Ατομική Διπλωματική Εργασία

ΣΧΕΔΙΑΣΗ ΚΑΙ ΑΝΑΠΤΥΞΗ ΑΝΑΣΤΡΕΨΙΜΩΝ ΑΛΓΟΡΙΘΜΩΝ ΚΩΔΙΚΟΠΟΙΗΣΗΣ ΚΑΙ ΚΡΥΠΤΟΓΡΑΦΗΣΗΣ

Χρίστος Παναγή

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2017

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σχεδίαση και Ανάπτυξη Αναστρέψιμων Αλγορίθμων Κωδικοποίησης και Κρυπτογράφησης

Χρίστος Παναγή

Επιβλέπουσα Καθηγήτρια
Άννα Φιλίππου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2017

Ευχαριστίες

Θα ήθελα να εκφράσω τις ευχαριστίες μου και την ευγνωμοσύνη μου στην Δρ. Άννα Φιλίππου, Αναπληρώτρια Καθηγήτρια του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου που ως επιβλέπουσα καθηγήτρια προσέφερε τις δικές της γνώσεις όσο αφορά το θέμα του Αναστρέψιμου Υπολογισμού και την απαραίτητη καθοδήγηση για να μπορέσει να έρθει εις πέρας η εν λόγω Ατομική Διπλωματική Εργασία.

Περίληψη

Η διπλωματική εργασία ασχολείται με το θέμα του *αναστρέψιμου υπολογισμού (reversible computing)*. Ο αναστρέψιμος υπολογισμός είναι ένα είδος υπολογισμού το οποίο λαμβάνει υπόψη όχι μόνο την εμπρόσθια κατεύθυνση ενός αλγορίθμου αλλά δίνει και την δυνατότητα μετάβασης προς την ανάστροφη κατεύθυνση. Η διπλωματική αυτή καταπιάνεται με το γεγονός ότι ως έννοια ο αναστρέψιμος υπολογισμός προτάθηκε αρκετά χρόνια πριν αλλά ακόμη απέχουμε πολύ από την πλήρη υλοποίηση ενός υπολογιστή ολοκληρωτικά αναστρέψιμου. Για την κατασκευή ενός πλήρως αναστρέψιμου υπολογιστή χρειάζεται μελέτη και προσαρμογή ολόκληρου του φάσματος του υπολογισμού. Χρειάζεται τροποποίηση όλων των λειτουργιών υλικού αλλά και λογισμικού ώστε να μπορούν να εκτελούν ανάστροφες πράξεις. Επιπρόσθετα, παρουσιάζονται το κίνητρο που οδήγησε στη μελέτη του αναστρέψιμου υπολογισμού και διάφορα προβλήματα και περιορισμοί που εμποδίζουν την ανάπτυξη του. Στη συνέχεια η εργασία καταπιάνεται με την δημιουργία αναστρέψιμων αλγορίθμων και προτείνονται αναστρέψιμες εκδοχές δημοφιλών αλγορίθμων κωδικοποίησης όπως Leveling, Burrows-Wheeler και Huffman. Επιπρόσθετα, δίνονται αναστρέψιμες προσομοιώσεις σε αλγόριθμους κρυπτογραφίας όπως Data Encryption Standard και Blowfish. Για όλους του αλγόριθμους (πλην Huffman) παρέχονται υλοποιήσεις στην αναστρέψιμη γλώσσα προγραμματισμού Janus. Τα συμπεράσματα στα οποία καταλήγει η διπλωματική εργασία είναι ότι η δημιουργία αναστρέψιμων αλγορίθμων είναι δυσκολότερη από την δημιουργία μη αναστρέψιμων, αφού πλέον στην σχεδίαση του αλγόριθμου πρέπει να λαμβάνεις υπόψη και τις δύο κατευθύνσεις. Το σημαντικό επίτευγμα όμως είναι ότι μέσα από αυτή τη διπλωματική εργασία φαίνεται ότι είναι δυνατή η μετατροπή αλγορίθμων σε αναστρέψιμους ακόμα και τώρα που ο αναστρέψιμος υπολογισμός βρίσκεται σε πρώιμο στάδιο ανάπτυξης. Τέλος, παρατίθενται εισηγήσεις για βελτίωση της γλώσσα Janus για ευκολότερη χρήσης της και για να ωθήσει στην ανάπτυξη του αναστρέψιμου προγραμματισμού.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Αναστρέψιμος Υπολογισμός

Ακούγοντας την λέξη υπολογισμός, είναι απολύτως φυσιολογικό η πρώτη μας σκέψη να είναι ο εμπρόσθιος ή συμβατικός υπολογισμός. Ο εμπρόσθιος υπολογισμός ήταν και εξακολουθεί να βρίσκεται στο επίκεντρο της έρευνας για όλα τα υπολογιστικά συστήματα. Πληροφορική, Μαθηματικά, Φυσική είναι μόνο λίγες από τις περιοχές όπου υπάρχει «έμφυτη» η έννοια της αναστρεψιμότητας. Στην Πληροφορική η αναστρεψιμότητα βρίσκει εφαρμογές σε αλγόριθμους όπου μας ενδιαφέρει η προηγούμενη κατάσταση και η δυνατότητα μετάβασης σε αυτή με τον ίδιο τρόπο. Στα Μαθηματικά η αναστρεψιμότητα εμφανίζεται στις συναρτήσεις $(F(x))$ και συγκεκριμένα στις αντίστροφες συναρτήσεις $(F^{-1}(x))$, όπου συμπεριφέρονται αντίστροφα από τις αυθεντικές. Στη Φυσική υπάρχει η ανάγκη για αναστρεψιμότητα των φυσικών φαινομένων και συγκεκριμένα στη θερμοδυναμική υπάρχει ο όρος των «αναστρέψιμων διεργασιών». Σε μια αναστρέψιμη διεργασία, το σώμα είναι σε θερμική ισορροπία και με την εκτέλεση απειροελάχιστων αλλαγών είναι δυνατή η αναστροφή της κατάστασης του χωρίς αύξηση της εντροπίας. Παρ' όλα αυτά, η ανάπτυξη των επιστημών επικεντρώθηκε στην συμβατική έννοια του υπολογισμού. Δηλαδή ξεκινώντας από μια αρχική κατάσταση x και εκτελώντας ένα σύνολο πράξεων $s_i \in$ S πάνω στην κατάσταση x θα καταλήξω στην τελική κατάσταση y. Η ευθεία όμως γραμμή έχει δύο κατευθύνσεις. Η αντίστροφη

κατεύθυνση, αυτή του *αναστρέψιμου υπολογισμού* [13] δηλώνει ότι ξεκινώντας από την τελική κατάσταση y και εκτελώντας αντίστροφα τις πράξεις $s_i \in$ S θα καταλήξεις στην αρχική κατάσταση x. Επομένως αποτελεί ένα μοντέλο μη συμβατικού υπολογισμού, αναπτυσσόμενο και πολλά υποσχόμενο.

Ο αναστρέψιμος υπολογισμός δεν έρχεται να καταλύσει τον εμπρόσθιο υπολογισμό. Έρχεται εν μέρει να τον συμπληρώσει. Όμοια, στα μαθηματικά, η προσθήκη των μιγαδικών αριθμών όχι μόνο δεν εμπόδισε την ανάπτυξη τους αλλά τους έδωσε μια πιο γενικευμένη υπόσταση και ώθηση για ανάπτυξη [11]. Η δημιουργία και η χρήση ανάποδων πράξεων είναι κάτι που δεν συναντάται εύκολα στην φύση. Στον υπολογισμό όμως, η δυνατότητα της γνώσης της προηγούμενης και της επόμενης κατάστασης με σιγουριά είναι τόσο δελεαστικό όπου υποκίνησε την ανάπτυξη ενός νέου είδους υπολογισμού. Ο αναστρέψιμος υπολογισμός λοιπόν, είναι το είδος υπολογισμού το οποίο επιτρέπει πράξεις και προς τις δύο κατευθύνσεις, με την προϋπόθεση ότι οι πράξεις προς την ανάστροφη κατεύθυνση θα εκτελούνται, όπως εννοεί και το όνομα του όρου, με ανάστροφη σειρά [1]. Έτσι, στην πράξη, η μετάβαση από μια προηγούμενη κατάσταση και στη συνέχεια η μετάβαση ξανά προς τα μπροστά θεωρείται θεμελιώδης ιδιότητα του μοντέλου αυτού, κάτι που χρησιμοποιείται κατά κόρον σε συστήματα αποσφαλμάτωσης. όπως για παράδειγμα το σύστημα Igor το οποίο επιτυγχάνει αποσφαλμάτωση ενός συστήματος με ανάστροφη εκτέλεση του.

Το πραγματικό κίνητρο για την μελέτη του αναστρέψιμου υπολογισμού εμφανίστηκε το 1961 από τον R.Landauer [13]. Ανακαλύφθηκε ότι ένα μερίδιο της ηλεκτρικής ενέργειας που χρησιμοποιεί ένας υπολογιστής, διαρρέει σε μορφή θερμότητας. Μετά από μελέτες διαπιστώθηκε ότι η διαρροή αυτή οφείλεται στην διαγραφή δεδομένων ή αλλιώς στην πράξη της ανάθεσης. Με την πράξη της ανάθεσης τιμής, οποιαδήποτε προηγούμενη τιμή υπάρχει στην μεταβλητή στο ψηλό επίπεδο ή στον καταχωρητή στο χαμηλό, αντικαθίσταται και επομένως παύει πλέον να υπάρχει. Αυτό έχει ως αποτέλεσμα τη διαρροή θερμότητας και την αύξηση της εντροπίας στο σύστημα.

Όμως παρά την ανακάλυψη αυτή, η ανάγκη για την ανάπτυξη του αναστρέψιμου υπολογισμού δεν εμφανίστηκε μέχρι πρόσφατα, όταν η βιομηχανία των επεξεργαστών ξεκίνησε να οδεύει προς τα όρια της [13]. Ως συνέπεια, έρευνες σχετικές με όλες τις πτυχές του φάσματος του

αναστρέψιμου υπολογισμού διεξάγονται με ένα κοινό στόχο, την δημιουργία ενός πλήρως αναστρέψιμου υπολογιστή.

Το κίνητρο για την εκπόνηση της διπλωματικής εργασίας έγκειται στο γεγονός ότι η ανάπτυξη αλγορίθμων σε αναστρέψιμο επίπεδο δεν έχει μελετηθεί στο έπακρο. Επομένως, βασικό μέλημα είναι η υλοποίηση αλγορίθμων οι οποίοι έχουν εκ φύσεως την έννοια της αναστρεψιμότητας (αλγόριθμοι κωδικοποίησης και κρυπτογράφησης) με στόχο να φανεί το όφελος από την ανάπτυξη τους. Ο στόχος αυτός επιτυγχάνεται στο γεγονός ότι μπορούμε να υλοποιήσουμε τους αλγορίθμους σε αναστρέψιμο επίπεδο χωρίς επιπλέον κόστος (σε μη επιθυμητά επίπεδα) δίνοντας το πράσινο φως για περαιτέρω ανάπτυξη αυτής της πτυχής του αναστρέψιμου υπολογισμού.

1.2 Σκοπός της Διπλωματικής

Ο πρώτος στόχος αυτής της διπλωματικής εργασίας είναι η μελέτη και η κατανόηση της έννοιας της αναστρεψιμότητας. Στη συνέχεια, μετά την αφομοίωση και ενδελεχή κατανόηση του αναστρέψιμου υπολογισμού, ο κύριος στόχος της διπλωματικής εργασίας είναι η ανάπτυξη οικογενειών αλγορίθμων οι οποίες έχουν ενσωματωμένη την ανάγκη για αναστρεψιμότητα όπως οι αλγόριθμοι κωδικοποίησης και κρυπτογράφησης. Με το πέρας της ανάπτυξης των πιο πάνω οικογενειών αλγορίθμων, η διπλωματική αυτή καλείται να απαντήσει στο ερώτημα αν όλοι οι αλγόριθμοι μπορούν να υλοποιηθούν σε αναστρέψιμο επίπεδο και καλείται να αναλύσει τα τυχόν προβλήματα που υπάρχουν σχετικά με την ανάπτυξη αλγορίθμων και να επιχειρηματολογήσει για το αν το επιπλέον κόστος που δημιουργείται κατά την ανάπτυξη αναστρέψιμων αλγορίθμων αντισταθμίζει την ανάγκη για μετέπειτα ανάπτυξη του αναστρέψιμου υπολογισμού.

1.3 Δομή της Διπλωματικής

Η διπλωματική αυτή επικεντρώνεται στην ανάπτυξη αναστρέψιμων αλγορίθμων και στον αναστρέψιμο προγραμματισμό.

Στο επόμενο κεφάλαιο γίνεται ιστορική αναδρομή όσο αφορά την αναστρεψιμότητα ως έννοια και την έρευνα που έχει επικεντρωθεί στον τομέα αυτό. Στη συνέχεια παρουσιάζονται οι περιορισμοί που εμποδίζουν την ανάπτυξη του αναστρέψιμου υπολογισμού και οι ενέργειες που πρέπει να παρθούν για αντιμετωπιστούν. Τέλος γίνεται αναφορά στη γλώσσα αναστρέψιμου υπολογισμού Janus και στην -μέχρι τώρα- δουλειά που έχει γίνει όσο αφορά τη μετατροπή μη αναστρέψιμων αλγορίθμων σε αναστρέψιμη εκδοχή.

Στο Κεφάλαιο 3 παρουσιάζονται αναστρέψιμες εκδοχές γνωστών αλγορίθμων κωδικοποίησης. Ο πρώτος αλγόριθμος που υλοποιήθηκε είναι ο αλγόριθμος Leveling ο οποίος κωδικοποιεί ένα κείμενα βασισμένος στην έννοια των επιπέδων. Επιπρόσθετα, γίνεται παρουσίαση της αναστρέψιμης εκδοχής του αλγόριθμου μετατροπής Burrows-Wheeler και, τέλος, παρουσιάζεται μια προσέγγιση για αναστρέψιμη υλοποίηση του αλγόριθμου Huffman. Για όλους τους αλγόριθμους παρουσιάζεται ανάλυση της χωρικής και χρονικής της πολυπλοκότητας.

Στο Κεφάλαιο 4 παρουσιάζονται αναστρέψιμες εκδοχές γνωστών αλγορίθμων κρυπτογράφησης. Ο πρώτος αλγόριθμος που υλοποιήθηκε είναι το πρότυπο κρυπτογράφησης Data Encryption Standard (DES). Στη συνέχεια, δίδεται μια υλοποίηση του αλγορίθμου Blowfish, προσαρμοσμένη και απλοποιημένη στα δεδομένα της γλώσσας Janus.

Στο Κεφάλαιο 5 δίνονται συμπεράσματα όσον αφορά τη μελέτη και την ανάπτυξη αναστρέψιμων αλγορίθμων. Επιπρόσθετα, αναφέρονται οι δυσκολίες που συναντήθηκαν και ο τρόπος με τον οποίο αντιμετωπίστηκαν για την επιτυχή ανάπτυξη των αλγορίθμων. Τέλος, δίνονται εισηγήσεις για ανάπτυξη της γλώσσας Janus ούτως ώστε να μπορεί να ανταγωνιστεί τις σύγχρονες γλώσσες προγραμματισμού.

Οι πλήρεις υλοποιήσεις σε κώδικα Janus για τους αλγόριθμους που αναφέρονται στα Κεφάλαια 3 και 4 υπάρχουν στα Παραρτήματα Α και Β αντίστοιχα.

Κεφάλαιο 2

Ιστορικό

2.1 Μελέτη της αναστρεψιμότητας

Η αναστρεψιμότητα ως έννοια είναι θεμελιώδης στις επιστήμες. Υπάρχει έμφυτη στα Μαθηματικά, στη Φυσική, στη Βιολογία και στην Πληροφορική. Στον τομέα της Πληροφορικής, η έννοια της αναστρεψιμότητας εκδηλώνεται υπό μορφή υπολογισμών. Ο αναστρέψιμος υπολογισμός είναι η μορφή του υπολογισμού που έχει την δυνατότητα προσπέλασης καταστάσεων προς τα πίσω, δηλαδή στην αντίθετη πλευρά από το συμβατικό υπολογιστικό μοντέλο που είναι η βάση των υπολογιστών όπως τους ξέρουμε.

Η αρχική εντύπωση όσο αφορά την αναστρεψιμότητα ήταν ότι μπορεί να εφαρμοστεί σε οποιοδήποτε πρόβλημα. Ωστόσο, η δυνατότητα μετάβασης σε προηγούμενη κατάσταση έγκειται στη φύση του προβλήματος και δεν είναι κάτι που μπορεί να γίνει ρητά κατά τη σχεδίαση ενός αναστρέψιμου αλγορίθμου [1]. Έτσι διαπιστώθηκε ότι η αναστρεψιμότητα περιορίζεται μόνο στο σύνολο των προβλημάτων που η συνάρτηση μετάβασης τους ορίζεται ως ένα-προς-ένα. Ένας συμβατικός υπολογιστής μπορεί να εκτελέσει πράξεις οι οποίες να είναι εμπρόσθια ντετερμινιστικές [4]. Για να μπορεί ένας υπολογιστής να ξεφύγει από τα όρια του συμβατικού -μη αναστρέψιμου- και να εισέλθει στο φάσμα του αναστρέψιμου υπολογισμού χρειάζεται να προσαρμοστεί έτσι ώστε να παρέχει ντετερμινιστικές πράξεις σε εμπρόσθιο αλλά

και σε ανάστροφο επίπεδο. Έτσι, η συνάρτηση μετάβασης του πρέπει να μετατραπεί από πολλά-προς-ένα που υπάρχει στους συμβατικούς υπολογιστικές σε ένα-προς-ένα [4]. Η μελέτη τέτοιων υπολογιστικών μοντέλων μέχρι στιγμής περιορίστηκε σε αναστρέψιμες μηχανές Turing (RTM), κυτταρικά αυτόματα (cellular) και συνδυαστική λογική [1][11]. Είναι εύκολο να φτάσει κάποιος στο (λανθασμένο) συμπέρασμα ότι ίσως να μην είναι εφικτή η πραγματική υλοποίηση ενός αναστρέψιμου υπολογιστή που να αντικαταστήσει τον συμβατικό υπολογιστή που όλοι ξέρουμε και χρησιμοποιούμε καθημερινά [11].

Παρόλα αυτά, επιστήμονες από διάφορους τομείς με ενδιαφέρον και γνώσεις σε θέματα τεχνολογίας πιστεύουν ότι ο αναστρέψιμος υπολογισμός είναι το μέλλον όσο αφορά τα όρια του υπολογισμού. Σύμφωνα με τον νόμο του Moore, η υπολογιστική ισχύ των επεξεργαστών για ένα συμβατικό υπολογιστή διπλασιάζεται κάθε 12-18 μήνες. Αυτό δεν αποτελεί νόμο της φυσικής, αλλά τεχνολογικό μοτίβο το οποίο για πέρα από 40 χρόνια εξακολουθεί να ισχύει. Όμως, πλησιάζουμε στο σημείο όπου φτάνουμε τα υλικά όρια του -συμβατικού- υπολογισμού [6]. Τα όρια αυτά προκύπτουν από τους θεμελιώδεις νόμους της μηχανικής και της θερμοδυναμικής και αφορούν την κατανάλωση ηλεκτρικής ενέργειας. Έχει αποδειχτεί από τον Landauer ότι στους συμβατικούς υπολογιστές παρατηρείται διαρροή ενέργειας υπό μορφή θερμότητας όταν εκτελείται η μη αναστρέψιμη πράξη της διαγραφής των δυφίων. Χάρη στη συνεχή σμίκρυνση του υλικού αναμένεται ότι η βιομηχανία μικροεπεξεργαστών και κατ' επέκταση επεξεργαστών θα φτάσει στο κατώτατο όριο των von Neumann-Landauer όσο αφορά την απώλεια θερμότητας λόγω διαγραφής πληροφορίας [6].

Ο διάσημος ερευνητής της IBM Charles Bennett ήταν ο πρώτος που αναγνώρισε ότι για να μπορέσουμε να δημιουργήσουμε αναστρέψιμες μηχανές πρέπει να εγκαταλείψουμε τη μη αναστρέψιμη λογική των ψηφιακών κυκλωμάτων και να υιοθετήσουμε αναστρέψιμη λογική ένα-προς-ένα λογικών πράξεων [4]. Παρά το γεγονός ότι η μελέτη της αναστρεψιμότητας ξεκίνησε πολλά χρόνια πριν ακόμη δεν έχει φτάσει σε πρακτικό επίπεδο. Αυτό οφείλεται σε δύο σημαντικούς λόγους, (1) η δημιουργία τεχνολογίας που να επιτρέπει τέτοιου είδους πράξεις είναι ένα πρόβλημα πιο δύσκολο από όσο οι μηχανικοί και οι επιστήμονες της θεωρίας υπολογισμού τείνουν να εκτιμούν , (2) η τεχνολογία για αυτό το διάστημα ήταν αποδοτική όσο αφορά την ενέργεια και δεν υπήρχε η ανάγκη για αλλαγή. Όπως έχει ήδη αναφερθεί ο λόγος (2) είναι πλέον ανεπαρκής καθώς πλησιάζουμε τα όρια του υπολογισμού.

Κρίνεται λοιπόν αναγκαία η μελέτη και η ανάπτυξη αυτού του μοντέλου. Σύμφωνα με τον Michael P. Frank [6], για τη δημιουργία αναστρέψιμων μηχανών, πρέπει να αντιμετωπίσουμε ένα σύνολο περιορισμών. Οι περιορισμοί αυτοί εξηγούνται με συντομία πιο κάτω.

1. Φυσική μοντελοποίηση συσκευής: η νανοτεχνολογία πρέπει να αναπτυχθεί σε σημείο όπου με υψηλή ακρίβεια να μπορεί να εντοπίζει τι μπορεί να συμβεί στο «νέφος» από επόμενες, πιθανές καταστάσεις που αντιστοιχούν σε μια λογική κατάσταση όταν εκτελεστεί μια λογική, επικοινωνιακή μετάβαση ή μετάβαση αποθήκευσης σ' αυτή.
2. Μηχανισμοί ανάκτησης ενέργειας: Είναι σημαντικό σε μια ανατρέψιμη υλοποίηση να μπορούμε να ανιχνεύσουμε ανά πάσα στιγμή όχι μόνο τη ροή της πληροφορίας στο σύστημα αλλά και τη ροή της ενέργειας. Άρα για μην υπάρχει διαρροή ενέργειας με την απελευθέρωση θερμότητας είναι απαραίτητο με το πέρας μιας λειτουργίας, το κομμάτι ενέργειας που χρησιμοποιήθηκε να ανακατευθυνθεί αμέσως σε μια άλλη λειτουργία. Η έρευνα μέχρι στιγμής έχει επικεντρωθεί σε «ενεργειακά ρολόγια» (power clocks) τα οποία κρατούν την ενέργεια απασχολημένη χρησιμοποιώντας κυκλικές διαδικασίες.
3. Κωδικοποίηση λογικών καταστάσεων: Πρέπει να βρεθεί ο τρόπος κωδικοποίησης των καταστάσεων, αν και έχει παρατηρηθεί ότι ο συμβατικός τρόπος αναπαράστασης των bits με το επίπεδο της τάσης λειτουργεί και σε αναστρέψιμη λογική
4. Λογική μετάβαση διεργασιών: Η μετάβαση μιας λογικής κατάστασης σε αναστρέψιμα συστήματα πρέπει να γίνεται με μηδενική απώλεια θερμότητας, επομένως πρέπει να βρεθεί το φυσικό μέσο το οποίο να μειώνει το ποσοστό αυτό σε ελάχιστο, ιδανικά μηδενικό επίπεδο.
5. Λογικές πύλες: Όπως έχει προαναφερθεί, οι λογικές πύλες σε μια αναστρέψιμη υλοποίηση πρέπει να μπορούν να εκτελούν πράξεις ένα-προς-ένα μεταξύ μιας προηγούμενης και μιας επόμενης κατάστασης. Έτσι, είναι δυνατή η μετάβαση μπρος-πίσω και πίσω-μπρος με την ίδια ευκολία. Οι λογικές πύλες πλέον δεν περιορίζονται στις συμβατικές not, and, or, xor αλλά έχουν προταθεί αρκετές νέες υλοποιήσεις που βοηθούν την επίτευξη του στόχου αυτού.
6. Μηχανισμοί συγχρονισμού: Ιδανικά μια αναστρέψιμη μηχανή πρέπει να είναι κατασκευασμένη με τέτοια ακρίβεια ώστε οι καθυστερήσεις μεταξύ κάθε

υπολογιστικού μονοπατιού να είναι γνωστές, προβλέψιμες και ο αναμενόμενος χρόνος άφιξης των σημάτων να είναι ενσωματωμένος στα συστατικά που θα τα παραλάβουν.

7. Σχεδίαση συναρτησιακών μονάδων: Όταν έχει καθοριστεί το είδος των λογικών πυλών, επόμενο βήμα είναι η σχεδίαση συναρτησιακών μονάδων ψηλότερου επιπέδου όπως καταχωρητές, adders, κωδικοποιητές, πολυπλέκτες, κλπ. Είναι προφανές ότι πρέπει οι μονάδες αυτές να λειτουργούν με τρόπο τέτοιο ώστε αν μια είσοδος Α έχει ως αποτέλεσμα την έξοδο Β τότε από την έξοδο Β να μπορεί να παραχθεί η είσοδος Α.
8. Αρχιτεκτονική επεξεργαστών: Όσο η ανάγκη για αποδοτικότερη χρήση της ενέργειας αυξάνεται, τόσο περισσότερες αλλαγές χρειάζεται να γίνουν σε τμήματα του υπολογιστικού φάσματος μέχρι την ολοκληρωτική αλλαγή των Κεντρικών Μονάδων Επεξεργασίας (CPU). Η αλλαγή θα πρέπει να εφαρμοστεί κυρίως στην προγραμματιζόμενη αρχιτεκτονική για να μπορεί να αντικατοπτρίζει την βαθύτερη λογική αναστρεψιμότητα. Αυτό συμβαίνει γιατί λαμβάνοντας υπόψη το κόστος του αναστρέψιμου υπολογισμού δεν μπορούμε να περιμένουμε από μια μηχανή ή ένα μεταγλωττιστή να εντοπίζει αυτόματα ένα καλό αναστρέψιμο αλγόριθμο για διεκπεραίωση ενός υπολογισμού που ο προγραμματιστής έχει δηλώσει ρητά σε μη-αναστρέψιμη λογική. Γενικά, ο πιο αποδοτικός ανατρέψιμος αλγόριθμος για ένα έργο μπορεί να είναι δομημένος εντελώς διαφορετικά από τον πιο αποδοτικό μη αναστρέψιμο αλγόριθμο για το ίδιο έργο.
9. Γλώσσες σχεδίασης: Οι γλώσσες σχεδίασης όπως VHDL και Verilog αλλά και οι γλώσσες υψηλότερου επιπέδου όπως η C πρέπει να προσαρμοστούν ώστε να παρέχουν τις πιο κάτω σημαντικές και απαραίτητες αναστρέψιμες λειτουργίες: (1) χρήση τελεστών τύπου += για αναπαραγωγή της πλέον «νεκρής» πράξης της ανάθεσης τιμής. Η πράξη της ανάθεσης τιμής όπως την ξέρουμε, πχ x=3 όπου ανατίθεται στο x η τιμή 3, αποτελεί πράξη ριζικά αντίθετη με τα θεμελιώδη αξιώματα του αναστρέψιμου υπολογισμού και της μη διαγραφής πληροφορίας αφού οποιαδήποτε τιμή προϋπήρχε στη μεταβλητή x πριν την πιο πάνω ανάθεση έχει χαθεί. Επομένως σε ένα αναστρέψιμο περιβάλλον για να δώσουμε την τιμή 3 στη μεταβλητή x πρέπει με τη χρήση αναστρέψιμων τελεστών να τροποποιήσουμε την προηγούμενη τιμή της μεταβλητής. Για παράδειγμα, έστω x=7 με την πράξη x-=4, λαμβάνουμε το επιθυμητό αποτέλεσμα x=3. (2) δημιουργία συμμετρικών δομών ελέγχου ροής, (3) αναστρέψιμη

κλήση ρουτίνων, (4) κατάργηση της αυτόματης συλλογής σκουπιδιών καθώς οδηγεί σε απώλεια πληροφορίας και αναιρεί την αναστρεψιμότητα.

10. Αλγόριθμοι και Εφαρμογές: Η σχεδίαση των αλγορίθμων πρέπει πλέον να ακολουθεί το αναστρέψιμο πρότυπο. Αυτό αρχικά φαίνεται ακατόρθωτο στους προγραμματιστές οι οποίοι έχουν προσαρμοστεί στη χρήση μη αναστρέψιμων διεργασιών από την αρχή της προγραμματιστικής εποχής.

Ο αναστρέψιμος υπολογισμός βρίσκει εφαρμογές σε μια πληθώρα από πεδία. Το αντιπροσωπευτικότερο από αυτά είναι ο κβαντικός υπολογισμός. Στον κβαντικό υπολογισμό, ο υπολογισμός είναι μια ακολουθία από ατομικές, αδιάσπαστες πράξεις για κάθε κατάσταση. Αφού λοιπόν κάθε ατομική πράξη είναι και αναστρέψιμη, ο κβαντικός υπολογισμός είναι εξ ορισμού αναστρέψιμος. Η ανάπτυξη του κβαντικού υπολογισμού είναι ακράδαντα συνδεδεμένη με την ανάπτυξη του αναστρέψιμου υπολογισμού [11].

Συνοψίζοντας, ένα αναστρέψιμο μοντέλο πρέπει να είναι εξ' ολοκλήρου αναστρέψιμο, δηλαδή να χρησιμοποιεί αναστρέψιμες λογικές πράξεις σε επίπεδο υλικού και αναστρέψιμες υπολογιστικές πράξεις σε προγραμματιστικό επίπεδο. Το μοντέλο αυτό θα πρέπει να ενισχύσει το ήδη υπάρχων μοντέλο υπολογισμού (von Neumann) ούτως ώστε να λαμβάνει υπόψη και την ανάστροφη κατεύθυνση υπολογισμού. Επιπρόσθετα, το νέο αυτό μοντέλο, πρέπει να έχει ως βάση του τη μη απώλεια πληροφορίας, δηλαδή πρέπει να τηρεί την αρχή της αναστρεψιμότητας που δεν επιτρέπει τη διαγραφή πληροφοριών. Έτσι, η δημιουργία ενός τέτοιου μοντέλου θα βοηθήσει να ξεπεραστούν τα όρια που υπάρχουν στο συμβατικό υπολογιστικό μοντέλο.

2.2 Αναστρέψιμος προγραμματισμός

Ο αναστρέψιμος προγραμματισμός είναι το είδος προγραμματισμού που βασίζεται στο αναστρέψιμο μοντέλο υπολογισμού. Η υλοποίηση των προγραμμάτων γίνεται λαμβάνοντας υπόψη τόσο την εμπρόσθια κατεύθυνση υπολογισμού όσο και την ανάστροφη κατεύθυνση υπολογισμού.

Για την ανάπτυξη του τομέα αυτού χρειάζεται ανάπτυξη αναστρέψιμων γλωσσών προγραμματισμού υψηλού επιπέδου. Οι γλώσσες αυτές πρέπει να προσφέρουν λειτουργίες και

ιδιότητες εφάμιλλες της γλώσσας προγραμματισμού C. Επιπρόσθετα, οι αναστρέψιμες γλώσσες προγραμματισμού πρέπει να παρέχουν δομές διακλάδωσης και εντολές που να υποστηρίζουν την αναστρεψιμότητα. Μέχρι σήμερα έχουν προταθεί μερικές γλώσσες αναστρέψιμου προγραμματισμού για διάφορες χρήσεις [11].

Η αναστρέψιμη γλώσσα προγραμματισμού SRL (Simple Reversible Language) βασίζεται στο γεγονός ότι το σύνολο των συναρτήσεων υπολογίσιμων από μηχανές Turing είναι το ίδιο όπως το σύνολο των μερικώς αναδρομικών συναρτήσεων. Ένα υποσύνολο αυτού του συνόλου είναι το σύνολο των αρχέγονων αναδρομικών συναρτήσεων οι οποίες ορίζονται στους φυσικούς αριθμούς και έχουν πεπερασμένο αριθμό επαναλήψεων. Η SLR ορίζεται σε αυτό το σύνολο συναρτήσεων και υποστηρίζει τις πράξεις της αύξησης (INC var), της μείωσης (DEC var) και καθορισμένων επαναλήψεων (FOR var P, όπου P οι εντολές του βρόγχου). Η γλώσσα παρουσιάζει προβλήματα διαγραφής πληροφορίας όταν εκτελείται η πράξη της μείωσης σε μηδενικές μεταβλητές. Για διόρθωση του προβλήματος αυτού, επεκτάθηκε το σύνολο των αριθμών από το σύνολο των φυσικών αριθμών στο σύνολο των ακεραίων και προστέθηκε η πράξη της άρνησης (NEG var) η οποία αλλάζει το πρόσημο της μεταβλητής. Παρ' όλα αυτά όμως, χρειάζονται επιπλέον μεταβλητές για να κρατούν την προηγούμενη κατάσταση της γλώσσας. Η εκτεταμένη εκδοχή της γλώσσας ονομάζεται ESRL (Extended Simple Reversible Language) και χρησιμοποιείται για θεωρητικούς σκοπούς, για κατανόηση των βασικών ιδιοτήτων της απλούστερης αναστρέψιμης γλώσσας προγραμματισμού.

Η γλώσσα προγραμματισμού EPA (Event-Predicate-Action) είναι μια αναστρέψιμη γλώσσα προγραμματισμού η οποία μέσω μιας υποδομής κατά της διάρκεια εκτέλεσης δέχεται γεγονότα από το περιβάλλον του χρήστη και υπολογίζει το σύνολο των κατηγορημάτων που προκύπτουν καθώς και ποια από τα κατηγορήματα αυτά πρέπει να πληρούνται με την είσοδο των νέων γεγονότων. Η EPA ορίζει δύο τύπους ανάθεσης σε μεταβλητές, ένα κατασκευαστικό και ένα «καταστροφικό» όπου ο πρώτος μπορεί να αναστραφεί με την εκτέλεση της ανάστροφης του πράξης ενώ μόνο για τον δεύτερο χρειάζεται επιπλέον μνήμη για αποθήκευση της προηγούμενης τιμής (πριν την ανάθεση).

Η γλώσσα R προτάθηκε από τον Michael Frank και αρχικά προοριζόταν ως αναστρέψιμη γλώσσα προγραμματισμού για την αρχιτεκτονική Pendulum η οποία είχε ως στόχο την

ασυμπτωτικά μηδενική κατανάλωση ενέργειας κατά τον υπολογισμό. Η γλώσσα R ενσωματώνει μια πληθώρα αναστρέψιμων δομών οι οποίες βασίζονται στην αρχιτεκτονική Pendulum και στις αρχέγονες δομές της για την αναστρεψιμότητα τους.

Επιπρόσθετα έχουν προταθεί αρκετές συναρτησιακές και λογικές αναστρέψιμες γλώσσες προγραμματισμού οι οποίες στην πράξη παρουσιάζουν διαγραφές μνήμης και «παρενέργειες» που αναιρούν την έννοια της αναστρεψιμότητας.

Τέλος η γλώσσα Arrow αποτελεί μια γλώσσα εφάμιλλη της γλώσσας Janus (θα μελετηθεί εις βάθος στην ενότητα 2.3) με επιπλέον δομές δεδομένων και αναστρέψιμες πράξεις πολλαπλασιασμού και διαίρεσης. Παρόλα αυτά όμως η χρήση της είναι πολύ πιο άτεχνη και ακριβή.

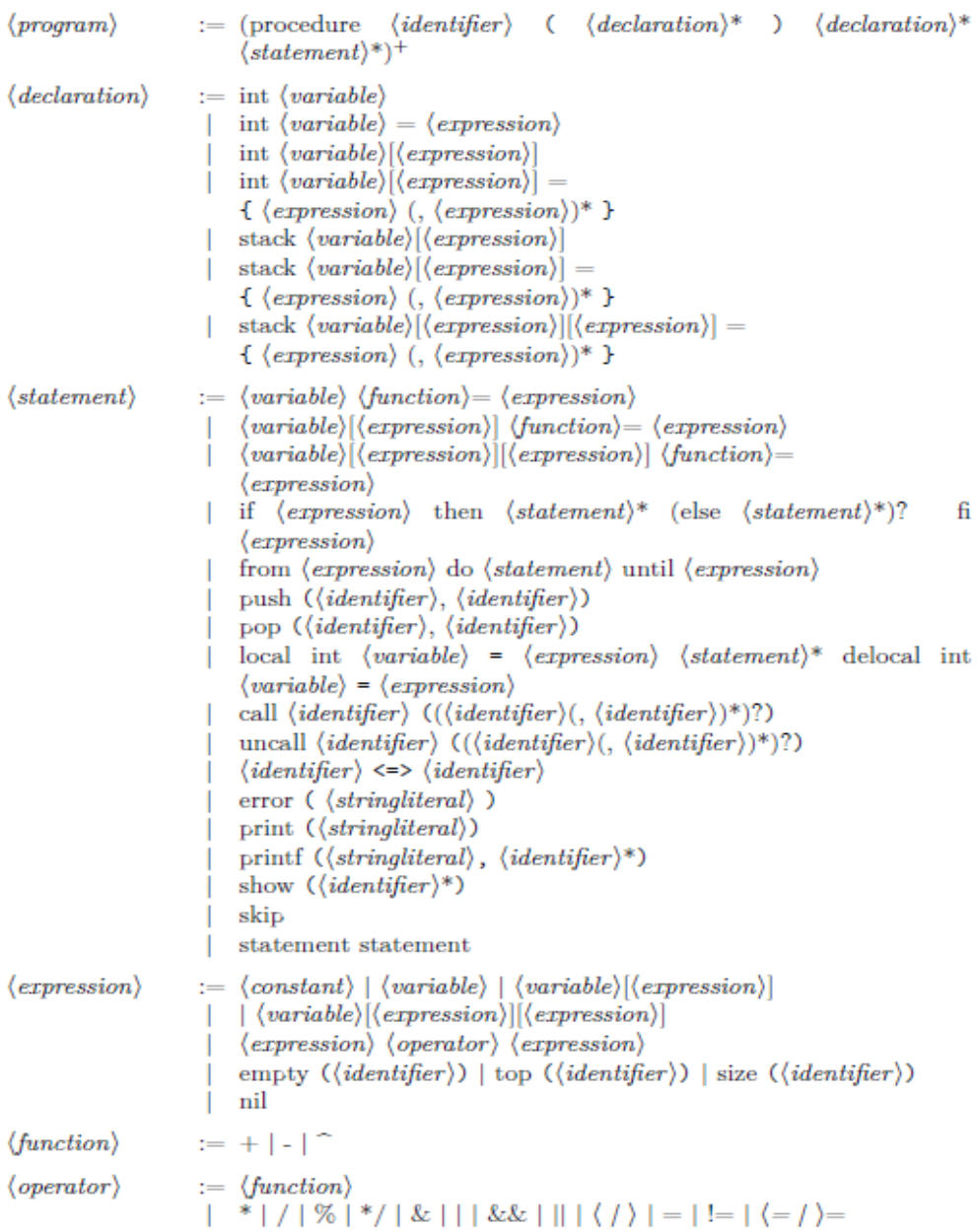
2.3 Η γλώσσα Janus

Πιο πάνω, στην ενότητα 2.1, αναφέρθηκαν προϋποθέσεις, περιορισμοί και δυσκολίες που αφορούν την λογική και υλική κατασκευή ενός πραγματικά και ολοκληρωτικά αναστρέψιμου μοντέλου. Το κομμάτι του αναστρέψιμου προγραμματισμού είναι κάτι που ακόμη και σήμερα έχει γνωρίσει ελάχιστη ανάπτυξη. Η γλώσσα Janus αποτελεί την πρώτη ολοκληρωμένη προσπάθεια για μια αναστρέψιμη γλώσσα προγραμματισμού. Η γλώσσα αρχικά προτάθηκε από τους Lutz και Derby το 1982 στο Caltech [8]. Οι Yokoyama, Axelsen και Glück παρουσίασαν μια εκτεταμένη εκδοχή της γλώσσας το 2007. Η γλώσσα αυτή θα παρουσιαστεί πιο κάτω [13][14]. Η γλώσσα Janus έχει υλοποιηθεί με βάση τη γλώσσα προγραμματισμού Python. Η σύνταξη της γλώσσας είναι απλή δίνοντας την εντύπωση ότι κάποιος με βασικές γνώσεις όσο αφορά αρχές προγραμματισμού μπορεί να την κατανοήσει. Όμως, για να μπορέσει κάποιος να μάθει τη γλώσσα και να μπορεί να υλοποιήσει αναστρέψιμα προγράμματα τα οποία να τηρούν τους κανόνες της αναστρεψιμότητας πρέπει να έχει καλιεργήσει τις γνώσεις του και να έχει αφιερώσει χρόνο όσο αφορά την κατανόηση του αναστρέψιμου υπολογισμού.

2.3.1 Σύνταξη

Η Janus είναι απλή, με τρόπο σύνταξης παρόμοιο με την C όσο είναι δυνατό. Είναι ισχυρή και υλοποιεί το αναστρέψιμο μοντέλο στις πράξεις ανάθεσης και τις δομές ελέγχου ροής του προγράμματος καθώς και επιτυγχάνει την αναστρεψιμότητα με την ανάστροφη κλήση συναρτήσεων [13][14].

Γενικά οι κανόνες σύνταξης της γλώσσας φαίνονται στο Σχήμα 2.1. Όταν ξεκινά ένα πρόγραμμα σε γλώσσα Janus, είναι απαραίτητη η δήλωση όλων των μεταβλητών που θα χρησιμοποιηθούν ως είσοδο στο πρόγραμμα καθώς και όλων των μεταβλητών που θα χρησιμοποιηθούν για την έξοδο. Δηλαδή αν η είσοδος του προγράμματος υπάρχει στον πίνακα input[] και η έξοδος του προγράμματος θα φανεί στη μεταβλητή output, πρέπει να έχουν δηλωθεί στην αρχή του προγράμματος. Δεν επιτρέπεται η δήλωση μεταβλητών σε μεταγενέστερο στάδιο με την εξαίρεση των τοπικών μεταβλητών οι οποίες θα παρουσιαστούν πιο κάτω. Παρατηρείται επίσης ότι ο εντοπισμός της εμβέλειας γίνεται χωρίς την χρήση ερωτηματικών (;) ή αγκύλων ({ }). Αντίθετα, ο εντοπισμός της εμβέλειας επιτυγχάνεται με την χρήση των λέξεων κλειδιών που εμπεριέχονται στη σύνταξη της γλώσσας. Τέλος, όσο αφορά τη δήλωση συναρτήσεων, πρώτα δηλώνεται η συνάρτηση main και στη συνέχεια μπορούν να δηλωθούν όσες επιπλέον συναρτήσεις κρίνεται απαραίτητο από τον προγραμματιστή. Οι συναρτήσεις δεν επιστρέφουν τιμή και γι’ αυτό το λόγο οι παράμετροι περνούνται δια αναφοράς.

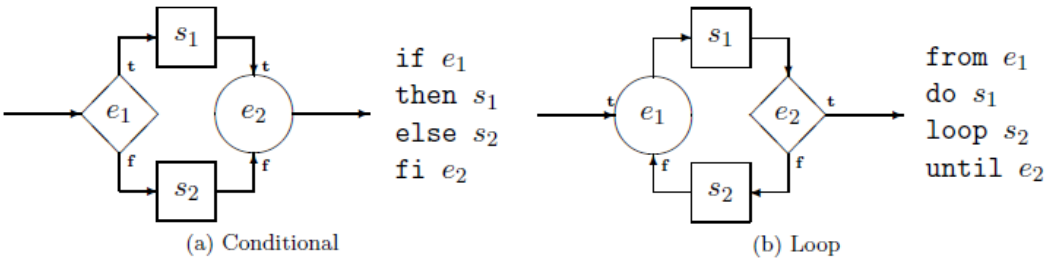


Σχήμα 2.1 Σύνταξη Janus

Οι τύποι δεδομένων που υποστηρίζει η γλώσσα περιορίζονται μέχρι στιγμής στο τύπο int που αναπαριστά ακέραιες τιμές, πίνακες ακεραίων, δυσδιάστατους πίνακες και στοίβες.

Πέρα από τις αρχικές μεταβλητές, η γλώσσα Janus επιτρέπει την δήλωση τοπικών μεταβλητών με την χρήση της λέξης κλειδί local. Οι τοπικές μεταβλητές πρέπει κατά τη δήλωση να τους ανατεθεί τιμή ρητά. Στη συνέχεια μπορούν να χρησιμοποιηθούν στο πρόγραμμα σαν κανονικές

μεταβλητές, να περαστούν ως παράμετροι σε συνάρτηση και να εκτελέσουν πράξεις υπολογισμού. Ωστόσο, όταν πλέον δεν χρειάζονται στο πρόγραμμα, η Janus δίνει την δυνατότητα αποδέσμευσης τους με τη χρήση της λέξης κλειδί delocal. Σημαντικό είναι να τονίσουμε ότι για να μπορέσουμε με επιτυχία να χρησιμοποιήσουμε τη εντολή αυτή τότε πρέπει να γνωρίζουμε τη τιμή που περιέχει η μεταβλητή που θέλουμε να αποδεσμεύσουμε ή να υπάρχει τρόπος να υπολογιστεί.



Εικόνα 2.2 (α) Δομή διακλάδωσης, (b) Βρόγχος επανάληψης

Στην Εικόνα 2.2 φαίνονται (a) η δομή διακλάδωσης if και (b) ο βρόγχος loop που υποστηρίζει η γλώσσα Janus. (a) Η δήλωση της δομής διακλάδωσης if γίνεται με τον ίδιο τρόπος όπως και σε κάθε γλώσσα δηλαδή if (condition) then S else P, όπου condition είναι η συνθήκη η οποία πρέπει να αποτιμηθεί ως αληθής για να εκτελεστεί το κομμάτι κώδικα S ενώ σε περίπτωση που είναι ψευδής τότε εκτελείται το κομμάτι P. Η Janus χρειάζεται επιπλέον ένα ισχυρισμό στο τέλος της δομής για να μπορεί να υποστηρίζει αναστρέψιμα προγράμματα. Μετά την εκτέλεση του κώδικα S αν δεν υπάρχει else ή του κώδικα P αν υπάρχει, έχουμε την εντολή fi assertion όπου ο ισχυρισμός assertion πρέπει να είναι αληθής αν το πρόγραμμα εκτέλεσε τον κώδικα S και ψευδής αν εκτέλεσε τον κώδικα P. (b) Ο βρόγχος επανάληψης της Janus δηλώνεται ως εξής: from assertion loop S until condition. Ο ισχυρισμός assertion πρέπει να αποτιμείται αληθής μόνο κατά την πρώτη επανάληψη του βρόγχου και είναι απαραίτητη προϋπόθεση για την εκτέλεση του κώδικα S. Τέλος, ο βρόγχος τερματίζει όταν η συνθήκη condition γίνει ψευδής.

Το κυριότερο συστατικό στο οπλοστάσιο της Janus, που την καθιστά πρότυπο γλώσσας αναστρέψιμου προγραμματισμού είναι η εμπρόσθια και ανάστροφη κλήση συναρτήσεων. Με τη χρήση της λέξης κλειδί call γίνεται κλήση μιας συνάρτησης με τον συμβατικό, εμπρόσθιο

τρόπο υπολογισμού. Με την χρήση της λέξης κλειδί uncall γίνεται κλήση μιας συνάρτησης ανάστροφα. Κατά την διάρκεια της ανάστροφης κλήσης τα βήματα εκτελούνται σε ανάστροφη σειρά και οι δομές διακλάδωσης και επανάληψης ελέγχουν την ροή με τον τρόπο που εξηγήθηκε πιο πάνω.

Στη γλώσσα υπάρχουν νέοι τελεστές ενημέρωσης οι οποίοι αντικαθιστούν τον τελεστή ανάθεσης (=). Οι τελεστές ενημέρωσης περιορίζονται στους τελεστές += και -= οι οποίοι εκτελούν τις αντίστροφες πράξεις της αύξησης και μείωσης αντίστοιχα. Εκτελώντας ένα πρόγραμμα σε ανάστροφη κατεύθυνση το οποίο περιέχει εντολή με τον τελεστή += θα εκτελέσει την ίδια εντολή με τον τελεστή -= για επίτευξη της αναστρεψιμότητας. Επιπρόσθετα, ο τελεστής ^= ο οποίος εκτελεί την πράξη XOR. Η πράξη XOR είναι εξ ορισμού ανάστροφη αφού αν έχουμε σε εμπρόσθια κλήση x=1 , x ^=1, τότε x = 0 και σε ανάστροφη κλήση x =0 , x^=1, τότε x=1. Τέλος, ο τελεστής <=> επιτρέπει την εναλλαγή μεταξύ δύο μεταβλητών (swap).

2.3.2 Παράδειγμα

Για καλύτερη κατανόηση των βασικών κανόνων σύνταξης της γλώσσας θα εξηγηθεί ένα παράδειγμα υλοποίησης του αλγόριθμο κρυπτογράφησης του Καίσαρα.

```
procedure main()
   a. int key = 35
   b. int input[] = { 0,0,0,0,0,0 }
   c. int output[size(input)] = {40,45,41,38,37,53}
   d. if key = 35 then
        i. uncall cipher(input,output,key)
   e. fi key = 35

procedure cipher(int input[], int output[], int key)
   a. local int i =0
   b. from i =0 loop
        i. input[i] <=> output[i]
       ii. output[i] += key
      iii. i+=1
   c. until i = size(input)
```

```
d. delocal int i = size(input)
```

Όπως φαίνεται πιο πάνω, η δήλωση της συνάρτησης main προηγείται οποιασδήποτε άλλης δήλωσης συνάρτησης. Στα βήματα a,b,c δίνονται οι μεταβλητές εισόδου και εξόδου της γλώσσα λόγω της αδυναμίας της γλώσσας να υποστηρίζει ορίσματα στη γραμμή εντολών για είσοδο από τον χρήστη. Κατά τη δήλωση των μεταβλητών γίνεται ανάθεση τιμής για το λόγο ότι δεν υπάρχει προηγούμενη τιμή και έτσι δεν χάνεται πληροφορία. Η συνάρτηση size() αποτελεί αρχέγονη συνάρτηση ενσωματωμένη στη γλώσσα και επιστρέφει το μέγεθος ενός πίνακα ή λίστας. Στη συνέχεια, στο βήμα d φαίνεται η δομή διακλάδωσης if. Αν η μεταβλητή key έχει την τιμή 35 τότε εκτελείται ο επόμενος κώδικας. Ο ισχυρισμός στο βήμα e είναι τετριμμένος καθώς δεν υπάρχει τμήμα else και δεν εκτελείται κάποια πράξη μετατροπής της τιμής της μεταβλητής key. Στο πιο πάνω παράδειγμα η συνθήκη είναι αληθής επομένως προχωρούμε στην ανάστροφη κλήση της συνάρτησης cipher στο βήμα d.i.

Η συνάρτηση cipher παίρνει ως παραμέτρους το κλειδί, την είσοδο και τον πίνακα όπου θα αποθηκεύεται η έξοδος. Στα βήματα a και d, γίνεται δήλωση της τοπικής μεταβλητής i και αποδέσμευση της, αντίστοιχα. Γνωρίζουμε ότι ο βρόγχος θα εκτελεστεί τόσες φορές όσες και το μέγεθος της εισόδου επομένως είναι γνωστό ότι με το πέρας του βρόγχου η τιμή του i θα είναι ίση με το μέγεθος της εισόδου κατά την αποδέσμευση. Ακόμα, στο πιο πάνω παράδειγμα, φαίνεται η χρήση του βρόγχου επανάληψης της γλώσσας Janus. Στο βήμα b έχουμε τον ισχυρισμό ότι κατά την είσοδο στο βρόγχο η τιμή του i πρέπει να είναι ίση με 0. Αυτό είναι απαραίτητο να ισχύει μόνο κατά την είσοδο της ροής στο βρόγχο και στη συνέχεια να είναι πάντα ψευδής. Μέσα στο βρόγχο γίνεται ανταλλαγή των τιμών μεταξύ της εισόδου και της εξόδου χρησιμοποιώντας τον τελεστή <=> και έπειτα ενημερώνεται η έξοδος για την τιμή του κλειδιού με χρήση του τελεστή +=. Τέλος, αυξάνεται ο μετρητής και η ροή εκτέλεσης φτάνει στη συνθήκη όπου σε περίπτωση που είναι αληθής έχουμε επανάληψη του βρόγχου ενώ σε αντίθεση περίπτωση έχουμε έξοδο από το βρόγχο.

Στο συγκεκριμένο παράδειγμα έχουμε ανάστροφη κλήση της συνάρτησης cipher επομένως η εκτέλεση θα ξεκινήσει από κάτω προς τα πάνω. Η αρχική τιμή της μεταβλητής i θα είναι ίση με το μέγεθος της εισόδου και καθώς προχωρούμε στο βρόγχο, θα μειώνεται η τιμή της εξόδου ανάλογα με την τιμή του κλειδιού (χρήση αντίστροφου τελεστή του -=) και μετά θα γίνει η

ανταλλαγή. Επομένως, μετά το πέρας της εκτέλεσης της συνάρτησης cipher, ο πίνακας Ouput θα είναι μηδενικός και ο πίνακας input θα έχει τις τιμές { 5,10,11,3,2,18}.

2.3.3 Προηγούμενες εργασίες

Παρά το γεγονός ότι η έννοια της αναστρεψιμότητας εμφανίστηκε και μελετάτε εδώ και μερικές δεκαετίες, ο αναστρέψιμος προγραμματισμός δεν έχει αναπτυχθεί σε επίπεδο όπου να μπορεί να χρησιμοποιηθεί ευρέως. Αυτό συμβαίνει γιατί υπάρχει έλλειψη βασικών μεθοδολογιών και βιβλιοθηκών για οικογένειες αλγορίθμων στη γλώσσα Janus.

Αλγόριθμοι Ταξινόμησης:
Με το πρόβλημα αυτό ασχολήθηκαν οι Axelsen και Yokoyama παρουσιάζοντας τρεις μεθοδολογίες για δημιουργία αναστρέψιμων αλγορίθμων ταξινόμησης και δύο θεμελιώδεις όρους που να αξιολογούν τους αναστρέψιμους αλγορίθμους. Στο άρθρο παρουσιάζονται υλοποιήσεις των γνωστών αλγορίθμων ταξινόμησης όπως bubble sort, insertion sort, selection sort, merge sort και quick sort [2].

Η πρώτη τεχνική που χρησιμοποίησαν για την δημιουργία των αλγορίθμων είναι η ενσωμάτωση ιστορικού. Η ενσωμάτωση ιστορικού αναγκάζει τον προγραμματιστή να αποθηκεύει ένα bit πληροφορίας κάθε φορά που αποφασίζει για την εκτέλεση ενός βήματος. Αυτό έχει ως αποτέλεσμα την παρουσίαση των βημάτων εκτέλεσης ως μια ακολουθία από bits.

Στη συνέχεια, έδειξαν ότι με την χρήση αρθρωτού προγραμματισμού σε μέγιστο επίπεδο, είναι δυνατή η δημιουργία προγραμμάτων με την σύμβαση της εμπρόσθιας-ανάστροφης κλήσης. Οι αλγόριθμοι που δημιουργούνται με την σύμβαση αυτή έχουν την εξής μορφή: εμπρόσθια κλήση συνάρτησης, αποθήκευση εξόδου, ανάστροφη κλήση συνάρτησης. Με αυτό τον τρόπο επιτυγχάνεται αποδέσμευση μνήμης (σκουπίδια) όμως γίνεται χρήση περισσότερης μνήμης για ολόκληρη την εκτέλεση του προγράμματος.

Τέλος, για την υλοποίηση των αλγορίθμων ταξινόμησης, ανακάλυψαν ότι μπορούν να χρησιμοποιήσουν το κόλπο μετάθεσης ταυτότητας (identity permutation trick) για ευκολότερο εντοπισμό της αρχικής θέσης των στοιχείων μετά την ταξινόμηση.

Αλγόριθμοι σε γράφους:

Η πτυχιακή εργασία της Sarah Vang Nøhr από το Πανεπιστήμιο της Κοπεγχάγης αφορά την υλοποίηση αναστρέψιμων αλγορίθμων σε γράφους στη γλώσσα Janus. Η Janus όμως δεν υποστηρίζει την δομή του γράφου. Στα πλαίσια της διπλωματικής χρησιμοποίησε τη μέθοδο αναπαράστασης γράφων υπό μορφή λίστας γειτνίασης και δημιούργησε βοηθητικές δομές δεδομένων όπως ουρά, δυαδική σωρός, ελάχιστη ουρά προτεραιότητας και ξένα σύνολα [10].

Η Sarah Vang Nøhr παρουσίασε δύο ομάδες αλγορίθμων. Η πρώτη αποτελείται από τα δύο βασικά είδη αλγορίθμων διάσχισης, κατά βάθος αναζήτηση (Depth First Search – DFS) και κατά πλάτος αναζήτηση (Breadth First Search – BFS). Γίνεται χρήση και λεπτομερής περιγραφή του αλγόριθμου DFS στο κεφάλαιο 3, ενότητα 3.

Επιπρόσθετα, η άλλη ομάδα αλγορίθμων που δημιουργήθηκε αφορά την εύρεση ελάχιστων γεννητορικών δέντρων και την εύρεση ελάχιστου μονοπατιού. Η υλοποίηση αφορά τους αλγόριθμους Prim, Kruskal και Dijkstra αντίστοιχα.

2.4 Πιστότητα και Υγιεινή

Για την αξιολόγηση των αναστρέψιμων αλγορίθμων, οι Axelsen και Yokoyama [2] προτείνουν τους εξής τρόπους:

1. Ένας αναστρέψιμος αλγόριθμος θεωρείται *πιστή (faithful)* προσομοίωση του αντίστοιχου μη αναστρέψιμου αν διατηρεί την χρονική πολυπλοκότητα του αυθεντικού αλγορίθμου χωρίς την προσθήκη επιπλέον ασυμπτωτικού χρόνου και περιορίζει το επιπλέον κόστος σε μνήμη (σκουπίδια) ως μια συνάρτηση g(n) όπου n το μέγεθος της εισόδου.
2. Ένας αναστρέψιμος αλγόριθμος θεωρείται *υγιής (hygienic)* προσομοίωση του αντίστοιχου μη αναστρέψιμου αν ισχύουν όλα τα πιο πάνω και η συνάρτηση που περιορίζει τα σκουπίδια είναι ελάχιστη, δηλαδή δεν μπορεί να μειωθεί περαιτέρω.

Κεφάλαιο 3

Αλγόριθμοι Κωδικοποίησης

Οι αλγόριθμοι κωδικοποίησης είναι αλγόριθμοι οι οποίοι έχουν έμφυτη την ανάγκη για αναστρεψιμότητα. Η κωδικοποίηση ενός χαρακτήρα μπορεί να θεωρηθεί και ως συμπίεση αφού ο ίδιος χαρακτήρας πλέον αναπαρίσταται με μερικά δυφία. Επομένως, η αντίθετη πράξη της κωδικοποίησης/συμπίεσης είναι η πράξη της αποκωδικοποίησης/αποσυμπίεσης, όπου έχουμε την ανάγκη για επαναφορά των δυφίων στον αρχικό χαρακτήρα τον οποίο αναπαριστούσαν. Αυτή λοιπόν η έμφυτη αναστρεψιμότητα είναι ιδιαίτερα ενδιαφέρουσα όσο αφορά την συγχώνευση του αλγόριθμου κωδικοποίησης και αποκωδικοποίησης και έτσι, για τον λόγο αυτό, σε αυτό το κεφάλαιο θα παρουσιαστούν αναστρέψιμες υλοποιήσεις αλγορίθμων κωδικοποίησης μεταβλητού μήκους που αναπτύχθηκαν στα πλαίσια της διπλωματικής εργασίας. Οι αλγόριθμοι κρατούν σταθερή τη χρονική πολυπλοκότητα καθώς προσθέτουν επιπλέον «σκουπίδια» σε ανεχτά επίπεδα.

3.1 Ο αλγόριθμος Leveling

Ο αλγόριθμος Leveling αποτελεί μια μορφή κωδικοποίησης μεταβλητού μήκους η οποία βασίζεται στην έννοια των επιπέδων για την διεκπεραίωση του. Προτάθηκε το 2016 από τον Javier Joglar Alcubilla [3].

Ο αλγόριθμος Leveling βασίζεται στην έννοια των επιπέδων για την κωδικοποίηση των χαρακτήρων. Τα επίπεδα ορίζονται ως οι γραμμές ενός πίνακα και κάθε νέο επίπεδο αυξάνει το μήκος της κωδικοποίησης κατά ένα δυφίο. Για παράδειγμα, αν ένας χαρακτήρας κωδικοποιείται στο πρώτο επίπεδο και έχει κωδικοποίηση μήκους τριών δυφίων, τότε κάθε χαρακτήρας που κωδικοποιείται σε επόμενο επίπεδο θα έχει κωδικοποίηση μεγαλύτερη ή ίση των τεσσάρων δυφίων. Επιπρόσθετα, βασικές έννοιες για την υλοποίηση του αλγορίθμου είναι η δομή initial_length η οποία υπολογίζεται με βάση την είσοδο και δηλώνει τον αριθμό αρχικό επίπεδο κωδικοποίησης, δηλαδή αν το initial_length υπολογιστεί ως 3 τότε το ελάχιστο μήκος κωδικοποίησης για τη συγκεκριμένη είσοδο θα είναι 3 δυφία. Σημαντική παρατήρηση είναι το γεγονός ότι αυτό δηλώνει ότι ο αριθμός του επιπέδου δεν ξεκινάει πάντα από το 1. Για παράδειγμα, αν η εκτέλεση του αλγόριθμου χρειάζεται σύνολο 2 επίπεδα και το initial_length έχει υπολογιστεί 3 τότε έχουμε κωδικοποιήσεις τριών και τεσσάρων δυφίων. Τέλος, ο πίνακας reduced_code χρησιμοποιείται για να ορίσει πόσοι χαρακτήρες αντιστοιχούν σε κάθε επίπεδο ενώ στη πρώτη θέση δείχνει το μήκος του αρχικού επιπέδου.

3.1.1 Ο αυθεντικός αλγόριθμος

Ο αυθεντικός αλγόριθμος έχει ως εξής: αρχικά γίνεται υπολογισμός των πιθανοτήτων εμφάνισης για κάθε χαρακτήρα της εισόδου και ταξινόμηση τους κατά φθίνουσα σειρά. Αυτό συμβαίνει γιατί θέλουμε ο χαρακτήρας με την μεγαλύτερη πιθανότητα εμφάνισης να κωδικοποιείται με το μικρότερο δυνατό μήκος για καλύτερη απόδοση συμπίεσης. Στη συνέχεια, γίνεται ο υπολογισμός του αρχικού μήκους κωδικοποίησης initial_length. Ο υπολογισμός γίνεται με τον εξής τρόπο: έστω ότι το άθροισμα όλων των πιθανοτήτων εμφάνισης sum =1. Τότε αθροίζω τις πιθανότητες εμφάνισης των ταξινομημένων χαρακτήρων μέχρι το άθροισμα τους να ξεπεράσει το μισό (1/2) και μετρώ πόσα διαφορετικά στοιχεία χρειάστηκαν μέχρι το σημείο αυτό. Έπειτα, αντιστοιχώ τον αριθμό του μετρητή με βάση τον Πίνακα 3.1 και παίρνω την τιμή του αρχικού μήκους initial_length.

Αριθμός συμβόλων στο sum/2	1	2-3	4-7	8-15	16-31	..	$2^{n-1} - (2^n - 1)$
Αρχικό μήκος (initial_length)	1	2	3	4	5	..	n

Πίνακας 3.1 Αντιστοιχεία αρχικού μήκους

Στη συνέχεια γίνεται υπολογισμός του πίνακα reduced_code. Για τον υπολογισμό του reduced_code χρειαζόμαστε το μετρητή j=1 ο οποίος κρατά τη θέση του εκάστοτε επιπέδου, τη μεταβλητή d=initial_length και τη μεταβλητή b που καθορίζει τον κάθε χαρακτήρα. Επαναλαμβάνουμε για κάθε χαρακτήρα b: Διπλασίασε το d, χώρισε το άθροισμα ων πιθανοτήτων εμφάνισης σε d κομμάτια. Για κάθε κομμάτι: αν [P(b)<1/d και (P(b+1)/2)>1/d)] ή P(b) >= 1/d τότε αύξησε τη τιμή reduced_code[j] κατά 1, τροποποίηση τη τιμή P(b+1) ως εξής P(b+1)=P(b+1)+P(b) -1/d, μετακίνησε το b στον επόμενο χαρακτήρα, αλλιώς αύξησε τον μετρητή j για να προχωρήσει στο επόμενο επίπεδο και επέστρεψε στο αρχικό βήμα.

Στο επόμενο βήμα δημιουργείται ένας δισδιάστατος πίνακας με γραμμές τόσες όσες και το μήκος του πίνακα reduced_code -1 όπου κάθε γραμμή i>=1 είναι χωρισμένη σε 2^i τμήματα τα οποία έχουν εναλλάξ τιμές 0 ή 1, ξεκινώντας από το 0. Αθροίζοντας τις πιθανότητες εμφάνισης σε αναλογία με τον πίνακα, ένας χαρακτήρας ανήκει στο εκάστοτε τμήμα κάθε επιπέδου αν η πιθανότητα εμφάνισης του ανήκει τουλάχιστον στο μισό τμήμα. Έτσι γίνεται αντιστοίχιση κάθε χαρακτήρα με την κωδικοποίηση του. Ο χαρακτήρας με την μεγαλύτερη πιθανότητα εμφάνισης πάρει κωδικοποίηση με μόνο το σύμβολο 0 μήκους ίσου με το initial_length και ο χαρακτήρας με την μικρότερη πιθανότητα εμφάνισης θα πάρει κωδικοποίηση με μόνο 1 μήκους ίσου με το μέγιστο επίπεδο.

Τέλος, για κάθε στοιχείο της εισόδου μπορούμε να δώσουμε μια μετατροπή η οποία προσδίδει νόημα στην κωδικοποίηση μας ως εξής: m(1) =0 , m(i)= m(i-1)+1 όσο προχωρούμε σε στοιχεία που βρίσκονται στο ίδιο επίπεδο, m(i)=[m(i-1)+1]*2 όταν προχωρούμε σε στοιχεία σε επόμενο επίπεδο.

3.1.2 Ο αναστρέψιμος αλγόριθμος

Πιο κάτω δίνονται επιγραμματικά τα βήματα του αυθεντικού αναστρέψιμου αλγορίθμου τα οποία πρέπει να τροποποιηθούν ούτως ώστε να λειτουργούν σε αναστρέψιμο επίπεδο.

Βήματα αλγορίθμου:

1. Εύρεση, υπολογισμός και ταξινόμηση πιθανοτήτων εμφάνισης
2. Υπολογισμός initial length
3. Δημιουργία πίνακα reduce code
4. Κωδικοποίηση leveling
5. Κωδικοποίηση meaning

Ο αλγόριθμος ξεκινάει με την διαδικασία η οποία υπολογίζει τον αριθμό εμφανίσεων κάθε χαρακτήρα εισόδου και στη συνέχεια υπολογίζει την πιθανότητα εμφάνισης κάθε χαρακτήρα. Πριν το πέρας της διαδικασίας γίνεται ταξινόμηση των πιθανοτήτων σε φθίνουσα σειρά. Η πιο κάτω διαδικασία υλοποιείται στις συναρτήσεις generate_probabilities, bsortDecreasing που θα δοθούν στην Ενότητα 3.1.3.

Η πιο κάτω διαδικασία παίρνει ως είσοδο τον πίνακα input που περιέχει την είσοδο προς κωδικοποίηση και τον δισδιάστατο πίνακα alphabet και επιστρέφει ως έξοδο τις πιθανότητες εμφάνισης των χαρακτήρων P[].

1. Αρχικοποίησε πίνακα alphabet[1] με το αλφάβητο (26 γράμματα αγγλικού αλφάβητου)
2. local int i=0
3. from i=0 loop
    a. alphabet[0][input[i]]+=1
    b. Αύξησε μετρητή i
4. until i=size(input)
5. delocal int i=size(input)
6. Αναστρέψιμη ταξινόμηση πίνακα εμφανίσεων (alphabet[0])
7. Υπολόγισε τον αριθμό διαφορετικών στοιχείων, diff
8. P[b] = alphabet[0] (όπου alphabet[1] =b) / size(input) για κάθε b

Η πιο πάνω διαδικασία εκτελείται όπως και στον αυθεντικό αλγόριθμο και για κάθε στοιχείο στην είσοδο μετρά τις φορές εμφάνισης και στη συνέχεια υπολογίζει την πιθανότητα εμφάνισης κάθε στοιχείου.

Αφού πλέον έχουμε υπολογίσει τις πιθανότητες εμφάνισης με αναστρέψιμο αλγόριθμο, πρέπει να υπολογίσουμε το initial_length, το αρχικό μήκος δηλαδή της κωδικοποίησης. Ο πιο κάτω αλγόριθμος παίρνει ως είσοδο τις πιθανότητες εμφάνισης κάθε στοιχείου της εισόδου και υλοποιείται στις συναρτήσεις calculate_InitialLength και match_corresponding που θα δοθούν στο 3.1.3.

1. local int aggregate=0
2. local int i=0
3. from i = 0 loop
    a. aggregate+= P[i]
    b. if aggregate <= 1 /2 then
        i. Αύξησε initial_length κατά 1
    c. fi aggregate <= 1 /2
    d. Αύξησε μετρητή i κατά 1
4. until i=size(input)
5. delocal int i=size(input)
6. delocal int aggregate =1
7. Άλλαξε τη νέα τιμή του initial_length με floor(log(initial_length))+1

Στη πιο πάνω διαδικασία αθροίζουμε τις πιθανότητες εμφάνισης ξεκινώντας από την μεγαλύτερη μέχρι το άθροισμα να ξεπεράσει το 1/2. Στη συνέχεια αντικαθιστούμε το μετρητή με την τιμή που του αναλογεί από τον Πίνακα 3.1.

Έπειτα, όπως και στην αυθεντική εκδοχή του αλγορίθμου, ακολουθεί ο υπολογισμός του πίνακα reduced_code. Ο πίνακας reduced_code δείχνει πόσα στοιχεία αντιστοιχούν σε κάθε επίπεδο. Η διαδικασία παίρνει ως είσοδο τις πιθανότητες εμφάνισης και το initial_length που υπολογίσαμε πιο πάνω. Η συνάρτηση calculate_ReducedCode αποτελεί την υλοποίηση του πιο κάτω αλγόριθμου στη γλώσσα Janus.

1. Αρχικοποίησε τη πρώτη θέση του reduced_code με την τιμή initial length
2. Αρχικοποίησε μετρητή d με τη τιμή initial length
3. local int b = 0
4. from b =0 && j = 1 loop
    a. local int aux = d
    b. d+=aux
    c. delocal int aux = d/2
    d. if (P[b] < 1/d && (P[b]+P[b+1]/2 >1/d)) || P[b] >= 1/d then
        a. reduced_code[j]+=1
        b. P[b+1] +=P[b] – 1/d
        c. Αύξησε b για να προχωρήσει στο επόμενο στοιχείο
    e. else
        a. Αύξησε j για να προχωρήσει στο επόμενο επίπεδο
    f. fi reduced_code[j] > 0
5. until b = diff
6. delocal int b = diff

Ο πίνακας reduced code περιέχει πληροφορίες σχετικά με το πόσοι χαρακτήρες εμπεριέχονται σε κάθε επίπεδο κωδικοποίησης. Η πρώτη θέση του πίνακα περιέχει το αρχικό μήκος κωδικοποίησης και κάθε επόμενη θέση τον αριθμό των χαρακτήρων στο εκάστοτε επίπεδο κωδικοποίησης. Ο αλγόριθμος εκτελεί τα ίδια βήματα με τον αυθεντικό με την εξαίρεση ότι τώρα προστίθενται οι ισχυρισμοί στις δομές διακλάδωσης και επανάληψης για επίτευξη της αναστρεψιμότητας.

Μετά τον υπολογισμό των βασικών συστατικών του αλγορίθμου, ακολουθεί η κωδικοποίηση. Για να γίνει δυνατή η κωδικοποίηση χρειάζεται η δημιουργία ενός δυσδιάστατου πίνακα ο οποίος να έχει γραμμές όσες και την τιμή του μέγιστου επιπέδου κωδικοποίησης. Κάθε γραμμή i είναι χωρισμένη σε 2^i τμήματα τα οποία έχουν εναλλάξ τιμές 0 και 1, ξεκινώντας από 0. Τέλος, η κωδικοποίηση γίνεται κλιμακώνοντας τον πίνακα αυτό με τον πίνακα P που περιέχει τις πιθανότητες των χαρακτήρων και για κάθε χαρακτήρα βρίσκουμε την κωδικοποίηση που του αναλογεί σε κάθε επίπεδο βρίσκοντας σε πιο τμήμα αντιστοιχεί τουλάχιστον η μισή πιθανότητα εμφάνισης του. Ο πιο κάτω αλγόριθμος υλοποιείται στη συνάρτηση Leveling και

δέχεται ως παραμέτρους τη μεταβλητή initial_length, τον πίνακα reduced_code και τη βοηθητική στοίβα aux που θα αποθηκεύσει τα επιπλέον σκουπίδια.

1. local queue arxi
2. local queue telos
3. local int letters[] = Ταξινομημένη είσοδος σε φθίνουσα σειρά με βάση τον αριθμό εμφανίσεων
4. local int codes[][] = Αρχικοποιημένος με 0 και 1 εναλλάξ
5. local int metritis=0
6. local int i= 0
7. from i=0 loop
    a. Βρες την αρχή του στοιχείου i στον πίνακα letters
    b. Βρες το τέλος του στοιχείου i στον πίνακα letters
    c. Εισαγωγή αρχής και τέλους στις ουρές arxi και telos αντίστοιχα
    d. Προχώρησε στο επόμενο στοιχείο
8. until i=diff
9. delocal int i = diff
10. local int pos =1
11. from size(arxi) = diff loop
    a. local int start =Εξαγωγή πρώτου στοιχείου από arxi
    b. local int finish = Εξαγωγή πρώτου στοιχείου από telos
    c. local int miso = (start + finish)/2
    d. local int k = μέγιστο επίπεδο
    e. from κ = μέγιστο επίπεδο
        I. Κωδικοποίηση(i) U {codes[k][miso]}
        II. k-=1
    f. until k < 0
    g. delocal int k=-1
    h. delocal int miso = (start+finish)/2
    i. Αποθήκευση start και finish στη στοίβα aux για σκοπούς αναστρεψιμότητας
    j. metritis +=1

k. delocal int finish = 0
l. delocal int start =0
m. if metritis >= reduced_code[pos] then
    I. pos+=1
    II. metritis -= reduced_code[pos-1]
n. fi metritis = 0

12. until size(arxi) = 0
13. delocal int metritis =0
14. delocal int codes[][] =Αρχικοποιημένος με 0 και 1 εναλλάξ
15. local int letters[] = Ταξινομημένη είσοδος σε φθίνουσα σειρά με βάση τον αριθμό εμφανίσεων
16. delocal stack telos = nil
17. delocal stack starts = nil

Η ιδέα του αλγορίθμου είναι απλή, αν οι εμφανίσεις ενός στοιχείου αντιστοιχούν σε περισσότερο από το μισό μιας ομάδας 0 ή 1, τότε για το συγκεκριμένο επίπεδο, το συγκεκριμένο στοιχείο παίρνει την αντίστοιχη κωδικοποίηση. Ο πίνακας codes του πιο πάνω αλγορίθμου φαίνεται στον Πίνακα 3.2.

0								1							
0				1				0				1			
0		1		0		1		0		1		0		1	
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Πίνακας 3.2 Ο πίνακας codes για παράδειγμα κωδικοποίησης 4 επιπέδων

Τέλος, η κωδικοποίηση meaning έρχεται για να δώσει νόημα στην κωδικοποίηση που μόλις έχουμε ολοκληρώσει. Αυτό συμβαίνει γιατί δίνει μια «λογική σειρά» στην κωδικοποίηση leveling. Τα στοιχεία πλέον παίρνουν ένα αριθμό που αντικατοπτρίζει τη θέση τους όσο αφορά τις πιθανότητες εμφάνισης καθώς και την ομαδοποίηση τους ανά επίπεδα.

Η συνάρτηση που υλοποιεί τον πιο κάτω αλγόριθμο είναι η συνάρτηση find_meaning και δέχεται ως παραμέτρους τον πίνακα reduced_code

Υπολόγισε το meaning κάθε διαφορετικού στοιχείου ως εξής :

1. ίδιο επίπεδο : meaning[i]+=meaning[counter-1]+1
2. διαφορετικό επίπεδο : $(\text{meaning}[\text{counter} - 1] + 1)^2$

Συνοψίζοντας, τα βήματα του αναστρέψιμου αλγορίθμου εκτελούνται με τον ίδιο τρόπο όπως και στον αυθεντικό αλγόριθμο. Το καινούριο συστατικό στην αναστρέψιμη εκδοχή, είναι η εύρεση των απαραίτητων ισχυρισμών για την ορθή διακλάδωση της ροής εκτέλεσης του αλγορίθμου καθώς και η εύρεση της απαραίτητης πληροφορίας όσο αφορά ποιες μεταβλητές είναι δυνατό να αποδεσμεύσω και ποιες χρειάζονται για την αναστρεψιμότητα καθώς δεν μπορεί να υπολογιστεί με άλλα μέσα η τιμή τους.

3.1.3 Η υλοποίηση στη γλώσσα Janus

Σε αυτή την ενότητα θα δοθεί η υλοποίηση του αλγόριθμου κωδικοποίησης Leveling στην γλώσσα Janus. Η γλώσσα Janus η πιο εύχρηστη αναστρέψιμη γλώσσα προγραμματισμού και προσφέρει τις περισσότερες δυνατότητες για ανάπτυξη αναστρέψιμων προγραμμάτων. Όμως, όπως έχει αναφερθεί στο Κεφάλαιο 2, οι περιορισμοί τις γλώσσας μπορεί να σταθούν τροχοπέδη στην επιτυχή «μετάφραση» ενός αλγορίθμου στη σύνταξη της γλώσσας.

Τα βήματα του αλγορίθμου και οι συναρτήσεις οι οποίες τα υλοποιούν φαίνονται στον Πίνακα 3.3.

Εύρεση, υπολογισμός και ταξινόμηση πιθανοτήτων εμφάνισης	generate_probabilities, bsortDecreasing
Υπολογισμός initial length	calculate_InitialLength, match_corresponding
Δημιουργία πίνακα reduce code	calculate_ReducedCode
Κωδικοποίηση leveling	Leveling
Κωδικοποίηση meaning	find_meaning

Πίνακας 3.3 Αντιστοιχία βημάτων και συναρτήσεων

Για την υλοποίηση της συνάρτησης bsortDecreasing στη γλώσσα Janus χρειάζεται ένας βοηθητικός πίνακας μήκους ίσου με την είσοδο όπου για κάθε στοιχείο δείχνει πόσες θέσεις μετακινήθηκε ο κάθε χαρακτήρας.

Εξ ορισμού οι πιθανότητες παίρνουν τιμές στο διάστημα μεταξύ [0,1], επομένως είναι αδύνατη η αναπαράσταση τους σε αυτή της μορφή στη γλώσσα Janus διότι δεν υποστηρίζει ακόμη αριθμούς κινητής υποδιαστολής float. Για τον λόγο αυτό οι πιθανότητες εμφάνισης έχουν αντικατασταθεί με αριθμό εμφανίσεων. Επομένως αν για ένα χαρακτήρα είχαμε πιθανότητα εμφάνισης 3/8, τώρα θα έχουμε 3 φορές εμφάνισης, όπου το σύνολο των εμφανίσεων όλων των διαφορετικών χαρακτήρων είναι 8. Για τον υπολογισμό των διαφορετικών στοιχείων και την καταμέτρηση των ξεχωριστών εμφανίσεων, χρησιμοποιήθηκε ένας πίνακας μήκους ίσου με το αλφάβητο όπου για κάθε εμφάνιση του χαρακτήρα j αυξάνεται ο μετρητής στη θέση j+1 κατά 1.

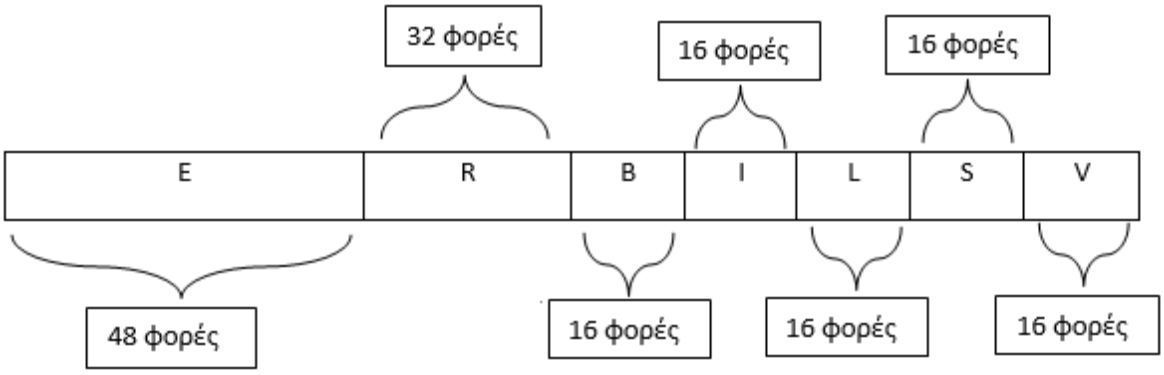
Στη συνέχεια, για τον υπολογισμό του initial_length η συνάρτηση calculate_InitialLength μετρά τον ελάχιστο αριθμό των στοιχείων που το άθροισμα των εμφανίσεων τους σε φθίνουσα σειρά ξεπερνά τη τιμή μέγεθος εισόδου/2. Η αντιστοίχιση του αριθμού αυτού με το πραγματικό intial_length δεν μπορεί να γίνει κατά την εκτέλεση στη γλώσσα Janus, επομένως οι αντιστοιχίες δόθηκαν στατικά σε δυσδιάστατο πίνακα.

Λόγω των περιορισμών της γλώσσας Janus όσο αφορά τους τύπους δεδομένων, ο αλγόριθμος υλοποιήθηκε για χρήση μόνο ακέραιων αριθμών. Για τον υπολογισμό του πίνακα reduced_code στην συνάρτηση calculate_ReducedCode αρχικοποιούμε την μεταβλητή d=initial length*2 και ελέγχουμε σε κάθε επανάληψη αν [$P(b)<1/d$ and ($P(b)+(P(b+1)/2)>1/d$)] or [$P(b)>=1/d$]. Στην αναστρέψιμη εκδοχή του αλγορίθμου όμως δεν έχουμε την δυνατότητα χρήσης δεκαδικών αριθμών και ως εκ τούτου πιθανοτήτων. Επομένως, χρειάστηκε η μετατροπή της συνθήκης ούτως ώστε να κάνει χρήση μόνο ακέραιων τιμών. Για να το πετύχουμε αυτό, οι πιθανότητες εμφάνισης μετατράπηκαν σε αριθμούς εμφάνισης και σε όλα τα κλάσματα εφαρμόστηκε απαλοιφή παρονομαστών. Ο πίνακας modified περιέχει τις φορές εμφάνισης των στοιχείων 2d φορές και σε κάθε μετέπειτα επίπεδο διπλασιάζεται όπως και το d, για να συμβαδίζει με την κανονική εκδοχή του αλγορίθμου. Επομένως η πιο πάνω συνθήκη λαμβάνει την τελική της

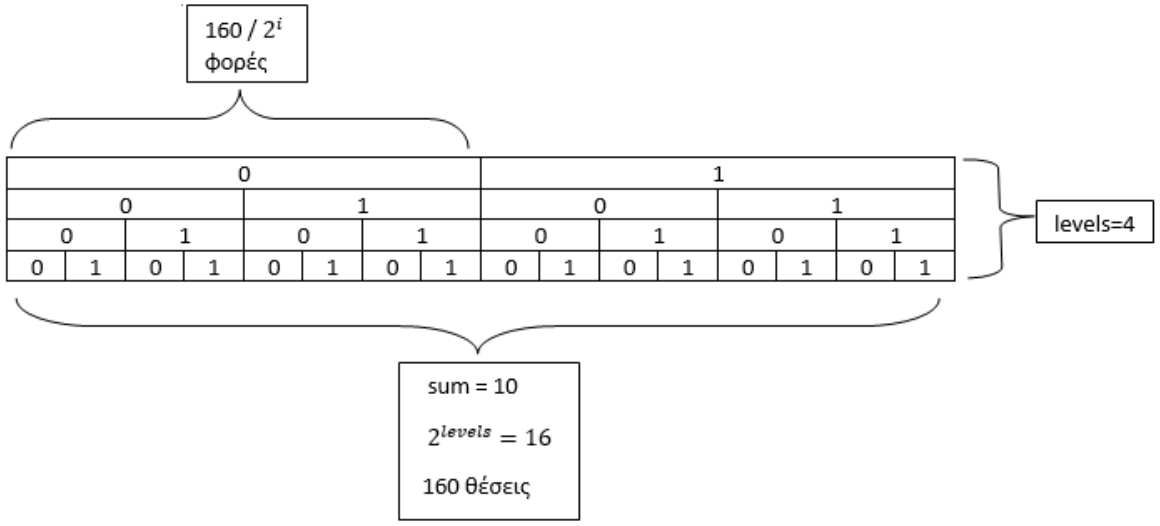
μορφή ως εξής: (modified[b] < sum && 2*modified[b] + modified[b+1] > 2*sum) || modified[b] >=sum.

Η υλοποίηση της συνάρτησης Leveling χρειάζεται πρώτα την δημιουργία δύο άλλων, βοηθητικών πινάκων, του πίνακα codes και του πίνακα letters. Για το λόγο ότι η γλώσσα υποστηρίζει μόνο ακέραιους αριθμούς, οι διαστάσεις και το περιεχόμενο των πινάκων χρειάζεται να τροποποιηθούν. Το μέγεθος πίνακα codes θα εξακολουθεί να είναι ίσο με τον μέγιστο αριθμό επιπέδων στη μια διάσταση αλλά κάθε διαφορετικό 0 ή 1 στη δεύτερη διάσταση του πίνακα θα εμφανίζεται τόσες φορές όσες και το μέγεθος της εισόδου, δηλαδή το άθροισμα των εμφανίσεων των στοιχείων. Επομένως, οι διαστάσεις του πίνακα codes είναι N x ($2^{logN} * N$). Το μέγεθος του πίνακα letters θα αυξηθεί με τον ίδιο τρόπο, κάθε εμφάνιση στοιχείου θα εμφανίζεται N φορές. Έτσι οι 2 πίνακες θα έχουν το ίδιο μέγεθος στη μια διάσταση και ο έλεγχος κατά πόσο ένα στοιχείο ανήκει στην εκάστοτε κατηγορία για κάθε επίπεδο γίνεται εφικτός. Οι νέοι πίνακες letters και codes φαίνονται σχηματικά στους Πίνακες 3.4 και 3.5 αντίστοιχα. Ως βελτιστοποίηση για τη μέθοδο αυτή, προτίνεται να χρησιμοποιηθεί το Ελάχιστο Κοινό Πολλαπλάσιο των μεγεθών των δύο πινάκων όπου θα εξοικονομήσει μνήμη. Στη συνέχεια είναι αναγκαίος ο υπολογισμός της μέσης κάθε στοιχείου στον πίνακα letters. Αυτό επιτυγχάνεται με την βοήθεια του πίνακα array ο οποίος περιέχει ταξινομημένη την είσοδο με βάση τις φορές εμφάνισης των στοιχείων.

Τέλος, ξεκινώντας από το επίπεδο που βρίσκεται ο μετρητής στον πίνακα reduced_code, εισάγουμε σε στοίβα το στοιχείο στην θέση του πίνακα codes που αντιστοιχεί με τη μέση των εμφανίσεων κάθε διαφορετικού στοιχείου στον πίνακα letters και προχωρούμε προς τα πάνω.



Εικόνα 3.4 Ο πίνακας letters για είσοδο REVERSIBLE



Πίνακας 3.5 Ο πίνακας codes στην Janus για είσοδο "REVERSIBLE"

3.1.4 Ανάλυση αλγορίθμου

Σε αυτή την ενότητα δίνεται χρονική και χωρική πολυπλοκότητα του αναστρέψιμου αλγορίθμου.

Αρχικά, συνάρτηση calculate_probabilities υπολογίζει τον αριθμό εμφάνισης κάθε χαρακτήρα στην είσοδο και στη συνέχεια γίνεται η ταξινόμηση μέσω της συνάρτησης bsortDecreasing. Η χρονική πολυπλοκότητα της calculate_probabilities είναι ίση με το πλήθος των στοιχείων στην είσοδο. Η ταξινόμηση είναι σταθερή καθώς το αλφάβητο που χρησιμοποιεί θεωρούμε ότι είναι το αγγλικό, επομένως μόνο 26 γράμματα. Επομένως, η χρονική πολυπλοκότητα της καθορίζεται από την συνάρτηση calculate_probabilities και είναι της τάξεως του Θ(N), όπου N το πλήθος των στοιχείων στην είσοδο. Η χωρική πολυπλοκότητα της πιο πάνω διαδικασίας είναι της τάξεως του Θ(N) αφού χρειάζεται ένας πίνακας που περιέχει την είσοδο για την διεκπεραίωση της διαδικασίας.

Ο υπολογισμός του αρχικού μήκους initial_length γίνεται στη συνάρτηση calculate_initialLength η οποία εκτελεί μια γραμμική διάσχιση στα δεδομένα με σκοπό τον υπολογισμό του ελάχιστου αριθμού των στοιχείων με άθροισμα μεγαλύτερο του μισού του πλήθους των στοιχείων στην είσοδο. Το ίδιο ισχύει και για την συνάρτηση match_corresponding που αντιστοιχεί τον αριθμό αυτό, με γραμμική διάσχιση, στο αντίστοιχο

initial_length. Η χρονική πολυπλοκότητα της πιο πάνω διαδικασίας είναι της τάξεως Θ(N) όπου N το πλήθος των διαφορετικών στοιχείων. Η χωρική πολυπλοκότητα είναι της τάξεως του μήκους της εισόδου Θ(N) καθώς οι πίνακες που χρειάζονται οι ανάλογοι του μήκους της εισόδου.

Η συνάρτηση calculate_ReducedCode υπολογίζει τα στοιχεία που αντιστοιχούν σε κάθε επίπεδο κωδικοποίησης και τα καταχωρεί στον πίνακα reduced_code. Η χρονική πολυπλοκότητα της πιο πάνω διαδικασίας είναι ανάλογη του αριθμού των επιπέδων και των στοιχείων (όποιο είναι μεγαλύτερο). Είναι τετριμμένο ότι ο αριθμός των επιπέδων καθώς και ο αριθμός των διαφορετικών στοιχείων είναι <=N, όπου N το πλήθος των στοιχείων στην είσοδο. Η συνάρτηση αυτή θα εκτελεστεί για όλα τα διαφορετικά στοιχεία και ο πίνακας modified έχει μήκος ίσο με N. Σε κάθε επανάληψη, οι τιμές του πίνακα modified διπλασιάζονται, επομένως η συνάρτηση αυτή είναι της τάξεως του Θ(N^2). Η χωρική πολυπλοκότητα της πιο πάνω διαδικασίας είναι της τάξεως του Θ(N) όπου το N το πλήθος των διαφορετικών στοιχείων και αυτό προκύπτει από το μήκος του πίνακα modified.

Όσο αφορά τη κωδικοποίηση που γίνεται στην υλοποίηση συνάρτηση Leveling: Η δημιουργία και αρχικοποίηση του πίνακα codes χρειάζεται χρόνο ανάλογο του αριθμού των επιπέδων * $2^{μέγιστο\ μέγεθος\ κωδικοποίησης}$ * αριθμό διαφορετικών στοιχείων. Ο αριθμός των επιπέδων είναι της τάξεως του μέγιστου μεγέθους κωδικοποίησης. Ο αριθμός των επιπέδων είναι της τάξεως του logN όπου N το πλήθος διαφορετικών στοιχείων. Άρα η διαδικασία είναι της τάξεως του Θ($\log N * (N * 2^{logN})$ άρα Θ(N^2logN). Η δημιουργία του πίνακα letters είναι της τάξεως του Θ(N * N) επομένως Θ(N^2). Η κωδικοποίηση leveling θα εκτελεστεί σε κάθε διαφορετικό στοιχείο και η ανάκτηση της κωδικοποίησης από κάτω προς τα πάνω στον πίνακα codes είναι της τάξεως του αριθμού των επιπέδων. Άρα η διαδικασία είναι της τάξεως του O(NlogN) + Θ(N^2 logN)) επομένως Θ(N^2 logN). Η χωρική πολυπλοκότητα είναι της τάξεως των διαστάσεων του πίνακα codes. Η χωρική και χρονική πολυπλοκότητα του αλγορίθμου είναι της τάξεως του Θ(NlogN), καθώς θεωρούμε δυνατή τη χρήση float μεταβλητών επομένως δεν έχουμε αύξηση της τάξεως του N στις διαστάσεις των πινάκων.

Η κωδικοποίηση meaning υλοποιείται στην συνάρτηση find_meaning και εκτελείται πάνω σε όλα τα διαφορετικά στοιχεία. Η χρονική πολυπλοκότητα της διαδικασίας είναι γραμμική, της

τάξεως των διαφορετικών στοιχείων άρα Θ(N). Η χωρική πολυπλοκότητα είναι της τάξεως των διαφορετικών στοιχείων, Θ(N) αφού χρειαζόμαστε 1 μεταβλητή για κάθε στοιχείο.

Η χρονική και η χωρική πολυπλοκότητα του αλγορίθμου είναι της τάξεως του Θ(NlogN) και αυτό προκύπτει από την αρχικοποίηση του πίνακα codes. Αυτό καθιστά τον αλγόριθμο πιστό καθώς διατηρεί τη χρονική πολυπλοκότητα και υγιή καθώς τα σκουπίδια που προκύπτουν οφείλωνται σε τιμές που προκύπτουν κατά τις μετρήσεις των βασικών δομών του αλγορίθμου (initial length, reduced code) και επομένως η μη αποθήκευση των σκουπιδιών επηρεάζει τη δημιουργία τους. Επομένως, τα σκουπίδια θεωρούνται ελάχιστα καθώς είναι αδύνατη η μειωση τους. Παρατηρούμε ότι η χρονική πολυπλοκότητα της υλοποίησης είναι Θ(N^2logN) ενώ η χωρική πολυπλοκότητα μπορεί να εκφραστεί σε συνάρτηση του πλήθους των στοιχείων στην είσοδο Θ(N^2logN). Αυτό καθιστά την υλοποίηση μας μη πιστή υλοποίηση του αυθεντικού αλγορίθμου. Επιπλέον, η υλοποίηση δεν είναι υγιής καθώς λόγω των περιορισμών της Janus το κόστος σε μνήμη δεν είναι ελάχιστο, κάτι που θα μπορούμε να έχουμε με περισσότερες δομές δεδομένων και τύπους float.

3.2 Ο αλγόριθμος Burrows-Wheeler

Ο αλγόριθμος Burrows-Wheeler αποτελεί αλγόριθμο μετασχηματισμού κειμένου ο οποίος εκτελείται πάνω σε ένα τμήμα δεδομένων και έχει ως αποτέλεσμα πιο αποδοτική κωδικοποίηση των χαρακτήρων. Προτάθηκε το 1994 από τους M. Burrows και D. J. Wheeler [5].

Ο αλγόριθμος χωρίζεται σε τέσσερα μέρη, τα οποία αποτελούνται από δύο ζεύγη αντίστροφων πράξεων. Το πρώτο ζεύγος πράξεων αποτελείται από τον μετασχηματισμό
και την αντίστροφη πράξη ενώ το δεύτερο ζεύγος πράξεων αποτελείται από τον αλγόριθμο Move-to-Front, την κωδικοποίηση με βάση κάποιο αλγόριθμο κωδικοποίησης μεταβλητού μήκους και την αντίστροφη τους πράξη.

Ο αλγόριθμος μετασχηματισμού C έχει ως εξής: Θεωρούμε ότι έχουμε την είσοδο S με μήκος N. Δημιουργούμε ένα πίνακα M με διαστάσεις N x N ο οποίος περιέχει στην πρώτη γραμμή την είσοδο. Στη συνέχεια, εκτελούμε επαναληπτικά για κάθε επόμενη γραμμή κυκλική μετάθεση προς τα αριστερά πάνω στην προηγούμενη γραμμή, δηλαδή για κάθε χαρακτήρα k

στην γραμμή i και θέση j, η γραμμή i+1 περιέχει τον χαρακτήρα k στη θέση j-1, με εξαίρεση ότι ο πρώτος χαρακτήρας μετατίθεται κυκλικά στην τελευταία θέση. Στη συνέχεια, εκτελείται λεξικογραφική ταξινόμηση στον πίνακα M και δημιουργείται ο πίνακα L ο οποίος περιέχει την τελευταία στήλη του ταξινομημένου πίνακα M'. Τέλος υπολογίζεται ο δείκτης I ο οποίος δείχνει την πρώτη γραμμή στην οποία υπάρχει η είσοδος με την μορφή που βρίσκεται στο S.

Ο αλγόριθμος αναίρεσης του μετασχηματισμού D έχει ως είσοδο την έξοδο του αλγόριθμου C, δηλαδή τον πίνακα L και τον δείκτη I. Δημιουργούμε τον πίνακα F ταξινομώντας τον πίνακα L. Παρατηρούμε ότι όλες οι στήλες του πίνακα M είναι μεταθέσεις της εισόδου S, επομένως και οι πίνακες L και F είναι μεταθέσεις της εισόδου καθώς και ο ένας του άλλου. Η ταξινόμηση του L έχει ως αποτέλεσμα ο F να είναι η πρώτη στήλη του πίνακα M. Έπειτα, εκτελούμε δεξία μετάθεση σε κάθε γραμμή του M και δημιουργείται ο πίνακας M'. Στη συνέχεια, με τη χρήση των πινάκων L και F δημιουργείται ο πίνακας T ο οποίος δείχνει την αντιστοιχία μεταξύ των δύο πινάκων και κατ' επέκταση την αντιστοιχία μεταξύ των πινάκων M' και M. Ο πίνακας T είναι τέτοιος ώστε κάθε δείκτης j σε γραμμή του M' αντιστοιχεί στη γραμμή T[j] του M. Τέλος, αφού γνωρίζουμε ότι κάθε γραμμή είναι και μια μετάθεση του S τότε είναι εύκολο να δείξουμε ότι κάθε L[i] προηγείται κυκλικά του F[i] στην είσοδο S. Ο δείκτης I ορίζεται στον αλγόριθμο C και εξ ορισμού ο τελευταίος χαρακτήρας του S είναι ο L[I]. Για να βρούμε τους υπόλοιπους κάνουμε χρήση του πίνακα Τα, όπου για κάθε i S[N-1-i] = L[T^i[I]] όπου
T^0 [x] = x και T^{i+1} [x] = T[T^i [x]]. Αυτό ολοκληρώνει την αναίρεση του αλγόριθμου μετασχηματισμού.

Ο αλγόριθμος Move-to-front coding αποτελεί συνέχεια αλγόριθμου C και έχει ως ανάστροφο του τον αλγόριθμο Move-to-front decoding. Αρχικά ο αλγόριθμος παίρνει ως είσοδο τον πίνακα L και τον δείκτη I που υπολογίστηκαν στον αλγόριθμο C. Αρχικοποιούμε την λίστα Y η οποία περιέχει το αλφάβητο της εισόδου. Στη συνέχεια, θέλουμε να δημιουργήσουμε τον πίνακα R που θα περιέχει ένα πρώτο στάδιο κωδικοποίησης. Για κάθε i θέτουμε τη θέση R[i] ως τον αριθμό των χαρακτήρων που προηγούνται του L[i] στον πίνακα Y και στη συνέχεια μετακινούμε τον χαρακτήρα L[i] στην πρώτη θέση του Y. Τέλος, εκτελούμε ένα αλγόριθμο κωδικοποίησης μεταβλητού μήκους πάνω στον πίνακα R και παίρνουμε ως έξοδο το αποτέλεσμα.

Ο ανάστροφος αλγόριθμος Move-to-front decoding εκτελείται για αναίρεση του πιο πάνω. Πρώτο βήμα είναι η αναίρεση του βήματος κωδικοποίησης με τον τρόπο που ορίζει ο αλγόριθμος που έχουμε χρησιμοποιήσει. Στη συνέχεια, αρχικοποιούμε ξανά τη λίστα Y όπως και στον προηγούμενο αλγόριθμο, όμως αυτή τη φορά οι πράξεις εκτελούνται διαφορετικά. Για κάθε i, ορίζουμε τη θέση L[i] ως τον χαρακτήρα Y[R[i]] και μετακινούμε τον χαρακτήρα στην πρώτη θέση.

Για επιτυχή εκτέλεση και αναίρεση του αλγορίθμου πρέπει οι αλγόριθμοι να εκτελεστούν με τη πιο κάτω σειρά:

1. Αλγόριθμος C
2. Move-to-front encoding
3. Move-to-front decoding
4. Αλγόριθμος D

3.2.2 Ο αναστρέψιμος αλγόριθμος

Για τη δημιουργία του αναστρέψιμου αλγόριθμο χρειαζόμαστε μόνο τα βήματα 1 και 2 από πιο πάνω. Οι αντίστροφες πράξεις που εκφράζονται στα βήματα 3 και 4 ενσωματώνονται στον αναστρέψιμο αλγόριθμο.

Το πρώτο βήμα του αναστρέψιμου αλγόριθμου είναι η δημιουργία του πίνακα M με την χρήση κυκλικών μεταθέσεων στη είσοδο. Η συνάρτηση createM δέχεται ως παραμέτρους τον πίνακα input που αντιπροσωπεύει την είσοδο και τον πίνακα M που θα αποθηκεύσει την έξοδο.

1. Αρχικοποίησε την πρώτη γραμμή του πίνακα με την είσοδο
2. local int i=1
3. from i=1 loop
    a. local int j = 0
    b. from j=0 loop
        i. M[i][j]+=M[i-1][j+1]
        ii. j+=1
    c. until j=size(input)-1
    d. M[i][j] +=M[i-1][0]

        e. i+=1
        f. delocal int j =size(input)-1
    4. until i=size(input)
    5. delocal int i=size(input)

Αρχικά δίνεται η είσοδος input στη πρώτη γραμμή του πίνακα M. Για κάθε επόμενη γραμμή εφαρμόζουμε κυκλικές μεταθέσεις προς τα αριστερά πάνω στην προηγούμενη γραμμή. Η συνάρτηση που υλοποιεί το πιο πάνω στη Janus είναι η initialize.

Στη συνέχεια εκτελείται αναστρέψιμη συνάρτηση ταξινόμησης όπως δίνεται στο [2] με τη διαφορά ότι για την επιτυχή λεξικογραφική ταξινόμηση πρέπει να γίνει χρήση αναστρέψιμης συνάρτησης σύγκρισης συμβολοσειρών. Η συνάρτηση string_compare φαίνεται πιο κάτω:

1. local int i = 0
2. from i=0
3. loop
    a. if (M[s][i] > M[f][i]) then
        i. result+=1
    b. fi M[s][i] > M[f][i]
    c. if (M[s][i] < M[f][i]) then
        i. result -= 1
    d. fi M[s][i] < M[f][i]
    e. i+=1
4. until result != 0
5. push(i,aux)
6. delocal int i = 0

Η πιο πάνω συνάρτηση εκτελεί ένα βρόγχο μέχρι να βρεθεί διαφορετικό στοιχείο μεταξύ δύο συμβολοσειρών. Αν η πρώτη συμβολοσειρά προηγείται της δεύτερης τότε το αποτέλεσμα result γίνεται 1 ενώ -1 αντίθετα. Για σκοπούς αναστρεψιμότητας η τιμή του μετρητή επαναλήψεων i πρέπει να αποθηκευτεί ως επιπλέον σκουπίδι γιατί δεν μπορούμε να γνωρίζουμε ή να υπολογίσουμε διαφορετικά αυτή τη τιμή. Η συνάρτηση lexicographical

αποτελεί υλοποίηση των πιο πάνω. Έπειτα, η δημιουργία του πίνακα L είναι τετριμμένη καθώς απλά παίρνουμε τη τελευταία στήλη του πίνακα M.

Στη συνέχεια πρέπει να υπολογίσουμε το αλφάβητο που εμπεριέχεται στην είσοδο, δηλαδή τον πίνακα Y. Αυτό γίνεται με τη βοήθεια ενός βοηθητικού πίνακα που αποθηκεύει τον αριθμό εμφανίσεων κάθε χαρακτήρα και στη συνέχεια βρίσκει ποιοι χαρακτήρες έχουν μη-μηδενικές εμφανίσεις.

1. Δημιουργία πίνακα array[] με τις εμφανίσεις των διαφορετικών στοιχείων, ταξινομημένο σε αύξουσα σειρά
2. local int count = 0
3. local int i= 0
4. from i=0 loop
    a. if (array[i] !=0) then
        1. different +=1
        2. Y[count]+=1
        3. count+=1
    b. fi array[i] !=0
    c. i+=1
5. until i=size(array)
6. delocal int i=size(array)
7. delocal int count = different

Για τον λόγο ότι τα στοιχεία της εισόδου εμφανίζονται σε αύξουσα σειρά στον πίνακα array, ο πίνακας Y που δημιουργείται είναι και αυτός ταξινομημένος σε αύξουσα σειρά. Ο ισχυρισμός στο βήμα 4.b είναι τετριμμένος αφού δεν έχουμε διακλάδωση else. Η συνάρτηση που το υλοποιεί ονομάζεται createY.

Τέλος, η δημιουργία του πίνακα R για τον αλγόριθμο Move-to-front coding ολοκληρώνει τον αναστρέψιμο αλγόριθμο.

1. local int i= 0
2. from i=0 loop

a. local int num=0
b. call countNum(num,Y,L[i])
c. R[i]+=num
d. local int j = num
e. from j = num loop
    - Y[j] ⇔ Y[j-1]
    - j-=1
f. until j=0
g. delocal int num = R[i]
h. i+=1

3. until i = size(input)
4. delocal int i=size(input)

Η πιο πάνω διαδικασία δημιουργεί τον τελικό πίνακα R ο οποίος είναι ο τελικός πίνακας που προκύπτει από τον μετασχηματισμό του αλγόριθμου Burrows-Wheeler. Η συνάρτηση countNum υπολογίζει τον αριθμό των στοιχείων που προηγούνται του εκάστοτε στοιχείου του πίνακα L (τελευταία στήλη ταξινομημένου πίνακα M) στον πίνακα Y. Στη συνέχεια ανταλλάζουμε τις θέσεις των στοιχείων ανά 2 μέχρι η εμφάνιση του στοιχείου στον πίνακα Y να βρίσκεται στην πρώτη θέση του πίνακα. Αυτό θα έχει ως αποτέλεσμα διαφορετικές εμφανίσεις του ίδιου στοιχείου να ανατεθούν σε διαφορετική κωδικοποίηση. Η συνάρτηση αυτή υλοποιείται στην createR. Επόμενο βήμα είναι η επιλογή ενός αποδοτικού αλγόριθμου κωδικοποίησης μεταβλητού μήκους και η εφαρμογή του στον πίνακα R.

Με ανάποδη εκτέλεση των πιο πάνω βημάτων για τα δύο μέρη του αλγορίθμου παίρνουμε το ίδιο αποτέλεσμα με την εκτέλεση των τεσσάρων βημάτων του αυθεντικού αλγόριθμου. Επομένως οι αντίθετες πράξεις συγχωνεύονται στην αναστρέψιμη εκδοχή.

3.2.3 Η υλοποίηση στη Janus

Η υλοποίηση στη γλώσσα Janus για τον συγκεκριμένο αλγόριθμο δεν χρειάστηκε αλλαγές για επίτευξη της αναστρεψιμότητας καθώς ο αλγόριθμος ορίζεται ως αναστρέψιμος και μπορεί να υλοποιηθεί μόνο με την χρήση πινάκων οι οποίοι είναι διαθέσιμοι στη Janus.

3.2.4 Ανάλυση αλγόριθμου

Η συνάρτηση initializeM εκτελεί αρχικοποίηση πάνω σε ένα πίνακα διαστάσεων N*N όπου το N είναι ο αριθμός των στοιχείων στην είσοδο. Επομένως η χρονική πολυπλοκότητα της διαδικασίας είναι της τάξεως $\Theta(N^2)$, όπου N είναι το πλήθος των στοιχείων στην είσοδο. Η χωρική πολυπλοκότητα είναι της τάξεως του πίνακα M, άρα $\Theta(N^2)$.

Η συνάρτηση lexicographical εκτελεί αναστρέψιμο αλγόριθμο ταξινόμησης. Σε κάθε σύγκριση καλείται η συνάρτηση string_compare η οποία εκτελεί σύγκριση ανά στοιχείο στις 2 γραμματοσειρές. Η χρονική πολυπλοκότητα της string_compare είναι της τάξεως του O(N). Επομένως, χρονική πολυπλοκότητα της συνάρτησης είναι $O(N^2 * logN)$. Όσο αφορά την χωρική πολυπλοκότητα του αλγορίθμου είναι της τάξεως $O(N^2)$ διοτι χρειάζεται η αποθήκευση επιπλέον N πινάκων μήκους N ως βοηθητικούς για την ταξινόμηση.

Η χρονική πολυπλοκότητα για την συνάρτηση calculateR εκτελείται για κάθε στοιχείο και το αλφάβητο που περιέχεται στην είσοδο μήκους N δεν μπορεί να ξεπεράσει τα N στοιχεία. Επομένως έχουμε N επαναλήψεις όπου κάθε επανάληψη χρειάζεται το πολύ N συγκρίσεις. Άρα η χρονική πολυπλοκότητα είναι της τάξεως του $O(N^2)$. Η χωρική πολυπλοκότητα είναι της τάξεως του Θ(N), αφού θα γίνει μετασχηματισμός κάθε στοιχείου στην είσοδο και θα δημιουργηθεί ο πίνακας R.

3.3 Ο αλγόριθμος Huffman

Ο αλγόριθμος κωδικοποίησης Huffman αποτελεί πλέον τον πιο διαδεδομένο αλγόριθμο κωδικοποίησης μεταβλητού μήκους. Ο αλγόριθμος προτάθηκε το 1951 από τον David Huffman κατά τη διάρκεια των σπουδών του [7].

Ο αλγόριθμος χρησιμοποιεί συγκεκριμένο τρόπο αναπαράστασης της κωδικοποίησης όπου δίνει μια προθεματική κωδικοποίηση. Αυτό σημαίνει ότι η κωδικοποίηση που αντιπροσωπεύει ένα συγκεκριμένο σύμβολο, δεν θα είναι ποτέ πρόθεμα σε καμία κωδικοποίηση άλλου συμβόλου.

3.3.1 Ο αυθεντικός αλγόριθμος

Σε αυτή την ενότητα θα δοθεί ο αυθεντικός αλγόριθμος για την κωδικοποίηση μεταβλητού μήκους Huffman, όπως ορίστηκε από τον David Huffman.

Για την κωδικοποίηση του αλγορίθμου χρειαζόμαστε τις πιθανότητες εμφάνισης του κάθε αντικειμένου. Αυτό γίνεται στην αρχή, πριν από την δημιουργία του δέντρου. Στη συνέχεια είναι αναγκαία η δημιουργία του δέντρου. Για την δημιουργία του δέντρου θα χρειαστούμε μια ουρά προτεραιότητας όπου η προτεραιότητα ορίζεται ως η ελάχιστη πιθανότητα εμφάνισης. Για κάθε σύμβολο δημιουργούμε νέο κόμβο και τον εισάγουμε στην ουρά προτεραιότητας. Αυτοί οι κόμβοι αντιπροσωπεύουν τα φύλλα του δέντρου. Έπειτα όσο υπάρχουν περισσότεροι από ένας κόμβοι στην ουρά προτεραιότητας γίνεται εξαγωγή των δύο κόμβων με την μεγαλύτερη προτεραιότητα, δηλαδή την χαμηλότερη πιθανότητα εμφάνισης και δημιουργείται νέος ενδιάμεσος κόμβος με πιθανότητα εμφάνισης ίση με το άθροισμα των δύο κόμβων που πλέον είναι τα παιδία του. Όταν η ουρά προτεραιότητας έχει ένα κόμβο τότε αυτός ο κόμβος γίνεται η ρίζα και ολοκληρώνεται η δημιουργία του δέντρου. Για ανάθεση κωδικοποίησης σε χαρακτήρες απλά εκτελείται κατά βάθος αναζήτηση στο δέντρο και για κάθε ακμή που δείχνει προς αριστερό παιδί έχουμε κωδικοποίηση 0 ενώ για δεξί παιδί 1. Όταν φτάσουμε σε φύλλο ολοκληρώνεται η κωδικοποίηση του χαρακτήρα.

Όσο αφορά την αποκωδικοποίηση χρειάζεται πάλι ανακατασκευή του δέντρου χωρίς την γνώση των πιθανοτήτων εμφάνισης. Μια λύση είναι να στέλνουμε το δέντρο μαζί με τα δεδομένα όπου το δυφίο 0 να αντιστοιχεί σε ενδιάμεσο κόμβο ή τη ρίζα και το δυφίο 1 να αντιστοιχεί σε φύλλο. Αυτό όμως για αλφάβητο 8 δυφίων αντιστοιχεί σε επιπλέον κόστος 2-320 bytes.

3.3.2 Ο αναστρέψιμος αλγόριθμος

Πιο κάτω θα δοθεί λεπτομερώς μια αναστρέψιμη εκδοχή του αλγορίθμου. Για την υλοποίηση του πιο αλγόριθμου απαραίτητη προϋπόθεση είναι η ύπαρξη των πιο κάτω δομών δεδομένων :

1. Δομή η οποία αντιστοιχεί τον αριθμό εμφανίσεων ενός στοιχείου με το στοιχείο αυτό και με τον αριθμό του κόμβου. Αντιστοιχεί στα φύλλα του Huffman tree: pair1(# occurrences ,node, node#)
2. Δομή η οποία αντιστοιχεί τον αριθμό εμφανίσεων 2 κόμβων με τα 2 παιδία και τον αριθμό του κόμβου. Αντιστοιχεί στους ενδιαμέσους κόμβους και την ρίζα του Huffman tree: pair2(# occurrences, left , right ,node #)
3. Ουρά προτεραιότητας όπου η πιο ψηλή προτεραιότητα δίνεται στα στοιχεία με το χαμηλότερο αριθμό εμφανίσεων: ascending order queue (mpqueue)

Πιο κάτω δίνεται ο ψευδοκώδικας για υπολογισμό των φορών εμφάνισης κάθε συμβόλου. Η πιθανότητα εμφάνισης αντιστοιχεί στον αριθμό που υπολογίζεται διά το μήκος της εισόδου. Η συνάρτηση calculateOccurrences δέχεται ως παράμετρο την είσοδο input και ένα βοηθητικό πίνακα array για υπολογισμό των εμφανίσεων για κάθε σύμβολο.

1. local int i=0
2. from i=0 loop
    a. array[input[i]-1]+=1
    b. i+=1
3. until i=size(input)
4. delocal int i= size(input)

Η πιο κάνω διαδικασία δημιουργεί τα φύλλα του Huffman tree με την χρήση της δομής pair1 και τα εισάγει στην ουρά. Θα δημιουργηθούν τόσα φύλλα όσα και οι διαφορετικοί χαρακτήρες στην είσοδο. Η μεταβλητή counter είναι παράμετρος της συνάρτησης createTree

1. local int i=0
2. from i=0 loop
    a. if (array[i] !=0) then
        i. Create pair1(array[i],i+1,counter)
        ii. Enqueue pair1 in mpqueue
        iii. counter+=1
    b. fi array[i]!=0
    c. i+=1

3. until i=size(array)
4. delocal int i=size(array)

Η πιο πάνω διαδικασία δημιουργεί τα φύλλα του Huffman tree με την χρήση της δομής pair1 και τα εισάγει στην ουρά. Θα δημιουργηθούν τόσα φύλλα όσα και οι διαφορετικοί χαρακτήρες στην είσοδο.

Εδώ γίνεται η δημιουργία των εσωτερικών κόμβων του Huffman tree. Για κάθε εσωτερικό κόμβο, εξάγουμε τα 2 πρώτα στοιχεία της ουράς προτεραιότητας και δημιουργούμε αντικείμενο τύπου pair2 με αριθμό εμφανίσεων το άθροισμα των 2 , δείκτες προς τους χαρακτήρες που αντιπροσωπεύει και τον αριθμό του κόμβου. Η πιο κάτω συνάρτηση ονομάζεται createInnerNodes και παίρνει ως παραμέτρους τον αριθμό των διαφορετικών στοιχείων different, την ουρά προτεραιότητας mpqueue και τον μετρητή counter ο οποίος αριθμεί τους κόμβους.

1. local int i= different -1
2. from i=different -1 loop
    a. Dequeue 2 pairs p1 , p2 from mpqueue
    b. Create pair2(p1.occurrences + p2.occurrences,p1.node,p2.node,counter)
    c. Enqueue pair2 in mpqueue
    d. counter+=1
    e. i-=1
3. until i<0
4. delocal int i=-1

Για την αναπαράσταση του δέντρου με αναστρέψιμη λογική, θα γίνει χρήση πίνακα γειτνίασης. Ο πίνακας γειτνίασης έχει την εξής μορφή : Για κάθε κόμβο p στη ουρά, αν το p είναι φύλλο τότε αποθηκεύεται για σκοπούς αναστρεψιμότητας, αν το p είναι ενδιάμεσος κόμβος τότε το βάρος της ακμής μεταξύ p και p.left είναι =0 και μεταξύ p και p.right =1. Οι τιμές των βαρών θα χρησιμοποιηθούν για την κωδικοποίηση. Η συνάρτηση convertToMatrix αποθηκεύει τον πίνακα γειτνίασης στη δομή HuffmanTree και παίρνει ως παραμέτρους τη βοηθητική στοίβα

stack που αποθηκεύει τους κόμβους για σκοπούς αναστρεψιμότητας, την ουρά προεραιότητας mpqueue και τον αριθμό των διαφορετικών στοιχείων different.

1. local int i=0
2. from i=0 loop
    a. Dequeue p from mpqueue
    b. if p.type=pair1 then
        i. push(p,stack)
    c. else
        i. left=p.left
        ii. right=p.right
        iii. HuffmanTree[p.counter][left.counter]+=1
        iv. HuffmanTree[p.counter][right.counter]+=2
        v. enqueue left and right in mpqueue
    d. fi p.type=pair1
    e. i+=1
3. until size(stack)=different
4. times+=i
5. delocal int i=different+different-1

Η πιο κάτω διαδικασία μετατρέπει ένα πίνακα γειτνίασης 2 διαστάσεων σε μια λίστα γειτνίασης η οποία αντιπροσωπεύεται με 2 πίνακες. Ο πίνακας G ο οποίος στην πρώτη διάσταση λειτουργεί ως δείκτης προς κάθε στοιχείο ενώ στη δεύτερη διάσταση δείχνει το βάρος κάθε ακμής και ο πίνακας nodes του οποίου η πρώτη στήλη δείχνει την αρχή κάθε λίστας γειτνίασης, η δεύτερη στήλη την χρονική στιγμή σε μονάδες χρόνου την οποία επισκεφτήκαμε τον κόμβο κατά την κάθοδο του αλγόριθμου διάσχισης και η τρίτη στήλη την χρονική στιγμή σε μονάδες χρόνου την οποία επισκεφτήκαμε τον κόμβο κατά την άνοδο/επιστροφή προς την ρίζα του αλγόριθμου διάσχισης. Η συνάρτηση createAdjacencyList παίρνει ως παραμέτρους τον πίνακα HuffmanTree από τη συνάρτηση convertToMatrix και τους πίνακες G και nodes που λειτουργούν και ως η έξοδος της.

1. Έστω HuffmanTree
2. Έστω G[][] , nodes[][]

3. int count =0
4. local int i=0
5. local int j=0
6. from i=0 loop
    a. nodes[i][0]+=count
    b. from i=0 loop
        i. if matrix[i][j] != -1 then
            1. G[count][0]+=j+1
            2. G[count][1]+=matrix[i][j]
            3. count+=1
        ii. fi matrix[i][j] !=-1
        iii. j+=1
    c. until j=size(matrix)
    d. j-=size(matrix)
    e. i+=1
7. until i=size(matrix)
8. delocal int j=0
9. delocal int i=size(matrix)

Η συνάρτηση DSF καλεί τη αναδρομική συνάρτηση DSFvisit πάνω σε όλες τις λίστες γειτνίασης οι οποίες δεν έχουν επισκεφτεί ακόμα, άρα έχουν αρχικό χρόνο 0. Στον συγκεκριμένο αλγόριθμο πρόκειται για δέντρο το οποίο από την ρίζα μπορούμε να πάμε σε οποιοδήποτε κόμβο, επομένως μόνο 1 εκτέλεση του βρόγχου είναι αρκετή. Οι 2 τελευταίες στήλες του πίνακα nodes αντιπροσωπεύουν την ώρα στην οποία «ανοίγουμε» μια λίστα και την ώρα που την «κλείνουμε» για σκοπούς αναστρεψιμότητας. Κατά την ορθή κλήση της διαδικασίας ξεκινούμε από την χρονική στιγμή 0 και καταλήγουμε στην χρονική στιγμή 2*P (όπου P το μέγεθος των κόμβων) ενώ κατά την ανάστροφη κλήση ξεκινούμε από τον κόμβο με κλείσιμο την χρονική στιγμή 2*P και καταλήγουμε στον κόμβο με αρχικό χρόνο 0 [10]. Στο βήμα 8 της DFSvisit, κατά την διάσχιση του γράφου εισάγουμε το βάρος της ακμής σε μια στοίβα. Όταν φτάσουμε σε φύλλο ουσιαστικά έχουμε ολοκληρώσει την ανάγνωση των δυφίων που αποτελούν την κωδικοποίηση του αντίστοιχου χαρακτήρα. Επομένως γίνεται εξαγωγή και ένωση τους ως 1 αριθμό και εισαγωγή στον πίνακα encodings.

DFS(int G[][],int nodes[][], stack codes, int encodings[][])

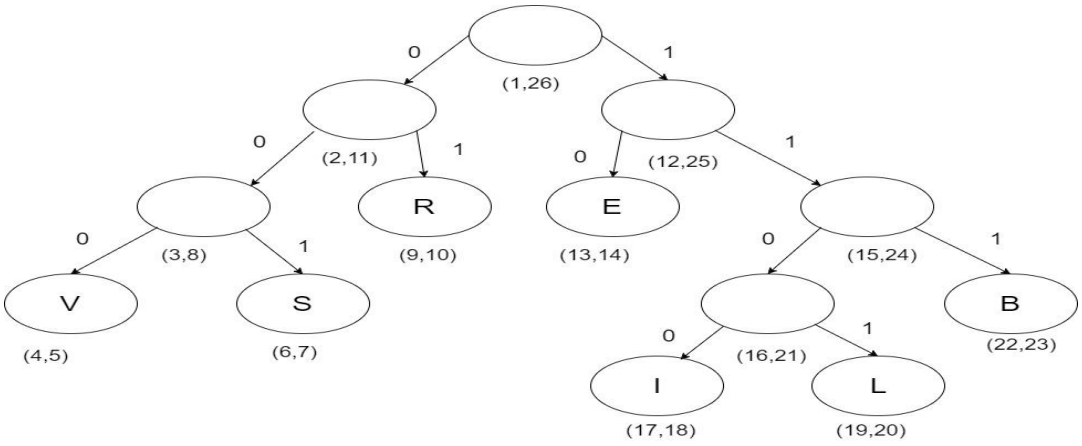
1. local int time=0
2. local int i=0
3. from i=0 loop
    a. if nodes[i][1]=0 then
        i. call DFSvisit(G,nodes,i,time)
    b. fi nodes[i][2] = time
    c. i+=1
4. until i=size(nodes)
5. delocal int time=size(nodes)*2
6. delocal int i=size(nodes)

DFSvisit(int G[][],int nodes[][],int u,int time, stack codes, int encodings[][])

1. time+=1
2. nodes[u][1]+=time
3. local int end=0
4. if u=size(nodes) then
    a. end+=size(G)
5. else
    a. end+=nodes[u+1][0]
6. fi u=size(nodes)-1
7. local int r=nodes[u][0]
8. from r=nodes[u][0] then
    a. if nodes[G[r][0]-1][1]=0 then
        i. push(G[r][0] , codes)
        ii. call DFSvisit(G,nodes,G[r][0]-1,time)
    b. fi nodes[G[r][0]-1][2]=time
    c. r+=1
9. until r=end
10. delocal int r= end
11. if nodes[u].isleaf then
    a. local int x =0
    b. pop(x,codes) //ως 1 αριθμό

c. Encodings[k] +=x
d. x-=Encodings[k]
e. k+=1
f. delocal int x=0

12. fi node[u].isleaf
13. time+=1
14. nodes[u][2]+=time
15. if u=size(nodes)-1 then
    a. end-=size(G)
16. else
    a. end-=nodes[u+1][0]
17. fi u=size(nodes)-1
18. delocal int end =0



Εικόνα 3.4 Παράδειγμα Huffman tree με είσοδο "REVERSIBLE". Τα ζεύγη (x,y) αντιστοιχούν στον αρχικό και τελικό χρόνο.

Πιο κάτω δίνεται ο αλγόριθμος για αντιστοίχιση των κωδικοποιήσεων με το κάθε γράμμα.

1. Ταξινόμησε τον πίνακα encodings ανά γραμμή με βάση το μέγεθος σε ψηφία, σε φθίνουσα σειρά
2. Εξαγωγή κάθε στοιχείου της στοίβας stack και εισαγωγή του σε mpqueue

3. local int i=0
4. from i=0 loop
    a. Dequeue p from mpqueue
    b. encodings[i][1]+=p.node
    c. i+=1
5. until i=size(encodings)
6. delocal int i=size(encodings)
7. Χρήση διπλού βρόγχου για μετατροπή Input σε bits (encodings)

Η συνάρτηση matchEncodings παίρνει ως παραμέτρους την στοίβα stack που περιέχει τους χαρακτήρες εισόδου από προηγούμενο βήμα και τον πίνακα encodings με τις κωδικοποιήσεις. Επομένως γίνεται εξαγωγή τους και εισαγωγή τους στην ταξινομημένη ουρά. Έτσι, στην πρώτη θέση της ουράς θα βρίσκεται ο χαρακτήρας με τις λιγότερες εμφανίσεις και στην πρώτη θέση του πίνακα Encodings θα βρίσκεται η κωδικοποίηση με το μεγαλύτερο μήκος. Άρα έχουμε την αντιστοιχία μεταξύ των 2 ως το αποτέλεσμα του αλγορίθμου.

3.3.3 Η υλοποίηση στη Janus

Η υλοποίηση στη γλώσσα Janus προϋποθέτει την ύπαρξη των δομών που αναφέρθηκαν στο 3.3.2. Ο πιο πάνω αναστρέψιμος αλγόριθμος χρειάζεται τις δομές αυτές για να μπορέσει να υλοποιηθεί με επιτυχία. Η γλώσσα Janus όμως ακόμη δεν υποστηρίζει πιο σύνθετες δομές πέραν της στοίβας και πινάκων, επομένως δεν μπορούμε να υλοποιήσουμε τον πιο πάνω αλγόριθμο ακόμα. Όταν και εφόσον η γλώσσα αναπτυχθεί για να υποστηρίξει τις πιο πάνω δομές η μετατροπή του αλγορίθμου είναι τετριμμένη καθώς έχει δοθεί με αρκετή λεπτομέρεια (assertions, local-delocal) για να μπορεί να μεταφερθεί εύκολα σε κώδικα Janus.

3.3.4 Ανάλυση αλγορίθμου

Η χρονική πολυπλοκότητα της συνάρτηση calculateOccurrences είναι της τάξεως του Θ(N) όπου N το μήκος της εισόδου. Αυτό συμβαίνει γιατί χρειάζεται ο υπολογισμός τις πιθανότητας εμφάνισης κάθε στοιχείου στην είσοδο. Η χωρική πολυπλοκότητα της συνάρτησης είναι σταθερή καθώς ο πίνακας που κρατά τις εμφανίσεις όλων των στοιχείων έχει μήκος ίσο με το

αλφάβητο που δηλώνουμε. Αν θεωρήσουμε το αγγλικό αλφάβητο με 26 στοιχεία τότε έχουμε χωρική πολυπλοκότητα της τάξης του Θ(1).

Η συνάρτηση createTree εκτελεί προσπέλαση όλου του πίνακα array. Αυτό γίνεται σε σταθερό χρόνο επομένως η χρονική πολυπλοκότητα της διαδικασίας έγκειται στην εισαγωγή όλων των διαφορετικών στοιχείων στην ουρά προτεραιότητας. Η εισαγωγή στην ουρά προτεραιότητας χρειάζεται χρόνο τάξεως O(logN), όπου N ο αριθμός των διαφορετικών στοιχείων, επομένως η εισαγωγή N στοιχείων στην ουρά προτεραιότητας θέλει χρόνο τάξης O(N*logN). Όσο αφορά τη χωρική πολυπλοκότητα, χρειαζόμαστε χώρο μνήμης για την ουρά, η οποία θα λάβει αντικείμενα ίσα με τον αριθμό των φύλλων. Ο αριθμός των φύλλων είναι N, όπου N ο αριθμός των διαφορετικών στοιχείων. Επομένως, η χωρική πολυπλοκότητα είναι της τάξεως Θ(N).

Στη συνέχεια έχουμε τη συνάρτηση createInnerNodes. Αφού έχουμε N φύλλα, τότε θα έχουμε N-1 εσωτερικούς κόμβους. Ο συνολικός αριθμός των κόμβων που προκύπτουν είναι 2N-1, επομένως η χρονική πολυπλοκότητα είναι ίση με O($N * logN$) και αυτό προκύπτει από την εισαγωγή των εσωτερικών κόμβων τάξεως Θ(N) στην ουρά προτεραιότητας. Η χωρική πολυπλοκότητα της συνάρτησης έγκειται στον συνολικό αριθμό κόμβων στην ουρά προτεραιότητας. Αφού λοιπόν ο συνολικός αριθμός κόμβων είναι 2N-1 τότε και η χωρική πολυπλοκότητα είναι της τάξεως Θ(N).

Ακολουθεί η συνάρτηση convertToMatrix. Η χρονική πολυπλοκότητα της διαδικασίας είναι O(N*logN) λόγω της εξαγωγής από την ταξινομημένη ουρά. Αυτό προκύπτει γιατί γίνεται εξαγωγή όλων των στοιχείων από την ουρά για να δημιουργηθεί ο πίνακας γειτνίασης που να αντιπροσωπεύει το δέντρο. Ο πίνακας γειτνίασης έχει διαστάσεις (2*N-1)*(2*N-1) αφού πρόκειται για τετραγωνικό πίνακα. Επομένως, η χωρική πολυπλοκότητα είναι της τάξεως των διαστάσεων του πίνακα δηλαδή Θ(N^2).

Για την δημιουργία της λίστας γειτνίασης που θα χρησιμοποιηθεί για αναστρέψιμη κατά βάθος αναζήτηση είναι απαραίτητη η προσπέλαση ολόκληρου του πίνακα. Επομένως η χρονική πολυπλοκότητα της συνάρτησης είναι της τάξεως των διαστάσεων του πίνακα, άρα της τάξεως Θ(N^2). Η χωρική πολυπλοκότητα είναι της τάξεως Θ(N) καθώς η λίστα γειτνίασης αποθηκεύει

πληροφορίες για κάθε κόμβο, επομένως για κόμβους της τάξης του N η χωρική πολυπλοκότητα που προκύπτει είναι της τάξης του Θ(N).

Για την ανάλυση της συνάρτησης DFS πρέπει πρώτα να δούμε τη πολυπλοκότητα της DFSvisit. Στην DFSvisit προχωρούμε αναδρομικά προς όλους τους γειτονικούς κόμβους από τον τρέχων κόμβο. Αφού κάθε κόμβο θα τον επισκεφτούμε μόνο μια φορά τότε η εκτέλεση του βρόγχου αυτού είναι της τάξεως του αριθμού των ακμών που υπάρχουν. Στην DFS ο βρόγχος εκτελείται για κάθε στοιχείο, όμως στη δική μας περίπτωση δεν υπάρχει περιοχή του δέντρου μη συνδεδεμένη με το υπόλοιπο, αρά ο συγκεκριμένος βρόγχος θα εκτελεστεί μόνο μια φορά. Επομένως, η χρονική πολυπλοκότητα είναι της τάξεως O(E), όπου E ο αριθμός των ακμών, όπως και στην μη αναστρέψιμη υλοποίηση. Η χωρική πολυπλοκότητα έγκειται στη δημιουργία του πίνακα encodings. Ο πίνακας encodings αποθηκεύει τις κωδικοποιήσεις των N στοιχείων. Το δέντρο που κτίζεται για σκοπό του αλγόριθμου Huffman έχει ύψος της τάξεως του O(logN) και για να βρούμε τη κωδικοποίηση κάθε στοιχείου διασχίζουμε κατά βάθος μέχρι να φτάσουμε σε φύλλο. Επομένως μια κωδικοποίηση δεν μπορεί να έχει μήκος μεγαλύτερο του O(logN). Αφού έχουμε N στοιχεία για κωδικοποίηση τότε οι διαστάσεις του πίνακα encodings είναι O(N*logN), επομένως και η χωρική πολυπλοκότητα.

Τέλος πρέπει να αντιστοιχίσουμε τα στοιχεία της εισόδου με τις κωδικοποίησης του πίνακα encodings. Αυτό γίνεται στη συνάρτηση matchEncodings και είναι απαραίτητη η ταξινόμηση των στοιχείων των δύο πινάκων (input και encodings), επομένως αυτή η διαδικασία αντιστοιχεί σε O(N*logN). Στη συνέχεια για την μετατροπή ολόκληρου του κειμένου σε κωδικοποιημένο κείμενο χρειάζεται η αντικατάσταση logN bits για κάθε N στοιχείο της εισόδου. Επομένως η χρονική πολυπλοκότητα της συνάρτησης είναι O(N*logN). Η χωρική πολυπλοκότητα της συνάρτησης έγκειται σε προηγούμενα βήματα καθώς δεν δημιουργεί νέα δομή για αποθήκευση, χρησιμοποιεί και τροποποιεί ήδη υπάρχουσες δομές.

Η ολική χρονική πολυπλοκότητα του αλγορίθμου είναι O(N*logN) όπως και στην αυθεντική εκδοχή του αλγορίθμου. Η χωρική πολυπλοκότητα της συνάρτησης είναι $\Theta(N^2)$ και αυτό προκύπτει από τον πίνακα γειτνίασης. Τα πιο πάνω καθιστούν τον αλγόριθμο πιστή υλοποίηση του αυθεντικού αλγορίθμου.

Ας δούμε πιο ενδελεχώς την χωρική πολυπλοκότητα. Η χωρική πολυπλοκότητα θα μπορούσε να μειωθεί στα ίδια επίπεδα με τη μη αναστρέψιμη του αλγορίθμου αν και μόνο αν δεν γινόταν χρήση του πίνακα γειτνίασης. Ο πίνακας γειτνίασης όμως χρησιμοποιείται για να γίνει δυνατή η υλοποίηση της αναστρέψιμης αναζήτησης κατά βάθος. Επομένως είναι αδύνατη η μείωση της μνήμης όσο κάνουμε χρήση του πίνακα αυτού. Άρα ο αναστρέψιμος αλγόριθμος θεωρείται υγιής υλοποίηση του αυθεντικού μέχρι να υλοποιηθεί μια καλύτερη λύση αναστρέψιμης κατά βάθους αναζήτησης από πλευράς μνήμης.

Κεφάλαιο 4

Αλγόριθμοι Κρυπτογράφησης

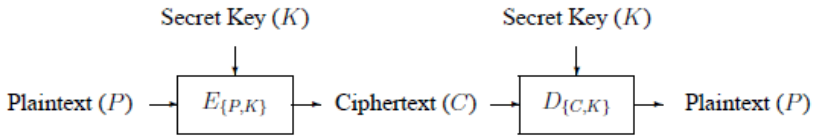
Το Κεφάλαιο αυτό καταπιάνεται με το πρόβλημα της κρυπτογραφίας. Η κρυπτογραφία είναι η τεχνική στην οποία ένα κείμενο τροποποιείται με συγκεκριμένο τρόπο τέτοιο ώστε το κείμενο να παρουσιάζεται αλλοιωμένο σε βαθμό όπου να μην μπορεί να διαβαστεί στην παρούσα μορφή. Χρησιμοποιείται για σκοπούς ασφάλειας όπου δύο εμπλεκόμενα μέρη θέλουν να επικοινωνήσουν μεταξύ τους. Ας θεωρήσουμε ότι ο Α είναι ο αποστολέας και ο Β είναι ο παραλήπτης, τότε ο Α θα κρυπτογραφήσει το κείμενο που θέλει να στείλει βάση κάποιου αλγορίθμου και θα το στείλει στον Β στη νέα του μορφή. Στη συνέχεια ο Β πρέπει να εφαρμόσει τον αντίστοιχο αλγόριθμο αποκρυπτογράφησης για να μπορέσει να διαβάσει το αυθεντικό κείμενο.

Είναι εύκολο κάποιος να παρατηρήσει ότι οι δύο αυτές πράξεις συμπληρώνουν και ταυτόχρονα αναιρούν η μια την άλλη. Δηλαδή είναι προφανές ότι αυτές οι δύο πράξεις, η κρυπτογράφηση και η αποκρυπτογράφηση έχουν έμφυτη την έννοια της αναστρεψιμότητας. Η επιτυχής μετατροπή των αλγορίθμων κρυπτογράφησης (με τους αντίστοιχους αλγόριθμους αποκρυπτογράφησης) σε αναστρέψιμους θα σημάνει μια νέα εποχή για την κρυπτογραφία. Αυτό συμβαίνει γιατί οι δύο αλγόριθμοι θα συγχωνευθούν σε ένα αλγόριθμο και έτσι τα εμπλεκόμενα μέρη θα χρειάζεται να έχουν μόνο ένα αλγόριθμο για επιτυχή επικοινωνία με

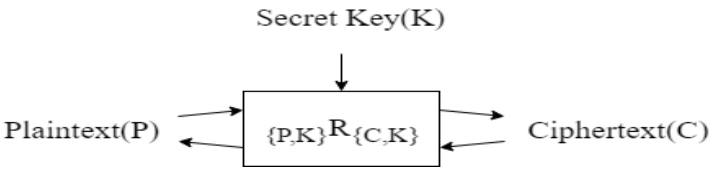
χρήση κρυπτογράφησης και οι πράξεις της κρυπτογράφησης και της αποκρυπτογράφησης θα είναι το εμπρόσθιο πέρασμα και το ανάστροφο πέρασμα του ίδιου αναστρέψιμου αλγορίθμου αντίστοιχα. Σε αυτή την παρατήρηση βρίσκεται το κίνητρο για την μελέτη και ανάπτυξη αναστρέψιμων αλγορίθμων κρυπτογράφησης στα πλαίσια της διπλωματικής εργασίας.

4.1 Data Encryption Standard (DES)

Ο αλγόριθμος DES είναι ένας από τους πιο διαδεδομένους αλγόριθμους κρυπτογραφίας. Είναι ένας συμμετρικός αλγόριθμος κρυπτογράφησης τμημάτων 64bits, δηλαδή έχει κρυφό κλειδί και κρυπτογραφεί 64bits δεδομένων κάθε φορά.

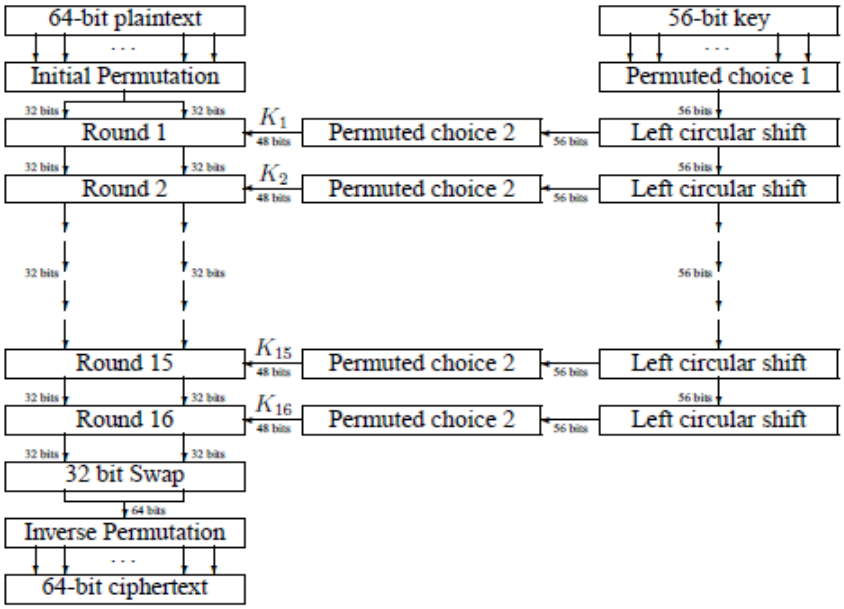


Σχήμα 4.1. Μη αναστρέψιμη υλοποίηση



Σχήμα 4.2. Αναστρέψιμη υλοποίηση

Η γενική ιδέα του αλγορίθμου είναι βασισμένη στο δίκτυο Feistelτο οποίο αποτελείται από ένα αριθμό γύρων (επαναλήψεων) όπου κάθε γύρος αποτελείται από μεταθέσεις δυφίων, μη γραμμικές αντικαταστάσεις μέσω S-boxes και πράξεις XOR. Παίρνει ως είσοδο το κείμενο προς κρυπτογράφηση και το μυστικό κλειδί [9]. Η γενική δομή του αλγορίθμου φαίνεται στο σχήμα 4.3.



Σχήμα 4.3 FeistelNetwork

Ένα δίκτυο Feistelαποτελείται από ένα αριθμό επαναλήψεων n, μια συνάρτηση f η οποία εκτελείται σε κάθε επανάληψη και τόσα κλειδιά όσα και ο αριθμός των επαναλήψεων K_i για κάθε I τέτοιο ώστε 0<=i<n. Τότε η βασική διαδικασία έχει ως εξής: Αρχικά μοιράζουμε το κείμενο προς κρυπτογράφηση σε L_i και R_i για την εκάστοτε επανάληψη όπου L αριστερό μέρος του κειμένου και R το δεξί. Στη συνέχεια και για κάθε επόμενη επανάληψη εφαρμόζουμε την πράξη $L_{i+1} = R_i$ και $R_{i+1} = L_i \, XOR \, f(R_i, K_i)$. Το δίκτυο Feistelαποτελεί το κύριο συστατικό για τους αλγόριθμους κρυπτογράφησης τμημάτων.

4.1.1 Ο αυθεντικός αλγόριθμος

Ο αλγόριθμος Data Encyption Standard κρυπτογραφεί πληροφορίες σε μορφή δυφίων. Επομένως, πρώτο βήμα για τον αλγόριθμο είναι η μετατροπή του κλειδιού και του κειμένου στο δυαδικό σύστημα. Στη συνέχεια για κάθε 64bits του κλειδιού μετατίθενται χρησιμοποιώντας τον πίνακα PC-1 (Permutation Choice 1) ο οποίος είναι σχεδιασμένος με τέτοιο τρόπο ώστε να μην χρησιμοποιεί το $8^{ο}$ δυφίο κάθε byte του κλειδιού. Αυτό έχει ως αποτέλεσμα το αποτέλεσμα της μετάθεσης του PC-1 να είναι κλειδί μήκους 56bits.

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Πίνακας 4.1 Permuted Choice One

Στη συνέχεια το νέο κλειδί μοιράζεται σε δύο τμήματα C και D τα οποία αποτελούνται από 28bits το καθένα. Το C αντιστοιχεί στα πρώτα 28bits και το D στα υπόλοιπα. Θέλουμε να δημιουργήσουμε 16 νέα κλειδιά χρησιμοποιώντας τα τμήματα C και D. Επομένως, εκτελούμε κυκλικές μεταθέσεις πάνω στα τμήματα C και D ούτως ώστε να προκύψουν 16 καινούρια τμήματα από τα προηγούμενα. Ο αριθμός των κυκλικών μεταθέσεων για κάθε μια από τις 16 επαναλήψεις φαίνεται στον πίνακα 4.2.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

Πίνακας 4.2 Μεταθέσεις ανά επανάληψη

Τέλος συγχωνεύουμε τα νέα C,D που προκύπτουν και εκτελούμε μεταθέσεις σε κάθε νέο κλειδί βάση του πίνακα PC-2 (Permuted Choice 2). Ο πίνακα PC-2 μετατρέπει το κλειδί από 56bits σε 48bits και φαίνεται στον Πίνακα 4.3. Η μετάθεση με τη χρήση του πίνακα γίνεται ως εξής: Το στοιχείο του νέου κλειδιού στη θέση i είναι το στοιχείο του παλιού κλειδιού στη θέση που αναγράφεται στη i θέση του PC-2. Άρα, έστω K56 το προηγούμενο κλειδί τότε K48[1] = K56[PC-2[1]].

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Πίνακας 4.3 Permuted Choice Two

Με την εκτέλεση των μεταθέσεων του πίνακα PC-2 παίρνουμε τα τελικά κλειδιά που θα χρησιμοποιηθούν στην κρυπτογράφηση. Επομένως το επόμενο βήμα είναι η κρυπτογράφηση του κειμένου. Ο αλγόριθμος κρυπτογραφεί τμήματα 64bits ανά επανάληψη επομένως η διαδικασία που θα περιγραφεί πιο κάτω εκτελείται για κάθε τμήμα 64bits του κειμένου εισόδου.

Αρχικά, εκτελείται μια μετάθεση στο τμήμα M που περιέχει το τμήμα 64bits προς κρυπτογράφηση. Η αρχική αυτή μετάθεση γίνεται βάση του πίνακα IP (Initial Permutation) ο οποίος λειτουργεί όπως και οι πίνακες PC1 και PC2. Ο πίνακας IP κρατά σταθερές τις διαστάσεις του M. Έπειτα, χωρίζουμε την είσοδο σε τμήματα L και R τα οποία αντιστοιχούν στο αριστερό και δεξί μισό του M αντίστοιχα και εφαρμόζουμε το δίκτυο Feistelπάνω στα δύο τμήματα. Το δίκτυο Feistelεκτελείται όπως πιο πάνω με 16 επαναλήψεις όπου σε κάθε επανάληψη δημιουργούνται νέες τιμές των L και R με τις πράξεις: $L_{i+1} = R_i$ και $R_{i+1} = L_i \; XOR \; f(R_i, K_i)$. Τέλος αντιστρέφουμε τη σειρά των R_{16} και L_{16} και μεταθέτουμε το νέο τμήμα $R_{16}L_{16}$ με βάση τον αντίστροφο του πίνακα IP. Ο IP^{-1} λειτουργεί όπως και οι πίνακες PC1, PC2 και IP. Ο IP^{-1} φαίνεται στον Πίνακα 4.4.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Πίνακας 4.4 Initial Permutation (Inverse)

Κατά την εκτέλεση του δικτύου Feistelγίνεται η χρήση της συνάρτησης f. Η συνάρτηση f για κάθε επανάληψη i παίρνει ως είσοδο το δεξί τμήμα R_{i-1} και το κλειδί K_i. Το κλειδί όμως είναι μήκους 48bits ενώ το R έχει μήκος 32bits. Χρειάζεται επομένως να γίνει μια τροποποίηση του

R για να έχει το ίδιο μήκος με το κλειδί. Για τον λόγο αυτό περνούμε το τμήμα R μέσα από τον πίνακα E-bit Selection. Ο πίνακας E-bit Selection μετασχηματίζει το R επαναλαμβάνοντας κάποια τμήματα του R. Ο E-bit Selection φαίνεται στον Πίνακα 4.5.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Πίνακας 4.5 E Bit Selection

Στη συνέχεια εκτελείται πρόσθεση ανά δυφίο (bit-by-bit addition) στην έξοδο του πίνακα E-bit Selection και του κλειδιού και έχουμε ως αποτέλεσμα ένα τμήμα 48bits. Έπειτα, η έξοδος χωρίζεται σε 8 τμήματα των 6bits. Κάθε τμήμα περνά μέσα από ένα διαφορετικό Sbox. Τα Sboxes είναι πίνακες οι οποίοι δέχονται ως είσοδο τη γραμμή και τη στήλη και επιστρέφουν τη τιμή που αναγράφεται στη συγκεκριμένη θέση. Τα 8 Sboxes φαίνονται στον Πίνακα 4.6.

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

(α) S box 1

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

(β) S box 2

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

(γ) S box 3

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

(δ) S box 4

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

(ε) S box 5

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

(ζ) S box 6

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

(η) S box 7

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

(θ) S box 8

Πίνακας 4.6 S box 1-8 στα σχήματα α-θ αντίστοιχα

Για τον υπολογισμό της στήλης και της γραμμής που θα μας δώσουν την έξοδο του Sbox χρησιμοποιούμε την εξής τεχνική: 1) για τον υπολογισμό της γραμμής παίρνουμε το πρώτο και το τελευταίο δυφίο της εισόδου Β και το μετατρέπουμε στο δυαδικό σύστημα. Έτσι έχουμε μια τιμή στο διάστημα [0,3], όσες και οι γραμμές, (2) για τον υπολογισμό της στήλης παίρνουμε τα ενδιάμεσα 4 δυφία και τα μετατρέπουμε στο δυαδικό σύστημα. Έτσι έχουμε μια τιμή στο διάστημα [0,15], όσες και οι στήλες. Τέλος, μετατρέπουμε τις εξόδους των Sboxes στο δυαδικό σύστημα, τις συγχωνεύουμε σε μια τιμή και τις περνούμε ως είσοδο στον πίνακα μεταθέσεων P. Ο πίνακας μεταθέσεων P μετασχηματίζει την είσοδο του σε 32bits έξοδο για να μπορεί να αντικαταστήσει την τιμή του R (και αυτό 32bits). Ο πίνακας P φαίνεται στον Πίνακα 4.7.

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Πίνακας 4.7 Permutation P

Στο τελικό βήμα γίνεται μετάθεση του αποτελέσματος με τη χρήση του πίνακα IP^{-1} ο οποίος φαίνεται στον Πίνακα 4.8.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Πίνακας 4.8 Initial Permutation Inverse

4.1.2 Ο αναστρέψιμος αλγόριθμος

Σε αυτή την ενότητα θα δοθούν διάφορα κομμάτια του αναστρέψιμου αλγορίθμου σε μορφή συναρτήσεων και στη συνέχεια θα δοθεί η σειρά που πρέπει να εκτελεστούν για να παραχθεί ο αναστρέψιμος αλγόριθμος.

Η πιο κάτω συνάρτηση ονομάζεται convert_to_64binary και δέχεται ως παραμέτρους τους πίνακες key και K64 και τη στοίβα garbage. Ο πίνακας key αποτελεί την είσοδο η οποία θα μετατραπεί σε μια σειρά από 64bits και θα αποθηκευτεί στον πίνακα K64.

1. local int count = 0
2. local stack result = nil
3. local int i=0
4. from i=0 loop
    a. local int temp = key[i]
    b. call BinaryDecimal(temp,result)
    c. delocal int temp =0

d. local int j = size(result)
e. from j = size(result) loop
    i. count+=1
    ii. j+=1
f. until j =4
g. delocal int j =4
h. local int j = size(result)
i. local int z =j
j. from z=j loop
    i. local int temp =0
    ii. pop(temp,result)
    iii. K64[count] +=temp
    iv. delocal int temp = K64[count]
    v. count+=1
    vi. z-=1
k. until 0
l. delocal int z=0
m. push(j,garbage)
n. delocal int j=0
o. i+=1

5. until i=16
6. delocal int i=16
7. delocal stack result = nil
8. delocal int count = size(K64)

Ο αλγόριθμος μετατρέπει το κλειδί από 16 ακέραιες τιμές σε 64bits. Στο βήμα 5.b γίνεται κλήση της συνάρτηση BinaryDecimal η οποία μετατρέπει την είσοδο της στο δυαδικό σύστημα και επιστρέφει το αποτέλεσμα σε μια στοίβα. Το αποτέλεσμα όμως είναι στην ελάχιστη μορφή ψηφίων που απαιτούνται, δηλαδή το 5 μετατρέπεται σε 101. Επομένως, για σκοπούς αναστρεψιμότητας, στα μετέπειτα βήματα της διαδικασίας προσθέτουμε 0 στην στοίβα μέχρι το αποτέλεσμα της στοίβας να έχει μήκος 4bits διότι χρειαζόμαστε συνοχή στα αποτελέσματα μας για να μειώσουμε την μνήμη που θα χρειαστεί για τα επιπλέον σκουπίδια. Δηλαδή σε περίπτωση που έχουμε σαν είσοδο την τιμή 5 το αποτέλεσμα της στοίβας από 101 θα γίνει 0101

ενώ σε περίπτωση που έχουμε είσοδο 15 το αποτέλεσμα θα παραμείνει 1111. Αυτό έχει ως αποτέλεσμα την αποθήκευση επιπλέον σκουπιδιών (τα επιπρόσθετα 0). Τέλος, το αποτέλεσμα αποθηκεύεται στον πίνακα K64.

Η πιο κάτω συνάρτηση ονομάζεται convert_64key_to_56key και παίρνει ως παραμέτρους τους πίνακες K64 και K56 καθώς και τον πίνακα μεταθέσεων PC1. Οι δύο πίνακες αποτελούν την είσοδο και την έξοδο της αντίστοιχα. Η συνάρτηση είναι υπεύθυνη για την μετάθεση του κλειδιού βάση του πίνακα PC1 που φαίνεται στον Πίνακα 4.1.

1. local int i=0
2. from i=0 loop
    a. K56[i] += K64[PC1[i]-1]
    b. i+=1
3. until i=56
4. delocal int i=56
5. delocal int PC1[56]

Στη μετατροπή από 64bits σε 56bits γίνεται χρήση του πίνακα Permuted Choice One (PC1). Κάθε θέση του νέου κλειδιού ισούται με την τιμή του προηγούμενου κλειδιού με δείκτη την τιμή του πίνακα PC1. Οι τιμές του πίνακα είναι τέτοιες ώστε το $8^{ο}$ bit κάθε byte να μην επιλέγεται σε καμία περίπτωση.

Η πιο κάτω συνάρτηση ονομάζεται create_C_D και παίρνει ως παραμέτρους το κλειδί K56, τους πίνακες δισδιάστατους C,D και τον πίνακα shifts ο οποίος περιέχει τον αριθμό των κυκλικών μεταθέσεων για κάθε επανάληψη όπως φαίνονται στον Πίνακα 4.2.

1. local int i =0
2. from i=0 loop
    a. if I < 28 then
        i. C[0][i%28] +=K56[i]
    b. else
        i. D[0][i%28] +=K56[i]

    c. fi i<28
    d. i+=1
3. until i=56
4. delocal int i=56
5. local int i=1
6. from i=1 loop
    a. local int j=shifts[i-1]
    b. local int z=0
    c. from z=0 loop
        i. C[i][z] +=C[i-1][z+j]
        ii. D[i][z] +=D[i-1][z+j]
        iii. z+=1
    d. until z=28-j
    e. local int p=0
    f. from p =0 loop
        i. C[i][z] +=C[i-1][p]
        ii. D[i][z] +=D[i-1][p]
        iii. p+=1
        iv. z+=1
    g. until p = j
    h. delocal int p=j
    i. delocal int z =28
    j. delocal int j = shifts[i-1]
    k. i+=1
7. until i=17
8. delocal int i=17

Οι πίνακες C και D έχουν 17 γραμμές μήκους ίσου με το μισό του τροποποιημένου κλειδιού των 56bits. Για κάθε επανάληψη εκτελούμε αριστερές μεταθέσεις στην προηγούμενη γραμμή με βάση τον πίνακα shifts. Έτσι, το αποτέλεσμα της πιο πάνω συνάρτησης είναι δύο πίνακες (C και D) οι οποίοι περιέχουν το πρώτο και δεύτερο μισό του κλειδιού στη πρώτη γραμμή και στις υπόλοιπες δεκαέξι γραμμές έχουν μεταθέσεις για δημιουργία ισάριθμων νέων κλειδιών.

Προχωρούμε στη δημιουργία των τελικών κλειδιών με τη συνάρτηση create_final_key η οποία δέχεται ως παραμέτρους το αρχικό τελικό κλειδί finalKey όπου θα είναι και η έξοδος της συνάρτησης, τους πίνακες C,D που υπολογίστηκαν προηγουμένως και τον πίνακα PC2 αφού για τη δημιουργία των τελικών κλειδιών χρειάζεται για τις μεταθέσεις.

1. local int i =0
2. from i=0 loop
    a. local int j =0
    b. from j =0 loop
        i. if PC2[j] <=28 then
            1. finalKey[i][j] += C[i+1][PC2[j]-1]
        ii. else
            1. finalKey[i][j] += D[i+1][PC2[j]-28-1]
        iii. fi PC2[j] <=28
        iv. j+=1
    c. until j=48
    d. delocal int j =48
    e. i+=1
3. until i=16
4. delocal int i=16

Για την δημιουργία των τελικών κλειδιών γίνεται χρήση ενός δεύτερου πίνακα μεταθέσεων, του Permuted Choice 2 (PC2) ο οποίος φαίνεται στον πίνακα 4.3. Η διαδικασία δημιουργίας είναι η ίδια με πιο πάνω όπου χρησιμοποιούνται οι τιμές του πίνακα PC2 ως δείκτες για την νέα τιμή. Τις τιμές για το πρώτο μισό κάθε κλειδιού γίνεται χρήση του πίνακα C ενώ για το υπόλοιπο κλειδί γίνεται χρήση του πίνακα D. Επομένως, για κάθε μια από τις 16 επαναλήψεις συγχωνεύουμε τους πίνακες CD και μεταθέτουμε τον νέο πίνακα βάση του PC2. Τέλος, προσθέτουμε τη νέα τιμή στον πίνακα finalKey.

Η πιο κάτω συνάρτηση ονομάζεται transform και δέχεται ως ορίσματα τον πίνακα input_bin, ο οποίος περιέχει την είσοδο σε δυαδική μορφή η οποία προκύπτει από τη συνάρτηση convert_to_64binary και τον πίνακα IP ο οποίος περιέχει τις πληροφορίες για την αρχική μετάθεση της εισόδου.

1. local int IP[]
2. local int i=0
3. from i=0 loop
    a. transformed[i] +=input_bin[IP[i]-1]
    b. i+=1
4. until i=64
5. delocal int i=64
6. delocal int IP[]

Σε αυτό το βήμα του αλγορίθμου χρησιμοποιούμε τον πίνακα Initial Permutation (πίνακας 4.4) για μετάθεση της εισόδου. Επομένως, για κάθε στοιχείο του input_bin η νέα τιμή είναι η τιμή του input_bin στη θέση της τιμής του IP, όπως φαίνεται στο βήμα 3.α.

Η επόμενη συνάρτηση είναι υπεύθυνη για την δημιουργία των κομματιών L και R με βάση την είσοδο καθώς και τη συνάρτηση f του δικτύου feistel. Η συνάρτηση create_L_R παίρνει ως είσοδο τα τελικά κλειδία finalKey και την είσοδο transformed. Ως παράμετροι που θα αποθηκεύσουν την έξοδο έχουμε τους πίνακες L και R και τον πίνακα F ο οποίος θα αποθηκεύει την έξοδο της συνάρτησης functionF η οποία βάση της λογικής του δικτύου Feistelγίνεται XOR με το L.

1. local int i =0
2. from i=0 loop
    a. if (i<32) then
        i. L[0][i] +=transformed[i]
    b. else
        i. R[0][i%32] += transformed[i]
    c. fi i<32
    d. i+=1
3. until i=64
4. delocal int i=64
5. local int i=1
6. from i=1 loop

    a. call functionF(R,i-1,finalkey,F,garbage)
    b. local int j=0
    c. from j=0 loop
        i. L[i][j] +=R[i-1][j]
        ii. R[i][j] +=(L[i-1][j] + F[i-1][j])%2
        iii. j+=1
    d. until j=32
    e. delocal int j =32
    f. i+=1
7. until i=17
8. delocal int i=17

Οι πίνακες L και R αποτελούνται από 17 γραμμές. Στην πρώτη τους γραμμή αρχικοποιούνται με το πρώτο και δεύτερο μισό της τροποποιημένης εισόδου αντίστοιχα. Στη συνέχεια, εκτελούμε επαναληπτικά για κάθε επόμενη γραμμή των L,R: $L_n = R_{n-1}$, $R_n = L_{n-1} + f(R_{n-1},K_n)$ όπου K_n το κλειδί στη συγκεκριμένη γραμμή και f η συνάρτηση τροποποίησης του δικτύου feistel.

Η συνάρτηση functionF αποτελεί την υλοποίηση της συνάρτησης f. Αρχικά θα εξηγήσουμε την προεργασία που πρέπει να γίνει προτού δείξουμε τον αλγόριθμο για την συνάρτηση f. Όπως έχει φανεί πιο πάνω, οι πίνακες L και R έχουν μήκος ίσο με το μισό της εισόδου, άρα 32bits. Για να μπορέσουμε να εκτελέσουμε πράξεις πάνω στους πίνακες μαζί με το κλειδί πρέπει να επεκτείνουμε τον πίνακα R στα 48bits. Η συνάρτηση createE δέχεται ως παραμέτρους τον πίνακα E ο οποίος περιέχει τις πληροφορίες του πίνακα E-bit Selection όπως φαίνεται στον Πίνακα 4.5, τον πίνακα finalKey ο οποίος περιέχει το κλειδί για την εκάστοτε επανάληψη και τον πίνακα R ο οποίος περιέχει τη μισή είσοδο για την εκάστοτε επανάληψη. Τέλος, η συνάρτηση έχει ως παράμετρο τον πίνακα E_out ο οποίος αποτελεί και την έξοδο. Η διαδικασία που εκτελείται είναι η εξής: για κάθε ένα από τα 48bits του πίνακα E_out, ενημέρωσε τη κάθε θέση του πίνακα με το αποτέλεσμα της πράξης (R'+finalKey)%2 όπου το R έχει πρώτα επεκταθεί από τον πίνακα E.

1. local int i=0
2. from i=0 loop

    a. E_out[i] += (R[E[i]-1] +finalKey[i]) %2
    b. i+=1
3. until i=48
4. delocal int i=48

Στη συνέχεια, χωρίζουμε την έξοδο της προηγούμενης διαδικασίας (E_out) σε 8 τμήματα των 6bits με τη συνάρτηση createBoxes. Η createBoxes παίρνει ως παραμέτρους την έξοδο της προηγούμενης συνάρτησης E_out και τον πίνακα Boxes ο οποίος θα αποθηκεύσει την έξοδο.

1. local int i=0
2. from i=0 loop
    a. local int j=0
    b. from j=0 loop
        i. Boxes[i][j] +=E_out[i*6+j]
        ii. j+=1
    c. until j=6
    d. delocal int j=6
    e. i+=1
3. until i=8
4. delocal int i=8

Για κάθε γραμμή του πίνακα Boxes, παίρνουμε το πρώτο και τελευταίο bit και το μετατρέπουμε στο δεκαδικό σύστημα για να βρούμε τον δείκτη της γραμμής για το εκάστοτε S box και τα υπόλοιπα 4 bits στο δεκαδικό αντιστοιχούν στον δείκτη της στήλης του εκάστοτε S box. Η συνάρτηση Sbox παίρνει ως παράμετρο τον αριθμό number που είναι η έξοδος του Sbox, τον πίνακα Boxes και τη βοηθητική στοίβα garbage που αποθηκεύει τα επιπλέον σκουπίδια.

1. local int S[8][64] = Τα 8 Sboxes στις αντίστοιχες γραμμές του πίνακα.
2. local int i=0
3. from i=0 loop
    a. local int row = Box[i]0]*10+Box[i][5]
    b. local int column =Box[i][1]*1000+Box[i][2]*100+Box[i][3]*10+Box[i][4

    c. uncall DecimalBinary(row)
    d. uncall DecimalBinary(column)
    e. number[i] += S[i][row*16+column]
    f. push(row,garbage)
    g. push(column,garbage)
    h. delocal int column =0
    i. delocal int row =0
    j. i+=1
4. until i=8
5. delocal int i=8
6. delocal int S[8][64]

Αρχικά παίρνουμε τις τιμές των αντίστοιχων bit και με ανάστροφη κλήση της συνάρτησης DecimalBinary παίρνουμε την αντίστοιχη δεκαδική τιμή. Στη συνέχεια ενημερώνουμε τη μεταβλητή number με το αποτέλεσμα του Sbox. Για σκοπούς αναστρεψιμότητας, πρέπει να αποθηκευτεί η τιμή της κάθε γραμμής και στήλης στη στοίβα garbage καθώς δεν έχουμε άλλο τρόπο να το υπολογίσουμε για να το αποδεσμεύσουμε. Ο βρόγχος εκτελείται για κάθε ένα από τα 8 τμήματα (Boxes).

Τέλος, η συνάρτηση functionF υλοποιεί τη συνάρτηση f του δικτύου Feistelγια τον αλγόριθμο Data Encryption Standard. Η συνάρτηση δέχεται ως παραμέτρους τον πίνακα με τα τελικά κλειδιά finalKey, τη βοηθητική στοίβα garbage που αποθηκεύει τα επιπλέον σκουπίδια, τον πίνακα number ο οποίος περιέχει την έξοδο των 8 Sboxes και τον πίνακα P ο οποίος περιέχει πληροφορίες σχετικά με την σμίκρυνση του αποτελέσματος για να χωρέσει στα 32bits του πίνακα R. Ο πίνακας P είναι όπως φαίνεται στον Πίνακα 4.7.

1. local int aux[32]={0}
2. local int i=0
3. from i=0 loop
    a. local stack result =nil
    b. local int temp =number[i]
    c. call BinaryDecimal(temp,result)
    d. delocal int temp =0

e. local int j=size(result)
f. if (j<4) then
    i. local int z=j
    ii. from z=j loop
        1. push(0,result)
        2. z+=1
    iii. until z=4
    iv. delocal int z=4
g. fi (j<4)
h. push(j,help)
i. delocal int j=0
j. local int j=0
k. from j=0 loop
    i. local int temp=0
    ii. pop(temp,result)
    iii. aux[4*i+j] += temp
    iv. temp-=aux[4*i+j]
    v. delocal int temp =0
    vi. j+=1
l. until j=4
m. delocal int j=4
n. delocal stack result =nil
o. i+=1

4. until i=8
5. delocal int i=8
6. Μεταφορά ‘σκουπιδιών’ στην στοίβα garbage
7. local int i=0
8. from i=0 loop
    a. F[index][i] +=aux[P[i]-1]
    b. push(aux[P[i]-1],garbage)
    c. aux[P[i]-1] -= top(garbage)
    d. i+=1

9. until i=32
10. delocal int i=32
11. delocal int aux[32]={0}

Στον πιο πάνω βρόγχο μετατρέπουμε την έξοδο των S boxes στο δυαδικό σύστημα και εφαρμόζουμε την ίδια τεχνική με πριν, προσθέτοντας ασήμαντα bits 0 μέχρι κάθε στοιχείο να έχει 4bits. Το τελικό βήμα της συνάρτησης είναι μια σμίκρυνση της εξόδου από 48bits σε 32bits. Το αποτέλεσμα της συνάρτησης functionF βρίσκεται στον πίνακα F.

Πιο κάτω δίνεται η συνάρτηση encryption η οποία ολοκληρώνει τον αλγόριθμο με την τελευταία μετάθεση του πίνακα IP^{-1} που φαίνεται στον Πίνακα 4.8. Η συνάρτηση παίρνει ως παράμετρο τον πίνακα IP_REV ο οποίος περιέχει τα στοιχεία του πίνακα IP^{-1} και τους πίνακες R και L. Επιπρόσθετα, δέχεται ως παράμετρο τον πίνακα encrypted που λειτουργεί ως έξοδο για τη συνάρτηση.

1. local int IP_REV[64]
2. local int i=0
3. from i=0 loop
    a. if (IP_REV[i] <= 32) then
        i. encrypted[i] += R[16][IP_REV[i] -1]
    b. else
        i. encrypted[i] += L[16][IP_REV[i] -1 -32]
    c. IP_REV[i] <=32
    d. i+=1
4. until i=64
5. delocal int i=64
6. delocal int IP_REV[64]

Στο τελικό βήμα αντιστρέφεται ο πίνακας L με τον R και γίνεται μια τελευταία μετάθεση με τον πίνακα IP^{-1}.
Στο τελικό βήμα του αλγορίθμου έχουμε τη συνάρτηση createFinalText η οποία δέχεται ως παράμετρο τον πίνακα encrypted που αποτελεί το κρυπτογραφημένο κείμενο σε bits και τον

πίνακα EncryptedText που αποτελεί την έξοδο της συνάρτησης. Η συνάρτηση μετατρέπει τον πίνακα encrypted στο δεκαδικό σύστημα και τον αποθηκεύει στον πίνακα EncryptedText.

1. local int encrypted[64]={0}
2. call encryption(key,input,encrypted)
3. local int i=0
4. from i=0 loop
    a. local int j=encrypted[4*i]*1000 +encrypted[4*i+1]*100 +encrypted[4*i+2] + encrypted[4*i+3]
    b. uncall DecimalBinary(j)
    c. EncryptedText[i] +=j
    d. delocal int j= EncryptedText[i]
    e. i+=i
5. until i=16
6. delocal int i=16
7. uncall encryption(key,input,encrypted)
8. delocal int encrypted[64]={0}

Επομένως τα βήματα του αναστρέψιμου αλγορίθμου αποτελούνται από την εμπρόσθια κλήση των εξής συναρτήσεων με την ακόλουθη σειρά.

1. call convert_to_binary64(key, K64, garbage)
2. call convert_to_binary64(input, input_bin, garbage)
3. call convert_64key_to_56key(K64, K56,PC1)
4. call create_C_D(C, D, shifts)
5. call create_final_key(finalKey, C, D, PC2)
6. call transform(input_bin, IP)
7. call create_L_R(finalKey, L, R, F, transform)
8. call encryption(L, R, IP_REV, encrypted)
9. call createFinalText(encrypted, EncryptedText)

4.1.3 Η υλοποίηση στη γλώσσα Janus

Για επίτευξη αποδοτικής αναστρεψιμότητας και χάρη στο γεγονός ότι στην αυθεντική εκδοχή του ο αλγόριθμος είναι αρθρωτός, κάθε διαδικασία εκτελείται διαδοχικά και με το πέρας του αλγορίθμου γίνεται ανάστροφη κλήση των διαδικασιών με ανάποδη σειρά. Έτσι πετυχαίνουμε κρυπτογραφία χωρίς επιπλέον σκουπίδια, καθιστώντας την αναστρέψιμη υλοποίηση το ίδιο ασφαλή με την μη αναστρέψιμη αφού οι δύο εμπλεκόμενοι μπορούν να επικοινωνήσουν με την αποστολή μόνο του κρυπτογραφημένου κειμένου.

Για την υλοποίηση με τον τρόπο αυτό πρέπει να αλλάξουν οι παράμετροι κάθε συνάρτησης, καθώς και οι συναρτήσεις πλέον στην αρχή εκτελούν εμπρόσθια κλήση της συνάρτησης που εκτελεί το προηγούμενο βήμα, στη συνέχεια τη λειτουργία που καλούνται υλοποιήσουν και τέλος γίνεται ανάστροφη κλήση της συνάρτησης του προηγούμενου βήματος για αποδέσμευση οποιασδήποτε μνήμης χρειάστηκε να υπολογισμό της συνάρτησης. Αυτή η μέθοδος προγραμματισμού ονομάζεται "compute-uncompute" καθώς υπολογίζει και στην συνέχεια αναστρέφει τον προηγούμενο υπολογισμό.

Ο πλήρης κώδικας υλοποίησης σε γλώσσα Janus με τη μέθοδο "compute-uncompute" φαίνεται στο παράρτημα Β.

4.1.4 Ανάλυση Αλγορίθμου

Ο αλγόριθμος εκτελείται για κάθε τμήμα 64bits που υπάρχει στην είσοδο. Κάθε συνάρτηση του αλγορίθμου εκτελεί μια γραμμική πράξη είτε μετάθεσης είτε XOR πάνω στα δεδομένα. Επομένως είναι εύκολο να παρατηρήσουμε ότι οι συναρτήσεις έχουν γραμμική χρονική πολυπλοκότητα. Άρα, η χρονική πολυπλοκότητα του αλγορίθμου είναι της τάξεως Θ(N) όπου N ο αριθμός των τμημάτων 64bits που υπάρχουν στην είσοδο.

Η χρονική πολυπλοκότητα του αλγορίθμου εξαρτάται από τα τμήματα των 64bits που υπάρχουν στην είσοδο. Κάθε συνάρτηση χρησιμοποιεί επιπλέον πίνακες ίδιου μήκους για αποθήκευση των αποτελεσμάτων όμως αυτό δεν αλλάζει την ασυμπτωτική χωρική

πολυπλοκότητα της συνάρτησης. Επομένως, η χωρική πολυπλοκότητα της συνάρτησης είναι της τάξεως Θ(N).

Για καλύτερη μελέτη της χρονικής πολυπλοκότητας έγινε καταμέτρηση του χρόνου εκτέλεσης της υλοποίησης του αλγορίθμου στη γλώσσα Janus. Οι μετρήσεις φαίνονται στον Πίνακα 4.9 και οι χρόνοι είναι σε δευτερόλεπτα. Δίνονται δέκα μετρήσεις για κάθε είσοδο με σκοπό να έχουμε πιο ορθό υπολογισμό της χρονική πολυπλοκότητας παίρνοντας το μέσο όρο.

64bits	128bits	172bits	256bits	320bits	384bits	4096bits
4,993	11,715	15,062	19,968	25,048	30,246	327,079
5,005	10,007	15,241	20,217	25,078	30,236	329,828
4,967	10,148	15,238	20,083	25,088	30,268	335,612
5,013	10,119	15,219	20,248	24,867	29,959	321,165
4,976	10,214	15,536	20,26	24,845	29,749	335,214
5,002	10,171	15,284	20,324	25,012	30,637	298,925
5,075	10,061	15,399	20,318	24,953	30,651	338,582
5,022	10,239	15,178	20,363	24,987	30,888	322,402
5,247	10,393	15,226	20,419	24,969	29,705	328,191
5,005	10,11	15,171	20,28	25,067	29,885	328,395
Μέσος όρος						
5,0305	10,3177	15,2554	20,248	24,9914	30,2224	326,5393

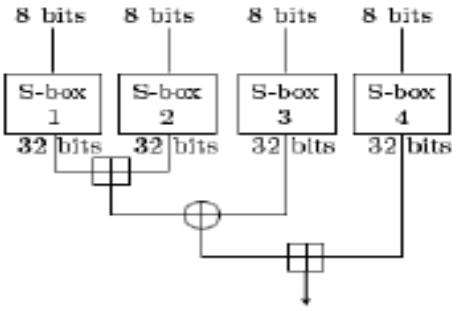
Πίνακας 4.9 Πειραματική αξιολόγηση χρονικής πολυπλοκότητας

Η υλοποίηση του αλγορίθμου είναι πιστή καθώς κρατά σταθερή την πολυπλοκότητα του αλγορίθμου και είναι υγιής καθώς δεν χρειάζεται επιπλέον σκουπίδια με το πέρας της εκτέλεσης. Επομένως η συνάρτηση κρατά τη συνάρτηση αναπαράστασης των σκουπιδιών στο ελάχιστο και εξ ‘ου η «υγιεινή» της υλοποίησης.

4.2 Ο αλγόριθμος Blowfish

Ο αλγόριθμος Blowfish αποτελεί αλγόριθμο κρυπτογράφησης με μυστικό κλειδί βασισμένο σε δίκτυο Feistelκαι παρακάτω θα παρουσιαστεί με λεπτομέρεια η αναστρέψιμη υλοποίηση του. Ο αλγόριθμος προτάθηκε από τον Bruce Schneier το 1994 [12]. Η συνάρτηση f και το Feistelδίκτυο για τον αλγόριθμο blowfish φαίνονται στο σχήμα 4.3.

Ο αλγόριθμος τροποποιεί τα δεδομένα εισόδου και το κλειδί με βάση τον πίνακα P και τέσσερα μοναδικά S boxes τα οποία είναι αρχικοποιημένα με τα δεκαεξαδικά ψηφία του π εκτός των πρώτων τριών. Το δίκτυο Feistelγια τον αλγόριθμο Blowfish αποτελείται από 16 επαναλήψεις και μόνο πράξεις XOR και πρόσθεσης.



Σχήμα 4.3 F function

4.2.1 Ο αυθεντικός αλγόριθμος

Ο αλγόριθμος Blowfish είναι ένας συμμετρικός αλγόριθμος κρυπτογράφησης με κρυφό κλειδί ο οποίος εκτελείται σε τμήματα των 64bits. Ο αλγόριθμος εκτελεί την κρυπτογράφηση σε δύο μέρη. Στο πρώτο μέρος αρχικοποιούνται οι πίνακες P και Sboxes με τα ψηφία του π. Ο πίνακας P αποτελείται από 18 στοιχεία και υπάρχουν 4 πίνακες Sbox που αποτελούνται από 256 στοιχεία. Κάθε στοιχείο μετατρέπεται σε 32bits και εκτελούμε πράξη XOR με κάθε 32bits του κλειδιού. Αν το κλειδί εξαντληθεί τότε επαναλαμβάνουμε ξανά τα τμήματα των 32bits του. Με το πέρας των πράξεων XOR και αφότου έχουν ενημερωθεί όλες οι τιμές των P και Sboxes κρυπτογραφούμε μια ακολουθία από 0 μήκους ίσου με την είσοδο. Αντικαθιστούμε το P1 και P2 με το πρώτο και δεύτερο μισό της εξόδου της κρυπτογράφησης αντίστοιχα. Στη συνέχεια ξανά κρυπτογραφούμε τη έξοδο αυτή και αντικαθιστούμε το P3, P4 όπως και πριν. Επαναλαμβάνουμε τη διαδικασία μέχρι να ενημερωθούν όλοι οι πίνακες και χρειαζόμαστε σύνολο 521 επαναλήψεις για να τερματίσει.

Με το πέρας του πρώτου μέρους, προχωρούμε στη κρυπτογράφηση της εισόδου. Εδώ αξίζει να σημειωθεί ότι η κρυπτογράφηση που γίνεται στο πρώτο μέρος πάνω στην μηδενική είσοδο είναι η ίδια που θα εξηγηθεί πιο κάτω. Αρχικά, χωρίζουμε την είσοδο σε δύο μέρη των 32bit, L και R. Στη συνέχεια εκτελούμε 16 επαναλήψεις με τις πράξεις που ορίζει το δίκτυο feistel. Εκτελούμε πράξη XOR μεταξύ του L και του P_i όπου i ο αριθμός της επανάληψης και μετά εισάγουμε το αποτέλεσμα στο L. Το τμήμα L στη συνέχεια περνά μέσα από την συνάρτηση F

και εκτελείται XOR με το αποτέλεσμα και το τμήμα R. Το τμήμα R ενημερώνεται με το αποτέλεσμα. Στη συνέχεια ανταλλάζουμε τις τιμές L και R και προχωρούμε στην επόμενη επανάληψη. Με το πέρας των 16 επαναλήψεων, ανταλλάζουμε ξανά τις τιμές L και R αναιρώντας έτσι τη τελευταία ανταλλαγή και εκτελούμε XOR με τους πίνακες P17 και P18 αντίστοιχα. Τέλος, συγχωνεύουμε τα δύο τμήματα και έχουμε το τελικό αποτέλεσμα.

Η συνάρτηση F για το δίκτυο Feisteltου αλγόριθμου Blowfish έχει ως εξής: Αρχικά χωρίζουμε το L σε 4 κομμάτια των 8bits τα οποία είναι τα a,b,c,d αντίστοιχα. Το αποτέλεσμα της συνάρτησης F είναι το αποτέλεσμα της πράξης ((S1[a] + S2[b] mod 2^{32}) XOR S3[c]) + S4[d] mod 2^{32}.

4.2.2 Ο αναστρέψιμος αλγόριθμος

Στο πρώτο βήμα του αλγορίθμου μετατρέπουμε το κλειδί σε 64bits όπως και στον αλγόριθμο DES με την συνάρτηση convert_to_64bits. Στη συνέχεια χωρίζουμε το κλειδί σε δύο τμήματα που περιέχουν το πρώτο και το δεύτερο μισό αντίστοιχα. Η συνάρτηση αυτή ονομάζεται createLR και δέχεται ως παραμέτρους τον πίνακα K64 που περιέχει το κλειδί σε μορφή 64bits και τους πίνακες L,R που θα αποθηκεύσουν την έξοδο.

1. local int i=0
2. from i=0 loop
    a. L[i]+=K64[i]
    b. i+=1
3. until i=32
4. from i=32 loop
    a. R[i-32] +=K64[i]
    b. i+=1
5. until i=64
6. delocal int i=64

Η διαδικασία αποτελείται από δύο βρόγχους όπου ο πρώτος ενημερώνει τον πίνακα L με τις πρώτες 32 τιμές του K64 και ο δεύτερος ενημερώνει τον πίνακα R με τις υπόλοιπες 32.

Η πιο κάτω συνάρτηση είναι υπεύθυνη για ενημέρωση του πίνακα P. Η initialize_P παίρνει ως παραμέτρους το πίνακα P, τους πίνακες L και R, τη βοηθητική στοίβα garbage για αποθήκευση σκουπιδιών και τον πίνακα Parray που θα αποθηκεύσει το αποτέλεσμα. Ο πίνακας P αρχικά έχει τιμές τις δεκαεξαδικές αναπαραστάσεις των ψηφίων του π εκτός των πρώτων 3 και ο πίνακας Parray θα λάβει την νέα τροποποιημένη τιμή με βάση το κλειδί και τον πίνακα P.

1. local int i=0
2. from i=0 loop
    a. local int AuxP[32] ={0}
    b. call binaryP(P,i,AuxP,garbage)
    c. if (i%2=0) then
        i. local int j=0
        ii. from j=0 loop
            1. Parray[i][j] +=AuxP[j]
            2. Parray[i][j] ^=L[j]
            3. j+=1
        iii. until j=32
        iv. delocal int j=32
    d. else
        i. local int j=0
        ii. from j=0 loop
            1. Parray[i][j] +=AuxP[j]
            2. Parray[i][j] ^= R[j]
            3. j+=1
        iii. until j=32
        iv. delocal int j=32
    e. fi i%2=0
    f. uncall binary(P,i,AuxP,garbage)
    g. delocal int AuxP[32]={0}
    h. i+=1
3. until i=18
4. delocal int i=18

Η πιο πάνω διαδικασία υλοποιεί τα βήματα που έχουν αναφερθεί πιο πάνω και μεταβάλλει την τιμή του πίνακα με βάση το κλειδί. Σε κάθε επανάληψη, αν ο δείκτης είναι άρτιος τότε χρησιμοποιούμε το πρώτο μισό του κλειδιού (L) και εκτελούμε πράξη XOR με την δυαδική τιμή του πίνακα P (βήμα 2.b, κλήση συνάρτησης binaryP). Όταν η τιμή του δείκτη είναι περιττή τότε χρησιμοποιούμε το δεύτερο μισό (R).

Η επεξεργασία των τιμών των 4 Sboxes γίνεται με τον ίδιο τρόπο με τον πίνακα P. Οι αρχικές τιμές είναι όπως και στην προηγούμενη συνάρτηση. Η συνάρτηση που ενημερώνει τα Sboxes είναι η initialize_Sbox και παίρνει ως παραμέτρους τον πίνακα Sbox0 που περιέχει τα αρχικά Sboxes, τον πίνακα Sbox που θα αποθηκεύσει τις νέες τιμές στις ίδιες θέσεις, τον αλλαγμένο πίνακα P.

Πιο κάτω δίνεται η συνάρτηση F που εκτελείται κατά τη διάρκεια των 16 επαναλήψεων του δικτύου Feistel. Η συνάρτηση F παίρνει ως παραμέτρους τα Sboxes (S0,S1,S2,S3), τις τιμές a,b,c,d που προκύπτουν από το διαχωρισμό του L σε 4 τμήματα των 8bit και τη βοηθητική στοίβα garbage για να αποθηκευτούν τα επιπλέον σκουπίδια.

1. local int res1[32]={0}
2. call addition(S0,a,S1,b,res1,garbage)
3. local int res2[32]={0}
4. local int i=0
5. call aux_F(res1,res2,S2,c)
6. local int i=31
7. from i=31 loop
    a. F[i] +=((res2[i] +S3[d][i]) %2) % 2^{32}
    b. i-=1
8. until i=-1
9. delocal int i=-1
10. uncall aux_F(res1,res2,S2,c)
11. delocal int res2[32]={0}
12. uncall addition(S0,a,S1,b,res1,garbage)
13. delocal int res1[32]={0}

Η βοηθητική συνάρτηση addition :

1. local int i=31
2. from i=31 loop
    a. Res[i] +=(S[a][i] + B[b][i]) %2
    b. i-=1
3. until i=-1
4. delocal int i=-1

Η βοηθητική συνάρτηση aux_F :

1. local int i=0
2. from i=0 loop
    a. res2[i]+=res1[i]
    b. res2[i]^=S2[c][i]
    c. i+=1
3. until i=32
4. delocal int i=32

Η συνάρτηση f αποτελεί την κύρια πράξη επεξεργασίας μεταξύ των δεδομένων, ούτως ώστε το τελικό κρυπτογραφημένο κείμενο να μην μοιάζει καθόλου με την αρχική είσοδο. Η πράξη η οποία εκτελείται σ' αυτή την συνάρτηση είναι F = ((S1[a] + S2[b] mod 2^{32}) XOR S3[c]) + S4[d] mod 2^{32}. Η συνάρτηση είναι υλοποιημένη με τη μέθοδο compute-uncompute για διαγραφή των σκουπιδιών σε κάθε βήμα.

Όπως και στον αλγόριθμο DES (Κεφάλαιο 4.1), ο αλγόριθμος Blowfish εκτελεί μια σειρά από επαναλήψεις και σε κάθε επανάληψη γίνεται η προεργασία, η κλήση και η τελική επεξεργασία των αποτελεσμάτων της συνάρτηση F. Η συνάρτηση innerRounds δέχεται ως ορίσματα τα τέσσερα Sboxes, τον πίνακα P, τη στοίβα garbage και τους πίνακες L,R οι οποίοι θα τροποποιηθούν κατά την εκτέλεση της συνάρτησης.

1. local int i=0
2. from i=0 loop

a. local int j=0
b. from j =0 loop
    i. L[j]^=P[i][j]
    ii. j+=1
c. until j=32
d. delocal int j=32
e. local int F[32]={0}
f. local int a= Στοιχεία πίνακα L στο διάστημα[0,7] σε «δεκαδική» μορφή
g. local int b= Στοιχεία πίνακα L στο διάστημα[8,15] σε «δεκαδική» μορφή
h. local int c= Στοιχεία πίνακα L στο διάστημα[16,23] σε «δεκαδική» μορφή
i. local int d = Στοιχεία πίνακα L στο διάστημα[24,31] σε «δεκαδική» μορφή
j. Ανάστροφη κλήση της συνάρτησης DecimalBinary για κάθε ένα από τα a,b,c,d
k. Κλήση συνάρτησης f
l. local int j=0
m. from j=0 loop
    i. R[j] ^=F[j]
    ii. j+=1
n. until j=32
o. delocal int j=32
p. Ανάστροφη κλήση συνάρτησης f
q. Κλήση της συνάρτησης DecimalBinary για κάθε ένα από τα a,b,c,d
r. delocal int a= Στοιχεία πίνακα L στο διάστημα[0,7] σε «δεκαδική» μορφή
s. delocal int b= Στοιχεία πίνακα L στο διάστημα[8,15] σε «δεκαδική» μορφή
t. delocal int c= Στοιχεία πίνακα L στο διάστημα[16,23] σε «δεκαδική» μορφή
u. delocal int d = Στοιχεία πίνακα L στο διάστημα[24,31] σε «δεκαδική» μορφή
v. local int j=0
w. from j=0 loop
    i. R[j] <=> L[j]
    ii. j+=
x. until j=32
y. delocal int j=32
z. i+=1

3. until i=16
4. delocal int i=16
5. local int j=0
6. from j=0 loop
    a. R[j] <=> L[j]
    b. R[j] ^=P[16][j]
    c. L[j] ^= P[17][j]
    d. j+=1
7. until j=32
8. delocal int j=32

Αρχικά, στο βήμα 2.a – 2.d εκτελούμε πράξη XOR μεταξύ του πρώτου μισού της εισόδου (L) και της γραμμής του πίνακα P για την αντίστοιχη επανάληψη. Στο βήμα 2.f – 2.i χωρίζουμε τον πίνακα L σε 4 τμήματα των 8bits, μετατρέπουμε την δυαδική αναπαράσταση του αριθμού σε δεκαδική της ίδιας μορφής για σκοπούς υπολογισμού, για παράδειγμα $(101)_2 \rightarrow (101)_{10}$ ούτως ώστε η ανάστροφη κλήση της συνάρτησης DecimalBinary να μας επιστρέψει την πραγματική, δυαδική, αναπαράσταση του κάθε αριθμού. Οι τιμές, όπως είναι αναμενόμενο, βρίσκονται στο διάστημα [0,255]. Στη συνέχεια, γίνεται κλήση της συνάρτησης f και εκτελούμε πράξη XOR μεταξύ του πίνακα R και της εξόδου της συνάρτησης f. Το βήμα 2.p - 2.u αποτελεί αναίρεση των προηγούμενων βημάτων για σκοπούς αναστρεψιμότητας. Τέλος, ο βρόγχος τερματίζει με ανταλλαγή των στοιχείων των πινάκων L και R. Με το πέρας των 16 γύρων του αλγορίθμου, αναιρούμε την τελευταία ανταλλαγή που έγινε και εκτελούμε πράξη XOR μεταξύ των πινάκων R, L με τις γραμμές 17 και 18 του πίνακα P αντίστοιχα.

Στη συνέχεια εκτελείται η συνάρτηση rounds που δέχεται ως παραμέτρους μια είσοδο σε 64bits στον πίνακα input64, τον πίνακα P, τα τέσσερα Sboxes, τη στοίβα garbage και τον πίνακα encrypted που θα χρησιμοποιηθεί ως η τιμή εξόδου.

1. local int L[32]={0}
2. local int R[32]={0}
3. call createLR(L,R,input64)
4. call innerRounds(P,Sbox0,Sbox1,Sbox2,Sbox3,garbage,L,R)
5. local int i=0

6. from i=0 loop
    a. if (i<32) then
        i. encrypted[i] +=L[i]
    b. else
        i. encrypted[i] +=R[i-32]
    c. fi i<32
    d. i+=1
7. until i=64
8. delocal int i=64
9. uncall innerRounds(P,Sbox0,Sbox1,Sbox2,Sbox3,garbage,L,R)
10. uncall createLR(L,R,input64)
11. delocal int R[32]={0}
12. delocal int L[32]={0}

Αρχικά δημιουργούνται τα L και R. Η κλήση και ανάστροφη κλήση της συνάρτησης innerRounds στα βήματα 4 και 9 αντίστοιχα αποτελεί υλοποίηση των 16 γύρων του δικτύου feistel.

Ο αλγόριθμος Blowfish έχει την εξής ιδιαιτερότητα η οποία ενισχύει την ασφάλεια του και καθιστά εξαιρετικά πολύπλοκη ως αδύνατη την αποκρυπτογράφηση του από τρίτους που πιθανών προσπαθούν να κλέψουν πληροφορίες από μια συνομιλία. Εκτελεί την κρυπτογράφηση σε δύο μέρη.

Αφού έχουν δοθεί τα επιμέρους κομμάτια του αναστρέψιμου αλγορίθμου, πιο κάτω θα δοθεί η πλήρης ακολουθία βημάτων:

1. call convert_to64bits(key, K64, garbage)
2. call createLR(L, R, K64)
3. call initialize_P(P, P_array, K64, garbage, L,R)
4. call initialize_Sbox(Sbox0, Sbox, K64, garbage, L,R)
5. local int count =0
6. from count = 0 loop
    a. rounds(zero_string, P,Sbox, garbage, encrypted)

   b. createLR(P[count],P[count+1], encrypted)
   c. count+=1
7. until count = 9
8. count -= 9
9. from count =0 loop
   a. rounds(zero_string, P,Sbox, garbage, encrypted)
   b. createLR(Sbox[count],Sbox[count+1], encrypted)
   c. count+=1
10. until count = 512
11. delocal int count = 512
12. rounds(input64,P, Sbox, garbage , encrypted)

4.2.3 Η υλοποίηση στη Janus

Η υλοποίηση στη Janus αποτελεί μια πιο απλοποιημένη εκδοχή του πιο πάνω αλγορίθμου. Στην κανονική εκδοχή του αλγορίθμου μετά την κρυπτογράφηση της μηδενικής εισόδου, η αλλαγή των P και Sbox γίνεται ανά 2 και μετά επαναλαμβάνεται η κρυπτογράφηση και η μεταβολή των στοιχείων, σύνολο 521 επαναλήψεις. Στην αναστρέψιμη όμως εκδοχή θα έπρεπε να γίνουν 521 διαφορετικές κλήσεις με νέες μεταβλητές. Για το λόγο αυτό, η υλοποίηση στη Janus εκτελεί την κρυπτογράφηση μετά από μόνο δύο αλλαγές ολόκληρων των πινάκων P και Sbox. Αυτό σημαίνει ότι αρχικά ενημερώνονται με τη πράξη XOR και το κλειδί όπως καθορίζει ο αλγόριθμος, μετά κρυπτογραφείται η μηδενική ακολουθία και μετά λειτουργεί ως το κλειδί για τον υπόλοιπο αλγόριθμο, ενημερώνοντας ξανά τους πίνακες με πράξη XOR και την κρυπτογραφημένη μηδενική ακολουθία. Τέλος, για λόγους μνήμης οι πίνακες Sbox και P μειώθηκαν στα 64 στοιχεία και οι μεταβλητές a,b,c,d που χρησιμοποιούνται στη συνάρτηση F παίρνουν πλέον τιμές μέχρι το 64.

Η πλήρης υλοποίηση του αλγορίθμου σε γλώσσα Janus βρίσκεται στο παράρτημα Β.

4.2.4 Ανάλυση αλγορίθμου

Οι πράξεις που ορίζονται από τον αλγόριθμο αποτελούν μόνο πράξεις XOR και πρόσθεσης οι οποίες έχουν γραμμική χρονική πολυπλοκότητα αφού εκτελούνται πάνω σε όλη τη είσοδο και το κλειδί. Ο αλγόριθμος εκτελείται σε κάθε τμήμα 64bits της εισόδου επομένως η τελική χρονική πολυπλοκότητα του αλγορίθμου είναι της τάξης Θ(N), όπου N το πλήθος των τμημάτων 64bits στην είσοδο. Η μνήμη που χρειάζεται ο αλγόριθμος έγκειται στις δομές P και Sbox η οποία είναι σταθερή. Στην αναστρέψιμη εκδοχή χρησιμοποιούνται δύο πίνακες για το καθένα και αυτό συμβαίνει για τις δύο φάσεις της κρυπτογράφησης. Επομένως η χωρική πολυπλοκότητα της συνάρτησης είναι της τάξης του Θ(N).

Στον Πίνακα 4.10 φαίνονται οι μετρήσεις που υπολογίστηκαν για σκοπούς πειραματικής αξιολόγησης της χρονικής πολυπλοκότητας του αλγορίθμου. Οι μετρήσεις επικυρώνουν το γεγονός ότι η αναστρέψιμη υλοποίηση του αλγορίθμου έχει γραμμική χρονική πολυπλοκότητα. Όλες οι μετρήσεις είναι σε δευτερόλεπτα. . Δίνονται δέκα μετρήσεις για κάθε είσοδο με σκοπό να έχουμε πιο ορθό υπολογισμό της χρονική πολυπλοκότητας παίρνοντας το μέσο όρο.

64bits	128bits	172bits	256bits	320bits	384bits
84,528	158,933	274,335	373,42	409,779	643,221
81,373	159,137	262,757	329,623	476,748	527,763
80,476	160,195	253,602	307,28	436,113	543,53
82,417	179,987	285,436	336,652	489,921	610,131
82,002	178,847	245,988	351,738	460,405	517,083
81,49	173,244	277,542	351,581	420,484	592,653
80,402	172,96	279,815	326,77	386,547	553,722
80,528	172,764	237,785	328,168	491,969	501,177
81,976	171,166	256,5	325,093	535,818	490,775
81,933	191,913	269,78	343,209	633,638	568,229
81,7125	171,9146	264,354	337,3534	474,1422	554,8284

Πίνακας 4.10 Πειραματική αξιολόγηση της χρονική πολυπλοκότητας

Όσο αφορά την εκτέλεση των 4096bits, χρειάζεται περίπου 90 λεπτά (5430,9 δευτερόλεπτα) η ολοκλήρωση της εκτέλεσης του.

Η υλοποίηση του αλγορίθμου είναι πιστή καθώς κρατά σταθερή την πολυπλοκότητα του αλγορίθμου και είναι υγιής καθώς δεν χρειάζεται επιπλέον σκουπίδια με το πέρας της

εκτέλεσης. Επομένως η συνάρτηση κρατά τη συνάρτηση αναπαράστασης των σκουπιδιών στο ελάχιστο και εξ ‘ου η «υγιεινή» της υλοποίησης.

Κεφάλαιο 5

Συμπεράσματα

5.1 Κόστος της αναστρεψιμότητας

Η διπλωματική αυτή ασχολήθηκε με τη ανάπτυξη αναστρέψιμων αλγορίθμων σε θέματα κωδικοποίησης και κρυπτογράφησης αλλά και με την αναστρεψιμότητα γενικότερα. Η μελέτη των αναστρέψιμων αλγορίθμων είναι μόνο μια από τις πτυχές του φάσματος του αναστρέψιμου υπολογισμού. Η ανάπτυξη όμως των αλγορίθμων αυτών και η υλοποίηση τους σε μια αναστρέψιμη γλώσσα προγραμματισμού προσδίδει νόημα στην ανάπτυξη ενός αναστρέψιμου υπολογιστή. Θεωρώ ότι τα αποτελέσματα αυτής της διπλωματικής εργασίας είναι θετικά όσο αφορά την ανάπτυξη αναστρέψιμων αλγορίθμων και προγραμμάτων. Στα πλαίσια της διπλωματικής εργασίας φάνηκε ότι η υλοποίηση των αναστρέψιμων αλγορίθμων είναι εφικτή. Είναι σημαντικό να τονίσουμε ότι αυτό γίνεται με κόστος μνήμης. Είναι αδύνατο να μην δημιουργηθούν καθόλου σκουπίδια κατά την υλοποίηση ενός αναστρέψιμου αλγορίθμου όμως μια καλή σχεδίαση του αλγορίθμου θα κρατήσει το επίπεδο των σκουπιδιών στο ελάχιστο.

Η δυσκολία λοιπόν βρίσκεται στην ανάπτυξη του αλγορίθμου με τέτοιο τρόπο, ώστε να διατηρούνται οι αυθεντικές πολυπλοκότητες (χρονική και χωρική). Σε μερικά προβλήματα η υλοποίηση είναι τετριμμένη, όμως υπάρχουν προβλήματα τα οποία χρειάζονται εξαιρετική κατανόηση των βασικών έννοιων του αναστρέψιμου υπολογισμού αλλά και ανεπτυγμένη αλγοριθμική σκέψη για δημιουργία αλγορίθμου που να εκτελεί βήματα με την ίδια σειρά όπως ο αυθεντικός και ταυτόχρονα να είναι αναστρέψιμος.

Όπως αναφέρθηκε στο Κεφάλαιο 2, χρειάζεται ακόμα πολλή δουλειά μέχρι την επίτευξη πλήρης αναστρεψιμότητας σε υπολογιστικές μηχανές. Η χρήση μηχανισμών για ανακύκλωση της ενέργειας που χρησιμοποιείται και η ανακάλυψη υλικού τέτοιου ώστε να μην επιτρέπει την διαρροή θερμότητας είναι μόνο κάποιες από τις προκλήσεις που πρέπει να αντιμετωπιστούν προτού μπορέσουμε να προχωρήσουμε στην ανάπτυξη ενός αναστρέψιμου υπολογιστή. Αντίθετα, η δημιουργία τέτοιων υλικών είναι πιθανό να αναιρέσει την πρόοδο που έχει επιτευχθεί όσο αφορά το μέγεθος των επεξεργαστών και κατ' επέκταση ολόκληρου του υπολογιστή, καθώς για την μη διαρροή θερμότητας κατά τη λειτουργία του υπολογιστή ίσως απαιτείται περισσότερος όγκος στα συστατικά του υπολογιστή. Ζώντας λοιπόν στην εποχή των έξυπνων φορετών συσκευών και των αεναώς μικρότερων υπολογιστών, πόσα από αυτά είμαστε έτοιμοι να θυσιάσουμε για την επίτευξη της αναστρεψιμότητας; Είναι ασφαλής η εικασία ότι αν στο εγγύς μέλλον δημιουργηθεί ένας πραγματικά και ολοκληρωτικά αναστρέψιμος υπολογιστής δεν θα είναι ευρείας χρήσης για πολλά χρόνια ακόμα. Ο αναστρέψιμος υπολογιστής θα βρει εφαρμογές σε μηχανές οι οποίες παρουσιάζουν μεγάλη κατανάλωση ενέργειας και κατ' επέκταση μεγάλη διαρροή θερμότητας. Όσο αφορά τους προσωπικούς υπολογιστές δεν κρίνεται αναγκαία η αντικατάσταση τους με αναστρέψιμους, τουλάχιστον με τα σημερινά δεδομένα. Παρ' όλα αυτά η πραγματοποίηση του στόχου θα προσδώσει μια νέα διάσταση στην έννοια «υπολογισμός».

Στο Κεφάλαιο 3 παρουσιάστηκαν πιστές υλοποιήσεις των αλγορίθμων κωδικοποίησης. Αρχικά, το συμπέρασμα από την μετατροπή του αλγορίθμου Leveling είναι ότι όταν οι αλγόριθμοι μας χρησιμοποιούν δομές τις οποίες το μέγεθος τους υπολογίζεται κατά την εκτέλεση του αλγορίθμου, παρατηρείται σπατάλη μνήμης. Αυτό προκύπτει από το γεγονός ότι ο υπολογισμός των επιπέδων εξαρτάται από το μέγεθος της εισόδου και τις πιθανότητες εμφάνισης κάθε στοιχείου του κειμένου. Σε ένα αναστρέψιμο περιβάλλον η γνώση εκ των

προτέρων αυτών των τιμών είναι απαραίτητη επομένως πρέπει να εφαρμοστεί μια προσεγγιστική λύση για την επίλυση του προβλήματος αυτού. Στη συγκεκριμένη περίπτωση η χρήση δυαδικού συστήματος και η εκθετική αύξηση τάξεως του 2 του αριθμού των στοιχείων σε κάθε επίπεδο ήταν βοηθητική ώστε να δώσουμε ένα άνω ασυμπτωτικό φράγμα ίσο με το μισό του μεγέθους της εισόδου. Επιπρόσθετα, η υλοποίηση που παρουσιάστηκε είναι προσαρμοσμένη στα δεδομένα της γλώσσας Janus και επομένως στην μη ύπαρξη αριθμών κινητής υποδιαστολής. Έτσι οι μετατροπές που παρουσιάστηκαν για να μπορέσει να υλοποιηθεί ο αλγόριθμος προσθέτουν κόστος μνήμης το οποίο καθιστά την υλοποίηση μη υγιεινή.

Ο αλγόριθμος Burrows-Wheeler αποτελεί αναστρέψιμη μορφή μετασχηματισμού ενός κειμένου. Επομένως, διαπιστώθηκε ότι σε αλγόριθμους που εκτελούν γραμμικό μετασχηματισμό των δεδομένων στη μη αναστρέψιμης εκδοχής τους, είναι εύκολο να υλοποιήσουμε την αναστρέψιμη εκδοχή του. Η αναστρέψιμη εκδοχή του αλγορίθμου εκτελεί την κωδικοποίηση στο εμπρόσθιο πέρασμα και την αποκωδικοποίηση στο ανάστροφο πέρασμα, συγχωνεύοντας έτσι τους δύο αλγόριθμους. Η αποκωδικοποίηση του αυθεντικού αλγορίθμου αποτελεί αναστροφή των βημάτων της κωδικοποίησης μεν, είναι πιο πολύπλοκη δε, πράγμα που καθιστά την αναστρέψιμη υλοποίηση πολύ βοηθητική αφού χρησιμοποιεί τον ίδιο αλγόριθμο για τις δύο πράξεις. Η υλοποίηση του Burrows-Wheeler αποτελεί πιστή και υγιεινή προσομοίωση του γνωστού αλγορίθμου.

Η υλοποίηση του αλγόριθμου κωδικοποίησης Huffman, έδειξε την ανάγκη ύπαρξης πιο σύνθετων δομών σε αναστρέψιμο επίπεδο. Η χρήση δεικτών βασίζεται κατά κόρον στην πράξη της ανάθεσης αφού ανατίθεται η θέση μνήμης στην οποία πρέπει να δείξει. Επομένως, η δημιουργία πιο σύνθετων δομών πρέπει να γίνει με χρήση διαφορετικών τεχνασμάτων τα οποία όποια να βασίζονται σε πράξεις οι οποίες να είναι αναστρέψιμες (όπως για παράδειγμα η δημιουργία ουράς με χρήση 2 στοίβων). Στην υλοποίηση του αλγορίθμου γίνεται χρήση λίστας γειτνίασης για την αναπαράσταση του γράφου το οποίο προσδίδει κόστος τόσο σε μνήμη όσο και σε χρόνο (όχι ικανό να ξεπεράσει την ασυμπτωτική χρονική πολυπλοκότητα). Η υλοποίηση είναι πιστή. Αν η υλοποίηση χρειάζεται αλλαγές για να μπορεί να κατηγοριοποιηθεί ως υγιής επαφίεται στην υλοποίηση των δομών σε αναστρέψιμο επίπεδο.

Στα πλαίσια του Κεφαλαίου 4 παρατηρήθηκε ότι οι αλγόριθμοι κρυπτογράφησης που βασίζονται στην λογική του δικτύου Feisteleίναι αναστρέψιμοι εκ φύσεως. Οι αλγόριθμοι εκτελούν τις ίδιες πράξεις επαναληπτικά πάνω στα δεδομένα και μετατρέπουν επιτόπου τα κλειδιά και όλες τις βοηθητικές δομές (Sbox). Όπως έχει ήδη αναφερθεί, η πράξη της ανάθεσης δεν είναι αναστρέψιμη και επομένως για τις αλλαγές αυτές χρειάστηκε επιπλέον μνήμη. Ο αλγόριθμος Blowfish δεν μπορεί να υλοποιηθεί στην παρούσα φάση της γλώσσας Janus. Η μετατροπή των κλειδιών απαιτεί 521 επαναλήψεις συνολικά επομένως και τις ανάλογες επιπλέον μεταβλητές (σε μορφή πίνακα). Οι υλοποιήσεις δεν παύουν να είναι πιστές και υγιείς όμως αφού η ανάγκη για επιπλέον μνήμη είναι απαραίτητη. Έτσι η συνάρτηση που ορίζει τα σκουπίδια παραμένει ελάχιστη.

Στο Κεφάλαιο 4, οι υλοποιήσεις των αλγορίθμων έγιναν χρησιμοποιώντας τη μέθοδο compute-uncompute. Η μέθοδος υλοποιείται ως εξής: κάθε βήμα του αλγορίθμου υλοποιείται σε ξεχωριστή συνάρτηση. Κάθε συνάρτηση σαν πρώτο βήμα καλεί τη συνάρτηση που υλοποιεί το προηγούμενο βήμα, στη συνέχεια εκτελεί την πράξη για την οποία είναι υπεύθυνη και τέλος, εκτελεί ανάστροφη κλήση της συνάρτησης που υλοποιεί το προηγούμενο βήμα για αποδέσμευση των σκουπιδιών και όλης της μνήμης που χρειάστηκε κατά την εκτέλεση. Αυτή η μέθοδος εξοικονομεί μνήμη όσο αφορά τη μνήμη που παραμένει σε χρήση με το πέρας της συνάρτησης. Επομένως, συνίσταται η χρήση της μεθόδου αυτής όταν ένας αλγόριθμος είναι αρθρωτός και εκτελεί μια ακολουθία βημάτων στην οποία η είσοδος σε κάθε βήμα είναι η έξοδος που προκύπτει από το προηγούμενο βήμα και οι τιμές αυτές δεν χρησιμοποιούνται πουθενά αλλού στο σύστημα. Άρα σε αλγόριθμους κρυπτογράφησης ή άλλους αλγόριθμους που εκτελούν μετασχηματισμό πάνω σε ένα κομμάτι δεδομένων κατ' επανάληψη, η χρήση της μεθόδου αυτής θα συμβάλει στη μείωση των σκουπιδιών.

Συνοψίζοντας τα πιο πάνω, είναι λογικό συμπέρασμα ότι όσο πιο πολύπλοκες δομές χρησιμοποιεί μια μη αναστρέψιμη υλοποίηση αλγορίθμου και όσο πιο πολλές φορές αλλάζουν επί τόπου οι τιμές μιας μεταβλητής (ή δομής όπως πίνακα) τόσο πιο πολλή μνήμη χρειάζεται για μετατραπεί σε αναστρέψιμη. Η επιπλέον μνήμη, τα σκουπίδια, είναι ρητά συνδεδεμένα με τις πιο πάνω πράξεις. Επιπρόσθετα, στο παρόν επίπεδο, δεν είναι δυνατή η υλοποίηση όλων των αλγορίθμων σε αναστρέψιμους. Χρειάζεται ανάπτυξη της αλγοριθμικής σκέψης καθώς και

νέων μεθόδων σχεδίασης αλγορίθμων που να λαμβάνουν υπόψη και τη δυνατότητα αναστροφής για να μπορέσουμε να προσεγγίσουμε λύσεις για όλα τα προβλήματα.

5.2 Εισηγήσεις για βελτίωση της γλώσσας

Η γλώσσα Janus αποτελεί πανίσχυρο εργαλείο για την ανάπτυξη του αναστρέψιμου υπολογισμού. Δεν παύει όμως να είναι πολύ περιοριστική όσο αφορά της λειτουργίες που προσφέρει. Αρχικά, η ύπαρξη μόνο μεταβλητών τύπου integer είναι τεράστιο μειονέκτημα. Είναι προφανές ότι δεν έχουν εισαχθεί οι τύποι real, float, double στη γλώσσα για λόγους ακρίβειας διότι το αποτέλεσμα τις πράξης 1/3= 0,33 ή 0,333 ή 0,33334 έχει πραγματική σημασία όταν μας ενδιαφέρει η αποθήκευση κάθε πληροφορίας του προγράμματος, αλλά αυτό δεν σημαίνει ότι η εισαγωγή τους είναι αδύνατη. Με την εισαγωγή αριθμών κινητής υποδιαστολής στη γλώσσα, καθίσταται δυνατή και η εισαγωγή νεών τελεστών όπως *= και /= που θα βοηθήσουν τις αριθμητικές πράξεις του πολλαπλασιασμού και διαίρεσης αντίστοιχα. Δευτερεύων ανάγκη, όσο αφορά τους τύπους δεδομένων, είναι η εισαγωγή συμβολοσειρών και χαρακτήρων. Η προσθήκη τους θα διευρύνει τις δυνατότητες της γλώσσας σε επίπεδα όπου θα μπορεί να συναγωνιστεί τις υπόλοιπες, πιο σύγχρονες γλώσσες προγραμματισμού.

Επιπρόσθετα, κρίνεται αναγκαία η ενσωμάτωση πιο σύνθετων δομών δεδομένων. Για παράδειγμα η υλοποίηση της ουράς μπορεί να γίνει με χρήση δύο στοιβών και να ενσωματωθούν στο λεξιλόγιο της γλώσσας οι λέξεις κλειδιά enqueue και dequeue που να εκτελούν και τις αντίστοιχες πράξεις. Ακόμη, συνιστάται η εισαγωγή δομών βασισμένων σε γράφους και κατ’ επέκταση δέντρα, για πλήρη προγραμματιστική κάλυψη.

Κατά την εκτέλεση των προγραμμάτων διαπιστώθηκαν τα υπολογιστικά όρια της γλώσσας Janus όσο αφορά τις βελτιστοποιήσεις που υπάρχουν όσο αφορά την κλήση συναρτήσεων και το πέρασμα των παραμέτρων. Εκτελώντας την εκδοχή του κώδικα Blowfish με μήκος πινάκων 256 πάνω σε ένα μέτριου βεληνεκούς υπολογιστή (6Gb RAM, Intel Core i3 3^{ης} γενιάς, CPU 2.6 GHz), χρειάστηκαν περίπου 8 ώρες ώστε ο υπολογιστής να αναγκαστεί να σταματήσει την εκτέλεση λόγω έλλειψης κύριας μνήμης. Επομένως χρειάζεται βελτιστοποίηση στο πέρασμα των παραμέτρων, καθώς μεγάλοι πίνακες προκαλούν σοβαρό πρόβλημα στις υπολογιστικές δυνατότητες της γλώσσας.

Τέλος, ο λεκτικός αναλυτής της γλώσσας πρέπει να προσαρμοστεί έτσι ώστε να επιτρέπει την δήλωση μεταβλητών σε όλο το εύρος του προγράμματος. Αυτό προκύπτει από την ανάγκη δήλωσης μεταβλητών εξόδου με μέγεθος το οποίο πρέπει να υπολογιστεί στη φάση εκτέλεσης του προγράμματος.

5.3 Μελλοντική εργασία

Έχει αναφερθεί αρκετές φορές μέχρι στιγμής ότι η έρευνα επικεντρώθηκε στην ανάπτυξη του υλικού για την υποστήριξη της αναστρεψιμότητας. Σε αντίθεση, από την στιγμή της σύλληψης της έννοιας της αναστρεψιμότητας μέχρι και σήμερα, η ανάπτυξη αναστρέψιμων αλγορίθμων δεν ήταν ποτέ στο επίκεντρο. Κρίνεται λοιπόν αναγκαία η πρόοδος όσο αφορά την ανάπτυξη αναστρέψιμων αλγορίθμων και την υλοποίηση τους στη γλώσσα Janus. Μέχρι στιγμής έχουν μελετηθεί οικογένειες αλγορίθμων διάσχισης γράφων, ταξινόμησης, κωδικοποίησης και κρυπτογραφίας οι οποίοι δεν χρειάζονται πράξεις κινητής υποδιαστολής. Επομένως με την εισαγωγή περισσότερων τύπων δεδομένων στη γλώσσα θα μπορούσαν να μελετηθούν σε μεγαλύτερο βάθος οι αλγόριθμοι κωδικοποίησης και κρυπτογραφίας. Επιπρόσθετα με την εισαγωγή νέων δομών δεδομένων καθίσταται δυνατή η μελέτη αλγορίθμων απόκρυψης πληροφορίων σε εικόνες και αλγόριθμοι επεξεργασίας εικόνας. Η ανάγκη για αναστρεψιμότητα των δύο ομάδων αλγορίθμων είναι μεγάλη καθώς στους συμβατικούς υπολογιστές γίνεται χρήση επιπλέον μηχανισμών για την επίτευξη τους.

Βιβλιογραφία

[1] H. B. Axelsen and R. Glück, “What do reversible programs compute?”, In proceedings of FOSSACS ‘11, pp. 42-56, 2011.

[2] H. B. Axelsen and T. Yokoyama, “Programming techniques for reversible comparison sorts”, In proceedings of APLAS ‘15, pp. 407-426, 2015.

[3] J. J. Alcubilla, “Leveling: An efficient VLC for lossless data compression”, International Journal of Signal Processing, Image Processing and Pattern Recognition, vol 9, pp. 199-218, 2016.

[4] C. H. Bennett, “Time/Space trade-offs for reversible computation”, SIAM J. Comput., Vol 18, No. 4, pp. 766-776, 1989.

[5] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm”, SRC 124, 1994.

[6] M. P. Frank, “Approaching the physical limits of computing”, In proceedings of ISMVL ‘05, pp. 168-185, 2005.

[7] D. A. Huffman, “A method for the construction of minimum-redundancy codes”, In proceedings of the I.R.E, pp.1098-1101, 1952.

[8] C. Lutz and H. Derby, “Letter to Landauer”, 1986

[9] P. Mahjan and A. Sachdeva, “A study for encryption algorithms AES, DES and RSA for security, Global Inc., Vol 13, Issue 15, 2013.

[10] S. V. Nohr, “Reversible graph algorithms”, Bachelor Thesis, University of Copenhagen, 2015.

[11] K. S.Perumall, “Introduction to Reversible Computing”, Chapman & Halls/CRC Computation Science Series, 2014

[12] B. Schneier, “Fast software encryption”, In proceedings of Cambridge Security Workshop, pp.161-204, 1994.

[13] T. Yokoyama, H. B. Axelsen and R. Glück, “Principles of a reversible programming language”, In proceedings of Computing Frontiers ‘08, pp.43-54, 2008.

[14] T. Yokoyama, “Reversible computation and reversible programming languages”, Electr. Notes Theor. Comput. Sci. Vol 253, No. 6, pp.71-81, 2010

Παράρτημα Α

Σε αυτό το παράρτημα δίνεται ο κώδικας σε γλώσσα Janus για την υλοποίηση των αλγορίθμων κωδικοποίησης Leveling, Burrows-Wheeler.

Υλοποίηση αλγορίθμου Leveling:

```
/*
This function sorts a 2d matrix based on the its first column using
bubble sort. Array d is used as garbage for storing the times an
element has switched spots.
*/
procedure bsortD(int a[][], int n , int d[])
   1. local int i = 0
         a. from i = 0
         b. loop local int j = n-1
              i.  from j = n-1
             ii.  loop if a[0][j] > a[0][j-1] then
                     1. a[0][j-1] <=> a[0][j]
                     2. a[1][j-1] <=> a[1][j]
                     3. d[j-i-1] +=1
                     4. fi d[j-i-1] = i+1
                     5. j -= 1
            iii.  until j = i
             iv.  delocal int j = i
              v.  i+=1
         c. until i = n
         d. delocal int i = n

/*
This function initializes array[1] with the "letters of the alphabet"
represented as the number of their place in the alphabet. Then,
calculates the number each character is seen in the input and stores
it in the corresponding row.
*/
```

```
procedure generate_probabilities(int input_size, int input[], int
array[][],int sizee)
   1. local int i = 0
   2. from i = 0
   3. loop
        a. array[1][i] += i+1
        b. i+=1
   4. until i = sizee
   5. delocal int i = size
   6. local int x = 0
   7. from x = 0
   8. loop
        a. if input[x] != 0 then
             i. array[0][input[x]-1] += 1
        b. fi input[x] != 0
        c. x+=1
   9. until x = input_size
  10. delocal int x = input_size

/*
This function is responsible for calculating the number of elements
whose probability of occurrence "fits" into 1/2. This is done using
the counter for each element calculated using generate_probabilities
function.
*/
procedure calc_ini_len(int array[][],int sum, int input_size,int
ini_len)
   1. local int aggr =0
   2. local int i = 0
   3. from aggr= 0
   4. loop
        a. aggr +=array[0][i]
        b. if (aggr <= sum /2) then
             i. ini_len +=1
        c. fi (aggr <= sum/2)
        d. i+=1
```

```
   5. until (i > input_size)
   6. delocal int i = input_size +1
   7. delocal int aggr = sum

/*
This function matches the initial_length calculated in calc_ini_len
with its corresponding value which is log(N) +1, but Janus doesn't
support logarithmic functions so the values are static.
*/
procedure match_corresponding_ini_len(int help, int ini_len,int
input_size )
   1. local int symbols[10][2]=
      {{1,1},{2,2},{3,2},{4,3},{5,3},{6,3},{7,3},{8,4},{9,4}, {10,4}}
   2. help += ini_len
   3. local int i = 0
   4. from i = 0
   5. loop
         a. if ( ini_len = symbols[i][0] ) then
              i. local int val = 0
             ii. val += symbols[i][1]
            iii. help <=> val
             iv. delocal int val = ini_len
         b. fi ini_len = symbols[i][0]
         c. i+=1
   6. until i = input_size
   7. ini_len <=> help
   8. delocal int i = input_size
   9. delocal int symbols[10][2] =
      {{1,1},{2,2},{3,2},{4,3},{5,3},{6,3},{7,3},{8,4},{9,4},{10,4}}

/*
This function is responsible for calculating the reduced_code vector
for the program. Due to Janus limitations all conditions and operations
involving them are converted to integer use.
*/
procedure calc_reduced_code(int reduced_code[],int ini_len, int d, int
j, int modified[], int input_size, int sum,int diff,int array[][])
```

1. local int b=0
2. reduced_code[0] += ini_len
3. d +=ini_len
4. j+=1
5. local int i = 0
6. from i = 0
7. loop
    a. modified[i] += array[0][i] * d *2
    b. i+=1
8. until i = input_size
9. delocal int i = input_size
10. from b = 0 && j = 1
11. loop
    i. if ((modified[b] < sum && 2*modified[b]+modified[b+1] > 2*sum)|| modified[b] >=sum) then
    ii. reduced_code[j] +=1
    iii. modified[b+1] += modified[b] -sum
    iv. b+=1
    v. else
    vi. local int prev = d
    vii. d+= prev
    viii. delocal int prev = d/2
        1. local int i = 0
    ix. from i = 0
    x. loop
        1. local int mod = modified[i]
        2. modified[i] += mod
        3. delocal int mod = modified[i] /2
        4. i+=1
        5. until i = input_size
        6. delocal int i = input_size
    xi. j +=1
    xii. fi reduced_code[j] > 0
12. until b = diff
13. delocal int b = diff

```
/*
This function calculates the number 2^j which is the maximum number of
columns needed for matrix codes and letters for this particular input.
*/
procedure calc_count(int count , int j )
   1. local int i = 0
   2. from i = 0
   3. loop
         a. local int aux = count
         b. count+= aux
         c. delocal int aux = count / 2
         d. i+=1
   4. until i =j
   5. delocal int i = j

/*
This function creates codes matrix for leveling encoding
*/
procedure make_table(int codes[][], int input_size, int
reduced_code[], int count,int j)
   1. local int num = 0
   2. local int parts = 2
   3. local int i = 0
   4. from i = 0
   5. loop
         a. local int z = 0
         b. from z= 0
         c. loop
              i. local int k = 0
             ii. from k = 0
            iii. loop
             iv. codes[i][z*((count*input_size)/parts)+k] += num
                    1. k+=1
              v. until k = (count*input_size) / parts
             vi. delocal int k = (count*input_size) / parts
            vii. num ^= 1
```

```
          viii.  z+=1
        d. until z = parts
        e. delocal int z = parts
        f. local int aux  = parts
        g. parts+=aux
        h. delocal int aux = parts/2
        i. i+=1
   6. until i = j-1+reduced_code[0]
   7. delocal int i = j-1+reduced_code[0]
   8. delocal int parts = count*2
   9. delocal int num = 0

/*
This function creates the matrix letters for leveling encoding
*/
procedure make_letters(int letters[],int input_size,int array[][],int
different)
   1. local int i = 0
   2. from i=0
   3. loop
        a. if (array[0][i] != 0 ) then
             i.  different+=1
        b. fi array[0][i] != 0
        c. i+=1
   4. until i = input_size
   5. delocal int i = input_size
   6. local int g =0
   7. local int i = 0
   8. from i = 0
   9. loop
        a. local int k = 0
        b. from k = 0
        c. loop
             i.  letters[g] += array[1][i]
            ii.  k+=1
           iii.  g+=1
```

```
        d. until k = (array[0][i])* (size(letters)/input_size)
        e. delocal int k = (array[0][i])*(size(letters)/input_size)
        f. i+=1
  10. until i = different
  11. delocal int i = different
  12. delocal int g = size(letters)

/*
Reversible enqueue function simulating a queue using two stacks.
*/
procedure enqueue( int x , stack A)
   1. local stack B = nil
   2. if size(A) = 0 then
   3. push(x, A)
   4. else
   5. local int i = size(A) *2
   6. from i = size(A)*2
   7. loop
              i. local int y = 0
             ii. pop(y, A)
            iii. push(y,B)
             iv. delocal int y=0
              v. i-=1
   8. until i = size(B)
   9. delocal int i = size(B)
  10. push(x,B)
  11. local int i = size(B)*2
  12. from i = size(B)*2
  13. loop
        a. local int y = 0
        b. pop(y,B)
        c. push(y,A)
        d. delocal int y = 0
        e. i-=1
  14. until i = size(A)
  15. delocal int i = size(A)
```

```
  16. fi size(A) = 1
  17. delocal stack B = nil

/*
Janus implementation of the Leveling technique using two arrays, codes
and letters simulating the leveling encoding using only integers.
*/
procedure leveling(int array[][],int sum,int count,int j,int ini_len,
int input_size,int reduced_code[],stack aux, stack code)
        a. local int different = 0
        b. local stack starts = nil
   2. local stack finishs = nil
   3. local int codes[ini_len+j-1][sum*count]= {{0},{0},{0},{0}}
   4. local  int letters[count*sum] = {0}
   5. call make_table(codes,input_size,reduced_code,count,j)
   6. call make_letters(letters,input_size,array,different)
   7. local int counter = 0
   8. local int i = 0
   9. local int start = 0
  10. local int finish = 0
  11. from i=0
  12. loop
        a. finish+=start + array[0][i]*(size(letters)/input_size)
        b. call enqueue(start,starts)
        c. start+=finish
        d. finish -=1
        e. call enqueue(finish,finishs)
        f. i+=1
  13. until i = different
  14. delocal int finish = 0
  15. delocal int start = size(letters)
  16. delocal int i = different
  17. local int pos=1
  18. from size(starts) = different
  19. loop
        a. local int start = 0
```

```
    b. local int finish = 0
    c. pop(start,starts)
    d. pop(finish,finishs)
    e. local int sinolo =0
    f. local int miso =0
    g. sinolo += start+finish
    h. if ( sinolo % 2 = 0 ) then
    i. miso += sinolo /2
    j. else
         i. miso += sinolo /2 +1
    k. fi sinolo %2 = 0
    l. local int k =reduced_code[0]+ pos-2
    m. from k =reduced_code[0]+ pos-2
    n. loop
         i. push(codes[k][miso],code)
        ii. k-=1
    o. until k < 0
    p. delocal int k = -1
    q. push(letters[miso]*(-1),code)
    r. delocal int miso = sinolo/2 + (sinolo%2)
    s. delocal int sinolo = start + finish
    t. push(start,aux)
    u. push(finish,aux)
    v. delocal int finish = 0
    w. delocal int start =0
    x. counter+=1
    y. if ( counter >= reduced_code[pos]) then
         i. pos+=1
        ii. counter-= reduced_code[pos-1]
    z. fi  counter=0
20. until size(starts) =0
21. delocal int pos = j+1
22. delocal int counter = 0
23. uncall make_letters(letters,input_size,array,different)
24. uncall make_table(codes,input_size,reduced_code,count,j)
25. delocal int letters[count*sum] = {0}
```

```
26. delocal int codes[ini_len+j-1][sum*count]={{0},{0},{0},{0}}
27. delocal stack finishs = nil
28. delocal stack starts = nil
29. delocal int different = 0
```

```
/*
This function calculates the meaning code for every encoding showing that encodings come in a logical order in decimal which doesn't translate well in binary.
*/
procedure find_meaning(int meaning[],int reduced_code[],int b,int j)
   1. local int counter = 1
   2. local int index = 1
   3. from index= 1
   4. loop
         a. if index > 1 then
               i. meaning[counter]    +=    (meaning[counter-1]+1)    *
                  (meaning[counter-1]+1)
              ii. counter+=1
         b. fi index > 1
         c. local int i=1
         d. from i = 1
         e. loop
               i. meaning[counter] += meaning[counter-1]+1
              ii. counter+=1
             iii. i+=1
         f. until i = reduced_code[index]
         g. delocal int i = reduced_code[index]
         h. index+=1
   5. until index > j
   6. delocal int index = j+1
   7. delocal int counter = b
```

```
/*
This function counts the number of non zero elements in array, which
is the number of different elements in the input.
*/
procedure calc_diff(int diff,int array[][])
   1. from diff = 0
   2. loop
         a. diff+=1
   3. until array[0][diff] = 0
```

Υλοποίηση αλγορίθμου Burrows-Wheeler:

```
/*
This procedure initializes the 2D matrix M with the input and N-1
cyclic shifts in all subsequent rows.
*/
procedure initializeM(int input[],int input_size, int M[][],int
array[])
   1. local int i = 0
   2. from i = 0
   3. loop
         a. array[input[i]] +=1
         b. i+=1
   4. until i = input_size
   5. delocal int i= input_size
   6. local int i = 0
   7. from i = 0
   8. loop
         a. M[0][i] += input[i]
         b. input[i] -= M[0][i]
         c. i+=1
   9. until i = input_size
  10. delocal int i = input_size
  11. local int i = 1
  12. from i = 1
  13. loop
```

```
        a. local int j = 0
        b. from j = 0
        c. loop
             i. M[i][j] += M[i-1][j+1]
            ii. j+=1
        d. until j = input_size - 1
        e. M[i][j] += M[i-1][0]
        f. j+=1
        g. delocal int j = input_size
        h. i+=1
  14. until i = input_size
  15. delocal int i=input_size


/*
Creates a copy of matrix c to matrix d then zero clears  matrix c
using auxiliary stack sorts.
*/
procedure copy_matrix(int c[],int d[],stack sorts)
   1. local int i = 0
   2. from i = 0
   3. loop
        a. local int x = c[i]
        b. d[i] += c[i]
        c. c[i] -= x
        d. push(x,sorts)
        e. delocal int x = 0
        f. i+=1
   4. until i = size(d)
   5. delocal int i = size(d)


/*
Get last column of a sorted M matrix and store it in vector L
*/
procedure createL(int M[][],int input_size, int L[])
   1. local int i = 0
```

```
   2. from i = 0
   3. loop
         a. L[i]+= M[i][input_size-1]
         b. i+=1
   4. until i = input_size
   5. delocal int i = input_size

/*
This function creates  the alphabet of input by counting the non zero
values of array[]. Matrix Y stores this output which will be later
used for move-to-front encoding.
*/
procedure   createY(int   input[],int   array[],int   input_size,int
different,int Y[])
   1. local int count = 0
   2. local int i= 0
   3. from i =0
   4. loop
         a. if array[i] != 0 then
               i.  different+=1
              ii.  Y[count] += i
             iii.  count+=1
         b. fi array[i] != 0
         c. i+=1
   5. until i = size(array)
   6. delocal int i = size(array)
   7. delocal int count = different

/*
This function calculates the number of items in front of L[i] in vector
Y. If L[i] is the first item then the function is not executed and the
return value is 0.
*/
procedure countNum(int num , int Y[], int L)
   1. if (Y[0] != L) then
   2. from num = 0
   3. loop
```

```
        a. num+=1
   4. until Y[num] = L
   5. fi (Y[0] !=L)

/*
This function creates the R vector which is the output of the algorithm
as it prepares the data using move-to-front encoding to achieve better
encoding results.
*/
procedure createR(stack sorts, int Y[], int L[],int R[],int
input_size)
   1. local int i = 0
   2. from i = 0
   3. loop
        a. local int num = 0
        b. call countNum(num,Y,L[i])
        c. R[i] += num
        d. local int j = num
        e. from j = num
        f. loop
             i. Y[j] <=> Y[j-1]
            ii. j-= 1
        g. until j =0
        h. delocal int j = 0
        i. push(num,sorts)
        j. delocal int num = 0
        k. i+=1
   4. until i = input_size
   5. delocal int i=input_size

/*
Reversible implementation of the string compare function. This
function outputs 0 if both rows of M are identical, +1 if the first
comes first (in ascending order) and -1 if the second row comes first.
*/
procedure str_cmp(int M[][],int input_size,int s, int f,int
result,stack aux)
```

```
   1. local int i = 0
   2. from i=0
   3. loop
        a. if ( M[s][i] > M[f][i]) then
             i. result+=1
        b. fi M[s][i] > M[f][i]
        c. if ( M[s][i] < M[f][i] ) then
             i. result -= 1
        d. fi M[s][i] < M[f][i]
        e. i+=1
   4. until result != 0
   5. push(i,aux)
   6. delocal int i = 0

/*
This function sorts the 2D matrix M lexicographically by comparing
each row using string compare function.
*/
procedure lexicographical(int input_size,int M[][],stack aux,stack
res,int d[])
   1. local int i = 0
   2. from i = 0
   3. loop local int j = input_size - 1
   4. from j = input_size - 1
   5. loop
             i. local int result = 0
            ii. call str_cmp(M,input_size,j-1,j,result,aux)
           iii. if ( result = 1 ) then
            iv. local int k = 0
             v. from k = 0
            vi. loop
                   1. M[j][k] <=> M[j-1][k]
                   2. k+=1
           vii. until k = input_size
          viii. delocal int k = input_size
            ix. d[j-i-1] +=1
```

```
            x.  fi d[j-i-1] = i+1
           xi.  push(result,res)
          xii.  delocal int result = 0
         xiii.  j-=1
  6. until j = i
  7. delocal int j = i
  8. i+=1
  9. until i= input_size
 10. delocal int i = input_size
```

Παράρτημα Β

Σε αυτό το παράρτημα δίνεται ο κώδικας σε γλώσσα Janus για την υλοποίηση των αλγορίθμων Data Encryption Standard (DES) και Blowfish.

Υλοποίηση αλγορίθμου Data Encryption Standard (DES):

```
/*
Reversible function  the converts number n from decimal to binary. Binary representation is returned as "bits" in stack result. Inverse call of this function with execute the reversible counterpart action, which is binary to decimal.
*/
procedure BinaryDecimal(int n, stack result)
   1. from empty(result)
   2. loop
         a. local int remainder = n % 2
         b. if remainder = 0 then
               i.  local int k = n / 2
              ii.  k <=> n
             iii.  delocal int k = n * 2
         c. else
               i.  local int k = n / 2
              ii.  k <=> n
             iii.  delocal int k = n * 2 + 1
         d. fi remainder = 0
   3. push(remainder, result)
         a. delocal int remainder = 0
   4. until n = 0
```

```
/*
This function is a reversible implementation of binary conversion.
Input n is converted to binary and stored in n.
*/
procedure DecimalBinary(int n)
   1. local stack result = nil
   2. from empty(result)
   3. loop
         a. local int remainder = n % 2
         b. if remainder = 0 then
               i.  local int k = n / 2
              ii.  k <=> n
             iii.  delocal int k = n * 2
         c. else
               i.  local int k = n / 2
              ii.  k <=> n
             iii.  delocal int k = n * 2 + 1
         d. fi remainder = 0
         e. push(remainder, result)
         f. delocal int remainder = 0
   4. until n = 0
   5. from n = 0
   6. loop
         a. local int x = 0
         b. pop(x, result)
         c. local int k = n * 10 + x
         d. k <=> n
         e. delocal int k = n / 10
         f. delocal int x = n % 10
   7. until empty(result)
   8. delocal stack result = nil

/*
This function converts input parameter key into its equivalent binary
representation and outputs it in K64. The binary representation uses
```

```
4bits to represent any character. Auxiliary stack garbage stores the
number of extra bits used for each character.
*/
procedure convert_to_64binary(int key[],int K64[],stack garbage)
   1. local int count = 0
   2. local stack result = nil
   3. local int i = 0
   4. from i = 0
   5. loop
         a. local int temp = key[i]
         b. call BinaryDecimal(temp,result)
         c. delocal int temp = 0
         d. local int j = size(result)
         e. from j=size(result)
         f. loop
              i. count+=1
             ii. j+=1
         g. until j = 4
         h. delocal int j = 4
         i. local int j = size(result)
         j. local int z = j
         k. from z= j
         l. loop
              i. local int temp = 0
             ii. pop(temp,result)
            iii. K64[count] += temp
             iv. delocal int temp = K64[count]
              v. count+=1
             vi. z-=1
         m. until z = 0
         n. delocal int z = 0
         o. push(j,garbage)
         p. delocal int j=0
         q. i+=1
   6. until i= 16
   7. delocal int i=16
```

```
   8. delocal stack result = nil
   9. delocal int count = size(K64)

/*
This function converts a 64bits input into a 56bits output using PC1
table.
*/
procedure convert_64key_to_56key(int K64[],int K56[])
   1. local int PC1[56]=
      {57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19
      ,11,3,60,52,44,36,63,55,47,39,31,23,15,7,62,54,46,38,30,22,14,6
      ,61,53,45,37,29,21,13,5,28,20,12,4}
   2. local int i= 0
   3. from i=0
   4. loop
         a. K56[i] += K64[PC1[i]-1]
         b. i+=1
   5. until i=56
   6. delocal int i=56
   7. delocal int PC1[56]=
      {57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19
      ,11,3,60,52,44,36,63,55,47,39,31,23,15,7,62,54,46,38,30,22,14,6
      ,61,53,45,37,29,21,13,5,28,20,12,4}

/*
This function creates the final key combining the values of C and D
vectors and using the PC2 permutation table. The final key has 48bits.
*/
procedure create_final_key(int key[],int finalKey[][],stack garbage)
   1. local int C[17][28]=
   2. {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
      },{0}}
   3. local int D[17][28]=
   4. {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
      },{0}}
   5. local int K64[64]={0}
   6. local int K56[56]={0}
```

```
 7. call convert_to_64binary(key,K64,garbage)
 8. call convert_64key_to_56key(K64,K56)
 9. local int shifts[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1}
10. local int PC2[48]=
    {14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,
    2,41,52,31,37,47,55,30,40,51,45,33,48,44,49,39,56,34,53,46,42,5
    0,36,29,32}
11. call create_C_D(C,D, K56,shifts)
12. local int i = 0
13. from i=0
14. loop
       a. local int j =0
       b. from j =0
       c. loop
            i. if PC2[j] <= 28 then
                  1. finalKey[i][j] += C[i+1][PC2[j]-1]
           ii. else
          iii. finalKey[i][j] += D[i+1][(PC2[j]-28)-1]
           iv. fi PC2[j] <= 28
            v. j+=1
       d. until j =48
       e. delocal int j = 48
       f. i+=1
15. until i= 16
16. delocal int i = 16
17. uncall create_C_D(C,D, K56,shifts)
18. delocal int PC2[48] =
19. {14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,
   2,41,52,31,37,47,55,30,40,51,45,33,48,44,49,39,56,34,53,46,42,50
   ,36,29,32}
20. delocal int shifts[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1}
21. uncall convert_64key_to_56key(K64,K56)
22. uncall convert_to_64binary(key,K64,garbage)
23. delocal int K56[56]={0}
24. delocal int K64[64]={0}
25. delocal int D[17][28]=
```

```
    {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
    ,{0}}
 26. delocal int C[17][28]=
    {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
    ,{0}}

/*
This function splits K56 key into C and D and applies cyclic shifts
as expressed in shifts array to create 16 permutations.
*/
procedure create_C_D(int C[][],int D[][],int K56[],int shifts[])
   1. local int i = 0
   2. from i=0
   3. loop
         a. if i<28 then
              i. C[0][i%28] +=K56[i]
         b. else
              i. D[0][i%28] +=K56[i]
         c. fi i<28
         d. i+=1
 4. until i=56
 5. delocal int i = 56
 6. local int i =1
 7. from i=1
 8. loop
         a. local int j = shifts[i-1]
         b. local int z = 0
         c. from z= 0
         d. loop
              i. C[i][z] +=C[i-1][z+j]
             ii. D[i][z] +=D[i-1][z+j]
            iii. z+=1
         e. until z = 28-j
         f. local int p = 0
         g. from p = 0
         h. loop
```

```
            i. C[i][z] += C[i-1][p]
           ii. D[i][z] +=D[i-1][p]
          iii. p+=1
           iv. z+=1
        i. until p = j
        j. delocal int p = j
        k. delocal int z = 28
        l. delocal int j = shifts[i-1]
        m. i +=1
 9. until i = 17
10.  delocal int i = 17

/*
This function  applies in initial permutation IP on the input after it converts it to 64bit binary representation using convert_to_64binary function.
*/
procedure transform(int input[],int transformed[],stack garbage)
   1. local int input_bin[64]={0}
   2. call convert_to_64binary(input,input_bin,garbage)
   3. local int IP[64] =
      {58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,62,54,46,38,30,2
      2,14,6,64,56,48,40,32,24,16,8,57,49,41,33,25,17,9,1,59,51,43,35
      ,27,19,11,3,61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7}
   4. local int i = 0
   5. from i =0
   6. loop
         a. transformed[i]+= input_bin[IP[i]-1]
         b. i+=1
   7. until i= 64
   8. delocal int i = 64
   9. delocal int IP[64]=
      {58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,62,54,46,38,30,2
      2,14,6,64,56,48,40,32,24,16,8,57,49,41,33,25,17,9,1,59,51,43,35
      ,27,19,11,3,61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7}
  10. uncall convert_to_64binary(input,input_bin,garbage)
```

```
 11. delocal int input_bin[64]={0}

/*
Reversible implementation of the Feistelnetwork for DES algorithm.
This function call the F function to compute the necessary functions
for encryption.
*/
procedure create_L_R(int L[][],int R[][], int finalKey[][], int
transformed[], int F[][],stack garbage)
   1. local int i = 0
   2. from i = 0
   3. loop
         a. if ( i < 32 ) then
               i. L[0][i] +=transformed[i]
         b. else
               i. R[0][i%32] += transformed[i]
         c. fi i<32
         d. i+=1
   4. until i = 64
   5. delocal int i = 64
   6. local int i = 1
   7. from i = 1
   8. loop
         a. call functionF(R,i-1,finalKey,F,garbage)
         b. local int j = 0
         c. from j = 0
         d. loop
         e. L[i][j] += R[i-1][j]
         f. R[i][j] += (L[i-1][j] + F[i-1][j] ) %2
         g. j+=1
         h. until j = 32
 9. delocal int j = 32
10. i+=1
11. until i=17
12. delocal int i = 17
```

```
/*
Apply E bit selection extension on R matrix to extend it from 32bits
to 48bits.
*/
procedure createE(int R[][],int index,int finalKey[][],int E_out[][])
   1. local int E[48]=
      {32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13,14,15,16,17,16,
      17,18,19,20,21,20,21,22,23,24,25,24,25,26,27,28,29,28,29,30,31,
      32,1}
   2. local int i = 0
   3. from i=0
   4. loop
         a. E_out[index][i] += (R[index][E[i]-1]+finalKey[index][i])%2
         b. i+=1
   5. until i=48
   6. delocal int i=48
   7. delocal int E[48]=
      {32,1,2,3,4,5,4,5,6,7,8,9,8,9,10,11,12,13,12,13,14,15,16,17,16,
      17,18,19,20,21,20,21,22,23,24,25,24,25,26,27,28,29,28,29,30,31,
      32,1}


/*
Split E_out into 8 boxes for 6bits each to use later in the Sboxes.
*/
procedure createBoxes(int E_out[][],int index,int Boxes[][])
   1. local int i = 0
   2. from i = 0
   3. loop
         a. local int j = 0
         b. from j = 0
         c. loop
               i. Boxes[i][j]+=E_out[index][i*6+j]
              ii. j+=1
         d. until j= 6
         e. delocal int j = 6
```

```
        f. i+=1
   4. until i=8
   5. delocal int i = 8

/*
This function calculates the output of Sbox operation. Each box each
converted to a row and column using the first and last bit for row and
the rest for column. The function outputs number which is the value
of Sbox[][row*16+column].
*/
procedure Sbox(int number[],int Box[][],stack garbage)
   1. local int S[8][64] =
      {{14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,0,15,7,4,14,2,13,1,10,6
      ,12,11,9,5,3,8,4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,15,12,8,2,
      4,9,1,7,5,11,3,14,10,0,6,13},
      {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,3,13,4,7,15,2,8,14,12,0,
      1,10,6,9,11,5,0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,13,8,10,1,3
      ,15,4,2,11,6,7,12,0,5,14,9},
      {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,13,7,0,9,3,4,6,10,2,8,5,
      14,12,11,15,1,13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,1,10,13,0,6
      ,9,8,7,4,15,14,3,11,5,2,12},
      {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,13,8,11,5,6,15,0,3,4,7,2
      ,12,1,10,14,9,10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,3,15,0,6,10
      ,1,13,8,9,4,5,11,12,7,2,14},
      {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,14,11,2,12,4,7,13,1,5,0,
      15,10,3,9,8,6,4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,11,8,12,7,1
      ,14,2,13,6,15,0,9,10,4,5,3},
      {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,10,15,4,2,7,12,9,5,6,1,1
      3,14,0,11,3,8,9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,4,3,2,12,9,
      5,15,10,11,14,1,7,6,0,8,13},
      {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,13,0,11,7,4,9,1,10,14,3,
      5,12,2,15,8,6,1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,6,11,13,8,1
      ,4,10,7,9,5,0,15,14,2,3,12},
      {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,1,15,13,8,10,3,7,4,12,5,
      6,11,0,14,9,2,7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,2,1,14,7,4,
      10,8,13,15,12,9,0,3,5,6,11}}
```

```
2. local int i = 0
3. from i=0
4. loop
      a. local int row = Box[i][0]*10+Box[i][5]
      b. local int column =
         Box[i][1]*1000+Box[i][2]*100+Box[i][3]*10+Box[i][4]
      c. uncall DecimalBinary(row)
      d. uncall DecimalBinary(column)
      e. number[i] += S[i][row*16+column]
      f. push(row,garbage)
      g. push(column,garbage)
      h. delocal int column = 0
      i. delocal int row = 0
      j. i+=1
5. until i = 8
6. delocal int i =8
7. delocal int S[8][64] =
   {{14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,0,15,7,4,14,2,13,1,10,6
   ,12,11,9,5,3,8,4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,15,12,8,2,
   4,9,1,7,5,11,3,14,10,0,6,13},
   {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,3,13,4,7,15,2,8,14,12,0,
   1,10,6,9,11,5,0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,13,8,10,1,3
   ,15,4,2,11,6,7,12,0,5,14,9},
   {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,13,7,0,9,3,4,6,10,2,8,5,
   14,12,11,15,1,13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,1,10,13,0,6
   ,9,8,7,4,15,14,3,11,5,2,12},
   {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,13,8,11,5,6,15,0,3,4,7,2
   ,12,1,10,14,9,10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,3,15,0,6,10
   ,1,13,8,9,4,5,11,12,7,2,14},
   {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,14,11,2,12,4,7,13,1,5,0,
   15,10,3,9,8,6,4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,11,8,12,7,1
   ,14,2,13,6,15,0,9,10,4,5,3},
   {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,10,15,4,2,7,12,9,5,6,1,1
   3,14,0,11,3,8,9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,4,3,2,12,9,
   5,15,10,11,14,1,7,6,0,8,13},
```

```
      {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,13,0,11,7,4,9,1,10,14,3,
      5,12,2,15,8,6,1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,6,11,13,8,1
      ,4,10,7,9,5,0,15,14,2,3,12},
      {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,1,15,13,8,10,3,7,4,12,5,
      6,11,0,14,9,2,7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,2,1,14,7,4,
      10,8,13,15,12,9,0,3,5,6,11}}

/*
F function of the Feistelnetwork for DES  algorithm.
*/
procedure functionF(int R[][],int index,int finalKey[][],int F[][],
stack garbage)
   1. local int aux[32]={0}
   2. local int Boxes[8][6] ={{0},{0},{0},{0},{0},{0},{0},{0}}
   3. local              int              E_out[16][48]              =
      {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
      }}
   4. call createE(R,index,finalKey,E_out)
   5. call createBoxes(E_out,index,Boxes)
   6. local stack help = nil
   7. local int number[8]={0}
   8. call Sbox(number,Boxes,garbage)
   9. local int i = 0
  10. from i = 0
  11. loop
        a. local stack result=nil
        b. local int temp = number[i]
        c. call BinaryDecimal(temp,result)
        d. delocal int temp = 0
        e. local int j = size(result)
        f. if ( j < 4 ) then
             i. local int z = j
            ii. from z = j
           iii. loop
                  1. push(0,result)
                  2. z+=1
```

```
            iv. until z = 4
             v. delocal int z = 4
       g. fi ( j < 4 )
       h. push(j,help)
       i. delocal int j=0
       j. local int j = 0
       k. from j = 0
       l. loop
             i. local int temp = 0
            ii. pop(temp,result)
           iii. aux[4*i+j] += temp
            iv. temp -= aux[4*i+j]
             v. delocal int temp = 0
            vi. j+=1
       m. until j= 4
       n. delocal int j= 4
       o. delocal stack result = nil
       p. i+=1
12. until i= 8
13. delocal int i = 8
14. uncall Sbox(number,Boxes,garbage)
15. delocal int number[8]={0}
16. local int i = 0
17. from i = 0
18. loop
       a. local int voithitiko = 0
       b. pop(voithitiko,help)
       c. push(voithitiko,garbage)
       d. delocal int voithitiko=0
       e. i+=1
19. until size(help)=0
20. push(i,garbage)
21. delocal int i = 0
22. delocal stack help = nil
23. uncall createBoxes(E_out,index,Boxes)
24. uncall createE(R,index,finalKey,E_out)
```

```
25. delocal int E_out[16][48] =
    {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
    }
26. delocal int Boxes[8][6] ={{0},{0},{0},{0},{0},{0},{0},{0}}
27. local int P[32]=
    {16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,32,27,3,
    9,19,13,30,6,22,11,4,25}
28. local int i= 0
29. from i =0
30. loop
        a. F[index][i]+=aux[P[i]-1]
        b. push(aux[P[i]-1],garbage)
        c. aux[P[i]-1] -= top(garbage)
        d. i+=1
31. until i = 32
32. delocal int i = 32
33. delocal int P[32]=
    {16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,32,27,3,
    9,19,13,30,6,22,11,4,25}
34. delocal int aux[32] = {0}
```

```
/*
Final step towards encryption of input. Every step is executed in this
function and the final permutation IP inverse is applied on the output.
*/
procedure encryption(int key[],int input[],int encrypted[])
   1. local stack garbage = nil
   2. local int finalKey[16][48]=
   3. {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
      }}
   4. call create_final_key(key,finalKey,garbage)
   5. local int transformed[64]={0}
   6. call transform(input,transformed, garbage)
   7. local int L[17][32]=
   8. {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
      },{0}}
```

```
 9. local int R[17][32]=
    {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0
    },{0}}
10. local int F[16][32]=
   {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
   }
11. call create_L_R(L,R,finalKey,transformed,F,garbage)
12. local int IP_REV[64]=
   {40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,38,6,46,14,54,22,
   62,30,37,5,45,13,53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,5
   1,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25}
13. local int i = 0
14. from i = 0
15. loop
       a. if ( IP_REV[i] <= 32 ) then
            i. encrypted[i] += R[16][IP_REV[i]-1]
       b. else
            i. encrypted[i] += L[16][IP_REV[i]-1-32]
       c. fi IP_REV[i] <= 32
       d. i+=1
16. until i=64
17. delocal int i = 64
18. delocal int IP_REV[64] =
   {40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,38,6,46,14,54,22,
   62,30,37,5,45,13,53,21,61,29,36,4,44,12,52,20,60,28,35,3,43,11,5
   1,19,59,27,34,2,42,10,50,18,58,26,33,1,41,9,49,17,57,25}
19. uncall create_L_R(L,R,finalKey,transformed,F,garbage)
20. delocal int F[16][32]=
   {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
   }
21. delocal int R[17][32]=
   {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
   ,{0}}
22. delocal int L[17][32]=
   {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
   ,{0}}
```

```
  23. uncall transform(input,transformed, garbage)
  24. delocal int transformed[64]={0}
  25. uncall create_final_key(key,finalKey,garbage)
  26. delocal int finalKey[16][48]=
     {{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0},{0}
     }
  27. delocal stack garbage = nil

/*
Convert binary encryption into a decimal representation and replace
the text with the output.
*/
procedure createFinalText(int key[],int input[],int EncryptedText[])
   1. local int encrypted[64]={0}
   2. call encryption(key,input,encrypted)
   3. local int i =0
   4. from i = 0
   5. loop
         a. local int j = encrypted[4*i]*1000 +encrypted[4*i+1]*100
            +encrypted[4*i+2]*10 +encrypted[4*i+3]
         b. uncall DecimalBinary(j)
         c. EncryptedText[i] += j
         d. delocal int j = EncryptedText[i]
         e. i+=1
  6. until i = 16
  7. delocal int i = 16
  8. uncall encryption(key,input,encrypted)
  9. delocal  int encrypted[64]={0}
```

Υλοποίηση αλγορίθμου Blowfish:

```
/*
Reversible function  the converts number n from decimal to binary.
Binary representation is returned as "bits" in stack result. Inverse
```

```
call of this function with execute the reversible counterpart action,
which is binary to decimal.
*/
procedure BinaryDecimal(int n, stack result)
   1. from empty(result)
   2. loop
         a. local int remainder = n % 2
         b. if remainder = 0 then
              i.  local int k = n / 2
             ii.  k <=> n
            iii.  delocal int k = n * 2
         c. else
              i.  local int k = n / 2
             ii.  k <=> n
            iii.  delocal int k = n * 2 + 1
         d. fi remainder = 0
   3. push(remainder, result)
   4. delocal int remainder = 0
   5. until n = 0



/*
This function is a reversible implementation of binary conversion.
Input n is converted to binary and stored in n.
*/
procedure DecimalBinary(int n)
  1. local stack result = nil
  2. from empty(result)
  3. loop
         a. local int remainder = n % 2
         b. if remainder = 0 then
              i.  local int k = n / 2
             ii.  k <=> n
            iii.  delocal int k = n * 2
         c. else
```

```
            i. local int k = n / 2
           ii. k <=> n
          iii. delocal int k = n * 2 + 1
      d. fi remainder = 0
      e. push(remainder, result)
      f. delocal int remainder = 0
   4. until n = 0
   5. from n = 0
   6. loop
      a. local int x = 0
      b. pop(x, result)
      c. local int k = n * 10 + x
      d. k <=> n
      e. delocal int k = n / 10
      f. delocal int x = n % 10
   7. until empty(result)
   8. delocal stack result = nil

/*
This function converts input parameter key into its equivalent binary
representation and outputs it in K64. The binary representation uses
4bits to represent any character. Auxiliary stack garbage stores the
number of extra bits used for each character.
*/
procedure convert_to_64binary(int key[],int K64[],stack garbage)
   1. local int count = 0
   2. local stack result = nil
   3. local int i = 0
   4. from i = 0
   5. loop
      a. local int temp = key[i]
      b. call BinaryDecimal(temp,result)
      c. delocal int temp = 0
      d. local int j = size(result)
      e. from j=size(result)
```

```
        f. loop
             i. count+=1
            ii. j+=1
        g. until j = 4
        h. delocal int j = 4
        i. local int j = size(result)
        j. local int z = j
        k. from z= j
        l. loop
             i. local int temp = 0
            ii. pop(temp,result)
           iii. K64[count] += temp
            iv. delocal int temp = K64[count]
             v. count+=1
            vi. z-=1
        m. until z = 0
        n. delocal int z = 0
        o. push(j,garbage)
        p. delocal int j=0
        q. i+=1
   6. until i= 16
   7. delocal int i=16
   8. delocal stack result = nil
   9. delocal int count = size(K64)

/*
This function splits K64 into 2 vectors containing the first half and
the second half. These vectors are L and R respectively.
*/
procedure createLR(int L[],int R[],int K64[])
   1. local int i =0
   2. from i =0
   3. loop
        a. L[i]+=K64[i]
        b. i+=1
   4. until i=32
```

```
   5. from i = 32
   6. loop
         a. R[i-32]+=K64[i]
         b. i+=1
   7. until i=64
   8. delocal int i = 64

/*
This function converts P array to binary representation. This
representation uses 4bits for every element.
*/
procedure binaryP(int P[][],int index,int AuxP[],stack garbage)
   1. local int count=0
   2. local stack result = nil
   3. local int i = 0
   4. from i = 0
   5. loop
         a. local int temp = P[index][i]
         b. call BinaryDecimal(temp,result)
         c. delocal int temp = 0
         d. local int j = size(result)
         e. from j=size(result)
         f. loop
              i. count+=1
             ii. j+=1
         g. until j = 4
         h. delocal int j = 4
         i. local int j = size(result)
         j. local int z = j
         k. from z= j
         l. loop
              i. local int temp = 0
             ii. pop(temp,result)
            iii. AuxP[count] += temp
             iv. delocal int temp = AuxP[count]
              v. count+=1
```

```
            vi.  z-=1
         m. until z = 0
         n. delocal int z = 0
         o. push(j,garbage)
         p. delocal int j=0
         q. i+=1
   6. until i= 8
   7. delocal int i=8
   8. delocal stack result = nil
   9. delocal int count =32

/*
Call initialize_P function with P array's fixed values.
*/
procedure initialize_Parray(int Parray[][], int K64[],stack garbage)
   1. local int P[18][8] = {{2,4,3,15,6,10,8,8},{8,5,10,3,0,8,13,3},
      {1,3,1,9,8,10,2,14},{0,3,7,0,7,3,4,4},{10,4,0,9,3,8,2,2},{2,9,9
      ,15,3,1,13,0},{0,8,2,14,15,10,9,8},{14,12,4,14,6,12,8,9},{4,5,2
      ,8,2,1,14,6},{3,8,13,0,1,3,7,7},{11,14,5,4,6,6,12,15},{3,4,14,9
      ,0,12,6,12},{12,0,10,12,2,9,11,7},{12,9,7,12,5,0,13,13},{3,15,8
      ,4,13,5,11,5},{11,5,4,7,0,9,1,7},{9,2,1,6,13,5,13,9},{8,9,7,9,1
      5,11,1,11}}
  2. call initialize_P(P,Parray,K64,garbage)
  3. delocal int P[18][8] = {{2,4,3,15,6,10,8,8},{8,5,10,3,0,8,13,3},
     {1,3,1,9,8,10,2,14},{0,3,7,0,7,3,4,4},{10,4,0,9,3,8,2,2},{2,9,9,
     15,3,1,13,0},{0,8,2,14,15,10,9,8},{14,12,4,14,6,12,8,9},{4,5,2,8
     ,2,1,14,6},{3,8,13,0,1,3,7,7},{11,14,5,4,6,6,12,15},{3,4,14,9,0,
     12,6,12},{12,0,10,12,2,9,11,7},{12,9,7,12,5,0,13,13},{3,15,8,4,1
     3,5,11,5},{11,5,4,7,0,9,1,7},{9,2,1,6,13,5,13,9},{8,9,7,9,15,11,
     1,11}}

/*
This function modifies P vectors by using XOR operation on every
P vector with half the key.
*/
```

```
procedure initialize_P(int P[][],int Parray[][],int K64[],stack
garbage)
   1. local int L[32] = {0}
   2. local int R[32] = {0}
   3. call createLR(L,R,K64)
   4. local int i = 0
   5. from i = 0
   6. loop
         a. local int AuxP[32] ={0}
         b. call binaryP(P,i, AuxP,garbage)
         c. if (i%2=0) then
              i. local int j =0
             ii. from j = 0
            iii. loop
                    1. Parray[i][j] +=AuxP[j]
                    2. Parray[i][j] ^=L[j]
                    3. j+=1
             iv. until j = 32
              v. delocal int j = 32
         d. else
              i. local int j =0
             ii. from j = 0
            iii. loop
                    1. Parray[i][j] +=AuxP[j]
                    2. Parray[i][j] ^=R[j]
                    3. j+=1
             iv. until j = 32
              v. delocal int j = 32
         e. fi i%2=0
         f. uncall binaryP(P,i,AuxP,garbage)
         g. delocal int AuxP[32]={0}
         h. i+=1
   7. until i =18
   8. delocal int i = 18
   9. uncall createLR(L,R,K64)
  10. delocal int R[32]={0}
```

```
  11. delocal int L[32]={0}

/*
This function uses binaryP to  convert each Sbox values to binary
representation and then XOR each of these values with L and R in order
to modify its values.
*/
procedure initialize_Sbox(int Sbox0[][],int Sbox[][],int K64[],stack
garbage)
   1. local int L[32] = {0}
   2. local int R[32] = {0}
   3. call createLR(L,R,K64)
   4. local int i = 0
   5. from i = 0
   6. loop
         a. local int AuxP[32] ={0}
         b. call binaryP(Sbox0,i, AuxP,garbage)
         c. if ( i%2=0) then
              i. local int j =0
             ii. from j = 0
            iii. loop
                    1. Sbox[i][j] +=AuxP[j]
                    2. Sbox[i][j] ^=L[j]
                    3. j+=1
             iv. until j = 32
              v. delocal int j = 32
         d. else
              i. local int j =0
             ii. from j = 0
            iii. loop
                    1. Sbox[i][j] +=AuxP[j]
                    2. Sbox[i][j] ^=R[j]
                    3. j+=1
             iv. until j = 32
              v. delocal int j = 32
         e. fi i%2=0
```

```
        f. uncall binaryP(Sbox0,i,AuxP,garbage)
        g. delocal int AuxP[32]={0}
        h. i+=1
   7. until i =64
   8. delocal int i = 64
   9. uncall createLR(L,R,K64)
  10. delocal int R[32]={0}
  11. delocal int L[32]={0}

/*
This function uses initialize_Sbox to modify existing values of Sboxes
with the same way as the previous functions (using XOR operation).
*/
procedure    initialize_Sboxes(int    Sbox0[][],int    Sbox1[][],int
Sbox2[][],int Sbox3[][],int K64[],stack garbage)
   1. local int S0= //fixed values of pi in hexadecimal
   2. local int S1= //fixed values of pi in hexadecimal
   3. local int S2= //fixed values of pi in hexadecimal
   4. local int S3= //fixed values of pi in hexadecimal
   5. call initialize_Sbox(S0,Sbox0,K64,garbage)
   6. call initialize_Sbox(S1,Sbox1,K64,garbage)
   7. call initialize_Sbox(S2,Sbox2,K64,garbage)
   8. call initialize_Sbox(S3,Sbox3,K64,garbage)
   9. delocal int S3= //fixed values of pi in hexadecimal
  10. delocal int S2= //fixed values of pi in hexadecimal
  11. delocal int S1= //fixed values of pi in hexadecimal
  12. delocal int S0= //fixed values of pi in hexadecimal

/*
Auxiliary function used in F function that computes bit by bit
addition. It is used in order to be able to reverse the results using
inverse call (uncall).
*/
procedure addition(int S[][],int a,int B[][],int b,int RES[],stack
garbage)
   1. local int i = 31
   2. from i=31
```

```
   3. loop
         a. RES[i] += (S[a][i] + B[b][i]) %2
         b. i-=1
   4. until i = -1
   5. delocal int i =-1

/*
Auxiliary function used for reversibility in F function.
*/
procedure aux_F(int res1[],int res2[],int S2[][],int c)
   1. local int i=0
   2. from i=0
   3. loop
         a. res2[i]+= res1[i]
         b. res2[i]^=S2[c][i]
         c. i+=1
   4. until i=32
   5. delocal int i = 32

/*
F function of the Feistelnetwork using XOR and Addition operations
using Sboxes' values.
*/
procedure  F_function(int  F[],int  S0[][],int  S1[][],int  S2[][],int
S3[][],int index,int a,int b,int c,int d,stack garbage)
   1. local int res1[32]={0}
   2. call addition(S0,a,S1,b,res1,garbage)
   3. local int res2[32]={0}
   4. call aux_F(res1,res2,S2,c)
   5. local int i = 31
   6. from i=31
   7. loop
         a. F[i] += (res2[i] + S3[d][i] ) %2
         b. i-=1
   8. until i = -1
   9. delocal int i=-1
  10. uncall aux_F(res1,res2,S2,c)
```

```
  11. delocal int res2[32]={0}
  12. uncall addition(S0,a,S1,b,res1,garbage)
  13. delocal int res1[32]={0}

/*
Function that iterates 16 times and implementing the necessary
operations for the Feistelnetwork. The encrypted text is the output
value.
*/
procedure rounds(int input64[],int P[][],int encrypted[],int
Sbox0[][],int Sbox1[][],int Sbox2[][],int Sbox3[][],stack garbage)
   1. local int L[32]={0}
   2. local int R[32]={0}
   3. call createLR(L,R,input64)
   4. call innerRounds(P,Sbox0,Sbox1,Sbox2,Sbox3,garbage,L,R)
   5. local int i = 0
   6. from i=0
   7. loop
         a. if ( i< 32) then
               i. encrypted[i] +=L[i]
         b. else
               i. encrypted[i] +=R[i-32]
         c. fi i < 32
         d. i+=1
   8. until i=64
   9. delocal int i=64
  10. uncall innerRounds(P,Sbox0,Sbox1,Sbox2,Sbox3,garbage,L,R)
  11. uncall createLR(L,R,input64)
  12. delocal int R[32]={0}
  13. delocal int L[32]={0}

/*
This function splits L and creates a,b,c,d. Then converts them to
decimal and uses them as parameters for function F that will choose
the value of each Sbox.
*/
```

```
procedure innerRounds(int P[][],int Sbox0[][],int Sbox1[][],int
Sbox2[][],int Sbox3[][],stack garbage,int L[],int R[])
   1. local int i = 0
   2. from i =0
   3. loop
        a. local int j = 0
        b. from j =0
        c. loop
             i. L[j]^=P[i][j]
            ii. j+=1
        d. until j=32
        e. delocal int j =32
        f. local int F[32]={0}
        g. local int a=
           L[7]+L[6]*10+L[5]*100+L[4]*1000+L[3]*10000+L[2]*100000+L[
           1]*1000000+L[0]*10000000
        h. uncall DecimalBinary(a)
        i. local int b=
           L[15]+L[14]*10+L[13]*100+L[12]*1000+L[11]*10000+L[10]*100
           000+L[9]*1000000+L[8]*10000000
        j. uncall DecimalBinary(b)
        k. local int c=
           L[23]+L[22]*10+L[21]*100+L[20]*1000+L[19]*10000+L[18]*100
           000+L[17]*1000000+L[16]*10000000
        l. uncall DecimalBinary(c)
        m. local int d=
           L[31]+L[30]*10+L[29]*100+L[28]*1000+L[27]*10000+L[26]*100
           000+L[25]*1000000+L[24]*10000000
        n. uncall DecimalBinary(d)
        o. call
           F_function(F,Sbox0,Sbox1,Sbox2,Sbox3,i,(a%64),(b%64),(c%6
           4),(d%64),garbage)
        p. local int j = 0
        q. from j =0
        r. loop
             i. R[j]^=F[j]
```

```
        ii.  j+=1
      s. until j=32
      t. delocal int j =32
      u. uncall
         F_function(F,Sbox0,Sbox1,Sbox2,Sbox3,i,(a%64),(b%64),(c%6
         4),(d%64),garbage)
      v. call DecimalBinary(d)
      w. delocal int d=
         L[31]+L[30]*10+L[29]*100+L[28]*1000+L[27]*10000+L[26]*100
         000+L[25]*1000000+L[24]*10000000
      x. call DecimalBinary(c)
      y. delocal int c=
         L[23]+L[22]*10+L[21]*100+L[20]*1000+L[19]*10000+L[18]*100
         000+L[17]*1000000+L[16]*10000000
      z. call DecimalBinary(b)
     aa. delocal int b=
         L[15]+L[14]*10+L[13]*100+L[12]*1000+L[11]*10000+L[10]*100
         000+L[9]*1000000+L[8]*10000000
     bb. call DecimalBinary(a)
     cc. delocal           int           a=           L[7]+L[6]*10
      +L[5]*100+L[4]*1000+L[3]*10000+L[2]*100000+L[1]*1000000+L[0]
      *10000000
     dd. delocal int F[32]={0}
     ee. local int j = 0
     ff. from j =0
     gg. loop
           i.  R[j]<=>L[j]
          ii.  j+=1
     hh. until j=32
     ii. delocal int j =32
     jj. i+=1
  4. until i=16
  5. delocal int i=16
  6. local int j = 0
  7. from j =0
  8. loop
```

```
        a. R[j]<=>L[j]
        b. R[j]^=P[16][j]
        c. L[j]^=P[17][j]
        d. j+=1
   9. until j=32
  10. delocal int j =32

/*
Update the values of P using the new key which is the encrypted zero
sequence.
*/
procedure InitializeRealP(int Parray[][],int AuxP[][],int K64[])
   1. local int L[32] = {0}
   2. local int R[32] = {0}
   3. call createLR(L,R,K64)
   4. local int i = 0
   5. from i = 0
   6. loop
        a. if ( i%2=0) then
             i. local int j =0
            ii. from j = 0
           iii. loop
                   1. Parray[i][j] +=AuxP[i][j]
                   2. Parray[i][j] ^=L[j]
                   3. j+=1
            iv. until j = 32
             v. delocal int j = 32
        b. else
             i. local int j =0
            ii. from j = 0
           iii. loop
                   1. Parray[i][j] +=AuxP[i][j]
                   2. Parray[i][j] ^=R[j]
                   3. j+=1
            iv. until j = 32
             v. delocal int j = 32
```

```
        c. fi i%2=0
        d. i+=1
   7. until i =18
   8. delocal int i = 18
   9. uncall createLR(L,R,K64)
  10. delocal int R[32]={0}
  11. delocal int L[32]={0}

/*
Function that updates Sbox values using the new key.
*/
procedure InitializeRealSbox(int Parray[][],int AuxP[][],int K64[])
   1. local int L[32] = {0}
   2. local int R[32] = {0}
   3. call createLR(L,R,K64)
   4. local int i = 0
   5. from i = 0
   6. loop
        a. if ( i%2=0) then
             i. local int j =0
            ii. from j = 0
           iii. loop
                   1. Parray[i][j] +=AuxP[i][j]
                   2. Parray[i][j] ^=L[j]
                   3. j+=1
            iv. until j = 32
             v. delocal int j = 32
        b. else
             i. local int j =0
            ii. from j = 0
           iii. loop
                   1. Parray[i][j] +=AuxP[i][j]
                   2. Parray[i][j] ^=R[j]
                   3. j+=1
            iv. until j = 32
             v. delocal int j = 32
```

```
        c. fi i%2=0
        d. i+=1
   7. until i =64
   8. delocal int i = 64
   9. uncall createLR(L,R,K64)
  10. delocal int R[32]={0}
  11. delocal int L[32]={0}

/*
Function to calculate the encryption. It is used a function for
compute-uncompute method.
*/
procedure Calc_Encryption(int K64[],stack garbage,int P[][],int
S0[][],int S1[][],int S2[][],int S3[][],int bit64[],int input[])
   1. local int encrypted[64]={0}
   2. local int encrypted64[64]={0}
   3. local int allZero[16]={0}
   4. local int allZero64[64]={0}
   5. local int Parray[18][32]={{0}}
   6. local int Sbox0[64][32]={{0}}
   7. local int Sbox1[64][32]={{0}}
   8. local int Sbox2[64][32]={{0}}
   9. local int Sbox3[64][32]={{0}}
  10. local int input64[64]={0}
  11. call convert_to_64binary(allZero,allZero64,garbage)
  12. call initialize_Parray(Parray,K64,garbage)
  13. call initialize_Sboxes(Sbox0,Sbox1,Sbox2,Sbox3,K64,garbage)
  14. call rounds(allZero64,Parray,encrypted,Sbox0,Sbox1,Sbox2,Sbox3,
      garbage)
  15. call convert_to_64binary(input,input64,garbage)
  16. call InitializeRealP(P,Parray,encrypted)
  17. call InitializeRealSbox(S0,Sbox0,encrypted)
  18. call InitializeRealSbox(S1,Sbox1,encrypted)
  19. call InitializeRealSbox(S2,Sbox2,encrypted)
  20. call InitializeRealSbox(S3,Sbox3,encrypted)
  21. call rounds(input64,P,encrypted64,S0, S1,S2,S3, garbage)
```

```
 22. local int i =0
 23. from i=0
 24. loop
       a. bit64[i]+=encrypted64[i]
       b. i+=1
 25. until i=64
 26. delocal int i =64
 27. uncall rounds(input64,P,encrypted64,S0, S1,S2,S3, garbage)
 28. uncall InitializeRealSbox(S3,Sbox3,encrypted)
 29. uncall InitializeRealSbox(S2,Sbox2,encrypted)
 30. uncall InitializeRealSbox(S1,Sbox1,encrypted)
 31. uncall InitializeRealSbox(S0,Sbox0,encrypted)
 32. uncall InitializeRealP(P,Parray,encrypted)
 33. uncall convert_to_64binary(input,input64,garbage)
 34. uncall rounds
    (allZero64,Parray,encrypted,Sbox0, Sbox1,Sbox2,Sbox3, garbage)
 35. uncall initialize_Sboxes(Sbox0,Sbox1,Sbox2,Sbox3,K64,garbage)
 36. uncall initialize_Parray(Parray,K64,garbage)
 37. uncall convert_to_64binary(allZero,allZero64,garbage)
 38. delocal int input64[64]={0}
 39. delocal int Sbox3[64][32]={{0}}
 40. delocal int Sbox2[64][32]={{0}}
 41. delocal int Sbox1[64][32]={{0}}
 42. delocal int Sbox0[64][32]={{0}}
 43. delocal int Parray[18][32]={{0}
 44. delocal int allZero64[64]={0}
 45. delocal int allZero[16]={0}
 46. delocal int encrypted64[64]={0}
 47. delocal int encrypted[64]={0}

/*
Fully integrated blowfish encryption function. Forward call to this
function executes the algorithm and creates the output.
*/
procedure      BlowfishEncryption(int      input[],int      key[],int
EncryptedText[])
```

```
 1. local stack garbage = nil
 2. local int bit64[64]={0}
 3. local int P[18][32]={{0}}
 4. local int S0[64][32]={{0}}
 5. local int S1[64][32]={{0}}
 6. local int S2[64][32]={{0}}
 7. local int S3[64][32]={{0}}
 8. local int K64[64]={0}
 9. call convert_to_64binary(key,K64,garbage)
10. callCalc_Encryption(K64,garbage,P,S0,S1,S2,S3,bit64,input)
11. local int i =0
12. from i = 0
13. loop
      a. local int j=
         bit64[4*i]*1000+bit64[4*i+1]*100+bit64[4*i+2]*10
         +bit64[4*i+3]
      b. uncall DecimalBinary(j)
      c. EncryptedText[i] += j
      d. delocal int j = EncryptedText[i]
      e. i+=1
14. until i = 16
15. delocal int i = 16
16. uncall Calc_Encryption(K64,garbage,P,S0,S1,S2,S3,bit64,input)
17. uncall convert_to_64binary(key,K64,garbage)
18. delocal int K64[64]={0}
19. delocal int S3[64][32]={{0}}
20. delocal int S2[64][32]={{0}}
21. delocal int S1[64][32]={{0}}
22. delocal int S0[64][32]={{0}}
23. delocal int P[18][32]={{0}}
24. delocal int bit64[64]={0}
25. delocal stack garbage = nil
```

```
/*
This function call blowfish encryption on every 64bit block in the
input.
*/
procedure Blowfish(int Fullinput[], int key[], int
FullEncryptedText[])
   1. local int iterations= 0
   2. from iterations = 0
   3. loop
         a. local int input[16]={0}
         b. local int encryptedText[16]={0}
         c. local int i=0
         d. from i =0
         e. loop
              i. input[i]+=Fullinput[iterations*16+i]
             ii. i+=1
         f. until i=16
         g. delocal int i =16
         h. call BlowfishEncryption(input,key,encryptedText)
         i. local int i = 0
         j. from i = 0
         k. loop
              i. FullEncryptedText[iterations*16+i]+=encryptedText[i]
             ii. encryptedText[i]-=FullEncryptedText[iterations*16+i]
            iii. input[i]-=Fullinput[iterations*16+i]
             iv. i+=1
         l. until i=16
         m. delocal int i = 16
         n. delocal int encryptedText[16]={0}
         o. delocal int input[16]={0}
         p. iterations+=1
   4. until iterations = size(Fullinput) /16
   5. delocal int iterations = size(Fullinput) / 16
```