ΣΥΜΠΙΕΣΗ JAVA BYTECODE

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Δημήτριο Σαούγκο

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Οκτώβριος 2006

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

ΠΕΡΙΛΗΨΗ

Δημήτριος Σαούγκος του Γεωργίου και της Ανθούλας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος, 2004. Η συμπίεση του Java Bytecode. Επιβλέποντας: Γεώργιος Μανής.

Οι τεχνικές συμπίεσης του Bytecode της Java με την χρήση των patterns στηρίζονται στην αναγνώριση όμοιων τμημάτων κώδικα που εμφανίζονται παραπάνω από μία φορά. Κάθε εμφάνιση του συγκεκριμένου τμήματος στον κώδικα αντικαθίσταται από μια συγκεκριμένη εντολή ενώ το τμήμα αυτό του κώδικα ορίζει ένα pattern που χρησιμοποιείται ως επέκταση του συνόλου των εντολών με την νέα εντολή που το συμβολίζει. Εναλλακτικά η νέα αυτή εντολή μπορεί απλά να χρησιμοποιηθεί ως δείκτης σε μια θέση ενός λεξικού που περιέχει όλα τα patterns και το οποίο χρησιμοποιείται για την αποσυμπίεση του κώδικα. Στην εργασία αυτή χρησιμοποιούμε μια τεχνική για εύρεση patterns που μας επιτρέπει να εντοπίζουμε patterns με οποιοδήποτε μήκος και με οποιοδήποτε πλήθος από μπαλαντέρ στην θέση κάποιων εντολών ή ορισμάτων εντολών μέσα στο pattern. Αυτήν την τεχνική την εφαρμόζουμε πειραματικά σε μια υλοποίηση του προτύπου MIDP της Java, το οποίο είναι ένα περιβάλλον για την ανάπτυξη εφαρμογών Java για φορητές συσκευές.

EXTENDED ABSTRACT IN ENGLISH

Saougos Dimitrios. MSc, Computer Science Department, University of Ioannina, Greece. July 2004. The compression of the Java Bytecode. Thesis Supervisor: Georgios Manis.

The Java language has become a dominant means for the realization of embedded and mobile computing environments. The main feature of Java that led to the previous is its *portability*. More specifically, the compilation of Java applications results in device independent code, generated in terms of a standard format, called Java *bytecode*. The Java bytecode can then execute on top of different device-specific Java Virtual Machines (JVMs), which take charge of translating the bytecode into device-specific machine code. The memory limitations imposed by embedded and mobile devices certainly constrain the set of applications that may possibly execute on top of them. Confronting the aforementioned issue fosters research towards two orthogonal directions. The first one concerns the reduction of the physical size and cost of memory chips, while the second one involves reducing the size of the code of embedded applications. Advances in both of the previous research directions are equally valuable. No matter how much we increase the amount of available memory, there will always be more demanding applications. Similarly, even if we manage to diminish the size of embedded and mobile applications we may always require to concurrently execute as many of them as possible.

Lots of significant research efforts have already been done towards the generation of compressed code . However, most of these efforts involve the compression of either machine or assembly code. Amongst the few approaches that focus on the case of Java bytecode we have the ones who examine bytecode compression relying on Huffman codes and Markov chains Moreover there have been approaches based on the discovery of instruction sequences that occur more than once within the Java bytecode. Each sequence of instructions defines a *pattern*. Each pattern occurrence is substituted by a single instruction that is often called a *macro*. So far no attempt has been made to use a more generic form of pattern (which we call a parametric one). The main contribution of this thesis is to assess the use of parametric patterns in the context of compressing the bytecode. For that to be achieved we use a well known clustering method called agglomerative clustering algorithm which we alter to fit our needs. Our case study is the MIDP platform which is the main tool used to create software for embedded and portable devices. We measure compression ratios achieved as well as the time overhead introduced in the execution of the compressed programs.

The results show that good percentages can be achieved by our method as long as certain settings are maintained while the total overhead may reach a percentage of 40% which is acceptable given the fact that our method can reduce the size of a bytecode by even more than 25% and that the compressed bytecode can be executed without the need of prior decompression.

ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

1.1. Εισαγωγή

Η γλώσσα προγραμματισμού Java έχει γίνει το κυρίαρχο μέσο για την υλοποίηση των ενσωματωμένων (embedded) και κινητών υπολογιστικών συστημάτων. Το κύριο χαρακτηριστικό της Java που την οδήγησε σε αυτό είναι η μεταφερσιμότητά της. Πιο συγκεκριμένα, Η μεταγλώττιση μιας εφαρμογής Java οδηγεί στην δημιουργία κώδικα που είναι ανεξάρτητος από οποιαδήποτε συσκευή ακολουθώντας ένα συγκεκριμένο πρότυπο. Ο κώδικας αυτός (που ονομάζεται *bytecode*) στην συνέχεια μπορεί να εκτελεστεί από διάφορες αρχιτεκτονικές συστημάτων μεταφράζοντας τον κώδικα αυτό σε κώδικα που μπορεί να εκτελεστεί από το εκάστοτε σύστημα.

Οι περιορισμοί μνήμης που επιβάλλονται από τις ενσωματωμένες και κινητές συσκευές μειώνουν και το εύρος των εφαρμογών που μπορούν να εκτελεστούν από αυτά. Η αντιμετώπιση αυτού του προβλήματος οδηγεί σε δύο διαφορετικές ερευνητικές κατευθύνσεις. Η πρώτη ασχολείται με την μείωση του μεγέθους και του κόστους των τσιπ, ενώ η δεύτερη ερευνά την μείωση του μεγέθους του κώδικα των εφαρμογών που απευθύνονται σε κινητά συστήματα. Η πρόοδος και στις δύο κατευθύνσεις είναι εξ ίσου σημαντική. Αφ’ ενός όσο και να μεγαλώσουμε το ποσό της διαθέσιμης μνήμης σε ένα σύστημα, πάντα θα εμφανίζονται πιο απαιτητικές

εφαρμογές, αφ’ εταίρου όσο και να μειώσουμε το μέγεθος μιας εφαρμογής, πάντα θα υπάρχει ανάγκη για την ταυτόχρονη εκτέλεση όλο και περισσότερων εφαρμογών.

Πολλή έρευνα έχει ήδη γίνει στον τομέα της συμπίεσης του κώδικα [1]. Οι πιο πολλές όμως προσπάθειες στοχεύουν στην συμπίεση είτε του κώδικα μηχανής είτε του συμβολικού κώδικα (assembly). Μεταξύ των λίγων προσεγγίσεων στον τομέα της Java έχουμε αυτές που προτείνονται στα [2], [3] και [4]. Στο [2] οι συγγραφείς εξετάζουν διάφορες προσεγγίσεις στην συμπίεση του ενδιάμεσου κώδικα χρησιμοποιώντας κώδικες Huffman και αλυσίδες Markov. Στο [3] η συμπίεση βασίζεται στην χρήση κανονικών μορφών Huffman και στην παραγωγή γρήγορων αποσυμπιεστών. Στο [4], η προσέγγιση βασίζεται στην εύρεση ακολουθιών εντολών που εμφανίζονται παραπάνω της μιας φοράς μέσα στον κώδικα. Κάθε τέτοια ακολουθία εντολών ορίζει ένα *pattern*. Κάθε εμφάνιση ενός pattern μέσα στον κώδικα αντικαθίσταται από μία και μόνο εντολή που ονομάζεται macro.

Ένα pattern στην ευρύτερη έννοια που μπορεί να έχει, παρουσιάζει τα εξής χαρακτηριστικά:

- μπορεί να έχει οποιοδήποτε μήκος.
- Μπορεί να περιέχει μπαλαντέρ στην θέση οποιασδήποτε εντολής ή ορίσματος εντολής.

Στο εξής ο όρος παραμετρικά patterns θα αναφέρεται στα patterns που περιέχουν οποιοδήποτε πλήθος μπαλαντέρ. Αντίστοιχα θα χρησιμοποιείται ο όρος μη παραμετρικά για να αναφερθεί στα patterns που δεν περιέχουν μπαλαντέρ.

Μέχρι στιγμής, οι υπάρχουσες προσεγγίσεις στην συμπίεση του bytecode δεν αξιοποιούν την πλήρως γενικευμένη μορφή ενός pattern. Το γεγονός αυτό τονίζεται στο [4] όπου οι συγγραφείς τονίζουν την ανάγκη για αξιοποίηση ποιο εξελιγμένων τεχνικών για την εύρεση γενικευμένων patterns.

1.2. Στόχοι

Ο συνολικός στόχος της διατριβής είναι να μελετηθεί η χρήση μιας τέτοιας τεχνικής για την εύρεση παραμετρικών patterns με οποιοδήποτε μήκος στα πλαίσια της συμπίεσης του bytecode της Java. Συγκεκριμένα, οι επιμέρους στόχοι μας είναι:

- Να μετατρέπουμε έναν γνωστό αλγόριθμο αναγνώρισης προτύπων που ονομάζεται agglomerative clustering [5], [6], ώστε να μπορεί να χρησιμοποιηθεί για τον εντοπισμό παραμετρικών και μη-παραμετρικών patterns μέσα στον bytecode. Η τεχνική μπορεί να παράξει patterns με διάφορα μήκη και με διαφορετικό πλήθος από μπαλαντέρ.
- Να αξιολογούμε τα πλεονεκτήματα και τους περιορισμούς της παραπάνω τεχνικής σε διάφορα σενάρια που στοχεύουν στην συμπίεση του MIDP, ενός πρότυπου περιβάλλοντος της Java που υποστηρίζει την ανάπτυξη εφαρμογών για κινητές συσκευές. Το κυριότερο πλεονέκτημα της τεχνικής με τα παραμετρικά patterns είναι ότι επιτρέπει την εύρεση μιας μεγάλης ποικιλίας από patterns που μπορούν να συνδυαστούν για να επιτευχθεί η μεγαλύτερη συμπίεση του κώδικα. Αυτό όμως είναι και ο μεγαλύτερος περιορισμός. Το να ερευνηθεί ποιος είναι ο καλύτερος συνδυασμός από patterns μέσα από ένα πολύ μεγάλο σύνολο patterns είναι μια περίπλοκη διαδικασία. Έχοντας αυτό υπ’ όψη:
    - Χρησιμοποιούμε και συγκρίνουμε δύο ευρετικές μεθόδους για την εύρεση ενός καλού συνδυασμού. Βασιζόμενοι στις δύο αυτές ευρετικές, ερευνούμε την επίπτωση που έχει η χρήση των παραμετρικών patterns στην συμπίεση του bytecode. Για τον σκοπό αυτό συμπιέζουμε το MIDP χρησιμοποιώντας patterns που έχουν μεταβλητό πλήθος μπαλαντέρ, patterns που δεν περιέχουν μπαλαντέρ και patterns που περιέχουν συγκεκριμένο πλήθος μπαλαντέρ και συγκρίνουμε τα αποτελέσματα μεταξύ τους.

    - Επιπλέον μελετούμε τον αντίκτυπο που έχει η αύξηση του μήκους των patterns στην συμπίεση που επέρχεται.

- Τέλος μελετούμε τον επιπλέον χρόνο που απαιτεί η αποσυμπίεση του συμπιεσμένου κώδικα κατά την εκτέλεση του προγράμματος.

1.3. Δομή της Διατριβής

Η διατριβή περιέχει 6 κεφάλαια: Το Κεφάλαιο 2 περιέχει το υπόβαθρο που χρειάζεται για την κατανόηση κάποιων όρων καθώς επίσης και την σχετική δουλειά που έχει γίνει στον τομέα μαζί με μια μικρή περιγραφή της μεθόδου που εμείς χρησιμοποιούμε. Στο κεφάλαιο 3 παρουσιάζονται δύο μέθοδοι για την εύρεση των patterns καθώς και ένα παράδειγμα που συγκρίνει τις δύο αυτές ιδέες. Στο κεφάλαιο 4 αναλύεται η κωδικοποίηση και αποκωδικοποίηση της μεθόδου μας. Στο κεφάλαιο 5 παρουσιάζονται τα πειραματικά αποτελέσματα που είχαμε ενώ στο κεφάλαιο 6 παρουσιάζεται μια μικρή συζήτηση και συμπεράσματα επί της όλης μεθόδου.

ΚΕΦΑΛΑΙΟ 2. ΥΠΟΒΑΘΡΟ ΚΑΙ ΣΧΕΤΙΚΗ ΔΟΥΛΕΙΑ

2.1. Υπόβαθρο

Η γλώσσα Java δημιουργήθηκε από την εταιρεία SUN ως μια γλώσσα που να μπορεί να εκτελεστεί σε οποιοδήποτε υπολογιστικό σύστημα την υποστηρίζει χωρίς την ανάγκη μεταγλώττισης (recompiling). Κατά βάση είναι διερμηνευόμενη (interpreted) αλλά πρώτα ο αρχικός κώδικας πρέπει να περάσει από μια διαδικασία μεταγλώττισης. Το αποτέλεσμα που παράγεται από την μεταγλώττιση είναι ένας κώδικας μηχανής Java που ονομάζεται bytecode και είναι native στα τσιπ Java. Αυτό σημαίνει ότι αν υπάρχει κάποιο hardware που να μπορεί να εκτελεί την γλώσσα μηχανής Java θα μπορούσε απευθείας να εκτελέσει τον bytecode. Στους υπολογιστές λοιπόν που δεν έχουν κάποιο τέτοιο κύκλωμα, η όλη λειτουργία του συστήματος εξομοιώνεται από κάποια εφαρμογή που αναλαμβάνει να φορτώσει τα δεδομένα του αρχείου που περιέχει τον bytecode και να τον εκτελέσει. Στην περίπτωση αυτή λοιπόν ο bytecode απλά εκτελείται με διερμηνεία (interpretation) και η εφαρμογή που

εκτελεί το πρόγραμμα Java ονομάζεται Εικονική Μηχανή Java (Java Virtual Machine, JVM σε συντομογραφία).

Ο bytecode βρίσκεται αποθηκευμένος σε ένα αρχείο τύπου class. Η μορφή του αρχείου αυτού είναι σαφώς ορισμένη από την SUN και περιέχει όλες τις πληροφορίες που περιέχονται σε μια κλάση του αρχικού κώδικα της Java. Σε αυτά τα αρχεία επίσης αποθηκεύονται πληροφορίες για τα exceptions, τα πεδία της κλάσης και οτιδήποτε άλλο χρειάζεται για να εκτελεστεί σωστά η κλάση. Ένα μεγάλο μέρος της πληροφορίας σε κάθε αρχείο class αποτελεί το constant pool. Αυτό στην ουσία είναι ένας πίνακας (array) που έχει όλες τις σταθερές που δηλώνονται σε μια κλάση. Ακολουθεί επιγραμματικά όλη η δομή του αρχείου καθώς και πληροφορίες επί αυτών (σημειώνεται ότι οι συμβολισμοί u1, u2 και u4 αναφέρονται σε ακέραιους αριθμούς μήκους 1, 2 και 4 bytes αντίστοιχα που χρησιμοποιούν την στοίχιση τύπου big endian (δηλαδή το πιο σημαντικό byte στην αρχή)):

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
```

```
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Το πεδίο magic είναι ο λεγόμενος «μαγικός αριθμός» και ισούται με την τιμή 0xCAFEBABE. Χρησιμοποιείται ως αναγνωριστικό του συγκεκριμένου τύπου αρχείου. Τα πεδία minor και major version δηλώνουν την έκδοση του συγκεκριμένου class file. Η τιμή του constant_pool_count ισούται με το πλήθος των καταχωρήσεων στον πίνακα του constant pool αυξημένη κατά 1. Το constant pool καθαυτό είναι ένας πίνακας από δομές που αναπαριστούν αλφαριθμητικά, ονόματα κλάσεων και interfaces ονόματα πεδίων και λοιπών σταθερών στις οποίες αναφέρονται οι υπόλοιπες δομές και υποδομές του αρχείου class. Το πεδίο access flags δηλώνει τον τύπο της κλάσης (αν είναι private, public, final, abstract ή interface). Τα πεδία this_class και super_class είναι δείκτες στον πίνακα του constant pool και αναφέρονται στην παρούσα κλάση και στην κλάση από την οποία κληρονομήθηκε η παρούσα αντίστοιχα. Το πεδίο interfaces_count περιέχει το πλήθος των interfaces που υπάρχουν στην κλάση ενώ ο πίνακας interfaces περιέχει τα interfaces. Το πεδίο fields_count περιέχει το πλήθος των πεδίων με τον πίνακα fields να περιέχει τα πεδία καθαυτά. Το πεδίο methods_count περιέχει το πλήθος των μεθόδων με τον πίνακα methods να περιέχει τις μεθόδους. Τέλος τα πεδία attributes_count και attributes περιέχουν το πλήθος των attributes και τα attributes αντίστοιχα.

Εμείς θα σταθούμε κυρίως στο πεδίο methods. Κάθε μέθοδος αποτελείται από την εξής δομή:

```
method_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Το πεδίο access_flags ορίζει τις ιδιότητες της μεθόδου (δηλαδή αν είναι public, private, protected, static, final, synchronized, native και abstract). Το πεδίο name_index είναι ένας δείκτης στο constant pool που δείχνει το όνομα της μεθόδου ή τα ειδικά ονόματα <init> και <clinit>. Το πεδίο descriptor_index είναι ένας δείκτης στο constant pool που δείχνει στον descriptor της μεθόδου και είναι τύπου αλφαριθμητικού. Τέλος τα πεδία attributes_count και attributes περιέχουν το πλήθος των attributes και τα attributes καθαυτά.

Μία μέθοδος μπορεί να περιέχει πολλά attributes (όπως για παράδειγμα πληροφορίες για τα exceptions), ένα εκ των οποίων είναι το attribute Code που περιέχει πληροφορίες για τον κώδικα της μεθόδου. Το Code attribute είναι ως εξής:

```
Code_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
```

```
        {       u2 start_pc;
                u2 end_pc;
                u2  handler_pc;
                u2  catch_type;
        }       exception_table[exception_table_length];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Τα δύο πρώτα πεδία είναι τα ίδια σε κάθε attribute που κωδικοποιείται στο class αρχείο. Το attribute_name_index δείχνει σε ένα αλφαριθμητικό στο constant pool όπου στην περίπτωση του Code attribute υπάρχει η λέξη “Code”. Το πεδίο attribute_length περιέχει την πληροφορία για το μήκος όλου του attribute (σε bytes). Από όλα τα υπόλοιπα πεδία, εμείς στεκόμαστε στο ζεύγος πεδίων code_length και code. Το μεν πρώτο περιέχει το μήκος του κώδικα της συγκεκριμένης μεθόδου ενώ το πεδίο code είναι ένας πίνακας μήκους code_length που περιέχει τον bytecode της συγκεκριμένης μεθόδου.

Γνωρίζοντας όλες αυτές τις πληροφορίες μπορούμε εύκολα να δημιουργήσουμε έναν disassembler για τα αρχεία class ο οποίος να εντοπίζει τον bytecode μέσα στο αρχείο και να τον εξάγει σε μια εύκολα χρησιμοποιήσιμη μορφή.

Επειδή η Java ως γλώσσα μπορεί να μεταφέρεται μέσω του Internet και να εκτελείται μέσα από web browsers, το specification της γλώσσας είναι αρκετά αυστηρό και ορίζει όλες τις λεπτομέρειες που απαιτούνται για την εκτέλεση ενός προγράμματος σε όλους τους υπολογιστές που τρέχουν μια JVM (από την σειρά αποθήκευσης των bytes σε ένα καταχωρητή, μέχρι το πώς αποθηκεύεται ένας αριθμός κινητής υποδιαστολής).

Η Java ως μηχανή είναι μηχανή στοίβας. Αυτό σημαίνει ότι όλες οι πράξεις γίνονται πάνω στην στοίβα (δηλαδή για να γίνει η πράξη a = b + c απαιτείται να φορτωθεί στην στοίβα το b, έπειτα το c, να αφαιρεθούν από την στοίβα και το αποτέλεσμα της πρόσθεσης να τοποθετηθεί στην κορυφή και τέλος από την κορυφή της στοίβας να αποθηκευτεί στο a. Κάθε εντολή του bytecode είναι της μορφής

Opcode operand1 operand2 ….

Όπου opcode ένας κωδικός αριθμός από το 0 έως το 255 (ένα byte) που αντιστοιχεί και σε κάποια εντολή από το set εντολών του κώδικα μηχανής Java. Έπειτα από κάθε εντολή ακολουθεί και μια σειρά από ορίσματα (operands) για την εντολή αυτή μήκους 1 byte το κάθε ένα. Η εκτέλεση του κώδικα γίνεται στο main loop της JVM με πολύ απλό τρόπο: Για κάθε opcode που διαβάζεται γίνεται fetch των ορισμάτων που χρειάζονται και η εντολή εκτελείται. Αυτό συνεχίζεται μέχρι το τέλος του προγράμματος.

2.2. Σχετική Δουλειά

Γενικά έχει υπάρξει πολύ μεγάλη ερευνητική δραστηριότητα πάνω στον τομέα της συμπίεσης κώδικα. Κατ’ αρχάς ο κώδικας μπορεί απλά να θεωρηθεί ως απλά δεδομένα που δεν διαφέρουν σε τίποτα άλλο από μια απλή σειρά από δυαδικά αρχεία. Αυτό σημαίνει ότι όλες οι παραδοσιακές τεχνικές συμπίεσης δεδομένων (Huffman κ.τ.λ.) βρίσκουν εφαρμογή στον τομέα αυτό. Αν όμως ξεφύγουμε από αυτό το γενικό πλάνο και δούμε ποιες ιδιαιτερότητες έχει ο κώδικας ενός προγράμματος μπορούμε να δούμε συγκεκριμένες τεχνικές που βρίσκουν εφαρμογή πάνω στο συγκεκριμένο θέμα. Επί αυτού υπάρχουν δύο μεγάλες κατηγορίες. Στην μια εντάσσονται οι αλγόριθμοι και οι τεχνικές που στοχεύουν σε μέγιστη συμπίεση κώδικα (για γρήγορη μεταφορά του μέσω δικτύου) που αδιαφορούν για τον χρόνο της αποσυμπίεσης ενώ στην άλλη εντάσσονται κατηγορίες που προσπαθούν να κρατήσουν ένα καλό

ποσοστό συμπίεσης αλλά το συμπιεσμένο αποτέλεσμα να μπορεί να εκτελεστεί σε πραγματικό χρόνο.

Στο [7] αναφέρεται ότι η συμπίεση του κώδικα οδηγεί σε μικρότερα μήκη σελίδων μνήμης πράγμα το οποίο μπορεί να οδηγήσει σε αύξηση της απόδοσης ενός συστήματος. Το παραπάνω ισχύει λόγω του ότι στα σημερινά συστήματα ένα μεγάλο χρονικό διάστημα σπαταλάται (ο επεξεργαστής δεν παράγει ουσιαστικό έργο) λόγω του ότι αναμένονται οι σελίδες μνήμης είτε από τον δίσκο είτε από την μνήμη. Ένας άλλος παράγοντας στον οποίο συμβάλλει η συμπίεση του κώδικα είναι η μεταφορά ενός προγράμματος μέσω δικτύου από τον έναν υπολογιστή στον άλλον. Ειδικά σε περιπτώσεις όπου το εύρος ζώνης (bandwidth) είναι περιορισμένο η αξία της συμπίεσης φαίνεται ακόμα πιο πολύ. Βάσει των παραπάνω δύο σεναρίων λοιπόν, οι τεχνικές συμπίεσης χωρίζονται σε δύο κατηγορίες ανάλογα με το που υπάρχει το bottleneck: στην μετάδοση και στην μνήμη

Στην μετάδοση είναι αναγκαίο να υπάρξει η μέγιστη συμπίεση για να είναι γρήγορη η μεταφορά ενώ έχουμε την πολυτέλεια να αποσυμπιέσουμε τον κώδικα πριν την εκτέλεση. Ο συμπιεσμένος κώδικας που παράγεται ονομάζεται “wire” κώδικας (επειδή χρειάζεται αποσυμπίεση –τοπική ή ολική- για να εκτελεστεί)

Στην μνήμη ο κώδικας (ή τουλάχιστον ένα κομμάτι του που εκτελείται συχνά) πρέπει να βρίσκεται αποθηκευμένος αλλά και να εκτελείται σε συμπιεσμένη μορφή. Επίσης επειδή θα κάνει άλματα (jumps) σε τυχαία σημεία χρειάζεται να έχουμε τυχαία προσπέλαση στα basic blocks

Το JAZZ [8] είναι ένα σύστημα αντίστοιχο του ήδη υπάρχοντος JAR. Το JAR είναι μια μέθοδος για το πακετάρισμα και την συμπίεση πολλών αρχείων class μαζί για ευκολότερη διανομή μέσω του δικτύου. Η δομή του είναι ουσιαστικά ένας μεγάλος

φάκελος μέσα στον οποίο περιέχονται τα επί μέρους αρχεία σε συμπιεσμένη μορφή. Δυστυχώς όμως είναι τέτοια η μορφή και το μέγεθος ενός class file που δεν επιτρέπουν μεγάλα ποσοστά συμπίεσης. Το JAZZ δημιουργήθηκε για να λαμβάνει υπ' όψιν την συνολική πληροφορία που περιέχεται στα αρχεία αυτά αντί να τα συμπιέζει ένα ένα. Με αυτόν τον τρόπο επιτυγχάνει πολύ καλύτερα ποσοστά συμπίεσης από απλά συστήματα τύπου ZIP (όπως είναι το JAR)

Το project Slim Binaries [9] στοχεύει κυρίως στην παραγωγή ενός ενδιάμεσου object code το οποίο όμως να μπορεί να αποσυμπιέζεται και να εκτελείται on the fly. Η βασική ιδέα πίσω από αυτό το project είναι το γεγονός του ότι σήμερα υπάρχουν πολλές διαφορετικές αρχιτεκτονικές επεξεργαστών, οι εταιρείες όμως κατασκευής λογισμικού παράγουν κώδικα για μια συγκεκριμένη αρχιτεκτονική και από εκεί και πέρα αναλαμβάνει ο εκάστοτε επεξεργαστής να το εξομοιώσει στην δική του native πλατφόρμα. Το όνομα Slim binaries προήλθε από τα λεγόμενα "fat binaries" τα οποία είναι κομμάτια object κώδικα αλλά μέσα τους περιέχουν κώδικα για πολλές διαφορετικές αρχιτεκτονικές ώστε να μπορούν να εκτελεστούν natively από κάθε σύστημα. Το Slim Binaries μετά την μετάφραση ενός προγράμματος (και αφού χρησιμοποιήσει τεχνικές πρόβλεψης και συμπίεσης όμοιων τμημάτων κώδικα)
παράγει έναν ενδιάμεσο κώδικα ο οποίος όμως είναι σε μια μορφή που μπορεί να εκτελεστεί πολύ γρήγορα (λόγω πολύ μικρού μεγέθους).

Τέλος όσον αφορά την πρώτη κατηγορία συμπίεσης αξίζει να αναφερθούμε στο [2] και στο [10]. Στο [2] εξετάζονται διάφορες μέθοδοι για την μείωση του μεγέθους των αρχείων class της JAVA ειδικά για προγράμματα που απευθύνονται σε embedded συστήματα. Ανάμεσα σε αυτές αναφέρεται η αλλαγή του format του constant pool (που προσφέρει και τα μεγαλύτερα ποσοστά συμπίεσης) και η αναδιοργάνωση των op codes ενώ καταλήγουν στο συμπέρασμα ότι η συμπίεση είναι μεγαλύτερη σε ένα jar

αρχείο όταν ληφθεί υπ’ όψιν πληροφορία που βρίσκεται σε όλα τα αρχεία απ’ ότι αν κάθε αρχείο συμπιέζονταν ανεξάρτητα από το άλλο.

Στο [10] ο Pugh κινείται με παρόμοιο τρόπο με τις προηγούμενες μεθοδολογίες και εφαρμόζει 3 διαφορετικές τεχνικές για την μείωση του μεγέθους των αρχείων JAR. Ο ίδιος δίνει μεγαλύτερη βάση στην ταχύτητα μετάδοσης παρά στον χρόνο αποσυμπίεσης που θα χρειαστεί το αρχείο. Οι τεχνικές που χρησιμοποιεί έχουν αναφερθεί και στα προηγούμενα papers:
Αναδιοργάνωση της πληροφορίας στα αρχεία class (αλλαγή της σειράς στο constant pool, διαγραφή debugging πληροφοριών κτλ..)
Αλλαγή της σειράς με την οποία υπάρχουν οι πληροφορίες σε ένα αρχείο ώστε να δοθεί στο gzip η ευκαιρία να ανακαλύψει πιο πολλά patterns και να αποφέρει μεγαλύτερη συμπίεση
Ομαδοποίηση της κοινής πληροφορίας των constant pool των διαφόρων αρχείων class που βρίσκονται σε ένα αρχείο JAR ώστε να μειωθεί το πλήθος των πλεοναζόντων δεδομένων που θα αποσταλούν. Αυτό γίνεται εντοπίζοντας την κοινή πληροφορία που βρίσκεται στα constant pools των αρχείων και αντικαθιστώντας την με κάποιο macro.

Στην δεύτερη κατηγορία εντάσσονται τεχνικές συμπίεσης που στοχεύουν στην δημιουργία συμπιεσμένου κώδικα ο οποίος όμως μπορεί να διερμηνευτεί (είναι δηλαδή interpretable) χωρίς να χρειάζεται να αποσυμπιεστεί πλήρως για να εκτελεστεί. Οι διάφορες τεχνικές πάνω σε αυτήν την γενική ιδέα βασίζονται είτε στην κωδικοποίηση (για παράδειγμα Huffman) είτε στην αναγνώριση των patterns μέσα στον κώδικα και αντικατάσταση αυτών με δείκτες σε κάποιο λεξικό (dictionary).

Η κωδικοποίηση Huffman αντιστοιχίζει την μικρότερη αλληλουχία από bits στα σύμβολα που έχουν την μεγαλύτερη συχνότητα εμφάνισης σε ένα κείμενο. Με τον

τρόπο αυτό μπορεί να αποφέρει πολύ καλά ποσοστά συμπίεσης στο κείμενο στο οποίο εφαρμόζεται. Τεχνικές που βασίζονται σε αυτού του είδους την κωδικοποίηση γενικά έχουν δεχθεί κριτική λόγω του ότι εμφανίζουν αυξημένη πολυπλοκότητα κατά την αποσυμπίεσή τους, παρόλα αυτά στην εργασία [3] παρουσιάζονται μέθοδοι για την κατασκευή γρήγορων διερμηνέων για κωδικοποιημένα κείμενα Huffman. Οι εργασίες [11] και [12] χρησιμοποιούν κωδικοποίηση Huffman αλλά και αριθμητική κωδικοποίηση αντίστοιχα για να επιτύχουν την επιθυμητή συμπίεση. Ειδικά στο [11] η εκτέλεση του προγράμματος γίνεται από εντολές που αποσυμπιέζονται μέσα στην cache και έτσι επιτυγχάνεται σχετικά καλός χρόνος εκτέλεσης.

Οι τεχνικές συμπίεσης που στηρίζονται στην εύρεση patterns παράγουν συμπιεσμένο κώδικα ο οποίος όμως μπορεί να εκτελεστεί χωρίς να χρειάζεται να γίνει πρώτα η αποσυμπίεση του (ολόκληρου του κώδικα ή κατά τμήματα) και μετά να εκτελεστεί. Η κεντρική ιδέα στην οποία στηρίζονται είναι ο εντοπισμός αλληλουχιών κώδικα που βρίσκονται σε πολλαπλά σημεία μέσα στον κώδικα. Κάθε τέτοια αλληλουχία ονομάζεται pattern. Τα patterns συνήθως αποθηκεύονται σε κάποιο λεξικό ενώ οι εμφανίσεις τους στον αρχικό κώδικα αντικαθίστανται από συγκεκριμένα opcodes που ονομάζονται macros. Συνήθως ένα macro έχει μήκος όσο και μια εντολή (στην περίπτωση της Java ένα byte) και λειτουργεί ως δείκτης (index) μέσα στο λεξικό (dictionary).

Συγκεκριμένα στο [13], οι συγγραφείς ερευνούν την χρήση των patterns (οι οποίοι τα ονομάζουν “repeats” στην συγκεκριμένη περίπτωση) στην συμπίεση κώδικα μηχανής που αναφέρεται σε μηχανές τύπου RISC. Ποιο συγκεκριμένα κωδικοποιούν τις εντολές μαζί με τα ορίσματά τους σε κωδικό-σύμβολα και την συνέχεια αναζητούν repeats μέσα στο κείμενο από τα σύμβολα αυτά. Τα repeats επιτρέπεται να έχουν οποιοδήποτε μήκος αλλά και μπαλαντέρ (wildcards) μέσα τους. Η διαδικασία εύρεσης των repeats γίνεται με την δημιουργία ενός δέντρου (suffix-tree) ενώ η

διαδικασία επιλογής των καταλληλότερων repeats γίνεται ταξινομώντας κάθε repeat με βάση την αναμενόμενη συμπίεση που μπορεί να προσφέρει το κάθε ένα. Με παρόμοια λογική κινείται και το CodePack Compressor της IBM, που εφαρμόζεται στα PowerPc, μόνο που χρησιμοποιεί patterns χωρίς μπαλαντέρ αλλά με απεριόριστο μήκος.

Οι συγκεκριμένες εργασίες πηγάζουν από τις ιδέες που προτείνονται στο [14]. Στη συγκεκριμένη εργασία περιγράφεται η ιδέα του να χρησιμοποιεί κανείς αποθηκευμένα patterns σε κάποιο λεξικό και να τα κάνει fetch στην μνήμη όταν αυτά χρειάζονται. Επίσης αναφέρεται ότι οι μέθοδοι που χρησιμοποιούν στατιστικές μεθόδους συμπίεσης (όπως για παράδειγμα μέθοδοι τύπου Huffman) μπορούν πάντα να προσφέρουν την ίδια ή και καλύτερη συμπίεση από οποιαδήποτε μέθοδο που στηρίζεται σε dictionary, παρόλα αυτά όμως είναι πιο δύσκολη και πιο περίπλοκη από υπολογιστικής απόψεως η αποσυμπίεση και αυτό επειδή η στατιστική συμπίεση δεν προσφέρει κωδικοποιημένο αποτέλεσμα που να είναι στοιχισμένο με την μνήμη. Έτσι ο υπολογισμός του offset κάθε φορά μπορεί να είναι χρονοβόρος.

Στο [15] χρησιμοποιούνται πολλές διαφορετικές τεχνικές για την επίτευξη της συμπίεσης. Χρησιμοποιείται μια συνάρτηση «αποτυπώματος» για την αναγνώριση «όμοιων» basic blocks και έτσι χρησιμοποιούνται παραμετρικά patterns (δηλαδή patterns με μπαλαντέρ) για την παραγοντοποίηση του κώδικα (factorizing). Επίσης χρησιμοποιούν πολλές τεχνικές βελτιστοποίησης που αξιοποιούν οι σημερινοί compilers (ανάλυση του γράφου των basic blocks για την απαλοιφή πλεονάζοντα κώδικα, εύρεση όμοιων τμημάτων κώδικα πριν και μετά τα basic blocks και περαιτέρω σύμπτυξη τους σε διαδικασίες και τέλος χρήση ιδιοτήτων συγκεκριμένων αρχιτεκτονικών για ακόμα καλύτερα αποτελέσματα)

Οι δημιουργοί του SQUEEZE++ [16] προχωράνε ένα βήμα παραπέρα πάνω στην όλη ιδέα. Κατ' αρχάς εφαρμόζουν τις τεχνικές τους σε ένα βασικό κομμάτι της διαδικασίας της μετάφρασης που παραμένει (σύμφωνα με τους συγγραφείς) ανεκμετάλλευτο γενικά: Τον linker. Στο επίπεδο σύνδεσης λοιπόν εφαρμόζουν την ιδέα της ομοιότητας κώδικα σε επίπεδο διαδικασιών αλλά και σε επίπεδο κώδικα, basic blocks, αλλά και σε επίπεδο sub blocks. Λόγω αντικειμενοστρέφειας εμφανίζεται το εξής φαινόμενο όταν χρησιμοποιούνται templates: Δημιουργούνται πολλές διαδικασίες που είναι ίδιες τελείως. Λόγω αυτού το SQUEEZE++ χρησιμοποιεί μόνο μια πολλές φορές. Επίσης αναζητούνται διαδικασίες που μοιάζουν μεταξύ τους και αντικαθίστανται από μια η οποία είναι παραμετρική και επίσης αναζητούνται όμοια basic blocks και sub blocks τα οποία παραγοντοποιούνται (factorized).

Ο krinke στο [17] αναφέρει μια μέθοδο για την εύρεση όμοιων κομματιών κώδικα αναζητώντας όμοια υπογραφήματα στο γράφο εξαρτήσεων (dependency graph) ώστε να αξιοποιηθούν για την δημιουργία παραμετρικών διαδικασιών. Παρόμοια με αυτήν την εργασία κινείται και το σύστημα που δημιουργήθηκε στο [20]. Οι συγγραφείς ανέπτυξαν ένα σύστημα ώστε να μπορεί να εντοπίζει όμοια «δύσκολα» τμήματα κώδικα και να τα αντικαθιστά με διαδικασίες.

Στο BRISC [7] οι δημιουργοί χρησιμοποιούν παραμετρικά patterns αλλά με σταθερό μήκος 2. Τα patterns μπορούν να περιέχουν μπαλαντέρ στην θέση των ορισμάτων των εντολών. Παρόμοια κινείται και η εργασία [18] στην οποία αναγνωρίζονται ομάδες εντολών που μοιάζουν μεταξύ τους. Αυτές αντικαθίστανται από μια παραμετρική συνάρτηση η οποία λειτουργεί για όλες τις περιπτώσεις που αντικαθιστά χρησιμοποιώντας κάποια flags στον επεξεργαστή. Η εργασία αυτή έγινε πάνω στην αρχιτεκτονική του ARM.

Στο [19] παρουσιάζεται μια διαφορετική ιδέα. Δημιουργήθηκε ένα σύστημα το οποίο παίρνει ως είσοδο την γραμματική μιας γλώσσας και μια σειρά από προγραμμάτων – δειγμάτων (τα οποία αποτελούν το σύνολο εκπαίδευσης). Μετά την κατάλληλη επεξεργασία παραλλάσσει την αρχική γραμματική με τέτοιο τρόπο ώστε τα συγκεκριμένα προγράμματα να μπορούν να εκφρασθούν με λιγότερη πληροφορία και άρα επιτυγχάνει συμπίεση. Χρησιμοποιώντας την νέα γραμματική λοιπόν σε όσα προγράμματα επιθυμεί ο χρήστης παίρνει μια στατιστικά σημαντική μείωση στο μέγεθος (ανάλογα πάντα με το τι προγράμματα – δείγματα δόθηκαν για εκπαίδευση του συστήματος). Αυτόματα επίσης παράγεται και ο νέος μεταφραστής για αυτήν την γραμματική.

Αν δούμε πιο συγκεκριμένα τον τομέα του bytecode της Java, στο [4] οι συγγραφείς προτείνουν μια τεχνική που χρησιμοποιεί μη παραμετρικά patterns (δηλαδή χωρίς την χρήση μπαλαντέρ) οποιουδήποτε μήκους. Συγκεκριμένα, χρησιμοποιούν μια ευρετική τεχνική για την εύρεση των τμημάτων του κώδικα που είναι ίδια, προσέχοντας παράλληλα για το αν ξεφεύγουν από τα όρια των basic blocks. Από εκεί και πέρα επιλέγουν τα patterns που οδηγούν σε ένα καλό αποτέλεσμα και δημιουργούν το λεξικό καθώς και τα macros στον κώδικα. Η κύρια συνεισφορά της συγκεκριμένης εργασίας είναι ότι δεν χρησιμοποιούν απευθείας την έννοια του λεξικού. Αντίθετα εξειδικεύουν την JVM με ποιο σύνθετες εντολές που στην ουσία εκτελούν τις εντολές του κάθε pattern. Με τον τρόπο αυτό τα patterns αντικαθίστανται από opcodes των νέων «εντολών» και έτσι μειώνεται το μέγεθος του αρχικού κώδικα. Επίσης προτείνεται και μια επέκταση του java class file format ώστε να μπορεί να αποθηκεύει και να υποστηρίζει macros.

Η συγκεκριμένη εργασία αποτελεί φυσική εξέλιξη της παραπάνω ιδέας. Εφαρμόζοντας τεχνικές από την θεωρία της αναγνώρισης προτύπων, εντοπίζουμε παραμετρικά patterns οποιουδήποτε μήκους. Αυτά με τη σειρά τους αποθηκεύονται

στο λεξικό αφού πρώτα έχει επιλεγεί ένας καλός συνδυασμός αυτών (με την χρήση ευρετικών μεθόδων). Προς το παρόν χρησιμοποιείται στον κώδικα που βρίσκεται μέσα στα basic blocks αλλά μπορεί κάλλιστα να αξιοποιηθεί και σε ολόκληρες μεθόδους χωρίς ιδιαίτερες αλλαγές (Μια μέθοδος μπορεί να περιέχει πολλά basic blocks). Θεωρώντας όλες τις τεχνικές για την εύρεση παραμετρικών patterns που αναφέρθηκαν σε αυτό το κεφάλαιο, η συγκεκριμένη προσέγγιση είναι πιο λεπτομερής μιας και δεν αναζητούνται ομοιότητες μεταξύ ολόκληρων basic blocks (η διαδικασιών). Αντιθέτως η αναζήτηση για όμοια τμήματα κώδικα γίνεται μέσα στον κώδικα των basic blocks. Για την εύρεση αυτή αξιοποιείται ο agglomerative clustering algorithm [5], [6].

2.3. Γενική Περιγραφή της Μεθόδου

Η όλη μέθοδος αποτελείται από τα παρακάτω βήματα:

- Εύρεση των patterns μέσα στον αρχικό κώδικα. Αυτό περιλαμβάνει 3 βήματα.
- Προεπεξεργασία.
- Disassembly του κώδικα.
- Εύρεση των basic clocks.
- Δημιουργία αλφαριθμητικού (string) εισόδου για τον αλγόριθμο εύρεσης των patterns.
- Εύρεση των patterns χρησιμοποιώντας τον agglomerative clustering αλγόριθμο.
- Μεταεπεξεργασία: Μετατροπή της εξόδου του agglomerative σε μια μορφή που είναι εύκολα αξιοποιήσιμη και αναγνώσιμη και στην συνέχεια φιλτράρισμα του συνόλου των patterns. Το φιλτράρισμα έχει ως στόχο να διαγράψει από το σύνολο τα patterns που δεν προσφέρουν συμπίεση στον κώδικα ώστε να κάνει την μετέπειτα διαδικασία της διαλογής ποιο εύκολη.

- Εύρεση του συνδυασμού των patterns που επιφέρει την μεγαλύτερη συμπίεση στον κώδικα. Αν το πλήθος των patterns είναι αρκετά μικρό(το πολύ μέχρι 20) απλά κάνουμε εξαντλητική αναζήτηση στο σύνολο όλων των διαθέσιμων συνδυασμών. Αν όχι τότε χρησιμοποιούμε μια από τις δύο ευρετικές που έχουν δημιουργηθεί για τον σκοπό αυτό.
- Εφαρμογή του βέλτιστου συνδυασμού πάνω στον κώδικα για να πάρουμε το τελικό συμπιεσμένο αποτέλεσμα.

ΚΕΦΑΛΑΙΟ 3. Η ΕΥΡΕΣΗ ΤΩΝ PATTERNS

3.1. Ο Agglomerative Clustering Αλγόριθμος

Στο κεφάλαιο αυτό περιγράφεται λεπτομερώς η μέθοδος εύρεσης των patterns με την χρήση του agglomerative clustering (AC) algorithm. Η προσέγγιση αυτή είναι μια ιεραρχική μέθοδος που δημιουργεί σταδιακά μια δενδρική δομή από τα φύλλα προς την ρίζα. Κάθε κόμβος του δέντρου αυτού είναι και στην ουσία ένα pattern.

Ας δούμε όμως κατ' αρχάς ποια είναι η δομή ενός cluster και πώς από αυτήν την δομή εξάγουμε πληροφορία για το pattern που κωδικοποιείται από το cluster αυτό. Έστω ότι έχουμε 4 αλφαριθμητικά (strings) τα οποία περιλαμβάνονται στο cluster μήκους 5 χαρακτήρων το καθένα: ABCDE, ABBDB, ABCDD, ABCDE. Με αυτά τα strings δημιουργείται ένας δυσδιάστατος πίνακας:

A	B	C	D	E
A	B	B	D	B
A	B	C	D	D
A	B	C	D	E

Σε αυτόν τον πίνακα, κάθε γραμμή είναι και ένα από τα strings και κάθε στήλη αντιστοιχεί στο ι-οστό στοιχείο του κάθε string. Θέλοντας λοιπόν να εντοπίσουμε το pattern που παράγεται από αυτόν τον πίνακα, πρέπει για κάθε σύμβολο που υπάρχει στο αλφάβητο μας (A, B, C, D, E) να εντοπίσουμε πόσες φορές εμφανίζεται το σύμβολο αυτό σε κάθε στήλη του πίνακα:

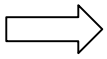
	A	B	C	D	E
1	4	0	0	0	0
2	0	4	0	0	0
3	0	1	3	0	0
4	0	0	0	4	0
5	0	1	0	1	2

Με τον τρόπο αυτό βρήκαμε το πλήθος των εμφανίσεων όλων των συμβόλων σε κάθε στήλη. Οπότε μπορούμε πλέον να βασιστούμε σε αυτό και να υπολογίσουμε την πιθανότητα εμφάνισης του κάθε συμβόλου σε κάθε στήλη, διαιρώντας το πλήθος των εμφανίσεων με το πλήθος των strings που υπάρχουν στο cluster μας (στην συγκεκριμένη περίπτωση 4). Τελικά παίρνουμε τον εξής πίνακα όπου ουσιαστικά είναι και η πληροφορία που αποθηκεύεται στο κάθε cluster:

	A	B	C	D	E
1	1,00	0	0	0	0
2	0	1,00	0	0	0
3	0	0.25	0.75	0	0
4	0	0	0	1,00	0
5	0	0.25	0	0.25	0.5

Ο τρόπος με τον οποίο αξιοποιούμε την πληροφορία αυτή για την παραγωγή του pattern είναι απλός. Σαρώνουμε κάθε γραμμή του πίνακα για να βρούμε το σύμβολο το οποίο έχει και την μεγαλύτερη πιθανότητα εμφάνισης. Για κάθε γραμμή του πίνακα (που αντιστοιχεί και σε μια ι-οστή θέση στο pattern) επιλέγουμε το σύμβολο του οποίου η πιθανότητα εμφάνισης είναι η μεγαλύτερη από τις υπόλοιπες αλλά και που ξεπερνά ένα συγκεκριμένο κατώφλι. Το κατώφλι σχετίζεται με το πόσα strings υπάρχουν και είναι της μορφής (n-k)/n όπου n το πλήθος των strings και k μία παράμετρος που ορίζεται από εμάς. Αν κανένα σύμβολο δεν έχει πιθανότητα εμφάνισης μεγαλύτερη του κατωφλιού για μια συγκεκριμένη γραμμή του πίνακα τότε στο pattern τοποθετείται ο μπαλαντέρ (*). Στο συγκεκριμένο παράδειγμα λοιπόν αν επιλέξουμε ως κατώφλι το 0.9 έχουμε:

	A	B	C	D	E
1	1,00	0	0	0	0
2	0	1,00	0	0	0
3	0	0.25	0.75	0	0
4	0	0	0	1,00	0
5	0	0.25	0	0.25	0.5



A
B
*
D
*

Ενώ αν επιλέξουμε ως κατώφλι το 0.75 με παρόμοιο τρόπο παίρνουμε το

A	B	C	D	*

Έχοντας δει πώς μπορούμε να παράξουμε το pattern από την πληροφορία που παρέχει ένα cluster προχωράμε στην περιγραφή του agglomerative clustering αλγορίθμου. Στην βασική του μορφή, ο αλγόριθμος χρειάζεται 3 παραμέτρους εισόδου: το μήκος των pattern που θέλουμε να υπολογιστούν (που από εδώ και πέρα συμβολίζεται με το σύμβολο L), το πλήθος των patterns που θέλουμε να υπολογιστούν (που θα συμβολίζεται με το σύμβολο K), και το κείμενο εισόδου το οποίο είναι στην ουσία κάποιο string.

Ως πρώτο βήμα χωρίζουμε το string εισόδου σε μια συλλογή X_L από substrings μήκους L ολισθαίνοντας ένα παράθυρο μήκους L κατά 1 από την θέση 1 μέχρι την θέση W-L-1 όπου W το μήκος της εισόδου. Με τον τρόπο αυτό παράγονται W-L+1 strings μήκους L το καθένα. Αυτά τα strings αποτελούν και τα αρχικά clusters. Ο αλγόριθμος δρα επαναληπτικά και σε κάθε βήμα επιλέγει τα δύο clusters που «μοιάζουν» πιο πολύ, τα απομακρύνει από το σύνολο των clusters και στην θέση τους τοποθετεί ένα cluster που είναι η ένωσή τους. Σε κάθε βήμα λοιπόν, το πλήθος των

clusters μειώνεται διαρκώς κατά 1. Ο αλγόριθμος σταματά όταν απομείνουν K clusters, οπότε και εξάγει τα K patterns από τα clusters και τα επιστρέφει ως έξοδο.

Πώς όμως γίνεται η επιλογή των δύο πιο «όμοιων» clusters; Έστω κατ' αρχάς ότι με το σύμβολο Ω συμβολίζουμε το αλφάβητό μας, δηλαδή το σύνολο των διαφορετικών συμβόλων που εμφανίζονται μέσα στο string εισόδου. Όπως είδαμε και από το προηγούμενο παράδειγμα κάθε cluster είναι στην ουσία ένας δυσδιάστατος πίνακας L x |Ω|. Έστω ότι ο πίνακας αυτός συμβολίζεται με θ οπότε κάθε στοιχείο του θ[m, l] ισούται με την πιθανότητα το σύμβολο m να εμφανιστεί στην θέση l του pattern. Ποιο φορμαλιστικά, $\theta[m, l] = n_{ml} / n$, με n_{ml} να είναι το πλήθος των εμφανίσεων του συμβόλου m στην θέση l των strings και n είναι το πλήθος των δειγμάτων που έχουμε στο cluster. Αν στο προηγούμενο παράδειγμα δούμε των πρώτο δυσδιάστατο πίνακα που δημιουργήθηκε και τον συμβολίσουμε με x, τότε τα στοιχεία x[i, m] είναι οι χαρακτήρες που βρίσκονται στην ι-οστή θέση του string m. Αν επίσης θεωρήσουμε τον χαρακτήρα c_l ως το σύμβολο που βρίσκεται στην θέση l του συνόλου Ω, τότε μπορούμε να συμβολίσουμε την ποσότητα δ_{iml} ως εξής: $\delta_{iml} = 1$ αν το $x[i,m] = c_l$ και 0 διαφορετικά. Αυτό το μέγεθος με την σειρά του μας επιτρέπει να ορίσουμε την ποσότητα n_{ml} που αναφέρθηκε πριν ως εξής:

$$n_{ml} = \sum_{i=1}^{n} \delta_{iml}$$

Έχοντας δημιουργήσει τα αρχικά clusters (τα οποία αποτελούν τα φύλλα του δέντρου που θα δημιουργηθεί) ο AC αρχίζει τις επαναλήψεις ενώνοντας τα clusters μεταξύ τους. Υπάρχουν 2 πιθανά μέτρα απόστασης:

Αν και τα δύο clusters περιέχουν μόνο από ένα string, όπως δηλαδή συμβαίνει στην αρχή του αλγορίθμου, τότε η απόσταση μεταξύ δύο cluster ισούται με την απόσταση

hamming μεταξύ των 2 strings. Η απόσταση hamming είναι ουσιαστικά το πλήθος των χαρακτήρων που διαφέρουν μεταξύ τους τα δύο strings στην αντίστοιχη θέση (δηλαδή το ABC με το ABD έχουν απόσταση hamming 1 ενώ το ABC με το CBA έχουν απόσταση hamming 2)

Σε οποιαδήποτε άλλη περίπτωση, η απόσταση μεταξύ δύο clusters u, v ορίζεται από τον τύπο D(u,v) = L(u) + L(v) – L(u, v). Η ποσότητα L(u) είναι η λογαριθμική πιθανοφάνεια του cluster u και ορίζεται από τον τύπο:

$$L(u) = \sum_{m=1}^{k} \sum_{l=1}^{|\Omega|} \log(\theta[\mathrm{m}, \mathrm{l}])$$

Η πιο σημαντική έννοια που παρουσιάζεται στον AC algorithm είναι αυτή της απόστασης μεταξύ των clusters D(u, v). Ουσιαστικά αντιπροσωπεύει την μείωση της πιθανοφάνειας που προέρχεται από την ένωση δύο clusters. Σε κάθε ένωση που κάνουμε, επιλέγουμε τα δύο clusters που επιφέρουν και την λιγότερη μείωση. Γενικά, μεγιστοποίηση της πιθανοφάνειας οδηγεί και σε μικρότερο μέγεθος του συμπιεσμένου κειμένου, άρα σε αύξηση του ποσοστού συμπίεσης.
Αυτός είναι ο AC αλγόριθμος στην γενική του μορφή. Στην συγκεκριμένη εργασία όμως κάναμε μια παραλλαγή αυτού. Αντί ο σχηματισμός του δέντρου να φτάσει μέχρι K clusters, το δέντρο αφήνεται να σχηματιστεί μέχρι την ρίζα του (δηλαδή αριθμός από clusters K = 1). Φυσικά η ρίζα περιέχει όλα τα αρχικά strings που δημιουργήθηκαν γι’ αυτό και δεν παράγεται κάποιο pattern από αυτόν τον κόμβο. Σε αυτήν την φάση πλέον κάνουμε μια αναδρομική διάσχιση του δένδρου που δημιουργήθηκε (προδιατεταγμένη για την ακρίβεια). Σε κάθε κόμβο που βρισκόμαστε εξάγουμε το pattern από το cluster και μετράμε το πλήθος των σταθερών συμβόλων που υπάρχουν (δηλαδή πόσα σύμβολα δεν είναι ο μπαλαντέρ *). Αν το πλήθος αυτό

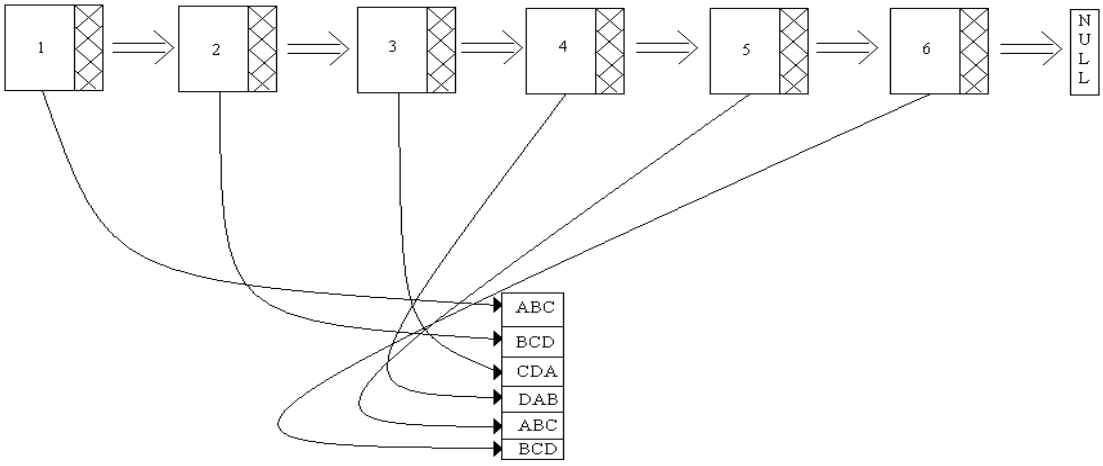
ικανοποιεί κάποιο κατώφλι (το οποίο το ορίζουμε εμείς) τότε επιστρέφεται ως τμήμα της εξόδου το συγκεκριμένο pattern και δεν επεκτεινόμαστε στα παιδιά του κόμβου. Διαφορετικά προχωράμε αναδρομικά στα παιδιά του κόμβου κάνοντας τις ίδιες ενέργειες. Σε όλες τις μετρήσεις που κάναμε χρησιμοποιήσαμε ως κατώφλι το L/2.

Με τον τρόπο αυτό μπορούμε να απαλλαχθούμε από την μία παράμετρο της εισόδου (το πλήθος των patterns K). Το πλήθος των παραμέτρων εισόδου γίνεται πλέον 2, το κείμενο εισόδου και το μήκος των pattern L. Ο αλγόριθμος εκτελείται επαναληπτικά για διάφορες τιμές του L (από 2 μέχρι κάποιο αριθμό n που ορίζουμε εμείς) και όλα τα patterns συλλέγονται σε ένα αρχείο όπου και ταξινομούνται και επιλέγονται

Στη βασική μορφή του αλγορίθμου (η οποία και υλοποιήθηκε πρώτα) όλη η δενδρική δομή δεν είναι τίποτα παραπάνω από μια απλά συνδεδεμένη λίστα. Κάθε cluster είναι και ένας κόμβος αυτής της λίστας και ουσιαστικά περιέχει δύο πληροφοριακές οντότητες: Τα strings τα οποία περιέχονται στο cluster και ένας δείκτης στον επόμενο κόμβο της λίστας.
Όλα τα strings τοποθετούνται σε κάποιο δυσδιάστατο πίνακα χαρακτήρων ενώ σε κάθε cluster ουσιαστικά αποθηκεύονται δείκτες σε αυτόν τον πίνακα. Με τον τρόπο αυτό αποφεύγεται επανάληψη πληροφορίας και οδηγούμαστε σε μικρότερη κατανάλωση μνήμης.

Σχηματικά λοιπόν όλη η δομή δεδομένων είναι όπως φαίνεται στο σχήμα που ακολουθεί. Σε αυτήν την εικόνα παρουσιάζεται το πώς έχουν σχηματιστεί οι δομές δεδομένων πριν αρχίσει η φάση της συγχώνευσης των clusters. Το κείμενο εισόδου είναι το ABCDABCD και αναζητούμε patterns μήκους 3. Παράγονται λοιπόν τα strings ABC, BCD, CDA, DAB, ABC, BCD. Για κάθε ένα string δημιουργείται και ένα cluster με μόνο ένα στοιχείο, το index του string στον πίνακα, όπως φαίνεται στο

Σχήμα 3.1. (Από εδώ και πέρα παρουσιάζεται βηματικά η εκτέλεση του αλγορίθμου με σχήματα)



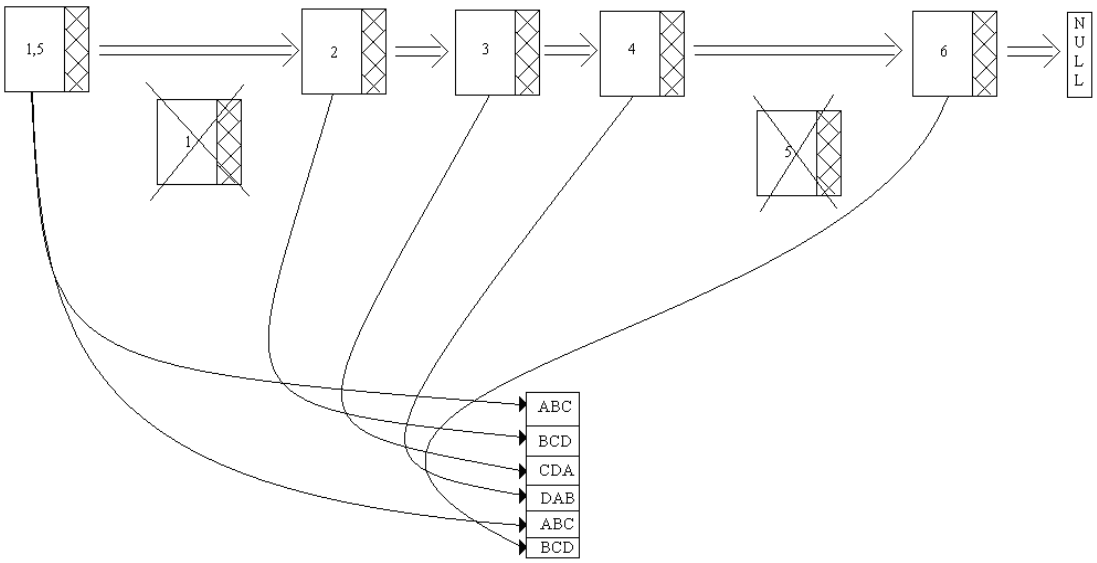
Σχήμα 3.1. Η Αρχική Δομή της Λίστας των Κόμβων – Φύλλων.

Σε αυτήν την φάση λοιπόν αρχίζει η συγχώνευση των clusters. Έστω ότι αναζητούμε 2 patterns, αυτό με την σειρά του σημαίνει ότι στο τέλος θα
απομείνουν 2 clusters άρα θα έχουμε 4 επαναλήψεις. Στην πρώτη επανάληψη θα υπολογιστούν οι αποστάσεις hamming μεταξύ όλων των κόμβων (αφού έχουμε μόνο από ένα string σε κάθε cluster) και την μικρότερη απόσταση την έχουν τα clusters 1, 5 και 2, 6. Επειδή όμως σε κάθε βήμα γίνεται μόνο από ένα merge των clusters στο πρώτο βήμα θα ενωθούν τα clusters 1 και 5 σε ένα νέο cluster, το (1,5). Κατά την ένωση αυτή, το νέο cluster περιέχει την ένωση των indices των δύο προηγούμενων που ενώθηκαν (έχοντας αφαιρέσει παράλληλα τις διπλές εγγραφές). Τοποθετείται στην αρχή της λίστας ενώ τα άλλα δύο clusters που δημιουργήθηκαν αφαιρούνται από την λίστα. (Σχήμα 3.2)

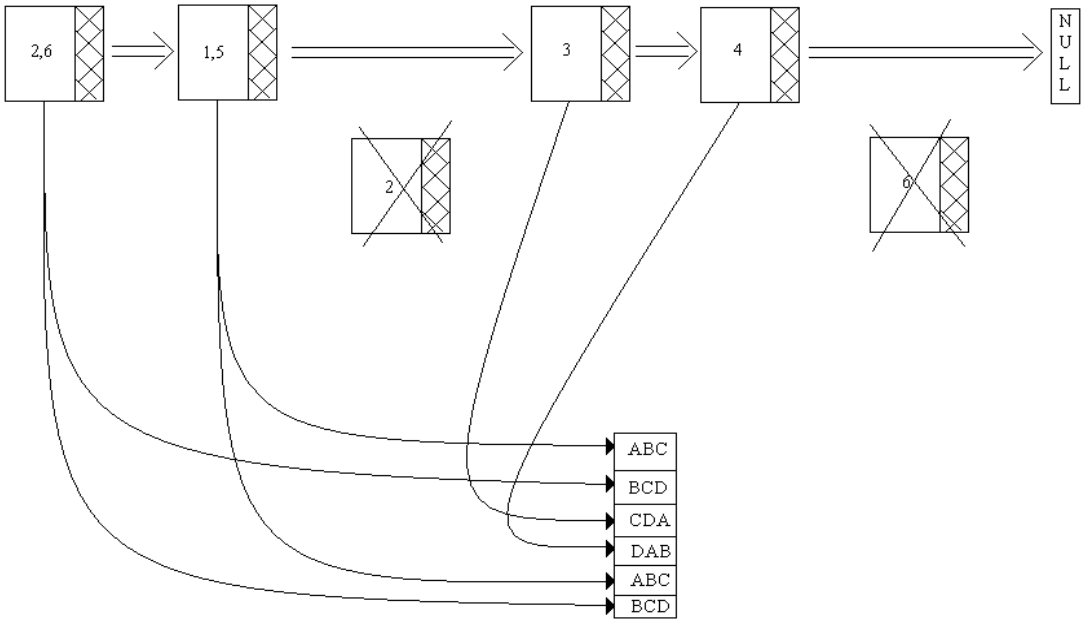
Στην επόμενη επανάληψη επιλέγονται προς ένωση τα clusters 2 και 6. Δημιουργείται το cluster (2,6) το οποίο με την σειρά του τοποθετείται στην αρχή της λίστας ακολουθούμενο από την διαγραφή των 2 και 6 από την λίστα (Σχήμα 3.3). Στην τρίτη

επανάληψη, ο αλγόριθμος εντοπίζει ότι την μικρότερη απόσταση την έχουν τα ζεύγη (1,5), 4 και (2,6), 3. Επειδή όμως μόνο μια ένωση συμβαίνει ανά επανάληψη, σε αυτήν την επανάληψη ενώνεται το πρώτο ζεύγος και στην επόμενη το δεύτερο(Σχήματα 3.4 και 3.5 αντίστοιχα). Σε αυτό το στάδιο πλέον, έχουν απομείνει μόνο δύο clusters οπότε και ο αλγόριθμος τελειώνει και τα patterns εξάγονται από το cluster.

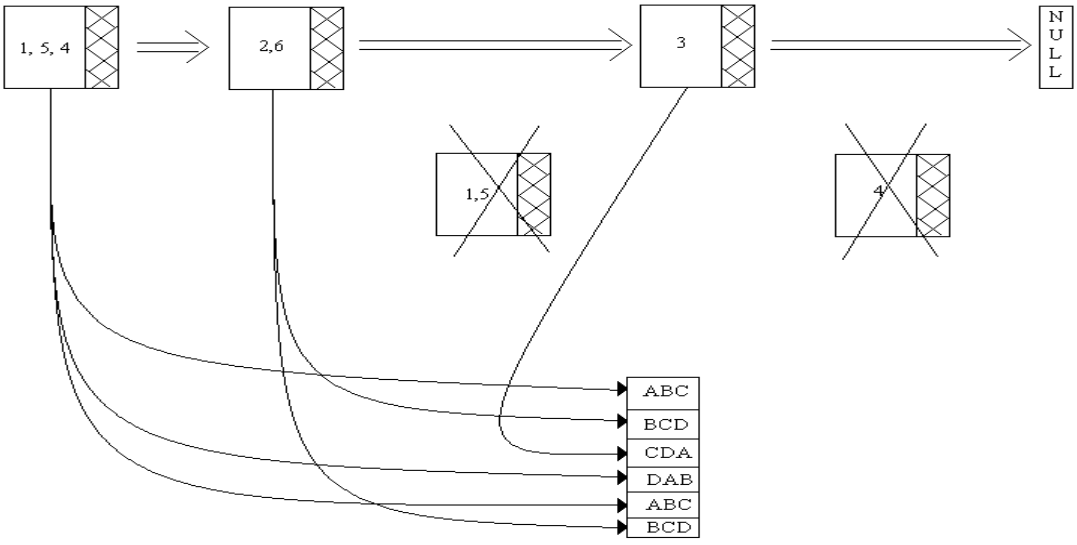
Τα επόμενα σχήματα δείχνουν τις μετατροπές στην λίστα κατά τις τέσσερις επαναλήψεις:



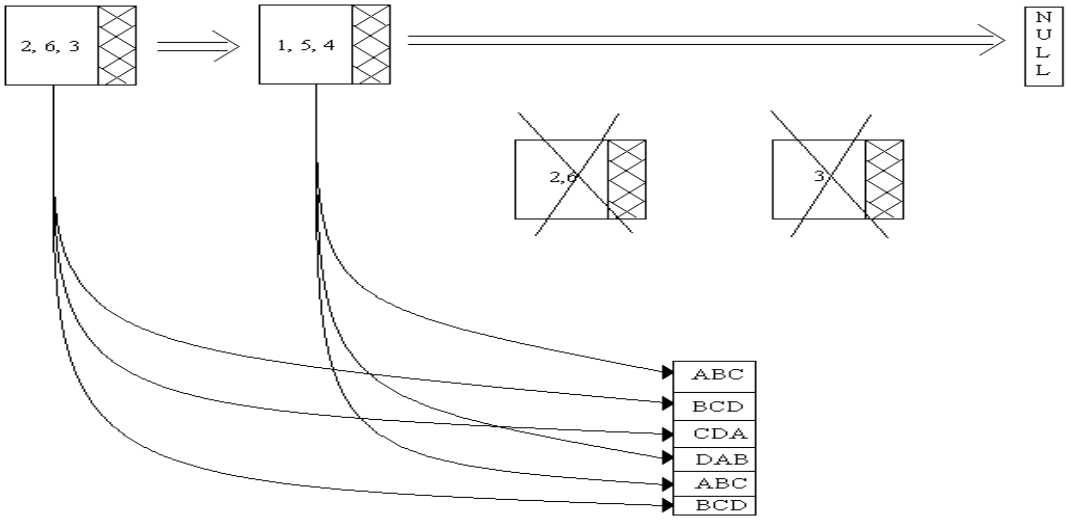
Σχήμα 3.2. Πρώτη Συνένωση Κόμβων με Ταυτόχρονη Διαγραφή των Κόμβων



Σχήμα 3.3. Δεύτερη Συνένωση Κόμβων (Ενώθηκαν ο 2 με τον 6)



Σχήμα 3.4. Τρίτη Συνένωση Κόμβων (Ενώθηκαν ο (1, 5) με τον 4)



Σχήμα 3.5. Τέταρτη Συνένωση Κόμβων (Ενώθηκαν ο (2, 6) με τον 3)

Η εξαγωγή των patterns γίνεται με τον τρόπο που αναφέρθηκε στην αρχή αυτού του κεφαλαίου. Κατ' αρχάς υπάρχουν δύο clusters άρα η διαδικασία εξαγωγής του pattern θα επαναληφθεί δύο φορές:

1^{o} pattern:

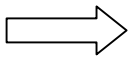
A	B	C
D	A	B
A	B	C

Υπολογίζονται οι συχνότητες εμφάνισης των συμβόλων:

	A	B	C	D
1	2/3	0	0	1/3
2	1/3	2/3	0	0
3	0	1/3	2/3	0

Το κατώφλι μας εδώ είναι (4-2)/4 = 2/4 =0.5. Συνεπώς παράγεται το pattern:

	A	B	C	D
1	0,66	0,00	0,00	0,33
2	0,33	0,66	0,00	0,00
3	0	0,33	0,66	0



A	B	C

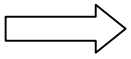
2^{o} pattern:

B	C	D
B	C	D
C	D	A

	A	B	C	D
1	0	2/3	1/3	0
2	0	0	2/3	1/3
3	1/3	0	0	2/3

Με παρόμοιο τρόπο το κατώφλι υπολογίζεται σε 0.5:

	A	B	C	D
1	0,00	0,66	0,33	0,00
2	0,00	0,00	0,66	0,00
3	0,33	0,00	0,00	0,66



B	C	D

Είναι προφανές από τα παραπάνω ότι η τιμή που έχει το κατώφλι είναι πολύ σημαντική για τα αποτελέσματα που θα πάρουμε. Στο συγκεκριμένο παράδειγμα με κατώφλι (n-2)/n πήραμε τα patterns ABC, BCD ενώ αν το κατώφλι είχε μεγαλύτερη τιμή (της τάξης του 0.7 για παράδειγμα) θα παίρναμε τα A*C, B*D.

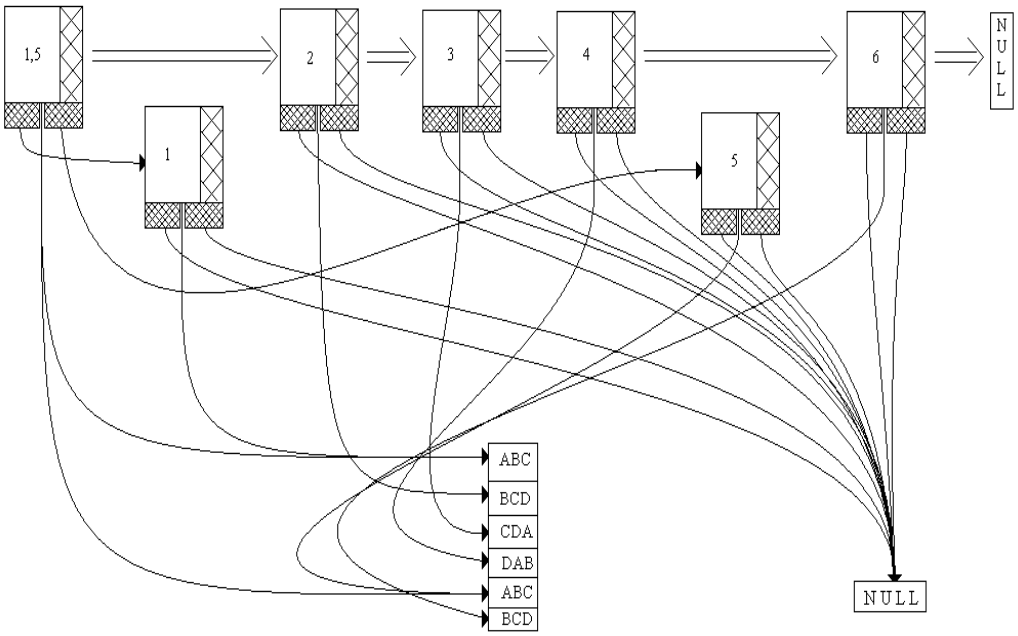
Με αυτόν τον τρόπο υλοποιήθηκε η βασική μορφή του αλγορίθμου. Για να απαλλαχθούμε όμως από την παράμετρο του πλήθους των patterns, Η δομή που ορίζει ένα cluster επαυξήθηκε με δύο παραπάνω δείκτες (εκτός του δείκτη next που ορίζει τον επόμενο κόμβο στην λίστα): Το δείκτη left_child, και τον δείκτη right_child. Όταν δύο κόμβοι ενώνονται, ο νέος κόμβος που παράγεται δίνει στον δείκτη left_child που έχει την τιμή του ενός εκ των δύο κόμβων και στον δείκτη right_child την τιμή του δεύτερου κόμβου. Η σειρά με την οποία τοποθετούνται οι κόμβοι στους δύο δείκτες δεν έχει σημασία. Τα αρχικά clusters που δημιουργούνται έχουν τιμή NULL στους δύο αυτούς δείκτες. Επίσης όταν οι δύο κόμβοι διαγράφονται από την λίστα δεν απελευθερώνονται και από την μνήμη του συστήματος (με free), διότι πλέον παραμένουν ως κόμβοι του δένδρου. Απλά ενημερώνονται οι δείκτες next των προηγούμενών τους κόμβων στην λίστα. Αφού πλέον δεν υπάρχει παράμετρος K, ο αλγόριθμος αφήνεται να δημιουργήσει το δέντρο μέχρι τον κόμβο-ρίζα. Εκεί αρχίζει η δεύτερη φάση του αλγορίθμου, η αναδρομική διάσχιση του δέντρου.

Σε αυτήν την φάση, Καλείται μια συνάρτηση (traverse) με όρισμα τον κόμβο ρίζα. Αυτή η συνάρτηση είναι αναδρομική και κάνει τα εξής βήματα:

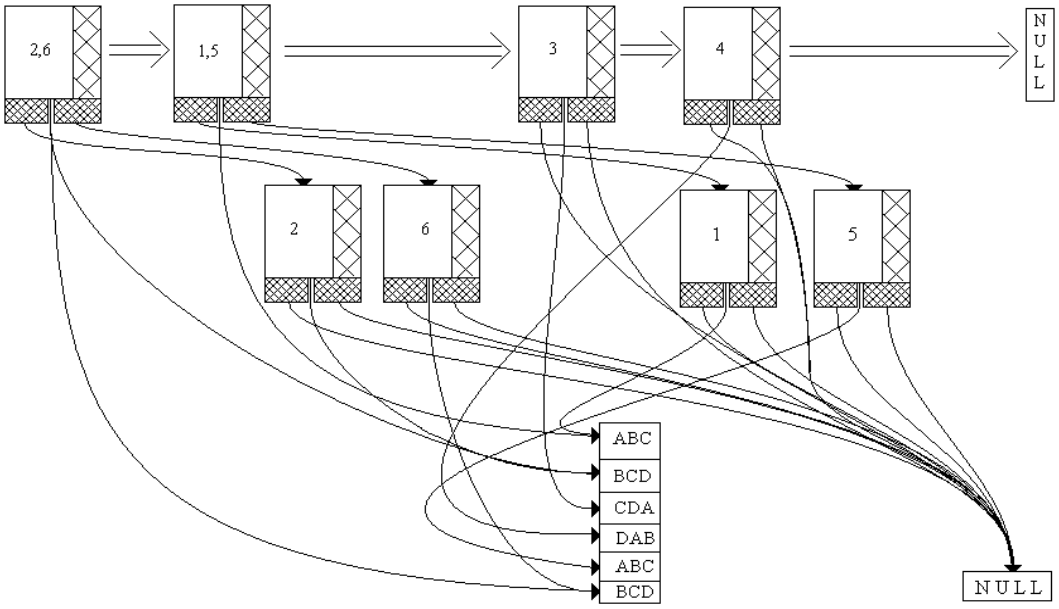
- Αν ο δείκτης έχει τιμή NULL τότε επιστρέφει.
- Εξάγει το pattern που βρίσκεται κωδικοποιημένο στο cluster του συγκεκριμένου κόμβου.

- Μετράει το πλήθος των σταθερών στοιχείων. Αν αυτό είναι μεγαλύτερο ή ίσο του L/2 τότε το pattern αυτό γράφεται σε ένα αρχείο και η συνάρτηση επιστρέφει.
- Αν το πλήθος των σταθερών στοιχείων δεν ικανοποιεί την παραπάνω σύγκριση, τότε η συνάρτηση καλεί τον εαυτό της δύο φορές, την μία με όρισμα το left_child και την άλλη με όρισμα το right_child.

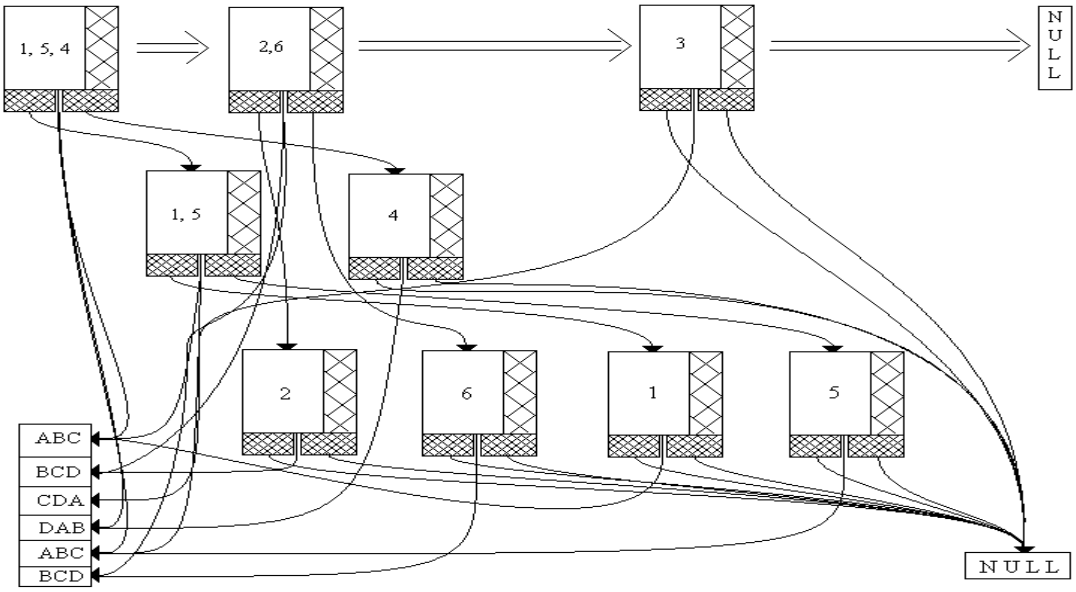
Ακολουθεί σχηματικά η διαδικασία παραγωγής του δέντρου βασιζόμενη στο προηγούμενό μας παράδειγμα: (Επιγραμματικά, Στην αρχή δημιουργείται ο κόμβος (1,5) μετά από συνένωση των 1 και 5. Οι 1 και 5 αφαιρούνται από την λίστα αλλά τοποθετούνται ως παιδιά του 1,5 (Σχήμα 3.6.). Με παρόμοιο τρόπο δημιουργείται ο (2, 6) με παιδιά τους 2 και 6. (Σχήμα 3.7.). Συνεχίζοντας, δημιουργείται ο (1, 5, 4) (Σχήμα 3.8.) και έπειτα ο (2, 6, 3) (Σχήμα 3.9.). Στο τέλος ενώνονται και αυτοί οι κόμβοι δημιουργώντας τον ένα και μοναδικό κόμβο της λίστας (1, 2, 3, 4, 5, 6) (Σχήμα 3.10). Στο Σχήμα 3.11. βλέπουμε το δέντρο στο οποίο αντιστοιχεί.)



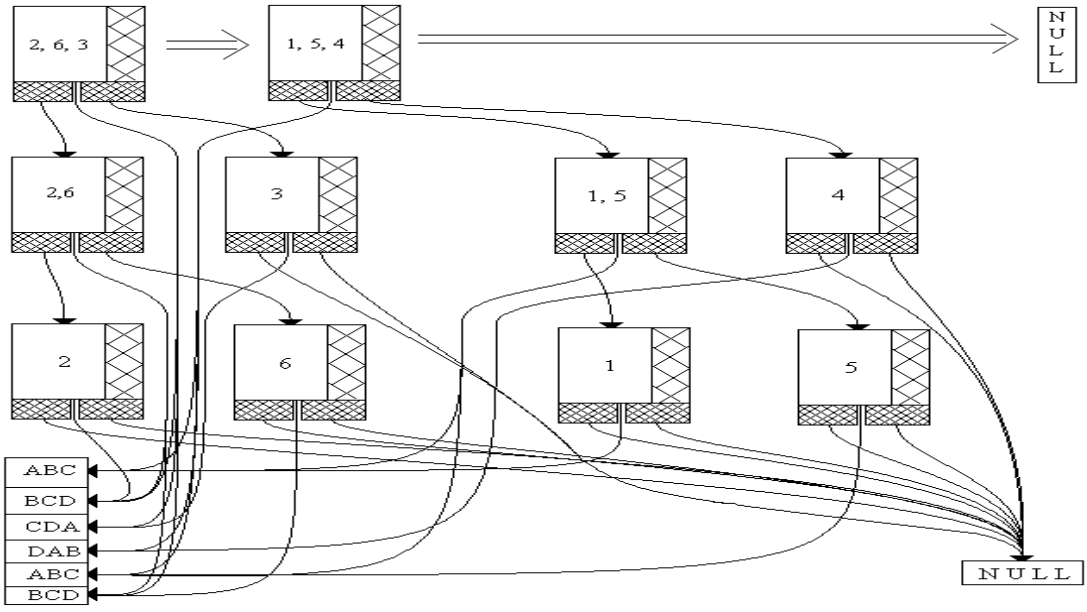
Σχήμα 3.6. Συνένωση των Κόμβων 1 και 5. Ο Νέος Κόμβος τους Έχει ως Παιδιά.



Σχήμα 3.7. Συνένωση των Κόμβων 2 και 6. Ο Νέος Κόμβος τους Έχει ως Παιδιά.

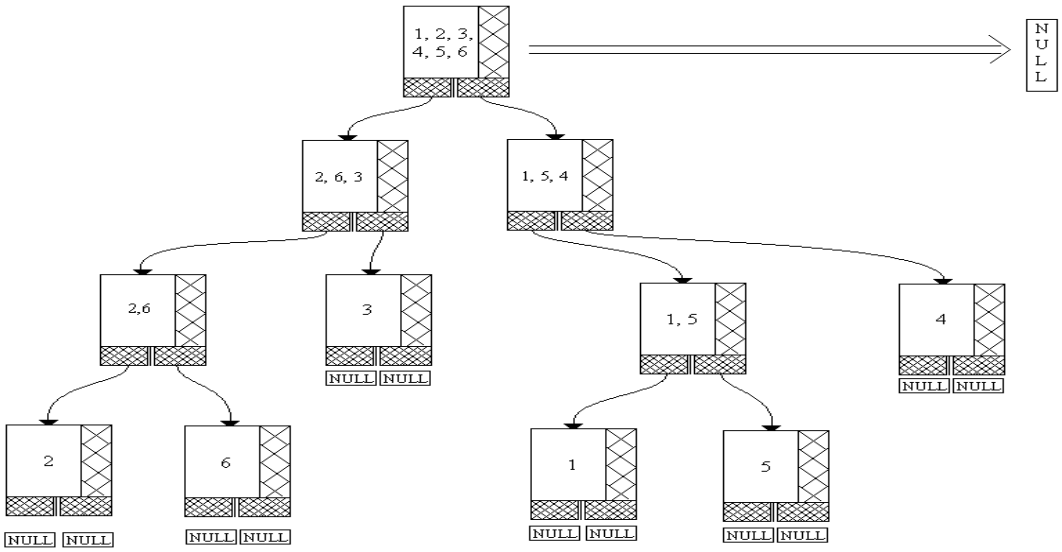


Σχήμα 3.8. Συνένωση του (1, 5) με το 4



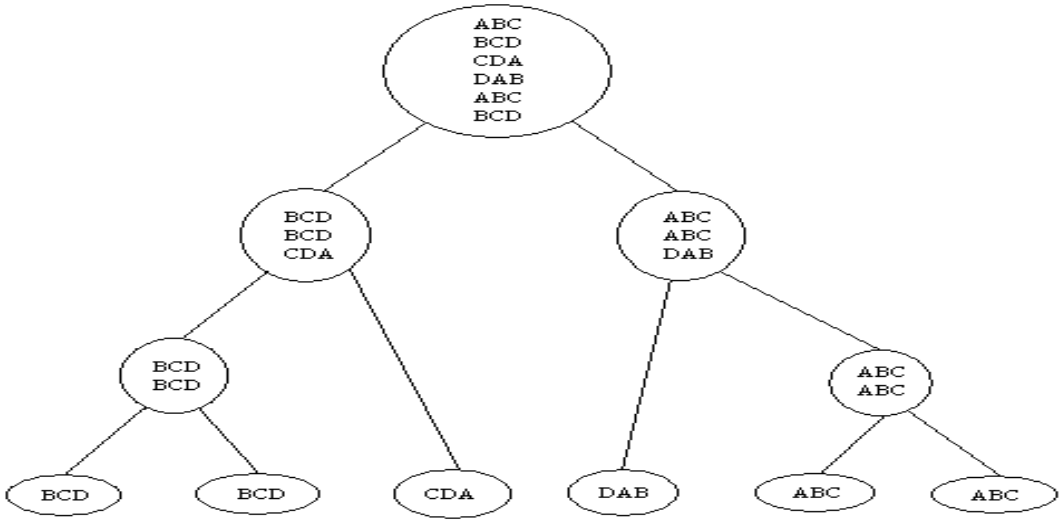
Σχήμα 3.9. Συνένωση του (2, 6) με το 3

Το οποίο τελικά καταλήγει στο:



Σχήμα 3.10. Η τελική Μορφή της Δομής Δεδομένων

Το οποίο ισοδυναμεί με:



Σχήμα 3.11. Το Δένδρο που Αντιστοιχεί στην Δενδρική Δομή που Έχουμε.

Σε αυτό το δένδρο λοιπόν καλείται η αναδρομική κατάβαση με όρισμα την ρίζα του. Εφόσον από τον κόμβο ρίζας δεν μπορεί να εξαχθεί κάποιο pattern με ικανό πλήθος σταθερών στοιχείων, η συνάρτηση καλείται πρώτα με όρισμα το αριστερό παιδί και έπειτα με όρισμα το δεξί παιδί. Από αυτά τα δύο εξάγονται τα δύο patterns BCD, ABC και η συνάρτηση επιστρέφει.

Ο παραπάνω είναι και ο τρόπος με τον οποίο δουλεύει ο αλγόριθμος. Η είσοδος του είναι ένα string και η έξοδός του είναι τα patterns σε μια μορφή που χρειάζεται περαιτέρω επεξεργασία για να μπορούν να χρησιμοποιηθούν πιο εύκολα. Αυτό μας οδηγεί στην προεπεξεργασία (preprocessing) και στην μεταεπεξεργασία (post processing)

3.1.1. Η Προεπεξεργασία

Στο στάδιο της προεπεξεργασίας έχουμε ως αρχική είσοδο το class αρχείο από ένα πρόγραμμα Java που το έχουμε κάνει compile. Αφού το περάσουμε από disassembly (βλέπε παράρτημα), διατρέχουμε τον κώδικα για να εντοπίσουμε τα basic blocks. Ο εντοπισμός είναι αρκετά εύκολος και λίγο πολύ αυτονόητος: Εντοπίζουμε όλες τις εντολές για jump (δηλαδή το jump και το άλμα υπό συνθήκη) σε ποιες θέσεις βρίσκονται μέσα στον κώδικα και εντοπίζουμε οι εντολές αυτές σε ποια σημεία του κώδικα δείχνουν. Με τον τρόπο αυτό έχουμε πλέον τα τμήματα του κώδικα τα οποία δεν διακόπτονται από κάποιο jump και δεν τα διακόπτει κάποιο jump από αλλού. Επίσης σε αυτά προστίθεται η πληροφορία για το τέλος κάθε μεθόδου.

Η πληροφορία αυτή χρησιμοποιείται έπειτα στο πρώτο τμήμα του agglomerative clustering αλγορίθμου για να διαγράψει από το αρχικό σετ των substrings αυτά τα

οποία έχουν όρια εκτός των basic blocks. Επίσης χρησιμοποιείται αργότερα στο κομμάτι του filtering στην μεταεπεξεργασία.

Έπειτα οι κώδικες από την κάθε μέθοδο της κλάσης συνενώνονται και παράγεται το string εισόδου ως εξής: Κάθε εντολή είναι ένας opcode με τιμή 0 – 196. Κάθε όρισμα αποτελεί επίσης μια τιμή από το 0 - 255. Άρα χρειαζόμαστε ένα byte για την εντολή και από ένα bytes για κάθε όρισμα που μπορεί να έχει. Με τον τρόπο αυτό τοποθετούνται όλα τα bytes σε ένα μεγάλο array και αυτό το array αποτελεί και το string της εισόδου.

3.1.2. Μεταεπεξεργασία

Σε αυτό το στάδιο έχουμε όλα τα patterns σε ακατέργαστη μορφή (raw) και εφαρμόζουμε μερικές πράξεις πάνω τους:

- Σε κάθε pattern αφαιρούμε όλους τους μπαλαντέρ (*) από την αρχή και το τέλος τους. Κάθε pattern πρέπει να αρχίζει με σταθερό στοιχείο και να τελειώνει με σταθερό στοιχείο. Μετά το πέρας αυτής της πράξης, τα patterns ξαναγράφονται στο αρχείο, ένα pattern σε κάθε γραμμή και την αρχή της κάθε γραμμής υπάρχει και σαν πληροφορία το μήκος του pattern.
- Λόγω των πολλών επαναλήψεων που εκτελεί ο αλγόριθμος, εμφανίζονται τα ίδια patterns παραπάνω από μία φορά. Για το λόγο αυτό σαρώνεται η λίστα με τα patterns και διαγράφονται όλες οι διπλές εμφανίσεις του κάθε pattern.
- Εφαρμόζουμε την πληροφορία που έχουμε από τα basic blocks για να δούμε αν κάποιο pattern ξεφεύγει των ορίων. Αν ναι, τότε διαγράφεται.

- Εφαρμόζουμε την λίστα με τα patterns στο string εισόδου και υπολογίζουμε το ποσοστό συμπίεσης που επιφέρει το κάθε pattern από μόνο του, ανεξάρτητα από τα υπόλοιπα. Αυτή η πληροφορία καταχωρείται σε έναν πίνακα.
- Διαγράφουμε τα patterns που παρουσιάζουν αρνητικό ή μηδενικό ποσοστό συμπίεσης. Λόγω του ότι για κάθε pattern πρέπει να έχουμε μια καταχώρηση στο λεξικό μας αυτό μπορεί να οδηγήσει σε patterns που αν και μειώνουν την αρχική είσοδο κατά μερικά bytes, εντούτοις χρειάζονται πιο πολλά bytes για την αποθήκευσή τους. Αυτό οδηγεί σε αρνητικά ποσοστά συμπίεσης.
- Τα patterns που απομένουν τα αποθηκεύουμε πλέον στο αρχείο εξόδου αφού πρώτα τα ταξινομήσουμε κατά φθίνουσα σειρά με βάση το ποσοστό συμπίεσης που μπορεί να επιφέρει το κάθε pattern. Ο λόγος για την ταξινόμηση θα αναφερθεί σε μετέπειτα κεφάλαιο.

3.2. Εύρεση των Patterns Χωρίς Μπαλαντέρ

Μια άλλη προσέγγιση στην συμπίεση με την χρήση patterns είναι να χρησιμοποιηθούν patterns χωρίς μπαλαντέρ μεταξύ των εντολών και των ορισμάτων τους. Ο τρόπος για την εύρεση αυτού του είδους των patterns είναι πιο απλός. Συγκεκριμένα: Χωρίζουμε το string της εισόδου σε υπο-strings μήκους W τα οποία παράγοντας ολισθαίνοντας ένα παράθυρο μήκους W κατά μήκος του αρχικού string. Αυτά όλα τα υπο-strings ελέγχονται μεταξύ τους για να εντοπιστεί ποια είναι ίδια. Με τον τρόπο αυτό έχουμε στο τέλος μια λίστα με patterns μήκους W (όπου το W το επιλέγουμε εμείς και συνήθως η μέθοδος καλείται επαναληπτικά για W από 1 μέχρι κάποιο αριθμό n). Μετά από αυτήν την διαδικασία, εντοπίζονται τα basic blocks μέσα

στον κώδικα και φιλτράρονται τα patterns που πέφτουν εκτός ορίων. Μετά την εύρεση του βέλτιστου συνδυασμού αυτών των patterns, αυτά που έχουν επιλεχθεί αποθηκεύονται σε κάποιο λεξικό και αντικαθίστανται οι εμφανίσεις τους στον αρχικό κώδικα. Κάθε εμφάνιση αντικαθίσταται από ένα δείκτη στο λεξικό στο αντίστοιχο pattern. Η αποθήκευση στο λεξικό γίνεται σειριακά το ένα pattern μετά το άλλο και στο τέλος του κάθε pattern προστίθεται ένα ειδικό byte για να σηματοδοτήσει το τέλος του pattern που ονομάζεται END_OF_PATTERN.

3.3. Ένα Παράδειγμα που Συγκρίνει τις δύο Μεθόδους

Εύλογα λοιπόν γεννάται το ερώτημα γιατί να χρησιμοποιηθεί η μέθοδος με τον agglomerative έναντι της πιο απλής μεθόδου μιας και η διαφορά στην πολυπλοκότητα είναι εμφανής. Η απάντηση είναι ότι η μέθοδος με τα μπαλαντέρ έχει πιο πολλά να προσφέρει από άποψη ποσοστού συμπίεσης. Αυτό μπορούμε να το δούμε με το παρακάτω παράδειγμα:

Έστω ότι έχουμε ένα τρισδιάστατο διάνυσμα v = (x, y, z) και θέλουμε να υπολογίσουμε το μέτρο του. Ως γνωστόν αυτό ισούται με |v| = $\sqrt{x^2 + y^2 + z^2}$. Αν προγραμματίζαμε μια κλάση σε Java για το διάνυσμα αυτό θα είχαμε τον εξής πηγαίο κώδικα:

```
Class xyz {
        public float x, y, z;
        public xyz (float x1, float y1, float z1) {
                x=x1;
                y=y1;
                z=z1;
}
public double distance () {
        return Math.sqrt(x*x+y*y+z*z);
}
```

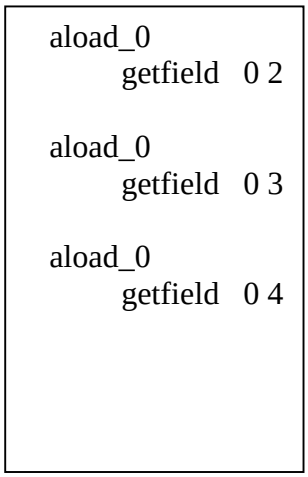
Σχήμα 3.12. Ο Πηγαίος Κώδικας της Κλάσης xyz που Κωδικοποιεί ένα Διάνυσμα

Αν τώρα περάσουμε τον κώδικα αυτόν από την διαδικασία του compiling, παίρνουμε τον εξής bytecode (παρουσιάζεται μόνο το τμήμα του κώδικα για την μέθοδο distance που μας ενδιαφέρει):

```
Code:
aload_0
getfield    0  2
aload_0
getfield    0  2
fmul
aload_0
getfield    0  3
aload_0
getfield    0  3
fmul
fadd
getfield    0  4
aload_0
getfield    0  4
fmul
fadd
f2d
invokestatic  0  5
dreturn
```

Σχήμα 3.13. Ο Bytecode της Μεθόδου distance

Αν θεωρήσουμε ότι αναζητούμε patterns χωρίς μπαλαντέρ, μπορούμε εύκολα να εντοπίσουμε τα εξής patterns:



Σχήμα 3.14. Τα Patterns Χωρίς Μπαλαντέρ που Προκύπτουν από τον Bytecode.

Αν δοκιμάσουμε τώρα να αντικαταστήσουμε τα patterns στον bytecode που έχουμε, έχουμε τα εξής: Το πρώτο pattern έχει μήκος 4 bytes που σημαίνει ότι τόσα bytes αντικαθιστά από κάθε εμφάνιση. Συνεπώς το πρώτο pattern αφαιρεί 8 bytes από τον αρχικό κώδικα. Ομοίως γίνεται και με τα υπόλοιπα patterns. Επομένως τα 3 patterns αφαιρούν συνολικά 24 bytes. Από εκεί και πέρα όμως κάθε pattern χρειάζεται 1 byte για να χρησιμοποιηθεί ως δείκτης στο λεξικό που έχουμε οπότε έχουμε 3 x 2 x 1 = 6 bytes ως δείκτες. Όσον αφορά την αποθήκευση του κάθε pattern στο λεξικό μας, θέλουμε 4 bytes για κάθε pattern καθώς και ένα extra byte που λειτουργεί ως σημάδι ότι το pattern έχει τέλος και λέγεται END_OF_PATTERN. Αν λάβουμε και αυτά υπ' όψιν έχουμε 3 x (4 + 1) = 15 bytes χρειάζονται για το dictionary. Αν σε αυτά προσθέσουμε τα 6 που χρησιμοποιούνται ως δείκτες παίρνουμε ότι χρειαζόμαστε τελικά 21 bytes πληροφορίας για την σωστή κωδικοποίηση και αποκωδικοποίηση των pattens άρα η ολική συμπίεση που επιτυγχάνεται είναι ίση με 24 – 21 = 3 bytes στο σύνολο των 33 bytes άρα μιλάμε για μια ποσοστιαία μείωση της τάξεως του 9%

Αν τώρα θεωρήσουμε ότι χρησιμοποιούμε μπαλαντέρ στα patterns, αμέσως εντοπίζουμε ένα pattern που καλύπτει χωρικά αυτά που θα βρίσκαμε χωρίς μπαλαντέρ:

```
aload_0
getfield    0    *
aload_0
getfield    0    *
fmul
```

Σχήμα 3.15. Το Pattern με Μπαλαντέρ που Βρίσκουμε με τον AC

Αυτό το pattern όταν το αντικαθιστούμε στον bytecode, συμβαίνουν τα εξής: Για να λειτουργήσει σωστά, εκτός από ένα δείκτη στο λεξικό που τοποθετείται στην θέση του pattern, χρειαζόμαστε και τα bytes που βρίσκονται στην θέση των μπαλαντέρ. Άρα, σε κάθε εμφάνιση του pattern στον bytecode χρειαζόμαστε 3 bytes (ένα δείκτη και 2 bytes για τα μπαλαντέρ). Το pattern εμφανίζεται 3 φορές οπότε στον bytecode θα χρειαστούμε 3 x 3 = 9 bytes. Για την αποθήκευση του pattern αυτού στο λεξικό χρειαζόμαστε 7 bytes (όσα δηλαδή δεν είναι μπαλαντέρ) συν του END_OF_PATTERN. Λόγω του ότι χρησιμοποιούμε μπαλαντέρ πλέον η πληροφορία αυτή δεν αρκεί. Χρειαζόμαστε ένα byte ακόμα για να μας δείξει σε ποια θέση στο pattern υπάρχουν οι μπαλαντέρ. Αυτό μας δίνει 7 + 1 + 1 = 9 bytes στο λεξικό. Σε συνδυασμό τώρα με τα υπόλοιπα 9 bytes που υπολογίσαμε πριν έχουμε 18 bytes στο σύνολο πληροφορίας. Το pattern καθαυτό τώρα αφαιρεί 27 bytes από τον bytecode δηλαδή επιφέρει τελική συμπίεση 27 – 18 = 11 bytes δηλαδή επέρχεται ποσοστιαία συμπίεση της τάξης του 33% που είναι κατά πολύ μεγαλύτερη από το 9% που προσφέρει η μέθοδος χωρίς τα μπαλαντέρ.

Διαισθητικά λοιπόν κατανοούμε ότι λόγω της γενίκευσης που επιφέρει η παραμετρική μέθοδος στην βασική μέθοδο χωρίς μπαλαντέρ θα έχουμε καλύτερα ποσοστά συμπίεσης στην γενική περίπτωση. Το κατά πόσο καλύτερα είναι τα ποσοστά αυτά θα το δούμε στο κεφάλαιο που περιέχει τις μετρήσεις.

ΚΕΦΑΛΑΙΟ 4. Η ΚΩΔΙΚΟΠΟΙΗΣΗ ΚΑΙ ΑΠΟΚΩΔΙΚΟΠΟΙΗΣΗ ΤΟΥ BYTECODE

4.1. Το Πρόβλημα της Εύρεσης του Βέλτιστου Συνδυασμού.

Έχοντας πλέον βρει όλα τα patterns που υπάρχουν στον κώδικά μας, προχωρούμε στο σημείο που εντοπίζουμε ποια από αυτά θα πάρουν μέρος στην τελική συμπίεση. Αυτό συμβαίνει λόγω του ότι τα patterns μεταξύ τους επικαλύπτονται στον κώδικα. Αν δεν υπήρχε καμία επικάλυψη μεταξύ των patterns τότε η λύση θα ήταν απλά να χρησιμοποιήσουμε όλα τα διαθέσιμα patterns και να πάρουμε απευθείας το βέλτιστο ποσοστό συμπίεσης.

Η πιο απλή ιδέα στο παραπάνω πρόβλημα θα έλεγε να βρούμε όλους τους διαθέσιμους συνδυασμούς των patterns που έχουμε, να τους εφαρμόσουμε στον bytecode και να κρατήσουμε αυτόν που επιφέρει την βέλτιστη συμπίεση. Αυτή η ιδέα

εγγυάται μεν ότι θα πάρουμε το καλύτερο δυνατό αποτέλεσμα αλλά δεν είναι εφαρμόσιμη. Ας δούμε το γιατί:

Εάν έχουμε ένα σύνολο P από n patterns $P = \{p_1, p_2, \ldots, p_n\}$ τότε ο πιο απλός τρόπος για να συμβολίσουμε ποιο από αυτά τα patterns ανήκει στο τελικό αποτέλεσμα θα είναι να ορίσουμε μια σειρά από bits μήκους n τα οποία θα έχουν την τιμή 1 ή 0 ανάλογα με το εάν το αντίστοιχο pattern βρίσκεται στο τελικό σύνολο ή όχι αντίστοιχα. Αν για παράδειγμα έχουμε ένα αρχικό σύνολο με 5 patterns και στο τέλος εντοπίζουμε ότι κρατάμε μόνο το πρώτο και το 5^o, η σειρά των bits θα έχει την μορφή 10001. Βάσει αυτού, για να βρούμε όλους τους συνδυασμούς από n patterns θα πρέπει να βρούμε όλους τους συνδυασμούς από n bits. Αυτό σημαίνει ότι θα πρέπει να βρούμε όλους τους αριθμούς που στην δυαδική αριθμητική συμβολίζονται με n bits. Οι αριθμοί αυτοί είναι από το 0 μέχρι το 2^n-1. Το πλήθος των αριθμών αυτών είναι ίσο με $2^n - 1 - 0 + 1 = 2^n$. Συνεπώς για να βρούμε τον βέλτιστο συνδυασμό θα πρέπει να σαρώσουμε 2n διαφορετικούς συνδυασμούς. Με τα σημερινά δεδομένα υπολογισμού αν το n ξεπεράσει την τιμή του 30 αυτό γίνεται πλέον ακατόρθωτο και όπως θα δούμε παρακάτω το n έχει σχεδόν πάντα τιμές πολύ μεγαλύτερες του 30.

Εφ' όσων λοιπόν δεν γίνεται να βρούμε το βέλτιστο εκ συνθηκών προχωρούμε στο να βρούμε μεθόδους οι οποίες να δίνουν «καλά» αποτελέσματα σε κανονικούς χρόνους. Συγκεκριμένα παρουσιάζονται 2 τέτοιες μέθοδοι, η πρώτη χρειάζεται γραμμικό χρόνο εκτέλεσης (είναι δηλαδή της τάξης του O(n)) ενώ η δεύτερη τετραγωνικού (είναι της τάξης του $O(n^2)$).

4.1.1. Η Πρώτη Ευρετική Μέθοδος

Μια απλή ιδέα αλλά ταυτόχρονα αρκετά γρήγορη και που μπορεί να δώσει καλά αποτελέσματα συμπίεσης είναι η εξής (παρουσιάζεται βηματικά):

1. Ταξινομούμε τα patterns στο σύνολό μας σε φθίνουσα σειρά με βάση το ποσοστό συμπίεσης που επιφέρει το κάθε pattern μόνο του (Δηλαδή αν το σύνολο των patterns περιείχε μόνο ένα pattern πόσο τελική συμπίεση θα είχαμε στον bytecode). Το pattern που μπορεί να προσφέρει το μεγαλύτερο ποσοστό συμπίεσης τοποθετείται πρώτο ενώ το pattern με το μικρότερο ποσοστό τελευταίο.
2. Δημιουργούμε μια σειρά από bits μήκους ίσο με το πλήθος των patterns η οποία στην αρχή είναι «κενή» (έχει όλα τα bits αρχικοποιημένα στην τιμή 0).
3. Αρχικοποιούμε έναν μετρητή επαναλήψεων να δείχνει στο πρώτο pattern και μια μεταβλητή που μετράει το μέγιστο ποσοστό που έχουμε επιτύχει μέχρι στιγμής με την τιμή 0 και αρχίζουμε έναν βρόχο εντολών που θα επαναληφθεί n φορές.
4. Για κάθε ι-οστό pattern που συναντάμε το προσθέτουμε στην σειρά με bits (δηλαδή κάνουμε το ι-οστό bit να ισούται με το 1) και εφαρμόζουμε το μέχρι τώρα σύνολο με bits στον bytecode που έχουμε. Μετράμε το ποσοστό συμπίεσης και αν αυτό είναι μεγαλύτερο από την μεταβλητή που μετράει το μέχρι τώρα μεγαλύτερο τότε ενημερώνεται η μεταβλητή αυτή με την νέα τιμή και συνεχίζουμε τις επαναλήψεις. Αν το ποσοστό συμπίεσης που θα υπολογίσουμε δεν είναι μεγαλύτερο της μεταβλητής μας τότε μηδενίζουμε το ι-οστό bit και συνεχίζουμε τις επαναλήψεις.

Μετά το πέρας όλων των επαναλήψεων ξέρουμε ποιο είναι το βέλτιστο ποσοστό συμπίεσης που μπορεί να επιτύχει η συγκεκριμένη μέθοδος και έχουμε και την σειρά με τα bits που δείχνουν ποια patterns δίνουν το ποσοστό αυτό.

Το μεγαλύτερο πλεονέκτημα της παραπάνω μεθόδου είναι η ταχύτητά της. Δίνει πολύ γρήγορα αποτέλεσμα πράγμα που ίσως μας ενδιαφέρει σε πιθανές εφαρμογές που θέλουμε να εκτελούνται γρήγορα. Η λογική της μεθόδου είναι ότι ουσιαστικά αν δύο patterns επικαλύπτονται στον bytecode και δεν μπορούν να εφαρμοστούν και τα δύο, αυτό που κρατάμε είναι αυτό που έχει την μεγαλύτερη προτεραιότητα, δηλαδή αυτό που μπορεί να δώσει την μεγαλύτερη συμπίεση, διαισθητικά.

Η εφαρμογή της παραπάνω μεθόδου οδηγεί πάντα στο πρώτο pattern στην ταξινομημένη λίστα να γίνεται αποδεκτό. Αυτό με την σειρά του οδηγεί σε κάποια patterns να μην γίνονται αποδεκτά λόγω συγκρούσεων με το συγκεκριμένο pattern. Η συγκεκριμένη διαπίστωση οδηγεί με την σειρά της στην επόμενη ευρετική που αποτελεί γενίκευση αυτής.

4.1.2. Η Δεύτερη Ευρετική Μέθοδος

Η λογική της δεύτερης ευρετικής μεθόδου είναι να δώσουμε σε όλα τα patterns την ευκαιρία να τοποθετηθούν στο σύνολο των patterns που δοκιμάζουμε για να δούμε το κατά πόσο μπορούμε να βελτιώσουμε το ποσοστό συμπίεσης. Αυτά μπορούμε να το επιτύχουμε απλά ως εξής:

Αντί ο βρόχος επαναλήψεων να αρχίζει πάντα από την τιμή 1 και να καταλήγει στην τιμή n, Περικλείουμε τον βρόχο αυτόν με έναν άλλο βρόχο που περιέχει έναν μετρητή j που ξεκινάει από το 1 και καταλήγει στο n. Ο εσωτερικός πλέον βρόχος αντί να

αρχίζει από το 1 κάθε φορά αρχίζει από το j. Εφ’ όσων όμως αρχίζει από το j και πρέπει να κάνει n επαναλήψεις κάποια στιγμή το j ξεπερνάει το n. Εκείνη τη στιγμή η αναζήτηση των patterns πλέον προχωράει από το πρώτο pattern, δηλαδή κλείνει κύκλο. Πιο φορμαλιστικά έχουμε τα εξής:

1. Αρχικοποιούμε μια σειρά από bits μήκους n με 0 που ονομάζουμε bs1.
2. Αρχικοποιούμε μια μεταβλητή που κρατάει το μέγιστο ποσοστό συμπίεσης μέχρι στιγμής στο 0 που ονομάζουμε mc1.
3. Ο βρόχος j αρχίζει τις επαναλήψεις από το 0 μέχρι το n-1.
4. Αρχικοποιούμε μια μεταβλητή που κρατάει το μέγιστο ποσοστό συμπίεσης για την συγκεκριμένη επανάληψη με την τιμή 0 που ονομάζεται mc.
5. Αρχικοποιούμε μια σειρά από bits μήκους n με 0 που ονομάζουμε bs.
6. Ένας βρόχος i αρχίζει τις επαναλήψεις από το 0 μέχρι το n-1.
7. Μια μεταβλητή c παίρνει την τιμή ((j + i) mod n) + 1 και δείχνει στο συγκεκριμένο pattern μέσα στο σύνολο των patterns που έχουμε (Η χρήση της πράξης του υπολοίπου εγγυάται ότι ο μετρητής κλείνει κύκλο κάποια στιγμή και αρχίζει από την πρώτη θέση).
8. Στην θέση c μέσα στο bs θέτουμε την τιμή 1.
9. Εφαρμόζουμε το σύνολο bs στον bytecode που έχουμε και μετράμε τι ποσοστό συμπίεσης επέρχεται με την προσθήκη στο σύνολο του pattern c. Αν το ποσοστό αυτό είναι μεγαλύτερο από το mc τότε το mc παίρνει την νέα τιμή αλλιώς η τιμή του bs στην θέση c γίνεται 0.
10. Αν η τιμή του mc είναι μεγαλύτερη από αυτήν του mc1 τότε η τιμή του mc1 γίνεται ίση με αυτήν του mc και η σειρά bs αντιγράφεται στην σειρά bs1.

Από τα παραπάνω καταλαβαίνουμε εύκολα ότι η δεύτερη ευρετική μέθοδος δεν είναι τίποτα άλλο παρά n επαναλήψεις της πρώτης. Συγκεκριμένα η πρώτη επανάληψη της δεύτερης ευρετικής είναι ακριβώς ίδια με την πρώτη ευρετική μέθοδο. Για τον λόγο αυτό είναι βέβαιο ότι η δεύτερη ευρετική μπορεί να επιτύχει ποσοστά συμπίεσης

μεγαλύτερα ή στην χειρότερη περίπτωση ίσα με την πρώτη ευρετική. Εύλογα λοιπόν μπορεί κάποιος να αναρωτηθεί σε τι μπορεί να χρησιμεύσει η πρώτη ευρετική εφ' όσων είναι υποσύνολο της δεύτερης; Η απάντηση στο ερώτημα αυτό είναι ο χρόνος. Η δεύτερη ευρετική χρειάζεται ακριβώς n φορές τον χρόνο της πρώτης ευρετικής για να ολοκληρωθεί. Στα μικρά παραδείγματα γενικά αυτό δεν είναι πρόβλημα αλλά όταν το πλήθος των patterns ξεπεράσει κάποια τιμή όπως το 1000, τότε η χρονική διαφορά γίνεται πλέον εμφανής (η πρώτη μέθοδος τελειώνει σε δευτερόλεπτα ενώ η δεύτερη θέλει ώρες).

4.2. Η Κωδικοποίηση του Bytecode

Βρισκόμαστε πλέον στο στάδιο όπου έχουμε βρει τα patterns που θα συμμετάσχουν στην διαδικασία της συμπίεσης και προχωρούμε πλέον στην συμπίεση καθαυτή. Σε κάθε σημείο του κώδικα που εντοπίζουμε ένα pattern αντικαθιστούμε όλο το τμήμα του κώδικα με ένα σύμβολο που υποδηλώνει την ύπαρξη του συγκεκριμένου pattern. Για να το καταφέρουμε αυτό εύκολα εκμεταλευόμαστε το σετ εντολών του bytecode.
Τα opcodes που χρησιμοποιούνται έχουν τιμές από το 0 μέχρι το 196. Από εκεί και πέρα δεν χρησιμοποιούνται τα υπόλοιπα. Άρα αν εμείς χρησιμοποιήσουμε το 196 ως βάση και το προσθέτουμε στα patterns με αριθμούς 1, 2, 3… τότε θα έχουμε σύμβολα με τιμές 197, 198, κτλ. Με τον τρόπο αυτό εύκολα κωδικοποιούμε ότι ο αριθμός που συνάντησε η JVM είναι δείκτης στο λεξικό και όχι opcode. Ο τρόπος που τα patterns αποθηκεύονται στο λεξικό είναι επίσης απλός για την μέθοδο χωρίς μπαλαντέρ αλλά η παραμετρική μέθοδος εισάγει μια ανάγκη για ύπαρξη περαιτέρω πληροφορίας από τα προφανή. Ας αρχίσουμε από τα βασικά. Η μη παραμετρική μέθοδος απαιτεί για κάθε pattern την αποθήκευση του καθώς και ένα byte που χρησιμεύει ως ενδεικτικό ότι το ένα pattern τελείωσε και αρχίζει το επόμενο στο αρχείο που αποθηκεύονται. Το byte αυτό έχει την συμβολική ονομασία END_OF_PATTERN. Μια JVM

διαβάζοντας τα δεδομένα που είναι αποθηκευμένα μπορεί με βάση τα παραπάνω να διαβάσει και τα patterns ένα ένα αφού θα εντοπίζει το END_OF_PATTERN ανάμεσά τους. Τα παραμετρικά patterns όμως παρουσιάζουν ένα πρόβλημα. Χρειαζόμαστε έναν τρόπο κατά την αποθήκευση τους για να δείξουμε σε ποιες θέσεις είναι τα μπαλαντέρ μέσα στο pattern. Η λύση σε αυτό το πρόβλημα είναι αρκετά εύκολη. Χρησιμοποιούμε ένα byte ακόμα για κάθε pattern. Κάθε bit του pattern αυτού αν είναι 1 ή 0 δηλώνει το αν ακολουθεί byte αποθηκευμένο στο pattern ή αν ακολουθεί μπαλαντέρ. Αν το μήκος του pattern είναι μεγαλύτερο του 10 τότε χρειαζόμαστε και 2^{o} byte. Αλλιώς ένα byte είναι αρκετό γιατί δεν πρέπει να ξεχνάμε ότι τα patterns υποχρεωτικά αρχίζουν και τελειώνουν με σταθερό σημείο, δηλαδή δεν έχουν μπαλαντέρ στο τέλος. Συγκεκριμένα, για κάποιο pattern μήκους K χρειαζόμαστε $\lceil (K-2)/8 \rceil$ bytes πληροφορίας.

4.3. Η Αποκωδικοποίηση του Bytcode

Για την αποκωδικοποίηση του bytecode, επαυξάνουμε μια standard JVM με ένα τμήμα υπεύθυνο για την αποσυμπίεση (decompression module) το οποίο κατά την αρχικοποίηση του συστήματος, διαβάζει το λεξικό με τα patterns και τα αποθηκεύει στην μνήμη του. Κατά την διάρκεια της εκτέλεσης ενός συμπιεσμένου προγράμματος, όταν ο interpreter της JVM συναντήσει έναν κωδικό όπου δεν ανήκει σε γνωστό opcode κατανοεί ότι πρόκειται για pattern και μετακινεί τον έλεγχο του προγράμματος στο εσωτερικό του λεξικού. Αν η μέθοδος κωδικοποίησης που εφαρμόσθηκε ήταν η μη παραμετρική, τότε στο εσωτερικό του λεξικού απλά τα σύμβολα γίνονται fetch το ένα μετά το άλλο, με στόχο την δημιουργία ολόκληρων εντολών (διαβάζεται ο opcode και έπειτα ένα ένα τα ορίσματα που αντιστοιχούν στην εντολή αυτή. Όταν συμπληρωθεί η εντολή απλά εκτελείται). Όταν διαβαστεί το

σύμβολο END_OF_PATTERN, ο έλεγχος του προγράμματος μεταφέρεται από το λεξικό πίσω στον αρχικό κώδικα.

Στην παραμετρική μέθοδο επαναλαμβάνεται η ίδια διαδικασία αλλά με κάποιες επιπλέον πράξεις. Κατ΄ αρχάς, όταν αρχικοποιείται το decompression module με το δεδομένο λεξικό, για κάθε pattern που διαβάζει, εξετάζει τα extra bytes για να εντοπίσει το πλήθος των μπαλαντέρ που περιέχονται στο εσωτερικό του. Όταν ο interpreter συναντήσει ένα άγνωστο opcode εντοπίζει σε ποιο pattern αναφέρεται, και τοποθετεί σε ένα buffer τα σύμβολα που αντιστοιχούν στα μπαλαντέρ του pattern. Έπειτα ο έλεγχος του προγράμματος μεταφέρεται στο εσωτερικό του pattern. Με παρόμοιο τρόπο δημιουργούνται οι εντολές μέσα στο pattern (για κάθε opcode που γίνεται fetch γίνονται επίσης και τα ορίσματά του) αλλά εδώ η διαφορά είναι στην πράξη του fetch. Για κάθε σύμβολο που πρόκειται να διαβαστεί ελέγχουμε κάθε φορά το αντίστοιχο bit από τα extra bytes πληροφορίας για την θέση των μπαλαντέρ. Αν η τιμή του bit είναι 1 τότε διαβάζεται το σύμβολο από τον buffer που δημιούργησε ο interpreter αλλιώς το σύμβολο διαβάζεται από το λεξικό. Τέλος, όταν διαβαστεί το σύμβολο END_OF_PATTERN αντί για κάποιο γνωστό opcode η εκτέλεση του pattern σταματά και ο έλεγχος του προγράμματος επιστρέφει στον αρχικό κώδικα.

ΚΕΦΑΛΑΙΟ 5. ΠΕΙΡΑΜΑΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

5.1. Εισαγωγικά

Ο κύριος στόχος των πειραμάτων που κάναμε ήταν να αξιολογηθεί η μέθοδος μας από όλες τις πιθανές σκοπιές: Από άποψη ποσοστών συμπίεσης και από άποψη επιπλέον χρόνου που εισάγεται κατά την εκτέλεση ενός συμπιεσμένου προγράμματος (overhead).

Το σύνολο εισόδου (input set) που χρησιμοποιήσαμε για τα πειράματά μας ήταν το MIDP (Mobile Information Device Profile). Το MIDP είναι ένα σύνολο από βιβλιοθήκες δημιουργημένο από την SUN με στόχο την διευκόλυνση αλλά και προτυποποίηση της δημιουργίας εφαρμογών JAVA για κινητές (mobile) και ενσωματωμένες (embedded) συσκευές. Στο σύνολό του, αποτελείται από 11 πακέτα. Στα πειράματα χρησιμοποιήθηκαν τα 6 από αυτά. Τα μεγέθη τους παρατίθενται στο παρακάτω σχήμα:

MIDP Package	Class file size (bytes)	Bytecode size (bytes)
java.io	19568	3454
java.lang	27714	3998
java.microedition.io	4599	509
java.microedition.media	2065	362
java.microedition.midlet	2025	121
java.microedition.pki	2563	184

Σχήμα 5.1. Τα Μεγέθη των Πακέτων του MIDP σε Bytes

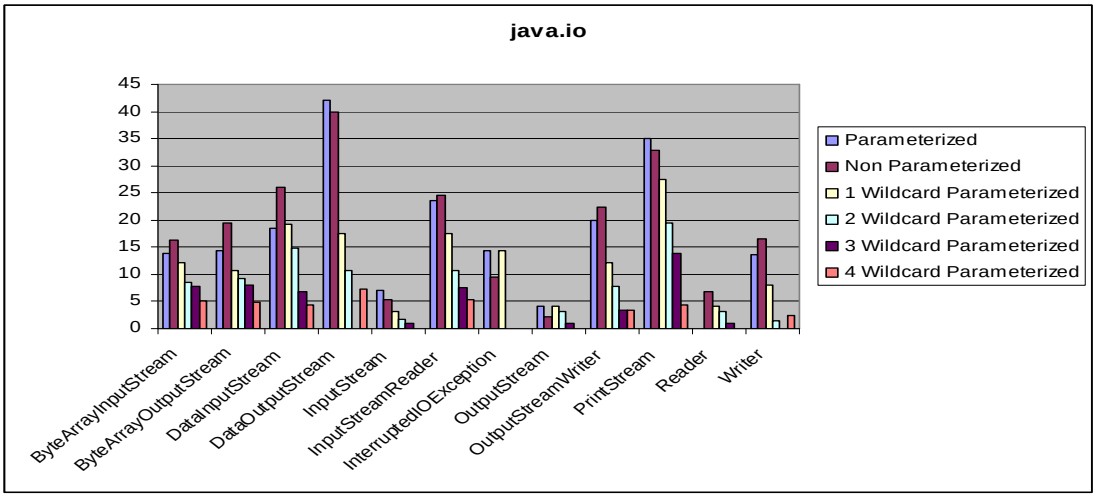
Ενδεικτικά λίγα λόγια για το κάθε πακέτο:

- το java.io παρέχει βασικές λειτουργίες για είσοδο / έξοδο από ροές δεδομένων (data streams).
- το java.lang περιέχει βασικές κλάσεις από την standard έκδοση της Java.
- το javax.microedition.io προσφέρει υποστήριξη δικτύου.
- το javax.microedition.media προσφέρει κλάσεις για την ανάπτυξη εφαρμογών πολυμέσων.
- το javax.microedition.midlet παρέχει τις βασικές κλάσεις για την ανάπτυξη εφαρμογών για κινητές και ενσωματωμένες συσκευές.
- το javax.microedition.pki προσφέρει υποστήριξη διαχείρισης πιστοποιητικών ασφαλείας για ασφαλείς συνδέσεις μέσω του internet.

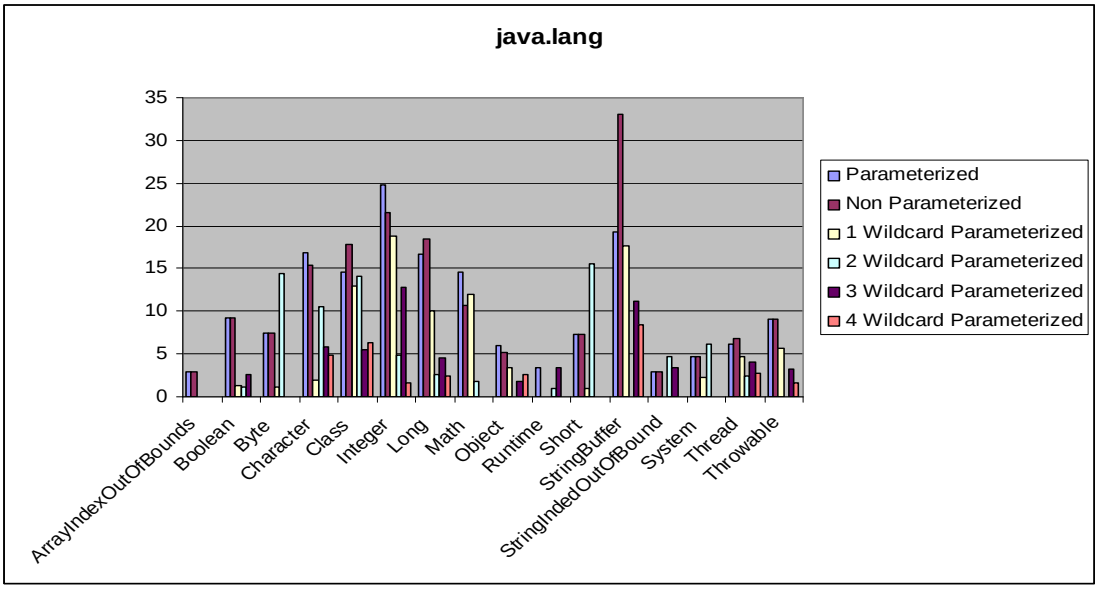
Όσον αφορά την άποψη των ποσοστών συμπίεσης, έγινε σύγκριση των δύο ευρετικών μεθόδων για μήκη patterns από 2 έως 9 και επίσης ως ξεχωριστό πείραμα συγκρίθηκαν ποσοστά συμπίεσης όταν το μέγιστο μήκος pattern αλλάζει από 9 σε 11.

5.2. Σύγκριση των Ευρετικών

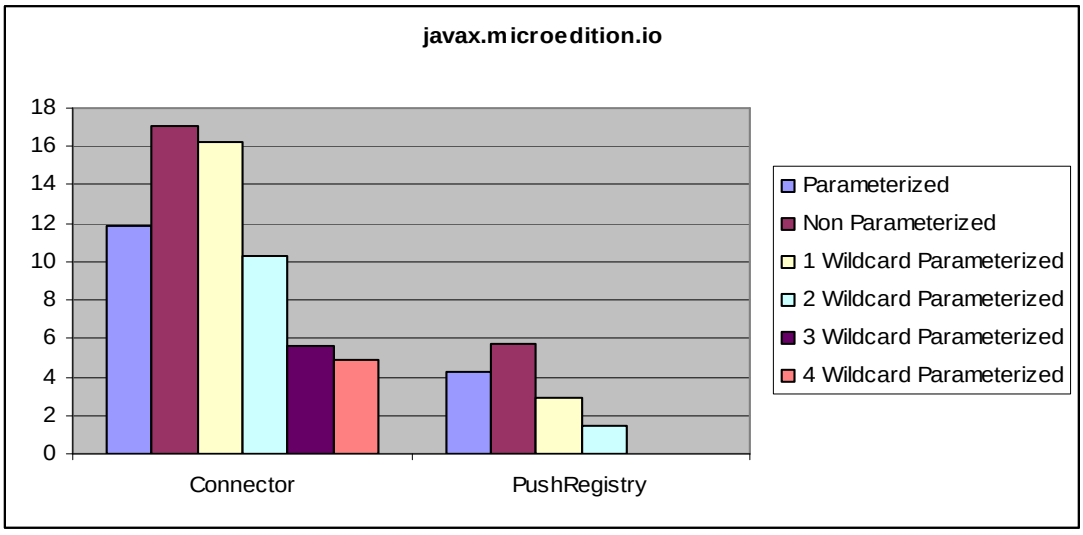
Στις παρακάτω γραφικές παραστάσεις φαίνεται η σύγκριση των ποσοστών συμπίεσης που παίρνουμε για κάθε πακέτο χρησιμοποιώντας την πρώτη ευρετική μέθοδο. Θέλουμε να δούμε εκτός από τα βασικά ποσοστά συμπίεσης που παράγονται από την παραμετρική και μη παραμετρική μέθοδο, ποια είναι η συνεισφορά των μπαλαντέρ στην όλη συμπίεση. Για το λόγο αυτό χρησιμοποιήσαμε και σύνολα με μόνο 1, μόνο 2, μόνο 3 και μόνο 4 μπαλαντέρ. Σε κάθε γραφική παράσταση συγκρίνονται τα ποσοστά χρησιμοποιώντας όλο το σύνολο των patterns που παρήχθησαν από την παραμετρική μέθοδο, το σύνολο των patterns από την μη παραμετρική μέθοδο, και επίσης τα σύνολα από patterns που περιέχουν patterns με μόνο 1, μόνο 2, μόνο 3 και μόνο 4 μπαλαντέρ στο εσωτερικό τους. Παρουσιάζονται τα αποτελέσματα ανά πακέτο του MIDP:



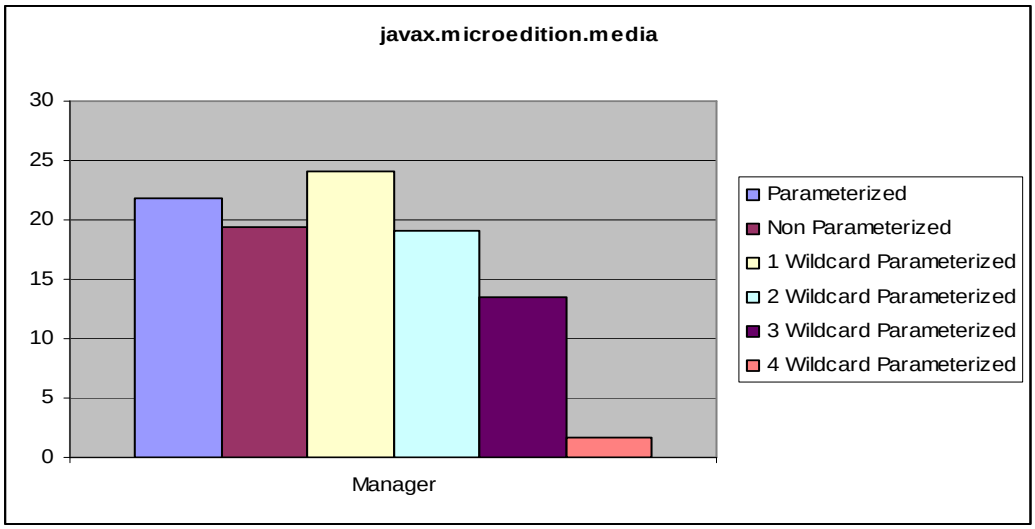
Σχήμα 5.2. Τα Ποσοστά Συμπίεσης των Διαφόρων Συνόλων για το Πακέτο Java.io



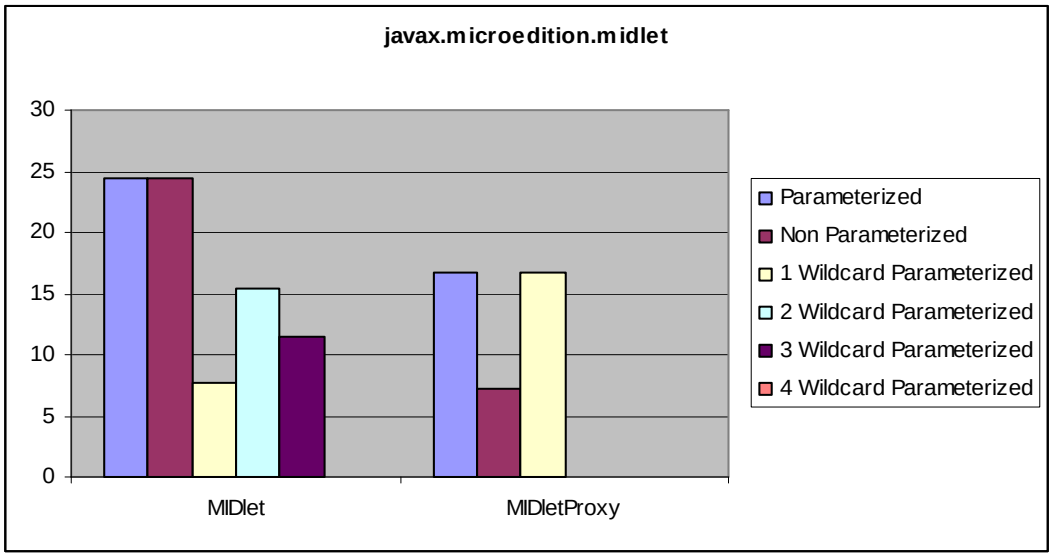
Σχήμα 5.3. Τα Ποσοστά Συμπίεσης των Διαφόρων Συνόλων για το Πακέτο Java.lang



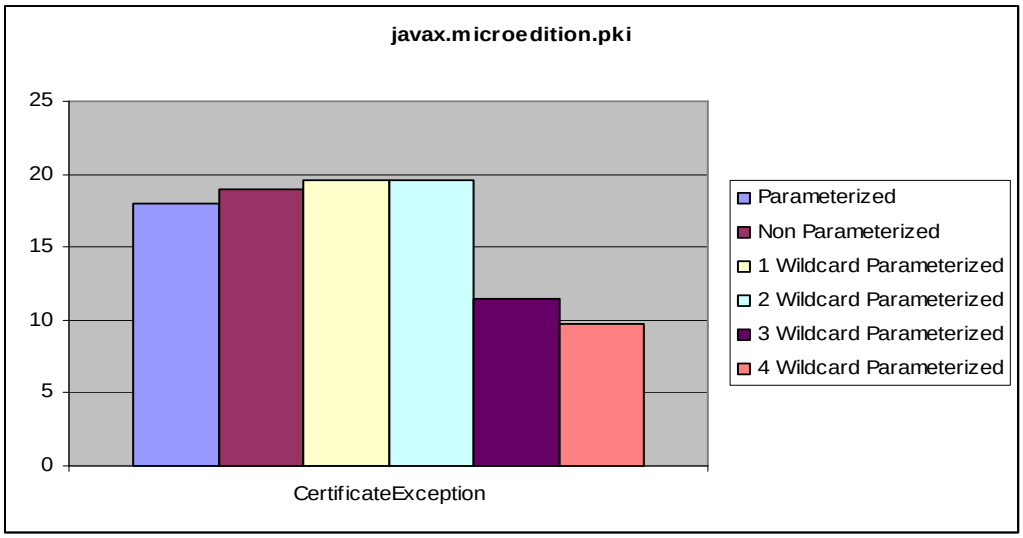
Σχήμα 5.4. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.io



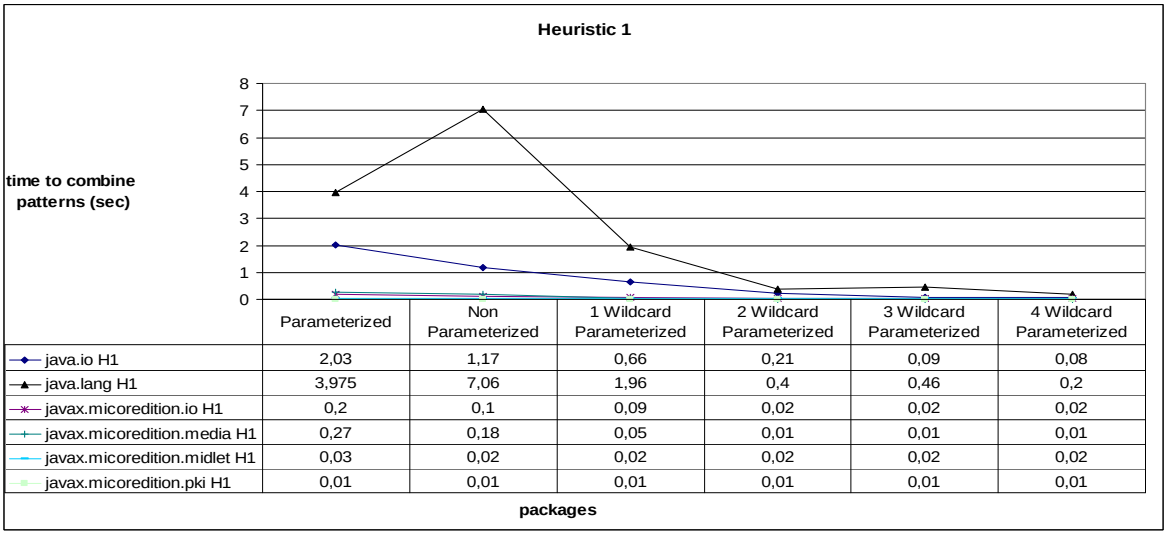
Σχήμα 5.5. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.media



Σχήμα 5.6. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.midlet



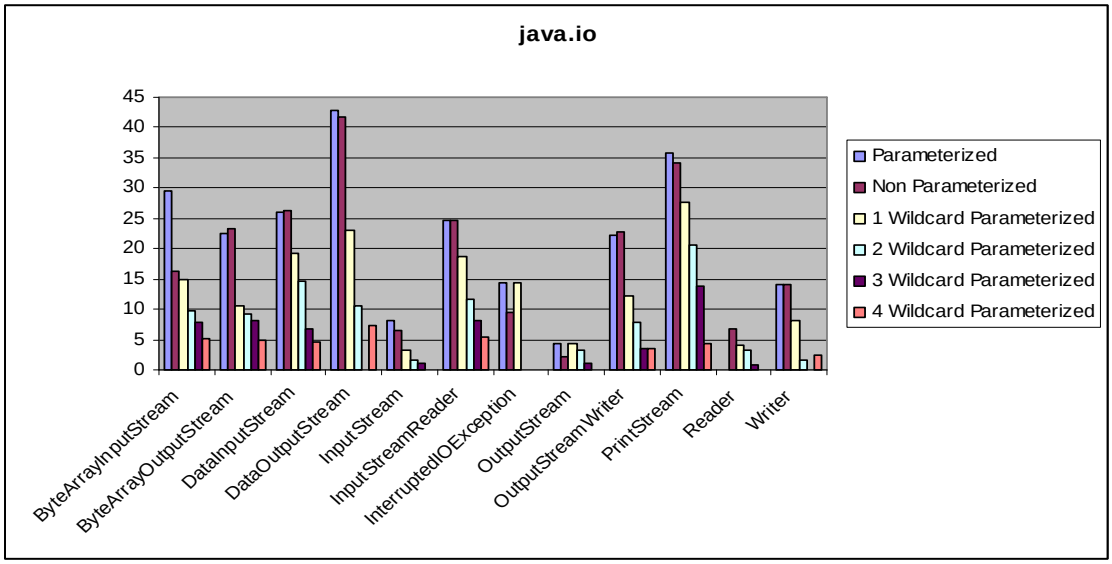
Σχήμα 5.7. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.pki



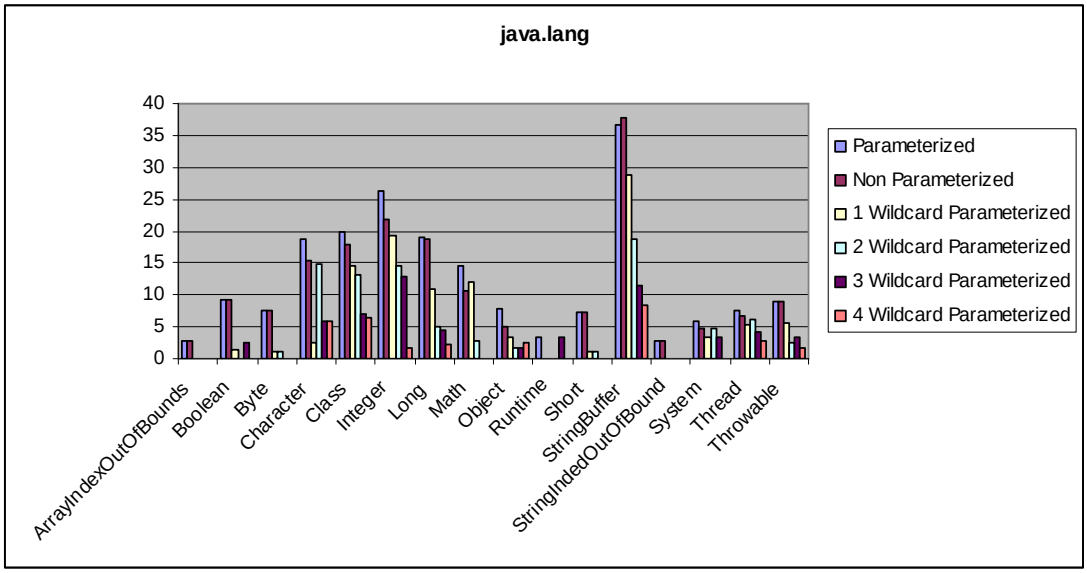
	Parameterized	Non Parameterized	1 Wildcard Parameterized	2 Wildcard Parameterized	3 Wildcard Parameterized	4 Wildcard Parameterized
java.io H1	2,03	1,17	0,66	0,21	0,09	0,08
java.lang H1	3,975	7,06	1,96	0,4	0,46	0,2
javax.micoredition.io H1	0,2	0,1	0,09	0,02	0,02	0,02
javax.micoredition.media H1	0,27	0,18	0,05	0,01	0,01	0,01
javax.micoredition.midlet H1	0,03	0,02	0,02	0,02	0,02	0,02
javax.micoredition.pki H1	0,01	0,01	0,01	0,01	0,01	0,01

Σχήμα 5.8. Οι Συνολικοί Χρόνοι σε Δευτερόλεπτα της Πρώτης Ευρετικής

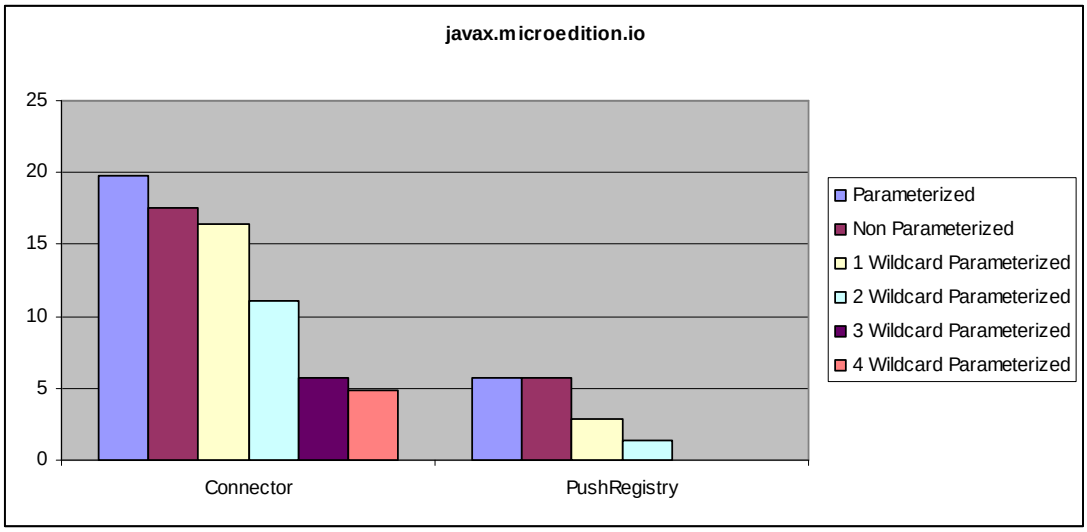
Προχωρώντας στην σύγκριση, παραθέτονται τα αντίστοιχα νούμερα για την δεύτερη ευρετική:



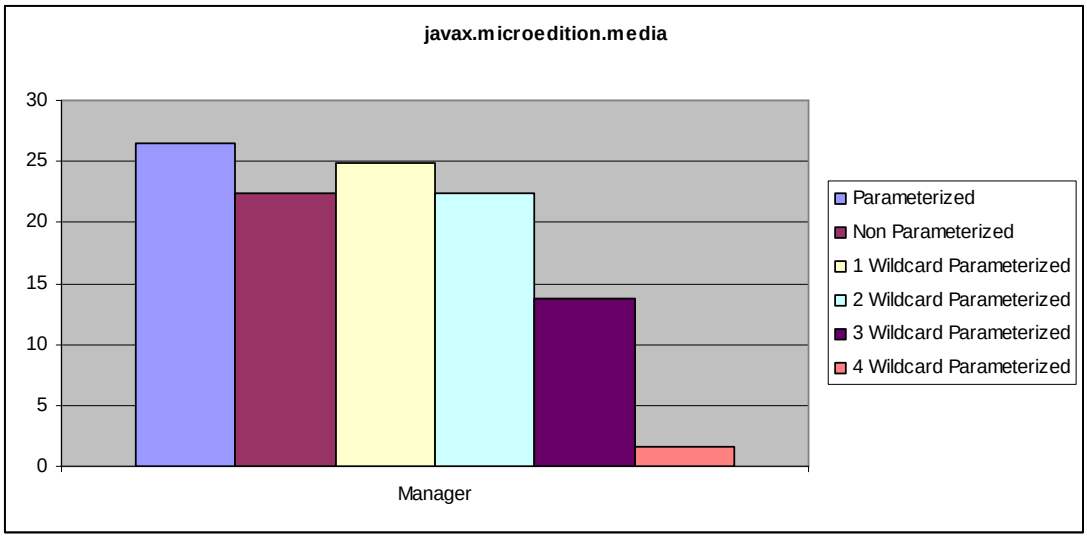
Σχήμα 5.9. Τα Ποσοστά Συμπίεσης της Δεύτερης Ευρετικής για το Πακέτο java.io



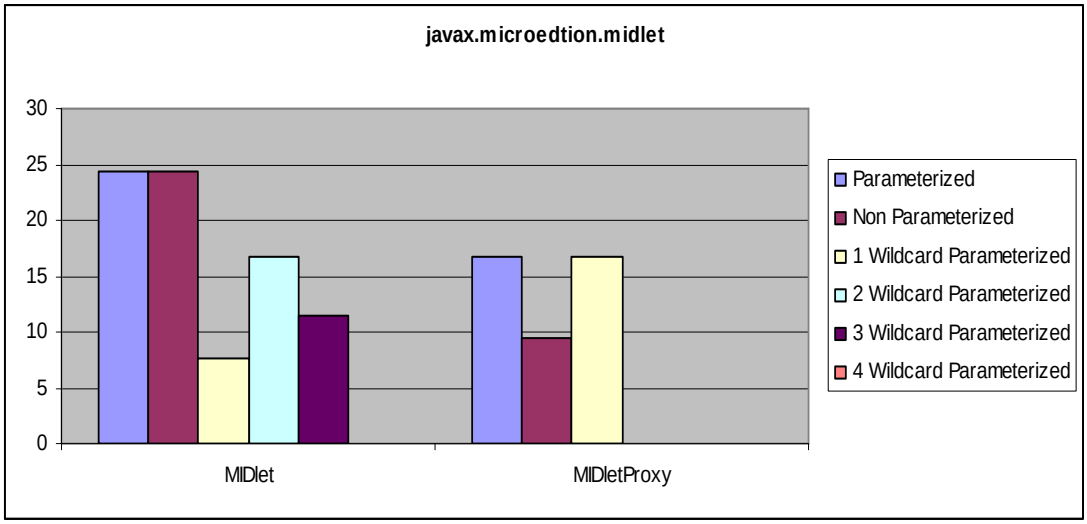
Σχήμα 5.10. Τα Ποσοστά Συμπίεσης της Δεύτερης Ευρετικής για το Πακέτο java.lang



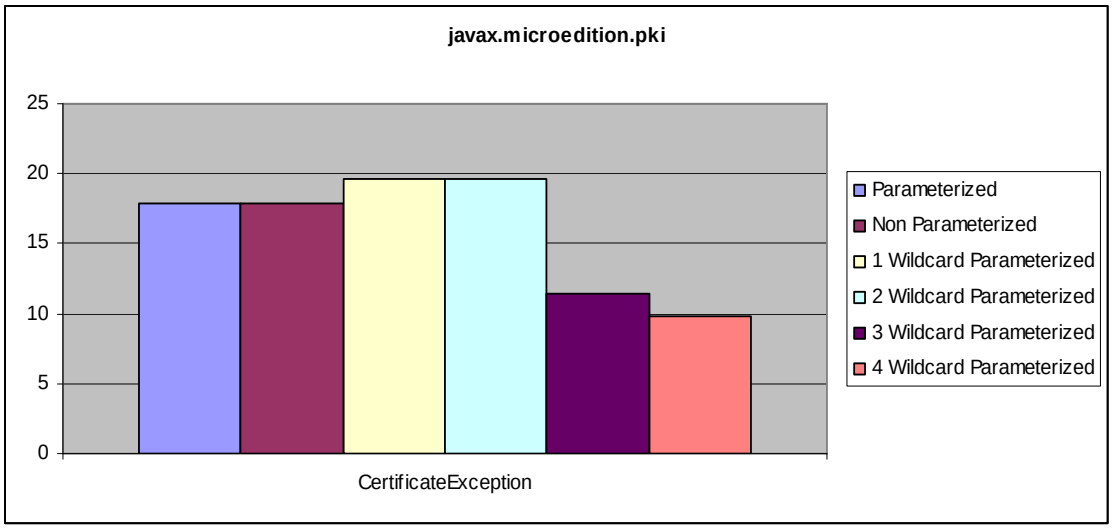
Σχήμα 5.11. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.io



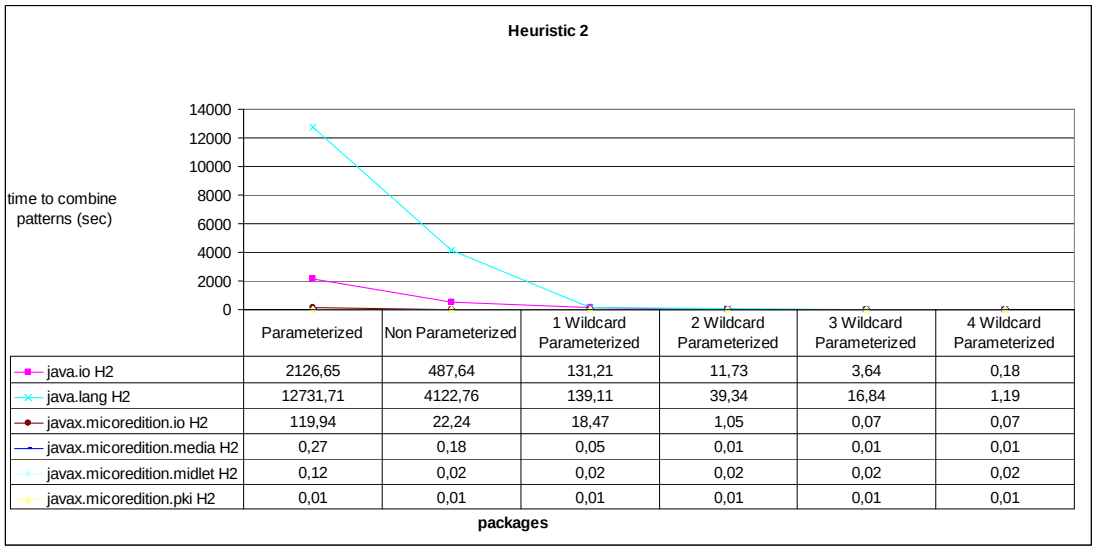
Σχήμα 5.12. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.media



Σχήμα 5.13. Τα Ποσοστά Συμπίεσης για το Πακέτο javax.microedition.midlet



Σχήμα 5.14. Τα ποσοστά Συμπίεσης για το Πακέτο javax.microedition.pki



	Parameterized	Non Parameterized	1 Wildcard Parameterized	2 Wildcard Parameterized	3 Wildcard Parameterized	4 Wildcard Parameterized
java.io H2	2126,65	487,64	131,21	11,73	3,64	0,18
java.lang H2	12731,71	4122,76	139,11	39,34	16,84	1,19
javax.micoredition.io H2	119,94	22,24	18,47	1,05	0,07	0,07
javax.micoredition.media H2	0,27	0,18	0,05	0,01	0,01	0,01
javax.micoredition.midlet H2	0,12	0,02	0,02	0,02	0,02	0,02
javax.micoredition.pki H2	0,01	0,01	0,01	0,01	0,01	0,01

Σχήμα 5.15. Οι Συνολικοί Χρόνοι Εκτέλεσης Δεύτερης Ευρετικής ανά Πακέτο

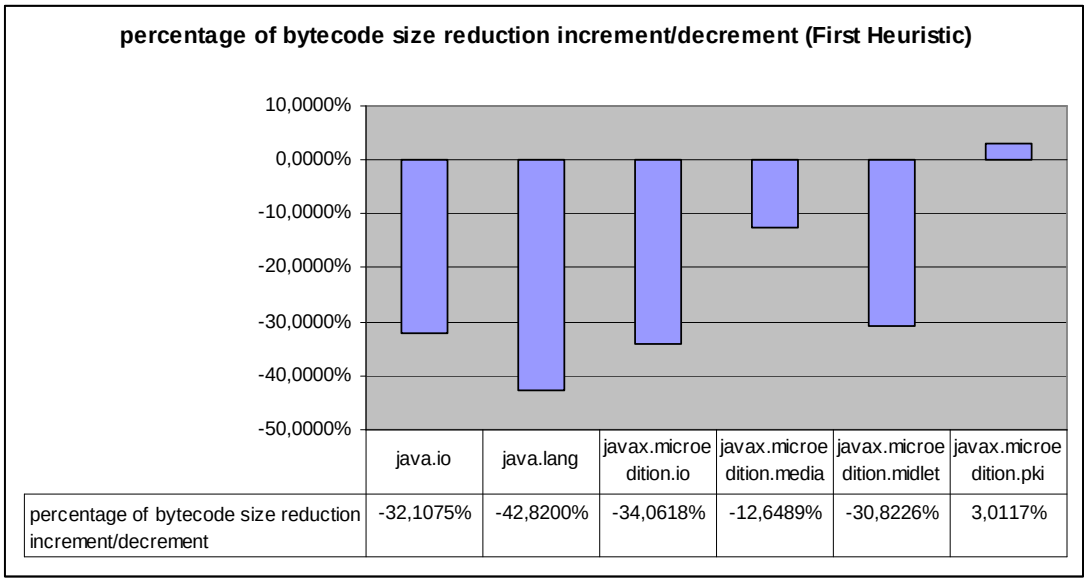
Παρατηρώντας τα αποτελέσματα από τις 2 μεθόδους συμπεραίνουμε ότι το κύριο πλεονέκτημα της πρώτης ως προς την δεύτερη είναι ο πολύ μικρότερος χρόνος εκτέλεσης. Κατά τα άλλα, η πρώτη ευρετική είναι πολύ απρόβλεπτη σε σχέση με την δεύτερη. Δεν μπορούμε να δούμε κάποιο μοτίβο στα αποτελέσματα που παράγει και καταλήγουμε να πούμε ότι η έξοδος της πρώτης ευρετικής στηρίζεται πολύ στην σειρά με την οποία θα τοποθετηθούν τα patterns μέσα στο αρχικό σύνολο εισόδου της ευρετικής.

Σε αντίθεση, η δεύτερη ευρετική δίνει τα αναμενόμενα αποτελέσματα, το σύνολο που περιέχει όλα τα patterns (παραμετρικά και μη) προσφέρει και το μεγαλύτερο ποσοστό συμπίεσης ενώ ακολουθεί το σύνολο με τα μη παραμετρικά patterns. Ο χρόνος που απαιτεί η δεύτερη ευρετική είναι κατά πολύ μεγαλύτερος από αυτόν της πρώτης αλλά και πάλι παραμένει σε αποδεκτά πλαίσια. Συνεπώς, δεν υπάρχει κανένας λόγος να

προτιμηθεί η πρώτη ευρετική έναντι της δεύτερης εκτός και αν ο χρόνος εκτέλεσης έχει πολύ μεγάλη σημασία

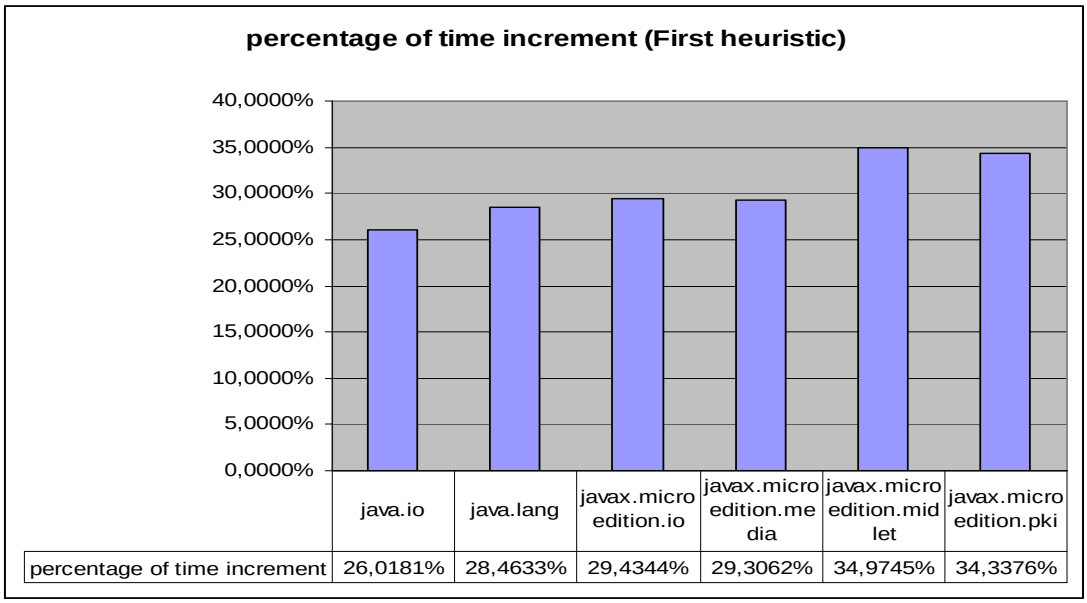
Όσον αφορά τα patterns με μόνο 1, 2, 3, 4 μπαλαντέρ το συμπέρασμά μας είναι ότι όσο λιγότερα μπαλαντέρ περιέχονται μέσα στο pattern τόσο μεγαλύτερο είναι και το ποσοστό συμπίεσης που αυτό επιφέρει. Αυτό είναι και αναμενόμενο μιας και όσα πιο πολλά μπαλαντέρ περιέχονται σε ένα pattern τόσο λιγότερη πληροφορία κωδικοποιείται από αυτό.

Εκτός από αυτά τα πειράματα, δοκιμάσαμε να δούμε την αλλαγή στο ποσοστό συμπίεσης που επέρχεται όταν αλλάζουμε το μέγιστο μήκος pattern από 9 σε 11. Όπως ξέρουμε με ένα επιπλέον byte πληροφορίας μπορούμε να κωδικοποιήσουμε την θέση των μπαλαντέρ σε patterns με μήκος το πολύ 10. Ξεπερνώντας πλέον αυτό το κατώφλι τα δεδομένα αλλάζουν. Μπορεί μεν να παράγονται πιο πολλά patterns αυτό δεν σημαίνει όμως ότι θα έχουμε και αύξηση του συνολικού ποσοστού συμπίεσης. Για κάθε pattern που αποθηκεύεται στο λεξικό χρειαζόμαστε 2 επιπλέον bytes πληροφορίας και αυτό με την σειρά του μπορεί να οδηγήσει σε μείωση του συνολικού ποσοστού συμπίεσης. Επίσης αύξηση του μέγιστου μήκους οδηγεί σε αύξηση του χρόνου εκτέλεσης του Agglomerative Clustering καθώς και σε αύξηση του χρόνου εκτέλεσης των δύο ευρετικών συναρτήσεων. Τα αποτελέσματα που πήραμε από την αλλαγή του μέγιστου μήκους για τις δύο ευρετικές φαίνονται στις επόμενες δύο εικόνες:



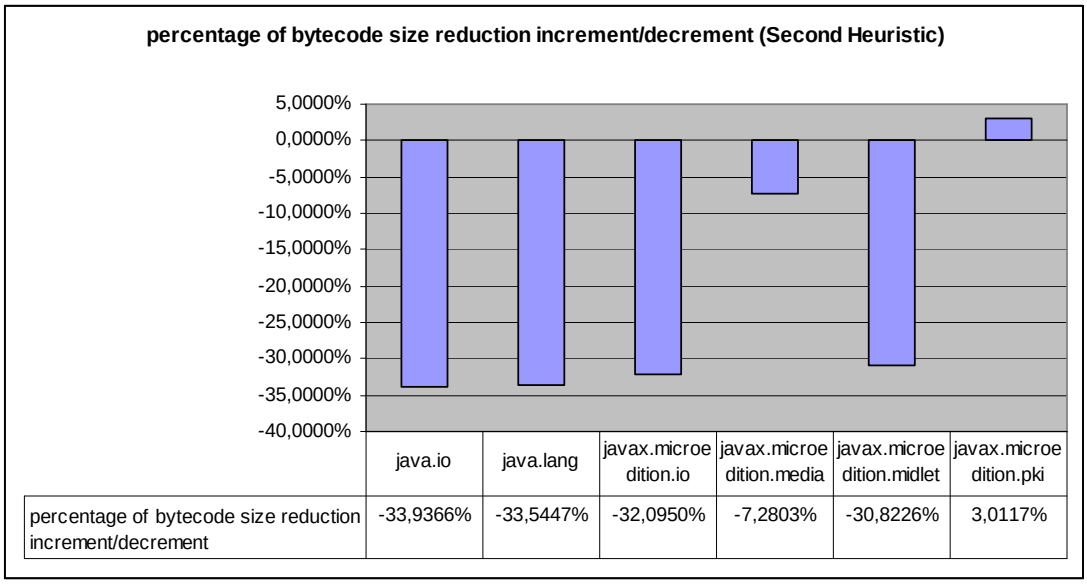
	java.io	java.lang	javax.microedition.io	javax.microedition.media	javax.microedition.midlet	javax.microedition.pki
percentage of bytecode size reduction increment/decrement	-32,1075%	-42,8200%	-34,0618%	-12,6489%	-30,8226%	3,0117%

Σχήμα 5.16. Η Μεταβολή στα Ποσοστά Συμπίεσης με Μέγιστο Μήκος Pattern 11



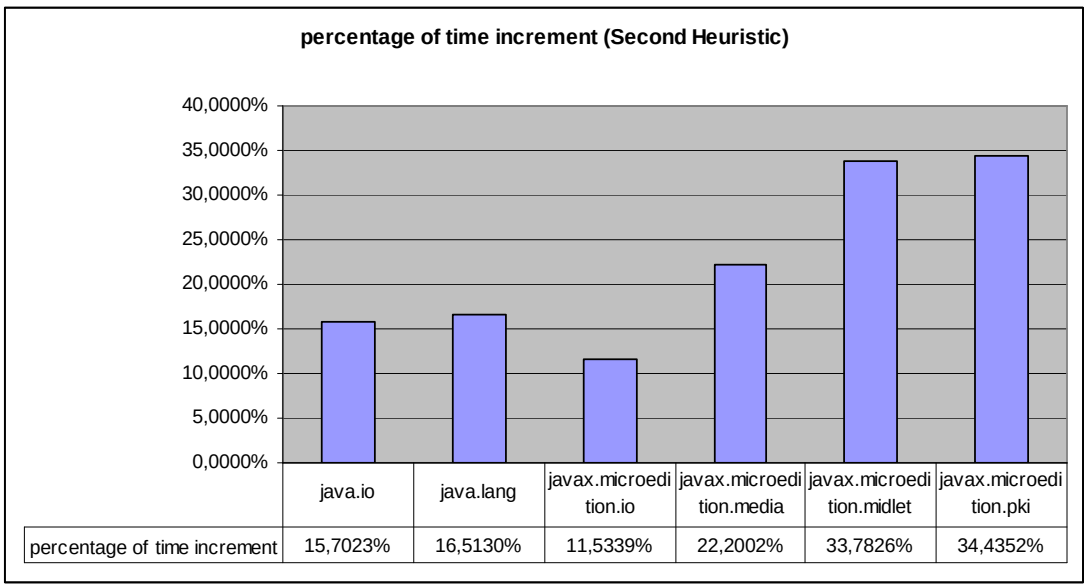
	java.io	java.lang	javax.microedition.io	javax.microedition.media	javax.microedition.midlet	javax.microedition.pki
percentage of time increment	26,0181%	28,4633%	29,4344%	29,3062%	34,9745%	34,3376%

Σχήμα 5.17. Η Μεταβολή του Χρόνου Εκτέλεσης της Ευρετικής με Μήκος Pattern 11



	java.io	java.lang	javax.microedition.io	javax.microedition.media	javax.microedition.midlet	javax.microedition.pki
percentage of bytecode size reduction increment/decrement	-33,9366%	-33,5447%	-32,0950%	-7,2803%	-30,8226%	3,0117%

Σχήμα 5.18. Η Μεταβολή των Ποσοστών Συμπίεσης στην Δεύτερη Ευρετική



	java.io	java.lang	javax.microedition.io	javax.microedition.media	javax.microedition.midlet	javax.microedition.pki
percentage of time increment	15,7023%	16,5130%	11,5339%	22,2002%	33,7826%	34,4352%

Σχήμα 5.19. Η Μεταβολή του Χρόνου Εκτέλεσης της Δεύτερης Ευρετικής

Παρατηρούμε λοιπόν μια αύξηση του χρόνου εκτέλεσης των ευρετικών στο εύρος 11% με 34% η οποία όμως συνοδεύεται από μείωση των ποσοστών συμπίεσης σε όλες τις περιπτώσεις εκτός από το πακέτο javax.microedition.pki το οποίο μας οδηγεί

στο συμπέρασμα ότι αύξηση της επιπλέον πληροφορίας για τα μπαλαντέρ στο λεξικό οδηγεί σε καταστρεπτικά αποτελέσματα στα ποσοστά.

Η πιο σωστή τακτική λοιπόν είναι να βρίσκουμε τα ποσοστά συμπίεσης με μέγιστο μήκος pattern 10 και έπειτα να δοκιμάζουμε και για μεγαλύτερα μήκη αν θέλουμε να επιτύχουμε κάτι καλύτερο αλλά θα πρέπει να είμαστε προετοιμασμένοι για μείωση του τελικού ποσοστού συμπίεσης

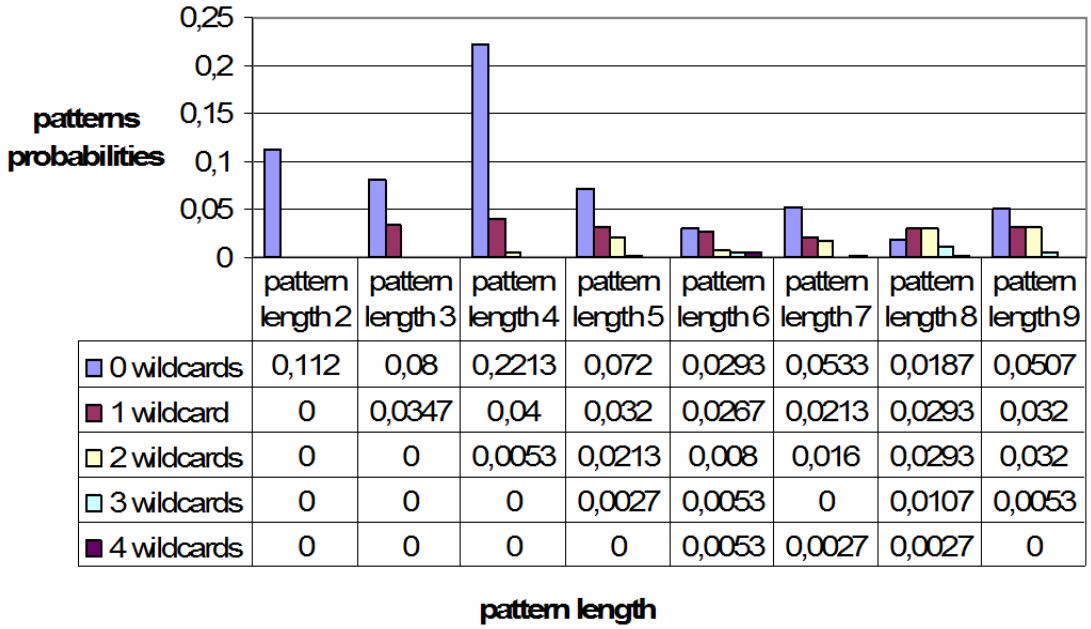
5.3. Αξιολόγηση του Overhead

Για να αξιολογήσουμε σωστά το overhead που εισάγεται στην εκτέλεση ενός συμπιεσμένου προγράμματος, χρειαστήκαμε ένα σύνολο από δοκιμαστικά προγράμματα για να κάνουμε τις μετρήσεις μας. Για τον σκοπό αυτό, δημιουργήσαμε μια γεννήτρια (generator) τυχαίων προγραμμάτων το οποίο δοθέντος ενός ζητούμενου ποσοστού συμπίεσης παράγει ένα τμήμα bytecode το οποίο όταν συμπιεστεί με την μέθοδό μας να συμπιέζεται κατά αυτό το ποσοστό. Για την παραγωγή του εκάστοτε bytecode ο generator χρειάζεται γνώση των εξής χαρακτηριστικών:

- Το μέγιστο μήκος K των patterns. Με την πληροφορία αυτή παράγονται patterns ομοιόμορφα κατανεμημένα στο διάστημα [2, K].
- Το μέγιστο πλήθος των μπαλαντέρ M. Το πλήθος των μπαλαντέρ στο εκάστοτε pattern είναι στο διάστημα [0, M].
- Το μέγιστο πλήθος εμφανίσεων των patterns L. Το πλήθος των επαναλήψεων του κάθε pattern είναι στο διάστημα [2,L].

Εκτός όμως αυτών των χαρακτηριστικών, σημαντικό ρόλο παίζει ένας δυσδιάστατος πίνακας KxM ο οποίος στην κάθε θέση του [k, m] περιέχει την πιθανότητα εμφάνισης

ενός pattern μήκους k με m μπαλαντέρ στο εσωτερικό του. Ο τρόπος με τον οποίο δημιουργήθηκε αυτός ο πίνακας είναι απλός: Για κάθε παράδειγμα του MIDP μετρήσαμε ποια patterns συνδυάστηκαν στην τελική συμπίεση και αθροίσαμε όλες τις εμφανίσεις των patterns σε όλα τα παραδείγματα. Με τον τρόπο αυτό πήραμε συχνότητες εμφάνισης του κάθε pattern. Οι συχνότητες που πήραμε παρουσιάζονται στο επόμενο σχήμα:



Σχήμα 5.20. Οι Συχνότητες Εμφάνισης που Χρησιμοποιήθηκαν στον Generator

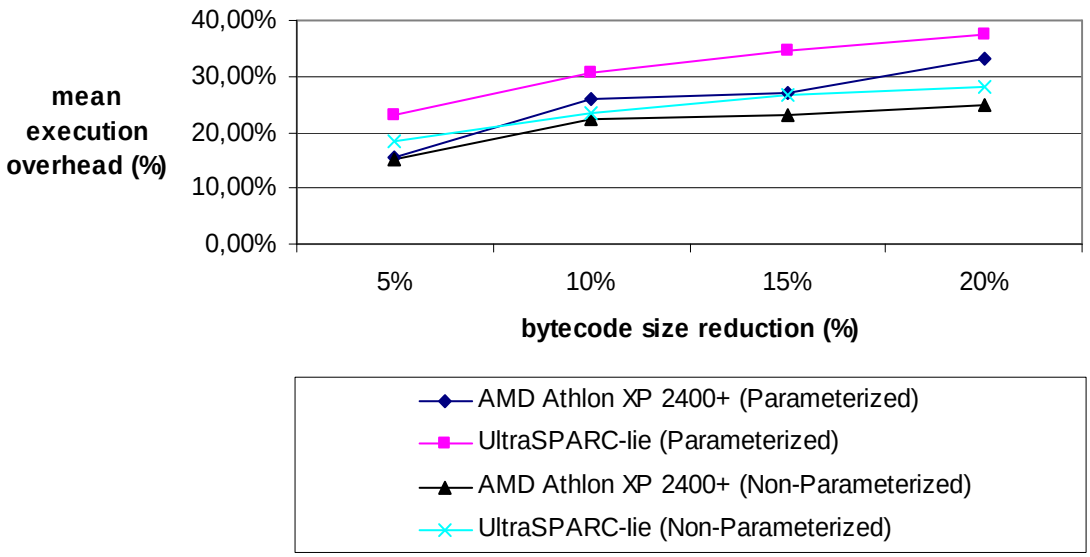
Βάσει όλων των παραπάνω δημιουργήσαμε 4 σετ εισόδου παραμετρικών patterns με επιθυμητά ποσοστά συμπίεσης 5%, 10%, 15% και 20% αντίστοιχα. Ομοίως παράξαμε και αντίστοιχα σύνολα εισόδου για την μη παραμετρική μέθοδο.

Για την μέτρηση του overead υλοποιήσαμε το main loop μιας JVM το οποίο επαυξήθηκε με τα 2 decompression modules (ένα για την παραμετρική και ένα για

την μη παραμετρική τεχνική). Το πρόγραμμα αυτό εκτελέστηκε σε δύο διαφορετικές αρχιτεκτονικές υπολογιστών:

- Έναν Athlon XP 2400+ στα 2.0 Ghz με 256kb L2 cache
- Έναν 500Mhz UltraSparc-II με 256kb L2 cache

Τα τελικά αποτελέσματα που πήραμε παρουσιάζεται στην παρακάτω εικόνα:

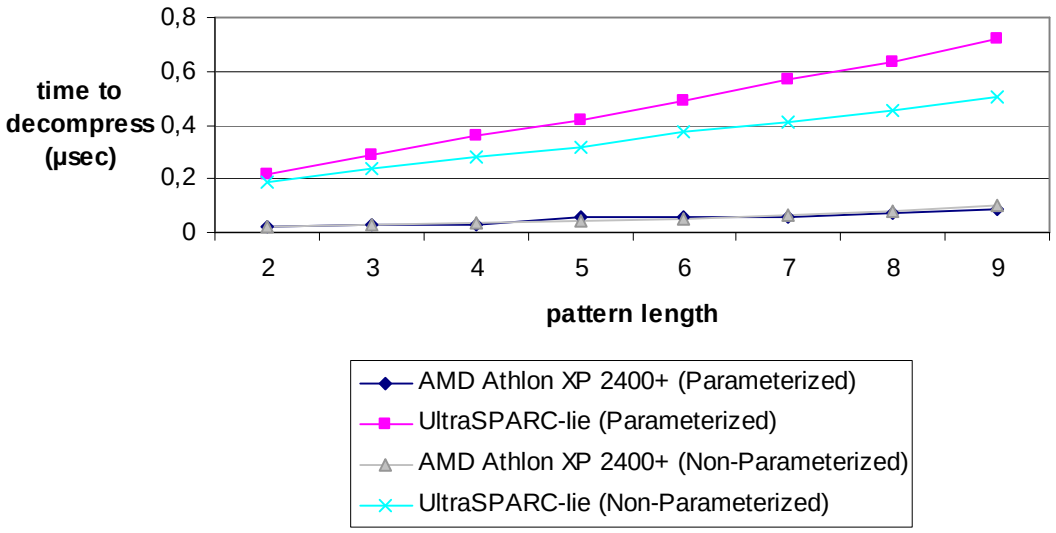


Σχήμα 5.21. Η Αύξηση του Overhead σε Σχέση με το Επιθυμητό Ποσοστό Συμπίεσης

Παρατηρούμε ότι το overhead αυξάνεται με περίπου γραμμικό ρυθμό σε σχέση με το ποσοστό συμπίεσης αλλά κυμαίνεται σε αποδεκτά επίπεδα. Αυτό είναι λογικό εφ' όσων όσο μεγαλύτερο είναι το ποσοστό συμπίεσης, τόσο πιο πολλά patterns έχουν εμφανιστεί μέσα στον bytecode και άρα τόσες πιο πολλές κλήσεις γίνονται στο εκάστοτε decompression module. Επίσης παρατηρούμε ότι η παραμετρική μέθοδος εισάγει μεγαλύτερο overhead σε σχέση με την μη παραμετρική μιας και η πολυπλοκότητα εκτέλεσης ενός παραμετρικού pattern είναι μεγαλύτερη (πρέπει σε

κάθε βήμα να κοιτάμε το bit πληροφορίας για να ξέρουμε ποιο σύμβολο να διαβάσουμε)

Εκτός από αυτές τιμές μετρήσαμε τους απόλυτους χρόνους που απαιτεί η αποσυμπίεση ενός pattern μήκους από 2 έως 9. Τα αποτελέσματα παρουσιάζονται στην επόμενη εικόνα (οι χρόνοι δίνονται σε μικροδευτερόλεπτα):



Σχήμα 5.22. Οι Απόλυτοι Χρόνοι Αποσυμπίεσης Ενός Pattern σε Σχέση με το Μήκος

Σε αυτήν την εικόνα φαίνεται καθαρά ότι ο χρόνος αποσυμπίεσης ενός pattern αυξάνεται γραμμικά με το μήκος του pattern. Επίσης είναι φανερή η διαφορά στον χρόνο που απαιτεί η παραμετρική μέθοδος σε σχέση με την μη παραμετρική

ΚΕΦΑΛΑΙΟ 6. ΣΥΜΠΕΡΑΣΜΑΤΑ

Στην εργασία αυτή, μελετήσαμε την εφαρμογή των παραμετρικών patterns στον τομέα της συμπίεσης του bytecode. Η μέθοδος στο σύνολό της στηρίζεται στην δημιουργία ενός λεξικού από επαναλαμβανόμενων σειρών εντολών μέσα στον bytecode (τα οποία ονομάζονται patterns) και αντικατάσταση των σειρών αυτών από δείκτες στο λεξικό. Η μέθοδος μπορεί να υστερεί σε τελικά ποσοστά συμπίεσης σε σχέση με άλλες μεθόδους που βασίζονται στην αριθμητική κωδικοποίηση και σε κώδικες huffman αλλά έχει το μεγάλο πλεονέκτημα ότι τα συμπιεσμένα προγράμματα εκτελούνται ως έχουν χωρίς να χρειάζονται την οποιαδήποτε αποσυμπίεση εκ των προτέρων (ολική ή μερική)

Για να παράξουμε τα patterns χρησιμοποιήσαμε μια γνωστή μέθοδο για ομαδοποίηση (clustering) την οποία και προσαρμόσαμε για να μας δίνει τα αποτελέσματα που θέλουμε. Η χρήση αυτής της μεθόδου αν και μας έδωσε πολύ καλά τελικά ποσοστά δημιούργησε ένα νέο πρόβλημα: Παράγει πολύ μεγάλο πλήθος από patterns. Αυτό με την σειρά του μας απαγορεύει από το να αναζητήσουμε την βέλτιστη λύση μέσα σε όλους τους δυνατούς συνδυασμούς. Για τον λόγο αυτό δημιουργήσαμε δύο ευρετικές μεθόδους οι οποίες δίνουν πολύ καλά υποβέλτιστα αποτελέσματα. Κυρίως σταθήκαμε στην δεύτερη μιας και λειτουργεί πολύ καλύτερα από την πρώτη (αλλά είναι και πιο αργή) και τελικά σε κάποιο λογικά μικρό και αποδεκτό χρονικό

διάστημα καταλήγουμε να έχουμε ένα πολύ καλό (που σε μερικές περιπτώσεις μπορεί να είναι και βέλτιστο) αποτέλεσμα.

Από το σημείο που είχαμε τον τελικό συνδυασμό και το συμπιεσμένο αποτέλεσμα, θελήσαμε να εξετάσουμε τον αντίκτυπο που έχει η χρήση των patterns στην εκτέλεση των προγραμμάτων. Δημιουργήσαμε με τυχαίο τρόπο (ο οποίος όμως στηρίζονταν στα πειραματικά αποτελέσματα που πήραμε) μια πλειάδα από παραδείγματα για τις μετρήσεις μας (benchmarks). Αυτό που είδαμε είναι το overhead μπορεί σε υψηλά ποσοστά συμπίεσης να αγγίξει το 40% το οποίο όμως παρόλα αυτά παραμένει αποδεκτό ποσοστό για πολλές εφαρμογές.

Το βασικό συμπέρασμα που παίρνουμε από όλα τα πειράματα είναι ότι η παραμετρική μέθοδος μπορεί να δώσει πολύ καλά αποτελέσματα αλλά μόνο όταν συνδυάσουμε και παραμετρικά και μη παραμετρικά patterns. Το να βασιστούμε μόνο στα παραμετρικά οδηγεί σε πολύ χαμηλά ποσοστά τα οποία μάλιστα μειώνονται καθώς αυξάνεται το πλήθος των μπαλαντέρ στο εσωτερικό των patterns. Το μέγιστο μήκος των patterns προτείνεται να είναι το πολύ 10 ώστε να βρισκόμαστε στο όριο του 1 extra byte πληροφορίας και επίσης τα παραδείγματα μας έδειξαν ότι την μεγαλύτερη συχνότητα εμφάνισης παρουσιάζουν τα patterns με μήκος 4. Αύξηση του μέγιστου μήκους απαιτεί αλλαγές στην μέθοδο ώστε να διαχωριστούν τα patterns που είναι αποθηκευμένα στο λεξικό σε κατηγορίες ανάλογα με το πλήθος των extra bytes που απαιτούν για την αποθήκευσή τους. Αν δεν γίνει αυτό τότε κατά πάσα πιθανότητα αύξηση του μέγιστου μήκους οδηγεί σε μείωση του συνολικού ποσοστού συμπίεσης.

Ως μελλοντική δουλειά θέλουμε να δοκιμάσουμε και άλλες μεθόδους εύρεσης patterns καθώς επίσης και μια πιο ευρεία γκάμα από ευρετικές συναρτήσεις για την

εύρεση συνδυασμών (όπως είναι η προσομοιούμενη ανόπτυση) οι οποίες να μπορούν να δίνουν εξίσου καλά αποτελέσματα αλλά σε πολύ λιγότερο χρόνο.

ΑΝΑΦΟΡΕΣ

---

[1] A. Beszedes, R. Ferenc, T. Gyim´othy, A. Dolenc, and K. Karsisto, “Survey of Code-Size Reduction Methods,” *ACM Computing Surveys*, vol. 35, no. 3, pp. 223–267, 2003.

[2] D. Rayside, E. Mamas, and E. Hons, “Compact Java Binaries for Embedded Systems,” in Proceedings of the 1999 ACM Conference of the Centre for Advanced Studies on Collaborative Research (CASCON’99), 1999, pp. 1–14.

[3] M. Latendresse and M. Feeley, “Generation of Fast Interpreters for Huffman Compressed Bytecode,” *Science of Computer Programming (Advances in Interpreters, Virtual Machines and Emulators)*, vol. 57, no. 3, pp. 295–317, 2005.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller, “Java Bytecode Compression for Low-End Embedded Systems,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 3, pp. 471–489, 2000.

[5] M. Meila and D. Hecherman, “An experimental comparison of model-based clustering methods,” *Machine Learning*, vol. 42, pp. 9–29, 2001.

[6] K. Blekas and A. Likas, “Incremental Mixture Learning for Clustering Discrete Data,” in *Lecture Notes in Artificial Intelligence*, vol. 3025. Springer-Verlag, 2004, pp. 210–219.

[7] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, “Code Compression,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’97)*, 1997, pp. 358–365.

[8] Q. Brandly, R. N. Horspool, and J. Viter, “JAZZ: An Efficient Compressed Format for Java Archive Files,” in *Lecture Notes in Artificial Intelligence*, vol. 3025. Springer-Verlag, 2004, pp. 210–219.

[9] M. Franz and T. Kistler, “Slim Binaries,” *Communications of the ACM*, vol. 40, no. 12, pp. 87–94, 1997.

[10] W. Pugh, “Compressing Java Class Files,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’99)*, 1999, pp. 247–258.

[11] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter, “Practical Extraction Techniques for Java,” *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 6, pp. 625–666, 2002.

[13] M. Kozuch and A. Wolfe, “Compression of Embedded System Programs,” in *Proceedings of the International IEEE Conference on Computer Design: VLSI in Computers & Processors*, 1994, pp. 270–277.

[14] H. Lekatsas and A. Wolfe, “SAMC: A Code Compression Algorithm for Embedded Processors,” *IEEE Transactions on Computer Aided Design*, vol. 18, no. 12, pp. 1689–1701, 1999.

[16] IBM, “CodePack: PowerPC Code Compression Utility User’s Manual v3.0,” IBM Corporation, Tech. Rep., 1998.

[17] C. R. Lefurgy, P. L. Bird, I-C. Chen and T. N. Mudge, “Improving Code Density Using Compression Techniques,” in *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture (Micro-30)*, 1997, pp. 194–203.

[18] S. K. Debray, W. Evans, R. Muth and B. De Sutter, “Compiler Techniques for Code Compaction,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 378–415, 2000.

[19] B. De Sutter, B. De Bus and K. De Bosschere, “Sifting out the Mud: Low Level C++ Code Reuse,” in *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA’02)*, 2002, pp. 275–291.

[20] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” in *Proceedings of the 8th IEEE Working Conference on Reverse Engineering*, 2001, pp. 301–309.

[21] W. Cheung, W. Evans, and J. Moses, *Software and Compilers for Embedded Systems: 7th International Workshop (SCOPES’03)*, ser. LNCS. Springer-Verlag, 2003, vol. 2826, ch. Predicated Instructions for Code Compaction, pp. 17–31.

[22] W. Evans and C. Fraser, “Bytecode Compression via Profiled Grammar Rewriting,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’01)*, 2001, pp. 148–155.

ΣΥΝΤΟΜΟ ΒΙΟΓΡΑΦΙΚΟ

Ο Σαούγκος Δημήτριος γεννήθηκε το 1983 στα Ιωάννινα. Εισήχθη στο τμήμα Πληροφορικής της Σχολής Θετικών Επιστημών του Πανεπιστημίου Ιωαννίνων το 2000 από το οποίο οποίο αποφοίτησε το 2004. Στο διάστημα 2004 με 2006 παρακολούθησε το μεταπτυχιακό πρόγραμμα σπουδών του ιδίου τμήματος.