

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ



-Πτυχιακή Εργασία-

Δεικτοδότηση κειμένων με δομές δεδομένων και ανεπτυγμένες τεχνικές
(Text indexing with Data structures and advanced techniques)

Αθανάσιος Τσίτσος

Επιβλέπων: Θεόδωρος Ανδρόνικος

2019

Επιβλέπων

Θεόδωρος Ανδρόνικος, *Αναπληρωτής καθηγητής*

Ιόνιο Πανεπιστήμιο

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ



-Πτυχιακή Εργασία-

Δεικτοδότηση κειμένων με δομές δεδομένων και ανεπτυγμένες τεχνικές
(Text indexing with Data structures and advanced techniques)

Αθανάσιος Τσίτσος

Επιβλέπων: Θεόδωρος Ανδρόνικος

2019

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον καθηγητή , κ . Ανδρόνικο Θεόδωρο για την πολύτιμη βοήθεια του απαντώντας έγκαιρα και με σαφήνεια στις ερωτήσεις μου. Επίσης θα ήθελα να ευχαριστήσω και την οικογένεια μου για όλη την υποστήριξη που μου παρείχαν για την ολοκλήρωση αυτής της πτυχιακής εργασίας .

Περιεχόμενα

Ευχαριστίες	4
Περιεχόμενα.....	5
Λίστα εικόνων.....	8
Συντομεύσεις.....	9
Γλωσσάρι Ξενικών Όρων	10
Περίληψη	12
Abstract	13
Εισαγωγή.....	14
Κεφάλαιο 1 ^ο : Παρουσίαση αλγορίθμων	16
1.1: Αλγόριθμος Udayan K. και Anirudh K. [1]	16
1.1.1: Σχετικά με τον αλγόριθμο	16
1.1.2: Κεντρική ιδέα	17
1.1.3: Συμπύεση.....	18
Φάση 1η	18
Φάση 2η	18
Φάση 3η	19
Φάση 4η	19
1.1.4: Αναζήτηση.....	19
Βήμα 1.....	20
Βήμα 2.....	20
Βήμα 3.....	20
1.2: Αλγόριθμος Gawrychowski P. [2]	22
1.2.1: Ορολογία	23
1.2.2: Προϋποθέσεις	23
1.2.3: Σχετικά με τον αλγόριθμο	24
1.2.4: Λήμματα και υπορουτίνες του αλγορίθμου.....	25
Λήμμα 1	25
Λήμμα 2	25
1.3: Αλγόριθμος αναζήτησης σε πολλαπλές αναφορές [3]	26
1.3.1: Εισαγωγικές έννοιες	26
Ορισμός 1.....	27
1.3.2: Συμπύεση ανόμοιων συμβολοσειρών	28

1.3.3: Αναζήτηση.....	29
Στάδιο 1.....	30
Ορισμός 2.....	30
Στάδιο 2.....	30
Ορισμός 3.....	31
1.3.4: Βρίσκοντας όλες τις αντιστοιχίσεις.....	32
1.3.5: MRCSI.....	34
Ορισμός 4.....	34
Κεφάλαιο 2 ^ο : Παρουσίαση δομών δεδομένων	36
2.1: Suffix arrays και suffix trees	36
2.1.1: Suffix trees [4].....	37
2.1.1.1: Δομή ενός suffix tree	38
2.1.1.2: Σύστημα παρενθέσεων σε ένα δέντρο	39
2.1.2: Συμπιεσμένα suffix trees [4]	40
2.1.2.1: Προαπαιτούμενα	40
2.1.2.2: Αναπαράσταση κόμβων.....	40
Ορισμός 5.....	40
Λήμμα 3	41
2.1.2.3: Αναζήτηση στο suffix tree.....	42
2.1.2.4: Υπολογισμός των συνδέσεων επιθεμάτων	43
Λήμμα 4	43
2.1.3: Suffix arrays [5].....	44
2.1.3.1: Αναζήτηση στα suffix arrays	45
2.1.3.2: Προβλήματα χωρητικότητας [6].....	46
2.1.4: Συμπιεσμένα suffix arrays [6]	47
Βήμα 1.....	47
Βήμα 2.....	48
Βήμα 3.....	48
Βήμα 4.....	48
Λήμμα 5	49
Διαδικασία compress	50
Συνάρτηση lookup	50
2.2: Suffix Intervals [7]	50

Λήμμα 6	51
Ορισμός 6.....	51
2.3: Suffix Trays [7]	51
Ορισμός 7.....	52
Λήμμα 7	52
2.3.1: Δομή ενός suffix tray.....	53
Λήμμα 8	54
2.3.2: Αναζήτηση query.....	54
Λήμμα 9	55
2.4: Balanced Indexing Structures και Suffix Trists [7]	55
2.4.1: Εισαγωγή νέων κόμβων στο BIS.....	56
Λήμμα 10	56
2.4.2: Εντοπίζοντας έναν καινούργιο σ-κόμβο.....	57
2.4.3: Ενημερώνοντας τον καινούργιο σ-κόμβο	58
Λήμμα 11	59
2.4.4: Όταν ένας σ-κόμβος γίνεται διακλαδωμένος-σ-κόμβος.....	59
Λήμμα 12	60
Κεφάλαιο 3 ^ο : Σύγκριση δομών δεδομένων.....	61
Κεφάλαιο 4 ^ο : Πρακτική μελέτη των suffix arrays	64
4.1: Κώδικας σε γλώσσα Java	64
4.2: Επεξήγηση του κώδικα.....	67
4.3: Παραδείγματα εφαρμογής του κώδικα.....	69
4.3.1: Παράδειγμα 1	69
4.3.2: Παράδειγμα 2.....	70
4.3.3: Ανάλυση αποτελεσμάτων	71
Συμπεράσματα	72
Βιβλιογραφία	74

Λίστα εικόνων

Εικόνα 1: Βασική ιδέα συμπίεσης με πολλαπλές αναφορές.....	26
Εικόνα 2: Παράδειγμα ευρετηρίων για τις συμπιεσμένες συμβολοσειρές.....	31
Εικόνα 3: Το suffix tree για ένα κείμενο $T = \text{"ababac\$"}'$ μαζί με την αναπαράσταση των παρενθέσεων του.	37
Εικόνα 4: Suffix array κατασκευασμένο από ένα κείμενο T	44
Εικόνα 5: Παράδειγμα εφαρμογής των βημάτων 1-4, για το κείμενο $T = \text{"abbabbabbabbabaaabababbabbabbabba\#"}$	49
Εικόνα 6: Παράδειγμα εφαρμογής του κώδικα για το κείμενο $T = \text{"abbabbabbabbabaaabababbabbabbabba\#"}$	70

Συντομεύσεις

BIS	Balanced Indexing Structure
lcp	Longest Common Prefix
MRC SI	Multi-Reference Compressed Search Index
MRCSD	Multi-Reference Compressed String Database
RMQ	Range Minimum Query
RME	Referential Match Entry

Γλωσσάρι Ξενικών Όρων

Balanced Indexing Structure	Ισορροπημένη Δομή Ευρετηρίου
Bit Vector	Δυαδικό Διάνυσμα
Child	Παιδί
Codeword	Κωδική Λέξη
Compressed Suffix Array	Συμπιεσμένος Πίνακας Επιθεμάτων
Compressed Suffix Tree	Συμπιεσμένο Δέντρο Επιθεμάτων
Indexing	Ευρετηρίαση
Interleaving	Διεμπλεκόμενος
Internal Node	Εσωτερικός Κόμβος
Interval Tree	Δέντρο Διαστήματος
Label	Ετικέτα
Leaf	Φύλλο
Longest Common Prefix	Μακρύτερο Κοινό Πρόθεμα
Multi-Reference Compressed Search Index	Ευρετήριο Συμπιεσμένης Αναζήτησης Πολλαπλών Αναφορών
Multi-Reference Compressed String Database	Συμπιεσμένη Βάση Δεδομένων Συμβολοσειρών Πολλαπλών Αναφορών
Node	Κόμβος
Node-depth	Βάθος Κόμβου
Ordered Tree	Διατεταγμένο Δέντρο
Overlap Map	Χάρτης Επικάλυψης
Path	Διαδρομή
Pattern	Μοτίβο

Prefix	Πρόθεμα
Query	Ερώτημα
Referential Match Entry	Εγγραφή Αναφοράς Αντιστοίχισης
RME Propagation Map	Χάρτης μετάδοσης RME
Root	Ρίζα
Seed-and-extend	Σπορά-και-επέκταση
String-depth	Βάθος Συμβολοσειράς
Suffix Array	Πίνακας Επιθεμάτων
Suffix Interval Node	Κόμβος Επιθέματος Διαστήματος
Suffix Link	Σύνδεση Επιθεμάτων
Suffix Tree	Δέντρο Επιθεμάτων

Περίληψη

Η ευρετηρίαση (indexing) ενός κειμένου αποτελεί έναν από τους κυριότερους παράγοντες για μία αποτελεσματική αναζήτηση. Η ιδέα είναι να προεξεργαστούμε ένα κείμενο και να κατασκευάσουμε έναν μηχανισμό ο οποίος θα μπορεί αργότερα να δίνει απάντηση σε ερωτήματα (queries) της μορφής “εμφανίζεται το μοτίβο (pattern) P στο κείμενο” σε χρόνο ανάλογο με το μέγεθος του μοτίβου αντί του κειμένου.

Στην παρούσα πτυχιακή εργασία θα γίνει μία παρουσίαση μιας σειράς αλγορίθμων καθώς και μερικών δομών δεδομένων που έχουν ως στόχο την αποτελεσματική αποθήκευση κειμένων και την ταχεία αναζήτηση μοτίβων μέσα σε αυτά. Στη συνέχεια, γίνεται μία σύγκριση αυτών των δομών με σκοπό την εύρεση της αποδοτικότερης και ταχύτερης δομής. Επιπλέον, θα γίνει μία πρακτική μελέτη της λειτουργίας και της απόδοσης αυτών των δομών.

Abstract

Indexing a text is one of the key factors for an effective search. The idea is to pre-process a text and construct a mechanism that can later answer queries of the "pattern P appears in the text" format at a time proportional to the size of the pattern instead of the text.

This thesis will present a series of algorithms as well as some data structures that aim to efficiently store texts and quickly search for patterns within them. A comparison is then made of these structures in order to find the most efficient and fastest structure. In addition, a practical study of the operation and performance of these structures will be undertaken.

Εισαγωγή

Στην σημερινή εποχή η τεχνολογία αναπτύσσεται με ιλιγγιώδη ταχύτητα μαζί με την επιστήμη της πληροφορικής και οι υπολογιστές παντός τύπου έχουν ενταχθεί στην ζωή μας σε πολύ μεγάλο βαθμό. Ως αποτέλεσμα αυτού, ο όγκος των ηλεκτρονικών δεδομένων που δημιουργείται καθημερινά ξεπερνάει τον αντίστοιχο από κάθε παλαιότερη περίοδο αφού όλο και περισσότερες πτυχές της ζωής μας καταγράφονται ηλεκτρονικά. Παράλληλα με την αύξηση του ρυθμού παραγωγής δεδομένων ακολουθούν και έρευνες για την δημιουργία πιο μεγάλων χώρων αποθήκευσης, υπολογιστικών συσκευών με βελτιωμένες αποδόσεις και δυνατότητες, όμως καθώς πρόκειται για φυσικά υλικά στα οποία αποθηκεύονται τα δεδομένα έχουν κι αυτά τους περιορισμούς τους. Ως εκ τούτου, η ανεύρεση βέλτιστων μεθόδων αποθήκευσης δεδομένων φαντάζει όχι μόνο αναμενόμενη, αλλά μάλλον αναγκαία. Η αποθήκευση πληροφοριών όμως δεν είναι αυτοσκοπός, θα χρειαστεί να γίνει επεξεργασία όσων αποθηκεύονται. Από τις πρώτες έρευνες για την μείωση του όγκου των δεδομένων που αποθηκεύονται που έφεραν αποτελέσματα επετεύχθη μείωση, με την μορφή συμπίεσης δεδομένων, όμως για την προσπέλαση τους ήταν απαραίτητη η αποσυμπίεση, και αυτό είχε ως συνέπεια την επιβράδυνση της χρήσης των δεδομένων. Ενώ ο διαθέσιμος χώρος για αποθήκευση είναι περιορισμένος το ίδιο ισχύει και για τον χρόνο για την ολοκλήρωση της αναζήτησης, οπότε εμφανίστηκε η ανάγκη για βελτιωμένη αποθήκευση δίχως καθυστέρηση κατά την προσπέλαση. Έτσι δεν άργησε να εμφανιστεί το πρόβλημα που αποκαλείται «Πρόβλημα εύρεσης συμπιεσμένου μοτίβου» (compressed pattern matching problem). Πώς θα μπορούσε, δηλαδή, να γίνει αναζήτηση σε κείμενο που έχει συμπιεστεί με σκοπό βελτιωμένη αποθήκευση, χωρίς όμως την ανάγκη να αποσυμπιεστεί το κείμενο μιας και είναι χρονοβόρα διαδικασία. Έκτοτε, μια ολόκληρη σειρά από μελέτες και έρευνες εμφανίστηκαν με σκοπό να λύσουν αυτό το πρόβλημα ώστε να επιτευχθεί αποτελεσματική συμπίεση και ταυτόχρονα βελτιωμένη ταχύτητα αναζήτησης. Τα αποτελέσματα δείχνουν ότι η δυνατότητα για αναζήτηση ποικίλλει ανάλογα τον τρόπο που έγινε η συμπίεση.

Μια άλλη οδός που ακολουθήθηκε με τελικό σκοπό την βελτίωση αποθήκευσης και ταχύτερη προσπέλαση είναι η δημιουργία ειδικών δομών δεδομένων που κατά μια

έννοια αποτελούν μια συμπιεσμένη μορφή των αρχικών δεδομένων που αναπαριστούν, ενώ ταυτόχρονα παρέχουν διευκολύνσεις για γρήγορη αναζήτηση. Μερικές από τις πλέον γνωστές και ευρέως χρησιμοποιούμενες δομές είναι τα δέντρα επιθεμάτων (suffix trees) και οι πίνακες επιθεμάτων (suffix arrays). Αργότερα, δημιουργήθηκαν βελτιωμένες εκδοχές αυτών των δομών, όπως είναι τα συμπιεσμένα δέντρα επιθεμάτων (compressed suffix trees) και οι συμπιεσμένοι πίνακες επιθεμάτων (compressed suffix arrays), καθώς και δομές οι οποίες συνδυάζουν τα πλεονεκτήματα των suffix trees και των suffix arrays, βελτιώνοντας το χρόνο αναζήτησης.

Κεφάλαιο 1^ο: Παρουσίαση αλγορίθμων

1.1: Αλγόριθμος Udayan K. και Anirudh K. [1]

Ένας αλγόριθμος συμπίεσης που επιτρέπει ταυτόχρονα ταχεία αναζήτηση στη συμπίεσμένη συλλογή εγγράφων παρουσιάζεται από τους Udayan Khurana και Anirudh Koul στο Thapar Institute Of Engineering and Technology στο Punjab της Ινδίας.

1.1.1: Σχετικά με τον αλγόριθμο

Ο αλγόριθμός τους αναπτύχθηκε ειδικά για συλλογές εγγράφων σε φυσική γλώσσα και ως εκ τούτου είναι πολύ πιο αποτελεσματικός από αλγορίθμους γενικού σκοπού. Εκμεταλλεύεται την δομή και την μορφή τέτοιων συλλογών καθώς και στατιστικές έρευνες σχετικά με την συχνότητα εμφάνισης λέξεων και φράσεων. Τα ποσοστά συμπίεσης που επιτυγχάνει είναι 70 με 80 τοις εκατό, ενώ ο χρόνος σειριακής αναζήτησης μειώνεται κατά 80 με 85 τοις εκατό συγκριτικά με μια σειριακή αναζήτηση σε μη συμπίεμένο κείμενο. Οι έρευνες που χρησιμοποιήθηκαν και οι μελέτες που έγιναν για την δημιουργία του αλγορίθμου πραγματοποιήθηκαν στην Αγγλική γλώσσα. Αυτό σημαίνει ότι χρησιμοποιώντας στατιστικές άλλων γλωσσών μπορεί εύκολα να προσαρμοστεί σε κάθε γλώσσα. (Παρακάτω όπου γίνεται λόγος για ανθρώπινη γλώσσα υπονοείται η Αγγλική καθώς σε αυτήν προσαρμόστηκαν οι λεπτομέρειες του αλγόριθμου από τους εφευρέτες του.)

1.1.2: Κεντρική ιδέα

Η κεντρική ιδέα είναι πως συλλογές εγγράφων σε φυσική γλώσσα αποτελούνται από λέξεις χωρισμένες με κενά ή / και σύμβολα μεταξύ τους και ότι σχεδόν όλες αυτές οι λέξεις βρίσκονται καταγεγραμμένες σε ένα τυπικό λεξικό της γλώσσας στην οποία έχει γραφτεί το έγγραφο (π.χ. στο Αγγλικό λεξικό). Κατά συνέπεια τα κενά μπορούν να παραληφθούν και κάθε μοναδική λέξη να καταγραφεί σε μια λίστα και να της ανατεθεί ένας μοναδικός αριθμός, ο οποίος κατά την φάση της συμπίεσης θα αντικαταστήσει κάθε στιγμιότυπο αυτής της λέξης.

Σαφώς για να έχει αποτέλεσμα αυτή η μέθοδος θα πρέπει ο αριθμός να είναι στις πλείστες των περιπτώσεων μικρότερος από την λέξη που αντικαθιστά ως προς τα Bytes που καταλαμβάνει. Οι στατιστικές έρευνες όμως που χρησιμοποιήθηκαν δείχνουν τα παρακάτω. Υπάρχουν περίπου 40 χιλιάδες λέξεις και κατά συνέπεια αρκούν δύο Bytes για την καταχώρηση κάθε λέξης, μιας και δύο Bytes είναι 2^{16} αριθμοί, δηλαδή 65.536. Ο μέσος όρος του μήκους των λέξεων που χρησιμοποιούνται είναι 4,3 χαρακτήρες (Bytes), και κατά συνέπεια 2 Bytes λιγότερα. Σε αυτή την σύγκριση πρέπει επίσης να υπολογιστεί και το κενό μεταξύ των λέξεων που παραλείπεται κατά την συμπίεση θεωρώντας την ύπαρξή του δεδομένη, οπότε το μέσο μήκος των λέξεων που αντικαθίσταται από τον αλγόριθμο είναι 5,3. Έτσι με μια πρώτη εκτίμηση γίνεται συμπίεση $(5,3 - 2)/5,3 = 0,62$ άρα 62%.

Σχετικά με τις λέξεις που δεν ανήκουν στο λεξικό της γλώσσας που χρησιμοποιείται - είτε γιατί είναι ονόματα, έχουν ορθογραφικά λάθη ή όποια άλλη αιτία- χρησιμοποιείται ένας αριθμός, δηλαδή δύο Bytes, για να αναπαραστήσει δύο χαρακτήρες (Bytes) από την νέα λέξη. Κάθε Byte της λέξης μπορεί να έχει μια εκ των τιμών [a- z] U [A-Z] U [0-9] ή την τιμή του 'κανένας χαρακτήρας'. Συνολικά πρόκειται για $2 \times 26 + 10 + 1 = 63$ ενδεχόμενες τιμές ο κάθε χαρακτήρας από τους δύο, άρα $63 \times 63 = 3969$ συνδυασμούς οι δύο χαρακτήρες μαζί. Άρα χρειάζονται τόσοι αριθμοί διαθέσιμοι για να αντιστοιχισθούν με όλους τους πιθανούς συνδυασμούς δύο χαρακτήρων που μπορεί να προκύψουν. Επειδή όμως υπάρχει ανάγκη να είναι διακριτό πότε οι δύο χαρακτήρες που αντικαθίστανται με έναν αριθμό είναι η αρχή μιας λέξης ή όχι, για την περίπτωση δύο λέξεων κατά σειρά που δεν ανήκουν στο λεξικό, χρησιμοποιούνται οι διπλάσιοι συνδυασμοί δηλαδή 7938.

Για διευκρίνιση αναφέρεται ότι 40 χιλιάδες, που είναι οι λέξεις, συν 8 χιλιάδες, που είναι οι δυνατοί συνδυασμοί από δυάδες Byte, λέξεων που δεν ανήκουν στο λεξικό, είναι πάλι λιγότερες από τις 65 χιλιάδες που είναι οι δυνατοί συνδυασμοί των δύο Bytes.

1.1.3: Συμπίεση

Ο αλγόριθμος αποτελείται από τέσσερις διαδοχικές φάσεις που κάθε μια έχει ως είσοδο το αποτέλεσμα της προηγούμενης. Η τρίτη φάση δεν έχει υλοποιηθεί, ενώ η τέταρτη φάση βελτιώνει την συμπίεση αλλά καθυστερεί την αναζήτηση.

Φάση 1η

- Ανάλυση του κειμένου σε λέξεις που τις ακολουθεί (ως διαχωριστής) ένα κενό ή ένα σύμβολο ή συνδυασμός των δύο μαζί.
- Αντικατάσταση κάθε λέξης και διαχωριστή με τον αντίστοιχο μοναδικό αριθμό που τους αντιστοιχεί ή αν δεν είναι στην λίστα με τις γνωστές λέξεις με την αλληλουχία από αριθμούς που αντιστοιχούν στις δυάδες των χαρακτήρων τους.
- Παράληψη των διαχωριστών που αποτελούνται από ένα κενό, και αντικατάσταση αυτών που αποτελούνται από συνδυασμό κενού και συμβόλου με τον μοναδικό αριθμό που τους αντιστοιχεί.

Φάση 2η

- Προσπέλαση του κειμένου με σκοπό να εντοπιστούν συχνοί συνδυασμοί λέξεων όπως είναι καταχωρημένοι στην λίστα συνδυασμών και αντικατάστασή τους με τον μοναδικό αριθμό που τους αντιστοιχεί.

Φάση 3η

- Εύρεση και αντικατάσταση δομών προτάσεων όπως έχουν καταχωρηθεί στη λίστα προτάσεων.
- Καταγραφή της συχνότητας εμφάνισης σε μια ξεχωριστή δομή.

Φάση 4η

- Εύρεση επαναλαμβανόμενων μοτίβων μοναδικών αριθμών.
- Αντιστοίχιση ενός μοναδικού αριθμού για τα μοτίβα αυτά, μέσα στο διάστημα των 65.536 δυνατών συνδυασμών που προκύπτουν από δύο Bytes και αντικατάστασή τους.

Για να επιτευχθεί συμπίεση μεγαλύτερη κατά μέσο όρο από 60% καταστρώθηκαν δύο τεχνικές:

- Παρατήρηση συχνών μοτίβων λέξεων, όπως 'I am', 'we are', 'because of', που μπορούν να αντικατασταθούν από έναν μοναδικό αριθμό αντί για έναν για κάθε λέξη στο μοτίβο.
- Αντιστοίχιση των πιο συχνών λέξεων με μοναδικούς αριθμούς εντός ενός συγκροτήματος από 256 συνδυασμούς στην λίστα με γνωστές λέξεις, λ.χ. τους πρώτους 256, έτσι ώστε όταν θα γίνει η αντικατάστασή τους το πρώτο Byte των μοναδικών αριθμών που θα τοποθετηθούν να είναι το ίδιο. Αυτό επιτρέπει για μεγαλύτερη συμπίεση καθώς θα αυξήσει την επανάληψη στο αποτέλεσμα του αλγορίθμου.

1.1.4: Αναζήτηση

Η αναζήτηση είναι δυνατή δίχως την ανάγκη για αποσυμπίεση της συλλογής εγγράφων αφού κάθε λέξη αντιστοιχεί σε έναν μοναδικό αριθμό που είναι

καταγεγραμμένος σε μια λίστα και επίσης η συμπίεση γίνεται σειριακά. Συνεπώς αρκεί η εύρεση του μοναδικού αριθμού που αντιστοιχεί στην λέξη προς αναζήτηση και έπειτα η εύρεση του αριθμού αυτού στη συμπιεσμένη συλλογή.

Έστω:

- 'Σ' η συμπιεσμένη συλλογή δεύτερης φάσης.
- 'Λ' μια λέξη ή συνδυασμός προς εύρεση.
- 'L' η λίστα αντιστοίχισης μοναδικού αριθμού προς λέξη.
- 'C' η λίστα των συνδυασμών λέξεων.
- 'C₁' το σύνολο των λέξεων του L που συμμετέχουν σε συνδυασμούς στο C.
- 'C_w' το σύνολο των συνδυασμών του C που εμπεριέχουν την λέξη Λ
- 'R' μια ταξινομημένη σειρά από μοναδικούς αριθμούς που περιγράφουν μια λέξη που δεν ανήκει στο L.

Βήμα 1

Αν το Λ ανήκει στο C₁, ανεύρεση όλων των μοναδικών αριθμών για κάθε συνδυασμό του C_w και αναζήτηση για αυτούς στο Σ.

Βήμα 2

Αν το Λ ανήκει στο L, ανεύρεση του μοναδικού αριθμού που αντιστοιχεί στο Λ και αναζήτηση για αυτόν στο Σ.

Βήμα 3

Αν δεν ισχύει η συνθήκη του βήματος 1 ούτε του βήματος 2, αναζήτηση όλων των αριθμών του R στο Σ, μετατροπή στις αντίστοιχες αποσυμπιεσμένες λέξεις και σύγκριση τους με το Λ.

Παρακάτω παρουσιάζονται εκτιμήσεις την επιτάχυνσης.

Ο λόγος για τον οποίο αυξάνεται η επιτάχυνση της αναζήτησης είναι ότι χρειάζεται να γίνουν πολύ λιγότερες συγκρίσεις στην συμπίεσμένη συλλογή σε αντίθεση με την ασυμπίεστη. Συγκεκριμένα, κάθε λέξη της συλλογής που καταλαμβάνει κατά μέσο όρο 5,3 χαρακτήρες (μαζί με το κενό) αντικαθίσταται με ένα μόνο μοναδικό αριθμό. Έτσι θεωρώντας ότι για την ασυμπίεστη συλλογή πρέπει να γίνουν συγκρίσεις όσες και οι χαρακτήρες της συλλογής, έστω 5,3 φορές A' , τότε για την συμπίεσμένη αρκούν A' συγκρίσεις, αφού η συμπίεσμένη συλλογή αποτελείται από A' μοναδικούς αριθμούς. Έτσι μια εκτίμηση της επιτάχυνσης δείχνει $(5,3A' - A') / 5,3A' = 0,81$ άρα 81%.

Ακολουθούν μερικά ενδεικτικά αποτελέσματα που παρέχουν οι εφευρέτες του αλγορίθμου:

Συμπίεση σε E-books

Αρχείο	Αρχικό μέγεθος (KB)	Συμπίεσμένο μέγεθος (B)	Συμπίεση
1	14,8	3846	74,62%
2	1302,12	346189	74,03%
3	573,212	131682	77,60%
4	462,74	121139	74,43%
5	310,79	81557	74,37%

Συμπίεση μικρών αρχείων

Αριθμός Αμερικανικής Πατέντας	Αρχικό μέγεθος (B)	Συμπιεσμένο μέγεθος (B)	Συμπίεση
6.227.334	1246	304	75,60%
6.227.354	1538	346	77,50%
6.227.352	584	137	76,50%
6.237.756	230	55	76,10%
6.227.332	276	64	76,80%

Αναζήτηση

Μέγεθος (Bytes)			Χρόνος αναζήτησης (ms)		
Ασυμπίεστο	Συμπιεσμένο	Συμπίεση %	Ασυμπίεστο	Συμπιεσμένο	Επιτάχυνση %
5014247	1420802	72 %	447	73.4	84%
2507126	6998952	72 %	207.7	37	82%
1253565	330117	74%	105	17.7	83%

1.2: Αλγόριθμος Gawrychowski P. [2]

Ένας άλλος αλγόριθμος για αναζήτηση σε συμπιεσμένο κείμενο, κατά μέθοδο Lempel-Ziv-Welch, αναπτύχθηκε από τον Pawel Gawrychowski στο πανεπιστήμιο του Wroclaw υπό την επίβλεψη του καθηγητή Krzysztof Lorys.

1.2.1: Ορολογία

Χρησιμοποιείται συγκεκριμένη ορολογία από τους δημιουργούς του αλγορίθμου κατά την περιγραφή του, συνεπώς παρατίθεται εδώ για την καλύτερη κατανόησή του.

- Σύνоро – Σύνоро μιας συμβολοσειράς s $[1 - |s|]$, όπου $|s|$ είναι το μήκος της συμβολοσειράς, είναι ένα τμήμα που είναι ταυτόχρονα και πρόθεμα και επίθεμα της συμβολοσειράς. Για παράδειγμα $s[1 - i] = s[(|s| - i + 1) - |s|]$, όπου $i < |s|$.
- Σύνορα(s) – είναι το σύνολο $\{i_1, \dots, i_k\}$ όλων των συνόρων της συμβολοσειράς, όπου κάθε δείκτης αποτελεί και το μήκος του συνόρου.
- Σύνоро μιας λέξης – Το πιο εκτενές σύνορο της λέξης.
- Κωδικές λέξεις (codewords) – Παράγεται κατά την ανάλυση συμβολοσειράς κατά Lempel-Ziv, η συμβολοσειρά χωρίζεται σε μια αλληλουχία από κωδικές λέξεις.
- Απόσπασμα – είναι οποιοδήποτε υποσύνολο μια συμβολοσειράς.

1.2.2: Προϋποθέσεις

Έστω ότι έχουμε ένα ταξινομημένο αλφάβητο Σ . Ας θεωρήσουμε δύο ενδεχόμενα, ότι είτε το Σ σταθερό είναι πεπερασμένο είτε ότι αποτελείται από ακεραίους που μπορούν να ταξινομηθούν σε γραμμικό χρόνο. Ο αλγόριθμος έχει την ίδια πολυπλοκότητα και για τα δύο ενδεχόμενα αλλά χρειάζεται επιπροσθέτως μερικές σημαντικές ιδέες για να πιστοποιηθεί ότι δεν αυξάνει η πολυπλοκότητα όταν το αλφάβητο Σ δεν είναι σταθερό.

Ας θεωρήσουμε ότι οι συμβολοσειρές του Σ έχουν δοθεί σε συμπιεσμένη κατά αλγόριθμο Lempel-Ziv μορφή, και επομένως αποτελούνται από αλληλουχίες από τμημάτων όπου κάθε τμήμα είναι είτε ένας χαρακτήρας είτε οποιοδήποτε τμήμα συμβολοσειράς του ήδη δηλωμένου προθέματος. Η αναπαράσταση μιας συμβολοσειράς $t[1 - N]$ Lempel-Ziv είναι μια σειρά από τριάδων (αρχή, μήκος, επόμενο) για κάθε $i \in [1 - n]$, όπου $[1 - n] \subseteq \mathbb{N}$ και n είναι το μέγεθος της αναπαράστασης, αρχή _{i} και μήκος _{i} μη αρνητικοί αριθμοί και το επόμενο _{i} $\in \Sigma$. Οι τριάδες αναφέρονται σε τμήματα του κειμένου $t[\text{αρχή}_i - (\text{αρχή}_i + \text{μήκος}_i - 1)]$ και

καθορίζουν $t[(1 + \sum_{j < i} \text{μήκος}_j) - (\sum_{j \leq i} \text{μήκος}_j)] = t[\text{αρχή}_i - (\text{αρχή}_i + \text{μήκος}_i - 1)]$ επόμενοι. Χρειάζεται να ισχύει, $\text{αρχή}_i = 0$ αν και μόνο αν $\text{μήκος}_i = 0$, και, $\text{αρχή}_i \leq \sum_{j < i} \text{μήκος}_j$ αν $\text{μήκος}_i > 0$. Η αναπαράσταση δεν είναι αυτοαναφορική αν όλα τα τμήματα στα οποία γίνεται αναφορά έχουν ήδη καθοριστεί. Για παράδειγμα, $\text{αρχή}_i + \text{μήκος}_i - 1 \leq \sum_{j < i} \text{μήκος}_j$, $\forall i$. Στην περίπτωση που είναι αυτοαναφορική επιτρέπονται περιγραφές όπως $[(0,0,a), (1, 1, b)]$ που καθορίζουν $a^{1+1}b$. Σε μη αυτοαναφορική περίπτωση χρειάζεται μια περιγραφή μεγέθους $\Theta(\log l)$. Η αλληλουχία τριάδων συχνά αποκαλείται «Lempel-Ziv ανάλυση κειμένου». Ενώ είναι γνωστό πώς να υπολογιστεί αυτή η ανάλυση σε γραμμικό χρόνο, πρακτικά αυτή η γραμμική πολυπλοκότητα δεν είναι αρκετά ικανοποιητική εξαιτίας των μεγάλων σταθερών που εμπλέκονται. Ως εκ τούτου έχει νόημα να αναζητηθούν απλούστερα μοντέλα συμπίεσης. Το πιο γνωστό από αυτά τα μοντέλα είναι η αναπαράσταση κατά Lempel-Ziv-Welch, όπου χωρίζεται μια συμβολοσειρά σε μια αλληλουχία από κωδικές λέξεις, που αποκαλείται η ανάλυση κατά Lempel-Ziv-Welch, με κάθε κωδική λέξη να αποτελεί είτε έναν χαρακτήρα είτε μια προηγούμενη κωδική λέξη συνενωμένη με έναν χαρακτήρα. Το αποτέλεσμα της συμπίεσης έχει ως αποτέλεσμα μια σχετικά απλή διαδικασία συμπίεσης και αποσυμπίεσης αλλά δυστυχώς απαιτεί να έχει ως έξοδο τουλάχιστον $\Omega(\sqrt{N})$ κωδικές λέξεις. Παρ' αυτά η απλότητα και το ποσοστό συμπίεσης που επιτυγχάνεται σε πραγματικές συνθήκες το κάνουν ένα αποτελεσματικό μοντέλο.

1.2.3: Σχετικά με τον αλγόριθμο

Στόχος του αλγορίθμου είναι ο εντοπισμός μιας λέξης σε ένα δεδομένο κείμενο που έχει συμπίεστεί με τον αλγόριθμο Lempel-Ziv-Welch. Παρόλο που η γενική γραμμή είναι να γίνει αναπαράσταση του αλγορίθμου των Knuth, Morris και Pratt που αναπτύχθηκε για γραμμική αναζήτηση λέξης σε κείμενο, επειδή θα απέφερε καθυστέρηση στην διαδικασία η ανάλυση του κειμένου σε επίπεδο χαρακτήρα, γίνεται η προσπάθεια να προσπεραστούν ολόκληρες σειρές από χαρακτήρες. Φαίνεται όμως ότι δουλεύοντας άμεσα σε επίπεδο αλληλουχίας χαρακτήρων δεν είναι βολικό καθώς συχνά δεν θα έχουν ομοιότητες με το μοτίβο κι έτσι δεν μπορεί να γίνει

ιδιαίτερη προεργασία σε στη συλλογή που θα επέτρεπε την επεξεργασία ολόκληρων τμημάτων σε σταθερό χρόνο. Έτσι αρχικά μειώνεται το πρόβλημα σε μια πιο ομοιογενή παραλλαγή, όπου γίνεται η υπόθεση ότι υπάρχει ένα μοτίβο και μια συλλογή συμβολοσειρών που δημιουργείται ενώνοντας κάποια τμήματα του μοτίβου το ένα μετά το άλλο. Σύμφωνα με τους ερευνητές ένα τέτοιου είδους πρόβλημα μπορεί να λυθεί σε βέλτιστο γραμμικό χρόνο και το αποκαλούν «Ταίριασμα μοτίβων σε αλληλουχία αποσπασμάτων». Αυτό που ανακάλυψαν είναι ότι το ταίριασμα ενός μοτίβου σε συλλογή συμπίεσμένη κατά Lempel-Ziv-Welch μειώνεται σε τέτοιου είδους πρόβλημα σε γραμμικό χρόνο αν το αλφάβητο της συλλογής είναι σταθερό.

1.2.4: Λήμματα και υπορουτίνες του αλγορίθμου

Λήμμα 1

«Αν το d και το d' είναι περίοδοι της λέξης λ , και $d + d' \leq |\lambda| + \gcd(d, d')$, τότε και το $\gcd(d, d')$ είναι περίοδος».

Λήμμα 2

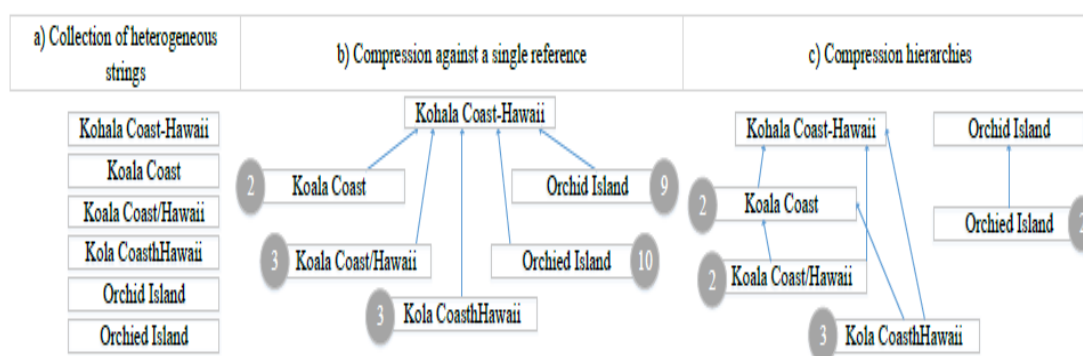
«Αν το σύνορο μιας λέξης λ έχει μήκος $b \geq |\lambda|/2$ τότε όλα τα σύνορα μήκους τουλάχιστον $|\lambda|/2$ δημιουργούν μια αριθμητική πρόοδο. Πιο συγκεκριμένα, η τομή $\text{σύνορα}(\lambda)$ με το σύνολο $\{|\lambda|/2, \dots, |\lambda|\}$ είναι ίσο με το σύνολο $\{|\lambda| - \alpha d : 1 \leq \alpha \leq |\lambda|/2d\}$, όπου $d = |\lambda| - b$ είναι η περίοδος της λέξης. Αυτό το σύνολο αποκαλείται τα μακρά σύνορα της λέξης.»

Απόδειξη:

$d = |\lambda| - b$, είναι η περίοδος του $|\lambda|$ οπότε κάθε αd είναι επίσης μια περίοδος, όσο ισχύει $\alpha \leq |\lambda|/d$. Έτσι όλα τα στοιχεία της αριθμητικής προόδου του λήμματος είναι σύνορα. Πρέπει να δειχθεί ότι δεν υπάρχουν άλλα σύνορα με μήκος τουλάχιστον $|\lambda|/2$. Έστω b' είναι ένα τέτοιο σύνορο. Παρατηρείται ότι $d' = |\lambda| - b'$ είναι μια περίοδος και $d + d' \leq |\lambda|$ οπότε $\gcd(d, d')$ είναι περίοδος επίσης. Αλλά το d είναι η πιο μικρή περίοδος οπότε το d διαιρεί το d' και το b' ανήκει στην πρόοδο.

1.3: Αλγόριθμος αναζήτησης σε πολλαπλές αναφορές [3]

Το ευρετήριο συμπίεσμένης αναζήτησης πολλαπλών αναφορών (Multi-Reference Compressed Search index – MRCSI), το οποίο μπορεί να εργάζεται με πολλαπλές αναφορές για να αυξήσει τα ποσοστά συμπίεσης, επιλέγει αυτόματα αυτές τις αναφορές και παρέχει γρήγορη προσεγγιστική αναζήτηση. Η κεντρική ιδέα παρουσιάζεται στην Εικόνα 1. Το MRCSI δημιουργεί μία ιεραρχία συμπίεσης στην οποία συμπιέζονται συμβολοσειρές σε σχέση με διάφορες άλλες συμβολοσειρές.



Εικόνα 1: Βασική ιδέα συμπίεσης με πολλαπλές αναφορές.

Τα ορθογώνια αναπαριστούν συμβολοσειρές προς συμπίεση. Οι ακμές αναπαριστούν αναφορική συμπίεση. Οι αριθμοί στους κύκλους δείχνουν τον αριθμό των καταχωρίσεων στη συμπίεσμένη αναπαράσταση.

(Πηγή: Wandelt S. and Leser, Ul. (2015), σελ.2).

1.3.1: Εισαγωγικές έννοιες

Μία συμβολοσειρά s είναι μια πεπερασμένη ακολουθία χαρακτήρων ενός αλφαβήτου Σ . Η συνένωση δύο συμβολοσειρών s και t συμβολίζεται με: $s \circ t$. Μία συμβολοσειρά s είναι υποσυμβολοσειρά της συμβολοσειράς t , αν υπάρχουν δύο συμβολοσειρές u και v (πιθανότατα μήκους 0), τέτοιες ώστε $t = u \circ s \circ v$. Το μήκος μίας συμβολοσειράς s συμβολίζεται με $|s|$ και η αρχική θέση i της συμβολοσειράς με μήκος n συμβολίζεται με $s(i, n)$. Το $s(i)$ είναι συντομογραφία του $s(i, 1)$. Όλες οι

θέσεις μίας συμβολοσειράς βασίζονται στο 0. Για παράδειγμα, ο πρώτος χαρακτήρας προσπελαύνεται από το $s(0)$.

Έστω δύο συμβολοσειρές s και t . Τότε, το s είναι κατά k -προσέγγιση παρόμοιο με το t , το οποίο συμβολίζεται ως $s \sim_k t$, αν το s μπορεί να μετατραπεί σε t με το πολύ k πράξεις (αντικατάσταση, διαγραφή ή προσθήκη ενός συμβόλου στο s).

Έστω δύο συμβολοσειρές s και q . Τότε, το σύνολο όλων των αντιστοιχίσεων στο s σε σχέση με το q κατά k -προσέγγιση, συμβολίζεται ως $search(s)_q^k$ και ορίζεται ως το σύνολο $search(s)_q^k = \{(i, s(i, j)) | s(i, j) \sim_k q\}$. Αυτός ο ορισμός επεκτείνεται στην αναζήτηση μιας βάσης δεδομένων S από συμβολοσειρές $\{S_1, \dots, S_n\}$.

Έστω ένα ερώτημα (query) q και μία παράμετρος k . Ορίζουμε το σύνολο όλων των αντιστοιχίσεων κατά k -προσέγγιση για το q στο S ως $DBsearch(S)_q^k = \{(l, search(s_l)_q^k) | s_l \in S\}$. Για παράδειγμα: $Orchid \sim_1 Orchied$, γιατί το σύμβολο 'e' μπορεί να αφαιρεθεί από το 'Orchied' με μία πράξη διαγραφής για να πάρουμε τη συμβολοσειρά 'Orchid'. Αν $S = \{s_1, s_2\}$, με $s_1 = Orchid$ και $s_2 = Orchied$, $DBsearch(S)_{hid}^1 = \{(1, \{(3, hi), (3, hid), (4, id)\}), (2, \{(3, hi), (3, hie), (3, hied)\})\}$.

Ορισμός 1

«Έστω REF ένα σύνολο από συμβολοσειρές αναφοράς και s μία συμβολοσειρά προς συμπίεση. Μία πλειάδα $rme = (refid, αρχή, μήκος, αναντιστοιχία)$ είναι μία εγγραφή αναφοράς αντιστοίχισης (referential match entry – RME) αν το $refid \in REF$ είναι (ένας προσδιορισμός για) μία συμβολοσειρά αναφοράς, η αρχή είναι ένας αριθμός που υποδεικνύει την αρχή μίας αντιστοιχίας μέσα στο $refid$, το μήκος υποδηλώνει το μήκος της αντιστοιχίας και η αναντιστοιχία υποδηλώνει ένα σύμβολο. Μία αναφορική συμπίεση ενός REF, είναι μία λίστα από RME: $rsc = [(refid_1, αρχή_1, μήκος_1, αναντιστοιχία_1), \dots, (refid_n, αρχή_n, μήκος_n, αναντιστοιχία_n)]$ τέτοια ώστε

$(refid_1(αρχή_1, μήκος_1) \text{ ο αναντιστοιχία}_1), \text{ ο } \dots \text{ ο,}$
 $(refid_n(αρχή_n, μήκος_n) \text{ ο αναντιστοιχία}_n) = s.$ »

Για παράδειγμα, έχουμε ένα σετ από συμβολοσειρές $S = \{s_1, \dots, s_6\}$ με $s_1 = \text{Kohala Coast-Hawaii}$, $s_2 = \text{Koala Coast}$, $s_3 = \text{Koala Coast/Hawaii}$, $s_4 = \text{Kola CoastthHawaii}$, $s_5 = \text{Orchid Island}$, $s_6 = \text{Orchid Island}$.

Ένα παράδειγμα αναφορικής συμπίεσης του s_2 με το s_1 είναι $\text{rcs}_2 = [(1,0,2,a)(1,4,7,t)]$, αφού έχουμε $s_2 = s_1(0,2)$ ο a ο $s_1(4,7)$ ο t .

Παρόμοια μπορούμε να συμπίεσουμε αναφορικά το s_5 με το s_1 :

$\text{rcs}_5 = [(1,0,0,O), (1,0,0,r), (1,0,0,c), (1,2,1,i), (1,0,0,d), (1,6,1,I)(1,10,1,l)(1,3,1,n)(1,0,0,d)]$. Όμως, μία τέτοια αναφορική συμπίεση δεν είναι αποδοτική διότι ο αριθμός των RME (9) είναι πολύ κοντά στον αριθμό των συμβόλων (13) που υπάρχουν στη συμβολοσειρά s_5 . Η καλύτερη λύση είναι να συμπεστούν συμβολοσειρές με παρόμοιες αναφορές μόνο, έτσι ώστε να εκμεταλλευτούμε ομοιότητες για συμπίεση. Επιπλέον, η εκμετάλλευση των ομοιοτήτων με πολλαπλές αναφορές συνήθως μειώνει περαιτέρω τον αριθμό των RME.

Στο παρακάτω παράδειγμα, το s_1 και το s_5 είναι ασυμπίεστες αναφορές, το s_2 είναι αναφορικά συμπεσμένο με το $\{s_1\}$, το s_3 και το s_4 είναι συμπεσμένα με το $\{s_1, s_2\}$ και το s_6 είναι συμπεσμένο με το $\{s_5\}$.

$\text{comp}(s_1, \{s_1\}) = [(1,0,18, i)]$
 $\text{comp}(s_2, \{s_1\}) = [(1,0,2,a)(1,4,7, t)]$
 $\text{comp}(s_3, \{s_1, s_2\}) = [(2,0,11,/), (1,13,5, i)]$
 $\text{comp}(s_4, \{s_1, s_2\}) = [(2,0,2,l), (2,4,7,h)(1,13,5,i)]$
 $\text{comp}(s_5, \{s_5\}) = [(5,0,12,d)]$
 $\text{comp}(s_6, \{s_5\}) = [6 : (5,0,5,e), (5,5,7,d)]$.

Μία κύρια αναφορά δηλώνεται ως συμπεσμένη με τον εαυτό της, όπως το $\text{comp}(s_1, \{s_1\})$ στο παραπάνω παράδειγμα.

1.3.2: Συμπίεση ανόμοιων συμβολοσειρών

Μία συμπεσμένη βάση δεδομένων με συμβολοσειρές πολλαπλών αναφορών κωδικοποιεί μόνο κάποιες συμβολοσειρές από μία βάση δεδομένων με συμβολοσειρές σε μία ασυμπίεστη αναπαράσταση. Αυτές οι ασυμπίεστες συμβολοσειρές που ονομάζονται κύριες αναφορές (primary references),

αποθηκεύονται και χρησιμεύουν ως προσχέδια για τις συμπιεσμένες συμβολοσειρές. Αυτές αποθηκεύονται ως αναφορικές συμπίεσεις, χρησιμοποιώντας όλες τις ασυμπίεστες πρωτεύοντες αναφορές και όλες τις άλλες συμπιεσμένες συμβολοσειρές ως πιθανές αναφορές.

Για μία βάση δεδομένων από συμβολοσειρές $S = \{s_1, \dots, s_n\}$, μία συμπιεσμένη βάση δεδομένων συμβολοσειρών πολλαπλών αναφορών (multi-reference compressed string database – MRCSD) αποτελείται από ένα ζεύγος (PREF, COMP), όπου το PREF $\in S$ είναι το σετ από τις κύριες αναφορές και COMP είναι ένα σετ από αναφορικές συμπίεσεις, τέτοιες ώστε $\forall rcs_i \in COMP: decomp(rcs_i) = s_i$.

1.3.3: Αναζήτηση

Οι αλγόριθμοι οι οποίοι χρησιμοποιούν αναφορική συμπίεση με μία αναφορά, συνήθως πραγματοποιούν αναζήτηση κατά k-προσέγγιση σε δύο στάδια. Στο Στάδιο 1, αναζητείται η συμβολοσειρά του query q στην μοναδική αναφορά που υπάρχει, η οποία αποθηκεύεται σε ευρετηριακή μορφή (index), για παράδειγμα χρησιμοποιώντας δέντρα επιθεμάτων (suffix trees, τα οποία θα δούμε αναλυτικά σε επόμενη ενότητα). Όλες οι αντιστοιχήσεις για το q μεταδίδονται σε όλα τα RME που περιέχουν αυτή την αναφορά. Σε κάθε εκτέλεση της αναζήτησης πρέπει να μπορούν να βρίσκονται όλες οι κατά k-προσέγγιση αντιστοιχήσεις. Στο Στάδιο 2, πρέπει να ανιχνευτούν οι αντιστοιχήσεις που εκτείνονται σε περισσότερα του ενός RME. Συστήματα όπως το RCSI τοποθετούν ένα σετ από επιπλέον συμβολοσειρές μέσα σε ευρετήρια. Οι συμβολοσειρές αυτές εκτείνονται σε περισσότερα από (τουλάχιστον) δύο γειτονικά RME, καλύπτοντας έτσι τα τμήματα των συμπιεσμένων συμβολοσειρών που δεν είναι διαδοχικές υποσυμβολοσειρές στην αναφορά. Ο συνδυασμός των αποτελεσμάτων από τα δύο στάδια, παράγει όλες τις αντιστοιχήσεις του q κατά k-προσέγγιση στις συμπιεσμένες συμβολοσειρές.

Εφαρμόζουμε την ίδια τεχνική και στα RMCS τα οποία έχουν πιο περίπλοκες δομές αναφοράς. Πρώτα βρίσκουμε όλες τις αντιστοιχήσεις μέσα στα RME, αναζητώντας ένα ευρετήριο πάνω από συμβολοσειρές στο PREF. Στο Στάδιο 2, εξάγουμε όλες τις υποσυμβολοσειρές από κάθε συμβολοσειρά του COMP που δεν αντιπροσωπεύεται μέσα σε ένα μόνο RME, σε ένα στάδιο προ-επεξεργασίας. Επίσης, πρέπει να

προωθήσουμε όλες τις αντιστοιχίσεις που βρέθηκαν, μέσα από τον γράφο εξάρτησης αναφοράς, ξεκινώντας από όλες τις κύριες αναφορές.

Στάδιο 1

Αποθηκεύουμε όλες τις κύριες αναφορές ως συμπιεσμένα suffix trees. Υπολογίζουμε όλες τις αντιστοιχίσεις του query q , χρησιμοποιώντας το πρότυπο σπορά-και-επέκταση (seed-and-extend). Στη συνέχεια μεταδίδουμε αντιστοιχίσεις από αναφορές στις συμπιεσμένες συμβολοσειρές, χρησιμοποιώντας έναν χάρτη μετάδοσης RME (RME propagation map) ο οποίος παρακολουθεί όλες τις συμπιεσμένες αναφορές.

Ορισμός 2

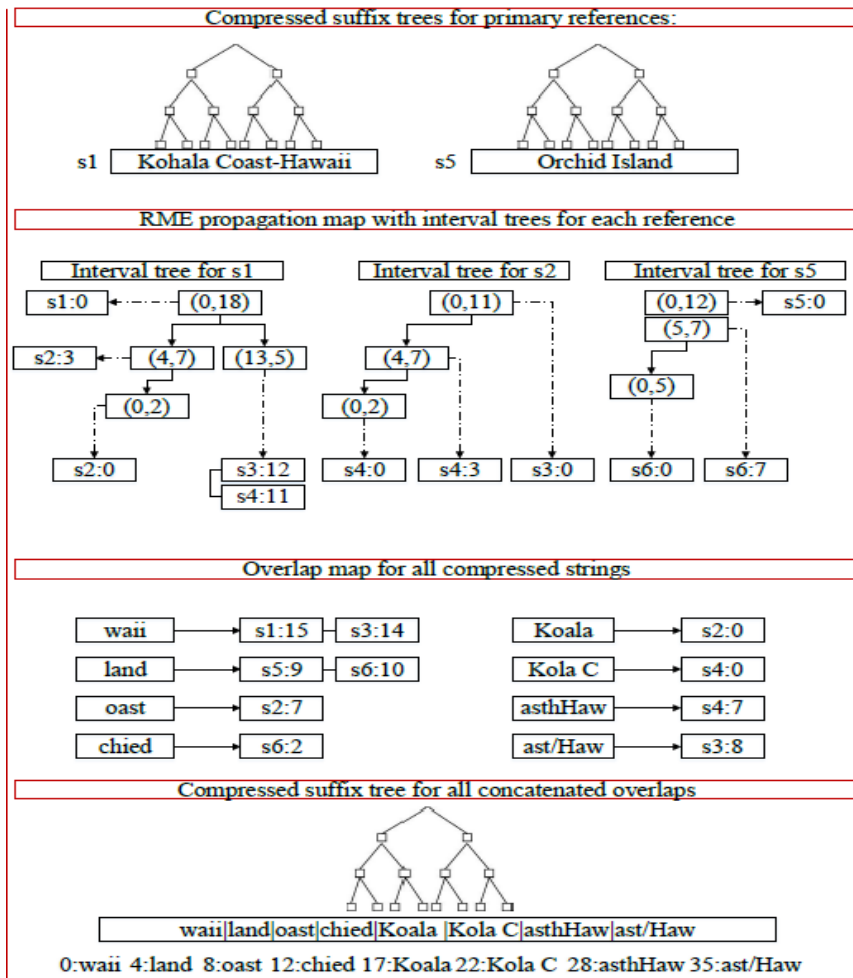
«Έστω μία αναφορική συμπίεση $r_{cs} = [rme_1, \dots, rme_n]$ και οι αναφορές $ref = \{s_1, \dots, s_m\}$. Ένας RME propagation map pm , σχεδιάζει τα id των συμβολοσειρών $\{s_1, \dots, s_m\}$ σε δέντρα διαστήματος (interval trees), έτσι ώστε το διάστημα (αρχή, μήκος) να περιλαμβάνεται στο interval tree $pm(s_i)$, αν και μόνο αν υπάρχει στο r_{cs} ένα RME $rme_j = (s_i, αρχή, μήκος, αναντιστοιχία)$.»

Χρησιμοποιούμε τα RME propagation maps για να μεταδώσουμε αντιστοιχίσεις από τις αναφορικές συμβολοσειρές στις συμπιεσμένες συμβολοσειρές. Έστω ότι κατά k -προσέγγιση εμφανίσεις του q έχουν βρεθεί στην αναφορά s_1 , ξεκινώντας από η θέση $match_s$ με μήκος $match_l$. Με το interval tree του s_1 , βρίσκουμε όλα τα RME των συμπιεσμένων ακολουθιών τα οποία περιέχουν το διάστημα $(match_s, match_l)$. Όλες οι συμπιεσμένες συμβολοσειρές με αυτά τα RME, έχουν τουλάχιστον μία κατά k -προσέγγιση εμφάνιση του q .

Στάδιο 2

Οι αντιστοιχίσεις οι οποίες δεν περιλαμβάνονται μέσα σε ένα RME, πρέπει να επικαλύψουν τουλάχιστον 2 RME. Η θέση και το μήκος αυτών των επικαλύψεων

εξαρτάται από το ακριβές μήκος του query και το όριο σφάλματος k . Έστω $\delta = \max_{ql} + \max_{ed} - 1$, όπου το $\max_{ql}$ είναι το μέγιστο μήκος ενός query και το $\max_{ed}$ είναι το μέγιστο όριο επεξεργασμένης απόστασης (edit-distance). Η τιμή δ περιορίζει την περιοχή γύρω από τους χαρακτήρες αναντιστοιχίας (δ χαρακτήρες στα αριστερά/δεξιά) μέσα στους οποίους μπορεί να εμφανιστεί μια αντιστοίχιση για ένα query. Εξάγουμε όλες αυτές τις επικαλυπτόμενες συμβολοσειρές και καταγράφουμε τις θέσεις τους σε έναν χάρτη επικάλυψης (overlap map).



Εικόνα 2: Παράδειγμα ευρετηρίων για τις συμπιεσμένες συμβολοσειρές

Παράδειγμα ευρετηρίων για τις συμπιεσμένες συμβολοσειρές $s1$ = Kohala Coast-Hawaii, $s2$ = Koala Coast, $s3$ = Koala Coast/Hawaii, $s4$ = Kola CoastHawaii, $s5$ = Orchid Island, $s6$ = Orchied Island.

Πηγή: (Wandelt S. and Leser, Ul. (2015)., σελ. 5).

Ορισμός 3

«Έστω ένα MRCSD (PREF, COMP). Το overlap map του είναι ένας χάρτης από συμβολοσειρές σε ένα σετ από ζεύγη (rcs, pos) και ορίζεται ως $onl(o) = \{(rcs, pos) \mid rcs \in COMP^o \text{ είναι μία συμβολοσειρά επικάλυψης ενός } rme \in rcs \text{ η οποία ξεκινάει από το } pos\}$. Με το overlap map μπορούμε να ανακτήσουμε τις θέσεις των επικαλυπτόμενων συμβολοσειρών μέσα στις συμπίεσμένες συμβολοσειρές.»

Για να βρούμε αντιστοιχήσεις σε επικαλύψεις, εργαζόμαστε ως εξής. Δημιουργούμε ένα ευρετήριο για όλες τις επικαλυπτόμενες συμβολοσειρές. Έστω ότι το overlap map περιέχει τις συμβολοσειρές $o_1, \dots, o_n$ ως κλειδιά. Δημιουργούμε ένα συμπίεσμένο suffix tree πάνω στη συνένωση $o_{total} = o_1 o \dots o o_n$. Το overlap map μαζί με το συμπίεσμένο suffix tree για το o_{total} ονομάζεται ευρετηριασμένο overlap map (indexed overlap map).

1.3.4: Βρίσκοντας όλες τις αντιστοιχήσεις

Στην αρχή εντοπίζονται όλες οι αντιστοιχήσεις που περιέχονται στις κύριες αναφορές με το πρότυπο seed-and-extend, καθώς και τα συμπίεσμένα suffix trees για τις κύριες αναφορές. Στη συνέχεια εντοπίζονται όλες οι επικαλυπτόμενες αναφορές, χρησιμοποιώντας το ίδιο πρότυπο με πριν, καθώς και τα συμπίεσμένα suffix trees για όλες τις συνδεδεμένες επικαλύψεις. Σε αυτό το στάδιο έχει βρεθεί ένα σετ από αρχικές αντιστοιχήσεις σε κάποιες συμβολοσειρές, τις οποίες συμβολίζουμε με M . Οι αντιστοιχήσεις αυτές μεταφέρονται από τις αναφορές στις αναφορικές συμπίεσεις, με το δέντρο διαστήματος (interval tree). Για κάθε αντιστοιχία στο M , προσπελαύνουμε το interval tree της συμβολοσειράς που περιέχει την αντιστοιχία και εντοπίζουμε όλα τα διαστήματα από τις συμπίεσμένες συμβολοσειρές που περιλαμβάνονται σε αυτή την αντιστοιχία. Από αυτά τα διαστήματα αναπαράγουμε τις αρχικές θέσεις εμφάνισης όλων των αντιστοιχήσεων και τις προσθέτουμε στο M . Μετά από μία επανάληψη, παίρνουμε ένα καινούργιο M , το οποίο περιέχει τις αρχικές αντιστοιχήσεις μαζί με τις καινούργιες που προστέθηκαν. Η διαδικασία επαναλαμβάνεται για το καινούργιο M , μέχρις ότου να φτάσει σε ένα σημείο όπου πλέον δεν υπάρχουν άλλες καινούργιες αντιστοιχήσεις.

Για παράδειγμα, έχουμε το σετ από συμβολοσειρές $S = \{s_1, \dots, s_6\}$ με
 $s_1 = \text{Kohala Coast-Hawaii}$, $s_2 = \text{Koala Coast}$, $s_3 = \text{Koala Coast/Hawaii}$,
 $s_4 = \text{Kola CoastHawaii}$, $s_5 = \text{Orchid Island}$, $s_6 = \text{Orchid Island}$.

Στην Εικόνα 2 φαίνονται τα ευρετήρια για τις συμπίεσμένες αυτές συμβολοσειρές. Έστω ότι θέλουμε να κάνουμε αναζήτηση συμβολοσειρών για το query “oad” με $k=1$. Αρχικά βρίσκουμε τις 1-κατά προσέγγιση αντιστοιχίσεις του “oad” στα συμπίεσμένα suffix trees για το s_1 και s_5 . Το query “oad” χωρίζεται στους σπόρους (seeds) “oa” και “d”. Οι θέσεις των σπόρων ανακτώνται και επεκτείνονται για πλήρεις 1-κατά προσέγγιση αντιστοιχίσεις. Αυτό αποφέρει τις 1-κατά προσέγγιση αντιστοιχίσεις (8,oa) και (8,oas) στο s_1 και καμία αντιστοίχιση στο s_5 . Το αποτέλεσμα για το M είναι: $\{(s_1, \{(8,oa), (8,oas)\})\}$. Στη συνέχεια βρίσκουμε τις 1-κατά προσέγγιση αντιστοιχίσεις του “oad” στο συμπίεσμένο suffix tree των επικαλύψεων. Το query “oad” χωρίζεται και πάλι στους σπόρους “oa” και “d”. Οι θέσεις των σπόρων ανακτώνται και επεκτείνονται για πλήρεις 1-κατά προσέγγιση αντιστοιχίσεις. Αυτό αποφέρει τις ακόλουθες 1-κατά προσέγγιση αντιστοιχίσεις:

1. (0,oa) και (0,oad) στο “oast”.
 Επειδή το “oast” εμπεριέχεται στο s_2 στη θέση 7, προσθέτουμε τις αντιστοιχίσεις (7,oa) στο s_2 και (7,oad) στο s_2 , στο M.
2. (1,oa) και (1,oal) στο “Koala”.
 Επειδή το “Koala” εμπεριέχεται στο s_2 στη θέση 0, προσθέτουμε τις αντιστοιχίσεις (1,oa) στο s_2 και (1,oal) στο s_2 , στο M.

Έτσι έχουμε $M = \{(s_1, \{(8,oa), (8,oas)\}), (s_2, \{(1,oa), (1,oal)\})\}$ και θέτουμε $M^* = M$. Τα διαστήματα (8,2) και (8,3) αναζητούνται στο interval tree για το s_1 : Και τα δύο καλύπτονται με το (0,18) από το $s_1:0$ και (4,7) από το $s_2:3$. Έχουμε ήδη τις αντιστοιχίσεις στο s_1 και έτσι προσθέτουμε μόνο το $\{(s_3, \{(1,oa), (1,oal)\})\}$ στο M^* . Η πρώτη επανάληψη έχει τώρα ολοκληρωθεί και θέτουμε $M = M^*$. Πρέπει μόνο να ελέγξουμε για καινούργια $\{(s_2, \{(7,oa), (7,oas)\})\}$ και $\{(s_3, \{(1,oa), (1,oal)\})\}$. Επειδή δεν έχουμε RME propagation map για το s_3 , οι αντιστοιχίσεις για το s_3 δεν μπορούν να μεταδοθούν περαιτέρω. Τα διαστήματα (7,oa) και (7,oas) αναζητούνται στο interval tree για το s_2 . Έτσι, έχουμε τα διαστήματα κάλυψης (0,11) και (4,7), τα οποία οδηγούν στις ακόλουθες νέες αντιστοιχίσεις για το M^* : $\{(s_3, \{(7,oa), (7,oas)\})\}$ και $\{(s_4, \{(6,oa), (6,oas)\})\}$. Η δεύτερη επανάληψη έχει τώρα ολοκληρωθεί και θέτουμε $M=M^*$. Η διαδικασία τερματίζεται εδώ, επειδή δεν έχουμε RME propagation maps

ούτε για το s_3 , ούτε για το s_4 . Το τελικό αποτέλεσμα είναι:
 $M = \{(s_1, \{(8, oa), (8, oas)\}), (s_2, \{(1, oa), (1, oal), (7, oa), (7, oas)\}), (s_3, \{(1, oa), (1, oal), (7, oa), (7, oas)\}), (s_4, \{(6, oa), (6, oas)\})\}$.

1.3.5: MRCSI

Το σύνολο των ευρετηρίων των κύριων αναφορών μαζί με το ευρετήριο του MRE propagation map και το ευρετήριο του overlap map, δίνουν ένα ευρετήριο συμπίεσμνης αναζήτησης πολλαπλών αναφορών (MRCSI). Η απαιτούμενη χωρητικότητα μιας βασικής δομής MRCSI καθορίζεται από τον αριθμό των συμβολοσειρών, το μέγιστο μήκος των συμβολοσειρών και τον βαθμό της ομοιότητάς τους.

Ορισμός 4

«Έστω μία βάση δεδομένων από συμβολοσειρές $S = \{s_1, \dots, s_n\}$ και μία συμπίεσμένη βάση δεδομένων από συμβολοσειρές πολλαπλών αναφορών $mrcsd$ για το S . Ένα ευρετήριο συμπίεσμνης αναζήτησης πολλαπλών αναφορών (MRCSI) είναι ένα σύνολο από εγγραφές της μορφής $mrcsi = (IPREF, rmemap, onlmap)$ όπου το IPREF είναι μία συλλογή από συμπίεσμένα suffix trees για τις κύριες αναφορές στο $mrcsd$ και τα $rmemap$ και $onlmap$ είναι ευρετήρια του MRE propagation map και του overlap map, αντίστοιχα, για το $mrcsd$.»

Μία δομή MRCSI περιέχει όλα τα απαραίτητα στοιχεία σε συλλογές από συμβολοσειρές, για μία αναζήτηση κατά k -προσέγγιση. Οι κατά k -προσέγγιση αντιστοιχήσεις σε μία συμπίεσμένη συμβολοσειρά r cs πρέπει είτε να καλύπτονται πλήρως από ένα RME, είτε να επικαλύπτουν τουλάχιστον δύο RME. Στην πρώτη περίπτωση, αν η αντιστοιχία βρίσκεται μέσα σε ένα RME, τότε αυτό πρέπει να συμβεί σε μία από τις αναφορές του r cs. Αν η αναφορά είναι κύρια, τότε η αντιστοιχία βρίσκεται μέσω των συμπίεσμένων suffix trees στο IPREF και προωθείται στο r cs με το RME propagation map. Στη δεύτερη περίπτωση, το ευρετήριο του overlap map περιέχει όλες τις επικαλύψεις που έχουν μήκος μέχρι δ σε

δύο ή περισσότερα RME. Αφού μία αντιστοίχιση κατά k -προσέγγιση δεν είναι ποτέ μεγαλύτερη από δ , η αντιστοίχιση βρίσκεται μέσα στο overlap map και μεταδίδεται στη θέση της συμπιεσμένης ακολουθίας, χρησιμοποιώντας τις αντισταθμισμένες πληροφορίες που βρίσκονται μέσα στο interval tree.

Κεφάλαιο 2^ο: Παρουσίαση δομών δεδομένων

2.1: Suffix arrays και suffix trees

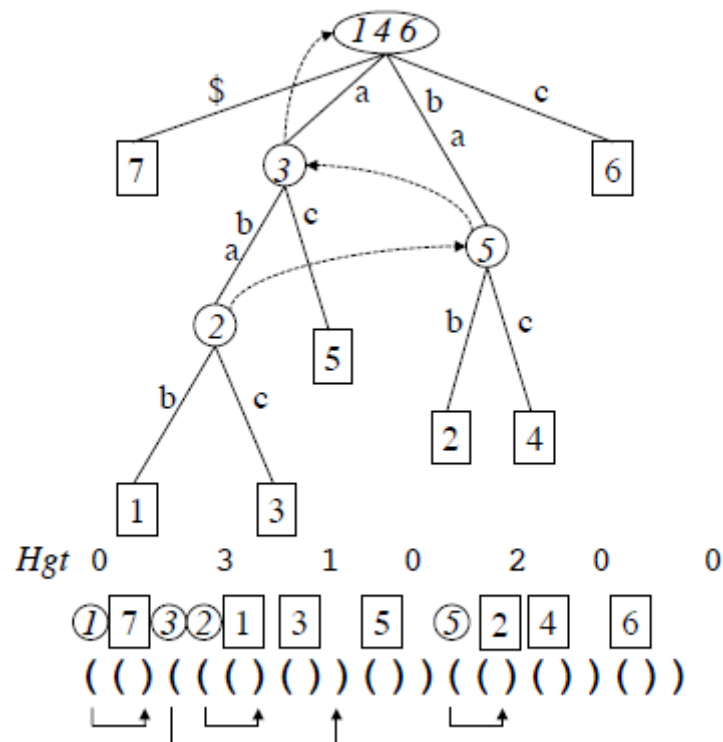
Είναι βασικές δομές δεδομένων των οποίων η σπουδαιότητά γίνεται ολοένα και μεγαλύτερη, λόγω της αυξημένης εφαρμογής τους. Εκτός από την αναζήτηση συμβολοσειρών, χρησιμοποιούνται επίσης και στη μοριακή βιολογία, τη συμπίεση δεδομένων, την εξόρυξη δεδομένων και την ανάκτηση κειμένου.

Έστω $T[1..n] = T[1]T[2] \dots T[n]$ ένα κείμενο μήκους n από ένα αλφάβητο Σ με $|\Sigma| \leq n$. Υποθέτουμε ότι $T[n] = \#$ ή $T[n] = \$$ είναι ένα μοναδικό σύμβολο τερματισμού που αλφαβητικά προηγείται από όλα τα άλλα σύμβολα. Το i -οστό επίθεμα (suffix) του T ορίζεται ως $T[i..n] = T[i]T[i + 1] \dots T[n]$ και εκφράζεται με το T_i . Μία υποσυμβολοσειρά $T[1..i]$ ονομάζεται πρόθεμα (prefix) του T .

Αποθηκεύουμε τους δείκτες i των επιθεμάτων T_i με αυτή τη σειρά σε έναν πίνακα ακέραιων $SA[1..n]$. Αυτός ο πίνακας ονομάζεται πίνακας επιθεμάτων (suffix array).

Τα suffix trees και τα suffix arrays οργανώνουν τα επιθέματα έτσι ώστε να υποστηρίξουν την αποτελεσματικότερη αναζήτηση των προθεμάτων τους.

2.1.1: Suffix trees [4]



Εικόνα 3: Το suffix tree για ένα κείμενο $T = \text{"ababac\$"}$ μαζί με την αναπαράσταση των παρενθέσεών του.

Με τη διακεκομμένη γραμμή αναπαριστώνται τα suffix links.

(Πηγή: Grossi, R. & Vitter, J.Sc. (2005), σελ. 3).

Ένα suffix tree ενός κειμένου $T[1..n]$ είναι ένα συμπιεσμένο δέντρο, κατασκευασμένο πάνω σε όλα τα επιθέματα του T . Έχει n φύλλα (leaves), καθένα από τα οποία αντιστοιχεί σε ένα επίθεμα του T . Κάθε ακμή (edge) του δέντρου προσδιορίζεται από μία ετικέτα (label) που ονομάζεται edge-label. Η συνένωση των ετικετών σε μία διαδρομή (path) από την ρίζα (root) σε έναν κόμβο (node) ονομάζεται path-label. Το path-label για κάθε φύλλο, συμπίπτει με ένα επίθεμα. Για κάθε εσωτερικό κόμβο (internal node) τα παιδιά του (children) με την αλφαβητική σειρά των πρώτων χαρακτήρων των edge-labels. Τα φύλλα ταξινομούνται με τη λεξικογραφική σειρά των επιθεμάτων. Το βάθος συμβολοσειράς (string-depth) ενός κόμβου είναι το μήκος της συμβολοσειράς που αντιπροσωπεύεται από τον κόμβο. Το βάθος κόμβου (node-depth) ενός κόμβου είναι ο αριθμός των κόμβων στη διαδρομή από τη ρίζα προς τον

κόμβο, συμπεριλαμβανομένου του κόμβου αλλά όχι και της ρίζας. Συνδέσεις επιθεμάτων (suffix links) καθορίζονται για όλους τους εσωτερικούς κόμβους του suffix tree, εκτός από τη ρίζα του. Οι συνδέσεις επιθεμάτων είναι απαραίτητες για τη δημιουργία ενός suffix tree και χρησιμεύουν στην αποτελεσματικότερη επίλυση των προβλημάτων. Στην Εικόνα 3, φαίνεται το suffix tree του κειμένου $T = \text{"ababac\$"}$. Τα τετράγωνα συμβολίζουν τα φύλλα και ο αριθμός που βρίσκεται σε κάθε τετράγωνο αναπαριστά τα στοιχεία του suffix array. Οι κύκλοι συμβολίζουν τους εσωτερικούς κόμβους και οι αριθμοί αναπαριστούν τις inorder τάξεις τους, οι οποίες αναλύονται σε επόμενη υποενότητα.

2.1.1.1: Δομή ενός suffix tree

Ένα suffix tree αποτελείται από πέντε μέρη: κείμενο, τοπολογία δέντρου, node-depths, edge-depths και συνδέσεις επιθεμάτων. Το κείμενο είναι μία συμβολοσειρά T , η οποία κωδικοποιείται σε $n \log_2 |\Sigma|$ bits. Η τοπολογία δέντρου αναπαριστά σχέσεις κόμβων γονέα-παιδιού και αποτελείται από $O(n)$ δείκτες οι οποίοι καταλαμβάνουν $O(n \log_2 n)$ bits. Τα node-depths αποθηκεύουν το βάθος των συμβολοσειρών (string-depth) για κάθε εσωτερικό κόμβο σε $n \log_2 n$ bits. Ένα edge-label ανάμεσα σε εσωτερικούς κόμβους, παριστάνεται από έναν δείκτη στο κείμενο σε $n \log_2 n$ bits. Ένα edge-label ανάμεσα σε έναν εσωτερικό κόμβο και ένα φύλλο παριστάνεται από το suffix array του T , το οποίο καταλαμβάνει $n \log_2 n$ bits. Οι συνδέσεις επιθεμάτων είναι δείκτες ανάμεσα σε εσωτερικούς κόμβους και παριστάνονται σε $n \log_2 n$ bits.

Έστω ότι v και w είναι δύο κόμβοι ενός suffix tree και c ένας χαρακτήρας ενός αλφαβήτου Σ . Ένα suffix tree ενός κειμένου, υποστηρίζει τις ακόλουθες λειτουργίες:

1. `root()`: επιστρέφει τον κόμβο της ρίζας.
2. `isleaf(v)`: επιστρέφει «Ναι» αν ο κόμβος είναι φύλλο ή «Όχι» αν δεν είναι.
3. `edge(v, d)`: επιστρέφει τον d -οστό χαρακτήρα του edge-label μίας ακμής που δείχνει στο v .
4. `depth(v)`: επιστρέφει το string-depth του κόμβου v .
5. `child(v, c)`: επιστρέφει τον κόμβο w που είναι παιδί και το edge(v, w) ξεκινάει με τον χαρακτήρα c , ή επιστρέφει 0 αν ο χαρακτήρας δεν υπάρχει.
6. `sibling(v)`: επιστρέφει τον επόμενο αδερφό του κόμβου v .

7. $\text{parent}(v)$: επιστρέφει τον γονεϊκό κόμβο του v .
8. $\text{lca}(v, w)$: επιστρέφει τον μικρότερο κοινό πρόγονο μεταξύ των κόμβων v και w .
9. $\text{sl}(v)$: επιστρέφει τον κόμβο που υποδεικνύεται από τη σύνδεση επιθέματος από το v .

2.1.1.2: Σύστημα παρενθέσεων σε ένα δέντρο

Ένα διατεταγμένο δέντρο (ordered tree) m κόμβων μπορεί να κωδικοποιηθεί σε m εμφωλευμένες (nested) ανοιχτές και κλειστές παρενθέσεις ως εξής. Κατά τη διάρκεια ενός preorder traversal του δέντρου, γράφουμε μία ανοιχτή παρένθεση όταν ένας κόμβος έχει προσπελαστεί, έπειτα διασχίζουμε όλα τα υποδέντρα του κόμβου και γράφουμε μία κλειστή παρένθεση. Ένα παράδειγμα φαίνεται στην Εικόνα 3. Κάθε κόμβος σε ένα δέντρο αναπαριστάται από ένα ζευγάρι από μία ανοιχτή και μία κλειστή παρένθεση $(\dots)$. Επειδή μπορούμε να υπολογίσουμε τη θέση της κλειστής παρένθεσης από αυτή της ανοιχτής παρένθεσης σε σταθερό χρόνο, αναπαριστούμε έναν κόμβο από τη θέση της ανοιχτής παρένθεσης στην ακολουθία.

Κάποιες από τις λειτουργίες πλοήγησης σε ένα δέντρο, είναι οι εξής: rank_p , select_p , findclose και enclose . Η συνάρτηση $\text{rank}_p(i)$ επιστρέφει τον αριθμό των εμφανίσεων του μοτίβου p μέχρι τη θέση i . $\text{select}_p(i)$ επιστρέφει τη θέση της i -οστής εμφάνισης του μοτίβου p . Η συνάρτηση $\text{findclose}(i)$ επιστρέφει τη θέση της κλειστής παρένθεσης που ταιριάζει με την ανοιχτή παρένθεση στη θέση i στην ακολουθία των παρενθέσεων. Η συνάρτηση enclose επιστρέφει τον γονεϊκό κόμβο ενός κόμβου.

Μπορούμε να εκτελέσουμε διάφορες λειτουργίες σε ένα δέντρο, χρησιμοποιώντας τις παραπάνω λειτουργίες. Ανάμεσα σε αυτές, οι παρακάτω είναι σημαντικές για τα συμπιεσμένα suffix trees:

1. $\text{leftfrank}(v)$: επιστρέφει τον αριθμό των φύλλων προς τα αριστερά του κόμβου v σε preorder.
2. $\text{leftmost}(v)$: επιστρέφει το αριστερότερο φύλλο ενός υποδέντρου που έχει ως ρίζα τον κόμβο v .
3. $\text{rightmost}(v)$: επιστρέφει το δεξιότερο φύλλο ενός υποδέντρου που έχει ως ρίζα τον κόμβο v .

2.1.2: Συμπίεσμένα suffix trees [4]

2.1.2.1: Προαπαιτούμενα

Οι δημιουργοί των συμπίεσμένων suffix trees παραθέτουν κάποιες τροποποιήσεις που έχουν κάνει στη λειτουργία των συμπίεσμένων suffix trees, τις οποίες χρησιμοποιούν στη δική τους δομή δεδομένων. Από αυτές, αυτή που θα εξετάσουμε σε επόμενη ενότητα αυτού του υποκεφαλαίου, είναι η ακόλουθη συνάρτηση.

$$\Psi[i] = \begin{cases} i' & \text{τέτοιο ώστε } SA[i'] = SA[i] + 1 \text{ (αν } SA[i] < n) \\ 0 & \text{(αν } SA[i] = n) \end{cases}$$

2.1.2.2: Αναπαράσταση κόμβων

Όπως αναφέραμε στην προηγούμενη υποενότητα, ένας κόμβος μπορεί να αναπαρασταθεί από ένα ζεύγος παρενθέσεων ‘(. . .)’ σε μια εμφωλευμένη ακολουθία παρενθέσεων. Συνεπώς, αναπαριστούμε έναν κόμβο $\bar{v}$ ενός suffix tree με έναν ακέραιο $v \in [1, 2n]$ που είναι η θέση της ανοιχτής παρένθεσης που αντιπροσωπεύει τον κόμβο στην ακολουθία παρενθέσεων P . Για την εκτέλεση κάποιων λειτουργιών στο suffix tree, χρειάζεται να αναπαρασταθεί ο κόμβος από το preorder και το inorder του.

Η αλληλουχία παρενθέσεων P παράγεται κατά τη διάρκεια ενός preorder traversal του δέντρου. Συνεπώς το preorder i ενός κόμβου $\bar{v}$ και η θέση του v μπορούν να μετατραπούν εύκολα από το ένας στο άλλο σε σταθερό χρόνο:

$$i = \text{rank}_P(v)$$

$$v = \text{select}_P(i)$$

Ορισμός 5

«Η inorder τάξη ενός εσωτερικού κόμβου v ορίζεται ως ο αριθμός των εσωτερικών κόμβων που έχουν προσπελαστεί, συμπεριλαμβανομένου και του v , όταν αυτό έχει

προσπελαστεί από ένα παιδί του και ένα άλλο παιδί του είναι το επόμενο που θα προσπελαστεί.»

Μόνο οι εσωτερικοί κόμβοι έχουν inorder τάξεις και ένας εσωτερικός κόμβος μπορεί να έχει δύο ή περισσότερες τάξεις αν έχει περισσότερα από δύο παιδιά. Ένα τέτοιο παράδειγμα φαίνεται στην Εικόνα 3. Κάθε κύκλος αναπαριστά την inorder τάξη ενός εσωτερικού κόμβου. Η ρίζα έχει τρεις inorder τάξεις 1, 4 και 6.

Το ακόλουθο λήμμα παρουσιάζει έναν αλγόριθμο υπολογισμού του inorder ενός εσωτερικού κόμβου.

Λήμμα 3

«Έστω $\bar{v}$ ένας εσωτερικός κόμβος και i το inorder του κόμβου. Τότε το v και το i μπορούν να μετατραπούν από το ένα στο άλλο σε σταθερό χρόνο με τους ακόλουθους τύπους:»

$$i = \text{rank}_0(\text{findclose}(v + 1))$$

$$v = \text{parent}(\text{select}_\ell(i) + 1)$$

Απόδειξη:

Με δεδομένο τον εσωτερικό κόμβο $\bar{v}$, το $v+1$ αναπαριστά τη θέση μιας ανοιχτής παρένθεσης που αντιπροσωπεύει το αριστερότερο παιδί w του v . Έτσι, η θέση $u = \text{findclose}(v + 1)$ είναι η αντίστοιχη κλειστή παρένθεση και το υποδέντρο που έχει ως ρίζα του το w , βρίσκεται ανάμεσα στο v και στο u στην ακολουθία των παρενθέσεων. Επειδή κάθε εσωτερικός κόμβος έχει το πολύ δύο παιδιά, μία inorder τάξη του v ορίζεται από τον αριθμό των φύλλων που έχουν μικρότερες preorder τάξεις από το v και υπολογίζεται από το $\text{rank}_0(u)$. Το inorder i ενός κόμβου v είναι ένας αριθμός που δείχνει πόσες φορές ανεβαίνουμε μία ακμή και πηγαίνουμε αμέσως κάτω, σε μία άλλη ακμή, κατά τη διάρκεια του preorder traversal από τη ρίζα στο v . Αυτή η κίνηση αναπαριστάται με ℓ στην ακολουθία παρενθέσεων. Γι' αυτό, πρέπει πρώτα να υπολογίσουμε τη θέση $x = \text{select}_\ell(P, i)$. Τότε, η ανοιχτή παρένθεση στη θέση $x + 1$ αναπαριστά ένα παιδί του v . Επειδή θέλουμε να μάθουμε ποιες παρενθέσεις αντιστοιχούν στο v , υπολογίζουμε το $\text{parent}(P, x+1)$, το οποίο επιστρέφει τη θέση της ανοιχτής παρένθεσης του v .

Η αναπαράσταση των παρενθέσεων για το παράδειγμα του κειμένου $T = \text{“ababac$”}$ βρίσκεται στο κάτω μέρος της Εικόνας 3. Ένας εσωτερικός κόμβος αναπαριστάται από μία ανοιχτή παρένθεση ακολουθούμενη από μία ακόμα ανοιχτή παρένθεση και είναι διατεταγμένος με βάση την preorder τάξη του. Η inorder τάξη του είναι ο αριθμός των $()$ μέχρι τη θέση που υποδεικνύεται με το βέλος από την ανοιχτή παρένθεση.

2.1.2.3: Αναζήτηση στο suffix tree

Εύρεση της ρίζας: Η ρίζα μπορεί να βρεθεί εύκολα γιατί το preorder της είναι 1 και αναπαριστάται από την πρώτη ανοιχτή παρένθεση στο P . Έτσι, έχουμε $\text{root}() = 1$.

Εύρεση ενός κόμβου παιδιού και ενός κόμβου αδερφού: Η εύρεση ενός αδερφού ενός κόμβου είναι απαραίτητη για την διάσχιση όλων των κόμβων. Για την αποτελεσματικότητα των query, για κάθε κόμβο, αποθηκεύουμε δείκτες προς το πρώτο του παιδί και τον επόμενο αδερφό, το οποίο μπορεί να αναπαρασταθεί από την ακολουθία των παρενθέσεων. Για να βρούμε τον επόμενο αδερφό, υπολογίζουμε το $\text{sibling}(w) = \text{findclose}(P, w) + 1$.

Η εύρεση ενός παιδιού είναι απαραίτητη για τα pattern queries. Από την κατασκευή της ακολουθίας των παρενθέσεων, το πρώτο παιδί του κόμβου v υπολογίζεται από το $w = v + 1$. Στη συνέχεια, υπολογίζουμε επαναληπτικά τον επόμενο αδερφό.

Εύρεση του γονέα: Για να βρούμε τον γονεϊκό κόμβο $\bar{w}$ ενός κόμβου $\bar{v}$, αρκεί να υπολογίσουμε το $w = \text{enclose}(v)$.

Υπολογισμός του lca

Σε ένα συμπίεμένο suffix tree, ο μικρότερος κοινός πρόγονος (lowest common ancestor) ανάμεσα σε δύο κόμβους, υπολογίζεται ως εξής:

Έστω v και w οι θέσεις οι οποίες αναπαριστούν τους κόμβους $\bar{v}$ και $\bar{w}$ σε μία ακολουθία παρενθέσεων. Μπορούμε εύκολα να ελέγξουμε αν το $\bar{v}$ είναι πρόγονος του $\bar{w}$ και αντίστροφα. Έστω ότι $v < w$. Τότε το $\bar{v}$ είναι πρόγονος του $\bar{w}$, αν και μόνο αν $\text{findclose}(v) > \text{findclose}(w)$. Τότε $\text{lca}(v, w) = v$.

Έστω ότι το v δεν είναι πρόγονος του w και το αντίστροφο. Τότε η θέση u της ανοιχτής παρένθεσης που αναπαριστά τον κόμβο $\text{lca}(v, w)$, υπολογίζεται από τον τύπο

$$u = \text{parent}(\text{RMQ}_{P'}(v, w) + 1)$$

όπου το P' είναι μία ακολουθία από ακέραιους, τέτοιους ώστε $P'[i] = \text{rank}_l(i) - \text{rank}_r(i)$. Το RMQ (range minimum query) είναι ένας αλγόριθμος που δημιουργήθηκε από τους Bender and Farach-Colton και επιστρέφει τον δείκτη m του ελάχιστου στοιχείου στο $P'[i..j]$. Έτσι, η θέση της ανοιχτής παρένθεσης του κόμβου $\text{lca}(v, w)$ υπολογίζεται από το $\text{parent}(m + 1)$.

2.1.2.4: Υπολογισμός των συνδέσεων επιθεμάτων

Λήμμα 4

«Σε μία ακολουθία παρενθέσεων P , έστω η θέση v που αναπαριστά έναν κόμβο $\bar{v}$ ο οποίος δεν είναι ρίζα. Τότε η θέση w μίας ανοιχτής παρένθεσης που αναπαριστά τον κόμβο $\text{sl}(v)$ υπολογίζεται ως εξής:»

$$\begin{aligned} x &= \text{rank}_0(v - 1) + 1 \\ y &= \text{rank}_0(\text{findclose}(v)) \\ x &= \Psi[x] \\ y &= \Psi[y] \\ w &= \text{lca}(\text{select}_0(x), \text{select}_0(y)). \end{aligned}$$

Απόδειξη:

Ο αλγόριθμος υπολογίζει πρώτα τα αριστερότερα και τα δεξιότερα φύλλα που είναι απόγονοι του v . Το αριστερότερο φύλλο που είναι απόγονος του v είναι το πρώτο φύλλο που εμφανίζεται μετά το v στην ακολουθία των παρενθέσεων. Ο λεξικογραφικός δείκτης αυτού του φύλλου υπολογίζεται από το $x = \text{rank}_0(v-1)+1$. Όσον αφορά τον δείκτη του δεξιότερου φύλλου, επειδή όλα τα φύλλα κάτω από το v κωδικοποιούνται σε ακολουθίες παρενθέσεων, ανάμεσα στις ανοιχτές και κλειστές παρενθέσεις που αναπαριστούν το v , η λεξικογραφική σειρά του δεξιότερου φύλλου υπολογίζεται από το $y = \text{rank}_0(P, \text{findclose}(v))$. Τα φύλλα αναπαριστούν τα επιθέματα $T_{\text{SA}[x]}$ και $T_{\text{SA}[y]}$. Το $\text{leaf}(x')$ και το $\text{leaf}(y')$ αναπαριστούν τα επιθέματα

$T_{SA[x']} = T_{SA[x]+1}$ και $T_{SA[y']} = T_{SA[y]+1}$, αντίστοιχα. Έστω $l = \text{lcp}(T_{SA[x]}, T_{SA[y]})$. Τότε το l ισούται με το string-depth του κόμβου v επειδή τα $\text{leaf}(x)$ και $\text{leaf}(y)$ είναι ο αριστερότερος και δεξιότερος απόγονος του v . Επιπλέον, $l-1 = \text{lcp}(T_{SA[x']}, T_{SA[y']})$. Τότε ο κόμβος $\text{lca}(\text{select}_0(P, x'), \text{select}_0(P, y'))$, έχει string-depth $l-1$ που αντιστοιχεί στο $\text{sl}(v)$.

2.1.3: Suffix arrays [5]

1	15	aaabababbabbbabba#
2	16	aabababbabbbabba#
3	31	a#
4	13	abaaabababbabbbabba#
5	17	abababbabbbabba#
6	19	ababbabbbabba#
7	28	abba#
8	10	abbabaaabababbabbbabba#
9	7	abbabbabaaabababbabbbabba#
10	4	abbabbabbabaaabababbabbbabba#
11	1	abbabbabbabbabaaabababbabbbabba#
12	21	abbabbbabba#
13	24	abbbabba#
14	32	#
15	14	baaabababbabbbabba#
16	30	ba#
17	12	babaaabababbabbbabba#
18	18	bababbabbbabba#
19	27	babba#
20	9	babbabaaabababbabbbabba#
21	6	babbabbabaaabababbabbbabba#
22	3	babbabbabbabaaabababbabbbabba#
23	20	babbabbbabba#
24	23	babbbabba#
⋮	⋮	⋮
32	25	bbbabba#

Εικόνα 4: Suffix array κατασκευασμένο από ένα κείμενο T

$T = \text{"abbabbabbabbabaaabababbabbbabba\#"}'$, όπου $a < \# < b$.

(Πηγή: Grossi, R. & Vitter, J.Sc., 2005, σελ. 4).

Έστω ένα κείμενο T . Ένα suffix array είναι μια ταξινομημένη λίστα όλων των επιθεμάτων του T . Όταν ένα suffix array συνοδεύεται από πληροφορίες σχετικά με τα μακρύτερα κοινά προθέματα (longest common prefixes – lcp) γειτονικών στοιχείων

στο suffix array, η αναζήτηση συμβολοσειρών μπορεί να πραγματοποιηθεί σε χρόνο $O(m + \log n)$, όπου m είναι το μέγεθος του query και n είναι το μήκος το κειμένου. Το suffix array και οι συνοδευτικές πληροφορίες για τα lcp του, καταλαμβάνουν $2n$ ακέραιους και οι αναζητήσεις απαιτούν το πολύ $P + \lceil \log_2 (n - 1) \rceil$ συγκρίσεις μεμονωμένων συμβόλων. Ο πιο γρήγορος τρόπος για να κατασκευάσουμε ένα suffix array είναι να κατασκευάσουμε ένα suffix tree πρώτα και έπειτα να κατασκευάσουμε το suffix array διασχίζοντας το suffix tree χρησιμοποιώντας τις πληροφορίες για τα lcp.

2.1.3.1: Αναζήτηση στα suffix arrays

Έστω ένα κείμενο T μήκους n από ένα αλφάβητο Σ μεγέθους $|\Sigma|$. Ορίζουμε ως T_i το επίθεμα του T που ξεκινάει από τη θέση i . Για να πραγματοποιήσουμε αναζήτηση στο κείμενο, πρέπει πρώτα να κατασκευάσουμε το suffix array και να ταξινομήσουμε λεξικογραφικά τα επιθέματά του, καθώς και να έχουμε υπολογίσει τις πληροφορίες για τα lcp. Οι εφευρέτες των suffix arrays παραθέτουν τη δική τους μεθοδολογία κατασκευής του ταξινομημένου αυτού πίνακα.

Η ταξινόμηση ολοκληρώνεται σε $\log_2 (n + 1)$ στάδια. Στο πρώτο στάδιο, τα επιθέματα τοποθετούνται σε buckets σύμφωνα με το πρώτο τους σύμβολο. Κάθε επόμενο στάδιο διαχωρίζει ακόμα περισσότερο τα buckets ταξινομώντας τα σύμφωνα με το διπλάσιο αριθμό των συμβόλων. Τα στάδια αριθμούνται με βάση τον αριθμό των συμβόλων που έχουν επηρεαστεί, δηλαδή μετά το στάδιο 1 έχουμε το στάδιο 2 (επειδή επηρεάστηκαν δύο σύμβολα) και μετά από αυτό έχουμε το στάδιο 4 (επειδή επηρεάστηκαν τέσσερα σύμβολα). Έτσι, σε κάποιο επόμενο στάδιο G θα έχουν επηρεαστεί G σύμβολα. Σε έναν πίνακα BH αποθηκεύονται Boolean τιμές οι οποίες οριοθετούν το διαχωρισμό των επιθεμάτων σε m_1 buckets ($m_1 \leq |\Sigma|$). Κάθε bucket διατηρεί τα επιθέματα που έχουν ίδιο το πρώτο τους σύμβολο. Μετά το στάδιο G , τα επιθέματα διαχωρίζονται σε m_G buckets όπου ο καθένας διατηρεί επιθέματα με ίδια τα πρώτα G σύμβολα. Το αποτέλεσμα αυτής της ταξινόμησης αποθηκεύεται σε έναν πίνακα SA , ο οποίος είναι το suffix array. Το $SA[k]$ συμβολίζει τη θέση που ξεκινάει το k -οστό μικρότερο επίθεμα του T . Ένα παράδειγμα φαίνεται στην Εικόνα 4. Η δεύτερη στήλη αναπαριστά το περιεχόμενο του suffix array για ένα κείμενο $T =$

“abbabbabbabbabaaabababbabbbabba#”. Οι αριθμοί αυτοί είναι οι δείκτες των επιθεμάτων του κειμένου με λεξικογραφική σειρά.

Ο υπολογισμός των πληροφοριών για τα lcp πραγματοποιείται κατά τη διάρκεια της ταξινόμησης. Η κύρια ιδέα του υπολογισμού είναι η ακόλουθη. Υποθέτουμε ότι μετά το στάδιο G της ταξινόμησης ξέρουμε τα lcp ανάμεσα σε επιθέματα σε γειτονικά buckets (μετά το πρώτο στάδιο, τα lcp αυτά είναι 0). Στο στάδιο $2G$ τα buckets διαχωρίζονται σύμφωνα με $2G$ σύμβολα. Έτσι, τα lcp ανάμεσα σε επιθέματα σε νέα γειτονικά buckets πρέπει να είναι τουλάχιστον G και το πολύ $2G-1$. Μπορούμε να προσδιορίσουμε τα lcp ανάμεσα σε οποιαδήποτε δύο επιθέματα σε χρόνο $O(\log n)$ χρησιμοποιώντας μία ισορροπημένη δομή δέντρου η οποία καταγράφει το ελάχιστο ζεύγος lcp από ένα σύνολο από διαστήματα του suffix array. Η δομή αυτή ονομάζεται δέντρο διαστήματος (interval tree).

Έστω ότι θέλουμε να βρούμε όλες τις εμφανίσεις της συμβολοσειράς W που έχει μήκος $p \leq n$ στο κείμενο T . Αφού έχουμε κατασκευάσει το ταξινομημένο suffix array και έχουμε υπολογίσει τις πληροφορίες για τα lcp, μπορούμε τώρα να πραγματοποιήσουμε μία δυαδική αναζήτηση στο suffix array σε χρόνο $O(m + \log n)$. Οι πληροφορίες που έχουμε υπολογίσει για τα lcp μας βοηθούν στο να καταλάβουμε αν θα πρέπει να ανατρέξουμε στην αριστερή ή στη δεξιά πλευρά του μεσαίου σημείου M στη δυαδική αναζήτηση.

2.1.3.2: Προβλήματα χωρητικότητας [6]

Τα μεγέθη των συνόλων δεδομένων μπορεί να γίνουν πολύ μεγάλα και η χωρητικότητα είναι συχνά ένα κρίσιμο ζήτημα. Αυτό μπορούμε να το δείξουμε με μία πιο προσεκτική μελέτη των απαιτήσεων χώρου.

Υποθέτουμε ότι κάθε σύμβολο στο κείμενο T κωδικοποιείται από $\log_2 |\Sigma|$ bits, συνολικά $n \log_2 |\Sigma|$ bits. Στα suffix arrays, οι θέσεις των n επιθεμάτων του T αποθηκεύονται χρησιμοποιώντας $n \log_2 n$ bits. Τα suffix trees απαιτούν πολύ περισσότερη χωρητικότητα: ανάμεσα σε $4n \log_2 n$ bits και $5n \log_2 n$ bits. Θεωρητικά, αν το αλφάβητο είναι πολύ μεγάλο, συγκεκριμένα αν $\log_2 |\Sigma| = \Theta(\log_2 n)$, τότε τα suffix arrays χρειάζονται περίπου τον ίδιο αριθμό bit με το κείμενο. Στην πράξη, όμως, το μέγεθος του αλφαβήτου $|\Sigma|$ είναι μία σταθερά, όπως για παράδειγμα, $|\Sigma| =$

256 σε έγγραφα ηλεκτρονικής μορφής ASCII (ή μεγαλύτερης, όπως η UNICODE), και $|\Sigma| = 4$ σε αλληλουχίες DNA. Σε αυτές τις περιπτώσεις, τα suffix trees και τα suffix arrays είναι μεγαλύτερα από το κείμενο κατά έναν σημαντικό πολλαπλασιαστικό παράγοντα $\Theta(\log_2 |\Sigma| n) = \Theta(\log_2 n)$. Για παράδειγμα, μία αλληλουχία DNA με n σύμβολα (με $|\Sigma| = 4$) μπορεί να αποθηκευτεί με $2n$ σύμβολα σε έναν υπολογιστή. Το suffix array, όμως, γι' αυτήν την ακολουθία χρειάζεται τουλάχιστον n λέξεις των 4 bytes για την κάθε μία ή $32n$ bits, που είναι 16 φορές μεγαλύτερο από το ίδιο το κείμενο.

2.1.4: Συμπιεσμένα suffix arrays [6]

Ορίζουμε το suffix array SA για μία δυαδική συμβολοσειρά T ως έναν αφηρημένο τύπο δεδομένων που υποστηρίζει λειτουργίες συμπίεσης και αναζήτησης. Υποθέτουμε ότι το T είναι μία δυαδική συμβολοσειρά μεγέθους $n-1$ πάνω από το αλφάβητο $\Sigma = \{a, b\}$ και τερματίζεται στην n -οστή θέση από το σύμβολο $\#$, τέτοιο ώστε $a < \# < b$. Όπως έχουμε δει, το suffix array δημιουργείται σύμφωνα με τη λεξικογραφική σειρά των επιθεμάτων του T . Για παράδειγμα, το $SA[i]$ είναι θέση που ξεκινάει το i -οστό επίθεμα σε λεξικογραφική σειρά.

Έστω το suffix array SA για τη δυαδική συμβολοσειρά T και n ο αριθμός των εγγράφων του, ο οποίος θεωρούμε ότι είναι αριθμός μίας δύναμης του 2. Για $k \geq 0$, ξεκινάμε με το suffix array SA_k , το οποίο έχει $n_k = n/2^k$ εγγραφές. Ακολουθούμε τέσσερα κύρια βήματα για να μετατρέψουμε τον πίνακα σε μία ισοδύναμη αλλά πιο συνοπτική αναπαράσταση.

Βήμα 1

Δημιουργούμε ένα δυαδικό διάνυσμα (bit vector) B_k με n_k bits, τέτοιο ώστε $B_k[i] = 1$ αν το $SA_k[i]$ είναι άρτιος και $B_k[i] = 0$ αν το $SA_k[i]$ είναι περιττός.

Βήμα 2

Συνδέουμε κάθε 0 στο B_k με το συνοδευτικό του 1. Ένα συγκεκριμένο 0 είναι συνοδευτικό (companion) ενός συγκεκριμένου 1, αν η περιττή είσοδος στο SA, η οποία σχετίζεται με το 0 είναι μία λιγότερη από την άρτια είσοδο στο SA, η οποία σχετίζεται με το 1. Αυτό μπορούμε να το ορίσουμε με μία μερική συνάρτηση Ψ_k , όπου $\Psi_k(i) = j$, αν και μόνο αν το $SA_k[i]$ είναι άρτιος και $SA_k[j] = SA_k[i] + 1$. Το $\Psi_k(i) = j$ υποδηλώνει ότι $B_k[i] = 0$ και $B_k[j] = 1$. Μπορούμε να μετατρέψουμε το Ψ_k σε μία ολική συνάρτηση, αν ορίσουμε ότι $\Psi_k(i) = i$ όταν το $SA_k[i]$ είναι άρτιος. Άρα, για κάθε $1 \leq i \leq n_k$, έχουμε:

$$\Psi_k(i) = \begin{cases} j & \text{αν } SA_k[j] = SA_k[i] + 1 \\ i & \text{σε κάθε άλλη περίπτωση} \end{cases}$$

Βήμα 3

Υπολογίζουμε το πλήθος των 1 για κάθε πρόθεμα του B_k . Γι' αυτό το σκοπό, χρησιμοποιούμε τη συνάρτηση rank_k . Έτσι, το $\text{rank}_k(j)$, υπολογίζει πόσα 1 υπάρχουν στα πρώτα j bits του B_k .

Βήμα 4

Βάζουμε μαζί τις άρτιες τιμές του SA_k και διαιρούμε την κάθε μία με το 2.

Η Εικόνα 5 απεικονίζει το αποτέλεσμα μίας μόνο εφαρμογής των βημάτων 1-4. Στο παράδειγμα αυτό, $\Psi_0(25) = 16$, $SA_0[25] = 29$ και $SA_0[16] = 30$. Το καινούργιο suffix array SA_1 αποθηκεύει τους δείκτες επιθεμάτων (διααιρεμένους με το 2) για τα επιθέματα που ξεκινούν από άρτιες θέσεις στο αρχικό κείμενο T. Για παράδειγμα, το $SA_1[3] = 5$ μας δείχνει ότι το λεξικογραφικά τρίτο επίθεμα που ξεκινάει από μία άρτια θέση στο T, είναι εκείνο που ξεκινάει από τη θέση $2 \times 5 = 10$, δηλαδή *abbabaa . . . #*.

Διαδικασία compress

Αναπαριστούμε συνοπτικά το SA εκτελώντας τα βήματα 1-4, της φάσης $k = 0, 1, \dots, l - 1$, όπου $l = \Theta(\log_2 \log_2 n)$. Ως αποτέλεσμα έχουμε $l + 1$ επίπεδα πληροφοριών, αριθμημένα ως $0, 1, \dots, l$ τα οποία σχηματίζουν τη συμπιεσμένη αναπαράσταση του suffix array SA ως εξής:

1. Το επίπεδο k , για κάθε $0 \leq k < l$, αποθηκεύει τα B_k , Ψ_k , και rank_k . Το SA_k δεν αποθηκεύεται.
2. Το τελευταίο επίπεδο $k = l$, αποθηκεύει το SA_l , το l -οστό επίπεδο λειτουργικότητας των δομών B_l , Ψ_l , και rank_l δεν είναι απαραίτητη.

Συνάρτηση lookup

Ορίζουμε την συνάρτηση $\text{lookup}(i) = \text{rlookup}(i, 0)$, όπου το $\text{rlookup}(i, k)$ αναφέρεται στην παρακάτω συνάρτηση

συνάρτηση $\text{rlookup}(i, k)$:

αν $k = l$ τότε

επέστρεψε $\text{SA}_l[i]$

αλλιώς

επέστρεψε $2 * \text{rlookup}(\text{rank}_k(\Psi_k(i)), k + 1) + (B_k[i] - 1)$

Αν το k είναι το τελευταίο επίπεδο l , τότε πραγματοποιεί μια άμεση αναζήτηση στο $\text{SA}_l[i]$. Αλλιώς, εκμεταλλεύεται το Λήμμα 3 και την επαγωγική υπόθεση, έτσι ώστε το $\text{rlookup}(i)$ να επιστρέφει την τιμή $2 * \text{rlookup}(\text{rank}_k(\Psi_k(i)), k + 1) + (B_k[i] - 1)$ στο $\text{SA}_k[i]$.

2.2: Suffix Intervals [7]

Έστω ένα κείμενο T μήκους n και $T_1, T_2, \dots, T_n$ τα επιθέματά του, SA το suffix array για το κείμενο T και SA^{-1} ο αντίστροφός του και v η θέση η οποία αναπαριστά τον κόμβο $\bar{v}$ του suffix tree για το κείμενο T . Το $\text{leaf}(T_i)$ υποδηλώνει το φύλλο που αντιστοιχεί στο T_i , για το suffix tree του T . Ορίζουμε το $L(v) = \text{SA}^{-1}[i]$, αν το $\text{leaf}(T_i)$

είναι το αριστερότερο φύλλο του υποδέντρου του $\bar{v}$. Υποθέτουμε ότι τα παιδιά ενός κόμβου στο suffix tree διατηρούνται σε λεξικογραφική σειρά. Τότε, για όλα τα T_j ($i \leq j$) τέτοια ώστε το $\text{leaf}(T_j)$ να είναι ένας απόγονος του v , $T_i \leq T_j$. Παρόμοια ορίζουμε και το $R(v) = \text{SA}^{-1}[i]$, αν το $\text{leaf}(T_i)$ είναι το δεξιότερο φύλλο του υποδέντρου του $\bar{v}$. Τότε, για όλα τα T_j τέτοια ώστε το $\text{leaf}(T_j)$ να είναι ένας απόγονος του v , $T_i \geq T_j$. Έτσι, το $[L(v), R(v)]$ είναι ένα διάστημα (interval) του suffix array που περιέχει ακριβώς όλα τα επιθέματα T_j για τα οποία το $\text{leaf}(T_j)$ είναι ένας απόγονος του v .

Στο ακόλουθο λήμμα υποθέτουμε ότι τα παιδιά ενός κόμβου σε ένα suffix tree διατηρούνται σε λεξικογραφική σειρά.

Λήμμα 6

«Έστω ότι έχουμε μια συμβολοσειρά T και το suffix tree της. Έστω ένας κόμβος v στο suffix tree και $v_1, \dots, v_r$ τα παιδιά του. Έστω $1 \leq i \leq j \leq r$ και $[L(v_i), R(v_j)]$ ένα διάστημα του suffix array. Τότε $k \in [L(v_i), R(v_j)]$ αν και μόνο αν το $\text{leaf}(T_k)$ βρίσκεται σε ένα από τα υποδέντρα τα οποία έχουν ρίζα στο $v_i, \dots, v_j$.»

Έτσι, καταλήγουμε στον ακόλουθο ορισμό.

Ορισμός 6

«Έστω μία συμβολοσειρά T και $\{T_{i_1}, \dots, T_{i_n}\}$ η λεξικογραφική σειρά των επιθεμάτων της. Το διάστημα $[j, k] = \{i_j, \dots, i_k\}$, για $j \leq k$, ονομάζεται διάστημα επιθέματος (suffix interval).»

Από το Λήμμα 6, για τα παιδιά του v και για κάθε $1 \leq i \leq j \leq r$, το suffix interval $[L(v_i), R(v_j)]$ το ονομάζουμε (i,j) -suffix interval.

2.3: Suffix Trays [7]

Είναι μία δομή δεδομένων η οποία προέρχεται από των συνδυασμό των suffix trees και των suffix arrays και στόχο έχει την μείωση του χρόνου στην απάντηση των queries, κάνοντάς την πιο γρήγορη από τα suffix arrays και τα suffix trees.

Τα suffix trays χρησιμοποιούν την έννοια των διαστημάτων (intervals), η οποία είναι κοινή και στα suffix trees και στα suffix arrays. Για τα suffix trays θα δημιουργήσουμε ειδικούς κόμβους, οι οποίοι αντιστοιχούν στα suffix intervals, και ονομάζονται κόμβοι επιθέματος διαστήματος (suffix interval nodes). Ένα μέρος του suffix tray είναι ένα suffix array. Κάθε suffix interval κόμβος μπορεί να θεωρηθεί ως ένας κόμβος ο οποίος διατηρεί τα τελικά σημεία του διαστήματος εντός του ολοκληρωμένου suffix array.

Διατηρούμε πίνακες μήκους $|\Sigma|$ σε ένα επιλεγμένο υποσύνολο κόμβων, το οποίο υποσύνολο δεν πρέπει να περιέχει παραπάνω από $n/|\Sigma|$ κόμβους (όπου $|\Sigma|$ είναι το μέγεθος του αλφάβητου).

Ορισμός 7

«Έστω T μια συμβολοσειρά μεγέθους n από ένα αλφάβητο Σ . Ένας κόμβος v στο suffix tree του T , ονομάζεται σ-κόμβος αν ο αριθμός των φύλλων στο υποδέντρο του suffix tree το οποίο έχει ως ρίζα το v είναι τουλάχιστον $|\Sigma|$. Ένας σ-κόμβος v ονομάζεται διακλαδωμένος-σ-κόμβος αν τουλάχιστον δύο από τα παιδιά του v στο suffix tree είναι σ-κόμβοι και καλείται σ-φύλλο αν όλα τα παιδιά του στο suffix tree δεν είναι σ-κόμβοι.»

Αν ένας κόμβος v είναι σ-κόμβος, τότε όλοι οι πρόγονοί του είναι και εκείνοι σ-κόμβοι. Αυτό σημαίνει ότι σε ένα υποδέντρο σ-φύλλου δεν υπάρχουν σ-κόμβοι.

Λήμμα 7

«Έστω T μια συμβολοσειρά μεγέθους n από ένα αλφάβητο Σ . Ο αριθμός των διακλαδωμένων-σ-κόμβων στο suffix tree του T , αντιστοιχεί σε $O(n/|\Sigma|)$.»

Απόδειξη: Ο αριθμός των σ-φύλλων είναι το πολύ $n/|\Sigma|$ γιατί όλα έχουν τουλάχιστον $|\Sigma|$ φύλλα στο υποδέντρο τους και τα υποδέντρα τους είναι ξένα (disjoint). Έστω Δ το δέντρο που επηρεάζεται από τους σ-κόμβους και συνάπτεται στους διακλαδωμένους-σ-κόμβους και τα σ-φύλλα μόνο. Τότε το Δ είναι ένα δέντρο με $n/|\Sigma|$ φύλλα και κάθε εσωτερικός του κόμβος έχει τουλάχιστον δύο παιδιά.

Αυτό σημαίνει ότι μπορούμε να διατηρήσουμε πίνακες σε κάθε διακλαδωμένο-σ-κόμβο.

2.3.1: Δομή ενός suffix tray

Ένα suffix tray κατασκευάζεται από ένα suffix tree. Το suffix tray περιέχει όλους τους σ-κόμβους του suffix tree. Ως παιδιά των σ-κόμβων, εισάγουμε κάποιους suffix interval κόμβους.

Κάθε σ-κόμβος μετατρέπεται κατάλληλα από το suffix tree στο suffix tray. Για ένα σ-φύλλο v , το v μετατρέπεται σε έναν suffix interval κόμβο με suffix interval $[L(v), R(v)]$. Για κάθε σ-κόμβο v που δεν είναι φύλλο, έστω $v_1, \dots, v_r$ τα παιδιά του v στο suffix tree και $vl_1, \dots, vl_x$ το υποσύνολο των παιδιών που είναι σ-κόμβοι. Τότε θα είναι στο suffix tray με διεμπλεκόμενους (interleaving) suffix interval κόμβους και σ-κόμβους, για παράδειγμα $(1, l_1-1)$ -suffix interval κόμβος, $vl_1, (l_1+1, l_2-1)$ -suffix interval κόμβος, $vl_2, \dots, vl_x, (l_x+1, r)$ -suffix interval κόμβος.

Σε κάθε διακλαδωμένο-σ-κόμβο v στο suffix tray, διατηρούμε έναν πίνακα μεγέθους $|\Sigma|$, τον οποίο συμβολίζουμε με A_v και περιέχει τα ακόλουθα δεδομένα. Για κάθε παιδί w του v που είναι σ-κόμβος, η θέση τ στο A_v όπου το τ είναι ο πρώτος χαρακτήρας στο $\text{edge}(v, w)$, δείχνει στο w . Οι υπόλοιπες θέσεις στο A_v δείχνουν στον κατάλληλο suffix interval κόμβο ή σε έναν δείκτη NIL, αν δεν υπάρχει κανένα τέτοιο suffix interval.

Σε κάθε σ-κόμβο v που δεν είναι ούτε διακλαδωμένος-σ-κόμβος ούτε σ-φύλλο, αποθηκεύουμε τον πρώτο χαρακτήρα τ στο $\text{edge}(v, w)$, τον οποίο αποκαλούμε διαχωριστικό χαρακτήρα (separating character).

Έτσι, το suffix tray έχει γραμμικό μέγεθος.

Λήμμα 8

«Εστω T μια συμβολοσειρά μεγέθους n . Τότε το μέγεθος του suffix tray για το T είναι $O(n)$.»

Απόδειξη:

Το suffix array είναι σίγουρα μεγέθους $O(n)$ και ο αριθμός των suffix interval κόμβων είναι οριοθετημένος από τον αριθμό των κόμβων στο suffix tree. Επίσης, για κάθε μη διακλαδωμένο-σ-κόμβο, οι βοηθητικές πληροφορίες είναι σταθερού μεγέθους.

Οι βοηθητικές πληροφορίες που είναι αποθηκευμένες σε κάθε διακλαδωμένο-σ-κόμβο, έχουν μέγεθος $O(|\Sigma|)$. Από το Λήμμα 7, υπάρχουν $n/|\Sigma|$ διακλαδωμένοι-σ-κόμβοι. Ως εκ τούτου, όλα αυτά έχουν μέγεθος $O(n)$.

2.3.2: Αναζήτηση query

Εστω w ένα παιδί του κόμβου v που είναι σ-κόμβος. Όταν λάβουμε ένα query $P = p_1 \dots p_m$, ξεκινάμε να διασχίζουμε το suffix tray από την ρίζα. Ας υποθέσουμε ότι έχουμε διασχίσει το suffix tray με $p_1 \dots p_{i-1}$ και πρέπει να συνεχίσουμε με το $p_i \dots p_m$. Σε κάθε διακλαδωμένο-σ-κόμβο προσπελαύνουμε τη θέση p_i του πίνακα A_v για να μάθουμε ποιόν κόμβο του suffix tray πρέπει να αναζητήσουμε. Για άλλους σ-κόμβους που δεν είναι ούτε διακλαδωμένοι-σ-κόμβοι ούτε σ-φύλλα, συγκρίνουμε το p_i με τον διαχωριστικό χαρακτήρα τ . Αυτοί οι κόμβοι έχουν μόνο ένα παιδί που είναι σ-κόμβος.

Έτσι, στο suffix tray, τα παιδιά του v είναι:

1. Ένας suffix interval κόμβος στα αριστερά του w ,
2. Το w ,
3. Ένας suffix interval κόμβος στα δεξιά του w .

Αν $p_i < \tau$, αναζητούμε τον suffix interval κόμβο στα αριστερά του w . Αν $p_i > \tau$, αναζητούμε τον suffix interval κόμβο στα δεξιά του w . Αν $p_i = \tau$, αναζητούμε το

μοναδικό παιδί του v που είναι ένας σ -κόμβος. Αν το v είναι ένα σ -φύλλο, τότε βρισκόμαστε στο suffix interval του v .

Για να πραγματοποιήσουμε αναζήτηση μέσα σε ένα suffix interval $[j,k]$, εφαρμόζουμε την τυπική μέθοδο αναζήτησης suffix array, ξεκινώντας με τα όρια $[j,k]$. Ο χρόνος αναζήτησης μιας τέτοιας δομής είναι $O(m + \log I)$, όπου το I είναι το μέγεθος του διαστήματος.

Λήμμα 9

«Κάθε suffix interval σε ένα suffix tray έχει μέγεθος $O(|\Sigma|^2)$.»

Απόδειξη:

Έστω ένα (i,j) -suffix interval, για παράδειγμα το διάστημα $[L(v_i), R(v_j)]$, το οποίο προέρχεται από έναν κόμβο v με παιδιά $v_1, \dots, v_r$. Από το Λήμμα 6, το (i,j) -suffix interval περιέχει τις καταλήξεις οι οποίες αναπαριστώνται από τα φύλλα στο υποδέντρο του $v_i, \dots, v_j$. Όμως, τα $v_i, \dots, v_j$ δεν είναι σ -κόμβοι. Έτσι, κάθε υποδέντρο αυτών των κόμβων περιέχει το πολύ $|\Sigma|-1$ φύλλα. Επειδή $j - i + 1 \leq |A|$, το συνολικό μέγεθος του (i,j) -suffix interval είναι $O(|\Sigma|^2)$.

Ένα suffix interval $[L(v), R(v)]$, διατηρείται μόνο για σ -φύλλα. Καθώς κανένα από τα παιδιά του v δεν είναι σ -κόμβος, αυτή είναι μια ειδική περίπτωση για το (i,j) -suffix interval.

Από τα παραπάνω και από το Λήμμα 9, ο χρόνος που απαιτείται για την απάντηση ενός query είναι $O(m + \log |\Sigma|)$.

2.4: Balanced Indexing Structures και Suffix Trists [7]

Μία ισορροπημένη δομή ευρετηρίου (Balanced Indexing Structure – BIS) είναι ένα δυαδικό δέντρο αναζήτησης πάνω από τα επιθέματα, όπου η διάταξη είναι λεξικογραφική. Επιπλέον, ένα BIS υποστηρίζει query ευρετηρίασης σε χρόνο $O(m + \log n)$, όπου m είναι το μέγεθος του query.

Αυτό οδήγησε στην ιδέα της κατασκευής ενός suffix trist. Το suffix trist προέρχεται από τον συνδυασμό των suffix trees και τη δυναμική έκδοση των suffix arrays, δηλαδή τα suffix lists.

2.4.1: Εισαγωγή νέων κόμβων στο BIS

Όταν πραγματοποιούμε μία επέκταση κειμένου από T σε aT , το suffix tree ενημερώνεται ώστε να αναπαραστήσει το καινούργιο κείμενο aT . Πιο συγκεκριμένα, ένα καινούργιο φύλλο το οποίο ανταποκρίνεται στη νέα κατάληξη, εισάγεται στο suffix tree. Είναι πιθανόν να εισαχθεί και ένας καινούργιος εσωτερικός κόμβος, ο οποίος θα είναι ο γονέας του καινούργιου φύλλου. Αυτό συμβαίνει σε περίπτωση που η νέα κατάληξη αποκλίνει από μία ακμή σε μία θέση που δεν προϋπήρχε κανένας κόμβος. Σε αυτή την περίπτωση, η ακμή πρέπει να χωριστεί στα δύο και να εισαχθεί ένας καινούργιος εσωτερικός κόμβος σε εκείνο το σημείο.

Λήμμα 10

«Για έναν κόμβο v σε ένα suffix tree, αν το v δεν είναι σ -κόμβος τότε όλα τα φύλλα στο υποδέντρο του v ανήκουν στο ίδιο BIS.»

Θα ήταν εύχρηστο να διατηρήσουμε έναν δείκτη $\text{leaf}(v)$ προς κάποιο φύλλο στο υποδέντρο του v για κάθε κόμβο v στο suffix tree.

Για να βρούμε σε ποιο BIS πρέπει να εισαχθεί ο καινούργιος κόμβος, πρέπει να λάβουμε υπ' όψιν δύο περιπτώσεις. Πρώτον, πρέπει να εξετάσουμε την περίπτωση όπου το νέο φύλλο v στο suffix tree έχει εισαχθεί ως ένα παιδί ενός ήδη υπάρχον εσωτερικού κόμβου w . Αν το w δεν είναι σ -κόμβος, τότε από το Λήμμα 10 ξέρουμε ότι το $\text{leaf}(w)$ και το v πρέπει να βρίσκονται στο ίδιο BIS. Διασχίζοντας το BIS προς τα πάνω, από το $\text{leaf}(w)$ ως τη ρίζα (σε χρόνο $O(\log |\Sigma|)$), μπορούμε να βρούμε την ρίζα του BIS η οποία πρέπει να περιέχει τον νέο κόμβο v .

Έπειτα, πρέπει να εξετάσουμε την περίπτωση όπου το καινούργιο φύλλο v στο suffix tree έχει εισαχθεί ως παιδί ενός νέου εσωτερικού κόμβου w . Έστω u ο γονέας του w

και v' το άλλο παιδί του w . Πρώτα αγνοούμε τελείως το v με το να χειριστούμε το νέο δέντρο ως το suffix tree του T , όπου το edge (u, v') έχει χωριστεί στα δύο, δημιουργώντας τον καινούργιο κόμβο w . Αφού ενημερώσουμε το trist ώστε να συμπεριλαμβάνει το w , μπορούμε να εισάγουμε το v , όπως θα κάναμε και στην περίπτωση που το w ήταν ήδη ένας εσωτερικός κόμβος. Για να καθορίσουμε την κατάσταση του w , παρατηρούμε ότι το w δεν μπορεί να είναι διακλαδωμένος-σ-κόμβος. Επιπλέον, παρατηρούμε ότι ο αριθμός των φύλλων στο υποδέντρο του w είναι ο ίδιος με τον αριθμό των φύλλων στο υποδέντρο το οποίο έχει ως ρίζα του το v' (αν αγνοήσουμε το v). Αν το v' δεν είναι σ-κόμβος, τότε το w δεν είναι σ-κόμβος. Σε διαφορετική περίπτωση, το w είναι σ-κόμβος με ένα διαχωριστικό χαρακτήρα ο οποίος είναι ο πρώτος χαρακτήρας της ετικέτας edge (w, v') . Παρατηρούμε ότι όλη η διαδικασία τελειώνει σε χρόνο $O(\log |\Sigma|)$, όπως απαιτείται.

2.4.2: Εντοπίζοντας έναν καινούργιο σ-κόμβο

Έστω v ένας καινούργιος σ-κόμβος και w ο γονέας του. Ακριβώς πριν το v γίνει σ-κόμβος:

1. Το w πρέπει ήδη να είναι σ-κόμβος
2. $v \in \{w_i, \dots, w_j\}$ και σχετίζεται με ένα (i,j) -suffix interval που αντιπροσωπεύεται από έναν suffix interval κόμβο u ο οποίος είναι παιδί του w στο suffix trist.

Έτσι, θα μπορέσουμε να εντοπίσουμε πότε ένας καινούργιος σ-κόμβος έχει δημιουργηθεί, αν διατηρήσουμε μετρητές για καθέναν από τους κόμβους του suffix tree $w_i, \dots, w_j$ για να μετρήσουμε τον αριθμό των φύλλων στα υποδέντρα τους. Αυτοί οι μετρητές διατηρούνται σε ένα δυαδικό δέντρο αναζήτησης, το οποίο συσχετίζουμε με το BIS και κάθε μετρητής w_k ευρετηριάζεται από τον πρώτο χαρακτήρα στο edge (w, w_k) .

Η ενημέρωση των μετρητών γίνεται ως εξής. Όταν ένα νέο φύλλο εισάγεται στο BIS ενός suffix trist στον suffix interval κόμβο u του BIS, όπου v είναι ο γονέας του u , πρέπει να αυξήσουμε τον μετρητή του w_k στο BIS, όπου το w_k είναι ο κόμβος (από τους κόμβους $w_i, \dots, w_j$) ο οποίος είναι ένας πρόγονος του καινούργιου φύλλου. Ο μετρητής μπορεί να βρεθεί σε χρόνο $O(\log |\Sigma|)$.

Παρατηρούμε ότι όταν ένας καινούργιος εσωτερικός κόμβος εισάγεται στο suffix tree, είναι πιθανό ότι ο καινούργιος εσωτερικός κόμβος είναι τώρα ένας από τους κόμβους $w_i, \dots, w_j$ για ένα (i,j) -suffix interval. Σε μια τέτοια περίπτωση, όταν εισαχθεί ο νέος κόμβος, αντιγράφει τον αριθμό των φύλλων στο υποδέντρο του από το μοναδικό παιδί του, καθώς το παιδί διατηρούσε τον αριθμό των φύλλων στο υποδέντρο του.

2.4.3: Ενημερώνοντας τον καινούργιο σ-κόμβο

Έστω v ένας καινούργιος σ-κόμβος (ο οποίος είναι σ-φύλλο) και w ο γονέας του. Η ενημέρωση του νέου σ-κόμβου απαιτεί δύο πράγματα:

1. Πρέπει να χωρίσουμε το BIS σε τρία τμήματα. Δύο νέα BIS και το νέο σ-φύλλο που χωρίζεται ανάμεσά τους.
2. Για το νέο σ-φύλλο θα χρειαστεί να προσθέσουμε τον διαχωριστικό χαρακτήρα και να δημιουργήσουμε ένα καινούργιο σετ από μετρητές για τα παιδιά του v .

Για το πρώτο, πρέπει να διαχωρίσουμε το BIS που έχει μόλις ενημερωθεί, σε τρία τμήματα: τους κόμβους που αντιστοιχούν στα επιθέματα στο υποδέντρο του v , στους κόμβους που αντιστοιχούν στα επιθέματα τα οποία είναι λεξικογραφικά μικρότερα από τα επιθέματα στο υποδέντρο του v και στους κόμβους οι οποίοι αντιστοιχούν σε επιθέματα που είναι λεξικογραφικά μεγαλύτερα από τα επιθέματα στο υποδέντρο του v .

Για το δεύτερο, χρειάζεται να δημιουργήσουμε νέους μετρητές. Ορίζουμε τα παιδιά του v ως $v_1, \dots, v_k$. Παρατηρούμε ότι ο αριθμός των επιθεμάτων σε ένα υποδέντρο του v_i μπορεί να υπολογιστεί σε χρόνο $O(\log |\Sigma|)$. Υπάρχει αρκετός χρόνος για να ενημερωθούν όλοι οι μετρητές των $v_1, \dots, v_k$ προτού κάποιος απ' αυτούς να γίνει σ-κόμβος, διατηρώντας το όριο $O(\log |\Sigma|)$ ανά ενημέρωση.

Πιο συγκεκριμένα, θα ενημερώσουμε τους μετρητές κατά τις πρώτες k εισαγωγές στο BIS του v (ακολουθώντας το γεγονός ότι το v γίνεται σ-κόμβος). Σε κάθε εισαγωγή ενημερώνουμε έναν από τους μετρητές.

Λήμμα 11

«Έστω v ένας κόμβος στο suffix tree και $v_1, \dots, v_k$ τα παιδιά του. Έστω ότι το v έχει μόλις γίνει ένας σ -κόμβος. Τότε, εκείνη τη στιγμή, ο αριθμός των φύλλων σε καθένα από τα υποδέντρα των παιδιών του v είναι το πολύ $|\Sigma| - k + 1$.»

Απόδειξη:

Ας υποθέσουμε ότι αυτό δεν ισχύει. Συγκεκριμένα, ας υποθέσουμε ότι το παιδί v_i έχει $|\Sigma| - k + 2$ φύλλα στο υποδέντρο του εκείνη τη στιγμή. Βλέπουμε ότι ο αριθμός των φύλλων σε καθένα από τα υποδέντρα είναι τουλάχιστον ένα. Έτσι, ο αριθμός των φύλλων σε όλα τα υποδέντρα των $v_1, \dots, v_k$ είναι τουλάχιστον $|\Sigma| - k + 2 + k - 1 = |\Sigma| + 1$, που έρχεται σε αντίθεση με το γεγονός ότι το v έχει μόλις γίνει ένας σ -κόμβος (θα έπρεπε ήδη να είναι σ -κόμβος).

2.4.4: Όταν ένας σ -κόμβος γίνεται διακλαδωμένος- σ -κόμβος

Έστω v ένας σ -κόμβος που αλλάζει την κατάστασή του σε διακλαδωμένο- σ -κόμβο. Μόλις πριν να αλλάξει την κατάστασή του, είχε ακριβώς ένα παιδί v_j το οποίο ήταν σ -κόμβος. Η αλλαγή στην κατάσταση πρέπει να γίνει επειδή ένα άλλο παιδί στο suffix tree, έστω το v_i έχει γίνει σ -φύλλο. Τώρα που το v έχει δύο παιδιά τα οποία είναι σ -κόμβοι, έχει γίνει διακλαδωμένος- σ -κόμβος.

Ακριβώς πριν γίνει διακλαδωμένος- σ -κόμβος, το v περιείχε έναν διαχωριστικό χαρακτήρα τ , τον πρώτο χαρακτήρα στο edge (v, v_j) και δύο suffix interval κόμβους w και x που αντιστοιχούν στο αριστερό και το δεξί διάστημα του v . Τώρα που το v_i έγινε σ -φύλλο, το w χωρίστηκε σε τρία τμήματα, όπως περιγράφηκε σε προηγούμενη υποενότητα. Υποθέτουμε ότι το v_i βρίσκεται πριν απ' το v_j στη λίστα με τα παιδιά του v .

Έτσι, στο suffix trist, τα παιδιά του v είναι:

1. Ένας suffix interval κόμβος w_L
2. Ένα σ -φύλλο v_i

3. Ένας suffix interval κόμβος w_R
4. Ένας σ -κόμβος v_j
5. Ένας suffix interval κόμβος x

Ορίζουμε με B_1 , B_2 και B_3 τα BIS που αναπαριστούν τους suffix interval κόμβους w_L , w_R και x .

Λήμμα 12

«Από τη στιγμή που το w γίνει διακλαδωμένος- σ -κόμβος, χρειάζεται να γίνουν τουλάχιστον $|\Sigma|$ εισαγωγές στο B_1 , B_2 ή B_3 προτού κάποιος κόμβος στο υποδέντρο του v στο suffix tree που δεν βρίσκεται στα υποδέντρα του v_i ή του v_j , γίνει διακλαδωμένος- σ -κόμβος.»

Απόδειξη:

Εκείνη τη στιγμή, κάθε κόμβος στο υποδέντρο του v που δεν βρίσκεται στο υποδέντρο του v_i ή του v_j έχει λιγότερα από $|\Sigma|$ φύλλα στο υποδέντρο του. Παρατηρούμε ότι κάθε διακλαδωμένος- σ -κόμβος πρέπει να έχει τουλάχιστον $2|\Sigma|$ φύλλα στο υποδέντρο του, καθώς έχει τουλάχιστον δύο παιδιά τα οποία είναι σ -κόμβοι όπου ο καθένας έχει τουλάχιστον $|\Sigma|$ φύλλα. Έτσι, για έναν κόμβο στο υποδέντρο του v ο οποίος δεν βρίσκεται στο υποδέντρο του v_i ή του v_j , χρειάζεται να προστεθούν τουλάχιστον $|\Sigma|$ φύλλα στο υποδέντρο του για να γίνει διακλαδωμένος- σ -κόμβος.

Κεφάλαιο 3^ο: Σύγκριση δομών δεδομένων

Η έννοια του suffix tree εφευρέθηκε από τον Peter Weiner στα πλαίσια της εργασίας του τον οργανισμό ‘The Rand Corporation’, στη Σάντα Μόνικα της Καλιφόρνια το 1973 [8], ενώ δύο δεκαετίες περίπου αργότερα, το 1989, στο τμήμα της Επιστήμης των Υπολογιστών του πανεπιστημίου της Αριζόνα δημιουργήθηκε μία νέα εννοιολογικά απλή δομή δεδομένων από τους Udi Manber και Gene Meyers την οποία ονόμασαν suffix arrays. Και οι δύο αυτές δομές έλυσαν έναν μεγάλο αριθμό προβλημάτων που σχετίζονταν κυρίως με την αναζήτηση συμβολοσειρών. Έχουν ευρύ φάσμα εφαρμογών, καθώς πέρα από την πληροφορική, χρησιμοποιούνται στη και σε πολλούς άλλους τομείς.

Τα suffix arrays υπερτερούν σε θέματα χωρητικότητας έναντι των suffix trees. Το κύριο πλεονέκτημα των suffix arrays είναι ότι χρησιμοποιούν τρεις με πέντε φορές λιγότερη χωρητικότητα απ’ ότι τα suffix trees. Στα suffix trees, ένα query μπορεί να απαντηθεί σε χρόνο $O(m \log |\Sigma|)$, όπου το m είναι το μέγεθος του query και $|\Sigma|$ είναι το μέγεθός του αλφάβητου, ενώ για τα suffix arrays ο χρόνος αυτός είναι $O(m + \log n)$, όπου το n είναι το μήκος του κειμένου. Ο χρόνος αναζήτησης χρησιμοποιώντας τα suffix arrays είναι συγκρίσιμος με (και σε μερικές περιπτώσεις είναι σχετικά καλύτερος από) των suffix trees. Στη χειρότερη περίπτωση, τα suffix trees μπορούν να κατασκευαστούν σε χρόνο $O(n)$, ενώ τα suffix arrays σε χρόνο $O(n \log n)$. Οι δημιουργοί του επισημαίνουν ότι ο χρόνος αυτός μπορεί να μειωθεί σε αναμενόμενο χρόνο $O(n)$, αν κατασκευάσουμε το suffix array διασχίζοντας το suffix tree και χρησιμοποιώντας πληροφορίες για το μικρότερο κοινό πρόθεμα, αλλά αυτό έχει ως αποτέλεσμα τη μικρότερη αποδοτικότητα χωρητικότητας.

Οι διαφορές στους χρόνους οφείλονται στο διαφορετικό τρόπο απάντησης των queries. Σε ένα suffix tree μπορούμε να απαντήσουμε ένα query διασχίζοντας το δέντρο από τη ρίζα. Σε κάθε κόμβο πρέπει να ξέρουμε πώς να συνεχίσουμε την διάσχιση και πρέπει να αποφασίσουμε ανάμεσα από $|\Sigma|$ ταξινομημένες επιλογές. Αυτό εξηγεί τον παράγοντα $O(\log |\Sigma|)$. Από την άλλη μεριά, σε ένα suffix array πραγματοποιούμε δυαδική αναζήτηση σε όλα τα επιθέματα (αυτό εξηγεί τον παράγοντα $\log n$) και χρησιμοποιούμε τα queries του μακρύτερου κοινού

προθέματος (longest common prefix) για να αποφασίσουμε γρήγορα αν το μοτίβο χρειάζεται να συγκριθεί με ένα συγκεκριμένο επίθεμα.

Οι απαιτήσεις χωρητικότητας είναι ένα κρίσιμο ζήτημα. Σε κάποιες περιπτώσεις, τα suffix trees και τα suffix arrays είναι μεγαλύτερα από το κείμενο κατά έναν πολλαπλασιαστικό παράγοντα $O(\log_2 |\Sigma| n) = O(\log_2 n)$, ο οποίος είναι σημαντικός όταν το Σ είναι σταθερού μεγέθους, όπως στον κώδικα ASCII ή UNICODE. Για το λόγο αυτό, δημιουργήθηκαν τα συμπιεσμένα suffix arrays και τα συμπιεσμένα suffix trees από τους Roberto Grossi και Jeffrey Scott Vitter το 2005.

Τα συμπιεσμένα suffix trees και τα συμπιεσμένα suffix arrays είναι νέες μορφές ευρετηρίων κειμένου τα οποία βασίζονται σε συμπιεσμένες αναπαραστάσεις των suffix trees και των suffix arrays. Επιτυγχάνουν γρήγορη αναζήτηση σε χρόνο $O(m/\log|\Sigma| n + \log^c|\Sigma| n)$, στην χειρότερη περίπτωση, για οποιαδήποτε σταθερά $0 < c \leq 1$, χρησιμοποιώντας το πολύ $(c^{-1} + O(1)) n \log|\Sigma|$ bits αποθηκευτικού χώρου, σε σύγκριση με τα suffix arrays που καταλαμβάνουν $n \log n$ bits αποθηκευτικού χώρου. Για παράδειγμα, το συμπιεσμένο suffix array για ένα τυπικό αρχείο ASCII μεγέθους 100MB μπορεί να απαιτεί 30-40MB χωρητικότητα ή λιγότερα, ενώ ένα suffix array απαιτεί 500MB χωρητικότητα. Έτσι, τα συμπιεσμένα suffix trees και τα συμπιεσμένα suffix arrays βελτιώνουν το χρόνο και τη χωρητικότητα των προηγούμενων ευρετηρίων κειμένου.

Τα suffix trays και τα suffix trists είναι δύο δομές δεδομένων οι οποίες δημιουργήθηκαν από τους Richard Cole, Tsvi Kopelowitz και Moshe Lewenstein. Τα suffix trays προέρχονται από το συνδυασμό των suffix trees και των suffix arrays, ενώ τα suffix trists προέρχονται από τον συνδυασμό των suffix trees και των suffix lists.

Τα suffix trays έχουν πιο γρήγορο χρόνο αναζήτησης από τα suffix trees και τα suffix arrays, με χρόνο $O(m + \log|\Sigma|)$, σε αντίθεση με τα επόμενα που έχουν χρόνο $O(m \log|\Sigma|)$ και αντίστοιχα $O(m + \log n)$. Τα suffix trists επιτυγχάνουν χρόνο αναζήτησης $O(m + \log|\Sigma|)$, όπως και τα suffix trays.

Από αυτές τις δομές, οι καλύτερες είναι τα συμπιεσμένα suffix arrays, τα συμπιεσμένα suffix trees, τα suffix trays και τα suffix trists, διότι είναι νεότερες, συνδυάζουν τα πλεονεκτήματα των παλαιότερων δομών και ανταποκρίνονται αποτελεσματικότερα στα σύγχρονα δεδομένα απ' ό,τι τα suffix arrays και τα suffix

trees. Επιπλέον, οι νεότερες αυτές δομές είναι ταχύτερες ως προς τον χρόνο αναζήτησης και καταλαμβάνουν λιγότερα bits αποθηκευτικού χώρου.

Κεφάλαιο 4^ο: Πρακτική μελέτη των suffix arrays

Σε αυτό το κεφάλαιο θα γίνει μελέτη της λειτουργίας των suffix arrays. Για την ανάπτυξη του κώδικα χρησιμοποιήθηκε το προγραμματιστικό περιβάλλον NetBeans IDE, σε γλώσσα Java. Παρακάτω παρατίθεται ο κώδικας για την κατασκευή του suffix array και αναζήτηση ενός μοτίβου.

4.1: Κώδικας σε γλώσσα Java

```
1  import java.io.*;
2
3  public class SuffixArray {
4  public static void main(String[] args) {
5      String temp, T = "";
6      try {
7          FileReader fr = new FileReader("test1.txt");
8          BufferedReader br = new BufferedReader(fr);
9
10         while (true) {
11             temp = br.readLine();
12             if (temp == null) break;
13             T += temp;
14         }
15         br.close();
16     }
17     catch (IOException exc) {
18         System.out.println("ERROR... " + exc.toString());
19     }
20
21     //arxikopoiisi pinaka pou tha periexei ta mi taksinomimena
22     epithemata
23     String[] suffixes = new String[T.length()];
24
25     //arxikopoiisi tou suffix array
26     int[] SA = new int[T.length()];
27     for(int i = 0; i < SA.length; i++) {
28         SA[i] = i;
29     }
30
31     //arxikopoiisi tou pinaka lcp (longest common prefix)
32     int[] LCP = new int[T.length()];
33     //orizoume tin proti grammi tou pinaka LCP na einai isi me 0
34     dioti den tin xrisimopoioume
35     LCP[0] = 0;
36
37     //dimiourgia tou pinaka me ta epithemata
38     for (int i = 0; i < suffixes.length; i++) {
39         suffixes[i] = T;
40         //afairoume ton proto xaraktira apo to T
41         T = T.substring(1);
42     }
```

```

43
44      //kaloume tin methodo sort i opoia dimiourgei to suffix array
45 me tin taksinomisi quick sort
46      //upologizoume ton xrono oloklirosis tis methodou me to
47 System.nanoTime()
48      long start = System.nanoTime();
49      myUtils.sort(suffixes, SA);
50      long finish = System.nanoTime();
51      long constructionTime = finish - start;
52
53      //upologismos ton pliroforion lcp
54      myUtils.LCPArrayConstruction (LCP, suffixes, SA);
55
56      String pattern = "bbab";
57      int pos[] = new int[2];
58      pos[0] = -1;
59      pos[1] = -1;
60
61      //anazitisi tou pattern kai upologismos tou xronou anazitisis
62 me to System.nanoTime()
63      start = System.nanoTime();
64      myUtils.search (suffixes, SA, pattern, pos);
65      finish = System.nanoTime();
66      long searchTime = finish - start;
67
68      //emfanisi apotelesmaton
69      if(pos[0] != -1) System.out.println("The first occurrence of
70 pattern " + pattern + " was found at position " + pos[0] + ". \nThe
71 last occurrence was found at position " + pos[1] + ".");
72      else System.out.println("No matches of pattern " + pattern +
73 " were found.");
74      System.out.println("The suffix array was constructed in " +
75 constructionTime + " nanoseconds. \nThe search finished in " +
76 searchTime + " nanoseconds.");
77    }
78  }
79
80  class myUtils {
81      public static void LCPArrayConstruction (int[] LCP, String[]
82 suffixes, int[] SA) {
83          for(int i = 1; i < LCP.length; i++) {
84              int j = 0;
85              int count = 0;
86              while(suffixes[SA[i-1]].charAt(j) ==
87 suffixes[SA[i]].charAt(j))
88              {
89                  count++;
90                  j++;
91                  if (suffixes[SA[i-1]].length() == j ||
92 suffixes[SA[i]].length() == j) break;
93              }
94              LCP[i] = count;
95          }
96      }
97
98      //Begin quick sort
99      private static void swap (String A[], int[] B, int x, int y) {
100          String tempA = A[x];
101          A[x] = A[y];
102          A[y] = tempA;
103

```

```

104         int tempB = B[x];
105         B[x] = B[y];
106         B[y] = tempB;
107     }
108
109     private static int partition(String[] A, int[] B, int first, int
110 last) {
111         String pivot = A[last];
112         int i = (first-1);
113
114         for (int j = first; j < last; j++) {
115             if (A[j].compareTo(pivot) <= 0) {
116                 i++;
117                 swap(A,B,i,j);
118             }
119         }
120         swap(A,B,i+1,last);
121
122         return i+1;
123     }
124
125     private static void Qsort (String[] A, int[] B, int first, int
126 last) {
127         if (first < last) {
128             int pivot_index = partition (A, B, first, last);
129             Qsort (A, B, first, pivot_index-1);
130             Qsort (A, B, pivot_index+1, last);
131         }
132     }
133
134     public static void sort (String[] temp, int[] B) {
135         String[] A = new String[temp.length];
136         System.arraycopy(temp, 0, A, 0, temp.length);
137         Qsort (A, B, 0, A.length-1);
138     }
139     //End quick sort
140
141     //Begin binary search
142     public static void search (String[] array, int[] SA, String key,
143 int[] pos) {
144         int left = 0, right = array.length-1;
145         binarySearch(array, SA, key, left, right, pos);
146     }
147
148     private static void binarySearch (String[] array, int[] SA,
149 String key, int left, int right, int[] pos) {
150         int mid = (left+right)/2;
151         if (left > right) {}
152         else if (array[SA[mid]].startsWith(key)) {
153             pos[0] = mid;
154             pos[1] = mid;
155
156             //euresi tis protis emfanisis tou pattern. Apothikeuetai
157             sto pos[0]
158             int temp = mid;
159             while(temp > left && array[SA[temp-1]].startsWith(key)) {
160                 temp--;
161                 pos[0] = temp;
162             }
163

```

```

164         //euresi tis teleutaias emfanisis tou pattern.
165         Apothikeuetai sto pos[1]
166         temp = mid;
167         while(temp < right && array[SA[temp+1]].startsWith(key)){
168             temp++;
169             pos[1] = temp;
170         }
171     }
172     else if (key.compareTo(array[SA[mid]]) > 0) binarySearch
173 (array, SA, key, mid+1, right, pos);
174     else binarySearch (array, SA, key, left, mid-1, pos);
175 }
176 //End binary search
177 }

```

4.2: Επεξήγηση του κώδικα

Στις γραμμές 6-19, το πρόγραμμα διαβάζει ένα αρχείο κειμένου (στη συγκεκριμένη περίπτωση το αρχείο test1.txt) και αποθηκεύει τα περιεχόμενά του στο String T.

Στις γραμμές 22-35, αρχικοποιούνται οι πίνακες suffixes, SA και LCP. Η πρώτη γραμμή του πίνακα LCP είναι ίση με μηδέν, διότι δεν χρησιμοποιείται.

Ο πίνακας suffixes θα περιέχει όλα τα επιθέματα του String T και δημιουργείται στις γραμμές 38-42.

Ο πίνακας SA αντιστοιχεί το suffix array και θα περιέχει τους ταξινομημένους δείκτες όλων των επιθεμάτων του πίνακα suffixes. Στη γραμμή 49 καλείται η μέθοδος sort της κλάσης myUtils, η οποία κατασκευάζει το suffix array με την quick sort, η οποία είναι μία από τις γρηγορότερες μεθόδους ταξινόμησης. Η μέθοδος του προγράμματος δέχεται ως είσοδο τους πίνακες suffixes και SA. Για να δείξουμε την χρησιμότητα των suffix arrays ως δείκτη ταξινομημένων επιθεμάτων, δημιουργούμε ένα προσωρινό αντίγραφο του πίνακα suffixes (γραμμή 136) μέσα στη μέθοδο sort και τον χρησιμοποιούμε για να ταξινομήσουμε τον πίνακα SA με βάση τον προσωρινό πίνακα, έτσι ώστε ο πίνακας suffixes να παραμείνει αναλλοίωτος. Στη γραμμή 51, η μεταβλητή constructionTime μας δείχνει πόσα nanosecond διαρκεί αυτή η μέθοδος.

Ο πίνακας LCP που είναι τύπου integer, περιέχει τις πληροφορίες για τα μακρύτερα κοινά προθέματα και μπορεί να χρησιμοποιηθεί για να βελτιωθεί ο χρόνος αναζήτησης κάποιου μοτίβου. Στην παρούσα πτυχιακή εργασία θα μελετήσουμε μόνο

τον τρόπο κατασκευής του. Στη γραμμή 54 καλείται η μέθοδος `LCPArrayConstruction` της κλάσης `myUtils`, η οποία κατασκευάζει τον πίνακα `LCP` και υλοποιείται στις γραμμές 81-96. Η μεταβλητή `count` μετράει πόσους ίδιους χαρακτήρες έχει ένα ταξινομημένο επίθεμα με το επόμενο του και καταχωρεί το αποτέλεσμα στον πίνακα `LCP`. Η διαδικασία συνεχίζεται και για όλα τα υπόλοιπα επιθέματα του πίνακα `suffixes`.

Στη γραμμή 56 ορίζουμε το `pattern` προς αναζήτηση και στη γραμμή 57 ορίζουμε τον διαθέσιμο πίνακα `pos` ο οποίος στην πρώτη γραμμή θα περιέχει την πρώτη εμφάνιση του `pattern` στα ταξινομημένα επιθέματα του πίνακα `suffixes` και αντίστοιχα στη δεύτερη γραμμή θα περιέχει την τελευταία εμφάνιση. Αρχικοποιούμε τον πίνακα `pos` με την τιμή -1 και για τις δύο γραμμές του.

Στη γραμμή 64 καλείται η μέθοδος `search` της κλάσης `myUtils`, η οποία βρίσκει την πρώτη και τελευταία εμφάνιση του `pattern`, πραγματοποιώντας δυαδική αναζήτηση, η υλοποίηση της οποίας βρίσκεται στις γραμμές 142-175. Σε περίπτωση που βρεθεί το `pattern`, καταχωρεί τις προαναφερθείσες θέσεις αυτές στον πίνακα `pos`. Η δυαδική αναζήτηση δεν επιστρέφει πάντα την πρώτη εμφάνιση κάποιου `pattern`. Όταν εντοπιστεί η θέση του `pattern`, την αποθηκεύουμε σε μία προσωρινή μεταβλητή `temp` και πραγματοποιούμε αναζήτηση στον ταξινομημένο πίνακα `suffixes` ξεκινώντας από τη θέση `temp` και συνεχίζοντας προς τα αριστερά του πίνακα, μέχρις ότου να βρούμε ένα στοιχείο του πίνακα `suffixes` το οποίο δεν ταιριάζει με το `pattern`. Αντίστοιχα, για να βρούμε την τελευταία εμφάνιση του `pattern` πραγματοποιούμε αναζήτηση προς τα δεξιά του πίνακα `suffixes`. Στη γραμμή 66, η μεταβλητή `searchTime` μας δείχνει πόσο διήρκεσε η μέθοδος της αναζήτησης σε nanoseconds.

Η εμφάνιση των αποτελεσμάτων πραγματοποιείται στις γραμμές 69-76, όπου εκτός από την εύρεση των θέσεων εμφάνισης του `pattern` (εφόσον αυτές υπάρχουν), μελετάμε και τον χρόνο κατασκευής του suffix array (μεταβλητή `constructionTime`) καθώς και τον χρόνο αναζήτησης του `pattern` (μεταβλητή `searchTime`).

4.3: Παραδείγματα εφαρμογής του κώδικα

Σε αυτή την υποενότητα θα μελετήσουμε δύο παραδείγματα εφαρμογής του κώδικα. Στο πρώτο παράδειγμα θα χρησιμοποιηθεί το κείμενο $T = \text{"abbabbabbabbabaaabababbabbbabba\#"}$, από το παράδειγμα της Εικόνας 4, ενώ για το δεύτερο παράδειγμα θα χρησιμοποιηθεί ένα κείμενο 40.000 τυχαίων χαρακτήρων το οποίο έχει δημιουργηθεί με την ιστοσελίδα <https://www.random.org/>.

4.3.1: Παράδειγμα 1

Για να δούμε τα αποτελέσματα για το κείμενο $T = \text{"abbabbabbabbabaaabababbabbbabba\#"}$, πρέπει να το έχουμε αποθηκεύσει σε ένα αρχείο με το όνομα test1.txt. Αν προσθέσουμε επιπλέον κώδικα για την εμφάνιση των πινάκων SA, LCP και suffixes (με δείκτη το SA) με αυτή τη σειρά, θα πάρουμε τα αποτελέσματα που φαίνονται στην Εικόνα 6. Παρατηρούμε ότι η σειρά των ταξινομημένων επιθεμάτων είναι ακριβώς η ίδια με εκείνη του παραδείγματος της Εικόνας 4, με τη μόνη διαφορά ότι σε εκείνο το παράδειγμα, το $\text{"\#"}$ είναι ένα σύμβολο, τέτοιο ώστε $a < \# < b$, ενώ για το παράδειγμα του κώδικα ισχύει $\# < a < b$.

Για την αναζήτηση του μοτίβου "bbab" , παίρνουμε τα ακόλουθα αποτελέσματα.

```
The first occurrence of pattern bbab was found at position 25.  
The last occurrence was found at position 30.  
The suffix array was constructed in 698700 nanoseconds.  
The search finished in 20900 nanoseconds.
```

Αν επαναλάβουμε την εκτέλεση του προγράμματος, θα παρατηρήσουμε ότι ο χρόνος κατασκευής του suffix array και ο χρόνος αναζήτησης είναι λίγο διαφορετικοί από εκείνους της προηγούμενης εκτέλεσης.

```
The suffix array was constructed in 626300 nanoseconds.  
The search finished in 22800 nanoseconds.
```

Μετά από μερικές ακόμα εκτελέσεις του προγράμματος, βρέθηκε ότι ο γρηγορότερος χρόνος αναζήτησης για το συγκεκριμένο μοτίβο ήταν 14700 nanoseconds και ο χρόνος κατασκευής του suffix array ήταν 619000 nanoseconds.

	SA	LCP	suffixes
0	31	0	#
1	30	0	a#
2	14	1	aaabababbabbbabba#
3	15	2	aabababbabbbabba#
4	12	1	abaaabababbabbbabba#
5	16	3	abababbabbbabba#
6	18	4	ababbabbbabba#
7	27	2	abba#
8	9	4	abbabaaabababbabbbabba#
9	6	5	abbabbabaaabababbabbbabba#
10	3	8	abbabbabbabaaabababbabbbabba#
11	0	11	abbabbabbabbabaaabababbabbbabba#
12	20	6	abbabbabba#
13	23	3	abbbabba#
14	29	0	ba#
15	13	2	baaabababbabbbabba#
16	11	2	babaaabababbabbbabba#
17	17	4	bababbabbbabba#
18	26	3	babba#
19	8	5	babbabaaabababbabbbabba#
20	5	6	babbabbabaaabababbabbbabba#
21	2	9	babbabbabbabaaabababbabbbabba#
22	19	7	babbabbabba#
23	22	4	babbabba#
24	28	1	bba#
25	10	3	bbabaaabababbabbbabba#
26	25	4	bbabba#
27	7	6	bbabbabaaabababbabbbabba#
28	4	7	bbabbabbabaaabababbabbbabba#
29	1	10	bbabbabbabbabaaabababbabbbabba#
30	21	5	bbabbabba#
31	24	2	bbbabba#

Εικόνα 6: Παράδειγμα εφαρμογής του κώδικα για το κείμενο T = “abbabbabbabbabaaabababbabbbabba#”.

Στη δεύτερη στήλη εμφανίζεται το περιεχόμενο του πίνακα SA, στην τρίτη εμφανίζεται το περιεχόμενο του πίνακα LCP, ενώ στην τελευταία στήλη εμφανίζονται τα επιθέματα, ταξινομημένα σε λεξικογραφική σειρά.

4.3.2: Παράδειγμα 2

Σε αυτό το παράδειγμα θα εξετάσουμε τα αποτελέσματα του κειμένου που δημιουργήθηκε με τυχαίους χαρακτήρες από την ιστοσελίδα random.org.

Θα πραγματοποιήσουμε αναζήτηση για το pattern “00”. Παρακάτω φαίνονται τα αποτελέσματα της πρώτης εκτέλεσης του προγράμματος.

```
The first occurrence of pattern 00 was found at position 0.
The last occurrence was found at position 10.
The suffix array was constructed in 51265400 nanoseconds.
The search finished in 49300 nanoseconds.
```

Παρατηρούμε ότι ο χρόνος κατασκευής του suffix array καθώς και ο χρόνος αναζήτησης του μοτίβου είναι μεγαλύτεροι από εκείνους του προηγούμενου παραδείγματος, λόγω του μεγαλύτερου χώρου που καταλαμβάνει το suffix array και ο πίνακας των επιθεμάτων.

Μετά από αρκετές εκτελέσεις του προγράμματος, βρέθηκε ότι ο καλύτερος χρόνος αναζήτησης του συγκεκριμένου μοτίβου ήταν 41500 nanoseconds με χρόνο κατασκευής του suffix array σε 41177399 nanoseconds.

4.3.3: Ανάλυση αποτελεσμάτων

Από την εκτέλεση των δύο παραδειγμάτων, παρατηρούμε ότι η αναζήτηση κάποιου μοτίβου ολοκληρώνεται σε αρκετά ικανοποιητικό χρονικό διάστημα, ανεξάρτητα από την χωρητικότητα που καταλαμβάνει το αρχείο κειμένου. Συγκεκριμένα, το κείμενο του παραδείγματος 2 περιέχει 1250 φορές περισσότερους χαρακτήρες από το κείμενο του παραδείγματος 1, αλλά ο χρόνος αναζήτησης του μοτίβου είναι μόνο 3 φορές μεγαλύτερος από εκείνον του παραδείγματος 1.

Ο χώρος, όμως, που απαιτείται για το suffix array είναι αρκετά μεγάλος, αφού καταλαμβάνει τόσες θέσεις στον πίνακα όσοι είναι και οι χαρακτήρες στο κείμενο. Για παράδειγμα, για το κείμενο $T = \text{“abbabbabbabbabaaabababbabbbabba\#”}$ το οποίο αποτελείται από 32 χαρακτήρες, το suffix array του καταλαμβάνει επίσης 32 θέσεις στον πίνακα.

Συμπεράσματα

Στην παρούσα πτυχιακή εργασία μελετήθηκαν αλγόριθμοι αναζήτησης σε συμπίεμένο κείμενο. Ο αλγόριθμος των Udayan K. και Anirudh K. παρέχει υψηλά ποσοστά συμπίεσης και γρήγορης αναζήτησης σε όλο το κείμενο. Αφήνει ανοιχτά πολλά περιθώρια για βελτίωση και περεταίρω έρευνα και υλοποίηση της φάσης 3 του αλγορίθμου. Για τον ίδιο αλγόριθμο θα μπορούσε να γίνει εκτεταμένη μελέτη για να επιτευχθεί η μέγιστη συμπίεση, καθώς και να τροποποιηθεί το σχήμα συμπίεσης για γρηγορότερη αναζήτηση. Μία άλλη πρόκληση είναι να επιτευχθεί η ενσωμάτωση της ευρετηρίασης.

Μία άλλη μέθοδος που μελετήθηκε ήταν η αναζήτηση σε πολλαπλές αναφορές με το ευρετήριο συμπίεσμένης αναζήτησης πολλαπλών αναφορών. Το κύριο πλεονέκτημά του αλγορίθμου είναι η ικανότητά του να εκμεταλλεύεται πολλαπλές αναφορές κατά τη διάρκεια της συμπίεσης, γεγονός που της καθιστά αποδοτικές και σε συλλογές με ανόμοιες συμβολοσειρές. Υπάρχουν περιθώρια για βελτίωση αν γίνει διερεύνηση των μοντέλων κόστους για τους χρόνους απάντησης των queries. Επίσης ενδιαφέρουσα θα ήταν και η ανάλυση της χρήσης της κύριας μνήμης κατά τη δημιουργία του ευρετηρίου.

Στη συνέχεια παρουσιάστηκαν κάποιες δομές δεδομένων που στοχεύουν στην ταχεία αναζήτηση μοτίβων μέσα σε ένα κείμενο. Η παλαιότερη από αυτές είναι τα suffix trees. Μερικά χρόνια αργότερα, εφευρέθηκαν τα suffix arrays τα οποία βελτιώνουν τον χρόνο αναζήτησης των μοτίβων. Οι απαιτήσεις χωρητικότητας των δύο αυτών δομών, όμως, παρέμενε ένα άλυτο πρόβλημα, μέχρις ότου εφευρέθηκαν καινούργιες δομές δεδομένων οι οποίες βασίζονταν στην κεντρική ιδέα των suffix trees και των suffix arrays και χρησιμοποιώντας τα πλεονεκτήματά τους βελτίωσαν ακόμη περισσότερο τον χρόνο κατασκευής τους, τις απαιτήσεις χωρητικότητας αλλά και το χρόνο αναζήτησης. Αυτές οι δομές είναι τα συμπίεσμένα suffix trees, τα συμπίεσμένα suffix arrays, τα suffix trays και τα suffix trists. Οι δημιουργοί των δομών αυτών αφήνουν ανοιχτά τα ενδεχόμενα για περαιτέρω έρευνα και βελτίωση της απόδοσής τους σε θέματα χωρητικότητας και χρόνων.

Τέλος, πραγματοποιήθηκε μία πρακτική μελέτη της λειτουργίας και της απόδοσης των suffix arrays με πρόγραμμα γραμμένο σε γλώσσα Java, όπου μέσω δύο

παραδειγμάτων εκτέλεσης του κώδικα, διαπιστώθηκε το πόσο ταχεία είναι η αναζήτηση μοτίβου σε κείμενο με συγκεκριμένο πλήθος χαρακτήρων. Τα αποτελέσματα ήταν αρκετά ικανοποιητικά, όμως, όπως προαναφέρθηκε, το κύριο μειονέκτημα των suffix arrays είναι οι απαιτήσεις χωρητικότητας.

Βιβλιογραφία

- [1] Khurana, Udayan & Koul, Anirudh. (2005). *Text Compression and Superfast Searching*. Thapar Institute Of Engineering and Technology, Patiala, Punjab, India
- [2] Takeda M., Shinohara A. (2016). Pattern Matching on Compressed Text (pp. 17-29), In: Kao MY. (eds) *Encyclopedia of Algorithms*. Springer, New York, NY
- [3] Wandelt S. and Leser, Ul. (2015). MRCSI: Compressing and Searching String Collections with Multiple References. *Proceedings of the VLDB Endowment*, 8(5).
- [4] Manber U, Myers G. (1993). Suffix arrays: a new method for on-line string searches. *SIAM J Comput.* 22(5):935–48.
- [5] Grossi, R. & Vitter, J.Sc. (2005). Compressed suffix arrays and suffix trees with application to text indexing and string matching. *Siam J. Comput. Society for Industrial and Applied Mathematics*, 35(2), 378–407
- [6] Sadakane, K. (2006). Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41, 589-607.
- [7] Cole, Richard & Kopelowitz, Tsvi & Lewenstein, Moshe. (2006). Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*. 72, 358-369.
- [8] Weiner, P. (1973). *Linear Pattern Matching Algorithms*. SWAT.