Υπογραμμικοί Αλγόριθμοι για το Gap Edit Distance Πρόβλημα

Κωνσταντίνος Μάλαμας

Διπλωματική Εργασία

Επιβλέπων: Αναπλ. Καθ. Λουκάς Γεωργιάδη

Ιωάννινα, Οκτώβριος 2022

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους γονείς μου και τους φίλους μου, που με στήριξαν καθ' όλη την διάρκεια των σπουδών μου, καθώς και τον καθηγητή Λουκά Γεωργιάδη, για την πολύτιμη καθοδήγηση του στην εκπόνηση της διπλωματικής εργασίας.

Περίληψη

Το edit distance είναι ένας τρόπος ποσοτικοποίησης της ανομοιότητας δύο συμβολοσειρών, μετρώντας τον μικρότερο αριθμό πράξεων εισαγωγής, διαγραφής και αντικατάστασης συμβόλων που είναι απαραίτητες για την μετατροπή της μίας συμβολοσειράς στην άλλη. Έχει εφαρμογή σε πεδία όπως της υπολογιστικής βιολογίας, αναγνώρισης προτύπων, ανάκτησης δεδομένων κ.α. Σε αυτήν την διπλωματική εργασία, ο στόχος μας είναι η υλοποίηση αλγορίθμων που υπολογίζουν την τιμή του edit distance ή ελέγχουν αν η τιμή αυτή βρίσκεται εντός ή εκτός ενός διαστήματος τιμών (gap edit distance). Επιπλέον θα παρουσιάσουμε τα αποτελέσματα των πειραματικών δοκιμών που εκτελέσαμε, προκειμένου να συγκρίνουμε τις αποδόσεις των αλγορίθμων σε ταχύτητα και ορθότητα.

Αρχικά θα εξετάσουμε τον αλγόριθμο δυναμικού προγραμματισμού των Robert A. Wagner και Michael J. Fischer, που υπολογίζει ακριβώς το edit distance δύο συμβολοσειρών. Έπειτα θα συνεχίσουμε με τον υπογραμμικό αλγόριθμο των Elazar Goldenberg, Robert Krauthgamer και Barna Saha που δίνει μια προσεγγιστικό αποτέλεσμα στο πρόβλημα του gap edit distance. Θέλοντας να έχουμε μια ολοκληρωμένη εικόνα του αλγορίθμου, θα εξετάσουμε αρχικά τις υλοποιήσεις δύο αλγορίθμων πρόδρομών του, και μετά θα προχωρήσουμε στην τελική υλοποίηση του.

Λέξεις Κλειδιά: Edit Distance, Gap Edit Distance, Δυναμικός Προγραμματισμός, Υπογραμμικοί Αλγόριθμοι, Πειράματα

Abstract

The edit distance is a way of quantifying how similar two strings are to one another by counting the minimum number of character insertions, deletions, and substitutions required to transform one string into the other. The objective of this diploma thesis is the implementation of algorithms that compute the value of edit distance or test if this value lies inside or outside of a given interval (gap edit distance). Moreover, we are going to present the results of experimental tests we conducted, in order to compare their performance in terms of speed and accuracy.

At first, we will focus on the dynamic programming algorithm of Robert A. Wagner and Michael J. Fischer, that computes the precise edit distance of the two strings. Afterwards, we will continue by looking into the sublinear algorithm by Elazar Goldenberg, Robert Krauthgamer and Barna Saha that gives an approximate result to the gap edit distance problem. In order to have a clear view of the algorithm we are going to implement and examine two precursory algorithms, and after that we will proceed to the implementation of the final algorithm

Keywords: Edit Distance, Gap Edit Distance, Dynamic Programming, Sublinear Algorithms, Experiments

Πίνακας Περιεχομένων

Κεφάλαιο 1.

Εισαγωγή

1.1 Το πρόβλημα Edit Distance

Το Edit Distance, είναι μια μετρική συμβολοσειρών που χρησιμοποιήθηκε αρχικά από τον μαθηματικό Vladimir Levenshtein το 1965[1]. Στην σύγχρονη εποχή έχει βρει εφαρμογές σε πολλά πεδία, όπως της υπολογιστικής γεωμετρίας, αναγνώριση προτύπων, επεξεργασία κειμένου, ανάκτησης πληροφοριών κ.α. Για να υπολογίσουμε το edit distance μεταξύ δύο συμβολοσειρών A και B, όπου στο εξής θα συμβολίζουμε ως $\Delta_e(A, B)$, αρκεί να βρούμε το ελάχιστο πλήθος πράξεων εισαγωγής, διαγραφής και αντικατάστασης χαρακτήρων, προκειμένου να μετατρέψουμε την μία συμβολοσειρά στην άλλη.

Παράδειγμα 1.1

A: “cat”, B: “hat”

A: substitute A_3 = ‘c’ with ‘h’

A: “cat” ⇒ “hat”

A ⇒ B

$\Delta_e(A, B) = 1$

Παράδειγμα 1.2

A: “food”, B: “floor”

A: A_2 insert ‘l’

A: “food” ⇒ “flood”

A: substitute A_5 = ‘r’ with ‘d’

A: “flood” ⇒ “floor”

$\Delta_e(A, B) = 2$

Στα παραδείγματα 1.1 και 1.2 παρατηρούμε δύο απλές μετατροπές συμβολοσειρών από Α σε Β. Στο παράδειγμα 1.1 είναι απαραίτητη μόνο μία πράξη επεξεργασίας (αντικατάσταση), οπότε το edit distance μεταξύ των συμβολοσειρών “cat” και “hat” είναι ίσο με ένα. Αντίστοιχα στο παράδειγμα 1.2 η μετατροπή της συμβολοσειράς “food” σε “floor” απαιτεί μία εισαγωγή συμβόλου και μία αντικατάσταση, οπότε το edit distance μεταξύ τους είναι ίσο με δύο. Και στις δύο περιπτώσεις απορρίψαμε ακολουθίες πράξεων για τις οποίες το μονοπάτι από την μια συμβολοσειρά στην άλλη απαιτεί περισσότερες πράξεις, καθώς αυτό θα ήταν πιο ακριβό, και μας ενδιαφέρουν μόνο τα μονοπάτια με ελάχιστο πλήθος πράξεων.

1.2 Πολυπλοκότητα και υπογραμμικοί αλγόριθμοι

Λόγω των αμέτρητων εφαρμογών του edit distance, έχει γίνει επιτακτική η ανάγκη υπολογισμού του edit distance σε γρήγορο χρόνο. Ένας αλγόριθμος δυναμικού προγραμματισμού, όπως αυτός των Wagner-Fischer, μπορεί να λύσει το πρόβλημα σε χρόνο $O(n^2)$ για συμβολοσειρές ίδιου μεγέθους. Όμως για πολύ μεγάλα δεδομένα ένας αλγόριθμος τετραγωνικής πολυπλοκότητας είναι τουλάχιστον απαγορευτικός. Μέσα στα χρόνια έχουν σχεδιαστεί αλγόριθμοι που λύνουν το πρόβλημα σε γρηγορότερους χρόνους υπό συγκεκριμένες συνθήκες.

Στην περίπτωση του αλγόριθμου των Goldenberg-Krauthgamer-Saha, ο αλγόριθμος έχει υπογραμμικό χρόνο εκτέλεσης. Υπογραμμικός είναι κάποιος αλγόριθμος του οποίου ο χρόνος εκτέλεσης, f(n), αυξάνεται πιο αργά από ότι το μέγεθος n του προβλήματος, $\lim_{n\to\infty}\left(\frac{n}{f(n)}\right)=0$, αλλά μας δίνει μόνο μια προσεγγιστική ή πιθανώς σωστή απάντηση. Δεδομένου ότι πολλές από τις εφαρμογές του edit distance απαιτούν τον γρήγορο έλεγχο του αν η τιμή του είναι μικρότερη ενός χαμηλού κατωφλίου t, ο αλγόριθμος μπορεί να κάνει αυτόν τον υπολογισμό σε γρήγορο χρόνο, χρησιμοποιώντας τεχνικές δειγματοληψίας. Συγκεκριμένα μας λύνει το t vs $f(t) = t^2$ gap πρόβλημα σε χρόνο $O\left(\frac{n}{t}+t^3\right)$, με μία πιθανότητα αποτυχίας.

Κεφάλαιο 2. Υπολογισμός Edit Distance με Δυναμικό Προγραμματισμό

Πρόβλημα: Δοθέντος δύο πεπερασμένων συμβολοσειρών Α και Β, θέλουμε να υπολογιστεί το ελάχιστο πλήθος πράξεων εισαγωγής, διαγραφής και αντικατάστασης, προκείμενου η μία συμβολοσειρά να προκύψει στην άλλη. Έστω |A| και |B| ο αριθμός των συμβόλων στις συμβολοσειρές Α και Β αντίστοιχα, A_i το i-οστο σύμβολο της συμβολοσειράς και Λ η κενή συμβολοσειρά.

2.1 Αλγόριθμος Wagner-Fisher

Για να εξηγήσουμε τον αλγόριθμο είναι σημαντικό να μιλήσουμε για τα τις πράξεις επεξεργασίας κειμένου και τα κόστη τους. Πράξη επεξεργασίας (edit operation) είναι ένα ζευγάρι $(a, b) \neq (\Lambda,\Lambda)$ συμβολοσειρών με μήκος μικρότερο ή ίσο με 1 και γράφεται ως $a \rightarrow b$. Η εφαρμογή μιας ή παραπάνω πράξεων σε μία συμβολοσειρά έχει ως αποτέλεσμα την μετατροπή της, $A \Rightarrow B$. Καλούμε $a \rightarrow b$:

- πράξη εισαγωγής, αν $a = \Lambda$ και $b \neq \Lambda$, με κόστος $cost(a \rightarrow b) = 1$
- πράξη διαγραφής, αν $a \neq \Lambda$ και $b = \Lambda$, με κόστος $cost(a \rightarrow b) = 1$
- πράξη αντικατάστασης, αν $a \neq \Lambda$ και $b \neq \Lambda$, με κόστος $cost(a \rightarrow b) = 0$ αν $a = b$, και με κόστος $cost(a \rightarrow b) = 1$ αν $a \neq b$.

Έστω S μια αλληλουχία πράξεων $s_1, s_2, \ldots s_m$ από την οποία προκύπτει $A \Rightarrow B$. Και Έστω $cost(S) = cost(s_1) + cost(s_2) + \ldots + cost(s_m)$, ενώ $\Delta_e(A, B)$ μια συνάρτηση που μας δίνει το edit distance. Τότε $\Delta_e(A, B) = \min(cost(S))$.

Πίνακας 2.1

		f	o	o	d
	0	1	2	3	4
f	1	0	1	2	3
l	2	1	1	2	3
o	3	2	1	1	2
o	4	3	2	1	2
r	5	4	3	2	2

Ο αλγόριθμος λειτουργεί με την μέθοδο δυναμικού προγραμματισμού. Η λειτουργία του βασίζεται στην λογική ότι μπορούμε να αποθηκεύσουμε σε ένα δυσδιάστατο πίνακα το edit distance κάθε προθήματος των δύο συμβολοσειρών. Ξεκινώντας από το πρώτο στοιχείο συμπληρώνουμε όλον τον πίνακα με την τιμή edit distance των κάθε προθημάτων. Για να βρούμε το edit distance του κάθε προθήματος, αρκεί να συγκρίνουμε το edit distance των γειτονικών στοιχείων, συν της βάρος της πράξης που εκτελούν, μεταξύ τους. Το στοιχείο υιοθετεί το μικρότερο από αυτά (edit distance + βαρος) και το αποθηκεύει ως το edit distance του προθήματος που αντιστοιχεί. Κάθε υιοθέτηση από τα αριστερά προς τα δεξιά αναπαριστά την πράξη της εισαγωγής, από πάνω προς τα κάτω την πράξη της διαγραφής και από διαγώνια πάνω αριστερά προς διαγώνια κάτω δεξιά την πράξη της αντικατάστασης. Υπενθυμίζουμε ότι οι πράξεις έχουν βάρους 1, εκτός της πράξης της αντικατάστασης όπου αν a = b, τότε η πράξη έχει βάρος 0. Επαναλαμβάνουμε για όλα τους πιθανούς συνδυασμούς προθημάτων και η τελευταία τιμή που θα προκύψει είναι το τελικό edit distance των δύο συμβολοσειρών.

Για παράδειγμα στον Πίνακα 2.1 έχουμε οπτικοποιήσει τον υπολογισμό του edit distance μεταξύ της συμβολοσειράς “floor” και “food”. Στο κάθε τετράγωνο αποδίδεται το edit distance των προθημάτων των συμβολοσειρών που έχουμε στην πρώτη γραμμή και στην πρώτη στήλη αντίστοιχα. Το κάθε τετράγωνο συγκρίνει τα γειτονικά τετράγωνα πάνω, αριστερά και διαγώνια αριστερά πάνω, προσθέτοντας το αντίστοιχο βάρος, και υιοθετεί το πιο φθηνό από τα τρία. Τα μπλε τετράγωνα είναι αυτά που εν τέλη αποδίδουν το edit distance στο τελευταίο, δημιουργώντας μια αλυσίδα υιοθεσίας από το στοιχείο (0, 0), δηλαδή το κενό πρόθημα, ως την ολοκλήρωση της συμβολοσειρά (|A|, |B|). Μπορούμε να παρατηρήσουμε ότι σε αυτό το παράδειγμα προκειμένου να προκύψει η φθηνότερη τιμή edit distance ξεκινάμε κάνοντας μία πράξη αντικατάστασης, συνεχίζουμε με μία διαγραφής και τρεις ακόμα αντικατάσταση για να καταλήξουμε ότι Δ_e(“floor”, “food”) = 2.

Αλγόριθμος 2.1 Wagner-Fisher Algorithm

INPUT: String A, B

01: declare D[0…|A|, 0…|B|]

02: D[0,0] = 0

03: for i = 1 to |A| do D[i, 0] = D[i – 1, 0] + cost($A_i \rightarrow \Lambda$)

04: for j = 1 to |B| do D[0, j] = D[0, j - 1] + cost($\Lambda \rightarrow B_j$)

05: for i = 1 to |A| do

06: for j = 1 to |B| do

07: m_1 = D[i – 1, j - 1] + cost($A_i \rightarrow B_j$)

08: m_2 = D[i – 1, j] + cost($A_i \rightarrow \Lambda$)

09: m_3 = D[i, j – 1] + cost($\Lambda \rightarrow B_i$)

10: D[i, j] = min(m_1, m_2, m_3)

11: end for

12: end for

13: return D[|A|, |B|]

2.2 Αλγόριθμος Shortest Path in a Grid Graph

Πρόκειται για τον πρώτο από τους δύο προκαταρκτικούς αλγόριθμους για να ακολουθήσει αργότερα ο τελικός υπογραμμικός αλγόριθμος των Goldenberg-Krauthgamer-Saha.

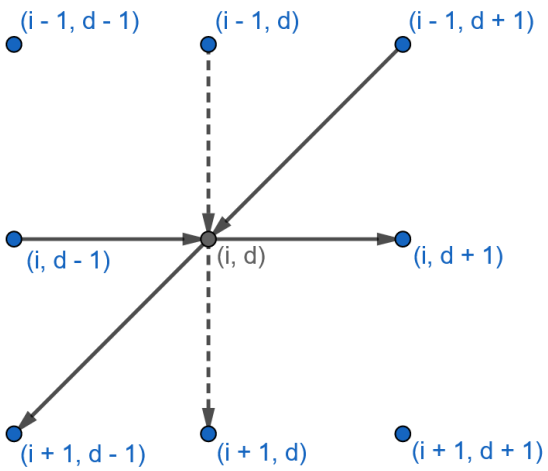
Με την εισαγωγή δύο συμβολοσειρών A, B $\in \{0,1\}^n$, θα θεωρήσουμε ένα κατευθυνόμενο γράφημα σε μορφή πλέγματος (grid graph) το οποίο θα συμβολίζουμε ως $G_{A,B}$. Επιπλέον, από εδώ και στο εξής (i, d) καλούμε i την γραμμή και d την διαγώνιο μίας κορυφής. Το μέγεθος του grid graph είναι [0..n] × [−n..n], και έχει τις εξής ακμές με βάρος/κόστος (εφόσον δεν βρίσκονται σε κάποια άκρη του γραφήματος):

i) Ακμές διαγραφής: (i, d) → (i+1, d-1), με κόστος 1 που αντιστοιχεί σε διαγραφή χαρακτήρα

ii) Ακμές εισαγωγής: (i, d) → (i, d+1), με κόστος 1 που αντιστοιχεί σε εισαγωγή χαρακτήρα

iii) Ακμές ομοιότητας/αντικατάστασης: (i, d) → (i+1, d), με βάρος 0 ή 1 ανάλογα με το αν ισχύει η συνθήκη $A_{i+1}=B_{i+d+1}$. Αυτές οι ακμές αντιστοιχούν σε ομοιότητα ή αντικατάσταση χαρακτήρα.

Εικόνα 2.1



Κάθε κορυφή έχει κόστος c(i,d) σύμφωνα με το συντομότερο μονοπάτι από το (0, 0) έως το (i, d).Μπορούμε να υπολογίσουμε το κόστος κάθε κορυφής επεξεργάζοντας τις κορυφές ανά γραμμή, από -n έως n. Συγκεκριμένα, για να υπολογίσουμε το κόστος μίας κορυφής (i, d), αρκεί να ελέγξουμε μόνο το κόστος των κορυφών των οποίων οι ακμές εισέρχονται σε αυτήν, δηλαδή τις κορυφές (i − 1, d), (i − 1, d + 1) και (i, d − 1).

Το μονοπάτι για κάθε κορυφή έχει 1-1 συσχέτιση με ακολουθία πράξεων επεξεργασίας στις A και B συμβολοσειρές. Κατά επέκταση το συντομότερο μονοπάτι που προκύπτει ακολουθώντας τον αλγόριθμο, αποτελεί το ελάχιστο αριθμό πράξεων που χρειαζόμαστε για την μετατροπή της μίας συμβολοσειράς στην άλλη. Άρα το κόστος που προκύπτει στην κορυφή sink (n, 0) ακολουθώντας το συντομότερο μονοπάτι από το την κορυφή source (0, 0), είναι το edit distance, δηλαδή $\Delta_e(|A|, |B|) = c(n,0)$.

Αλγόριθμος 3.1 shortest-path-in-a-grid-graph Algorithm

INPUT: String A, B (|A| = |B| = n)

01: declare D[0…n, -n..n]

02: D[0, 0] = 0

03: for i = 0 to n do

04: for d = -n to n do

05: $m_1 = D[i - 1, d] + cost(A_i \rightarrow B_{i+d})$

06: $m_2 = D[i - 1, d+1] + cost(A_i \rightarrow \Lambda)$

07: $m_3 = D[i, d - 1] + cost(\Lambda \rightarrow B_{i+d})$

08: $D[i , d] = \min(m_1, m_2, m_3)$

09: end for

10: end for

11: print D[n,0]

Note: Ignore null vertices and null edges

Κεφάλαιο 3. Υπογραμμικοί Αλγόριθμοι για το Gap Edit Distance Πρόβλημα

Σε αυτό το κεφάλαιο μελετάμε το εξής πρόβλημα. Δοθέντων δύο πεπερασμένων συμβολοσειρών A και B, θέλουμε να προσδιορίσουμε αν το edit distance μεταξύ των συμβολοσειρών αυτών βρίσκεται εντός ή εκτός ενός διαστήματος τιμών το οποίο εξαρτάται από μια παράμετρο t.

3.1 Αλγόριθμος Shortest Path in a Sampled Grid Graph

Τώρα θα περιγράψουμε έναν τυχαιοποιημένο αλγόριθμο, οποίος με είσοδο δύο συμβολοσειρές και μία παράμετρο t μπορεί με μεγάλη πιθανότητα να μας δώσει απάντηση στο αν $\Delta_e(A, B) \leq t$ ή $\Delta_e(A, B) = \Omega(t2)$. Ο αλγόριθμος έχει ασύμμετρη πολυπλοκότητα στην εξέταση των δύο συμβολοσειρών, δηλαδή ελέγχει δείγματα της συμβολοσειράς A με ρυθμό $\frac{\log n}{t}$, αλλά μπορεί να χρειαστεί να ελέγξει ολόκληρη τη συμβολοσειρά B.

Ο αλγόριθμος αυτός είναι μία καινούρια εκδοχή του προηγούμενου αλγορίθμου, με εφαρμογή δειγματοληψίας στο grid graph. Αρχικά ο αλγόριθμος επιλέγει ένα σύνολο $S \subseteq [n]$, όπου κάθε γραμμή i μπορεί να ενταχθεί στο σύνολο με πιθανότητα $\frac{\log n}{t}$, και εντάσσει την γραμμή 0. Έστω $s = |S|$ και δηλώνουμε τις γραμμές στο S ως $0 = i_i < \ldots < i_s$. Επίσης έστω ότι G_S είναι ένα grid graph με πλέγμα

S × [−t..t] ∪ {(n, 0)}, στο οποίο ενώνουμε κάθε κορυφή (i_j, d), όπου i_j ∈ S and d ∈ [−t..t], με ακμές προς: α) (i_j+1, d − 1) με βάρος 1, β) (i_j , d + 1) με βάρος 1 και, γ) (i_j+1, d), με βάρος 1 αν $A_{i_j}+1 \neq B_{i_j+d+1}$ ή βάρος 0 αν $A_{i_j}+1=B_{i_j+d+1}$. Τέλος ενώνουμε κάθε κορυφή (i_s, d) της τελευταίας γραμμής του grid graph με την sink (n, 0) κορυφή, με βάρος |d|.

Αφού ο αλγόριθμος κατασκευάσει το G_S, τότε υπολογίζει το κοντινότερο μονοπάτι από το (0, 0) στο (n, 0). Ο αλγόριθμος για Δe(A, B) ≤ t τυπώνει “close” με πιθανότητα 1, ενώ για Δe(A, B) > $6t^2$ τυπώνει “far” με πιθανότητα 2/3.

Αλγόριθμος 3.2 shortest-path-in-a-sampled-grid-graph Algorithm

INPUT: String A, B, t

01: pick random set S from [0 …. n] with probability (logn)/t for each row

02: declare D[0…|S|, -t…t]

03: D[0, 0] = 0

04: for i = 0 to |S| do

05: for d = -t to t do

06: $m_1 = D[i - 1, d] + cost(A_{S[i]+1} \rightarrow B_{S[i]+d+1})$

07: $m_2 = D[i - 1, d+1] + cost(A_{S[i]} \rightarrow \Lambda)$

08: $m_3 = D[i, d - 1] + cost(\Lambda \rightarrow B_{S[i]+d})$

09: $D[i, d] = \min(m_1, m_2, m_3)$

10: end for

11: end for

12: $D[n, 0] = \min(\text{for } d = -t \text{ to } t \text{ do } m_d = D[|S|, d] + |d|)$

13: if D[n, 0] ≤ t do print “close”

14: else do print ”far”

Note: Ignore null vertices and null edge

3.2 Αλγόριθμος Sublinear Algorithm for Quadratic Gap

Αφού εξετάσαμε τις προηγούμενες περιπτώσεις αλγορίθμων, πλέον μπορούμε να συνεχίσουμε με τον τελικό υπογραμμικό αλγόριθμο της διπλωματικής εργασίας. Η λειτουργία του είναι υβριδική ως προς την επεξεργασία των γραμμών, καθώς ανάλογα με τις συνθήκες ο αλγόριθμος μεταβαίνει από ομοιόμορφα δειγματοληπτική επεξεργασία γραμμών σε συνεχή. Επίσης ακολουθείται μια τεχνική κατά την οποία πολλές φορές μας επιτρέπεται να προσπεράσουμε ορισμένες κορυφές και να κάνουμε πιο επιλεκτικό έλεγχο.

Για τις ανάγκες του αλγορίθμου, πρώτα θα προχωρήσουμε στον ορισμό κάποιων ιδιοτήτων των κορυφών, καθώς και στην επεξήγηση της εναλλαγής λειτουργίας του αλγορίθμου που προαναφέραμε.

3.2.1 Ορισμοί

Dominated κορυφές Έστω (i, d) μία κορυφή στο $G_{A,B}$ με κόστος h = c(i, d). Αν οποιοσδήποτε από τους εισερχόμενους γείτονες (i, d - 1) και (i - 1, d + 1) έχει κόστος h - 1, τότε λέμε ότι η (i, d) είναι dominated από αυτήν την γειτονική κορυφή.

Ας υποθέσουμε ότι μια κορυφή (i, d) είναι dominated από την γειτονική κορυφή (i, d - 1). Αν για την διαγώνιο d - 1 και για την γραμμή i + 1 υπάρχει ισότητα χαρακτήρων, δηλαδή ισχύει $A_{i+1} = B_{i+1+(d-1)}$, τότε υπάρχει ελάχιστο μονοπάτι το οποίο δεν περνάει από το (i, d) και συνεπώς μπορούμε να αποφύγουμε αυτήν την κορυφή, ακολουθώντας το μονοπάτι $(i, d-1) \rightarrow (i+1, d-1) \rightarrow (i+1, d)$. Παρατηρούμε ότι η κορυφή (i + 1, d) πρέπει επίσης να είναι dominated. Αν αντιθέτως στην διαγώνιο d - 1 ισχύει $A_{i+1} \neq B_{i+1+(d-1)}$, τότε μπορεί κάθε ελάχιστο μονοπάτι προς το (i + 1, d) να περνάει από το (i, d).

Potent κορυφές Σε επακόλουθο του πάνω ορισμού, τώρα θα ορίσουμε τις potent κορυφές. Λέμε ότι μια διαγώνιος d είναι potent σε μία γραμμή i αν ικανοποιούνται οι παρακάτω συνθήκες:

- αν (i, d) είναι dominated από το (i, d - 1), τότε απαιτείται η διαγώνιος d – 1 να είναι potent στην γραμμή i και να έχει ανομοιότητα χαρακτήρων στην γραμμή i + 1, και

- αν (i, d) είναι dominated από το (i - 1, d + 1), τότε απαιτείται η διαγώνιος d + 1 να είναι potent στην γραμμή i - 1 και να έχει ανομοιότητα χαρακτήρων στην γραμμή i.

Αν το (i, d) δεν είναι dominated τότε ικανοποιούνται οι παραπάνω συνθήκες και είναι potent. Η κορυφή source (0, 0) εξ' ορισμού δεν είναι dominated, και συνεπώς είναι potent.

Στην ανάπτυξη του αλγόριθμου, μας είναι σημαντικό να γνωρίζουμε αν μία κορυφή είναι potent, καθώς ισχύουν τα παρακάτω τρία πορίσματα:

i. Κάθε κορυφή στο $G_{A,B}$ έχει ελάχιστο μονοπάτι από το (0,0), για το οι non-potent κορυφές, εμφανίζονται μόνο μετά τις potent κορυφές.
ii. Αν η (i, d) δεν είναι potent, τότε c(i + 1, d) = c(i, d)
iii. Έστω (i, d) potent. Αν η (i + 1, d) έχει ανομοιότητα χαρακτήρων, τότε c(i + 1, d) = c(i, d) + 1. Στην περίπτωση που έχει ομοιότητα χαρακτήρων, τότε c(i + 1, d) = c(i, d) και (i + 1, d) είναι potent.

Active κορυφές Σε αυτό το σημείο, θα περιγράψουμε έναν αλγόριθμο ο οποίος θα υπολογίζει επαναληπτικά για κάθε γραμμή i = 0, 1, … , n μία λίστα D_i από διαγώνιους που θα ονομάζουμε active. Πρόκειται για ένα υπερσύνολο potent διαγώνιων για κάθε γραμμή i, η οποία δημιουργείται με την χρήση μίας λίστας $D_{next,}$, που θα αποθηκεύουμε τις διαγώνιους της επόμενης γραμμής, καθώς και μίας ακόμη λίστας c_A για αποθήκευση των κοστών των διαγώνιων για την προκειμένη γραμμή που επεξεργαζόμαστε. Αμφότερες και οι δύο βοηθητικές λίστες ανανεώνονται με την κάθε διαγώνιο που επεξεργαζόμαστε, ενώ η D_{next} αδειάζει με την αλλαγή της γραμμής, αφού πρώτα μεταφέρει τα στοιχεία της στην νέα γραμμή του D, D_{i+1}.

Ξεκινάμε αρχικοποιώντας την $D_0 = \{0\}$, κάθε άλλη λίστα $D_i = \emptyset$, $c_A[d] = |d|$ για κάθε $d \in [-n..n]$ και $D_{next} = \emptyset$. Ο αλγόριθμος μετά αρχίζει να επεξεργάζεται μια προς μία τις διαγώνιους από -n έως n για κάθε γραμμή i = 0, 1, … , n και να ελέγχει αν είναι potent. Για να το ελέγξουμε αυτό, έχουμε ότι χρειαζόμαστε, καθώς απαιτείται μόνο να γνωρίζουμε τα κόστη των κάθε διαγώνιων, τα οποία μπορούμε να βρούμε στην c_A, και γνώση του αν ορισμένες εισερχόμενες κορυφές είναι potent, που μπορούμε να την βρούμε ανατρέχοντας στο αντίστοιχο δείκτη της προηγούμενης του D_i. Το τελευταίο που ενδέχεται να χρειαστεί, είναι το να ελέγξουμε αν υπάρχει ομοιότητα ή ανομοιότητα χαρακτήρων σε κάποια διαγώνιο, το οποίο μπορούμε να το ελέγξουμε εύκολα, συγκρίνοντας τα αντίστοιχα σύμβολα των συμβολοσειρών A και B. Αφού ολοκληρωθεί αυτή η διαδικασία για κάθε γραμμή, στο D_i έχουν εισαχθεί όλες οι κορυφές που είναι potent για το $G_{A,B}$.

3.2.2 Επεξήγηση Αλγόριθμου

Αρχικά στον αλγόριθμο δίνονται σαν είσοδος δύο συμβολοσειρές A, B και μία παράμετρος t. Ξεκινάει επιλέγοντας ένα σετ $S \subseteq [n]$, όπου κάθε γραμμή επιλέγεται ανεξάρτητα, με πιθανότητα (log n)/t. Σε κάθε γύρο ο αλγόριθμος επεξεργάζεται από μία γραμμή $i \in [n]$ και επιλεκτικά κάποιες από τις κορυφές της, σε αύξοντα σειρά της διαγωνίου d. Θα τον λέμε από εδώ και στο εξής γύρο i.

Προκειμένου να βελτιώσουμε την χρονική πολυπλοκότητα σε $O\left(\frac{n}{t} + t^3\right)$, ο αλγόριθμος θα χρειαστεί να εναλλάσσεται μεταξύ ομοιόμορφης δειγματοληψίας και συνεχούς δειγματοληψίας. Συνεπώς έχει δύο τρόπους λειτουργίας, τον συνεχή και τον δειγματοληπτικό, (contiguous mode και sampling mode) αντίστοιχα.

Περιληπτικά στο sampling mode αξιοποιούμε την πληροφορία που λάβαμε από το D_i, για να αποφύγουμε άσκοπες εκτελέσεις. Ο αλγόριθμος επεξεργάζεται μόνο τις $i \in S$ σειρές και από αυτές μόνο τις active κορυφές. Στην περίπτωση που $|D_i| > 1$, ο αλγόριθμος ελέγχει αν οι συμβολοσειρές ακολουθούν ένα συγκεκριμένο περιοδικό μοτίβο p (periodicity check). Αν στην διάρκεια της εκτέλεσης παρατηρηθεί ότι δεν ακολουθείται άλλο το μοτίβο αυτό, από τουλάχιστον μία από τις δύο συμβολοσειρές, τότε ο αλγόριθμος μεταβαίνει σε contiguous mode.

Αν $|D_i| = 1$, τότε αρκεί απλά να συγκρίνουμε τους δύο χαρακτήρες των συμβολοσειρών για την μοναδική active διαγώνιο d (shift check). Όπως παρατηρούμε το sampling mode είναι αρκετά γρήγορο και ελαφρύ καθώς επεξεργάζεται σειρές με ρυθμό 1/t και εκτελεί ελάχιστες πράξεις.

Αλγόριθμος 3.3 Periodicity Check

```
01: if A_{i+1} == B_{i+maxDi+1} == p_{(i-ipat+1) mod g} do
02:     continue to next round
03: else do
04:     find row j
05:     for y = j to j + maxD_i - minD_i do
06:         for d in D_y do
07:             if A_{y+1} ≠ B_{y+d+1} do
08:                 c_A[d] = c_A[d] + 1
09:                 add {d, d + 1, d - 1} to D_next
10:             end if (can not enter again for the same d)
11:         end for
12:     end for
13:     if d* exists do
14:         set S* ⊆ [i_pat ... i] at rate (logn)/t
15:         for j' in S* do
```

16: if $A_{j'} \neq B_{j'+d^*}$ do

17: $c_A[d^*] = c_A[d^*] + 1$

18: add {d* + 1, d* - 1} to D_{next}

19: go to 22

20: end if

21: end for

22: add {d*} to D_{next}

23: end if

24: switch to contiguous mode in the next round

25: end if

Note: j row conditions are

i) $j \in [i_{pat} + 2(maxD_i - minD_i)..i]$

ii) for all $j' \in [j - 2(maxD_i - minD_i)..j]$, we have $A_{j'} = B_{j'}+maxD_i = p_{(j'-ipat)modg}$

iii) either A_{j+1} or y_j+maxD_{i+1} is not equal to $p_{(j+1-ipat) \bmod g}$

Note: we assume that (n mod p) returns a value in the range [1..p] (rather than [0..p −1] as usual)
Note: d* is a possible exception in the condition 07

Αλγόριθμος 3.4 Shift Check

01: d = the single d in D_i

02: if $A_{i+1} == B_{i+1+d}$ do

03: add {d} to D_{next}

04: else do

05: $c_A[d] = c_A[d] + 1$

06: add {d - 1, d, d + 1} to D_{next}

07: switch to contiguous mode in the next round

08: end if

Σε αντίθεση στο contiguous mode επεξεργάζονται οι σειρές μια προς μια, και για κάθε γύρο i, ο αλγόριθμος συγκρίνει το x_i με το y_{i+d} για κάθε active διαγώνιο $d \in D_i$. Αυτήν την λειτουργία την εφαρμόζουμε σε κομμάτια των τουλάχιστον O(t) συνεχόμενων σειρών, μέχρι το σετ των active διαγωνίων για ένα κομμάτι να μην αλλάξει (δηλαδή καμία active διαγώνιος να μην έχει ανομοιότητα χαρακτήρων σε αυτές τις σειρές). Σε αυτήν την περίπτωση συμπεραίνουμε ότι υπάρχει κάποιο επαναλαμβανόμενο μοτίβο μεταξύ αυτών των μερών του A και του B και αλλάζουμε σε sampling mode

Αλγόριθμος 3.5 Contiguous mode

```
01: for d in D_i
02:     if c_A(d) <= t − |d|
03:         if (i, d) is potent do
04:             add {d} to D_next
05:             if A_{i+1} != B_{i+d+1} do
06:                 add {d + 1} to D_i
07:                 add {d - 1} to D_next
08:                 c_A[d] = c_A[d] + 1
09:             end if
10:         end if
11:     end if
12: end for
13: if no mismatch for d ∈ D_i', where i' ∈ [i - 2(max D_i − min D_i), i], do
14:     g = gcd(d - d') where d > d' ∈ D_i
15:     i_pat = i - 2(maxD_i - minD_i) + 1
16:     p = A_[ipat ... ipat + g - 1]
17:     switch to sampling mode in the next round
```

Κεφάλαιο 4.

Πειραματικά Αποτελέσματα

Με το πέρας της ανάλυσης των αλγορίθμων μπορούμε πλέον να εκτελέσουμε τις υλοποιήσεις τους και να εξάγουμε συμπεράσματα ως προς την ταχύτητα με την οποία φτάνουν σε αποτελέσματα, αλλά και ως προς την ορθότητα των αποτελεσμάτων.

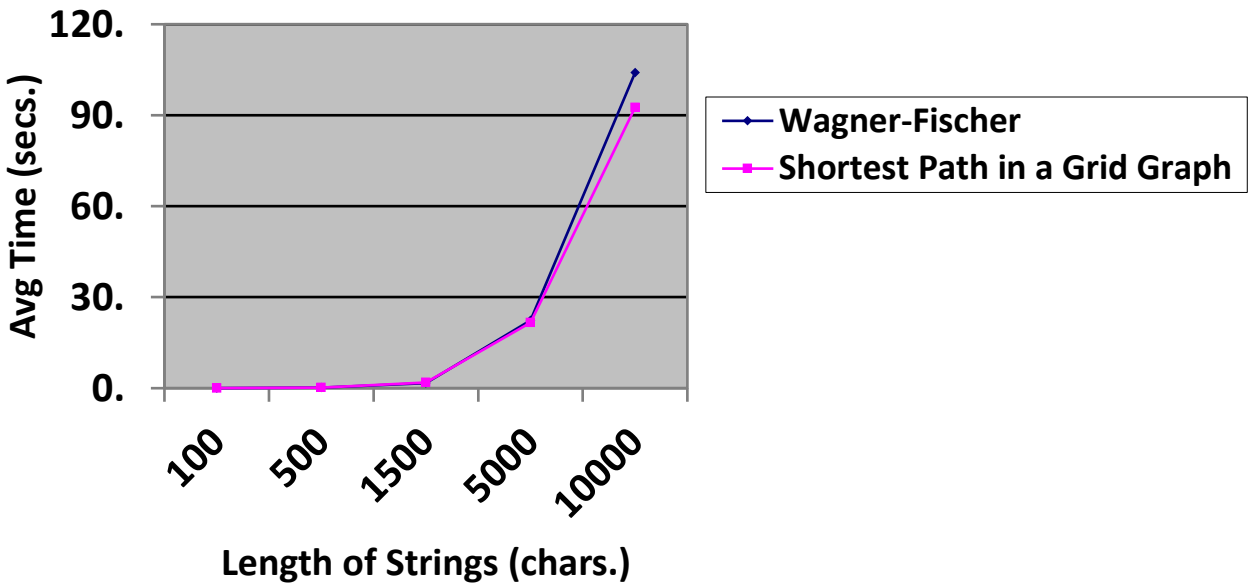
Σχετικά με την υλοποίηση, οι αλγόριθμοι γράφτηκαν σε γλώσσα python 3.9 και εκτελέστηκαν σε προσωπικό υπολογιστή Apple MacBook Air 2020 (Processor 1,1 GHz Dual-Core Inter Core i3/RAM 8 GB). Η καταμέτρηση χρόνου επιτεύχθηκε με την χρήση συνάρτησης time(). Οι συμβολοσειρές που εξετάσαμε, δημιουργήθηκαν τυχαία, μέσω μίας γεννήτριας τυχαίων συμβολοσειρών που αναπτύξαμε για τις ανάγκες της διπλωματικής. Είναι σημαντικό να αναφέρουμε ότι για να δοκιμάσουμε τους αλγόριθμους σε διάφορες περιπτώσεις, δοκιμάστηκαν είσοδοι συμβολοσειρών με διαφορετικό μήκος, καθώς και με διαφορετικές αποστάσεις hamming μεταξύ τους. Δοκιμάστηκαν μόνο δυαδικές συμβολοσειρές για τα πειράματα, όμως οι υλοποιήσεις των αλγορίθμων λειτουργούν κανονικά και για συμβολοσειρές οποιονδήποτε χαρακτήρων. Παρακάτω παραθέτουμε τα αποτελέσματα και των τεσσάρων αλγορίθμων που αναλύσαμε.

Πίνακας 4.1

Hamming Distance: irrelevant

Length of String (chars.)	*Wagner-Ficher Algorithm avg time (secs.)*	*Shortest Path in a Grid Graph avg time (secs.)*
20	0.0008	0.0009
100	0.0129	0.0163
500	0.1295	0.2253
1500	1.6947	1.8882
5000	22.3628	21.6148
10000	104.0328	92.5452

Γράφημα 4.1



Πίνακας 4.2

Hamming Distance: small (bellow or equal to t value)

Length of strings (chars.)	*t value/ Hamming Distance*	*Printed Result*	*Shortest Path in a Sampled Grid Graph avg time (secs.)*
1000	*5/5*	"Close" $^4/_4$	0.0101
	10/5	"Close" $^4/_4$	0.0096
	30/5	"Close" $^4/_4$	0.0090
10000	*20/20*	"Close" $^4/_4$	0.0754
	50/20	"Close" $^4/_4$	0.0724
	100/20	"Close" $^4/_4$	0.0724
50000	*50/50*	"Close" $^4/_4$	0.3988
	100/50	"Close" $^4/_4$	0.3888
	200/50	"Close" $^4/_4$	0.3832
200000	*100/100*	"Close" $^4/_4$	1.7898
	200/100	"Close" $^4/_4$	1.8142
	400/100	"Close" $^4/_4$	1.8282
1000000	*200/200*	"Close" $^4/_4$	10.3291
	500/200	"Close" $^4/_4$	10.9585
	1000/200	"Close" $^4/_4$	10.7485

Πίνακας 4.3

Hamming Distance: mediocre (6t)

Length of strings (chars.)	*t value / Hamming Distance*	*Printed Result*	*Shortest Path in a Sampled Grid Graph avg time (secs.)*
1000	*5/30*	"Far" $^4/_4$	0.0103
	10/60	"Far" $^4/_4$	0.0099
	30/180	"Far" $^4/_4$	0.0091
10000	*20/120*	"Far" $^4/_4$	0.0779
	50/300	"Close" $^4/_4$	0.0755
	100/600	"Close" $^4/_4$	0.0772
50000	*50/300*	"Close" $^4/_4$	0.3985
	100/600	"Close" $^4/_4$	0.3930
	200/1200	"Close" $^4/_4$	0.3929
200000	*100/600*	"Close" $^4/_4$	1.7902
	200/1200	"Close" $^4/_4$	1.8029
	400/2400	"Close" $^4/_4$	1.9047
1000000	*200/1200*	"Close" $^4/_4$	11.3205
	500/2400	"Close" $^4/_4$	12.4378
	1000/6000	"Close" $^4/_4$	11.5857

Πίνακας 4.4

Hamming Distance: big ($6t^2$)

Length of strings (chars.)	*t value / Hamming Distance*	*Printed Result*	*Shortest Path in a Sampled Grid Graph avg time (secs.)*
10000	*20/2400*	"Far" $^4/_4$	0.0747
50000	*50/15000*	"Far" $^4/_4$	0.4394
200000	*100/60000*	"Far" $^4/_4$	1.9383
1000000	*200/240000*	"Far" $^4/_4$	11.4400

Πίνακας 4.5

Hamming Distance: small (bellow or equal to t value)

Length of strings (chars.)	*t value/ Hamming Distance*	*Printed Result*	*Sublinear Algorithm for Gap Edit Distance avg time (secs.)*
1000	*5/5*	"Close" $^4/_4$	0.0061
	10/5	"Close" $^4/_4$	0.0110
	30/5	"Close" $^4/_4$	0.0163
10000	*20/20*	"Close" $^4/_4$	0.0439
	50/20	"Close" $^4/_4$	0.1670
	100/20	"Close" $^4/_4$	0.6671
50000	*50/50*	"Close" $^4/_4$	0.4117
	100/50	"Close" $^4/_4$	1.3564
	200/50	"Close" $^4/_4$	6.8947
200000	*100/100*	"Close" $^4/_4$	3.1530
	200/100	"Close" $^4/_4$	10.4857
	400/100	"Close" $^4/_4$	61.4870
1000000	*200/200*	"Close" $^4/_4$	35.0020
	500/200	"Close" $^4/_4$	211.7807

Πίνακας 4.6

Hamming Distance: mediocre (6t)

Length of strings (chars.)	*t value / Hamming Distance*	*Printed Result*	*Shortest Path in a Sampled Grid Graph avg time (secs.)*
1000	*5/30*	"Far" $^4/_4$	0.0021
	10/60	"Far" $^4/_4$	0.0051
	30/180	"Far" $^4/_4$	0.0242
10000	*20/120*	"Close" $^1/_4$, ""Far" $^3/_4$	0.0336
	50/300	"Close" $^4/_4$	0.1714
	100/600	"Close" $^4/_4$	0.4928
50000	*50/300*	"Close" $^4/_4$	0.3885
	100/600	"Close" $^4/_4$	1.3287
	200/1200	"Close" $^4/_4$	7.2117
200000	*100/600*	"Close" $^4/_4$	3.1326
	200/1200	"Close" $^4/_4$	8.3862
	400/2400	"Close" $^4/_4$	69.3668
1000000	*200/1200*	"Close" $^4/_4$	35.5396
	500/2400	"Close" $^4/_4$	219.8553

Πίνακας 4.7

Hamming Distance: big $(6 \times t^2)$

Length of strings (chars.)	*t value / Hamming Distance*	*Printed Result*	*Sublinear Algorithm for Gap Edit Distance avg time (secs.)*
10000	*20/2400*	"Far" $^4/_4$	0.02318
50000	*50/15000*	"Far" $^4/_4$	0.02824
200000	*100/60000*	"Far" $^4/_4$	0.1467
1000000	*200/240000*	"Far" $^4/_4$	1.1785

4.1 Παρατηρήσεις και Συμπεράσματα

Έχοντας τα αποτελέσματα των πειραμάτων μπορούμε να κάνουμε πολλές ενδιαφέρον παρατηρήσεις συγκρίνοντας στατιστικά τους αλγόριθμους μεταξύ τους.

Θα ξεκινήσουμε από τους αλγόριθμους Wagner-Fischer και Shortest Path in a Grid Graph στον Πίνακα 4.1 και στο Γράφημα 4.1. Οι αλγόριθμοι έχουν σχεδόν τον ίδιο μέσο χρόνο υπολογισμού του edit distance. Ο αλγόριθμος Wagner-Fischer είναι λίγο γρηγορότερος για μικρότερες συμβολοσειρές, και ο αλγόριθμο Shortest Path in a Grid Graph είναι γρηγορότερος για μεγαλύτερες αντίστοιχα. Γενικά και οι δύο αλγόριθμοι μας δίνουν ένα γρήγορο αποτέλεσμα για μικρές συμβολοσειρές, όμως η απότομη αύξηση του απαιτούμενου χρόνου για συμβολοσειρές μεγάλου μήκους, τους κάνει απαγορευτικούς για ανάλυση μεγάλων δεδομένων.

Σε αντίθεση οι αλγόριθμοι Shortest Path in a Sampled Grid Graph και Sublinear Algorithm for Gap Edit Distance είναι μια καλή λύση στο πρόβλημα του gap του edit distance για μεγάλες συμβολοσειρές. Ο πρώτος χρειάζεται μόλις περίπου έντεκα δευτερόλεπτα για να δώσει μια προσεγγιστική απάντηση στο πρόβλημα για συμβολοσειρές των ενός εκατομμυρίου χαρακτήρων, ενώ ο υπογραμμικός χρειάζεται περίπου τριανταπέντε δευτερόλεπτα για t = 200 και διακόσια είκοσι δευτερόλεπτα για t = 500 (δες Πίνακα 4.2, 4.3, 4.5 και 4.6) σε συμβολοσειρές αντίστοιχου μεγέθους. Είναι σημαντικό όμως να σημειωθεί σε αυτό το σημείο, ότι τα απαιτούμενα δευτερόλεπτα για τον υπογραμμικό αλγόριθμο, θα ήταν πιθανότατα πολύ λιγότερα αν γινόταν βελτιστοποίηση του αλγορίθμου μετά την συγγραφή του, ενώ ενδέχεται και ορισμένες δομές τις python να επηρεάζουν αρνητικά τον χρόνο υπολογισμού.

Κάτι άλλο που μπορούμε να παρατηρήσουμε σχετικά με τον αλγόριθμο Shortest Path in a Sampled Grid Graph και τον Sublinear Algorithm for Gap Edit Distance, είναι ότι ο πρώτος δεν φαίνεται να επηρεάζεται από την τιμή του t ενώ ο δεύτερος όσο μεγαλύτερο είναι το t, τόσος περισσότερος είναι ο απαιτούμενος χρόνος.

Η διαφορά τους είναι επίσης εμφανής όταν εισάγουμε συμβολοσειρές με μεγάλο hamming distance(Πίνακες 4.4 και 4.7). Στον υπογραμμικό αλγόριθμο μπορούμε να επιβεβαιώσουμε ότι δύο συμβολοσειρές έχουν μεγάλο edit distance σε πολύ λίγο χρόνο, μακράν πιο γρήγορα από ότι μπορεί να επιβεβαιώσει ο αλγόριθμος Shortest Path in a Sampled Grid Graph. Αυτό οφείλεται στο ότι ο υπογραμμικός αλγόριθμος σταματάει την εκτέλεση του κώδικα μόλις ελεγχθεί ότι το edit distance ξεπερνάει το πάνω κατώφλι που θέσαμε, και εκτυπώνει το αποτέλεσμα, ενώ ο άλλος εκτελεί μέχρι τέλους και μετά μας δίνει το αποτέλεσμα.

Ως προς την ορθότητα των αποτελεσμάτων και οι δύο αλγόριθμοι εκτύπωσαν σωστά αποτελέσματα για αρκετά μικρό και αρκετά μεγάλο Hamming Distance μεταξύ των συμβολοσειρών τους. Όπως μπορούμε να δούμε στους Πίνακες 4.2 και 4.5 στο 100% των πειραμάτων οι αλγόριθμοι εκτύπωσαν ότι το edit distance είναι μικρό (“Close”), ενώ αντίστοιχα στους Πίνακες 4.4 και 4.7 για μεγάλο Hamming Distance στο 100% των πειραμάτων οι αλγόριθμοι εκτύπωσαν ότι το edit distance είναι μεγάλο (“Far”). Στην περίπτωση των Πινάκων 4.3 και 4.4, όπου το Hamming Distance δεν είναι πολύ μεγάλο αλλά ξεπερνάει το t, για μικρές συμβολοσειρές των δέκα χιλιάδων χαρακτήρων δόθηκε η σωστή απάντηση στα πειράματα (“Far”), όμως για μεγαλύτερες συμβολοσειρές με μεγαλύτερο t μας δόθηκε λάθος απάντηση (“Close”). Συνεπώς μπορούμε να συμπεράνουμε ότι για μικρότερες συμβολοσειρές και οι δύο αλγόριθμοι μας δίνουν ορθότερα αποτελέσματα, για ελαφρώς μεγαλύτερο edit distance.

Βιβλιογραφία

[1] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[2] Robert A. Wagner, Michael J. Fischer. *The String-to-String Correction Problem*, 1974

[3] Elazar Goldenberg, Robert Krauthgmaer, Barna Saha. *Sublinear Algorithms for Gap Edit Distance*, 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, 1989