ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

ΜΑΘΗΜΑ: **Τεχνολογίες Αποκεντρωμένων Δεδομένων**

ΤΙΤΛΟΣ ΕΡΓΑΣΙΑΣ

Implementation and Experimental Evaluation of Basic DHTs

Δελημπαλταδάκης Γρηγόριος
ΑΜ: 1084647

Διασάκος Δαμιανός
ΑΜ: 1084632

Ζήκος Σπυρίδων
ΑΜ: 1084581

Κυριακουλόπουλος Καλλίνικος
ΑΜ: 1084583

ΠΑΤΡΑ
ΦΕΒΡΟΥΑΡΙΟΣ 2025

Contents

1. Project Overview

Αυτό το έργο αφορά την υλοποίηση και σύγκριση δύο κατανεμημένων πρωτοκόλλων, των Pastry [1] και Chord [2], που χρησιμοποιούνται για την αποθήκευση και ανάκτηση δεδομένων σε ένα κατανεμημένο σύστημα DHT (Distributed Hash Table).

1.1. Project Structure

Παρακάτω παραθέτουμε το file system tree του project.

```
Implementation-of-Basic-DHTs
├── compare.py
├── constants.py
├── dashboard.py
├── helper_functions.py
├── main.py
├── requirements.txt
│
├── Chord
│   ├── ChordResults.json
│   ├── chord_gui.py
│   ├── network.py
│   ├── node.py
│   ├── plot_hops.py
│   ├── test.py
│   │
│   ├── Plots
│   │   └── chord_hops.png
│   │
│   └── Tests
│       ├── demo.py
│       ├── multithreaded_test.py
│       └── unit_tests.py
│
├── Coffee_Reviews_Dataset
│
├── Comparison Plot
│   └── dht_performance_comparison.png
│
├── Multidimensional_Data_Structures
│   ├── kd_tree.py
│   └── lsh.py
│
└── Pastry
    ├── network.py
    ├── node.py
    ├── pastry_gui.py
    │
    └── Tests
        ├── multithreaded_test.py
        ├── test.py
        │
        └── Hops
            ├── all_test.py
            ├── build_test.py
            ├── delete_test.py
            ├── leave_test.py
            ├── lookup_test.py
            ├── PastryResults.json
            ├── plot_hops.py
            ├── update_test.py
            │
            └── Plots
                └── pastry_hops.png
```

Στο root του project directory υπάρχει το main script (main.py), για την εκτέλεση της συνολικής υλοποίησης μέσω του GUI. Συμπληρωματικά, στο root έχουν τοποθετηθεί κοινές σταθερές και βοηθητικές συναρτήσεις (constants.py, helper_functions.py), το parent class για το GUI (dashboard.py) και ένα script για την σύγκριση των πειραματικών αποτελεσμάτων των δύο DHT's (compare.py).

Στο Multidimensional_Data_Structures directory, έχουν το τοποθετηθεί η υλοποιήσεις του KD-Tree (kd_tree.py) και του μηχανισμού LSH (lsh.py).

Στο Chord directory, βρίσκεται η υλοποίηση του Chord DHT, με scripts για την περιγραφή του δικτύου (network.py), του κόμβου (node.py) και του γραφικού περιβάλλοντος του (chord_gui.py). Συμπληρωματικά, σε αυτόν τον φάκελο και στον υποφακέλο Tests βρίσκονται test scripts, για τον έλεγχο της λειτουργίας του DHT και την καταγραφή πειραματικών αποτελεσμάτων.

Στο Pastry directory, βρίσκεται η υλοποίηση του Pastry DHT, με scripts για την περιγραφή του δικτύου (network.py), του κόμβου (node.py) και του γραφικού περιβάλλοντος του (pastry_gui.py). Συμπληρωματικά, στον υποφάκελο Tests, βρίσκονται test scripts, για τον έλεγχο της λειτουργίας του DHT και την καταγραφή πειραματικών αποτελεσμάτων.

1.2. Execution Instructions

Αρχικά, πριν την εκτέλεση πρέπει να εγκαταστήσετε τα requirements του project. Οπότε στο root του project εκτελέστε:

```
Implementation-of-Basic-DHTs> pip install -r requirements.txt
```

Η εκτέλεση της υλοποίησης έχει επικυρωθεί για python versions >= 3.11.5.

Για την εκτέλεση του project προσφέρονται κάποιες εναλλακτικές επιλογές.

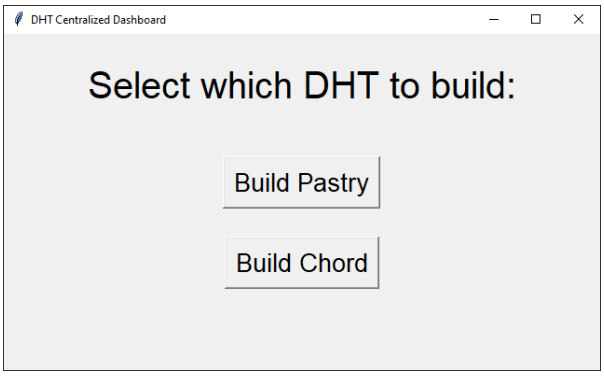
1.2.1. Main Execution Method (GUI)

Ο κύριος τρόπος εκτέλεσης πραγματοποιείται μέσω του script main.py.

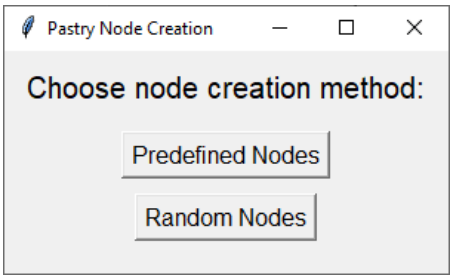
```
Implementation-of-Basic-DHTs> python main.py
```

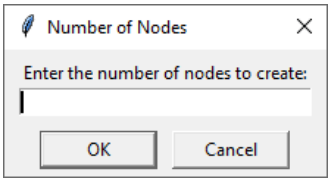
Με την εκτέλεση αυτού του script, ξεκινά το γραφικό περιβάλλον (GUI) της εφαρμογής.

Αρχικά εμφανίζεται το main dashboard, οπού μπορεί να γίνει επιλογή του DHT που θα αρχικοποιηθεί.



Για κάθε DHT παρέχεται επιλογή για την αρχικοποίηση με 13 προκαθορισμένους κόμβους, που είναι hardcoded στο constants.py. Εναλλακτικά μπορεί να γίνει αρχικοποίηση με έναν αριθμό από κόμβους με τυχαία IDs. Στην δεύτερη περίπτωση προτείνουμε ο αριθμός να μην ξεπεράσει τους 20 κόμβους, λαμβάνοντας υπόψιν τους περιορισμούς μιας αποκεντρωμένης προσομοίωσης, σε πόρους συστήματος και socket communication overhead.

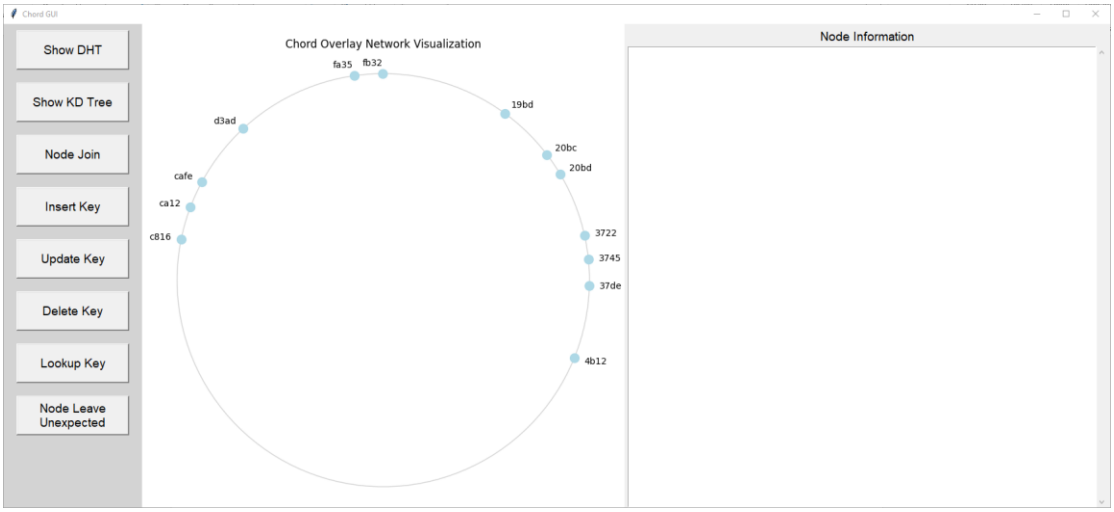




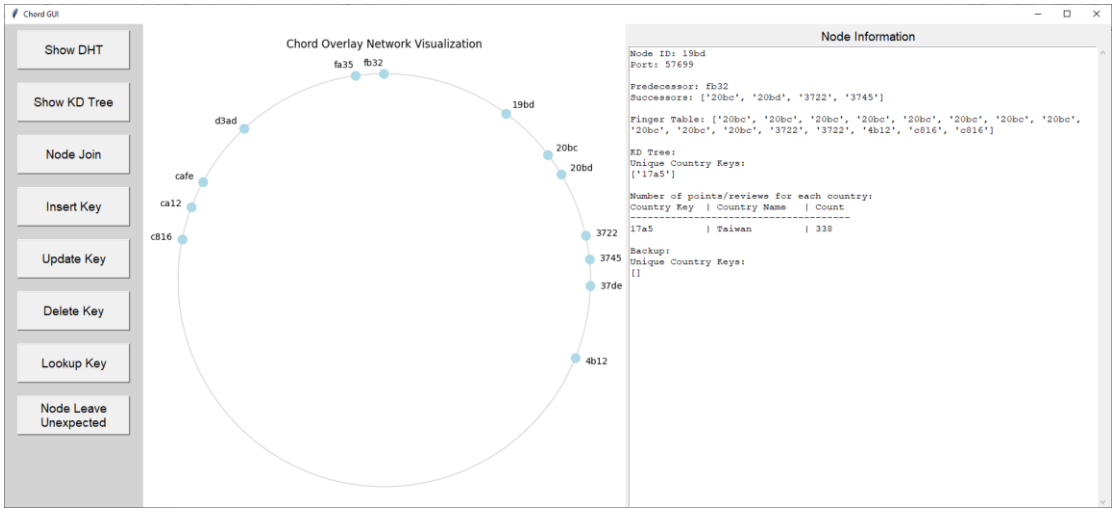
Μετά την επιλογή μεθόδου αρχικοποίησης, περιμένετε λίγα δευτερόλεπτα για την εισαγωγή των κόμβων και κλειδιών από το dataset. Όταν ολοκληρωθεί η αρχικοποίηση εμφανίζεται το dashboard για το επιλεγμένο DHT.

Chord Dashboard

Αριστερά παρέχονται κουμπιά για την εκτέλεση των υλοποιημένων λειτουργιών. Κεντρικά φαίνεται μια οπτικοποίηση των κόμβων πάνω στο ring, με βάση το ID τους και δεξιά εμφανίζεται η κατάσταση ενός κόμβου, όταν αυτός επιλεχθεί στο ring.

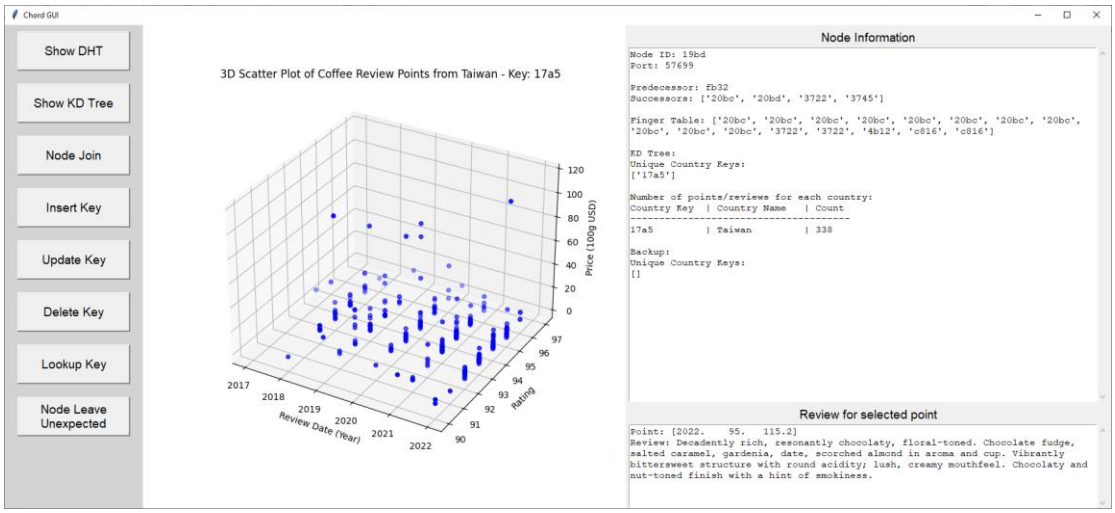


Για παράδειγμα, επιλέγοντας τον κόμβο 19bd, φαίνονται ο predecessor του, οι successors, το finger table και τα κλειδιά που έχει στο KD-Tree του.



Κουμπί Show KD Tree:

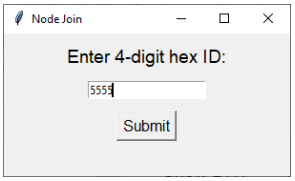
Αφού επιλεχθεί ένας κόμβος, μπορεί να πατηθεί το κουμπί Show KD Tree για την εμφάνιση ενός 3D scatter plot με τα σημεία μίας επιλεγμένης χώρας.

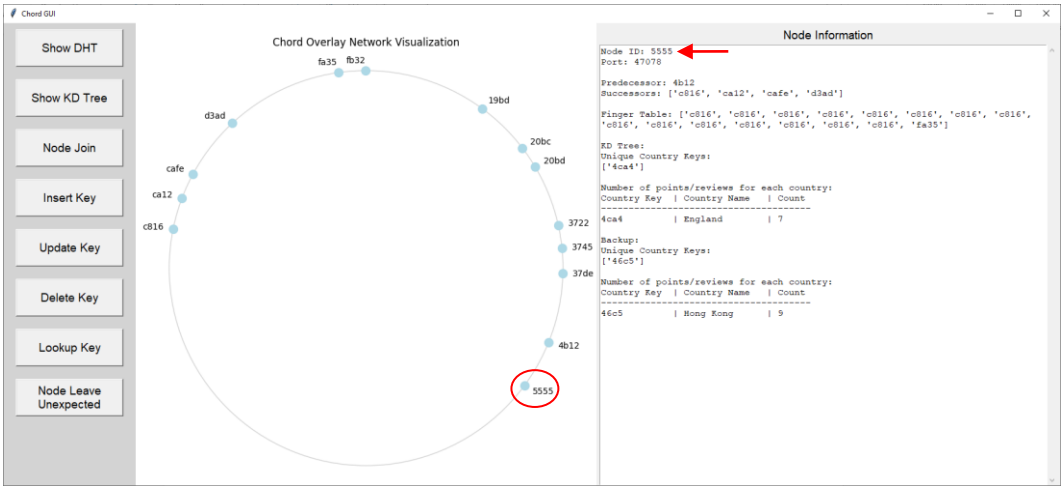


Σε αυτή την φάση μπορεί να επιλεχθεί οποιοδήποτε σημείο στο plot για την εμφάνιση του αντίστοιχου review κάτω δεξιά.

Κουμπί Node Join:

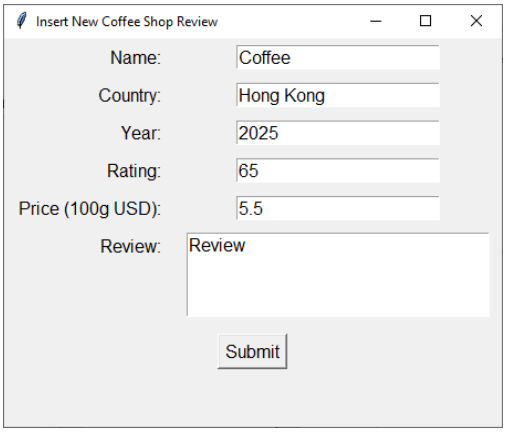
Πατώντας αυτό το κουμπί, μπορεί να εισαχθεί ένας νέος κόμβος στο δίκτυο, ενημερώνοντας την οπτικοποίηση του ring και τις καταστάσεις των υπόλοιπων κόμβων.

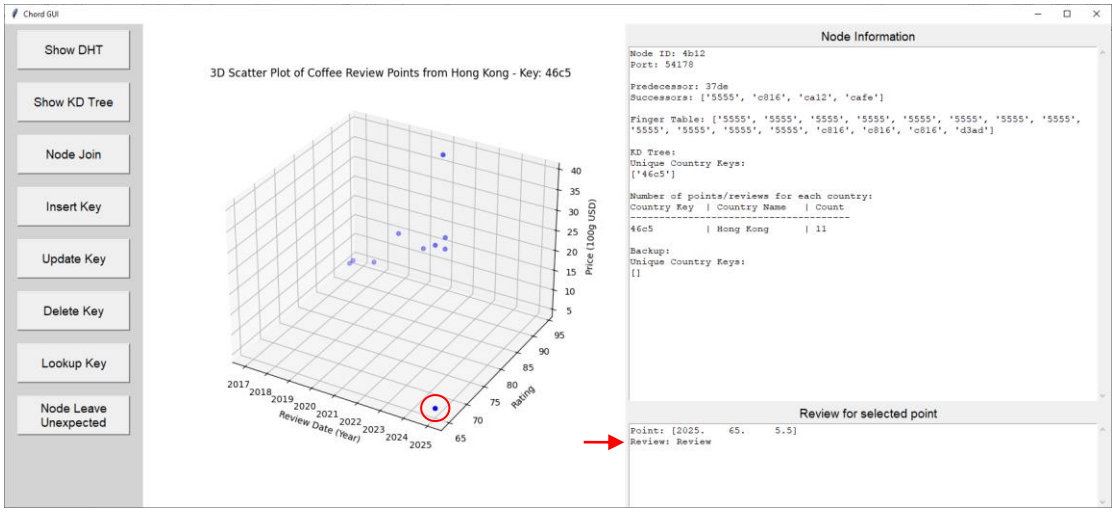




Κουμπί Insert Key:

Εισάγει ένα κλειδί στον κατάλληλο κόμβο του δικτύου, ξεκινώντας το αίτημα από τον επιλεγμένο κόμβο.

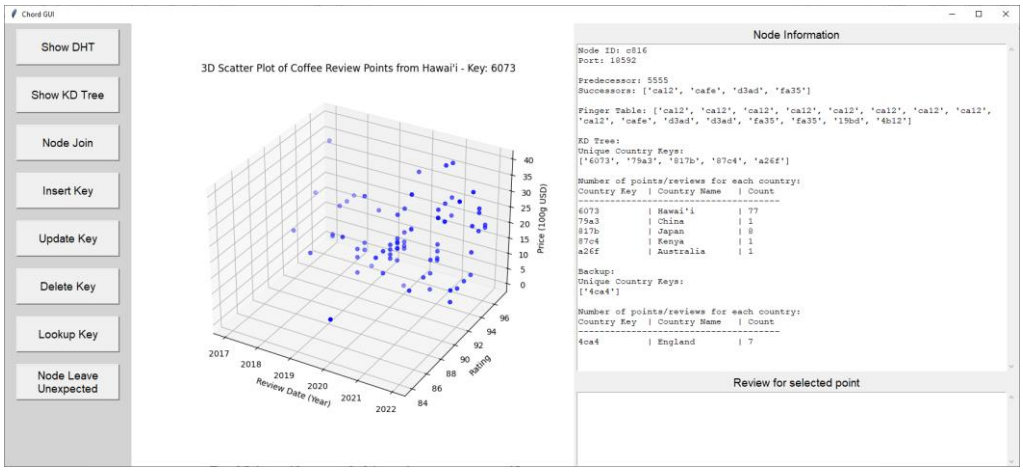




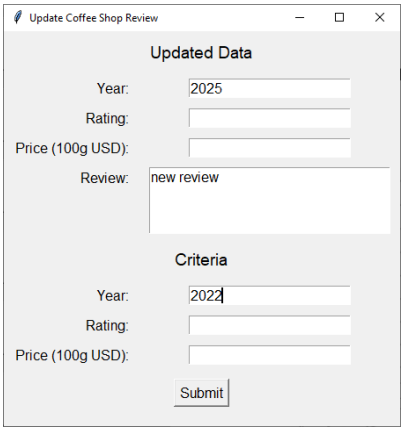
Κουμπί Update Key:

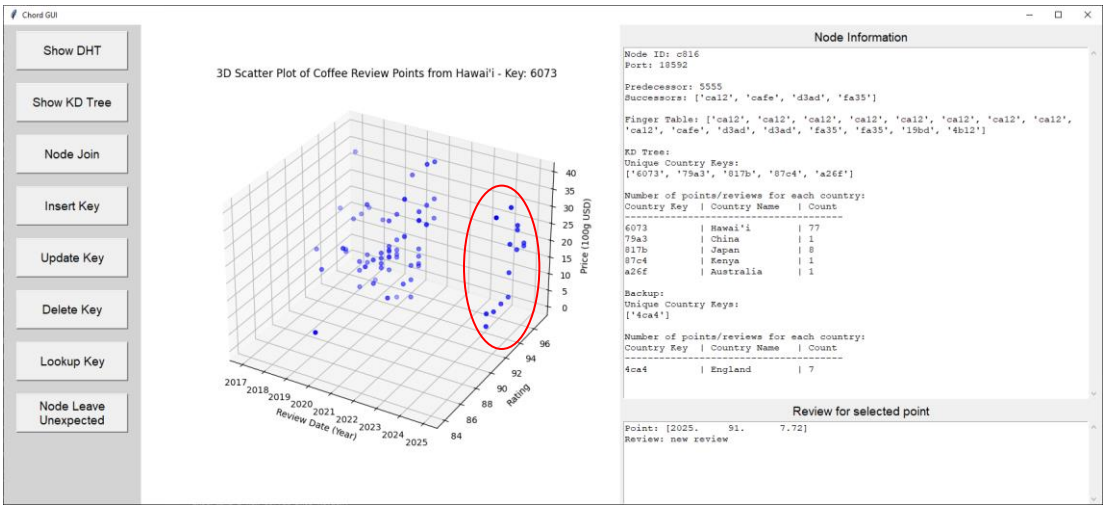
Ενημερώνει τα σημεία και reviews ενός key, βάση συγκεκριμένων κριτηρίων.

Για παράδειγμα μπορούμε να τροποποιήσουμε την χρονολογία όλων των σημείων του κλειδιού που αντιστοιχεί στην Hawaii και έχουν year = 2022, σε 2025.



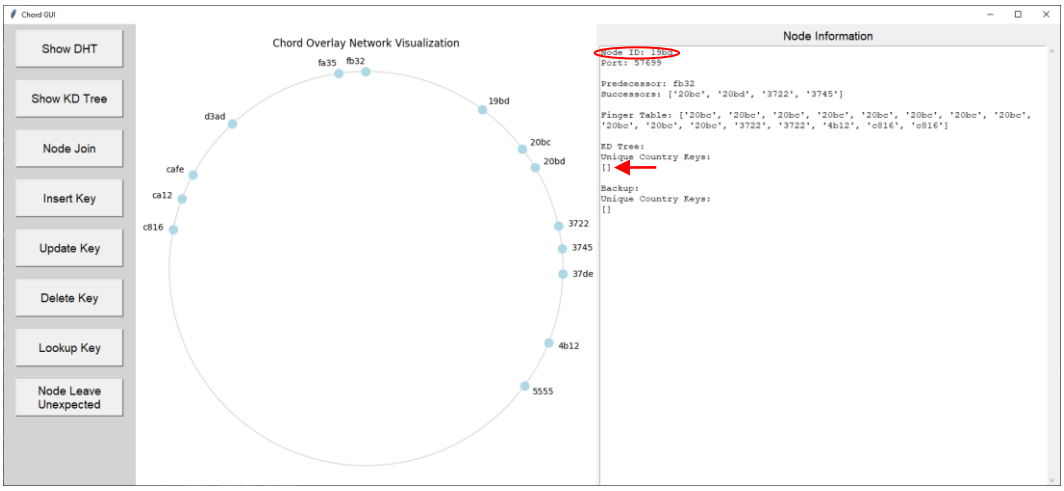
Αποτέλεσμα:





Κουμπί Delete Key:

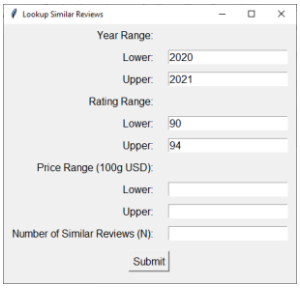
Διαγράφει όλα τα points και reviews για ένα key.

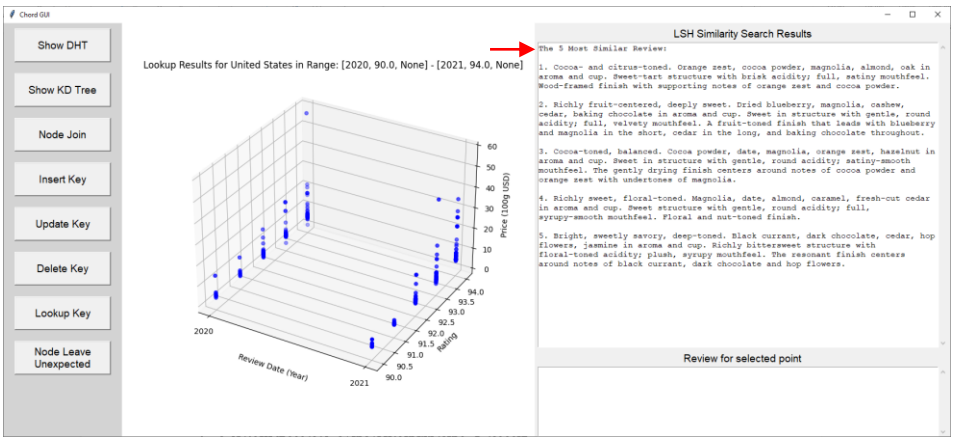


Κουμπί Lookup Key:

Πραγματοποιεί range search εντός κάποιων bounds για κάποιο κλειδί και εμφανίζει τα N reviews με την μεγαλύτερη ομοιότητα.

Για παράδειγμα μπορούμε να κάνουμε lookup πάνω στο κλειδί που αντιστοιχεί στην χώρα “United States” και να αναζητήσουμε σημεία με έτος ανάμεσα στο [2020, 2021] και rating ανάμεσα στο [90, 94].



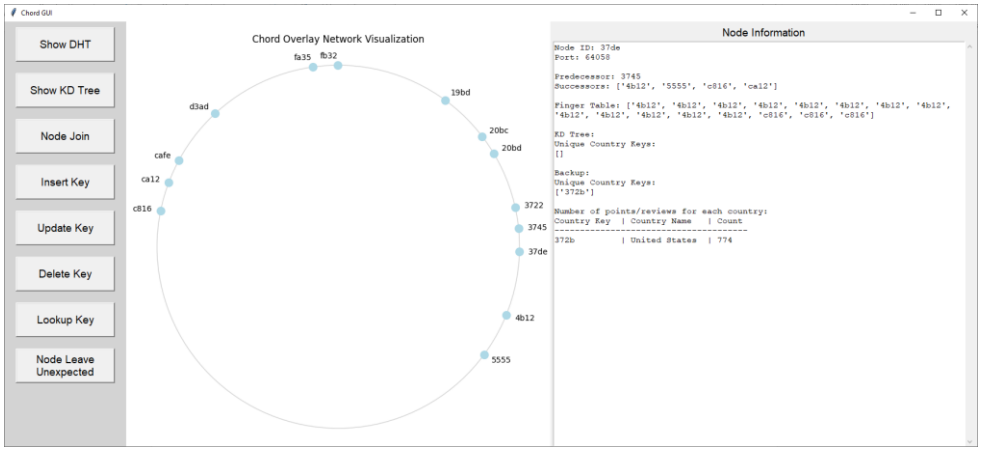


Κουμπί Node Leave Unexpected:

Παρέχει την δυνατότητα προσομοίωσης της αποτυχίας ενός επιλεγμένου κόμβου. Μετά την αποχώρηση, οι καταστάσεις των υπόλοιπων κόμβων θα ενημερωθούν κατάλληλα, μέσω της περιοδικής ενημέρωσης που πραγματοποιούν πάνω στο finger table και το successor list τους.

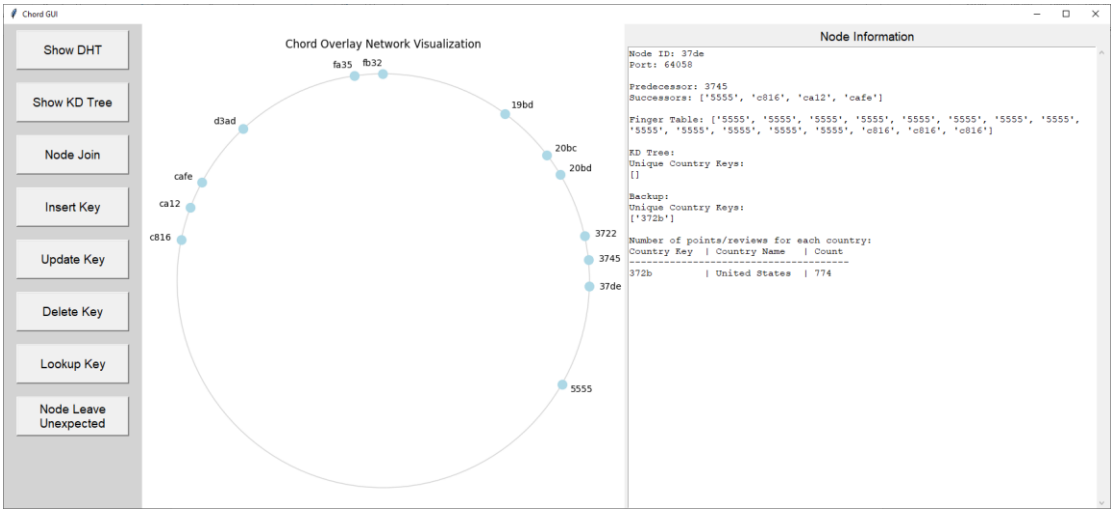
Για παράδειγμα, ελέγχουμε την αποχώρηση του κόμβου 4b12 και παρατηρούμε την κατάσταση του 37de.

Πριν την αποχώρηση του 4b12:

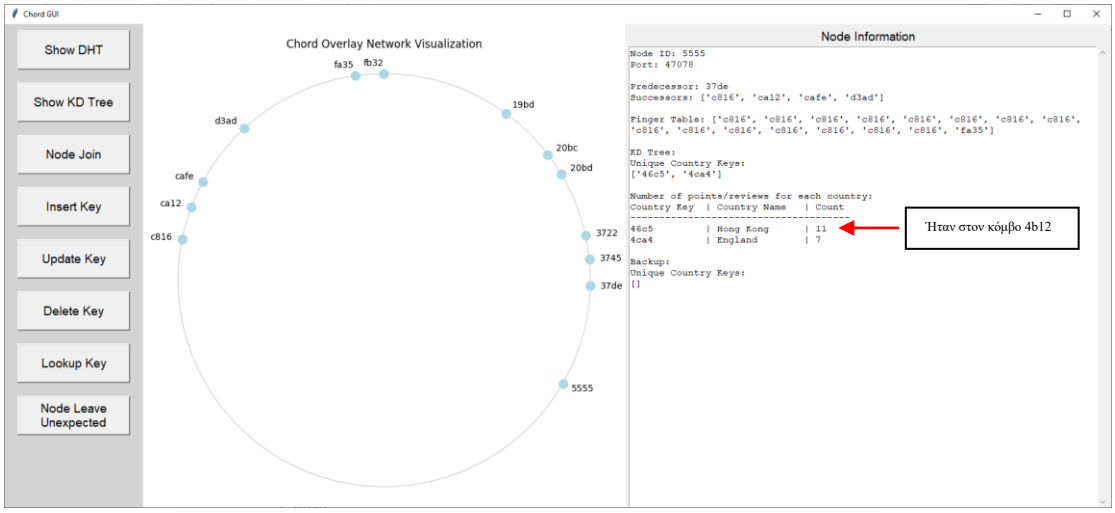


Μετά την αποχώρηση του 4b12:

Παρατηρούμε την σωστή ενημέρωση των successors και του finger table του 37de.



Συμπληρωματικά, αναφέρεται πως τα δεδομένα του κόμβου 4b12 δεν χάθηκαν, καθώς ήταν backed up στον κόμβο 5555.

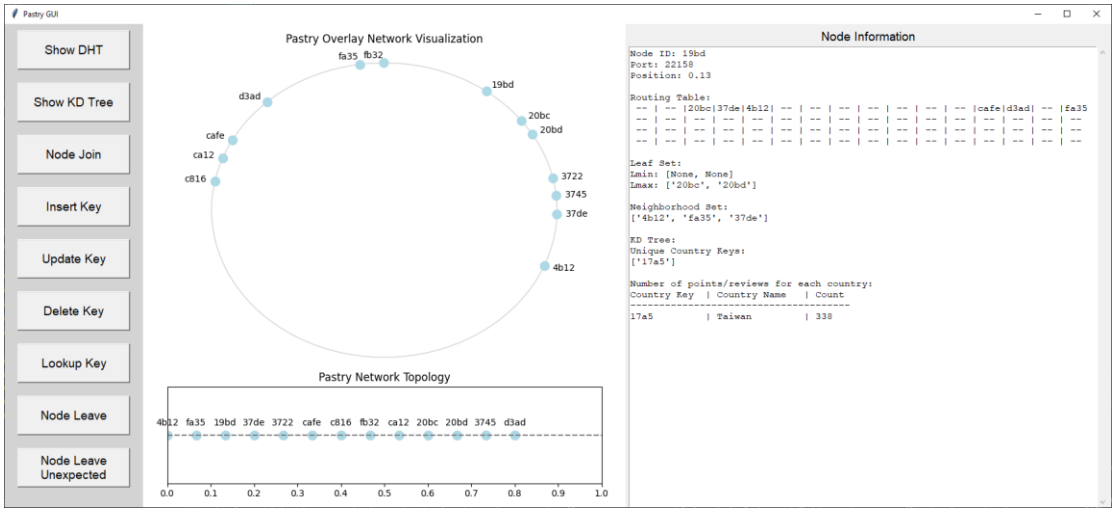

Chord GUI
Show DHT
Show KD Tree
Node Join
Insert Key
Update Key
Delete Key
Lookup Key
Node Leave Unexpected
Chord Overlay Network Visualization
fa35
fb32
19bd
20bc
20bd
3722
3745
37de
5555
c816
ca12
cafe
d3ad
Node Information
Node ID: 5555
Port: 47078
Predecessor: 37de
Successors: ['c816', 'ca12', 'cafe', 'd3ad']
Finger Table: ['c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'c816', 'fa35']
KD Tree:
Unique Country Keys:
['46c5', '4ca4']
Number of points/reviews for each country:
Country Key | Country Name | Count
46c5 | Hong Kong | 11
4ca4 | England | 7
Backup:
Unique Country Keys:
[]
Ήταν στον κόμβο 4b12

Pastry Dashboard

Στο Pastry Dashboard οι περισσότερες λειτουργίες είναι ίδιες με το Chord, οπότε θα αναφερθούν μόνο τα επιπλέον στοιχεία.

Αρχικά, στο Pastry λαμβάνουμε υπόψιν και την γεωγραφική τοπολογία των κόμβων, που στην περίπτωσή μας έχει προσομοιωθεί μέσω του attribute position των κόμβων, το οποίο παίρνει τιμές στο εύρος [0, 1]. Οπότε η αναπαράσταση του δικτύου συμπληρώνεται με την οπτικοποίηση της τοπολογίας σε έναν οριζόντιο άξονα.

Επίσης, στο Node Information Panel τώρα, εμφανίζονται το routing table, leaf set και neighborhood set του επιλεγμένου κόμβου.



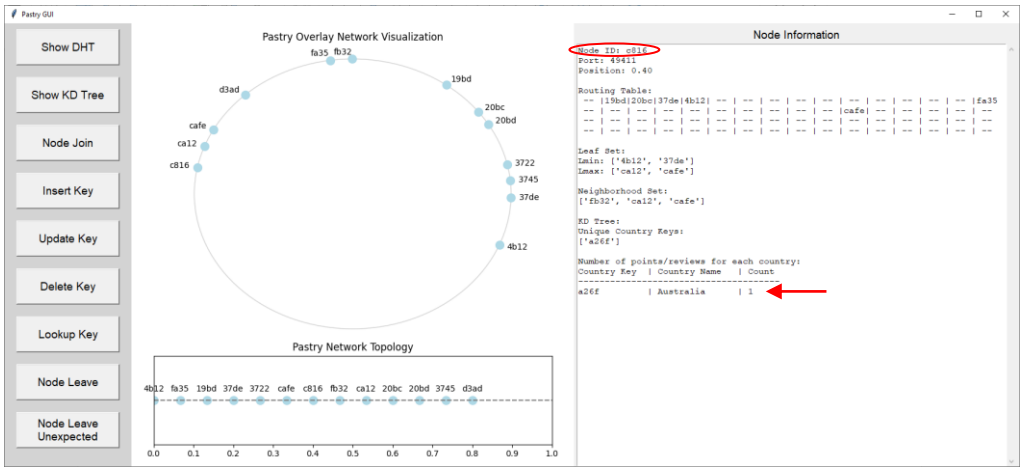
Κουμπί Node Leave:

Το κουμπί αυτό, αφαιρεί έναν κόμβο από το δίκτυο gracefully, δηλαδή οι κόμβοι που επηρεάζονται από την αποχώρηση ενημερώνουν τους πίνακές τους.

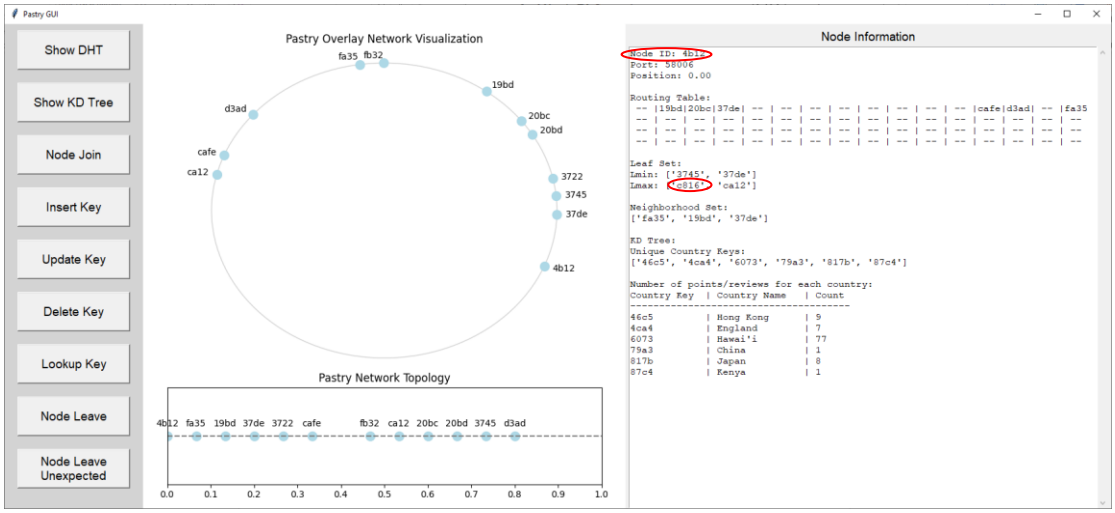
Κουμπί Node Leave Unexpected:

Όμοια με το Node Leave του Chord, αφαιρεί έναν κόμβο χωρίς να ενημερώσει το δίκτυο, προσομοιώνοντας την αποτυχία του κόμβου.

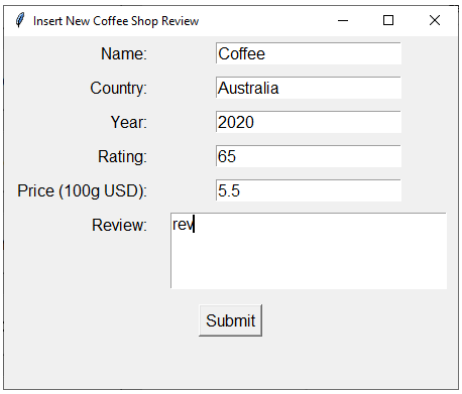
Για παράδειγμα αφαιρούμε τον κόμβο c816, που αποθηκεύει σημεία και reviews από την Αυστραλία με country key a26f.

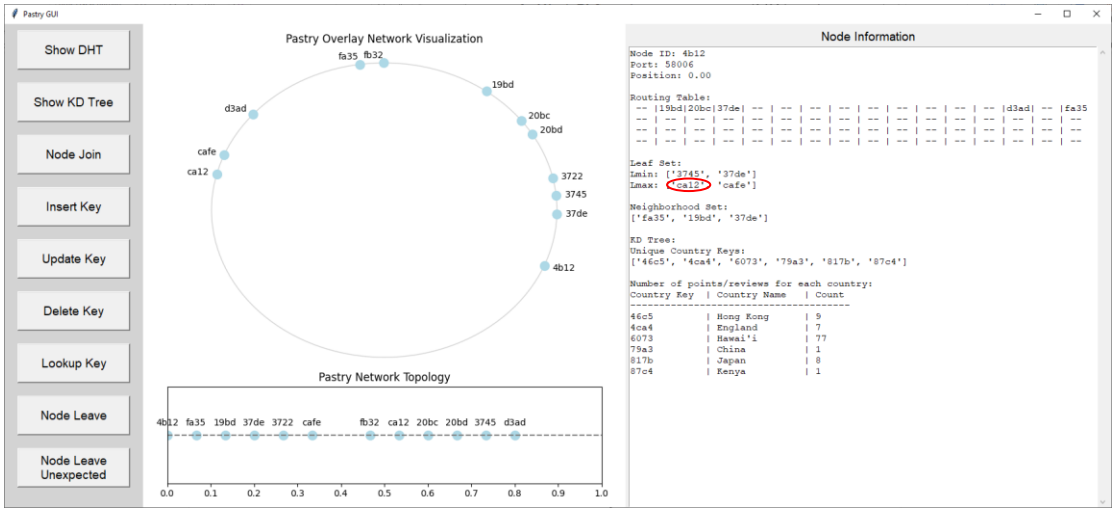


Μετά την αναχώρηση ελέγχουμε τον κόμβο 4b12 και παρατηρούμε, πως ο c816 εξακολουθεί να βρίσκεται στο Lmax του.

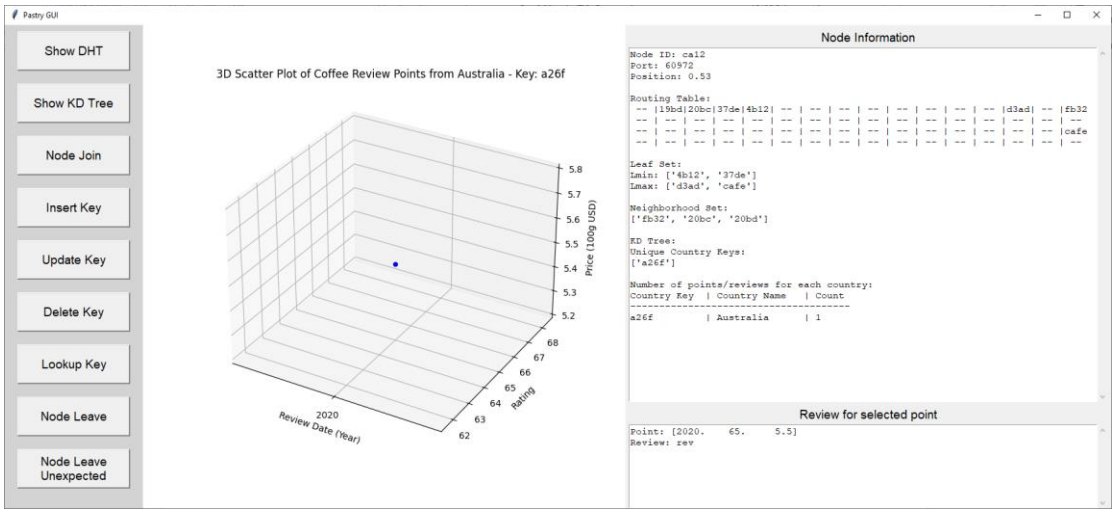


Ωστόσο εάν δοκιμάσουμε να κάνουμε Insert Key από τον 4b12, με το key να αντιστοιχεί στην Αυστραλία. Κατά την διαδικασία δρομολόγησης ο 4b12 θα διαπιστώσει πως ο κόμβος c816 δεν μπορεί να λάβει αιτήματα και θα διορθώσει το leaf set του, αντικαθιστώντας τον με τον ca12.





Το νέο κλειδί τελικά θα αποθηκευτεί στον κόμβο ca12.



1.2.2. Isolated Chord Execution

Multithreaded Test

Εκτελώντας:

```
Implementation-of-Basic-DHTs\Chord\Tests> python multithreaded_test.py
```

Δίνεται η δυνατότητα να αρχικοποιήσετε το Chord DHT και να εκτελεστούν παράλληλα operations πάνω στο δίκτυο. Το script δημιουργεί δύο threads που εκτελούν τα επιλεγμένα operations.

Τρόπος Χρήσης:

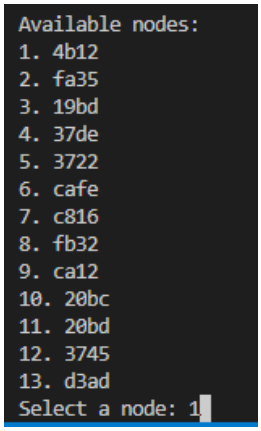
Αρχικά, ζητείται η επιλογή του operation για το Thread 1. Έστω ότι το Thread 1 πραγματοποιεί Insert Key.

```
Select operation for Thread 1:
1. Insert Key
2. Update Key
3. Delete Key
4. Lookup Key
5. Join Node
6. Leave Node
Enter your choice (1-6):
```

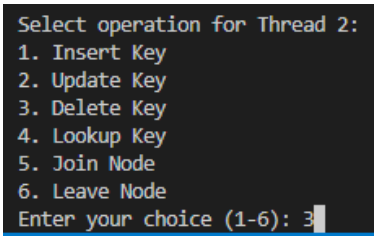
Έπειτα ανάλογα με την επιλογή, ζητούνται τα απαραίτητα δεδομένα.

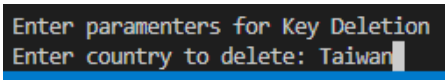
```
Enter paramenters for Key Insertion
Enter country: Taiwan
Enter name: Coffee
Enter year: 2025
Enter rating: 65
Enter price: 5.5
Enter review: review
```

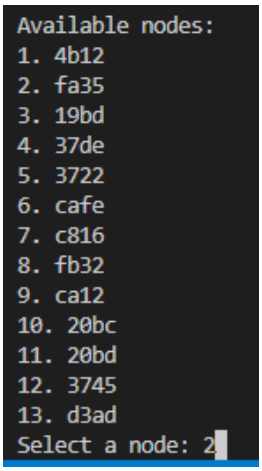
Τέλος, ζητείται η επιλογή του κόμβου που θα εκτελέσει το operation. Έστω ότι επιλέγουμε τον 4b12.



Στην συνέχεια επιλέγεται το operation του Thread 2. Έστω ότι κάνουμε Delete Key του κλειδιού που αντιστοιχεί στην χώρα “Taiwan” από τον κόμβο fa35.







Αποτελέσματα:

Τα δυο Threads ξεκινούν την εκτέλεση παράλληλα.

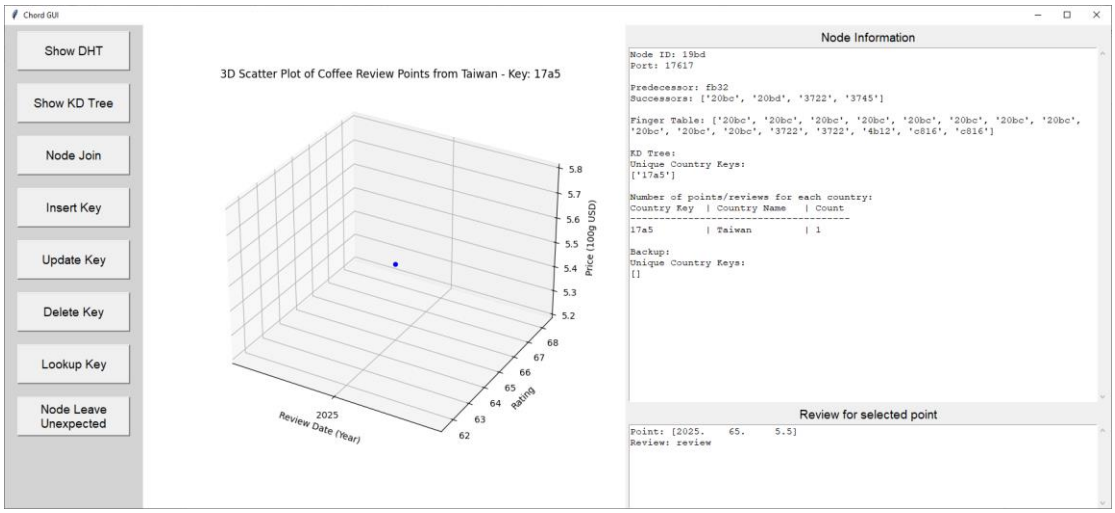
```
Starting concurrent operations...

[Thread 68360] Node 4b12: Inserting Key: 17a5, Country: Taiwan, Name: Coffee


[Thread 83756] Node fa35: Deleting Key: 17a5
```

Παρατηρούμε στην έξοδο, ότι και τα δύο operations ολοκληρώθηκαν επιτυχώς. Τελικά ο κόμβος 19bd, που αποθηκεύει το κλειδί 17a5 (Taiwan), περιέχει μόνο το κλειδί που εισήγαγε ο κόμβος 4b12.

```
[Thread 83756] Node fa35 Response: {'status': 'success', 'message': 'Deleted Key 17a5.', 'hops': 1}
[Thread 68360] Node 4b12 Response: {'status': 'success', 'message': 'Key 17a5 stored at 19bd', 'hops': 3}
```



Hops Test

Το test αφορά την πειραματική καταγραφή των hops που χρειάζεται το Chord για κάθε operation που μπορεί να εκτελεστεί.

Εντολή εκτέλεσης script:

```
Implementation-of-Basic-DHTs\Chord> python test.py
```

Αρχικά γίνεται εισαγωγή στο δίκτυο των προκαθορισμένων κόμβων (13 συνολικά), και για κάθε εισαγωγή κρατάμε τον αριθμό των hops. Στο τέλος όταν θα έχουν μπει όλοι οι κόμβοι στο δίκτυο υπολογίζουμε τον Μ.Ο. των hops που χρειάζεται για να εισαχθεί ένας κόμβος.

Έπειτα εξετάζεται ο αριθμός των βημάτων για την εισαγωγή κλειδιών. Εισάγονται στο δίκτυο όλα τα δεδομένα του dataset επομένως κάποιο κλειδί μπορεί να εισαχθεί περισσότερες από μια φορές. Όταν όλα τα κλειδιά εισαχθούν υπολογίζουμε ξανά τον Μ.Ο. hops που χρειάστηκαν.

```
-> Inserting Keys from Dataset...
-> End of Insertion
Average hops for Insertion: 1.9020866773675762
Sum of hops: 2370
Total num of Inserts: 1246
```

Για τις υπόλοιπες πράξεις update, lookup και delete βρίσκουμε από το dataset τις μοναδικές τιμές κλειδιών και εκτελούμε για αυτά τα κλειδιά την κάθε πράξη με τη σειρά όπως αναφέρονται. Πάλι στο τέλος υπολογίζεται ο M.O. hops για κάθε operation που εκτελέστηκε.

Για την διαδικασία διαγραφής κόμβου δεν γίνεται πείραμα διότι ο κάθε κόμβος είναι ανεξάρτητος και μπορεί να αποχωρήσει από το δίκτυο χωρίς να χρειαστεί να επικοινωνήσει με κάποιον άλλον κόμβο του δικτύου.

Τέλος για την καταμέτρηση των hops θεωρούμε ότι κάθε φορά που γίνεται κλήση στη συνάρτηση για εύρεση successor είναι και ένα hop, καθώς ο τρέχων κόμβος επικοινωνεί με κάποιων άλλο για την επίτευξη του operation.

1.2.3. Isolated Pastry Execution

Multithreaded Test

Εκτελώντας:

```
Implementation-of-Basic-DHTs\Pastry\Tests> pythom multithreaded_test.py
```

Μπορείτε να εκτελέσετε παράλληλα operations πάνω στο δίκτυο Pastry. Η λειτουργία αυτού του script είναι πανομοιότυπη με αυτή Chord/Tests/multithreaded_test.py.

Hops Test

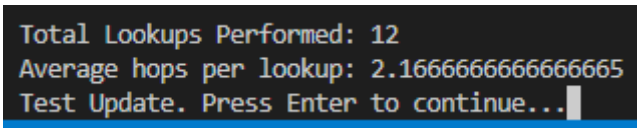
Εκτελώντας:

```
Implementation-of-Basic-DHTs\Pastry\Tests\Hops> python all_test.py
```

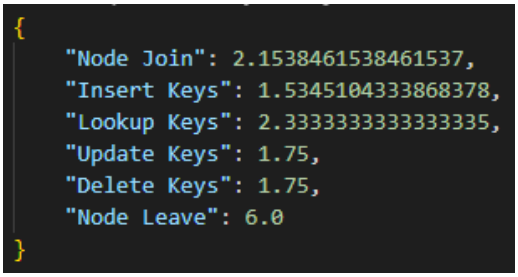
Πραγματοποιείται μια σειρά από tests πάνω στα operations: Node Join, Insert Keys, Lookup Keys, Update Keys, Delete Keys και Node Leave, με αυτή την σειρά. Κάθε ένα από αυτά τα operations εκτελούνται πολλαπλές φορές για όλους τους κόμβους του δικτύου στην περίπτωση του Join και Leave και για όλα τα κλειδιά από τυχαίο κόμβο κάθε φορά για τα υπόλοιπα operations.

Σε κάθε φάση ελέγχου αποτυπώνονται στην έξοδο ο μέσος όρος των hops για το operation και αποθηκεύονται στο Pastry/Tests/Hops/PastryResults.json.

Για παράδειγμα μετά τα Lookup Key Tests:



PastryResults.json μετά την ολοκλήρωση του script:



2. Distributed Environment Setup

2.1. Chord

Για την προσομοίωση του κατανεμημένου περιβάλλοντος έγινε χρήση των βιβλιοθηκών socket και threading. Κάθε κόμβος του δικτύου είναι αυτόνομος και επικοινωνεί με τους υπόλοιπους μέσω αιτημάτων με sockets. Κάθε κόμβος έχει μοναδικό socket και όλοι ανήκουν στο ίδιο δίκτυο 127.0.0.1 (localhost).

Επίσης ο κάθε κόμβος διαθέτει threadpool στο οποίο μπαίνουν οι διεργασίες και γίνεται δυναμική διαχείριση threads.

Κατά την εκκίνηση του κόμβου στο threadpool μπαίνουν τρεις διεργασίες, η μια αφορά την προσομοίωση του server που δέχεται και εξυπηρετεί αιτήματα και οι άλλες δύο διεργασίες που εκτελούνται περιοδικά για την ενημέρωση του finger table και successors list. Κάθε επιπλέον διεργασία για εξυπηρέτηση αιτήματος προστίθεται στο threadpool.

Τέλος γίνεται χρήση thread lock σε σημεία που υπάρχει race condition, για παράδειγμα στην ενημέρωση των τιμών ενός κλειδιού στο kdtree .

2.2. Pastry

Η προσομοίωση του αποκεντρωμένου περιβάλλοντος του Pastry DHT, για την παράλληλη λειτουργία των κόμβων, επιτυγχάνεται μέσω την χρήση των βιβλιοθηκών threading και socket.

Συγκεκριμένα, κάθε κόμβος διαθέτει ένα deamon thread για την συνεχή εκτέλεση της μεθόδου _server και ένα threadpool, στο οποίο ο server υποβάλει τα tasks εξυπηρέτησης των εισερχόμενων αιτημάτων. Φυσικά σε κρίσιμα σημεία των μεθόδων εξυπηρέτησης αιτημάτων έχει τοποθετηθεί threading.Lock, για την αποτροπή race conditions.

Η επικοινωνία μεταξύ των κόμβων γίνεται μέσω TCP Sockets. Κάθε κόμβος διαθέτει ένα μοναδικό port, ενώ όλοι είναι bound στην IP διεύθυνση 127.0.0.1.

3. Implementation

3.1. Dataset

Το dataset που χρησιμοποιήθηκε στο project για την εισαγωγή δεδομένων στα DHTs είναι το simplified_coffee.csv. Περιέχει πληροφορίες για coffee shops σε διάφορες χώρες.

Στην υλοποίηση μας, δημιουργείται από ένα KD-Tree σε κάθε κόμβο, οπού αποθηκεύονται 3D σημεία της μορφής [review_data, rating, price_per_100g], αντλώντας τα δεδομένα από τις αντίστοιχες στήλες του csv. Συμπληρωματικά αποθηκεύουμε στο KDTree class τα reviews των σημείων και χρησιμοποιούμε το SHA1 hash της χώρα (από την στήλη loc_country), για την δημιουργία κλειδιών.

3.2. KD - Tree

Κάθε κόμβος των δικτύων Chord και Pastry διαθέτουν ένα κεντροποιημένο KD-Tree για την αποθήκευση κλειδιών και των αντίστοιχων δεδομένων. Το KD-Tree αυτό ορίζεται ως ένα instance της κλάσης **KDTree**.

Η κλάση αυτή διαθέτει τα εξής attributes:

```
self.tree = None  # Το ίδιο το δέντρο (instance της κλάσης KDTree της βιβλιοθήκης sklearn)
self.points = points  # Numpy array με τα points του δέντρου
self.reviews = reviews  # Numpy array με τα reviews των points
self.country_keys = country_keys  # Numpy array με τα keys που αντιστοιχούν στις χώρες
self.countries = countries if countries is not None else []  # Numpy array με τις χώρες των points
```

Η κλάση περιλαμβάνει επίσης, μεθόδους για την κατασκευή του δέντρου (**build**), προσθήκη σημείου (**add_point**), διαγραφή σημείων (**delete_points**), ενημέρωση σημείων (**update_points**), range search (**search**) και οπτικοποίηση των σημείων σε ένα 3D scatter plot (**visualize**).

Εξήγηση Μεθόδων

build(self, points):

Κατασκευάζει το KD-Tree χρησιμοποιώντας την κλάση KDTree της βιβλιοθήκης sklearn.neighbors.

add_point(self, new_point, new_review, new_country):

Προσθέτει ένα νέο σημείο, review country και country_key στις λίστες points, reviews, countries και country_keys. Στο τέλος καλεί την μέθοδο build για την ανακατασκευή του δέντρου.

delete_points(self, country_key):

Διαγράφει όλα τα points, reviews, countries που αντιστοιχούν στο δοθέν country_key.

update_points(self, country_key=None, criteria=None, update_fields=None):

Ενημερώνει τα points και reviews για ένα country_key με βάση ευέλικτα κριτήρια.

Το όρισμα update_fields, ορίζει τις τιμές που θα αποκτήσουν τα ενημερωμένα σημεία και reviews:

```
update_fields = {
   “point”: [new_date, new_rating, new_price],
   “attributes”: {
      “review_date”: new_date,
      “rating”: new_rating,
      “price”: new_price
   }
   “review”: new_review
}
```

Εάν επιθυμούμε να ενημερώσουμε όλους τους άξονες των σημείων, τότε χρησιμοποιείται το κλειδί “point”, διαφορετικά χρησιμοποιείται το “attributes” με τις κατάλληλες τιμές στους άξονες που επιθυμούμε να τροποποιήσουμε. Επιπλέον το κλειδί “review” ορίζει το ενημερωμένο review για τα σημεία και μπορεί να μην εισαχθεί για να παραμείνουν αμετάβλητα τα reviews.

Το όρισμα criteria, επιτρέπει φιλτράρισμα των σημείων που θα ενημερωθούν βάση συγκεκριμένων χαρακτηριστικών:

```
criteria = {
    “review_date”: year_criteria,
    “rating”: rating_criteria,
    “price”: price_criteria
}
```

Το όρισμα μπορεί να μην εισαχθεί για να ενημερωθούν όλα τα points / reviews για το δοθέν κλειδί.

search(self, country_key, lower_bounds, upper_bounds):

Πραγματοποιεί range search σε όλα τα σημεία που αντιστοιχούν στο country_key και βρίσκονται εντός των δοθέντων bounds.

Τα bounds έχουν της εξής μορφή:

```
lower_bounds = [year_lower, rating_lower, price_lower]
upper_bounds = [year_ upper, rating_ upper, price_upper]
```

Πρέπει πάντα να εισαχθούν bounds για έναν τουλάχιστον άξονα, όμως μπορούν οι υπόλοιποι να είναι None. Σε αυτή την περίπτωση θα αντικατασταθούν με την ελάχιστη και μέγιστη τιμή του κάθε άξονα.

Για το range search χρησιμοποιείται η μέθοδος query_radius του tree object.

```
center = [(lower_bounds[i] + upper_bounds[i]) / 2 for i in range(3)]
radius = np.linalg.norm((np.array(upper_bounds) - np.array(lower_bounds)) / 2)

indices = self.tree.query_radius([center], r=radius + 1e-8)[0]
```

visualize(self, ax, canvas, points=None, reviews=None, country_key=None, country=None, title=None):

Αυτή η μέθοδος καλείται από το GUI για την οπτικοποίηση των σημείων που αντιστοιχούν στο δοθέν country_key σε ένα 3D scatter plot. Συμπληρωματικά μέσω του on_pick event handler, παρέχεται η δυνατότητα επιλογής ενός σημείου στο plot και εμφάνισης του αντίστοιχου review στο GUI.

3.3. LSH

Κατά την εκτέλεση του Lookup Key operation όταν το kd_tree object επιστρέψει τα points και reviews του εύρους αναζήτησης, αν ο αριθμός των reviews δεν είναι μηδενικός, εκτελείται ο μηχανισμός LSH.

Η υλοποίηση του LSH βρίσκεται στην κλάση LSH του αρχείου Multidimensional_Data_Structures/lsh.py.

Η κλάση αυτή διαθέτει τα εξής attributes:

```
self.num_bands = num_bands  # Αριθμός των LSH bands
self.num_rows = num_rows  # Αριθμός των rows σε κάθε band
self.hash_tables = [{} for _ in range(num_bands)]  # Λίστα των hash tables (ένα για κάθε band)
self.documents = []  # Λίστα με όλα τα review vectors
```

Η κλάση περιλαμβάνει επίσης, μεθόδους για την προσθήκη vectors στο LSH index (**add_document**), εύρεση των N πιο όμοιων document pairs (**find_similar_pairs**) και επιστροφή των N πιο όμοιων documents συνολικά (**find_similar_docs**).

find_similar_pairs(self, N=5):

Η μέθοδος αυτή αναζητά documents στα hash tables, που βρίσκονται σε κοινά buckets και επιστρέφει τα N ζεύγη με την μεγαλύτερη ομοιότητα, βάση της μετρικής cosine similarity.

find_similar_docs(self, similar_pairs, original_docs_texts, N):

Δημιουργεί ένα σύνολο με documents, από τα ζεύγη που επέστρεψε η παραπάνω συνάρτηση και επιστρέφει τα πρώτα N.

3.4. Constants and Helper Functions

Στο root directory του project βρίσκονται τα αρχεία constants.py και helper_functions.py, τα οποία περιέχουν σταθερές και υλοποιήσεις βοηθητικών συναρτήσεων για τα δύο DHTs.

Κοινά Constants:

Οι κοινές σταθερές είναι τα **predefined_ids**, που είναι μια λίστα με 13 ID κόμβων, που χρησιμοποιείται για την αρχικοποίηση του ίδιου DHT σε κάθε εκτέλεση και το **HASH_HEX_DIGITS**, που ορίζει τον αριθμό των δεκαεξαδικών ψηφίων των IDs.

Chord Constants:

Περιέχει σταθερές που ορίζουν την δομή δικτύου Chord και βοηθούν στο testing. Το ότι η σταθερά S είναι 4, παρόλο που κάθε κόμβος έχει έναν successor θεωρητικά, αποτελεί σχεδιαστική απόφαση και αποδίδεται στον μηχανισμό αντικατάστασης του αμέσου successor, εάν αυτός βγει απρόοπτα από το δίκτυο.

```
M = 16  # Bit size for the hash space (e.g. if M=4, 2^4 = 16 nodes)
R = 2**M  # Max possible number of nodes in the network
S = 4

# Operations for testing
chord_operations = ["Node Join", "Insert Keys", "Delete Keys", "Update Keys", "Lookup Keys"]
```

Pastry Constants:

Περιέχει σταθερές που ορίζουν την δομή των κόμβων του δικτύου Pastry και βοηθούν στο testing.

```
b = 4  # Routing Base. 2^b = Number of posssible entries in each row of the routing table
r = 2  # Replication Factor
L = 2 * r  # Number of Nodes to store in the Leaf Set

NEIGHBORHOOD_SIZE = 3 # Number of nodes to store in the neighbourhood set

main_operations = ["NODE_JOIN", "NODE_LEAVE", "INSERT_KEY", "LOOKUP",
"UPDATE_KEY", "DELETE_KEY"]

# Operations for testing
pastry_operations = ["Node Join", "Node Leave", "Insert Keys", "Delete Keys", "Update Keys",
"Lookup Keys"]
```

Helper Functions:

Οι συναρτήσεις του helper_functions.py υποστηρίζουν τις υλοποιήσεις των Chord και Pastry, εκτελώντας βασικές λειτουργίες όπως key hashing (**hash_key**), μετατροπή ακεραίου σε δεκαεξαδικό (**int_to_hex**), υπολογισμό αποστάσεων (**topological_distance**, **hex_distance**, **distance**), εύρεση κοινού προθέματος (**common_prefix_length**) και σύγκριση ids κόμβων (**hex_compare**).

3.5. Chord DHT

3.5.1. Chord Network

Το αρχείο **network.py** ορίζει την κλάση ChordNetwork, η οποία είναι υπεύθυνη για τη διαχείριση του δικτύου των κόμβων Chord.

Η κλάση ChordNetwork είναι το κέντρο του δικτύου. Διαχειρίζεται τους κόμβους, τις διευθύνσεις τους και φροντίζει να λειτουργούν όλα σωστά. Βοηθά τους κόμβους να επικοινωνούν μεταξύ τους, να μοιράζονται δεδομένα και να προσαρμόζονται όταν κάποιοι μπαίνουν ή βγαίνουν από το δίκτυο.

Η ChordNetwork κρατάει μια λίστα με όλους τους κόμβους και τις διευθύνσεις τους. Αυτό είναι σημαντικό γιατί έτσι το δίκτυο και κατά αντιστοιχία όλοι οι κόμβοι γνωρίζουν σε ποιο port να δρομολογήσουν ένα μήνυμα βάση του επιθυμητού ID.

```
self.nodes = {}  # Dictionary. Keys are node IDs, values are Node objects
self.used_ports = []
self.gui = ChordDashboard(self, main_window)
```

Για να λειτουργεί βέλτιστα το Chord, οι κόμβοι πρέπει να είναι κατανεμημένοι ομοιόμορφα.

Το ChordNetwork συνδέεται με το γραφικό περιβάλλον (ChordDashboard) για την εύκολη διαχείριση του δικτύου. Συμπληρωματικά, το γραφικό περιβάλλον βοηθά στο debugging και στην παρακολούθηση των αποτελεσμάτων των operations:

self.gui = ChordDashboard(self, main_window) # Σύνδεση με το GUI

build: Η μέθοδος build() στο ChordNetwork είναι υπεύθυνη για τη δημιουργία και αρχικοποίηση ενός δικτύου Chord. Παρέχει δύο τρόπους κατασκευής του δικτύου:

Χρήση προκαθορισμένων (predefined) Node IDs, όπου οι κόμβοι δημιουργούνται με συγκεκριμένα αναγνωριστικά.

Αν δοθεί η λίστα predefined_ids, τότε δημιουργούνται κόμβοι με προκαθορισμένα **Node IDs**.

Διαφορετικά, το δίκτυο αρχικοποιείται με node_num random κομβους.

3.5.2. Chord Node

Το αρχείο **node.py** ορίζει την κλάση ChordNode, η οποία αναπαριστά έναν κόμβο μέσα στο δίκτυο Chord. Κάθε κόμβος είναι υπεύθυνος για την αποθήκευση δεδομένων, τη δρομολόγηση μηνυμάτων και τη διατήρηση πληροφοριών για τους γείτονές του.

Η ChordNode είναι η βασική μονάδα του δικτύου Chord. Κάθε κόμβος έχει ένα μοναδικό ID και δομές δεδομένων για να επικοινωνεί με άλλους κόμβους.

Κάθε κόμβος έχει ένα μοναδικό **ID**, το οποίο μπορεί να δημιουργηθεί τυχαία ή να καθοριστεί κατά την προσθήκη του στο δίκτυο. Επίσης, κάθε κόμβος εκτελείται σε μία μοναδική **θύρα** (**port**) στο localhost για επικοινωνία μέσω δικτύου.

```
self.node_id = node_id if node_id is not None else self._generate_id(self.port)
self.port = self._generate_port()  # Θύρα επικοινωνίας
```

Κάθε κόμβος διαθέτει διαφορετικές δομές που τον βοηθούν στη γρήγορη δρομολόγηση των μηνυμάτων:

Finger Table: Ένας πίνακας όπου κάθε κόμβος αποθηκεύει ID άλλων κόμβων για γρήγορη αναζήτηση. Ο πίνακας έχει μέγεθος ίσο με τον αριθμό των ψηφίων των κλειδιών (M). Το id σε κάθε θέση είναι ο successor του κλειδιού που προκύπτει από τον τύπο (current_node_id + 2^i)mod2^M. Επομένως ο τρέχων κόμβος έχει την πληροφορία για το ποιος είναι ο successor αυτών των κλειδιών

Successors List: Διατηρεί τους S επόμενους κόμβους ώστε σε περίπτωση που ο successor του φύγει να μπορεί να βρει τον επόμενο και το δίκτυο να μην “πέσει”.

Predecessor: Αποθηκεύει τον πίσω κόμβο του.

Κάθε κόμβος διαθέτει ένα **KD-Tree**, που επιτρέπει την αποδοτική αναζήτηση και αποθήκευση δεδομένων. Χρησιμοποιείται για την οργάνωση των δεδομένων του κόμβου και την εκτέλεση χωρικών ερωτημάτων.

```
self.kd_tree = None  # Κεντρική δομή KD-Tree για αποθήκευση δεδομένων
```

Επίσης κάθε κόμβος κρατάει back up το kdtree του predecessor του έτσι ώστε εάν αντιληφθεί ο successor του κόμβου που έφυγε ότι δεν είναι ενεργός να ειδοποιήσει τον νέο του successor ότι άλλαξε ο predecessor του και να συγχωνεύσει το backup του κόμβου που έφυγε με το kdtree του και τέλος να φτιάξει νέο backup για τον νέο του predecessor.

Για κάθε κόμβο υπάρχει μια boolean μεταβλητή που δηλώνει την κατάστασή του, ως true για active node και false για inactive node.

Τέλος κάθε κόμβος έχει αντικείμενο lock για να αποτρέπει τα race conditions, ένα αντικείμενο event που όταν ενεργοποιείται σταματούν όλα τα threads του κόμβου και ένα threadpool για δυναμική διαχείριση των διαθέσιμων thread κάθε κόμβου.

3.5.3. Node Join

Για την επεξήγηση της μεθόδου join θα πρέπει να αναλυθούν και οι μέθοδοι closest_preceding_node, _handle_find_successor οι οποίες χρησιμοποιούνται και σε άλλες μεθόδους.

Η closest_preceding_node ψάχνει τον προηγούμενο, πιο κοντινό κόμβο ενός κλειδιού στο finger table του κόμβου που την καλεί. Ξεκινάει από το τέλος του finger table και πηγαίνει προς την αρχή. Αν βρει κάποιον κόμβο ο οποίος να είναι ενεργός και να προηγείται του κλειδιού, τον επιστρέφει, αλλιώς επιστρέφει τον τελευταίο κόμβο.

Η _handle_find_successor ψάχνει για τον ακριβώς επόμενο κόμβο ενός κλειδιού. Έτσι αν ο κόμβος που ψάχνει αυτόν τον successor έχει ίδιο id με το key, τότε επιστρέφει το id του κόμβου. Αν η απόσταση (στον δακτύλιο) του κλειδιού από τον κόμβο που καλεί την μέθοδο είναι μικρότερη από την απόσταση του κλειδιού από τον successor του κόμβου που καλεί την μέθοδο, τότε επιστρέφεται το id του κόμβου που καλεί την μέθοδο. Διαφορετικά, ψάχνει για τον κοντινότερο προηγούμενο κόμβο του κλειδιού και η μέθοδος εκτελείται σε αυτόν αναδρομικά.

-> Η μεθοδος join χρησιμοποιείται όταν ένας κόμβος θέλει να μπει στον δακτύλιο Chord.

Παίρνει ως όρισμα το id του successor (και τον ίδιο τον κόμβο). Μετά, κάνει τον predecessor του successor και τον successor του predecessor να δείχνουν στον κόμβο που κάνει join. Ύστερα, ενημερώνει το πρώτο στοιχείο του finger table και θέτει τους successors του και τον predecessor του.
Αν ο successor του κόμβου έχει kd-tree τότε παίρνει κάποια από τα κλειδιά του successor του και τα βάζει στο δικό του kd-tree αφαιρώντας τα από το kd-tree του successor. Επίσης, ενημερώνει το backup του successor του και το δικό του.

Με λίγα λόγια, η μέθοδος αυτή χειρίζεται όλη τη διαδικασία ένταξης ενός νέου κόμβου στο δακτύλιο, δηλαδή την ενημέρωση των successors, predecessor, finger table, της αναδιανομής των δεδομένων και των backup.

Επίσης, οι μέθοδοι update_finger_table, update_successors_on_join, update_successors_on_leave καλούνται τακτικά για την ενημέρωση του finger table και των successors των κόμβων σε περίπτωση που μπει ή βγει κάποιος κόμβος από το δίκτυο.

Η update_finger_table ενημερώνει την πρώτη θέση του finger table με τον successor του κόμβου. Μετά, ενημερώνει τις άλλες θέσεις βρίσκοντας τον successor του κλειδιού "keyId + 2^i % R".

Η update_successors_on_join ελέγχει αν έχει μπει κάποιος κόμβος στο δίκτυο και αν έχει μπει, ενημερώνει τους successors και καλεί τον εαυτό της αναδρομικά.

Η update_successors_on_leave ελέγχει αν έχει φύγει κάποιος κόμβος και αν έχει φύγει ενημερώνει τους successors και καλεί τον εαυτό της αναδρομικά.

3.5.4. Key Insert

Στην εισαγωγή κλειδιού η είσοδος έχει την εξής μορφή: key, point, review, country

Η εισαγωγή γίνεται με βάση το πεδίο country το οποίο μας δίνει την τιμή key με την συνάρτηση κατακερματισμού. Το όρισμα points είναι μια 3αδα τιμών που αποθηκεύεται στο kdtree και το όρισμα review είναι ένα string το οποίο χρησιμοποιείται για lsh.

Για να γίνει η εισαγωγή επιλέγεται τυχαία ένας κόμβος του δικτύου. Ο κόμβος αυτός αναλαμβάνει να βρει τον successor του κλειδιού καλώντας τη συνάρτηση handle_find_successor. Εάν ο ίδιος κόμβος που κάνει την εισαγωγή δεν έχει id ίσο με το κλειδί αλλά ούτε και ο successor του κόμβου εισαγωγής δεν είναι successor του κλειδιού, τότε καλείται η συνάρτηση closest_preceding_node για να βρει από το finger table του κόμβου τον κοντινότερο preceding node του κλειδιού. Στον κόμβο που επέστρεψε η συνάρτηση στέλνεται μήνυμα από τον τρέχων κόμβο και αναλαμβάνει αυτός πλέον να βρει τον successor του κλειδιού. Η διαδικασία επαναλαμβάνεται αναδρομικά μέχρι είτε το κλειδί να είναι ίσο με τον κόμβο είτε ο successor του κόμβου είναι και του κλειδιού. Όταν επιστραφεί στον αρχικό κόμβο εισαγωγής το id του κόμβου που είναι ο successor στέλνεται μήνυμα σε αυτό τον κόμβο με τα ορίσματα και το κλειδί ώστε να ενημερωθεί.

Ο κόμβος θα λάβει την πληροφορία και θα πρέπει να την αποθηκεύσει στο kdtree. Αν δεν έχει δημιουργεί και κάνει εισαγωγή διαφορετικά απλά εισάγει την νέα πληροφορία.

Τέλος θα καλέσει τη συνάρτηση request_backup_update για να ενημερώσει ο successor του το backup. Η ενημέρωση του backup είναι η ίδια διαδικασία με την εισαγωγή του κλειδιού. Θα σταλεί μήνυμα στον successor με την πληροφορία και αυτός θα την εισάγει στο kdtree backup που έχει.

3.5.5. Key Update

Για την διαδικασία του update το πρόγραμμα παίρνει την εξής είσοδο: key, updated_data, criteria.

Το key είναι το hashed country και έχει γίνει insert στο δίκτυο με κάποιες 3αδες τιμών αποθηκευμένες στο kdtree και ένα review για κάθε 3αδα τιμών αντίστοιχα. Επομένως ένα κλειδί μπορεί να έχει περισσότερες από 1 εισαγωγές κάνει στο δίκτυο.

Το όρισμα updated_data περιέχει τα πεδία που θα ανανεωθούν και τις νέες τιμές τους. Στην 3αδα τιμών η πρώτη αντιστοιχεί στο "review_date", η δεύτερη στο "rating" και η τρίτη στο "100g_USD".

Στο όρισμα criteria ορίζονται πάλι τιμές όπως στο updated_data έτσι ώστε να γίνει στοχευμένο update στις εγγραφές με τις συγκεκριμένες τιμές.

Η διαδικασία στη συνέχεια είναι ίδια με της εισαγωγής κλειδιού. Η πρώτος τυχαίος κόμβος που αναλαμβάνει την εισαγωγή ξεκινά την αναζήτηση για τον successor του κλειδιού και μόλις βρεθεί στέλνεται αίτημα για την ενημέρωση. Ο successor ενημερώνει τα πεδία στο kdtree που ζητάει το αίτημα και αναλαμβάνει να στείλει μήνυμα και στον successor του για να ενημερώσει το backup του.

3.5.6. Key Lookup

Στην αναζήτηση κλειδιού δίνεται όρισμα ένα κλειδί, ανώτατο και κατώτατο όριο (είναι 3αδες τιμών) και μία σταθερά N για το lsh για το πόσα κείμενα να επιστρέψει.

Η διαδικασία ξεκινά από ένα τυχαίο κόμβο του δικτύου και αναζητείται ο successor του κλειδιού που δόθηκε με τη διαδικασία που ακολουθήθηκε και στην εισαγωγή κλειδιού. Όταν βρεθεί ο successor θα ελέγξει εάν στο kdtree του υπάρχει καταχώρηση από αυτό το κλειδί και αν υπάρχει θα εκτελέσει αναζήτηση με βάση τα διαστήματα τιμών που δόθηκαν.

Τα επιστρεφόμενα δεδομένα θα φιλτραριστούν με βάση το κλειδί και σε αυτά τα αποτελέσματα θα εφαρμοστεί για το review lsh. Από το lsh θα επιστραφούν τα N πιο όμοια reviews.

3.5.7. Key Delete

Για να γίνει διαγραφή κλειδιού θα πρέπει να δοθεί ένα κλειδί και από ένα τυχαίο κόμβο του δικτύου θα αναζητηθεί ο successor αυτού του κλειδιού.

Ο κόμβος που είναι ο successor του θα λάβει μήνυμα με το κλειδί και το operation που πρέπει να εκτελέσει δηλ. delete_key και αν υπάρχουν στο kdtree εγγραφές από αυτό το κλειδί τότε θα πρέπει να διαγράψει όλες τις εγγραφές αυτού του κλειδιού από το kdtree και τα reviews και μετά θα στείλει αίτημα στον successor του έτσι ώστε να διαγράψει και αυτός τις πληροφορίες για το συγκεκριμένο κλειδί από το backup.

3.5.8. Node Leave

Η leave προσομοιώνει την απότομη αποχώρηση ενός κόμβου. Θέτει την μεταβλητή running του κόμβου σε False για να δηλώσει ότι ο κόμβος έφυγε από τον δακτύλιο και σταματάει το thread που τρέχει τον κόμβο.

3.6. Pastry DHT

Το Pastry είναι ένα αποκεντρωμένο πρωτόκολλο που υλοποιεί μια διανεμημένη δομή κατακερματισμού (Distributed Hash Table – DHT) και επιτρέπει την αποθήκευση και ανάκτηση δεδομένων σε ένα peer-to-peer δίκτυο μεγάλης κλίμακας. Μέσω του Pastry, κάθε κόμβος στο δίκτυο αναλαμβάνει μια μοναδική ταυτότητας (ID), που προκύπτει από έναν

κατακερματισμό (hash) και βοηθά στην ομοιόμορφη κατανομή των δεδομένων. Με αυτόν τον τρόπο, τα δεδομένα κατανέμονται ομοιόμορφα στο δίκτυο, ενώ κάθε μήνυμα μπορεί να δρομολογηθεί με αποδοτικότητα σε περίπου $log_{2^b} N$ βήματα, όπου το N είναι ο αριθμός των κόμβων και 2^b το σύνολο των πιθανών συνδυασμών IDs.

Επιπλέον, το Pastry χρησιμοποιεί μηχανισμούς όπως το leaf set και το routing table, ώστε κάθε κόμβος να έχει πρόσβαση στους πιο κοντινούς γείτονές του βάση αριθμητικής απόσταση να δρομολογεί μηνύματα με βάση το κοινό πρόθεμα των ID. Αυτοί οι πίνακες δρομολόγησης διαμορφώνονται δυναμικά καθώς νέοι κόμβοι εισέρχονται στο δίκτυο ή υπάρξουν αποχωρήσεις, εξασφαλίζοντας έτσι την ανθεκτικότητα και τη συνέπεια της δομής.

Με το Pastry, το δίκτυο μπορεί να ανταπεξέλθει σε αλλαγές όπως απροσδόκητες αποτυχίες κόμβων ή την είσοδο νέων κόμβων χωρίς να διαταράσσεται η συνολική λειτουργικότητα του συστήματος. Αυτή η δυναμική προσαρμοστικότητα το καθιστά ιδανικό για κατανεμημένες εφαρμογές όπου απαιτείται υψηλή διαθεσιμότητα και αξιοπιστία.

3.6.1 Pastry Network

Το αρχείο **network.py** ορίζει την κλάση PastryNetwork, η οποία είναι υπεύθυνη για τη διαχείριση του δικτύου των κόμβων Pastry.

Η κλάση PastryNetwork είναι το κέντρο του δικτύου. Διαχειρίζεται τους κόμβους, τις διευθύνσεις τους και φροντίζει να λειτουργούν όλα σωστά. Βοηθά τους κόμβους να επικοινωνούν μεταξύ τους, να μοιράζονται δεδομένα και να προσαρμόζονται όταν κάποιοι μπαίνουν ή βγαίνουν από το δίκτυο.

Η PastryNetwork κρατάει μια λίστα με όλους τους κόμβους και τις διευθύνσεις τους. Αυτό είναι σημαντικό γιατί έτσι το δίκτυο και κατά αντιστοιχία όλοι οι κόμβοι γνωρίζουν σε ποιο port να δρομολογήσουν ένα μήνυμα βάση του επιθυμητού ID.

```
self.nodes = {}  # Όλοι οι κόμβοι
self.node_ports = {}  # Αντιστοίχιση ID κόμβου σε ports
self.used_ports = []  # Λίστα με τις θύρες που έχουν ήδη χρησιμοποιηθεί
```

Οι κόμβοι είναι κατανεμημένοι ομοιόμορφα τοπολογικά. Η κλάση PastryNetwork δημιουργεί θέσεις κόμβων μεταξύ 0 και 1, ώστε το δίκτυο να έχει καλή κατανομή και ισορροπία.

```
EVENLY_SPACED_NODES = 16  # Αρχικοί κόμβοι
self.positions = np.linspace(0, 1, EVENLY_SPACED_NODES)  # Θέσεις από 0 έως 1
self.used_positions = list(self.positions)  # Θέσεις που έχουν ήδη χρησιμοποιηθεί
```

Το PastryNetwork συνδέεται με το γραφικό περιβάλλον (PastryDashboard) για την εύκολη διαχείριση του δικτύου. Συμπληρωματικά, το γραφικό περιβάλλον βοηθά στο debugging και στην παρακολούθηση των αποτελεσμάτων των operations:

```
self.gui = PastryDashboard(self, main_window=main_window)  # Σύνδεση με το GUI
```

build: Η μέθοδος build() στο PastryNetwork είναι υπεύθυνη για τη δημιουργία και αρχικοποίηση ενός δικτύου Pastry. Παρέχει δύο τρόπους κατασκευής του δικτύου:

Χρήση προκαθορισμένων (predefined) Node IDs, όπου οι κόμβοι δημιουργούνται με συγκεκριμένα αναγνωριστικά.

Χρήση τυχαίου αριθμού κόμβων (random nodes), όπου το δίκτυο δημιουργείται δυναμικά με έναν καθορισμένο αριθμό κόμβων.

Αν δοθεί η λίστα predefined_ids, τότε δημιουργούνται κόμβοι με προκαθορισμένα **Node IDs**.

Αντίθετα, αν καθοριστεί το node_num, το δίκτυο δημιουργεί τον αντίστοιχο αριθμό κόμβων με τυχαία **Node IDs**.

3.6.2 Pastry Node

Το αρχείο **node.py** ορίζει την κλάση PastryNode, η οποία αναπαριστά έναν κόμβο μέσα στο δίκτυο Pastry. Κάθε κόμβος είναι υπεύθυνος για την αποθήκευση δεδομένων, τη δρομολόγηση μηνυμάτων και τη διατήρηση πληροφοριών για τους γείτονές του.

Η PastryNode είναι η βασική μονάδα του δικτύου Pastry. Κάθε κόμβος έχει ένα μοναδικό ID, μία τοπολογική θέση και δομές δεδομένων που του επιτρέπουν να επικοινωνεί με άλλους κόμβους και να συμμετέχει στη διαχείριση του κατανεμημένου πίνακα κατακερματισμού (DHT).

Κάθε κόμβος έχει ένα μοναδικό **ID**, το οποίο μπορεί να δημιουργηθεί τυχαία ή να καθοριστεί κατά την προσθήκη του στο δίκτυο. Επίσης, κάθε κόμβος εκτελείται σε μία μοναδική **θύρα** (**port**) για επικοινωνία μέσω δικτύου.

```
self.node_id = node_id if node_id is not None else self._generate_id(self.port)
self.port = self._generate_port()  # Θύρα επικοινωνίας
```

Κάθε κόμβος λαμβάνει μια **θέση** στο εύρος **[0,1]**, ώστε να φαίνονται ευδιάκριτα στην οπτικοποίηση.

```
self.position = None  # Η θέση δημιουργείται δυναμικά από το δίκτυο
```

Κάθε κόμβος διαθέτει διαφορετικές δομές που τον βοηθούν στη γρήγορη δρομολόγηση των μηνυμάτων:

Routing Table: Ένας πίνακας 2D όπου κάθε κόμβος αποθηκεύει ID άλλων κόμβων για γρήγορη αναζήτηση.

Leaf Set: Διατηρεί τους πιο κοντινούς κόμβους σε αριθμητική απόσταση από τον κόμβο, ώστε να μπορεί να βρει την καλύτερη διαδρομή για κάποιο μήνυμα.

Neighborhood Set: Περιέχει γειτονικούς κόμβους με βάση την τοπολογική θέση.

```
self.routing_table = [[None for j in range(pow(2, b))] for i in range(HASH_HEX_DIGITS)]
self.Lmin = [None for x in range(L // 2)]
self.Lmax = [None for x in range(L // 2)]
self.neighborhood_set = [None for x in range(M)]
```

Κάθε κόμβος διαθέτει ένα **KD-Tree**, που επιτρέπει την αποδοτική αναζήτηση και αποθήκευση δεδομένων. Χρησιμοποιείται για την οργάνωση των δεδομένων του κόμβου και την εκτέλεση χωρικών ερωτημάτων.

```
self.kd_tree = None  # Κεντρική δομή KD-Tree για αποθήκευση δεδομένων
```

3.6.3. Node Join

Η διαδικασία εισόδου ενός νέου κόμβου στο δίκτυο Pastry είναι σημαντική στη συνοχή και την ισορροπημένη κατανομή των δεδομένων. Κάθε νέος κόμβος ενσωματώνεται δυναμικά στο δίκτυο, ενημερώνει κάποιους από τους υπόλοιπους κόμβους για την παρουσία του και λαμβάνει τα δεδομένα που του αντιστοιχούν.

Όταν ένας νέος κόμβος προσπαθήσει να εισέλθει στο δίκτυο, αρχικά ελέγχεται αν το node_id του ήδη είναι στο δίκτυο. Αν υπάρχει ήδη κόμβος με το ίδιο ID, η είσοδος απορρίπτεται.

Στη συνέχεια, ο νέος κόμβος αποκτά ένα μοναδικό position στο δίκτυο. Οι πρώτοι 16 κόμβοι είναι ομοιόμορφα κατανεμημένοι στο διάστημα [0, 1], για καλύτερη εκλεξιμότητα και οι υπόλοιποι αποκτούν ένα τυχαίο position.

Μόλις καθοριστεί το position, ο κόμβος προστίθεται στο δίκτυο μέσω των βασικών δομών του:

self.nodes: Dictionary με όλους τους κόμβους του δικτύου και

self.node_ports: Dictionary που αντιστοιχεί κάθε ID κόμβου με το port του.

```
self.nodes[new_node_id] = new_node
self.node_ports[new_node_id] = new_node.port
```

Για να ενταχθεί σωστά, ο νέος κόμβος πρέπει να συνδεθεί με έναν ήδη υπάρχοντα κόμβο. Χρησιμοποιώντας την τοπολογική απόσταση, επιλέγεται ο πλησιέστερος κόμβος και αντιγράφει το Neighborhood Set του, ώστε να έχει άμεση γνώση των κοντινότερων γειτόνων του.

```
closest_node_id, closest_neighborhood_set = self._find_topologically_closest_node(new_node)
```

Ο νέος κόμβος στέλνει αίτημα σύνδεσης (NODE_JOIN) στον πλησιέστερο κόμβο, ώστε να ξεκινήσει η διαδικασία δρομολόγησης του αιτήματος εισαγωγής.

```
join_request = {
            "operation": "NODE_JOIN",
            "joining_node_id": new_node_id,
            "common_prefix_len": common_prefix_length(new_node_id, closest_node_id),
            "hops": [],  # Initialize an empty hops list
        }
response = new_node.send_request(self.node_ports[closest_node_id], join_request)
```

Το common prefix length μεταξύ του νέου κόμβου και του κοντινότερου, εισάγεται στο αίτημα για την διαχείριση της ειδικής περίπτωσης οπού ο κοντινότερος κόμβος πρέπει να μεταφέρει το routing table του μέχρι και την γραμμή που αντίστοιχή στο common prefix.

Μετά την επιτυχή σύνδεση, και αρχικοποίηση του routing table και leaf set, ο νέος κόμβος ανακοινώνει την ύπαρξή του στο δίκτυο, ώστε ορισμένοι κόμβοι να ενημερώσουν τις αντίστοιχες δομές τους (Routing Table, Leaf Set, Neighborhood Set). Τέλος, αν υπάρχουν δεδομένα που πρέπει να αποθηκευτούν στον νέο κόμβο, μεταφέρονται αυτόματα.

```
new_node.transmit_state()
new_node.get_keys()
```

transmit_state:

Όταν ένας νέος κόμβος εισέρχεται στο δίκτυο **Pastry**, πρέπει να ενημερώσει τους υπόλοιπους κόμβους ώστε να προσαρμόσουν τις δομές τους (Routing Table, Leaf Set, Neighborhood Set). Δημιουργείται ένα αίτημα ενημέρωσης (UPDATE_PRESENCE), το οποίο περιέχει το node_id του νέου κόμβου το οποίο στέλνεται σε όλους τους κόμβους που περιλαμβάνονται στις δομές του νέου κόμβου.

```
request = { "operation": "UPDATE_PRESENCE", "joining_node_id": self.node_id, "hops": [], }
```

get_keys:

Μεταφέρει δεδομένα από τους κοντινότερους κόμβους στον νέο κόμβο.

Ο νέος κόμβος αποστέλλει ένα μήνυμα GET_KEYS στους κοντινούς κόμβους, ζητώντας να μεταφέρουν τα κλειδιά που πρέπει να ανήκουν σε αυτόν:

```
request = {"operation": "GET_KEYS", "node_id": self.node_id, "hops": []}
self.send_request(self.network.node_ports[node_id], request)
```

Οι κόμβοι που λαμβάνουν το αίτημα υπολογίζουν ποια κλειδιά πρέπει να μετακινηθούν, χρησιμοποιώντας τη συνάρτηση _handle_get_keys_request.

Για κάθε κλειδί στη δομή KD-Tree του κόμβου, ελέγχεται αν ο νέος κόμβος είναι "πιο κοντά" στο κλειδί από τον τρέχοντα κόμβο, με βάση το κοινό πρόθεμα των Node IDs:

```
l = common_prefix_length(self.node_id, country_key)
if self._is_closer_node(request_node_id, country_key, l, self.node_id):
    keys_to_move.append(country_key)
```

Αν το request_node_id είναι πιο κοντά στο country_key, τότε το κλειδί μετακινείται στον νέο κόμβο. Μετά την επιτυχή εισαγωγή των κλειδιών στον νέο κόμβο, αφαιρούνται από τον αρχικό κόμβο:

find_topologically_closest_node:

Όταν ένας νέος κόμβος εντάσσεται στο δίκτυο Pastry, πρέπει να βρει τον πλησιέστερο κόμβο που ήδη υπάρχει. Αυτή η πληροφορία είναι κρίσιμη γιατί επιτρέπει στον νέο κόμβο να προσαρμόσει τις δομές δρομολόγησης του και να ενσωματωθεί ομαλά στο δίκτυο.

Η συνάρτηση υπολογίζει την απόσταση μεταξύ του νέου κόμβου και κάθε υπάρχοντος, με το να σταλθεί ένα αίτημα απόστασης στον κάθε κόμβο (DISTANCE).

```
for existing_node_id in self.node_ports.keys():
    # Skip the new node
    if existing_node_id == new_node.node_id:
        continue
    dist_request = {"operation": "DISTANCE", "node_position": new_node.position,"hops":
[],}
    response = new_node.send_request(self.node_ports[existing_node_id], dist_request)
```

3.6.4. Key Insert

Η εισαγωγή δεδομένων στο Pastry βασίζεται στη δρομολόγηση του κλειδιού (key) προς τον κατάλληλο κόμβο. Αυτή η διαδικασία εξασφαλίζει ότι τα δεδομένα αποθηκεύονται στον πιο κατάλληλο κόμβο με βάση το ID.

Το σύστημα χρησιμοποιεί τον αλγόριθμο δρομολόγησης για να βρει τον κόμβο που θα αποθηκεύσει το κλειδί. Αυτό γίνεται με τη μέθοδο _find_next_hop(key), η οποία βασίζεται στο Leaf Set και Routing Table.

```
next_hop_id = self._find_next_hop(key)
```

Αν το κλειδί ανήκει στον τρέχοντα κόμβο τότε το αποθηκεύει τοπικά στο KD-Tree του. Αν δεν υπάρχει ήδη KD-Tree, δημιουργείται και αποθηκεύει το κλειδί, αλλιώς εισάγονται κατευθείαν τα δεδομένα.

```
if not self.kd_tree:
    self.kd_tree = KDTree(
        points=np.array([request["point"]]),
        reviews=np.array([request["review"]]),
        country_keys=np.array([hash_key(request["country"])]),
        countries=np.array([request["country"]]),
    )
else:
    self.kd_tree.add_point(request["point"], request["review"], request["country"])
```

find_next_hop(key):

Όταν ένας κόμβος λαμβάνει ένα αίτημα για ένα συγκεκριμένο key, πρέπει να αποφασίσει σε ποιον κόμβο να το στείλει στη συνέχεια.

Αν το key ανήκει στο εύρος του Leaf Set, τότε βρίσκεται αριθμητικά κοντά στον τρέχοντα κόμβο. Καλεί τη find_closest_leaf_id(key) για να επιστρέψει τον κοντινότερο κόμβο στο Leaf Set.

```
if self._in_leaf_set(key):
    # Αν το key βρίσκεται στο Leaf Set
    closest_leaf_id = self._find_closest_leaf_id(key)
    return closest_leaf_id
```

Αν δεν υπάρχει στο Leaf Set, ψάχνουμε στο Routing Table.

```
else:
    i = common_prefix_length(self.node_id, key)
    next_hop = self.routing_table[i][int(key[i], 16)]
```

Αν το Routing Table δεν έχει καταχώρηση, βρίσκουμε το κοντινότερο διαθέσιμο κόμβο από όλες τις δομές.

```
if next_hop is not None:
    return next_hop
else:
    next_hop = self._find_closest_node_id_all(key)
    return next_hop
```

find_closest_leaf_id(key):

Η συνάρτηση _find_closest_leaf_id βρίσκει τον πιο κοντινό κόμβο στο key, μέσα στο Leaf Set (Lmin και Lmax).

find_closest_node_id_all(key):

Η συνάρτηση σαρώνει όλους τους διαθέσιμους κόμβους στο δίκτυο και βρίσκει τον πιο κοντινό κόμβο στον οποίο θα πρέπει να προωθηθεί ένα αίτημα με το συγκεκριμένο key.

3.5.5. Key Lookup

Με την λειτουργία Key Lookup πραγματοποιείται range search στα δεδομένα ενός KD-Tree για ένα συγκεκριμένο key. Στην συνέχεια, πραγματοποιείται LSH similarity search των N πιο όμοιων reviews, ανάμεσα σε αυτά που επιστράφηκαν από την αναζήτηση.

Η μέθοδος lookup ενός κόμβου καλείται ως εξής:

```
lower_bounds = [year, rating, price]
upper_bounds = [year, rating, price]
N = number_of_similar_reviews

node.lookup(key, lower_bounds, upper_bounds, N=N)
```

Στα lower και upper bounds υπάρχει δυνατότητα κάποια ζεύγη να είναι None.

Πχ.

```
lower_bounds = [2020, None, None]
upper_bounds = [2021, None, None]
```

Σε αυτή την περίπτωση θα πραγματοποιηθεί αναζήτηση για σημεία στο εύρος ετών [2020, 2021] και τα υπόλοιπα bounds θα συμπληρωθούν αυτόματα με τις ελάχιστες και μέγιστες τιμές στον κάθε άξονα που λείπει.

lookup:

Εσωτερικά στο στην μέθοδο lookup δημιουργείται το εξής request:

```
request = {
        "operation": "LOOKUP",
        "key": key,
        "lower_bounds": lower_bounds,
        "upper_bounds": upper_bounds,
        "N": N,
        "hops": [],  # Initialize hops tracking
    }
response = self._handle_lookup_request(request)
```

Το request αυτό δρομολογείται στο δίκτυο με την μέθοδο που έχει περιγραφεί παραπάνω, μέχρι να φτάσει στον κόμβο στον οποίο αντιστοιχεί το κλειδί.

_handle_lookup_request:

Όταν βρεθεί ο κόμβος και αυτός διαθέτει μη κενό KD-Tree, καλείται η μέθοδος search του αντικειμένου kd_tree.

```
points, reviews = self.kd_tree.search(key, lower_bounds, upper_bounds)
```

Η μέθοδος αυτή επιστρέφει τα επιθυμητά points και reviews.

Στην συνέχεια, αν ο αριθμός των reviews δεν είναι μηδενικός εκτελείται το LSH Similarity Search για την εύρεση των N reviews με την μεγαλύτερη ομοιότητα.

```
vectorizer = TfidfVectorizer()
doc_vectors = vectorizer.fit_transform(reviews).toarray()

lsh = LSH(num_bands=4, num_rows=5)
for vector in doc_vectors:
    lsh.add_document(vector)

similar_pairs = lsh.find_similar_pairs(N)
similar_docs = lsh.find_similar_docs(similar_pairs, reviews, N)
```

3.6.6. Key Delete

Στο Pastry, η διαγραφή ενός κλειδιού ακολουθεί μια διαδικασία παρόμοια με την εισαγωγή, με τη βασική διαφορά ότι το ζητούμενο κλειδί αφαιρείται από το δίκτυο.

Όπως και στο Insert Key, το σύστημα χρησιμοποιεί το Leaf Set και το Routing Table για να βρει τον κόμβο που αποθηκεύει το κλειδί. Αυτό επιτυγχάνεται με τη μέθοδο _find_next_hop(key), η οποία δρομολογεί το αίτημα προς τον κατάλληλο κόμβο.

Αν ο κόμβος που λαμβάνει το αίτημα δεν αποθηκεύει το κλειδί, προωθεί το αίτημα στον επόμενο κατάλληλο κόμβο.

```
print(f"Forwarding DELETE_KEY Request: {hops}")
response = self.send_request(self.network.node_ports[next_hop_id], request)

return response
```

3.6.7. Key Update

Η ενημέρωση κλειδιού στο δίκτυο Pastry επιτρέπει την τροποποίηση των δεδομένων που είναι αποθηκευμένα στο KD-Tree του υπεύθυνου κόμβου. Η διαδικασία αυτή διασφαλίζει ότι οι αλλαγές εφαρμόζονται σωστά, διατηρώντας τη συνοχή των δεδομένων.

Όπως και στο Insert Key, το σύστημα χρησιμοποιεί το Leaf Set και το Routing Table για να βρει τον κόμβο που αποθηκεύει το κλειδί. Αυτό επιτυγχάνεται με τη μέθοδο _find_next_hop(key), η οποία δρομολογεί το αίτημα προς τον κατάλληλο κόμβο.

Κατά την ενημέρωση δεδομένων σε ένα KD-Tree, χρησιμοποιούνται δύο βασικές παράμετροι:

updated_data – Περιέχει τις νέες τιμές που θα αποθηκευτούν.

criteria – Καθορίζει ποια δεδομένα πρέπει να ενημερωθούν (φιλτράρισμα).

Το updated_data είναι ένα dictionary που περιέχει τα πεδία και τις νέες τιμές που θέλουμε να εφαρμόσουμε. Αν δεν καθοριστεί κάποιο criteria, η ενημέρωση θα εφαρμοστεί σε όλα τα σημεία που αντιστοιχούν στο συγκεκριμένο country_key. Πχ:

```
taiwan_country_key = hash_key("Taiwan")

# Ενημερώνουμε την τιμή όλων των σημείων της Ταϊβάν σε 35.0
update_to = {"attributes": {"price": 35.0}}
first_node.update_key(key=taiwan_country_key, updated_data=update_to)
```

Το criteria λειτουργεί ως φίλτρο και επιτρέπει την ενημέρωση μόνο συγκεκριμένων δεδομένων που ταιριάζουν σε ορισμένες συνθήκες. Πχ:

```
update_fields_that_have = {"review_date": 2019, "rating": 94, "price": 35.0}
update_to = {"attributes": {"price": 36.0}}

first_node.update_key(key=taiwan_country_key, updated_data=update_to,
criteria=update_fields_that_have)
```

Για ταχύτερη ενημέρωση, στις περιπτώσεις που ενημερώνονται όλα τα πεδία, μπορούμε να ενημερώσουμε τις συντεταγμένες (point) των δεδομένων στο KD-Tree. Πχ:

```
update_to = {"point": [25.034, 121.564]}
first_node.update_key(key=taiwan_country_key, updated_data=update_to)
```

3.6.8. Node Leave

Στο **Pastry**, η αποχώρηση ενός κόμβου μπορεί να γίνει με **δύο διαφορετικούς τρόπους**, ανάλογα με το αν οι υπόλοιποι κόμβοι ειδοποιούνται για την αποχώρηση ή όχι:

Απροσδόκητη αποχώρηση **(leave_unexpected)** και αποχώρηση με ενημέρωση (**leave**).

leave_unexpected:

Όταν ένας κόμβος αποτυγχάνει απροσδόκητα, δεν υπάρχει προειδοποίηση προς τους υπόλοιπους κόμβους. Αυτό μπορεί να συμβεί, για παράδειγμα, αν ο κόμβος κλείσει ξαφνικά ή αποσυνδεθεί από το δίκτυο. Οι υπόλοιποι κόμβοι εξακολουθούν να πιστεύουν ότι ο κόμβος υπάρχει. Αυτό σημαίνει ότι ο αποτυχημένος κόμβος εξακολουθεί να εμφανίζεται στα Routing Table, Leaf Set, και Neighborhood Set των άλλων κόμβων, προκαλώντας προβλήματα κατά τη δρομολόγηση. Όταν κάποιος προσπαθήσει να επικοινωνήσει με τον αποτυχημένο κόμβο, θα καταλάβει ότι λείπει.

```
# Αφαίρεση του κόμβου χωρίς ενημέρωση των άλλων
del self.nodes[failing_node_id]
del self.node_ports[failing_node_id]
```

Αν κάποιος άλλος κόμβος προσπαθήσει να στείλει ένα μήνυμα (lookup, insert, update, delete, join) στον αποτυχημένο κόμβο, θα πάρει σφάλμα σύνδεσης και τότε θα ξεκινήσει η επιδιόρθωση. Η επιδιόρθωση γίνεται σταδιακά μέσω τριών λειτουργιών:

Επιδιόρθωση του Routing Table (repair_routing_table_entry):

Όταν ανιχνεύεται ένας αποτυχημένος κόμβος στο Routing Table, η επιδιόρθωση ακολουθεί τα εξής βήματα:

Αφαίρεση του αποτυχημένου κόμβου από το Routing Table αυτού που τον ανίχνευσε.

Αναζήτηση αντικατάστασης από άλλους κόμβους της ίδιας γραμμής (row) του Routing Table.

Αν δεν βρεθεί αντικατάσταση, γίνεται προοδευτική αναζήτηση χρησιμοποιώντας τη συνάρτηση _find_next_hop.

Αν δεν βρεθεί κατάλληλος κόμβος, η εγγραφή μένει κενή μέχρι να προστεθεί νέος κόμβος στο δίκτυο.

Επιδιόρθωση του Leaf Set (repair_leaf_set):

Όταν ένας κόμβος ανιχνεύσει ότι ένας γειτονικός του κόμβος έχει αποτύχει, εκτελεί τα εξής βήματα:

Αναγνωρίζει αν ο αποτυχημένος κόμβος ανήκε στο Lmin ή Lmax. Τον αφαιρεί από το αντίστοιχο Leaf Set.

Στέλνει αιτήσεις (GET_LEAF_SET) στους υπόλοιπους κόμβους του Leaf Set για να βρει υποψήφιους αντικαταστάτες.

Συγκεντρώνει όλους τους υποψήφιους κόμβους και τους ταξινομεί με βάση την αριθμητική απόσταση από το δικό του node_id.

Ανανεώνει το Lmin και το Lmax επιλέγοντας τους L/2 κοντινότερους κόμβους.

Επιδιόρθωση του Neighborhood Set (repair_neighborhood_set):

Όταν ένας κόμβος ανιχνεύσει ότι ένας γείτονάς του έχει αποτύχει, εκτελείται η εξής διαδικασία:

Ανιχνεύει αν ο αποτυχημένος κόμβος ανήκε στο Neighborhood Set

Αφαιρεί τον κόμβο από τη λίστα των γειτόνων

Αναζητά νέους κόμβους μέσω των γειτόνων του και του Leaf Set

Συλλέγει τους υποψήφιους αντικαταστάτες και ταξινομεί με βάση την τοπολογική απόσταση

Ανανεώνει το Neighborhood Set, κρατώντας τους πιο κοντινούς κόμβους

leave:

Η λειτουργία Leave στο Pastry διαφέρει από το Leave Unexpected, καθώς όταν ένας κόμβος αποφασίζει να αποχωρήσει κανονικά από το δίκτυο, ειδοποιεί τους υπόλοιπους κόμβους, ώστε να γίνει ανακατανομή των δεδομένων και αναδόμηση των δομών του δικτύου.

Όταν ένας κόμβος αποχωρεί κανονικά από το δίκτυο:

Ανακτά τα δεδομένα του (κλειδιά και σημεία από τον KD-Tree) και τα αποθηκεύει προσωρινά για αναδιανομή.

```
# Extract points, reviews, and countries from the KDTree
keys_to_store = [
      {"key": country_key,"position": point,"review": review,"country": country,}

       for country_key, point, review, country in zip(
                  leaving_node.kd_tree.country_keys,
                  leaving_node.kd_tree.points,
                  leaving_node.kd_tree.reviews,
                  leaving_node.kd_tree.countries,  # Use the original country)]
```

Αφαιρείται από το δίκτυο και διαγράφεται από τους πίνακες των κόμβων.
Ενημερώνει όλους τους γειτονικούς κόμβους για την αποχώρησή του.

```
del self.nodes[leaving_node_id]
del self.node_ports[leaving_node_id]

available_nodes = list(self.nodes.keys())
node_positions = {node_id: self.nodes[node_id].position for node_id in self.nodes}

for node_id in available_nodes:
   leave_request = {"operation": "NODE_LEAVE","leaving_node_id": leaving_node_id,
            "available_nodes": available_nodes,"node_positions": node_positions,
            "hops": hops,  # Pass the hops list}

response = self.nodes[node_id].send_request(self.node_ports[node_id], leave_request)
```

Ανακατανέμει τα δεδομένα του σε άλλους κόμβους που βρίσκονται στο δίκτυο.

```
self.nodes[closest_node_id].insert_key(key, position, review, country)
reinserted_count += 1
```

Όταν ένας κόμβος αποχωρεί από το δίκτυο Pastry ενημερώνοντας, δεν αρκεί μόνο να απομακρυνθεί από το δίκτυο. Οι υπόλοιποι κόμβοι πρέπει να ενημερωθούν, ώστε να ανανεώσουν τις δομές δρομολόγησης (Routing Table, Leaf Set, Neighborhood Set) και να προσαρμόσουν τη δομή τους. Η επεξεργασία αποχώρησης κόμβου γίνεται μέσω του NODE_LEAVE request που στάλθηκε σε όλους τους κόμβους. Κάθε κόμβος θα καλέσει την handle_leave_request() μέθοδο του. Αυτά είναι τα βήματα:

Αφαιρεί τον αποχωρούντα κόμβο από τα Leaf Sets, Neighborhood Set και Routing Table

Κάθε κόμβος που λαμβάνει το αίτημα αναπροσαρμόζει τις δομές του με ένα REBUILD_NODE_STATE αίτημα το οποίο εκτελείται από την μέθοδο _rebuild_node_state().

Η _rebuild_node_state() εκτελείται όταν λαμβάνεται ένα αίτημα REBUILD_NODE_STATE και ακολουθεί τα εξής βήματα:

Ανακατασκευή του Leaf Set

Οι δομές Lmin και Lmax αναδομούνται αναζητώντας τους **π**λησιέστερους αριθμητικά κόμβους μέσω των find_closest_lower_nodes και find_closest_higher_nodes.

```
self.Lmin = self._find_closest_lower_nodes(available_nodes)
self.Lmax = self._find_closest_higher_nodes(available_nodes)
```

Ανακατασκευή του Neighborhood Set

Χρησιμοποιώντας τη συνάρτηση _update_closest_neighbors(), επιλέγονται οι πιο κοντινοί κόμβοι με βάση τη γεωγραφική θέση τους.

```
self.neighborhood_set = self._update_closest_neighbors(available_nodes, node_positions)
```

Διόρθωση του Routing Table

Αφαιρούμε τον αποχωρούντα κόμβο από τον πίνακα δρομολόγησης (Routing Table) και βρίσκουμε αντικαταστάσεις.

```
cleared_positions = []
for row_idx, row in enumerate(self.routing_table):
    for col_idx, entry in enumerate(row):
        if entry == leaving_node_id:
            self.routing_table[row_idx][col_idx] = None
            cleared_positions.append((row_idx, col_idx))

for row_idx, col_idx in cleared_positions:
    for node_id in available_nodes:
        if common_prefix_length(self.node_id, node_id) == row_idx:
            if int(node_id[row_idx], 16) == col_idx:
                self.routing_table[row_idx][col_idx] = node_id
                break
```

Όταν τελειώσει και το rebuild τότε τα δεδομένα του αποχωρούντα κόμβου θα πρέπει να κατανεμηθούν πάλι στο δίκτυο.

Για κάθε key, βρίσκουμε τον πλησιέστερο κόμβο αριθμητικά με βάση το ID του. Αυτό διασφαλίζει ότι το key θα ανατεθεί στον πιο κατάλληλο κόμβο σύμφωνα με τη δομή του Pastry.

```
closest_node_id = min(
    available_nodes, key=lambda node_id: abs(int(node_id, 16) - int(key, 16)))
```

Έπειτα εισάγουμε τα κλειδιά στον κόμβο. Χρησιμοποιείται η insert_key() για να εισαχθεί το key στον νέο κόμβο. Αν συμβεί κάποιο σφάλμα, το key δεν εισάγεται και το πρόγραμμα το καταγράφει.

```
try:
   print(
      f"Network: Redirecting key {key} (Country: {country}) from Node: {node_id}  to Node
{closest_node_id}."
   )
   self.nodes[closest_node_id].insert_key(key, position, review, country)
   reinserted_count += 1
except Exception as e:
   print(
      f"Network: Failed to redirect key {key} to Node {closest_node_id}. Error: {e}"
   )
```

3.7 GUI

Για την καλύτερη παρουσίαση των υλοποιημένων DHTs, δημιουργήθηκε ένα γραφικό περιβάλλον (GUI), βασισμένο στην βιβλιοθήκη tkinter.

Η υλοποίηση του GUI, χωρίζεται σε τέσσερα αρχεία: main.py, dashboard.py, Pastry/pastry_gui.py και Chord/chord_gui.py, με τις κλάσεις MainLauncher, Dashboard, PastryDashboard και ChordDashboard αντίστοιχα.

MainLauncher:

Αυτή η κλάση εμφανίζει ένα παράθυρο επιλογής του DHT, και δημιουργεί ένα instance του Network class του επιλεγμένου DHT, αρχικοποιώντας το με κόμβους και όλα τα δεδομένα από το dataset. Εσωτερικά στο κάθε Network class δημιουργείται και ένα instance του αντίστοιχου Dashboard class. Οπότε το MainLauncher μπορεί να ανοίξει το επιλεγμένο Dashboard, μέσω του Network object.

Παρακάτω φαίνεται η μέθοδος για την αρχικοποίηση και εκκίνηση του Pastry Dashboard. Ενώ υπάρχει και αντίστοιχη μέθοδος για το Chord.

```
def handle_pastry_choice(self, dialog, predefined):
      """Handle the user's choice for Pastry network creation."""
      dialog.destroy()

      if predefined:
         # Build with predefined nodes
         self.withdraw()
         pastry_network = PastryNetwork(self)
         pastry_network.build(predefined_ids)
         pastry_network.gui.show_dht_gui()
         pastry_network.gui.root.mainloop()
          ….
          ….
          ….
```

Dashboard:

Αποτελεί parent class των ChordDashboard και PastryDashboard, και είναι υπεύθυνη για την δημιουργία της βασική δομής του παραθύρου του κάθε dashboard. Συμπληρωματικά, σε αυτή την κλάση βρίσκονται κοινές μέθοδοι για την διαχείριση της εκτέλεσης των κοινών λειτουργιών των δύο DHTs.

Μέθοδοι για την διαχείριση της εκτέλεσης λειτουργιών με διαφορές ανάμεσα στα DHTs, καθώς επίσης και για τη εμφάνιση των διαφορετικών οπτικοποιήσεων, είναι υλοποιημένες στις κλάσεις ChordDashboard και PastryDashboard.

ChordDashboard:

Περιέχει μεθόδους για την οπτικοποίηση του Chord ring και για την ειδική κλήση των μεθόδων του ChordNetwork και ChordNode, που αφορούν τις λειτουργίες Node Join και Node Leave.

PasrtyDashboard:

Περιέχει μεθόδους για την οπτικοποίηση του Pastry Overlay Network και Pastry Topology και για την ειδική κλήση των μεθόδων του PastryNetwork και PastryNode, που αφορούν τις λειτουργίες Node Join και Node Leave. Συγκεκριμένα για το Node Leave υπάρχει διαφορετική διαχείριση ανάλογα με την επιλογή απλού Node Leave (graceful), ή Node Leave Unexpected.

4. Experimental Evaluation

4.1. Performance Metric

Η μετρική που θα αξιολογηθεί για κάθε λειτουργία είναι τα **hops**, δηλαδή ο αριθμός των αλμάτων μεταξύ κόμβων για την δρομολόγηση ενός αιτήματος στο δίκτυο. Συγκεκριμένα για μια πιο αξιόπιστη αξιολόγηση, υπολογίζεται ο μέσος όρος των hops, με διαφορετικό κόμβο να εκτελεί την λειτουργία κάθε φορά.

4.2. Method

4.2.1. Chord

Όπως έχει ήδη αναφερθεί στην περιγραφή των αρχείων το αρχείο Chord\test.py χρησιμοποιείται για την πειραματική καταγραφή των hops για τα operations: join node, insert key, update key, look up key και delete key.

Στο τέλος της εκτέλεσης του προγράμματος αποθηκεύονται τα αποτελέσματα στο αρχείο ChordResults.json που βρίσκεται στο ίδιο directory με το script που εκτελέστηκε.

Εισαγωγή των προκαθορισμένων κόμβων στο δίκτυο:

```
hops = []
    # Node Insertion
    for node_id in predefined_ids:
        node = ChordNode(network, node_id=node_id)
        node.start_server()
        hops.append(len(network.node_join(node)) - 1)

    print(f"Total num of hops to insert {len(hops)} nodes: {sum(hops)}")
    print(f"Avarage num of hops to insert a node: {sum(hops)/len(hops)}")
    Results[chord_operations[0]] = sum(hops) / len(hops)
```

Εισαγωγή όλων των κλειδιών και αποθήκευση των hops για κάθε insert:

```
def insert_keys(network, keys, points, reviews, countries, names):
    """Insert all keys sequentially."""
    responses = []
    for key, point, review, country, name in zip(keys, points, reviews, countries, names):
        response = network.insert_key(key, point, review, country)
        responses.append(response["hops"])
    return responses
```

Ενημέρωση όλων των κλειδιών (μοναδικές τιμές κλειδιών) με τις παρακάτω παραμέτρους:

```
def update_keys(network, keys):
    """Update all keys sequentially."""
    updated_data = {
        "point": [2018, 94, 5.5],
        "review": "New review text",
        "attributes": {"rating": 95},
    }
    hops = 0
    for key in keys:
        hops += network.update_key(key, updated_data)["hops"]
    return hops / len(keys)
```

Αναζήτηση όλων των κλειδιών (μοναδικές τιμές κλειδιών) με τις παρακάτω παραμέτρους:

```
def lookups(network, keys):
    """Perform lookups for all keys."""
    N = 0
    lower_bounds = [2018, 0, 0]
    upper_bounds = [2018, 0, 0]
    hops = 0
    for key in keys:
        hops += network.lookup(key, lower_bounds, upper_bounds, N)["hops"]
    return hops / len(keys)
```

Διαγραφή όλων των κλειδιών (μοναδικές τιμές κλειδιών):

```
def delete_keys(network, keys):
    """Delelte all keys sequentially."""
    responses = []
    for key in keys:
        response = network.delete_key(key)
        responses.append(response["hops"])
    return responses
```

4.2.2. Pastry

Για κάθε operation του pastry δημιουργήθηκε και ένα test αρχείο ώστε να μετρηθούν πόσα average hops χρειάζονται. Τα test αρχεία είναι τα παρακάτω:

1. **Node Join** και **Key Insert** μέσω του αρχείου build_test.py:

Για την είσοδο νέων κόμβων στο δίκτυο, μετρήθηκε ο αριθμός των hops που απαιτούνται για την εύρεση της σωστής θέσης του κόμβου στο δίκτυο.

```
avg_join_hops, _ = network.build(predefined_ids=predefined_ids, dataset_path="data.csv")
print(f"Μέσος αριθμός hops για Node Join: {avg_join_hops}")
```

Το **avg_join_hops** προέρχεται από την συνάρτηση build

```
node.start_server()
response = self.node_join(node)
if response and "hops" in response:
    num_join_hops += len(response["hops"])
print(f"\nNode Added: ID = {node.node_id}, Position = {node.position}")
print("\n" + "-" * 100)
avg_join_hops = num_join_hops / node_num
```

Τα hops που επιστρέφονται μαζί με το response είναι αποτέλεσμα του handle_join_request που επιστρέφεται στην build() μέσω της συνάρτησής node_join() που μετράει πόσα hops μεταξύ κόμβων χρειάστηκε για να γίνει join ο κόμβος.

- Η εισαγωγή δεδομένων στο δίκτυο απαιτεί την εύρεση του σωστού κόμβου που θα αποθηκεύσει το **key**.

Για την εισαγωγή ενός κλειδιού, μετρήθηκε ο αριθμός των hops που απαιτούνται για την εύρεση του κατάλληλου κόμβου για την εισαγωγή των κλειδιών.

```
_, avg_insert_hops = network.build(predefined_ids=predefined_ids, dataset_path="data.csv")
print(f"Μέσος αριθμός hops για Key Insertion: {avg_insert_hops}")
```

Το **avg_insert_hops** προέρχεται από την συνάρτηση build

```
response = random_node.insert_key(key, point, review, country)
if response and "hops" in response:
   num_insert_hops += len(response["hops"])
avg_insert_hops = num_insert_hops / len(keys)
```

Τα hops που επιστρέφονται μαζί με το response είναι αποτέλεσμα του handle_insert_key_request που επιστρέφεται στην build() μέσω της συνάρτησής insert_key() που μετράει πόσα hops μεταξύ κόμβων χρειάστηκε για να γίνει insert το κλειδί.

2. **Key Delete** μέσω του αρχείου delete_test.py:

Πρώτα βρίσκουμε όλα τα μοναδικά country keys που υπάρχουν στους κόμβους.

```
unique_country_keys = set()
for node in network.nodes.values():
   if node.kd_tree is not None:
      keys, _ = node.kd_tree.get_unique_country_keys()
      unique_country_keys.update(keys)
```

Έπειτα κάθε country key διαγράφεται από έναν τυχαίο κόμβο μετράμε τον αριθμό των hops.

```
for key in unique_country_keys:
   random_node = random.choice(list(network.nodes.values())) # Επιλέγει τυχαίο κόμβο
response = random_node.delete_key(key) # Διαγράφει το κλειδί
if response and "hops" in response:
   total_hops += len(response["hops"])
   total_deletions += 1
```

Στο τέλος υπολογίζουμε και τα average hops.

```
if total_deletions > 0:
  average_hops = total_hops / total_deletions
```

3. **Lookup Key** μέσω του αρχείου lookup_test.py:

   Με παρόμοιο τρόπο όπως το delete key γίνεται και το test για το lookup. Κάθε κλειδί γίνεται lookup από έναν τυχαίο κόμβο. Γίνεται καταγραφή των συνολικών hops και έπειτα υπολογίζουμε τα average hops.

4. **Key Update** μέσω του αρχείου update_test.py:

   Με παρόμοιο τρόπο όπως το delete key γίνεται και το test για το update. Κάθε κλειδί γίνεται update από έναν τυχαίο κόμβο. Γίνεται καταγραφή των συνολικών hops και έπειτα υπολογίζουμε τα average hops.

5. **Node Leave** μέσω του αρχείου leave_test.py:

Με παρόμοιο τρόπο γίνεται και το test για το Leave. Όλοι οι κόμβοι κάνουν leave τυχαία ενημερώνοντας τους υπόλοιπους κόμβους του δικτύου. Γίνεται καταγραφή των συνολικών hops που χρειάστηκε να ενημερώσει τους άλλους.

6. Ταυτόχρονο test για **όλα τα operations** μέσω του αρχείου all_test.py:
   Το αρχείο αυτό χρησιμοποιεί τα αρχεία/συναρτήσεις που αναφέρθηκαν παραπάνω και εκτελεί όλα τα operation build – lookup – update – delete– leave.

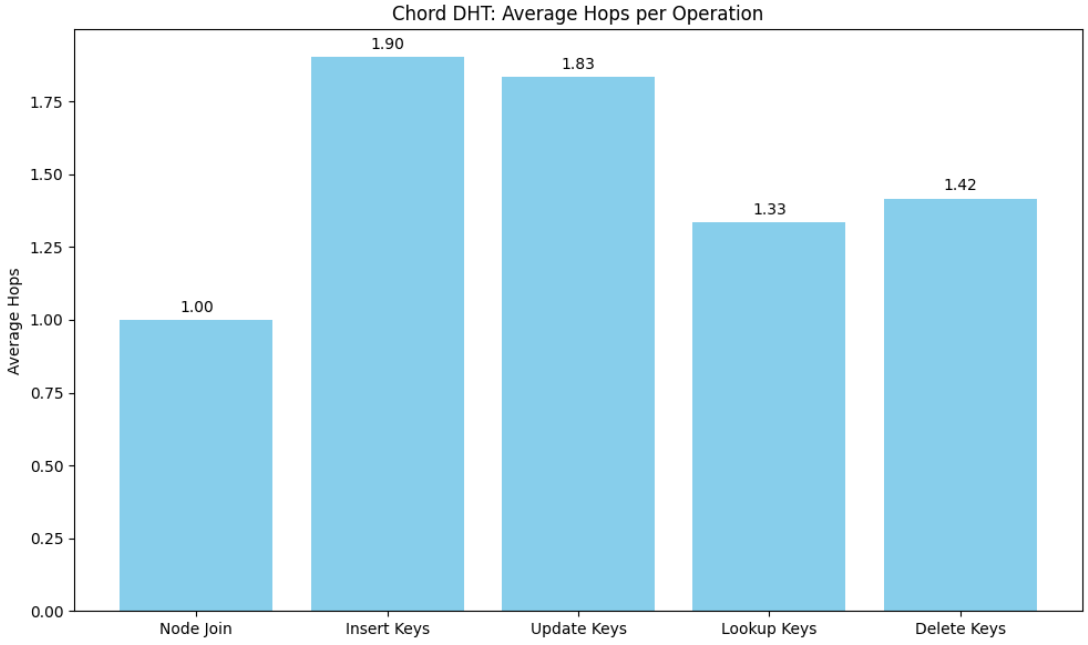
4.3. Results

Παρακάτω απεικονίζονται τα plots για τα average hops των Chord και Pastry.

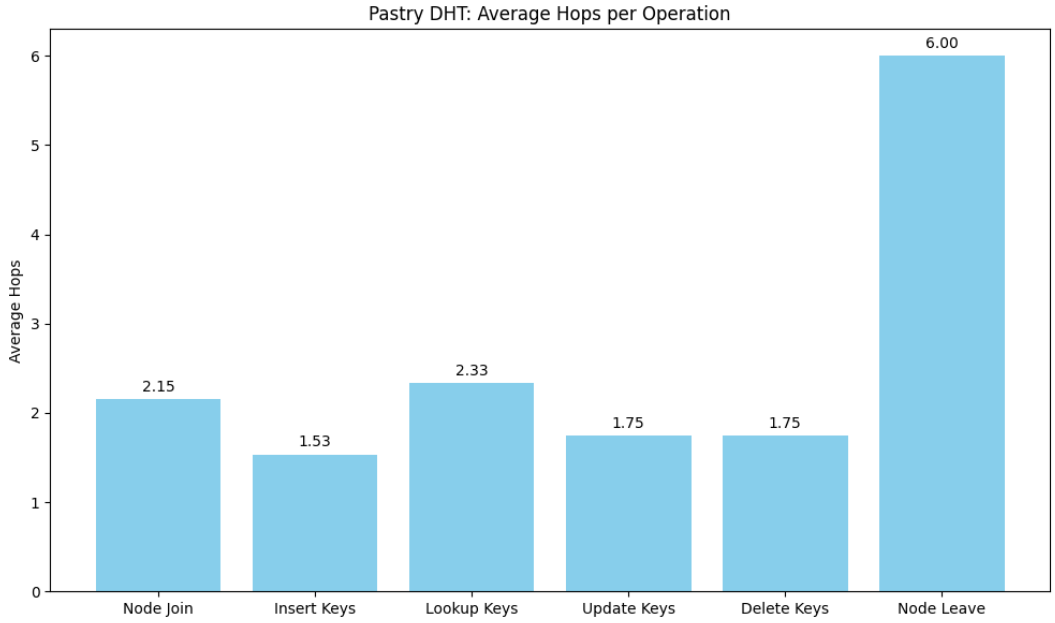
Στο Chord υπάρχουν τα average hops για Node Join, Insert Keys, Update Keys, Lookup Keys και Delete Keys.

Στο Pastry υπάρχουν τα average hops για Node Join, Insert Keys, Update Keys, Lookup Keys, Delete Keys και Node Leave όταν ενημερώνει όλους τους κόμβους.

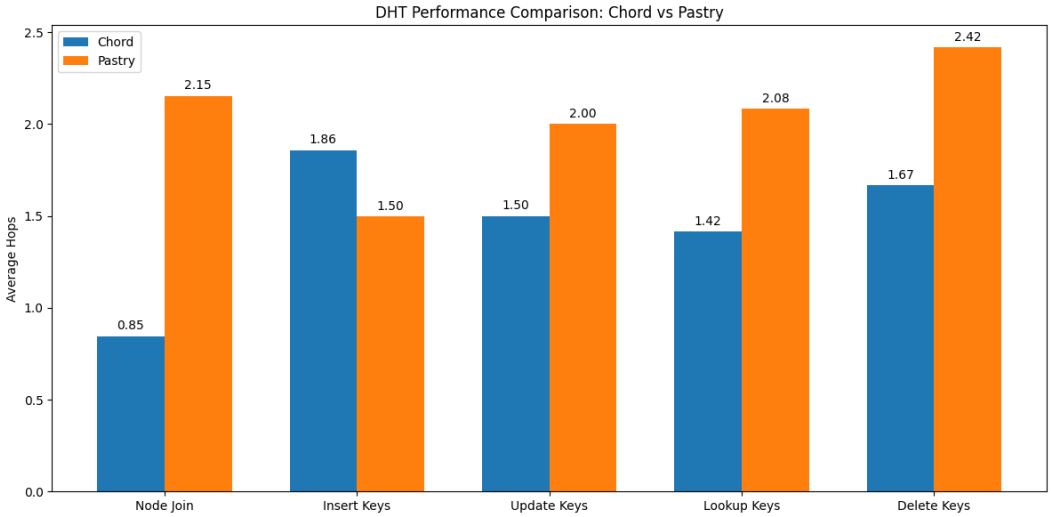
4.3.1. Chord



4.3.2. Pastry



4.3.3. Comparisons



4.4. Analysis

Παρατηρείται πως και για τα δύο DHTs ο μέσος όρος των hops για 13 κόμβους, είναι σχετικά κοντά στο θεωρητικό $\log_{16} N = 0.93$.

Σημειώνεται πως για το Pastry, στο Node Leave απεικονίζεται μεγάλος αριθμός από hops, μιας και αναφέρεται στο Expected Node Leave, που απαιτεί πολλαπλά hops για την ενημέρωση των κόμβων.

Συγκρίνοντας τα αποτελέσματα των δύο DHTs, παρατηρείται πως το Pastry απαιτεί περισσότερα hops συγκριτικά με το Chord. Αυτό συμβαίνει επειδή στην υλοποίηση μας το ID

space είναι αρκετά μεγαλύτερο από τον αριθμό των κόμβων κάτι που επηρεάζει αρνητικά την απόδοση του Pastry.

Τα εκθετικά άλματα του Chord εγγυούνται την μείωση της απόστασης στο μισό σε κάθε hop, ενώ το prefix matching του Pastry, μπορεί να χρειαστεί κάποιες διορθώσεις, ακόμα και αν υπάρχουν κοντινότερα μονοπάτια.

References

[1] Rowstron, A., Druschel, P. (2001). Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Guerraoui, R. (eds) Middleware 2001. Middleware 2001. Lecture Notes in Computer Science, vol 2218. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45518-3_18

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 4 (October 2001), 149–160. https://dl.acm.org/doi/10.1145/964723.383071