Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Project Report

Παναγιώτης Μαυρίδης 1115201300097
Νίκος Καραδήμας 1115201300061
Μιχαήλ Κοντούλης 1115201300073

Table of Contents

Part 1

Σχεδιασμός και Σχεδιαστικές Επιλογές

Για την βασική υλοποίηση του Trie έχουμε υλοποιήσει τις 3 βασικές functions

Add() ,Delete() , search()

Δομές

Index
Trie_Node
Trie_Node ** Total_NG_Matrix (Βοηθητική δομή για search)
Trie_Node ** N_GramMatrix (Βοηθητική δομή για search)

Index

Περιέχει τον Trie_Node * root και την Init_And_Work()

Trie_Node

-Trie_Node * children ο οποίος αποθηκεύει τα πεδία του Node σε ένα array το οποίο διατηρεί το locality, στο οποίο αρχικοποιούμε όλες τις θέσεις του, ακόμα και αν είναι “κενές”
-int state ο οποίος χρησιμεύει για να ξεχωρίζουμε αν ένα Node περιέχει small ή big word ή το αν είναι “κενή” η θέση.
-int numberOfChildren κρατάει τον αριθμών των παιδιών που έχει ο Node, και χρησιμεύει στην αποφυγή search κενής θέσης όταν κάνουμε add και βοηθάει και στην υλοποίηση τις binarySrearch.
-int sPrinted οποίος χρησιμεύει για την αποφυγή εκτύπωσης διπλότυπων.
-int isFinal δηλώνει το αν είναι final λέξη ενός ngram
-char word[SMALL_WORD_SIZE] αποθηκεύουμε τις λέξεις με μήκος μικρότερο του SMALL_WORD_SIZE εδώ γιατί γίνετε πιο γρήγορα η εκτέλεση γιατί διατηρείται το locality.
-char *bigWord αποθηκεύουμε τις λέξεις με μήκος μεγαλύτερο του SMALL_WORD_SIZE εδώ για να υποστηρίζουμε όλες τις λέξεις ανεξαρτήτως μεγέθους.

Total_NG_Matrix

Matrix που κρατάει pointers, σε nodes του Trie μας για την ανάθεση των isPrinted flags. Για την αποφυγή διπλοτύπων.

N_GramMatrix

Matrix που κρατάει pointers, σε nodes του Trie μας για να τυπώνει τα ngrams που βρίσκει.

Add

Η add πρώτα με τη βοήθεια ενός ορίσματος int choice που το περνάμε όταν καλούμε την συνάρτυση ξεχωρίζει αν την καλέσαμε απο init ή απο το work αρχειο. Αν είναι απο init διαβάζει κανονικά τις λέξεις που τις έχουν περαστεί και τις προσθέτει στο δέντρο, αν είναι απο work προσπερνάει την πρώτη λέξη γιατί θα είναι το αναγνωριστικό A το οποίο δεν είναι κιομματι του ngram. Έπειτα ξεκινάει από το root και ψάχνει ένα ένα τα παιδιά του σειριακά. Οι συγκρύσεις διευκολύνονται χάρης στο int state που έχει το κάθε node,γιατί ξέρουμε αν θα συγκρύνουμε τα περιεχόμενα των word ή των bigword. Αν βρεί ότι το node δεν έχει πίνακα με παιδιά τότε τον αρχικοποιεί πριν συνεχίσει. Αν δε βρει την επόμενη λέξη στο n-gram που δόθηκε την προσθέτει και συνεχίζει το ίδιο μέχρι να τελειώσει το ngram που δόθηκε. Αν βρει ότι υπάρχει τότε απλα πέρνει τον επόμενο Node ως current και συνεχίζει. Αν ο node που προσθέθηκε ή είδαμε ότι υπάρχει ήδη αντιστοιχεί στην τελευταία λέξη του ngram, τότε του προσθέτουμε και το isFinal = true. Έπειτα τελειώνει η συνάρτηση.

Delete

Η delete ξεκινάει από το root και ψάχνει ένα ένα τα παιδιά του κάθε node σειριακά(το αλλάξαμε αργότερα σε binary). Αν δε βρεί την επόμενη λέξη στο n-gram που δόθηκε σταματάει και επιστρέφει με αποτυχία.Κάθε φορά που βρίσκει μια νέα λέξη που ταιριάζει στο n-gram που θέλουμε να διαγράψουμε, την προσθέτει σε μια βοηθητική δομή τις συνάρτησης Trie_Node *traversed[countWords]. Αν βρεί και την τελευταία λέξη τότε κοιτάει αν έχει παιδια, και οπότε ανηκει σε ένα μεγαλύτερο n-gram και σε αυτή την περίπτωση απλά βγάζει το final flag. Αλλιώς σβήνει το node και οποιοδήποτε απο τους γονείς του που δεν είναι κομμάτι άλλου ngram. Αυτό γίνεται με το traversed που έχει κρατήσει όλα του nodes που αποτελούν το ngram. Πηγαίνοντας ένα ένα προς τα πίσω, τεσκάρουμε άν το node είναι Final με το flag isFinal και αν δεν είναι το σβήνουμε απο το Trie. Αλλιώς μόλις, ένα Node είναι Final τότε σταματάει η εκτέλεση και τελειώνει η Delete.

Search

Η search αποτελείται από δύο loops. Χρησιμοποιεί δύο pointers, οι οποίοι διασχίζούν το n-gram το οποίο ψάχνουμε στο Trie. Ό πρώτος pointer δηλώνει την αρχή του n-gram και τον μεταφέρουμε να δείχνει στην επόμενη λέξη σε καθε επανάληψη του εξωτερικού loop, ενώ ο δεύτερος δείχνει στην επόμενη λέξη σε κάθε επανάληψη του εσωτερικού loop ,και όταν φτάσει στο σημείο που θα τερματίσει το εσωτερικού loop, δείχνει όπου και ο πρώτος pointer. Όταν ο δεύτερος pointer βρει μια λέξη από το Query στο δέντρο την αποθηκεύει στο Ngram_Matrix και στο Total_NG_Matrix . Αν το node είναι final και δεν έχει ήδη τυπωθεί τότε τυπώνει μετά από κατάλληλη επεξεργασία για σωστό formatting ,απευθείας στην έξοδο τα περιεχόμενα του Ngram_Matrix το οποίο θέτει και τα isPrinted flags. Αν δεν βρεί λέξη που αντιστοιχεί στο ngram ή τελειώσει το εσωτερικό loop, τότε κάνει clean μόνο τον Total_NG_Matrix.

Στο τέλος του προηγράμματος κάνει clean και τους δύο πίνακες.

Testing and Results

Computer 1

CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Memory: 8gb

	Small
Static first test	8,46 sec
Static second test	8,39 sec
Static third test	8,43 sec
Static fourth test	8,42 sec

Part 2

Σχεδιασμός και Σχεδιαστικές Επιλογές

Δομές

BloomFilter

Δομη για την υλοποιηση του Bloomfilter.

Info_Hash

Δομη που περιέχεται σε κάθε Trie_Node . Ωστόσο στην ουσία υπάρχει μόνο στο root. Κάθε Info_Hash περιέχει buckets που το καθένα περιέχει ένα table από Trie_Nodes (στην ουσία το πρώτο tier του δέντρου). Μετά συνεχίζει το δέντρο να αναπτύσσεται από τα παιδιά των κόμβων που υπάρχουν στο table του κάθε bucket κανονικά χωρίς να ορίζουν αυτή την Info_Hash δομή τους (NULL pointer για όλα τα άλλα Trie_Nodes εκτός του root).

Buckets

Η δομή των buckets έχει μέσα πληροφορίες για τον αριθμό των nodes που έχει μέσα το καθένα αλλά και μεταβλητές για την υλοποίηση του αλγορίθμου της εκφώνησης . (splits,doubling …) . Περιέχει επίσης τον πίνακα με τα Trie_Nodes του πρώτου tier.

Updated Δομές

Trie_Node

-Τα όλες οι συναρτήσεις μας χρησιμοποιούν Binary_Search πλέον

Dynamic

Bloom Filter

Βασισμένοι στο paper[που αναφέρουμε στη βιβλιογραφία] η υλοποίηση του Bloom Filter χρειάζεται μόνο 2 hash functions για την επιτυχή λειτουργία της.Με μόνο 2 hash functions ,η υλοποίηση δεν χάνει καθόλου false positive probability. Έτσι γλιτώνουμε πού υπολογιστική δύναμη και πιθανώς λιγότερη τύχη στην πράξη. Για την δημιουργία αυτών των 2, χρησιμοποιούμε την murmur hash function η οποία επιστρέφει 2 128bit αριθμούς τους οποίους χρησιμοποιούμε σαν το αποτέλεσμα. Επίσης σύμωνα με το paper για υλοποιούμε extended double hashing

στο οποίο χρησιμοποιούμε μια ανεξάρτητη function (pow(i^3)) η οποία αυξάνει την flase positive πιθανότητα να είναι ίδια με ένα κανονικό bloom filter.
Η υλοποίηση του bit vector γίνεται ως ένας array απο integers αλλα η λειτουργεικοτητά του εκτελείται κανονικά στο επίπεδο των bits. Ο λόγος που το κάνουμε αυτό είναι ότι σύμφωνα με το standard (C99 6.2 5p5)

- "A "plain" int object has the natural size suggested by the architecture of the execution environment."

Οπότε συνήθως είναι πιο αποδοτικό.

Hash Table

Το Hash table έχει υλοποιηθεί σύμφωνα με τις διαφάνειες που δώθηκαν και με αυτές τις διαφάνειες στη βιβλιογραφία Το hashing γίνεται με τη murmur. Έχει διατηρηθεί το locality του Trie , γιατί αποθηκεύουμε τα Trie_Nodes μες στα Buckets.

Top-K

Για την υλοποίηση του Top-K χρησιμοποιήσαμε 2 δομές , την Heap και την Node.
Η Heap περιέχει εκτός από πληροφορίες για το πλήθος των δεδομένων που εισέρχονται σε αυτή και μια δομή Node. Αυτή η δομή περιέχει όλες τις πληροφορίες για τα δεδομένα που βάζουμε , όπως τη συχνότητα του κάθε entry (int frequency) αλλά και την τιμή του (char *string) . Κατα την αναζήτηση τοποθετούνται στη δομή όλα τα ngrams που θέλουμε με σωστό format . Στην διαδικασία της εισαγωγής πάντα διατηρείται η ιδιότητα του σωρού (πατέρας μεγαλύτερος από παιδιά). Αφού βρεθούμε σε “F” γραμμή του αρχείου ελέγχουμε πόσα top θέλουμε και εκτελούμε ένα sort στον πίνακα μας , αρχικά με βάση τα frequencies των Ngrams στα πρώτα K στοιχεία (όχι σε όλο τον πίνακα) , και στην συνέχεια ελέγχουμε τα ngrams με ίδιο frequency για να ταξινομηθούν και με βάση το string value τους . Τέλος εκτύπωνουμε τις K
Θέσεις του πίνακα μας και κάνουμε reset τον πίνακα για τις επόμενες εντολές.

Παρατηρήσεις σχετικά με Sorting:

Έγιναν πειράματα με διαφορετικές μεθόδους ταξινόμησης , όπως bubblesort , insertsort,selectionsort,quicksort. Οι μεγάλες διαφορές παρατηρήθηκαν σε insertsort και selectsort όπου οι χρόνοι μειώνονται αρκετά(τάξης των 15 seconds) . Οι αλγόριθμοι των bubblesort και quicksort πετυχαίνουν περίπου τους ίδιους χρόνους . Στην τελική έκδοση έχει χρησιμοποιηθεί quicksort.

Static

Η μετατροπή από dynamic σε static γίνεται με την βοήθεια μιας δομής Path_To_Be_Compressed. Κατά το πέρασμα του δέντρου αποθηκεύουμε σε αυτή την δομή δείκτες στα nodes τα οποία ανιχνεύουμε ότι πρέπει να συμπιεστούν σε ένα super node.

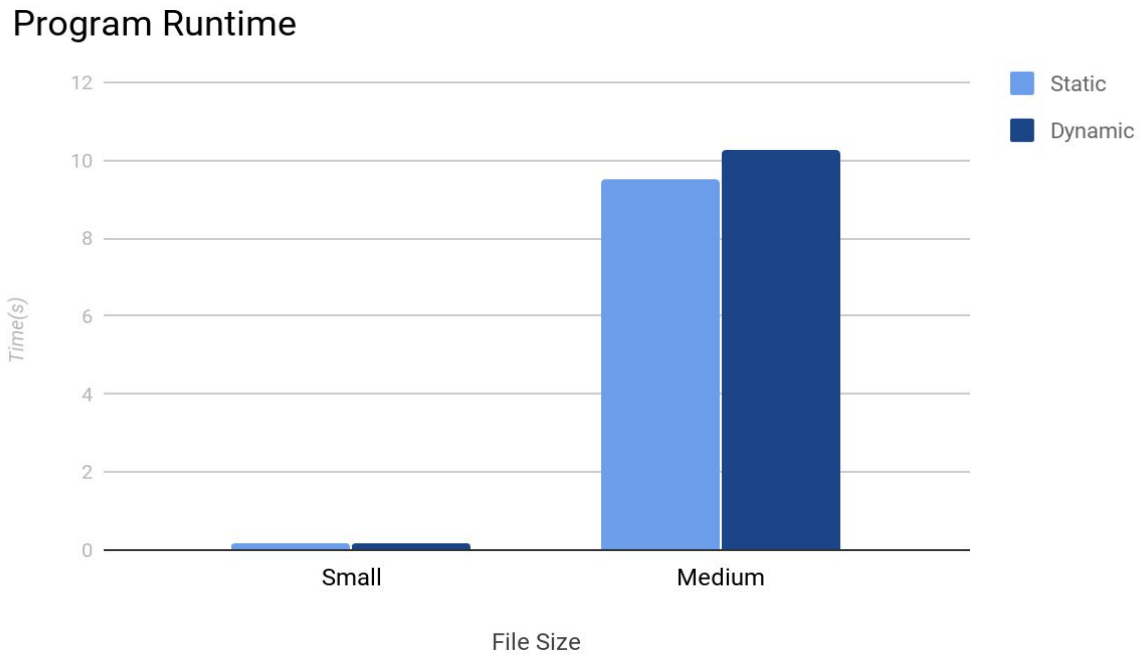
Αφού βρούμε όλα τα nodes που θα συμπιεστούν μαζί, ανατρέχουμε τη δομή αυτή και ένα ένα μεταφέρουμε τα δεδομένα των nodes στο super node και τα σβήνουμε. Στο τέλος μεταφέρουμε τα παιδιά του τελευταίου (αν υπάρχουν) στο super node.

Testing and Results

Computer 1

CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Memory: 8gb

	Small	Medium
Static	16s	9m 53s
Dynamic	16s	10m 24s



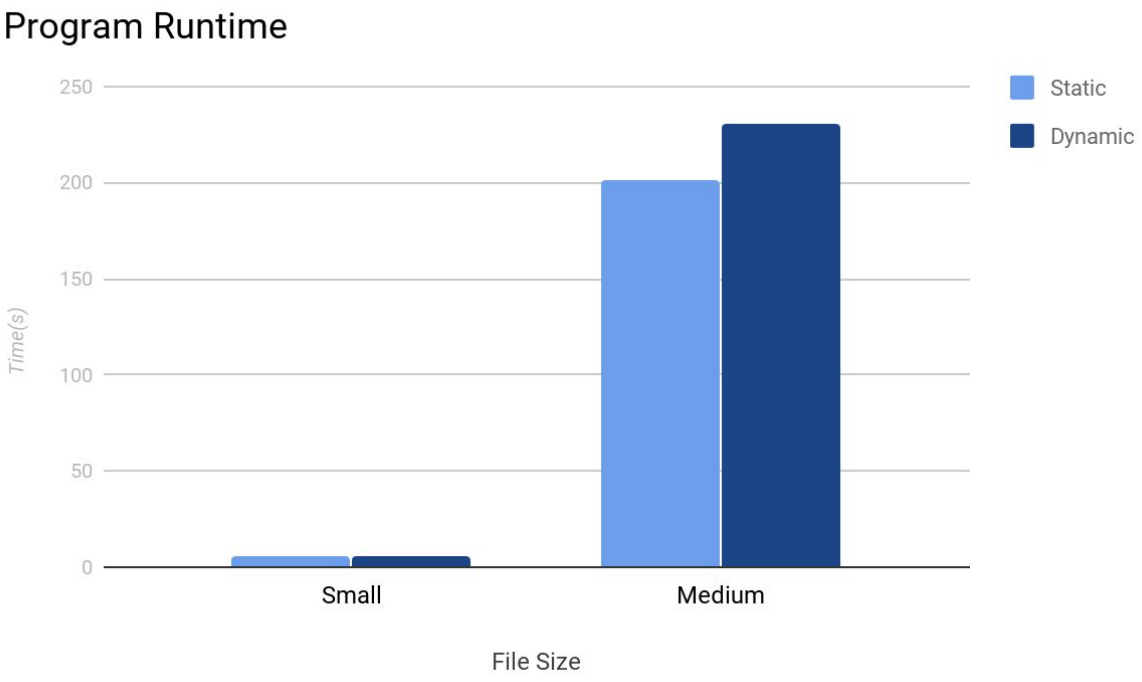
Charts

Computer 2

CPU: Intel® Core™ i5-2500K CPU @ 3.30GHz × 4
Memory: 8gb

	Small	Medium
Static	59.s	3m 35s
Dynamic	6.2s	4m 25s

Charts



Testing Conclusions

Παρατηρούμε από τα διαγράμματα ότι υπάρχει μία μικρή διαφορά στους χρόνους για τα static και dynamic tries . Επίσης τα αποτελέσματα των διαγραμμάτων έχουν προκύψει μετά από εκτέλεση των προγραμμάτων 10 φορές για το κάθε είδος (static, dynamic) και έχει παρουσιαστεί ο μέσος όρος των μετρήσεων αυτών.

Part 3

Σχεδιασμός και Σχεδιαστικές Επιλογές

Δομές

Parrallel_Printing_Array ---> Helper Matrix , αποθηκεύει αποτελέσματα από κάθε Query
Σε ξεχωριστή θέση για το κάθε ένα

Job_Static ---> Δομή για τα jobs . Στην ουσία την χρησιμοποιούμε ως wrapper για το τι δουλειά θα βάζουμε στους scheduler.

Scheduler ---> Δομή υλοποίησης ουράς των Jobs. Περιέχει μία ουρά στην οποία τοποθετούνται jobs.

Στο 3 part του project κληθήκαμε να υλοποιήσουμε παραλληλοποιήση στη δομή που είχαμε δημιουργήσει στα προηγούμενα parts. Για την επίτευξη του δημιουργήσαμε καινούργιες δομές Scheduler , Queue , Parrallel_Printing_Array τις οποίες χρησιμοποιούμε την υλοποίηση του Job Scheduler .
Η δομή Scheduler περιέχει την Queue και είναι αναγκαία για την υλοποίηση της.
Η δομή Queue έχει τιε λειτουργείες οιποιουδήποτε Queue, (δηλ, push() , pop(), first())
Αλλά έχει υλοποιηθεί με την μορφή array , για διατήρηση του Locality, και πιο γρήγορης υλοποίησης από λίστα η οποία θα περιείχε πολλά calls για απόκτηση και ελευθέρωσης μνήμης.
Η δομή Parrallel_Printing_Array χρησομοποιήται για την αποθύκευση των αποτελεσμάτων των search και την αποφυγή χρήσης critical section όταν επιστρέφουν τα threads τα αποτελέσματα αυτά.
Υπάρχουν 2 int Job_Id , Print_Id όπου ο πρώτος χρησιμοποιήται για την ανάθεση Job_Id και ο δεύτερος για την χρήση του Parrallel_Printing_Array.

Static

Η main δημιουργεί τα threads τα οποία περιμένουν μέχρι η main, να γεμίσει το Scheduler με όλα τα jobs αυτού του batch. Εφόσον τα φορτώσει όλα , τα threads πέρνουν ένα ένα job (με τη χρήση critical section) και έπειτα εκτελούν τα searches παράλληλα. Όταν τελειώνει το search αποθυκέυουν τα αποτελέσματα στην κατάλληλη θέση στο Parallel_Printing_Array η οποία αναθέτε σε σχέση με το Job_Id του κάθε Query , έτσι ώστε να διατηρείται η σειρά που καλέσθηκαν και για να αποφύγουμε critical section. Με το άδεισμα του Scheduler ειδοποιείται η main ,η οποία περίμενε μέχρις στιγμής , η τυπώνει τα αποτελέσματα. Έπειτα επαναλαμβάνεται η επεξεργασία αυτή εσώτου διαβασθεί όλο το αρχείο.

Dynamic

Η main δημιουργεί τα threads τα οποία περιμένουν μέχρι η main, να γεμίσει το Scheduler με όλα τα jobs αυτού του batch. Κατα την δημιουργία του κάθε job, τσεκάρει αν είναι add ή delete. Εφόσον είναι κάτι από αυτά, τα αποθηκεύει σε ένα βοηθητικού Queue ,έτσι ώστε να τα χωρίσει από τα Queries και την αποφυγή sorting των Jobs. Έπειτα υλοποιεί τα add και delete σειριακά, με την χρήση versioning. Εφόσον τελειώσει , τα threads παίρνουν ένα ένα job (με τη χρήση critical section) και έπειτα εκτελούν τα searches παράλληλα. Όταν τελειώνει το search αποθηκεύουν τα αποτελέσματα στην κατάλληλη θέση στο Parallel_Printing_Array η οποία αναθέτει σε σχέση με το Job_Id του κάθε Query , έτσι ώστε να διατηρείται η σειρά που καλέστηκαν και για να αποφύγουμε critical section. Με το άδειασμα του Scheduler ειδοποιείται η main ,η οποία περίμενε μέχρις στιγμής , η οποία τυπώνει τα αποτελέσματα. Έπειτα επαναλαμβάνεται η επεξεργασία αυτή εσώτου διαβασθεί όλο το αρχείο.

Testing and Results

Computer 1

CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Memory: 8gb

Static

Threads 2

	Small	Medium	Large
Static	6.7s	5m 26s	3m 28s
Dynamic	7.7s	7m 20s	3m 30s

Threads 4

	Small	Medium	Large
Static	5.6s	4m 20s	2m 25s
Dynamic	6.5s	6m 28s	3m 06s

Threads 8

	Small	Medium	Large
Static	5.7s	4m 13s	2m 28s

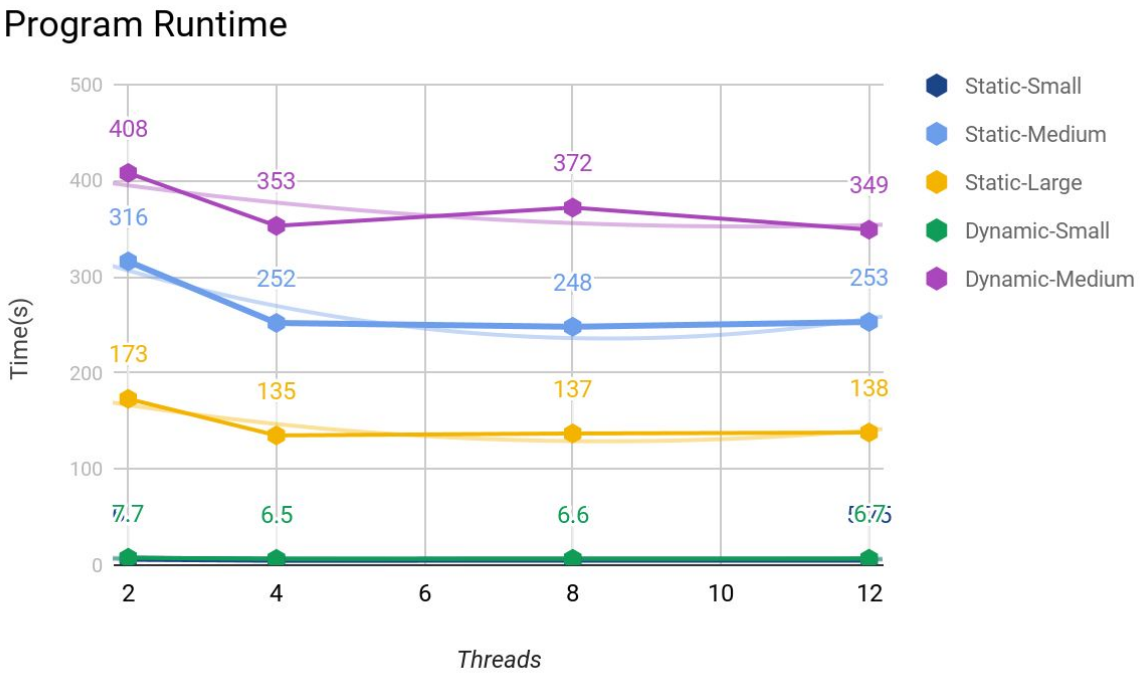
Dynamic	6.6s	6m 20s	2m 60s

Threads 12

	Small	Medium	Large
Static	5.72s	4m 21s	2m 30s
Dynamic	6.6s	6m 21s	2m 33s

Σημείωση : Οι παραπάνω χρόνοι δεν εκτελέστηκαν με compiler optimized programm (-02). Οι χρόνοι βελτιώνονται κατά 15-20 seconds με το παραπάνω flag στο makefile , στο συγκεκριμένο PC.

Charts



Dynamic-Small και Static-Small πέφτουν μαζί και δεν φαίνονται οι διαφορές

Computer 2

CPU: Intel® Core™ i5-2500K CPU @ 3.30GHz × 4
Memory: 8gb

Static

Threads 2

	Small	Medium	Large
Static	2.61s	1m 22s	56.98s
Dynamic	2.99s	1m 36s	1m 06s

Threads 4

	Small	Medium	Large
Static	1.41s	51.48s	37.83s
Dynamic	1.66s	1m 05s	51.03s

Threads 8

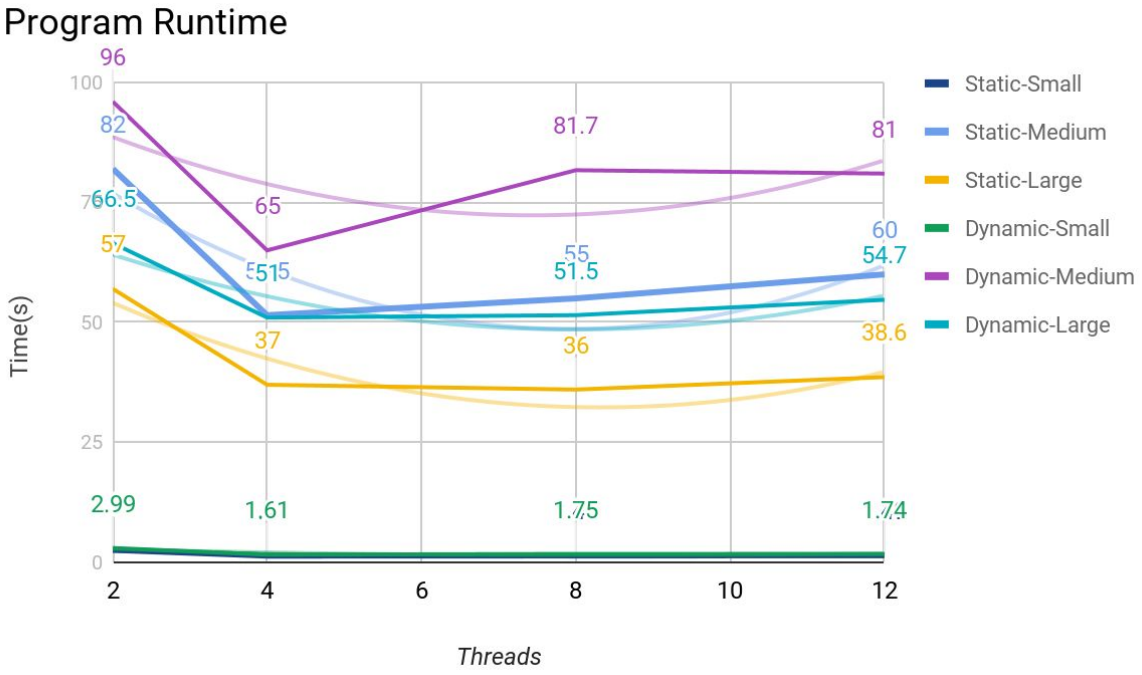
	Small	Medium	Large
Static	1.45s	55.16s	36.44s
Dynamic	1.75s	1m 21s	51.55s

Threads 12

	Small	Medium	Large
Static	1.49s	59.98s	38.66s
Dynamic	1.74s	1m 20s	54.73s

Σημείωση: Οι παραπάνω χρόνοι εκτελέστηκαν με compiler optimization (-O2) και παρατηρήθηκε διαφορά 30 seconds στο συγκεκριμένο PC , από ότι χωρίς το συγκεκριμένο flag.

Charts



Dynamic-Small και Static-Small πέφτουν μαζί και δεν φαίνονται οι διαφορές

Testing Details

Οι χρόνοι μετρήθηκαν με τη χρήση προγράμματος time αλλά και με χρήση του system clock.

Testing Conclusions

-Τα τέστ επιβεβαιώνουν ότι τα dynamic files χρειάζονται παραπάνω χρόνο από τα static, το οποίο είναι προφανές εξαιτίας του επιπλέον φόρτου και του ότι ο επιπλέον φόρτος υλοποιείται γραμμικά και δεν εκμεταλλεύεται τα threads. (Κέρδος Static > Κέρδος Dynamic)

-Επιπλέον τα βλέπουμε ότι υπάρχει πολύ καλό scaling στην παραλληλία και όσο πιο μεγάλος ο φόρτος τόσο πιο μεγάλο το κέρδος. (Κέρδος Small < Κέρδος Large < Κέρδος Medium)

-Όμως αν ο φόρτος επιπλέον είναι κυρίως στο linear κομμάτι, τα αποτελέσματα δεν δείχνουν τόσο μεγάλο κέρδος.(Κέρδος Large < Κέρδος Medium)
Τέλος σε σχέση με την γραμμική υλοποίηση έχουμε πολύ μεγαλύτερο κέρδος την τάξης του 250%.

-Για τον πρώτο υπολογιστή παρατηρούμε all-time low σε χρόνο για το medium dataset τα 8 threads σε static Trie , όπου το πρόγραμμα έτρεξε σε 4m 13s , ενώ all-time high για το medium έχουμε τα 2 threads σε dynamic Trie , όπου παίρνουμε χρόνο 7m 20s.

-Έχουμε τον καλύτερο χρόνο σε large dataset αυτόν από 4 threads σε static Trie 2m 25s , ενώ τον χειρότερο σε large είναι με 3m 30s για 2 threads dynamic. Ωστόσο σημειώνεται ότι πολύ κοντά σε all-time low για το large dataset βρίσκεται και ο χρόνος με 8 threads σε static Trie , 2m 28s.

Για τον δεύτερο υπολογιστή παρατηρούμε all-time low σε χρόνο για το medium dataset τα 4 threads σε static Trie , όπου το πρόγραμμα έτρεξε σε 51s , ενώ all-time high για το medium έχουμε τα 2 threads σε dynamic Trie , όπου παίρνουμε χρόνο 1m 36s.
Επίσης έχουμε τον καλύτερο χρόνο σε large dataset αυτόν από 8 threads σε static Trie 36s , ενώ τον χειρότερο σε large είναι με 1m 06s για 2 threads dynamic. Ωστόσο σημειώνεται ότι πολύ κοντά σε all-time low για το large dataset βρίσκονται και οι χρόνοι για 4 και 12 threads με τιμές 37.83s και 38.66s.

Για το small τα δεδομένα είναι πολύ μικρά για να κάνουμε κάποια σύγκριση.

Από τις παραπάνω παρατηρήσεις βλέπουμε ότι η πιο αργή εκτέλεση είναι αυτή του Dynamic Trie με 2 threads ανεξαρτήτως υπολογιστή , ενώ η πιο γρήγορη αυτή του Static Trie με 4 ή 8 threads (οι χρόνοι είναι πολύ κοντά και συνεπώς αμελητέες οι διαφορές) . Αυτό είναι και η αναμενόμενη συμπεριφορά καθώς τα static Trie είναι πιο συμπυκνωμένα από ότι τα Dynamic και αποτελούν ένα optimization σε σχέση με Dynamic τόσο σε χρόνο όσο και σε χώρο . Επίσης λογικό είναι οι χρόνοι που είναι οι μεγαλύτεροι να παρατηρούνται σε 2 threads καθώς η παραλληλοποίηση του προγράμματος δεν γίνεται σε μεγάλο βαθμό.

Unit Testing

Ταυτόχρονα με την υλοποίηση της κύριας εργασίας φτιάχναμε και ακόμα ένα πρόγραμμα το οποίο βρίσκεται στο φάκελο “UnitTesting” στο παραδοτέο . Μέσα σε αυτό το πρόγραμμα προσθέταμε κάθε φορά και μία περίπτωση που μας δημιουργούσε πρόβλημα στο κυρίως πρόγραμμα . Κάθε φορά που κάναμε μία αλλαγή testarame πρώτα αν το πρόγραμμα μας περνάει όλα τα υπάρχοντα test που υπάρχουν μέσα στο βοηθητικό μας πρόγραμμα .

Βιοβλιογραφία

https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf