Γράφοι

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

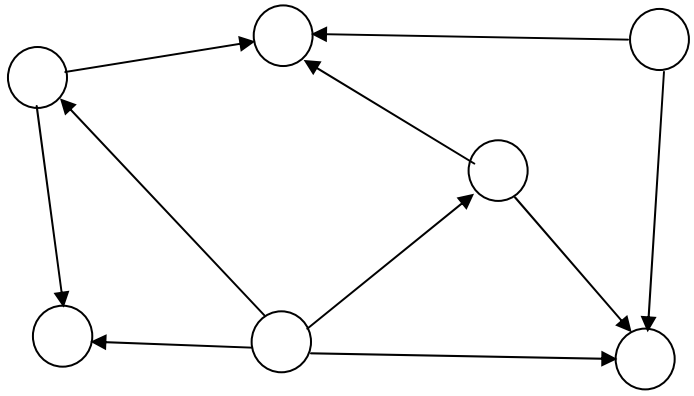
Γράφοι - ορισμοί και υλοποίηση

Τοπολογική Ταξινόμηση

Διάσχιση Γράφων

Γράφοι

- Η πιο γενική μορφή δομής δεδομένων, με την έννοια ότι όλες οι προηγούμενες δομές μπορούν να θεωρηθούν ως περιπτώσεις γράφων.
- Ένα γράφος αποτελείται από
  - ένα σύνολο V *κορυφών* (vertices), ή σημείων, ή κόμβων , και
  - ένα σύνολο E *ακμών* (edges), ή τόξων, ή γραμμών. Μια ακμή είναι ένα ζεύγος (u,v) από κορυφές.
- Παράδειγμα γράφου:



Γράφοι

- Οι γράφοι προσφέρουν μια χρήσιμη μέθοδο για τη διατύπωση και λύση πολλών προβλημάτων, σε
  - δίκτυα και συστήματα τηλεπικοινωνιών (π.χ. το Internet),
  - χάρτες - επιλογή δρομολογίων,
  - προγραμματισμό εργασιών (scheduling),
  - ανάλυση προγραμμάτων (flow charts).

- Η θεωρία των γράφων θεωρείται ότι ξεκίνησε από τον Euler στις αρχές του 18ου αιώνα (1736).

Ορισμοί

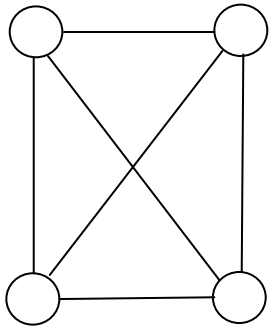
- Ένας γράφος ονομάζεται ***κατευθυνόμενος*** (directed graph, digraph) αν κάθε μια από τις ακμές του είναι προσανατολισμένη προς μία κατεύθυνση.
- Ένας γράφος ονομάζεται ***μη-κατευθυνόμενος*** (undirected) αν οι ακμές του δεν είναι προσανατολισμένες.
- Αν (u,v) είναι ακμή τότε λέμε ότι οι κορυφές u και v είναι *γειτονικές* (adjacent) ή ότι *γειτνιάζουν*.
- ***Μονοπάτι*** ή ***διαδρομή*** (path) ενός γράφου μήκους n, είναι μια ακολουθία κόμβων $v_0, v_1, \ldots, v_n$, όπου για κάθε i , $0 \leq i < n$, (v_i , v_{i+1}) είναι ακμή του γράφου. ***Μήκος*** ενός μονοπατιού είναι ο αριθμός ακμών που περιέχει.
- Μια διαδρομή ενός γράφου ονομάζεται ***απλή*** (simple) αν όλες οι κορυφές της είναι διαφορετικές μεταξύ τους, εκτός από την πρώτη και την τελευταία οι οποίες μπορούν να είναι οι ίδιες.
- ***Κύκλος*** (cycle) ονομάζεται μια διαδρομή με μήκος >1 που ικανοποιεί $v_0 = v_n$.

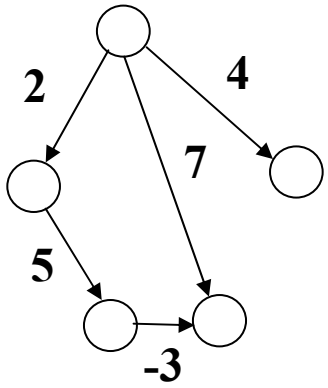
Ορισμοί

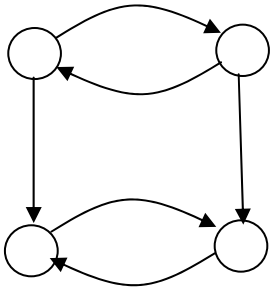
- Ένας γράφος που δεν περιέχει κύκλους ονομάζεται ***άκυκλος*** (acyclic)
- Έστω G=(V,E) και G΄= (V΄, E΄) γράφοι, όπου V΄⊆V και E΄⊆ E. Τότε ο γράφος G΄ είναι ***υπογράφος*** (subgraph) του γράφου G.
- Η ***απόσταση*** δύο κορυφών είναι το μήκος της συντομότερης διαδρομής που οδηγεί από τη μια κορυφή στην άλλη.
- Ένας μη κατευθυνόμενος γράφος λέγεται ***συνεκτικός*** (connected) αν για κάθε ζευγάρι κορυφών υπάρχει διαδρομή που τις συνδέει.
- Ένας κατευθυνόμενος γράφος που ικανοποιεί την ίδια ιδιότητα ονομάζεται ***ισχυρά συνεκτικός*** (strongly connected). Αν ο μη-κατευθυνόμενος γράφος στον οποίο αντιστοιχεί είναι συνεκτικός, τότε ο γράφος ονομάζεται ***ελαφρά συνεκτικός*** (weakly connected).

Ορισμοί

- Πόσες ακμές μπορεί να έχει ένας γράφος με n κορυφές;
- Ένας γράφος λέγεται ***αραιός*** (sparse) αν ο αριθμός των ακμών του είναι της τάξης O(n), όπου n είναι ο αριθμός κορυφών του, διαφορετικά λέγεται ***πυκνός*** (dense).
- Συχνά συσχετίζουμε κάθε ακμή ενός γράφου με κάποιο βάρος (weight). Τότε ο γράφος ονομάζεται ***γράφος με βάρη*** (weighted graph).
- Ποιες ιδιότητες ικανοποιούν οι πιο κάτω γράφοι;







Αναπαράσταση γράφων

Αναπαράσταση γράφων με πίνακες γειτνίασης (adjacency matrix)

- Ένας γράφος G=(V,E) με n κορυφές μπορεί να αναπαρασταθεί ως ένας n×n πίνακας που περιέχει τις τιμές 0 και 1, και όπου
  - αν η (i,j) είναι ακμή τότε A[(i,j)]=1, διαφορετικά A[(i,j)]=0.
- Αν ο γράφος είναι γράφος με βάρη, και το βάρος κάθε ακμής είναι τύπου t, τότε για την αναπαράσταση του γράφου μπορεί να χρησιμοποιηθεί πίνακας τύπου t με
  - A[(i,j)] = βάρος(i,j), αν υπάρχει ακμή (i,j)
  - A[(i,j)] = ∞, αν δεν υπάρχει ακμή (i,j)
- Αυτή η αναπαράσταση απαιτεί χώρο $O(n^2)$, όπου n=|V|.
- Αν ο γράφος είναι αραιός η μέθοδος οδηγεί σε σπάταλη χώρου.

Αναπαράσταση γράφων

Αναπαράσταση γράφων με λίστες γειτνίασης (adjacency lists)

- Ένας γράφος G=(V,E) αναπαρίσταται ως ένας μονοδιάστατος πίνακας A.
- Για κάθε κορυφή ω, A[ω] είναι ένας δείκτης σε μια συνδεδεμένη λίστα στην οποία αποθηκεύονται οι κορυφές που γειτνιάζουν με την ω.
- Η μέθοδος απαιτεί χώρο O(|V|+ |E|).
- Επιτυγχάνεται εξοικονόμηση χώρου για αραιούς γράφους.
- Στην περίπτωση γράφων με βάρη στη λίστα γειτνίασης αποθηκεύουμε επίσης το βάρος κάθε ακμής.

Δομές Υλοποίησης

- Πίνακας Γειτνίασης

```
struct graph1{
        int         matrix[max][max];
        int         size;
}
```

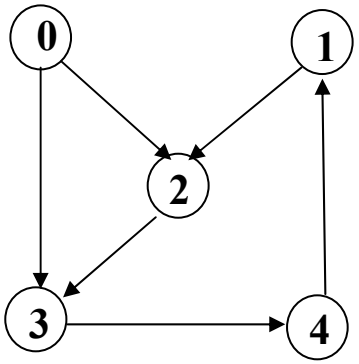
- Λίστα Γειτνίασης

```
struct node{
  int vertex;
  struct node *next ;
}
```

```
struct graph2{
   struct node *head[max];
   int          size; }
```

Αναπαράσταση γράφων

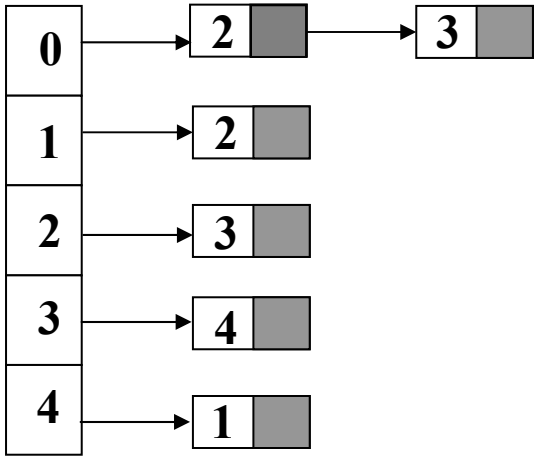
Γράφος



Πίνακας Γειτνίασης

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

Λίστα Γειτνίασης



Διάσχιση Γράφων

- Αν θέλουμε να επισκεφτούμε όλους τους κόμβους ενός γράφου μπορούμε να χρησιμοποιήσουμε έναν από πολλούς τρόπους, οι οποίοι διαφέρουν στη σειρά με την οποία εξετάζουν τους κόμβους.
- Διαδικασίες διάσχισης χρησιμοποιούνται για τη διακρίβωση ύπαρξης μονοπατιού μεταξύ δύο κόμβων κ.α.
- Έχουν πολλές εφαρμογές.

Depth-First Search

- Γενίκευση της προθεματικής διάσχισης δένδρων: Ξεκινώντας από ένα κόμβο v, επισκεπτόμαστε πρώτα τον v και ύστερα καλούμε αναδρομικά τη διαδικασία στο καθένα από τα παιδιά του.
- Πως επηρεάζει η ύπαρξη κύκλων την πιο πάνω ιδέα;
- Θα διατηρήσουμε ένα πίνακα `Visited` ο οποίος θα κρατά πληροφορίες ως προς το ποιους κόμβους έχουμε επισκεφθεί ανά πάσα στιγμή.

Διαδικασία Προθεματικής Διάσχισης

```
DepthFirstSearch(graph G,vertex v){
   for each vertex w in G
       visited[w] = False;
   DFS(v);
}

DFS(vertex v){
   visited[v] = True;
   for each w adjacent to v
       if (visited[w]==False)
             DFS(w)
}
```

Εξειδίκευση DFS για πίνακες γειτνίασης

```
DepthFirstSearch(struct graph1 *G, int v){
   for (w=1; w <= G->size; w++)
       visited[w] = 0;
   DFS(G, visited, v);
}

DFS(struct graph1 *G, int *visited[max], int v){
   visited[v] = 1;
   for (w=1; w <= G->size; w++)
       if (G->matrix[v,w] == 1 && visited[w]==0)
             DFS(G, visited, w)
}
```

Χρόνος Εκτέλεσης: $O(|V|^2)$

Εξειδίκευση DFS για λίστες γειτνίασης

```
DepthFirstSearch(struct graph2 *G, int v){
   for (w=1; w <= G->size; w++)
       visited[w] = 0;
   DFS(G, visited, v);
}

DFS(struct graph2 *G, int *visited[max], int v){
   visited[v] = 1;
   for (p = G->head[v]; p != NULL; p = p->next)
       w = p->vertex;
       if (Visited[w]==0)
             DFS(G, visited, w)
}
```

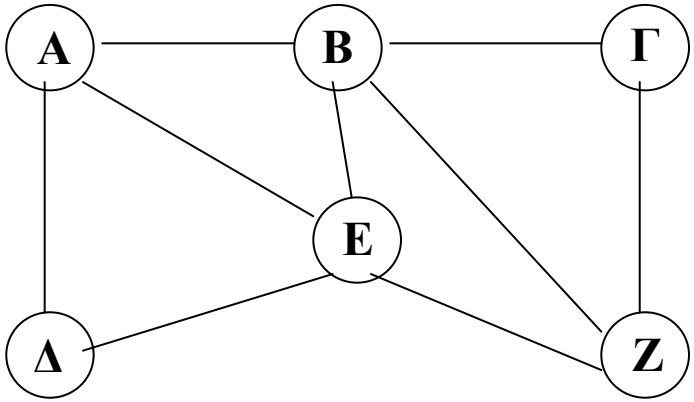
Χρόνος Εκτέλεσης: O(|V|+|E|)

Προθεματική Διάσχιση

- Αν ο γράφος δεν είναι συνεκτικός αυτή η στρατηγική πιθανό να αγνοήσει μερικούς κόμβους. Αν ο στόχος μας είναι να επισκεφθούμε όλους τους κόμβους τότε μετά το τέλος της εκτέλεσης του DFS(v) θα πρέπει να ελέγξουμε τον πίνακα Visited να βρούμε τους κόμβους που δεν έχουμε επισκεφθεί και να καλέσουμε σε αυτούς τη διαδικασία DFS:

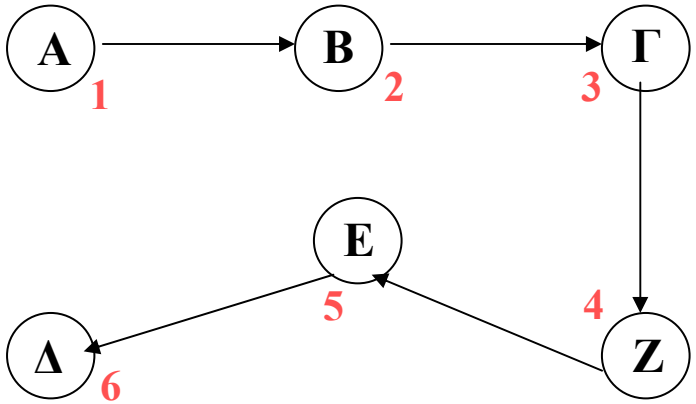
```
DepthFirstSearch(graph G,vertex v){
  for each vertex w in G
      Visited[w] = False;
  DFS(v);
  for each vertex w in G
      if (Visited[w]== False)
         DFS(w);
}
```

Παράδειγμα Depth-First-Search 1

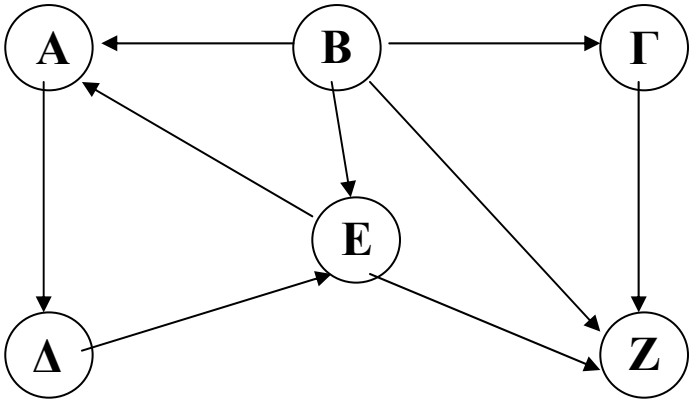


	A	B	Γ	Δ	E	Z	
	0	1	0	1	1	0	A
	1	0	1	0	1	1	B
	0	1	0	0	0	1	Γ
	1	0	0	0	1	0	Δ
	1	1	0	1	0	1	E
	0	1	1	0	1	0	Z

DepthFirstSearch(G, A)

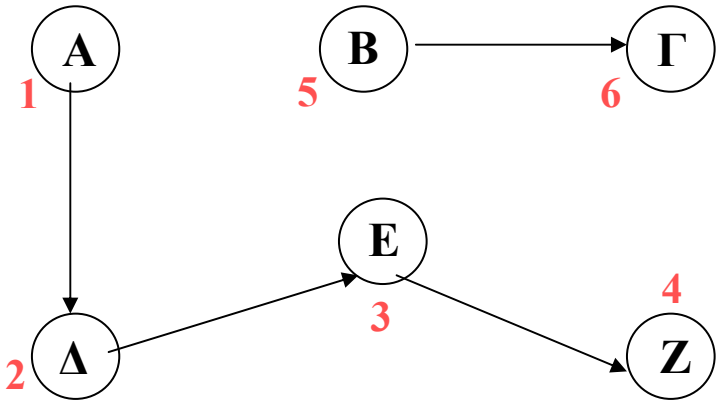


Παράδειγμα Depth-First-Search 2



	A	B	Γ	Δ	E	Z	
	0	0	0	1	0	0	A
	1	0	1	0	1	1	B
	0	0	0	0	0	1	Γ
	0	0	0	0	1	0	Δ
	1	0	0	0	0	1	E
	0	0	0	0	0	0	Z

DepthFirstSearch(G, A)

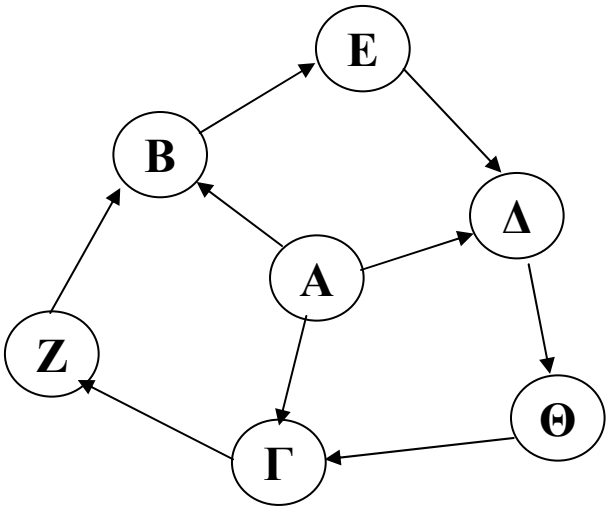


Μερικά Σχόλια

- Η διαδικασία καλείται σε κάθε κόμβο το πολύ μια φορά.
- Χρόνος Εκτέλεσης: O(|V| + |E|), δηλαδή γραμμικός ως προς τον αριθμό των ακμών και κορυφών.
- ***Αρίθμηση DFS*** των κορυφών ενός γράφου ονομάζεται η σειρά με την οποία επισκέπτεται η διαδικασία DepthFirstSearch τις κορυφές του γράφου.
- Η διαδικασία μπορεί να κληθεί και για μη-κατευθυνόμενους και για κατευθυνόμενους γράφους.
- Αντί με αναδρομή η διαδικασία μπορεί να υλοποιηθεί, ως συνήθως, με τη χρήση στοιβών.

Breadth-First Search

- Ξεκινώντας από ένα κόμβο v, επισκεπτόμαστε πρώτα το v, ύστερα τους κόμβους που γειτνιάζουν με τον v, ύστερα τους κόμβους που βρίσκονται σε απόσταση 2 από τον v, και ούτω καθεξής.



Output: **Α Β Δ Γ Ε Θ Ζ**

Διαδικασία Breadth-First Search

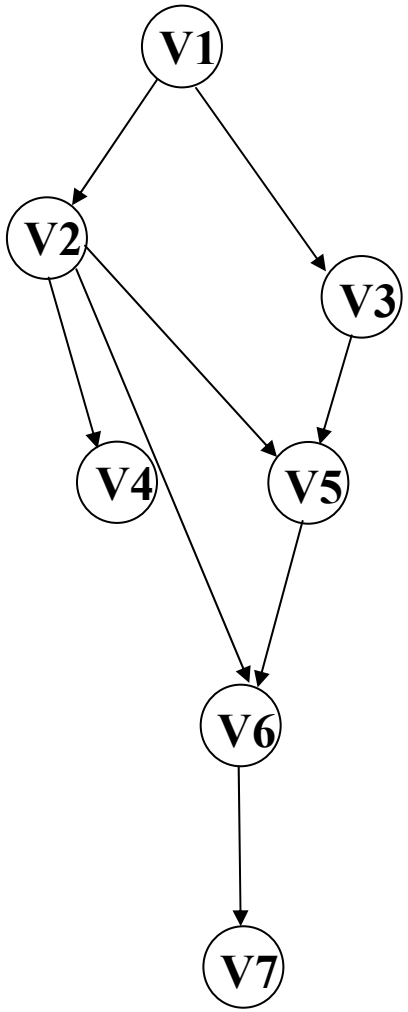
```
BFSearch(graph G, vertex v){
   Q=MakeEmptyQueue();
   for each w in G
    Visited[w]=False;

Visited[v]= True;
Enqueue(v,Q);

while (!IsEmpty(Q)){
       w = Dequeue(Q);
       Visit(w);
       for each u adjacent to w
              if (Visited[u]=False)
                     Visited[u]=True;
                     Enqueue(u,Q);
}
```

Χρόνος Εκτέλεσης:

Τοπολογική Ταξινόμηση (Topological Sort)

- Δίνεται ένα σύνολο εργασιών και θέλουμε να ορίσουμε τη σειρά με την οποία πρέπει να εκτελέσει τις εργασίες ένας επεξεργαστής, δεδομένης της ύπαρξης περιορισμών ως προς την προτεραιότητά τους.
- Κάθε εργασία έχει ένα σύνολο προαπαιτούμενων εργασιών, δηλαδή δεν μπορεί να εκτελεσθεί προτού συμπληρωθεί κάθε μια από τις προαπαιτούμενες.
- Μπορούμε να παραστήσουμε το πρόβλημα ως έναν κατευθυνόμενο γράφο:
  - Οι κορυφές του γράφου αντιστοιχούν σε κάθε μια από τις εργασίες, και
  - η ύπαρξη ακμής από την κορυφή Α στην κορυφή Β δηλώνει ότι η εργασία Α πρέπει να εκτελεστεί πριν από τη Β.
- Τοπολογική ταξινόμηση του γράφου είναι μια σειρά των κορυφών του, $v_1, \ldots, v_n$, ώστε αν (v_i, v_j) είναι ακμή του γράφου τότε $i<j$.

Παράδειγμα



Τοπολογικές Ταξινομήσεις του γράφου:

V1, V2, V4, V3, V5, V6, V7

V1, V2, V3, V5, V6, V7, V4

V1, V3, V2, V5, V6, V4, V7

…

Αλγόριθμος για Τοπολογική Ταξινόμηση

- Ο ***βαθμός εισόδου*** (in-degree) ενός κόμβου είναι ο αριθμός των ακμών που καταλήγουν στον κόμβο. (Στο πρόβλημα μας, ο αριθμός των προαπαιτούμενων εργασιών)
- Για κάθε κορυφή u έστω I[u] ο βαθμός εισόδου της u.
- Επαναλαμβάνουμε τα εξής βήματα:
  1. διαλέγουμε κορυφή A με I[A]=0,
  2. τυπώνουμε την A,
  3. για όλες τις κορυφές B, με (A,B) μειώνουμε την τιμή I[B] κατά 1.
- Πως μπορούμε να διακρίνουμε την ύπαρξη κύκλων;

Προσπάθεια 1

```
topSort1( graph G ){
  int I[|V|];
  for each vertex u
    I[u]=0;
  for each vertex u
    for each edge (u,v)
              I[v]++;
  for (i=1; i <= |V|; i++){
    v = FindVertexOfIndegree0;
    if (v == Not_a_Vertex)
          Error("Graph has a cycle");
          return;
    output v;
    for each edge (v,w)
        I[w]--;
  }}
```

Χρόνος Εκτέλεσης:

Προσπάθεια 2 (με χρήση βοηθητικής δομής)

```
topologicalSort( graph G ){
  queue Q;
  int I[|V|];

  for each vertex u
       I[u]=0;
  for each vertex u
       for each edge (u,v)
            I[v]++;
  for each vertex u
       if (I[u]==0) Enqueue(u, Q);
  while (! IsEmpty(Q)){
       u = Dequeue(Q);
       output u;
       for each (u,v)
              I[v]--;
              if (I[v]==0) Enqueue (v, Q);
}}
```

Χρόνος Εκτέλεσης: