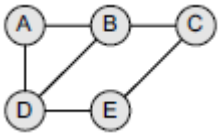
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ
ΜΑΘΗΜΑ: ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΕΡΓΑΣΤΗΡΙΟ 9 – ΣΗΜΕΙΩΣΕΙΣ

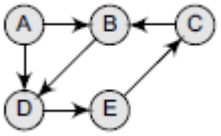
Γράφοι (Graphs)

Οι **γράφοι (graphs)** είναι μια δομή δεδομένων που χρησιμοποιείται για να αναπαραστήσει την μαθηματική έννοια των γράφων. Πρόκειται για ένα σύνολο από **κορυφές / κόμβους (nodes) και ακμές (edges)** που συνδέουν τους κόμβους αυτούς. Συχνά, οι γράφοι θεωρούνται μια γενίκευση των δένδρων όπου αντί να έχουμε μια 'καθαρή' συσχέτιση πατρικού κόμβου με κόμβους παιδιά, υλοποιούμε μια πιο περίπλοκη διασύνδεση των κόμβων.

Ένας γράφος G είναι ένα διατεταγμένο σύνολο (V,E) όπου G(V) είναι το σύνολο των κόμβων του γράφου και G(E) είναι το σύνολο των ακμών. Η ακόλουθη εικόνα μας παρουσιάζει ένα παράδειγμα ενός γράφου.



Οι γράφοι μπορεί να είναι **κατευθυνόμενοι (directed) ή μη (undirected)**. Στους κατευθυνόμενους γράφους οι ακμές έχουν συγκεκριμένη κατεύθυνση, επιτρέποντας την 'κίνηση' μόνο προς την κατεύθυνση που δείχνουν. Στους μη κατευθυνόμενους γράφους, οι ακμές δεν έχουν κάποια συγκεκριμένη κατεύθυνση.



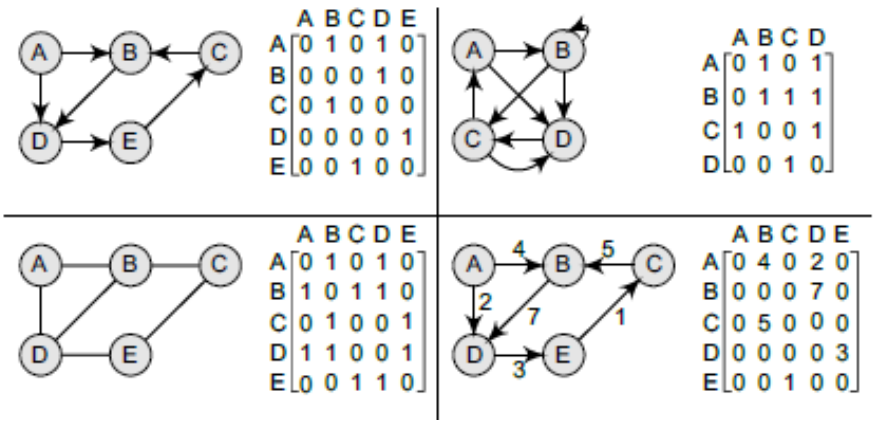
Για κάθε ακμή e=(u,v) που συνδέει τους κόμβους u, v, οι κόμβοι u,v ονομάζονται γειτονικοί και αποτελούν τα τελικά σημεία της ακμής. Όταν οι ακμές φέρουν ετικέτα στην οποία ανατίθενται κάποια δεδομένα (π.χ. ένας αριθμός), τότε ο γράφος ονομάζεται **γράφος με βάρη (weighted graph)**.

Για την αναπαράσταση των γράφων, στη βιβλιογραφία συναντάμε τρεις (3) τρόπους:

- Ακολουθιακά, χρησιμοποιώντας ένα **πίνακα γειτνίασης (adjacency matrix)**
- Αναπαράσταση με λίστα και πιο συγκεκριμένα με **λίστα γειτνίασης (adjacency list)**, όπου αποθηκεύονται για κάθε κόμβο οι γείτονές του σε μια λίστα
- Πολλαπλές λίστες γειτνίασης που αποτελεί μια επέκταση του δεύτερου αναπαράστασης των γράφων

Πίνακες Γειτνίασης

Οι πίνακες γειτνίασης αποθηκεύουν το ποιοι κόμβοι γειτνιάζουν με ποιους. Εξ' ορισμού, δύο κόμβοι γειτνιάζουν εφόσον υπάρχει ακμή που τους συνδέει. Για κάθε γράφο που έχει Ν κόμβους, ο πίνακας γειτνίασης θα έχει διάσταση ΝΧΝ. Οι γραμμές και οι στήλες του πίνακα θα έχουν ως ετικέτες τα ονόματα των κόμβων. Κάθε στοιχείο a_{ij} του πίνακα θα είναι ίσο με 1 αν οι κόμβοι i, j είναι γείτονες διαφορετικά θα είναι ίσο με 0. Ακολουθούν κάποια παραδείγματα πινάκων γειτνίασης.



Τα ακόλουθα συμπεράσματα μπορούν εύκολα να εξαχθούν:

- Για ένα γράφο που δεν έχει κύκλους, η διαγώνιος του πίνακα γειτνίασης θα είναι ίση με 0
- Ο πίνακας γειτνίασης ενός μη κατευθυνόμενου γράφου θα είναι συμμετρικός
- Η μνήμη που απαιτείται για την αποθήκευση του πίνακα γειτνίασης θα είναι $O(N^2)$ – N είναι το πλήθος των κόμβων
- Ο πίνακας γειτνίασης για ένα γράφο με βάρη, θα αποθηκεύει σε κάθε στοιχείο το βάρος της αντίστοιχης ακμής

Code 1

```
#include<stdio.h>

#include<conio.h>
int dir_graph();
int undir_graph();
int read_graph(int adj_mat[50][50], int n );

int main()
{
  int option;
  do
  {
      printf("\n A Program to represent a Graph by using an ");
                    printf("Adjacency Matrix method \n ");
                    printf("\n 1. Directed Graph ");
                    printf("\n 2. Un-Directed Graph ");
                    printf("\n 3. Exit ");
                    printf("\n\n Select a proper option : ");
                     scanf("%d", &option);
                    switch(option)
                    {
                     case 1 : dir_graph();
                       break;
           case 2 : undir_graph();
                       break;
           case 3 : return 0;
                    } // switch
   } while(1);
}

int dir_graph()
{
   int adj_mat[50][50];
   int n;
   int in_deg, out_deg, i, j;
   printf("\n How Many Vertices ? : ");
   scanf("%d", &n);
   read_graph(adj_mat, n);
   printf("\n Vertex \t In_Degree \t Out_Degree \t Total_Degree ");
   for (i = 1; i <= n ; i++ )
   {
```

```
    in_deg = out_deg = 0;
                for ( j = 1 ; j <= n ; j++ )
                {
      if ( adj_mat[j][i] == 1 )
        in_deg++;
                }
    for ( j = 1 ; j <= n ; j++ )
      if (adj_mat[i][j] == 1 )
        out_deg++;
      printf("\n\n %5d\t\t\t%d\t\t%d\t\t%d\n\n",i,in_deg,out_deg,in_deg+out_deg);
  }
  return 0;
}

int undir_graph()
{
  int adj_mat[50][50];
  int deg, i, j, n;
  printf("\n How Many Vertices ? : ");
  scanf("%d", &n);
  read_graph(adj_mat, n);
  printf("\n Vertex \t Degree ");
  for ( i = 1 ; i <= n ; i++ )
  {
    deg = 0;
    for ( j = 1 ; j <= n ; j++ )
      if ( adj_mat[i][j] == 1)
        deg++;
    printf("\n\n %5d \t\t %d\n\n", i, deg);
  }
  return 0;
}

int read_graph ( int adj_mat[50][50], int n )
{
  int i, j;
  int reply;
  for ( i = 1 ; i <= n ; i++ )
  {
    for ( j = 1 ; j <= n ; j++ )
    {
      if ( i == j )
      {
        adj_mat[i][j] = 0;
        continue;
      }
      printf("\n Vertices %d & %d are Adjacent ? (Y:1/N:0) :",i,j);
      scanf("%d", &reply);
      if ( reply == 1 )
        adj_mat[i][j] = 1;
```

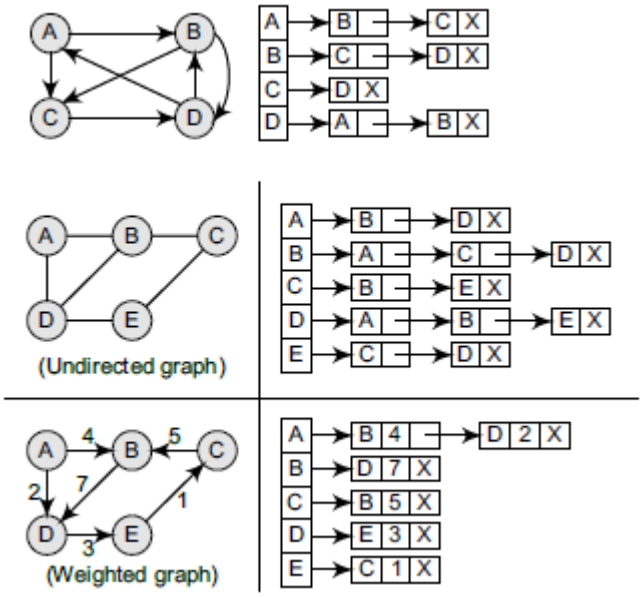
```
        else
            adj_mat[i][j] = 0;
                    }
    }
    return 0;
}
```

Λίστες Γειτνίασης

Η δομή αυτή αποτελείται από μια λίστα στην οποία αποθηκεύονται όλοι οι κόμβοι του γράφου. Ο κάθε ένας κόμβος στη συνέχεια συνδέεται μέσω μιας λίστας με τους γειτονικούς του κόμβους. Τα πλεονεκτήματα αυτής της τεχνικής είναι τα ακόλουθα:

- Μπορούμε εύκολα να ακολουθήσουμε τις λίστες
- Είναι προτιμότερη δομή όταν έχουμε να αναπαραστήσουμε αραιούς γράφους
- Η προσθήκη νέων κόμβων γίνεται πιο εύκολα σε αντίθεση με τους πίνακες γειτνίασης

Ακολουθούν κάποια παραδείγματα λιστών γειτνίασης.



Code 2

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

struct node
{
        char vertex;
        struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);

int main()
{
```

```
	struct node *Adj[10];
	int i, no_of_nodes;
	printf("\n Enter the number of nodes in G: ");
	scanf("%d", &no_of_nodes);
	for(i = 0; i < no_of_nodes; i++)
		Adj[i] = NULL;
	createGraph(Adj, no_of_nodes);
	printf("\n The graph is: ");
	displayGraph(Adj, no_of_nodes);
	deleteGraph(Adj, no_of_nodes);
	getch();
	return 0;
}

void createGraph(struct node *Adj[], int no_of_nodes)
{
	struct node *new_node, *last;
	int i, j, n, val;
	for(i = 0; i < no_of_nodes; i++)
	{
		last = NULL;
		printf("\n Enter the number of neighbours of %d: ", i);
		scanf("%d", &n);
		for(j = 1; j <= n; j++)
		{
			printf("\n Enter the neighbour %d of %d: ", j, i);
			scanf("%d", &val);
			new_node = (struct node *) malloc(1 * sizeof(struct node));
			new_node -> vertex = val;
			new_node -> next = NULL;
			if (Adj[i] == NULL)
				Adj[i] = new_node;
			else
				last -> next = new_node;
				last = new_node;
		}
	}
}

void displayGraph (struct node *Adj[], int no_of_nodes)
{
	struct node *ptr;
	int i;
	for(i = 0; i < no_of_nodes; i++)
	{
		ptr = Adj[i];
		printf("\n The neighbours of node %d are:", i);
		while(ptr != NULL)
		{
			printf("\t%d", ptr -> vertex);
```

```
                ptr = ptr -> next;
            }
        }
}

void deleteGraph (struct node *Adj[], int no_of_nodes)
{
        int i;
        struct node *temp, *ptr;
        for(i = 0; i <= no_of_nodes; i++)
        {
            ptr = Adj[i];
            while(ptr != NULL)
            {
                temp = ptr;
                ptr = ptr -> next;
                free(temp);
            }
            Adj[i] = NULL;
        }
}
```

Αλγόριθμοι Διάσχισης Γράφων

Για τη διάσχιση γράφων χρησιμοποιούνται δύο μέθοδοι:

- **Αναζήτηση κατά πλάτος (breadth-first search)**
- **Αναζήτηση κατά βάθος (depth-first search)**

Η αναζήτηση κατά πλάτος υιοθετεί τη δομή της **ουράς** ώστε να πραγματοποιήσει την αναζήτηση ενώ η αναζήτηση κατά βάθος υιοθετεί τη δομή της **στοίβας**. Και οι δύο αλγόριθμοι βασίζονται σε μια μεταβλητή STATUS που υποδεικνύει την κατάσταση των κόμβων. Φυσικά, μπορεί να υπάρξουν κα άλλες υλοποιήσεις για την αναπαράσταση της κατάστασης των κόμβων. Ο επόμενος πίνακας παρουσιάζει τις τιμές STATUS και τη σημασία τους.

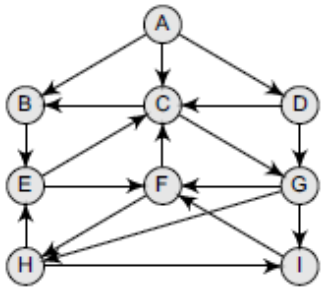
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Α. Αναζήτηση κατά πλάτος

Ο BFS (breadth-first search) ξεκινά από το ριζικό κόμβο και εξετάζει όλους τους γείτονές του. Αφού ολοκληρώσει την εξέταση των γειτόνων, για κάθε ένα από αυτούς εξετάζει όλους τους γειτονικούς τους αν δεν έχουν ήδη εξεταστεί. Ο αλγόριθμος έχει ως εξής:

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Ακολουθεί ένα παράδειγμα εκτέλεσης του αλγορίθμου. Στο παράδειγμα αυτό, πέρα από τους απαραίτητους δείκτες της ουράς που επεξεργαζόμαστε, κρατάμε στον πίνακα ORIG τον πατρικό κάθε κόμβου κατά την εκτέλεση του BFS. Η εκτέλεση τερματίζεται όταν φτάσουμε στον κόμβο I.



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

FRONT = 0	QUEUE = A
REAR = 0	ORIG = \0

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Execution: https://visualgo.net/dfsbfs

Code 3

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

```
#define MAX 100

#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);

int queue[MAX], front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

int main()
{
   create_graph();
   BF_Traversal();
   return 0;
}

void BF_Traversal()
{
   int v;

   for(v=0; v<n; v++)
      state[v] = initial;

   printf("Enter Start Vertex for BFS: \n");
   scanf("%d", &v);
   BFS(v);
}

void BFS(int v)
{
   int i;

   insert_queue(v);
   state[v] = waiting;

   while(!isEmpty_queue())
   {
      v = delete_queue( );
      printf("%d ",v);
      state[v] = visited;
```

```
      for(i=0; i<n; i++)
      {
         if(adj[v][i] == 1 && state[i] == initial)
         {
            insert_queue(i);
            state[i] = waiting;
         }
      }
   }
   printf("\n");
}

void insert_queue(int vertex)
{
   if(rear == MAX-1)
      printf("Queue Overflow\n");
   else
   {
      if(front == -1)
         front = 0;
      rear = rear+1;
      queue[rear] = vertex ;
   }
}

int isEmpty_queue()
{
   if(front == -1 || front > rear)
      return 1;
   else
      return 0;
}

int delete_queue()
{
   int delete_item;
   if(front == -1 || front > rear)
   {
      printf("Queue Underflow\n");
      exit(1);
   }

   delete_item = queue[front];
   front = front+1;
   return delete_item;
}

void create_graph()
{
   int count,max_edge,origin,destin;
```

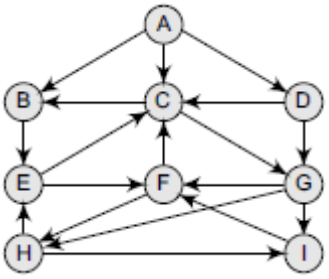
```
   printf("Enter number of vertices : ");
   scanf("%d",&n);
   max_edge = n*(n-1);

   for(count=1; count<=max_edge; count++)
   {
      printf("Enter edge %d( -1 -1 to quit ) : ",count);
      scanf("%d %d",&origin,&destin);

      if((origin == -1) && (destin == -1))
         break;

      if(origin>=n || destin>=n || origin<0 || destin<0)
      {
         printf("Invalid edge!\n");
         count--;
      }
      else
      {
         adj[origin][destin] = 1;
      }
   }
}
```

Β. Αναζήτηση κατά βάθος

Ο DFS (depth-first search) ξεκινά από τον αρχικό / ριζικό κόμβο και συνεχίζει προς τους κόμβους παιδιά μέχρι να βρεθεί ο κόμβος αναζήτησης ή να φτάσουμε σε κόμβο χωρίς παιδιά. Όταν ο αλγόριθμος φτάσει σε αδιέξοδο, τότε οπισθοχωρεί επιστρέφοντας στον πιο πρόσφατο κόμβο που δεν έχει πλήρως εξερευνηθεί. Ο αλγόριθμος έχει ως εξής:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Το ακόλουθο παράδειγμα ξεκινά από τον κόμβο H και προσπαθεί να τυπώσει όλους τους κόμβους που είναι προσβάσιμοι από αυτόν.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

	STACK: H
PRINT: H	STACK: E, I
PRINT: I	STACK: E, F
PRINT: F	STACK: E, C
PRINT: C	STACK: E, B, G
PRINT: G	STACK: E, B
PRINT: B	STACK: E
PRINT: E	STACK:

Execution: https://visualgo.net/dfsbfs

Code 4

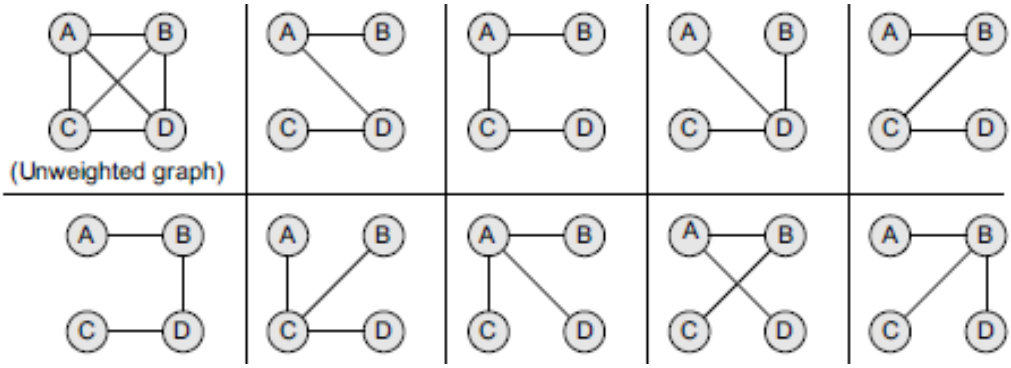
```
#include<stdio.h>

void DFS(int);
int G[20][20],visited[20],n;    //n is no of vertices and graph is sorted in array G[10][10]

int main()
{
   int i,j;
   printf("Enter number of vertices:");

   scanf("%d",&n);

   //read the adjecency matrix
   printf("\nEnter adjecency matrix of the graph:");

   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
         scanf("%d",&G[i][j]);

   //visited is initialized to zero
  for(i=0;i<n;i++)
```

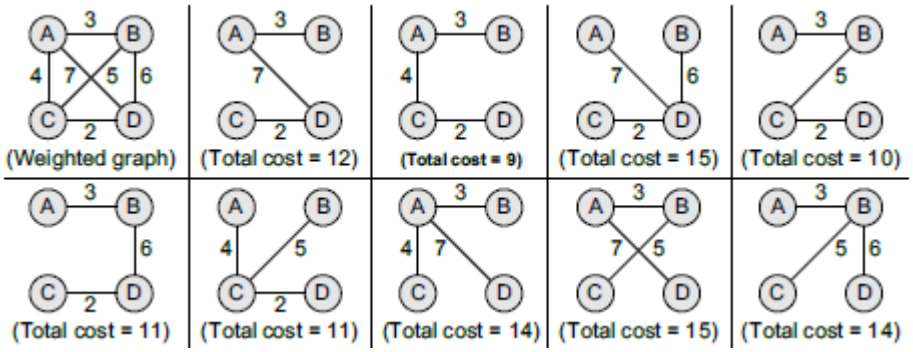
```
      visited[i]=0;

   DFS(0);
}

void DFS(int i)
{
   int j;
   printf("\n%d",i);
   visited[i]=1;

   for(j=0;j<n;j++)
     if(!visited[j]&&G[i][j]==1)
        DFS(j);
}
```

Δένδρα Επικάλυψης Ελάχιστου Κόστους (Minimum Spanning Trees)

Τα **δένδρα επικάλυψης (spanning trees)** είναι δένδρα τα οποία καλύπτουν όλους τους κόμβους ενός γράφου. Ένας γράφος μπορεί να έχει περισσότερα από ένα δένδρα επικάλυψης. Αν έχουν αποδοθεί βάρη στις ακμές του γράφου, τότε το **δένδρο επικάλυψης ελαχίστου κόστους (minimum spanning tree - MST)** είναι το δένδρο επικάλυψης που έχει το μικρότερο άθροισμα βαρών από όλα τα δένδρα επικάλυψης.

Παραδείγματα:





Ο Αλγόριθμος του Prim

Ο αλγόριθμος υιοθετεί την άπληστη μέθοδο και διατηρεί τρία σύνολα κόμβων:

- **Tree vertices**. Οι κόμβοι αποτελούν τμήμα του MST.
- **Fringe vertices**. Οι κόμβοι αυτοί δεν είναι τμήμα του MST αλλά είναι γείτονες κόμβων που ανήκουν στο MST.

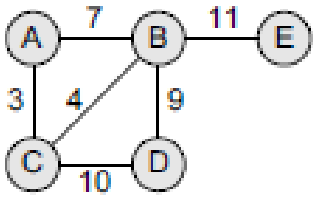
- **Unseen vertices**. Όλοι οι υπόλοιποι κόμβοι.

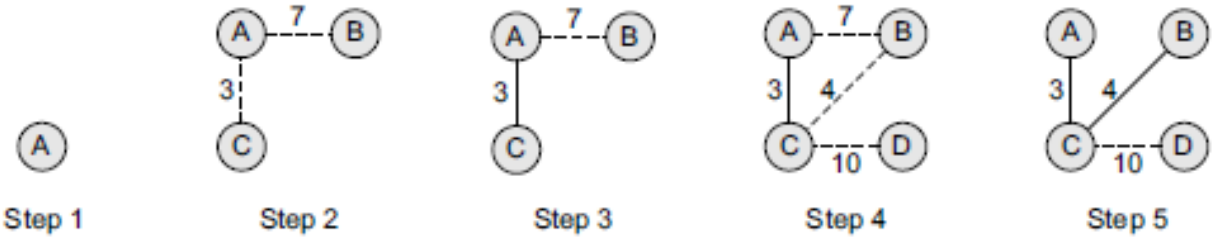
Ο αλγόριθμος έχει ως εξής:

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

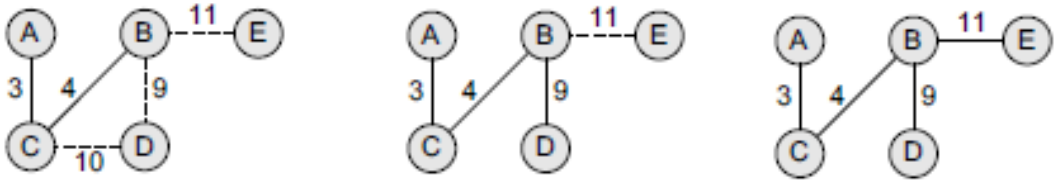
Παραδείγματα εκτέλεσης.

Α. Εκκίνηση από τον κόμβο Α



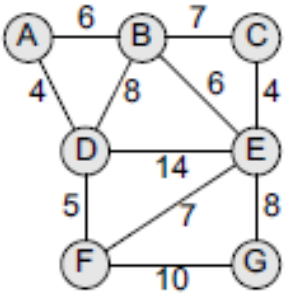


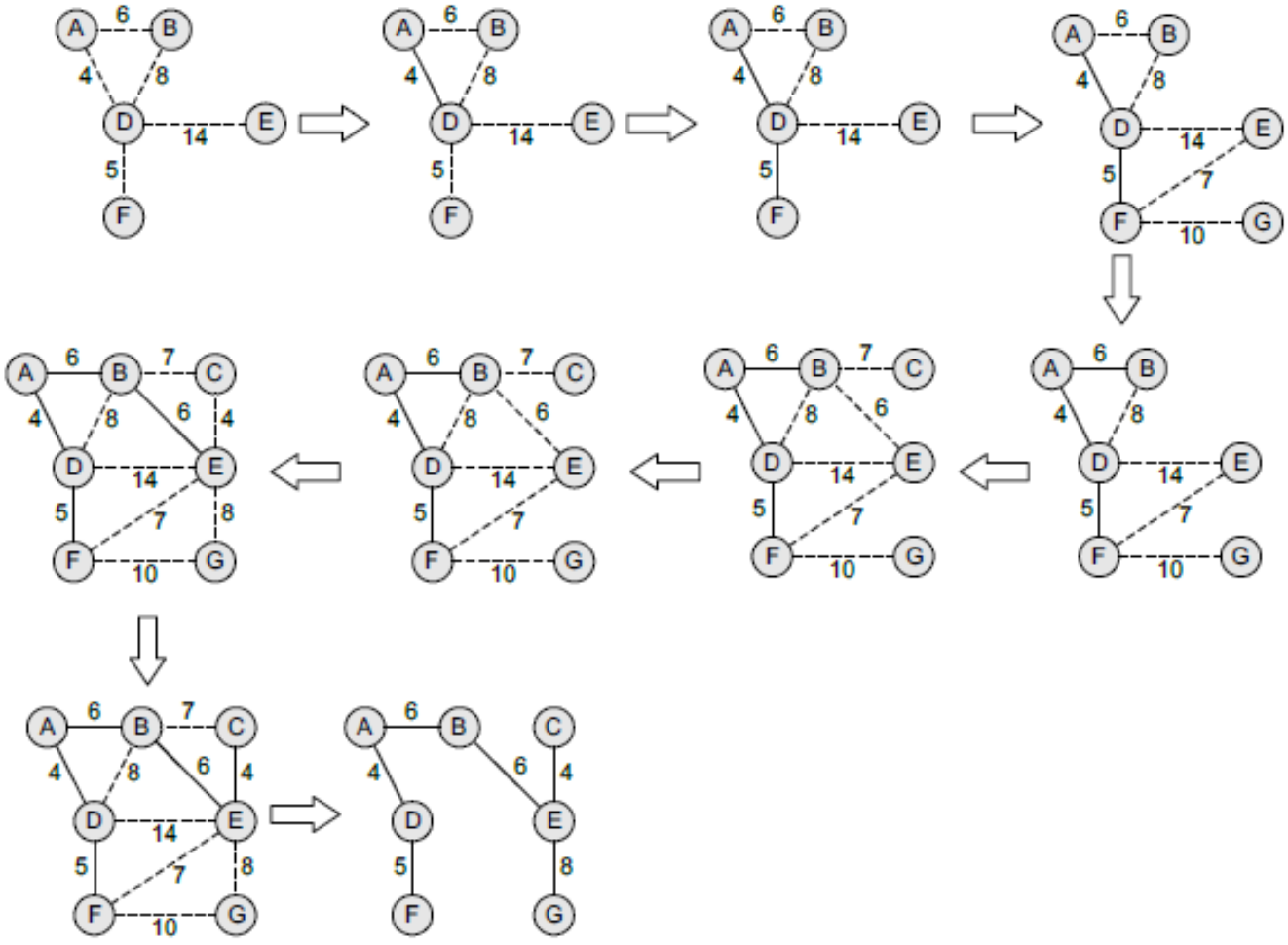
Step 1 Step 2 Step 3 Step 4 Step 5



Step 6 Step 7 Step 8

Β. Εκκίνηση από τον κόμβο D





Code 5

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (mstSet[v] == false && key[v] < min)
```

```
        min = key[v], min_index = v;

  return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
  printf("Edge   Weight\n");
  for (int i = 1; i < V; i++)
    printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V];   // Key values used to pick minimum weight edge in cut
    bool mstSet[V];  // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
      // Pick the minimum key vertex from the set of vertices
      // not yet included in MST
      int u = minKey(key, mstSet);

      // Add the picked vertex to the MST Set
      mstSet[u] = true;

      // Update key value and parent index of the adjacent vertices of
      // the picked vertex. Consider only those vertices which are not yet
      // included in MST
      for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
       if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])
          parent[v]  = u, key[v] = graph[u][v];
    }
```

```
    // print the constructed MST
    printMST(parent, V, graph);
}


// driver program to test above function
int main()
{
  /* Let us create the following graph
        2   3
     (0)--(1)--(2)
      |   / \   |
     6| 8/   \5 |7
      | /     \ |
     (3)-------(4)
           9         */
  int graph[V][V] = {{0, 2, 0, 6, 0},
                     {2, 0, 3, 8, 5},
                     {0, 3, 0, 0, 7},
                     {6, 8, 0, 0, 9},
                     {0, 5, 7, 9, 0},
                    };

   // Print the solution
   primMST(graph);

   return 0;
}
```

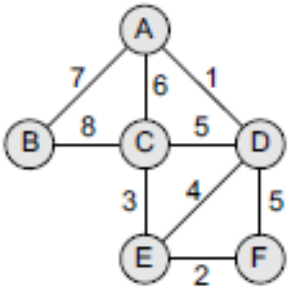
Execution: https://visualgo.net/mst

Ο Αλγόριθμος του Kruskal

Ο αλγόριθμος αναζητά το υποσύνολο ακμών που σχηματίζουν ένα δένδρο που περιλαμβάνει όλους τους κόμβους. Αν ο γράφος δεν είναι συνδεδεμένος, ο αλγόριθμος επιστρέφει ένα δάσος από MSTs. Ο αλγόριθμος υιοθετεί μια ουρά με προτεραιότητα στην οποία οι ακμές με το μικρότερο βάρος έχουν μεγαλύτερη προτεραιότητα έναντι των υπολοίπων. Ο αλγόριθμος έχει ως εξής:

```
Step 1: Create a forest in such a way that each graph is a separate
        tree.
Step 2: Create a priority queue Q that contains all the edges of the
        graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4:      Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees,
        then Add it to the forest (for combining two trees into one
        tree).
        ELSE
             Discard the edge
Step 6: END
```

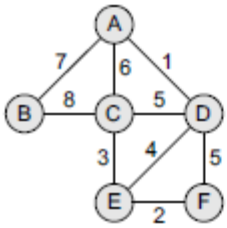
Παράδειγμα



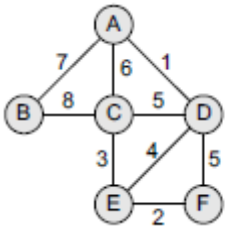
Αρχικά έχουμε:

$$F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\}$$
$$MST = \{\}$$
$$Q = \{(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)\}$$

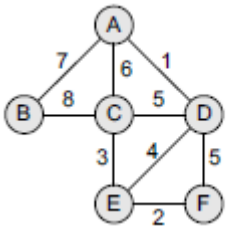
Οπότε



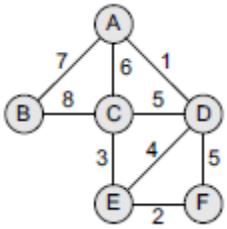
F = {{A, D}, {B}, {C}, {E}, {F}}
MST = {A, D}
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}



F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}



F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

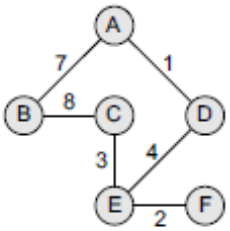


F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

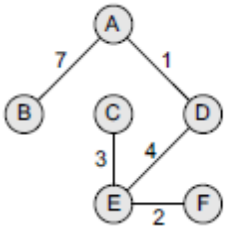
$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, C), (A, B), (B, C)\}$$

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, B), (B, C)\}$$



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}

Code 6

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
   int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
   // V-> Number of vertices, E-> Number of edges
   int V, E;

   // graph is represented as an array of edges. Since the graph is
   // undirected, the edge from src to dest is also edge from dest
```

```
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
```

```
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
```

```
        // If including this edge does't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                        result[i].weight);
    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
             10
        0--------1
        | \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);


    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
```

```
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Execution: https://visualgo.net/mst