Υπολογιστική Γεωμετρία - Εργασία Σεπτεμβρίου

Θεμελής Αλέξανδρος (ΑΜ: 1115201900062)

Σεπτέμβριος 2022

Υλοποίηση 1

1. Η υλοποίηση του αλγορίθμου περιτυλίγματος στο επίπεδο βρίσκεται στο convexHull.py στην συνάρτηση GiftWrapping_2D.

2. Η υλοποίηση του αυξητικού αλγόριθμου στο επίπεδο βρίσκεται στο αρχείο conveHull.py στην συνάρτηση Incremental.

   Λόγω έλλειψης χρόνου, δεν πρόλαβα να υλοποιήσω τον αυξητικό αλγόριθμο σε 3 διαστάσεις

3. Η χρονική πολυπλοκότητα του αλγορίθμου περιτυλίγματος είναι $\mathcal{O}(nh)$ με n το πλήθος των σημείων και h το πλήθος των κορυφών στο κυρτό περίβλημα. Η πολυπλοκότητα προέρχεται από την βασική ιδέα του αλγορίθμου δηλαδή να παίρνουμε κάθε ακμή του πολυγώνου που έχουμε καταλήξει ότι είναι στο κυρτό περίβλημα και να την συγκρίνουμε με τα άλλα σημεία.

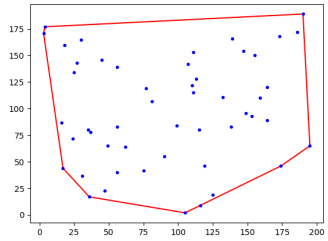
   Η χρονική πολυπλοκότητα του αυξητικού αλγόριθμου είναι $\mathcal{O}(nlogn)$. Η πολυπλοκότητα προέρχεται καθώς ο αλγόριθμος λειτουργεί με ένα divide & conquer στύλ. Δηλαδή, ταξινομούμε τα στοιχεία με $\mathcal{O}(nlogn)$ πολυπλοκότητα, έπειτα σπάμε στην μέση τα στοιχεία, φτιάχνουμε δύο (υπο)κυρτά περιβλήματα ελέγχοντας μέσω του κατηγορήματος προσανατολισμού, τι στροφή ορίζουν τα στοιχεία που επιλέγουμε. Τέλος, ενώνουμε τα 2 υποπροβλήματα, δηλαδή τα δύο κυρτά περιβλήματα και έτσι έχουμε το κυρτό περίβλημα του προβλήματος.

4. Ο αυξητικός αλγόριθμος μπορεί να γενικευτεί σε διαστάσεις d όπου $d > 3$. Ο τρόπος θα ήταν να αντιστοιχίσουμε τα ¨συστατικά' του χρωματικού αλγόριθμου σε d διαστάσεις. Για παράδειγμα, το κατηγόρημα του προσανατολισμού μπορεί εύκολα να γενικευτεί σε d διαστάσεις προσθέτοντας μια παραπάνω στήλη και γραμμή, κάνοντας τον πίνακα που βρίσκεται η ορίζουσα, $(d+1) \times (d+1)$.

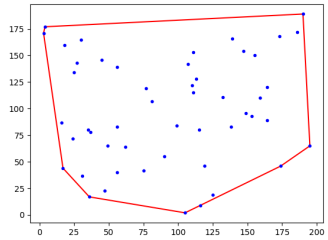
Εφαρμογή 1

1. (α′) Παρακάτω φαίνεται η εκτέλεση των αλγορίθμων θεωρόντας 50 τυχαία σημεία στο επίπεδο, με τα αποτελέσματα σαν λίστα σημείων αλλά και σχηματικά.

```
alexthem@alexthem-TM1703:~/Desktop/YpolGeo$ python3 convexHull.py
How many random points do you want to be generated? 50
The points of the convex Hull generated by incremental algorithm are
[[  3 171]
 [ 17  44]
 [ 36  17]
 [105   2]
 [116   9]
 [174  46]
 [195  65]
 [190 189]
 [  4 177]]
Incremental did 0.0004394054412841797 seconds
The points of the convex Hull generated by the GiftWrapping algorithm are
[[  3 171]
 [ 17  44]
 [ 36  17]
 [105   2]
 [116   9]
 [174  46]
 [195  65]
 [190 189]
 [  4 177]]
GiftWrapping_2D did 0.0034515857696533203 seconds
```



Αποτελέσματα μέσω του αυξητικού αλγόριθμου



Αποτελέσματα μέσω του αλγόριθμου περιτυλίγματος

(β′)

2. Για μικρό πλήθος σημείων δεν υπάρχουν ουσιαστικές διαφορές μεταξύ των 2 αλγορίθμων. Για παράδειγμα, για 10000 σημεία, και οι δύο αλγόριθμοι έχουν περίπου αντίστοιχο χρόνο εκτέλεσης, κάτω απο 1 δευτερόλεπτο. Για

μεγάλο πλήθος σημείων όμως, φαίνονται οι διαφορές στην πολυπλοκότητα του κάθε αλγορίθμου. Για παράδειγμα για πλήθος σημείων ίσο με 100000, ενώ ο αλγόριθμος περιτυλίγματος κάνει περίπου 53 δευτερόλεπτα, ο αυξητικός τελειώνει μόλις στα 3. Σημαντικό είναι να σημειωθεί ότι σε οποιοδήποτε πλήθος σημείων, η ποιότητα των αποτελεσμάτων δεν αλλάζει καθώς και οι 2 αλγόριθμοι επιστρέφουν την βέλτιστη λύση πάντα. Βέβαια, λόγο της δικιάς μου υλοποίησης, αν χρησιμοποιηθούν ακέραιες συντεταγμένες στα σημεία, ο αυξητικός αλγόριθμος δεν θα επιστρέψει τα σημεία που τέμνονται από ήδη υπάρχοντες γραμμές του κυρτού περιβλήματος, ενώ ο αλγόριθμος περιτυλίγματος θα επιστρέψει. Φυσικά, αυτό είναι λεπτομέρεια της υλοποίησης και μπορεί να αλλάξει ανάλογα με το τι βολεύει να έχουμε για την εφαρμογή μας.

3.

Υλοποίηση 2

1. Η υλοποίηση του αλγορίθμου βρίσκεται στο αρχείο kd.py. Η δημιουργία του kd-δέντρου ακολουθεί τον αλγόριθμο των σημειώσεων. Ανάλογα με το αν είναι άρτιος ή περιττός ο αριθμός του βάθους, ταξινομούνται τα σημεία ανάλογα με την τετμημένη ή τεταγμένη αντίστοιχα και καλέιται η συνάρτηση αναδρομικά, καλώντας ξεχωριστά το κάτω μισό και το άνω μισό των ταξινομημένων σημείων και ώς ρίζα του δέντρου (ή υποδέντρου το ενδιάμεσο στοιχείο).

2. Μπορούμε να φτάσουμε στο συμπέρασμα ότι η πολυπλοκότητα του αλγορίθμου ικανοποιεί την αναδρομική σχέση :

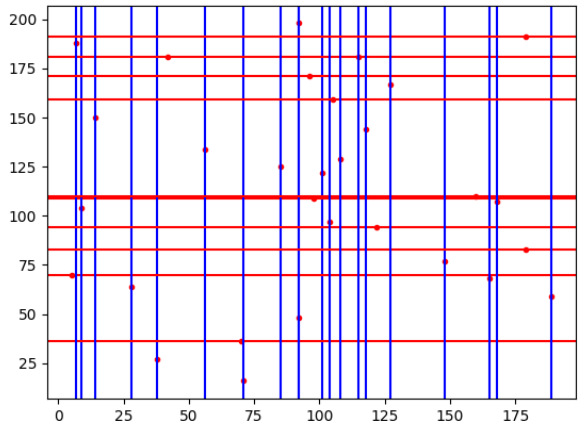
$$T(n) = \begin{cases} \mathcal{O}(1) & \text{αν } n = 1, \\ \mathcal{O}(n) + 2T(\lceil n/2 \rceil) & \text{αν } n > 1 \end{cases} \tag{1}$$

Η σχέση προέρχεται καθώς για $n = 1$ απλα επιστρέφουμε ένα σημείο άρα $\mathcal{O}(1)$. Αν $n > 1$, τότε μπορούμε να ταξινομήσουμε τα στοιχεία με $\mathcal{O}(nlogn)$ είτε να βρούμε την μέση με $\mathcal{O}(n)$ και έπειτα 2 αναδρομικές κλήσεις της $T(n)$ με μέγιστο $\lceil n/2 \rceil$ στοιχεία. Μέσω της θεωρίας αλγορίθμων, γνωρίζουμε ότι η παραπάνω αναδρομική σχέση έχει λύση $\mathcal{O}(nlogn)$.

Εφαρμογή 2

Το kd-δέντρο που παράχθηκε τρέχοντας τον αλγόριθμο που υλοποιήθηκε στο προηγούμενο ερώτημα είναι το εξής (το format του δέντρου είναι ότι χωρίζεται σε ένα tuple με 3 στοιχεία, το 1ο έχει το αριστερό υποδέντρο που είναι αντίστοιχο tuple με το αρχικό, αντίστοιχα το 2ο είναι το ίδιο για το δεξιά υποδέντρο και το τελευταίο στοιχείο είναι οι συντεταγμένες της ρίζας του δέντρου (ή υποδέντρου):

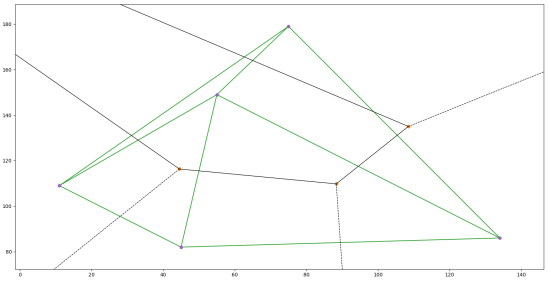
και η αντίστοιχη υποδιαίρεση του επιπέδου για το σύνολο αυτών των σημείων είναι η εξής (λόγω έλλειψης χρόνου δεν κατάφερα να υλοποιήσω στην απεικόνιση να μην υπάρχει όλη η οριζόντια/κάθετη γραμμή αλλά μόνο μέχρι την τομή της με την αντίστοιχα κάθετη/οριζόντια):



Ο κώδικας βρίσκεται στο αρχείο kd.py

Εφαρμογή 3

1. Η υλοποίηση του κώδικα βρίσκεται στο αρχείο voronoiDelaunay.py. Η υλοποίηση των διαγραμμάτων Voronoi και της τριγωνοποίησης Delaunay έγινε μέσω της βιβλιοθήκης SciPy. Παρακάτω φαίνεται η απεικόνιση μιας τριγωνοποίησης Delaunay και ενός διαγράμματος Voronoi για ένα σύνολο 5 σημείων στο επίπεδο. Μέσω του σχήματος που δημιουργήθηκε, μπορούμε να διακρίνουμε την αντιστοιχία μεταξύ της τριγωνοποίησης Delaunay και του διαγράμματος Voronoi. Πιό συγκεκριμένα, μπορούμε να παρατηρήσουμε ότι η τριγωνοποίηση Delaunay δεδομένου συνόλου εστιών στο επίπεδο είναι ο δυϊκός γράφος του διαγράμματος Voronoi των εστιών. Ο κώδικας βρίσκεται στο αρχείο voronoiDelaunay.py



2. Παρατηρούμε ότι η πολυπλοκότητα δημιουργίας και της τριγωνοποίησης Delaunay αλλά και του διαγράμματος Voronoi είναι πολυωνυμική και αρκέτα αποδοτική. Μπορούμε να παρατηρήσουμε ότι η πολυπλοκότητα τον αλγορίθμων είναι περίπου γραμμική (στην ακρίβεια η πολυπλοκότητα είναι $\mathcal{O}(nlogn)$ καθώς ο χρόνος περίπου 10πλασιάζεται κάθε φορά που δεκαπλασιάζουμε τον αριθμό του των σημείων. Ακόμα, μπορεί να διακριθεί και η άμεση σχέση μεταξύ τριγωνοποίησης Delaunay με το διάγραμμα Voronoi που τους συνδέει αυτή η σχέση δυϊκότητας που μοιράζονται καθώς οι χρόνοι εκτέλεσης και η πολυπλοκότητα τους είναι στην ουσία ίδια. Ενδεικτικά, κάτω είναι μια εκτέλεση της υλοποίησης με πλήθος σημείων ίσο με 1 εκατομμύριο.

```
alexthem@alexthem-TM1703:~/Desktop/YpolGeo$ python3 voronoiDelaunay.py 1000000
The Voronoi diagram took  5.500706195831299 seconds to be created with 1000000 number of points!
The Delaunay triangulation took  5.473406553268433 seconds to be created with 1000000 number of points!
```

Εφαρμογή 4 (Bonus)

Ένα σημαντικό πρόβλημα στον τομέα της πληροφορικής, ειδικά στην περιοχή της μηχανικής μάθησης, είναι το λεγόμενο classification problem. Η ουσία του προβλήματος είναι ότι προσπαθούμε να αναγνωρίσουμε ένα input, παρατηρόντας τα δεδομένα που ¨μοιάζουν' στο input μας. Στην ουσία, σχηματίζουμε κάθε υπάρχον δεδομένο (training set) με n στοιχεία στον $\mathbb{R}^n$ και έπειτα υπολογίζουμε την απόσταση με μία μετρική (να σημειωθεί ότι και η επιλογή της μετρικής είναι σημαντική) των σημείων αυτόν με το σημείο που θέλουμε να κατηγοροποιήσουμε.

Ο πιο απλός τρόπος που θα μπορούσαμε να το κάνουμε αυτό είναι μέσω μιας brute-force λογικής, στην οποία βρίσκουμε όλες τις αποστάσεις, τις ταξινομούμε και βρίσκουμε τους πιο κοντινούς ¨γείτονες' του input μας, παρατηρόντας σε ποια κατηγορία ανήκουν οι περισσότεροι γείτονες. Στην παρούσα υλοποίηση, δείχνουμε μέσω την χρήση kd-δέντρων ότι μπορούμε να επιτύγχουμε πολύ μεγάλες διαφορές πολυπλοκότητας και χρόνου εκτέλεσης, κατι πολύ σημαντικό καθώς στις πραγματικές εφαρμογές τα μεγέθη των δεδομένων ξεπερνούν κατα πολύ ακόμα και τα εκατομμύρια.

Στην εργασία έχει υλοποιηθεί ο αλγόριθμος knn (k-nearest-neighbours) και μέσω kd-δέντρο αλλά και με brute-force τρόπο ώστε να φανεί η μεγάλη διαφο-

ρά χρόνων εκτέλεσης. Πρέπει να σημειωθεί ότι για να υπάρξει η βελτίωση της πολυπλοκότητας, το kd-δέντρο πρέπει να προϋπάρχει, καθώς είναι κοστοβόρα η δημιουργία του. Φυσικά, αυτό μπορεί να γίνει καθώς μόνο μια φορά πρέπει να δημιουργηθεί το δέντρο στην αρχή με το training set και έπειτα θα μας απασχολεί μόνο κάθε καινούργιο σημείο που θα θέλουμε να κατηγοροποιήσουμε.

Ο κώδικας βρίσκεται στο αρχείο knn.py