ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

Υπολογιστικοί Αλγόριθμοι για τις Άλγεβρες Yokonuma-Hecke

Καρβούνης Κωνσταντίνος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επιβλέπουσα: Σοφία Λαμπροπούλου, ΕΜΠ

Περιεχόμενα

0 Εισαγωγή 3
1 Κόμβοι και κοτσίδες 5
1.1 Κόμβοι 5
1.2 Ομάδες κοτσίδων (braid group) 7
1.2.1 Απλές κοτσίδες 7
1.2.2 Πλαισιωμένες κοτσίδες 10
1.3 Άλγεβρες Hecke και Yokonuma-Hecke 13
1.3.1 Άλγεβρες Hecke 13
1.3.2 Άλγεβρες Yokonuma-Hecke 15
1.3.3 Δυσκολίες των υπολογισμών 17
2 Περιγραφή του υπολογιστικού προγράμματος 19
2.1 Εισαγωγή 19
2.1.1 Γλώσσα προγραμματισμού 19
2.1.2 Δομή του προγράμματος 19
2.2 Βιβλιοθήκη PolyLib 21
2.2.1 Κλάση MultiVarTerm 22
2.2.2 Κλάση Polynomial 25
2.2.3 Διαγράμματα των κλάσεων 29
2.3 Braids: υπολογισμός του ίχνους 31
2.3.1 Βασικά στοιχεία του προγράμματος 31
2.3.2 Κλάση Word 35
2.3.3 Αλγόριθμος υπολογισμού ίχνους Ocneanu στις άλγεβρες Hecke 44
2.3.4 Αλγόριθμος υπολογισμού ίχνους Juyumaya στις άλγεβρες Yokonuma-Hecke 48
2.4 Αποτελέσματα του προγράμματος 52
2.5 Μελλοντική χρήση και επέκταση του προγράμματος 53
3 Κώδικας προγράμματος 54
3.1 PolyLib 54
3.1.1 MultiVarTerm.cs 54

3.1.2 Polynomial.cs 58
3.2 Braids 67
3.2.1 Parser.cs 67
3.2.2 Compute.cs 68
3.2.3 Comparer.cs 69
3.2.4 Word.cs 69
3.2.5 MainWindow.xaml 92
3.2.6 MainWindow.xaml.cs 93
4 Βιβλιογραφία 97

0 Εισαγωγή

Ο στόχος αυτής της διπλωματικής εργασίας είναι η ανάπτυξη ενός υπολογιστικού πακέτου για τον υπολογισμό του ίχνους Juyumaya [Ju] πάνω στις άλγεβρες Yokonuma-Hecke [Yo]. Αυτό το πρόγραμμα θα χρησιμοποιηθεί στον υπολογισμό αναλλοίωτων κλασικών κόμβων, πλαισιωμένων (framed) κόμβων, κόμβων singular και κόμβων transverse, που έχουν κατασκευαστεί μέσω των αλγεβρών Yokonuma-Hecke [JuLa1, JuLa2, JuLa3, JuLa4]. Επίσης θα χρησιμοποιηθεί για υπολογισμούς στις άλγεβρες Yokonuma-Temperley-Lieb και στις άλγεβρες Hecke τύπου Β. Πιο συγκεκριμένα:

Στο Κεφάλαιο 1 δίνεται μία εισαγωγή στη Θεωρία Κόμβων και στις ομάδες κοτσίδων. Οι ομάδες κοτσίδων δίνουν μία αλγεβρική δομή στους κόμβους μέσω των θεωρημάτων Alexander και Markov. Στη συνέχεια δίνονται οι ορισμοί και οι ιδιότητες δύο αλγεβρών που θα παίξουν τον πρωταρχικό ρόλο στην υλοποίηση του προγράμματος: πρόκειται για τις άλγεβρες Hecke τύπου Α και τις άλγεβρες Yokonuma-Hecke. Μέσω των αλγεβρών αυτών είναι δυνατόν να ορίσουμε πολυωνυμικές αναλλοίωτες κόμβων, για παράδειγμα το πολυώνυμο Jones [Jo]. Οι άλγεβρες Hecke τύπου Α σχετίζονται με τους κλασικούς κόμβους και οι άλγεβρες Yokonuma-Hecke σχετίζονται με τους πλαισιωμένους κόμβους. Και οι δύο άλγεβρες έχουν τους ίδιους γεννήτορες κοτσίδων, όμως η άλγεβρα Yokonuma-Hecke έχει επιπλέον γεννήτορες για το framing ενώ η τετραγωνική της σχέση είναι πολύ πιο πολύπλοκη από εκείνη της άλγεβρας Hecke. Ο υπολογισμός του ίχνους Ocneanu για τις άλγεβρες Hecke ήταν εφικτός λόγω της απλότητας της τετραγωνικής σχέσης της άλγεβρας, όμως το ίχνος Juyumaya στις άλγεβρες Yokonuma-Hecke έχει μεγάλη υπολογιστική πολυπλοκότητα, πράγμα που κάνει αναγκαία την ανάπτυξη του τρέχοντος υπολογιστικού πακέτου. Το υπολογιστικό πακέτο θα υπολογίζει τα ίχνη των λέξεων των δύο παραπάνω αλγεβρών.

Το Κεφάλαιο 2 αποτελεί το κύριο μέρος αυτής της εργασίας. Πρόκειται για την παρουσίαση του προγράμματος που υλοποιήθηκε με στόχο να υπολογιστεί το ίχνος για τις άλγεβρες Yokonuma-Hecke. Περιγράφεται αναλυτικά η δομή του προγράμματος καθώς και τα προβλήματα που συναντήθηκαν κατά την ανάπτυξή του. Πρώτα αναπτύξαμε το πρόγραμμα για τις άλγεβρες Hecke τύπου Α προκειμένου να ανάγουμε πολύπλοκες λέξεις σε γραμμικούς συνδυασμούς λέξεων μικρότερου μήκους που, ταυτόχρονα, να συμφωνούν με τους κανόνες τους ίχνους Ocneanu. Στη συνέχεια το πρόγραμμα προσαρμόστηκε στις άλγεβρες Yokonuma-Hecke προσθέτοντας τους γεννήτορες framing και τις σχέσεις τους με τους γεννήτορες κοτσίδων ενώ ταυτόχρονα αντικαταστάθηκε η τετραγωνική σχέση Hecke από την τετραγωνική σχέση Yokonuma-Hecke. Το ίχνος Ocneanu οδηγεί σε πολυώνυμα Laurent 2 μεταβλητών ενώ το ίχνος Juyumaya οδηγεί σε πολυώνυμα Laurent $d+1$ μεταβλητών. Για αυτό δημιουργήσαμε μία προγραμματιστική βιβλιοθήκη για την αναπαράσταση στον υπολογιστή

πολυωνύμων πολλών μεταβλητών. Το Κεφάλαιο 2 ακολουθεί τη δομή του προγράμματος και παράλληλα εξηγείται το πώς υλοποιεί την εκάστοτε συνάρτηση ίχνους. Στο τέλος του κεφαλαίου περιγράφονται αναλυτικά οι αλγόριθμοι υπολογισμού του ίχνους στις άλγεβρες Hecke και στις άλγεβρες Yokonuma-Hecke. Στα πλαίσια της δημιουργίας του αλγορίθμου δημιουργήθηκε και ένα απλό διαδραστικό περιβάλλον για τη χρήση του προγράμματος. Στη συνέχεια, παραθέτουμε παραδείγματα υπολογισμών σε γνωστούς ισοτοπικούς κόμβους. Τέλος, στο Κεφάλαιο 3 παρατίθεται ο κώδικας του προγράμματος. Τα αποτελέσματα της εργασίας αυτής είναι πρωτότυπα.

1 Κόμβοι και κοτσίδες

1.1 Κόμβοι

Η Θεωρία Κόμβων είναι ένας κλάδος της Τοπολογίας χαμηλών διαστάσεων. Στην Τοπολογία δεν εξετάζονται οι γεωμετρικές ιδιότητες των αντικειμένων, αλλά οι ιδιότητες οι οποίες παραμένουν αναλλοίωτες ως προς την αλλαγή καμπυλότητας, στρέψης και γενικά ως προς ελαστικές παραμορφώσεις. Για παράδειγμα μία σφαίρα και ένας κύβος θεωρούνται τοπολογικά όμοια αντικείμενα. Πιο συγκεκριμένα, η σχέση ισοδυναμίας στην Τοπολογία είναι ο ομοιομορφισμός. Δύο τοπολογικοί χώροι λέγονται ομοιομορφικοί αν υπάρχει μεταξύ τους μία αμφιμονοσήμαντη και συνεχής απεικόνιση με αντίστροφη επίσης συνεχή.

Ένας *κόμβος (knot)* είναι η ομοιομορφική εικόνα του κύκλου στον τρισδιάστατο χώρο. Διαισθητικά ένας κόμβος είναι απλά μία κλειστή, μονοδιάστατη, συνεχής μη τεμνόμενη καμπύλη στον τρισδιάστατο χώρο.

Τετριμμένος κόμβος (unknot)

Trefoil

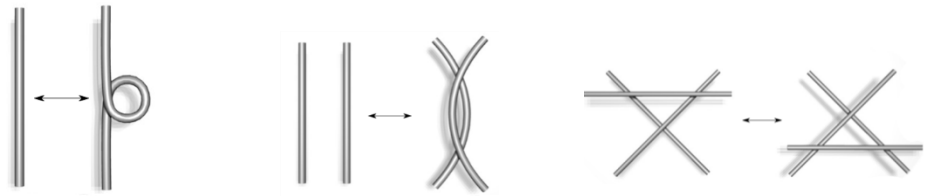
8_1

Ένας *κρίκος (link)* με k συνιστώσες είναι η ομοιομορφική εικόνα k κύκλων. Προφανώς ένας κόμβος είναι ένας κρίκος με 1 συνιστώσα. Στο εξής θα λέμε κόμβοι εννοώντας κόμβοι και κρίκοι. Το σημαντικότερο ανοικτό πρόβλημα στη Θεωρία Κόμβων είναι η ταξινόμηση των κόμβων ως προς την σχέση της *ισοτοπίας*, δηλαδή η καταγραφή όλων των δυνατών τύπων κόμβων. Δύο κόμβοι K_1 και K_2 λέγονται ισοτοπικοί αν υπάρχει ομοιομορφισμός του ζεύγους (S^3, K_1) στο ζεύγος (S^3, K_2). Η ταξινόμηση των κόμβων είναι ένα δύσκολο πρόβλημα καθώς υπάρχουν κόμβοι οι οποίοι μπορεί να φαίνονται εξαιρετικά πολύπλοκοι αλλά τελικά να είναι ισοτοπικοί με κάποιον ήδη γνωστό κόμβο. Για παράδειγμα, ο παρακάτω κόμβος μπορεί να παραμορφωθεί ελαστικά χωρίς να τον «κόψουμε» στον τετριμμένο κόμβο.

Συνήθως τους κόμβους τους αναπαριστούμε με *διαγράμματα*. Τα διαγράμματα αυτά αποτελούν προβολές των κόμβων στο επίπεδο σύμφωνα με τους κανόνες:

i. Δεν επιτρέπονται σημεία αναστροφής (cusps).
ii. Δεν επιτρέπονται εφαπτομενικά σημεία.
iii. Δεν επιτρέπονται τριπλά (ή περισσότερα) σημεία προβολής.
iv. Το σύνολο των *σημείων διασταύρωσης (crossings)* πρέπει να είναι πεπερασμένο.

Αποδεικνύεται ότι δύο κόμβοι είναι ισοτοπικοί αν και μόνο αν ο ένας παράγεται από τον άλλον από μία ακολουθία *κινήσεων Reidemeister*. Το Θεώρημα Reidemeister κάνει το πρόβλημα της ταξινόμησης κόμβων διακριτό.



Κίνηση RI (twist / untwist) *Κίνηση RII (poke / unpoke)* *Κίνηση RIII (slide)*

Η συνήθης μέθοδος για να διακρίνουμε κόμβους είναι η κατασκευή *αναλλοίωτων κόμβων*. Μία αναλλοίωτη κόμβων είναι μία συνάρτηση από το σύνολο των κόμβων σε ένα αλγεβρικό αντικείμενο, του οποίου η τιμή εξαρτάται μόνο από την κλάση ισοτοπίας στην οποία ανήκει ο κόμβος. Συνήθως ως σύνολο τιμών χρησιμοποιούνται είτε σύνολα αριθμών είτε πολυώνυμα, εφόσον είναι εύκολο να κάνουμε πράξεις και με τα δύο. Έχοντας κατασκευάσει μία αναλλοίωτη κόμβων, είναι εύκολο να αποφανθούμε αν δύο κόμβοι δεν είναι ισοτοπικοί: αν οι εικόνες τους μέσω της αναλλοίωτης διαφέρουν, τότε δεν είναι ισοτοπικοί. Οι αναλλοίωτες κόμβων μπορούν να μας δείξουν μόνο αν δύο κόμβοι *δεν* είναι ισοτοπικοί. Δηλαδή αν δύο κόμβοι έχουν την ίδια τιμή μέσω της αναλλοίωτης, αυτό δεν σημαίνει ότι είναι ισοτοπικοί. Τέλος, συνήθως για τις αναλλοίωτες κόμβων υπάρχει κάποιος αλγόριθμος ο οποίος υπολογίζει την τιμή τους από το διάγραμμα ενός κόμβου.

1.2 Ομάδες κοτσίδων (braid group)

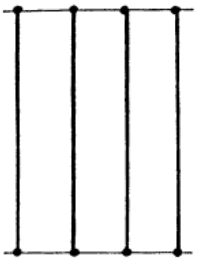
1.2.1 Απλές κοτσίδες

Το πρόβλημα με τους κόμβους είναι ότι δεν μπορούμε να τους μελετήσουμε άμεσα με αλγεβρικά μέσα. Δηλαδή δεν μπορούμε να πούμε ότι το σύνολο των κόμβων είναι ομάδα για παράδειγμα. Όμως αυτό το πρόβλημα λύνεται ορίζοντας τις *ομάδες των κοτσίδων*, τις οποίες μπορούμε να μελετήσουμε αλγεβρικά.

Γεωμετρικός ορισμός (Artin, 1923): Μία *κοτσίδα (braid) με* n *κλωστές (strands)* ορίζεται ως ένα σύνολο μη τεμνόμενων αυξουσών πολυγωνικών κλωστών στον $\mathbb{R}^3$ που συνδέουν σημεία $A_1, A_2, \dots, A_n$ με σημεία $B_1, B_2, \dots, B_n$, όπου $A_i = (i, 0, 0)$ και $B_i = (i, 0, 1)$. Λέμε μία πολυγωνική κλωστή *αύξουσα* όταν, θεωρώντας ένα σημείο που κινείται πάνω στην κλωστή ξεκινώντας από ένα A_i καταλήγει σε ένα B_j, η συντεταγμένη z του σημείου αυξάνει μονότονα.

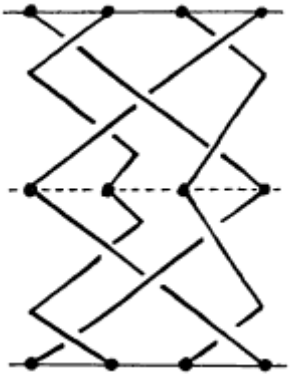
Θεωρώντας ότι οι κλωστές μιας κοτσίδας είναι ελαστικές και ότι μπορούμε να τις μετακινήσουμε (χωρίς αυτοτομές), τότε μπορούμε να κατατάξουμε τις κοτσίδες σε κλάσεις ισοδυναμίας. Από εδώ και στο εξής θεωρούμε ότι μία κοτσίδα εκπροσωπεί όλη την κλάση ισοδυναμίας της, όπως προκύπτει με αυτόν τον διαισθητικό τρόπο.

Στο σύνολο των κοτσίδων με n κλωστές μπορούμε να ορίσουμε μία πράξη: θεωρούμε ως γινόμενο δύο κοτσίδων την κοτσίδα που προκύπτει αν «ενώσουμε» τις δύο κοτσίδες βάζοντας τη μία κάτω από την άλλη. Το γινόμενο δύο κοτσίδων είναι τότε μία προσεταιριστική πράξη, υπάρχει ταυτοτική κοτσίδα η οποία αντιστοιχεί στο κάθε σημείο A_i το σημείο B_i, όπως φαίνεται στο παρακάτω σχήμα.



Η ταυτοτική κοτσίδα με 4 κλωστές

Επίσης υπάρχει και η αντίστροφη μιας κοτσίδας. Δηλαδή η κοτσίδα η οποία «ξεκινάει» από τα σημεία B_i και καταλήγει στα σημεία A_i της αρχικής κοτσίδας και είναι κατοπτρική ως προς το ενδιάμεσο επίπεδο που προκύπτει αν πάρουμε το γινόμενό τους, βλ. παρακάτω σχήμα.



Μία κοτσίδα με την αντίστροφή της

Σύμφωνα με τα παραπάνω, οι κοτσίδες έχουν μία δομή ομάδας θεωρώντας ως διμελή πράξη της ομάδας το γινόμενο κοτσίδων. Φυσικά η ομάδα των κοτσίδων δεν είναι αβελιανή. Αυτό ήταν ένα αναμενόμενο αποτέλεσμα αφού μία κοτσίδα επάγει φυσικά μια μετάθεση, και από τη στιγμή που η ομάδα των μεταθέσεων S_n του συνόλου $\{1, 2, \dots, n\}$ δεν είναι αντιμεταθετική, τότε δεν θα είναι και η ομάδα των κοτσίδων με n κλωστές.

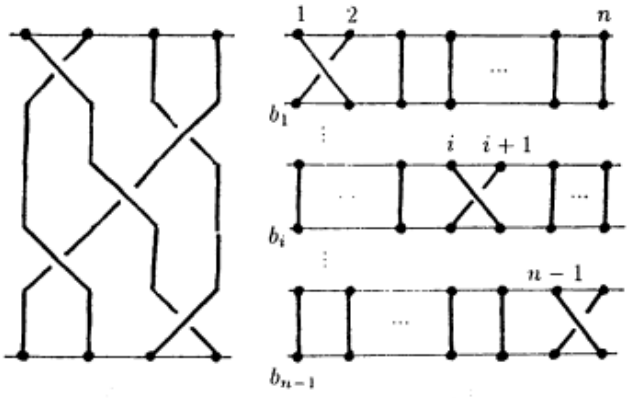
Όμως υπάρχει μία παράσταση της ομάδας των κοτσίδων, μέσω της οποίας η ομάδα των κοτσίδων ορίζεται αλγεβρικά.

Αλγεβρικός ορισμός (Artin, 1923): Ορίζουμε ως *ομάδα κοτσίδων Artin (Artin braid group)* μία ομάδα με γεννήτορες αντίστοιχους μιας ομάδας Coxeter και ίδιες σχέσεις εκτός της σχέσης $\sigma_i^2 = 1$. Σύμφωνα με αυτόν τον ορισμό η παράσταση της ομάδας B_n των κοτσίδων με n κλωστές τύπου Α είναι:

$$B_n = < \sigma_1, \dots, \sigma_n \mid \sigma_i\sigma_j = \sigma_j\sigma_i, |i-j| > 1; \sigma_i\sigma_j\sigma_i = \sigma_j\sigma_i\sigma_j, |i-j| = 1 >$$

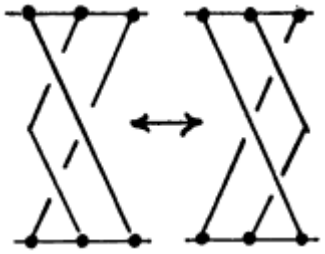
Οι σχέσεις που περιγράφουν την B_n ονομάζονται *braid relations*. Η πρώτη σχέση ονομάζεται *far commutativity* λόγω του ότι η αντιμεταθετική ιδιότητα δεν ισχύει για όλα τα ζεύγη κοτσίδων. Παρατηρούμε ότι η B_n παρίσταται με τις ίδιες σχέσεις που περιγράφουν την αντίστοιχη ομάδα μεταθέσεων S_n – η οποία είναι η ομάδα Coxeter τύπου Α – μόνο που λείπει η σχέση $\sigma_i^2 = 1$, όπως ακριβώς περιγράφει ο ορισμός.

Οι γεννήτορες σ_i ορίζονται με φυσικό τρόπο από τους γεννήτορες των ομάδων μεταθέσεων: με σ_i περιγράφεται η κοτσίδα η οποία αντιστοιχεί τον αριθμό i στον αριθμό $i+1$, τον αριθμό $i+1$ στον αριθμό i και αφήνει όλους τους άλλους αριθμούς ίδιους. Διαγραμματικά οι γεννήτορες απεικονίζονται στο παρακάτω σχήμα:

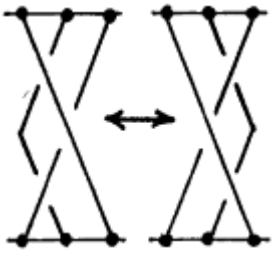


Μία κοτσίδα και οι γεννήτορες της

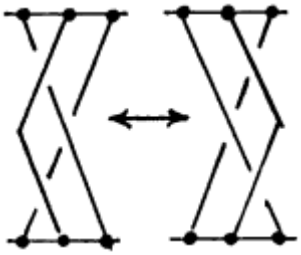
Ο Chow απέδειξε το 1925 ότι οι δύο ορισμοί ταυτίζονται. Οπότε μπορούμε να περιγράψουμε αλγεβρικά τις γεωμετρικές σχέσεις μέσω των οποίων μπορούμε να οδηγηθούμε από μία κοτσίδα σε μία *ισοτοπική* της, δηλαδή σε μία κοτσίδα η οποία θα προκύπτει από την πρώτη μέσω μίας ακολουθίας κινήσεων (όμοιες με τις κινήσεις Reidemeister για τους κόμβους). Οι δυνατές αυτές κινήσεις και οι αλγεβρικές σχέσεις που τις περιγράφουν είναι οι εξής:



$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$



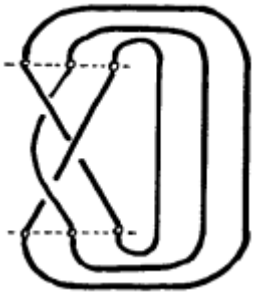
$\sigma_i\sigma_{i+1}\sigma_i^{-1} = \sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}$



$\sigma_i^{-1}\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}^{-1}$

Αυτές οι τρεις κινήσεις είναι όλες οι βασικές ισοτοπίες σε επίπεδο διαγραμμάτων για κοτσίδες.

Οι κοτσίδες παίζουν σημαντικό ρόλο στη Θεωρία Κόμβων καθώς αν «κλείσουμε» μία κοτσίδα, δηλαδή αν ενώσουμε τα άκρα της με απλά τόξα, τότε προκύπτει ένας προσαναολισμένος κρίκος.



Το κλείσιμο μίας κοτσίδας

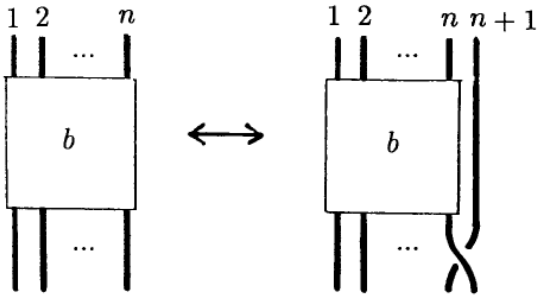
Αντίστροφα, το θεώρημα Alexander πρακτικά μας λέει ότι σε κάθε κόμβο μπορούμε να αντιστοιχίσουμε μία κοτσίδα:

Θεώρημα (Alexander, 1923): Κάθε προσανατολισμένος κρίκος είναι ισοτοπικός με το κλείσιμο μίας κοτσίδας.

Από τη στιγμή που σε κάθε κόμβο έχουμε αντιστοιχίσει μία κοτσίδα, είναι φυσικό να ερωτήσουμε πως σχετίζονται δύο κοτσίδες που αντιστοιχούν σε ισοτοπικούς κόμβους. Σε αυτό μας βοηθούν οι δύο παρακάτω κινήσεις:

1. Συζυγία: $b \leftrightarrow aba^{-1}$ όπου $a, b \in B_n$.
2. Κίνηση Markov: $b \leftrightarrow b\sigma_n^{\pm 1}$.

Η συζυγία μας δείχνει ότι οι κλειστές κοτσίδες ab και ba είναι ισοτοπικές. Η κίνηση Markov αντιστοιχεί πρακτικά μία κοτσίδα της ομάδας B_n σε μία κοτσίδα της ομάδας B_{n+1} προσθέτοντας μία κλωστή η οποία διασταυρώνεται με την τελευταία, όπως φαίνεται στο παρακάτω σχήμα.



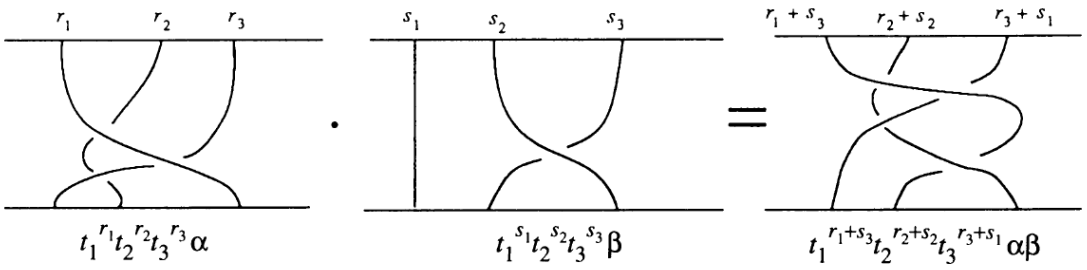
Οι παραπάνω δύο κινήσεις κοτσίδων αποδεικνύεται ότι είναι αρκετές για να βρούμε αν τα κλεισίματα δύο κοτσίδων αντιστοιχούν σε ισοτοπικούς κόμβους:

Θεώρημα (Markov, 1935): Τα κλεισίματα δύο κοτσίδων είναι ισοτοπικά αν και μόνον αν μία κοτσίδα προέρχεται από την άλλη από μία πεπερασμένη ακολουθία κινήσεων 1 και 2.

Από τα θεωρήματα Alexander και Markov αποδεικνύεται ότι πολλά προβλήματα της Θεωρίας Κόμβων μπορούν να προσεγγιστούν χρησιμοποιώντας τις ομάδες των κοτσίδων, με τις οποίες μπορούμε να δουλέψουμε αλγεβρικά.

1.2.2 Πλαισιωμένες κοτσίδες

Στη συνέχεια θα ορίσουμε τις *πλαισιωμένες κοτσίδες (framed braids)*. Διαισθητικά, μία πλαισιωμένη κοτσίδα είναι μια κοτσίδα με n κλωστές, όπου σε κάθε κλωστή επισυνάπτουμε έναν ακέραιο αριθμό. Επιπλέον, όταν συνθέτουμε δύο πλαισιωμένες κοτσίδες, οι αριθμοί που έχουμε επισυνάψει σε κάθε κλωστή προστίθενται σύμφωνα με τη μετάθεση που επάγεται.



Η ομάδα των πλαισιωμένων κοτσίδων $\mathcal{F}_n$ ορίζεται ως ένα ημιευθύ γινόμενο ομάδων.

Ορισμός: Έστω G μία ομάδα, H μία υποομάδα της G και N μία κανονική υποομάδα της G. Αν κάθε στοιχείο της G γράφεται μοναδικά ως ένα γινόμενο ενός στοιχείου της H και ενός στοιχείου της N τότε λέμε ότι η G είναι το *ημιευθύ γινόμενο* των ομάδων H και N και συμβολίζουμε $G = N \rtimes H$.

Ορισμός: Η ομάδα των πλαισιωμένων κοτσίδων $\mathcal{F}_n$ ορίζεται ως το ημιευθύ γινόμενο της ομάδας $\mathbb{Z}^n$ με την ομάδα των κοτσίδων, δηλαδή $\mathcal{F}_n = \mathbb{Z}^n \rtimes B_n$.

Στους παραπάνω ορισμούς οι πλαισιωμένες κοτσίδες που δημιουργούνται έχουν σε κάθε κλωστή από έναν ακέραιο αριθμό. Όμως είναι δυνατόν να ορίσουμε πλαισιωμένες κοτσίδες στις οποίες να επισυνάψουμε κάποιο στοιχείο μίας ομάδας υπολοίπου $\mathbb{Z}_d$. Τότε συμβολίζουμε την ομάδα των πλαισιωμένων κοτσίδων modulo d που προκύπτει ως $\mathcal{F}_{d,n} = \left(\frac{\mathbb{Z}}{d\mathbb{Z}}\right)^n \rtimes B_n$. Η $\mathcal{F}_{d,n}$ μπορεί να οριστεί ως η ομάδα πηλίκο $\frac{\mathcal{F}_n}{<t_i^d=1>}$, όπου t_i ο γεννήτορας framing για την i κλωστή.

Υπάρχει παράσταση και για την ομάδα των πλαισιωμένων κοτσίδων. Η διαφορά τώρα θα είναι ότι θα έχουμε δύο σύνολα γεννητόρων. Το πρώτο σύνολο γεννητόρων θα είναι το σύνολο $\{\sigma_1, \dots, \sigma_n\}$ (οι γεννήτορες της B_n). Το δεύτερο σύνολο γεννητόρων θα είναι το σύνολο $\{t_1, \dots, t_n\}$ (framings) όπου t_i είναι η ταυτοτική κοτσίδα στην οποία επισυνάπτουμε στην i-κλωστή το 1 της $\mathbb{Z}_d$ και σε όλες τις άλλες το 0. Τότε, κάθε πλαισιωμένη κοτσίδα μπορεί να γραφεί ως ένα γινόμενο από αυτούς τους γεννήτορες.

Στην ομάδα $\mathcal{F}_{d,n}$ ισχύουν οι επιπλέον σχέσεις:

i. $t_i t_j = t_j t_i, \forall\, i,j$
ii. $t_j \sigma_i = \sigma_i t_{s_i(j)}, \forall\, i,j$
iii. $t_j^d = 1, \forall j$

Όπου $s_i(j)$ είναι το αποτέλεσμα της αντιμετάθεσης $s_i = (i, i+1)$ στο j.

Κλείνοντας μία πλαισιωμένη κοτσίδα λαμβάνουμε έναν *πλαισιωμένο κρίκο*. Δηλαδή έναν κρίκο όπου σε κάθε συνιστώσα επισυνάπτεται ένας ακέραιος.

Είναι γνωστό ότι κάθε τρισδιάστατη πολλαπλότητα (συμπαγής, χωρίς σύνορο, συνεκτική και προσανατολίσιμη) μπορεί να κατασκευαστεί από την S^3 μέσω «χειρουργικής» κατά μήκος ενός πλαισιωμένου κρίκου.

1.3 Άλγεβρες Hecke και Yokonuma-Hecke

Οι άλγεβρες Hecke τύπου A (ή Iwahori-Hecke) όπως και οι άλγεβρες Yokonuma-Hecke παίζουν σημαντικό ρόλο στην μελέτη των κοτσίδων και των πλαισιωμένων κοτσίδων αντίστοιχα. Συγκεκριμένα, οι άλγεβρες Hecke αποτελούν αναπαράσταση των κλασσικών κοτσίδων και οι άλγεβρες Yokonuma-Hecke των πλαισιωμένων κοτσίδων. Σε αυτό την παράγραφο θα ορίσουμε τις δύο παραπάνω άλγεβρες, και θα δούμε και τις συναρτήσεις ίχνους που ορίζονται σε αυτές.

1.3.1 Άλγεβρες Hecke

Η *άλγεβρα Hecke τύπου A*, $H_n(q)$ αποτελείται από ένα σύνολο γεννητόρων $\{1, g_1, g_2, \dots, g_{n-1}\}$ και έχει την παρακάτω παράσταση:

$$H_n(q) = < g_1, g_2, \dots, , g_{n-1} |\ g_i^2 = (q-1)g_i + q;$$
$$g_i g_{i+1} g_i = g_{i+1} g_i g_{i+1}; g_i g_j = g_j g_i\ |i-j| > 1 >$$

όπου $q \in \mathbb{C}$, μία απροσδιόριστη.

Παρατηρούμε ότι οι σχέσεις αυτές είναι ίδιες με τις σχέσεις που διέπουν την ομάδα των κλασικών κοτσίδων καθώς και την ομάδα μεταθέσεων S_n. Η μόνη διαφορά τώρα από την S_n είναι η τετραγωνική σχέση που είναι πιο σύνθετη (στην S_n θα είχαμε $\sigma_i^2 = 1$). Για $q = 1$ έχουμε ότι η άλγεβρα $H_n(q)$ δεν είναι παρά η group algebra $\mathbb{C}S_n$. Επίσης η διάσταση της $H_n(q)$ είναι $n!$.

Στην $H_n(q)$ μπορούμε να γενικεύσουμε την τετραγωνική σχέση για να έχουμε κλειστό τύπο για κάθε εκθέτη $n \in \mathbb{N}$. Για $n = 3$ έχουμε:

$$\begin{aligned} g_i^3 = g_i^2 g_i = [(q-1)g_i + q]g_i = (q-1)g_i^2 + qg_i \\ = (q-1)[(q-1)g_i + q] + qg_i = (q-1)^2 g_i + q(q-1) + qg_i \\ = (q^2 - 2q + 1)g_i + q^2 - q + qg_i = (q^2 - q + 1)g_i + (q^2 - q) \end{aligned}$$

Επαγωγικά έχουμε την εξής σχέση για θετικούς εκθέτες:

$$g_i^n = (q^n - q^{n-1} + \dots - (-1)^{n+1})g_i + (q^n - q^{n-1} + \dots - q(-1)^n)$$

Μπορούμε όμως να βρούμε και κλειστή σχέση για αρνητικούς εκθέτες. Για τον αντίστροφο ενός γεννήτορα έχουμε:

$$g_i^2 = (q-1)g_i + q \Leftrightarrow g_i^2 g_i^{-1} = (q-1) + qg_i^{-1} \Leftrightarrow$$

$$g_i^{-1} = q^{-1}g_i + (q^{-1} - 1)$$

Η παραπάνω σχέση γενικεύεται και για οποιονδήποτε αρνητικό εκθέτη ($n < 0$) στην σχέση:

$$g_i^n = (q^n - q^{n+1} + \dots - q(-1)^{n+1})g_i + (q^n - q^{n-1} + \dots - (-1)^n)$$

Το συμπέρασμα είναι ότι για οποιονδήποτε εκθέτη μία λέξη αναλύεται σε δύο απλούστερες. Η μόνη διαφορά είναι ότι οι πολυωνυμικοί συντελεστές διαφέρουν ανάλογα με τον εκθέτη.

Θεώρημα Ocneanu, 1984 (ίχνος Markov για την $H_n(q)$): Για κάθε n υπάρχει μία μοναδική γραμμική συνάρτηση ίχνους στην $H_n(q)$ που καθορίζεται από τους κανόνες (όπου $z \in \mathbb{C}$ απροσδιόριστη):

1. $tr(ab) = tr(ba)$
2. $tr(1) = 1$
3. $tr(ag_n) = z\, tr(a),\ a \in H_n(q)$

Αξίζει να σημειωθεί ότι για να εφαρμοστεί ο κανόνας (3) θα πρέπει να περιέχει ακριβώς μία φορά τον γεννήτορα g_n. Οπότε με το ίχνος Ocneanu κάθε λέξη της άλγεβρας Hecke αναπαρίσταται από ένα μιγαδικό πολυώνυμο Laurent δύο μεταβλητών $q,\ z$. Θα δούμε ότι το ίχνος Ocneanu θα λειτουργήσει ως αναλλοίωτη για τους προσανατολισμένους κόμβους με τη βοήθεια του πολυωνύμου Jones:

Ας θεωρήσουμε την group algebra $\mathbb{C}B_n$. Η άλγεβρα αυτή θα περιέχει αθροίσματα όπου οι όροι θα είναι κοτσίδες με έναν μιγαδικό συντελεστή. Τότε, μπορούμε να αντιστοιχίσουμε την άλγεβρα $\mathbb{C}B_n$ στην άλγεβρα Hecke $H_n(q)$ αντιστοιχώντας κάθε γεννήτορα σ_i της $\mathbb{C}B_n$ στον γεννήτορα g_i της $H_n(q)$. Οπότε η αντιστοιχία είναι καλά ορισμένη και αποδεικνύεται εύκολα ότι είναι επιμορφισμός. Τονίζουμε ότι η διάσταση της $\mathbb{C}B_n$ είναι άπειρη ενώ η διάσταση της $H_n(q)$ είναι πεπερασμένη.

Οπότε η διαδικασία για την κατασκευή του πολυωνύμου Jones συνοψίζεται στα εξής: ξεκινώντας από έναν προσανατολισμένο κόμβο, από το θεώρημα Alexander υπάρχει κοτσίδα της οποίας το κλείσιμο μας δίνει αυτόν τον κόμβο. Η κοτσίδα αυτή όμως μπορεί να αναπαρασταθεί στην άλγεβρα Hecke $H_n(q)$. Στη συνέχεια υπολογίζουμε το ίχνος Ocneanu της λέξης που δημιουργείται και λαμβάνουμε ως αποτέλεσμα ένα μιγαδικό πολυώνυμο Laurent δύο μεταβλητών. Άρα υπολογίζοντας τα ίχνη για δύο διαφορετικούς προσανατολισμένους κόμβους, μπορούμε να αποφανθούμε αν δεν είναι ισοτοπικοί. Συγκεκριμένα, για έναν προσανατολισμένο κόμβο L έχουμε την εξής αναλλοίωτη - *πολυώνυμο Jones δύο μεταβλητών* (Jones, 1984):

$$X_L(q,\lambda) = \left(-\frac{1-\lambda q}{\sqrt{\lambda}\,(1-q)}\right)^{n-1} \left(\sqrt{\lambda}\right)^{e} tr(\pi(a))$$

Όπου $a \in B_n$ είναι μία οποιαδήποτε κοτσίδα της οποίας το κλείσιμο δίνει τον προσανατολισμένο κόμβο L και π είναι η αναπαράσταση της B_n στην $H_n(q)$. Η παραπάνω κατασκευή αναλλοίωτης κόμβων είναι η πρώτη που έγινε

χρησιμοποιώντας τη θεωρία των κοτσίδων και για αυτήν την κατασκευή ο Jones τιμήθηκε με το μετάλιο Fields.

Ο υπολογισμός του ίχνους της άλγεβρας Hecke είναι μία σχετικά εύκολη υπόθεση. Το μόνο πρόβλημα που υπάρχει είναι η τετραγωνική σχέση η οποία «σπάει» τη λέξη σε δύο κομμάτια. Αυτός είναι και ο λόγος για τον οποίο ο υπολογισμός αυτός γίνεται ταχύτερα από υπολογιστή.

1.3.2 Άλγεβρες Yokonuma-Hecke

Οι άλγεβρες Yokonuma-Hecke ορίστηκαν αρχικά από τον Yokonuma [Yo] και μπορούν να οριστούν ισοδύναμα ως άλγεβρες πηλίκα των modular πλαισιωμένων κοτσίδων $\mathbb{C}\mathcal{F}_{d,n}$ ως προς μία τετραγωνική σχέση, πολύ πιο πολύπλοκη από εκείνη των αλγεβρών Hecke.

Ορισμός: Η άλγεβρα Yokonuma-Hecke $Y_{d,n}(u)$ αποτελείται από ένα σύνολο γεννητόρων $\{1, g_1, g_2, \dots, g_{n-1}, t_1, t_2, \dots, t_n\}$ που ικανοποιούν τις εξής σχέσεις:

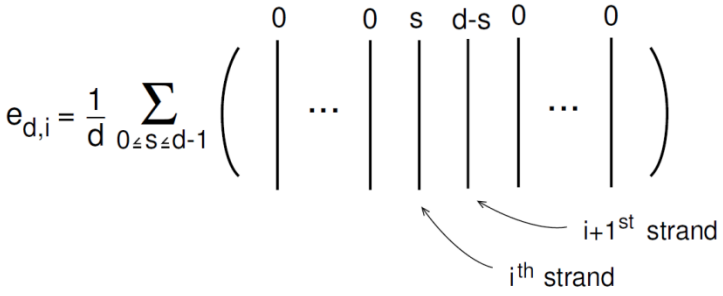
i. $g_i g_j = g_j g_i$ για $|i-j| > 1$
ii. $g_i g_j g_i = g_j g_i g_j$ για $|i-j| = 1$
iii. $t_i t_j = t_j t_i, \forall\, i, j$
iv. $t_j g_i = g_i t_{s_i(j)}, \forall\, i, j$
v. $t_j^d = 1, \forall j$

όπου $u \in \mathbb{C}$ μία απροσδιόριστη, και $s_i(j)$ είναι το αποτέλεσμα της αντιμετάθεσης $s_i = (i, i+1)$ στο j, μαζί με την τετραγωνική σχέση:

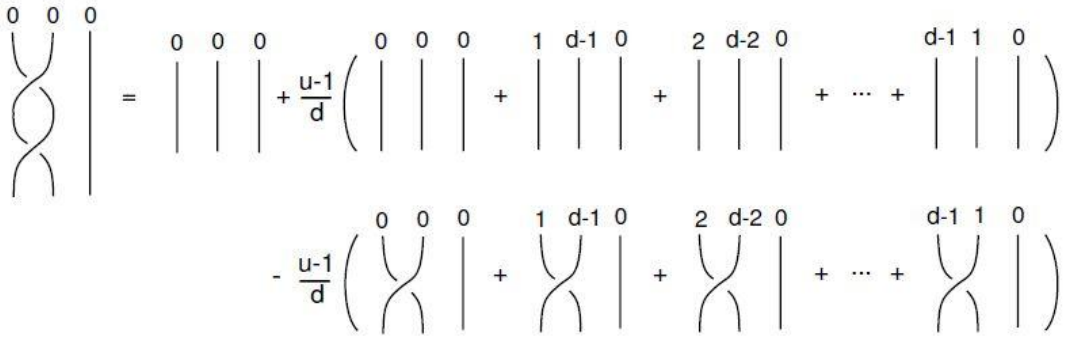
vi. $g_i^2 = 1 + (u-1)e_{d,i} - (u-1)e_{d,i}g_i, \ \forall i$, όπου

$$e_{d,i} = \frac{1}{d}\sum_{m=0}^{d-1} t_i^m t_{i+1}^{-m}$$

Διαγραμματικά το στοιχείο $e_{d,i}$ αναπαρίσταται ως εξής:



Οπότε η πολυπλοκότητα της τετραγωνικής σχέσης φαίνεται άμεσα από το εξής σχήμα, αντικαθιστώντας το $e_{d,i}$ σχηματικά:



Διαγραμματική αναπαράσταση του στοιχείου g_1^2

Από τις παραπάνω σχέσεις αποδεικνύονται εύκολα οι παρακάτω σχέσεις:

vii. $e_{d,i}^2 = e_{d,i}$
viii. $e_{d,i}e_{d,j} = e_{d,j}e_{d,i}$
ix. $e_{d,i}g_i = g_j e_{d,i}$ για $j = i$ και για $|i - j| > 1$
x. $e_{d,j}g_i g_j = g_i g_j e_{d,i}$ για $|i - j| = 1$

Όπως από την group algebra των απλών κοτσίδων $\mathbb{C}B_n$ υπάρχει επιμορφισμός στην άλγεβρα Hecke $H_n(q)$, με τον ίδιο τρόπο υπάρχει επιμορφισμός από την group algebra των πλαισιωμένων κοτσίδων $\mathbb{C}\mathcal{F}_n$ στην άλγεβρα Yokonuma-Hecke $Y_{d,n}(u)$. Συγκεκριμένα, απεικονίζουμε κάθε braiding γεννήτορα $\sigma_i \in \mathbb{C}B_n$ στον γεννήτορα $g_i \in Y_{d,n(u)}$, και κάθε framing γεννήτορα $t_i \in \mathbb{C}B_n$ στο στοιχείο $t_i \in Y_{d,n(u)}$. Γενικά παρατηρούμε ότι οι σχέσεις που περιγράφουν την άλγεβρα Yokonuma-Hecke είναι οι ίδιες που ισχύουν και για τις πλαισιωμένες κοτσίδες (εξαιρώντας την τετραγωνική σχέση).

Θεώρημα Juyumaya (Ίχνος Markov για την $Y_{d,n}(u)$): Έστω $z, x_1, x_2, \dots, x_{d-1} \in \mathbb{C}$ και έστω $d \in \mathbb{N}, d > 0$. Για κάθε $n \in \mathbb{N}$ υπάρχει μία μοναδική γραμμική απεικόνιση $tr_d = (tr_{d,n})_{n\in\mathbb{N}}$ $tr_d : Y_{d,n} \to \mathbb{C}$ η οποία καθορίζεται από τους κανόνες:

1. $tr_{d,n}(ab) = tr_{d,n}(ba)$
2. $tr_{d,n}(1) = 1$
3. $tr_{d,n+1}(ag_n) = z\, tr_{d,n}(a)$ όπου $a \in Y_{d,n}$
4. $tr_{d,n+1}(at_{n+1}^m) = x_m tr_{d,n}(a)$ όπου $a \in Y_{d,n}, 1 \le m \le d-1$

Αξίζει να σημειωθεί ότι για να εφαρμοστεί ο κανόνας (3) θα πρέπει να εμφανίζεται ακριβώς μία φορά ο γεννήτορας g_n και να μην περιέχεται στη λέξη ο γεννήτορας t_{n+1}. Αντίστοιχα για να εφαρμοστεί ο κανόνας (4) θα πρέπει να μην περιέχεται στη λέξη ο γεννήτορας g_n. Και εδώ, μέσω του ίχνους Juyumaya δύο διαφορετικά στοιχεία της άλγεβρας απεικονίζονται σε

δύο διαφορετικά μιγαδικά πολυώνυμα Laurent πολλών μεταβλητών. Λόγω της τετραγωνικής σχέσης, η οποία αναλύει την αρχική λέξη σε πολλές απλούστερες λέξεις (συγκεκριμένα $2d+1$ λέξεις κάθε φορά) είναι εμφανές ότι ο υπολογισμός του ίχνους για τις άλγεβρες Yokonuma-Hecke είναι αρκετά δύσκολος. Η τετραγωνική σχέση γενικεύεται για οποιοδήποτε ακέραιο εκθέτη ως εξής:

1. Για $m>0$, έστω $\alpha_m=(u-1)\sum_{l=0}^{k-1}u^{2l}$ αν $m=2k$, και $\beta_m=(u-1)\sum_{l=0}^{k-1}u^{2l}$ αν $m=2k+1$. Τότε
$$g_i^m=\begin{cases}1+a_m e_{d,i}-\alpha_m e_{d,i}g_i, & m=2k\\ g_i-\beta_m e_{d,i}+\beta_m e_{d,i}g_i, & m=2k+1\end{cases}$$

2. Για $m<0$, έστω $\alpha'_m=u^{-1}(u^{-1}-1)\sum_{l=0}^{k-1}u^{-2l}$ αν $m=-2k$, και $\beta'_m=(u^{-1}-1)\sum_{l=0}^{k-1}u^{-2l}$ αν $m=-2k+1$. Τότε
$$g_i^m=\begin{cases}1+a'_m e_{d,i}-\alpha'_m e_{d,i}g_i, & m=-2k\\ g_i-\beta'_m e_{d,i}+\beta'_m e_{d,i}g_i, & m=-2k+1\end{cases}$$

Η συνάρτηση ίχνους μπορεί θεωρητικά να μας βοηθήσει να κατασκευάσουμε μία αναλλοίωτη πλαισιωμένων κόμβων. Ξεκινώντας δηλαδή από έναν πλαισιωμένο κόμβο, θα βρούμε μία κοτσίδα που του αντιστοιχεί μέσω του θεωρήματος Alexander και αφού βρούμε την εικόνα της κοτσίδας στην άλγεβρα Yokonuma-Hecke θα μπορούμε να υπολογίσουμε το ίχνος, που θα είναι ένα μιγαδικό πολυώνυμο Laurent πολλών μεταβλητών.

Παραδόξως το ίχνος Juyumaya είναι το μόνο στην βιβλιογραφία που δεν κανονικοποιείται άμεσα σύμφωνα με την θετική και αρνητική κίνηση Markov.

1.3.3 Δυσκολίες των υπολογισμών

Οι υπολογισμοί της συνάρτησης ίχνους στις άλγεβρες Yokonuma-Hecke είναι αρκετά δύσκολοι λόγω της περίπλοκης τετραγωνικής σχέσης που περιέχει το στοιχείο $e_{d,i}$.

Η συνήθης πρακτική στους υπολογισμούς του ίχνους είναι να χωρίζουμε τη λέξη σε δύο μέρη: στο *framing part* και στο *braiding part*, φέρνοντας στην αρχή της λέξης – μέσω των σχέσεων (iv) του ορισμού – τα στοιχεία t_i. Για παράδειγμα έχουμε: $g_1t_2g_2^2t_5=t_1t_5g_1g_2^2$. Αυτή η μορφή μας διευκολύνει αρκετά στους υπολογισμούς, και είναι η τεχνική που θα χρησιμοποιηθεί από το υπολογιστικό πρόγραμμα.

Πράγματι, τα στοιχεία $e_{d,i}$ δεν έχουν, δυστυχώς, την ίδια συμπεριφορά με τα t_i, δηλαδή δεν μπορούμε να τα μεταφέρουμε στην αρχή της λέξης, λόγω των σχέσεων (viii) και (ix). Οπότε δεν μπορούμε να εφαρμόσουμε μία ανάλογη τεχνική διαχωρισμού και για τα $e_{d,i}$ έτσι ώστε να γίνεται γρήγορα ο υπολογισμός. Αυτός είναι και ο λόγος που είναι αναγκαίος ο υπολογιστής για τον υπολογισμό του ίχνους. Η μόνη λύση στο πρόβλημα είναι να

αντικαθιστούμε τα $e_{d,i}$ σύμφωνα με τη σχέση $e_{d,i} = \frac{1}{d}\sum_{m=0}^{d-1} t_i^m t_{i+1}^{-m}$, η οποία όμως αναλύει τη λέξη σε d καινούριες λέξεις. Για παράδειγμα, στη λέξη $g_3 g_2^{-2} g_3^{2a+2} g_2 g_3^{-1} g_1^{-1} g_2 g_1 g_1^{2b+2}$ για θετικά a και b, είναι εμφανές ότι για να υπολογιστεί το ίχνος θα πρέπει να εφαρμοστεί τουλάχιστον 5 φορές – για αρχή – η τετραγωνική σχέση, πράγμα που δυσχεραίνει πολύ τους υπολογισμούς.

Παρατηρούμε στην τετραγωνική σχέση ότι η αρχική μας κοτσίδα αναλύεται σε πολλές απλούστερες λέξεις. Στην καλύτερη περίπτωση (αν $d = 2$) η κοτσίδα θα σπάσει σε 7 καινούριες λέξεις. Όμως υπάρχει περίπτωση σε κάποιες από τις 7 νέες λέξεις να εφαρμοστεί πάλι η τετραγωνική σχέση που σημαίνει ότι θα αναλυθούν και πάλι. Η πολυπλοκότητα του προβλήματος αυξάνει αρκετά όσο μεγαλύτερο είναι το d και όσο περισσότερους εκθέτες έχει μία λέξη, έτσι ώστε είναι πρακτικά αδύνατον να υπολογιστούν τα ίχνη κάποιον λέξεων «στο χαρτί».

Συμπερασματικά, ξεκινώντας από μία πλαισιωμένη κοτσίδα αναλύουμε τους εκθέτες των γεννητόρων braiding όσες φορές χρειαστεί και σε κάθε λέξη που προκύπτει διαχωρίζουμε κάθε φορά τους γεννήτορες framing από τους γεννήτορες braiding. Στη συνέχεια επεξεργαζόμαστε το braiding part με τρόπο ανάλογο που κάνουμε με την άλγεβρα Hecke και συνεχίζουμε με όμοιο τρόπο.

Συνοπτικά, το υπολογιστικό πακέτο θα μπορεί να υπολογίζει και να συγκρίνει ίχνη για διάφορες λέξεις σχετικά γρήγορα, έτσι ώστε να μπορούμε να αποφανθούμε αν οι λέξεις αντιστοιχούν σε μη ισοτοπικούς κόμβους.

2 Περιγραφή του υπολογιστικού προγράμματος

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα περιγραφεί η δομή του υπολογιστικού προγράμματος που δημιουργήθηκε προκειμένου να υπολογιστούν τα ίχνη στις άλγεβρες Hecke και Yokonuma-Hecke.

2.1.1 Γλώσσα προγραμματισμού

Σύμφωνα με όσα αναφέρθηκαν στο $1^{ο}$ κεφάλαιο για τις δυσκολίες των υπολογισμών, είναι εμφανές ότι θα χρειαστεί να προσομοιωθούν ως αντικείμενα οι λέξεις στις άλγεβρες Hecke και Yokonuma-Hecke. Επίσης είναι υποχρεωτική η ύπαρξη ενός υποπρογράμματος το οποίο θα διαχειρίζεται πολυώνυμα πολλών μεταβλητών πάνω από το σώμα των πραγματικών αριθμών.

Οπότε για τους σκοπούς της ανάπτυξης του προγράμματος επιλέχθηκε η C# 4.0 ως γλώσσα προγραμματισμού. Η C# 4.0 είναι μία αντικειμενοστραφής γλώσσα προγραμματισμού, οπότε μπορούμε να ορίσουμε αντικείμενα πάνω στα οποία θα εφαρμόζονται συναρτήσεις. Αυτή είναι και η τεχνική που θα ακολουθηθεί στο πρόγραμμα. Συγκεκριμένα θα κατασκευαστεί – όπως θα δούμε αναλυτικά παρακάτω – μία κλάση η οποία θα προσομοιώνει μία λέξη σε κάποια άλγεβρα. Οι συναρτήσεις ίχνους θα είναι πρακτικά συναρτήσεις της κλάσης. Οπότε θα δημιουργούμε νέα αντικείμενα – λέξεις και θα καλούμε τη συνάρτηση ίχνους η οποία θα υπολογίζει το κατάλληλο πολυώνυμο.

Επίσης η C# 4.0 διαθέτει κάποια στοιχεία συναρτησιακού προγραμματισμού (ανώνυμες συναρτήσεις, λ-εκφράσεις και τη δυνατότητα δημιουργίας ερωτημάτων [queries] μέσα στη γλώσσα [LINQ]) τα οποία θα μας διευκολύνουν στις πράξεις με τις λίστες που θα χρησιμοποιηθούν εκτενώς. Η δυνατότητα χρήσης δομών δεδομένων μέσω του generics θα μας φανεί χρήσιμη έτσι ώστε να μην γίνονται άσκοπες μετατροπές τύπων.

Τέλος, η εγγενής υποστήριξη των μιγαδικών αριθμών (μέσω της κλάσης `System.Numerics.Complex`) είναι ένα σημαντικό χαρακτηριστικό που διευκόλυνε την ανάπτυξη του προγράμματος.

Η C# 4.0 τρέχει μόνο σε Windows, αν και υπάρχει ένα project ανοιχτού κώδικα (Mono) που επιτρέπει την εκτέλεση του προγράμματος σε Linux και Mac OS X με κάποιες μικρές τροποποιήσεις. Η ανάπτυξη του προγράμματος έγινε μέσω του προγραμματιστικού περιβάλλοντος Visual Studio 2010.

2.1.2 Δομή του προγράμματος

Το πρόγραμμα θα χωριστεί σε δύο μέρη: το πρώτο μέρος θα αναλάβει τις πράξεις μεταξύ πολυωνύμων και το δεύτερο μέρος τον υπολογισμό του ίχνους και το γραφικό περιβάλλον.

Συγκεκριμένα, το πρώτο μέρος το οποίο ονομάζεται `PolyLib` είναι μια βιβλιοθήκη για πολυώνυμα πολλών μεταβλητών. Η ανάπτυξη της βιβλιοθήκης αυτής είναι απολύτως ανεξάρτητη από το υπόλοιπο πρόγραμμα. Αυτό έχει ως αποτέλεσμα την πιο εύκολη αποσφαλμάτωση του κώδικα σε περίπτωση που εντοπιστούν λάθη. Επίσης λόγω του διαχωρισμού υπάρχει η δυνατότητα η βιβλιοθήκη αυτή να χρησιμοποιηθεί και σε άλλα project, χωρίς καμία απολύτως μετατροπή.

Η βιβλιοθήκη `PolyLib` είναι αρκετά ευέλικτη όσον αφορά τα πολυώνυμα. Έχει σχεδιαστεί έτσι ώστε να μπορούμε να προσομοιώσουμε οποιοδήποτε πολυώνυμο με πραγματικούς συντελεστές, ανεξάρτητα δηλαδή από τον αριθμό των μεταβλητών ή τον βαθμό του πολυωνύμου. Μέχρι τώρα, οι περισσότερες βιβλιοθήκες συνήθως έχουν γραφεί για πολυώνυμα 1, 2 ή 3 μεταβλητών, που είναι και το συνηθέστερο, ενώ η βιβλιοθήκη `PolyLib` που δημιουργήθηκε δεν έχει κανέναν περιορισμό.

Το δεύτερο μέρος του προγράμματος είναι το κύριο μέρος που ονομάζεται `Braids`. Σε αυτό το μέρος προσομοιώνουμε τις λέξεις για τις άλγεβρες Hecke και Yokonuma-Hecke και υπολογίζουμε τα ίχνη. Το μέρος αυτό χρησιμοποιεί εκτενώς την βιβλιοθήκη `PolyLib` ως ένα εξωτερικό πρόγραμμα χωρίς να παρεμβαίνει στη δομή και στη λειτουργία της `PolyLib`. Συγκεκριμένα το μέρος `Braids` φορτώνει την βιβλιοθήκη `PolyLib` ως ένα αρχείο DLL.

Η σχεδίαση του μέρους `Braids` είναι τέτοια ώστε να είναι δυνατή η επέκταση του προγράμματος και για μελλοντικούς υπολογισμούς χωρίς να επηρεάζεται στο ελάχιστο η εγκυρότητα των υπολογισμών που μπορούν να γίνουν μέχρι τώρα. Επίσης, διαθέτει και ένα στοιχειώδες γραφικό περιβάλλον.

2.2 Βιβλιοθήκη PolyLib

Όπως έχει αναφερθεί, η βιβλιοθήκη `PolyLib` είναι η βάση του προγράμματος `Braids` που θα υπολογίζει τα ίχνη.

Η πρώτη βασική ανάγκη που καλύπτει η βιβλιοθήκη `PolyLib` είναι η προσομοίωση πολυωνύμων πολλών μεταβλητών. Για να γίνει αυτό χρειάζεται προφανώς να αναπαρασταθεί το πολυώνυμο στον υπολογιστή, δηλαδή να δημιουργήσουμε μία δομή δεδομένων όπου θα αποθηκεύονται δεδομένα τα οποία θα περιγράφουν μοναδικά ένα πολυώνυμο. Η πιο κλασσική αναπαράσταση για ένα πολυώνυμο μίας μεταβλητής είναι ο πίνακας. Όμως για ένα πολυώνυμο με n μεταβλητές και βαθμό r δεν είναι βολικό να δημιουργηθεί ένας n-διάσταστος πίνακας (array) με μήκος m σε κάθε διάσταση, όπου σε κάθε θέση του πίνακα θα αποθηκεύεται ο κατάλληλος συνετλεστής, που είναι το προφανές, λόγω του ότι θα έχουμε δεδομένα τα οποία θα επαναλαμβάνονται (αφού για παράδειγμα ο συντελεστής του x^2y θα πρέπει να είναι ο ίδιος με τον συντελεστή του yx^2 αφού πρόκειται για τους ίδιους όρους) και λόγω του ότι θα υπάρχουν πολλά κελιά του πίνακα που θα περιέχουν 0, αφού συνήθως οι περισσότεροι όροι δεν υπάρχουν καν σε ένα πολυώνυμο.

Οπότε κρίνεται αναγκαία να δημιουργηθεί μία αναπαράσταση η οποία να μην επαναλαμβάνει δεδομένα, να μην περιέχει μηδενικούς συντελεστές και προφανώς να μην πιάνει τόσο χώρο στη μνήμη όσο ένας n-διάστατος πίνακας. Στην άλγεβρα ένα πολυώνυμο ορίζεται τυπικά ως το παρακάτω άπειρο άθροισμα:

$$p(x) = \sum_{i=0}^{\infty} a_i x^i$$

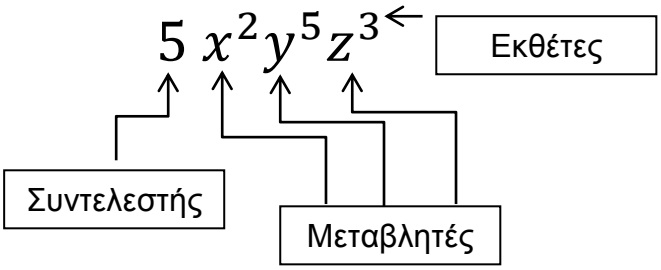
όπου οι συντελεστές a_n προέρχονται από έναν δακτύλιο (έτσι ώστε να ορίζονται οι πράξεις της πρόσθεσης και του πολλαπλασιασμού των πολυωνύμων μέσω των ανάλογων πράξεων του δακτυλίου). Για περισσότερες απροσδιόριστες έχουμε συνήθως τη μορφή:

$$p(x_1, \dots, x_n) = \sum_{i_n=0}^{\infty} \dots \sum_{i_1=0}^{\infty} a_{i_1} a_{i_2} \dots a_{i_n} x_1^{i_1} x_2^{i_2} \dots x_n^{i_n}$$

Σε κάθε περίπτωση τα πολυώνυμα θεωρούνται ως ένα άπειρο άθροισμα *όρων*. Οπότε θα μπορούσαμε να αναπαραστήσουμε ένα πολυώνυμο ως μία *λίστα όρων στην οποία δεν θα επιτρέπονται διπλότυπα (duplicates)*. Σε αυτή την περίπτωση στη λίστα δεν θα υπάρχουν επαναλαμβανόμενα δεδομένα, αφού κάθε συντελεστής ενός όρου θα υπάρχει μόνο μία φορά. Επίσης είναι εύκολο να απαγορέψουμε να εισάγονται όροι με μηδενικό συντελεστή, έτσι ώστε να γίνεται εξοικονόμηση χώρου. Οι πράξεις μεταξύ των πολυωνύμων θα

είναι εύκολο να οριστούν. Η πρόσθεση για παράδειγμα θα αποτελεί πρακτικά την προσθήκη των όρων του δεύτερου προσθετέου στον πρώτο προσθετέο.

Οπότε καθορίσαμε σίγουρα ότι ένα πολυώνυμο θα αναπαρίσταται από μία λίστα όρων. Όμως μένει να ορίσουμε την αναπαράσταση ενός όρου. Ένας πολυωνυμικός όρος έχει έναν συντελεστή που προέρχεται από τον δακτύλιο πάνω στον οποίο είναι ορισμένο το πολυώνυμο και διάφορες μεταβλητές ή απροσδιόριστες οι οποίες θα είναι υψωμένες σε κάποιον εκθέτη.



Όμως ο αριθμός των μεταβλητών που θα περιέχει κάθε όρος δεν είναι εκ των προτέρων γνωστός. Μπορεί κάλλιστα σε ένα πολυώνυμο να έχουμε τον όρο $3xy^2$ αλλά να υπάρχει και ο όρος $5d^5k^3x^3y^7z^8$. Το μόνο σίγουρο είναι ότι κάθε μεταβλητή έχει έναν εκθέτη (θα θεωρήσουμε προφανώς ότι αν δεν υπάρχει εκθέτης τότε αυτός είναι ο 1). Άρα για την αναπαράσταση ενός όρου θα χρειαστούμε μία μεταβλητή που να αποθηκεύει τον συντελεστή του όρου και έναν πίνακα που να αποθηκεύει την κάθε μεταβλητή μαζί με τον εκθέτη της. Όμως επειδή ο υπολογιστής θα πρέπει να καταλαβαίνει ότι οι όροι $5x^2y$ και $3yx^2$ είναι οι ίδιοι απλά με διαφορετικό συντελεστή, θα πρέπει σε κάθε όρο να ταξινομούνται οι μεταβλητές με αλφαβητική σειρά.

Οπότε μέχρι τώρα έχουμε ως δεδομένο ότι ένα πολυώνυμο θα είναι μία λίστα όρων όπου κάθε όρος έχει μία μεταβλητή για να αποθηκεύεται ο συντελεστής και μία δομή δεδομένων όπου θα αποθηκεύονται – κατά αλφαβητική σειρά – οι μεταβλητές μαζί με τον εκθέτη τους. Για αυτό τον λόγο δημιουργήθηκαν οι κλάσεις `MultiVarTerm` και `Polynomial`.

2.2.1 Κλάση MultiVarTerm

Η κλάση `MultiVarTerm` θα αναπαριστά έναν πολυωνυμικό όρο. Λόγω του ότι τα αντικείμενα αυτής της κλάσης θα ανήκουν σε ένα πολυώνυμο, θα χρειάζεται να ελέγχεται αν δύο όροι έχουν τις ίδιες μεταβλητές ή όχι έτσι ώστε να ορίζεται σωστά η πρόσθεση πολυωνύμων. Επίσης θα χρειαστεί να συγκρίνουμε τους όρους μεταξύ τους έτσι ώστε αργότερα να ταξινομηθούν οι όροι μέσα στο πολυώνυμο με αλφαβητική σειρά.

Πρέπει να αναφερθεί ότι χρησιμοποιείται κλάση αντί για struct λόγω των περισσότερων δυνατοτήτων που δίνει μία κλάση. Η struct είναι ένας value type, δηλαδή όταν δίνουμε $b = a$ τότε το αντικείμενο a αντιγράφεται στο b,

ενώ η κλάση είναι ένας reference type δηλαδή στην παραπάνω περίπτωση το b δεν θα ήταν αντίγραφο του a, αλλά απλά θα είναι μία αναφορά στο a. Οπότε αν αλλάζαμε την τιμή του b, σε περίπτωση που έχουμε struct τότε δεν θα αλλάξει η τιμή του a, ενώ αν έχουμε κλάση τότε θα αλλάξει. Όμως λόγω του ότι εμείς στο πρόγραμμα θα χρειαζόμαστε να δημιουργούνται αντικείμενα που δεν θα αναφέρονται το ένα στο άλλο αλλά και τις δυνατότητες μίας κλάσης, έχουν σχεδιασθεί μέθοδοι που αντιγράφουν τα δεδομένα και οι κατασκευαστές είναι κατάλληλα υλοποιημένοι.

2.2.1.1 Αναπαράσταση ενός όρου

Προφανώς για τον συντελεστή θα χρησιμοποιηθεί μία μεταβλητή τύπου `double`. Όσον αφορά τις μεταβλητές και τους εκθέτες επιλέχθηκε η δομή δεδομένων `SortedDictionary<char, int>`. Η `SortedDictionary<char, int>` δημιουργεί ένα ταξινομημένο λεξικό το οποίο θα περιέχει ένα ζεύγος που θα αποτελείται από μία μεταβλητή τύπου char (η μεταβλητή του πολυωνύμου) που ονομάζεται *κλειδί (key)* και από μία ακέραια μεταβλητή (ο εκθέτης της μεταβλητής του πολυωνύμου) που ονομάζεται *τιμή (value)*.

Η δομή αυτή δεν δέχεται δύο φορές τον ίδιο κλειδί, οπότε με αυτόν τον τρόπο αποτρέπεται το να έχουμε δύο φορές την ίδια μεταβλητή μέσα στον ίδιο όρο. Επίσης κρατά πάντα ταξινομημένες τις μεταβλητές με αλφαβητική σειρά χωρίς να χρειαστεί να γραφεί κάποιος κώδικας για αυτό.

2.2.1.2 Σύγκριση όρων

Η κλάση `MultiVarTerm` θα υλοποιεί τις διαπροσωπείες `IEquatable<MultiVarTerm>` και `IComparable<MultiVarTerm>`. Εδώ είναι και πρώτη χρήση του generics, όπου τα αντικείμενα θα συγκρίνονται άμεσα χωρίς να χρειάζεται η οποιαδήποτε μετατροπή τύπου δεδομένων.

Αρχικά θα ορίσουμε τυπικά μία διάταξη στους πολυωνυμικούς όρους. Έστω $a = c\, x_1^{i_1} \dots x_n^{i_n}$ ένας τυχαίος πολυωνυμικός όρος όπου οι όροι είναι με αλφαβητική σειρά. Ορίζουμε ως $f(a) = x_1 x_2 \dots x_n$ την συμβολοσειρά που προκύπτει να βάλουμε με αλφαβητική σειρά τους όρους τον ένα δίπλα στον άλλον και ως $g(a) = i_1 i_2 \dots i_n$ την συμβολοσειρά που προκύπτει αν βάλουμε τους εκθέτες των αντίστοιχων μεταβλητών στη σειρά. Για παράδειγμα αν $a = 8x^5y^3z\text{^} - 3$ τότε έχουμε $f(a) = "xyz"$ και $g(a) = "53 - 3"$ Τότε μπορούμε να ορίσουμε την εξής διάταξη για τους όρους:

$$a < b \Leftrightarrow \begin{cases} f(a) < f(b) \\ g(a) < g(b)\ \alpha\nu\ f(a) = f(b) \end{cases}$$

Η $f(a) < f(b)$ και η $g(a) < g(b)$ ορίζεται μέσω της γνωστής σύγκρισης συμβολοσειρών μέσω του κώδικα ASCII. Ο χαρακτήρας '-' που ενδέχεται να υπάρχει στις συμβολοσειρές των εκθετών δε θα μας δημιουργήσει πρόβλημα, αφού στον κώδικα ASCII το '-' είναι ο αριθμός 45 και ο χαρακτήρας '0' είναι ο

αριθμός 48 οπότε το '-' είναι μικρότερο από κάθε αριθμό όσον αφορά τις συμβολοσειρές. Αυτή η ταξινόμηση των όρων αντιστοιχεί στην *λεξικογραφική ταξινόμηση (lexicographical order)* των όρων ενός πολυωνύμου.

Όσων αφορά την απεικόνιση f που ορίσαμε, αυτή υλοποιείται από τη συνάρτηση `ToStringOnlyVar()` και η g από την `ToStringOnlyExp()`. Μέσω αυτών των δύο συναρτήσεων έχει υλοποιηθεί η `CompareTo(MultiVarTerm t)` της διαπροσωπείας `IComparable<MultiVarTerm>`. Τέλος η `Equals(MultiVarTerm t)` έχει υλοποιηθεί από την `CompareTo(MultiVarTerm t)` με προφανή τρόπο.

2.2.1.3 Κατασκευαστές

Στην κλάση αυτή έχουν οριστεί 4 κατασκευαστές κυρίως για ευκολία. Οι κατασκευαστές είναι οι εξής:

1. `MultiVarTerm(double coeff = 1)`: Δημιουργεί έναν νέο σταθερό όρο. Ο όρος προφανώς δεν περιέχει μεταβλητές και για συντελεστή έχει αυτόν που του δίνει ο χρήστης. Σε περίπτωση που δεν δώσει ο χρήστης κάποιον συντελεστή τότε αυτόματα θεωρείται ως συντελεστής η μονάδα.
2. `MultiVarTerm(SortedDictionary<char, int> var, double coeff = 1)`: Δημιουργεί έναν νέο όρο δοθέντων ενός ταξινομημένου λεξικού και ίσως ενός συντελεστή. Ο κατασκευαστής αυτός δουλεύει με προφανή τρόπο, αφού ο όρος τον οποίο μοντελοποιεί δίνεται στον κατασκευαστή «ετόιμος». Σε περίπτωση που δεν δοθεί ο συντελεστής, πάλι θεωρείται ως συντελεστής η μονάδα.
3. `MultiVarTerm(double coeff = 1, params Tuple<char, int>[] exp)`: Δημιουργεί έναν νέο όρο δοθέντων ενός πίνακα που έχει ζευγάρια (`Tuple<char, int>`) από μία μεταβλητή και τον εκθέτη της και ίσως ενός συντελεστή. Ο κατασκευαστής αυτός δημιουργεί ένα νέο ταξινομημένο λεξικό μέσα στο οποίο εισάγει τις τιμές που έρχονται από τον πίνακα. Σε περίπτωση που δεν δοθεί ο συντελεστής, πάλι θεωρείται ως συντελεστής η μονάδα.
4. `MultiVarTerm(double coeff, char c, int exp)`: Δημιουργεί έναν όρο μίας μεταβλητής. Ο χρήστης εδώ καλείται να δώσει υποχρεωτικά τον συντελεστή, τον χαρακτήρα της μεταβλητής και τον εκθέτη της μεταβλητής.

2.2.1.4 Πράξεις μεταξύ όρων

Είναι λογικό να δίνεται η δυνατότητα να πολλαπλασιάσουμε δύο όρους μεταξύ τους. Για αυτόν τον λόγο έχει οριστεί ο τελεστής * στην κλάση αυτή. Η πράξη της πρόσθεσης μεταξύ δύο όρων δεν έχει οριστεί στη συνηθέστερη περίπτωση δύο όροι όταν προστίθενται δεν δίνουν έναν νέο όρο αλλά ένα πολυώνυμο. Όμως δίνουμε τη δυνατότητα να πολλαπλασιάσουμε έναν όρο

και με έναν αριθμό. Οπότε ο πολλαπλασιασμός έχει οριστεί μέσω των παρακάτω μεθόδων:

1. MultiVarTerm operator *(MultiVarTerm t1, MultiVarTerm t2): Πολλαπλασιάζει δύο όρους μεταξύ τους, δημιουργώντας έναν νέο όρο. Απλά δημιουργούμε ένα νέο κενό αντικείμενο MultiVarTerm όπου βάζουμε μέσα – προσέχοντας τις περιπτώσεις όπου έχουμε μία μεταβλητή που υπάρχει και στους δύο όρους οπότε και προσθέτουμε τους εκθέτες – τις μεταβλητές και τους εκθέτες από κάθε όρο. Τέλος πολλαπλασιάζουμε και τους πραγματικούς συντελεστές των δύο όρων.
2. MultiVarTerm operator *(MultiVarTerm t, double c): Πολλαπλασιάζει έναν όρο με έναν αριθμό. Απλά δηλαδή πολλαπλασιάζει τον συντελεστή του όρου με τον αριθμό που δίνεται.
3. MultiVarTerm operator *(double c, MultiVarTerm t): Πολλαπλασιάζει πάλι έναν όρο με έναν αριθμό. Ορίζεται καταχρηστικά έτσι ώστε να είναι δυνατόν να γράφουμε $3 * x$ αλλά και $x * 3$ αν x είναι ένα αντικείμενο MultiVarTerm.

Τέλος έχει οριστεί και ο τελεστής - ο οποίος απλά επιστρέφει τον αντίθετο όρο (δηλαδή τον ίδιο όρο αλλά με αντίθετο συντελεστή).

2.2.1.5 Άλλες μέθοδοι

Υπάρχουν και κάποιες άλλες μέθοδοι οι οποίες χρησιμοποιούνται βοηθητικά κυρίως μέσα στο πρόγραμμα. Αυτές είναι οι εξής:

1. AddToCoeff(double c): Προσθέτει έναν αριθμό στον συντελεστή του τρέχοντα όρου. Η μέθοδος αυτή μας βοηθάει αρκετά από τη στιγμή που δεν έχει οριστεί η πράξη της πρόσθεσης για τους όρους.
2. string ToString(): Επιστρέφει μία συμβολοσειρά για τον τρέχοντα όρο. Για παράδειγμα ο όρος $5x^2y$ θα τυπωθεί ως «5 x^2 y».
3. ToStringOnlyVar(): Επιστρέφει μία συμβολοσειρά που περιέχει μόνο τις μεταβλητές του όρου.
4. ToStringOnlyExp(): Επιστρέφει μία συμβολοσειρά που περιέχει μόνο τις εκθέτες του όρου.
5. MultiVarTerm Copy(): *Ε*πιστρέφει ένα αντίγραφο του τρέχοντα όρου. Πρακτικά η μέθοδος αυτή μοντελοποιεί την ιδιότητα που έχει μια struct και χάνεται λόγω του ότι χρησιμοποιούμε κλάσεις.

2.2.2 Κλάση Polynomial

Έχοντας φτιάξει την κλάση MultiVarTerm που μοντελοποιεί έναν πολυωνυμικό όρο, είναι εύκολο να κατασκευαστεί η κλάση Polynomial η οποία απλά θα είναι μία λίστα από αντικείμενα MultiVarTerm. Για την αποθήκευση των όρων θα χρησιμοποιηθεί μία λίστα (List<MultiVarTerm>). Το σημαντικότερο ρόλο θα παίξουν οι διαπροσωπείες IList<MultiVarTerm> και IEquatable<MultiVarTerm> που υλοποιεί η κλάση Polynomial.

2.2.2.1 Διαπροσωπείες

Εδώ θα αναφερθούμε αναλυτικά στις διαπροσωπείες `IList<MultiVarTerm>` και `IEquatable<Polynomial>` της C#.

Η `IList<MultiVarTerm>` κληρονομεί και τις ιδιότητες και τις μεθόδους των διαπροσωπειών `ICollection<MultiVarTerm>` και `IEnumerable<MultiVarTerm>`. Αυτό σημαίνει ότι λόγω του `IEnumerable` το πολυώνυμο ως λίστα θα μπορεί να απαριθμηθεί, δηλαδή να χρησιμοποιηθούν `Enumerators` και βρόχοι `foreach` για την προσπέλασή του. Λόγω του `ICollection` το πολυώνυμο θεωρείται εγγενώς από τη C# ως μία συλλογή δεδομένων, και άρα διευκολύνεται η τυχόν μετατροπή του σε άλλον τύπο δεδομένων, καθώς και η δημιουργία πολυωνύμων από όρους που είναι αποθηκευμένοι σε οποιαδήποτε συλλογή δεδομένων (πίνακα, λίστα, λεξικό κλπ). Τέλος η `IList` αναπαριστά υποχρεωτικά το πολυώνυμο ως μία λίστα. Ακόμα και αν εμείς εσωτερικά χρησιμοποιούσαμε έναν πίνακα για να αναπαραστήσουμε το πολυώνυμο, εξωτερικά θα φαίνεται πάντα ως λίστα. Τέλος, η διαπροσωπεία `IEquatable` θα μας ορίσει ισότητα πολυωνύμων. Μέσω αυτής της διαπροσωπείας είναι που θα βλέπουμε αν δύο ίχνη (δύο πολυώνυμα δηλαδή) είναι ίσα ή όχι.

Οι μέθοδοι και οι ιδιότητες (properties) που υλοποιούνται λόγω των παραπάνω διαπροσωπειών είναι οι εξής:

ICollection<MultiVarTerm>:

1. `int Count`: Πρόκειται για μία ιδιότητα η οποία απλά θα αποθηκεύει τον αριθμό των στοιχείων της λίστας, δηλαδή τον αριθμό των όρων που θα έχει ένα πολυώνυμο.
2. `bool IsReadOnly`: Αυτή η ιδιότητα θα καθορίζει αν το πολυώνυμο για είναι read-only (μόνο για ανάγνωση). Σε περίπτωση που έχει την τιμή `true`, δεν θα επιτρέπεται καμία απολύτως αλλαγή στο πολυώνυμο.
3. `void Add(MultiVarTerm item)`: Προσθέτει έναν όρο στο πολυώνυμο.
4. `void Clear()`: Σβήνει όλο το πολυώνυμο και το αντικαθιστά πρακτικά με το σταθερο πολύωνυμο 1.
5. `bool Contains(MultiVarTerm item)`: Ελέγχει αν ένας όρος υπάρχει ήδη στο πολυώνυμο, αγνοώντας τον συντελεστή του όρου. Δηλαδή ελέγχει αν υπάρχει ένας όρος με τις ίδιες μεταβλητές και τους ίδιους εκθέτες με αυτόν που δίνει ο χρήστης. Η μέθοδος αυτή έχει οριστεί έτσι ώστε να είναι εύκολο να ορίσουμε την πρόσθεση των πολυωνύμων (γιατί αν ένας όρος υπάρχει ήδη μέσα στο πολυώνυμο απλά με άλλον συντελεστή τότε τους προσθέτουμε μεταξύ τους, δεν χρειάζεται να εισάγουμε ξανά τον ίδιο όρο).
6. `void CopyTo(MultiVarTerm[] array, int arrayIndex)`: Αντιγράφει το πολυώνυμο σε έναν πίνακα.

7. `bool Remove(MultiVarTerm item)`: Αφαιρεί έναν όρο από το πολυώνυμο.

IEnumerable<MultiVarTerm>:

1. `IEnumerator<MultiVarTerm> GetEnumerator()`: Επιστρέφει έναν διαπροσπελαστή για μία λίστα αντικειμένων `MultiVarTerm`.
2. `IEnumerator IEnumerable.GetEnumerator()`: Και αυτή η μέθοδος επιστρέφει έναν διαπροσπελαστή, με την διαφορά ότι είναι ένας αφηρημένος διαπροσπελαστής.

IList<MultiVarTerm>:

1. `MultiVarTerm this[int index]`: Μέσω αυτής της μεθόδου είναι που επιτυγχάνουμε τη δομή της λίστας στο πολυώνυμο. Αν υποθέσουμε ότι poly είναι ένα αντικείμενο της κλάσης `Polynomial`, μέσα από αυτή τη μέθοδο οι εκφράσεις `poly[i]`, όπου `i` είναι ένας ακέραιος, έχουν νόημα.
2. `int IndexOf(MultiVarTerm item)`: Βρίσκει σε ποια θέση βρίσκεται ένας όρος μέσα στο πολυώνυμο. Και εδώ αγνοείται ο συντελεστής.
3. `void Insert(int index, MultiVarTerm item)`: Εισάγει έναν όρο σε μία συγκεκριμένη θέση μέσα στο πολυώνυμο.
4. `void RemoveAt(int index)`: Αφαιρεί έναν όρο από μία συγκεκριμένη θέση από το πολυώνυμο.

IEquatable<Polynomial>

1. `Boolean Equals(Polynomial p)`: Η μέθοδος αυτή εξετάζει το αν δύο πολυώνυμα είναι ίσα. Η ισότητα μεταξύ δύο πολυωνύμων ορίζεται όπως ορίζεται και αλγεβρικά, δηλαδή θα πρέπει οι συντελεστές των ίδιων όρων που υπάρχουν και στα δύο πολυώνυμα να είναι ίσοι.

Επίσης έχουν οριστεί αρκετοί τελεστές για πράξεις μεταξύ πολυωνύμων, όρων και αριθμών:

1. `Polynomial operator *(Polynomial p1, Polynomial p2)`: Πολλαπλασιάζει δύο πολυώνυμα μεταξύ τους. Ο αλγόριθμός σε αυτή την περίπτωση έχει πολυπλοκότητα $O(nm)$ όπου n το πλήθος των όρων του πρώτου πολυωνύμου και m του δεύτερου πολυωνύμου. Υπάρχει δυνατότητα βελτίωσης του αλγορίθμου του πολλαπλασιασμού μέσω του FFT (Fast Fourier Transform).
2. `Polynomial operator *(Polynomial p, MultiVarTerm t)`: Εδώ πολλαπλασιάζεται ένα πολυώνυμο με έναν πολυωνυμικό όρο.
3. `Polynomial operator *(MultiVarTerm t, Polynomial p)`: Ορίζεται καταχρηστικά και έχει το ίδιο αποτέλεσμα με το 2.

4. Polynomial operator *(Polynomial p, double c): Πολλαπλασιάζει ένα πολυώνυμο με έναν πραγματικό αριθμό. Απλά δηλαδή, πολλαπλασιάζονται όλοι οι συντελεστές με αυτόν τον πραγματικό αριθμό.
5. Polynomial operator *(double c, Polynomial p): Ορίζεται καταχρηστικά και έχει το ίδιο αποτέλεσμα με το 4.
6. Polynomial operator +(Polynomial p1, Polynomial p2): Προσθέτει δύο πολυώνυμα. Δημιουργείται δηλαδή ένα νέο πολυώνυμο το οποίο περιέχει τους όρους και των δύο πολυωνύμων. Προφανώς αν υπάρχει ένας όρος με τις ίδιες μεταβλητές και τους ίδιους εκθέτες αλλά με διαφορετικούς συντελεστές στα δύο πολυώνυμα, τότε απλά οι συντελεστές προστίθενται.
7. Polynomial operator +(Polynomial p, MultiVarTerm t): Προσθέτει ένα πολυώνυμο με έναν πολυωνυμικό όρο. Η διαδικασία είναι όμοια με την πρόσθεση δύο πολυωνύμων.
8. Polynomial operator +(MultiVarTerm t, Polynomial p): Ορίζεται καταχρηστικά και έχει το ίδιο αποτέλεσμα με το 6.
9. Polynomial operator +(Polynomial p, double c): Προσθέτει ένα πολυώνυμο με έναν αριθμό. Ο αριθμός προστίθεται στη λίστα του πολυωνύμου ως ένας σταθερός πολυωνυμικός όρος.
10. Polynomial operator +(double c, Polynomial p): Ορίζεται καταχρηστικά και έχει το ίδιο αποτέλεσμα με το 9.
11. Polynomial operator -(Polynomial p): Υπολογίζει το αντίθετο πολυώνυμο. Αυτό επιτυγχάνεται επιστρέφοντας ένα νέο πολυώνυμο το οποίο έχει τους ίδιους όρους αλλά με τους αντίθετους συντελεστές.
12. Polynomial operator -(Polynomial p1, Polynomial p2): Αφαιρεί δύο πολυώνυμα μεταξύ τους. Η μέθοδος αυτή ορίζεται ως η πρόσθεση του αντίθετου του p2 στο p1.
13. Polynomial operator -(Polynomial p, MultiVarTerm t): Αφαιρεί έναν όρο με ένα πολυώνυμο.
14. Polynomial operator -(MultiVarTerm t, Polynomial p): Αφαιρεί ένα πολυώνυμο με έναν όρο.
15. Polynomial operator -(Polynomial p1, double c): Αφαιρεί έναν αριθμό με ένα πολυώνυμο.
16. Polynomial operator -(double c, Polynomial p): Αφαιρεί ένα πολυώνυμο με έναν αριθμό.

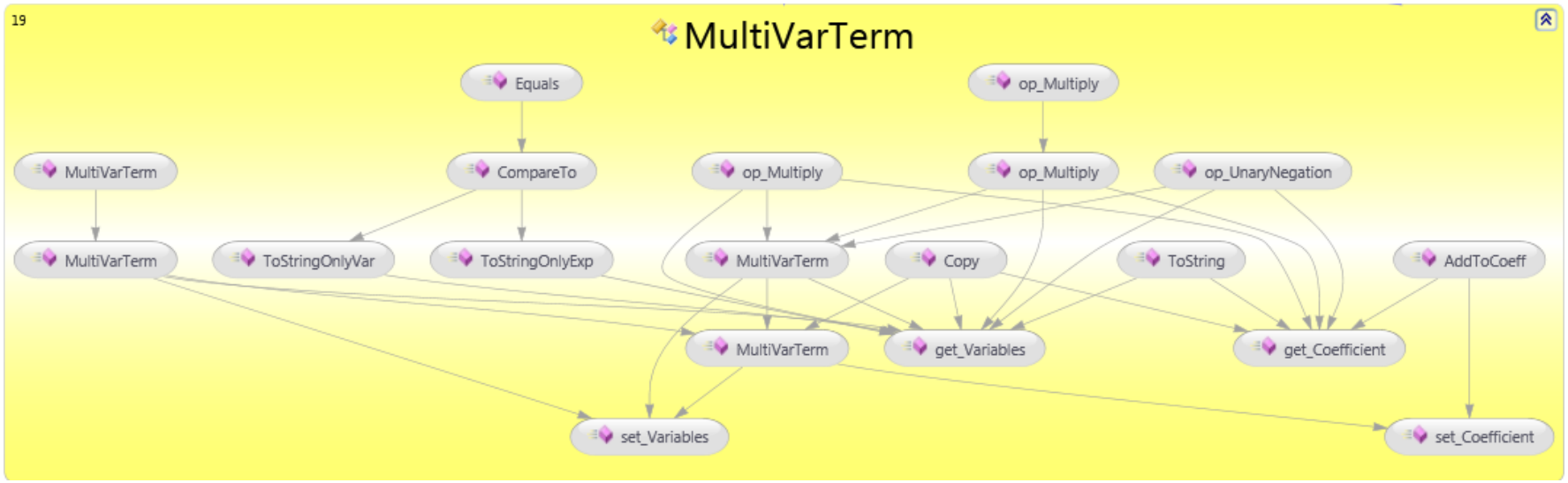
Τέλος υπάρχουν και κάποιες λίγες βοηθητικές μέθοδοι. Κάποιες από αυτές είναι ιδιαίτερα χρήσιμες και χρησιμοποιούνται αρκετές φορές:

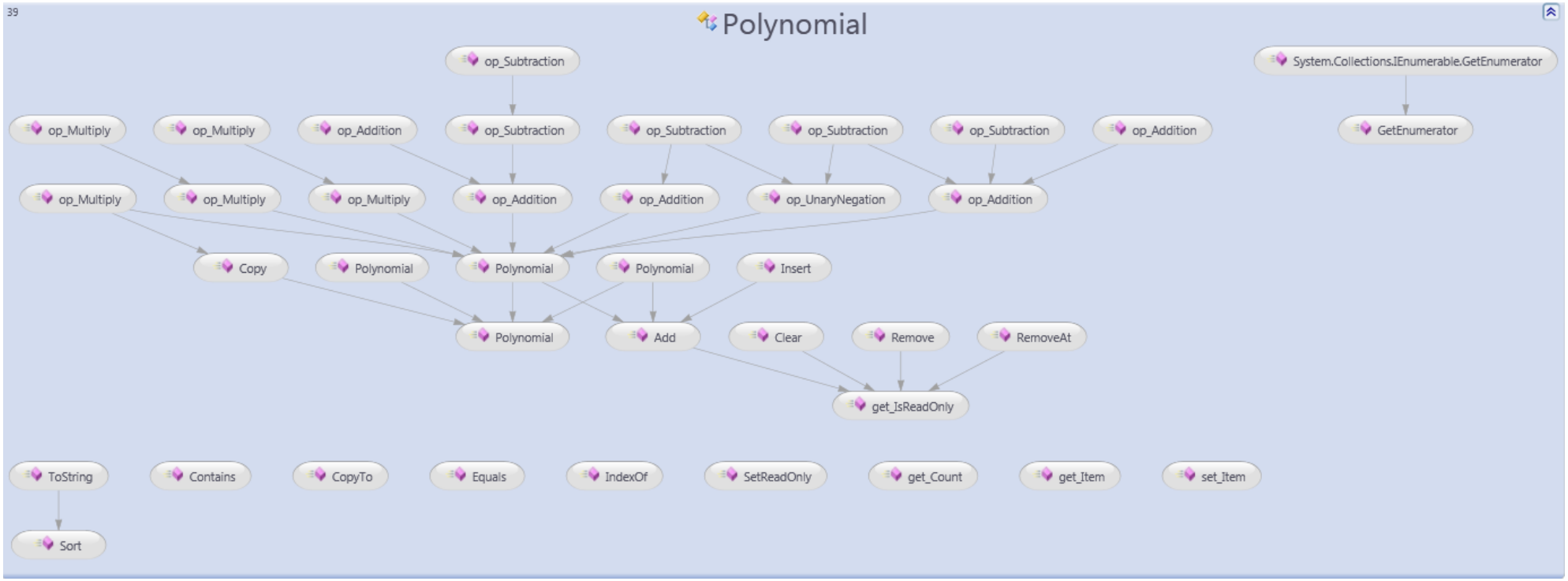
1. string ToString(): Επιστρέφει μία συμβολοσειρά που αναπαριστά το πολυώνυμο ακριβώς όπως θα το γράφαμε και κανονικά.

2. Polynomial Copy(): Όπως και στην κλάση MultiVarTerm η μέθοδος αυτή δημιουργεί ένα ανεξάρτητο αντίγραφο του τρέχοντος πολυωνύμου.
3. void Sort(): Ταξινομεί το πολυώνυμο. Η ταξινόμηση βασίζεται στο πως συγκρίνονται οι όροι τύπου MultiVarTerm όπως περιγράφηκε στο προηγούμενο κεφάλαιο.
4. void SetReadOnly(bool readOnly): Αλλάζει την τιμή της ιδιότητας IsReadOnly.

2.2.3 Διαγράμματα των κλάσεων

Τα παρακάτω διαγράμματα δείχνουν ξεκάθαρα την δομή των κλάσεων MultiVarTerm και Polynomial. Φαίνονται όλες οι μέθοδοι που υπάρχουν στις κλάσεις και τις αλληλεξαρτήσεις που υπάρχουν μεταξύ τους. Ένα βέλος συμβολίζει ότι μία μέθοδος καλεί μία άλλη μέθοδο. Τα διαγράμματα έχουν παραχθεί μέσα από το Visual Studio 2010.


19
MultiVarTerm
Equals
op_Multiply
MultiVarTerm
CompareTo
op_Multiply
op_Multiply
op_UnaryNegation
MultiVarTerm
ToStringOnlyVar
ToStringOnlyExp
MultiVarTerm
Copy
ToString
AddToCoeff
MultiVarTerm
get_Variables
get_Coefficient
set_Variables
set_Coefficient


39
Polynomial
op_Subtraction
System.Collections.IEnumerable.GetEnumerator
op_Multiply
op_Multiply
op_Addition
op_Subtraction
op_Subtraction
op_Subtraction
op_Subtraction
op_Addition
GetEnumerator
op_Multiply
op_Multiply
op_Multiply
op_Addition
op_Addition
op_UnaryNegation
op_Addition
Copy
Polynomial
Polynomial
Polynomial
Insert
Polynomial
Add
Clear
Remove
RemoveAt
get_IsReadOnly
ToString
Contains
CopyTo
Equals
IndexOf
SetReadOnly
get_Count
get_Item
set_Item
Sort

2.3 Braids: υπολογισμός του ίχνους

2.3.1 Βασικά στοιχεία του προγράμματος

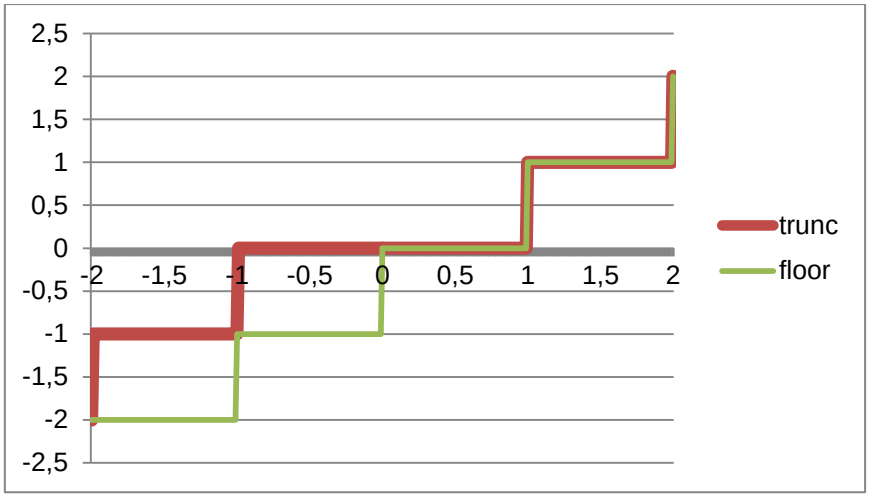
Σε αυτό το κεφάλαιο θα περιγραφεί η δομή του προγράμματος Braids το οποίο θα υπολογίσει τα ίχνη στις άλγεβρες Hecke τύπου A και στις άλγεβρες Yokonuma-Hecke. Πριν ξεκινήσει η περιγραφή της αναπαράστασης μίας λέξης (που θα ανήκει σε μία από τις δύο άλγεβρες) στον υπολογιστή και τον τρόπο υπολογισμού του ίχνους, θα δούμε τι χρειάζεται να υπάρχει πιο πριν για τον υπολογισμό του ίχνους.

2.3.1.1 Αλγόριθμοι διαίρεσης και υπόλοιπο

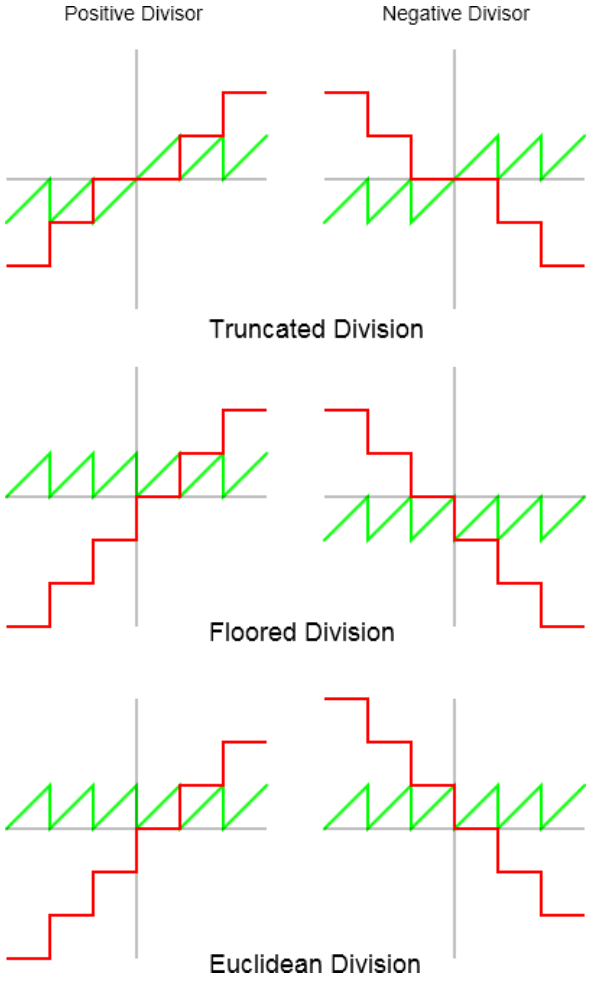
Όπως ήδη γνωρίζουμε στην άλγεβρα Yokonuma-Hecke $Y_{d,n}(u)$ υπάρχει η σχέση $t_i^d = 1$. Αυτό σημαίνει ότι οι πράξεις στους εκθέτες του framing part γίνονται $mod\ d$. Όμως συνήθως στις γλώσσες προγραμματισμού (όπως συμβαίνει και με την C#) το υπόλοιπο μεταξύ δύο αριθμών υπολογίζεται μέσω του τελεστή $\%$. Το πρόβλημα είναι ότι στις γλώσσες προγραμματισμού το υπόλοιπο και το πηλίκο δεν υπολογίζονται πάντα βάσει του Ευκλείδειου αλγορίθμου διαίρεσης. Συγκεκριμένα υπάρχουν οι εξής τρεις εναλλακτικοί ορισμοί της διαίρεσης στους υπολογιστές (θεωρούμε n τον διαιρετέο, d τον διαιρέτη, q το πηλίκο και r το υπόλοιπο):

1. Ευκλείδειος αλγόριθμος: $n = dq + r$ όπου $0 \leq r < d$. Εδώ το πρόσημο του πηλίκου εξαρτάται από το πρόσημο του διαιρέτη και το πρόσημο του διαιρετέου και το υπόλοιπο είναι είτε 0 είτε ένας θετικό αριθμός.
2. Truncated διαίρεση: $n = dq + r$ όπου $q = trunc(n/d)$ και $r = a - nq$. Η συνάρτηση trunc κόβει όλα τα δεκαδικά ψηφία ενός αριθμού (από ένα σημείο και μετά) χωρίς να τον στρογγυλοποιεί. Οπότε εδώ το υπόλοιπο έχει το πρόσημο του διαιρέτη.
3. Floored διαίρεση: $n = dq + r$ όπου $q = floor(n/d)$ και $r = a - n\lfloor \frac{a}{n} \rfloor$. Η συνάρτηση floor είναι πρακτικά το κάτω δεκαδικό μέρος ενός αριθμού.

Στο παρακάτω διάγραμμα φαίνονται τα αποτελέσματα των συναρτήσεων trunc και floor, οι οποίες χρησιμοποιούνται για το πηλίκο:



Παρατηρούμε ότι για θετικούς αριθμούς οι δύο συναρτήσεις έχουν τις ίδιες τιμές, οπότε αν ο διαιρέτης και ο διαιρετέος έχουν τις ίδιες τιμές, οι δύο διαιρέσεις θα βγάζουν τις ίδιες τιμές. Σε περίπτωση όμως που έχουν διαφορετικά πρόσημα, τότε τα πηλίκα που θα βγουν από τους δύο αντίστοιχους τύπους διαίρεσης θα διαφέρουν κατά ένα, οπότε και τα υπόλοιπα θα είναι διαφορετικά. Το παρακάτω διάγραμμα περιγράφει τα αποτελέσματα και των τριών αλγορίθμων διαίρεσης:



Οι πράσινες γραμμές είναι το υπόλοιπο της διαίρεσης και οι κόκκινες το πηλίκο.

Στη C# χρησιμοποιείται ο αλγόριθμος της truncated διαίρεσης, ο οποίος για θετικούς διαιρετέους δίνει το ίδιο αποτέλεσμα με την Ευκλείδεια διαίρεση. Όμως για αρνητικούς διαιρετέους αυτό δεν συμβαίνει. Οπότε αν χρησιμοποιούσαμε τον τελεστή `%` της C# θα είχαμε πρόβλημα στους υπολογισμούς σε περίπτωση που είχαμε αρνητικούς εκθέτες στο framing part. Για παράδειγμα θα είχαμε ότι `-2 % 3 = -2` ενώ σύμφωνα με την Ευκλείδεια διαίρεση $-2 \equiv 1\ (mod\ 3)$.

Για αυτόν τον λόγο κατασκευάστηκε η κλάση `Compute`. Η κλάση περιέχει μόνο μία συνάρτηση, την `int Mod(int a, int n)` η οποία υπολογίζει το υπόλοιπο της διαίρεσης του `a` με τον `n` μέσω της Ευκλείδειας διαίρεσης. Σε όλο το πρόγραμμα χρησιμοποιείται η συνάρτηση `Mod` εκτός ελαχίστων περιπτώσεων που χρησιμοποιείται ο τελεστής `%` (όπου είναι βέβαιο ότι το αποτέλεσμα είναι σωστό).

2.3.1.2 Κλάση Parser

Το πρόγραμμα έχει κατασκευαστεί έτσι ώστε η λέξη να δίνεται από το χρήστη με τη μορφή που θα την έγραφε κανονικά. Δηλαδή το πρόγραμμα λαμβάνει ως είσοδο μία συμβολοσειρά της μορφής “g1^2 t2^-3 g3 g5^7”. Για αυτόν τον λόγο κατασκευάστηκε ένας parser. Ως parser ονομάζουμε ένα πρόγραμμα που δέχεται ως είσοδο ένα κείμενο που περιέχει δεδομένα και το μετατρέπει σε μία μορφή κατανοητή για τον υπολογιστή.

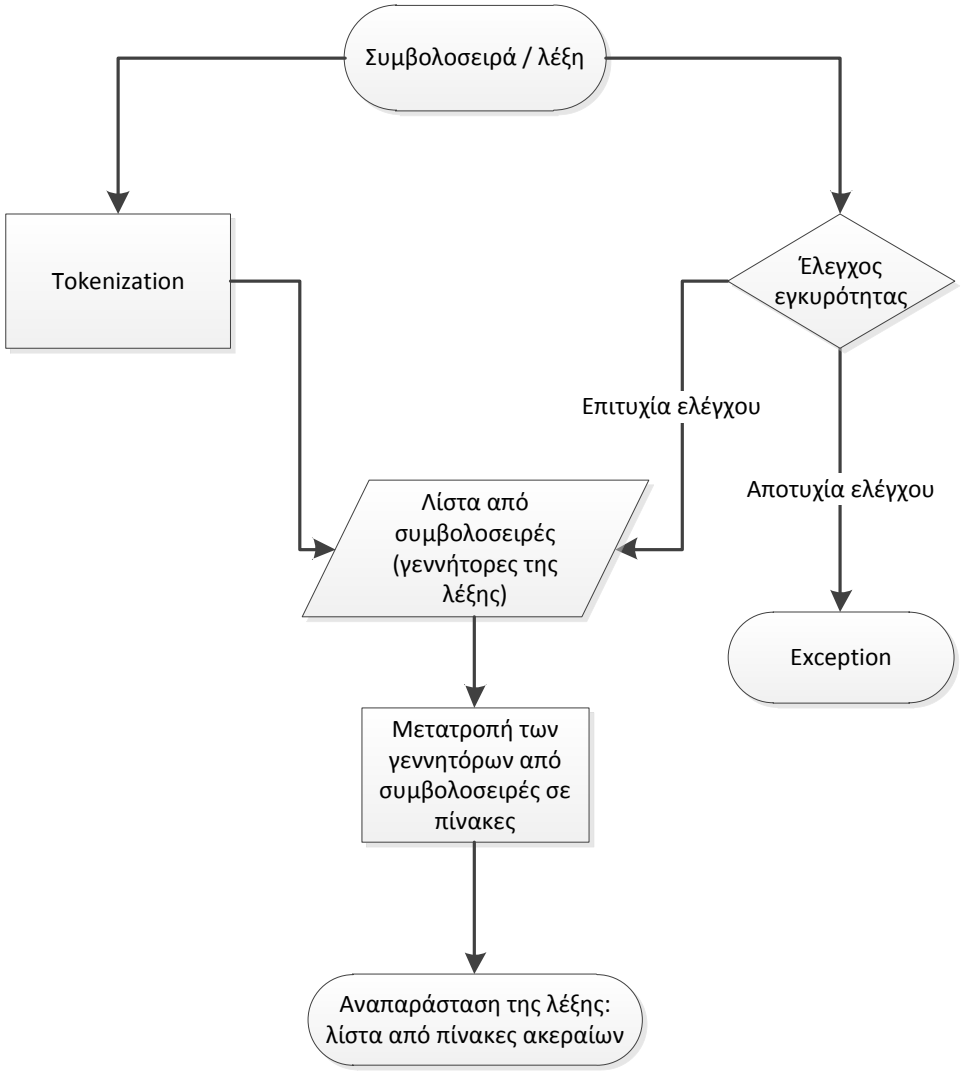
Η περίπτωση μας είναι αρκετά απλή για να υλοποιηθεί. Ο parser καθώς διαβάζει το κείμενο υπάρχει περίπτωση να συναντήσει τους εξής χαρακτήρες:

- ‘g’: σημαίνει ότι ξεκινάει ένα μέρος του braiding part
- ‘t’: σημαίνει ότι ξεκινάει ένα μέρος του framing part
- ^: έχουμε υψώσει ένα στοιχείο σε κάποια δύναμη
- Αριθμό: ένας αριθμός που θα μπορούσε να είναι είτε δείκτης είτε εκθέτης
- Κενός χαρακτήρας: δεν λαμβάνονται καθόλου υπ’ όψιν.

Δηλαδή τα «κουπόνια» (tokens) για τον parser μας θα είναι ο χαρακτήρας ‘g’, ο χαρακτήρας ‘t’, ο χαρακτήρας ‘^’ και οποιοσδήποτε αριθμός προκύψει.

Ως ένα πρώτο βήμα θα πρέπει ο parser να «κόβει» τη λέξη στα στοιχεία από τα οποία αποτελείται (π.χ. η λέξη “g1^2 t2^-3 g3 g5^7” να διασπάται στις συμβολοσειρές “g1^2” “t2^-3” “g3” “g5^7” – η διαδικασία αυτή αποτελείται tokenization) και να ελέγχεται παράλληλα η εγκυρότητα της λέξης. Τέλος κάθε ένα από τα στοιχεία αυτά θα πρέπει να μετατρέπεται σε μορφή αναγνώσιμη από τον υπολογιστή.

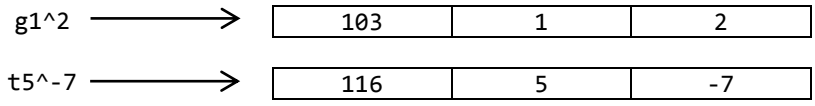
Πριν αναλυθεί ακριβώς η λειτουργία του Parser, δίνεται το παρακάτω διάγραμμα ροής της λειτουργίας του:



Την λειτουργία του parser που περιγράψαμε υλοποιεί η κλάση Parser. Η κλάση αυτή είναι στατική, δηλαδή δεν μπορούν να δημιουργηθούν αντικείμενα με βάση αυτή τη κλάση αλλά μόνο να κληθούν οι μέθοδοι που περιέχει, οι οποίες είναι και αυτές στατικές.

Η πρώτη μέθοδος είναι η `List<String> Validate(String word)`. Η μέθοδος αντιστοιχεί στα στάδια του tokenization και του ελέγχου εγκυρότητας και δεν καλείται κάθε φορά που δεχόμαστε μία λέξη από τη δεύτερη μέθοδο. Ως έξοδο δίνει μία λίστα από συμβολοσειρές που περιέχουν τους γεννήτορες της λέξης. Συγκεκριμένα, διαβάζει έναν προς έναν τους χαρακτήρες της λέξης, και κάθε φορά που συναντάει είτε ‘g’ είτε ‘t’ σπάει τη λέξη. Σε περίπτωση που εντοπιστεί κάποιο συντακτικό λάθος (π.χ. η λέξη “g2^t2” θεωρείται λάθος γιατί δεν έχει προσδιοριστεί ο εκθέτης του γεννήτορα “g2”) η μέθοδος επιστρέφει ένα exception το οποίο περιέχει την περιγραφή του λάθους που παρουσιάστηκε. Αν δεν υπάρχει κάποιο λάθος στη λέξη, απλά επιστρέφεται η λίστα με τα στοιχεία της λέξης.

Η δεύτερη μέθοδος είναι η `Word Parse(String word)`. Το πρώτο πράγμα που κάνει είναι να καλέσει τη μέθοδο `Validate` η οποία της επιστρέφει τη λίστα με τα στοιχεία της λέξης. Έπειτα η `Parse` σπάει κάθε στοιχεία σε έναν πίνακα ακεραίων με 3 στοιχεία: στο πρώτο στοιχείο είναι η αναπαράσταση σε κώδικα ASCII του χαρακτήρα 'g' ή του χαρακτήρα 't', το δεύτερο στοιχείο είναι ο δείκτης και το τρίτο στοιχείο είναι ο εκθέτης. Για παράδειγμα δύο μετατροπές θα μπορούσαν να είναι οι εξής:



Τέλος η μέθοδος επιστρέφει ένα αντικείμενο τύπου `Word`. Όπως θα δούμε αργότερα, έχει οριστεί κλάση που μοντελοποιεί μία λέξη. Προς το παρόν όμως μπορούμε να φανταστούμε ότι η `Parse` επιστρέφει μία λίστα με ακέραιους μονοδιάστατους πίνακες με 3 στοιχεία.

2.3.1.3 Κλάσεις FramingComparer και DistComparer

Στη C# όταν χρειαστεί να ταξινομηθεί ένας πίνακας μέσω της μεθόδου Sort, τότε αυτόματα η γλώσσα αναζητά έναν Comparer για τα αντικείμενα του πίνακα, δηλαδή ένα αντικείμενο το οποίο θα συγκρίνει δύο αντικείμενα του πίνακα και θα επιστρέφει μία ακέραια τιμή ανάλογα με το πιο είναι μικρότερο.

Εδώ έχουμε δημιουργήσει δύο κλάσεις που υλοποιούν την διαπροσωπεία `IComparer<int[]>`. Η `FramingComparer` μπορεί να συγκρίνει ακέραιους πίνακες μεταξύ τους. Θεωρούμε ότι οι πίνακες αυτοί θα έχουν δύο στοιχεία. Η `FramingComparer` θα μας χρειαστεί όταν θα θέλουμε να ταξινομήσουμε το framing part μίας λέξης, αφού από τη στιγμή που η αντιμεταθετική ιδιότητα ισχύει για όλα τα στοιχεία του framing part, μπορούμε να τα ταξινομήσουμε με όποια σειρά θέλουμε. Η `DistComparer` συγκρίνει και αυτό ακέραιους πίνακες. Εδώ θεωρούμε ότι έχουμε 3 στοιχεία σε κάθε πίνακα. Συγκεκριμένα θα δέχεται πίνακες που τα δύο πρώτα στοιχεία θα είναι δείκτες που θα δείχνουν σε κάποιες θέσεις του braiding part και το τρίτο στοιχείο είτε η εσωτερική είτε η εξωτερική τους απόσταση και θα συγκρίνει τους πίνακες με βάση αυτή την απόσταση.

2.3.2 Κλάση Word

2.3.2.1 Αναπαράσταση λέξης

Πριν τον υπολογισμό του ίχνους είναι αναγκαίο να αναπαρασταθεί στον υπολογιστή η οποιαδήποτε λέξη που θα δέχεται ως είσοδο το πρόγραμμα. Στο πρόγραμμα υπάρχουν δύο περιπτώσεις: θα υπολογίζεται το ίχνος είτε για την άλγεβρα Hecke είτε για την άλγεβρα Yokonuma-Hecke, άρα θα ήταν πρακτικό να έχουμε μία αναπαράσταση και για τις δύο άλγεβρες. Δηλαδή να δημιουργηθεί μία μόνο κλάση η οποία θα περιγράφει μία λέξη σε οποιαδήποτε από τις δύο άλγεβρες. Οι δύο αυτές άλγεβρες διαφέρουν στο ότι μία λέξη στην

Yokonuma-Hecke διαθέτει και framing part, πράγμα που δεν υπάρχει στην Hecke.

Η λύση σε αυτό το πρόβλημα έρχεται από την τεχνική που εφαρμόζουμε όταν υπολογίζεται ένα ίχνος στην άλγεβρα Yokonuma-Hecke: μέσω της δράσης των γεννητόρων t_i πάνω στους γεννήτορες g_i μεταφέρουμε τους γεννήτορες t_i στο αρχικό μέρος της λέξης, η οποία πλέον ξεχωρίζει σε δύο μέρη: το braiding part και το framing part. Οπότε η αναπαράσταση μίας λέξης στον υπολογιστή είναι εύκολη: μία λέξη θα αναπαρίσταται από δύο λίστες, μία για το framing part και μία για το braiding part.

Όμως κατά τον υπολογισμό του ίχνους θα εμφανίζονται πολυώνυμα ως συντελεστές σε κάποιες λέξεις. Ακόμα και με την απλή τετραγωνική σχέση της άλγεβρας Hecke εμφανίζονται τέτοιοι συντελεστές. Για παράδειγμα:

$$g_3 g_1^2 g_2 = g_3[(q-1)g_1 + q]g_2 = (q-1)g_3 g_1 g_2 + q g_3 g_2$$

Για αυτόν τον λόγο, οι πολυωνυμικοί συντελεστές που δημιουργούνται θα αποθηκεύονται μαζί με τη λέξη στο ίδιο αντικείμενο. Με αυτό το σκεπτικό δημιουργήθηκε η κλάση Word η οποία θα περιέχει τις εξής ιδιότητες/πεδία:

- `Polynomial Coeff:` αποθηκεύει τον πολυωνυμικό συντελεστή που εμφανίζεται μπροστά από τη λέξη. Σε περίπτωση που δεν υπάρχει κάποιος, τότε θεωρούμε τη μονάδα.
- `List<int[]> Framing:` αποθηκεύει το framing part για το πρόγραμμα. Πρόκειται για μία λίστα ακεραίων πινάκων 2 θέσεων που περιέχουν τον δείκτη και τον εκθέτη κάθε γεννήτορα. Η λίστα αυτή θα ταξινομείται με φθίνουσα σειρά ως προς τους δείκτες των στοιχείων t_i έτσι ώστε να βρίσκουμε γρήγορα τον μέγιστο δείκτη που υπάρχει στο framing part.
- `List<int[]> Braiding:` αποθηκεύει το braiding part για το πρόγραμμα. Πρόκειται για μία λίστα ακεραίων πινάκων 2 θέσεων που περιέχουν τον δείκτη και τον εκθέτη κάθε γεννήτορα.

Για παράδειγμα έστω η λέξη $t_1 g_3 g_4^2 t_2$. Ως ένα πρώτο βήμα θα μετακινήσουμε όλους τους γεννήτορες t_i στην αρχή της λέξης. Οπότε έχουμε $t_1 g_3 g_4^2 t_2 = t_1 g_3 t_2 g_4^2 = t_1 t_3 g_3 g_4^2$. Με την ταξινόμηση του framing part η τελική μας λέξη θα είναι η $t_3 t_1 g_3 g_4^2$. Οπότε οι ιδιότητες της κλάσης θα πάρουν τις εξής τιμές:

- `Coeff = 1` αφού δεν υπάρχει πολυωνυμικός συντελεστής
- `Framing = {(3,1), (1,1)}`
- `Braiding = {(3,1), (4,2)}`

Οπότε οι δύο παρακάτω λίστες θα δημιουργηθούν:

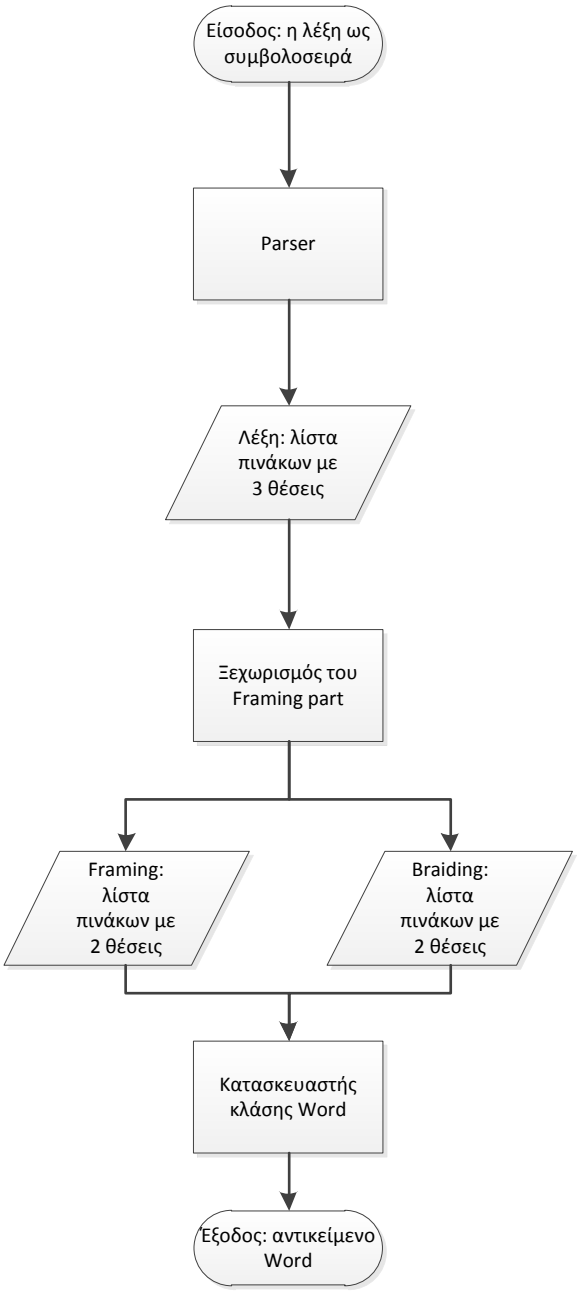
Framing	
3	1
1	1

Braiding	
3	1
4	2

Τα οφέλη από αυτή την αναπαράσταση της λέξης είναι αρκετά:

1. Η αποθήκευση των πολυωνυμικών συντελεστών των λέξεων βοηθάει αρκετά στον υπολογισμό του ίχνους. Το ίχνος μίας λέξης θα ορίζεται ως ο πολυωνυμικός συντελεστής επί το αποτέλεσμα της συνάρτησης ίχνους.
2. Η χρήση δύο ξεχωριστών λιστών αντί μίας λίστας που θα περιείχε όλα τα στοιχεία της λέξης, βοηθάει στο να βρίσκουμε άμεσα τους μέγιστους δείκτες σε κάθε μέρος της λέξης και αν υπάρχουν δυνάμεις. Λόγω του ότι οι δύο λίστες έχουν ακριβώς την ίδια δομή (αποτελούνται από ακέραιους πίνακες), υπάρχει η δυνατότητα να δημιουργηθούν μέθοδοι οι οποίοι θα εντοπίζουν κάποια ιδιότητα (π.χ. την ύπαρξη εκθέτη) οι οποίες θα είναι απλές και θα μπορούν να χρησιμοποιηθούν και για τις δύο λίστες, χωρίς να χρειαστεί να γραφεί κώδικας για κάθε λίστα ξεχωριστά.
3. Η αναπαράσταση της λέξης ως δύο λίστες ταιριάζει απόλυτα με τη δομή της άλγεβρας Yokonuma-Hecke, αλλά είναι δυνατόν να αναπαρίσταται με τον ίδιο ακριβώς τρόπο μία λέξη της άλγεβρας Hecke χωρίς καμία τροποποίηση: για τον υπολογισμό του ίχνους μίας λέξης στην άλγεβρα Hecke η λίστα `Framing` δεν θα λαμβάνεται καθόλου υπ' όψιν. Αυτό δίνει τη δυνατότητα να δοθεί ως είσοδος στο πρόγραμμα μία λέξη της άλγεβρας Yokonuma-Hecke, να υπολογιστεί το ίχνος στη Yokonuma-Hecke και ύστερα να υπολογιστεί το ίχνος στη Hecke σαν να μην υπήρχαν στην λέξη οι γεννήτορες t_i. Τέτοιες περιπτώσεις υπολογισμών είναι συχνές.

Διαγραμματικά η διαδικασία της δημιουργίας ενός αντικειμένου Word θα είναι η εξής:



2.3.2.2 Ιδιωτικές μέθοδοι

Τον βασικότερο ρόλο στην κλάση Word τον έχουν οι ιδιωτικές (private) μέθοδοι που έχουν δημιουργηθεί για να κάνουν απλές εργασίες σε μία λέξη. Πάνω σε κάποιες από αυτές τις μεθόδους βασίζονται οι κατασκευαστές της κλάσης όπως και οι μέθοδοι υπολογισμού των συναρτήσεων ίχνους. Αν και ο ρόλος τους γίνεται πιο σαφής κατά την περιγραφή του υπολογισμού του ίχνους, σε αυτό το κεφάλαιο περιγράφεται αναλυτικά η λειτουργία τους.

Κατά τον υπολογισμό των συναρτήσεων ίχνους και στις δύο άλγεβρες, ψάχνουμε για γέφυρες στο braiding part. Ως γέφυρα ορίζεται μία ακολουθία γεννητόρων $g_n g_{n-1} \cdots g_{i+1} g_i g_{i+1} \cdots g_{n-1} g_n$. Από τις ιδιότητες των αλγεβρών Hecke και Yokonuma-Hecke έχουμε ότι μία γέφυρα αντικαθίσταται ως εξής:

$$g_n g_{n-1} \cdots g_{i+1} g_i g_{i+1} \cdots g_{n-1} g_n = g_i g_{i+1} \cdots g_{n-1} g_n g_{n-1} \cdots g_{i+1} g_i$$

Οπότε η αντικατάσταση μίας γέφυρας πρακτικά εξαφανίζει το ένα από τα δύο g_n και σε περίπτωση που ο δείκτης του είναι ο μέγιστος που εμφανίζεται, μπορούμε να εφαρμόσουμε την ιδιότητα:

$$tr(ag_n) = ztr(a)$$

Ας θεωρήσουμε μία λέξη στην άλγεβρα Hecke η οποία δεν έχει κανέναν εκθέτη (δηλαδή οι εκθέτες είναι όλοι ίσοι με 1) και η οποία έχει παραπάνω από ένα γεννήτορα με μέγιστο δείκτη. Τότε επιλέγουμε δύο τυχαίους δείκτες από αυτούς, και προσπαθούμε να βγάλουμε έξω τα στοιχεία ανάμεσά τους τα οποία μπορούν να βγουν «έξω» από τα στοιχειά με μέγιστο δείκτη μέσω της far commutativity. Για παράδειγμα ας δούμε τη λέξη $g_5 g_3 g_1 g_4 g_3 g_5 g_1$. Τα βήματα που θα ακολουθηθούν είναι τα εξής:

1. Επιλογή δύο μέγιστων δεικτών: εδώ έχουμε μόνο δύο ούτως η άλλως: $\boldsymbol{g_5} g_3 g_1 g_4 g_3 \boldsymbol{g_5} g_1$
2. Για κάθε ένα γεννήτορα ανάμεσα στους μέγιστους γεννήτορες, εφαρμόζουμε την far commutativity προς τα έξω: $g_5 \boldsymbol{g_3} g_1 g_4 g_3 g_5 g_1 = g_3 g_5 \boldsymbol{g_3} g_4 g_3 g_5 g_1 = g_3^2 g_5 g_4 \boldsymbol{g_3} g_5 g_1 = g_3^2 g_5 g_4 g_5 g_3 g_1$
3. Τώρα έχει εμφανιστεί μία γέφυρα και ο υπολογισμός του ίχνους είναι πλέον εύκολος: $tr(g_3^2 g_5 g_4 g_5 g_3 g_1) = tr(g_3^2 g_4 g_5 g_4 g_3 g_1) = ztr(g_3^2 g_4^2 g_3 g_1)$

Συγκεκριμένα έχουμε το παρακάτω λήμμα:

Λήμμα: Έστω μία λέξη στην άλγεβρα Hecke $H_{n+1}(q)$ που περιέχει δύο στοιχεία g_n . Τότε εφαρμόζοντας την far commutativity κατάλληλα, σχηματίζεται μία γέφυρα ή εμφανίζεται ένας γεννήτορας υψωμένος στο τετράφωνο.

Απόδειξη:

Εφαρμόζουμε την ιδιότητα της far commutativity για να μετακινήσουμε προς τα δεξιά τον πρώτο γεννήτορα g_n και προς τα αριστερά τον δεύτερο γεννήτοραg_n.

…	g_n	…	…	…	g_{n-1}	…	g_{n-1}	…	g_n	…

Με τη μεταφορά αυτή οι γεννήτορεςg_n θα είναι δίπλα σε ένα γεννήτορα g_{n-1}. Τότε έχουμε τρεις περιπτώσεις: ή έχει σχηματιστεί η λέξη $\dots g_n g_{n-1} g_n \dots$ οπότε και η γέφυρα έχει εμφανιστεί και ο αλγόριθμος τερματίζει ή έχει σχηματιστεί η λέξη $\dots g_n g_{n-1}^2 g_n \dots$ πάλι τερματίζει ο αλγόριθμος ή έχει σχηματιστεί η λέξη $\dots g_n g_{n-1} \dots g_{n-1} g_n \dots$. Σε αυτή την περίπτωση μετακινούμε τους γεννήτορες g_{n-1} με τον ίδιο τρόπο επαγωγικά, μαζί με τους γεννήτορες

g_n. Δηλαδή για $|n-1-i|>1$ έχουμε: $g_n g_{n-1} g_i = g_i g_n g_{n-1}$ και $g_i g_{n-1} g_n = g_{n-1} g_n g_i$.

Οπότε καταλήγουμε είτε στη λέξη $\dots g_n g_{n-1} g_{n-2} g_{n-1} g_n \dots$ (οπότε πάλι έχουμε γέφυρα και ο αλγόριθμος τερματίζει) είτε στη λέξη $\dots g_n g_{n-1} g_{n-2} \dots g_{n-2} g_{n-1} g_n \dots$ είτε σε λέξη με τετράγωνο. Επαγωγικά συνεχίζουμε την διαδικασία και για μικρότερους δείκτες. Ο αλγόριθμος αυτός σίγουρα τερματίζει με γέφυρα γιατί στην χειρότερη περίπτωση θα σχηματιστεί η λέξη $\dots g_n g_{n-1} \dots g_1 \dots g_{n-1} g_n \dots$, αφού δεν υπάρχει μικρότερος δείκτης από το 1. ■

Για όλη αυτή την διαδικασία έχουν οριστεί μέθοδοι που εκτελούν η καθεμία και από διαφορετικό κομμάτι της μεθόδου. Φυσικά ο αλγόριθμος αυτός λειτουργεί ακριβώς με την ίδια λογική και στο braiding part μίας λέξης στην άλγεβρα Yokonuma-Hecke. Οι μέθοδοι αυτές είναι οι εξής:

- `void Swap(List<int[]> part, int data1, int data2)`: Η μέθοδος αυτή μετακινεί ένα γεννήτορα σε μία λίστα που βρίσκεται στη θέση `data1`, στην θέση `data2`.
- `int BubbleHecke(int start, int end, int offset, bool yokonuma)`: Βρίσκει μέχρι ποιο σημείο μπορεί να μετακινηθεί ένα στοιχείο που βρίσκεται στη θέση `start` προς την θέση `end`. Η λειτουργία της είναι όμοια με την BubbleSort η οποία βρίσκει μέχρι ποια θέση του πίνακα μπορεί να «ανέβει» ένα στοιχείο και ύστερα το μετακινεί. Η μεταβλητή `offset` χρησιμοποιείται σε περίπτωση που θέλουμε να μετακινήσουμε και διπλανά στοιχεία. Η μέθοδος επιστρέφει τη θέση στην οποία έχει μετακινηθεί πλέον ο γεννήτορας που βρισκόταν στη θέση `start`. Η Boolean μεταβλητή `yokonuma` ορίζει αν υπολογίζουμε το ίχνος στις άλγεβρες Yokonuma-Hecke.
- `bool Commute(int startIndex, int endIndex, bool ignoreExp, int offset, bool yokonuma)`: Εφαρμόζει όλες τις πιθανές αντιμεταθέσεις ανάμεσα σε δύο γεννήτορες με μέγιστους δείκτες. Ως όρισμα στη συνάρτηση δίνουμε τις θέσεις των δύο μέγιστων στοιχείων. Επειδή όμως είναι πιθανόν να υπάρχουν γεννήτορες με εκθέτες ανάμεσα στις δύο θέσεις, η μέθοδος αυτή επιστρέφει μία Boolean μεταβλητή η οποία είναι true αν έχει βρεθεί γεννήτορας με εκθέτη. Σε αυτή την περίπτωση δεν γίνεται καμία αντιμετάθεση στοιχείων.
- `void SearchCommute(int[] maxIndexes, bool yokonuma)`: Δέχεται ως όρισμα έναν πίνακα με τους δείκτες όπου βρίσκονται τα μέγιστα στοιχεία. Αναζητά αυτά με την μεγαλύτερη απόσταση μεταξύ τους, και καλεί την `Commute` για να γίνουν οι αντιμεταθέσεις.

- `Tuple<int, int> FindBridge()`: Βρίσκει αν υπάρχει γέφυρα στη λέξη και επιστρέφει τους δείκτες στους οποίους βρίσκονται τα μέγιστα στοιχεία.
- `void ReplaceBridge(int start, int end)`: Αντικαθιστά τη γέφυρα σε μία λέξη.

Επίσης υπάρχουν και κάποιες άλλες ιδιωτικές μέθοδοι οι οποίες εκτελούν πολύ πιο απλές διεργασίες αλλά χρειάζονται για τον υπολογισμό του ίχνους:

- `bool HasPowers(List<int[]> part)`: Ελέγχει αν υπάρχουν γεννήτορες με εκθέτη διάφορο της μονάδας. Δέχεται ως όρισμα είτε τη λίστα του braiding part είτε τη λίστα του framing part. Επιστρέφει μία Boolean μεταβλητή ανάλογα με το αν βρει εκθέτες η όχι.
- `int[] FindPowers(List<int[]> part)`: Σε αντίθεση με την `HasPowers` που απλά εντοπίζει το αν υπάρχουν δυνάμεις, η μέθοδος αυτή επιστρέφει ένα πίνακα με τους δείκτες των γεννητόρων όπου υπάρχει ο μέγιστος εκθέτης που εμφανίζεται.
- `int[] FindMaxIndexes(List<int[]> part)`: Η μέθοδος αυτή βρίσκει τους δείκτες των μέγιστων γεννητόρων και τους επιστρέφει σε έναν πίνακα.
- `void RemoveDuplicates()`: Υπάρχει περίπτωση στη λέξη που θα δώσει ο χρήστης να υπάρχουν διπλανοί γεννήτορες με ίδιο δείκτη π.χ. $g_2 g_1 g_1$. Η μέθοδος αυτή αναλαμβάνει να τα ενώσει σε ένα και να προσθέσει τους εκθέτες. Ελέγχει και το braiding part και το framing part.
- `void RemoveModDuplicates(int d)`: Εφαρμόζει τη σχέση $t_i^d = 1$ στο framing part. Δηλαδή υπολογίζει τους εκθέτες των γεννητόρων του framing part ως προς modulo d.
- `void DeleteZeroExp()`: Διαγράφει γεννήτορες με μηδενικό εκθέτη από το braiding και από framing part
- `Word[] Split(bool frame, int pos)`: Σπάει τη λέξη σε δύο λέξεις. Συγκεκριμένα δοθέντος ενός δείκτη η μέθοδος αυτή επιστρέφει το έναν πίνακα που περιέχει δύο λέξεις: η μία είναι το κομμάτι της αρχικής λέξης που βρίσκεται αριστερά από την θέση και η άλλη το δεξιό κομμάτι.
- `static List<int[]> MergeHecke(Word w1, int[] elem)`: Ενώνει μία λέξη της άλγεβρας Hecke με ένα στοιχείο. Πρακτικά είναι η σύνθεση δυο λέξεων όπου η μία έχει μόνο ένα στοιχείο.
- `static List<int[]> MergeHeckeWords(Word w1, Word w2)`: Συνθέτει δύο λέξεις της άλγεβρας Hecke και δημιουργεί μία καινούρια. Αν και έχει οριστεί τελεστής σύνθεσης / πολλαπλασιασμού στην κλάση Word, είναι καλά ορισμένος για την άλγεβρα Yokonuma-Hecke και όχι για την Hecke.

- `Tuple< List<int[]>, List<int[]> > MoveFramingPart( List<int[]> elements )`: Η μέθοδος αυτή λαμβάνει τη λίστα που εξάγει ο Parser και εξάγει δύο λίστες, μία για το framing part και μία για το braiding part. Λαμβάνει υπ' όψιν της τη δράση των framing γεννητόρων πάνω στους braiding γεννήτορες.

2.3.2.3 Κατασκευαστές

Στη κλάση αυτή υπάρχουν 4 κατασκευαστές οι οποίοι δημιουργούν ένα αντικείμενο τύπου Word. Συγκεκριμένα είναι οι παρακάτω:

- `Word()`: Δημιουργεί μία νέα κενή λέξη. Δηλαδή μία λέξη με πολυωνυμικό συντελεστή τη μονάδα, και κενές τις λίστες που αντιστοιχούν στο framing και στο braiding part.
- `Word(List<int[]> elements)`: Δημιουργεί μία λέξη από μία λίστα ακεραίων πινάκων. Είναι ο κατασκευαστής που καλείται συχνότερα, αφού είναι ο κατασκευαστής που δέχεται τη λίστα που εξάγει ο parser. Διαχωρίζει το framing και το braiding part μέσω της μεθόδου `MoveFramingPart`. Η λέξη που θα δημιουργηθεί θα έχει ως πολυωνυμικό συντελεστή τη μονάδα.
- `Word(List<int[]> elements, Polynomial coefficient)`: Έχει την ίδια ακριβώς λειτουργία με τον πιο πάνω κατασκευαστή, με την διαφορά ότι ο πολυωνυμικός συντελεστής που θα χρησιμοποιηθεί δίνεται ως όρισμα.
- `Word(List<int[]> framingPart, List<int[]> braidingPart, Polynomial coefficient)`: Ο κατασκευαστής αυτός δημιουργεί μία νέα λέξη δοθέντων του framing part, του braiding part και του πολυωνυμικού συντελεστή.

Όλοι οι κατασκευαστές καλούν τη μέθοδο RemoveDupliacates(). Δηλαδή ένα αντικείμενο της κλάσης Word θα αναπαριστά μία λέξη στην οποία δεν θα υπάρχουν δύο φορές ίδιοι framing γεννήτορες και στην οποία το framing part θα είναι ταξινομημένο κατά φθίνουσα σειρά των δεικτών. Όμως οι εκθέτες του framing part μπορεί να είναι οποιοσδήποτε αριθμός, οι πράξεις modulo d γίνονται την ώρα του υπολογισμού του ίχνους.

2.3.2.4 Γινόμενο λέξεων

Στο πρόγραμμα έχει οριστεί – σε πλήρη αντιστοιχία με την πράξη του γινομένου της άλγεβρας Yokonuma-Hecke – μία πράξη γινομένου ανάμεσα σε αντικείμενα τύπου Word. Συγκεκριμένα θα είναι δυνατόν να πολλαπλασιάζουμε δύο αντικείμενα που αναπαριστούν δύο λέξεις και το αποτέλεσμα να είναι ένα νέο αντικείμενο που θα αναπαριστά το γινόμενο των δύο λέξεων.

Έστω ότι έχουμε δύο αντικείμενα w1 και w2 τύπου Word που αναπαριστούν δύο διαφορετικές λέξεις. Σε περίπτωση που η w2 δεν έχει framing part τότε ο

πολλαπλασιασμός των λέξεων είναι απλός: απλά ενώνουμε τα braiding part των δύο λέξεων. Αν όμως η w2 διαθέτει framing part, τότε ενώνουμε τις δύο λέξεις σε μία λίστα με πίνακες 3 θέσεων, με την ίδια μορφή που εξάγει και ο parser και δίνουμε τη λίστα αυτή ως όρισμα στον κατασκευαστή, ο οποίος αναλαμβάνει αυτόματα να ξεχωρίσει το framing part της νέας λέξης. Φυσικά οι πολυωνυμικοί συντελεστές των δύο λέξεων πολλαπλασιάζονται μεταξύ τους.

Έχει οριστεί ακόμα και η δυνατότητα του πολλαπλασιασμού μίας λέξης με έναν πίνακα 3 θέσεων, ο οποίος θα αναπαριστά είτε έναν framing γεννήτορα είτε έναν braiding γεννήτορα. Η δυνατότητα αυτή είναι ιδιαιτέρως χρήσιμη όταν στον αλγόριθμο υπολογισμού του ίχνους θα αναλύουμε τη λέξη σε απλούστερες.

Επίσης έχει οριστεί και πολλαπλασιασμός μεταξύ λέξης και πολυωνύμου. Η πράξη αυτή γίνεται με τον προφανή τρόπο (το πολυώνυμο πολλαπλασιάζεται με τον πολυωνυμικό συντελεστή της λέξης).

Οι τελεστές πράξεων που έχουν οριστεί στην κλάση είναι οι εξής:

- `Word operator *(Word w1, Word w2):` Πολλαπλασιάζει δύο λέξεις μεταξύ τους.
- `Word operator *(Word w, Polynomial p):` Πολλαπλασιάζει μία λέξη με ένα πολυώνυμο
- `Word operator *(Polynomial p, Word w):` Πολλαπλασιάζει μία λέξη με ένα πολυώνυμο. Έχει οριστεί καταχρηστικά.
- `Word operator *(Word w, int[] elem):` Πολλαπλασιάζει μία λέξη με έναν ακέραιο πίνακα τριών θέσεων που αναπαριστά είτε έναν framing γεννήτορα είτε έναν braiding γεννήτορα.

2.3.2.5 Συναρτήσεις υπολογισμού ίχνους

Το σημαντικότερο κομμάτι της κλάσης αυτής είναι οι συναρτήσεις που υπολογίζουν το ίχνος Ocneanu στις άλγεβρες Hecke και το ίχνος Juyumaya στις άλγεβρες Yokonuma-Hecke.

Ο υπολογισμός του ίχνους (οποιοδήποτε και αν είναι αυτό) έχει κατακερματιστεί προγραμματιστικά σε τρία κομμάτια: τον υπολογισμό των απλών περιπτώσεων (π.χ. λέξεις που περιέχουν έναν γεννήτορα μόνο), την ανάλυση μίας λέξης σε πολλές απλούστερες μέσω της τετραγωνικής σχέσης (για οποιονδήποτε ακέραιο εκθέτη) και τέλος τον υπολογισμό του ίχνους κάθε αυτόν. Ο κατακερματισμός αυτός παρουσιάζει αρκετά οφέλη προγραμματιστικά. Για παράδειγμα μπορούμε να αλλάξουμε την τετραγωνική σχέση χωρίς να επηρεάσουμε το υπόλοιπο πρόγραμμα. Επίσης είναι πιο εύκολος ο εντοπισμός πιθανών λογικών λαθών.

Σε αυτή την παράγραφο θα δώσουμε απλή περιγραφή των μεθόδων που έχουν δημιουργηθεί για τον υπολογισμό των δύο ιχνών. Στις επόμενες παραγράφους θα παρουσιαστούν αναλυτικά οι αλγόριθμοι για τα δύο ίχνη.

- `Polynomial TraceHeckeUnit(int[] symbol):` Υπολογίζει το ίχνος Ocnenau για μία λέξη με έναν μόνο γεννήτορα. Ο γεννήτορας δίνεται με τη μορφή πίνακα 2 θέσεων.
- `List<Word> QuadraticRelationHecke(int pos):` Αναλύει τη λέξη εφαρμόζοντας την τετραγωνική σχέση στον γεννήτορα στη θέση `pos`. Η μέθοδος αυτή δεν υπολογίζει τίποτα, παρά μόνο αναλύει τη λέξη στις απλούστερες που θα δημιουργηθούν. Επιστρέφει μία λίστα με τις λέξεις.
- `Polynomial TraceHecke():` Είναι η μέθοδος που υπολογίζει το ίχνος του Ocneanu. Επιστρέφει ένα πολυώνυμο, που είναι το αποτέλεσμα.
- `Polynomial TraceYokonumaHeckeUnit(int framingPower, bool hasBraiding):` Υπολογίζει το ίχνος Juyumaya για απλές περιπτώσεις, δηλαδή για λέξεις που έχουν το πολύ έναν braiding γεννήτορα (με εκθέτη 1) και έναν framing γεννήτορα (με οποιονδήποτε εκθέτη).
- `List<Word> QuadraticRelationYokonumaHecke(int d, int pos):` Αναλύει τη λέξη εφαρμόζοντας την τετραγωνική σχέση της Yokonuma-Hecke στον γεννήτορα στην θέση `pos`. Το ακέραιο όρισμα `d`, είναι ο αριθμος d της $Y_{d,n}(u)$.
- `Polynomial TraceYokonumaHecke(int d):` Υπολογίζει το ίχνος Juyumaya.

2.3.3 Αλγόριθμος υπολογισμού ίχνους Ocneanu στις άλγεβρες Hecke

Αφού έχει περιγραφεί αναλυτικά η δομή του προγράμματος, μπορούμε να δούμε τον αλγόριθμο υπολογισμού του ίχνους Ocneanu για μία λέξη στις άλγεβρες Hecke.

Ο αλγόριθμος θα είναι ένας αναδρομικός αλγόριθμος, δηλαδή θα καλεί τον εαυτό του. Από τον ορισμό του ίχνους Ocneanu, βλέπουμε ότι ο κανόνας (2) μπορεί κάλλιστα να αποτελέσει μία επαγωγική βάση για τον αλγόριθμο. Όμως η λογική αυτή έχει επεκταθεί στον συγκεκριμένο αλγόριθμο. Ως επαγωγική βάση θα χρησιμοποιήσουμε τον υπολογισμού του ίχνους της λέξης g_i^n. Για $n = 0$ προκύπτει ο κανόνας (2) του ίχνους Ocneanu. Για $n > 0$ θα έχουμε:

$$\begin{aligned} tr(g_i^n) &= tr[(q^n - q^{n-1} + \cdots - (-1)^{n+1})g_i + (q^n - q^{n-1} + \cdots - q(-1)^n)] \\ &= (q^n - q^{n-1} + \cdots - (-1)^{n+1})z + (q^n - q^{n-1} + \cdots - q(-1)^n) \end{aligned}$$

Επίσης για $n < 0$ θα έχουμε:

$$tr(g_i^n) = (q^n - q^{n+1} + \cdots - q(-1)^{n+1})z + (q^n - q^{n-1} + \cdots - (-1)^n)$$

Δηλαδή για λέξεις με έναν γεννήτορα έχουμε κατευθείαν το αποτέλεσμα. Αυτές τις περιπτώσεις καλύπτει η μέθοδος `TraceHeckeUnit`.

Έχοντας ορίσει την επαγωγική βάση του αναδρομικού αλγορίθμου μένει να γίνει η αναγωγή σύνθετων λέξεων σε πιο απλές. Το πρώτο βήμα είναι να γίνει απαλοιφή όλων των εκθετών. Δηλαδή να γίνει εφαρμογή της τετραγωνικής σχέσης έτσι ώστε να αναλυθεί η αρχική λέξη σε απλούστερες λέξεις χωρίς εκθέτες. Αυτή η τεχνική ονομάζεται «διαίρει και βασίλευε». Δηλαδή ανάγουμε το πρόβλημα μας σε πολλά μικρότερα προβλήματα τα οποία είναι εύκολο να λυθούνε, και στο τέλος συνδυάζουμε τις λύσεις των μικρών προβλημάτων για να βρεθεί η λύση του αρχικού μας προβλήματος. Στην συγκεκριμένη περίπτωση θα υπολογιστούν τα ίχνη των απλούστερων (χωρίς εκθέτες) λέξεων και θα προστεθούν για να υπολογιστεί το ίχνος της αρχικής λέξης.

Το πρόβλημα μας τώρα έχει αναχθεί στον υπολογισμό του ίχνους λέξεων που δεν περιέχουν εκθέτες. Υπάρχουν δύο περιπτώσεις σε αυτές τις λέξεις: είτε εμφανίζεται μόνο μία φορά γεννήτορας με τον μέγιστο δείκτη – οπότε γίνεται εφαρμογή του κανόνα (3) του ορισμού του ίχνους Ocneanu – είτε υπάρχουν περισσότεροι από ένας που σημαίνει ότι υπάρχει μία γέφυρα στη λέξη ή ότι μπορεί να σχηματιστεί ένας γεννήτορας υψωμένος στο τετράγωνο (Λήμμα §2.3.2.2). Σε περίπτωση που εμφανιστεί τετράγωνο τότε εφαρμόζουμε την τετραγωνική σχέση. Αλλιώς αν έχουμε γέφυρα την αντικαθιστούμε σύμφωνα με την 3-braid relation και συνεχίζουμε τον υπολογισμό μας.

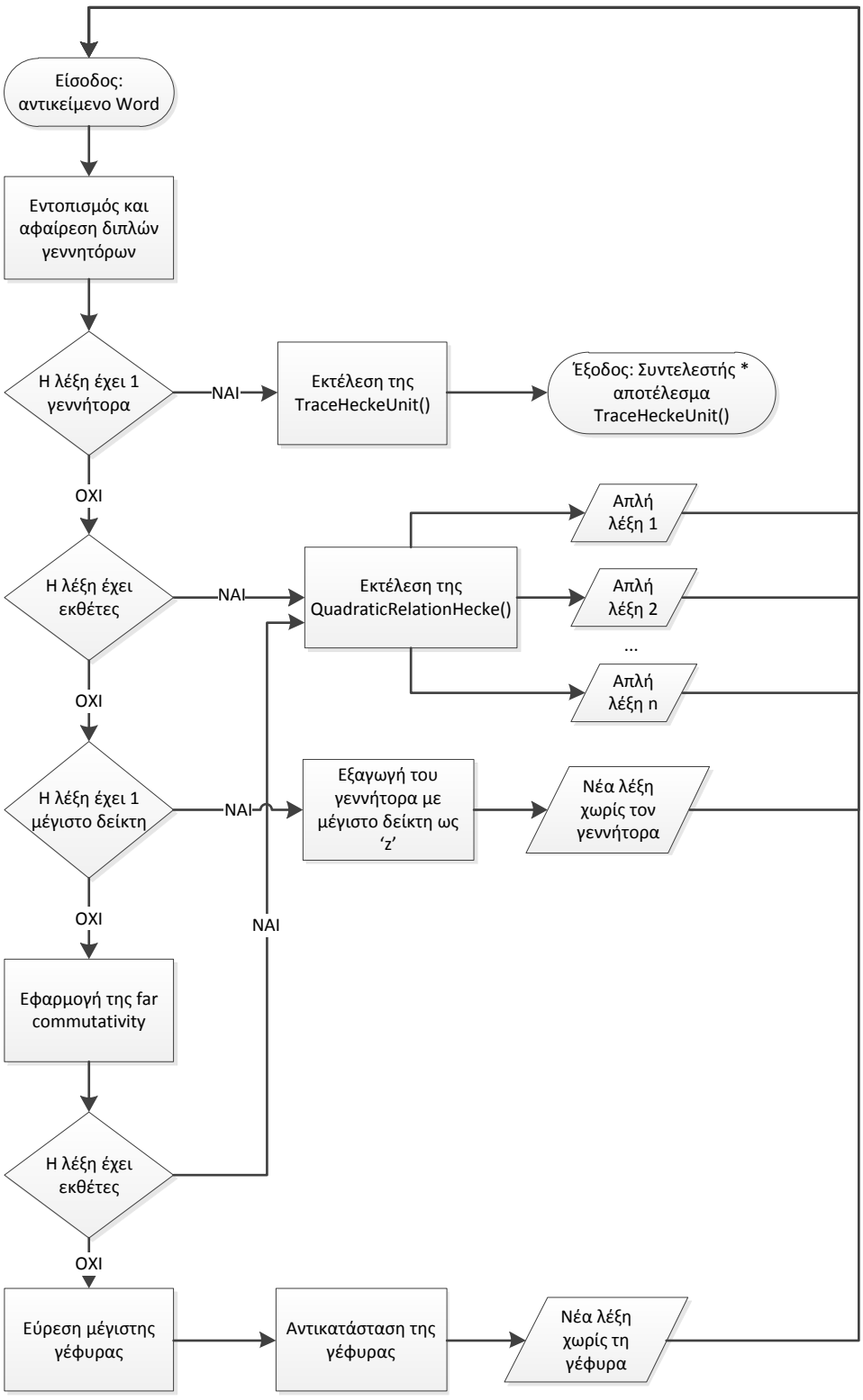
Η δυσκολία του αλγορίθμου είναι να σχηματιστεί η γέφυρα ανάμεσα σε δύο γεννήτορες. Φυσικά εδώ χρησιμοποιούμε το λήμμα της §2.3.2.2 για να δημιουργήσουμε μία πιθανή γέφυρα ανάμεσα σε δύο γεννήτορες. Όμως δοθέντων δύο γεννητόρων υπάρχει δύο πιθανότητες: είτε η γέφυρα να μπορεί να δημιουργηθεί ανάμεσα τους είτε από έξω τους, χρησιμοποιώντας τον κανόνα (1) του ίχνους του Ocneanu. Για παράδειγμα στον υπολογισμό $tr(g_3g_4g_5g_2g_3g_5g_4)$ υπάρχει ήδη μία γέφυρα η οποία δεν φαίνεται καθαρά. Έστω $a = g_3g_4g_5$ και $b = g_2g_3g_5g_4$. Τότε εφαρμόζοντας τον κανόνα της αντιμετάθεσης έχουμε $tr(g_3g_4g_5g_2g_3g_5g_4) = tr(ab) = tr(ba) = tr(g_2g_3g_5g_4g_3g_4g_5)$. Τώρα η γέφυρα φαίνεται πλέον καθαρά.

Δοθέντων δύο γεννητόρων με μέγιστο δείκτη ορίζουμε δύο αποστάσεις: την *εσωτερική* και την *εξωτερική*. Ύστερα βρίσκουμε το ζευγάρι γεννητόρων με μέγιστο δείκτη που έχει την μεγαλύτερη απόσταση και ψάχνουμε σε αυτό για γέφυρα. Επιλέγοντας την μεγαλύτερη απόσταση, υπάρχει περίπτωση ένας από τους δύο γεννήτορες να ενωθεί με έναν άλλον γεννήτορα με μέγιστο δείκτη που ίσως να υπάρχει μεταξύ τους και να δημιουργηθεί τετράγωνο. Τότε εφαρμόζουμε την τετραγωνική σχέση. Αλλιώς το πιθανότερο είναι να βρεθούμε σε γέφυρα.

Ο αλγόριθμος είναι δηλαδή ο εξής:

1. Απαλοιφή γεννητόρων με ίδιο δείκτη που είναι δίπλα.
2. Αν η λέξη έχει έναν γεννήτορα τότε υπολογίζουμε το ίχνος και ο αλγόριθμος τερματίζει.
3. Αν η λέξη περιέχει έναν εκθέτη, επιλέγουμε τον πρώτο και εφαρμόζουμε την τετραγωνική σχέση. Υπολογίζουμε το ίχνος των απλούστερων λέξεων και τα προθέτουμε και ο αλγόριθμος τερματίζει.
4. Αν η λέξη περιέχει μόνο έναν γεννήτορα με τον μέγιστο δείκτη, εφαρμόζουμε τον κανόνα $tr(ag_n) = ztr(a)$ και υπολογίζουμε το ίχνος της λέξης που απομένει.
5. Αφού η λέξη δεν έχει εκθέτες, αλλά περιέχει περισσότερους από έναν γεννήτορες με τον μέγιστο δείκτη, τότε είτε υπάρχει γέφυρα είτε μπορεί να εμφανιστεί τετράγωνο για να εφαρμοστεί η τετραγωνική σχέση. Επιλέγουμε το ζευγάρι γεννητόρων με την μέγιστη απόσταση και εφαρμόζουμε την far commutativity στα στοιχεία που βρίσκονται ανάμεσά τους μέχρι να καταλήξουμε είτε σε γέφυρα είτε σε τετράγωνο.

Το διάγραμμα ροής του αλγορίθμου είναι το παρακάτω:



Όσα χρησιμοποιούνται στον αλγόριθμο για τον υπολογισμό του ίχνους τους Ocneanu στις άλγεβρες Hecke θα χρησιμοποιηθούν και όταν θα υπολογίζουμε το ίχνος Juyumaya στις Yokonuma-Hecke. Το braiding part στην άλγεβρα Yokonuma-Hecke θα απλοποιείται σε απλούστερες λέξεις βάσει της λογικής που αναπτύχθηκε για τις άλγεβρες Hecke. Αυτός ήταν και ο

λόγος που προηγήθηκε πρώτα η ανάπτυξη του αλγορίθμου στις άλγεβρες Hecke.

2.3.4 Αλγόριθμος υπολογισμού ίχνους Juyumaya στις άλγεβρες Yokonuma-Hecke

Ο αλγόριθμος του υπολογισμού του ίχνους Juyumaya στις άλγεβρες Yokonuma-Hecke αποτελεί πρακτικά μία τροποποίηση του αλγορίθμου του ίχνους Ocneanu στις άλγεβρες Hecke.

Η πρώτη σημαντική διαφορά είναι η τετραγωνική σχέση:

$$g_i^2 = 1 + (u-1)e_{d,i} - (u-1)e_{d,i}g_i, \ \forall i, \text{ όπου}$$
$$e_{d,i} = \frac{1}{d}\sum_{m=0}^{d-1} t_i^m t_{i+1}^{-m}$$

Αναλύοντας περαιτέρω την τετραγωνική σχέση έχουμε:

$$g_i^2 = 1 + (u-1)\frac{1}{d}\sum_{m=0}^{d-1} t_i^m t_{i+1}^{-m} - (u-1)\frac{1}{d}\sum_{m=0}^{d-1} t_i^m t_{i+1}^{-m}\, g_i$$

Έστω μία λέξη της μορφής $ag_i^2 b$ σε μία άλγεβρα $Y_{d,n}(u)$. Τότε θα έχουμε:

$$g_i^2 = ab + (u-1)\frac{1}{d}\sum_{m=0}^{d-1} at_i^m t_{i+1}^{-m} b - (u-1)\frac{1}{d}\sum_{m=0}^{d-1} at_i^m t_{i+1}^{-m}\, g_i b$$

Παρατηρούμε δηλαδή ότι εμφανίζονται πολλές νέες λέξεις ($at_i^m t_{i+1}^{-m} b$ και $at_i^m t_{i+1}^{-m} g_i b$) που περιέχουν framing γεννήτορες «στη μέση», οι οποίοι θα πρέπει να μεταφερθούν στην αρχή της λέξης για να γίνουν οι υπολογισμοί.

Δυστυχώς δεν είναι δυνατόν να κρατήσουμε τα $e_{d,i}$ χωρίς να τα αντικαταστήσουμε. Ο υπολογισμός θα γινόταν αρκετά πιο εύκολος αν χωρίζαμε μία λέξη σε framing part, braiding part και ένα "$e_{d,i}$" part. Αλλά λόγω των σχέσεων (viii) και (ix) στον ορισμό του ίχνους Juyumaya κάτι τέτοιο είναι αδύνατον. Οπότε η μόνη επιλογή είναι η άμεση αντικατάσταση τους.

Για την τετραγωνική σχέση δημιουργήθηκε η μέθοδος `QuadraticRelationYokonumaHecke`. Η μέθοδος αυτή εφαρμόζει την τετραγωνική σχέση σε έναν γεννήτορα με εκθέτη n και χωρίζει τη λέξη σε $2d-1$ νέες απλούστερες λέξεις. Επιστρέφει τις λέξεις μέσα σε μία λίστα, έχοντας ξεχωρίσει σε κάθε μία το framing part από το braiding part.

Η δεύτερη διαφορά είναι οι κανόνες του ίχνους στις άλγεβρες Yokonuma-Hecke. Ο αλγόριθμος στις Yokonuma-Hecke θα χρειαστεί να ψάχνει να βρίσκει γεννήτορες με μέγιστο δείκτη ξεχωριστά στο framing part και ξεχωριστά στο braiding part. Όμως υπάρχουν πολλές περιπτώσεις:

1. Υπάρχει μόνο t_{n+1}^m χωρίς να εμφανίζεται ο γεννήτορας g_n. Τότε απλά εφαρμόζεται ο κανόνας 4.
2. Υπάρχει μόνο ο γεννήτορας g_n ο οποίος εμφανίζεται μόνο μία φορά. Τότε απλά εφαρμόζεται ο κανόνας 3.
3. Υπάρχει ο γεννήτορας g_n ο οποίος όμως εμφανίζεται περισσότερες από μία φορές. Τότε ο αλγόριθμος θα πρέπει να ψάχνει για γέφυρες με τον ίδιο τρόπο που το κάνει και ο αλγόριθμος στις άλγεβρες Hecke.
4. Υπάρχει ο γεννήτορας t_{n+1}^m και ο γεννήτορας g_n που εμφανίζεται μόνο μία φορά. Τότε μεταφέρεται ο t_{n+1}^m δεξιά του g_n εφαρμόζοντας τον κανόνα $t_{n+1}^m g_n = g_n t_n^m$. Τώρα πλέον εμφανίζεται μόνο ο g_n και μπορεί να εφαρμοστεί ο κανόνας 3 και ο κανόνας 4.

Δηλαδή ενώ στον υπολογισμό του ίχνους Ocneanu απλά ελέγχαμε αν υπάρχει ένας γεννήτορας με μέγιστο δείκτη, στην περίπτωση του ίχνους Juyumaya θα χρειαστεί να γίνουν περισσότεροι έλεγχοι.

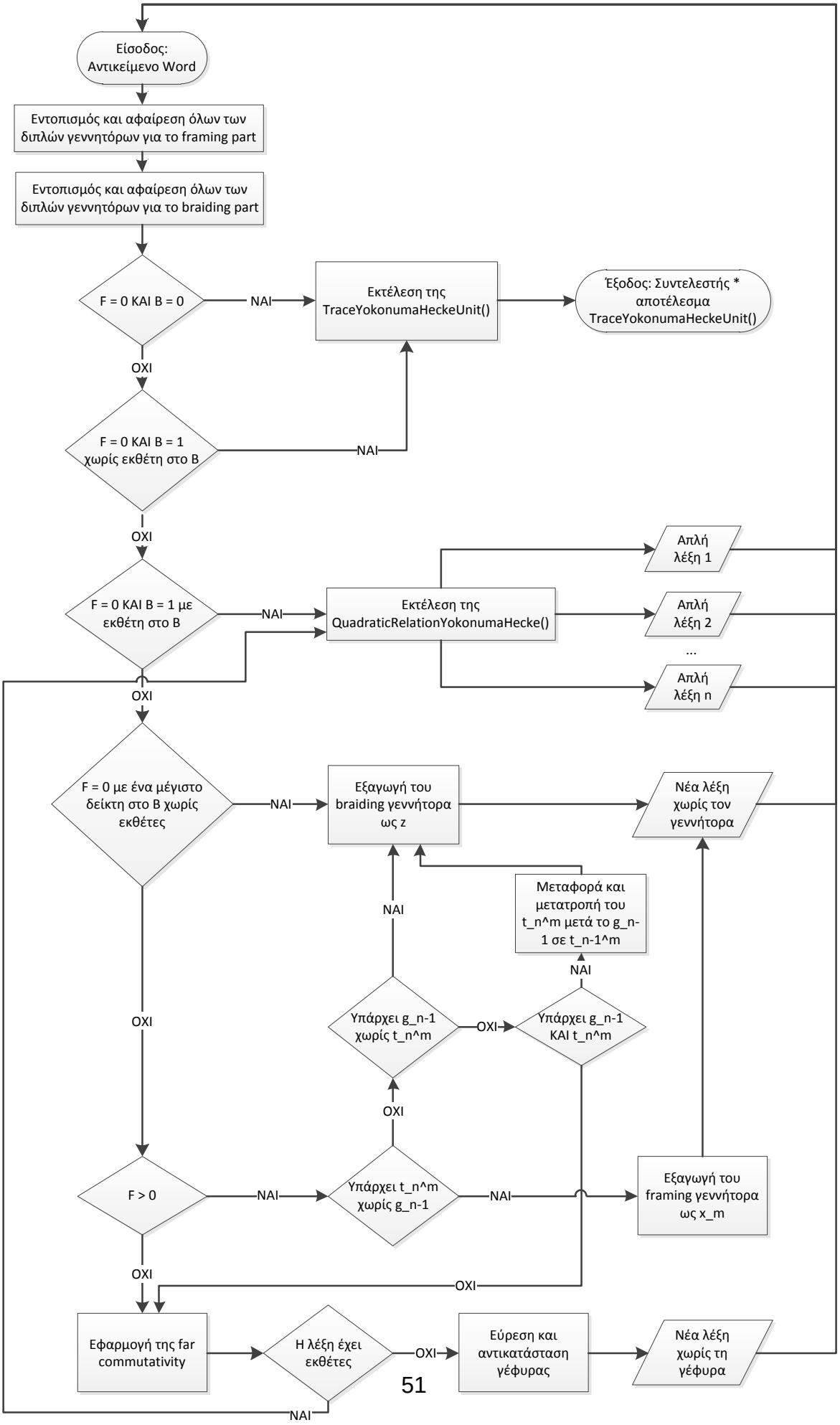
Συνοπτικά ο αλγόριθμος ακολουθεί τα εξής βήματα:

1. Απαλοιφή braiding γεννητόρων με ίδιο δείκτη που είναι δίπλα και απαλοιφή framing γεννητόρων με εκθέτη ένα πολλαπλάσιο του d.
2. Έλεγχος για το αν η λέξη δεν έχει ούτε framing part ούτε braiding part. Τότε ο αλγόριθμος επιστρέφει τον πολυωνυμικό συντελεστή της λέξης.
3. Έλεγχος για το αν η λέξη δεν έχει framing part και έχει έναν braiding γεννήτορα χωρίς κάποιον εκθέτη. Τότε υπολογίζεται το ίχνος από την `TraceYokonumaHeckeUnit`.
4. Έλεγχος για το αν η λέξη δεν έχει framing part και έχει έναν braiding γεννήτορα με κάποιον εκθέτη. Τότε εφαρμόζεται η τετραγωνική σχέση μέσω της QuadraticRelationYokonumaHecke η οποία επιστρέφει $2d + 1$ απλούστερες λέξεις. Υπολογίζεται το ίχνος σε αυτές τις λέξεις και προσθέτοντας τα αποτελέσματα υπολογίζεται το ίχνος της αρχικής λέξης.
5. Έλεγχος για το αν η λέξη δεν έχει framing part και περιέχει στο braiding part μόνο έναν γεννήτορα με μέγιστο δείκτη. Τότε εφαρμόζεται ο κανόνας (3) από τον ορισμό του ίχνους, και υπολογίζεται αναδρομικά το ίχνος στην λέξη που απομένει.
6. Έλεγχος για το αν η λέξη έχει framing part. Αν ναι ο αλγόριθμος συνεχίζει στο 7. Αν όχι συνεχίζει στο 10.
7. Έλεγχος για το αν υπάρχει ο γεννήτορας t_{n+1}^m χωρίς να υπάρχει ο g_n. Αν ναι τότε εφαρμόζουμε τον κανόνα 3.
8. Έλεγχος για το αν υπάρχει ο γεννήτορας g_n μία μόνο φορά χωρίς να υπάρχει ο t_{n+1}^m. Αν ναι εφαρμόζεται ο κανόνας 4 του ίχνους.
9. Έλεγχος για τον αν υπάρχει ο γεννήτορας t_{n+1}^m και ο γεννήτορας g_n μόνο μία φορά. Τότε μετακινούμε τον t_{n+1}^m δεξιά του g_n έτσι ώστε να

εφαρμοστεί η σχέση $t_{n+1}^{m} g_n = g_n t_n^{m}$. Τότε εφαρμόζεται ο κανόνας 3 και έπειτα ο κανόνας 4 του ίχνους.

10. Αφού έχουν καλυφθεί όλες οι πιθανές εύκολες περιπτώσεις, ο αλγόριθμος ελέγχει για γέφυρα. Αν βρει την αντικαθιστά. Αν εμφανιστεί τετράγωνο κατά την διαδικασία τότε εφαρμόζεται πάλι η τετραγωνική σχέση.

Το διάγραμμα ροής του αλγόριθμου περιγράφει όλη την διαδικασία πολύ απλά:


Είσοδος: Αντικείμενο Word
Εντοπισμός και αφαίρεση όλων των διπλών γεννητόρων για το framing part
Εντοπισμός και αφαίρεση όλων των διπλών γεννητόρων για το braiding part
F = 0 ΚΑΙ B = 0
ΝΑΙ
Εκτέλεση της TraceYokonumaHeckeUnit()
Έξοδος: Συντελεστής * αποτέλεσμα TraceYokonumaHeckeUnit()
ΟΧΙ
F = 0 ΚΑΙ B = 1 χωρίς εκθέτη στο B
ΝΑΙ
ΟΧΙ
F = 0 ΚΑΙ B = 1 με εκθέτη στο B
ΝΑΙ
Εκτέλεση της QuadraticRelationYokonumaHecke()
Απλή λέξη 1
Απλή λέξη 2
...
Απλή λέξη n
ΟΧΙ
F = 0 με ένα μέγιστο δείκτη στο B χωρίς εκθέτες
ΝΑΙ
Εξαγωγή του braiding γεννήτορα ως z
Νέα λέξη χωρίς τον γεννήτορα
Μεταφορά και μετατροπή του t_n^m μετά το g_n-1 σε t_n-1^m
ΝΑΙ
ΝΑΙ
ΟΧΙ
Υπάρχει g_n-1 χωρίς t_n^m
ΟΧΙ
Υπάρχει g_n-1 ΚΑΙ t_n^m
ΟΧΙ
F > 0
ΝΑΙ
Υπάρχει t_n^m χωρίς g_n-1
ΝΑΙ
Εξαγωγή του framing γεννήτορα ως x_m
ΟΧΙ
ΟΧΙ
Εφαρμογή της far commutativity
Η λέξη έχει εκθέτες
ΟΧΙ
Εύρεση και αντικατάσταση γέφυρας
Νέα λέξη χωρίς τη γέφυρα
ΝΑΙ

2.4 Αποτελέσματα του προγράμματος

Το πρόγραμμα Braids ελέγχθηκε με διάφορα παραδείγματα κατά την διάρκεια ανάπτυξης του, έτσι ώστε να ελεγχθεί η ορθότητά του. Πέρα από αυτό όμως υπολογίστηκαν και ίχνη που χωρίς την ύπαρξη του προγράμματος θα ήταν αδύνατον να υπολογιστούν.

Οι πρώτες λέξεις των οποίων τα ίχνη Ocneanu υπολογίστηκαν είναι τα ζευγάρια λέξεων $\sigma_1^{2p+1}\sigma_2^{2q}\sigma_1^{2r}\sigma_2^{-1}$ και $\sigma_1^{2p+1}\sigma_2^{-1}\sigma_1^{2r}\sigma_2^{2q}$ με $p,q,r>1,\ q\neq r$ [OrSh]. Οι κόμβοι που αντιστοιχούν σε αυτές τις λέξεις είναι ισοτοπικοί ως κλασσικοί κόμβοι αλλά είναι μη ισοτοπικοί ως transverse κόμβοι. Το πρόγραμμα υπολόγισε τα ίχνη Ocneanu για τα παραπάνω ζευγάρια λέξεων για $2\leq\ p,q,r\leq 10$, δηλαδή για 448 ζευγάρια λέξεων, και τα συνέκρινε μεταξύ τους. Το αποτέλεσμα είναι ότι οι λέξεις σε κάθε ένα από τα ζευγάρια έχουν το ίδιο ίχνος Ocneanu, πράγμα που επιβεβαιώνει ότι το πρόγραμμα είναι ορθό.

Αξίζει να σημειωθεί ότι τα ίχνη Ocneanu των 448 ζευγαριών λέξεων υπολογίστηκαν σε περίπου 7 λεπτά. Δηλαδή υπολογιζόταν ένα ίχνος ανά μισό δευτερόλεπτο.

Όμως τα παραπάνω ζευγάρια λέξεων, αν προστεθεί μπροστά τους ένας framing γεννήτορας, έχουν διαφορετικά ίχνη Jyuymaya. Δηλαδή υπολογίστηκαν τα ίχνη των λέξεων $t_1^k\sigma_1^{2p+1}\sigma_2^{2q}\sigma_1^{2r}\sigma_2^{-1}$ και $t_1^k\sigma_1^{2p+1}\sigma_2^{-1}\sigma_1^{2r}\sigma_2^{2q}$ για διάφορες τιμές του k και όντως βρέθηκε ότι τα ίχνη είναι διαφορετικά. Φυσικά οι υπολογισμοί σε αυτή την περίπτωση ήταν πιο αργοί λόγω της πολυπλοκότητας του αλγορίθμου στις άλγεβρες Yokonuma-Hecke (περίπου 30 δευτερόλεπτα ανά ίχνος).

Τέλος υπολογίστηκαν και τα ίχνη Juyumaya των ζευγαριών λέξεων $\sigma_3\sigma_2^{-2}\sigma_3^{2a+2}\sigma_2\sigma_3^{-1}\sigma_1^{-1}\sigma_2\sigma_1^{2b+2}$ και $\sigma_3\sigma_2^{-2}\sigma_3^{2a+2}\sigma_2\sigma_3^{-1}\sigma_1^{2b+2}\sigma_2\sigma_1^{-1}$. Αυτό αποτελεί breakthrough αφού ο υπολογισμός ήταν αδύνατος προηγουμένως. Για διάφορες τιμές του a και του b, βρέθηκε ότι τα ίχνη Juyumaya των δύο αυτών λέξεων διαφέρουν, όπως περιμέναμε αφού οι transverse κόμβοι που αντιστοιχούν σε αυτές τις δύο λέξεις είναι μη ισοτοπικοί.

2.5 Μελλοντική χρήση και επέκταση του προγράμματος

Το πρόγραμμα αυτό μπορεί να χρησιμοποιηθεί μελλοντικά και για άλλους υπολογισμούς. Για παράδειγμα, με μικρές τροποποιήσεις, είναι εφικτό να υπολογιστεί το ίχνος για τις άλγεβρες Yokonuma-Temperley-Lieb, οι οποίες ορίζονται ως τις άλγεβρες Yokonuma-Hecke με μία παραπάνω σχέση. Ένας τέτοιος υπολογισμός θα ήταν μεγάλο βήμα, καθώς η πολυπλοκότητα των υπολογισμών στις άλγεβρες Yokonuma-Temperley-Lieb αυξάνεται λόγω της επιπλέον σχέσης.

Επίσης θα μπορούσαμε να βελτιστοποιήσουμε τον αλγόριθμο έτσι ώστε να γίνει πιο γρήγορος. Στον αλγόριθμο εμφανίζονται πολλοί πολλαπλασιασμοί πολυωνύμων και ο πολλαπλασιασμός πολυωνύμων έχει πολυπλοκότητα $O(nm)$ με την τρέχουσα υλοποίηση. Είναι γνωστό ότι για πολυώνυμα μίας μεταβλητής μπορεί να εκτελεστεί ο πολλαπλασιασμός πιο γρήγορα χρησιμοποιώντας τις μιγαδικές ρίζες της μονάδας και ο διακριτός μετασχηματισμός Fourier (Discrete Fourier Transform – DFT). Το πρόβλημα είναι ότι στο πρόγραμμα έχουμε μιγαδικά πολυώνυμα Laurent (οπότε έχουμε και αρνητικούς εκθέτες) και ότι έχουμε πολλές μεταβλητές. Ενδεχομένως, ίσως να είναι εφικτό να βρεθεί μία μέθοδος που θα βασίζεται στον DFT για να πολλαπλασιάσουμε πολυώνυμα Laurent πολλών μεταβλητών.

Μία άλλη βελτίωση που μπορεί να γίνει είναι να μετατραπεί ο αλγόριθμος σε παράλληλο αλγόριθμο ώστε να μπορεί το πρόγραμμα να εκμεταλλευτεί πολλαπλούς επεξεργαστές ή επεξεργαστικούς πυρήνες. Η τετραγωνική σχέση στις άλγεβρες Yokonuma-Hecke αναλύει τη λέξη σε $2d + 1$ απλούστερες, οπότε θα μπορούσαμε να υπολογίσουμε τα ίχνη αυτών των λέξεων παράλληλα και όχι σειριακά. Κατά την ανάπτυξη του προγράμματος δοκιμάστηκε μία τέτοια τεχνική και το πρόγραμμα ήταν περίπου 2.7 φορές γρηγορότερο (η δοκιμή έγινε σε επεξεργαστή με 4 πυρήνες και 2-way SMT).

Τέλος, μία άλλη βελτίωση που μπορεί να γίνει είναι να γραφεί αλγόριθμος που να παραγοντοποιεί ένα μιγαδικό πολυώνυμο Laurent πολλών μεταβλητών. Αυτό θα ήταν ιδιαίτερα χρήσιμο για να μπορούν να ερμηνευθούν τα αποτελέσματα του προγράμματος πιο εύκολα.

3 Κώδικας προγράμματος

3.1 PolyLib

3.1.1 MultiVarTerm.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Numerics;

namespace PolyLib
{
    /// <summary>
    /// Models a single term of the polynomial in the field of real numbers
    /// </summary>
    public class MultiVarTerm : IEquatable<MultiVarTerm>,
IComparable<MultiVarTerm>
    {
        public SortedDictionary<char, int> Variables { get; set; } //Saves the
variables and their exponents
        public double Coefficient { get; set; } //The coefficient of the term

        #region Constructors
        /// <summary>
        /// Creates a new MultiVarTerm which equals to the given <paramref
name="coeff"/>
        /// </summary>
        /// <param name="coeff">The coefficient of the MultiVarTerm</param>
        public MultiVarTerm(double coeff = 1)
        {
            Variables = new SortedDictionary<char, int>();
            Coefficient = coeff;
        }

        /// <summary>
        /// Creates a new MultiVarTerm from a given SortedDictionary and a
given coefficient
        /// </summary>
        /// <param name="var">A SortedDictionary object containing the
variables and their exponents</param>
        /// <param name="coeff">The coefficient of the MultiVarTerm</param>
        public MultiVarTerm(SortedDictionary<char, int> var, double coeff = 1)
            : this(coeff)
        {
            if (coeff != 0)
                //Deep copy of dictionary
                foreach (char k in var.Keys)
                    if (var[k] != 0)
                        Variables.Add(k, var[k]);
            else
                Variables = new SortedDictionary<char, int>();
        }

        /// <summary>
```

```
        /// Creates a new MultiVarTerm from an array of tuples and a given
coefficient
        /// </summary>
        /// <param name="coeff">The coefficient of the MultiVarTerm</param>
        /// <param name="exp">An array of tuples containing the variables and
their exponents</param>
        public MultiVarTerm(double coeff = 1, params Tuple<char, int>[] exp)
            : this(coeff)
        {
            if (coeff != 0)
                for (int i = 0; i < exp.Length; i++)
                    if (exp[i].Item2 != 0)
                        Variables.Add(exp[i].Item1, exp[i].Item2);
            else
                Variables = new SortedDictionary<char, int>();
        }

        /// <summary>
        /// Creates a new MultiVarTerm from a given coefficient, a given
character (variable) and a given exponent
        /// </summary>
        /// <param name="coeff">The coefficient</param>
        /// <param name="c">The variable</param>
        /// <param name="exp">The exponent</param>
        public MultiVarTerm(double coeff, char c, int exp)
            : this(coeff, new Tuple<char, int>(c, exp)) { }
        #endregion

        #region Interfaces
        /// <summary>
        /// Checks if a given MultiVarTerm is equal to the current MultiVarTerm
        /// </summary>
        /// <param name="t">The MultiVarTerm to be compared to the current
MultiVarTerm</param>
        /// <returns>A boolean value determining whether the given MultiVarTerm
equals the current MultiVarTerm</returns>
        public Boolean Equals(MultiVarTerm t)
        {
            return CompareTo(t) == 0;
        }

        /// <summary>
        /// Compares the current MultiVarTerm with an other one, based on
lexicographical sorting but ignores coefficients
        /// </summary>
        /// <param name="t">MultiVarTerm</param>
        /// <returns>An integer value that indicates the relative order of the
objects being compared.</returns>
        public int CompareTo(MultiVarTerm t)
        {
            int a = ToStringOnlyVar().CompareTo(t.ToStringOnlyVar());
            if (a == 0)
                return ToStringOnlyExp().CompareTo(t.ToStringOnlyExp());
            else
                return a;

        }
        #endregion

        #region Operators
        /// <summary>
        /// Multiplies two MultiVarTerm objects
```

```
        /// </summary>
        /// <param name="t1">The first MultiVarTerm object</param>
        /// <param name="t2">The second MultiVarTerm object</param>
        /// <returns>The product of the two MultiVarTerm objects</returns>
        public static MultiVarTerm operator *(MultiVarTerm t1, MultiVarTerm t2)
        {
            if (t1.Variables.Count == 0) //t1 = double number
                return new MultiVarTerm(t2.Variables, t1.Coefficient *
t2.Coefficient);
            else if (t2.Variables.Count == 0) //t2 = double number
                return new MultiVarTerm(t1.Variables, t1.Coefficient *
t2.Coefficient);
            else
            {
                MultiVarTerm result = new MultiVarTerm(t1.Variables,
t1.Coefficient * t2.Coefficient); //Creates a new MultiVarTerm based on t1
                //Copies the variables and the exponents of t2 into result
                foreach (char c in t2.Variables.Keys)
                {
                    if (result.Variables.ContainsKey(c))
                        result.Variables[c] += t2.Variables[c];
                    else
                        result.Variables.Add(c, t2.Variables[c]);
                }
                return result;
            }
        }

        /// <summary>
        /// Multiplies a MultiVarTerm with a double number
        /// </summary>
        /// <param name="t">The MultiVarTerm object</param>
        /// <param name="c">The double number</param>
        /// <returns>The prodcut of the MultiVarTerm with the number</returns>
        public static MultiVarTerm operator *(MultiVarTerm t, double c)
        {
            return new MultiVarTerm(t.Variables, t.Coefficient * c);
        }

        /// <summary>
        /// Multiplies a MultiVarTerm with a double number
        /// </summary>
        /// <param name="t">The MultiVarTerm object</param>
        /// <param name="c">The double number</param>
        /// <returns>The prodcut of the MultiVarTerm with the number</returns>
        public static MultiVarTerm operator *(double c, MultiVarTerm t)
        {
            return t * c;
        }

        /// <summary>
        /// Calculates the opposite of a MultiVarTerm
        /// </summary>
        /// <param name="t">The MultiVarTerm object</param>
        /// <returns>The opposite MultiVarTerm object</returns>
        public static MultiVarTerm operator -(MultiVarTerm t)
        {
            return new MultiVarTerm(t.Variables, -t.Coefficient);
        }
        #endregion

        #region Other Methods
```

```
        /// <summary>
        /// Adds a number to the coefficient
        /// </summary>
        /// <param name="c">The number to be added</param>
        public void AddToCoeff(double c)
        {
            Coefficient += c;
        }

        /// <summary>
        /// Returns the MultiVarTerm in the form of a string
        /// </summary>
        /// <returns>A string</returns>
        public override string ToString()
        {
            StringBuilder str = new StringBuilder();

            if (Coefficient < 0)
            {
                if (Coefficient == -1 && Variables.Count == 0)
                {
                    str.Append(" - 1");
                    return str.ToString();
                }
                else if (Coefficient == -1 && Variables.Count > 0)
                {
                    str.Append(" -");
                }
                else
                    str.Append(" - " + -Coefficient);
            }
            else if (Coefficient > 0)
            {
                if (!(Coefficient == 1 && Variables.Count != 0))
                    str.Append(Coefficient);
            }


            foreach (KeyValuePair<char, int> k in Variables)
            {
                if (str.Length > 0) { str.Append(" "); }
                str.Append(k.Key.ToString());
                if (k.Value != 1) { str.Append("^" + k.Value); }
            }
            return str.ToString();
        }

        /// <summary>
        /// Returns the MultiVarTerm in the form of a string (ignores the
coefficients)
        /// </summary>
        /// <returns>A string</returns>
        public string ToStringOnlyVar()
        {
            StringBuilder str = new StringBuilder();
            foreach (KeyValuePair<char, int> k in Variables)
            {
                str.Append(k.Key.ToString());
            }
            return str.ToString();
        }
```

```
        /// <summary>
        /// Returns the MultiVarTerm in the form of a string (ignores the
coefficients)
        /// </summary>
        /// <returns>A string</returns>
        public string ToStringOnlyExp()
        {
            StringBuilder str = new StringBuilder();
            foreach (KeyValuePair<char, int> k in Variables)
            {
                str.Append(k.Value);
            }
            return str.ToString();
        }

        /// <summary>
        /// Creates a deep copy
        /// </summary>
        /// <returns>A new MultiVarTerm</returns>
        public MultiVarTerm Copy()
        {
            MultiVarTerm copy = new MultiVarTerm(this.Coefficient);

            foreach (char k in this.Variables.Keys)
                copy.Variables.Add(k, this.Variables[k]);

            return copy;
        }
        #endregion
    }
}
```

3.1.2 Polynomial.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PolyLib
{
    /// <summary>
    /// Models a Laurent polynomial with coefficients in the field of real
numbers
    /// </summary>
    public class Polynomial :IEquatable<Polynomial>, IList<MultiVarTerm>
    {
        #region Constructors
        /// <summary>
        /// Creates an empty Polynomial
        /// </summary>
        public Polynomial()
        {
            _terms = new List<MultiVarTerm>();
            _isReadOnly = false;
        }

        /// <summary>
        /// Creates a new Polynomial that is equal to a double number
        /// </summary>
        /// <param name="c">The double number</param>
```

```
        public Polynomial(double c) : this()
        {
            _terms.Add(new MultiVarTerm(c));
        }

        /// <summary>
        /// Creates a new Polynomial based on an array of MultiVarTerm objects
        /// </summary>
        /// <param name="terms">The array of MultiVarTerm objects</param>
        public Polynomial(params MultiVarTerm[] terms) : this()
        {
            for (int i = 0; i < terms.Length; i++)
                Add(terms[i].Copy());
        }

        /// <summary>
        /// Creates a new Polynomial based on a collection of MultiVarTerm
objects
        /// </summary>
        /// <param name="terms">The collection of the MultiVarTerm
objects</param>
        public Polynomial(IEnumerable<MultiVarTerm> terms) : this()
        {
            foreach (MultiVarTerm item in terms)
                Add(item);
        }
        #endregion

        #region ICollection
        private bool _isReadOnly;

        /// <summary>
        /// Gets the number of MultiVarTerm elements contained in the
Polynomial.
        /// </summary>
        public int Count
        {
            get { return _terms.Count; }

        }

        /// <summary>
        /// Gets a value indicating whether the Polynomial is read-only.
        /// </summary>
        public bool IsReadOnly
        {
            get { return _isReadOnly; }
        }

        /// <summary>
        /// Adds a MultiVarTerm to the Polynomial.
        /// </summary>
        /// <param name="item">The MultiVarTerm object to be added to the
Polynomial.</param>
        public void Add(MultiVarTerm item)
        {
            if (IsReadOnly) { throw new NotSupportedException("The Polynomial
is read-only."); }
            if (item.Coefficient != 0)
            {
                if (_terms.Contains(item)) //The term is already in the
Polynomial, so it does not need to be added, just add the coefficients
```

```
                {
                    int index = _terms.IndexOf(item);
                    if (_terms[index].Coefficient == -item.Coefficient)
                        _terms.RemoveAt(index);
                    else
                        _terms[index].Coefficient += item.Coefficient;
                }
                else
                {
                    _terms.Add(item);
                }
            }
        }

        /// <summary>
        /// Removes all MultiVarTerm objects from the Polynomial.
        /// </summary>
        public void Clear()
        {
            if (IsReadOnly) { throw new NotSupportedException("The Polynomial
is read-only."); }
            _terms.Clear();
        }

        /// <summary>
        /// Determines whether the Polynomial contains a specific MultiVarTerm
(ignores the coefficient).
        /// </summary>
        /// <param name="item">The MultiVarTerm object to locate in the
Polynomial.</param>
        /// <returns>A boolean value determining whether <paramref
name="item"/> is contained in the Polynomial</returns>
        public bool Contains(MultiVarTerm item)
        {
            return _terms.Contains(item);
        }

        /// <summary>
        /// Copies the elements of the Polynomial to an Array, starting at a
particular Array index.
        /// </summary>
        /// <param name="array">The one-dimensional Array that is the
destination of the elements copied from Polynomial. The Array must have zero-
based indexing.</param>
        /// <param name="arrayIndex">The zero-based index in array at which
copying begins.</param>
        public void CopyTo(MultiVarTerm[] array, int arrayIndex)
        {
            if (array.Length == 0) { throw new ArgumentNullException("array",
"array is null"); }
            if (arrayIndex < 0) { throw new
ArgumentOutOfRangeException("arrayIndex", arrayIndex, "arrayIndex is less than
0"); }
            if (array.Length - arrayIndex < _terms.Count) { throw new
ArgumentException("The number of elements in the source Polynomial is greater
than the available space from arrayIndex to the end of the destination
array."); }

            for (int i = 0; i < _terms.Count; i++)
                array[arrayIndex + i] = _terms[i].Copy();
        }
```

```
        /// <summary>
        /// Removes a specific MultiVarTerm from the Polynomial.
        /// </summary>
        /// <param name="item">The object to remove from the
Polynomial.</param>
        /// <returns>true if item was successfully removed from the Polynomial;
otherwise, false. This method also returns false if item is not found in the
original Polynomial.</returns>
        public bool Remove(MultiVarTerm item)
        {
            if (IsReadOnly) { throw new NotSupportedException("The Polynomial
is read-only."); }
            bool rem = _terms.Remove(item);
            return rem;
        }
        #endregion

        #region IEnumerable
        /// <summary>
        /// Gets an Enumerator for the Polynomial object
        /// </summary>
        /// <returns>An IEnumerator<MultiVarTerm>
object</MultiVarTerm></returns>
        public IEnumerator<MultiVarTerm> GetEnumerator()
        {
            return _terms.GetEnumerator();
        }

        /// <summary>
        /// Gets an Enumerator for the Polynomial object
        /// </summary>
        /// <returns>An IEnumerator object</MultiVarTerm></returns>
        IEnumerator IEnumerable.GetEnumerator()
        {
            return (IEnumerator) GetEnumerator();
        }
        #endregion

        #region IList
        private List<MultiVarTerm> _terms;

        /// <summary>
        /// Gets or sets the MultiVarTerm at the specified index.
        /// </summary>
        /// <param name="index">The zero-based MultiVarTerm of the element to
get or set.</param>
        /// <returns>The MultiVarTerm at the specified index.</returns>
        public MultiVarTerm this[int index]
        {
            get { return _terms[index]; }
            set { _terms[index] = value; }
        }

        /// <summary>
        /// Determines the index of a specific item in the Polynomial.
        /// </summary>
        /// <param name="t">The MultiVarTerm object to locate in the
Polynomial.</param>
        /// <returns>The index of MultiVarTerm if found in the Polynomial;
otherwise, -1.</returns>
        public int IndexOf(MultiVarTerm item)
        {
```

```
            return _terms.IndexOf(item);
        }

        /// <summary>
        /// Adds a MultiVarTerm to the Polynomia
        /// </summary>
        /// <param name="index">The zero-based index at which item should be
inserted</param>
        /// <param name="item">The object to insert into the Polynomial</param>
        public void Insert(int index, MultiVarTerm item)
        {
            _terms.Insert(index, item);
        }

        /// <summary>
        /// Removes the MultiVarTerm at the specified index.
        /// </summary>
        /// <param name="index">The zero-based index of the MultiVarTerm to
remove.</param>
        public void RemoveAt(int index)
        {
            if (IsReadOnly) { throw new NotSupportedException("The Polynomial
is read-only."); }
            _terms.RemoveAt(index);
        }
        #endregion

        #region IEquatable
        /// <summary>
        /// Checks if the current Polynomial is equal to a given Polynomial
        /// </summary>
        /// <param name="p">A Polynomial</param>
        /// <returns>A boolean value determining whether the current Polynomial
is equal to <paramref name="p"/></returns>
        public Boolean Equals(Polynomial p)
        {
            return (this.ToString() == p.ToString());
        }
        #endregion

        #region Operators
        /// <summary>
        /// Multiplies a Polynomial with a Polynomial
        /// </summary>
        /// <param name="p1">The first Polynomial</param>
        /// <param name="p2">The second Polynomial</param>
        /// <returns>The product of the two objects</returns>
        public static Polynomial operator *(Polynomial p1, Polynomial p2)
        {
            if (p1._terms.Count == 0) { return p2.Copy(); }
            if (p2._terms.Count == 0) { return p1.Copy(); }
            List<MultiVarTerm> temp =  new List<MultiVarTerm>();
            foreach (MultiVarTerm m1 in p1._terms)
                foreach (MultiVarTerm m2 in p2._terms)
                    temp.Add(m1*m2);
            return new Polynomial(temp);
        }

        /// <summary>
        /// Multiplies a Polynomial with a MultiVarTerm
        /// </summary>
        /// <param name="p">The Polynomial</param>
```

```
/// <param name="t">The MultiVarTerm</param>
/// <returns>The product of the two objects</returns>
public static Polynomial operator *(Polynomial p, MultiVarTerm t)
{
    List<MultiVarTerm> temp = new List<MultiVarTerm>();
    foreach (MultiVarTerm m1 in p._terms)
            temp.Add(m1 * t);
    return new Polynomial(temp);
}

/// <summary>
/// Multiplies a Polynomial with a MultiVarTerm
/// </summary>
/// <param name="p">The Polynomial</param>
/// <param name="t">The MultiVarTerm</param>
/// <returns>The product of the two objects</returns>
public static Polynomial operator *(MultiVarTerm t, Polynomial p)
{
    return p * t;
}

/// <summary>
/// Multiplies a Polynomial with a double number
/// </summary>
/// <param name="p">The Polynomial</param>
/// <param name="c">The double number</param>
/// <returns>The product of the two objects</returns>
public static Polynomial operator *(Polynomial p, double c)
{
    List<MultiVarTerm> temp = new List<MultiVarTerm>();
    foreach (MultiVarTerm m1 in p._terms)
        temp.Add(m1 * c);
    return new Polynomial(temp);
}

/// <summary>
/// Multiplies a Polynomial with a double number
/// </summary>
/// <param name="p">The Polynomial</param>
/// <param name="c">The double number</param>
/// <returns>The product of the two objects</returns>
public static Polynomial operator *(double c, Polynomial p)
{
    return p * c;
}

/// <summary>
/// Adds a Polynomial with a Polynomial
/// </summary>
/// <param name="p1">The first Polynomial</param>
/// <param name="p2">The second Polynomial</param>
/// <returns>The sum of the two objects</returns>
public static Polynomial operator +(Polynomial p1, Polynomial p2)
{
    List<MultiVarTerm> temp = new List<MultiVarTerm>();
    foreach (MultiVarTerm m in p1._terms) //Adds the terms of p1
        temp.Add(m.Copy());
    foreach (MultiVarTerm m in p2._terms) //Adds the terms of p2
        temp.Add(m.Copy());
    return new Polynomial(temp);
}
```

```
        /// <summary>
        /// Adds a Polynomial with a MultiVarTerm
        /// </summary>
        /// <param name="p">The Polynomial</param>
        /// <param name="t">The MultiVarTerm</param>
        /// <returns>The sum of the two objects</returns>
        public static Polynomial operator +(Polynomial p, MultiVarTerm t)
        {
            List<MultiVarTerm> temp = new List<MultiVarTerm>();
            foreach (MultiVarTerm m in p._terms)
                temp.Add(m.Copy());
            temp.Add(t.Copy());
            return new Polynomial(temp);
        }

        /// <summary>
        /// Adds a Polynomial with a MultiVarTerm
        /// </summary>
        /// <param name="p">The Polynomial</param>
        /// <param name="t">The MultiVarTerm</param>
        /// <returns>The sum of the two objects</returns>
        public static Polynomial operator +(MultiVarTerm t, Polynomial p)
        {
            return p + t;
        }

        /// <summary>
        /// Adds a Polynomial with a double number
        /// </summary>
        /// <param name="p">The Polynomial</param>
        /// <param name="c">The double number</param>
        /// <returns>The sum of the two objects</returns>
        public static Polynomial operator +(Polynomial p, double c)
        {
            List<MultiVarTerm> temp = new List<MultiVarTerm>();
            foreach (MultiVarTerm m in p._terms)
                temp.Add(m.Copy());
            temp.Add(new MultiVarTerm(c));
            return new Polynomial(temp);
        }

        /// <summary>
        /// Adds a Polynomial with a double number
        /// </summary>
        /// <param name="p">The Polynomial</param>
        /// <param name="c">The double number</param>
        /// <returns>The sum of the two objects</returns>
        public static Polynomial operator +(double c, Polynomial p)
        {
            return p + c;
        }

        /// <summary>
        /// Calculates the opposite Polynomial
        /// </summary>
        /// <param name="p">The Polynomial</param>
        /// <returns>The opposite Polynomial</returns>
        public static Polynomial operator -(Polynomial p)
        {
            List<MultiVarTerm> temp = new List<MultiVarTerm>();
            foreach (MultiVarTerm t in p._terms) //Calculates the opposite
terms
```

```
            temp.Add(-t.Copy());
        return new Polynomial(temp);
    }

    /// <summary>
    /// Substracts a Polynomial with a Polynomial
    /// </summary>
    /// <param name="p1">The first Polynomial</param>
    /// <param name="p2">The second Polynomial</param>
    /// <returns>The difference of the two objects</returns>
    public static Polynomial operator -(Polynomial p1, Polynomial p2)
    {
        return p1 + (-p2);
    }

    /// <summary>
    /// Substracts a Polynomial with a MultiVarTerm
    /// </summary>
    /// <param name="p">The Polynomial</param>
    /// <param name="t">The MultiVarTerm</param>
    /// <returns>The difference of the two objects</returns>
    public static Polynomial operator -(Polynomial p, MultiVarTerm t)
    {
        return p + (-t);
    }

    /// <summary>
    /// Substracts a Polynomial with a MultiVarTerm
    /// </summary>
    /// <param name="p">The Polynomial</param>
    /// <param name="t">The MultiVarTerm</param>
    /// <returns>The difference of the two objects</returns>
    public static Polynomial operator -(MultiVarTerm t, Polynomial p)
    {
        return - p + t;
    }

    /// <summary>
    /// Substracts a Polynomial with a double number
    /// </summary>
    /// <param name="p">The Polynomial</param>
    /// <param name="c">The double number</param>
    /// <returns>The difference of the two objects</returns>
    public static Polynomial operator -(Polynomial p1, double c)
    {
        return p1 + (-c);
    }

    /// <summary>
    /// Substracts a Polynomial with a double number
    /// </summary>
    /// <param name="p">The Polynomial</param>
    /// <param name="c">The double number</param>
    /// <returns>The difference of the two objects</returns>
    public static Polynomial operator -(double c, Polynomial p)
    {
        return p - c;
    }
    #endregion

    #region Other Methods
    /// <summary>
```

```
        /// Returns the Polynomial in the form of a string
        /// </summary>
        /// <returns>A string</returns>
        public override string ToString()
        {
            Sort();
            StringBuilder str = new StringBuilder();

            bool first = true;
            for (int i = 0; i < _terms.Count; i++)
            {
                if (Math.Sign(_terms[i].Coefficient) == 1 && !first) {
str.Append(" + "); }
                first = false;
                str.Append(_terms[i].ToString());
            }

            return str.ToString();
        }

        /// <summary>
        /// Creates a deep copy of the Polynomial
        /// </summary>
        /// <returns>A copy of the Polynomial</returns>
        public Polynomial Copy()
        {
            Polynomial copy = new Polynomial();

            foreach (MultiVarTerm term in this._terms)
                copy._terms.Add(term.Copy());
            //copy._terms.Sort();
            return copy;
        }

        /// <summary>
        /// Sorts the terms of the current Polynomial
        /// </summary>
        public void Sort()
        {
            _terms.Sort();
        }

        /// <summary>
        /// Sets the read-only property of the Polynomial
        /// </summary>
        /// <param name="readOnly">A boolean value determining whether the
Polynomial will be read-only</param>
        public void SetReadOnly(bool readOnly)
        {
            _isReadOnly = readOnly;
        }
        #endregion
    }
}
```

3.2 Braids

3.2.1 Parser.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Braids
{
    /// <summary>
    /// Parses a string and converts it to a word
    /// </summary>
    public static class Parser
    {
        /// <summary>
        /// Checks if the string is a valid word
        /// </summary>
        /// <param name="word">The string</param>
        /// <returns>A List of String containing the elements of the
word</returns>
        private static List<String> Validate(String word)
        {
            List<String> output = new List<String>();

            String str = "";
            if (!(word[0] == 'g' || word[0] == 't')) { throw new
Exception("Word must begin with g or t"); }
            str = word[0].ToString();
            for (int i = 1; i < word.Length; i++)
            {
                if (word[i] == 'g' || word[i] == 't')
                {
                    output.Add(str);
                    if (str.IndexOf('^') != str.LastIndexOf('^'))
                        throw new Exception("Check the syntax of the word");
                    str = word[i].ToString();
                }
                else
                {
                    str += word[i];
                }
            }
            output.Add(str);

            return output;
        }

        /// <summary>
        /// Parses a word
        /// </summary>
        /// <param name="word">The word</param>
        /// <returns>A word object</returns>
        public static Word Parse(String word)
        {
            List<int[]> output = new List<int[]>();
            try
            {
                List<String> input = Validate(word);
                string[] temp;
                string s;
```

```
                int t0, t1, t2;
                for (int i=0; i<input.Count; i++)
                {
                    s = input[i];
                    s = s.Replace("(", ""); //Future use
                    s = s.Replace(")", ""); //Future use

                    t0 = s[0];
                    s = s.Remove(0, 1);
                    if (s.Contains('^'))
                    {
                        temp = s.Split('^');
                        t1 = Convert.ToInt32(temp[0]);
                        t2 = Convert.ToInt32(temp[1]);
                        output.Add( new int[] {t0, t1, t2});
                    }
                    else
                    {
                        t1 = Convert.ToInt32(s);
                        output.Add(new int[] { t0, t1, 1});
                    }
                }

                Word outWord = new Word(output);
                return outWord;
            }
            catch (Exception e)
            {
                throw e;
            }
        }
    }
}
```

3.2.2 Compute.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using PolyLib;

namespace Braids
{
    /// <summary>
    /// A class containing some extra computations functions
    /// </summary>
    static class Compute
    {
        /// <summary>
        /// Computes the Euclidean remainder of a divided by n
        /// </summary>
        /// <param name="a">The dividend</param>
        /// <param name="n">The divisor</param>
        /// <returns>The remainder</returns>
        public static int Mod(int a, int n)
        {
            int result = a % n;
            if (result >= 0) { return result; }
            else
                return result + n;
        }
```

```
    }
}
```

3.2.3 Comparer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Braids
{
    class DistComparer : IComparer<int[]>
    {
        public int Compare(int[] t1, int[] t2)
        {
            return t1[2] - t2[2];
        }
    }

    class FramingComparer : IComparer<int[]>
    {
        public int Compare(int[] t1, int[] t2)
        {
            return t1[0] - t2[0];
        }
    }
}
```

3.2.4 Word.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PolyLib;

namespace Braids
{
    /// <summary>
    /// A class modeling a Word for Hecke and Yokonuma-Hecke algebras.
    /// </summary>
    public class Word
    {
        #region Properties
        /// <summary>
        /// The framing part of the word (if available)
        /// </summary>
        public List<int[]> Framing { get; set; }

        /// <summary>
        /// The braiding part of the word
        /// </summary>
        public List<int[]> Braiding { get; set; }

        /// <summary>
        /// The polynomial coefficient of the word
        /// </summary>
        public Polynomial Coeff { get; set; }
        #endregion
```

```
        #region Constructors
        /// <summary>
        /// Creates a unitary Word (1)
        /// </summary>
        public Word()
        {
            Framing = new List<int[]>();
            Braiding = new List<int[]>();
            Coeff = new Polynomial(1);
        }

        /// <summary>
        /// Creates a new Word
        /// </summary>
        /// <param name="elements">A list of integer arrays containing the
elements of the word</param>
        public Word(List<int[]> elements)
        {
            Tuple<List<int[]>, List<int[]>> temp = MoveFramingPart(elements);
//Separates the framing and braiding parts
            Framing = temp.Item1; //Sets the framing part
            Braiding = temp.Item2; //Sets the braiding part
            Coeff = new Polynomial(1); //Sets the default coefficient (1)
            RemoveDuplicates(); //Removes any duplicate elements
        }

        /// <summary>
        /// Creates a new Word
        /// </summary>
        /// <param name="elements">A list of integer arrays containing the
elements of the word</param>
        /// <param name="coefficient">A polynomial coefficient</param>
        public Word(List<int[]> elements, Polynomial coefficient)
        {
            Tuple<List<int[]>, List<int[]>> temp = MoveFramingPart(elements);
//Separates the framing and braiding parts
            Framing = temp.Item1; //Sets the framing part
            Braiding = temp.Item2; //Sets the braiding part
            Coeff = coefficient.Copy(); //Sets the coefficient
            RemoveDuplicates(); //Removes any duplicate elements
        }

        /// <summary>
        /// Creates a new Word
        /// </summary>
        /// <param name="framingPart">A list of integer arrays containing the
framing part</param>
        /// <param name="braidingPart">A list of integer arrays containing the
framing part</param>
        /// <param name="coefficient">A polynomial coefficient</param>
        public Word(List<int[]> framingPart, List<int[]> braidingPart,
Polynomial coefficient)
        {
            //Checks if the integer arrays are valid
            foreach (int[] arr in framingPart)
                if (arr.Length < 2) { throw new Exception("Data are missing
from an framing element"); }
            foreach (int[] arr in braidingPart)
                if (arr.Length < 2) { throw new Exception("Data are missing
from an braiding element"); }

            //Sets the framing part
```

```
            Framing = new List<int[]>();
            for (int i = 0; i < framingPart.Count; i++)
                Framing.Add(new int[] {framingPart[i][0], framingPart[i][1]});

            //Sets the braiding part
            Braiding = new List<int[]>();
            for (int i = 0; i < braidingPart.Count; i++)
                Braiding.Add(new int[] {braidingPart[i][0],
braidingPart[i][1]});

            Coeff = coefficient.Copy(); //Sets the coefficient
            RemoveDuplicates(); //Removes any duplicate elements
        }
        #endregion

        #region Operators
        /// <summary>
        /// Multiplies two Words by multiplying the coefficients and merging
the elements
        /// </summary>
        /// <param name="w1">The first Word</param>
        /// <param name="w2">The second Word</param>
        /// <returns>A Word object (the product of the two words)</returns>
        public static Word operator *(Word w1, Word w2)
        {
            if (w2.Framing.Count == 0 && w2.Braiding.Count == 0) //w2 is an
empty word
                return new Word(w1.Framing, w1.Braiding, w1.Coeff * w2.Coeff);
            else if (w1.Framing.Count == 0 && w1.Braiding.Count == 0) //w1 is
an empty word
                return new Word(w2.Framing, w2.Braiding, w1.Coeff * w2.Coeff);
            else if (w2.Framing.Count == 0) //w2 has no framing
            {
                //Copies the braiding parts of both words to a new list
                List<int[]> braid = new List<int[]>();
                for (int i = 0; i < w1.Braiding.Count; i++)
                    braid.Add(new int[] { w1.Braiding[i][0], w1.Braiding[i][1]
});
                for (int i = 0; i < w2.Braiding.Count; i++)
                    braid.Add(new int[] { w2.Braiding[i][0], w2.Braiding[i][1]
});
                //Creates the new word
                return new Word(w1.Framing, braid, w1.Coeff * w2.Coeff);
            }
            else //w2 has framing and braiding part
            {
                //Copies the elements of the first word to the new list
                List<int[]> temp = new List<int[]>();
                for (int i = 0; i < w1.Framing.Count; i++)
                    temp.Add(new int[] { 't', w1.Framing[i][0],
w1.Framing[i][1] });
                for (int i = 0; i < w1.Braiding.Count; i++)
                    temp.Add(new int[] { 'g', w1.Braiding[i][0],
w1.Braiding[i][1] });
                //Copies the elements of the first word to the new list
                for (int i = 0; i < w2.Framing.Count; i++)
                    temp.Add(new int[] { 't', w2.Framing[i][0],
w2.Framing[i][1] });
                for (int i = 0; i < w2.Braiding.Count; i++)
                    temp.Add(new int[] { 'g', w2.Braiding[i][0],
w2.Braiding[i][1] });
                return new Word(temp, w1.Coeff * w2.Coeff);
```

```
            }
        }

        /// <summary>
        /// Multiplies a word with a polynomial
        /// </summary>
        /// <param name="w">The word</param>
        /// <param name="p">The polynomial</param>
        /// <returns>A Word object (the product)</returns>
        public static Word operator *(Word w, Polynomial p)
        {
            return new Word(w.Framing, w.Braiding, w.Coeff * p);
        }

        /// <summary>
        /// Multiplies a word with a polynomial
        /// </summary>
        /// <param name="p">The polynomial</param>
        /// <param name="w">The word</param>
        /// <returns>A Word object (the product)</returns>
        public static Word operator *(Polynomial p, Word w)
        {
            return w * p;
        }

        /// <summary>
        /// Multiplies a word with an element
        /// </summary>
        /// <param name="w1">The word</param>
        /// <param name="w2">The element</param>
        /// <returns>A Word object (the product)</returns>
        public static Word operator *(Word w, int[] elem)
        {
            if (elem.Length < 3) { throw new Exception("Data are missing from
an element"); }

            if (elem[0] == 'g') //elem belongs to the braiding part
            {
                //Creates a list that contains the braiding part plus elem
                List<int[]> braiding = new List<int[]>();
                for (int i = 0; i < w.Braiding.Count; i++)
                    braiding.Add(w.Braiding[i]);
                braiding.Add(new int[] { elem[1], elem[2] });
                return new Word(w.Framing, braiding, w.Coeff);
            }
            else
            {
                //Creates a list that contains both braiding and framing pars
                List<int[]> temp = new List<int[]>();
                for (int i = 0; i < w.Framing.Count; i++)
                    temp.Add(new int[] { 't', w.Framing[i][0], w.Framing[i][1]
});
                for (int i = 0; i < w.Braiding.Count; i++)
                    temp.Add(new int[] { 'g', w.Braiding[i][0],
w.Braiding[i][1] });
                temp.Add(elem);
                return new Word(temp, w.Coeff);
            }
        }
        #endregion

        #region Private Methods
```

```
        /// <summary>
        /// Searches and applies all possible commutes
        /// </summary>
        /// <param name="maxIndexes">An array containing the indexes of the
symbols with maximum index</param>
        private void SearchCommute(int[] maxIndexes, bool yokonuma)
        {
            List<int[]> dist = new List<int[]>();

            for (int i = 0; i < maxIndexes.Length; i++)
            {
                for (int j = i; j < maxIndexes.Length; j++)
                    if (i != j)
                    {
                        if (maxIndexes[i] < maxIndexes[j]) { dist.Add(new int[]
{ maxIndexes[i], maxIndexes[j], maxIndexes[j] - maxIndexes[i] - 1 }); }
                        else { dist.Add(new int[] { maxIndexes[i],
maxIndexes[j], Braiding.Count - (maxIndexes[i] - maxIndexes[j] - 1) - 2 }); }
                    }
            }
            dist.Sort(new DistComparer());
            dist.Reverse();
            if (dist.Count > 0) { Commute(dist[0][0], dist[0][1], false, 0,
yokonuma); }
        }

        /// <summary>
        /// Applies all possible commutative relations for the item in
<paramref name="startIndex"/>
        /// </summary>
        /// <param name="Elements">A Elements with no powers of gi from
<paramref name="startIndex"/> to <paramref name="endIndex"/></param>
        /// <param name="startIndex">The start</param>
        /// <param name="endIndex">The end</param>
        /// <param name="ignoreExp">If true, it ignores any exponent -> returns
always false</param>
        /// <param name="offset">An offset (for future use)</param>
        /// <returns>A boolean value describing whether the Elements contains a
power</returns>
        private bool Commute(int startIndex, int endIndex, bool ignoreExp, int
offset, bool yokonuma)
        {
            int newPos = -1; //The new position of the item
            //TODO: int max =

            if (startIndex < offset)
            {
                throw new Exception();
            }
            if (Braiding.Count - endIndex - 1 < offset)
            {
                throw new Exception();
            }

            //Moves start item from left to right
            int oldCount = Braiding.Count; //Word length before any move
            int posL = BubbleHecke(startIndex, endIndex, offset, yokonuma);
//The new startIndex
            int newCount = Braiding.Count; //Word length after

            if (ignoreExp)
            {
```

```
                if (oldCount - newCount > 0)
                    endIndex -= oldCount - newCount; //endIndex has been moved
            }
            else
            {
                if (Braiding[posL][1] > 1)
                    return true; //The item has an exponent larger than 1
            }

            if (Braiding[posL][1] > 1 && !ignoreExp)
                return true; //The item has an exponent larger than 1 --->
replace!

            //Moves end item from right to left
            int posR = BubbleHecke(endIndex, startIndex, offset, yokonuma);
//The new endIndex
            if (Braiding[posR][1] > 1 && !ignoreExp)
                return true; //The item has an exponent larger than 1 --->
replace!

            int middle = posL + (posR - posL) / 2; //The middle index

            //Commute from middle to left
            for (int i = posL + 1; i <= middle; i++) //Middle will be moved too
            {
                newPos = BubbleHecke(i, posL, offset, yokonuma);
                bool beforeWord = newPos == posL; //The item is moved before
the start item
                bool exp = Braiding[newPos][0] > 1; //Checks if an exponent
greater than 1 is created (BubbleHecke automatically finds squares)

                if (beforeWord)
                {
                    if (exp && !ignoreExp)
                        return true; //(iii) Square found by BubbleHecke
                    else
                    {
                        posL++; //(i) The index of the left item is moved
                        middle = posL + (posR - posL) / 2; //Middle must be
recalculated
                    }
                }
                else
                    if (exp && !ignoreExp)
                        return true; //(ii) Square found by BubbleHecke
            }

            //Commute from middle to right
            for (int i = middle + 1; i < posR; i++)
            {
                newPos = BubbleHecke(i, posR, offset, yokonuma);
                bool afterWord = newPos == posR - 1; //The item is moved before
the start item
                bool exp = Braiding[newPos][0] > 1; //Checks if an exponent
greater than 1 is created (BubbleHecke automatically finds squares)

                if (afterWord)
                {
                    if (exp && !ignoreExp)
                        return true; //(iii) Square found by BubbleHecke
                    else
                    {
```

```
                        posR--; //(i) The index of the right item is moved
                        middle = posL + (posR - posL) / 2; //Middle must be
recalculated
                    }
                }
                else
                    if (exp && !ignoreExp)
                        return true; //(ii) Square found by BubbleHecke
            }

            return false; //No powers, all are OK
        }

        /// <summary>
        /// Makes all possible commutes in braiding part for the item located
at <paramref name="start"/>
        /// </summary>
        /// <param name="start">The start index</param>
        /// <param name="end">The end index</param>
        /// <returns>The new position of the item</returns>
        private int BubbleHecke(int start, int end, int offset, bool yokonuma)
        {
            if (start < end)
            {
                int pos = start; //The new position of the item
                do
                    pos++;
                while (pos <= end && Math.Abs(Braiding[pos][0] -
Braiding[start][0]) > 1); //Checks for all possible commutes (far
commutativity)
                pos--; //because of the while loop

                for (int i = 0; i <= offset; i++)
                    Swap(Braiding, start - i, pos - i); //Moves the item

                //Checks for duplicates
                if (Braiding[pos][0] == Braiding[Compute.Mod((pos + 1),
Braiding.Count)][0])
                {
                    Braiding[pos][1] += Braiding[Compute.Mod((pos + 1),
Braiding.Count)][1];
                    Braiding.RemoveAt((pos + 1) % Braiding.Count);
                }
                if (!yokonuma && Braiding[pos][0] == Braiding[Compute.Mod((pos
- 1), Braiding.Count)][0])
                {
                    Braiding[Compute.Mod((pos - 1), Braiding.Count)][1] +=
Braiding[pos][1];
                    Braiding.RemoveAt(pos);
                    return pos - 1; //Returns the new position of the item
                }
                return pos; //Returns the new position of the item
            }
            else if (start == end) //No commute needed
                return start; //Returns the start index
            else
            {
                int pos = start; //The new position of the item
                do
                    pos--;
```

```
                while (pos >= end && Math.Abs(Braiding[pos][0] -
Braiding[start][0]) > 1); //Checks for all possible commutes (far
commutativity)
                pos++; //because of the while loop

                for (int i = 0; i <= offset; i++)
                    Swap(Braiding, start + i, pos + i); //Moves the item

                //Checks for duplicates
                if (Braiding[Compute.Mod((pos - 1), Braiding.Count)][0] ==
Braiding[pos][0])
                {
                    Braiding[Compute.Mod((pos - 1), Braiding.Count)][1] +=
Braiding[pos][1];
                    Braiding.RemoveAt(pos);
                    return pos - 1;
                }
                if (!yokonuma && Braiding[pos][0] == Braiding[Compute.Mod((pos
+ 1), Braiding.Count)][0])
                {
                    Braiding[pos][1] += Braiding[Compute.Mod((pos + 1),
Braiding.Count)][1];
                    Braiding.RemoveAt(Compute.Mod((pos + 1), Braiding.Count));
                }
                return pos; //Returns the new position of the item
            }
        }

        /// <summary>
        /// Swaps <paramref name="data1"/> and <paramref name="data2"/> (and
the items between) in the <paramref name="part"/>
        /// </summary>
        /// <param name="part">The framing or the braiding part</param>
        /// <param name="data1">The first item</param>
        /// <param name="data2">The second item</param>
        private void Swap(List<int[]> part, int data1, int data2)
        {
            if (data1 < data2) //Swap to left ( i <- i + 1)
            {
                int[] temp = part[data1];
                //Moves all data to the left
                for (int i = data1; i < data2; i++)
                    part[i] = part[i + 1];
                //Moves data1 to data2
                part[data2] = temp;
            }
            else if (data1 > data2) //Swap to right ( i - 1 -> i)
            {
                int[] temp = part[data1];
                //Moves all data to the right
                for (int i = data1; i > data2; i--)
                    part[i] = part[i - 1];
                //Moves data1 to data2
                part[data2] = temp;
            }
            else //The two items are the same -> no swap needed
                return;
        }

        /// <summary>
        /// Checks if the <paramref name="part"/> contains any power
        /// </summary>
```

```
        /// <param name="part">The framing or the braiding part</param>
        /// <returns>A boolean value determining whether the <paramref
name="part"/> contains any exponent != 1</returns>
        private bool HasPowers(List<int[]> part)
        {
            return ((from w in part where w[1] != 1 select w).Count() > 0);
        }

        /// <summary>
        /// Returns the indexes of the <paramref name="part"/> are raised to a
power
        /// </summary>
        /// <param name="part">The framing or the braiding part</param>
        /// <returns>An array of integer containing the indexes</returns>
        private int[] FindPowers(List<int[]> part)
        {
            return (from e in part where e[1] != 1 select
part.IndexOf(e)).ToArray<int>();
        }

        /// <summary>
        /// Finds the indexes with the maximum subscript in <paramref
name="part"/>
        /// </summary>
        /// <param name="part">The framing or the braiding part</param>
        /// <returns>An array of integers containing the indexes</returns>
        private int[] FindMaxIndexes(List<int[]> part)
        {
            List<int> index = new List<int>();
            List<int> subscr = (from w in part select w[0]).ToList<int>();
            for (int i = 0; i < part.Count; i++)
                if (subscr[i] == subscr.Max())
                    index.Add(i);
            return index.ToArray<int>();
        }

        /// <summary>
        /// Checks if the braiding part contains a brigde
        /// </summary>
        /// <returns>An integer tuple with 2 items, containing the start and
the end of the bridge</returns>
        private Tuple<int, int> FindBridge()
        {
            List<int> localMinimaList = new List<int>();

            for (int i = 1; i < Braiding.Count - 1; i++)
                if (Braiding[i][0] == Braiding[i + 1][0] - 1 && Braiding[i][0]
== Braiding[i - 1][0] - 1)
                    localMinimaList.Add(i);

            int[] localMinima = localMinimaList.ToArray<int>();
            int count = localMinima.Length; //Count of local minima

            //Check if local minima exist
            //If there isn't any local minimum, the Elements must contain a
power, so we shouldn't have called this method
            //But I check for the existence only for consistency
            if (count > 0)
            {
                int[] bridgeSize = new int[count]; //Contains the size of every
bridge found for each local minimum
```

```
                //Fills the list with zeros
                for (int i = 0; i < count; i++)
                    bridgeSize[i] = 0;

                int dist; //Dump variable ---> helps in checking the
                for (int i = 0; i < count; i++)
                {
                    dist = 1;
                    while (Braiding[Compute.Mod(localMinima[i] - dist,
Braiding.Count)][0] == Braiding[Compute.Mod(localMinima[i] + dist,
Braiding.Count)][0] //The left and the right elements are the same
                        && Braiding[Compute.Mod(localMinima[i] - dist,
Braiding.Count)][0] == Braiding[Compute.Mod(localMinima[i] - dist + 1,
Braiding.Count)][0] + 1)//+1 from the previous elements
                    {
                        dist++;
                        bridgeSize[i]++;
                    }
                }

                Array.Sort(bridgeSize, localMinima); //Sorts the arrays with
descending sorting based on brigdeSize array

                return new Tuple<int, int>(Compute.Mod(localMinima[count - 1] -
bridgeSize[count - 1], Braiding.Count), Compute.Mod(localMinima[count - 1] +
bridgeSize[count - 1], Braiding.Count));
            }


            return new Tuple<int, int>(-1, -1);
        }

        /// <summary>
        /// Replaces a bridge with a mountain
        /// </summary>
        /// <param name="start">The start of the bridge</param>
        /// <param name="end">The end of the bridge</param>
        private void ReplaceBridge(int start, int end)
        {
            if (start < end) //Internal bridge
            {
                int half = (end - start) / 2; //The middle index
                int max = Braiding[start][0]; //The maximum item
                int min = Braiding[start + half][0]; //The minimum item

                for (int i = 0; i < half; i++) //Replaces the items
                {
                    int[] temp = new int[] { min + i, 1 }; //The new item
                    Braiding[start + i] = temp; //Replace of the left item
                    Braiding[end - i] = temp; //Replace of the right item
                }
                Braiding[start + half] = new int[] { max, 1 }; //Replace of the
middle item
            }
            else if (end < start) //External bridge
            {
                int[] temp;
                //Swap tr(ab) = tr(ba)

                if (end <= Braiding.Count / 2) //Swaps from the end to the
beginning of the Elements
                {
```

```
                    for (int i = 0; i < Braiding.Count - start; i++)
                    {
                        temp = Braiding[Braiding.Count - 1];
                        Braiding.RemoveAt(Braiding.Count - 1);
                        Braiding.Insert(0, temp);
                    }
                    ReplaceBridge(0, end + Braiding.Count - start); //Replaces
the new internal bridge
                }
                else //Swaps from the beginning to the end of the Elements
                {
                    for (int i = 0; i <= end; i++)
                    {
                        temp = Braiding[0];
                        Braiding.RemoveAt(0);
                        Braiding.Add(temp);
                    }
                    ReplaceBridge(start - end - 1, Braiding.Count - 1);
//Replaces the new internal bridge
                }

            }
            RemoveDuplicates();
        }

        /// <summary>
        /// Removes duplicates from both framing and braiding parts
        /// </summary>
        private void RemoveDuplicates()
        {
            Framing.Sort(new FramingComparer());
            Framing.Reverse();

            DeleteZeroExp();

            for (int i = 0; i < Framing.Count - 1; i++)
                if (Framing[i][0] == Framing[i + 1][0])
                {
                    Framing[i][1] += Framing[i + 1][1];
                    Framing.RemoveAt(i + 1);
                    i--;
                }
            if (Framing.Count > 1 && Framing[0][0] == Framing[Framing.Count -
1][0]) //Checks if the first and the last item have the same index (count = 1
<=> start = end)
            {
                Framing[0][1] += Framing[Framing.Count - 1][1];
                Framing.RemoveAt(Braiding.Count - 1);
            }

            for (int i = 0; i < Braiding.Count - 1; i++)
                if (Braiding[i][0] == Braiding[i + 1][0])
                {
                    Braiding[i][1] += Braiding[i + 1][1];
                    Braiding.RemoveAt(i + 1);
                    i--;
                }
            if (Framing.Count == 0 && Braiding.Count > 1 && Braiding[0][0] ==
Braiding[Braiding.Count - 1][0]) //Checks if the first and the last item have
the same index (count = 1 <=> start = end)
            {
                Braiding[0][1] += Braiding[Braiding.Count - 1][1];
```

```
                Braiding.RemoveAt(Braiding.Count - 1);
            }
        }

        /// <summary>
        /// Makes all apporopriate calculations (mod d) at the exponents of the
framing part
        /// </summary>
        /// <param name="d">The integer d of Y_d,n(u)</param>
        private void RemoveModDuplicates(int d)
        {
            for (int i = 0; i < Framing.Count; i++)
                Framing[i][1] = Compute.Mod(Framing[i][1], d);
        }

        /// <summary>
        /// Deletes all the elements with zero exponent in the framing part
        /// </summary>
        private void DeleteZeroExp()
        {
            for (int i = 0; i < Framing.Count; i++)
                if (Framing[i][1] == 0)
                {
                    Framing.RemoveAt(i);
                    i--;
                }
            for (int i = 0; i < Braiding.Count; i++)
                if (Braiding[i][1] == 0)
                {
                    Braiding.RemoveAt(i);
                    i--;
                }
        }

        /// <summary>
        /// Splits the word to two different parts (the coefficient is copied
only to the first word)
        /// </summary>
        /// <param name="frame">True if <paramref name="pos"/> is in the
framing part, false for the braiding part</param>
        /// <param name="pos">The position of the split in <paramref
name="part"/></param>
        /// <returns>An array of Words containing the left and the right
parts</returns>
        private Word[] Split(bool frame, int pos)
        {
            Word[] temp = new Word[2];
            temp[0] = new Word();
            //temp[0].Coeff = Coeff;
            temp[1] = new Word();

            if (frame)
            {
                //Left part
                for (int i = 0; i < pos; i++)
                    temp[0].Framing.Add(new int[] { Framing[i][0],
Framing[i][1] });
                //Right part
                for (int i = pos + 1; i < Framing.Count; i++)
                    temp[1].Framing.Add(new int[] { Framing[i][0],
Framing[i][1] });
                for (int i = 0; i < Braiding.Count; i++)
```

```
                    temp[1].Braiding.Add(new int[] { Braiding[i][0],
Braiding[i][1] });
            }
            else
            {
                //Left part
                for (int i = 0; i < Framing.Count; i++)
                    temp[0].Framing.Add(new int[] { Framing[i][0],
Framing[i][1] });
                for (int i = 0; i < pos; i++)
                    temp[0].Braiding.Add(new int[] { Braiding[i][0],
Braiding[i][1] });
                //Right part
                for (int i = pos + 1; i < Braiding.Count; i++)
                    temp[1].Braiding.Add(new int[] { Braiding[i][0],
Braiding[i][1] });
            }
            temp[0].RemoveDuplicates();
            //temp[1].RemoveDuplicates();
            return temp;
        }

        /// <summary>
        /// Moves the framing part to the start of the word
        /// </summary>
        /// <param name="elements">A list containing all the elements of the
word</param>
        /// <returns>A Tuple containing the framing and the braiding parts
separately</returns>
        private static Tuple<List<int[]>, List<int[]>>
MoveFramingPart(List<int[]> elements)
        {
            foreach (int[] arr in elements)
                if (arr.Length < 3) { throw new Exception("Data are missing
from an element"); }

            List<int[]> framing = new List<int[]>();
            List<int[]> braiding = new List<int[]>();
            for (int i = 0; i < elements.Count; i++)
            {
                if (elements[i][0] == 't')
                {
                    int result = elements[i][1];
                    //σ_i(j)
                    for (int j = i - 1; j > -1; j--)
                    {
                        if (elements[j][1] == result && elements[j][0] == 'g'
&& elements[j][2] % 2 != 0)
                            result++;
                        else if (elements[j][1] + 1 == result && elements[j][0]
== 'g' && elements[j][2] % 2 != 0)
                            result--;
                    }
                    elements[i][1] = result;
                    framing.Add(new int[] { elements[i][1], elements[i][2] });
                    elements.RemoveAt(i);
                    i--;
                }
            }

            for (int i = 0; i < elements.Count; i++)
                braiding.Add(new int[] { elements[i][1], elements[i][2] });
```

```
            return new Tuple<List<int[]>, List<int[]>>(framing, braiding);
        }

        /// <summary>
        /// Multiplies two Words by multiplying the coefficients and merging
the elements
        /// </summary>
        /// <param name="w1">The first Word</param>
        /// <param name="w2">The second Word</param>
        /// <returns>A Word object (the product of the two words)</returns>
        private static List<int[]> MergeHecke(Word w1, int[] elem)
        {
            //Copies the braiding parts of both words to a new list
            List<int[]> braid = new List<int[]>();
            for (int i = 0; i < w1.Braiding.Count; i++)
                braid.Add(new int[] { w1.Braiding[i][0], w1.Braiding[i][1] });
            braid.Add(elem);
            return braid;
        }

        /// <summary>
        /// Multiplies two Words by multiplying the coefficients and merging
the elements
        /// </summary>
        /// <param name="w1">The first Word</param>
        /// <param name="w2">The second Word</param>
        /// <returns>A Word object (the product of the two words)</returns>
        private static List<int[]> MergeHeckeWords(Word w1, Word w2)
        {
            //Copies the braiding parts of both words to a new list
            List<int[]> braid = new List<int[]>();
            for (int i = 0; i < w1.Braiding.Count; i++)
                braid.Add(new int[] { w1.Braiding[i][0], w1.Braiding[i][1] });
            for (int i = 0; i < w2.Braiding.Count; i++)
                braid.Add(new int[] { w2.Braiding[i][0], w2.Braiding[i][1] });
            return braid;
        }
        #endregion

        #region Trace Calculation
        /// <summary>
        /// Calculates the Ocneanu trace for the braiding part.
        /// </summary>
        /// <returns>A complex polynomial.</returns>
        public Polynomial TraceHecke()
        {
            RemoveDuplicates();
            if (Braiding.Count == 1)
                return Coeff * TraceHeckeUnit(Braiding[0]);
            else
            {
                int[] maxExp = FindPowers(Braiding);
                if (maxExp.Length > 0) //The Elements contains powers
                {
                    for (int i = 0; i < maxExp.Length; i++)
                    {
                        List<Word> words = QuadraticRelationHecke(maxExp[0]);
                        Polynomial poly = Coeff * (words[0].TraceHecke() +
words[1].TraceHecke());
                        return poly;
                    }
```

```
                }
                else //No powers in Elements
                {
                    int[] maxInd = FindMaxIndexes(Braiding); //Finds the
indexes of maximum subscripts
                    if (maxInd.Length == 1) //Only one maximum subscript
                    {
                        Braiding.RemoveAt(maxInd[0]); //Removes the letter
containing the maximum subscript
                        Polynomial poly = TraceHecke();
                        return new Polynomial(new MultiVarTerm(1, 'z', 1)) *
poly; //z * tr(remaining Elements)
                    }
                    else //Multiple maximum subscripts
                    {
                        SearchCommute(maxInd, false);
                        maxExp = FindPowers(Braiding); //Re-calculates for
powers
                        if (maxExp.Length > 0) //The Elements contains powers
                        {
                            for (int i = 0; i < maxExp.Length; i++)
                            {
                                List<Word> words =
QuadraticRelationHecke(maxExp[0]);
                                Polynomial poly = Coeff *
(words[0].TraceHecke() + words[1].TraceHecke());
                                return poly;
                            }
                        }
                        else //No powers after commute
                        {
                            Tuple<int, int> index = FindBridge(); //Searches
for a bridge

                            if (index.Item1 == -1) //MUST NOT BE REACHED: if
reached, there is a set of braids whose trace can not be calculated
                                throw new Exception("The trace for this word
cannot be calculated"); //Exception thrown
                            else //Bridge found
                            {
                                ReplaceBridge(index.Item1, index.Item2);
//Replaces the bridge with a mountain
                                Polynomial poly = TraceHecke();
                                return poly; //Calls recursively the trace
function
                            }
                        }
                    }
                }
            }

            return null; //TODO: Delete
        }

        /// <summary>
        /// Calculates the Ocneanu trace for a single element
        /// </summary>
        /// <param name="symbol">The element</param>
        /// <returns>A complex polynomial</returns>
        private static Polynomial TraceHeckeUnit(int[] symbol)
        {
            if (symbol[1] == 0)
```

```
                return new Polynomial(new MultiVarTerm(1));
            else if (symbol[1] == 1)
                return new Polynomial(new MultiVarTerm(1, new Tuple<char,
int>[] { new Tuple<char, int>('z', 1) }));
            else if (symbol[1] > 1)
            {
                int sign = 1;
                List<MultiVarTerm> temp = new List<MultiVarTerm>();
                for (int i = Math.Abs(symbol[1]) - 1; i > 0; i--)
                {
                    temp.Add(new MultiVarTerm(sign, new Tuple<char, int>('q',
Math.Sign(symbol[1]) * i)));
                    sign *= -1;
                }
                Polynomial poly = new Polynomial(temp);
                MultiVarTerm z = new MultiVarTerm(1, new Tuple<char, int>[] {
new Tuple<char, int>('z', 1) });
                return (poly - (int)Math.Pow(-1, symbol[1])) * z + poly;
            }
            else //<0
            {
                int sign = 1;
                List<MultiVarTerm> temp = new List<MultiVarTerm>();
                for (int i = Math.Abs(symbol[1]); i > 0; i--)
                {
                    temp.Add(new MultiVarTerm(sign, new Tuple<char, int>('q',
Math.Sign(symbol[1]) * i)));
                    sign *= -1;
                }
                Polynomial poly = new Polynomial(temp);
                MultiVarTerm z = new MultiVarTerm(1, new Tuple<char, int>[] {
new Tuple<char, int>('z', 1) });
                return (poly * z + poly + (int)Math.Pow(-1, symbol[1]));
            }
        }

        /// <summary>
        /// Applies the Hecke quadratic relation for a word
        /// </summary>
        /// <param name="pos">The position of the letter in the braiding part
to be removed and substituted from the trace</param>
        /// <returns>A list of words whose sum is equal to the original
word</returns>
        public List<Word> QuadraticRelationHecke(int pos)
        {
            int pow = Braiding[pos][1]; //The power to which the specified
element is raised
            int sign = 1; //Dump variable (used to construct the polynomail-
coefficient)

            if (pow > 0)
            {
                //Constructs q^(n-1) - q^(n-2) + ... + (-1)^n*q polynomial
                Polynomial poly = new Polynomial();
                for (int i = Math.Abs(pow) - 1; i > 0; i--)
                {
                    poly.Add(new MultiVarTerm(sign, new Tuple<char, int>('q',
Math.Sign(pow) * i)));  //q^(n-1) + ...
                    sign *= -1;
                }
```

```
                Word[] split = Split(false, pos); //Splits the word to two
parts

                List<int[]> braid1 = new List<int[]>();
                List<int[]> braid2 = new List<int[]>();

                for (int i = 0; i < split[0].Braiding.Count; i++)
                {
                    braid1.Add(new int[] { 'g', split[0].Braiding[i][0],
split[0].Braiding[i][1] });
                    braid2.Add(new int[] { 'g', split[0].Braiding[i][0],
split[0].Braiding[i][1] });
                }
                braid1.Add(new int[] { 'g', Braiding[pos][0], 1 });
                for (int i = 0; i < split[1].Braiding.Count; i++)
                {
                    braid1.Add(new int[] { 'g', split[1].Braiding[i][0],
split[1].Braiding[i][1] });
                    braid2.Add(new int[] { 'g', split[1].Braiding[i][0],
split[1].Braiding[i][1] });
                }

                Word w1 = new Word(braid1);//w1 = A gi B //εδώ είναι το λάθος,
το στοιχείο μπαίνει μαζί με τη δύναμη
                w1.Coeff *= poly - (int)Math.Pow(-1, pow); //w1 = (C - (-1)^n)
A gi B
                Word w2 = new Word(braid2); //w2 = A B
                w2.Coeff *= poly; //w2 = C A B
                List<Word> output = new List<Word>(); //Creates the list of the
new words
                //w1.RemoveDuplicates();
                //w2.RemoveDuplicates();
                output.Add(w1);
                output.Add(w2);
                return output;
            }
            else if (pow < 0)
            {
                //Constructs q^(n-1) - q^(n-2) + ... + (-1)^n*q polynomial
                Polynomial poly = new Polynomial();
                for (int i = Math.Abs(pow); i > 0; i--)
                {
                    poly.Add(new MultiVarTerm(sign, new Tuple<char, int>('q',
Math.Sign(pow) * i)));  //q^(n-1) + ...
                    sign *= -1;
                }

                Word[] split = Split(false, pos); //Splits the word to two
parts

                List<int[]> braid1 = new List<int[]>();
                List<int[]> braid2 = new List<int[]>();

                for (int i = 0; i < split[0].Braiding.Count; i++)
                {
                    braid1.Add(new int[] { 'g', split[0].Braiding[i][0],
split[0].Braiding[i][1] });
                    braid2.Add(new int[] { 'g', split[0].Braiding[i][0],
split[0].Braiding[i][1] });
                }
                braid1.Add(new int[] { 'g', Braiding[pos][0], 1 });
                for (int i = 0; i < split[1].Braiding.Count; i++)
```

```
                {
                    braid1.Add(new int[] { 'g', split[1].Braiding[i][0],
split[1].Braiding[i][1] });
                    braid2.Add(new int[] { 'g', split[1].Braiding[i][0],
split[1].Braiding[i][1] });
                }

                Word w1 = new Word(braid1);//w1 = A gi B //εδώ είναι το λάθος,
το στοιχείο μπαίνει μαζί με τη δύναμη
                w1.Coeff *= poly; //w1 = (C - (-1)^n) A gi B
                Word w2 = new Word(braid2); //w2 = A B
                w2.Coeff *= poly + (int)Math.Pow(-1, pow); //w2 = C A B
                List<Word> output = new List<Word>(); //Creates the list of the
new words
                //w1.RemoveDuplicates();
                //w2.RemoveDuplicates();
                output.Add(w1);
                output.Add(w2);
                return output;
            }
            else
            {
                Word[] split = Split(false, pos); //Splits the word to two
parts
                List<Word> output = new List<Word>(); //Creates the list of the
new words
                output.Add(split[0] * split[1]);
                return output;
            }

        }

        /// <summary>
        /// Calculates the Yokonuma-Hecke trace for simple words (one framing
element and/or one braiding element with exp=1)
        /// </summary>
        /// <param name="framingPower">The exponent of the framing
element</param>
        /// <param name="hasBraiding">A boolean determining whether there
exists a braiding element</param>
        /// <returns></returns>
        private static Polynomial TraceYokonumaHeckeUnit(int framingPower, bool
hasBraiding)
        {
            if (framingPower == 0)
            {
                if (!hasBraiding)
                    return new Polynomial(1);
                else
                    return new Polynomial(new MultiVarTerm(1, 'z', 1));
            }
            else
            {
                if (!hasBraiding)
                    return new Polynomial(new MultiVarTerm(1, new Tuple<char,
int>((char)(64 + framingPower), 1))); //ASCII: 65 = 'A'
                else
                    return new Polynomial(new MultiVarTerm(1, 'z', 1), new
MultiVarTerm(1, new Tuple<char, int>((char)(64 + framingPower), 1))); //ASCII:
65 = 'A'
            }
        }
```

```
        /// <summary>
        /// Applies the Yokonuma-Hecke quadratic relation for a word
        /// </summary>
        /// <param name="d">The integer d of Y_d,n(u)</param>
        /// <param name="pos">The position of the letter in the braiding part
to be removed and substituted from the trace</param>
        /// <returns>A list of words whose sum is equal to the original
word</returns>
        public List<Word> QuadraticRelationYokonumaHecke(int d, int pos)
        {
            RemoveModDuplicates(d); //Makes mod d calculations
            RemoveDuplicates(); //Removes possible zero exponents

            int m = Braiding[pos][1]; //The power to which the specified
element is raised
            int index = Braiding[pos][0]; //The index of the element to be
substituted from the quadratic relation
            List<Word> output = new List<Word>();

            Word[] split = Split(false, pos);
            List<int[]> frame = new List<int[]>();
            Polynomial poly = new Polynomial();

            if (m > 0)
            {
                if (m % 2 == 0)
                {
                    output.Add(split[0] * split[1]);

                    for (int l = 0; l < m / 2; l++)
                    {
                        poly.Add(new MultiVarTerm(1, new Tuple<char, int>('u',
2 * l + 1), new Tuple<char, int>('d', -1)));
                        poly.Add(new MultiVarTerm(-1, new Tuple<char, int>('u',
2 * l), new Tuple<char, int>('d', -1)));
                    }
                    for (int n = 0; n < d; n++)
                    {
                        frame.Clear();
                        frame.Add(new int[] { index, n }); //t_i^m
                        frame.Add(new int[] { index + 1, -n }); //t_i^-m
                        output.Add(split[0] * new Word(frame, new
List<int[]>(), poly) * split[1]);
                        output.Add(split[0] * new Word(frame, new
List<int[]>(), -poly) * new int[] { 'g', index, 1 } * split[1]);
                    }
                }
                else
                {
                    output.Add(split[0] * new int[] { 'g', index, 1 } *
split[1]);

                    for (int l = 0; l < m / 2; l++)
                    {
                        poly.Add(new MultiVarTerm(1, new Tuple<char, int>('u',
2 * l + 2), new Tuple<char, int>('d', -1)));
                        poly.Add(new MultiVarTerm(-1, new Tuple<char, int>('u',
2 * l + 1), new Tuple<char, int>('d', -1)));
                    }
                    for (int n = 0; n < d; n++)
                    {
```

```
                        frame.Clear();
                        frame.Add(new int[] { index, n }); //t_i^m
                        frame.Add(new int[] { index + 1, -n }); //t_i^-m

                        output.Add(split[0] * new Word(frame, new
List<int[]>(), -poly) * split[1]);
                        output.Add(split[0] * new Word(frame, new
List<int[]>(), poly) * new int[] { 'g', index, 1 } * split[1]);
                    }
                }
            }
            else //Cannot be m=0 because zero exponents are removed from
RemoveDuplicates()
            {
                if (m % 2 == 0)
                {
                    output.Add(split[0] * split[1]);

                    for (int l = 0; l < Math.Abs(m / 2); l++)
                    {
                        poly.Add(new MultiVarTerm(1, new Tuple<char, int>('u',
-2 * l - 2), new Tuple<char, int>('d', -1)));
                        poly.Add(new MultiVarTerm(-1, new Tuple<char, int>('u',
-2 * l - 1), new Tuple<char, int>('d', -1))); //1/d [u^(2l+1) - u^(2l)]
                    }
                    for (int n = 0; n < d; n++)
                    {
                        frame.Clear();
                        frame.Add(new int[] { index, n }); //t_i^m
                        frame.Add(new int[] { index + 1, -n }); //t_i^-m

                        output.Add(split[0] * new Word(frame, new
List<int[]>(), poly) * split[1]);
                        output.Add(split[0] * new Word(frame, new
List<int[]>(), -poly) * new int[] { 'g', index, 1 } * split[1]);
                    }
                }
                else
                {
                    output.Add(split[0] * new int[] { 'g', index, 1 } *
split[1]);

                    for (int l = 0; l < Math.Abs(m / 2) + 1; l++)
                    {
                        poly.Add(new MultiVarTerm(1, new Tuple<char, int>('u',
-2 * l - 1), new Tuple<char, int>('d', -1)));
                        poly.Add(new MultiVarTerm(-1, new Tuple<char, int>('u',
-2 * l), new Tuple<char, int>('d', -1))); //1/d [u^(2l+1) - u^(2l)]
                    }
                    for (int n = 0; n < d; n++)
                    {
                        frame.Clear();
                        frame.Add(new int[] { index, n }); //t_i^m
                        frame.Add(new int[] { index + 1, -n }); //t_i^-m

                        output.Add(split[0] * new Word(frame, new
List<int[]>(), -poly) * split[1]);
                        output.Add(split[0] * new Word(frame, new
List<int[]>(), poly) * new int[] { 'g', index, 1 } * split[1]);
                    }
                }
            }
```

```
            foreach (Word w in output)
            {
                w.RemoveModDuplicates(d);
                w.RemoveDuplicates();
            }
            return output;
        }

        /// <summary>
        /// Calculates the trace for the Yokonuma-Hecke algebra
        /// </summary>
        /// <param name="d">The d integer</param>
        /// <returns>A Polynomial object</returns>
        public Polynomial TraceYokonumaHecke(int d)
        {
            RemoveModDuplicates(d); //Modulo d exponents
            RemoveDuplicates();

            int[] frame = FindMaxIndexes(Framing);

            int exp;
            if (frame.Length == 0) { exp = 0; } //No framing part
            else { exp = Framing[frame[0]][1]; }

            int[] braid = FindMaxIndexes(Braiding);

            if (exp == 0 && Braiding.Count == 0) //Simple case
                return Coeff * TraceYokonumaHeckeUnit(0, false);
            else if (exp == 0 && Braiding.Count == 1 && Braiding[0][1] == 1)
//Simple case
                return Coeff * TraceYokonumaHeckeUnit(0, true); //Simple case
            else if (exp == 0 && Braiding.Count == 1 && Braiding[0][1] != 1)
//Quadratic relation needed
            {
                List<Word> words = QuadraticRelationYokonumaHecke(d, braid[0]);
                Polynomial poly = words[0].TraceYokonumaHecke(d);
                for (int j = 1; j < words.Count; j++)
                    poly += words[j].TraceYokonumaHecke(d);
                return Coeff * poly;
            }
            else if (exp == 0 && braid.Length == 1 && !HasPowers(Braiding))
//Simple case
            {
                Word newWord = new Word(Framing, Braiding, new Polynomial(1));
                newWord.Braiding.RemoveAt(braid[0]);
                return Coeff * TraceYokonumaHeckeUnit(0, true) *
newWord.TraceYokonumaHecke(d);
            }
            else if (exp > 0)
            {
                int frameIndex = Framing[frame[0]][0];
                bool max = true;
                for (int i = 0; i < Braiding.Count; i++)
                    if (frameIndex <= Braiding[i][0] + 1)
                        max = false;

                if (max) //The framing element t_i^m comes out as x_m
                {
                    Word newWord = new Word(Framing, Braiding, new
Polynomial(1));
                    newWord.Framing.RemoveAt(frame[0]);
```

```
                    return Coeff * TraceYokonumaHeckeUnit(exp, false) *
newWord.TraceYokonumaHecke(d);
                }
                else if (braid.Length == 1 && frameIndex <=
Braiding[braid[0]][0] && Braiding[braid[0]][1] == 1)
                {
                    Word newWord = new Word(Framing, Braiding, new
Polynomial(1));
                    newWord.Braiding.RemoveAt(braid[0]);
                    return Coeff * TraceYokonumaHeckeUnit(0, true) *
newWord.TraceYokonumaHecke(d);
                }
                else if (braid.Length == 1 && frameIndex ==
Braiding[braid[0]][0] + 1 && Braiding[braid[0]][1] == 1) //The braiding element
g_i comes out as z
                {
                    Word[] split = Split(false, braid[0]);
                    split[0].Framing.RemoveAt(frame[0]);
                    Word newWord = split[0] * split[1]; //* new int[] { 't',
frameIndex - 1, exp } *
                    return Coeff * new Polynomial(new MultiVarTerm(1, 'z', 1),
new MultiVarTerm(1, new Tuple<char, int>((char)(64 + exp), 1))) *
newWord.TraceYokonumaHecke(d);
                }
                else
                {
                    SearchCommute(FindMaxIndexes(Braiding), true);
                    if (HasPowers(Braiding)) //Powers are present
                    {
                        int[] maxExp = FindPowers(Braiding);
                        List<Word> words = QuadraticRelationYokonumaHecke(d,
maxExp[0]);
                        Polynomial poly = words[0].TraceYokonumaHecke(d);
                        for (int j = 1; j < words.Count; j++)
                            poly += words[j].TraceYokonumaHecke(d);
                        return Coeff * poly;
                    }
                    else //No powers at braiding part
                    {
                        //Bridge check
                        Tuple<int, int> index = FindBridge(); //Searches for a
bridge

                        if (index.Item1 == -1) //MUST NOT BE REACHED: if
reached, there is a set of braids whose trace can not be calculated
                            throw new Exception("The trace for this word cannot
be calculated"); //Exception thrown
                        else //Bridge found
                        {
                            ReplaceBridge(index.Item1, index.Item2); //Replaces
the bridge with a mountain
                            Polynomial poly = TraceYokonumaHecke(d);
                            return poly; //Calls recursively the trace function
                        }
                    }
                }
            }
            else
            {
                SearchCommute(FindMaxIndexes(Braiding), true);
                if (HasPowers(Braiding)) //Powers are present
                {
```

```
                    int[] maxExp = FindPowers(Braiding);
                    List<Word> words = QuadraticRelationYokonumaHecke(d,
maxExp[0]);
                    Polynomial poly = words[0].TraceYokonumaHecke(d);
                    for (int j = 1; j < words.Count; j++)
                        poly += words[j].TraceYokonumaHecke(d);
                    return Coeff * poly;
                }
                else //No powers at braiding part
                {
                    //Bridge check
                    Tuple<int, int> index = FindBridge(); //Searches for a
bridge

                    if (index.Item1 == -1) //MUST NOT BE REACHED: if reached,
there is a set of braids whose trace can not be calculated
                        throw new Exception("The trace for this word cannot be
calculated"); //Exception thrown
                    else //Bridge found
                    {
                        ReplaceBridge(index.Item1, index.Item2); //Replaces the
bridge with a mountain
                        Polynomial poly = TraceYokonumaHecke(d);
                        return poly; //Calls recursively the trace function
                    }
                }
            }
        }

        #endregion

        /// <summary>
        /// Creates a string representation of the Word
        /// </summary>
        /// <returns>The Word as a string</returns>
        public override string ToString()
        {
            String str = ""; //Empty string

            //Framing part
            for (int i = 0; i < Framing.Count; i++)
            {
                str += "t" + Framing[i][0];
                if (Framing[i][1] != 1) { str += "^" + Framing[i][1]; }
                str += " ";
            }

            //Braiding part
            for (int i = 0; i < Braiding.Count; i++)
            {
                str += "g" + Braiding[i][0];
                if (Braiding[i][1] != 1) { str += "^" + Braiding[i][1]; }
                str += " ";
            }
            str = str.Remove(str.Length - 1);

            return str;
        }
    }
}
```

3.2.5 MainWindow.xaml

```
<Window x:Class="Braids.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="385" Width="552" Loaded="Window_Loaded">
    <Grid Background="Tan">
        <TabControl Name="Tabs" Height="214" VerticalAlignment="Top"
Background="Tan" Margin="12,12,204,0">
            <TabItem Header="Calculation" Name="CalculationTab">
                <Grid>
                    <Label Content="Word" Height="28"
HorizontalAlignment="Left" Margin="6,6,0,0" Name="WordLabel"
VerticalAlignment="Top" />
                    <TextBox Height="23" Margin="141,8,6,0" Name="WordTextBox"
VerticalAlignment="Top" />
                    <Button Content="Compute" IsDefault="True" Margin="0,0,6,6"
Name="ComputeButton" Height="23" VerticalAlignment="Bottom"
HorizontalAlignment="Right" Width="115" Click="ComputeButton_Click"/>
                </Grid>
            </TabItem>
            <TabItem Header="Compare" Name="CompareTab">
                <Grid HorizontalAlignment="Stretch"
VerticalAlignment="Stretch">
                    <Label Content="Word 1" Height="28"
HorizontalAlignment="Left" Margin="6,6,0,0" Name="Word1Label"
VerticalAlignment="Top" />
                    <TextBox Height="23" Margin="141,8,6,0" Name="Word1TextBox"
VerticalAlignment="Top" />
                    <Label Content="Word" Height="28"
HorizontalAlignment="Left" Margin="6,36,0,0" Name="Word2Label"
VerticalAlignment="Top" />
                    <TextBox Height="23" Margin="141,38,6,0"
Name="Word2TextBox" VerticalAlignment="Top" />
                    <Button Content="Compare" Margin="0,0,6,6"
Name="CompareButton" Width="115" Height="23" VerticalAlignment="Bottom"
HorizontalAlignment="Right" Click="CompareButton_Click" />
                </Grid>
            </TabItem>
            <TabItem Header="Tests" Name="TestsTab">
                <Grid>
                    <RadioButton Content="Test 1 (p, q, r)" Height="16"
HorizontalAlignment="Left" Margin="6,9,0,0" Name="Test1RadioButton"
VerticalAlignment="Top" />
                    <RadioButton Content="Test 2 (a, b)" Height="16"
HorizontalAlignment="Left" Margin="6,37,0,0" Name="Test2RadioButton"
VerticalAlignment="Top" />
                    <Button Content="Compute" Height="23"
HorizontalAlignment="Right" IsDefault="True" Margin="0,0,6,6"
Name="ComputeTestButton" VerticalAlignment="Bottom" Width="115"
Click="ComputeTestButton_Click" />
                    <TextBox Height="23" HorizontalAlignment="Right"
IsEnabled="{Binding ElementName=Test1RadioButton, Path=IsChecked}"
Margin="0,6,145,0" Name="FrameTextBox" VerticalAlignment="Top" Width="50" />
                </Grid>
            </TabItem>
        </TabControl>
        <RadioButton Content="Hecke algebra" Height="16"
HorizontalAlignment="Right" Margin="0,39,106,0" Name="HeckeRadioButton"
VerticalAlignment="Top" />
        <RadioButton Content="Yokonuma-Hecke algebra" Height="16"
HorizontalAlignment="Right" Margin="0,61,45,0" Name="YokonumaHeckeRadioButton"
VerticalAlignment="Top" IsChecked="True" />
```

```
        <RichTextBox FontFamily="Consolas" FontSize="13" Margin="12,232,12,12"
Name="richTextBox1" VerticalScrollBarVisibility="Auto" />
        <Label Content="Yokonuma-Hecke d" Height="28" IsEnabled="{Binding
ElementName=YokonumaHeckeRadioButton, Path=IsChecked}" Margin="0,133,82,0"
Name="YokonumaLabel" VerticalAlignment="Top" HorizontalAlignment="Right"
Width="116" />
        <TextBox Height="23" HorizontalAlignment="Right" IsEnabled="{Binding
ElementName=YokonumaHeckeRadioButton, Path=IsChecked}" Margin="0,135,12,0"
Name="YokonumaTextBox" VerticalAlignment="Top" Width="50" />
    </Grid>
</Window>
```

3.2.6 MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using PolyLib;
using System.Threading.Tasks;

namespace Braids
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            this.Title = "Braids v" +
System.Diagnostics.FileVersionInfo.GetVersionInfo("Braids.exe").FileVersion +
                " + PolyLib v" +
System.Diagnostics.FileVersionInfo.GetVersionInfo("PolyLib.dll").FileVersion;
        }

        private void ComputeButton_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                Word w = Parser.Parse(WordTextBox.Text);
                if (HeckeRadioButton.IsChecked == true)
                {
                    Polynomial p = w.TraceHecke();
                    richTextBox1.AppendText("\r" + p.ToString() + "\r");
                }
                else if (YokonumaHeckeRadioButton.IsChecked == true)
                {
                    Polynomial p =
w.TraceYokonumaHecke(Convert.ToInt32(YokonumaTextBox.Text));
                    richTextBox1.AppendText("\r" + p.ToString() + "\r");
                }
            }
            catch (Exception ex)
            {
                richTextBox1.AppendText("Calculation suspended. Reason: \"" +
ex.Message + "\"");
```

```
            }
        }

        private void printWord(List<int[]> word)
        {
            for (int i = 0; i < word.Count; i++)
            {
                richTextBox1.AppendText("g" + word[i][0]);
                if (word[i][1] != 1)
                    richTextBox1.AppendText("^" + word[i][1]);
            }
            richTextBox1.AppendText("\r");
        }

        private void ComputeTestButton_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                DateTime dt1 = DateTime.Now;

                List<string> words1 = new List<string>();
                List<string> words2 = new List<string>();

                int count = 10;

                if (Test1RadioButton.IsChecked == true)
                {

                    for (int p = 2; p < count; p++)
                        for (int q = 2; q < count; q++)
                            for (int r = 2; r < count; r++)
                            {
                                if (q != r)
                                {
                                    words1.Add("t1^" + FrameTextBox.Text + "
g1^" + Convert.ToString(2 * p + 1) + " g2^" + Convert.ToString(2 * q) + " g1^"
+ Convert.ToString(2 * r) + " g2^-1");
                                    words2.Add("t1^" + FrameTextBox.Text + "
g1^" + Convert.ToString(2 * p + 1) + " g2^-1" + " g1^" + Convert.ToString(2 *
r) + " g2^" + Convert.ToString(2 * q));
                                }
                            }
                }
                else if (Test2RadioButton.IsChecked == true)
                {
                    for (int a = 2; a < count; a++)
                        for (int b = 2; b < count; b++)
                        {
                            words1.Add("g3g2^-2g3" + Convert.ToString(2 * a +
2) + " g2g3^-1g1^-1g2g1^" + Convert.ToString(2 * b + 2));
                            words2.Add("g3g2^-2g3" + Convert.ToString(2 * a +
2) + " g2g3^-1g1^" + Convert.ToString(2 * b + 2) + "g2g1^-1");
                        }
                }

                DateTime dt2 = DateTime.Now;

                List<string> output = new List<string>();
                richTextBox1.AppendText(words1.Count.ToString() + " pairs of
words to be calculated.\r");

                for (int i = 0; i < words1.Count; i++)
```

```
                {
                    Word w1 = Parser.Parse(words1[i]);
                    Word w2 = Parser.Parse(words2[i]);
                    if (HeckeRadioButton.IsChecked == true)
                    {
                        Polynomial p1 = w1.TraceHecke();
                        Polynomial p2 = w2.TraceHecke();
                        richTextBox1.AppendText("\r" + p1.Equals(p2).ToString()
+ "\r");
                    }
                    else if (YokonumaHeckeRadioButton.IsChecked == true)
                    {
                        Polynomial p1 =
w1.TraceYokonumaHecke(Convert.ToInt32(YokonumaTextBox.Text));
                        Polynomial p2 =
w2.TraceYokonumaHecke(Convert.ToInt32(YokonumaTextBox.Text));
                        richTextBox1.AppendText("\r" + p1.Equals(p2).ToString()
+ "\r");
                        Console.WriteLine("\r" + p1.Equals(p2).ToString() +
"\r");
                    }
                }

                DateTime dt3 = DateTime.Now;

                richTextBox1.AppendText("\r" + "\r");
                richTextBox1.AppendText("Words creation: " + (dt2 -
dt1).TotalSeconds.ToString() + "\r");
                richTextBox1.AppendText("Parsing and calculation: " + (dt3 -
dt2).TotalSeconds.ToString() + "\r");
                richTextBox1.AppendText("Total " + (dt3 -
dt1).TotalSeconds.ToString() + "\r");
            }
            catch (Exception ex)
            {
                richTextBox1.AppendText("Calculation suspended. Reason: \"" +
ex.Message + "\"");
            }
        }

        private void CompareButton_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                Word w1 = Parser.Parse(Word1TextBox.Text);
                Word w2 = Parser.Parse(Word2TextBox.Text);
                if (HeckeRadioButton.IsChecked == true)
                {
                    Polynomial p1 = w1.TraceHecke();
                    Polynomial p2 = w2.TraceHecke();
                    richTextBox1.AppendText("\r" + p1.Equals(p2).ToString() +
"\r");
                }
                else if (YokonumaHeckeRadioButton.IsChecked == true)
                {
                    Polynomial p1 =
w1.TraceYokonumaHecke(Convert.ToInt32(YokonumaTextBox.Text));
                    Polynomial p2 =
w2.TraceYokonumaHecke(Convert.ToInt32(YokonumaTextBox.Text));
                    richTextBox1.AppendText("\r" + p1.Equals(p2).ToString() +
"\r");
                }
```

```
            }
            catch (Exception ex)
            {
                richTextBox1.AppendText("Calculation suspended. Reason: \"" +
ex.Message + "\"");
            }

        }

    }
}
```

4 Βιβλιογραφία

Adams, Colin. *The Knot Book.* n.d.

C# Language Specification (ECMA-334). ECMA, 2006.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition.* The MIT Press, 2002.

Dingle, Brent M. *Designing a Multivariate Polynomial Class - A Starting Point.* 2004.

[Jo] Jones, V. F. R. "Hecke algebra representations of braid groups and link polynomials." *Annals of Mathematics* 126 (1987): 335-388.

[Ju] Juyumaya, J. «Markov trace on the Yokonuma-Hecke algebra.» *J. Knot Theory Ramificaitions* 13 (2004): 25-39.

[JuLa1] Juyumaya, J., and S. Lambropoulou. "An adelic extension of the Jones polynomial." Edited by M. Banagl and D. Vogel. *The mathematics of knots, Contributions in the Mathematical and Computational Sciences* (Springer) Vol. 1 (2009).

[JuLa2] Juyumaya, J., and S. Lambropoulou. "p-adic framed braids." *Topology and tis Applications* 154, no. 8 (2007): 1804-1826.

[JuLa3] Juyumaya, J., and S. Lambropoulou. "p-adic framed braids II (submitted to Advances)." 2009.

[JuLa4] Juyumaya, J., και S. Lambropoulou. «An invariant for singular knots.» *J. Knot Theory Ramifications* 18, αρ. 6 (2009): 825-840.

[KhLe] Khindhawit, Tirasan, and NG Lenhard. "A family of Transversely Nonsimple Knots." 2008.

Leijen, Daan. *Division and Modulus for Computer Scientists.* 2001.

Nagel, Christian, Bill Evjen, Jay Glynn, Karli Watson, and Skinner Morgan. *Professional C# 2008.* 2008.

[OrSh] Orevkov, S. S., and V. V. Shevchishin. "Links, Markov Theorem for Transversal Links." 2001.

Prasolov, V. V., and A. B. Sossinsky. *Translations of Mathematical Monographs, Volume 154 - Knots, Links, Braids and 3-Manifolds.* American Mathematical Society, 1997.

Shoup, Victor. *Μία υπολογιστική εισαγωγή στη θεωρία αριθμών και την άλγεβρα.* Εκδόσεις Κλειδάριθμος, 2007.

[Yo] Yokonuma, T. «Sur la structure des anneaux de Hecke d' un groupe de Chevalley fini.» *C. R. Acad. Sc. Paris* 264 (1967): 344-347.

Παναγίδης, Κωνσταντίνος. *Τι είναι η Θεωρία Κόμβων στα Μαθηματικά.* n.d. http://math.ntua.gr/~sofia/kostas/.