Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα 2017-2018
Μέρος 2ο

Αργυρός Ιωάννης - 1115201200009
Γιαννούδης Αστέριος - 1115201200025
Κολυβάς Κωνσταντίνος - 1115201200066

Γλώσσα: C++11

Μεταγλώττιση:

```
$ cd build
$ cmake ..
$ make ngrams
```

Εκτέλεση:

```
$ ./ngrams -i <init_file> -q <query_file>
```

Μεταγλώττιση και εκτέλεση των Unit Tests

```
$ make basic_tests
$ ./basic_tests
```

Καθάρισμα φακέλου build :

```
sh clean.sh
```

1. Linear Hash Table:

Αποτελεί τα “παιδιά” του κόμβου-ρίζα του trie και του static_trie (βλ. 4.3).

1.1. Δομή:

Είναι template κλάση η οποία αποτελείται από έναν πίνακα από δείκτες σε mstd::vector. Τα παιδιά εντός του κάθε bucket διατηρούνται ταξινομημένα. Έτσι, για τον έλεγχο ύπαρξης κάποιας λέξης, γίνεται δυαδική αναζήτηση στο κατάλληλο bucket.

1.2. Εισαγωγή:

Κατά την εισαγωγή, ελέγχεται αν ήδη υπάρχει στο hash table, στην οποία περίπτωση δεν γίνεται κάποια αλλαγή. Αν δεν υπήρχε, τότε πρώτα ελέγχεται αν η εισαγωγή του νέου κόμβου θα προκαλούσε το load factor του hash table να περάσει το 90%, στην οποία περίπτωση, ακολουθείται η γνωστή μέθοδος “χωρισμού” του κατάλληλου bucket. Είτε, εν τέλει, χρειάστηκε να γίνει χωρισμός, είτε όχι, το στοιχείο εισάγεται στην κατάλληλη θέση του κατάλληλου bucket.

1.3. Αναζήτηση-Διαγραφή:

Και στις δύο αυτές λειτουργίες, αρχικά ελέγχεται η ύπαρξη της λέξης προς αναζήτηση. Αν υπάρχει, τότε επιστρέφεται/διαγράφεται αντίστοιχα, διαφορετικά δεν γίνεται κάποια αλλαγή.

2. Bloom Filter:

Χρησιμοποιείται για να ελεγχθεί αν κάποιο ngram έχει ήδη βρεθεί κατά ένα Query.

2.1. Δομή:

Χρησιμοποιεί ένα bit vector μεγέθους 34816 και 23 hash functions. Με τις συγκεκριμένες τιμές, τα small και medium datasets δεν έχουν λάθη. Σε περίπτωση λάθους κατά ελέγχους με άλλα αρχεία, θα χρειαστεί να αλλαχθούν αυτές οι τιμές. Οι τιμές βρέθηκαν μέσω ελέγχων και τη χρήση της συνάρτησης πιθανότητας false positive.

2.2. Hash Function(s):

Για την εύρεση των bit που θα ενεργοποιηθούν, χρησιμοποιείται η murmur 3 hash function η οποία παράγει 2 αποτελέσματα ($h_1 - h_2$).

Έπειτα, χρησιμοποιείται ο τύπος:

$h_i = h_1 + i\,x\,h_2 + i^2 \; mod\; m,\; i = 1,\; 2,\; ...,\; k$ και m το μεγεθος του *bit vector*

(source)

για να βρεθούν όλα τα k indices.

2.3. Αναζήτηση - Εισαγωγή:

Για τον έλεγχο ύπαρξης ενός ngram στο bloom filter, υπολογίζονται k indices και ελέγχεται αν το bit σε αυτές τις θέσεις ήταν ενεργοποιημένο. Αν έστω και ένα bit δεν ήταν ενεργοποιημένο, τότε το ngram δεν υπάρχει. Αν όλα ήταν ενεργοποιημένα, τότε ίσως υπάρχει. Σε κάθε περίπτωση, διατηρείται η πληροφορία για το αν το ngram υπήρχε και ενεργοποιούνται τα bits σε όλα τα k indices.

3. Top K:

Για την λειτουργικότητα της εύρεσης των K πιο συχνά εμφανιζόμενων απαντήσεων μεταξύ των ερωτημάτων μίας ριπής χρησιμοποιήθηκαν οι δομές linear_hash_int, pair και minHeap. Η διαδικασία εκτελείται στη συνάρτηση print_and_topk στο αρχείο source/main.cpp.

3.1. Pair:

Είναι η δομή που προσομοιώνει το ζευγάρι <λέξη,συχνότητα εμφάνισης λέξης> . Χρησιμοποιήθηκε για να φτιάξουμε hashtable από pairs.

3.2. Linear hash int:

Η δομή αυτή μοιάζει πολύ με την linear_hash που χρησιμοποιήθηκε για την αποθήκευση των παιδιών της ρίζας του trie. Η βασική διαφορά είναι πως εργάζεται με pairs και γνωρίζει ανά πάσα στιγμή ποία είναι η μέγιστη συχνότητα εμφάνισης λέξης από τις λέξεις που προσπάθησαν να εισαχθούν στο hashtable.

3.3. Min Heap:

Κλασική δομή min heap υλοποιημένη με πίνακα και επαναληπτική heapify. Προσφέρει τη δυνατότητα ταξινόμησης με τη μέθοδο heap sort. Η ταξινόμηση γίνεται στον εσωτερικό πίνακα του minHeap και συνεπώς χαλάει τις ιδιότητες του σωρού.

3.4. Λειτουργία:

α) Στο ερώτημα αυτό χρησιμοποιούμε το hashtable για να εισαγάγουμε μία μία τις απαντήσεις κατά την εκτύπωσή τους, τις οποίες παίρνουμε από την ουρά των απαντήσεων μετά το πέρας της ριπής. Με την ολοκλήρωση της δημιουργίας του hashtable γνωρίζουμε τη συχνότητα της πιο συχνά εμφανιζόμενης απάντησης. Έστω max_freq αυτή.

β) Δημιουργούμε πίνακα από vectors πλήθους max_freq. Σε κάθε i vector τοποθετούμε το pair από το hashtable που έχει συχνότητα i. Το αποτέλεσμα θα είναι κάπως έτσι :

```
  0   1     2          3                 max_freq
[ [ ], [ ], [ a ], [ b, c, d ], [ ], …, [ foo ] ]
```

γ) Ξεκινώντας από το τέλος του παραπάνω πίνακα, χρησιμοποιούμε το minHeap και την heapsort για να ταξινομήσουμε το εκάστοτε vector και εμφανίζουμε από την αρχή του κάθε vector τις απαντήσεις μέχρις ότου να συναντήσουμε K στοιχεία. Στην χειρότερη περίπτωση θα ταξινομήσουμε K vectors (του ενός στοιχείου μόνο).

δ) Τελικά θα έχουμε βρει τις K πιο συχνά εμφανιζόμενες απαντήσεις σε μία ριπή, ταξινομημένες με φθίνουσα σειρά εμφάνισης και με αλφαριθμητική αύξουσα σειρά λέξης για εκείνες που έχουν την ίδια συχνότητα. Αν n είναι όλες οι απαντήσεις μιας ριπής, x οι διαφορετικές και K οι ζητούμενες, η διαδικασία αυτή γίνεται σε O(n + x + x + klogk) = O(n + klogk).

4. Static Trie:

4.1. Δομή:

Για την υλοποίηση του στατικού trie, έχουν χρησιμοποιηθεί οι δομές static_trie, static_root_node και static_node, οι οποίες είναι θηγατρικές κλάσεις των trie, static_node και trie_node αντίστοιχα.

4.2. Συμπίεση:

Η συμπίεση επιτεύχθηκε με μια κατά βάθος προσπέλαση του αρχικού trie. Κατά την διαδικασία αυτή, ο κόμβος στον οποίο βρισκόμαστε πρόκειται να συμπιεστεί έχει ακριβώς ένα παιδί, τότε “κλέβει” την απαραίτητη πληροφορία από το παιδί αυτό (λέξη και παιδιά) και συνεχίζει την διαδικασία μέχρις ότου δεν έχει ακριβώς ένα παιδί. Ταυτόχρονα, ενημερώνεται και ο πίνακας των short ακεραίων με το μήκος και το πλήθος των λέξεων έως εκείνη τη στιγμή. Οι μη συμπιεσμένοι κόμβοι παραμένουν αμετάβλητοι, εκτός από τη δημιουργία του κατάλληλου πίνακα.

4.3. Αναζήτηση:

Επειδή δεν είναι δυνατόν να γίνει δυαδική αναζήτηση σε ολόκληρη τη συμβολοσειρά ενός συμπιεσμένου κόμβου, η συνάρτηση static_node::get_word(int index) επιστρέφει την λέξη που βρίσκεται στη θέση <index>. Η συνάρτηση static_bsearch, χρησιμοποιεί την get_word(0) για να ελέγξει την πρώτη λέξη του κάθε κόμβου. Έπειτα, ελέγχεται αν ο κόμβος που βρέθηκε είναι συμπιεσμένος και διαθέτει κι άλλη πληροφορία προς αναζήτηση, αλλιώς συνεχίζουμε σε επόμενο κόμβο.

4.4. Μετρήσεις:

Για τις μετρήσεις μεταξύ ενός κανονικού trie κι ενός στατικού trie, εκτελέσαμε τα δοσμένα αρχεία για το στατικό trie (smal_static.init κλπ) και με τα δύο είδη trie. Συγκεκριμένα, στο μικρό αρχείο (small_static.init) δεν υπήρχε αισθητή διαφορά στον χρόνο, ενώ στην εκτέλεση του μεσαίου

μεγέθους αρχείο (medium_static.init) παρατηρήθηκε διαφορά της τάξης των 10 δευτερολέπτων στα δικά μας μηχανήματα. Ένα αναμενόμενο αποτέλεσμα, καθώς το στατικό trie κερδίζει στο κομμάτι του χώρου παρά του χρόνου.