ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΠΟΛΥΜΕΣΩΝ

# ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

## Επεξεργασία και υλοποίηση Αλγορίθμων για την επίλυση προβλημάτων ζευγνυόντων δέντρων (spanning trees).

**Σπουδάστρια: Δημητρίου Σωτηρούλα**

**Εισηγητής: Ξεζωνάκης Ιωάννης**

# Περιεχόμενα

# Εισαγωγή

Οι δομές χωρίζονται σε δύο μεγάλες ενότητες, στις :

I) Γραμμικές Δομές

II) Μη Γραμμικές Δομές

**Γραμμικές Δομές:**

Στις γραμμικές δομές δεδομένων τα δεδομένα είναι γραμμικά διατεταγμένα, δηλαδή κάποιο στοιχείο είναι πρώτο και κάποιο τελευταίο, ενώ για οποιοδήποτε υπάρχει ένα προηγούμενο και ένα επόμενο στοιχείο.

**Μη γραμμικές Δομές:**

Στις μη γραμμικές δομές οι σχέσεις μεταξύ των δεδομένων είναι περισσότερο περίπλοκες. Οι δομές αυτού του είδους, με τις οποίες θα ασχοληθούμε εδώ, είναι τα δένδρα και οι γράφοι. Στα δένδρα κάθε στοιχείο έχει ένα μόνο προηγούμενο, αλλά 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.

# Γενικές έννοιες – Ορισμοί

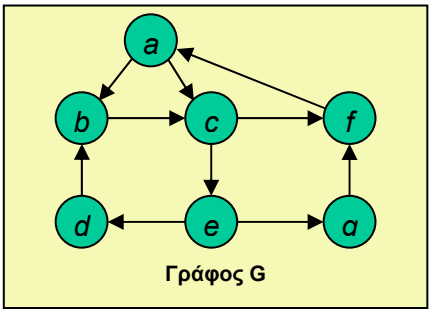
Ένας γράφος είναι μια τέτοια δομή, όπου κάθε στοιχείο μπορεί να μην έχει κανένα, ή να έχει πολλά προηγούμενα και επόμενα στοιχεία. Η δομή ενός γράφου είναι η πιο γενική μορφή δομής δεδομένων.

Το αξιοσημείωτο με τους γράφους είναι ότι έχουν μελετηθεί ευρύτατα ως μέρος των μαθηματικών δομών (κλάδος εφαρμοσμένων μαθηματικών). Η θεωρία των γράφων είναι μεγάλο και σπουδαίο αντικείμενο μελέτης και έρευνας και διδάσκεται ως ανεξάρτητο μάθημα σε τμήματα πληροφορικής και άλλων επιστημών. Στο τομέα της πληροφορικής χρησιμοποιούνται στη μελέτη Βάσεων Δεδομένων, Γλωσσών Προγραμματισμού, Δικτύων Υπολογιστών, Λειτουργικών συστημάτων κ.α.

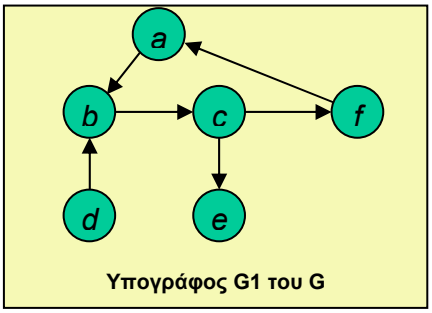
Ένας γράφος αποτελείται από ένα σύνολο ***σημείων (points) ή κορυφών (verticals)*** ή ***κόμβων (nodes)*** και ένα σύνολο ***ακμών (edges) ή τόξων (arcs) ή γραμμών (lines)*** που ενώνουν μερικά ή όλα τα σημεία του. Σε μαθηματική γλώσσα ένας γράφος συμβολίζεται ως: G=(V,E) όπου V = { $V_1$,$V_2$, ... , Vn }, με n>0 είναι το σύνολο των κορυφών και 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.

Σχηματικά τα παραπάνω δείχνονται στα σχ. 1 (α) και (β) που ακολουθούν:




Σχ. 1 (α)




Σχ. 1 (β)

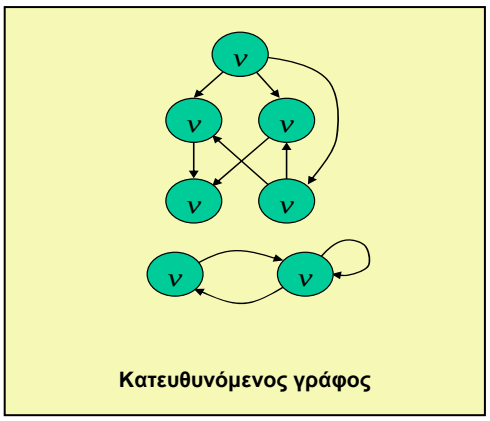
Ένας γράφος είναι ***κατευθυνόμενος*** ή ***μη κατευθυνόμενος***, αν οι ακμές του είναι ή δεν είναι προσανατολισμένες προς μία κατεύθυνση, αντίστοιχα.

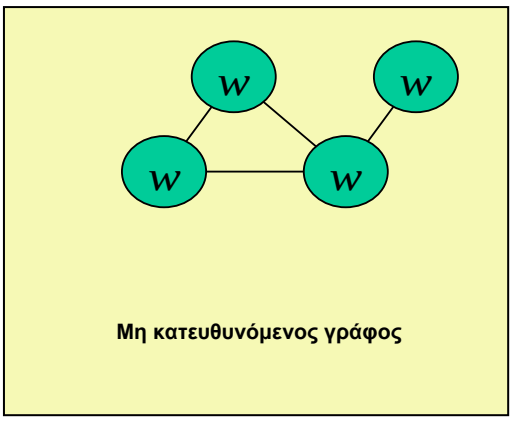
Στον κατευθυνόμενο γράφο ο προσανατολισμός της ακμής συμβολίζεται με ένα βέλος (στη συνέχεια θα λέγεται ***ακμή*** η γραμμή που συνδέει δύο κόμβους ενός μη κατευθυνόμενου γράφου μόνο, σε αντίθεση με την γραμμή που συνδέει δύο κόμβους ενός κατευθυνόμενου γράφου η οποία θα λέγεται ***τόξο***).

Στον μη κατευθυνόμενο γράφο τα ζεύγη των κορυφών που ορίζουν τις ακμές του στερούνται διάταξης.

Για παραδειγμα, μεταξύ δύο κορυφών u και v το βέλος από τη u στη v και από τη v στη u, θεωρούνται μία ακμή και όχι δύο. Στους κατευθυνόμενους γράφους θεωρούνται δύο ξεχωριστές ακμές ανάλογα με την κατεύθυνση τους. Για κατεύθυνση από τη u στη v, το u είναι η ***ουρά*** (tail) και το v η ***κεφαλή*** (head) και αντίστροφα για κατεύθυνση από τη v στη u. Από τα παραπάνω προκύπτει ότι ένας μη κατευθυνόμενος γράφος μπορεί να θεωρηθεί και ως συμμετρικός κατευθυνόμενος γράφος.

Σχηματικά, παράδειγμα κατευθυνόμενου και μη κατευθυνόμενου γράφου για τα παραπάνω δείχνονται αντίστοιχα στα σχ. 2 (α) και (β) που ακολουθούν:




Σχ. 2 (α)

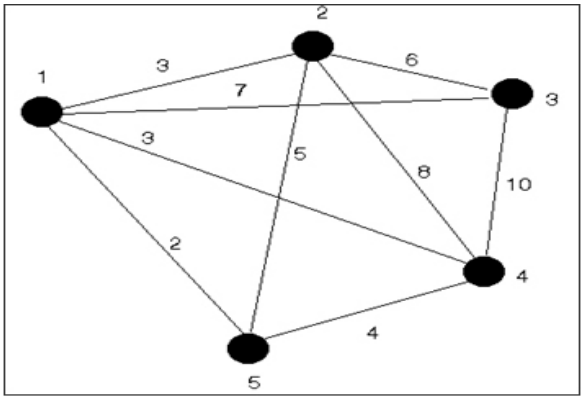
Σχ. 2(β)

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), με τον μέγιστο αριθμό ακμών λέγεται ***πλήρης***.

***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.

Σχηματικά, παράδειγμα ζυγισμένου γράφου φαίνεται στο σχ. 3 που ακολουθεί:




Σχ. 3

Δύο κορυφές οι οποίες ειναι διπλανές σε ένα γράφο ονομάζονται ***γειτονικές***.
Π.χ. : στο σχ. 3 οι κορυφές 1 και 2 είναι γειτονικές.
Σε περίπτωση που δεν συνδέονται μεταξύ τους , λέγονται ***ανεξάρτητες***.
Επίσης σε ένα γράφο υπάρχουν μονοπάτια.

***Μονοπάτι*** είναι μία διαδρομή του γραφήματος στην οποία μετέχει μια ακολουθία κορυφών που έχουν την ιδιότητα ότι οι μεταξύ τους ακμές ανήκουν στο γράφημα**.**
Π.χ. : το 1,2,5 είναι ένα μονοπάτι στο γράφο του σχ. 3.

***Μήκος*** ενός μονοπατιού είναι ο αριθμός ακμών που περιέχει.
Π.χ. : το μονοπάτι 1,2,3,4 στο σχ. 3 έχει μήκος = 4. Δηλαδή έχει 4 ακμές όπου ενώνουν τις κορυφές ακολουθόντας τη σειρά με την οποία γράφονται.

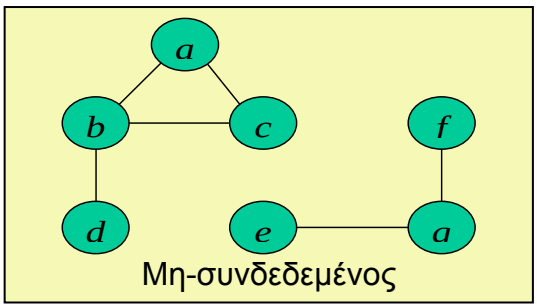
***Απλό μονοπάτι*** λέγεται το μονοπάτι εκείνο όπου όλες οι κορυφές του είναι διαφορετικές.
Π.χ. : στο σχ. 3 το μονοπατι 1,2,3,4 και 5 είναι απλό εφόσον περνάει απο διαφορετικές κορυφές.

***Κύκλος*** είναι ένα κλειστό μονοπάτι, όπου ταυτίζεται η πρώτη με την τελευταία κορυφή.
Π.χ. : το μονοπάτι 1,2,3,1 στο σχ. 3 είναι κλειστό, δηλαδή κύκλος. Ξεκινάει από την κορυφή 1 και καταλήγει σε αυτή.

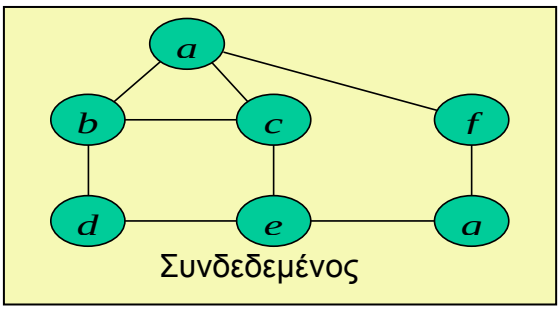
***Άκυκλος (acyclic) γράφος*** ονομάζεται ο γράφος στον οποίο δεν υπάρχουν μονοπάτια που να επιστρέφουν στον κόμβο από όπου ξεκίνησαν.

***Συνδεδεμένος (connected) γράφος*** ονομάζεται ο γράφος στον οποίο υπάρχει μονοπάτι μεταξύ οποιωνδήποτε δύο κόμβων του.

Ακολουθούν στα πιο κάτω σχήματα δύο παραδείγματα γράφων. Στο σχ. 4 (α) ένας μη συνδεδεμένος κατευθυνόμενος γράφος και στο σχ. 4 (β) ένας συνδεδεμένος κατευθυνόμενος γράφος.




Σχ.4 (α)



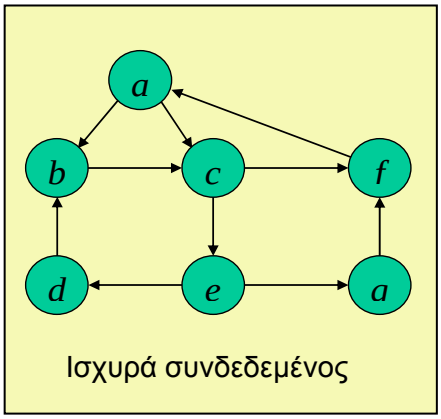

Σχ. 4 (β)

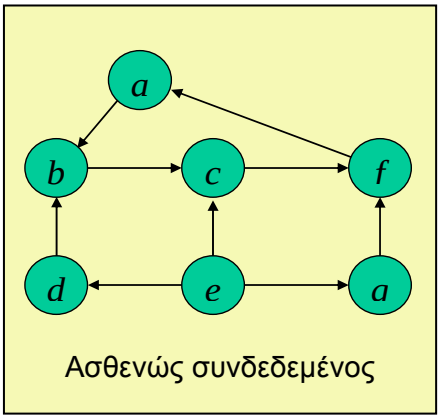
Υπάρχουν δύο κατηγορίες συνενδεμένων γράφων: Οι ισχυρά συνδεδεμένοι γράφοι (strongly connected) και οι ασθενώς συνδεδεμένοι (weakly connected).

***Ισχυρά συνδεδεμένος (strongly connected) γράφος*** ονομάζεται ο γράφος ο οποίος είναι συνδεδεμένος με όλες τις κορυφές του.

***Ασθενώς συνδεδεμένος (weakly connected) γράφος*** ονομάζεται ο γράφος ο οποίος ειναι συνδεδεμένος σε κάποιες από τις κορυφές του , αν αγνοήσουμε τις κατευθύνσεις των ακμών.

Ακολουθούν στα σχ. 5 (α) και (β) ένας ισχυρα συνδεδεμένος γράφος και ένας ασθενώς συνδεδεμένος γράφος αντίστοιχα.




Σχ. 5 (α)

Σχ. 5 (β)

***Βαθμός κορυφής*** λέγεται ο αριθμός των ακμών που είναι περιστατικά της κορυφής. 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.

**Υλοποίηση-Αποθήκευση γράφων:**

Δύο είναι οι πιο συνηθισμένοι τρόποι να παραστήσουμε ένα γράφο στον υπολογιστή:

α)με τον πίνακα γειτνίασης (adjacency matrix) ή αλλιώς πίνακα διπλανών κορυφών
β)με τη λίστα γειτνίασης (adjacency list) ή αλλιώς λίστα διπλανών κορυφών
γ) Ορθογώνια Λίστα
δ)Λίστα ακμών

Για ένα μη ζυγισμένο γράφο με n κόμβους ο πίνακας γειτνίασης A= (aij) με i,j=1,2,...,n σχηματίζεται έτσι ώστε aij=0, αν δεν υπάρχει.

Η λίστα γειτνίασης αποτελείται από n λίστες.

Στην i-οστή λίστα καταγράφονται οι κόμβοι προς τους οποίους υπάρχει ακμή από τον κόμβο vi.

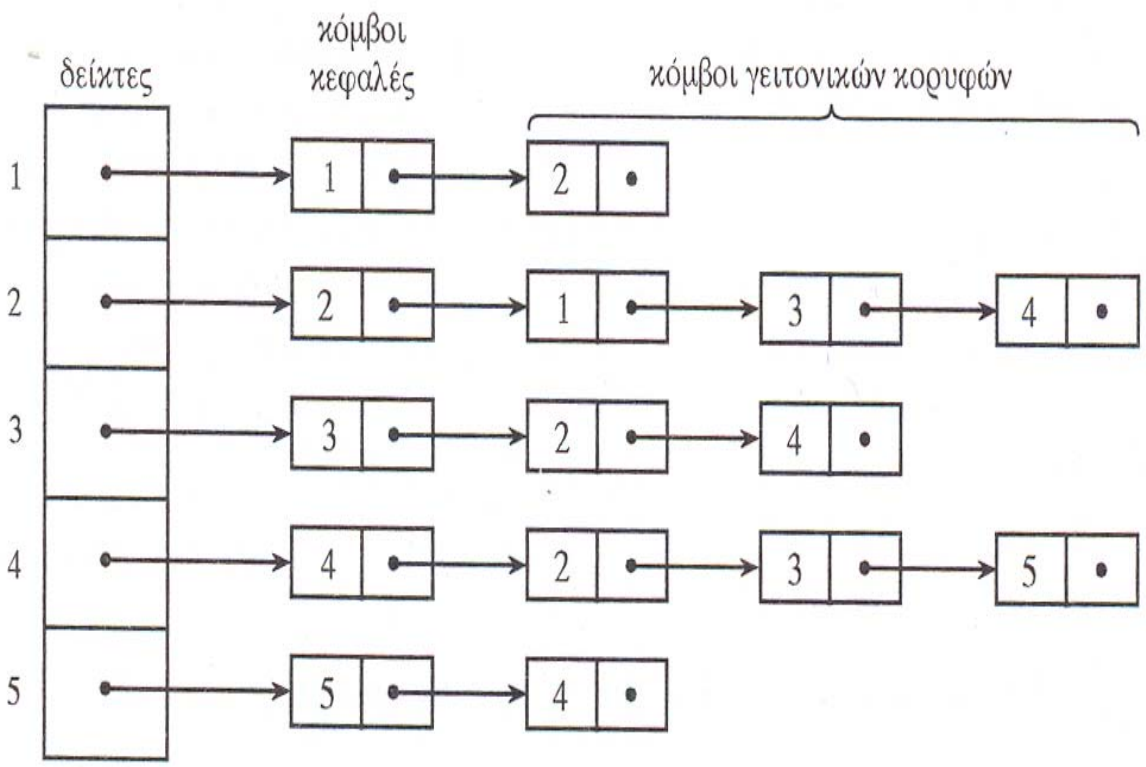
Ακολουθούν παραδείγματα στον πιν. 1 και στο σχ. 6.

Α) Πίνακας

| | Α | Β | Γ | Δ |
|---|---|---|---|---|
| Α | 0 | 1 | 1 | 0 |
| Β | 1 | 0 | 0 | 0 |
| Γ | 0 | 0 | 0 | 0 |
| Δ | 0 | 0 | 1 | 0 |

Πιν. 1

Β) Λίστα




Σχ. 6

***Αν ο γράφος είναι ζυγισμένος, τότε ο αντίστοιχος πίνακας θα περιέχει το γειτονικό κόμβο και το βάρος.***

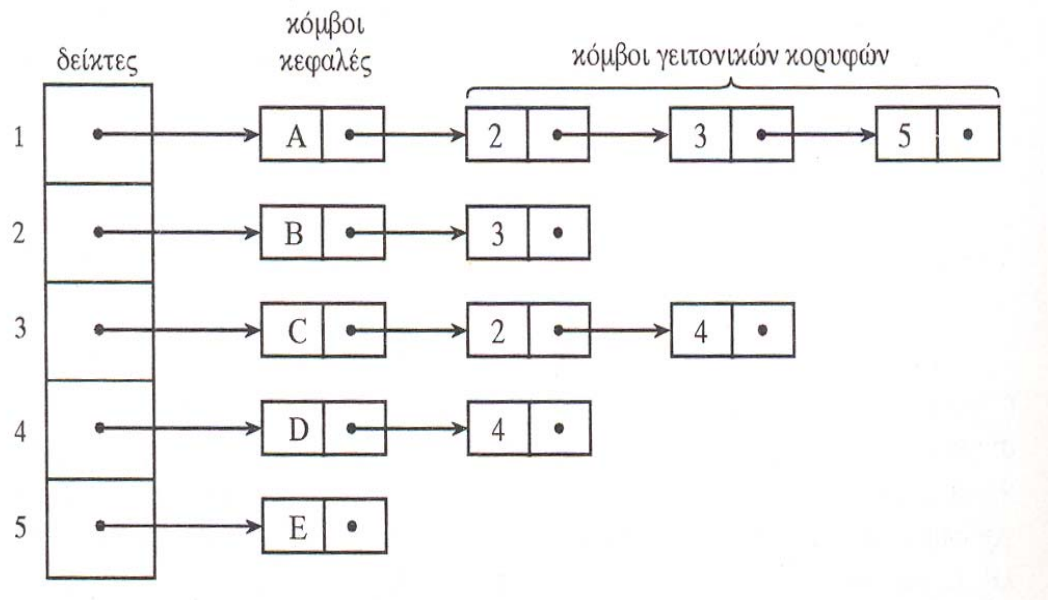
Παραδείγματα ακολουθούν στο πιν. 2 και στο σχ. 7

Α) Πίνακας

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 5 | 13 | 6 |
| 2 | 5 | 0 | 9 | 8 |
| 3 | 13 | 9 | 0 | 10 |
| 4 | 6 | 8 | 10 | 0 |

Πιν. 2

Β) Λίστα




Σχ. 7

**Γ) <u>Ορθογώνια Λίστα</u>:**

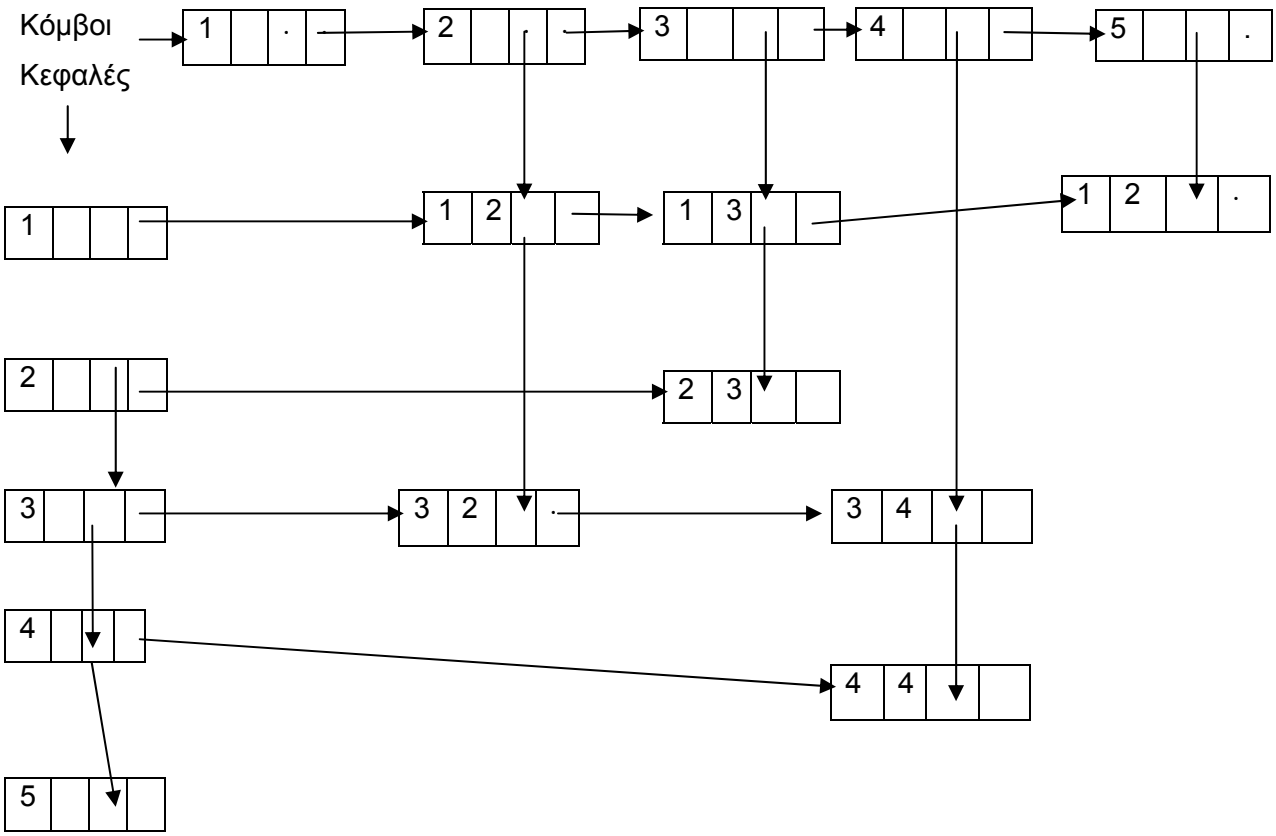
Σε αυτή τη μέθοδο χρησιμοποιείται μια απλουστευμένη μορφή παράστασης αραιών πινάκων. Κάθε κόμβος παριστάνει μια ακμή
του γραφήματος και έχει τέσσερα πεδία.

*<u>Δομή κάθε κόμβου:</u>*

| Ουρά | κεφαλή | Σύνδεσμος στήλη για κεφαλή | Σύνδεσμος γραμμής για ουρά |
|---|---|---|---|

<u>Σχ. 8</u>

Ένα παράδειγμα δομής κόμβου ακολουθεί στο σχ. 9.




<u>Σχ. 9</u>

**Δ) Λίστα ακμών:**

Χρησιμοποιεί μια συνδεδεμένη λίστα με τον ακόλουθο τρόπο:

Οι κορυφές του γραφήματος παριστάνονται σε μια συνδεδεμένη λίστα από κόμβους κεφαλές. Κάθε κόμβος κεφαλή έχει τη μορφή που φαίνεται στο σχ.10.

| |
|---|
| Δείκτης Λίστας ακμών |
| Δεδομένα |
| Σύνδεσμος γραμμής για ουρά |

Σχ. 10

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.

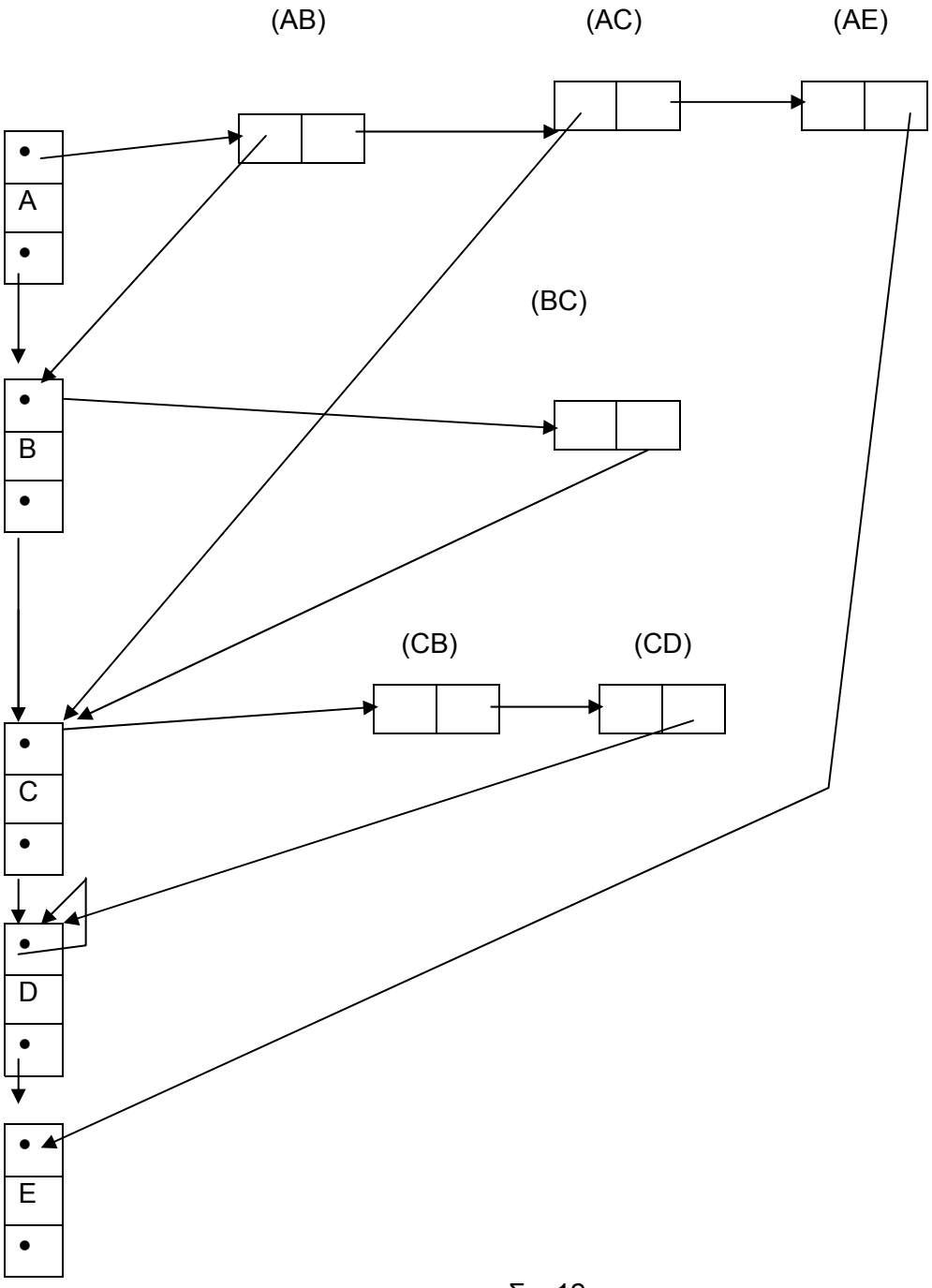
Κάθε κόμβος κεφαλή είναι η κεφαλή μίας λίστας κόμβων, η οποία καλείται λίστα ακμών. Κάθε κόμβος της λίστας ακμών παριστάνει μία ακμή του γραφήματος. Το πεδίο δείκτης λίστας ακμών είναι ένας δείκτης στη λίστα ακμών, η οποία περιλαμβάνει όλες τις ακμές που ξεκινούν από τη κορυφή του γραφήματος που παριστάνει ο κόμβος κεφαλή. Κάθε κόμβος της λίστας ακμών έχει τη μορφή που παρουσιάζεται στο σχ. 11 .

| | |
|---|---|
| Κόμβος κεφαλή | Επομένη ακμή |

Σχ. 11

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 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. 12 ) .




Σχ. 12

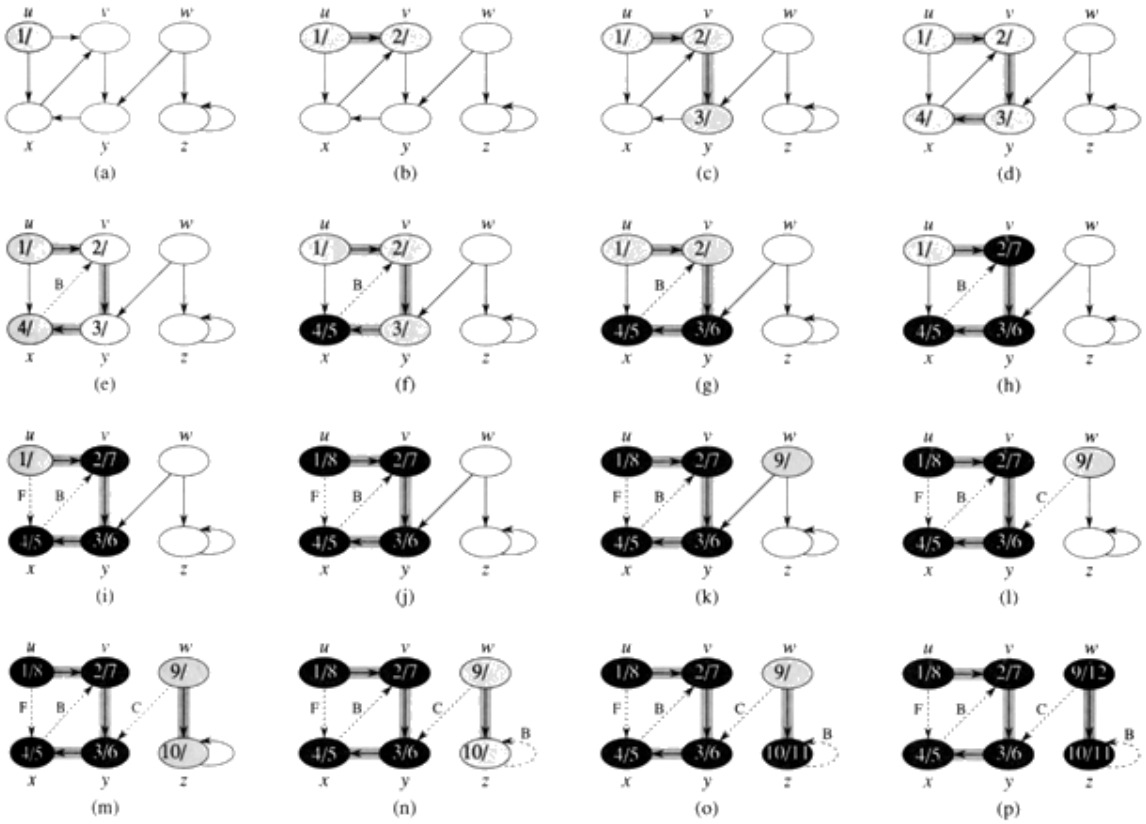
**<u>Διάσχιση γράφων:</u>**

Το πρόβλημα αυτό είναι ανάλογο με τη διάσχιση δέντρου. Δηλαδή, δεδομένου ότι έχουμε μη κατευθυνόμενο γράφο, γίνεται η επίσκεψη όλων των κόμβων ξεκινώντας από έναν από αυτούς.

Υπάρχουν δύο είδη αλγορίθμων διάσχισης ενός γράφου:

ι) *<u>Αλγόριθμος Αναζήτησης **κατά βάθος (DFS)**</u>*

Φροντίζει να επισκευθεί όλους τους κόμβους του γράφου πηγαίνοντας κάθε φορά και σε ένα νέο κόμβο (δηλ που δεν είχε επισκευθεί προηγουμένως), ο οποίος είναι γειτονικός (δηλ ενώνεται με ακμή) με τον προηγούμενο. Αρχίζει από την πρώτη κορυφή και μαρκάρει όσες έχει επισκευθεί. Αν δεν υπάρχει τέτοιος κόμβος, γίνεται επιστροφή προς τα πίσω και συνεχίζει με τις γειτονικές της προηγούμενης κορυφής. Η ανζήτηση τελειώνει όταν όλες οι κορυφές έχουν ελεγθεί. Ο έλεγχος των κορυφών γίνεται με κατακόρυφη κατεύθυνση. Λεπτομέρειες ακολουθούν στο σχ. 13 .

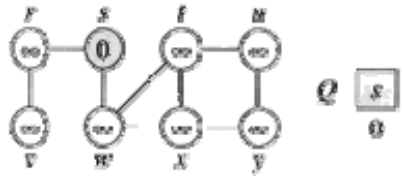



<u>Σχ. 13</u>

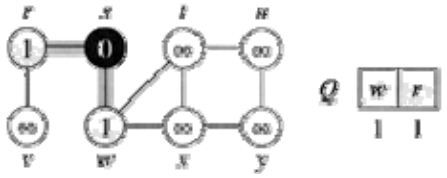
ii) *Αλγόριθμος Αναζήτησης* ***κατά πλάτος (BFS)***

Ισχύουν τα προηγούμενα με μόνη διαφορά το γεγονός ότι οι κορυφές ελέγχονται πρώτα κατά την οριζόντια αντί την κατακόρυφη κατεύθυνση. Ακολουθεί σχηματική εμφάνιση του αλγορίθμου στο σχ. 14.

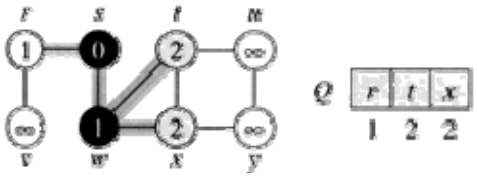
Βήμα 1



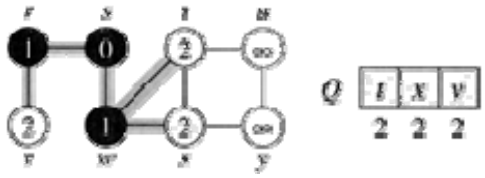
Βήμα 2



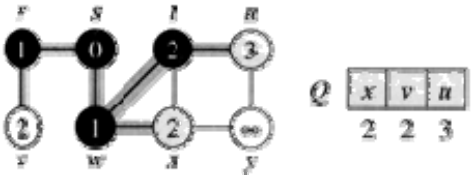
Βήμα 3



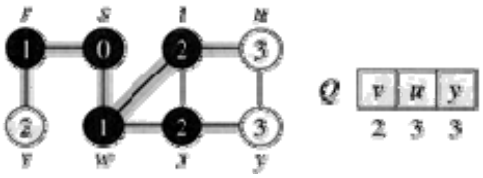
Βήμα 4



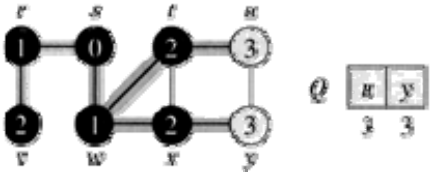
Βήμα 5



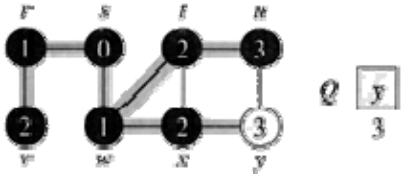
Βήμα 6



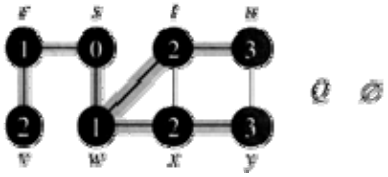
Βήμα 7



Βήμα 8



Βήμα 9

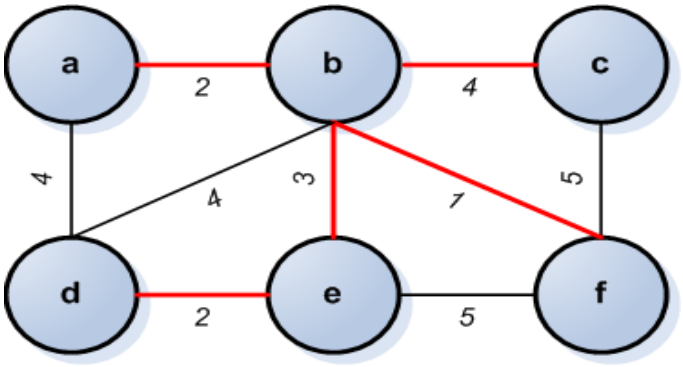


Σχ. 14

Οι πιο πάνω αλγόριθμοι δεν λύνουν το πρόβλημα διάσχισης ενός λαβυρίνθου αλλά 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 καταναλωτή, δηλαδή εύρεση του συντομότερου μονοπάτιου διάσχισης μεταξύ πολλών πόλεων (συντομότερο μονοπάτι γράφου).

***Ζευγνύον δέντρο ή δέντρο επικάλυψης (spanning tree)*** ενός γραφήματος ονομάζεται το δέντρο που έχει δημιουργηθεί από μερικές ακμές του γράφου.
Είναι δηλαδή, το ελάχιστο πλήθως ακμών που επιτρέπει την επικοινωνία μεταξύ οποινδήποτε κορυφών του αρχικού γραφήματος.
Το ζευγνύον δέντρο ονομάζεται και σκελετός ή max δέντρο.
Κάθε ζευγνύον δέντρο n κόμβων έχει n-1 ακμές.
Ένα παράδειγμα ζευγνύοντος δέντρου ακολουθεί στο σχ. 15

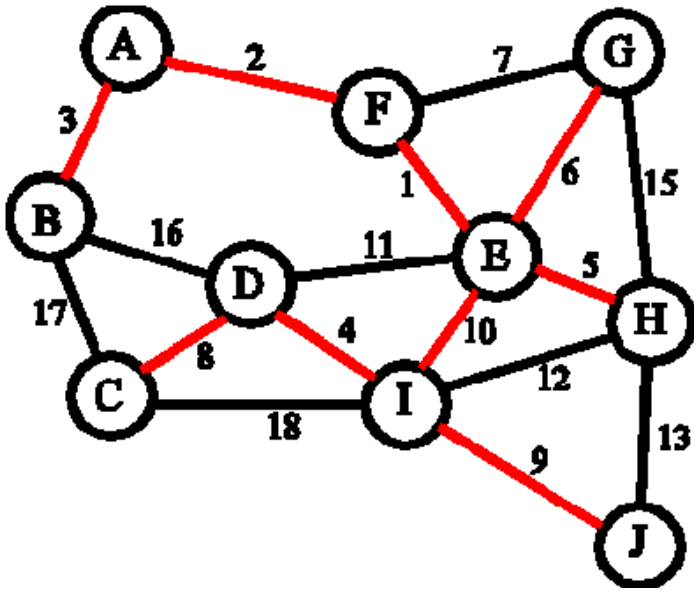



Σχ. 15

***Διάμετρος Δέντρου*** ορίζεται το μονοπάτι με την ελάχιστη απόσταση που καλύπτει όλες τις κορυφές.

***Ελαχίστο ζευγνύον δέντρο (Minimum spanning tree)*** ονομάζεται το ζευγνύον δέντρο με το ελάχιστο βάρος. Το πρόβλημα αυτό έχει μεγάλη σημασία αφού μπορεί να ανακύψει σε πληθώρα καταστάσεων, όπου πρέπει να κατασκευαστεί ένα δίκτυο από κόμβους (πχ τηλεπικοινωνιακό) που συνδέονται ανά δύο με κάποιο επιμέρους κόστος.

Στο σχ. 16 φαίνεται ένα ελάχιστο ζευγνύον δέντρο ( minimum spanning tree).




Σχ. 16

***Μεταβατικός γράφος*** ονομάζεται ένας γράφος αν και μονο αν είναι άκυκλος.

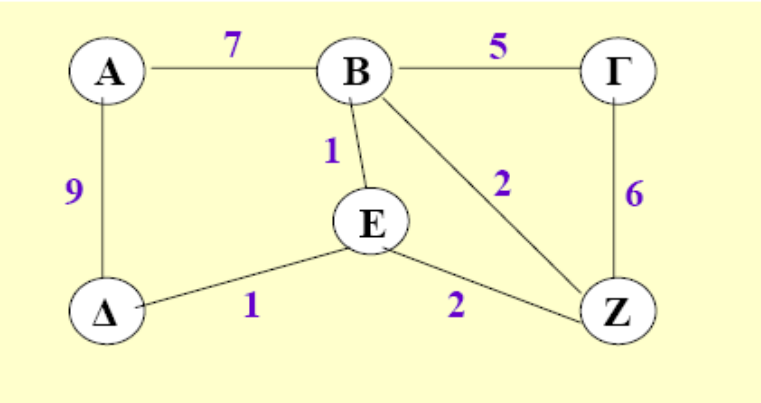
**Αλγόριθμος Kruskal:**

Ο αλγόριθμος του Kruskal επεξεργάζεται μια-μια τις ακμές του γράφου.

Οι επιλεγμένες ακμές σχηματίζουν ένα δάσος (ένα σύνολο από δένδρα).

Κεντρική ιδέα:

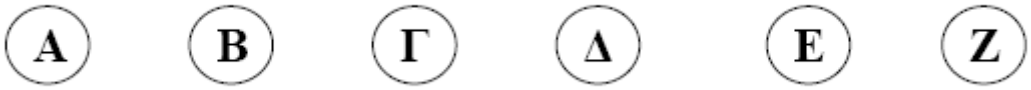
Αρχικά το Τ είναι άδειο. Επεξεργαζόμαστε μια-μια τις ακμές, σε αύξουσα σειρά βάρους: αν η ακμή e έχει το ελάχιστο βάρος από αυτές που δεν έχουμε μέχρι στιγμής επεξεργασθεί, ελέγχουμε αν η εισαγωγή της e στο Τ δεν προκαλεί κύκλο. Αν η απάντηση είναι θετική, προσθέτουμε την e στο Τ, δηλ. Τ: = Τ □ {e}.



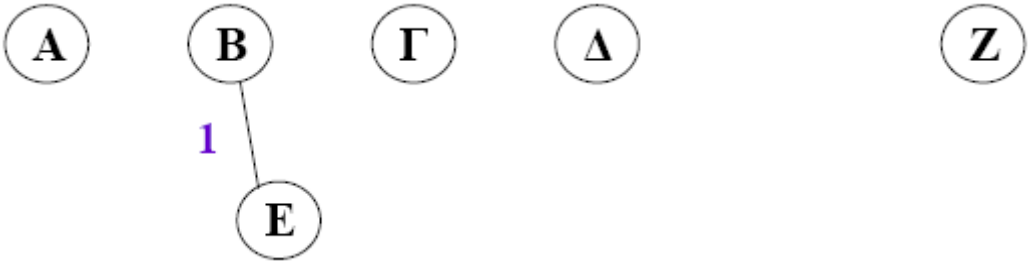

Σχ. 17

Ακολουθεί ένα παράδειγμα στον πιν. 3 βασισμένο στο σχ. 17.

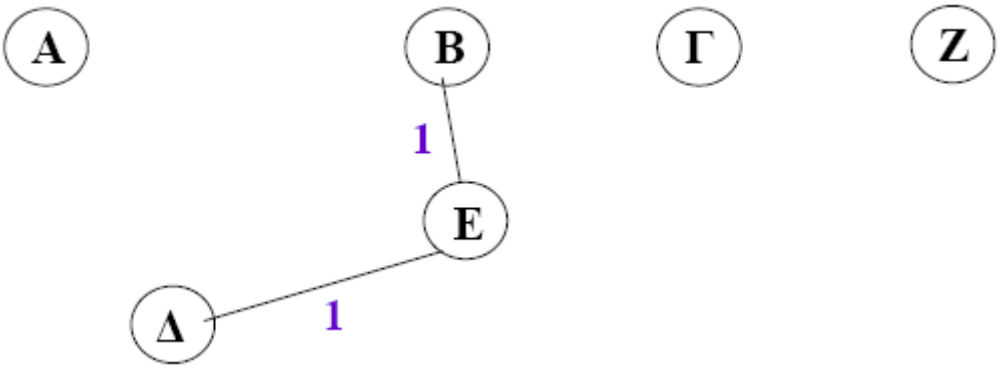
***Αρχική κατάσταση:***



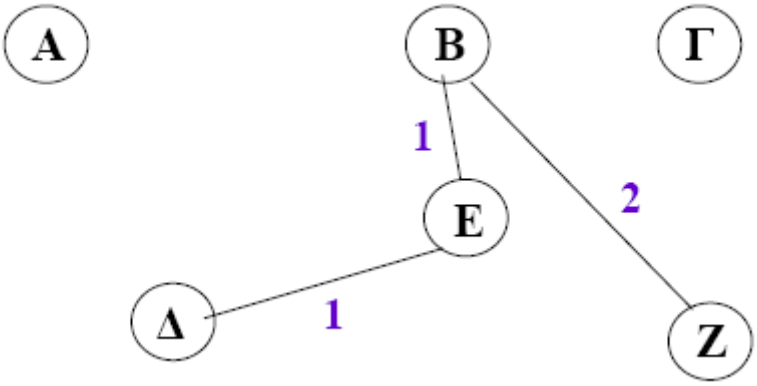

***Μετά από επιλογή της πρώτης ακμής (Β,Ε):***



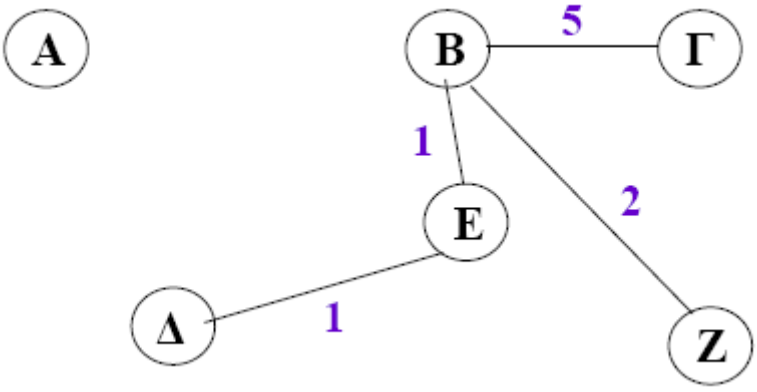

***Μετά από επιλογή της δεύτερης ακμής (Δ,Ε):***



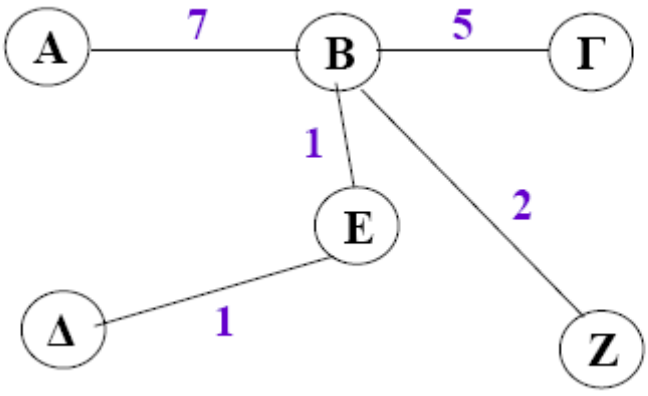

***Μετά από επιλογή της τρίτης ακμής (Β,Ζ):***




***Μετά από επιλογή της τέταρτης ακμής (Β,Γ):***




***Μετά από επιλογή της πέμπτης ακμής (Α,Β):***




Πιν. 3

Ακολουθεί ο ψευδοκώδικας του αλγόριθμου Kruskal στο πιν. 4 .

```
Kruskal (graph G)
{
buildHeap(H); (που περιέχει όλες τις ακμές του G)
partition A = {{v1},{v2},…,{vn}};
I = 1;
while (I < n)
{
(u,v) = DeleteMin(H);
U = find(A,u);
V = find(A,v);
if (U!=V)
T = T□{(u,v)};
union(A, U, V);
i++;
}
}
```

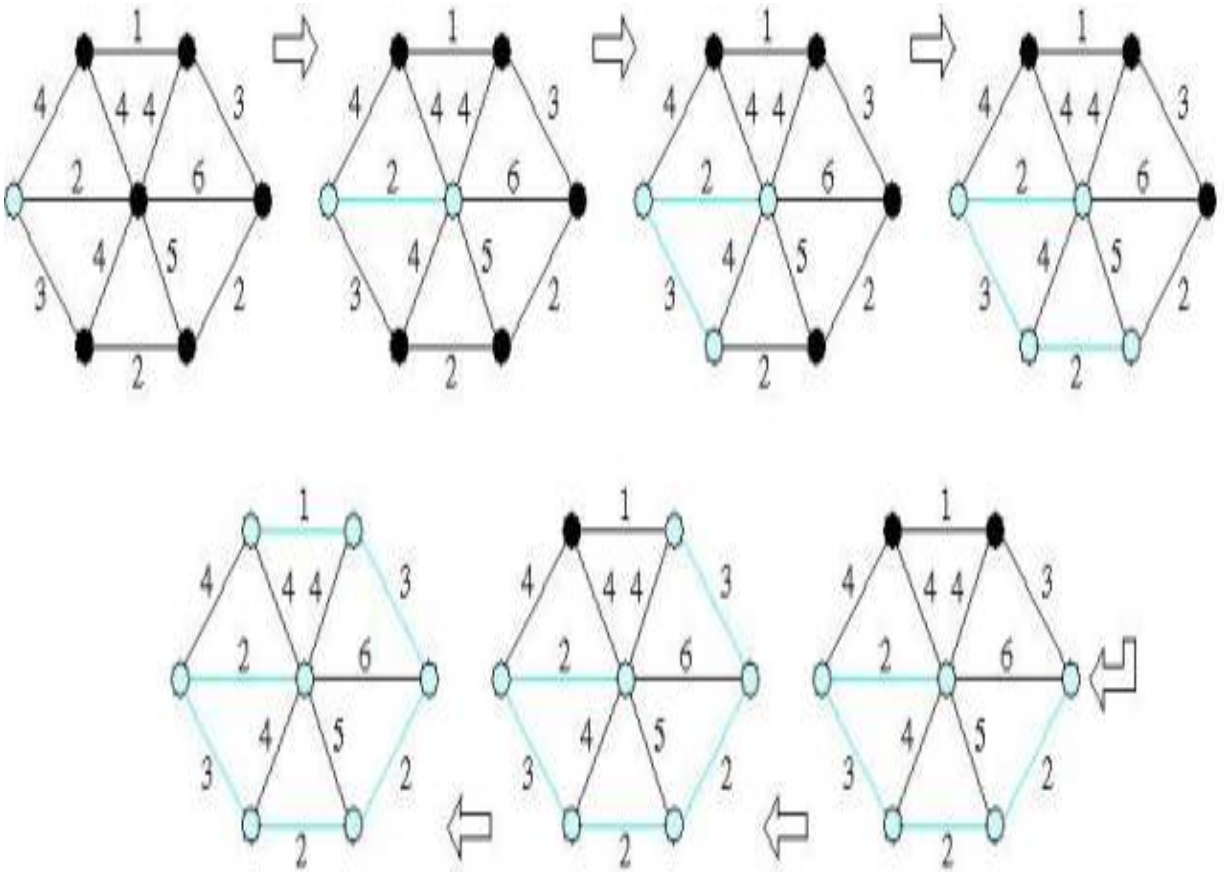
Πιν. 4

• Η διαδικασία **find(A,u)** βρίσκει και επιστρέφει το υποσύνολο του Α που περιέχει το στοιχείο u και η διαδικασία **union(A,U,V)** επιστρέφει το σύνολο Α΄ = Α – {U,V} □{ U□V} .

**Ο ΑΛΓΟΡΙΘΜΟΣ ΤΟΥ ΤΟΥ PRIM**

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

Ένα σχηματικό παράδειγμα του αλγόριθμου PRIM ακολουθεί στο σχ. 18 .



Σχ. 18

Στο πιν. 5 ακολουθεί ο ψευδοκώδικας 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] }
```

Πιν. 5

**Αλγόριθμος dijkstra:**

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

**Βασική ιδέα:**

Αρχικά βρίσκουμε τον κόμβο K που συνδέεται με τον 1 με το μικρότερο μήκος σύνδεσης. Στη συνέχεια, ο συντομότερος δρόμος θα περιλαμβάνει είτε έναν άλλο κόμβο K' που συνδέεται με τον 1 είτε έναν άλλον κόμβο K'' που συνδέεται με το K', 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 = $\infty$, αν οι i και j δεν συνδέονται).

Αναλυτικά η βασική ιδέα του αλγορίθμου εμφανίζεται στο πιν.16.

*Αρχικά*:

$P = \{1\}$, $D1 = 0$, $Dj = dj1$, $j \neq 1$.

*Βήμα 1*:

Βρείτε $i \notin P$ τέτοιο ώστε $Di = \min j \notin P Dj$.

Θέσατε $P := P \cap \{i\}$.

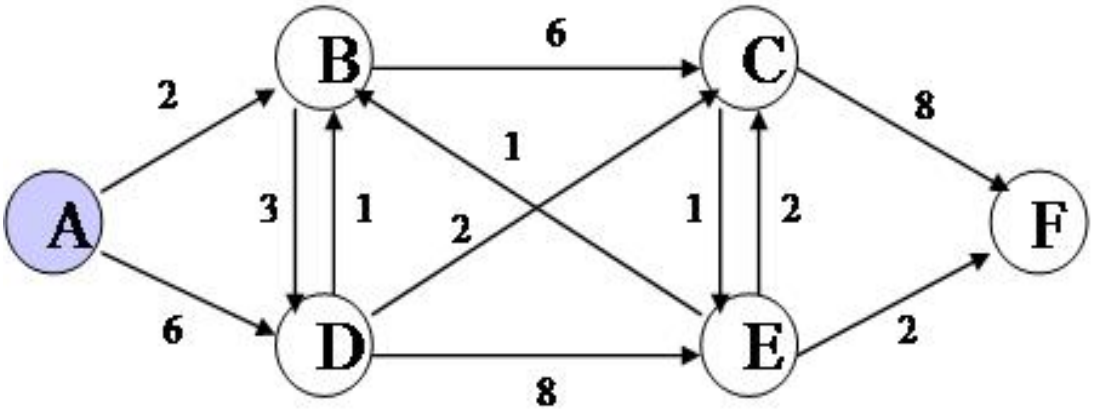
Αν $P = S$, τερματίστε.

*Βήμα 2*:

Για κάθε $j \notin P$, θέσατε $Dj = \min[Dj, dji + Di]$.

Πάτε στο βήμα 1.

Πιν. 6




Σχ. 19

Ακολουθεί στο πιν. 7 ένα παράδειγμα εφαρμογής του Αλγόριθμου Dijkstra βασισμένο στο σχ.19 .

Ξεκινάμε τυχαία με την A 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.

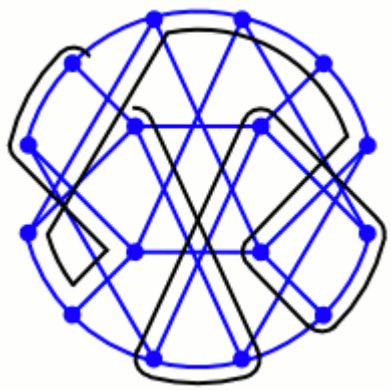
CE=1, CF=8. Επομένως παίρνουμε την E. Και τελευταία είναι η F.

Επομένως το μονοπάτι μας είναι: ¨ABDCF¨.

Πιν. 7

**Κύκλος Hamilton** λέγεται ο κύκλος που περνά από όλες τις κορυφές ενός συνδεδεμένου γραφήματος (n κορυφές).

**Μονοπάτι Hamilton** έχουμε όταν από ένα κύκλο Hamilton διαγραφεί μια ακμή (n-1 κορυφές).
Στο σχ. 20 εμφανίζεται το μονοπάτι Hamilton .

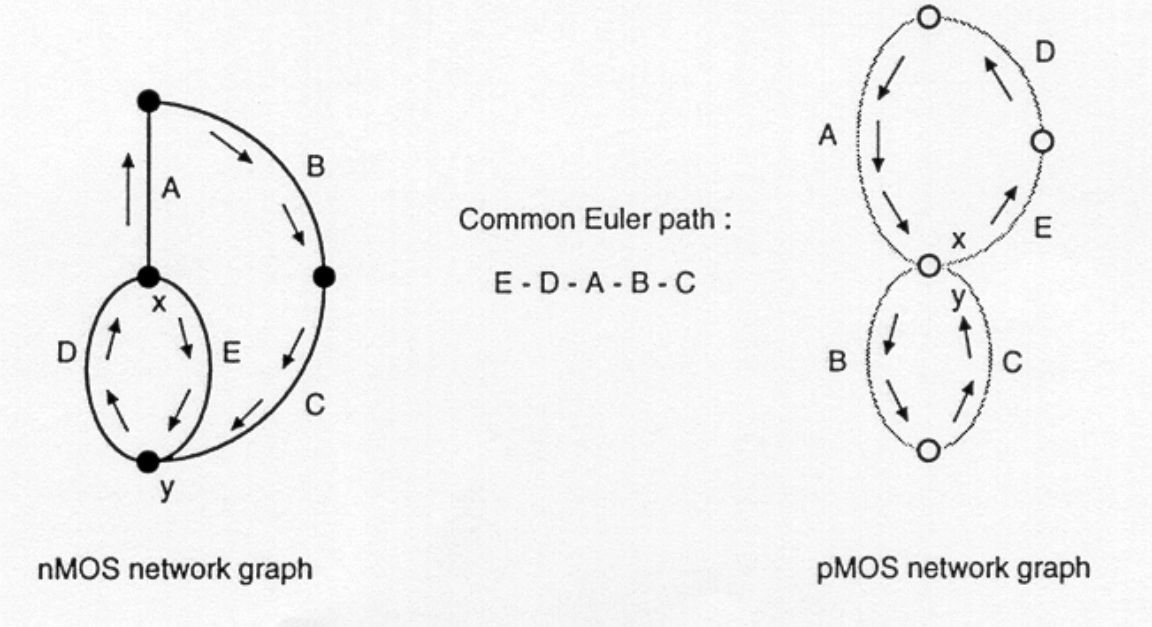


Σχ. 20

**Μονοπάτι Euler** ονομάζεται ο κύκλος που περνά από όλες τις κορυφές του γραφήματος ακριβώς μια φορά.

Η θεωρία των γράφων θεωρείται ότι ξεκίνησε με τον Euler και το πρόβλημα των γεφυρών του Koenigsberg, τότε η πόλη της ανατολικής Πρωσίας, σήμερα η πόλη της Ρωσσίας (Καλλίνιγκραντ). Το πρόβλημα που τέθηκε ήταν να βρεθεί τρόπος να διασχίσει κανείς το Koenigsberg περνώντας από τις γέφυρες του ακριβώς μια φορά από την καθεμιά. Ο Euler απέδειξε ότι είναι αδύνατο, παρατηρώντας ότι οι περιοχές της ξηράς έχουν περιττό αριθμό γεφυρών. Είναι αξιοσημείωτο ότι η συγκεκριμένη λύση παραμένει η βάση για τον καλύτερο αλγόριθμο που υπάρχει μέχρι σήμερα για τη λύση του προβλήματος.

Ακολουθεί ένα παράδειγμα στο σχ. 21.




Σχ. 21

**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.

## Εφαρμογές γραφημάτων:

Η εύρεση του πλήθους των μονοπατιών μεταξύ δύο κορυφών και το κοστος επικοινωνίας είναι δύο από τα προβλήματα των γραφημάτων που παρουσιάζουν μεγαλύτερο ενδιαφέρον. Τα γραφήματα έχουν πολλές εφαρμογές... π.χ. στα δίκτυα.
Το ζευγνύον δέντρο (Spanning tree) σχεδιάστηκε για τη λύση των προβλημάτων της συμφόρησης που δημιουργείται από τη σύνδεση τοπικών δικτύων (LANs) με υπεράριθμες γέφυρες (bridges) μεταφοράς. Ο πυρήνας του προβλήματος είναι η ποιότητα που έχουν αυτές οι γέφυρες. Ο τρόπος με τον οποίο ξέρουν πώς γίνεται η προώθηση της συμφόρησης μεταξύ των θυρών τους γίνεται μέσω «αδιάκριτων» Ethernet πλαίσια (frames) και μαθαίνοντας πια MAC διεύθυνση (address) αντιστοιχεί σε κάθε θύρα της γέφυρας (bridge).Όταν ένα πλαίσιο (frame) έρχεται για μία δοθέντα MAC διεύθυνση (address) η γέφυρα ξέρει σε πια θύρα να το στείλει. Αν ένα πλαίσιο (frame) φτάσει και η MAC διεύθυνση (address) προορισμού του είναι άγνωστη στη γέφυρα τότε θα στείλει το πλαίσιο (frame) σε όλες τις θύρες. Οι βρόγχοι της γέφυρας (οι αναταραχές που δημιουργούνται) αναγνωρίζονται γρήγορα σαν το χειρότερο πράγμα που θα μπορούσε να συμβεί σε ένα δίκτυο με γέφυρες. Έτσι αναπτύχθηκε μια εύρωστη λύση. Το ζευγνύον δέντρο (Spanning Tree) λύνει το πρόβλημα αυτό μετακινώντας (ή κόβοντας) όλα τα περιττά μονοπάτια. Μειώνει την 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 . Σε κάθε τμήμα του τοπικού δικτύου (LAN) θα υπάρχει και μία γέφυρα με θύρα προορισμού όπου θα είναι υπεύθυνη για την προώθηση των πλαισίων (frames) στα υπόλοιπα τμήματα. Ο ρόλος αυτών των θυρών είναι σταθερός με τη χρήση του πρωτοκόλλου γεφυρών BPDUs (Bridge Protocol Data Units), όπου προέρχεται από κάθε γέφυρα και στέλνεται στις γειτονικές της γέφυρες. Χρησιμοποιώντας τις πληροφορίες στις BPDUs γέφυρες επιτρέπει στο σύστημα να καθορίζει τους ρόλους που θα παίξουν όλες οι θύρες του. Όταν οι ρόλοι είναι ξεκάθαροι, οι θύρες που δεν ανήκουν στις θύρες της κυρίως γέφυρας ή στις θύρες προορισμού τοποθετούνται σε μια περιοχή μπλοκαρίσματος. Οι κυρίως θύρες και οι προορισμού ξεκινούν μια προμελετημένα μακράς διάρκειας διαδικασία (τυπικά 30 δευτερόλεπτα) για προετοιμασία προώθησης τους προς τα έξω.

# Η Γλώσσα Προγραμματισμού C

Η **C** είναι μια γενικής χρήσης διαδικαστική γλώσσα προγραμματισμού. Αναπτύχθηκε στις αρχές της δεκαετίας 1970-1980 από τον **Dennis Ritchie** με στόχο τη χρήση της για την ανάπτυξη του λειτουργικού συστήματος UNIX. Η C είναι μια σχετικά μινιμαλιστική γλώσσα προγραμματισμού.

Η C αναπτύχθηκε στα **AT&T Bell Labs** ανάμεσα στο 1969 και το 1973, αλλά σύμφωνα με τον D. Ritchie, η πιο δημιουργική περίοδος υπήρξε το 1972. Η "νέα γλώσσα" ονομάστηκε "C" λόγω του ότι πολλά από τα χαρακτηριστικά της προήλθαν από μια παλαιότερη γλώσσα, η οποία ονομαζόταν "B". Οι πηγές δεν επιτρέπουν την πλήρη εξακρίβωση για την προέλευση του ονόματος "B".

Ανάμεσα στους σχεδιαστικούς στόχους που έπρεπε να καλύψει η γλώσσα περιλαμβανόταν το ότι θα μπορούσε να μεταγλωττιστεί (να γίνεται compile) άμεσα με τη χρήση **single-pass compiler** , δηλαδή ότι θα απαιτούνταν μόνο ένας μικρός αριθμός από **εντολές** (instructions) σε **γλώσσα μηχανής** (machine language) για κάθε βασικό στοιχείο της, χωρίς εκτεταμένη run-time υποστήριξη. Ως αποτέλεσμα, είναι δυνατό να γραφτεί κώδικας σε C σε χαμηλό επίπεδο (low level) προγραμματισμού με ακρίβεια ανάλογη της **συμβολικής γλώσσας** (**Assembly language**). Γι'αυτό και η C ορισμένες φορές αποκαλείται και " high-level assembly " ή " portable assembly " .

Μέχρι το 1973, η C είχε γίνει αρκετά ισχυρή και αποτελεσματική, ώστε το μεγαλύτερο μέρος του πυρήνα του UNIX (UNIX kernel), γραμμένο αρχικά σε PDP-11/20 assembly, επανεγγράφηκε σε C. Ήταν ένας από τους πρώτους πυρήνες που υλοποιήθηκε σε μια γλώσσα διαφορετική της assembly .

Έτσι από τότε χρησιμοποιείται ευρύτατα, και ιδιαίτερα για ανάπτυξη **προγραμμάτων συστήματος** (system software) αλλα και για **απλές εφαρμογές**. Οι λόγοι της ραγδαίας ανάπτυξης της συγκεκριμένης γλώσσας προγραμματισμού είναι η **ταχύτητα** της, καθώς και η **διαθεσιμότητα** της στα περισσότερα σημερινά **λειτουργικά συστήματα**.

Η γλώσσα **C** είναι μία από τις παλαιότερες γλώσσες και εξακολουθεί να χρησιμοποιείται αρκετά και στις μέρες μας .

## Λεπτομέρειες για την Ιστορία της γλώσσας C

### K&R C

Το 1978, ο **Dennis Ritchie** και ο **Brian Kernighan** δημοσίευσαν την πρώτη έκδοση του *"The C Programming Language"*. Το συγκεκριμένο βιβλίο, γνωστό στους προγραμματιστές της C ως *"***K&R***",* χρησίμευσε πολλά χρόνια ως ένας ανεπίσημος ορισμός της γλώσσας. Η έκδοση της C που περιγράφει αναφέρεται συνήθως ως " K&R C " ή " Common C ". ( Η δεύτερη έκδοση του βιβλίου καλύπτει το μεταγενέστερο πρότυπο ANSI για τη C (ANSI C standard) ) . Το K&R εισήγαγε στη γλώσσα τα χαρακτηριστικά που φαίνονται στο πιν. 8 .

| |
|---|
| struct data types |
| long int data type |
| unsigned int data type |
| Ο τελεστής (operator) =+ αλλάχθηκε σε += για να αφαιρεθεί η αμφισβήτηση σημαντικότητας (semantic ambiguity) που δημιουργούνταν με κατασκευές όπως i=+10, που μπορούσε να μεταφραστεί είτε i =+ 10 είτε i = +10. |

Πιν.8

Η K&R C συχνά λογίζεται ως το βασικό μέρος της γλώσσας που πρέπει να υποστηρίζει ένας C compiler. Για αρκετά χρόνια, ακόμη και μετά την εισαγωγή της ANSI C, θεωρούνταν ο "ελάχιστος συνήθης παρονομαστής" στον οποίο έπρεπε να προσαρμοστούν οι προγραμματιστές της C σε περιπτώσεις κατά τις οποίες ήταν επιθυμητή η μέγιστη μεταφερσιμότητα (portability), καθώς δεν είχε γίνει ενημέρωση (update) σε όλους τους compilers για πλήρη υποστήριξη της ANSI C, και διότι με προσοχή, ο κώδικας σε K&R C μπορούσε να γραφεί ώστε να είναι σύμφωνος και με το πρότυπο ANSI .

### ANSI C

Το 1983, το **American National Standards Institute (ANSI)** όρισε επιτροπή, τη X3J11, για να δώσει ένα σύγχρονο, πλήρη ορισμό της C. Μετά από μακρά και επίπονη επεξεργασία, το πρότυπο (standard) ολοκληρώθηκε το 1989 και επικυρώθηκε ως ANSI X3.159-1989 "Programming Language C " . Η συγκεκριμένη έκδοση της γλώσσας ονομάζεται συχνά ANSI C, ή ορισμένες φορές C89 (για να διαχωρίζεται από τη C99). Το 1990, το πρότυπο ANSI για τη C (με ορισμένες μικρές τροποποιήσεις)

υιοθετήθηκε από τον Οργανισμό Διεθνών Προτύπων ( International Organization for Standardization ( ISO ) ) ως ISO/IEC 9899:1990. Αυτή η έκδοση ονομάζεται C90. Έτσι , βλέπουμε ότι οι όροι "C89" και "C90" αναφέρονται στην ίδια γλώσσα.
Ένας από τους στόχους της διαδικασίας δημιουργίας του προτύπου ANSI για τη C ήταν να δημιουργήσει ένα υπερσύνολο της K&R C, το οποίο θα απορροφούσε πολλά χαρακτηριστικά που είχαν εισαχθεί στην πορεία. Παρόλα αυτά, η επιτροπή συμπεριέλαβε και ορισμένα νέα χαρακτηριστικά, όπως function prototypes (δανεισμένα από τη C++), και ένα πιο ικανό προεπεξεργαστή ( preprocessor ). Η σύνταξη για τους ορισμούς παραμέτρων άλλαξε επίσης, ώστε να αντικατοπτρίζει το στυλ της C++.

**C99**

Μετά τη διαδικασία καθορισμού του προτύπου ANSI, ο ορισμός της γλώσσας C παρέμενε σχετικά σταθερός για ορισμένο καιρό, ενώ παράλληλα η C++ συνέχιζε να αναπτύσσεται. Ο Normative Amendment 1 δημιούργησε μία νέα έκδοση της γλώσσας C το 1995, αλλά σπάνια είναι γνωστή. Ωστόσο, το πρότυπο επανεξετάστηκε προς το τέλος της δεκαετίας του '90, γεγονός που οδήγησε στην έκδοση του **ISO 9899:1999** το 1999. Το πρότυπο αυτό συχνά αναφέρεται ως "**C99**". Υιοθετήθηκε ως πρότυπο ANSI το Μάρτιο του 2000.
Ο GCC και μερικοί άλλοι C compilers υποστηρίζουν πλέον τα περισσότερα χαρακτηριστικά του C99. Ωστόσο, υπάρχει μικρότερη υποστήριξη από εταιρίες όπως η Microsoft και η Borland που εστίασαν περισσότερο στη C++, γιατί αυτή παρέχει παρόμοια λειτουργικότητα και συχνά ασύμβατους τρόπους ( π.χ. , η complex template class ).
Ο Brandon Bray από τη Microsoft είπε: "Σε γενικές γραμμές, έχουμε δει μικρές απαιτήσεις για πολλά χαρακτηριστικά του C99. Μερικά χαρακτηριστικά έχουν μεγαλύτερη ζήτηση από άλλα, και θα τη λάβουμε υπόψιν μας σε μελλοντικές releases εφόσον είναι συμβατά με τη C++." Ακόμη και ο GCC με την εκτεταμένη υποστήριξη του C99 δεν προσεγγίζει μια καθ'όλα συμβατή υλοποίηση. Ορισμένα χαρακτηριστικά-κλειδιά λείπουν ή δεν λειτουργούν σωστά.

## Λεπτομέρειες για την Λειτουργία της γλώσσας C

Η C είναι μία γλώσσα με βιβλιοθήκες οι οποίες περιέχουν κάποιες δικές τους συναρτήσεις. Επίσης είναι μια γλώσσα που μας επιβάλλει να χρησιμοποιούμε δικές μας συναρτήσεις σε μεγάλα προγράμματα.

Εχει μια κυρίως συνάρτηση τη main και μέσα από αυτή καλούμε τις υπόλοιπες που έχουμε δημιουργήσει ώστε να ενεργοποιηθούν. Καθε φορά που καλείται κάποια συνάρτηση μέσα από τη main εκτελεί τις εντολές που της έχουμε δώσει και επιστρέφει το αποτέλεσμα πίσω στη main και ελευθερώνει ότι χώρο δέσμευσε στη μνήμη.
Η C είναι μια γλώσσα που μας δίνει τη δυνατότητα να κατασκευάσουμε μονοι μας τις δικές μας συναρτήσεις. Στην ουσία μας αναγκάζει να τις κατασκευάσουμε γιατί δε μας τις παρέχει.

# *ΕΙΣΑΓΩΓΗ ΠΡΟΒΛΗΜΑΤΟΣ ΚΑΙ ΤΡΟΠΟΙ ΕΠΙΛΥΣΗΣ*

Το πρόβλημα των **ζευγνυόντων δέντρων εξαναγκασμένης διαμέτρου** (Diameter-Constrained Minimum Spanning Tree, DCMST) μπορεί να τεθεί ως ακολούθως: **με δεδομένο ένα μη κατευθυντικό 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 $3.342171n^{1/2}$, η δε διάμετρος με την μέγιστη πιθανότητα έχει τιμή $3.2015131n^{1/2}$.

## ΑΝΑΜΕΝΟΜΕΝΗ ΤΙΜΗ ΤΗΣ ΔΙΑΜΕΤΡΟΥ ΕΝΟΣ 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.

Για να υπολογίσουμε την αναμενομένη τιμή της διαμέτρου για όλα τα δέντρα n κόμβων, αριθμημένων κορυφών, μπορούμε να υλοποιήσουμε την πιο κάτω μέθοδο (μέθοδος Riordan). Αρχικά υπολογίζουμε το πλήθος των δέντρων με n κόμβους, με δεδομένη ρίζα και ύψος μέχρι h, χρησιμοποιώντας την ακόλουθη εξίσωση:

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

όπου:

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

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

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

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

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

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

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

Μετά τα παραπάνω, υπολογίζουμε τον αριθμό $\delta_n(d)$ 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 $\overline{d}_n$, το οποίο εκφράζει την αναμενομένη τιμή της διαμέτρου για ένα δέντρο αριθμημένων κορυφών με n κόμβους, δίνεται από την παρακάτω σχέση:

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

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

# *ΑΝΑΠΤΥΧΘΕΝΤΕΣ ΑΛΓΟΡΙΘΜΟΙ*

# *ΚΩΔΙΚΑΣ ΚΑΙ ΣΧΟΛΙΑΣΜΟΣ ΠΡΟΓΡΑΜΜΑΤΟΣ*

**ΓΕΝΙΚΑ:**

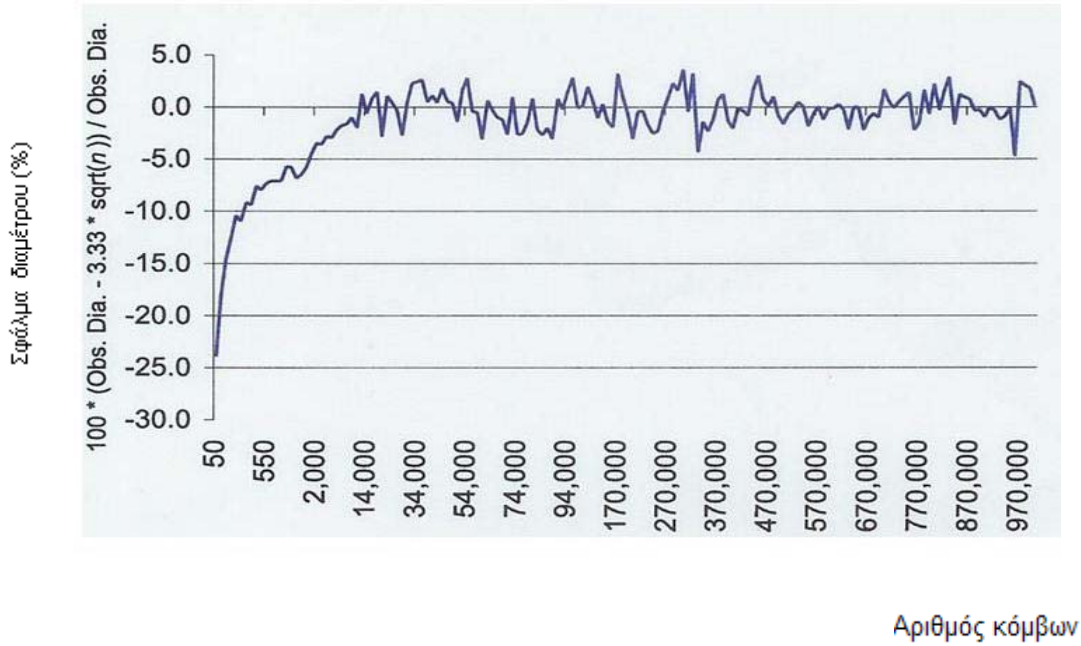
Στην εργασία που ακολουθεί αναπτύχθηκαν αλγόριθμοι χειρισμού γράφων εξ ολοκλήρου στην γλώσσα προγραμματισμού C. Η γλώσσα αυτή επιλέχθηκε κυρίως λόγω της απλότητας χειρισμού της, αλλά και επειδή μας έδινε την δυνατότητα να «πειραματιστούμε» στον προγραμματισμό ενός σχετικά περίπλοκου θέματος με την χρήση ενός απλού εργαλείου. Ο προγραμματισμός με την χρήση μιας αντικειμενοστραφούς γλώσσας (C++ για παράδειγμα), πιθανόν να αποδειχθεί ευκολότερος στο αντικείμενο που εξετάζουμε.

Οι αλγόριθμοι που αναφέρονται παρακάτω, αναπτύχθηκαν από μηδενική βάση. Δόθηκε ιδιαίτερο βάρος στην πρωτοτυπία τους κατά το δυνατό και όχι τόσο στην αποτελεσματικότητά τους. Έτσι, για μεγάλο αριθμό κόμβων οι αλγόριθμοι γίνονται αργοί, μπορούν να βελτιωθούν όμως σημαντικά εάν αλλάξει η λογική δημιουργίας του γράφου, η οποία τώρα βασίζεται στην εξέταση όλων των δυνατών περιπτώσεων για κάποιο σύνολο κόμβων. Στην εργασία αυτή έγινε προσπάθεια επιβεβαίωσης κάποιων από τα αποτελέσματα που παρουσιάζονται στο παρακάτω άρθρο:

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

Στην εργασία μας εξετάστηκαν γράφοι με 4, 5 και 6 κόμβους. Η πιθανή εξέταση πολύ περισσοτέρων κόμβων είναι προφανώς δυνατή με τη χρήση των αναπτυχθέντων αλγορίθμων και την τροποποίηση κάποιων σημείων τους, όμως κάτι τέτοιο θα ήταν εξαιρετικά χρονοβόρο. Γι' αυτό περιοριστήκαμε σε λίγους κόμβους, λαμβανομένου υπ' όψιν και του σκοπού που προαναφέρθηκε, δηλαδή μόνο την επιβεβαίωση των αποτελεσμάτων. 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 αναμενομένη 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 αναφέρεται και στο άρθρο στο οποίο βασίσαμε την εργασία, αυτό συμβαίνει για αριθμό κόμβων μεγαλύτερο από 1100. 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 σχήμα:




Έτσι, θεωρούμε ότι η τιμή που επιτύχαμε είναι ικανοποιητική, δεδομένης μάλιστα της ανάπτυξης ενός αριθμού πρωτοτύπων αλγορίθμων.

# *ΚΩΔΙΚΑΣ*

```
#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, diameter=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)
			{
				diameter++;
//				getch();
//				tree_m_display(2);
//				tree_m_display(3);
//				tree_m_display(4);
//				tree_display();
			}
//		  paths_display();
			plithos++;
}
	}
	printf ("\nπλήθος Δένδρων=%8ld\n", plithos);
	printf ("πλήθος Δένδρων Διαμέτρου 2 = %d\n", diameter);
}
```

```
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 DWSE XARAKTHRA\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 DWSE XARAKTHRA\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 DWSE XARAKTHRA\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 DWSE XARAKTHRA\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;
}
```

# ***ΣΧΟΛΙΑ***

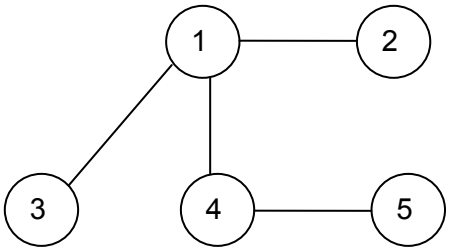
**1.** Η περιγραφή ενός γράφου στην εργασία μας γίνεται ακολουθώντας την μέθοδο του πίνακα γειτονικών κορυφών. Όπως αναφέρθηκε στην αρχή του μέρους αυτού, η μελέτη των γράφων είναι ισοδύναμη με την μελέτη δέντρων αριθμημένων κορυφών χωρίς βάρη ακμών. Έτσι, ένας γράφος Ν κορυφών παριστάνεται με την χρήση ενός τετραγωνικού πίνακα ακεραίων NxN. Στο πρόγραμμα που προηγείται, 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 ακυκλικός 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 Ν κορυφές.

Για παράδειγμα, έστω ότι διαθέτουμε ένα γράφο 5 κορυφών. Ο πίνακας του σχ. 22(α) παριστάνει το δέντρο του σχ. 22(β). 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. 22(α)




Σχ. 22(β)

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.

**2.** Το πρόγραμμα αρχικά δημιουργεί γράφους. Όπως προαναφέρθηκε, 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 υπ’ όψη 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 | κλπ |

Η συνάρτηση **hex_to_bin( )** μετατρέπει σε δυαδικό τον δεκαεξαδικό αριθμό που μας δείχνει τον συνδυασμό “0” και “1” που 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.

**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. 22(α) είναι ο πίνακας του σχ. 23(α), 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. 23(α)

| | | |
|---|---|---|
| 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. 23(β)

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 Ν(Ν-1)/2 γραμμές και 3 στήλες. Ο pathrev, για το παράδειγμα του σχ. 1(α) που προαναφέρθηκε, φαίνεται στο σχ. 23(β).

Με τους πίνακες path και pathrev σχετίζονται οι συναρτήσεις:

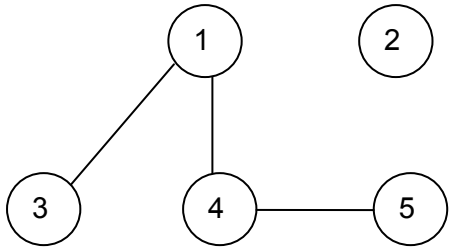
- **path_init ( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα path.
- **pathrev_init ( ) :** θέτει τιμή μηδέν σε όλες τις θέσεις του πίνακα pathrev.
- **paths_display ( ) :** εμφανίζει στην οθόνη τα περιεχόμενα των πινάκων path και pathrev.
- **path_construction ( ) :** δημιουργεί τον πίνακα path.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

**4.** 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.

**5.** Το πρόγραμμα χρησιμοποιεί την συνάρτηση **one_isolated( )**. Η συνάρτηση ελέγχει κατά πόσον υπάρχει μεμονωμένος κόμβος 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 άθροισμα των στοιχείων της ίσο με μηδέν. Η συνάρτηση επιστρέφει τιμή 1 εάν υπάρχει μεμονωμένος κόμβος στον γράφο και τιμή μηδέν εάν τέτοιος κόμβος δεν υπάρχει. Για παράδειγμα, ο πίνακας γειτονικών κορυφών του σχ. 24(α) είναι αυτός που αντιστοιχεί στον γράφο του σχ. 24(β).

| | 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. 24(α)

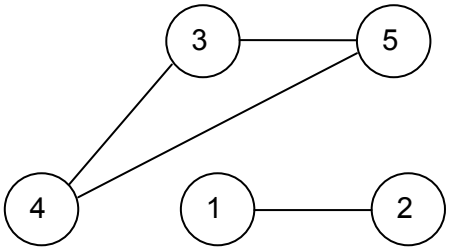



Σχ. 24(β)

**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 του σχ. 25(α) είναι αυτός που αντιστοιχεί στον γράφο του σχ. 25(β), 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. 25(α)




Σχ. 25(β)

Ο πίνακας path του παραπάνω γράφου είναι αυτός του σχ. 25(γ). :

| | | |
|---|---|---|
| 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. 25(γ)

Η ύπαρξη μεμονωμένου ζεύγους στον γράφο σημαίνει δύο κόμβους, οι οποίοι συνδέονται μόνο μεταξύ τους και κανείς από τους δύο δεν συνδέεται και με κάποιον άλλο. Ο πίνακας path που αναφέρθηκε πιο πάνω στην παράγραφο 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 πίνακα του σχ. 25(γ) για παράδειγμα, ο οποίος αντιστοιχεί στον γράφο του σχ. 25(β), 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, άρα τα ζεύγη (3,4) και (3,5) δεν είναι μεμονωμένα.

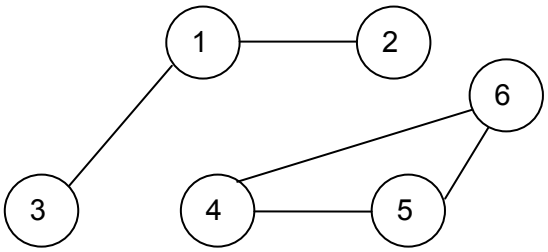
Η συνάρτηση έχει ιδιαίτερη χρησιμότητα σε περιπτώσεις όπως αυτή του σχ. 25(β). Στο σχήμα αυτό παρατηρούμε ότι εκτός από το μεμονωμένο ζεύγος, ο υπόλοιπος γράφος είναι κλειστός. Εάν δεν υπήρχε η συνάρτηση two_isolated( ), ο γράφος θα εθεωρείτο κανονικός, αφού έχει τον κατάλληλο αριθμό ακμών.

**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 του σχ. 26(α) είναι αυτός που αντιστοιχεί στον γράφο του σχ. 26(β).

| | 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. 26 (α)




Σχ. 26(β)

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 (βλ. και ανωτέρω παράγραφο 6 για την συνάρτηση two_isolated( )).

Ο πίνακας path του πιο πάνω γράφου είναι αυτός του σχ. 26(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. 26(γ)

| | | |
|---|---|---|
| 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. 26(δ)

Αντίστοιχος με τον path είναι ο πίνακας pathrev. Για τον γράφο του σχ. 26(β) ο pathrev φαίνεται στο σχ. 26(δ). 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

**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 το 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.

**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.

**10.** Η συνάρτηση **tree_m_display( )** δέχεται ως όρισμα ένα 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.

**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 όπως αναφέρθηκε πιο πάνω, στην παράγραφο 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 προγραμματιστικής ευκολίας λοιπόν, χρησιμοποιείται η συνάρτηση **tree_modify( )**, η οποία μετατρέπει τα μη μηδενικά περιεχόμενα του κάθε πίνακα σε "1".

**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.

# ΒΙΒΛΙΟΓΡΑΦΙΑ


1. **“Random-tree diameter and the diameter constrained MST”**, Ayman Abdalla and Narsingh Deo, International Journal of Computer Mathematics, 2002, Vol. 79(6), pp. 651-663.
2. **“Δομές Δεδομένων με C”**, Ν. Μισυρλής, ISBN 960-92031-1-6.
3. **“Εισαγωγή στην Ανάλυση & Σχεδίαση Αλγορίθμων”,** Anany Levitin, Εκδόσεις Τζιόλα, ISBN 978-960-418-143-8.
4. **“Algorithms + Data Structures = Programs”,** Niklaus Wirth, Prentice-Hall, Inc., 1976, ISBN 0-13-022418-9.
5. **“ΑΛΓΟΡΙΘΜΟΙ Σχεδιασμός και Ανάλυση”,** Παναγιώτης Μποζάνης, Εκδόσεις Τζιόλα, 2003, ISBN 960-418-014-2.
6. **“ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ Έννοιες, Τεχνικές και Αλγόριθμοι”**, Γεώργιος Γεωργακόπουλος, Π.Ε.Κ., 2002, ISBN 960-524-125-0.