ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Αξιολόγηση ενός αλγορίθμου υπολογισμού ισχυρά συνεκτικών συνιστωσών παρουσία αφαιρέσεων ακμών

Evaluation of a decremental strongly connected components algorithm

Διπλωματική Εργασία

Δημήτριος Κ. Μπέλλος

Επιβλέποντες Καθηγητές: Παναγιώτης Δ. Μποζάνης
Αναπληρωτής Καθηγητής

Παναγιώτα Τσομπανοπούλου
Επίκουρη Καθηγήτρια

Βόλος, Οκτώβριος 2013

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες καθηγητές της διπλωματικής εργασίας κ. Παναγιώτη Μποζάνη και κ. Παναγιώτα Τσομπανοπούλου, για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου, αλλά και αυτά που μας δίδασκαν όλα αυτά τα χρόνια σε συνδυασμό με άλλους καθηγητές, δηλαδή "την τέχνη του ψαρέματος".

Επίσης θα ήθελα να πω ένα μεγάλο ευχαριστώ στην οικογένεια μου που στήριζε αδιαλείπτως την πορεία των σπουδών μου, αλλά και τους φίλους μου που κάνανε αυτήν την πορεία πιο ομαλή.

Περίληψη

Σε αλγόριθμους που επεξεργάζονται γραφήματα είναι σύνηθες φαινόμενο να επικεντρώνονται στην γρήγορη και σωστή βελτιστοποίηση των γραφημάτων, τα οποία υπόκεινται προσθήκες/διαγραφές διάφορων στοιχείων. Σε αυτήν την εργασία το κεντρικό θέμα είναι η διατήρηση των ισχυρά συνεκτικών συνιστωσών σε κατευθυνόμενα γραφήματα τα οποία υπόκεινται διαγραφή ακμών. Πιο συγκεκριμένα, η βελτίωση ενός αλγορίθμου, με την παρουσίαση ενός νέου, ώστε να γίνει ταχύτερος, και το πως αξιολογείται αυτό ως προς τις προηγούμενες έρευνες και αποτελέσματα.

Περιεχόμενα

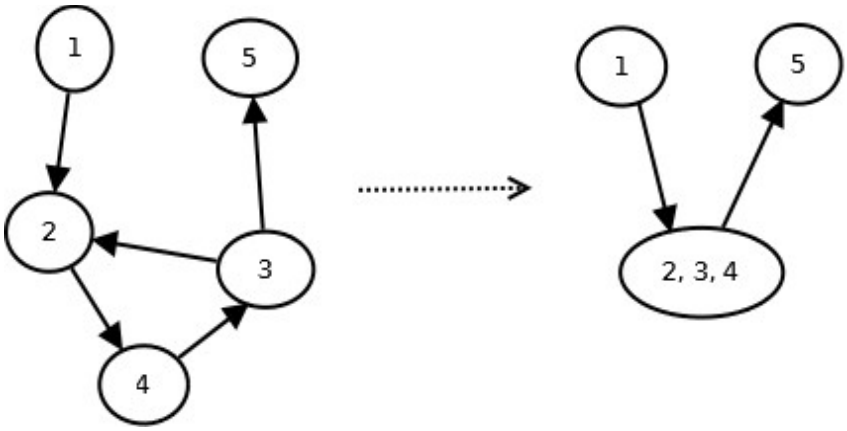
1. Εισαγωγή

1.1 Περιγραφή του προβλήματος

Στον κόσμο των υπολογιστών, και συγκεκριμένα στο προγραμματιστικό σκέλος, χρησιμοποιούνται διάφορες μεθοδολογίες για την υλοποίηση ιδεών ως προγράμματα. Αυτού του είδους οι μεθοδολογίες ονομάζονται αλλιώς και ως ”αλγόριθμοι”. Άτυπα, με τον όρο αλγόριθμος χαρακτηρίζεται κάθε καλώς ορισμένη υπολογιστική διαδικασία, η οποία καλείται να επιλύσει ένα συγκεκριμένο πρόβλημα εντός πεπερασμένου χρόνου[5]. Φυσικά ο άνθρωπος χρησιμοποιούσε πάντα αλγόριθμους, πολύ πριν την εμφάνιση υπολογιστικών μηχανών. Όμως με την εμφάνιση του υπολογιστή, και την ικανότητα να προγραμματίζει πάνω σε αυτόν, άρχισε να δημιουργεί ή να χρησιμοποιεί περισσότερο αλγόριθμους για να λύνει πιο εύκολα (και γρήγορα!!) τα διάφορα προβλήματα του.

Γενικά, οι αλγόριθμοι γράφονται ή αναπαρίστανται είτε σε φυσική γλώσσα είτε με την μορφή ψευδοκώδικα, δηλαδή σε προγραμματιστική ψευδογλώσσα. Η συγκέντρωση αλγορίθμων σε διάφορες “αποθήκες”, για την εύκολη χρησιμοποίηση τους σε εκάστωτε περιπτώσεις γινόταν με βάση τον παραπάνω τρόπο περιγραφή τους. Σαν συνέπεια, μία σημαντική προσθήκη στην περιγραφή του κάθε αλγορίθμου ήταν και η αναπαράστασή του σχηματικά, αναλόγως με το πρόβλημα που επίλυε. Έτσι, λοιπόν, άρχισαν να κατηγοριοποιούνται ακόμη πιο εύκολα, ώστε να είναι διακριτοί.

Το θέμα που αναπτύσσεται σε αυτή την εργασία αφορά το πρόβλημα *της διατήρησης των ισχυρά συνεκτικών συνιστωσών(ΙΣΣ) σε κατευθυνόμενα γραφήματα που υποβάλλονται διαγραφές ακμών*, και σχετίζεται με την βελτίωση ενός αλγόριθμου πάνω σε γραφήματα ή αλλιώς “γραφαλγόριθμος”. Αυτό είναι ένα θεμελιώδες πρόβλημα πάνω σε δυναμικούς γραφαλγόριθμους, και για μερικούς είναι στοιχείο-κλειδί η διατήρηση της ικανότητας μετάβασης μεταξύ κορυφών ενός γραφήματος, όπως φαίνεται στο Σχήμα 1.1.



Σχήμα 1.1. *Μία ισχυρά συνεκτική συνιστώσα είναι μία ομάδα κορυφών οι οποίες συνδέονται με ακμές μεταξύ τους, με τέτοιο τρόπο ώστε να μπορεί οποιαδήποτε κορυφή από αυτές να μεταβεί σε οποιαδήποτε άλλη. Στο παράδειγμα οι κορυφές 2,3 και 4 έχουν αυτή την ιδιότητα μεταξύ τους και αποτελούν την μοναδική ισχυρα συνεκτική συνιστώσα(ΙΣΣ) του γραφήματος. Δηλαδή δεν είναι επιθυμητό να σπάσει η ισχυρά συνεκτική συνιστώσα (2,3,4)*

Το πρόβλημα της μείωσης κυριαρχίας των ισχυρά συνεκτικών συνιστωσών αναφέρθηκε αναμφίβολα πρώτη φορά από τον Frigioni[4]. Εκεί παρουσιάστηκε ένας αλγόριθμος με συνολικό χρόνο βελτιστοποίησης του γραφήματος O(m), ο οποίος χρόνος υπολογισμού είναι τόσο αργός, όσο με το να υπολογίζεται μετά από κάθε διαγραφή ακμής, ξανά, όλες οι ισχυρά συνεκτικές συνιστώσες. Παρόλα αυτά, έδειξε ότι αν όλες οι ακμές που διαγράφονται επιλέγονται τυχαία, τότε ο συνολικός χρόνος υπολογισμού είναι O(mn). Έπειτα ο Roditty μαζί με τον Zwick [6] παρουσίασαν τον αλγόριθμο *Las-Vegas* με μέσο συνολικό χρόνο βελτιστοποίησης O(mn), και ανταποκρίνεται σε κάθε αναζήτηση σε χρόνο O(1) στην χειρότερη περίπτωση. Βασιζόμενος στην αυθαίρετη επιλογή ακμών, ο αλγόριθμος ήταν σχετικά ικανοποιητικός.

Πρόσφατα ο Lacki παρουσίασε [2] έναν ντετερμινιστικό αλγόριθμο ο οποίος έχει συνολικό χρόνο βελτιστοποίησης O(mn), και έτσι λύνει το πρόβλημα που τέθηκε από τον Roditty και Zwick με τον Las-Vegas αλγόριθμο[6]. Το πλεονέκτημα του αλγορίθμου του Lacki απέναντι των Roditty και Zwick[6] είναι φανερό, αφού ως ντετερμινιστικός αλγόριθμος δουλεύει σε ένα πιο γενικό δυναμικό μοντέλο με τον ίδιο συνολικό χρόνο βελτιστοποίησης γραφήματος. Από την άλλη υποφέρει από 2 σημαντικά ελαττώματα.

2

Πρώτον, ο χρόνος προεπεξεργασίας του γραφήματος είναι O(mn). Δεύτερον, ο χρόνος βελτιστοποίησης της χειρότερης περίπτωσης μετά από διαγραφή μίας ακμής μπορεί να είναι O(mn) επίσης. Δεν είναι δύσκολο να δει κανείς ότι ο αλγόριθμος των Roditty και Zwick[6] δεν έχει αυτά τα ελαττώματα, αφού βασίζεται στην μείωση κυριαρχίας των αναζήτηση κατά πλάτος δέντρα, τα οποία μπορούν να δημιουργηθούν σε χρόνο O(m) και να βελτιστοποιηθούν μετά από μία διαγραφή ακμών σε χρόνο O(m) στην χειρότερη περίπτωση[3]. Βασισμένος στα παραπάνω συμπεράσματα, ο Roditty[1] καταλήγει στο ακόλουθο θεώρημα:

ΘΕΩΡΗΜΑ 1.1. *Υπάρχει ένας ντετερμινιστικός αλγόριθμος ο οποίος προεπεξεργάζεται ένα κατευθυνόμενο γράφημα G = (V, E) σε χρόνο O(m*log*n) μέσα σε μία δομή δεδομένων μεγέθους O(m + n), η οποία διατηρεί τις ισχυρά συνεκτικές συνιστώσες του G κατά την διάρκεια διαγραφής ακμών με συνολικό χρόνο βελτιστοποίησης του γραφήματος O(mn), την χειρότερη περίπτωση σε χρόνο O(m*log*n) και χρόνο αναζήτησης O(1).*

Ο χρόνος προεπεξεργασίας και ο χρόνος βελτιστοποίησης στην χειρότερη περίπτωση είναι θεωρητικά και πρακτικά πολύ σημαντικά θέματα. Στο πρακτικό σκέλος, όταν δεχόμαστε σαν είσοδος ένα γράφημα G, δεν έχουμε στοιχεία για τα πόσες ακμές θα διαγραφούν από το G. Σε αυτήν την περίπτωση, η επένδυση χρόνου O(mn) για την προεπεξεργασία του γραφήματος και η δημιουργία μίας δυναμικής δομής δεδομένων μπορεί να θεωρηθεί σπατάλη χρόνου, αν μετά από αυτό διαγραφεί ένας μικρός αριθμός ακμών από το γράφημα. Πιο συγκεκριμένα, αν ο χρόνος προεπεξεργασίας είναι O(mn), τότε σε περίπτωση που ο αριθμός ακμών που διαγραφούν από το γράφημα είναι μικρότερος του Ρίζα(n), θα ήταν πιο επαρκές αν χρησιμοποιόταν ένας στατικός αλγόριθμος για κάθε ισχυρά συνεκτική συνιστώσα. Βέβαια, χωρίς τον χρόνο προεπεξεργασίας, θα ήταν πιο γρήγορος από την χρήση ενός στατικού αλγόριθμου. Χρησιποιώντας έναν επαρκή καινούριο αλγόριθμο προεπεξεργασίας, που δημιούργησε ο Roditty, αποφεύεται το προηγούμενο σενάριο και πλεόν υπάρχει και γρήγορος χρόνος προεπεξεργασίας και επάρκεια μιας δυναμικής δομής δεδομένων. Από μία άλλη πρακτική άποψη, το οτί μία απλή διαγραφή ακμής μπορεί να πάρει χρόνο O(mn), μπορεί σε μερικές εφαρμογές να είναι απαγορευμένο.

Από θεωρητική άποψη αυτό το πρόβλημα είναι ενδιαφέρον, καθώς συνδέεται στενά με το πρόβλημα της διατήρησης της μεταβατικής κλειστότητας ενός κατευθυνόμενου γραφήματος. Ο συνολικός χρόνος της μείωσης κυριαρχίας των ισχυρά συνεκτικών συνιστωσών της μεταβατικής κλειστότητας είναι O(mn)[6, 2], οποίος είναι και ο καλύτερος τον οποίο μπορεί να επιτευχθεί. Από την άλλη όμως, η μείωση κυριαρχίας των ισχυρά συνεκτικών συνιστωσών είναι ένα διαφορετικό πρόβλημα, και δεν υπάρχει κάποιος φανερός λόγος για τον οποίο θα έπρεπε ο χρόνος βελτιστοποίησης να παραμείνει O(mn), αφού ο υπολογισμός των ισχυρά συνεκτικών συνιστωσών χρειάζεται χρόνο O(m). Αναγκαία προϋπόθεση για την δημιουργία ενός ντετερμινιστικού αλγορίθμου με συνολικό χρόνο βελτιστοποίησης $O(mn^{1-\varepsilon})$, όπου $\varepsilon > 0$, είναι η απόκτηση ενός ντετερμινιστικού αλγορίθμου με χρόνο προεπεξεργασίας $O(mn^{1-\varepsilon})$.

Ο Roditty[1] δημιούργησε έναν ντετερμινιστικό αλγόριθμο προεπεξεργασίας κατευθυνόμενου γραφήματος, ανάλογο του Lacki[2], και μέσα από λογικά συμπεράσματα, δείχνει ότι αυτός είναι εξίσου αποτελεσματικός, αλλά σαφέστατα γρηγορότερος με τους χρόνους που δείχνει το Θεώρημα 1.1.

1.2 Σκοπός και διάρθρωση της εργασίας

Στόχος αυτής της εργασίας είναι η ανάλυση του αλγορίθμου του Roditty[1] συκριτικά με αυτόν του Lacki σε θεωρητικό και πρακτικό επίπεδο, και η αξιολόγησή του. Συγκεκριμένα η περιεκτική περιγραφή του αλγορίθμου βήμα προς βήμα, σύγκριση και των δύο, και μαζί με τα λογικά συμπεράσματα και αποδείξεις, τα οποία οδηγούν στα αποτελέσματα τους, δηλαδή τους χρόνους εκτέλεσης σε ασύμπτωτο χρόνο για το κυρίως πρόβλημα, την μείωση κυριαρχίας των ισχυρά συνεκτικών συνιστωσών κατά την διάρκεια διαγραφής ακμών του γραφήματος. Στο κεφάλαιο 2, αφού αναφερθούμε στον αλγόριθμο του Lacki[2], έπειτα περιγράφουμε αναλυτικά τον αλγόριθμο του Roditty[1], και κατόπιν γίνεται η σύγκριση.

Στο πρακτικό κομμάτι, στο 3ο κεφάλαιο, το πρώτο βήμα της εργασίας είναι η προσομοίωση των αλγορίθμων σε προγραμματιστικό επίπεδο. Γι' αυτό το λόγο χρησιμοποίηθηκε μία βιβλιοθήκη της C++, η LEDA. Η βιβλιοθήκη LEDA είναι ένα περιβάλλον κατάλληλο για ανάπτυξη και υλοποίηση δομών δεδομένων και ανάλυση αλγορίθμων, όπως στην προκειμένη περίπτωση. Περιέχει μία μεγάλη γκάμα επιλογών ως προς την προτίμηση του χρήστη, και αυτός είναι ο λόγος που επιλέχθηκε για την δική μας προσομοίωση. Έπειτα, αφού προσομοιωθούν οι αλγόριθμοι στην προγραμματιστική γλώσσα C++, τρέχουμε διάφορα πειράματα με πληθώρα γραφημάτων, και συγκρίνονται οι χρόνοι τους, για να εξακριβώσουμε αν όντως ισχύουν αυτά που αναφέρθηκαν στο 2ο κεφάλαιο.

Τέλος, στο 4ο κεφάλαιο, βγάζουμε τα συμπεράσματα μας βασιζόμενοι στην ανάλυση του 2ου κεφαλαίου και στις υλοποιήσεις και αποτελέσματα του 3ου.

2. Ανάλυση αλγορίθμων

2. Προκαταρκτικές ορολογίες

Παραθέτουμε κάποιες ορολογίες οι οποίες είναι απαραίτητες για την κατανόηση της ανάλυσής μας:
Έστω ένα κατευθυνόμενο γράφημα G = (V, E) με 'V' τις κορυφές του και 'E' τις ακμές του. Έστω (V^{SCC}, E^{SCC}) = SCC(G) είναι οι ισχυρά συνεκτικές συνιστώσες(ΙΣΣ) του γραφήματος G, όπου οι κορυφές V^{SCC} είναι αντιπροσωπούν τις ΙΣΣ του G, και οι E^{SCC} τις ακμές που προβάλλονται από μία ΙΣΣ σε μία άλλη, και συνδέουν αυτές τις συνιστώσες. Για ευκολία θα χρησιμοποιούμε τις κορυφές του V^{SCC} σαν σετ(ομάδες) κορυφών από το G, όπου εάν ένα σετ κορυφών C ανήκει στο V^{SCC} και μία κορυφή u ανήκει στο V και σε ένα σετ από V^{SCC} του G, όπως το C, τότε μπορούμε να πούμε ότι το u ανήκει στο C. Επίσης να αναφέρουμε ότι μία κορυφή υ αποτελεί από μόνη της ένα σετ κορυφών το οποίο περιέχει τις κορυφές{υ}. Όταν ένα σετ C περιέχει υποσύνολα σετ ομάδων από ΙΣΣ, όπου, C={C_1, ...,C_l} και κάθε $C_ι$ μπορεί να είναι μία ισχυρά συνεκτική συνιστώσα, με το VL(C) υποδεικνύουμε τις κορυφές του αρχικού γραφήματος εισόδου G στο οποίο ανήκει το C. Σε τέτοια περίπτωση C_i ανήκει στο C, σημαίνει ότι το C_i είναι στο πρώτο επίπεδο του C. Για μία κορυφή υ ανήκει στο V, χρησιμοποιούμε το G\υ για να υποδείξουμε ότι το γράφημα που είναι ως αποτέλεσμα από την διαγραφή της κορυφής υ και όλων των ακμών της στο G. Για μία ακμή e ανήκει στο E, χρησιμοποιούμε το G\e για να υποδείξουμε το γράφημα που είναι ως αποτέλεσμα της διαγραφής της ακμής e από το G. Όταν λέγεται η λέξη ιεραρχία, τότε σημαίνει ότι δημιουργείται το δέντρο εκτέλεσης του αλγορίθμου.

2.1 Αλγόριθμος του Lacki

2.1.1 Περιγραφή του αλγορίθμου

Αρχικά θα περιγράψουμε την προσέγγιση του Lacki[2] σχετικά με το πρόβλημα που μας ενδιαφέρει, δηλαδή την μείωση κυριαρχίας των ισχυρά συνεκτικών συνιστωσών. Ο αλγόριθμος αρχίζει χτίζοντας μία γραφική αναπαράσταση η οποία μας επιτρέπει να βρούμε τις αλλαγέςστην δομή των ισχυρά συνεκτικών συνιστωσών, και αυτή η αναπαράσταση ορίζεται αναδρομικά. Για την αναπαράσταση μίας μεγάλης ισχυρά συνεκτικής συνιστώσας, επιλέγεται αυθαίρετα μία κορυφή υ και διασπάζεται σε δύο άλλες κορυφές, τις $υ_{in}$ και την $υ_{out}$. Οι ακμές, μετά τη διάσπαση της κορυφής υ, μοιράζονται στις άλλες δύο αναλόγως με το αν εισέρχονται(in) ή αν εξέρχονται(out) από αυτήν. Έπειτα βρίσκουμε όλες τις ΙΣΣ και τις συνδέουμε στις 2 καινούριες κορυφές, δημιουργώντας ένα κατευθυνόμενο άκυκλο γράφημα(ΚΑΓ) από το γράφημα G το οποίο μπορεί να χρησιμοποιηθεί ως ένα καλό πρότυπο.

ΟΡΙΣΜΟΣ 1. (ΔΙΑΣΠΑΣΜΕΝΟ ΓΡΑΦΗΜΑ) *Έστω μία κορυφή υ ανήκει στο V είναι μία τυχαία κορυφή του G. Έστω (V^{SCC}, E^{SCC}) = SCC(G\υ). Το διασπασμένο γράφημα $G_υ$= ($V_υ$, $E_υ$) ορίζεται ως εξής. Το σετ κορυφών $V_υ$ είναι ίσο με V^{SCC} ένωση με{$υ_{in}$, $υ_{out}$}. Το σετ ακμών $E_υ$ ισούται με E^{SCC} ενωμένο με H, όπου:*

H = {(C,u_{in}) | C ανήκει στο V^{SCC} τομή ∃ (u,υ) ανήκει στο E τομή u ανήκει στο C}
ένωση {u_{out}, C | C ανήκει V_SCC τομή ∃ (υ,u)ανήκει E τομή u ανήκει C}

Μία ακμή (a,b) ανήκει E τοποθετείται σε μία ακμή $E_υ$ αν α = υ ή b = υ ή α ανήκει C και b ανήκει C' ,όπου C, C' ανήκουν V_SCC και C != C'.

Φαίνεται ξεκάθαρα από το παραπάνω ότι το $G_υ$ είναι ένα ΚΑΓ. Αναφερόμαστε στο $G_υ$ ως το διασπασμένο γράφημα G γύρω από την κορυφή υ. Η επόμενη παρατήρηση δείχνει ότι όντως $G_υ$ είναι ένα καλό πρότυπο της ισχυρής συνεκτικότητας του G.

ΠΑΡΑΤΗΡΗΣΗ 1. *Έστω G = (V, E) ένα ισχυρά συνεκτικό γράφημα και έστω υ ανήκει V μία αυθαίρετη κορυφή του. Τότε* $G_\upsilon = (V_\upsilon, E_\upsilon)$ *είναι το διασπασμένο γράφημα του G γύρω από την κορυφή υ. Το γράφημα G είναι ισχυρά συνεκτικό αν και μόνο αν η κορυφή* υ_{out} *μπορεί να μεταβεί σε κάθε κορυφή του* G_υ *και κάθε κορυφή του* G_υ *μπορεί να μεταβεί στην* u_{in}.

Χρησιμοποιώντας τον ορισμό του διασπασμένου γραφήματος μπορούμε να πάρουμε μία ιεραρχική αποσύνθεση για ένα ισχυρά συνεκτικό γράφημα G = (V, E). Επιλέγοντας αυθαίρετα μία κορυφή υ ανήκει V για την οποία κατασκευάζουμε ένα διασπασμένο γράφημα G_υ. Η ρίζα αυτής της ιεραρχίας έχει τις ακόλουθες πληροφορίες:

- Το όνομα της συνιστώσας V
- Η διασπασμένη κορυφή υ
- Το διασπασμένο γράφημα $G_\upsilon = (V_\upsilon , E_\upsilon)$
- Τις ακμές του E που βρίσκονται και στο E_υ

Έστω $(V^{SCC}, E^{SCC}) = SCC(G\backslash\upsilon)$, όπου $V^{SCC} = \{C_1, C_2, \ldots, C_l\}$. Έπειτα δημιουργούμε ένα κόμβο στην ιεραρχία για κάθε C ανήκει V^{SCC}. Αυτό επιτυγχάνεται με την αυθαίρετη επιλογή μίας κορυφής s ανήκει C και υπολογίζοντας το διασπασμένο γράφημα C_s. Μέσα στον κόμβο της ιεραρχίας στο οποίο διαμορφώνεται η C διατηρούμε την ίδια πληροφορία όπως και για την ρίζα. Θεωρούμε ότι κάθε κόμβος περιέχει ένα δείκτη προς τον πατέρα του και δείκτες προς τα παιδιά του. Η επεξεργασία λειτουργεί με αναδρομικό τρόπο για κάθε κόμβο της ιεραρχίας του οποίου το διασπασμένο γράφημα περιέχει μιά κορυφή η οποία ισοδυναμεί σε ένα σετ των 2 ή περισσότερων κορυφών του V. Δοθέντος μίας ΙΣΣ μεγέθους l στην ιεραρχία, είναι πιθανό να απαριθμήσει τις κορυφές της σε χρόνο O(l). Ένας ψευδοκώδικας από το [1] δίνεται παρακάτω:

2.1.2 Ο αλγόριθμος σε ψευδοκώδικα

Αλγόριθμος 2.1

. PreProcess($G^{INP} = (V^{INP}, E^{INP})$*)*

Let v be an arbitrary vertex from V^{INP};
$(V^{\text{SCC}}, E^{\text{SCC}}, E^{\text{INTRA}}) \leftarrow SCC(G^{\text{INP}} \setminus v)$;
for all $C \in V^{\text{SCC}}$ **do**
  $p(C) \leftarrow V^{\text{INP}}$;
end for
$s(V^{\text{INP}}) = v$;
$V_v^{\text{INP}} \leftarrow V^{\text{SCC}} \cup \{v_{in}, v_{out}\}$;
$E_v^{\text{INP}} \leftarrow E^{\text{SCC}} \cup \{(v_{out}, C) \mid (v, x) \in E^{\text{INP}} \wedge x \in C\} \cup \{(C, v_{in}) \mid (x, v) \in E^{\text{INP}} \wedge x \in C\}$;
for all $C \in V^{\text{SCC}}$ **do**
  PreProcess$(C, E^{\text{INTRA}}(C))$, where $E^{\text{INTRA}}(C) \subseteq E^{\text{INP}}$ is the set of edges with both endpoints in C;
end for

Έπειτα δίνεται η πολυπλοκότητα του αλγορίθμου και αναλύεται η κατασκευή αυτής της ιεραρχίας από τον παραπάνω αλγόριθμο[2, 1] με παράδειγμα.

2.1.3 Πολυπλοκότητα

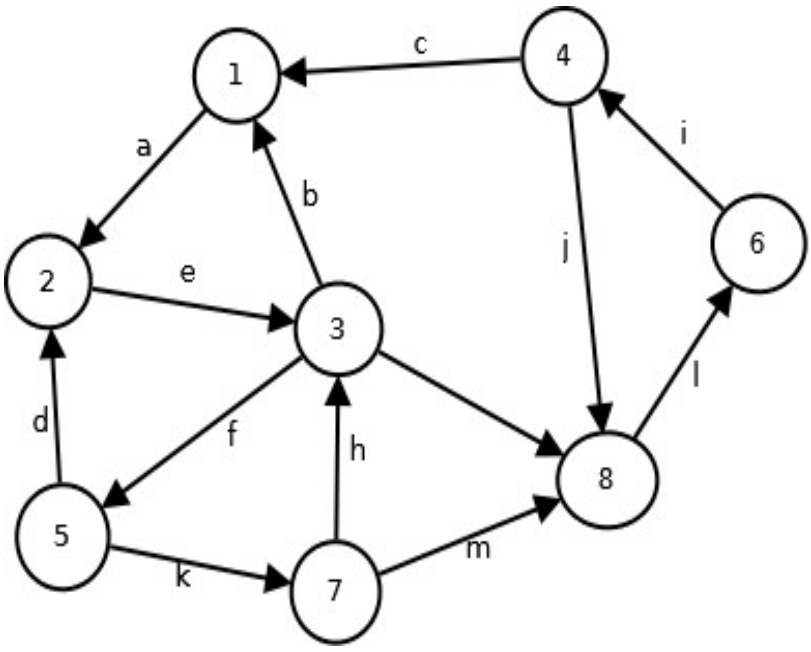
Η μέση και χειρότερη περίπτωση τρεξίματος του αλγορίθμου ως προς τον χρόνο περιγράφεται από το Λήμμα 2.1.

ΛΗΜΜΑ 2.1 *Ο χρόνος που χρειάζει ο αλγόριθμος 2.1 για να κατασκευάσει την ιεραρχία εκτέλεσης είναι O(mn) στην χειρότερη περίπτωση.*

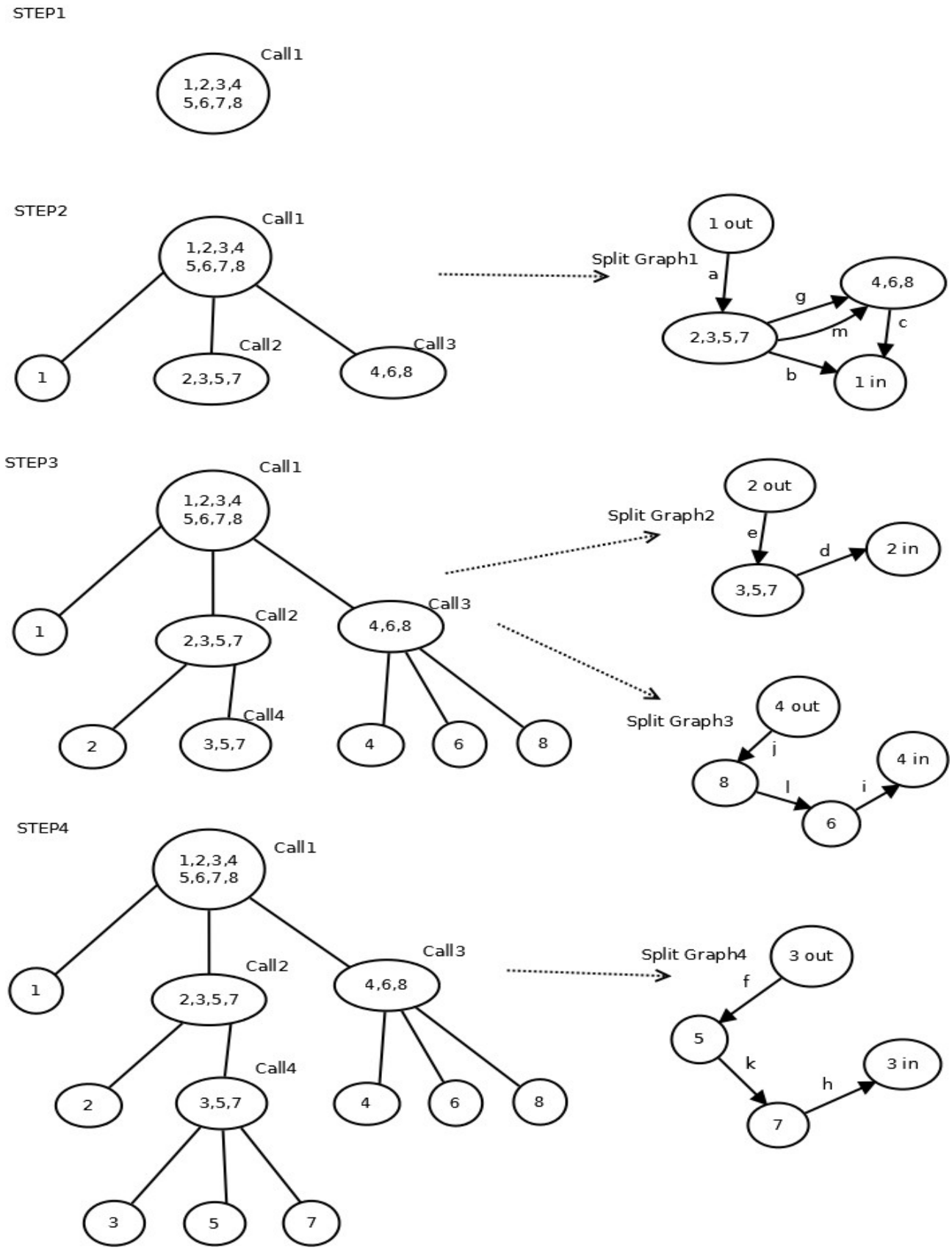
ΑΠΟΔΕΙΞΗ. 'Εστω h το βάθος της ιεραρχίας και C είναι μία ΙΣΣ σε βάθος i της ιεραρχίας, όπου 1<= i <= h. Το κύριο κόστος στην δημιουργία ενός κόμβου μίας ΙΣΣ C στην ιεραρχία είναι το κόστος υπολογισμού του διασπασμένου γραφήματος C_s γύρω από την κορυφή s ανήκει C. Αυτό το κόστος είναι γραμμικό ως προς τον αριθμό ακμών του E, με όλες τις ακμές να προσπίπτουν μέσα στην C. Ακόμη, μία ακμή δεν μπορεί να υπάρχει σε μία συνιστώ του επιπέδου i. Κάθε ακμή διαπερνάται σε κάθε επίπεδο το πολύ μία φορά, και αν υπάρχουν h επίπεδα, το συνολικό κόστος είναι O(mh). Στην χειρότερη περίπτωση που h = n, τότε ο χρόνος τρεξίματος του αλγορίθμου είναι O(mn).

2.1.4 Παράδειγμα

Παραθέτουμε ένα παράδειγμα, το οποίο είναι η ολοκληρωμένη περιγραφή και συνεχίζει το ένα παράδειγμα του [2], βασισμένο στο Σχήμα 2.1, στο οποίο το διάγραμμα είναι μία ΙΣΣ. Έπειτα στο Σχήμα 2.2 φαίνεται σταδιακά η δημιουργία της ιεραρχίας, ακολουθούμενη απότην εξήγηση του τρεξίματος του αλγορίθμου.



ΣΧΗΜΑ 2.1. *Μία ισχυρά συνεκτική συνιστώσα(ΙΣΣ). Όλες οι κορυφές μπορούν να μεταβούν προς οποιαδήποτε άλλη.*



ΣΧΗΜΑ 2.2. *Στην αριστερή πλευρά του σχήματος σταδιακά δημιουργείται η ιεραρχία(δέντρο εκτέλεσης) μέχρι και το τελικό Step4. Δεξιά βρίσκονται τα διασπασμένα γραφήματα τα οποία έχουν δημιουργηθεί αναλόγως σε κάθε βήμα, π.χ στο Step3 δημιουργούνται 2 διασπασμένα γραφήματα.*

Το τρέξιμο του Αλγορίθμου 2.1 έχει ως εξής:

Step1: Ο αλγόριθμος διαλέγει τυχαία την κορυφή 1. Γύρω από αυτήν θα χτίσει το 1ο διασπασμένο γράφημα το οποίο φαίνεται στο Step2. Το Call1 δείχνει ότι η 1η κλήση έγινε έχοντας ως είσοδο το γράφημα με τις κορυφές που δείχνει, δηλαδή 1,2,3,4,5,6,7 και 8. Έπειτα διαλέγονται οι εσωτερικές ΙΣΣ του Call1 που θα χρησιμοποιηθούν ως είσοδοι για νέες (αναδρομικές)κλήσεις του αλγορίθμου.

Step2: Έχοντας ως εισόδους τις ΙΣΣ (2,3,5,7) και (4,6,8) πραγματοποιούνται ανάλογες κλήσεις του Step1. Η κορυφή 1 αποτελεί πλέον φύλλο του δέντρου εκτέλεσης. Τα διασπασμένα γραφήματα των δύο ΙΣΣ φαίνονται στο Step3. Ομοίως γίνονται τα Call2 και Call3 και επιλέγονται οι εσωτερικές ΙΣΣ, αν υπάρχουν.

Step3: Ομοίως με το Step2, όπου τώρα γίνονται φύλλα οι κορυφές 4,6,8.

Step4: Ομοίως με τα προηγούμενα δύο Step, μόνο που τώρα δεν υπάρχουν νέες κλήσεις. Αυτή είναι και η τελική μορφή του δέντρου εκτέλεσης(ιεραρχία) με 4 επίπεδα βάθους. Παρατηρούμε ότι σε κάθε επίπεδο δημιουργούνται φύλλα λόγω της κατασκευής Split Graph.

Με το τέλος του τρεξίματος του αλγορίθμου δημιουργήθηκαν τέσσερα διασπασμένα γραφήματα, τα οποία είναι ΚΑΓ. Πάνω σε αυτά θα γίνει η κύρια επεξεργασία διαγραφής ακμών όπως προτείνει ο Lacki[2]. Π.χ. στο Split Graph1, θέλοντας να διατηρήσουμε την συνεκτικότητα του, μπορούμε να διαγράψουμε τις ακμές g και b, επιτυγχάνοντας τον αρχικό μας στόχο.

2.2 Αλγόριθμος του Roditty

2.2.1 Περιγραφή του αλγορίθμου

Ο αλγόριθμος του Roditty είναι ένας, σχεδόν, γραμμικός ως προς τον χρόνο αλγόριθμος για την κατασκευή διασπασμένων γραφημάτων της ιεραρχίας εκτέλεσης. Ο αλγόριθμος έχει ως εξής:

Έστω G=(V,E) ένα ΙΣΣ γράφημα και $V=\{\upsilon_1,\upsilon_2, ...,\upsilon_n\}$. Θεωρούμε ότι $id(\upsilon_i) = i$ για κάθε $1<= i <=n$. Ο αλγόριθμος δέχεται ως είσοδο ένα σετ κορυφών V^{INP} το οποίο είναι μία ΙΣΣ και ένα E^{INP} το οποίο ανήκει στην ΙΣΣ. Το id κάθε κορυφής, το οποίο είναι επίσης μία κορυφή του γραφήματος G, είναι είτε το κανονικό του id είτε ∞. Το id μίας που είναι μία ΙΣΣ(είτε του G, έιτε άλλης ΙΣΣ) είναι ∞. Έστω V^{VRTX} υποσύνολο του V^{INP} είναι το σετ κορυφών με id διαφορετικό από το ∞ και V^{LOW} υποσύνολο του V^{VRTX} είναι οι κορυφές με τα $\{|V^{VRTX}|/2\}$(σε περίπτωση μονού αριθμού κορυφών επιλέγεται το μικρότερο μισό σύνολο) μικρότερα ids του V^{VRTX}. Το V^{HIGH} περιέχει τις κορυφές με τα $\{|V^{VRTX}|/2\}$(αντιστοίχως επιλέγεται το μεγαλύτερο μισό σύνολο) υψηλότερα ids του V^{VRTX} και τις υπόλοιπες κορυφές του V^{INP}, που είναι οι κορυφές με id ∞. Έστω E^{HIGH} υποσύνολο του E^{INP} είναι το σετ ακμών με όλες τις ακμές να προσπίπτουν στο V^{HIGH}. Ο αλγόριθμος υπολογίζει τις ΙΣΣ για το γράφημα (V^{HIGH},E^{HIGH}). Θεωρούμε ότι ο αλγόριθμος που υπολογίζει τις ΙΣΣ επιστρέφει ένα σετ από συνιστώσες του V^{SCC}, το σετ ακμών στο εσωτερικών της συνιστώσας E^{INTPA} υποσύνολο του E^{HIGH}. Έπειτα δημιουργούμε ένα νέο γράφημα (V^{NEW}, E^{NEW}). Οι κορυφές του V^{NEW} περιέχουν τις κορυφές του V^{SCC} και τις κορυφές του V^{LOW}. Το σετ ακμών E^{NEW} περιέχει τις ακμές που συνδέουν τις συνιστώσες ΙΣΣ μεταξύ τους, τις ακμές που συνδέουν μεταξύ τους τις κορυφές του V^{LOW} και τις ακμές που συνδέουν τις κορυφές του V^{LOW} με τις ΙΣΣ. Κατόπιν ο αλγόριθμος ελέγχει το σύνολο των κορυφών του V^{LOW} και συνεχίζει αναλόγως με αυτό το μέγεθος, όπου υπάρχουν 2 περιπτώσεις.

Case 1 : [$|V^{LOW}| = 1$].

Σε αυτήν την περίπτωση ο κόμβος της ιεραρχίας που ανταποκρίνεται στην είσοδο ΙΣΣ V^{INP} μπορεί να αποκτηθεί από μία απλή τροποποίηση στο γράφημα (V^{NEW},E^{NEW}). Έστω υ ανήκει $V^{LOW.}$ Η κορυφή υ επιλέγεται για να διασπαστεί s(V^{INP}). Το διασπασμένο γράφημα ($V^{INP}{}_s$, $E^{INP}{}_s$) χτίζεται ομοίως με την διαδικασία που περιγράφηκε στον αλγόριθμο του Lacki.

Case 2: [$|V^{LOW}| > 1$]

Σε αυτήν την περίπτωση ο αλγόριθμος εκτελεί αναδρομική κλήση με είσοδο το γράφημα (V^{NEW},E^{NEW}).

Και στις δύο περιπτώσεις, σαν τελικό βήμα, ο αλγόριθμος διατρέχει όλες τις κορυφές του V^{SCC} και για κάθε συνιστώσα C ανήκει V^{SCC} που δημιουργείται για πρώτη φορά σε αυτήν την κλήση, όπου C den ανήκειV^{INP}, εκτελεί μία αναδρομική κλήση με είσοδο το γράφημα (C,E^{INTRA}(C)), όπου C είναι η συνιστώσα με το σετ κορυφών από το V^{HIGH} που περιέχει το C και το E^{INTRA}(C) είναι το σετ ακμών με τις ακμές να προσπίπτουν μέσα στην C. Ένας ψευδοκώδικας ο οποίος εκτελεί την παραπάνω περιγραφή του αλγορίθμου δίνεται παρακάτω.

2.2.2 Ο αλγόριθμος σε ψευδοκώδικα

Αλγόριθμος 2.2

PreProcess($G^{INP} = (V^{INP}, E^{INP})$)

$V^{VRTX} \leftarrow \{v \mid v \in V^{INP} \wedge id(v) \neq \infty\}$
$V^{LOW} \leftarrow \{v \mid v \in V^{VRTX}$ and v's id is among the $\lfloor |V^{VRTX}|/2 \rfloor$ smallest ids$\}$
$V^{HIGH} \leftarrow V^{INP} \setminus V^{LOW}$
$E^{LOW} \leftarrow \{(x,y) \mid (x,y) \in E^{INP} \wedge x \in V^{LOW} \wedge y \in V^{LOW}\}$
$E^{HIGH} \leftarrow \{(x,y) \mid (x,y) \in E^{INP} \wedge x \in V^{HIGH} \wedge y \in V^{HIGH}\}$
$(V^{SCC}, E^{SCC}, E^{INTRA}) \leftarrow SCC(V^{HIGH}, E^{HIGH})$
for all $C \in V^{SCC}$ **do**
  $id(C) \leftarrow \infty$;
end for
$V^{NEW} \leftarrow V^{LOW} \cup V^{SCC}$
$E^{NEW} \leftarrow \{(x,C) \mid x \in V^{LOW} \wedge C \in V^{SCC} \wedge \exists y \in C$ s.t. $(x,y) \in E^{INP}\}$
$E^{NEW} \leftarrow E^{NEW} \cup \{(C,x) \mid x \in V^{LOW} \wedge C \in V^{SCC} \wedge \exists y \in C$ s.t. $(y,x) \in E^{INP}\}$
$E^{NEW} \leftarrow E^{NEW} \cup E^{SCC} \cup E^{LOW}$
if $|V^{LOW}| = 1$ **then**
  for all $C \in V^{SCC}$ **do**
    $p(C) \leftarrow V^{INP}$
  end for
  $s(V^{INP}) = v$, where $v \in V^{LOW}$
  $V_v^{INP} \leftarrow V^{SCC} \cup \{v_{in}, v_{out}\}$
  $E_v^{INP} \leftarrow E^{SCC} \cup \{(v_{out}, C) \mid (v,C) \in E^{NEW}\} \cup \{(C, v_{in}) \mid (C,v) \in E^{NEW}\}$
else
  PreProcess(V^{NEW}, E^{NEW})
end if

for all $C \in V^{SCC} \wedge C \notin V^{INP}$ **do**
PreProcess$(C, E^{INTRA}(C))$, where $E^{INTRA}(C) \subseteq E^{HIGH}$ is the set of edges with both endpoints in C
end for

Έπειτα δίνεται η πολυπλοκότητα του αλγορίθμου και αναλύεται η κατασκευή αυτής της ιεραρχίας από τον παραπάνω αλγόριθμο[1] με παράδειγμα.

2.2.3 Πολυπλοκότητα

Η ιεραρχία δημιουργείται από την κλήση της συνάρτησης PreProcess με είσοδο το γράφημα G=(V,E). Για την ανάλυση του αλγορίθμου ως προς τον χρόνο εκτέλεσης του δίνεται ο παρακάτω ορισμός.

ΟΡΙΣΜΟΣ 2. (ΔΕΝΤΡΟ ΕΚΤΕΛΕΣΗΣ). *Κάθε κόμβος στο δέντρο εκτέλεσης ανταποκρίνεται σε μία κλήση της συνάρτησης PreProcess. Σχετίζουμε τον κάθε κόμβο N με όλα τα σετ κορυφών τα οποία δέχεται ως είσοδο η PreProcess, όπως επίσης και τα σετ κορυφών τα οποία δημιουργεί κατά την εκτέλεσή της. Το N υποδεικνύει με αυτό τον τρόπο την συσχέτιση. Η ρίζα τουδέντρου ανταποκρίνεται στην αρχική κλήση PreProcess(V,E). Τα παιδιά ενός κόμβου N του δέντρου είναι κόμβοι του δέντρου τα οποία ανταποκρίνονται σε κλήσεις της PreProcess τα οποία αρχικοποιήθηκαν από την κλήση της PreProcess που ανταποκρίνεται στον κόμβο N.*

Παρακάτω παραθέτουμε τα πιο σημαντικά από τα λήμματα που έδειξε ο Roditty στο [1], σύμφωνα με τα οποία ισχυρίζεται ότι ο αλγόριθμος 2.2 είναι ταχύτερος του αλγορίθμου 2.1 :

ΛΗΜΜΑ 2.2.1. *Έστω T το δέντρο εκτέλεσης του PreProcess(V, E). Έστω C υποσύνολο του V είναι μία ΙΣΣ του G ναι υποθέτουμε ότι υπάρχει ένας κόμβος N ανήκει στο T ώστε VL(V^{INP}_N) = C. Ο αλγόριθμος βρίσκει το διασπασμένο γράφημα για κάθε C χρησιμοποιώντας την κορυφή με το μικρότερο id που βρίσκεται στην* C.

ΛΗΜΜΑ 2.2.2 . *Ο αλγόριθμος 2.2 υπολογίζει την ιεραρχία του αλγορίθμου 2.1*

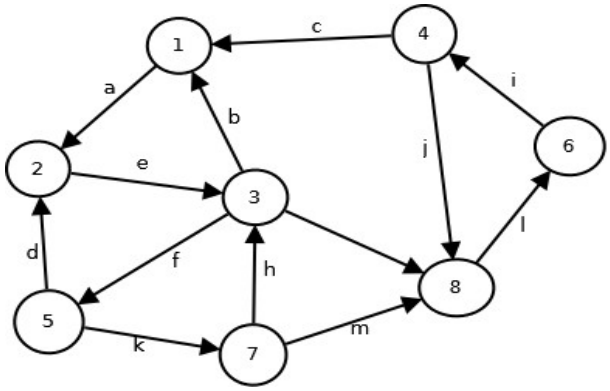
Σύμφωνα με την εξήγηση του παρακάτω παραδείγματος αποδεικνύονται αυτοί οι ισχυρισμοί.

2.1.4 Παράδειγμα

Παραθέτουμε ένα παράδειγμα βασισμένο στο Σχήμα 2.1, στο οποίο το διάγραμμα είναι μία ΙΣΣ. Έπειτα στο Παράδειγμα 1 φαίνεται σταδιακά η δημιουργία της ιεραρχίας μαζί με το τρέξιμο του αλγορίθμου, ακολουθούμενη από την εξήγηση του τρεξίματός του.

ΠΑΡΑΔΕΙΓΜΑ 1

Figure: Input graph



INIT:
$V^{INP} \leftarrow$ {1,2,3,4,5,6,7,8}
$E^{INP} \leftarrow$ {a,b,c,d,e,f,g,h,i,j,k,l,m}

PreProcess(V^{INP}={1,2,3,4,5,6,7,8}, E^{INP}={a,b,c,d,e,f,g,h,i,j,k,l,m}) ### Call 1

$V^{VRTX} \leftarrow$ {1,2,3,4,5,6,7,8}
$V^{LOW} \leftarrow$ {1,2,3,4}
$V^{HIGH} \leftarrow$ {5,6,7,8}
$E^{LOW} \leftarrow$ {a,b,c,e}
$E^{HIGH} \leftarrow$ {k,l,m}

$V^{SCC} \leftarrow$ {(5),(6),(7),(8)}
$E^{SCC} \leftarrow$ {k,l,m}
$E^{INTRA} \leftarrow$ {}

$V^{NEW} \leftarrow$ {1,2,3,4,(5),(6),(7),(8)}
$E^{NEW} \leftarrow$ {a,b,c,d,e,f,g,h,i,j,k,l,m}

IF(V^{LOW} == 1) **? NO**

ELSE
PreProcess(V^{NEW}, E^{NEW}) ### Call 2
END_IF

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
No Calls
END_FOR

Execution tree:

Σχήμα -Step1

PreProcess(V^{INP} = {1,2,3,4,(5),(6),(7),(8)}, E^{INP} = {a,b,c,d,e,f,g,h,i,j,k,l,m}) ### Call 2

$V^{VRTX} \leftarrow$ {1,2,3,4}
$V^{LOW} \leftarrow$ {1,2}
$V^{HIGH} \leftarrow$ {3,4,(5),(6),(7),(8)}
$E^{LOW} \leftarrow$ {a}
$E^{HIGH} \leftarrow$ {f,g,h,i,j,k,l,m}

$V^{SCC} \leftarrow$ {(3,5,7),(4,6,8)}
$E^{SCC} \leftarrow$ {g,m}
$E^{INTRA} \leftarrow$ {f,h,i,j,k,l}

$V^{NEW} \leftarrow$ {1,2,(3,5,7),(4,6,8)}
$E^{NEW} \leftarrow$ {a,b,c,d,e,g,m}

IF(V^{LOW} == 1) **? NO**

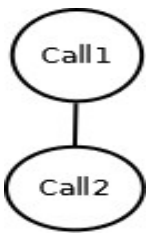
ELSE

PreProcess(V^{NEW}, E^{NEW}) ### Call 3

END_IF

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
PreProcess({3,5,7},{f,h,k}) ### Call 4
PreProcess({4,6,8},{i,j,l}) ### Call 5
END_FOR

Execution tree:



Σχήμα -Step2

PreProcess(V^{INP} = {1,2,(3,5,7),(4,6,8)}, E^{INP} = {a,b,c,d,e,g,m}) ### Call 3

$V^{VRTX} \leftarrow$ {1,2}
$V^{LOW} \leftarrow$ {1}
$V^{HIGH} \leftarrow$ {2,(3,5,7),(4,6,8)}
$E^{LOW} \leftarrow$ {}
$E^{HIGH} \leftarrow$ {d,e,g,m}

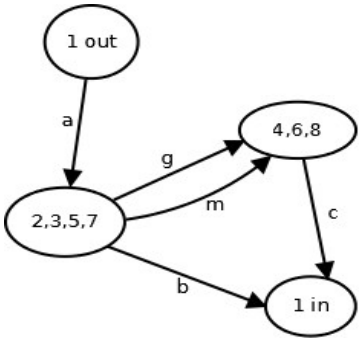
$V^{SCC} \leftarrow$ {(2,3,5,7),(4,6,8)}
$E^{SCC} \leftarrow$ {g,m}
$E^{INTRA} \leftarrow$ {d,e,f,h,i,j,k,l}
$V^{NEW} \leftarrow$ {1,(2,3,5,7),(4,6,8)}
$E^{NEW} \leftarrow$ {a,b,c,g,m}

IF(V^{LOW} == 1) **? YES**

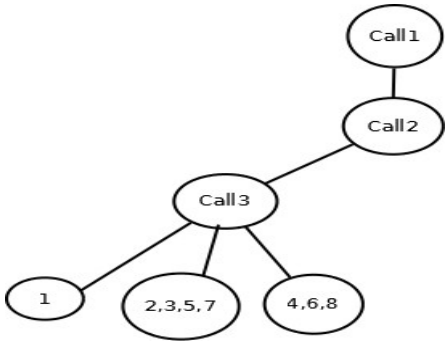
FORALL C in V^{SCC}
Place as the parent of the component in the execution tree V^{INP}
END_FOR
split(V^{INP}) based on vertex of $V^{LOW} = 1$

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
PreProcess({2,3,5,7},{d,e,f,h,k}) ### Call 6
END_FOR

Split graph N*o1:*



Execution tree:



Σχήμα -Step3

PreProcess(V^{INP} = {3,5,7}, E^{INP} = {f,h,k}) ### Call 4

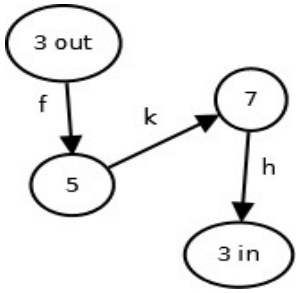
$V^{VRTX} \leftarrow$ {3,5,7}
$V^{LOW} \leftarrow$ {3}
$V^{HIGH} \leftarrow$ {5,7}
$E^{LOW} \leftarrow$ {}
$E^{HIGH} \leftarrow$ {k}

$V^{SCC} \leftarrow$ {(5),(7)}
$E^{SCC} \leftarrow$ {k}
$E^{INTRA} \leftarrow$ {}
$V^{NEW} \leftarrow$ {3,(5),(7)}
$E^{NEW} \leftarrow$ {f,h,k}

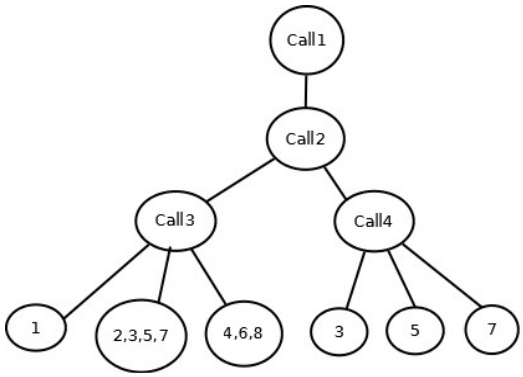
IF(V^{LOW} == 1) **? YES**
FORALL C in V^{SCC}
Place as the parent of the component in the execution tree V^{INP}
END_FOR
split(V^{INP}) based on vertex of V^{LOW} = 3

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
No Calls
END_FOR

Split graph N*o2:*



Execution tree:



Σχήμα -Step4

PreProcess(V^{INP} = {4,6,8}, E^{INP}= {i,j,l}) ### Call 5

$V^{VRTX} \leftarrow$ {4,6,8}
$V^{LOW} \leftarrow$ {4}
$V^{HIGH} \leftarrow$ {6,8}
$E^{LOW} \leftarrow$ {}
$E^{HIGH} \leftarrow$ {l}

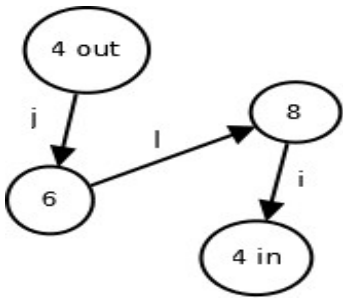
$V^{SCC} \leftarrow$ {(6),(8)}
$E^{SCC} \leftarrow$ {l}
$E^{INTRA} \leftarrow$ {}
$V^{NEW} \leftarrow$ {4,(6),(8)}
$E^{NEW} \leftarrow$ { j,i,l}

IF(V^{LOW} == 1) **? YES**

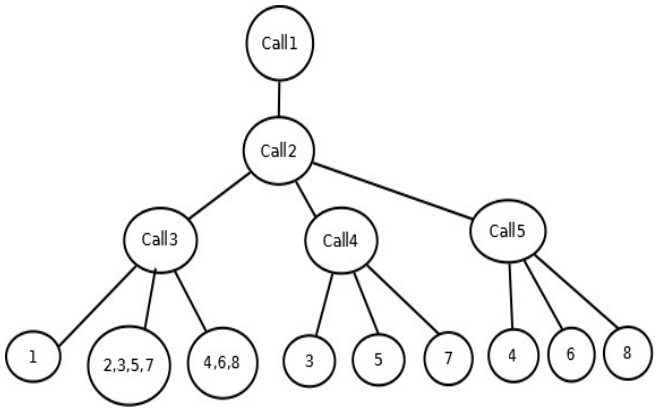
FORALL_COMPONENTS in V^{SCC}
Place as the parent of the component in the execution tree V^{INP}
END_FOR
split(V^{INP}) based on vertex of $V^{LOW} = 4$

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
No Calls
END_FOR

Split graph N*o3:*



Execution tree:



Σχήμα -Step5

PreProcess(V^{INP} = {2,3,5,7}, E^{INP} = {d,e,f,h,k}) ### Call 6

$V^{VRTX} \leftarrow$ {2,3,5,7}
$V^{LOW} \leftarrow$ {2,3}
$V^{HIGH} \leftarrow$ {5,7}
$E^{LOW} \leftarrow$ {}
$E^{HIGH} \leftarrow$ {k}

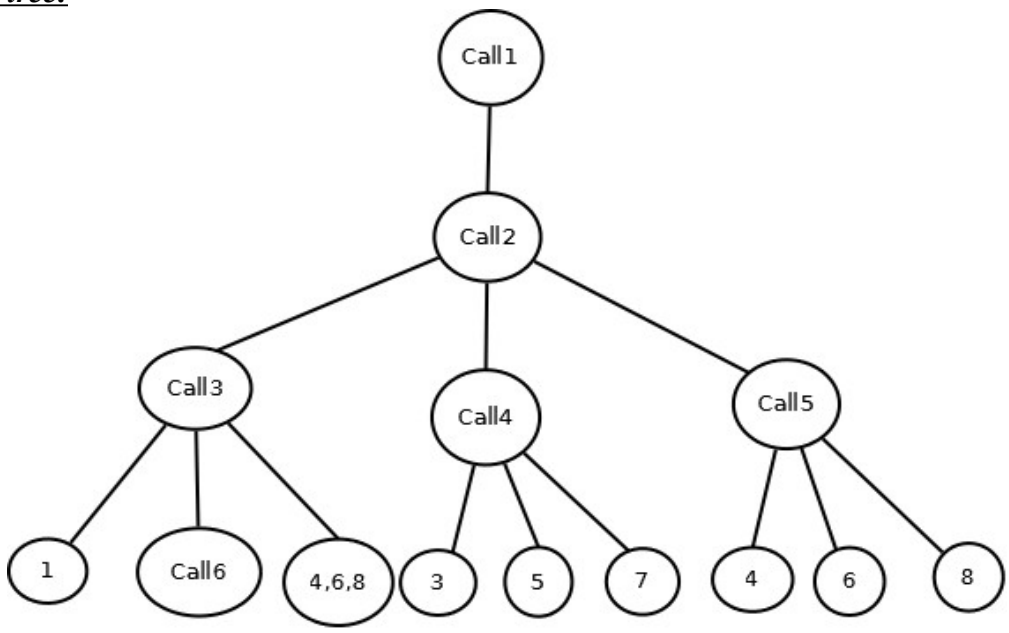
$V^{SCC} \leftarrow$ {(5),(7)}
$E^{SCC} \leftarrow$ {k}
$E^{INTRA} \leftarrow$ {}
$V^{NEW} \leftarrow$ {2,3,(5),(7)}
$E^{NEW} \leftarrow$ {d,e,f,h,k}

IF(V^{LOW} == 1) **? NO**

ELSE
    PreProcess(V^{NEW}, E^{NEW}) ### Call 7
END_IF

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
  No Calls
END_FOR

Execution tree:



Σχήμα -Step6

PreProcess(V^{INP} = {2,3,(5),(7)}, E^{INP} = {d,e,f,h,k}) ### Call 7

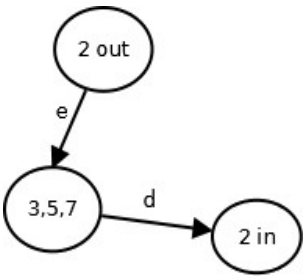
$V^{VRTX} \leftarrow$ {2,3}
$V^{LOW} \leftarrow$ {2}
$V^{HIGH} \leftarrow$ {3,(5),(7)}
$E^{LOW} \leftarrow$ {}
$E^{HIGH} \leftarrow$ {f,h,k}

$V^{SCC} \leftarrow$ {(5),(7)}
$E^{SCC} \leftarrow$ {f,h,k}
$E^{INTRA} \leftarrow$ {}
$V^{NEW} \leftarrow$ {2,3,(5),(7)}
$E^{NEW} \leftarrow$ {d,e,f,h,k}

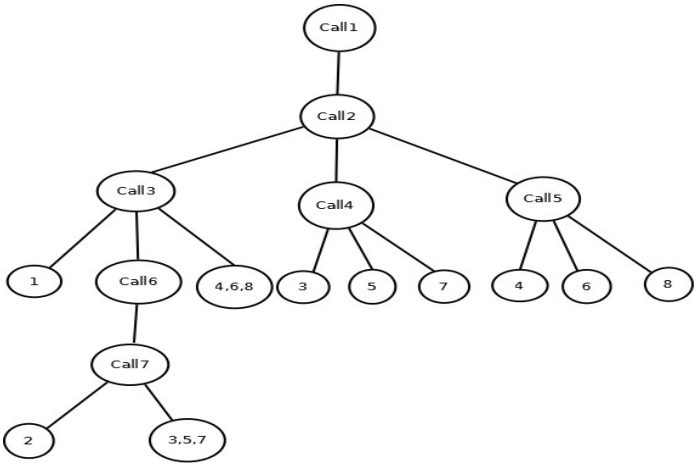
IF(V^{LOW} == 1) **? YES**
FORALL C in V^{SCC}
Place as the parent of the component in the execution tree V^{INP}
END_FOR
split(V^{INP}) based on vertex of V^{LOW} = 2

FORALL_COMPONENTS(V^{SCC}) ### that is formed for first time and does not exist in V^{INP}
No Calls
END_FOR

Split graph N*o4:*



Execution tree:



Σχήμα -Step7

Το παραπάνω παράδειγμα, που φτάνει εώς και το Σχήμα-Step7 είναι με βάση τον αλγόριθμο 2.2. Το τρέιξιμο του έχει ως εξής:

Step1 εώς Step3-Step4-Step5: Αυτό θεωρείται ένα βήμα, καθώς στη διαδικασία μέχρι εκείνη τη στιγμή το μόνο που κάνει ο αλγόριθμος είναι να ομαδοποιεί τα δεδομένα(κορυφές) ώστε να είναι ομαδοποιημένα κατάλληλα για την δημιουργία διασπασμένων γραφημάτων.

Step3-Step4-Step5: Δημιουργούνται 3 Split Graphs σε βάθος δέντρου βαθμού = 1.

Step6-Step7: Ομοίως με το Step1 εώς Step3-Step4-Step5.

Step7: Δημιουργείται το τελευταίο Split Graph No4. Ο αλγόριθμος δεν συνεχίζει στην ΙΣΣ (3,5,7) διότι το έχει ήδη αναλυθεί αυτή σε υψηλότερο επίπεδο του δέντρου.

Με το τέλος του αλγορίθμου δημιουργήθηκαν τέσσερα διασπασμένα γραφήματα, έτοιμα για επεξεργασία για την διαδικασία διαγραφής ακμών. Το βάθος του δέντρου βασίζεται στο πόσα επίπεδα παράγουν Split Graphs. Αυτό γίνεται σε δύο επίπεδα, πράγμα που σημαίνει ότι το βάθος του δέντρου εκτέλεσης είναι h = 3. Αυτό συμβαίνει διότι τα Step1 εώς Step3-Step4-Step5 και Step6-Step7 συμπτύσονται αντίστοιχα. Όντως βλέπουμε ότι κάθε ΙΣΣ αναλύθηκε ως προς το διασπασμένο γράφημα της και ο,τι αυτά που παράχθηκαν είναι ίδια ακριβώς με αυτά του Σχήματος 2.2, επαληθεύοντας την αυθεντικότητα του αλγορίθμου.

2.3 Σύγκριση αλγορίθμων Lacki-Roditty

Αρχικά να τονίσουμε ότι τα δέντρα εκτέλεσης(ιεραρχίες) της προεπεξεργασίας κατευθυνόμενων γραφημάτων είναι δέντρα που διατρέχονται κατά βάθος, δηλαδή κάθε επίπεδο διατρέχεται σε χρονική μιά χρονική στιγμή O(1). Επίσης παρατηρούμε ότι σε κάθε επίπεδο διατρέχονται όλες οι ακμές για επεξεργασία και δημιουργία των διασπασμένων γραφημάτων. Εφόσον οι ακμές είναι m, τότε διατρέχεται ένα επίπεδο σε χρόνο O(m), καθώς κάθε επίπεδο έχει το πολύ m ακμές, όπως φαίνεται και από τα παραδείγματα.

Από την άλλη υπάρχει το θέμα του βάθους του δέντρου. Ο Lacki με τον γενικό του αλγόριθμο, αν και σωστός, θεωρείται αργός. Αυτό συμβαίνει διότι εξετάζει μία προς μία τις εμφωλευμένες ΙΣΣ, σε περίπτωση που βρίσκονται σε τέτοια μορφή. Η ΙΣΣ (3,5,7) εξετάσθηκε στο Σχήμα 2.2 στο επίπεδο μετά την ΙΣΣ (2,3,5,7) ως εμφωλευμένη, ενώ αντίθετα στο ΠΑΡΑΔΕΙΓΜΑ 1 εξετάσθηκε ένα επίπεδο πιο πάνω. Με λίγα λόγια, στην χειρότερη περίπτωση, που μετά την αφαίρεση διάσπαση μίας κορυφής υπάρχει ΙΣΣ με όλες τις υπόλοιπες κορυφές, το βάθος του δέντρου θα φτάσει σε τέτοιο επίπεδο ώστε να μην υπάρχουν άλλες κορυφές προς διάσπαση δηλαδή σε επίπεδο h = n, όπου n ο αριθμός των κορυφών του γραφήαμτος εισόδου και το γράφημα θα διαπερνάται σε χρόνο O(mh) = O(mn). Ο Roditty με τον αλγόριθμο 2.2 καταφέρνει να μειώνει αυτόν τον χρόνο σε $O(mn^{1-\varepsilon})$ όπου ε > 0, ή αλλιώς O(mlogn), καθώς το επίπεδο βάθους του δέντρου θα είναι πάντα μικρότερο του n και ταυτόχρονα αποφεύγοντας την χειρότερη περίπτωση του Lacki.

Στο επόμενο κεφάλαιο προσομοιώνουμε τους αλγόριθμους και παρουσιάζουμε τα αποτελέσματα σε τρεξίματα τους.

3. Πειραματική αξιολόγηση

3.1 Προσομοίωση σε περιβάλλον LEDA

Για την επαλήθευση των συμπερασμάτων του προηγούμενου κεφαλαίου επιχειρήσαμε να προσομοιώσουμε τους δύο αλγόριθμους σε μία προγραμματιστική γλώσσα. Γι' αυτό το λόγο η C++ η οποία σε συνδυασμό με την βιβλιοθήκη LEDA, πληρεί τις προϋποθέσεις για την δημιουργία γραφημάτων και την επεξεργασία αυτών.

3.1.1 LEDA

Η LEDA παρέχει μία μεγάλου εύρους γκάμα επιλογών από δομές δεδομένων και αλγόριθμους σε μορφή η οποία είναι φιλική προς τον απλό χρήστη, για εύκολη χρησιμοποίηση τους. Περιέχει σχεδόν όλους τους γνωστούς αλγόριθμους και δομές δεδομένων στον χώρο αυτό, και συνεχίζει συνεχώς να εμπλουτίζεται. Μαζί με αυτά δίνει και την ακριβή περγραφή των τύπων που μπρούν να χρησιμοποιηθούν για τους σκοπούς του χρήστη.

Πιο συγκεκριμένα περιέχει μία έυκολα χρησιμοποιήσιμη δομή δεδομένων για γραφήματα. Προσφέρει τους κλασσικούς βρόχους που βοηθούν στην διαπέραση γραφημάτων όπως, π.χ. σε όλους τους κόμβους ή τις ακμές, επιτρέποντας τις διαγραφές ή προσθήκες σε αυτά, αναλόγως με την θέληση του χρήστη. Η δημιουργία γραφημάτων είτε από επιλογή είτε τυχαία, η προσθήκη/αφαίρεση κόμβων/ακμών, η διαπέραση εισερχόμενων/εξερχόμενων ακμών ή κόμβων, η προσθήκη βάρους μονοπατιού, η εύρεση συνιστωσών ή ισχυρά συνεκτικών συνιστωσών και η απεικόνιση γραφημάτων είναι μερικές από τις πάρα πολλές επιλογές που προσφέρει η LEDA, αξιόπιστα και εύκολα.

3.1.2 Προσομοίωση

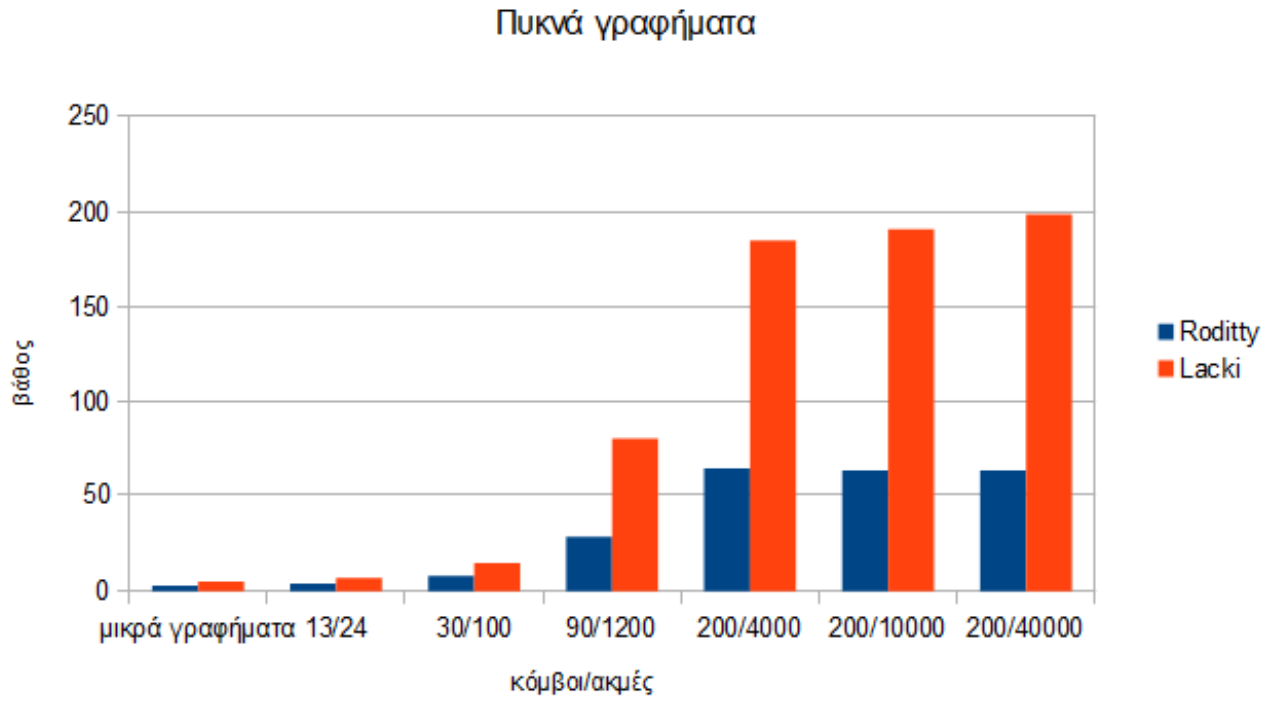
Η προσομοίωση των αλγορίθμων έγινε με βάση τα παραδείγματα του προηγούμενου κεφαλαίου, σε όσο πιο απλή μορφή γινόταν, με σκοπό την προεπεξεργασία ΙΣΣ διάφορων γραφημάτων. Το παράρτημα Β περιέχει τον κώδικα που υλοποιήθηκε, χωρίς την προσθήκη δημιουργίας διασπασμένων γραφημάτων, καθώς η δημιουργία τους είναι βήμα της επεξεργασίας και όχι της προ-επεξεργασίας κατευθυνόμενων γραφημάτων.

Πιο συγκεκριμένα χρησιμοποιήθηκε η συνάρτηση της LEDA *random_graph()* η οποία παράγει τυχαία κατευθυνόμενα γραφήματα, αναλόγως με τα μεγέθη που δίνονται ως είσοδοι, δηλαδή αριθμό κόμβων και ακμών. Έπειτα, από κάθε γράφημα, παίρνουμε τις ΙΣΣ, με την συνάρτηση της LEDA *STRONG_COMPONENTS()*, ως καινούρια μικρότερα γραφήματα, και τις χρησιμοποιούσαμε ως εισόδους στους αλγορίθμους μας, γιατί τέτοιες περιπτώσεις έπρεπε να εξετάσουμε. Από αυτά τα τρεξίματα τα οποία έγιναν πολλές φορές στα ίδια , αλλά και σε διάφορα μεγέθη κρατήθηκαν ανα ζευγάρια, οι τιμές του επίπεδου βάθους των 2 αλγορίθμων. Οι τιμές που αποτυπώνονται στα γραφήματα είναι οι μέσοι όροι και σαν αποτελέσματα σε βάθος αλλά και σαν μέσοι όροι από τα μεγέθη των γραφημάτων, δηλαδή περίπου πόσοι κόμβοι και πόσες ακμές.

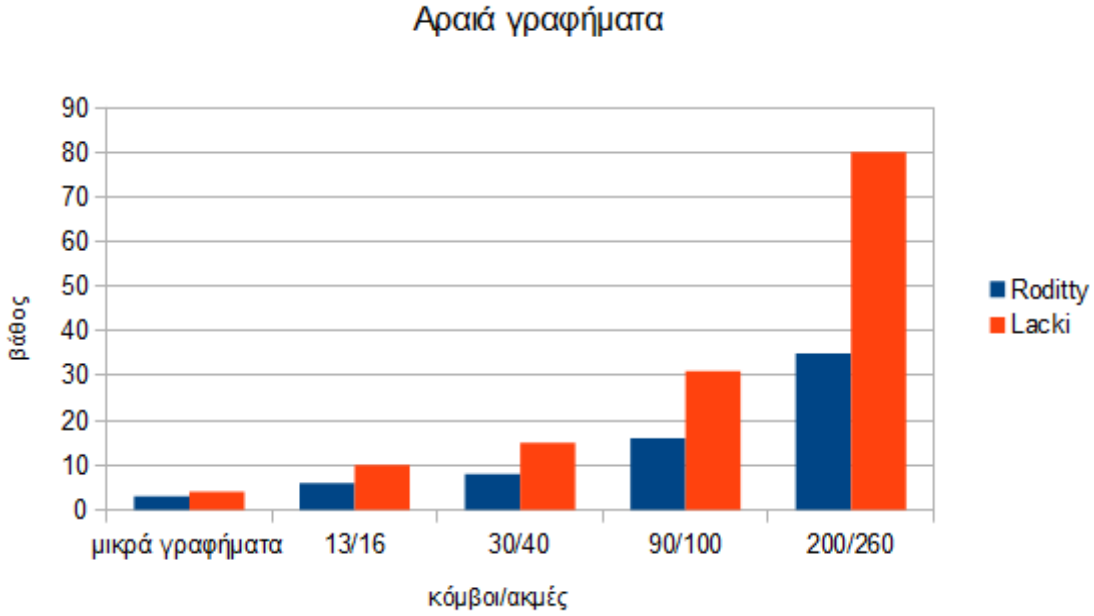
3.2 Αξιολόγηση

Η αξιολόγηση των συμπερασμάτων του προηγούμενου κεφαλαίο αφορά τους χρόνους εκτέλεσής τους που ήταν O(mn) και O(mlogn) για τους αλγορίθμους 2.1 και 2.2 αντίστοιχα. Αυτοί οι χρόνοι, όπως δείξαμε, σχετίζονται άμεσα με το βάθος του δέντρου καθώς αυτά τα δέντρα εκτέλεσης(ή ιεραρχίες) τα οποία είναι δέντρα που διαπερνούνται κατά πλάτος. Επομένως, αυτό που χρειαζόνταν να μετρήσουμε είναι το

βάθος των δέντρων που δημιουργούνται από τους δύο αυτούς αλγόριθμους. Οι μετρήσεις έγιναν με συνεχή δημιουργία διαφόρων γραφημάτων που ποικιλούσαν σε μέγεθος, και για κάθε κατηγορία που υπάρχει στα γραφήματα βγήκε ένας αντιπροσωπευτικός μέσος όρος, π.χ. στην κατηγορία 90/1200 τα βάθη των 2 αλγορίθμων ήταν ανάμεσα στις τιμές 27-35 και 77-82 αντίστοιχα. Ο μέσος όρος των τιμών ήταν 30-80 για Roditty – Lacki αντίστοιχα.



ΣΧΗΜΑ3.1. *Στο σχήμα απεικονίζονται οι μέσες τιμές από το βάθος δημιουργίας μίας ιεραρχίας από τον κάθε αλγόριθμο, σε πυκνά γραφήματα. Οι μπλε στήλες είναι με βάση τον ψευδοκώδικα του Roditty, ενώ η πορτοκαλί του Lacki. Π.χ. Στο 30-100 σημαίνει ότι σε γραφήματα που περιέχουν (σχεδόν) 30 κόμβους και (σχεδόν)100 ακμές, το βάθος του 1ου είναι περίπου το μισό από το βάθος του 2ου.*



ΣΧΗΜΑ3.2. *Ομοίως με το Σχήμα 3.1, αλλά για αραιά γραφήματα*

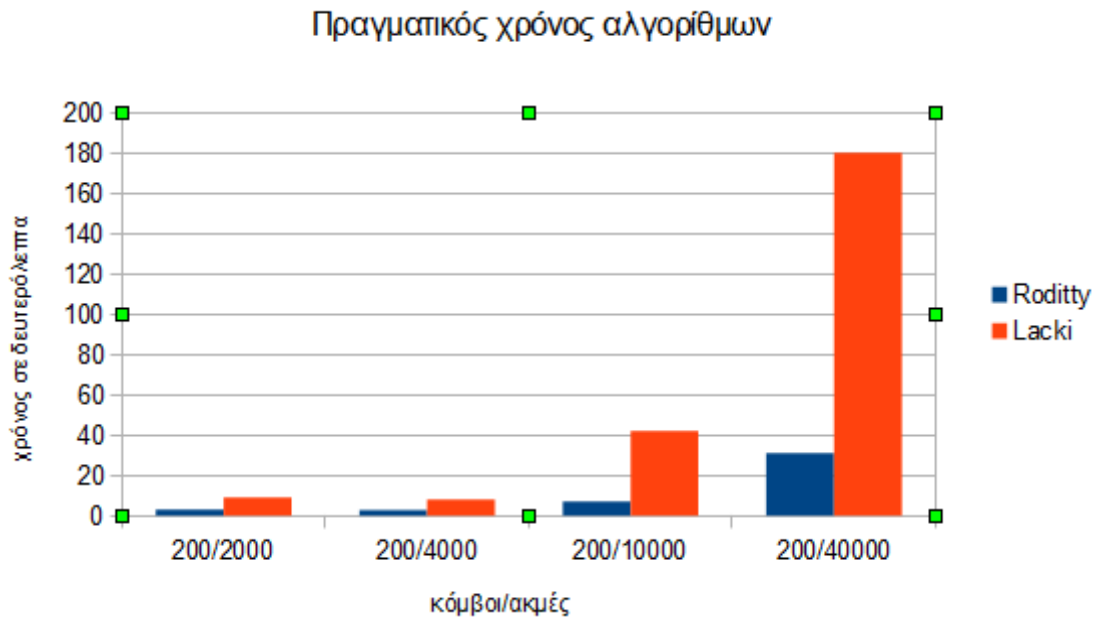
Τα αποτελέσματα έδειξαν ότι ο αλγόριθμος 2.2 δημιουργεί δέντρα χαμηλότερου βάθους από αυτόν του 2.1. Τα κατευθυνόμενα γραφήματα που δοκιμάστηκαν ποικιλούσαν ως προς το κατά πόσο αραιά ή πυκνά ήταν. Από αυτά τα αποτελέσματα των τρεξιμάτων παρατηρήθηκαν τα εξής:

1. *Ο αλγόριθμος του Roditty ποτέ δεν φτιάχνει δέντρα μεγαλύτερου βάθους.*

2. *Όσο πιο πυκνό και πιο μεγάλου μεγέθους είναι ένα γράφημα που αποτελεί μία ΙΣΣ, τόσο μεγαλώνει και η διαφορά ανάμεσα στο βάθος των δύο συγκρινόμενων ιεραρχιών, καθιστώντας αποτελεσματικότερο με εμφανή διαφορά τον αλγόριθμο του Roditty.Αντίστοιχα όσο πιο αραιό είναι, το μέγεθος διαφοράς τους ήταν αισθητό, αλλά όχι τόσο όσο με πυκνά γραφήματα.*

3. *Θεωρώντας ότι ένα γράφημα μπορεί να γίνει πλήρες, δηλαδή την μέγιστη μορφή πυκνού γραφήματος, η διαφορά βάθους ανάμεσα στους δύο αλγορίθμους έφτανε σε ένα μέγιστο. Όταν ένα γράφημα ήταν πλήρες, δηλαδή κοντά στην χειρότερη περίπτωση, ο αλγόριθμος του Roditty έφτιαχνε δέντρα με ελάχιστο βάθος το 30% από αυτά τα δέντρα που έφτιαχνε ο αλγόριθμος του Lacki.*

Από τα παραπάνω συμπεραίνουμε ότι ο αλγόριθμος του Roditty, είναι σαφέστατα αποδοτικότερος από αυτόν του Lacki. Επίσης παραθέτουμε στο Σχήμα 3.3 τα αποτελέσματα του πραγματικού χρόνου δημιουργίας των ιεραρχιών από την προσομοίωσή μας όπου η προσέγγιση του αλγορίθμου 2.2 γινόταν περίπου 6 φορές πιο γρήγορα από αυτήν του αλγορίθμου 2.1.



ΣΧΗΜΑ3.3. *Δείγματα πραγματικού χρόνου στην κατηγορία γραφημάτων 200 κόμβων, και δημιουργία γραφημάτων από αραιό σταδιακά σε πυκνό.*

4. Συμπεράσματα

Στη συγκεκριμένη εργασία παρουσιάσαμε το θέμα διατήρησης συνεκτικότητας σε γραφήματα, και πιο συγκεκριμένα την διατήρηση των ΙΣΣ συνεκτικών συνιστωσών σε κατευθυνόμενα γραφήματα, που βρίσκονται σε επεξεργασία για διαγραφή ακμών. Αφού αναφέραμε τις αρχικές προτάσεις πάνω στο θέμα φτάσαμε στις πιο πρόσφατες, και επικρατέστερες, των Lacki και Roditty. Αρχικά αναλύσαμε τους αλγόριθμους, στους οποίους ο δεύτερος υποστήριζε ότι ο πρώτος είναι αποτελεσματικός, αλλά όχι αποδοτικός, και παρουσίασε έναν δικό του αλγόριθμο. Με το τέλος της ανάλυσής μας φάνηκε ότι όντως ο Roditty βάσιζόταν σε επιχειρήματα που είναι ορθά, και ο αλγόριθμος 2.2 υπερίσχυε του αλγορίθμου 2.1. Κατόπιν, στο 3ο Κεφάλαιο, προσομοιώσαμε τους αλγορίθμους σε ένα κατάλληλο, για τους σκοπούς μας, περιβάλλον και αποδείξαμε ότι όντως ο αλγόριθμος προεπεξεργασίας του Roditty είναι τόσο αποτελεσματικός όσο και αποδοτικός.

4.1 Μελλοντικές προτάσεις

- Ανάλυση της διαγραφής ακμών με τον τρόπο του Lacki[2], δηλαδή της κύριας επεξεργασίας
- Ανάπτυξη των αλγορίθμων σε περιβάλλον LEDA με περισσότερη ακρίβεια, για πειράματα σε πραγματικό χρόνο σε ολόκληρη την βελτιστοποίηση των γραφημάτων, μαζί με την διαγραφή ακμων, καθώς και δημιουργία γραφικής εφαρμογής για τρέξιμο των αλγορίθμων

Παράρτημα Α

Α. Ορολογίες

ΙΣΣ: σημαίνει ισχυρά συνεκτική συνιστώσα. Όταν σε μία συνεκτική συνιστώσα υπάρχει κατευθυνόμενο μονοπάτι από κάθε κορυφή προς κάθε άλλη της ίδιας συνιστώσας, τότε αυτή λέγεται ΙΣΣ.

Πλήρες γράφημα: όταν σε ένα κατευθυνόμενο γράφημα, κάθε κορυφή ενώνεται με κάθε άλλη κορυφή με μία ακμή, όπου αυτή η ακμή προσπίπτει προς την άλλη κορυφή, τότε αυτό είναι ένα πλήρες γράφημα.

V: σύνολο κορυφών

E: ούνολο ακμών

n: κόμβοι

m: ακμές

O(mn): ο χρόνος εκτέλεσης είναι τάξεως m επί n

Αραιό/πυκνό γράφημα: όταν οι ακμές ενός γραφήματος είναι σχεδόν το τετράγωνο του αριθμου των κόμβων τότε αυτό λέγεται πυκνό, αλλιώς λέγεται αραιό

Παράρτημα Β

Β. Κώδικες προσομοίωσης

Οι δύο συναρτήσεις σε γλώσσα C++, με την χρήση προτύπων(template) από την βιβλιοθήκη LEDA. Με τον αριθμό σε κάθε σχόλιο αντιστοιχείται και μία εντολή από τον ψευδοκώδικα.

```
 /*                    ALGORITHM STARTS   -----> Roditty
//*************************************************************************
//*************************************************************************/


void PreProcess(GRAPH<list<int>,char> G_pre,int depth)
{
```

```
GRAPH<list<int>,char> G_VRTX, G_LOW, G_HIGH, G_NEW, G_INP, G_SCC, G_INTRA;
G_INP = G_pre;
node v, v_lowest, v_temp, v_temp2, v_temp3, v_temp4;
edge e, e_temp;
int counter = 1;
int i_cnt,k_help;
list_item it, it2, it3, it4;
int poil=depth;

/* 1
 V_VRTX <-- {v | v belongs to V_INP && id(v)!= infinite }
                                        */

forall_nodes(v,G_INP){
  if(!G_INP[v].search(-1)){
    G_VRTX.new_node(G_INP[v]);
  }else{
    G_HIGH.new_node(G_INP[v]);     //this is associated with instruction No3
  }
}

/* 2
  V_LOW <-- {v | v belongs to V_VRTX && v's id is among |V_VRTX|/2 smallest ids}
                                                                        */

 int num_low = G_VRTX.number_of_nodes()/2;

while(counter <= num_low){

  v_lowest = G_VRTX.choose_node();

  forall_nodes(v,G_VRTX){

     if(G_VRTX[v].head() < G_VRTX[v_lowest].head())
           v_lowest = v;

  }

  G_LOW.new_node(G_VRTX[v_lowest]);
  G_VRTX.del_node(v_lowest);
  counter++;

}

/* 3
  V_HIGH <-- V_INP \ V_LOW
                                */

forall_nodes(v,G_VRTX)
   G_HIGH.new_node(G_VRTX[v]);

/* 4
  E_LOW <-- {(x,y) | (x,y) belongs to E_INP && x belongs to V_LOW && y belongs to V_LOW}
                                                                                */
```

```
forall_edges(e_temp,G_INP){
  forall_nodes(v,G_LOW){
    if(G_INP[G_INP.source(e_temp)].head() == G_LOW[v].head()){
          forall_nodes(v_temp,G_LOW){
            if(G_INP[G_INP.target(e_temp)].head() == G_LOW[v_temp].head()){
              G_LOW.new_edge(v,v_temp, G_INP[e_temp]);
            }
          }
    }
  }
}


/* 5
  E_HIGH <-- {(x,y) | (x,y) belongs to E_INP && x belongs to V_HIGH && y belongs to V_HIGH}
                                                                                          */

forall_edges(e_temp,G_INP){
  forall_nodes(v,G_HIGH){
    if(G_INP[G_INP.source(e_temp)].head() == G_HIGH[v].head()){
          forall_nodes(v_temp,G_HIGH){
            if(G_INP[G_INP.target(e_temp)].head() == G_HIGH[v_temp].head()){
              G_HIGH.new_edge(v,v_temp, G_INP[e_temp]);
            }
          }
    }
  }
}



/* 6
  (V_SCC, E_SCC, E_INTRA) <-- SCC(V_HIGH, E_HIGH)
                                                  */


node_array<int> compnum(G_HIGH);
int num_sccs = STRONG_COMPONENTS(G_HIGH,compnum);
array<list<int> > component(num_sccs);

// cout << endl << " num_scc:"<< num_sccs<<" " <<endl;

forall_nodes(v,G_HIGH){
  for(it = G_HIGH[v].first(); it != nil; it = G_HIGH[v].succ(it)){
    if(!(component[compnum[v]].search(-1) && G_HIGH[v][it] == -1 ))
          component[compnum[v]].append(G_INP[v][it]);
  }
}


for(i_cnt = 0; i_cnt<num_sccs;i_cnt++)
  G_SCC.new_node(component[i_cnt]);
/* P.S.
    We computed V_SCC, but not E_SCC or E_INTRA because they aren't necessary for the time being.
    When needed, they will be included.
                                                                                          */
```

```
 forall_edges(e,G_INP){
  forall_nodes(v,G_SCC){
         if(G_SCC[v].search(G_INP[G_INP.source(e)].head())){
          forall_nodes(v_temp3,G_SCC){
             if(G_SCC[v_temp3].search(G_INP[G_INP.target(e)].head())){
                 if(v_temp3 != v){
                   G_SCC.new_edge(v,v_temp3,G_INP[e]);
                 }
             }
           }
          }
  }
 }


 /* 7
         forall(C which belong to V_SCC)do
          id(C) <-- infinite;
         end for
                                                    */

 forall_nodes(v,G_SCC)
   if(!G_SCC[v].search(-1))
     G_SCC[v].append(-1);  //Instead of infinite we add the value "-1" so we can identify a component 'C'



 /* 8
    V_NEW <-- union(V_LOW, V_SCC)
                                    */


 forall_nodes(v,G_LOW)
  G_NEW.new_node(G_LOW[v]);


 forall_nodes(v,G_SCC)
  G_NEW.new_node(G_SCC[v]);



/* 9
   E_NEW <-- {(x,C) | x belongs to V_LOW && C belongs to V_SCC && y belongs to C s.t. (x,y) belongs
to E_INP}

*/

 forall_edges(e,G_INP){
  forall_nodes(v,G_LOW){
    if(G_INP[G_INP.source(e)] == G_LOW[v]){
         forall_nodes(v_temp,G_SCC){
          forall_items(it,G_SCC[v_temp]){
           if(G_SCC[v_temp][it] != -1 && G_SCC[v_temp][it] == G_INP[G_INP.target(e)].head() ){
            forall_nodes(v_temp3,G_NEW){
                 if(G_NEW[v_temp3] == G_LOW[v]){
```

```
                    forall_nodes(v_temp4,G_NEW){
                     if(G_SCC[v_temp].length() == G_NEW[v_temp4].length() ){
                          for(it = G_SCC[v_temp].first(),it2=G_NEW[v_temp4].first(); it != nil,it2 != nil; it =
G_SCC[v_temp].succ(it),G_NEW[v_temp4].succ(it2)){
                           if(G_SCC[v_temp][it] != G_NEW[v_temp4][it2] ){
                            break;
                           }
                           G_NEW.new_edge(v_temp3,v_temp4,G_INP[e]);
                           }
                          }
                    }
                  }
              }
             }
            }
           }
     }
    }
  }


  /* 10
    E_NEW <-- union(E_NEW, {(C,x) | x belongs to V_LOW && C belongs to V_SCC && y belongs to C
s.t. (y,x) belongs to E_INP})

*/

   forall_edges(e,G_INP){
    forall_nodes(v,G_LOW){
     if(G_INP[G_INP.target(e)] == G_LOW[v]){
          forall_nodes(v_temp,G_SCC){
           forall_items(it,G_SCC[v_temp]){
            if(G_SCC[v_temp][it] != -1 && G_SCC[v_temp][it] == G_INP[G_INP.source(e)].head() ){
              forall_nodes(v_temp3,G_NEW){
                    if(G_NEW[v_temp3] == G_LOW[v]){
                     forall_nodes(v_temp4,G_NEW){
                      if(G_SCC[v_temp].length() == G_NEW[v_temp4].length() ){
                          for(it = G_SCC[v_temp].first(),it2=G_NEW[v_temp4].first(); it != nil,it2 != nil; it =
G_SCC[v_temp].succ(it),G_NEW[v_temp4].succ(it2)){
                           if(G_SCC[v_temp][it] != G_NEW[v_temp4][it2] ){
                            break;
                           }
                           G_NEW.new_edge(v_temp4,v_temp3,G_INP[e]);
                           }
                          }
                    }
                  }
              }
             }
            }
           }
     }
    }
  }
```

```
/* 11
     E_NEW <-- union(E_NEW, E_SCC, E_LOW)
                                                */


forall_edges(e,G_INP){
 forall_nodes(v,G_LOW){
  if(G_INP[G_INP.source(e)] == G_LOW[v]){
   forall_nodes(v_temp,G_LOW){
        if(G_INP[G_INP.target(e)] == G_LOW[v_temp]){
          forall_nodes(v_temp3,G_NEW){
            if(G_NEW[v_temp3] == G_LOW[v]){
              forall_nodes(v_temp4,G_NEW){
                   if(G_NEW[v_temp4] == G_LOW[v_temp]){
                     G_NEW.new_edge(v_temp3,v_temp4,G_INP[e]);
                   }
                }
             }
            }
           }
        }
   }
  }
 }
}



forall_edges(e,G_SCC){
  forall_nodes(v,G_NEW){
    if(G_SCC[G_SCC.source(e)].head() == G_NEW[v].head()){
          forall_nodes(v_temp,G_NEW){
            if(G_SCC[G_SCC.target(e)].head() == G_NEW[v_temp].head()){
              G_NEW.new_edge(v,v_temp,G_SCC[e]);
            }
          }
    }
  }
}

 //cout << G_SCC << endl;

if(num_low == 1){


box_depth[poil]++;

//cout << endl <<"The final graph" << endl;
//cout << endl << G_NEW << endl;



  /* 12
  if(|V_LOW| == 1) then
    forall(C which belong to V_SCC) do
          parent(C) <-- V_INP
```

```
    endfor
                    ************ 'if' continues to 'else' occasion in Instr.No16 and finishes at Instr.No17
                                    */
 /* 13
    split(V_INP) = v, where v belongs to V_LOW
                                                    */
 /* 14
        splitted_to_v(V_INP) <-- union(V_SCC, (v_in && v_out))
                                                                */
 /* 15
       edges_of_splitted_to_v_V_INP(E_INP) <-- union(E_SCC, {(v_out,C) | (v,C) belongs to E_NEW},
{(C,v_in) | (C,v) belongs to E_NEW})

*/



 }else{

    /* 16
   else
        PreProcess(V_NEW, E_NEW)
                                    */

  PreProcess(G_NEW,poil);

 }




    /* 17
     forall(C which belong to V_SCC && C does not belong to V_INP) do
        PreProcess(C,E_INTRA(C)), where E_INTRA(C) as sub-total of E_HIGH
        is the set of edges with both endpoints in C
     endfor
    endif ******** 'endif' is from Instr.No12
  */

  int HELP = 0;
   forall_nodes(v,G_SCC){
      if(G_SCC[v].length() > 2){/*Greater than 2 means that it does not have only one vertex and (-1)-->
identifier of scc**/

        forall_nodes(v_temp,G_INP){
              if(G_SCC[v].length()  ==  G_INP[v_temp].length()  &&  G_SCC[v].head()  ==
G_INP[v_temp].head()){
           //Do nothing-- this component should not be called
           HELP = 1;
           break;
         }
        }
        if(HELP == 0){

           GRAPH<list<int>,char> G_SMALL;
```

```
            for(it = G_SCC[v].first(); it != nil; it = G_SCC[v].succ(it)){
              if(G_SCC[v][it]!= -1){
                    //cout<< "Items " << G_SCC[v][it] << endl;
                    node v_extra = G_SMALL.new_node();
                    G_SMALL[v_extra].append(G_SCC[v][it]);
              }
            }
              forall_edges(e,G_INP){
                    forall_nodes(v,G_SMALL){
                      if(G_SMALL[v].search(G_INP[G_INP.source(e)].head())){
                        forall_nodes(v_temp3,G_SMALL){
                          if(G_SMALL[v_temp3].search(G_INP[G_INP.target(e)].head())){
                                if(v_temp3 != v){
                                  G_SMALL.new_edge(v,v_temp3,G_INP[e]);
                                }
                          }
                        }
                      }
                    }
              }

            //cout << G_SMALL << endl;

             /*********Exactly here we call the strongly connected component as a new tree, creating a new
path in the execution tree*****************/

            PreProcess(G_SMALL,++poil);
          }
          HELP = 0;
      }
    }



}









/*                      ALGORITHM STARTS   -----> Lacki
  //*************************************************************************
  //*************************************************************************/



void Lacki(GRAPH<list<int>,char> G_pre, int depth2){

  GRAPH<list<int>,char> G_VRTX1, G_LOW1, G_HIGH1, G_NEW1, G_INP1, G_SCC1, G_INTRA1;
  G_INP1 = G_pre;
  node v1, v_lowest1, v_temp1, v_temp21, v_temp31, v_temp41;
  edge e1, e_temp1;
```

```
int counter1 = 1;
int i_cnt1,k_help1;
list_item it1, it21, it31, it41;
int poil = depth2;
  /* 1
 V_VRTX <-- {v | v belongs to V_INP && id(v)!= infinite }
                                                  */
forall_nodes(v1,G_INP1){
  if(!G_INP1[v1].search(-1)){
    G_VRTX1.new_node(G_INP1[v1]);
  }else{
    G_HIGH1.new_node(G_INP1[v1]);     //this is associated with instruction No3
  }
}

/* 2
  V_LOW <-- {v | v belongs to V_VRTX && v's id is among |V_VRTX|/2 smallest ids}
                                                                                  */

 int num_low1 = 1;

while(counter1 <= num_low1){

  v_lowest1 = G_VRTX1.choose_node();

  forall_nodes(v1,G_VRTX1){

     if(G_VRTX1[v1].head() < G_VRTX1[v_lowest1].head())
           v_lowest1 = v1;

  }

  G_LOW1.new_node(G_VRTX1[v_lowest1]);
  G_VRTX1.del_node(v_lowest1);
  counter1++;

}

/* 3
  V_HIGH <-- V_INP \ V_LOW
                                  */

forall_nodes(v1,G_VRTX1)
   G_HIGH1.new_node(G_VRTX1[v1]);



/* 4
  E_LOW <-- {(x,y) | (x,y) belongs to E_INP && x belongs to V_LOW && y belongs to V_LOW}
                                                                                    */

forall_edges(e_temp1,G_INP1){
  forall_nodes(v1,G_LOW1){
    if(G_INP1[G_INP1.source(e_temp1)].head() == G_LOW1[v1].head()){
          forall_nodes(v_temp1,G_LOW1){
```

```
          if(G_INP1[G_INP1.target(e_temp1)].head() == G_LOW1[v_temp1].head()){
            G_LOW1.new_edge(v1,v_temp1, G_INP1[e_temp1]);
          }
         }
   }
  }
}


/* 5
  E_HIGH <-- {(x,y) | (x,y) belongs to E_INP && x belongs to V_HIGH && y belongs to V_HIGH}
                                                                                          */

forall_edges(e_temp1,G_INP1){
  forall_nodes(v1,G_HIGH1){
    if(G_INP1[G_INP1.source(e_temp1)].head() == G_HIGH1[v1].head()){
         forall_nodes(v_temp1,G_HIGH1){
           if(G_INP1[G_INP1.target(e_temp1)].head() == G_HIGH1[v_temp1].head()){
             G_HIGH1.new_edge(v1,v_temp1, G_INP1[e_temp1]);
           }
         }
    }
  }
}


  /* 6
  (V_SCC, E_SCC, E_INTRA) <-- SCC(V_HIGH, E_HIGH)
                                                  */


node_array<int> compnum1(G_HIGH1);
int num_sccs1 = STRONG_COMPONENTS(G_HIGH1,compnum1);
array<list<int> > component1(num_sccs1);

// cout << endl << " num_scc:"<< num_sccs<<" " <<endl;

forall_nodes(v1,G_HIGH1){
  for(it1 = G_HIGH1[v1].first(); it1 != nil; it1 = G_HIGH1[v1].succ(it1)){
    if(!(component1[compnum1[v1]].search(-1) && G_HIGH1[v1][it1] == -1 ))
         component1[compnum1[v1]].append(G_INP1[v1][it1]);
  }
}



for(i_cnt1 = 0; i_cnt1<num_sccs1;i_cnt1++)
  G_SCC1.new_node(component1[i_cnt1]);

/* P.S.
    We computed V_SCC, but not E_SCC or E_INTRA because they aren't necessary for the time being.
    When needed, they will be included.
                                                                                          */
forall_edges(e1,G_INP1){
  forall_nodes(v1,G_SCC1){
```

```
          if(G_SCC1[v1].search(G_INP1[G_INP1.source(e1)].head())){
            forall_nodes(v_temp31,G_SCC1){
               if(G_SCC1[v_temp31].search(G_INP1[G_INP1.target(e1)].head())){
                    if(v_temp31 != v1){
                      G_SCC1.new_edge(v1,v_temp31,G_INP1[e1]);
                    }
               }
             }
            }
   }
  }


  /* 7
          forall(C which belong to V_SCC)do
            id(C) <-- infinite;
          end for
                                                        */
  box_depth2[poil]++;
  forall_nodes(v1,G_SCC1)
    if(!G_SCC1[v1].search(-1))
      G_SCC1[v1].append(-1);  //Instead of infinite we add the value "-1" so we can identify a component 'C'



  /* 8
     V_NEW <-- union(V_LOW, V_SCC)
                                          */


  forall_nodes(v1,G_LOW1)
    G_NEW1.new_node(G_LOW1[v1]);


  forall_nodes(v1,G_SCC1)
    G_NEW1.new_node(G_SCC1[v1]);


  /* 9
    E_NEW <-- {(x,C) | x belongs to V_LOW && C belongs to V_SCC && y belongs to C s.t. (x,y) belongs
to E_INP}

*/






  forall_edges(e1,G_INP1){
    forall_nodes(v1,G_LOW1){
      if(G_INP1[G_INP1.source(e1)] == G_LOW1[v1]){
          forall_nodes(v_temp1,G_SCC1){
            forall_items(it1,G_SCC1[v_temp1]){
              if(G_SCC1[v_temp1][it1] != -1 && G_SCC1[v_temp1][it1] ==
```

```
 G_INP1[G_INP1.target(e1)].head() ){
              forall_nodes(v_temp31,G_NEW1){
                    if(G_NEW1[v_temp31] == G_LOW1[v1]){
                      forall_nodes(v_temp41,G_NEW1){
                        if(G_SCC1[v_temp1].length() == G_NEW1[v_temp41].length() ){
                             for(it1 = G_SCC1[v_temp1].first(),it21=G_NEW1[v_temp41].first(); it1 != nil,it21 !
= nil; it1 = G_SCC1[v_temp1].succ(it1),G_NEW1[v_temp41].succ(it21)){
                              if(G_SCC1[v_temp1][it1] != G_NEW1[v_temp41][it21] ){
                                break;
                              }
                              G_NEW1.new_edge(v_temp31,v_temp41,G_INP1[e1]);
                              }
                            }
                        }
                    }
                }
              }
            }
          }
        }
    }
   }
  }


  /* 10
    E_NEW <-- union(E_NEW, {(C,x) | x belongs to V_LOW && C belongs to V_SCC && y belongs to C
s.t. (y,x) belongs to E_INP})

*/
   forall_edges(e1,G_INP1){
    forall_nodes(v1,G_LOW1){
     if(G_INP1[G_INP1.target(e1)] == G_LOW1[v1]){
          forall_nodes(v_temp1,G_SCC1){
           forall_items(it1,G_SCC1[v_temp1]){
                              if(G_SCC1[v_temp1][it1]      !=      -1      &&      G_SCC1[v_temp1][it1]      ==
G_INP1[G_INP1.source(e1)].head() ){
              forall_nodes(v_temp31,G_NEW1){
                    if(G_NEW1[v_temp31] == G_LOW1[v1]){
                      forall_nodes(v_temp41,G_NEW1){
                        if(G_SCC1[v_temp1].length() == G_NEW1[v_temp41].length() ){
                             for(it1 = G_SCC1[v_temp1].first(),it21=G_NEW1[v_temp41].first(); it1 != nil,it21 !
= nil; it1 = G_SCC1[v_temp1].succ(it1),G_NEW1[v_temp41].succ(it21)){
                              if(G_SCC1[v_temp1][it1] != G_NEW1[v_temp41][it21] ){
                                break;
                              }
                              G_NEW1.new_edge(v_temp41,v_temp31,G_INP1[e1]);
                              }
                            }
                        }
                    }
                }
              }
            }
          }
        }
    }
```

```
  }
 }


 /* 11
      E_NEW <-- union(E_NEW, E_SCC, E_LOW)
                                                */


 forall_edges(e1,G_INP1){
  forall_nodes(v1,G_LOW1){
   if(G_INP1[G_INP1.source(e1)] == G_LOW1[v1]){
    forall_nodes(v_temp1,G_LOW1){
         if(G_INP1[G_INP1.target(e1)] == G_LOW1[v_temp1]){
          forall_nodes(v_temp31,G_NEW1){
           if(G_NEW1[v_temp31] == G_LOW1[v1]){
            forall_nodes(v_temp41,G_NEW1){
                 if(G_NEW1[v_temp41] == G_LOW1[v_temp1]){
                  G_NEW1.new_edge(v_temp31,v_temp41,G_INP1[e1]);
                 }
            }
           }
          }
         }
        }
    }
   }
  }
 }




 forall_edges(e1,G_SCC1){
  forall_nodes(v1,G_NEW1){
   if(G_SCC1[G_SCC1.source(e1)].head() == G_NEW1[v1].head()){
         forall_nodes(v_temp1,G_NEW1){
          if(G_SCC1[G_SCC1.target(e1)].head() == G_NEW1[v_temp1].head()){
           G_NEW1.new_edge(v1,v_temp1,G_SCC1[e1]);
          }
         }
    }
   }
  }

 //cout << "display :"<<G_SCC1 << endl;

  int HELP1 = 0;
   forall_nodes(v1,G_SCC1){
     if(G_SCC1[v1].length() > 2){/*Greater than 2 means that it does not have only one vertex and (-1)-->
identifier of scc**/
         forall_nodes(v_temp1,G_INP1){
              if(G_SCC1[v1].length() == G_INP1[v_temp1].length() && G_SCC1[v1].head() ==
G_INP1[v_temp1].head()){
           //Do nothing-- this component should not be called
           HELP1 = 1;
```

```
            break;
          }
        }
        if(HELP1 == 0){

          GRAPH<list<int>,char> G_SMALL1;
          for(it1 = G_SCC1[v1].first(); it1 != nil; it1 = G_SCC1[v1].succ(it1)){
           if(G_SCC1[v1][it1]!= -1){
                //cout<< "Items " << G_SCC[v][it] << endl;
                node v_extra1 = G_SMALL1.new_node();
                G_SMALL1[v_extra1].append(G_SCC1[v1][it1]);
           }
          }
           forall_edges(e1,G_INP1){
                forall_nodes(v1,G_SMALL1){
                 if(G_SMALL1[v1].search(G_INP1[G_INP1.source(e1)].head())){
                  forall_nodes(v_temp31,G_SMALL1){
                   if(G_SMALL1[v_temp31].search(G_INP1[G_INP1.target(e1)].head())){
                        if(v_temp31 != v1){
                          G_SMALL1.new_edge(v1,v_temp31,G_INP1[e1]);
                        }
                   }
                  }
                 }
                }
           }

          //cout << G_SMALL << endl;

           /*********Exactly here we call the strongly connected component as a new tree, creating a new
path in the execution tree****************/
          Lacki(G_SMALL1,poil);
        }
        HELP1 = 0;
     }
    }

}
```

Βιβλιογραφία

[1] L. Roditty. Decremental maintenance of strongly connected components. *In Proc. of 24th of SODA*, pages 1143-1150, 2013

[2] J. Lacki. Improved deterministc algorithms for decremental transitive closure and strongly connected components. *In Proc. of 22nd SODA*, pages 1438-1445, 2011

[3] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. *In Proc. of 40th FOCS*, pages 81-89, 1999

[4] D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithmics,* 6, 2001.

[5] P. Bozanis. Algorithms, published at Thessaloniki, 2005

[6] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J.Comput.*, 37(5):1455-1471, 2008

[7] http://www.algorithmic-solutions.info/leda_manual/manual.html

[8] http://www.leda-tutorial.org/en/unofficial/

[9] http://www.algorithmic-solutions.com/