

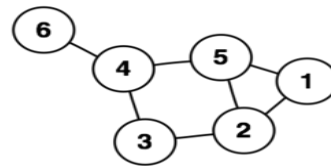
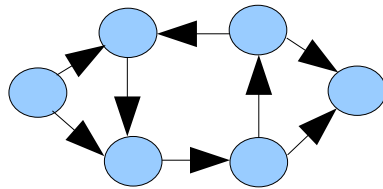


23/6/2010

## ***Grafos***

Tomás Arredondo Vidal

La teoría de los grafos estudia las propiedades de colecciones de objetos llamados **nod**os (o **vértices**) conectados por vínculos llamados **enlaces** (varios otros nombres son: arcos, aristas, elementos). Los enlaces de un grafo pueden o no tener orientación.



### **1. Definiciones**

#### ***Grafo***

Un grafo es un **par**  $G = (V, E)$ , donde

- $V$  es un conjunto de puntos, llamados nodos, y
- $E$  es un conjunto de pares de nodos, llamados enlaces.
- Un enlace  $\{n, m\}$  se puede denotar  $nm$ .



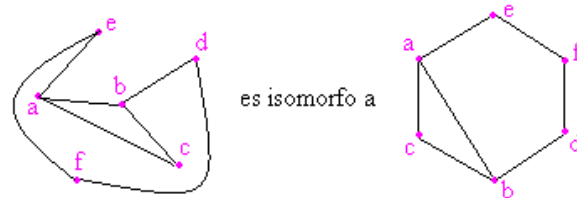
Grafos se pueden usar para modelar, estudiar y optimizar muchos tipos de redes y sistemas por ejemplo: redes de routers en internet, carreteras que conectan ciudades, redes y circuitos eléctrico, redes de alcantarillados, manejo de proyectos complejos, etc.

#### ***Nodos adyacentes***

- Dos nodos son adyacentes si existe solo un enlace entre ellos.

## Isomorfismos

En la teoría de los grafos, sólo queda lo esencial del dibujo de un grafo. **La forma de los nodos no son relevantes, sólo importan sus enlaces.** La posición de los nodos se pueden variar para obtener un grafo más claro, y hasta sus nombres se pueden cambiar. Estos cambios se llaman **isomorfismos** de grafos. Generalmente, se considera que colocar los vértices en forma de polígono regular da grafos muy legibles.



## Lazos (o bucles)

- Un **lazo o bucle** en un grafo es un enlace cuyos puntos finales son **el mismo nodo**.
- Un grafo se dice **simple** si no tiene lazos y existe como mucho un enlace entre cada par de nodos (no hay enlaces en **paralelo**).

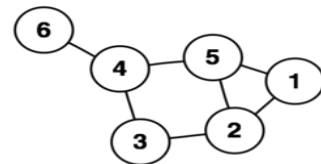
## Grado de incidencia (o valencia)

- El **grado de incidencia** de un nodo es el numero de enlaces que son incidentes en el.
- Si los enlaces tienen dirección entonces el **grado entrante** es el numero de enlaces que entran en el nodo.
- El **grado saliente** es el numero que sale de el.
- El **grado de un nodo** seria la suma de ambos. Un lazo cuenta por dos enlaces en el calculo de grado de incidencia.

## Ejemplo:

Un grafo simple con nodos  $V = \{1, 2, 3, 4, 5, 6\}$  y enlaces  $E = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\}, \{4,6\}\}$ .

- Los nodos 1 y 3 tienen una valencia de 2, los nodos 2,4 y 5 la tienen de 3 y el nodo 6 la tiene de 1.
- Los vértices 1 y 2 son adyacentes, pero no así los 2 y 4.
- El conjunto de vecinos para un vértice consiste de aquellos vértices adyacentes a él mismo.
- El vértice 1 tiene dos vecinos: el vértice 2 y el nodo 5.





## Completo

- Un grafo **completo** es un grafo simple en el que cada nodo es adyacente a cualquier otro nodo.
- El grafo completo en  $n$  nodos se denota a menudo por  $K_n$ . Tiene  $n(n-1)/2$  enlaces.

## Arbol

- Un árbol es un grafo conexo simple **acíclico**. Algunas veces, un nodo del árbol es distinguido llamándolo raíz.
- Enlaces de los arboles se denominan **ramas**.

## Densidad

- La **densidad** es el promedio de los grados de incidencia de los nodos. Si sumamos los grados de incidencia se tendrá un valor  $2E$  (se cuenta dos veces cada enlace).
- Entonces la densidad ( $D$ ) resulta:  $D = 2E/V$ .
- Si un grafo tiene una densidad proporcional a  $V$  entonces es **denso** (el numero de enlaces sera proporcional a  $V^2$  en caso contrario es un grafo **liviano** (sparse)).

## Subgrafos

- Un **subgrafo** ( $S$ ) de un grafo  $G$  es un grafo en el cual sus sets de vertices y enlaces ( $V_s, E_s$ ) son subconjuntos de los conjuntos de vertices y enlaces ( $V, E$ ) de  $G$ .

## Grafos dirigidos u orientados

En grafos dirigidos se impone un sentido a los enlaces, por ejemplo, si se quiere representar la red de las calles de una ciudad con sus inevitables direcciones únicas. Los enlaces son entonces pares ordenados de nodos, con  $(a,b) \neq (b,a)$ , como en el ejemplo:

En este grafo hay un enlace en el nodo (c) que tiene su inicio y termino en el mismo, es un lazo (o rizo, bucle).

También hay **un enlace sin flecha**: significa que el enlace se puede recorrer en cualquier sentido: es **bidireccional**, y corresponde a dos enlaces orientados.



$G = (V, E)$ ,  $V = \{ a, b, c, d, e \}$ , y  $E = \{ (a,c), (d,a), (a,e), (b,e), (c,a), (c,c), (d,b) \}$ .

Del vértice  $d$  sólo salen enlaces: es una **fuentes** (source). Al vértice  $e$  sólo entran enlaces: es un **pozo** (sink).

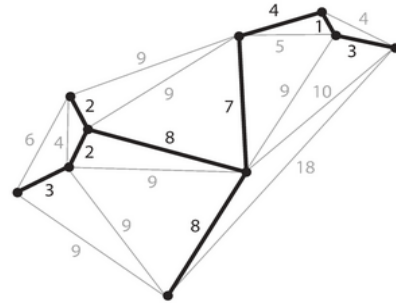
## ***Grafos ponderados, con pesos***

- Un **grafo ponderado** asocia un valor (o costo) a cada enlace en el grafo.
- El **peso de un camino** en un grafo ponderado es la suma de los pesos de todos los enlaces atravesados.

## ***Árbol de cobertura (spanning tree) y mínimo árbol de cobertura***

Dado un grafo conectado, sin dirección, un **árbol de cobertura** es un sub-grafo (que es un árbol) que **conecta todos los nodos**. Un grafo puede tener muchos posibles arboles de cobertura.

- Si el árbol tiene asignados pesos a cada enlace entonces se puede calcular un costo a cada posible árbol de cobertura al sumar el costo de travesar los enlaces.
- El **mínimo árbol de cobertura** (minimum spanning tree o MST) es un árbol de cobertura que tienen un peso menor que o igual al peso de todos los otros arboles de cobertura posibles.



Ejemplos del uso del MST abundan por ejemplo en Internet cuando se quiere hacer un broadcast (transmisión a múltiples destinos) las redes de routers calculan el MST y cuando quieren hacer un broadcast cada router reenvía paquetes a los routers en el MST.

Prim y Kruskal son dos algoritmos comúnmente usados para calcular el MST.

## ***Busca de ruta mínima (routing algorithms)***

Muchas veces se quiere encontrar la ruta mínima entre dos nodos en una red. Se considera una red a un grafo con enlaces orientados y con pesos. Es el caso del Internet en el cual se quiere determinar la ruta con el mínimo costo para enviar paquetes desde un origen a un destino. Ya que los terminales están conectados a un router origen el problema se reduce a determinar rutas de mínimo costo desde un router fuente a un router destino. El algoritmo **Dijkstra** comúnmente se utiliza para calcular la ruta de mínimo costo de un router a todos los otros routers de la red.

## 2.Representaciones

Se puede representar un grafo como una **matriz de adyacencia**, como una **lista de adyacencia** o como un **conjunto de enlaces**.

### **Matriz de adyacencia**

Se emplea una matriz cuadrada (V-1, V-1) donde V es el número de nodos:

- Se coloca un 1 en (v, w) si existe un enlace de v hacia w; 0 en caso contrario.
- La matriz es **simétrica si el grafo no es dirigido**.
- Si no se aceptan lazos, la diagonal está formada por ceros.

a	0	1	..	w	..	V-1
0						
1						
..						
v				1		
..						
V-1						

### **Lista de adyacencia**

Para cada nodo (de 0 a V-1) se listan los nodos adyacentes.

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

### **Matrices estáticas en C**

La notación: a[v][w] recuerda que la matriz a se visualiza como un arreglo de v renglones, donde cada renglón está formado por w columnas.

Ejemplo:

La matriz m se define según:

```
int m[2][4];
```


2 x 4

La matriz se inicializa mediante:

```
int m[2][4]={ {0,1,2,3},{4,5,6,7}};
```

## ***Matrices como argumentos***

- Si se pasa una matriz a una función, la declaración del argumento debe incluir la dimensión de la columna.
- La dimensión del renglón es irrelevante, ya que se pasa un puntero: `f(int m[ ][4])` ó `f(int (*m)[4])`

## ***Matrices dinámicas en C (Sedgewic)***

### **Declaración de un grafo**

```
struct graph {  
    int V; //Número de vértices  
    int E; //Número de enlaces  
    int **adj; // Matriz de adyacencias  
};  
typedef struct graph *Graph;
```

### **Declaración de un enlace**

Un enlace puede describirse por:

```
typedef struct  
{  
    int v; // nodo inicial. Desde.  
    int w; // nodo final. Hasta.  
} Edge;
```

### **Función para crear un enlace**

```
Edge EDGE(int v, int w)  
{  
    Edge t;  
    t.v=v;  
    t.w=w;  
    return (t);  
}
```

La siguiente definición crea un enlace y EDGE lo inicializa para ir del nodo 1 al 2:

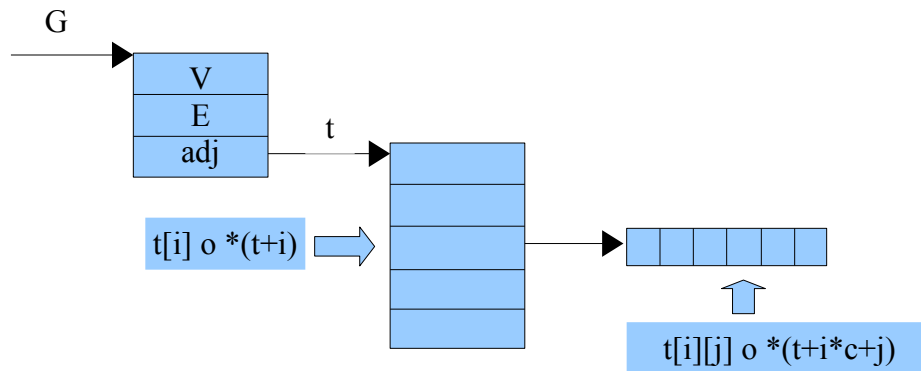
```
Edge enlace;  
enlace= EDGE(1,2);
```

### Las variables usadas para la creación de un grafo

Se define un grafo G, cuya matriz de adyacencias se define según un arreglo de r punteros (filas) para almacenar los arreglos de c enlaces (columnas):

```
Graph G = malloc(sizeof *G);      // Crea la cabecera del grafo.  
int **t = malloc(r * sizeof(int *)); // Crea arreglo de r renglones de punteros  
G->adj = t; //Pega el arreglo de punteros  
for (i = 0; i < r; i++)           // Crea y pega los renglones de c columnas:  
    t[i] = malloc(c * sizeof(int));
```

El siguiente diagrama ilustra las variables.





## ***Funciones para grafos descritos por su matriz de adyacencias***

**Funciones para la creación de un grafo vacío con V nodos:**

```
Graph GRAPHinit(int V)
{
    Graph G = malloc(sizeof *G);      // Crea cabecera del grafo
    G->V = V; G->E = 0;                // Con V nodos y 0 enlaces
    G->adj = MATRIXinit(V, V, 0);      // Lo inicia con ceros
    return G;
}
```

MATRIXinit crea la matriz de r renglones, c columnas y la inicializa con val.

```
int **MATRIXinit(int r, int c, int val)
{
    int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i = 0; i < r; i++)
        t[i] = malloc(c * sizeof(int));    // t[i] equivale a *(t+i)
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            t[i][j] = val;                 // equivale a ***(t+i*c+j) = val;
    return t;
}
```

De complejidad:  $O(r + r*c)$

### **Función para la liberación del espacio asociado al grafo**

Hay que tener cuidado de liberar en orden, de tal modo de no perder las referencias.

```
void BorreGrafo(Graph G)
{
    int i;
    int **t = G->adj;
    for (i = 0; i < G->V; i++)
        free(t[i]);    // primero borra los renglones
    free(t);           // luego el arreglo de punteros a los renglones
    free(G);           // finalmente la cabecera
}
```

### **Código para la creación del Grafo**

```
Graph Grafo;  
  
// Crear matriz de adyacencias del grafo  
#define VERTICES 5  
Grafo = GRAPHinit(VERTICES);
```

### **Función para la inserción de un enlace en un grafo**

La siguiente función inserta un enlace en un grafo G

```
void GRAPHinsertE(Graph G, Edge e)  
{  
    if (G->adj[e.v][e.w] == 0)  
        G->E++; // Aumenta el número de enlaces del grafo  
    G->adj[e.v][e.w] = 1;  
    G->adj[e.w][e.v] = 1; // Si el grafo no es dirigido.  
}
```

### **Función para la eliminación de un enlace**

La función siguiente remueve el enlace del grafo G

```
void GRAPHremoveE(Graph G, Edge e)  
{  
    if (G->adj[e.v][e.w] == 1)  
        G->E--; // Disminuye el contador de enlaces  
    G->adj[e.v][e.w] = 0;  
    G->adj[e.w][e.v] = 0;  
}
```

La acción siguiente, inserta el enlace en el Grafo

```
GRAPHinsertE(G, e);
```

### Creación de un conjunto de enlaces

El siguiente código crea el conjunto de enlaces de un grafo, como un arreglo llamado Enlaces.

Esta es otra forma de definir un grafo. Requiere 2E datos para ser definida, en lugar de V2 que necesita la matriz de adyacencia.

enlace	v	w
0	1	2
1	1	4
2	2	3
3	3	4
4	4	0
5	3	0

```
#define ENLACES 6
Edge Enlaces[ENLACES]={ {1,2}, {1,4}, {2,3}, {3,4}, {4,0}, {3,0} };
```

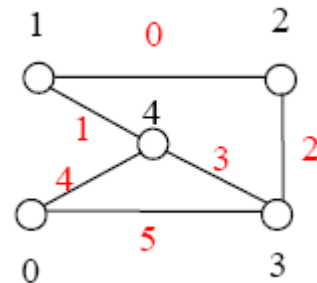
### Creación de matriz de adyacencia desde un arreglo de enlaces

Entonces la creación de la matriz de adyacencia de un grafo de cinco vértices, a partir del arreglo de enlaces, definido por seis enlaces, puede realizarse, según:

```
for(i=0; i<ENLACES; i++)
    GRAPHinsertE(Grafo, Enlaces[i] );
```

Esto tiene complejidad  $O(E)$ .

El grafo resultante, puede visualizarse, según:



Mostrando que la matriz de incidencia puede obtenerse del arreglo de los enlaces, lo cual indica que son representaciones equivalentes.

Arreglo de enlaces

enlace	v	w
0	1	2
1	1	4
2	2	3
3	3	4
4	4	0
5	3	0



Matriz de adyacencia

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

### Generación de arreglo de enlaces a partir de la matriz de adyacencia

Si el grafo ya está construido, la generación de los enlaces, a partir del grafo, se logra con la función:

```
int GRAPHedges(Edge a[], Graph G)
{
    int v, w, E = 0;           // numera los enlaces desde el cero.
    for (v = 0; v < G->V; v++) // para todos los renglones
        for (w = v+1; w < G->V; w++) //revisa por columnas
            if (G->adj[v][w] == 1)
                a[E++] = EDGE(v, w); // escribe por referencia
    return E;                   // retorna el número de enlaces
}
```

Se advierte que debido a los dos for anidados es  $O((V^2 - V)/2)$ , ya que revisa sobre la diagonal. Este es el código para ejecutar la función:

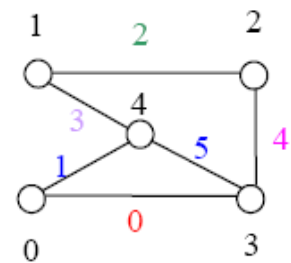
```
GRAPHedges(Enlaces, Grafo); //Llena el arreglo a partir del Grafo.
```

Nótese que al recorrer la submatriz sobre la diagonal, por renglones va reasignando, a partir de cero, los nombres de los enlaces (o elementos), y sus correspondientes nodos.

Debido a que la matriz de adyacencias no almacena información sobre el nombre de los enlaces, el arreglo de enlaces toma los nombres dados por el recorrido.

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

elemento	v	w
0	0	3
1	0	4
2	1	2
3	1	4
4	2	3
5	3	4



### 3.Existencia de trayectorias entre dos nodos

Un problema en relación con los grafos es determinar si existe una **trayectoria entre dos nodos** v y w. Si se define un arreglo en que se marque si los nodos han sido o no visitados, puede plantearse el siguiente esquema recursivo:

Si el vértice inicial y el final son iguales, hay trayectoria (fin de recursión).

Marcar el vértice inicial como visitado.

Revisar todos los vértices, conectados al inicial:

Si uno no ha sido revisado: ver si hay trayectoria entre ese y el final.

Si revisados todos no hay trayectoria, entonces no existe la trayectoria buscada.

```
int pathR(Graph G, int v, int w)
{
    int t;
    if (v == w)
        return 1;      //Existe trayecto. Nodo inicial y final son iguales.
    visited[v] = 1;
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[v][t] == 1)      //Si v está conectado con t
        {
            if (visited[t] == 0)    //y t no ha sido visitado
            {
                printf("%d-%d ", v, t); // ver enlace de trayectoria
                if (pathR(G, t, w))
                    return 1;
            }
        }
    }
    return 0;
}
```

```

int GRAPHpath(Graph G, int v, int w)
{
    int t;
    for (t = 0; t < G->V; t++)
        visited[t] = 0;
    return pathR(G, v, w); //Inicia búsqueda recursiva.
}

```

Ejemplo de invocación:

```

if( GRAPHpath(Grafo, 1, 3))
    printf("existe trayectoria, entre 1 y 3\n");
else
    printf("no existe trayectoria.\n");

```

## 4. Exploración de Grafos

Para **determinar propiedades de los grafos** (e.g. determinar los grados de incidencia o densidades) es necesario **recorrerlos**. Este proceso es equivalente a la exploración de un laberinto. Hay dos métodos básicos el de búsqueda en profundidad (depth first search) y el de búsqueda en extensión (breath first search).

### ***Búsqueda en profundidad***

- Se recorre el grafo, siempre hacia adelante hasta que se llega al final o hasta una trayectoria que ya se ha recorrido; luego se devuelve y busca trayectorias no exploradas.
- El objetivo es generar un **árbol de las trayectorias**.
- Se **visita y marca un vértice (nodo) como visitado**, luego se visitan (recursivamente) todos los vértices adyacentes al recién marcado que no estén visitados.
- Esto va formando un árbol, con el orden en que se van visitando los vértices.



Si el vértice inicial es el 0, se lo visita y marca con la cuenta 0, y se generan llamados con los enlaces (0,3) y (0,4), en ese orden.

El llamado con el enlace (0, 3) marca el 3 con la cuenta 1, y se generan los llamados: con (3, 2), (3,4).

El llamado con el enlace (3, 2) marca el 2 con la cuenta 2, y se genera el llamado: (2,1).

El llamado con el enlace (2, 1) marca el 1 con la cuenta 3, y se genera el llamado: (1,4).

El llamado con el enlace (1, 4) marca el 4 con la cuenta 4 y no genera nuevos llamados.

Resultando el árbol T(1, 2, 3, 4) ilustrado en la figura a la derecha.

### ***Función para búsqueda en profundidad (depth first search)***

Este es el código (version recursiva) del algoritmo presentado anteriormente:

```
void dfsR(Graph G, Edge e)
{
    int t, w = e.w;
    pre[w] = cnt++;      // Marca con contador creciente
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[w][t] != 0) //Si hay conexión entre w y t
            if (pre[t] == -1)
                dfsR(G, EDGE(w, t)); // Si t no está visitado sigue
    }
}
```

La función que efectúa la búsqueda en profundidad:

```
void GRAPHsearchDFS(Graph G) //depth-first-search
{
    int v;
    cnt = 0;
    for (v = 0; v < G->V; v++)
        pre[v] = -1;      //Marca todos como no visitados
    for (v = 0; v < G->V; v++) //Revisa todos los vértices.
        if (pre[v] == -1)
        {
            //Llama varias veces si G no es conectado
            dfsR(G, EDGE(v, v));
        }
}
```

Se visitan todos los enlaces y todos los vértices conectados al vértice de partida, no importando el orden en que revisa los enlaces incidentes en ese vértice. La estrategia recursiva implica un orden: el último que entró es el primero que salió (LIFO), de los enlaces posibles se elige el más recientemente encontrado.



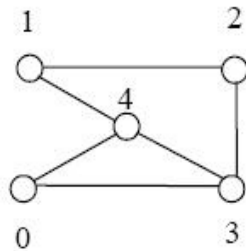
### *Modificación para entender la operación de la función*

Se agrega la variable estática `indente`, para ir mostrando los enlaces que son sometidos a revisión. Cada vez que se **genera un llamado se produce un mayor nivel de indentación**; el cual es repuesto al salir del llamado recursivo.

Se agrega un **asterisco para mostrar los enlaces que generan llamados recursivos**.

```
static int indente=0;
void dfsR(Graph G, Edge e)
{
    int t,j, w = e.w;
    pre[w] = cnt++;
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[w][t] != 0)
        {
            for (j = 0; j < indente; j++)
                printf(" ");
            printf("%d-%d \n", w, t);
            if (pre[t] == -1)
            {
                indente++;
                putchar('*');dfsR(G, EDGE(w, t));
                indente--;
            }
            else putchar(' ');
        }
    }
}
```

Para el siguiente grafo, se produce el listado:



```

0-3
* 3-0
  3-2
*   2-1
*   1-2
    1-4
*   4-0
    4-1
    4-3
      2-3
      3-4
0-4

```

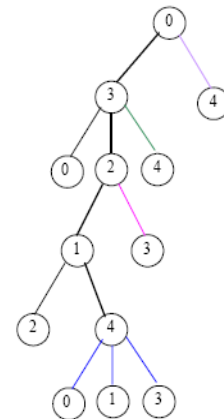
### Arreglo de padres

La siguiente modificación permite almacenar en el arreglo **st** el **padre del vértice w**. Esa información es útil para aplicaciones de este algoritmo. Y es una forma de describir el árbol.

```

static int st[VERTICES];
void dfsR(Graph G, Edge e)
{
    int t,j, w = e.w;
    pre[w] = cnt++;      //Marca con contador creciente
    st[e.w] = e.v;      //Se guarda el padre de w.
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[w][t] != 0) //Si hay conexión entre w y t
            if (pre[t] == -1) // Y t no esta visitado
                dfsR(G, EDGE(w, t)); // Buscar nodo t
    }
}

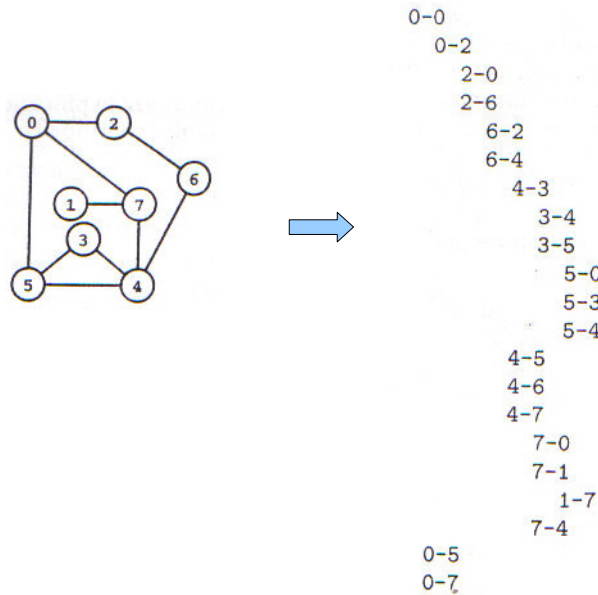
```



Para el ejemplo, quedarían almacenados en **st**: **0, 2, 3, 0, 1**.

- Y en **pre**: **0, 3, 2, 1, 4**. El árbol que genera este algoritmo, se produce revisando los enlaces en el siguiente orden: **0-3** , **\*3-0** , **3-2** , **\*2-1** , **\*1-2** , **1-4** , **\*4-0** , **4-1** , **4-3** , **2-3** , **3-4** , **0-4**.
- Los que generan llamados recursivos, se preceden con un asterisco. La ilustración muestra que se avanza primero en profundidad.

Otro ejemplo:

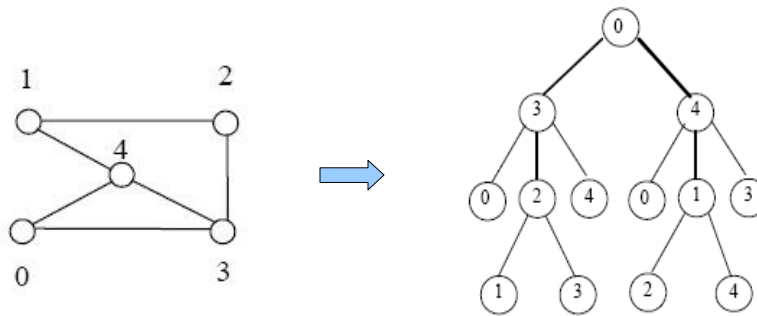


### ***Búsqueda en extensión.***

La búsqueda en extensión tiene por objetivo encontrar rutas más cortas entre dos vértices dados. También se genera un árbol.

- Se parte de un nodo, y se busca el siguiente nodo en todas las rutas de largo uno que existan; luego se buscan todas las rutas de largo dos, y así sucesivamente.
- Explorar los vértices, de acuerdo a su distancia al de partida, implica que de los enlaces que son posibles, se escoge uno y los otros se salvan para ser posteriormente explorados.
- Este orden es: el primero que se encuentra, es el primero en ser procesado (FIFO). Para visitar un vértice: se buscan los enlaces que son incidentes con ese vértice, y los enlaces que tienen el otro vértice no visitado, se encolan.

Ejemplo:



A partir del nodo inicial 0, se marca el 0, y se exploran los enlaces: 0-3 y 0-4.  
Se encolan el 0-3 y el 0-4. Ya que el 3 y 4 no han sido visitados aún.  
Se desencola el 0-3, se marca el 3, y se revisan el 3-0, el 3-2 y el 3-4.  
Se encolan el 3-2 y el 3-4. Ya que el 2 y el 4 no han sido visitados aún.  
Se desencola el 0-4, se marca el 4, y se revisan el 4-0, 4-1 y 4-3. Se encola el 4-1.  
Se desencola el 3-2, se marca el 2, y se revisan el 2-1, 2-3. Se encola el 2-1.  
Se desencola el 3-4 sin procesar.  
Se desencola el 4-1, se marca el 1, y se revisan el 1-2 y 1-4.  
Se desencola el 2-1 sin procesar.  
Del árbol formado, se visualiza, que el vértice 0 está a distancia uno de los vértices 3 y 4, y que está a distancia 2, de los vértices 1 y 2.  
Los vértices se marcan en el siguiente orden: 0, 3, 4, 2, 1.

### ***Código para búsqueda en extensión en un grafo***

Esta es la función que encola los enlaces.

```
void bfs(Graph G, Edge e) //breadth-first-search
{
    int t;
    QUEUEput(e);
    while (!QUEUEempty())
    {
        if (pre[(e = QUEUEget()).w] == -1)
        {
            pre[e.w] = cnt++; // en pre queda el orden de escoger los vértices
            st[e.w] = e.v;    //en st queda el padre.
            for (t = 0; t < G->V; t++)
                if (G->adj[e.w][t] == 1)
                    if (pre[t] == -1)
                        QUEUEput(EDGE(e.w, t));
        }
    }
}
```

La función que genera el árbol BFS.

```
void GRAPHsearchBFS(Graph G)
{
    int v;
    cnt = 0;
    QUEUEinit(ENLACES);
    for (v = 0; v < G->V; v++) {
        pre[v] = -1; st[v] = -1;
    }
    for (v = 0; v < G->V; v++) // Para todos los vértices
        if (pre[v] == -1)
            bfs(G, EDGE(v, v)); //Se invoca una vez, si el grafo es conectado.
    QUEUEDestroy();
}
```

## 5. Grafos Ponderados

Hay varios problemas que para ser resueltos requieren que **el grafo tenga pesos asociados a sus enlaces**. Para incluir pesos va a ser necesario modificar las estructuras de datos para incorporarlos a los elementos del grafo.

Una manera de implementar los pesos es que cada peso tenga un valor real entre 0 y menor que 1. Esto puede lograrse dividiendo los pesos reales por el peso del mayor levemente incrementado.

### ***Modificación de las funciones para tratar grafos con pesos***

Se modifica la estructura de un enlace:

```
typedef struct
{
    int v;          // vértice inicial
    int w;          // vértice final
    float wt;       // peso. Puede ser un double
} Edge;
```

El constructor queda ahora:

```
Edge EDGE(int v, int w, float wt)
{
    Edge t;
    t.v=v;
    t.w=w;
    t.wt=wt;
    return (t);
}
```

Para un grafo se definen:

```
struct graph {
    int V; //Número de vértices
    int E; //Número de enlaces
    float **adj; // Matriz de adyacencias
};

typedef struct graph *Graph;
```

Para marcar la **no** existencia de adyacencias, se define:

```
#define maxWT 1. //Todo enlace tiene menor peso que maxWT.
```

La inicialización de un grafo vacío, se logra con:

```
Graph GRAPHinit(int V)
{
    Graph G = malloc(sizeof *G); //crea cabecera del grafo
    G->V = V;
    G->E = 0;
    G->adj = MATRIXfloat(V, V, maxWT);
    return G;
}
```

Donde la función que localiza espacio dinámico para la matriz de adyacencias es:

```
float **MATRIXfloat(int r, int c, float wt)
{
    int i, j;
    float **t = malloc(r * sizeof(float *));
    for (i = 0; i < r; i++)
        t[i] = malloc(c * sizeof(float));
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            t[i][j] = wt; //equivalente a *(t+i+j) = wt;
    return t;
}
```

El resto de las funciones se modifican para tratar grafos ponderados:

```
void BorreGrafo(Graph G) //Libera el espacio adquirido por malloc.
{
    int i;
    float **t = G->adj;
    for (i = 0; i < G->V; i++)
        free(t[i]);
    free(t);
    free(G);
}

void GRAPHinsertE(Graph G, Edge e) //Inserta enlace
{
    if (G->adj[e.v][e.w] == maxWT)
        G->E++;
    G->adj[e.v][e.w] = e.wt;
    G->adj[e.w][e.v] = e.wt; //suprimir para grafos dirigidos
}

void GRAPHremoveE(Graph G, Edge e) //Remueve enlace
{
    if (G->adj[e.v][e.w] != maxWT)
        G->E--;
    G->adj[e.v][e.w] = maxWT;
    G->adj[e.w][e.v] = maxWT; //suprimir para grafos dirigidos
}

int GRAPHedges(Edge a[], Graph G) //Forma arreglo de enlaces del grafo
{
    int v, w, E = 0;
    for (v = 0; v < G->V; v++)
        for (w = v+1; w < G->V; w++)
            if (G->adj[v][w] != maxWT)
                a[E++] = EDGE(v, w, G->adj[v][w]);
    return E;
}
```



Muestra la matriz mediante listas de vértices conectados a cada vértice

```
void GRAPHshowL(Graph G)
{
    int i, j;
    printf("%d vertices, %d edges\n", G->V, G->E);
    for (i = 0; i < G->V; i++) {
        printf("%d: ", i);
        for (j = 0; j < G->V; j++)
            if (G->adj[i][j] != maxWT)
                printf(" %0.2f", G->adj[i][j]);
        putchar('\n');
    }
}
```

Muestra Matriz de adyacencias.

```
void GRAPHshowM(Graph G)
{
    int i, j;
    printf("%d vertices, %d edges\n", G->V, G->E);
    printf(" ");
    for (j = 0; j < G->V; j++)
        printf(" %4d ", j);
    printf("\n");
    for (i = 0; i < G->V; i++)
    {
        printf("%2d:", i);
        for (j = 0; j < G->V; j++)
            if (G->adj[i][j] != maxWT)
                printf(" %0.2f", G->adj[i][j]);
            else
                printf(" * ");
        putchar('\n');
    }
}
```

El siguiente código describe un grafo ponderado a partir de sus enlaces. Se limita a dos el número de cifras significativas:

```
#define VERTICES 8
#define ENLACES 12
//Variables
Graph Grafo;
//Edge enlaces[ENLACES] = {      {0,2,.29},{4,3,.34},{5,3,.18},{7,4,.46},\
                                   {7,0,.31},{7,6,.25},{7,1,.21},{0,6,.51},\
                                   {6,4,.52},{4,5,.40},{5,0,.59},{0,1,.32} };
```

La lista de vértices adyacentes incluye 8 vértices, 12 enlaces y sus pesos:

0: 0.32 0.29 0.59 0.51 0.31	0: 1 2 5 6 7
1: 0.32 0.21	1: 0 7
2: 0.29	2: 0
3: 0.34 0.18	3: 4 5
4: 0.34 0.40 0.52 0.46	4: 3 5 6 7
5: 0.59 0.18 0.40	5: 0 3 4
6: 0.51 0.52 0.25	6: 0 4 7
7: 0.31 0.21 0.46 0.25	7: 0 1 4 6

La matriz de adyacencias con sus pesos es:

	0	1	2	3	4	5	6	7
0:	*	0.32	0.29	*	*	0.59	0.51	0.31
1:	0.32	*	*	*	*	*	*	0.21
2:	0.29	*	*	*	*	*	*	*
3:	*	*	*	*	0.34	0.18	*	*
4:	*	*	*	0.34	*	0.40	0.52	0.46
5:	0.59	*	*	0.18	0.40	*	*	*
6:	0.51	*	*	*	0.52	*	*	0.25
7:	0.31	0.21	*	*	0.46	*	0.25	*

### ***Búsqueda en profundidad en un árbol ponderado***

El siguiente código hace una búsqueda en profundidad en un árbol ponderado. **Se agrega el vector wt con los pesos.**

```
static int cnt;
static int pre[VERTICES];
static int st[VERTICES];
static float wt[VERTICES];
void dfsR(Graph G, Edge e)
{
    int t;
    pre[e.w] = cnt++;
    st[e.w] = e.v;           // Se guarda el vértice v padre de w.
    wt[e.w]=G->adj[e.w][e.v]; // Se guarda el peso del enlace w-v
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[e.w][t] != maxWT)
            if (pre[t] == -1)
            {
                dfsR(G, EDGE(e.w, t, maxWT));
                /*printf("%d-%d \n", e.w, t);*/
            }
    }
}

void GRAPHsearchDFS(Graph G)
{
    int v;  cnt = 0;
    for (v = 0; v < G->V; v++)
    {
        pre[v] = -1;
        st[v] = -1;} //Inicialización
    for (v = 0; v < G->V; v++)
        if (pre[v] == -1)
            dfsR(G, EDGE(v, v, maxWT));
}
```

El código entrega el siguiente árbol de cobertura para el grafo indicado anteriormente:

0	1	2	3	4	5	6	7	Nodo: i
0	0	0	4	7	3	4	1	Padre de nodo: st[i]
1.00	0.32	0.29	0.34	0.46	0.18	0.52	0.21	Peso entre nodo y padre: wt[i]

La raíz del árbol (vértice 0) tiene un lazo de peso infinito (valor 1.0) consigo misma. Este árbol cubre todos los nodos, pero **no es** un árbol de cobertura mínima (MST).

### ***Mínimo árbol de cobertura (MST)***

- Dado un grafo conectado, sin orientación, un árbol **de cobertura** es un subgrafo que es un árbol y que conecta todos los nodos del grafo.
- Un grafo puede tener muchos arboles de cobertura, si se le asigna un peso a cada enlace y se suman estos pesos se puede calcular el peso total del árbol de cobertura.
- El **mínimo árbol de cobertura (MST)** es un árbol con peso menor o igual al peso de todos los otros posibles arboles de cobertura. Para un grafo dado existe un muy elevado número de árboles (un bosque).
- Los algoritmos de **Prim** y **Kruskal** son comúnmente utilizados para encontrar un MST.

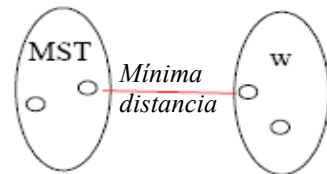
### **Algoritmo de Prim**

El algoritmo de Prim, elige un vértice cualquiera como inicial.

Luego repite para los (V-1) vértices restantes:

Agregar enlace de peso mínimo que conecte los vértices del MST, con los vértices que aún no pertenecen al MST.

Si se tienen vértices en el MST, se busca un vértice w que aún no está en el árbol, tal que esté a menor distancia de los vértices del MST. Para esto es preciso registrar las menores distancias de cada nodo (no del MST) al MST, y elegir la **menor**.



Para diseñar el algoritmo se emplean tres arreglos:

El árbol se describe por el **arreglo st[v]**, donde se almacena el padre del vértice v. Se inicia con valores iguales a menos uno, para indicar que ningún vértice forma parte del árbol.

Se agrega un arreglo **fr[w]**, en el cual, durante el procesamiento, se dejará el vértice **más cercano a w** del árbol. Se inicia con **fr[i] = i**, dado que no se conoce inicialmente.

Se tiene un arreglo **wt[w]** para almacenar los pesos de los enlaces. Se usa para almacenar la mínima distancia al árbol, si el vértice **w** aún no pertenece al árbol, y la distancia al padre si el vértice **w** ya pertenece al MST. Al inicio se lo llena con **maxWT**, para indicar que esta información sobre pesos mínimos aún no se conoce.

En la **variable local min** se almacena el vértice, que aún no pertenece al MST, y el cual debe cumplir la propiedad de tener distancia mínima con los vértices que ya están en el MST. Para asociarle un peso, se agrega una entrada al arreglo **wt**, con valor **maxWT**, y se lo inicia con valor **V** (vértice que no existe en el grafo). De esta forma **wt[min]** tendrá un espacio y valor definido.

```
#define P G->adj[v][w]

void GRAPHmstPrim(Graph G) //minimum spanning tree
{
    int v, w, min;
    for (v = 0; v < G->V; v++) { //Inicia arreglos
        st[v] = -1; fr[v] = v; wt[v] = maxWT;
    }
    st[0] = 0; wt[0] = 0; //Elige primero al vértice 0 como parte del MST
    wt[G->V] = maxWT; //Coloca centinela. Requiere una posición adicional
    for (min = 0; min != G->V; ) //Busca un vértice que no pertenece al MST
    {
        v = min;
        st[min] = fr[min]; // agrega vértice v al MST
        for (w = 0, min = G->V; w < G->V; w++)
            //Busca la minima distancia

            //Al inicio min es un vértice que no existe.
            if (st[w] == -1) { //si w no está aún en el MST
                if (P < wt[w]) { //salva distancia menor y el vértice.
                    wt[w] = P; fr[w] = v;
                }
                if (wt[w] < wt[min]) min = w; //Cambia el mínimo
            }
    }
}
```

Para el grafo anterior, luego de ejecutado el Algoritmo de Prim, se genera:

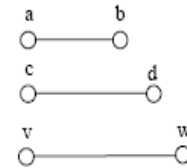
0	1	2	3	4	5	6	7	Nodo: i
0	7	0	4	7	3	7	0	Padre del nodo: st[i]
1.00	0.21	0.29	0.34	0.46	0.18	0.25	0.31	Peso enlace entre nodo y su Padre: wt[i]
0	7	0	4	7	3	7	0	Nodo que dista menos del nodo del árbol

La suma = 0.21+0.29+0.34+0.46+0.18+0.25+0.31 es la mínima

### Algoritmo de Kruskal.

Se usa para construir un **minimum spanning tree (MST)**. Se agrega un enlace por iteración que conecta dos subárboles MST de forma mínima. Empieza con un bosque degenerado con árboles de un nodo cada uno.

- 1- Ordenar enlaces según su peso.
- 2- Incluir en MST el enlace con menor peso:  $MST = \{a, b\}$ .
- 3- Se agregan los enlaces de acuerdo a su peso que no estén presentes en el MST y que no causen lazos.
- 4- Se termina cuando el MST tenga  $V-1$  nodos



En el arreglo mst de enlaces, se deja el árbol, se emplea como variable global.

```
Edge mst[VERTICES-1];
```

```
void GRAPHmstKruskal(Graph G, Edge *e)
{
    int i, k, E;
    E=GRAPHEdges(e, G); //Se crear arreglo de enlaces a partir del grafo.
    qsortnw(e, 0, E-1); //se ordenan los enlaces, según el peso.
    UFininit(G->V); //crea conjunto de vértices.
    for (i= 0, k = 0; i < E && k < G->V-1; i++) // agregar máximo V-1 ramas
        if (!UFfind(e[i].v, e[i].w)) //si v no está conectado a w
        {
            UFunion(e[i].v, e[i].w); //se agregan vértices al conjunto
            mst[k++] = e[i]; //se agrega rama e al árbol mst
        }
    Ufdestroy(); }
```

La rutina de ordenamiento quicksort, se modifica, para adecuarla a los tipos de datos del grafo, y para ordenar según el peso:

```
void qsortnw(Edge *a, int l, int r)
{
    int i=l, j=r;
    Edge temp;
    float piv=a[(l+r)/2].wt;
    do {
        while ( a[i].wt < piv) i++;
        while( piv < a[j].wt) j--;
        if( i<=j){
            if (i!=j) {
                temp=a[i], a[i]= a[j], a[j]=temp;
            }
            i++; j--;
        }
    } while(i<=j);
    if( l < j)
        qsortnw( a, l, j);
    if( i < r )
        qsortnw( a, i, r);
}
```

El siguiente código prueba la función, e imprime el mínimo árbol de cobertura:

```
GRAPHmstKruskal(Grafo, Enlaces);
for (i = 0; i < Grafo->V-1; i++)
    printf("%2d-%2d =%0.2f\n", mst[i].v, mst[i]. w,mst[i].wt);
```

## Funciones usadas por el algoritmo Kruskal para manejo de conjuntos

Las funciones que manejan conjuntos, se encuentra en las primeras páginas del texto de R. Sedgewick. En el arreglo `id`, se identifican los vértices de un grafo. Se lleva la cuenta del número de nodos de cada subconjunto en `sz`.

```
static int *id, *sz;
void UFininit(int N)
{
    int i;
    id = (int *) malloc(N*sizeof(int));
    sz = (int *) malloc(N*sizeof(int));
    for (i = 0; i < N; i++)
        { id[i] = i; sz[i] = 1; }
}
```

id	sz
0	1
1	1
N-1	1

Cada vértice apunta a otro en el mismo subconjunto, sin ciclos. Los vértices conectados, de un subconjunto apuntan a la raíz.

```
int find(int x)
{
    int i = x;
    while (i != id[i])
        i = id[i];
    return i;
}
```

Esta función retorna uno si están conectados

```
int UFfind(int p, int q)
{ return (find(p) == find(q)); }
```



```

int UFunion(int p, int q)
{
    int i = find(p), j = find(q);
    if (i == j) return 1;
    if (sz[i] < sz[j]) //si el subconjunto i es menor que el subconjunto j
    { id[i] = j; sz[j] += sz[i]; } //el i se pega al j y se mantienen los contadores
    else
    { id[j] = i; sz[i] += sz[j]; } //sino el j se pega al i y se mantienen los contadores
    return 0;
}

```

```

void UFdestroy(void)
{
    free(id);
    free(sz);
}

```

### **Otro diseño de las rutinas:**

//Retorna uno si están conectados

```

int UFfind1(int p, int q)
{
    return (id[p] == id[q]); } // O(1)

```

//Cada vértice apunta a otro en el mismo subconjunto, sin ciclos.

//Los vértices conectados, de un subconjunto apuntan a la raíz

```

void UFunion1(int p, int q, int N)
{
    int t,i;
    for(t=id[p], i=0; i<N; i++) // O(N)
    if(id[i] == t) id[i] = id[q]; //le coloca raíz q a todos los conectados al conjunto p
}

```

## 6. Grafos orientados ponderados

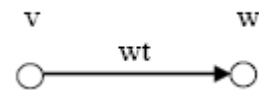
- Grafos orientados con pesos se denominan redes.
- Existen tres tipos de problemas en redes.
- Uno de ellos es dados dos vértices en una red encontrar la trayectoria orientada más corta entre los vértices, se lo denomina fuente-sumidero.
- Otro es encontrar todos los trayectos más cortos entre un vértice y todos los demás; esto equivale a encontrar un árbol (SPT shortest-path-tree) que conecte al vértice con todos los demás vértices que son alcanzables, de tal modo que la trayectoria en el árbol sea la más corta dentro de la red.
- Un tercer problema es encontrar todos los pares de rutas más cortas.

### ***Modificación de las funciones para tratar grafos orientados con pesos***

La definición del enlace debe entenderse como dirigido de v hacia w.

Edge EDGE(int v, int w, float wt)

```
{  
    Edge t;  
    t.v=v; t.w=w; t.wt=wt;  
    return (t);  
}
```



Ahora se coloca en la matriz que cada vértice está a distancia cero consigo mismo.

```
//Asignación dinámica de arreglo bidimensional  
float **MATRIXfloat(int r, int c, float wt)  
{  
    int i, j;  
    float **t = malloc(r * sizeof(float *));
```

```

    for (i = 0; i < r; i++)
        t[i] = malloc(c * sizeof(float));
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            if(i==j) t[i][j] = 0.; else t[i][j] = wt; //se aceptan lazos en vértices
    return t;
}

```

```

void BorreGrafo(Graph G)
{
    int i;
    float **t = G->adj;
    for (i = 0; i < G->V; i++)
        free(t[i]);
    free(t);
    free(G);
}

```

```

Graph GRAPHinit(int V) //Crea grafo vacío, sin enlaces. Sólo los vértices.
{
    Graph G = malloc(sizeof *G); //crea cabecera del grafo
    G->V = V; G->E = 0;
    G->adj = MATRIXfloat(V, V, maxWT);
    return G;
}

```

```

void GRAPHinsertE(Graph G, Edge e) //Inserta enlace
{
    if (G->adj[e.v][e.w] == maxWT) G->E++;
    G->adj[e.v][e.w] = e.wt; // Sólo el enlace dirigido.
}

```

```
}
```

```
void GRAPHremoveE(Graph G, Edge e) //Remueve enlace
```

```
{
```

```
    if (G->adj[e.v][e.w] != maxWT) G->E--;
```

```
    G->adj[e.v][e.w] = maxWT;
```

```
}
```

```
int GRAPHedges(Edge a[], Graph G) //Crea arreglo a de enlaces a partir de G.
```

```
{
```

```
    int v, w, E = 0;
```

```
    for (v = 0; v < G->V; v++)
```

```
        for (w = 0; w < G->V; w++)
```

```
            if ((G->adj[v][w] != maxWT)&&(G->adj[v][w] != 0))
```

```
                a[E++] = EDGE(v, w, G->adj[v][w]);
```

```
    return E;
```

```
}
```

```

//muestra la matriz mediante listas de vértices conectados a cada vértice
void GRAPHshowL(Graph G)
{
    int i, j;
    printf("%d vértices, %d enlaces.\n", G->V, G->E);
    for (i = 0; i < G->V; i++)
    {
        printf("%d: ", i);
        for (j = 0; j < G->V; j++)
            if (G->adj[i][j] != maxWT)
                printf(" %2d-%0.2f", j, G->adj[i][j]); //dos decimales
        putchar('\n');
    }
}

//Muestra Matriz de incidencia.
void GRAPHshowM(Graph G)
{
    int i, j;
    printf("%d vértices, %d enlaces.\n", G->V, G->E);
    printf(" ");
    for (j = 0; j < G->V; j++)
        printf(" %4d ", j);
    printf("\n");
    for (i = 0; i < G->V; i++)
    {
        printf("%2d:", i);
        for (j = 0; j < G->V; j++)
            if (G->adj[i][j] != maxWT)
                printf(" %0.2f", G->adj[i][j]); else printf(" * ");
        putchar('\n');
    }
}

```

Las siguientes definiciones, permiten crear un grafo de seis vértices y 11 enlaces.

```
#define VERTICES 6
#define ENLACES 11
//Variables
Graph Grafo;
Edge Enlaces[ENLACES]={ {0,1,.41},{1,2,.51},{2,3,.50},{4,3,.36},\
                          {3,5,.38},{3,0,.45},{0,5,.29},{5,4,.21},\
                          {1,4,.32},{4,2,.32},{5,1,.29} };
```

Se inicia el grafo con:

```
Grafo = GRAPHinit(VERTICES);
```

Y la inserción de los enlaces con sus pesos se logra con:

```
for(i=0; i<ENLACES; i++)
    GRAPHinsertE(Grafo, Enlaces[i] );
```

La lista de los vértices conectados a cada vértice se logra con:

```
GRAPHshowL(Grafo);
```

La lista de vértices adyacentes incluye 6 vértices, 11 enlaces y sus pesos:

0: 0.00 0.41 0.29	0: 1 5
1: 0.00 0.51 0.32	1: 2 4
2: 0.00 0.50	2: 3
3: 0.45 0.00 0.38	3: 0 5
4: 0.32 0.36 0.00	4: 2 3
5: 0.29 0.21 0.00	5: 1 4

La matriz de adyacencia de la red se muestra con:

GRAPHshowM(Grafo);

Que muestra 6 vértices y 11 enlaces:

	(w)					
	0	1	2	3	4	5
0:	0.00	0.41	*	*	*	0.29
1:	*	0.00	0.51	*	0.32	*
(v) 2:	*	*	0.00	0.50	*	*
3:	0.45	*	*	0.00	*	0.38
4:	*	*	0.32	0.36	0.00	*
5:	*	0.29	*	*	0.21	0.00

### ***Funciones para obtener un path tree a partir de grafo orientado***

El siguiente código se puede usar para obtener un path tree (usando DFS) de un grafo orientado.

```

GRAPHsearchDFS(Grafo);
static int cnt;
static int pre[VERTICES];
static int st[VERTICES];
static float wt[VERTICES];

```

```

void dfsR(Graph G, Edge e)
{
    int t;
    pre[e.w] = cnt++; //orden en que recorre el árbol
    st[e.w] = e.v; //se guarda el vértice v, padre de w.
    //en wt se guarda el peso del enlace v-w más el acumulado desde la raíz
    wt[e.w]=wt[e.v]+G->adj[e.v][e.w];
    for (t = 0; t < G->V; t++)
    {
        if (G->adj[e.w][t] != maxWT)
            if (pre[t] == -1)
            {
                dfsR(G, EDGE(e.w, t, maxWT));
                /*printf("%d-%d \n", e.w, t);*/
            }
    }
}

```

La rutina recursiva anterior es llamada por:

```

void GRAPHsearchDFS(Graph G)
{
    int v;
    cnt = 0;
    for (v = 0; v < G->V; v++) {pre[v] = -1;st[v] = -1;}
    for (v = 0; v < G->V; v++)
        if (pre[v] == -1)
            dfsR(G, EDGE(v, v, maxWT));
}

```

La cual genera el siguiente árbol, para la red descrita anteriormente:

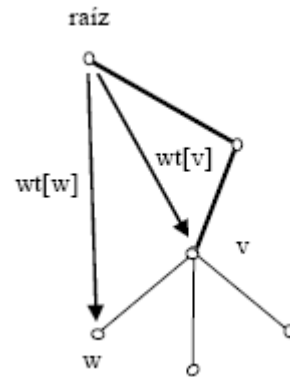
0	1	2	3	4	5	Nodo: i
0	0	1	2	5	3	Padre del nodo: st[i]
0.00	0.41	0.92	1.42	2.01	1.80	Peso ruta nodo a raíz: wt[i]
0	1	2	3	5	4	Orden en que visita los vértices.



La solución previa no es un shortest path tree (SPT).

### Como determinar un Shortest Path Tree: Algoritmo de Dijkstra

- **Determina un SPT (árbol con las mínimas trayectorias, desde un vértice a los demás).**
- No se aceptan enlaces en paralelo, ni enlaces con pesos negativos. El algoritmo consiste inicialmente en colocar el vértice de origen en el SPT.
- Luego, se agrega un enlace por vez. Siempre tomando el enlace que tenga el trayecto más corto entre la fuente y el vértice que no está en el SPT.



Es una solución similar al algoritmo de Prim, pero se van agregando vértices que estén a la menor distancia de la raíz:

Escoger raíz (vértice de origen)

Repetir hasta agregar todos los vértices:

Encontrar un nodo (sea min) cuya distancia a la raíz sea la menor entre todos los nodos no pertenecientes al SPT.

Marcar ese nodo (sea v) como perteneciente al SPT.

Repetir para cada w nodo no perteneciente al SPT:

Si hay conexión entre v y w, y la distancia del nodo raíz a v más la distancia de v a w es menor que la distancia actual de la raíz a w:

Actualizar la distancia de la raíz a w como la distancia de la raíz a v más la distancia de v a w.

Actualizar el nuevo padre de w

Actualizar el vértice que está a distancia mínima.

Se requieren tres arreglos:

```
static int st[VERTICES]; //arreglo de padres
static int fr[VERTICES]; //vértice que dista menos del vértice del árbol.
static float wt[VERTICES+1];
//en wt se guarda el peso del enlace v-w más el acumulado desde la raíz
#define P wt[v]+G->adj[v][w] //distancia de fuente a destino.
```

```

void GRAPHsptDijkstra(Graph G, int raiz)
{
    int v, w, min;
    for (v = 0; v < G->V; v++)
    {
        st[v] = -1; //SPT vacío
        fr[v] = v; //No se conoce cual es el vértice más cercano del SPT.
        wt[v] = VERTICES; //Peso máximo de los trayectos= número de ramas+1
    }
    wt[raiz] = 0; //raíz del SPT a distancia cero.
    wt[G->V] = VERTICES; //centinela. Una posición adicional con peso máximo.
    for (min = raiz; min != G->V; )
    {
        v = min; st[min] = fr[min]; //agrega vértice v al SPT
        for (w = 0, min = G->V; w < G->V; w++)
        //Al inicio min es un vértice que no existe.
        if (st[w] == -1) //si w no está en el SPT
        {
            if ((G->adj[v][w] != maxWT) && (P < wt[w]))
            //Si hay conexión desde v a w y
            //Si la distancia del nodo raíz a v más la distancia de v a w es
            //menor que la distancia actual de la raíz a w
            {
                wt[w] = P; //actualiza distancia de la raíz a w
                fr[w] = v; //salva al padre de w.
            }
        }
        if (wt[w] < wt[min]) min = w; //actualiza el vértice candidato al mínimo
    }
}

```

El siguiente llamado genera un shortest path tree, como un arreglo de padres st, a partir de la red dada anteriormente, con fuente o raíz 2.

GRAPHsptDijkstra(Grafo, 2);

0 1 2 3 4 5 Vértices.

3 5 2 2 5 3 Padre del vértice.

0.95 1.17 0.00 0.50 1.09 0.88 Peso trayecto entre vértice y raíz.

3 5 2 2 5 3 Vértice que dista menos del vértice del árbol

La raíz a distancia **cero** de sí misma.

El SPT con fuente en el vértice 0 se genera con:

GRAPHsptDijkstra(Grafo, 0);

0 1 2 3 4 5 Vértices.

0 0 4 4 5 0 Padre del vértice.

0.00 0.41 0.82 0.86 0.50 0.29 Peso trayecto entre vértice y raíz.

0 0 4 4 5 0 Vértice que dista menos del vértice del árbol

El SPT con fuente en el vértice 1 se genera con GRAPHsptDijkstra(Grafo, 1);

0 1 2 3 4 5 Vértices.

3 1 1 4 1 3 Padre del vértice.

1.13 0.00 0.51 0.68 0.32 1.06 Peso trayecto entre vértice y raíz.

3 1 1 4 1 3 Vértice que dista menos del vértice del árbol

GRAPHsptDijkstra(Grafo, 5), produce:

0 1 2 3 4 5 Vértices.

3 5 4 4 5 5 Padre del vértice.

1.02 0.29 0.53 0.57 0.21 0.00 Peso trayecto entre vértice y raíz.

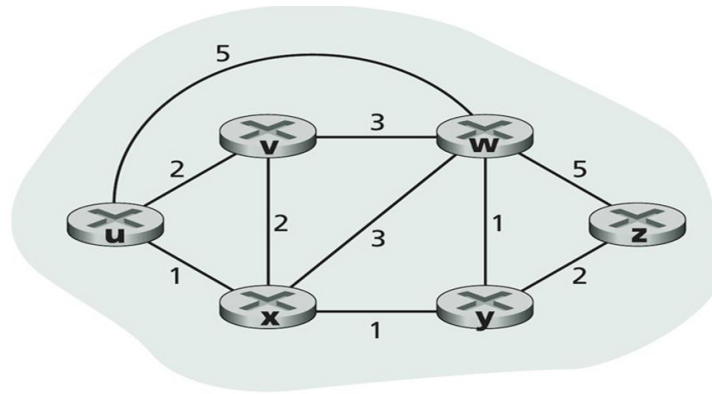
GRAPHsptDijkstra(Grafo, 4), produce:

0 1 2 3 4 5 Vértices.

3 5 4 4 4 3 Padre del vértice.

0.81 1.03 0.32 0.36 0.00 0.74 Peso trayecto entre vértice y raíz.

Ejemplo:



step	$N'$	$D(v),p(v)$	$D(w),p(w)$	$D(x),p(x)$	$D(y),p(y)$	$D(z),p(z)$
0	u	2,u	5,u	<u>1,u</u>	$\infty$	$\infty$
1	ux	2,u	4,x		<u>2,x</u>	$\infty$
2	uxy	<u>2,u</u>	3,y			4,y
3	uxyv		<u>3,y</u>			4,y
4	uxyvw					<u>4,y</u>
5	uxyvwz					

## Referencias

- 1- Sedgewick, R., Algorithms in C, 3rd ed, Addison Wesley 2002
- 2- Apuntes del Prof. Leopoldo Silva Bijit, curso EDA-320, UTFSM, 2005-1
- 3- Kernighan, B., Ritchie, D., The C Programming Language, Prentice Hall, 1988
- 4- <http://es.wikipedia.org/wiki/Grafos>
- 5- [http://en.wikipedia.org/wiki/Kruskal's\\_algorithm](http://en.wikipedia.org/wiki/Kruskal's_algorithm)
- 6- [http://en.wikipedia.org/wiki/Prim's\\_algorithm](http://en.wikipedia.org/wiki/Prim's_algorithm)
- 7- [http://en.wikipedia.org/wiki/Dijkstras\\_Algorithm](http://en.wikipedia.org/wiki/Dijkstras_Algorithm)
- 8- Javascript Dijkstra: [http://www.itonsite.co.uk/allanboydproject/section4\\_2.htm](http://www.itonsite.co.uk/allanboydproject/section4_2.htm)
- 9- Javascript Kruskal: <http://students.ceid.upatras.gr/~papagel/project/kruskal.htm>