της διαδρομής. Η καινούργια απόσταση κάθε φορά είναι το μέγιστο της παλιάς και της απόστασης των νέων σημείων. Εφαρμόζουμε στη συνέχεια κάποιες συνθήκες ώστε να αποφύγουμε την έξοδο από τα άκρα των καμπυλών.

Συγκεκριμένα, αν κάποια από τις θέσεις pos-a και pos-b έχει φτάσει στο τέλος, αναγκαστικά αυξάνουμε μόνο την άλλη θέση. Σε κάποιο τυχαίο βήμα της πορείας λοιπόν, υπολογίζουμε τις δυνητικές αποστάσεις για κάθε ένα από τα 3 πιθανά βήματα και επιλέγουμε το βήμα εκείνο το οποίο ελαχιστοποιεί την απόσταση.

Για να κατανοήσουμε την διαδικασία, ένα γράφημα θα βοηθήσει. Έχουμε σχεδιάσει τα διάφορα βήματα του αλγορίθμου για την καμπύλη αναζήτησης και μια τυχαία καμπύλη από την δεξαμενή των υποψηφίων:

Το άνω φράγμα της απόστασης Fréchet είναι η μεγαλύτερη από όλες αυτές τις



Σχήμα 2.6: Ο άπληστος αλγόριθμος για τον καθορισμό ενός άνω ορίου για την απόσταση Fréchet

αποστάσεις. Όλες αυτές οι αποστάσεις στην ουσία προσπαθούν να υπολογίσουν ποιά θα ήταν η ελάχιστη απόσταση Fréchet που χρειάζεται κάθε σημείο της μιας καμπύλης ώστε να αντιστοιχηθεί με κάποιο σημείο της άλλης καμπύλης. Η μέγιστη από αυτές τις αποστάσεις θα αποτελεί και ένα άνω φράγμα για την απόσταση Fréchet αφού όλα τα υπόλοιπα σημεία μπορούν να αντιστοιχηθούν με μικρότερη απόσταση και άρα υπάρχει σύνоро στο διάγραμμα ελεύθερου χώρου.

Χρειάζεται βέβαια ιδιαίτερη προσοχή. Πρώτον, το παραπάνω φίλτρο δεν ελέγχει όλα τα δυνατά ταιριάσματα ενός σημείου με τα σημεία της άλλης ευθείας αλλά μόνο τοπικά. Δεύτερον, το φίλτρο έχει κάποια αξία μόνο για καμπύλες όπως και αυτές των δεδομένων, δηλαδή παρόμοιες με λίγες μικρές γωνίες και σχεδόν παράλληλες. Τρίτον, ο αλγόριθμος μπορεί να υπερεκτιμήσει το άνω φράγμα για την απόσταση Fréchet λόγω ακριβώς του ότι ψάχνει μόνο τοπικά ταιριάσματα των σημείων. Σε κάθε περίπτωση, στο προβλημά μας οι καμπύλες έχουν καλή συμπεριφορά και έτσι αν η μέγιστη απόσταση όλων αυτών των βημάτων είναι μεγαλύτερη από δ τότε μπορούμε με μεγάλη βεβαιότητα να απορίψουμε την καμπύλη για την οποία συμβαίνει κάτι τέτοιο. Για περισσότερες λεπτομέρειες σχετικά με αυτό το φίλτρο για το άνω φράγμα δείτε την αντίστοιχη εργασία της βιβλιογραφίας.

2.3.4 Δοκιμή του θετικού φίλτρου

Ας εφαρμόσουμε το φίλτρο για το άνω φράγμα στα δεδομένα μας με τον εξής τρόπο:

```

1 exact_distance_needed = []
2 for idx in candidates:
3     curve = extract_curve("curve_dataset.csv", idx)
4     upper_bound = greedy_frechet_upper_bound(k, curve)
5     if upper_bound > delta:
6         exact_distance_needed.append(idx)
7
8 print("After greedy check, curves remaining:")
9 print("Curves left to find exact distance" ,
10       exact_distance_needed)
11 print("Total number left after greedy:" ,
12       len(exact_distance_needed))

```

Listing 2.16: Apply the greedy upper boundary filter to candidate curves

Ορίζουμε ένα πίνακα `exact-distance-needed` ο οποίος θα αποθηκεύσει τις καμπύλες εκείνες για τις οποίες θα χρειαστεί ο αναλυτικός υπολογισμός της απόστασης Fréchet ώστε να είμαστε σίγουροι για τη σχέση της με το δ . Στη συνέχεια, για κάθε καμπύλη των υποψηφίων τρέχουμε τον άπληστο αλγόριθμο σε σχέση με την καμπύλη αναζήτησης και ελέγχουμε τη μέγιστη απόσταση με το δ . Αν η μέγιστη απόσταση είναι μεγαλύτερη του δ αυτό σημαίνει πως υπάρχει πιθανότητα η πραγματική απόσταση Fréchet να βρεθεί μικρότερη του δ και άρα πρέπει να τρέξουμε τον ακριβή υπολογισμό. Διαφορετικά, η απόσταση Fréchet είναι σίγουρα μικρότερη του δ καθώς το άνω όριο της είναι και αυτό μικρότερο του δ . Το ποιά είναι ακριβώς η απόσταση Fréchet δεν μας ενδιαφέρει. Αυτό που έχει σημασία για το πρόβλημά μας είναι να είναι μικρότερη του δ .

Το αποτέλεσμα του παραπάνω φίλτρου στις υποψήφιες καμπύλες είναι το παρακάτω. Τυπώνουμε και το άνω όριο που υπολογίσαμε σε κάθε περίπτωση ώστε να έχουμε μια καλύτερη εικόνα:

```

After greedy check, curves remaining:
Curves left to find exact distance []
Total number left after greedy: 0
Candidate 21: Upper bound = 0.005501
Candidate 4: Upper bound = 0.005297
Candidate 41: Upper bound = 0.005109
Candidate 13: Upper bound = 0.005610
.....

```

Λοιπόν, φαίνεται πως αυτό το φίλτρο για το άνω όριο της απόστασης Fréchet μας έχει λύσει κιόλας το πρόβλημα. Όλες οι υποψήφιες καμπύλες έχουν φιλτραριστεί και άρα δεν χρειάζεται να ελέγξουμε τίποτα επιπλέον. Σε κάθε περίπτωση,

είναι ένας απλός αλγόριθμος γραμμικού χρόνου (όσα τα σημεία της μιας συν της άλλης καμπύλης $O(n + m)$) ο οποίος μπορεί γρήγορα και αποτελεσματικά να μας γλιτώσει από κάποιους επιπλέον υπολογισμούς.

Ενδιαφέρον θα έχει να τρέξουμε τον παραπάνω αλγόριθμο σε όλες τις καμπύλες της βάσης δεδομένων και όχι μόνο στις υποψήφιες ώστε να δούμε αν θα καταλήξουμε περίπου στις ίδιες καμπύλες όπως με το κάτω φράγμα. Περιμένουμε ο αλγόριθμος να επιστρέψει όλες τις υπόλοιπες καμπύλες πλην όσες βγήκαν υποψήφιες από τον αλγόριθμο του κάτω φράγματος. Το μόνο που αλλάζει στον παραπάνω κώδικα είναι να σαρώσουμε όλη τη βάση και όχι μόνο τους candidates, δηλαδή τροποποιούμε την εντολή for σε: for idx in range(100).

After greedy check on all the database, curves remaining:

Curves left to find exact distance

[0, 1, 6, 7, 9, 11, 16, 17, 18, 22, 24,
25, 27, 28, 29, 30, 32, 33, 36, 38, 39,
42, 44, 47, 49, 51, 53, 54, 55, 60, 61,
63, 64, 65, 68, 71, 73, 76, 78, 81, 82,
85, 86, 87, 89, 92, 93, 94, 95, 96, 97]

Total number left after greedy: 51

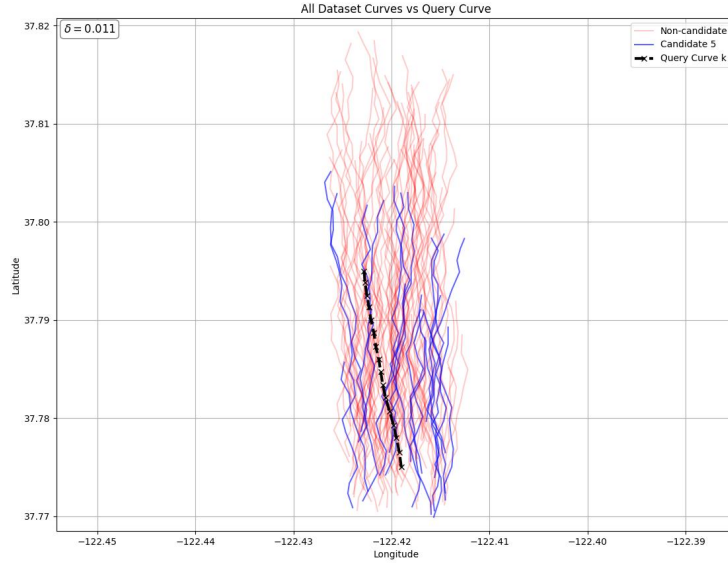
Παρατηρούμε εδώ ένα διαφορετικό αποτέλεσμα. Έχουμε επιστρέψει μόνο τις 51 καμπύλες για τις οποίες το άνω φράγμα τους είναι μεγαλύτερο του δ . Αυτό σημαίνει πως οι υπόλοιπες 49 καμπύλες έχουν άνω φράγμα μικρότερο του δ και άρα γίνονται αυτόματα δεκτές. Ωστόσο, το τετραγωνικό δέντρο έχει δώσει μόνο 29 υποψήφιες καμπύλες. Τι συμβαίνει λοιπόν εδώ; Η καλύτερη οδός είναι να εμπιστευτούμε το τετραγωνικό δέντρο. Ο άπληστος αλγόριθμος που είδαμε παραπάνω είναι προσεγγιστικός και υπάρχει σημαντική πιθανότητα να έχει εισάγει κάποιο σφάλμα στα αποτελέσματα.

Ποιές είναι όμως οι αποστάτες καμπύλες, δηλαδή οι 20 καμπύλες που σύμφωνα με τα παραπάνω έχουν περάσει τον άπληστο αλγόριθμο αλλά έχουν αποτύχει στο τετραγωνικό δέντρο. Με μια σύγκριση των 49 καμπύλων που πέρασαν τον άπληστο αλγόριθμο (αυτές που δεν αναγράφονται στην παραπάνω λίστα) και στις 29 υποψήφιες καμπύλες, οι 20 που διαφέρουν είναι οι:

strange_curves

[5, 8, 10, 12, 19, 23, 26, 31, 34, 45,
50, 52, 56, 58, 59, 69, 72, 74, 83, 99]

Ας τις ζωγραφίσουμε ώστε να έχουμε μια εικόνα για την συμπεριφορά τους:



Σχήμα 2.7: Με μπλε οι καμπύλες οι οποίες πέρασαν τον άπληστο αλγόριθμο αλλά απέτυχαν στο τετραγωνικό δέντρο.

Παρατηρούμε λοιπόν, πως αυτές οι 20 καμπύλες έχουν μια ενδιαμέση απόσταση από την καμπύλη αναζήτησης υπό την έννοια πως είτε έχουν μεσαίο μήκος και άρα τα τελικά τους σημεία βρίσκονται κάπως μακριά, είτε όλα τους τα σημεία βρίσκονται σε μια μέση απόσταση. Είμαστε λοιπόν πάνω στο όριο της απόστασης Fréchet σε σχέση με το δ . Εκεί λοιπόν φαίνεται πως ο άπληστος αλγόριθμος δυσκολεύεται.

2.3.5 Αρνητικό φίλτρο για την απόσταση Fréchet

Θεωρητικά έχουμε τελειώσει αφού ο άπληστος αλγόριθμος επιβεβαίωσε πως όλες οι υποψήφιες καμπύλες έχουν $d_F \leq \delta$ όπως είναι το ζητούμενο ίσως με κάποιο σφάλμα. Ωστόσο, μπορεί απλώς να είμαστε τυχεροί για την βάση δεδομένων και την καμπύλη που έχουμε επιλέξει για την αναζήτηση.

Για τον λόγο αυτό εισάγουμε ακόμη ένα φίλτρο για τις υποψήφιες καμπύλες του τετραγωνικού δέντρου με την εξή λογική: οι υποψήφιες καμπύλες έχουν κάτω όριο για την απόσταση Fréchet μικρότερο του δ . Η πραγματική απόσταση Fréchet λοιπόν, μπορεί να είναι μεγαλύτερη του δ . Μπορεί να βρίσκεται κοντά στο δ ή να είναι κατά πολύ μεγαλύτερη. Βέβαια για τις σχεδόν ρεαλιστικές καμπύλες με τις οποίες ασχολούμαστε δεν μπορεί να είναι κατά πολύ μεγαλύτερη του δ . Σε κάθε περίπτωση, φτιάχνουμε ένα αλγόριθμό ο οποίος να υπολογίζει αν δυο καμπύλες

βρίσκονται αρκετά μακριά σε όρους γεωμετρίας και έτσι η απόσταση Fréchet τους είναι μεγαλύτερη του ορίου.

Όπως και για τον άπληστο αλγόριθμο θα χρησιμοποιήσουμε μια προσέγγιση υπό την έννοια πως θα δουλέψουμε μόνο με τα σημεία της καμπύλης. Μια πιο ακριβή υλοποίηση θα ήταν να θεωρήσουμε και ενδιάμεσα σημεία αλλά το υπολογιστικό κόστος θα ανέβει πολύ. Η βασική λογική είναι η εξής: Επιλέγουμε ένα βήμα, συνήθως την ίδια την τιμή του δ και παίρνουμε δειγματοληπτικά σημεία στη μια καμπύλη. Από τα σημεία αυτά, προσπαθούμε να βρούμε το πρώτο σημείο της άλλης καμπύλης το οποίο βρίσκεται εντός απόστασης δ . Αν δεν υπάρχει τέτοιο σημείο, τότε οι καμπύλες δεν μπορούν να έχουν απόσταση Fréchet μικρότερη του δ . Επειδή η κατάσταση δεν είναι ακριβώς συμμετρική, τρέχουμε την ίδια λογική με τους ρόλους των καμπυλών ανάποδα.

Πάνω κάτω η λογική είναι η ίδια με τον άπληστο αλγόριθμο. Διατρέχουμε τα σημεία της μιας καμπύλης και υπολογίζουμε αποστάσεις. Ωστόσο, το άπληστος αλγόριθμος μπορεί να θεωρηθεί και ως θετικό φίλτρο γιατί τελικά αποδέχεται καμπύλες που ανήκουν στο όριο που έχουμε θέσει, ενώ ο παρόν αλγόριθμος ονομάζεται και αρνητικό φίλτρο γιατί τελικά θα απορρίπτει καμπύλες με μεγάλη απόσταση.

Ας ξεκινήσουμε όπως πάντα ορίζοντας μια συνάρτηση:

```

1 def curve_length(c, i, j):
2     #Compute the sum of Euclidean distances between c[i] to c[j].
3     c = np.array(c)
4     return np.sum(np.linalg.norm(c[i+1:j+1] - c[i:j], axis=1))

```

Listing 2.17: Support function to calculate the length of part of a curve

Η παραπάνω συνάρτηση δέχεται τα σημεία της καμπύλης και δυο δείκτες εξ αυτών και υπολογίζει το μήκος από το ένα σημείο στο άλλο πάνω όμως στην καμπύλη. Πρόκειται δηλαδή για ένα άθροισμα ευθύγραμμων τμημάτων. Διαισθητικά θα χρησιμοποιήσουμε το μήκος από το μονοπάτι αυτό για τον εξής λογω: Αν το ένα σημείο με δείκτη i της μιας καμπύλης βρίσκεται γεωμετρικά μακριά από το σημείο με δείκτη j της άλλης καμπύλης, τότε υπάρχει πιθανότητα το σημείο αυτό να ταιριάζει με κάποιο άλλο σημείο πιο πέρα στην καμπύλη με μικρότερη γεωμετρική απόσταση. Ωστόσο, λόγω της ρεαλιστικότητας των καμπυλών, το σημείο αυτό δεν μπορεί να βρίσκεται πολύ μακριά από το τμήμα της καμπύλης γύρω από το σημείο με δείκτη j . Θα δούμε την εφαρμογή της συνάρτησης παρακάτω.

Συνεχίζουμε με τον κορμό του αρνητικού φίλτρου:

```

1 def next_close_point(c, i, p, d):
2     #Greedy find the first point on curve c (starting at i)
       within distance d of p.

```

```

3   c = np.array(c)
4   delta_step = 1
5   k = i
6
7   while True:
8       # Check if we are at the end
9       if k == len(c) - 1:
10          return k if euclidean(c[k], p) <= d else len(c)
11      else:
12          # Make sure we dont go over the end when jumping delta
13          delta_step = min(delta_step, len(c) - 1 - k)
14          # Find shortest distance thap p could be from any
15          # point in k , k+step
16          if euclidean(p, c[k]) - curve_length(c, k, k +
17          delta_step) > d:
18              # Modify k with varying step
19              k += delta_step
20              delta_step *= 2
21          elif delta_step > 1:
22              delta_step //= 2
23          else:
24              return k

```

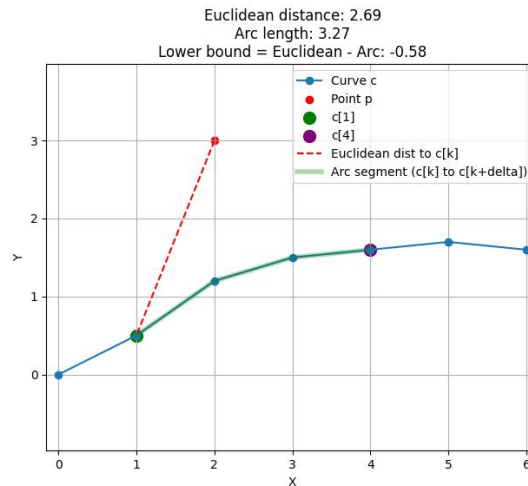
Listing 2.18: Main logic function. Find out the closest point within distance δ .

Η παραπάνω συνάρτηση στην ουσία βρίσκει το επόμενο σημείο της καμπύλης c με αφετηρία το δείκτη i , ώστε το σημείο αυτό να απέχει απόσταση d από ένα δεδομένο σημείο p . Αρχικά ορίζουμε το c ως ένα πίνακα numpy ώστε να εκτελέσουμε διανυσματικές πράξεις. Ορίζουμε delta_step το βήμα με το οποίο θα προχωρούμε στα σημεία και με την μεταβλητή k την αφετηρία. Στη συνέχεια, εκτελούμε συνεχώς την επανάληψη έως ότου καταλήξουμε σε κάποιο τελικό σημείο.

Πρώτα ελέγχουμε αν το έχουμε φτάσει στο τέλος της διαδρομής. Στην περίπτωση αυτή ελέγχουμε την απόσταση και είτε την επιστρέφουμε αν είναι μικρότερη του δ είτε επιστρέφουμε το πλήθος των σημείων σαν μήνυμα λάθους. Σε διαφορετική περίπτωση, συνεχίζουμε στο `else` όπου τώρα προσέχουμε ώστε το βήμα να μην μας οδηγή εκτός του τέλους της καμπύλης.

Στη συνέχεια ακολουθεί το ζουμί της υπόθεσης. Ψάχνουμε αν η Ευκλείδεια απόσταση μεταξύ του σταθερού σημείου p της μιας καμπύλης και του υπό εξέταση σημείου της άλλης καμπύλης μείον το μήκος του τόξου μεταξύ των σημείων k και $k + \text{delta_step}$ είναι μεγαλύτερο ή μικρότερο του δ . Η λογική εδώ είναι η εξής. Έστω πως η απόσταση μεταξύ των σημείων p και k είναι κάποια τιμή μεγαλύτερη του δ . Τότε, καθώς κινούμαστε προς το σημείο $k + 1$, το καλύτερο σενάριο από την άποψη της απόστασης είναι να πηγαίνουμε καρφί στο σημείο p . Αν όμως το μήκος τόξου μεταξύ k και $k + 1$ είναι αρκετά μικρό, τότε ακόμη και αν πηγαίνουμε καρφί προς το

p δεν θα πλησιάζουμε αρκετά σε απόσταση κάτω του δ και άρα η απόσταση Fréchet θα παραμείνει υψηλή. Αν όμως το μήκος τόξου είναι αρκετά μεγάλο, υπάρχει περίπτωση να πλησιάσω το σημείο πιο κοντά από δ και άρα πρέπει να συνεχίσω τον έλεγχο στο επόμενο σημείο της καμπύλης. Ένα γράφημα θα βοηθήσει στην κατανόηση:



Σχήμα 2.8: Όταν η διαφορά απόστασης μήκος τόξου βγει θετική συνεχίζω το έλεγχο με το επόμενο σημείο, αλλιώς μειώνω το βήμα ώστε να έχω μια πιο αναλυτική εικόνα.

Ανάλογα με το αποτέλεσμα της αφαίρεσης, αυξομειώνεται και το βήμα. Αν είναι μεγαλύτερη του δ τότε δεν υπάρχει πιθανότητα να φτάσω αρκετά κοντά από αυτό το σημείο εκκίνησης και άρα προχωρώ στο επόμενο σημείο. Ταυτόχρονα διπλασιάζω και το βήμα αφού πολύ πιθανόν να μην χρειάζεται να ελέγξω κάθε ένα σημείο ξεχωριστά. Ωστόσο, αν η αφαίρεση είναι μικρότερη του δ υπάρχει κάποιο πιο κοντινό σημείο και πρέπει να μειωθεί το βήμα ώστε να υπολογίσουμε κάποιο μικρότερο τόξο και να ξέρουμε ακριβώς ποιο είναι το σημείο που βρίσκεται πιο κοντά.

Ωρα να εφαρμόσουμε την παραπάνω συνάρτηση για όλα τα όλα τα σημεία της κάθε ευθείας και από τις δυο πλευρές:

```

1 def neg_filter(c1, c2, d):
2     #Negative filter: returns True if curves can be rejected.
3     c1 = np.array(c1)
4     c2 = np.array(c2)
5
6     delta_step = max(len(c1), len(c2)) - 1
7     while delta_step >= 1:
8         i = 0
9         for j in range(0, len(c2), delta_step):

```

```

10         i = next_close_point(c1, i, c2[j], d)
11         if i >= len(c1):
12             return True # Rejected
13
14         j = 0
15         for i_ in range(0, len(c1), delta_step):
16             j = next_close_point(c2, j, c1[i_], d)
17             if j >= len(c2):
18                 return True # Rejected
19
20         delta_step //= 2
21
22     return False # Not rejected

```

Listing 2.19: Finally decide which curves are too far away or not

Ορίζουμε δυο μεταβλητές με τα σημεία των δυο καμπυλών. Το σημείο εκκίνησης του δ step το ορίζουμε ως το πλήθος των σημείων της καμπύλης με τα περισσότερα σημεία. Η επανάληψη θα τρέξει έως ότου το βήμα μηδενιστεί ή έως ότου πάρουμε κάποιο αποτέλεσμα. Το δ step ελέγχει το πλήθος της δειγματοληψίας. Στην πρώτη επανάληψη για παράδειγμα, το δείγμα θα είναι το πολύ δυο σημεία το αρχικό και το τελικό.

Εντός του πρώτου `for` λοιπόν έχουμε την εξής κατάσταση: το j είναι τα σημεία δειγματοληψίας πάνω στην καμπύλη $c2$. Για κάθε τέτοιο σημείο, ψάχνουμε ένα ταίριασμα με ένα σημείο στην καμπύλη $c1$ εντός απόστασης δ χρησιμοποιώντας την συνάρτηση `next-close-point`. Αν η συνάρτηση `next-close-point` δεν βρει κάποιο τέτοιο σημείο, θα καταλήξει να επιστρέψει `len(c)` (δείτε τον κώδικα για την συνάρτηση) που εδώ είναι το `len(c1)`. Συνεπώς, τότε το i θα γίνει ίσο με `len(c1)` και άρα ξέρουμε πως δεν υπάρχει κατάλληλο σημείο. Για αυτό η επόμενη συνθήκη `if` ελέγχει ακριβώς αυτό και επιστρέφει `True`, ότι δηλαδή ναι οι καμπύλες βρίσκονται αρκετά μακριά μεταξύ τους και μην ψάξεις την ακριβή απόσταση Fréchet. Το επόμενο `for` κοιτάει τις καμπύλες από την αντίθετη σκοπιά. Αν σε κανένα σημείο της επανάληψης δεν καταλήξουμε σε συμπέρασμα, επιστρέφουμε τελικά `False` και θα πρέπει να ελέγξουμε ακριβώς την απόσταση Fréchet στην συνέχεια του κώδικα.

2.3.6 Δοκιμή του αρνητικού φίλτρου

Για να τρέξουμε τον κώδικα πάνω στη βάση δεδομένων και στη καμπύλη αναζήτησης γράφουμε τις παρακάτω γραμμές:

```

1 reject = 0
2 accept = 0
3 for idx in candidates:

```

```

4     candidate_curve = extract_curve("curve_dataset.csv", idx)
5     if neg_filter(k, candidate_curve, delta):
6         print("Curve ", idx , " was rejected")
7         reject += 1
8         continue # reject early
9     else:
10        print("Curve ", idx , " was accepted")
11        accept += 1
12        # proceed to upper bound or exact Fréchet
13 print("Number of curves too far apart from query: ", reject)
14 print("Number of curves close to query: ", accept)

```

Listing 2.20: Find out which curves pass the filter or not

Θυμηθείτε πως η συνάρτηση neg-filter επιστρέφει True όταν δεν βρεθεί σημείο της απέναντι καμπύλης εντός του δ και άρα οι δυο καμπύλες βρίσκονται αρκετά μακριά μεταξύ τους. Για τον λόγο αυτό όταν η συνθήκη ισχύει και άρα η συνάρτηση έχει επιστρέψει True τότε η καμπύλη έχει απορριφθεί και άρα αυξάνουμε το reject κατά ένα. Σε διαφορετική περίπτωση, η καμπύλη είναι αποδεκτή και αυξάνουμε το accept. Το αποτέλεσμα είναι το παρακάτω:

```

.....
Curve 14  was accepted
Curve 77  was accepted
Number of curves too far apart from query:  0
Number of curves close to query:  29

```

Όπως γίνεται άμεσα κατανοητό, καμία από τις υποψήφιες καμπύλες δεν βρίσκεται μακριά από την καμπύλη αναζήτησης. Το τετραγωνικό δέντρο έχει κάνει καταπληκτική δουλειά στον εντοπισμό των κατάλληλων καμπύλων. Ποιό όμως το αποτέλεσμα αν εφαρμόσουμε το φίλτρο σε όλες τις καμπύλες της βάσης δεδομένων; Αυτό θα μας δώσει μια εικόνα για την αποτελεσματικότητα του φίλτρου. Όπως και για τον άπληστο αλγόριθμο, χρειάζεται μόνο η αλλαγή της παραπάνω for σε for idx in range(100):

```

.....
Curve 97  was rejected
Curve 98  was accepted
Curve 99  was accepted
Number of curves too far apart from query:  50
Number of curves close to query:  50

```

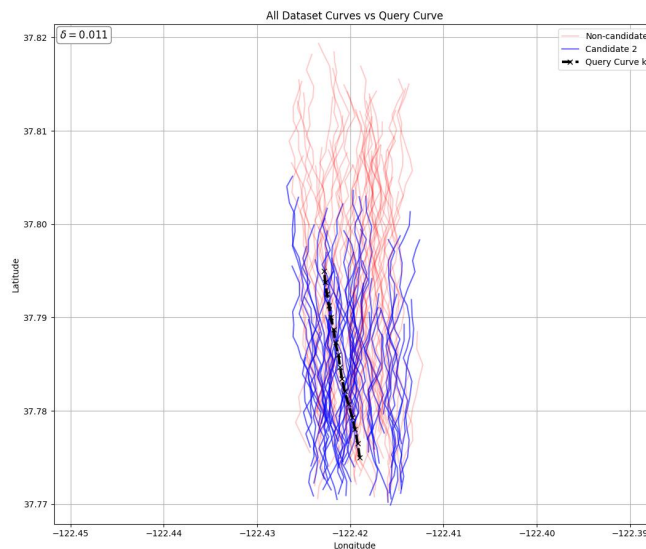
Παρατηρούμε μια καταπληκτική ισοροπία με τις μισές καμπύλες να απορρίπτονται ενώ τις υπόλοιπες μισές να γίνονται αποδεκτές. Οι καμπύλες που έγιναν δεκτές (βρίσκονται κοντά στην καμπύλη αναζήτησης) είναι οι:

[2, 3, 4, 5, 8, 10, 12, 13, 14, 15,
19, 20, 21, 23, 24, 26, 31, 34, 35, 37,
40, 41, 43, 45, 46, 48, 50, 52, 56, 57,
58, 59, 62, 66, 67, 69, 70, 72, 74, 75,
77, 79, 80, 83, 84, 88, 90, 91, 98, 99]

ενώ οι καμπύλες που απορρίφθηκαν (βρίσκονται μακριά από την καμπύλη αναζήτησης) είναι οι:

[0, 1, 6, 7, 9, 11, 16, 17, 18, 22,
25, 27, 28, 29, 30, 32, 33, 36, 38, 39,
42, 44, 47, 49, 51, 53, 54, 55, 60, 61,
63, 64, 65, 68, 71, 73, 76, 78, 81, 82,
85, 86, 87, 89, 92, 93, 94, 95, 96, 97]

Ας σχεδιάσουμε τις παραπάνω καμπύλες σε ένα γράφημα. Με μπλε χρώμα οι αποδεκτές ενώ με κόκκινο αυτές που απορρίφθηκαν:

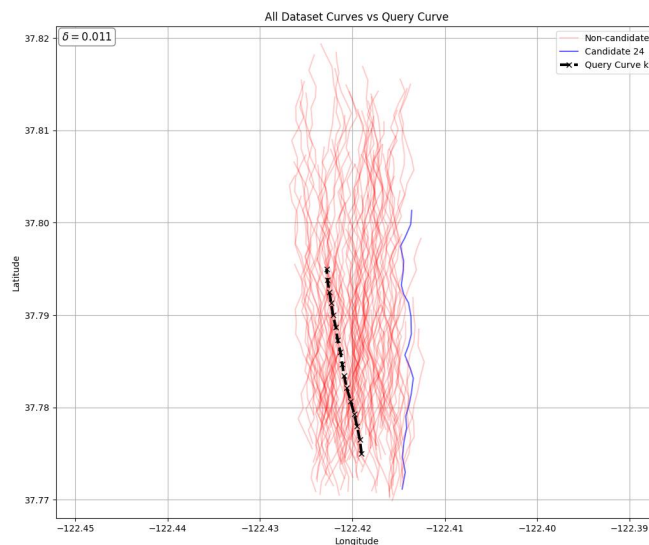


Σχήμα 2.9: Με μπλε χρώμα οι καμπύλες που έγιναν αποδεκτές από το αρνητικό φίλτρο ενώ με κόκκινο αυτές που απορρίφθηκαν

Το αρνητικό φίλτρο έχει κάνει καλή δουλειά στο να βρει καμπύλες κοντά στην καμπύλη αναζήτησης. Έχει σωστά κρατήσει εκείνες τις καμπύλες οι οποίες ενδεχομένως να βρίσκονται στο κατάλληλο όριο απόστασης. Καμπύλες με μεγάλο μήκος και με τα σημεία τους μακριά από την καμπύλη αναζήτησης έχουν απορριφθεί. Βέβαια, ισχύουν και εδώ οι ίδιες παρατηρήσεις με το άπληστο φίλτρο όσον αφορά τις καμπύλες σε σχέση με την δεξαμενή των υποψηφίων. Ανατρέξτε σε εκείνο το σημείο για περισσότερες λεπτομέρειες.

2.3.7 Σύγκριση θετικού και αρνητικού φίλτρου

Ένα ενδιαφέρον στοιχείο εδώ είναι ποια καμπύλη είναι διαφορετική στο θετικό φίλτρο και στο αρνητικό φίλτρο. Αν δείτε τις καμπύλες που έχουν απορριφθεί από το αρνητικό φίλτρο (50 τον αριθμό), και εκείνες που απέτυχαν στον άπληστο αλγόριθμο (51 τον αριθμό) είναι οι ίδιες εκτός μιας. Αυτό σημαίνει πως τα δυο φίλτρα έχουν παρόμοιες επιδόσεις, τουλάχιστον για την συγκεκριμένη βάση δεδομένων. Αυτή η μια καμπύλη που διαφέρει στα δυο σύνολα είναι εκείνη με index 24. Ας ζωγραφίσουμε την συγκεκριμένη καμπύλη:



Σχήμα 2.10: Με μπλε χρώμα η καμπύλη με δείκτη 24 η οποία διαφέρει στα δυο φίλτρα.

Η καμπύλη πράγματι βρίσκεται σε μια γκρίζα ζώνη σχετικά με την αν η απόσταση Fréchet της θα είναι κάτω ή πάνω από το όριο. Η απάντηση στο ερώτημα αυτό θα δώσει και τον νικητή των δυο φίλτρων. Για τον σκοπό αυτό θα τρέξουμε τον ακριβή αλγόριθμο υπολογισμού της απόστασης Fréchet που είδαμε στο πρώτο κεφάλαιο:

Frechet distance: 0.011214
 Lower Bound is:
 0.011185186721726413

Όπως φαίνεται τα φίλτρα είχαν δίκαιο να δυσκολεύονται καθώς βρισκόμαστε στο όριο του στατιστικού σφάλματος. Θυμηθείτε πως έχουμε ορίσει $\delta = 0.011$. Το θετικό φίλτρο έχει απορρίψει την παραπάνω καμπύλη και άρα μας λέει στην ουσία πως το άνω όριο της για την απόσταση Fréchet είναι μεγαλύτερο του δ και να βρούμε αναλυτικά την απόσταση. Από την άλλη, το αρνητικό φίλτρο έχει αποδεχθεί την καμπύλη 24 και άρα μας λέει πως οι καμπύλες δεν βρίσκονται πολύ μακριά και άρα καλό θα ήταν να υπολογίσουμε χειροκίνητα την απόσταση Fréchet και άρα δεν χρειάζεται να ασχοληθούμε περαιτέρω αφού σίγουρα θα είναι $d_F > \delta$. Συνεπώς, και τα δυο φίλτρα έχουν κάνει σωστά την δουλειά τους και μας λένε πως το καλύτερο θα ήταν να υπολογίσουμε επακριβώς την απόσταση Fréchet και μετά να αποφασίσουμε. Το αποτέλεσμα λοιπόν είναι ισόπαλο.

2.4 Αναδρομικός αλγόριθμος για την απόσταση Fréchet

2.4.1 Ο αλγόριθμος και ο κώδικας

Με την εφαρμογή των φίλτρων των προηγούμενων παραγράφων, έχουμε καταλήξει πλέον σε κάποιες υποψήφιες καμπύλες για τις οποίες δεν ξέρουμε άλλους προσεγγιστικούς αλγόριθμους και άρα θα πρέπει να υπολογίσουμε επακριβώς την απόσταση Fréchet ώστε να κρίνουμε αν τελικά είναι μικρότερη ή όχι από το όριο δ .

Μπορούμε πολύ απλά να εφαρμόσουμε τον αλγόριθμο που υλοποιήσαμε στο πρώτο κεφάλαιο, δηλαδή τον κλασσικό αλγόριθμο των Alt και Godau και να τελειώσουμε μια ώρα αρχύτερα. Αυτό όμως θα ήταν βαρετό και καθόλου ενδιαφέρον. Ένα βασικό πρόβλημα του κλασσικού αλγορίθμου είναι πως εκτελεί περισσότερους από τους απαραίτητους υπολογισμούς, αφού ψάχνει όλα τα κελιά του διαγράμματος ελεύθερου χώρου. Όπως είδαμε στα παραδείγματα, τα περισσότερα κελιά είναι μάλλον κενά από ελεύθερο χώρο και άρα άχρηστα για την αναζήτησή μας. Η χρονική πολυπλοκότητα εκείνου του αλγορίθμου ήταν της μορφής $O(pm \log(pm))$. Αν και υπάρχουν ισχυρές αποδείξεις ότι ο θεωρητικός ανώτατος χρόνος δεν μπορεί να πέσει κάτω από αυτό το όριο, εντούτοις σε πρακτικές εφαρμογές μπορούμε να πετύχουμε καλύτερα αποτελέσματα μέσω της μεθόδου που θα περιγράψουμε παρακάτω.

Εδώ έρχεται να βοηθήσει ο αλγόριθμος των Baldus και Bringmann. Αντί να

υπολογίζει και να αποθηκεύει ρητά ολόκληρο το διάγραμμα ελεύθερου χώρου, η αναδρομική προσέγγιση διαιρεί το πρόβλημα χωρικά και εξερευνά μόνο τα προσβάσιμα τμήματα του διαγράμματος ανάλογα με τις ανάγκες. Φανταστείτε το σαν ένα τετράγωνο δέντρο για το διάγραμμα ελεύθερου χώρου όπως αυτό που είδαμε παραπάνω. Εάν ένα τμήμα του διαγράμματος δεν μπορεί να διανυθεί (με βάση τα γεωμετρικά όρια), κλαδεύεται νωρίς και έτσι αποφεύγουμε τον υπολογισμό κάθε κελιού. Πρακτικά λοιπόν, όπως η δομή δεδομένων της δυαδικής αναζήτησης και του τετραγωνικού δέντρου, έτσι και εδώ θα δημιουργήσουμε μια δομή δεδομένων η οποία θα επιτρέπει ταχείες αναζητήσεις.

Ο αλγόριθμος θα είναι αναδρομικός υπό την έννοια πως ξεκινώντας από τον συνολικό ελεύθερο χώρο αυτός θα διαρείται σε κάθε βήμα σε μικρότερα μέρη τα οποία θα επεξεργαστούμε ανεξάρτητα. Για κάθε υποπεριοχή θα ελέγχουμε αν υπάρχει δυνατότητα μετάβασης από κάτω αριστερά προς τα πάνω δεξιά με την ίδια λογική που χρησιμοποιοποιήσαμε στον κλασσικό αλγόριθμο. Αν κάτι τέτοιο δεν είναι δυνατόν, η περιοχή απορρίπτεται και γλιτώνουμε τον αναλυτικό υπολογισμό για τα κελιά της. Το ελάχιστο επίπεδο διαμερίσεων που μπορούμε να φτάσουμε είναι κάθε υπόπεριοχή να αφορά ένα κελί του αρχικού χώρου.

Θα ξεκινήσουμε σταδιακά με τον αλγόριθμό και όλες οι ενδιαφέροντες επιλογές θα φανούν στην πορεία. Όπως πάντα ορίζουμε κάποιες βασικές συναρτήσεις:

```

1 def intersection_interval(p, d, q0, q1):
2     result = interval_within_epsilon((q0, q1), p, d)
3     if result is None:
4         return (1, 0)
5     return result
6
7 def get_reachable_horizontal_border(i, j, a, b, d):
8     if j + 1 >= len(b) or i >= len(a): # Care not to overflow
9         return [1, 0]
10    start, end = intersection_interval(a[i], d, b[j], b[j + 1])
11    return [start + j, end + j]
12
13 def get_reachable_vertical_border(i, j, a, b, d):
14     if j + 1 >= len(a) or i >= len(b):
15         return [1, 0]
16    start, end = intersection_interval(b[i], d, a[j], a[j + 1])
17    return [start + j, end + j]
```

Listing 2.21: Support function to find the free space of the border of a box

Οι παραπάνω βοηθητικές συναρτήσεις θα χρησιμοποιηθούν για την εύρεση του ελεύθερου χώρου στα σύνορα κάθε υποχώρου. Η λογική είναι ίδια με την κλασσική περίπτωση και μάλιστα χρησιμοποιούμε την ίδια συνάρτηση `interval-within-epsilon` (Listing 1.3 και 1.4 σελ. 25) μόνο που τώρα επιστρέφουμε κάτι διαφορετικό

αν δεν υπάρχει σύνορο. Στη συνέχεια, αν σκεφτούμε ένα κουτί στον ελεύθερο χώρο με τα δυο άκρα του να είναι τα τμήματα $a[j]$, $a[j+1]$ και $b[j]$, $b[j+1]$, οι δυο συναρτήσεις `get-reachable` δέχονται τα i και j καθώς και τα σημεία a και b των δυο υποσυνόλων της καμπύλης μαζί με την τιμή ανοχής και επιστρέφουν τα δυο άκρα του ελεύθερου χώρου στα σύνορα. Στο τέλος μετατοπίζουμε αυτό το διάστημα κατά j ώστε να γνωρίζουμε την θέση στο συνολικό αρχικό χώρο.

Προσθέτουμε τώρα, ακόμη μερικές βοηθητικές συναρτήσεις:

```

1 def is_empty_interval(interval):
2     return interval[0] > interval[1]
3
4 def merge_intervals(intervals, new_interval, eps=1e-9):
5     if is_empty_interval(new_interval):
6         return
7         # Check whether the first value of the new interval
8         # is lower or equal to the second value of the rightmost
9         # interval space of the list
10    if intervals and new_interval[0] - eps <= intervals[-1][1]:
11        intervals[-1][1] = max(intervals[-1][1], new_interval[1])
12    else:
13        intervals.append(list(new_interval))

```

Listing 2.22: Support functions to merge overlapping borders

Η πρώτη συνάρτηση είναι αρκετά ξεκάθαρη. Ελέγχει αν το υπό εξέταση διάστημα υπάρχει. Ένα φυσιολογικό διάστημα είναι της μορφής $[a, b]$ με $b \geq a$. Συνεπώς, αν παρατηρήσουμε το αντίθετο, το διάστημα είναι άκυρο. Γίνεται έτσι φανερός και ο λόγος που επιστρέφουμε $[1, 0]$ στην περίπτωση που δεν υπάρχει ελεύθερος χώρος σε κάποιο σύνορο.

Η επόμενη συνάρτηση, η `merge-intervals` χρησιμοποιείται ώστε να υπάρχει συνέπεια μεταξύ των διαστημάτων ελεύθερου χώρου. Ως είσοδο λαμβάνει δυο λίστες, μια με το παλιό διάστημα και μια με ένα καινούργιο. Για παράδειγμα μπορεί αρχικά να έχουμε ως ελεύθερο χώρο το διάστημα $[0.1, 0.5]$ και στη συνέχεια να μας προκύψει το διάστημα $[0.4, 0.7]$. Αν τρέξουμε την συνάρτηση με αυτά τα δυο ορίσματα τότε θα μας επιστρέψει το διάστημα $[0.1, 0.7]$. Αν από την άλλη το νέο διάστημα δεν υπερκαλύπτεται με κάποιο από τα προηγούμενα, θα αποθηκευτεί στη λίστα με τα διαστήματα όπως είναι. Για παράδειγμα αν έρθει το $[1.1, 1.2]$ η λίστα θα γίνει $[[0.1, 0.7], [1.1, 1.2]]$. Επιπλέον ορίζουμε και κάποιο ελάχιστο όριο ώστε να αποφύγουμε σφάλματα δεκαδικών αριθμών.

Για την συνέχεια μια απλή συνάρτηση:

```

1 def overlap(intervals, i0, i1):
2     for i in intervals:

```

```

3         if i[0] <= i1 and i[1] >= i0:
4             return i
5     return None

```

Listing 2.23: Support function to check overlapping intervals

Στην παραπάνω λογική ελέγχουμε αν κάποιο νέο διάστημα υπερκαλύπτει κάποιο από τα παλιά. Αν κάτι τέτοιο ισχύει τότε επιστρέφουμε το διάστημα το οποίο υπερκαλύπτεται. Συνεχίζουμε ακάθεκτοι:

```

1 def get_last_reachable_point_from_start(P, Q, d):
2     j = 0 # Start from the 1st point of Q
3     # With P[0] fixed, swipe through all points of Q
4     # and find the last one that is within d distance
5     while j < len(Q) - 2 and dist(P[0], Q[j + 1]) <= d:
6         j += 1
7     # Find the free space on the border of the cell that
8     # corresponds to P[0] and the Q[j].
9     return get_reachable_horizontal_border(0, j, P, Q, d)[1]

```

Listing 2.24: This function finds the furthest reachable point index

Εδώ τώρα, δίνουμε ως ορίσματα δυο καμπύλες (που μπορεί να προέρχονται από την υποδιαίρεση των αρχικών καμπυλών) και το κατώφλι. Ξεκινούμε φιζάροντας το πρώτο σημείο της καμπύλης P . Στη συνέχεια, σαρώνουμε όλα τα σημεία της καμπύλης Q και ψάχνουμε το πιο μακρινό σε δείκτη σημείο για το οποίο η απόσταση με το $P[0]$ παραμένει εντός του κατωφλιού. Στη συνέχεια επιστρέφουμε το τελευταίο σημείο του ελεύθερου χώρου στο οριζόντιο σύνορο, του κελιού των δυο τμημάτων που φέρονται από τα σημεία $P[0]$ και $Q[j]$.

Επόμενη συνάρτηση:

```

1 def get_dist_to_point_sqr(curve, pt):
2     return max(dist_sqr(p, pt) for p in curve)

```

Listing 2.25: Find the max distance between a point and all the curve points

Η συνάρτηση αυτή υπολογίζει τη μέγιστη τετραγωνική απόσταση από ένα σημείο pt σε κάθε σημείο μιας καμπύλης. Θα την χρησιμοποιήσουμε ευθύς αμέσως:

```

1 def is_frechet_distance_at_most(P, Q, d):
2     if dist(P[0], Q[0]) > d or dist(P[-1], Q[-1]) > d:
3         return False
4     if len(P) == 1 and len(b) == 1:
5         return True
6     elif len(P) == 1:
7         return get_dist_to_point_sqr(Q, P[0]) <= sqr(d)
8     elif len(Q) == 1:

```

```

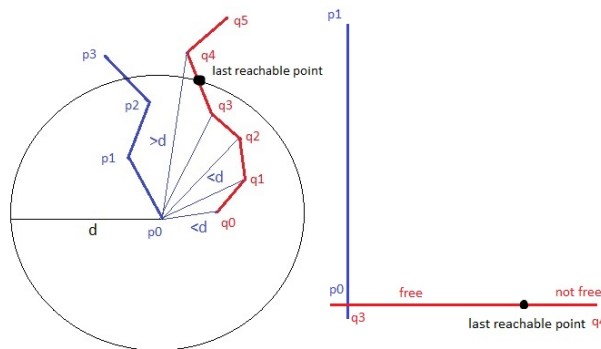
9      return get_dist_to_point_sqr(P, Q[0]) <= sqr(d)
10
11     rP = [[0, get_last_reachable_point_from_start(P, Q, d)]]
12     rQ = [[0, get_last_reachable_point_from_start(Q, P, d)]]
13     rP_out, rQ_out = [], []
14
15     get_reachable_intervals(0, len(P) - 1, 0, len(Q) - 1, P, Q,
16                             d, rQ, rP, rQ_out, rP_out)
17
18     return rP_out and rP_out[-1][1] >= len(Q) - 1.5

```

Listing 2.26: Checks if the Fréchet distance between curves P and Q is $\leq d$

Συνοπτικά, η παραπάνω συνάρτηση ελέγχει αν η απόσταση Fréchet για τις καμπύλες P και Q είναι το πολύ d . Ο πρώτος έλεγχος γίνεται στα αρχικά και τελικά σημείο. Όπως έχουμε δει και σε πορηγούμενες ενότητες, η απόσταση μεταξύ αρχικών και τελικών σημείων αποτελεί ένα κάτω όριο για την απόσταση Fréchet. Συνεπώς, αν αυτές οι αποστάσεις είναι μεγαλύτερες του d έχουμε αποτύχει και επιστρέφουμε False. Η επόμενη αλληλουχία προτάσεων if else if ελέγχει τις περιπτώσεις για τις οποίες οι καμπύλες αποτελούνται μόνο από ένα σημείο. Αν οποιαδήποτε από τις καμπύλες αποτελείται από ένα σημείο, τότε ελέγχουμε αν η απόστασή του από τα σημεία της άλλης καμπύλης είναι ποτέ μικρότερη από το d και επιστρέφουμε ανάλογα.

Στη συνέχεια ξεκινά η επαναληπτική διαδικασία. Ορίζουμε τα δυο διαστήματα rP και rQ . Για κάτω άκρο του διαστήματος θέτουμε το σημείο με δείκτη 0, δηλαδή το αρχικό σημείο των καμπυλών. Για το άνω άκρο, τρέχουμε την συνάρτηση get-last-reachable-point την οποία είδαμε λίγο παραπάνω. Η συνάρτηση αυτή θα μας επιστρέψει το τελευταίο σημείο του ελεύθερου χώρου στο σύνορο των δυο καμπυλών. Δείτε για παράδειγμα το παρακάτω σχήμα:



Σχήμα 2.11: Αριστερά οι καμπύλες, δεξιά το διάγραμμα ελεύθερου χώρου για το συγκεκριμένο κελί.

Απομένει τώρα να καλέσουμε την αναδρομική συνάρτηση η οποία θα αρχίσει να χωρίζει τον χώρο σε μικρότερα τμήματα. Είναι ώρα λοιπόν, να περάσουμε στο ζουμί της υπόθεσης που είναι η επόμενη συνάρτηση:

```

1 def get_reachable_intervals(i_min, i_max, j_min, j_max,
2                             P, Q, d, rQ, rP, rQ_out, rP_out):

```

Listing 2.27: Recursive function to compute which parts of the free-space diagram are reachable

Αυτή η συνάρτηση θα υπολογίζει αναδρομικά ποια μέρη του διαγράμματος ελεύθερου χώρου είναι προσβάσιμα, επιτρέποντας την απόφαση σχετικά με το αν η απόσταση Frechet είναι μικρότερη ή ίση με ένα κατώφλι d . Είναι νομίζω απαραίτητο να δώσουμε μια αναλυτική λίστα με τις παραμέτρους ώστε να γίνει κατανοητή η λογική:

- $i_{\min}, i_{\max}$: Αρχικός και τελικός δείκτης των σημείων της καμπύλης P .
- $j_{\min}, j_{\max}$: Αρχικός και τελικός δείκτης των σημείων της καμπύλης Q .
- P, Q : Οι δύο καμπύλες ως λίστες σημείων.
- d : Το κατώφλι της απόστασης Fréchet.
- rQ : Ελεύθερος χώρος στα οριζόντια σύνορα, δηλαδή κατά μήκος της καμπύλης Q .
- rP : Ελεύθερος χώρος στα κατακόρυφα σύνορα, δηλαδή κατά μήκος της καμπύλης P .
- rQ_{out} : Μεταβλητή που συλλέγει τα νέα οριζόντια διαστήματα ελεύθερου χώρου.
- rP_{out} : Μεταβλητή που συλλέγει τα νέα κατακόρυφα διαστήματα ελεύθερου χώρου.

Θα δούμε αναλυτικά την χρήση κάθε παραμέτρου εντός της συνάρτησης στη συνέχεια. Συνεχίζουμε:

```

1     tQ = overlap(rQ, j_min, j_max)
2     tP = overlap(rP, i_min, i_max)
3
4     if tQ is None and tP is None:
5         return

```

Listing 2.28: Check for any reachable boundary

Αρχικά ορίζουμε δυο νέα διαστήματα tP και tQ . Για τα διαστήματα αυτά, τρέχουμε τη συνάρτηση `overlap` που είδαμε παραπάνω. Θυμίζουμε πως η `overlap` δέχεται μια λίστα από διαστήματα, καθώς και τα άκρα από ένα νέο διάστημα και ελέγχει αν κάποιο από αυτά υπερκαλύπτεται από το νέο διάστημα. Στην περίπτωση που αυτό ισχύει, επιστρέφει το υπερκαλυπτόμενο διάστημα. Κατά το πρώτο βήμα, το νέο διάστημα που δίνουμε είναι όλο το μήκος της καμπύλης και άρα σίγουρα θα μας επιστρέψει κάποιο αποτέλεσμα, το rQ και το rP που υπολογίσαμε εντός της `is-frechet-distance-at-most`. Δείτε τώρα την συνέχεια:

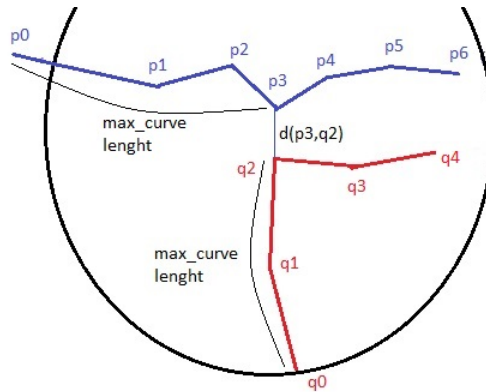
```

1 if tQ and tP:
2     i_mid = (i_min + 1 + i_max) // 2
3     j_mid = (j_min + 1 + j_max) // 2
4     d_est = dist(P[i_mid], Q[j_mid]) + max(
5         curve_length(P, i_min + 1, i_mid),
6         curve_length(P, i_mid, i_max)
7     ) + max(
8         curve_length(Q, j_min + 1, j_mid),
9         curve_length(Q, j_mid, j_max)
10    )

```

Listing 2.29: Try to short cut full sub-box if reachable

Βρίσκουμε τα ενδιάμεσα σημεία και υπολογίζουμε την Ευκλείδεια απόστασή τους. Σε αυτήν ωστόσο, προσθέτουμε και κάποιες επιπλέον τιμές, το μέγιστο από τις εξής ποσότητες: το μήκος της καμπύλης από την αρχή ως το ενδιάμεσο σημείο και το μήκος της καμπύλης από το ενδιάμεσο σημείο έως το τέλος και για τις δυο καμπύλες. Το παραπάνω άθροισμα αποτελεί ένα ανώτατο όριο για το μήκος Fréchet. Γιατί ισχύει αυτό μπορούμε να το δούμε με το παρακάτω σχήμα:



Σχήμα 2.12: Το άθροισμα των παραπάνω ποσοτήτων δίνει το ανώτατο όριο της απόστασης Fréchet.

Βλέπουμε λοιπόν στο σχήμα πως το άθροισμα των 3 αποστάσεων έχει συνδέσει τα δυο πιο απομακρυσμένα σημεία των καμπυλών και άρα αν ορίσουμε ως ϵ την

απόσταση αυτή, το μήκος για το λουρί του σκύλου είναι αρκετό ώστε να μας κουβαλήσει καθόλη την διάρκεια του περιπάτου. Φανταστείτε και το χειρίστο σενάριο όπου το p_0 βρίσκεται κατακόρυφα επάνω από το p_3 και το q_0 κατακόρυφα κάτω από το q_2 . Συνεπώς, η συνέχεια του κώδικα είναι η εξής:

```

1         if d_est <= d:
2             merge_intervals(rQ_out, [j_min, j_max])
3             merge_intervals(rP_out, [i_min, i_max])
4             return

```

Listing 2.30: If the upper boundary of Frechet distance is small accept.

Αν λοιπόν αυτό το ανώτατο όριο είναι μικρότερο από το κατώφλι, τότε σταματάμε και αποδεχόμαστε όλο το κουτί. Αυτή είναι μια πολύ ισχυρή βελτιστοποίηση ώστε να μην εκτελούμε αχρείαστες επαναλήψεις. Πάμε τώρα στην περίπτωση που καταλήξουμε σε ένα κουτί που είναι ακριβώς ένα κελί του διαγράμματος ελεύθερου χώρου. Τότε

```

1         if i_max == i_min + 1 and j_max == j_min + 1:
2             aa = get_reachable_horizontal_border(i_max, j_min, P, Q,
3             d)
4             bb = get_reachable_vertical_border(i_min, j_max, P, Q, d)
5
6             if tP is None and tQ:
7                 aa[0] = max(aa[0], tQ[0])
8             elif tQ is None and tP:
9                 bb[0] = max(bb[0], tP[0])
10
11             merge_intervals(rQ_out, aa)
12             merge_intervals(rP_out, bb)
13             return

```

Listing 2.31: Case of a cell in free space diagram.

Αν ο μέγιστο δείκτης είναι ένα μεγαλύτερος από το μέσο δείκτη, βρισκόμαστε στην περίπτωση ενός κελιού, δηλαδή του ελεύθερου χωρού μεταξύ δυο ακμών των καμπυλών. Εδώ πρέπει να κάνουμε αναλυτικό υπολογισμό και έτσι βρίσκουμε τον ελεύθερο χώρο στα σύνορα του κελιού και στη συνέχεια ενώνουμε τα σύνορα με τα ήδη υπάρχοντα μέσω της συνάρτησης merge intervals.

Ακολουθεί τώρα το σημαντικό σημείο της συνάρτησης:

```

1         if (j_max - j_min) > (i_max - i_min):
2             j_split = (j_min + j_max) // 2
3             ra_middle = []
4             get_reachable_intervals(i_min, i_max, j_min, j_split, P,
5             Q, d, rQ, rP, rQ_out, ra_middle)
6             get_reachable_intervals(i_min, i_max, j_split, j_max, P,
7             Q, d, rQ, ra_middle, rQ_out, rP_out)

```

```

6     else:
7         i_split = (i_min + i_max) // 2
8         rb_middle = []
9         get_reachable_intervals(i_min, i_split, j_min, j_max, P,
10                                Q, d, rQ, rP, rb_middle, rP_out)
11         get_reachable_intervals(i_split, i_max, j_min, j_max, P,
12                                Q, d, rb_middle, rP, rQ_out, rP_out)

```

Listing 2.32: Recursive subdivision of the free space.

Αυτό είναι και το επαναληπτικό βήμα, όπου καλούμε εκ νέου την συνάρτηση από το ίδιο της το σώμα. Αν δεν μπορούσαμε να διώξουμε το συγκεκριμένο κουτί του ελεύθερου χώρου μέσω του ελέγχου για το ανώτατο όριο της απόστασης Fréchet, τότε θα το χωρίσουμε σε μικρότερα τμήματα και θα εστιάσουμε σε αυτά. Στην ουσία, ξεκινώντας από το πλήρες μήκων των καμπυλών, σε κάθε βήμα τις χωρίζουμε στα δυο, όπως φαίνεται και από τα *i-split* και *j-split* και εφαρμόζουμε τον αλγόριθμο ξανά για τα μικρότερα τμήματα. Προσοχή όμως, επιλέγουμε να χωρίσουμε στα δυο την καμπύλη με το μεγαλύτερο πλήθος σημείων ώστε να υπάρχει μια ισοροπία στα πόσα σημεία έχουμε από κάθε καμπύλη. Αν αυτό το δούμε από την σκοπία τον άξονων του διαγράμματος ελεύθερου χώρου, τότε χωρίζουμε στη μέση τον άξονα με το μεγαλύτερο μήκος.

Αν σε κάποια από τα κουτιά στα παραπάνω βήματα εντοπίσουμε πως δεν έχει καθόλου ελεύθερο χώρο στα σύνορά του, το απορρίπτουμε και δεν συνεχίζουμε τον αλγόριθμο. Αν από την άλλη ισχύει το κριτήριο της ανώτατης απόστασης Fréchet που είδαμε παραπάνω το αποδεχόμαστε και ξανά δεν συνεχίζουμε την διαμέριση. Εδώ γίνεται τώρα πλήρης κατανοητή η χρήση των *tQ* και *tP*. Μας δείχνουν για κάθε κουτί, ποιο τμήμα του ελεύθερου χώρου που έχουμε υπολογίσει αρχικά περιλαμβάνει. Αν δούμε την όλη διαδικασία σαν τετραγωνικό δέντρο, οι περιπτώσεις που έχουμε καταλήξει σε κουτί ίσο με κάποιο κελί ισοδυναμεί με τα φύλλα του δέντρου όπου δεν είναι δυνατή επιπλέον διαμέριση και άρα πρέπει να γίνει ο υπολογισμός.

Τώρα μπορεί να γίνει κατανοητή και η επιστροφή της συνάρτησης *is-frechet-distance-at-most* στο εσωτερικό της οποίας τρέχουμε το δέντρο για τον ελεύθερο χώρο:

```

1     return rP_out and rP_out[-1][1] >= len(Q) - 1.5

```

Listing 2.33: Return line from *is-frechet-distance-at-most*.

Η συνάρτηση *is-frechet-distance-at-most* θέλει να αποφασίσει αν μπορούμε να πάμε από το αρχικό σημείο $(0, 0)$ του διαγράμματος ελεύθερου χώρου στο τελικό σημείο πάνω δεξιά. Το *rP-out* κρατά τα διαστήματα ελεύθερου χώρου στον άξονα της καμπύλης P ενώ το *rQ-out* κρατά τα διαστήματα ελεύθερου χώρου στον άξονα

της καμπύλης Q . Αυτές οι μεταβλητές θυμηθείτε πως υπολογίζονται εντός της συνάρτησης `get-reachable-intervals`. Για κάθε διάστημα σε αυτή την λίστα, σημαίνει πως υπάρχει ο αντίστοιχος ελεύθερος χώρος στην αντίστοιχη καμπύλη.

Ελέγχουμε λοιπόν, τα εξής: αν υπάρχει η λίστα `rP-out` δηλαδή αν περιλαμβάνει στοιχεία. Διαφορετικά δεν υπάρχει και ελεύθερος χώρος. Επιπλέον, ελέγχουμε το τελευταίο στοιχείο του τελευταίου διαστήματος, για το αν είναι ίσο με τον δείκτη του πάνω δεξιά στοιχείου της καμπύλης, δηλαδή του τελικού στοιχείου του διαγράμματος ελεύθερου χώρου. Αυτό θα σημαίνει πως ακολουθώντας τα διαδοχικά διαστήματα ελεύθερου χώρου, θα έχουμε φτάσει από την αρχή στο τέλος εντός τιμής κατωφλίου και άρα επιστρέφουμε `True`.

Παρατηρείστε πως η συνθήκη δεν είναι $\text{len}(Q) - 1.0$ αλλά $\text{len}(Q) - 1.5$. Αυτό είναι ένα έξυπνο `trick` ώστε πρώτον να είμαστε σίγουροι ότι έχουμε φτάσει στο τελικό κουτί που περιλαμβάνει το πάνω δεξιά σημείο, αλλά και να προστατευτούμε από τυχόν προβλήματα ακριβείας λόγω και των μισών βημάτων που συχνά εκτελούμε.

Έχουμε ολοκληρώσει την γενική εικόνα του αναδρομικού αλγορίθμου. Είδαμε πως ακολουθήσαμε με αρκετά περίπλοκη διαδικασία με πολλές αλληλοεξαρτώμενες συναρτήσεις. Θα προσπαθήσουμε να κάνουμε μια ανακεφαλαίωση. Αρχίζουμε λοιπόν, από το συνολικό διάγραμμα ελεύθερου χώρου και τον ελεύθερο χώρο που βλέπει το αρχικό σημείο κάτω αριστερά για κάποιο συγκεκριμένο κατώφλι απόστασης. Τρέχουμε την επαναληπτική διαδικασία και ξεκινούμε να χωρίζουμε τον χώρο σε τμήματα. Για κάθε τμήμα υπολογίζουμε τον ελεύθερο χώρο στα σύνορα και συγχωνεύουμε με τον προηγούμενο ελεύθερο χώρο. Υπολογίζουμε την ανώτατη απόσταση *Fréchet* για κάθε τμήμα και αν είναι μικρότερη από το κατώφλι τότε όλο το διάστημα του τμήματος αποθηκεύεται ως ελεύθερος χώρος χωρίς περαιτέρω διαμερισμούς. Παρόμοια, αν δεν υπάρχει καθόλου ελεύθερος χώρος, όλο το κουτί απορρίπτεται. Αν τελικά φτάσουμε σε κουτί το οποίο να αποτελεί κελί του αρχικού χώρου, δεν μπορούμε να διαμεριστούμε περισσότερο και πρέπει να υπολογίσουμε αναλυτικά τα ελεύθερα σύνορα. Τελικά, ελέγχουμε τα διαστήματα ελεύθερου χώρου πάνω σε κάποια από τις καμπύλες και αν διαπιστώσουμε πως υπάρχει μονοπάτι, επιστρέφουμε `True`, αλλιώς `False`.

Απομένει μόνο να τρέξουμε τον αλγόριθμο. Η λογική εδώ είναι ίδια με την κλασσική περίπτωση. Βρίσκουμε όλα τα κρίσιμα ϵ σημεία και εκτελούμε δυαδική αναζήτηση πάνω σε αυτά για να βρούμε το ελάχιστο για το οποίο ο επαναληπτικός αλγόριθμος επιστρέφει `True`:

```

1 def frechet_distance_recursive(P, Q):
2     critical_epsilons = compute_critical_epsilons(P, Q)
3     left = 0
4     right = len(critical_epsilons) - 1
5     result = critical_epsilons[-1]
```

```
6
7  # Binary search on critical epsilons
8  # Find the smaller epsilon that returns True
9  while left <= right:
10     mid = (left + right) // 2
11     eps = critical_epsilons[mid]
12     if is_frechet_distance_at_most(P, Q, eps):
13         result = eps
14         right = mid - 1
15     else:
16         left = mid + 1
17
18  return result
19
20 distance = frechet_distance_recursive(k, P)
21 print(f"Frechet distance recursive: {distance:.8f}")
```

Listing 2.34: Find the Frechet distance using the recursive algorithm.

Φαίνεται ξεκάθαρα η δυαδική αναζήτηση μόνο που τώρα χρησιμοποιήσουμε τον νέο βελτιωμένο αλγόριθμο αντί του κλασσικού.

2.4.2 Η έξοδος του κώδικα

Μένει να δούμε το αποτέλεσμα. Και δεν υπάρχει καλύτερη καμπύλη να το δοκιμάσουμε από την καμπύλη 24 της βάσης δεδομένων που είδαμε την περίεργη συμπεριφορά της στα φίλτρα. Δείτε το αποτέλεσμα:

```
Frechet distance classic: 0.01121375
Lower Bound is: 0.011185186721726413
Frechet distance recursive: 0.01118519
```

Έχουμε κρατήσει και το κάτω σύνορο γιατί μας φάνηκε ενδιαφέρουσα η ταύτιση του με την επαναληπτική απόσταση Fréchet. Φαίνεται λοιπόν ο αλγόριθμος να λειτουργεί, αλλά να υπάρχει μια μικρή απόκλιση από την κλασσική περίπτωση. Ας δοκιμάσουμε μια πληθώρα καμπυλών σε σχέση με την καμπύλη αναζήτησης ώστε να συγκρίνουμε τα αποτελέσματα:

```
Curve 1:
  Frechet distance classic = 0.02196756
  Frechet distance recursive = 0.02196756
```

```

-----
Curve 2:
  Frechet distance classic  = 0.01317630
  Frechet distance recursive = 0.01286871
-----
Curve 3:
  Frechet distance classic  = 0.00566988
  Frechet distance recursive = 0.01312394
-----
Curve 4:
  Frechet distance classic  = 0.00374274
  Frechet distance recursive = 0.00370138
-----
Curve 5:
  Frechet distance classic  = 0.00529723
  Frechet distance recursive = 0.01119320
-----
Curve 6:
  Frechet distance classic  = 0.00818862
  Frechet distance recursive = 0.00839437
-----
Curve 7:
  Frechet distance classic  = 0.02141045
  Frechet distance recursive = 0.02137966
-----
Curve 8:
  Frechet distance classic  = 0.01476042
  Frechet distance recursive = 0.01474233
-----
Curve 9:
  Frechet distance classic  = 0.00914234
  Frechet distance recursive = 0.00914234
-----
Curve 10:
  Frechet distance classic  = 0.02184077
  Frechet distance recursive = 0.02178219
-----

```

Τα πράγματα φαίνεται να λειτουργούν σωστά, ωστόσο παρατηρούμε κάποιες περίεργες ενδείξεις. Δείτε τις περιπτώσεις για την καμπύλη 3 και την καμπύλη 5. Η τιμή της επαναληπτικής μεθόδου δίνει σχεδόν διπλάσια τιμή για την απόσταση σε αυτές τις καμπύλες. Το πιθανότερο πρόβλημα ίσως εντοπίζεται στα όρια των δεικτών για τις επαναλήψεις των συναρτήσεων. Ίσως σε συγκεκριμένες καμπύλες με συγκεκριμένη δομή η επανάληψη τελειώνει πρόωρα ή κάποιο τελευταία κουτιά δεν

λαμβάνονται υπόψιν. Ας δοκιμάσουμε και τις 10 επόμενες ώστε να δούμε αν εντοπίζουμε και εκεί κάποια σημαντική απόκλιση:

Curve 19:

```
Frechet distance classic = 0.01215025
Frechet distance recursive = 0.01570795
```

Curve 20:

```
Frechet distance classic = 0.00813002
Frechet distance recursive = 0.01204789
```

Υπάρχουν ακόμη περίεργα αποτελέσματα. Μετά από πληθώρα δοκιμών, εντοπίσαμε το εξής πρόβλημα. Δείτε την παρακάτω συνάρτηση:

```
1 def get_reachable_vertical_border(i, j, a, b, d):
2     if j + 1 >= len(a) #or i >= len(b):
3         return [1, 0]
4     start, end = intersection_interval(b[i], d, a[j], a[j + 1])
5     return [start + j, end + j]
```

Listing 2.35: This function caused out of index problems.

Η συνθήκη $i \geq \text{len}(b)$ δεν υπήρχε στην αρχική έκδοση του κώδικα και αυτός επέστρεφε σφάλμα για τις περίεργες καμπύλες:

```
start, end = intersection_interval(b[i], d, a[j], a[j + 1])
IndexError: list index out of range
```

Για τον λόγο αυτό, στην έκδοση που παρουσιάσαμε εδώ προσθέσαμε την παραπάνω συνθήκη για το i . Ο αλγόριθμος πλέον έτρεχε, αλλά έβγαζε τα λάθος αποτελέσματα που παρατηρούμε. Τα δεδομένα λοιπόν έχουν ως εξής: Προσθέσαμε μια συνθήκη η οποία επέτρεψε στον αλγόριθμο να τρέξει για κάποιες συγκεκριμένες περιπτώσεις, οι οποίες όμως τώρα είναι λάθος. Και δημιουργείται το ερώτημα μήπως η συνθήκη πρέπει να επιστρέφει κάτι διαφορετικό. Για τον σκοπό αυτό, χωρίζουμε τις δυο συνθήκες:

```
1 def get_reachable_vertical_border(i, j, a, b, d):
2     if j + 1 >= len(a):
3         return [1, 0]
4     if i >= len(b):
```

```

5     return [1, 0]
6     start, end = intersection_interval(b[i], d, a[j], a[j + 1])
7     return [start + j, end + j]

```

Listing 2.36: We have split the two conditions.

Δοκιμάζουμε τώρα, διαφορετικές επιστροφές για την συνθήκη if $i \geq \text{len}(b)$. Δοκιμάζοντας τις $[0,1]$, $i = \text{len}(b) - 1$ και $[0,0]$, τα αποτελέσματα ήταν ακριβώς ίδια με αυτά που παρουσιάσαμε εδώ. Ωστόσο, προσπαθώντας να σπάσουμε τον κώδικα δοκιμάσαμε κάποιες ακραίες τιμές όπως $[10,10]$ και $[20,20]$. Το περίεργο είναι πως αυτές οι ακραίες τιμές όχι μόνο δεν έσπασαν τον κώδικα, αλλά φαίνεται διόρθωσαν το αποτέλεσμα! Η τελική μορφή της συνάρτησης είναι η παρακάτω:

```

1 def get_reachable_vertical_border(i, j, a, b, d):
2     if j + 1 >= len(a):
3         return [1, 0]
4     if i >= len(b):
5         return [15, 15]
6     start, end = intersection_interval(b[i], d, a[j], a[j + 1])
7     return [start + j, end + j]

```

Listing 2.37: Corrected border function.

η οποία δείτε τώρα τα αποτελέσματα που επέστρεψε για τις 3 και 5:

Curve 3:

```

Frechet distance classic = 0.00566988
Frechet distance recursive = 0.00504462
-----

```

Curve 5:

```

Frechet distance classic = 0.00529723
Frechet distance recursive = 0.00529723
-----

```

Τα πράγματα έχουν βελτιωθεί αρκετά μέσω αυτής της περίεργης επιστροφής. Τι μπορούμε να πούμε για αυτές τις μικρές αποκλίσεις που συνεχίζουμε να βλέπουμε; Στον επαναληπτικό αλγόριθμο έχουν εισαχθεί αρκετά προσεγγιστικά στοιχεία που μειώνουν την τελική ακρίβεια. Ίσως αυτά επηρεάζουν σε κάποιο βαθμό. Από την άλλη, μόνο σε μερικές καμπύλες υπάρχει απόκλιση. Φαίνεται δηλαδή το αποτέλεσμα να εξαρτάται από την μορφή της καμπύλης σε συνάρτηση με τα βήματα του κώδικα. Αφήνουμε στους έμπειρους θεωρητικούς και πειραματικούς να βρουν την απάντηση σε αυτές τις ερωτήσεις και στο γιατί η παραπάνω περίεργη επιστροφή βοήθησε. Είδαμε όμως και ένα χρήσιμο τρόπο διόρθωσης του κώδικα χωρίς να είναι γνωστή η ακριβής αιτία του σφάλματος και της λύσης.

Τέλος, η χειρότερη πολυπλοκότητα για τον επαναληπτικό αλγόριθμο είναι επίσης $O(nm \log(nm))$ όπως στην κλασσική περίπτωση, αλλά πρακτικά πολύ μικρότερος. Θα συγκρίνουμε τους δυο χρόνους στην επόμενη ενότητα.

Κεφάλαιο 3

Χρόνος εκτέλεσης των αλγορίθμων

3.1 Σύγκριση των δυο αλγορίθμων για την απόσταση Fréchet

Επόμενο βήμα είναι ο υπολογισμός του χρόνου εκτέλεσης των αλγορίθμων που είδαμε στα προηγούμενα κεφάλαια που είναι και το σημαντικό της υπόθεσης. Ο βασικός στόχος της επίπονης και δύσκολης διαδικασίας της συγγραφής του επαναληπτικού αλγορίθμου και των φίλτρων για την απόσταση Fréchet έγινε ώστε να μειωθεί ο υπολογιστικός χρόνος. Η χρονική ανάλυση λοιπόν, είναι αρκετά σημαντική και θα μας δείξει το δρόμο της επιτυχίας ή της αποτυχίας.

Στο αρχικό στάδιο θα χρησιμοποιήσουμε την ίδια βάση δεδομένων που χρησιμοποιήθηκε στο κεφάλαιο 2. Για τον σκοπό αυτό δείτε την αντίστοιχη ενότητα. Επιπλέον, η καμπύλη βάση της οποίας θα γίνει η αναζήτηση είναι επίσης η ίδια. Θα αλλάξουμε στη συνέχεια τις καμπύλες αυτές για μεγαλύτερη ποικιλία αποτελεσμάτων.

Είναι εύκολο τώρα να προσθέσουμε μερικές χρονικές εντολές στον ήδη έτοιμο κώδικα. Ας δούμε πως:

```
1 classic_total = 0
2 recursive_total = 0
3 total = 10
4 k = extract_curve("query_dataset.csv", 0) # Query curve
5 for i in range(total):
6
7     P = extract_curve("curve_dataset.csv", i) # Target curve
8
```

```

9      start = time.process_time()
10     frechet = frechet_distance(k, P)
11     end = time.process_time()
12
13     classic_total = classic_total + (end - start)
14     #print(f"Curve {i}: Classic time: {classic_total:.6f}
15         seconds")
16
17     start = time.process_time()
18     distance = frechet_distance_recursive(k, P)
19     end = time.process_time()
20
21     recursive_total = recursive_total + (end-start)
22     #print(f"Curve {i}: Recursive time: {recursive_total:.6f}
23         seconds")
24
25     print(f"Curve {i+1}:")
26     print(f"   Frechet distance classic   = {frechet:.8f}")
27     print(f"   Frechet distance recursive = {distance:.8f}")
28     print("-" * 40)
29
30 print("-" * 60)
31
32 print(f"Average time for classic   : {(classic_total / total):.6f}
33     seconds")
34 print(f"Average time for recursive: {(recursive_total /
35     total):.6f} seconds")

```

Listing 3.1: Calculate the runtime of both the classic and recursive functions.

Περικλείουμε κάθε μια από τις συναρτήσεις με εντολές για τον υπολογισμό του χρόνου. Ο υπολογισμός θα γίνει ως ο μέσος όρος του χρόνου για όλες τις 10 καμπύλες. Προσέξτε τις εντολές με σχόλια. Αυτές τυπώνουν τον χρόνο για κάθε μια καμπύλη ξεχωριστά. Χρησιμοποιήσαμε αυτούς τους ξεχωριστούς χρόνους ώστε να ελέγχουμε αν υπάρχουν ακραίες περιπτώσεις. Μερικά από τα αποτελέσματα φαίνονται παρακάτω:

```

Curve 0: Classic time: 0.156250 seconds
Curve 0: Recursive time: 0.031250 seconds
-----
Curve 1: Classic time: 0.265625 seconds
Curve 1: Recursive time: 0.046875 seconds
-----
Curve 2: Classic time: 0.328125 seconds
Curve 2: Recursive time: 0.078125 seconds

```

Αφού δεν υπάρχουν ακραίες συμπεριφορές, μπορούμε τώρα να δούμε τον μέσο

χρόνο που δίνει ως έξοδο ο παραπάνω κώδικας:

```
Average time for classic : 0.103125 seconds
Average time for recursive: 0.031250 seconds
```

Φαίνεται πως η δουλειά μας είχε αποτέλεσμα. Σε αυτές τις 10 καμπύλες με από 10 ως 30 σημεία, μειώσαμε τον υπολογιστικό χρόνο στο ένα τρίτο. Βέβαια, έχουμε πληρώσει κάτι σε ακρίβεια, αλλά διαφορές παρατηρούνται μόνο στο δεύτερο στο τρίτο και πολλές φορές στο τέταρτο σημαντικό ψηφίο.

Για όλες τώρα τις 100 καμπύλες της βάσης δεδομένων, αλλάζουμε τον δείκτη total από το 10 στο 100. Δείτε το αποτέλεσμα:

```
.....
Curve 99:
  Frechet distance classic = 0.00684632
  Frechet distance recursive = 0.00655988
-----
Curve 100:
  Frechet distance classic = 0.00682988
  Frechet distance recursive = 0.00669675
-----
-----
Average time for classic : 0.096719 seconds
Average time for recursive: 0.031094 seconds
```

Οπότε ο χρόνος είναι περίπου ο ίδιος, άρα όλες οι καμπύλες της βάσης δεδομένων συμπεριφέρονται ισοδύναμα.

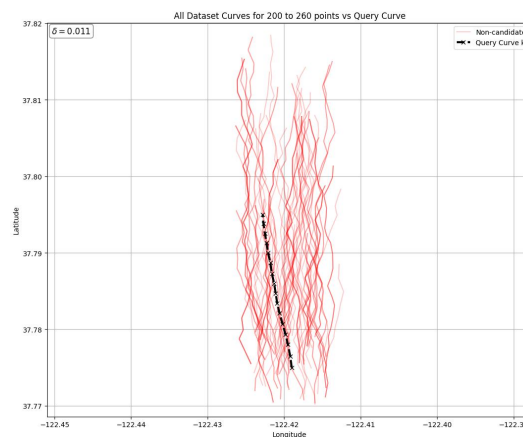
3.2 Συμπεριφορά για διαφορετικό μέγεθος καμπυλών

Ένα ενδιαφέρον στοιχείο τώρα, είναι η μελέτη της πολυπλοκότητας κάθε μεθόδου, κλασσικής και επαναληπτικής. Ο πρώτος δείκτης ως προς τον οποίο θα δούμε τον χρόνο εκτέλεσης για κάθε αλγόριθμο είναι ο αριθμός των σημείων για κάθε καμπύλη. Θυμηθείτε πως οι πολυγωνικές καμπύλες της βάσης δεδομένων που χρησιμοποιούσαμε έως τώρα είχαν μέγεθος από 10 ως 30 σημεία.

Μπορούμε εύκολα να μεταβάλλουμε τον αριθμό των σημείων, προσθέτοντας επιπλέον συντεταγμένες στο τέλος κάθε καμπύλης. Θα διατηρήσουμε την μορφή των καμπυλών, δηλαδή να είναι ρεαλιστικές, να ανταποκρίνονται σε διαδρομές εντός μιας μεγάλης πόλης και ο συνολικός τους αριθμός να είναι 100. Δημιουργούμε μια πληθώρα αρχείων καθένα από τα οποία θα έχει τα εξής χαρακτηριστικά:

- ▶ Μια βάση με καμπύλες από 10 έως 30 σημεία (αρχική βάση δεδομένων).
- ▶ Μια βάση με καμπύλες από 30 έως 60 σημεία.
- ▶ Μια βάση με καμπύλες από 60 έως 100 σημεία.
- ▶ Μια βάση με καμπύλες από 100 έως 150 σημεία.
- ▶ Μια βάση με καμπύλες από 150 έως 200 σημεία.
- ▶ Μια βάση με καμπύλες από 200 έως 260 σημεία.

Ας πάρουμε μια εικόνα για το πως μοιάζουν αυτές οι βάσεις δεδομένων. Σχεδιάζουμε την βάση δεδομένων με τα 60-100 σημεία.



Σχήμα 3.1: Οι καμπύλες της βάσης δεδομένων με τα 200 έως 260 σημεία.

Παρατηρήστε πως δεν υπάρχουν μεγάλες διαφορές σε σχέση με την βάση δεδομένων με τα 10 ως 30 σημεία. Τα νέα σημεία βρίσκονται πάνω στο παλιό μήκος των καμπυλών και άρα δεν θα χρειαστεί να αλλάξουμε την καμπύλη αναζήτησης και το δ .

Η βασική ιδέα είναι τώρα, να ζητήσουμε από κάθε αλγόριθμο να υπολογίσει τις αποστάσεις Fréchet μεταξύ καμπύλης αναζήτησης και καμπυλών βάσης δεδομένων και να εξάγουμε τον μέσο όρο, όπως και παραπάνω. Ας δούμε μερικά αποτελέσματα:

For curves between 30-60 points:

Total time for classic : 24.031250 seconds

Total time for recursive: 7.515625 seconds

Average time for classic : 0.240313 seconds

Average time for recursive: 0.075156 seconds

For curves between 60-100 points:

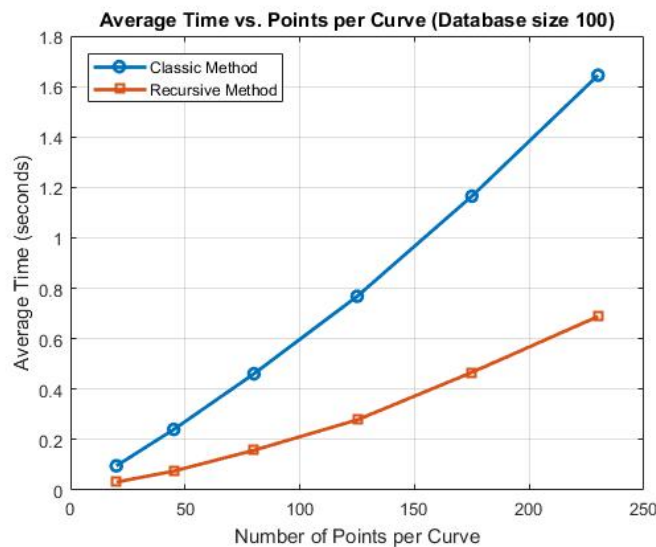
Total time for classic : 46.109375 seconds

Total time for recursive: 15.796875 seconds

Average time for classic : 0.461094 seconds

Average time for recursive: 0.157969 seconds

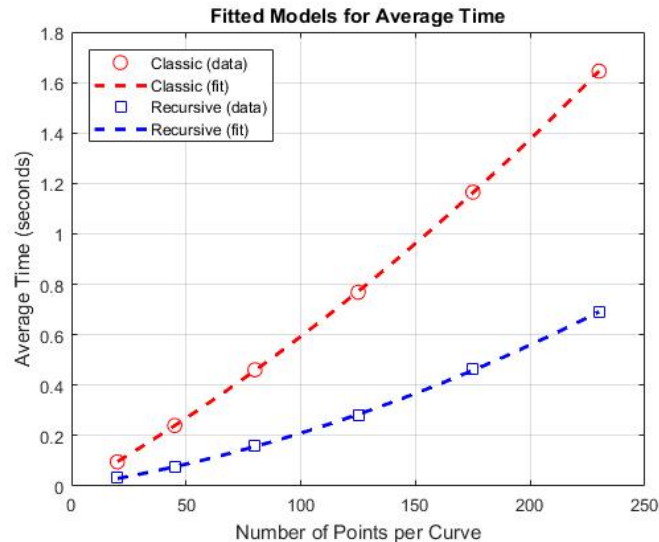
Η λογική γίνεται κατανοητή και για αυτό δεν θα εμφανίσουμε τα υπόλοιπα αποτελέσματα. Ο καλύτερος τρόπος να αποτυπωθούν αυτά είναι σε γραφική παράσταση. Θα σχεδιάσουμε λοιπόν την γραφική παράσταση για τον μέσο χρόνο που θέλει ο κάθε αλγόριθμος να βρει την απόσταση Fréchet, συναρτήσει του πλήθους των σημείων. Θα πάρουμε ως αριθμό σημείων τον μέσο όρο του διαστήματος που περιλαμβάνει κάθε βάση δεδομένων ώστε να έχουμε καλύτερη εικόνα για την συμπεριφορά της. Η γραφική παράσταση έχει την παρακάτω μορφή:



Σχήμα 3.2: Η γραφική παράσταση του μέσου χρόνου υπολογισμού της απόστασης Fréchet μεταξύ καμπύλης αναζήτησης και καμπυλών της βάσης δεδομένων σε σχέση με τα σημεία κάθε καμπύλης της βάσης δεδομένων.

Παρατηρούμε λοιπόν, πως ο χρόνος συνεχώς αυξάνει και μάλιστα με αυξανόμενο ρυθμό. Αυτό δεν μας κάνει ιδιαίτερη εντύπωση, αν θυμηθείτε την τετραγωνική

πολυπλοκότητα του κλασσικού αλγορίθμου, αλλά και του επαναληπτικού, σε θεωρητικό επίπεδο. Πρακτικά βέβαια, ο επαναληπτικό αλγόριθμος ακολουθεί μια πολύ πιο ήπια αύξηση στην πολυπλοκότητά του. Το χάσμα μεταξύ των δυο ανήγει συνεχώς. Για να είμαστε απόλυτα βέβαιοι, μπορούμε να προσαρμόσουμε ένα πολυώνυμο δευτέρου βαθμού στα δεδομένα. Το αποτέλεσμα φαίνεται παρακάτω:



Σχήμα 3.3: Προσαρμογή τετραγωνικού πολυωνύμου στα δεδομένα

με τα παρακάτω αποτελέσματα:

Classic fit: $y = 0.00001 x^2 + 0.00513 x + -0.00926$

Recursive fit: $y = 0.00001 x^2 + 0.00142 x + -0.00107$

R^2 for Classic fit: 0.999984

R^2 for Recursive fit: 0.999660

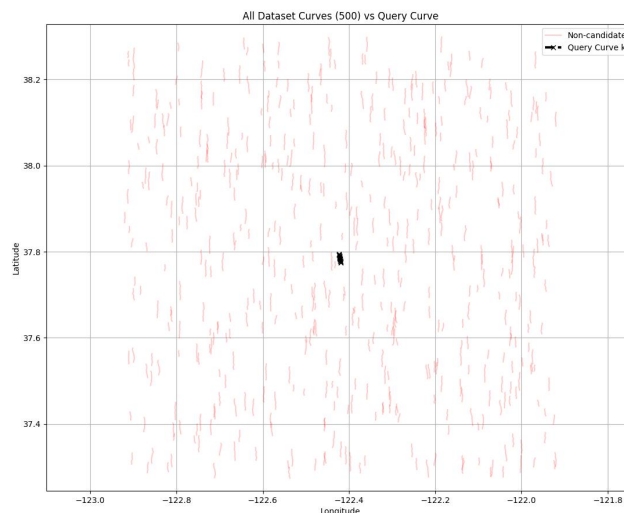
Μην σας προβληματίζει ο μικρός συντελεστής μπροστά από το x^2 . Η συνεισφορά αυτού του όρου ακόμη και για τα 230 σημεία είναι σχεδόν το 50% της εξίσωσης. Η μετρική των ελαχίστων τετραγώνων είναι 1 για την τέλεια προσαρμογή, άρα καταλαβαίνει κανείς πως η τετραγωνική προσαρμογή είναι πολύ κοντά στο πραγματικό αποτέλεσμα.

3.3 Συμπεριφορά για διαφορετικό μέγεθος βάσης δεδομένων

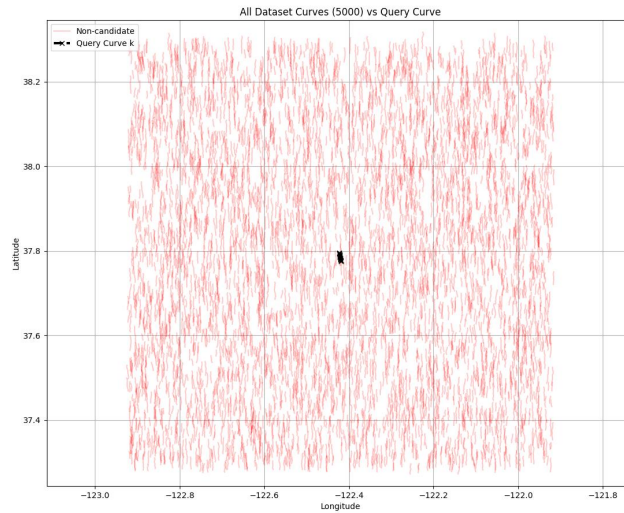
Σε αυτήν την ενότητα θα αλλάξουμε το μέγεθος της βάσης δεδομένων. Αντίθετα με ότι συνέβη προηγουμένως, όπου είχαμε 100 καμπύλες κάθε φορά με διαφορετικό πλήθος σημείων, τώρα τα σημεία θα παραμείνουν από 10 ως 30 όπως στην αρχική βάση αλλά θα προσθέσουμε επιπλέον καμπύλες στην βάση. Δημιουργούμε μια πληθώρα από αρχεία βάσης δεδομένων, καθένα από τα οποία θα έχει τα εξής χαρακτηριστικά:

- Μια βάση με 100 καμπύλες από 10 έως 30 σημεία (αρχική βάση δεδομένων).
- Μια βάση με 500 καμπύλες από 10 έως 30 σημεία.
- Μια βάση με 1000 καμπύλες από 10 έως 30 σημεία.
- Μια βάση με 3000 καμπύλες από 10 έως 30 σημεία.
- Μια βάση με 5000 καμπύλες από 10 έως 30 σημεία.
- Μια βάση με 10000 καμπύλες από 10 έως 30 σημεία.

Ας πάρουμε μια εικόνα για το πως είναι κατανομημένες οι καμπύλες στις παραπάνω βάσεις δεδομένων. Ας δούμε την βάση με 500 και την βάση με 5000 καμπύλες:

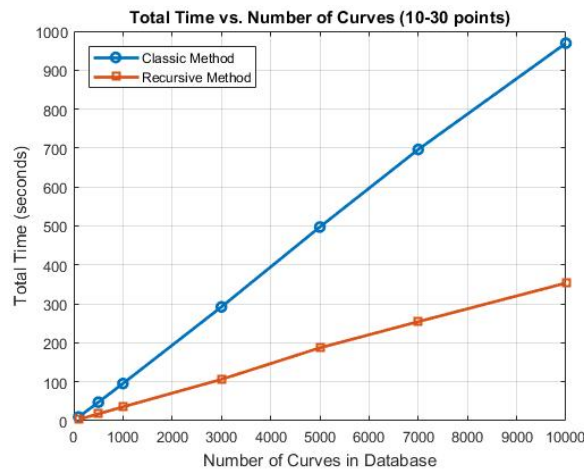


Σχήμα 3.4: Η γνωστή καμπύλη αναζήτησης και οι 500 καμπύλες της βάσης.



Σχήμα 3.5: Η γνωστή καμπύλη αναζήτησης και οι 5000 καμπύλες της βάσης.

Η διαδικασία είναι παρόμοια με την προηγούμενη ενότητα. Θα τρέξουμε τους αλγορίθμους για την απόσταση Fréchet για κάθε ζεύγος καμπύλης αναζήτησης και καμπύλης της βάσης και θα παρουσιάσουμε την γραφική παράσταση των αποτελεσμάτων. Το προσδοκώμενο αποτέλεσμα είναι μια γραμμική σχέση αυτή την φορά αφού δεν αλλάζουμε το πλήθος των σημείων των καμπυλών από το οποίο εξαρτάται η πολυπλοκότητα. Ας δούμε το αποτέλεσμα:



Σχήμα 3.6: Η γραφική παράσταση του συνολικού χρόνου εκτέλεσης σε σχέση με το πλήθος των καμπυλών.

Τα συμπεράσματα είναι τρία. Πρώτον, η συμπεριφορά είναι εμφανώς γραμμική άρα η μόνη πολυπλοκότητα προέρχεται από τον αριθμό των σημείων κάθε καμπύλης. Ο μέσος χρόνος μάλιστα για όλες τις περιπτώσεις είναι ο ίδιος με τον

μέσο χρόνο που είδαμε στην προηγούμενη ενότητα για την αρχική βάση δεδομένων με καμπύλες από 10 ως 30 σημεία. Δεύτερον, η απόσταση μεταξύ καμπύλης αναζήτησης και καμπύλης εξέτασης δεν επηρεάζει χρονικά το αποτέλεσμα. Αυτό γίνεται κατανοητό και από τον αλγόριθμο, αφού η μεγαλύτερη απόσταση σημαίνει μόνο διαφορετικά νούμερα στους υπολογισμούς. Τρίτον, θα χρησιμοποιήσουμε ένα από τα αρχεία με τις πολλές καμπύλες στη συνέχεια όπου θα κάνουμε ένα γενικό έλεγχο για τα φίλτρα.

3.4 Σύγκριση θετικού και αρνητικού φίλτρου

Θα επιχειρήσουμε τώρα να μετρήσουμε τον χρόνο εκτέλεσης των δυο φίλτρων αλλά και να καταλάβουμε το πόσο αποτελεσματικά είναι και αν επιστρέφουν τις ίδιες υποψήφιες καμπύλες. Δείτε τις αντίστοιχες ενότητες για την πλήρη περιγραφή των δυο φίλτρων. Ως βάση δεδομένων, θα χρησιμοποιήσουμε την βάση με τις 1000 καμπύλες από 10 έως 30 σημεία. Η καμπύλη αναζήτησης θα είναι η γνωστή που χρησιμοποιούμε έως τώρα. Δείτε τα σχήματα 3.4 και 3.5 για να δείτε το πως κατανέμονται οι καμπύλες στο χώρο. Ορίζουμε εδώ ένα $\delta = 0.25$ καθώς το εύρος έχει μεγαλώσει αρκετά.

Η λογική του πειράματος είναι η εξής: θα τρέξουμε το θετικό και το αρνητικό φίλτρο διαδοχικά ώστε να μας επιστρέψουν ποιές από τις 100 καμπύλες έχουν απόσταση Fréchet με την καμπύλη αναζήτησης μικρότερη του δ . Παράλληλα θα μετρήσουμε και τον χρόνο που τρέχει το κάθε φίλτρο.

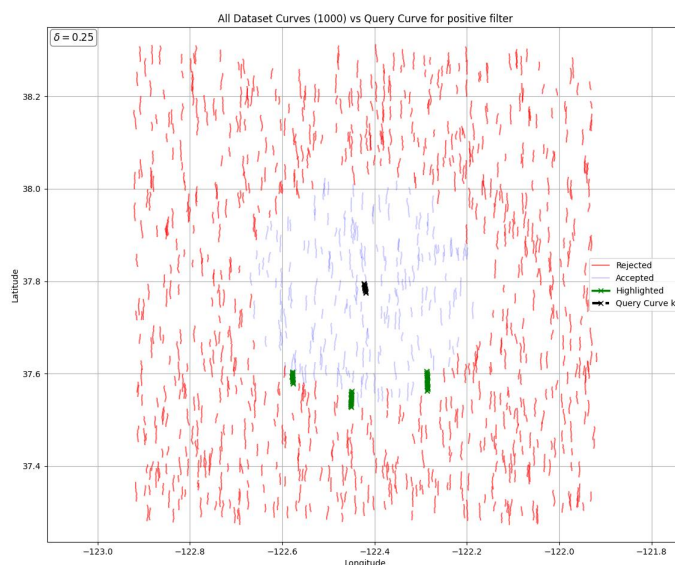
Ας δούμε τα πρώτα αποτελέσματα για τους χρόνους θετικού και αρνητικού φίλτρου:

```
Total number rejected by positive filter: 806
Total time to run positive filter 0.455823 seconds
```

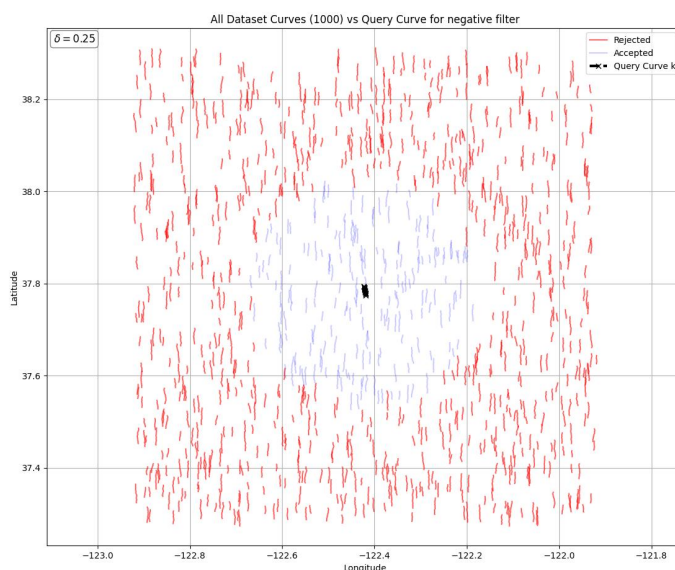
```
Total number rejected by negative filter: 803
Total time to run negative filter 0.294535 seconds
```

Παρατηρούμε τα εξής: το θετικό φίλτρο θέλει σχεδόν διπλάσιο χρόνο να τρέξει σε σχέση με το αρνητικό. Επιπλέον, τα αποτελέσματα των δυο φίλτρων είναι παρόμοια, εκτός από λίγες περιπτώσεις. Είδαμε και στα αντίστοιχα κεφάλαια πως αυτές είναι οριακές περιπτώσεις όπου η τιμή που υπολογίζουν τα φίλτρα είναι πολύ κοντά στην τιμή που έχουμε ορίσει για το δ .

Είναι όμως σωστή η κατανομή των καμπυλών που έχουν απορρίψει τα φίλτρα; Θα το διαπιστώσουμε μέσω των παρακάτω γραφικών:



Σχήμα 3.7: Οι καμπύλες που έχει αποδεχθεί (μπλε) και απορρίψει (κόκκινο) το θετικό φίλτρο. Με πράσινο οι 3 επιπλέον καμπύλες που έχει απορρίψει το θετικό φίλτρο. Πρόκειται για οριακές περιπτώσεις.



Σχήμα 3.8: Οι καμπύλες που έχει αποδεχθεί (μπλε) και απορρίψει (κόκκινο) το αρνητικό φίλτρο.

Δείτε με πράσινο τις επιπλέον καμπύλες που έχει απορρίψει το θετικό φίλτρο. Αποτελούν οριακές περιπτώσεις.

3.5 Τελική δοκιμασία: κλασσική μέθοδος vs κάτω φράγμα + φίλτρα + επαναληπτικός αλγόριθμος

Θα απαντήσουμε τώρα, στην ερώτηση του τίτλου και γενικά της εργασίας. Για μια καμπύλη αναζήτησης και για μια βάση δεδομένων, ποιές καμπύλες έχουν απόσταση Fréchet μικρότερη από ένα κατώφλι δ (ή ε). Για τον σκοπό αυτό θα χρησιμοποιήσουμε ξανά την βάση δεδομένων με τις 1000 καμπύλες από 10-30 σημεία.

Αρχικά, για την κλασσική μέθοδο θα τρέξουμε μια απλή εντολή for. Για κάθε καμπύλη θα υπολογίσουμε την απόσταση Fréchet και έπειτα θα την συγκρίνουμε με το κατώφλι. Για όσες είναι μικρότερη τις αποδεχόμαστε. Οι υπόλοιπες απορρίπτονται. Για την επαναληπτική μέθοδο, θα τρέξουμε πρώτα το τετραγωνικό δέντρο για το ελάχιστο όριο και μετά όλα τα διαδοχικά φίλτρα. Μόνο για όσες επιβιώσουν θα υπολογίσουμε αναλυτικά την απόσταση Fréchet.

Ο κώδικας για όλα αυτά φαίνεται παρακάτω:

```

1 curves_below_delta = []
2
3 # 1. Extract candidate curves once
4 candidate_curves = [extract_curve("generated_curves_1000.csv",
5                               idx) for idx in candidates]
6
7 # Track times if needed
8 pos_total = 0
9 neg_total = 0
10 recursive_total = 0
11 delta=0.25
12
13 exact_distance_needed = []
14 accepted_idxes = []
15 idxs = [] # rejected by neg_filter
16 reject = 0
17 accept = 0
18
19 # 2. Second filter: upper bound check
20 for i, idx in enumerate(candidates):
21     curve = candidate_curves[i]
22
23     start = time.perf_counter()
24     upper_bound = greedy_frechet_upper_bound(k, curve)
25     end = time.perf_counter()
26     pos_total += (end - start)
27
28     if upper_bound <= delta:
29         # Curve passes upper bound filter, add immediately
30         curves_below_delta.append(idx)

```

```

30     else:
31         # Needs exact check
32         exact_distance_needed.append(idx)
33
34 print(f"Total time for positive filter: {(pos_total):.6f}
    seconds")
35
36 # 3. Third filter: neg_filter on exact_distance_needed
37 for idx in exact_distance_needed:
38     candidate_curve = extract_curve("generated_curves_1000.csv",
    idx)
39
40     start = time.perf_counter()
41     boolean = neg_filter(k, candidate_curve, delta)
42     end = time.perf_counter()
43     neg_total += (end - start)
44
45     if boolean:
46         # Curve rejected early by neg_filter
47         idxs.append(idx)
48         reject += 1
49     else:
50         # Curve accepted by neg_filter
51         accept += 1
52         accepted_idxes.append(idx)
53
54 print(f"Total time for negative filter: {(neg_total):.6f}
    seconds")
55
56 # 4. Final exact Frechet distance check on accepted curves
57 for i in accepted_idxes:
58
59     P = extract_curve("generated_curves_1000.csv", i)
60
61     start = time.process_time()
62     distance = frechet_distance_recursive(k, P)
63     end = time.process_time()
64     recursive_total += (end - start)
65
66     if distance <= delta:
67         curves_below_delta.append(i)
68
69 print(f"Total time for recursive algorithm:
    {(recursive_total):.6f} seconds")
70 print(curves_below_delta)
71
72
73
74 print("-" * 60)
75
76 # Classic algorithm. Just a for command.

```

3.5. ΤΕΛΙΚΗ ΔΟΚΙΜΑΣΙΑ: ΚΛΑΣΣΙΚΗ ΜΕΘΟΔΟΣ VS ΚΑΤΩ ΦΡΑΓΜΑ + ΦΙΛΤΡΑ + ΕΠΑΝΑΛΗΠΤΙΚΟΣ ΑΛΓΟΡΙΘΜΟΣ

```
77 curves_below_delta = []
78 classic_total = 0
79 for i in range(1000):
80
81     P = extract_curve("generated_curves_1000.csv", i)
82
83     start = time.process_time()
84     #frechet = frechet_distance(k, P)
85     end = time.process_time()
86
87     classic_total += (end - start)
88
89     if frechet <= delta:
90         curves_below_delta.append(i)
91
92 print(f"Total time for classic algorithm: {(classic_total):.6f}
93       seconds")
94
95 print(curves_below_delta)
```

Listing 3.2: Full code to run our query.

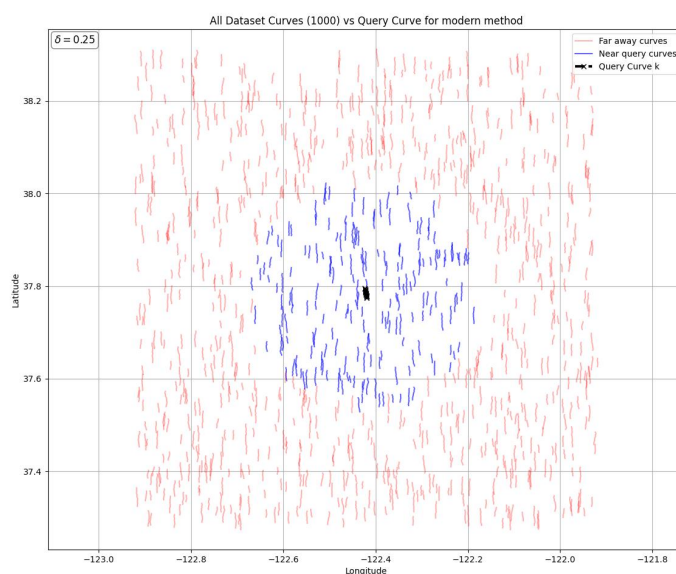
Τα αποτελέσματα φαίνονται παρακάτω:

```
Total time to create quad tree 0.165344 seconds
Total time to find candidates 0.330425 seconds
Total time for positive filter: 0.088091 seconds
Total time for negative filter: 0.007775 seconds
Total time for recursive algorithm: 0.093750 seconds
Total time for all actions: 1.478204 seconds
```

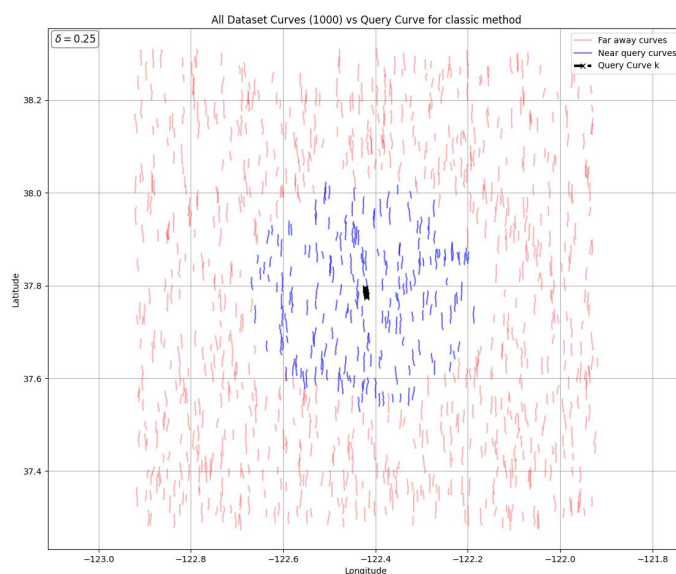
```
Total time for classic algorithm: 94.562500 seconds
```

Το αποτέλεσμα είναι εντυπωσιακό. Μαζί με τα φίλτρα, η σύγχρονη εκδοχή για τον υπολογισμό της απόστασης Fréchet συμβαίνει μόλις στο 1.56% του χρόνου για τον κλασσικό αλγόριθμο χωρίς κανένα φίλτρο. Όσο αφορά το ποιές καμπύλες επέστρεψαν οι δυο εκδοχές, ας δούμε τα παρακάτω γραφήματα:

Όπως φαίνεται παρακάτω, τα αποτελέσματα είναι ακριβώς τα ίδια. Με την σύγχρονη μέθοδο, έχουμε καταλήξει στο ίδιο αποτέλεσμα στο ένα εκατοστό του χρόνου σε σχέση με την κλασσική μέθοδο.



Σχήμα 3.9: Οι κοντινές καμπύλες στην καμπύλη αναζήτησης (μπλε) για την σύγχρονη μέθοδο και $\delta = 0.25$.



Σχήμα 3.10: Οι κοντινές καμπύλες στην καμπύλη αναζήτησης (μπλε) για την κλασική μέθοδο και $\delta = 0.25$.

3.6 Δοκιμή με πραγματικές καμπύλες

Θα επιχειρήσουμε τώρα να ξεφύγουμε από το ασφαλές περιβάλλον που έχουν θέσει έως τώρα με τις πλασματικές ρεαλιστικές καμπύλες και να δοκιμάσουμε τους αλγορίθμους σε πραγματικά δεδομένα. Επιλέγουμε ένα από τα πολλά πακέτα δεδομένων που κυκλοφορούν και έχουν σχέση με καμπύλες δεδομένων από GPS πορείες εντός πόλεων.

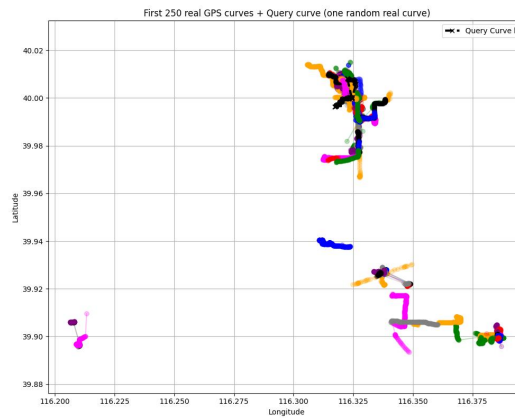
Στη συγκεκριμένη περίπτωση έχουμε επιλέξει το Geolife Trajectories. Σύμφωνα με τους δημιουργούς του πακέτου: αυτό το σύνολο δεδομένων τροχιάς GPS συλλέχθηκε στο πλαίσιο του προγράμματος Geolife (Microsoft Research Asia) από 182 χρήστες σε μια περίοδο άνω των πέντε ετών (από τον Απρίλιο του 2007 έως τον Αύγουστο του 2012). Μια τροχιά GPS αυτού του συνόλου δεδομένων αντιπροσωπεύεται από μια ακολουθία σημείων με χρονοσήμανση, καθένα από τα οποία περιέχει πληροφορίες για το γεωγραφικό πλάτος, το γεωγραφικό μήκος και το υψόμετρο. Αυτό το σύνολο δεδομένων περιέχει 17.621 τροχιές με συνολική απόσταση 1.292.951 χιλιόμετρα και συνολική διάρκεια 50.176 ώρες.

Από όλα τα παραπάνω εμείς θα κρατήσουμε τα δεδομένα που μας ενδιαφέρουν, δηλαδή σημεία γεωγραφικού μήκους και πλάτους σε ένα δισδιάστατο Ευκλείδειο χώρο. Έτσι, τα δεδομένα θα έχουν την ίδια μορφή με τα πλασματικά δεδομένα και άρα ο κώδικας που θα χρησιμοποιήσουμε δεν χρειάζεται αλλαγές. Το μόνο που έχουμε να κάνουμε είναι να επιλέξουμε το αρχείο με την καινούργια βάση δεδομένων. Ως καμπύλη αναζήτηση επιλέγουμε μια τυχαία από την βάση δεδομένων, την οποία στη συνέχεια αφαιρούμε βέβαια από την βάση. Μπορούμε λοιπόν, τώρα, να πάρουμε μια εικόνα για την μορφή των πραγματικών αυτών καμπυλών στις επόμενες εικόνες:

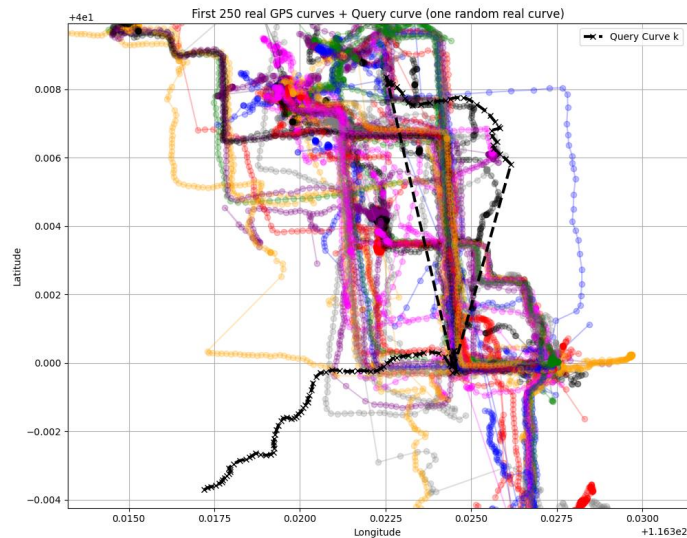
Έχουμε συνολικά 300 καμπύλες στην νέα βάση δεδομένων, κάθε μια εκ των οποίων έχει από 10 ως 705 σημεία, με μέσο όρο τα 156 σημεία. Συνεπώς, είμαστε χρονικά στο πάνω ράφι όσον αφορά την πολυπλοκότητα του κώδικα. Αυτό θα φανεί και από τους χρόνους που θα υπολογίσουμε.

Θα τρέξουμε κατευθείαν τον τελικό αλγόριθμο που συγκρίνει την κλασσική μέθοδο με όλες τις σύγχρονες μεθόδους, δηλαδή φίλτρα και επαναληπτικός αλγόριθμος. Πρέπει να επιλέξουμε κάποιο δ . Διαλέγουμε ένα μικρό δ ώστε να είμαστε εντός της πυκνής περιοχής που βρίσκεται η καμπύλη αναζήτησης. Η τιμή $\delta = 0.01$ είναι ότι πρέπει.

Πριν προχωρήσουμε, είναι απαραίτητο να προσθέσουμε κάποιες γραμμές στον κώδικα, ένα έλεγχο στη συνάρτηση interval-within-epsilon (δείτε τα listing 1.3 και



Σχήμα 3.11: Οι 150 (εκ των 300) πραγματικές καμπύλες σε σχέση με την καμπύλη αναζήτησης.



Σχήμα 3.12: Εδώ έχουμε μεγενθύνει στην περιοχή όπου βρίσκεται η καμπύλη αναζήτησης, η οποία φαίνεται με τα x ξεκινώντας από κάτω αριστερά.

1.4 σελίδα 25), σχετικά με το αν το a είναι ίσο με το μηδέν ή όχι, καθώς υπάρχει διαίρεση με το a . Στην πρώτη εφαρμογή του κώδικα, αυτό το σφάλμα χτύπησε. Προσθέτουμε επίσης εντολές οι οποίες εμφανίζουν στην κονσόλα την καμπύλη με την οποία δουλεύουμε. Λόγω της πολυπλοκότητας των καμπυλών, κάθε καμπύλη παίρνει κάποιον χρόνο και έτσι θέλουμε να είμαστε σίγουροι πως ο κώδικας τρέχει, ιδιαίτερα στην κλασσική περίπτωση.

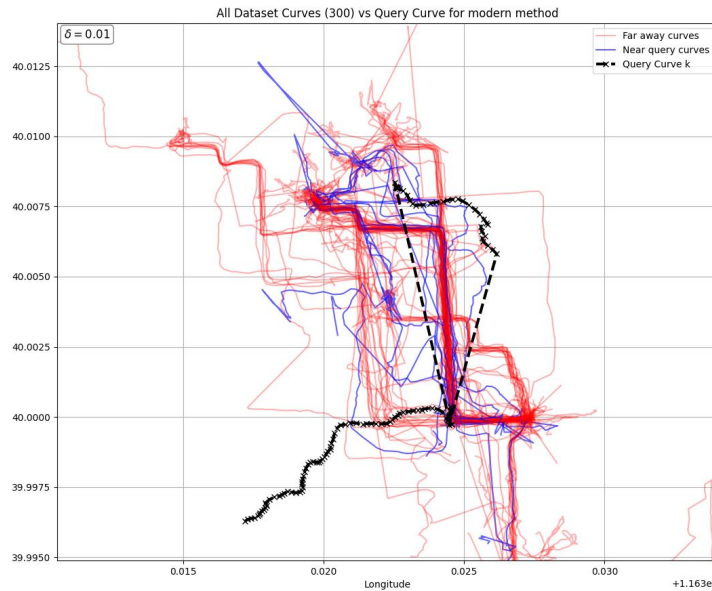
Ας δούμε όμως τα τελικά αποτελέσματα:

Results for real data

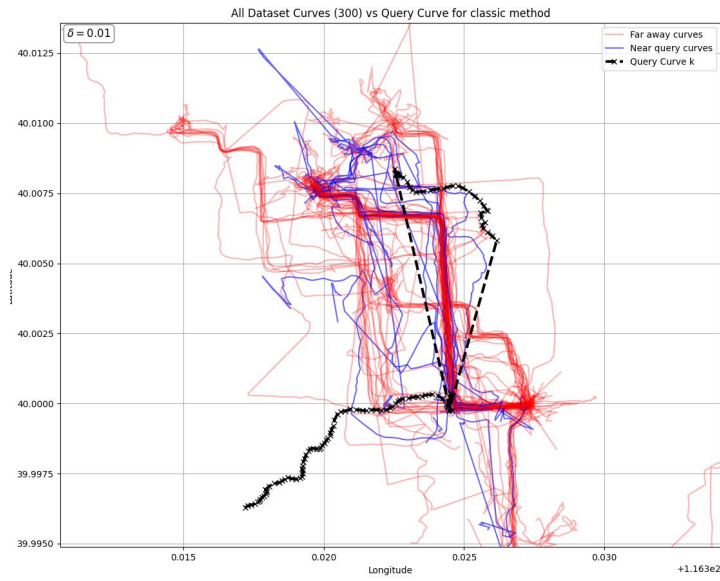
Total time to create quad tree 0.0632581 seconds
 Total time to find candidates 0.108194 seconds
 Total number of candidates: 245
 Total time for positive filter: 1.290312 seconds
 Total time for negative filter: 3.297736 seconds
 Total time for recursive algorithm: 460.890625 seconds (7.6 min)
 Valid Curves:
 [1, 128, 5, 20, 148, 242, 254, 119, 108, 147,
 133, 47, 180, 14, 7, 111, 17, 16, 150, 18]

Total time for classic algorithm: 3485.765625 seconds (58 min)
 Valid Curves:
 [1, 5, 7, 11, 13, 14, 16, 17, 18, 20, 56, 108, 111,
 119, 128, 133, 147, 148, 150, 189, 237, 242, 254]

Παρατηρείστε την δραματική αύξηση στον χρόνο εκτέλεσης και των δυο μεθόδων σε σχέση με τα πλασματικά δεδομένα. Από την άλλη, ο χρόνος εκτέλεσης για το τετραγωνικό δέντρο και τα φίλτρα έχει παραμείνει ελάχιστος. Οι δυο λίστες που εκτυπώνονται είναι οι δείκτες των καμπυλών της βάσης δεδομένων (δηλαδή η γραμμή στην οποία βρίσκονται) για τις οποίες η απόσταση Fréchet είναι μικρότερη του $\delta = 0.01$. Ας δούμε γραφικά για τις δυο περιπτώσεις το ποιές είναι οι καμπύλες αυτές:



Σχήμα 3.13: Οι καμπύλες για τις οποίες $d_f \leq \delta$ για την επαναληπτική μέθοδο.



Σχήμα 3.14: Οι καμπύλες για τις οποίες $d_f \leq \delta$ για την κλασσική μέθοδο.

Βλέπουμε την περιοχή της καμπύλης αναζήτησης σε μεγένθυση. Προφανώς, όλες οι υπόλοιπες καμπύλες σε διαφορετικές περιοχές του γραφήματος μακριά από το κέντρο, δεν περιλαμβάνονται σε κανένα αλγόριθμο. Αν λοιπόν, θεωρήσουμε τον κλασσικό αλγόριθμο ως τον απόλυτα ακριβή, έχουμε χάσει 3 καμπύλες στις 300 (οριακές περιπτώσεις βέβαια). Ωστόσο, έχουμε χρησιμοποιήσει μόλις το 1/10 του χρόνου. Σίγουρα η ανταλλαγή αυτή αξίζει για τους σκοπούς μας.

Κεφάλαιο 4

Βιβλιογραφία

1. Julian Baldus and Karl Bringmann. A fast implementation of near neighbors queries for Fréchet distance (GIS Cup), 2018.
2. Helmut Alt and Michael Godau. Computing the Fréchet distance between two polygonal curves. *Internat. J. Comput. Geom. Appl.*, 1995.
3. Karl Bringmann and Wolfgang Mulzer. Approximability of the discrete Fréchet distance. *JoCG*, 2016
4. Helmut Alt and Maike Buchin. Can we compute the similarity between surfaces? *Discrete and Computational Geometry*, 2010.
5. <https://www1.pub.informatik.uni-wuerzburg.de/pub/dijk/frechet/> (Link for the Fréchet distance app, Julius Maximilians, University of Wurzburg.)