Ο Μετασχηματισμός Burrows-Wheeler και προτάσεις βελτιστοποίησής του

Ονοματεπώνυμο: Μοσχόπουλος Νικόλαος
ΑΜ: 1054315
Έτος φοίτησης: 5ο

Εργασία στα πλαίσια του μαθήματος:
Ανάκτηση Πληροφορίας

Περιεχόμενα

Εισαγωγή

Φαίνεται ότι δεν υπάρχει όριο στην ποσότητα των δεδομένων που βρίσκονται αποθηκευμένα στους υπολογιστές ή σε δεδομένα που χρειάζεται να αποσταλούν. Παρόλο που η τεχνολογία παρέχει μεγαλύτερους δίσκους αποθήκευσης και ταχύτερα δίκτυα επικοινωνίας, η ανάγκη των ταχύτερων και πιο αποδοτικών αλγορίθμων συμπίεσης δεδομένων είναι πάντα στην πρώτη γραμμή της τεχνολογικής έρευνας. Η πρόοδος όσον αφορά τη συμπίεση των δεδομένων αποτελείται συνήθως από μια μακρά σειρά από μικρές βελτιώσεις και μικροσυντονισμό των υπαρχόντων αλγορίθμων. Ωστόσο κάποιες φορές η έρευνα βιώνει γιγαντιαία άλματα, όπως συνέβη στην περίπτωση της εισαγωγής του μετασχηματισμού Burrows-Wheeler (Burrows-Wheeler Transform – BWT) στον τομέα της συμπίεσης χωρίς απώλειες δεδομένων (μη απωλεστική συμπίεση).

Ο BWT, που επινοήθηκε από τον Michael Burrows και τον David Wheeler το 1994, είναι ένας αλγόριθμος μετασχηματισμού δεδομένων που αναδιαρθρώνει τα δεδομένα με τέτοιο τρόπο, ώστε το μετασχηματισμένο μήνυμα να είναι πιο συμπιέσιμο. Τεχνικά, είναι μια λεξικογραφική αναστρέψιμη παραλλαγή των χαρακτήρων ενός string. Η πιο σημαντική εφαρμογή του BWT βρίσκεται στις βιολογικές επιστήμες όπου τα γονιδιώματα (μακροσκελείς συμβολοσειρές γραμμένες σε A, C, T, G αλφάβητα) έχουν πολλές επαναλήψεις. Η ιδέα του BWT είναι να δημιουργήσει έναν πίνακα του οποίου οι σειρές είναι όλες κυκλικές μετατοπίσεις του string εισόδου με τη σειρά του λεξικού και να επιστρέψει την τελευταία στήλη του πίνακα που τείνει να έχει μακρά σειρά πανομοιότυπων χαρακτήρων. Το πλεονέκτημά του είναι ότι μόλις οι χαρακτήρες ομαδοποιηθούν, έχουν ουσιαστικά μια σειρά, η οποία μπορεί να κάνει τη συμβολοσειρά πιο συμπιέσιμη για άλλους αλγορίθμους όπως η κωδικοποίηση Huffman. Το αξιοσημείωτο πάντως για τον BWT είναι ότι αυτός ο μετασχηματισμός είναι και αντιστρέψιμος.

Στην επιστήμη των υπολογιστών, ένα FM-index είναι ένα συμπιεσμένο, πλήρους κειμένου, substring ευρετήριο που βασίζεται στον BWT με κάποιες ομοιότητες με το suffix array. Οι δημιουργοί του Paolo Ferragina και Giovanni Manzini το περιγράφουν ως μια δομή δεδομένων που επιτρέπει τη συμπίεση του κειμένου εισόδου, ενώ ταυτόχρονα επιτρέπει και γρήγορα substring ερωτήματα. Μπορεί να χρησιμοποιηθεί για τον αποτελεσματικό εντοπισμό του αριθμού εμφανίσεων ενός προτύπου στο συμπιεσμένο κείμενο, καθώς και για τον εντοπισμό της θέσης κάθε εμφάνισης.

Ενότητα 1: Ο μετασχηματισμός Burrows-Wheeler

Μερικοί ευρέως χρησιμοποιούμενοι αλγόριθμοι συμπίεσης δεδομένων βασίζονται στους διαδοχικούς συμπιεστές δεδομένων Lempel και Ziv. Στατιστικά μοντελοποιημένες τεχνικές μπορεί να παράγουν καλύτερη συμπίεση, αλλά είναι αρκετά πιο αργές. Σε αυτή την ενότητα, παρουσιάζεται μια τεχνική που πετυχαίνει συμπίεση εντός ενός ποσοστού ή περίπου αυτού που επιτεύχθηκε με τεχνικές στατιστικής μοντελοποίησης, αλλά σε συγκρίσιμες ταχύτητες με εκείνους τους αλγορίθμους που βασίζονται στους Lempel και Ziv. Ο αλγόριθμος δεν επεξεργάζεται την είσοδό του διαδοχικά, αλλά αντίθετα επεξεργάζεται ένα μπλοκ κειμένου σαν μια ενότητα. Η ιδέα είναι να εφαρμοστεί ένας αναστρέψιμος μετασχηματισμός σε ένα μπλοκ κειμένου για να σχηματιστεί ένα νέο μπλοκ που περιέχει τους ίδιους χαρακτήρες, αλλά είναι ευκολότερο για συμπίεση με απλούς αλγορίθμους συμπίεσης. Ο μετασχηματισμός τείνει να ομαδοποιεί χαρακτήρες μαζί, έτσι ώστε η πιθανότητα εύρεσης ενός χαρακτήρα κοντά σε έναν άλλο με παρουσία του ίδιου χαρακτήρα να αυξάνεται σημαντικά. Κείμενο αυτού του είδους μπορεί να συμπιεστεί εύκολα με γρήγορους τοπικά προσαρμοστικούς αλγορίθμους, όπως η move-to-front κωδικοποίηση σε συνδυασμό με κωδικοποίηση Huffman ή αριθμητική κωδικοποίηση.

Πρώτα, θα περιγραφούν δυο υποαλγόριθμοι. Ο αλγόριθμος C εκτελεί τον αναστρέψιμο μετασχηματισμό που εφαρμόζεται σε ένα μπλοκ κειμένου πριν συμπιεστεί και ο αλγόριθμος D εκτελεί την αντίστροφη διαδικασία. Τα string αντιμετωπίζονται σαν διανύσματα των οποίων τα στοιχεία είναι χαρακτήρες.

Αλγόριθμος C: Μετασχηματισμός συμπίεσης

Ο αλγόριθμος παίρνει ως είσοδο ένα string S από N χαρακτήρες S[0],…,S[N-1] επιλεγμένους από ένα ταξινομημένο αλφάβητο Χ χαρακτήρων. Τα βήματά του είναι τα εξής:

1. [Ταξινόμηση περιστροφών] Σχηματισμός ενός σχετικού N x N πίνακα M του οποίου τα στοιχεία είναι χαρακτήρες και του οποίου οι σειρές είναι οι περιστροφές (κυκλικές μετατοπίσεις) του S, ταξινομημένες σε λεξικογραφική σειρά. Τουλάχιστον μια από τις σειρές του M περιέχει το αρχικό string S. Έστω I ο δείκτης της πρώτης τέτοιας σειράς, μετρώντας από το μηδέν.
2. [Εύρεση τελευταίων χαρακτήρων περιστροφών] Έστω το string L ότι είναι η τελευταία στήλη του M, με χαρακτήρες L[0],…,L[N-1] (ίσο με το M[0, N-1],…,M[N-1, N-1]. Η έξοδος του μετασχηματισμού είναι το ζεύγος (L, I).

Αλγόριθμος D: Μετασχηματισμός αποσυμπίεσης

Ο αλγόριθμος D χρησιμοποιεί την έξοδο (L, I) του αλγορίθμου C για να ανακατασκευάσει την είσοδό του, το string S μήκους N. Τα βήματά του είναι τα εξής:

1. [Εύρεση πρώτων χαρακτήρων των περιστροφών] Αυτό το βήμα υπολογίζει την πρώτη στήλη F του πίνακα M του αλγορίθμου C. Αυτό γίνεται με την ταξινόμηση των χαρακτήρων του L για να δημιουργηθεί το F. Τα L και F είναι και τα δυο παραλλαγές του S, και επομένως το ένα του άλλου. Επίσης, οι χαρακτήρες στο F ταξινομούνται.
2. [Κατασκευή λίστας προηγούμενων χαρακτήρων] Εξετάζονται οι σειρές του πίνακα M που ξεκινούν με κάποιο δοσμένο χαρακτήρα ch. Ορίζεται ο πίνακας M’ που σχηματίζεται περιστρέφοντας κάθε σειρά του M ένα χαρακτήρα δεξιά, ώστε για κάθε i = 0,…,N-1 και κάθε j = 0,…,N-1, να προκύψει M’[i,j] = M[i, (j-1) mod N]. Όπως στο M, κάθε σειρά του M’ είναι μια περιστροφή του S, και για κάθε σειρά του M υπάρχει μια αντίστοιχη γραμμή στο M’. Το M’ κατασκευάστηκε από το M έτσι ώστε οι γραμμές του M’ να ταξινομούνται λεξικογραφικά ξεκινώντας από το δεύτερο τους χαρακτήρα. Συμπερασματικά, για οποιονδήποτε δεδομένο χαρακτήρα ch, οι γραμμές στο M που ξεκινούν με ch εμφανίζονται στην ίδια σειρά ως γραμμές στο M’ που ξεκινούν με το ch. Χρησιμοποιώντας τα F και L (οι πρώτες στήλες του M και M’ αντίστοιχα) υπολογίζεται ένα διάνυσμα T που δείχνει την αντιστοιχία μεταξύ των γραμμών των δυο πινάκων, με

την έννοια ότι για κάθε j = 0,…,N-1 η γραμμή j του M' αντιστοιχεί στη γραμμή T[j] του M. Αν το L[j] είναι η k-οστή παρουσία του ch στο L, τότε T[j] = i όπου F[i] είναι η k-οστή παρουσία του ch στο F. Επίσης F[T[j]] = L[j].

3. [σχηματισμός εξόδου S] Τώρα, για κάθε i = 0,…,N-1, οι χαρακτήρες L[i] και F[i] είναι οι τελευταίοι και οι πρώτοι χαρακτήρες της γραμμής i του M. Ο χαρακτήρας L[i] προηγείται κυκλικά του χαρακτήρα F[i] στο S. Αντικαθιστώντας i = T[j], στο F[T[j]] = L[j] , προκύπτει ότι το L[T[j]] προηγείται κυκλικά του L[j] στο S. Ο τελευταίος χαρακτήρας του S είναι L[I]. Χρησιμοποιώντας το διάνυσμα T δίνεται στους προηγούμενους του κάθε χαρακτήρα, για κάθε i = 0,…,N-1: S[N-1-i] = L[T^i[I]], όπου T^0[x] = x, και T^{i+1}[x] = T[T^i[x]]. Αυτό αποδίδει S, την αρχική είσοδο στον συμπιεστή.

Η ακολουθία T^i[I] για i = 0,…,N-1 δεν είναι απαραίτητα μια παραλλαγή των αριθμών 0,…,N-1. Οι επαναλήψεις στο S δημιουργούνται με επίσκεψη στα ίδια στοιχεία του T επαναλαμβανόμενα.

Ο αλγόριθμος C ταξινομεί τις περιστροφές ενός string εισόδου S και δημιουργεί το string L που αποτελείται από τον τελευταίο χαρακτήρα κάθε περιστροφής, κάτι το οποίο μπορεί να οδηγήσει σε αποτελεσματική συμπίεση. Η πιθανότητα ο δοθείς χαρακτήρας ch να εμφανιστεί σε ένα δεδομένο σημείο στο L είναι αρκετά υψηλή αν το ch εμφανίζεται κοντά σε αυτό το σημείο στο L, αλλιώς είναι χαμηλή. Αυτή η ιδιότητα είναι ακριβώς εκείνη που απαιτείται για αποτελεσματική συμπίεση από έναν move-to-front κωδικοποιητή, ο οποίος κωδικοποιεί μια εμφάνιση του χαρακτήρα ch από το πλήθος των διακριτών χαρακτήρων που έχει δει από την επόμενη προηγούμενη εμφάνιση του ch. Όταν εφαρμόζεται στο string L, η έξοδος ενός move-to-front κωδικοποιητή θα κυριαρχείται από χαμηλούς αριθμούς, που μπορούν να κωδικοποιηθούν αποτελεσματικά με έναν Huffman ή αριθμητικό κωδικοποιητή. Παρακάτω παρατίθεται ένας πιθανός τρόπος κωδικοποίησης της εξόδου του αλγορίθμου C, και της αντίστοιχης αντίστροφης διαδικασίας. Ένας πλήρης συμπιεσμένος αλγόριθμος που δημιουργείται συνδυάζοντας αυτές τις διαδικασίες κωδικοποίησης και αποκωδικοποίησης με τους Αλγορίθμους C και D.

<u>Αλγόριθμος M: Move-to-front κωδικοποίηση</u>

Αυτός ο αλγόριθμος κωδικοποιεί την έξοδο (L, I) του αλγορίθμου C, όπου το L είναι ένα string μήκους N και το I είναι ένας δείκτης. Κωδικοποιεί το L χρησιμοποιώντας έναν move-to-front αλγόριθμο και ένα Huffman ή αριθμητικό κωδικοποιητή. Τα βήματά του είναι τα εξής:

1. [move-to-front κωδικοποίηση] Αυτό το βήμα κωδικοποιεί κάθε έναν από τους χαρακτήρες στο L εφαρμόζοντας την τεχνική move-to-front στους μεμονωμένους χαρακτήρες. Ορίζεται ένα διάνυσμα ακεραίων R[0],…,R[N-1], που είναι οι κώδικες για τους χαρακτήρες L[0],…,L[N-1].

   Αρχικοποιείται μια λίστα Y από χαρακτήρες που περιέχουν κάθε χαρακτήρα στο αλφάβητο X ακριβώς μια φορά. Για κάθε i = 0,…,N-1 με τη σειρά, θέτεται R[i] στον αριθμό των χαρακτήρων του προηγούμενου χαρακτήρα L[i] στη λίστα Y και μετά μετακινείται ο χαρακτήρας L[i] μπροστά από το Y.
2. [Κωδικοποίηση] Εφαρμόζεται Huffman ή αριθμητική κωδικοποίηση στα στοιχεία του R, αντιμετωπίζοντας το κάθε στοιχείο ως ξεχωριστό για κωδικοποίηση. Οποιαδήποτε τεχνική κωδικοποίησης μπορεί να εφαρμόζεται εφ' όσον ο αποσυμπιεστής μπορεί να εκτελέσει την αντίστροφη διαδικασία. Η έξοδος αυτής της διαδικασίας κωδικοποίησης καλείται OUT. Η έξοδος του Αλγορίθμου C είναι το ζευγάρι (OUT, I) όπου I είναι η τιμή που υπολογίστηκε στο βήμα 1 (του αλγορίθμου C).

<u>Αλγόριθμος W: Move-to-front αποκωδικοποίηση</u>

Αυτός ο αλγόριθμος είναι το αντίστροφο του Αλγορίθμου M. Υπολογίζει το ζευγάρι (L, I) από το ζευγάρι (OUT, I). Έστω ότι η αρχική τιμή της λίστας Y που χρησιμοποιήθηκε στο παραπάνω

βήμα 1 είναι διαθέσιμη στον αποσυμπιεστή και ότι το σχήμα κωδικοποίησης που χρησιμοποιήθηκε στο παραπάνω βήμα 2 έχει αντίστροφη λειτουργία.

1. [Αποκωδικοποίηση] Αποκωδικοποίηση του κωδικοποιημένου stream OUT χρησιμοποιώντας το αντίστροφο του σχήματος κωδικοποίησης που χρησιμοποιήθηκε στο βήμα 2 (του αλγορίθμου M). Το αποτέλεσμα είναι ένα διάνυσμα R των N ακεραίων.
2. [Αντίστροφη move-to-front κωδικοποίηση] Ο στόχος αυτού του βήματος είναι να υπολογίσει ένα string L από N χαρακτήρες, δοθέντων των κωδικών move-to-front R[0],…,R[N-1].
   Αρχικοποιείται μια λίστα Y χαρακτήρων που περιέχει χαρακτήρες του αλφαβήτου X στην ίδια σειρά όπως στο βήμα 1 (του αλγορίθμου M). Για κάθε i = 0,…,N-1 με τη σειρά, θέτεται L[i] να είναι ο χαρακτήρας στη θέση R[i] στην λίστα Y (μετρώντας από το 0) και μετά μετακινείται αυτός ο χαρακτήρας στο μπροστινό μέρος του Y. Το string L που προκύπτει είναι η τελευταία στήλη του πίνακα M του αλγορίθμου C. Η έξοδος αυτού του αλγορίθμου είναι το ζευγάρι (L, I) που είναι η είσοδος του αλγορίθμου D.

Ο πιο σημαντικός παράγοντας στην ταχύτητα συμπίεσης είναι ο χρόνος που απαιτείται για την ταξινόμηση των περιστροφών του μπλοκ εισόδου. Ένας γρήγορος τρόπος για την εφαρμογή του αλγορίθμου C είναι να μειωθεί το πρόβλημα της ταξινόμησης των περιστροφών στο πρόβλημα της ταξινόμησης των suffixes ενός σχετικού string. Το γρηγορότερο σχέδιο που έχει δοκιμαστεί γι' αυτό, είναι ο Αλγόριθμος Q, που χρησιμοποιεί μια παραλλαγή του quicksort για να δημιουργήσει την ταξινομημένη λίστα των suffixes. Ο αλγόριθμος χρησιμοποιεί μόνο δυο λέξεις ανά χαρακτήρα εισόδου.

<u>Αλγόριθμος Q: Γρήγορο quicksorting στα suffixes</u>

Ο αλγόριθμος ταξινομεί τα suffixes του string S, το οποίο περιέχει N χαρακτήρες S[0,…,N-1]. Λειτουργεί εφαρμόζοντας μια τροποποιημένη εκδοχή του quicksort στα suffixes του S.

1. [Σχηματισμός εκτεταμένου string] Έστω k ο αριθμός των χαρακτήρων που ταιριάζουν σε μια λέξη μηχανής. Σχηματισμός του string S' από το S προσαρτώντας k επιπλέον EOF χαρακτήρες στο S, όπου το EOF δεν εμφανίζεται στο S.
2. [Σχηματισμός πίνακα λέξεων] Αρχικοποίηση ενός πίνακα W από N λέξεις W[0,…,N-1], έτσι ώστε το W[i] να περιέχει τους χαρακτήρες S'[i,…,i+k-1] διατεταγμένους έτσι ώστε οι συγκρίσεις ακεραίων στις λέξεις να συμφωνούν με τις λεξικογραφικές συγκρίσεις στους k-χαρακτήρες strings.
3. [Σχηματισμός πίνακα suffixes] Σε αυτό το βήμα αρχικοποιείται ένας πίνακας V των N ακεραίων, έτσι ώστε για κάθε i=0,…,N-1: V[i]=i. Αν ένα στοιχείο του V περιέχει j, αναπαριστά το suffix του S' του οποίου ο πρώτος χαρακτήρας είναι S'[j]. Όταν ο αλγόριθμος ολοκληρωθεί, το suffix V[i] θα είναι το i-οστό suffix σε λεξικογραφική σειρά.
4. [radix sort] Ταξινόμηση των στοιχείων του V, χρησιμοποιώντας τους δυο πρώτους χαρακτήρες κάθε suffix ως κλειδί ταξινόμησης. Αυτό μπορεί να γίνει αποτελεσματικά χρησιμοποιώντας radix sort.
5. [Επανάληψη σε κάθε χαρακτήρα στο αλφάβητο] Για κάθε χαρακτήρα ch στο αλφάβητο, εκτέλεση των βημάτων 6, 7. Μόλις αυτή η επανάληψη ολοκληρωθεί, το V θα αναπαριστά τα ταξινομημένα suffixes του S και ο αλγόριθμος θα τερματίσει.
6. [quicksort suffixes που ξεκινούν με ch] Για κάθε χαρακτήρα ch' στο αλφάβητο: Εφαρμόζεται quicksort στα στοιχεία του V που ξεκινούν με ch ακολουθούμενο από ch'. Στην εκτέλεση του quicksort, συγκρίνονται τα στοιχεία του V συγκρίνοντας τα suffixes που αναπαριστούν ευρετηριάζοντας στον πίνακα W. Σε κάθε επαναληπτικό βήμα, καταγράφεται ο αριθμός των χαρακτήρων που έχουν συγκριθεί ισοδύναμα στο σύνολο, έτσι ώστε να μη χρειάζεται να συγκριθούν ξανά. Συνεπώς, όλα τα suffixes που ξεκινούν με ch θα έχουν ταξινομηθεί στις τελικές τους θέσεις στο V.

7. [Ενημέρωση κλειδιών ταξινόμησης] Για κάθε στοιχείο V[i] που αντιστοιχεί σε ένα suffix που ξεκινάει με ch (αυτό είναι, για όποια S[V[i]] = ch), θέτεται W[V[i]] σε μια τιμή με ch στα υψηλής τάξης bits του (όπως προηγουμένως) και με i στα χαμηλής τάξης bits του (το οποίο βήμα 2 θέτει στους k-1 χαρακτήρες που ακολουθούν το αρχικό ch). Η νέα τιμή του W[V[i]] ταξινομείται στην ίδια θέση όπως η παλιά τιμή, αλλά έχει την ιδιότητα ότι είναι ξεχωριστή από όλες τις άλλες τιμές στο W. Αυτό επιταχύνει επόμενες λειτουργίες ταξινόμησης, αφού οι συγκρίσεις με τα νέα στοιχεία δε μπορούν να συγκριθούν ισοδύναμα.

Ο παραπάνω βασικός αλγόριθμος μπορεί να τελειοποιηθεί με διάφορους τρόπους. Μια προφανής βελτίωση είναι να επιλεγεί ο χαρακτήρας ch στο βήμα 5 ξεκινώντας με τον λιγότερο κοινό χαρακτήρα στο S και προχωρώντας στον πιο κοινό. Επίσης, ο αλγόριθμος έχει χαμηλή απόδοση όταν το S περιέχει πολλά επαναλαμβανόμενα substrings. Αυτό το πρόβλημα αντιμετωπίζεται με δυο μηχανισμούς. Ο πρώτος μηχανισμός χειρίζεται strings ενός απλού επαναλαμβανόμενου χαρακτήρα αντικαθιστώντας το βήμα 6 με τα ακόλουθα βήματα:

1. [quicksort suffixes που ξεκινούν με ch, ch' όπου ch ≠ ch']
2. [ταξινόμηση μικρών suffixes που ξεκινούν με ch, ch] Αυτό το βήμα ταξινομεί τα αρχικά των suffixes που θα ταξινομούνταν πριν από ένα άπειρο string ch χαρακτήρων.
3. [ταξινόμηση μεγάλων suffixes που ξεκινούν με ch, ch] Αυτό το βήμα ταξινομεί τα αρχικά των suffixes που θα ταξινομούνταν μετά από ένα άπειρο string ch χαρακτήρων.

Ο δεύτερος μηχανισμός χειρίζεται μεγάλα strings ενός επαναλαμβανόμενου substring περισσότερων του ενός χαρακτήρα χρησιμοποιώντας και επιπλέον στα βήματά του μια συγκεκριμένη τεχνική διπλασιασμού. Γενικά, ο συνδυασμός του radix sort, μιας προσεκτικής εφαρμογής του quicksort και των μηχανισμών για την αντιμετώπιση επαναλαμβανόμενων strings παράγουν έναν αλγόριθμο ταξινόμησης που είναι εξαιρετικά γρήγορος στις περισσότερες εισόδους και είναι σχεδόν απίθανο να αποδώσει πολύ άσχημα σε πραγματικά δεδομένα εισόδου. Μια ακόμη παραλλαγή του quicksort προκύπτει εφαρμόζοντας τα βήματα A-C σε όλα τα αλληλεπικαλυπτόμενα επαναλαμβανόμενα strings και αποθηκεύοντας προσωρινά τις προηγούμενες υπολογισμένες σειρές ταξινόμησης σε έναν κατακερματισμένο πίνακα. Έτσι δημιουργείται ένας αλγόριθμος που ταξινομεί τα suffixes ενός string σε περίπου ίδιο χρόνο με τον τροποποιημένο αλγόριθμο Q, αλλά χρησιμοποιώντας μόνο 6 bytes χώρου για κάθε χαρακτήρα εισόδου. Αυτή η προσέγγιση έχει χαμηλή απόδοση στη χειρότερη περίπτωση, αλλά οι κακές περιπτώσεις φαίνεται να είναι σπάνιες στην πράξη. Τα βήματα 1 και 2 (του αλγορίθμου D) μπορούν να επιτευχθούν αποτελεσματικά με μόνο δυο περάσματα στα δεδομένα και ένα πέρασμα στο αλφάβητο. Στο πρώτο πέρασμα, κατασκευάζονται δυο πίνακες: C[αλφάβητο] και P[0,…,N-1]. C[ch] είναι ο συνολικός αριθμός των εμφανίσεων στο L των χαρακτήρων που προηγούνται του χαρακτήρα ch στο αλφάβητο. P[i] είναι ο αριθμός των εμφανίσεων του χαρακτήρα L[i] στο prefix L[0,…,i-1] του L. Δεδομένων των πινάκων L, C, P, ο πίνακας T που ορίζεται στο βήμα 2 (του αλγορίθμου D) δίνεται από: T[i] = P[i] + C[L[i]], για κάθε i = 0,…,N-1. Στο δεύτερο πέρασμα στα δεδομένα, χρησιμοποιείται η αρχική θέση I για να συμπληρωθεί ο Αλγόριθμος D και αυτός να δημιουργήσει το S.

Ο αλγόριθμος μπορεί να τροποποιηθεί για να χρησιμοποιήσει ένα διαφορετικό σχεδιάγραμμα κωδικοποίησης αντί του move-to-front στο βήμα 1 (του αλγορίθμου M). Η συμπίεση μπορεί να βελτιωθεί ελαφρώς αν το move-to-front αντικατασταθεί από ένα σχεδιάγραμμα στο οποίο οι κώδικες που δεν έχουν βρεθεί για ένα μεγάλο χρονικό διάστημα μετακινούνται μόνο εν μέρει προς τα εμπρός. Σχετικά με το βήμα 2 (του αλγορίθμου M), αν και οι απλοί Huffman και αριθμητικοί κωδικοποιητές τα πηγαίνουν καλά, ένα περισσότερο σύνθετο σχεδιάγραμμα κωδικοποίησης θα μπορούσε να τα καταφέρει ελαφρώς καλύτερα.

Σχετικά με τα αποτελέσματα των πειραμάτων του αλγορίθμου, από αυτά προκύπτει ότι ο αλγόριθμος τα καταφέρνει και σε εισόδους χωρίς κείμενο και σε εισόδους κειμένου. Οι

μετρήσεις χρόνου στη CPU έγιναν σε ένα DECstation 5000/200, που έχει ένα MIPS R3000 ρολόι επεξεργαστή στα 25MHz με κρυφή μνήμη 64 Kbytes. Η συμπίεση βελτιώνεται με αυξανόμενο μέγεθος μπλοκ, ακόμα και όταν το μέγεθος μπλοκ είναι αρκετά μεγάλο. Ωστόσο, η αύξηση του μεγέθους μπλοκ πάνω από μερικές δεκάδες εκατομμύρια χαρακτήρες έχει μόνο μια μικρή επίδραση. Εάν το μέγεθος μπλοκ αυξηθεί υπερβολικά, μάλλον ο αλγόριθμος δεν θα προσεγγίσει τη θεωρητική βέλτιστη συμπίεση. Ίσως ένα καλύτερο σχήμα κωδικοποίησης να πετύχαινε βέλτιστη συμπίεση ασυμπτωτικά.

Ενότητα 2: Υπολογισμός BWT χωρίς την πλήρη κατασκευή του suffix array

Από όλα τα παραπάνω γίνεται σαφές ότι ο BWT είναι ένας μετασχηματισμός κειμένου που κατέχει κεντρικό ρόλο σε μερικές από τις καλύτερες μεθόδους συμπίεσης δεδομένων. Ωστόσο ένα μειονέκτημά του είναι ότι πρέπει να αποθηκεύει το suffix array, που μπορεί να είναι πολύ μεγαλύτερο από το κείμενο ή τον BWT. Η απαίτηση μεγάλου χώρου του suffix array είναι ένα πρόβλημα επειδή περιορίζει σημαντικά το μέγεθος των κειμένων που μπορεί κανείς να επεξεργαστεί σε έναν υπολογιστή. Σε αυτή την ενότητα στόχος είναι η απαλλαγή από την ανάγκη αποθήκευσης του πλήρους suffix array. Η ιδέα είναι ότι εάν μπορεί να κατασκευαστεί ένα μικρό κομμάτι ή μπλοκ suffix array τη φορά (π.χ. SA[i]), μπορεί να υπολογιστεί το αντίστοιχο μπλοκ του BWT (BWT[i]) χωρίς να χρειάζεται όλο το SA μπλοκ.

Έστω ότι το κείμενο $T[0,n)=t_0t_1…t_{n-1}$ είναι ένα string μήκους n σε ένα γενικό αλφάβητο. Για i ε [0,n], έστω ότι το S_i δηλώνει το suffix $T[i,n) = t_it_{i+1}…t_{n-1}$. Αυτό επεκτείνεται επίσης στα σύνολα: για $C \subseteq [0,n]$, $S_C = \{S_i \mid i \varepsilon C\}$. Ο BWT του T είναι το string BWT[0,n]:

$$\mathrm{BWT}[i] = \begin{cases} T[\mathrm{SA}[i]-1] & \text{if } \mathrm{SA}[i] \neq 0 \\ \$ & \text{if } \mathrm{SA}[i] = 0. \end{cases} \quad (1)$$

Ο αλγόριθμος για την επίλυση του παραπάνω προβλήματος είναι ο εξής:

1) Επιλογή ενός τυχαίου δείγματος C μεγέθους r-1 από το [0,n).
2) Ταξινόμηση του συνόλου S_C των χωριστών suffixes. Έστω ότι $j_1,j_2,…,j_{r-1}$ είναι τα στοιχεία του C ταξινομημένα έτσι ώστε $S_{j1} < S_{j2} < … < S_{jr-1}$. Για ευκολία, έστω $S_{j0} = S_n$ το άδειο suffix, δηλαδή το μικρότερο από τα suffixes και έστω S_{jr} ένα string που είναι μεγαλύτερο από κάθε suffix.
3) Για κάθε k ε [0,r):
   a) Για κάθε j ε [0,n], εξετάζεται εάν $S_{jk} <= S_j < S_{jk+1}$ είναι αληθές ή όχι. Αν ναι, αποθηκεύεται το j στο B_k.
   b) Υπολογίζεται το $SA[i_k,i_{k+1})$ ταξινομώντας τα suffixes S_{Bk}.
   c) Υπολογίζεται ο $BWT[i_k,i_{k+1})$ από το $SA[i_k,i_{k+1})$ χρησιμοποιώντας το (1).

Η ανταλλαγή χώρου-χρόνου του αλγορίθμου ελέγχεται από δυο παραμέτρους, υ και b_{max}.

Ορισμός: Έστω M ένα σύνολο από m strings. Το διακριτό prefix ενός string S στο M είναι το μικρότερο prefix του S που δεν είναι prefix κάποιου άλλου string στο M. Έστω ότι το $DP_M(S)$ δηλώνει το μήκος του διακριτού prefix και έστω ότι το DP(M) δηλώνει το άθροισμα των μηκών στο M, δηλαδή, $DP(M) = \Sigma_{S \varepsilon M} DP_M(S)$. Τα υ-περιορισμένα διακριτά μέτρα prefix ορίζονται ως $DP_M^{\upsilon}(S) = \min(\upsilon, DP_M(S))$ και $DP^{\upsilon}(M) = \Sigma_{S \varepsilon M} DP_M^{\upsilon}(S)$.

Χρησιμοποιώντας τον multikey quicksort αλγόριθμο, ένα σύνολο M από m strings μπορεί να ταξινομηθεί σε O(m logm + DP(M)) χρόνο, απαιτώντας O(logm) παραπάνω χώρο. Ο συνολικός χρόνος πολυπλοκότητας των ταξινομημένων βημάτων (Βήματα 2 και 3b) είναι $O(n \log n + DP(S_{[0,n]}))$. Η πολυπλοκότητα χώρου (χωρίς το κείμενο και χωρίς κάποιο μπλοκ B_k να είναι μεγαλύτερο από το b_{max}) είναι $O(r + b_{max})$. Επίσης, ο χρόνος γίνεται $O(n \log n + \upsilon n)$ (χωρίς επαναλήψεις κειμένου μεγαλύτερες από υ, δηλαδή οποιαδήποτε δυο suffixes να μπορούν να συγκριθούν σε χρόνο O(υ)) και O(n logn) κατά μέσο όρο για τυχαία κείμενα.

Τώρα, σειρά έχει το βήμα 3a του αλγορίθμου. Υπάρχει καλύτερη μέθοδος για την εκτέλεσή του που βασίζεται σε έναν γραμμικού χρόνου αλγόριθμο για την εύρεση όλων των suffixes ενός κειμένου T που είναι λεξικογραφικά μικρότερα από ένα string ερωτήματος P. Το όνομά του είναι SmallerSuffixes και η ιδέα του έχει ως εξής: υπολογισμός του μήκους του μεγαλύτερου κοινού prefix (lcp) μεταξύ του P και κάθε suffix του T και σύγκριση των χαρακτήρων που είναι σε αναντιστοιχία για τον καθορισμό της σειράς. Ο lcp-υπολογισμός εκμεταλλεύεται τις ήδη υπολογισμένες lcp-τιμές μεταξύ του P και ενός νωρίτερου suffix του T και μεταξύ του P και του δικού του suffix. Έτσι, βρίσκει όλα τα suffixes του κειμένου T[0,n) που είναι λεξικογραφικά μικρότερα από το string P[0,m) σε O(n+m) χρόνο και O(m) επιπλέον

χώρο. Η συνολική χρονική πολυπλοκότητα των βημάτων κατασκευής μπλοκ (Βήμα 3a) είναι $O(rn)$. Η πολυπλοκότητα χώρου (εκτός του κειμένου) είναι $O(b_{max} + \upsilon)$ (εφόσον το κείμενο δεν έχει επαναλήψεις μεγαλύτερες από υ και κανένα μπλοκ B_k δεν είναι μεγαλύτερο από b_{max}).

Όμως, εάν το κείμενο έχει επαναλήψεις μεγαλύτερες από υ, τότε δημιουργούνται δυο προβλήματα:

1) Ο χρόνος ταξινόμησης χειρότερης περίπτωσης αυξάνεται σε $\Theta(n^2)$, επειδή το $DP(S_{Bk})$ μπορεί να γίνει $\Theta(n|B_k|)$.
2) Ο προϋπολογισμένος lcp πίνακας στο SmallerSuffixes αλγόριθμο ίσως χρειαστεί να μεγαλώσει στο μέγεθος Θ(n).

Και τα δυο προβλήματα λύνονται χρησιμοποιώντας ένα difference cover sample (DCS). Ένα difference cover sample $DCS_\upsilon(T)$ ενός κειμένου T είναι μια δομή δεδομένων που επιτρέπει την αποτελεσματική σύγκριση των suffixes. Το $DCS_\upsilon(T)$ του κειμένου T με περίοδο υ ε [3,n] μπορεί να κατασκευαστεί σε $O(|D|\log|D| + DP^\upsilon(D))$ χρόνο και $O(\upsilon + |D|)$ χώρο (εκτός του κειμένου), όπου D είναι ένα σύνολο από $\Theta(n/\sqrt{\upsilon})$ suffixes. Έστω S_i και S_j δυο suffixes του T με ένα κοινό prefix μήκους υ-1 δηλαδή $T[i,i+\upsilon-1) = T[j,j+\upsilon-1)$. Δοθέντος του $DCS_\upsilon(T)$ η λεξικογραφική σειρά του S_i και του S_j μπορεί να προσδιοριστεί σε σταθερό χρόνο. Ο BWT αλγόριθμος στην αρχή κατασκευάζει το $DCS_\upsilon(T)$ και μετά το χρησιμοποιεί μαζί με κάποιες τεχνικές ώστε να μειώσει τη χρονική πολυπλοκότητα ταξινόμησης σε $O(n \log n + DP^\upsilon(S_{[0,n]}))$. Το $DCS_\upsilon(T)$ μπορεί να κατασκευαστεί σε $O((n/\sqrt{\upsilon}) \log(n/\sqrt{\upsilon}) + DP^\upsilon(S_{[0,n]}))$ χρόνο και $O(\upsilon + n/\sqrt{\upsilon})$ χώρο (εκτός του κειμένου).

Για τον αλγόριθμο θα πρέπει κανένα μπλοκ να μην είναι μεγαλύτερο από b_{max} και παράλληλα να διατηρηθεί μικρός ο αριθμός r των μπλοκ επειδή η επεξεργασία κάθε μπλοκ χρειάζεται τουλάχιστον Ω(n) χρόνο. Υπάρχει μια ντετερμινιστική διαδικασία, που τροποποιεί το βήμα 3a και κάνει όλα τα μεγέθη μπλοκ μεταξύ $b_{max}/2$ και b_{max} να επιτυγχάνουν τον ασυμπτωτικά βέλτιστο αριθμό $r = O(n/b_{max})$ των μπλοκ. Έτσι, η συνολική χρονική πολυπλοκότητα του τροποποιημένου βήματος 3a είναι $O(n^2/b_{max} + n \log n + DP^\upsilon(S_{[0,n]}))$. Η πολυπλοκότητα χώρου (εκτός του κειμένου και του $DCS_\upsilon(T)$) είναι $O(b_{max} + \upsilon + n/b_{max})$. Πάντως, στη συγκεκριμένη περίπτωση η τροποποίηση του βήματος 3a δεν αλλάζει την ασυμπτωτική χρονική πολυπλοκότητα ολόκληρου του αλγορίθμου. Για να μπορούν να συνδυαστούν μικρά μπλοκ, χρειάζεται να είναι γνωστά τα μεγέθη τους. Τα τελευταία υπολογίζονται χρησιμοποιώντας μια δυαδικού string τεχνική αναζήτησης, που αναπτύχθηκε για δυαδική αναζήτηση σε suffix arrays. Το αποτέλεσμα της τεχνικής είναι το εξής: έστω M ένα ταξινομημένο σύνολο από m strings. Το σύνολο M μπορεί να προεπεξεργαστεί σε $O(DP(M))$ χρόνο και $O(m)$ χώρο έτσι ώστε μια δυαδική αναζήτηση στο M, χρησιμοποιώντας ένα string ερωτήματος S, να μπορεί να επιτευχθεί σε $O(\log m + DP_M(S))$ χρόνο.

Συνοψίζοντας τις ιδιότητες του προτεινόμενου BWT αλγορίθμου, φαίνεται ότι ο αλγόριθμος ελέγχεται από δυο παραμέτρους: υ, την περίοδο του difference cover sample και b_{max}, το μέγιστο μέγεθος μπλοκ. Η άθροιση των χρονικών πολυπλοκοτήτων ομοίως, καθώς και η προσεκτική παρατήρηση του αλγορίθμου οδηγούν στα εξής συμπεράσματα για αυτόν:

1) Ο BWT ενός κειμένου μήκους n μπορεί να υπολογιστεί σε $O(n \log n + \sqrt{\upsilon}\, n + DP^\upsilon(S_{[0,n]}))$ χρόνο και $O(n/\sqrt{\upsilon})$ χώρο (επιπλέον στο κείμενο και τον BWT), για κάθε υ ε $[3,n^{2/3}]$.
2) Ο BWT ενός κειμένου μήκους n μπορεί να υπολογιστεί σε $O(n \log^2 n)$ χρόνο χειρότερης περίπτωσης και σε $O(n \log n)$ μέσο χρόνο, χρησιμοποιώντας $O(n)$ bits χώρο επιπλέον στο κείμενο και στον BWT.
3) Ο BWT ενός κειμένου μήκους n σε ένα αλφάβητο μήκους σ μπορεί να υπολογιστεί σε $O(n(\log n + \log_\sigma^2 n))$ χρόνο χειρότερης περίπτωσης, σε $O(n \log n)$ μέσο χρόνο και σε $O(n \log \sigma)$ bits χώρο.

Όσον αφορά την εκτέλεση του αλγορίθμου με πειραματικά δεδομένα, ο αλγόριθμος υλοποιήθηκε ως ένα πρόγραμμα bwt που διαβάζει το κείμενο από ένα αρχείο και γράφει το BWT σε ένα άλλο αρχείο. Επίσης, υλοποιήθηκε και ένα δεύτερο πρόγραμμα dnabwt για μερικά συγκεκριμένα πειραματικά δεδομένα. Κάθε πρόγραμμα έχει διαφορετική τιμή υ, με

σκοπό να επιτύχει την ελάχιστη συνολική κατανάλωση χώρου. Για σύγκριση χρησιμοποιήθηκαν δυο προγράμματα που βασίζονται σε γρήγορους και χωρικά αποδοτικούς αλγορίθμους για την κατασκευή ολόκληρου του suffix array. Το πρώτο (MF) είναι ο deep-shallow αλγόριθμος του Manzini και του Ferragina και το δεύτερο (BK) είναι ο αλγόριθμος του Burkhardt και του Karkkainen. Τα προγράμματα γράφτηκαν σε C++ και μεταφράστηκαν με g++ -03 εκτός του MF που είναι γραμμένο σε C και μεταφράστηκε με gcc -03. Οι δοκιμές έτρεξαν σε έναν υπολογιστή με 2.6 GHz Intel Pentium 4 επεξεργαστή και 4 GB κύρια μνήμη σε Linux. Ακόμη, χρησιμοποιήθηκαν έξι διαφορετικά είδη κειμένου και τα αποτελέσματα ήταν τα εξής: ο bwt είναι σχετικά ανταγωνιστικός σε ταχύτητα. Είναι 2-3 φορές πιο αργός από το MF για τα περισσότερα κείμενα, αλλά πολύ πιο γρήγορος στα επαναλαμβανόμενα δεδομένα καθώς παίρνει μόλις το ένα-τρίτο του χώρου. Οι χρόνοι για τον bwt και το BK είναι παρόμοιοι, επειδή και οι δυο σπαταλούν τον περισσότερο χρόνο τους στην ταξινόμηση των strings. Η μεγαλύτερη επιβράδυνση του bwt για επαναλαμβανόμενα δεδομένα οφείλεται πιθανώς στην μεγαλύτερη τιμή της παραμέτρου υ. Το dnabwt είναι σημαντικά πιο αργό από το bwt, αλλά αρκετά γρήγορο για τον υπολογισμό του BWT για κείμενα πολλών gigabytes. Παρόλα αυτά, σημαντική αύξηση ταχύτητας μπορεί να επιτευχθεί χρησιμοποιώντας συγκεκριμένες τεχνικές.

<u>Παρόμοιοι αλγόριθμοι μεταγενέστεροι (συνοπτικά):</u>

Δεδομένου ότι υπάρχουν και αρκετοί άλλοι μεταγενέστεροι αλγόριθμοι που βασίζονται στην ίδια ιδέα με αυτόν που αναφέρθηκε σε αυτή την ενότητα, δηλαδή τον κατευθείαν υπολογισμό του BWT χωρίς την κατασκευή του πλήρους suffix array ως ενδιάμεση δομή, αξίζει ορισμένοι από αυτούς να αναφερθούν συνοπτικά (με χρονική σειρά):

1. Οι W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung και S.M. Yiu δημιούργησαν έναν αλγόριθμο που βασίζεται στην ιδέα της κατασκευής συμπιεσμένων suffix arrays κατευθείαν από το κείμενο. Χρειάζεται O(n lgn) χρόνο και O(n lgσ) bits χώρο. Επίσης, αυτός ο αλγόριθμος είναι ιδιαίτερα αποδοτικός για κείμενα με μεγάλα αλφάβητα.
2. Οι D. Okanohara και K. Sadakane κατασκεύασαν έναν αλγόριθμο για τον υπολογισμό του BWT κατευθείαν σε γραμμικό χρόνο O(n) τροποποιώντας τον αλγόριθμο κατασκευής του suffix array βάσει της μεθόδου induced sorting. Ο χώρος που απαιτεί είναι O(n logσ $\log_\sigma$n) bits για κάθε σ, όπου το σ είναι το μέγεθος του αλφαβήτου.
3. Οι J.I. Munro, G. Navarro και Y. Nekrich έδειξαν ότι το συμπιεσμένο suffix array και το συμπιεσμένο suffix tree ενός string T, μπορεί να κατασκευαστεί σε O(n) ντετερμινιστικό χρόνο χρησιμοποιώντας O(n logσ) bits χώρο, όπου n είναι το μήκος του string και σ είναι το μέγεθος του αλφαβήτου.

Ενότητα 3: Παράλληλοι αλγόριθμοι για BW συμπίεση και αποσυμπίεση

Η συμπίεση δεδομένων χωρίς απώλειες είναι μια λειτουργία που χρησιμοποιείται συχνά σε μια πληθώρα υπολογιστικών συστημάτων. Ένας αλγόριθμος συμπίεσης χωρίς απώλειες για μια δοσμένη συνάρτηση συμπίεσης χωρίς απώλειες είναι ένας αλγόριθμος που δέχεται ως είσοδο ένα string S μήκους n σε κάποιο αλφάβητο Σ και παράγει ένα string C(S) κάποιου αλφαβήτου Σ' ως έξοδο. Ο αντίστοιχος αλγόριθμος αποσυμπίεσης χωρίς απώλειες δέχεται C(S) για κάποια S ως είσοδο και παράγει S ως έξοδο. Ένα πρόβλημα της συμπίεσης BW είναι ο υπολογισμός της συνάρτησης συμπίεσης χωρίς απώλειες όπως ορίζεται από τον παραπάνω αλγόριθμο, ενώ η αποσυμπίεση BW δυσκολεύεται στον υπολογισμό του αντιστρόφου. Εδώ, ο κύριος στόχος είναι η κατασκευή ενός $O(\log^2 n)$ χρόνου και O(n) έργου PRAM αλγορίθμου για τη λύση του προβλήματος της BW συμπίεσης, καθώς και η κατασκευή ενός O(logn) χρόνου και O(n) έργου PRAM αλγορίθμου για τη λύση του προβλήματος της BW αποσυμπίεσης.

Αρχικά το ST χρησιμοποιείται για να δηλώσει την αλληλουχία των strings S και T, το S[i] για να δηλώσει τον i-οστό χαρακτήρα του string S(0 <= i < |S|) και το S[i,j] για να δηλώσει το substring S[i]…S[j] (0 <= i <= j < |S|). Ο αρχικός BWT αποτελείται από τα εξής τρία στάδια: έναν αντιστρέψιμο μετασχηματισμό ταξινόμησης μπλοκ (BST), μια move-to-front (MTF) κωδικοποίηση και μετά μια Huffman κωδικοποίηση. Ο αλγόριθμος συμπίεσης παίρνει ως είσοδο ένα string S μήκους n, ενός αλφαβήτου Σ, με μια σταθερά |Σ| και ακολουθώντας τα τρία παραπάνω στάδια, δίνει στην έξοδό του ένα δυαδικό string S^{BW}. Ο αντίστοιχος αλγόριθμος αποσυμπίεσης εκτελεί την αντίστροφη διαδικασία. Δοθέντος ενός string S μήκους n ως είσοδο, το BST παράγει ως έξοδο (μέσα από τρία συγκεκριμένα βήματα) το S^{BST}, μια παραλλαγή του S. Το BST έχει δυο ιδιότητες που το κάνουν χρήσιμο για συμπίεση χωρίς απώλειες: (1) έχει αντίστροφο και (2) η έξοδός του τείνει να έχει πολλές εμφανίσεις οποιουδήποτε δοσμένου χαρακτήρα σε κλειστή εγγύτητα, ακόμα κι όταν η είσοδός του δεν τείνει.

Είναι σημαντικό να αποδειχθεί ότι το BST είναι αντιστρέψιμο. Η είσοδος στο αντίστροφο του BST (IBST) είναι η έξοδος του BST, S^{BST}, που είναι η δεξιότερη στήλη στον πίνακα M (προκύπτει μετά την εφαρμογή του BST σε δεδομένα εισόδου). Για την απόδειξη, πρώτα θα δειχθεί ένας όχι τόσο χρήσιμος τρόπος ώστε να αντληθεί ο ολόκληρος πίνακας M που χρησιμοποιείται από το στάδιο BST. Αυτό γίνεται με τη βοήθεια μιας επαγωγικής διαδικασίας. Ακολουθώντας το βήμα i της επαγωγής, παράγονται οι πρώτες i στήλες του M:

1) Για να ληφθεί η πρώτη στήλη του M, ταξινομούνται οι γραμμές (χαρακτήρες) του S^{BST}. Αν υπάρχουν πολλαπλές εμφανίσεις του ίδιου χαρακτήρα, διατηρείται η σχετική τους σειρά (stable sorting).
2) Για να ληφθούν οι δυο πρώτες στήλες, εισάγεται το S^{BST} στα αριστερά της πρώτης στήλης και μετά ταξινομούνται λεξικογραφικά οι γραμμές για να ληφθούν οι δυο πρώτες στήλες του M. Κατά τη σύγκριση των γραμμών, υπάρχουν δυο περιπτώσεις:
   a) Αν δυο γραμμές ξεκινούν με διαφορετικούς χαρακτήρες, διατάσσονται σύμφωνα με τους πρώτους τους χαρακτήρες.
   b) Αν δυο γραμμές ξεκινούν με τον ίδιο χαρακτήρα, διατηρείται η σχετική σειρά αυτών των δυο γραμμών.
3) Το βήμα 2 επαναλαμβάνεται ώστε να ληφθούν οι τρεις πρώτες στήλες, οι τέσσερις πρώτες στήλες και ούτω καθεξής μέχρι το M να συμπληρωθεί εντελώς.

Διατηρείται η γραμμή του M για την οποία το $ είναι στην δεξιότερη στήλη για να είναι η έξοδος του IBST αφού οι γραμμές του M είναι περιστροφές του string εισόδου όπου η πρώτη γραμμή (το ίδιο το string εισόδου) ήταν αυτή για την οποία το $ ήταν ο τελευταίος χαρακτήρας. Για να μειωθεί η παραπάνω κατασκευή ώστε να επισκέπτεται μόνο O(n) από τις εισόδους της, προτείνεται ένας O(n) σειριακός αλγόριθμος που ξετυλίγει αρκετά το M και αναπαράγει την έξοδο: και συγκεκριμένα τη γραμμή του M για την οποία το $ είναι στην

δεξιότερη στήλη. Αυτός ο O(n)-χρόνου σειριακός αλγόριθμος υπολογίζει το IBST του S^{BST} και περιγράφεται από τον ακόλουθο ψευδοκώδικα:

// Είσοδος: $S^{BST} = x_1x_2…x_n$ Έξοδος: S, το IBST του S^{BST}

1. Εφαρμογή ενός αλγορίθμου ταξινόμησης σταθερών ακεραίων για ταξινόμηση των στοιχείων των $x_1x_2…x_n$. Η έξοδος είναι μια παραλλαγή αποθήκευσης του βαθμού του i-οστού στοιχείου x_i στο T[i]
2. L[0] := 0 // Το L[j] είναι η γραμμή του $ στη στήλη j
3. Για j := 0 μέχρι n-2
   3.1. L[j+1] = T[L[j]]
   3.2. S[n-2-j] := S^{BST}[L[j]]

Δοθείσης της εξόδου S^{BST} του προηγούμενου σταδίου ως είσοδο, η MTF κωδικοποίηση αντικαθιστά κάθε χαρακτήρα με έναν ακέραιο που δείχνει τον αριθμό των διαφορετικών χαρακτήρων μεταξύ αυτού του χαρακτήρα και της προηγούμενης εμφάνισής του. Η MTF κωδικοποίηση παράγει ως έξοδο ένα string S^{MTF} στο αλφάβητο των ακεραίων [0,n-1], με $|S^{MTF}|=|S^{BST}|$. Έστω L_i μια λίστα από τους διαφορετικούς χαρακτήρες στο S^{BST}[0,i-1] σύμφωνα με τη σειρά της τελευταίας τους εμφάνισης. Ο ψευδοκώδικας του αλγορίθμου MTF κωδικοποίησης είναι ο εξής:

// Είσοδος: S^{BST} Έξοδος: S^{MTF}

1. L := L_0
2. Για i := 0 μέχρι n-1 // Στην αρχή της επανάληψης i, L=L_i
   2.1. Ορίζεται j στο ευρετήριο του S^{BST}[i] στο L
   2.2. S^{MTF}[i] := j
   2.3. Μετακίνηση του L[j] μπροστά από το L

Ο αλγόριθμος MTF αποκωδικοποίησης παίρνει το S^{MTF} ως είσοδο και παράγει το S^{BST} ως έξοδο. Ο ψευδοκώδικάς του είναι ίδιος με τον προηγούμενο εκτός από το 2.1 (j := S^{MTF}) και το 2.2 (S^{BST}[i] := L[j]). Έτσι, αφού τα στάδια κωδικοποίησης BST και MTF είναι αντιστρέψιμα, το S μπορεί να ανακτηθεί από το S^{MTF}.

Η είσοδος στο στάδιο κωδικοποίησης Huffman είναι το string S^{MTF} και παράγει ως έξοδο (1) το string S^{BW}, ένα δυαδικό string του οποίου το μήκος είναι Θ(n) και (2) ένα κωδικοποιημένο πίνακα T, του οποίου το μέγεθος είναι σταθερό δοθέντος ότι το |Σ| είναι σταθερό. Η κωδικοποίηση Huffman γίνεται σε τρία βήματα:

1) Μέτρημα του αριθμού των φορών που κάθε χαρακτήρας του Σ εμφανίζεται στο S^{MTF} για τη δημιουργία ενός πίνακα συχνοτήτων F.
2) Χρησιμοποίηση του F για την κατασκευή ενός κωδικοποιημένου πίνακα T, έτσι ώστε για κάθε δυο χαρακτήρες a,b ε Σ, αν F(a) < F(b), τότε |T(a)| >= |T(b)|.
3) Αντικατάσταση κάθε χαρακτήρα του S^{MTF} με την αντίστοιχή του κωδικολέξη στο T για την δημιουργία του S^{BW}.

Οι παράλληλοι BW αλγόριθμοι συμπίεσης και αποσυμπίεσης ακολουθούν την ίδια σειρά των παραπάνω σταδίων, αλλά κάθε στάδιο διενεργείται από έναν PRAM αλγόριθμο αντί για σειριακό. Υπάρχουν αξιοσημείωτες διαφορές μεταξύ των αλγορίθμων συμπίεσης και αποσυμπίεσης και γι' αυτό περιγράφονται ξεχωριστά. Ο αλγόριθμος συμπίεσης δέχεται ως είσοδο ένα string S μήκους n σε ένα αλφάβητο Σ. Το πρώτο του στάδιο είναι ο μετασχηματισμός ταξινόμησης μπλοκ (BST), ο οποίος υπολογίζεται με συγκεκριμένη διαδικασία από την οποία προκύπτουν τα εξής συμπεράσματα:

1. Ο αλγόριθμος βρίσκει το BST ενός string μήκους n σε ένα αλφάβητο σταθερού μεγέθους σε $O(\log^2 n)$ χρόνο χρησιμοποιώντας O(n) έργο.
2. Ο αλγόριθμος βρίσκει το BST ενός string μήκους n σε ένα αλφάβητο Σ του οποίου το μέγεθος είναι πολυωνυμικό στο n σε $O(\log^2 n)$ χρόνο χρησιμοποιώντας $O(n + |\Sigma|^{2\log^* n})$ έργο και $O(n^2 + |\Sigma|^{2\log^* n})$ χώρο.

Ο υπολογισμός του αριθμού MTF για κάθε χαρακτήρα στην έξοδο S^{BST} στο BST ισοδυναμεί με την εύρεση του L_i για κάθε i >= 0 και i < n. Αυτό επιτυγχάνεται χρησιμοποιώντας prefix

sums μαζί με έναν δυαδικό τελεστή $\oplus$. Πρώτα ορίζεται η συνάρτηση MTF(X), που κρατάει την τοπική συνεισφορά του substring X στις λίστες L_i. Μετά χρησιμοποιείται το $\oplus$ για να ενώσει αυτές τις τοπικές πληροφορίες κατά ζεύγη, παράγοντας όλα τα L_i για το ολικό string S^{BST}. Έστω ότι MTF(X) είναι η λίστα των διαφορετικών χαρακτήρων στο X σε αντίστροφη σειρά από την τελευταία εμφάνιση στο X. Το MTF(X) είναι στην ουσία η MTF λίστα του X. Το ευρετήριο ενός χαρακτήρα στο MTF(X) ισοδυναμεί με τον αριθμό των διαφορετικών χαρακτήρων που ακολουθούν την τελευταία του εμφάνιση στο X. Όταν το X είναι prefix του S^{BST}, αυτός ο ορισμός συμπίπτει με αυτόν του L_i. Τα συμπεράσματα που προκύπτουν είναι τα ακόλουθα:

1. Για ένα string που αποτελείται από έναν απλό χαρακτήρα c, MTF(c) = (c), η λίστα περιέχει το c ως το μόνο της στοιχείο.
2. Για οποιαδήποτε δυο strings X και Y, MTF(XY) = MTF(X) $\oplus$ MTF(Y).

Στόχος είναι να υπολογιστούν όλες οι λίστες L_i. Αυτό είναι ισοδύναμο με τον υπολογισμό των MTF λιστών όλων των prefixes του S^{BST}, λαμβάνοντας υπ' όψιν το υποτιθέμενο prefix. Αυτό υπολογίζεται σε έναν prefix sum υπολογισμό στον πίνακα A, όπου το A[i] αρχικοποιείται στην απλή λίστα (S^{BST}[i]). Οι λίστες που παράγονται από τον τελεστή $\oplus$ δεν έχουν περισσότερα από $|\Sigma|$ στοιχεία και έτσι ο τελεστής $\oplus$ μπορεί να υπολογιστεί σε $O(|\Sigma|)$ χρόνο και έργο. Επομένως, τα prefix sums μπορούν να υπολογιστούν σε $O(|\Sigma|\log n)$ χρόνο και $O(|\Sigma|n)$ έργο από τον PRAM αλγόριθμο για υπολογισμό όλων των prefix sums. Παρόλα αυτά ο παράλληλος αλγόριθμος χρειάζεται n φορές περισσότερο χώρο σε σχέση με τον σειριακό. Όμως, αυτές οι απαιτήσεις χώρου στον παράλληλο αλγόριθμο μπορούν να μειωθούν με ορισμένες διαδικασίες.

Ο PRAM αλγόριθμος για την κωδικοποίηση Huffman είναι μια προέκταση του σειριακού αλγορίθμου κωδικοποίησης Huffman που αναφέρθηκε παραπάνω και χρειάζεται $O(|\Sigma|\log n)$ χρόνο και $O(|\Sigma|n)$ έργο για να ολοκληρωθεί. Έτσι, μαζί και με τα προηγούμενα προκύπτουν τα εξής:

1. Ο παράλληλος αλγόριθμος λύνει το πρόβλημα της Burrows-Wheeler συμπίεσης για ένα string μήκους n σε ένα αλφάβητο του οποίου το μέγεθος είναι πολυωνυμικό στο n, σε $O(\log^2 n + |\Sigma|\log n)$ χρόνο χρησιμοποιώντας $O(|\Sigma|n + |\Sigma|^{2\log^* n})$ έργο.
2. Ο παράλληλος αλγόριθμος λύνει το πρόβλημα της Burrows-Wheeler συμπίεσης για ένα string μήκους n σε ένα σταθερό αλφάβητο, σε $O(\log^2 n)$ χρόνο χρησιμοποιώντας $O(n)$ έργο.

Ο αλγόριθμος αποσυμπίεσης δέχεται ως είσοδο το string S^{BW}. Η έξοδός του είναι το αρχικό string S που προκύπτει εφαρμόζοντας τα αντίστροφα των σταδίων του αλγορίθμου συμπίεσης σε αντίστροφη σειρά. Το πρώτο στάδιό του είναι η Huffman αποκωδικοποίηση. Το κύριο εμπόδιο στην αποκωδικοποίηση του S^{BW} παράλληλα είναι ότι, επειδή οι κώδικες Huffman είναι μεταβλητού μήκους κώδικες, δεν είναι γνωστό που βρίσκονται τα όρια μεταξύ των κωδικολέξεων στο S^{BW}. Δεν γίνεται η αποκωδικοποίηση να ξεκινήσει από οποιαδήποτε θέση, καθώς το αποτέλεσμα θα είναι λάθος αν η αποκωδικοποίηση ξεκινήσει στη μέση μιας κωδικολέξης. Έτσι, πρέπει πρώτα να βρεθεί ένα σύνολο από έγκυρες αρχικές θέσεις για την αποκωδικοποίηση. Μετά, μπορούν να αποκωδικοποιηθούν τα substrings του S^{BW} με τις αντίστοιχες αρχικές θέσεις, παράλληλα. Τα επόμενα βήματα είναι σχετικά πιο απλά και υπολογίζεται ότι η ολόκληρη η αποκωδικοποίηση Huffman απαιτεί $O(\log n + |\Sigma|)$ χρόνο και $O(n)$ έργο.

Ο παράλληλος MTF αλγόριθμος αποκωδικοποίησης μοιάζει στον παράλληλο MTF αλγόριθμο κωδικοποίησης, αλλά χρησιμοποιεί ένα διαφορετικό τελεστή για το βήμα των prefix sums. Η MTF αποκωδικοποίηση χρησιμοποιεί τους χαρακτήρες του S^{MTF} κατευθείαν ως δείκτες στις MTF λίστες L_i. Επομένως, για κάθε χαρακτήρα στο S^{MTF}, είναι γνωστή η επίδραση του αμέσως προηγούμενου χαρακτήρα στο L_i. Τώρα πρέπει να γίνει γνωστή, για κάθε χαρακτήρα στο S^{MTF}, η συνολική επίδραση όλων των προηγούμενων χαρακτήρων. Αυτό μπορεί να επιτευχθεί χρησιμοποιώντας prefix sums μαζί με συναρτήσεις σύνθεσης, όπως οι

προσεταιριστικοί δυαδικοί τελεστές. Όλος ο MTF αλγόριθμος αποκωδικοποίησης χρειάζεται O(|Σ|logn) χρόνο και O(|Σ|n) έργο.

Στο τρίτο και τελευταίο στάδιο, ο αντίστροφος μετασχηματισμός ταξινόμησης μπλοκ (IBST) μοιάζει αρκετά με τον αντίστοιχο IBST του σειριακού αλγορίθμου. Η κύρια διαφορά εντοπίζεται στο βήμα 3.1 του σειριακού ψευδοκώδικα. Στον παράλληλο αλγόριθμο αυτή η διαδικασία γίνεται μέσω μιας "αντίστροφης ένδειξης": για τον $ χαρακτήρα κάθε γραμμής i του M, βρίσκεται η στήλη του. Αυτό πραγματοποιείται μέσω αναγωγής στο πρόβλημα list ranking. Γενικότερα, το πρόβλημα list ranking αναφέρεται στη μείωση του αριθμού των απαιτούμενων περασμάτων για επεξεργασία της εισόδου. Εδώ, η είσοδος στο πρόβλημα list ranking είναι μια συνδεδεμένη λίστα που αναπαρίσταται από έναν πίνακα που περιλαμβάνει τα στοιχεία της λίστας. Κάθε είσοδος i στον πίνακα δείχνει σε μια άλλη είσοδο που περιέχει το επόμενό του next(i) στην συνδεδεμένη λίστα. Το πρόβλημα list ranking βρίσκει για κάθε στοιχείο την απόστασή του από το τέλος της λίστας. Πάντως, το IBST χρειάζεται O(logn + |Σ|) χρόνο και O(n) έργο.

Συνολικά, από τις τρεις παραπάνω παραγράφους συμπεραίνουμε τα ακόλουθα:

1. Ο παράλληλος αλγόριθμος λύνει το πρόβλημα της Burrows-Wheeler αποσυμπίεσης για ένα string μήκους n σε ένα γενικό αλφάβητο, σε O(|Σ|logn) χρόνο χρησιμοποιώντας O(|Σ|n) έργο.
2. Ο παράλληλος αλγόριθμος λύνει το πρόβλημα της Burrows-Wheeler αποσυμπίεσης για ένα string μήκους n σε ένα σταθερό αλφάβητο, σε O(logn) χρόνο χρησιμοποιώντας O(n) έργο.

Ενότητα 4: Παράλληλος υπολογισμός του BWT σε συμπαγή χώρο

Από την εισαγωγή του ο BWT είναι ο πυρήνας πολλών δομών δεδομένων για συμπίεση κειμένου και ευρετηρίαση. Ένα κλασικό ευρετήριο που υποστηρίζει διάφορες αναζητήσεις σε μια ακολουθία S[1..n] είναι το suffix array SA[1..n]. Ο BWT B[1..n] του S ορίζεται ως B[i] = S[SA[i]-1], υποθέτοντας ότι S[0] = S[n]. Συμπαγή ευρετήρια κειμένου χρησιμοποιούν το B σαν αντικαταστάτη του S και του SA για να μειώσουν σημαντικά τις χωρικές απαιτήσεις. Γενικά, είναι εύκολο να υπολογιστεί ο BWT από τον ορισμό του από την πρώτη κατασκευή SA, αν και απαιτεί τουλάχιστον n lgn bits ενδιάμεσου χώρου για να καταλήξει σε μια μικρότερη δομή δεδομένων, με κύριο πλεονέκτημα ότι μπορεί να χωρέσει στην κύρια μνήμη όποτε τα n lgn bits του SA δεν μπορούν. Υπάρχουν αρκετοί αλγόριθμοι για τον άμεσο υπολογισμό του BWT, χωρίς τη δημιουργία του SA ως ενδιάμεση δομή. Στο σενάριο παράλληλου υπολογισμού ωστόσο, δεν υπάρχουν αποδοτικοί ως προς το χώρο αλγόριθμοι για τον υπολογισμό του BWT. Εναλλακτικά μπορούν να χρησιμοποιηθούν παράλληλοι αλγόριθμοι για τον υπολογισμό του SA που δαπανά O(n lgn) bits και μετά να υπολογιστεί ο BWT σε O(n) επιπλέον έργο και O(1) βάθος. Σε αυτή την περίπτωση υπάρχουν αλγόριθμοι CREW PRAM και EREW PRAM που δουλεύουν αρκετά καλά. Σκοπός είναι να αποδειχθεί το εξής:

Θεώρημα 1. Υπάρχει ένας CREW PRAM αλγόριθμος που υπολογίζει το BWT μιας ακολουθίας S[1..n] του αλφαβήτου [1..σ] με $O(n\sqrt{\lg n})$ έργο και $O(\lg^3 n/\lg \sigma)$ βάθος, χρησιμοποιώντας O(n lg σ) bits χώρο εργασίας.

Το $O(n\sqrt{\lg n})$ είναι το καλύτερο γνωστό έργο ενός αλγορίθμου παράλληλης ταξινόμησης. Καθώς η ταξινόμηση (με σ = n) μπορεί να μειωθεί σε έναν υπολογισμό BWT, η βελτίωση του έργου απαιτεί βελτίωση στην παράλληλη ταξινόμηση. Ο αλγόριθμος που πρέπει να κατασκευαστεί ενδιαφέρει μόνο στην περίπτωση που lgσ = o(lgn). Διαφορετικά, κανένας αλγόριθμος δεν μπορεί να χρησιμοποιήσει o(n lgn) bits χώρο και σε αυτή την περίπτωση υπάρχουν άλλοι αλγόριθμοι που αποτελούν καλύτερες εναλλακτικές. Επίσης, ο αλγόριθμος θα πρέπει να είναι και πρακτικός.

Έστω ότι S = S[1..n] είναι μια ακολουθία μήκους |S|=n σε ένα αλφάβητο Σ = [1..σ], με σ <= n. Δεδομένης μιας ακολουθίας S, υπάρχουν τρεις λειτουργίες: η $\text{rank}_c(S,i)$ που αναφέρει τον αριθμό των φορών που εμφανίζεται το σύμβολο c στο prefix S[1..i], η $\text{select}_c(S,i)$ που δίνει τη θέση της i-οστης εμφάνισης του c στο S και η $\text{insert}_c(S,i)$ που μετακινεί όλα τα σύμβολα στο suffix S[i..n] μία θέση δεξιά και εισάγει το c στο S[i]. Το suffix array SA[1..n] μιας ακολουθίας S[1..n] είναι μια παραλλαγή του [1..n] έτσι ώστε για κάθε 1 <= i < n, S[SA[i]..n] < S[SA[i+1]..n] σε λεξικογραφική σειρά. Ο BWT B[1..n] του S είναι μια αναδιάταξη των συμβόλων του S που ορίζονται ως B[i] = S[SA[i]-1], λαμβάνοντας S[0] = S[n]. Για την ανάλυση του αλγορίθμου, χρησιμοποιούμε το μοντέλο υπολογισμού RAM, με μια λέξη μηχανής των Θ(lgn) bits που υποστηρίζει τις τυπικές λειτουργίες. Χρησιμοποιώντας έναν βέλτιστο αλγόριθμο χρονοδρομολόγησης ο χρόνος λειτουργίας χρησιμοποιώντας επεξεργαστές P σε PRAM μηχανή οριοθετείται σε O(W/P+D), πάντα θεωρώντας ένα μοντέλο ενοποιημένης μνήμης αναγνώσεων και εγγραφών (PRAM CREW).

Ο παράλληλος αλγόριθμος που πρέπει να κατασκευαστεί αντιστοιχεί στον παραλληλισμό του ακολουθιακού, χωρικά αποδοτικού αλγορίθμου του Munro. Ο ακολουθιακός αλγόριθμος διαιρεί την ακολουθία εισόδου S σε υποακολουθίες μεγέθους $\Delta = [\lg_\sigma n]$, κατασκευάζοντας το BWT B σε Δ βήματα. Το n είναι διαιρούμενο από το Δ. Στο πρώτο και το δεύτερο βήμα, ο αλγόριθμος συνενώνει τις ακολουθίες S_1 και S_2, όπου το S_j προκύπτει περιστρέφοντας S j σύμβολα δεξιά ($S_1 \circ S_2$). Τότε, ο αλγόριθμος υπολογίζει το suffix array του $S_1 \circ S_2$ σε γραμμικό χρόνο και χώρο. Το προκύπτον suffix array είναι ισοδύναμο με το suffix array των suffixes του S ξεκινώντας από τις θέσεις iΔ και iΔ-1, με 1 <= i <= n/Δ. Λαμβάνοντας υπόψιν τη σειρά στο suffix array τα σύμβολα στις θέσεις iΔ-1 και iΔ-2 εισάγονται στο B. Ένας πίνακας W αποθηκεύει τη θέση των suffixes iΔ-1 στο B(iΔ-j+1, στο j-οστό βήμα) και ένας

άλλος πίνακας Acc[a] αποθηκεύει τον αριθμό των εμφανίσεων των συμβόλων c<a στο B. Για τα υπόλοιπα βήματα j = 3,4,…,Δ του αλγορίθμου, τα σύμβολα S[iΔ-j] εισάγονται στις θέσεις p_i = Acc[a] + $rank_a$(B,W[i]) + c_i, όπου a = S[iΔ-j+1] και το c_i αντιστοιχεί στον αριθμό των S_1 suffixes που εμφανίζονται πριν το suffix S[iΔ-j+1..n] στο suffix array του $S_1 \circ S_j$. Σε κάθε βήμα, n/Δ νέα σύμβολα εισάγονται στο B. Στο τέλος κάθε βήματος, οι πίνακες W και Acc ενημερώνονται λαμβάνοντας υπόψιν τα νέα εισαγόμενα σύμβολα. Μόλις ολοκληρωθεί το Δ-οστό βήμα, ο πίνακας B είναι ο BWT του S.

Για τον παράλληλο αλγόριθμο ισχύει το εξής: δοθέντος ενός μετασυμβόλου M, οι λειτουργίες GETLM(M) και GETRM(M) παίρνουν τον αριστερότερο και δεξιότερο χαρακτήρα του M, αντίστοιχα. Αυτές οι λειτουργίες μπορούν να υπολογιστούν σε σταθερό χρόνο. Όπως ο ακολουθιακός αλγόριθμος έτσι και ο παράλληλος προχωρά σε Δ=[$\lg_\sigma$n] βήματα και σε κάθε βήμα εισάγει n_{ms}=[n/Δ] νέα σύμβολα στο BWT B. Το W είναι ένας πίνακας ζευγών όπου το πρώτο και το δεύτερο μέρος αναφέρονται ως W[i].f και W[i].s αντίστοιχα, ενώ το Acc[a] είναι ένας πίνακας ακεραίων που αποθηκεύει τον αριθμό των εμφανίσεων των συμβόλων c < a στο B. Στην αρχή υπάρχουν τα n lg σ bits της εξόδου του BWT B. Κατά την εκτέλεση του παράλληλου αλγορίθμου πρώτα θα μειωθεί το μετα-αλφάβητο μεγέθους σ^Δ σε ένα αλφάβητο μεγέθους 2n/Δ. Αντί να ταξινομηθεί ένας πίνακας μετασυμβόλων, ταξινομούνται τα ευρετήρια που τα αντιπροσωπεύουν στο R, χρησιμοποιώντας τα μετασύμβολα μόνο για τις συγκρίσεις. Η ταξινόμηση μπορεί να γίνει λαμβάνοντας ως είσοδο τον πίνακα R και τα μετασύμβολα και επιστρέφοντας τα ταξινομημένα ευρετήρια του πίνακα R. Κάθε μετασύμβολο αντικαθίσταται από τη θέση του ευρετηρίου του στον ταξινομημένο πίνακα. Στην περίπτωση που τα ευρετήρια αντιπροσωπεύουν ίσα μετασύμβολα, χρησιμοποιείται η μικρότερη θέση μεταξύ αυτών των ευρετηρίων. Γι' αυτό πραγματοποιείται μια σάρωση από αριστερά προς τα δεξιά στον πίνακα R μεταδίδοντας τέτοιες θέσεις μέσω του εύρους των ευρετηρίων που αντιπροσωπεύουν το ίδιο μετασύμβολο. Αυτή η σάρωση δεν απαιτεί επιπλέον χώρο. Μετά υπολογίζεται το suffix array της ακολουθίας T_c και εισάγονται τα πρώτα $2n_{ms}$ σύμβολα στο B εξάγοντας το δεξιότερο σύμβολο κάθε μετασυμβόλου. Η θέσεις των συμβόλων που προέρχονται από το S_2 αποθηκεύονται στο W. Ο πίνακας Acc γεμίζει με τη συσσωρευμένη συχνότητα των συμβόλων.

Στα εναπομείναντα Δ-3 βήματα, ο αλγόριθμος υπολογίζει τη θέση στο B των n_{ms} νέων συμβόλων. Υπολογίζονται οι τιμές $rank_a$(B,W[k]) και c_k, για κάθε 1 <= k <= n_{ms}. Για τον υπολογισμό των λειτουργιών rank, πρώτα ταξινομούνται με βάση τα ευρετήριά τους και στη συνέχεια με τον αλγόριθμο BATCHRANK απαντάται το batch των n/Δ rank λειτουργιών. Τα αποτελέσματα των rank ερωτημάτων προστίθενται στις αντίστοιχες τιμές του Acc. Για να υπολογιστούν οι c_k τιμές, υπολογίζονται το αντίστοιχο αλφάβητο του $S_1 \circ S_j$ και η ισοδύναμη ακολουθία T_c. Μετά υπολογίζεται το suffix array του T_c. Τέλος, ένα παράλληλο prefix sum εκτελείται πάνω από το suffix array μετρώντας μόνο τα suffixes του S_1. Μόλις οι πρώτες θέσεις του W περιέχουν τις τελικές θέσεις των νέων συμβόλων, το W ταξινομείται ξανά από την πρώτη θέση W[·].f και μετά εισάγεται στο B χρησιμοποιώντας τον αλγόριθμο BATCHINSERT. Ο πίνακας συχνοτήτων Acc ενημερώνεται μετά την εισαγωγή όλων των συμβόλων. Ο αλγόριθμος ταξινομεί τις καταχωρήσεις του W κατά το δεύτερό τους στοιχείο, για να αποκατασταθούν τα νέα σύμβολα και οι θέσεις τους, στις αρχικές τους θέσεις, σύμφωνα με τη διάταξη των μετασυμβόλων στο S_j. Αυτός ο παράλληλος αλγόριθμος έχει $O(\Delta(W_{sort}(n/\Delta) + W_{rank}(n/\Delta) + W_{SA}(n/\Delta) + W_{ins}(n/\Delta)) + n)$ έργο και $O(\Delta(D_{sort}(n/\Delta) + D_{rank}(n/\Delta) + D_{SA}(n/\Delta) + D_{ins}(n/\Delta)) + \Delta\lg(n/\Delta)$ βάθος. Στα Δ βήματα, η παράλληλη ταξινόμηση χρειάζεται $O(n\sqrt{\lg n})$ έργο και $O(\lg^2 n/\lg\sigma)$ βάθος. Για τον υπολογισμό του suffix array χρησιμοποιείται ο αλγόριθμος του Karkkainen, που τρέχει σε $O((n/\Delta)\sqrt{\lg n})$ έργο και $O(\lg^2 n)$ βάθος σε EREW PRAM. Οι SA κατασκευές απαιτούν $O(n\sqrt{\lg n})$ έργο και $O(\lg^3 n/\lg\sigma)$ βάθος. Ο αλγόριθμος ταξινόμησης των Han και Shen χρησιμοποιείται σαν υπορουτίνα στον SA αλγόριθμο και στον παράλληλο αλγόριθμο και λαμβάνει κλειδιά των Θ(lgn) bits. Όταν τα ταξινομημένα κλειδιά είναι σύμβολα lg σ bits, μπορεί να επεκταθεί κάθε σύμβολο ώστε να χρησιμοποιεί lgn bits

και να έχει μέγιστη κατανάλωση μνήμης O(n lg σ) bits, αφού στο μέγιστο n/Δ οι τιμές ταξινομούνται.

Τώρα πρέπει να παρουσιαστεί η κατασκευή μιας δομής δεδομένων που θα υποστηρίζει batches των m=Θ(n/Δ) rank και insert λειτουργιών χρησιμοποιώντας O(n lg σ) bits, σε μια ακολουθία B μήκους n και αλφάβητο [1..σ]. Η λύση για μικρά αλφάβητα αντιστοιχεί σε ένα wavelet tree όπου κάθε λειτουργία (rank και insert) απαιτεί O(lg σ) χρόνο. Έτσι αυτή η λύση είναι κατάλληλη μόνο για $\sigma=2^{O(\sqrt{\lg n})}$, έτσι ώστε οι O(n) λειτουργίες rank του παράλληλου αλγορίθμου να απαιτούν O(n lg σ) = O(n√lgn) έργο, που είναι επιθυμητό. Από την άλλη, η λύση για μεγάλα αλφάβητα θέλει O(n $\lg_\sigma n$) έργο, το οποίο είναι το επιθυμητό O(n√lgn) όταν το μέγεθος του αλφαβήτου είναι $\sigma=2^{\Omega(\sqrt{\lg n})}$. Όσον αφορά τα batches των rank και insert λειτουργιών σε δυαδικές ακολουθίες, πρέπει πρώτα να παρουσιαστεί μια δομή o(l) bits στην κορυφή ενός bitvector B[1..l] που υποστηρίζει m λειτουργίες rank σε O(m) έργο και O(lg l) βάθος, ενώ μπορεί να υποστηρίξει και m λειτουργίες insert σε O(m+l/lg(l+m)) έργο και O(lgn) βάθος, όπου l+m <= n σε μια Θ(lgn)-bits λέξη RAM μηχανής. Είναι γνωστό ότι οι m λειτουργίες rank μπορούν να υποβληθούν σε επεξεργασία ξεχωριστά σε O(m) συνολικό έργο και O(1) βάθος, σε ένα CREW PRAM. Για να εκτελεστεί το batch εισαγωγών $Q=\{insert_{b1}(B,i_1),\ldots,insert_{bm}(B,i_m)$ στο B[1..l] χρειάζεται χώρος για την προκύπτουσα ακολουθία, B'[1..l + m]. Οι εισαγωγές ταξινομούνται βάσει του αυξανόμενου δείκτη i_k και αναφέρονται στις τελικές θέσεις μετά τις εισαγωγές. Αυτοί οι δείκτες καθορίζουν τα εύρη των καταχωρήσεων του B που θα αντιγραφούν στο B'. Η διαδικασία της αντιγραφής πραγματοποιείται σε O(l/lgn) έργο και O(lgn) βάθος. Μετά την αντιγραφή, θέτεται το $B'[i_k]=b_k$, για 1 <= k <= m και η rank δομή δεδομένων του Shun ξαναδημιουργείται από την αρχή στο B' σε O((l+m)/lg(l+m)) έργο και O(lg(l+m)) βάθος. Το μέγιστο της κατανάλωσης μνήμης είναι O(l+m) bits και κυριαρχείται από το χώρο του B'. Όσον αφορά τα batches των rank και insert λειτουργιών για $\sigma < 2^{\sqrt{\lg n}}$ η συνολική επίπτωση στην BWT κατασκευή είναι O(n√lgn) έργο και $O(\lg^2 n)$ βάθος. Για την κατασκευή τους το B αναπαρίσταται ως ένα στατικό δυαδικό wavelet tree. Η ρίζα αυτού του δέντρου αποθηκεύει ένα bitvector B_{root} με τα υψηλότερα bits των συμβόλων του B. Το αριστερό (δεξί) του παιδί, $B_{left}(B_{right})$ αναπαριστά την υποακολουθία του B που σχηματίζεται από τα σύμβολα των οποίων τα υψηλότερα bits είναι O(1). Κάθε παιδί είναι, αναδρομικά, η ρίζα ενός wavelet tree που αναπαριστά αυτή την υποακολουθία, η οποία με τη σειρά της διαμερίζεται σύμφωνα με το επόμενο υψηλότερο bit και ούτω καθεξής. Το wavelet tree τότε, θα έχει ύψος lg σ και θα αποτελεί αναπαράσταση του B.

Με τον EREW PRAM αλγόριθμο ταξινόμησης ακεραίων με O(l√lg l) έργο και O(lg l) βάθος, δημιουργείται ένας CREW PRAM αλγόριθμος κατασκευής O(l lg σ) έργου και O(lg l lg σ) βάθους για το wavelet tree μιας ακολουθίας μήκους l. Όλα τα bitvectors B_u στο ίδιο βάθος d αποθηκεύονται σαν ένα ενιαίο bitvector B_d[1..l], στο οποίο οι κόμβοι u δείχνουν για αναγνώριση την αρχική θέση O_u της αντίστοιχής τους υποακολουθίας B_u. Μια λειτουργία $rank_b(B_u,i)$ μεταφράζεται στην αφαίρεση δυο rank λειτουργιών: $rank_b(B_d, i+O_u-1) – rank_b(B_d, O_u-1)$, που διατηρεί το σταθερό χρόνο. Για την κατασκευή του wavelet tree στα 2n/Δ σύμβολα του παράλληλου αλγορίθμου και για $\sigma < 2^{\sqrt{\lg n}}$ απαιτούνται: O(n/Δ lg σ) έργο, O(lg(n/Δ) lg σ) βάθος και O(n lg σ) bits χώρο. Ένα batch rank λειτουργιών, για να εκτελεστεί, χρειάζεται O(n√lgn) έργο και O(lgn) βάθος. Για να υποστηριχτεί ένα batch των m λειτουργιών insert $Q=\{insert_{c1}(B,i_1),\ldots,insert_{cm}(B,i_m)\}$ στο wavelet tree του B, πρέπει πρώτα να εξαχθούν τα υψηλότερα bits $b_1,\ldots,b_m$ του $c_1,\ldots c_m$ σε O(m) έργο και O(1) βάθος και να εκτελεστεί το $Q_{root}=\{insert_{b1}(B_{root},i_1),\ldots,insert_{bm}(B_{root},i_m)\}$ σε O(m) συνολικό έργο και O(lgn) βάθος. Στη συνέχεια, μοιράζονται οι λειτουργίες του Q μεταξύ των δυο παιδιών της ρίζας (αριστερό και δεξί). Ένας τρόπος να υπολογιστούν οι νέες θέσεις insert σε κάθε παιδί είναι με i_k <- $rank_{bk}(B_{root},i_k)$. Αυτή η διαδικασία απαιτεί O(m) έργο και O(lgn) βάθος και συνεχίζει με τον ίδιο τρόπο στα παρακάτω επίπεδα. Για να διατηρηθεί η πολυπλοκότητα σε βαθύτερα επίπεδα, ωστόσο, απαιτείται όλες οι m λειτουργίες insert σε κάθε επίπεδο να εκτελούνται

μαζί, στο ενιαίο bitvector $B_d[1..l]$. Αυτή η διαδικασία αποφέρει συνολικά O(n lg σ) έργο και $O(lg^2n)$ βάθος.

Τώρα πρέπει να αναλυθεί η διαχείριση των batches των rank/insert λειτουργιών για μεγάλα αλφάβητα (δηλαδή $\sigma >= 2^{\sqrt{lgn}}$) ώστε να αποφέρουν $O(n\sqrt{lgn})$ συνολικό έργο και $O(lg^2n/lg\ \sigma)$ βάθος. Για αυτή τη διαδικασία θα πρέπει πρώτα να απλοποιηθεί η δομή και μετά να παραλληλοποιηθεί η κατασκευή και η χρήση της. Η δουλειά πάνω στη δομή περιλαμβάνει αρχικά το μοίρασμα του S[1..n] σε μια linked list των $[n/\sigma]$ κομματιών C_g μήκους σ. Για την υποστήριξη rank/insert λειτουργιών μέσα σε κάθε κομμάτι C, τα σύμβολά του C[i]=c αναπαρίστανται ως μια ακολουθία R με (c, i, $rank_c(C,i)$). Έστω R_c το εύρος του R με σύμβολο c. Για την κατασκευή της δομής σε ένα κομμάτι C[1..σ], το τελευταίο αναπαρίσταται σαν την ακολουθία R με (C[i],i,0). Το R ταξινομείται κατάλληλα και βρίσκονται οι αρχικές θέσεις r_c των ευρών R_c. Το τρίτο στοιχείο κάθε R[j]=(c,i,0) συμπληρώνεται με $j-r_c+1$. Οι εσωτερικές δομές δεδομένων των κομματιών για τα πρώτα 2n/Δ σύμβολα του παράλληλου αλγορίθμου κατασκευάζονται σε $O(n/\Delta\ \sqrt{lg}\ \sigma)$ έργο και O(lg σ) βάθος. Μετά την κατασκευή των εσωτερικών δομών δεδομένων, κατασκευάζονται οι καθολικοί bitvectors M_c, $1 <= c <= \sigma$. Η κατασκευή των M_c για τα 2n/Δ σύμβολα χρειάζεται O(n/Δ) έργο, O(lg (n/(Δ σ)) βάθος και O(n lg σ) bits. Μόλις τερματιστεί ο αλγόριθμος κατασκευής BWT, λαμβάνεται η επιθυμητή ακολουθία από κάθε κομμάτι του R σε $O(n\sqrt{lg}\ \sigma)$ συνολικό έργο και O(lg σ) βάθος.

Για το batch των rank λειτουργιών, έστω ότι $Q = \{rank_{c1}(B,i_1),\ldots,rank_{cm}(B,i_m)\}$ είναι τα m ερωτήματα rank. Η απάντηση του καθενός περιλαμβάνει δυο στοιχεία, $rank_{ck}(B,i_k) = r_{k,1} + r_{k,2}$. Το πρώτο μετρά τον αριθμό των c_ks που προηγούνται του i_k πριν το κομμάτι του και το δεύτερο τα c_ks μέχρι το i_k εντός του κομματιού του. Πρώτα ανακαλύπτεται το ευρετήριο g_k του κομματιού που ανήκει κάθε ευρετήριο i_k. Τα παράλληλα prefixes και οι συγχωνεύσεις που γίνονται απαιτούν $O(m+W_{merge}(m))$ έργο και $O(lgm+D_{merge}(m))$ βάθος. Έστω Q_g το batch των t rank ερωτημάτων που θα απαντηθούν μέσα στο g-οστό κομμάτι C. Τα ερωτήματα αναπαρίστανται με $(c_k,i_k,0)$ και μέσα από κατάλληλες διεργασίες υπολογίζεται ο δείκτης k. Μετά και την εκτέλεση του BWT κατασκευαστικού αλγορίθμου όλα τα batched rank ερωτήματα γίνονται σε $O(nlg_\sigma n + n\sqrt{lgm})$ έργο και $O(lg^2n/lg\ \sigma)$ βάθος. Σχετικά με το batch των insert λειτουργιών η εισαγωγή των m νέων συμβόλων λύνεται σε δυο στάδια: το πρώτο είναι η τροποποίηση των εσωτερικών δομών δεδομένων των ενημερωμένων κομματιών και το δεύτερο είναι η ενημέρωση των καθολικών δυαδικών ακολουθιών M_c. Για την επιτέλεση αυτού του σκοπού χρειάζεται συνολικά $O(n\sqrt{lgn})$ έργο και $O(lg^2n/lg\ \sigma)$ βάθος. Από όλες τις διαδικασίες της υπερχείλισης, συμπεριλαμβανομένων και των Δ-2 επαναλήψεων για την κατασκευή του BWT, προκύπτει συνολικά: $O(n\sqrt{lgn})$ έργο και $O(lg^2n/lg\ \sigma)$ βάθος. Μετά τις υπερχειλίσεις στο C, χρειάζεται και η ανάλογη ενημέρωση των bitvectors M_c. Το συνολικό κόστος που προκύπτει από αυτό είναι: $O(n\sqrt{lgn})$ έργο και $O(lg^2n/lg\ \sigma)$ βάθος.

Όλα αυτά που αναφέρθηκαν παραπάνω πιστοποιούνται και πρακτικά, με την εκτέλεση του παράλληλου αλγορίθμου με χρήση πολλαπλών νημάτων και p επεξεργαστές σε μια w-bit μηχανή. Πιο συγκεκριμένα, η πρακτική εφαρμογή του αλγορίθμου γίνεται σε O(n lg σ) bits χώρο εργασίας και έχει μέση πολυπλοκότητα (με p νήματα), $O((n/p)\ lg(n/\Delta) + ((n\Delta\sigma\ lg\sigma\ lglgn)/(p\ lg^2n)) + \Delta\sigma + polylog\ n)$. Το μέσο έργο είναι $O(n\ lg(n/\Delta) + ((n\Delta\sigma\ lg\sigma\ lglgn)/(lg^2n) + \Delta\sigma)$ και το βάθος $O(\Delta\sigma + \Delta^{1-\varepsilon}n^{\varepsilon})$, εξαιρώντας τα batches των inserts που γίνονται ακολουθιακά για εξοικονόμηση χώρου. Η μέγιστη κατανάλωση χώρου είναι $9(n/\Delta)\ lgn + o(n)$ bits. Μια πειραματική μελέτη αυτού του αλγορίθμου γίνεται σε C++ και μεταγλωττίζεται με GCC 6.3 και επιλογή -03. Από την εκτέλεση αυτού του αλγορίθμου με άλλους παρόμοιους, γίνεται αντιληπτό ότι αποτελεί μια ενδιαφέρουσα πρόταση στον τομέα, δεδομένου κιόλας ότι εστιάζει στον περιορισμό κατανάλωσης της κύριας μνήμης. Είναι ιδιαίτερα ανταγωνιστικός αφού χρησιμοποιεί $O(n\sqrt{lg}\ n)$ έργο και $O(lg^3n/lg\ \sigma)$ βάθος και χρησιμοποιεί το λιγότερο χώρο μεταξύ των διάφορων προηγμένων ακολουθιακών και παράλληλων εναλλακτικών, εντός ανταγωνιστικού χρόνου κατασκευής.

Ενότητα 5: Κατασκευή ενός FM-index ανεξάρτητο του αλφαβήτου

Ένα full-text index είναι μια δομή δεδομένων που επιτρέπει τον προσδιορισμό των occ εμφανίσεων ενός μικρού προτύπου $P = p_1p_2…p_m$ σε ένα μεγάλο κείμενο $T = t_1t_2…t_n$ χωρίς την ανάγκη σάρωσης όλου του κειμένου T. Το κείμενο και το πρότυπο είναι ακολουθίες χαρακτήρων σε ένα αλφάβητο Σ μεγέθους σ. Στην πράξη κάποιος θέλει να ξέρει τις τιμές των occ (ερώτημα καταμέτρησης), τις θέσεις κειμένου που εμφανίζονται αυτά τα occ (ερώτημα εντοπισμού) και συχνά ένα περιβάλλον κειμένου γύρω τους (ερώτημα προβολής). Η μεγάλη απαίτηση χώρου των παραδοσιακών full-text ευρετηρίων έχει εγείρει ένα φυσικό ενδιαφέρον για συνοπτικά full-text ευρετήρια που επιτυγχάνουν καλές αντισταθμίσεις μεταξύ του χρόνου αναζήτησης και της πολυπλοκότητας χώρου. Εδώ παρουσιάζεται μια εναλλακτική προσέγγιση για την αφαίρεση του μεγάλου χώρου συναρτήσει του FM-index, χρησιμοποιώντας συμπίεση Huffman στο κείμενο και μετά εφαρμογή του BWT επάνω του. Η προκύπτουσα δομή μπορεί να θεωρηθεί σαν ένα FM-index σε μια δυαδική ακολουθία. Το ευρετήριο αυτό χρειάζεται $n(2H_0 + 3 + \varepsilon)(1 + o(1))$ bits χώρο, για κάθε $0 < \varepsilon < 1$. Επιλύει ερωτήματα καταμέτρησης σε $O(m(H_0+1))$ μέσο χρόνο. Η θέση κειμένου της κάθε εμφάνισης μπορεί να εντοπιστεί στη χειρότερη σε $O((1/\varepsilon)(H_0+1) \log n)$ χρόνο. Οποιοδήποτε substring κειμένου μήκους L μπορεί να εμφανιστεί σε $O((H_0+1) L)$ μέσο χρόνο, επιπλέον στον προαναφερόμενο χειρότερο χρόνο που απαιτείται για τον εντοπισμό μιας θέσης κειμένου. Στη χειρότερη περίπτωση όλοι οι όροι (H_0+1) στις χρονικές πολυπλοκότητες γίνονται $\log n$.

Σε αυτό το σημείο κρίνεται σκόπιμο να γίνει μια σύντομη αναφορά στον αλγόριθμο αναζήτησης των Ferragina και Manzini στο FM-index. Για να περιγραφεί ο αλγόριθμος αναζήτησης χρειάζονται τρεις χρήσιμοι ορισμοί:

1. Το suffix array A του κειμένου T$ (το $ είναι ειδικός τελικός δείκτης) είναι ο πίνακας M: $A[i] = j$ αν και μόνο αν η i-οστή σειρά του M περιέχει το string $t_jt_{j+1}…t_n\$t_1…t_{j-1}$.
2. Δοθέντος ενός κειμένου T σε ένα διατεταγμένο αλφάβητο $\Sigma = \{c_1,…,c_\sigma\}$, το $C[c_1,c_\sigma]$ αποθηκεύει στο $C[c_i]$ τον αριθμό των εμφανίσεων των χαρακτήρων $\{c_1,…,c_{i-1}\}$ στο T.
3. Αν X είναι μια ακολουθία, τότε Occ(X,c,i) είναι ο αριθμός των εμφανίσεων του χαρακτήρα c στο prefix X[1,i].

Ο αλγόριθμος αναζήτησης βρίσκει το διάστημα του A που περιέχει τις εμφανίσεις του προτύπου P και επιστρέφει τον αριθμό των εμφανίσεων. Ο αλγόριθμος χρησιμοποιεί τον πίνακα C και τη συνάρτηση Occ(X,c,i) που ορίστηκαν παραπάνω. Αυτό που διατηρείται αμετάβλητο είναι το εξής: Μετά τη φάση i, με i από m έως 1, η μεταβλητή sp δείχνει στην πρώτη γραμμή του M prefixed με P[i,m] και η μεταβλητή ep δείχνει στην τελευταία γραμμή του M prefixed με P[i,m]. Η ορθότητα του αλγορίθμου προκύπτει από αυτή την παρατήρηση. Ο Ferragina και ο Manzini περιγράφουν μια εκτέλεση του $Occ(T^{bwt},c,i)$ που χρησιμοποιεί μια συμπιεσμένη μορφή του T^{bwt}. Δείχνουν τον τρόπο υπολογισμού του $Occ(T^{bwt},c,i)$ για κάθε c και i σε σταθερό χρόνο. Το FM-index μπορεί επίσης να εντοπίσει τις θέσεις κειμένου όπου εμφανίζεται το πρότυπο P και να εμφανίσει οποιοδήποτε substring κειμένου.

Αυτό που πρέπει να ειπωθεί αρχικά για το νέο ευρετήριο που προτείνεται είναι ότι από εδώ και στο εξής το T θα περιέχει ήδη το τερματικό $ στο end^b. Αυτό το κείμενο T θα συμπιεστεί με Huffman σε ένα δυαδικό stream T’ και τα αρχικά των κωδικολέξεων θα σημειωθούν στο Th (το τελευταίο ευρετήριο δε θα αποθηκεύει T’ ούτε Th). Η ιδέα είναι ότι μπορεί να κωδικοποιηθεί με Huffman το P σε P’ και να γίνει αναζήτηση στο δυαδικό κείμενο T’ για το P’. Ωστόσο πρέπει να είναι σίγουρο ότι οι εμφανίσεις του P’ είναι ευθυγραμμισμένες βάσει των κωδικολέξεων.

Ορισμός: Έστω ότι T’[1,n’] είναι το δυαδικό stream που προκύπτει από τη συμπίεση Huffman T, όπου $n' < (H_0+1)n$ αφού το (δυαδικό) Huffman θέτει τη μέγιστη αναπαράσταση πάνω από 1 bit ανά σύμβολο. Έστω ότι Th[1,n’] είναι ένα δεύτερο δυαδικό stream έτσι ώστε $Th[i] = 1$ αν και μόνο αν i είναι η αρχική θέση μιας κωδικολέξης Huffman στο T’. Στον κώδικα Huffman, πρέπει το τελευταίο bit που εκχωρείται στο τελικό σύμβολο $ να είναι 0. Σχετικά με τη δομή

του παρουσιαζόμενου ευρετηρίου, πρώτα εφαρμόζεται ο BWT στο κείμενο T', έτσι ώστε να ληφθεί B = $(T')^{bwt}$.

Ορισμός: Έστω ότι A'[1,n'] είναι το suffix array για το κείμενο T'. Αυτό είναι μια μετάθεση του [1,n'] έτσι ώστε T'[A'[i],n'] < T'[A'[i+1],n'] σε λεξικογραφική σειρά, για όλα τα 1 <= i < n'. Σε αυτές τις λεξικογραφικές συγκρίσεις, αν ένα string x είναι ένα prefix του y, τότε θεωρείται x<y.

Το ευρετήριο θα αναπαριστά το A' σε συνοπτική μορφή, μέσω ενός πίνακα B και ενός άλλου πίνακα Bh που χρησιμοποιείται για να ανιχνεύει τα αρχικά των κωδικολέξεων στο $(T')^{bwt}$.

Ορισμός: Έστω ότι B[1,n'] είναι ένα δυαδικό stream έτσι ώστε B[i] = T'[A'[i]-1] (εκτός από B[i] = T[n'] αν A'[i] = 1). Έστω ότι Bh[1,n'] είναι ένα άλλο δυαδικό stream έτσι ώστε Bh[i] = Th[A'[i]]. Αυτό δείχνει αν η θέση i στο A' δείχνει στην αρχή μιας κωδικολέξης.

Στόχος τώρα είναι η αναζήτηση του B ακριβώς όπως στο FM-index. Γι' αυτό χρειάζεται ο πίνακας C και η συνάρτηση Occ που εφαρμόζονται στο T' και στο B και υπολογίζονται εύκολα χάρη στη γνωστή συνάρτηση rank.

Ορισμός: Δεδομένης μιας δυαδικής ακολουθίας X, το rank(X,i) είναι ο αριθμός των 1 στο X[1,i]. Συγκεκριμένα το rank(X,0) = 0. Η αντίστροφη συνάρτηση, select(X,j), δείχνει την εμφάνιση του j-οστού 1 στο X.

Οι συναρτήσεις rank και select μπορούν να υπολογιστούν σε σταθερό χρόνο χρησιμοποιώντας μόνο o(n) επιπλέον bits πάνω από την αρχική ακολουθία των n bits. Ο πίνακας C, η συνάρτηση Occ καθώς και οι τύποι C[c]+Occ(T^{bwt},c,i) λύνονται στο ευρετήριο χρησιμοποιώντας rank στο B. Υπάρχει ωστόσο, ένα μπέρδεμα καθώς η δυαδική ακολουθία T' δεν έχει τερματικό και έτσι δεν εμφανίζεται τερματικό στο B. Έστω ότι "#" (# < 0 < 1) το τερματικό που θα έπρεπε να εμφανίζεται στο T'. Θέτεται B[$p_{\#}$] στο τελευταίο bit του T'. Αυτό είναι το τελευταίο bit της κωδικολέξης Huffman που εκχωρείται στο τερματικό $ του T και είναι 0. Ως εκ τούτου η σωστή ακολουθία B θα είναι μήκους n'+1, ξεκινώντας με 0 και θα έχει B[$p_{\#}$]=#. Γενικά ο τύπος C[c]+Occ(T^{bwt},c,i) υπολογίζεται ως εξής:

$$C[c] + Occ(T^{bwt}, c, i) = \begin{cases} i - rank(B, i) + [i < p_{\#}], & \text{if } c = 0 \\ n - rank(B, n') + rank(B, i), & \text{if } c = 1 \end{cases}$$

όπου $p_{\#} = (A')^{-1}[1]$. Επομένως, προεπεξεργάζοντας το B για τη λύση rank ερωτημάτων, μπορεί να αναζητηθεί το B ακριβώς όπως στο FM-index. Το πρότυπο αναζήτησης δεν είναι το αρχικό P, αλλά η δυαδική του κωδικοποίηση P'[1,m'] χρησιμοποιώντας τον κώδικα Huffman που εφαρμόστηκε στο T. Ωστόσο, σε αυτήν την αναζήτηση του T' για P', επιστρέφεται ο αριθμός των suffixes του T' που ξεκινούν με P'. Βέβαια σε αυτό τον αριθμό εμφανίζονται και άλλα suffixes που κανονικά δε θα έπρεπε να εμφανίζονται και γι' αυτό ο πίνακας Bh έχει ως ρόλο να φιλτράρει τις αχρείαστες εμφανίσεις suffixes και να παρουσιάζει τον αληθινό αριθμό κάνοντας την πράξη rank(Bh,ep) – rank(Bh,sp-1). Αρχικά υπολογίζεται ότι το ευρετήριο απαιτεί στο μέγιστο $2n(H_0+1)(1+o(1)) + \sigma \log n$ bits χώρο, αν και όπως θα φανεί στη συνέχεια ο χώρος αυτός θα αυξηθεί ελαφρώς. Σχετικά με το χρόνο που χρειάζεται για τα ερωτήματα καταμέτρησης, η πολυπλοκότητα αναζήτησης του είναι $O(m(H_0+1))$ και έτσι ο χρόνος του είναι καλύτερος σε σχέση με άλλα ευρετήρια. Η χειρότερη περίπτωση κόστους αναζήτησης εξαρτάται από το μέγιστο ύψος ενός Huffman δέντρου με συνολική συχνότητα n και για να βρεθεί χρειάζεται η αρωγή της ακολουθίας Fibonacci F(i). Έτσι, ένα Huffman δέντρο με βάθος d χρειάζεται το κείμενο να έχει μήκος τουλάχιστον $n >= 1 + \Sigma_{i=1}^{d} F(i) = F(d+2)$. Επομένως, το μέγιστο μήκος μιας κωδικολέξης είναι $F^{-1}(n)-2 = \log_{\phi}(n)-2+o(1)$, όπου $\phi=(1+\sqrt{5})/2$. Έτσι, το κωδικοποιημένο πρότυπο P' δεν μπορεί να είναι μεγαλύτερο από O(m logn) και αυτό είναι επίσης το χειρότερο κόστος αναζήτησης. Είναι όμως δυνατό, να μειωθεί ο χειρότερος χρόνος σε O(m log σ), χωρίς αλλαγή του μέσου χρόνου αναζήτησης ή του χώρου που χρησιμοποιείται.

Πέρα από το χρόνο που απαιτείται για τον προσδιορισμό του διαστήματος suffix array που περιέχει όλες τις εμφανίσεις, χρειάζεται να είναι γνωστές και οι θέσεις κειμένου όπου εμφανίζονται και ίσως και ένα γενικό πλαίσιο κειμένου. Γενικά, αυτό που χρειάζεται κάποιος

είναι να εξάγει αυθαίρετα substrings κειμένου από το ευρετήριο. Το ευρετήριο, χρησιμοποιώντας (1+ε)n επιπλέον bits, μπορεί να εντοπίσει κάθε εμφάνιση σε O((1/ε)(H_0+1) logn) χρόνο και να εμφανίσει ένα πλαίσιο κειμένου σε O(L log σ + logn) χρόνο, προσθέτοντας το χρόνο εντοπισμού. Σε μια μέση περίπτωση η συνολική πολυπλοκότητα για την απεικόνιση ενός διαστήματος κειμένου είναι O((H_0+1)(L+(1/ε)logn)). Ένα πρώτο πρόβλημα είναι πώς θα εξαχθούν, σε O(occ) χρόνο, οι occ θέσεις των bits που ορίζονται στο Bh[sp,ep]. Μια απλή λύση είναι το selectnext(Bh,j), που επιστρέφει τη θέση του πρώτου 1 στο Bh[j,n]. Έστω r=rank(Bh,sp-1). Τότε οι θέσεις των bits που ορίζονται στο Bh είναι select(Bh,r+1), select(Bh,r+2),…,select(Bh,r+occ). Αφού occ = rank(Bh,ep) – rank(Bh,sp-1), τότε χρησιμοποιώντας το selectnext οι θέσεις pos_1,…,pos_{occ} μπορούν να βρεθούν θέτοντας: pos_1 = selectnext(Bh,sp), pos_{i+1} = selectnext(Bh,pos_i+1). Για την ολοκλήρωση των διαδικασιών εντοπισμού και εμφάνισης χρειάζονται επιπλέον δομές. Το T’ δοκιμάζεται σε περίπου ίσα διαστήματα, έτσι ώστε να δειγματοληπτούνται μόνο τα αρχικά των κωδικολέξεων. Μια παράμετρος δειγματοληψίας 0 < ε < 1 θα ελέγχει την πυκνότητα της δειγματοληψίας και την αντίστοιχη ανταλλαγή χώρου/χρόνου.

Ορισμός: Δοθέντος 0 < ε < 1, έστω l = [(2n’/εn) logn] το βήμα δειγματοληψίας. Η δειγματοληψία του T’ είναι μια ακολουθία S[1,[(εn)/(2logn)]], έτσι ώστε το S[i] να είναι η πρώτη θέση της κωδικολέξης που καλύπτει τη θέση 1+l(i-1) στο T’, δηλαδή S[i] = select(Th,rank(Th,1+l(i-1))).

Το ευρετήριο περιλαμβάνει τρεις επιπλέον δομές: ST, TS και S. Ο TS είναι ένας πίνακας που αποθηκεύει τις θέσεις του A’ που δείχνουν τις δειγματοληπτημένες θέσεις στο T’, κατά αύξουσα σειρά θέσης κειμένου.

Ορισμός: TS[1,[(εn)/(2logn)]] είναι ένας πίνακας έτσι ώστε TS[i] = j αν και μόνο αν A’[j] = S[i]. Ο πίνακας ST σχηματίζεται χρησιμοποιώντας τις ίδιες θέσεις του A’, αλλά ταξινομημένες κατά θέση στο A’ και αποθηκεύει τη θέση τους στο T.

Ορισμός: ST[1,[(εn)/(2logn)]] είναι ένας πίνακας έτσι ώστε ST[i] = rank(Th,A’[j]), όπου j είναι η i-οστή θέση στο A’ που δείχνει σε μια θέση που φαίνεται στο S.

Τέλος, το S[i] λέει αν η i-οστή καταχώρηση του A’ που δείχνει την αρχή μιας κωδικολέξης, δείχνει δείγμα μιας θέσης κειμένου.

Ορισμός: S[1,n] είναι ένας δυαδικός πίνακας έτσι ώστε S[i] = 1 αν και μόνο αν A’[select(Bh,i)] είναι στο S.

Τώρα πρέπει να υπολογιστεί η θέση κειμένου που αντιστοιχεί σε μια καταχώρηση A’[i] για την οποία ισχύει Bh[i]=1, δηλαδή είναι μια έγκυρη εμφάνιση. Γίνεται χρήση του δυαδικού πίνακα S[rank(Bh,i)] για να προσδιοριστεί εάν το A’[i] δείχνει ή όχι στην αρχή μιας κωδικολέξης στη θέση στο ST[rank(S,rank(Bh,i))]. Αν ναι, τότε η διαδικασία τελειώνει. Αλλιώς, υπολογίζεται η θέση i’ της οποίας η τιμή είναι A’[i’] = A’[i]-1. Αυτή η διαδικασία επαναλαμβάνεται μέχρι μια νέα αρχή κωδικολέξης να βρεθεί, δηλαδή Bh[i’]=1. Μετά ελέγχεται ξανά εάν αυτή η θέση είναι δειγματοληπτημένη και ούτω καθεξής μέχρι να βρεθεί μια δειγματοληπτημένη αρχή κωδικολέξης. Αν βρεθεί τελικά θέση pos μετά από d επαναλήψεις, η απάντηση είναι pos+d καθώς θα υπάρχει μετακίνηση προς τα πίσω d θέσεις στο T. Για τον υπολογισμό του i’ από το i, το ευρετήριο έχει κατάλληλο αλγόριθμο. Για την προβολή ενός substring κειμένου T[l,r] μήκους L=r-l+1, το πρώτο βήμα είναι η δυαδική αναζήτηση στο TS για τη μικρότερη δειγματοληπτημένη θέση κειμένου που είναι μεγαλύτερη από r. Έστω j το ευρετήριο που βρέθηκε στο TS. Δοθείσης της τιμής i=TS[j], είναι γνωστό ότι S[rank(Bh,i)]=1 αφού i είναι μια δειγματοληπτημένη καταχώρηση στο A’. Η αντίστοιχη θέση στο T είναι ST[rank(S,rank(Bh,i))]. Μόλις βρεθεί η πρώτη δειγματοληπτημένη θέση κειμένου που ακολουθεί το r, γίνεται γνωστή η αντίστοιχή της θέση i στο A’. Μετά, πρέπει να γίνει κίνηση προς τα πίσω στο T’, θέση με θέση, μέχρι να βρεθεί το πρώτο bit της κωδικολέξης για T[r+1]. Μόλις ληφθούν οι L προηγούμενοι χαρακτήρες του T, διασχίζοντας κι άλλο το T’ προς τα πίσω, συλλέγονται όλα του τα bits μέχρι να βρεθεί το πρώτο bit της κωδικολέξης για T[l]. Το bit stream που συλλέγεται, αντιστρέφεται και αποκωδικοποιείται κατά Huffman, για να

βρεθεί το T[l,r]. Συνολικά, υπολογίζεται ότι χρειάζονται (1+ε)n(1+o(1)) επιπλέον bits χώρο για ερωτήματα εντοπισμού και προβολής. Αυτό αυξάνει την τελική απαίτηση χώρου σε $n(2H_0+3+\varepsilon)(1+o(1)) + \sigma\log n$ bits. Σχετικά με το χρόνο των ερωτημάτων εντοπισμού, η χειρότερη περίπτωση πολυπλοκότητας είναι $O((1/\varepsilon)(H_0+1)\log n)$, ενώ για το χρόνο των ερωτημάτων προβολής η μέση πολυπλοκότητα είναι $O((H_0+1)(L+(1/\varepsilon)\log n))$ και η χειρότερη περίπτωση πολυπλοκότητας είναι $O(L \log n + (H_0+1)(1/\varepsilon)\log n)$.

Θεώρημα: Δοθέντος ενός κειμένου T[1,n] σε ένα αλφάβητο σ και με μηδενικής τάξης εντροπία H_0, το FM-Huffman index απαιτεί $n(2H_0+3+\varepsilon)(1+o(1)) + \sigma\log n$ bits χώρο, για κάθε σταθερό $0 < \varepsilon < 1$ καθορισμένο στο χρόνο κατασκευής. Μπορεί να μετρήσει τις εμφανίσεις του P[1,m] στο T σε μέσο χρόνο $O(m(H_0+1))$ και χρόνο χειρότερης περίπτωσης $O(m \log n)$. Κάθε τέτοια εμφάνιση μπορεί να εντοπιστεί στη χειρότερη περίπτωση σε χρόνο $O((1/\varepsilon)(H_0+1)\log n)$. Κάθε substring κειμένου μήκους L μπορεί να απεικονιστεί σε μέσο χρόνο $O((H_0+1)(L+(1/\varepsilon)\log n))$ και σε χρόνο χειρότερης περίπτωσης $O((L+(H_0+1)(1/\varepsilon))\log n)$.

Τώρα θα μελετηθεί το K-ary Huffman, ένας τρόπος για τη μείωση του μεγέθους του Bh. Πιο συγκεκριμένα, αντί να χρησιμοποιηθεί Huffman σε ένα δυαδικό κωδικοποιημένο αλφάβητο, μπορεί να χρησιμοποιηθεί ένα κωδικοποιημένο αλφάβητο των k>2 συμβόλων, έτσι ώστε κάθε σύμβολο να χρειάζεται [logk] bits. Επίσης, χρησιμοποιούνται μόνο δυνάμεις του 2 για τις k τιμές, για να γίνεται αναπαράσταση κάθε συμβόλου χωρίς σπατάλη χώρου. Οι απαιτήσεις χώρου ποικίλουν ανάλογα των συνθηκών. Το μέγεθος του B αυξάνεται αφού ο λόγος της συμπίεσης Huffman μειώνεται καθώς το k μεγαλώνει, ενώ αντιθέτως το μέγεθος του Bh μειώνεται αφού χρειάζεται ένα bit ανά σύμβολο, δηλαδή n' bits. Ο συνολικός χώρος που χρησιμοποιείται από τις δομές B και Bh είναι $n'(1+\log k) < n(H_0^{(k)}+1)(1+\log k)$, που είναι μικρότερος από τις απαιτήσεις χώρου της δυαδικής εκδοχής, δηλαδή $2n(H_0+1)$, για $1 <= \log k <= H_0$. Ακόμη, όπου $H_0^{(k)} = H_0/\log_2 k$. Ο χώρος για τις rank δομές αλλάζει επίσης. Η rank δομή του Bh έχει μέγεθος $o(H_0^{(k)} n)$ bits, ενώ τα ερωτήματα Occ(B,c,i), για τα οποία ισχύει Occ(B,c,i) = rank(B_c,i), χρειάζονται $o(kH_0^{(k)} n)$ bits. Σχετικά με τις χρονικές πολυπλοκότητες, το πρότυπο έχει μέσο μήκος μικρότερο από $m(H_0^{(k)}+1)$ σύμβολα. Αυτή είναι η πολυπλοκότητα καταμέτρησης, η οποία μειώνεται καθώς αυξάνεται το k. Με τη χρήση της τιμής $k = 2^{\sqrt{H0}}$, ο χρόνος καταμέτρησης είναι $O(m\sqrt{H_0})$. Από την άλλη ο μέσος χρόνος καταμέτρησης μπορεί να γίνει O(m) χρησιμοποιώντας μια σταθερά α. Για τα ερωτήματα εντοπισμού και προβολής κειμένου, η k-ary εκδοχή μπορεί να εντοπίσει τη θέση μιας εμφάνισης σε $O((1/\varepsilon)(H_0^{(k)}+1)\log n)$ χρόνο. Ομοίως, ο χρόνος που απαιτεί για να εμφανίσει ένα substring μήκους L είναι $O((H_0^{(k)}+1)(L+(1/\varepsilon)\log n))$ κατά μέσο όρο και $O(L \log n + (H_0^{(k)}+1) (1/\varepsilon)\log n)$ στη χειρότερη περίπτωση.

Η κωδικοποίηση Kautz-Zeckendorf, είναι μια προσπάθεια να φύγει εντελώς ο πίνακας Bh, αντικαθιστώντας την κωδικοποίηση Huffman με άλλη για την οποία το bit stream επιτρέπει το ίδιο, το συγχρονισμό στα όρια των κωδικολέξεων. Αυτή η τεχνική στηρίζεται στην ακολουθία Fibonacci. Πιο συγκεκριμένα, είναι εύκολο να αποδειχθεί ότι κάθε φυσικός αριθμός N μπορεί να σπάσει μοναδικά σε ένα άθροισμα αριθμών Fibonacci, όπου κάθε αριθμός αθροίζεται το πολύ μια φορά και δεν χρησιμοποιούνται δυο διαδοχικά στοιχεία της ακολουθίας στο σπάσιμο. Έτσι το N μπορεί να αναπαρασταθεί σαν ένα bit vector, του οποίου το i-οστό bit ορίζεται αν και μόνο αν ο i-οστός αριθμός Fibonacci χρησιμοποιείται για την αναπαράσταση του N. Η επανάληψη είναι $f_i^{(k)} = i$ για $i <= k$ και $f_{i+k}^{(k)}=f_{i+k-1}^{(k)}+f_{i+k-2}^{(k)}+...+f_{i+1}^{(k)}+f_i^{(k)}$, όπου k αριθμός bits. Σε αυτή την αναπαράσταση δεν εμφανίζεται κανένα stream των k άσσων στο bit vector. Λειτουργικά, οι κώδικες ενός δοσμένου μήκους λαμβάνονται με τη δημιουργία όλων των δυαδικών ακολουθιών αυτού του μήκους και μετά αφαιρώντας εκείνους που έχουν k συνεχόμενους άσσους. Απαιτείται επίσης, η κωδικολέξη να τελειώνει με μηδέν. Μετά δημιουργούνται οι κώδικες αυξάνοντας το μήκος και επιπλέον όλες οι κωδικολέξεις με μια ακολουθία k άσσων προηγούνται, ακολουθούμενες από ένα μηδέν. Αν κατά τη διάρκεια του

LF-mapping, διαβαστεί ένα 0 και μετά k διαδοχικοί άσσοι από το T', τότε βρισκόμαστε στην αρχή μιας κωδικολέξης. Έτσι, γίνεται αντιληπτό ότι το Bh, το select και το selectnext δεν χρειάζονται. Επειδή οι κωδικολέξεις τελειώνουν σε 0, οι μεγαλύτερες διαδρομές των άσσων είναι ακριβώς οι κεφαλίδες των κωδικολέξεων, των k άσσων. Αυτά είναι τα λεξικογραφικά μεγαλύτερα suffixes του T', και έτσι οι χαρακτήρες που προηγούνται καταλαμβάνουν τις n μεγαλύτερες θέσεις στο B. Καθώς όλοι αυτοί οι προηγούμενοι χαρακτήρες είναι 0, μπορούν να αφαιρεθούν τα τελευταία n bits από το B γνωρίζοντας ότι αυτά θα είναι μηδέν. Αυτή η διαδικασία εξοικονομεί ένα επιπλέον bit ανά σύμβολο στο T.

Τα πειραματικά αποτελέσματα σχετικά με τα ερωτήματα καταμέτρησης, εντοπισμού και προβολής και η σύγκριση της αποδοτικότητας των παραπάνω ευρετηρίων με άλλα παρόμοια παρουσιάζουν πολύ ενδιαφέρον. Για τα πειράματα εξετάστηκαν τρεις τύποι κειμένου: αγγλικό κείμενο, DNA και πρωτεϊνικές αλληλουχίες. Τα πειράματα έτρεξαν σε έναν Intel(R) Xeon(TM) επεξεργαστή στα 3.06 GHz, μια μνήμη RAM των 2GB, μια κρυφή μνήμη των 512KB και λειτουργικό σύστημα Gentoo Linux 2.6.10. Ο κώδικας μεταγλωττίστηκε με gcc 3.4.2 χρησιμοποιώντας επιλογή βελτιστοποίησης -09. Τα αποτελέσματα που προέκυψαν είναι τα εξής: για τα ερωτήματα καταμέτρησης το FM-Huffman index που προτείνεται με k=16 είναι το γρηγορότερο σχετικά με αγγλικά και πρωτεΐνες, ενώ το FM-KZ είναι ο ξεκάθαρος νικητής στο DNA. Για τα ερωτήματα εντοπισμού, τα ευρετήρια δεν παρέχουν ανταγωνιστικές ανταλλαγές χώρου/χρόνου στα αγγλικά αλλά ούτε και στις πρωτεΐνες, αλλά για το DNA το FM-KZ index δίνει εντελώς την καλύτερη ανταλλαγή. Όσον αφορά το χρόνο προβολής, οι παραλλαγές του FM-Huffman index είναι και πάλι οι πιο γρήγορες. Σε γενικές γραμμές, το FM-Huffman index είναι σε πολλές περιπτώσεις το ταχύτερο, αν και δεν μπορεί να λειτουργήσει σε πολύ μικρό χώρο όπως άλλα ευρετήρια. Στο DNA, από την άλλη, το FM-KZ είναι στις περισσότερες περιπτώσεις το μικρότερο και το ταχύτερο ευρετήριο. Πάντως, τα πειραματικά αποτελέσματα δείχνουν ότι τα ευρετήρια που αναλύθηκαν παραπάνω είναι ανταγωνιστικά στην πράξη έναντι άλλων εφαρμοσμένων εναλλακτικών λύσεων.

Ενότητα 6: Ένα "κλιμακούμενο του αλφαβήτου εισόδου" FM-index

Ένα full-text index είναι μια δομή δεδομένων που βασίζεται σε ένα string κειμένου T[1, n] που υποστηρίζει την αποτελεσματική αναζήτηση ενός αυθαίρετου προτύπου ως ένα substring του ευρετηριοποιημένου κειμένου. Ένα self-index είναι ένα full-text index που ενσωματώνει το ευρετηριοποιημένο κείμενο T, χωρίς επομένως να απαιτείται η ρητή αποθήκευσή του. Το FM-index είναι το πρώτο self-index που πετυχαίνει μια χωρητικότητα κοντά στην k-οστή σειρά εντροπίας του T που συμβολίζεται με $H_k(T)$. Ακριβώς, το FM-index καταλαμβάνει το πολύ $5nH_k(T) + o(n)$ bits αποθήκευσης και επιτρέπει την αναζήτηση των occ εμφανίσεων ενός προτύπου P[1, p] εντός T σε $O(p+occ \log^{1+\varepsilon} n)$ χρόνο, όπου ε>0 είναι μια αυθαίρετη σταθερά που καθορίζεται εκ των προτέρων. Μπορεί να εμφανίσει οποιοδήποτε substring κειμένου μήκους l σε $O(l+\log^{1+\varepsilon} n)$ χρόνο. Από το σχεδιασμό του, το FM-index φαίνεται ότι είναι ένα είδος συμπιεσμένου suffix array που εκμεταλλεύεται τη δυνατότητα συμπίεσης του ευρετηριοποιημένου κειμένου προκειμένου να επιτευχθεί πληρότητα χώρου κοντά στο ελάχιστο Information Theoretic. Τα πάνω όρια στην πληρότητα χώρου στο FM-index και το χρόνο ερωτήματος έχουν ληφθεί με την προϋπόθεση ότι το μέγεθος του αλφαβήτου εισόδου είναι σταθερό. Ο χρόνος αναζήτησης είναι $O(p + occ\ |\Sigma| \log^{1+\varepsilon} n)$ και ο χρόνος εμφάνισης ενός substring κειμένου είναι $O((l + \log^{1+\varepsilon} n)|\Sigma|)$. Στη συνέχεια, θα αναλυθεί ένα πιο "φιλικό προς το αλφάβητο" FM-index. Στόχος είναι να χρησιμοποιηθεί η τεχνική ενισχυμένης συμπίεσης και η δομή δεδομένων wavelet tree για να σχεδιαστεί μια έκδοση του FM-index που κλιμακώνεται καλά με το μέγεθος του αλφαβήτου. Δεδομένου ότι υπάρχουν αρκετά συμπιεσμένα full-text indexes που επιτυγχάνουν ενδιαφέρουσες αντισταθμίσεις χρόνου/χώρου, η προκύπτουσα δομή δεδομένων για να είναι ανταγωνιστική θα πρέπει να είναι εξαιρετικά απλή, να έχει τη μικρότερη πληρότητα χώρου και να μετρά τις εμφανίσεις αρκετά γρήγορα.

Εφεξής, έστω ότι το T[1,n] είναι το κείμενο που πρέπει να ευρετηριαστεί, να συμπιεστεί και να εξεταστεί. Το T προέρχεται από ένα αλφάβητο Σ μεγέθους |Σ|. Με T[i] δηλώνεται ο i-οστός χαρακτήρας του T, το T[i,n] δηλώνει το i-οστό suffix κείμενο και το T[1,i] δηλώνει το i-οστό prefix κείμενο. Γράφοντας |w| δηλώνεται το μήκος του string w. Ακολουθώντας μια καθιερωμένη πρακτική στη Θεωρία Πληροφορίας, χρησιμοποιώντας την έννοια της εμπειρικής εντροπίας μικραίνει ο χώρος που απαιτείται για την αποθήκευση ενός string T. Η βασική ιδιότητα της εμπειρικής εντροπίας είναι ότι ορίζεται προς τα δεξιά για κάθε string T και μπορεί να χρησιμοποιηθεί για τη μέτρηση της απόδοσης των αλγορίθμων συμπίεσης ως συνάρτηση της δομής string, χωρίς καμία παραδοχή για την πηγή εισόδου. Τυπικά, η εμπειρική εντροπία μηδενικής τάξης του T ορίζεται ως $H_0(T) = -\Sigma_i(n_i/n) \log(n_i/n)$, όπου n_i είναι ο αριθμός των εμφανίσεων του i-οστού χαρακτήρα αλφαβήτου στο T, $n = \Sigma_i n_i = |T|$ και όλοι οι αλγόριθμοι έχουν βάση το 2 (με 0log0 = 0). Ένα μήκους-k context w στο T είναι ένα από τα substrings του μήκους k. Δοθέντος w, ορίζεται με $\vec{w}_T$ το string που σχηματίζεται συνδυάζοντας όλα τα σύμβολα που ακολουθούν τις εμφανίσεις του w στο T, ληφθέντα από αριστερά προς τα δεξιά. Η εμπειρική εντροπία k-οστής τάξης του T ορίζεται ως:

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} |\vec{w}_T|\, H_0(\vec{w}_T). \qquad (1)$$

Η εμπειρική εντροπία k-οστής τάξης $H_k(T)$ είναι ένα κατώτερο όριο στο μέγεθος εξόδου οποιουδήποτε συμπιεστή που κωδικοποιεί κάθε χαρακτήρα του T χρησιμοποιώντας ένα μοναδικά αποκωδικοποιήσιμο κώδικα που εξαρτάται μόνο από τον ίδιο το χαρακτήρα και από τους k προηγούμενους χαρακτήρες του. Για κάθε k >= 0 τότε $H_k(T) <= \log|\Sigma|$.

Στο σημείο αυτό αξίζει να ξαναγίνει μια αναφορά στον BWT και στο FM-index συνοπτικά, αλλά ίσως και πιο διαφορετικά από ότι προηγουμένως. Ο BWT αποτελείται από τρία βασικά βήματα: (1) προσάρτηση στο τέλος του T ενός ειδικού χαρακτήρα # μικρότερο από οποιονδήποτε άλλο χαρακτήρα κειμένου, (2) σχηματισμός ενός εννοιολογικού πίνακα M_T του οποίου οι γραμμές είναι οι κυκλικές μετατοπίσεις του string T# ταξινομημένου με

λεξικογραφική σειρά και (3) κατασκευή του μετασχηματισμένου κειμένου T^{bwt} παίρνοντας την τελευταία στήλη του πίνακα M_T. Κάθε στήλη του M_T είναι μια παραλλαγή του T#. Συγκεκριμένα η πρώτη στήλη του M_T, που λέγεται F, δημιουργείται με λεξικογραφική ταξινόμηση των χαρακτήρων του T#. Εξαιτίας του ειδικού χαρακτήρα #, όταν ταξινομούνται οι σειρές του M_T ουσιαστικά ταξινομούνται τα suffixes του T. Ο πίνακας M_T έχει επίσης κι άλλες αξιοσημείωτες ιδιότητες που για να αναλυθούν θα πρέπει πρώτα να διευκρινιστούν τα εξής:

- Έστω C[·] ο πίνακας μήκους |Σ| έτσι ώστε το C[c] να περιέχει το συνολικό αριθμό των χαρακτήρων κειμένου οι οποίοι είναι αλφαβητικά μικρότεροι από c.
- Έστω Occ(c,q) ο αριθμός των εμφανίσεων του χαρακτήρα c στο prefix $T^{bwt}[1,q]$.
- Έστω LF(i) = $C[T^{bwt}[i]] + Occ(T^{bwt}[i],i)$.

Το LF(·) αντιπροσωπεύει τη Last-to-First σχεδίαση στήλης αφού ο χαρακτήρας $T^{bwt}[i]$, στην τελευταία στήλη του M_T, βρίσκεται στην πρώτη στήλη F στη θέση LF(i). Η σχεδίαση του LF(·) επιτρέπει τη σάρωση του κειμένου T προς τα πίσω. Δηλαδή, αν $T[k]=T^{bwt}[i]$ τότε $T[k-1]=T^{bwt}[LF(i)]$.

Το FM-index είναι ένα self-index που επιτρέπει την αποτελεσματική αναζήτηση των εμφανίσεων ενός αυθαίρετου προτύπου P[1,p] ως ένα substring του κειμένου T[1,n]. Ο αριθμός των εμφανίσεων προτύπου στο T υποδεικνύεται με occ. Ο όρος self-index εστιάζει στο γεγονός ότι το T δεν αποθηκεύεται ρητά αλλά μπορεί να εξαχθεί από το FM-index. Το FM-index αποτελείται από μια συμπιεσμένη αναπαράσταση του T^{bwt} μαζί με ορισμένες βοηθητικές πληροφορίες που καθιστούν δυνατό τον υπολογισμό σε O(1) χρόνο της τιμής Occ(c,q) για κάθε χαρακτήρα c και για κάθε q, 0 <= q <= n. Οι δυο βασικές διαδικασίες για τη λειτουργία του FM-index είναι: η καταμέτρηση του αριθμού εμφανίσεων προτύπων (εν συντομία get_rows) και ο εντοπισμός των θέσεών τους στο κείμενο T (εν συντομία get_position). Η διαδικασία καταμέτρησης επιστρέφει την τιμή occ, ενώ η διαδικασία εντοπισμού επιστρέφει occ ξεχωριστούς ακέραιους στο εύρος [1,n]. Μετά το τέλος της get_rows προκύπτει ότι: occ = Last – First + 1, όπου η παράμετρος First δείχνει στην πρώτη γραμμή του BWT πίνακα M_T και η παράμετρος Last δείχνει στην τελευταία γραμμή του M_T. Ο χρόνος λειτουργίας της get_rows κυριαρχείται από το κόστος των 2p υπολογισμών των τιμών Occ( ). Η get_position χρειάζεται $O(|\Sigma| (\log^2 n/\log \log n))$ χρόνο και παρατηρείται ότι η σήμανση μιας θέσης κάθε $\Theta(\log^2 n/\log \log n)$ απαιτεί $\Theta(n \log \log n/\log n)$ bits συνολικά. Συνδυάζοντας τα συμπεράσματα της get_position με αυτά της get_rows, προκύπτει το εξής:

Θεώρημα 1. Για κάθε string T[1, n] που παράγεται από ένα σταθερού μεγέθους αλφάβητο Σ, το FM-index μετρά τις εμφανίσεις οποιουδήποτε προτύπου P[1, p] εντός του T που χρειάζεται O(p) χρόνο. Η θέση κάθε εμφάνισης προτύπου διαρκεί $O(|\Sigma| \log^2 n/\log \log n)$ χρόνο. Το μέγεθος του FM-index οριοθετείται από $5nH_k(T) + o(n)$ bits, για οποιοδήποτε $k >= 0$.

Για να ανακτηθεί το περιεχόμενο του T[l,r], χρειάζεται μια διαδικασία με πολυπλοκότητα $O((l + \log^2 n/\log \log n) |\Sigma|)$. Επισημαίνεται η ύπαρξη μιας παραλλαγής του FM-index η οποία εκμεταλλεύεται την αλληλεπίδραση μεταξύ του BWT αλγορίθμου και του LZ78 αλγορίθμου και επιτυγχάνει την ίδια διαδικασία σε O(p+occ) χρόνο ερωτήματος, χρησιμοποιώντας $O(nH_k(T) \log^{\varepsilon} n) + o(n)$ bits αποθήκευσης. Γενικότερα, το κύριο μειονέκτημα του FM-index είναι ότι, κρυμμένες στον όρο o(n) του δεσμευμένου χώρου, υπάρχουν σταθερές οι οποίες εξαρτώνται εκθετικά από το μέγεθος του αλφαβήτου |Σ|.

Η έννοια της ενίσχυσης συμπίεσης έχει ανοίξει την πόρτα σε μια νέα προσέγγιση στη συμπίεση δεδομένων. Η βασική ιδέα είναι ότι κάποιος μπορεί να πάρει έναν αλγόριθμο του οποίου η απόδοση μπορεί να περιοριστεί υπό τους όρους της μηδενικής τάξης εντροπίας και να βρει, μέσω του ενισχυτή, ένα νέο συμπιεστή του οποίου η απόδοση μπορεί να περιοριστεί υπό τους όρους της k-οστής τάξης εντροπίας, ταυτόχρονα για όλα τα k. Για λόγους απλότητας αναφέρεται το επόμενο θεώρημα:

Θεώρημα 2. Έστω ότι A είναι ένας αλγόριθμος που συμπιέζει κάθε string s σε λιγότερο από $|s|H_0(s) + f(|s|)$ bits, όπου $f(\cdot)$ είναι μια μη φθίνουσα κοίλη συνάρτηση. Δοθέντος T[1, n] υπάρχει μια O(n) διαδικασία χρόνου που υπολογίζει μια διαμέριση $s_1, s_2, \ldots, s_z$ του T^{bwt} έτσι ώστε για οποιοδήποτε k >= 0, να προκύπτει:

$$\sum_{i=1}^{z} |\mathcal{A}(s_i)| \;\leq\; \sum_{i=1}^{z} \left(|s_i|H_0(s_i) + f(|s_i|)\right) \;\leq\; nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k).$$

Ο ενισχυτής επιτρέπει τη συμπίεση του T μέχρι την k-οστής τάξης εντροπία του χρησιμοποιώντας μόνο τον μηδενικής τάξης συμπιεστή A. Η παράμετρος k δεν είναι γνωστή ούτε στον A ούτε στον ενισχυτή, παίζει ρόλο μόνο στην ανάλυση πολυπλοκότητας χώρου. Επιπλέον, ο οριοθετημένος χώρος στο Θεώρημα 2 ισχύει ταυτόχρονα για κάθε k >= 0. Η μόνη πληροφορία που απαιτείται από τον ενισχυτή είναι η συνάρτηση f(n) έτσι ώστε $|s|H_0(s) + f(|s|)$ να είναι άνω όριο στο μέγεθος της εξόδου που παράγεται από τον A στην είσοδο s.

Δοθέντος μιας δυαδικής ακολουθίας S[1,m] και b ε {0,1}, αν το $Rank_b(S,i)$ υπολογίζει τον αριθμό των b στο S[1,i] και το $Select_b(S,i)$ υπολογίζει τη θέση του i-οστού b στο S[1,i], τότε μπορεί να ειπωθεί το παρακάτω θεώρημα:

Θεώρημα 3. Έστω S[1,m] μια δυαδική ακολουθία που περιέχει t εμφανίσεις του ψηφίου 1. Εκεί υπάρχει μια δομή δεδομένων (που ονομάζεται FID) που υποστηρίζει το $Rank_b(S,i)$ και το $Select_b(S,i)$ σε σταθερό χρόνο και χρησιμοποιεί $[\log(^m_t)] + O((m \log \log m)/\log m) = mH_0(S) + O((m \log \log m)/\log m)$ bits χώρο.

Αν, αντί για δυαδική ακολουθία, υπάρχει μια ακολουθία W[1,w] σε ένα αυθαίρετο αλφάβητο Σ, μια συμπιεσμένη και ευρετηριοποιήσιμη αναπαράσταση του W παρέχεται από ένα wavelet tree που είναι μια έξυπνη γενίκευση της δομής δεδομένων FID.

Θεώρημα 4. Έστω W[1, w] που δηλώνει ένα string ενός αυθαίρετου αλφαβήτου Σ. Το wavelet tree που δημιουργείται στο W χρησιμοποιεί $wH_0(W) + O(\log|\Sigma|$ (w log logw)/logw bits αποθήκευσης και υποστηρίζει σε $O(\log|\Sigma|)$ χρόνο τις παρακάτω λειτουργίες:

- Δοθέντος q, 1 <= q <= w, την ανάκτηση του χαρακτήρα W[q].
- Δοθέντος c ε Σ και q, 1 <= q <= w, τον υπολογισμό του αριθμού των εμφανίσεων $Occ_w(c, q)$ του c στο W[1, q].

Τώρα υπάρχουν όλα τα εργαλεία που χρειάζονται για τη δημιουργία μιας εκδοχής του FM-index που ταιριάζει καλά με το μέγεθος του αλφαβήτου. Για να δημιουργηθεί το FM-index πρέπει να λυθούν δυο προβλήματα: α) η συμπίεση του T^{bwt} μέχρι και $H_k(T)$ και β) ο υπολογισμός του Occ(c, q) σε χρόνο ανεξάρτητο του n. Χρησιμοποιώντας την τεχνική ενίσχυσης μετατρέπεται το πρόβλημα α) στο πρόβλημα συμπίεσης των strings $s_1, s_2, \ldots, s_z$ έως τη μηδενικής τάξης τους εντροπία και χρησιμοποιώντας το wavelet tree δημιουργείται μια συμπιεσμένη (μέχρι H_0) και ευρετηριοποιήσιμη αναπαράσταση κάθε s_i επιλύοντας έτσι ταυτόχρονα τα προβλήματα α) και b). Συνολικά, το "φιλικό προς το αλφάβητο" FM-index χρειάζεται $O(p \log|\Sigma|)$ χρόνο για να μετρήσει τις εμφανίσεις ενός προτύπου P[1, p] και $O(\log|\Sigma|$ $(\log^2 n/\log \log n)$ χρόνο για να ανακτήσει τη θέση της κάθε εμφάνισης (προκύπτουν κάνοντας χρήση και των Θεωρημάτων 3 και 4). Χρησιμοποιώντας τα Θεωρήματα 2 και 3 προκύπτει ότι η συνολική πληρότητα χώρου οριοθετείται από:

$$nH_k(T) + O\left(n\frac{\log|\Sigma|\log\log n}{\log(n/|\Sigma|^k)}\right) + O\left(|\Sigma|^{k+1}\log n\right). \qquad (2)$$

Το παρών κείμενο ενδιαφέρεται μόνο για τον περιορισμό της πληρότητας χώρου σε όρους του H_k μόνο για k <= $a\log_{|\Sigma|}n$ για κάποια a<1. Σε αυτή την περίπτωση $|\Sigma|^k <= n^a$ και το (2) γίνεται:

$$nH_k(T) + O(\log|\Sigma|(n\log\log n)/\log n)\,. \qquad (3)$$

Έτσι προκύπτει το εξής θεώρημα:

Θεώρημα 5. Η δομή δεδομένων που περιγράφτηκε παραπάνω ευρετηριάζει ένα string T[1, n] από ένα αυθαίρετο αλφάβητο |Σ|, χρησιμοποιώντας έναν αποθηκευτικό χώρο οριοθετημένο από

$nH_k(T) + O(\log|\Sigma|\ (n \log\log n)/\log n)$

bits για οποιοδήποτε k <= a $\log_{|\Sigma|}n$ και 0 < a < 1. Μπορεί να μετρηθεί ο αριθμός των εμφανίσεων ενός προτύπου P[1,p] στο T σε O(p log|Σ|) χρόνο, να εντοπιστεί κάθε εμφάνιση σε $O(\log|\Sigma|\ (\log^2 n / \log\log n))$ χρόνο και να εμφανιστεί ένα substring κειμένου μήκους l σε $O((l + \log^2 n / \log\log n) \log|\Sigma|)$ χρόνο. Μάλιστα, δεν γίνεται κάποια άλλη πιο εξελιγμένη δομή δεδομένων να επιτύχει ένα $nH_k(T) + o(n)$ διάστημα χωρίς κάποιο περιορισμό στο μέγεθος του αλφαβήτου ή στο μήκος του πλαισίου.

Επίλογος

Παρουσιάστηκε ο μετασχηματισμός Burrows-Wheeler (BWT) καθώς και κάποιες προτάσεις πάνω σε αυτόν για περαιτέρω βελτίωση. Από όλα τα παραπάνω είναι σημαντικό να κρατηθούν τα εξής. Ο BWT έχει προκαλέσει μεγάλη πρόοδο στον τομέα του, αφού ο ταξινομημένος πίνακας που δημιουργεί συμπιέζεται πολύ καλά (συγκριτικά και με άλλα αξιόλογα στατιστικά μοντέλα), όμως έχει και κάποια μειονεκτήματα, όπως για παράδειγμα ότι δεν είναι online (δηλαδή θα πρέπει να έχει τη δυνατότητα να επεξεργαστεί ένα μεγάλο μέρος της εισόδου πριν παραχθεί ένα ενιαίο κομμάτι εξόδου). Αυτό το ζήτημα έχει μελετηθεί ερευνητικά, αλλά όχι ακόμα πλήρως. Γενικότερα, η επιπλέον μελέτη πάνω σε αυτόν τον αλγόριθμο μπορεί να επιφέρει ακόμη καλύτερα αποτελέσματα στον τομέα του.

Στην Ενότητα 2 παρουσιάστηκε ένας αλγόριθμος για τον υπολογισμό του BWT ενός κειμένου χρησιμοποιώντας πολύ λίγο χώρο αλλά σε μεγάλη ταχύτητα και στη θεωρία και στην πράξη. Επίσης αναφέρθηκαν και ορισμένοι άλλοι μεταγενέστεροι αλγόριθμοι που εκτελούν παρόμοια διαδικασία, αλλά σε διαφορετικό χώρο και χρόνο.

Στην Ενότητα 3 έγινε αναφορά στους πρώτους γρήγορους βέλτιστους PRAM αλγορίθμους για προβλήματα BW συμπίεσης και αποσυμπίεσης. Αποδείχθηκε ότι οι στοιχειώδεις παράλληλες ρουτίνες, όπως τα prefix sums, μπορεί να αποδειχθούν πολύ χρήσιμες για έρευνες στον τομέα.

Στην Ενότητα 4 αναφέρθηκε η πρώτη παράλληλη κατασκευή του BWT που δουλεύει σε συμπαγή χώρο, ο οποίος είναι, O(n lgσ) bits για μια ακολουθία μήκους n στο αλφάβητο [1..σ]. Μια ενδιαφέρουσα πρόκληση θα ήταν να δημιουργηθούν σε παράλληλο και σε συμπαγή χώρο τα άλλα μέρη ενός συμπιεσμένου suffix tree: η τοπολογία δέντρου και το (παραμορφωμένο) μεγαλύτερο κοινό prefix array. Αυτά απαιτούν O(n) παραπάνω bits και μπορούν να δημιουργηθούν σε O(n) ακολουθιακό χρόνο και O(n lgσ) bits από τον BWT.

Στην Ενότητα 5 παρουσιάστηκε μια πρακτική δομή δεδομένων εμπνευσμένη από το FM-index, το οποίο αφαιρεί τη μεγάλη εξάρτησή του από το αλφάβητο μεγέθους σ. Η βασική ιδέα είναι να συμπιεστεί με Huffman το κείμενο πριν εφαρμοστεί ο μετασχηματισμός Burrows-Wheeler σε αυτό. Η δομή αυτή έχει το πλεονέκτημα ότι (σχεδόν) δεν εξαρτάται από το μέγεθος του αλφαβήτου και επίσης έχει καλύτερες πολυπλοκότητες από άλλα ευρετήρια για κάποιες λειτουργίες. Μελλοντικά, θα μπορούσε να συνεχιστεί η έρευνα σχετικά με τις παραλλαγές μεθόδων κωδικοποίησης, των οποίων οι ιδιότητες μπορούν να χρησιμοποιηθούν για να περιοριστεί το μέγεθος του ευρετηρίου.

Στην Ενότητα 6 συνδυάζοντας μια υπάρχουσα τεχνική ενισχυμένης συμπίεσης με τη δομή δεδομένων wavelet tree, σχεδιάζεται μια παραλλαγή του FM-index που προσαρμόζεται καλά στο μέγεθος της εισόδου του αλφαβήτου Σ. Ο χρόνος που απαιτείται για: α) τη μέτρηση των εμφανίσεων ενός αυθαίρετου προτύπου P σε ένα string T, β) τον εντοπισμό κάθε εμφάνισης προτύπου και γ) την αναφορά ενός substring κειμένου μήκους l, είναι πολύ αξιόλογος. Σε ενδεχόμενες μελλοντικές έρευνες θα άξιζε σίγουρα να χρησιμοποιηθεί και η συγκεκριμένη μελέτη.

Βιβλιογραφία

[1] Parallel Computation of the Burrows Wheeler Transform in Compact Space, J. Fuentes-Sepulveda, G. Navarro, Y. Nekrich

[2] A simple alphabet-independent FM-index, S. Grabowski, G. Navarro, R. Przywarski, A. Salinger, V. Makinen

[3] An Alphabet-Friendly FM-Index, P. Ferragina, G. Manzini, G. Navarro

[4] A Block-sorting Lossless Data Compression Algorithm, M. Burrows, D.J. Wheeler

[5] Fast BWT in small space by blockwise suffix sorting, J. Karkkainen

[6] Parallel algorithms for Burrows-Wheeler compression and decompression, J.A. Edwards, U. Vishkin

[7] A space and time efficient algorithm for constructing compressed suffix arrays, W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung, S.M. Yiu

[8] A linear-time Burrows-Wheeler transform using induced sorting, D. Okanohara, K. Sadakane

[9] Space-efficient construction of compressed indexes in deterministic linear time, J.I. Munro, G. Navarro, Y. Nekrich

[10] Μέθοδος συμπίεσης συμβολοσειρών με σχεδόν βέλτιστο χώρο και βέλτιστο χρόνο ανάκτησης υποσυμβολοσειράς, Καλλίκης Μικές

[11] www.wikipedia.org