ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ & ΔΙΚΤΥΩΝ

Τεχνικές Αναζήτησης Προτύπων σε Αρχεία Κειμένου

Διπλωματική Εργασία

του

Οδυσσέα Ε. Καλαμιώτη

Επιβλέπων: Παναγιώτης Μποζάνης, Επίκουρος Καθηγητής
Συνεπιβλέπων: Δημήτριος Κατσαρός, Συμβασιούχος ΠΔ407/80

Βόλος, Οκτώβριος 2008

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ & ΔΙΚΤΥΩΝ

Τεχνικές Αναζήτησης Προτύπων σε Αρχεία Κειμένου

Διπλωματική Εργασία

του

Οδυσσέα Ε. Καλαμιώτη

Επιβλέπων: Παναγιώτης Μποζάνης, Επίκουρος Καθηγητής
Συνεπιβλέπων: Δημήτριος Κατσαρός, Συμβασιούχος ΠΔ407/80

Βόλος, Οκτώβριος 2008

Περίληψη

Η παρούσα διπλωματική εργασία αποτελεί μία εισαγωγή στους αλγορίθμους συμβολοσειρών και πιο συγκεκριμένα στις διάφορες τεχνικές αναζήτησης προτύπων σε αρχεία κειμένου. Κατεβλήθη δε, προσπάθεια τόσο για την θεωρητική όσο και την πρακτική παρουσίαση των διαφόρων ζητημάτων που θα μας απασχολήσουν στη συνέχεια. Πιο συγκεκριμένα, υιοθετήθηκε η προσέγγιση της περιγραφής των εκάστοτε αλγορίθμων σε μία «ψευδογλώσσα» (συνήθως κοντινή προς τη C και ενίοτε την Java), ενώ δόθηκε έμφαση στην ανάλυση της πολυπλοκότητάς τους, με την κατάλληλη μαθηματική αφαίρεση και τα ανάλογα μαθηματικά εργαλεία. Παράλληλα, για κάθε αλγόριθμο που παρουσιάζεται, κατεβλήθη προσπάθεια για την πρακτική του έκθεση, χρησιμοποιώντας πολλά και κατανοητά παραδείγματα, καθώς και διάφορες επεξηγηματικές εικόνες, καθιστώντας ευκολότερη την ανάγνωση της παρούσας εργασίας.

Όσον αφορά την θεματολογία που καλύπτεται, θα πρέπει να σημειωθούν τα εξής: Η εργασία χωρίζεται σε πέντε βασικά μέρη: Εισαγωγή, Βασικές Έννοιες και Ορισμοί, Ακριβές Ταίριασμα Προτύπου, Προσεγγιστικό Ταίριασμα Προτύπου, Εφαρμογές του Προβλήματος στην Βιοπληροφορική. Συγκεκριμένα, το ***κεφάλαιο 1*** αποτελεί μία εισαγωγή στο πρόβλημα του ταιριάσματος προτύπου. Στο εισαγωγικό αυτό κεφάλαιο, ο αναγνώστης μπορεί να «εισαχθεί» εύκολα και κατανοητά στο πρόβλημα, μιας και αυτό αναλύεται επαρκώς και στις δύο μορφές του, την ακριβή και την προσεγγιστική, δηλαδή, αναζήτηση συμβολοσειρών σε αρχεία κειμένου. Παράλληλα, στην ίδια ενότητα παρουσιάζονται συνοπτικά οι διάφορες εφαρμογές του προβλήματος αυτού. Στο ***κεφάλαιο 2***, παρουσιάζονται πολλοί βασικοί και χρήσιμοι ορισμοί, που αφορούν τις συμβολοσειρές. Ο αναγνώστης, καλείται να δώσει ιδιαίτερη έμφαση σε αυτήν την ενότητα, μιας και έτσι θα καταστεί ευκολότερη και πιο κατανοητή η ανάγνωση της υπόλοιπης εργασίας. Στα επόμενα δύο κεφάλαια, αναλύονται ενδελεχώς οι δύο βασικές μορφές του υπό εξέταση προβλήματος. Στο ***κεφάλαιο 3***, παρουσιάζεται το πρόβλημα του ακριβούς ταιριάσματος συμβολοσειράς ή αλλιώς προτύπου. Συγκεκριμένα, παρουσιάζονται οι σημαντικότεροι αλγόριθμοι επίλυσης αυτού του προβλήματος, όπως η απλοϊκή λύση, ο αλγόριθμος Boyer-Moore και κάποιες παραλλαγές του, ο αλγόριθμος Knuth-Morris-Pratt καθώς και ο αριθμητικός αλγόριθμος των Karp-Rabin. Στο ***κεφάλαιο 4***, αναλύεται το προσεγγιστικό ταίριασμα προτύπων και οι αντίστοιχες τεχνικές επίλυσής του. Ιδιαίτερη έμφαση δίνεται σε μεθόδους μέτρησης της αποστάσεως μεταξύ συμβολοσειρών, όπως η ελάχιστη απόσταση διαμορφώσεως (edit distance) και η μέγιστη κοινή υποσυμβολοσειρά (longest common subsequence). Επιπλέον, παρουσιάζονται, κάποιες βασικές υποκατηγορίες του αρχικού προσεγγιστικού προβλήματος, όπως το προσεγγιστικό ταίριασμα με λάθη, προσφυματικά δένδρα ή ειδικούς χαρακτήρες (κενά). Τέλος, στο ***κεφάλαιο 5***, αναλύεται η εφαρμογή του υπό εξέταση προβλήματος στον τομέα της βιοπληροφορικής. Ιδιαίτερη, δε, έμφαση δίνεται στο πρόβλημα της στοίχισης ακολουθιών, αφού αυτό είναι το πλέον διαδεδομένο στην επιστήμη της υπολογιστικής βιολογίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συμβολοσειρές και Ανάλυση Τεχνικών Αναζήτησης Συμβολοσειρών
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: συμβολοσειρά, πρότυπο, κείμενο, ακριβές ταίριασμα, προσεγγιστικό ταίριασμα, προσφυματικά δένδρα, “don’t care” χαρακτήρες

Περιεχόμενα

Πίνακας Αλγορίθμων

Πίνακας Σχημάτων

1 Εισαγωγή

Περιεχόμενα

1.1 Ορισμός του προβλήματος

Στην πληροφορική, το ***Ταίριασμα Συμβολοσειράς*** (String Matching) ή αλλιώς ***Ταίριασμα Προτύπου*** (Pattern Matching) έγκειται στον εντοπισμό όλων των εμφανίσεων μίας δεδομένης συμβολοσειράς-λέξης-προτύπου σε μία άλλη συμβολοσειρά-κείμενο (συνήθως μεγαλύτερου μήκους). Η απλούστερη περίπτωση είναι αυτή της εύρεσης μιας δεδομένης λέξης μέσα σε μία πρόταση ή ένα κείμενο.

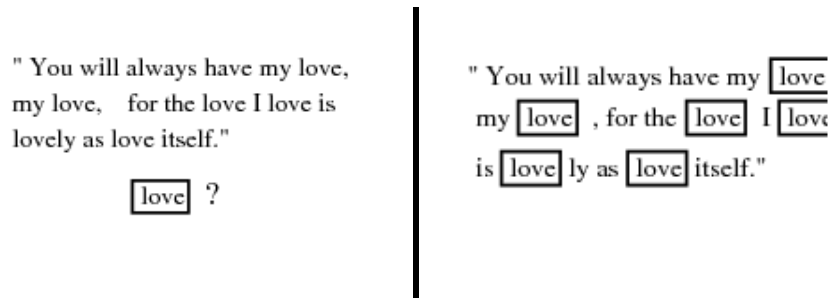
Το ταίριασμα προτύπου αποτελεί ίσως ένα από τα πιο σημαντικά προβλήματα στην αλγοριθμική που σχετίζεται με λέξεις και κείμενα. Χρησιμοποιείται δε, για την πρόσβαση και απόκτηση πληροφορίας, και χωρίς αμφιβολία, στις μέρες μας πολλοί υπολογιστές επιλύουν αυτό το πρόβλημα ως μία απλή και συνηθισμένη λειτουργία σε κάποιο σύστημα εφαρμογής. Το ταίριασμα προτύπου είναι, υπό αυτήν την οπτική, συγκρίσιμο με την ταξινόμηση ή με απλές αριθμητικές λειτουργίες.

Το πρόβλημα του ταιριάσματος προτύπου χωρίζεται σε δύο βασικές κατηγορίες: το ***Ακριβές Ταίριασμα Προτύπου*** (Exact Pattern Matching) και το ***Προσεγγιστικό Ταίριασμα Προτύπου*** (Approximate Pattern Matching).

1.1.1 Ακριβές Ταίριασμα Προτύπου

Ορισμός 1.1: *Δοθεισών μίας συμβολοσειράς-κείμενο T, μήκους n, και μίας συμβολοσειράς-πρότυπο P, μήκους m, με $m \leq n$, το πρόβλημα του ακριβούς ταιριάσματος προτύπου έγκειται στον εντοπισμό όλων των εμφανίσεων του προτύπου στο κείμενο.*

Από πρακτικής απόψεως, η σημαντικότητα του ακριβούς ταιριάσματος προτύπου θα έπρεπε να είναι εμφανής στον καθένα από εμάς που χρησιμοποιεί έναν απλό ηλεκτρονικό υπολογιστή. Ένα πολύ απλό και κατανοητό παράδειγμα, αποτελεί μία απλή αναζήτηση που εκτελούμε στο διαδίκτυο μέσω κάποιας μηχανής αναζήτησης (search engine). Η λέξη προς αναζήτηση που εισάγουμε εμείς, αποτελεί στην ουσία ένα πρότυπο για την μηχανή αναζήτησης, το οποίο προσπαθεί να εντοπίσει και να ταιριάξει στα περιεχόμενα των διαφόρων ιστοσελίδων. Επίσης, η αναζήτηση κάποιας λέξης στα διάφορα προγράμματα κειμενογράφων μέσω ειδικών εντολών (π.χ. find), αποτελεί άλλο ένα απλό και κατανοητό παράδειγμα εφαρμογής του προβλήματος του ακριβούς ταιριάσματος προτύπου.



Σχήμα 1.1: Ακριβές Ταίριασμα Προτύπου

Το ακριβές ταίριασμα προτύπου αποτελεί ένα από τα παλαιότερα και πιο κυρίαρχα προβλήματα στην επιστήμη της πληροφορικής. Εφαρμογές που απαιτούν κάποια μορφή ταιριάσματος συμβολοσειράς μπορούν να βρεθούν ουσιαστικά παντού. Εντούτοις, τα τελευταία χρόνια έχει παρατηρηθεί μια δραματική αύξηση του ενδιαφέροντος σε τέτοιου είδους προβλήματα, όπως για παράδειγμα στις εφαρμογές της ανάκτησης πληροφοριών και της υπολογιστικής βιολογίας.

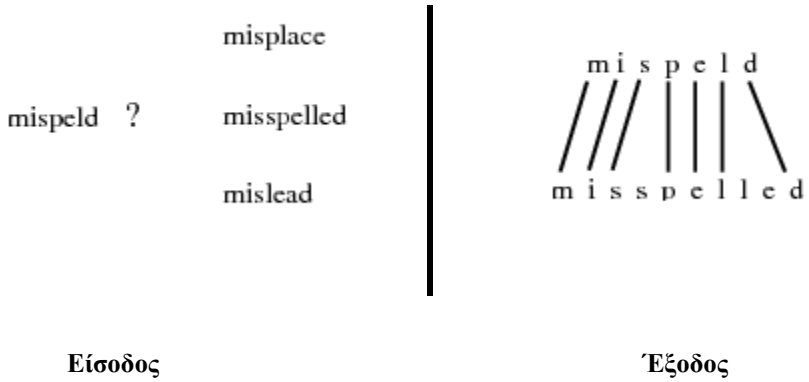
Εφαρμογές όπως αυτές, δεν αντιμετωπίζουν απλά μια δραστική αύξηση στα μεγέθη κειμένων που πρέπει να διαχειριστούν, αλλά, συγχρόνως, απαιτούν να εκτελέσουν όλο και περισσότερο περίπλοκες αναζητήσεις. Τα πιθανά πρότυπα προς αναζήτηση δεν περιορίζονται μόνο σε απλές συμβολοσειρές αποτελούμενες από χαρακτήρες ή αριθμούς, αλλά ενδέχεται να δίνονται σε διαφορετικές μορφές, όπως για παράδειγμα ακολουθίες, δένδρα, κανονικές εκφράσεις, κ.α.

Το ταίριασμα προτύπου κυμαίνεται σε ένα ευρύ φάσμα εφαρμογών, τόσο θεωρητικών όσο και πρακτικών. Οι θεωρητικές λύσεις του προβλήματος αυτού έχουν αποτελέσει την αφορμή και το κίνητρο σε σημαντικά αλγοριθμικά επιτεύγματα, εντούτοις είναι σπάνια χρήσιμες στην πράξη.

Στις μέρες μας υπάρχουν πολλοί αλγόριθμοι που επιλύουν αποδοτικά αυτό το πρόβλημα. Οι δύο πιο βασικοί αλγόριθμοι ακριβούς ταιριάσματος προτύπου (exact pattern matching) είναι ο αλγόριθμος των ***Knuth-Morris-Pratt*** και ο αλγόριθμος των ***Boyer-Moore***.

1.1.2 Προσεγγιστικό Ταίριασμα Προτύπου

Ο ορισμός ενός ταιριάσματος μπορεί επίσης να επιτρέπει μικρές διαφορές μεταξύ του προτύπου και της εμφάνισης αυτού στο κείμενο. Αυτό καλείται "κατά προσέγγιση ταίριασμα" ή αλλιώς "προσεγγιστικό ταίριασμα" και παρουσιάζει ιδιαίτερο ενδιαφέρον σε εφαρμογές όπως η ανάκτηση κειμένων και η υπολογιστική βιολογία.



Σχήμα 1.2: Προσεγγιστικό Ταίριασμα Προτύπου

Το προσεγγιστικό, λοιπόν, ταίριασμα είναι το πρόβλημα εντοπισμού όλων των εμφανίσεων ενός προτύπου σε ένα κείμενο, όπου το πρότυπο και η αντίστοιχη εμφάνισή του στο κείμενο μπορούν να έχουν έναν περιορισμένο αριθμό διαφορών. Μοντελοποιείται δε, συνήθως, χρησιμοποιώντας κάποια συνάρτηση υπολογισμού της απόστασης μεταξύ συμβολοσειρών, η οποία μας επιτρέπει να προσδιορίζουμε την ομοιότητα αυτών των συμβολοσειρών (κατά πόσο δηλαδή αυτές οι συμβολοσειρές μοιάζουν μεταξύ τους). Δεδομένου ενός προτύπου προς αναζήτηση, ενός κειμένου στο οποίο θα αναζητήσουμε το πρότυπο κι ενός κατώτατου ορίου k, το οποίο αποτελεί την μέγιστη επιτρεπόμενη απόσταση μεταξύ του προτύπου και των εμφανίσεων του στο κείμενο, επιθυμούμε να βρούμε όλες τις εμφανίσεις του προτύπου αυτού στο κείμενο. Σε αυτήν την εργασία επικεντρωνόμαστε περισσότερο στην ***απόσταση Levenshtein*** ή αλλιώς ***απόσταση διαμορφώσεως (edit distance)***, η οποία είναι ο ελάχιστος αριθμός εισαγωγών, διαγραφών και αντικαταστάσεων χαρακτήρων που απαιτούνται προκειμένου μία συμβολοσειρά να μετατραπεί σε μία άλλη. Πολλές εφαρμογές χρησιμοποιούν παραλλαγές αυτής της απόστασης.

Ορισμός 1.2: *Δοθεισών μίας συμβολοσειράς - πρότυπο P, μήκους m, μίας συμβολοσειράς - κείμενο T, μήκους n, κι ενός θετικού ακεραίου k, με $0 \leq k < m$, το πρόβλημα του προσεγγιστικού ταιριάσματος προτύπου έγκειται στον εντοπισμό όλων των εμφανίσεων του P στο T, έτσι ώστε το πολύ σε k θέσεις το κείμενο και το πρότυπο να έχουν διαφορετικούς χαρακτήρες.*

Εναλλακτικά, μπορούμε δώσουμε τον ακόλουθο ορισμό:

Ορισμός 1.3: *Δοθεισών μίας συμβολοσειράς x, μήκους n, και μίας συμβολοσειράς y, μήκους m, με $m \leq n$, το πρόβλημα του προσεγγιστικού ταιριάσματος προτύπου έγκειται στον υπολογισμό της ομοιότητας των x και y.*

Το προσεγγιστικό ταίριασμα προτύπου παίζει καθοριστικό ρόλο σε εφαρμογές επεξεργασίας βάσεων δεδομένων και κειμένων, μιας και δε πρέπει να ξεχνάμε ότι ζούμε σε έναν επιρρεπή σε λάθη κόσμο. Για παράδειγμα, κάθε επεξεργαστής κειμένου πρέπει να περιέχει έναν μηχανισμό προκειμένου να ψάχνει το τρέχον έγγραφο/κείμενο για αυθαίρετες συμβολοσειρές. Επιπλέον, οι ορθογράφοι ελέγχουν ένα κείμενο για λέξεις που είναι καταχωρημένες στο λεξικό ενώ απορρίπτουν οποιεσδήποτε άλλες.

Οι αλγόριθμοι αναζήτησης συμβολοσειρών είναι πολύ σημαντικοί σε πολλά είδη εφαρμογών τα οποία συναντάμε καθημερινά στη ζωή μας. Στους επεξεργαστές κειμένων, ίσως επιθυμούμε να αναζητήσουμε σε κάποιο πολύ μεγάλο έγγραφο (αποτελούμενου από εκατομμύρια χαρακτήρες) μία δεδομένη συμβολοσειρά ή λέξη (ίσως δεκάδων στο πλήθος χαρακτήρων). Από την άλλη, στα εργαλεία ανάκτησης κειμένων (text retrieval tools), ενδεχομένως να θέλουμε να ψάξουμε μέσω χιλιάδων εγγράφων (τα οποία υπό κανονικές συνθήκες θα έπρεπε να ταξινομηθούν κατάλληλα, καθιστώντας το αυτό μία περιττή διαδικασία). Άλλες εφαρμογές πάλι απαιτούν αλγορίθμους προσεγγιστικού ταιριάσματος συμβολοσειρών/προτύπων ως μέρος ενός πιο σύνθετου αλγορίθμου (π.χ., η εντολή του «diff» του συστήματος Unix που επιστρέφει τις διαφορές μεταξύ δύο όμοιων αρχείων κειμένων).

Επιπλέον, η επιστήμη της βιολογίας (Βιοπληροφορική - Bioinformatics) αποφαίνεται πως ομοιότητα στις ακολουθίες - συμβολοσειρές DNA, RNA κτλ, συνεπάγεται και

δομική ή λειτουργική ομοιότητα, καθώς στην φύση, συχνά, παρατηρείται πλεονασμός. Η γνωστή βάση δεδομένων Genbank έχει αποτελέσει ένα θεμελιώδες εργαλείο για τη μοριακή βιολογία, παρέχοντας τη δυνατότητα αναζητήσεων ομολογιών (ομοιοτήτων) σε γονιδιωματικές ακολουθίες DNA.

Κάποιοι από τους διαθέσιμους αλγόριθμους επίλυσης του προσεγγιστικού ταιριάσματος στηρίζονται στην τεχνική του δυναμικού προγραμματισμού. Αυτή η τεχνική, αν και αποτελεί την παλαιότερη προσέγγιση του προβλήματος, εντούτοις, δεν συγκαταλέγει τους αλγορίθμους που την χρησιμοποιούν στους πλέον αποδοτικότερους. Παρόλα αυτά, εμείς στη συνέχεια θα παρουσιάσουμε και αναλύσουμε τέτοιου είδους αλγορίθμους.

1.2 Εφαρμογές του Προβλήματος

Οι πρώτες αναφορές στο πρόβλημα της αναζήτησης προτύπων σε κείμενα, έγιναν στη δεκαετία του ’60 και του ’70, οπότε το ταίριασμα προτύπου εμφανίστηκε σ’ ένα πλήθος διαφορετικών πεδίων εφαρμογής. Βασικά κίνητρα στην ανάπτυξη αυτού του είδους αναζήτησης, αποτέλεσαν η **υπολογιστική βιολογία** (Computational Biology), η **επεξεργασία σημάτων** (Signal Processing) καθώς και η **ανάκτηση κειμένων** (Text Retrieval).

Ωστόσο, ο αριθμός των εφαρμογών του ταιριάσματος προτύπου αυξάνεται συνεχώς. Τα τελευταία χρόνια, έχουν βρεθεί λύσεις σε πολλά και διαφορετικά προβλήματα τα οποία βασίζονται στο ταίριασμα συμβολοσειράς. Ενδεικτικά αναφέρουμε τα εξής:

- συμπίεση εικόνας (image compression)
- εξόρυξη δεδομένων (data mining)
- ανίχνευση ιών (virus detection)
- αναγνώριση οπτικών χαρακτήρων (optical character recognition)
- σύγκριση αρχείων (file comparison)
- αναγνώριση χειρόγραφων (hand-writing recognition).

Βιβλιογραφία Κεφαλαίου

[1] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[2] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[3] Π.Δ. Μποζάνης, *Αλγόριθμοι: Σχεδιασμός και Ανάλυση*, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002.

[5] Paolo Ferragina, *String Algorithms and Data Structures*, University of Pisa, Italy.

2 Βασικές Έννοιες και Ορισμοί

Περιεχόμενα

2.1 Συμβολοσειρές

Συμβολοσειρά (string), στην επιστήμη των υπολογιστών, καλείται μια ακολουθία/σειρά από διαδοχικά σύμβολα τα οποία είναι στοιχεία ενός πεπερασμένου συνόλου ή αλλιώς ***αλφάβητου (alphabet) Σ***. Ανάλογα με τα στοιχεία που διαθέτει το αλφάβητο Σ, διαφοροποιείται και το περιεχόμενο της συμβολοσειράς. Τα στοιχεία αυτά μπορεί να είναι γράμματα (letters), χαρακτήρες (characters) ή σύμβολα (symbols).

Στην περίπτωση που το αλφάβητο περιορίζεται σε γράμματα και αριθμούς, η συμβολοσειρά λέγεται και ***αλφαριθμητικό***. Οποιοδήποτε λογοτεχνικό κείμενο μπορεί να θεωρηθεί ως συμβολοσειρά, παραγόμενη βάσει του αλφαβήτου

$$\Sigma = \{\alpha, \ldots, \omega, A, \ldots, \Omega, 0, \ldots, 9, . , - , / , \cdot , : , \textbf{,} , ! , « , » , ; , (,)\}$$

ενώ, στις εφαρμογές της βιοχημείας, οι ακολουθίες DNA αναπαρίστανται με συμβολοσειρές από το αλφάβητο των νουκλεοτιδίων

$$\Sigma = \{A, C, G, T\},$$

όπου το ‘A’ συμβολίζει την αδενίνη, το ‘C’ την κυτοσίνη, το ‘G’ την γουανίνη και το ‘T’ την θυμίνη.

Εάν Σ είναι το αλφάβητο, τότε το σύνολο όλων των πεπερασμένου μήκους συμβολοσειρών ορίζεται ως Σ^*. Σε αυτό το σύνολο ανήκει και η κενή συμβολοσειρά, η οποία απεικονίζεται με το ε.

Το ***μήκος*** μιας συμβολοσειράς x συμβολίζεται με |x|. Για παράδειγμα, το μήκος της συμβολοσειράς x = “abcde” ισούται με |x| = 5. Προφανώς, η κενή συμβολοσειρά έχει μήκος |ε| = 0.

Μία συμβολοσειρά x, μήκους n = |x|, αναπαρίσταται, συνήθως, είτε με έναν πίνακα x[1…n] n θέσεων, είτε ως μία ακολουθία $x = x_1x_2\ldots x_n$ n χαρακτήρων.

Έστω, τώρα, μια συμβολοσειρά x και μια συμβολοσειρά y ενός αλφαβήτου Σ. Η ***συνένωση (concatenation)*** της x με τη y συμβολίζεται με w = xy και αποτελεί μια νέα συμβολοσειρά η οποία προκύπτει από τους χαρακτήρες της x ακολουθούμενους από εκείνους της y, συνολικού μήκους |w| = |x| + |y|.

Μια συμβολοσειρά x καλείται ***πρόθεμα (prefix)*** μίας άλλης συμβολοσειράς y, εάν υπάρχει συμβολοσειρά w η οποία να ανήκει στο Σ^*, τέτοια ώστε y = xw.

Μια συμβολοσειρά x καλείται ***πρόσφυμα ή επίθεμα (suffix)*** μίας άλλης συμβολοσειράς y, εάν υπάρχει συμβολοσειρά w η οποία να ανήκει στο Σ^*, τέτοια ώστε y = wx.

Όταν $w \neq \varepsilon$, το πρόθεμα ή το πρόσφυμα χαρακτηρίζεται ως ***γνήσιο (proper)***.

Μια συμβολοσειρά w καλείται ***υποσυμβολοσειρά (substring)*** ή ***παράγοντας (factor)*** μίας άλλης συμβολοσειράς x, εάν αυτή αποτελείται από **συνεχόμενους** χαρακτήρες

της x, έτσι ώστε w = uxv, όπου u, v δύο οποιεσδήποτε συμβολοσειρές (πιθανώς, και οι κενές συμβολοσειρές).

Σε αντίθεση με τον προηγούμενο ορισμό, ως ***υπακολουθία (subsequence)*** w μίας συμβολοσειράς x, καλείται κάθε υποσυμβολοσειρά (substring) της x, η οποία αποτελείται, **όχι απαραίτητα,** από συνεχόμενους χαρακτήρες της x. Πιο συγκεκριμένα, εάν $x = x_1x_2\ldots x_n$ είναι μία συμβολοσειρά, τότε κάθε συμβολοσειρά της μορφής

$$w = w_{i1}w_{i2}\ldots w_{ik} \qquad \text{με} \qquad i_1 < i_2 < \ldots < i_k \leq n$$

καλείται υπακολουθία (subsequence), με μήκος k, της x. Χρησιμοποιώντας έναν εναλλακτικό ορισμό, μπορούμε να πούμε ότι $w_0w_1\ldots w_{i-1}$ είναι μία υπακολουθία της $x = x_0x_1\ldots x_{n-1}$ εάν υπάρχει μία **"αυστηρά"** αύξουσα ακολουθία από ακεραίους της μορφής $k_0k_1\ldots k_{i-1}$, τέτοια ώστε για $0 \leq k \leq i-1$, $w_j = x_{kj}$.

Ένας ακέραιος p καλείται ***περίοδος (period)*** μιας συμβολοσειράς x μήκους n, εάν ισχύει ότι $w[i] = w[i+p]$, για $0 \leq i < n-p$. Η μικρότερη περίοδος της x συμβολίζεται με per(x).

Μία συμβολοσειρά x, μήκους k, καλείται ***περιοδική (periodic)***, εάν το μήκος της μικρότερης περιόδου της είναι μικρότερο ή ίσο με k/2, διαφορετικά καλείται ***μη-περιοδική (non-periodic)***.

Μία συμβολοσειρά z καλείται ***σύνορο (border)*** μίας συμβολοσειράς x, εάν υπάρχουν δύο άλλες συμβολοσειρές u και v, έτσι ώστε x = uz = zv. Με άλλα λόγια, η συμβολοσειρά z αποτελεί συγχρόνως και πρόθεμα (prefix) και πρόσφυμα (suffix) της x. Αξίζει, επίσης, να σημειωθεί ότι στην περίπτωση αυτή, η τιμή |u| = |v| αποτελεί μία περίοδο της συμβολοσειράς x.

Η ***αντίστροφη (reverse)*** συμβολοσειρά, $\mathbf{x^R}$, μίας συμβολοσειράς x μήκους k, αποτελείται από τους χαρακτήρες της x με αντίστροφη, όμως, σειρά και ισούται με $x^R = x[k-1]x[k-2]\ldots x[1]x[0]$.

2.2 Ακριβές Ταίριασμα Συμβολοσειράς

Το πρόβλημα του ***Ακριβούς Ταιριάσματος Συμβολοσειράς (Exact String Matching)*** ορίζεται ως εξής:

Ορισμός 2.1: *Δοθεισών μίας συμβολοσειράς x, μήκους n, και μίας συμβολοσειράς y, μήκους m, με m ≤ n, αιτείται ο εντοπισμός όλων των εμφανίσεων της x μέσα στην y.*

Η συμβολοσειρά x, μήκους n, καλείται ***κείμενο (text)*** και συμβολίζεται ως $T = t_1t_2\ldots t_n$, ενώ η συμβολοσειρά y, μήκους m, καλείται ***πρότυπο (pattern)*** και συμβολίζεται με $P = p_1p_2\ldots p_m$. Για το λόγο αυτό, το παραπάνω πρόβλημα συναντάται συνήθως και ως ***Ακριβές Ταίριασμα Προτύπου (Exact Pattern Matching)***.

2.3 Προσεγγιστικό Ταίριασμα Συμβολοσειράς

Το πρόβλημα του ***Προσεγγιστικού Ταιριάσματος Συμβολοσειράς (Approximate String Matching)*** ορίζεται ως εξής:

Ορισμός 2.2: *Δοθεισών μίας συμβολοσειράς x, μήκους n, και μίας συμβολοσειράς y, μήκους m, με $m \leq n$, αιτείται ο υπολογισμός της ομοιότητας των x και y.*

Η συμβολοσειρά x, μήκους n, καλείται ***κείμενο (text)*** και συμβολίζεται ως $T = t_1t_2\ldots t_n$, ενώ η συμβολοσειρά y, μήκους m, καλείται ***πρότυπο (pattern)*** και συμβολίζεται με $P = p_1p_2\ldots p_m$. Για το λόγο αυτό, το παραπάνω πρόβλημα συναντάται συνήθως και ως ***Προσεγγιστικό Ταίριασμα Προτύπου (Approximate Pattern Matching)***.

Ένας εναλλακτικός ορισμός είναι ο ακόλουθος:

Ορισμός 2.3: *Δοθεισών μίας συμβολοσειράς - πρότυπο P, μήκους m, μίας συμβολοσειράς - κείμενο T, μήκους n, κι ενός θετικού ακεραίου k, με $0 \leq k < m$, το πρόβλημα του προσεγγιστικού ταιριάσματος προτύπου έγκειται στον εντοπισμό όλων των εμφανίσεων του P στο T, έτσι ώστε το πολύ σε k θέσεις το κείμενο και το πρότυπο να έχουν διαφορετικούς χαρακτήρες.*

Βιβλιογραφία Κεφαλαίου

[1] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[2] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[3] Π.Δ. Μποζάνης, *Αλγόριθμοι: Σχεδιασμός και Ανάλυση*, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002.

[5] Ricardo A. Baeza-Yates and Chris H. Perleberg, *Fast and Practical Approximate String Matching*

3 Ακριβές Ταίριασμα Προτύπου

Περιεχόμενα

3.1 Εισαγωγή

Το πρόβλημα του **Ακριβούς Ταιριάσματος Συμβολοσειράς (Exact String Matching)** ορίζεται ως εξής:

Ορισμός 2.1: *Δοθεισών μίας συμβολοσειράς x, μήκους n, και μίας συμβολοσειράς y, μήκους m, με m ≤ n, αιτείται ο εντοπισμός όλων των εμφανίσεων της x μέσα στην y.*

Η συμβολοσειρά x, μήκους n, καλείται **κείμενο (text)** και συμβολίζεται ως $T = t_1t_2\ldots t_n$, ενώ η συμβολοσειρά y, μήκους m, καλείται **πρότυπο (pattern)** και συμβολίζεται με $P = p_1p_2\ldots p_m$. Για το λόγο αυτό, το παραπάνω πρόβλημα συναντάται συνήθως και ως **Ακριβές Ταίριασμα Προτύπου (Exact Pattern Matching)**.

Αυτό το απλό, στην έκφραση, πρόβλημα βρίσκει εφαρμογές, λ.χ., στα προγράμματα κειμενογράφων, την αναζήτηση μίας συγκεκριμένης υποδομής σε μία ακολουθία DNA ή την αναζήτηση λέξεων στα περιεχόμενα μιας ιστοσελίδας.

3.2 Ο Απλοϊκός Αλγόριθμος

3.2.1 Περιγραφή

Στην απλούστερη λύση **(brute force ή naive algorithm),** το πρότυπο P αντιπαραβάλλεται με το κείμενο T, αρχίζοντας από όλες τις δυνατές θέσεις 1, ..., n-m+1 του T. Πιο συγκεκριμένα, το αριστερό άκρο του προτύπου P ευθυγραμμίζεται με το αριστερό άκρο του κειμένου T και οι χαρακτήρες του κειμένου, συγκρίνονται ένας προς ένας, από τα αριστερά προς τα δεξιά, με τους αντίστοιχους του προτύπου[1]. Η διαδικασία αυτή συνεχίζεται έως ότου, είτε εντοπιστούν δύο διαφορετικοί χαρακτήρες,είτε εξαντληθεί το πρότυπο P, οπότε και έχουμε το πρώτο ταίριασμα. Στην πρώτη περίπτωση, όπου οι χαρακτήρες διαφέρουν, το P ολισθαίνει μία θέση προς τα δεξιά και η διαδικασία της σύγκρισης των χαρακτήρων επαναλαμβάνεται απο το αριστερό άκρο-πρώτος χαρακτήρας-του προτύπου. Ουσιαστικά μπορούμε να φανταστούμε το πρότυπο P σαν ένα παράθυρο, μήκους m, το οποίο ολισθαίνει κατά μήκος του κειμένου T. Ο αλγόριθμος αυτός τερματίζει όταν το δεξί άκρο του P «προσπεράσει» το δεξί άκρο του κειμένου T. Μία ενδεικτική υλοποίηση του απλοικού αυτού αλγορίθμου φαίνεται παρακάτω (αλγ. 3.1):

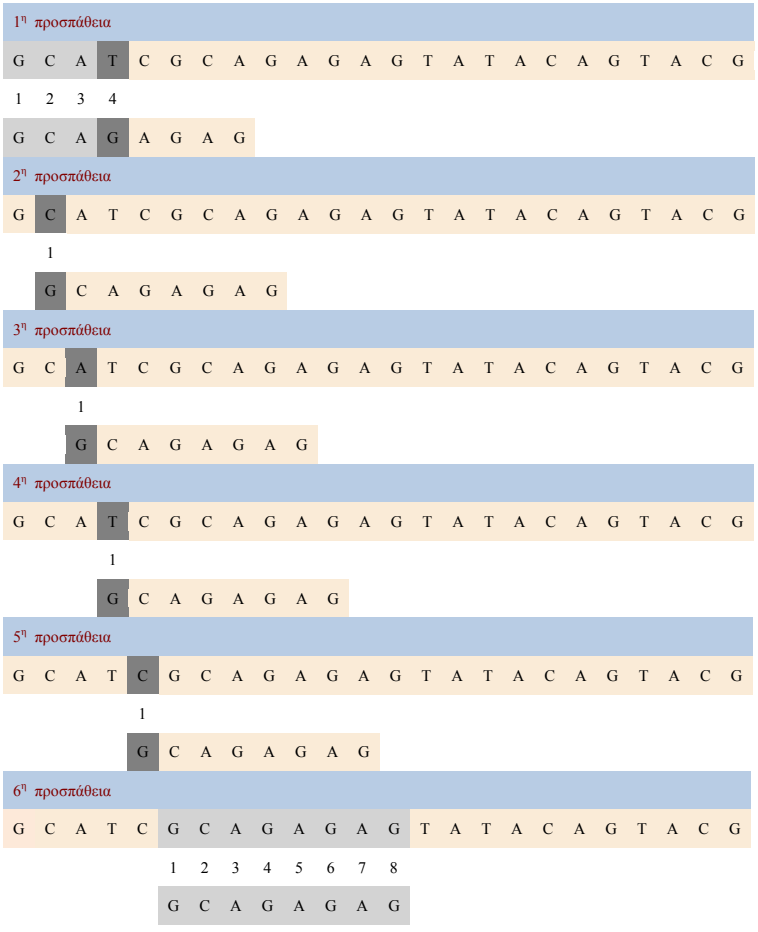
```
Αλγόριθμος: bruteForce(char[]  T, char[] P)
Είσοδος: Ένα κείμενο T
Έξοδος: Εντοπισμός του P στο T

1.      n=T.length; m=P.length;
2.      for (i=1; i=n-m+1; i++){       // i η τρέχουσα θέση του παραθύρου
3.        j=1;                         // δείκτης θέσεως εντός παραθύρου-προτύπου
4.        while ((j≤m) && (T[i+j-1] == P[j]))
5.              j++;
```

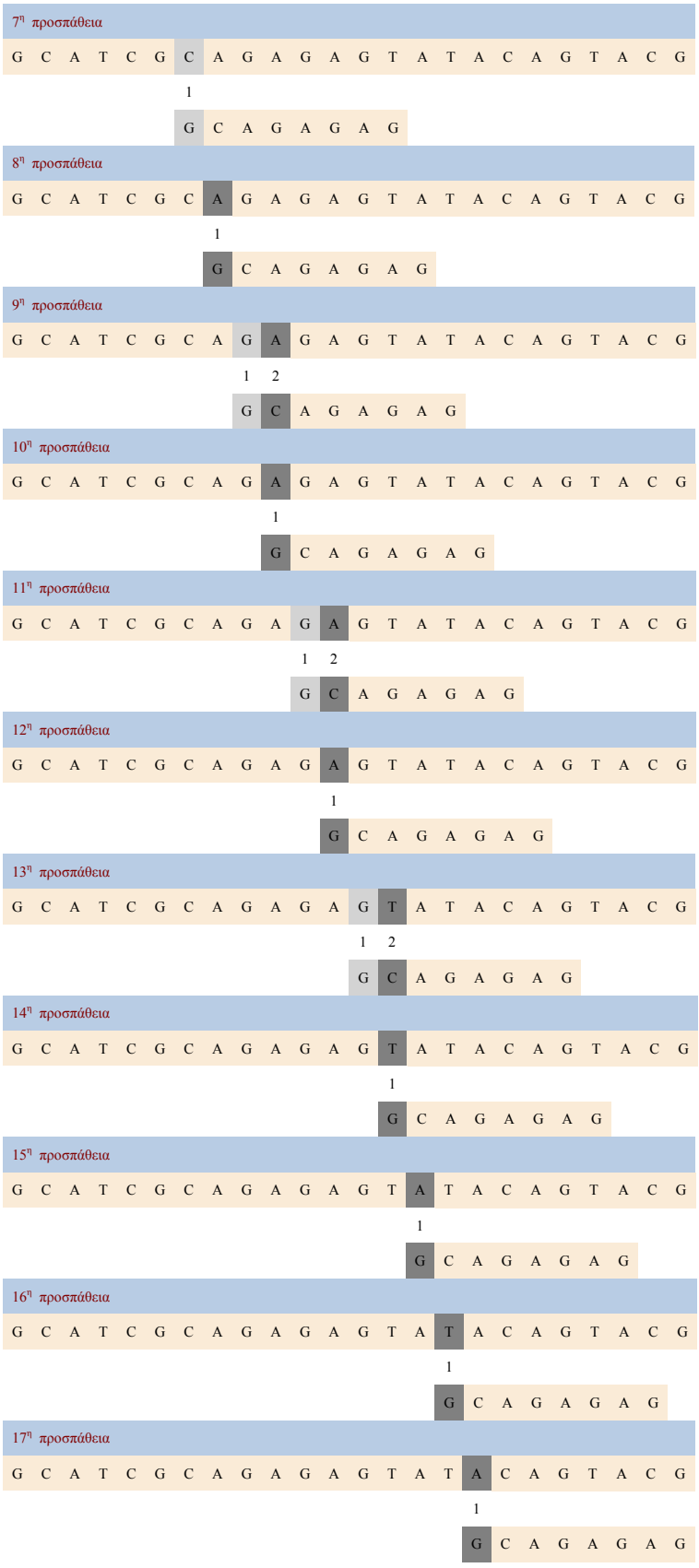
[1] Υπάρχουν διάφορες υλοποιήσεις του αλγορίθμου ανάλογα με το πώς εκτελείται η σύγκριση - από το αριστερό άκρο του προτύπου P προς το δεξιό άκρο του P ή το αντίστροφο.

```
6.        if (j == m+1)
7.              return i;              // βρέθηκε η αρχική θέση του προτύπου
8.      }
9.      return -1;                     // δεν υπάρχει το πρότυπο
```

Αλγόριθμος 3.1: Απλοϊκή λύση ταιριάσματος προτύπου

Ως παράδειγμα, ας θεωρήσουμε ως κείμενο T, τη συμβολοσειρά x = GCATCGCAGAGAGTATACAGTACG, με μήκος n = 24, και ως πρότυπο P, τη συμβολοσειρά p = GCAGAGAG, με μήκος m = 8. Στην πρώτη προσπάθεια, μετά από 3 επιτυχημένες συγκρίσεις διαπιστώνουμε ότι ο τέταρτος χαρακτήρας του προτύπου "g" διαφέρει από τον τέταρτο χαρακτήρα του κειμένου T , "t", οπότε , σύμφωνα με τον αλγόριθμο 3.1, το παράθυρο - πρότυπο P - ολισθαίνει κατά μία θέση δεξιότερα. Κατόπιν τούτου, η διαδικασία επανεκκινείται από την δεύτερη θέση του κειμένου, όπου παρατηρούμε ότι δεν υπάρχει κανένα ταίριασμα. Το ίδιο συμβαίμει και στις επόμενες τρεις προσπάθειες, έχοντας βέβαια ολισθήσει κάθε φορά κατά μία θέση δεξιότερα το πρότυπο. Στην έκτη προσπάθεια, εντοπίζουμε την πρώτη εμφανιση του προτύπου P μέσα στο κείμενο T, ενώ παράλληλα ολισθαίνουμε το P μία θέση δεξιά. Η διαδικασία σύγκρισης των χαρακτήρων συνεχίζεται με τον ίδιο τρόπο έως ότου το δεξί άκρο του προτύπου P ευθυγραμμιστεί με το δεξί άκρο του κειμένου T, οπότε και τερματίζει ο αλγόριθμος. Στο σχήμα 3.1 φαίνεται ένα στιγμιότυπο αυτής της λύσεως.





Σχήμα 3.1: Στιγμιότυπο απλοϊκής λύσης ταιριάσματος προτύπου

3.2.2 Ανάλυση Πολυπλοκότητας

Βασικό μειονέκτημα της παραπάνω απλοϊκής λύσης αποτελεί το γεγονός ότι «ξεχνά» οποιοδήποτε ταίριασμα χαρακτήρων έχει προηγηθεί. Επίσης ο αλγόριθμος, κάθε φορά ολισθαίνει το πρότυπο P κατά **μία μόνο** θέση δεξιότερα, ξεκινώντας την σύγκριση από τον πρώτο χαρακτήρα του P. Αυτό, όπως γίνεται κατανοητό, οδηγεί σε πολλές και «άσκοπες» συγκρίσεις ενώ παράλληλα καθιστά τον αλγόριθμο αυτό πάρα πολύ αργό για πολύ μεγάλα κείμενα και πρότυπα. Η πολυπλοκότητα της λύσης, στην χειρότερη περίπτωση, είναι O(*nm*), καθώς υπάρχει το ενδεχόμενο, κάθε φορά να γίνονται m-1 επιτυχημένα ταιριάσματα και ένα ανεπιτυχές. Στην περίπτωση αυτή οδηγούμαστε σε n-m-1 ολισθήσεις, κόστους O(*m*) έκαστη, ενώ ο αναμενόμενος αριθμός συγκρίσεων χαρακτήρων του κειμένου είναι 2n. Χαρακτηριστικό παράδειγμα το οποίο οδηγεί τον αλγόριθμο σε συμπεριφορά χειρότερης περιπτώσεως αποτελεί η αναζήτηση του προτύπου $P=a^{m-1}b$ στο κείμενο $T=a^n$.

3.3 Ο Αλγόριθμος Knuth-Morris-Pratt (KMP)

Όπως διαπιστώσαμε και παραπάνω, η απλοϊκή λύση αγνοεί ή «ξεχνά» κάθε πληροφορία από προηγούμενα ταιριασμένα σύμβολα. Αυτό έχει ως συνέπεια ο αλγόριθμος να συγκρίνει τον ίδιο χαρακτήρα κειμένου με διαφορετικούς χαρακτήρες του προτύπου ξανά και ξανά, με αποτέλεσμα να οδηγούμαστε στην πολυπλοκότητα χειρότερης περιπτώσεως, O(*nm*).

Οι Donald Knuth, Vaughan Pratt και J.H.Morris δημοσίευσαν το 1977 την δική τους προσέγγιση για το πρόβλημα του ακριβούς ταιριάσματος συμβολοσειράς. Στην πραγματικότητα, ο εν λόγω αλγόριθμος ανακαλύφθηκε ανεξάρτητα από τους Knuth, Pratt και τον Morris και, στη συνέχεια, οι τρεις ερευνητές συνεργάστηκαν και το δημοσίευσαν, σε κοινό επιστημονικό άρθρο.

Ο αλγόριθμος των Knuth-Morris-Pratt (KMP), σε αντίθεση με την απλοϊκή λύση, κάνει χρήση της πληροφορίας που λαμβάνεται από προηγούμενες συγκρίσεις χαρακτήρων και έτσι ποτέ δεν ξανασυγκρίνει ένα σύμβολο του κειμένου το οποίο έχει ήδη ταιριάξει επιτυχώς με ένα σύμβολο του προτύπου. Με τον τρόπο αυτό, το παράθυρο/πρότυπο ολισθαίνει όσο το δυνατόν περισσότερες θέσεις δεξιότερα αντί της μίας που ολισθαίνει στον απλοϊκό αλγόριθμο. Αυτό αποτελεί και τη βασική ιδέα στην οποία στηρίζεται ο KMP αλγόριθμος.

Η μέθοδος χωρίζεται σε δύο φάσεις, την **φάση προεπεξεργασίας του προτύπου (pattern-preprocessing phase)** και την **φάση αναζήτησης (searching phase).** Η πρώτη φάση περιλαμβάνει την εξαγωγή ενός **πίνακα/συνάρτησης αποτυχίας (failure table or function) f**, μήκους m. Ο πίνακας καλείται έτσι, αφού σε κάθε αποτυχία σύγκρισης των χαρακτήρων του προτύπου με τους χαρακτήρες του κειμένου, ανατρέχουμε σε αυτόν προκειμένου να αποφανθούμε για τον αριθμό των θέσεων που πρέπει να ολισθήσουμε δεξίοτερα το παράθυρο. Στην φάση αναζήτησης, όπως γίνεται εύκολα κατανοητό, κύριος σκοπός μας είναι ο εντοπισμός όλων των εμφανίσεων του προτύπου P στο κείμενο T, με τη βοήθεια πάντα της συνάρτησης αποτυχίας. Παρακάτω θα αναλύσουμε κάθε φάση χωριστά.

3.3.1 Φάση Προεπεξεργασίας Προτύπου

Όπως αναφέρθηκε και προηγουμένως, ο αλγόριθμος KMP, πριν από την φάση αναζήτησης, επεξεργάζεται το πρότυπο P υπολογίζοντας έναν πίνακα αποτυχίας f, ο οποίος υποδεικνύει τη μέγιστη δυνατή ολίσθηση του παραθύρου σε περίπτωση ανεπιτυχούς ταιριάσματος. Ο πίνακας αυτός, μεγέθους m, περιέχει σε κάθε θέση του j, το μήκος του μέγιστου μη-τετριμμένου επιθέματος x της υποσυμβολοσειράς

$$P_j = P_1 P_2 \ldots P_j$$

του προτύπου, το οποίο αποτελεί συγχρόνως και πρόθεμά της. Με άλλα λόγια, η j-οστή θέση του πίνακα f, δηλώνει το μήκος του μέγιστου μη-τετριμμένου προσφύματος x της συμβολοσειράς P_j. Δηλαδή, ισχύει

$$P_j = xy = zx \quad \text{με} \quad x \text{ μέγιστο.}$$

Παρακάτω, παρατίθεται μία ενδεικτική υλοποίηση του αλγορίθμου υπολογισμού της συναρτήσεως αποτυχίας f (αλγ. 3.2). Αξιοσημείωτο είναι το γεγονός ότι ο αλγόριθμος αυτός αιτεί γραμμική *O(m)*, χρονική και χωρική προεπεξεργασία του προτύπου P, για την εξαγωγή του πίνακα f.

```
Αλγόριθμος: failureKMP(char[]  P, int[] f)
Είσοδος: Ένα κείμενο P
Έξοδος: Υπολογισμός πίνακα/συνάρτησης f

1.      int m = P.length; k = 0;
2.      f[1]=0;
3.      for (j=2; j≤m; j++){
4.        while ((k>0) && (P[k+1] != P[j]))     //εφ'όσων P[f[k]+1] ≠ P[j]
5.            k = f[k];
6.        if (P[k+1] == P[j])
7.            k++;
8.        f[j] = k;
9.      }
10.     return f;
```

Αλγόριθμος 3.2: Αλγόριθμος υπολογισμού συνάρτησης αποτυχίας

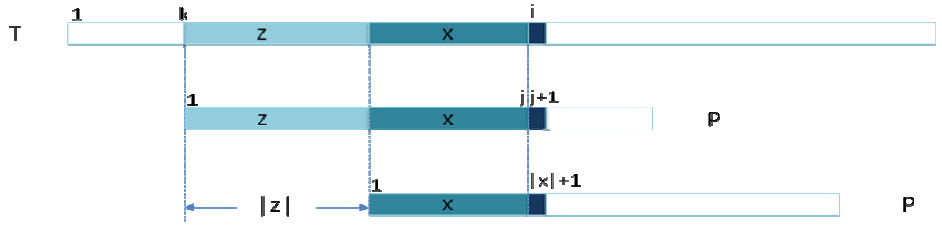
Ως παράδειγμα, ας θεωρήσουμε ως πρότυπο την συμβολοσειρά P = ababaa. Σύμφωνα με τον αλγόριθμο 3.2, οι τιμές του πίνακα αποτυχίας που προκύπτουν, φαίνονται στο σχήμα 3.2. Για j=5, παρατηρούμε ότι το μέγιστο πρόσφυμα της συμβολοσειράς P_5=ababa, είναι η υποσυμβολοσειρά "aba", με μήκος 3. Οπότε, έχουμε f[5]=3.

j	1	2	3	4	5	6
P[j]	a	b	a	b	a	a
f[j]	0	0	1	2	3	1

Σχήμα 3.2: Υπολογισμός συνάρτησης αποτυχίας f της συμβολοσειράς ababaa

3.3.2 Φάση Αναζήτησης

Έστω, τώρα, ότι κατά την επεξεργασία του τρέχοντος παραθύρου (σχήμα 3.3), έχουν ταιριαστεί επιτυχώς οι πρώτοι j χαρακτήρες του προτύπου, P[0..j] = T[k..i-1], αλλά κατά την σύγκριση του T_i με τον p_{j+1} εντοπίζουμε την πρώτη διαφορά, $b = T_i \neq P_{j+1} = a$. Στην περίπτωση αυτή, όπως φαίνεται και στο σχήμα, η επόμενη σύγκριση του χαρακτήρα T_i μπορεί να γίνει, χωρίς κανένα πρόβλημα, με τον χαρακτήρα $P_{|x|+1}$, καθώς είναι αδύνατον να υπάρχει στιγμιότυπο του P με θέση εκκινήσεως τον χαρακτήρα T_p, $i-j \leq p < i$, του κειμένου. Ουσιαστικά, με τον τρόπο αυτό, το παράθυρο ολισθαίνει κατά $|z| = j-|x|$ θέσεις δεξιά. Παράλληλα, ο δείκτης j του τρέχοντος χαρακτήρα του προτύπου μεταπηδά στην θέση |x|+1.



Σχήμα 3.3: Ολίσθηση παραθύρου στον αλγόριθμο KMP

Ο αλγόριθμος 3.3 αποτελεί μία ενδεικτική υλοποίηση της μεθόδου των Knuth-Morris-Pratt. Σε γενικές γραμμές, διατηρεί έμμεσα το παράθυρο συγκρίσεως, μέσω των δεικτών θέσεων i, j του κειμένου T και του προτύπου P. Σε περίπτωση επιτυχημένου ταιριάσματος, και οι δύο δείκτες μετακινούνται κατά μία θέση δεξιότερα. Όταν, όμως, η σύγκριση των χαρακτήρων P[j] και T[i] αποβεί ανεπιτυχής, $P[j] \neq T[i]$, τότε το παράθυρο συγκρίσεως ολισθαίνει, έμμεσα, κατά την τιμή του j - f[j-1], διατηρώντας το δείκτη i στην θέση του και μετακινώντας τον δείκτη προτύπου στην θέση f[j-1] + 1.

Αλγόριθμος: KMP(char[] T, char[] P)
Είσοδος: Ένα κείμενο T
Έξοδος: Εντοπισμός του P στο T

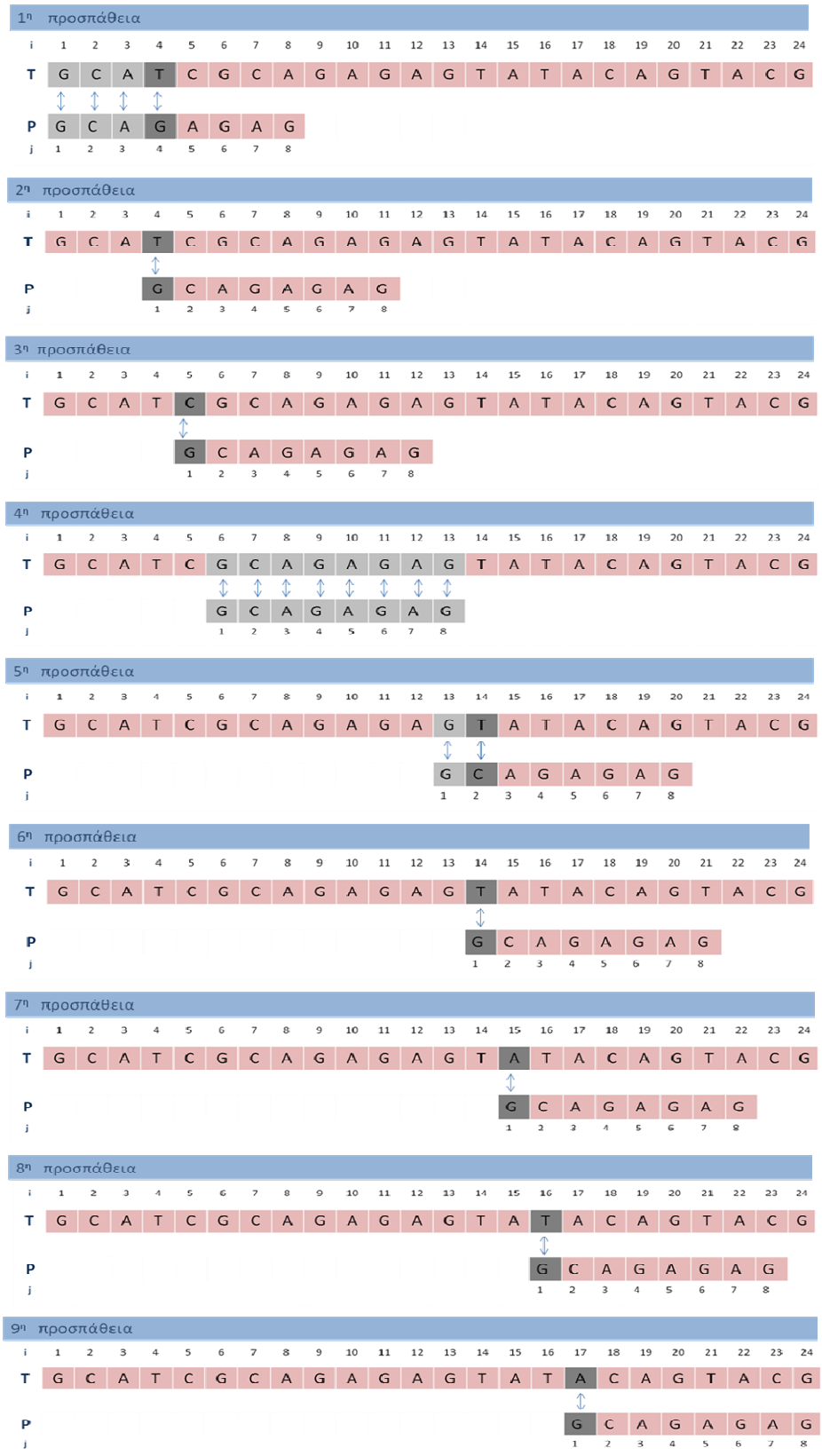
```
1.      int n = T.length; m = P.length;
2.      int f = failureKMP(P, new int[] m)
3.      int i = 1; j = 1;
4.      while (i ≤ n){
5.          if (P[j] == T[j])
6.              if (j = m)                  //το πρότυπο έχει βρεθεί
7.                  return i-m+1;
8.              else{
9.                  i++;
10.                 j++;
11.             }
12.         else if (j > 1)                 //ο δείκτης παραθύρου προχώρησε
13.             j = f(j-1) + 1;             //οπότε γίνεται χρήση της failure
14.         else                            //δοκιμάζουμε νέα τοποθετηση του παραθύρου
15.             i++;                        //η αρχική του θέση
16.     }
17.     return -1;                          //το πρότυπο δεν βρέθηκε
```

Αλγόριθμος 3.3: Αλγόριθμος ταιριάσματος προτύπου βάσει Knuth-Morris-Pratt (KMP)

Στο σχήμα 3.4(β) φαίνεται ένα στιγμιότυπο τρεξίματος του αλγορίθμου KMP, ώστε να εντοπιστεί η εμφάνιση της συμβολοσειράς GCAGAGAG στο κείμενο GCATCGCAGAGAGTATACAGTACG. Έχοντας, πρώτα, υπολογίσει την συνάρτηση αποτυχίας (σχ. 3.4(α)), εκτελούμε την αναζήτηση του προτύπου. Κατά την αρχική τοποθέτηση του παραθύρου, έπειτα από τρεις επιτυχημένες συγκρίσεις, ανακαλύπτεται η πρώτη διαφορά. Στην τέταρτη θέση, το κείμενο έχει το χαρακτήρα 'T', ενώ το πρότυπο το χαρακτήρα 'G'. Σύμφωνα με τον αλγόριθμο, προκαλείται ολίσθηση κατα 3 - f[3] = 3 θέσεις, ενώ παράλληλα, ο τέταρτος χαρακτήρας του κειμένου T συγκρίνεται με τον χαρακτήρα της θέσεως f[3] + 1 = 1 του προτύπου. Στην δύο επόμενες προσπάθειες, διαπιστώνουμε από την πρώτη κιόλας σύγκριση, ότι δεν υπάρχει ταίριασμα, οπότε το παράθυρο ολισθαίνει κατά 1 – f[1] = 1 θέση (j = f[1] + 1 = 1). Στην τέταρτη τοποθέτηση του παραθύρου, εντοπίζεται η πρώτη εμφάνιση του προτύπου στο κείμενο. Κατόπιν τούτου, η διαδικασία επανεκκινείται από την 14η θέση του κειμένου, όπου διαπιστώνεται ότι δεν υπάρχει ταίριασμα κ.ό.κ.

j	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
P[j]	G	C	A	G	A	G	A	G
f[j]	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**

(α)



(β)

Σχήμα 3.4: Αλγόριθμος Knuth-Morris-Pratt: **(α)** Υπολογισμός συνάρτησης αποτυχίας f της συμβολοσειράς P = GCAGAGAG, **(β)** Στιγμιότυπο τρεξίματος

3.3.3 Ανάλυση Πολυπλοκότητας

Όπως αναφέρθηκε και προηγουμένως, ο υπολογισμός της συναρτήσεως αποτυχίας f, στην φάση προεπεξεργασίας του προτύπου, απαιτεί O(m) χρονική και χωρική πολυπλοκότητα, στο μέγεθος m του προτύπου. Επιπρόσθετα, και στην φάση αναζήτησης, η μέθοδος των Knuth-Morris-Pratt, παρουσιάζει γραμμική O(n) πολυπλοκότητα, στην χειρότερη περίπτωση, στο μήκος του κειμένου T, γεγονός που αποδεικνύεται και μέσω του επόμενου θεωρήματος.

Θεώρημα 3.1: *Στον αλγόριθμο KMP, εκτελούνται το πολύ 2n συγκρίσεις χαρακτήρων, όπου n το μήκος του κειμένου T.*

Απόδειξη. Ας θεωρήσουμε ότι ο αλγόριθμος χωρίζεται σε φάσεις ολίσθησης /σύγκρισης, όπου κάθε φάση σηματοδοτείται από την ολίσθηση του παραθύρου και την σύγκριση των αντίστοιχων χαρακτήρων μεταξύ κειμένου T και προτύπου P. Ύστερα από κάθε ολίσθηση, οι συγκρίσεις γίνονται από τα αριστερά προς τα δεξιά και αρχίζουν πάντα, είτε από τον τελευταίο χαρακτήρα του T που συγκρίθηκε στην προηγούμενη φάση, είτε από τον αμέσως επόμενό του. Εφόσων το P δεν ολισθαίνει ποτέ προς τα αριστερά, κάθε φάση, περιλαμβάνει το πολύ μία σύγκριση ενός χαρακτήρα του κειμένου, ο οποίος έχει ήδη συγκριθεί. Επομένως, ο συνολικός αριθμός των συγκρινόμενων χαρακτήρων, δεν ξεπερνάει τους n+s, όπου s το συνολικό πλήθος των ολισθήσεων που πραγματοποιούνται στον αλγόριθμο. Ωστόσο, s < n, εφόσων μετά από n ολισθήσεις, το δεξί άκρο του προτύπου P ευθυγραμμίζεται με το δεξιό άκρο του κειμένου T. Κατά συνέπεια, ο συνολικός αριθμός των συγκρινόμενων χαρακτήρων στον αλγόριθμο, φράσσεται από την τιμή 2n. □

3.4 Ο Αλγόριθμος Boyer-Moore (BM)

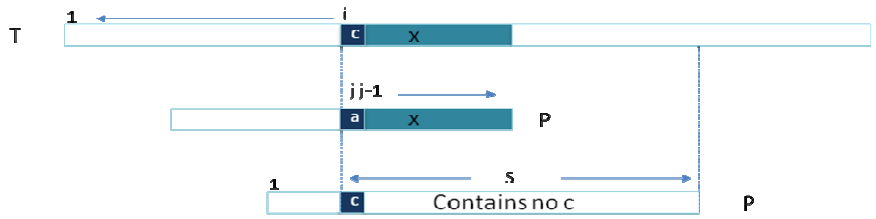
Παράλληλα με τους Knuth, Morris και Pratt, το 1977, οι Bob Boyer και J. Strother Moore, δημοσίευσαν την δική τους μέθοδο αναζήτησης προτύπου σε συμβολοσειρά. Η μέθοδος αυτή αποδείχθηκε ιδιαίτερα αποδοτική σε συνήθεις εφαρμογές, σε σχέση με τις υπόλοιπες, ενώ παράλληλα αποτέλεσε και πρότυπο αναφοράς στην επίλυση του προβλήματος του ακριβούς ταιριάσματος προτύπου (exact pattern matching).

Ο αλγόριθμος των Boyer- Moore (BM), σε αντίθεση με την απλοϊκή λύση και τη μέθοδο των Knuth-Morris-Pratt, σαρώνει το πρότυπο από τα δεξιά προς τα αριστερά, αρχίζοντας πάντα την διαδικασία σύγκρισης από τον δεξιότερο χαρακτήρα. Κάθε φορά που σημειώνεται ένα επιτυχημένο ταίριασμα χαρακτήρα, ο έλεγχος ταιριάσματος μετακινείται προς τα αριστερά. Στην περίπτωση όμως μίας ανεπιτυχούς σύγκρισης ή όταν εντοπιστεί η εμφάνιση του προτύπου στο κείμενο, το παράθυρο ολισθαίνει κατά έναν αριθμό θέσεων προς τα δεξιά. Ο αριθμός αυτός αποτελεί την μέγιστη από τις δύο τιμές που υπαγορεύουν δύο βασικά ευρετικά κριτήρια του αλγορίθμου, το **ευρετικό εμφανίσεως (occurrence heuristic)** ή **ευρετικό κακού χαρακτήρα (bad-character heuristic)** και το **ευρετικό ταιριάσματος (match heuristic)** ή αλλιώς **ευρετικό καλού προσφύματος (good-suffix heuristic)**. Παρακάτω, θα αναλύσουμε ξεχωριστά, κάθε ευρετικό κριτήριο.

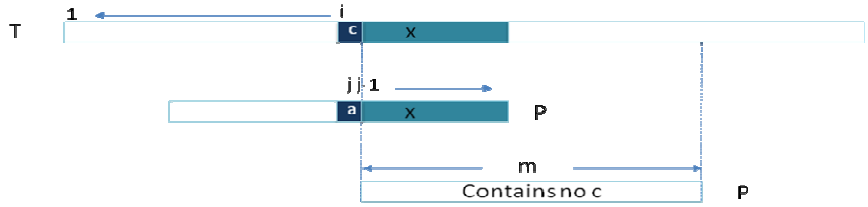
3.4.1 Φάση Προεπεξεργασίας Προτύπου

3.4.1.1 Ευρετικό Κριτήριο Εμφανίσεως

Ας υποθέσουμε ότι ο i-οστός χαρακτήρας του προτύπου P είναι ο 'c', και ότι ο αντίστοιχος χαρακτήρας j του κειμένου με τον οποίο αυτός συγκρίνεται είναι ο 'a' ≠ 'c'. Στην περίπτωση αυτή (σχ. 3.5(α)), αν γνωρίζουμε την δεξιότερη εμφάνιση του χαρακτήρα 'c' στο P, είναι ασφαλές να ολισθήσουμε το παράθυρο προς τα δεξιά, έτσι ώστε αυτός να ευθυγραμμιστεί με τον αντίστοιχο χαρακτήρα 'c' του κειμένου που προκάλεσε την αποτυχία. Βασική, βέβαια, προϋπόθεση αυτής της ολίσθησης, είναι η εμφάνιση του 'c' στο P να είναι αριστερότερα της υπό εξέτασης θέσεως j του προτύπου. Σε αντίθετη περίπτωση, όπου ο χαρακτήρας 'c' δεν εμφανίζεται στο P, το παράθυρο ολισθαίνει, ούτως ώστε το αριστερό του άκρο να ευθυγραμμιστεί με τον αμέσως επόμενο χαρακτήρα του 'c' στο κείμενο, αυτόν, δηλαδή, που βρίσκεται στη θέση i+1 (σχ. 3.5.(β)).



(α)



(β)

Σχήμα 3.5: Ευρετικό κριτήριο εμφανίσεως BM: **(α)** εμφάνιση χαρακτήρα 'c' στο πρότυπο, **(β)** ο χαρακτήρας 'c' δεν υπάρχει στο πρότυπο

Τυπικότερα, το ευρετικό κριτήριο εμφανίσεως, έγκειται στον υπολογισμό ενός πίνακα, ο οποίος σε κάθε του θέση, αντιστοιχίζει κάθε χαρκτήρα 'c' του αλφαβήτου που συναντάται στο κείμενο T, με την επιτρεπόμενη ολίσθηση d(c), βάσει και της δεξιότερης εμφάνισής του στο πρότυπο P. Δηλαδή, η ολίσθηση που προτείνεται για κάθε χαρακτήρα 'c' του αλφαβήτου είναι η

$$d(c) = \min\{s \mid 0 \le s \le m \text{ και } p_{m-c} = c\}.$$

Η υλοποίηση αυτού του υπολογισμού είναι εξαιρετικά απλή (αλγ. 3.4), καθώς αρκεί για κάθε χαρακτήρα να αφαιρεθεί από το m η δεξιότερη εμφάνισή του στο πρότυπο P. Σε περίπτωση που κάποιος χαρακτήρας του κειμένου δεν εμφανίζεται στο πρότυπο, η αντίστοιχη τιμή ολίσθησης είναι m.

```
Αλγόριθμος: occurBM(char[]  P, char[] Σ, int[] d)
Είσοδος: Ένα κείμενο P, ένα αλφάβητο Σ και ένας ακέραιος d
Έξοδος: Υπολογισμός ευρετικού κριτηρίου εμφανίσεως

1.      for (k = 0; k < Σ.length; k++)
2.              d[k] = 0;
3.      for (k = 2; k ≤ P.length; k++){
4.              d[c2i(P[k])] = m-k;        // η συνάρτηση c2i μετατρέπει τον χαρακτήρα
5.      return d;                          // στον αριθμό διατάξεώς του στο Σ
```

Αλγόριθμος 3.4: Υλοποίηση ευρετικού κριτηρίου εμφανίσεως

j	1	2	3	4	5	6	7	8
P[j]	G	C	A	G	A	G	A	G

(α)

A	C	G	T
1	6	1	8

(β)

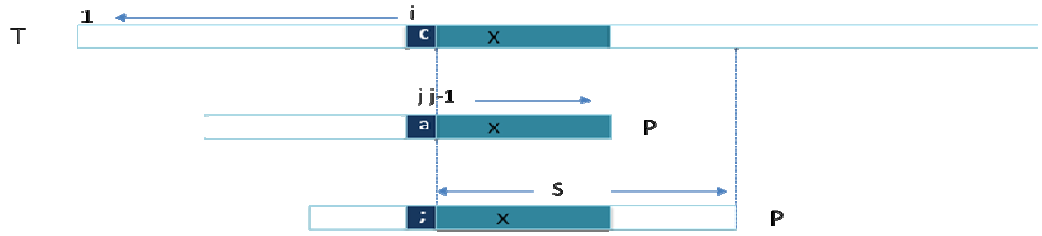
Σχήμα 3.6: Παράδειγμα υπολογισμού α' κριτηρίου αλγορίθμου BM: **(α)** υπολογισμός δεξιότερης εμφάνισης χαρακτήρων του προτύπου, και **(β)** επιτρεπόμενη ολίσθηση κάθε χαρακτήρα που εμφανίζεται στο κείμενο

Στο σχήμα 3.6 παρουσιάζεται ένα παράδειγμα υπολογισμού του πρώτου κριτηρίου του αλγορίθμου BM για T = GCATCGCAGAGAGTATACAGTACG και P = GCAGAGAG. Αφού, αρχικά, εντοπίσουμε τη δεξιότερη εμφάνιση κάθε χαρακτήρα του αλφαβήτου που εμφανίζεται στο κείμενο T (σχ. 3.6(α)), κατόπιν, υπολογίζουμε την επιτρέπομενη ολίσθηση με βάση το κριτήριο εμφανίσεως (σχ. 3.6(β)). Για παράδειγμα, για το χαρκτήρα 'A', του οποίου η πιο δεξιά εμφάνιση είναι στη θέση j=7 του προτύπου, η απόσταση d(A) ισούται με d(A) = m-j = 8-7=1. Επειδή ο χαρακτήρας 'T' δεν εμφανίζεται στο P, θα ισχύει d(T) = 8-0= 8.

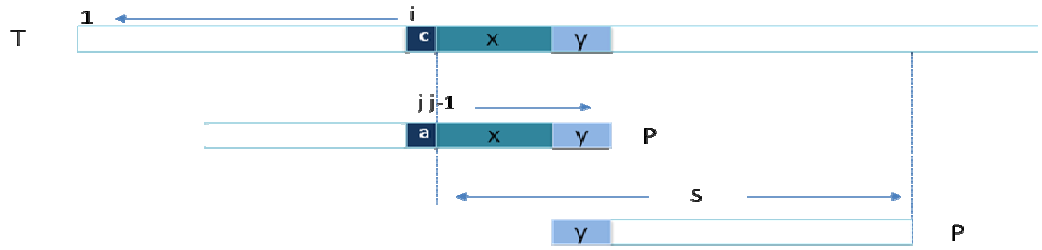
3.4.1.2 Ευρετικό κριτήριο ταιριάσματος

Ας υποθέσουμε, ότι για μία δεδομένη τοποθέτηση του παραθύρου P, έχουν ταιριαστεί επιτυχώς οι j-1 χαρακτήρες του προσφύματος x με τους αντίστοιχους της

υποσυμβολοσειράς T[i...k] του κειμένου T. Τότε, σύμφωνα με το κριτήριο ταιριάσματος του αλγορίθμου BM, αφού εντοπίσουμε το επόμενο στιγμιότυπο του x στο P, κινούμενοι προς τα αριστερά, αρκεί να ολισθήσουμε το παράθυρο δεξιά, τόσο, ώστε η νέα υποσυμβολοσειρά x να ευθυγραμμιστεί με την αντίστοιχη του κειμένου T. Απαραίτητη, ωστόσο, προϋπόθεση της ολίσθησης αυτής, είναι ο χαρακτήρας του προτύπου που προηγείται του ταιριασμένου προσφύματος να είναι διαφορετικός από αυτόν που προκάλεσε την αποτυχία κατά την σύγκρισή του με αυτόν του κειμένου (σχ. 3.7(α)). Σε περίπτωση που δεν υπάρχει άλλο δεξιότερο στιγμιότυπο του x στο P με διαφορετικό χαρακτήρα, τότε προσπαθούμε να ταιριάσουμε όσο το δυνατόν μεγαλύτερο πρόθεμά του με κάποιο πρόσφυμα της υποσυμβολοσειράς x του κειμένου (σχ. 3.7(β)).



(α)



(β)

i	0	1	2	3	4	5	6	7
P[i]	G	C	A	G	A	G	A	G
match	7	7	7	2	7	4	7	1

(γ)

Σχήμα 3.7: Ευρετικό κριτήριο ταιριάσματος αλγορίθμου BM : **(α)** ταίριασμα προσφύματος με διαφορετικό χαρακτήρα, **(β)** ταίριασμα προθέματος, και **(γ)** παράδειγμα υπολογισμού

Η υλοποίηση του ευρετικού κριτηρίου ταιριάσματος (αλγ. 3.5) παρουσιάζει κάποια ομοιότητα με την προεπεξεργασία του προτύπου στον Knuth-Morris-Pratt αλγόριθμο. Έτσι λοιπόν, και εδώ, αρχικά υπολογίζουμε τη συνάρτηση αποτυχίας f_{suf}, με τη μόνη

διαφορά ότι τώρα το πρότυπο σαρώνεται από τα δεξιά προς τα αριστερά. Στο σχήμα 3.7(γ) εικονίζεται ο υπολογισμός του πίνακα match για την συμβολοσειρά "GCAGAGAG".

```
Αλγόριθμος: matchBM(int[] P, int[] match)
Είσοδος: Ένας πίνακας ακεραίων P και ένας πίνακας ακεραίων match
Έξοδος: Υπολογισμός ευρετικού κριτηρίου ταιριάσματος

1.      int fsuf = new int[P.length+1];
2.      for (k = 1; k ≤ P.length; k++)          //αρχικοποίηση
3.              match[k] = m+1;                 //με άκυρη τιμή ολισθήσεως
4.      fsuf[m] = m+1;
5.      for (k = m-1; k ≥ 0; k--){              //υπολογισμός συνάρτησης αποτυχίας
6.              j = fsuf[k+1];
7.              while (j ≤ P.length){
8.                      if (P[k+1] == P[j])
9.                              break;
10.                     match[j] = min(match[j], j-(k+1));
11.                     j = fsuf(j);
12.             }
13.             fsuf[k] = j-1;
14.     }
15.     l =1; shft = fsuf[0];
16.     while (shft ≤ P.length){                //ολίσθηση βάσει μέγιστου ταιριάσματος
17.             for (k = l; k ≤ shft; k++)      //προθέματος σε πρόσφυμα εάν δεν βρέθηκε
18.                     match[k] = min(match[k], shft);   //ενδιάμεση υποσυμβολοσειρά
19.             l = shft+1;
21.             shft = fsuf[shft];
22.     }
23.     for (j = l; j ≤ P.length; j++)          //πρόσθεση στην ολίσθηση του πλήθους
24.             match[j] += (P.length-j);       //των χαρακτήρων που έχουν ταιριαστεί
25.     return match;
```

Αλγόριθμος 3.5: Υλοποίηση ευρετικού κριτηρίου ταιριάσματος

3.4.2 Φάση Αναζήτησης

Έχοντας μελετήσει μέχρι τώρα τα δύο ευρετικά κριτήρια του αλγορίθμου BM, το ευρετικό εμφανίσεως και το ευρετικό ταιριάσματος, μέσω των οποίων εξάγονται οι πίνακες d[] και match[] αντίστοιχα, προχωρούμε πλέον στην φάση αναζήτησης του προτύπου στο κείμενο. Σύμφωνα με τον αλγόριθμο 3.6, σε κάθε τοποθέτηση του παραθύρου, η οποία δηλώνεται από το i, όσο σημειώνονται επιτυχημένες συγκρίσεις χαρακτήρων, οι θέσεις του προτύπου σαρώνονται από τα δεξιά προς τα αριστερά. Σε περίπτωση αποτυχίας, το παράθυρο μετακινείται βάση του κανόνα που δίδει τη μεγαλύτερη ολίσθηση, δηλαδή

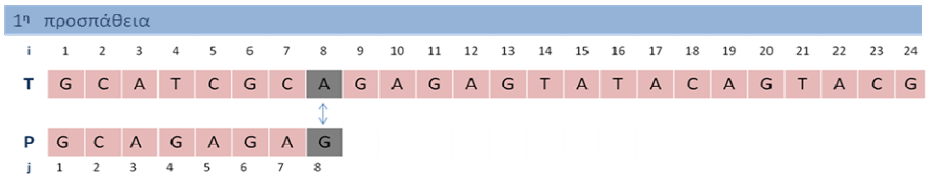
$$i = i + \max\{d(T[i]), match[j]\},$$

όπου i, j, οι δείκτες θέσεως για το κείμενο και το πρότυπο αντίστοιχα.

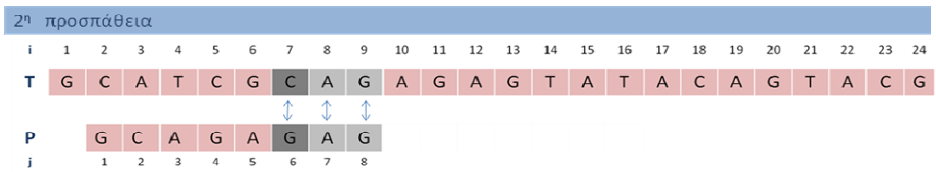
```
Αλγόριθμος: BM(char[]  T, char[] P, char[] Σ)
Είσοδος: Ένα κείμενο T
Έξοδος: Εντοπισμός του P στο T

1.      int n = T.length; m = P.length;
2.      int d = occurBM(P, Σ, new int[Σ.length]);    // συνάρτηση εμφανίσεως
3.      int match = matchBM(P, new int[m]);          // συνάρτηση ταιριάσματος
4.      int i = m;
5.      while (i ≤ n){
6.          j = m;
7.          while ((j>0) && (P[j] = T[i])){       // ταίριασμα χαρακτήρων
8.                  j--;
9.                  i--;
10.         }
11.         if (j == 0)                          // το πρότυπο βρέθηκε
12.                 return i+1;
13.         else                                 // ολίσθηση παραθύρου κατά το μέγιστο
14.                 i += max(match[j], d(T[i]));
15.     }
16.     return -1;
```

Αλγόριθμος 3.6: Αλγόριθμος ταιριάσματος προτύπου βάσει Boyer-Moore

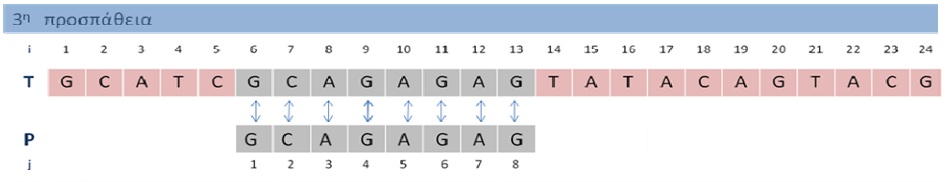
Στο παράδειγμα του σχήματος που ακολουθεί (σχ. 3.8), αιτείται ο εντοπισμός του προτύπου P=GCAGAGAG στο κείμενο T=GCATCGCAGAGAGTATACAGTACG, σύμφωνα με τον αλγόριθμο BM.



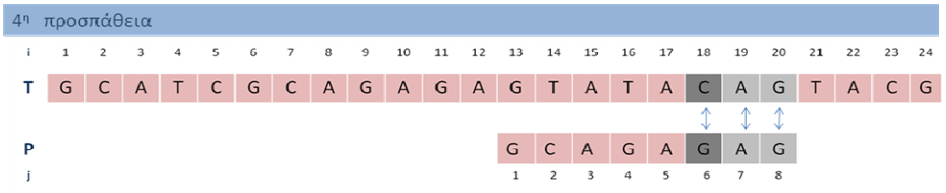
Ολίσθηση κατά 1



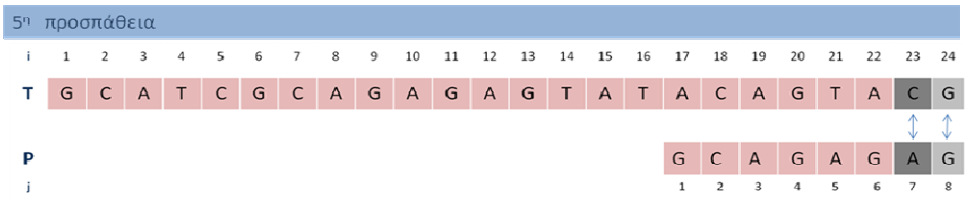
Ολίσθηση κατά 4



Ολίσθηση κατά 7



Ολίσθηση κατά 4



Ολίσθηση κατά 7

Σχήμα 3.8: Παράδειγμα εφαρμογής του BM αλγορίθμου

3.4.3 Ανάλυση Πολυπλοκότητας

Ο αλγόριθμος αναζήτησης έχει πολυπλοκότητα χειρότερης περίπτωσης O(*mn*), δηλαδή όσο και του αλγορίθμου brute-force. Παρόλα αυτά, στην πράξη κατά μέσο όρο είναι υπογραμμικός με πολυπλοκότητα O(*logmn/m*). Έτσι, αν και ο αλγόριθμος Knuth-Morris-Pratt έχει βέλτιστη χειρότερη συμπεριφορά, στην πράξη ο Boyer-Moore κατά μέσο όρο δουλεύει γρηγορότερα, ανακτώντας λιγότερους χαρακτήρες από αυτούς που διαθέτει το πρότυπο και το κείμενο.

Μειονέκτημα του αλγορίθμου BM είναι η εξάρτηση της προεπεξεργασίας από το μέγεθος του αλφάβητου. Έτσι, αν το αλφάβητο είναι μεγάλο τότε είναι προτιμότερη η χρήση του αλγορίθμου KMP. Σε κάθε άλλη περίπτωση και ιδιαίτερα για μεγάλα κείμενα συμφέρει η χρήση του BM.

3.5 Παραλλαγές του Αλγόριθμου BM

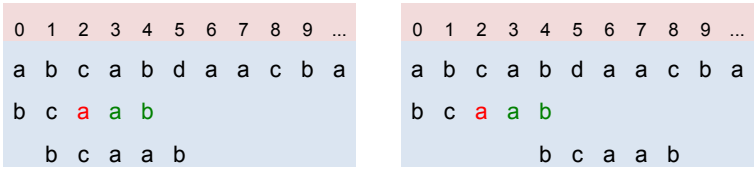
Ο αλγόριθμος Boyer-Moore, αν και παρουσιάζει χειρότερη συμπεριφορά όμοια με του απλοϊκού, στην πράξη δουλεύει πολύ γρήγορα, ανακτώντας λιγότερους χαρακτήρες από αυτούς που διαθέτει το πρότυπο και το κείμενο. Παρόλα αυτά, υπάρχουν κάποιες εκδοχές του (παραλλαγές του αρχικού BM), όπως οι αλγόριθμοι **BM-Horspool** και **Sunday,** που αποτελούν ακόμα γρηγορότερες και αποτελεσματικότερες λύσεις στο πρόβλημα του ταιριάσματος προτύπου.

3.5.1 Ο Αλγόριθμος Horspool

3.5.1.1 Περιγραφή

Ο αλγόριθμος των Boyer-Moore χρησιμοποιεί δύο ευρετικά κριτήρια προκειμένου να καθορίσει την απόσταση ολίσθησης του προτύπου σε περίπτωση αποτυχίας, το ευρετικό εμφανίσεως και το ευρετικό ταιριάσματος. Εφόσον η υλοποίηση του δεύτερου κριτηρίου είναι αρκετά πολύπλοκη και δύσκολη, υπάρχει ανάγκη για έναν πιο απλό αλγόριθμο που να βασίζεται αποκλειστικά στο ευρετικό κριτήριο εμφανίσεως. Έτσι λοιπόν, κατά την άποψη του Horspool, αντί του «κακού χαρακτήρα» που προκάλεσε την αποτυχία, σε κάθε περίπτωση, χρησιμοποιούμε τον δεξιότερο χαρακτήρα του τρέχοντος παραθύρου του κειμένου έτσι ώστε να αποφανθούμε για την μέγιστη επιτρεπόμενη απόσταση ολισθήσεώς του.

Στο παράδειγμα που ακολουθεί γίνεται πιο κατανοητή η διαφορά μεταξύ των δύο αλγορίθμων. Ας υποθέσουμε ότι $t_0,\ldots,t_4$ = abcab είναι το τρέχον παράθυρο του κειμένου το οποίο συγκρίνεται με το πρότυπο p = bcaab. Παρατηρούμε ότι ενώ το πρόσφυμά του ‘ab’ έχει ταιριάξει, η σύγκριση μεταξύ των χαρακτήρων ‘a’,‘c’ του κειμένου και του προτύπου αντίστοιχα, προκαλεί την πρώτη αποτυχία. Το ευρετικό κριτήριο εμφανίσεως του αλγορίθμου Boyer-Moore (σχ. 3.9(α)), χρησιμοποιεί τον “κακό” χαρακτήρα ‘c’ του κειμένου για τον υπολογισμό της απόστασης ολίσθησης. Αντίθετα, ο αλγόριθμος Horspool (σχ. 3.9(β)), κάνει χρήση του δεξιότερου χαρακτήρα ‘b’ του τρέχοντος παραθύρου του κειμένου. Το πρότυπο μπορεί να ολισθήσει τόσες θέσεις έτσι ώστε η δεξιότερη εμφάνιση του χαρακτήρα ‘b’ σε αυτό, εκτός αυτής στη τελευταία θέση, να ταιριάξει με το χαρακτήρα ‘b’ του κειμένου.



(α) Boyer-Moore **(β)** Horspool

Σχήμα 3.9: Ολίσθηση προτύπου με βάση τον: (α) αλγόριθμο Boyer-Moore, και **(β)** αλγόριθμο Horspool

Πιο συγκεκριμένα, ας υποθέσουμε ότι κατά τη διάρκεια αναζήτησης της εμφάνισης ενός οποιουδήποτε προτύπου P σε ένα οποιοδήποτε κείμενο T, 'c' είναι ο χαρακτήρας του τρέχοντος παραθύρου του κειμένου που συγκρίνεται με τον δεξιότερο χαρακτήρα του προτύπου. Διακρίνουμε τις εξής τέσσερις περιπτώσεις:

- Εάν δεν υπάρχει εμφάνιση του χαρακτήρα 'c' στο πρότυπο, ολισθαίνουμε το P κατά m, όπου m το μήκος του προτύπου.

- Εάν ο χαρακτήρας 'c' εμφανίζεται σε μία ή περισσότερες θέσεις του προτύπου **αλλά όχι** στην τελευταία, ολισθαίνουμε το P έτσι ώστε η δεξιότερη εμφάνιση του 'c' να ευθυγραμμιστεί με τον αντίστοιχο χαρακτήρα 'c' του κειμένου.

- Εάν 'c' είναι ο τελευταίος χαρακτήρας του προτύπου και δεν υπάρχουν άλλες εμφανίσεις του 'c' σε αυτό, ολισθαίνουμε το P κατά m.

- Εάν το 'c' είναι ο τελευταίος χαρακτήρας του προτύπου και υπάρχουν και άλλες εμφανίσεις του ίδιου χαρακτήρα σε αυτό, ολισθαίνουμε το P έτσι ώστε η δεξιότερη εμφάνιση του 'c' μεταξύ των m-1 χαρακτήρων του να ευθυγραμμιστεί με το χαρακτήρα 'c' του κειμένου.

Με βάση τα παραπάνω, πριν από την φάση αναζήτησης, κρίνεται σκόπιμη η κατασκευή ενός πίνακα ολίσθησης (ShiftTable). Ο πίνακας αυτός περιέχει μία εισαγωγή για κάθε σύμβολο του αλφαβήτου. Για κάθε χαρακτήρα 'c' που συναντάται στο κείμενο T, η τιμή της ολίσθησης καθορίζεται ως εξής:

- Εάν ο χαρακτήρας 'c' δεν εμφανίζεται ανάμεσα στους m-1 χαρακτήρες του προτύπου, τότε t(c) = m (όπου m το μήκος του προτύπου).

- Διαφορετικά**,** t(c) = η απόσταση της δεξιότερης εμφάνισης του 'c' μεταξύ των m-1 χαρακτήρων του προτύπου P από τον τελευταίο χαρακτήρα του.

Για παράδειγμα, ας θεωρήσουμε ως πρότυπο τη συμβολοσειρά P = abcd με μήκος m=4 και ως κείμενο τη συμβολοσειρά T = abdebcabfdeabgd. Ο πίνακας ολίσθησης που προκύπτει φαίνεται στο σχήμα 3.10.

c	a	b	c	d	e	f	g
t(c)	3	2	1	4	4	4	4

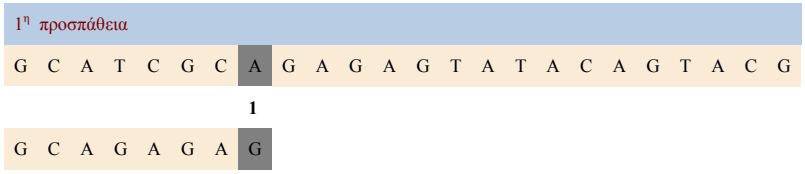
Σχήμα 3.10: Πίνακας ολίσθησης για P=abcd και T= abdebcabfdeabgd

Έχοντας υπολογίσει τον πίνακα ολίσθησης, και αφού ευθυγραμμίσουμε το πρότυπο με την αρχή του κειμένου, ο αλγόριθμος Horspool αρχίζει τη διαδικασία αναζήτησης ως εξής: σαρώνουμε το πρότυπο P, όπως και στον BM αλγόριθμο, από τα δεξιά προς τα αριστερά, συγκρίνοντας έναν προς έναν τους χαρακτήρες του με τους αντίστοιχους

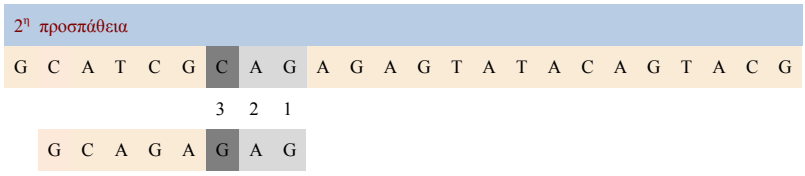
χαρακτήρες του τρέχοντος παραθύρου του κειμένου T. Σε περίπτωση ανεπιτυχούς ταιριάσματος, ολισθαίνουμε το πρότυπο προς τα δεξιά κατά **t(c)** θέσεις, όπου 'c' είναι ο χαρακτήρας του τρέχοντος παραθύρου του κειμένου που συγκρίνεται με τον τελευταίο χαρακτήρα του προτύπου. Κατόπιν, η διαδικασία σύγκρισης συνεχίζεται κατά τον ίδιο τρόπο για κάθε νέα τοποθέτηση του προτύπου P, έως ότου είτε εντοπιστούν μία ή περισσότερες εμφανίσεις του P στο κείμενο, ή έχουμε φτάσει στο τέλος του κειμένου T. Στο σχήμα που ακολουθεί παρουσιάζεται αναλυτικά η αναζήτηση της συμβολοσειράς P = GCAGAGAG στο κείμενο T = GCATCGCAGAGAGTATACAGTACG.

c	A	C	G	T
t(c)	1	6	2	8

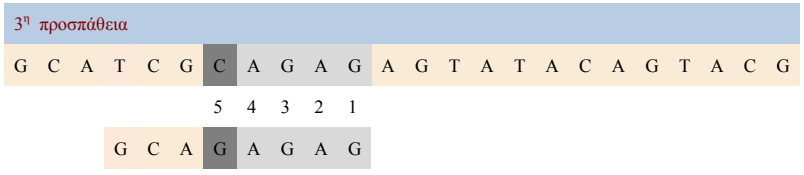
(α)



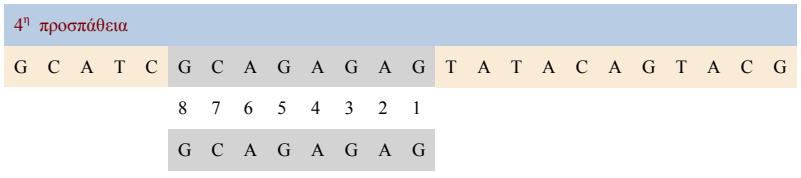
Ολίσθηση κατά **t(A)=1**



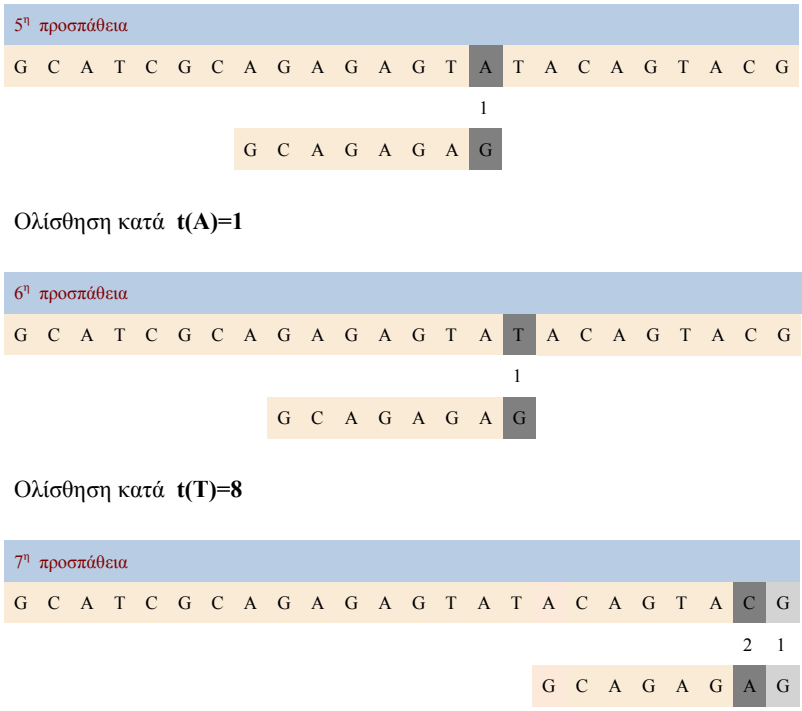
Ολίσθηση κατά **t(G)=2**



Ολίσθηση κατά **t(G)=2**



Ολίσθηση κατά **t(G)=2**



(β)

Σχήμα 3.11: Αλγόριθμος Horspool: **(α)** Υπολογισμός πίνακα ολίσθησης για P = GCAGAGAG και T = GCATCGCAGAGAGTATACAGTACG, **(β)** Στιγμιότυπο τρεξίματος

3.5.1.2 Ανάλυση Πολυπλοκότητας

Η πολυπλοκότητα του αλγορίθμου, στην χειρότερη περίπτωση, είναι O(*nm*), καθώς είναι δυνατόν, κάθε φορά να γίνονται m-1 επιτυχημένες συγκρίσεις και μία αποτυχημένη. Επομένως, θα συμβούν n-m+1 ολισθήσεις, κόστους O(*m*) έκαστη. Ως ακραίο παράδειγμα, το οποίο οδηγεί τον αλγόριθμο σε συμπεριφορά χειρότερης περιπτώσεως, θεωρούμε την αναζήτηση του προτύπου 10…0, δηλαδή, ενός άσσου ακολουθούμενου από m-1 μηδενικά, σε ένα κείμενο αποτελούμενο από n 0.

Ο αλγόριθμος Horspool έχει αποδειχτεί ότι αποτελεί την καλύτερη επιλογή ανάμεσα σε πολλούς αλγορίθμους αναζήτησης προτύπου σε συμβολοσειρά. Αν και τα επίπεδα αποδοτικότητάς του είναι παρόμοια με αυτά της απλοϊκής λύσης, εντούτοις, ο Horspool αλγόριθμος εκτελεί μεγαλύτερα άλματα/ολισθήσεις με αποτέλεσμα αυτό να τον καθιστά γρηγορότερο. Παράλληλα, ο αλγόριθμος αυτός, παραμένει αρκετά αποδοτικός για σχεδόν όλα τα μήκη προτύπων και μεγέθη αλφαβήτων. Λόγω της χαμηλής χωρικής πολυπλοκότητάς του, συνίσταται η χρήση του σε κάθε περίπτωση ακριβούς ταιριάσματος προτύπου, ανεξαρτήτως μεγέθους προτύπου. Επιπρόσθετα, Horspool είναι αρκετά απλός και ως προς την υλοποίησή του.

3.5.2 Ο ΑΛΓΟΡΙΘΜΟΣ SUNDAY

3.5.2.1 Περιγραφή

Όπως έχουμε δει μέχρι τώρα, ο αλγόριθμος Boyer Moore, χρησιμοποιεί τον χαρακτήρα κειμένου που προκάλεσε το ανεπιτυχές ταίριασμα, προκειμένου να υλοποιήσει το ευρετικό κριτήριο ταιριάσματος. Από την άλλη, ο αλγόριθμος Horspool, χρησιμοποιεί το δεξιότερο σύμβολο του τρέχοντος παραθύρου του κειμένου για τον υπολογισμό της μέγιστης επιτρεπόμενης απόστασης ολισθήσεως. Ωστόσο, ένας άλλος επιστήμονας, ο Sunday, παρατήρησε ότι ίσως είναι ακόμα καλύτερα να χρησιμοποιηθεί το αμέσως επόμενο σύμβολο δεξιότερα του παραθύρου του κειμένου, μιας και σε κάθε περίπτωση, ο χαρακτήρας αυτός, περιλαμβάνεται στο επόμενο πιθανό ταίριασμα του προτύπου.

Στο παράδειγμα που ακολουθεί, $t_0,\ldots,t_4$ = abcab είναι το τρέχον παράθυρο του κειμένου το οποίο συγκρίνεται με το πρότυπο p = bcaab. Παρατηρούμε ότι ενώ το πρόσφυμά του 'ab' έχει ταιριάξει, η σύγκριση μεταξύ των χαρακτήρων 'a','c' του κειμένου και του προτύπου αντίστοιχα, προκαλεί την πρώτη αποτυχία. Το ευρετικό κριτήριο εμφανίσεως του αλγορίθμου Boyer-Moore (σχ. 3.12(α)), χρησιμοποιεί τον "κακό" χαρακτήρα 'c' του κειμένου για τον υπολογισμό της απόστασης ολίσθησης. Από την άλλη, ο αλγόριθμος Horspool (σχ. 3.12(β)), κάνει χρήση του δεξιότερου χαρακτήρα 'b' του τρέχοντος παραθύρου του κειμένου, όπως είδαμε και στην προηγούμενη ενότητα. Αντίθετα, σύμφωνα με τον αλγόριθμο που προτάθηκε από τον Sunday (σχ. 3.12(γ)), η σύγκριση του προτύπου με το κείμενο αρχίζει από τον αμέσως επόμενο χαρακτήρα που βρίσκεται δεξιότερα του τρέχοντος παραθύρου του κειμένου, στην συγκεκριμένη περίπτωση το χαρακτήρα 'd'. Ωστόσο, μιας και ο χαρακτήρας 'd' δεν εμφανίζεται στο πρότυπο, αυτό μπορεί να ολισθήσει μία θέση πιο δεξιά, έτσι ώστε ο πρώτος χαρακτήρας του προτύπου, 'b', να ευθυγραμμιστεί με το χαρακτήρα 'a' του κειμένου.

(α) Boyer-Moore **(β)** Horspool **(γ)** Sunday

Σχήμα 3.12: Ολίσθηση προτύπου με βάση τον: **(α)** αλγόριθμο Boyer-Moore, **(β)** αλγόριθμο Horspool, και **(γ)** αλγόριθμο Sunday

Σε αντίθεση με τους αλγορίθμους των Boyer-Moore και Horspool, στη μέθοδο του Sunday, τα σύμβολα του προτύπου μπορούν να συγκριθούν με οποιαδήποτε αυθαίρετη σειρά, και όχι απαραίτητα από τα δεξιά προς τα αριστερά. Για παράδειγμα, η σειρά αυτή μπορεί να καθοριστεί από τις πιθανότητες των συμβόλων, δηλαδή την συχνότητα εμφάνισής τους, υπό την προϋπόθεση ότι αυτές είναι γνωστές εκ των

προτέρων. Έτσι, το σύμβολο με την μικρότερη πιθανότητα συγκρίνεται πρώτο, κι αν το ταίριασμα είναι ανεπιτυχές τότε το πρότυπο μπορεί να ολισθήσει δεξιότερα.

Στο παράδειγμα που ακολουθεί (σχ. 3.13), η σύγκριση των χαρακτήρων αρχίζει με το σύμβολο 'c', μιας και αυτό έχει τη μικρότερη πιθανότητα εμφάνισης. Μετά το πρώτο αποτυχημένο ταίριασμα, το πρότυπο ολισθαίνει δύο θέσεις δεξιότερα, αφού το σύμβολο 'd' δεν εμφανίζεται σε αυτό και άρα μπορεί να προσπεραστεί.

Σχήμα 3.13: Στιγμιότυπο εφαρμογής του αλγορίθμου Sunday

3.5.2.2 Ανάλυση Πολυπλοκότητας

Όπως και στους αλγορίθμους Boyer-Moore και Horspool, έτσι και ο αλγόριθμος Sunday θεωρεί την βέλτιστη περίπτωσή του αν κάθε φορά στην πρώτη σύγκριση το σύμβολο του κειμένου προς σύγκριση δεν εμφανίζεται στο πρότυπο. Υπό αυτήν την προϋπόθεση, ο αλγόριθμος εκτελεί μόλις $O(n/m)$ συγκρίσεις.

3.6 Ο Αλγόριθμος Karp-Rabin

Το 1987, οι Michael Rabin και Richard Karp, παρουσίασαν την δική τους εκδοχή για την επίλυση του προβλήματος του ακριβούς ταιριάσματος προτύπου. Ο αλγόριθμος, τον οποίον πρότειναν, αποτελεί μία τυπική μέθοδο αναζήτησης προτύπου σε συμβολοσειρά, η οποία χρησιμοποιεί την τεχνική κατακερματισμού ή αλλιώς τεχνική **hashing**.

3.6.1 Περιγραφή

Η τεχνική hashing, πρωτοπαρουσιάστηκε από τον Harrison, αλλά αναλύθηκε πλήρως αργότερα από τον Karp και τον Rabin. Θεωρητικά, οι συναρτήσεις κατακερματισμού παρέχουν έναν απλό τρόπο προκειμένου να αποφευχθεί ο τετραγωνικός αριθμός των συγκρινόμενων συμβόλων στις περισσότερες των περιπτώσεων.

Ουσιαστικά, αυτό που κάνουν είναι να μετατρέπουν κάθε συμβολοσειρά σε μία αριθμητική τιμή, η οποία καλείται τιμή κατακερματισμού ή τιμή hash. Η μέθοδος των Karp-Rabin εκμεταλλεύεται το γεγονός ότι αν δύο συμβολοσειρές είναι όμοιες, τότε και οι αντίστοιχές τους τιμές κατακερματισμού είναι επίσης ίσες. Κατά συνέπεια,

αυτό που έχουμε να κάνουμε είναι να υπολογίσουμε την τιμή hash του προτύπου και στη συνέχεια να αναζητήσουμε στη συμβολοσειρά κειμένου για κάποια υποσυμβολοσειρά με την ίδια τιμή hash. Βέβαια, στην πράξη αυτό δεν είναι και τόσο αποτελεσματικό, καθώς παρουσιάζεται ένα σημαντικό πρόβλημα. Επειδή υπάρχουν πάρα πολλές διαφορετικές συμβολοσειρές, προκειμένου να διατηρηθούν σε χαμηλά επίπεδα οι τιμές κατακερματισμού, κρίνεται σκόπιμο να ανατεθούν σε κάποιες από αυτές οι ίδιες τιμές. Αυτό, όμως, σημαίνει ότι αν οι τιμές hash δύο συμβολοσειρών είναι ίσες, τότε δεν είναι απόλυτα σίγουρο ότι και οι συμβολοσειρές θα είναι όμοιες. Θα πρέπει λοιπόν σε αυτήν την περίπτωση να σιγουρευτούμε για το αν είναι ή όχι όμοιες, πράγμα το οποίο κοστίζει αρκετό χρόνο για μεγάλες υποσυμβολοσειρές. Παρόλα αυτά, με χρήση μίας καλής συνάρτησης κατακερματισμού, στη πλειοψηφία των περιπτώσεων αυτό δε θα συμβεί, γεγονός το οποίο διατηρεί τον μέσο χρόνο αναζήτησης σε πολύ ικανοποιητικά επίπεδα.

Η συνάρτηση hashing επιλέγεται έτσι ώστε να παράγει μοναδικές τιμές για όλους τους πιθανούς συνδυασμούς αλφάβητων όλων των μεγεθών. Αυτή η μοναδικότητα περιορίζει το πρόβλημα της αναζήτησης προτύπων σε μία απλή μαθηματική σύγκριση των τιμών hash.

Μία καλή συνάρτηση κατακερματισμού θα πρέπει να έχει τις ακόλουθες ιδιότητες:

1) να είναι αποτελεσματικός ο υπολογισμός

2) να έχει υψηλή διάκριση για τα αλφαριθμητικά, δηλαδή, αν $x \neq y$ τότε και $hash(x) \neq hash(y)$

3) εύκολος υπολογισμός μίας τιμής κατακερματισμού με βάση μία προηγούμενη τιμή, έτσι ώστε να μειωθεί ο χρόνος επεξεργασίας

4) η συνάρτηση πρέπει να είναι της μορφής $h(k)=k \bmod q$, όπου q είναι ένας μεγάλος πρώτος αριθμός ώστε να αποφύγουμε συχνές συγκρούσεις.

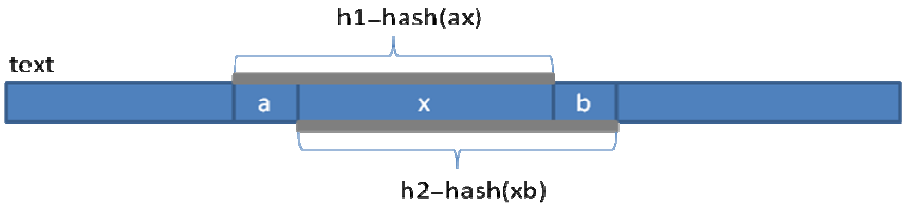
Ο αλγόριθμος Karp-Rabin στηρίζεται στην εξής λογική. Αντί να ελέγχουμε σε κάθε διαδοχική θέση του κειμένου T για το αν το πρότυπο P ταυτίζεται , όπως ακριβώς γίνεται στον αλγόριθμο Brute Force, φαίνεται περισσότερο αποτελεσματικό να ελέγχουμε μόνο αν ένα τμήμα m (όπου m είναι το μήκος του προτύπου P) χαρακτήρων του T είναι ταυτισμένο με το P. Για να μπορέσουμε να ελέγξουμε την ομοιότητα ή την ταύτιση ανάμεσα στο τμήμα m χαρακτήρων του T με το P, πρέπει να χρησιμοποιήσουμε τη συνάρτηση κατακερματισμού. Δηλαδή, κατά τη διάρκεια της φάσης αναζήτησης για το P, ο αλγόριθμος συγκρίνει την τιμή κατακερματισμού των m χαρακτήρων του T[i] σε διαδοχικές θέσεις ανάμεσα από το 0 έως n-m, με την τιμή κατακερματισμού των m χαρακτήρων του P. Εάν δεν έχουμε ισότητα των τιμών κατακερματισμού, τότε μετατοπίζουμε το P μια θέση προς τα δεξιά, υπολογίζουμε την τιμή των επόμενων χαρακτήρων του T και την συγκρίνουμε με την τιμή των m χαρακτήρων του P.

```
1 function RabinKarp(string t[1..n], string p[1..m]){
2     hp := hash(p[1..m])
3     for i from 1 to n-m+1
4         if ht = hp
5             if t[i..i+m-1] = p
6                 return i
7         ht := hash(t[i+1..i+m])
8     return not found
9 }
```

Αλγόριθμος 3.7: Ενδεικτική υλοποίηση του αλγορίθμου Karp-Rabin

Στο σημείο αυτό παρουσιάζεται άλλο ένα σημαντικό πρόβλημα για τον αλγόριθμο. Οι διαδοχικοί υπολογισμοί των τιμών κατακερματισμού των υποσυμβολοσειρών του κειμένου έπειτα από κάθε ολίσθηση του παραθύρου, αυξάνει τη χρονική πολυπλοκότητα του αλγορίθμου, καθιστώντας τον αρκετά αργό, τόσο, όσο και η απλούστερη λύση ακριβούς ταιριάσματος προτύπου που αναλύθηκε στην ενότητα 3.2. Πιο συγκεκριμένα, παρατηρούμε ότι οι γραμμές 2, 5 και 7 του αλγορίθμου 3.7 απαιτούν O(m) χρόνο για τους υπολογισμούς. Ωστόσο, η γραμμή 2 εκτελείται μόνο μία φορά, ενώ η γραμμή 5 εκτελείται μόνο αν οι τιμές κατακερματισμού είναι ίσες, γεγονός που θεωρείται απίθανο να συμβεί πολλές φορές. Έτσι, το μόνο πρόβλημά μας παραμένει η γραμμή 7. Εάν κάθε φορά επαναϋπολογίζαμε την τιμή hash για την υποσυμβολοσειρά t[i+1..i+m], αυτό θα απαιτούσε χρόνο της τάξης O(m), και από τη στιγμή που αυτό γίνεται σε κάθε loop του αλγορίθμου, αυτός θα χρειαζόταν Ω(mn) χρόνο, όπως και η απλοϊκή λύση. Προκειμένου να λύσουμε το πρόβλημα αυτό, αρκεί να παρατηρήσουμε ότι η μεταβλητή hs (γραμμή 7, αλγ. 3.7) περιλαμβάνει την τιμή hash της υποσυμβολοσειράς t[i..i+m-1]. Έτσι λοιπόν, και με βάση την τρίτη ιδιότητα της συνάρτησης κατακερματισμού, που αναφέρθηκε παραπάνω, ο αλγόριθμος υπολογίζει την τιμή hash μίας υποσυμβολοσειράς με βάση αυτή της προηγούμενης (της υποσυμβολοσειράς, δηλαδή, από την οποία διαφέρει κατά ένα χαρακτήρα, έπειτα από κάθε ολίσθηση του παραθύρου) (σχήμα 3.14). Πιο συγκεκριμένα, η μέθοδος Karp-Rabin υπολογίζει την νέα τιμή hash(t[i+1..i+m]) με βάση την ήδη υπολογισμένη τιμή hash(t[i..i+m-1]), με τη βοήθεια της ακόλουθης ισότητας:

$$s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m].$$



Σχήμα 3.14: Υπολογισμός της h2 με βάση την h1, h2=f(a,b,h1)

Ο αλγόριθμος 3.8 αποτελεί μια διαφορετική υλοποίηση του Karp-Rabin με τη μορφή βημάτων. Οι υπολογισμοί των τιμών κατακερματισμού γίνονται με βάση το modulo, έτσι ώστε να αποφύγουμε συχνές συγκρούσεις από ίδιες αναθέσεις τιμών σε διαφορετικές συμβολοσειρές καθώς και τη διαχείριση πολύ μεγάλων τιμών hash.

RABIN-KARP (T, P, n, m)

Βήμα 1: Θέσε: i ← 1, c ← d^(m-1) mod q,
hP ← (P[1]*d^(m-1) + P[2]*d^(m-2)+…+P[m]) mod q
hT ← (T[1]*d^(m-1) + T[2]*d^(m-2)+…+T[m]) mod q

Βήμα 2: Αν i≤ n-m+1 τότε πήγαινε στο **βήμα 3**, διαφορετικά πήγαινε στο **βήμα 5**

Βήμα 3: Αν (hP = hT) και (P[0..m-1]=T[i..i+m-1]) τότε εμφάνισε την θέση i,
διαφορετικά θέσε:

hT ← (hT+d*q-T[i]*c) mod q
hT ← (hT+d+T[i]+m) mod q
i ← i+1

Βήμα 4: Πήγαινε στο **βήμα 2**

Βήμα 5: Αν i > n-m+1 τότε θέσε i ← 0, διαφορετικά εκτύπωσε την θέση i

Βήμα 6: Τερμάτισε

Αλγόριθμος 3.8: Ενδεικτική υλοποίηση του αλγορίθμου Karp-Rabin με τη μορφή βημάτων

Κλειδί για την επιτυχία και την καλή απόδοση του αλγορίθμου Karp-Rabin καθίσταται ο αποτελεσματικός υπολογισμός των τιμών κατακερματισμού των διαδοχικών συμβολοσειρών του κειμένου. Μία πολύ καλή επιλογή αποτελεί η **rolling hash** συνάρτηση, η οποία χρησιμοποιώντας μόνο πολλαπλασιασμούς και προσθέσεις, μετατρέπει κάθε υποσυμβολοσειρά σε έναν αριθμό σύμφωνα με τον ακόλουθο τύπο:

$$H = c_1 * b^{k-1} + c_2 * b^{k-2} + c_3 * b^{k-3} \ldots + c_k * b^0. \quad (3.1)$$

Προκειμένου να αποφύγουμε τον χειρισμό πολύ μεγάλων τιμών H, όλες οι πράξεις γίνονται με βάση το ***modulo n***. Πολύ σημαντική αποτελεί και η επιλογή των τιμών b και n, ούτως ώστε να πετύχουμε έναν καλό κατακερματισμό. Συνήθως, η τιμή b, η βάση δηλαδή που χρησιμοποιούμε, επιλέγεται να είναι ένας μεγάλος πρώτος αριθμός. Ολίσθηση όλων των χαρακτήρων (του τρέχοντος παραθύρου κειμένου) κατά μία θέση αριστερά ισοδυναμεί με πολλαπλασιασμό ολόκληρου του αθροίσματος H με το b. Αντίθετα, η ολίσθηση κατά μία θέση δεξιότερα απαιτεί τη διαίρεση του H με την βάση b. Ας σημειωθεί επίσης ότι στην αριθμητική με βάση το modulo, το b διαθέτει και έναν πολλαπλασιαστικό αντίστροφο, το b^*, με τον οποίον αν πολλαπλασιαστεί το άθροισμα H μπορούμε να πάρουμε το αποτέλεσμα της διαίρεσης χωρίς ουσιαστικά να εκτελέσουμε την πράξη της διαίρεσης.

Με βάση τα παραπάνω, αν υποθέσουμε ότι έχουμε την υποσυμβολοσειρά **'abc'** και η βάση που έχουμε επιλέξει είναι ο αριθμός 101, τότε σύμφωνα και με τον παραπάνω

τύπο (3.1), η τιμή hash που προκύπτει θα ισούται με $97\times101^2 + 98\times101^1 + 99\times101^0 =$ 999494 (97 είναι η αναπαράσταση ASCII για το 'a', 98 για το 'b' και 99 για το 'c').

Το ουσιαστικό όφελος που αποκομίζουμε χρησιμοποιώντας αυτήν την αναπαράσταση, αποτελεί το γεγονός ότι μπορούμε εύκολα να υπολογίσουμε την τιμή hash της επόμενης υποσυμβολοσειράς με βάση την προηγούμενη, εκτελώντας μόνο έναν σταθερό αριθμό μαθηματικών πράξεων, ανεξάρτητα από τα μήκη των ίδιων των συμβολοσειρών.

Για παράδειγμα, ας υποθέσουμε ότι έχουμε το κείμενο **'abracadabra'** και ότι αναζητούμε σε αυτό ένα πρότυπο μήκους 3. Προκειμένου να υπολογίσουμε την τιμή hash της υποσυμβολοσειράς 'bra' με βάση την τιμή της 'abr' (υποσυμβολοσειρά προτού μετακινήσουμε το παράθυρο κειμένου), αρκεί να αφαιρέσουμε τον αριθμό 97×101^2 που προστέθηκε για το αρχικό 'a' της 'abr', να πολλαπλασιάσουμε με την βάση (στην περίπτωσή μας χρησιμοποιούμε την τιμή 101) και τέλος να προσθέσουμε την τιμή $97 \times 101^0 = 97$ που αφορά τον τελευταίο όρο 'a' της 'bra'.

Στο σχήμα 3.15 παρουσιάζεται λεπτομερώς, η αναζήτηση του προτύπου GCAGAGAG στο κείμενο GCATCGCAGAGAGTATACAGTACG με βάση τον αλγόριθμο Karp-Rabin. Αρχικά, υπολογίζουμε τη hash τιμή του προτύπου προς αναζήτηση. Ο υπολογισμός γίνεται σύμφωνα με τον τύπο 3.1 και για βάση χρησιμοποιούμε τη τιμή 2, b=2. Έτσι λοιπόν, για το P προκύπτει η παρακάτω τιμή:

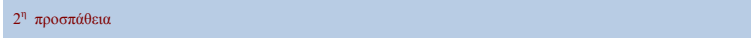
$$\begin{aligned} H[GCAGAGAG] &= 71\times2^7+67\times2^6+65\times2^5+71\times2^4+65\times2^3+71\times2^2+65\times2^1+71\times2^0 \\ &= 17597 \end{aligned}$$

Στη συνέχεια, για κάθε νέα τοποθέτηση του παραθύρου κειμένου, το οποίο έχει μήκος όσο και το μέγεθος του προτύπου, m, υπολογίζουμε την τιμή κατακερματισμού της αντίστοιχης υποσυμβολοσειράς. Στην πρώτη, λοιπόν, προσπάθεια, η τιμή hash της 'GCATCGCA' που προκύπτει είναι η ακόλουθη:

$$\begin{aligned} H[GCATCGCA] &= 71\times2^7+67\times2^6+65\times2^5+84\times2^4+67\times2^3+71\times2^2+67\times2^1+65\times2^0 \\ &= 9088+4288+2080+1344+536+284+134+65 \\ &= 17819 \end{aligned}$$

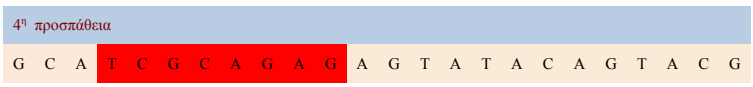
Συγκρίνοντάς την με την αντίστοιχη τιμή hash του προτύπου, διαπιστώνουμε ότι οι δύο τιμές δεν είναι ίσες και άρα και οι αντίστοιχες συμβολοσειρές είναι ανόμοιες. Η διαδικασία συνεχίζεται με τον ίδιο τρόπο, μέχρι να φτάσουμε στην $6^η$ προσπάθεια, οπότε και οι δύο τιμές hash που υπολογίζουμε είναι ίσες, H[GCAGAGAG] = H[GCAGAGAG] = 17597. Αν και κάποιες φορές η ισότητα αυτή δε συνεπάγεται απαραίτητα και ομοιότητα των αντίστοιχων συμβολοσειρών, στην συγκεκριμένη περίπτωση έχουμε το πρώτο επιτυχημένο ταίριασμα.

H[GCATCGCA]= 17819 ≠ H[GCAGAGAG]=17597

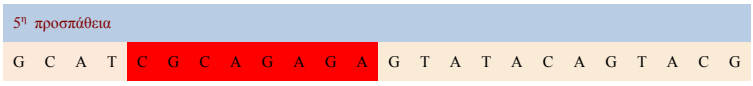


H[CATCGCAG]= 17533 ≠ H[GCAGAGAG]=17597

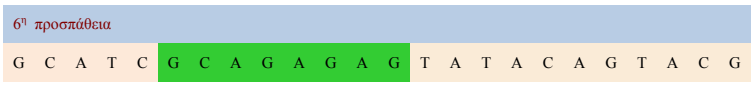
H[ATCGCAGA]= 17979 ≠ H[GCAGAGAG]=17597



H[TCGCAGAG]= 19389 ≠ H[GCAGAGAG]=17597



H[CGCAGAGA]= 17339 ≠ H[GCAGAGAG]=17597



H[GCAGAGAG]= 17597 = H[GCAGAGAG]
Το πρότυπο βρέθηκε!

H[CAGAGAGT]= 17102 ≠ H[GCAGAGAG]=17597

H[AGAGAGTA]= 17117 ≠ H[GCAGAGAG]=17597

H[GAGAGTAT]= 17678 ≠ H[GCAGAGAG]=17597

H[AGAGTATA]= 17245 ≠ H[GCAGAGAG]=17597

H[GAGTATAC]= 17917 ≠ H[GCAGAGAG]=17597

H[AGTATACA]= 17723 ≠ H[GCAGAGAG]=17597

H[GTATACAG]= 18877 ≠ H[GCAGAGAG]=17597

H[TATACAGT]= 19662 ≠ H[GCAGAGAG]=17597

H[ATACAGTA]= 17885 ≠ H[GCAGAGAG]=17597

H[TACAGTAC]= 19197 ≠ H[GCAGAGAG]=17597

H[ACAGTACG]= 16961 ≠ H[GCAGAGAG]=17597

Σχήμα 3.15: Παράδειγμα εφαρμογής του αλγόριθμου Karp-Rabin

3.6.2 Ανάλυση Πολυπλοκότητας

Όσον αφορά την χρονική και χωρική πολυπλοκότητα του αλγορίθμου Karp-Rabin, ο υπολογισμός της τιμής hash ενός προτύπου με μήκος m, απαιτεί σταθερό χώρο και O(m) χρόνο. Από την άλλη μεριά, η εφαρμογή της συνάρτησης κατακερματισμού στις διαδοχικές υποσυμβολοσειρές του κειμένου μήκους n, αιτεί O(mn) χρονική πολυπλοκότητα, ενώ ο αναμενόμενος αριθμός των συγκρινόμενων χαρακτήρων είναι O($m+n$).

Πιο συγκεκριμένα, προκειμένου να υπολογίσουμε τον συνολικό αριθμό αριθμητικών πράξεων που λαμβάνουν χώρα στον αλγόριθμο, θα διακρίνουμε δύο διαφορετικές περιπτώσεις, ανάλογα με το αν το πρότυπο εμφανίζεται μέσα στο κείμενο ή όχι. Έστω, c1 το κόστος εφαρμογής της συνάρτησης hashing στο πρότυπο ή στην πρώτη υποσυμβολοσειρά του κειμένου προς εξέταση, και c2 το κόστος από την εφαρμογή της hashing σε κάθε νέα διαδοχική υποσυμβολοσειρά με βάση την προηγούμενή της. Τότε, έχουμε:

Περίπτωση 1: Το πρότυπο δεν εμφανίζεται στο κείμενο:

Συνολικό κόστος: $2 \times c1 + (n-m) \times c2 + (n–m+1)$ συγκρίσεις

Περίπτωση 2: Το πρότυπο εμφανίζεται στο κείμενο:

- Το πρότυπο εντοπίζεται στην πρώτη προσπάθεια (αρχή του κειμένου):

  Συνολικό κόστος: $2 \times c1 + 1$ σύγκριση

- Το πρότυπο εντοπίζεται στην τελευταία προσπάθεια (τέλος του κειμένου):

  Συνολικό κόστος: $2 \times c1 + (n-m) \times c2 + (n–m+1)$ συγκρίσεις

- Το πρότυπο εντοπίζεται κάπου στη μέση του κειμένου:

  Συνολικό κόστος: $2 \times c1 + ((n-m)/2) \times c2 + ((n–m+1)/2)$ συγκρίσεις.

3.7 Παραλλαγές του Αλγόριθμου Karp-Rabin

Ο αλγόριθμος Karp-Rabin υστερεί στην απλή αναζήτηση προτύπου σε σχέση με άλλους γρηγορότερους αλγορίθμους, όπως ο Knuth Morris Pratt και ο Boyer Moore, εξαιτίας της αργής του συμπεριφοράς σε κατάσταση χειρότερης περιπτώσεως. Αντίθετα, ο αλγόριθμος αυτός αποτελεί μία καλή επιλογή για την αναζήτηση πολλαπλών προτύπων (multiple pattern searching).

Για το λόγο αυτό, έχουν δημιουργηθεί διάφορες παραλλαγές του Karp-Rabin, χάρη στις οποίες μπορούμε να αναζητήσουμε οποιαδήποτε συμβολοσειρά η οποία ανήκει σε ένα δεδομένο σύνολο προτύπων, καθορισμένου μήκους, μέσα σε ένα κείμενο. Συνήθως, οι αλγόριθμοι αυτοί στηρίζονται σε **πιθανολογικές δομές δεδομένων (probabilistic data structures)**, όπως είναι τα φίλτρα Bloom (Bloom filters), ή σε απλές καθορισμένες δομές (set data structure), όπως για παράδειγμα οι πίνακες. Ουσιαστικά, αυτό που κάνουν είναι να ελέγχουν εάν η τιμή κατακερματισμού μίας δεδομένης συμβολοσειράς ανήκει στο σύνολο τιμών hash των προτύπων για τα οποία ενδιαφερόμαστε.

```
1 function RabinKarpSet(string t[1..n], set of string subs, m) {
2     set hsubs := emptySet
3     for each sub in subs
4         insert hash(sub[1..m]) into hsubs
5     ht := hash(t[1..m])
6     for i from 1 to n-m+1
7         if ht ∈ hsubs
8            if t[i..i+m-1] = a substring with hash ht
```

```
9                   return i
10              ht := hash(t[i+1..i+m])
11          return not found
12  }
```

Αλγόριθμος 3.9: Παραλλαγή του Karp-Rabin αλγόριθμου

Ο αλγόριθμος 3.8, ο οποίος αποτελεί μία παραλλαγή του Karp-Rabin αλγόριθμου, συγκρίνει την τρέχουσα τιμή hash με τις αντίστοιχες τιμές όλων των υποσυμβολοσειρών ταυτόχρονα, εκτελώντας μία γρήγορη αναζήτηση στη δεδομένη δομή δεδομένων, (στην συγκεκριμένη περίπτωση στο σύνολο *hsubs*), και επιβεβαιώνοντας κάθε επιτυχημένο ταίριασμα. Στην υλοποίηση αυτή, υποθέτουμε ότι όλες οι υποσυμβολοσειρές έχουν σταθερό μέγεθος m, αν και αυτό δεν είναι απαραίτητο.

Όσον αφορά την πολυπλοκότητα, αξίζει να σημειωθεί ότι αλγόριθμοι οι οποίοι ψάχνουν για ένα πρότυπο σε O(n) χρόνο, χρειάζονται για την αναζήτηση k προτύπων O(nk) χρόνο. Σε αντίθεση με αυτούς, η συγκεκριμένη παραλλαγή του Karp-Rabin, μπορεί να εντοπίσει τις k συμβολοσειρές προτύπων σε O(n+k) χρόνο, καθώς ένας πίνακας hash ελέγχει για το αν η τιμή hash κάποιας υποσυμβολοσειράς ισούται με κάποια από τις αντίστοιχες τιμές των προτύπων σε σταθερό O(1) χρόνο.

Βιβλιογραφία Κεφαλαίου

[1] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[2] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[3] Π.Δ. Μποζάνης, *Αλγόριθμοι: Σχεδιασμός και Ανάλυση*, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002.

[5] Yu Jiangsheng, *String Matching*, Institute of Computational Linguistics, Peking University, 2002.

[6] James Worrell, *String-matching Algorithms,* Oxford University Computing Laboratory, UK.

[7] http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/indexen.htm

4 Προσεγγιστικό Ταίριασμα Προτύπου

Περιεχόμενα

4.1 Εισαγωγή

Συχνά απαιτείται να βρεθεί πόσο όμοιες είναι δύο συμβολοσειρές x, y. Για παράδειγμα, ένα πρόγραμμα-αράχνη των μηχανών αναζητήσεως (search engine) πρέπει να κρίνει εάν και κατά πόσο οι σελίδες που προσπελαύνει είναι διαφορετικές, ώστε να μην τις καταχωρεί άδικα. Από την άλλη, η επιστήμη της βιολογίας (Βιοπληροφορική - Bioinformatics) αποφαίνεται πως ομοιότητα στις ακολουθίες-συμβολοσειρές DNA, RNA κτλ συνεπάγεται και δομική ή λειτουργική ομοιότητα, καθώς στην φύση, συχνά, παρατηρείται πλεονασμός. Στη συνέχεια, θα μελετήσουμε τους βασικότερους τρόπους μέτρησης της ομοιότητας, όπως η απόσταση διαμορφώσεως, η μέγιστη κοινή υπακολουθία, κ.α.

4.2 Ελάχιστη Απόσταση Διαμορφώσεως (Edit Distance)

Ένα φυσικό μέτρο της απόστασης μεταξύ δύο συμβολοσειρών είναι ο βαθμός στον οποίον αυτές μπορούν να ευθυγραμμιστούν ή να ταιριάξουν. Τεχνικά, ευθυγράμμιση δύο συμβολοσειρών/ακολουθιών είναι απλά ένας τρόπος σύγκρισης των χαρακτήρων/συμβόλων που τις απαρτίζουν. Για παράδειγμα, στο σχήμα 4.1 φαίνονται δύο πιθανές ευθυγραμμίσεις των λέξεων ‘SNOWY’ και ‘SUNNY’:

S	-	N	O	W	Y	
S	U	N	N	-	Y	

(α)

-	S	N	O	W	-	Y
S	U	N	-	-	N	Y

(β)

Σχήμα 4.1: Δύο διαφορετικές ευθυγραμμίσεις για τις λέξεις ‘SNOWY’ και ‘SUNNY’: το κόστος στην περίπτωση **(α)** είναι 3, ενώ στην περίπτωση **(β)** είναι 5

Με το σύμβολο ‘-’ υποδεικνύονται οι κενοί χαρακτήρες ή αλλιώς τα **κενά (gaps)**. Σε κάθε συμβολοσειρά μπορεί να τοποθετηθεί οποιοσδήποτε αριθμός από αυτά. Το κόστος μίας ευθυγράμμισης δύο συμβολοσειρών αντιστοιχεί στον αριθμό των συμβόλων που αυτές διαφέρουν. Έτσι λοιπόν και η **ελάχιστη απόσταση διαμορφώσεως (edit distance)** μεταξύ δύο συμβολοσειρών αποτελεί το κόστος της βέλτιστης δυνατής ευθυγράμμισής τους. Στο παραπάνω σχήμα (σχ. 4.1) εύκολα παρατηρούμε ότι η βέλτιστη ευθυγράμμιση μεταξύ των λέξεων ‘SNOWY’ και ‘SUNNY’ είναι η πρώτη περίπτωση (σχ. 4.1(α)), μιας και το κόστος αυτής (κόστος = 3) είναι μικρότερο σε σχέση με αυτό της δεύτερης (σχ. 4.1(β)) περίπτωσης (κόστος = 5).

Το πρόβλημα της διαμορφώσεως μίας συμβολοσειράς x βάσει μίας δεύτερης y αιτεί τον μετασχηματισμό της x στην μορφή της y. Έτσι λοιπόν, η **ελάχιστη απόσταση διαμορφώσεως (edit distance)** είναι ο ελάχιστος αριθμός από πράξεις (διαμορφώσεις) που απαιτούνται προκειμένου να μετατραπεί η συμβολοσειρά x στην y, και συμβολίζεται ως **edit(x,y)**. Για το λόγο αυτό υπάρχουν 3 βασικές πράξεις:

- **Διαγραφή (deletion)** ενός χαρακτήρα c της x
- **Ένθεση (insertion)** ενός χαρακτήρα c στην x
- **Αντικατάσταση (substitution)** ενός χαρακτήρα c της x με κάποιον άλλον c'.

Για παράδειγμα, η μετατροπή της λέξεως 'SNOWY' σε 'SUNNY' με βάση το σχήμα 4.1(α) απαιτεί τις ακόλουθες τρεις πράξεις: ένθεση(U), αντικατάσταση(O→N) και διαγραφή(W).

Εφόσον αναφερόμαστε σε απόσταση μεταξύ λέξεων/συμβολοσειρών, αυτομάτως θα πρέπει να ικανοποιούνται και οι ακόλουθες ιδιότητες:

- edit(x, y) ≥ 0,
- edit(x, y) = 0 εάν x = y,
- edit(x, y) = edit(y, x) (συμμετρική ιδιότητα),
- edit(x, y) ≤ edit(x, z) + edit(z, y) (τριαδική ανισότητα).

Η συμμετρική ιδιότητα οφείλεται στην δυαδικότητα μεταξύ των διαγραφών και των ενθέσεων. Ουσιαστικά, η διαγραφή ενός χαρακτήρα, έστω c, από την συμβολοσειρά *x* προκείμενου να πάρουμε την *y*, ισοδυναμεί με την ένθεση του c στην *y* έτσι ώστε να προκύψει η *x*.

Γενικά, μπορεί να υπάρχουν πάρα πολλοί πιθανοί εναλλακτικοί τρόποι μετατροπής μίας συμβολοσειράς σε μία άλλη, γεγονός που καθιστά ιδιαίτερα αναποτελεσματική την εξέταση όλων αυτών προκειμένου να βρεθεί η βέλτιστη δυνατή. Για το λόγο αυτό, θα παρουσιάσουμε στη συνέχεια μία λύση η οποία βασίζεται στην τεχνική του δυναμικού προγραμματισμού.

4.2.1 Υπολογισμός Ε.Α.Δ. με Χρήση Δυναμικού Προγραμματισμού

Ας υποθέσουμε ότι έχουμε τις ακόλουθες δύο συμβολοσειρές εισόδου, *n* και *m* χαρακτήρων αντίστοιχα:

$$x = x_1 x_2 \ldots x_{n-1} x_n \qquad \text{και} \qquad y = y_1 y_2 \ldots y_{m-1} y_m ,$$

ενώ με x_i , y_j, συμβολίζουμε τα προθέματα, μήκους $0<i\leq n$ και $0<j\leq m$, των *x*, *y* αντίστοιχα. Ορίζουμε ως **edit(*i, j*)** την ελάχιστη απόσταση διαμορφώσεως μεταξύ των πρώτων *i* χαρακτήρων της συμβολοσειράς *x* και των πρώτων *j* χαρακτήρων της συμβολοσειράς *y*:

$$\text{edit}(i, j) = \text{edit}(x[1..i], y[1..j]) .$$

Η ελάχιστη απόσταση διαμορφώσεως μεταξύ ολόκληρων των δύο συμβολοσειρών συμβολίζεται ως **edit(*n, m*)**. Επίσης, ορίζουμε ως **edit(*0, m*)** την ελάχιστη απόσταση διαμορφώσεως μεταξύ της κενής συμβολοσειράς *x* και της *y*, η οποία ισούται με ***m***. Αντιστοίχως, ισχύει **edit(*n, 0*) = *n***.

$$\text{edit}(n, 0) = n \qquad \text{edit}(n, 0) = n$$

Αυτό που καλούμαστε λοιπόν να κάνουμε εδώ είναι ο υπολογισμός της edit(*n, m*) με βάση έναν αριθμό υποπροβλημάτων της μορφής edit(*i,j*).

Έστω ότι βρισκόμαστε στο τελευταίο στάδιο του μετασχηματισμού της *x* στην *y* και ότι για κάθε πράξη απαιτείται μοναδιαίο κόστος, *O(1)*. Οι τελευταίοι χαρακτήρες x_n και y_m, μπορεί είτε να υπάρχουν στις x, y αντίστοιχα, είτε όχι. Κατά συνέπεια, υπάρχουν μόνο τρεις διαφορετικές πιθανές πράξεις για τη διενέργεια αυτού του μετασχηματισμού:

- **Ένθεση(y_m):** ο τελευταίος χαρακτήρας της *y* δεν υπάρχει, οπότε αρχικά η *x* μετασχηματίζεται στην υποσυμβολοσειρά y_{m-1} και ακολούθως προστίθεται ο χαρακτήρας y_m. Στην περίπτωση αυτή, edit(*n, m*) = edit(*n, m-1*) + 1.

- **Διαγραφή(x_n)**: ο τελευταίος χαρακτήρας της x περισσεύει στον βέλτιστο μετασχηματισμό, οπότε και διαγράφεται. Στην περίπτωση αυτή, edit(*n, m*) = edit(*n-1, m*) + 1.

- **Αντικατάσταση(x_n, y_m):** οι τελευταίοι χαρακτήρες x_n και y_m υπάρχουν στις συμβολοσειρές *x* , *y* αντίστοιχα. Εάν αυτοί είναι ίδιοι ($x_n = y_m$), τότε αρκεί απλά η βέλτιστη μετατροπή από την υποσυμβολοσειρά x_{n-1} στην y_{m-1}, με edit(*n, m*) = edit(*n-1, m-1*) . Εάν οι χαρακτήρες είναι διαφορετικοί ($x_n \neq y_m$), τότε αρχικά μετατρέπεται η x_{n-1} σε y_{m-1} και κατόπιν αντικαθίσταται ο χαρακτήρας x_n από τον y_m. Στην περίπτωση αυτή, edit(*n, m*) = edit(*n-1, m-1*) +1.

Συμπερασματικά, η ελάχιστη απόσταση διαμορφώσεως, edit(n,m), μεταξύ δύο συμβολοσειρών x, y, με μήκη n και m χαρακτήρες αντίστοιχα, είναι η μικρότερη από τις ακόλουθες τέσσερις τιμές:

$$\textbf{edit(n, m)} = \min \begin{cases} \text{edit}(n, m-1) + 1 \\ \text{edit}(n-1, m) + 1 \\ \text{edit}(n-1, m-1) + \partial(x_n, y_m) \end{cases} \qquad (4.1)$$

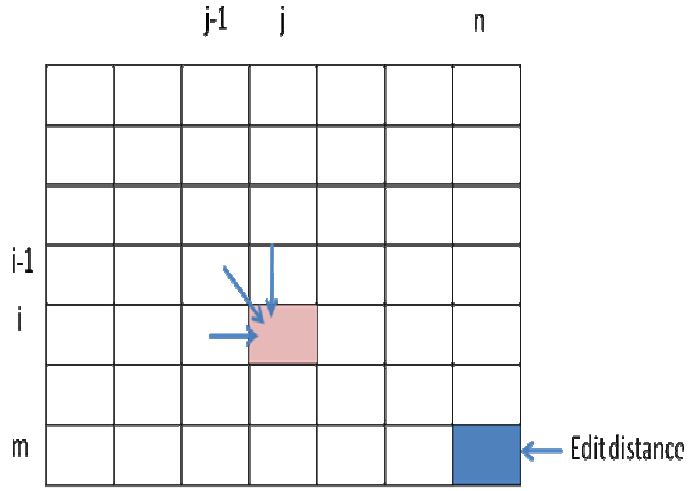
Η συνάρτηση ∂ επιστρέφει την τιμή 0 εάν οι τελευταίοι χαρακτήρες x_n, y_m είναι ίδιοι ($x_n = y_m$), διαφορετικά επιστρέφει 1, όσο δηλαδή και το απαιτούμενο κόστος μίας πράξης.

Αλγόριθμος: edit(**char**[] x, **char**[] y)
Είσοδος: Οι ακολουθίες x, y
Έξοδος: Η ελάχιστη απόσταση διαμορφώσεως

```
1.      |x|=m;   |y|=n;   int EDIT[];
2.      for i := 0 to m do EDIT[i, 0] := i;
3.      for j := 1 to n  do EDIT[0, j] := j;
4.      for i := 1 to m do
5.              for j := 1 to n  do
6.                      EDIT[i, j] = min(EDIT[i-1, j]+1, EDIT[i, j-1]+1,
7.                                              EDIT[i-1, j-1] + ∂(xi, yi) );
8.      return EDIT[m, n];
```

Αλγόριθμος 4.1: Εύρεση ελάχιστης απόστασης διαμορφώσεως

Ο αλγόριθμος 4.1 υπολογίζει την ελάχιστη απόσταση διαμορφώσεως των συμβολοσειρών *x, y*. Ουσιαστικά, ο αλγόριθμος αυτός υπολογίζει έναν *m*×*n* πίνακα για όλες τις δυνατές τιμές EDIT(*i, j*). Σε πρώτη φάση, αρχικοποιεί κατάλληλα την πρώτη (0^n) γραμμή και την πρώτη (0^n) στήλη, κάθε μία σε σταθερό χρόνο. Προκειμένου στη συνέχεια να υπολογίσει οποιαδήποτε άλλη είσοδο (κουτί) του πίνακα, αρκεί να γνωρίζει τις τιμές που βρίσκονται ακριβώς πάνω, ακριβώς αριστερά καθώς και διαγώνια προς τα πάνω από αυτήν. Με άλλα λόγια, για να υπολογισθεί η τιμή EDIT(*i, j*), χρειάζονται μόνο οι τιμές EDIT(*i, j-1*), EDIT(*i-1, j-1*) και EDIT(*i-1, j*) (σχήμα 4.3). Για το λόγο αυτό, αρκεί μία διαπέραση του πίνακα EDIT από πάνω προς τα κάτω, και από αριστερά προς τα δεξιά για τον πλήρη υπολογισμό του. Η τιμή της συνολικής απόστασης διαμορφώσεως (edit distance) προκύπτει από τον υπολογισμό της τιμής της θέσης EDIT(m, n).



Σχήμα 4.2: Ο πίνακας των υποπροβλημάτων. Για τον υπολογισμό της τιμής edit(i, j) απαιτούνται μόνο οι τιμές edit(i, j-1), edit(i-1, j) και edit(i-1, j-1).

Στο παράδειγμα του σχήματος 4.3 παρουσιάζεται ένα στιγμιότυπο του αλγορίθμου. Κάθε θέση *i, j* πρέπει να συμπληρωθεί με την μικρότερη από τις τιμές συν ένα του

'πάνω' *i-1, j* κουτιού (διαγραφή του x_i), και του 'αριστερού' κουτιού *i, j-1* (ένθεση του y_i) και είτε την τιμή του διαγώνιου κουτιού *i-1, j-1*, εάν $x_i=y_i$ (καμία πράξη), είτε την τιμή του διαγώνιου κουτιού *i-1, j-1* συν ένα (αντικατάσταση του x_i από το y_i). Για παράδειγμα, για τον υπολογισμό της τιμής edit(4,3), η οποία αντιστοιχεί στα προθέματα EXPO και POL, προκύπτει ότι:

edit(4,3) = min{edit(3,3)+1, edit(4,2)+1,edit(3,2)+1} =
= min {4, 3, 3} =
= 3

Μόλις συμπληρωθεί η κάτω δεξιά θέση του πίνακα, τότε έχουμε και την τιμή της ελάχιστης απόστασης διαμορφώσεως. Στο παράδειγμά μας, η απόσταση αυτή είναι ίση με **6**. Τα χρωματισμένα τετράγωνα υποδεικνύουν το μονοπάτι που πρέπει κανείς να ακολουθήσει προκειμένου να μετασχηματίσει πλήρως τη λέξη EXPONENTIAL σε POLYNOMIAL, εκτελώντας και τις ανάλογες πράξεις. Κάθε κίνηση προς τα κάτω ισοδυναμεί με τη διαγραφή ενός χαρακτήρα, κάθε κίνηση προς τα δεξιά με την ένθεση ενός χαρακτήρα, ενώ κάθε διαγώνια κίνηση ισοδυναμεί είτε με την αντικατάσταση ενός χαρακτήρα από έναν άλλον είτε με ένα ταίριασμα. Εδώ, η ακολουθία πράξεων που απαιτείται για τον μετασχηματισμό EXPONENTIAL→ POLYNOMIAL είναι η ακόλουθη:

διαγραφή(E), διαγραφή(X), =, =, αντικατάσταση(N), αντικατάσταση(E), =, ένθεση(O), αντικατάσταση(T), =, =, =.

		P	O	L	Y	N	O	M	I	A	L
	0→	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	2	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

↓ : Διαγραφή χαρακτήρα x

→ : Ένθεση χαρακτήρα x

↘ : Αντικατάσταση χαρακτήρα x από άλλο χαρακτήρα y ή καμία πράξη (-)

← Edit distance

Σχήμα 4.3: Πίνακας υπολογισμού του μετασχηματισμού φθηνότερου κόστους της λέξης 'EXPONENTIAL' σε 'POLYNOMIAL'. Τα χρωματισμένα τετράγωνα υποδεικνύουν το μονοπάτι που πρέπει κάποιος να ακολουθήσει για να μετασχηματίσει πλήρως τη λέξη εκτελώντας και τις ανάλογες πράξεις. Η ελάχιστη απόσταση διαμορφώσεως προκύπτει από το κάτω δεξιό τετραγωνάκι και είναι ίση με 6.

Ουσιαστικά, αυτό που κατορθώσαμε να δείξουμε χρησιμοποιώντας την τεχνική του δυναμικού προγραμματισμού, είναι ότι η βέλτιστη ολική λύση για την εύρεση της αποστάσεως δύο συμβολοσειρών μπορεί εύκολα να προκύψει από το συνδυασμό βέλτιστων λύσεων υποπροβλημάτων, τα οποία μάλιστα είναι πολυωνυμικά στο πλήθος και συγκεκριμένα, το πολύ $m \times n$. Προφανώς, υπάρχουν κι άλλοι διαφορετικοί τρόποι υπολογισμού της ελάχιστης απόστασης διαμορφώσεως μεταξύ δύο ακολουθιών. Ένας από αυτούς είναι και η χρήση του αλγορίθμου Dijkstra για την εύρεση των ελαχίστων μονοπατιών σε γράφους. Από τα παραπάνω εύκολα προκύπτει ότι:

Θεώρημα 4.1: *Τα προβλήματα της ελάχιστης απόστασης διαμορφώσεως και του βέλτιστου μετασχηματισμού μίας συμβολοσειράς x μήκους n σε μία άλλη y μήκους m μπορούν να λυθούν εντός χρόνου O(mn).*

4.3 Μέγιστη Κοινή Υπακολουθία (Longest Common Subsequence)

Στην ενότητα αυτή, θα περιγράψουμε άλλον έναν τρόπο για τη μέτρηση της ομοιότητας μεταξύ δύο συμβολοσειρών x, y. Το πρόβλημα της **μέγιστης κοινής υπακολουθίας (longest common subsequence problem)** έγκειται, όπως εύκολα γίνεται κατανοητό, στην εύρεση της μεγαλύτερης δυνατής υπακολουθίας μεταξύ δύο ή περισσοτέρων (συνήθως δύο) συμβολοσειρών οποιουδήποτε μήκους. Εάν $x=x_1x_2\ldots x_n$ είναι μία συμβολοσειρά, τότε κάθε συμβολοσειρά της μορφής

$$x' = x_{i1}x_{i2}\ldots x_{ik} \qquad \text{με} \qquad i_1 < i_2 < \ldots < i_k \leq n$$

καλείται **υπακολουθία[2] (subsequence),** με μήκος k, της x. Χρησιμοποιώντας έναν εναλλακτικό ορισμό, μπορούμε να πούμε ότι $w_0w_1\ldots w_{i-1}$ είναι μία υπακολουθία της $x=x_0x_1\ldots x_{n-1}$ εάν υπάρχει μία 'αυστηρά' αύξουσα ακολουθία από ακεραίους της μορφής $k_0k_1\ldots k_{i-1}$, τέτοια ώστε για $0 \leq k \leq i-1$, $w_j = x_{kj}$. Μία λέξη w αποτελεί την μέγιστη κοινή υπακολουθία δύο συμβολοσειρών x και y, με μήκη n και m αντίστοιχα, εάν η w είναι μία υπακολουθία για την x, μία υπακολουθία για την y και το μήκος της είναι το μέγιστο δυνατό που μπορεί να υπάρξει. Έστω, λοιπόν,

$$x = x_1x_2\ldots x_n \qquad \text{και} \qquad y = y_1y_2\ldots y_m$$

οι δύο συμβολοσειρές εισόδου. Η μέγιστη κοινή υπακολουθία τους (lcs(x, y)) ορίζεται ως:

$$w = \text{lcs}(x, y) = x_{i1}x_{i2}\ldots x_{ik} = y_{i1}y_{i2}\ldots y_{ik} \qquad \text{με} \begin{cases} 1 \leq i_1 < i_2 < \ldots < i_k \leq n \\ 1 \leq j_1 < j_2 < \ldots < j_k \leq m \end{cases}$$

[2] Η υπακολουθία δε θα πρέπει σε καμία περίπτωση να συγχέεται με την υποσυμβολοσειρά. Κάθε υποσυμβολοσειρά αποτελείται **μόνο** από συνεχόμενους χαρακτήρες της κύριας συμβολοσειράς, ενώ στην υπακολουθία αυτό δεν ισχύει απαραίτητα.

και το k έχει την μέγιστη δυνατή τιμή. Με άλλα λόγια δεν υπάρχει κοινή υπακολουθία μεγαλύτερου μήκους. Αξίζει να σημειωθεί ότι δύο συμβολοσειρές μπορούν να έχουν περισσότερες από μία μέγιστες κοινές υπακολουθίες. Για το λόγο αυτό, ορίζουμε ως Lcs(x,y) το το σύνολο των μέγιστων κοινών υπακολουθιών των x,y. Ας θεωρήσουμε ως παράδειγμα τις συμβολοσειρές x='abcd' και y='badc'. Οι συμβολοσειρές αυτές, έχουν τέσσερις μέγιστες κοινές υπακολουθίες, τις 'ac', 'ad', 'bc', 'bd', έκαστη μήκους 2. Στην γενικότερη περίπτωση, όπου έχουμε έναν αυθαίρετο αριθμό από συμβολοσειρές εισόδου, το πρόβλημα της μέγιστης κοινής υπακολουθίας είναι NP-Hard. Όταν ο αριθμός των ακολουθιών είναι σταθερός, τότε το πρόβλημα μπορεί να λυθεί σε πολυωνυμικό χρόνο, χρησιμοποιώντας την τεχνική του δυναμικού προγραμματισμού. Μία απλοϊκή λύση θα απαιτούσε εκθετική πολυπλοκότητα $O(max(n, m)2^{max(n, m)})$ προκειμένου να δημιουργήσει και να συγκρίνει όλες τις υπακολουθίες για όλα τα δυνατά μήκη. Αντίθετα, η προσέγγιση του δυναμικού προγραμματισμού, απαιτεί $O(nm)$ χώρο και χρόνο για την εύρεση της μέγιστης κοινής υπακολουθίας δύο συμβολοσειρών x, y με μήκη n και m αντίστοιχα. Για το λόγο αυτό, κι εμείς, θα προσπαθήσουμε να αναπαράγουμε τη βέλτιστη ολική λύση από ένα συνδυασμό πολυωνυμικών, το πολύ, στο πλήθος, βέλτιστων λύσεων υποπροβλημάτων.

4.3.1 Υπολογισμός Μ.Κ.Υ. με Χρήση Δυναμικού Προγραμματισμού

Έστω, $x = x_1x_2\ldots x_n$ και $y = y_1y_2\ldots y_m$ οι δύο συμβολοσειρές εισόδου. Χρησιμοποιώντας την τεχνική του δυναμικού προγραμματισμού θα προσπαθήσουμε να υπολογίσουμε τα μήκη όλων των κοινών υπακολουθιών και κατ' επέκταση την μέγιστη κοινή υπακολουθία των x, y. Για το λόγο αυτό, θα χρησιμοποιήσουμε έναν δισδιάστατο πίνακα c, μεγέθους $(n+1)\times(m+1)$, ο οποίος ορίζεται ως ακολούθως:

- $c[i,0] = c[0,j] = 0$ για $0 \le i \le n$ και $0 \le j \le n$
- $c[i,j] = lcs(x_0x_1\cdots x_i, y_0y_1\cdots y_j)$ για $0 \le i \le n$ και $0 \le j \le m$.

Η τιμή της μέγιστης κοινής υπακολουθίας των x, y, lcs(x, y), θα ισούται με την τιμή c[n, m] (lcs(x, y) = c[n, m]), θεωρώντας ότι ισχύει ο ακόλουθος αναδρομικός τύπος:

$$c[i, j] = \begin{cases} 0 & i=0 \text{ ή } j=0, \\ c[i-1, j-1]+1 & i,j > 0 \text{ και } x_i=y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{διαφορετικά.} \end{cases} \quad (4.2)$$

Θεώρημα 4.2: *Έστω $x = x_1x_2\ldots x_n$ και $y = y_1y_2\ldots y_m$ οι δύο συμβολοσειρές εισόδου, $z \in Lcs(x_0x_1\cdots x_{i-1}, y_0y_1\cdots y_{j-1})$, $z' \in Lcs(x_0x_1\cdots x_{i-1}, y_0y_1\cdots y_j)$ και $z'' \in Lcs(x_0x_1\cdots x_i, y_0y_1\cdots y_{j-1})$. Τότε, για $0 \le i \le n$ και $0 \le j \le m$, για την μέγιστη κοινή υπακολουθία w ορίζονται οι ακόλουθες σχέσεις:*

$$w = \begin{cases} zx_i & \text{εάν } x_i = y_j \\ z' & \text{εάν } x_i \neq y_j \text{ και } c[i-1, j] \geq c[i, j-1] \\ z'' & \text{εάν } x_i \neq y_j \text{ και } c[i-1, j] < c[i, j-1] \end{cases}$$

Απόδειξη. Έστω z ϵ Lcs($x_0x_1\cdots x_{i-1}$, $y_0y_1\cdots y_{j-1}$), z' ϵ Lcs($x_0x_1\cdots x_{i-1}$, $y_0y_1\cdots y_j$) και z'' ϵ Lcs($x_0x_1\cdots x_i$, $y_0y_1\cdots y_{j-1}$). Με άλλα λόγια, το z είναι μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_{i-1}$ και $y_0y_1\cdots y_{j-1}$, το z' μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_{i-1}$ και $y_0y_1\cdots y_j$ και το z'' μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_i$ και $y_0y_1\cdots y_{j-1}$. Ισχύουν τα εξής:

- |z| = c[i-1,j-1]
- |z'| = c[i-1,j]
- |z''| = c[i,j-1]

Τότε, προφανώς, εάν $x_i = y_j$, ισχύει ότι c[i, j]=1+c[i-1, j-1] και w = zx_i είναι μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_i$ και $y_0y_1\cdots y_j$.

Εάν $x_i \neq y_j$, τότε διακρίνουμε τις ακόλουθες δύο περιπτώσεις:

- c[i-1, j] ≥ c[i, j-1], οπότε c[i, j] = c[i-1, j] και το z', το οποίο είναι μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_{i-1}$ και $y_0y_1\cdots y_j$ με μήκος c[i-1, j], είναι επίσης και μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_i$ και $y_0y_1\cdots y_j$ μήκους c[i, j]

- c[i-1, j] < c[i, j-1], οπότε c[i, j] = c[i, j-1] και το z'', το οποίο είναι μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_i$ και $y_0y_1\cdots y_{j-1}$ με μήκος c[i, j-1], αποτελεί επίσης και μία μέγιστη κοινή υπακολουθία των $x_0x_1\cdots x_i$ και $y_0y_1\cdots y_j$ μήκους c[i, j]. □

Ο αλγόριθμος LongestCommonSubsequence (αλγ. 4.2), χρησιμοποιεί τον αναδρομικό τύπο 4.2 για τον υπολογισμό του πίνακα c.

```
Αλγόριθμος: LongestCommonSubsequence (char[] x, char[] y)
Είσοδος: Οι ακολουθίες x, y
Έξοδος: Ο πίνακας c

1.      |x|=n;   |y|=m;  int c[];
2.      for i := 0 to n do c[i, 0] := 0;
3.      for j := 0 to m  do c[0, j] := 0;
4.      for i := 1 to n do
5.              for j := 1 to m  do
6.                      if xi = yj
7.                          then   c[i, j] = c[i-1, j-1]+1
8.                          else   c[i, j] = max(c[i, j-1],c[i-1, j]);
9.      return c;
```

Αλγόριθμος 4.2: Υπολογισμός του πίνακα τιμών κοινών υπακολουθιών των x,y

Προκειμένου μάλιστα, να αναπαραχθεί η μέγιστη κοινή υπακολουθία, η βοηθητική συνάρτηση PrintLongestCommonSubsequence() (αλγ. 4.3), οπισθοδρομεί αναδρομικά από την θέση c[n+1,m+1] στην c[0,0], τυπώνοντας συγχρόνως κάθε χαρακτήρα της.

Αλγόριθμος: PrintLongestCommonSubsequence (**char**[] x, **char**[] y, **int**[] c)
Είσοδος: Οι ακολουθίες x, y
Έξοδος: Η μέγιστη κοινή υπακολουθία των x, y

```
1.      |x|=n;   |y|=m;
2.      k = c[n, m]-1;
3.      while i > 0 and j > 0
4.              do if c[i, j] = c[i-1, j-1]+1;
5.                      then    wk = xi;
6.                              i = i-1;
7.                              j = j-1;
8.                              k = k-1;
9.              else if  c[i-1, j] = c[i, j-1]
10.                     then    i=i-1;
11.             else            j=j-1;
12.     return w;
```

Αλγόριθμος 4.3: Εύρεση μέγιστης κοινής υπακολουθίας

Παράδειγμα τρεξίματος του αλγορίθμου αποτελεί το σχήμα 4.4. Με ↖ , ↑ ,← δηλώνονται, αντίστοιχα, το ταίριασμα, η απόρριψη χαρακτήρα της συμβολοσειράς x και η απόρριψη χαρακτήρα της συμβολοσειράς y. Αφού αρχικοποιηθούν η πρώτη γραμμή και η πρώτη στήλη, στη συνέχεια σαρώνουμε την πίνακα από πάνω προς τα κάτω και από αριστερά προς τα δεξιά. Έστω ότι πρέπει να συμπληρώσουμε τη θέση i,j. Εάν ισχύει $x_i=y_j$, τότε, η τιμή προκύπτει από το περιεχόμενο της διαγώνιας προς τα πάνω θέσης συν 1, ενώ παράλληλα «κρατείται» και ο κοινός χαρακτήρας x_i,y_j. Διαφορετικά, συγκρίνεται το περιεχόμενο της πάνω θέσεως i-1,j με αυτό της αμέσως αριστεράς i,j-1 και προτιμάται το πρώτο, εάν είναι μεγαλύτερο ή ίσο από το δεύτερο, αλλιώς το δεύτερο. Όταν συμπληρωθεί η κάτω δεξιά θέση του πίνακα, τότε έχουμε, αφ' ενός την τιμή, αφ' ετέρου, ακολουθώντας το μονοπάτι που υποδεικνύουν τα βέλη, αναπαραγωγή της λύσεως.

	j	0	1	2	3	4	5	6	7	8	9	10
i	x \ y		P	O	L	Y	N	O	M	I	A	L
0		0	0	0	0	0	0	0	0	0	0	0
1	E	↑0	0	0	0	0	0	0	0	0	0	0
2	X	↑0	0	0	0	0	0	0	0	0	0	0
3	P	0	↖1	1	1	1	1	1	1	1	1	1
4	O	0	1	↖2	←2	←2	2	2	2	2	2	2
5	N	0	1	2	2	2	↖3	←3	←3	3	3	3
6	E	0	1	2	2	2	3	3	↑3	3	3	3
7	N	0	1	2	2	2	3	3	↑3	3	3	3
8	T	0	1	2	2	2	3	3	↑3	3	3	3
9	I	0	1	2	2	2	3	3	3	↖4	4	4
10	A	0	1	2	2	2	3	3	3	4	↖4	5
11	L	0	1	2	3	3	3	3	3	4	5	↖6 ← lcs(x,y)

↑ : Απόρριψη χαρακτήρα της x

← : Απόρριψη χαρακτήρα της y

↖: Ταίριασμα χαρακτήρων

Σχήμα 4.4: Στιγμιότυπο υπολογισμού μέγιστης κοινής υπακολουθίας των λέξεων 'EXPONENTIAL' και 'POLYNOMIAL'. Τα χρωματισμένα τετράγωνα υποδεικνύουν τη διαδρομή που πρέπει κάποιος να ακολουθήσει για να αναπαράγει πλήρως την υπακολουθία εκτελώντας και τις ανάλογες πράξεις. Η τιμή της μέγιστης κοινής υπακολουθίας προκύπτει από το κάτω δεξιό τετραγωνάκι και είναι ίση με 6.

Τέλος, από τα παραπάνω, εύκολα συμπεραίνουμε πως:

Θεώρημα 4.3: *Ο υπολογισμός μίας μέγιστης κοινής υπακολουθίας δύο συμβολοσειρών x, y με μήκη n και m αντίστοιχα, απαιτεί O(nm) χωρική και χρονική πολυπλοκότητα.*

4.4 Ταίριασμα Συμβολοσειράς με το πολύ *k* Λάθη

Το πρόβλημα του προσεγγιστικού ταιριάσματος συμβολοσειράς με λάθη έγκειται στην εύρεση όλων των εμφανίσεων ενός προτύπου P σ' ένα κείμενο T, επιτρέποντας, ταυτόχρονα, την ύπαρξη διαφορετικών χαρακτήρων στις αντίστοιχες συγκρινόμενες θέσεις των δύο συμβολοσειρών. Μάλιστα, όταν ο αριθμός των διαφορετικών αυτών χαρακτήρων φράσσεται από έναν δεδομένο σταθερό αριθμό *k*, τότε έχουμε το προσεγγιστικό ταίριασμα συμβολοσειράς με το πολύ *k* λάθη. Το συγκεκριμένο πρόβλημα ορίζεται ως εξής:

Ορισμός 4.1: *Δεδομένου ενός προτύπου P, μήκους m, ενός κειμένου T, μήκους n, κι ενός θετικού ακεραίου k, με $0 \leq k < m$, ζητούμε να εντοπίσουμε όλες τις εμφανίσεις του P στο T, έτσι ώστε το πολύ σε k θέσεις το κείμενο και το πρότυπο να έχουν διαφορετικούς χαρακτήρες.*

Για τον μέγιστο αριθμό επιτρεπόμενων λαθών *k*, όπως διατυπώνεται και στον παραπάνω ορισμό, ισχύει ότι $0 \leq k < m$ ($m \leq n$). Η περίπτωση όπου $k = 0$ αντιστοιχεί στο γνωστό, από το προηγούμενο κεφάλαιο, πρόβλημα του ακριβούς ταιριάσματος συμβολοσειράς (exact string matching).

Αξίζει να σημειωθεί, επίσης, ότι το προσεγγιστικό ταίριασμα με λάθη συνδέεται στενά με την ελάχιστη απόσταση διαμορφώσεως (Edit Distance), την οποία μελετήσαμε στην ενότητα 4.1. Στον ακόλουθο εναλλακτικό ορισμό διατυπώνεται η στενή σχέση μεταξύ αυτών των δύο προβλημάτων.

Ορισμός 4.2: *Δεδομένου ενός προτύπου P, μήκους m, ενός κειμένου T, μήκους n, κι ενός θετικού ακεραίου k, με $0 \leq k < m$, ζητούμε να εντοπίσουμε κάθε υποσυμβολοσειρά του P στο T, για την οποία ισχύει ότι η ελάχιστη απόσταση διαμορφώσεως ανάμεσα σε αυτήν και στο πρότυπο είναι ίση ή μικρότερη από την τιμή k.*

Ως ελάχιστη απόσταση διαμορφώσεως (edit distance), ορίσαμε τον ελάχιστο αριθμό από πράξεις οι οποίες χρειάζεται να εκτελεστούν προκειμένου μία συμβολοσειρά x να μετατραπεί σε μία άλλη y. Οι πράξεις αυτές, όπως έχουμε δει, μπορεί να είναι είτε ενθέσεις είτε διαγραφές είτε αντικαταστάσεις χαρακτήρων. Με βάση, λοιπόν, τον παραπάνω ορισμό, το πρόβλημα του προσεγγιστικού ταιριάσματος με λάθη, έγκειται στην εύρεση κάθε υποσυμβολοσειράς t_j ($1 \leq j \leq n$) του κειμένου, για την οποία ισχύει ότι $edit(t_j, P) \leq k$. Με άλλα λόγια, αν ο ελάχιστος αριθμός πράξεων, που πρέπει να γίνουν προκειμένου το πρότυπο να ταυτιστεί με κάποια υποσυμβολοσειρά του κειμένου η οποία τελειώνει στη θέση t_j, είναι μικρότερος ή ίσος του *k*, τότε λέμε ότι το πρότυπο εμφανίζεται στη θέση j του κειμένου με το πολύ *k* λάθη.

4.4.1 Απλοϊκή Λύση

Ο απλοϊκός αλγόριθμος επίλυσης του προβλήματος, περιλαμβάνει την αντιπαραβολή του προτύπου P με το κείμενο T και τον υπολογισμό του πλήθους των αποτυχημένων ταιριασμάτων, ξεκινώντας από όλες τις δυνατές θέσεις του T. Για κάθε τοποθέτηση του προτύπου, εάν το πλήθος αυτό υπερβεί τον αριθμό k (μέγιστος επιτρεπόμενος αριθμός λαθών), τότε το πρότυπο ολισθαίνει κατά μία θέση δεξιότερα και η διαδικασία επαναλαμβάνεται με τον ίδιο τρόπο. Διαφορετικά, αν το πλήθος των ανεπιτυχών συγκρίσεων είναι μικρότερο ή ίσο με την τιμή *k*, τότε έχουμε εντοπίσει μία εμφάνιση του προτύπου P στο κείμενο T με το πολύ *k* λάθη. Η απλοϊκή αυτή λύση, δεδομένου ενός κειμένου n χαρακτήρων και ενός προτύπου μήκους m, απαιτεί $O(mn)$ χρόνο στην χειρότερη περίπτωση και $O(kn)$ στη μέση. Επίσης, στην χειρότερη περίπτωση το πλήθος των συγκρίσεων που εκτελούνται είναι $m \times (n-m+1)$.

4.4.2 Λύση με Χρήση Δυναμικού Προγραμματισμού

Χρησιμοποιώντας την τεχνική του δυναμικού προγραμματισμού, το πρόβλημα του προσεγγιστικού ταιριάσματος με λάθη, μπορεί να λυθεί σε χρόνο $O(mn)$. Δεδομένου ενός δισδιάστατου πίνακα, c, μεγέθους $(n+1) \times (m+1)$, έστω $c(i, j)$ ($0 \leq i \leq m$ και $0 \leq j \leq n$) ο ελάχιστος αριθμός λαθών/διαφορών μεταξύ της υποσυμβολοσειράς $x_1x_2 \ldots x_i$ του προτύπου και της υποσυμβολοσειράς του κειμένου η οποία τελειώνει στον χαρακτήρα y_j. Τότε, η τιμή $c(i, j)$ ορίζεται με βάση την ακόλουθη σχέση:

$$c[i, j] = \min\{c[i-1, j-1] + p, c[i, j-1] + 1, c[i-1, j] + 1\},$$

όπου p=0 εάν $x_i = y_j$ και p=1 σε διαφορετική περίπτωση. Επίσης, ο πίνακας c αρχικοποιείται κατάλληλα έτσι ώστε c(0, j) = 0 και c(i, 0) = i για όλα τα i, j.

Ο αλγόριθμος 4.4 αποτελεί μία ενδεικτική υλοποίηση του προσεγγιστικού ταιριάσματος συμβολοσειράς με το πολύ *k* λάθη, χρησιμοποιώντας την τεχνική του δυναμικού προγραμματισμού. Εδώ, ως λάθος μπορεί να θεωρηθεί μία από τις παρακάτω τρεις περιπτώσεις:

- Ένας χαρακτήρας του προτύπου αντιστοιχεί σε διαφορετικό χαρακτήρα στο κείμενο.
- Ένας χαρακτήρας του προτύπου δεν αντιστοιχεί με κάποιον χαρακτήρα στο κείμενο (κενός χαρακτήρας, '-').
- Ένας χαρακτήρας του κειμένου δεν αντιστοιχεί με κάποιον χαρακτήρα στο πρότυπο (κενός χαρακτήρας, '-').

Αλγόριθμος: k-errors(**char**[] x, **char**[] y, **int** *k*)
Είσοδος: Οι ακολουθίες x, y και ο μέγιστος αριθμός επιτρεπόμενων λαθών *k*
Έξοδος: Όλες οι υποσυμβολοσειρές της x που διαφέρουν το πολύ κατά *k* από την y

```
1.      |x|=n;   |y|=m;  int c[];
2.      for j := 0 to m do c[0, j+1] := 0;
3.      for i := 0 to n  do c[i+1, 0] := 0;
4.      for j := 1 to m+1 do
5.              for i := 1 to n+1  do
6.                      if xi = yj
7.                          then   p=0;
8.                          else   p=1;
9.                      c[i, j] = min{c[i-1, j-1]+p, c[i, j-1]+1, c[i-1, j]+1};
10.             if  c[m, j] ≤ k
11.                     then return j;
```

Αλγόριθμος 4.4: Προσεγγιστικό ταίριασμα συμβολοσειράς με το πολύ *k* λάθη

Η αρχικοποίηση όλων των τιμών της πρώτης σειράς του πίνακα σε 0 (γραμμή 2), προέρχεται από το γεγονός ότι το κόστος των εισαγωγών χαρακτήρων της y στην αρχή της x είναι μηδενικό. Οι λύσεις του προβλήματος δίνονται από όλες τις τιμές της τελευταίας σειράς του c οι οποίες είναι ίσες ή μικρότερες της τιμής *k*.

Στο σχήμα 4.5 παρουσιάζεται ένα παράδειγμα υπολογισμού του πίνακα c και εντοπισμού όλων των εμφανίσεων της συμβολοσειράς "GATAA" στο κείμενο "CAGATAAGAGAA" με μέγιστο αριθμό επιτρεπόμενων λαθών *k*=1.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12
i	y / x		C	A	G	A	T	A	A	G	A	G	A	A
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	1	1	1	0	1	1	1	1	0	1	0	1	1
2	A	2	2	1	1	0	1	1	1	1	0	1	0	1
3	T	3	3	2	2	1	0	1	2	2	1	1	1	1
4	A	4	4	3	3	2	1	0	1	2	2	2	1	1
5	A	5	5	4	4	3	2	1	0	1	2	3	2	1

Σχήμα 4.5: Στιγμιότυπο εντοπισμού όλων των εμφανίσεων της συμβολοσειράς "GATAA" στην "CAGATAAGAGAA". Τα χρωματισμένα τετράγωνα υποδεικνύουν τις λύσεις του προβλήματος για τις οποίες ισχύει ότι $k \leq 1$.

Οι λύσεις του παραδείγματος αυτού, αντιστοιχούν στις ακόλουθες επτά ευθυγραμμίσεις/στοιχίσεις μεταξύ των δύο συμβολοσειρών:

```
      G A T A A
C A G A T - A A G A G A A
-------------------------
      G A T A A
C A G A T A - A G A G A A
-------------------------
      G A T A A
C A G A T A A G A G A A
-------------------------
  - G A T A A
C A G A T A A G A G A A
-------------------------
      G A T A A
C A G - A T A A G A G A A
-------------------------
      G A T A A -
C A G A T A A G A G A A
-------------------------
                G A T A A
C A G A T A A G A G A A
-------------------------
```

4.4.3 Ο αλγόριθμος των Landau και Vishkin

Ενώ ο απλός αλγόριθμος του δυναμικού προγραμματισμού χρειάζεται O(*mn*) χρόνο για την επίλυση του ταιριάσματος συμβολοσειράς με k λάθη, η μέθοδος που πρότειναν οι Landau και Vishkin αιτεί O(*(k+logm)n*) χρονική πολυπλοκότητα για ένα γενικό αλφάβητο. Η μέθοδος αυτή, υπολογίζει τον ίδια πληροφορία c(i, j), όπως και ο απλός αλγόριθμος δυναμικού προγραμματισμού που εξετάσαμε προηγουμένως, χρησιμοποιώντας, όμως, τις διαγώνιους του πίνακα c. Μία διαγώνιος d του πίνακα c αποτελείται από όλες εκείνες τις τιμές c(i, j) για τις οποίες ισχύει ότι j-i=d.

Για έναν δεδομένο αριθμό λαθών e και μία διαγώνιο d, ως L(d,e) ορίζεται η μεγαλύτερη σειρά i του πίνακα c, τέτοια ώστε c(i, j) = e και j-i = d. Στην περίπτωση του πίνακα του σχήματος 4.6, προκύπτει ότι L(3,0) = 0, L(3,1) = 1 και L(3,2) = 4. Αξίζει να σημειωθεί ότι η τιμή c(i,j) για j-i=d, αυξάνεται μονοτονικά καθώς μεγαλώνει το i. Ως εκ τούτου, για το ταίριασμα συμβολοσειράς με k λάθη, αρκεί να υπολογίσουμε μόνο τις τιμές L(d,e) για τις οποίες e ≤ k. Εφόσον το πλήθος των διαγωνίων είναι O(*n*), ο αριθμός των τιμών L(d,e) που χρειάζεται να υπολογιστούν είναι O(*kn*).

		0	1	2	3	4	5	6	7
	╲		b	c	c	a	b	a	d
0		0	0	0	0	0	0	0	0
1	c	1	1	0	0	1	1	1	1
2	a	2	2	1	1	0	1	1	2
3	a	3	3	2	2	1	1	1	2
4	b	4	3	3	3	2	1	2	2

Σχήμα 4.6: Παράδειγμα ενός πίνακα c για P=”caab” και T=”bccabad”

Ο αλγόριθμος 4.5 αποτελεί την υλοποίηση της μεθόδου των Landau και Vishkin.

```
Procedure LandauVishkin(P,T)

1.  begin
2.   for all d such that 0 ≤ d ≤ n  do L(d,-1)←-1;
3.   for all d such that –(k+1) ≤ d ≤ -1  do
4.       begin
5.          L(d,|d|-1)←|d|-1;     L(d,|d|-2)←|d|-2;
6.       end;
7.   for all e such that -1 ≤ e ≤ k  do L(n+1,e)←-1;
8.   for e = 0 to k do
9.       for all d such that -e ≤ d ≤ n do
10.         begin
11.              row←max(L(d,e-1)+1, L(d-1,e-1), L(d+1,e-1)+1);
12.              row←min(row,m);
```

```
13.            while row < m  and  row+d < n  and  p_{row+1}=t_{row+1+d} do
14.               row←row+1;
15.            L(d,e)←row;
16.            if L(d,e) = m then output "There is an occurrence ending at t_{d+m}"
17.         end
18.  end
```

Αλγόριθμος 4.5: Ενδεικτική υλοποίηση του αλγόριθμου των Landau και Vishkin

Εάν ο παραπάνω αλγόριθμος υλοποιηθεί ως έχει, χρειάζεται O(*mn*) χρόνο για την επίλυση του προβλήματος. Ωστόσο οι Landau και Vishkin απέδειξαν ότι η γραμμή 13 του κώδικα μπορεί να υπολογιστεί σε σταθερό O(*1*) χρόνο, εάν το προσφυματικό δένδρο (suffix tree) το οποίο σχετίζεται με τον όρο T · P · γραμμή 13, έχει κατασκευαστεί εκ των προτέρων (με $s_1 \cdot s_2$ ορίζεται η αλυσίδα/αλληλουχία των s_1,s_2). Συνεπώς, η διαδικασία LandauVishkin(P,T) "τρέχει" σε O(*kn*) χρόνο, ενώ το πρόβλημα του προσεγγιστικού ταιριάσματος με k λάθη μπορεί να λυθεί σε O(*(k+logm)n*) χρόνο για ένα γενικό αλφάβητο συμπεριλαμβάνοντας και την κατασκευή του προσφυματικού δέντρου.

4.5 Ταίριασμα Συμβολοσειράς με Προσφυματικά Δένδρα

Όλοι οι αλγόριθμοι που εξετάσαμε μέχρι τώρα προεπεξεργάζονται το πρότυπο P και, κατόπιν, χρησιμοποιούν την εξαγόμενη πληροφορία, κατά την διαδικασία ταιριάσματος επί του κειμένου T. Η αντίστροφη διαδικασία, δηλαδή η προεπεξεργασία του T, ώστε να είναι δυνατά ακόλουθα ψαξίματα, υλοποιείται με την μορφή των **προσφυματικών[3] δένδρων (suffix trees)** και εφαρμόζεται, όταν αναμένονται πολλαπλές ερωτήσεις επί του ιδίου κειμένου. Στη συνέχεια, θα περιγράψουμε τα προσφυματικά δένδρα, καθώς, και το πώς αυτά μπορούν να χρησιμοποιηθούν για την επίλυση του προβλήματος του προσεγγιστικού ταιριάσματος συμβολοσειράς.

4.5.1 Προσφυματικά Δένδρα

Τα προσφυματικά δένδρα αποτελούν μία δομή δεδομένων η οποία αναπαριστά όλα τα προσφύματα (suffixes) μίας δοσμένης συμβολοσειράς/ακολουθίας/κειμένου με τέτοιον τρόπο, ώστε να τα καθιστά ιδιαίτερα ευέλικτα και αποδοτικά στην υλοποίηση αρκετών βασικών προβλημάτων τα οποία σχετίζονται με συμβολοσειρές. Τα δένδρα αυτά ορίζονται βάσει της γνωστής δομής κύριας μνήμης **Trie**. Για το λόγο αυτό, καθίσταται αναγκαία η παρουσίαση και ανάλυση της δομής αυτής, προκειμένου να κατανοήσουμε πλήρως την λειτουργία των προσφυματικών δένδρων. Στη συνέχεια, δίνουμε τον ορισμό των δένδρων trie, αρχικά, στην δυαδική τους μορφή, και ακολούθως σε μία πιο γενικευμένη.

[3] Στην ελληνική βιβλιογραφία τα suffix trees συναντώνται είτε ως προφυματικά είτε ως επιθεματικά δένδρα. Εδώ, χρησιμοποιούμε την πρώτη ονομασία.

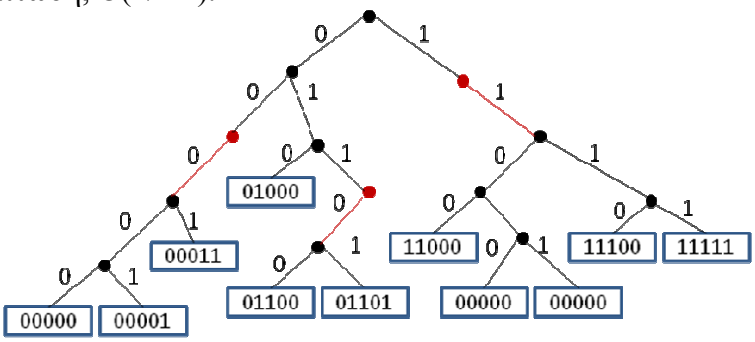
Έστω S ένα σύνολο διακριτών δυαδικών συμβολοσειρών μήκους l. Ένα δυαδικό δένδρο trie T για το σύνολο αυτό ορίζεται ως εξής: Εάν το S είναι το κενό σύνολο, τότε και το δένδρο είναι κενό. Εάν το σύνολο αποτελείται από ένα μόνο στοιχείο σ, τότε το T θα αποτελείται από έναν κόμβο που αντιστοιχεί στο σ. Διαφορετικά, το T αποτελείται από έναν εσωτερικό κόμβο, ο οποίος θα έχει, ως αριστερό υποδένδρο, ένα trie για τα στοιχεία που το πρώτο τους στοιχείο είναι το '0' και, ως δεξιό υποδένδρο, ένα trie που τα στοιχεία του έχουν ως πρώτο στοιχείο το '1'. Και τα δύο υποδένδρα trie ορίζονται για τις αντίστοιχες συμβολοσειρές, αφού τους αφαιρεθεί το πρώτο ψηφίο (και άρα έχουν μήκος l-1).

Επιπρόσθετα, κάθε δομή Trie επιδεικνύει τις ακόλουθες ιδιότητες:

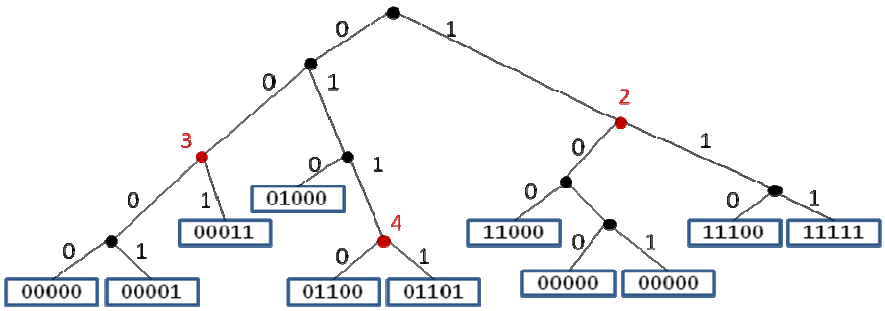
- Υπάρχει ένα μοναδικό Trie για κάθε σύνολο συμβολοσειρών, ανεξαρτήτως της σειράς με την οποία γίνονται οι ενθέσεις. Επιπλέον, τα στοιχεία είναι διατεταγμένα με αύξουσα σειρά, από αριστερά προς τα δεξιά.

- Μία ένθεση, στην χειρότερη περίπτωση, παίρνει χρόνο O(l). Στην μέση περίπτωση χρειάζεται O(logn) χρόνος.

- Στην χειρότερη περίπτωση, n τυχαία στοιχεία χρειάζονται χώρο O(nl), ενώ στην μέση περίπτωση, O(n/ln2).



(α)



(β)

Σχήμα 4.7: (α) Στιγμιότυπο δυαδικού δένδρου Trie, και **(β)** Στιγμιότυπο συμπιεσμένου δυαδικού δένδρου Trie

Το σημαντικότερο, ωστόσο, πρόβλημα των δομών tries είναι οι μεγάλες τους απαιτήσεις σε χώρο. Το μειονέκτημα αυτό, το επιλύουν τα **συμπιεσμένα Trie (compressed Tries)**. Οι δομές αυτές προκύπτουν, εάν στα απλά δένδρα Trie,

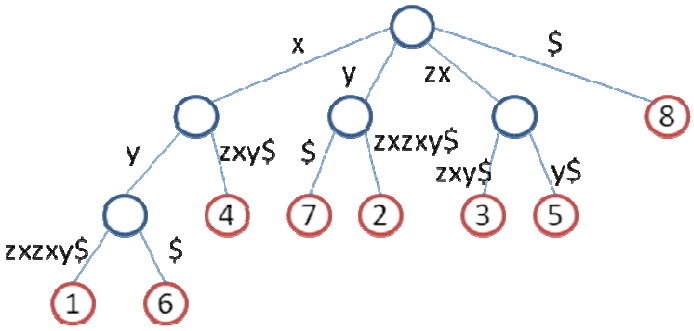
αφαιρεθούν οι κόμβοι βαθμού 2, και στους εναπομείναντες κόμβους σημανθεί ο αριθμός του ψηφίου, βάση του οποίου θα γίνει η διακλάδωση. Το Trie του σχήματος 4.7(β) αποτελεί την συμπιεσμένη εκδοχή του αντίστοιχου του σχήματος 4.7(α).

Λήμμα 4.1: *Ο χώρος που απαιτείται από ένα συμπιεσμένο Trie για την αποθήκευση n συμβολοσειρών είναι O(n).*

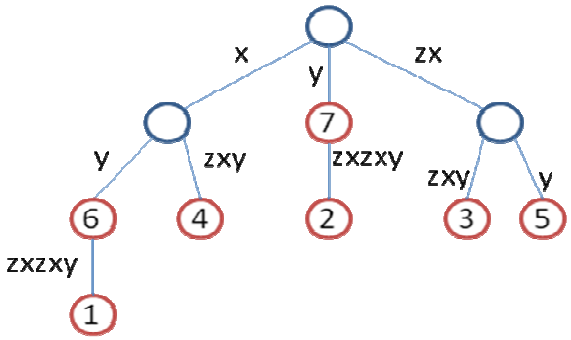
Στην γενικευμένη του μορφή, εάν το αλφάβητο Σ αποτελείται από $|\Sigma| > 2$ χαρακτήρες, τότε, αντί για δυαδικό δένδρο, χρησιμοποιείται $|\Sigma|$-δικό, με κάθε εξερχόμενη ακμή να αντιστοιχείται σε ακριβώς έναν χαρακτήρα.

Με βάση τα παραπάνω, μπορούμε να προχωρήσουμε στον ορισμό των προσφυματικών δένδρων. Έτσι λοιπόν:

Ορισμός 4.3: *Ένα προσφυματικό δένδρο επί κειμένου T, ορίζεται ως το συμπιεσμένο Trie επί όλων των δυνατών προθεμάτων του κειμένου T$.*



(α)



(β)

Σχήμα 4.8: (α) Στιγμιότυπο προσφυματικού δένδρου, και **(β)** το ίδιο δένδρο χωρίς την παρουσία του ειδικού συμβόλου $

Ο χαρακτήρας $ δηλώνει ένα σύμβολο το οποίο δεν ανήκει στο αλφάβητο του Τ. Η προσθήκη του εγγυάται πως όλα τα προσφύματα είναι διακεκριμένα και, συνεπώς, υπάρχει μοναδικό μονοπάτι στο δένδρο. Για παράδειγμα, ας θεωρήσουμε το προσφυματικό δένδρο του σχήματος 4.8(α) για την συμβολοσειρά κειμένου xyzxzxy$. Εάν η συμβολοσειρά αυτή δεν περιείχε τον τελικό χαρακτήρα $, το δένδρο θα παρέμενε σε γενικές γραμμές το ίδιο (σχήμα 4.8(β)). Ωστόσο, αν το παρατηρήσουμε καλά, θα διαπιστώσουμε πως το πρόσφυμα 'xy' δεν τελειώνει σε κάποιο φύλλο, γεγονός που έρχεται σε αντίθεση με τον ορισμό των προσφυματικών δένδρων. Το πρόβλημα έγκειται στο γεγονός ότι το πρόσφυμα 'xy' αποτελεί επίσης κι ένα πρόθεμα της συμβολοσειράς. Αυτό μπορεί να αποφευχθεί εισάγοντας στο τέλος κάθε κειμένου έναν ειδικό χαρακτήρα ο οποίος δεν εμφανίζεται πουθενά αλλού, μιας και τότε κανένα πρόσφυμα δεν μπορεί συγχρόνως να είναι και πρόθεμα (εκτός από την ίδια την συμβολοσειρά). Για το λόγο αυτό, θεωρούμε πως κάθε συμβολοσειρά κειμένου τελειώνει με το ειδικό σύμβολο $. Κατά αυτόν τον τρόπο, ενώ υπάρχουν O(n) προσφύματα σε ένα κείμενο μήκους n, ο απαιτούμενος χώρος του συμπιεσμένου Trie είναι O(n) (!)

Στη συνέχεια θα περιγράψουμε έναν απλό τρόπο κατασκευής των προσφυματικών δένδρων ο οποίος αιτεί O(n^2) πολυπλοκότητα. Στόχος του αλγόριθμου αυτού, είναι το σταδιακό χτίσιμο του προσφυματικού δένδρου για μία συμβολοσειρά S[1…n], εκτελώντας O(n) διαδοχικές ενθέσεις προσφυμάτων.

Έστω, $T_1,T_2,\ldots,T_n$ τα ενδιάμεσα δένδρα που προκύπτουν στα n στάδια κατασκευής του αλγορίθμου. Αρχικά, το δένδρο T_1 αποτελείται από μία ακμή με ετικέτα S[1…n], ενώ ο κόμβος στον οποίον αυτή καταλήγει έχει ετικέτα 1. Στο στάδιο i, επεξεργαζόμαστε το δένδρο T_i, έτσι ώστε το πρόσφυμα S[i…n] να μπορεί να χειριστεί κατάλληλα. Για να γίνει αυτό, ξεκινάμε από την ρίζα και ακολουθούμε το μονοπάτι κατά μήκος του δένδρου, ταιριάζοντας συγχρόνως χαρακτήρες του προσφύματος S[i…n]. Αυτή η διαδικασία απαιτεί μονάχα διαδοχικές συγκρίσεις χαρακτήρων, ενώ το μονοπάτι που ακολουθείται είναι σίγουρα μοναδικό, μιας και δεν μπορούν να υπάρχουν δύο διαφορετικές ακμές που να εξέρχονται από έναν κόμβο και να έχουν ως ετικέτα συμβολοσειρά που να αρχίζει με τον ίδιο χαρακτήρα. Κάποια στιγμή, προφανώς, θα φτάσουμε σε κάποιο σημείο όπου δε θα είναι εφικτά επιπλέον ταιριάσματα χαρακτήρων. Αυτό, βέβαια, δεν μπορεί να συμβεί σε κάποιο κόμβο φύλλου, αφού η συμβολοσειρά κειμένου τελειώνει με τον ειδικό χαρακτήρα $. Αντίθετα, αυτό μπορεί να προκύψει όταν ο χαρακτήρας ταιριάσματος είναι είτε στην μέση μίας ακμής ή σε έναν κόμβο. Στην πρώτη περίπτωση, «σπάμε» την παλιά ακμή σε δύο νέες, εισάγοντας έναν καινούριο κόμβο. Η ακμή η οποία οδηγεί στο νέο κόμβο περιέχει όλους εκείνους τους χαρακτήρες οι οποίοι έχουν ταιριάξει μέχρι στιγμής κατά μήκος της παλιάς ακμής. Από την άλλη, η ακμή που εξέρχεται του νέου κόμβου περιέχει όλους τους υπόλοιπους χαρακτήρες από την παλιά ακμή. Πλέον, μπορούμε να προσθέσουμε στο δένδρο T_i το υπόλοιπο του προσφύματος S[i…n], εισάγοντας μία επιπλέον ακμή από το νέο κόμβο. Στην δεύτερη περίπτωση, μπορούμε απλά να προσθέσουμε μία νέα ακμή, από τον κόμβο εκείνο με το υπόλοιπο του προσφύματος S[i…n]. Και στις δύο παραπάνω περιπτώσεις, όταν εισάγουμε τη νέα ακμή, επικολλάμε ως ετικέτα στο νέο κόμβο την τιμή i.

Ο χρόνος που απαιτείται για την προσθήκη κάποιου προσφύματος είναι ανάλογος του μήκους αυτού, γεγονός που μας οδηγεί, τελικά, σε έναν O(n^2) αλγόριθμο. Αν και ο

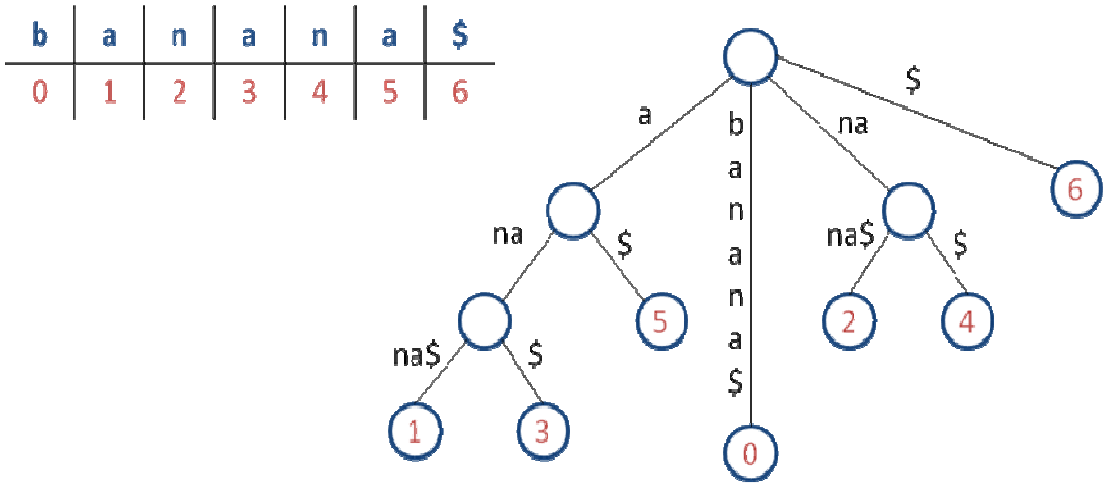
αλγόριθμος αυτός είναι πολύ απλός, ο τετραγωνικός χρόνος κατασκευής του τον καθιστά ιδιαίτερα αναποτελεσματικό σε κάποιες περιπτώσεις. Τα προσφυματικά δένδρα, για παράδειγμα, χρησιμοποιούνται σήμερα ευρέως σε πολύ μεγάλα προβλήματα ταιριάσματος ακολουθιών, όπου οι συμβολοσειρές εισόδου μπορεί να είναι ακολουθίες DNA, αποτελούμενες από χιλιάδες ή ακόμα και εκατομμύρια στο πλήθος χαρακτήρες. Ο τετραγωνικός χρόνος, λοιπόν, ενός τέτοιου, απλού στη πράξη, αλγορίθμου δεν αποδίδει σε εφαρμογές όπως αυτή. Ευτυχώς, υπάρχουν πιο αποτελεσματικοί, πλην, όμως, πολύπλοκοι τρόποι κατασκευής προσφυματικών δένδρων, οι οποίοι αιτούν γραμμικό $O(n)$ χρόνο.

4.5.2 Ταίριασμα Συμβολοσειράς με Προσφυματικά Δένδρα

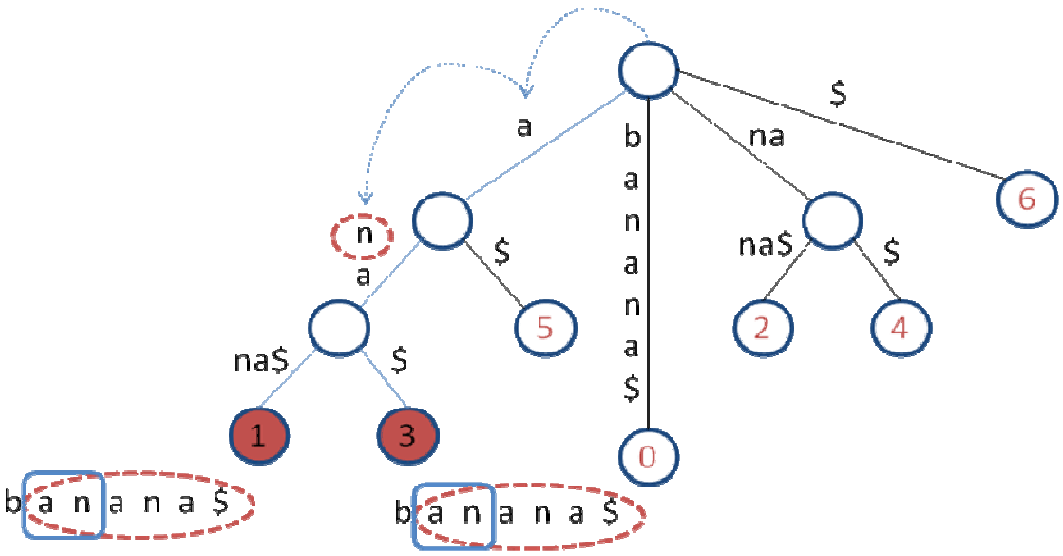
Αφού έχουμε παρουσιάσει τα προσφυματικά δέντρα καθώς και έναν απλό αλγόριθμο κατασκευής τους, μπορούμε πλέον να προχωρήσουμε, επιδεικνύοντας τον τρόπο που αυτά μπορούν να χρησιμοποιηθούν για την επίλυση του προσεγγιστικού ταιριάσματος.

Λήμμα 4.2: *Δεδομένου ενός κειμένου T, ενός προτύπου P, κι ενός προσφυματικού δένδρου S επί του T, μπορούμε να εντοπίσουμε μία εμφάνιση του προτύπου στο κείμενο σε χρόνο O(|P|).*

Προκειμένου να εντοπίσουμε όλες τις εμφανίσεις του P στο T, η διαδικασία την οποία ακολουθούμε έχει ως εξής: Αρχίζοντας από την ρίζα του S και, χρησιμοποιώντας τον τρέχοντα, κάθε φορά, χαρακτήρα του προτύπου, προσπαθούμε να αποφανθούμε για το ποία ακμή πρέπει να ακολουθήσουμε από τον τρέχοντα κόμβο του S. Όταν συναντήσουμε κάποια ακμή του S που φέρει ετικέτα με περισσότερους από έναν χαρακτήρες, τότε, εξετάζοντας διαδοχικά τα σύμβολα της ετικέτας, προσπαθούμε να τα ταιριάξουμε με τους τρέχοντες, κάθε φορά, χαρακτήρες του P. Η διαδικασία συνεχίζεται με τον ίδιο τρόπο, διακλαδώνοντας, κάθε φορά που έχουμε εξετάσει (και άρα ταιριάξει) όλους τους χαρακτήρες της ετικέτας κάποιας ακμής. Ο αλγόριθμος σταματάει, όταν, είτε: i) έχουμε εξετάσει όλους τους χαρακτήρες του P, στην οποία περίπτωση, έχουμε εντοπίσει μία εμφάνιση του προτύπου στο κείμενο και σημειώνουμε τον τρέχοντα κόμβο (δηλαδή, τον κόμβο που βρίσκεται κάτω από την τρέχουσα ακμή), ή ii) ο τρέχων χαρακτήρας του P δεν ταιριάζει με το τρέχον σύμβολο της τρέχουσας ακμής, στην οποία περίπτωση, δεν υπάρχει ταίριασμα, ή iii) πρέπει να αποφασίσουμε ποια διακλάδωση (ακμή) θα ακολουθήσουμε, αλλά, δεν υπάρχει διακλάδωση που ταιριάζει με το τρέχον σύμβολο του προτύπου P, στην οποία περίπτωση, επίσης, δεν υπάρχει ταίριασμα. Το παράδειγμα του σχήματος 4.10 παρουσιάζει την παραπάνω διαδικασία για αναζήτηση του προτύπου P = ’an’ στο κείμενο T = ’banana’. Με κόκκινο χρώμα επισημαίνονται οι κόμβοι-φύλλα των οποίων οι ετικέτες αντιστοιχούν στις θέσεις του κειμένου όπου εμφανίζεται το πρότυπο, δηλαδή, στις θέσεις 1 και 3.



Σχήμα 4.9: Στιγμιότυπο προσφυματικού δένδρου για την συμβολοσειρά “banana$”



Σχήμα 4.10: Αναζήτηση του “an” στο προσφυματικό δένδρο της συμβολοσειράς “banana$”. Με κόκκινο χρώμα επισημαίνονται οι κόμβοι-φύλλα των οποίων οι ετικέτες αντιστοιχούν στις θέσεις του κειμένου όπου εμφανίζεται το πρότυπο.

Η ορθότητα του παραπάνω αλγορίθμου, μπορεί εύκολα να αποδειχτεί, βάσει των εξής δύο γεγονότων: α) στο προσφυματικό δένδρο κάθε συμβολοσειράς αποθηκεύεται **ακριβώς** το σύνολο των προσφυμάτων της αντίστοιχης συμβολοσειράς, και β) κάθε εμφάνιση του P, έστω στην θέση i του T, αντιστοιχεί σε ένα πρόσφυμα T[i:][4].

Από τα παραπάνω, γίνεται προφανές ότι ο αλγόριθμος χρειάζεται O(|*P*|) χρόνο για τον εντοπισμό μιας εμφάνισης του προτύπου, μήκους |P|, στο S.

[4] Με T[i:] δηλώνεται η υποσυμβολοσειρά του T η οποία αρχίζει στην θέση i του κειμένου και τελειώνει στο τέλος αυτού

Αξίζει, επίσης, να σημειωθεί, ότι ο αλγόριθμος ,ουσιαστικά, επιστρέφει ένα υποδένδρο (στην πραγματικότητα επιστρέφει έναν δείκτη σε έναν κόμβο) και ότι οι κόμβοι αυτού αντιστοιχούν σε όλες τις εμφανίσεις του P στο T. Για να γίνει πιο κατανοητό αυτό, ας υποθέσουμε ότι το P εμφανίζεται στις θέσεις $i_1,i_2,\ldots,i_k$ και έστω $T[i_1:]$, $T[i_2:],\ldots,T[i_k:]$ οι αντίστοιχες υποσυμβολοσειρές του κειμένου. Κάθε μία από αυτές τις υποσυμβολοσειρές αποτελεί ένα πρόσφυμα του T, και άρα έχει αποθηκευτεί στο προσφυματικό δένδρο. Επιπλέον, καθεμιά από αυτές, αρχίζει με την ίδια συμβολοσειρά (δηλαδή το P), και ως εκ τούτου, όλες τους θα έχουν τους ίδιους «πρόγονους» στο δένδρο (από την κατασκευή του συμπιεσμένου Trie). Με άλλα λόγια, όλες αυτές οι υποσυμβολοσειρές αποτελούν μέρος του ίδιου υποδένδρου.

4.6 Ταίριασμα Συμβολοσειράς με 'don't care' Σύμβολα

Το κλασσικό πρόβλημα ταιριάσματος συμβολοσειράς, όπως έχουμε δει μέχρι τώρα, έγκειται στην αναζήτηση όλων των εμφανίσεων ενός δεδομένου προτύπου P, με μήκος m, σε ένα κείμενο T, μήκους n. Τόσο το πρότυπο P, όσο και το κείμενο T, απαρτίζονται από χαρακτήρες ενός πεπερασμένου αλφάβητου Σ. Στην ενότητα αυτή, θα μελετήσουμε μία ενδιαφέρουσα, αλλά και χρήσιμη σε αρκετές περιπτώσεις, επέκταση του προαναφερθέντος προβλήματος. Η νέα διάσταση αυτού του προβλήματος έγκειται στην εμφάνιση των ειδικών συμβόλων **'don't care'**, είτε στο κείμενο, είτε στο πρότυπο. Οι ειδικοί αυτοί χαρακτήρες, γνωστοί επίσης στη βιβλιογραφία και ως **κενά (gaps)**, συμβολίζονται συνήθως με ***** ή **Ø** (εδώ χρησιμοποιούμε το σύμβολο Ø), και ταιριάζουν επιτυχώς με οποιονδήποτε άλλο χαρακτήρα του δοσμένου αλφαβήτου Σ (σχήμα 4.11).

X	b	Ø	a	b	Ø	a	a
Y	b	a	a	Ø	b	Ø	Ø

Σχήμα 4.11: Η συμβολοσειρά X ταιριάζει επιτυχώς με τη συμβολοσειρά Y, χάρη στα ειδικά σύμβολα Ø.

Ορισμός 4.4: *Καλούμε 'don't care' χαρακτήρα και τον συμβολίζουμε με Ø, κάθε χαρακτήρα για τον οποίον ισχύουν: (i) Ø δεν ανήκει στο σύνολο Σ, και (ii) Ø ταιριάζει με κάθε άλλο χαρακτήρα α του συνόλου Σ. Για το λόγο αυτό λέμε ότι: Ø = α, για όλα τα α που ανήκουν στο Σ.*

4.6.1 Εμφάνιση συμβόλων 'don't care' στο πρότυπο

Ορισμός 4.5: *Κάθε συμβολοσειρά προτύπου, P, η οποία περιέχει 'don't care' χαρακτήρες, μπορεί να θεωρηθεί ως ένα σύνολο υποσυμβολοσειρών P_i , $1 \le i \le l$, όπου κάθε P_i αποτελεί μία υποσυμβολοσειρά ορισμένη στο αλφάβητο Σ. Για κάθε $1 \le i \le l-1$, ορίζεται η παράμετρος k_i, η οποία υποδεικνύει το πλήθος των συμβόλων 'don't care' που εμφανίζονται μεταξύ των P_i και P_{i+1}.*

Με βάση τον παραπάνω ορισμό μπορούμε να φανταστούμε το πρότυπο P ως:

$$P = P_1 Ø^{k1} P_2 Ø^{k2} \dots Ø^{kl-1} P_l ,$$

ενώ το πλήθος των συμβόλων Ø θα ισούται με $K=\sum_{1 \le i \le l-1} k_i$

Ορισμός 4.6: *Δεδομένου ενός κειμένου T, ορισμένου στο σύνολο Σ, κι ενός προτύπου P, ορισμένου στο σύνολο ΣU{Ø}, το πρόβλημα* ***'DCP' (Don't Care in Pattern),*** *έγκειται στην εύρεση όλων των εμφανίσεων του P στο T.*

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	A	C	C	A	A	C	A	A	C	A	G	C	A
P	A	Ø	C	Ø	Ø	C	A						
P				A	Ø	C	Ø	Ø	C	A			
P							A	Ø	C	Ø	Ø	C	A

Σχήμα 4.12: Στιγμιότυπο του προβλήματος DCP

Έστω $Y = y_1y_2\dots y_m$ το πρότυπο, μήκους m, με κάθε y_i να ανήκει στο σύνολο ΣU{Ø}, και $X = x_1x_2\dots x_n$ η συμβολοσειρά κειμένου, μήκους n, όπου κάθε x_i ανήκει στο Σ = {1,2,…,s}. Λέμε ότι η συμβολοσειρά $X(j) = x_jx_{j+1}\dots x_{j-1+m}$ ταιριάζει επιτυχώς με την συμβολοσειρά Y εάν:

$$x_{j-1+i} = y_i \qquad \text{ή} \qquad y_i = Ø,$$

για όλα τα $0 \le i \le m$.

Ο αλγόριθμος 4.6 εντοπίζει όλες τις εμφανίσεις του προτύπου στο κείμενο, υπολογίζοντας όλες τις θέσεις j για τις οποίες η X(j) ταιριάζει με την συμβολοσειρά Y.

Αλγόριθμος: dontcarefunction(**char[]** x, **char[]** y, **int** N)
Είσοδος: Το κείμενο x, το πρότυπο y και ένας ακέραιος N
Έξοδος: Όλες οι εμφανίσεις του y στο x

```
1.      for 1 ≤ i ≤ m do
2.              if   yi=Ø
3.                      ri=0;
4.                      i++;
5.              else
6.                      ri random from {1,2,…,N};
7.                      i++;
```

8. $t = \sum^{m}_{i=1} y_i r_i$;
9. **for** $1 \leq j \leq n-m$ **do**
10. $s_j = \sum^{m}_{i=1} x_{j-1+i} r_i$; (using FFT)
11. **return** matches for those j's where s(j)=t;

Αλγόριθμος 4.6: Ταίριασμα συμβολοσειράς με "don't care" σύμβολα

Εάν υπάρχει κάποια συμβολοσειρά X(j) η οποία να ταιριάζει με την Y, τότε ο παραπάνω αλγόριθμος σίγουρα θα την εντοπίσει. Διαφορετικά, για κάποια i σίγουρα ισχύει ότι, $y_i \neq x_{j-1+i}$ και $y_i \neq Ø$. Η εξίσωση s(j) = t, έχει μία μοναδική λύση για r_i , την οποία και θα επιλέξουμε με πιθανότητα λάθους το πολύ 1/N. Ο αλγόριθμος αιτεί O(*n logm*) πολυπλοκότητα χρόνου, θεωρώντας ότι μπορούμε να κάνουμε υπολογισμούς της τάξης log(NΣn) σε σταθερό χρόνο, από τη στιγμή που οι γρήγοροι μετασχηματισμοί fourier (FFT) μπορούν να γίνουν σε χρόνο O(*n logm*). Όλοι οι υπολογισμοί θα μπορούσαν να γίνουν επίσης με χρήση του modulo κι ενός τυχαίου πρώτου αριθμού p>N.

4.6.2 Εμφάνιση συμβόλων 'don't care' στο κείμενο

Ορισμός 4.7: *Κάθε συμβολοσειρά κειμένου, T, η οποία περιέχει 'don't care' χαρακτήρες, μπορεί να θεωρηθεί ως ένα σύνολο υποσυμβολοσειρών T_i , $1 \leq i \leq l$, όπου κάθε P_i αποτελεί μία υποσυμβολοσειρά ορισμένη στο αλφάβητο Σ. Για κάθε $1 \leq i \leq l-1$, ορίζεται η παράμετρος k_i, η οποία υποδεικνύει το πλήθος των συμβόλων 'don't care' που εμφανίζονται μεταξύ των T_i και T_{i+1}.*

Με βάση τον παραπάνω ορισμό μπορούμε να φανταστούμε το κείμενο T ως:

$$T = T_1 Ø^{k_1} T_2 Ø^{k_2} \ldots Ø^{k_{l-1}} T_l ,$$

ενώ το πλήθος των συμβόλων Ø θα ισούται με $K=\sum_{1 \leq i \leq l-1} k_i$.

Ορισμός 4.8: *Δεδομένου ενός προτύπου P, ορισμένου στο σύνολο Σ, κι ενός κειμένου T, ορισμένου στο σύνολο ΣU{Ø}, το πρόβλημα* ***'DCT' (Don't Care in Text),*** *έγκειται στην εύρεση όλων των εμφανίσεων του P στο T.*

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	A	C	Ø	A	Ø	Ø	A	G	Ø	A	G	Ø	A
P	A	C	C	A	G	C	A						
P				A	C	C	A	G	C	A			

Σχήμα 4.13: Στιγμιότυπο του προβλήματος DCT

Ο αλγόριθμος 4.6 μπορεί εύκολα να επεκταθεί προκειμένου να χειρίζεται περιπτώσεις όπου και το κείμενο μπορεί να περιέχει χαρακτήρες "don't care". Για το λόγο αυτό αντικαθιστούμε τον τύπο για τον υπολογισμό της τιμής t (γραμμή 8) με τον ακόλουθο: $t(j) = \sum^{m}_{1\le i\le m|x_{i-1+i}\neq Ø} y_i r_i$. Η τιμή αυτή εξαρτάται πλέον από την μεταβλητή j, ενώ ο παραπάνω τύπος δεν είναι τίποτε άλλο από τον μετασχηματισμό fourier (FFT) των y_i και r_i για $1\le i\le m$. Εάν αντικαταστήσουμε επίσης τους χαρακτήρες "don't care" με μηδενικά στην γραμμή 10 του κώδικα, τότε εύκολα θα πάρουμε ότι $s(j)=t(j)$, εάν για κάποιο j ισχύει ότι το X(j) ταιριάζει επιτυχώς με το Y. Όπως και παραπάνω, έτσι και σ' αυτήν την περίπτωση, η πιθανότητα αποτυχίας είναι ίση το πολύ με 1/N.

Βιβλιογραφία Κεφαλαίου

[1] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[2] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[3] Π.Δ. Μποζάνης, *Αλγόριθμοι: Σχεδιασμός και Ανάλυση*, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002.

[5] Tatsuya Akutsu, *Approximate string matching with don't care characters*, Department of Computer Science, Gunma University, Japan.

[6] A. Amir, M. Lewenstein, E. Porat, *Faster Algorithms for String Matching with k Mismatches,* June 28, 2002.

[7] Ricardo A. Baeza-Yates and Chris H. Perleberg, *Fast and Practical Approximate String Matching,* Department of Computer Science, University of Chile.

[8] Ricardo A. Baeza-Yates and Gonzalo Navarro, *Fast Approximate String Matching,* Department of Computer Science, University of Chile.

[9] Costas S. Iliopoulos and M. Sohel Rahman, *Pattern Matching Algorithms with Don't Cares,* Department of Computer Science, King's College London, England.

[10] Adam Kalai, *Efficient Pattern-Matching with Don't Cares.*

[11] Ricardo A. Baeza-Yates and Gaston H. Gonnet, *Fast String Matching with Mismatches.*

[12] Gonzalo Navarro, *A Guided Tour to Approximate String Matching*, Department of Computer Science, University of Chile.

5 Εφαρμογές του προβλήματος στην Βιοπληροφορική

Περιεχόμενα

5.1 Εισαγωγή

Οι διάφορες τεχνικές ταιριάσματος συμβολοσειρών, που μελετήσαμε έως τώρα, αποτέλεσαν κατά καιρούς ένα, ιδιαίτερα, χρήσιμο εργαλείο στην επίλυση πολλών προβλημάτων τα οποία σχετίζονται με την πληροφορική, συμπεριλαμβανομένης της αναζήτησης και ανάκτησης πληροφοριών από μεγάλες βάσεις δεδομένων, της επεξεργασίας κειμένων, καθώς και της συμπίεσης και κρυπτογράφησης διαφόρων δεδομένων. Τα τελευταία, ωστόσο, χρόνια, οι ραγδαίες εξελίξεις στους τομείς της μοριακής βιολογίας και της γενετικής έχουν καταστήσει αναγκαία την χρήση τέτοιων μεθόδων και τεχνικών και σε αυτούς τους τόσο σημαντικούς επιστημονικούς κλάδους. Δεδομένου ότι οι πληροφορίες που διαχειρίζονται αυτές οι επιστήμες (μεγάλης κλίμακας γονιδιωματικές αλληλουχίες - DNA, RNA, πρωτεΐνες -), παράγονται με ιδιαίτερα γοργούς ρυθμούς, κρίνεται σκόπιμη και αναγκαία η ύπαρξη ειδικών υπολογιστικών εργαλείων και τεχνικών για την καλύτερη ανάλυση και διαχείριση αυτών των γονιδιωματικών ακολουθιών.

Η **Βιοπληροφορική (bioinformatics)** αποτελεί ένα νέο, σχετικά, επιστημονικό κλάδο, ο οποίος προέκυψε από τη συνεργασία των επιστημών της Βιολογίας και της Πληροφορικής. Θεωρώντας τα βιολογικά δεδομένα (DNA, RNA, πρωτεΐνες) ως ψηφιακή πληροφορία, εφαρμόζει αλγορίθμους για την επεξεργασία τους και την παραγωγή χρήσιμων συμπερασμάτων με γρήγορο και εύκολο τρόπο. Συνήθως χρησιμοποιούνται μέθοδοι της τεχνητής νοημοσύνης, όπως π.χ. η εξόρυξη δεδομένων.

Τα βιολογικά μόρια, όπως το DNA, το RNA και οι πρωτεΐνες, μπορούν να θεωρηθούν ως ακολουθίες συμβόλων, δηλαδή συμβολοσειρές. Για παράδειγμα το DNA μπορεί να θεωρηθεί ως μια ακολουθία χιλιάδων νουκλεοτιδίων ή βάσεων. Υπάρχουν τέσσερα είδη βάσεων και αντίστοιχα τέσσερα είδη νουκλεοτιδίων: η αδενίνη, η θυμίνη, η γουανίνη και η κυτοσίνη. Αντιστοιχίζοντας σε καθεμία απ' αυτές ένα σύμβολο, π.χ. "A" για την αδενίνη, "T" για τη θυμίνη, "G" για τη γουανίνη και "C" για την κυτοσίνη, μπορούμε να κατασκευάσουμε τέτοιες συμβολοσειρές/ακολουθίες (π.χ. "...AAGATCGGTAC..."). Αυτού του είδους η αναπαράσταση είναι ιδιαίτερα βολική για επεξεργασία σε ηλεκτρονικό υπολογιστή και μπορεί να μας δώσει χρήσιμα αποτελέσματα.

Τα βασικότερα προβλήματα που απασχολούν σήμερα τους επιστήμονες της Βιοπληροφορικής, σχετικά με την ανάλυση ακολουθιών βιολογικών δεδομένων και, σε συνδυασμό πάντα με το πρόβλημα του ταιριάσματος συμβολοσειρών, είναι τα εξής:

- Ακριβής Εύρεση Προτύπου – Exact pattern matching problem
- Ακριβής Εύρεση Πολλαπλών Προτύπων – Exact multiple pattern matching problem
- Στοίχιση ζεύγους ακολουθιών – Pairwise alignment problem
- Πολλαπλή στοίχιση ακολουθιών – Multiple alignment problem
- Μέγιστη κοινή υποσυμβολοσειρά δύο ακολουθιών - Longest Common Substring Problem
- DNA Contamination Problem
- Εύρεση Κοινών Μοτίβων σε δύο ή περισσότερες Βιολογικές Ακολουθίες

- Εύρεση Επαναλήψεων σε Βιολογικές Ακολουθίες
- Αναζήτηση ομόλογων πρωτεϊνών μέσα σε μια μεγάλη βάση δεδομένων

Τα θέματα αυτά απασχολούν χρόνια τους επιστήμονες, και αποκτούν όσο περνάει ο καιρός, όλο και μεγαλύτερη σημασία, καθώς αυξάνεται ραγδαία το μέγεθος των διαθέσιμων βάσεων δεδομένων, αλλά και η ευκολία πρόσβασης σ' αυτές για τους ερευνητές. Στη συνέχεια θα περιγράψουμε κάποια από τα προβλήματα αυτά και θα παρουσιάσουμε διάφορους χρήσιμους αλγορίθμους/τεχνικές καθώς και υπολογιστικά εργαλεία που χρησιμοποιούνται σήμερα ευρέως στην καλύτερη, ευκολότερη και, κυρίως, γρηγορότερη επίλυση αυτών. Ιδιαίτερη, δε, έμφαση θα δοθεί στο πρόβλημα της στοίχισης ακολουθιών, μιας και αποτελεί το πιο συχνά εφαρμοζόμενο πρόβλημα σε εφαρμογές της βιοπληροφορικής.

5.2 Ακριβής Εύρεση Προτύπου

Το πρόβλημα της ακριβούς εύρεσης προτύπου ορίζεται ως εξής:

Ορισμός 5.1: *Δεδομένου ενός προτύπου P, μήκους $|P|=n$, κι ενός κειμένου T, μήκους $|T|=m$, αιτείται ο εντοπισμός όλων των εμφανίσεων του P στο T.*

Η ακριβής εύρεση προτύπου σε εφαρμογές της βιοπληροφορικής επιλύεται συνήθως πιο αποτελεσματικά με την χρήση των δένδρων προσφυμάτων (suffix trees), τα οποία και μελετήσαμε στο προηγούμενο κεφάλαιο.

Η μεθοδολογία που υιοθετείται στην επίλυση αυτού του προβλήματος ακολουθεί τα εξής βήματα:

- Κατασκευή του δένδρου προσφυμάτων (suffix tree) για την ακολουθία εισόδου T σε $O(|T|)$ χρόνο,
- Ξεκινώντας από τη ρίζα, σύγκριση ένα προς ένα των χαρακτήρων του P, ακολουθώντας το κατάλληλο μονοπάτι:
    - Εάν εμφανιστεί κάποιο μη-ταίριασμα, τότε το πρότυπο δεν εμφανίζεται στην ακολουθία,
    - διαφορετικά αναφορά ως απάντηση όλων των φύλλων που βρίσκονται κάτω από τον κόμβο του τελευταίου χαρακτήρα του P.

Η αναζήτηση στοιχίζει $O(n+k)$ χρόνο, για ένα πρότυπο μήκους $|P| = n$, ενώ η τιμή k αντιστοιχεί στο πλήθος εμφανίσεων του P στο κείμενο T.

Η μέθοδος του ακριβούς ταιριάσματος προτύπου, δεν χρησιμοποιείται ιδιαίτερα στην υπολογιστική βιολογία (βιοπληροφορική), αφού η φύση των γονιδιωματικών ακολουθιών είναι τέτοια ώστε σπανίως το πρότυπο να ταιριάζει πλήρως στο κείμενο (δεδομένη αλληλουχία).

5.3 Ακριβής Εύρεση Πολλαπλών Προτύπων

Το πρόβλημα της ακριβούς εύρεσης πολλαπλών προτύπων ορίζεται ως εξής:

Ορισμός 5.2: *Δεδομένου ενός συνόλου P, αποτελούμενο από k πρότυπα, $P = \{P_1, P_2, \ldots, P_k\}$, κι ενός κειμένου T, αιτείται ο εντοπισμός όλων των εμφανίσεων οποιουδήποτε προτύπου P_i, $1 \leq i \leq k$, στο κείμενο T.*

Και στο πρόβλημα αυτό, αφού, αρχικά, κατασκευάσουμε το δένδρο προσφυμάτων για την ακολουθία εισόδου T σε $O(|T|)$ χρόνο, ξεκινώντας από τη ρίζα, αναζητούμε διαδοχικά όπως και στην προηγούμενη εφαρμογή το σύνολο των προτύπων $P=\{P_1, P_2, \ldots, P_k\}$.

Η αναζήτηση αυτή στοιχίζει $O(n+|P|+ k_\rho)$ χρόνο, όπου $|P|= |P_1| + |P_2| + \ldots$ και k_ρ το πλήθος εμφανίσεων των προτύπων στο T.

5.4 Στοίχιση Ζεύγους Ακολουθιών

Η **στοίχιση ζεύγους ακολουθιών (pairwise sequence alignment)**, είναι μια διαδικασία κατά την οποία δύο ακολουθίες ή αλλιώς συμβολοσειρές τοποθετούνται η μία κάτω από την άλλη, με τέτοιον τρόπο που τα κοινά τους σύμβολα να είναι τοποθετημένοι στην ίδια θέση. Σκοπός είναι να βρεθεί η "βέλτιστη στοίχιση", δηλαδή η στοίχιση στην οποία οι δύο ακολουθίες ταιριάζουν περισσότερο μεταξύ τους. Η διαδικασία αυτή χρησιμοποιείται ιδιαίτερα στη βιοπληροφορική, όπου ως ακολουθίες χρησιμοποιούνται τμήματα DNA, RNA ή πρωτεϊνών. Οι μέθοδοι στοίχισης των ακολουθιών αυτών είναι πολύ χρήσιμες στον ερευνητή, προκειμένου να ταυτοποιήσει, π.χ. μια άγνωστη πρωτεΐνη, αλλά και για να εξαγάγει πληροφορίες, τόσο εξελικτικές, όσο και λειτουργικές.

Η διαδικασία της στοίχισης, όταν συμβαίνει σε συμβολοσειρές μεγάλου μήκους, όπως αυτές που προκύπτουν από τα βιολογικά δεδομένα, είναι μια σχετικά δύσκολη διαδικασία. Χαρακτηριστικά, αξίζει να αναφέρουμε ότι, αν υποθέσουμε ότι έχουμε 2 ακολουθίες πρωτεϊνών, μήκους n, τότε αποδεικνύεται ότι οι πιθανές στοιχίσεις των δυο αυτών ακολουθιών είναι:

$$\binom{2n}{n} = \frac{(2n!)}{(n!)^2} \approx \frac{2^{2n}}{\sqrt{2\pi n}}$$

Ο αριθμός αυτός είναι τεράστιος ακόμα και για μικρές ακολουθίες, και γι' αυτό το λόγο έχουν αναπτυχθεί αλγόριθμοι δυναμικού προγραμματισμού οι οποίοι βρίσκουν την καλύτερη στοίχιση των ακολουθιών στο μικρότερο δυνατό χρόνο.

Η στοίχιση ακολουθιών χρησιμοποιείται στο πεδίο της βιοπληροφορικής για την εύρεση ομοιοτήτων μεταξύ βιολογικών ακολουθιών (DNA, RNA, πρωτεΐνες...), μια εργασία πολύ συχνή και με ιδιαίτερη σημασία. Η σημασία της πηγάζει από το γεγονός ότι οι ομοιότητες αυτές υποδηλώνουν συνήθως κάποια βιολογική συσχέτιση, κάτι που μας οδηγεί σε καλύτερη κατανόηση των διαφόρων βιολογικών μηχανισμών.

Υποθέτοντας ότι δυο αλληλουχίες **a** και **b** έχουν μεταξύ τους εξελικτική σχέση (δηλαδή ότι οι μεταξύ τους διαφορές είναι αποτέλεσμα εξελικτικών γεγονότων: μεταλλάξεων, εισαγωγών, διαγραφών), ο στόχος της διαδικασίας στοίχισης τους είναι να αποκαλύψει αυτά τα εξελικτικά γεγονότα. Εάν για παράδειγμα, **a** : AGTACCACTTTGA και **b** : AGCAACTTAATGA , τότε η στοίχιση :

AGTACCACTT - -TGA
AGCA- - ACTTAATGA

υποδηλώνει την παρουσία μιας μετάλλαξης (T-C) στη θέση 3, καθώς και δυο γεγονότων εισαγωγής/διαγραφής δυο βάσεων στις θέσεις που καταλαμβάνονται από τα κενά.

Για την επίλυση του προβλήματος (και μιας και η πραγματική εξελικτική ιστορία των αλληλουχιών μας είναι άγνωστη), καταφεύγουμε στην εξής (a posteriory) λύση : Εάν έχουμε έναν τρόπο να βαθμολογούμε μια στοίχιση με τέτοιο τρόπο ώστε όσο μεγαλύτερη η βαθμολογία της, τόσο μεγαλύτερη η πιθανότητα τα εξελικτικά γεγονότα που υποδηλώνει να είναι αληθή, τότε η στοίχιση που θέλουμε είναι αυτή η οποία αντιστοιχεί στο ολικό μέγιστο της βαθμολογίας όλων των δυνατών στοιχίσεων. Με αυτό τον τρόπο το πρόβλημα μετατίθεται από την εύρεση της εξελικτικής ιστορίας δυο ακολουθιών, στην εύρεση του ολικού μεγίστου μιας συνάρτησης των ακολουθιών (στην θέση της εξέλιξης τώρα έχουμε τη συνάρτηση που υπολογίζει τη βαθμολογία μιας στοίχισης). Άρα, τα προβλήματα είναι : α) δεδομένης μιας στοίχισης, πως υπολογίζουμε τη βαθμολογία της, και, β) πως, από όλες τις δυνατές στοιχίσεις δυο αλληλουχιών, βρίσκουμε εκείνη με τη μεγαλύτερη βαθμολογία.

Ο βασικός κανόνας της στοίχισης ακολουθιών είναι η τοποθέτηση των δύο, υπό εξέταση, ακολουθιών με τέτοιον τρόπο, την μία κάτω από την άλλη, ώστε να ταιριάζουν κατά το βέλτιστο δυνατόν. Από αυτό φαίνεται ότι μπορεί κάποια από τα σύμβολα των ακολουθιών να μην ταιριάζουν μετά τη στοίχιση. Το τι γίνεται τότε, έχει να κάνει με τον τρόπο με τον οποίο μετράμε το "ταίριασμα". Συνήθως γίνεται προσπάθεια να στοιχιστούν οι ακολουθίες με τέτοιον τρόπο ώστε να μεγιστοποιηθεί κάποιο score. Κάθε ζεύγος συμβόλων που ταιριάζουν μεταξύ τους παίρνει κάποιο θετικό score (π.χ. +1), ενώ κάθε ζεύγος που αποτυγχάνει να ταιριάξει παίρνει κάποιο αρνητικό score (π.χ. -1). Αν επιτρέπεται να εισάγουμε κενούς χαρακτήρες για αποφυγή αποτυχίας σε ένα σημείο, τότε κάθε κενό κοστολογείται και αυτό με αρνητικό score (π.χ. -1). Δύο κενά δεν γίνεται να ταιριάζουν μεταξύ τους. Αθροίζοντας όλα τα επιμέρους score υπολογίζεται το συνολικό score της στοίχισης.

Πιο συγκεκριμένα, έστω δυο ακολουθίες **a** και **b**, και μια στοίχιση αυτών των ακολουθιών στην οποία υπάρχουν $\mathbf{N_{i,j}}$ ζεύγη στοιχισμένων αμινοξέων A_i και B_j (στις θέσεις i και j των ακολουθιών **a** και **b**), και ένα σύνολο **Ngap** κενών (καθ' όλο το μήκος της στοίχισης). Εάν $\mathbf{s(A_i, B_j)}$ είναι η βαθμολογία υποκατάστασης του αμινοξέος A_i από το B_j με βάση κάποιον δεδομένο πίνακα υποκατάστασης, τότε η βαθμολογία της στοίχισης είναι :

$$\text{score} = \sum_{i,j} N_{i,j} \cdot s(A_i, B_j) - \alpha \text{Ngap},$$

όπου *α*Ngap: το αρνητικό κόστος από την εισαγωγή των Ngap κενών στην στοίχιση (gap penalty).

Υπάρχουν δύο είδη στοίχισης, η **ολική στοίχιση (global alignment)** και η **τοπική στοίχιση (local alignment)**, οι οποίες χρησιμοποιούνται ανάλογα με τον τύπο του προβλήματος, που αντιμετωπίζεται.

Κατά την ολική στοίχιση ακολουθιών (global sequence alignment), γίνεται προσπάθεια να στοιχιστεί κάθε σύμβολο κάθε ακολουθίας. Οι δύο ακολουθίες στοιχίζονται σε ολόκληρο το μήκος τους με τον καλύτερο δυνατό τρόπο. Κάθε σύμβολο κάθε ακολουθίας, λοιπόν, αντιστοιχίζεται σε ένα σύμβολο της άλλης ή σε ένα κενό. Ένας γνωστός αλγόριθμος για ολική στοίχιση ακολουθιών είναι ο αλγόριθμος Needleman-Wunsch, τον οποίο και θα παρουσιάσουμε στη συνέχεια.

Κατά την τοπική στοίχιση ακολουθιών (local sequence alignment), γίνεται η καλύτερη δυνατή στοίχιση μεταξύ τμημάτων των δύο ακολουθιών. Δηλαδή, επιτρέπεται κάποια κομμάτια που "χαλάνε" τη στοίχιση να μείνουν εκτός. Ένα παράδειγμα που χρησιμοποιείται συχνά είναι ότι μπορούμε να χρησιμοποιήσουμε την τοπική στοίχιση ακολουθιών προκειμένου να βρούμε τις προτάσεις δύο κειμένων, οι οποίες παρουσιάζουν την περισσότερη ομοιότητα. Ένας γνωστός αλγόριθμος για τοπική στοίχιση ακολουθιών είναι ο αλγόριθμος Smith-Waterman, τον οποίον, επίσης, θα παρουσιάσουμε παρακάτω.

Τόσο ο αλγόριθμος Needleman & Wunsch, όσο και ο αλγόριθμος Smith & Waterman, προσπαθούν να απαντήσουν στο εξής ερώτημα : από όλες τις δυνατές στοιχίσεις ανάμεσα στις ακολουθίες **a** και **b**, πως βρίσκω εκείνη τη στοίχιση για την οποία η βαθμολογία λαμβάνει τη μέγιστη τιμή της ;

5.4.1 Ο Αλγόριθμος Needleman & Wunsch

Μια συστηματική έρευνα όλων των δυνατών στοιχίσεων δυο αλληλουχιών με μήκη n και m θα απαιτούσε την εξέταση ενός εκθετικά μεγάλου αριθμού στοιχίσεων. Ο αλγόριθμος **Needleman & Wunsch** είναι ένας αλγόριθμος που χρησιμοποιείται για ολική στοίχιση ακολουθιών, μια διαδικασία ιδιαίτερα χρήσιμη στην βιοπληροφορική Συγκεκριμένα χρησιμοποιείται για να υπολογιστεί η βέλτιστη ολική στοίχιση μεταξύ δύο ακολουθιών/συμβολοσειρών, με υπολογιστικό κόστος ανάλογο του γινομένου του μηκών των αλληλουχιών O(*nm*). Η διαφορά ανάμεσα σε ένα υπολογιστικό κόστος ανάλογο του O(*nm*) και ενός εκθετικά μεγάλου κόστους είναι τεράστια : για δυο αλληλουχίες μήκους, π.χ. 1000 χαρακτήρων, είναι η διαφορά ανάμεσα σε δευτερόλεπτα και ώρες (εάν όχι ημέρες) υπολογισμού. Ο συγκεκριμένος αλγόριθμος ανήκει στη φιλοσοφία αλγορίθμων δυναμικού προγραμματισμού.

Σε γενικές γραμμές, στον πυρήνα του αλγορίθμου Needleman-Wunsch για ολική στοίχιση ακολουθιών υπάρχει ένας πίνακας του οποίου οι γραμμές αντιστοιχούν στα σύμβολα της μίας ακολουθίας και οι στήλες στα σύμβολα της άλλης. Κάθε κελί αυτού του πίνακα αντιστοιχεί σε ένα ταίριασμα γραμμάτων των δύο ακολουθιών, και περιέχει δύο τιμές: ένα σκορ (που υπολογίζεται με βάση ένα συγκεκριμένο σχήμα)

και έναν δείκτη (του οποίου η χρήση γίνεται πιο σαφής παρακάτω). Κατά την εκτέλεση ακολουθείται μια αλληλουχία τριών φάσεων:

1. Αρχικοποίηση πίνακα,

2. Γέμισμα πίνακα,

3. Οπισθοχώρηση (traceback).

Στην πρώτη φάση προσδίδονται τιμές στην πρώτη γραμμή και στην πρώτη στήλη του πίνακα. Αυτό υπονοεί ότι για κάθε κελί της πρώτης γραμμής και της πρώτης στήλης του πίνακα γίνονται δύο αναθέσεις: ανατίθεται ένα σκορ και ένας δείκτης. Στην δεύτερη φάση γεμίζονται τα κελιά του πίνακα με τιμές. Και σε αυτήν την φάση, σε κάθε κελί του πίνακα ανατίθεται ένα σκορ και ένας δείκτης. Αυτή η ανάθεση γίνεται με βάση τις τιμές των γειτόνων του κελιού του πίνακα. Τέλος, στην τρίτη φάση ανατρέχουμε τον πίνακα προς τα πίσω, με βάση τους δείκτες, για να βρούμε την επιθυμητή στοίχιση.

Πιο συγκεκριμένα, η θεμελιώδης παρατήρηση στην οποία βασίζεται ο αλγόριθμος είναι ότι ***οποιοδήποτε υποσύνολο της βέλτιστης στοίχισης πρέπει επίσης να είναι βέλτιστο***, ειδάλλως η στοίχιση θα μπορούσε να βελτιστοποιηθεί και άλλο (μέσω της βελτίωσης της υποστοίχισης).

Έστω, λοιπόν, μια βέλτιστη στοίχιση ανάμεσα σε ένα υποσύνολο k αμινοξέων από την αλληλουχία **a**, $(a_1a_2\ldots a_k)$ και σε ένα υποσύνολο m αμινοξέων από την αλληλουχία **b**, $(b_1b_2\ldots b_m)$, και έστω ότι η τρέχουσα τιμή βαθμολογίας (αυτής της βέλτιστης υποστοίχισης) είναι $S_{k,m}$. Θέλουμε να επεκτείνουμε αυτή τη βέλτιστη υποστοίχιση κατά μια θέση προς τα δεξιά. Υπάρχουν τρεις μόνο δυνατότητες για να επιτευχθεί αυτή η επέκταση :

1. Στοιχίζουμε το αμινοξύ (a_{k+1}) με το αμινοξύ (b_{m+1}). Αυτό θα αυξήσει τη βαθμολογία της στοίχισης κατά τόσο, όσο η βαθμολογία υποκατάστασης του αμινοξέος (a_{k+1}) από το αμινοξύ (b_{m+1}), όπως αυτή δίνεται από τον πίνακα βαθμολόγησης. Άρα, σε αυτή την περίπτωση, η καινούργια τιμή της συνολικής βαθμολογίας θα είναι :

$$S_{k+1,m+1} = S_{k,m} + s(a_{k+1} , b_{m+1}).$$

2. Προσθέτουμε το αμινοξύ (a_{k+1}) στην αλληλουχία **a** και εισαγάγουμε ένα κενό στην αλληλουχία **b**. Σε αυτή την περίπτωση :

$$S_{k+1,m+1} = S_{k+1,m} - (\text{gap penalty}),$$

όπου $S_{κ+1,μ}$ είναι η βαθμολογία που αντιστοιχεί στη βέλτιστη στοίχιση των αμινοξέων $(a_1a_2\ldots a_{k+1})$ και $(b_1b_2\ldots b_m)$.

3. Προσθέτουμε το αμινοξύ (b_{m+1}) στην αλληλουχία **b** και εισαγάγουμε ένα κενό στην αλληλουχία **a**. Σε αυτή την περίπτωση :

$$S_{k+1,m+1} = S_{k,m+1} - (\text{gap penalty})$$

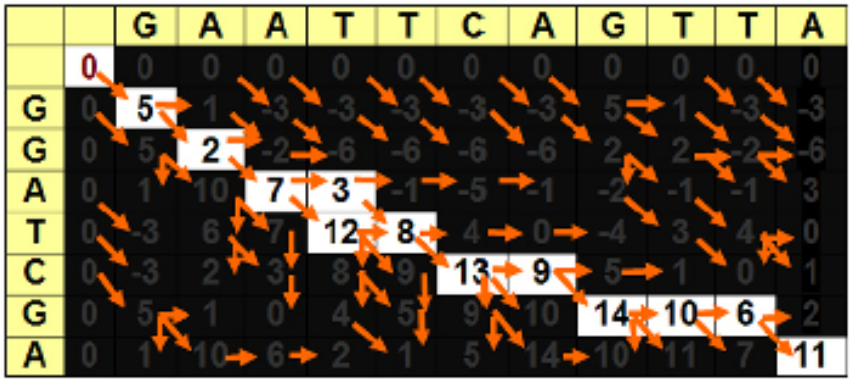
όπου $S_{k,m+1}$ είναι η βαθμολογία που αντιστοιχεί στη βέλτιστη στοίχιση των αμινοξέων ($(a_1a_2\ldots a_k)$ και $(b_1b_2\ldots b_{m+1})$. Η καινούργια βέλτιστη βαθμολογία στοίχισης θα είναι προφανώς το μέγιστο των τριών αυτών τιμών της $S_{k+1,m+1}$ δηλαδή :

$$S_{k+1,m+1} = \max \left\{ \begin{array}{l} S_{k,m} + s(a_{k+1}, b_{m+1}) \\ S_{k+1,m+1} = S_{k+1,m} - (\text{gap penalty}) \\ S_{k+1,m+1} = S_{k,m+1} - (\text{gap penalty}) \end{array} \right\}$$

Άρα, εάν έχουμε τιμές για τα $S_{k,m}$, $S_{k+1,m}$ και $S_{k,m+1}$ μπορούμε να υπολογίσουμε το $S_{k+1,m+1}$ απλά με το να βρούμε το μέγιστο από τις τρεις αριθμητικές εκφράσεις που δίδονται μέσα στις αγκύλες της παραπάνω εξίσωσης. Επομένως, το μόνο που χρειαζόμαστε για να μπορέσουμε να υπολογίσουμε όλα τα $S_{\kappa,\mu}$ (για κάθε τιμή του k και m) είναι οι αρχικές τιμές για την πρώτη στήλη και πρώτη γραμμή του πίνακα. Για τον αλγόριθμο Needleman & Wunsch η αρχικοποίηση των τιμών είναι :

$$S_{0,0} = 0$$
$$S_{k,0} = -\,k\ (\text{gap penalty})$$
$$S_{0,m} = -\,m\ (\text{gap penalty}).$$

Ας σημειωθεί ότι η αρίθμηση των αλληλουχιών αρχίζει από το 1 (και κατά συνέπεια το $S_{k,0}$ είναι ίσο με τη βαθμολογία που αντιστοιχεί στην απλή εισαγωγή k κενών πριν την αρχή της αλληλουχίας **b**). Τέλος, στο σχήμα 5.1 παρουσιάζεται ένα στιγμιότυπο ολικής στοίχισης των ακολουθιών "GGATCGA" και "GAATTCAGTTA".



(α)

```
GAATTCAGTTA      GAATTCAGTTA
|  | ||  |  |     |  ||  |  |
GGA-TC-G--A      GGAT-C-G--A
```

(β)

Σχήμα 5.1: Στιγμιότυπο παραδείγματος ολικής στοίχισης ακολουθιών : **(α)** πίνακας δυναμικού προγραμματισμού και οπισθοχώρηση (traceback) με βάση τους δείκτες, και **(β)** οι δύο εναλλακτικές στοιχίσεις που προκύπτουν

5.4.2 Ο Αλγόριθμος Smith & Waterman

Ο αλγόριθμος των **Smith & Waterman** είναι ένας αλγόριθμος που χρησιμοποιείται για τοπική στοίχιση ακολουθιών, ήτοι, την καλύτερη δυνατή στοίχιση μεταξύ τμημάτων των δύο ακολουθιών. Ο αλγόριθμος, σε γενικές γραμμές, είναι πανομοιότυπος με αυτόν των Needleman & Wunsch με μόνη διαφορά ότι οποτεδήποτε η βαθμολογία $S_{k,m}$ πάρει τιμή μικρότερη του μηδενός, η τιμή της τίθεται ίση με το μηδέν. Άρα, η βασική εξίσωσή του είναι :

$$S_{k+1,m+1} = \max \begin{cases} 0 \\ S_{k,m} + s(a_{k+1}, b_{m+1}) \\ S_{k+1,m+1} = S_{k+1,m} - (\text{gap penalty}) \\ S_{k+1,m+1} = S_{k,m+1} - (\text{gap penalty}) \end{cases}$$

και η αρχικοποίηση των τιμών του πίνακα $S_{k,m}$ είναι :

$$S_{0,0} = 0$$
$$S_{k,0} = 0$$
$$S_{0,m} = 0.$$

Στο σχήμα 5.2 παρουσιάζεται ένα στιγμιότυπο τοπικής στοίχισης των ακολουθιών “GGATCGA” και “GAATTCAGTTA”.

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	1	0	0	0	0	0	5	1	0	0
G	0	5	2	0	0	0	0	0	5	2	0	0
A	0	1	10	7	3	0	0	5	1	2	0	5
T	0	0	6	7	12	8	4	1	2	6	8	4
C	0	0	2	3	8	9	13	9	5	2	4	5
G	0	5	1	0	4	5	9	10	14	10	6	2
A	0	1	10	6	2	1	5	14	10	11	7	11

(α)

```
GAATTCAG      GAATTCAG
|  |  ||  |    |  ||  |  |
GGA-TC-G      GCAT-C-G

GAATTC-A      GAATTC-A
|  |  ||  |    |  ||  |  |
GGA-TCGA      GCAT-CGA
```

(β)

Σχήμα 5.2: Στιγμιότυπο παραδείγματος τοπικής στοίχισης ακολουθιών : **(α)** πίνακας δυναμικού προγραμματισμού, και **(β)** οι τέσσερις εναλλακτικές στοιχίσεις που προκύπτουν

5.4.3 Ευριστικοί Αλγόριθμοι

Είναι αλγόριθμοι οι οποίοι αυξάνουν κατά πολύ την ταχύτητα της διαδικασίας στοίχισης επιτρέποντας έτσι την γρήγορη έρευνα των βάσεων δεδομένων για την αναζήτηση ομολογιών. Το κόστος αυτής της αύξησης της ταχύτητας, είναι ότι για τούς αλγορίθμους αυτούς, η εύρεση της βέλτιστης στοίχισης (είτε ολική, είτε τοπική) δεν είναι εγγυημένη. Το **BLAST** και το **FASTA** αποτελούν τους πιο συχνά χρησιμοποιούμενους ευριστικούς (heuristic) αλγορίθμους και βρίσκονται διαθέσιμοι στο διαδίκτυο. Τέτοιου είδους αλγόριθμοι, βρίσκουν μια λύση κοντινή στη βέλτιστη (close to the best one) γρήγορα και εύκολα.

5.4.3.1 Ο Αλγόριθμος FASTA

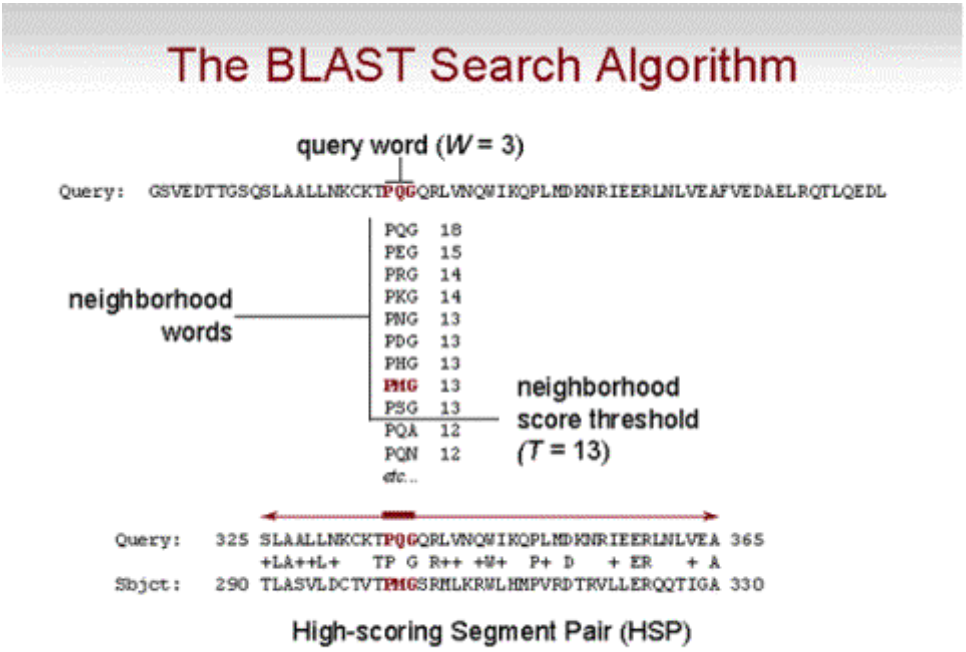
Είναι ο πρώτος ευριστικός (heuristic) αλγόριθμος ευρείας χρήσεως αναζήτησης ομοιοτήτων σε βιολογικές βάσεις. Συγκρίνει είτε μια πρωτεϊνική ακολουθία με μια άλλη πρωτεϊνική ακολουθία ή με μια πρωτεϊνική βάση δεδομένων, είτε μια DNA ακολουθία με μια άλλη ακολουθία DNA ή με μια DNA βάση.

Το πρόγραμμα **FASTA3** ψάχνει να βρει τις βέλτιστες τοπικές στοιχίσεις από την ανίχνευση της ακολουθίας για μικρές αντιστοιχίες που ονομάζονται “λέξεις” (words). Αρχικά, υπολογίζεται το σύνολο των τμημάτων (“init1”) στα οποία υπάρχουν πολλαπλές “λέξεις”. Τα αποτελέσματα διάφορων τμημάτων μπορούν να αθροιστούν για να παράγουν ένα “initn”. Μια βελτιστοποιημένη στοίχιση που περιλαμβάνει ανοίγματα (gaps) εμφανίζεται ως “opt’. Η ευαισθησία και η ταχύτητα της αναζήτησης είναι αντιστρόφως σχετιζόμενες και ελεγχόμενες από τη μεταβλητή “k-tup” που προσδιορίζει το μέγεθος μιας λέξης. Το FASTA υπολογίζει τη στατιστική σημαντικότητα επί “τόπου” (on the fly) από το σύνολο των δεδομένων (πιο ακριβές αλλά έχει προβλήματα αν το σύνολο των δεδομένων είναι πολύ μικρό).

5.4.3.2 Ο Αλγόριθμος BLAST

Ο αλγόριθμος **BLAST** χρησιμοποιείται για τη σύγκριση μιας ακολουθίας με μια βάση δεδομένων. Είναι ένας ευριστικός (heuristic) αλγόριθμος σύγκρισης ακολουθιών βελτιστοποιημένης ταχύτητας που χρησιμοποιείται για να ψάχνει σε βάσεις ακολουθιών την άριστη τοπική στοίχιση με μια αναζήτηση. Η αρχική αναζήτηση γίνεται για μια λέξη μήκους “W” (3 στο blastp) που σημειώνει score μεγαλύτερο από ένα προκαθορισμένο όριο (Threshold -“T”) όταν στοιχίζονται με την δοθείσα ακολουθία (query) και με ένα δεδομένο πίνακα υποκατάστασης (substitution matrix). Οι επιτυχείς λέξεις που έχουν score T ή μεγαλύτερο επεκτείνονται και προς τις δύο κατευθύνσεις σε μια απόπειρα να παραχθούν στοιχίσεις που να υπερβαίνουν το προκαθορισμένο κατώφλι (threshold) “S”. Οι περιοχές που ικανοποιούν αυτή τη συνθήκη ονομάζονται HSP (High-scoring Segment Pair). Η παράμετρος “T” καθορίζει την ταχύτητα και την ευαισθησία της αναζήτησης. Ενώ το BLAST υπολογίζει τις παραμέτρους της EVD (Extreme Value Distribution), από τις οποίες θα υπολογίσει τη στατιστική σημαντικότητα από εξομοιώσεις που έχει πραγματοποιήσει από πριν, το FASTA τις υπολογίζει από όλες τις άλλες ακολουθίες της βάσης δεδομένων και για αυτό το λόγο είναι και πιο αργό.

Στο σχήμα 5.1 παρουσιάζεται ένα στιγμιότυπο του αλγορίθμου BLAST. Στη συγκεκριμένη περίπτωση η τριπλέτα "PQG" έχει βρεθεί να στοιχίζεται στην "PMG", και καθώς χρησιμοποιείται ο πίνακας BLOSUM62, το score της στοίχισης είναι 13>T (P-P =7, G-G=6, Q-M=0). Κατόπιν αυτή η τριπλέτα θα επεκταθεί και προς τις δυο κατευθύνσεις για να δώσει τη μέγιστη τοπική στοίχιση. Τα HSPs που πληρούν αυτά τα κριτήρια θα αναφερθούν από το BLAST, υπό τον όρο ότι δεν υπερβαίνουν τον αριθμό των επιτρεπτών προς εμφάνιση ακολουθιών.



Σχήμα 5.3: Στιγμιότυπο του αλγόριθμου BLAST

5.4.4 Στατιστική Σημασία Στοιχίσεων

Ένα πολύ μεγάλο θέμα που προκύπτει στη στοίχιση ακολουθιών είναι η εύρεση της στατιστικής σημαντικότητας των αποτελεσμάτων. Συγκεκριμένα, μας ενδιαφέρει το πως μπορούμε να διαχωρίσουμε «τυχαία» ευρήματα από «σημαντικά». Βασιζόμενοι σε μια βέλτιστη στοίχιση μεταξύ δυο αλληλουχιών, θέλουμε να συνάγουμε την ύπαρξη (ή όχι) ομολογίας ανάμεσα στις αλληλουχίες (εάν, δηλαδή, αυτές οι αλληλουχίες έχουν όντως εξελικτική σχέση, και συνεπώς, ίσως και λειτουργική). Για να το επιτύχουμε αυτό, θα πρέπει να είμαστε σε θέση να εκτιμήσουμε κατά πόσο η ομοιότητα που δηλώνεται από τη βαθμολογία (S) της στοίχισης είναι στατιστικά σημαντική, δηλαδή, πόσο πιθανό ή απίθανο είναι να παρατηρήσουμε μια τέτοια βαθμολογία μεταξύ δυο αλληλουχιών οι οποίες δεν έχουν εξελικτική σχέση. Στην περίπτωση ολικών στοιχίσεων, αυτός ο στατιστικός έλεγχος μπορεί να πραγματοποιηθεί ως εξής (z-score test) :

- Επαναδιευθετούμε με τυχαίο τρόπο τη σειρά των αμινοξέων της κάθε αλληλουχίας.

- Στοιχίζουμε τις προκύπτουσες αλληλουχίες και σημειώνουμε την τιμή της προκύπτουσας βαθμολογίας, $S_{random,1}$

- Επαναλαμβάνουμε τη διαδικασία αυτή n φορές.

- Χρησιμοποιούμε τα $S_{random,1}$, $S_{random,2}$,…, $S_{random,n}$ για να υπολογίσουμε το μέσο όρο (μ_S) και την τυπική απόκλιση (σ_S) τους.

- Υπολογίζουμε πόσες τυπικές αποκλίσεις (σ_S) απέχει η παρατηρούμενη βαθμολογία (S) από το μέσο (μ_S) :

$$\text{z-score} = (S - \mu_S) / \sigma_S$$

  Όσο μεγαλύτερη είναι η τιμή του z-score τόσο μικρότερη είναι η πιθανότητα η παρατηρούμενη ομοιότητα να είναι αποτέλεσμα τυχαιότητας αντί για αποτέλεσμα εξελικτικής διαδικασίας.

5.4.5 Πίνακες Υποκαταστάσεων

Μια από τις πιο σημαντικές παραμέτρους που πρέπει να λαμβάνουμε υπ' όψη μας στη χρήση των προγραμμάτων τοπικής στοίχισης, είναι το είδος του πίνακα υποκατάστασης (substitution matrix) που χρησιμοποιούμε. Ανεξαρτήτως του αλγορίθμου που χρησιμοποιούμε, σε γενικές γραμμές, το μέγεθος το οποίο καθορίζει τη στοίχιση είναι το score, το οποίο παίρνει κάποια τιμή θετική ή αρνητική για κάθε αντιστοίχιση των αμινοξέων, από κάποιο 20×20 πίνακα υποκαταστάσεων. Οι πίνακες αυτοί [**PAM** (Point Accepted Mutation), **BLOSUM** (BLOcks SUbstitution Matrix), κλπ] εκφράζουν σε γενικές γραμμές την εξελικτική σχέση των 20 αμινοξέων, π.χ. η βαλίνη και η ισολευκίνη θα έχουν σχεδόν πάντα θετική συνεισφορά στο score, αν συναντηθούν σαν ζεύγος, ενώ η αργινίνη και η φενυλαλανίνη αρνητική. Πρέπει να τονιστεί ότι ο κάθε πίνακας είναι φτιαγμένος για να πραγματοποιεί καλύτερες στοιχίσεις, σε μια δεδομένη εξελικτική σχέση των πρωτεϊνών. Άλλος δηλαδή πίνακας ταιριάζει στην περίπτωση που έχουμε δυο πολύ ομόλογες πρωτεΐνες και άλλος για δυο πρωτεΐνες πολύ απομακρυσμένες εξελικτικά. Σε μια αναζήτηση όμως έναντι ολόκληρης της βάσης είναι καλύτερο να χρησιμοποιούνται πιο αποδεκτοί πίνακες, όπως ο BLOSUM62.

Παρακάτω δίνονται οι δυο πιο γνωστοί πίνακες υποκαταστάσεων, ο BLOSUM62, και ο PAM250, όπου βλέπουμε τα παρόμοια (εξελικτικά και φυσικοχημικά) αμινοξέα να δίνουν θετικό score αν συναντηθούν ως ζεύγος, ενώ τα διαφορετικά να δίνουν αρνητικό. Παρατηρούμε, επίσης, ότι παρ' όλο που δίνουν ανάλογα αποτελέσματα, αυτά δεν είναι εντελώς όμοια. Η διαφορά των δυο πινάκων φαίνεται από τα δύο μεγέθη Entropy (σχετική εντροπία - Kullback & Leibler) και Expected score (αναμενόμενο score ανά κατάλοιπο).

```
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected Score =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -2 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0  0 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1  0 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -2 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

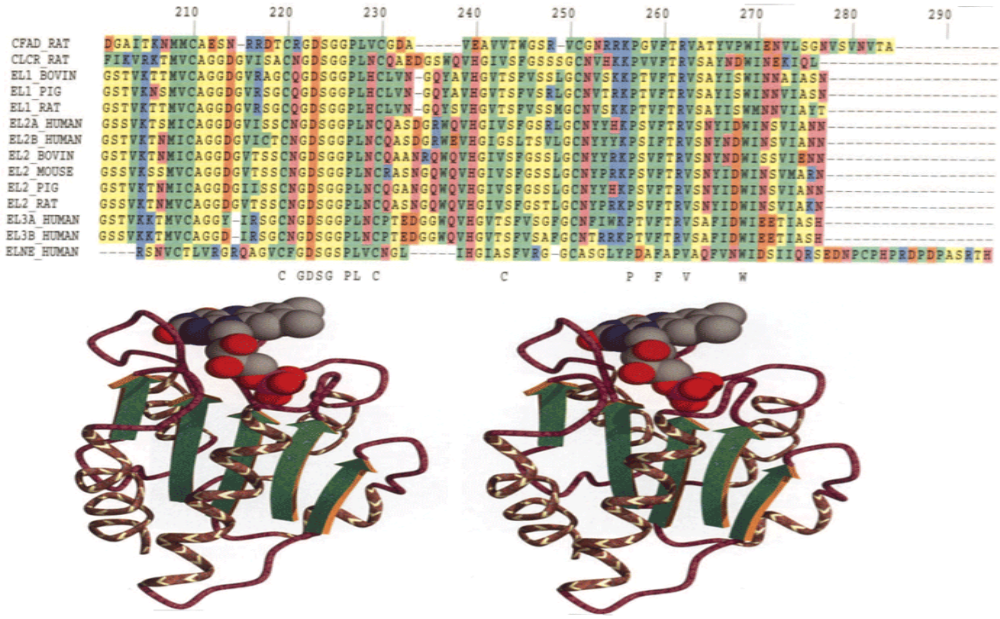
Σχήμα 5.4 : Ο πίνακας υποκαταστάσεων BLOSUM62

```
# This matrix was produced by "pam" Version 1.0.6 [28-Jul-93]
# PAM 250 substitution matrix, scale = ln(2)/3 = 0.231049
# Expected score = -0.844, Entropy = 0.354 bits
# Lowest score = -8, Highest score = 17
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  2 -2  0  0 -2  0  0  1 -1 -1 -2 -1 -1 -3  1  1  1 -6 -3  0  0  0  0 -8
R -2  6  0 -1 -4  1 -1 -3  2 -2 -3  3  0 -4  0  0 -1  2 -4 -2 -1  0 -1 -8
N  0  0  2  2 -4  1  1  0  2 -2 -3  1 -2 -3  0  1  0 -4 -2 -2  2  1  0 -8
D  0 -1  2  4 -5  2  3  1  1 -2 -4  0 -3 -6 -1  0  0 -7 -4 -2  3  3 -1 -8
C -2 -4 -4 -5 12 -5 -5 -3 -3 -2 -6 -5 -5 -4 -3  0 -2 -8  0 -2 -4 -5 -3 -8
Q  0  1  1  2 -5  4  2 -1  3 -2 -2  1 -1 -5  0 -1 -1 -5 -4 -2  1  3 -1 -8
E  0 -1  1  3 -5  2  4  0  1 -2 -3  0 -2 -5 -1  0  0 -7 -4 -2  3  3 -1 -8
G  1 -3  0  1 -3 -1  0  5 -2 -3 -4 -2 -3 -5  0  1  0 -7 -5 -1  0  0 -1 -8
H -1  2  2  1 -3  3  1 -2  6 -2 -2  0 -2 -2  0 -1 -1 -3  0 -2  1  2 -1 -8
I -1 -2 -2 -2 -2 -2 -2 -3 -2  5  2 -2  2  1 -2 -1  0 -5 -1  4 -2 -2 -1 -8
L -2 -3 -3 -4 -6 -2 -3 -4 -2  2  6 -3  4  2 -3 -3 -2 -2 -1  2 -3 -3 -1 -8
K -1  3  1  0 -5  1  0 -2  0 -2 -3  5  0 -5 -1  0  0 -3 -4 -2  1  0 -1 -8
M -1  0 -2 -3 -5 -1 -2 -3 -2  2  4  0  6  0 -2 -2 -1 -4 -2  2 -2 -2 -1 -8
F -3 -4 -3 -6 -4 -5 -5 -5 -2  1  2 -5  0  9 -5 -3 -3  0  7 -1 -4 -5 -2 -8
P  1  0  0 -1 -3  0 -1  0  0 -2 -3 -1 -2 -5  6  1  0 -6 -5 -1 -1  0 -1 -8
S  1  0  1  0  0 -1  0  1 -1 -1 -3  0 -2 -3  1  2  1 -2 -3 -1  0  0  0 -8
T  1 -1  0  0 -2 -1  0  0 -1  0 -2  0 -1 -3  0  1  3 -5 -3  0  0 -1  0 -8
W -6  2 -4 -7 -8 -5 -7 -7 -3 -5 -2 -3 -4  0 -6 -2 -5 17  0 -6 -5 -6 -4 -8
Y -3 -4 -2 -4  0 -4 -4 -5  0 -1 -1 -4 -2  7 -5 -3 -3  0 10 -2 -3 -4 -2 -8
V  0 -2 -2 -2 -2 -2 -2 -1 -2  4  2 -2  2 -1 -1 -1  0 -6 -2  4 -2 -2 -1 -8
B  0 -1  2  3 -4  1  3  0  1 -2 -3  1 -2 -4 -1  0  0 -5 -3 -2  3  2 -1 -8
Z  0  0  1  3 -5  3  3  0  2 -2 -3  0 -2 -5  0  0 -1 -6 -4 -2  2  3 -1 -8
X  0 -1  0 -1 -3 -1 -1 -1 -1 -1 -1 -1 -1 -2 -1  0  0 -4 -2 -1 -1 -1 -1 -8
* -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8 -8  1
```

Σχήμα 5.5 : Ο πίνακας υποκαταστάσεων PAM250

5.5 Πολλαπλή Στοίχιση Ακολουθιών

Η πολλαπλή στοίχιση ακολουθιών αποτελεί μία φυσική γενίκευση της στοίχισης για δύο συμβολοσειρές. Πιο συγκεκριμένα, έγκειται στην εύρεση της "βέλτιστης στοίχισης" περισσότερων από δύο, έστω k ακολουθιών, S={ S_1, S_2,…., $S_κ$}, δηλαδή, στην εύρεση της στοίχισης στην οποία οι k ακολουθίες ταιριάζουν περισσότερο μεταξύ τους. Η διαδικασία αυτή έχει αρκετές εφαρμογές στην βιοπληροφορική.



Σχήμα 5.6: Η πολλαπλή στοίχιση ακολουθιών καθίσταται σημαντική στην αναγνώριση της δομής των πρωτεϊνών.

Η πολλαπλή στοίχιση ακολουθιών χρησιμοποιείται:

- στην αναγνώριση και αναπαράσταση πρωτεϊνικών οικογενειών και υπερ-οικογενειών,
- στην αναπαράσταση των χαρακτηριστικών που μεταφέρονται στις ακολουθίες DNA ή στις πρωτεϊνικές ακολουθίες,
- στην αναπαράσταση της εξελικτικής ιστορίας (φυλογενετικά δέντρα) από ακολουθίες DNA ή πρωτεϊνών.

5.5.1 Το Πρόγραμμα CLUSTALW

Το **CLUSTALW** είναι το πιο διαδεδομένο πρόγραμμα **πολλαπλής στοίχισης (multiple alignment)** βιολογικών ακολουθιών. Το πρόγραμμα χρησιμοποιεί έναν ιδιαίτερα πολύπλοκο αλγόριθμο προοδευτικής στοίχισης (progressive alignment) για να κάνει σταδιακή στοίχιση πολλαπλών πρωτεϊνικών ή νουκλεοτιδικών (DNA) ακολουθιών. Όλες οι ακολουθίες πρέπει να είναι ένα αρχείο, η μία μετά την άλλη.

Το **CLUSTALX** είναι μια έκδοση του προγράμματος, το οποίο χρησιμοποιεί το περιβάλλον των X-Windows για να τρέχει το πρόγραμμα CLUSTALW. Το CLUSTALX δίνει την επιλογή δύο τρόπων στοίχισης ακολουθιών. Την πολλαπλή στοίχιση (multiple alignment mode) και την στοίχιση με βάση κάποιο profile (profile alignment mode).

Για να γίνει πολλαπλή στοίχιση ενός συνόλου ακολουθιών πρέπει να επιλεγεί ο τρόπος πολλαπλής στοίχισης (multiple alignment mode). Όλες οι ακολουθίες συγκρίνονται μεταξύ τους, επιτρέποντας την κατασκευή ενός δενδρογράμματος (dendrogram) που δείχνει την κατά προσέγγιση ομαδοποίηση των ακολουθιών λόγο της ομοιότητάς τους. Η τελική πολλαπλή στοίχιση των ακολουθιών γίνεται χρησιμοποιώντας ως οδηγό αυτό το δενδρόγραμμα.

Ο δεύτερος τρόπος στοίχισης ακολουθιών είναι η στοίχιση με βάση κάποιο profile (profile alignment mode). Υπάρχουν οι περιοχές στοιχείων των ακολουθιών (sequence data areas), που επιτρέπουν την στοίχιση τους. Τα profiles χρησιμοποιούνται επίσης για την προσθήκη μιας νέας ακολουθίας σε μια παλιά στοίχιση ή χρησιμοποιούν μια δευτεροταγή δομή για να κατευθύνουν τη διαδικασία της στοίχισης. Τα κενά (gaps) παλαιότερων στοιχίσεων δηλώνονται με το χαρακτήρα "-".

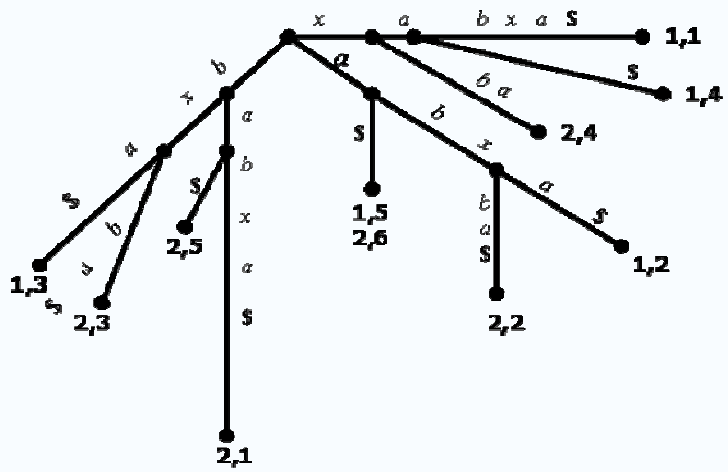
Το CLUSTAL μπορεί να εκτελέσει πολλαπλές στοιχίσεις για περισσότερες από 100 νουκλεοτιδικές (DNA) ή πρωτεϊνικές ακολουθίες, αποτελούμενες μέχρι και 5000 κατάλοιπα (περιλαμβανομένων των κενών στην τελική στοίχιση).

5.6 Μέγιστη κοινή υποσυμβολοσειρά δύο ακολουθιών

Η μεθοδολογία επίλυσης του προβλήματος υπολογισμού της μέγιστης κοινής υποσυμβολοσειράς δύο ακολουθιών στηρίζεται στην χρήση των **Γενικευμένων Δένδρων Προσφυμάτων** (Generalized Suffix Tree).

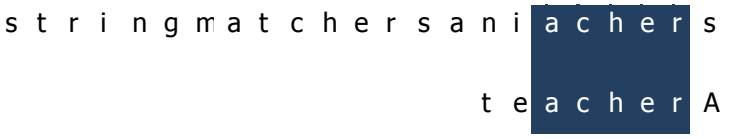
Ορισμός 5.3: *Το **Γενικευμένο Δένδρο Προσφυμάτων** (Generalized Suffix Tree), αποτελεί ένα Δένδρο Προσφυμάτων το οποίο αποθηκεύει όλα τα δυνατά προσφύματα ενός συνόλου συμβολοσειρών $S=\{S_1,S_2,...S_n\}$.*

Στο σχήμα 5.7 παρουσιάζεται ένα στιγμιότυπο υπολογισμού του γενικευμένου δένδρου προσφυμάτων των ακολουθιών x_1 = xabxa και x_2 = babxba.

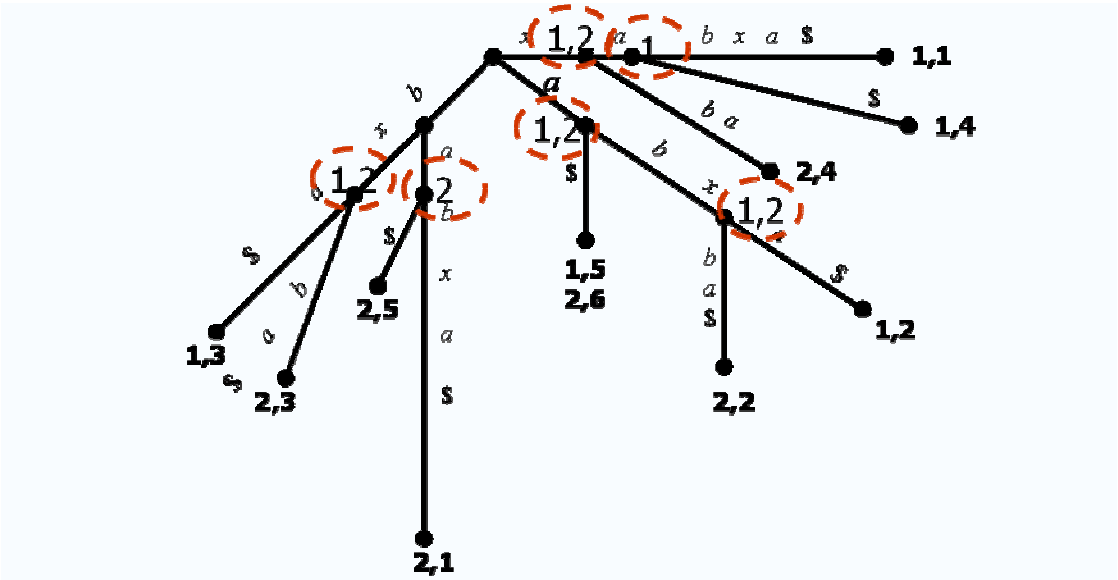


Σχήμα 5.7: Στιγμιότυπο γενικευμένου δένδρου επιθεμάτων των ακολουθιών x_1 = xabxa και x_2 = babxba

Αρχικά κατασκευάζουμε το γενικευμένο δένδρο προσφυμάτων για τις ακολουθίες εισόδου S_1, S_2. Στη συνέχεια, σημειώνουμε κάθε εσωτερικό κόμβο του δένδρου *u,* με "1" ή "2", αν εμπεριέχει στο υποδένδρο του *u*, κάποιο φύλλο που αναπαριστά κάποιο επίθεμα της ακολουθίας S_1 ή S_2. Η ετικέτα μονοπατιού - path label, κάθε εσωτερικού κόμβου που σημειώνεται ταυτόχρονα με "1" και "2", αποτελεί μια κοινή υποσυμβολοσειρά των δυο ακολουθιών S_1 και S_2. Στο σχήμα 5.9 παρουσιάζεται ένα στιγμιότυπο υπολογισμού της μέγιστης κοινής υποσυμβολοσειράς των ακολουθιών x_1 = xabxa και x_2 = babxba.



Σχήμα 5.8: Μέγιστη κοινή υποσυμβολοσειρά δύο ακολουθιών



Σχήμα 5.9: Στιγμιότυπο υπολογισμού της μέγιστης κοινής υποσυμβολοσειράς τους των ακολουθιών x_1 = xabxa και x_2 = babxba

5.7 DNA Contamination Problem

Το DNA Contamination Problem ορίζεται ως εξής:

Ορισμός 5.4: *Για μια δοσμένη ακολουθία DNA S_1, που έχει πρόσφατα απομονωθεί και ταυτοποιηθεί, και μια ήδη γνωστή ακολουθία S_2, (επιμέρους τμήματα που πιθανά έχουν μολυνθεί), αναζητούμε όλες τις υποσυμβολοσειρές της S_2 που εμφανίζονται στην S_1, με μήκος μεγαλύτερο από λ.*

Η μεθοδολογία επίλυσης του προβλήματος αυτού έγκειται στην κατασκευή του **Γενικευμένου Δένδρου Προσφυμάτων** για τις ακολουθίες S_1 και S_2, και εν συνεχεία, στην αναφορά όλων των κόμβων με βάθος *string-depth(u)* ≥ λ.

5.8 Εύρεση Κοινών Μοτίβων σε δύο ή περισσότερες Βιολογικές Ακολουθίες

Το Πρόβλημα της Εύρεσης κοινών μοτίβων σε δύο ή περισσότερες ακολουθίες ορίζεται ως εξής:

Ορισμός 5.5: *Για ένα σύνολο K ακολουθιών με συνολικό μήκος Σ(|K|)= n, και έναν ακέραιο k, (2<k<K), ορίζουμε ως λ(k), το μήκος του μέγιστου μοτίβου που εμφανίζεται σε τουλάχιστον k υποσυμβολοσειρές. Το πρόβλημα ανάγεται στον υπολογισμό όλων των δυνατών τιμών του λ(k) και λύνεται σε γραμμικό χρόνο O(n), ως προς το μήκος των ακολουθιών εισόδου.*

Ως παράδειγμα, ας θεωρήσουμε το σύνολο ακολουθιών K={sandollar, sandlot, handler, grand, pantry}. Στον ακόλουθο πίνακα φαίνονται τα κοινά μοτίβα των ακολουθιών που απαρτίζουν το σύνολο K:

k	*λ(k)*	*μοτίβο*
2	*4*	*sand*
3	*3*	*and*
4	*3*	*and*
5	*2*	*an*

Σχήμα 5.10: Κοινά μοτίβα των ακολουθιών {sandollar, sandlot, handler, grand, pantry}

5.9 Εύρεση Επαναλήψεων σε Βιολογικές Ακολουθίες

Οι επαναλήψεις σε βιολογικές ακολουθίες κατηγοριοποιούνται στις εξής τρεις βασικές κατηγορίες:

- επαναλήψεις περιορισμένου μήκους που εμφανίζονται σε τοπικό επίπεδο, και των οποίων η λειτουργία είναι γνωστή (**συμπληρωματικά παλίνδρομα** σε ακολουθίες DNA & RNA που ρυθμίζουν τη μετεγγραφή του DNA, **εμφωλευμένα συμπληρωματικά παλίνδρομα** σε ακολουθίες tRNA),

- επαναλήψεις περιορισμένου μήκους που εμφανίζονται σε όλο το μήκος της ακολουθίας, και των οποίων η λειτουργία δεν είναι απόλυτα γνωστή (**συνεχόμενες επαναλήψεις – tandem repeats, δορυφορικά τμήματα DNA – satellite DNA, (micro & mini satellite DNA**)),

- δομημένες επαναλήψεις μεγάλου μήκους των οποίων η λειτουργία δεν έχει προσδιοριστεί (**SINE-Short Interspersed Nuclear Sequences (π.χ.** *Alu family)*, **LINE-Long Interspersed Nuclear Sequences**)

Ορισμός 5.6: *Ένα **παλίνδρομο – palindrome** αποτελεί την επαναλαμβανόμενη εμφάνιση της υποσυμβολοσειράς που διαβάζεται ως ίδια και προς τις 2 κατευθύνσεις (από αριστερά προς τα δεξιά και από δεξιά προς τα αριστερά), π.χ. xyaayx.*

Ορισμός 5.7: *Ένα παλίνδρομο σε μια ακολουθία DNA ή RNA, ονομάζεται **συμπληρωματικό παλίνδρομο – complemented palindrome,** αν προκύπτει από την αντικατάσταση όλων των χαρακτήρων από την αρχή έως τη μέση με τις αντίστοιχες συμπληρωματικές βάσεις, π.χ. agctcgcgagct.*

Βιβλιογραφία Κεφαλαίου

[1] Lok-Lam, Cheng David W. Cheung, Siu-Ming Yiu, *Approximate String Matching in DNA Sequences*, Department of Computer Science and Information Systems, University of Hong Kong.

[2] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[3] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002

[4] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[5] Περδικούρη Αικατερίνη, Μακρής Χρήστος, *Εισαγωγή στην Βιοπληροφορική*.

[6] Eric C. Rouchka, *Pattern Matching Techniques and Their Applications to Computational Molecular Biology - A Review*.

[7] www.bioalgorithms.info, *An Introduction to Bioinformatics Algorithms - Combinatorial Pattern Matching*.

[8] http://en.wikipedia.org/wiki/Main_Page

[9] http://pubs.acs.org/subscribe/journals/mdd/v05/i03/html/03sites.html

Βιβλιογραφία

[1] Maxime Crochemore, Wojciech Rytter, *Jewels of Stringology*, Cambridge: World Scientific Pub.,2003.

[2] D. Gusfield, *Algorithms on Strings, Trees and Sequencies:* Computer Science and Computational Biology, Cambridge, Cambridge University Press, 1977.

[3] Π.Δ. Μποζάνης, *Αλγόριθμοι: Σχεδιασμός και Ανάλυση*, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Gonzalo Navarro, Mathieu Raffinot, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences,* Cambridge University Press, 2002.

[5] Paolo Ferragina, *String Algorithms and Data Structures*, University of Pisa, Italy.

[6] Yu Jiangsheng, *String Matching*, Institute of Computational Linguistics, Peking University, 2002.

[7] James Worrell, *String-matching Algorithms,* Oxford University Computing Laboratory, UK.

[8] http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/indexen.htm

[9] Tatsuya Akutsu, *Approximate string matching with don't care characters*, Department of Computer Science, Gunma University, Japan.

[10] A. Amir, M. Lewenstein, E. Porat, *Faster Algorithms for String Matching with k Mismatches,* June 28, 2002.

[11] Ricardo A. Baeza-Yates and Chris H. Perleberg, *Fast and Practical Approximate String Matching,* Department of Computer Science, University of Chile.

[12] Ricardo A. Baeza-Yates and Gonzalo Navarro, *Fast Approximate String Matching,* Department of Computer Science, University of Chile.

[13] Costas S. Iliopoulos and M. Sohel Rahman, *Pattern Matching Algorithms with Don't Cares,* Department of Computer Science, King's College London, England.

[14] Adam Kalai, *Efficient Pattern-Matching with Don't Cares.*

[15] Ricardo A. Baeza-Yates and Gaston H. Gonnet, *Fast String Matching with Mismatches.*

[16] Gonzalo Navarro, *A Guided Tour to Approximate String Matching*, Department of Computer Science, University of Chile.

[17] Περδικούρη Αικατερίνη, Μακρής Χρήστος, *Εισαγωγή στην Βιοπληροφορική*.

[18] Eric C. Rouchka, *Pattern Matching Techniques and Their Applications to Computational Molecular Biology - A Review*.

[19] www.bioalgorithms.info, *An Introduction to Bioinformatics Algorithms - Combinatorial Pattern Matching*.

[20] http://wikipedia.org/

[21] http://www.google.com/

[22] http://pubs.acs.org/subscribe/journals/mdd/v05/i03/html/03sites.html