Πολυδιάστατες Δομές Δεδομένων

Project 2021-2022

DHTs (Chord)

Στυλιανός Στυλιανάκης	1059713
Χάρης Καπελετιώτης	1057772
Κωνσταντίνος Κωστόπουλος	1067482

Link to GitHub

Πίνακας περιεχομένων

Λίγη θεωρία

Distributed Hash Tables

Οι κατανεμημένοι πίνακες κατακερματισμού είναι μία δομή **αποκεντροποιημένης** διαμοίρασης αρχείων, δομημένων σε ζευγάρια κλειδιού-τιμής.

Κάθε **κόμβος** ενός DHT είναι υπεύθυνος για την αποθήκευση ενός υποσυνόλου κλειδιών και των τιμών τους, οι οποίες μπορεί να είναι οποιουδήποτε τύπου δεδομένων. Τα κλειδιά είναι μοναδικά αναγνωριστικά που δημιουργούνται μετά από **κατακερματισμό** των δεδομένων τιμών, ενώ οι κόμβοι που συμμετέχουν στη δομή δρουν ως **ισάξιες** οντότητες για τον διαμοιρασμό των αρχείων.

Ένα από τα κύρια πλεονεκτήματα των DHTs είναι ότι οι κόμβοι μπορούν να εισέρχονται και να απομακρύνονται από το σύστημα με συνοπτικές διαδικασίες ανανέωσης κλειδιών. Για αυτόν το λόγο, τέτοια συστήματα έχουν τη δυνατότητα να **κλιμακώνονται** μέχρι και έναν πολύ μεγάλο αριθμό κόμβων, ενώ η **ανοχή** τους **στις βλάβες** τα καθιστά ικανά να διαχειρίζονται **ταυτόχρονες** αφίξεις και αναχωρήσεις κόμβων.

Chord

Το Chord είναι ένα **πρωτόκολλο** κατανεμημένων πινάκων κατακερματισμού, το οποίο παρουσιάστηκε από το MIT το 2001 και βελτιώνει δραματικά την **πολυπλοκότητα** (και άρα το χρόνο) των αναζητήσεων κλειδιών.

Για την ισοκατανομή των κλειδιών στους κόμβους, το Chord χρησιμοποιεί **consistent hashing** αλγόριθμους για τον κατακερματισμό των κλειδιών. Συγκεκριμένα, ο βασικός αλγόριθμος που χρησιμοποιείται είναι ο **SHA-1**, ο οποίος χρησιμοποιήθηκε και στην εργασία, ύστερα από τροποποίησή του για να περιοριστούν οι τιμές σε μικρότερους **χώρους κατακερματισμού**.

Για την επιτάχυνση της αναζήτησης κλειδιών, οι κόμβοι αποθηκεύουν έναν πίνακα δρομολόγησης, γνωστό ως **finger table**, ο οποίος περιέχει KS εγγραφές. Κάθε εγγραφή αποτελείται από μία θέση και έναν κόμβο, ο οποίος ευθύνεται (πιθανώς μεταξύ άλλων και) για τα κλειδιά τα οποία ανήκουν στο διάστημα [θέση, κόμβος εγγραφής]. Λόγω του πίνακα αυτού, το κόστος της αναζήτησης ενός κλειδιού είναι O(log(n)).

Υλοποίηση

Για την υλοποίηση της εργασίας δημιουργήθηκαν 4 αρχεία κώδικα. Το node.py, το interface.py το benchmarks.py και φυσικά η main.py.

Το αρχείο node.py

Βοηθητικές συναρτήσεις στο node.py

hash_func

Ο αλγόριθμος του κατακερματισμού. Στο παρόν πόνημα, ο βασικός αλγόριθμος κατακερματισμού που χρησιμοποιείται είναι ο SHA-1.

```
def hash_func(data) -> int:
    hash_out = hashlib.sha1()
    hash_out.update(bytes(data.encode("utf-8")))
    return int(hash_out.hexdigest(), 16) % HS
```

cw_dist

Επιστρέφει την clockwise απόσταση των δύο ορισμάτων (συνήθως κλειδιά).

```
def cw_dist(k1: int, k2: int) -> int:
    """Clockwise distance of 2 keys"""

    if k1 <= k2:
        return k2 - k1
    else:
        return (HS) + (k2 - k1)
```

comp_cw_dist

Με είσοδο 3 τιμές k1, k2 και dest χρησιμοποιεί την συνάρτηση `cw_dist` και ελέγχει αν ισχύει η σχέση k2 ∈ (k1, dest]. Επιστρέφει True αν ισχύει, αλλιώς επιστρέφει False.

```
def comp_cw_dist(k1: int, k2: int, dest: int) -> bool:
    """Returns true if clockwise distance of k1 from dest
    is bigger than clockwise distance of k2 from dest.
    In other words, k2 ∈ (k1, dest]"""

    if cw_dist(k1, dest) > cw_dist(k2, dest):
        return True
    return False
```

Κλάση Node

Η βασική κλάση της υλοποίησης, αναπαριστά έναν **κόμβο** δικτύου ο οποίος μπορεί να τρέξει όλες τις απαραίτητες διαδικασίες, όπως παρουσιάζονται παρακάτω.

`node.id`	Το id του κόμβου.
`node.items`	Dictionary που περιέχει τα στοιχεία του κόμβου. Τα κλειδιά του λεκτικού είναι τα unhashed κλειδιά των ζεύγων (κλειδιών, δεδομένων) και οι τιμές τους τα δεδομένα.
`node.f_table`	Το finger table του κόμβου (λίστα). Κάθε εγγραφή της λίστας αποτελείται από μία λίστα 2 στοιχείων της μορφής [position, node].
`node.pred`	Ο αμέσως προηγούμενος κόμβος.
`node.succ_list`	Λίστα με τους άμεσους successors μεγέθους SLS, μία global σταθερά που εμείς έχουμε ορίσει να είναι 3.

Εδώ να σημειωθεί πως το μέτρο κατά των **massive node failures** δεν καταφέραμε να το υλοποιήσουμε λόγω έλλειψης χρόνου και δυναμικού. Παρόλα αυτά, έχουν υλοποιηθεί τα θεμέλια για τη λειτουργία αυτή, αφού αποθηκεύουμε τη λίστα λίστα `succ_list` σε κάθε κόμβο, η οποία ενημερώνεται. Αυτό που λείπει λοιπόν είναι η **σύνδεση** της λίστας αυτής με τη διαδικασία **αναζήτησης κλειδιού** και η σωστή **ενημέρωση** των **finger tables** σε περίπτωση που κάποιος successor δεν ανταπεξέρχεται.

Συναρτήσεις της κλάσης Node

`find_successor`

Επιστρέφει τον υπεύθυνο κόμβο για ένα κλειδί. Δηλαδή, τον κόμβο με id μεγαλύτερο ή ίσο με το κλειδί που έχει δοθεί, με την βοήθεια των μεθόδων closest_pre_node και comp_cw_dist.

```python
def find_successor(self, key: int) -> 'Node':
    """Returns the node with the shortest
    clockwise distance from the given key"""

    current = self.closest_pre_node(key)
    next = current.closest_pre_node(key)

    while comp_cw_dist(current.id, next.id, key):
        current = next
        next = current.closest_pre_node(key)

    if current.id == key:
        return current

    return current.f_table[0][1]
```

closest_pre_node

Ψάχνοντας στο finger table του κόμβου από τον οποίο έχει καλεστεί, επιστρέφει τον κοντινότερο predecessor κόμβο ενός κλειδιού.

```
def closest_pre_node(self, key: int) -> 'Node':
    """Returns the last predecessor from THIS node's finger table"""

    current = self
    for i in range(KS):
        # current.successor ∈ (current, key]
        if comp_cw_dist(current.id, current.f_table[i][1].id, key):
            current = current.f_table[i][1]
    return current
```

fix_fingers

Ανανεώνει το finger table ενός κόμβου.

```
def fix_fingers(self) -> None:
    """Called periodically.
    Refreshes finger table entries."""

    for i in range(KS - 1):
        next_in_finger = self.f_table[i + 1]
        next_in_finger[1] = self.f_table[i][1].find_successor(next_in_finger[0])
```

fix_successor_list

Ανανεώνει τη λίστα με τους αμέσως επόμενους successors ενός κόμβου.

```
def fix_successor_list(self) -> None:
    """Called periodically.
    Refreshes successor list."""

    next_successor = self
    for i in range(SLS):
        if next_successor.f_table[0][1] == self:
            break
        self.succ_list[i] = next_successor.f_table[0][1]
        next_successor = next_successor.f_table[0][1]
```

insert_new_pred

Η συνάρτηση που καλείται όταν εισέρχεται ένας καινούριος κόμβος στο δίκτυο. Ο λόγος που ονομάζεται έτσι, είναι επειδή καλείται στον πρώτο successor του καινούριου κόμβου, στον οποίο «εισάγουμε» έναν καινούριο predecessor.

```python
def insert_new_pred(self, new_n: 'Node') -> None:
    """Inserts new node to the network as this node's predecessor.
    Also updates neighboring nodes and necessary finger tables."""

    # New node's successor is this node
    new_n.f_table.append([(new_n.id + 1) % (HS), self])
    # Predecessor's new successor is the new node
    self.pred.f_table[0][1] = new_n
    # New node's predecessor is this node's predecessor
    new_n.pred = self.pred
    # This node's predecessor is the new node
    self.pred = new_n

    self.move_items_to_pred()
    new_n.initialize_finger_table()
    new_n.fix_successor_list()
    new_n.update_necessary_fingers(joinning=True)
```

insert_item_to_node

Προσθέτει ένα καινούριο αντικείμενο στον κόμβο.

```python
def insert_item_to_node(self, new_item: tuple, print_item=False) -> None:
    """Insert data in the node."""

    if print_item:
        print(f"Item with key {new_item[0]} before updating record:\n{self.items[new_item[0]]}")
    self.items[new_item[0]] = new_item[1]
    if print_item:
        print(f"Item with key {new_item[0]} after updating record:\n{self.items[new_item[0]]}")
```

delete_item_from_node

Δεδομένου ενός κλειδιού, ελέγχει αν υπάρχει το αντίστοιχο αντικείμενο στον κόμβο και το διαγράφει. Αλλιώς εκτυπώνει ότι δεν βρέθηκε.

```python
def delete_item_from_node(self, key: str, item_print: bool = False) -> None:
    """Given a key, checks if item exists in a node's item
    and removes it. Otherwise prints not found."""

    if key in self.items:
        if item_print:
            print(f"Node before removing item with key {key}:")
            self.print_node(items_print=True)
        del(self.items[key])
        if item_print:
            print(f"Node after removing item with key {key}:")
            self.print_node(items_print=True)
        return
    print(f"Key {key} not found")
```

move_items_to_pred

Μετά την είσοδο ενός καινούριου κόμβου στο δίκτυο, καλείται στον successor του. Μεταφέρει στον καινούριο κόμβο τα αντικείμενα που έχουν κλειδί μικρότερο ή ίσο από το id του καινούριου κόμβου.

```
def move_items_to_pred(self) -> None:
    """Moves node's items to predecessor.
    Used after a new node joins the network.
    Assumes all predecessors are up to date."""

    for key in sorted(self.items):
        # key ∉ (previous predecessor, new node (current predecessor)]
        if not comp_cw_dist(self.pred.pred.id, hash_func(key), self.pred.id):
            break
        self.pred.items[key] = self.items[key]
        del self.items[key]
```

initialize_finger_table

Αρχικοποιεί το finger table ενός καινούριου κόμβου.

```
def initialize_finger_table(self) -> None:
    """Initialize node's finger table.
    Assumes node's successor is up to date."""

    i = 1
    while i < KS:
        pos = (self.id + (2**i)) % (HS)

        # While pos ∈ (new_n.id, new_n.successor]
        while (i < KS) and comp_cw_dist(self.id, pos, self.f_table[0][1].id):
            # new_n [i] = new_n.successor
            self.f_table.append([pos, self.f_table[0][1]])
            i += 1
            pos = (self.id + (2**i)) % (HS)

        if i == KS:
            break

        self.f_table.append([pos, self.f_table[0][1].find_successor(pos)])
        i += 1
```

`leave`

Αποχώρηση του κόμβου από το δίκτυο

```python
def leave(self) -> None:
    """Removes node from the network."""

    # Move all keys to successor node
    self.f_table[0][1].items = self.f_table[0][1].items | self.items
    # Update successor's predecessor
    self.f_table[0][1].pred = self.pred
    # Update predecessor's successor
    self.pred.f_table[0][1] = self.f_table[0][1]

    self.update_necessary_fingers()
```

`update_necessary_fingers`

Ανανεώνει τα finger tables (και το successor list) μετά την άφιξη και την αποχώρηση κόμβων.

```python
def update_necessary_fingers(self, joinning = False) -> None:
    """Updates necessary finger tables on node join/leave"""

    furthest_possible_pred_id = self.calc_furth_poss_pred()
    next_pred = self.pred
    if next_pred == self or next_pred is None:
        return

    i = 0
    # next_pred.id ∈ (furthest_possible_pred_id, self]
    while comp_cw_dist(furthest_possible_pred_id, next_pred.id, self.id):
        next_pred.fix_fingers()
        if i < SLS:
            next_pred.fix_successor_list()
            i += 1
        next_pred = next_pred.pred
        if next_pred == self or next_pred is None:
            return
    if not joinning:
        comp = self
    else:
        comp = self.f_table[0][1]

    # next_pred last = current node
    while next_pred.f_table[KS-1][1] == comp:
        next_pred.fix_fingers()
        next_pred = next_pred.pred
        if next_pred == self or next_pred is None:
            return
```

print_node

Εκτυπώνει έναν κόμβο. Ως είσοδο δέχεται boolean μεταβλητές που λειτουργούν ως επιλογές εκτύπωσης των αντικειμένων μέσα του και του finger table.

```
def print_node(self, items_print = False, finger_print = False) -> None:
    print(f"Node ID: {hex(self.id)}")
    print(f"Predecessor ID: {hex(self.pred.id)}")
    self.print_succ()
    if items_print:
        print(f"Items in node: {[key for key in self.items.keys()]}")
    if finger_print:
        print("Finger table:")
        for entry in self.f_table:
            print(f"{hex(entry[0])} -> {hex(entry[1].id)}")
    print()
```

Το αρχείο interface.py

parse_csv

Η συνάρτηση αυτή διαβάζει το dataset (csv), δημιουργεί τα αντικείμενα (τα οποία είναι λεξικά) και τα περνάει σε μία λίστα, την οποία τελικά επιστρέφει.

```
def parse_csv(filename: str) -> dict:
    """Parses csv and returns a list of items."""

    items = {}
    df = pd.read_csv(filename)

    for i in range(len(df)):
        key = '_'.join([df.values[i][0], str(df.values[i][2])])

        data = {
            'Date': df.values[i][0],
            'Block': df.values[i][1],
            'Plot': df.values[i][2],
            'Experimental_treatment': df.values[i][3],
            'Soil_NH4': df.values[i][4],
            'Soil_NO3': df.values[i][5],
        }
        items[key] = data

    return items
```

Κλάση Interface

Η κλάση αυτή αποτελεί, όπως λέει και το όνομά της, ένα interface από το οποίο **προσομοιάζεται** η πρόσβαση στους διαφορετικούς κόμβους του δικτύου, ώστε να μπορούμε να ξεκινάμε διαδικασίες ως οποιοσδήποτε κόμβος θέλουμε. Οι κόμβοι του δικτύου αποθηκεύονται στο λεξικό **nodes** του interface.

Συναρτήσεις της κλάσης Interface

`build_network`

Δημιουργεί το δίκτυο είτε με συγκεκριμένα node ids που έχουν περαστεί ως λίστα είτε με τυχαία, εφόσον ορίσουμε τον αριθμό κόμβων που θέλουμε να δημιουργηθούν.

Αν δεν δώσουμε το όρισμα node_ids στην συνάρτηση, αρχικοποιείται το δίκτυο με κόμβους τυχαίου id και εντός του Hashing Space, διαφορετικά αρχικοποιείται το δίκτυο με κόμβους με τα αντίστοιχα ids που έχουν δοθεί. Για την εισαγωγή ενός κόμβου στο δίκτυο χρησιμοποιείται η συνάρτηση node_join(), η οποία εξηγείται παρακάτω.

```python
def build_network(self, node_count: int, node_ids: list = []) -> None:
    """Creates nodes and inserts them into the network."""

    if node_ids == []:
        final_ids = random.sample(range(HS), node_count)
    else:
        final_ids = node_ids

    for x in final_ids:
        self.node_join(new_node_id=x)
```

`get_node`

Επιστρέφει τον κόμβο με id node_id. Εάν δεν βρεθεί, επιστρέφει τον **πρώτο** κόμβο που εισήλθε στο δίκτυο. Χρησιμοποιείται από τις συναρτήσεις του interface, ώστε να μπορούμε να ξεκινάμε διαδικασίες από **οποιονδήποτε** κόμβο του δικτύου θέλουμε. Για τον σκοπό αυτό, οι συναρτήσεις παίρνουν σαν προαιρετικό όρισμα ένα `start_node_id` που έχει default τιμή `None` και στη συνέχεια καλούν τη συνάρτηση `get_node` με το όρισμα αυτό.

```python
def get_node(self, node_id: int = None) -> Node:
    """Returns node with id node_id. If it's not found,
    it returns the first node that joined the network."""

    # If start_node is specified
    if node_id != None:
        if node_id in self.nodes:
            return self.nodes[node_id]
        else:
            print(f"Node with id {node_id} not found.")
            return

    # If nodes dictionary is not empty
    if self.nodes:
        # Return first inserted node
        first_in_node = list(self.nodes.items())[0][1]
        #print(f"Returning first inserted node with id: {hex(first_in_node.id)}")
        return first_in_node
```

node_leave

Δεδομένου ενός id, διαγράφει έναν κόμβο από το δίκτυο.

```python
def node_leave(self, node_id: int, start_node_id: int = None, print_node = False) -> None:
    """Removes node from network."""

    node_to_remove = self.get_node(start_node_id).find_successor(node_id)
    if node_to_remove.id != node_id:
        print(f"Node {node_id} not found.")
        return

    if print_node:
        print("Node that will be removed from network:")
        node_to_remove.print_node(items_print=True)
        print(f"Successor node before {hex(node_id)} leave:")
        successor = node_to_remove.f_table[0][1]
        successor.print_node(items_print=True)

    node_to_remove.leave()
    del(self.nodes[node_id])

    if print_node:
        print(f"Successor node after {hex(node_id)} leave:")
        successor.print_node(items_print=True)
```

node_join

Δεδομένου ενός id, δημιουργεί και προσθέτει έναν κόμβο στο δίκτυο. Αρχικά ελέγχεται εάν το id του κόμβου είναι μέσα στο Hashing Space, και εάν δεν είναι, δεν εισάγεται ο κόμβος και η συνάρτηση τερματίζει εκεί. Διαφορετικά,

- εάν είναι ο πρώτος κόμβος που εισάγεται, υπολογίζεται το finger table του, και έπειτα εισάγεται στο δίκτυο.
- εάν **δεν** είναι ο πρώτος κόμβος που εισάγεται, μέσω της find_successor() υπολογίζεται ο successor S του κόμβου K, και ο K αποθηκεύεται στο δίκτυο ως predecessor στον κόμβο S (το finger table του κόμβου K υπολογίζεται μέσω της insert_new_pred()).

```python
def node_join(self, new_node_id: int, start_node_id: int = None, print_node: boolean = False) -> None:
    """Adds node to the network."""

    if new_node_id not in range(HS):
        print(f"{hex(new_node_id)} not in hashing space, can't create node.")
        return
    if print_node:
        print(f"Creating and adding node {hex(new_node_id)} to the network...")
    new_node = Node(new_node_id)
    # First node.
    if not self.nodes:
        new_node.pred = new_node
        # Initialize finger table.
        new_node.f_table = [ [(new_node.id + 2**i) % HS, new_node] for i in range(KS) ]
    else:
        start_node = self.get_node(start_node_id)
        # Find new node successor and insert the new node before it.
        start_node.find_successor(new_node.id).insert_new_pred(new_node)

    self.nodes[new_node.id] = new_node
```

`exact_match`

Δεδομένου ενός id, βρίσκει και επιστρέφει τον κόμβο με αυτό το id.

```python
def exact_match(self, key: int, start_node_id: int = None) -> Node  | None:
    """Finds and returns node with id same as a given key, if it exists."""

    node = self.get_node(start_node_id).find_successor(key)
    if node.id != key:
        print(f"Couldn't find node with id {key}.")
        return
    return node
```

range_query

Επιστρέφει όλους τους κόμβους σε ένα εύρος.

```python
def range_query(self, start: int, end:int, start_node_id: int = None) -> list[Node]:
    """Lists the nodes in the range [start, end]."""

    nodes_in_range = []
    first_node = self.get_node(start_node_id).find_successor(start)
    current = first_node

    # current id ∈ [start, end]
    while cw_dist(start, end) >= cw_dist(current.id, end):
        nodes_in_range.append(current)
        current = current.f_table[0][1]
        if (current == first_node):
            return nodes_in_range

    return nodes_in_range
```

get_item

Δεδομένου ενός κλειδιού, βρίσκει τον υπεύθυνο κόμβο για ένα item επιστρέφει τον υπεύθυνο κόμβο και το αντικείμενο με το κλειδί αυτό.

```python
def get_item(self, key: str, start_node_id: int = None) -> tuple | None:
    """Given a key, finds node responsible for
    an item and returns both in a tuple."""

    start_node = self.get_node(start_node_id)
    resp_node = start_node.find_successor(hash_func(key))
    if resp_node is not None:
        return (resp_node, resp_node.items[key])
    return
```

insert_item

Βρίσκει τον υπεύθυνο κόμβο για ένα item και καλεί τη συνάρτηση insert_item_to_node σε εκείνον, εισάγοντας το αντικείμενο.

```python
def insert_item(self, new_item: tuple, start_node_id: int = None) -> None:
    """Inserts an item (key, value) to the correct node of the network."""

    start_node = self.get_node(start_node_id)
    succ = start_node.find_successor(hash_func(new_item[0]))
    succ.insert_item_to_node(new_item)
```

delete_item

Βρίσκει τον υπεύθυνο κόμβο για ένα item και καλεί τη συνάρτηση delete_item_from_node σε εκείνον, διαγράφοντας το αντικείμενο.

```python
def delete_item(self, key: str, start_node_id: int = None, item_print=False):
    """Finds node responsible for key and removes the (key, value) entry from it."""

    start_node = self.get_node(start_node_id)
    start_node.find_successor(hash_func(key)).delete_item_from_node(key,item_print=item_print)
```

update_record

Βρίσκει τον υπεύθυνο κόμβο για ένα αντικείμενο, ελέγχει αν υπάρχει το αντικείμενο και ανανεώνει τα δεδομένα του.

```python
def update_record(self, new_item: tuple, start_node_id: int = None, print_item: bool = False) -> None:
    """Updates the record (value) of an item given its key."""

    start_node = self.get_node(start_node_id)
    responsible_node = start_node.find_successor(hash_func(new_item[0]))
    if new_item[0] in responsible_node.items:
        responsible_node.insert_item_to_node(new_item, print_item=print_item)
        return
    print(f"Could not find item with key {new_item[0]}")
```

insert_all_data

Εισάγει όλα τα αντικείμενα που έχουν συλλεχθεί από το dataset (csv αρχείο) στους υπεύθυνους κόμβους.

```python
def insert_all_data(self, dict_items: list[tuple], start_node_id: int = None) -> None:
    """Inserts all data from parsed csv into the correct nodes."""

    for dict_item in dict_items:
        self.insert_item(dict_item, start_node_id)
```

print_all_nodes

Εκτυπώνει όλους τους κόμβους του δικτύου.

```python
def print_all_nodes(self, items_print = False, finger_print=False) -> None:
    """Prints all nodes of the network"""

    sorted_nodes = sorted(list(self.nodes.items()))
    print([hex(sor_id[0]) for sor_id in sorted_nodes])
    for n in sorted_nodes:
        n[1].print_node(finger_print=finger_print, items_print=items_print)
```

knn_query

Επιστρέφει τους k κοντινότερους κόμβους ενός κόμβου.

```python
def knn(self, k: int, node_id: int, start_node_id: int = None) -> list[Node]:
    """Lists the k nearest nodes of node, given a specific id."""

    neighbours = []

    node = self.exact_match(key=node_id ,start_node_id=start_node_id)
    if node.id is None:
        return

    next_succ  = node.f_table[0][1]
    succ_hops = 0
    next_pred = node.pred
    pred_hops = 0

    while len(neighbours) < k:
        # Difference of next_succ and next_pred distance from node
        succ_pred_difference = (abs(node_id - next_succ.id) % HS) - (abs(node_id - next_pred.id) % HS)

        # Next successor is closer
        if succ_pred_difference < 0:
            neighbours.append(next_succ)
            next_succ = next_succ.f_table[0][1]
            succ_hops += 1
        # Next predecessor is closer
        elif succ_pred_difference > 0:
            neighbours.append(next_pred)
            next_pred = next_pred.pred
            pred_hops += 1

        # Equal distance
        else:
            if succ_hops <= pred_hops:
                neighbours.append(next_succ)
                next_succ = next_succ.f_table[0][1]
                succ_hops += 1
            else:
                neighbours.append(next_pred)
                next_pred = next_pred.pred
                pred_hops += 1

    return neighbours
```

Το αρχείο main.py

Στην αρχή της main, ορίζουμε κάποιες σταθερές. Πιο συγκεκριμένα, ορίζουμε το μέγεθος του κλειδιού της συνάρτησης κατακερματισμού μας (γνωστό και ως digest), το hashing space (το οποίο είναι προφανές ότι εξαρτάται από το μέγεθος του κλειδιού), το μέγεθος

```python
# Key size (bits)
KS = 4
# Hashing space
HS = 2**KS
# Successor list size
SLS = 3
# Node count
NC = 5
```

του successor list για το μέτρο κατά των massive nodes failure και το πλήθος κόμβων για τη δημιουργία του αρχικού δικτύου.

Η συνάρτηση main

Δημιουργία δικτύου

Αρχικά, δημιουργούμε ένα στιγμιότυπο της κλάσης Interface και το ονομάζουμε **interface**. Στη συνέχεια, δημιουργούμε ένα δίκτυο με τυχαίους κόμβους μέσω της συνάρτησης build_network. Σε σχόλια υπάρχει και ο κώδικας που θα χρησιμοποιούσαμε για να δημιουργήσουμε ένα δίκτυο από κόμβους με συγκεκριμένα id.

```
def main():
    interface = iff.Interface()

    # Create random network with NC nodes
    interface.build_network(NC)

    # Create predefined network with node ids
    #node_ids=[6, 5, 12, 10, 3, 14]
    #interface.build_network(node_count=len(node_ids), node_ids=node_ids)
```

Στη συνέχεια, ελέγχουμε τη σωστή λειτουργία των βασικών διαδικασιών.

Εισαγωγή δεδομένων

Αρχικά, διαβάζουμε το dataset μας και εισάγουμε τα αντικείμενα στους σωστούς κόμβους, τους οποίους μετά εκτυπώνουμε.

```
# Data insertion
items = iff.parse_csv("NH4_NO3.csv")
interface.insert_all_data(items.items())
interface.print_all_nodes(finger_print=True, items_print=False)
input("Press any key to continue...\n")
```

Κατά την εκτέλεση του παραπάνω τμήματος κώδικα, παρατηρούμε την έξοδο:

```
['0x0', '0x1', '0x6', '0x7']
Node ID: 0x0
Predecessor ID: 0x7
Successor list: ['0x1', '0x6', '0x7']
Finger table:
0x1 -> 0x1
0x2 -> 0x6
0x4 -> 0x6
0x8 -> 0x0

Node ID: 0x1
Predecessor ID: 0x0
Successor list: ['0x6', '0x7', '0x0']
Finger table:
0x2 -> 0x6
0x3 -> 0x6
0x5 -> 0x6
0x9 -> 0x0
```

```
Node ID: 0x6
Predecessor ID: 0x1
Successor list: ['0x7', '0x0']
Finger table:
0x7 -> 0x7
0x8 -> 0x0
0xa -> 0x0
0xe -> 0x0

Node ID: 0x7
Predecessor ID: 0x6
Successor list: ['0x0', '0x6']
Finger table:
0x8 -> 0x0
0x9 -> 0x0
0xb -> 0x0
0xf -> 0x0

Press any key to continue...
```

Αναζήτηση κλειδιού

Ξεκινώντας, παίρνουμε ένα τυχαίο κλειδί που ξέρουμε ότι βρίσκεται μέσα στα αντικείμενα. Έπειτα το ψάχνουμε, βρίσκουμε τον υπεύθυνο κόμβο και εκτυπώνουμε το κλειδί (πριν και μετά τον κατακερματισμό), το id του κόμβου, αλλά και το αντικείμενο με το κλειδί που ψάξαμε.

```
# Get random key from items
random_key = random.sample(sorted(items), 1)[0]
print(random_key)

# Key lookup
lookup = {"key": random_key, "start_node_id": None}
resp_node, item = interface.get_item(lookup["key"])
print(f"Looking up key: {lookup['key']} with hashed value: {hex(iff.hash_func(lookup['key']))}.\
    \nResponsible node: {hex(resp_node.id)}.\nItem: {item}\n")
input("Press any key to continue...\n")
```

Κατά την εκτέλεση του παραπάνω τμήματος κώδικα, παρατηρούμε την έξοδο:

```
8/27/2014_309
Looking up key: 8/27/2014_309 with hashed value: 0x9.
Responsible node: 0x0.
Item: {'Date': '8/27/2014', 'Block': 3, 'Plot': 309, 'Experimental_treatment': 'Alfalfa I', 'Soil_NH4': 6.0, 'Soil_NO3': 19.3}

Press any key to continue...
```

Ανανέωση αντικειμένου & διαγραφή κλειδιού

Στη συνέχεια, δοκιμάζουμε τις διαδικασίες ανανέωσης και διαγραφής κλειδιού και εκτυπώνουμε τα αποτελέσματα.

```
# Update record based on key
random_key = random.sample(sorted(items), 1)[0]
data = "This is a test string to demonstrate the update record function."
interface.update_record((random_key, data), start_node_id=None, print_item=True)
input("Press any key to continue...\n")

# Delete key
interface.delete_item(random_key, item_print=True)
input("Press any key to continue...\n")
```

Κατά την εκτέλεση των παραπάνω τμημάτων κώδικα, παρατηρούμε τις εξόδους:

```
Item with key 8/27/2014_309 before updating record:
{'Date': '8/27/2014', 'Block': 3, 'Plot': 309, 'Experimental_treatment': 'Alfalfa I', 'Soil_NH4': 6.0, 'Soil_NO3': 19.3}
Item with key 8/27/2014_309 after updating record:
This is a test string to demonstrate the update record function.

Press any key to continue...
```

```
Node before removing item with key 8/27/2014_309:
Node ID: 0x0
Predecessor ID: 0x7
Successor list: ['0x1', '0x6', '0x7']
Items in node: ['5/29/2014_105', '5/29/2014_111', '5/29/2014_113', '5/29/2014_305', '5/29/2014_311', '5/29/2014_409', '5/29/2014_414',
'6/27/2014_105', '6/27/2014_107', '6/27/2014_113', '6/27/2014_305', '6/27/2014_309', '7/24/2014_113', '7/24/2014_305', '7/24/2014_308',
 '7/24/2014_309', '7/24/2014_406', '7/24/2014_414', '8/27/2014_107', '8/27/2014_113', '8/27/2014_309', '8/27/2014_406', '11/9/2014_111'
, '11/9/2014_308', '11/9/2014_311', '11/9/2014_404', '11/9/2014_406', '11/9/2014_409', '4/17/2015_105', '4/17/2015_107', '4/17/2015_111
', '4/17/2015_113', '4/17/2015_305', '4/17/2015_308', '4/17/2015_309', '4/17/2015_404', '4/17/2015_406', '4/17/2015_409', '4/17/2015_41
4', '6/10/2015_105', '6/10/2015_111', '6/10/2015_308', '6/10/2015_406', '6/10/2015_414', '8/10/2015_105', '8/10/2015_107', '8/10/2015_1
11', '8/10/2015_305', '8/10/2015_308', '8/10/2015_311', '8/10/2015_404', '8/10/2015_409', '8/10/2015_414', '9/18/2015_107', '9/18/2015_
113', '9/18/2015_308', '9/18/2015_309', '9/18/2015_311', '9/18/2015_406', '9/18/2015_409', '9/18/2015_414', '10/30/2015_105', '10/30/20
15_113', '10/30/2015_308', '10/30/2015_311', '10/30/2015_409', '12/1/2015_111', '12/1/2015_113', '12/1/2015_305', '12/1/2015_308', '12/
1/2015_309', '12/1/2015_311', '12/1/2015_404', '12/1/2015_406']
```

```
Node after removing item with key 8/27/2014_309:
Node ID: 0x0
Predecessor ID: 0x7
Successor list: ['0x1', '0x6', '0x7']
Items in node: ['5/29/2014_105', '5/29/2014_111', '5/29/2014_113', '5/29/2014_305', '5/29/2014_311', '5/29/2014_409', '5/29/2014_414',
'6/27/2014_105', '6/27/2014_107', '6/27/2014_113', '6/27/2014_305', '6/27/2014_309', '7/24/2014_113', '7/24/2014_305', '7/24/2014_308',
 '7/24/2014_309', '7/24/2014_406', '7/24/2014_414', '8/27/2014_107', '8/27/2014_113', '8/27/2014_406', '11/9/2014_111', '11/9/2014_308'
, '11/9/2014_311', '11/9/2014_404', '11/9/2014_406', '11/9/2014_409', '4/17/2015_105', '4/17/2015_107', '4/17/2015_111', '4/17/2015_113
', '4/17/2015_305', '4/17/2015_308', '4/17/2015_309', '4/17/2015_404', '4/17/2015_406', '4/17/2015_409', '4/17/2015_414', '6/10/2015_10
5', '6/10/2015_111', '6/10/2015_308', '6/10/2015_406', '6/10/2015_414', '8/10/2015_105', '8/10/2015_107', '8/10/2015_111', '8/10/2015_3
05', '8/10/2015_308', '8/10/2015_311', '8/10/2015_404', '8/10/2015_409', '8/10/2015_414', '9/18/2015_107', '9/18/2015_113', '9/18/2015_
308', '9/18/2015_309', '9/18/2015_311', '9/18/2015_406', '9/18/2015_409', '9/18/2015_414', '10/30/2015_105', '10/30/2015_113', '10/30/2
015_308', '10/30/2015_311', '10/30/2015_409', '12/1/2015_111', '12/1/2015_113', '12/1/2015_305', '12/1/2015_308', '12/1/2015_309', '12/
1/2015_311', '12/1/2015_404', '12/1/2015_406']

Press any key to continue...
```

Άφιξη & αναχώρηση κόμβου

Έπειτα, δοκιμάζουμε τις διαδικασίες άφιξης και αναχώρησης κόμβου, εκτυπώνοντας το δίκτυο μετά από την κάθε διαδικασία.

```python
# Node Join
rnid = interface.get_id_not_in_net()
interface.node_join(rnid, print_node=True)
interface.print_all_nodes(finger_print=True)
input("Press any key to continue...\n")
```

```python
# Node Leave
rnid = interface.get_random_node().id
interface.node_leave(rnid, print_node=True)
interface.print_all_nodes(finger_print=True)
input("Press any key to continue...\n")
```

Κατά την εκτέλεση των παραπάνω τμημάτων κώδικα, παρατηρούμε τις εξόδους:

```
Creating and adding node 0x2 to the network...
['0x0', '0x1', '0x2', '0x6', '0x7']
Node ID: 0x0
Predecessor ID: 0x7
Successor list: ['0x1', '0x2', '0x6']
Finger table:
0x1 -> 0x1
0x2 -> 0x2
0x4 -> 0x6
0x8 -> 0x0

Node ID: 0x1
Predecessor ID: 0x0
Successor list: ['0x2', '0x6', '0x7']
Finger table:
0x2 -> 0x2
0x3 -> 0x6
0x5 -> 0x6
0x9 -> 0x0
```

```
Node ID: 0x2
0x4 -> 0x6
0x6 -> 0x6
0xa -> 0x0

Node ID: 0x6
Predecessor ID: 0x2
Successor list: ['0x7', '0x0']
Finger table:
0x7 -> 0x7
0x8 -> 0x0
0xa -> 0x0
0xe -> 0x0

Node ID: 0x7
Predecessor ID: 0x6
Successor list: ['0x0', '0x6']
Finger table:
0x8 -> 0x0
0x9 -> 0x0
0xb -> 0x0
0xf -> 0x0

Press any key to continue...
```

```
Node that will be removed from network:
Node ID: 0x2
Predecessor ID: 0x1
Successor list: ['0x6', '0x7', '0x0']
Items in node: []
```

```
Successor node before 0x2 leave:
Node ID: 0x6
Predecessor ID: 0x2
Successor list: ['0x7', '0x0']
Items in node: ['5/29/2014_107', '5/29/2014_406', '6/27/2014_111', '6/27/2014_308', '6/27/2014_404', '6/27/2014_409', '7/24/2014_105',
'7/24/2014_107', '7/24/2014_111', '7/24/2014_311', '8/27/2014_111', '8/27/2014_305', '8/27/2014_308', '8/27/2014_311', '8/27/2014_404',
 '8/27/2014_409', '8/27/2014_414', '11/9/2014_105', '11/9/2014_107', '11/9/2014_113', '11/9/2014_305', '11/9/2014_309', '11/9/2014_414'
, '6/10/2015_113', '6/10/2015_309', '6/10/2015_311', '6/10/2015_404', '6/10/2015_409', '8/10/2015_113', '9/18/2015_105', '9/18/2015_111
', '10/30/2015_111', '10/30/2015_305', '10/30/2015_309', '10/30/2015_406', '10/30/2015_414', '12/1/2015_105', '12/1/2015_409']

Successor node after 0x2 leave:
Node ID: 0x6
Predecessor ID: 0x1
Successor list: ['0x7', '0x0']
Items in node: ['5/29/2014_107', '5/29/2014_406', '6/27/2014_111', '6/27/2014_308', '6/27/2014_404', '6/27/2014_409', '7/24/2014_105',
'7/24/2014_107', '7/24/2014_111', '7/24/2014_311', '8/27/2014_111', '8/27/2014_305', '8/27/2014_308', '8/27/2014_311', '8/27/2014_404',
 '8/27/2014_409', '8/27/2014_414', '11/9/2014_105', '11/9/2014_107', '11/9/2014_113', '11/9/2014_305', '11/9/2014_309', '11/9/2014_414'
, '6/10/2015_113', '6/10/2015_309', '6/10/2015_311', '6/10/2015_404', '6/10/2015_409', '8/10/2015_113', '9/18/2015_105', '9/18/2015_111
', '10/30/2015_111', '10/30/2015_305', '10/30/2015_309', '10/30/2015_406', '10/30/2015_414', '12/1/2015_105', '12/1/2015_409']
```

```
['0x0', '0x1', '0x6', '0x7']
Node ID: 0x0
Predecessor ID: 0x7
Successor list: ['0x1', '0x6', '0x7']
Finger table:
0x1 -> 0x1
0x2 -> 0x6
0x4 -> 0x6
0x8 -> 0x0

Node ID: 0x1
Predecessor ID: 0x0
Successor list: ['0x6', '0x7', '0x0']
Finger table:
0x2 -> 0x6
0x3 -> 0x6
0x5 -> 0x6
0x9 -> 0x0
```

```
Node ID: 0x6
Predecessor ID: 0x1
Successor list: ['0x7', '0x0']
Finger table:
0x7 -> 0x7
0x8 -> 0x0
0xa -> 0x0
0xe -> 0x0

Node ID: 0x7
Predecessor ID: 0x6
Successor list: ['0x0', '0x6']
Finger table:
0x8 -> 0x0
0x9 -> 0x0
0xb -> 0x0
0xf -> 0x0

Press any key to continue...
```

Range & kNN query

Τέλος, εκτελούμε ερωτήματα εύρους και κοντινότερου γείτονα (σε τυχαίο κόμβο το 2ο) και εκτυπώνουμε τους αντίστοιχους κόμβους.

```python
# Range query
rq = {"start": 5, "end": 10}
print(f"Nodes in range: [{hex(rq['start'])}, {hex(rq['end'])}]:")
print([hex(n.id) for n in interface.range_query(rq["start"], rq["end"])])
input("Press any key to continue...\n")
```

```python
# kNN query
rnid = interface.get_random_node().id
kNN = {"k": 3,"key": rnid}
print(f"{kNN['k']} nearest neighbours of {hex(kNN['key'])}:")
print([hex(n.id) for n in interface.knn(kNN["k"], kNN["key"])])
input("Press any key to continue...\n")
```

Μετά την εκτέλεση των παραπάνω τμημάτων κώδικα, παρατηρούμε τις εξόδους:

```
Nodes in range: [0x5, 0xa]:
['0x6', '0x7']
Press any key to continue...

3 nearest neighbours of 0x1:
['0x0', '0x6', '0x7']
Press any key to continue...
```

Το αρχείο benchmarks.py

Συναρτήσεις του benchmarks.py

benchmark

Παράγει τα δεδομένα ώστε να διεξαχθούν τα αποτελέσματα των μετρήσεων.

```python
def benchmark(NC: int, results: dict) -> dict:
    interface = iff.Interface()

    # Build network with NC nodes
    build_start = perf_counter()
    interface.build_network(NC)
    build_end = perf_counter()

    # Insert all data to the network
    data_start = perf_counter()
    items = iff.parse_csv("NH4_NO3.csv")
    interface.insert_all_data(items.items())
    data_end = perf_counter()

    # Generate random numbers to use for benchmarking
    run_count = 10
    random_nodes = random.sample(sorted(interface.nodes), run_count*2)
    (leave_n, ex_match_n) = (random_nodes[:run_count], random_nodes[run_count:])

    search_keys = random.sample(range(HS), run_count)
    first_node = interface.get_node()
    join_nodes = []
    count = 0
    for i in range(HS):
        if i not in interface.nodes:
            count += 1
            join_nodes.append(i)
            if count >= run_count//2:
                break

    in_keys = []
    in_data = []
    up_data = []

    for i in range(run_count):
        in_keys.append(f"In key {i}")
        in_data.append(f"In data {i}")
        up_data.append(f"Updata {i}")
```

```python
# Start process benchmarking NC Nodes

# Insert key
print("Benchmarking insert key...")
in_key_start = perf_counter()
for index, key in enumerate(in_keys):
    interface.insert_item((key, in_data[index]), first_node.id)
in_key_end = perf_counter()

# Update record based on key
print("Benchmarking Update record based on key...")
up_key_start = perf_counter()
for key in in_keys:
    interface.update_record((key, up_data[index]), first_node.id)
up_key_end = perf_counter()

# Delete key
print("Benchmarking Delete key...")
del_key_start = perf_counter()
for key in in_keys:
    interface.delete_item(key=key, start_node_id=first_node.id)
del_key_end = perf_counter()
```

```python
# Key lookup
print("Benchmarking Key lookup...")
query_start = perf_counter()
for key in search_keys:
    first_node.find_successor(key)
query_end = perf_counter()

# Node join
print("Benchmarking Node join...")
join_start = perf_counter()
for key in join_nodes:
    interface.node_join(new_node_id=key)
join_end = perf_counter()

# Exact match
print("Benchmarking Exact match...")
ex_match_start = perf_counter()
for key in ex_match_n:
    interface.exact_match(key=key)
ex_match_end = perf_counter()
```

```python
# Node join
print("Benchmarking Node join...")
join_start = perf_counter()
for key in join_nodes:
    interface.node_join(new_node_id=key)
join_end = perf_counter()

# Exact match
print("Benchmarking Exact match...")
ex_match_start = perf_counter()
for key in ex_match_n:
    interface.exact_match(key=key)
ex_match_end = perf_counter()

# Node Leave
print("Benchmarking Node Leave...")
leave_start = perf_counter()
for key in leave_n:
    interface.node_leave(key)
leave_end = perf_counter()
```

```python
# Range query
print("Benchmarking Range query...")
range_start = perf_counter()
for key in search_keys:
    interface.range_query(key, (key+20) % HS )
range_end = perf_counter()

# kNN query
print("Benchmarking kNN query...")
knn_start = perf_counter()
for key in join_nodes:
    interface.knn(5, key)
knn_end = perf_counter()
```

Στο παρακάτω τμήμα κώδικα γίνεται ο υπολογισμός των αποτελεσμάτων των βασικών πράξεων, συναρτήσει των χρόνων που υπολογίστηκαν παραπάνω.

```
    #results['Build'][NC] = (build_end - build_start) * 1000
    #results['Insert all data'][NC] = (data_end - data_start) * 1000
    results['Insert key'][NC] = (in_key_end - in_key_start) * 1000 / run_count
    results['Delete key'][NC] = (del_key_end - del_key_start) * 1000 / run_count
    results['Update key'][NC] = (up_key_end - up_key_start) * 1000 / run_count
    results['Key lookup'][NC] = (query_end - query_start) * 1000 / run_count
    results['Node Join'][NC] = (join_end - join_start) * 1000 / (run_count/2)
    results['Node Leave'][NC] = (leave_end - leave_start) * 1000 / run_count
    #results['Massive Nodes\' failure'][NC] = (mnn_end - mnn_start)
    results['Exact match'][NC] = (ex_match_end - ex_match_start) * 1000 / run_count
    results['Range Query'][NC] = (range_end - range_start) * 1000 / run_count
    results['kNN Query'][NC] = (knn_end - knn_start) * 1000 / run_count
    return results
```

benchmark_all

Παράγει τα δεδομένα των μετρήσεων για διαφορετικό αριθμό κόμβων.

```
def benchmark_all() -> dict:
    answer = {
        #"Build": {},
        #"Insert all data": {},
        "Insert key": {},
        "Delete key": {},
        "Update key": {},
        "Key lookup": {},
        "Node Join": {},
        "Node Leave": {},
        #"Massive Nodes' failure": {},
        "Exact match": {},
        "Range Query": {},
        "kNN Query": {}
    }
    for i in range(20, 301, 40):
        print(f"\nBenchmarking {i} nodes...\n")
        answer = benchmark(i, answer)
    return answer
```

results_print & plot results

Τέλος, οι παρακάτω συναρτήσεις εκτυπώνουν τα αποτελέσματα και παράγουν τα plots των benchmarks.

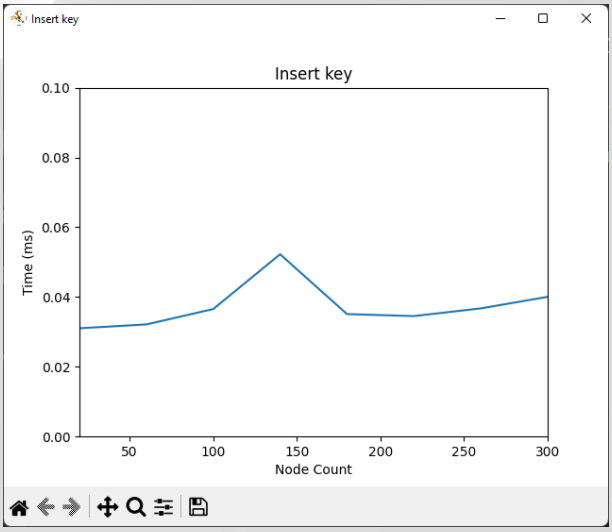
```python
def results_print(results: dict) -> None:
    for process in results.items():
        print(f"\n{process[0]} times:")
        for node_count, time in process[1].items():
            print(f"{process[0]} time for {node_count} nodes: {time}")

def plot_results(results: dict) -> dict[plt.plot]:
    # X axis = Node count, Y axis = Time
    plots = {}
    for process in results.items():
        plt.figure(process[0])
        plots[process[0]] = plt.plot([nc[0] for nc in process[1].items()],\
            [t[1] for t in process[1].items()])
        if process[0] != "Node Join" and process[0] != "Node Leave":
            plt.axis([20, 300, 0, 0.1])
        plt.xlabel("Node Count")
        plt.ylabel("Time (ms)")
        plt.title(process[0])
        plt.show()

    return plots
```

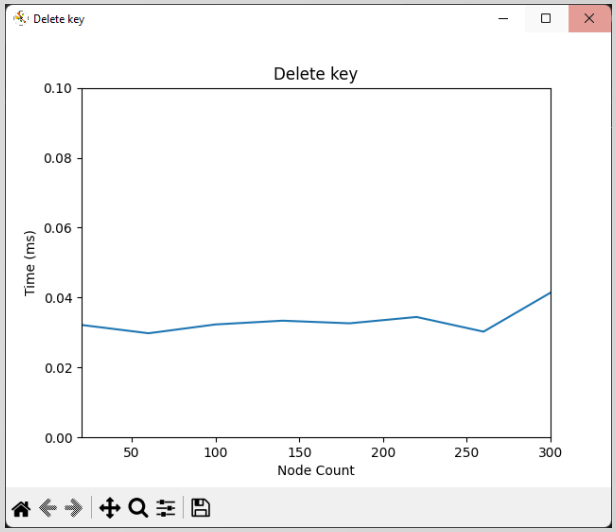
Εκτέλεση των benchmarks & εμφάνιση γραφημάτων

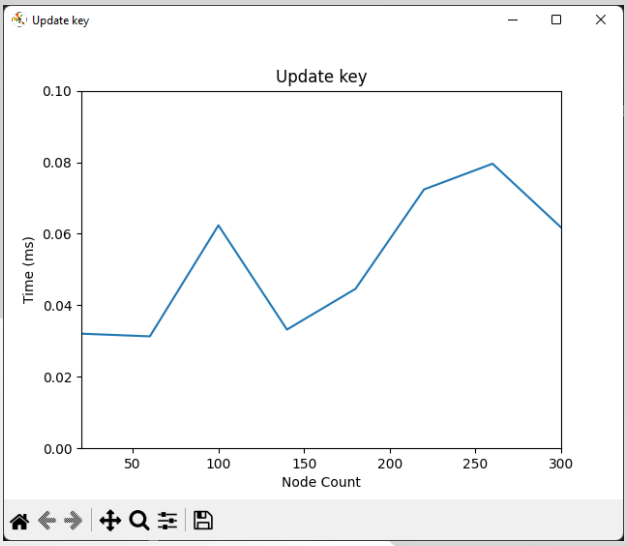
```python
results = benchmark_all()
plot_results(results)
```

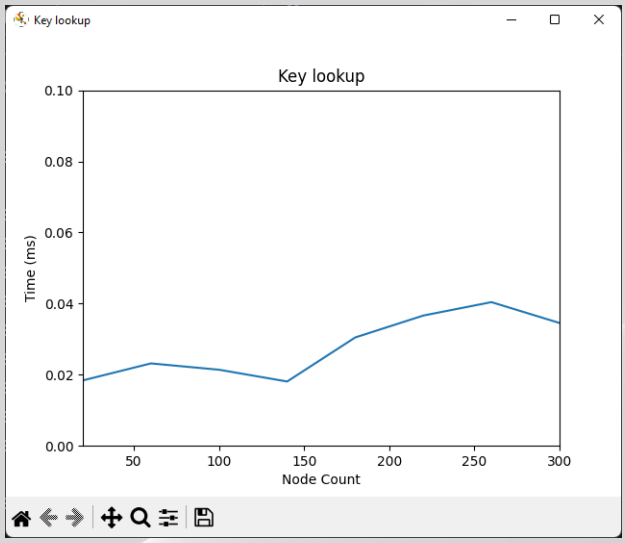
Πειραματική μελέτη

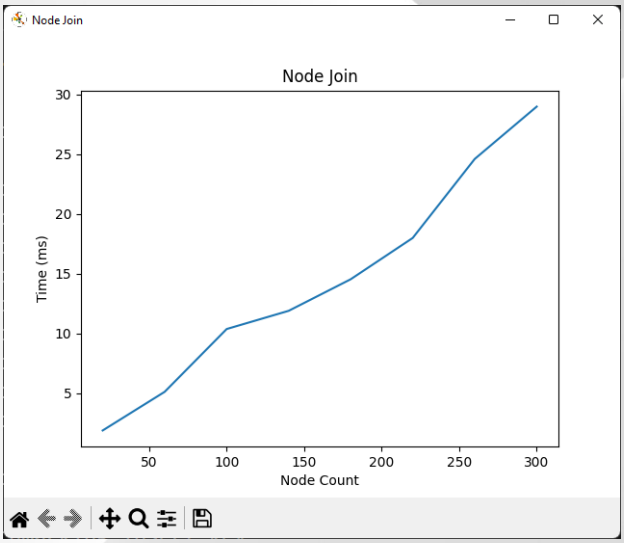
Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης των ζητούμενων διαδικασιών σε δίκτυα με διάφορους αριθμούς κόμβων (από 20 μέχρι 300). Για την εκτέλεση των benchmarks, το **KS** ορίστηκε στο **10** (άρα το hashing space είχε μέγεθος 1024), ενώ κάθε διαδικασία εκτελέστηκε 10 φορές για κάθε δίκτυο και πήραμε τον **μέσο όρο εκτέλεσης**.

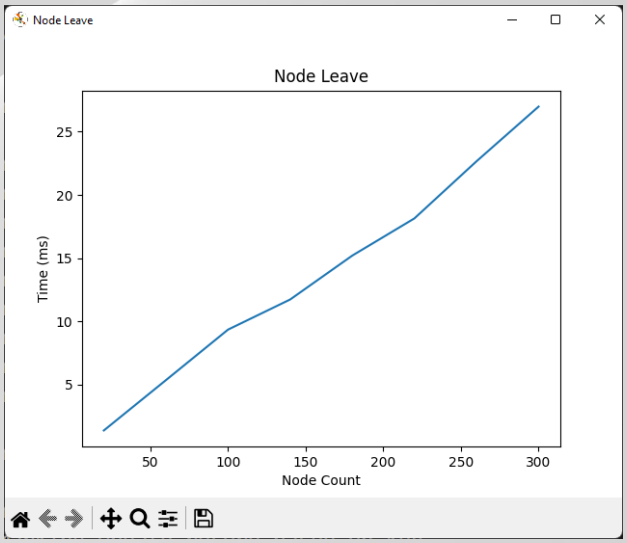


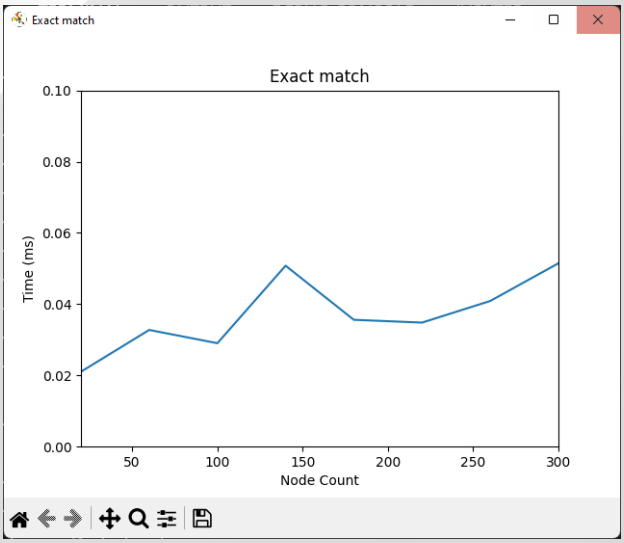


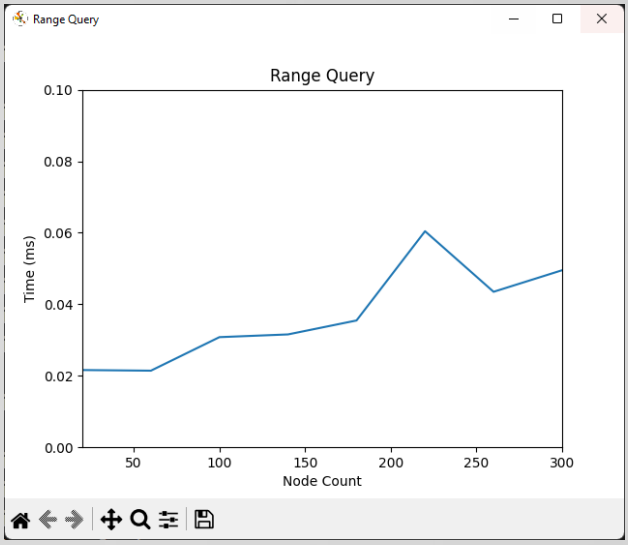

Update key
Update key
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300

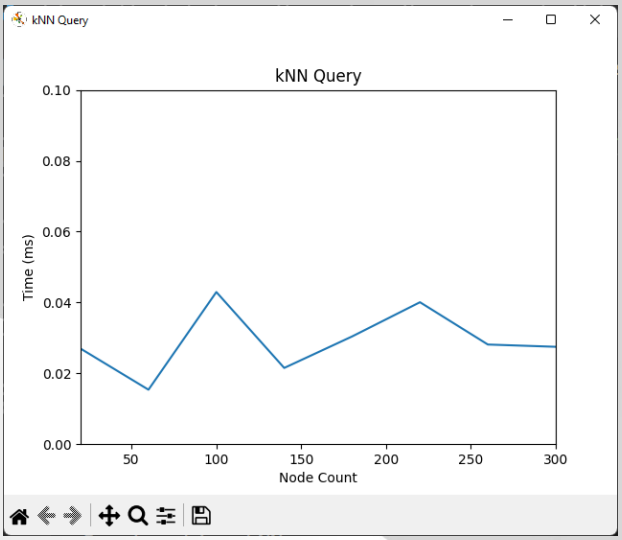

Key lookup
Key lookup
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300


Node Join
Node Join
Time (ms)
Node Count
5
10
15
20
25
30
50
100
150
200
250
300


Node Leave
Node Leave
Time (ms)
Node Count
5
10
15
20
25
50
100
150
200
250
300


Exact match
Exact match
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300


Range Query
Range Query
Time (ms)
Node Count
0.00
0.02
0.04
0.06
0.08
0.10
50
100
150
200
250
300



Για το **range query** χρησιμοποιήθηκαν τυχαίες εμβέλειες μεγέθους 20, ενώ για το **kNN** χρησιμοποιήθηκαν τυχαίοι κόμβοι με k = 5.

* Τα πειράματα διεκπεραιώθηκαν σε φορητό υπολογιστή με επεξεργαστή Intel Core i7 8550u (15W, 1.8GHz Base Clock, Benchmarking Clocks: 3.1-3.3GHz).*

Συμπέρασμα

Από τα παραπάνω γραφήματα, παρατηρούμε ότι οι **μόνοι** χρόνοι που επηρεάστηκαν ουσιαστικά από την αύξηση του αριθμού των κόμβων δικτύου είναι οι χρόνοι εισόδου και εξόδου κόμβου στο δίκτυο. Αυτό είναι λογικό, καθώς **δε χρησιμοποιούμε πολυνημάτωση** ώστε ο κάθε κόμβος να μπορεί να τρέχει σε άλλο νήμα, κάτι που όμως στον πραγματικό κόσμο **αποφεύγεται**, καθώς το φορτίο μοιράζεται στους υπολογιστές-κόμβους. Το σημαντικό είναι ότι οι χρόνοι εκτέλεσης των υπόλοιπων βασικών πράξεων **δεν παρατηρούν** κάποια αξιοσημείωτη **αύξηση** και έτσι το σύστημα παραμένει γρήγορο ανεξαρτήτως του αριθμού των κόμβων.