Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Υλοποίηση κατασκευής δέντρου επιθεμάτων σε Hadoop Mapreduce

Διπλωματική Εργασία

του

ΑΛΕΞΑΝΔΡΟΥ ΚΩΝΣΤΑΝΤΙΝΑΚΗ - ΚΑΡΜΗ

Επιβλέπων: Τιμολέων Σελλής
Καθηγητής Ε.Μ.Π.

Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων
Αθήνα, Ιούλιος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων

Υλοποίηση κατασκευής δέντρου επιθεμάτων σε Hadoop Mapreduce

Διπλωματική Εργασία

του

ΑΛΕΞΑΝΔΡΟΥ ΚΩΝΣΤΑΝΤΙΝΑΚΗ - ΚΑΡΜΗ

Επιβλέπων: Τιμολέων Σελλής
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τη 12η Ιουλίου 2010.

.....	.....	.....
Τιμολέων Σελλής	Νεκτάριος Κοζύρης	Δημήτριος Φωτάκης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

.....

Αλέξανδρος Κωνσταντινάκης - Κάρμης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Συστημάτων Βάσεων Γνώσεων και Δεδομένων

Περίληψη

Τα δέντρα επιθεμάτων χρησιμοποιούνται ως ευρετήρια για ακολουθίες βιολογικών δεδομένων. Η χρήση τους είναι απαραίτητη σε αλγορίθμους αναζήτησης που χρησιμοποιούνται στη βιολογία. Τα τελευταία χρόνια ο όγκος των βιολογικών δεδομένων αυξάνεται συνεχώς λόγω των εξελίξεων της επιστήμης. Στο πλαίσιο της διπλωματικής μελετήθηκαν οι κυριότεροι αλγόριθμοι για κατασκευή δέντρων επιθεμάτων στη μνήμη, στον σκληρό δίσκο και σε παράλληλους υπολογιστές. Έπειτα, υλοποιήθηκαν αλγόριθμοι παράλληλης κατασκευής δέντρων επιθεμάτων με χρήση της τεχνολογίας Hadoop MapReduce έχοντας ως βάση τη μέθοδο Trellis, της πιο αποτελεσματικής τεχνικής κατασκευής δέντρων επιθεμάτων στο σκληρό δίσκο. Τέλος πραγματοποιήθηκαν μετρήσεις, οι οποίες δείχνουν τη συμπεριφορά των αλγορίθμων σε παράλληλη εκτέλεση.

Λέξεις Κλειδιά

DNA, ευρετήρια για ακολουθίες, βιολογικά δεδομένα, δέντρα επιθεμάτων, κύρια μνήμη, Hadoop MapReduce

Abstract

Suffix trees are a form of index widely used for sequences of biological data. Their use is crucial for search algorithms used in biology. In recent years science has produced growing numbers of biological data. The aim of this diploma thesis was to study the main algorithms for constructing suffix trees in memory, in the hard drive and in parallel systems. Secondly, to implement algorithms for parallel construction of suffix trees using Hadoop MapReduce and based on the technique of Trellis, the most efficient suffix tree construction method in the hard drive today. Finally, experiments were conducted to evaluate the behaviour of these algorithms in parallel execution.

Keywords

DNA, sequence index, biological data, suffix trees, main memory, Hadoop MapReduce

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Κεφάλαιο 1

Εισαγωγή

1.1 Αντικείμενο της διπλωματικής εργασίας

Το 2000 αποκωδικοποιήθηκε για πρώτη φορά το γονιδίωμα του ανθρώπου. Η επιστημονική αυτή επιτυχία σήμανε την αρχή μιας νέας εποχής, με τον κλάδο της βιολογίας να υπόσχεται ότι το μέλλον της ανθρωπότητας βρίσκεται στην κατανόηση της λειτουργίας του ανθρώπινου DNA. Εξελίξεις στην επιστήμη κατέστησαν δυνατό να διαβαστεί και να αποθηκευτεί σε υπολογιστές μεγάλος όγκος βιολογικών δεδομένων υπό τη μορφή ακολουθιών. Η ανάγκη για στατιστική ανάλυση και αποδοτική επεξεργασία αυτών των δεδομένων γέννησε τον κλάδο της Υπολογιστικής Βιολογίας. Οι αλγόριθμοι για βιολογικά δεδομένα εκτελούνται πολύ αποδοτικότερα όταν γίνεται χρήση ευρετηρίου. Το σημαντικότερο από τα ευρετήρια που έχουν προταθεί είναι το δέντρο επιθεμάτων (suffix tree). Καθώς το προς ευρετηριοποίηση βιολογικό υλικό γίνεται ολοένα και περισσότερο, ξεπερνώντας την χωρητικότητα της κεντρικής μνήμης των υπολογιστών, δημιουργήθηκε η ανάγκη αποδοτικής κατασκευής των δέντρων επιθεμάτων στον σκληρό δίσκο. Στο πλαίσιο αυτής της διπλωματικής εργασίας μελετώνται οι αλγόριθμοι κατασκευής δέντρων επιθεμάτων στη μνήμη, στον σκληρό δίσκο και παράλληλα. Ακόμα, γίνεται μια προσπάθεια παραλληλοποίησης του αποδοτικότερου από τους αλγορίθμους κατασκευής στον σκληρό δίσκο μέσω του Hadoop MapReduce. Το Hadoop MapReduce είναι μια πρόσφατη τεχνολογία που επιχειρεί να καταστήσει προσιτή τόσο την ανάπτυξη παράλληλων εφαρμογών, όσο και την εκτέλεσή τους σε cluster απλών υπολογιστών.

1.2 Συνεισφορά

Αρχικά μελετήθηκαν αλγόριθμοι κατασκευής δέντρων επιθεμάτων. Ιδιαίτερο βάρος δόθηκε στη μελέτη του αλγόριθμου του Ukkonen που κατασκευάζει σε γραμμικό χρόνο και χώρο το δέντρο επιθεμάτων μιας συμβολοσειράς, διαβάζοντας την από αριστερά προς τα δεξιά. Ο αλγόριθμος του Ukkonen κατασκευάζει το δέντρο σε n φάσεις. Κάθε φάση περιλαμβάνει μια σειρά επεκτάσεων των ακμών του υπάρχοντος δέντρου έτσι ώστε να προστεθεί ο n-οστός χαρακτήρα. Μια σειρά έξυπνων παρατηρήσεων, όπως η χρήση συνδέσμων επιθέματος, επιταχύνουν την εξεύρεση των ακμών που χρειάζονται επέκταση σε κάθε φάση, προσδίδοντας

γραμμικότητα στον αλγόριθμο. Όμως ο αλγόριθμος του Ukkonen αποτυγχάνει για μεγάλες εισόδους, καθώς το δέντρο επιθεμάτων χρειάζεται χώρο πολλαπλάσιο της ακολουθίας που περιγράφει, με αποτέλεσμα να ξεπερνά σε μέγεθος την χωρητικότητα της κύριας μνήμης των υπολογιστών. Έτσι στη συνέχεια μελετήθηκαν οι τεχνικές που έχουν προταθεί για αντιμετώπιση αυτού του προβλήματος και κατασκευάζουν το δέντρο επιθεμάτων στον σκληρό δίσκο. Η αποδοτικότερη αυτών των τεχνικών ονομάζεται Trellis. Η προσέγγιση του Trellis είναι να Χωρίσει την είσοδο σε κομμάτια σταθερού μεγέθους, κατασκευάζοντας για το κάθε ένα το αντίστοιχο δέντρο επιθεμάτων χρησιμοποιώντας τον αλγόριθμο του Ukkonen. Στη συνέχεια κατακερματίζει το κάθε δέντρο σε προθεματικά υποδέντρα με βάση προθέματα μεταβλητού μήκους που έχει προϋπολογίσει. Στο τέλος συγχωνεύει τα υποδέντρα που φέρουν το ίδιο αρχικό πρόθεμα και δίνει ως έξοδο προθεματικά δέντρα που περιέχουν συνολικά την ίδια πληροφορία με το πλήρες επιθεματικό δέντρο, χωρίς όμως το καθένα ξεχωριστά να ξεπερνά σε απαιτήσεις την κεντρική μνήμη του υπολογιστή. Τα προθέματα μεταβλητού μήκους υπολογίζονται έτσι ώστε το καθένα να έχει συχνότητα λιγότερη από ένα κατώφλι t στο αρχείο εισόδου. Ο ίδιος αριθμός χρησιμοποιείται ως μήκος κομματιού για το χωρισμό της εισόδου. Το κατώφλι είναι αριθμός υπολογισμένος έτσι ώστε ένα δέντρο επιθεμάτων με t φύλλα να χωράει σε ένα δοσμένο μέγεθος κεντρικής μνήμης. Έτσι το Trellis διασφαλίζει ότι όλες οι εργασίες συμβαίνουν αποδοτικά στη μνήμη. Όπως διαπιστώνουμε από την παραπάνω περιγραφή, η επεξεργασία οποιουδήποτε κομματιού της ακολουθίας εισόδου είναι ανεξάρτητη από την επεξεργασία των υπόλοιπων. Επίσης, η συγχώνευση των υποδέντρων με το ίδιο πρόθεμα είναι ανεξάρτητη από όλα τα άλλα υποδέντρα. Έτσι φαίνεται ότι η διαδικασία μπορεί να παραλληλοποιηθεί, αναθέτοντας τις επιμέρους ανεξάρτητες εργασίες σε διαφορετικούς υπολογιστές.

Η μελέτη συνεχίστηκε με το Hadoop MapReduce που χρησιμοποιήθηκε για την υλοποίηση του προγράμματος. Το Hadoop MapReduce είναι μια νέα τεχνολογία που απλοποιεί την κατασκευή και εκτέλεση παράλληλων προγραμμάτων. Πρόκειται για ένα framework γραμμένο σε Java το οποίο υποστηρίζει εφαρμογές παράλληλης επεξεργασίας μεγάλου όγκου δεδομένων και είναι ελεύθερη υλοποίηση του Google MapReduce. Το Hadoop προσφέρει ένα επίπεδο αφαίρεσης στη διαδικασία ανάπτυξης παράλληλων προγραμμάτων, αναλαμβάνοντας να διαχειριστεί το διαμοιρασμό των δεδομένων, την συγκέντρωση των αποτελεσμάτων, τις πιθανές αποτυχίες κόμβων και άλλα θέματα που κανονικά θα χρειαζόταν ο προγραμματιστής να υλοποιήσει. Ο προγραμματιστής αρκεί να γράψει πρόγραμμα σε Java χρησιμοποιώντας βιβλιοθήκες του Hadoop και υλοποιώντας τον αλγόριθμο του σύμφωνα με τη λογική των συναρτήσεων Map και Reduce, έτσι ώστε να εκμεταλλευτεί την παράλληλη επεξεργασία που προσφέρει το cluster. Συγκεκριμένα ένα πρόγραμμα MapReduce αποτελείται από δύο συναρτήσεις την map και την reduce. Η κάθε μία εκτελείται παράλληλα από το cluster, αλλά οι δύο φάσεις εκτελούνται σειριακά μεταξύ τους, με τα αποτελέσματα της map να αποτελούν είσοδο στη reduce. Η συνάρτηση map πρέπει να είναι έτσι προγραμματισμένη ώστε να δέχεται ένα κομμάτι της εισόδου και να δίνει ως έξοδο ζευγάρια (κλειδί, τιμή). Οι έξοδοι όλων των κόμβων του cluster για τη συνάρτηση map ομαδοποιούνται με βάση το κλειδί και γίνεται μία κλήση της συνάρτησης reduce για κάθε μοναδικό κλειδί, δίνοντας ως είσοδο όλες τις τιμές που βρίσκονται σε ζευγάρι με το συγκεκριμένο κλειδί. Τα αποτελέσματα αποθηκεύονται στο

δίσκο και αποτελούν την τελικό αποτέλεσμα.

Στη συνέχεια υλοποιήθηκε παράλληλο πρόγραμμα κατασκευής δέντρων επιθεμάτων βασισμένο στην κεντρική ιδέα του Trellis και στο προγραμματιστικό μοντέλο του MapReduce. Αναλυτικά, η τεχνική του Trellis υλοποιήθηκε κατά MapReduce με δύο εργασίες που εκτελούνται σειριακά μεταξύ τους. Η πρώτη υπολογίζει τα προθέματα μεταβλητού μήκους, χωρίζοντας την είσοδο σε κομμάτια και μετρώντας σε κάθε κομμάτι τη συχνότητα μιας λίστας προθεμάτων. Στη συνέχεια στη φάση reduce, αθροίζονται οι συχνότητες του κάθε προθέματος έτσι ώστε να βρεθεί η συχνότητά του συνολικά στην είσοδο. Τα προθέματα που έχουν συχνότητα λιγότερη από το κατώφλι κρατιούνται ως τελικά, ενώ τα υπόλοιπα επιμηκύνονται κατά ένα χαρακτήρα μία φορά για κάθε γράμμα του αλφάβητου και αποτελούν είσοδο στην επόμενη επανάληψη της εργασίας. Κατόπιν υλοποιήθηκαν τα στάδια κατασκευής προθεματικών υποδέντρων και συγχώνευσης τους σε προθεματικά δέντρα του Trellis. Τα στάδια αυτά προγραμματίστηκαν σε μια εργασία MapReduce, με τη συνάρτηση map να δέχεται ως είσοδο ένα κομμάτι της εισόδου, να κατασκευάζει για αυτό το κομμάτι τα προθεματικά υποδέντρα και να δίνει έξοδο ζευγάρια (πρόθεμα, προθεματικό υποδέντρο). Το Hadoop ομαδοποιεί τα ζευγάρια κατά πρόθεμα και η συνάρτηση reduce αναλαμβάνει να συγχωνεύσει όλα τα υποδέντρα που μοιράζονται το ίδιο πρόθεμα. Αξίζει να σημειωθεί ότι για τη συνάρτηση map υλοποιήθηκαν 3 εκδοχές με βάση διαφορετικούς αλγορίθμους. Συγκεκριμένα υλοποιήθηκε ένας απλός αλγόριθμος που σαρώνει μια φορά το κείμενο για κάθε πρόθεμα και κατασκευάζει απευθείας το προθεματικό υποδέντρο. Επίσης υλοποιήθηκε ο αλγόριθμος Ukkonen σε Java και αποτέλεσε βάση για μια δεύτερη εκδοχή. Τέλος μια υλοποίηση του αλγορίθμου του Ukkonen από βιβλιοθήκη της java με αλγορίθμους βιολογικών δεδομένων χρησιμοποιήθηκε σε μια τρίτη εκδοχή του προγράμματος. Στις εκδόσεις που χρησιμοποιήθηκε ο αλγόριθμος του Ukkonen, η τεχνική κατασκευής των προθεματικών υποδέντρων συνίσταται στην κατασκευή ολόκληρου το δέντρου επιθεμάτων για το κομμάτι που είναι είσοδος στη map και στη συνέχεια στην εξαγωγή των προθεματικών υποδέντρων μέσω μιας διαδικασίας ακριβούς ταιριάσματος του κάθε προθέματος στο δέντρο και εξαγωγή του υποδέντρου που ορίζεται από το τελικό σημείο των συγκρίσεων. Το πλεονέκτημα του απλού, μη γραμμικού αλγορίθμου είναι ότι κατασκευάζοντας απευθείας τα προθεματικά υποδέντρα έχει μικρότερες απαιτήσεις σε μνήμη και άρα η ακολουθία εισόδου μπορεί να χωριστεί σε λιγότερα κομμάτια και να γίνει καλύτερη χρήση ενός μικρού cluster με ταυτόχρονη μείωση του αριθμού των ενδιάμεσων εγγραφών. Αντίθετα, ο γραμμικός αλγόριθμος του Ukkonen απαιτεί τεμαχισμό της εισόδου σε μικρά κομμάτια, χαρακτηριστικό που μας αναγκάζει να δημιουργήσουμε πολλές μικρές εργασίες με την ακόλουθη αύξηση και στον αριθμό των ενδιάμεσων εγγραφών. Η διαδικασία συγχώνευσης επιχειρεί να προσθέσει όλες τις ακμές όλων των υποδέντρων που είναι είσοδος στη συνάρτηση reduce σε ένα κύριο προθεματικό δέντρο που είναι και η τελική έξοδος της συνάρτησης.

Η πειραματική αξιολόγηση στόχευσε στη σύγκριση της απόδοσης των παραπάνω εκδοχών τόσο για αυξανόμενου μεγέθους ακολουθίες εισόδου όσο και για αυξανόμενο πλήθος υπολογιστών στο cluster. Η παραλληλοποίηση αυτού του αλγορίθμου θα έπρεπε να δώσει απογοητευτικά αποτελέσματα λόγω της ανάγκης παρουσίας ολόκληρων των ακολουθίων εισόδου σε κάθε κόμβο του cluster για τη διαδικασία της συγχώνευσης. Η ανάγκη του αλγορίθμου για

μεταφορά πολλών δεδομένων είναι ένας σημαντικός παράγοντας καθυστέρησης, δεν δρα όμως αναστολτικά στο να βελτιωθούν τα χαρακτηριστικά της εκτέλεσης σε σχέση με το Trellis. Συνολικά οι μετρήσεις μας δείχνουν ότι με χρήση ενός μικρού cluster 10 υπολογιστών η διαδικασία ευρετηριοποίησης παρουσιάζει αύξηση απόδοσης, και κυρίως έχει τη δυνατότητα να ευρετηριοποιήσει αποδοτικά μεγαλύτερες ακολουθίες από ότι το Trellis.
Συνοπτικά, η συνειφορά της διπλωματικής αφορά στα παρακάτω θέματα:α

- Μελέτη τεχνικών αποδοτικής κατασκευής δέντρων επιθεμάτων
- Υλοποίηση προγράμματος παράλληλης κατασκευής δέντρων επιθεμάτων
- Πειραματική αξιολόγηση υλοποιήσεων

1.3 Οργάνωση

Το κείμενο της διπλωματικής αποτελείται από 6 κεφάλαια. Το παρόν κεφάλαιο είναι το πρώτο και σε αυτό γίνεται αναφορά στο αντικείμενο της διπλωματικής, στη συνεισφορά της εργασίας στον τομέα έρευνας που αφορά το αντικείμενο και στον τρόπο οργάνωσης του υπόλοιπου κειμένου. Στο δεύτερο κεφάλαιο επιχειρείται η παροχή του απαραίτητου υποβάθρου που χρειάζεται ο αναγνώστης για να κατανοήσει το σύνολο της εργασίας. Συγκεκριμένα, παρουσιάζονται αρχικά κάποιες βασικές έννοιες τις βιολογίας. Στη συνέχεια παρουσιάζονται διάφορα ευρετήρια και αλγόριθμοι προσεγγιστικού ταιριάσματος που έχουν προταθεί για τα βιολογικά δεδομένα, με έμφαση στα δέντρα επιθεμάτων και τους αλγορίθμους προσεγγιστικού ταιριάσματος που τα χρησιμοποιούν. Έπειτα αναλύονται διάφοροι αλγόριθμοι κατασκευής δέντρων επιθεμάτων που δρουν στη μνήμη, στον σκληρό δίσκο και σε παράλληλα συστήματα. Τέλος, ο αναγνώστης εισάγεται στην τεχνολογία Hadoop MapReduce. Στο τρίτο κεφάλαιο γίνεται ανάλυση του αλγορίθμου Trellis, του αποδοτικότερου αλγορίθμου κατασκευής δέντρων επιθεμάτων στον σκληρό δίσκο. Στο τέταρτο κεφάλαιο παρουσιάζεται η συνεισφορά της εργασίας αυτής σε επίπεδο υλοποιήσεων. Συγκεκριμένα, παρουσιάζεται αρχικά ο τρόπος με τον οποίο η βασική ιδέα του Trellis προσαρμόζεται στο προγραμματιστικό μοντέλο του MapReduce. Στη συνέχεια παρουσιάζεται η υλοποίηση για το στάδιο κατασκευής προθεμάτων, οι διαφορετικές υλοποιήσεις για το στάδιο κατασκευής προθεματικών υποδέντρων και η υλοποίηση του σταδίου συγχώνευσης υποδέντρων σε δέντρα. Στο πέμπτο κεφάλαιο παρουσιάζονται οι μετρήσεις που έγιναν για τις παραπάνω υλοποιήσεις με σκοπό την κατανόηση της συμπεριφοράς τους σε πραγματικές συνθήκες και μεγάλες εισόδους. Ακολουθούν στο έκτο κεφάλαιο μια σύνοψη των συμπερασμάτων της διπλωματικής και προτάσεις για μελλοντικές εργασίες. Στο τέλος βρίσκεται η βιβλιογραφία.

Κεφάλαιο 2

Υπόβαθρο

Σε αυτό το εισαγωγικό κεφάλαιο θα ασχοληθούμε αρχικά με την έννοια των βιολογικών δεδομένων και τις τεχνολογίες που έχουν αναπτυχθεί για να έχουμε αποδοτική αναζήτηση μέσα σε αυτά. Στη συνέχεια θα αναφερθούμε στην ιστορική εξέλιξη των μεθόδων κατασκευής ενός συγκεκριμένου είδους ευρετηρίου βιολογικών δεδομένων που έχει αποδειχθεί πως επιταχύνει ιδιαίτερα τη διαδικασία αναζήτησης. Τέλος, θα παρουσιαστεί η τεχνολογία Hadoop MapReduce που μας επιτρέπει να κατασκευάζουμε και να εκτελούμε παράλληλα προγράμματα.

2.1 Βιολογικά Δεδομένα και Αλγόριθμοι Ταιριάσματος

Η κληρονομικότητα των χαρακτηριστικών ανθρώπων και ζώων αλλά και η ποικιλομορφία μεταξύ ατόμων του ιδίου είδους ήταν πάντα γνωστή στον άνθρωπο. Έτσι γνώριζε ότι αν διασταυρώσει τα πιο παραγωγικά του ζώα, τότε ολόκληρη η επόμενη γενιά θα είναι πιο παραγωγική ή ότι γάμοι μεταξύ μελών της ίδιας οικογένειας θα δώσουν ασθενικά παιδιά. Ο μηχανισμός που κρύβεται πίσω από την κληρονομικότητα χαρακτηριστικών δεν είναι άλλος από το γενετικό υλικό. Το DNA είχε ανακαλυφθεί από τον 19ο αιώνα, όμως η δομή του έγινε κατανοητή το 1953 από τους Watson και Crick. Σύμφωνα με την εργασία τους, το DNA αποτελείται από δύο αλυσίδες συμπληρωματικές μεταξύ τους. Η κάθε αλυσίδα αποτελείται από μονάδες που ονομάζονται νουκλεοτίδια. Τα νουκλεοτίδια είναι αζωτούχες βάσεις και διακρίνονται 4 είδη: αδενίνη (A), θυμίνη (T), κυτοσίνη (C) και γουανίνη (G). Οι βάσεις απαντώνται σε συμπληρωματικά ζευγάρια που βρίσκονται αντικριστά στις ίδιες θέσεις των δύο αλυσίδων. Έτσι η αδενίνη (A) βρίσκεται πάντα απέναντι από τη θυμίνη (T) και η κυτοσίνη (C) απέναντι από τη γουανίνη (G). Το DNA είναι υπεύθυνο και για την παραγωγή του σημαντικότερου συστατικού ενός κυττάρου, των πρωτεϊνών. Αντίστοιχα με το DNA, οι πρωτεΐνες αποτελούνται και αυτές από 20 διαφορετικές διακριτές μονάδες που ονομάζονται αμινοξέα. Η μελέτη του γενετικού υλικού δίνει αποτελέσματα στην ιατρική και τη βιολογία, βοηθώντας στη μελέτη της εξέλιξης των οργανισμών και την αντιμετώπιση γενετικών ασθενειών. Τα συμπεράσματα στα οποία είχαν καταλήξει οι πρόγονοί μας σήμερα συστηματοποιούνται, ενώ διαρκώς ανακαλύπτονται και νέα μυστικά του γενετικού κώδικα. Για να

γίνουν όλα αυτά δυνατά, είναι απαραίτητο να γίνονται αναζητήσεις στα βιολογικά δεδομένα με αποτέλεσμα ομολογίες μεταξύ γενετικού υλικού, δηλαδή ομοιότητες στη δομή και λειτουργία δύο βιομορίων που προέρχονται από διαφορετικό άτομο. Για παράδειγμα μπορεί έτσι να μελετηθεί ποιο γονίδιο πέρασε από δύο υγιείς γονείς στο παιδί τους, με αποτέλεσμα το παιδί να κληρονομήσει κάποια σοβαρή γενετική ασθένεια. Η επιστήμη της πληροφορικής είναι βοηθός στη διαδικασία αυτή. Τόσο το DNA, όσο και οι πρωτεΐνες μπορούν να αναπαρασταθούν ως συμβολοσειρές. Η αναζήτηση ομολογιών μπορεί να γίνει με τη χρήση κάποιου αλγορίθμου ταιριάσματος ακολουθιών. Ο όγκος των βιολογικών δεδομένων είναι τέτοιος που έθεσε νέες απαιτήσεις σε ήδη γνωστές τεχνολογίες της επιστήμης της πληροφορικής. Όπως θα δούμε, η ανάγκη για αποδοτικές σε χρόνο αναζητήσεις σε μεγάλο όγκο βιολογικών δεδομένων οδήγησε στην ανάπτυξη νέων δομών ευρετηρίων και τεχνικών ταιριάσματος ακολουθιών, εξειδικευμένων στα βιολογικά δεδομένα.

2.1.1 Δομές δεδομένων και αλγόριθμοι προσεγγιστικού ταιριάσματος

Στην ενότητα αυτή παρουσιάζονται δομές δεδομένων που έχουν προταθεί για χρήση ως ευρετήρια συμβολοσειρών, καθώς επίσης και τεχνικές που εφαρμόζονται από αλγορίθμους για προσεγγιστικό ταίριασμα συμβολοσειρών και χρησιμοποιούν τα παραπάνω ευρετήρια. Περισσότερες λεπτομέρειες μπορούν να βρεθούν στο [2].

Τεχνικές προσεγγιστικού ταιριάσματος

Οι αλγόριθμοι που εφαρμόζονται στις παραπάνω δομές αφορούν κυρίως την αναζήτηση μικρών ακολουθιών βιολογικών δεδομένων (πρότυπα) πάνω στην μεγάλη ακολουθία που περιγράφεται από το ευρετήριο. Η αναζήτηση αυτή μπορεί να αφορά είτε την ακριβή ταύτιση του προτύπου (exact matching), είτε την προσεγγιστική ταύτισή του (approximate matching). Η ακριβής ταύτιση προτύπου συνίσταται στην εύρεση όλων των θέσεων της ακολουθίας που εμφανίζεται ένα πρότυπο. Ήδη αναφέρθηκε η μέθοδος με την οποία γίνεται ακριβής ταύτιση προτύπου σε δέντρα επιθεμάτων. Η προσεγγιστική ταύτιση επιχειρεί να βρει υποακολουθίες της ακολουθίας εισόδου που να μοιάζουν αρκετά με το πρότυπο. Συγκεκριμένα, η προσεγγιστική ταύτιση θεωρεί μια υποακολουθία ταυτόσημη με το πρότυπο όταν μεταξύ τους υπάρχουν το πολύ k λάθη. Λάθος ορίζεται η εισαγωγή, διαγραφή ή αντικατάσταση χαρακτήρα. Υπάρχουν παραλλαγές στους αλγορίθμους προσεγγιστικής ταύτισης, ανάλογα αν θεωρείται ότι τα λάθη βρίσκονται στο πρότυπο ή στην ακολουθία. Όλοι οι αλγόριθμοι προσεγγιστικής αναζήτησης ακολουθούν συγκεκριμένες τεχνικές. Αυτές οι τεχνικές είναι οι εξής:

Τεχνική "παραγωγής γειτονιάς" (neighborhood generation): Η τεχνική αυτή παράγει όλες τις συμβολοσειρές που έχουν k λάθη σε σχέση με την δοσμένη και ψάχνει για αυτές μέσα στο κείμενο χρησιμοποιώντας ευρετήριο. Συγκεκριμένα παράγει για ένα πρότυπο το σύνολο των εναλλακτικών μορφών που μπορεί να αποκτήσει όταν γίνουν k αλλαγές (εισαγωγές, διαγραφές και αντικαταστάσεις) στους χαρακτήρες του. Στη συνέχεια ψάχνει κάθε μία εναλλακτική με ακριβές ταίριασμα στη δομή δεδομένων. Το σύνολο των συμβολοσειρών

που ταιριάζει με ένα πρότυπο με μέγιστο αριθμό λαθών k είναι πεπερασμένο και η κάθε συμβολοσειρά που ανήκει στο σύνολο έχει μήκος μέχρι m+k, όπου m το μήκος του προτύπου. Φυσικά το μέγεθος της k-γειτονιάς μεγαλώνει εκθετικά με πολυπλοκότητα $O(m^{kk})$. Ο αλγόριθμος αυτός δουλεύει καλά για μικρά m και k, δηλαδή μικρό μήκος προτύπου και λίγα λάθη. Σε αντίθετη περίπτωση πρέπει να ελεγχθούν πολλές εναλλακτικές. Ο τύπος δεδομένων που προτείνεται για αυτό τον αλγόριθμο είναι είτε το δέντρο επιθεμάτων, είτε ο πίνακας επιθεμάτων. Σε αυτούς και μόνο τους τύπους δεδομένων μπορούμε να ψάξουμε για οποιαδήποτε υποσυμβολοσειρά της ακολουθίας εισόδου.

Τεχνική "διαμοιρασμού σε ακριβή ταιριάσματα" (partitioning into exact searching): Η τεχνική αυτή επιλέγει υποσυμβολοσειρές του προτύπου που θεωρεί πως δεν έχουν λάθη, ψάχνει για αυτές στο ευρετήριο και συγκρίνει την υπόλοιπη συμβολοσειρά του προτύπου με τη υπόλοιπη ευρεθείσα για να μετρήσει το συνολικό αριθμό λαθών. Τα λάθη σε αυτόν τον αλγόριθμο μπορεί να βρίσκονται τόσο στο πρότυπο όσο και στο κείμενο. Συγκεκριμένα η βασική ιδέα αυτού του αλγορίθμου είναι ότι κάθε συμβολοσειρά με k λάθη μπορεί να χωριστεί σε k+1 τμήματα και τότε τουλάχιστον μία υποσυμβολοσειρά δεν θα περιέχει λάθη. Ο αλγόριθμος ψάχνει στο ευρετήριο όλα τα κομμάτια στα οποία έχει χωριστεί η συμβολοσειρά, και στη συνέχεια ελέγχει τα μονοπάτια στα οποία εμφανίζονται σε σχέση με το πρότυπο. Ο εντοπισμός των υποσυμβολοσειρών μέσα στο κείμενο γίνεται με ευρετήριο, ενώ η σύγκριση των θεωρούμενων ως λανθασμένων κομματιών με on-line αλγόριθμο. Όταν θεωρούμε ότι τα λάθη βρίσκονται στο πρότυπο, τότε το χωρίζουμε σε k+s κομμάτια. Είναι σίγουρο ότι s κομμάτια δεν θα περιέχουν λάθη σε οποιαδήποτε παραλλαγή του προτύπου εμφανίζεται στο κείμενο. Οι συμβολοσειρές του κειμένου που εμφανίζουν και τα s κομμάτια ελέγχονται περαιτέρω. Δεν είναι ξεκάθαρο αν το s συμφέρει να είναι μεγάλο ή μικρό. Στην περίπτωση που είναι μεγάλο, τα κομμάτια που πρέπει να αναζητηθούν είναι μικρά και παράγουν πολλά αποτελέσματα, επιβαρύνοντας τη δεύτερη φάση του αλγορίθμου. Αν είναι μικρό, το φιλτράρισμα είναι πολύ αυστηρό και χάνουμε αποτελέσματα. Η δομή δεδομένων που θεωρείται η καταλληλότερη για ευρετήριο είναι τα Q-samples, τα οποία όμως δεν είναι χρήσιμα αν το ποσοστό λαθών είναι μεγάλο. Θεωρώντας ότι τα λάθη συμβαίνουν στο κείμενο και δεδομένου ότι χρησιμοποιούμε Q-grams ή Q-samples για την αποθήκευση του ευρετηρίου, τότε εξάγουμε από το κείμενο ανά σταθερό διάστημα χαρακτήρων h, ένα δείγμα με μέγεθος q χαρακτήρες. Στη διάρκεια της αναζήτησης τα m-q+1 Q-grams εξάγονται και γίνεται αναζήτηση τους στο ευρετήριο. Όταν κάποιο ταιριάζει, η περιοχή ελέγχεται για σωστό αποτέλεσμα. Το q πρέπει να είναι μικρό, ώστε να μην έχουμε πολύ μεγάλο σύνολο από q-samples, αλλά ταυτόχρονα αρκετά μεγάλο για να έχουμε λίγα σχετικά στοιχεία για επιβεβαίωση.

Υβριδική τεχνική (intermediate partitioning): Η τεχνική αυτή διαλέγει έναν μέσο δρόμο μεταξύ των προηγούμενων δύο, επιλέγοντας υποσυμβολοσειρές που πρέπει να εμφανίζονται αυτούσιες και παράγοντας την υπόλοιπη συμβολοσειρά με "παραγωγή γειτονιάς". Και για αυτόν τον αλγόριθμο υπάρχουν εκδοχές τόσο για λάθος στο πρότυπο όσο και για λάθος στο κείμενο. Αρχικά, όπως στο "διαμοιρασμό σε ακριβή ταιριάσματα", φιλτράρει το ευρετήριο για να βρει υποσυμβολοσειρές οι οποίες να περιέχουν περιέχουν λάθη. Στις υποσυμβολοσειρές αυτές εφαρμόζει την τεχνική της "παραγωγής γειτονιάς". Δεν υπάρχει γνωστή υλοποίηση

του αλγορίθμου αυτού με δέντρο επιθεμάτων, αλλά θα τον αναλύσουμε επιγραμματικά για την πληρότητα του κειμένου. Η κεντρική ιδέα είναι ότι αν μια συμβολοσειρά A έχει το πολύ k λάθη σε σχέση με μια συμβολοσειρά B, τότε υπάρχει υποσυμβολοσειρά της A που να εμφανίζεται με το πολύ k λάθη μέσα στο B. Αν θεωρήσουμε ότι τα λάθη βρίσκονται στο πρότυπο, τότε το χωρίζουμε σε j κομμάτια, όπως στο "διαμοιρασμό σε ακριβή ταιριάσματα", και μέσω του ευρετηρίου με διαδικασία ίδια με αυτή της "παραγωγής γειτονιάς" βρίσκουμε τα σημεία του κειμένου στα οποία εμφανίζονται τα κομμάτια αυτά, παρουσιάζοντας το πολύ k/j λάθη. Στις θέσεις που αποτελούν το αποτέλεσμα του πρώτου βήματος, ελέγχουμε την υπόλοιπη συμβολοσειρά με on-line αλγόριθμο. Η επιλογή του j είναι ιδιαίτερα κρίσιμη. Για j=1 ο αλγόριθμος εξομοιώνεται με τον "παραγωγή γειτονιάς", ενώ για j=k+1 εξομοιώνεται με τον αλγόριθμο "διαμοιρασμού σε ακριβή ταιριάσματα". Όσο μεγαλύτερο είναι το j, τόσο μειώνεται το κόστος εύρεσης των υποσυμβολοσειρών του προτύπου μέσα στο ευρετήριο, αλλά αυξάνεται το κόστος. Αν θεωρήσουμε ότι τα λάθη περιέχονται στο κείμενο, τότε αυτό θα αποτελείται από j Q-samples. Σύμφωνα με το λήμμα που αναφέρθηκε παραπάνω πρέπει να υπάρχει τουλάχιστον ένα Q-sample που να παρουσιάζεται στο πρότυπο έχοντας το πολύ k/j λάθη. Αυτή η μέθοδος ψάχνει κάθε block Q_i στο ευρετήριο των Q-samples, αναζητώντας το μικρότερο αριθμό λαθών για κάθε ταίριασμα του q-sample μέσα στο Q_i. Αν μια ομάδα από q-samples βρεθεί και τα λάθη είναι το πολύ k, τότε ελέγχεται με on-line αλγόριθμο.
Όπως θα γίνει προφανές από το επόμενο κεφάλαιο, δεν μπορούν να χρησιμοποιηθούν όλες οι δομές δεδομένων με όλους τους αλγορίθμους. Στη συνέχεια θα παρουσιάσουμε τις διάφορες δομές δεδομένων και στη συνέχεια θα επικεντρωθούμε σε αλγορίθμους προσεγγιστικού ταιριάσματος πάνω σε δένδρα επιθεμάτων. Για αλγορίθμους σε άλλες δομές μπορούν να βρεθούν περισσότερες πληροφορίες στο [2].

Δομές δεδομένων

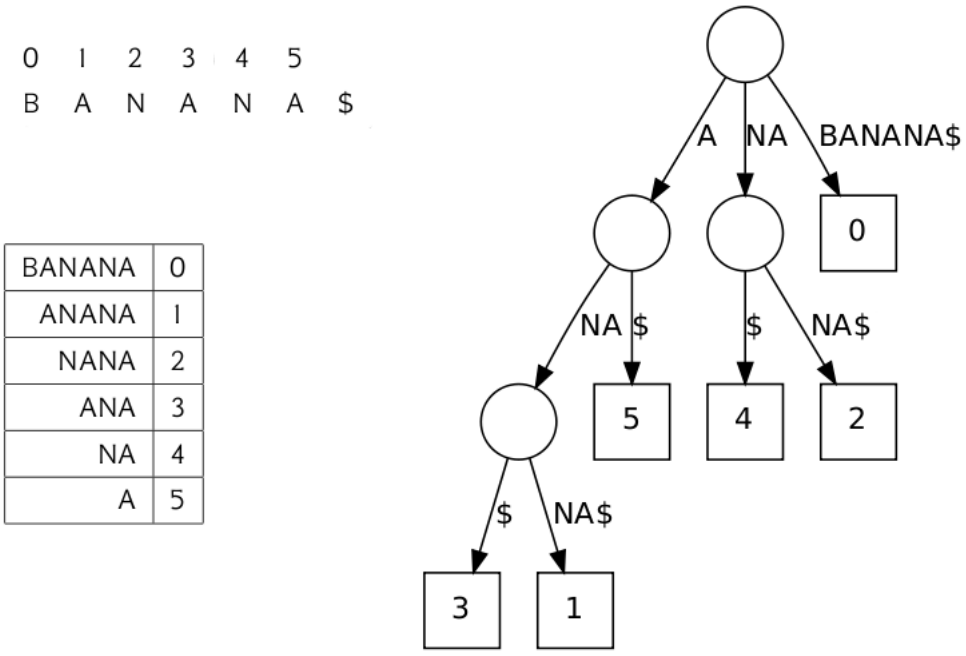
Οι δομές δεδομένων που χρησιμοποιούνται στην ευρετηριοποίηση βιολογικών δεδομένων παρουσιάζουν διαφοροποίηση ως προς τις δυνατότητες τους και το χώρο που χρειάζονται για την αποθήκευσή τους. Μέσα από τη μελέτη μας έγινε φανερό ότι οι διάφορες προσεγγίσεις προσπαθούν να προσφέρουν είτε μικρούς χρόνους αναζήτησης, είτε μικρές απαιτήσεις χώρου. Γενικά, οι δομές που δίνουν μικρότερους χρόνους κατά την αναζήτηση, απαιτούν μεγαλύτερο χώρο στη μνήμη. Έτσι τα **δέντρα επιθεμάτων**(Suffix Trees) απαιτούν τον περισσότερο χώρο, αλλά δίνουν την δυνατότητα γρήγορης αναζήτησης για οποιαδήποτε λέξη μέσα στο κείμενο. Οι **πίνακες επιθεμάτων** (Suffix Arrays) έχουν τις ίδιες δυνατότητες αναζήτησης, απαιτούν λιγότερο χώρο για την αποθήκευσή τους, αλλά είναι και πιο αργοί. Τα **Q-Grams** επιτρέπουν την αναζήτηση μόνο υποσυμβολοσειρών που είναι μεγαλύτερες από q χαρακτήρες. Τέλος τα **Q-Samples** επιτρέπουν τα ίδια με τα προηγούμενα, αλλά για ορισμένες μόνο υποσυμβολοσειρές. Πριν προχωρήσουμε στην παρουσίαση των δύο πρώτων δομών αξίζει να ορίσουμε την έννοια του επιθέματος. Ως **επίθεμα** (suffix) ορίζεται η υποσυμβολοσειρά (substring) μιας συμβολοσειράς (string) που έχει αρχή κάποιο χαρακτήρα της συμβολοσειράς και πέρας τον τελευταίο χαρακτήρα της. Έτσι για την συμβολοσειρά "BANANA" το σύνολο

των επιθεμάτων φαίνεται στον πίνακα 2.1.

BANANA
ANANA
NANA
ANA
NA
A

Πίνακας 2.1: τα επιθέματα της συμβολοσειράς BANANA

Δέντρο Επιθεμάτων: Το **δέντρο επιθεμάτων** (suffix tree) είναι μια δενδρική δομή δεδομένων στην οποία αποθηκεύεται το ευρετήριο μίας συμβολοσειράς. Στα κλαδιά του δέντρου αναπτύσσονται όλα τα επιθέματα της συμβολοσειράς εισόδου και στα φύλλα του οι θέσεις του πρώτου χαρακτήρα του κάθε επιθέματος. Ένα παράδειγμα με το δέντρο επιθεμάτων για τη συμβολοσειρά BANANA εικονίζεται στο σχήμα 2.1.



Σχήμα 2.1: Η συμβολοσειρά BANANA, τα επιθέματα της σε αντιστοίχιση με τις θέσεις που εμφανίζονται και το αντίστοιχο δένδρο επιθεμάτων

Στο σχήμα παρατηρούμε κάποιες από τις ιδιότητες των δέντρων επιθεμάτων:

1. Ένα δέντρο επιθεμάτων που περιγράφει μια συμβολοσειρά με μήκος m έχει m φύλλα.
2. Κάθε εσωτερικός κόμβος, εκτός από τη ρίζα, έχει τουλάχιστον 2 παιδιά.
3. Κάθε ακμή φέρει μη κενή υποσυμβολοσειρά που ανήκει στη συμβολοσειρά που περιγράφεται από το ευρετήριο.

4. Ακμές που ξεκινούν από τον ίδιο κόμβο δεν μπορούν να έχουν ίδιο αρχικό χαρακτήρα.
5. Η συρραφή των υποσυμβολοσειρών των ακμών που ορίζουν ένα μονοπάτι από τη ρίζα του δέντρου σε ένα φύλλο μας δίνει ένα έγκυρο επίθεμα και η τιμή του φύλλου την θέση του επιθέματος μέσα στην συμβολοσειρά.
6. Η παρουσία ενός χαρακτήρα που δεν ανήκει στο αλφάβητο της συμβολοσειράς (εδώ το \$) είναι απαραίτητη για να διασφαλίσει ότι κανένα επίθεμα δεν είναι πρόθεμα (prefix) άλλου, καθώς και τον κανόνα 1 για τον ίσο αριθμό χαρακτήρων και φύλλων στη συμβολοσειρά και το δέντρο.

Η παραδοσιακή χρήση του δέντρου επιθεμάτων είναι για γρήγορη αναζήτηση για ακριβείς (χωρίς λάθη) εμφανίσεις μιας μικρής συμβολοσειράς μέσα στη συμβολοσειρά πάνω στην οποία έχει οικοδομηθεί το δέντρο. Η διαδικασία συνίσταται στο να ακολουθήσουμε τις ακμές του δέντρου από τη ρίζα προς τα φύλλα. Το πλεονέκτημα αυτής της δομής είναι ότι ο χρόνος αναζήτησης είναι ίσος με τον χρόνο που χρειαζόμαστε για να επισκεφθούμε τους αντίστοιχους κόμβους. Είναι εμφανές ότι η πολυπλοκότητα της αναζήτησης είναι $O(m)$ όπου m το μήκος της υποσυμβολοσειράς που αναζητούμε. Αυτό είναι το βασικό πλεονέκτημα της χρήσης του δένδρου επιθεμάτων σε μια τέτοια αναζήτηση, καθώς ο χρόνος που απαιτείται για την αναζήτηση μιας μικρής υποσυμβολοσειράς είναι ανεξάρτητος από το πόσο μεγάλη είναι η συμβολοσειρά των δεδομένων. Τα βασικά μειονεκτήματα του δένδρου επιθεμάτων είναι δύο. Πρώτον, το μέγεθος του δένδρου είναι πολύ μεγάλο: στην καλύτερη περίπτωση ένα δένδρο επιθεμάτων μπορεί να είναι περίπου 9 φορές μεγαλύτερο από μέγεθος των αρχικών δεδομένων. Δεύτερον, η φάση κατασκευής του είναι ιδιαίτερα χρονοβόρα και μάλιστα εμπλέκει τυχαίες προσπελάσεις πάνω στα δεδομένα κάτι το οποίο μειώνει ακόμα περισσότερο την απόδοση.

Πίνακας Επιθεμάτων: Ο Πίνακας Επιθεμάτων (Suffix Array) είναι μια διαφορετική εκδοχή του δέντρου επιθεμάτων, με λιγότερες απαιτήσεις χώρου, περίπου 4 φορές περισσότερες από το αρχικό κείμενο. Αντίστοιχα μας δίνει και μεγαλύτερους χρόνους αναζήτησης. Για να το δημιουργήσουμε, αρκεί να επισκεφτούμε τα φύλλα του δέντρου επιθεμάτων από αριστερά προς τα δεξιά και να καταγράψουμε τον αριθμό του φύλλου. Έτσι ο πίνακας επιθεμάτων είναι ένας μονοδιάστατος πίνακας από αυτούς τους αριθμούς. Αξίζει πάντως να σημειωθεί ότι έχει αποδειχθεί [1] πώς με την προσθήκη πολύ μικρής πληροφορίας, οποιοσδήποτε αλγόριθμος για δέντρα επιθεμάτων μπορεί να μεταφερθεί με την ίδια χρονική πολυπλοκότητα σε πίνακες επιθεμάτων.

Q-gram: Το Q-gram έχει ακόμα λιγότερες δυνατότητες, με αντίστοιχο κέρδος σε χώρο. Αυτός ο τύπος δεδομένων περιορίζει το μήκος των προτύπων που μπορούμε να αναζητήσουμε στο ευρετήριο, με μέγιστο έστω q. Το ευρετήριο αυτό αποθηκεύει για όλες τις διαφορετικές υποσυμβολοσειρές με μήκος q τις θέσεις που εμφανίζονται στο κείμενο.

Q-sample: Το Q-sample μειώνει ακόμα περισσότερο τις απαιτήσεις σε χώρο αποθηκεύοντας μόνο κάποια Q-grams από όλα όσα έχει το κείμενο.

2.2 Προσεγγιστικά ταιριάσματα με χρήση δέντρων επιθεμάτων

Σε αυτό το κεφάλαιο θα επικεντρωθούμε στους αλγορίθμους προσεγγιστικού ταιριάσματος προτύπου που είναι εξειδικευμένοι για τη δομή των δέντρων επιθεμάτων. Αρχικά θα ορίσουμε την έννοια του προσεγγιστικού ταιριάσματος και στη συνέχεια θα αναλύσουμε 3 αλγορίθμους που χρησιμοποιούν την παραπάνω έννοια για να κάνουν προσεγγιστικά ταιριάσματα με χρήση δέντρου επιθεμάτων. Το προσεγγιστικό ταίριασμα αποδεικνύεται στην πράξη πολύ πιο χρήσιμο από το ακριβές. Αυτό ισχύει ιδιαίτερα για το πεδίο των βιολογικών δεδομένων, αφού το γενετικό υλικό υφίσταται αλλαγές τόσο κατά τη διάρκεια ζωής ενός οργανισμού όσο και από γενιά σε γενιά. Έχει αποδειχθεί ότι μικρές αλλαγές στο γενετικό υλικό έχουν ως συνέπεια μεγάλες αλλαγές στις λειτουργίες ενός οργανισμού. Έτσι είναι φανερό ότι όταν αναζητούμε πληροφορίες σε βιολογικό υλικό δεν μπορούμε να ορίσουμε το πρότυπο με μαθηματική ακρίβεια, αλλά αναμένουμε και αποτελέσματα που έχουν υποστεί αλλαγές. Μια γενική εικόνα για τους αλγορίθμους που θα αναλύσουμε δίνεται από το [12].

Πριν προχωρήσουμε στην ανάλυση των τεχνικών για προσεγγιστικό ταίριασμα σε δέντρα επιθεμάτων, πρέπει πρώτα να τυποποιήσουμε το βαθμό στον οποίο διαφέρουν μεταξύ τους δύο ακολουθίες. Η πιο απλή τυποποίηση εστιάζει στο να βρεθεί το πλήθος των συντακτικών πράξεων που είναι απαραίτητες για να γίνουν ίδιες οι δύο ακολουθίες. Το πλήθος αυτό ονομάζεται συντακτική απόσταση (edit distance) των 2 ακολουθιών. Οι επιτρεπτές συντακτικές πράξεις είναι εισαγωγή ή διαγραφή χαρακτήρα στην πρώτη ακολουθία και η αντικατάσταση ενός χαρακτήρα της δεύτερης ακολουθίας με χαρακτήρα της πρώτης. Ένα παράδειγμα φαίνεται στον πίνακα 2.2.

Πράξη	I	M	M	M	D	D	M	R
Ακολουθία 1	-	μ	ε	σ	ι	τ	ώ	ν
Ακολουθία 2	α	μ	έ	σ	-	-	ω	ς

Πίνακας 2.2: οι πράξεις μετασχηματισμού της λέξης "μεσιτών" στη λέξη "αμέσως"

Η ακολουθία με αλφάβητο I, D, M, R που περιγράφει το μετασχηματισμό μιας ακολουθίας σε μία δεύτερη ονομάζεται συντακτικό μεταγραφής (edit transcript) ή απλούστερα, μεταγραφή. Ο υπολογισμός της συντακτικής απόστασης βρίσκεται στην καρδιά της προσεγγιστικής αναζήτησης ενός προτύπου. Θα δούμε πώς μπορούμε με χρήση δυναμικού προγραμματισμού να υπολογίσουμε τη συντακτική απόσταση δύο ακολουθιών. Η προσέγγιση αυτή μπορεί να μετατραπεί και σε αλγόριθμο προσεγγιστικής αναζήτησης. Εδώ υποθέτουμε ότι δεν διαθέτουμε κάποιο ευρετήριο για τις ακολουθίες.

Υπολογισμός συντακτικής απόστασης: Έστω ακολουθίες S1, S2 με |S1| = m και |S2| = n. Θα συμβολίζουμε με D(i, j) τη συντακτική απόσταση των προθεμάτων S1[1...i] και S2[1...j]. Όπως αφήνει να εννοηθεί η χρήση του όρου δυναμικός προγραμματισμός, μπορούμε να υπολογίσουμε τη συντακτική απόσταση των δύο αυτών προθεμάτων αναδρομικά, υπολογίζοντας τη συντακτική απόσταση μικρότερων προθεμάτων. Αν η ακολουθία S1 έχει m

γράμματα και ακολουθία S2 έχει n γράμματα, τότε η συντακτική απόσταση τους θα συμβολίζεται με D(m, n). Για να υπολογίσουμε το τελευταίο, αρκεί με δυναμικό προγραμματισμό να υπολογιστούν τα D(i, j) για όλα τα δυνατά i και j. Η προσέγγιση του δυναμικού προγραμματισμού απαιτεί τρία βήματα:

1. την αναδρομική σχέση (recursive relation)
2. τον υπολογισμό πίνακα (tabular computation)
3. την προς τα πίσω αναζήτηση (trackback)

Η αναδρομική σχέση συνδέει την τιμή της D(i, j) με τιμές του D για i'<i, j'<j. Οι δείκτες αυτοί είναι θετικοί και η αναδρομή θα εξαντληθεί για αρχικές συνθήκες οι οποίες ορίζονται D(i, 0) = i και D(0, j) = j. Η πρώτη αρχική συνθήκη είναι σωστή, γιατί για να μετασχηματίσουμε i χαρακτήρες της πρώτης ακολουθίας σε 0 χαρακτήρες της δεύτερης, αρκεί να διαγραφούν όλοι με χρήση i διαγραφών. Η δεύτερη αρχική συνθήκη είναι επίσης σωστή γιατί απαιτούνται j εισαγωγές στην αρχικά κενή πρώτη ακολουθία ώστε αυτή να γίνει ίδια με j χαρακτήρες της δεύτερης ακολουθίας. Μπορούμε τώρα να ορίσουμε την αναδρομική σχέση ως εξής:

$$D(i,j) = min[D(i-1,j)+1, D(i,j-1), D(i-1,j-1)+t(i,j)]$$

όπου t(i,j) μια συνάρτηση που ισούται με το 1 αν $S1(i) \neq S2(j)$ και 0 όταν $S1(i) = S2(j)$. Αφού ορίστηκε η αναδρομική συνάρτηση, στη συνέχεια πρέπει να βρεθεί τεχνική για αποδοτικό υπολογισμό του D(m,n). Μια απλοϊκή προσέγγιση θα ήταν η χρήση του αναδρομικού ορισμού. Όμως αυτή η προσέγγιση δεν είναι αποδοτική γιατί ο αριθμός των αναδρομικών κλήσεων αυξάνει εκθετικά με την αύξηση των m, n. Μάλιστα υπάρχουν $(m+1)\cdot(n+1)$ συνδυασμοί των i, j, άρα υπάρχουν $(m+1)\cdot(n+1)$ αναδρομικές κλήσεις που είναι δυνατόν να γίνουν και το πλήθος γίνεται δυσανάλογα μεγάλο. Παρατηρούμε ότι η μέθοδος υπολογισμού από πάνω προς τα κάτω δεν είναι αποδοτική. Η αντίστροφη πορεία υπολογισμού, από κάτω προς τα πάνω, είναι πολύ καλύτερη. Σε αυτή την προσέγγιση υπολογίζεται πρώτα η τιμή της D(i,j) για τις μικρότερες δυνατές τιμές των i, j. Κατόπιν υπολογίζεται το D(i,j) αυξάνοντας τις τιμές των i, j. Ο υπολογισμός γίνεται με τη χρήση πίνακα δυναμικού προγραμματισμού μεγέθους $(m+1)\cdot(n+1)$. Ένας πίνακας δυναμικού προγραμματισμού για τις συμβολοσειρές "writers" και "vintner" φαίνεται στον πίνακα 2.3. Τα στοιχεία που αντιστοιχούν στην γραμμή και τη στήλη 0 είναι οι αρχικές συνθήκες της αναδρομικής σχέσης. Τα υπόλοιπα $m \cdot n$ στοιχεία του πίνακα μπορούν να υπολογιστούν αυξάνοντας αρχικά το i (μία σειρά τη φορά) και μετά αυξάνοντας το j (όλες οι στήλες με τη σειρά για αυτή τη σειρά). Ο υπολογισμός των στοιχείων του πίνακα μπορεί να γίνει σε σταθερό χρόνο. Η συμπλήρωση του πίνακα απαιτεί $(m+1)\cdot(n+1)$ πράξεις. Άρα η συντακτική απόσταση δύο ακολουθιών μήκους m και n απαιτεί χρόνο O(mn). Κατόπιν θα πρέπει να εξάγουμε το βέλτιστο συντακτικό μεταγραφής. Ο ευκολότερος τρόπος είναι να δημιουργήσουμε κατάλληλους δείκτες κατά τη συμπλήρωση του πίνακα. Συγκεκριμένα, όταν υπολογίζεται η τιμή του στοιχείου (i,j), θέτουμε ένα δείκτη από το στοιχείο (i,j) σε ένα δεύτερο στοιχείο. Το δεύτερο αυτό στοιχείο είναι το:

1. (i,j-1), αν D(i,j) = D(i,j-1)+1

2. (i-1,j), αν D(i,j) = D(i-1,j)+1
3. (i-1,j-1), αν D(i,j) = D(i-1,j-1) + t(i,j)

Ο κανόνας αυτός ισχύει για όλα τα στοιχεία του πίνακα, ακόμα και για όσα βρίσκονται στη γραμμή ή τη στήλη 0. Τα στοιχεία της γραμμής 0 δείχνουν το στοιχείο αριστερά τους, ενώ τα στοιχεία της στήλης 0 το στοιχείο πάνω τους. Τα υπόλοιπα στοιχεία του πίνακα έχουν συνήθως πάνω από έναν δείκτες. Για να ανακτήσουμε το βέλτιστο συνταχτικό μεταγραφής με χρήση των δεικτών αυτών αρκεί να ακολουθήσουμε ένα από τα μονοπάτια που συνδέουν το στοιχείο (m,n) με το (0,0). Η ερμηνεία των βημάτων μας από το (i,j) είναι η εξής: κίνηση προς το (i,j-1) αντιστοιχεί με εισαγωγή (I) του χαρακτήρα S2(j) στο S1, κίνηση προς το (i-1,j) αντιστοιχεί με διαγραφή (D) του χαρακτήρα S1(i) από το S1 και τέλος κίνηση προς το (i-1,j-1) αντιστοιχεί σε αντικατάσταση του S1(i) από το S2(j), αν $S1(i) \neq S2(j)$ ή ταύτιση των δύο χαρακτήρων αν S1(i)=S2(j). Αν υπάρχουν παραπάνω από ένας δείκτες, τότε μπορούμε να επιλέξουμε οποιονδήποτε από αυτούς για να κινηθούμε. Αφού κάθε φορά μετακινούμαστε 1 στήλη ή 1 γραμμή ή 1 στήλη και μία γραμμή σε έναν πίνακα m γραμμών και n στηλών, το μονοπάτι θα απαιτήσει το πολύ m+n βήματα, άρα σε χρόνο O(m+n). Ένα παράδειγμα πίνακα δυναμικού προγραμματισμού με συμπληρωμένους τους δείκτες φαίνεται στον πίνακα 2.4.

Η παραπάνω μέθοδος ανακτά όλες τις βέλτιστες δυνατές μεταγραφές. Στη συνέχεια θα αναλύσουμε πώς ο παραπάνω αλγόριθμος σε συνδυασμό με δέντρο επιθεμάτων έχει οδηγήσει σε 3 διαφορετικούς αλγορίθμους τύπου "παραγωγή γειτονιάς".

Αλγόριθμος Jokinen & Ukkonen: Μια πρώτη προσπάθεια για ανάπτυξη τεχνικών για προσεγγιστικό ταίριασμα σε δέντρο επιθεμάτων έγινε από τους Jokinen και Ukkonen το 1991 [20]. Στην εργασία τους χρησιμοποιούν τη τεχνική δυναμικού προγραμματισμού που περιγράψαμε παραπάνω. Επιπρόσθετα, ορίζουν το συντομότερο επίθεμα L(i,j), όπου i και j η αρχική και τελική θέση του επιθέματος μέσα στο κείμενο T. Η συντακτική απόσταση του L(i,j) από το πρότυπο P μπορεί να υπολογιστεί μέσω δυναμικού προγραμματισμού και τελικά το D(i,j) ισούται με την συντακτική απόσταση των P[1...i] και T[j'...j], όπου j' = j - L(i,j) + 1. Ο αλγόριθμος χρησιμοποιεί δέντρο επιθεμάτων που ονομάζει SA(T). Επιπλέον χρησιμοποιεί

D(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4	4	4	*			
n	5	5							
e	6	6							
r	7	7							

Πίνακας 2.3: ο πίνακας δυναμικού προγραμματισμού για τις λέξεις "writers" και "vintner"

πίνακα m+1 κελιών παρόμοια με τις δομές D και L ο οποίος δρα βοηθητικά στην πληροφορία του δέντρου επιθεμάτων. Η στήλη που αναπαριστά τη συντακτική απόσταση στην κατάσταση r αναφέρεται ως dcol(r) και η στήλη που αναπαριστά το μήκος των ακολουθιών αναφέρεται ως lcol(r). Χρησιμοποιώντας αυτούς τους ορισμούς προχωρούν σε μια μέθοδο δυναμικού προγραμματισμού που βρίσκει τα προσεγγιστικά ταιριάσματα του προτύπου P στο κείμενο T μέσω του δέντρου επιθεμάτων SA(T). Η μέθοδός τους δουλεύει με τα εξής βήματα:

1. Διάσχιση του χρήσιμου υποδέντρου U(P,k) του SA(T), αρχίζοντας από τη ρίζα του δέντρου και χρησιμοποιώντας αλλαγμένο τον αλγόριθμο του Dijkstra για το ελάχιστο μονοπάτι.
2. Όταν η διάσχιση μπει στην κατάσταση r με μια μετακίνηση του τύπου goto(s,a)=r, τότε γίνεται επεξεργασία των dcol(r) και lcol(r) μέσω δυναμικού προγραμματισμού από το a, dcol(s), rcol(s).
3. Αν το $dcol(r)(m) \leq k$, τότε σημείωσε όλες τις καταστάσεις που είναι γειτονικές της κατάστασης r και δεν είναι ήδη σημειωμένες. Δώσε ως αποτέλεσμα το βάθος του q, όπου q γειτονική κατάσταση του r που δεν σημειώθηκε.

Αναλυτικότερα, έστω d(i,x) η ελάχιστη συντακτική απόσταση μεταξύ οποιουδήποτε επιθέματος του x και του προτύπου p. Έστω επίσης l(i,x) το μήκος του μικρότερου τέτοιου επιθέματος. Η διάσχιση του δέντρου αρχίζει από τη ρίζα. Αρχικά τα dcol και lcol μπορούν να δείχνουν οποιοδήποτε i. Η διάσχιση του δέντρου γίνεται με τέτοιο τροπο ώστε όλα τα προσεγγιστικά ταιριάσματα του P να βρεθούν χωρίς περιττές κινήσεις. Αυτό είναι εφικτό αν κάθε δείκτης (goto) του SA(T) δεν γίνεται διάσχιση παραπάνω από μία φορά. Αφού υπάρχουν O(n) μετακινήσεις και κάθε μετακίνηση χρειάζεται O(m) χρόνο, χρειαζόμαστε συνολικά O(mn) για τη συνολική διάσχιση. Ας ορίσουμε τώρα το χρήσιμο υποδέντρο U(P,k). Έστω λ(x) το μήκος του μεγαλύτερου επιθέματος ψ της συμβολοσειράς x. Προφανώς λ(x) = l(i,x). Το χρήσιμο υποδέντρο U(P,k) του SA(T) είναι ο υπογράφος που περιέχει όλες τις χρήσιμες καταστάσεις και τους δείκτες μετακίνησης των καταστάσεων αυτών.

D(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
v	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4	↖← 5	↖← 6	↖← 7
i	2	↑ 2	↖↑ 2	↖ 2	↖ 2	← 3	← 4	← 5	← 6
n	3	↑ 3	↖↑ 3	↖↑ 3	↖↑ 3	↖ 3	↖↑ 4	↖↑ 5	↖↑ 6
t	4	↑ 4	↖↑ 4	↖↑ 4	↖↑ 4	↖ 3	↖↑ 4	↖↑ 5	↖↑ 6
n	5	↑ 5	↖↑ 5	↖↑ 5	↖↑ 5	↑ 4	↖ 4	↖↑ 5	↖↑ 6
e	6	↑ 6	↖↑ 6	↖↑ 6	↖↑ 6	↑ 5	↖ 4	↖↑ 5	↖↑ 6
r	7	↑ 7	↖↑ 7	↖ 6	↖↑ 7	↑ 6	↑ 5	↖ 4	← 5

Πίνακας 2.4: ο πίνακας δυναμικού προγραμματισμού για τις λέξεις “writers” και “vintner” με τους απαραίτητους δείκτες

Αλγόριθμος Ukkonen: Σε μετέπειτα εργασία του Ukkonen [18] αναπτύσσεται ακόμα μία μέθοδος για δυναμικό προγραμματισμό πάνω σε δέντρα επιθεμάτων με την τεχνική της "παραγωγής γειτονιάς". Ο πιο φυσικός τρόπος για να εφαρμοστεί η τεχνική του δυναμικού προγραμματισμού σε ένα δέντρο επιθεμάτων είναι να γίνει μια διάσχισή του κατά βάθος που θα βρει όλες τις υποσυμβολοσειρές P' του κειμένου με συντακτική απόσταση μέχρι k από το P. Η αναζήτηση είναι εύκολη γιατί απλά χρειάζεται να ελεγχθεί ένα μονοπάτι μέχρι να συμβούν περισσότερα από k λάθη και το μονοπάτι να είναι ψευδές ή μέχρι να φτάσουμε σε φύλλο οπότε το μονοπάτι είναι στοιχείο του P'. Η εργασία του Ukkonen βελτιώνει την πολυπλοκότητα του αλγορίθμου αυτού στη χειρότερη περίπτωση από $\Theta(mn)$ σε $O(n)$. Η βασική ιδέα είναι ότι το κείμενο μπορεί περιέχει πολλές επαναλήψεις των ίδιων χαρακτήρων. Θέτει λοιπόν ένα όριο k και ελέγχει τα κελιά του πίνακα δυναμικού προγραμματισμού χαρακτηρίζοντας ως απαραίτητα μόνο τα κελιά με τιμή $\leq k$. Καλεί ακόμα βιώσιμο k-προσεγγιστικό πρόθεμα το πρόθεμα από το οποίο εξαρτάται μια στήλη του πίνακα. Έτσι ίδιοι χαρακτήρες τις εισόδου θα έχουν ίδια βιώσιμα k-προσεγγιστικά προθέματα. Για να αποφύγει να εξετάσει κάποια στήλη της οποίας το βιώσιμο πρόθεμα έχει ήδη εξεταστεί, κρατάει πίνακες-στήλες στο δέντρο επιθεμάτων. Έτσι μια στήλη με βιώσιμο πρόθεμα Q αποθηκεύεται μαζί με το μονοπάτι του δέντρου από τη ρίζα μέχρι να φτάσουμε αυτό το πρόθεμα. Η διάσχιση του δέντρου επιθεμάτων αποφεύγει να εξετάσει το βιώσιμο πρόθεμα κάποιου χαρακτήρα, αν το έχει ήδη ελέγξει.

Αλγόριθμος Cobbs: Η πιο πρόσφατη προσέγγιση επιχειρήθηκε από τον Cobbs [5]. Κύρια έννοια στην εργασία αυτή έχει το βιώσιμο επίθεμα V(i,j) που ορίζεται ως το πρόθεμα που προέκυψε από την κανονικοποιημένη διάσχιση του πίνακα δυναμικού προγραμματισμού για το D(i,j). Όπως είδαμε στην ανάλυση της μεθόδου δυναμικού προγραμματισμού, για ένα ζευγάρι τιμών (i,j) μπορεί να υπάρχουν πολλά μονοπάτια ελάχιστου κόστους για να φτάσουμε στην πρώτη γραμμή. Ο αλγόριθμος του Cobbs επιλέγει να μετακινείται στον πίνακα, προτιμώντας κινήσεις προς τα πάνω, μετά διαγώνια και τέλος οριζόντια. Το αποτέλεσμα είναι ένα κανονικοποιημένο μοναδικό μονοπάτι από το (i,j). Το μονοπάτι αυτό είναι βέλτιστο, και ορίζει το βιώσιμο επίθεμα V(i,j). Ο αλγόριθμος δρα κατόπιν σε m γύρους για την κατασκευή των βιώσιμων προθεμάτων. Στον i-οστό γύρο κατασκευάζει το V_i από το V_{i-1}. Αρχικά το V_i είναι κενό. Κατασκευάζουμε υποψήφια μέλη για το V_i κάνοντας επεκτάσεις στα μέλη του V_{i-1} τόσο οριζόντια όσο και κάθετα. Όταν ένα υποψήφιο πρόθεμα p δημιουργηθεί, ελέγχεται η κανονικότητα του και προστίθεται στο V_i. Η διαδικασία συνεχίζεται μέχρι να μην υπάρχουν άλλα υποψήφια προθέματα. Όταν παραχθούν τα προθέματα για το τελικό V_m, τότε καθένα από αυτά αναζητείται με προτεραιότητα βάθους στο δέντρο επιθεμάτων T.

2.3 Η διαχρονική εξέλιξη των τεχνολογιών δέντρων επιθεμάτων

Όπως είναι φανερό από τα παραπάνω, τα δέντρα επιθεμάτων είναι μια δομή που εξυπηρετεί ιδιαίτερα για τις λειτουργίες που επιζητούμε από ένα ευρετήριο ακολουθιακών βιολογικών δεδομένων. Σε αυτό το κεφάλαιο θα μελετήσουμε τις διάφορες αλγοριθμικές προσεγγίσεις και προγράμματα που έχουν αναπτυχθεί για την κατασκευή δέντρων επιθεμάτων. Αρχικά τα δέντρα

επιθεμάτων κατασκευάζονταν στη μνήμη. Πρόκειται για πρώιμες τεχνικές που ασχολούνται με την ευρετηριοποίηση μικρών εισόδων. Γρήγορα έγινε φανερό ότι το δέντρο επιθεμάτων μπορεί να γίνει πολύ μεγάλο για τη μνήμη των υπολογιστών. Έτσι αναπτύχθηκαν τεχνικές που χρησιμοποιούν το σκληρό δίσκο για την κατασκευή. Τέλος παρουσιάζουμε και μια παράλληλη μέθοδο που εκμεταλλεύεται τους σύγχρονους πολυπύρηνους υπολογιστές.

2.3.1 Κατασκευή δέντρων επιθεμάτων στη μνήμη

Θα ξεκινήσουμε με γενικούς αλγορίθμους κατασκευής δέντρου επιθεμάτων οι οποίοι στοχεύουν στην ευρετηριοποίηση δεδομένων που χωράνε στη μνήμη. **Απλοϊκή προσέγγιση:** Η απλοϊκή προσέγγιση συνίσταται στο να να εισάγουμε με τη σειρά όλα τα επιθέματα σε όλες τις δυνατές θέσεις. Πρώτη εισάγεται η ίδια η συμβολοσειρά μαζί με το τερματικό σύμβολο, S[0..n-1]$, και κατόπιν όλα τα υπόλοιπα επιθέματα. Για κάθε επίθεμα που εισάγεται στο δέντρο, ξεκινάμε από τη ρίζα του δέντρου και προς τα φύλλα του δέντρου, συγκρίνοντας τους χαρακτήρες του επιθέματος με τους χαρακτήρες που είναι αποθηκευμένοι στην ακμή. Άμα φτάσουμε σε κόμβο θα διαλέξουμε ως επόμενη ακμή αυτή που αρχίζει με επόμενο χαρακτήρα τον επόμενο χαρακτήρα του επιθέματος . Λόγω του τελικού συμβόλου $ είναι σίγουρο ότι θα φτάσουμε σε χαρακτήρα του επιθέματος που διαφέρει από τον αντίστοιχο της ακμής. Σε αυτό το σημείο θα προσθέσουμε καινούργια ακμή, με τους χαρακτήρες του επιθέματος που περίσσεψαν, καθώς και καινούργιο φύλλο με όνομα τη θέση του προθέματος στην ακολουθία. Η προσέγγιση αυτή, ενώ δίνει σωστά αποτελέσματα, είναι πολύ αργή με πολυπλοκότητα $O(n^2)$ καθώς απαιτεί n+1 επαναλήψεις για την εισαγωγή n+1 επιθεμάτων (το μήκος της ακολουθίας και ο τερματικός χαρακτήρας) και περίπου O(n) συγκρίσεις χαρακτήρων στο εκάστοτε δέντρο.

Γραμμικός Αλγόριθμος Weiner: Ο πρώτος γραμμικός αλγόριθμος κατασκευής δέντρο επιθεμάτων διατυπώθηκε από τον P. Weiner το 1973, [21]. Σε αυτή την εργασία προτείνεται για πρώτη φορά η χρήση συνδέσμων επιθεμάτων. Επιτρέπει την πρόσθεση χαρακτήρων και προς τα αριστερά και προς τα δεξιά. Αποτελεί τη βάση για όλους τους μετέπειτα γραμμικούς αλγόριθμους. Χρησιμοποιεί όμως εξαιρετικά πολύ χώρο, $O(n * s)$ όπου s το μέγεθος του αλφάβητου και αυτό είναι το κύριο μειονέκτημά του.

Γραμμικός Αλγόριθμος McCreight: Ο αλγόριθμος του McCreight [11] είναι λειτουργικά ισοδύναμος με του Weiner, αλλά απαιτεί 25% λιγότερο χώρο. Ο αλγόριθμος επιτυγχάνει γραμμική απόδοση με τη χρήση συνδέσμων επιθέματος. Η μείωση σε απαιτήσεις χώρου κατά 25% είναι αποτέλεσμα της πρόσθεσης συμβολοσειράs από αριστερά προς τα δεξιά. Έτσι γλυτώνει ένα δείκτη στη δομή που είναι απαραίτητος σε άλλους αλγορίθμους για επέκταση του υποσυμβολοσειρά προς τα αριστερά.

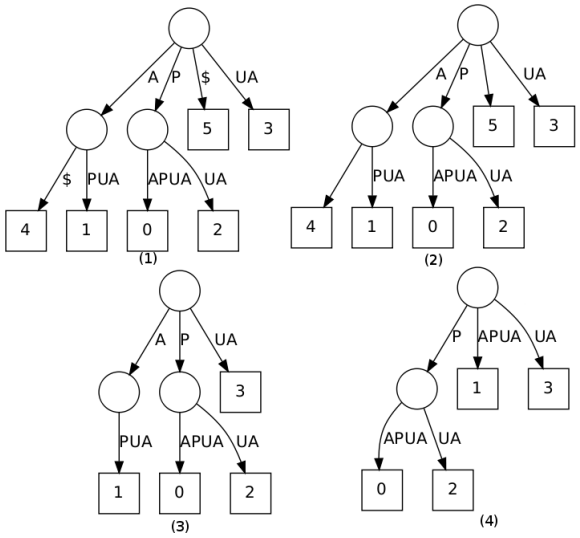
Γραμμικός Αλγόριθμος Ukkonen: Ο αλγόριθμος του Ukkonen [19] παρουσιάστηκε το 1995 και είναι ένας online αλγόριθμος κατασκευής δέντρου επιθεμάτων που διαβάζει την είσοδο από αριστερά προς τα δεξία. Για μια συμβολοσειρά n χαρακτήρων λειτουργεί σε n φάσεις, προσθέτοντας κάθε φορά τον επόμενο χαρακτήρα. Η χρονική και χωρική του πολυπλοκότητα είναι O(n). Αρχικά θα παρουσιαστεί η έννοια του πεπλεγμένου δέντρου που είναι

η μορφή του δέντρου που κατασκευάζει ο αλγόριθμος μέχρι και πριν το τελευταίο βήμα. Στη συνέχεια θα παρουσιάσουμε τη βασική ιδέα του αλγορίθμου μαζί με μια απλοϊκή υλοποίηση. Έπειτα θα παρουσιαστούν οι βελτιστοποιήσεις που καθιστούν γραμμικό τον αλγόριθμο.

Κύρια έννοια του αλγορίθμου είναι το πεπλεγμένο δέντρο επιθεμάτων (implicit suffix tree). Το πεπλεγμένο δένδρο επιθεμάτων μιας συμβολοσειράς S προκύπτει από το δένδρο επιθεμάτων για τη συμβολοσειρά S$ μετά από μια σειρά βημάτων:

1. Πρέπει να διαγραφεί ο χαρακτήρας $ από τελικός χαρακτήρας όλων των φύλλων.
2. Ακμές που φέρουν κενή ετικέτα (δηλαδή έφεραν πριν μόνο τον χαρακτήρα $) διαγράφονται.
3. Κόμβοι που έχουν μόνο μία εξερχόμενη ακμή διαγράφονται και η ακμή αυτή συγχωνεύεται με την εισερχόμενη.

Η διαδικασία φαίνεται στο σχήμα 2.2.



Σχήμα 2.2: (1)Το πεπλεγμένο δέντρο επιθεμάτων για το PAPUA$. (2)Μετά τη διαγραφή του $. (3)Μετά τη διαγραφή των ακμών με άδεια συμβολοσειρά. (4)Το πραγματικό δέντρο επιθεμάτων

Ένα πεπλεγμένο δέντρο επιθεμάτων για τη συμβολοσειρά S θα έχει λιγότερα φύλλα από το δέντρο επιθεμάτων για το S$ αν κάποιο από τα επιθέματα του S είναι πρόθεμα κάποιου άλλου επιθέματος. Στο παράδειγμά μας το 'a' είναι πρόθεμα του "apua". Ο τερματικός χαρακτήρας $ αποσκοπεί στο να μην υπάρχει επίθεμα που να είναι πρόθεμα άλλου επιθέματος. Το πεπλεγμένο δέντρο επιθεμάτων δε διαθέτει φύλλο για κάθε επίθεμα, περιέχει όμως όλα τα επιθέματα, έστω και αν το μονοπάτι κάποιων δεν καταλήγει στη μέση μιας ακμής και όχι σε φύλλο. Για αυτά τα επιθέματα δεν μπορεί να ανακτηθεί η θέση τους μέσα στην συμβολοσειρά. Το πεπλεγμένο δένδρο επιθεμάτων φέρει λιγότερη πληροφορία από ότι το γνήσιο δένδρο επιθεμάτων. Θα συμβολίσουμε το πεπλεγμένο δένδρο επιθεμάτων της ακολουθίας S[1...i] με Ti όπου το i μπορεί να πάρει τιμές μεταξύ του 1 και του n. Ο αλγόριθμος του Ukkonen κατασκευάζει ένα πεπλεγμένο δέντρο επιθεμάτων Ti για κάθε πρόθεμα S[1...i], ξεκινώντας από το T1 και

προσθέτοντας ένα χαρακτήρα κάθε φορά μέχρι να κατασκευάσει το Tn. Στο τέλος το γνήσιο δέντρο επιθεμάτων κατασκευάζεται από το Tn με την προσθήκη ενός τερματικού χαρακτήρα που δεν υπάρχει στη συμβολοσειρά, για παράδειγμα $.

Ακολουθεί μια απλοϊκή προσέγγιση υλοποίησης του αλγορίθμου του Ukkonen σε ψευδοκώδικα:

```
1. Κατασκεύασε το πεπλεγμένο δέντρο για το επίθεμα S[1]
2. i = 2
3. Κατασκεύασε το πεπλεγμένο δέντρο για το S[1...i]
   με βάση το προηγούμενο πεπλεγμένο δέντρο S[1...i-1]
4. i++
5. αν i == n+1 τότε
     κατασκεύασε το κανονικό δέντρο από το πεπλεγμένο
   αλλιώς
     πήγαινε στο βήμα 3
```

Με μια συστηματικότερη ανάλυση, ο αλγόριθμος του Ukkonen χωρίζει τη λειτουργία του σε **n** φάσεις, μία για κάθε δυνατό πρόθεμα του κειμένου. Σε κάθε φάση ο αλγόριθμος κατασκευάζει το πεπλεγμένο δέντρο T_i από το προηγούμενο T_{i-1} μέσω i επεκτάσεων, μία για κάθε επίθεμα του προθέματος S[1...i]. Κατά την επέκταση j της φάσης i ο αλγόριθμος βρίσκει το τέλος του μονοπατιού για την υποσυμβολοσειρά S[j...i-1] και την επεκτείνει προσθέτοντας τον χαρακτήρα S(i). Συγκεκριμένα, αν S[j...i-1] το j-οστό επίθεμα του S[1...i-1], τότε κατά την επέκταση j της φάσης i, όταν ο αλγόριθμος βρει το επίθεμα αυτό θα το επεκτείνει έτσι ώστε το επίθεμα S[j...i-1]S(i) να βρίσκεται στο νέο πεπλεγμένο δέντρο. Η επέκταση αυτή μπορεί να γίνει με βάση κάποιον από τους τρεις παρακάτω κανόνες:

1. **Κανόνας 1** Αν ο αλγόριθμος βρει το τέλος του επιθέματος S[j...i-1] σε φύλλο του δέντρου, τότε απλά προσθέτει τον χαρακτήρα S(i) στο τέλος της ακμής.
2. **Κανόνας 2** Αν ο αλγόριθμος βρει το τέλος του επιθέματος σε εσωτερικό ακμής, τότε δημιουργεί νέο κόμβο. Σε αυτή την περίπτωση, καθώς και στην περίπτωση που το τέλος του επιθέματος βρίσκεται σε εσωτερικό κόμβο, εφόσον δεν υπάρχει ο S(i) ως επόμενος χαρακτήρας, τον προσθέτει σε νέα ακμή.
3. **Κανόνας 3** Αν ο αλγόριθμος βρει το τέλος του επιθέματος και ο επόμενος χαρακτήρας είναι ο S(i), τότε δεν χρειάζεται να πράξει τίποτα γιατί το επίθεμα βρίσκεται ήδη μέσα στο δέντρο.

Σύμφωνα με τα παραπάνω εφόσον βρεθεί το τέλος του επιθέματος S[j...i-1] η επέκταση μπορεί να πραγματοποιηθεί σε σταθερό χρόνο. Η απλοϊκή τεχνική που ακολουθήθηκε στον ψευδοκώδικα και στην περιγραφή των επεκτάσεων απαιτεί $O(n^3)$ χρόνο εκτέλεσης και $O(n^2)$ χώρο. Ακολουθεί μια σειρά τεχνασμάτων που δίνουν γραμμική πολυπλοκότητα χώρου και χρόνου στον αλγόριθμο του Ukkonen.

1. **Σύνδεσμοι επιθέματος** Έστω μια συμβολοσειρά χa με χ χαρακτήρα και a συμβολοσειρά, επιτρέποντας η a να είναι η κενή συμβολοσειρά. Για έναν εσωτερικό κόμβο

v του οποίου το μονοπάτι είναι το χa, υπάρχει άλλος κόμβος s(v) με μονοπάτι a. Ο δείκτης που προστίθεται από τον v στον s(v) ονομάζεται **σύνδεσμος επιθέματος**. Σε περίπτωση που το a είναι η κενή συμβολοσειρά, τότε ο σύνδεσμος επιθέματος δείχνει στη ρίζα του δέντρου. Το δέντρο επιθεμάτων με τους συνδέσμους επιθέματος για την συμβολοσειρά BANANA φαίνεται στο σχήμα 2.3. Χρησιμοποιούμε συνδέσμους επιθέματος για να κινηθούμε στο δέντρο. Οι σύνδεσμοι επιθέματος είναι δείκτες από κόμβο σε κόμβο που μας δείχνουν πού πρέπει να γίνει η επόμενη προσθήκη. Η ιδιότητα των συνδέσμων επιθέματος είναι ότι αν το μονοπάτι από τη ρίζα σε ένα κόμβο σχηματίζει μια συμβολοσειρά S[j...i], τότε ο σύνδεσμος επιθέματος του κόμβου αυτού δείχνει στον κόμβο του οποίου το μονοπάτι σχηματίζει το συμβολοσειρά S[j+1...i]. Αποδεικνύεται ότι αν προστεθεί ένας νέος εσωτερικός κόμβος v στο δέντρο με μονοπάτι χa, τότε είτε το μονοπάτι με ετικέτα a υπάρχει ήδη και τελειώνει σε εσωτερικό κόμβο του δέντρου, είτε θα δημιουργηθεί στην επόμενη επέκταση της φάσης αυτής. Άρα κάθε εσωτερικός κόμβος του δέντρου έχει σύνδεσμο επιθέματος.

Αναλυτικά η διαδικασία με την οποία χρησιμοποιείται ο κανόνας πρόσθεσης νέων χαρακτήρων με τους συνδέσμους επιθέματος είναι ο εξής:

(α΄) **Βήμα 1** Βρίσκουμε τον πρώτο εσωτερικό κόμβο v στον οποίο τελειώνει το μονοπάτι για S[j-1...i-1] ή είναι ρίζα του δέντρου. Έστω ακόμα γ η ετικέτα μεταξύ του μονοπατιού του v και του τέλους της ακολουθίας S[j-1...i-1] που μπορεί να μην είναι κενή.

(β΄) **Βήμα 2** Αν ο v είναι ρίζα τότε γίνεται διάσχιση του μονοπατιού για το S[j...i-1]. Αν το v δεν είναι ρίζα τότε γίνεται η μετάβαση στον επόμενο κόμβο με χρήση του συνδέσμου επιθεμάτων και διάσχιση για το γ.

(γ΄) **Βήμα 3** Γίνεται χρήση του κανόνα για επέκταση των επιθεμάτων από S[j...i-1] -> S[j...i-1].

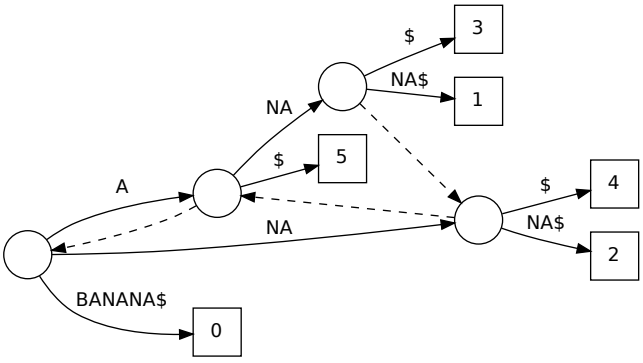
(δ΄) **Βήμα 4** Αν κατά την επέκταση δημιουργήθηκε νέος εσωτερικός κόμβος, πρέπει να οριστεί ο σύνδεσμος επιθέματος του ως ο προηγούμενος κόμβος στον οποίο έγινε επέκταση.

2. **Skip and Count** Με μια απλοϊκή προσέγγιση, η διάσχιση του μονοπατιού του a θα γινόταν με συνεχή σύγκριση χαρακτήρων. Όμως από τον ορισμό δεν μπορεί να ξεκινούν από τον ίδιο κόμβο 2 ακμές με τον ίδιο πρώτο χαρακτήρα. Έτσι αν βρούμε ακμή με το σωστό πρώτο γράμμα και το μήκος της ακμής είναι μικρότερο από τους χαρακτήρες που έχουμε ακόμα να ελέγξουμε για να φτάσουμε στο τέλος του μονοπατιού, τότε μπορούμε απλά να μεταβούμε στον κόμβο που εισέρχεται η ακμή αυτή. Έτσι δεν χρειάζεται να ελέγξουμε κανέναν χαρακτήρα ακμής παρά μόνο τον πρώτο. Αντί να διατρέξουμε όλους τους χαρακτήρες της ακμή, αρκεί να πάμε στον κόμβο-πατέρα του και να ακολουθήσουμε το σύνδεσμο επιθέματος. Βασική προϋπόθεση είναι να ξέρουμε ότι και τα δύο ακμές στις οποίες προσθέτουμε χαρακτήρες έχουν το ίδιο μήκος. Μια εφαρμογή του κανόνα φαίνεται στο σχήμα 2.4
3. **Πρόσθεση χαρακτήρων** Αρχικά παρατηρούμε ότι ένα φύλλο του δέντρου θα παρα-

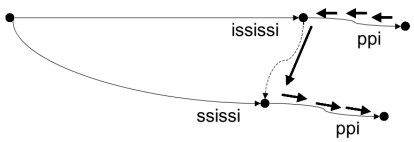
μείνει φύλλο σε όλη τη διάρκεια της διαδικασίας, παίρνοντας σε κάθε φάση τον επιπλέον χαρακτήρα που διαβάζουμε. Έτσι μπορούμε εξαρχής να δηλώσουμε ότι το φύλλο έχει τελικό χαρακτήρα το τέλος της συμβολοσειράς και να μην προσθέσουμε τίποτα σε αυτό στην υπόλοιπη διάρκεια εκτέλεσης του αλγορίθμου. Επίσης, για μία ακμή με label S[j..i] μπορεί να υπάρχει ήδη επόμενη ακμή που αρχίζει με τον χαρακτήρα i+1, σχηματίζοντας υπαρκτό μονοπάτι για το S[j...i+1]. Και σε αυτή την περίπτωση θα προσθέτουμε σε κάθε φάση τον χαρακτήρα που διαβάσαμε και μπορούμε να εφαρμόσουμε το παραπάνω τέχνασμα. Αυτές οι δύο περιπτώσεις αντιστοιχούν στους κανόνες 1 και 3. Άρα οι μόνες προσθήκες που χρειάζεται να κάνουμε είναι όταν δεν υπάρχει μονοπάτι από το S[j..i] στο i+1, οπότε και πρέπει να προσθέσουμε νέο φύλλο στο δέντρο. Αυτή η προσθήκη μπορεί να γίνει σε γραμμικό χρόνο.

4. **Edge Compression** Αντί να κρατάμε αναλυτικά τους χαρακτήρες για κάθε ακμή, κρατάμε απλά την αρχική και τελική θέση του υποσυμβολοσειρά που αντιπροσωπεύουν. Είναι εφικτό, καθώς κρατάμε ένα αντίγραφο της εισόδου στη μνήμη. Αυτή η τεχνική δίνει στον Ukkonen γραμμική απαίτηση χώρου.

Συνολικά οι κανόνες αυτοί εξασφαλίζουν στον Ukkonen γραμμική απαίτηση χώρου και χρόνου. Αφού προσθέσουμε όλα τα επιθέματα της συμβολοσειράς στο δέντρο, το δέντρο μας παραμένει πεπλεγμένο, όπως φαίνεται στο σχήμα 2.5. Η μόνη πράξη που απαιτείται για να πάρουμε ένα γνήσιο δέντρο επιθεμάτων είναι να τρέξουμε μια ακόμα επανάληψη φάσης για χαρακτήρα που δεν υπάρχει μέσα στην συμβολοσειρά, για παράδειγμα τον χαρακτήρα $.



Σχήμα 2.3: το δέντρο επιθεμάτων για τα επιθέματα του BANANA$. Οι διακεκομμένες γραμμές είναι οι σύνδεσμοι επιθέματος

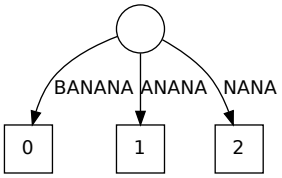


Σχήμα 2.4: Η τεχνική skip and count σε πεπλεγμένο δέντρο της συμβολοσειράς mississippi

2.3.2 Κατασκευή δέντρων επιθεμάτων στο δίσκο

Παρουσιάστηκε το πώς μπορεί να γίνει αποδοτικά η κατασκευή δέντρου επιθεμάτων στη μνήμη. Όπως έχει αναφερθεί το δέντρο επιθεμάτων είναι πολύ μεγαλύτερο από την ακολουθία που περιγράφει. Επίσης, κατά την κατασκευή απαιτεί τυχαίες προσπελάσεις της ακολουθίας. Έτσι στην πρώτη περίπτωση μπορεί το δέντρο να μην χωράει στην κύρια μνήμη, ενώ στην δεύτερη να χρειάζεται συνεχόμενη επικοινωνία σκληρού δίσκου και μνήμης για την ανάκτηση των δεδομένων. Τα προβλήματα αυτά κάνει τους αλγορίθμους μνήμης ακατάλληλους από μόνους τους να αντέξουν τις απαιτήσεις ευρετηριοποίησης μεγάλων ακολουθιών βιολογικών δεδομένων. Για την αποδοτική λύση του προβλήματος αυτού έχουν προταθεί διάφορες τεχνικές. Αρκετές από αυτές εγκαταλείπουν την χρήση των συνδέσμων επιθεμάτων, γιατί προκαλούν τυχαίες προσπελάσεις κατά την ανάγνωση.

Hunt: Μια πρώτη προσέγγιση έγινε από τους Hunt, Atkinson και Irving [9]. Η εργασία εκμεταλλεύεται την αναδρομική δομή των δέντρων επιθεμάτων: Για οποιοδήποτε κόμβο του δέντρου που έχει μονοπάτι με ετικέτα p τότε το υποδέντρο που αυτός ορίζει περιέχει όλα τα επιθέματα που έχουν πρόθεμα το p. Έτσι το δέντρο μπορεί να διαμεριστεί με κριτήριο τα προθεματικά υποδέντρα. Η τεχνική που χρησιμοποιούν βρίσκει τη συχνότητα προθεμάτων σταθερού (fixed) μήκους. Αν το μήκος αυτό είναι αρκετά μεγάλο, τότε πιθανότατα τα προθεματικά υποδέντρα να χωρούν στη μνήμη. Όμως τα προθεματικά υποδέντρα αυτά δεν περιέχουν όλα τα επιθέματα μιας ακολουθίας και έτσι δεν αποτελούν δέντρα επιθεμάτων. Για αυτό δεν μπορεί να χρησιμοποιηθεί κάποιος αλγόριθμος όπως του Ukkonen, που βασίζεται στην χρήση συνδέσμων επιθέματος για κάθε εσωτερικό κόμβο. Οι σύνδεσμοι επιθέματος αυτοί δεν μπορούν να ανακτηθούν καθώς δείχνουν σε άλλες διαμερίσεις. Έτσι εγκαταλείπεται τελείως η χρήση συνδέσμων επιθεμάτων. Μάλιστα στην εργασία γίνεται κριτική στους συνδέσμους επιθεμάτων, καθώς θεωρούνται η κύρια αιτία του προβλήματος της τυχαίας προσπέλασης. Αυτό συμβαίνει γιατί με τους συνδέσμους επιθέματος η διάσχιση του δέντρου κατά την κατασκευή του γίνεται όχι μόνο από πάνω προς τα κάτω (ρίζα προς φύλλα), αλλά και οριζόντια, από εσωτερικούς κόμβους σε άλλους εσωτερικούς κόμβους. Η ανάκτηση των συνδέσμων επιθέματος θα πρέπει να γίνει μετά την κατασκευή των υποδέντρων με κόστος $O(n^2)$ χρόνο. Ακόμα ένα πρόβλημα της εργασίας αυτής είναι η υπόθεση ότι τα βιολογικά δεδομένα έχουν ομοιόμορφη κατανομή ως προς τη συχνότητα των διάφορων υποακολουθιών. Αυτή η υπόθεση όπως θα δούμε στο Trellis, δεν ισχύει. Αντίθετα, τα προθέματα σταθερού μήκους οδηγούν σε κάποια



Σχήμα 2.5: το τελικό πεπλεγμένο δέντρο επιθεμάτων για το BANANA

υποδέντρα που είναι πολύ μεγάλα και δεν χωράνε στη μνήμη και κάποια υποδέντρα που είναι πολύ μικρά. Η εργασία προσπαθεί να λύσει αυτό το πρόβλημα χρησιμοποιώντας προθέματα μεγαλύτερου μήκους και την τεχνική bin packing, που όμως ανήκει στα NP προβλήματα. Με τη μέθοδό τους κατάφεραν να κατασκευάσουν το δέντρο επιθεμάτων για 20M ζευγάρια βάσεις σε 7 λεπτά, χρησιμοποιώντας 2GB κύρια μνήμη.

DynaCluster και TOP-Q: Οι τεχνικές DynaCluster [4] και TOP-Q [3] καταφέρνουν να κατασκευάσουν τα δέντρα στη μνήμη διατηρώντας την τοπικότητα αναφοράς, αντιμετωπίζοντας το πρόβλημα της ανισοκατανομής των δεδομένων και κάνοντας χρήση των συνδέσμων επιθεμάτων. Ο DynaCluster βασίζει την κατασκευή του δέντρου στη δυναμική ομαδοποίηση των δεδομένων της ακολουθίας. Τα ομαδοποιημένα δεδομένα έχουν τοπικότητα αναφοράς και το δέντρο κατασκευάζεται σε φάσεις, με μία ομάδα να προστίθεται ανά φάση. Στο τέλος γίνεται προαιρετική ανάκτηση των συνδέσμων επιθέματος. Όπως αναφέρθηκε και προηγούμενα, η μη ομοιόμορφη κατανομή των βάσεων στο DNA σημαίνει πως δεν έχουν όλες οι υποακολουθίες την ίδια πιθανότητα εμφάνισης μέσα στα βιολογικά δεδομένα. Ο αλγόριθμος TOP-Q επιχειρεί να εφαρμόσει μια στρατηγική έξυπνου buffering που βασίζεται σε στοχαστικές ιδιότητες των κόμβων του δέντρου επιθεμάτων κατά τη διάρκεια της κατασκευής του. Το πόσο συχνά θα προσπελαστεί ένας κόμβος κατά τη διάρκεια κατασκευής του δέντρου έχει να κάνει με τις στοχαστικές ιδιότητες της ακολουθίας. Έχουν γίνει πολλές προσπάθειες για την ταξινόμηση των ακολουθιών στη βάση των στοχαστικών τους ιδιοτήτων. Ένα από τα απλούστερα μοντέλα που προτείνεται για να προσεγγιστούν οι στοχαστικές αυτές ιδιότητες είναι οι γεννήτριες Bernoulli. Με την παραπάνω μοντελοποίηση, η ακολουθία θεωρήθηκε αποτέλεσμα διαδοχικών δοκιμών σε μια γεννήτρια Bernoulli. Από τα στατιστικά στοιχεία που συλλέχθηκαν κατά την κατασκευή του δέντρου επιθεμάτων για ακολουθίες που παρήχθησαν από τη γεννήτρια Bernoulli διαπιστώθηκε ότι οι κόμβοι που βρίσκονται κοντά στη ρίζα προσπελαύνονται περισσότερες φορές από ότι οι κόμβοι που βρίσκονται χαμηλότερα στο δέντρο. Άρα κρίνεται σκόπιμο οι κόμβοι αυτοί να αποθηκευθούν στη μνήμη, έτσι ώστε να επιταχυνθούν οι προσπελάσεις τους. Ο αλγόριθμος που αναπτύχθηκε πρέπει να έχει τη δυνατότητα να αναγνωρίσει τους κρίσιμους κόμβους κατά τη διάρκεια της κατασκευής του δέντρου. Διαπιστώθηκε ακόμα πειραματικά ότι όσο μεγαλύτερη είναι η ετικέτα μιας ακμής του δέντρου, τόσο μεγαλύτερες είναι οι πιθανότητες αυτή να διασπαστεί, καθώς εισάγονται νέοι κόμβοι στο δέντρο. Σημειώνεται ότι οι ακμές με μήκος 1 δεν μπορούν να διασπαστούν. Επίσης, με χρήση της στατιστικής διαπιστώθηκε ότι για κάποιο μικρό ποσοστό κόμβων δεν μπορεί να γίνει ικανοποιητική εκτίμηση, αλλά όσο αυξάνεται το μέγεθος του δέντρου επιθεμάτων, τόσο μειώνεται η συχνότητα των εσφαλμένων εκτιμήσεων. Για να εκμεταλλευτεί τις παραπάνω διαπιστώσεις ο αλγόριθμος θεωρεί ότι οι σελίδες του δίσκου περιέχουν κόμβους που είναι είτε εσωτερικοί, είτε φύλλα, αλλά όχι ανάμειξη των δύο. Η ανάμιξη αποφεύγεται γιατί τα φύλλα έχουν μικρότερο μέγεθος στη μνήμη από τους εσωτερικούς κόμβους και η ανάμιξη τους δεν θα εκμεταλλευόταν πλήρως τη χωρητικότητα της σελίδας του δίσκου. Η κάθε σελίδα χαρακτηρίζεται από τη μέση τιμή του μήκους μονοπατιού όλων των κόμβων που περιέχει. Χρησιμοποιώντας την εκτίμηση βάθους για κάθε σελίδα που αναφέραμε, η προτεραιότητα για παραμονή στη μνήμη δίνεται σε σελίδες με μικρότερο μήκος μονοπατιού. Η μέθοδος

αυτή ονομάστηκε TOP (κορυφή) για να καταδείξει το γεγονός ότι προσπαθεί να κρατήσει στη μνήμη τις σελίδες του δέντρου επιθεμάτων που εκτιμάται ότι περιέχουν τους κόμβους που βρίσκονται πιο κοντά στη ρίζα. Η τελική εκδοχή του αλγορίθμου ονομάζεται TOP-Q και περιλαμβάνει μια επιπλέον βελτιστοποίηση. Από τα πειραματικά αποτελέσματα της εργασίας φάνηκε ότι η πολιτική απομάκρυνσης σελίδων του TOP δεν ήταν πάντα εύστοχη, γιατί κόμβοι που εισάγονται χαμηλά στο δέντρο χρειάζεται πολλές φορές να προσπελαστούν λίγους χαρακτήρες αργότερα. Έτσι δεν πρέπει να μεταφέρονται οι κόμβοι που βρίσκονται χαμηλά στα δέντρα από την cache στο δίσκο. Δημιουργήθηκε έτσι μια δομή ουράς για την αποθήκευση των σελίδων, που εξασφαλίζει ότι θα καθυστερήσει η απομάκρυνση των παραπάνω σελίδων από την cache. Τα αποτελέσματα έδειξαν σαφή βελτίωση των αποτελεσμάτων και πολύ μεγαλύτερο hit-rate από την LRU (Least Recently Used) πολιτική αντικατάστασης buffer. Οι αλγόριθμοι DynaCluster και TOP-Q αποδεικνύονται αποδοτικοί για παραγωγή ευρετηρίων της τάξης των 100M ζευγαριών βάσεων, αλλά η απόδοση τους πέφτει ραγδαία και σίγουρα δεν μπορούν να χρησιμοποιηθούν για την παραγωγή ευρετηρίου για το σύνολο του ανθρώπινου γονιδιώματος που φτάνει τα 3G ζευγάρια βάσεις.

TDD: Η τεχνική TDD (Top-Down Disk Based) [17] είναι ο πρώτος αλγόριθμος που είναι αρκετά αποδοτικός για να χαρτογραφήσει το ανθρώπινο γονιδίωμα. Βασίζεται στη μέθοδο του διαμερισμού και εγγραφής μόνο από τη ρίζα προς τα φύλλα (partition and write only top down), που συνίσταται στη διαμέριση της ακολουθίας και τη δημιουργία του υποδέντρου για κάθε διαμέριση με χρήση του αλγορίθμου wotdeager. Ο αλγόριθμος wotdeager διασφαλίζει ότι οποιοσδήποτε κόμβος του δέντρου θα γραφτεί μόνο μια φορά. Ξεκινά την κατασκευή από τη ρίζα και συνεχίζει με τα παιδιά της ρίζας και τα παιδιά των παιδιών της ρίζας και ούτω καθεξής μέχρι τα φύλλα. Αρχικά όλα τα επιθέματα της ακολουθίας τοποθετούνται σε έναν πίνακα και ταξινομούνται με βάση τον πρώτο τους χαρακτήρα. Η ταξινόμηση αυτή μας επιτρέπει να χωρίσουμε τα προθέματα σε 5 κατηγορίες ανάλογα αν ξεκινάνε με κάποια από τις 4 βάσεις ή από τον χαρακτήρα $ για την εφαρμογή σε ακολουθία εισόδου DNA. Στη συνέχεια ελέγχονται οι ομάδες στις οποίες χωρίστηκαν τα προθέματα. Αν μια ομάδα περιέχει παραπάνω από 2 επιθέματα, τότε είναι σίγουρο ότι υπάρχει κάποιος εσωτερικός κόμβος με πρόθεμα τον αρχικό χαρακτήρα της ομάδας. Κατασκευάζεται λοιπόν αυτός ο εσωτερικός κόμβος και ως ετικέτα στην ακμή που προηγείται βάζουμε το μέγιστο κοινό πρόθεμα όλων των επιθεμάτων της κατηγορίας. Αν η κατηγορία περιέχει μόνο ένα επίθεμα, τότε υπάρχει φύλλο με αυτό το επίθεμα. Κατασκευάζεται το φύλλο και ως ετικέτα λαμβάνει όλο το αντίστοιχο επίθεμα. Αφού συμβούν τα παραπάνω για όλες τις κατηγορίες, τότε ο αλγόριθμος αυξάνει για κάθε κατηγορία τους δείκτες των επιθεμάτων της κατηγορίας κατά το μήκος του μέγιστου κοινού προθέματος και στη συνέχεια επαναλαμβάνουμε την ίδια διαδικασία για τα επιθέματα της προηγούμενης κατηγορίας. Τα επιθέματα επαναταξινομούνται με βάση τον πρώτο χαρακτήρα και η διαδικασία επαναλαμβάνεται γραμμικά μέχρι την ολοκλήρωση της κατασκευής του δέντρου. Η μέθοδος PWOTD διαμερίζει την ακολουθία με βάση προθέματα σταθερού μήκους και κατασκευάζει για κάθε διαμέριση ένα δέντρο με χρήση του αλγορίθμου wotdeager που αναλύθηκε πριν. Η μέθοδος απαιτεί 4 δομές δεδομένων: έναν πίνακα για την ακολουθία εισόδου, έναν πίνακα με επιθέματα, έναν προσωρινό πίνακα και ένα δέντρο επιθεμάτων. Ο

πίνακας επιθεμάτων γεμίζει με τις θέσεις ενός επιθέματος συγκεκριμένου μήκους. Έπειτα ταξινομεί τον πίνακα αυτό με βάση τον πρώτο χαρακτήρα. Έπειτα, με καταμέτρηση καταρτίζεται προσωρινός πίνακας με δείκτες επιθεμάτων ως προς τον πρώτο χαρακτήρα και προσδιορίζεται έτσι ο αριθμός των επιθεμάτων που έχει κάθε κατηγορία. Στη συνέχεια δρα ο αλγόριθμος wotdeager, κατασκευάζοντας κατάλληλα τους εσωτερικούς κόμβους και τα φύλλα. Οι κόμβοι που δεν μπορούν να εισαχθούν αμέσως φυλάσσονται σε μια στοίβα. Η διαδικασία συνεχίζει έως ότου όλοι οι κόμβοι να επεκταθούν με τη δημιουργία των υποδέντρων τους και η στοίβα να αδειάσει. Η παραπάνω προσέγγιση εγκαταλείπει τη λογική των συνδέσμων επιθεμάτων και θα αναμέναμε να έχει χειρότερη πολυπλοκότητα από τον Ukkonen. Ωστόσο, αν οι δομές χωρούν ολόκληρες στη μνήμη, ο αλγόριθμος είναι καλύτερος, γιατί με τη μέθοδο της κατασκευής του δέντρου από πάνω προς τα κάτω εξασφαλίζεται μεγάλη χωρική τοπικότητα. Το γεγονός αυτό σε συνδυασμό με τις μεγάλες cache των σημερινών επεξεργαστών κάνουν την προσέγγιση αυτή πολύ γρήγορη. Όμως αν η ακολουθία δεν χωράει στη μνήμη, ο αλγόριθμος παρουσιάζει μεγάλες καθυστερήσεις λόγω του I/O που απαιτείται κατά την κατασκευή. Για την περίπτωση αυτή προτάθηκε η βελτίωση ST-Merge [TTH+05], που χωρίζει την ακολουθία σε συνεχόμενες υποακολουθίες. Στη συνέχεια χρησιμοποιεί το TDD για την κατασκευή ενός δέντρου για κάθε μία από τις υπακολουθίες. Το πλήρες δέντρο προκύπτει από συγχώνευση των μικρότερων δέντρων.

Trellis: Το πρόγραμμα Trellis [14] προτάθηκε το 2007 και αποτελεί την αποδοτικότερη προσέγγιση για την κατασκευή δέντρων στο δίσκο. Το καινοτόμο στοιχείο του Trellis είναι ότι χρησιμοποιεί ως κριτήριο διαχωρισμού του δέντρου επιθεμάτων προθέματα μεταβλητού μήκους. Γενικά λειτουργεί σε 4 στάδια. Στο πρώτο αποφασίζει μια λίστα προθεμάτων μεταβλητού μήκους όπου το κάθε στοιχείο της λίστας να εμφανίζεται το πολύ t φορές στην ακολουθία εισόδου. Έπειτα διαιρεί το κείμενο σε t κομμάτια και κτίζει για το κάθε κομμάτι και το κάθε πρόθεμα ένα προθεματικό υποδέντρο επιθεμάτων. Τα υποδέντρα με το ίδιο πρόθεμα συγχωνεύονται σε προθεματικά δέντρα. Προαιρετικά γίνεται ανάκτηση των συνδέσμων επιθέματος. Το όριο t είναι κατάλληλα υπολογισμένο έτσι ώστε ένα δέντρο επιθεμάτων με μήκος t να χωρά σίγουρα σε κάποια χωρητικότητα κύριας μνήμης. Το Trellis θα αναλυθεί σε βάθος σε επόμενο κεφάλαιο.

2.3.3 Παράλληλη κατασκευή δέντρων επιθεμάτων

Wavefront: Το Wavefront [8] είναι ο πρώτος γνωστός αλγόριθμος κατασκευασμένος ειδικά για παράλληλα συστήματα. Δημοσιεύθηκε το 2009 στο συνέδριο SIGMOD και είναι εργασία της IBM. Το Wavefront έχει ως στόχο να εκμεταλλευτεί τα χαρακτηριστικά των σύγχρονων παράλληλων υπολογιστών. Πρόκειται για υπολογιστές με πάρα πολλούς πυρήνες (1024) οι οποίοι όμως δεν διαθέτουν σκληρό δίσκο και η κύρια μνήμη τους είναι περιορισμένη (512MB). Είναι διασυνδεδεμένοι με δίκτυο πολύ χαμηλής απόκρισης. Επίσης, ο αλγόριθμος Wavefront επιχειρεί να κατασκευάσει το δέντρο με τοπικές αναφορές στην ακολουθία, έτσι ώστε να εκμεταλλευτεί τα μεγάλα buffer των σύγχρονων σκληρών δίσκων.

Ο ίδιος ο αλγόριθμος Wavefront είναι σχεδιασμένος για να έχει σταθερές απαιτήσεις

μνήμης. Ξεφεύγει από τις προσεγγίσεις τύπου "διαίρεση και συγχώνευση" και φτιάχνει το δέντρο απευθείας. Η είσοδος του αλγορίθμου είναι το μέγεθος της κύριας μνήμης M και η ακολουθία εισόδου S. Το Wavefront εκτελείται σε 3 στάδια.

1. Κατασκευή προθεμάτων Σε αυτό το στάδιο βρίσκει μια ομάδα από προθέματα P, τέτοια ώστε το προθεματικό δέντρο επιθεμάτων να χωράει στη μνήμη.
2. Κατασκευή υποδέντρων Σε αυτό το στάδιο κατασκευάζει ένα προθεματικό δέντρο επιθεμάτων για κάθε πρόθεμα P.
3. Ανάκτηση συνδέσμων επιθεμάτων Σε αυτό το στάδιο γίνεται επανάκτηση των συνδέσμων επιθεμάτων.

Κατασκευή προθεμάτων: Το Wavefront επιλέγει να κατασκευάσει προθέματα μεταβλητού μήκους έτσι ώστε τα προθεματικά δέντρα της δεύτερης φάσης να χωράνε στη μνήμη. Για να δημιουργήσει τη λίστα προθεμάτων P, σαρώνει την είσοδο πολλές φορές ανά B bytes. Στο τέλος της κάθε σάρωσης ένα πρόθεμα προστίθεται στη λίστα P.
Κατασκευή υποδέντρων: Ο στόχος αυτής της φάσης είναι να κατασκευαστεί ένα υποδέντρο του ολικού δέντρου επιθεμάτων για κάθε πρόθεμα του συνόλου P. Αυτό το στάδιο καταναλώνει τον περισσότερο χρόνο του αλγορίθμου. Η τεχνική κατασκευής του προθεματικού δέντρου είναι μια διαφορετική προσέγγιση στον αλγόριθμο της Hunt. Όπως προαναφέραμε, ο αλγόριθμος αυτός εισάγει κάθε επίθεμα P_i στο δέντρο αρχίζοντας από τη ρίζα. Αφού βρει το μέγιστο κοινό πρόθεμα, δημιουργεί καινούργιο κόμβο και φύλλο για τους χαρακτήρες που δεν ταιριάζουν. Ο αλγόριθμος αυτός δουλεύει πολύ καλά στην πράξη με την προϋπόθεση ότι το υποδέντρο και η ακολουθία χωράνε στη μνήμη. Όμως όπως αναφέραμε παρουσιάζονται προβλήματα σε περίπτωση που κάτι τέτοιο δεν ισχύει. Οι αλλαγές στον αλγόριθμο αυτό έχουν ως βασική ιδέα ότι στα υποδέντρα επιθεμάτων η θέση του πρώτου χαρακτήρα κάθε ακμής είναι αριθμός μεγαλύτερος από τον πρώτο χαρακτήρα της προηγούμενης ακμής που βρίσκεται πιο κοντά στη ρίζα. Άρα οι ακμές ενός δέντρου μπορούν να διαχωριστούν με βάση τη θέση του αρχικού τους χαρακτήρα, έτσι ώστε κάθε ακμή να έχει την προηγούμενή της μέσα στην ίδια διαμέριση. Για μια ακολουθία εισόδου διαμερισμένη σε κομμάτια μεγέθους B, οι ακμές είναι διαμερισμένες σε n/B διαφορετικές διαμερίσεις σύμφωνα με τον αρχικό τους χαρακτήρα και η κάθε διαμέριση περιλαμβάνει O(B) ακμές. Υποθέτουμε ότι η ακολουθία εισόδου έχει χωριστεί σε B κομμάτια, δίνοντας μας συνολικά n/B κομμάτια δέντρου. Τότε το υποδέντρο επιθεμάτων θα κατασκευαστεί σε n/B βήματα. Στο i-οστό βήμα, κάθε επίθεμα που αρχίζει με το πρόθεμα p μπαίνει στο δέντρο επιθεμάτων. Στο τέλος του i-οστού βήματος κάθε επίθεμα με πρόθεμα p είτε έχει εισαχθεί πλήρως στο δέντρο, είτε έχει εισαχθεί μέχρι ένα σημείο στο οποίο όλες οι αναφορές στην ακολουθία εισόδου μέχρι και το i έχουν ολοκληρωθεί. Για τη δεύτερη κατηγορία κρατάμε μια δομή front για το επόμενο βήμα. Για κάθε ένα από αυτά τα επιθέματα κρατάμε πληροφορίες για το σημείο μέχρι το οποίο έχει εισαχθεί. Το σημείο αυτό είναι η τελευταία ακμή που προσπελάστηκε και ονομάζεται ενεργή ακμή (Active Edge). Επίσης η δομή front Κρατάει πληροφορίες για την ταυτότητα του δέντρου και το μέγεθος του επιθέματος που μόλις εισάγαμε από την ακολουθία εισόδου. Αυτή η δομή κρατείται σε στοίβα. Όταν η στοίβα αδειάσει πλήρως, σταματάμε τη διαδικασία. Η παραπάνω διαδικασία

υποθέτει ότι όλα τα επιθέματα είναι πάντοτε διαθέσιμα στην κύρια μνήμη. Αυτή η παραδοχή δεν μπορεί να ισχύσει για πολύ μεγάλες ακολουθίες εισόδου. Αυτό ο περιορισμός αίρεται, διαμορφώνοντας τη σειρά των προσβάσεων έτσι ώστε να γίνονται όταν ένα επίθεμα μπαίνει στο δέντρο. Έστω πάλι ότι η ακολουθία εισόδου χωρίζεται σε κομμάτια μεγέθους B που ονομάζονται κομμάτια εισαγωγής (Insert Block). Κάθε i-οστό κομμάτι του δέντρου χρειάζεται μόνο τα κομμάτια εισαγωγής με ταυτότητα μεγαλύτερη από i. Με αυτή τη διαδικασία, θα υπάρξουν επιθέματα που διασχίζουν διαδοχικά κομμάτια εισαγωγής που δεν έχουν εισαχθεί πλήρως. Αυτά σώζονται στο τέλος του κομματιού και γίνεται επεξεργασία τους στο i+1 βήμα. Για στοίβα front με O(n) εγγραφές, γίνεται επεξεργασία για μόνο O(B) από αυτές με ένα κομμάτι εισαγωγής.

Ανάκτηση συνδέσμων επιθεμάτων: Η ανάκτηση των συνδέσμων επιθεμάτων γίνεται σε δύο φάσεις. Στην πρώτη φάση προσπελαύνουμε κάθε προθεματικό δέντρο επιθεμάτων και βρίσκουμε όλους τους εσωτερικούς κόμβους που δείχνουν σε ρίζα άλλων προθεματικών δέντρων. Η διαδικασία απαιτεί πρόσβαση στο προθεματικό δέντρο μόνο μέχρι ένα βάθος τέτοιο, ώστε να ανακτηθεί ο σύνδεσμος επιθέματος. Και εδώ αλλάζει η μέθοδος με την οποία προσπελαύνονται τα επιθέματα. Στη δεύτερη φάση φορτώνονται στη μνήμη τόσο το προθεματικό δέντρο, όσο και όσα προθεματικά υποδέντρα έχουν διαπιστωθεί ότι έχουν σύνδεσμο επιθέματος σε αυτό και χωράνε στη μνήμη.

Πολυπλοκότητα: Η πολυπλοκότητα της κατασκευής προθεμάτων είναι πολύ μικρή, μόλις $O(log(|P|))$ στην πράξη, όπου P το πλήθος των προθεμάτων που δημιουργούνται. Στην χειρότερη περίπτωση όμως φτάνει το $O(n)$. Η πολυπλοκότητα του σταδίου κατασκευής υποδέντρων είναι στη χειρότερη περίπτωση $O(n^2)$. Η ακολουθία χωρίζεται σε κομμάτια το πλήθος των οποίων είναι το πολύ 2(n/B + 1). Για κάθε i-οστό κομμάτι του δένδρου (treeBlock) ελέγχουμε j κομμάτια εισαγωγής (indexBlock). Για κάθε ζευγάρι θα έχουμε το πολύ B πράξεις. Έτσι καταλήγουμε στην παραπάνω πολυπλοκότητα. Στη φάση ανάκτησης συνδέσμων η πολυπλοκότητα είναι $O(|P|n)$, καθώς τόσες είναι οι πράξεις φόρτωσης κομματιών της ακολουθίας εισόδου. Πέρα από αυτό οι πράξεις του αλγορίθμου απαιτούν σταθερό χρόνο. Συνολικά η προσέγγιση του Wavefront επιτυγχάνει να ευρετηριοποιήσει ακολουθίες εισόδου πολύ μεγαλύτερες από τις μέγιστες που καταφέρνουν άλλοι αλγόριθμοι. Χαρακτηριστικά αναφέρεται η μεγάλη πτώση απόδοσης που παρουσιάζει το Trellis σε είσοδο άνω των 1000 MB. Αντίθετα το Wavefront καταφέρνει σε υπερυπολογιστή 1024 επεξεργαστών να ευρετηριοποιήσει το ανθρώπινο γονιδίωμα των 3G ζευγαριών βάσεων σε μόλις 15 λεπτά.

2.4 Το περιβάλλον MapReduce

Σε αυτό το κεφάλαιο θα παρουσιάσουμε το περιβάλλον MapReduce που μας επιτρέπει να προγραμματίσουμε και να εκτελέσουμε με ευκολία παράλληλα προγράμματα.

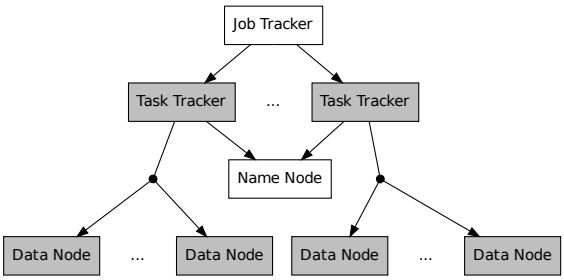
2.4.1 Εισαγωγή

To Apache Hadoop Map/Reduce είναι ένα framework γραμμένο σε Java το οποίο υποστηρίζει εφαρμογές παράλληλης επεξεργασίας μεγάλου όγκου δεδομένων. Η ιδέα προτάθηκε

αρχικά από την εταιρεία Google το 2004 [6], [7], αλλά η υλοποίηση δεν είναι διαθέσιμη. Το Hadoop είναι ένα σύνολο εφαρμογών που τρέχουν σε υπολογιστές συνδεδεμένους μεταξύ τους μέσω δικτύου (cluster υπολογιστών). Το Hadoop προσφέρει ένα επίπεδο αφαίρεσης στη διαδικασία ανάπτυξης παράλληλων προγραμμάτων, αναλαμβάνοντας να διαχειριστεί το διαμοιρασμό των δεδομένων, την συγκέντρωση των αποτελεσμάτων, τις πιθανές αποτυχίες κόμβων και άλλα θέματα που κανονικά θα χρειαζόταν ο προγραμματιστής να υλοποιήσει. Ο προγραμματιστής αρκεί να γράψει πρόγραμμα σε Java χρησιμοποιώντας βιβλιοθήκες του Hadoop και υλοποιώντας τον αλγόριθμο του σύμφωνα με τη λογική των συναρτήσεων Map και Reduce, έτσι ώστε να εκμεταλλευτεί την παράλληλη επεξεργασία που προσφέρει το cluster.

Το ίδιο το Hadoop αποτελείται εσωτερικά από τα εξής ενδιαφέροντα για την εφαρμογή μας κομμάτια:

1. Hadoop Common: Η υπηρεσία που δίνει πρόσβαση στα filesystems που υποστηρίζει το Hadoop.
2. Hadoop HDFS: Το filesystem του Hadoop.
3. Hadoop MapReduce: Η μηχανή του Hadoop που αναλαμβάνει να τρέξει παράλληλα το πρόγραμμα.



Σχήμα 2.6: η αρχιτεκτονική του Hadoop. Με λευκό σημειώνονται οι διεργασίες master και με γκρι οι διεργασίες slave

2.4.2 Hadoop HDFS

Το εσωτερικό filesystem του Hadoop είναι το HDFS. Τρέχει πάνω από το filesystem του λειτουργικού συστήματος και είναι σχεδιασμένο σύμφωνα με τη master/slave αρχιτεκτονική. Σκοπός του HDFS είναι να αποθηκεύει μεγάλο όγκο δεδομένων σε μεγάλης κλίμακας cluster. Παράλληλα έχει ανοχή σε αποτυχίες υλικού. Άλλα χαρακτηριστικά που κάνουν το HDFS ελκυστικό είναι η διατήρηση μεταστοιχείων για τα αρχεία (user ownership, permissions), η καταγραφή στοιχείων γειτνίασης των κόμβων, έτσι ώστε να επιταχύνονται οι μεταφορές δεδομένων, και ο μηχανισμός του rebalancer που ισοκατανέμει τα δεδομένα μεταξύ των κόμβων δεδομένων του cluster (Datanodes).

Επίπεδο Διεργασιών Hadoop: Τα δεδομένα είναι αποθηκευμένα στα **Datanodes** (slaves), χωρισμένα σε κομμάτια που λέγονται blocks. Τα blocks υπάρχουν σε πολλά αντίγραφα, ο αριθμός των οποίων καθορίζεται από τον παράγοντα επανάληψης των δεδομένων (replication factor). Το project προτείνει την ύπαρξη τουλάχιστον 3 αντιγράφων του κάθε block συνολικά στο cluster, προσδίδοντας έτσι στο σύστημα μεγάλη ανοχή σε αστοχίες υλικού (χαλασμένοι σκληροί δίσκοι) ή δικτύου (αποσύνδεση μέρους του δικτύου). Ο παράγοντας αυτός καθιστά επίσης αχρείαστες λύσεις τύπου RAID για την αποθήκευση δεδομένων, καθιστώντας την κατασκευή του cluster οικονομικά συμφέρουσα. Το HDFS απαιτεί και την παρουσία ενός **Namenode** (master). Το Namenode έχει το ρόλο διαχειριστή του filesystem. Κρατάει αρχείο με το ποιο block είναι διαθέσιμο από ποιο datanode, καθώς και διάφορα μεταδεδομένα για τα αρχεία, όπως file permissions. Εάν το Namenode αποτύχει, αυτόματα συνεπάγεται ότι το filesystem καταρρέει και σταματάει να λειτουργεί. Όμως η τελευταία έγκυρη κατάσταση του Namenode σώζεται από το μηχανισμό του Secondary Namenode. Κατά την επανεκκίνηση του, ο Namenode θα διαβάσει το log που κρατάει το Secondary Namenode και τίθεται στη σωστή κατάσταση, χωρίς απώλεια πληροφορίας.

2.4.3 Hadoop MapReduce

Το MapReduce είναι το σύστημα που ελέγχει την παράλληλη εκτέλεση του προγράμματος που υποβάλλει ο χρήστης στο cluster. Είναι και αυτό σχεδιασμένο σε master/slave αρχιτεκτονική. Είναι επιφορτισμένο με την ευθύνη του διαμοιρασμού των δεδομένων από το HDFS στους κόμβους που θα κάνουν την επεξεργασία, την ομαδοποίηση και επαναδιαμοιρασμό των ενδιάμεσων δεδομένων και τη συλλογή των τελικών αποτελεσμάτων. Επίσης διαχειρίζεται τις αποτυχίες των κόμβων, με αποτέλεσμα να είμαστε σίγουροι για την επιτυχία του συνόλου των εργασιών.

Επίπεδο Διεργασιών Hadoop: Ο **Jobtracker** (master) οργανώνει την εκτέλεση των εργασιών (jobs) που υποβάλλει ο χρήστης. Σκοπός του είναι να σπάσει τη συνολική εργασία σε μικρότερες, που ονομάζονται tasks, και να τις αναθέσει σε **TastTrackers** (slaves). Σε αυτή τη διαδικασία λαμβάνει υπόψη του την τοπολογία του δικτύου, έτσι ώστε τα δεδομένα να μεταφερθούν σε κοντινό κόμβο, ελαχιστοποιώντας έτσι την κίνηση στο δίκτυο. Για να γίνει η ανάθεση του task, ο JobTracker επικοινωνεί με τους Namenodes για να μάθει την ακριβή θέση των δεδομένων εισόδου της εργασίας. Η τακτική αυτή του προγράμματος αναφέρεται συχνά ως "είναι φτηνότερο να αναθέσεις την επεξεργασία εκεί που βρίσκονται τα δεδομένα, από το να τα μεταφέρεις". Για να καθορίσει κατά πόσο η διαδικασία της επεξεργασίας των δεδομένων προχωράει κανονικά, ο JobTracker δέχεται παλμούς από τους TaskTrackers ανά τακτά χρονικά διαστήματα. Σε περίπτωση που δεν δεκτεί παλμό από κάποιο TaskTracker, ο JobTracker χαρακτηρίζει FAILED το συγκεκριμένο task και το αναθέτει σε άλλο κόμβο. Οι αποτυχίες αντιμετωπίζονται ως ένα φυσιολογικό γεγονός και ο μόνος αντίκτυπος στην συνολική εργασία είναι η καθυστέρηση που προκαλείται λόγω της ανάγκης επανεπεξεργασίας των δεδομένων. Επίσης αν διαπιστωθεί ότι κάποια εργασία πηγαίνει πιο αργά από τις άλλες, αναθέτει το ίδιο task και σε δεύτερο TaskTracker. Κρατείται το αποτέλεσμα του node που

θα τελειώσει πρώτο ενώ η δεύτερη πιο αργή εκτέλεση τερματίζεται (KILLED). Η διαδικασία αυτή ονομάζεται υποθετική εκτέλεση (speculative execution) και είναι ουσιαστικά μια μέθοδος εξισορρόπησης της διαφορετικής απόδοσης των TaskTrackers είτε λόγω διαφορών στην ταχύτητα του υλικού, είτε λόγω πολυπλοκότητας της εισόδου. Έτσι αποφεύγεται το σενάριο στο οποίο ένας TaskTracker καθυστερεί συνολικά τη δουλειά ενώ οι υπόλοιποι έχουν τελειώσει και δεν κάνουν επεξεργασία. Όπως είναι φανερό, οι TaskTrackers (slaves) αναλαμβάνουν να τρέξουν τα tasks στα οποία έχει χωριστεί το job. Γνωστοποιούν στον JobTracker την επιτυχία ή αποτυχία της εκτέλεσης του task, καθώς επίσης του στέλνουν ανά τακτά χρονικά διαστήματα παλμό, δηλώνοντας την ορθή λειτουργία τους. Το επίπεδο διεργασιών του Hadoop φαίνεται στο σχήμα 2.6

2.4.4 Προγραμματιστικό Μοντέλο

Όπως αναφέραμε, ο προγραμματιστής καλείται να υλοποιήσει τον προς παραλληλοποίηση αλγόριθμο στη λογική δύο συναρτήσεων, της map και της reduce. Η γενική ιδέα είναι πως η map επεξεργάζεται είσοδο τύπου ζευγάρι (κλειδί, τιμή) ((key, value)) και δίνει ίδιας μορφής έξοδο. Τα αποτέλεσμά της ομαδοποιούνται με κριτήριο το κλειδί και δίνονται ως είσοδος στην reduce, έτσι ώστε κάθε κλήση της reduce να έχει είσοδο της μορφής (κλειδί, λίστα τιμών). Το αποτέλεσμα της Reduce είναι το τελικό ζητούμενο του προγράμματος.

Όπως αναφέραμε οι συναρτήσεις map και reduce που καλείται να υλοποιήσει ο προγραμματιστής είναι και οι δύο δομημένες να δέχονται είσοδο και να δίνουν έξοδο σε μορφή ζευγαριού τιμών κλειδιού και τιμής. Η συνάρτηση map παίρνει ζευγάρια δεδομένων και επιστρέφει μια λίστα από ζευγάρια δεδομένων.

```
Map :: (key1, value1) -> list (key2, value2)
```

Εφαρμόζεται παράλληλα σε κάθε κομμάτι που έχει διαχωριστεί η είσοδος, παράγοντας έτσι μια λίστα από ζευγάρια σε κάθε κλήση της. Έπειτα το MapReduce συγκεντρώνει όλα τα ζευγάρια από όλες τις λίστες, τις βάζει σε σειρά και τις ομαδοποιεί έτσι ώστε μια ομάδα να περιλαμβάνει όλες τις τιμές εξόδου για ένα συγκεκριμένο κλειδί. Η ομάδα αυτή αποτελεί είσοδο για τη συνάρτηση reduce, που θα τρέξει όπως είναι φυσικό σε διαφορετικούς υπολογιστές. Όπως βλέπουμε από το type annotation, δεν υπάρχει περιορισμός που να απαιτεί η έξοδος να έχει ίδιο τύπο με την είσοδο, είτε στο κλειδί, είτε στην τιμή. Η συνάρτηση reduce παίρνει ένα ζευγάρι που αποτελείται από κλειδί και λίστα τιμών και επιστρέψει μια λίστα από τιμές

```
Reduce :: (key2, list (value2)) -> list (value3)
```

Τα αποτελέσματα όλων των κλήσεων της reduce ομαδοποιούνται ως το τελικό αποτέλεσμα του MapReduce. Έτσι συνολικά το MapReduce έχει δεχτεί μια λίστα (κλειδί, τιμή) και έχει δώσει ως αποτέλεσμα μια λίστα απο τιμές. Και εδώ βλέπουμε ότι οι τιμές εξόδου μπορούν να έχουν διαφορετικό τύπο από τις τιμές εισόδου. Προαιρετικά ο προγραμματιστής χρησιμοποιεί και μια τρίτη συνάρτηση, τη συνάρτηση combine. Η συνάρτηση combine είναι μια συνάρτηση τύπου reduce που τρέχει τοπικά στον υπολογιστή που έτρεξε τη συνάρτηση map. Η κλήση της γίνεται πριν τα αποτελέσματα της map φύγουν για τη reduce μέσω δικτύου,

όσο είναι προσωρινά αποθηκευμένα στη μνήμη. Έτσι ένα ποσοστό του Reduce έχει γίνει χωρίς μεταφορά δεδομένων στο δίκτυο, έχοντας ως αποτέλεσμα γρηγορότερη επεξεργασία. Το υπολογιστικό μοντέλο που περιγράφηκε αποδεικνύεται ότι είναι Turing Complete. Έτσι οποιοδήποτε πρόβλημα έχει υπολογιστική λύση μπορεί να προσαρμοστεί στο μοντέλο Map και Reduce. Το παρακάτω είναι το τυπικό παράδειγμα που μετράει την συχνότητα εμφάνισης λέξεων (WordCount) σε διάφορα έγγραφα σε ψευδοκώδικα.

```
void map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    EmitIntermediate(w, ''1'');

void reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  int result = 0;
  for each pc in partialCounts:
    result += ParseInt(pc);
  Emit(AsString(result));
```

Το κάθε έγγραφο χωρίζεται σε λέξεις και η κάθε λέξη έχει αρχικά μια συχνότητα ίση με 1. Η ίδια η λέξη είναι το κλειδί με βάση το οποίο θα ομαδοποιηθούν οι συχνότητες. Άρα, η έξοδος του Map είναι της μορφής ((word1, 1), (word2, 1), (word1, 1)), ενώ η είσοδος του Reduce για το κλειδί word1 θα είναι (word1, (1,1)). Η reduce αρκεί να αθροίσει τη λίστα από τους άσσους για να βρει τη συνολική συχνότητα της λέξης. Η χρήση της συνάρτησης reduce ως συνάρτηση combine στον παραπάνω κώδικα είναι πολύ συνηθισμένη. Τα αποτελέσματα της map θα συγκεντρωθούν τότε σε λίστες στην κύρια μνήμη αντί για τον δίσκο. Η έξοδος του κόμβου προς τη reduce θα είναι τότε (word, συχνότητα στο συγκεκριμένο κομμάτι του είσοδο).

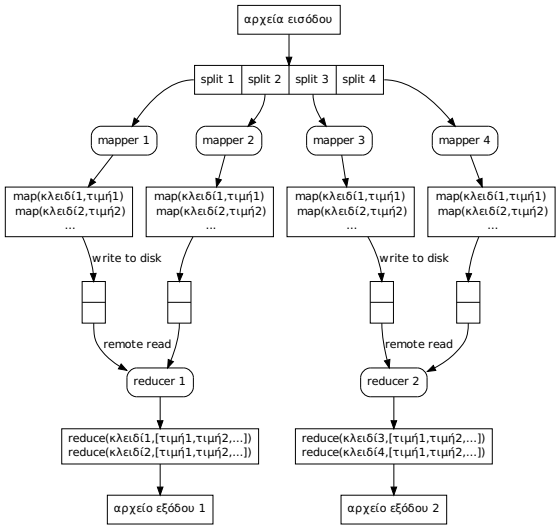
2.4.5 Επίπεδο Ροής Δεδομένων

Μπορούμε τώρα να εξετάσουμε συνολικά τη ροή δεδομένων κατά το τρέξιμο ενός MapReduce προγράμματος, ομαδοποιώντας τους κόμβους του δικτύου με τις συναρτήσεις map και reduce. Τα στάδια της ροής δεδομένων είναι:

1. **Input Reader** Ο Input Reader αναλαμβάνει να χωρίσει την είσοδο σε κατάλληλου μεγέθους κομμάτια που ονομάζονται splits και τυπικά είναι μεταξύ 16 και 128 MB. Κάθε κομμάτι ανατίθεται σε ένα και μόνο Mapper. Ο Input Reader έχει ως είσοδο δεδομένα από το HDFS και ως έξοδο παράγει ζευγάρια (key, value). Εσωτερικά αυτό το στάδιο είναι υλοποιημένο με δύο βήματα, το InputFormat που χωρίζει την είσοδο σε splits και το Record Reader, που χωρίζει το split σε ζευγάρια (κλειδί, τιμή).

2. **συνάρτηση Map** Ο κάθε κόμβος του δικτύου αναλαμβάνει έναν αριθμό Mappers, τυπικά τόσους όσοι και οι πυρήνες του μηχανήματος. Ο κάθε Mapper έχει μοναδικό κομμάτι εισόδου (split), το οποίο χωρίζεται σε ζευγάρια (κλειδί, τιμή). Κάθε ζευγάρι (κλειδί, τιμή) αντιστοιχεί σε μία κλήση της συνάρτησης Map για αυτόν τον Mapper. Τα αποτελέσματα του Mapper γράφονται στον δίσκο ταξινομημένα σύμφωνα με το κλειδί.
3. **συνάρτηση Partition** Η έξοδος από κάθε συνάρτηση map ανατίθεται στον κατάλληλο Reducer μέσω της συνάρτησης partition. Ως είσοδο παίρνει το κλειδί και τον αριθμό των Reducers και επιστρέφει το id του Reducer στον οποίο θα ανατεθεί το κλειδί αυτό.
4. **συνάρτηση Sorting Comparison** Η είσοδος για την κάθε συνάρτηση reduce μεταφέρεται από το μηχάνημα που έτρεξε η συνάρτηση map και ταξινομείται σύμφωνα με αυτή τη συνάρτηση.
5. **συνάρτηση Grouping Comparison** Η συνάρτηση αυτή αποφασίζει ποια θα είναι η είσοδος για μια κλήση της συνάρτησης reduce. Οι τρεις παραπάνω συναρτήσεις αντιπροσωπεύουν όλες μαζί τη φάση Shuffle, που δρα ανάμεσα στη φάση Map και τη φάση Reduce για να μεταφέρει σωστά την έξοδο της Map ως είσοδο στη Reduce.
6. **συνάρτηση Reduce** Ο κάθε κόμβος του δικτύου αναλαμβάνει έναν αριθμό από Reducers, τυπικά τόσους όσοι και οι πυρήνες του μηχανήματος. Ο κάθε Reducer έχει είσοδο μια λίστα από (κλειδί, λίστα τιμών). Η συνάρτηση reduce καλείται μία φορά για κάθε μοναδικό κλειδί που προέκυψε από την ταξινόμηση του προηγούμενου βήματος, προσπελαύνει όλα τα values που σχετίζονται με το συγκεκριμένο key και επιστρέφει μια λίστα από values.
7. **Output Writer** Ο Output Writer αναλαμβάνει να γράψει τα αποτελέσματα της φάσης Reduce στο HDFS.

Στο παραπάνω χρησιμοποιήθηκαν οι εκφράσεις "φάση Map", "Mapper" και "συνάρτηση map". Αξίζει να αναλυθεί η σημασία τους. Η φάση Map είναι το στάδιο του MapReduce στο οποίο όλα τα μηχανήματα-κόμβοι εκτελούν παράλληλα συναρτήσεις map. Ο κάθε κόμβος εκτελεί έναν αριθμό Mappers, με τον κάθε Mapper να αναλαμβάνει μοναδικό κομμάτι (split) της εισόδου, να το χωρίζει σε ζευγάρια (κλειδί, τιμή) και να καλεί τη συνάρτηση map για κάθε τέτοιο ζευγάρι. Αντίστοιχα συμβαίνει και στη φάση Reduce. Οι δύο αυτές φάσεις εκτελούνται σειριακά μεταξύ τους. Η κάθε μία ξεχωριστά εκτελείται παράλληλα από το Hadoop MapReduce στο cluster. Μετά τη φάση Map και πριν τη φάση Reduce εκτελείται μια ενδιάμεση φάση που αναλαμβάνει το σωστό διαμοιρασμό της εξόδου της φάσης Map στη φάση Reduce. Τα παραπάνω φαίνονται στο σχήμα 2.7



Σχήμα 2.7: η ροή δεδομένων σε ένα τυπικό MapReduce πρόγραμμα

Κεφάλαιο 3

Ο αλγόριθμος Trellis

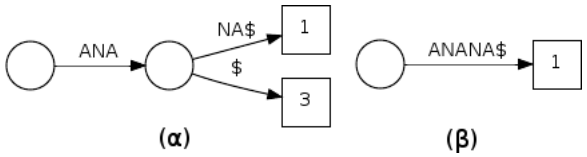
Στο προηγούμενο κεφάλαιο αναφερθήκαμε επιγραμματικά στον αλγόριθμο Trellis, τον πιο αποδοτικό αλγόριθμο κατασκευής δέντρων επιθεμάτων στον δίσκο. Σε αυτό το κεφάλαιο θα περάσουμε σε μια εις βάθος ανάλυση του αλγορίθμου αυτού.

3.1 Εισαγωγή

Ο αλγόριθμος Trellis [14] είναι ένας αλγόριθμος κατασκευής δέντρων επιθεμάτων βασισμένος στη χρήση σκληρού δίσκου και εξειδικευμένος για είσοδο που αποτελείται από μεγάλες ακολουθίες DNA. Αναλύσαμε τη δομή του δέντρου επιθεμάτων σε προηγούμενο κεφάλαιο. Όπως αναφέρθηκε η απαιτούμενη μνήμη για την αποθήκευση του δέντρου επιθεμάτων είναι πολύ μεγαλύτερη από την είσοδο που περιγράφει. Σε ένα πρόγραμμα βελτιστοποιημένο για ένα συγκεκριμένο αλφάβητο το μέγεθος του δέντρου επιθεμάτων είναι τουλάχιστον 10 φορές μεγαλύτερο από την είσοδο, ενώ σε μια γενική υλοποίηση έως και 50 φορές! Έτσι ακόμα και για είσοδο μερικών εκατοντάδων MB, το μέγεθος της εξόδου θα ξεπερνάει την χωρητικότητα της κύριας μνήμης και των πιο προηγμένων υπολογιστών σήμερα, με αποτέλεσμα την αποθήκευση των αποτελεσμάτων στο δίσκο (swapping). Η κατάσταση επιτείνεται από το γεγονός ότι οι αλγόριθμοι κατασκευής δέντρων επιθεμάτων έχουν πολύ μικρή χωρική τοπικότητα και κατασκευάζουν το δέντρο κάνοντας μη προβλέψιμα άλματα μεταξύ απομακρυσμένων κόμβων. Το αποτέλεσμα είναι κατακόρυφη μείωση της απόδοσης, καθώς απαιτείται συνεχόμενη επικοινωνία σκληρού δίσκου και κύριας μνήμης. Το πρόβλημα αυτό λύνει το Trellis μέσα από μια σειρά βημάτων που εξασφαλίζουν ότι το υπό κατασκευή δέντρο επιθεμάτων χωράει σίγουρα στη μνήμη. Για αυτό χρησιμοποιεί την έννοια του προθεματικού δέντρου επιθεμάτων (prefixed suffix tree) και του προθεματικού υποδέντρου επιθεμάτων (prefixed suffix subtree).

Το προθεματικό δέντρο επιθεμάτων είναι ένα δέντρο επιθεμάτων από τη ρίζα του οποίου ξεκινά μόνο μία ακμή, η ακμή που περιγράφει το πρόθεμα. Στο δέντρο αυτό περιγράφονται μόνο τα επιθέματα της εισόδου που ξεκινάνε με το συγκεκριμένο πρόθεμα. Το προθεματικό υποδέντρο επιθεμάτων είναι ένα προθεματικό δέντρο επιθεμάτων το οποίο αφορά μόνο ένα κομμάτι της εισόδου. Στο προθεματικό υποδέντρο επιθεμάτων περιγράφονται μόνο τα επιθέματα της εισόδου που ξεκινάνε με ένα συγκεκριμένο πρόθεμα και αρχίζουν σε ένα συγκεκριμένο

διάστημα της εισόδου. Τα παραπάνω φαίνονται για το BANANA, το πρόθεμα AN και το κομμάτι BAN φαίνεται στο Σχήμα 3.1.



Σχήμα 3.1: (α) το προθεματικό δέντρο επιθεμάτων του BANANA$ για το πρόθεμα AN.(β) το αντίστοιχο προθεματικό υποδέντρο για το κομμάτι BAN

Ο αλγόριθμος του Trellis υλοποιείται σε 4 κύρια βήματα .

1. Στάδιο Δημιουργίας Προθεμάτων Δημιουργεί προθέματα μεταβλητού μήκους με συχνότητα στην είσοδο κάτω από το κατώφλι.
2. Στάδιο Διαμερισμού Εισόδου Διαμερισμός της εισόδου σε κομμάτια μήκους ίσου με το κατώφλι και δημιουργία προθεματικών υποδέντρων επιθεμάτων
3. Στάδιο Συγχώνευσης Δέντρων Συγχώνευση των προθεματικών υποδέντρων επιθεμάτων με το ίδιο πρόθεμα σε προθεματικά δέντρα επιθεμάτων
4. Στάδιο ανάκτησης συνδέσμων επιθέματος Σε αυτό το στάδιο ανακτούνται οι σύνδεσμοι επιθέματος που χάθηκαν από τα προηγούμενα βήματα. Το στάδιο είναι προαιρετικό.

3.2 Στάδιο Δημιουργίας Προθεμάτων

Η ιδέα της χρήσης προθεμάτων έχει προταθεί από παλαιότερα. Κάποιες αρχικές προσπάθειες, όπως [9] που παρουσιάστηκε σε προηγούμενο κεφάλαιο, χρησιμοποίησαν προθέματα σταθερού μήκους. Όμως οι 4 βάσεις του DNA δεν έχουν την ίδια συχνότητα μέσα στα γονιδιώματα. Ενδεικτικά οι συχνότητα των βάσεων είναι 30%, 20%, 20%, 30% αντίστοιχα για τις βάσεις A, C, G, T. Το αποτέλεσμα είναι πως δεν υπάρχει ισορροπία στα δέντρα που προκύπτουν από τη χρήση προθεμάτων σταθερού μήκους, με κάποια δέντρα να είναι τόσο μικρά που σπαταλούνται πόροι για την επεξεργασία τους και άλλα τόσο μεγάλα που ξεπερνούν σε μέγεθος την κύρια μνήμη.

Το πρόβλημα αυτό λύνεται με την χρήση προθεμάτων μεταβλητού μήκους. Συγκεκριμένα το Trellis σαρώνει πολλές φορές το κείμενο, βρίσκοντας την συχνότητα όλων των δυνατών προθεμάτων μέχρι ένα ορισμένο μήκος. Κατόπιν κάνει μια διαλογή με βάση το κατώφλι. Τα προθέματα με συχνότητα μικρότερη από το κατώφλι κρατιούνται ως τελικά, ενώ τα υπόλοιπα χρησιμοποιούνται για την αναζήτηση στην επόμενη σάρωση. Από τα παραπάνω προκύπτει ότι κάθε πρόθεμα που ελέγχεται είτε κρατείται, είτε ελέγχονται τα 4 προθέματα μήκους $+1$ που το έχουν ως πρόθεμα. Δεν επιτρέπεται δηλαδή η παρουσία στην τελική λίστα 2 προθεμάτων που το ένα να έχει το άλλο ως πρόθεμα. Είναι χρήσιμο να σημειωθεί πως με χρήση κατωφλιού $t = 10^6$ και με είσοδο το ανθρώπινο γονιδίωμα μήκους 3Gbp (3G βάσεις) χρειάζονται μόλις δύο σαρώσεις, η πρώτη αναζητώντας προθέματα από 4 έως 8 χαρακτήρες και η δεύτερη από 8 έως 16. Μάλιστα υπάρχουν μόνο 2 προθέματα με συχνότητα πάνω από το t για μήκος 15.

Στον πίνακα 3.1 φαίνεται το μήκος και ο αριθμός προθεμάτων που ξεπερνούν σε συχνότητα το κατώφλι για το ανθρώπινο γονιδίωμα.

Μήκος	Αριθμός Προθεμάτων
1-3	Όλα
4	253
5	781
6	930
7	68
8-15	2

Πίνακας 3.1: Το μήκος και ο αριθμός προθεμάτων που ξεπερνούν σε συχνότητα το κατώφλι $t = 10^6$ για το ανθρώπινο γονιδίωμα.

Από την παραπάνω παρατήρηση, οι δημιουργοί του Trellis αποφάσισαν να αναζητούν σε κάθε στάδιο προθέματα μήκους μέχρι 2 φορές μεγαλύτερου από την προηγούμενη φάση. Όταν μια φάση τελειώσει, τα προθέματα μέγιστου μήκους που ξεπέρασαν σε συχνότητα το κατώφλι γίνονται είσοδος για την επόμενη φάση. Αντίθετα, αν κάποιο πρόθεμα μικρότερου μήκους έχει συχνότητα κάτω από το κατώφλι, κρατείται ως τελικό και όλα τα υπόλοιπα προθέματα που εξετάστηκαν έχοντας αυτό ως πρόθεμα, διαγράφονται. Συνολικά, τα προθέματα για το ανθρώπινο γονιδίωμα έχουν μήκος από 4 έως 16 χαρακτήρες και το πλήθος τους είναι περίπου 6400. Ακολουθεί το βήμα αυτό σε μορφή ψευδοκώδικα.

```
αλφάβητο = {A, C, G, T}
i = 4
παραγωγή όλων των προθεμάτων μήκους i έως i*2
κάνε
i = i*2
  για κάθε πρόθεμα p
    μέτρηση συχνότητας p στην είσοδο
  για κάθε πρόθεμα p
    αν η συχνότητα είναι μικρότερη του t
      p κρατείται ως τελικό
      σβήνονται όλα τα υπόλοιπα προθέματα στα οποία το p είναι πρόθεμα
    αλλιώς
      το p χρησιμοποιείται ως πρόθεμα για να παραχθούν προθέματα μήκους
      i έως i*2 για έλεγχο στο επόμενο loop
όσο υπάρχουν προθέματα με συχνότητα μεγαλύτερη από t
```

3.3 Στάδιο διαμερισμού εισόδου

Το Trellis διαιρεί την είσοδο σε $r = \lceil \frac{n+1}{t} \rceil$ κομμάτια, όπου n ο αριθμός χαρακτήρων της εισόδου και t το κατώφλι. Η επεξεργασία κάθε κομματιού μπορεί να γίνει στην κεντρική μνήμη,

καθώς ένα δέντρο επιθεμάτων με t χαρακτήρες χωράει πλήρως στη μνήμη. Για την κατασκευή των προθεματικών υποδέντρων επιθεμάτων χρησιμοποιείται ο αλγόριθμος του Ukkonen λόγω του γραμμικού χώρου και χρόνου που απαιτεί. Στο στάδιο αυτό δημιουργούνται, για κάθε κομμάτι μήκους t της εισόδου, τόσα προθεματικά υποδέντρα επιθεμάτων όσο και το πλήθος των προθεμάτων που έχει αποφασιστεί στο προηγούμενο στάδιο.

Ένα πρόβλημα που αντιμετώπισαν οι δημιουργοί του Trellis ήταν ότι η χρήση του αλγορίθμου του Ukkonen απαιτεί η συμβολοσειρά για την οποία κατασκευάζεται το ευρετήριο να τελειώνει σε $ έτσι ώστε να δώσει γνήσιο δέντρο επιθεμάτων. Όμως οι συμβολοσειρές εισόδου σε αυτό το στάδιο του Trellis δεν παρουσιάζουν τον πραγματικό τελευταίο χαρακτήρα, επειδή το τέλος του κάθε κομματιού έχει οριστεί από το κατώφλι t, και ο αλγόριθμος του Ukkonen θα δώσει πεπλεγμένα δέντρα επιθεμάτων. Η προσθήκη του χαρακτήρα $ στα πρώτα m-1 κομμάτια δημιουργεί προβλήματα γιατί οδηγεί τον Ukkonen σε τερματισμό χωρίς να είναι βέβαιο ότι κάθε θέση του κομματιού αντιστοιχεί σε φύλλο δέντρου. Για παράδειγμα αν έχω ως είσοδο την ακολουθία $AACGT|AACGT\$$ χωρισμένη με το χαρακτήρα | και πρόθεμα το $GTAA$ είναι φανερό ότι δεν θα προστεθεί ούτε στο suffix tree του πρώτου κομματιού, ούτε του δεύτερου. Η λύση σε αυτό το πρόβλημα είναι να διαβάζει το πρόγραμμα για το κάθε κομμάτι και κάποιους χαρακτήρες από το επόμενο κομμάτι στη σειρά, μέχρι ο αριθμός των φύλλων στο δέντρο να ισούται με το μήκος του κομματιού. Κατόπιν γίνεται χρήση του τερματικού συμβόλου ώστε να λάβουμε ένα πραγματικό δέντρο επιθεμάτων. Συνήθως οι επιπλέον χαρακτήρες που πρέπει να αναγνωστούν είναι λίγοι, το πολύ 1000 για την εφαρμογή με είσοδο το ανθρώπινο γονιδίωμ και $t = 10^6$. Αφού δημιουργηθεί το πλήρες δέντρο για το κομμάτι της εισόδου, πρέπει να εξαχθούν από αυτό τα προθεματικά υποδέντρα επιθεμάτων σύμφωνα με τη λίστα τελικών προθεμάτων που διαμορφώθηκε στο προηγούμενο στάδιο. Αυτό γίνεται με τον μηχανισμό ακριβούς αναζήτησης σε δέντρο επιθεμάτων που έχει αναφερθεί. Αναλυτικά, για κάθε πρόθεμα p και ξεκινώντας από τη ρίζα του δέντρου σχηματίζουμε μονοπάτι προς τα φύλλα, έτσι ώστε η συρροή του κειμένου των ακμών να μας δίνει το πρόθεμα p. Το προθεματικό υποδέντρο ορίζεται με ρίζα τον τελευταίο κόμβο που επισκεφθήκαμε, εξάγεται και αποθηκεύεται στον δίσκο για την επόμενη φάση. Στο τέλος αυτής της φάσης έχουμε στον σκληρό δίσκο $p \cdot r$ προθεματικά υποδέντρα επιθεμάτων, όπου p ο αριθμός των επιθεμάτων με συχνότητα μικρότερη από t συνολικά στην είσοδο και r ο αριθμός των κομματιών μήκους t που διαμερίστηκε η είσοδος. Ακολουθεί το βήμα αυτό σε μορφή ψευδοκώδικα.

```
για κάθε κομμάτι r
  tree = ukkonen(r)
  διάσχιση των κόμβων του δέντρου
    αν σχηματίζεται από το μονοπάτι τελικό πρόθεμα p τότε
      εξαγωγή προθεματικού υποδέντρου (tree, p)
```

3.4 Στάδιο συγχώνευσης δέντρων

Σε αυτό το στάδιο σκοπός του Trellis είναι να συγχωνεύσει τα προθεματικά υποδέντρα επιθεμάτων που έχουν παραχθεί στο προηγούμενο στάδιο και έχουν κοινό πρόθεμα, σε ένα προθεματικό δέντρο επιθεμάτων. Το τελικό δέντρο αποθηκεύεται στον σκληρό δίσκο. Το πρόγραμμα εκτελεί για κάθε πρόθεμα τη συγχώνευση των r προθεματικών υποδέντρων επιθεμάτων σε ένα προθεματικό δέντρο επιθεμάτων. Η διαδικασία της συγχώνευσης δουλεύει κάθε φορά σε δύο δέντρα, το δένδρο που δέχεται τις προσθήκες και θα αποτελέσει το τελικό δέντρο και ένα από τα προθεματικά υποδέντρα του προηγούμενου σταδίου. Ελέγχονται κάθε φορά δύο κόμβοι, αρχίζοντας με τις ρίζες. Οι ακμές του μικρού δέντρου που ξεκινάνε από βάση που δεν υπάρχει στη λίστα ακμών του κόμβου του μεγάλου δέντρου προστίθενται αυτούσιες (περίπτωση (a) στο σχήμα 3.2), ενώ αν βρεθούν ακμές που αρχίζουν από την ίδια βάση, γίνεται σύγκριση χαρακτήρων μέχρι να βρεθεί το μέγιστο κοινό πρόθεμα δύο ακμών. Τότε δημιουργείται στο σημείο εκείνο νέος κόμβος και προστίθεται η κατάλληλη ακμη από το προθεματικό υποδέντρο (περίπτωση (b) στο σχήμα 3.2). Η συγχώνευση των υποδέντρων απαιτεί τυχαία προσπέλαση στην ακολουθία εισόδου. Για να μην υπάρχουν μεγάλες καθυστερήσεις στο πρόγραμμα λόγω επικοινωνίας κύριας μνήμης και σκληρού δίσκου, απαιτείται το σύνολο της ακολουθίας να βρίσκεται στην κύρια μνήμη. Αυτό είναι εφικτό με την τεχνική του bit packing. Το αλφάβητο του DNA έχει μόνο 4 γράμματα-καταστάσεις, επιτρέποντάς μας να τις κωδικοποιήσουμε σε 2 bits. Έτσι οι απαιτήσεις μας σε χώρο υποτετραπλασιάζονται, καθώς 1 byte που κανονικά κρατάει 1 char, τώρα έχει κωδικοποιημένα 4. Με αυτό τον τρόπο ολόκληρο το ανθρώπινο γονιδίωμα των 3G βάσεων χωράει σε μόλις 750 MB. Ακολουθεί το βήμα αυτό σε μορφή ψευδοκώδικα.

```
για κάθε πρόθεμα
  για κάθε προθεματικό υποδέντρο
    για τα δύο rootNodes, mainNode και tmpNode
      για κάθε πρώτο χαρακτήρα ακμής που αρχίζει από το tmpNode
        αν ο χαρακτήρας δεν υπάρχει στο mainNode
          η ακμή έχει νέο αρχικό node το mainNode
        αν ο χαρακτήρας υπάρχει στο mainNode
          βρες το μέγιστο κοινό πρόθεμα των ακμών
          δημιούργησε νέο εσωτερικό κόμβο
          και πρόσθεσε τη νέα ακμή
```

3.5 Στάδιο ανάκτησης συνδέσμων επιθέματος

Οι σύνδεσμοι επιθέματος είναι απαραίτητοι για την απόδοση αλγορίθμων αναζήτησης σε δέντρα επιθεμάτων. Οι σύνδεσμοι επιθέματος δημιουργήθηκαν από τον αλγόριθμο του Ukkonen στο δεύτερο στάδιο, όμως λόγω της επεξεργασίας εξαγωγής των προθεματικών

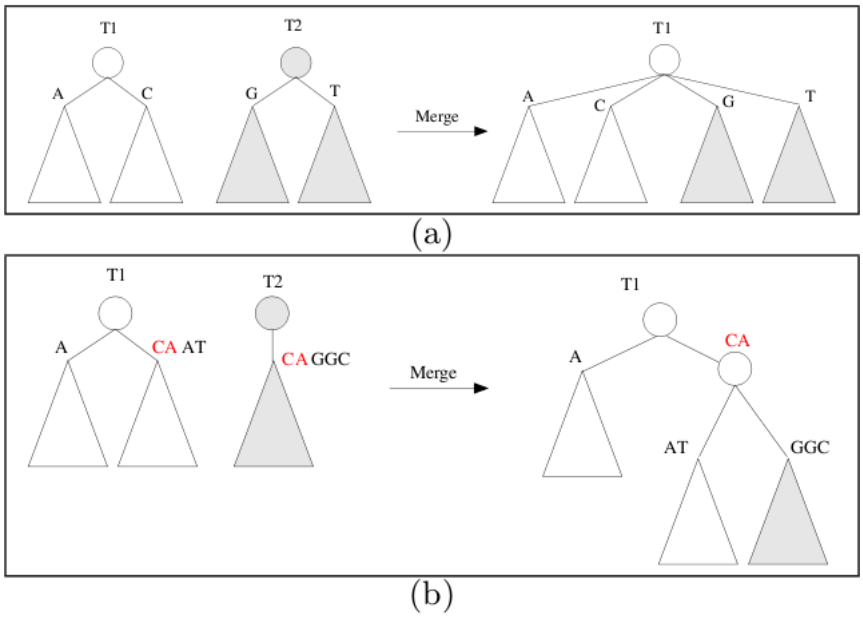
υποδέντρων χάθηκαν. Επίσης στα δέντρα προστέθηκαν νέοι κόμβοι κατά τη διάρκεια του τρίτου σταδίου. Το Trellis υποστηρίζει την ανάκτηση τους μετά το πέρας της κατασκευής των προθεματικών υποδέντρων. Διασχίζει το κάθε προθεματικό δέντρο κατά βάθος, ξεκινώντας από τη ρίζα του δέντρου. Για κάθε εσωτερικό κόμβο, εντοπίζει τον κόμβο στον οποίο έδειχνε ο σύνδεσμος επιθέματος κατά τη διάρκεια του αλγορίθμου του Ukkonen. Ο παλιός αυτός επιθεματικός κόμβος (Suffix Node) συνηθέστερα βρίσκεται σε κάποιο άλλο προθεματικό δέντρο το οποίο πρέπει να φορτωθεί στη μνήμη. Ο επιθεματικός κόμβος εντοπίζεται με την τεχνική skip and count και ένας νέος σύνδεσμος επιθέματος σχηματίζεται. Η κατά βάθος διάσχιση του δέντρου σημαίνει ότι τα προθέματα προσπελαύνονται με λεξικογραφική σειρά. Έτσι κάθε άλλο επιθεματικό δέντρο πέραν αυτού του οποίου κατασκευάζονται οι σύνδεσμοι επιθέματος φορτώνεται μία φορά.

Με το στάδιο ανάκτησης συνδέσμων επιθέματος τελειώνει η περιγραφή των βασικών βημάτων του Trellis. Στη συνέχεια θα αναλύσουμε κάποια κενά της παραπάνω περιγραφής, καθώς και την πολυπλοκότητα του αλγορίθμου.

3.6 Επιπλέον στοιχεία και πολυπλοκότητα

3.6.1 Επιλογή του κατάλληλου κατωφλιού

Όπως διαπιστώσαμε το κατώφλι επιτελεί δύο ρόλους στο Trellis. Σε πρώτη φάση δρα ως φίλτρο για την επιλογή προθεμάτων μεταβλητού μήκους κατάλληλης συχνότητας. Σε δεύτερη φάση, είναι το μήκος του κάθε κομματιού στο οποίο διαχωρίζεται η ακολουθία εισόδου. Η γενική ιδέα είναι ότι το δέντρο επιθεμάτων μιας συμβολοσειράς μήκους όσο το κατώφλι θα χωράει σίγουρα στην κύρια μνήμη. Έτσι, σύμφωνα με τα παραπάνω, ένα προθεματικό δέντρο



Σχήμα 3.2: Οι δύο περιπτώσεις συγχώνευσης στο Trellis

επιθεμάτων που το πρόθεμα του έχει συχνότητα στο κείμενο μέχρι t θα χωράει σίγουρα στη μνήμη. Επίσης, ένα δέντρο επιθεμάτων που αναπαριστά κομμάτι της εισόδου με μήκος ίσο με t πάλι θα χωράει σίγουρα στη μνήμη.

Ο υπολογισμός του κατωφλιού είναι συνάρτηση της διαθέσιμης κύριας μνήμης, του ποσοστού εσωτερικών κόμβων ως προς τον αριθμό των αριθμό φύλλων και του μεγέθους των δομών που τα υλοποιούν στο πρόγραμμα. Από μετρήσεις έχει βρεθεί ότι ο αριθμός εσωτερικών κόμβων σε σχέση με τα φύλλα για ένα δέντρο επιθεμάτων που περιγράφει ακολουθίες DNA είναι 70%. Ο αριθμός των φύλλων είναι ίσος με το μήκος της ακολουθίας εισόδου. Κατά τη φάση της συγχώνευσης πρέπει να έχουμε 2 δέντρα στην κύρια μνήμη, άρα χρειαζόμαστε περίπου $0.7t + 0.7t = 1.4t$. Η υλοποίηση του Trellis απαιτεί 40 bytes για κάθε εσωτερικό κόμβο και 16 bytes για κάθε φύλλο. Έτσι, για διαθέσιμη κύρια μνήμη M θα ισχύει:

$$M \geq \frac{n}{4} + [(1.4 \cdot 40) + 16] \cdot t \Rightarrow 0 \geq t \geq \frac{M - \frac{n}{4}}{72}$$

3.6.2 Ανάλυση υπολογιστικής πολυπλοκότητας

Στάδιο δημιουργίας επιθεμάτων: Όπως αναφέρθηκε, το στάδιο δημιουργίας επιθεμάτων δουλεύει σε φάσεις. Σε κάθε φάση σαρώνεται ολόκληρη η ακολουθία εισόδου και υπολογίζεται η συχνότητα για τα προθέματα μέχρι ένα συγκεκριμένο μήκος. Αν η συχνότητα κάποιου προθέματος ξεπερνά το κατώφλι, επιμηκύνεται και αποτελεί είσοδο στην επόμενη φάση της διαδικασίας. Έτσι στο τέλος έχουμε ένα σύνολο p διαφορετικών προθεμάτων μεταβλητού μήκους, που το καθένα εμφανίζεται στην είσοδο μέχρι t φορές.

Έστω l το πρόθεμα με το μεγαλύτερο μήκος στο σύνολο των προθεμάτων μίας φάσης. Αν το l είναι γνωστό από την αρχή και γίνεται μοναδική σάρωση της εισόδου S, για κάθε θέση της S θα πρέπει να αυξήσουμε την συχνότητα λ προθεμάτων. Τα λ προθέματα είναι οι υποσυμβολοσειρές μήκους 1 έως λ που ξεκινούν από τη δεδομένη θέση. Τότε η πολυπλοκότητα χρόνου θα είναι $O(nl)$ και η πολυπλοκότητα χώρου $(n + |\Sigma^{l+1}|)$, όπου Σ το αλφάβητο. Στην πράξη, λόγω των συνεχόμενων φάσεων του στάδιο και με τη χρήση του κατωφλιού, τα προθέματα που αναζητούνται είναι πολύ λιγότερα από $|\Sigma^i|$.

Στάδιο Διαμερισμού: Κατά τη φάση του διαμερισμού η ακολουθία σπάει σε $r = \lceil \frac{n+1}{t} \rceil$ κομμάτια και σε κάθε ένα από αυτά εφαρμόζεται ο αλγόριθμος του Ukkonen για να κατασκευαστεί ένα δέντρο επιθεμάτων. Αφού το κάθε κομμάτι έχει μήκος το πολύ t, χρειάζεται O(t) χρόνο και χώρο για να κατασκευαστεί. Η πολυπλοκότητα κατά μήκος όλων των διαμερίσεων είναι $r \cdot O(t) = O(n)$. Επίσης χρειάζεται μια πλήρης διάσχιση του δέντρου έτσι ώστε να εξαχθούν τα προθεματικά υποδέντρα. Όπως έχουμε αναφέρει, ο αλγόριθμος σαρώνει έναν αριθμό χαρακτήρων και μετά το τέλος της διαμέρισης. Ο αριθμός των χαρακτήρων που είναι απαραίτητος να διαβαστούν είναι μικρός λόγω της ψευδοτυχαίας κατανομής των βάσεων στο DNA και δεν επηρεάζει τη διαδικασία. Από άποψη λειτουργιών I/O, σε αυτό το στάδιο έχουμε $r \cdot p$ υποδέντρα, που το καθένα έχει μέγεθος $(\frac{t}{p})$. Τα υποδέντρα αυτά γράφονται στον δίσκο σε $O(rp)$ προσπελάσεις, αφού το πρόγραμμα διαβάζει και γράφει δεδομένα στο δίσκο σε ένα βήμα. Το γεγονός ότι τα δέντρα αυτά χωράνε σίγουρα στη μνήμη διευκολύνει στις εγγραφές και αναγνώσεις, καθώς δεν χρειάζεται κάποια στρατηγική buffering.

Στάδιο Συγχώνευσης: Στο στάδιο αυτό τα προθεματικά υποδέντρα όλων των διαμερίσεων συγχωνεύονται σε προθεματικά δέντρα σύμφωνα με το πρόθεμά τους. Για κάθε κλήση της συνάρτησης συγχώνευσης πρέπει να διασχιστούν 4 ακμές και να γίνει σύγκριση τόσων χαρακτήρων όσο και το μήκος του μέγιστου κοινού προθέματος (LCP) των ακμών που συγχωνεύονται. Έστω μ το μέσο μήκος LCP. Ο μέσος χρόνος για τη συγχώνευση θα είναι 4μ = O(μ). Ο συνολικός χρόνος συγχώνευσης είναι O(μn), για το σύνολο των προθεματικών υποδέντρων, αφού το πλήθος των κόμβων και των ακμών στο πλήρες δέντρο φράσσεται από το O(n). Αφού στη χειρότερη περίπτωση μ = O(n), η συνολική χρονική πολυπλοκότητα της συγχώνευσης είναι $O(n^2)$. Όμως το πραγματικό μέσο μήκος του LCP είναι σε αυτή τη φάση περίπου 30 χαρακτήρες, δηλαδή κοντά στο logn. Έτσι η πολυπλοκότητα σε αυτό το στάδιο είναι O(nlogn) και στην χειρότερη περίπτωση $O(n^2)$. Η συνολική χωρική πολυπλοκότητα της φάσης συγχώνευσης παραμένει γραμμική O(n), αφού υπάρχουν O(n) κόμβοι στο πλήρες δέντρο και ο χώρος που απαιτεί η ακολουθία εισόδου είναι O(n). Η ακολουθία εισόδου θα σαρωθεί αρκετές φορές, καθώς σε κάθε πράξη συγχώνευσης πρέπει να διαβαστούν κατά μέσο όρο μ χαρακτήρες.

Στάδιο Ανάκτησης Συνδέσμων Επιθέματος: Στο στάδιο αυτό ανακτώνται οι σύνδεσμοι επιθεμάτων για όλους τους O(n) εσωτερικούς κόμβους των προθεματικών υποδέντρων. Σε κάθε εσωτερικό κόμβο γίνεται διάσχιση μιας ακμής προς τα πάνω, έτσι ώστε να βρεθεί ο κόμβος-γονέας που καταλήγει ο σύνδεσμος επιθέματος. Στη συνέχεια ακολουθείται ο σύνδεσμος με την τεχνική skip and count στο κόμβο του προθεματικού δένδρου που αντιστοιχεί ο σύνδεσμος επιθέματος. Η διάσχιση κόμβων προς τα επάνω και η εγγραφή του συνδέσμου επιθέματος χρειάζονται γραμμικό χρόνο. Επίσης αποδεικνύεται ότι τα skip and count βήματα κατά τη δημιουργία δέντρου είναι το πολύ 3n = O(n). Άρα η συνολική χρονική πολυπλοκότητα για την ανάκτηση των συνδέσμων επιθέματος είναι O(n). Η ανάκτηση των συνδέσμων απαιτεί την παρουσία 2 προθεματικών δέντρων για τα οποία γίνεται η ανάκτηση συνδέσμων στη μνήμη. Στη χειρότερη περίπτωση κάθε προθεματικό δέντρο μπορεί να διαβαστεί p φορές, μία για κάθε πρόθεμα μεταβλητού μήκους. Έτσι στη χειρότερη περίπτωση το κόστος είναι O(p). Στην πράξη όμως απαιτούνται πολύ λίγα φορτώματα από τον σκληρό δίσκο, μειώνοντας το πραγματικό κόστος I/O. Συνολικά το Trellis έχει στη χειρότερη περίπτωση υπολογιστικό κόστος $O(n^2)$, ενώ σε πραγματική λειτουργία $O(nlogn)$. Η χωρική πολυπλοκότητα παραμένει πάντα $O(n)$.

Κεφάλαιο 4

Υλοποίηση Μεθόδου Trellis στο Περιβάλλον MapReduce

Στα προηγούμενα κεφάλαια αναλύθηκε η χρησιμότητα των δέντρων επιθεμάτων και καταδείχθηκαν οι δυσκολίες που υπάρχουν στην κατασκευή τους όταν αυτά περιγράφουν πολύ μεγάλη είσοδο. Περιγράφηκε με λεπτομέρεια ο αλγόριθμος Trellis, η αποδοτικότερη σήμερα μέθοδος κατασκευής δέντρων επιθεμάτων στον σκληρό δίσκο. Επίσης περιγράφηκε το μοντέλο προγραμματισμού και περιβάλλον εκτέλεσης παράλληλων προγραμμάτων MapReduce. Σε αυτό το κεφάλαιο θα αναλυθεί πώς η βασική ιδέα για τον αλγόριθμο Trellis αποτέλεσε τη βάση μιας υλοποίησης παράλληλης κατασκευής δέντρων επιθεμάτων προγραμματισμένης σε MapReduce.

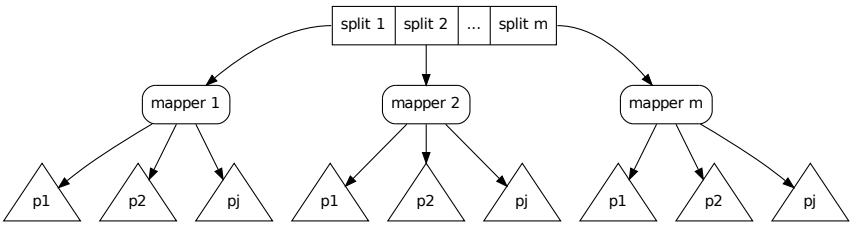
4.1 Εισαγωγή

Στο κεφάλαιο αυτό θα αναλύσουμε πώς προσαρμόστηκε η κεντρική ιδέα της μεθόδου του Trellis στο προγραμματιστικό μοντέλο του MapReduce. Όπως είδαμε στο κεφάλαιο του Trellis, ο αλγόριθμος αυτός είναι ιδιαίτερα αποτελεσματικός στην κατασκευή μεγάλων δέντρων επιθεμάτων. Όπως αναφέρθηκε, ο αλγόριθμος απαιτεί τον χωρισμό της εισόδου σε κομμάτια. Στη συνέχεια, οι εργασίες οργανώνονται σε δύο βασικά βήματα: (α) την κατασκευή προθεματικών υποδένδρων επιθεμάτων για κάθε κομμάτι και (β) η συγχώνευση των αντίστοιχων προθεματικών υποδένδρων από τα διάφορα κομμάτια σε ένα υποδένδρο για κάθε πρόθεμα. Τα δύο βήματα μπορεί να σχετίζονται μεταξύ τους, όμως είναι κατά βάση ανεξάρτητα. Πράγματι, η επεξεργασία του κάθε κομματιού για την παραγωγή προθεματικών υποδέντρων είναι ανεξάρτητη από όλα τα υπόλοιπα κομμάτια. Επίσης, από τη στιγμή που έχουν παραχθεί όλα τα υποδέντρα για ένα πρόθεμα, τότε ανεξάρτητα από όλα τα άλλα υποδέντρα μπορούμε να συγχωνεύσουμε τα υποδέντρα αυτά. Άρα μπορούμε να αντιστοιχίσουμε ένα κομμάτι εισόδου σε μια εργασία κατασκευής προθεματικών υποδέντρων και όλα τα υποδέντρα ενός προθέματος σε μια εργασία συγχώνευσης υποδέντρων σε δέντρα. Όπως είναι φανερό, οι εργασίες αυτές μπορούν να ανατεθούν σε διαφορετικούς υπολογιστές. Συγκεκριμένα έγινε χρήση της τεχνολογίας MapReduce. Ένα πρόγραμμα MapReduce χωρίζεται σε δύο φάσεις, με τα αποτε-

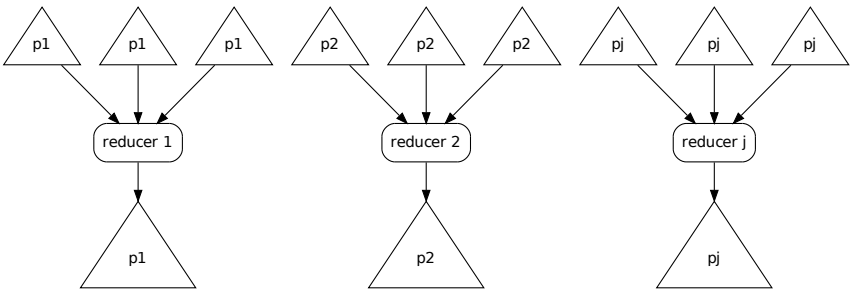
λέσματα της πρώτης να ομαδοποιούνται σύμφωνα με κάποιο χαρακτηριστικό και να αποτελούν είσοδο στη δεύτερη φάση. Οι δύο φάσεις είναι η map και η reduce, με τα ενδιάμεσα δεδομένα να είναι εκφρασμένα σε ζευγάρια. Συνολικά λοιπόν η λογική του Trellis φαίνεται να μπορεί να προσαρμοστεί στο προγραμματιστικό μοντέλο MapReduce ως εξής:

1. **Φάση map:** Δίνεται είσοδος κομμάτι της ακολουθίας εισόδου, κατασκευάζονται τα προθεματικά υποδέντρα επιθεμάτων για το κομμάτι και δίνονται ως έξοδος ζευγάρια (πρόθεμα, προθεματικό υποδέντρο επιθεμάτων)
2. **Φάση reduce:** Τα προθεματικά υποδέντρα επιθεμάτων με ίδιο πρόθεμα (κλειδί) ομαδοποιούνται και αποτελούν είσοδο σε μια κλήση της reduce, η οποία δίνει συνολική έξοδο τα προθεματικά δέντρα επιθεμάτων για όλη την ακολουθία εισόδου.

Τα παραπάνω φαίνονται σχηματικά και στα σχήματα 4.1 για τη φάση map και 4.2 για τη φάση reduce.



Σχήμα 4.1: Η φάση map του προγράμματος, η είσοδος χωρίζεται σε κομμάτια και από το κάθε κομμάτι παράγονται υποδέντρα, ένα για κάθε πρόθεμα



Σχήμα 4.2: Η φάση reduce του προγράμματος, τα υποδέντρα με το ίδιο πρόθεμα ενώνονται σε προθεματικά δέντρα

Από την παραπάνω περιγραφή απουσιάζει η πρώτη φάση του Trellis που είναι ο υπολογισμός των προθεμάτων με συχνότητα μέχρι t στην ακολουθία εισόδου. Αυτή η φάση του Trellis μπορεί να μοντελοποιηθεί κατά MapReduce ως μια εφαρμογή του παραδείγματος WordCount:

1. **Φάση map** Δίνεται είσοδος κομμάτι της ακολουθίας εισόδου και λαμβάνονται ως έξοδος ζευγάρια (πρόθεμα, 1)
2. **Φάση reduce** Συγκεντρώνονται όλα τα ζευγάρια με το ίδιο πρόθεμα (κλειδί) και αθροίζονται οι τιμές (1) ώστε να βρεθεί η συχνότητα του προθέματος στην ακολουθία εισόδου.

Πριν προχωρήσουμε στην ανάλυση της υλοποίησης μας, είναι χρήσιμο να περιγραφεί ένα γενικό πρότυπο πρόγραμμα MapReduce σε επίπεδο κώδικα.

```
public static void main(String args[]) throws Exception {
  Configuration conf = new Configuration();
  conf.setProperty(property, default value);
  Job job = new job(conf);
  job.setJobName(''Name of Job'');

  job.setMapperClass(MyMapper.class);
  job.setReducerClass(MyReducer.class);

  job.setMapOutputKeyClass(TypeA.class);
  job.setMapOutputValueClass(TypeB.class);
  job.setOutputKeyClass(TypeC.class);
  job.setOutputValueClass(TypeD.class);

  job.setInputFormatClass(MyInputFormat.class);
  job.setOutputFormatClass(MyOutputFormat.class);
  MyInputFormat.addInputPath(job, path_in);
  MyOutputFormat.setOutputPath(job, path_out);

  job.waitForCompletion(true);
}
```

Στο παραπάνω καταγράφεται μια τυπική διάταξη της συνάρτησης main σε ένα πρόγραμμα MapReduce. Αρχικά κατασκευάζουμε ένα νέο αντικείμενο τύπου Configuration. Σε αυτό μπορούμε να ορίσουμε τις τιμές διάφορων ιδιοτήτων της εκτέλεσης. Στη συνέχεια κατασκευάζουμε ένα αντικείμενο Job. Το αντικείμενο Job είναι ο πυρήνας της εργασίας μας. Σε αυτό ορίζουμε το όνομα της εργασίας, την κλάση που υλοποιεί τον Mapper και την κλάση που υλοποιεί τον Reducer, καθώς και τους τύπους εξόδου της map και της reduce. Πρέπει επίσης να συγκεκριμενοποιήσουμε τη μορφή των δεδομένων εισόδου (InputFormat), καθώς και τη διαδρομή στο HDFS από όπου αυτά θα διαβαστούν. Με την κλάση InputFormat, το MapReduce γνωρίζει πώς να χωρίσει την είσοδο σε κομμάτια (splits) και να εκκινήσει αντίστοιχο αριθμό από Mapper Tasks. Ακόμα, με την κλάση RecordReader που συνδέεται άμεσα με την κλάση InputFormat, το MapReduce γνωρίζει πως να χωρίσει ένα κομμάτι σε ζευγάρια

(κλειδί, τιμή) (records) που αποτελούν είσοδο κατά την κλήση της συνάρτησης map. Επίσης ορίζουμε τη μορφή των δεδομένων εξόδου (OutputFormat), καθώς και τη διαδρομή στο HDFS που αυτά θα εγγραφούν. Στο τέλος τρέχουμε την εργασία στο cluster, υποδεικνύοντας στη συνάρτηση main να περιμένει την ολοκλήρωση της εκτέλεσης της εργασίας.

```
public class MyMapper
  extends Mapper<TypeA, TypeB, TypeC, TypeD> {

  public void setup (Context context)
    throws IOException, InterruptedException {
  }

  public void map(TypeA key, TypeB value, Context context)
    throws IOException, InterruptedException {
    do_stuff(key,value)
    TypeC output_key = ...;
    TypeD output_value = ...;
    context.write(output_key, output_value);
  }
}
```

Στο παραπάνω καταγράφεται μια τυπική διάταξη της κλάσης Mapper σε ένα πρόγραμμα MapReduce. Παρατηρούμε ότι πρέπει να δηλώσουμε τους τύπους εισόδου και τους τύπους εξόδου της συνάρτησης. Στη συνάρτηση setup καταγράφουμε εντολές που θέλουμε να εκτελεσθούν πριν τις κλήσεις των συναρτήσεων map. Αντίστοιχα μπορεί να γραφεί και η συνάρτηση cleanup που τρέχει μετά το τέλος των εκτελέσεων της συνάρτησης map. Στη συνάρτηση map βλέπουμε ότι η τελική μας πράξη πρέπει να είναι η context.write η οποία δίνει τα αποτελέσματα της συνάρτησης map στο MapReduce. Αντίστοιχη διάταξη έχει και η συνάρτηση Reduce.

Εδώ αξίζει να σημειώσουμε μια λεπτομέρεια για τους τύπους που χρησιμοποιούνται ως κλειδιά και τιμές. Οι κλάσεις που χρησιμοποιούνται ως τύποι εισόδου, εξόδου και ενδιάμεσων δεδομένων θα πρέπει να υλοποιούν κάποιες απαραίτητες μεθόδους. Έτσι για να χρησιμοποιηθεί μια κλάση ως κλειδί ή τιμή πρέπει να ορίζει τις μεθόδους write και readFields. Αυτές αναλαμβάνουν να σειριοποιήσουν το αντικείμενο στο δίσκο και να το φορτώσουν στη μνήμη όποτε χρειαστεί, χωρίς να παρεμβληθεί μια ενδιάμεση μορφή δεδομένων, όπως η XML. Οι συναρτήσεις αυτές καλούνται για την ανάγνωση της εισόδου της map, την αποθήκευση των αποτελεσμάτων της στο δίσκο, την ανάγνωση της εισόδου της reduce και την αποθήκευση των αποτελεσμάτων της στο HDFS. Επιπλέον των παραπάνω μεθόδων και ειδικά για τις κλάσεις που θα είναι κλειδί πρέπει να γραφεί μια μέθοδος comparator. Αυτή η μέθοδος χρησιμοποιείται από το MapReduce στο ενδιάμεσο βήμα Shuffle για να ομαδοποιήσει τις τιμές ανά κλειδί και έτσι μια κλήση της συνάρτησης reduce να λάβει όλες τις τιμές που αντιστοιχούν στο συγκεκριμένο κλειδί. Επίσης αξίζει να διευκρινιστεί ότι η είσοδός μας είναι ένα αρχείο με

μια ακολουθία DNA. Οι 4 βάσεις είναι κωδικοποιημένες με τους αριθμούς 1, 2, 3, 4 και η ακολουθία δεν χωρίζεται σε γραμμές αλλά είναι ενιαία σε μια σειρά.

4.2 Δημιουργία προθεμάτων

Σε αυτό το κεφάλαιο θα περιγράψουμε πώς υλοποιήθηκε το στάδιο δημιουργίας προθεμάτων του Trellis στο MapReduce. Στο στάδιο αυτό θέλουμε να δημιουργήσουμε όλα τα προθέματα μεταβλητού μήκους που έχουν συχνότητα μικρότερη από το κατώφλι. Για το σκοπό αυτό αναθέτουμε σε μια εργασία MapReduce να σαρώσει συνολικά όλη την ακολουθία εισόδου για μια λίστα από προθέματα και να μας επιστρέψει τη συχνότητα των προθεμάτων. Μια πρώτα παρατήρηση είναι πώς το κομμάτι αυτό δεν μπορεί να υλοποιηθεί μόνο από μία εργασία MapReduce. Πράγματι, υπάρχει εξάρτηση δεδομένων μεταξύ της μέτρησης των συχνοτήτων των προθεμάτων μήκους length και της μέτρησης των συχνοτήτων των προθεμάτων length+1. Αυτό συμβαίνει γιατί είναι απαραίτητο να μετρηθούν οι συχνότητες μόνο κάποιων προθεμάτων length+1 και όχι όλων όσων μπορούν να δημιουργηθούν από το αλφάβητο για μήκος length+1. Συγκεκριμένα αναζητούμε μόνο τα προθέματα μήκους length+1 που έχουν ως πρόθεμα μήκους length ένα πρόθεμα που είχε συχνότητα εμφάνισης μέσα στο κείμενο περισσότερο από το κατώφλι. Το MapReduce δεν προσφέρει κάτι συγκεκριμένο για την αντιμετώπιση αυτών των καταστάσεων. Η προτιμότερη λύση είναι το επαναληπτικό τρέξιμο διαφορετικών εργασιών, έτσι ώστε κάθε φορά να αναπροσαρμόζεται η είσοδος της επόμενης εργασίας σύμφωνα με την έξοδο της προηγούμενης.

Έπειτα θα πρέπει να αποφασίσουμε πώς θα παραλληλοποιήσουμε την εργασία. Μια αρχική σκέψη θα ήταν να δημιουργήσουμε ένα Task για κάθε κομμάτι του κειμένου και κάθε υποψήφιο πρόθεμα. Η map θα μετρούσε τότε τη συχνότητα ενός προθέματος σε ένα κομμάτι και η reduce θα άθροιζε τις συχνότητες του προθέματος για όλα τα κομμάτια για να δώσει έξοδο τη συχνότητα του προθέματος συνολικά στο κείμενο. Από τη στιγμή που και το κείμενο και το πρόθεμα είναι εκφρασμένα με κλάσεις του MapReduce, θα περιμέναμε να δίνεται η δυνατότητα διαμερισμού και των δύο εισόδων. Κάτι τέτοιο όμως δεν υποστηρίζεται αυτή τη στιγμή, καθώς δεν υπάρχει κατάλληλο InputFormat και αντίστοιχα RecordReader. Είμαστε αναγκασμένοι να προγραμματίσουμε την εργασία χωρίζοντας τη μία είσοδο και μόνο. Αρχικά υπήρχαν σκέψεις πως αυτό μπορεί να περιορίσει την παραλληλοποίηση του προγράμματος, καθώς δεν μας δίνεται η δυνατότητα να χωρίσουμε τη διαδικασία σε μικρότερα κομμάτια. Λύσεις που ελέγχθηκαν σε αυτό το στάδιο ήταν η περίπτωση του να δημιουργηθεί νέας μορφής είσοδος με το καρτεσιανό γινόμενο των δύο εισόδων. Για το παράδειγμά μας αυτό αντιστοιχεί σε ένα αρχείο εισόδου από ζευγάρια (πρόθεμα, κομμάτι ακολουθίας). Η λύση αυτή έχει δύο κύρια μειονεκτήματα: απαιτεί προεπεξεργασία των προθεμάτων και του κειμένου για να βρεθούν στην κατάλληλη μορφή και χρειάζεται πολλαπλάσιο χώρο από την πραγματική πληροφορία που αποθηκεύει, περίπου ίση με το πλήθος των προθεμάτων p επί τους χαρακτήρες της ακολουθίας εισόδου n. Άλλη λύση που προτάθηκε ήταν η απλοποίηση της παραπάνω μεθόδου ώστε τα κομμάτια της ακολουθίας εισόδου να μην αποθηκεύονται, αλλά να παρέχεται δείκτης στο κατάλληλο σημείο του αρχείου που αποθηκεύει την ακολουθία. Τέλος, προτάθηκε

και η δημιουργία μιας αλυσίδας από εργασίες MapReduce, έτσι ώστε στην πρώτη εργασία να γίνεται καταμερισμός των προθεμάτων και στη δεύτερη καταμερισμός της ακολουθίας. Ούτε αυτή η λύση χρησιμοποιήθηκε μιας και στην πράξη το κόστος της έναρξης καινούργιων εργασιών είναι πολύ μεγαλύτερο από τα οφέλη. Στην πραγματικότητα ο περιορισμός αυτός του MapReduce δεν έχει επιπτώσεις στο πόσο παράλληλο μπορεί να είναι το πρόγραμμά μας. Κάθε cluster έχει ένα όριο Tasks που μπορεί να εκτελέσει ταυτόχρονα και αυτό ορίζεται από το συνολικό άθροισμα των επεξεργαστών των κόμβων του cluster. Κάθε πυρήνας όταν τελειώσει επιτυχημένα ένα Task, αναλαμβάνει ένα επόμενο. Επίσης, σε περίπτωση που το πρόγραμμά μας τρέξει σε μεγαλύτερο cluster, μπορούμε απλά να αυξήσουμε τον αριθμό κομματιών που χωρίζεται η είσοδος για να εκμεταλλευτούμε τους επιπλέον πυρήνες. Από τη στιγμή που το MapReduce επιτρέπει τον χωρισμό μίας εισόδου και μόνο, θα πρέπει να αποφασίσουμε με βάση ποια είσοδο θα παραλληλοποιήσουμε τη διαδικασία. Το κριτήριο για αυτή την επιλογή είναι πάντα το μέγεθος της εισόδου, άρα θα αναθέσουμε στο MapReduce να χωρίσει την ακολουθία εισόδου. Τίθεται το ερώτημα πώς θα δώσουμε στους κόμβους πρόσθετα δεδομένα που είναι απαραίτητα για αυτό το στάδιο, όπως τη λίστα προθεμάτων που πρέπει να ελεγχθεί η συχνότητά τους. Αυτό μπορεί να γίνει με δύο τρόπους:

1. Αν πρόκειται για απλή τιμή αριθμού ή συμβολοσειράς που είναι αναγκαίο να γνωρίζει μια κλάση, μπορεί να προγραμματιστεί μια ιδιότητα και να τεθεί η τιμή της μέσω του αντικειμένου Configuration.
2. Αν πρόκειται για πιο σύνθετες δομές δεδομένων, τότε μπορεί να χρησιμοποιήσει ο μηχανισμός του Distributed Cache. Το Distributed Cache είναι μια μέθοδος με την οποία πριν την εκτέλεση μιας εργασίας, μεταφέρονται αρχεία από το HDFS στο τοπικό φάκελο προσωρινών δεδομένων (temporary folder) του κάθε κόμβου.

Κοινό χαρακτηριστικό και των δύο μεθόδων είναι ότι η ροή δεδομένων είναι ορισμένη μονοσήμαντα, από τον JobTracker (master) προς τους TaskTrackers. Δεν δίνεται η δυνατότητα δηλαδή με κάποια από αυτές τις μεθόδους να ληφθούν επιπλέον αποτελέσματα από τους κόμβους.

Σύμφωνα με τα παραπάνω, η υλοποίηση του σταδίου δημιουργίας προθεμάτων μεταβλητού μήκους υλοποιηθήκε ως εξής: Αρχικά παράγονται όλα τα προθέματα μήκους 4 του αλφαβήτου και εγγράφονται σε μια λίστα. Στη συνέχεια ορίζεται μια νέα εργασία για το Hadoop έτσι ώστε να μετρηθούν οι συχνότητες όλων των προθεμάτων που είναι αποθηκευμένα στη λίστα. Η λίστα δίνεται σε όλους τους κόμβους μέσω του μηχανισμού Distributed Cache. Τα προθέματα και οι συχνότητες επιστρέφονται σε ένα αντικείμενο τύπου MapFile, το οποίο διαβάζεται. Τα προθέματα που έχουν συχνότητα μη μηδενική και ταυτόχρονα μικρότερη από το κατώφλι κρατιούνται σε μια λίστα τελικών προθεμάτων, ενώ για κάθε ένα από τα υπόλοιπα παράγονται 4 νέα προθέματα με πρόσθεση των 4 χαρακτήρων του αλφαβήτου στο τέλος του προθέματος. Τα επιμηκυμένα αυτά προθέματα εισάγονται σε μια νέα λίστα και δημιουργείται μια νέα εργασία για να καταγράψει τις συχνότητες τους. Η διαδικασία επαναλαμβάνεται μέχρι να διαπιστωθεί ότι κανένα από τα προθέματα δεν ξεπέρασε το κατώφλι. Τα παραπάνω εκφράσμένα σε ψευδοκώδικα:

$$...|\underbrace{AACGTGCTTAG\overbrace{C|GGC}}_{\text{είσοδος mapper}}ATTCAGCAT|...$$

Σχήμα 4.3: Με | σημειώνεται το σημείο που χωρίζεται το κείμενο σε κομμάτια. Ελέγχοντας για προθέματα μήκους 4, παρατηρούμε ότι πρέπει να διαβάσουμε και 3 χαρακτήρες από το επόμενο κομμάτι για το τελευταίο προς έλεγχο πρόθεμα που σημειώνεται.

```
threshold;
length=4
παραγωγή όλων των προθεμάτων με μήκος 4 και αποθήκευση στη λίστα
όσο υπάρχουν προθέματα με μήκος μεγαλύτερο από threshold
  τρέξε μια νέα καταμέτρηση συχνοτήτων των προθεμάτων της λίστας
  για κάθε πρόθεμα p του οποίου καταμετρήθηκε η συχνότητα
    αν η συχνότητα είναι μικρότερη του threshold
    και διάφορη του μηδενός
      κράτησε το πρόθεμα ως τελικό
    αλλιώς
      παραγωγή 4 νέων προθεμάτων μήκους length+1 με πρόθεμα το p
      και πρόσθεσή τους στη λίστα
  length++
```

Η εργασία που ορίζουμε χρησιμοποιεί μια ειδική κλάση InputFormat η οποία είναι εξειδικευμένη για αρχεία εισόδου που όλη η πληροφορία παρουσιάζεται συνεχόμενη χωρίς κενά ή αλλαγή γραμμής. Η κλάση αυτή τροποποιήθηκε έτσι ώστε να μπορεί ο χρήστης να ορίζει εξωτερικά πόσο μεγάλα θα είναι τα κομμάτια στα οποία θα χωριστεί η είσοδος και πόσους επιπλέον χαρακτήρες θα αναγνώσει ο κάθε mapper από το επόμενο κομμάτι. Το τελευταίο χαρακτηριστικό είναι πολύ σημαντικό για να διασφαλίσουμε ότι η καταμέτρηση των προθεμάτων ενός συγκεκριμένου μήκους δεν θα επηρεαστεί από το χωρισμό της ακολουθίας. Όπως προκύπτει από το σχήμα 4.3, πρέπει σε κάθε κομμάτι να προσθέσουμε τους length-1 χαρακτήρες του επόμενου, με length το μήκος του prefix που ελέγχει η συγκεκριμένη εργασία.

Ο κώδικας της συνάρτησης map είναι ιδιαίτερα απλός: το κομμάτι του κειμένου σαρώνεται με ένα παράθυρο μήκους length και σε κάθε βήμα αναζητείται η συμβολοσειρά που σχηματίζεται από το παράθυρο στην ειδική λίστα που κρατάει τα προθέματα. Αν βρεθεί, τότε δίνεται έξοδος (πρόθεμα, 1), που σημαίνει ότι το συγκεκριμένο πρόθεμα βρέθηκε 1 φορά. Στη συνέχεια η reduce αθροίζει τις τιμές για να βρει τη συνολική συχνότητα του προθέματος, παρόμοια με το παράδειγμα καταμέτρησης λέξεων που δόθηκε στο κεφάλαιο του MapReduce. Η συνάρτηση Reduce χρησιμοποιείται και ως Combine για να γίνουν πρώτα τοπικά αθροίσματα των συχνοτήτων στους Mappers για το κάθε πρόθεμα και το κάθε κομμάτι εισόδου. Έστω ένα πρόθεμα p με συχνότητα f σε μια ακολουθία με n κομμάτια. Χωρίς την χρήση της combine, θα πρέπει να μεταφερθούν μέσω δικτύου f ζευγάρια (p, 1) και να γίνουν f προσθέσεις. Με την χρήση της combine θα χρειαστεί να μεταφερθούν και να αθροιστούν μόνο n ζευγάρια.

Ειδικά για προθέματα μικρού μήκους σε μεγάλα κομμάτια εισόδου (πχ μήκος 4 σε κομμάτι 1M ακολουθίας 100M) είναι σίγουρο ότι f»n. Άρα βλέπουμε ότι η χρήση της combine μειώνει ιδιαίτερα τα ενδιάμεσα δεδομένα, δίνοντάς μας καλύτερους χρόνους στην ενδιάμεση φάση του Shuffle και συνολικά στην εργασία.

```
map:
 pos=0
 όσο το pos είναι μικρότερο του μήκους του κομματιού
   διάβασε length χαρακτήρες από τη θέση pos
   αν το πρόθεμα υπάρχει στη λίστα προθεμάτων
     δώσε έξοδο (πρόθεμα, 1)
   pos++

reduce:
  sum = 0
  για κάθε τιμή
    sum += τιμή
  δώσε έξοδο (κλειδί, sum)
```

Στην υλοποίηση υπάρχουν διαφορές σε σχέση με τη διαδικασία του Trellis οι οποίες οφείλονται σε περιορισμούς της παρούσας έκδοσης του MapReduce. Συγκεκριμένα, ο τύπος MapFile δεν επιτρέπει την παρουσία κλειδιών διαφορετικού μήκους. Παράλληλα, επειδή χρειαζόμαστε μοναδικό αρχείο εξόδου, περιοριζόμαστε αυτή τη στιγμή σε έναν και μοναδικό reducer. Επίσης δεν είναι δυνατόν να υλοποιηθεί μια έκδοση η οποία να διαχωρίζει τα τελικά προθέματα στον reducer, γιατί θα απαιτούσε ο reducer να μπορεί να γράψει σε 2 εξόδους, μία για τα προθέματα με συχνότητα λιγότερη από το κατώφλι και μία για τα προθέματα με συχνότητα μεγαλύτερη. Οι περιορισμοί αυτοί δεν θα υπάρχουν στην επόμενη έκδοση του Hadoop, οπότε και μπορεί να υλοποιηθεί μια καλύτερη έκδοση αυτού του κομματιού. Στην πράξη η υλοποίηση αυτή έχει το πλεονέκτημα ότι μειώνει στο ελάχιστο τα ενδιάμεσα δεδομένα. Στο πλαίσιο της εργασίας αυτής υλοποιήθηκε και μια δοκιμαστική έκδοση που καταμετρά τις συχνότητες προθεμάτων μήκους 4 με 8 και συγκρίθηκε με την απλή έκδοση. Τα αποτελέσματα παρατίθενται στο κεφάλαιο των μετρήσεων. Σε κάθε περίπτωση, μετά το πέρας αυτής της φάσης έχουμε όλα τα τα προθέματα μιας ακολουθίας εισόδου που έχουν συχνότητα λιγότερη από το κατώφλι.

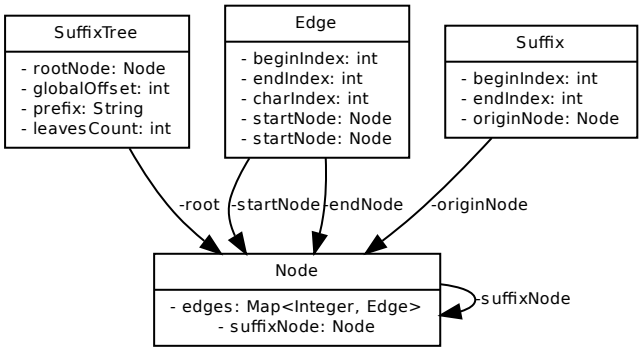
4.3 Κατασκευή Υποδέντρων Επιθεμάτων

Αφού έχει οριστικοποιηθεί το σύνολο των προθεμάτων, περνάμε στην επόμενη εργασία του προγράμματος που είναι η κατασκευή των προθεματικών υποδέντρων επιθεμάτων στη φάση map και η συγχώνευση τους σε δέντρα στη φάση reduce. Για να εκτελεστούν τα βήματα αυτά πρέπει να ακολουθήσουμε λίγο διαφορετική πρακτική από πριν. Ο λόγος σχετίζεται με τη φάση της συγχώνευσης των υποδένδρων: όπως έχει αναφερθεί, η φάση εμπλέκει τυχαία

προσπέλαση στο σύνολο της ακολουθίας εισόδου. Αυτή η τυχαία προσπέλαση φαίνεται να μην μπορεί να αποφευχθεί με προφανή τρόπο. Για το λόγο αυτό η ακολουθία εισόδου πρέπει να είναι διαθέσιμη από όλους τους reducers. Κάτι τέτοιο μπορεί να επιτευχθεί με τη χρήση της κατανεμημένης κρυφής μνήμης (distributed cache) που προσφέρει το Hadoop: Η ακολουθία εισόδου μεταφέρεται στην κατανεμημένη κρυφή μνήμη. Αυτό δημιουργεί ένα τοπικό αντίγραφο σε κάθε κόμβο και έτσι όλοι οι κόμβοι έχουν πρόσβαση σε αυτή. Για να μειώσουμε το κόστος μεταφοράς και το χώρο που καταλαμβάνει η ακολουθία εισόδου, χρησιμοποιούμε συμπίεση: Όπως έχει αναφερθεί, το αλφάβητο του αρχείου εισόδου είναι 4 γράμματα που αντιστοιχούν σε 4 καταστάσεις και επομένως μπορούν να χρησιμοποιηθούν μόλις 2bits για την αναπαράσταση ενός χαρακτήρα του αλφαβήτου. Με αυτόν τον τρόπο μπορούμε να υποτετραπλασιάσουμε τις απαιτήσεις χώρου και να φορτώσουμε ένα αρχείο μεγέθους 1GB χρησιμοποιώντας μόλις 256MB. Αφού έχουμε φορτώσει την ακολουθία εισόδου στην κατανεμημένη κρυφή μνήμη για να τη χρησιμοποιήσουν οι Reducers, αυτή είναι διαθέσιμη επίσης κατά τη φάση Map. Θα ήταν λοιπόν σπατάλη να φορτώνεται από αρχεία στο HDFS το κομμάτι της ακολουθίας που χρειάζεται ο κάθε Mapper, γιατί αυτό το κομμάτι το έχουμε ήδη στην κατανεμημένη κρυφή μνήμη. Είναι φανερό ότι θέλουμε να αναθέσουμε όλες τις μεταφορές δεδομένων στην κατανεμημένη κρυφή μνήμη, αλλά και να δημιουργήσουμε ένα συγκεκριμένο αριθμό από Map Jobs. Αυτό είναι δυνατόν αν ορίσουμε ειδικό τύπο InputFormat στον οποίο θέτουμε ως ιδιότητα τον αριθμό των Map Jobs και αυτό αναλαμβάνει να δημιουργήσει ανάλογο αριθμό, στέλνοντάς τους splits μηδενικού μεγέθους. Ο αριθμός των Map Jobs είναι ιδιότητα η οποία προσδιορίζεται στο αντικείμενο Configuration της εργασίας. Αρχιζοντας από τη φάση Map θα πρέπει να προσδιορίσουμε το κομμάτι της ακολουθίας εισόδου για το οποίο η συνάρτηση map θα κατασκευάσει τα προθεματικά υποδέντρα. Για αυτό το σκοπό αρκεί ο κάθε mapper να λάβει την τιμη του μήκους του split, $splitLength = fileLength/numberofmappers$, μέσω ιδιότητας του Configuration και τότε μπορεί να υπολογίσει τον πρώτο και τελευταίο χαρακτήρα του κομματιού ως συνάρτηση του TaskID, του μοναδικού αριθμού που χαρακτηρίζει το Task. Τα TaskID αριθμούνται από το 0. Έτσι, όπως φαίνεται και στον παρακάτω ψευδοκώδικα, ο πρώτος προς επεξεργασία χαρακτήρας είναι πάντα ο $ID \cdot splitLength$, ενώ ο τελευταίος είναι είτε ο τελευταίος χαρακτήρας με θέση $fileLength$ για το τελευταίο κομμάτι, είτε ο $begin + splitLength$ για τα υπόλοιπα κομμάτια.

```
int fileLength = context.getConfiguration().getInt(SuffixTreeMapper.FILE_LENGTH, 0);
int splitLength = context.getConfiguration().getInt(SuffixTreeMapper.SPLIT_LENGTH, 0);
int ID = Integer.parseInt(context.getTaskAttemptID().toString().split(''_'')[4]);
int begin = ID * splitLength;
int end;
if (begin + splitLength >= fileLength) {
  end = fileLength;
} else {
  end = begin + splitLength;
}
```

Πριν προχωρήσουμε είναι σκόπιμο να περιγραψουμε τις κλάσεις που υλοποιούν το δέντρο επιθεμάτων. Ένα σχηματικό διάγραμμα φαίνεται στο σχήμα 4.4.



Σχήμα 4.4: Οι κλάσεις που απαρτίζουν τα δέντρα επιθεμάτων

Η κλάση Edge έχει ως πεδία references στον κόμβο που εισέρχεται και στον κόμβο από τον οποίο εξέρχεται. Ακόμα, η κλάση της ακμής κρατάει τη θέση του πρώτου και του τελευταίου χαρακτήρα της ετικέτας της, όπως και την τιμή του πρώτου χαρακτήρα της ετικέτας. Η κλάση Node έχει ως πεδία το σύνδεσμο επιθέματος ως reference σε άλλο Node, καθώς και ένα HashMap που αντιστοιχεί τους πρωτους χαρακτήρες των εξερχόμενων από αυτήν ακμών με της ακμές αυτές. Η κλάση SuffixTree χρειάζεται να έχει ένα reference για το Node που είναι ρίζα του δέντρου. Ακόμα αποθηκεύουμε τον αριθμό των φύλλων του δέντρου, καθώς και το πρόθεμα και τη θέση του κομματιού που περιγράφει. Τέλος η κλάση Suffix είναι μια βοηθητική κλάση για την πρόσθεση επιθεμάτων στο δέντρο στην υλοποίηση του αλγορίθμου του Ukkonen και δεν αποτελεί στοιχείο που θα χρειαστούμε μετά την κατασκευή του δέντρου.

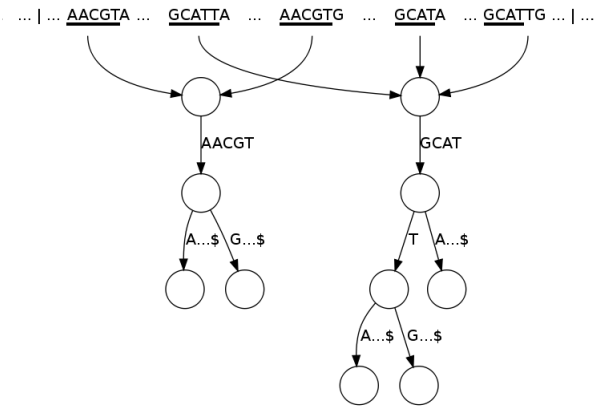
Στη συνάρτηση map συγκεκριμενοποιούμε τον αλγόριθμο που θα παράξει τα ενδιάμεσα υποδέντρα. Σε αυτό το σημείο χρησιμοποιήθηκαν τρεις διαφορετικοί αλγόριθμοι.

4.3.1 Απλοϊκή Προσέγγιση

Η πρώτη απλοϊκή προσέγγιση είναι να σαρώσουμε το κομμάτι της ακολουθίας εισόδου μία φορά για κάθε επίθεμα p. Κάθε φορά που εντοπίζουμε το p, προσθέτουμε στο προθεματικό δέντρο ένα νέο φύλλο ως εξής: ξεκινώντας από τη ρίζα, κάνουμε όσες συγκρίσεις χρειάζονται μέχρι να βρεθεί διαφορά με κάποια υπάρχουσα ακμή. Τότε η ακμή αυτή χωρίζεται και δημιουργείται ένας νέος εσωτερικός κόμβος.

Ο αλγόριθμος που εκτελείται σε έναν κόμβο μόνο έχει ως εξής: Έστω P το σύνολο των προθεμάτων p_i που το καθένα έχει συχνότητα f_i μικρότερη του κατωφλιού σε μια ακολουθία εισόδου n χαρακτήρων. Τότε στην προσέγγιση αυτή οι n χαρακτήρες θα διαβαστούν |P| φορές και η συνάρτηση προσθήκης νέου φύλλου θα κληθεί για κάθε πρόθεμα p_i, f_i φορές. Αφού κατασκευαστεί το προθεματικό υποδέντρο του προθέματος p_i συνεχίζουμε για το p_{i+1}

ξεκινώντας από κενό δέντρο. Συνολικά η συνάρτηση προσθήκης φύλλου θα κληθεί n φορές. Σε κάθε κλήση της συνάρτησης χρειάζονται να γίνουν τόσες συγκρίσεις όσες και το μέγιστο κοινό πρόθεμα (LCP) της υπάρχουσας ακμής και του επιθέματος που προστίθεται. Όπως είναι φανερό από την παραπάνω περιγραφή, ο αλγόριθμος απέχει πολύ από το να είναι γραμμικός σε χρόνο, αλλά παρουσιάζει καλά χαρακτηριστικά σε απαιτήσεις χώρου. Στην πράξη παρουσιάζει και καλά χρονικά χαρακτηριστικά, καθώς η είσοδος χωρίζεται σε κομμάτια και η επεξεργασία εκτελείται παράλληλα από πολλούς υπολογιστές. Έστω τότε $p_{i,j}$ το πρόθεμα p_i στο j-οστό κομμάτι της εισόδου. Η συνάρτηση προσθήκης επιθέματος καλείται $f_{i,j}$ φορές, όπου $f_{i,j}$ η συχνότητα του i-οστού προθέματος στο j-οστό κομμάτι εισόδου. Επειδή το $f_{i,j}$ είναι αρκετά μικρό, το μέγιστο κοινό πρόθεμα είναι αρκετά σύντομο και οι απαιτούμενες συγκρίσεις λίγες. Η μέθοδος φαίνεται στο σχήμα 4.5.



Σχήμα 4.5: Ο απλοϊκός αλγόριθμος για την κατασκευή προθεματικών υποδέντρων κατασκευάζει απευθείας το υποδέντρο σκανάροντας όλο το κομμάτι για κάθε πρόθεμα

Για κάθε κομμάτι της εισόδου ο αντίστοιχος κόμβος εκτελεί:

```
για κάθε πρόθεμα p:
  εντόπισε την επόμενη θέση του στο κομμάτι;
  πρόσθεσε το επίθεμα στο προθεματικό υποδέντρο
  κάνοντας συγκρίσεις μέχρι να βρεθεί το LCP;
  πρόσθεσε νέο εσωτερικό κόμβο και
  νέο φύλλο;
```

4.3.2 Ukkonen

Ο δεύτερος αλγόριθμος είναι μια υλοποίηση του αλγορίθμου του Ukkonen σε Java βασισμένος στην υλοποίηση [13] σε C++. Στη φάση map για να κατασκευαστούν τα προθεματικά υποδέντρα κατά Ukkonen έχουμε τρία στάδια:

1. Κατασκευή του υποδέντρου επιθεμάτων κατά Ukkonen για το κομμάτι της ακολουθίας εισόδου που αντιστοιχεί στον συγκεκριμένο Mapper
2. Κανονικοποίηση του δέντρου

3. Εξαγωγή των προθεματικών υποδέντρων επιθεμάτων από το υποδέντρο επιθεμάτων

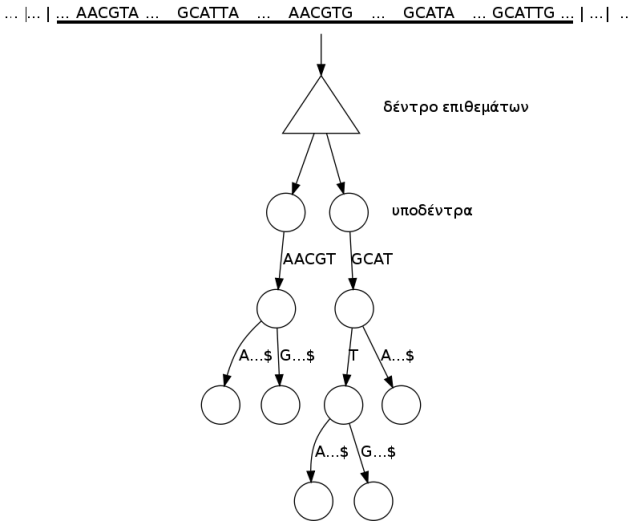
Κατασκευή υποδέντρου επιθεμάτων: Εδώ πρέπει να κατασκευάσουμε το δέντρο επιθεμάτων για το κομμάτι εισόδου που αντιστοιχεί στον συγκεκριμένο mapper. Ο αλγόριθμος ξεκινάει διαβάζοντας έναν - έναν τους χαρακτήρες του κομματιού. Κάθε χαρακτήρας που διαβάζεται αντιστοιχεί σε μια φάση του αλγορίθμου του Ukkonen και καλείται για αυτόν μια συνάρτηση addPrefix η οποία αναλαμβάνει να κάνει όλες τις επεκτάσεις. Η addPrefix χρειάζεται ως είσοδο τη θέση του τελευταίου χαρακτήρα και ένα αντικείμενο Suffix που δείχνει το ενεργό σημείο του δέντρου ως συνάρτηση ενός κόμβου-πατέρα originNode και μιας υποσυμβολοσειράς της ακολουθίας εισόδου με αρχή το beginIndex και τέλος το endIndex. Η δομή αυτή είναι global στο πρόγραμμα και όποιες αλλαγές γίνουν σε αυτήν μέσα στην addPrefix θα επηρεάσουν τις επόμενες φάσεις πρόσθεσης χαρακτήρων. Ο κώδικας της μεθόδου addPrefix εκτελείται μέχρι να μην χρειάζονται άλλες επεκτάσεις. Κάθε φορά ελέγχεται η δομή Suffix και συγκεκριμένα το μήκος της υποσυμβολοσειράς που αυτή περιγράφει, beginIndex - endIndex. Αν το beginIndex είναι μεγαλύτερο, τότε βρισκόμαστε σε explicit ακμή που σημαίνει πως το επίθεμα δεν τερματίζει σε κάποια υπάρχουσα ακμή. Σε αυτή την περίπτωση ελέγχουμε αν υπάρχει ο πρώτος χαρακτήρας στο originNode. Σε περίπτωση θετικής απάντησης διακόπτουμε την πρόσθεση επεκτάσεων για αυτό τον χαρακτήρα, μιας και σύμφωνα με τους κανόνες του Ukkonen, αν ο χαρακτήρας βρεθεί να είναι ήδη τοποθετημένος διακόπτουμε τις επεκτάσεις. Αλλιώς προσθέτουμε την καινούργια ακμή και δημιουργούμε το σύνδεσμο επιθέματος. Αν το endIndex είναι μεγαλύτερο του beginIndex, τότε η ακμή που επιχειρούμε να προσθέσουμε τελειώνει στη μέση κάποιας υπάρχουσας ακμής. Ελέγχουμε αν ο καινούργιος χαρακτήρας υπάρχει στην ακμή που εξετάζουμε. Αν οι χαρακτήρες είναι ίδιοι τότε εχουμε πάλι εφαρμογή του κανόνα του ukkonen και σταματάμε τις επεκτάσεις για αυτή τη φάση. Αν οι χαρακτήρες διαφέρουν τότε σπάμε σε αυτό το σημείο την ακμή του δέντρου, δημιουργούμε νέο εσωτερικό κόμβο και εισάγουμε σε αυτόν την υπόλοιπη παλαία ακμή και την καινούργια, που περιγράφει το επίθεμα του νέου χαρακτήρα. Σε περίπτωση που δεν έχουμε διακοπή των επεκτάσεων με βάση τα κριτήρια που ειπώθηκαν παραπάνω, προχωρούμε στην επόμενη επέκταση, ακολουθώντας το σύνδεσμο επιθέματος του originNode του Suffix. Αν οι επεκτάσεις σταματήσουν, προσαρμόζουμε κατάλληλα τη δομή του Suffix για την επόμενη φάση-χαρακτήρα, αυξάνοντας το beginIndex κατά ένα.

Μια λεπτομέρεια που παραλήφθηκε είναι πότε θα σταματήσουμε να προσθέτουμε καινούργιους χαρακτήρες στο δέντρο επιθεμάτων. Όπως αναφέρθηκε και στο κεφάλαιο του Trellis, όλα τα κομμάτια του κειμένου εκτός από το τελευταίο, δεν περιλαμβάνουν τον τερματικό χαρακτήρα $, και έτσι ο αλγόριθμος του Ukkonen θα δώσει πεπλεγμένο δέντρο επιθεμάτων. Για να λάβουμε ένα πραγματικό δέντρο επιθεμάτων το οποίο να περιλαμβάνει όλα τα επιθέματα του κομματιού θα χρειαστεί να ελέγξουμε χαρακτήρες και μετά το πέρας του κομματιού. Αυτό δεν αποτελεί πρόβλημα, καθώς όλη η ακολουθία εισόδου βρίσκεται φορτωμένη στη μνήμη, και οι επιπλέον χαρακτήρες τόσο λίγοι που δεν επηρεάζουν την πολυπλοκότητα. Η μεταβλητή που αλλάζει με το τέλος των επεκτάσεων μίας φάσης είναι το beginIndex του Suffix. οπότε συνεχίζουμε να τρέχουμε νέες φάσεις μέχρι το beginIndex να φτάσει στον τελευταίο χαρακτήρα του κομματιού που θα είναι ο πρώτος χαρακτήρας του τελευταίου επιθέματος που θα

προστεθεί στο δέντρο. Διασφαλίζουμε έτσι ότι ο αριθμός των φύλλων στο δέντρο θα είναι τόσος όσοι και οι χαρακτήρες του κομματιού. Στη συνέχεια και σύμφωνα με τον αλγόριθμο του Ukkonen, προσθέτουμε στο δέντρο τον τερματικό χαρακτήρα για να λάβουμε ένα πραγματικό δέντρο επιθεμάτων.

Κανονικοποίηση του δέντρου: Στη συνέχεια θα πρέπει να αντικαταστήσουμε σε κάθε φύλλο τη θέση του τέλους της ακολουθίας με την πραγματική τιμή της. Στην πραγματικότητα αυτό το βήμα στην περίπτωσή μας έχει ήδη εκτελεστεί εσωτερικά στον αλγόριθμο εισαγωγής νέων ακμών. Το βήμα αυτό είναι απαραίτητο μόνο όταν ο αλγόριθμος του Ukkonen λειτουργεί on-line, δηλαδή λαμβάνει stream δεδομένων χωρίς να ξέρει το σημείο που τελειώνει η ακολουθία. Για την εφαρμογή μας κάτι τέτοιο δεν ισχύει, αφού ξέρουμε το μέγεθος του αρχείου και χρησιμοποιούμε τον πραγματικό τελευταίο χαρακτήρα. **Εξαγωγή προθεματικών υποδέντρων:** Το τελευταίο στάδιο είναι η εξαγωγή των προθεματικών υποδέντρων από το δέντρο επιθεμάτων. Για να γίνει η εξαγωγή του υποδέντρου για ένα πρόθεμα, πρέπει να αναζητήσουμε το πρόθεμα στο δέντρο με τη διαδικασία του ακριβούς ταιριάσματος. Αν καταφέρουμε να κάνουμε πλήρες ταίριασμα του προθέματος, τότε ο τελευταίος χαρακτήρας ορίζει το προθεματικό υποδέντρο για το πρόθεμα αυτό, ενώ αν δεν μπορεί να γίνει πλήρες ταίριασμα, τότε το συγκεκριμένο πρόθεμα δεν θα έχει υποδέντρο για αυτό το κομμάτι.

Όλη η διαδικασία φαίνεται στο σχήμα 4.6.



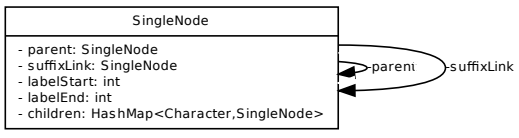
Σχήμα 4.6: Ο αλγόριθμος του Ukkonen απαιτεί την κατασκευή ολόκληρου του δέντρου επιθεμάτων του κομματιού και την εξαγωγή των προθεματικών υποδέντρων από αυτό.

```
για κάθε χαρακτήρα x του κομματιού της εισόδου:
  Node lastParentNode = null;
  Node parentNode;

  όσο χρειάζονται περισσότερες επεκτάσεις:
    Edge edge;
```

```
    parentNode = active.getOriginNode();

    αν η εξεταζόμενη ακμή είναι Explicit
    αν υπάρχει ακμή που να εξέρχεται από το active.originNode με πρώτο γράμμα το x:
        σταμάτα τις επεκτάσεις

    αλλιώς η εξεταζόμενη ακμή είναι Implicit
     βρες την ακμή που εξέρχεται από το πρώτο γράμμα της εξεταζόμενης ακμής
    αν ο χαρακτήρας χ είναι ίδιος με τον επόμενο χαρακτήρα της ακμής του δέντρου
       σταμάτα τις επεκτάσεις
     αλλιώς
     χώρισε την ακμή του δέντρου και δημιούργησε κανούργιο εσωτερικό κόμβο, τον parentNode

  εισαγωγή νέας ακμής που αναπαριστά το νέο επίθεμα
  προσθήκη του σύνδεσμου επιθέματος

  αν το originNode είναι η ρίζα
    αύξηση του active.beginIndex
    active.incBeginIndex();
  αλλιώς
    το active.originNode αλλάζει ακολουθώντας το σύνδεσμο επιθέματος.

αύξηση του active.endIndex
```

4.3.3 Ukkonen από βιβλιοθήκη προγραμμάτων

Η τρίτη διαφορετική υλοποίηση στο κομμάτι του map ήταν να χρησιμοποιηθεί έτοιμη υλοποίηση του αλγορίθμου του Ukkonen από ειδική βιβλιοθήκη αλγορίθμων για βιολογικά δεδομένα της Java [16]. Η υλοποίηση αυτή κωδικοποιεί κόμβους και ακμές σε μία κοινή δομή δεδομένων. Η δομή δεδομένων φαίνεται στο σχήμα 4.7. Παρατηρούμε ότι περιέχει την ίδια ουσιαστικά πληροφορία με την προηγούμενη υλοποίηση. Η λογική της κατασκευής του δέντρου είναι ίδια με πριν. Η υλοποίηση όμως χρησιμοποιεί 5 ενιαίους κανόνες για την κατασκευή και διάσχιση του δέντρου:



Σχήμα 4.7: Η κεντρική κλάση της υλοποίησης της βιβλιοθήκης biojava

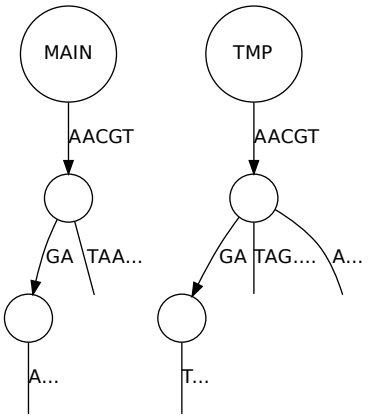
1. Η διαδικασία τελειώνει σε φύλλο.
2. Η διαδικασία πρέπει να προσθέσει ακμή σε εσωτερικό κόμβο.
3. Η διαδικασία πρέπει να χωρίσει μια ακμή.
4. Η διαδικασία τελειώνει εσωτερικά σε ακμή.
5. Η διαδικασία τελειώνει σε εσωτερικό κόμβο.

Με αυτό τον τρόπο αναλύει τις περιπτώσεις που αναφέραμε στην προηγούμενη ενότητα και επιλέγει κατευθείαν τον σωστό κανόνα έτσι ώστε να κάνει συνολικά λιγότερα βήματα. Η κατασκευή και η διάσχιση του δέντρου γίνεται εσωτερικά, με τον χρήστη να μπορεί να επέμβει μέσω προκαθορισμένων μεθόδων. Όπως θα δούμε στην πράξη, αυτή η υλοποίηση αποδείχθηκε και η πιο αποτελεσματική. Σε κάθε περίπτωση η έξοδος της φάσης map είναι ίδια και για τους δύο αλγορίθμους. Το υποδέντρο ελέγχεται ώστε να έχει τόσα φύλλα όσο και η συχνότητα εμφάνισης του προθέματος στο κομμάτι. Αν ο αριθμός των φύλλων είναι μηδέν, τότε το υποδέντρο δεν δίνεται ως έξοδος. Η έξοδος έχει τύπο (πρόθεμα, προθεματικό υποδέντρο επιθεμάτων).

4.4 Συγχώνευση των υποδέντρων σε δέντρα

Σε αυτή τη φάση συγχωνεύουμε τα προθεματικά υποδέντρα σε προθεματικά δέντρα. Η συνάρτηση reduce διαβάζει τη λίστα από τα υποδέντρα με το κοινό πρόθεμα. Συγχωνεύει κάθε φορά ένα από αυτά, που θεωρείται προσωρινό και ονομάζεται tmp, με ένα κύριο δέντρο που ονομάζεται main. Το κύριο δέντρο δέχεται κάθε φορά προσθήκες από το προσωρινό, έτσι ώστε στο τέλος να έχουν προστεθεί στο main όλα τα φύλλα όλων των υποδέντρων με το κοινό πρόθεμα και θα αποτελεί έτσι το προθεματικό δέντρο επιθεμάτων όλης της ακολουθίας. Η συνάρτηση συγχώνευσης είναι μια αναδρομική συνάρτηση που δουλεύει σε κάθε κλήση της πάνω σε δύο κόμβους, τον κόμβο mainNode του κύριου δέντρου main και τον κόμβο tmpNode του προσωρινού δέντρου tmp. Το σώμα της συνάρτησης επαναλαμβάνεται για κάθε εξερχόμενη ακμή του tmpNode. Αν δεν υπάρχει ακμή με τον ίδιο πρώτο χαρακτήρα στο mainNode, τότε απλά προσθέτουμε την ακμή στο mainNode μαζί με ολόκληρο το υποδέντρο που ορίζει. Αν υπάρχει ο χαρακτήρας, τότε μπαίνουμε σε μια διαδικασία σύγκρισης μέχρι να βρούμε το μέγιστο κοινό πρόθεμα. Συγκεκριμένα, συγκρίνουμε έναν - έναν τους χαρακτήρες που ορίζουν οι δύο ακμές. Αν βρεθεί διαφορά, τότε σε εκείνο το σημείο χωρίζουμε την ακμή του mainNode, εισάγουμε νέο εσωτερικό κόμβο και εισάγουμε νέα ακμή από το tmpNode, από το σημείο της διαφοράς και μετά. Αν οι συγκρίσεις ολοκληρωθούν και δεν βρεθεί διαφορά τότε υπάρχουν διάφορες περιπτώσεις. Μπορεί να εξαντλήθηκαν οι χαρακτήρες της ακμής του mainNode, οπότε πρέπει να επανακαλέσουμε την συνάρτηση συγχώνευσης έτσι ώστε να συγκρίνει τον κόμβο παιδί της mainNode με τον κόμβο της tmpNode από το τέλος της προηγούμενης σύγκρισης και μετά. Αν εξαντλήθηκαν οι χαρακτήρες του tmpNode, απλά καλούμε τη συνάρτηση συγχώνευσης για το mainNode και το endNode του tmpNode, ελέγχοντας το mainNode από το τέλος της προηγούμενης σύγκρισης και μετά. Υπάρχει τέλος και η περίπτωση οι ακμές να έχουν ίδιο μήκος και ίδιους χαρακτήρες οπότε καλούμε

αναδρομικά τη συνάρτηση συγχώνευσης για τα endNodes. Ένα απλό παράδειγμα των παραπάνω δείχνεται στο σχήμα 4.8. Όπως βλέπουμε, η ακμή του tmp δέντρου που αρχίζει από "A" και όλο το υποδέντρο που ορίζει μπορούν να προστεθούν άμεσα στον κόμβο του δέντρου main, γιατί ο τελευταίος δεν έχει ακμή που να αρχίζει από αυτό το γράμμα. Για την ακμή του tmp που αρχίζει από το "TAG", θα πρέπει να βρεθεί το μέγιστο κοινό πρόθεμα με την ακμή του δέντρου main που αρχίζει από "TAA". Αυτό είναι το "TA", η ακμή του main χωρίζεται στο σημείο εκείνο και δημιουργείται καινούργιος εσωτερικός κόμβος. Τέλος για τις ακμές που έχουν ίδια ετικέτα "GA", ο αλγόριθμος θα κάνει κλήση της συνάρτησης συγχώνευσης δίνοντας ως είσοδο τους κόμβους στους οποίους εισέρχονται οι ακμές αυτές. Ακολουθεί ο ψευδοκώδικας για το κομμάτι αυτό.



Σχήμα 4.8: Παράδειγμα για τις δύο περιπτώσεις της συγχώνευσης

```
mainNode = mainSuffixTree.getRootNode();
tmpNode = tmpSuffixTree.getRootNode();

merge:
για κάθε εξερχόμενη ακμή του tmpNode
  αν δεν υπάρχει εξερχόμενη ακμή με το ίδιο πρώτο γράμμα στο mainNode
    πρόσθεσε την ακμή στο mainNode
  αλλιώς
   σύγκρινε έναν - έναν τους χαρακτήρες των δύο ακμών που έχουν κοινό πρώτο γράμμα
      αν βρεθεί διαφορά
        χώρισε την ακμή του main και πρόσθεσε κατάλληλη ακμή από το tmp
      αλλιώς
        επανάλαβε τη σύγκριση κατεβαίνοντας κατά ένα κόμβο
        στην ακμή που εξαντλήθηκαν οι χαρακτήρες
```

Αξίζει να ασχοληθούμε με την ενδιάμεση φάση Shuffle. Στη φάση shuffle τα ζευγάρια (πρόθεμα, προθεματικό υποδέντρο) ομαδοποιούνται κατά πρόθεμα και μεταφέρονται σε ένα κόμβο ο οποίος θα αναλάβει τη συγχώνευσή τους. Σε αρχικές υλοποιήσεις έγινε η υπόθεση ότι τα προθεματικά υποδέντρα πρέπει να συγχωνευτούν με τη σειρά, ώστε το υποδέντρο του 1ου κομματιού να συγχωνευτεί με το υποδέντρο του 2ου κομματιού και μετά του 3ου μέχρι να ολοκληρωθούν όλες οι συγχωνεύσεις. Για να καταστεί αυτό δυνατόν θέλουμε οι τιμές που λαμβάνει η συνάρτηση reduce να είναι σε σειρά ανάλογα με το κομμάτι από το οποίο έγινε η κατασκευή τους. Η αναγκαιότητα αυτή προέκυπτε από το γεγονός ότι στα ενδιάμεσα υποδέντρα δεν κρατιόταν ο τελικός χαρακτήρας, με αποτέλεσμα ένα επίθεμα να μπορεί να είναι πρόθεμα άλλου. Έτσι υπήρχε η περίπτωση εξάντλησης των χαρακτήρων κατά τις συγκρίσεις για την συγχώνευση. Παρότι αυτό δεν είναι τελικά αναγκαίο στην υλοποίηση, αξίζει να δούμε πώς μπορεί να λυθεί. Το πρόβλημα δεν έχει απλή λύση, καθώς η λίστα τιμών για ένα συγκεκριμένο κλειδί είναι κλάσης Iterable, που πρακτικά σημαίνει ότι επιτρέπεται μόνο μία προσπέλαση της κάθε τιμής. Αρχικά δοκιμάστηκαν διάφορες λύσεις, όπως να γίνεται σύγκριση των δύο δέντρων εισόδου και να επιλέγεται ως κύριο δέντρο αυτό που έχει αρχική θέση πιο αριστερά στην είσοδο στη συνάρτηση συγχώνευσης. Αυτή η μέθοδος απαιτούσε πολλές κλήσεις της συνάρτησης new και καθυστερούσε υπερβολικά την εκτέλεση. Άλλη λύση ήταν η αντιγραφή των τιμών σε μια λίστα και το sorting της λίστας με κριτήριο τη θέση μέσα στην ακολουθία. Και αυτή η λύση απαιτούσε πολλές αντιγραφές δομών και έδωσε απογοητευτικούς χρόνους. Η τελευταία λύση που προτάθηκε ήταν στις θέσεις που δεν μπορεί να προχωρήσει η σύγκριση να αφήνεται ένα placeholder το οποίο να λύνεται εκτός εκτέλεσης map-reduce.

Όλες οι παραπάνω λύσεις έχουν φοβερά μειονεκτήματα και δεν λαμβάνουν υπόψη τους την καρδιά της διαδικασίας MapReduce που είναι η ενδιάμεση φάση Shuffle. Αυτή η φάση έχει τρία στάδια, ένα στη φάση Map και 2 στη φάση reduce. Το πρώτο στάδιο είναι το Partition, στο οποίο τα κλειδιά χωρίζονται σε ομάδες με την κάθε ομάδα να αποτελεί input για μία κλήση της reduce. Αφού τερματίσουν όλες οι συναρτήσεις map, τότε τα ενδιάμεσα δεδομένα αντιγράφονται τοπικά στον Reducer που αντιστοιχούν. Κατόπιν στα δεδομένα γίνεται sorting στο key για να μπουν σε σωστή σειρά και μετά grouping που καθορίζει ποιων κλειδιών οι τιμές θα δωθούν ως Iterable σε μια κλήση της reduce. Έτσι, δίνοντας ως κλειδί το πρόθεμα, το mapreduce θα στείλει τα ζευγάρια με το ίδιο πρόθεμα στον ίδιο reducer, ο reducer θα συγκρίνει τα προθεματα για να βάλει τα ζευγάρια σε σωστή σειρά και θα καλεσει τη reduce δίνοντάς της ως είσοδο όλες τις τιμές που έχουν ίδιο πρόθεμα. Όπως είναι φυσικό μπορεί ένας reducer να έχει λάβει τα ζευγάρια παραπάνω του ενός κλειδιού, αλλά απαραίτητα καλεί την reduce μία φορά για κάθε διαφορετικό κλειδί. Αν συνδυάσουμε τα παραπάνω θα δούμε ότι και στα τρία στάδια γίνεται επεξεργασία του κλειδιού. Μπορούμε να εκμεταλλευτούμε τα τρία αυτά στάδια για να πράξουν το sorting που χρειαζόμαστε για εμάς. Συγκεκριμένα, αρκεί να γίνει partition με βάση το πρόθεμα, sorting με πρώτο κριτήριο το πρόθεμα και δεύτερο τη θέση μέσα στην ακολουθία εισόδου, και grouping με βάση πάλι το πρόθεμα. Για να υλοποιηθούν τα παραπάνω αρκεί να φτιάξουμε μια νέα κλάση ειδικού κλειδιού με δύο πεδία, πρόθεμα και θέση στην ακολουθία εισόδου. Ο προεπιλεγμένος partitioner είναι ο HashPartitioner, που ομαδοποιεί τα ζευγάρια εξοδου της map με βάση το hashCode του κλειδιού. Δεν χρειάζεται

αλλαγή, απλά η νέα μας κλάση θα επιστρέφει ως hashCode το hashCode του προθέματος. Τα στάδια sorting και grouping λειτουργούν με ειδικές κλάσεις Comparator που δρουν σε επίπεδο bytes συγκρίνοντας δύο αντικείμενα της ίδιας κλάσης. Ο Sorting Comparator που ορίζουμε συγκρίνει πρώτα ως προς το πρόθεματα και έπειτα ως προς τη θέση. Ο Grouping Comparator συγκρίνει απλά ως προς το πρόθεμα. Το αποτέλεσμα είναι το Iterable των τιμών σε μια κλήση της reduce να είναι σε σειρά ως προς τη θέση του στην ακολουθία εισόδου.

Το τελικό δέντρο ελέγχεται ώστε να είναι βέβαιο ότι περιέχει τόσα φύλλα όσο και το άθροισμα των φύλλων των υποδέντρων που συγχωνεύτηκαν. Στο τέλος τα προθεματικά δέντρα επιθεμάτων αποθηκεύονται στο HDFS και το πρόγραμμά μας τερματίζει.

Κεφάλαιο 5

Πειραματική Αξιολόγηση

Σε αυτό το κεφάλαιο θα αξιολογηθούν πειραματικά οι διάφορες υλοποιήσεις που έγιναν για την προσαρμογή της ιδέας του Trellis στο MapReduce. Θα περιγραφεί αρχικά το στήσιμο και η δομή του cluster υπολογιστών στο οποίο πραγματοποιήθηκαν τα πειράματα. Στη συνέχεια θα δοθούν οι μετρήσεις με σχολιασμό και συμπεράσματα. Η δομή σε αυτό το κομμάτι θα είναι τέτοια ώστε να φανεί πώς οι μετρήσεις επηρέασαν την εξέλιξη του προγράμματος.

5.1 Εισαγωγή

Για την πραγματοποίηση των μετρήσεων δημιουργήθηκε η ανάγκη να στηθεί το Hadoop MapReduce σε ένα δίκτυο υπολογιστών που αποτέλεσαν το cluster των δοκιμών. Το δίκτυο υπολογιστών στο οποίο πραγματοποιήθηκαν οι μετρήσεις αποτελούνταν από έναν τετραπύρηνο υπολογιστή που ανέλαβε τις διαδικασίες master και 12 διπύρηνους υπολογιστές που ανέλαβαν τις διαδικασίες slave. Τα δεδομένα ήταν αποθηκευμένα σε 3 αντίγραφα, ένα στον υπολογιστή master και 2 σε υπολογιστές slave. Οι υπολογιστές λειτουργούσαν με λειτουργικό σύστημα GNU/Linux και εγκαταστάθηκε η έκδοση 0.20.2 του Hadoop MapReduce. Όπως φαίνεται και στον πίνακα 5.1, οι υπολογιστές slave δεν είχαν όλοι την ίδια επεξεργαστική ισχύ, με δύο από αυτούς να είναι αρκετά πιο γρήγοροι. Το γεγονός αυτό επηρέασε τις μετρήσεις μας, μιας και κάποιες εργασίες τέλειωναν πολύ γρηγορότερα από άλλες. Το Hadoop MapReduce χρησιμοποιεί έναν μηχανισμό υποθετικής εκτέλεσης για να ισορροπήσει τις διαφορές σε υπολογιστική δύναμη μεταξύ των κόμβων του cluster. Αν κάποιος πυρήνας δεν έχει εργασία να εκτελέσει, τότε αναλαμβάνει μία από τι εργασίες που τρέχουν ήδη σε άλλο κόμβο. Κρατούνται τα αποτελέσματα της διεργασίας που θα τερματίσει πρώτη, ενώ η πιο αργή τερματίζεται. Η διαδικασία εγκατάστασης του Hadoop MapReduce είναι αρκετά απλή. Προαπαιτεί την εγκατάσταση της έκδοσης 6 της Sun Java και τη δυνατότητα επικοινωνίας μεταξύ όλων των υπολογιστών μέσω SSH. Συγκεκριμένα απαιτεί κοινό όνομα χρήστη και λειτουργία του SSH με αρχείο κλειδιού και απενεργοποιημένη την επιβεβαίωση κωδικού. Η λειτουργία του cluster είναι και αυτή αρκετά απλή για τον τελικό χρήστη. Αφού καθορίσει κάποιες μεταβλητές όπως το φάκελο αποθήκευσης προσωρινών δεδομένων και το όνομα των κόμβων slave, αρκεί να αρχίσει τους daemons του Hadoop MapReduce. Στη συνέχεια μπο-

ρεί να φορτώσει αρχεία στο HDFS, να αναθέσει στο cluster την εκτέλεση εργασιών και να παρακολουθήσει την πορεία τους μέσω Web GUI. Τα προβλήματα που αντιμετωπίσαμε γύρω από τη λειτουργία του cluster αφορούσαν κυρίως το γεγονός ότι κατά την ενεργοποίηση των daemons το script φαίνεται να τερματίζει κανονικά και το cluster να είναι διαθέσιμο, ενώ στην πραγματικότητα χρειάζεται ένα χρονικό διάστημα για να συντονιστούν οι κόμβοι μεταξύ τους. Δεν υπάρχει όμως επιβεβαίωση από τη γραμμή εντολών όταν ο συντονισμός ολοκληρωθεί. Επίσης ένα δεύτερο πρόβλημα ήταν ότι ο κάθε κόμβος του cluster πρέπει να έχει σε αρχείο την αντιστοιχία των IP όλων των κόμβων με το ατομικό τους hostname. Η μη χρήση του DNS hostname είναι αξιοπερίεργη και αποτελεί δυσκολία στο να στηθεί το Hadoop MapReduce σε μεγάλα cluster.

Λειτουργία:	Master	Slave	
Όνομα κόμβου:	bias	labmachine 3,12	labmachine *
CPU:	Q6600	E5300	E2220
Πυρήνες:	4	2	2
Χρονισμός:	2.4GHz	2.6GHz	2.4GHz
Cache:	2x4MB	2MB	1MB
RAM:	2GB	4GB	4GB

Πίνακας 5.1: Τα μηχανήματα του cluster που χρησιμοποιήθηκε για τις μετρήσεις

Οι εφαρμογές δοκιμάστηκαν αρχικά σε τοπική εγκατάσταση του Hadoop MapReduce. Το Hadoop διασφαλίζει ότι αν η εφαρμογή εκτελείται ομαλά δίνοντας σωστά αποτελέσματα τοπικά, τότε η λειτουργία της θα είναι ομαλή και σε οσοδήποτε μεγάλο cluster. Η είσοδος που χρησιμοποιήθηκε ήταν το ανθρώπινο γονιδίωμα. Το κατώφλι που χρησιμοποιήθηκε σε όλες τις μετρήσεις ήταν $t = 10^6$, όσο αναφέρεται και στην παρουσίαση του Trellis.Η παραλληλοποίηση δεν διασφαλίζει την επιτάχυνση μιας εργασίας. Όπως θα δούμε σε όλα μας τα πειράματα υπάρχουν δύο αντίθετες δυνάμεις από τις οποίες εξαρτάται η τελική βελτίωση του χρόνου εκτέλεσης για είσοδο συγκεκριμένου μεγέθους. Από τη μία γίνεται καταμερισμός μιας εργασίας σε περισσότερους υπολογιστές με αποτέλεσμα περισσότερες ώρες επεξεργασίας συσσωρευμένες σε λιγότερο πραγματικό χρόνο. Από την άλλη η διαδικασία επιβαρύνεται από διάφορες επιπλέον πράξεις που πρέπει να συμβούν. Αυτές είναι κυρίως η αρχική μεταφορά των δεδομένων στους κόμβους και η μεταφορά και διαλογή των ενδιάμεσων δεδομένων. Επιπλέον λόγοι για να μην μπορεί να επιταχυνθεί η διαδικασία αναλύονται στο [10]. Σε αυτή την εργασία αποδεικνύεται ότι πολύ μεγάλη σημασία για την απόδοση μιας εργασίας MapReduce έχει η επιλογή του αριθμού των mappers και reducers. Συγκεκριμένα, φαίνεται ότι η απόδοση διαφέρει για διαφορετικό πλήθος mappers. Η βέλτιστη απόδοση παρουσιάζεται όταν ο αριθμός των map σε κάθε κόμβο είναι ίσος με τον αριθμό των πυρήνων του κόμβου και για την περίπτωση της μελέτης που αναφέρουμε, ίσος με 8. Για μεγαλύτερο ή μικρότερο αριθμό mappers η απόδοση πέφτει. Επιπλέον, η αύξηση του πλήθους των mappers γραμμικά από 1 έως 8 δεν αυξάνει γραμμικά την απόδοση. Αυτό συμβαίνει γιατί οι περιπτώσεις που εξετάζονται στην συγκεκριμένη εργασία παρουσιάζουν bottleneck λόγω I/O. Η καθυστέρηση οφείλεται στη

χρήση μηχανημάτων με 8 πυρήνες που μοιράζονται 2 σκληρούς δίσκους. Οι συγγραφείς της εργασίας παρατήρησαν ότι η απόδοση τετραπλασιάζεται όταν από 1 mapper/κόμβο χρησιμοποιηθούν 4, αλλά αυξάνεται μόνο μιάμιση φορά όταν από 4 mappers/κόμβο χρησιμοποιηθούν 8. Γενικά είναι λογικό ότι η μέγιστη απόδοση επιτυγχάνεται όταν γίνεται πλήρης εκμετάλλευση των πυρήνων του cluster, χωρίς όμως και καμία διαδικασία να περιμένει σειριακά για την εκτέλεσή της. Είναι λογικό ότι όταν οι mappers είναι λιγότεροι από τους πυρήνες του συστήματος, υπάρχουν πυρήνες που δεν γίνεται χρήση τους και άρα προκαλείται καθυστέρηση, ενώ αν οι mappers είναι περισσότεροι από τους πυρήνες, τότε υπάρχει αδυναμία για παράλληλη εκτέλεση όλων των εργασιών ταυτόχρονα. Αντίθετα, οι εργασίες αυτές θα εκτελεστούν με ένα παράθυρο που αντιστοιχεί στο πλήθος των επεξεργαστών του συστήματος. Στην πράξη πολλές φορές ο αριθμός των mappers ή των reducers πρέπει να αυξηθεί, γιατί οι απαιτήσεις του προγράμματος σε μνήμη ξεπερνούν τη διαθέσιμη μνήμη του συστήματος. Επισης, αρχικές μας μετρήσεις έδειξαν ότι είναι επιθυμητό ο αριθμός των reducers να κρατηθεί χαμηλός, γιατί αλλιώς αυξάνεται δυσανάλογα ο χρόνος που χρειάζεται η φάση shuffle. Αξίζει εδώ να σημειωθούν οι προτεινόμενες τιμές για τους mappers και τους reducers από το ίδιο το MapReduce. Οι mappers μπορούν να είναι 10-100 φορές περισσότεροι σε σχέση με τους πυρήνες, ενώ οι reducers δεν προτείνεται να είναι περισσότεροι από 2 φορές το πλήθος των πυρήνων. Αξίζει τέλος να σημειωθεί ότι δεν παρατίθεται ο χρόνος shuffle για τα παρακάτω πειράματα. Αυτό συμβαίνει γιατί η φάση shuffle τρέχει παράλληλα με τη φάση map και πριν τη φάση reduce. Ο χρόνος shuffle που αναφέρει το Hadoop MapReduce εξαρτάται από την πορεία της φάσης map. Αν για παράδειγμα τελειώσουν όλοι οι mappers ταυτόχρονα, θα δούμε μεγάλους χρόνους shuffle, γιατί θα πρέπει σε πολύ λίγο χρονικό διάστημα να επεξεργαστεί πολλά δεδομένα. Αν αντίθετα έχουμε καθυστέρηση της φάσης map λόγω λίγων αργών mappers, τότε ο χρόνος shuffle θα είναι ελάχιστος, αφού θα έχει επεξεργαστεί το σύνολο των ενδιάμεσων δεδομένων εκτός από την έξοδο λίγων mappers στο τέλος. Έτσι με τη μέθοδο συλλογής αποτελεσμάτων που εφαρμόστηκε δεν μπορούν να εξαχθούν ασφαλή συμπεράσματα για το χρόνο της φάσης shuffle, αν και επηρεάζει τον συνολικό χρόνο εκτέλεσης της εργασίας.

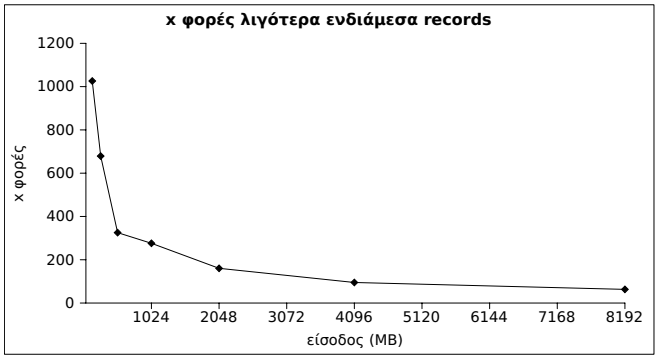
5.2 Κατασκευή προθεμάτων

Στο κεφάλαιο αυτό θα εξετάσουμε την αποτελεσματικότητα της φάσης κατασκευής προθεμάτων. Για τη φάση αυτή επιλέχθηκε μια λίγο διαφορετική υλοποίηση σε σχέση με το Trellis. Η εναλλακτική αυτή, σε συνδυασμό με τη χρήση ενός combiner για τη μείωση του πλήθους ζευγών των ενδιάμεσων εγγραφών. Η εναλλακτική αυτή λύση δεν φάνηκε να υστερεί ιδιαίτερα σε σχέση με τη μέθοδο του Trellis για αυτό το βήμα.

Όπως αναφέρθηκε στο κεφάλαιο της υλοποίησης, ο προγραμματισμός αυτής της φάσης υπήρξε εν μέρει προβληματικός λόγω περιορισμών της παρούσας έκδοσης του Hadoop MapReduce. Αν και αρχική μας θέληση ήταν η υλοποίηση της φάσης όπως και στο Trellis, δηλαδή σε κάθε σάρωση της ακολουθίας εισόδου να καταμετρούνται προθέματα διαφορετικού μήκους, αυτό δεν κατέστει δυνατό. Οι λόγοι αναλύονται στο κεφάλαιο της υλοποίησης. Στην πραγματικότητα, πειράματα με έκδοση η οποία εφαρμόζει την τακτική του Trellis και σύγκριση

του χρόνου εκτέλεσης με τον χρόνο της δικής μας υλοποίησης έδειξαν ότι η τελευταία δεν υστερεί. Η βελτίωση που παρατηρήθηκε για εισόδους μέχρι και 1024MB και κατώφλι $t = 10^6$ ήταν πρακτικά μηδαμινή.

Στο στάδιο της κατασκευής προθεμάτων έγινε χρήση του combiner. Ο combiner είναι τοπική κλήση της συνάρτησης reduce στα αποτελέσματα του mapper που βρίσκονται ακόμα στη μνήμη. Όπως αναφέρθηκε η εφαρμογή μας προσομοιάζει αρκετά την εφαρμογή επίδειξης του MapReduce που μετράει τη συχνότητα λέξεων σε ένα κείμενο. Συγκεκριμένα, στο τέλος της κλήσης μιας συνάρτησης map για ένα κομμάτι της εισόδου με μήκος n, μπορεί να υπάρχουν μέχρι και n ζευγάρια εξόδου της μορφής (πρόθεμα, 1). Η τοπική κλήση της συνάρτησης reduce θα περιορίσει τον όγκο των δεδομένων δίνοντας ως έξοδο ζευγάρια της μορφής (πρόθεμα, συχνότητα στο n-οστό κομμάτι). Έχει ενδιαφέρον να μετρήσουμε πόσο αποτελεσματική είναι αυτή η διαδικασία, συγκρίνοντας τα ζευγάρια εγγραφών εξόδου της map και της combine. Στο σχήμα 5.1 βλέπουμε την αποτελεσματικότητα εκφρασμένη ως κλάσμα $\frac{map}{combine}$.



Σχήμα 5.1: Η αποτελεσματικότητα της συνάρτησης combine στο στάδιο κατασκευής προθεμάτων

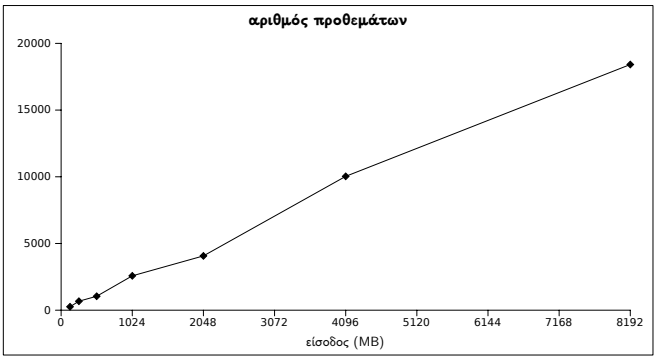
Η απόδοση βελτιώνεται εξαιρετικά με την χρήση του combiner σε αυτή την εργασία, καθώς μειώνει τόσο τον όγκο των δεδομένων όσο και τον αριθμό των εγγραφών. Σε δοκιμή για είσοδο 32MB η έκδοση της εργασίας που δεν χρησιμοποιούσε combiner χρειάστηκε διπλάσιο χρόνο από την έκδοση που κάνει χρήση του combiner.

Γενικά το στάδιο αυτό μπορεί να εκτελεστεί γρήγορα και σε έναν υπολογιστή για τα μεγέθη που μας ενδιαφέρουν. Θεωρητικά τουλάχιστον η μέθοδος που χρησιμοποιήθηκε, δηλαδή η κάθε εργασία να μετράει συχνότητες μόνο για ένα μήκος προθέματος, έχει σαφή μειονεκτήματα σε σχέση με τη μέθοδο του Trellis, καθώς αναγκάζεται να σαρώσει περισσότερες φορές την είσοδο. Το πλεονέκτημά της είναι πως τα ενδιάμεσα δεδομένα είναι τα ελάχιστα δυνατά. Στην πράξη αυτή η διαφορά δεν φάνηκε να επηρεάζει τον χρόνο εκτέλεσης των εργασιών. Το πλήθος των προθεμάτων για κατώφλι $t = 10^6$ για τις εισόδους που χρησιμοποιήθηκαν παρατίθενται στο σχήμα 5.2.

5.3 Κατασκευή προθεματικών δέντρων

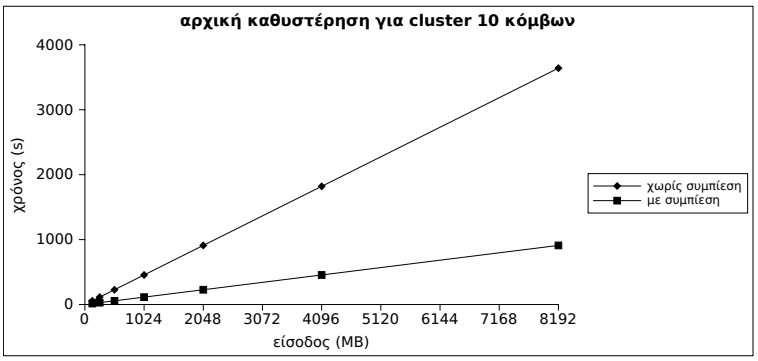
Σε αυτό το κεφάλαιο θα εξεταστεί η απόδοση της φάσης κατασκευής δέντρων επιθεμάτων για τους αλγορίθμους που υλοποιήθηκαν, τόσο για αυξανόμενα μεγέθη εισόδου, όσο και για μεταβαλλόμενα μεγέθη cluster.

Όπως αναφέρθηκε και στο κεφάλαιο της υλοποίησης, η απαίτηση της φάσης της συγχώνευσης για παρουσία όλης της ακολουθίας εισόδου στη μνήμη κατά την εκτέλεση μας οδήγησε στο να παρακάμψουμε τον μηχανισμό διαμοιρασμού της εισόδου του MapReduce, αλλά να το αναγκάσουμε να μεταφέρει πριν την έναρξη των διαδικασιών ολόκληρη την ακολουθία σε όλους τους κόμβους μέσω του μηχανισμού distributed cache. Μία από τις πρώτες παρατηρήσεις που έγιναν ήταν πως δίνοντας την είσοδο μέσω του distributed cache οι συναρτήσεις map αργούσαν να αρχίσουν. Αντίθετα, στις εργασίες κατασκευής προθεμάτων από τις πρώτες στιγμές της έναρξης της εργασίας οι συναρτήσεις map επεξεργάζονταν δεδομένα. Αυτό συμβαίνει γιατί η είσοδος που δίνεται από InputFormat, όπως στη φάση κατασκευής προθεμάτων, μεταφέρεται με streaming στους κόμβους. Έτσι η επεξεργασία μπορεί να αρχίσει πριν ολοκληρωθεί η μεταφορά των δεδομένων. Αντίθετα, το distributed cache μεταφέρει τα δεδομένα πριν την έναρξη της επεξεργασίας, μπλοκάροντας τις συναρτήσεις. Μέλημα μας είναι να περιορίσουμε όσο το δυνατόν αυτό το αρχικό διάστημα απραξίας των κόμβων του cluster. Οι δύο παράγοντες που συντελούν σε αυτό είναι η ταχύτητα του δικτύου και το μέγεθος των δεδομένων που μεταφέρονται. Η ταχύτητα του δικτύου εξαρτάται από το υλικό, που αποτελείται από κάρτες δικτύου των 10MBit. Αντίθετα, το μέγεθος των δεδομένων μπορεί να αλλάξει, πράττοντας τη μετατροπή του αρχείου εισόδου σε πίνακα από bytes πριν την έναρξη της εργασίας και μεταφέροντας το μέσω του distributed cache υπό την μορφή της κλάσης BytesWritable. Όπως έχουμε αναφέρει για να είναι δυνατή η φόρτωση μεγάλων αρχείων εισόδου στη μνήμη εφαρμόζεται μια τεχνική συμπίεσης που αντιστοιχεί ένα χαρακτήρα της ακολουθίας εισόδου σε 2 bits. Στη συνέχεια 4 χαρακτήρες μπαίνουν σε 1 byte. Έτσι επιτυγχάνεται μείωση του όγκου των δεδομένων κατά 4 φορές, αλλά και αντίστοιχη μείωση στην αρχική καθυστέρηση της εργασίας. Οι αντίστοιχες μετρήσεις φαίνονται στο σχήμα 5.3. Αξίζει να αναφερθεί ότι σε αρχική φάση των υλοποιήσεων η ακολουθία δεν φορτωνόταν στη μνήμη αλλά γινόταν χρήση ενός μεγάλου buffer πάνω στο αρχείο της ακολουθίας, αλλά η



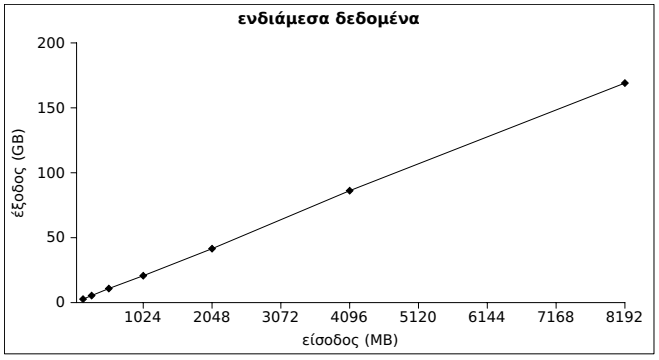
Σχήμα 5.2: Το πλήθος των προθεμάτων για τις εισόδους που χρησιμοποιήθηκαν.

προσέγγιση αυτή έδινε πολύ αργούς χρόνους εκτέλεσης. Αξίζει τέλος να σημειώθεί ότι ο όγκος των ενδιάμεσων δεδομένων είναι πρακτικά ίδιος με τον όγκο των τελικών δεδομένων. Οι επιπλέον εσωτερικοί κόμβοι που δημιουργούνται κατά τη διαδικασία της συγχώνευσης δεν φαίνεται να επηρεάζουν σημαντικά τον όγκο των δεδομένων.



Σχήμα 5.3: Αρχική καθυστέρηση χωρίς συμπίεση της ακολουθίας εισόδου και με συμπίεση

Όπως αναμέναμε, η μνήμη που καταλαμβάνουν τα δέντρα επιθεμάτων που κατασκευάζουμε είναι πολλαπλάσια της αρχικής ακολουθίας που περιγράφουν. Όπως φαίνεται και στο σχήμα 5.4, χρειαζόμαστε περίπου 20 φορές περισσότερη μνήμη για τα δέντρα μας σε σχέση με το μέγεθος της εισόδου. Το μέγεθος αυτό παρόλα αυτά είναι ιδιαίτερα μικρό και οφείλεται στις συναρτήσεις που καλούνται για την αποθήκευση των δομών στον σκληρό δίσκο. Συγκεκριμένα, χρησιμοποιήθηκαν κατά την υλοποίηση συναρτήσεις οι οποίες συμπιέζουν όσο το δυνατόν περισσότερο τους αντίστοιχους τύπους για να τους αποθηκεύσουν. Για παράδειγμα, ένας αριθμός τύπου int στην Java απαιτεί πάντα 4 byte στη μνήμη, ακόμα και αν περιγράφει έναν πολύ μικρό αριθμό που θα χωρούσε σε ένα και μόνο byte. Η συνάρτηση αποθήκευσης του, writeVInt, εξαλείφει για τους μικρούς αριθμούς τα αχρησιμοποίητα bytes και αποθηκεύει μόνο τη χρήσιμη πληροφορία.



Σχήμα 5.4: Ο όγκος των ενδιάμεσων δεδομένων

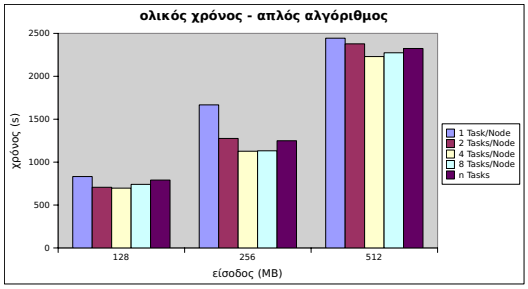
5.3.1 Συμπεριφορά των αλγορίθμων για διαφορετικά μεγέθη εισόδου

Οι υλοποιήσεις που παρουσιάστηκαν στο προηγούμενο κεφάλαιο πρέπει να προσαρμοστούν για να δέχονται αρχεία εισόδου μεγαλύτερα από 2GB. Οι αριθμοί που περιγράφουν την αρχική και τελική θέση της κάθε ακμής είναι χαρακτηρισμένοι ως int. Η Java διαθέτει μόνο signed ints που φτάνουν μέχρι το $2^{31} = 2GB$. Έτσι ο κώδικας πρέπει να μετατραπεί, αντικαθιστώντας τις τιμές αυτές με long. Στην πράξη αυτό δεν επηρεάζει την απόδοση του προγράμματος μας αρνητικά, καθώς όλοι οι υπολογιστές του cluster είχαν πυρήνες 64 bit και εγκατεστημένη έκδοση x86_64 του λειτουργικού συστήματος. Όλες οι μετρήσεις, ακόμα και για μικρότερες εισόδους, έγιναν με αυτές τις εκδόσεις.

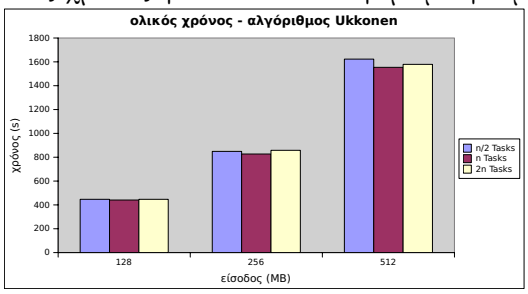
Στην πράξη η πρώτη από τις δύο υλοποιήσεις του αλγορίθμου του Ukkonen, αν και ήταν κατάλληλη για μια πρακτική παρουσίαση υλοποίησης του αλγορίθμου, αποδείχθηκε εξαιρετικά αργή. Συγκεκριμένα ήταν αργότερη από τον απλοϊκό αλγόριθμο. Στις άλλες υλοποιήσεις της συνάρτησης κατασκευής των προθεματικών υποδέντρων παρατηρούμε ότι υπάρχει μεγάλη διαφορά στις απαιτήσεις μνήμης. Συγκεκριμένα το πλεονέκτημα της απλοϊκής έκδοσης είναι ότι παράγοντας μικρά δέντρα μέσα σε loop έχει μικρές απαιτήσεις σε μνήμη. Αντίθετα, ο αλγόριθμος του Ukkonen απαιτεί την κατασκευή του δέντρου ολόκληρου του κομματιού στη μνήμη και κατόπιν την εξαγωγή των υποδέντρων από αυτό. Πρακτικά αυτό σημαίνει ότι ο απλοϊκός αλγόριθμος έχει τη δυνατότητα να τρέξει με λίγους mappers, ενώ ο αλγόριθμος του Ukkonen απαιτεί πάρα πολλούς. Ο όγκος των δεδομένων που δίνουν και οι δύο αλγόριθμοι είναι ίδιος, αλλά περισσότεροι mappers σημαίνουν περισσότερες ενδιάμεσες εγγραφές. Ο αλγόριθμος σύγκρισης που χρησιμοποιείται στη φάση shuffle είναι ο mergesort με πολυπλοκότητα O(nlogn). Ο αλγόριθμος της συγχώνευσης είναι επίσης O(nlogn) πολυπλοκότητας. Τα ερωτήματα λοιπόν είναι πόσο γρηγορότερα τρέχει ο αλγόριθμος του Ukkonen κατά τη φάση του map και κατά πόσο επιβαρύνονται οι φάσεις του shuffle και του reduce λόγω των επιπλέον ενδιάμεσων εγγραφών.

Αρχικά κάναμε μετρήσεις για μικρές εισόδους προσπαθώντας να εξάγουμε κάποια πρώτα συμπεράσματα για τη συμπεριφορά των αλγορίθμων, καθώς και τον ιδανικό αριθμό mappers. Όπως παρατηρούμε στο σχήμα 5.5 ο απλός αλγόριθμος δοκιμάστηκε σε 7 κόμβους για 7, 14, 28, 56 και n mappers, όπου n αριθμός τέτοιος ώστε να λάβει κάθε Task 1MB της εισόδου, για παράδειγμα n=128 για 128MB εισόδου. Παρατηρούμε ότι η καλύτερη απόδοση επετεύχθει όταν είχαμε 4 ή 8 Tasks ανά κόμβο. Ειδικά η δοκιμή για 1 Task/Node αποδείχθηκε πολύ αργή, συμφωνώντας με τη βιβλιογραφία. Στο σχήμα 5.6 παρατηρούμε τη συμπεριφορά του αλγορίθμου του Ukkonen για διαφορετικό αριθμό mappers. Εδώ η διακύμανση είναι πολύ μικρότερη σε σχέση με τον απλό αλγόριθμο. Μια καλή τιμή φαίνεται να είναι οι n mappers, δηλαδή η ανάθεση 1MB της εισόδου σε κάθε mapper. Μέσω αυτών των παρατηρήσεων θα προχωρήσουμε σε μετρήσεις μεγαλύτερου μεγέθους, αναθέτοντας περίπου 16MB σε κάθε mapper για τον απλό αλγόριθμο και 1MB σε κάθε mapper για τον αλγόριθμο του Ukkonen. Ουσιαστικά στα παραπάνω διαγράμματα βλέπουμε τις δύο αντίθετες δυνάμεις που προαναφέραμε: από τη μία περισσότεροι mappers αναλαμβάνουν ο καθένας μικρότερη είσοδο με αποτέλεσμα επι-

τάχυνση της φάσης map. Όμως περισσότεροι mappers σημαίνουν περισσότερες ενδιάμεσες εγγραφές και επιβάρυνση της φάσης reduce που πρέπει να συγχωνεύσει περισσότερα υποδέντρα. Έτσι για λίγους mappers έχουμε καθυστέρηση λόγω μεγάλης εισόδου στη φάση map, ενώ για πολλούς mappers έχουμε καθυστέρηση λόγω μεγάλης εισόδου στη φάση reduce. Το σημείο βέλτιστης απόδοσης βρίσκεται κάπου ενδιάμεσα.



Σχήμα 5.5: Ολικός χρόνος για τον απλό αλγορίθμο για μικρές εισόδους



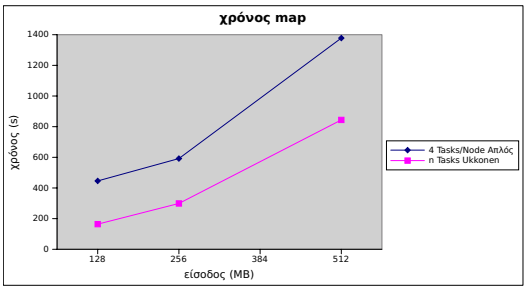
Σχήμα 5.6: Ολικός χρόνος για τον αλγορίθμο Ukkonen για μικρές εισόδους

Συγκρίνουμε τώρα την καλύτερη απόδοση του απλού αλγορίθμου και του αλγορίθμου του Ukkonen για μικρές εισόδους. Στο σχήμα 5.7 παρατηρούμε ότι ο αλγόριθμος του Ukkonen είναι σαφώς γρηγορότερος στη φάση του map και μάλιστα παρουσιάζει μικρότερη κλίση, που είναι αυτό που αναμέναμε από έναν γραμμικό αλγόριθμο σε σχέση με έναν μη γραμμικό. Στο σχήμα 5.8 παρατηρούμε ότι ο μεγάλος αριθμός ενδιάμεσων εγγραφών επιβαρύνει ελαφρά την εκτέλεση του αλγορίθμου του Ukkonen, αλλά όχι τόσο ώστε να τον καταστήσει συνολικά πιο αργό. Επαναλαμβάνουμε σε αυτό το σημείο ότι ο όγκος των ενδιάμεσων εγγραφών είναι ίσος για τους δύο αλγορίθμους, αλλά η ανάγκη του αλγορίθμου του Ukkonen για μεγαλύτερο τεμαχισμό της εισόδου και περισσότερους mappers οδηγεί σε πολλαπλάσιες ενδιάμεσες εγγραφές. Αξίζει να σημειωθεί ότι δεν είναι δυνατό να προγραμματιστεί αποδοτικός combiner για τη μείωση του πλήθους των ενδιάμεσων εγγραφών. Η συγχώνευση των υποδέντρων απαιτεί την παρουσία της ακολουθίας στη μνήμη και η συνάρτηση combine καλείται όταν συγκεντρωθούν αρκετές εγγραφές στη μνήμη και πριν τις εγγράψει στον σκληρό δίσκο. Η μνήμη του συστήματος απλά δεν επαρκεί για να διαχειριστεί τον όγκο όλων αυτών των δεδομένων, καθώς παράλληλα τρέχει η διαδικασία του map. Χαρακτηριστικά αναφέρουμε ότι η έκδοση με τον combiner έκανε πολλαπλάσιο χρόνο να τελειώσει από την απλή έκδοση λόγω συνεχούς swapping στο δίσκο.

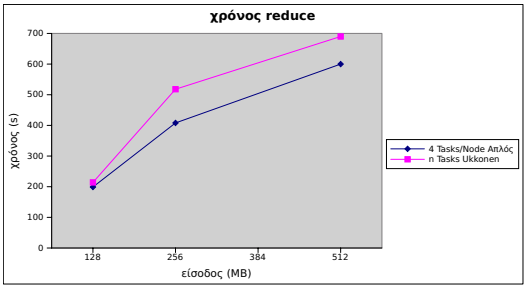
Στη συνέχεια αυξήθηκε το μέγεθος της εισόδου. Κανονικά δεν θα ήταν δυνατό να τρέξει η

φάση reduce για τόσο μεγάλες εισόδους, καθώς τα μηχανήματα δεν είχαν αρκετή μνήμη. Κάθε σύστημα διέθετε 4GB μνήμης και δύο πυρήνες. Αυτά τα 4GB μνήμης κατά τη φάση reduce θα έπρεπε να φιλοξενούν 2 φορές την ακολουθία εισόδου, μία για κάθε ξεχωριστό thread που εκτελεί τη συνάρτηση reduce, τον χώρο που χρειάζονται τα δέντρα για να συγχωνευθούν και όσο χώρο χρειάζονται οι εφαρμογές του λειτουργικού. Για να πραγματοποιηθούν οι παρακάτω μετρήσεις ήταν αναγκαίο να μειωθούν τα threads του reducer σε 1 ανά κόμβο και να διακοπεί η υποθετική εκτέλεση. Αποτέλεσμα μια μικρή αύξηση στους χρόνους εκτέλεσης. Στο σχήμα 5.9 φαίνονται τα αποτελέσματα των μετρήσεων. Ο απλός αλγόριθμος παρουσιάζει μεγάλες καθυστερήσεις για εισόδους μεγαλύτερες των 2 GB και δεν ήταν εφικτό να μετρηθεί ο χρόνος για είσοδο 8GB. Αντίθετα ο αλγόριθμος του Ukkonen συμπεριφέρεται αρκετά καλά. Οι χρόνοι για τη φάση του map φαίνονται στο σχήμα 5.10.

Στο σημείο αυτό είναι χρήσιμο να συγκρίνουμε τους χρόνους εκτέλεσης της εφαρμογής μας με τους χρόνους του Trellis. Καταρχήν ανατρέχουμε στη βιβλιογραφία. Η εργασία του Trellis [14] αναφέρει ότι το πρόγραμμα δοκιμάστηκε σε υπολογιστή server (διπύρηνος υπολογιστής server με 32GB RAM και SCSI σκληρούς δίσκους) και κατάφερε να ευρετηριοποιήσει το ανθρώπινο γονιδίωμα σε περίπου 4 ώρες. Ο χρόνος μας για την αντίστοιχη είσοδο είναι περίπου 3 ώρες με χρήση του αλγορίθμου του Ukkonen, ενώ προφανώς η εκτέλεση του Trellis σε έναν από του κόμβους του συστήματος θα χρειαζόταν περισσότερη ώρα από αυτή που αναφέρεται στο paper. Άρα προκύπτει μια μικρή βελτίωση στον χρόνο εκτέλεσης. Η εργασία του Wavefront [8] περιλαμβάνει το σχήμα 5.11 που δείχνει την αδυναμία του Trellis για πολύ μεγάλες εισόδους. Όπως βλέπουμε ο χρόνος εκτέλεσης της φάσης της συγχώνευσης



Σχήμα 5.7: Σύγκριση απλού αλγορίθμου και αλγορίθμου Ukkonen στη φάση map για μικρές εισόδους

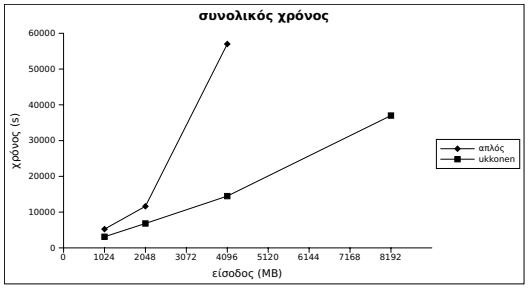


Σχήμα 5.8: Σύγκριση απλού αλγορίθμου και αλγορίθμου Ukkonen στη φάση reduce για μικρές εισόδους

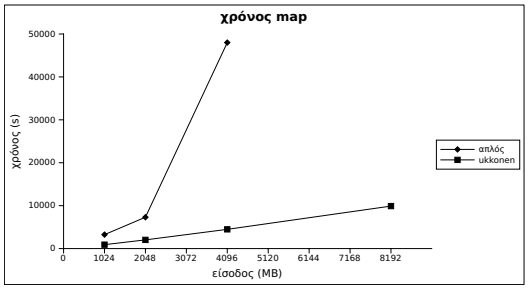
αυξάνεται εκθετικά για εισόδους μεγαλύτερες από 3G (ανθρώπινο γονιδίωμα). Όπως φαίνεται από την εργασία μας, η παραλληλοποίηση του σταδίου αυτού βελτιώνει τη συμπεριφορά του προγράμματος, επιτρέποντας αποδοτική ευρετηριοποίηση μεγαλύτερων ακολουθιών. Φυσικά αυτή η συμπεριφορά δεν θα διατηρηθεί για μεγαλύτερες εισόδους: όπως και ο μη γραμμικός αλγόριθμος κατασκευής προθεματικών υποδέντρων παρουσίαζε καλή συμπεριφορά μέχρι τα 2G, αλλά μετά ο χρόνος εκτέλεσης του αυξήθηκε εκθετικά, έτσι και η διαδικασία της παράλληλης συγχώνευσης θα παρουσιάσει δυσανάλογα μεγάλη αύξηση στο χρόνο εκτέλεσης για μεγαλύτερες εισόδους. Η εργασία του Wavefront [8] αναφέρει ότι οποιαδήποτε προσπάθεια παραλληλοποίησης αλγορίθμου του είδους Trellis θα αποτύχει λόγω της ανάγκης μεταφοράς μεγάλου όγκου δεδομένων. Γενικά ο μεγάλος όγκος δεδομένων είναι υπαρκτό πρόβλημα του αλγορίθμου, αλλά στην πράξη σημειώνονται βελτιώσεις στα σημεία που αναφέρθηκαν. Αναμένεται οι χρόνοι εκτέλεσης να είναι πολύ καλύτεροι για μεγαλύτερα cluster, ιδιαίτερα για τη φάση map, η οποία χρειάζεται αριθμό tasks πολλαπλάσιο του πλήθους των πυρήνων του cluster για να εκτελεστεί.

5.3.2 Συμπεριφορά των αλγορίθμων για διαφορετικά μεγέθη cluster

Σε αυτό το πείραμα δοκιμάστηκε η συμπεριφορά των δύο αλγορίθμων για διαφορετικά μεγέθη cluster. Συγκεκριμένα δοκιμάστηκαν οι δύο αλγόριθμοι σε cluster 3, 6, 9 και 12 υπολογιστών για μεγέθη εισόδου 128, 256, 512 και 1024 MB. Τα αποτελέσματα φαίνονται στα σχήματα 5.12 και 5.13. Η μορφή των σχεδιαγραμμάτων είναι ίδια και για τους δύο αλγορίθ-

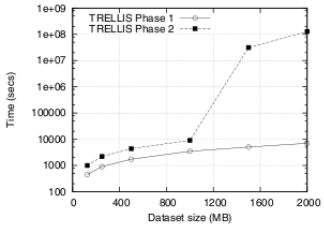


Σχήμα 5.9: Σύγκριση απλού αλγορίθμου και αλγορίθμου Ukkonen ως προς τον συνολικό χρόνο για μεγάλες εισόδους

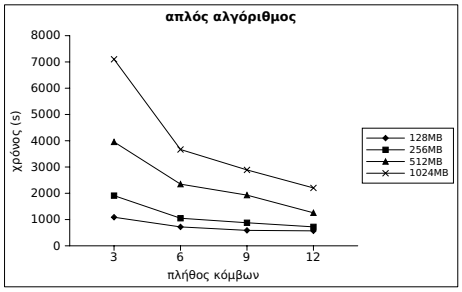


Σχήμα 5.10: Σύγκριση απλού αλγορίθμου και αλγορίθμου Ukkonen ως προς το χρόνο της φάσης map για μεγάλες εισόδους

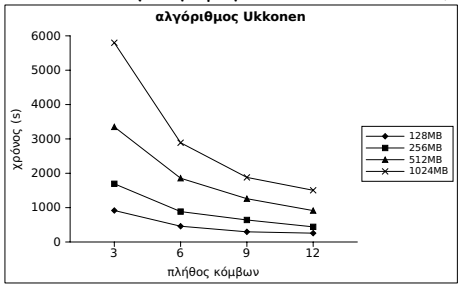
μους. Από τα σχεδιαγράμματα αυτά μπορούν να εξαχθούν τα διάφορα συμπεράσματα. Αρχικά παρατηρούμε ότι όσο μεγαλύτερη είναι η είσοδος τόσο μεγαλύτερη βελτίωση θα προκύπτει από την αύξηση του αριθμού των κόμβων του δικτύου: η κλίση της ευθείας για τα 128 MB είναι πολύ μικρότερη από ότι για τα 1024MB. Ακόμα, η βελτίωση της απόδοσης λόγω αύξησης των κόμβων του δικτύου είναι πεπερασμένη: η αύξηση του πλήθους των κόμβων από 6 σε 9 δίνει πολύ μικρότερη βελτίωση από την αύξηση από 3 σε 6. Από κάποιο σημείο αύξησης του πλήθους των κόμβων και μετά η βελτίωση της απόδοσης θα είναι αμελητέα. Επιπλέον, αύξηση του αριθμού των κόμβων στο δίκτυο και για μικρότερες εισόδους επέρχεται γρηγορότερα κορεσμός στη βελτίωση της απόδοσης: η κλίση της ευθείας για τα 128MB αλλάζει πολύ λιγότερο από την κλίση της ευθείας για είσοδο 1024MB. Τέλος μεγαλύτερα cluster κάνουν καλύτερο scale από μικρότερες σε μεγαλύτερες εισόδους: η διαφορά χρόνου εκτέλεσης για τους 12 κόμβους μεταξύ των εισόδων 128MB και 1024MB είναι πολύ μικρότερη από ότι η αντίστοιχη διαφορά στους 3 κόμβους.



Σχήμα 5.11: Η συμπεριφορά του Trellis για μεγάλες εισόδους



Σχήμα 5.12: Η συμπεριφορά του απλού αλγορίθμου



Σχήμα 5.13: Η συμπεριφορά του αλγορίθμου του Ukkonen

5.4 Συμπεράσματα

Από τις παραπάνω μετρήσεις μπορούμε να εξάγουμε τα εξής συμπεράσματα:

1. Μεγαλύτερα cluster μπορούν να χειριστούν καλύτερα μεγάλες εισόδους και να επιταχύνουν τη διαδικασία εκτέλεσης, αρκεί να χωρίσουμε την είσοδο τουλάχιστον σε τόσα κομμάτια όσοι και οι πυρήνες του δικτύου και να διασφαλίσουμε ότι η μνήμη που δίνεται σε κάθε mapper επαρκεί για την επεξεργασία του κομματιού.
2. Η συμπεριφορά ενός μη γραμμικού αλγορίθμου βελτιώνεται μέχρι ένα μέγεθος εισόδου λόγω της παράλληλης εκτέλεσης του.
3. Η διαδικασία μπορεί να παραλληλοποιηθεί δίνοντας βελτίωση στον όγκο των δεδομένων που μπορούν να ευρετηριοποιηθούν αποδοτικά.
4. Οι χρόνοι που μετρήθηκαν για το μέγεθος του cluster μας έδειξαν βελτίωση στο χρόνο εκτέλεσης και αποδοτική ευρετηριοποίηση για μεγαλύτερες εισόδους σε σχέση με το Trellis.

Κεφάλαιο 6

Επίλογος και μελλοντικές εργασίες

6.1 Σύνοψη και συμπεράσματα

Αντικείμενο της διπλωματικής αποτέλεσε η μελέτη της κατασκευής δέντρων επιθεμάτων που δεν χωράνε στην κεντρική μνήμη. Στο πλαίσιο αυτό μελετήθηκαν αλγόριθμοι που έχουν προταθεί στο παρελθόν και ιδιαίτερα ο αλγόριθμος Trellis. [14]. Στη συνέχεια η βασική ιδέα του αλγορίθμου Trellis χρησιμοποιήθηκε ως βάση για την υλοποίηση ενός παράλληλου προγράμματος κατασκευής δέντρων επιθεμάτων με χρήση της τεχνολογίας Hadoop MapReduce. Η **θεωρητική μελέτη** επικεντρώθηκε στους αλγορίθμους και τις τεχνικές που έχουν προταθεί για την αποδοτική κατασκευή δέντρων επιθεμάτων. Δόθηκε έμφαση στον αλγόριθμο του Ukkonen για αποδοτική κατασκευή στη μνήμη και στον αλγόριθμο Trellis για την αποδοτική κατασκευή δέντρων στο δίσκο. Ακόμα μελετήθηκε η προσέγγιση της Hunt που κατακερματίζει το δένδρο με βάση προθεμάτων σταθερού μήκους, οι αλγόριθμοι DynaCluster και TOP-Q που χρησιμοποιούν έξυπνες στρατηγικές buffering, ο αλγόριθμος TDD που υλοποιεί μια παραλλαγή του αλγοριθμου wotdeager και τέλος ο πρώτος αλγόριθμος εξειδικευμένος για παράλληλη επεξεργασία, Wavefront, που κατακερματίζει το δέντρο με βάση τη θέση του πρώτου χαρακτήρα της κάθε ακμής στον κείμενο. Παράλληλα, έγινε παρουσίαση των τεχνικών που έχουν αναπτυχθεί για το προσεγγιστικό ταίριασμα προτύπων με χρήση δέντρων επιθεμάτων, έτσι ώστε να τονιστεί η σημασία της δομής αυτής στον κλάδο της Υπολογιστικής Βιολογίας. Τέλος, αναλύθηκε η δομή και λειτουργία του Hadoop MapReduce που αποτέλεσε την πλατφόρμα υλοποίησης του προγράμματος που κατασκευάστηκε στο πλαίσιο της διπλωματικής αυτής εργασίας. Στις **υλοποιήσεις** δόθηκε βάρος στην παραλληλοποίηση της διαδικασίας κατασκευής υποδέντρων και συγχώνευσης αυτών. Συγκεκριμένα υλοποιήθηκε ένας απλός, μη γραμμικός αλγόριθμος, καθώς και ο αλγόριθμος του Ukkonen. Επίσης χρησιμοποιήθηκε μια δεύτερη υλοποίηση του αλγορίθμου του Ukkonen από βιβλιοθήκη. Κατασκευάστηκε ακόμα κατάλληλος αλγόριθμος συγχώνευσης υποδέντρων. Τέλος προγραμματίστηκε και μια απλή έκδοση υπολογισμού προθεμάτων. Οι **μετρήσεις** μας συνέκριναν δύο από τις τρεις υλοποιήσεις που έγιναν για την κατασκευή υποδέντρων ως προς την αποτελεσματικότητα τους για

αυξανόμενα μεγέθη εισόδου και για αυξανόμενο μέγεθος cluster. Γενικά στη βιβλιογραφία αναφέρεται ότι η προσπάθεια παραλληλοποίησης αλγορίθμου αυτού του είδους αποτυγχάνει λόγω της ανάγκης μεταφοράς μεγάλου όγκου δεδομένων [8]. Μέσα από την εργασία αυτή αποδεικνύεται ότι η αποδοτικότητα του αλγορίθμου βελτιώνεται ελαφρώς, αλλά κυρίως αυξάνεται το όριο της ακολουθίας εισόδου που μπορεί να ευρετηριοποιηθεί αποδοτικά. Έτσι, η εργασία αυτή μπορεί να αποτελέσει τη βάση για ένα τελικό πρόγραμμα κατασκευής δέντρων επιθεμάτων στο σκληρό δίσκο το οποίο να δίνει καλά αποτελέσματα για ακολουθίες μέχρι μια τάξη μεγέθους παραπάνω από το Trellis (από μερικά GB σε μερικές δεκάδες GB), όταν το κόστος υλικού πρέπει να κρατηθεί χαμηλά και δεν μπορεί να χρησιμοποιηθεί ο αλγόριθμος Wavefront, που απαιτεί χρήση υπερυπολογιστών.

6.2 Μελλοντικές Εργασίες

Η κυριότερη μελλοντική εργασία για να αποτελέσουν οι παραπάνω υλοποιήσεις ένα ενιαίο σύνολο είναι η υποστήριξη ανάκτησης των συνδέσμων επιθεμάτων. Όπως είδαμε, οι σύνδεσμοι επιθέματος χρησιμοποιούνται από τους αλγορίθμους προσεγγιστικού ταιριάσματος για ταχύτερη διάσχιση του δέντρου. Όμως με τη διαδικασία του κατακερματισμού του δέντρου επιθεμάτων σε προθεματικά δέντρα οι περισσότεροι σύνδεσμοι χάνονται. Η κύρια δυσκολία στην προσπάθεια παραλληλοποίησης αυτού του σταδίου είναι ότι ένας σύνδεσμο επιθέματος ενός προθεματικού δέντρου μπορεί να δείχνει σε οποιοδήποτε άλλο προθεματικό δέντρο. Έτσι, σύμφωνα τουλάχιστον με τον αλγόριθμο του Trellis, για να κάνει ένας κόμβος πλήρη ανάκτηση των συνδέσμων επιθέματος ενός δέντρου, πρέπει να έχει τοπικά όλα τα άλλα δέντρα. Ο όγκος των δεδομένων που θα πρέπει να μεταφερθεί είναι απαγορευτικός. Ωστόσο είναι ένα ζήτημα που χρίζει περισσότερης έρευνας.

Ένα σημείο που μπορεί να βελτιωθεί είναι ο χώρος που πιάνει το δέντρο επιθεμάτων στο δέντρο. Συγκεκριμένα, η υλοποίηση που χρησιμοποιήθηκε ήταν αρκετά σπάταλη όταν ήταν φορτωμένη στη μνήμη και μπορεί να απλοποιηθεί για περαιτέρω μείωση των απαιτήσεων μνήμης κατά την εκτέλεση. Αυτό συμβαίνει λόγω της αναδρομικής μορφής του δέντρου που έχει ανάγκη από πολλά references. Η νέες δομές δεδομένων δεν θα παρουσιάσουν όμως βελτίωση στον όγκο των ενδιάμεσων και τελικών δεδομένων, αφού κατά την αποθήκευση των δομών δεν αποθηκεύονται τα references αυτά, αλλά μόνο η αναγκαία πληροφορία, με τον βέλτιστο μάλιστα τρόπο.

Επίσης μπορούν να χρησιμοποιηθούν και οι διάφορες βελτιώσεις που έχουν προταθεί για το Trellis από τους ίδιους τους δημιουργούς του στην εργασία για το Trellis+ [15]. Το Trellis+ χρησιμοποιεί δύο διαφορετικά κατώφλια t_p και t_m αντί για ένα ενιαίο. Το πρώτο διασφαλίζει ότι η είσοδος χωρίζεται σε κομμάτια με τέτοιο μέγεθος ώστε το δέντρο επιθεμάτων που προκύπτει να χωράει στη μνήμη, ενώ το δεύτερο διασφαλίζει ότι το προθεματικό δέντρο για το πρόθεμα p χωράει στη μνήμη. Το Trellis χρησιμοποιεί ως ενιαίο κατώφλι το μικρότερο των t_p, t_m. Όμως πάντα ισχύει $t_m < t_p$. Άρα με αυτή την παρατήρηση μπορούμε να χωρίζουμε την είσοδο σε μεγαλύτερα κομμάτια, μειώνοντας τον αριθμό τους για μια δεδομένη είσοδο και επιταχύνοντας ταυτόχρονα την διαδικασία της συγχώνευσης. Μια δεύτερη

ομάδα βελτιώσεων εξαλείφει την ανάγκη παρουσίας όλης της ακολουθίας εισόδου στη μνήμη κατά τη διάρκεια της συγχώνευσης. Αρχικά γίνεται η παρατήρηση ότι η ετικέτα μιας εσωτερικής ακμής που περιγράφεται από την αρχική και τελική θέση των χαρακτήρων της μέσα στην ακολουθία εμφανίζεται αυτούσια και σε άλλες θέσεις μέσα στην ακολουθία. Έτσι αν οι συμβολοσειρές [0,1] και [1000,1001] είναι και οι δύο "AT", τότε μπορούμε να αλλάξουμε την αρχική και τελική θέση της ακμής [1000,1001] σε [0,1] χωρίς λάθος σε καμία διαδικασία μας. Μάλιστα μετρήθηκε ότι το 97% των εσωτερικών ακμών του ανθρώπινου γονιδιώματος μπορούν να μετατοπιστούν στους 2M πρώτους χαρακτήρες της ακολουθίας. Για να καταστεί αυτό δυνατό, αρκεί να κρατηθεί ένα δέντρο επιθεμάτων, καθώς και η ίδια η ακολουθία, για τους 2M πρώτους χαρακτήρες. Έτσι οι απαιτήσεις μνήμης για την ευρετηριοποίηση του ανθρώπινου γονιδιώματος μειώνονται από 700MB σε λίγες δεκάδες MB. Η βελτιστοποίηση αυτή δεν μπορεί να εφαρμοστεί για τις ακμές που καταλήγουν σε φύλλο, γιατί αυτές συνήθως έχουν πολύ μεγαλύτερες ετικέτες. Όμως και σε αυτές μπορεί να βελτιωθούν οι απαιτήσεις τους σε μνήμη, καθώς ισχύει ότι κατά τη διαδικασία της συγχώνευσης χρειάζεται συνήθως να διαβάσουμε μόνο λίγους χαρακτήρες από την αρχή της κάθε ακμής φύλλου μέχρι να βρεθεί η διαφορά. Έτσι μόνο οι πρώτοι χαρακτήρες από κάθε ακμή είναι απαραίτητο να είναι φορτωμένοι στη μνήμη, ενώ οι χαρακτήρες προς το τέλος του φύλλου που έχουν λιγότερες πιθανότητες να ελεγχθούν μπορούν να φορτωθούν από το δίσκο αν παραστεί ανάγκη. Με τη μείωση των αναγκών της διαδικασίας συγχώνευσης σε μνήμη, προβλέπουμε ότι θα καταστεί εφικτό να προγραμματιστεί κατάλληλη συνάρτηση combine για την περαιτέρω μείωση των ενδιάμεσων δεδομένων.

Βιβλιογραφία

[1] M.I. Abouelhoda, S. Kurtz και E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Baeza-Yates και R. G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[3] SJ Bedathur και JR Haritsa. Engineering a fast online persistent suffix tree construction. Στο *Data Engineering, 2004. Proceedings. 20th International Conference on*, σελίδες 720–731, q.q.

[4] C.F. Cheung, J.X. Yu και H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, σελίδες 90–105, 2005.

[5] A. Cobbs. Fast approximate matching using suffix trees. Στο *Combinatorial Pattern Matching*, σελίδες 41–54. Springer, 1995.

[6] J. Dean και S. Ghemawat. Map Reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM*, 51(1):107–114, 2008.

[7] S. Ghemawat, H. Gobioff και S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.

[8] A. Ghoting και K. Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. Στο *Proceedings of the 35th SIGMOD international conference on Management of data*, σελίδες 827–840. ACM, 2009.

[9] E. Hunt, M.P. Atkinson και R.W. Irving. A database index to large biological sequences. Στο *In VLDB*, 2001.

[10] K. Kambatla, A. Pathak και H. Pucha. Towards optimizing hadoop provisioning in the cloud. Στο *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009.

[11] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[12] G. Navarro, R. Baeza-Yates, E. Sutinen και J. Tarhio. Indexing methods for approximate string matching. *Bulletin of the Technical Committee on*, σελίδα 19, 2001.

[13] Mark Nelson. Ukkonen C++ implementation. 2006.

[14] B. Phoophakdee και M.J. Zaki. Genome-scale disk-based suffix tree indexing. Στο *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, σελίδα 844. ACM, 2007.

[15] B. PHOOPHAKDEE και M.J. ZAKI. TRELLIS+: an effective approach for indexing genome-scale sequences using suffix trees. Στο *Pacific Symposium on Biocomputing 2008: Kohala Coast, Hawaii, USA, 4-8 January 2008*, τόμος 3, σελίδα 90. World Scientific Pub Co Inc, 2008.

[16] R.C.G. Holland; T. Down; M. Pocock; A. Prlić; D. Huen; K. James; S. Foisy; A. Dräger; A. Yates; M. Heuer; M.J. Schreiber. BioJava: an Open-Source Framework for Bioinformatics. 2008.

[17] S. Tata, R.A. Hankins και J.M. Patel. Practical suffix tree construction. Στο *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, σελίδα 47. VLDB Endowment, 2004.

[18] E. Ukkonen. Approximate string-matching over suffix trees. Στο *Combinatorial Pattern Matching*, σελίδες 228–242. Springer, 1993.

[19] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[20] P.J.E. Ukkonen. Two algorithms for approximate string matching in static texts. Στο *Mathematical foundations of computer science, 1991: 16th international symposium, Kazimierz Dolny, Poland, September 9-13, 1991: proceedings*, σελίδα 240. Springer Verlag, 1991.

[21] P. Weiner. Linear patter matching algorithms. *14th IEEE Symp. Switching and Automata Theory*, σελίδες 1–11, 1973.