Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Μελέτη αλγορίθμων εύρεσης ελαχίστων ζευγνυόντων δέντρων**
Ονοματεπώνυμο Φοιτητή	**Τσιτσιρίγκος Κωνσταντίνος**
Πατρώνυμο	**Αθανάσιος**
Αριθμός Μητρώου	**ΜΠΠΛ / 08058**
Επιβλέπων	**Ευάγγελος Φούντας / Καθηγητής**

Ημερομηνία Παράδοσης **Σεπτέμβριος 2010**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Ευάγγελος Φούντας Καθηγητής	**Τσικούρας Παναγιώτης-Γεώργιος Καθηγητής**	**Αποστόλου Δημήτριος Επίκουρος Καθηγητής**

Κωνσταντίνος Α.Τσιτσιρίγκος (ΜΠΠΛ 08058)

Περιεχόμενα

Περίληψη

Το πρόβλημα εύρεσης ελαχίστων ζευγνυόντων δέντρων αποτελεί ένα από τα πιο βασικά επιστημονικά προβλήματα με σημαντικές εφαρμογές στη σχεδίαση και βελτιστοποίηση δικτύων. Το εν λόγω πρόβλημα έχει αποτελέσει αντικείμενο έρευνας και μελέτης πολλών επιστημόνων με αποτέλεσμα την ανάπτυξη πολλών αποδοτικών αλγορίθμων πολυωνυμικού χρόνου. Στόχος της εν λόγω διπλωματικής εργασίας είναι η παρουσίαση των σημαντικότερων από αυτούς τους αλγορίθμους, δίνοντας γραφικά παραδείγματα για την επεξήγηση τους, αλλά και παρουσίαση ψευδοκώδικα για την εκτέλεση καθένα από αυτούς. Επίσης, γίνεται υλοποίηση των αλγορίθμων Kruskal, Dijkstra και Prim με χρήση της γλώσσας C αλλά και Pascal. Ακόμα αναπτύχθηκαν αλγόριθμοι χειρισμού γράφων στην γλώσσα προγραμματισμού C και εξετάστηκαν γράφοι με 4, 5 και 6 κόμβους.

Abstract

The problem of finding minimum spanning trees is one of the most fundamental scientific problems. It has important applications in networks to design and optimizes them. This problem has been the subject of investigation and study of many scientists leading to the development of many efficient polynomial time algorithms. The goal of this thesis is to present the most important of these algorithms. This will be achieved through graphic examples of the algorithms in order to explain them. Simultaneously it will be given the pseudocode needed to execute each one of them. Furthermore in this thesis the algorithms Kruskal, Dijkstra and Prim are presented by using both C and Pascal programming languages. Finally, it will be presented the algorithms that were developed for handling graphs in C language for graphs with 4, 5 and 6 nodes.

Εισαγωγή

Στην καθημερινότητα μας η παρουσία των δικτύων είναι εμφανής. Δίκτυα ηλεκτρικής ισχύος που τροφοδοτούν ολόκληρες πόλεις, τηλεφωνικά δίκτυα που κατέστησαν την επικοινωνία των ανθρώπων δεδομένη. Αλλά ακόμα και εθνικά οδικά δίκτυα, σιδηροδρομικά και αεροπορικά δίκτυα, δίκτυα που μας επιτρέπουν να μετακινούμαστε ή να μεταφέρουμε ήχο ή ενέργεια σε εύλογο χρόνο. Επιπλέον έχουμε τα δίκτυα των υπολογιστών τα οποία γνώρισαν μεγάλη ανάπτυξη τα τελευταία χρόνια, με αποκορύφωμα τη δημιουργία του παγκόσμιου δικτύου υπολογιστών του Internet, αλλάζοντας έτσι τον τρόπο με τον οποίο μοιραζόμαστε την πληροφορία και διευκολύνοντας τόσο την εργασία μας όσο και την καθημερινή μας ζωή.

Σε όλα τα παραπάνω δίκτυα, αλλά και σε πολλά περισσότερα, στόχος μας είναι η μετακίνηση κάποιας οντότητας, για παράδειγμα ενός οχήματος, ενός ατόμου ενός μηνύματος, μίας τηλεφωνικής κλήσης, της ηλεκτρικής ενέργειας από ένα σημείο του δικτύου σε κάποιο άλλο. Η μετακίνηση αυτή θα πρέπει να γίνει όσο το δυνατόν αποτελεσματικότερη χρησιμοποιώντας αποδοτικά τις υφιστάμενες διόδους μετακίνησης. Τα παραπάνω προβλήματα ανήκουν στην κατηγορία των προβλημάτων συνδυαστικής βελτιστοποίησης (combinatorial optimization problems). Συγκεκριμένα, σε ένα πρόβλημα βελτιστοποίησης επιθυμούμε να επιλέξουμε τη βέλτιστη λύση από ένα σύνολο εναλλακτικών λύσεων, δηλαδή εκείνη τη λύση που ικανοποιεί συγκεκριμένα κριτήρια ή περιορισμούς.

Στην περίπτωση που επιθυμείται η εύρεση της βέλτιστης λύσης ικανοποιώντας ένα μόνο κριτήριο, π.χ. θέλουμε να στείλουμε ένα πακέτο πληροφορίας μεταξύ δυο συγκεκριμένων σημείων ενός δικτύου όσο πιο γρήγορα γίνεται, τα πράγματα είναι σχετικά απλά, αφού αρκετοί αλγόριθμοι πολυωνυμικού χρόνου έχουν αναπτυχθεί για την επίλυση ανάλογων προβλημάτων.

Ωστόσο, στους περισσότερους τομείς της ανθρώπινης δραστηριότητας πρέπει να βρεθούν βέλτιστες λύσεις και να ληφθούν σημαντικές αποφάσεις, λαμβάνοντας ταυτόχρονα υπόψη περισσότερα από ένα κριτήρια. Για παράδειγμα, στη βιομηχανία, στη σχεδίαση ενός συστήματος κινητού ή υπολογιστή, στην κατασκευή ενός αυτοκινήτου και αλλού. Κοινή επιδίωξη σε όλες τις περιπτώσεις η ελαχιστοποίηση του κόστους, είτε αυτός είναι σε ύλη είτε σε χρόνο και η μεγιστοποίηση της απόδοσης. Δυστυχώς όμως, τα δύο κριτήρια είναι αλληλοσυγκρουόμενα αφού συνήθως η μείωση του κόστους επιφέρει μείωση και της απόδοσης του συστήματος.

Σε όλα αυτά τα προβλήματα, το ζητούμενο είναι πώς μπορεί να οδηγηθεί κανείς στη λήψη μιας βέλτιστης για αυτόν απόφασης, που κάτω από δεδομένες συνθήκες μπορεί να είναι περισσότερες από μία. Δηλαδή, δεν οδηγούμαστε σε μία μοναδική βέλτιστη λύση, αλλά σε ένα σύνολο από «βέλτιστες λύσεις» και ο

ενδιαφερόμενος, ανάλογα με τα ιδιαίτερα χαρακτηριστικά του προβλήματος, κάνει την τελική επιλογή. Τα παραπάνω προβλήματα αποτελούν χαρακτηριστικά παραδείγματα προβλημάτων βελτιστοποίησης υπό πολλαπλά κριτήρια (multi-criteria optimization problems).

Για την επίλυση των προβλημάτων βελτιστοποίησης υπό πολλαπλά κριτήρια έχουν αναπτυχθεί διάφορες μέθοδοι. Οι προτεινόμενοι αλγόριθμοι εύρεσης ακριβούς λύσεις (exact algorithms) χρησιμοποιούν ένα ευρύ φάσμα μεθόδων, συμπεριλαμβανομένης της μεθόδου αθροίσματος σταθμισμένων βαρών (weighted sum scalarization), της μεθόδου ελαχιστοποίησης απόστασης από ένα ιδανικό σημείο (compromise solution method), της μεθόδου προγραμματισμού στόχων (goal programming) καθώς και διαφόρων μεθόδων διάταξης (ranking methods).

Τα προβλήματα βελτιστοποίησης με περισσότερα από ένα κριτήρια, τα οποία έχουν αποτελέσει αντικείμενο έρευνας και μελέτης, περιλαμβάνουν μεταξύ άλλων το πρόβλημα εύρεσης ελαχίστων γεννητικών δέντρων (minimum spanning tree problem), προβλήματα δικτυακών ροών (network flow problems), το πρόβλημα εύρεσης συντομότερων διαδρομών (shortest path problem), το πρόβλημα του περιοδεύοντος πωλητή (traveling salesman problem) καθώς και διάφορα προβλήματα μεταφορών (transportation problems)

Ειδικότερα το πρόβλημα εύρεσης ελαχίστων γεννητικών δέντρων αποτελεί ένα από τα πιο βασικά επιστημονικά προβλήματα με σημαντικές εφαρμογές στη σχεδίαση και βελτιστοποίηση δικτύων. Το εν λόγω πρόβλημα έχει αποτελέσει αντικείμενο έρευνας και μελέτης πολλών επιστημόνων με αποτέλεσμα την ανάπτυξη πολλών αποδοτικών αλγορίθμων πολυωνυμικού χρόνου. Στόχος της συγκεκριμένης διπλωματικής εργασίας είναι η παρουσίαση των σημαντικότερων από αυτούς τους αλγορίθμους. Αυτό πραγματώνεται μέσω γραφικών παραδειγμάτων για την επεξήγηση τους, αλλά και παρουσίαση ψευδοκώδικα για την εκτέλεση καθένα από αυτούς. Ακόμα έχει γίνει υλοποίηση των αλγορίθμων Kruskal, Dijkstra και Prim με χρήση της γλώσσας C αλλά και Pascal. Επίσης αναπτύχθηκαν αλγόριθμοι χειρισμού γράφων στην γλώσσα προγραμματισμού C και εξετάστηκαν γράφοι με 4, 5 και 6 κόμβους.

Κεφάλαιο 1

Γενικές έννοιες και ορισμοί

1.1 Εισαγωγικές έννοιες

Ξεκινώντας από το χώρο της δομής δεδομένων θα μπορούσαμε αρχικά να αναφέρουμε τις δύο μεγάλες ενότητες από τις οποίες αποτελούνται, τις «**Γραμμικές Δομές**» και τις «**Μη Γραμμικές Δομές**». Στις γραμμικές δομές δεδομένων τα δεδομένα είναι γραμμικά διατεταγμένα, δηλαδή κάποιο στοιχείο είναι πρώτο και κάποιο τελευταίο, ενώ για οποιοδήποτε στοιχείο υπάρχει ένα προηγούμενο και ένα επόμενο. Στις μη γραμμικές δομές οι σχέσεις μεταξύ των δεδομένων είναι περισσότερο περίπλοκες.

Οι δομές αυτού του είδους είναι τα δέντρα και οι γράφοι. Στα δέντρα κάθε στοιχείο έχει ένα μόνο προηγούμενο, ενώ μπορεί να έχει πολλά επόμενα στοιχεία. Στους γράφους κάθε στοιχείο μπορεί να μην έχει κανένα ή να έχει πολλά προηγούμενα και επόμενα στοιχεία. Η δομή ενός γράφου είναι η πιο γενική μορφή δομής δεδομένων.

1.2 Γενικές έννοιες - Ορισμοί

Ένας γράφος ή γράφημα (graph) G=(V, E) αποτελείται από δύο σύνολα αντικειμένων και συμβολίζεται ως εξής:

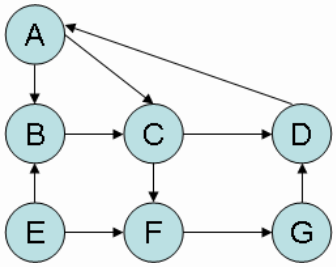
- ένα σύνολο σημείων (points) ή κορυφών (vertices) ή κόμβων (nodes) το πεπερασμένο σύνολο V
- ένα σύνολο ακμών (edges) ή τόξων (arcs) ή γραμμών (lines) που ενώνουν μερικά ή όλα τα σημεία του, το σύνολο E

Όπου,

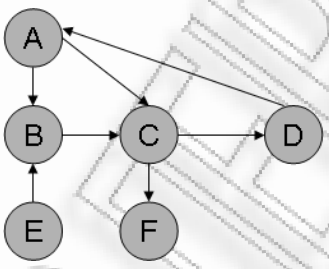
G=(V, E) με V = {V1,V2, … , Vn}, και n>0 είναι το σύνολο των κορυφών
E={[x,y] με x,y € V} το σύνολο των ακμών με |E| ≥0.

Οι **κορυφές ή οι ακμές** ενός γράφου χαρακτηρίζονται από ένα μοναδικό όνομα που ονομάζεται επιγραφή (label). Οι κορυφές ενός γραφήματος χρησιμοποιούνται για την παράσταση δεδομένων και οι ακμές για τη σχέση μεταξύ των δεδομένων.

Υπογράφος (**subgraph**) του G = (V, E) είναι ένας γράφος G΄ = (V΄, E΄) τέτοιος ώστε V΄ ⊆ V και E΄ ⊆ E (σχ.1α και 1β).



Σχ. 1α Γράφος G



Σχ. 1β Υπογράφος G_1 του G

Κατευθυνόμενος ή μη κατευθυνόμενος είναι ένας γράφος, αν οι ακμές του είναι ή δεν είναι προσανατολισμένες προς μία κατεύθυνση αντίστοιχα. Στον κατευθυνόμενο γράφο ο προσανατολισμός της ακμής συμβολίζεται με ένα βέλος (στη συνέχεια θα λέγεται **ακμή** η γραμμή που συνδέει δύο κόμβους ενός μη κατευθυνόμενου γράφου μόνο, σε αντίθεση με την γραμμή που συνδέει δύο κόμβους ενός κατευθυνόμενου γράφου η οποία θα λέγεται **τόξο**). Στον μη κατευθυνόμενο γράφο τα ζεύγη των κορυφών που ορίζουν τις ακμές του είναι μη διατεταγμένα (σχ.2α και 2β).

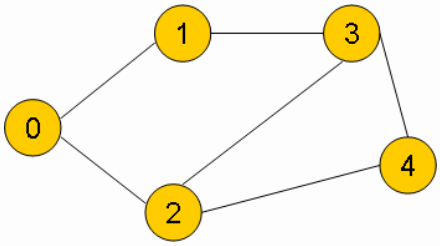
Σε ένα γράφο G=(V, E) η κορυφή i είναι γειτονική με την κορυφή j αν (I,j) € E και οι διαστάσεις του γράφου χαρακτηρίζονται από το πλήθος κορυφών |V| που καλείται **τάξη (order)** του G και το πλήθος των ακμών |E| που καλείται **μέγεθος** (**size**) του G. Το μέγεθος και η τάξη του γράφου επηρεάζουν τον χρόνο εκτέλεσης των αλγορίθμων.

Μη κατευθυνόμενος γράφος με |V| ≥5 και |E| ≥12

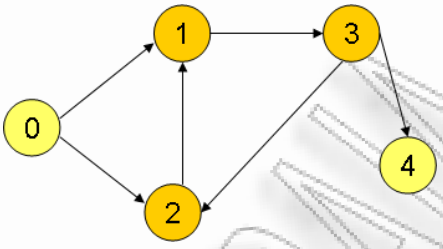
V = {0, 1, 2, 3, 4}
E = {(0,1), (0,2), (1,3), (1,0), (2,0), (2,3), (2,4), (3,4), (3,1), (3,2), (4,3), (4,2)}

Κατευθυνόμενος γράφος

V = {0, 1, 2, 3, 4 }
E = {(0,1), (0,2), (1,3), (2,1), (3,2), (3,4)}



Σχ. 2α Μη κατευθυνόμενος Γράφος G



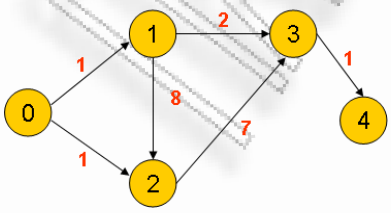
Σχ. 2β Κατευθυνόμενος Γράφος G

Για παράδειγμα, μεταξύ δύο κορυφών 1 και 3 τα βέλη από την 1 στη 3 και από τη 3 στη 1, θεωρούνται μία ακμή και όχι δύο. Στους κατευθυνόμενους γράφους θεωρούνται δύο ξεχωριστές ακμές ανάλογα με την κατεύθυνση τους. Για κατεύθυνση από τη 1 στη 3, το 1 είναι η **ουρά** (**tail**) και το 3 η **κεφαλή** (**head**) και αντίστροφα. Από τα παραπάνω προκύπτει ότι ένας μη κατευθυνόμενος γράφος μπορεί να θεωρηθεί και ως **συμμετρικός** κατευθυνόμενος γράφος (σχ. 2α και 2β).

Αποδεικνύεται ότι ο μέγιστος αριθμός ακμών για ένα μη κατευθυνόμενο γράφο με n κορυφές είναι n(n-1)/2, ενώ ο αντίστοιχος αριθμός σε κατευθυνόμενο γράφο είναι n(n-1). Ένας γράφος, κατευθυνόμενος ή μη, ονομάζεται **πλήρης** όταν έχει τον μέγιστο αριθμό ακμών.

Δίκτυο (network) ή **ζυγισμένος γράφος** λέγεται το γράφημα, όπου η κάθε ακμή του χαρακτηρίζεται από κάποιο αριθμό (έχει κάποια τιμή) που ονομάζεται βάρος ή βαρύτητα της ακμής. Στις εφαρμογές το βάρος μπορεί να δηλώσει το χρόνο μετάβασης από το ένα σημείο στο άλλο ή την απόσταση των δύο σημείων ή γενικότερα το κόστος (σχ. 3).

Υποθέτουμε ότι ο G=(V,E) είναι γράφος. Τότε κάθε συνάρτηση τύπου ω :E→N καλείται **συνάρτηση βάρους** (**weight function**). Ο γράφος G μαζί με τη συνάρτηση ω :E→N, καλείται γράφος με **βάρη** (**weighted graph**)



ω(0,1)=1 ω(1,3)=2
ω(0,2)=1 ω(2,3)=7
ω(1,2)=8 ω(3,4)=1

E = {(0,1), (0,2), (1,2), (1,3), (2,3), (3,4)}

Σχ. 3 Γράφος G με βάρη

Δύο κορυφές οι οποίες είναι διπλανές σε ένα γράφο ονομάζονται **γειτονικές**.
Για παράδειγμα στο σχ. 3 οι κορυφές 1 και 2 είναι γειτονικές.
Σε περίπτωση που δεν συνδέονται μεταξύ τους λέγονται **ανεξάρτητες**.
Για παράδειγμα στο σχ. 3 οι κορυφές 1 και 4 είναι ανεξάρτητές.

Αξίζει να σημειωθούν μερικές επιπλέον έννοιες:

Ένας **περίπατος** (**walk**) σε ένα γράφο G = (V, E) είναι μια ακολουθία κορυφών v_0, v_1, …,$v_k \in V$ τέτοια ώστε για κάθε i=1,…,k, $\{v_{i-1}, v_i\} \in E$. Σε αυτή την περίπτωση λέμε ότι έχουμε έναν περίπατο από τη v_0 στη v_k.

Αν όλες οι κορυφές είναι διακριτές, τότε ο περίπατος καλείται **μονοπάτι** (**path**), όπως για παράδειγμα το μονοπάτι $\{v_0, v_1, v_3, v_4\}$ στο γράφο G (σχ. 3).

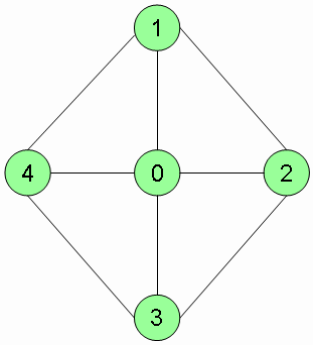
Αν όλες οι κορυφές είναι διακριτές εκτός από $v_0=v_k$ τότε ο περίπατος καλείται **κύκλος** (cycle), όπως για παράδειγμα ο κύκλος $\{v_0, v_1, v_2,\}$ στο γράφο G (σχ. 3).

Άκυκλος (acyclic) γράφος ονομάζεται ο γράφος στον οποίο δεν υπάρχουν μονοπάτια που να επιστρέφουν στον κόμβο από όπου ξεκίνησαν.

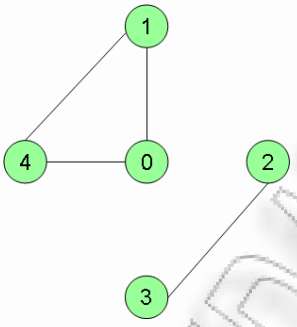
Ένας γράφος καλείται **συνεκτικός - συνδεδεμένος** (connected) αν για κάθε ζεύγος ξεχωριστών κορυφών χ,ψ υπάρχει ένας περίπατος από το χ στο ψ (σχ.4).
Ενώ **μη συνεκτικός - συνδεδεμένος** όταν υπάρχει ένα ζεύγος ξεχωριστών κορυφών για τις οποίες δεν υπάρχει περίπατος. (σχ.5)

Βαθμός κορυφής γράφου λέγεται ο αριθμός των ακμών που έχουν ως άκρο τους την κορυφή. Στην περίπτωση που ο γράφος είναι κατευθυνόμενος, τότε η έννοια του βαθμού επεκτείνεται στον έσω-βαθμό και τον έξω-βαθμό μίας κορυφής. Ο έσω και ο έξω βαθμός είναι ο αριθμός των ακμών στις οποίες η κορυφή είναι κεφαλή ή ουρά αντίστοιχα.

Σε συνέχεια των παραπάνω μπορούμε να δούμε ότι το 0, 1, 2, 0, 3, 4, 3 είναι περίπατος, αλλά όχι μονοπάτι ή κύκλος και ότι ο περίπατος 0, 1, 2, 3, 4 είναι μονοπάτι, ενώ ο περίπατος 0, 1, 2, 0 είναι κύκλος (σχ. 4).



Σχ. 4 Μονοπάτι και κύκλος σε γράφο G



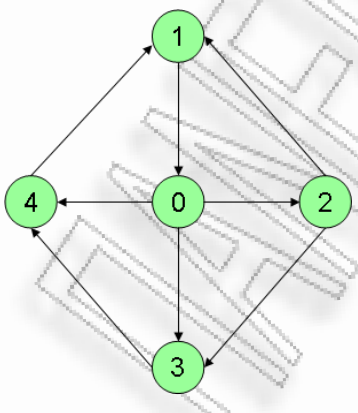
Σχ. 5 Μη συνδεδεμένος γράφος G

Υπάρχουν δύο κατηγορίες συνδεδεμένων γράφων:

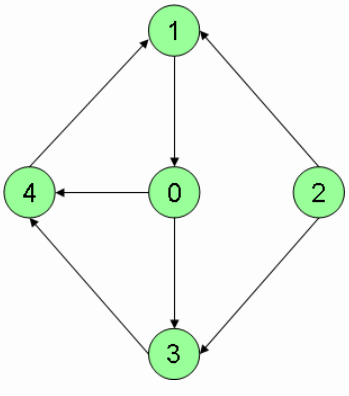
οι ισχυρά συνδεδεμένοι γράφοι (strongly connected)
οι ασθενώς συνδεδεμένοι (weakly connected).

Ισχυρά συνδεδεμένος (strongly connected) γράφος ονομάζεται ο γράφος ο οποίος είναι συνδεδεμένος με όλες τις κορυφές του ενώ **ασθενώς συνδεδεμένος (weakly connected)** γράφος ονομάζεται ο γράφος ο οποίος είναι συνδεδεμένος σε κάποιες από τις κορυφές του, αν αγνοήσουμε τις κατευθύνσεις των ακμών (σχ. 6α και 6β).

Ακολουθούν στα σχ. 6 (α) και (β) ένας ισχυρά συνδεδεμένος γράφος και ένας ασθενώς συνδεδεμένος γράφος αντίστοιχα.



Σχ. 6α Ισχυρά συνδεδεμένος γράφος G



Σχ. 6β Ασθενώς συνδεδεμένος γράφος G

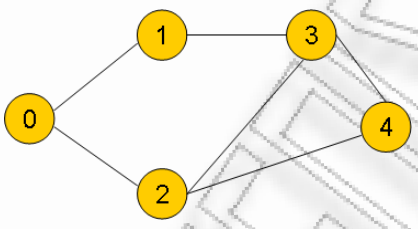
1.3 Υλοποίηση - Αποθήκευση γράφων

Στη συνέχεια, θα παρουσιάσουμε τους πιο συνηθισμένους τρόπους παράστασης ενός γράφου στον υπολογιστή:

- με τον πίνακα γειτνίασης (adjacency matrix)
- με τη λίστα γειτνίασης (adjacency list)
- με ορθογώνια λίστα
- με λίστα ακμών

1.3.1 Αναπαράσταση με πίνακα γειτνίασης

Για ένα γράφο με κόμβους n=0,1,2,3,…, |V| ο πίνακας γειτνίασης A= (aij), |V|x|V| ορίζεται ως aij =1 αν (i,j) ∈ E και αν για aij=0 δεν υπάρχει. Υπάρχει επιπλέον συμμετρία του πίνακα ως προς τη διαγώνιο aij όταν η ακμή ορίζεται με μη διατεταγμένο ζεύγος (μη κατευθυνόμενος) (σχ. 7).

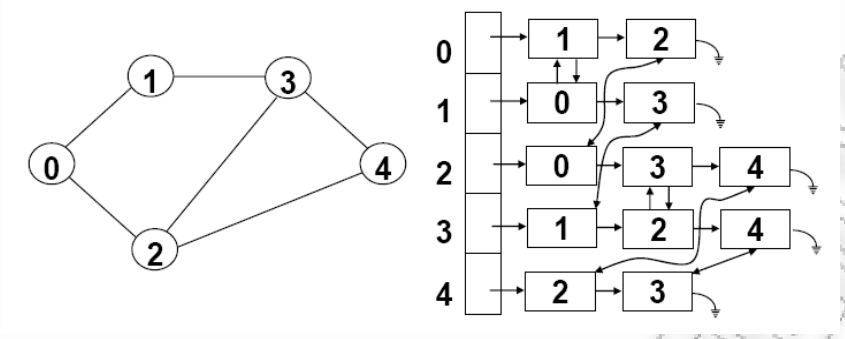


	0	1	2	3	4
0	0	1	1	0	0
1	1	0	0	1	0
2	1	0	0	1	1
3	0	1	1	0	1
4	0	0	1	1	0

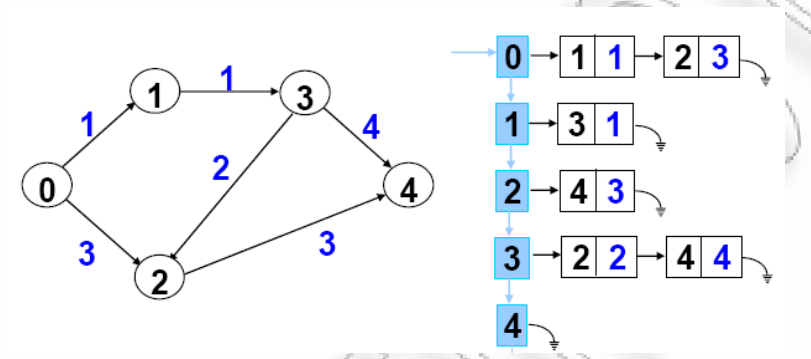
Σχ. 7 Γράφος G και πίνακας γειτνίασης

Καλούμε **βαθμό (degree)** μιας κορυφής u και συμβολίζουμε με deg(u) το πλήθος των ακμών που εισέρχονται και εξέρχονται στην κορυφή. Έτσι, όπως παρατηρούμε και στον παραπάνω πίνακα γειτνίασης, είναι δυνατός ο έλεγχος για ύπαρξη διασύνδεσης, ενώ τον θεωρούμε σπατάλη για αραιά γραφήματα πολυπλοκότητας O(V) αφού είναι καταλληλότερος για πυκνούς γράφους.

1.3.2 Αναπαράσταση με λίστα γειτνίασης



Σχ. 7α Αναπαράσταση με λίστα

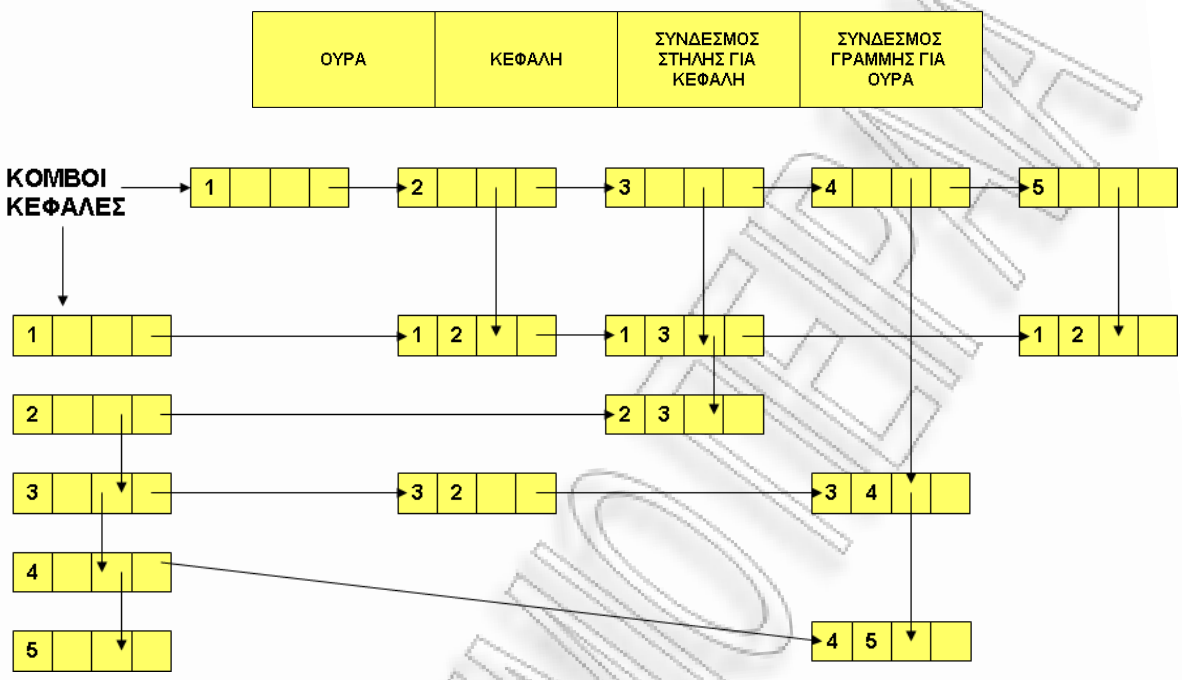


Σχ. 7β Αναπαράσταση με λίστα

Η αναπαράσταση με λίστα είναι καταλληλότερη για αραιά γραφήματα παρόλο το κόστος σε μνήμη στο πλήθος των ακμών του γραφήματος (|E| σε κατευθυνόμενο και 2|E| σε μη κατευθυνόμενο γράφημα). Στο παραπάνω σχήμα ορίζονται αρχικά οι δείκτες, έπειτα οι κόμβοι κεφαλές και οι κόμβοι γειτονικών κορυφών με τα αντίστοιχα βάρη (σχ. 7α,β).

1.3.3 Αναπαράσταση με ορθογώνια λίστα

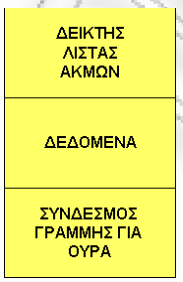
Σε αυτή τη μέθοδο χρησιμοποιείται μια απλουστευμένη μορφή παράστασης αραιών πινάκων, όπου κάθε κόμβος παριστάνει μια ακμή του γραφήματος και έχει τέσσερα πεδία με την ακόλουθη δομή (σχ. 8).



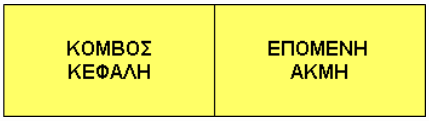
Σχ. 8 Αναπαράσταση με ορθογώνια λίστα

1.3.4 Αναπαράσταση με λίστα ακμών

Για την αναπαράσταση με λίστα ακμών χρησιμοποιούμε μια συνδεδεμένη λίστα όπου οι κορυφές του γραφήματος παριστάνονται σε μια συνδεδεμένη λίστα από κόμβους κεφαλές και κάθε κόμβος κεφαλή έχει τη μορφή που φαίνεται στο παρακάτω σχήμα (σχ.9α,β).



Σχ. 9α Αναπαράσταση με λίστα ακμών

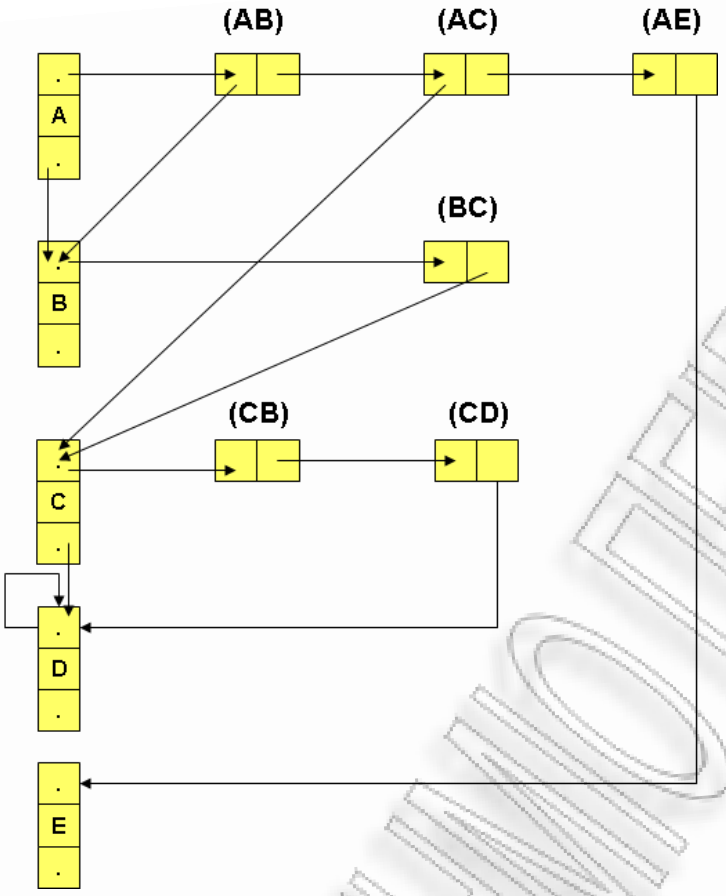


Σχ. 9β Αναπαράσταση με λίστα ακμών

Το πεδίο <<δεδομένα>> περιέχει όλες τις πληροφορίες που σχετίζονται με τη κορυφή του γραφήματος που παριστάνει ο κάθε κόμβος κεφαλή. Το πεδίο <<επόμενος κόμβος>> είναι ένας δείκτης στον κόμβο κεφαλή, που παριστάνει την επόμενη κορυφή του γραφήματος, αν υπάρχει. Κάθε κόμβος κεφαλή είναι η κεφαλή μίας λίστας κόμβων, η οποία καλείται λίστα ακμών. Κάθε κόμβος της λίστας ακμών παριστάνει μία ακμή του γραφήματος.

Το πεδίο <<δείκτης λίστας ακμών>> είναι ένας δείκτης στη λίστα ακμών, η οποία περιλαμβάνει όλες τις ακμές που ξεκινούν από τη κορυφή του γραφήματος που παριστάνει ο κόμβος κεφαλή. Κάθε κόμβος της λίστας ακμών έχει τη μορφή που παρουσιάζεται στο (σχ. 9β). Το πεδίο <<επόμενη ακμή>> είναι ένας δείκτης στην επόμενη ακμή που ξεκινά από την κορυφή του γραφήματος που παριστάνει ο κόμβος κεφαλής της. Τέλος, το πεδίο <<κόμβος κεφαλή>> είναι ένας δείκτης στο κόμβο κεφαλή που καταλήγει η ακμή. Ο τρόπος αυτός παράστασης ενός γραφήματος καλείται λίστα ακμών.

Ας σημειωθεί ότι οι κόμβοι κεφαλές και οι κόμβοι της λίστας ακμών έχουν διαφορετικές μορφές, πράγμα που σημαίνει ότι θα πρέπει να δημιουργηθούν δύο διαφορετικοί τύποι κόμβων ή ένας με χρήση ένωσης (union). Παρατηρώντας το πιο κάτω σχήμα μπορεί εύκολα να διαπιστωθεί ότι η κορυφή A είναι γειτονική με B,C και E ή ότι από τη κορυφή A ξεκινούν οι ακμές <AB>, <AC> και <AE> . Το γεγονός αυτό καθιστά εύκολη την επίσκεψη των κορυφών του γραφήματος αφού οι δείκτες της λίστας ακμών οδηγούν από τη μια κορυφή, διαμέσου της ακμής της, σε μία άλλη γειτονική της κορυφή. Οι δηλώσεις για την υλοποίηση αυτή φαίνονται στο ακόλουθο σχήμα (σχ. 9γ) .



Σχ. 9γ Αναπαράσταση με λίστα ακμών

Κεφάλαιο 2

Διαπεράσεις γράφων

2.1 Διαπεράσεις γράφων

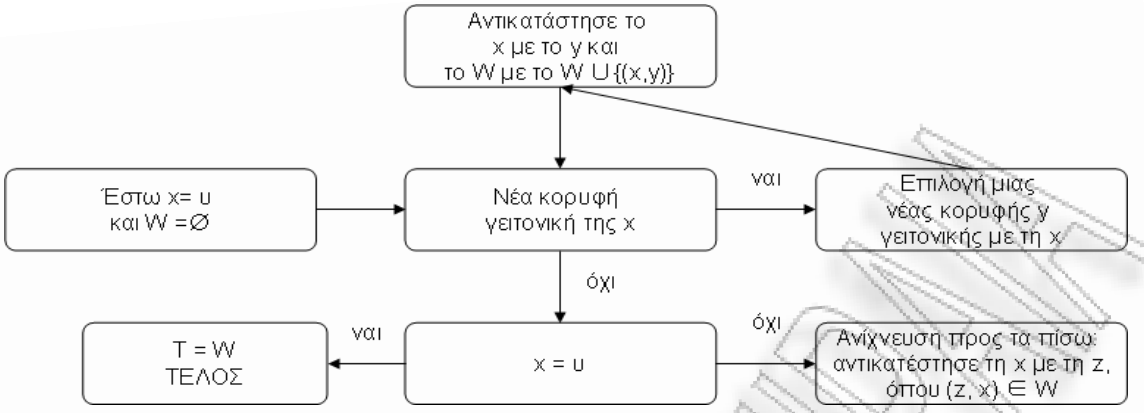
Κάθε διαδικασία συστηματικής και εξαντλητικής εξερεύνησης ενός γράφου, με την εξέταση των κορυφών και των ακμών ονομάζεται **διαπέραση** του γράφου. Παρόλη την απλοϊκότητα της λειτουργίας, οι διαπεράσεις είναι ισχυρές και ικανές να ανακαλύψουν τις ιδιότητες των γράφων. Ιδιαιτερότητα βέβαια θα έχουμε, όμοια με τη διάσχιση δέντρου, δεδομένου ότι σε μη κατευθυνόμενο γράφο γίνεται η επίσκεψη όλων των κόμβων ξεκινώντας από έναν από αυτούς. Για τη διάσχιση ενός γράφου υπάρχουν δύο είδη αλγορίθμων:

- Αλγόριθμος Αναζήτησης κατά βάθος (DFS - Depth First Search)
- Αλγόριθμος Αναζήτησης κατά πλάτος (BFS - Breadth First Search)

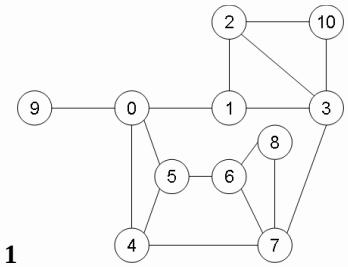
2.1.1 Αλγόριθμος Αναζήτησης κατά βάθος (DFS - Depth First Search)

Ο Αλγόριθμος DFS φροντίζει να επισκεφθεί όλους τους κόμβους του γράφου G=(V,E) πηγαίνοντας κάθε φορά και σε ένα νέο κόμβο τον οποίο δεν είχε επισκεφθεί προηγουμένως και είναι γειτονικός, δηλαδή ενώνεται με ακμή με τον προηγούμενο (σχ.11). Αρχίζει από την πρώτη κορυφή και μαρκάρει όσες έχει επισκεφθεί. Αν δεν υπάρχει τέτοιος κόμβος, γίνεται επιστροφή προς τα πίσω και συνεχίζει με τις γειτονικές της προηγούμενης κορυφής. Η αναζήτηση τελειώνει όταν όλες οι κορυφές έχουν ελεγχθεί. Ο έλεγχος των κορυφών γίνεται με κατακόρυφη κατεύθυνση. Έστω ο γράφος G=(V,E) με τα ακόλουθα βήματα (σχ.10) :

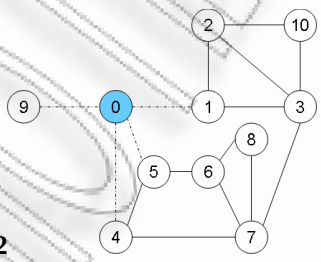
(1) Εκκίνηση από μια επιλεγμένη κορυφή $u \in V$
(2) Επαναληπτική επιλογή της επόμενης νέας γειτονικής κορυφής, έως ότου να μην υπάρχει νέα γειτονική κορυφή (επιλέγεται αυτή με το μικρότερο όνομα)
(3) Επιστροφή σε κορυφή που έχει νέα γειτονική κορυφή
(4) Επανέλαβε το βήμα 2 μέχρι να επιστρέψουμε στη u και να μην υπάρχει νέα γειτονική κορυφή (δηλ. έχουν εξεταστεί όλες)



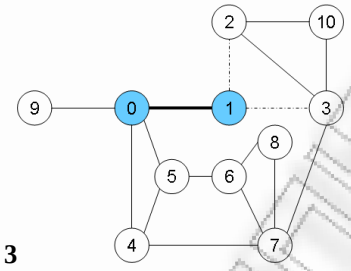
Σχ. 10 Σχηματική αναπαράσταση διαπέρασης σε βάθος



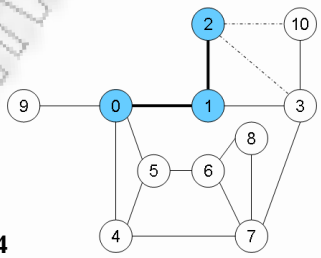
1



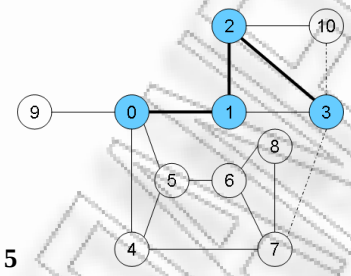
2



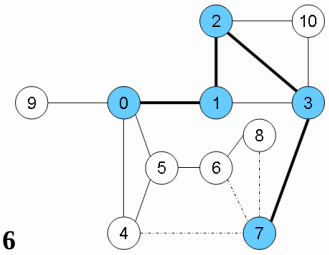
3



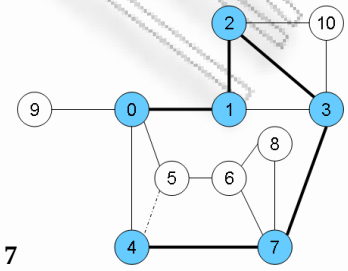
4



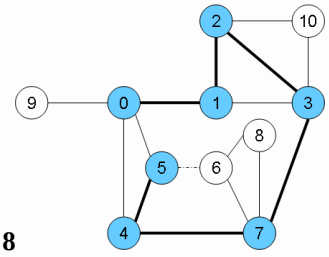
5



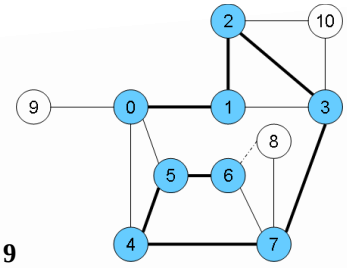
6



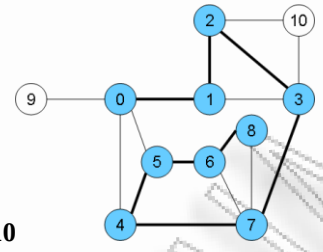
7



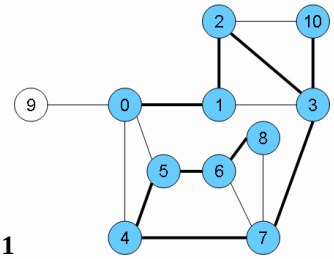
8



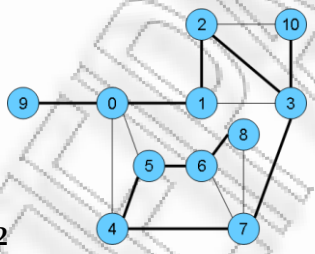
9



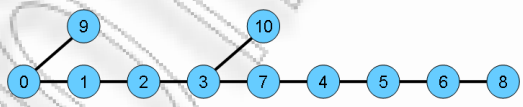
10



11



12



Σχ. 11 Παράδειγμα διαπέρασης σε βάθος

Μερικές από τις εφαρμογές της σε βάθους διαπέρασης είναι:

- Εντοπισμός κύκλου
- Εύρεση απλού μονοπατιού μεταξύ u και ω: εκκίνηση διαπέρασης σε βάθος από την u
- Επικαλύπτον δέντρο ή δάσος

Εάν το γράφημα είναι συνεκτικό, τότε το δέντρο που προκύπτει είναι επικαλύπτον. Αλλιώς, κάθε επανεκκίνηση ανακαλύπτει και μια συνεκτική συνιστώσα. Ακολουθεί, τέλος, ο ψευδοκώδικας του αλγόριθμου DFS (πιν.1).

```
1:    procedure dfs(V,E)
2:      V' := {v1} {v1 is the root of the spanning
3:    tree}
4:      E' := {} {no edges in the spanning tree yet}
5:      w := v1
6:      while true
7:          begin
8:              while there is an edge (w,v) that
9:    when added
10:                       to T does not create a cycle
11:   in T
12:               begin
```

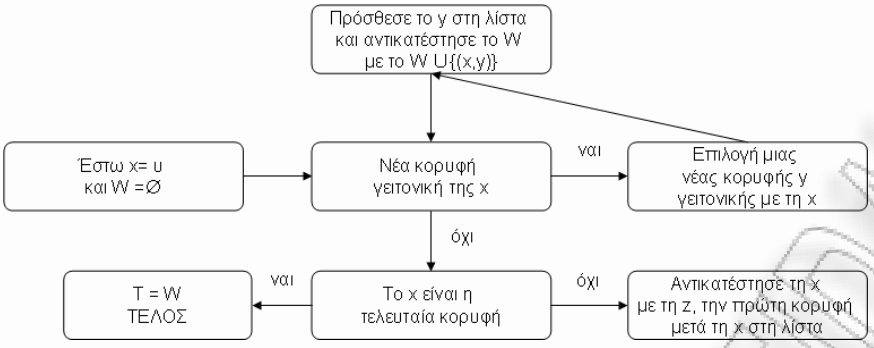
```
13:              Choose first v such that (w,v)
14:              does not create a cycle in T
15:              add (w,v) to E'
16:              add v to V'
17:              w := v
18:            end
19:          if w = v1 then
20:            return T
             w := parent of w in T {backtrack}
         end
     end dfs
```

Πιν. 1 Ψευδοκώδικας DFS

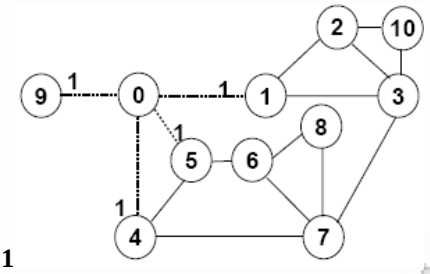
2.1.2 Αλγόριθμος Αναζήτησης κατά πλάτος (BFS - Breadth First Search)

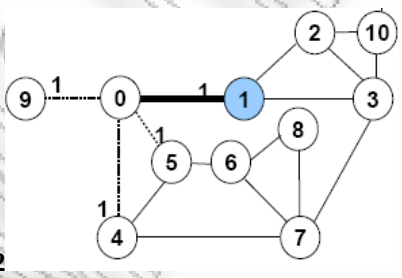
Ο Αλγόριθμος BFS φροντίζει να επισκεφθεί όλους τους κόμβους του γράφου G=(V,E) πηγαίνοντας κάθε φορά και σε ένα νέο κόμβο τον οποίο δεν είχε επισκεφθεί προηγουμένως και είναι γειτονικός δηλαδή ενώνεται με ακμή με τον προηγούμενο (σχ. 13α,β). Αρχίζει από την πρώτη κορυφή και μαρκάρει όσες έχει επισκεφθεί. Αν δεν υπάρχει τέτοιος κόμβος, γίνεται επιστροφή προς τα πίσω και συνεχίζει με τις γειτονικές της προηγούμενης κορυφής. Η αναζήτηση τελειώνει όταν όλες οι κορυφές έχουν ελεγχθεί. Ο έλεγχος των κορυφών γίνεται με οριζόντια κατεύθυνση. Έστω ο γράφος G=(V,E) με τα ακόλουθα βήματα (σχ.13) :

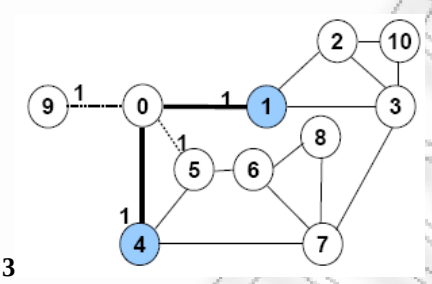
(1) Εκκίνηση από μια επιλεγμένη κορυφή $u \in V$
(2) Πρόσθεσε στη λίστα όλες τις νέες κορυφές που είναι γειτονικές στη u (κατά αύξουσα σειρά)
(3) Έστω η επόμενη κορυφή μετά τη u στη λίστα (FIFO σάρωση στη λίστα). Πρόσθεσε στη λίστα όλες τις νέες γειτονικές σε αυτή κορυφές
(4) Επανέλαβε τη διαδικασία με όλες τις κορυφές της λίστας, έως ότου φτάσουμε στην τελευταία κορυφή (αυτή που δεν έχει νέα γειτονική κορυφή)

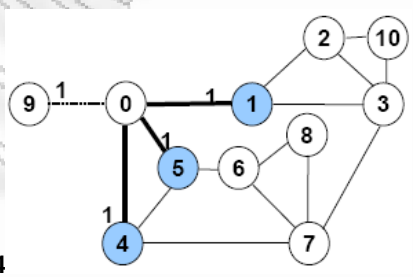


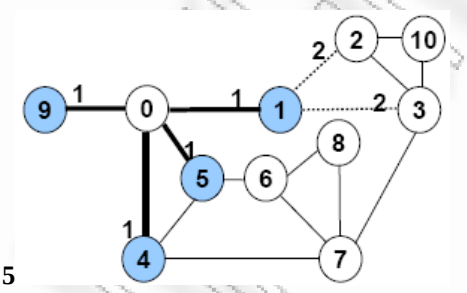
Σχ. 12 Σχηματική αναπαράσταση διαπέρασης σε πλάτος

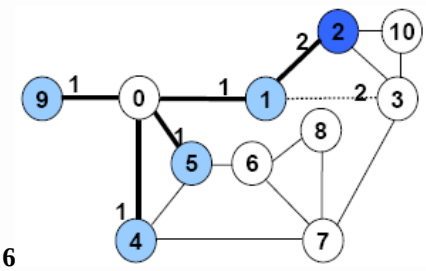


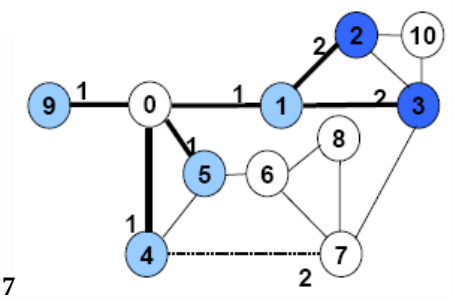




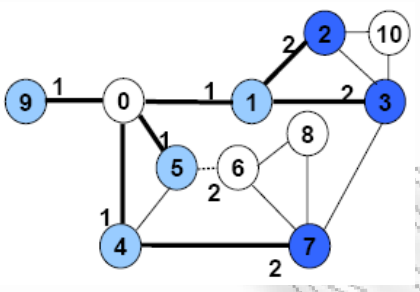




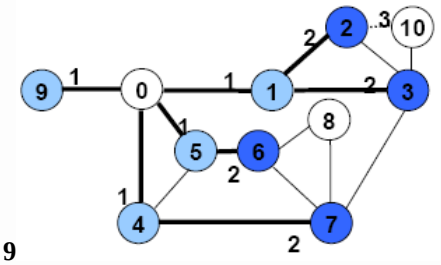




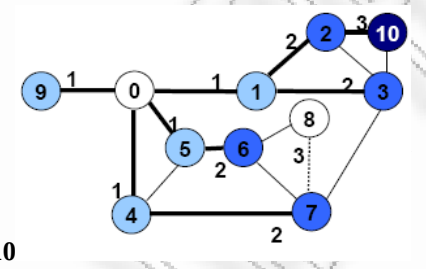
7



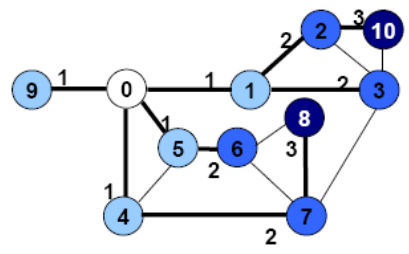
8



9

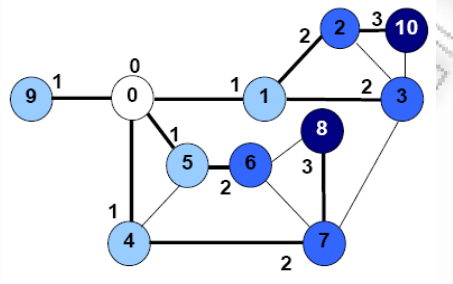


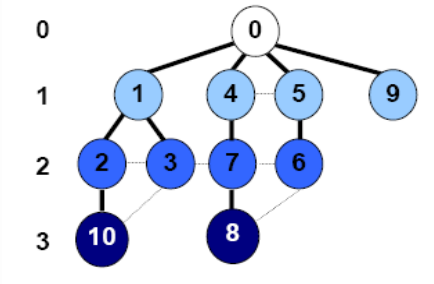
10



11

Σχ. 13α Παράδειγμα διαπέρασης σε πλάτος





Σχ. 13β Παράδειγμα διαπέρασης σε πλάτος - δέντρο

Μερικές από τις εφαρμογές της σε πλάτος διαπέρασης είναι:

- Εντοπισμός κύκλου
- Εύρεση απλού μονοπατιού μεταξύ υ και ω: εκκίνηση διαπέρασης πλάτους από την υ
- Επικαλύπτον δέντρο ή δάσος

Εάν το γράφημα είναι συνεκτικό, τότε το δέντρο που προκύπτει είναι επικαλύπτον (ορισμός κεφ. 3.2). Αλλιώς, κάθε επανεκκίνηση ανακαλύπτει και μια συνεκτική συνιστώσα. Ακολουθεί τέλος ο ψευδοκώδικας του αλγόριθμου BFS (πιν. 3).

```
1:   procedure bfs(V,E)
2:       S := (v1) {ordered list of vertices of a fix
3:   level}
4:       V' := {v1} {v1 is the root of the spanning
5:   tree}
6:       E' := {} {no edges in the spanning tree
7:   yet}
8:       while true
9:           begin
10:              for each x in S, in order,
11:                  for each y in V - V'
12:                      if (x,y) is an edge then
13:                          add edge (x,y) to E' and
14:  vertex y to V'
15:                  if no edges were added then
                         return T
                 S := children of S
             end
     end bfs
```

Πιν. 3 Ψευδοκώδικας BFS

2.2 Συμπεράσματα

Οι πιο πάνω αλγόριθμοι προσεγγίζουν, αλλά δεν λύνουν το πρόβλημα διάσχισης ενός λαβύρινθου. Παρόλα αυτά, τα περισσότερα από τα αποτελέσματα αποτελούν τη βάση για τη διατύπωση ενός ειδικού αλγορίθμου για τη λύση του προβλήματος. Παράλληλα, εξασφαλίζουν ότι κάποιος θα διασχίσει κάθε ακμή του γράφου μόνο δύο φορές, μία για κάθε κατεύθυνση. Επίσης, δεν δίνουν λύση στο πρόβλημα του περιοδεύοντος πωλητή, δηλαδή εύρεση του συντομότερου μονοπατιού διάσχισης μεταξύ πολλών πόλεων (συντομότερο μονοπάτι γράφου) για το οποίο υπάρχει διαφορετική μεθοδολογική προσέγγιση.

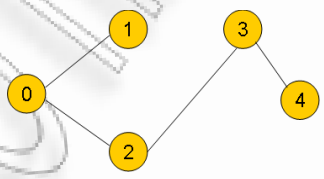
Κεφάλαιο 3

Δέντρα

3.1 Επικαλύπτοντα Δέντρα

Ένας γράφος T=(V,E) μπορούμε να πούμε ότι είναι δέντρο (tree) αν ικανοποιεί τις ακόλουθες συνθήκες:

- Ο Τ είναι συνεκτικός
- Ο Τ δεν περιέχει κύκλο



Επιπλέον, αναφέρουμε τις ακόλουθες προτάσεις:

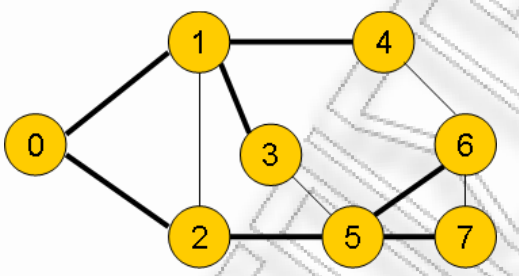
- Έστω ότι ο T=(V, E) είναι δέντρο με τουλάχιστο δύο κορυφές. Τότε, για κάθε ζεύγος διακριτών κορυφών x,y Є V υπάρχει ένα μοναδικό μονοπάτι στο Τ από το x στο y.
- Έστω ότι ο T=(V, E) είναι δέντρο με τουλάχιστο δύο κορυφές. Τότε, ο γράφος που προκύπτει από το Τ μετά την αφαίρεση μιας ακμής έχει δύο συνιστώσες (components), κάθε μια από τις οποίες είναι δέντρο.
- Έστω ότι ο T=(V, E) είναι δέντρο. Τότε |E| = |V| -1. Η επαγωγή στο πλήθος των κορυφών του Τ ισχύει και είναι:
  - ✓ Για |V| = 1 ισχύει
  - ✓ Υποθέτουμε ότι ισχύει για |V| = k
  - ✓ Έστω |V| = k+1. Αν αφαιρέσουμε μια ακμή από τον Τ τότε από την προηγούμενη πρόταση προκύπτει ένας γράφος με δύο συνιστώσες [έστω T1= (V1, E1) και T2= (V2, E2)] κάθε μια από τις οποίες είναι δέντρο. Επειδή |V1|<= k και |V2|<=k, από την υπόθεση ισχύει: |E1|= |V1| -1 και |E2| = |V2| -1. Επειδή |V| = |V1| + |V2| και |E| = |E1| + |E2| +1, προκύπτει το ζητούμενο.

3.2 Ζευγνύον δέντρο ή δέντρο επικάλυψης (Spanning Tree - ST)

Έστω ένας συνεκτικός γράφος G=(V, E). Ένα υποσύνολο T του E καλείται **επικαλύπτον δέντρο** του G, αν το T ικανοποιεί τις ακόλουθες δύο συνθήκες:

- Κάθε κορυφή στο V ανήκει σε μια ακμή του T.
- Οι ακμές στο T σχηματίζουν ένα δέντρο.

Είναι, δηλαδή, το ελάχιστο πλήθος ακμών που επιτρέπει την επικοινωνία μεταξύ οποιονδήποτε κορυφών του αρχικού γραφήματος (σχ.14). Το ζευγνύον δέντρο ονομάζεται και σκελετός ή max δέντρο. Κάθε ζευγνύον δέντρο n κόμβων έχει n-1 ακμές. Το επικαλύπτον δέντρο του παρακάτω σχήματος αποτελείται από τις ακμές (0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (5, 6), (5, 7) όπου, προφανώς, δεν υπάρχει ένα μόνο επικαλύπτον δέντρο. Επιπλέον, ορίζουμε ως **διάμετρο Δέντρου** το μονοπάτι με την ελάχιστη απόσταση που καλύπτει όλες τις κορυφές.



Σχ. 14 Παράδειγμα γράφου και επικαλύπτοντος δέντρου

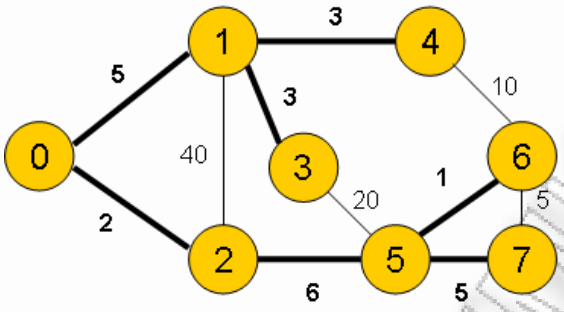
3.3 Ελάχιστο Ζευγνύον δέντρο (Minimum Spanning Tree - MST)

Έστω *G* ένας γράφος με βάρη. Υποθέτουμε ότι ο *G* είναι συνεκτικός και έστω T ένα επικαλύπτον δέντρο του *G*. Η τιμή:

$$\omega(T) = \sum_{e \in T} \omega(e)$$

μας δίνει το άθροισμα των βαρών των ακμών του T και καλείται βάρος του επικαλύπτοντος δέντρου.

Ελάχιστο ζευγνύον δέντρο (Minimum spanning tree) ονομάζεται το ζευγνύον δέντρο με το ελάχιστο βάρος ,δηλαδή το δέντρο με το μικρότερο άθροισμα min*ω(T).* Το πρόβλημα αυτό έχει μεγάλη σημασία αφού μπορεί να ανακύψει σε πληθώρα καταστάσεων, όπου πρέπει να κατασκευαστεί ένα δίκτυο από κόμβους (π.χ. τηλεπικοινωνιακούς) που συνδέονται ανά δύο με κάποιο επιμέρους κόστος (σχ.15).



Σχ. 15 *Ελάχιστο ζευγνύον δέντρο*

Παρακάτω αναφέρουμε μερικούς από τους ορισμούς των MST

- Για κάθε επικαλύπτον δέντρο T του G έχουμε ω(T) ∈ N.
- Υπάρχουν πεπερασμένα επικαλύπτοντα δέντρα T ενός γράφου G .
- Θα πρέπει να υπάρχει ένα επικαλύπτον δέντρο T για το οποίο η τιμή ω(T) είναι η ελάχιστη μεταξύ όλων των επικαλυπτόντων δέντρων του G.

Υποθέτουμε ότι ο G=(V,E) μαζί με μια συνάρτηση βάρους ω:E→N, σχηματίζουν ένα γράφο με βάρη .Έστω, επίσης, ότι ο G είναι συνεκτικός. Τότε, ένα επικαλύπτον δέντρο T του G, για το οποίο το βάρος ω(T) είναι το μικρότερο μεταξύ όλων των επικαλυπτόντων δέντρων του G, καλείται **ελάχιστο επικαλύπτον δέντρο**(**minimal spanning tree**) του G.

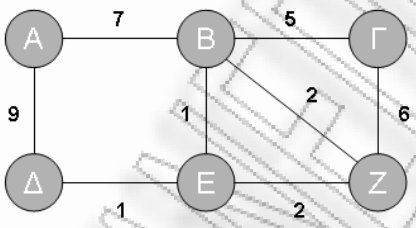
3.4 Αλγόριθμος Kruskal

Ο αλγόριθμος του Kruskal επεξεργάζεται μια προς μια τις ακμές του γράφου κατά αύξουσα τιμή βάρους και τις προσθέτει μόνο εφόσον δε δημιουργείται κύκλος. Οι επιλεγμένες ακμές σχηματίζουν ένα δάσος (ένα σύνολο δέντρων). Ξεκινά από V στοιχειώδη δέντρα, τα οποία με τις διαδοχικές προσαρτήσεις ακμών ενοποιούνται στο τελικό MST. Σχετικά με την ορθότητα έχουμε:

- Σε κάθε βήμα, έστω e=(a,b) η ακμή που επιλέγεται, a<b
- V_1= όλες οι γειτονικές, μέσω ακμών δέντρων, της a κορυφές
- V_2= V-V_1

Αναλυτικά φαίνονται τα παρακάτω βήματα,

(1) → Αρχικά το T είναι άδειο.
(2) → Επεξεργαζόμαστε μια-μια τις ακμές με σε αύξουσα σειρά βάρους
(3) → Η e έχει το ελάχιστο βάρος από αυτές που δεν έχουμε μέχρι στιγμής επεξεργασθεί
(4) → Ελέγχουμε αν η εισαγωγή της e στο T δεν προκαλεί κύκλο.
(5) → Αν η απάντηση είναι θετική, προσθέτουμε την e στο T, δηλ. T: = T - {e}.

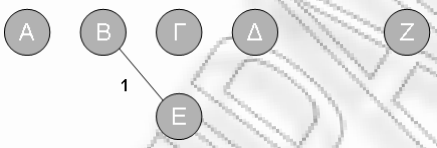


Σχ. 16 *Δέντρο - Παράδειγμα MST*

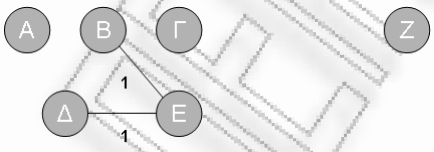
Σχετικά με το παραπάνω παράδειγμα (σχ.16) έχουμε τον εξής πίνακα βημάτων.

Αρχική Κατάσταση

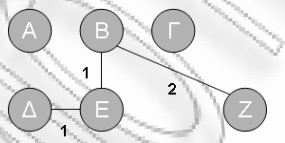
Μετά από επιλογή της πρώτης ακμής (B,E)



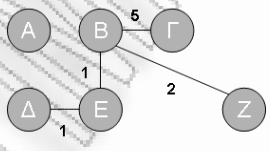
Μετά από επιλογή της δεύτερης ακμής (Δ,E)



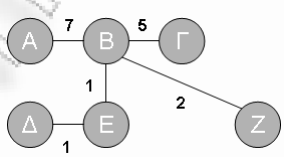
Μετά από επιλογή της τρίτης ακμής (B,Z)



Μετά από επιλογή της τέταρτης ακμής (B,Γ)



Μετά από επιλογή της πέμπτης ακμής (A,B,)



Πιν. 1 Βήματα στο παράδειγμα (σχ.16) MST

Ακολουθεί ο ψευδοκώδικας του αλγόριθμου Kruskal

- Μια ουρά προτεραιότητας αποθηκεύει τις ακμές έξω

από το σύννεφο

✓ Key: βάρος

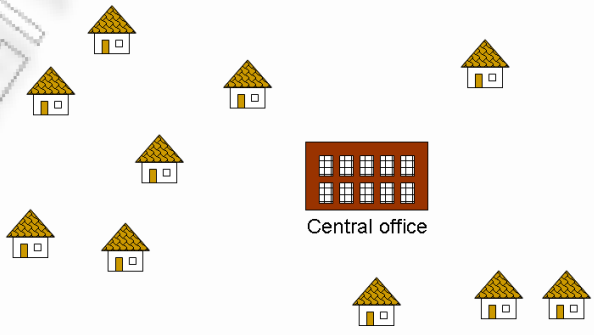
✓ Element: ακμή

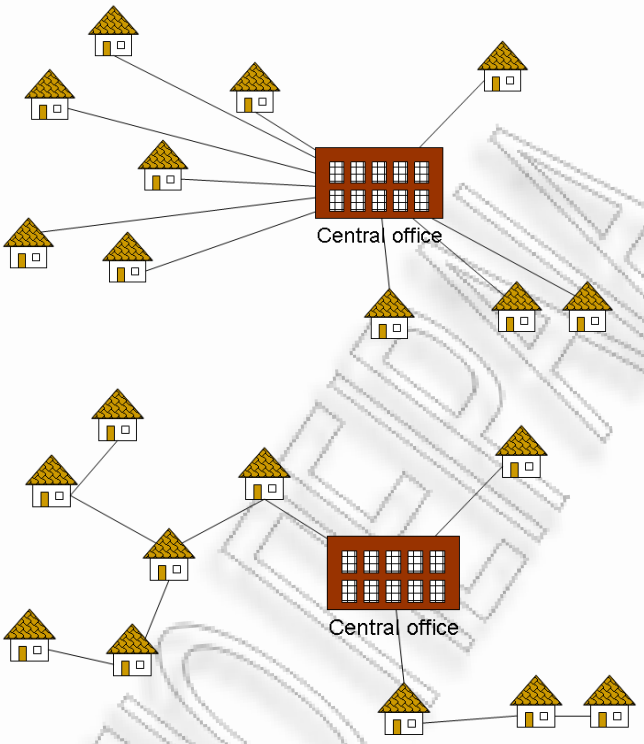
- Στο τέλος του αλγόριθμου
  - ✓ Μένουμε με ένα σύννεφο που περιγράφει το MST
  - ✓ Ένα δέντρο T το οποίο είναι το MST

Algorithm *KruskalMST*(*G*)
for each vertex ***V*** in ***G*** **do**
define a ***Cloud(v)*** of ← {***v***}
let *Q* be a priority queue.
Insert all edges into ***Q*** using their weights as the key
T ← ∅
while *T* has fewer than ***n*** -1 edges **do**
edge ***e*** = ***T.removeMin()***
Let ***u***, ***v*** be the endpoints of *e*
if *Cloud(v)* ≠ *Cloud(u)* then
Add edge ***e*** to ***T***
Merge ***Cloud(v)*** and ***Cloud(u)***
return *T*

Παρακάτω παραθέτουμε ένα κοινό πρόβλημα τηλεφωνικών συνδέσεων στο οποίο βρίσκει εφαρμογή ο αλγόριθμος Kruskal .

Πρόβλημα και αρχική κατάσταση





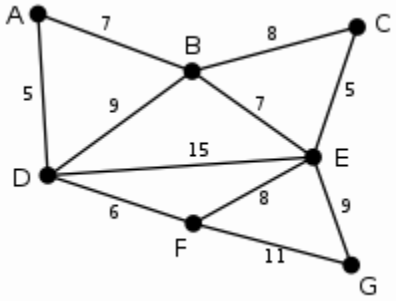
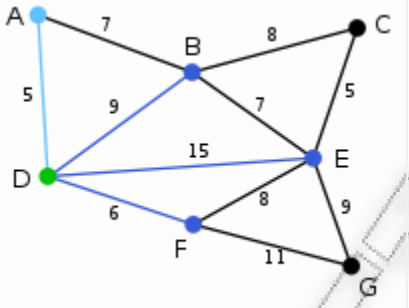
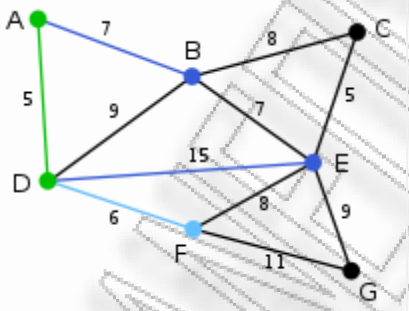
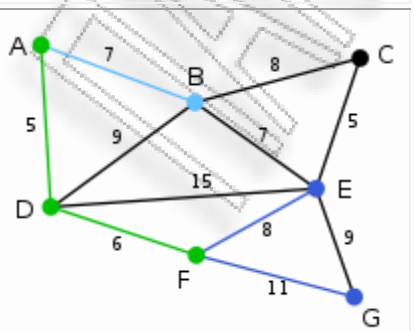
Πιν. 3 Πρόβλημα τηλεφωνικών συνδέσεων

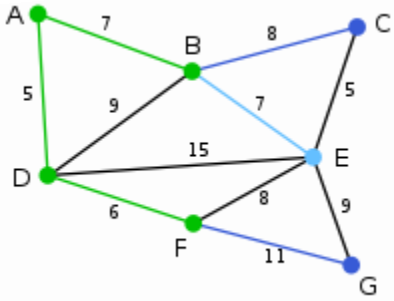
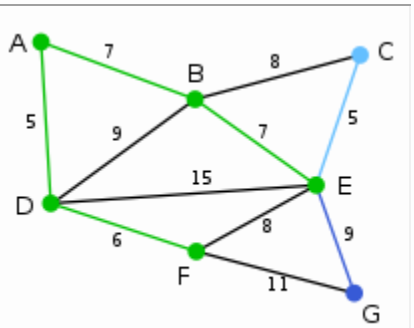
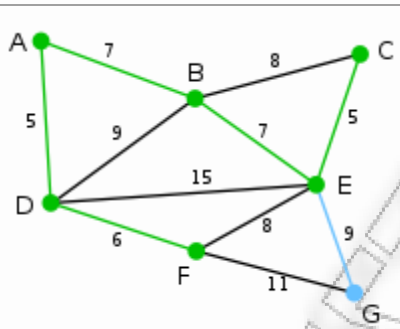
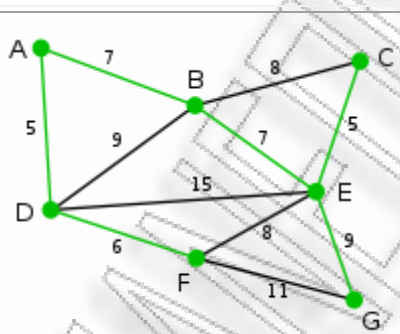
3.5 Ο αλγόριθμος του PRIM.

Ένα εκτεταμένο δέντρο σ' ένα γράφο είναι ένας υπογράφος που επεκτείνεται σ' όλους τους κόμβους και δεν έχει κύκλους (loops). Το ελάχιστο εκτεταμένο δέντρο είναι το εκτεταμένο δέντρο με το ελάχιστο άθροισμα των μηκών των συνδέσεων.

Περιγράφοντας τα βήματα του αλγόριθμου θα λέγαμε ότι αυξάνει συνεχώς το μέγεθος του δέντρου, το ένα άκρο κάθε φορά, ξεκινώντας με ένα δέντρο που αποτελείται από μία μόνο κορυφή μέχρι να καλύπτει όλες τις κορυφές. Ακολουθεί αναλυτικό γραφικό παράδειγμα:

Παράδειγμα.

Εικόνες	Περιγραφή
	Έστω ο αρχικός ζυγισμένος γράφος της διπλανής εικόνας. Οι αριθμοί πάνω από τις ακμές υποδεικνύουν το βάρος τους .
	Η κορυφή D έχει επιλεγεί τυχαία σαν αρχικό σημείο. Οι κορυφές A, B, E και F είναι συνδεδεμένες με την D . Η κορυφή A είναι η κοντινότερη στην D και θα επιλεγεί σαν τη δεύτερη κορυφή της διαπέρασης .
	Η επομένη κορυφή για επιλογή θα είναι είτε η κοντινότερη στην κορυφή D είτε στην A. Η κορυφή B απέχει 9 από την D και 7 από την A, ενώ η E απέχει 15 από την D και η F απέχει 6. Συνεπώς, θα επιλέξουμε την F, αφού είναι αυτή που απέχει λιγότερο.
	Ο αλγόριθμος συνεχίζεται με την ίδια λογική. Έτσι, η κορυφή B, η οποία απέχει 7 από την A, επιλέγεται .

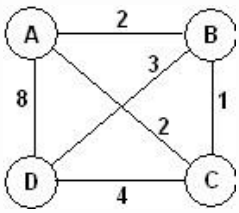
	Στο συγκεκριμένο βήμα μπορούμε να διαλέξουμε μεταξύ των κορυφών C, E, και G. Η κορυφή C απέχει 8 από την B, η κορυφή E 7 από την B, και η G 11 από την F. Η κορυφή E είναι αυτή που απέχει λιγότερο, συνεπώς την επιλέγουμε.
	Σε αυτό το βήμα οι μοναδικές κορυφές που μπορούν να επιλεγούν είναι η C και η G. Η C απέχει 5 από την E, ενώ η G απέχει 9 από την E. Συνεπώς, επιλέγουμε την C .
	Η μοναδική κορυφή που απομένει είναι η G .Απέχει 11 από την F και 9 από την E. Αφού είναι πιο κοντά στην E , επιλέγουμε το τόξο EG.
	Τέλος, έχουν επιλεγεί όλες οι κορυφές συνεπώς έχουμε βρει το ελάχιστο ζευγνύον δέντρο (minimum spanning tree) που είναι η πράσινη διαδρομή και έχει βάρος 39.

Ακολούθως θα παρουσιάσουμε το ψευδοκώδικα του αλγόριθμου Prim.

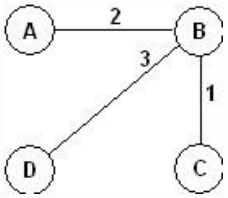
```
Prim(graph G)
{
int C[n]=∞, P[n];
int S[n]=0;
διάλεξε τυχαία κορυφή v;
S[v] = 1;
Tree = {};
for (i=1; i<|V|; i++)
{
για κάθε w γείτονα του v
if (d(v,w) < C[w])
P[w] = v; C[w] = d(v,w);
v = minVertex (S, C);
S[v]=1;
Tree = Tree _ {(P[v],v)};
}
}
minVertex(int S[], int C[])
{
επίστρεψε την κορυφή j για την οποία S[j] = 0 και για κάθε κορυφή k,
if S[k]=0 then C[j] <= C[k] }
```

3.6 Σύγκριση αλγορίθμου Prim και Kruskal .

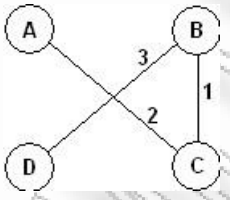
Θα πρέπει να αναφέρουμε ότι οι αλγόριθμοι Prim και Kruskal δεν είναι απαραίτητο να παράγουν το ίδιο ελάχιστο ζευγνύον δέντρο.
Για παράδειγμα στον γράφο:



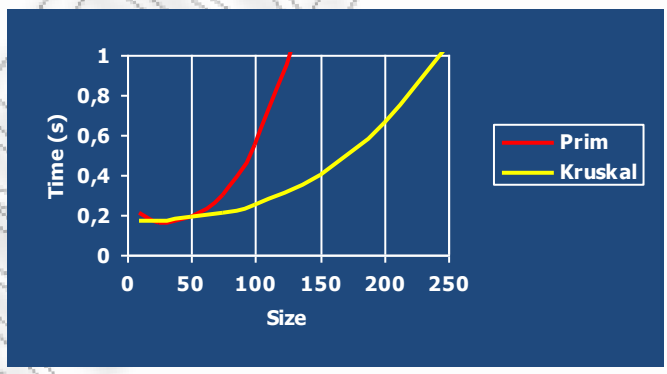
Ο αλγόριθμος του Kruskal έχει ως αποτέλεσμα το παρακάτω ελάχιστο ζευγνύον δέντρο:



Το ίδιο αποτέλεσμα θα παράγει και ο αλγόριθμος Prim αν ξεκινήσουμε από τις κορυφές A,B ή D. Αν, όμως, ξεκινήσουμε από την κορυφή C, τότε το ελάχιστο ζευγνύον δέντρο θα είναι:



Παρά το γεγονός ότι και οι δύο αλγόριθμοι δίνουν λύσεις στο πρόβλημα ελαχίστου δέντρου, παρατηρούμε ότι ο αλγόριθμος Prim είναι συγκριτικά πιο γρήγορος ενώ ο αλγόριθμος Kruskal είναι ευκολότερος στην υλοποίηση του μέσω κώδικα.



3.7 Ο αλγόριθμος dijkstra.

Ο αλγόριθμος dijkstra είναι ιδιαίτερα δημοφιλής στην εύρεση του ελάχιστου μονοπατιού σε ένα κατευθυνόμενο γράφημα.

Βασική ιδέα:

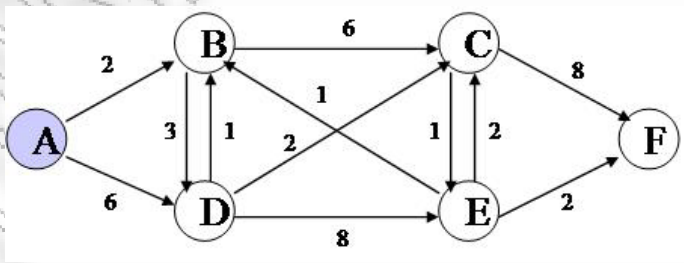
Αρχικά βρίσκουμε τον κόμβο Κ που συνδέεται με τον 1 με το μικρότερο μήκος σύνδεσης. Στη συνέχεια, ο συντομότερος δρόμος θα περιλαμβάνει είτε έναν άλλο κόμβο Κ' που συνδέεται με τον 1 είτε έναν άλλον κόμβο Κ'' που συνδέεται με το Κ', αναλόγως του ποιο από τα δυο είναι μικρότερο: d(K,K') ή d(K,K') + d(K',K''). Συνεχίζουμε μέχρι να φτάσουμε στον τελικό κόμβο του γραφήματος.
Συμβολίζουμε με dij το μήκος της σύνδεσης των i και j (όπου υποθέτουμε ότι dij = ∞, αν οι i και j δεν συνδέονται).

Αναλυτικά η βασική ιδέα του αλγορίθμου εμφανίζεται :

Αρχικά: P = {1}, D1 = 0, Dj = dj1, j ≠ 1.

Βήμα 1: Βρείτε i ∉ P τέτοιο ώστε Di = minj ∉ PDj.
Θέσατε P := P ∩ {i}. Αν P = S, τερματίστε.

Βήμα 2: Για κάθε j ∉ P, θέσατε Dj = min[Dj, dji + Di].
Πάτε στο βήμα 1.



Σχ. 17

Ακολουθεί ένα παράδειγμα εφαρμογής του Αλγόριθμου Dijkstra βασισμένο στο σχ.17 :

Ξεκινάμε τυχαία με την Α κορυφή.

AB=2, AD=6, AC=∞, AE= ∞, AF=∞

Παίρνουμε την AB απόσταση γιατί είναι η μικρότερη. Επομένως, τη κορυφή B.

BC=6, BE=∞, BD=3, BF=∞. Επομένως, παίρνουμε τη κορυφή D.

DE=8, DC=2, DF=∞. Επομένως, παίρνουμε τη κορυφή C.

CE=1, CF=8. Επομένως, παίρνουμε την E. Και τελευταία είναι η F.

Επομένως, το μονοπάτι μας είναι: ¨ABDCF¨.

3.8 Ο αλγόριθμος FORD.

Ο αλγόριθμος FORD είναι ένας αλγόριθμος εύρεσης συντομότερου μονοπατιού σε ένα γράφο.

Λειτουργία του αλγόριθμου FORD:

Δίνεται σε κάθε κόμβο μία τιμή λ η οποία είναι μηδενική για τον πρώτο κόμβο και ίση με ∞ για τους υπόλοιπους.
Υπολογίζονται οι διαφορές των τιμών λ μεταξύ δύο κόμβων i,j αν λj-λi μεγαλύτερο από το βάρος του τόξου i,j τότε το λj γίνεται ίσο με το άθροισμα των λi και του τόξου των δύο κόμβων.

Σε περίπτωση που δεν υπάρχει τόξο i-j τότε το τόξο i,j γίνεται ίσο με ∞.

Ο αλγόριθμος τερματίζεται όταν δεν μπορεί να μεταβληθεί άλλο το λi.

3.9 Εφαρμογές γραφημάτων

Η εύρεση του πλήθους των μονοπατιών μεταξύ δύο κορυφών και το κόστος επικοινωνίας είναι δύο από τα προβλήματα των γραφημάτων που παρουσιάζουν το μεγαλύτερο ενδιαφέρον. Τα γραφήματα έχουν πολλές εφαρμογές όπως για παράδειγμα στα δίκτυα. Στην περίπτωση, παραδείγματος χάρη, κατά την οποία θα θέλαμε να δοκιμάσουμε όλες τις πιθανότητες, χωρίς όμως να χρησιμοποιήσουμε την προτεινόμενη μεθοδολογία για ένα γράφο που απεικονίζει ένα σύστημα επικοινωνιών με n=75 κορυφές, τότε ο αριθμός των ST είναι n^{n-2} ,δηλαδή

75765628046446014790863186515904134648140 67\
83308840339247043281018024279971356804708193\
5219466686248779296875

Το ζευγνύον δέντρο (Spanning Tree) σχεδιάστηκε για τη λύση των προβλημάτων της συμφόρησης που δημιουργείται, για παράδειγμα, από τη σύνδεση τοπικών δικτύων (LANs) με υπεράριθμες γέφυρες (bridges) μεταφοράς. Ο πυρήνας του προβλήματος είναι η ποιότητα που έχουν αυτές οι γέφυρες. Με το ζευγνύον δέντρο λύνουμε αντίστοιχα προβλήματα μετακινώντας (ή κόβοντας) όλα τα περιττά μονοπάτια. Κατ' αυτόν τον τρόπο, μειώνεται η τοπολογία της δομής του δέντρου στα σημεία που εγγυείται πλήρη σύνδεση. Ο αλγόριθμος το επιτυγχάνει αυτό επιλέγοντας μια κυρίως γέφυρα και αναγκάζοντας κάθε άλλη γέφυρα στην τοπολογία αυτή να επιλέγει μία θύρα η οποία θα την συνδέει με το λιγότερο κόστος στην κυρίως γέφυρα.

3.10 Ζευγνύοντα δέντρα εξαναγκασμένης διαμέτρου (Diameter-Constrained Minimum Spanning Tree – DCMST)

Το πρόβλημα DCMST μπορεί να τεθεί ως ακολούθως: με δεδομένο ένα μη κατευθυνόμενο γράφο G, στον οποίο γνωρίζουμε το βάρος των ακμών και ένα ακέραιο, τον k, αναζητούμε ένα ζευγνύον δέντρο (spanning tree), με το μικρότερο βάρος μεταξύ όλων των ζευγνυόντων δέντρων, το οποίο να μη περιέχει μονοπάτι με περισσότερες από k ακμές. Το μήκος του μεγαλύτερου μονοπατιού στο δέντρο ονομάζεται διάμετρος του δέντρου.

Αποδεικνύεται ότι για ένα γράφο G με n κόμβους το πρόβλημα DCMST μπορεί να λυθεί σε πολυωνυμικό χρόνο για τις εξής ειδικές περιπτώσεις: για k=2, k=3, k=(n-1) ή όταν τα βάρη όλων των ακμών του είναι τα ίδια. Το πρόβλημα DCMST έχει εφαρμογές σε πολλά πεδία, όπως στα κατανεμημένα συστήματα, στα δίκτυα οπτικών επικοινωνιών και στην συμπίεση για την ανάκτηση πληροφοριών. Στα κατανεμημένα συστήματα, όπου γίνεται ανταλλαγή μηνυμάτων μεταξύ επεξεργαστών, κάποιοι αλγόριθμοι χρησιμοποιούν την μέθοδο DCMST για τον περιορισμό του αριθμού των μηνυμάτων και την ελαχιστοποίηση του κόστους του δικτύου. Η μέθοδος DCMST είναι, επίσης, χρήσιμη για την ανάκτηση πληροφοριών όταν χρησιμοποιούνται μεγάλες δομές δεδομένων, οι οποίες λέγονται bitmaps, στην συμπίεση μεγάλων αρχείων. Η μέθοδος βρίσκει επίσης εφαρμογή στα δίκτυα οπτικών επικοινωνιών, όπου είναι επιθυμητό να χρησιμοποιήσουμε ένα ζευγνύον δέντρο μικρής διαμέτρου για κάθε εκπομπή προς πολλούς αποδέκτες, προκειμένου να ελαχιστοποιήσουμε τις παρεμβολές στο δίκτυο.

Στην βιβλιογραφία αναφέρονται αλγόριθμοι για επίλυση του προβλήματος DCMST (π.χ. Achuthan, et al, 1994, “Computational methods for the diameter restricted minimum weight spanning tree problem”, Australas. J. Combin., 10, 51-71), οι οποίοι πάντως δεν είναι πρακτικοί στην περίπτωση γράφων με χιλιάδες κόμβους. Υπάρχει, επίσης, σημαντική βιβλιογραφία, η οποία αφορά την διάμετρο ενός τυχόντος δέντρου. Έχει αποδειχτεί (Szekers, 1983, “Distribution of labeled trees by diameter”, Lecture Notes in Math., 1036, 392-397) ότι για ένα τυχαίο δέντρο με n αριθμημένες κορυφές (labeled-tree) και καθώς το n τείνει στο άπειρο, η αναμενομένη τιμή της διαμέτρου είναι 3.342171n1/2, η δε διάμετρος με την μέγιστη πιθανότητα έχει τιμή 3.2015131n1/2.

3.11 Αναμενόμενη τιμή της διαμέτρου ενός MST

Σε ένα πλήρη γράφο με τυχαία βάρη ακμών, κάθε ζευγνύον δέντρο (ST) είναι εξ ίσου πιθανό να αποτελεί ελάχιστο ζευγνύον δέντρο (MST). Έτσι, υπάρχει αντιστοίχιση ένα προς ένα μεταξύ του συνόλου των πιθανών MST ενός πλήρους γράφου n κορυφών με τυχαία βάρη ακμών και του συνόλου των n^{n-2} δέντρων αριθμημένων κορυφών με n κόμβους. Μετά από τα παραπάνω, καταλαβαίνουμε ότι η συμπεριφορά της διαμέτρου του MST σε ένα πλήρη γράφο με τυχαία βάρη ακμών μπορεί να μελετηθεί χρησιμοποιώντας τυχαία δέντρα αριθμημένων κορυφών χωρίς βάρη ακμών.

$$t_n(h) = \sum_{m_1+m_2+\ldots+m_h=n-1} \frac{(n-1)!}{m_1!m_2!\ldots m_h!} m_1^{m_2} m_2^{m_3} \ldots m_{h-1}^{m_h} \quad \textbf{(1)}$$

όπου $1 \le h \le (n-2)$, $0 \le m_i \le (n-1)$, $1 \le i \le h$ και $t_n(1) = 0^0 = 1$

Είναι προφανές ότι $t_n(h) = n^{n-2}$ για $h \ge n-1$. Στη συνέχεια, οι τιμές του $t_n(h)$ αντικαθίστανται στην ακόλουθη έκφραση:

$$G_h(x) = \sum_{n=1}^{\infty} \frac{t_n(h)}{(n-1)!} x^n, \quad \text{με } h \ge 0 \quad \textbf{(2)}$$

Τώρα, το πλήθος των δέντρων αριθμημένων κορυφών με n κόμβους και ύψος ακριβώς ίσο με h δίνεται από τον τύπο (3):

$$H_h(x) = G_h(x) - G_{h-1}(x) = \sum_{n=1}^{\infty} \frac{t_n(h) - t_n(h-1)}{(n-1)!} x^n, \quad h \ge 1 \quad \textbf{(3)}$$

Ο αριθμός $\mathbf{D_d(x)}$ των δέντρων με διάμετρο d δίνεται για τις περιττές και τις άρτιες τιμές του d από τους παρακάτω τύπους:

$$D_{2h+1}(x) = \frac{1}{2} H_h^2(x), \quad \text{με } h \ge 0 \quad \textbf{(4)}$$

$$D_{2h}(x) = H_h(x) - H_{h-1}(x)\, G_{h-1}(x), \quad \text{με } h \ge 1 \quad (5)$$

Μετά τα παραπάνω, υπολογίζουμε τον αριθμό *δn(d)* των δέντρων αριθμημένων κορυφών με n κόμβους και διάμετρο d, συγκρίνοντας όρους από τις εξισώσεις (4) και (5) με την ακόλουθη εξίσωση:

$$D_d(x) = \sum_{n=1}^{\infty} \frac{\delta_n(d)}{n!} x^n, \qquad \text{με } d \geq 1 \qquad (6)$$

Τελικά, το $\bar{d}_n$, το οποίο εκφράζει την αναμενομένη τιμή της διαμέτρου για ένα δέντρο αριθμημένων κορυφών με n κόμβους, δίνεται από την παρακάτω σχέση:

$$\bar{d}_n = \left(\sum_{d=1}^{n} d \cdot \delta_n(d) \right) \cdot n^{2-n}, \qquad \text{με } n \geq 3 \qquad (7)$$

Παρατηρούμε από τα παραπάνω, ότι η εξίσωση (1) χρειάζεται τον υπολογισμό και την πρόσθεση *(n + h – 2)!/(n – 1)! (h – 1)!* όρων για κάποιες δεδομένες τιμές των n και h. Ο υπολογισμός αυτός είναι πολύ χρονοβόρος και περίπλοκος, γεγονός που κάνει την μέθοδο πολύ αργή, κυρίως όταν κάποιος πρέπει να υπολογίσει την ακριβή μέση διάμετρο για γράφους με χιλιάδες κόμβους .

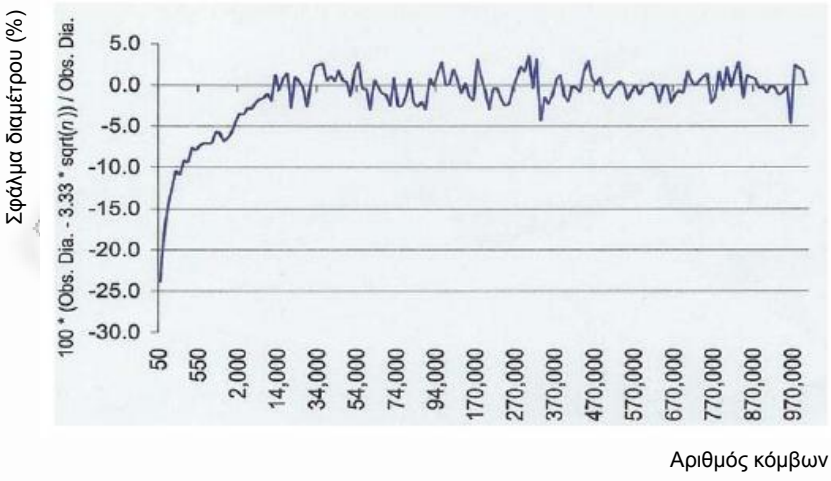
Κεφάλαιο 4

Κώδικας

4.1 Αναπτυχθέντες αλγόριθμοι – Γενικά στοιχεία

Στην εργασία που ακολουθεί αναπτύχθηκαν αλγόριθμοι χειρισμού γράφων στην γλώσσα προγραμματισμού C. Εξετάστηκαν γράφοι με 4, 5 και 6 κόμβους. Η εξέταση περισσοτέρων κόμβων είναι προφανώς δυνατή με τη χρήση των αναπτυχθέντων αλγορίθμων και την τροποποίηση σε κάποια από τα σημεία τους. Εφαρμόστηκε ο παραπάνω τύπος 7 και μετρήθηκαν τα δέντρα με διαμέτρους 2 και 3, ο αριθμός των οποίων προέκυψε από το πρόγραμμα και τους αλγορίθμους που αναπτύξαμε.

Σε κάθε περίπτωση η τιμή της διαμέτρου των δέντρων βρέθηκε ίση προς 3.25. Σύμφωνα με όσα προαναφέρθηκαν, η τιμή αυτή αναμένεται να είναι περίπου ίση προς 6.67, όμως η σταθεροποίηση στην τιμή αυτή παρουσιάζεται σε γράφους με πολύ μεγάλο αριθμό κόμβων, όπως για παράδειγμα για αριθμό κόμβων μεγαλύτερο από 1100 [27]. Σε μικρότερους αριθμούς κόμβων, ξεκινώντας μάλιστα από τους 50, η αναμενομένη τιμή της διαμέτρου εμφανίζει διακυμάνσεις και απόκλιση μεγαλύτερη από 30% (στους 50 κόμβους) από την πιο πάνω αναφερομένη τιμή, όπως φαίνεται και από το πιο κάτω πίνακα :



Πιν. 1 Διάμετρος Δέντρων

4.2 Σχόλια κώδικα

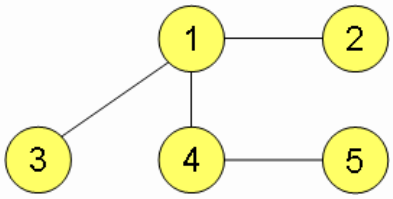
4.2.1 Σχόλιο 1

Η περιγραφή ενός γράφου γίνεται ακολουθώντας τη μέθοδο του πίνακα γειτονικών κορυφών. Όπως προαναφέραμε, η μελέτη των γράφων είναι ισοδύναμη με την μελέτη δέντρων αριθμημένων κορυφών χωρίς βάρη ακμών. Έτσι, ένας γράφος N κορυφών παριστάνεται με την χρήση ενός τετραγωνικού πίνακα ακεραίων N x N. Στον κώδικα που ακολουθεί είναι **ο πίνακας tree**.

Ο tree αρχικά έχει παντού τιμή 0. Προκειμένου, όμως, να περιγράψει ένα δέντρο, πρέπει να περιέχει κατάλληλο αριθμό 1 και μάλιστα σε τέτοιες θέσεις, ώστε να μη σχηματίζονται κύκλοι στον γράφο μας, αφού ένα δέντρο είναι εξ ορισμού μη κυκλικός γράφος. Εάν η χρήση του πίνακα ήταν για την περιγραφή ενός γράφου, τότε ο μέγιστος δυνατός αριθμός των “1” στον πίνακα θα ήταν N(N-1)/2, αφού αυτός είναι και ο μέγιστος αριθμός ακμών για ένα μη κατευθυνόμενο γράφο με N κορυφές.

Για παράδειγμα, έστω ότι διαθέτουμε ένα γράφο 5 κορυφών όπου ο πίνακα στο (σχ.18) παριστάνει το διπλανό του δέντρο και ο πίνακας είναι προφανώς συμμετρικός.

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	0
3	1	0	0	0	0
4	1	0	0	0	1
5	0	0	0	1	0



Σχ. 18 Παράδειγμα πίνακα - δέντρου

Με τον πίνακα tree σχετίζονται οι συναρτήσεις:

- ✓ **tree_init( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα tree.
- ✓ **tree_display( ) :** εμφανίζει στην οθόνη τα περιεχόμενα του πίνακα tree.

4.2.2 Σχόλιο 2

Το πρόγραμμα αρχικά δημιουργεί γράφους. Κάθε γράφος παριστάνεται με τον πίνακα γειτονικών κορυφών tree. Η δημιουργία των γράφων γίνεται με την τοποθέτηση στον πίνακα όλων των δυνατών συνδυασμών “0” και “1”, από τους οποίους θα αποκλειστούν τελικά οι μη έγκυροι. Λαμβάνοντας υπόψη και τη συμμετρικότητα του πίνακα tree, οι δυνατοί συνδυασμοί ανάλογα με το μέγεθος του πίνακα είναι:

Από	0	έως	3F	για πίνακα	4x4
Από	0	έως	3FF	για πίνακα	5x5
Από	0	έως	7FFF	για πίνακα	6x6
Από	0	έως	1FFFFF	για πίνακα	7x7
Από	0	έως	FFFFFFF	για πίνακα	8x8

Η συνάρτηση **hex_to_bin( )** μετατρέπει σε δυαδικό τον δεκαεξαδικό αριθμό που μας δείχνει τον συνδυασμό “0” και “1” τον οποίο θα καταχωρήσουμε στον πίνακα tree. Το αποτέλεσμα καταχωρείται σε ένα μονοδιάστατο πίνακα ακεραίων N*(N-1)/2 θέσεων που λέγεται **mono**, με τον οποίο σχετίζονται οι συναρτήσεις:

- ✓ **init_mono( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα mono.
- ✓ **mono_display( ) :** εμφανίζει στην οθόνη τα περιεχόμενα του πίνακα mono.

Η συνάρτηση **mono_to_square( )** τοποθετεί τον δυαδικό αριθμό που υπάρχει στον πίνακα mono στις θέσεις του πίνακα tree ολοκληρώνοντας την κατ’ αρχήν οργάνωση του τετραγωνικού πίνακα.

4.2.3 Σχόλιο 3

Στο πρόγραμμα χρησιμοποιείται ένας πίνακας δύο διαστάσεων, ο **path**. Για ένα γράφο N κόμβων, ο πίνακας path έχει N(N-1)/2 γραμμές και 3 στήλες. Αυτός ο πίνακας «μονοπατιών» χρησιμοποιείται, όπως θα φανεί παρακάτω, για να δείξει σε κάθε στιγμή τις θέσεις του πίνακα tree στις οποίες υπάρχουν ήδη “1”. Το τμήμα του tree, που εξετάζεται, είναι το πάνω από την κύρια διαγώνιο. Με την χρήση του path γίνεται δυνατό κάθε καινούργιος κόμβος να συνδεθεί στον ήδη υπάρχοντα γράφο σε τέτοια θέση, ώστε να μη κλείνει κύκλος. Υπενθυμίζουμε ότι οι γράφοι μας σχηματίζονται από όλους τους έγκυρους δυνατούς συνδυασμούς, άρα κατά τη δημιουργία πρέπει να αποκλείονται οι μη έγκυροι.

Οι δύο πρώτες στήλες του path περιέχουν την γραμμή και την στήλη του πίνακα tree, στις οποίες υπάρχει “1”. Στην τρίτη στήλη του path γράφεται το πλήθος των εμφανίσεων κάθε γραμμής στον path. Ο λόγος αυτής της αναγραφής θα φανεί στη συνέχεια. Το πλήθος των εμφανίσεων κάθε γραμμής στον path γράφεται στην τρίτη στήλη, μόνο όμως για την πρώτη εμφάνιση της γραμμής.

Έτσι, για παράδειγμα, ο πίνακας path που αντιστοιχεί στον πίνακα tree του (σχ. 18) είναι ο πίνακας του σχ. 19α, από τον οποίο μας «χρειάζονται», είναι δηλαδή εκμεταλλεύσιμες οι 4 πρώτες γραμμές:

1	2	3
1	3	0
1	4	0
4	5	1
0	0	6
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Σχ. 19α Παράδειγμα πίνακα path

5	4	1
4	1	1
3	1	1
2	1	1
0	0	6
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Σχ. 19β Παράδειγμα πίνακας pathrev

Στο πρόγραμμα χρησιμοποιείται, επίσης, ένας πίνακας δύο διαστάσεων, ο **pathrev**. Ο πίνακας αυτός είναι αντίστοιχος του path, αλλά για το κάτω από την διαγώνιο μέρος του tree. Για ένα γράφο Ν κόμβων, ο πίνακας pathrev, όπως και ο path, έχει N(N-1)/2 γραμμές και 3 στήλες. Ο pathrev, για το παράδειγμα που προαναφέρθηκε, φαίνεται στο σχ. 19β.

Με τους πίνακες path και pathrev σχετίζονται οι συναρτήσεις:

- ✓ **path_init ( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα path.
- ✓ **pathrev_init ( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα pathrev.
- ✓ **paths_display ( ) :** εμφανίζει στην οθόνη τα περιεχόμενα των πινάκων path και pathrev.
- ✓ **path_construction ( ) :** δημιουργεί τον πίνακα path.
- ✓ **pathrev_construction ( ) :** δημιουργεί τον πίνακα pathrev.

4.2.4 Σχόλιο 4

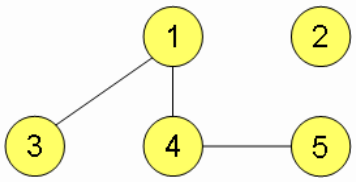
Η συνάρτηση **count_ones( )** μετράει το πλήθος των “1” στον πίνακα tree και επιστρέφει αυτό το πλήθος που μέτρησε. Η τιμή που επιστρέφει η συνάρτηση χρησιμοποιείται στην διαπίστωση του έγκυρου ή όχι γράφου, αφού ένας έγκυρος γράφος Ν κόμβων πρέπει να έχει ακριβώς Ν-1 ακμές. Ένα, λοιπόν, από τα κριτήρια της εγκυρότητας του γράφου είναι το πλήθος των ακμών, άρα τελικά το πλήθος των “1” στον πίνακα tree.

4.2.5 Σχόλιο 5

Το πρόγραμμα χρησιμοποιεί την συνάρτηση **one_isolated( )**. Η συνάρτηση ελέγχει την ύπαρξη μεμονωμένου κόμβου στον γράφο, κόμβος δηλαδή ο οποίος δεν ενώνεται με τους υπόλοιπους. Αυτό είναι ισοδύναμο με την ύπαρξη μιας γραμμής στον πίνακα tree, στην οποία δεν υπάρχουν καθόλου “1”, μιας γραμμής, δηλαδή με το άθροισμα των στοιχείων της να ισούται με μηδέν. Η συνάρτηση επιστρέφει τιμή “1” εάν υπάρχει μεμονωμένος κόμβος στον γράφο και τιμή “0” εάν τέτοιος κόμβος δεν υπάρχει. Για παράδειγμα, ο πίνακας γειτονικών κορυφών του σχ. 20α είναι αυτός που αντιστοιχεί στον γράφο του σχ. 20β.

	1	**2**	**3**	**4**	**5**
1	0	0	1	1	0
2	0	0	0	0	0
3	1	0	0	0	0
4	1	0	0	0	1
5	0	0	0	1	0

Σχ. 20α Παράδειγμα πίνακα path



Σχ. 20β Παράδειγμα πίνακας pathrev

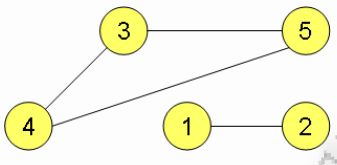
4.2.6 Σχόλιο 6

Το πρόγραμμα χρησιμοποιεί την συνάρτηση **two_isolated( )**. Η συνάρτηση ελέγχει κατά πόσον υπάρχει μεμονωμένο ζεύγος κόμβων στον γράφο, ζεύγος

δηλαδή το οποίο δεν ενώνεται με τους υπόλοιπους. Για παράδειγμα, ο πίνακας γειτονικών κορυφών (σχ.21α) είναι αυτός που αντιστοιχεί στο γράφο (σχ.21β), όπου υπάρχει ένα μεμονωμένο ζεύγος.

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	0	0	0
3	0	0	0	1	1
4	0	0	1	0	1
5	0	0	1	1	0

Σχ. 21α Πίνακας γειτονικών κορυφών



Σχ. 21β Γράφος

1	2	1
3	4	2
3	5	0
4	5	1
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Σχ. 21γ Ο πίνακας path του γράφου

Η ύπαρξη μεμονωμένου ζεύγους στον γράφο σημαίνει δύο κόμβους, οι οποίοι συνδέονται μόνο μεταξύ τους και κανείς από τους δύο δεν συνδέεται με κάποιον άλλο. Ο πίνακας path (σχόλιο 3) μας δείχνει ακριβώς αυτές τις υπάρχουσες συνδέσεις, για τις οποίες ο έλεγχος του πίνακα γίνεται από την συνάρτηση two_isolated( ). Το ζευγάρι των κόμβων είναι μεμονωμένο, εάν ο αριθμός μιας γραμμής και ο αριθμός μιας στήλης του tree εμφανίζονται μόνο μια φορά στις δυο πρώτες στήλες του πίνακα path. Στην περίπτωση αυτή η συνάρτηση επιστρέφει τιμή 2, ενώ εάν δεν υπάρχει μεμονωμένο ζεύγος η συνάρτηση επιστρέφει τιμή 0.

Στον παραπάνω πίνακα path (σχ.21γ), για παράδειγμα με τον αντίστοιχο γράφο (σχ.21β), παρατηρούμε ότι το 1 και το 2 εμφανίζονται μόνο μια φορά στις δύο πρώτες στήλες του πίνακα. Το 4 και το 5 εμφανίζονται επίσης μια μόνο φορά, όμως το 3, με το οποίο αποτελούν ζεύγη, εμφανίζεται περισσότερες από μια φορές, άρα τα ζεύγη (3,4) και (3,5) δεν είναι μεμονωμένα.

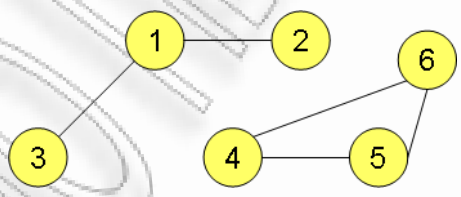
Η συνάρτηση έχει ιδιαίτερη χρησιμότητα σε αντίστοιχες με αυτή περιπτώσεις (σχ.21β). Στο σχήμα αυτό παρατηρούμε ότι εκτός από το μεμονωμένο ζεύγος, ο υπόλοιπος γράφος είναι κλειστός. Εάν δεν υπήρχε η συνάρτηση two_isolated( ), ο γράφος θα εθεωρείτο κανονικός, αφού έχει τον κατάλληλο αριθμό ακμών.

4.2.7 Σχόλιο 7

Το πρόγραμμα χρησιμοποιεί, επίσης, την συνάρτηση **three_isolated( )**. Η συνάρτηση ελέγχει κατά πόσον υπάρχει μεμονωμένη τριάδα κόμβων στον γράφο, τριάδα δηλαδή η οποία δεν ενώνεται με τους υπόλοιπους κόμβους. Αυτό έχει νόημα να διερευνηθεί μόνο σε γράφο με τουλάχιστον 6 κόμβους. Για παράδειγμα, ο πίνακας γειτονικών κορυφών (σχ.22α) είναι αυτός που αντιστοιχεί στο διπλανό γράφο του (σχ.22β).

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	0	0	0
3	1	0	0	0	0	0
4	0	0	0	0	1	1
5	0	0	0	1	0	1
6	0	0	0	1	1	0

Σχ. 22α Πίνακας γειτονικών κορυφών



Σχ. 22β Γράφος

Η συνάρτηση έχει ιδιαίτερη χρησιμότητα στην παραπάνω περίπτωση γράφου, ο οποίος διατηρεί τον κατάλληλο αριθμό ακμών, με ένα μέρος του να αποτελεί κλειστό κυκλικό γράφο (σχόλιο 6). Ο πίνακας path του πιο πάνω γράφου είναι αυτός του σχ. 22(γ). Η συνάρτηση ανιχνεύει το κατά πόσον υπάρχει μια μεμονωμένη τριάδα σύμφωνα με τον εξής αλγόριθμο:

Εάν μια γραμμή εμφανίζεται δύο μόνο φορές στον πίνακα path, τότε ο αύξων αριθμός κάθε στήλης, με τις οποίες η γραμμή αποτελεί ζεύγος, πρέπει να εμφανίζεται μία μόνο φορά στον πίνακα path είτε ως γραμμή είτε ως στήλη.

1	2	2
1	3	0
4	5	2
4	6	0
5	6	1
0	0	10
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Σχ. 22γ Πίνακας path

6	5	2
6	4	0
5	4	1
3	1	1
2	1	1
0	0	10
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Σχ. 22δ Πίνακας pathrev

Για τον γράφο (σχ. 19β) αντίστοιχος με τον path είναι και ο πίνακας pathrev (σχ. 22δ). Στον πίνακα αυτόν τοποθετούνται με την αντίστοιχη λογική που γίνεται για τον path τα “1” του πίνακα tree, τα οποία βρίσκονται κάτω από την κύρια διαγώνιο. Αυτό γίνεται συμπληρωματικά προς τον πίνακα path, ώστε να προβλεφθεί κάθε δυνατή περίπτωση πολλαπλής εμφάνισης οποιασδήποτε γραμμής. Είναι προφανές ότι και στον πίνακα pathrev εάν μια γραμμή εμφανίζεται δύο μόνο φορές, τότε ο αύξων αριθμός κάθε στήλης, με τις οποίες η γραμμή αποτελεί ζεύγος, πρέπει να εμφανίζεται μία μόνο φορά.

4.2.8 Σχόλιο 8

Η συνάρτηση **valid_tree ( )** διαπιστώνει εάν έχουμε έγκυρο δέντρο (που έχει προκύψει από τον γράφο που εξετάζουμε). Εάν το δέντρο είναι έγκυρο, η συνάρτηση επιστρέφει τιμή μηδέν. Λόγοι ακυρότητας του δέντρου μπορεί να είναι:

- ✓ Σε κάθε περίπτωση, ο μη σωστός αριθμός ακμών. Εάν το δέντρο έχει Ν κόμβους, ο αριθμός των ακμών πρέπει να είναι Ν-1. Στην περίπτωση αυτή η συνάρτηση επιστρέφει τιμή 1.

- ✓ Στην περίπτωση των 5 κόμβων, εκτός από την παραπάνω περίπτωση, λόγο ακυρότητας αποτελεί και η ύπαρξη μεμονωμένου κόμβου ή μεμονωμένου ζεύγους κόμβων. Στην περίπτωση αυτή η συνάρτηση επιστρέφει τιμή 5.

- ✓ Στην περίπτωση των 6 κόμβων, εκτός από τον μη σωστό αριθμό ακμών, λόγο ακυρότητας αποτελεί και η ύπαρξη μεμονωμένου κόμβου ή μεμονωμένου ζεύγους κόμβων ή μεμονωμένης τριάδας κόμβων. Στην περίπτωση αυτή η συνάρτηση επιστρέφει τιμή 6.

4.2.9 Σχόλιο 9

Η συνάρτηση **tree_mult_tree ( )** δίνει τιμές σε τρεις πίνακες ακεραίων NxN, τους **tree_2**, **tree_3** και **tree_4**. Αν ονομάσουμε Α τον πίνακα tree, τότε ο tree_2 είναι ο A*A, ο tree_3 είναι ο A*A*A και ο tree_4 είναι ο A*A*A*A.. Εφ' όσον ο πίνακας Α (ο tree) περιέχει τις συνδέσεις κάθε κόμβου με άλλον, ο πίνακας A*A περιέχει τις συνδέσεις, τα μονοπάτια δηλαδή μήκους 2. Ομοίως, ο A*A*A περιέχει τα μονοπάτια μήκους 3 και ο A*A*A*A τα μονοπάτια μήκους 4.

4.2.10 Σχόλιο 10

Η συνάρτηση **tree_m_display( )** δέχεται ως όρισμα έναν ακέραιο. Ανάλογα με την τιμή αυτού του ακεραίου, η συνάρτηση εμφανίζει στην οθόνη τα περιεχόμενα των πινάκων tree_2, tree_3 και tree_4. Έτσι, αν το όρισμα είναι 2 εμφανίζονται τα περιεχόμενα του tree_2, αν είναι 3 εμφανίζονται τα περιεχόμενα του tree_3, ενώ αν είναι 4 εμφανίζονται τα περιεχόμενα του tree_4.

4.2.11 Σχόλιο 11

Οι πίνακες tree_2, tree_3 και tree_4 πρέπει να ελεγχθούν ως προς το εάν σε κάποιες θέσεις τους έχουν τιμή διάφορη του μηδενός, άρα ύπαρξη αντίστοιχου μονοπατιού (σχόλιο 9) ή μηδέν. Εν προκειμένω, δεν μας ενδιαφέρει η ακριβής τιμή που υπάρχει σε κάθε πίνακα, αλλά μόνο εάν η τιμή είναι μηδέν ή διάφορη του μηδενός. Για λόγους βελτιστοποίησης του προγραμματισμού χρησιμοποιείται η συνάρτηση **tree_modify( )**, η οποία μετατρέπει τα μη μηδενικά περιεχόμενα του κάθε πίνακα σε "1".

4.2.12 Σχόλιο 12

Από το πρόγραμμα θα καταλήξουμε σε μέτρηση του πλήθους των δέντρων τα οποία έχουν μια συγκεκριμένη διάμετρο. Για παράδειγμα, θα μετρήσουμε τα δέντρα που έχουν διάμετρο 2, δηλαδή το πλήθος των δέντρων στα οποία έχουμε σύνδεση κόμβων με μονοπάτι δυο ακμών, αλλά ακριβώς και μόνο δύο ακμών. Η διαπίστωση αυτού του «ακριβώς», γίνεται στο πρόγραμμα με την χρήση της συνάρτησης **compare_trees( )**.

Η συνάρτηση επιστρέφει τιμή ‘‘0’’ εάν υπάρχει δέντρο με διάμετρο ακριβώς την ζητούμενη. Αυτό, στην περίπτωση των δέντρων διαμέτρου 2 που αναφέραμε ως παράδειγμα, σημαίνει ένα από τα εξής:

- ✓ Ότι το δέντρο tree_3 που προκύπτει από το αρχικό έχει σε όλες τις θέσεις του γράφου που το περιγράφει τιμή μηδέν. Αυτό διαπιστώνεται από την τιμή επιστροφής της συνάρτησης **tree_total_sum( )**, η οποία είναι μηδενική ή διάφορη του μηδενός
- ✓ Ότι, το δέντρο tree_3 που προκύπτει από το αρχικό δεν έχει σε όλες τις θέσεις του γράφου που το περιγράφει τιμή μηδέν, αλλά ταυτίζεται με το δέντρο tree_2 ή με το δέντρο tree και φυσικά το tree_2 δεν ταυτίζεται με το αρχικό.

4.3 Κώδικας σε C

```
#include <stdio.h>
#include <conio.h>

#define N 5
#define DIM N*(N-1)/2
#define NUM 0x3ff

int tree[N][N];
int tree_2[N][N];
int tree_3[N][N];
int tree_4[N][N];
int mono[DIM];
int path[DIM][3];
```

```
int pathrev[DIM][3];

void tree_init(void);
void tree_display(void);
void tree_m_display(int);
void mono_display(void);
void init_mono(void);
void path_init(void);
void pathrev_init(void);
void hex_to_bin(long);
void mono_to_square(void);
int count_ones(void);
void path_construction(void);
void pathrev_construction(void);
void paths_display(void);
int one_isolated(void);
int two_isolated(void);
int three_isolated(void);
int valid_tree(void);
void tree_mult_tree(void);
int tree_total_sum(void);
void tree_modify(void);
int compare_trees (int);

void main(void)
{
	long k, plithos=0;
	int valid, diametros=0;
//	for (k=0x04b; k<=0x04b; k++)
//	for (k=0x6007; k<=0x6007; k++)
	for (k=0x0; k<=NUM; k++)
	{
		tree_init();
		init_mono();
		path_init();
		pathrev_init();
		hex_to_bin(k);
//		mono_display();
		mono_to_square();
		path_construction();
		pathrev_construction();
//		paths_display();
```

```
        valid = valid_tree();
                if (valid == 0)
                {
//                      printf("%10lx", k);
//                      tree_display();
                      tree_mult_tree();
//                    tree_m_display(2);
//                    tree_m_display(3);
//                    tree_m_display(4);
                      tree_modify();
                      if (compare_trees(2) == 0)
                      {
                            diametros++;
//                          getch();
//                          tree_m_display(2);
//                          tree_m_display(3);
//                          tree_m_display(4);
//                          tree_display();
                       }
//                     paths_display();
                       plithos++;
}
        }
        printf ("\nΠλήθος Δέντρων=%8ld\n", plithos);
        printf ("Πλήθος Δέντρων Διαμέτρου 2 = %d\n", diametros);
}
void tree_init(void) {
        int j, k;

        for (j=0; j<N; j++)
            for (k=0; k<N; k++)
                tree[j][k] = 0;
}
void tree_m_display(int deg)
{
        int j, k;
        char ch;

        printf("\nPINAKAS tree_%1d\n", deg);

        switch (deg) {
        case 2:
```

```
                for (j=0; j<N; j++) {
                    for (k=0; k<N; k++)
                        printf("%3d", tree_2[j][k]);
                    printf("\n");
                }
                break;
        case 3:
                for (j=0; j<N; j++) {
                    for (k=0; k<N; k++)
                        printf("%3d", tree_3[j][k]);
                    printf("\n");
                }
                break;
        case 4:
                for (j=0; j<N; j++) {
                    for (k=0; k<N; k++)
                        printf("%3d", tree_4[j][k]);
                    printf("\n");
                }
                break;
        }
        printf("\nGIA SYNEXEIA DOSE XARAKTIRA\n");
        ch = getch();
}
void tree_display(void)
{
        int j, k;
        char ch;
        printf("\nPINAKAS tree\n");
        for (j=0; j<N; j++)
        {
            for (k=0; k<N; k++)
                printf("%3d", tree[j][k]);
            printf("\n");
        }
        printf("\nGIA SYNEXEIA DOSE XARAKTIRA\n");
        ch = getch();
}
void init_mono(void)
{
        int j;
        for (j=0; j<DIM-1; j++)
```

```
            mono[j] = 0;
}
void path_init(void)
{
        int j, k;
        for (j=0; j<DIM; j++)
             for (k=0; k<3; k++)
                  path[j][k] = 0;
}
void pathrev_init(void)
{
        int j, k;
           for (j=0; j<DIM; j++)
                for (k=0; k<3; k++)
                     pathrev[j][k] = 0;
}
void hex_to_bin(long num)
{
        int k = DIM-1;
        int pil=1, yp;
        while (pil != 0)
        {
                pil = num/2;
                yp = num%2;
                num = pil;
                mono[k--] = yp;
        }
}
void mono_display(void)
{
         int j;
         char ch;
         printf("\nPINAKAS mono\n");
         for (j=0; j<DIM; j++)
              printf("%2d", mono[j]);
         printf("\n\n");
         printf("\nGIA SYNEXEIA DOSE XARAKTIRA\n");
         ch = getch();
}
void mono_to_square(void)
{
        int k=N-1, j, i=0, m=1;
```

```
        while (k>0)
        {
                for (j=m; j<N; j++)
                    tree[N-k-1][j] = mono[i++];
                k--;
                m++;
        }
        for (j=0; j<N; j++)
            for (k=0; k<j; k++)
                tree[j][k] = tree[k][j];
}
int count_ones(void)
{
        int count=0, j, k;
        for (j=0; j<N; j++)
        {
            for (k=j+1; k<N; k++)
            {
                if (tree[j][k]==1)
                  count++;
            }
        }
        return count;
}
void path_construction(void)
{
        int j, k, pr=0, numr, count=0;
        for (j=0; j<N; j++) {
            for (k=j+1; k<N; k++) {
            {    if (tree[j][k]==1)
                 {
                    while (path[pr][0] != 0)
                            pr++;
                    path[pr][0] = j+1;
                    path[pr][1] = k+1;
                 }
            }
        {
        for (j=0; j<DIM; j++) {
            numr = path[j][0];
            for (k=0; k<DIM; k++)
            {
```

```
                if (path[k][0] == numr)
                  count++;
              }
              path[j][2] = count;
              j = j + count - 1;
              count = 0;
        }
}



void pathrev_construction(void)
{
        int j, k, pr=0, numr, count=0;
        for (j=N-1; j>=0; j--)
        {
             for (k=j; k>=0; k--)
             {
                  if (tree[j][k]==1)
                  {
                    while (pathrev[pr][0] != 0)
                            pr++;
                    pathrev[pr][0] = j+1;
                    pathrev[pr][1] = k+1;
                  }
             }
        }
        for (j=0; j<DIM; j++)
        {
             numr = pathrev[j][0];
             for (k=0; k<DIM; k++)
             {
                  if (pathrev[k][0] == numr)
                    count++;
             }
             pathrev[j][2] = count;
             j = j + count - 1;
             count = 0;
        }
}
void paths_display(void)
{
```

```
        int j, k;
        char ch;
        printf("\nPINAKAS path\tPINAKAS pathrev\n");
        for (j=0; j<DIM; j++)
       {
        if (path[j][0]==0)
          break;
          for (k=0; k<3; k++)
               printf("%3d", path[j][k]);
          printf(" ");
          for (k=0; k<3; k++)
                printf("%3d", pathrev[j][k]);
        printf("\n");
        }
        printf("\n");
        printf("\nGIA SYNEXEIA DOSE XARAKTIRA\n");
        ch = getch();
}




int one_isolated(void)
{
        int j, k, sum=0;
        for (j=0; j<N; j++)
        {
            for (k=0; k<N; k++)
                 sum += tree[j][k];
            if (sum == 0)
                return 1;
            else
                sum = 0;
        }
        return 0;
}
int two_isolated(void)
{
        int j, k, num0, num1;
        int flag0, flag1;
```

```
        for (j=0; j<DIM; j++)
        {
            flag0=0;
            flag1=0;
            num0 = path[j][0];
            num1 = path[j][1];
            for (k=0; k<DIM; k++)
            {
                if (num0 == path[k][0])
                  flag0++;
                if (num0 == path[k][1])
                  flag0++;
                if (num1 == path[k][0])
                  flag1++;
                if (num1 == path[k][1])
                  flag1++;
            }
            if (flag0 == 1 && flag1 == 1)
              return 2;
        }
        return 0;
}

int three_isolated(void)
{
        int j, numc1, numc2;
        int flag0=0, flag1=0, flagr=0, m;
        for (j=0; j<DIM; j++)
        {
            if (path[j][2] == 2)
            {
              for (m=0; m<DIM; m++)
              {
                  if (path[j][0] == path[m][1])
                  {
                    flagr = 1;
                    break;
                  }
              }
              if (flagr == 0)
              {
                numc1 = path[j++][1];
```

```
            numc2 = path[j][1];
            for (m=0; m<DIM; m++)
            {
                if (numc1 == path[m][0])
                  flag0++;
                if (numc1 == path[m][1])
                  flag0++;
                if (numc2 == path[m][0])
                  flag1++;
                if (numc2 == path[m][1])
                  flag1++;
            }
            if (flag0 == 1 && flag1 == 1)
               return 3;
            flag0 = 0;
            flag1 = 0;
            flagr = 0;
        }
    }
}
flagr = 0;
for (j=0; j<DIM; j++)
{
    if (pathrev[j][2] == 2)
    {
      for (m=0; m<DIM; m++)
      {
          if (pathrev[j][0] == pathrev[m][1])
          {
            flagr = 1;
            break;
          }
      }
      if (flagr == 0)
      {
        numc1 = pathrev[j++][1];
        numc2 = pathrev[j][1];
        for (m=0; m<DIM; m++)
        {
            if (numc1 == pathrev[m][0])
              flag0++;
            if (numc1 == pathrev[m][1])
```

```
                    flag0++;
                  if (numc2 == pathrev[m][0])
                    flag1++;
                  if (numc2 == pathrev[m][1])
                    flag1++;
                }
                if (flag0 == 1 && flag1 == 1)
                  return 3;
                flag0 = 0;
                flag1 = 0;
                flagr = 0;
              }
            }
          }
          return 0;
}



int valid_tree(void)
{
        if (count_ones() != (N-1))
            return 1;
        switch (N)
        {
        case 5:
                  if (one_isolated() || two_isolated())
                    return 5;
                  break;
        case 6:
                  if (one_isolated() || two_isolated() || three_isolated())
                    return 6;
                  break;
        }
        return 0;
}
void tree_mult_tree(void)
/* Creates the trees: tree_2, tree_3 and tree_4 */
{
        int j, k, m;
        for (j=0; j<N; j++)
        {
```

```
            for (k=0; k<N; k++)
            {
                tree_2[j][k] = 0;
                tree_3[j][k] = 0;
                tree_4[j][k] = 0;
            }
        }
        for (j=0; j<N; j++)
        {
            for (k=0; k<N; k++)
            {
                for (m=0; m<N; m++)
                    tree_2[j][k] += tree[j][m] * tree[m][k];
            }
        }
        for (j=0; j<N; j++)
        {
            for (k=0; k<N; k++)
            {
                for (m=0; m<N; m++)
                    tree_3[j][k] += tree_2[j][m] * tree[m][k];
            }
        }
        for (j=0; j<N; j++)
        {
            for (k=0; k<N; k++)
            {
                for (m=0; m<N; m++)
                    tree_4[j][k] += tree_3[j][m] * tree[m][k];
            }
        }
}


void tree_modify(void)
/* Changes trees' contents to 0 and 1 */
{
        int j, k;
        for (j=0; j<N; j++)
```

```
        {
            for (k=0; k<N; k++)
            {
                if (tree_2[j][k] != 0)
                  tree_2[j][k] = 1;
                if (tree_3[j][k] != 0)
                  tree_3[j][k] = 1;
                if (tree_4[j][k] != 0)
                  tree_4[j][k] = 1;
            }
        }
}
int tree_total_sum(int tr)
/* Checks whether the tree contains only 0's */
{
        int j, k, sum=0;
        switch (tr)
        {
                case 3:
                        for (j=0; j<N; j++)
                        {
                            for (k=0; k<N; k++)
                                sum += tree_3[j][k];
                        }
                        break;
                case 4:
                        for (j=0; j<N; j++)
                        {
                            for (k=0; k<N; k++)
                                sum += tree_4[j][k];
                        }
                        break;
        }
        return sum;
}
int compare_trees (int diam)
{
        int t12=0, t13=0, t14=0, t23=0, t24=0, t34=0;
/* 0 in tij means that the i-j trees are identical */
        int j, k, part1, part2;
        for (j=0; j<N; j++)
/* Compares tree to tree_2 */
```

```
        {
            for (k=0; k<N; k++)
                if (tree[j][k] != tree_2[j][k])
                {
                  t12 = 1;
                  break;
                }
         }
         for (j=0; j<N; j++)
/* Compares tree to tree_3 */
        {
         for (k=0; k<N; k++)
              if (tree[j][k] != tree_3[j][k])
              {
                 t13 = 1;
                 break;
              }
        }
         for (j=0; j<N; j++)
/* Compares tree to tree_4 */
        {
         for (k=0; k<N; k++)
             if (tree[j][k] != tree_4[j][k])
             {
               t14 = 1;
               break;
              }
        }
        for (j=0; j<N; j++)
/* Compares tree_2 to tree_3 */
        {
        for (k=0; k<N; k++)
             if (tree_2[j][k] != tree_3[j][k])
             {
               t23 = 1;
               break;
             }
        }
        for (j=0; j<N; j++)
/* Compares tree_2 to tree_4 */
        {
        for (k=0; k<N; k++)
```

```
            if (tree_2[j][k] != tree_4[j][k])
            {
            t24 = 1;
            break;
            }
        }
for (j=0; j<N; j++)
/*Compares tree_3 to tree_4 */
        {
        for (k=0; k<N; k++)
            if (tree_3[j][k] != tree_4[j][k])
            {
            t34 = 1;
            break;
            }
        }
        switch (diam)
        {
        case 2:
                part1 = tree_total_sum(3);
                if ((part1 == 0 || t13 == 0 || t23 == 0) && (t12 !=0))
                  return 0;
             break;
        case 3:
                part2 = tree_total_sum(4);
                if ((part2 == 0 || t14 == 0 || t24 == 0 || t34 == 0) && (t13 != 0) && (t23 !=
0))
                  return 0;
             break;
         }
         return 1;
}
```

4.4 Υλοποίηση του αλγόριθμου Kruskal σε γλώσσα C.

```
#include “stdio.h”
#define MAX 50
#define INFINITY 4000
void inputgraph(int);
void makeset(int);
void findmin(int);
int findset(int);
void unionset(int,int);
int wt[MAX][MAX];
int edge[MAX][MAX];
int edge1,edge2;
int c = INFINITY;
int mstree[ 2*MAX];
int n;
int set[MAX];
int flag;
int tedge = 0;
main()
{
  int i,j;
  int k=0;
  FILE *f = fopen(“dist.txt”, “r”);
  fscanf(f, “%d”, &n);
  for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    {
      fscanf(f, “%d”, &wt[i][j]);
      if(wt[i][j] == 0)
      {
          edge[i][j] = 0;
          edge[j][i] = 0;
          }
    if(wt[i][j] != 0)
    {
    edge[i][j] = 1;
    edge[j][i] = 1;
    tedge = tedge + 2;
```

```
    }
  }
fclose(f);
makeset(n);
for(i = 1;i<=tedge;i++)
{
findmin(n);
if(findset(edge1) != findset(edge2))
{
mstree[k++] = edge1;
mstree[k++] = edge2;
edge[edge1][edge2] = 0;
unionset(edge1,edge2);
}
}
k=0;
flag = 1;
for(i = 0;i
{
printf("Edge no. %d of minimum spanning tree hv vertices %d and %d
\n",flag,mstree[k],mstree[k+1]);
flag++;
k = k+2;
}
}
void makeset(int n)
{
int i;
for (i=1;i<=n;i++)
{
set[i] = i;
}
}
void findmin(int n)
{
int i,j;
for (i=0;i
{
for(j=i;j<=n;j++)
{
if(edge[i][j] == 1)
{
```

```
if(wt[i][j] <= c)
{
c = wt[i][j];
edge1 = i;
edge2 = j;
}
}
}
}
edge[edge1][edge2]= 0;
c = INFINITY;
}
int findset( int a)
{
return set[a];
}
void unionset(int a,int b)
{
int z;
int temp;
temp = set[b];
set[b] = set[a];
for(z = 0;z
{
if(set[z] == temp)
{
set[z] = set[a];
   }
  }
}
```

4.5 Υλοποίηση του αλγόριθμου Kruskal σε γλώσσα Pascal.

Ορίζουμε την συνάρτηση Modset.pas

ModSet.pas

```
01 unit ModSet;{$MODE DELPHI}{$r+}
02
03 interface
04
05 const
06   nVertices = 10;
07
08
09 procedure InitFirst (n : integer);
10 procedure Merge (a, b : integer);
11 function Find (x : integer) : integer;
12
13
14 implementation
15 type
16 MFSetRec = record
17   setHeaders : array[1..nVertices] of record
18     count : integer;
19     firstElement : integer;
20   end;
21   names : array[1..nVertices] of record
22     setName : integer;
23     nextElement : integer;
24   end;
25 end;
26
27 var
28   MFSet : MFSetRec;
29
30 procedure Initial (a, x : integer);
31 begin
32   MFSet.names[x].setName := a;
33   MFSet.names[x].nextElement := 0;
```

```
34   MFSet.setHeaders[a].count := 1;
35   MFSet.setHeaders[a].firstElement := x
36 end;
37
38 function Find (x : integer) : integer;
39 begin
40   find := MFSet.names[x].setName
41 end;
42
43 procedure Merge (a, b : integer);
44 var
45   i, temp : integer;
46 begin
47   if MFSet.setHeaders[b].count > MFSet.setHeaders[a].count then
48     begin temp := a; a := b; b := temp end;
49   i := MFSet.setHeaders[b].firstElement;
50   while MFSet.names[i].nextElement <> 0 do
51   begin
52     MFSet.names[i].setName := a;
53     i := MFSet.names[i].nextElement
54   end;
55   MFSet.names[i].setName := a;
56   MFSet.names[i].nextElement := MFSet.setHeaders[a].firstElement;
57   MFSet.setHeaders[a].firstElement := MFSet.setHeaders[b].firstElement;
58                 MFSet.setHeaders[a].count      :=      MFSet.setHeaders[a].count      +
MFSet.setHeaders[b].count
59 end;
60
61 procedure InitFirst (n : integer);
62   var i : integer;
63 begin
64   for i := 1 to n do
65     Initial (i, i)
66 end;
67
68
69 begin
70 end.
71
```

Ορίζουμε την συνάρτηση ModIput.pas:

ModInput.pas

```
01 unit ModInput;{$MODE DELPHI}{$r+}
02
03 interface
04
05 procedure ReadEdges;
06
07 implementation
08
09 uses ModEdges;
10
11 procedure ReadEdges;
12 var u, v, w : integer;
13 begin
14   while not eof do
15   begin
16     readln (u, v, w);
17     AddEdge (u, v, w)
18   end
19 end;
20
21 begin
22 end.
23
```

Ορίζουμε τη συνάρτηση Modedges.pas :

ModEdges.pas

```
01 unit ModEdges;{$MODE DELPHI}{$r+}
02
03 interface
```

```
04
05 const nEdges = 1024;
06
07 procedure AddEdge (a, b, w : integer);
08 function GetEdgeU (i : integer) : integer;
09 function GetEdgeV (i : integer) : integer;
10 function EdgeCount : integer;
11 function HighestVertex : integer;
12 procedure SortEdges;
13
14
15 implementation
16
17 type
18   edge = record
19     u, v, weight : integer;
20   end;
21
22   edgesRec = array[1..nEdges] of edge;
23
24 var
25   edges : edgesRec;
26   numEdges, vHighest : integer;
27
28 procedure AddEdge (a, b, w : integer);
29 begin
30   numEdges := numEdges + 1;
31   if a > vHighest then vHighest := a;
32   if b > vHighest then vHighest := b;
33   edges[numEdges].u := a;
34   edges[numEdges].v := b;
35   edges[numEdges].weight := w
36 end;
37
38 function GetEdgeU (i : integer) : integer;
39 begin
40   GetEdgeU := edges[i].u
41 end;
42
43 function GetEdgeV (i : integer) : integer;
44 begin
45   GetEdgeV := edges[i].v
```

```
46 end;
47
48 function EdgeCount : integer;
49 begin
50   EdgeCount := NumEdges
51 end;
52
53 procedure SortEdges;
54 var i, j : integer;
55   key : edge;
56 begin
57   for j := 2 to EdgeCount do
58   begin
59     key := edges[j];
60     i := j - 1;
61     while (i > 0) and (edges[i].weight > key.weight) do
62     begin
63       edges[i+1] := edges[i];
64       i := i - 1
65     end;
66     edges[i+1] := key
67   end
68 end;
69
70 function HighestVertex : integer;
71 begin
72   HighestVertex := vHighest
73 end;
74
75 begin
76   numEdges := 0
77 end.
```

Το κυρίως πρόγραμμα mailine.pas :

Mainline.pas

```
01 program kruskal (input, output);{$MODE DELPHI}{$r+}
02
03 uses sysutils, ModSet, ModEdges, ModInput;
04
05 var i, u, v : integer;
06
07 begin
08   try
09     ReadEdges;
10   except
11     writeln ('Error reading input');
12     exit
13   end;
14   try
15     InitFirst (HighestVertex);
16   except
17     writeln ('Error initializing vertices');
18     exit
19   end;
20   SortEdges;
21   for i := 1 to EdgeCount do
22   begin
23     u := GetEdgeU (i);
24     v := GetEdgeV (i);
25     if Find(u) <> Find (V) then
26     begin
27       writeln (u, '->', v);
28       Merge (find (u), Find (v))
29     end
30   end
31 end.
32
```

4.6 Υλοποίηση του αλγόριθμου Prim σε γλώσσα C.

```
#include "stdio.h"

/*
        The input file (dist.txt) look something like this
                4
                0 0 0 21
                0 0 8 17
                0 8 0 16
                21 17 16 0

        The first line contains n, the number of nodes.
        Next is an nxn matrix containg the distances between the nodes
        NOTE: The distance between a node and itself should be 0
*/

int n; /* The number of nodes in the graph */

int weight[100][100]; /* weight[i][j] is the distance between node i and node j;
                         if there is no path between i and j, weight[i][j] should
                         be 0 */

char inTree[100]; /* inTree[i] is 1 if the node i is already in the minimum
                     spanning tree; 0 otherwise*/

int d[100]; /* d[i] is the distance between node i and the minimum spanning
               tree; this is initially infinity (100000); if i is already in
               the tree, then d[i] is undefined;
               this is just a temporary variable. It's not necessary but speeds
               up execution considerably (by a factor of n) */

int whoTo[100]; /* whoTo[i] holds the index of the node i would have to be
                   linked to in order to get a distance of d[i] */

/* updateDistances(int target)
        should be called immediately after target is added to the tree;
        updates d so that the values are correct (goes through target's
        neighbours making sure that the distances between them and the tree
        are indeed minimum)
```

```
*/
void updateDistances(int target) {
	int i;
	for (i = 0; i < n; ++i)		if ((weight[target][i] != 0) && (d[i] > weight[target][i])) {
			d[i] = weight[target][i];
			whoTo[i] = target;
		}
}

main() {
	FILE *f = fopen("dist.txt", "r");
	int i,j,total,treeSize;
	fscanf(f, "%d", &n);
	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j)
			fscanf(f, "%d", &weight[i][j]);
	fclose(f);

	/* Initialise d with infinity */
	for (i = 0; i < n; ++i)
		d[i] = 100000;

	/* Mark all nodes as NOT beeing in the minimum spanning tree */
	for (i = 0; i < n; ++i)
		inTree[i] = 0;

	/* Add the first node to the tree */
	printf("Adding node %c\n", 0 + 'A');
	inTree[0] = 1;
	updateDistances(0);
	total = 0;
	for (treeSize = 1; treeSize < n; ++treeSize) {
		/* Find the node with the smallest distance to the tree */
		int min = -1;
		for (i = 0; i < n; ++i)			if (!inTree[i])
	if ((min == -1) || (d[min] > d[i]))
					min = i;

		/* And add it */
		printf("Adding edge %c-%c\n", whoTo[min] + 'A', min + 'A');
		inTree[min] = 1;
```

```
                total += d[min];

                updateDistances(min);
        }

        printf("Total distance: %d\n", total);

        return 0;
}
```

4.7 Υλοποίηση του αλγόριθμου Dijkstra σε γλώσσα C.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
 int graph[15][15],s[15],pathestimate[15],mark[15];
 int num_of_vertices,source,i,j,u,predecessor[15];
 int count=0;
 int minimum(int a[],int m[],int k);
 void printpath(int,int,int[]);
 printf("\nenter the no.of vertices\n");
 scanf("%d",&num_of_vertices);
 if(num_of_vertices<=0)
 {
  printf("\nthis is meaningless\n");
   exit(1);
 }
 printf("\nenter the adjacent matrix\n");
    for(i=1;i<=num_of_vertices;i++)
 {
  printf("\nenter the elements of row %d\n",i);
  for(j=1;j<=num_of_vertices;j++)
  {
   scanf("%d",&graph[i][j]);
  }
 }
 printf("\nenter the source vertex\n");
 scanf("%d",&source);
```

```
 for(j=1;j<=num_of_vertices;j++)
 {
  mark[j]=0;
  pathestimate[j]=999;
  predecessor[j]=0;
 }
 pathestimate=0;

 while(count<num_of_vertices)
 {
  u=minimum(pathestimate,mark,num_of_vertices);
  s[++count]=u;
  mark[u]=1;
  for(i=1;i<=num_of_vertices;i++)
  {
   if(graph[u][i]>0)
   {
    if(mark[i]!=1)
    {
    if(pathestimate[i]>pathestimate[u]+graph[u][i])
    {
     pathestimate[i]=pathestimate[u]+graph[u][i];
     predecessor[i]=u;
      }
     }
    }
   }
  }
 for(i=1;i<=num_of_vertices;i++)
 {
 printpath(source,i,predecessor);
 if(pathestimate[i]!=999)
 printf("->(%d)\n",pathestimate[i]);
    }
  }
int minimum(int a[],int m[],int k)
{
 int mi=999;
 int i,t;
 for(i=1;i<=k;i++)
 {
  if(m[i]!=1)
```

```
 {
 if(mi>=a[i])
  {
  mi=a[i];
  t=i;
    }
  }}
 return t;
 }
void printpath(int x,int i,int p[])
 {
 printf("\n");
if(i==x)
 {
printf("%d",x);
   }
else if(p[i]==0)
printf("no path from %d to %d",x,i);
else
 {
printpath(x,p[i],p);
printf("..%d",i);
 }
}
```

Βιβλιογραφία

- Louigi Addario-Berry, Nicolas Broutin and Bruce Reed. “The Diameter of the Minimum Spanning Tree of a Complete Graph”. Fourth Colloquium on Mathematics and Computer Science (DMTCS proc. AG, 2006, 237–248), Nancy, France.

- Matthew Johnson, Algorithms & Complexity - Lecture 3: Minimum Spanning Trees”. Durham University.

- Steven Skiena. “Lecture 17: Minimum Spanning Trees”. State University of New York, Department of computer Science (1997).

- Chris Brooks. “Data Structure and Algorithms - Spanning Trees”. University of San Francisco.

- Steve Palmer, Professor Dr. Wyatt. “Analysis of Algorithms – CSC560, Minimum Spanning Tree Project”. West Chester University of PA (2004).

- Benjamin Ertl, Prof. Dr. Franz Brandenburg. “Proseminararbeit: Algorithms and Data Structures“. University of Passau, Fakultat fur Mathematik und Informatik (2001).

- Bang Ye Wu and Kun-Mao Chao. “Spanning Trees and Optimization Problems” Chapman & Hall/CRC Press, USA (2004).

- Michel X. Goemans, Jan Vondrak, “Covering Minimum Spanning Trees of Random Subgraphs”. MIT, Dept of Mathematics, Cambridge, MA 02139.

- Anna Pagacz (1), Günther Raidl (2), Stanisław Zawislak (3). “Evolutionary Approach to Constrained Minimum Spanning Tree Problem – Commercial Software Based Application”. (1): Vienna University of Technology, Socrates student 2004/2005; regular since 2005-onwards, Vienna, Austria, (2): Vienna University of Technology, Institute of Computer Graphics and Algorithms, Vienna, Austria, (3): University of Bielsko-Biała, Faculty of Mechanical Engineering and Computer Science, Bielsko-Biała, Poland.

- Jim Kurose, Keith Ross. “Computer Networking: A Top Down Approach Featuring the Internet”. 3rd edition, Addison-Wesley, (2004).

- Vitaly Osipov, Peter Sanders, Johannes Singler. “The Filter-Kruskal Minimum Spanning Tree Algorithm”. Universität Karlsruhe (TH), Germany.

- Nabil Arman, “Generating Minimum - Cost Fault – Free Rule Bases Using Minimum Spanning Trees”. International Journal of Computing & Information Sciences, Vol. 4 No. 3, December 2006.

- Michel X. Goemans. “Minimum Spanning Trees - Radom Models”. Dept. Math., MIT (2005).

- Held M., & Karp R., “The Travelling-Salesman Problem and Minimum Spanning Trees” *Ops Research*, v.18, n.6, Nov-Dec 1970.

- Longin Jan Latecki, “minimum Spanning Trees” Temple University. Based on slides by David Matuszek, UPenn, Rose Hoberman, CMU,
- Bing Liu, U. of Illinois, Boting Yang, U. of Regina.

- Asst. Prof. Dr. Bunyarit Uyyanonvara. “Topic 10, Minimum Spanning Tree Problem” ITS033, IT Program, Image and Vision Computing Lab, School of Information, Computer and Communication Technology (ICT), Sirindhorn International Institute of Technology (SIIT), Thammasat University.

- A. Aho / J. E. Hopcroft / J. D. Ullman: Data Structures and Algorithms. Addison- Wesley Publishing Company, Reading Massachusetts, 1983, [p.233-239].

- A. Aho / J. D. Ullman: Informatik. Datenstrukturen und Konzepte der Abstraktion. International Thomson Publishing, Bonn [and others], 1996, [p.638-647].

- T. H. Cormen / C. E. Leiserson / R. L. Rivest: Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts [and others], 1990, [p.498-511].

- M. T. Goodrich / R. Tamassia: Data Structures and Algorithms in Java. John Wiley & Sons, Inc., 1998, [p.145-202, p.411-423].

- T. Ottmann / P. Widmayer: Algorithmen und Datenstrukturen. BI-Wissenschafts- Verlag, Mannheim [and others], 1990, [p.583-590].

- H.-P. Gumm / M. Sommer: Einführung in die Informatik. Addison-Wesley Publishing Company, Bonn [and others], 2. edition, 1995.

- R. Sedgewick: Algorithmen. Addison-Wesley Publishing Company, Bonn, 1991, [p.513-524].

- V. Claus / A. Schwill: Schülerduden-Informatik. Dudenverlag, Mannheim [and others], 2nd. edition, 1991.

- Australas. J. Combin: Achuthan, et al, 1994, “Computational methods for the diameter restricted minimum weight spanning tree problem”, 10, 51-71

- Szekers: “Distribution of labeled trees by diameter”, Lecture Notes in Math., 1036, 392-397, 1983)

- Ayman Abdalla, Narsingh Deo, “Random-tree diameter and the diameter constrained MST”, International Journal of Computer Mathematics, 2002, Vol. 79(6), pp. 651-663

- Γεωργιάδης Θεόφιλος, Μπότσαρης Χαράλαμπος. “Spanning Tree and Graph Partitioning”. Πανεπιστήμιο Πατρών.

- Δημητρίου Σωτηρούλα, Ξεζωνάκης Ιωάννης. “Επεξεργασία και υλοποίηση αλγορίθμων για την επίλυση προβλημάτων ζευγνυόντων δέντρων (Spanning Trees)”. ΤΕΙ Κρήτης, Τμήμα Εφαρμοσμένης Πληροφορικής και Πολυμέσων.

- Zachos Stathis, Pagourtzis Aris. “Algorithms & Complexity”. Courses - Core Lab, ECE, NTUA.