ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Πολυδιάστατες Δομές Δεδομένων

Project-1 Χειμερινού Εξαμήνου 2024

Διδάσκων: Σ. Σιούτας

Συντάκτες:

Ονομ/νο	**ΑΜ**	**Έτος**	**Email**
Χρήστος Καραμάνος	1072518	6ο	up1072518@ac.upatras.gr
Άγγελος Τσαμόπουλος	1072591	6ο	up1072591@ac.upatras.gr

Επιλέξαμε να υλοποιήσουμε το 1ο Project από τα δύο της εκφώνησης.

Περιβάλλον Υλοποίησης

Για την εκπόνηση της εργασίας χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python και το περιβάλλον του VSCode. Η δουλειά μοιράστηκε ισόποσα σε όλα τα μέλη.

Dataset

Ως dataset επιλέξαμε το “Used Cars Market Analysis:Scraped Data from Cars24” που βρήκαμε στην ιστοσελίδα Kaggle. Το συγκεκριμένο dataset περιέχει δεδομένα από το Cars24, μια δημοφιλή διαδικτυακή πλατφόρμα για την αγορά και πώληση μεταχειρισμένων αυτοκινήτων. Παρέχει λεπτομερείς πληροφορίες σχετικά με διάφορες καταχωρίσεις μεταχειρισμένων αυτοκινήτων.

Αποτελείται από τις εξής στήλες:

- **Model Name**
- **Price**
- **Manufacturing_year**
- **Engine capacity**
- **Spare key**
- **Transmission**
- **KM driven**
- **Ownership**
- **Fuel type**
- **Imperfections**
- **Repainted Parts**

Περιέχει 1446 εγγραφές που θεωρήσαμε πως είναι αρκετές για το σκοπό μας και το επιλέξαμε, πέρα των άλλων, διότι επιθυμούσαμε τα δεδομένα που θα διαχειριζόμασταν να σχετίζονταν με ένα αντικείμενο που θεωρούμε ενδιαφέρον.

Το link για το dataset: "https://www.kaggle.com/datasets/amanrajput16/used-car-price-data-from-cars24"

Α' Φάση

Αρχικά καλούμαστε να υλοποιήσουμε τις ζητούμενες δομές δεδομένων για διάσταση = k <= 4.

Εμείς επιλέξαμε k = 3, ώστε το χτίσιμο των δομών να μην είναι ιδιαίτερα απλοϊκό, αλλά και για να αναδείξουμε την διαστατικότητα των συγκεκριμένων δόμων. Τα 3 χαρακτηριστικά του dataset που περιέχει κάθε κόμβος από τις δομές μας είναι: **Price, Engine capacity, KM driven**.

Α1) Quad Trees

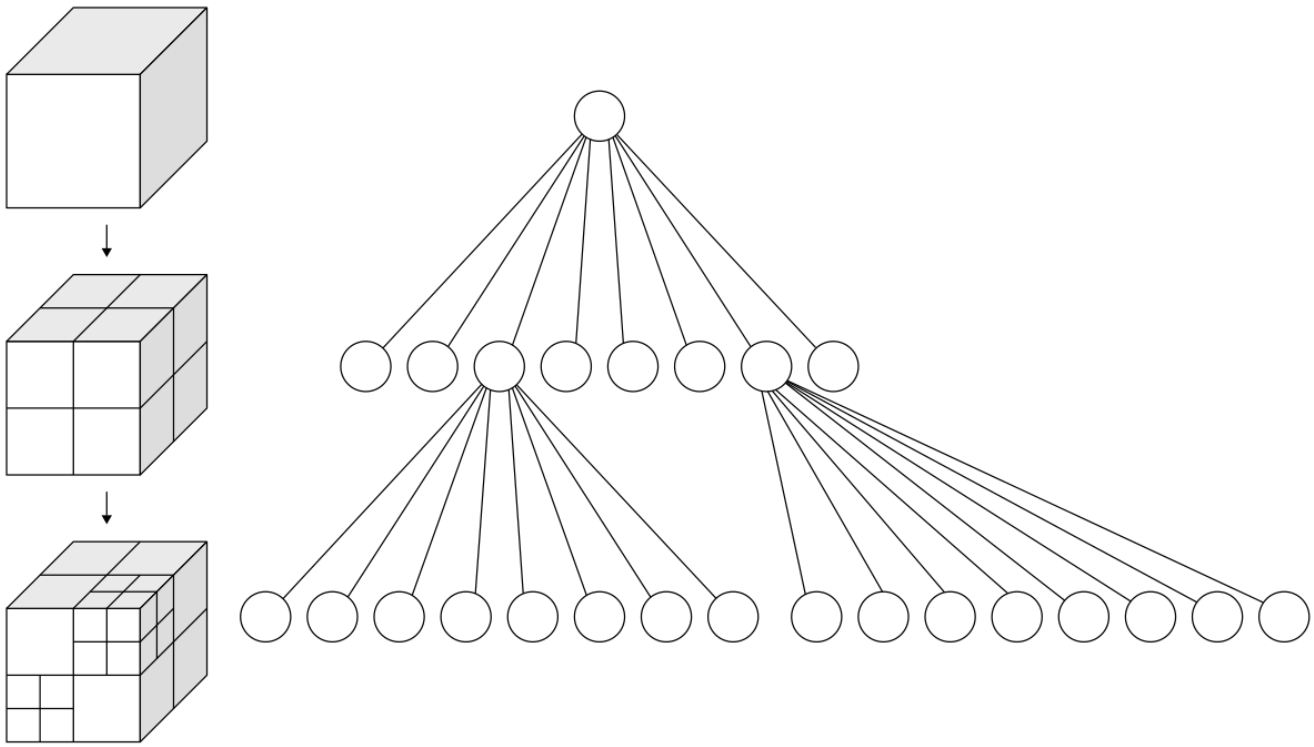
Το Quad Tree είναι δομή δεδομένων που εφαρμόζεται στον δισδιάστατο χώρο. Επειδή εμείς χρειαζόμαστε δομές στον τρισδιάστατο χώρο (αφού έχουμε τρία χαρακτηριστικά ανά αντικείμενο) θα υλοποιήσουμε μία "υποκατηγορία" του Quad Tree, το Octree.

Το Octree είναι μια ιεραρχική δομή δεδομένων που διαμερίζει αναδρομικά έναν τρισδιάστατο χώρο σε όμοιους υποκύβους, επιτρέποντας γρήγορη αποθήκευση και ανάκτηση χωρικών πληροφοριών. Στην κορυφή του – στον ριζικό κόμβο – βρίσκουμε έναν bounding cube, δηλαδή τον ελάχιστο κύβο που περιέχει όλα τα σημεία του dataset. Κάθε εσωτερικός κόμβος μπορεί να υποδιαιρεθεί σε οκτώ ίσα cubes (octants), δημιουργώντας έως και οκτώ δείκτες–παιδιά. Η διαδικασία υποδιαίρεσης συνεχίζεται αναδρομικά, μέχρι να εκπληρωθεί κάποιο κριτήριο τερματισμού, όπως για παράδειγμα η επίτευξη μέγιστου βάθους.

Υπάρχουν δύο κύριες εκδοχές του octree:

- Στο region octree, κάθε κόμβος αντιπροσωπεύει έναν όγκο–voxel στον χώρο, και υποδιαιρείται μόνο εφόσον ο voxel δεν είναι ομοιογενής (π.χ. περιέχει σημεία με διαφορετικές ιδιότητες).
- Στο point octree, κάθε κόμβος αποθηκεύει ένα μοναδικό σημειακό δείγμα (αντικειμένο) (x,y,z). Κάθε νέο σημείο συγκρίνεται με το κέντρο του τρέχοντος κόμβου και τοποθετείται αναδρομικά στο κατάλληλο octant (με βάση τριψήφιο δυαδικό δείκτη από τις συγκρίσεις σε x, y και z). Η κατανομή αυτή επιτρέπει γρήγορες εισαγωγές points - αντικειμένων στο δέντρο και αποδοτικά range queries σε χωρικά δεδομένα.

Στην παρούσα περίπτωση μας ενδιαφέρει η εκδοχή του point octree και η αναπαράσταση του octree ως δέντρο.



Εικόνα: Παράδειγμα αναπαράστασης Octree.

Αρχικά δημιουργούμε έναν Constructor με όνομα Node ο οποίος χρησιμοποιείται για την δημιουργία ενός κόμβου του δέντρου μας. Κάθε κόμβος έχει attributes (Model Name, Price, Engine capacity και KM driven) και οκτώ δείκτες, έναν για κάθε πιθανό παιδί του.

Στην συνέχεια, με την συνάρτηση InsertCar εισάγεται ένα αυτοκίνητο στην δομή μας ως εξής:

- Γίνεται έλεγχος για κενό δέντρο. Αν δεν υπάρχει ακόμα ριζικός κόμβος (δηλ. node είναι None), δημιουργείται ένας νέος κόμβος με τα δεδομένα του αυτοκινήτου και ορίζεται ως η ρίζα του δέντρου.
- Εξετάζονται τα χαρακτηριστικά Price, Engine capacity και KM driven του κόμβου προς εισαγωγή και συγκρίνονται με εκείνα του κόμβου που εξετάζεται εκείνη τη στιγμή. Μέσω διαδοχικών συγκρίσεων καθορίζεται σε ποιο από τα οκτώ πιθανά παιδιά του κόμβου που εξετάζεται (OOO, OOI, …, III) πρέπει να μπει το νέο αυτοκίνητο - κόμβος.
- Αν το παιδί που έχει επιλεγεί μέσω των συγκρίσεων είναι κενό (δηλαδή ουσιαστικά δεν υπάρχει), ο νέος κόμβος θα καταχωρηθεί εκεί. Αν όμως

υπάρχει ήδη κόμβος - παιδί, η ίδια διαδικασία επαναλαμβάνεται αναδρομικά (αναδρομική εκτέλεση της παρούσας συνάρτησης), συνεχίζοντας σε χαμηλότερα επίπεδα του δέντρου μέχρι να βρεθεί η κατάλληλη κενή θέση.

- Μόλις το νέο αυτοκίνητο ενταχθεί σε κάποιο φύλλο (ή δημιουργηθεί η ρίζα), η συνάρτηση επιστρέφει αναδρομικά όλους τους κόμβους της διαδρομής, διατηρώντας έτσι τη σωστή δομή του δέντρου.

Με αυτόν τον τρόπο κάθε νέο αυτοκίνητο τοποθετείται στο σημείο του Octree που αντιστοιχεί στη σχετική του θέση στον τρισδιάστατο χώρο που ορίζεται από τα χαρακτηριστικά Price, Engine capacity και KM driven.

Με την συνάρτηση RangeSearch πραγματοποιείται αναζήτηση όλων των κόμβων (αυτοκινήτων) που εμπίπτουν σε ένα τρισδιάστατο διάστημα τιμών που έχει ορίσει ο χρήστης (user query), με τον εξής τρόπο:

- Ο τρέχων κόμβος ελέγχεται πρώτα αν είναι None, και αν είναι τότε διακόπτεται η αναζήτηση και δεν επιστρέφεται κανένα αποτέλεσμα στο υποδέντρο που εξετάζεται εκείνη την στιγμή (με την εκάστοτε εκτέλεση της συνάρτησης).

- Ελέγχεται το αν ο κόμβος που εξετάζεται είναι εντός των ορίων που έχει θέσει ο χρήστης. Συγκρίνονται οι τρεις συντεταγμένες του κόμβου με τα εν λόγω όρια, και αν όλες οι τιμές εμπίπτουν στο δοσμένο εύρος, ο κόμβος προστίθεται στην λίστα CarsInRange.

- Η αναζήτηση περιορίζεται στα υποδέντρα που έχουν νόημα να εξεταστούν. Με άλλα λόγια, γίνεται “κλάδεμα” (pruning) στα υποδέντρα για τα οποία προκύπτει ότι σίγουρα δεν έχουν κόμβους που ανήκουν στο δοσμένο από τον χρήστη διάστημα τιμών.

Πρακτικά, ελέγχονται τα όρια του ερωτήματος σε σχέση με τα αντίστοιχα χαρακτηριστικά του κόμβου για να αποφασιστεί αν έχει νόημα να εξεταστεί το κάθε υποδέντρο.

Για κάθε ένα από τα οκτώ πιθανά παιδιά του κόμβου (OOO, OOI, …, III), εξετάζεται μία τριάδα συνθηκών (< ή >=) ώστε να καθοριστεί αν το υποδέντρο έχει νόημα να υποστεί περαιτέρω εξέταση / αναζήτηση ή όχι.

Μόνο τα υποδέντρα που είναι πιθανό να περιέχουν κόμβους - αντικείμενα εντός των ορίων του ερωτήματος εξετάζονται αναδρομικά (εκτελείται αναδρομικά η συνάρτηση RangeSearch με όρισμα την ρίζα κάθε τέτοιου υποδέντρου).

Στο τέλος της εκτέλεσης, η λίστα CarsInRange περιέχει όλα τα αυτοκίνητα, τα χαρακτηριστικά των οποίων ικανοποιούν ταυτόχρονα όλα τα όρια που έχει θέσει ο χρήστης.

Κατά την διάρκεια της εκτέλεσης του κώδικα, υπολογίζονται οι χρόνοι για την δημιουργία της δομής και για την εκτέλεση του range search, με σκοπό να χρησιμοποιηθούν αργότερα για ανάλυση και σύγκριση.

Παρακάτω παραθέτουμε και ένα παράδειγμα εκτέλεσης του κώδικα:

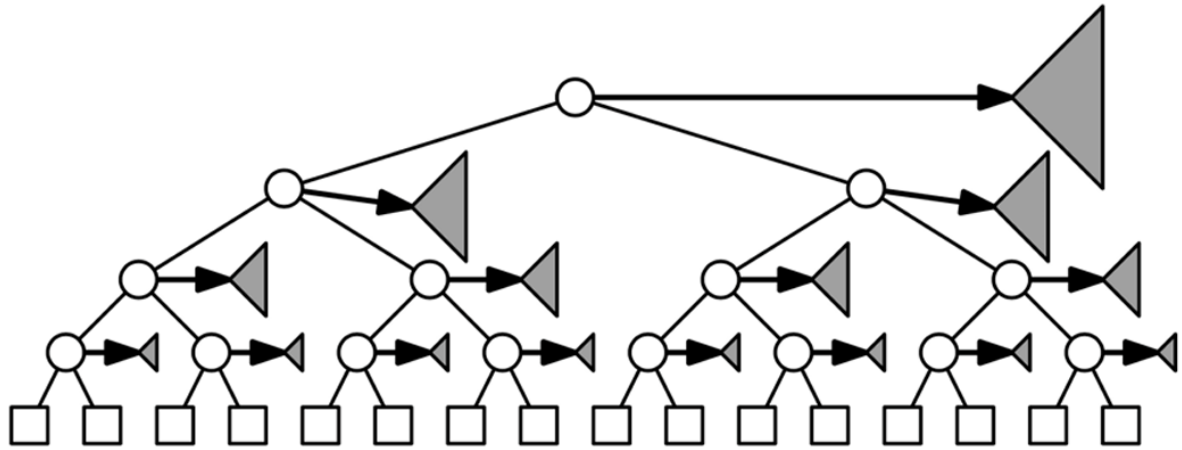
```
Total construction time: 0.008007049560546875 seconds



------------------------------------------------------------
User Query
Give the minimum price: 400000
Give the maximum price: 500000
Give the minimum engine capacity: 900
Give the maximum engine capacity: 1000
Give the minimum kilometers driven: 10000
Give the maximum kilometers driven: 20000
------------------------------------------------------------



Range search finished. Results:

['2018 Maruti Celerio X ZXI AMT', 494000, 998, 14268]
['2021 Maruti S PRESSO VXI+', 450000, 998, 10266]
['2016 Maruti Celerio VXI CNG', 426000, 998, 10347]
['2022 Maruti Alto K10 LXI', 423000, 998, 10776]
['2021 Maruti S PRESSO VXI+', 432000, 998, 12058]
['2023 Maruti Alto K10 VXI PLUS', 489000, 998, 10106]
['2022 Maruti S PRESSO VXI+', 476000, 998, 10815]
['2021 Maruti S PRESSO VXI+', 472000, 998, 10432]
['2020 Maruti S PRESSO VXI (O) AMT', 454000, 998, 13440]
['2022 Maruti S PRESSO VXI+', 455000, 998, 12690]
['2022 Maruti S PRESSO VXI+', 483000, 998, 13814]
['2023 Maruti Alto K10 LXI', 472000, 998, 16628]
```

A3) Range Trees

Τα Range Trees στηρίζονται στην αρχή των Φυλλοπροσανατολισμένων Ισοζυγισμένων Δυαδικών Δέντρων Αναζήτησης (BBST). Όταν πρόκειται για τρισδιάστατα δεδομένα, το 3D Range Tree αποτελείται από μια ιεραρχία πολλών BBST που συνεργάζονται για να δομήσουν τα δεδομένα στις τρεις διαστάσεις. Συγκεκριμένα, αν θεωρήσουμε ότι οι συντεταγμένες των σημείων είναι x, y και z, τότε το δέντρο ξεκινά με ένα BBST που ταξινομεί τις τιμές της διάστασης x. Για κάθε κόμβο αυτού του δέντρου, που

αντιστοιχεί σε μια τιμή x_i, κατασκευάζεται ένα δευτερεύον BBST το οποίο οργανώνει τις τιμές της διάστασης y για όλα τα σημεία με $x=x_i$, καθώς και για όλα τα σημεία που βρίσκονται στα φύλλα των υποδέντρων με ρίζα το x_i. Στη συνέχεια, κάθε κόμβος στο y-BBST συνδέεται με ένα ακόμη BBST που οργανώνει τις τιμές της διάστασης z με τον ίδιο τρόπο.

Στην παρακάτω εικόνα φαίνεται πως το Range Tree διαμορφώνεται όταν οι διαστάσεις είναι 2.



Εικόνα: Παράδειγμα αναπαράστασης 2D Range Tree (εικόνα από διαφάνεια μαθήματος)

Όσον αφορά τον κώδικα, αρχικά διαβάζουμε τα δεδομένα μας από το csv αρχείο και αποθηκεύουμε τις ζητούμενες στήλες σε μία λίστα αποτελούμενη από python tuples που με τη σειρά τους αποτελούνται από 3 τιμές: Price, Engine capacity, KM driven της κάθε εγγραφής.

Για το χτίσιμο του δέντρου δημιουργήσαμε 3 κλάσεις: RangeTree3D, RangeTree2D, RangeTree1D. Ξεκινώντας από τη **RangeTree3D** ο constructor μας δέχεται ως είσοδο τη λίστα με τα δεδομένα και αυτά ταξινομούνται κατά αύξουσα τιμή του x (price), ώστε να μπορέσουμε

εύκολα να βρούμε το median και να φτιάξουμε ισορροπημένο δέντρο. Στη συνέχεια, καλείται η "_build_recursive" για να ξεκινήσει η αναδρομική κατασκευή του κάθε κόμβου του δέντρου. Ο κάθε κόμβος αποτελείται από τα εξής πεδία:

- **value**: το σημείο (x,y,z)
- **range**: ελάχιστη και μέγιστη τιμή του x για το υποδέντρο με ρίζα τον τρέχοντα κόμβο (θα εξηγηθεί αναλυτικά μετέπειτα)
- **left** και **right**: το αριστερό και δεξί υποδέντρο του κόμβου (x<median, x>median αντίστοιχα) που κατασκευάζεται με αναδρομική κλήση της τρέχουσας κλάσης, αν αυτός δεν είναι φύλλο
- **yz_tree**: εμφωλευμένο yz-tree που προκύπτει από την κλήση του constructor της RangeTree2D κλάσης και δέχεται ως είσοδο τα σημεία του υποδέντρου με ρίζα τον τρέχοντα κόμβο

Για κάθε κόμβο λοιπόν του RangeTree3D κατασκευάζεται ένα υποδέντρο μέσω του **RangeTree2D**. Οι κύριες διαφορές του συγκεκριμένου υποδέντρου είναι ότι η ταξινόμηση των σημείων για την επιλογή του median για κάθε κόμβο γίνεται με βάση την τιμή y (Engine capacity) της τριάδας (x,y,z), το range του κάθε κόμβου καθορίζεται από την ελάχιστη και μέγιστη τιμή του του υποδέντρου του τρέχοντα κόμβου και για την υλοποίηση του εμφωλευμένου z-tree καλείται ο constructor της RangeTree1D.

Η **RangeTree1D** φτιάχνει ένα απλό BBST με βάση τη διάσταση z (KM driven). Να τονίσουμε ότι εδω ο median περιλαμβάνεται και στο δεξί υποδέντρο, καθώς στην αναζήτηση στο συγκεκριμένο δέντρο κάνουμε απλό φιλτράρισμα σε λίστα, ενώ στα προηγούμενα κάνουμε αναδρομική αναζήτηση με αποτέλεσμα να είναι πιθανό να επισκεφτούμε τον ίδιο κόμβο πάνω από μία φορά.

Σε αυτό το σημείο πρέπει να αναφέρουμε μία σημαντική έννοια των range trees και αυτή είναι τα **canonical subtrees**. Έστω ότι εκτελούμε ενα range search. Canonical subtree είναι ένα υποδέντρο του range tree όλα τα σημεία του οποίου βρίσκονται εξ'ολοκλήρου μέσα στο range της αναζήτησης μας για μία συγκεκριμένη διάσταση. Αυτό σημαίνει ότι για την αναζήτηση μας δε χρειάζεται να εξετάσουμε κάθε κόμβο του υποδέντρου ξεχωριστά, αφου γνωρίζουμε ότι όλοι ανήκουν στο range που ψάχνουμε και άρα τους επιστρέφουμε. Με αυτόν τον τρόπο μειώνεται το πλήθος των κόμβων που επισκεπτόμαστε και βελτιώνουμε το χρόνο αναζήτησης. Αυτό το σκοπό εξυπηρετεί το πεδίο range κάθε κόμβου: κρατάει το εύρος τιμών του υποδέντρου του με βάση την εκάστοτε διάσταση και χρησιμοποιείται στη διαδικασία της αναζήτησης.

Συνεχίζουμε με τις συναρτήσεις range_query_3D, range_query_2D και range_query_1D. Ξεκινώντας με τη **range_query_3D**, δέχεται ως είσοδο το κόμβο-ρίζα του x-tree (ο οποίος έχει εμφωλευμένο όλη τη δομή του x-tree) και τα ranges που ψάχνουμε για τις τρεις διαστάσεις (**x_range, y_range, z_range**). Αρχικά, ανακτά το εύρος τιμών x που καλύπτει το υποδέντρο του κόμβου (**range**) και από την εξέταση του προκύπτουν 3 περιπτώσεις:

1. Αν το range του κόμβου είναι υποσύνολο του x_range του ερωτήματος, τότε βρήκαμε canonical subtree και προχωράμε στο range_query_2D για το yz-tree.
2. Αν δεν υπάρχει καμία επικάλυψη ανάμεσα στο range του κόμβου και στο x_range, τότε επίστρεψε κενό
3. Αν υπάρχει μερική επικάλυψη, ελέγχει αν ο ίδιος ο κόμβος ανήκει στο εύρος του ερωτήματος και αν ναι τον προσθέτει στα αποτελέσματα. Έπειτα, συνεχίζει αναδρομικά την αναζήτηση στο αριστερό και δεξί υποδέντρο μέσω της range_query_3D και επιστρέφει τα αποτελέσματα.

Αν έχει κληθεί η **range_query_2D** σημαίνει ότι έχουμε βρει canonical subtree στην x διάσταση, οπότε και δέχεται ως είσοδο τον κόμβο-ρίζα του συγκεκριμένου υποδέντρου και τα y_range και z_range του ερωτήματος. Εκτελεί αντίστοιχη διαδικασία με τη range_query_3D, αλλά τώρα εξετάζοντας τις τιμές για τα y και z.

Η **range_query_1D** καλείται μόνο όταν έχει βρεθεί y-canonical subtree. Δέχεται μία λίστα από κόμβους και επιστρέφει μόνο αυτούς των οποίων το z ανήκει στο z_range, αφού έχουν προηγηθεί τα φίλτρα για τις x,y τιμές.

Τέλος, να σημειώσουμε δύο πράγματα:

- Παρά τα ονόματα των κλάσεων, κάθε δέντρο αποθηκεύει κόμβους που αποτελούνται από 3 τιμές (x,y,z). Έχει υλοποιηθεί έτσι η δομή, διότι κατά την διαδικασία "ψαξίματος" ελέγχουμε σε κάθε δέντρο αν ένας κόμβος ανήκει στα αποτελέσματα και αν ναι, τον επιστρέφει. Συνεπώς, πρέπει να κρατάει όλη την τριάδα των τιμών.
- Τα x_range, y_range και z_range δίνονται από το χρήστη μέσω terminal.

Παρακάτω παραθέτουμε και ένα παράδειγμα εκτέλεσης του κώδικα:

```
Building 3D range tree...
Build time for 3D Range Tree: 0.5167 seconds
Enter your 3D range query:
Price range (min max): 577000 718000
Engine capacity range (min max): 796 1197
KM driven range (min max): 47110 60017
Running 3D query...
Found 36 matching cars.
Printing the first 10:
2020 Maruti Baleno ZETA PETROL 1.2  | Price: 631000, Engine: 1197, KM: 55790
2021 Maruti New Wagon-R LXI CNG (O) 1.0 | Price: 585000, Engine: 998, KM: 51151
2020 Maruti Swift VXI               | Price: 577000, Engine: 1197, KM: 47110
2021 Maruti Celerio ZXI AMT         | Price: 592000, Engine: 998, KM: 48061
2017 Maruti Baleno DELTA CVT PETROL 1.2 | Price: 590000, Engine: 1197, KM: 49352
2021 Maruti Swift VXI               | Price: 589000, Engine: 1197, KM: 58463
2020 Maruti Swift VXI               | Price: 593000, Engine: 1197, KM: 49583
2018 Maruti Dzire ZXI AMT           | Price: 596000, Engine: 1197, KM: 50947
2020 Maruti New Wagon-R ZXI 1.2 AMT | Price: 598000, Engine: 1197, KM: 52423
2020 Maruti Baleno DELTA PETROL 1.2 | Price: 593000, Engine: 1197, KM: 54733
```

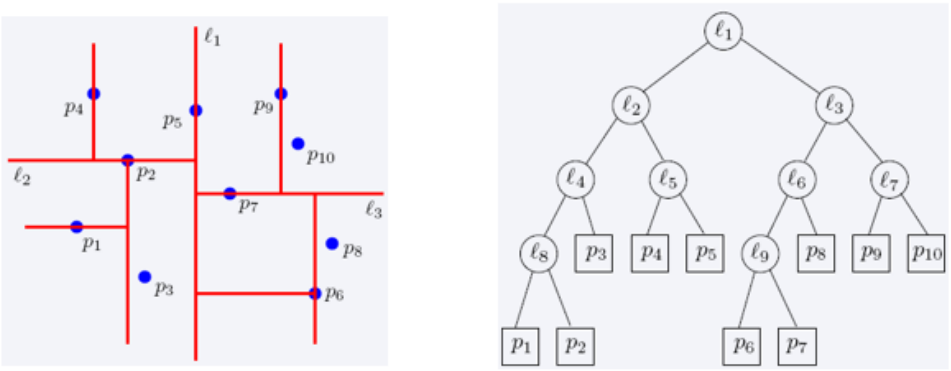
A4) KD-Trees

Τα k-d trees (k-dimensional trees) είναι δομές δεδομένων που χρησιμοποιούνται για την οργάνωση και την αποδοτική αναζήτηση σημείων σε έναν k-διάστατο Ευκλείδειο χώρο. Αποτελούν γενίκευση των δυαδικών δέντρων αναζήτησης (BST) για περισσότερες από μία διαστάσεις και είναι ιδιαίτερα χρήσιμα σε εφαρμογές όπου τα δεδομένα περιέχουν χωρική ή πολυδιάστατη πληροφορία. Οι πιο συνηθισμένες χρήσεις τους είναι για αναζήτηση πλησιέστερων γειτόνων (k-NN), ερωτήματα περιοχής (range queries), ταξινόμηση, σύσταση προϊόντων, πλοήγηση ρομπότ και υπολογιστική γεωμετρία.

Στην περίπτωσή μας το δέντρο χτίζεται πάνω σε τρεις διαστάσεις : Price, Engine Capacity, KM Driven. Η κατασκευή του kd-tree γίνεται αναδρομικά. Κάθε κόμβος στο δέντρο χρησιμοποιεί έναν άξονα (axis) για να διαχωρίσει τα δεδομένα, επιλέγοντας κυκλικά έναν από τους k διαθέσιμους. Τώρα που έχουμε 3 χαρακτηριστικά η σειρά είναι Price -

Engine – KM - επανάληψη. Το split γίνεται στον median της επιλεγμένης διάστασης, ώστε να υπάρχει σχετική ισορροπία μεταξύ του αριστερού και δεξιού υποδέντρου. Πιο αναλυτικά για το πώς πραγματοποιείται ο διαχωρισμός, σε έναν 2D χώρο με σημεία (x, y), το δέντρο εναλλάσσει τη διάσταση μεταξύ x και y, δημιουργώντας κατακόρυφες και οριζόντιες διαχωριστικές γραμμές. Σε 3D χώρο (όπως στην περίπτωσή μας), οι διαστάσεις μπορεί να είναι x (Price), y (Engine Capacity) και z (KM Driven), και τα splits γίνονται πάνω σε υπερεπίπεδα που είναι κάθετα σε έναν από αυτούς τους άξονες.

Στην παρακάτω εικόνα φαίνεται ο διαχωρισμός και η διαμόρφωση ενός kd-Tree σε 2 διαστάσεις.



Εικόνα: Παράδειγμα αναπαράστασης 2D kd-Tree (εικόνα από διαφάνεια μαθήματος)

Για την υλοποίηση του κώδικα ακολουθήσαμε την παραπάνω θεωρία που βασίζεται σε αναδρομή και σε διαχωρισμούς του πολυδιάστατου χώρου ανά άξονα. Το δέντρο κατασκευάζεται μέσω της κλάσης KDTreeCustom όπου αποτελεί τον πυρήνα της υλοποίησης. Στον

constructor της (_init_) περιλαμβάνεται όλη η λογική για την αναδρομική κατασκευή του δέντρου. Αρχικά, ελέγχεται αν υπάρχουν διαθέσιμα σημεία. Έπειτα καθορίζεται η διάσταση `axis` με βάση το βάθος (`axis = depth % k`), ταξινομούνται τα σημεία ως προς αυτήν τη διάσταση και επιλέγεται το σημείο στη θέση του median ως κόμβος. Η διαδικασία επαναλαμβάνεται για το αριστερό και δεξί υποσύνολο των δεδομένων, φτιάχνοντας το αριστερό και δεξί υποδέντρο. Η κυκλική εναλλαγή των αξόνων και η επιλογή του median επιτρέπουν ισορροπημένο και αποδοτικό δέντρο, σύμφωνο με τη θεωρία των k-d trees. Η κλάση KDNode είναι η βασική μονάδα του δέντρου και κάθε instance της περιέχει:

- Το σημείο `point`
- Τον άξονα διαχωρισμού `axis` (x,y,z)
- Τα υποδέντρα `left` και `right`.

Ο κόμβος λειτουργεί σαν αποθηκευτής δεδομένων αλλά και ως φορέας γεωμετρικής πληροφορίας για το που βρίσκεται στον χώρο.

Για την επιλογή του άξονα ευθύνεται η γραμμή (axis = depth % k) ορίζει ποια διάσταση χρησιμοποιείται για διαχωρισμό σε κάθε επίπεδο του δέντρου. Έτσι διασφαλίζεται η κυκλική αλλαγή των αξόνων. Στην συνέχεια το data.sort(key=lambda x: x[axis]) ταξινομεί τα σημεία σύμφωνα με τον επιλεγμένο άξονα, και το median = len(data) // 2 επιλέγει το "κέντρο" ως κόμβο. Έτσι, το δέντρο παραμένει **ισοζυγισμένο**, γεγονός που μειώνει σημαντικά τη μέση περίπτωση χρόνου αναζήτησης.

Για την αναπαράσταση του δέντρου χρησιμοποιούμε την μέθοδο print_tree() που είναι μια αναδρομική συνάρτηση, ξεκινά από τη ρίζα του δέντρου (node=None σημαίνει ξεκίνα από self.node). Για κάθε κόμβο εκτυπώνει:

- Την κατεύθυνση του (ROOT / LEFT / RIGHT),
- Το βάθος του κόμβου στο δέντρο,
- Τον άξονα στον οποίο έγινε το split (axis),

- Το σημείο (point).

Σύμφωνα με την θεωρία πρέπει να εμφανίζονται κυκλικά οι άξονες, στην περίπτωση μας δεν φαίνεται πάντα η κυκλικότητα των αξόνων. Παρότι η εναλλαγή των αξόνων γίνεται κυκλικά βάσει του βάθους (με axis = depth % k), αυτό δεν είναι πάντα εμφανές στην εκτύπωση της δομής μέσω της print_tree(). Ο λόγος είναι ότι κάθε υποδέντρο μπορεί να έχει διαφορετικό βάθος, ανάλογα με τον αριθμό και την κατανομή των δεδομένων που περιέχει. Αν κάποιο υποδέντρο περιέχει λίγα σημεία, τότε σταματά νωρίτερα — επομένως δεν εκτυπώνονται όλα τα βάθη και οι άξονες σε αυτό το κλαδί. Η print_tree() αποτυπώνει τη δυναμική, ασύμμετρη δομή που προκύπτει από τα πραγματικά δεδομένα και όχι ένα ιδεατό πλήρες δέντρο, γι' αυτό και ενδέχεται η κυκλική εναλλαγή να μην φαίνεται εξωτερικά παρότι εφαρμόζεται εσωτερικά σε όλους τους κόμβους.

Στην συνέχεια αναλύουμε την μέθοδο range_query() η οποία εφαρμόζει πολυδιάστατη αναζήτηση περιοχής. Δέχεται ως είσοδο ένα σύνολο από διαστήματα (ranges) ένα για κάθε διάσταση (Price, Engine Capacity, KM Driven), και επιστρέφει όλα τα σημεία που βρίσκονται εντός όλων αυτών των ορίων. Η υλοποίηση αυτή συμβαίνει με την συνθήκη in_range = all(ranges[i][0] <= point[i] <= ranges[i][1] for i in range(len(ranges))) η οποία ελέγχει κάθε διάσταση ανεξάρτητα, και μόνο αν το σημείο ικανοποιεί όλα τα κριτήρια, προστίθεται στα αποτελέσματα. Το πιο κρίσιμο στοιχείο της υλοποίησής είναι το "pruning". Αυτό σημαίνει "κλάδεμα" δηλαδή παραλείπεις σκόπιμα κάποια υποδέντρα κατά την αναζήτηση επειδή γνωρίζεις ότι δεν περιέχουν λύσεις. Η μέθοδος αυτή ελέγχει αν πρέπει να επισκεφτεί το αριστερό ή/και το δεξί υποδέντρο ανάλογα με τον άξονα στον οποίο βρίσκεται ο κόμβος και τη σχέση της τιμής του σημείου με το αντίστοιχο εύρος : if point[axis] >= ranges[axis][0]:

```
    self.range_query(node.left.node, ranges, results)
if point[axis] <= ranges[axis][1]:
    self.range_query(node.right.node, ranges, results)
```

Αν η τιμή του κόμβου είναι έξω από τα όρια του άξονα `axis`, τότε παραλείπεται το αντίστοιχο υποδέντρο.

Η συνάρτηση knn_search() υλοποιεί την αναζήτηση των k πλησιέστερων γειτόνων σε ένα δοσμένο σημείο αναφοράς (query point). Αρχικά, υπολογίζεται η ευκλείδεια απόσταση ανάμεσα στο τρέχον σημείο και το query: dist = euclidean_distance(point, query_point). Η πληροφορία αυτή χρησιμοποιείται για να δημιουργηθεί ένα **max-heap** (`best`), στο οποίο διατηρούνται τα k καλύτερα (πλησιέστερα) σημεία που έχουν βρεθεί ως τώρα. Αν ο σωρός έχει λιγότερα από k στοιχεία, το νέο σημείο εισάγεται αυτόματα. Αν είναι πλήρης, τότε αντικαθίσταται το πιο μακρινό σημείο μόνο αν το καινούριο είναι πιο κοντινό. Μετά γίνεται ο έλεγχος της αναδρομής στο άλλο υποδέντρο δηλαδή το δέντρο που δεν επιλέχθηκε πρώτα λόγω της θέσης του query point σε σχέση με τον διαχωριστικό άξονα: if len(best) < k or abs(diff) < -best[0][0]:recursive_search(second.node). Εδώ συγκρίνεται η κάθετη απόσταση από τον διαχωριστικό άξονα με τη χειρότερη απόσταση στο heap. Αν υπάρχει πιθανότητα να βρεθεί πιο κοντινό σημείο στο άλλο υποδέντρο, τότε γίνεται αναδρομή και εκεί.

Αυτός ο έλεγχος υλοποιεί τη **θεωρία της χωρικής τομής (hypersphere-hyperplane intersection)** και είναι η βασική τεχνική pruning για k-NN σε KD-Tree. Χάρη σε αυτό, η αναζήτηση παραμένει **αποδοτική και στοχευμένη**, αντί να είναι γραμμική.

Παρακάτω παραθέτουμε και ένα παράδειγμα εκτέλεσης του κώδικα:

```
                [RIGHT] Depth 8, Axis 2, Point [737000, 1462, 63804]
                  [LEFT] Depth 9, Axis 0, Point [747000, 1462, 57301]
                    [LEFT] Depth 10, Axis 1, Point [743000, 1462, 61765]
                  [RIGHT] Depth 9, Axis 0, Point [740000, 1462, 86706]
              [RIGHT] Depth 7, Axis 1, Point [961000, 1462, 57002]
                [LEFT] Depth 8, Axis 2, Point [819000, 1462, 62418]
                  [LEFT] Depth 9, Axis 0, Point [842000, 1462, 57060]
                    [LEFT] Depth 10, Axis 1, Point [779000, 1462, 60741]
                  [RIGHT] Depth 9, Axis 0, Point [840000, 1462, 70110]
                    [LEFT] Depth 10, Axis 1, Point [757000, 1373, 88132]
                [RIGHT] Depth 8, Axis 2, Point [1058000, 1462, 66111]
                  [LEFT] Depth 9, Axis 0, Point [1069000, 1462, 59304]
                    [LEFT] Depth 10, Axis 1, Point [999000, 1462, 54421]
                  [RIGHT] Depth 9, Axis 0, Point [989000, 1462, 68282]
Build time for 3D KD-Tree: 0.0142 seconds
Enter your 3D range query:
Price range (min max): 916000 10000000
Engine capacity range (min max): 1000 2000
KM driven range (min max): 45000 60000
Query time for KD-Tree: 0.1052 seconds

 Αποτελέσματα Range Query:
Model: 2019 Maruti XL6 ZETA AT | Price: 921000 | Engine: 1462 | KM: 54342
Model: 2022 Maruti XL6 ZETA MT | Price: 973000 | Engine: 1462 | KM: 49458
Model: 2022 Maruti Ciaz ZETA 1.5 SHVS MT PETROL | Price: 916000 | Engine: 1462 | KM: 46533
Model: 2019 Maruti XL6 ZETA AT | Price: 956000 | Engine: 1462 | KM: 48847
Model: 2020 Maruti Vitara Brezza ZXI PLUS AT SHVS | Price: 981000 | Engine: 1462 | KM: 50421
Model: 2021 Maruti XL6 ALPHA MT | Price: 1016000 | Engine: 1462 | KM: 48740
Model: 2021 Maruti XL6 ZETA MT | Price: 980000 | Engine: 1462 | KM: 53293
Model: 2021 Maruti Ciaz ALPHA AT 1.5 SHVS PETROL | Price: 961000 | Engine: 1462 | KM: 57002
Model: 2022 Maruti XL6 ZETA MT | Price: 1069000 | Engine: 1462 | KM: 59304
Model: 2021 Maruti XL6 ZETA MT | Price: 999000 | Engine: 1462 | KM: 54421


Enter your 3D point for k-NN search:

Enter price (x-axis): 763000

Enter engine capacity (y-axis): 1462

Enter KM driven (z-axis): 73050

🔍 Τα 3 πλησιέστερα αυτοκίνητα:
Model: 2016 Maruti Vitara Brezza ZDI | Price: 751000 | Engine: 1248 | KM: 76173
Model: 2019 Maruti Ciaz ALPHA 1.5 SHVS PETROL | Price: 756000 | Engine: 1462 | KM: 59733
Model: 2016 Maruti Vitara Brezza ZDI PLUS DUAL TONE | Price: 747000 | Engine: 1248 | KM: 72213
```

A1) R Trees

Β' Φάση: LSH

Τη μέθοδο LSH αποφασίσαμε να την εφαρμόσουμε στο πεδίο "Model name", αφού είναι το μόνο που οι τιμές του είναι χαρακτήρες και σε τέτοια δεδομένα έχει κυρίως εφαρμογή αυτός ο αλγόριθμος. Αποτελείται από τρία βήματα.

Το πρώτο ονομάζεται "Shingling" και σε αυτό κάθε όνομα μοντέλου μετατρέπεται σε ένα σύνολο από διγράμματα (shingles, k=2). Έπειτα φτιάχνεται το λεξιλόγιο μας που αποτελείται από το σύνολο των shingles όλων των ονομάτων των μοντέλων.

Το επόμενο βήμα ονομάζεται "MinHashing" και ξεκινάει με την δημιουργία τυχαίων hash functions. Συγκεκριμένα, η συνάρτηση κατακερματισμού που έχουμε επιλέξει είναι της μορφής h(x) = (a*x + b) mod prime. Τα a,b επιλέγονται τυχαία από ένα τεράστιο εύρος τιμών και ο prime είναι ένας πολύ μεγάλος αριθμός ($>2^{32}$). Σκόπος αυτής της συνάρτησης είναι να προσομοιώσει τις τυχαίες μεταθέσεις που απαιτούνται από τον LSH, αλλά είναι ιδιαίτερα ακριβές σε χώρο και χρόνο. Μία δυάδα (a,b) συνιστούν μία συνάρτηση κατακερματισμού και για κάθε όνομα μοντέλου που έχει μετατραπεί σε shingle_set εφαρμόζουμε 20 συναρτήσεις κατακερματισμού. Έτσι,

προκύπτει η MinHash υπογραφή του μοντέλου που είναι μία λίστα αποτελούμενη από 20 “τυχαίες” τιμές.

Το τελευταίο βήμα ονομάζεται “Locality Sensitive Hashing”. Εδώ κάθε signature χωρίζεται σε 10 bands με 2 στοιχεία το καθένα. Δημιουργείται μία λίστα (lsh_buckets), κάθε στοιχείο της οποίας είναι ένα python dictionary που έχει ως key τη hashed τιμή του band slice (bucket_key) και ως value το όνομα του μουντέλου. Η συγκεκριμένη λίστα αποτελείται από 10 στοιχεία, όσα και τα bands και κάθε στοιχείο περιέχει όλα τα dictionaries που αντιστοιχούν στο εκάστοτε band. Αν σε κάποιο band βρεθούν δύο μοντέλα με ίδιο bucket key θεωρούνται παρόμοια.

Για να τρέξουμε τον κώδικα ο χρήστης θα πρέπει να εισαγάγει το όνομα μοντέλου βάσει του οποίου θα τρέξουμε τον LSH για αυτά στο dataset μας. Τότε, θα ακολουθηθεί η ίδια διαδικασία για το query μοντέλο που αναφέρθηκε προηγουμένως για τα μοντέλα του dataset και θα καταλήξουμε σε 10 bucket keys για κάθε band. Για κάθε bucket key που θα προκύπτει θα ελέγχουμε στο αντίστοιχο band του lsh_buckets. Αν βρούμε bucket key που να ταυτίζεται θεωρούμε όλα τα μοντέλα που το μοιράζονται ως candidates και τα επιστρέφουμε.

Τέλος, τα candidate shingles και query shingles επαναξιολογούνται με βάση τη Jaccard Similarity τους, ώστε να δοθεί ένα σκορ ομοιότητας και να παρουσιαστούν ως ταξινομημένα αποτελέσματα.

Παρακάτω παραθέτουμε και ένα screenshot από εκτέλεση του κώδικα:

```
Enter the model name to query (e.g., 'Hyundai i10'): 211 Maru Xla

Top-5 models similar to '211 Maru Xla':
2021 Maruti XL6 ZETA MT        → Similarity: 0.286
2021 Maruti XL6 ALPHA AT       → Similarity: 0.273
2021 Maruti XL6 ALPHA MT       → Similarity: 0.273
2011 Maruti Swift VXI          → Similarity: 0.238
2011 Maruti Swift LXI          → Similarity: 0.238
```

Γ’ Φάση: Evaluation

Σε αυτό το σημείο καλούμαστε να συγκρίνουμε τις τέσσερις τρισδιάστατες δομές που δημιουργήσαμε και να καταλήξουμε σε ένα συμπέρασμα για την αποδοτικότητα τους. Αυτό αποφασίσαμε να το κάνουμε μετρώντας για τη κάθε δομή το build time, το query time, το χρόνο που χρειάστηκε στον αλγόριθμο LSH να επιστρεψει αποτελέσματα και στον αριθμό των αποτελεσμάτων που επέστρεψε ο LSH.

Για το σκοπό αυτό δημιουργήσαμε το αρχείο κώδικα “evaluation.py”. Εκεί κάνουμε import συναρτήσεις από την κάθε δομή που εκτελούν το χτίσιμο του εκάστοτε δέντρου, αλλά και το ψάξιμο για την ικανοποίηση ενός range query. Το σκεπτικό είναι να τρέξουμε την κάθε δομή υπολογίζοντας τις προαναφερόμενες μετρικές για πληθώρα ερωτημάτων και τέλος να παρουσιάσουμε συγκεντρωτικά τα αποτελέσματα.

Από τη θεωρία γνωρίζουμε τις πολυπλοκότητες των δομών που χτίσαμε και ειναι οι παρακάτω:

	Χρόνος	
	Κατασκευή	Range Search
KD tree	$O(n \log n)$	$O\left(n^{1-\frac{1}{3}} + k\right)$
Octree	$O(n \log n)$	$O(n)$
R tree	$O(n \log n)$	$O(n)$
Range Tree	$O(n \log^2(n))$	$O(\log^3(n) + k)$

Και περιμέναμε και αντίστοιχα αποτελέσματα κατά την εκτέλεση του κώδικα μας.

Για την δημιουργία των queries χρησιμοποιήσαμε τη συνάρτηση "**generate_random_query()**", η οποία σύμφωνα με τις ελάχιστες και μέγιστες τιμές των 3 χαρακτηριστικών του dataset βάσει των οποίων χτίζουμε τις δομές, επιστρέφει ένα range query που αποτελείται από τα τρία ζητούμενα ranges.

Για κάθε δομή έχουμε απο μία συνάρτηση ("**evaluate_range_tree**" για το range tree π.χ.), η οποία θα χτίσει το δέντρο και θα υπολογίσει το build_time, έπειτα θα παράξει μέσω της "generate_random_query()" 50 τυχαία queries, θα εκτελέσει το range search για το καθένα, θα υπολογίσει το query_time και τέλος θα υπολογίσει και το χρόνο που χρειάζεται να τρέξει ο LSH για τους candidates που επιστρέφει κάθε φορά, μαζί με το πλήθος τους.

Τέλος, εκτυπώνουμε τα αποτελέσματα, ένα παράδειγμα εκ των οποίων απεικονίζεται στο παρακάτω screenshot

```
=== Final Performance Comparison ===
Structure            | Build Time (s) | Avg Query Time (ms) | Avg LSH Time (ms) | Avg # Candidates
-------------------------------------------------------------------------------------------------
Range Tree + LSH     |         0.5129 |                0.11 |             25.99 |            32.44
KD-Tree + LSH        |         0.0067 |                0.27 |             19.52 |            32.36
QuadTree + LSH       |         0.0105 |                0.23 |             30.17 |            34.56
R-Tree + LSH         |         0.2999 |                0.30 |             39.29 |            39.18
```

Παρατηρούμε ότι το Range Tree είναι με εμφανή διαφορά το πιο αργό όσον αφορά το χτίσιμο του δέντρου, κάτι που περιμέναμε λόγω πολυπλοκότητας ($O(nlog^2n)$), όμως αναπληρώνει σε query time που σημειώνει το χαμηλότερο λόγω της εμφωλευμένης δομής τους. Τα KD-Tree και Quad Tree παρουσιάζουν πολύ χαμηλούς χρόνους κατασκευής, καθιστώντας τα ιδανικά για δυναμικά datasets, ενώ και τα query time τους είναι χαμηλά. Το R-Tree, αν και κατασκευάζεται σε $O(nlogn)$, εδώ εμφάνισε μεγαλύτερους χρόνους τόσο στο χτίσιμο, όσο και σε query time, αλλά και στις μετρικές σχετικές με το LSH. Αυτό αποδίδεται στα πιο ευέλικτα και επικαλυπτόμενα πλαίσια οριοθέτησής του, τα οποία είχαν ως αποτέλεσμα μεγαλύτερο μέσο αριθμό candidates που περνούσαν στη φάση LSH.

Συγκεκριμένα, ο χρόνος εκτέλεσης του LSH βρέθηκε να συσχετίζεται άμεσα με τον αριθμό των candidates που επιστρέφονται από κάθε δομή, επιβεβαιώνοντας ότι το αρχικό φιλτράρισμα των δεδομένων με βάση το εύρος τιμών και η χωρική διάσπαση που προκαλέι η κάθε δομή επηρεάζει σημαντικά την απόδοση της τελικής αναζήτησης ομοιότητας.

Συμπερασματικά, τα αποτελέσματα επιβεβαιώνουν τα γνωστά από τη θεωρία μας δεδομένα και αναδεικνύουν την αποτελεσματικότητα της κάθε δομής σε συνδυασμό με τη μέθοδο LSH.