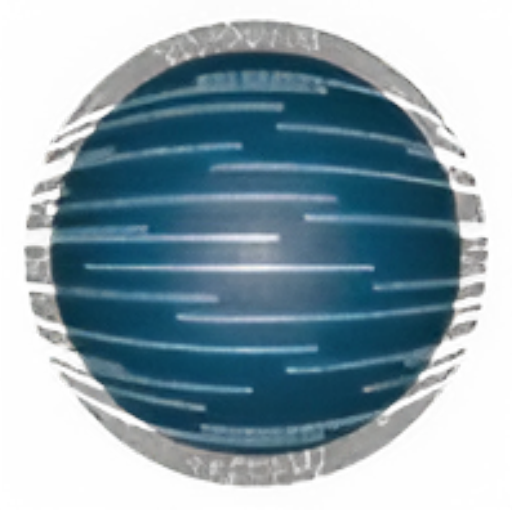
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΤΜΉΜΑ ΜΗΧΑΝΙΚΏΝ ΗΛΕΚΤΡΟΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΉΣ



Πολυδιάστατες Δομές Δεδομένων

Χειμερινό Εξάμηνο

Project Εξαμήνου

Καθηγητές: Σ. Σιούτας, Κ. Τσίχλας

ΑΜ	**Επώνυμο**	**Όνομα**	**Έτος**
1084567	Βιλλιώτης	Αχιλλέας	$3^ο$
1088098	Μπαρδάκης	Βασίλειος	$3^ο$
1090073	Πλάτωνας	Θεόδωρος	$3^ο$
1084589	Χάλλας	Χαράλαμπος-Μάριος	$3^ο$

ΠΆΤΡΑ, ΔΕΚΈΜΒΡΙΟΣ 2022

Περιεχόμενα

Μέρος A: 3D R-trees **3**
A.i Υλοποίηση . . . 3
A.ii Χρόνοι . . . 4

Μέρος B: Interval και Segment Trees **5**
B.i Segment Tree . . . 6
B.i.a Υλοποίηση . . . 6
B.i.b Σύγκριση με την θεωρία . . . 6
B.ii Interval Tree . . . 8
B.ii.a Υλοποίηση . . . 8
B.ii.b Σύγκριση με την θεωρία . . . 8
B.iii Δεδομένα . . . 8

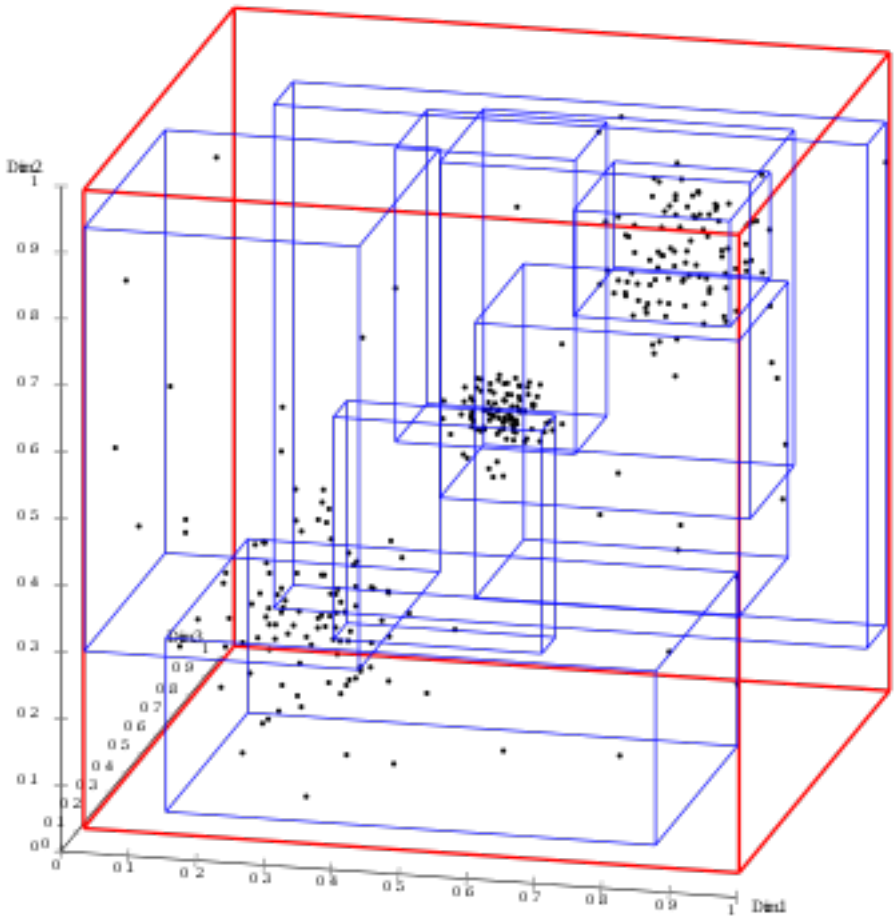
Μέρος C: Convex hull **9**
C.i Υλοποίηση . . . 9
C.ii Απόδειξη n*logn . . . 10
C.ii.a Εύρεση σταθεράς . . . 10

Μέρος D: Line Segment Intersection **11**
D.i Υλοποίηση . . . 11

Μέρος E: Παράρτημα **12**

Μέρος Α: 3D R-trees

Ένα R-tree είναι δομή δεδομένων που χρησιμοποιείται για την αποθήκευση και αναζήτηση γεωγραφικών αντικειμένων όπως σημεία ή περιοχές, βελτιώνοντας την απόδοση επερωτήσεων χωρικής αναζήτησης.



Ένα τρισδιάστατο R-tree (Πηγή urlwikipedia.org/wiki/R-tree)

A.i Υλοποίηση

Χρησιμοποιήσαμε την βιβλιοθήκη rtree της python με σκοπό την δημιουργία του rtree.[1] Γράψαμε ενα python script (data_maker.py) που δημιουργεί csv αρχείο με N αριθμό γραμμών με 2 χωρικές και μια χρονική μεταβλητή, το χρησιμοποιήσαμε για να μετρήσουμε την χρονική πολυπλοκότητα. Υλοποιήσαμε δύο προγράμματα python, το tree_grower.py χρησιμοποιεί αυτό το συνθετικό dataset του script καθώς το tree_grower2.py χρησιμοποιεί dataset πραγματικών δεδομένων το οποίο περιέχει διαδρομές ποδηλάτων. [2]

A.ii Χρόνοι

# Rows	Time in seconds
10	0.0 (μη-μετρήσιμο)
100	0.009984970092773438
1.000	0.01999378204345703
10.000	0.07003307342529297
100.000	0.4902318179072315

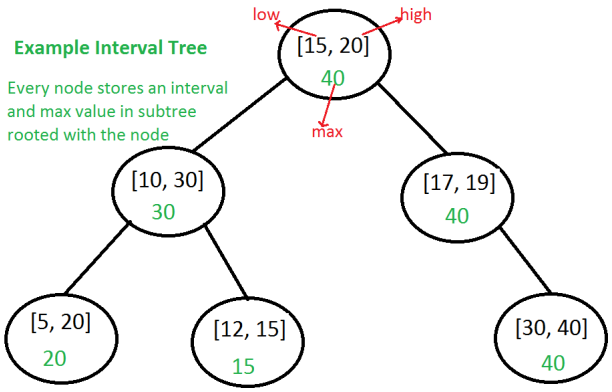
Table 1: Χρόνος του intersection ερωτήματος

# Rows	Time in seconds
10	0.00585484504699707
100	0.019006967544555664
1.000	0.1670389175415039
10.000	3.0737669467926025
100.000	66.73031044006348

Table 2: Χρόνος της υλοποίησης του δέντρου

Μέρος Β: Interval και Segment Trees

Τα Interval Trees είναι δομές δεδομένων που οργανώνουν διαστήματα και επιτρέπουν γρήγορη αναζήτηση, προσθήκη, και διαγραφή διαστημάτων, ενώ τα Segment Trees είναι δομές δεδομένων που διαχειρίζονται τμήματα σε έναν μεγάλο χώρο και υποστηρίζουν γρήγορες προσθαφαιρέσεις σε αυτά. Επιλέξαμε να υλοποιήσουμε τα δέντρα σε κώδικα C++ χωρίς την χρήση κλάσεων. Όμως σε περίπτωση που κάποιος θα ήθελε να χρησιμοποιήσει τον κώδικα ως βιβλιοθήκη η χρήση κλάσεων θα ήταν προτεινόμενη με στόχο την απόκρυψη κομματιών κώδικα.



Ένα Interval Tree με 6 nodes (Πηγή geeksforgeeks.org/interval-tree/)

B.i Segment Tree

B.i.a Υλοποίηση

Γενικότερα βρέθηκαν πολλαπλοί τρόποι υλοποίησης των segment trees όπως [3] [4] [5], εν τέλει επιλέχθηκε να γίνει η υλοποίηση που αναφέρεται στις διαφάνειες του μαθήματος. Η μετάφραση του ψευδοκώδικα από τις διαφάνειες ήταν απλή, ενώ κατά την κατασκευή του δέντρου δημιουργούνται μόνο τα nodes που απαιτούνται σε κάθε χρονική στιγμή με στόχο την εξοικονόμηση χώρου. Υλοποιήθηκαν insertions, deletions και queries.

B.i.b Σύγκριση με την θεωρία

Πετύχαμε περίπου $log^3 n$ απόδοση για το insertion και $logn + s$ για τα query. Συγκεκριμένα για το insertion, αυτή η χαμηλή σχετικά απόδοση μπορεί να οφείλεται μεταξύ άλλων, στο ότι δημιουργούνται `struct Node` μόνο όταν απαιτούνται, αντι να υπάρχει ολόκληρο το δέντρο από την αρχή, δηλαδή πρόκειται για trade-off χρόνου για χώρο, μπορεί επίσης να φταίει το O0 optimization. Όσον αφορά τα deletion, αφου αποτελούνται από το ίδιο κομμάτι κώδικα, ειναι αναμενόμενο να έχουν ίδια χρονική πολυπλοκότητα με τα query. Πράγματι, λόγω του μεγάλου s (1e6 intervals) σε μικρά range οπως [-3.6e1, 3.6e1] ο χρόνος είναι μεγάλος, αλλα έπειτα σταθεροποιείται. Παρακάτω παρατίθενται δεδομένα από test runs με παραμέτρους:

- 1.000.000 queries και 100 deletions.
- $[-n, n]$ range για τα interval, όπου n = 36, 360, .., $3,6e8$.
- Intel i5 4460, 16GB DDR3 1600MHz.
- Συντελεστής του αριθμού s ίσος με 100 (ώστε να προσεγγίσουμε την θεωρία).
- -O0 optimization.

Range Magnitude	Average Time (μs)	Constant Factor	Normalized C.F.
1e1	0,229	0,00097	100%
1e2	0,593	0,00069	71%
1e3	1,729	0,00082	85%
1e4	4,157	0,00099	102%
1e5	6,935	0,00094	97%
1e6	9,469	0,00128	133%
1e7	10,525	0,00059	61%
1e8	16,540	0,00065	67%

Table 3: Insertions, $ConstantFactor = time/(log^3 n)$

Range Magnitude	Average Time (μs)	Average s (Matches)	Constant Factor	Normalized C.F.
1e1	0,466	25.925	1,00000	100%
1e2	0,658	2.253	0,99996	100%
1e3	1,992	290,7	0,9963	100%
1e4	2,592	29,16	0,99537	100%
1e5	3,193	3,24	0,95256	95%
1e6	4,491	1,23	0,87470	87%
1e7	4.294	1,02	0,82902	83%
1e8	4,277	0,99	0,80403	80%

Table 4: Queries, $ConstantFactor = time/(log^3 n + 100 \times s)$, Matches means query interval matched.

Range Magnitude	Average Time (μs)
1e1	53436,530
1e2	1701,950
1e3	518,610
1e4	492,930
1e5	580,590
1e6	543,930
1e7	487,240
1e8	476,510

Table 5: Deletions, για 100 διαγραφές σε δέντρο με 1.000.000 κόμβους.

Τα πλήρη δεδομένα υπάρχουν στο παράρτημα 3, 4.

B.ii Interval Tree

B.ii.a Υλοποίηση

Επιλέχθηκε το Augmented Tree (Version 3 του powerpoint) [6], διότι θεωρήθηκε το πιο αποδοτικό και δυναμικό, λόγω του ότι είναι balanced. Δημιουργήθηκαν overloads για τις συναρτήσεις σύγκρισης ώστε να ακολουθείται ο τρόπος σύγκρισης που αναφέρεται στο μάθημα και στην συνέχεια έπρεπε παρά να γίνει μια επεξεργασία ενός AVL δέντρου (υπήρχε έτοιμος κώδικας που είχαμε δημιουργήσει στο μάθημα "Δομές Δεδομένων"). Υλοποιήθηκαν insertions, deletions και queries.

B.ii.b Σύγκριση με την θεωρία

Πετύχαμε περίπου $\sqrt[1.3]{n}$ απόδοση για το insertion και περίπου $\sqrt[1.35]{n}$ για τα deletion. Τα query είχαν υπεργραμμική απόδοση. Δυστυχώς δεν καταφέραμε να πιάσουμε καλές αποδόσεις. Ειδικότερα, γύρω στο n=100.000 η απόδοση εξαφανίζεται. Προσπαθήσαμε να βελτιώσουμε τις συναρτήσεις που χρησιμοποιούνται για το AVL δέντρο, χωρίς όπως να επιτύχουμε πολλά. Παρακάτω παρατίθενται δεδομένα από test runs με παραμέτρους:

- Μέχρι 100.000 queries.
- $[-3.6e7, 3.6e7]$ range για τα interval.
- Intel i5 4460, 16GB DDR3 1600MHz.
- 100 deletions κάθε φορά.
- -O3 optimization.

n	Average Time (μs)	C.F.	N.C.F.
600 700	4,732	0,500	100%
100.000	4.226,354	0,602	120%

Table 6: Insertions, οι αριθμοί προέκυψαν από τον χρονικό μέσο όρο για ∼100 insertions. $C.F. = time/\sqrt[1.3]{n}$

n	Average Time (μs)	C.F.	N.C.F.
1.000	29,3	0,00261	100%
10.000	696,3	0,00277	106%
100.000	15.149,1	0,00269	103%

Table 7: Deletions, οι αριθμοί προέκυψαν από τον χρονικό μέσο όρο για ∼100 deletions. $C.F. = time/\sqrt[1.35]{n}$

n	Average Time (μs)	C.F.	N.C.F.
1.000	0,942	7,48e-06	100%
10.000	55,479	8,79e-06	118%
100.000	1.731,81	5,48e-06	73%

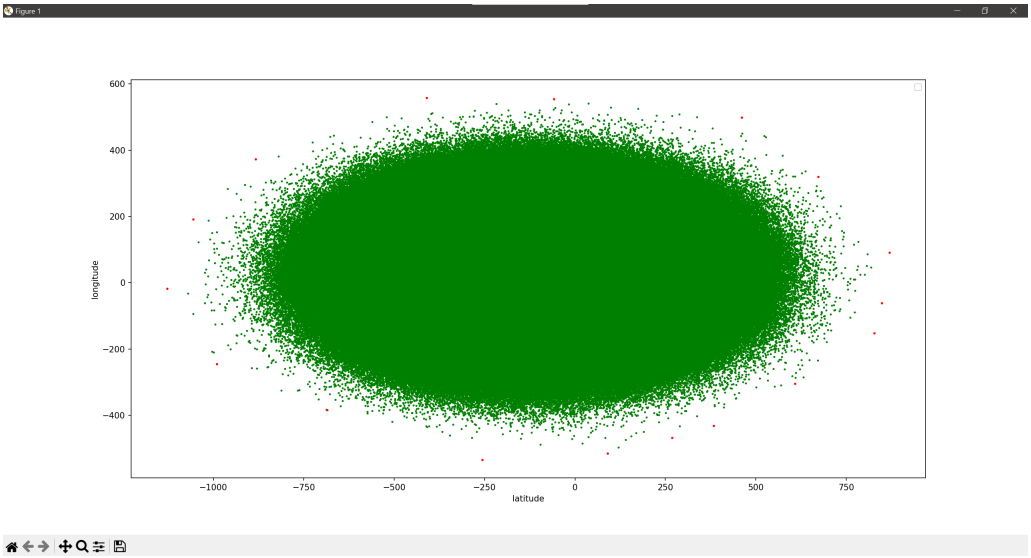
Table 8: Queries, οι αριθμοί προέκυψαν από τον χρονικό μέσο όρο για n queries. $C.F. = time/n^{1.7}$

B.iii Δεδομένα

Για τα Segment και Interval trees χρησιμοποιήθηκαν τεχνητά δεδομένα στο εύρος [-3.6, 3.6]xMagnitude, με την λογική πως θα μπορούσαν να αναπαριστούν χρόνο (πχ. χιλιοστά του δευτερολέπτου) για την δρομολόγηση ενός κομμάτι υλισμικού. Η παραγωγή δεδομένων έτσι βρίσκεται μέσα στο ίδιο αρχείο πηγαίου κώδικα και δεν απαιτείται η παραγωγή τους από τρίτο πρόγραμμα ή κάποιο dataset.

Μέρος C: Convex hull

Convex hull είναι το μικρότερο κυρτό πολύγωνο που περικλείει όλα τα δεδομένα σημεία, δηλαδή τα εξωτερικά σημεία του περιγράφουν ένα κυρτό πολύγωνο.



Έξοδος του graph.py

C.i Υλοποίηση

Ως ενδεικτικό dataset χρησιμοποιήσαμε το airbnb_listings_usa.csv [7], ένα σύνολο κρατήσεων απο airbnb στην Αμερική, με τις γεωγραφικές συντεταγμένες, την τιμή του καταλύματος και άλλες πληροφορίες σχετικά με την κάθε εγγραφή. Αφήσαμε μόνο τις στήλες longitude, latitude και price και δεν κάναμε καμία αλλοίωση τον δεδομένων.

Το parsing έγινε με έναν fast cpp csv parser απο github repo [8]. Υλοποιήσαμε δικό μας parser όμως η διαφορά χρόνων ήταν αρκετά μεγάλη.

Για το convex hull αναπτύξαμε ένα δικό μας graham scan, με βάση τη θεωρία των διαφανειών. Μια τεχνική λεπτομέρεια: Στη περίπτωση 3 συγγραμικών στοιχείων, διατηρείται μόνο το ψηλότερο (y coordinate) ή το χαμηλότερο στοιχείο, αναλόγως αν είναι left to right ή right to left scan. Να αναφερθεί ότι για πολύ πυκνά dataset (uniform) ενδέχεται να υπάρξουν λάθος αποτελέσματα, πιθανώς λόγω υπολογιστικών λαθών σε πολύ μικρούς αριθμούς. Για την - εν μέρει - αντιμετώπιση αυτού συγγραμικά δεν θεωρούνται δύο σημεια αν και μόνο αν η κλίση ισούται με μηδέν, αλλά υπάρχει ένα όριο γύρω απο το 0 (epsilon).

Επίσης, υλοποιήθηκε το graph.py, που με είσοδο points.csv (τα σημεία που διαβάστηκαν) και convex.csv (τα σημεία που ανήκουν στο convex hull), που είναι οι έξοδοι του convex hull, αναπαρίσταται γραφικά το σύνολο δεδομένων.

C.ii Απόδειξη n*logn

Για να αποδείξουμε πρακτικά την χρονική πολυπλοκότητα του convex hull, τροποποιήσαμε το πρόγραμμα ώστε να εκτελέι convex σε σταδιακά αυξανόμενα κομμάτια του dataset. Αφαιρέθηκαν πολλές εκτυπώσεις του προγράμματος και χρησιμοποιήθηκε νέο αρχείο εν ονόματι convexhull_benchmark.cpp Λόγω του μικρού αριθμού εγγραφών του airbnb listings, δημιουργήσαμε το παρακάτω python script, που δημιουργεί dataset τυχαίων τιμών longitude, latitude και price:

```
import os
import random

f = open(os.getcwd()+'\\airbnb_listings_usa.csv', mode='w')

f.write("latitude,longitude,price\n")
for j in range(30000000):
    lat = round(random.uniform(30, 100), 4)
    long = round(random.uniform(-120, 180), 4)
    rent = int(round(random.uniform(0, 3000), 0))
    output = str(lat)+","+str(long)+","+str(rent)+"\n"
    f.write(output)
```

Να αναφερθεί ότι κατά την εκτέλεση του, αντικαθίσταται το προηγούμενο airbnb_listings_usa.csv.

C.ii.a Εύρεση σταθεράς

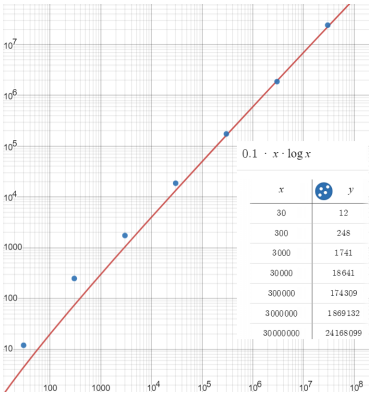
Διαιρώντας το χρόνο διά την χρονική πολυπλοκότητα που έχουμε υποθέσει, βρίσκουμε την σταθερά c του τύπου O(n) = c*n*logn. Παρατηρούμε πως η σταθερά παραμένει (σχεδόν) σταθερή σε κάθε τρέξιμο του convex hull, άρα πράγματι είναι n*logn πολυπλοκότητα.

# Rows (n)	Time in ns	$time/(n \times logn)$
30	1	0,022566416
300	17	0,022876016
3.000	194	0,01859776
30.000	2.399	0,01786118
300.000	29.752	0,018106835
3.000.000	339.095	0,017450911
30.000.000	3.678.704	0,016399823

Figure 1: Run 1, i3-1005G1, σταθεροποίηση κοντά στο 0,017

# Rows (n)	Time in ns	$time/(n \times logn)$
30	1	0,022566416
300	19	0,025567313
3.000	233	0,022336485
30.000	2.874	0,021397678
300.000	32.930	0,020040942
3.000.000	421.592	0,02169647
30.000.000	4.637.938	0,02067613

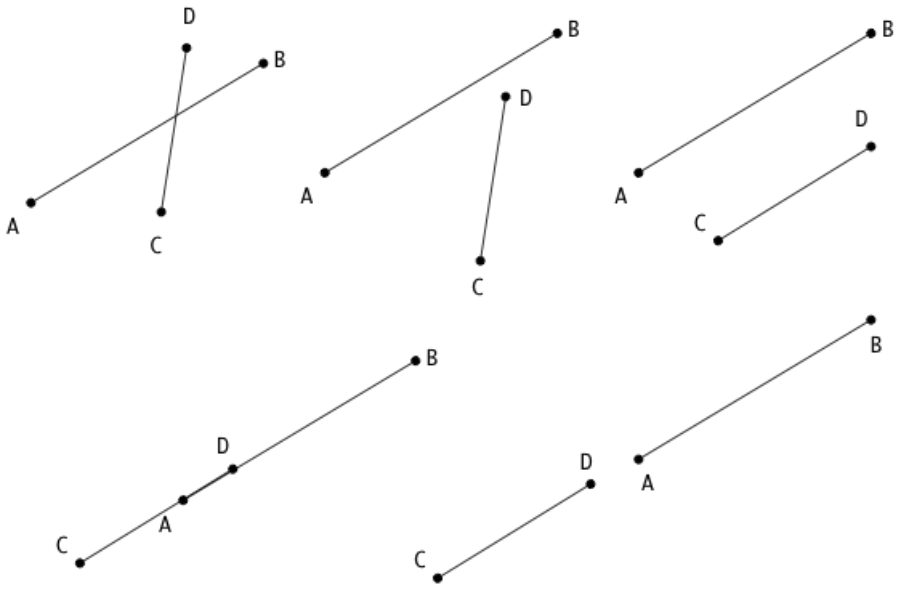
Figure 2: Run 1, i5-4460, σταθεροποίηση κοντά στο 0,02



Σύγκριση με λογαριθμική συνάρτηση

Μέρος D: Line Segment Intersection

Line Segment Intersection είναι η εύρεση των σημείων στον χώρο όπου δύο γραμμικά τμήματα τέμνονται, καθώς και η επιστροφή των τμημάτων ταυτόχρονα τέμνοντας. Χρησιμοποιείται σε πολλές εφαρμογές, όπως γραφικά υπολογιστών και γεωγραφικά συστήματα.



Γραμμικά τμήματα στον χώρο (Πηγή https://math.stackexchange.com/questions/149622/finding-out-whether-two-line-segments-intersect-each-other)

D.i Υλοποίηση

Για την υλοποίηση του Line Segment Inrestect χρησιμοποιήθηκε αλγόριθμος που βρέθηκε στο github [9] που δίνει ως output YES αν υπάρχουν τομές, αλλιώς NO. Ο αλγόριθμός τροποποιήθηκε ώστε να δέχεται input αρχείο .txt "LSIinput.txt", να αποθηκεύει τα αποτελέσματα σε αρχείο "LSIoutput.txt" και ακόμη να κρατάει και να εκτυπώνει το σημείο τομής στο "LSI.cpp" Για input κάναμε generate αρχείο txt με 200.000 cases της μορφής [X1 Υ1 X2 Υ2] σε "batches" των 1000, με εύρος X,Υ το [-100,100] . (κώδικας cpp με βοήθεια chatGPT) "LSItextGen.cpp"

Μέρος Ε: Παράρτημα

INSERTIONS								
avg(μs)	0,229	0,593	1,729	4,157	6,935	9,469	10,525	16,540
constant factor log2	0,03700	0,06236	0,13470	0,25732	0,35604	0,48615	0,40294	0,56177
normalized c.f. log2	100%	169%	364%	696%	962%	1314%	1089%	1518%
constant factor (log2)^3	0,00097	0,00069	0,00082	0,00099	0,00094	0,00128	0,00059	0,00065
normalized c.f. (log2)^3	100%	71%	85%	102%	97%	133%	61%	67%
constant factor sqrt	0,02680	0,02196	0,02023	0,01539	0,00812	0,01108	0,00123	0,00061
normalized c.f. sqrt	100%	82%	75%	57%	30%	41%	5%	2%
range (-3,6 to 3,6)	1E+01	1E+02	1E+03	1E+04	1E+05	1E+05	1E+07	1E+08

Figure 3: Segment Tree query πλήρη στατιστικά.

QUERIES																
avg(μs/results)	0,466	25925,33	0,658	2253,24	1,992	290,70	2,592	29,16	3,193	3,24	4,491	1,23	4,294	1,02	4,277	0,99
constant factor Cres=1	0,99978		0,99609		0,96428		0,70071		0,28310		0,23818		0,19566		0,17306	
normalized c.f. Cres=1	100%		100%		96%		70%		28%		24%		20%		17%	
constant factor Cres=10	0,99998		0,99961		0,99629		0,95593		0,68596		0,47886		0,39828		0,36030	
normalized c.f. Cres=10	100%		100%		100%		96%		69%		48%		40%		36%	
constant factor Cres=100	1,00000		0,99996		0,99963		0,99537		0,95256		0,87470		0,82902		0,80403	
normalized c.f. Cres=100	100%		100%		100%		100%		95%		87%		83%		80%	
range (-3,6 to 3,6)	1E+01		1E+02		1E+03		1E+04		1E+05		1E+06		1E+07		1E+08	

Figure 4: Segment Tree query πλήρη στατιστικά.

Βιβλιογραφία

[1] Cormen et al. Rtree: Spatial indexing for python. https://rtree.readthedocs.io/en/latest/, 2019.

[2] JEFFREY EMMONS. Cyclistic/divvy-data-aug2020-jul2021. https://www.kaggle.com/datasets/jeffreyemmons/cyclisticdivvydataaug2020jul2021?select=202102-divvy-tripdata.csv, 2021.

[3] Amit 'amitbansal7' Bansal. Data-structures-and-algorithms. https://github.com/amitbansal7/Data-Structures-and-Algorithms/tree/master/15.Segment%20Tree, 2017.

[4] GeeksforGeeks. Geeksforgeeks - segment tree. https://www.geeksforgeeks.org/segment-tree-data-structure/, 2023.

[5] Wikipedia et al. Wikipedia - segment tree. https://en.wikipedia.org/wiki/Segment_tree, 2023.

[6] Cormen et al. Section 14.3: Interval trees, pp. 348–354. https://en.wikipedia.org/wiki/Interval_tree#Augmented_tree, 2009.

[7] TAM LE. Airbnb usa listings. https://www.kaggle.com/datasets/tamle507/airbnb-listings-usa, 2022.

[8] Ben Strasser. Fast c++ csv parser. https://github.com/ben-strasser/fast-cpp-csv-parser, 2022.

[9] Jonathan-Uy. Cses-solutions. https://github.com/Jonathan-Uy/CSES-Solutions/blob/main/Geometry/Line%20Segment%20Intersection.cpp, 2023.