Σχολή Τεχνολογικών Εφαρμογών
Τμήμα Μηχανικών Πληροφορικής



Υλοποίηση αλγορίθμων ελέγχου ισοζύγισης προσημασμένων γράφων

Πτυχιακή εργασία

Καργάκης Μιχαήλ (ΑΜ: 2687)
Εισηγητής: Ξεζωνάκης Ιωάννης

ΗΡΑΚΛΕΙΟ 2014

Ευχαριστώ τους γονείς μου, τον Θεό και τα πακέτα που πηγαινοέρχονται στην κάρτα δικτύου του λάπτοπ μου. Χωρίς αυτά δεν θα ήξερα ούτε τα μισά από τα ήδη λίγα που ξέρω. Ευχαριστώ επίσης τον επιβλέποντα καθηγητή της παρούσας πτυχιακής κ. Ιωάννη Ξεζωνάκη για τις όποιες υποδείξεις, τον κ. Σμυρνάκη για την βοήθεια που δεν αρνήθηκε να μου δώσει όταν του τη ζήτησα αν και εν τέλει δεν κάτσαμε ποτέ να δούμε τι μπορούμε να κάνουμε, τον κ. Zdeněk Letko, ομιλητή του μαθήματος Java Programming Language στο Τεχνολογικό Πανεπιστήμιο του Μπρνο για την υπόδειξη του να χρησιμοποιήσω την Graphviz και τέλος τον Rob Pike και τους υπόλοιπους δημιουργούς της Go, καθώς η συγκεκριμένη γλώσσα είναι ίσως ο κύριος λόγος αυτή τη στιγμή που αγαπάω τον προγραμματισμό και που κατάφερα όχι να τελειώσω αυτή την πτυχιακή αλλά να την φτάσω στο επίπεδο που την έχω φτάσει.

Abstract

In this thesis random signed graphs will be generated and balance checked with the use of already developed algorithms from the bibliography. Such an algorithm has been published on International Journal of Computer Mathematics, named “A Dynamic Programming Algorithm To Test a Signed Graph for Balance”. This algorithm uses a BFS technique to traverse the graph. Respectively, another algorithm using a DFS technique will be implemented.

Σύνοψη

Στην πτυχιακή αυτή θα υλοποιηθούν τυχαίοι προσημασμένοι γράφοι και θα ελεγχθούν ως προς την ισοζύγιση με τη χρήση ήδη αναπτυγμένων αλγορίθμων από την βιβλιογραφία. Ένας τέτοιος αλγόριθμος έχει δημοσιευτεί στο περιοδικό International Journal of Computer Mathematics, με τίτλο “A Dynamic Programming Algorithm to Test a Signed Graph for Balance”. Στον αλγόριθμο αυτόν γίνεται διάσχιση του γράφου χρησιμοποιώντας τεχνική BFS. Θα αναπτυχθεί αντίστοιχος αλγόριθμος για έλεγχο ισοζύγισης του γράφου χρησιμοποιώντας μια τεχνική μορφής DFS.

ΠΕΡΙΕΧΟΜΕΝΑ

Ευχαριστίες.....I

Abstract.....II

Σύνοψη.....II

Περιεχόμενα.....III

1 Εισαγωγή.....1

1.1 Περίληψη.....1
1.2 Κίνητρο.....1
1.3 Σκοπός.....2
1.4 Δομή.....3

2 Μεθοδολογία υλοποίησης.....4

2.1 Αλγόριθμοι.....4
2.1.1 BFS.....6
2.1.2 DFS.....7
2.1.3 Αλγόριθμος δημιουργίας προσημασμένου γράφου.....8
2.2 Θεωρίες.....9
2.2.1 Θεωρία των γράφων.....9
2.2.2 Θεωρία των προσημασμένων γράφων.....9
2.2.3 Θεώρημα δομής.....10
2.2.4 Ισοζύγιση προσημασμένων γράφων.....10
2.2.5 Θεωρία της πολυπλοκότητας.....11
2.3 Μοντέλα.....12
2.3.1 Πίνακας γειτνίασης.....12
2.3.2 Λίστα ακμών.....13
2.3.3 Στοίβα.....14
2.3.4 Ουρά.....14
2.3.5 Δομή EDGE.....16

3 Σχέδιο δράσης εκπόνησης ΠΕ.....17

3.1 State of art.....17
3.1.1 Λειτουργικό σύστημα.....17
3.1.2 Γλώσσα προγραμματισμού.....18
3.1.3 Μεταγλωττιστής.....18
3.1.4 Valgrind.....19
3.1.5 Graphviz & Evince.....21
3.1.6 Ολοκληρωμένο περιβάλλον ανάπτυξης (IDE).....22
3.1.7 Έλεγχος εκδόσεων 22
3.1.8 Make.....23

3.1.9 Συνεχής ολοκλήρωση....24
3.1.10 Συγγραφή αναφοράς....24
3.2 Σημαντικοί στόχοι για την ολοκλήρωση της πτυχιακής....25
3.3 Χρονοδιάγραμμα εργασίας....25

4 Κύριο μέρος ΠΕ....26
4.1 Ανάλυση προβλήματος....26
4.2 Απαιτήσεις....28
4.3 Υλοποίηση....29
4.4 Σχολιασμός....33
4.4.1 types.h....34
4.4.2 test.c και test.h....34
4.4.3 algorithms.c και algorithms.h....35
4.4.4 main.c....37
4.4.5 Γραφική αναπαράσταση....37
4.4.6 Παρατηρήσεις....38

5 Αποτελέσματα....39

Βιβλιογραφία....40

Παράρτημα....42

ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ

Εικόνα 1: *Αναπαράσταση βημάτων BFS*....6
Εικόνα 2: *Αναπαράσταση βημάτων DFS*....7
Εικόνα 3: *Τυχαίος προσημασμένος γράφος*....8
Εικόνα 4: *Διαμερισμένος γράφος*....10
Εικόνα 5: *Διάγραμμα Euler για τα P, NP και NP-Ολοκληρωμένα σετ προβλημάτων*....11
Εικόνα 6: *Πίνακας γειτνίασης ενός προσημασμένου γράφου 10 κόμβων*....12
Εικόνα 7: *Λίστα ακμών (Στέργιος Β. Αναστασιάδης, 2005)*....13
Εικόνα 8: Στοίβα και λειτουργίες στοίβας....14
Εικόνα 9: Ουρά και λειτουργίες ουράς....15
Εικόνα 10: *Δομή EDGE με int πεδία*....19
Εικόνα 11: *Δομή EDGE με uint8_t πεδία*....20
Εικόνα 12: *Figure 7 από Structural Balance (Cartwright & Harary, 1956)*....21
Εικόνα 13: *Κατέβασμα πηγαίου κώδικα από το Github*....28
Εικόνα 14: *Αποσυμπίεση πηγαίου κώδικα*....29
Εικόνα 15: *make test και εκτέλεση των τεστ*....29
Εικόνα 16: *make viz και εκτέλεση του προγράμματος με δυνατότητα γραφικής αναπαράστασης*....30
Εικόνα 17: *Εκτέλεση προγράμματος με τυχαίο γράφο 10 κόμβων*....30
Εικόνα 18: *Γραφική αναπαράσταση γράφου εικόνας 17*....31
Εικόνα 19: *Programming is an art*....33

ΛΙΣΤΑ ΠΙΝΑΚΩΝ

Πίνακας 1: *Οι δέκα γλώσσες που χρησιμοποιούνται περισσότερο σε έργα λογισμικού ανοικτού κώδικα*.....2
Πίνακας 2: *Build history στο Travis CI*.....19

1. ΕΙΣΑΓΩΓΗ

Στον προγραμματισμό το δύσκολο κομμάτι δεν είναι να λύνεις προβλήματα, αλλά να επιλέγεις ποιά προβλήματα θα λύσεις – Paul Graham

1.1 Περίληψη

Οι προσημασμένοι γράφοι δημιουργήθηκαν από τον Frank Harary, Αμερικάνο μαθηματικό, που ήθελε να λύσει ένα πρόβλημα στην κοινωνική ψυχολογία. Έκτοτε έχουν βρει εφαρμογή σε πολλές άλλες επιστημονικές περιοχές.

Η παρούσα πτυχιακή σκοπεύει στην υλοποίηση τυχαίων προσημασμένων γράφων και στον έλεγχο ως προς την ισορροπία τους. Ο έλεγχος αυτός θα πραγματοποιείται από βιβλιογραφικά ήδη ανεπτυγμένους αλγόριθμους που χρησιμοποιούν γνωστές τεχνικές διάσχισης γράφων. Ο ένας από τους δύο αναπτύχθηκε από τον κ. Εμμανουήλ Λουκάκη και έχει δημοσιευτεί στο περιοδικό International Journal of Computer Mathematics, με τίτλο "A Dynamic Programming Algorithm to Test a Signed Graph for Balance". Ο άλλος δημιουργήθηκε από τον κ. Ιωάννη Ξεζωνάκη.

Πιο συγκεκριμένα, αρχικά θα δημιουργήσουμε έναν αλγόριθμο ο οποίος θα αναλάβει την δημιουργία τυχαίων προσημασμένων γράφων, με την πιθανότητα εισαγωγής κάποιων περιορισμών ως προς τους γράφους αυτούς, πχ. ελάχιστος/μέγιστος αριθμός ακμών μεταξύ κόμβων, ελάχιστος/μέγιστος αριθμός κόμβων. Στη συνέχεια θα αναπτύξουμε προγραμματιστικά ήδη βιβλιογραφικά ανεπτυγμένους αλγόριθμους ελέγχου ισοζύγισης που χρησιμοποιούν Αναζήτηση Κατά Πλάτος (**B**readth **F**irst **S**earch) και Αναζήτηση Κατά Βάθος (**D**epth **F**irst **S**earch) ενώ τέλος θα υπάρχει και η δυνατότητα γραφικής αναπαράστασης των γράφων μεσω του προγράμματος Graphviz και πιο συγκεκριμένα του neato.

Η υλοποίηση θα γίνει στην γλώσσα προγραμματισμού C. Ο κώδικας θα γραφτεί με στόχο την αναγνωσιμότητα και την απόδοση ενώ παράλληλα διάφορα εργαλεία και τεχνικές προγραμματισμού θα εφαρμοστούν πάνω στην εργασία και θα αναλυθούν στα περαιτέρω κεφάλαια.

1.2 Κίνητρο

Μέσω της πτυχιακής εργασίας, δίνεται η δυνατότητα στον φοιτητή να εργαστεί πάνω σε ένα project το οποίο πλησιάζει πραγματικές συνθήκες εργασίας. Γι'αυτόν και μόνο τον λόγο θα πρέπει να γίνεται προσεγμένη και έξυπνη επιλογή ως προς το θέμα της πτυχιακής. Η παρούσα εργασία μου δίνει την δυνατότητα να δουλέψω πάνω σε μια γλώσσα προγραμματισμού που αν και πλέον δεν έχει χρησιμοποιείται ευρύτατα επαγγελματικά όπως κάποτε, αποτελεί δυνατό εργαλείο για κάθε προγραμματιστή. Όντας μητρική ενός μεγάλου πλήθους σύγχρονων γλωσσών προγραμματισμού και αν και είναι μια από τις παλαιότερες καθώς εμφανίστηκε στις αρχές του '70 μέσα από τα Bell Labs, η C χρησιμοποιείται ευρέως μέχρι σήμερα ενώ κυριαρχεί στα έργα λογισμικού ανοικτού κώδικα.

Γλώσσα	Αριθμός έργων	Ποσοστό έργων
C	8393	21.2
C++	7632	19.2
Java	5970	15.1
PHP	4433	11.2
Perl	3618	9.1
Python	1765	4.5
Visual Basic	916	2.3
Unix Shell	835	2.1
Assembly	745	1.9
Javascript	738	1.9

Πίνακας 1: *Οι δέκα γλώσσες που χρησιμοποιούνται περισσότερο σε έργα λογισμικού ανοικτού κώδικα*[1]

Επίσης η περιοχή των αλγορίθμων αποτελεί τεράστιο κομμάτι στον προγραμματισμό. Η σχεδίαση τους και δη η βέλτιστη, πάντα ήταν, είναι και θα είναι μεγάλη πρόκληση. Σύμφωνα με τον γνωστό Αμερικάνο επιστήμονα υπολογιστών, Donald Knuth, βασικό χαρακτηριστικό ενός αποτελεσματικού αλγορίθμου είναι όλα τα βήματα και οι πράξεις που εκτελούνται να οδηγούν στην γρηγορότερη λύση του προβλήματος[2].

Η παρούσα πτυχιακή εργασία συνδυάζοντας την όσο το δυνατόν αποτελεσματικότερη ανάπτυξη αλγορίθμων πάνω στη C, σκοπεύει να ασχοληθεί με ένα διόλου τετριμμένο θέμα όπως αυτό του ελέγχου ισοζύγισης προσημασμένων γράφων και να προσθέσει στη βιβλιογραφία ένα χρήσιμο ντοκουμέντο για επερχόμενους μελετητές.

1.3 Σκοπός και στόχοι εργασίας

Στόχος της πτυχιακής είναι η ανάπτυξη αλγορίθμων και η εξέταση τους. Πιο συγκεκριμενα, ο σκοπός της είναι η ανάπτυξη ενός αλγορίθμου δημιουργίας τυχαίων προσημασμένων γράφων και η υλοποίηση αλγορίθμων βάσει της βιβλιογραφικής τους μορφής με τον βέλτιστο τρόπο, εάν είναι δυνατόν, ώστε να ελέγχουν την ισοζύγιση αυτών των γράφων. Ένας προσημασμένος γράφος μπορεί να χρησιμοποιηθέι σε πολλές εφαρμογές και μπορεί να τύχει πολλών ερμηνειών. Μια από αυτές είναι ο κοινωνικός μας περίγυρος. Επίσης εφαρμογή βρίσκει στην οργανική χημεία[3], τη φυσική[3] , τη γεωμετρία και άλλες επιστημονικές περιοχές.

Σκοπός του φοιτητή είναι η εμβάθυνση στην μελέτη και ανάπτυξη αλγορίθμων και δομών δεδομένων. Καθώς όμως η παρούσα εργασία αναπτυσσόταν, διάφορα χρήσιμα εργαλεία χρησιμοποιήθηκαν για την διεκπεραίωση της, εργαλεία που επεκτείνουν τις δυνατότητες του εκάστοτε χρήστη τους πάνω στον πρόγραμματισμό οπότε κρίνεται επιθυμητό η παρουσίαση και ανάλυση τους όπως θα δούμε στο τρίτο κεφάλαιο.

1.4 Δομή

Στα κεφάλαια της παρούσας εργασίας θα αναλύσουμε την διαδικασία που ακολουθήθηκε για να διεκπεραιωθεί η παρούσα πτυχιακή.

Πιο συγκεκριμένα, το πρώτο κεφάλαιο αποτελεί εισαγωγικό καθώς δίνεται μια περίληψη της εργασίας ενώ περιγράφεται το κίνητρο, οι στόχοι και οι σκοποί της εργασίας. Στο δεύτερο κεφάλαιο θα μιλήσουμε για το θεωρητικό μέρος της εργασίας. Ποιά είναι η έννοια του αλγορίθμου, ποιούς αλγόριθμους χρησιμοποιήσαμε, πως δουλεύουν αυτοί και πάνω σε ποιές θεωρίες βασίζεται άμεσα και έμμεσα το θέμα με το οποίο ασχολούμαστε. Στο τρίτο κεφάλαιο αναλύονται τα εργαλεία και οι λόγοι που αυτά χρησιμοποιήθηκαν αλλά και επιλέχθηκαν έναντι αντιστοίχων τους. Γιατί χρησιμοποιήσαμε το Sublime Text αντί κάποιου άλλου editor, με ποιούς compilers μεταγλωττίσαμε το πρόγραμμα μας κλπ. Αναφορά γίνεται επίσης στους στόχους της εργασίας αλλά και στο χρονοδιάγραμμα που ακολουθήσαμε. Στο τέταρτο κεφάλαιο θα αναλύσουμε το πρόγραμμα που δημιουργήσαμε, εξηγώντας όλο τον πηγαίο κώδικα, πως τον οργανώσαμε και πως να τρέξουμε το πρόγραμμα χρησιμοποιώντας τη γραφική αναπαράσταση στα Windows, καθώς είναι λίγο πιο σύνθετο από ότι σε Unix λειτουργικά. Στο πέμπτο και τελευταίο κεφάλαιο παρουσιάζουμε τα αποτελέσματα της εργασίας, τυχόν ελαττώματα αλλά και ποιούς μπορεί να ωφελήσει.

2. ΜΕΘΟΔΟΛΟΓΙΑ ΥΛΟΠΟΙΗΣΗΣ

Αυτό που παρατηρούμε δεν είναι αυτή καθαυτή η φύση, αλλά η φύση όπως εκτίθεται στη μέθοδο με την οποία θέτουμε ερωτήματα – Werner Heisenberg

2.1 Αλγόριθμοι

"Ως αλγόριθμος ορίζεται μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος. Πιο απλά αλγόριθμο ονομάζουμε μία σειρά από εντολές που έχουν αρχή και τέλος, είναι σαφείς και εκτελέσιμες που σκοπό έχουν την επίλυση κάποιου προβλήματος.

Η λέξη αλγόριθμος προέρχεται από μία μελέτη του Πέρση μαθηματικού του 8ου αιώνα μ.Χ. Αλ Χουαρίζμι (Abu Ja'far Mohammed ibn Musa Al-Khwarismi), η οποία περιείχε συστηματικές τυποποιημένες λύσεις αλγεβρικών προβλημάτων και αποτελεί ίσως την πρώτη πλήρη πραγματεία άλγεβρας. Πέντε αιώνες αργότερα η μελέτη μεταφράστηκε στα Λατινικά και άρχιζε με τη φράση "Algorithmus dixit ...." (ο Αλγόριθμος είπε...). Έτσι η λέξη αλγόριθμος καθιερώθηκε αργά τα επόμενα χίλια χρόνια με την έννοια «συστηματική διαδικασία αριθμητικών χειρισμών». Τη σημερινή της σημασία την οφείλει στη γρήγορη ανάπτυξη των ηλεκτρονικών υπολογιστών στα μέσα του 20ου αιώνα." [4]

Χάρη σ'αυτή την υπολογιστική ανάπτυξη έχουμε φτάσει σήμερα να έχουμε δημιουργήσει χιλιάδες διαφορετικούς αλγορίθμους για οποιοδήποτε πρόβλημα επιδέχεται λύση. Από απλές μαθηματικές πράξεις μέχρι πολυσύνθετα συστήματα παραγωγής ενέργειας και από τυχερά παιχνίδια μέχρι καρδιακή υποστήριξη, παντού υπάρχει ένας αλγόριθμος.

Επιπλέον ένας αλγόριθμος πρέπει να πληροί και τα ακόλουθα κριτήρια:

1) **Είσοδος** : Για να λειτουργήσει πρέπει να εισαχθούν Ν δεδομένα από κάποια εξωτερική πηγή (το Ν μπορεί να είναι και μηδέν).

2) **Έξοδος** : Με το τέλος του αλγόριθμου πρέπει να παράγεται τουλάχιστο ένα αντικείμενο σαν αποτέλεσμα .

3) **Καλά ορισμένος** : Η κάθε οδηγία πρέπει να είναι απόλυτα καθορισμένη , κατανοητή και να μην αφήνει κανένα περιθώριο αμφισβήτησης (πχ σε κάποια απόφαση).

4) **Πεπερασμένος** : Να τελειώνει σε πεπερασμένο αριθμό βημάτων .

5) **Αποτελεσματικός** : Κάθε οδηγία του αλγόριθμου πρέπει να είναι απόλυτα βασική και να μπορεί να εκτελεσθεί από μία υπολογιστική μηχανή. Δηλαδή το τρίτο κριτήριο δεν είναι αρκετό : πρέπει η οδηγία επιπλέον να είναι έτσι ώστε να φέρνει κάποιο αποτέλεσμα .

6) **Γενικός** : Εάν είναι δυνατόν , ο αλγόριθμος πρέπει να λύνει μία κλάση προβλημάτων και όχι ένα και μοναδικό πρόβλημα . Για παράδειγμα , καλός είναι ο αλγόριθμος που επιλύει την εξίσωση πρώτου βαθμού ax+b=0 και όχι μία συγκεκριμένη εξίσωση σαν την 5x+6=0 . Τέλος , ένας αλγόριθμος πρέπει να περιγράφεται αναλυτικά και με τέτοιο τρόπο ώστε να είναι απόλυτα κατανοητός ακόμη και σε κάποιον που δεν ξέρει το πρόβλημα που επιλύει.

Ανάλογα με την τεχνική επίλυσης ενός προβλήματος οι αλγόριθμοι διακρίνονται σε :

- Αναδρομικοί (recursive): αλγόριθμοι που χρησιμοποιούν αναδρομικές λύσεις προβλημάτων , πχ πολυώνυμα Hermite, υπολογισμός παραγοντικού , κ. α.
- Διαίρει και Βασίλευε (divide and conquer): επιλύουν το πρόβλημα αναγάγοντάς το σε μικρότερα ανάλογα προβλήματα , πχ quick sort, merge sort, κ. α.
- Άπληστοι (greedy): επιλύουν προβλήματα επιλέγοντας κάθε φορά την τοπικά βέλτιστη λύση προσδοκώντας την συνολικά βέλτιστη , πχ πρόβλημα επιστροφής ρέστων , χρονικός προγραμματισμός , κ. α.
- Δυναμικού Προγραμματισμού (dynamic programming): συνήθως αναδρομικοί αλγόριθμοι οι οποίοι χρησιμοποιούν τις ενδιάμεσα παραγόμενες λύσεις , πχ πολλαπλασιασμός αλυσίδας πινάκων , κ. α.
- Παράλληλοι (parallel): εύρεση λύσης όπου το πρόβλημα δεν λύνεται σειριακά αλλά πολλές σχέσεις του εκτελούνται παράλληλα , πχ πολλά προβλήματα πινάκων , τεχνικές τύπου « διαίρει και βασίλευε », κ. α.

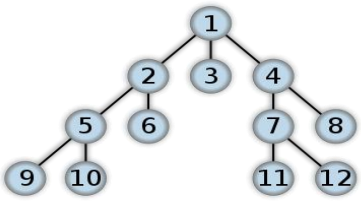
Επίσης , ανάλογα με την λύση που επιτυγχάνουν μπορούν να διακριθούν σε :

- Βέλτιστοι ή Άριστοι (optimal): εύρεση της βέλτιστης λύσης του προβλήματος , πχ επίλυσης μίας εξίσωσης δευτέρου βαθμού.
- Προσεγγιστικοί ή ευρεστικοί (approximation – heuristics): εύρεση « καλών » λύσεων σε άλυτα ή πολύ δύσκολα προβλήματα , πχ χρονοπρογραμματισμός εργασιών , χρωματισμός χάρτη κ. α.” [5]

Στην παρούσα πτυχιακή θα αναπτυχθούν τρεις αλγόριθμοι. Οι δύο υπάρχουν ήδη στην βιβλιογραφία σε μορφή ψευδοκώδικα, μπαίνουν στην κατηγορία των αλγορίθμων δυναμικού προγραμματισμού και χρησιμοποιούν τις γνωστές τεχνικές διάσχισης γράφων BFS και DFS. Ο τρίτος αλγόριθμος τον οποίο θα δημιουργήσουμε εμείς, θα δημιουργεί έναν τυχαίο προσημασμένο γράφο με οποιονδήποτε αριθμό κόμβων και ακμών.

2.1.1 BFS

BFS (Breadth First Search) ή Αναζήτηση Κατά Πλάτος είναι μια τεχνική διάσχισης γράφων όπου η εξερεύνηση ξεκινάει από έναν τυχαίο κόμβο, ανακαλύπτονται όλοι οι γειτονικοί του κόμβοι με κάθε έναν να αποθηκεύεται σε μια ουρά. Εν συνεχεία διασχίζεται η ακμή προς τον πρώτο κόμβο που βρίσκεται στη ουρά δηλαδή τον πρώτο κόμβο που εντοπίστηκε, αλλά δεν έχει επισκεφθεί ο αλγόριθμος.



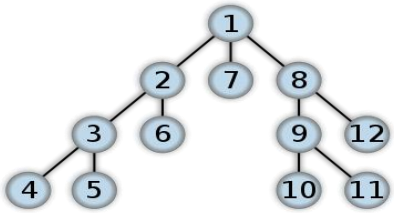
Εικόνα 1: *Αναπαράσταση βημάτων BFS*

Ο αλγόριθμος που χρησιμοποείται, βασιζόμενος στην τεχνική BFS, δημοσιεύτηκε από τον καθηγητή του τμήματος Οικονομικών Επιστημών του Αριστοτελείου Πανεπιστημίου, κ. Εμμανουήλ Λουκάκη το 2003 υπό την ονομασία A Dynamic Programming Algorithm to Test a Signed Graph for Balance[6].

```
Algorithm: Signed_Graph_Balance (H, s, balance)
begin
        visited[s] := true; balance := true; σ[s] := + or -
        for each x ∈ V(G) – {s} do
                visited[x] := false
        endfor
        first := last := 1
        for each x ∈ Adj[s] do
                σ[x] := σ[s] * σ[(s,x)]; last := last + 1;
                Q[last] := x; visited[x] := true
        endfor
        while (first <= last) and (balance := true) do
                x := Q[first]; first := first + 1;
                visited[x] := true
        for each y ∈ Adj[x] do
                if visited[y] := false then
                        visited[y] := true; last := last +1; Q[last] := y;
                        σ[y] := σ[x] * σ[(x,y)]
                else
                if σ[x] * σ[(x,y)] != σ[y] then
                        balance := false
                endif
              endif
        endfor
        endwhile
end
```

2.1.2 DFS

DFS (Depth First Search) ή Αναζήτηση Κατά Βάθος είναι μια άλλη τεχνική διάσχισης γράφων όπου η εξερεύνηση ξεκινάει από έναν τυχαίο κόμβο και συνεχίζει τυχαία σε έναν επόμενο μέχρι να φτάσει σε κάποιον ώστε να μην μπορεί να βρει νέο ανεξερεύνητο κόμβο. Τότε γυρίζει πίσω ψάχνοντας για άλλους νέους (backtracking) μέχρι να γίνει επίσκεψη σε όλους τους κόμβους του γράφου.



Εικόνα 2: *Αναπαράσταση βημάτων DFS*

Παρακάτω σε ψευδοκώδικα ο αλγόριθμος, σχεδιασμένος από τον κ. Ιωάννη Ξεζωνάκη, που μέσω DFS τεχνικής διαπιστώνει αν ένας προσημασμένος γράφος είναι ισοζυγισμένος.

Κάθε κόμβος χαρακτηρίζεται ως V εάν τον έχω ήδη επισκεφθεί και ως NV εάν δεν τον έχω επισκεφθεί. Χαρακτηρίζεται επίσης από το πρόσημο του. Πρόσημο έχουν μόνο οι V κόμβοι, διότι το αποκτούν κατά τη στιγμή της "επίσκεψης" σε αυτούς. Χρησιμοποιούνται δύο στοίβες για την υλοποίηση του αλγόριθμου, η A και η B. Όσο υπάρχουν NV κόμβοι, επιλέγω έναν κόμβο k και τον χαρακτηρίζω αυθαίρετα ως +.

Μετά:

- *Ελέγχω κάθε κόμβο, στον οποίο φθάνω από τον k μέσω ακμής θετικού προσήμου:*
  - *Αν αυτός ο κόμβος είναι NV, παίρνει πρόσημο sign(k)*(+), γίνεται V και τοποθετείται στη στοίβα A.*
  - *Αν είναι V, ελέγχεται εάν έχει πρόσημο +. ΑΝ όχι, υπάρχει λάθος και ο γράφος είναι μη ισοζυγισμένος.*

- *Ελέγχω κάθε κόμβο, στον οποίο φθάνω από τον k μέσω ακμής αρνητικού προσήμου:*
  - *Αν αυτός ο κόμβος είναι NV, παίρνει πρόσημο sign(k)*(-), γίνεται V και τοποθετείται στη στοίβα B.*
  - *Αν είναι V, ελέγχεται εάν έχει πρόσημο -. Αν όχι, υπάρχει λάθος και ο γράφος είναι μη ισοζυγισμένος.*

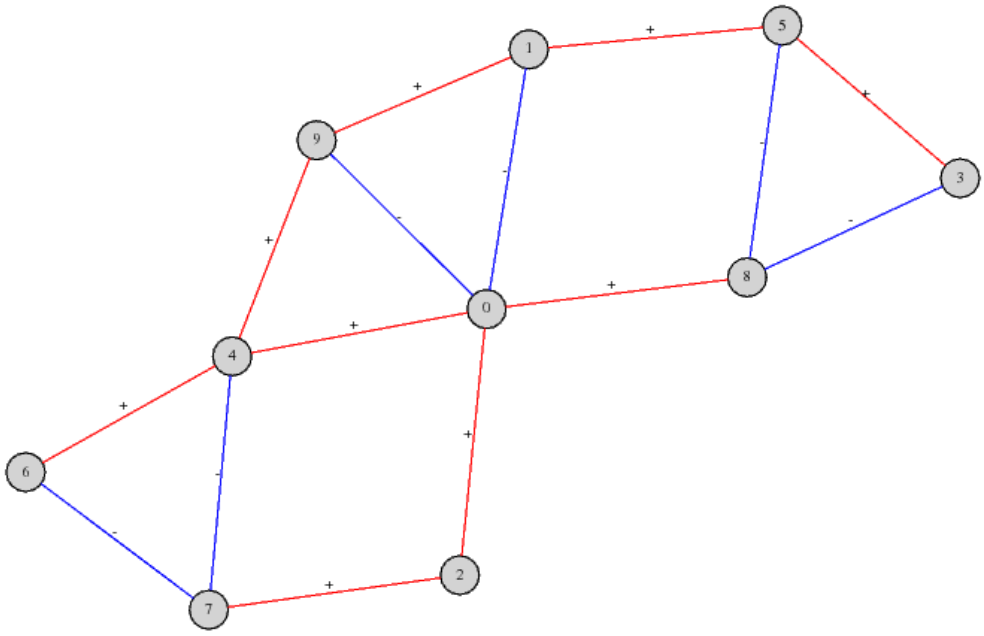
Παίρνω τον επόμενο κόμβο στη στοίβα A και επαναλαμβάνω την παραπάνω διαδικασία, θεωρώντας αυτόν ως κόμβο k. Αν η στοίβα A είναι κενή, παίρνω κόμβους από τη στοίβα B. Η διαδικασία επαναλαμβάνεται μέχρι όλοι οι κόμβοι να είναι V. Αν όλοι οι κόμβοι έχουν σημειωθεί ως V και δεν έχει "κολλήσει" ο αλγόριθμος πουθενά, τότε ο γράφος είναι ισοζυγισμένος.

2.1.3 Αλγόριθμος δημιουργίας προσημασμένου γράφου

Η ύπαρξη ενός προσημασμένου γράφου βασίζεται σε έναν διδιάστατο πίνακα μεγέθους N*N όπου (u,v) μια ακμή και N ο αριθμός των κόμβων που θέλουμε ο γράφος να έχει. Ο συγκεκριμένος πίνακας ονομάζεται πίνακας γειτνίασης (adjacency matrix) ή πίνακας διπλανών κορυφών. Ο αριθμός των ακμών που ορίζουμε να συνδέουν κάθε κόμβο με κάποιον άλλο είναι τυχαίος στο εύρος [2, 6] χωρίς όμως να περιορίζεται εκεί καθώς ένας κόμβος του οποίου ήδη έχουμε ορίσει κάποιες ακμές, μπορεί να συνδεθεί με έναν άλλο στον οποίο ορίζουμε ακμές. Το πρόσημο κάθε ακμής προσδιορίζεται τυχαία.

Ο λόγος που επιλέχθηκε το κατώτο όριο στο συγκεκριμένο εύρος είναι γιατί θέλουμε στον γράφο μας να μην υπάρχουν ακμές με ένα μόνο άκρο (half edges). Ο λόγος που επιλέχθηκε το ανώτατο όριο φαίνεται να εμπεριέχει κάποια αυθαιρεσία, πράγμα που εν πολλοίς είναι σωστό. Όμως, συνήθως σε γράφους που χρησιμοποιούνται στον πραγματικό κόσμο ένας κόμβος έχει ένα μικρό ποσοστό συνδέσεων σε σχέση με το μέγεθος του γράφου. Για παράδειγμα ο μέσος χρήστης στο Facebook έχει γύρω στους 200 φίλους[7] ενώ το Facebook αποτελείται από περισσότερους από 700 εκατομμύρια χρήστες[8] ή ένας μαθητής στο σχολείο κάνει παρέα με 5 άλλους μαθητές την ώρα που η τάξη του συνολικά αποτελείται από 60 άτομα. Θα μπορούσαμε θεωρητικά να είχαμε έως N-1 συνδέσεις αλλά σε έναν τυχαίο γράφο θα υπάρξουν πολλές τέτοιες περιπτώσεις με αποτέλεσμα 1) να μην πλησιάζουμε γράφους πραγματικού κόσμου και 2) σε μεγάλους γράφους η γραφική αναπαράσταση τους να μην είναι πρακτική.

Τέλος όλοι οι κόμβοι του γράφου κατά τη δημιουργία τους θεωρούνται UNVISITED.



Εικόνα 3: *Τυχαίος προσημασμένος γράφος 10 κόμβων*

2.2 Θεωρίες

Στο παρόν κεφάλαιο θα αναφερθούν θεωρίες που σχετίζονται άμεσα με την υλοποίηση της πτυχιακής όπως η θεωρία των γράφων, η θεωρία των προσημασμένων γράφων[9] και το Θεώρημα Δομής (Structure Theorem) του Frank Harary όπως αναφέρεται στο βιβλίο του Fred Roberts, *Discrete Mathematical Models*[10] αλλά και θεωρίες που έχουν να κάνουν με πιθανή εμβάθυνση σε σχετικά θέματα όπως η ισοζύγιση (και όχι απλά ο έλεγχος της) προσημασμένων γράφων και η θεωρία της πολυπλοκότητας.

2.2.1 Θεωρία των γράφων

"Ένα γράφημα (graph) αποτελείται από δύο πράγματα:

i. Ένα σύνολο V=V(G) τα στοιχεία του οποίου ονομάζονται κορυφές (vertices), σημεία (points), ή κόμβοι (nodes) του G.
ii. Ένα σύνολο E=E(G) μη διατεταγμένων ζευγών ξεχωριστών κορυφών που ονομάζονται ακμές (edges) του G.

Όταν θέλουμε να τονίσουμε τα δύο τμήματα του G, συμβολίζουμε ένα τέτοιο γράφημα με το G(V,E). Λέμε ότι οι κορυφές u και v είναι γειτονικές (adjacent) αν υπάρχει ακμή e={u,v}. Στην περίπτωση αυτή, τα u και v ονομάζονται άκρα σημεία (end points) της e, και λέμε ότι η e συνδέει (connects) το u με το v. Επίσης, λέμε ότι η ακμή e προσπίπτει (is incident) στο καθένα από τα άκρα της σημεία u και v.

Τα γραφήματα παριστάνονται με διαγράμματα του επιπέδου με φυσικό τρόπο. Ειδικότερα, κάθε κορυφή v του V παριστάνεται με μια τελεία (ή με έναν μικρό κύκλο), και κάθε ακμή $e=\{v_1, v_2\}$ παριστάνεται με μια καμπύλη που συνδέει τα άκρα της σημεία v_1 και v_2."

2.2.2 Θεωρία των προσημασμένων γράφων

"Προσημασμένος γράφος είναι ένα ζευγάρι (G, σ) που αποτελείται από έναν γράφο G=(V,E) και έναν χάρτη προσήμων σ ανάμεσα στις ακμές E και το σύνολο συμβόλων {+, -}. Ο γράφος μπορεί να έχει επαναληπτικές ακμές (loops) όπου τα δύο άκρα είναι το ίδιο σημείο, πολλές ακμές σε ένα άκρο αλλά και ακμή με ένα μόνο άκρο (half edge) ή και καθόλου άκρα (loose edge). Οι δύο τελευταίοι τύποι ακμών δεν λαμβάνουν πρόσημο ενώ ο λόγος ύπαρξης των ακμών χωρίς άκρα είναι κυρίως για να έχουμε πάντα τη δυνατότητα διαγραφής ακμής[11]. Το πρόσημο ενός κύκλου (ένα σετ ακμών που δημιουργεί κύκλο) ορίζεται από το γινόμενο των προσήμων των ακμών του. Με άλλα λόγια, το πρόσημο του κύκλου είναι θετικό όταν περιέχει ζυγό αριθμό αρνητικών ακμών και αρνητικό όταν περιέχει μονό αριθμό αρνητικών ακμών. Ένας προσημασμένος γράφος ή υπογράφος λέγεται ισοζυγισμένος όταν όλοι οι κύκλοι που περιέχει είναι θετικοί. Η ιδέα των προσημασμένων γράφων εισάχθηκε πρώτη φορά από τον Αμερικάνο μαθηματικό Frank Harary ώστε να χειριστεί ένα πρόβλημα κοινωνικών σχέσεων το 1956[12]." [2]

2.2.3 Θεώρημα δομής

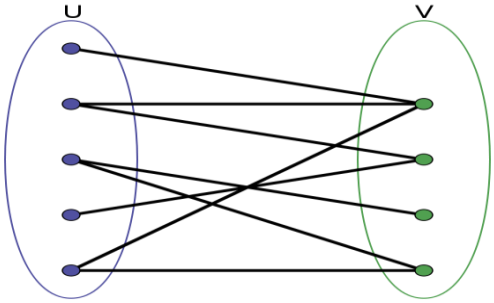
Υποθέτουμε ότι G=(V, E) είναι ένας προσημασμένος γράφος. Οι ακόλουθες δηλώσεις είναι ισοδύναμες:

(a) Ο G είναι ισορροπημένος.
(b) Κάθε κλειστό μονοπάτι (κύκλος) στον G είναι θετικό.
(c) Δύο οποιαδήποτε μονοπάτια μεταξύ ενός κόμβου v και ενός κόμβου u έχουν το ίδιο πρόσημο.
(d) Το σετ κόμβων V μπορεί να χωριστεί σε δύο σετ A και B έτσι ώστε κάθε θετική ακμή συνδέει κόμβους του ίδιου σετ και κάθε αρνητική ακμή συνδέει κόμβους των διαφορετικών σετ.

Η απόδειξη του παραπάνω θεωρήματος υπάρχει στο βιβλίο *Discrete Mathematical Models with applications to social, biological and environmental problems* του Fred Roberts. Πάνω στο συγκεκριμένο θεώρημα βασίστηκε ο κ. Εμμανουήλ Λουκάκης και ανέπτυξε τον αλγόριθμο που λειτουργεί με Αναζήτηση Κατά Πλάτος (BFS) τον οποίο υλοποιούμε προγραμματιστικά στην παρούσα εργασία.

2.2.4 Ισοζύγιση προσημασμένων γράφων

Το πρόβλημα της ισοζύγισης προσημασμένων γράφων ανήκει στην κατηγορία πολυπλοκότητας των μη ντετερμινιστικών πολυωνυμικού χρόνου ολοκληρωμένων προβλημάτων (NP-Complete ή NP-Hard). Οι Frank Huffner, Nadja Beitzler και Rolf Niedermeier στο άρθρο Optimal Edge Deletions for Signed Graph Balance[13] προτείνουν το πρόβλημα του ισορροπημένου υπογράφου (Balanced Subgraph problem) όπου δοσμένου ενός προσημασμένου γράφου G=(V,E), προσπαθούμε να βρούμε έναν ισοροπημένο υπογράφο $G' \subseteq G$ με όσο το δυνατόν περισσότερους κόμβους. Συνεχίζοντας στο ίδιο άρθρο το πρόβλημα αυτό αντικαθίσταται από το πρόβλημα διαγραφής όσο το δυνατόν λιγότερων ακμών (ή αλλαγής προσήμου τους) με αποτέλεσμα την δυνατότητα διαμερισμού του γράφου (Edge Bipartization problem)



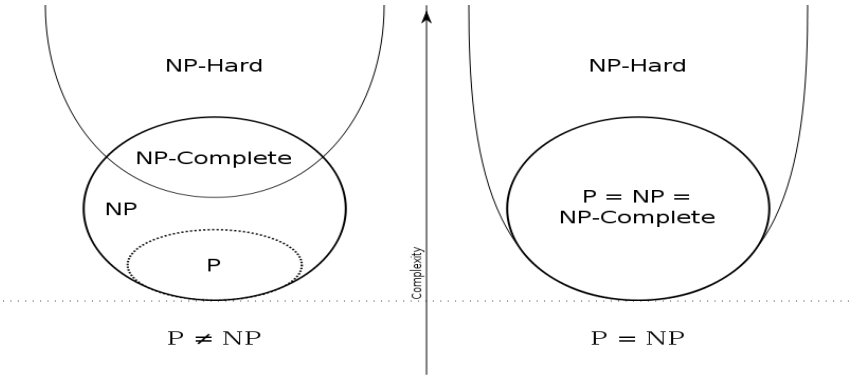
Εικόνα 4: *Διαμερισμένος γράφος*

2.2.5 Θεωρία της πολυπλοκότητας

"Η θεωρία πολυπλοκότητας είναι το μέρος εκείνο της θεωρίας υπολογισμού, το οποίο ασχολείται με την κοστολόγηση των πόρων που απαιτούνται για την αλγοριθμική επίλυση ενός προβλήματος. Επομένως η θεωρία πολυπλοκότητας αποτελεί βασικό δομικό λίθο της ανάλυσης αλγορίθμων και κεντρικό γνωστικό πεδίο της επιστήμης υπολογιστών. Οι συνηθέστεροι πόροι για τους οποίους ενδιαφερόμαστε είναι ο χρόνος, οπότε μιλάμε για τη χρονική πολυπλοκότητα του αλγορίθμου, δηλαδή πόσα «βήματα» χρειάζεται να εκτελέσει ο αλγόριθμος συναρτήσει της εισόδου του, και ο χώρος, οπότε μιλάμε για τη χωρική πολυπλοκότητα, δηλαδή πόσο χώρο (μνήμη) χρειάζεται ο αλγόριθμος συναρτήσει της εισόδου του. Εκτός από αυτούς τους πόρους, κατά περίπτωση, μπορεί να ενδιαφερόμαστε και για άλλους, όπως για παράδειγμα πόσοι παράλληλοι επεξεργαστές χρειάζονται για να λυθεί ένα πρόβλημα παράλληλα.

Η θεωρία πολυπλοκότητας ταξινομεί τα προβλήματα σε κλάσεις(σύνολα) ισοδυναμίας που ορίζουν ότι τα προβλήματα στην ίδια κλάση έχουν την ίδια δυσκολία. Ιδιαιτέρου ενδιαφέροντος στο πλαίσιο αυτό ειναι οι κλάσεις P(Deterministic Polynomial Time) και NP(Non Deterministic Polynomial Time). Γενικά θα μπορούσαμε να πούμε ότι η κλάση P περιλαμβάνει τα περισσότερα προβλήματα της NP. Θα προχωρήσουμε σε πιο τυπικούς ορισμούς οι οποιοι επαλυθεύουν τη διαίσθηση ότι $P \subseteq NP$.Θα πρέπει να τονίσουμε ότι οι κλάσεις P και NP ορίζονται ως προς προβλήματα απόφασης, δηλαδή προβλήματα στα οποία καλούμαστε να απαντήσουμε μια συγκεκριμένη ερώτηση με ΝΑΙ ή ΟΧΙ.

- Η κλάση P περιλαμβάνει όλα εκείνα τα προβλήματα απόφασης, τα οποία επιλύονται από ένα ντετερμινιστικό αυτόματο σε πολυωνυμικό χρόνο.
- Η κλάση NP περιλαμβάνει όλα τα προβλήματα απόφασης που επιλύονται από ένα μη ντετερμινιστικό αυτόματο σε πολυωνυμικό χρόνο. Ένας ισοδύναμος ορισμός της κλάσης NP έχει ως εξής: Η κλάση NP περιλαμβάνει όλα τα προβλήματα απόφασης για τα οποία αν μας δοθεί ένα πιστοποιητικό της απάντησης ΝΑΙ, μπορούμε να επαληθεύσουμε σε πολυωνυμικο χρόνο ότι είναι σωστή." [2]



Εικόνα 5: *Διάγραμμα Euler για τα P, NP και NP-Ολοκληρωμένα σετ προβλημάτων*

2.3 Μοντέλα

Κλείνοντας το δεύτερο κεφάλαιο θα σχολιάσουμε διάφορες δομές δεδομένων και μοντέλα που μας βοήθησαν να δημιουργήσουμε την απαραίτητη συμπεριφορά στον κώδικα μας ώστε να επιτύχουμε τους στόχους μας.

2.3.1 Πίνακας γειτνίασης

"Στα μαθηματικά και στην επιστήμη υπολογιστών, ο πίνακας γειτνίασης χρησιμοποιείται για να αναπαραστήσει τις κορυφές ενός γράφου, οι οποίες συνδέονται με άλλες κορυφές. Συγκεκριμένα, ο πίνακας γειτνίασης ενός πεπερασμένου γράφου G με n κορυφές είναι ο πίνακας διαστάσεων n × n όπου το μη διαγώνιο στοιχείο a_{ij} είναι ο αριθμός ακμών από την κορυφή i στην κορυφή j, και το διαγώνιο στοιχείο a_{ii}, είναι ανάλογα με τη σύμβαση είτε ο αριθμός είτε το διπλάσιο των ακμών (βρόχοι) από την κορυφή i στον εαυτό της. Σε μη κατευθυνόμενους γράφους συνήθως χρησιμοποιείται η δεύτερη σύμβαση (το διπλάσιο του αριθμού των βρόχων), ενώ σε κατευθυνόμενους γράφους κατά κανόνα χρησιμοποιείται η πρώτη σύμβαση. Υπάρχει μοναδικός πίνακας γειτνίασης για κάθε κλάση ισομορφισμού γράφων και είναι διάφορος του πίνακα γειτνίασης οποιασδήποτε άλλης κλάσης ισομορφισμού γράφων. Στην ειδική περίπτωση ενός πεπερασμένου απλού γράφου, ο πίνακας γειτνίασης είναι ένας (0,1)-πίνακας με μηδενικά στοιχεία στη διαγώνιό του. Αν ο γράφος είναι μη κατευθυνόμενος, ο πίνακας γειτνίασης είναι συμμετρικός." [2]

Ο τρόπος με τον οποίο αναπαριστάμε έναν γράφο μέσα στο πρόγραμμα μας είναι μέσω μιας μορφής πίνακα γειτνίασης. Όπως περιγράφεται στην παραπάνω παράγραφο, ένα μη διαγώνιο στοιχείο a_{ij} είναι ο αριθμός ακμών από την κορυφή i στην κορυφή j. Στον δικό μας πίνακα γειτνίασης ένα μη διαγώνιο στοιχείο a_{ij} αναπαριστά μια και μοναδική ακμή ανάμεσα στην κορυφή i και την κορυφή j. Υπάρχει ακόμα άλλος ένας τρόπος αναπαράστασης γράφων, τον οποίο θα αναλύσουμε στο επόμενο κεφάλαιο (2.3.2).

```
$ ./runme
Number of vertices in the graph?
10

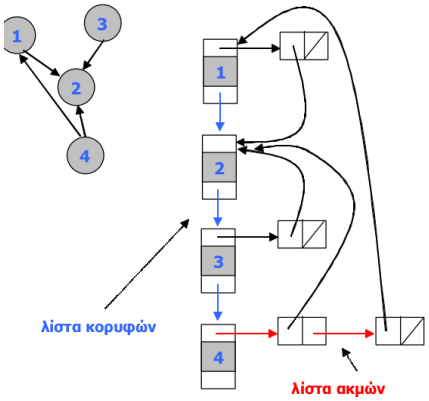
   0    0    0    0    0 NEG    0 POS    0    0
   0    0    0    0    0 POS    0 NEG    0    0
   0    0    0 POS    0    0 POS    0 POS    0
   0    0 POS    0 POS  NEG  POS  POS    0    0
   0    0    0 POS    0 POS  POS    0    0 POS
NEG  POS    0 NEG  POS    0    0 NEG    0 POS
   0    0 POS  POS  POS    0    0    0    0 POS
POS  NEG    0 POS    0 NEG    0    0 POS    0
   0    0 POS    0    0    0    0 POS    0    0
   0    0    0    0 POS  POS  POS    0    0    0
```

Εικόνα 6: *Πίνακας γειτνίασης ενός προσημασμένου γράφου 10 κόμβων*

2.3.2 Λίστα ακμών

"Στη θεωρία των γράφων και της επιστήμη των υπολογιστών, μια λίστα γειτνίασης ενός γράφου είναι μια συλλογή από λίστες, μία για κάθε κορυφή στον γράφο. Κάθε λίστα περιγράφει το σύνολο των γειτόνων της κορυφής της. Υπάρχουν πολλές παραλλαγές αυτής της βασικής ιδέας, διαφέροντας ως προς τις λεπτομέρειες για τον τρόπο με τον οποίο εφαρμόζουν τη συσχέτιση μεταξύ των κορυφών και των συλλογών, στον τρόπο με τον οποίο εφαρμόζουν τις συλλογές, στο αν περιλαμβάνουν τις κορυφές και τις ακμές ή μόνο κορυφές ως πρώτης κατηγορίας αντικείμενα, και τι είδη αντικειμένων χρησιμοποιούνται για να αντιπροσωπεύουν τις κορυφές και τις ακμές.

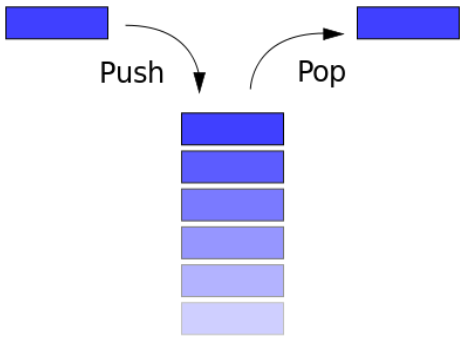
- Μια εφαρμογή που προτείνεται από τον *Guido van Rossum*, δημιουργό της Python, χρησιμοποιεί έναν πίνακα κατακερματισμού για να συνδέσει κάθε κορυφή σε έναν γράφο με μια σειρά από γειτονικές κορυφές. Σε αυτή την αναπαράσταση, μια κορυφή μπορεί να εκπροσωπείται από οποιαδήποτε αντικείμενο που έχει δυνατότητα κατακερματισμού. Δεν υπάρχει ρητή αναπαράσταση των ακμών ως αντικείμενα.
- Ο *Thomas Cormen*, ένας από τους συγγραφείς του *Introduction to Algorithms*, μεταξύ άλλων προτείνει μια εφαρμογή στην οποία οι κορυφές αντιπροσωπεύονται από index αριθμούς[14]. Για την αναπαράσταση τους χρησιμοποιείται ένας πίνακας ταξινομημένος βάσει των αριθμών των κορυφών, στον οποίο το κελί κάθε κορυφής δείχνει σε μια μονά συνδεδεμένη λίστα με τις γειτονικές κορυφές της εν λόγω κορυφής. Σε αυτή την αναπαράσταση, οι κόμβοι της μονά συνδεδεμένης λίστας μπορούν να ερμηνευθούν ως ακμές. Ωστόσο, δεν αποθηκεύουν τα πλήρη στοιχεία για κάθε ακμή (αποθηκεύουν μόνο το ένα από τα δύο άκρα της ακμής) και σε μη κατευθυνόμενους γράφους θα υπάρχουν δύο διαφορετικές συνδεδεμένες λίστες κόμβων για κάθε ακμή.
- Η αντικειμενοστραφής λίστα γειτονικών κορυφών που προτείνεται από τους Goodrich και Tamassia έχει ειδικές κλάσεις κορυφών και ακμών. Κάθε κορυφή έχει μια μεταβλητή που δείχνει σε ένα αντικείμενο συλλογής το οποίο απαριθμεί τις γειτονικές ακμές. Με τη σειρά της, κάθε ακμή δείχνει σε δύο κορυφές ως τα τελικά σημεία της. Αυτή η έκδοση της λίστας γειτνίασης χρησιμοποιεί περισσότερη μνήμη από την έκδοση στην οποία οι γειτονικές κορυφές αναφέρονται άμεσα, αλλά το ότι οι ακμές είναι αντικείμενα επιτρέπει επιπλέον ευελιξία στην αποθήκευση πρόσθετων πληροφοριών σχετικά με τις ακμές." [2]



Εικόνα 7: *Λίστα ακμών (Στέργιος Β. Αναστασιάδης, 2005)*

2.3.3 Στοίβα

"Η στοίβα αποτελεί συγκεκριμένο τύπο από τις δομές δεδομένων (για την επιστήμη των υπολογιστών), όπου τα στοιχεία της επεξεργάζονται ως εξής: Αυτό που εισήχθηκε τελευταίο στη στοίβα αυτό και είναι το πρώτο που εξάγεται προς επεξεργασία από τη στοίβα. Αυτός ο τρόπος επεξεργασίας ονομάζεται LIFO, από την αγγλική ορολογία *Last In Fist Out* δηλ. *Τελευταίο Μέσα Πρώτο Έξω*. Τα στοιχεία αφαιρούνται από τη στοίβα με την αντίθετη σειρά από αυτή που προσθέτονται. Επομένως, τα χαμηλότερα (αυτά που προστέθηκαν πρώτα) στοιχεία είναι τα στοιχεία που μένουν στη στοίβα περισσότερο χρόνο. Κύριες πράξεις είναι η ώθηση (push) στοιχείου στην κορυφή της στοίβας, και η απώθηση (pop) όπου αφαιρούμε το πρώτο στοιχείο από τη δομή, έτσι πετυχαίνουμε τη LIFO δομή. Μια στοίβα δεδομένων μοιάζει με μια στοίβα πιάτα που πλένονται και κατόπιν σκουπίζονται, όπου για το σκούπισμα λαμβάνεται το τελευταίο πιάτο που εισήλθε στη λίστα. Επίσης με στοίβα δεδομένων υλοποιείται και η στοίβα κλήσεων στον υπολογιστή." [2]

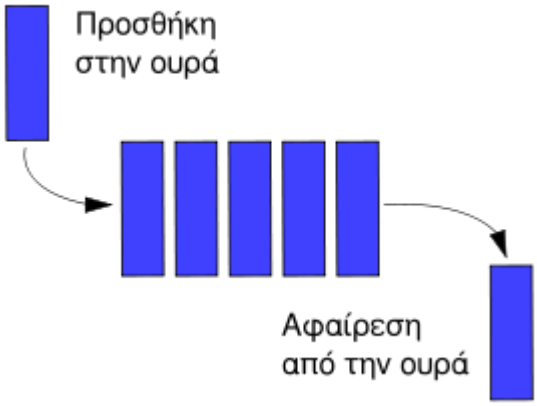


Εικόνα 8: Στοίβα και λειτουργίες στοίβας

Χρησιμοποιούμε δύο στοίβες κατά τη διάρκεια της λειτουργίας του DFS αλγορίθμου, μια όπου αποθηκεύουμε όλες τις κορυφές τις οποίες επισκέπτεται ο αλγόριθμος μέσω θετικής ακμής και μια για όσες επισκέπτεται μέσω αρνητικής ακμής. Στον κώδικα μας έχουμε σχεδιάσει κοινές λειτουργίες στοίβας και για τις δύο στοίβες μας.

2.3.4 Ουρά

"Η ουρά (queue) στην πληροφορική είναι μια δομή δεδομένων με τη μορφή παρατεταμένης συλλογής. Η βασική λειτουργικότητα είναι η εισαγωγή στοιχείων στην πίσω θέση και η εξαγωγή-διαγραφή στοιχείων από την μπροστινή θέση. Με αυτόν τον τρόπο, η ουρά είναι μια FIFO (*First-In-First-Out, Πρώτο-Μέσα-Πρώτο-Έξω*) δομή δεδομένων. Σε μια FIFO δομή δεδομένων, το πρώτο στοιχείο που εισάγεται στην ουρά θα είναι το πρώτο που θα αφαιρεθεί-εξυπηρετηθεί. Οι ουρές χρησιμοποιούνται στον προγραμματισμό υπολογιστών ως δομές δεδομένων. Είναι ο αφηρημένος τύπος δεδομένων μιας ουράς αναμονής στην καθημερινή ζωή (π.χ. μια ουρά εξυπηρέτησης πελατών σε ένα ταμείο). Στις αντικειμενοστραφείς γλώσσες προγραμματισμού υλοποιούνται ως κλάσεις-αντικείμενα. Συνήθεις χρήσεις των ουρών είναι σε κυκλικές προσωρινές μνήμες και στις διασυνδεδεμένες λίστες. Ένα άλλο παράδειγμα χρήσης των ουρών είναι στην λειτουργία της προσωρινής μνήμης (buffer).



Εικόνα 9: *Ουρά* και λειτουργίες ουράς

Η ιδιότητα μέσω της οποίας καθορίζεται μια δομή δεδομένων ως ουρά είναι το γεγονός ότι αυτή επιτρέπει πρόσβαση μόνο στην αρχή και στο τέλος της ουράς. Επιπροσθέτως, τα στοιχεία μπορούν να διαγραφούν μόνο από μπροστά και μπορούν να εισαχθούν μόνο πίσω. Έτσι, μια παρομοίωση που ταιριάζει, και συχνά χρησιμοποιείται για να δώσει μια αναπαράσταση της ουράς, είναι οι ουρές των ταμείων πληρωμής. Άλλα παραδείγματα καθημερινών ουρών είναι άνθρωποι που ανεβαίνουν από κυλιόμενες σκάλες, κομμάτια μηχανών τοποθετημένα σε αλυσίδα συναρμολόγησης ή αυτοκίνητα στη σειρά σε ένα βενζινάδικο.

Σε κάθε περίπτωση, το πρόσωπο ή το αντικείμενο στην αρχή της ουράς είναι το πρώτο που θα φύγει, ενώ στο τέλος της ουράς είναι αυτό που θα φύγει τελευταίο. Κάθε φορά που το πρόσωπο ή το αντικείμενο τελειώνει την δουλειά του, εκείνο φεύγει από την ουρά από μπροστά. Αυτό αναπαριστά τη διαδικασία "dequeue" (εξαγωγή). Κάθε φορά που ένα πρόσωπο ή ένα αντικείμενο μπαίνουν στην ουρά αναμονής, εισέρχονται από το τέλος της ουράς και αυτό αναπαριστά τη διαδικασία "enqueue" (εισαγωγή). Η συνάρτηση "size" επιστρέφει το μήκος της γραμμής και η συνάρτηση "empty" θα επέστρεφε true μόνο αν δεν υπήρχε κανείς στη γραμμή.

Θεωρητικά, ένα χαρακτηριστικό της ουράς είναι ότι δεν έχει συγκεκριμένο μέγεθος. Ασχέτως από το πόσα στοιχεία περιέχονται ήδη, ένα νέο στοιχείο μπορεί πάντα να εισαχθεί. Στην περίπτωση που η ουρά είναι άδεια δεν μπορεί να γίνει εξαγωγή "dequeue" στοιχείου πριν εισαχθεί κάποιο νέο.

Μια πρακτική υλοποίηση ουράς γίνεται συνήθως με δείκτες στην οποία το όριο μεγέθους περιορίζεται από την διαθέσιμη μνήμη που έχει ο υπολογιστής. Ένας άλλος τρόπος υλοποίησης της ουράς είναι σε μια δομή δεδομένων σε σταθερό μέγεθος εκχωρημένης μνήμης. Ο όρος υπερχείλιση μιας ουράς συμβαίνει κατά την προσπάθεια να εισαχθεί ένα στοιχείο σε μια γεμάτη ουρά, ενώ η υποχείλιση συμβαίνει κατά την προσπάθεια διαγραφής-εξαγωγής ενός στοιχείου από μια άδεια ουρά.” [2]

2.3.5 Δομή EDGE

Τα λεγόμενα από το 2.3.1 κεφάλαιο στοιχεία σε έναν πίνακα γειτνίασης θα μπορούσαν απλά να είναι ακέραιοι αριθμοί. Δυστυχώς στην περίπτωση μας χρειαζόμαστε κάτι πιο πολύπλοκο από έναν απλό πίνακα ακεραίων ώστε να αναπαραστήσουμε έναν προσημασμένο γράφο μέσω ενός πίνακα γειτνίασης γι'αυτό χρησιμοποιούμε μια δομή δεδομένων στην οποία αποθηκεύουμε τρεις ακεραίους και έχουμε ονομάσει *EDGE*. Από τους τρεις ακεραίους της δομής μας, ένας μας διευκρινίζει το πρόσημο της ακμής που αναπαριστάται από το συγκεκριμένο στοιχείο, ένας υποδεικνύει αν ο εκάστοτε αλγόριθμος αναζήτησης πέρασε από την συγκεκριμένη ακμή και ο τελευταίος χρησιμοποιείται από τους αλγόριθμους αναζήτησης για να ελέγχεται η ισορροπία στους γράφους μας.

3. ΣΧΕΔΙΟ ΔΡΑΣΗΣ ΕΚΠΟΝΗΣΗΣ ΠΕ

Δώστε μας τα εργαλεία και εμείς θα τελειώσουμε τη δουλειά – Winston Churchill

Σε αυτό το κεφάλαιο θα αναλυθούν το περιβάλλον και τα μέσα με τα οποία διεκπεραιώθηκε η πτυχιακή και γιατί κάποια προτιμήθηκαν έναντι αντιστοίχων τους. Για την ολοκλήρωση της έρευνας τεχνολογίας αιχμής (state of art) χρειάστηκε τόσο βιβλιογραφική αναζήτηση όσο και η ουσιαστική χρήση όλων των εργαλείων. Επίσης θα γίνει αναφορά στους σημαντικότερους στόχους που ήταν απαραίτητοι για την ολοκλήρωση της πτυχιακής και στο χρονοδιάγραμμα που ακολουθήθηκε.

3.1 State of Art

Το παρόν κεφάλαιο θα ασχοληθεί με τα εργαλεία με τα οποία διεκπεραιώθηκε η παρούσα πτυχιακή. Όχι απαραίτητα τεχνολογία αιχμής καθώς πολλές φορές τέτοιου είδους χαρακτηρισμοί δεν αντικατοπτρίζουν ποιά τεχνολογία ή ποιό εργαλείο είναι το καλύτερο για μια συγκεκριμένη δουλειά αλλά τεχνολογία και εργαλεία που κάνουν τον εκάστοτε χρήστη τους παραγωγικό. Έτσι το δικό μας state of art χαρακτηρίζεται από εργαλεία που ίσως να αποτελούν τεχνολογία αιχμής, ίσως όμως και όχι. Κοινός πολλαπλασιαστής είναι πως η παραγωγικότητα που προσφέρεται είναι de facto.

3.1.1 Λειτουργικό σύστημα

"Λειτουργικό σύστημα ονομάζεται στην επιστήμη της πληροφορικής το λογισμικό του υπολογιστή που είναι υπεύθυνο για τη διαχείριση και τον συντονισμό των εργασιών, καθώς και την κατανομή των διαθέσιμων πόρων. Το λειτουργικό σύστημα παρέχει ένα θεμέλιο, ένα μεσολαβητικό επίπεδο λογικής διασύνδεσης μεταξύ λογισμικού και υλικού, διαμέσου του οποίου οι εφαρμογές αντιλαμβάνονται εμμέσως τον υπολογιστή. Μια από τις κεντρικές αρμοδιότητες του λειτουργικού συστήματος είναι η διαχείριση του υλικού, απαλλάσσοντας έτσι το λογισμικό του χρήστη από τον άμεσο και επίπονο χειρισμό του υπολογιστή και καθιστώντας ευκολότερο τον προγραμματισμό τους." [2]

Υπάρχουν θεμελιώδεις διαφορές ανάμεσα στις δύο μεγαλύτερες οικογένειες λειτουργικών συστημάτων, αυτές των Windows και των Unix και ο προγραμματισμός ανάμεσα τους αναπόφευκτα διαφέρει. Το θέμα της παρούσας εργασίας όμως δεν έχει κάποια ιδιομορφία και ο κώδικας μπορεί να τρέξει με τα ίδια αποτελέσματα σε όλες τις πλατφόρμες. Ενώ το λειτουργικό σύστημα δεν παίζει κανένα ρόλο στον κώδικα μας, άξιο αναφοράς είναι πως τον έχουμε αναπτύξει ώστε να γίνεται δυναμική δέσμευση και αποδέσμευση μνήμης, κάτι που μπορεί να χαρακτηριστεί ως λειτουργία του λειτουργικού συστήματος, μιας και σε σχεδόν όλες τις σύγχρονες γλώσσες προγραμματισμού, η δέσμευση και αποδέσμευση μνήμης γίνεται από τον GC (Garbage Collector) της γλώσσας σε συνεργασία με το εκάστοτε λειτουργικό σύστημα. Το κύριο μέρος του προγράμματος μας (έλεγχος ισσοροπίας) υλοποιήθηκε σε Windows 7, η ανάπτυξη του κομματιού για την γραφική αναπαράσταση των γράφων υλοποιήθηκε σε Crunchbang Linux, μια παραλλαγή της οικογένειας των Debian λειτουργικών ενώ όλο το πρόγραμμα έτρεξε εκτός των προαναφερθέντων επιπλέον σε Zorin OS, από την οικογένεια των Ubuntu.

3.1.2 Γλώσσα προγραμματισμού

"Η C είναι μια διαδικαστική γλώσσα προγραμματισμού γενικής χρήσης, η οποία αναπτύχθηκε αρχικά, μεταξύ του 1969 και του 1973, από τον Ντένις Ρίτσι στα εργαστήρια AT&T Bell Labs για να χρησιμοποιηθεί για την ανάπτυξη του λειτουργικού συστήματος UNIX. Όπως οι περισσότερες διαδικαστικές γλώσσες προγραμματισμού που ακολουθούν την παράδοση της ALGOL, η C έχει δυνατότητες δομημένου προγραμματισμού και επιτρέπει τη χρήση αναδρομής (αλλά όχι και εμφωλευμένων συναρτήσεων), ενώ, ο στατικός ορισμός του τύπου των μεταβλητών που επιβάλλει, προλαμβάνει πολλά σφάλματα κατά την χρήση τους. Ο σχεδιασμός της περιλαμβάνει δομές που μεταφράζονται αποδοτικά σε τυπικές εντολές μηχανής (machine instructions) και εξ αιτίας αυτού χρησιμοποιείται συχνά σε εφαρμογές που παλιότερα γράφονταν σε συμβολική γλώσσα (assembly language). Αυτό ακριβώς το χαρακτηριστικό της, που έχει σαν συνέπεια και την αυξημένη ταχύτητα εκτέλεσης των εφαρμογών που γράφονται σε αυτή, καθώς και το γεγονός ότι είναι διαθέσιμη στα περισσότερα σημερινά λειτουργικά συστήματα, συνέβαλε κατά πολύ στην καθιέρωση της και την χρήση της για ανάπτυξη λειτουργικών συστημάτων και λοιπών προγραμμάτων συστήματος (system software), αλλά και απλών εφαρμογών.

Η C συγκαταλέγεται πλέον στις πιο ευρέως χρησιμοποιούμενες γλώσσες προγραμματισμού όλων των εποχών και πολλές νεώτερες γλώσσες έχουν επηρεαστεί άμεσα ή έμμεσα από αυτήν, συμπεριλαμβανομένων των C++, C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, καθώς και του κελύφους C (C shell) του Unix. Κάποιες από αυτές τις γλώσσες έχουν επηρεαστεί κυρίως στη σύνταξη τους, με το σύστημα τύπων, τα μοντέλα δεδομένων και το νόημα των εκφράσεων τους να διαφέρουν σημαντικά από την C. Η C++, ειδικά, ξεκίνησε σαν προεπεξεργαστής της C, αλλά έχει εξελιχθεί πλέον σε μια αντικειμενοστραφή γλώσσα, που αποτελεί υπερσύνολο της C." [2]

Η C δεν έχει πολλές βιβλιοθήκες και αυτό την κάνει σχετικά απλή για να ξεκινήσει μαζί της κάποιος. Από όλα τα χαρακτηριστικά που έχει, εμείς χρησιμοποιήσαμε δυναμική δέσμευση μνήμης όπου ήταν δυνατό, συγκεκριμένες δομές δεδομένων όπως στοίβες και ουρές, μονοδιάστατους και δισδιάστατους πίνακες, ενώ η χρήση «καθολικών» μεταβλητών και δεικτών κρατήθηκε όσο το δυνατόν μικρότερη. Η έκδοση της γλώσσας πάνω στην οποία δουλεύουμε, δεν έχει σημασία μιας και τα χαρακτηριστικά και οι βιβλιοθήκες που χρησιμοποιούμε δεν έχουν αλλάξει από την τελευταία, C99 (1999), στην καινούργια, C11 (2011).

3.1.3 Μεταγλωττιστής

"Μεταγλωττιστής ή μεταφραστής (στα αγγλικά compiler) ονομάζεται ένα πρόγραμμα που μετατρέπει/μεταφράζει κείμενο γραμμένο σε μια γλώσσα προγραμματισμού (πηγαία γλώσσα) σε μια άλλη γλώσσα προγραμματισμού (τη γλώσσα στόχο). Το κείμενο της εισόδου ονομάζεται πηγαίος κώδικας (source code) και η έξοδος του προγράμματος αντικειμενικός κώδικας (object code). Ο όρος «μεταγλωττιστής» χρησιμοποιείται κυρίως για προγράμματα που μεταφράζουν μια γλώσσα προγραμματισμού υψηλού επιπέδου σε μια γλώσσα χαμηλότερου επιπέδου (όπως η συμβολική γλώσσα ή η γλώσσα μηχανής)." [2]

Ο πιο γνωστός μεταγλωττιστής που θα μπορούσαμε να χρησιμοποιήσουμε όταν προγραμματίζουμε σε C δεν είναι άλλος από τον GCC (GNU Compiler Collection). Δημιουργήθηκε το 1987 υποστηρίζοντας ως πρώτη γλώσσα τη C και στη συνέχεια

δημιουργήθηκαν front-ends για C++, Objective-C, Java, Fortran, Go και άλλες γλώσσες[15]. Για την ιστορία, η έκδοση που χρησιμοποιούμε για compilation είναι η 4.8.1. Επιπλέον, ενσωματώθηκε υποστήριξη για το Travis CI (3.1.9), μέσω του οποίου ο κώδικας συντάσσεται σε GCC αλλά και σε Clang.

GCC	Clang	Build
1 min 16 sec	1 min 12 sec	6
52 sec	56 sec	7
51 sec	1 min 26 sec	8
46 sec	49 sec	9
41 sec	49 sec	10

Πίνακας 2: *Build history στο Travis CI*

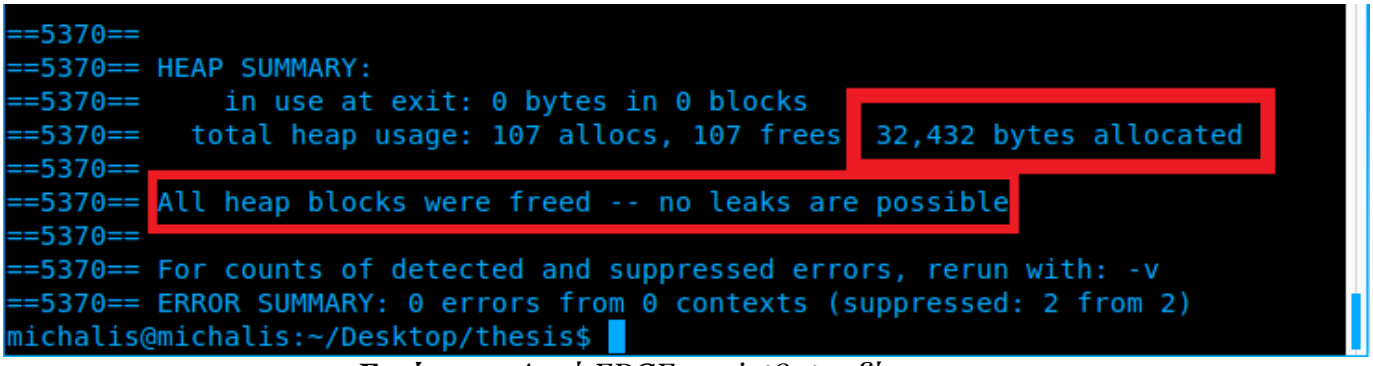
Στον πίνακα 2 μπορούμε να δούμε μέρος του ιστορικού δημιουργίας στο Travis CI. Με μια πρόχειρη ματιά μπορούμε να βγάλουμε συμπέρασμα πως το πρόγραμμα μας «προτιμά» περισσότερο τον GCC από τον Clang, δεν ενδεικνύεται όμως σε καμία περίπτωση για συμπεράσματα ως προς το ποιός μεταγλωττιστής είναι γρηγορότερος, καθώς ο καθένας έχει τα πλεονεκτήματα και μειονεκτήματα του ενώ τέτοιου είδους συγκρίσεις ξεφεύγουν από το θέμα της εργασίας αλλά και του επιπέδου γνώσης μας.

3.1.4 Valgrind

Το Valgrind είναι ένας C/C++ profiler και debugger για τα Linux, που μας επιτρέπει μέσω του περιβάλλοντός του να τρέξουμε οποιοδήποτε πρόγραμμα και να δούμε αν έχουμε διαρροές μνήμης. Ο κύριος λόγος που το χρησιμοποιήσαμε όμως είναι για συγκρίνουμε δύο διαφορετικές εκδόσεις του προγράμματος μας, μια όπου η δομή EDGE χρησιμοποιούσε int πεδία και αυτή που τελικά κρατήσαμε όπου πλέον έχουμε αντικαταστήσει τα int με uint8_t πεδία, δηλαδή 8-bit ακέραιους, το μικρότερο μέγεθος ακέραιου που μπορούμε να χρησιμοποιήσουμε στη C μέσω της stdint.h.

```
==5322==
==5322== HEAP SUMMARY:
==5322==     in use at exit: 0 bytes in 0 blocks
==5322==   total heap usage: 107 allocs, 107 frees, 122,432 bytes allocated
==5322==
==5322== All heap blocks were freed -- no leaks are possible
==5322==
==5322== For counts of detected and suppressed errors, rerun with: -v
==5322== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
michalis@michalis:~/Desktop/thesis$
```

Εικόνα 10: *Δομή EDGE με int πεδία*



Εικόνα 11: *Δομή EDGE με uint8_t πεδία*

Θεωρητικά γνωρίζαμε από πριν τι αποτελέσματα επρόκειτο να δούμε αλλά πάντα η θεωρία πρέπει να γίνεται και πράξη. Στην περίπτωση όπου η δομή που χρησιμοποιείται μακράν περισσότερο από κάθε άλλη δομή ή απλό τύπο μέσα στο πρόγραμμα μας, EDGE, είχε int πεδία, το συνολικό μέγεθος του προγράμματος με 100 κόμβους ανερχόταν στα 122,432 bytes ή γύρω στα 120 kB. Στην περίπτωση όπου πλέον έχουμε αντικαταστήσει τους int με uint8_t, το μέγεθος του προγράμματος μας [εννοείται πάλι με 100 κόμβους], πέφτει στα 32,432 bytes δηλαδή σχεδόν υποτετραπλασιάζεται. Στα θετικά επίσης το ότι δεν έχουμε καμία διαρροή μνήμης καθώς όλοι μας οι πίνακες δεσμεύονται και αποδεσμεύονται δυναμικά, κάτι που επίσης γνωρίζαμε θεωρητικά.

Το Valgrind έχει τη φήμη του καλύτερου profiler για την C και η αλήθεια είναι πως δεν δυσκολευτήκαμε καθόλου να το χρησιμοποιήσουμε. Για την ιστορία το τρέξαμε πάνω στο Zorin OS, ένα λειτουργικό της οικογένειας των Ubuntu, μέσω VirtualBox. Δυστυχώς στα Windows δεν υπάρχει κάποιο port του Valgrind και τα αντίστοιχα C profilers που βρήκαμε δεν ήταν τόσο αξιόπιστα.

Για να το εγκαταστήσουμε και να το χρησιμοποιήσουμε [σε Linux περιβάλλον πάντα] απλά χρησιμοποιούμε τις δύο παρακάτω εντολές:

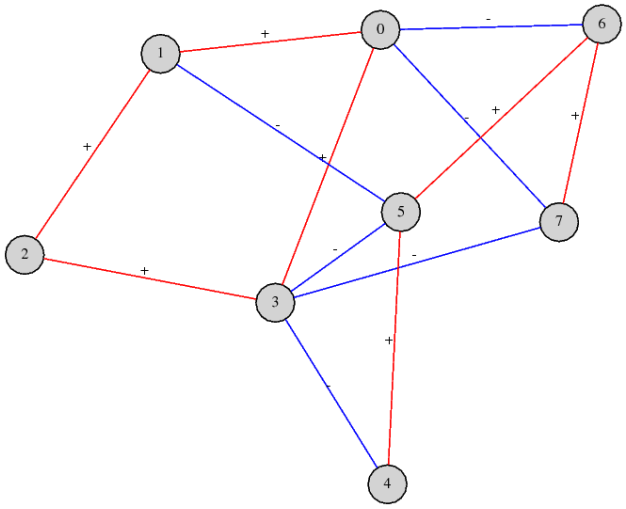
$ sudo apt-get install valgrind
$ valgrind –tool=memcheck ./ektelesimo-arxeio-mas

3.1.5 Graphviz & Evince

Η Graphviz[16] είναι μια βιβλιοθήκη για την γραφική αναπαράσταση γράφων. Ξεκίνησε ως έρευνητικό project μέσα από την AT&T και πλέον είναι ελεύθερο πρόγραμμα ανοιχτού κώδικα. Ως μέρος της βιβλιοθήκης υπάρχουν αρκετά εργαλεία (dot, neato, fdp, sfdp, twopi, circo) που χρησιμοποιούνται με διαφορετικές δυνατότητες το καθένα. Στην περίπτωση μας καταλληλότερο κρίθηκε το neato μιας και οι ακμές των γράφων που διαχειριζόμαστε δεν έχουν κατεύθυνση ενώ οι γράφοι που δημιουργούμε δεν είναι τόσο μεγάλοι (αν και θεωρητικά οι αλγόριθμοι ισοζύγισης μας μπορούν να εφαρμοστούν σε γράφους οποιουδήποτε μεγέθους).

Ως είσοδο σε οποιοδήποτε εργαλείο της Graphviz, πρέπει να έχουμε πάντα ένα .dot αρχείο με κατάλληλη σύνταξη (γλώσσα DOT[17]). Στα σχόλια που υπάρχουν στον πηγαίο κώδικα τον οποίο αναλύουμε στο τέταρτο κεφάλαιο, γίνεται αναφορά και στην DOT γλώσσα με παράθεση σε πηγή για περαιτέρω πληροφορίες.

Παράδειγμα γραφικής αναπαράστασης ενός γράφου δημιουργημένου από το neato είδαμε νωρίτερα στο κεφάλαιο 2.1.3 (Εικόνα 3). Χρησιμοποιούμε μπλε για την αναπαράσταση ακμών με αρνητικό πρόσημο, κόκκινο για ακμές με θετικό πρόσημο ενώ οι κόμβοι αναπαραστούνται ως γκρι κύκλοι. Ακολουθεί ένα δεύτερο παράδειγμα γράφου ο οποίος είναι ο γράφος* που χρησιμοποιήθηκε στο άρθρο των Cartwright και Harary, (Figure 7), στο οποίο συγκεκριμένο άρθρο είχαμε για πρώτη φορά τον ορισμό ενός προσημασμένου γράφου



Εικόνα 12: *Figure 7 από Structural Balance (Cartwright & Harary, 1956)*

Τέλος για την παρουσίαση της γραφικής αναπαράστασης χρησιμοποιείται το πρόγραμμα Evince. Επιλέχθηκε ανάμεσα σε διάφορους document viewers (Adobe Reader, Foxit, Nitro, κ.α.) καθώς είναι το πιο ελαφρύ, ενώ είναι και ελεύθερο λογισμικό ανοιχτού κώδικα.

**οπτικά δεν είναι οι ίδιοι καθώς όταν χρησιμοποιούμε την Graphviz, δεν μπορούμε να ελέγξουμε τις θέσεις στις οποίες θα αναπαρασταθεί κάποιος κόμβος. Είναι όμως οι ίδιοι όσον αφορά τις συνδέσεις μεταξύ των κόμβων. Ουσιαστικά δηλαδή δεν έχουν διαφορά.*

3.1.6 Ολοκληρωμένο περιβάλλον ανάπτυξης (IDE)

Στη C προσφέρονται διάφορα IDE όπως τα DevCpp, CodeBlocks και Visual Studio αλλά κανένα τους δεν αποτελεί τρομερή επιλογή. Άλλωστε η C ποτέ δεν βασίστηκε πάνω σε κάποιο IDE όπως πχ. η Java με τα IntelliJ, NetBeans και Eclipse. Γι'αυτό το λόγο εξ αρχής, ψάξαμε για κάποιον απλό κειμενογράφο. Αν και τα Windows δεν προσφέρουν προεγκατεστημένο κάποιον αξιοπρεπή κειμενογράφο όπως πχ. τα Crunchbang Linux έχουν τα gedit και Geany, η επιλογή του Sublime Text για την υλοποίηση του κώδικα της παρούσας πτυχιακής δεν ήταν καθόλου δύσκολη[18] . Ένα editor που με τα κατάλληλα plugins μεταμορφώνεται σε IDE, highlighting με χρώματα που κάνουν το πρόγραμμα ευάγνωστο, αυτοματισμούς όπως δημιουργία παρενθέσεων, αντικατάσταση λέξεων, πρόβλεψη κώδικα ή ακόμα και στατική ανάλυση, μικρογραφία όλου του προγράμματος και άλλες λεπτομέρειες, το Sublime Text έχει πλέον καθιερωθεί ως ο βασικός μας editor όσον αφορά τον προγραμματισμό.

3.1.7 Έλεγχος εκδόσεων

Μια πρακτική προγραμματισμού που τα τελευταία χρόνια έχει αποδειχθεί ως κυρίαρχη είναι ο έλεγχος εκδόσεων (version control) . Ο προγραμματιστής ανεβάζει σε ένα απομακρυσμένο χώρο (remote repository) τον κώδικα του και κάθε αλλαγή που κάνει σε αυτόν, καταγράφεται ως ιστορικό. Έτσι όταν ο κώδικας βρεθεί σε ένα σημείο που δεν τρέχει, ή για οποιονδήποτε άλλο λόγο, ο προγραμματιστής έχει τη δυνατότητα να επιστρέψει σε μια παλιότερη έκδοση άκοπα. Επίσης το συγκεκριμένο σύστημα έχει αποδειχθεί αναπόσταστο κομμάτι σε projects όπου συμμετέχουν πολλά άτομα καθώς προσφέρει πολλές άλλες δυνατότητες (branching, merging, κλπ). Το πρόγραμμα για έλεγχο εκδόσεων που χρησιμοποιήθηκε στην παρούσα εργασία είναι το git ενώ ο κώδικας φιλοξενείται στο Github. Για να δημιουργήσουμε το παρόν project στο Github και να το συνδέσουμε μέσω του git σε ένα τοπικό directory του υπολογιστή μας χρειαζόμαστε λογαριασμό στο Github, δημιουργία ενός νέου project από το γραφικό περιβάλλον του Github, εγκατάσταση του git και τις παρακάτω απλές εντολές σε κάποιο terminal:

```
$ mkdir thesis  # md thesis στο cmd των Windows
$ cd thesis
$ git init
$ git remote add origin https://github.com/kargakis/signed-graph-balance-check.git
```

Στη συνέχεια μπορούμε να δουλεύουμε πάνω στο τοπικό directory μας και όποιες αλλαγές κάνουμε να τις «σπρώχνουμε» στο Github.

```
# πχ. αλλάξαμε προσθέσαμε κάποια σχόλια στον κώδικα της main μας
$ git add main.c
$ git commit –m "added comments"
$ git push origin master
```

Στην εντολή git add μπορούμε να δώσουμε ως παραμέτρους παραπάνω από ένα αρχείο ή και wildcards. Το –m στο git commit σημαίνει ότι το σχόλιο που ακολουθεί θα εμφανίζεται ως το μήνυμα της αλλαγής μας. Μπορούμε να γράψουμε οτιδήποτε θέλουμε, με περιορισμό στο πλήθος των χαρακτήρων, αλλά κατάλληλο είναι το μήνυμα

μας να περιγράφει περιληπτικά την αλλαγή που κάναμε στον κώδικα μας ώστε είτε εμείς είτε άλλοι που θα διαβάσουν τον κώδικα μας να ξέρουν γιατί έγινε αυτή η αλλαγή και να μην χρειάζεται να ψάχνουμε όλο τον κώδικα για να την κατανοήσουμε. Τέλος αλλαγές στον κώδικα μας μπορούν να γίνουν και από το γραφικό περιβάλλον του Github. Για να ενημερωθεί όμως το τοπικό μας directory για αυτές τις αλλαγές θα πρέπει να τις «τραβήξουμε» μέσω του git.

```
# καθώς βρισκόμαστε στο τοπικό μας directory
$ git pull origin master
```

Υπάρχουν και άλλες εντολές του git που κρίνονται απαραίτητες σε projects όπου συμμετέχουν πολλά άτομα αλλά εμείς θα σταματήσουμε εδώ καθώς οι παραπάνω εντολές είναι αρκετές για χρήση σε προσωπικά projects.

3.1.8 Make

Ο τρόπος με τον οποίο είχαμε οργανώσει τον κώδικα στην αρχή του project ήταν απλούστατος: ένα .c αρχείο το οποίο περιείχε όλο τον κώδικα και μια απλή κλήση στον μεταγλωττιστή ώστε να δημιουργήσουμε ένα εκτελέσιμο αρχείο και να τρέξουμε το πρόγραμμα μας. Όσο όμως οι γραμμές κώδικά αυξάνονταν και έφτασαν να είναι γύρω στις 650 πλέον, εμφανίστηκε η ανάγκη για μια πιο σπονδυλωτή (modularity) και ευάγνωστη οργάνωση του κώδικα. Γι'αυτό το λόγο, το ένα .c αρχείο έσπασε σε τρία .c και τέσσερα .h, τα οποία αναλύουμε στο τέταρτο κεφάλαιο. Για να μεταγλωττιστούν και να συνδεθούν μεταξύ τους όμως όλα αυτά τα αρχεία, πρέπει να χρησιμοποιήσουμε το make. Μέσω ενός makefile, η δημιουργία του επιθυμητού εκτελέσιμου αρχείου είναι τόσο απλή όσο η κλήση του μεταγλωττισή πάνω σε ένα μονό .c αρχείο.

makefile

```
all: default clean

default: main.o algorithms.o
	$(CC) -o $(TARGET_NAME) main.o algorithms.o

viz: vizmode.o algorithms.o
	$(CC) -o $(TARGET_NAME) main.o algorithms.o
	$(CLEANUP)

test: testmode.o algorithms.o test.o
	$(CC) -o $(TARGET_NAME) main.o algorithms.o test.o
	$(CLEANUP)

main.o: main.c algorithms.h types.h
	$(CC) -c main.c

algorithms.o: algorithms.c algorithms.h types.h
	$(CC) -c algorithms.c

vizmode.o: main.c algorithms.h types.h
	$(CC) -c main.c -D VIZ

clean:
	$(CLEANUP)
```

Στα περιεχόμενα του makefile που παρουσιάζουμε παραπάνω παραλείπονται οι δηλώσεις μεταβλητών γι'αυτό τις εξηγούμε εδώ. CC είναι ο μεταγλωττιστής μας, στην συγκεκριμένη περίπτωση ο gcc, TARGETNAME είναι το όνομα που θέλουμε να έχει το εκτελέσιμο αρχείο μας και CLEANUP είναι μερικές εντολές που βοηθούν στην διαγραφή των object αρχείων μετά το τέλος του linking όπου και έχει δημιουργηθεί το εκτελέσιμο αρχείο. Όταν ένα makefile εκτελεστεί χωρίς να έχουν δοθεί παραμέτροι στο make, τότε η εκτέλεση ξεκινάει από το πρώτο label που θα βρεθεί, στην περίπτωση μας το all. Για να εκτελεστεί το all πρέπει να εκτελεστούν τα default και clean labels. Για να εκτελεστεί το default label απαιτούνται τα object αρχεία των main και algorithms, για να παραχθούν αυτά χρειάζονται τα αντίστοιχα .c και .h αρχεία που υπάρχουν στα αντίστοιχα labels και πάει λέγοντας. Αν δοθεί παράμετρος στο make μας, πχ. make viz τότε η εκτέλεση ξεκινάει από το viz label.

3.1.9 Συνεχής ολοκλήρωση

Το Travis CI είναι ένα περιβάλλον συνεχούς ολοκλήρωσης (continuous integration). Μέσω του κατάλληλου setup, κάποιος μπορεί να συνδέσει τον λογαριασμό του στο Github με την εν λόγω υπηρεσία και να εντάξει όποιο project του θέλει για συνεχή ολοκλήρωση. Ουσιαστικά αυτό που γίνεται, είναι ότι σε κάθε αλλαγή που γίνεται στον κώδικα ενός project στο Github, σχεδόν αμέσως στο Travis αυτό το project περνάει από τη φάση του compilation και επιστρέφεται μήνυμα αν υπάρχουν λάθη ή όχι. Επίσης υπάρχει η δυνατότητα, όπως είδαμε ήδη (3.2), να προσθέσουμε περισσότερους από έναν compilers και να βγάλουμε ακόμα περισσότερα συμπεράσματα και για τον κώδικα μας. Παρακάτω ο κώδικας YAML που απαιτείται να υπάρχει μέσα στο directory του project μας, ώστε το Travis να κάνει την δουλειά του σύμφωνα με τις ανάγκες μας.

.travis.yaml

```
language: c

before_install:
 - sudo apt-get update
 - sudo apt-get install graphviz evince

compiler:
 - clang
 - gcc

script: make test && make viz
```

Αν δεν ορίσουμε κάποιο συγκεκριμένο *script*, τότε το Travis τρέχει *./configure && make && make test*. Εμείς θέλουμε να τρέξουν τα τεστ πρώτα και μετά όλο μας το πρόγραμμα (όπως θα δούμε αργότερα, δεν τρέχει όλο με ένα απλό *make*) γι'αυτό τρέχουμε με *make test && make viz*.

3.1.10 Συγγραφή αναφοράς

Η αναφορά ξεκίνησε πάνω στη σουίτα ελεύθερου λογισμικού Libre Office και συγκεκριμένα στο LibreOffice Writer και ολοκληρώθηκε στο Microsoft Word. Η γραμματοσειρά που χρησιμοποιήθηκε για την συγγραφή των κεφαλαίων είναι η Georgia ενώ για την παρουσίαση του κώδικα χρησιμοποιείται η Arial. Όλες οι υπόλοιπες ιδιότητες κειμένου επιλέχθηκαν βάσει του οδηγού εκπόνησης πτυχιακών.

3.2 Σημαντικοί στόχοι για την ολοκλήρωση της πτυχιακής

Βασικότερος στόχος της παρούσας πτυχιακής είναι η σχεδίαση των αλγορίθμων ελέγχου ισοζύγισης προσημασμένων γράφων. Από εκεί και πέρα υπάρχουν πολλά που πρέπει να γίνουν και ακόμα περισσότερα που μπορούν να γίνουν μέχρι το τέλος τα οποία συνοψίζουμε παρακάτω.

- Έρευνα state of art
- Βιβλιογραφική έρευνα πάνω στο πρόβλημα
- Ανάπτυξη τεχνικού κομματιού πτυχιακής
    a. Δημιουργία αλγορίθμων αναζήτησης
    b. Δημιουργία αλγορίθμου δημιουργίας τυχαίων γράφων
- Έλεγχος λειτουργίας (testing)
- Αποσφαλμάτωση (debugging)
- Βελτιστοποίηση (profiling)
- Αναγνωσιμότητα κώδικα (refactoring)
- Συγγραφή αναφοράς εργασίας
- Δημιουργία περιληπτικής παρουσίασης
- Υποβολή αίτησης παρουσίασης εργασίας
- Προετοιμασία παρουσίασης
- Παρουσίαση

3.3 Χρονοδιάγραμμα εργασίας

Το χρονοδιάγραμμα που ακολουθήθηκε βάσει των παραπάνω στόχων έχει ως εξής:

Απρίλης 2013: Έρευνα state of art, βιβλιογραφική έρευνα πάνω στο πρόβλημα
Μαϊος 2013 – Ιούνιος 2013: Ανάπτυξη τεχνικού κομματιού πτυχιακής
Ιούλιος 2013 – Απρίλιος 2013: Testing, debugging, profiling, refactoring
Φεβρουάριος 2014 – Μάϊος 2014: Συγγραφή αναφοράς, δημιουργία παρουσίασης
Μαϊος 2014 – Ιούνιος 2014: Υποβολή αίτησης, προετοιμασία παρουσίασης, παρουσίαση

4. ΚΥΡΙΟ ΜΕΡΟΣ ΠΕ

Κάθε γλώσσα προγραμματισμού έχει έναν τελεστή βελτιστοποίησης. Στην C ο τελεστής αυτός είναι το '//'

Το κεφάλαιο αυτό ασχολείται με το κύριο μέρος της πτυχιακής δηλαδή με την ανάλυση του προβλήματος, την σχεδίαση της λύσης και την υλοποίηση της λύσης για το θέμα με το οποίο ασχολείται η εργασία.

4.1 Ανάλυση προβλήματος

Το πρόβλημα με το οποίο ερχόμαστε αντιμέτωποι και θα αναλύσουμε στο παρόν κεφάλαιο έχει να κάνει με την ανάπτυξη των αλγορίθμων που με κατάλληλες τεχνικές αναζήτησης θα ελέγχουν την ισορροπία ενός προσημασμένου γράφου. Ο τρόπος που πλέον ενδείκνυται πάνω σε προγραμματιστικά προβλήματα δεν περιλαμβάνει μόνο τον προγραμματισμό της λύσης μας αλλά και ένα σωρό άλλες τεχνικές που στόχο έχουν την διασφάλιση αλλά και την προγραμματιστική βελτιστοποίηση του προγράμματος μας. Σε γενικές γραμμές η λύση μας ξεκινάει με τον προγραμματισμό, και συνεχίζει με το τεστάρισμα του κώδικα, την βελτιστοποίηση και αναδιοργάνωση του κώδικα όπου κρίνεται απαραίτητο και τέλος ξανά το τεστάρισμα του.

Όσον αφορά τον **προγραμματισμό**, οι αλγόριθμοι αναζήτησης που θα εφαρμόσουμε είναι ήδη βιβλιογραφικά ανεπτυγμένοι άρα αυτό που μας μένει είναι να τους προγραμματίσουμε προσεκτικά βάσει της βιβλιογραφικής τους μορφής. Αφού καταφέρουμε να τους αναπτύξουμε, υπάρχουν πολλοί τρόποι ώστε να συνεχίσουμε την εργασία μας.

Ειδικότερα, οι αλγόριθμοί μας χρειάζονται κάποια δεδομένα (γράφοι) ως είσοδο οπότε αντί να δημιουργούμε με το χέρι διάφορους γράφους (κάτι το οποίο θα κάνουμε κατά τη διάρκεια του testing) επιθυμητό κρίνεται να δημιουργήσουμε έναν αλγόριθμο δημιουργίας τυχαίων προσημασμένων γράφων. Το ότι δεν ακολουθούμε μια βιβλιογραφική περιγραφή εδώ, όπως στους προηγούμενους αλγόριθμους, μας δίνει περισσότερη ελευθερία ως προς την σχεδίαση. Υπάρχουν συγκεκριμένοι περιορισμοί που θέτουμε στον αλγόριθμο δημιουργίας. Ο πιο σημαντικός είναι ότι περιορίζουμε τις ελάχιστες αρχικές συνδέσεις ενός κόμβου να είναι δύο, ενώ ανάλογα με το μέγεθος του γράφου μπορεί να φτάσουν μέχρι και τις έξι αρχικές συνδέσεις. Μπορεί να φαντάζει μικρός ο αριθμός των έξι συνδέσεων, ειδικά σε μεγάλους γράφους των 1000, 500 ή και 100 κόμβων ακόμα, αλλά πρέπει να λάβουμε υπ' όψιν τις δύο ακόλουθες παραμέτρους:

1) Ο πιο πάνω περιορισμός έχει να κάνει με τις <u>αρχικές</u> συνδέσεις του κάθε κόμβου, τις συνδέσεις που εμείς ορίζουμε τυχαία. Όταν ορίσουμε έναν αριθμό συνδέσεων σε έναν κόμβο πχ. τρεις συνδέσεις τότε στο παιχνίδι μπαίνουν και οι άλλοι τρεις κόμβοι που θα συνδεθούν με τον κόμβο από τον οποίο ξεκινάμε τις συνδέσεις και τους οποίους επιλέγουμε εντελώς τυχαία. Αυτούς τους κόμβους είτε ήδη τους έχει προσπελάσει ο αλγόριθμος δημιουργίας, με αποτέλεσμα να έχουν σίγουρα τουλάχιστον άλλες δύο συνδέσεις πάνω τους είτε δεν τους έχει προσπελάσει ακόμα. Την τελευταία περίπτωση μέχρι και μερικές εκδοχές του αλγορίθμου μας πριν την τελική, την αντιμετωπίζαμε και αυτό που κάναμε όταν ο αλγόριθμος έφτανε σε

κόμβο που είχε ήδη συνδέσεις είναι να αφαιρούμε τον αριθμό αυτών των συνδέσεων από τον τυχαίο ορισμένο αριθμό συνδέσεων που παράγει ο αλγόριθμος και είτε να συμπληρώνει τη διαφορά των κόμβων είτε αν η διαφορά ήταν αρνητική (άρα θα υπήρχαν ήδη περισσότερες συνδέσεις από τον τυχαία ορισμένο αριθμό) απλά να μην έκανε τίποτα και να συνέχιζε στον επόμενο κόμβο. Στην εκδοχή που έχουμε κρατήσει πλέον, αυτή η περίπτωση δεν αντιμετωπίζεται με αποτέλεσμα ο αριθμός συνδέσεων να είναι εντελώς τυχαίος για κάθε κόμβο ενώ μένουμε κοντά και στην δεύτερη παράμετρο σύμφωνα με την οποία

2) Σε γράφους πραγματικού κόσμου το πλήθος συνδέσεων ενός κόμβου είναι ελάχιστο σε σχέση με το μέγεθος του γράφου. Παραδείγματα σχετικά έχουμε δώσει ήδη στο 2.1.3.

Έχοντας πλέον ολοκληρώσει και τους τρεις αλγόριθμους της εργασίας μας, έχουμε ολοκληρώσει το κύριο μέρος της. Από εδώ και πέρα μπορούν να γίνουν πολλά πράγματα επιπλέον. Ένα το οποίο επιλέγουμε να κάνουμε είναι η γραφική αναπαράσταση των γράφων μας. Η κατάλληλη βιβλιοθήκη για αυτή την δουλειά δεν είναι άλλη από την Graphviz μέσω της οποίας μπορούμε να έχουμε μια πιο αισθητική άποψη των γράφων μας (Εικόνα 3) από ό,τι αποτελούν οι πίνακες γειτνίασης (Εικόνα 6).

Αφού τελειώσουμε με τον προγραμματισμό της λύσης μας, έρχεται η ώρα που πρέπει να επιβεβαιώσουμε ότι όλα λειτουργούν καθώς πρέπει. Και πιο συγκεκριμένα, θέλουμε να ξέρουμε ότι οι αλγόριθμοι αναζήτησης όντως μπορούν να βρουν αν ένας ισορροπημένος γράφος είναι όντως ισορροπημένος ή το αντίστροφο. Όλη αυτή η διαδικασία, ονομάζεται **testing** και μάλιστα τα τελευταία χρόνια έχει γιγαντωθεί η ανάγκη για testing σε όλο το φάσμα του προγραμματισμού καθώς βοηθάει πολύ στην γρήγορη αποσφαλμάτωση. Δυστυχώς σε πολλές γλώσσες προγραμματισμού, συμπεριλαμβανομένης της C, η δυνατότητα για τεστάρισμα του κώδικα, εξαρτάται από τη χρήση ενός εξωτερικού τρίτου προγράμματος (framework) το οποίο προσθέτει επιπλέον φόρτο στην καμπύλη μάθησης. Μετά από αρκετή έρευνα καταλήξαμε στην χρήση του Unity[19], ένα testing framework πάνω στη C. Ενώ μάθαμε να το χρησιμοποιούμε για τις ανάγκες της παρούσας πτυχιακής, στο τέλος αποφασίσαμε να χρησιμοποιήσουμε ενσωματωμένο testing στο πρόγραμμα μας το οποίο θα ήταν αρκετά απλό να υλοποιηθεί.

Για τις ανάγκες της βελτιστοποίησης χρησιμοποιήσαμε το Valgrind σε Linux τα οποία τρέξαμε μέσω Virtual Box. Ο λόγος που κάνουμε βελτιστοποίηση στο πρόγραμμα μας έχει να κάνει κυρίως με την μνήμη που καταλαμβάνει. Με μια απλή αλλαγή των πεδίων της δομής που χρησιμοποιείται περισσότερο μέσα στον κώδικα και δεν είναι άλλη από την δομή EDGE (2.3.5) καταφέρνουμε να εξοικονομήσουμε χώρο (Εικόνα 10 και Εικόνα 11) ο οποίος σε γράφο μεγάλου μεγέθους είναι πολύ σημαντικός.

Τέλος όσον αφορά το περιβάλλον χρήστη του προγράμματος μας, πέρα από το αποτέλεσμα της Graphviz (ένα .ps αρχείο το οποίο ανοίγει αυτόματα καθώς εκτελείται το πρόγραμμα –για περισσότερες πληροφορίες βλ. 4.5), δεν δόθηκε επιπλέον προσοχή και πιο συγκεκριμένα για την κατασκευή κάποιου γραφικού περιβάλλοντος, κυρίως γιατί αυτό θα προέβλεπε επιπλέον φόρτο μάθησης άσχετης με το αντικείμενο της εργασίας, ενώ ακόμα περαιτέρω τα γραφικά περιβάλλοντα φέρνουν επιπλέον περιπλοκότητα σε μια εφαρμογή. Θεωρούνται επίσης ιδιαίτερα σημαντικές η απλότητα και η αμεσότητα που προσφέρεται από ένα command-line εργαλείο.

4.2 Απαιτήσεις

Στον παρόν κεφάλαιο θα σχολιαστούν σύντομα οι απαιτήσεις για το πρόγραμμα μας που χρειάζονται από το σύστημα αλλά και οι απαιτήσεις που χρειάστηκαν ώστε να ολοκληρωθεί όλη η εργασία.

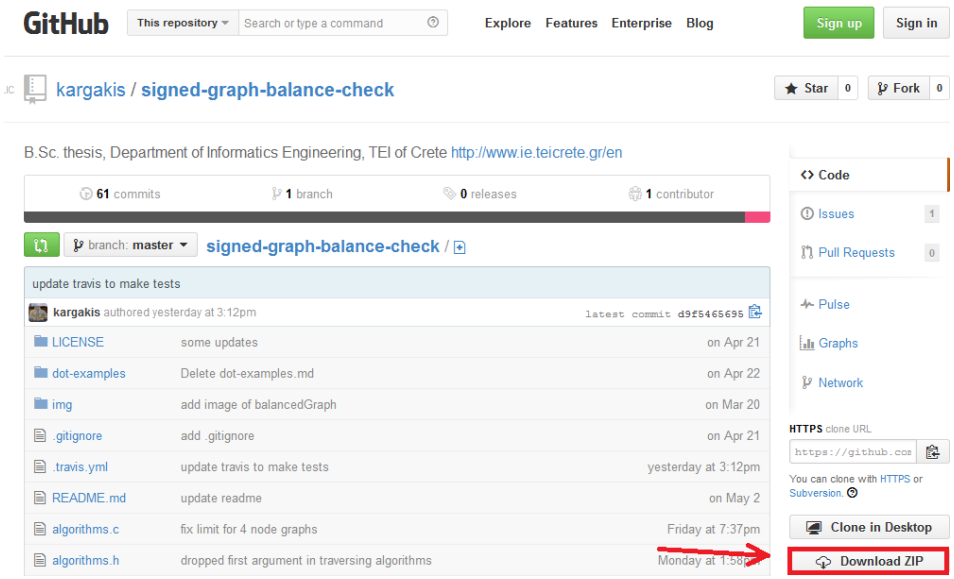
Κατ' αρχάς, σχετικά με το σύστημα, ο χρήστης επιβάλλεται να έχει εγκατεστημένο έναν C μεταγλωττιστή και το εργαλείο **make** ώστε να δημιουργήσει το απαραίτητο εκτελέσιμο αρχείο. Επίσης για γραφική αναπαράσταση των γράφων μέσω της Graphviz, χρειάζεται να είναι εγκατεστημένα η **Graphviz** και ένα πρόγραμμα προβολής PostScript αρχείων όπως το Adobe Reader, το FoxIt ή το **Evince** που εμείς χρησιμοποιούμε. (Αν κάποιος θέλει να χρησιμοποιήσει κάποιο διαφορετικό του Evince, θα πρέπει να αλλάξει στον πηγαίο κώδικα στη main.c στην τελευταία κλήση της system το evince με το όνομα του εκτελέσιμου αρχείου του προγράμματος επιλογής του – για περισσότερες πληροφορίες βλ. 4.5). Πέρα από τις απαιτήσεις γύρω από το λογισμικό που πρέπει να είναι εγκατεστημένο, δεν υπάρχει καμία απαίτηση όσον αφορά το υλικό του συστήματος, πέρα από την ύπαρξη αρκετής ελεύθερης μνήμης φυσικά, αναλόγως πάντα και με το μέγεθος των γράφων τους οποίους θέλει ο εκάστοτε χρήστης να εξερευνήσει.

Όσον αφορά τις απαιτήσεις που υπάρχουν ώστε να ολοκληρωθεί η συγκεκριμένη εργασία αυτές είναι βασικές ικανότητες προγραμματισμού στη C, γνώση δομών δεδομένων και αλγορίθμων και πιο συγκεκριμένα κατανόηση στους γράφους, τις στοίβες και τις ουρές αλλά και στους αλγόριθμους τους οποίους θα δημιουργήσουμε (BFS και DFS).

4.3 Υλοποίηση

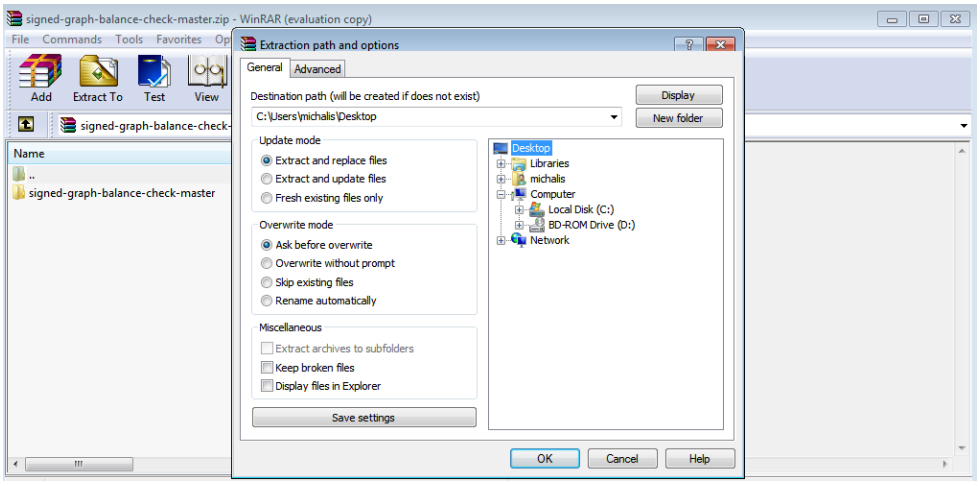
Οι απαιτήσεις για το απαραίτητο λογισμικό που χρειάζεται το πρόγραμμα μας σχολιάζονται στο προηγούμενο κεφάλαιο. Στο παρόν κεφάλαιο θα σχολιάσουμε το κατέβασμα, την εγκατάσταση και το περιβάλλον της εφαρμογής. Ας υποθέσουμε ότι είμαστε ένας χρήστης με περιέργεια είτε πάνω στους προσημασμένους γράφους γενικότερα είτε στην παρούσα εργασία ειδικότερα έχει εγκαταστήσει όλο το απαραίτητο λογισμικό. Τα βήματα που πρέπει να ακολουθήσουμε στην παρούσα φάση είναι:

1. Κατέβασμα πηγαίου κώδικα από τον λογαριασμό μας στο Github



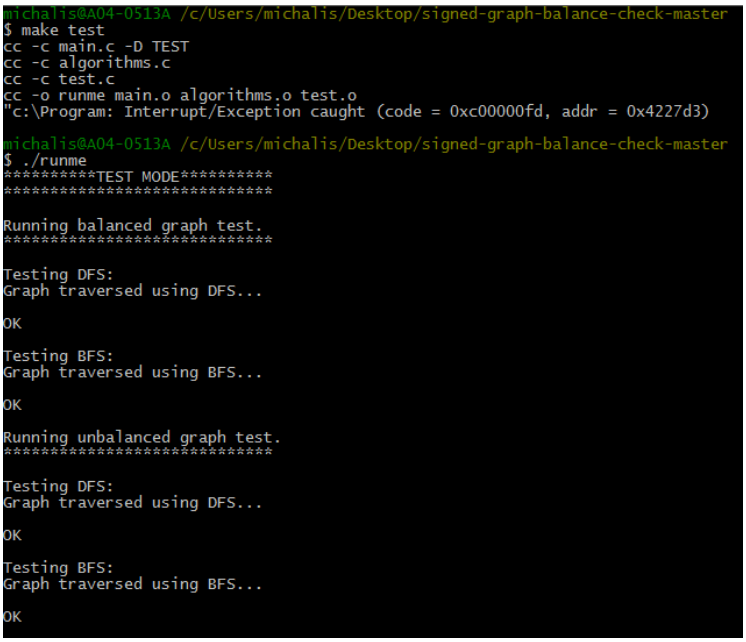
Εικόνα 13: *Κατέβασμα πηγαίου κώδικα από το Github*

2. Αποσυμπίεση (θεωρούμε πως ένα πρόγραμμα αποσυμπίεσης υπάρχει σε κάθε σύστημα χρήστη που έχει δείξει ενδιαφέρον για ένα προχωρημένο προγραμματιστικό θέμα όπως αυτό της ισορροπίας προσημασμένων γράφων) :



Εικόνα 14: *Αποσυμπίεση πηγαίου κώδικα*

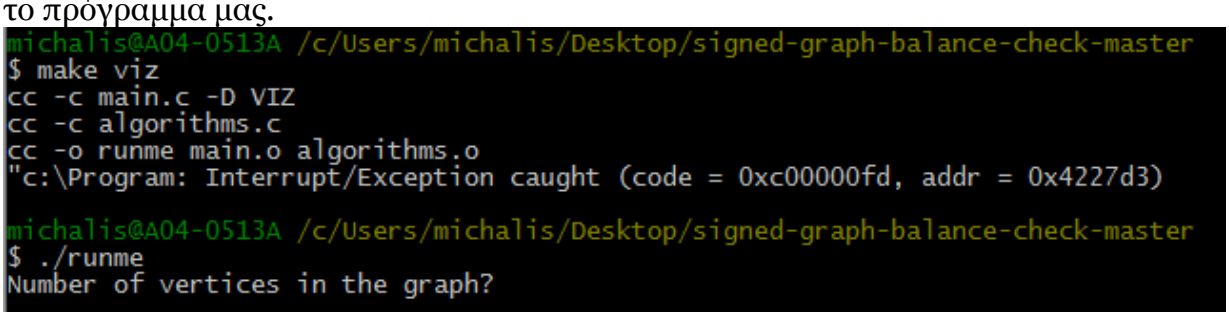
3. Αφού βρεθούμε μέσα στο αρχικό directory της εργασίας μέσω ενός terminal (συνιστάται σε χρήστες Windows να χρησιμοποιήσουν κάποιο εκ των Cygwin, Gitbash, κ.α. Unix terminals αντί του cmd), δημιουργούμε και τρέχουμε τα τεστ ώστε να ελέγξουμε αν λειτουργούν σωστά οι αλγόριθμοι αναζήτησης (OK μήνυμα). Αυτό γίνεται χρησιμοποιώντας το *make* εργαλείο και πιο συγκεκριμένα την εντολή *make test*.



Εικόνα 15: *make test και εκτέλεση των τεστ*

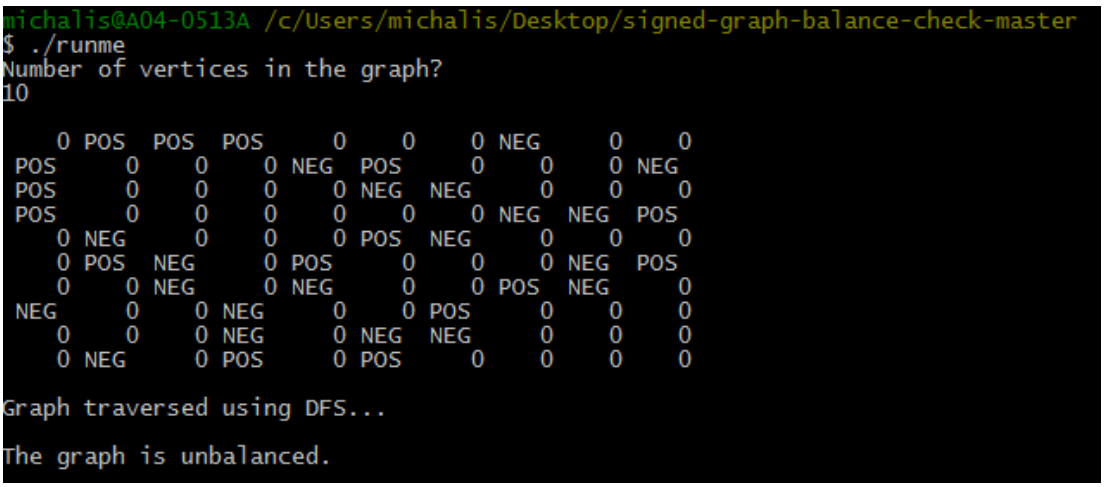
Αγνοούμε το exception που παίρνουμε όταν εκτελούμε το *make* καθώς αυτό συμβαίνει μόνο στα Windows επειδή δεν εκτελούνται οι εντολές που έχουμε μέσα στο makefile σχετικά με την διαγραφή των περιττών .o αρχείων.

4. Στη συνέχεια και αφού επιβεβαίωσουμε 4 OK στο terminal (2 για κάθε αλγόριθμο x (1 για ισορροπημένο γράφο + 1 για ανισόρροπο), μπορούμε χρησιμοποιώντας είτε απλά την εντολή *make* χωρίς παραμέτρους (αλλά και χωρίς τη δυνατότητα γραφικής αναπαράστασης) είτε εκτελώντας *make viz* να δημιουργήσουμε το απαραίτητο εκτελέσιμο αρχείο μέσω του οποίου θα τρέξουμε το πρόγραμμα μας.



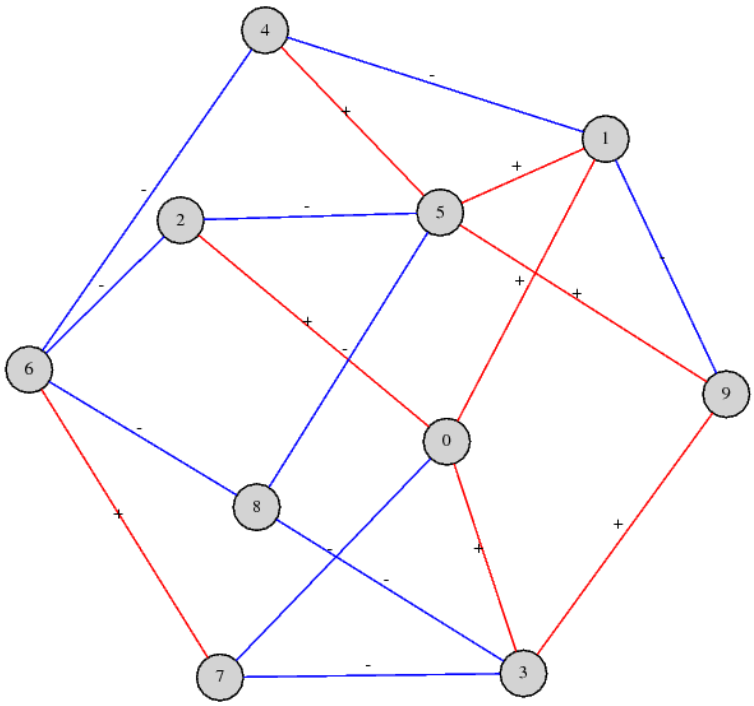
Εικόνα 16: *make viz και εκτέλεση του προγράμματος με δυνατότητα γραφικής αναπαράστασης*

Αφού έχουμε ήδη εγκατεστημένα τα Graphviz και Evince (ή όποιο άλλο πρόγραμμα της επιλογής μας και αφού έχουμε κάνει την απαραίτητη αλλαγή στον κώδικα την οποία σχολιάσαμε στο προηγούμενο κεφάλαιο (4.2)), θα μεταγλωττίσουμε το πρόγραμμα μας με όλες τις δυνατότητες που παρέχονται (επιπλέον γραφική αναπαράσταση). Καθώς τρέχουμε το παραγόμενο από το *make* εκτελέσιμο αρχείο, το πρόγραμμα, ξεκινώντας, μας ρωτάει πόσους κόμβους θέλουμε να έχει ο γράφος και περιμένει απάντηση μας. Οποιαδήποτε απάντηση έξω από το πεδίο ορισμού [3, 18000] θα μας δώσει περεταίρω πληροφορία για το τι πρέπει να δώσουμε ως είσοδο. Το ελάχιστο όριο των τριών κόμβων είναι γιατί τόσοι χρειάζονται ώστε να δημιουργηθεί ο μικρότερος δυνατός γράφος ενώ το ανώτατο όριο των 18000 κόμβων καταλαμβάνει γύρω στα 1,5 GB μνήμη οπότε σε περίπτωση εισόδου μεγαλύτερου αριθμού αρχίζει να υπάρχει πιθανή υπερχείλιση της μνήμης σε συστήματα με μνήμη 2 GB.



Εικόνα 17: *Εκτέλεση προγράμματος με τυχαίο γράφο 10 κόμβων*

5. Από τη στιγμή που θα δώσουμε έναν αριθμό ως είσοδο στο πρόγραμμα μας, έχει σχεδόν τελειώσει η διεπαφή μας σαν χρήστη απέναντι στο πρόγραμμα. Στο παρόν παράδειγμα δίνουμε 10 ως είσοδο και το πρόγραμμα με τη σειρά του δημιουργεί έναν τυχαίο γράφο 10 κόμβων, εκτυπώνει τον πίνακα γειτνίασης του, μας δίνει σαν πληροφορία με ποιόν αλγόριθμο διασχίζει τον γράφο, εκτυπώνει αν είναι ισορροπημένος ή όχι και τέλος το Evince ανοίγει αυτόματα με την γραφική αναπαράσταση του γράφου μας.

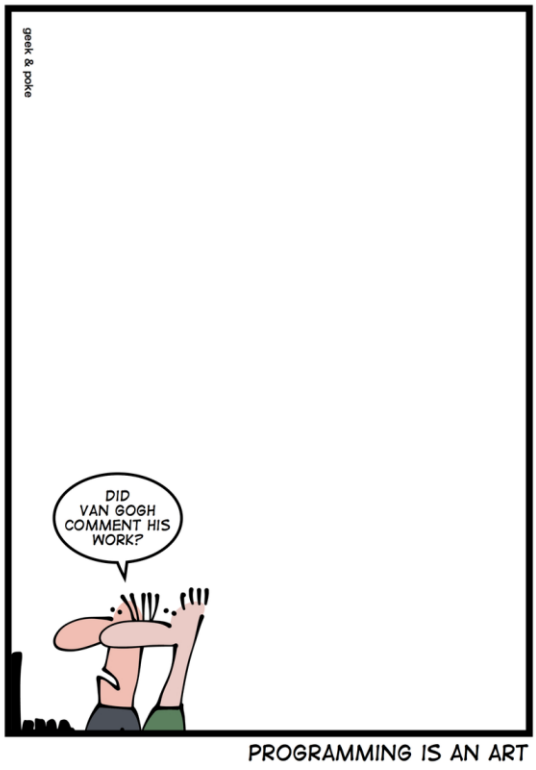


Εικόνα 18: *Γραφική αναπαράσταση γράφου εικόνας 17*

Όταν κλείσουμε το Evince ή πατήσουμε Ctrl+C στο terminal στέλνοντας σήμα τερματισμού, τότε τερματίζει και το πρόγραμμα μας. Αν είχαμε μεταγλωττίσει τον κώδικα χρησιμοποιώντας μόνο το *make*, τότε το πρόγραμμα απλά θα τερμάτιζε αφού μας έλεγε αν ο γράφος που διασχίζει ο αλγόριθμος μας είναι ισορροπημένος ή όχι. Η επιλογή αλγόριθμου διάσχισης γίνεται τυχαία.

4.4 Σχολιασμός

Ο σχολιασμός του κώδικα και η τεκμηρίωση του είναι μεγάλη υπόθεση. Η «ζωή» ενός project, η διατήρηση αλλά και η κατανόηση του εξαρτώνται άμεσα από το πόσο καλά τεκμηριωμένη είναι η βάση του κώδικα. Δεν είναι τυχαίο που πλέον υπάρχουν θέσεις εργασίας αποκλειστικά για διατήρηση και επέκταση documentation.



Εικόνα 19: *Programming is an art*

Το παρόν κεφάλαιο αποτελεί δικαιολογημένα το μεγαλύτερο της πτυχιακής εργασίας, καθώς σχολιάζεται ξεχωριστά κάθε αρχείο πηγαίου κώδικα που αποτελεί μέρος της εργασίας μας. Τα αρχεία που χρησιμοποιούμε είναι τα **types.h**, όπου έχουμε δηλώσει όλους τους τύπους και σταθερές που χρησιμοποιούνται στο πρόγραμμα μας, **algorithms.c** και **algorithms.h** όπου φυλάσσονται οι αλγόριθμοι μας, **test.c** και **test.h,** όπου έχουμε δηλώσει έτοιμους γράφους για να τεστάρουμε την ορθότητα των αλγορίθμων μας και τέλος το κύριο αρχείο μας, όπου όλα συνδέονται μεταξύ τους, το **main.c**. Όλος ο κώδικας υπάρχει στο Παράρτημα.

4.4.1 types.h

Ορίζουμε ένα header αρχείο που θα κρατάει κάποια macros, ώστε να γίνεται το πρόγραμμα μας πιο ευανάγνωστο, όπως τα ***POSITIVE*** και ***NEGATIVE*** που χαρακτηρίζουν το πρόσημο που μπορεί να πάρει μια ακμή ή ένας κόμβος, τα ***VISITED*** και ***UNVISITED*** που χαρακτηρίζουν αν ο αλγόριθμος έχει επισκεφθεί μια ακμή και το ***Allocate*** που λειτουργεί ως περικάλλυμα γύρω από την *malloc*. Οι global μεταβλητές **ip**, **rp** και **flag** χρησιμοποιούνται από τις συναρτήσεις ώθησης και ανάκλησης που θα δούμε παρακάτω, ενώ ο τύπος **stacker** χρησιμοποιείται για να δηλώσουμε μια γενική στοίβα. Σημαντικό εδώ είναι η δήλωση του τύπου **EDGE** που πρόκειται να έχουν οι μεταβλητές του πίνακα γειτνίασης μας όπου και θα αποθηκεύονται το πρόσημο της ακμής (*sign*), το πρόσημο του κόμβου που θα είναι στο πεδίο γραμμής πχ. για x[3][4] στην μεταβλητή *vertexSign* θα αποθηκεύεται το πρόσημο του κόμβου 3 (θεωρούμε πως υπάρχει και κόμβος 0) και τέλος η μεταβλητή *visited* θα μας δείχνει αν ο αλγόριθμος επισκέφθηκε μια ακμή.

4.4.2 test.c και test.h

Μέσω των αναφερθέντων αρχείων έχουμε δημιουργήσει ενσωματωμένο testing για τους αλγόριθμους μας. Υπήρχαν σκέψεις για χρήση ενός εξωτερικού testing framework όπως το Unity ή το CppTest, αλλά τελικά επιλέχθηκε το ήδη ανεπτυγμένο testing που είχαμε εξ'αρχής δημιουργήσει μέσα στο πρόγραμμα.

Δημιουργούμε μέσα στο *test.c* τις *balancedGraph* και *unbalancedGraph* έτσι ώστε, όπως υποδεικνύουν και τα ονόματα τους, να δημιουργούν έναν ισοζυγισμένο και έναν μη ισοζυγισμένο γράφο αντίστοιχα. Ο λόγος που χρησιμοποιούμε αυτές τις συναρτήσεις είναι για να ελέγξουμε αργότερα αν λειτουργούν σωστά οι αλγόριθμοι ελέγχου ισοζύγισης που θα υλοποιήσουμε. Για να εκτελεστούν τα test μας πρέπει να κάνουμε κλήση στο make με την κατάλληλη παράμετρο

```
$ make test
```

Δημιουργούμε επίσης μια συνάρτηση (*checkReturnCode*) που μας βοηθάει στην εκτύπωση του κατάλληλου αποτελέσματος μετά από κάθε τεστ.

Η συνάρτηση *TestAlgorithms* είναι αυτή που χρησιμοποιείται απευθείας από τη main του προγράμματος μας και καλεί τους BFS και DFS πάνω στους έτοιμους γράφους που δημιουργούμε από τις *balancedGraph* και *unbalancedGraph*, ώστε να δούμε αν θα έχουμε τα επιθυμητά αποτελέσματα άρα αν δουλεύουν σωστά οι αλγόριθμοι μας. Τα συγκεκριμένα τεστ δούλεψαν ενώ δημιουργήσαμε άλλον ένα ισορροπημένο γράφο (*Εικόνα 6*) τον οποίο επίσης πέρασαν οι αλγόριθμοι μας με επιτυχία.

4.4.3 algorithms.c και algorithms.h

Στα αρχεία αυτά βρίσκεται το μεγαλύτερο μέρος του κώδικα της πτυχιακής καθώς εκεί περιέχονται και οι τρεις αλγόριθμοι που αναπτύξαμε, με τις συναρτήσεις ώθησης και ανάκλησης ουράς και στοίβας.

Αλγόριθμος δημιουργίας προσημασμένων γράφων (createGraph)

Η *createGraph* δημιουργεί έναν τυχαίο προσημασμένο γράφο. Την αναπτύξαμε με τέτοιο τρόπο ώστε οι γράφοι που θα δημιουργούνται να πλησιάζουν γράφους πραγματικού κόσμου και να μπορούν να αναπαραστηθούν πρακτικά από την Graphviz.

Χρησιμοποιούμε την *srand(time(NULL))* ώστε οι αριθμοί που θα παίρνουμε ως αποτέλεσμα από την *rand()* να είναι τυχαίοι και όχι ψευδοτυχαίοι. Δημιουργούμε συνδέσεις όσο ο *max* τυχαίος αριθμός αλλά στην πραγματικότητα αρκετοί κόμβοι θα έχουν παραπάνω συνδέσεις, πχ. σε έναν γράφο 10 κόμβων τυχαίνει στον κόμβο 1 *limit* 4 συνδέσεων. Αφότου συνδεθεί τυχαία με τέσσερις τυχαίους άλλους κόμβους και προχωρήσει ο αλγόριθμος στους επόμενους 8 κόμβους, υπάρχει πιθανότητα αρκετοί να συνδεθούν με τον 1 με αποτέλεσμα στο τέλος να έχει παραπάνω από 4 συνδέσεις. Για να κάνουμε ακόμα πιο τυχαίο τον γράφο, επιτρέπουμε συγκρούσεις ανάμεσα στους κόμβους πχ. αν ο κόμβος 1 συνδεθεί με τον κόμβο 5, προχωρήσει ο αλγόριθμος και φτάσει η σειρά του 5 να συνδεθεί με διάφορους κόμβους τότε υπάρχει η πιθανότητα να τύχει ο κόμβος 1 και να υπάρξει overlapping της σύνδεσης. Μέχρι και λίγες αλλαγές πριν την ολοκλήρωση του κώδικα, συνηθίζαμε να αποτρέπουμε τέτοιες συγκρούσεις αλλά όπως αποδείχθηκε είναι βολικό να τις επιτρέπουμε ενώ παράλληλα ο γράφος μας γίνεται ακόμα πιο τυχαίος με την απουσία ενός ακόμα περιορισμού. Ο τρόπος με τον οποίο επιτρέπουμε τέτοιες συγκρούσεις είναι με το να μην κάνουμε *UNVISITED* το *mat[j][i].visited* όταν κάνουμε το αντίστοιχο *mat[i][j].visited* μέσα στη *while* μας. Φυσικά στο τέλος της δημιουργίας του γράφου μας, θα πρέπει να γίνουν όλα τα αντίστοιχα πεδία ίσα.

Συναρτήσεις ώθησης και ανάκλησης

Για την υλοποίηση του αλγόριθμου που θα διασχίζει τον γράφο με BFS τεχνική χρειαζόμαστε την δημιουργία μιας ουράς (First In First Out). Οι συναρτήσεις ώθησης και ανάκλησης από αυτή την ουρά φαίνονται στο παραπάνω κομμάτι κώδικα. Άξιο αναφοράς στην συνάρτηση queuPop, όταν ωθείται ένας κόμβος ο οποίος υπάρχει ήδη αποθηκευμένος στην ουρά (πάμε από έναν γνωστό κόμβο σε έναν άλλο γνωστό μέσω μη εξερευνημένης ακμής άρα κλείνει ένας κύκλος μέσα στον γράφο), τότε δίνεται σήμα στον αλγόριθμο μέσω της αλλαγής της *flag* να ελέγξει τον γράφο για ισορροπία.

Όπως αναφέρθηκε ήδη (2.1.2), ο αλγόριθμος που χρησιμοποιεί DFS τεχνική διάσχισης, υλοποιείται με δύο στοίβες. Παραπάνω οι συναρτήσεις ώθησης και ανάκλησης στοίβας έχουν σχεδιασεί με έναν γενικό τρόπο ώστε και οι δύο μας στοίβες να χρησιμοποιούν τις ίδιες συναρτήσεις χωρίς κανένα πρόβλημα.

Αλγόριθμος ελέγχου ισοζύγισης με τεχνική BFS

Ο κώδικας του αλγορίθμου που ελέγχει την ισορροπία του γράφου με BFS τεχνική είναι βασισμένος όπως αναφέρθηκε στο κεφάλαιο 2.1.1 στον αλγόριθμο *Algorithm: Signed_Graph_Balance* του Εμ. Λουκάκη. Ο κώδικας ξεκινάει με την τυχαία επιλογή του κόμβου έναρξης i του αλγορίθμου και στη συνέχεια μέσω μιας for ο αλγόριθμος εξερευνεί όλους τους πιθανούς γειτονικούς κόμβους j του κόμβου i που του δίνεται σαν είσοδος. Στην πρώτη ακμή (i, j) που θα βρεθεί, ο κόμβος i αποκτάει τυχαίο πρόσημο και ο κόμβος j προωθείται στην ουρά. Στη συνέχεια ελέγχεται αν η μεταβλητή *flag* έχει αλλάξει όπως αναφέρθηκε στις ***Συναρτήσεις ώθησης και ανάκλησης*** για την συνάρτηση *queuePush*. Αν ναι, τότε υπολογίζεται το γινόμενο των προσήμων του κόμβου i και της ακμής (i, j). Αν το πρόσημο που θα προκύψει είναι διαφορετικό του κόμβου j τότε ο γράφος είναι μη ισορροπημένος και ο αλγόριθμος σταματάει. Αν ο γράφος είναι ισορροπημένος (άρα ο αλγόριθμος συνεχίσει) ή αν η μεταβλητή *flag* δεν έχει αλλάξει από την *queuePush,* τότε ο αλγόριθμος συνεχίζει και υπολογίζει το πρόσημο που θα πάρει ο κόμβος j. Υπενθυμίζουμε ότι τα πρόσημα των κόμβων χρησιμοποιούνται μόνο για τον έλεγχο ισορροπίας που γίνεται αν ένας κύκλος κλείνει οπότε και πρέπει να ελεγθεί η ισορροπία του. Τέλος η ακμή (i, j) χαρακτηρίζεται ως εξερευνημένη (*VISITED*). Ο λόγος που επαναλαμβάνονται πολλές μεταβλητές αντίστροφα όπως εδώ η *mat[i][j].visited* και *mat[j][i].visited* είναι ότι δουλεύουμε σε δισδιάστατο πίνακα και ουσιαστικά αυτές οι δύο μεταβλητές είναι η ίδια ακμή στον γράφο. Αν ο γειτονικός κόμβος j τον οποίο εξερευνήσαμε είναι ο τελευταίος του κόμβου i που βρίσκεται ο αλγόριθμος, τότε ανακαλείται από την ουρά ο επόμενος κόμβος που θα συνεχίσει ο αλγόριθμος. Αν η ουρά είναι άδεια (άρα θα επιστρέψει -1) τότε ο αλγόριθμος σταματάει και ο γράφος είναι ισορροπημένος.

Αλγόριθμος ελέγχου ισοζύγισης με τεχνική DFS

Πριν ξεκινήσει η ουσιαστική λειτουργία του DFS πρέπει να δημιουργήσουμε τις στοίβες μας χρησιμοποιώντας τον ειδικό τύπο που ορίσαμε στο *types.h,* να τις προετοιμάσουμε αλλά και δεσμεύσουμε χώρο για τον πίνακα visited που χρησιμοποιείται από τον αλγόριθμο. Ο κώδικας του ουσιαστικού μέρους του αλγορίθμου βασίζεται κατά γράμμα στον αλγόριθμο που παρουσιάσαμε στην παράγραφο 2.1.2. Άξιο αναφοράς είναι ότι, όπως θα αναφέρουμε και στο κεφάλαιο για την main με τον πίνακα γειτνίασης, έτσι και εδώ όσους πίνακες δημιουργούμε (στοίβες, ουρά, visited) τους δεσμεύουμε και αποδεσμεύουμε δυναμικά. Επίσης, ενώ οι πρώτες εκδόσεις του κώδικα μας περιείχαν πολλές global μεταβλητές, πλέον χρησιμοποιούμε μόλις τρεις.

4.4.4 main.c

Στη *main* του προγράμματος, συνοψίζουμε όλη την δουλειά του προγράμματος, περνώντας διάφορα μέρη σε βοηθητικές συναρτήσεις. Στο Παράρτημα, στο τέλος της αναφοράς, έχουμε επιλέξει τα πιο σημαντικά μέρη με έντονο χρώμα. Αν τρέξουμε το *make* με την παράμετρο *test* (δηλαδή *make* test) τότε εκτελείται το κομμάτι όπου έχουμε ορίσει το *TEST* macro δηλαδή τα test πάνω στους αλγόριθμους μας μέσω των δύο κλήσεων στην *TestAlgorithms*. Το σημαντικότερο μέρος της δουλειάς εκτελείται στην *userIO* όπου καλείται ο εκάστοτε αλγόριθμος πάνω σε έναν τυχαίο γράφο και εκτυπώνεται το αν ο γράφος είναι ισορροπημένος ή όχι. Τέλος η *graphInit* χρησιμοποιείται, όπως υποδεικνύει το όνομα της, για αρχικοποίηση των γράφων μας (στην ουσία αρχικοποίηση του πίνακα γειτνίασης μας).

4.4.5 Γραφική αναπαράσταση

Για την γραφική αναπαράσταση ενός γράφου πρώτον κάνουμε ρητό το ότι δεν θα εκτελεστεί κατά την διάρκεια των τεστ (*#ifndef TEST)* ενώ δεν είναι υποχρεωτικό να εκτελεστεί η ίδια η γραφική αναπαράσταση (*#ifdef VIZ)* και δεύτερον χρησιμοποιούμε εντολές επεξεργασίας ροής δεδομένων (*fprintf, fopen, fclose*) καθώς απαραίτητη κρίνεται η δημιουργία ενός *.dot* αρχείου με κατάλληλη σύνταξη. Τα συγκεκριμένα αρχεία περιέχουν κώδικα σε *DOT* γλώσσα, η οποία χρησιμοποιείται για την περιγραφή γράφων. Μέσω της *fopen* δημιουργούμε ένα νέο αρχείο ή το ανοίγουμε αν υπάρχει ήδη στο directory που βρισκόμαστε. Αφού ολοκληρωθεί η εκτέλεση των εντολών επεξεργασίας ροής δεδομένων με την κλήση της *fclose* έχουμε ένα ολοκληρωμένο .dot αρχείο έτοιμο για είσοδο ως δεδομένα στο *neato* που θα μας αναπαραστήσει γραφικά τον γράφο. Το αποτέλεσμα από το *neato* είναι ένα νέο αρχείο, το οποίο απλά προβάλουμε μέσω του προγράμματος *Evince*.

Πρέπει να επιστήσουμε την προσοχή των χρηστών Windows λειτουργικών αναφορικά με το παραπάνω κομμάτι κώδικα και πιο συγκεκριμένα τις κλήσεις *system,* καθώς θα πρέπει να έχουν προσθέσει στην περιβαλλοντική μεταβλητή *PATH* τα paths που οδηγούν στα εκτελέσιμα αρχεία *neato* και *evince*. Στην δικιά μας περίπτωση τα παρακάτω paths μας βοήθησαν ώστε να τρέξουμε τις κλήσεις *system* μας χωρίς κανένα πρόβλημα.

C:\Program Files (x86)\Graphviz2.36\bin
C:\Users\michalis\AppData\Local\Apps\Evince-2.32.0.145\bin

Υπάρχουν δύο τρόποι να επεξεργαστεί κάποιος τις περιβαλλοντικές μεταβλητές του λειτουργικού του στα Windows (τουλάχιστον στα Windows 7 που εμείς χρησιμοποιήσαμε). Συνήθως αν ένας χρήστης έχει δημιουργήσει έναν απλό λογαριασμό, αν ακολουθήσει το path μέσα στις παρενθέσεις (*Start>Control Panel>User Accounts and Family Safety>User Accounts>Change my environmental variables*), δεν θα μπορέσει να επεξεργαστεί τις μεταβλητές του συστήματος παρά μόνο τις μεταβλητές του χρήστη.
Για να επεξεργαστεί κάποιος τις μεταβλητές του συστήματος, η σίγουρη περίπτωση είναι *Δεξί κλικ στο εικονίδιο Ο Υπολογιστής Μου>Properties>Advanced system settings>Environmental variables.*

4.4.6 Παρατηρήσεις

Στο παρόν κεφάλαιο θα αναφερθούν τυχόν σημαντικές αλλαγές που έγιναν στον κώδικα μέχρι να πάρει την τελική του μορφή αλλά και παρατηρήσεις σχετικά με τον παρόντα [κώδικα].

Η αρχική μορφή του project ήταν ένα απλό .c αρχείο το οποίο περιείχε τους τρεις αλγόριθμους μας (createGraph, BFS και DFS) και μια main να τα «κολλάει» όλα. Στην πορεία όμως προστέθηκε και η δυνατότητα γραφικής αναπαράστασης, ενώ όσο διάρκεσε η βελτιστοποίηση του κώδικα, βρέθηκαν αρκετά λεγόμενα “bugs” και έγιναν πολλές ανακατατάξεις στον κώδικα. Η σημαντικότερη ανακατάταξη εξ αυτών ήταν το σπάσιμο του κώδικα από ένα αρχείο σε εφτά, τρία .c και τέσσερα .h για καλύτερη αναγνωσιμότητα αλλά και ευκολότερη αποσφαλμάτωση.

Το σημείο του κώδικα που υπέστη τις περισσότερες αλλαγές μέχρι να έρθει στην τελική του μορφή ήταν ο αλγόριθμος δημιουργίας τυχαίων προσημασμένων γράφων και αυτό γιατί είναι το μοναδικό μέρος του κώδικα στο οποίο είχαμε σχεδόν πλήρη ελευθερία ως προς τον τρόπο σχεδίασής του με αποτέλεσμα να πειραματιστούμε αρκετά μέχρι να καταλήξουμε κάπου.

5. ΑΠΟΤΕΛΕΣΜΑΤΑ

Αν δεν ξέρετε τίποτα για τους υπολογιστές, απλά να θυμάστε ότι είναι μηχανήματα που κάνουν ακριβώς ότι τους πείτε αλλά συχνά σας εκπλήσσουν στο τέλος - Richard Dawkins

Στο τελευταίο κεφάλαιο ασχολούμαστε με τα αποτελέσματα της εργασίας μας, τι και ποιόν μπορεί να ωφελήσει η πτυχιακή αυτή και τι χρησιμότητα έχει. Καταρχάς και βασικότερο είναι ότι οι αλγόριθμοι αναζήτησης που αναπτύξαμε, δουλεύουν όπως πρέπει. Υπάρχουν σίγουρα περιθώρια βελτίωσης σε καθαρά προχωρημένα τεχνικά ζητήματα καθώς κώδικας σε C πάντα σηκώνει πολλές βελτιστοποιήσεις. Επίσης τα εργαλεία για **testing** και **profiling** γύρω από τη C…

1) απαιτούν έρευνα ώστε να κάποιος να επιλέξει τα σωστά και
2) χρειάζονται εκμάθηση μέχρι να μπορεί κάποιος να τα χρησιμοποιεί με ευχέρεια, οπότε ίσως η επιλογή μιας άλλης γλώσσας για την ανάπτυξη του κώδικα μας να ήταν καλύτερη λύση όπως πχ. η Go, στην οποία όλα αυτά τα εργαλεία έρχονται “out of the box” μαζί με την γλώσσα.

Σε κάθε περίπτωση όμως, το συνολικό αποτέλεσμα κρίνεται άκρως επιτυχημένο. Όχι μόνο γιατί υλοποιήσαμε όλους τους προαπαιτούμενους στόχους, αλλά επειδή αναπτύξαμε επιπρόσθετη δυνατότητα της γραφικής αναπαράστασης των γράφων μέσω της Graphviz, ενώ επιπλέον αναλύσαμε ένα εύρος προγραμμάτων (git, make, travis ci, valgrind) που μπορούν να αποτελέσουν χρήσιμα εργαλεία στην εργαλειοθήκη ενός προγραμματιστή.

Η εργασία αυτή μπορεί να ωφελήσει οποιονδήποτε έχει ένα ενδιαφέρον στους γράφους αλλά και επόμενους μελετητές, καθώς, όπως αναφέρουμε στο 1.3, το φάσμα χρήσης των προσημασμένων [γράφων] είναι αρκετά ευρύ. Το προσωπικό ώφελος που αποκομίζω εγώ είναι βαθιά γνώση πάνω στους γράφους και τους αλγόριθμους αναζήτησης. Επίσης μέσω της εργασίας μου πάνω στο συγκεκριμένο project απέκτησα καλύτερη κατανόηση για τα εργαλεία που χρησιμοποίησα ενώ και η ίδια η εργασία πλέον αποτελεί μια ένδειξη των δεξιοτήτων μου για πιθανούς μελλοντικούς εργοδότες μου.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Δεν υπάρχουν απαντήσεις, μόνο παραπομπές
– Ο Νόμος των Βιβλιοθηκών του Wiener

[1] Δ. Σπινέλλης, *Code Reading: The Open Source Perspective*, Addison-Wesley, 2005

[2] Ι. Παπουτσής, *Εισαγωγή στις Δομές Δεδομένων και στους Αλγόριθμους,* 2010, p.37

[3] T. Zaslavsky, A Mathematical Bibliography of Signed and Gain Graphs and Allied Areas Sixth Edition, Revision b, 1999 September 4

[4] Wikipedia

[5] *Ηλίας Κ. Σάββας, Αλγόριθμοι και Πολυπλοκότητα*

[6] E. Loukakis, *A Dynamic Programming Algorithm to Test a Signed Graph for Balance*, Intern. J. Computer Math., 2003, Vol. 80(4), pp. 499-507

[7] Anatomy of Facebook, Facebook

[8] The Average Facebook User, Huffington Post

[9] S. Lipschutz, M. Lipson, *Schaum's Outline of Theory and Problems of Discrete Mathematics*, 2nd edition, 1997

[10] F. Roberts, *Discrete Mathematical Models with applications to social, biological and environmental problems*, Pearson, 1st edition, 1976, p. 70-71

[11] W.T.Tuttle, *Graph Theory As I Have Known It*, Oxford Science Publications, 1998,

[12] O. Frank, F. Harary, *Balance in stochastic signed graphs,* Social Networks 2, p. 155-163

[13] F. Huffner, N. Beitzler, R. Niedermeier, *Optimal Edge Deletions for Signed Graph Balance*, Proc. 6Th WEA, Vol. 4525 in LNCS, 2007, p. 297-310

[14] https://eiu.bvdep.com/version-2012413/EIU/Help/indexnumbers.htm

[15] GCC, the GNU Compiler Collection

[16] Graphviz - Graph Visualization Software

[17] The DOT Language, Graphviz

[18] Official Sublime Text website

[19] ThrowTheSwitch.org

ΠΑΡΑΡΤΗΜΑ

Ο κώδικας που θα παρουσιαστεί στο παρόν παράρτημα δεν είναι απόλυτα στοιχισμένος όπως βρίσκεται μέσα στα αρχεία ώστε να ευνοηθεί η αναγνωσιμότητα του.

types.h

```
#ifndef types_h
#define types_h

#include <stdlib.h>
#include <stdint.h>

//Data structures and macros

#define POSITIVE 2
#define NEGATIVE 1
#define VISITED 2
#define UNVISITED 1
#define Allocate(type, count)        (type *)malloc((count) * sizeof(type))

int ip = 0, rp = 0, flag;

//stacker is a type used to represent a stack
typedef struct {
        int head;
        int *actualStack;
}stacker;

//EDGE is a type used to represent an edge in a graph
typedef struct {
        uint8_t sign;
        uint8_t vertexSign;
        uint8_t visited;
}EDGE;

#endif
```

test.h

```
#ifndef test_h
#define test_h

#include "types.h"

int balancedGraph(EDGE** mat);
int unbalancedGraph(EDGE** mat);
int balancedGraphFig7(EDGE** mat);
void checkReturnCode(int balance, int returnCode);
void TestAlgorithms(EDGE** mat, int vertices, int balance);

#endif
```

test.c

```
#include "types.h"
#include "test.h"
#include "algorithms.h"
#include <stdio.h>


// ======================================
// Testing graphs
// **************************************
// Balanced graph functions return 0
// Unbalanced graph functions return 1
// ======================================
        int balancedGraph(EDGE** mat){
        int i, j;

        mat[0][1].sign = NEGATIVE;      //0 -- 1 -
        mat[0][2].sign = POSITIVE;      //0 -- 2 +
        mat[0][3].sign = POSITIVE;      //0 -- 3 +
        mat[1][0].sign = NEGATIVE;
        mat[2][0].sign = POSITIVE;
        mat[3][0].sign = POSITIVE;
        mat[1][2].sign = NEGATIVE;      //1 -- 2 -
        mat[1][4].sign = POSITIVE;      //1 -- 4 +
        mat[2][1].sign = NEGATIVE;
        mat[4][1].sign = POSITIVE;
```

```
        mat[2][3].sign = POSITIVE;        //2 -- 3 +
        mat[3][2].sign = POSITIVE;
        mat[3][4].sign = NEGATIVE;        //3 -- 4 -
        mat[4][3].sign = NEGATIVE;


        for(i = 0; i < 5; i++){
        for(j = 0; j < 5; j++)
        if(mat[i][j].sign){
             mat[i][j].visited = UNVISITED;
             mat[j][i].visited = UNVISITED;
           }
        }

return 0;
}


int unbalancedGraph(EDGE** mat){
        int i, j;

        or(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
        mat[i][j].sign = POSITIVE;


        for(i = 0; i < 5; i++)
        for(j = 0; j < 5; j++)
        if(i == j)
        mat[i][j].sign = 0;

        mat[4][3].sign = NEGATIVE;
        mat[3][4].sign = NEGATIVE;

        for(i = 0; i < 5; i++){
        for(j = 0; j < 5; j++)
             f(mat[i][j].sign){
             mat[i][j].visited = UNVISITED;
             mat[j][i].visited = UNVISITED;
        }
        }

        return 1;
}
```

```
//Balanced graph shown on Figure 7 of STRUCTURAL BALANCE: A
GENERALIZATION
//OF HEIDER'S THEORY by Cartwright and Harary (1956)
        int balancedGraphFig7(EDGE** mat){
        int i, j;

        mat[0][1].sign = POSITIVE;      //A -- B +
        mat[1][0].sign = POSITIVE;
        mat[1][2].sign = POSITIVE;      //B -- C +
        mat[2][1].sign = POSITIVE;
        mat[2][3].sign = POSITIVE;      //C -- D +
        mat[3][2].sign = POSITIVE;
        mat[3][0].sign = POSITIVE;      //D -- A +
        mat[0][3].sign = POSITIVE;
        mat[3][4].sign = NEGATIVE;      //D -- E -
        mat[4][3].sign = NEGATIVE;
        mat[4][5].sign = POSITIVE;      //E -- F +
        mat[5][4].sign = POSITIVE;
        mat[5][3].sign = NEGATIVE;      //F -- D -
        mat[3][5].sign = NEGATIVE;
        mat[1][5].sign = NEGATIVE;      //F -- B -
        mat[5][1].sign = NEGATIVE;
        mat[5][6].sign = POSITIVE;      //F -- G +
        mat[6][5].sign = POSITIVE;
        mat[6][0].sign = NEGATIVE;      //G -- A -
        mat[0][6].sign = NEGATIVE;
        mat[6][7].sign = POSITIVE;      //G -- H +
        mat[7][6].sign = POSITIVE;
        mat[7][0].sign = NEGATIVE;      //H -- A -
        mat[0][7].sign = NEGATIVE;
        mat[7][3].sign = NEGATIVE;      //H -- D -
        mat[3][7].sign = NEGATIVE;


for(i = 0; i < 8; i++){
        for(j = 0; j < 8; j++)
        if(mat[i][j].sign){
              mat[i][j].visited = UNVISITED;
              mat[j][i].visited = UNVISITED;
        }
}

return 0;
}
```

```
// ========================================
// Other functions
// ========================================
void TestAlgorithms(EDGE** mat, int vertices, int balance){
int k, i, j;
EDGE** pin;

// Clone original matrix into a second one so both traversing algorithms can run
// independently
pin = Allocate(EDGE*, vertices);
for(i = 0; i < vertices; i++){
pin[i] = Allocate(EDGE, vertices);
}

for(i = 0; i < vertices; i++)
for(j = 0; j < vertices; j++){
pin[i][j].sign = mat[i][j].sign;
pin[i][j].vertexSign = mat[i][j].vertexSign;
pin[i][j].visited = mat[i][j].visited;
}

k = rand() % vertices;
printf("Testing DFS:\n");
checkReturnCode(balance, DFS(mat, vertices));

printf("Testing BFS:\n");
checkReturnCode(balance, BFS(pin, vertices));

for(i = 0; i < vertices; i++){
free(pin[i]);
}
free(pin);
}

void checkReturnCode(int balance, int returnCode){
if (balance == returnCode){
printf("OK\n\n");
} else {
printf("TEST FAILED!\n\n");
}
}
```

algorithms.h

```
#include "types.h"

void createGraph(EDGE** mat, int vertices);
int BFS(EDGE** mat, int vertices);
int DFS(EDGE** mat, int vertices);
void queuePush(int* queue, int elem);
int queuePop(int* queue);
void stackPush(stacker *stack, int elem);
int stackPop(stacker *stack);
```

algorithms.c

```
#include "algorithms.h"
#include "types.h"
#include <stdlib.h>
#include <stdio.h>

// ======================================
// Algorithmic functions
// **************************************
// * createGraph: creates a random signed graph
// * BFS and DFS: traverse a given signed graph checking for its balance
// * Stack and queue operations
// All functions have been designed in a generic-like way
// ======================================

int ip = 0, rp = 0, flag;

void createGraph(EDGE** mat, int vertices){
int limit, i, j;

srand(time(NULL));

for(i = 0; i < vertices; i++){

// limit is a random number of initial connections assigned to current node
// Minimum number of edges that will get randomly assigned to a vertex
// is 2 and maximum is 6
if(vertices == 3)
limit = 2;
else if(vertices == 4)
limit = rand() % 2 + 2;
```

```
else
limit = rand() % (vertices % 5 + 1) + 2;

while(limit){
j = rand() % vertices;

// An edge has to have different endpoints so essentially i and j have to be unequal
// If not or already an edge with the current (i,j) endpoints exists then no new edge
// should be added
if((j == i) || (mat[i][j].visited == UNVISITED)){
continue;
}

//The edge sign will be either 1 (NEGATIVE) or 2 (POSITIVE)
mat[i][j].sign = 2;//rand() % 2 + 1;
mat[j][i].sign = mat[i][j].sign;

//At start all the edges are unvisited
mat[i][j].visited = UNVISITED;

limit--;
}
}

for(i = 0; i < vertices; i++){
if(mat[i][j].visited == UNVISITED)
mat[j][i].visited = UNVISITED;
}

return;
}


int BFS(EDGE** mat, int vertices){
int i, j, product, count = 1, jcount = 0, k, vertexSign, *queue, returnCode = 0;

printf("Graph traversed using BFS...\n\n");
i = rand() % vertices;

queue = Allocate(int, vertices);

//Algorithm starts traversing here
//Crossing every possible adjacent (j) vertex of the current vertex (i)
for(j = 0; j < vertices; j++){
        if(mat[i][j].visited == UNVISITED){
            jcount ++;
```

```
//Random sign on the first vertex
if((count == 1) && (jcount == 1)){
        mat[i][j].vertexSign = rand() % 2 + 1;
        vertexSign = mat[i][j].vertexSign;

        for(k = 0; k < vertices; k++)
        if(mat[i][k].sign)
        mat[i][k].vertexSign = vertexSign;
}

queuePush(queue, j);

//If flag is 1 then (i,j) is a cycle-closing edge. If the product between the
// sign of vertex i and the sign of edge (i,j) is different from the sign of
// vertex j then the subgraph that (i,j) is closing, is not balanced
// and the algorithm breaks
if(flag == 1){
        product = mat[i][j].vertexSign * mat[i][j].sign;
// NEGATIVE * NEGATIVE (1 * 1) == positive (1) and
// POSITIVE * POSITIVE (2 * 2) == positive (4)
// the only case the product is going to be negative is when
// NEGATIVE * POSITIVE (1 * 2) == negative (2)
if(product == 2)
        product = NEGATIVE;
else
        product = POSITIVE;

if(product != mat[j][i].vertexSign){
        returnCode = 1;
        break;
        }
}

// If the product of the signs of vertex i and edge (i,j) is 2, a multiplication
// between a negative (1) and a positive sign (2) has to be preceded (1*2 = 2).
// So the sign of vertex j will be negative
product = mat[i][j].vertexSign * mat[i][j].sign;
if(product == 2)
        for(k = 0; k < vertices; k++){
        if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
        mat[j][k].vertexSign = NEGATIVE;
        }
else
        for(k = 0; k < vertices; k++){
        if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
```

```
        mat[j][k].vertexSign = POSITIVE;
        }

        mat[i][j].visited = VISITED;
        mat[j][i].visited = VISITED;
}

if(j == vertices - 1){
        count ++;
i = queuePop(queue);

        if(i == -1)
                break;

        j = -1;
        jcount = 0;
        }
}

free(queue);
return returnCode;
}


int DFS(EDGE** mat, int vertices){
        int *visited, i, j, k, product, returnCode = 0;
        stacker *stackA, *stackB;

        printf("Graph traversed using DFS...\n\n");
        i = rand() % vertices;

        //Initialize stacks
        stackA = Allocate(stacker, 1);
        stackA->actualStack = Allocate(int, vertices);
        stackA->head = 0;
        stackB = Allocate(stacker, 1);
        stackB->actualStack = Allocate(int, vertices);
        stackB->head = 0;

        visited = Allocate(int, vertices);
        visited[i] = VISITED;
        for(j = 0; j < vertices; j++){
                if(i == j)
                        continue;

                visited[j] = UNVISITED;
```

```
        }

        //Random sign on the first vertex
        k = rand() % 2 + 1;
        for(j = 0; j < vertices; j++)
                if(mat[i][j].sign)
                        mat[i][j].vertexSign = k;

        //Algorithm starts traversing here
        while(1){

        for(j = 0; j < vertices; j++){
                if(mat[i][j].visited == UNVISITED){

// Checking every possible adjacent vertex (j) that is reached
// through a POSITIVE edge
                if(mat[i][j].sign == POSITIVE){
                        if(visited[j] == UNVISITED){
                        visited[j] = VISITED;

                        product = mat[i][j].vertexSign * mat[i][j].sign;
                        if(product == 2)
                        for(k = 0; k < vertices; k++){
                                if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
                                        mat[j][k].vertexSign = NEGATIVE;
                                                }
                        else
                                for(k = 0; k < vertices; k++){
                                if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
                                mat[j][k].vertexSign = POSITIVE;
                                                }

                                stackPush(stackA, j);
                                }
// Checking if an adjacent vertex (j) has a different sign from the current (i)
// because we are on a POSITIVE edge
   else if((visited[j] == VISITED) && (mat[i][j].vertexSign != mat[j][i].vertexSign)){
        returnCode = 1;      //toggle returnCode - graph is unbalanced
        break;
        }
}
        else if(mat[i][j].sign == NEGATIVE){
                if(visited[j] == UNVISITED){
                        visited[j] = VISITED;

                        product = mat[i][j].vertexSign * mat[i][j].sign;
```

```
        if(product == 2)
        for(k = 0; k < vertices; k++){
                if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
                        mat[j][k].vertexSign = NEGATIVE;
        }
        else
        for(k = 0; k < vertices; k++){
                if(mat[j][k].sign && (mat[j][k].vertexSign == 0))
                        mat[j][k].vertexSign = POSITIVE;
        }

        stackPush(stackB, j);
        }
// Checking if an adjacent vertex (j) has the same sign as the current (i)
// because we are on a NEGATIVE edge
        else if((visited[j] == VISITED) && (mat[i][j].vertexSign == mat[j][i].vertexSign)){
                returnCode = 1;       //toggle returnCode - graph is unbalanced
                break;
                }
        }
     }
}

        if(returnCode == 1){
                break;
        }

        j = stackPop(stackA);
        if(j == -1){
        j = stackPop(stackB);
        }

        if(j == -1){
        break;//graph is balanced
        }

        mat[i][j].visited = VISITED;
        mat[j][i].visited = VISITED;

        i = j;
}

        stackA->head = 0;
        stackB->head = 0;

        free(stackA->actualStack);
```

```
	free(stackA);
	free(stackB->actualStack);
	free(stackB);
	free(visited);

	return returnCode;
}


// ========================================
// Queue operations (Used by BFS)
// ========================================
void queuePush(int* queue, int elem){
	int i;
	flag = 0;

	//If the vertex that is to be inserted in the queue is already there then toggle flag
	for(i = 0; i < ip; i++)
		if(queue[i] == elem)
			flag = 1;

	if(flag == 0)
		queue[ip++] = elem;
}


int queuePop(int* queue){
	if (ip == rp)
		return -1;
	else
		return queue[rp++];
}


// ========================================
// Stack operations (Used by DFS)
// ========================================
void stackPush(stacker *stack, int elem){
	stack->actualStack[stack->head++] = elem;
}


int stackPop(stacker *stack){
	stack->head--;

	if (stack->head < 0)
```

```
                return -1;
        else
                return stack->actualStack[stack->head];
}
        for(i = 0; i < vertices; i++){
                if(mat[i][j].visited == UNVISITED)
                        mat[j][i].visited = UNVISITED;
        }
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "types.h"
#include "algorithms.h"

#ifdef TEST
#include "test.h"
#endif

// ========================================
// Helper functions
// ========================================
void userIO(EDGE** mat, int vertices){
        int i, j, returnCode = 0;

        printf("\n");
        for(i = 0; i < vertices; i++){
                for(j = 0; j < vertices; j++){
                        if(mat[i][j].sign == 0)
                                printf("%5d", 0);
                        else if(mat[i][j].sign == NEGATIVE)
                printf(" NEG ");
          else if(mat[i][j].sign == POSITIVE)
                        printf(" POS ");
                }
       printf("\n");
   }
   printf("\n");

if(rand() % 2)
        returnCode = DFS(mat, vertices);
```

```
else
        returnCode = BFS(mat, vertices);

    if(returnCode == 1)
                printf("The graph is unbalanced.\n\n");
        else if(returnCode == 0)
                printf("The graph is balanced.\n\n");
}


//graphInit initializes the adjacency matrix
void graphInit(EDGE** mat, int vertices){
        int i, j;

        for(i = 0; i < vertices; i++)
                for(j = 0; j < vertices; j++){
                        mat[i][j].sign = 0;
                        mat[i][j].vertexSign = 0;
                        mat[i][j].visited = 0;
                }
}



int main (int argc, char **argv){

        EDGE** mat;
        int i, j, vertices;
        FILE *pFile;
        int fcount=0, iFile;

        #ifdef TEST
        printf("**********TEST MODE**********\n");
        printf("*****************************\n\n");
        vertices = 5;
        mat = Allocate(EDGE*, vertices);
        for(i = 0; i < vertices; i++){
                mat[i] = Allocate(EDGE, vertices);

        }

        graphInit(mat, vertices);

        printf("Running balanced graph test.\n");
        printf("*****************************\n\n");
        TestAlgorithms(mat, vertices, balancedGraph(mat));
```

```
        //or use balancedGraphFig7 but remember to switch vertices to 8!

        graphInit(mat, vertices);

        printf("Running unbalanced graph test.\n");
        printf("*****************************\n\n");
        TestAlgorithms(mat, vertices, unbalancedGraph(mat));



        #else
        printf("Number of vertices in the graph?\n");
        scanf("%d", &vertices);
        getchar();

// Sanity check
// A graph has to be consisted of at least three vertices in order to be a closed graph
// Although it can have up to as many verices as possible we contrain it
// to 18000 due to possible memory overflooding
// 18000 vertices equal to around 1,5 GB of RAM
        while((vertices < 3) || (vertices > 18000)){
                printf("hint: Give a number between 3 and 18000\n");
                scanf("%d", &vertices);
                getchar();
        }

        mat = Allocate(EDGE*, vertices);
        for(i = 0; i < vertices; i++){
                mat[i] = Allocate(EDGE, vertices);
        }

        graphInit(mat, vertices);

        createGraph(mat);
        userIO(mat, vertices);
        #endif


        //Optionally run Graphviz but in any case don't run it while testing
        #ifndef TEST
        #ifdef VIZ
        pFile = fopen("file.dot", "wb");

        if(pFile == NULL){
        perror("Error opening file.dot\n");
        fprintf(stderr, "Exiting...\n");
```

```
        exit(1);
        }

/*The DOT Language
==================
The following is a grammar part of the DOT language. Literal characters are given in
single quotes.Parentheses () indicate grouping when needed. Square brackets []
enclose optional items. The keywords node, edge, and graph are case-independent.
Note also that the allowed compass point values are not keywords, so these strings can
be used elsewhere as ordinary identifiers and, conversely, the parser will actually accept
any identifier.The language supports C-style comments and in addition, a line beginning
with a '#' character is considered a line output from a C preprocessor. More information
in the following link: http://graphviz.org/content/dot-language
 */
fprintf(pFile, "graph G {\n\tnode [shape=circle, style=filled];\n\n\nedge [len=3];\n");


for(i = 0; i < vertices; i++){
        for(j = 0; j < vertices; j++){
                if(mat[i][j].sign == NEGATIVE){
                        fcount++;
                        mat[j][i].sign = 0;
                        fprintf(pFile, "%d -- %d [label=\"%c\", color=\"%s\"]; ", i, j, '-', "blue");
                        if(fcount%3==0)
                                fprintf(pFile, "\n");
                }
                else if(mat[i][j].sign == POSITIVE){
                        fcount++;
                        mat[j][i].sign = 0;
                        fprintf(pFile, "%d -- %d [label=\"%c\", color=\"%s\"]; ", i, j, '+', "red");
                        if(fcount%3==0)
                                fprintf(pFile, "\n");
                }
        }
}

fprintf(pFile, "\n}");
fclose(pFile);

/*NEATO
========
Neato is a program that makes layouts of undirected graphs following the filter model of
DOT language.Its layout heuristic creates virtual physical models and runs an iterative
solver to find low energy configurations.More information in the following link:
http://graphviz.org/pdf/neatoguide.pdf
*/
```

```
system("neato -Tps file.dot -o file.ps");
system("evince file.ps");
#endif
#endif

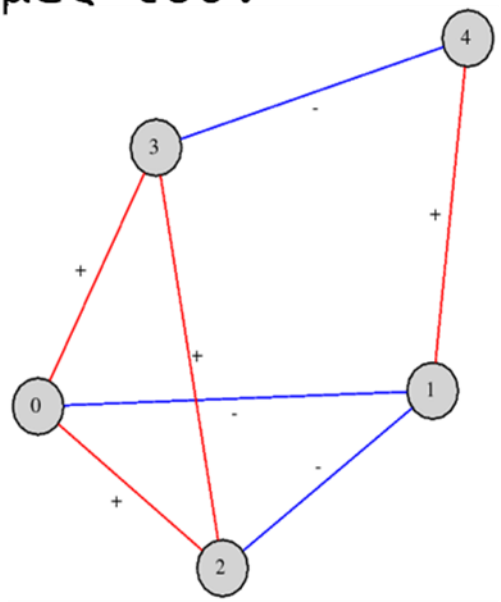
for(i = 0; i < vertices; i++){
        free(mat[i]);
}
free(mat);
return 0;
}
```

Ανάπτυξη αλγορίθμων ελέγχου ισορροπίας προσημασμένων γράφων

Μιχαήλ Καργάκης
AM: **2687**, github.com/kargakis
Επιτηρητής: **Ιωάννης Ξεζωνάκης**
Τμήμα Μηχανικών Πληροφορικής, ΤΕΙ Κρήτης

Προσημασμένος γράφος;

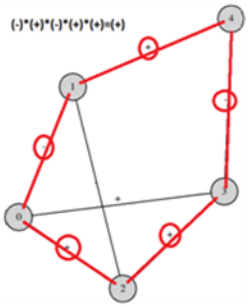
Ένας γράφος με πρόσημα (+, -) πάνω στις ακμές του.

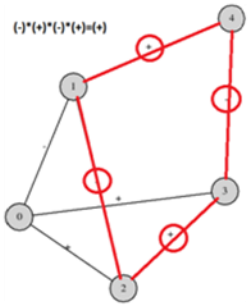


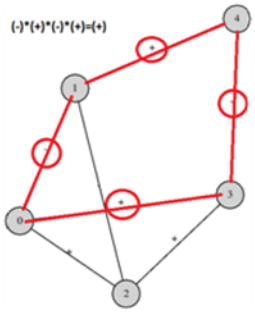
Ισορροπία σε προσημασμένους γράφους

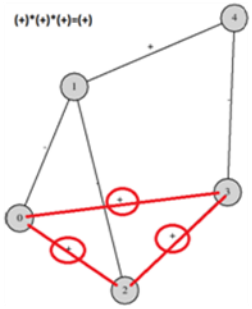
- Ένας γράφος θεωρείται ισορροπημένος αν το γινόμενο των προσήμων των ακμών του είναι θετικό.
- Ένας γράφος θεωρείται ισορροπημένος αν όλοι οι υπογράφοι που τον απαρτίζουν, είναι ισορροπημένοι.

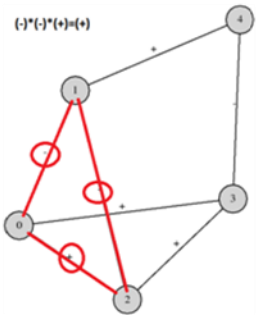
Ισορροπία σε προσημασμένους γράφους











Χρήση προσημασμένων γράφων

Η έννοια του προσημασμένου γράφου συστήθηκε από τον Αμερικανό μαθηματικό Frank Harrary το 1956 ώστε να λυθεί ένα πρόβλημα στην **κοινωνική ψυχολογία.**

Η έρευνα του Harrary βασίστηκε στη θεωρία του Heider σχετικά με την ισορροπία.

Βήματα

- Ανάπτυξη αλγόριθμου δημιουργίας τυχαίων γράφων
- Ανάπτυξη αλγορίθμων διάσχισης γράφων
- Έλεγχος ορθότητας λειτουργίας αλγορίθμων
- Οπτικοποίηση γράφων
- Βελτιστοποίηση πηγαίου κώδικα

Εργαλεία

- GCC
- make
- Sublime Text 2
- git/Github
- Travis CI
- valgrind
- GraphViz

Ανάπτυξη κώδικα

Αναπαράσταση γράφου μέσω ενός **δισδιάστατου πίνακα** με συγκεκριμένη δομή ώστε να περιγράφεται ικανοποιητικά ένας προσημασμένος γράφος.

```
typedef struct{
   uint8_t sign;
   uint8_t vertexSign;
   uint8_t visited;
}EDGE;
```

Ανάπτυξη κώδικα

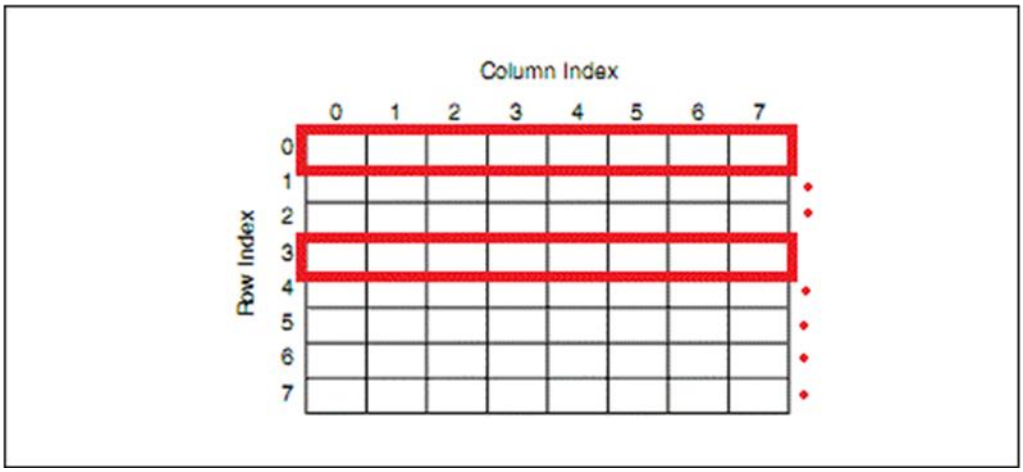
Δήλωση γράφου 7 κόμβων:

```
#define Allocate(type, count)    (type *)malloc((count) * sizeof(type))

EDGE** mat;
mat = Allocate(EDGE*,  7);
for(i = 0; i < vertices; i++){
   mat[i] = Allocate(EDGE,  7);
}
```

Ανάπτυξη κώδικα

Κόμβος θεωρείται κάθε σειρά στον δισδιάστατο πίνακα με το κάθε κελί (αν έχει τιμή) να αναπαριστά μια ακμή.

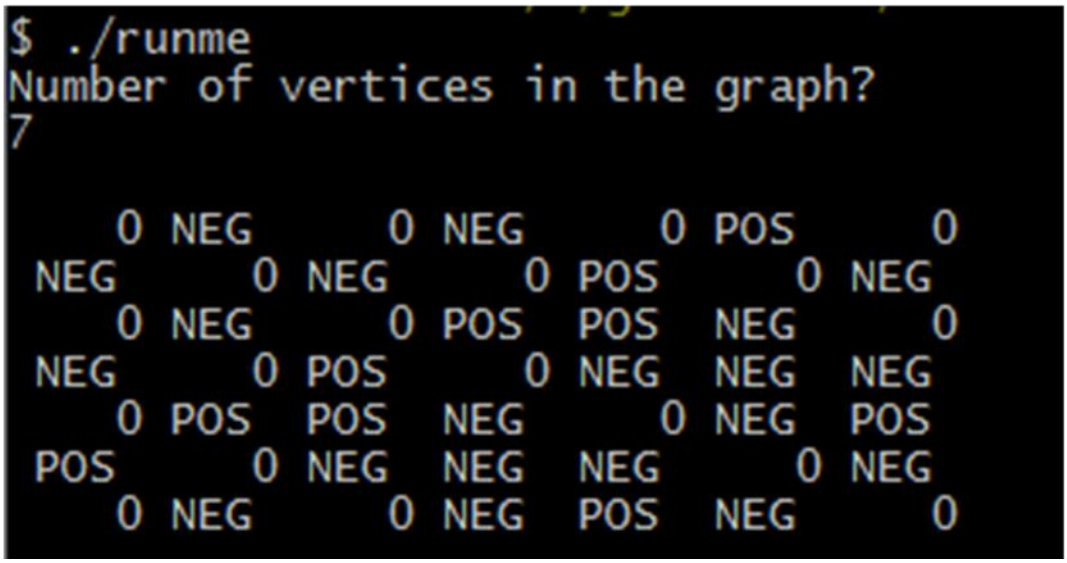


Ανάπτυξη κώδικα

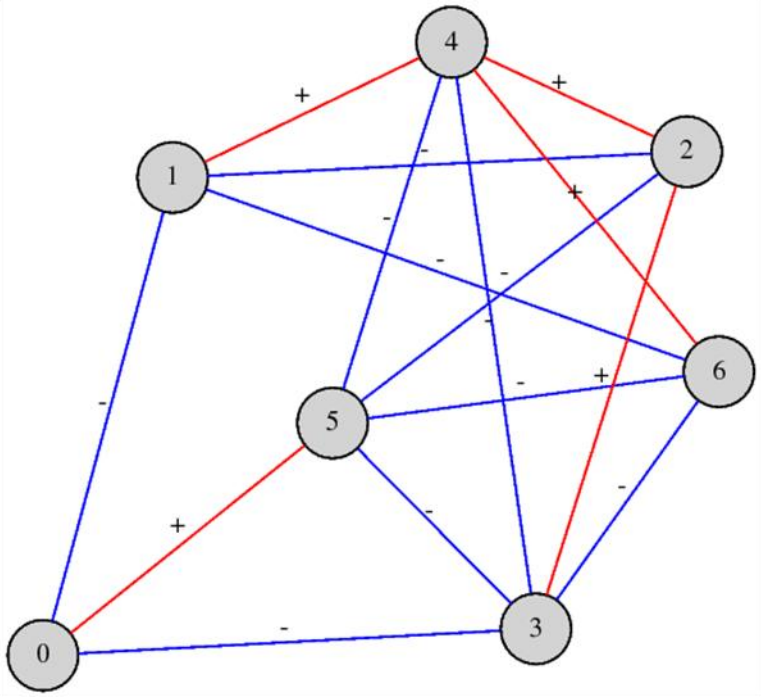
Ανάπτυξη αλγόριθμου δημιουργίας τυχαίων γράφων:

1. Επιλογή τυχαίου ***αριθμού*** αρχικών συνδέσεων του παρόντος κόμβου (ελάχιστο 2 συνδέσεις – μέγιστο 6*)
2. Επιλογή τυχαίου κόμβου *j*
3. Δημιουργία σύνδεσης (παρών, j)
4. Επανάληψη 2 και 3 όσες φορές είναι ο ***αριθμός***
5. Επανάληψη για όλους τους κόμβους

Ανάπτυξη κώδικα



Ανάπτυξη κώδικα



Ανάπτυξη κώδικα

Περιορισμοί κατά την δημιουργία γράφων:

- Ελάχιστος αριθμός κόμβων = 3

  Ο αριθμός κόμβων που απαιτείται ώστε να υπάρξει ο μικρότερος δυνατός κλειστός γράφος.

- Μέγιστος αριθμός κόμβων = 18000

  Γύρω στο 1.5 GB δεσμευμένης μνήμης.

- Ελάχιστος αριθμός συνδέσεων = 2

  Ώστε να αποφύγουμε ακμές με ένα άκρο.

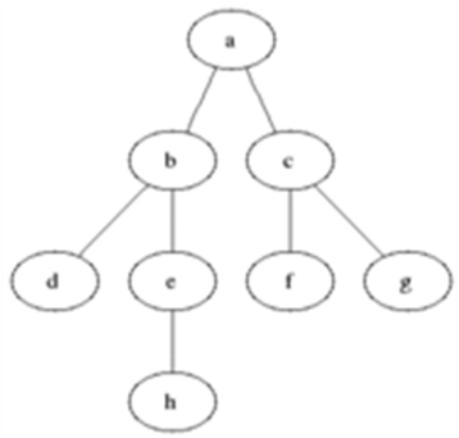
  **Ο αρχικός περιορισμός του μέγιστου αριθμού συνδέσεων στους 6 κόμβους δεν είναι αυστηρός, καθώς στο τέλος της δημιουργίας ενός γράφου πχ. 100 κόμβων συνήθως οι περισσότεροι κόμβοι έχουν πολλές συνδέσεις παραπάνω.*

Ανάπτυξη κώδικα

Αλγόριθμοι ελέγχου ισορροπίας:

- BFS

  Βασισμένος στον *A Dynamic Programming Algorithm to Test a Signed Graph for Balance* από τον κ. Ε. Λουκάκη



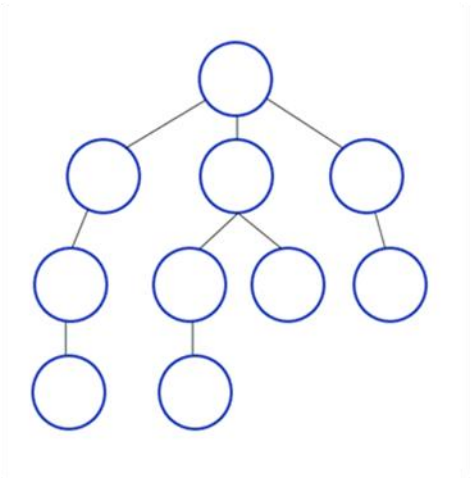
Ανάπτυξη κώδικα

```
Algorithm: Signed_Graph_Balance (H, s, balance)
begin
        visited[s] := true; balance := true; σ[s] := + or -
        for each x ∈V(G) – {s} do
                visited[x] := false
        endfor
        first := last := 1
        for each x ∈Adj[s] do
                σ[x] := σ[s] * σ[(s,x)]; last := last + 1;
                Q[last] := x; visited[x] := true
        endfor
        while (first <= last) and (balance := true) do
                x := Q[first]; first := first + 1;
                visited[x] := true
        for each y ∈Adj[x] do
                if visited[y] := false then
                        visited[y] := true; last := last +1; Q[last] := y;
                        σ[y] := σ[x] * σ[(x,y)]
                else
                if σ[x] * σ[(x,y)] != σ[y] then
                        balance := false
                endif
              endif
        endfor
        endwhile
end
```

Ανάπτυξη κώδικα

- DFS

  Βασισμένος στον κ. Ι. Ξεζωνάκη



Ανάπτυξη κώδικα

Κάθε κόμβος χαρακτηρίζεται ως V εάν τον έχω ήδη επισκεφθεί και ως NV εάν δεν τον έχω επισκεφθεί. Χαρακτηρίζεται επίσης από το πρόσημο του. Πρόσημο έχουν μόνο οι V κόμβοι, διότι το αποκτούν κατά τη στιγμή της "επίσκεψης" σε αυτούς. Χρησιμοποιούνται δύο στοίβες για την υλοποίηση του αλγόριθμου, η A και η B. Όσο υπάρχουν NV κόμβοι, επιλέγω έναν κόμβο k και τον χαρακτηρίζω αυθαίρετα ως +.

Μετά:

- *Ελέγχω κάθε κόμβο, στον οποίο φθάνω από τον k μέσω ακμής θετικού προσήμου:*
  - *Αν αυτός ο κόμβος είναι NV, παίρνει πρόσημο sign(k)*(+), γίνεται V και τοποθετείται στη στοίβα A.*
  - *Αν είναι V, ελέγχεται εάν έχει πρόσημο +. ΑΝ όχι, υπάρχει λάθος και ο γράφος είναι μη ισοζυγισμένος.*

- *Ελέγχω κάθε κόμβο, στον οποίο φθάνω από τον k μέσω ακμής αρνητικού προσήμου:*
  - *Αν αυτός ο κόμβος είναι NV, παίρνει πρόσημο sign(k)*(-), γίνεται V και τοποθετείται στη στοίβα B.*
  - *Αν είναι V, ελέγχεται εάν έχει πρόσημο -. Αν όχι, υπάρχει λάθος και ο γράφος είναι μη ισοζυγισμένος.*

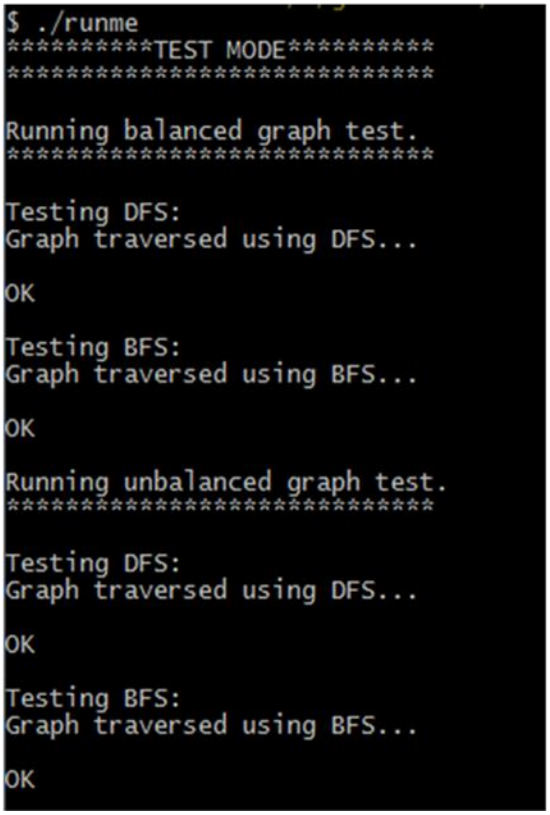
Παίρνω τον επόμενο κόμβο στη στοίβα A και επαναλαμβάνω την παραπάνω διαδικασία, θεωρώντας αυτόν ως κόμβο k. Αν η στοίβα A είναι κενή, παίρνω κόμβους από τη στοίβα B. Η διαδικασία επαναλαμβάνεται μέχρι όλοι οι κόμβοι να είναι V. Αν όλοι οι κόμβοι έχουν σημειωθεί ως V και δεν έχει "κολλήσει" ο αλγόριθμος πουθενά, τότε ο γράφος είναι ισοζυγισμένος.

Επαλήθευση ορθότητας αλγορίθμων

Πρέπει να ελέγξουμε την ορθότητα των αλγορίθμων ελέγχου ισορροπίας που χρησιμοποιούμε. Ο τρόπος που το καταφέρνουμε είναι με το να τους τρέξουμε πάνω σε γράφους των οποίων την ισορροπία ξέρουμε ήδη.

make test
./runme

Επαλήθευση ορθότητας αλγορίθμων



Εν τέλει

Όλος ο πηγαίος κώδικας είναι διαθέσιμος στο Github

github.com/kargakis/signed-graph-balance-check

Ερώτήσεις? ☺