ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

ΠΟΛΥΔΙΑΣΤΑΤΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΑΚΑΔΗΜΑΙΚΟ ΕΤΟΣ 2023-2024

ΔΙΔΑΣΚΩΝ: ΣΠΥΡΙΔΩΝ ΣΙΟΥΤΑΣ

Μέλη ομάδας:

ΚΟΛΑΓΚΗ ΕΥΑΓΓΕΛΙΑ ΑΜ: 1084599

ΚΑΡΑΓΙΑΝΝΗΣ ΓΕΩΡΓΙΟΣ ΑΜ: 1084586

ΜΑΝΤΕΣ ΜΗΛΤΙΑΔΗΣ ΑΜ: 1084661

ΧΡΥΣΑΥΓΗ ΠΑΤΕΛΗ ΑΜ: 1084513

Περιεχόμενα

0 Εισαγωγή

0.1 Web Crawler

Στα πλαίσια της παρούσας εργασίας μας ζητήθηκε να υλοποιήσουμε index δομές για αναζήτηση σε δεδομένα που θα εξάγουμε από τον ακόλουθο σύνδεσμο: List of computer scientists - Wikipedia.

Πρώτη μας ανάγκη, λοιπόν, είναι να δημιουργήσουμε ένα Dataset πάνω στο οποίο θα υλοποιήσουμε τις αναζητήσεις μεταξύ διαστημάτων και την αναζήτηση ομοιότητας.

Ορισμός Web Crawler

Ο crawler είναι λογισμικό για την αυτόματη λήψη σελίδων από τον ιστό. Όταν θέλουμε να κάνουμε ανάκτηση ιστού, οι σελίδες που έκανε λήψη ο crawler χρησιμοποιούνται για ευρετηρίαση και αναζήτηση. Ένας crawler δέχεται ως είσοδο ένα σύνολο από σελίδες εκκίνησης (seeds) που έχουν ληφθεί, αναλυθεί και ελεγχτεί για νέους συνδέσμους. Οι σύνδεσμοι που δείχνουν σε σελίδες που δεν έχουν ακόμη ληφθεί ,προστίθενται σε μια κεντρική ουρά διευθύνσεων URL για μεταγενέστερη ανάκτηση. Στη συνέχεια ο crawler επιλέγει για να κάνει λήψη μια νέα σελίδα από την ουρά και η διαδικασία επαναλαμβάνεται μέχρι να ικανοποιηθεί ένα κριτήριο διακοπής.

Δημιουργία Web Crawler

Αρχικά, για να δημιουργήσουμε έναν Web Crawler χρησιμοποιούμε τις εξής βιβλιοθήκες:

- **import requests**

Η βιβλιοθήκη requests χρησιμοποιείται για την αποστολή HTTP αιτημάτων σε ιστοσελίδες και τη λήψη των απαντήσεών τους. Χρησιμοποιείται για να ληφθεί ο πηγαίος κώδικας HTML της σελίδας.

- **from bs4 import BeautifulSoup**

Επιτρέπει να εξάγουμε δεδομένα από HTML, ταξινομώντας τα, εντοπίζοντας ετικέτες, και άλλες εργασίες σχετικές με το scraping.

- **import string**

Επεξεργασία αλφαριθμητικών.

- **import pandas as pd**

Διαχείριση δεδομένων που έχουν ληφθεί από τις ιστοσελίδες.

- **import re**

Η βιβλιοθήκη re παρέχει τη δυνατότητα χρήσης regular expressions. Χρησιμοποιείται για την εξαγωγή συγκεκριμένων τμημάτων κειμένου από την HTML.

- **import urllib.parse**

Η βιβλιοθήκη urllib.parse χρησιμοποιείται για την ανάλυση και τη διαχείριση URLs.

- **from selenium import webdriver**

Φορτώνει την κλάση webdriver, η οποία είναι υπεύθυνη για την επικοινωνία με τον προγραμματισμένο φυλλομετρητή.

- **from selenium.webdriver.common.by import By**

Φορτώνει τη σταθερά By, η οποία παρέχει διάφορες μεθόδους για τον εντοπισμό στοιχείων στην ιστοσελίδα χρησιμοποιώντας στρατηγικές όπως το ID, την κλάση, το όνομα κ.λπ.

- **from selenium.webdriver.support.ui import WebDriverWait**

Φορτώνει την κλάση WebDriverWait, η οποία επιτρέπει την αναμονή για συγκεκριμένες συνθήκες (π.χ. εμφάνιση στοιχείου στη σελίδα) πριν συνεχιστεί η εκτέλεση του κώδικα.

- **from selenium.webdriver.support import expected_conditions as EC**

Χρησιμοποιείται συνήθως με το WebDriverWait για να περιμένει μια συγκεκριμένη συνθήκη πριν συνεχιστεί το πρόγραμμα.

Γενικά η **selenium** συνήθως χρησιμοποιείται για την αλληλεπίδραση με δυναμικά δημιουργημένες ιστοσελίδες ή για την εκτέλεση εργασιών που απαιτούν πλοήγηση μέσα σε σελίδες που απαιτούν σύνδεση (π.χ. εκπλήρωση φόρμας, κλικ σε κουμπιά κ.λπ.).

Όσον αφορά το πηγαίο κώδικα, δημιουργούμε μια κλάση με όνομα ScientistInfoExtractor και λίστες για την αποθήκευση των ονομάτων, βραβείων, Dblps και δεδομένων εκπαίδευσης των επιστημόνων.

```
class ScientistInfoExtractor:
  def __init__(self):
    self.surnames = []
    self.awards_counts = []
    self.awards_lists = []
    self.education_info_list = []
    self.dblp_list = []
```

Μέσα στην κλάση αυτή δημιουργούμε τις συναρτήσεις που παρουσιάζονται παρακάτω:

Για να εξάγουμε τις πληροφορίες εκπαίδευσης για κάθε επιστήμονα δημιουργούμε την συνάρτηση **extract_education_info().** Η συνάρτηση έχει σκοπό να εξάγει τις πληροφορίες για την εκπαίδευση από τον HTML κώδικα της ιστοσελίδας https://en.wikipedia.org/wiki/scientist_name. Πιο συγκεκριμένα, αναζητούμε όλα τα **span στοιχεία** που περιέχουν την **κλάση mw-headline**, το τμήμα ουσιαστικά που περιέχει τα **headings**.Στην συνέχεια αναζητάμε όλα τα Headings που περιέχουν την λέξη education και εξάγουμε τις πληροφορίες που περιέχονται μετά από το heading και ανάλογα με το tag που υπάρχει μετά (p, ul, h2) εξάγουμε τις πληροφορίες κάνοντας τις κατάλληλες μορφοποιήσεις.

```
def extract_education_info(self, soup):
  headings = soup.find_all('span', {'class': 'mw-headline'})
  for heading in headings:
    if re.search(r'\beducation\b', heading.text, flags=re.IGNORECASE):
      education_paragraphs = heading.find_all_next(['p', 'ul', 'h2'])
      education_info = ''
      for elem in education_paragraphs:
        if elem.name == 'p':
          education_info += elem.text.strip() + ' '
        elif elem.name == 'ul':
          education_info += ', '.join([li.text.strip() for li in elem.find_all('li')]) + ' '
```

```
            elif elem.name == 'h2':
                break
        return education_info.strip()
    return "Education information not found"
```

Η συνάρτηση **get_max_record_info()** αντλεί πληροφορίες για τις δημοσιεύσεις κάθε επιστήμονα από την ιστοσελίδα https://dblp.org/. Για αρχή γίνεται αναζήτητση με βάση το όνομα του επιστήμονα που έχει αντληθεί από την ιστοσελίδα https://en.wikipedia.org/wiki/List_of_computer_scientists. Αν η αναζήτηση είναι επιτυχής και βρεθεί η ακριβής αντιστοίχιση με το όνομα ενεργοποιείται ένα Selenium WebDriver για να αλληλοεπιδράσει με την σελίδα Dblp. Η σελίδα ανοίγει για χρονικό διάστημα όχι μεγαλύτερο των 10 δευτερολέπτων και πραγματοποιείται αναζήτηση του για τον μέγιστο αριθμό δημοσιεύσεων. Τα αποτελέσματα αποθηκεύονται στην λίστα dblp_list και επιστρέφονται. Σε περίπτωση που προκύψουν σφάλματα κατά την διάρκεια της παραπάνω διαδικασίας εκτυπώνονται μηνύματα σφάλματος και αν δεν βρεθεί ακριβής αντιστοίχιση με το όνομα τότε παίρνει την τιμή 0.

```
def get_max_record_info(self,scientist_name):
    def extract_max_record_info(soup):
        max_record_info = soup.find('span', {'id': 'max-record-info'})
        if max_record_info:
            return max_record_info.text
        return "Max record information not found"

    try:
        # Δημιουργία URL για αναζήτηση στην DBLP με βάση το όνομα του επιστήμονα
        publication_url = f"https://dblp.org/search?q={'+'.join(scientist_name.split())}"

        # Αίτηση στη σελίδα αναζήτησης DBLP
        publication_response = requests.get(publication_url)

        if publication_response.status_code == 200:
            publication_soup = BeautifulSoup(publication_response.text, 'html.parser')

            # Εύρεση του σχετικού συνδέσμου (link)
            exact_match_link = publication_soup.find('a',
                                    string=re.compile(re.escape(scientist_name), re.IGNORECASE))

            if exact_match_link:
                # Selenium WebDriver, για Chrome browser, για την DBLP
                driver = webdriver.Chrome()

                try:
                    driver.get(exact_match_link['href'])

                    # Αναμονή για το φορτωμένο περιεχόμενο της σελίδας
                    WebDriverWait(driver, 10).until(
                        EC.presence_of_element_located((By.CSS_SELECTOR, "body"))
                    )

                    # Ανάκτηση πληροφοριών max record από τη σελίδα χρησιμοποιώντας Selenium
                    max_record_info = extract_max_record_info(BeautifulSoup(driver.page_source,
'html.parser'))
                    print(f"Max Record Info for {scientist_name}: {max_record_info}")
                    # Προσθήκη στη λίστα dblp
                    self.dblp_list.append(max_record_info)
```

```
            return max_record_info

        except Exception as e:
            print(f"Error processing {scientist_name}: {str(e)}")
            return f"Error processing {scientist_name}"

        finally:
            # Κλείσιμο του WebDriver για την DBLP
            driver.quit()

    else:
        print(f"Exact match link not found for {scientist_name}")
        self.dblp_list.append(0)
        return 0
  else:
    print(f"Failed to retrieve publication page for {scientist_name}")
    return "Failed to retrieve publication page"

except Exception as e:
  print(f"An unexpected error occurred: {str(e)}")
  return f"An unexpected error occurred: {str(e)}"
```

Έπειτα, έχουμε την συνάρτηση **extract_scientist_info()** η οποία χωρίζει τα ονοματεπώνυμα των επιστημόνων και κρατά μόνο τα επίθετα, τα οποία εισάγονται στην λίστα **self.surname**. Στην συνέχεια, συνθέτουμε το url για κάθε επιστήμονα αντικαθιστώντας στο ονοματεπώνυμα τα κενά με κάτω παύλες και κάνει αίτημα στην σελίδα της Wikipedia για τον επιστήμονα. Αν το αίτημα είναι σωστό διαβάζει το html περιεχόμενο του συγκεκριμένου επιστήμονα στην Wikipedia χρησιμοποιώντας BeautifulSoup που προαναφέραμε. Καλώντας την συνάρτηση **extract_education_info()** εξάγουμε τις πληροφοριές για την εκπαίδευση των επιστημόνων. Επιπλέον, αναζητάμε τα headings που περιέχουν διάφορες μορφές της λέξης Award και αν βρεθούν γίνεται επεξεργασία για να εξαχθεί το πλήθος των βραβείων ανάλογα με το **tag** που περιέχουν **(P, ul).** Αν δεν υπάρχει Heading εισάγει την τιμή 0 για τον συγκεκριμένο επιστήμονα.

```
def extract_scientist_info(self, scientist_name):
  name_parts = scientist_name.split()
  surname = name_parts[-1]
  self.surnames.append(surname)

  # Δημιουργία URL για αναζήτηση του επιστήμονα στη Wikipedia
  scientist_url = f"https://en.wikipedia.org/wiki/{urllib.parse.quote(scientist_name.replace(' ', '_'))}"

  # Αίτηση στη σελίδα του επιστήμονα στη Wikipedia
  scientist_response = requests.get(scientist_url)
  if scientist_response.status_code == 200:
    scientist_soup = BeautifulSoup(scientist_response.text, 'html.parser')
    print(f"Processing: {scientist_name}")

    # Εξαγωγή πληροφοριών εκπαίδευσης
    education_info = self.extract_education_info(scientist_soup)
    self.education_info_list.append(education_info)

    # Αναζήτηση των επικεφαλίδων που περιέχουν τις λέξεις 'Award' ή 'award'
    awards_heading = scientist_soup.find(
      lambda tag: tag.name == 'span' and re.search(r'(Award|award)', tag.text, re.IGNORECASE))
    if awards_heading:
```

```
            print("Awards heading found")

            # Εύρεση του επόμενου στοιχείου, είτε ως <ul>, είτε ως <p>
            next_sibling = awards_heading.find_next(['ul', 'p'])

            # Δημιουργία λίστας για αποθήκευση των βραβείων
            awards_list = []

            # Υπολογισμός του αριθμού των βραβείων
            if next_sibling and next_sibling.name == 'ul':
                awards_count = len(next_sibling.find_all('li'))
                self.awards_counts.append(awards_count)
                self.awards_lists.append([])  # Προσθήκη κενής λίστας για συνέπεια
                print(f"Awards count from <ul>: {awards_count}")
            elif next_sibling and next_sibling.name == 'p':
                # Εύρεση όλων των <p> που ακολουθούν τις επικεφαλίδες βραβείων
                p_elements = awards_heading.find_all_next('p')

                # Επεξεργασία των <p>
                for p_element in p_elements:
                    # Εξαγωγή μοναδικών βραβείων από το κείμενο
                    awards_text = p_element.get_text()
                    unique_awards = set(re.findall(r'(Award|awarded)', awards_text))
                    awards_list.extend(unique_awards)

                print(f"Unique Awards List: {awards_list}")

                # Προσθήκη της λίστας στην κύρια λίστα βραβείων
                self.awards_lists.append(awards_list)
                self.awards_counts.append(len(awards_list))  # Προσθήκη του πλήθους
            else:
                print("Invalid next sibling. Skipping.")
                self.awards_counts.append(0)
                self.awards_lists.append(0)  # Προσθήκη κενής λίστας για συνέπεια

        else:
            print("Awards heading not found")
            self.awards_counts.append(0)
            self.awards_lists.append(0)  # Προσθήκη κενής λίστας για συνέπεια

    else:
        print(f"Failed to retrieve page for {scientist_name}")
        self.awards_counts.append(0)
        self.education_info_list.append("Education information not found")
```

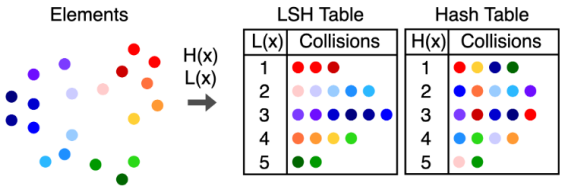
Στο url (https://en.wikipedia.org/wiki/List_of_computer_scientists), από το οποίο θα εξάγουμε όλα τα δεδομένα, αρχικά, γίνεται **αίτημα** χρησιμοποιώντας την μέθοδο **request.get.** Αν το αίτημα είναι επιτυχημένο εξάγουμε τον html κώδικα της παραπάνω ιστοσελίδας. Διατρέχουμε όλα τα γράμματα της αγγλικής αλφαβήτου και για κάθε γράμμα εντοπίζουμε το span στοιχείο που περιέχει την κλάση **mw-headline** και **id** το συγκεκριμένο γράμμα. Επίσης, εντοπίζουμε το επόμενο **tag ul**, ώστε να εντοπίσουμε την λίστα με όλους τους επιστήμονες που ξεκινάν με το τρέχον γράμμα και εξάγουμε το ονοματεπώνυμο από το **tag a** που βρίσκεται μέσα στο **tag li**. Επιπλέον, καλούμε τις συναρτήσεις **extract_scientist_info()**, **get_max_record_info()** με ορίσματα το ονοματεπώνυμο.

```
# Ορισμός του URL που θα κάνουμε scraping
url = "https://en.wikipedia.org/wiki/List_of_computer_scientists"

# Αίτηση GET για τη λήψη της ιστοσελίδας
response = requests.get(url)
max_record_infos = []

# Δημιουργία ενός αντικειμένου ScientistInfoExtractor
extractor = ScientistInfoExtractor()

# Έλεγχος αν η αίτηση ήταν επιτυχής
if response.status_code == 200:
    # Χρήση BeautifulSoup για την ανάλυση της HTML σελίδας
    soup = BeautifulSoup(response.text, 'html.parser')

    # Προσπέλαση των επικεφαλίδων για κάθε γράμμα του αλφαβήτου
    for letter in string.ascii_uppercase:
        span_element = soup.find('span', {'class': 'mw-headline', 'id': letter})
        if span_element:
            # Εύρεση της λίστας των επιστημόνων για το συγκεκριμένο γράμμα
            scientist_list = span_element.find_next('ul')
            # Προσπέλαση κάθε επιστήμονα στη λίστα
            for scientist in scientist_list.find_all('li'):
                scientist_name_element = scientist.find('a')
                if scientist_name_element:
                    scientist_name = scientist_name_element.text
                    # Εξαγωγή πληροφοριών για τον επιστήμονα με τον extractor
                    extractor.extract_scientist_info(scientist_name)
                    extractor.get_max_record_info(scientist_name)
```

Τέλος, δημιουργούμε ένα dataframe με στήλες Surname, Award, Education, Dblp και αποθηκεύουμε σε αυτό τα αποτελέσματα, τα οποία τα εξάγουμε και σε csv (scientists.csv).

```
# Αποθήκευση των δεδομένων που συλλέξαμε σε ένα DataFrame του pandas
data = {
    'Surname': extractor.surnames,
    'Awards': extractor.awards_counts,
    'Education': extractor.education_info_list,
    'Dblp': extractor.dblp_list
}
df = pd.DataFrame(data)

# Αποθήκευση των δεδομένων σε ένα αρχείο CSV
df.to_csv('scientists.csv', index=False, encoding='utf-8')
```

0.2 Αναζήτηση με LSH



Το LSH (**locality-sensitive hashing**) στην επιστήμη των υπολογιστών είναι μια νέα και δυναμική τεχνική, όπου οι περισσότεροι το χρησιμοποιούν για αναζήτηση κοντινού γείτονα, αλλά είναι επίσης χρήσιμο για σκιαγράφηση αλγορίθμων και ανάλυση δεδομένων υψηλών διαστάσεων. Η τεχνική αυτή προσπαθεί να αντιστοιχίσει παρόμοια αντικείμενα με μεγάλη πιθανότητα στο ίδιο Hash Bucket και ανόμοια σε διαφορετικά Buckets. Σε αντίθεσή με άλλες τεχνικές hashing οι οποίες τοποθετούν πολλές φορές σε ίδια Hash Buckets στοιχεία τα οποία είναι ανόμοια μεταξύ τους, αυξάνοντας έτσι την πολυπλοκότητα και τον χρόνο εκτέλεσης. Το LSH διαφέρει από τις άλλες συμβατές τεχνικές κατακερματισμού στο ότι οι συγκρούσεις κατακερματισμού μεγιστοποιούνται δίνοντας μας έτσι καλύτερα αποτελέσματα.

Για την υλοποίηση του LSH αρχικά δημιουργούμε την συνάρτηση **shingle_education()**, η οποία δέχεται σαν όρισμα ένα κείμενο (text) και ένα προκαθορισμένο μέγεθος shingle (k), το οποίο ορίζεται ως 2 αν δεν δοθεί άλλη τιμή. Έπειτα, δημιουργείται μια κενή λίστα **shingle_set** για την αποθήκευση των shingles. Ορίζεται ένας βρόχος for που επαναλαμβάνεται για κάθε θέση i στο κείμενο, έτσι ώστε να δημιουργηθούν όλα τα shingles και σε κάθε επανάληψη, λαμβάνονται k χαρακτήρες από το κείμενο, από τη θέση i μέχρι i + k - 1, και προστίθενται στη λίστα shingle_set. Τέλος, η λίστα με τα shingles μετατρέπεται σε σύνολο για να αφαιρεθούν τα διπλότυπα, και επιστρέφεται το σύνολο αυτό.

```
def shingle_education(text, k=3):
    shingle_set = []
    for i in range(len(text) - k + 1):
        shingle_set.append(text[i:i + k])
    return set(shingle_set)
```

Στην συνέχεια υλοποιούμε την συνάρτηση **jaccard(s1, s2)** η οποία υπολογίζει τον βαθμό ομοιότητας μεταξύ των συνόλων s1 και s2. Αρχικά υπολογίζουμε την τομή των δύο συνόλων, δηλαδή το πλήθος των κοινών στοιχείων |s1∩s2| με την συνάρτηση **len(s1.intersection(s2))**. Έπειτα υπολογίζουμε το μέγεθος της ένωσης των δύο στοιχείων |s1∪s2|, δηλαδή το πλήθος των μοναδικών στοιχείων που υπάρχουν στα σύνολα με την εντολή **len(s1.union(s2))**. Τέλος για τον υπολογισμό της Jaccard similarity πρέπει να υπολογίσουμε τον λόγο του μεγέθους της τομής προς το μέγεθος της ένωσης, δηλαδή υπολογίζουμε το |s1∩s2|/|s1∪s2| και αυτό επιτυγχάνεται με την εντολή **len(s1.intersection(s2)) / len(s1.union(s2)).**

```
def jaccard(s1, s2):
    # υπολογισμός |s1∩s2|
    # υπολογισμός |s1∪s2|
    # υπολογισμός Jaccard similarity
    return len(s1.intersection(s2)) / len(s1.union(s2))
```

Έπειτα υλοποιούμε την συνάρτηση **bucket_creator(sign, bands, rows)** η οποία θα εξάγει τα κοντινά στοιχεία στα ίδια buckets για την τεχνική του LSH. Αρχικά δημιουργούμε μια κενή λίστα για την αποθήκευση των buckets που θα δημιουργηθούν με την εντολή **buckets = [].** Στην συνέχεια για κάθε band υπολογίζουμε την αρχή και το τέλος της συνάφειας έναρξης του αλλά και έναρξης του επόμενου στον πίνακα (signature matrix) με τις εντολές **start = band * rows** και **end = (band + 1) * rows** ταυτόχρονα. Τέλος, με την εντολή

buckets.append(hash(tuple(sign[start:end]))) καταφέρνουμε και δημιουργούμε ένα hash από το σημείο έναρξης μέχρι και το τέλος του τρέχοντος band προσθέτοντας το στην λίστα **buckets** που υλοποιήσαμε αρχικά.

```
def bucket_creator(sign, bands, rows):
  buckets = []
  for band in range(bands):
    # Υπολογισμός αρχής συνάφειας του Band
    start = band * rows
    # Υπολογισμός τέλους συνάφειας του Band
    end = (band + 1) * rows
    # Αποθήκευση των buckets στην λίστα
    buckets.append(hash(tuple(sign[start:end])))
  return buckets
```

Εν συνεχεία, προκειμένου να ολοκληρώσουμε τις απαιτήσεις που χρειάζονται για την υλοποίηση του LSH δημιουργούμε την συνάρτηση **minhash_education(shingles, hashes=95)** την οποία μπορεί την θεωρήσουμε ως την «υπογραφή» της εκτίμησης μεταξύ των συνόλων μας. Αρχικά υπολογίζουμε με την εντολή **max_hash = 2 ** 32 - 1 και modulo = 2 ** 32** το μέγιστο δυνατό hash value για τις μέγιστες δυνατές υπογραφές που θα λάβουμε. Στην συνέχεια δημιουργούμε μια λίστα με τυχαία ζευγάρια αριθμών όπου κάθε ζευγάρι θα αντιπροσωπεύει και ένα σύνολο από hash functions και αυτό επιτυγχάνεται με την εντολή **funcs = [(random.randint(0, max_hash), random.randint(0, max_hash)) for _ in range(hashes)].** Έπειτα αφού δημιουργήσαμε τα τυχαία hash functions, δημιουργούμε μια λίστα με αυτά όπου κάθε hash function παίρνει ένα shingle ως είσοδο και επιστρέφει ένα hash value ως αποτέλεσα το οποίο θα μας βοηθήσει με την δημιουργία των υπογραφών. Αυτό επιτυγχάνεται με την εντολή **hash_functions = [lambda x, a=a, b=b: (a * hash(x) + b) % modulo for a, b in funcs].** Τέλος υπολογίζεται το hash value για κάθε shingle στο σύνολο shingles και αποθηκεύεται το ελάχιστο hash value από αυτά. Αυτό το ελάχιστο hash value αποτελεί την υπογραφή (signature) για το αντίστοιχο shingle set και αυτό υλοποιείται με την εντολή **sign_x = [min(func(shingle) for shingle in shingles) for func in hash_functions].**

```
def minhash_education(shingles, hashes=95):
  # Υπολογισμός μέγιστου δυνατού hash value
  max_hash = 2 ** 32 - 1
  modulo = 2 ** 32
  # γεννήτρια τυχαίων αριθμών για την αναπαραγωγή των αποτελεσμάτων.
  random.seed(42)
  # Δημιουργία ζευγαριών Hash Functions
  funcs = [(random.randint(0, max_hash), random.randint(0, max_hash)) for _ in range(hashes)]
  # Δημιουργία hash Values απο τα hash functions
  hash_functions = [lambda x, a=a, b=b: (a * hash(x) + b) % modulo for a, b in funcs]
  # Δημιουργία signature απο τα Hash Values
  sign_x = [min(func(shingle) for shingle in shingles) for func in hash_functions]
  return sign_x
```

Τέλος, αφού δημιουργήσαμε όλες τις απαραίτητες συναρτήσεις που χρειαζόμαστε για το Local Sensitive Hashing υλοποιούμε την τελική μας συνάρτηση **lsh_education(query, df, threshold)** η οποία και θα μας επιστρέψει τα τελικά αποτελέσματα από την εκτέλεση του query με το ποσοστό ομοιότητας να είναι το threshold. Αρχικά επειδή εκτελούμε το lsh με βάση το education κάθε εγγραφής, φιλτράρουμε τα δεδομένα του data frame (df) το οποίο περιέχει τα στοιχεία στα οποία θα εφαρμόσουμε το LSH. Συγκεκριμένα αφαιρούμε από την συλλογή τις εισαγωγές που έχουν σαν εγγραφή στην στήλη education την φράση Education information not

found διότι δεν μπορούν να αποτελέσουν στοιχείο σύγκρισης ομοιότητας εγγραφών. Στην συνέχεια με την βοήθεια της συνάρτησης **shingle_education** μετατρέπουμε τις εγγραφές σε σύνολα από singles όπως αιτιολογήσαμε παραπάνω. Έπειτα δημιουργούμε τις υπογραφές (signatures) για κάθε σύνολο shingles χρησιμοποιώντας τη συνάρτηση **minhash_education** και έπειτα για τις υπογραφές αυτές φτιάχνουμε τα απαραίτητα buckets χρησιμοποιώντας τη συνάρτηση **bucket_creator** που θα περιέχουν ομάδες υπογραφών που μπορεί να είναι όμοιες σχεδόν μεταξύ τους. Τέλος για κάθε τέτοιο bucket χρησιμοποιούμε την συνάρτηση **jaccard** η οποία και θα μας επιστρέφει τις πιθανές ομοιότητες τους και με βάση αυτές μας επιστρέφει σαν έξοδο τα ζευγάρια εισαγωγών που έχουν ομοιότητα μεγαλύτερη από ή ίση με ένα καθορισμένο κατώφλι ομοιότητας (threshold) το οποίο θέτουμε εμείς.

```
def lsh_education(query, df, threshold):
  # φιλτράρισμα εισαγωγών αν έχουν σαν εγγραφή στο education το "Education information not found"
  valid_entries = [(entry, surname) for entry, surname in zip(query, df['Surname']) if "Education information not found" not in entry]
  # Μετατροπή εγγραφών σε shingles
  shingles_education = [shingle_education(entry) for entry, _ in valid_entries]
  # Δημιουργεία κατάληλων υπογραφών για τα shingles μας
  signatures = [minhash_education(s) for s in shingles_education]
  # Ανάθεση των υπογραφών αυτών σε κατάλληλα Buckets
  bands = 15
  rows = 12
  buckets = [bucket_creator(sign, bands, rows) for sign in signatures]

  pairs = set()
  for i, buckets1 in enumerate(buckets):
    for j, buckets2 in enumerate(buckets):
      if i != j and any(b1 == b2 for b1, b2 in zip(buckets1, buckets2)):
        if i < j:
          pairs.add((i, j))
        else:
          pairs.add((j, i))
  # Υπολογισμός των πιθανοτήτων ομοιότητας των ζευγαριών με χρηση της jaccard
  # και επιστροφή κατάλληλων ζευγαριών με ομοιότητα μεγαλύτερη από ή ίση με το threshold μας
  final_pairs = []
  for i, j in pairs:
    similarity = jaccard(shingles_education[i], shingles_education[j])
    if similarity >= threshold:
      entry1, surname1 = valid_entries[i]
      entry2, surname2 = valid_entries[j]
      final_pairs.append(((entry1, surname1), (entry2, surname2)))

  return final_pairs
```

Σημειώνουμε εδώ πως η προεπεξεργασία Κειμένου (Text Preprocessing) είναι επιπλέον ένα κρίσιμο βήμα στην επεξεργασία φυσικής γλώσσας (NLP) και στόχος της είναι να κάνει τα δεδομένα κειμένου πιο δομημένα, συνεπή και ευκολότερα στην επεξεργασία. Γι' αυτό έχουμε υλοποιήσει την παρακάτω συνάρτηση, η οποία παίρνει τα δεδομένα εκπαίδευσης και αφαιρεί τις stop words (λέξεις που εμφανίζονται πολύ συχνά και δεν έχουν βαρύτητα ως προς την σημασία και την διάκριση του κειμένου π.χ σύνδεσμοι, άρθρα κλπ.) και κάνει συνέγκλιση (stemming) και κανονικοποίηση των λέξεων κρατώντας μόνο την ρίζα-πρόθεμα τους (πχ στην λέξη maths κρατάμε το πρόθεμα math έτσι ώστε λέξεις όπως mathematician/mathematics κτλ. να ταιριάζουν ώστε να βρίσκουμε ομοιότητα σε όρους με ίδιο νόημα). Έτσι επιτυγχάνεται η

ακριβέστερη σύγκριση των δεδομένων εκπαίδευσης και αυξάνεται η ανάκληση και η ακρίβεια των αποτελεσμάτων που θα εισάγουμε στο LSH.

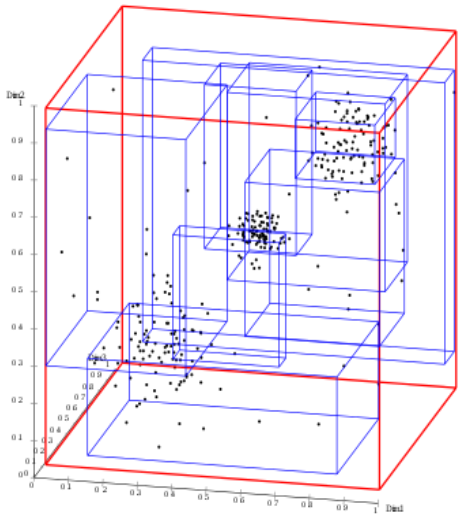
```
def preprocess_education(education_text):
    stop_words = set(stopwords.words('english'))
    stemmer = PorterStemmer()

    # μετατροπή όλων των γραμμάτων σε μικρά
    education_text = education_text.lower()

    # Tokenize το κείμενο σε λέξεις
    words = education_text.split()

    # εφαρμογή stemming and αφαίρεση stopwords
    processed_words = [stemmer.stem(word) for word in words if word not in stop_words]

    # σύνδεση στοιχείων λίστας ως μεμονωμένο string
    processed_text = ' '.join(processed_words)

    return processed_text
```

Οι βιβλιοθήκες που χρησιμοποιήσαμε είναι οι εξής:

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
nltk.download('stopwords')
```

- Εισάγουμε τη βιβλιοθήκη NLTK, η οποία παρέχει εργαλεία για την επεξεργασία φυσικής γλώσσας.
- Η συνάρτηση stopwords παρέχει λίστες με συνήθεις λέξεις που συνήθως δεν έχουν σημασία στην ανάλυση κειμένου.
- Ο PorterStemmer είναι ένας αλγόριθμος στελέχωσης (stemming) που αφαιρεί τα προθέματα και τα επιθήματα από τις λέξεις προκειμένου να επιστρέψει τη βασική μορφή της λέξης.
- nltk.download('stopwords'): Αυτή η εντολή λήψης κατεβάζει τη λίστα των stopwords από το NLTK για να λάβουμε τις λέξεις-κλειδιά.

1 R-tree



Τα R-trees ανήκουν στις πολυδιάστατες δομές δεδομένων και χρησιμοποιούνται για την εύρεση δεδομένων, όπως παραδείγματος χάριν τις γεωγραφικές συντεταγμένες, αλλά και αποθήκευση χωρικών αντικειμένων, όπως τοποθεσίες κτηρίων αλλά και διαφόρων πολυγωνικών αντικειμένων, όπως τα περιγράμματά λιμνών επάνω στους χάρτες. Η γενική ιδέα της υλοποίησης του δέντρου αυτού είναι η κατάταξη των δεδομένων μας επάνω στον χώρο είτε αυτός είναι 2D, 3D κ.τ.λ. και η ομαδοποίηση των κοντινότερων αντικειμένων αναπαριστώντας τα με το ελάχιστο οριοθετημένο ορθογώνιο ή αλλιώς MBR. Τα MBR χρησιμοποιούνται συχνά ως ένδειξη της γενικής θέσης ενός χαρακτηριστικού ή ενός συνόλου δεδομένων, είτε για λόγους προβολής είτε για χωρικό ερώτημα ή για σκοπούς χωρικής ευρετηρίασης. Τα R-trees αποτελούν ένα ισορροπημένο δέντρο αναζήτησης με εξωτερική μνήμη και οργάνωσης των δεδομένων σε σελίδες, πράγμα το οποίο μας βοηθάει όταν έχουμε μεγάλα σύνολα και βάσεις δεδομένων όπου οι κόμβοι μπορούν να σελιδοποιηθούν στη μνήμη.

1.1 Κατασκευή

Επειδή μας ζητήθηκε η δομή που θα υλοποιήσουμε να είναι ως προς τα 3 πεδία (surname, #awards, #DBLP_Record), η δομή του δέντρου μας θα υλοποιηθεί επάνω στον 3D χώρο με τα οριοθετημένα ορθογώνια να παίρνουν την εξής μορφή: **min(x), max(x), min(y), max(y), min(z), max(z)**.

Αρχικά ορίζουμε την **κλάση Rect** η οποία θα αναπαριστά μια ορθογώνια περιοχή στον τρισδιάστατο χώρο. Μέσα στην κλάση αυτή εκτελούνται η συνάρτηση **__init__** και η **get_points**. Η __init__ σαν **είσοδο** δέχεται το id, το οποίο είναι το αναγνωριστικό για κάθε ορθογώνιο, τα x_low, y_low, z_low, x_high, y_high, z_high που αναπαριστούν τις συντεταγμένες των πεδίων που θα πάρουμε σαν είσοδο. Επειδή το surname που θα μπει σαν είσοδος στο x_low και x_high αντίστοιχα είναι αλφαριθμητικό εκτελούμε κανονικοποίηση με την εντολή self.x_low = ord(x_low[0]) - ord('A') + 1 ώστε να μετατρέψουμε τον πρώτο χαρακτήρα του x_low και x_high σε αριθμητική τιμή για αποφυγή τυχόν λαθών. Επίσης, αρχικοποιούμε τις μεταβλητές **full_name** και **education** που θα μας βοηθήσουν στην συνέχεια με την εκτύπωση των αποτελεσμάτων αλλά και με το LSH αντίστοιχα.

Με την μέθοδο get_points επιστρέφουμε μια πλειάδα που περιέχει τις συντεταγμένες της κάτω και της άνω γωνίας του ορθογωνίου.

Ο κώδικας μας είναι ο εξής :

```
class Rect:
  def __init__(self, id, x_low, y_low, z_low, x_high, y_high, z_high, full_name, education):
    # Αρχικοποίηση συντεταγμένων ορθογωνίου
    self.id = id
    self.x_low = ord(x_low[0]) - ord('A') + 1  # Αλφαριθμητική κανονικοποίηση
    self.y_low = y_low
    self.z_low = z_low
```

```
        self.x_high = ord(x_high[0]) - ord('A') + 1  # Αλφαριθμητική κανονικοποίηση
        self.y_high = y_high
        self.z_high = z_high
        # Αρχικοποίηση μεταβλητών για αποθήκευση τιμών για το Range-Search + LSH
        self.full_name = full_name
        self.education = education

    # Επιστροφή πλειάδας συντεταγμένων
    @property
    def get_points(self):
        return self.x_low, self.y_low, self.z_low, self.x_high, self.y_high, self.z_high
```

Στην συνέχεια υλοποιούμε την **κλάση Node** η οποία αντιπροσωπεύει έναν κόμβο επάνω στο δέντρο μας. Κάθε κόμβος έχει ένα μοναδικό αναγνωριστικό (id) και μια λίστα (idx) για την αποθήκευση δεικτών μαζί με τα αντίστοιχα MBR τους. Με την **μέθοδο mbr** που έχουμε ορίσει μέσα στην κλάση μας, επιστρέφουμε τα MBR κάθε κόμβου. Τα MBR των σημείων τα αναπαριστούμε ως μία πλειάδα που έχει τα εξής χαρακτηριστικά (min_xl, min_yl, min_zl, max_xh, max_yh, max_zh). Η μέθοδος επαναλαμβάνεται στη λίστα idx, η οποία περιέχει πλειάδες της φόρμας (index, MBR). Βρίσκει τις ελάχιστες και μέγιστες τιμές για κάθε διάσταση (x, y, z) σε όλους τους θυγατρικούς κόμβους ή εγγραφές και κατασκευάζει το MBR με βάση αυτές τις ακραίες τιμές. Με την μέθοδο **add_idx(self, x)** προσθέτουμε στου αντίστοιχους κόμβους του δέντρου ευρετηρίου μας μαζί με το αντίστοιχο MBR όπως φαίνεται παρακάτω.

```
# Αρχικοποίηση της κλάσης Node για τη δημιουργία κόμβων στο δέντρο
class Node:
    def __init__(self, id: int):
        self.id = id  # Αναγνωριστικό κόμβου
        self.idx = []  # Λίστα αποθήκευσης δεικτών για τα MBR

    # Δημιουργία και επιστροφή MBR κάθε κόμβου
    # και εύρεση μέγιστης και ελάχιστης τιμής για κάθε διάσταση (x, y, z)
    @property
    def mbr(self):
        min_xl = self.idx[0][1][0]
        min_yl = self.idx[0][1][1]
        min_zl = self.idx[0][1][2]
        max_xh = self.idx[0][1][3]
        max_yh = self.idx[0][1][4]
        max_zh = self.idx[0][1][5]

        for i in self.idx:
            min_xl = min(min_xl, i[1][0])
            min_yl = min(min_yl, i[1][1])
            min_zl = min(min_zl, i[1][2])
            max_xh = max(max_xh, i[1][3])
            max_yh = max(max_yh, i[1][4])
            max_zh = max(max_zh, i[1][5])

        return min_xl, min_yl, min_zl, max_xh, max_yh, max_zh


# Προσθήκη ευρετηρίου στους αντίστοιχους κόμβους
    def add_idx(self, x):
        self.idx.append(x)
```

Παρακάτω, δημιουργούμε την **κλάση LeafNode** η οποία αντιπροσωπεύει τα φύλλα του δέντρου μας. Κάθε φύλλο έχει ένα μοναδικό αναγνωριστικό (id) και μια λίστα (entries) για την αποθήκευση καταχωρήσεων που ανήκουν σε αυτό το φύλλο οι οποίες περιγράφουν τις ελάχιστες οριοθετημένες περιοχές (MBR) των χώρων που καλύπτει το φύλλο. Με την μέθοδο MBR υπολογίζουμε όπως είπαμε και προηγουμένως το MBR του φύλλου για την εύρεση των ελάχιστων και μέγιστων τιμών για κάθε διάσταση (x, y, z) μέσω των καταχωρήσεων που περιέχονται στο φύλλο.

```
class LeafNode:
  def __init__(self, id: int, entries: list):
    self.id = id  # Αναγνψριστικό φύλλου
    self.entries = entries  # Καταχωρήσεις στα φύλλα

  # Δημιουργία και επιστροφή MBR κάθε φύλλου
  # και εύρεση μέγιστης και ελάχιστης τιμής για κάθε διάσταση (x, y, z)
  @property
  def mbr(self):
    min_x_l = min(e.x_low for e in self.entries)
    min_yl = min(e.y_low for e in self.entries)
    min_zl = min(e.z_low for e in self.entries)
    max_x_h = max(e.x_high for e in self.entries)
    max_y_h = max(e.y_high for e in self.entries)
    max_z_h = max(e.z_high for e in self.entries)

    return min_x_l, min_yl, min_zl, max_x_h, max_y_h, max_z_h
```

Στην συνέχεια φτιάχνουμε την **συνάρτηση calc** η οποία αναθέτει τον μέγιστο αριθμό καταχωρήσεων που μπορούν να υπάρχουν σε ένα κόμβο. Στην συνέχεια υπολογίζουμε τα **leaf_level_pages** που υπολογίζουν πόσα φύλλα (leaf nodes) θα απαιτούνται για να χωρέσουν στο συνολικό μέγεθος των δεδομένων, διαμοιρασμένο σε καταχωρήσεις με τον μέγιστο αριθμό n ανά κόμβο. Την μεταβλητή αυτή την χρησιμοποιούμε μόνο προκειμένου να υπολογιστεί ο αριθμός των φύλλων που θα χρειαστούν για την αποθήκευση των φύλλων του δέντρου.

```
def calc(size):
  n = 4  # Μέγιστος αριθμός σε κάθε κόμβο
  leaf_level_pages = ceil(size / n)  # Μέγιστος αριθμός φύλλων
  s = ceil(sqrt(leaf_level_pages))  # Μέγιστος αριθμός διαμοιράσεων
  return n, s
```

Τέλος, υλοποιούμε την **κλάση RTree** η οποία μας υπολογίζει το δέντρο μας. Αρχικά τα ορίσματα της κλάσης μας είναι:

- Το root που δείχνει την ρίζα του δέντρου.
- Το __id που χρησιμοποιείται για τη διατήρηση μοναδικών αναγνωριστικών κόμβων.
- Το max_cap που δείχνει τον μέγιστο αριθμό εισόδων που μπορεί να περιέχει ένας κόμβος.
- Το childList που δείχνει μια λίστα με τα παιδία του δέντρου.
- Το node_pool που δείχνει μια λίστα με όλους τους κόμβους του δέντρου.
- Το leaf_counter που καταγράφει τον αριθμό των LeafNode που έχουν προστεθεί.
- Το height που δείχνει το ύψος του R-tree.

Με την μέθοδο **insert_leaf(entries)** χειριζόμαστε την εισαγωγή νέων φύλλων στο R-tree με τις δοθείσες εισόδους. Πιο συγκεκριμένα καταγράφει τον συνολικό αριθμό των φύλλων στο δέντρο μας, με την εντολή **self.leaf_counter += 1** προσθέτει το αναγνωριστικό του νέου φύλλου σε μια λίστα με την εντολή **self.childList.append(self.getId**()) και δημιουργεί το φύλλο

και το προσθέτει σε μια λίστα κόμβων χρησιμοποιώντας την εντολή self.node_pool.append **(LeafNode (self.getId(),entries)).** Τέλος, αυξάνει τον μετρητή αναγνωριστικών για τα αναγνωριστικά των κόμβων με την εντολή **self.__increment().**

Ο κώδικας μας είναι ο εξής :

```
class RTree:
  def __init__(self, max_cap: int):
    self.root = None  # ρίζα δέντρου
    self.__id = 0  # Αναγνωριστικά κόμβων
    self.max_cap = max_cap  # Μέγιστος αριθμός εισόδων σε κόμβο
    self.childList = []  # Λίστα με παιδιά του δέντρου
    self.node_pool = []  # Λίστα με τους κόμβους του δέντρου

    self.leaf_counter = 0  # Αριθμός των leaf node που έχουν προστεθεί
    self.height = 1  #Ύψος του R-Tree

  # Συνάρτηση εισαγωγής φύλλων στο δέντρο
  def insert_leaf(self, entries):
    self.leaf_counter += 1  # Συνολικός αριθμός φύλλων στο δέντρο
    self.childList.append(self.getId())  # Προσθήκη αναγνωριστικού  νέου φύλλου στην λίστα
    self.node_pool.append(LeafNode(self.getId(), entries))  # Δημιουργία φύλλου
    self.__increment()  # Αύξηση αναγνωριστικών + 1
```

Με την **create_upper_levels** υπολογίζουμε πόσα νέα επίπεδα χρειάζονται για να χωρέσουν όλα τα παιδιά του τρέχοντος επιπέδου, με βάση τον μέγιστο αριθμό παιδιών που μπορεί να έχει κάθε κόμβος. Αν χρειάζεται μόνο ένα νέο επίπεδο, δημιουργείται ένας νέος κόμβος ρίζας όπου και προσθέτει τα παιδιά του τρέχοντος επιπέδου ως εγγόνια σε αυτόν. Στη συνέχεια, αυξάνει το ύψος του R-tree κατά ένα. Αν χρειάζονται περισσότερα από ένα νέα επίπεδα, καλεί την **insert_nodes()** για να δημιουργήσει τα απαραίτητα επιπλέον επίπεδα, αυξάνοντας επίσης το ύψος του R-tree κατά ένα.

```
def create_upper_levels(self):
  x = ceil(len(self.childList) / self.max_cap)
  # περίπτωση εισαγωγής μόνο ενός κόμβου
  if x == 1:
    self.root = Node(self.getId())  # Δημιουργία ενός νέου κόμβου ρίζας
    for idx in self.childList:
      child = self.node_pool[idx]  # Ανάκτηση των παιδιών του επιπέδου αυτού
      self.root.add_idx((child.id, child.mbr))  # προσθήκη ως εγγόνια σε αυτόν τον κόμβο

    self.node_pool.append(self.root)  # προσθήκη του νέου κόμβου στη λίστα με τους νέους κόμβους
    self.height += 1  # Αύξηση του ύψους του δέντρου κατά ένα
  # περίπτωση εισαγωγής περισσότερων του ενός κόμβου
  else:
    self.insert_nodes()  # κάλεσμα της συνάρτησης εισαγωγής κόμβων
    self.height += 1  # Αύξηση του ύψους του δέντρου κατά ένα
    self.create_upper_levels()  # επανάληψη της συνάρτησης
```

Με την συνάρτηση **insert_nodes** δημιουργούμε στο δέντρο μας νέους κόμβους στα επίπεδα πάνω από τα φύλλα του δέντρου μας για να διαχειριστεί τον υπερβολικό αριθμό παιδιών που υπερβαίνει τον μέγιστο επιτρεπτό αριθμό. Αρχικά, δημιουργούμε μια κενή λίστα buffer για την αποθήκευση αναγνωριστικών νέων κόμβων.

Έπειτα, διαχωρίζουμε τα αναγνωριστικά πεδία σε υποσύνολα μεγέθους self.max_cap και για κάθε τέτοιο υποσύνολο δημιουργούμε έναν νέο κόμβο με νέο αναγνωριστικό. Στην συνέχεια γίνεται προσθήκη των παιδιών στον νέο κόμβο και του νέου κόμβου στη λίστα self.node_pool, αντλώντας τα αναγνωριστικά και τα MBR από τα αντίστοιχα φύλλα του R-tree.

```
def insert_nodes(self):
  buffer = []  # Αναγνωριστικά νέων κόμβων
  child_indexes = [self.childList[i: i + self.max_cap] for i in
           range(0, len(self.childList), self.max_cap)]  # Θέτουμε την μέγιστη και ελάχιστη
χωρητικότητα
  # Για κάθε τέτοιο κομμάτι δημιουργούμε ένα νέο κόμβο και τον προσθέτουμε στο ευρετήριο
node_pool
  for chunk in child_indexes:
    buffer.append(self.getId())
    new_node = Node(self.getId())

    for idx in chunk:
      child = self.node_pool[idx]
      new_node.add_idx((child.id, child.mbr))

    self.node_pool.append(new_node)
    self.__increment()

  self.childList = buffer
```

Τέλος για να καταφέρουμε να εκτυπώσουμε το δέντρο μας χρησιμοποιούμε την μέθοδο printTree η οποία χρησιμοποιεί την __printNode. Πιο συγκεκριμένα η μέθοδος **__printNode** ελέγχει τον τύπο του κόμβου και εκτυπώνει τις κατάλληλες πληροφορίες ως εξής:

- Αν ο κόμβος είναι LeafNode, εκτυπώνει το MBR του και τις εγγραφές που περιέχει.
- Αν ο κόμβος είναι εσωτερικός κόμβος (Node), εκτυπώνει το MBR του και αναδρομικά καλεί τον εαυτό του για κάθε παιδί του.
- Αν ο κόμβος είναι άλλου τύπου (π.χ., Rect), εκτυπώνει τις κατάλληλες πληροφορίες.

```
def __printNode(self, node, depth, unique_mbrs, is_root=False):
  # Αν ο κόμβος είναι LeafNode, εκτυπώνει το MBR του και τις εγγραφές που περιέχει
  if isinstance(node, LeafNode):
    mbr = node.mbr
    if mbr not in unique_mbrs:
      # Αν ο κόμβος είναι ο root εκτύπωσε τον
      if is_root:
        print(f"Root - MBR: {mbr}")
      # Αν όχι εκτύπωσε το φύλλο του δέντρου σαν MBR και τις αντίστοιχες εγγραφές του
      else:
        print(f"{'  ' * depth}Leaf - MBR: {mbr}")
      unique_mbrs.add(mbr)
      for entry in node.entries:
        print(f"{'  ' * (depth + 1)}Entry: {entry.full_name}, {entry.y_low}, {entry.z_low}")
  # Αν ο κόμβος είναι Node, εκτυπώνει το MBR του και αναδρομικά καλεί τον εαυτό του για κάθε
παιδί του
  elif isinstance(node, Node):
    mbr = node.mbr
    if mbr not in unique_mbrs:
      if is_root:
        print(f"Root - MBR: {mbr}")
      else:
        print(f"{'  ' * depth}Node - MBR: {mbr}")
```

```
            unique_mbrs.add(mbr)
            for idx, (ptr, child_mbr) in enumerate(node.idx):
                child = self.node_pool[ptr]
                self.__printNode(child, depth + 2, unique_mbrs)
        # Αν ο κόμβος είναι Rect, εκτυπώνει τις κατάλληλες πληροφορίες που περιέχει
        elif isinstance(node, Rect):
            mbr = node.get_points
            if mbr not in unique_mbrs:
                print(f"{' ' * depth}Entry: Surname: {node.full_name}, MBR: {mbr}")
                unique_mbrs.add(mbr)
```

1.2 Αναζήτηση

Το range search σε ένα R δέντρο γίνεται με την βοήθεια ενός MBR στο οποίο εμείς αρχικοποιούμε τα άκρα του x,y,z, και σε κάθε επίπεδο αξιολογεί εάν το πλαίσιο οριοθέτησης ενός κόμβου τέμνεται με το καθορισμένο εύρος. Εάν υπάρχει μια τομή, ο αλγόριθμος εξερευνά αναδρομικά τους θυγατρικούς κόμβους του συγκεκριμένου κόμβου. Εάν δεν υπάρχει τομή, ολόκληρο το υποδέντρο που έχει ρίζες σε αυτόν τον κόμβο μπορεί να κλαδευτεί, καθώς δεν μπορεί να περιέχει αντικείμενα εντός του καθορισμένου εύρους. Το range search, ή και αλλιώς αναζήτηση εύρους, συνεχίζεται έως ότου ο αλγόριθμος φτάσει στους κόμβους φύλλων, οι οποίοι αντιπροσωπεύουν μεμονωμένα αντικείμενα ή μικρές ομάδες αντικειμένων. Τα αντικείμενα εντός των κόμβων φύλλων που τέμνονται με το καθορισμένο εύρος επιστρέφονται στη συνέχεια ως αποτέλεσμα της αναζήτησης εύρους.

Για την σωστή υλοποίηση της διαδικασίας αυτής δημιουργούμε μέσα στην κλάση RTree την συνάρτηση **range_search** η οποία σαν όρισμα δέχεται το MBR του ερωτήματος που θα θέσουμε αλλά και το threshold ομοιότητας για την εφαρμογή του LSH.

Πρώτα, δημιουργείται μια λίστα result_entries στην οποία θα αποθηκεύουμε τα αποτελέσματα της εκάστοτε αναζήτησης. Έπειτα, καλείται η συνάρτηση **__range_search_recursive** με είσοδο την ρίζα του δέντρου μας, το MBR του ερωτήματος, την παραπάνω λίστα αλλά και ένα αναγνωριστικό συμβολοσειράς για τη ρίζα. Η συνάρτηση **__range_search_recursive** έχει ως βασικό στόχο να διασχίζει αναδρομικά τη δομή του δέντρου R και να προσδιορίζει εγγραφές που ικανοποιούν τα καθορισμένα κριτήρια αναζήτησης εύρους εντός του δεδομένου MBR ερωτήματος. Αν ο αρχικός μας κόμβος είναι φύλλο του δέντρου, δηλαδή LeafNode, τότε επαναλαμβάνεται πάνω από κάθε καταχώρηση στη λίστα καταχωρήσεων του κόμβου. Για κάθε καταχώρηση, ελέγχει εάν βρίσκεται μέσα στο MBR ερωτήματος χρησιμοποιώντας την μέθοδο **__is_entry_inside_query**. Εάν η καταχώριση ικανοποιεί τα κριτήρια, προστίθεται στη λίστα **result_entries**. Εάν ο τρέχων κόμβος δεν αποτελεί φύλλο του δέντρου τότε επαναλαμβάνεται πάνω από κάθε θυγατρικό του κόμβου. Για κάθε παιδί, ανακτά το MBR του παιδιού και ελέγχει εάν το MBR βρίσκεται μέσα στο ορθογώνιο ερωτήματος χρησιμοποιώντας την ιδιωτική μέθοδο **__is_mbr_inside_query**. Εάν το MBR ικανοποιεί τα κριτήρια, η μέθοδος καλείται αναδρομικά στον θυγατρικό κόμβο.

Οι συναρτήσεις **__is_entry_inside_query** και **__is_mbr_ inside_query** ελέγχουν από την μεριά τους τα εξής:

__is_entry_inside_query:

Αν μια δεδομένη καταχώρηση (ένα στιγμιότυπο της κλάσης Rect) βρίσκεται μέσα σε ένα καθορισμένο ορθογώνιο ερωτήματος. Επιστρέφει True εάν οι συντεταγμένες της καταχώρισης εμπίπτουν στα αντίστοιχα όρια του ορθογωνίου ερωτήματος και False διαφορετικά.

__is_mbr_inside_query:

Αυτή η μέθοδος ελέγχει εάν ένα δεδομένο MBR βρίσκεται μέσα σε ένα καθορισμένο ορθογώνιο ερωτήματος. Το MBR αναπαρίσταται ως πλειάδα που περιέχει τις ελάχιστες και μέγιστες συντεταγμένες κατά μήκος κάθε διάστασης.

Ο κώδικας είναι ο ακόλουθος:

```
def range_search(self, query_rect, similarity):
    result_entries = set()
    self.__range_search_recursive(self.root, query_rect, result_entries, full_name='Root')

    print(f"Performing range search with query rectangle: {query_rect.get_points}")
for entry in result_entries:
    print(f"Entry: {entry.full_name}, {entry.y_low}, {entry.z_low}")


def __range_search_recursive(self, node, query_rect, result_entries, full_name=None):
    if isinstance(node, LeafNode):
        for entry in node.entries:
            if self.__is_entry_inside_query(entry, query_rect):
                result_entries.add(entry)
    elif isinstance(node, Node):
        for idx, (ptr, child_mbr) in enumerate(node.idx):
            child = self.node_pool[ptr]
            if self.__is_mbr_inside_query(child_mbr, query_rect):
                child_full_name = f"{full_name} - Child {idx+1}"
                self.__range_search_recursive(child, query_rect, result_entries, full_name=child_full_name)
def __is_entry_inside_query(self, entry, query_rect):
    return (
            query_rect.x_low <= entry.x_low <= query_rect.x_high and
            query_rect.y_low <= entry.y_low <= query_rect.y_high and
            query_rect.z_low <= entry.z_low <= query_rect.z_high
    )

def __is_mbr_inside_query(self, mbr, query_rect):
    return (
            query_rect.x_low <= mbr[3] and
            query_rect.x_high >= mbr[0] and
            query_rect.y_low <= mbr[4] and
            query_rect.y_high >= mbr[1] and
            query_rect.z_low <= mbr[5] and
            query_rect.z_high >= mbr[2]
    )
```

1.3 Αναζήτηση με LSH

Για την αναζήτηση με LSH, δημιουργούμε μια λίστα **education_data** για να ανακτήσουμε την εκπαίδευση του κάθε επιστρεφόμενου κόμβου από το αρχείο μας. Σε ένα σύνολο unique_surnames θα εισάγουμε τα μοναδικά επώνυμα με έγκυρη εκπαίδευση. Για κάθε κόμβο από αυτούς εξάγουμε τις τρεις συντεταγμένες τους (surname, awards, dblp) και ελέγχουμε αν έχει ήδη προστεθεί μια εγγραφή με έγκυρη εκπαίδευση για αυτό το επώνυμο. Αν έχει προστεθεί, η επόμενη εγγραφή παραλείπεται. Αυτό το βήμα είναι απαραίτητο για να εξαλείψουμε τις διπλότυπες εγγραφές σε περίπτωση που έχουμε δύο επιστήμονες με ίδιες συντεταγμένες αλλά ο ένας από αυτούς δεν έχει έγκυρη εκπαίδευση, οπότε δεν πρέπει να

συμπεριληφθεί κατά την εκτέλεση του LSH. Αφού εξαλείψουμε τα διπλότυπα, εξάγουμε το αντίστοιχο κείμενο εκπαίδευσης για το κάθε επώνυμο.

Στη συνέχεια, τα τελικά έγκυρα δεδομένα που συλλέχθηκαν για κάθε επιστήμονα (επώνυμο, βραβεία, δημοσιεύσεις, εκπαίδευση) προστίθενται στη λίστα **data** .

Τέλος, από αυτό το Dataframe εξάγουμε τη στήλη με τα δεδομένα εκπαίδευσης στη λίστα **education_texts**, την οποία **κανονικοποιούμε** με την βοήθεια την συνάρτησης **preprocess_education**. Καλούμε έπειτα την **lsh_education( )** η οποία παίρνει σαν **είσοδο** τα **education_texts_preprocessed, result_df** και το **threshold_education**, έτσι ώστε να επιστρέψουμε τους επιστήμονες με παρόμοια εκπαίδευση που πληρούν τα κριτήρια του εύρους αναζήτησης.

Τέλος, εκτυπώνουμε τους επιστήμονες που επιστρέφει η lsh_education μαζί με τα δεδομένα που τους αντιστοιχούν για πλήθος βραβείων και Dblp.

Τα αποτελέσματα αποθηκεύονται στο csv αρχείο **similar_pairs_education.csv**.

```
# κενή λίστα για ανάκτηση των αντίστοιχων δεδομένων εκπαίδευσης για κάθε κόμβο
file_path = "scientists.csv"
df = pd.read_csv(file_path)  # δημιουργία data frame από CSV αρχείο

data = []  # λίστα για αποθήκευση κόμβων μαζί με εκπαίδευση
unique_surnames = set()  # set για διαχείριση διπλότυπων
result_list = []
for entry in result_entries:
    surname = entry.full_name
    awards = entry.y_low
    dblp = entry.z_low

    # εισαγωγή ονομάτων, όχι διπλότυπων
    if surname in unique_surnames:
        continue

    # εξαγωγή δεδομένων εκπαίδευσης για το LSH
    education_text = df.loc[
        (df['Surname'] == surname) & (df['Awards'] == awards) & (df['Dblp'] == dblp),
        'Education'
    ].values[0]

    # ελεγχος για 'Education information not found'
    if education_text != 'Education information not found':
        unique_surnames.add(surname)  # πρόσθεση των επιστημόνων που έχουν δεδομένα εκπαίδευσης
        data.append([surname, awards, dblp, education_text])

    # μετατροπή του CSV σε DataFrame
    csv_columns = ['Surname', 'Awards', 'Dblp', 'Education']
    result_df = pd.DataFrame(data, columns=csv_columns)

# Αρχικοποίηση του threshold με κατάλληλη τιμή
threshold_education = similarity

# LSH για 'Education'
education_texts = result_df['Education'].tolist()
education_texts_preprocessed = [preprocess_education(edu) for edu in education_texts]
```

```
start2 = time.time()
similar_pairs_education = lsh_education(education_texts_preprocessed, result_df,
threshold_education)
end2 = time.time()

elapsed_time2 = end2 - start2

# Αποθηκεύουμε τα μοναδικά ονόματα από όλα τα ζευγάρια με κοινή εκπαίδευση
unique_surnames_in_pairs = set()

for pair in similar_pairs_education:
  (entry1, surname1), (entry2, surname2) = pair
  if surname1 in unique_surnames and surname1 not in unique_surnames_in_pairs:
    awards1, dblp1 = df.loc[df['Surname'] == surname1, ['Awards', 'Dblp']].iloc[0]
    print(f"Surname: {surname1}, #Awards: {awards1}, #DBLP_Record: {dblp1}")
    unique_surnames_in_pairs.add(surname1)
  if surname2 in unique_surnames and surname2 not in unique_surnames_in_pairs:
    awards2, dblp2 = df.loc[df['Surname'] == surname2, ['Awards', 'Dblp']].iloc[0]
    print(f"Surname: {surname2}, #Awards: {awards2}, #DBLP_Record: {dblp2}")
    unique_surnames_in_pairs.add(surname2)

    # μετατροπή αποτελεσμάτων results σε DataFrame
result_df_education = pd.DataFrame(similar_pairs_education, columns=['Entry1', 'Entry2'])
result_df_education.to_csv('similar_pairs_education.csv', index=False)

# Υπολογισμός συνολικής χρονομέτρησης
print('Execution time:', str(elapsed_time1 + elapsed_time2), 'seconds')
```

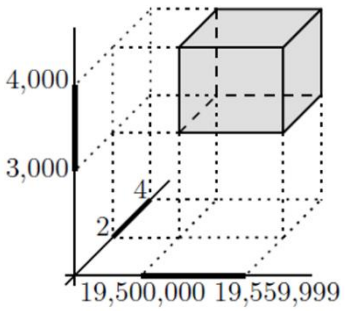
Αφού διαβάσουμε το csv αρχείο με τα δεδομένα εισαγωγής στο δέντρο, δημιουργούνται τα αντικείμενα **Rect,** χρησιμοποιώντας δεδομένα από το DataFrame, τα οποία αντιπροσωπεύουν μεμονωμένες εγγραφές στο σύνολο δεδομένων. Στην συνέχεια μετατρέπουμε το σύνολο των αντικειμένων Rect σε μια λίστα και το ταξινομούμε με βάση ορισμένα χαρακτηριστικά. Υπολογίζουμε τη μέγιστη χωρητικότητα (max_cap) και τον αριθμό των τμημάτων (slices) χρησιμοποιώντας τη συνάρτηση calc και στην συνέχεια χωρίζουμε τα ταξινομημένα δεδομένα με βάση τις υπολογισμένες τιμές max_cap και s. Τέλος αρχικοποιούμε την παρουσία της κλάσης RTree με το υπολογισμένο max_cap και στην συνέχεια γίνεται η εισαγωγή κόμβων φύλλων στο R-tree με χρήση των μεθόδων **insert_leaf** και **create_upper_levels** όπως φαίνεται παρακάτω.

```
def main_R_tree():
  data_set = set()
  file_path = "scientists.csv"   # Ανάγνωση δεδομένων εισαγωγής
  df = pd.read_csv(file_path)
  # Δημιουργεία αντικειμένων RECT με βάση τα δεδομένα που διαβάσαμε
  for idx, row in df.iterrows():
    entry = Rect(
      id=idx,
      x_low=row['Surname'],
      y_low=row['Awards'],
      z_low=row['Dblp'],
      x_high=row['Surname'],
      y_high=row['Awards'],
      z_high=row['Dblp'],
      full_name=row['Surname'],
      education=row['Education']
    )
```

```
        data_set.add(entry)
    # Μετατροπή αντικειμένων RECT σε λίστα
    data = list(data_set)
    # ταξινόμση των αντικειμένων με βάση τις διαστάσεις (x,y,z)
    data.sort(key=lambda entry: (entry.x_low, entry.y_low, entry.z_low, entry.x_high, entry.y_high,
entry.z_high,entry.full_name))

    # Μέγιστη χωριτικότητα και slices
    max_cap, s = calc(len(data))
    # Διαχωρίζουμε τα ταξινομηένα δεδομένα με βάση τις προιγούμενες τιμές
    data = [data[x:x + (s * max_cap)] for x in range(0, len(data), s * max_cap)]
    # Αρχικοποιούμε την κλάση του δέντρου μας
    tree = RTree(max_cap)

# Εισάγουμε τα διαχωρισμένα και ταξινομημένα αντικείμενα μας με βάση στο δέντρο μας
for sublist in data:
    for i in range(0, len(sublist), max_cap):
        tree.insert_leaf(sublist[i: i + max_cap])
tree.create_upper_levels()

# εμφάνιση δέντρου
# tree.printTree()

first_letter = input("Press the first letter of the query (ex.A, B, C...):").upper()
while not first_letter.isalpha():
    first_letter = input("Press the first letter you want in the right form:")
second_letter = input("Press the second letter of the query (ex.A, B, C...):").upper()
while not second_letter.isalpha():
    second_letter = input("Press the second letter you want in the right form:")
min_awards = int(input("Press the minimum awards of the query (ex.0,1,2...) :"))
max_awards = math.inf
min_dblps = int(input("Press the minimum DBLP records of the query (ex.0,1,2...) :"))
max_dblps = int(input("Press the maximum DBLP records of the query (ex.0,1,2...) :"))
lsh_sim = float(input("Give the LSH percentage of similarity between the scientists (ex.0.5->50%):"))
print("The query that you created is the following ! ")
print(
    "Find the computer scientists from Wikipedia that their letters belongs in the interval [" + first_letter +
"-" + second_letter + "],have won more than " + str(
        min_awards) + " awards and their DBLP record belongs in the interval [" + str(
        min_dblps) + " ," + str(max_dblps) + "] !")
print("Generating Answer")
# Υπολογισμός Range search
query_rect = Rect(id=-1, x_low=first_letter, y_low=min_awards, z_low=min_dblps,
x_high=second_letter,
                  y_high=max_awards, z_high=max_dblps, full_name='', education='')

# Εμφάνιση αποτελεσμάτων Range search
tree.range_search(query_rect, lsh_sim)
```

2 Range tree

Το Range Tree είναι μια δομή δεδομένων που χρησιμοποιείται κατά κύριο λόγο για την επίλυση προβλημάτων που σχετίζονται με την αναζήτηση σε πολυδιάστατες δομές-πίνακες.

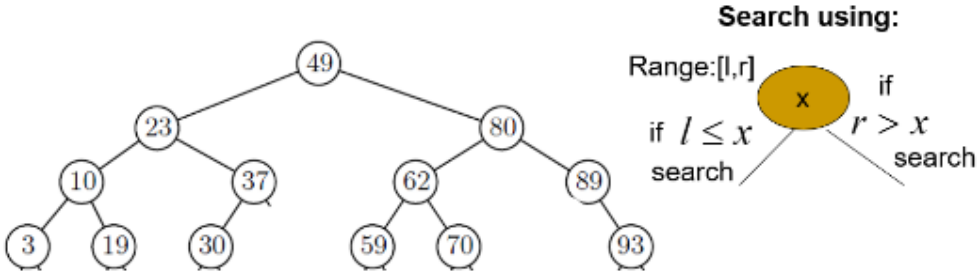


Στο διπλανό παράδειγμα βλέπουμε την αναζήτηση δεδομένων ταυτόχρονα σε 3 διαστάσεις. Τόσο στον άξονα x όσο και στον άξονα y και z, αναζητούμε τα σημεία μεταξύ των ορίων που θα σχηματίσουν έναν κύβο που θα «περικλείει» τα ανακτηθέντα δεδομένα.

Οι βασικές λειτουργίες που μπορεί να υποστηρίξει ένα Range Tree είναι η εισαγωγή ενός στοιχείου, η αναζήτηση στοιχείων σε ένα συγκεκριμένο εύρος και η διαγραφή ενός στοιχείου. Στην δική μας υλοποίηση χρησιμοποιούμε τις δύο πρώτες λειτουργίες που περιγράφονται παρακάτω.

Το συγκεκριμένο δέντρο ανάλογα με τον αριθμό διαστάσεων του μπορεί να πάρει διαφορετική μορφή. Παρακάτω αναλύουμε μεθόδους υλοποίησης του καθώς και τον τρόπος υλοποίησης στον δικό μας κώδικα.

Ένα **1D Range Tree** αποτελεί ένα δυαδικό δέντρο αναζήτησης (Balanced Binary Search Tree) με ταξινομημένα όλα τα στοιχεία ενός συνόλου. Η αναζήτηση σε αυτό το δέντρο δεν παρουσιάζει κάποια ιδιαίτερη δυσκολία αφού γίνεται μια δυαδική αναζήτηση ελέγχοντας κάθε φορά αν η τιμή του τρέχον κόμβου είναι μικρότερη ή μεγαλύτερη από τον κόμβο-στόχο.



Έτσι λοιπόν αν θέλουμε να κάνουμε μία αναζήτηση παραδείγματος χάρη μεταξύ του διαστήματος [19,70] θα χρειαστεί να γίνει δύο δυαδικές αναζητήσεις. Μία για την αναζήτηση του αριστερού άκρου (19) και μία για την αναζήτηση του δεξιού άκρου (70). Οι δύο αυτές αναζητήσεις απαιτούν χρόνο O(logn), όπου n ο αριθμός των κόμβων στο δέντρο. Ωστόσο η διαδικασία εύρεσης των ενδιάμεσων στο διάστημα κόμβων απαιτεί χρόνο O(k) όπου k= ο αριθμός των επιστρεφόμενων κόμβων.

Επομένως απαιτεί συνολικό χρόνο **O(logn + κ)**.

Ένα **ND Range Tree**

Όταν αυξάνονται οι διαστάσεις του Range Tree, η διαδικασία υλοποίησης αλλάζει.

Ένα πολυδιάστατο δέντρο διαστημάτων μπορεί να υλοποιηθεί με δύο τρόπους. Είτε με πολλαπλά δέντρο N όσα και οι διαστάσεις N είτε με ένα δέντρο το οποίο θα περιέχει κόμβους με N στοιχεία, όσες και οι διαστάσεις που δέντρου.

Πολλαπλά δέντρα

Στην προκειμένη περίπτωση το αρχικό μας δέντρο αποτελεί ένα δυαδικό δέντρο με τα δεδομένα των συντεταγμένων x. Κάθε κόμβος του δέντρου αυτού δείχνει σε ένα άλλο μονοδιάστατο δυαδικό δέντρο το οποίο περιέχει τα δεδομένα με συντεταγμένες y. Αν μιλάμε για 3 διαστάσεις, από το δεύτερο δέντρο θα ξανά δείχνει κάθε κόμβος σε ένα τρίτο δυαδικό δέντρο που περιέχει τα δεδομένα με z συντεταγμένες κοκ. Η συγκεκριμένη τεχνική χρησιμοποιεί πολλαπλά δέντρα τα οποία δείχνουν το ένα στο άλλο.

Ένα δέντρο

Μια διαφορετική υλοποίηση του range tree είναι η δημιουργία του με ένα δέντρο και όχι πολλαπλά. Αυτή αποτελεί και την τεχνική την οποία ακολουθήσαμε. Στόχος μας είναι η δημιουργία ενός δέντρου που περιέχει τα δεδομένα και εναλλάσσει μεταξύ συντεταγμένων x, y, z απλά χωρίς να υλοποιεί πολλαπλά δυαδικά δέντρα. Παρακάτω αναλύουμε τον κώδικα για την υλοποίηση του.

2.1 Κατασκευή

Αρχικά εισάγουμε την βιβλιοθήκη pandas έτσι ώστε να διαχειριστούμε τα δεδομένα από το CSV αρχείο ως data frames.

```
import pandas as pd
```

Έπειτα δημιουργούμε μια κλάση για να περιγράψουμε τους κόμβους του δέντρου.

```
class Node:
  def __init__(self, point, left=None, right=None):
    self.point = point
    self.left = left
    self.right = right
```

Κάθε κόμβος αποτελείται από σημεία points (x, y, z) και έχει ένα δεξί και ένα αριστερό υποδέντρο, left και right.

Δημιουργούμε μία ακόμα κλάση για το δέντρο, μέσα στην οποία αρχικοποιούμε την ρίζα root.

```
class ThreeDTree:
  def __init__(self):
    self.root = None
```

Μέσα στην κλάση δημιουργούμε μία συνάρτηση για την δημιουργία του δέντρου την οποία θα καλέσουμε μετά με τα κατάλληλα δεδομένα από το csv. Αυτή παίρνει σαν είσοδο τα σημεία points που περιέχουν τα δεδομένα με την εξής σειρά: (‘Surname’, ‘Awards’, ‘Dblps’), και τον αριθμό τον διαστάσεων **dim** (dimentions) αρχικοποιημένο με 0. Αυτή η συνάρτηση μας επιστρέφει την ρίζα του δέντρου εφόσον αυτό δεν είναι άδειο.

Επειδή το δέντρο είναι 3 διαστάσεων, για να επαναλάβουμε για κάθε διάσταση την εισαγωγή δεδομένων αρχικοποιήσαμε την διάσταση με 0. Με την εντολή **dim %= 3**, δημιουργούμε έναν

βρόχο επανάληψης τον εντολών για τις διαστάσεις 0,1,2 (3D). Όπως προαναφέραμε, τα σημεία-κόμβοι αποτελούνται από 3 στοιχεία, επόμενος το **point[0]** αναφέρεται στο Surname, το **point[1]** στην στήλη των Awards και το **point[2]** στην στήλη με τα Dbpls.

Εμείς για να ταξινομήσουμε τα στοιχεία σε κάθε διάσταση σύμφωνα τις συντεταγμένες x, y, z αντίστοιχα χρησιμοποιούμε την μέθοδο **.sort**. Έτσι για την διάσταση 0 ταξινομούμε με βάση το Surname, για την διάσταση 1 με βάση τα Awards και για την διάσταση 2 με βάση τα Dbpls. Κάθε φορά, για κάθε διάσταση βρίσκουμε το μέσον **mid** της λίστας points και το θέτουμε ως ρίζα. Τέλος δημιουργούμε τα αριστερα και δεξιά υποδέντρα (left/right – subtrees) από την ρίζα, καλώντας ξανά την συνάρτηση **create_tree**, παίρνοντας για το αριστερό υποδέντρο τα σημεία από την αρχή μέχρι το μέσο της λίστας, ενώ για το δεξί υποδέντρο τα σημεία από το μέσο της λίστας, αυξάνοντας την διάσταση κατά ένα κάθε φορά έως ότου φτάσουμε τις 3 διαστάσεις (**dim =2**).

```
def create_tree(self, points, dim=0):
    if not points:
        return None

    dim %= 3  # επαμάληψη κάθε φορά για αριθμό διάστασης απο 1 έως 3
    points.sort(key=lambda p: p[dim])  # ταξινόμησης της λίστας με τα σημεία σλυμφωνα με την
τρέχουσα διάσταση
    mid = int(len(points) // 2)  # το μέσο της λίστας
    root = Node(points[mid])  # κάνει το μέσο ρίζα

    root.left = self.create_tree(points[:mid], dim + 1)  # δημιουργία αριστερού υποδέντρου
    root.right = self.create_tree(points[mid + 1:], dim + 1)  # δημιουργία δεξιού υποδέντρου

    return root
```

Επόμενη συνάρτηση που δημιουργούμε είναι για την εισαγωγή δεδομένων στο δέντρο.

```
def insert_data_from_csv(self, csv_file_path):  # εισαγωγή δεδομένων απο το csv
    df = pd.read_csv(csv_file_path)  # δημιουργία df με το csv
    points = df[['Surname', 'Awards', 'Dblp']].apply(
        lambda row: (row['Surname'], row['Awards'], row['Dblp']),
        axis=1).tolist()  # τα points είναι οι στήλες Surname, Awards και Dblp
    self.root = self.create_tree(points)  # δημιουργία δέντρου για τα points αυτά
```

Στο παραπάνω κομμάτι κώδικα η συνάρτηση παίρνει σαν είσοδο το csv αρχείο, του οποίου το path ορίζουμε παρακάτω, με τα δεδομένα για όλους του επιστήμονες και δημιουργεί ένα data frame. Ορίζουμε τα σημεία του δέντρου ως μία λίστα με τις στήλες Surname, Awards και Dblp του αρχείου. Τέλος καλούμε την create tree με την παραπάνω λίστα δεδομένων.

Με την παραπάνω διαδικασία έχουμε ολοκληρώσει την εισαγωγή των δεδομένων στο δέντρο μας τα οποία με την **create_tree** θα ταξινομηθούνε για κάθε διάσταση ξεχωριστά στον τρισδιάστατο χ.

2.2 Αναζήτηση

Μας ζητήθηκε να αναζητήσουμε στο δέντρο και να επιστρέψουμε τα δεδομένα που βρίσκονται ανάμεσα σε διαστήματα που ορίζει το χρήστης. Στο πρόγραμμα μας ζητούνται να εισαχθούν maximum και minimum γράμματα μέσα στα οποία θα βρίσκονται τα ονόματα των επιστημόνων,

ελάχιστο αριθμό βραβείων που έχει λάβει ο επιστήμονας(ανοικτό διάστημα), καθώς και ελάχιστο και μέγιστο όριο αναγνωρίσεων.

Ορισμός διαστημάτων αναζήτησης από τον χρήστη:

```
min_surname = input("Press the first letter of the query (ex.A, B, C...):").upper()
while not min_surname.isalpha():
  min_surname = input("Press the first letter you want in the right form:")
max_surname = input("Press the second letter of the query (ex.A, B, C...):").upper()
while not max_surname.isalpha():
  max_surname = input("Press the second letter you want in the right form:")
min_awards = int(input("Press the minimum awards of the query (ex.0,1,2...) :"))
min_dblps = int(input("Press the minimum DBLP records of the query (ex.0,1,2...) :"))
max_dblps = int(input("Press the maximum DBLP records of the query (ex.0,1,2...) :"))
threshold_education = float(input("Give the LSH percentage of similarity between the scientists
(ex.0.5->50%):"))
```

Αφού διαβαστούν τα παραπάνω όρια των διαστημάτων, καλείται η συνάρτηση **range_search** η οποία παίρνει σαν όρισμα τα όρια αυτά και την ρίζα του δέντρου.

Αρχικοποιούμε τις διστάσεις με 0 έτσι ώστε να επαναλάβουμε την αναζήτηση για τις 3 διαστάσεις (0, 1, 2) όπως παραπάνω.

```
def range_search(self, root, min_surname, max_surname, min_awards, max_awards, min_dblps,
max_dblps,
           dim=0, result=None):
  if result is None:
    result = []

  if root is None:  # αν το δέντρο είναι κενό
    return result

  dim %= 3

  # έλεγχος για το αν η ρίζα είναι μεταξύ του διαστήματος σε κάθε διάσταση
  if (
      (min_surname <= root.point[0] < max_surname or root.point[0].startswith(max_surname)) and
      (min_awards <= root.point[1] <= max_awards) and
      (min_dblps <= root.point[2] <= max_dblps)
  ):
    result.append(root)
```

Αρχικά ελέγχουμε αν οι κάθε μία από τις τρεις ρίζες βρίσκονται εντός των διαστημάτων που ορίζουμε. Αν ναι, τότε εισάγουμε το σημείο στην λίστα αποτελεσμάτων **results**. Στην συνέχεια για κάθε διάσταση από 0 έως 2 ελέγχουμε αν οι συντεταγμένες κάθε σημείου που εισαγάγαμε στην λίστα results είναι εντός των ορίων. Αν το τρέχων στοιχείο(ρίζα) είναι μεγαλύτερο από το αριστερό άκρο του διαστήματος x>l, τότε μεταβαίνουμε αριστερά του (αριστερό υποδέντρο) και αυξάνουμε την διάσταση κατά ένα έτσι ώστε στην συνέχεια να ελέγξουμε την δεύτερη συντεταγμένη του σημείου (y) και εν συνέχεια την τρίτη συντεταγμένη του σημείου (z).

```
# εναλλαγή δεξιού και αριστερού υποδέντρου σύμφωνα με την τρέχουσα διάσταση
if dim == 0:
  if min_surname < root.point[dim] or root.point[0].startswith(min_surname):
    self.range_search(root.left, min_surname, max_surname, min_awards, max_awards,
              min_dblps, max_dblps, (dim + 1) % 3, result)
  if max_surname > root.point[dim] or root.point[0].startswith(max_surname):
```

```
        self.range_search(root.right, min_surname, max_surname, min_awards, max_awards,
                          min_dblps, max_dblps, (dim + 1) % 3, result)
    elif dim == 1:
        if min_awards <= root.point[dim]:
            self.range_search(root.left, min_surname, max_surname, min_awards, max_awards,
                              min_dblps, max_dblps, (dim + 1) % 3, result)
        if max_awards >= root.point[dim]:
            self.range_search(root.right, min_surname, max_surname, min_awards, max_awards,
                              min_dblps, max_dblps, (dim + 1) % 3, result)
    elif dim == 2:
        if min_dblps <= root.point[dim]:
            self.range_search(root.left, min_surname, max_surname, min_awards, max_awards,
                              min_dblps, max_dblps, (dim + 1) % 3, result)
        if max_dblps >= root.point[dim]:
            self.range_search(root.right, min_surname, max_surname, min_awards, max_awards,
                              min_dblps, max_dblps, (dim + 1) % 3, result)

    return result
```

Με τον παραπάνω κώδικα επιτυγχάνουμε την εύρεση των κόμβων που βρίσκονται μέσα στα δοθέντα διαστήματα, εξετάζοντας κάθε φορά διαφορετική διάσταση άρα διαφορετικές συντεταγμένες των σημείων.

Στο τελικό μας βήμα, δημιουργούμε ένα στιγμιότυπο της κλάσης **ThreeDTree**. Αυτό αποτελεί και το τελικό μας δέντρο.

Ορίζουμε το αρχείο από το οποίο θα διαβάσομε τα δεδομένα και καλούμε την συνάρτηση **insert_data_from_csv** για το δέντρο που μόλις δημιουργήσαμε.

```
# εισαγωγή δεδομένων στο δέντρο απο το CSV αρχείο
csv_file_path = 'scientists.csv'  # file path
tree.insert_data_from_csv(csv_file_path)
```

Έπειτα ορίζουμε τα αποτελέσματα να είναι οι κόμβοι που μας επιστρέφει η συνάρτηση του range search. Θέτουμε ως άνω όριο του διαστήματος των βραβείων ίσο με έναν μεγάλο αριθμό γιατί το συγκεκριμένο διάστημα δεν είναι κλειστό στο δεξιά όριο.

```
# αποθήκευση επιστρεφόμενων κόμβων
results = tree.range_search(tree.root, min_surname, max_surname, min_awards, float('inf'),
min_dblps, max_dblps)

# εμφάνιση των κόμβων μέσα στο διάστημα αναζήτησης
print(f"Points within the specified range:")
for node in results:
    print(node.point)
```

Εμφανίζουμε τους κόμβους που βρίσκονται ανάμεσα μέσα στα διαστήματα και συνεχίζουμε την εύρεση των κόμβων αυτών από τους επιστρεφόμενους που έχουν ορισμένη ομοιότητα.

Σκοπός μας είναι επίσης να χρονομετρήσουμε την διάρκεια που απαιτεί το πρόγραμμα να δημιουργήσει το δέντρο και να επιστρέψει τους επιστήμονες που έχουν ποσοστό ομοιότητας άνω του προκαθορισμένου, από τον χρήστη, ποσοστού. Για τον λόγο αυτός χρησιμοποιούμε την εντολή start_time = time.time() για να ξεκινήσει η μέτρησει του χρόνου για τον υπολογισμό του. Η έναρξη χρονομέτρησης ξεκινάει ακριβώς πριν την δημιουργία του δέντρου και

ολοκληρώνεται πριν την εκτύπωση των τελικών αποτελεσμάτων lsh όπως φαίνεται παρακάτω.

2.3 Αναζήτηση με LSH

Για να ανακτήσουμε την εκπαίδευση του κάθε επιστρεφόμενου κόμβου από το αρχείο μας ακολουθούμε την παρακάτω διαδικασία:

Δημιουργούμε μια λίστα **education_data** για την αποθήκευση των δεδομένων εκπαίδευσης των παραπάνω επιστρεφόμενων επιστημόνων και ένα σύνολο **unique_surnames** για να παρακολουθούμε τα μοναδικά επώνυμα με έγκυρη εκπαίδευση. Για κάθε κόμβο από αυτούς (point) εξάγουμε τις τρεις συντεταγμένες τους (surname, awards, dblp) και ελέγχουμε αν έχει ήδη προστεθεί μια εγγραφή με έγκυρη εκπαίδευση για αυτό το επώνυμο. Αν έχει προστεθεί, η επόμενη εγγραφή παραλείπεται. Αυτό το βήμα είναι απαραίτητο για να εξαλείψουμε τις διπλότυπες εγγραφές σε περίπτωση που έχουμε δύο επιστήμονες με ίδιες συντεταγμένες αλλά ο ένας από αυτούς δεν έχει έγκυρη εκπαίδευση, οπότε δεν πρέπει να συμπεριληφθεί κατά την εκτέλεση του LSH. Αφού εξαλείψουμε τα διπλότυπα, εξάγουμε το αντίστοιχο κείμενο εκπαίδευσης για το κάθε επώνυμο.

```
education_data = []
df = pd.read_csv(csv_file_path)  # δημιουργία data frame από CSV αρχείο

data = []  # λίστα για αποθήκευση κόμβων μαζί με εκπαίδευση
unique_surnames = set()  # set για διαχείρηση διπλότυπων

for node in results:
    point = node.point
    surname = point[0]
    awards = point[1]
    dblp = point[2]

    # εισαγωγή ονομάτων, όχι διπλότυπων
    if surname in unique_surnames:
        continue

    # εξαγωγή δεδομένων εκπαίδευσης για το LSH
    education_text = df.loc[
        (df['Surname'] == surname) & (df['Awards'] == awards) & (df['Dblp'] == dblp),
        'Education'
    ].values[0]

    # ελεγχος για 'Education information not found'
    if education_text != 'Education information not found':
        unique_surnames.add(surname)  # πρόσθεση των επιστημόνων που έχουν δεδομένα εκπαίδευσης
        data.append([surname, awards, dblp, education_text])
# μετατροπή του CSV σε DataFrame
csv_columns = ['Surname', 'Awards', 'Dblp', 'Education']
result_df = pd.DataFrame(data, columns=csv_columns)
# αποθήκευση κόμβων σε αρχείο csv
result_df.to_csv('query_results.csv', index=False)
```

Στη συνέχεια, τα τελικά έγκυρα δεδομένα που συλλέχθηκαν για κάθε επιστήμονα (επώνυμο, βραβεία, δημοσιεύσεις, εκπαίδευση) προστίθενται στη λίστα **data** την οποία μετατρέπουμε και σε Dataframe με όνομα **result_df** όπως φαίνεται παραπάνω.

Τέλος, από αυτό το Dataframe εξάγουμε τη στήλη με τα δεδομένα εκπαίδευσης στη λίστα **education_texts**, την οποία **κανονικοποιούμε** με την βοήθεια την συνάρτησης **preprocess_education**. Καλούμε έπειτα την **lsh_education( )** η οποία παίρνει σαν **είσοδο** τα **education_texts**, το **result_df** και το **κατώφλι** που ορίζει ο χρήστης, έτσι ώστε να κρατήσουμε τα παρόμοια κείμενα εκπαίδευσης με βάση τους επιστήμονες που πληρούν τα κριτήρια του εύρους αναζήτησης.

Τέλος, εκτυπώνουμε τους επιστήμονες που επιστρέφει η lsh_education μαζί με τα δεδομένα που τους αντιστοιχούν για πλήθος βραβείων και Dblp.

Τα αποτελέσματα αποθηκεύονται στο csv αρχείο **similar_pairs_education.csv**.

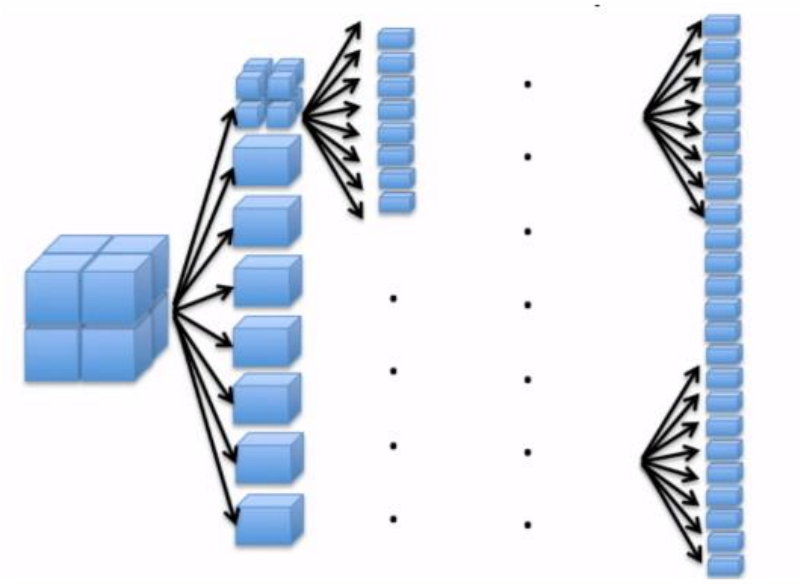
```
# LSH για 'Education'
education_texts = result_df['Education'].tolist()
education_texts_preprocessed = [preprocess_education(edu) for edu in education_texts]
start_time_2 = time.time()
similar_pairs_education = lsh_education(education_texts_preprocessed, result_df,
threshold_education)

# Αποθηκεύουμε τα μοναδικά ονόματα από όλα τα ζευγάρια με κοινή εκπαίδευση
unique_surnames_in_pairs = set()

for pair in similar_pairs_education:
  (entry1, surname1), (entry2, surname2) = pair
  if surname1 in unique_surnames and surname1 not in unique_surnames_in_pairs:
    awards1, dblp1 = df.loc[df['Surname'] == surname1, ['Awards', 'Dblp']].iloc[0]
    print(f"Surname: {surname1}, #Awards: {awards1}, #DBLP_Record: {dblp1}")
    unique_surnames_in_pairs.add(surname1)
  if surname2 in unique_surnames and surname2 not in unique_surnames_in_pairs:
    awards2, dblp2 = df.loc[df['Surname'] == surname2, ['Awards', 'Dblp']].iloc[0]
    print(f"Surname: {surname2}, #Awards: {awards2}, #DBLP_Record: {dblp2}")
    unique_surnames_in_pairs.add(surname2)

    # μετατροπή αποτελεσμάτων results σε DataFrame
result_df_education = pd.DataFrame(similar_pairs_education, columns=['Entry1', 'Entry2'])
result_df_education.to_csv('similar_pairs_education.csv', index=False)
```

Για να **χρονομετρήσουμε** τον υπολογισμό που χρειάζεται το δέντρο για range search και lsh search θα χρησιμοποιήσουμε τις εντολές **start_time** = time.time() και **end_time** = time.time() τόσο πριν και μετά το κάλεσμα της συνάρτησης **range_search** όσο και πριν και μετά αντίστοιχα το πέρας της εντολής **range_search**(). Θα υπολογίσουμε τον **elapsed time** των δύο χρονομετρήσεων και θα εκτυπώσουμε τον τελικό χρόνο που χρειάστηκε το δέντρο για να επιστρέψει τα τελικά αποτελέσματα. Με αυτόν τον τρόπο παίρνουμε τον ακριβές χρόνο χωρίς να παρεμβάλλονται εντολές (πχ print()) που δεν είναι αντιπροσωπευτικοί της διαδικασίας αναζήτησης.

```
# Υπολογισμός elapsed time
elapsed_time_1 = end_time_1 - start_time_1
elapsed_time_2 = end_time_2 - start_time_2

total_elapsed_time = elapsed_time_1 + elapsed_time_2
# εκτύπωση elapsed time
print(f"Time taken for lsh: {elapsed_time} seconds")
```

3 Quad-tree

Τα quadtrees είναι ιεραρχικές δομές χωρικών δέντρων που βασίζονται στην αρχή της αναδρομικής αποσύνθεσης του χώρου. Το quadtree χρησιμοποιείται για την αναπαράσταση δισδιάστατων δεδομένων αναπαρίσταται ως τετράγωνο και κάθε κόμβος χωρίζεται σε 4 υποδιαιρέσεις και έχει 4 υποκομβους. Στον τρισδιάστατο χώρο το quadtree γίνεται **octree** καθώς τώρα έχουμε 3 διαστάσεις και αναπαρίσταται ως κύβος και κάθε κύβος έχει 8 υπο-κύβους. Άρα κάθε κόμβος αντιστοιχίζεται σε ένα κύβο και έχει 8 παιδιά(υπο-κόμβους).



3.1 Κατασκευή

Αρχικά δημιουργούμε την συνάρτηση **letter_normalization()**, η οποία θα χρησιμοποιηθεί για την μετατροπή του πρώτου γράμματος κάθε επιθέτου σε αριθμό, ώστε να δημιουργηθεί η συντεταγμένη x.

```
#συνάρτηση για την μετατροπή των γραμμάτων σε αριθμό
def letter_normalization(letter):
  return max(ord(letter.upper()) - 65, 0)
```

Για την υλοποίηση του συγκεκριμένου δέντρου, αρχικά δημιουργούμε την κλάση **Point**, η οποία αναπαριστά ένα σημείο με 3 διαστάσεις (x,y,z)

X: προκύπτει από το επίθετο των επιστημόνων, μετατρέποντας το αρχικό γράμμα σε αριθμό με βάση την θέση του στο αγγλικό αλφάβητο.

Y: είναι το πλήθος των βραβείων.

Z: είναι το πλήθος των δημοσιεύσεων.

Επίσης περιέχει και την μεταβλητή **data** που θα είναι οι υπόλοιπες λεπτομέρειες που συνδέονται με αυτά τα στοιχεία, δηλαδή η εκπαίδευση.

```
# κλάση που αναπαριστά ένα 3D σημείο
class Point:
  def __init__(self, x, y, z, data=None):
```

```
    self.x = x
    self.y = y
    self.z = z
    self.data = data
```

Στην συνέχεια, υλοποιούμε την κλάση **OctreeNode**, η οποία αναπαριστά έναν κόμβο μέσα στο δέντρο. Περιέχει τις συναρτήσεις **__init__(), insert(), devide(), boundaries()**.

Μέσα στην κλάση ορίζουμε τη συνάρτηση **__init__()**, η οποία περιέχει τις συντεταγμένες του κέντρου (x,y,z) και τις διαστάσεις (w,h,d) του κόμβου και κάθε κόμβος θα έχει 8 υπο-κόμβους (subnodes) που αρχικοποιούνται με None. Επίσης, υπολογίζουμε και τα όρια του συγκεκριμένου κόμβου (left, right, top,bottom, back. front) με βάσης τις συντεταγμένες και τις διαστάσεις.

```
# κλάση που αναπαριστά έναν κόμβο μέσα στο Octree δέντρο
class OctreeNode:
  def __init__(self, x, y, z, w, h, d):
    self.x = x  #συντεταγμένη x
    self.y = y  #συντεταγμένη y
    self.z = z  #συντεταγμένη z
    self.w = w  #πλάτος
    self.h = h  #ύψος
    self.d = d  #βάθος
    self.points = []  # λίστα να αποθηκεύει όλα τα σημεία του κόμβου
    self.subnodes = [None] * 8  # λίστα με τους υπο-κόμβους

    # οριοθέτηση των ορίων του κύβου με βάση τις συντεταγμένες   (x, y, z) και τις διαστάσεις (w, h, d)
    self.left = x - w / 2
    self.right = x + w / 2
    self.bottom = y - h / 2
    self.top = y + h / 2
    self.back = z - d / 2
    self.front = z + d / 2
```

Η συνάρτηση **insert()**, αρχικά, ελέγχει αν το σημείο είναι μέσα στα όρια του τρέχοντα κόμβου, αν δεν είναι τότε δεν εισάγεται. Ύστερα, ελέγχει αν υπάρχει χώρος στον τρέχοντα κόμβο, αν δηλαδή τα σημεία που έχουν εισαχθεί είναι λιγότερα από 8, αν υπάρχει χώρος εισάγεται στον κόμβο. Αλλιώς, αν ο κόμβος δεν έχει διασπαστεί σε υποκόμβους διασπάται με την συνάρτηση **divide()** και το σημείο εισάγεται στον κατάλληλο υποκόμβο καλώντας τον εαυτό της.

```
def insert(self, point):

def insert(self, point):
  #έλεγχος αν το σημείο είναι μέσα στα όρια του κόμβου
  if not self.boundary(point):
    return

  # αν υπάρχει χώρος στον τρέχοντα κόμβο (points<8), προσθέτουμε το σημείο
  if len(self.points) < 8:
    self.points.append(point)

  else:

    # άμα ο κόμβος είναι γεμάτος και δεν έχει διασπαστεί τον διασπάμε
```

```
    if self.subnodes[0] is None:
        self.divide()

    #εισαγωγή του σημείου στον κατάλληλο υπο-κόμβο
    for subnode in self.subnodes:

        subnode.insert(point)
```

Όσον αφορά την συνάρτηση **devide(),** διαιρεί τον κόμβο σε 8 υπο-κόμβους. Κάθε υπο-κόμβος θα έχει διαστάσεις ίσες με το μισό του αρχικού κόμβου. Οι διαστάσεις και οι συντεταγμένες υπολογίζονται με βάση τις συντεταγμένες και τις διαστάσεις του αρχικού κόμβου και κάθε υπο-κόμβος ορίζεται ως καινούριος κόμβος με τις απαραίτητες συντεταγμένες και διαστάσεις.

```
# συνάρτηση που χρησιμοποιείται για την διαίρεση ενός κόμβου σε 8 υπο-κομβους
def divide(self):
    x, y, z = self.x, self.y, self.z

    w, h, d = self.w / 2, self.h / 2, self.d / 2  #Κάθε υπο-κόμβος θα έχει διαστάσεις ίσες με το μισό του αρχικού κόμβου


    # δημιουργία των 8 υπο-κομβων με τις κατάλληλες διαστάσεις

    self.subnodes[0] = OctreeNode(x - w / 2, y - h / 2, z - d / 2, w, h, d)

    self.subnodes[1] = OctreeNode(x + w / 2, y - h / 2, z - d / 2, w, h, d)

    self.subnodes[2] = OctreeNode(x + w / 2, y + h / 2, z - d / 2, w, h, d)

    self.subnodes[3] = OctreeNode(x - w / 2, y + h / 2, z - d / 2, w, h, d)

    self.subnodes[4] = OctreeNode(x - w / 2, y - h / 2, z + d / 2, w, h, d)

    self.subnodes[5] = OctreeNode(x + w / 2, y - h / 2, z + d / 2, w, h, d)

    self.subnodes[6] = OctreeNode(x + w / 2, y + h / 2, z + d / 2, w, h, d)

    self.subnodes[7] = OctreeNode(x - w / 2, y + h / 2, z + d / 2, w, h, d)
```

Η συνάρτηση **boundaries()** ελέγχει αν οι συντεταγμένες του σημείου είναι στα όρια του κύβου.

```
# συνάρτηση για να ελέγχει αν το σημείο είναι μέσα στα όρια του κόμβου
def boundary(self, point):

    return (

        self.left <= point.x < self.right and

        self.bottom <= point.y < self.top and

        self.back <= point.z < self.front
    )
```

Όσον αφορά την συνάρτηση **read_data()** είναι υπεύθυνη για την ανάγνωση των δεδομένων από το csv. Αρχικά, μετατρέπουμε το csv σε dataframe και με ένα βρόγχο επανάληψης for θα διατρέξουμε όλο το dataframe. Η διάσταση x θα προκύπτει από την στήλη Surname του csv, μετατρέποντας το αρχικό γράμμα σε αριθμό με βάση την θέση του στο αγγλικό αλφάβητο χρησιμοποιώντας την συνάρτηση **letter_normilazation()**,η διάσταση y προκύπτει από την στήλη Awards του csv και η διάσταση z προκύπτει από την στήλη Dblp. Στην συνέχεια δημιουργούμε ένα λεξικό **data** που θα περιέχει όλα τα στοιχεία του csv. Τέλος, δημιουργείται ένα αντικείμενο **point** που θα έχει συντεταγμένες (x,y,z) και θα συσχετίζονται με το λεξικό data. Όλα τα point που δημιουργούνται προστίθενται στην λίστα **points**. Η συνάρτηση επιστρέφει την λίστα με όλα τα point.

```
# συνάρτηση για την ανάγνωση των δεδομένων απο το csv
def read_data():
  df = pd.read_csv("scientists.csv")
  points = []

  for i in range(len(df)):

    x = letter_normalization(df.iloc[i]['Surname'][0])

    y = df.iloc[i]['Awards']

    z = df.iloc[i]['Dblp']

    data = {
      'Surname': df.iloc[i]['Surname'],
      'Awards': df.iloc[i]['Awards'],
      'Dblp': df.iloc[i]['Dblp'],
      'Education': df.iloc[i]['Education']
    }

    point = Point(x, y, z, data)

    points.append(point)

  return points
```

Η συνάρτηση **build_octree()** θα χρησιμοποιηθεί για να δημιουργήσει το octree. Ξεκινάμε αποθηκεύοντας στην μεταβλητή **points** την λίστα που επιστρέφεται καλώντας την συνάρτηση **read_data()**. Ύστερα, βρίσκουμε την μέγιστη και την ελάχιστη τιμή για κάθε διάσταση x.y.z ανάμεσα σε όλα τα στοιχεία που περιέχει η λίστα. Επιπλέον, υπολογίζουμε το πλάτος, το ύψος και το βάθος του δέντρου με βάση την μέγιστη και την ελάχιστη τιμή κάθε διάστασης. Στην συνέχεια, φτιάχνουμε την ρίζα του δέντρου καλώντας την **OctreeNode** και τοποθετείται στο κέντρο με βάση τους υπολογισμούς που έχουμε πραγματοποιήσει παραπάνω. Τέλος, εισάγουμε τα σημεία στο δέντρο προσαρμόζοντας τις συντεταγμένες ώστε τα σημεία που βρίσκονται ακριβώς πάνω στα όρια δέντρου να τοποθετηθούν μέσα σε αυτό. Η τιμή 0,001 επιλέχθηκε ώστε η διαφορά στις κανονικοποιημένες τιμές να είναι αμελητέα και να μην επηρεάζει την ακρίβεια του δέντρου.

```
#συνάρτηση για την κατασκευή του δέντρου
def build_octree():

  points = read_data()
```

```
# εύρεση της μέγιστης και της ελάχιστης τιμής σε κάθε διάσταση
min_values = {'x': min(point.x for point in points),

        'y': min(point.y for point in points),

        'z': min(point.z for point in points)}

max_values = {'x': max(point.x for point in points),

        'y': max(point.y for point in points),

        'z': max(point.z for point in points)}

# εύρεση πλάτους, ύψους, βάθους
octree_width = max(0, max_values['x'] - min_values['x'])
octree_height = max(0, max_values['y'] - min_values['y'])
octree_depth = max(0, max_values['z'] - min_values['z'])

# δημιουργία της ρίζας του Octree
octree = OctreeNode(
   min_values['x'] + octree_width / 2,

   min_values['y'] + octree_height / 2,

   min_values['z'] + octree_depth / 2,

   octree_width,

   octree_height,

   octree_depth
)

# εισαγωγή των σημείων μέσα στο δέντρο
for point in points:

   point.x = max(min(point.x, octree.right - 0.001), octree.left + 0.001)

   point.y = max(min(point.y, octree.top - 0.001), octree.bottom + 0.001)

   point.z = max(min(point.z, octree.front - 0.001), octree.back + 0.001)

   octree.insert(point)

return octree
```

3.2 Αναζήτηση

Η συνάρτηση η οποία χρησιμοποιείται για την αναζήτηση μέσα στο δέντρο είναι η **query_octree()**, γι' αυτό και δέχεται σαν ορίσματα το δέντρο, το μέγιστο και το ελάχιστο γράμμα που επιθυμεί ο χρήστης, τον ελάχιστο αριθμό βραβείων και το μέγιστο και το ελάχιστο αριθμό δημοσιεύσεων. Δημιουργούμε την λίστα **results** που θα αποθηκεύει τα αποτελέσματα που ανταποκρίνονται στα όρια αναζήτησης. Διατρέχουμε κάθε **Point** που είναι αποθηκευμένο στον τρέχον κόμβο του δέντρου και ελέγχουμε αν όλες οι συντεταγμένες ικανοποιούν τα κριτήρια αναζήτησης (το x πρέπει να είναι ανάμεσα ή ίσο στο μέγιστο και στο ελάχιστο γράμμα εισαγωγής, το y πρέπει να είναι μεγαλύτερο ή ίσο με το αριθμό βραβείων που εισάγεται και το

z πρέπει να είναι ανάμεσα ή ίσο με το μέγιστο και το ελάχιστο αριθμό δημοσιεύσεων που εισάγεται σαν όρισμα). Αν οι συνθήκες ικανοποιούνται τότε το συγκεκριμένο στοιχείο (point) εισάγεται στην λίστα, αλλιώς ξεκινά και ψάχνει στους υποκόμβους και καλεί τον εαυτό της ώστε να συνεχίσει η αναζήτηση. Στο τέλος επιστρέφεται η λίστα **results** με όλα τα αποτελέσματα που ικανοποιούν τα κριτήρια.

```
# συνάρτηση για αναζήτηση στο δέντρο σημείων που ικανοποιούν κάποια όρια
def query_octree(node, min_letter, max_letter, min_awards, min_dblp, max_dbpl):

  results = []
  if node is not None:
    for point in node.points:
      # έλεγχος αν το σημείο είναι μέσα στα όρια
      if (
        min_letter <= point.x <= max_letter and
        min_awards <= point.y and
        min_dblp <= point.z <= max_dbpl
      ):
        results.append(point.data)

    # αναζήτηση στους υποκόμβους
    for subnode in node.subnodes:
      results.extend(query_octree(subnode, min_letter, max_letter, min_awards, min_dblp,
max_dbpl))
  return results
```

Στη συνάρτηση **main_Oc_Tree()** αρχικά δημιουργούμε το δέντρο καλώντας την συνάρτηση **build_octree()** και το αποθηκεύουμε στην μεταβλητή **octree**. Ύστερα ζητάμε από τον χρήστη να εισάγει τα όρια που επιθυμεί ώστε να πραγματοποιηθεί η αναζήτηση καθώς και την ομοιότητα που θέλει να έχουν οι τελικοί επιστήμονες στο πεδίο education. Τα γράμματα που εισάγει ο χρήστης θα μετατραπούν σε αριθμούς χρησιμοποιώντας την συνάρτηση **letter_normilazation()**.

```
def main_Oc_Tree():

  # κατασκευή του Octree
  octree = build_octree()

  #εισαγωγή από τον χρήστη των σημείων αναζήτησης
  min_letter = input("Press the first letter of the query (ex.A, B, C...):").upper()
  while not min_letter.isalpha():
    min_letter = input("Press the first letter you want in the right form:")
  max_letter = input("Press the second letter of the query (ex.A, B, C...):").upper()
  while not max_letter.isalpha():
    max_letter = input("Press the second letter you want in the right form:")
  min_awards = int(input("Press the minimum awards of the query (ex.0,1,2...) :"))
  min_dblps = int(input("Press the minimum DBLP records of the query (ex.0,1,2...) :"))
  max_dblps = int(input("Press the maximum DBLP records of the query (ex.0,1,2...) :"))
  lsh_sim = float(input("Give the LSH percentage of similarity between the scientists (ex.0.5->50%):"))
  print("The query that you created is the following ! ")
  print(
    "Find the computer scientists from Wikipedia that their letters belongs in the interval [" + min_letter
+ "-" + max_letter + "],have won more than " + str(
      min_awards) + " prizes and their DBLP record belongs in the interval [" + str(
      min_dblps) + " ," + str(max_dblps) + "] !")
  print("Generating Answer")
```

```
#μετατροπή των γραμμάτων σε αριθμούς
min_letter = letter_normalization(min_letter)
max_letter = letter_normalization(max_letter)
```

Η αναζήτηση στο δέντρο πραγματοποιείται καλώντας την συνάρτηση **query_octree()** η οποία παίρνει σαν όρισμα το κατασκευασμένο δέντρο (octree) και τα όρια που έχουν δοθεί από τον χρήστη. Τα αποτελέσματα αποθηκεύονται στην μεταβλητή **query_results**. Επειδή θέλουμε να χρονομετρήσουμε την διάρκεια της αναζήτησης πριν την καλέσουμε την συνάρτηση query_octree() ξεκινάμε τον χρόνο **start_time_1 = time.time()** και μετά την εκτέλεση της σταματάμε τον χρόνο **end_time_1 = time.time()** και υπολογίζουμε το **elapsed_time_1.** Στην συνέχεια εκτυπώνονται τα αποτελέσματα.

```
# έναρξη χρόνου
start_time_1 = time.time()

# αναζήτηση
query_results = query_octree(octree, min_letter, max_letter, min_awards, min_dblps, max_dblps)

# λήξη χρόνου
end_time_1 = time.time()
#υπολογισμός συνολικού χρόνου για αναζήτηση
elapsed_time_1 = end_time_1 - start_time_1

# εκτυπωση αποτελεσμάτων αναζήτησης
for result in query_results:
  print(f"{result['Surname']},{result['Awards']},{result['Dblp']}")
```

3.3 Αναζήτηση με LSH

Για να πραγματοποιηθεί η αναζήτηση με lsh αρχικά φιλτράρουμε τα παραπάνω αποτελέσματα και στην μεταβλητή **results** κρατάμε μόνο τους επιστήμονες που έχουν education. Επίσης, αποθηκεύουμε στο dataframe **result_df** τα αποτελέσματα results. Στην συνέχεια εξάγουμε την στήλη education από τα αποτελέσματα, την αποθηκεύουμε στην μεταβλητή **education_data** και φιλτράρουμε τα κείμενα χρησιμοποιώντας την συνάρτηση **preprocess_education().** Επιπλέον, επειδή θέλουμε να μετρήσουμε τον χρόνο εκτέλεσης της αναζήτησης με lsh ξεκινάμε πάλι τον χρόνο **start_time_2 = time.time()** ύστερα καλούμε την συνάρτηση **lsh_education()**, η οποία δέχεται σαν όρισμα την επεξεργασμένη στήλη education(**education_data_preprocessed**), το **result_df** και το similarity (**lsh_sim**) που έχει εισάγει ο χρήστης, μετά τερματίζουμε τον χρόνο **end_time_2 = time.time()** και υπολογίζουμε το elapse_time_2. Τα αποτελέσματα εκτυπώνονται, αν δεν υπάρχουν επιστήμονες με την συγκεκριμένη ομοιότητα εκτυπώνεται κατάλληλο μήνυμα.

```
#φιλτραρουμε τα αποτελέσματα ώστε στο result να υπάρχουν μόνο αυτοί που έχουν education
results = [result for result in query_results if "Education information not found" not in
result['Education']]

# αποθήκευση των αποτελεσμάτων σε dataframe
result_df = pd.DataFrame(results)

# εξαγωγή 'Education' από τα παραπάνω αποτελέσματα
education_data = [result['Education'] for result in results]
education_data_preprocessed = [preprocess_education(edu) for edu in education_data]

# έναρξη χρόνου
```

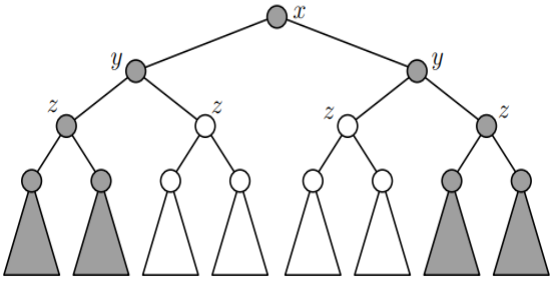
```
start_time_2 = time.time()
# εκτέλεση lsh ομοιότητας
similar_pairs_education = lsh_education(education_data_preprocessed, result_df, lsh_sim)

# λήξη χρόνου
end_time_2 = time.time()

# υπολογισμός συνολικού χρόνου για lsh
elapsed_time_2 = end_time_2 - start_time_2

# Αποθηκεύουμε τα μοναδικά ονόματα από όλα τα ζευγάρια με κοινή εκπαίδευση
unique_surnames_in_pairs = set()

#εκτύπωση αποτελεσμάτων ομοιότητας
if len(similar_pairs_education) == 0:
   print("\nThere are no scientists with similarity " + str(int(lsh_sim * 100)) + "%.")
else:
   print("\nThe scientists with similarity " + str(int(lsh_sim * 100)) + "% are the following:\n")
   for pair in similar_pairs_education:
      (entry1, surname1), (entry2, surname2) = pair
      if surname1 in [result['Surname'] for result in results] and surname1 not in
unique_surnames_in_pairs:
         awards1, dblp1 = result_df.loc[result_df['Surname'] == surname1, ['Awards', 'Dblp']].iloc[0]
         print(f"Surname: {surname1}, #Awards: {awards1}, #DBLP_Record: {dblp1}")
         unique_surnames_in_pairs.add(surname1)
      if surname2 in [result['Surname'] for result in results] and surname2 not in
unique_surnames_in_pairs:
         awards2, dblp2 = result_df.loc[result_df['Surname'] == surname2, ['Awards', 'Dblp']].iloc[0]
         print(f"Surname: {surname2}, #Awards: {awards2}, #DBLP_Record: {dblp2}")
         unique_surnames_in_pairs.add(surname2)

 # αποθήκευση των αποτελεσμάτων σε dataframe
result_df_education = pd.DataFrame(similar_pairs_education, columns=['Entry1', 'Entry2'])
result_df_education.to_csv('similar_pairs_education.csv', index=False)
```

Τέλος υπολογίζεται και εκτυπώνεται ο συνολικός χρόνος αναζήτησης και lsh προσθέτοντας τους χρόνους.

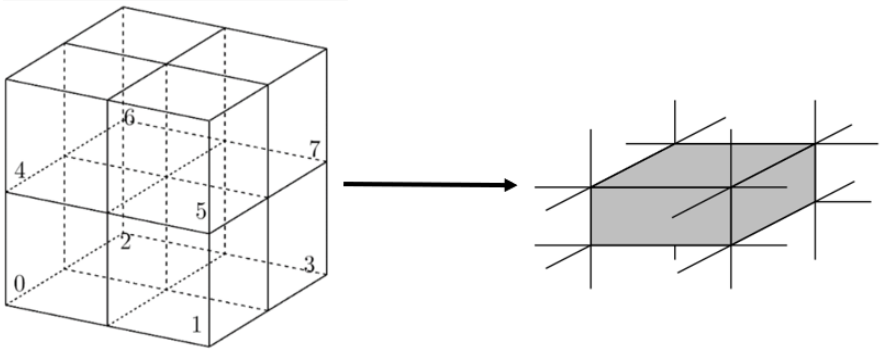
```
#συνολικός χρόνος για αναζητηση και lsh
elapsed_time = elapsed_time_1 + elapsed_time_2

# εκτύπωση του χρόνου
print(f"Time taken for lsh {elapsed_time} seconds")
```

4 kd-tree

Το **kd – tree** αποτελεί μια κατηγορία Δυαδικού Δένδρου Αναζήτησης όπου κάθε κόμβος είναι ένα πολυδιάστατο σημείο με k διαστάσεις. Στη δικιά μας περίπτωση, εφόσον μιλάμε για τρισδιάστατα σημεία στο χώρο, τότε κάθε κόμβος θα συμβολίζεται ως **P = (x, y, z)**. Κάθε κόμβος (εκτός από τα φύλλα) αντιπροσωπεύει στο χώρο R^3 ένα υπερεπίπεδο και στόχος μας είναι να διασπάσουμε τον χώρο αυτό σε μικρότερους υποχώρους, ώστε να γίνει η αναζήτηση πιο εύκολα.



Γενικά, ως υπερεπίπεδο σε έναν 3 – διάστατο χώρο θεωρούμε μια επίπεδη επιφάνεια η οποία θα χωρίζει το χώρο R^3 σε δύο υποχώρους. Ο ένας υποχώρος του R^3 θα αντιπροσωπεύει το σύνολο των σημείων P που είναι συγκεντρωμένα στο αριστερό υποδένδρο (left – subtree) και ο άλλος τα σημεία P που βρίσκονται στο δεξί υποδένδρο (right – subtree). Καθώς διατρέχουμε το δένδρο μετακινούμαστε ανάμεσα στους άξονες του R^3 χώρου εναλλάσσοντας κάθε φορά τη μια διάσταση από τις τρεις, ορίζοντας έτσι ένα υπερεπίπεδο το οποίο θα διαχωρίζει το χώρο κατά μήκος της συγκεκριμένης διάστασης. Αυτή η διαδικασία επαναλαμβάνεται αναδρομικά σε κάθε νέο υποχώρο που δημιουργείται επιλέγοντας τη διάσταση η οποία διαθέτει μεγαλύτερο εύρος τιμών. Έτσι θα έχουμε καταφέρει στο τέλος να μοιράσουμε τα σημεία του δένδρου σε κατάλληλους υποχώρους προκειμένου να προχωρήσουμε στη συνέχεια στην αναζήτηση. Συνεπώς, η αναζήτηση στο τρισδιάστατο δένδρο θα πραγμτοποιείται κάθε φορά εντός ενός κουτιού. Η μέθοδος που ακολουθούμε κατά τη διάσπαση σε υποχώρους βασίζεται στην **εύρεση του μεσαίου σημείου** σε κάθε επανάληψη από όλα τα σημεία του συνόλου (split by median), προκειμένου το δένδρο που θα δημιουργηθεί να είναι ένα Balanced Binary Tree. Το μεσαίο σημείο επιλέγουμε εμείς αυθαίρετα σε ποιον από τους δύο υποχώρους θα τοποθετηθεί.



4.1 Κατασκευή

Το μεσαίο σημείο από ένα σύνολο n σημείων μπορεί να υπολογιστεί σε O(n) χρόνο. Έστω επίσης T(n) ο χρόνος που απαιτείται για τη κατασκευή του δένδρου με n κόμβους. Τότε, θα είναι: T(1) = O(1), T(2) = 2T(1) + O(2), …, T(n) = 2T(n/2) + O(n). Άρα, η κατασκευή του αποδεικνύεται ότι θα απαιτεί συνολικά χρόνο της τάξης **O(nlogn).**

Αρχικά, ορίζουμε τη κλάση **Node**, η οποία θα αντιπροσωπεύει κάθε κόμβο στο kd-tree. Κάθε κόμβος έχει ένα τρισδιάστατο σημείο (**point**) που θα αντιστοιχεί σε ένα σημείο του χώρου R^3 και δύο παιδιά (**left** και **right**), τα οποία θα αντιστοιχούν στο αριστερό και δεξί υποδένδρο αντίστοιχα του κόμβου.

```
class Node:
    def __init__(self, point, left=None, right=None):
        self.point = point
        self.left = left
        self.right = right
```

Έπειτα, ορίζουμε τη συνάρτηση **build_kdtree( )**, η οποία θα κατασκευάσει με είσοδο μια λίστα σημείων (**points**) το ζητούμενο δένδρο με βάση τη πιο πάνω περιγραφή. Σαν πρώτο βήμα ελέγχουμε αν η λίστα **points** είναι κενή και αν ναι τότε επιστρέφουμε *None*, δηλαδή ένα κενό υπόδενδρο. Στη συνέχεια υπολογίζουμε τον άξονα (**axis**) με βάση τον οποίο θα γίνει ο διαχωρισμός των σημείων στον χώρο R^3. Πιο συγκεκριμένα, ο άξονας θα εναλλάσσεται μεταξύ των τριών διαστάσεων (x, y, z) καθώς το βάθος αυξάνεται, χρησιμοποιώντας το ακέραιο υπόλοιπο της διαίρεσης με το 3, έτσι ώστε να πραγματοποιηθεί η διαδοχική διαίρεση του χώρου σε υποχώρους κατά μήκος των τριών διαστάσεων. Τα σημεία της λίστας θα ταξινομούνται σύμφωνα με τις τιμές τους στον άξονα **axis** και αφού πραγματοποιηθεί η ταξινόμηση θα υπολογίζεται το μεσαίο σημείο της λίστας **points** (**median**) και θα δημιουργείται ο επόμενος κόμβος με βάση αυτό το σημείο. Τέλος, καλούμε αναδρομικά τη συνάρτηση για τα αριστερά και δεξιά υποσύνολα των σημείων από το **median**, αυξάνοντας το βάθος κατά 1 σε κάθε κλήση.

```
def build_kdtree(points, depth=0):
    if not points:
        return None

    k = len(points[0])
    axis = depth % k

    points.sort(key=lambda x: x[axis])

    median = len(points) // 2

    return Node(
        point=points[median],
        left=build_kdtree(points[:median], depth + 1),
        right=build_kdtree(points[median + 1:], depth + 1)
    )
```

4.2 Αναζήτηση

Όσον αφορά την αναζήτηση, ακολουθούμε τον εξής αλγόριθμο:

Βήμα 1: Μεταβαίνουμε στη ρίζα του δένδρου.

Βήμα 2: Επιλέγουμε τη διάσταση (x, y ή z) που αντιστοιχεί στο βάθος του κόμβου του τρέχοντος κλάδου.

Βήμα 3: Εξετάζουμε το σημείο αναζήτησης και το σημείο του κόμβου στην επιλεγμένη διάσταση για να αποφασίσουμε ποιο από τα παιδιά του κόμβου (αριστερό ή δεξί υποδένδρο) θα εξεταστεί επόμενο.

Βήμα 4: Αν το παιδί που επιλέχθηκε έχει δεδομένα, επαναλαμβάνουμε την αναζήτηση σε αυτό το παιδί, αυξάνοντας το βάθος κατά ένα.

Βήμα 5: Κάθε φορά που φτάνουμε σε ένα φύλλο ελέγχουμε αν το σημείο του φύλλου βρίσκεται εντός του εύρους αναζήτησης. Εάν ναι, τότε προσθέτουμε το σημείο στα αποτελέσματα της αναζήτησης.

Αυτά τα βήματα επαναλαμβάνονται αναδρομικά για κάθε κλάδο του δέντρου μέχρι να εξερευνηθούν όλα τα κατάλληλα κλαδιά και να βρεθούν όλα τα σημεία εντός του εύρους αναζήτησης.

Ορίζουμε πρώτα από όλα στη συνάρτηση **range_search( )**, η οποία θα λαμβάνει ως ορίσματα τους κόμβους (**node**) του δένδρου και το εύρος του ερωτήματος αναζήτησης (**query_point**) και θα επιστρέφει όλους τους κόμβους οι οποίοι θα έχουν συντεταγμένες εντός του δοθέντος εύρους. Αρχικά, θα πρέπει να αποφασίσουμε ποιο υποδένδρο πρέπει να αναζητηθεί σύμφωνα με το εύρος αναζήτησης και τη θέση του τρέχοντος κόμβου στον άξονα που εξετάζεται. Έτσι, ορίζουμε δύο αντίστοιχες μεταβλητές. Από τη μία, η μεταβλητή **search_left** θα ελέγχει αν πρέπει να αναζητηθεί το αριστερό υποδένδρο του τρέχοντος κόμβου. Αυτό θα πραγματοποιείται όταν η ελάχιστη τιμή του εύρους αναζήτησης στην τρέχουσα διάσταση (**query_point[axis][0]**) είναι μικρότερη ή ίση με την αντίστοιχη συνιστώσα του σημείου του κόμβου (**node.point[axis]**). Τότε η μεταβλητή **search_left** θα λάβει τιμή **True**, προτρέποντας έτσι τη συνέχιση της αναζήτησης στο αριστερό υποδένδρο. Αντίστοιχα θα λειτουργεί και η μεταβλητή **search_right**, η οποία θα ελέγχει αν πρέπει να αναζητηθεί το δεξί υποδένδρο του τρέχοντος κόμβου. Αν η ανώτατη τιμή του εύρους αναζήτησης στην τρέχουσα διάσταση (**query_point[axis][1]**) είναι μεγαλύτερη ή ίση με την αντίστοιχη συνιστώσα του σημείου του κόμβου (**node.point[axis]**), τότε η μεταβλητή **search_right** θα λάβει πάλι τιμή **True**, προτρέποντας έτσι τη συνέχιση της αναζήτησης στο δεξί υποδένδρο. Αφού επιλεγεί το αντίστοιχο υποδένδρο που πρέπει να ελεγχθεί, πρέπει να επιβεβαιώσουμε αν ο τρέχων κόμβος είναι εντός των ζητούμενων ορίων. Με τη μεταβλητή **in_range** για κάθε διάσταση ελέγχεται αν η συνιστώσα του τρέχοντος κόμβου **node.point[i]** βρίσκεται μεταξύ των αντίστοιχων κάτω και άνω ορίων (**lower_bound** και **upper_bound**) του εύρους αναζήτησης στη συγκεκριμένη διάσταση. Για τη πρώτη διάσταση, εφόσον έχουμε να κάνουμε με αλφαριθμητικούς χαρακτήρες (**'A'** – **'Z'**), προσαρμόζουμε το εύρος αναζήτησης προσθέτοντας στο εύρος και τον επόμενο χαρακτήρα (μέσω της **char(255)**) που ακολουθεί αλφαβητικά μετά από αυτόν στο **upper_bound**, προκειμένου να γίνει σωστά η σύγκριση των αλφαριθμητικών και να μην χαθούν τα επώνυμα που ξεκινούν με το τελευταίο χαρακτήρα του εύρους. Για τις υπόλοιπες δύο διαστάσεις, εφόσον μιλάμε για φυσικούς αριθμούς δεν χρειάζεται κάποια προσαρμογή στη σύγκριση. Αν η συνιστώσα βρίσκεται εντός του εύρους και για τις τρεις διαστάσεις, τότε η μεταβλητή **in_range** παραμένει με περιεχόμενο **True** και έτσι ο τρέχων κόμβος προστίθεται στα τελικά αποτελέσματα **results**, διαφορετικά τίθεται σε **False**.

```
def range_search(node, query_point, depth=0):
    if node is None:
        return []

    k = len(query_point)
    axis = depth % k
    result = []

    # Καθορίζουμε αν χρειάζεται να ψάξουμε στο αριστερό ή δεξί υποδέντρο
    search_left = query_point[axis][0] <= node.point[axis] # Γίνεται True όταν
    search_right = node.point[axis] <= query_point[axis][1]

    #  Αναδρομική αναζήτηση στο αριστερό υποδέντρο αν είναι απαραίτητο
    if search_left and node.left is not None:
        result.extend(range_search(node.left, query_point, depth + 1))

    # Ελέγχουμε αν ο τρέχων κόμβος είναι εντός του εύρους
    in_range = True
    for i in range(k):
        lower_bound = query_point[i][0] # Κάτω όριο
        upper_bound = query_point[i][1] # Άνω όριο
```

```
        # Προσαρμογή του ανωτάτου ορίου για συγκρίσεις ανάμεσα σε συμβολοσειρές
        if isinstance(node.point[i], str) and isinstance(upper_bound, str):
            in_range = in_range and lower_bound <= node.point[i] <= upper_bound + chr(255)
        else:
            in_range = in_range and lower_bound <= node.point[i] <= upper_bound

    if in_range:
        result.append(node.point)

    # Αναδρομική αναζήτηση στο δεξί υποδέντρο αν είναι απαραίτητο
    if search_right and node.right is not None:
        result.extend(range_search(node.right, query_point, depth + 1))

    return result
```

4.3 Αναζήτηση με LSH

Η συνάρτηση **result( )** λαμβάνει σαν όρισμα το δένδρο που κατασκευάσαμε, τα διαστήματα εύρους για κάθε διάσταση και το threshold όσον αφορά την ομοιότητα εκπαίδευσης σε όσους κόμβους πληρούν τις προηγούμενες προϋποθέσεις. Αρχικά, ορίζουμε το σημείο που περιέχει όλα τα εύρη κάθε διάστασης (**query_point**) καθώς και την μεταβλητή που δηλώνει τη χρονική στιγμή έναρξης της χρονομέτρησης της αναζήτησης στο kd – tree (**start_time_1**). Έπειτα, καλούμε τη συνάρτηση **range_search( )** που κατασκευάσαμε πιο πάνω και αποθηκεύουμε στη λίστα **points_in_range** όλους τους κόμβους που ταιριάζουν στα εύρη του **query_point**. Παράλληλα, λήγει και η χρονομέτρηση, με τη τελική χρονική στιγμή να αποθηκεύεται στο **end_time_1** και το συνολικό χρόνο εκτέλεσης της αναζήτησης να δίνεται ως η διαφορά ανάμεσα σε **end_time_1** και **start_time_1**. Ακόμα, δημιουργούμε ένα σύνολο **unique_surnames** για να παρακολουθούμε τα μοναδικά επώνυμα με έγκυρη εκπαίδευση. Για κάθε κόμβο από αυτούς (**point**) εξάγουμε τις τρεις συντεταγμένες του (**surname**, **awards**, **dblp**) και ελέγχουμε αν έχει ήδη προστεθεί μια εγγραφή με έγκυρη εκπαίδευση για αυτό το επώνυμο. Αν έχει προστεθεί, η επόμενη εγγραφή παραλείπεται. Αυτό το βήμα είναι απαραίτητο για να εξαλείψουμε τις διπλότυπες εγγραφές σε περίπτωση που έχουμε δύο επιστήμονες με ίδιες συντεταγμένες αλλά ο ένας από αυτούς δεν έχει έγκυρη εκπαίδευση, οπότε δεν πρέπει να συμπεριληφθεί κατά την εκτέλεση του LSH. Αφού εξαλείψουμε τα διπλότυπα, εξάγουμε το αντίστοιχο κείμενο εκπαίδευσης για το κάθε επώνυμο χρησιμοποιώντας τις πληροφορίες για τα βραβεία και τις δημοσιεύσεις που υπάρχουν στο **points_in_range**. Στη συνέχεια, τα τελικά έγκυρα δεδομένα που συλλέχθηκαν για κάθε επιστήμονα (επώνυμο, βραβεία, δημοσιεύσεις, εκπαίδευση) προστίθενται στη λίστα **csv_data** την οποία μετατρέπουμε και σε Dataframe με όνομα **result_df**. Τέλος, από αυτό το Dataframe εξάγουμε τη στήλη με τα κείμενα εκπαίδευσης στη λίστα **education_texts**, την οποία αφού επεξεργαστούμε κατάλληλα με τη **process_education( )** την περνάμε σαν όρισμα στη συνάρτηση **lsh_education( )** μαζί με το **result_df**, έτσι ώστε να κρατήσουμε τα παρόμοια κείμενα εκπαίδευσης με βάση το κείμενο εκπαίδευσης των ατόμων που πληρούν τα κριτήρια του εύρους αναζήτησης. Ταυτόχρονα, λήγει και η χρονομέτρηση, με τη τελική χρονική στιγμή να αποθηκεύεται στο **end_time_2** και το συνολικό χρόνο εκτέλεσης θα δίνεται ως η διαφορά ανάμεσα σε **end_time_2** και **start_time_2**. Στο τέλος, αθροίζουμε τα **elapsed_time_1** και **elapsed_time_2** για να υπολογίσουμε το συνολικό χρόνο εκτέλεσης της αναζήτησης εύρους μαζί με το LSH. Όλη αυτή η διαδικασία που περιγράφουμε εκτελείται μέσα στη συνάρτηση **main_KD_Tree( ),** την οποία καλούμε στη τελευταία γραμμή του κώδικά μας.

```
def main_KD_Tree():
    def result(kdtree, query_surname_range, query_award_range, query_publication_range,
threshold_education):
        # Εκτέλεση αναζήτησης εύρους
        query_point = (query_surname_range, query_award_range, query_publication_range)
```

```
    # Έναρξη χρονομέτρησης range search
    start_time_1 = time.time()
    points_in_range = range_search(kdtree, query_point)
    # Λήξη χρονομέτρησης range search
    end_time_1 = time.time()
    # Υπολογισμός συνολικού χρόνου για αναζήτηση
    elapsed_time_1 = end_time_1 - start_time_1

    # Δημιουργία λίστας για αποθήκευση δεδομένων για το CSV
    csv_data = []
    unique_surnames = set()  # Κρατάμε τα επώνυμα τα οποία διαθέτουν κάποια εκπαίδευση

    for point in points_in_range:
        surname, awards, dblp = point

        # Ελέγχουμε αν έχουμε προσθέσει ήδη μια έγκυρη εγγραφή με αυτό το επώνυμο για να
αφαιρέσουμε τα διπλότυπα
        if surname in unique_surnames:
            continue

        # Εξάγουμε το κείμενο που αφορά την εκπαίδευση για κάθε επιστήμονα
        education_text = df.loc[
            (df['Surname'] == surname) & (df['Awards'] == awards) & (df['Dblp'] == dblp),
            'Education'
        ].values[0]

        # Ελέγχουμε αν το κείμενο δεν είναι 'Education information not found'
        if education_text != 'Education information not found':
            unique_surnames.add(surname)  # Add to the set of unique surnames
            csv_data.append([surname, awards, dblp, education_text])

    # Μετατρέπουμε τα συνολικά δεδομένα του ολικου CSV σε DataFrame
    csv_columns = ['Surname', 'Awards', 'Dblp', 'Education']
    result_df = pd.DataFrame(csv_data, columns=csv_columns)

    # Εκτελούμε LSH ως προς τη στήλη 'Education'
    education_texts = result_df['Education'].tolist()
    education_texts_preprocessed = [preprocess_education(edu) for edu in education_texts]
    # Έναρξη χρονομέτρησης LSH
    start_time_2 = time.time()
    similar_pairs_education = lsh_education(education_texts_preprocessed, result_df,
threshold_education)
    # Λήξη χρονομέτρησης LSH
    end_time_2 = time.time()
    # Υπολογισμός συνολικού χρόνου για LSH
    elapsed_time_2 = end_time_2 - start_time_2

    for pair in similar_pairs_education:
        (entry1, surname1), (entry2, surname2) = pair
        print(f"Surname: {surname1} - Education: {entry1}")
        print(f"Surname: {surname2} - Education: {entry2}")
        print("---")

    result_df_education = pd.DataFrame(similar_pairs_education, columns=['Entry1', 'Entry2'])
    result_df_education.to_csv('kd_similar_pairs.csv', index=False)

    # Συνολικός χρόνος για αναζήτηση και LSH
    elapsed_time = elapsed_time_1 + elapsed_time_2
```

```
    print(f"Time taken for range search with LSH: {elapsed_time} seconds")

  # Εξάγουμε όλα τα σημεία, δηλαδή τους κόμβους του δέντρου
  csv_file_path = 'scientists.csv'
  df = pd.read_csv(csv_file_path)
  points = df[['Surname', 'Awards', 'Dblp']].apply(lambda row: (row['Surname'], row['Awards'],
row['Dblp']),
                                   axis=1).tolist()

  # Κατασκευάζουμε το KD-tree από τα σημεία που προκύπτουν
  kdtree = build_kdtree(points)

  # Εισαγωγή από τον χρήστη των σημείων αναζήτησης
  min_letter = input("Press the first letter of the query (ex.A, B, C...):").upper()
  while not min_letter.isalpha():
    min_letter = input("Press the first letter you want in the right form:")
  max_letter = input("Press the second letter of the query (ex.A, B, C...):").upper()
  while not max_letter.isalpha():
    max_letter = input("Press the second letter you want in the right form:")
  min_awards = int(input("Press the minimum awards of the query (ex.0,1,2...) :"))
  min_dblps = int(input("Press the minimum DBLP records of the query (ex.0,1,2...) :"))
  max_dblps = int(input("Press the maximum DBLP records of the query (ex.0,1,2...) :"))
  lsh_sim = float(input("Give the LSH percentage of similarity between the scientists (ex.0.5->50%):"))
  print("The query that you created is the following: ")
  print(
    "Find the computer scientists from Wikipedia that their surname first letter belongs in the interval ["
+ min_letter + "-" + max_letter + "],have won more than " + str(
      min_awards) + " prizes and their DBLP record belongs in the interval [" + str(
      min_dblps) + " ," + str(max_dblps) + "].")
  print("Generating Answer...")

  # Συγκεντρώνουμε όλα τα εύρη του ερωτήματος
  query_parameters = [
      ((min_letter, max_letter), (min_awards, np.Inf), (min_dblps, max_dblps), lsh_sim)
  ]

  # Καλούμε τη συνάρτηση result( ) για να εκτελέσουμε range search στο κατασκευασμένο δέντρο και
μετά LSH στα τελικά σημεία που επιστράφηκαν
  for surname_range, award_range, publication_range, threshold in query_parameters:
    result(kdtree, surname_range, award_range, publication_range, threshold)

main_KD_Tree()
```

5 Αποτελέσματα

5.1 Αποτελέσματα αναζήτησης εύρους

1) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [C, T], να έχουν αποσπάσει >4 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [4, 58] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >25%.

Surname: Diffie, #Awards: 4, #DBLP_Record: 43

Surname: Joseph, #Awards: 4, #DBLP_Record: 44

Surname: Ritchie, #Awards: 4, #DBLP_Record: 16

Surname: Thompson, #Awards: 7, #DBLP_Record: 38

Surname: Huffman, #Awards: 5, #DBLP_Record: 8

Surname: Hansen, #Awards: 6, #DBLP_Record: 55

Surname: Rossum, #Awards: 5, #DBLP_Record: 14

2) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [K, V], να έχουν αποσπάσει >1 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [5, 145] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >30%.

Surname: Lattner, #Awards: 2, #DBLP_Record: 14

Surname: Rulifson, #Awards: 1, #DBLP_Record: 6

Surname: Rossum, #Awards: 5, #DBLP_Record: 14

Surname: Leiserson, #Awards: 8, #DBLP_Record: 136

Surname: Lederberg, #Awards: 3, #DBLP_Record: 6

Surname: Saif, #Awards: 1, #DBLP_Record: 37

Surname: Ritchie, #Awards: 4, #DBLP_Record: 16

Surname: Schoenebeck, #Awards: 6, #DBLP_Record: 78

Surname: Kurtz, #Awards: 3, #DBLP_Record: 6

Surname: Sadrzadeh, #Awards: 2, #DBLP_Record: 120

Surname: Vapnik, #Awards: 1, #DBLP_Record: 84

Surname: Thompson, #Awards: 7, #DBLP_Record: 38

Surname: Kirstein, #Awards: 2, #DBLP_Record: 54

Surname: Kellis, #Awards: 1, #DBLP_Record: 47

Surname: Neumann, #Awards: 0, #DBLP_Record: 0

Surname: Minsky, #Awards: 2, #DBLP_Record: 33

Surname: Raman, #Awards: 2, #DBLP_Record: 5

Surname: Kayal, #Awards: 3, #DBLP_Record: 80

Surname: Rabin, #Awards: 3, #DBLP_Record: 67

Surname: Milner, #Awards: 3, #DBLP_Record: 107

Surname: Kowalski, #Awards: 1, #DBLP_Record: 10

3) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [B, S], να έχουν αποσπάσει >6 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [3, 246] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >47%.

Surname: Denning, #Awards: 9, #DBLP_Record: 68

Surname: Foley, #Awards: 6, #DBLP_Record: 109

4) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [D, W], να έχουν αποσπάσει >5 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [39, 300] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >36%.

Surname: Dean, #Awards: 6, #DBLP_Record: 111

Surname: Foley, #Awards: 6, #DBLP_Record: 109

Surname: Hennessy, #Awards: 6, #DBLP_Record: 113

Surname: Leiserson, #Awards: 8, #DBLP_Record: 136

Surname: Hansen, #Awards: 6, #DBLP_Record: 55

Surname: Hoare, #Awards: 21, #DBLP_Record: 195

Surname: Denning, #Awards: 9, #DBLP_Record: 68

Surname: Goldwasser, #Awards: 6, #DBLP_Record: 234

Surname: Etzioni, #Awards: 6, #DBLP_Record: 187

Surname: Maulik, #Awards: 11, #DBLP_Record: 214

Surname: Schoenebeck, #Awards: 6, #DBLP_Record: 78

5) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [J, T], να έχουν αποσπάσει >2 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [26, 256] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >12%.

Surname: Minsky, #Awards: 2, #DBLP_Record: 33

Surname: Mitchell, #Awards: 0, #DBLP_Record: 19

Surname: Satyanarayanan, #Awards: 5, #DBLP_Record: 220

Surname: Schoenebeck, #Awards: 6, #DBLP_Record: 78

Surname: Rabin, #Awards: 3, #DBLP_Record: 67

Surname: Pavón, #Awards: 2, #DBLP_Record: 148

Surname: Leiserson, #Awards: 8, #DBLP_Record: 136

Surname: Kayal, #Awards: 3, #DBLP_Record: 80

Surname: Kirstein, #Awards: 2, #DBLP_Record: 54

Surname: Maulik, #Awards: 11, #DBLP_Record: 214

Surname: Jones, #Awards: 0, #DBLP_Record: 0

Surname: Joseph, #Awards: 4, #DBLP_Record: 44

Surname: Liskov, #Awards: 2, #DBLP_Record: 163

Surname: Milner, #Awards: 3, #DBLP_Record: 107

Surname: Sadrzadeh, #Awards: 2, #DBLP_Record: 120

Surname: Kleinrock, #Awards: 3, #DBLP_Record: 181

Surname: Tardos, #Awards: 2, #DBLP_Record: 235

Surname: Thompson, #Awards: 7, #DBLP_Record: 38

6) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [A, G], να έχουν αποσπάσει >3 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [38, 380] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >26%.

Surname: Comer, #Awards: 8, #DBLP_Record: 82

Surname: Diffie, #Awards: 4, #DBLP_Record: 43

Surname: Denning, #Awards: 9, #DBLP_Record: 68

Surname: Bahl, #Awards: 17, #DBLP_Record: 127

Surname: Ahn, #Awards: 6, #DBLP_Record: 39

Surname: Abebe, #Awards: 9, #DBLP_Record: 40

Surname: Dean, #Awards: 6, #DBLP_Record: 111

Surname: Blum, #Awards: 0, #DBLP_Record: 0

Surname: Anthony, #Awards: 6, #DBLP_Record: 77

Surname: Aaronson, #Awards: 7, #DBLP_Record: 230

Surname: Fu, #Awards: 3, #DBLP_Record: 76

Surname: Abelson, #Awards: 5, #DBLP_Record: 51

Surname: Foley, #Awards: 6, #DBLP_Record: 109

Surname: Etzioni, #Awards: 6, #DBLP_Record: 187

Surname: Brassard, #Awards: 3, #DBLP_Record: 133

7) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [D, O], να έχουν αποσπάσει >0 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [22, 717] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >52%.

Surname: Hewitt, #Awards: 3, #DBLP_Record: 63

Surname: Liskov, #Awards: 2, #DBLP_Record: 163

Surname: Diffie, #Awards: 4, #DBLP_Record: 43

Surname: Irwin, #Awards: 14, #DBLP_Record: 370

Surname: Iverson, #Awards: 2, #DBLP_Record: 45

Surname: Liedtke, #Awards: 0, #DBLP_Record: 37

Surname: Hansen, #Awards: 6, #DBLP_Record: 55

Surname: Graham, #Awards: 0, #DBLP_Record: 48

Surname: Meyer, #Awards: 3, #DBLP_Record: 356

Surname: Golumbic, #Awards: 0, #DBLP_Record: 109

Surname: Fu, #Awards: 3, #DBLP_Record: 76

Surname: Nadin, #Awards: 0, #DBLP_Record: 49

8) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [Q, V], να έχουν αποσπάσει >2 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [55, 167] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >33%.

Surname: Sadrzadeh, #Awards: 2, #DBLP_Record: 120

Surname: Schoenebeck, #Awards: 6, #DBLP_Record: 78

Surname: Rabin, #Awards: 3, #DBLP_Record: 67

9) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [I, P], να έχουν αποσπάσει >3 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [109, 450] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >42%.

Surname: Irwin, #Awards: 14, #DBLP_Record: 370

Surname: Kleinrock, #Awards: 3, #DBLP_Record: 181

Surname: Maulik, #Awards: 11, #DBLP_Record: 214

Surname: Meyer, #Awards: 3, #DBLP_Record: 356

10) Βρείτε τους επιστήμονες της επιστήμης υπολογιστών από τη ΒΔ Wikipedia που το γράμμα τους να ανήκει στο διάστημα [B, X], να έχουν αποσπάσει >0 βραβεία, ο αριθμός δημοσιεύσεων στο DBLP Record να ανήκει στο εύρος [4, 500] και να έχουν ποσοστό ομοιότητας εκπαίδευσης >55%.

Surname: Conway, #Awards: 0, #DBLP_Record: 18

Surname: Brin, #Awards: 2, #DBLP_Record: 15

Surname: Iverson, #Awards: 2, #DBLP_Record: 45

Surname: Burnett, #Awards: 2, #DBLP_Record: 262

Surname: Irwin, #Awards: 14, #DBLP_Record: 370

Surname: Fu, #Awards: 3, #DBLP_Record: 76

Surname: Pavón, #Awards: 2, #DBLP_Record: 148

Surname: Rabin, #Awards: 3, #DBLP_Record: 67

Surname: Hansen, #Awards: 6, #DBLP_Record: 55

Surname: Fredkin, #Awards: 1, #DBLP_Record: 8

Surname: Nadin, #Awards: 0, #DBLP_Record: 49

Surname: Diffie, #Awards: 4, #DBLP_Record: 43

Surname: Starner, #Awards: 0, #DBLP_Record: 266

Surname: Golumbic, #Awards: 0, #DBLP_Record: 109

Surname: Shaw, #Awards: 0, #DBLP_Record: 66

Surname: Blum, #Awards: 0, #DBLP_Record: 0

Surname: Liskov, #Awards: 2, #DBLP_Record: 163

Surname: Liedtke, #Awards: 0, #DBLP_Record: 37

Surname: Minsky, #Awards: 2, #DBLP_Record: 33

5.2 Σύγκριση με θεωρητικούς χρόνους εκτέλεσης

Γενικά, αναμένουμε οι χρόνοι αναζήτησης σε κάθε δέντρο να ακολουθούν τις πιο κάτω χρονικές πολυπλοκότητες:

	Average Case	Worst Case
kd-tree	$O(\log_2(n))$	O(n)
Octree	$O(\log_{2^N}(n))$	O(n)
Range tree	$O(\log_M(n))$	O(n)
R-tree	$O(\log_2(n) + k)$	O(n)

όπου: N είναι το πλήθος διαστάσεων, δηλαδή θα είναι $O(\log_8(n))$ για το Octree,

M το μέγιστο πλήθος καταχωρίσεων σε κάθε σελίδα του R-tree,

k το πλήθος των κόμβων που επιστρέφει η αναζήτηση στο Range tree.

Πιο κάτω αναλύουμε με βάση τα παραπάνω κάποια βασικά μειονεκτήματα και πλεονεκτήματα κάθε δομής:

kd-tree:

Πλεονεκτήματα:

1) Απλή δομή.
2) Αποτελεσματικό για μικρό αριθμό διαστάσεων.
3) Καλή απόδοση για δεδομένα με μικρή διάσταση.

Μειονεκτήματα:

1) Μπορεί να μην είναι τόσο αποτελεσματικό σε υψηλές διαστάσεις λόγω του curse of dimensionality.
2) Η απόδοσή της μπορεί να επηρεαστεί από τη σειρά εισαγωγής των σημείων.

Octree:

Πλεονεκτήματα:

1) Αποτελεσματική για μεγάλο αριθμό διαστάσεων.
2) Μπορεί να αντιμετωπίσει το πρόβλημα του του curse of dimensionality.
3) Κατάλληλο για δεδομένα χαμηλής πυκνότητας.

Μειονεκτήματα:

1) Αυξημένη πολυπλοκότητα σε σύγκριση με το KD-Tree.
2) Απαιτεί περισσότερη μνήμη.

R-tree:

Πλεονεκτήματα:

1) Κατάλληλο για χώρους με υψηλή πυκνότητα.
2) Καλή απόδοση για δεδομένα μεγάλης διάστασης.
3) Αποτελεσματικό για εύρεση κοντινών γειτόνων.

Μειονεκτήματα:

1) Υψηλή πολυπλοκότητα στην εισαγωγή και ενημέρωση δεδομένων.
2) Απαιτεί συνεχή επικοινωνία με τη μνήμη λόγω της δομής του.

Range tree:
Πλεονεκτήματα:
1) Καλή απόδοση για πολλαπλά ερωτήματα εύρους.
2) Αποτελεσματική για διαστάσεις.
3) Εξαιρετική απόδοση σε ερωτήματα εύρους σε διαστασιολόγια σε μεγάλες διαστάσεις.

Μειονεκτήματα:
1) Υψηλή πολυπλοκότητα χώρου και μνήμης.
2) Υψηλή πολυπλοκότητα εισαγωγής και ενημέρωσης δεδομένων.

5.3 Σύγκριση με πειραματικούς χρόνους εκτέλεσης

Τελικό μας βήμα είναι να συγκρίνουμε τα δέντρα δεδομένων που παρουσιάσαμε παραπάνω.

Με την εκτέλεσης του αρχείου All_trees.py ο χρήστης μπορεί να επιλέξει ποιο δέντρο θέλει να εκτελέσει επιλέγοντας τον αριθμό που του αντιστοιχεί.

```
from R_Tree import *
from Range_Tree import *
from Oc_Tree import *
from KD_Tree import *

def switch(tree):
    if tree == "1":
        print("\n")
        return main_R_tree()
    elif tree == "2":
        print("\n")
        return main_Range_Tree()
    elif tree == "3":
        print("\n")
        return main_Oc_Tree()
    elif tree == "4":
        print("\n")
        return main_KD_Tree()

print("If you want to search with R-Tree press 1")
print("If you want to search with Range-Tree press 2")
print("If you want to search with Oc-Tree press 3")
print("If you want to search with KD-Tree press 4")
Decision = switch(input("\nChoose a number:"))
```

Για να καταφέρουμε να συγκρίνουμε αποτελεσματικά τους κώδικες μας ως προς την ταχύτητα εκτέλεσης τους, προτείναμε ερωτήματα(queries) και για κάθε δέντρο αποθηκεύσαμε τους χρόνους εκτέλεσης που απαιτεί κάθε ένα σε ένα αρχείο csv με όνομα **Times.csv.**

Για την αναπαράσταση και διαχείριση DataFrame και την εκτύπωση γραφικών παραστάσεων χρησιμοποιούμε τις παρακάτω βιβλιοθήκες:

```
import pandas as pd
import matplotlib.pyplot as plt
```

Αρχικά μεταφέραμε σε τέσσερεις λίστες τους χρόνους εκτέλεσης για κάθε είδος δέντρου (KD-tree, R-tree, Range-tree, Oct-tree) από το DataFrame που ορίσαμε να περιείχε τα στοιχεία του csv με τον εξής τρόπο :

```
# Διάβασμα των χρόνων απο το Times.csv
df = pd.read_csv('Times.csv')

# Εξαγωγή δεδομένων για κάθε δέντρο
kd_tree_data = df[df['Tree'] == 'Kd-tree']['Time'].tolist()
r_tree_data = df[df['Tree'] == 'R-tree']['Time'].tolist()
range_tree_data = df[df['Tree'] == 'Range-tree']['Time'].tolist()
oc_tree_data = df[df['Tree'] == 'Oc-tree']['Time'].tolist()
```

Στην συνέχεια χρησιμοποιώντας τις συναρτήσεις της βιβλιοθήκης matplotlib απεικονίζουμε τους χρόνους εκτέλεσης για κάθε είδος δέντρου σε ένα γράφημα γραμμής όπου κάθε δέντρο έχει τον χρόνο εκτέλεσής του για κάθε αριθμό ερωτήσεων, όπως ορίζεται από τη λίστα query_count.

```
# Παρουσίαση δεδομένων σε γραφική παράσταση
query_count = list(range(1, 11))
plt.figure(figsize=(10, 6))
plt.plot(query_count, kd_tree_data, label='KD-tree', marker='o', color='blue')
plt.plot(query_count, r_tree_data, label='R-tree', marker='o', color='green')
plt.plot(query_count, range_tree_data, label='Range-tree', marker='o', color='orange')
plt.plot(query_count, oc_tree_data, label='Oc-tree', marker='o', color='red')
plt.xlabel('Queries')
plt.ylabel('Time (seconds)')
plt.title('Time Taken for Different Queries')
plt.xticks(rotation=45)
plt.legend()
```

Επίσης, υπολογίζουμε τους μέσους χρόνους εκτέλεσης για κάθε δέντρο, χρησιμοποιώντας τη μέθοδο **mean( )** για τα δεδομένα του DataFrame df που αντιστοιχούν σε κάθε είδος δέντρου. Έπειτα, ορίζουμε τις λίστες trees και average_times, που περιέχουν τα είδη των δέντρων και τους μέσους χρόνους εκτέλεσης αντίστοιχα.

```
# Υπολογισμός Μ.Ο χρόνων για κάθε δέντρο
average_kd_time = df[df['Tree'] == 'Kd-tree']['Time'].mean()
average_r_time = df[df['Tree'] == 'R-tree']['Time'].mean()
average_range_time = df[df['Tree'] == 'Range-tree']['Time'].mean()
average_oc_time = df[df['Tree'] == 'Oc-tree']['Time'].mean()
# Απεικόνηση δεδομένων σε διάγρμα πίτας
trees = ['KD-tree', 'R-tree', 'Range-tree', 'Oc-tree']
average_times = [average_kd_time, average_r_time, average_range_time, average_oc_time]
```

Χρησιμοποιώντας πάλι την βιβλιοθήκη matplotlib δημιουργούμε ένα διάγραμμά-πίτα με τη χρήση της συνάρτησης **plt.pie( ).** Σε αυτήν, με διαφορετικό χρώμα, απεικονίζεται για κάθε είδος δέντρου ο μέσος χρόνος εκτέλεσης του.

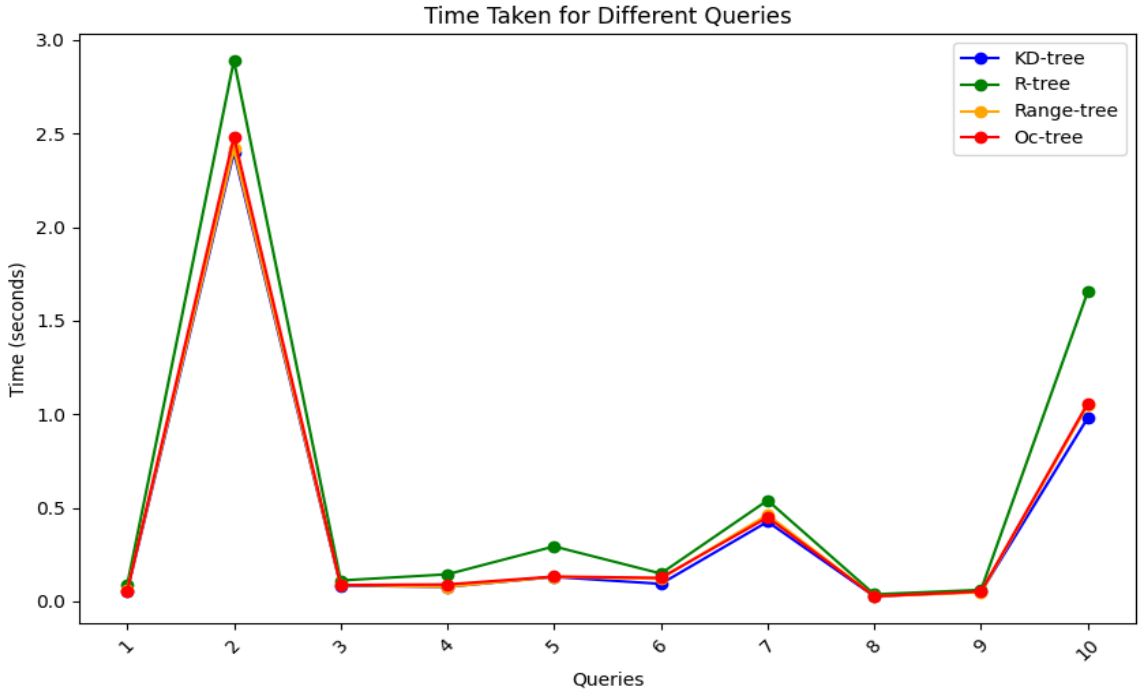
```
plt.figure(figsize=(8, 8))
plt.pie(average_times, labels=trees, autopct='%1.1f%%', colors=['blue', 'green', 'orange', 'red'],
startangle=90)
plt.title('Average Time for Each Tree Across All Queries')
plt.show()
```

Για την καταμέτρηση των χρόνων εκτέλεσης του κάθε δέντρου τρέξαμε τους κώδικες μας για τα παρακάτω διαστήματα αναζήτησης:

Διάστημα ονομάτων επιστημόνων	Κατώτατο όριο πλήθους βραβείων	Διάστημα πλήθους Dbpl	Ποσοστό ομοιότητας βάση εκπαίδευσης
'C', 'T'	4	4, 58	0.25
'K', 'V'	1	5, 145	0.3
'B', 'S'	6	3, 246	0.47
'D', 'W'	5	39, 300	0.36
'J', 'T'	2	26, 256	0.12
'A', 'G'	3	38, 380	0.26
'D', 'O'	0	22, 717	0.52
'Q', 'V'	2	55, 167	0.33
'I', 'P'	3	109, 450	0.42
'B', 'X'	0	4, 500	0.55

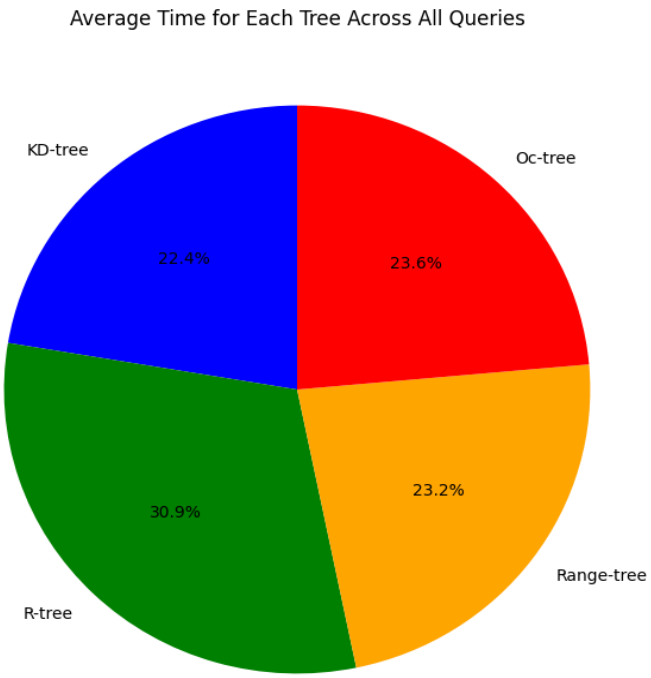
Έπειτα χρησιμοποιούμε τον κώδικα με τον οποίο θα αναπαραστήσαμε τους χρόνους απόκρισης σε διαγράμματα και τον μέσο χρόνο απόκρισης για να βρούμε ποια δομή από τις παραπάνω είναι βέλτιστη. Όπως προαναφέραμε, η εκκίνηση της καταμέτρησης του χρόνου ξεκινάει από την στιγμή που δημιουργείται το δέντρο μέχρι την στιγμή που θα ολοκληρωθεί η LSH αναζήτηση.

Η τελική γραφική παράσταση παρατίθεται παρακάτω:



Στο διπλανό σχήμα απεικονίζεται το ποσοστό του χρόνου που χρειάστηκε κατά μέσο όρο κάθε πρόγραμμα για να επιστρέψει τα τελικά αποτελέσματα.

Παρατηρούμε ότι τον καλύτερο χρόνο απόκρισης έχουν τα kd-tree με ποσοστό 22.4% Επομένως είναι και το πιο γρήγορα στην αναζήτηση που κάνουμε. Ακολουθεί το Range tree με 23.2% και αμέσως μετά, με μικρή απόκλιση από το προηγούμενο, το Octree. Τελευταίο σε ταχύτητα εκτέλεσης είναι το R-tree (30.9%) το οποίο και έχει τον χειρότερο χρόνο απόκρισης.



Συμπέρασμα:
Παρατηρούμε ότι για 10 συγκεκριμένα ερωτήματα περισσότερο χρόνο κατά μέσο όρο για αναζήτηση με LSH απαιτεί το R-tree. Έπειτα, ακολουθούν τα Octree και Range tree με σχεδόν παρόμοιες επιδόσεις. Τέλος, πιο γρήγορο από όλα τα δέντρα (αλλά πολύ κοντά στα Range trees και Octrees) αποδεικνύεται ότι είναι το kd-tree. Γενικά, όλα τα δέντρα λειτουργούν ορθά, εφόσον επιστρέφουν τα ίδια σημεία κατά την αναζήτηση, ωστόσο παρουσιάζουν μικρές αποκλίσεις όσον αφορά τη ταχύτητα.

GitHub link of the project:
https://github.com/chryssapat/Multidimesional_Data_Structures.git