Análisis y Diseño de Algoritmos

Algunos Algoritmos Sobre Gráficas

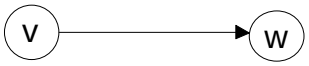
Arturo Díaz Pérez
Sección de Computación
Departamento de Ingeniería Eléctrica
CINVESTAV-IPN
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D. F. CP 07300

Tel. (5)747 3800 Ext. 3755
e-mail: adiaz@cs.cinvestav.mx

Contenido

Grafos Dirigidos

- Una gráfica dirigida o digrafo es una estructura (V, A) donde V es un conjunto de elementos llamados vértices, y A es un conjunto de pares ordenados (v,w) llamados arcos o aristas.
- Sea (v,w) un arco de un digrafo G, éste se expresa frecuentemente por $v \rightarrow w$ y se dibuja como:



- Se dice que el arco va de v a w, y que w es adyacente a v.

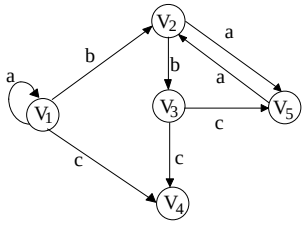
Análisis y Diseño de Algoritmos *GraphAlg-3*

Grafos Dirigidos

- Un camino en un digrafo es una secuencia de vértices v_1, v_2, ..., v_n, tal que, (v_1,v_2), (v_2,v_3), ..., (v_{n-1},v_n) son arcos. Este camino va de v_1 a v_n y pasa por v_2, v_3, ..., y v_{n-1}.
- La longitud de un camino es el número de arcos en él. Un camino que consta de un sólo vértice es un camino de longitud 0.
- Un camino es simple si todos sus vértices, excepto posiblemente el primero y el último, son diferentes. Un ciclo simple es un camino simple de longitud mayor o igual a 1 y que inicia y termina en el mismo vértice.
- Un digrafo etiquetado es un digrafo en el cual sus vértices y/o aristas tienen asociada una etiqueta. Una etiqueta es un valor de cualquier tipo.

Análisis y Diseño de Algoritmos *GraphAlg-4*

Ejemplo



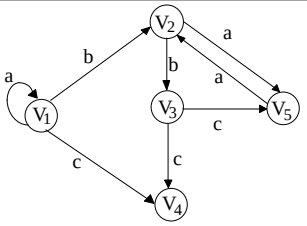
$\boldsymbol{G} = (V,A)$

$V = \{ V_1, V_2, V_3, V_4, V_5 \}$

$A = \{ (V_1,V_1), (V_1,V_2), (V_1,V_4), (V_2,V_3), (V_2,V_5), (V_3,V_4), (V_3,V_5), (V_5,V_2) \}$

Caminos	**Longitud**	
$C_1 = V_1, V_2, V_3, V_4$	3	**simple**
$C_2 = V_1, V_2, V_3, V_5, V_2, V_3$	5	**no simple**
$C_3 = V_2, V_5$	1	**simple**
$C_4 = V_2, V_3, V_5, V_2$	3	**ciclo simple**

Matrices de Adyacencia

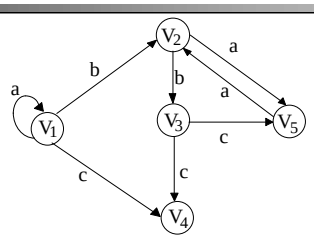


$$M[v,w] = \begin{cases} 1 & \text{si existe un arco } (v,w) \\ 0 & \text{en caso contrario} \end{cases}$$

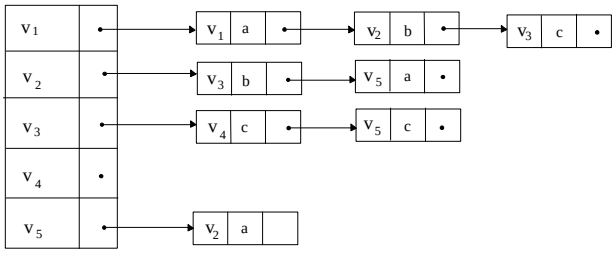
	V_1	V_2	V_3	V_4	V_5
V_1	1	1	0	1	0
V_2	0	0	1	0	1
V_3	0	0	0	1	1
V_4	0	0	0	0	0
V_5	0	1	0	0	0

	V_1	V_2	V_3	V_4	V_5
V_1	a	b		c	
V_2			b		a
V_3				c	c
V_4					
V_5		a			

Listas de Adyacencia



$L_v = \{ w \in V \mid (v, w) \in A \}$



Análisis y Diseño de Algoritmos *GraphAlg-7*

Recorrido en Profundidad

- Suponga que se tiene un grafo dirigido G en el cual todos los vértices se marcan inicialmente como no visitados.
- El recorrido en profundidad selecciona un vértice *v* de G como vértice inicial; y se marca como visitado.
  - Cada vértice adyacente a *v* no visitado se visita en turno, usando el recorrido en profundidad recursivamente.
  - Una vez que todos los vértices que se alcanzan desde *v* han sido visitados, el recorrido desde *v* se ha terminado.
  - Si algunos vértices de *G* permanecen como no visitados, se selecciona un vértice no visitado como nuevo vértice inicial. Se repite el proceso hasta que todos los vértices de *G* han sido visitados.

Análisis y Diseño de Algoritmos *GraphAlg-8*

Recorrido en Profundidad

- Supongamos que para cada vértice *v* existe una lista de vértices adyacentes a *v*, **L**[*v*].
- Supongamos además que en un arreglo Marca se indica si un vértice ha sido visitado o no visitado.

```
GRAFO.h:
typedef     . . .      VERTICE;
typedef     . . .       LISTA_ADYACENTES;
typedef     . . .       GRAFO;
.
      .
DFS.C:
#include   "GRAFO.h"
#define     VISITADO           . . .
#define     NO_VISITADO   . . .
#define     MAX_VERTICE  . . .

int  Marca[MAX_VERTICE];
```

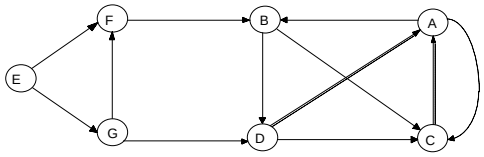

Recorrido en Profundidad

```
void  DFS( VERTICE  V )
{
   VERTICE   W;
   Marca[V] = VISITADO;
   for( cada vértice w en L[V] )
      if( Marca[W] == NO_VISITADO )
         DFS(W);
}

main()
{
   .
   for( cada vértice v del grafo )
      Marca[V] = NO_VISITADO;

   for( cada vertice v del grafo )
      if( Marca[V] = NO_VISITADO )
         DFS(V);
   .
}
```

Recorrido en Profundidad

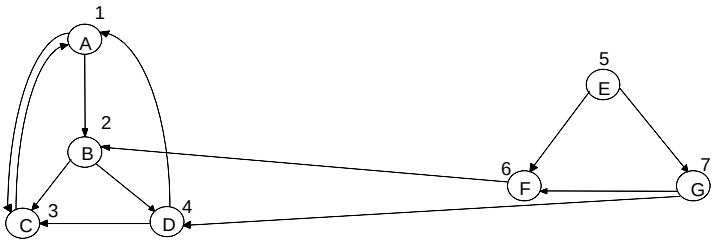
```
void  DFS( VERTICE  V )
{
   VERTICE   W;
   Marca[V] = VISITANDO;
   for( cada vértice W en L[V] ) {
      if( Marca[W] == NO_VISITADO ) {
        Lista de descendientes de W <- W;
        DFS(W);
        El archo (V,W) es de árbol;
        Agrega la lista de descendientes de W a la de V;
      } else if( Marca[W] == VISITANDO )
        El archo (V,W) es hacia atrás;
      else if( W en la lista de descendientes de V )
        El archo (V,W) es hacia adelante;
      else
        El archo (V,W) es cruzado;
   }
   Marca[V] = VISITADO;
}
```


DFS: Ejemplo

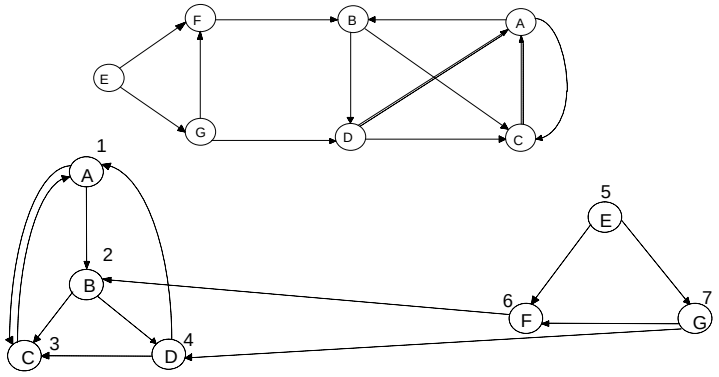


Recorrido en profundidad: A B C D E F G

Bosque de expansión del recorrido en profundidad



DFS: Ejemplo



Un arco **de árbol** es un arco que lleva a un vértice no visitado.
Un arco **hacia atrás** lleva de un descendiente a un ancestro propio.
Un arco **hacia adelante** lleva de un ancestro a un descendiente propio
Un arco **cruzado** es un arco que va entre vértices que ni son descendientes ni ancestros uno del otro.

Recorrido en Profundidad

```
int time = 0;

main()
{
   .
   for( cada vértice v del grafo ) {
      Marca[V] = NO_VISITADO;
      Ini[V] = α;
   }

   for( cada vertice v del grafo )
      if( Marca[V] = NO_VISITADO )
         DFS(V);
   .
}
```

Recorrido en Profundidad

```
void  DFS( VERTICE  V )
{
   VERTICE   W;
   Marca[V] = VISITANDO;
   time = time + 1; Ini[V] = time;
   for( cada vértice W en L[V] ) {
      if( Marca[W] == NO_VISITADO ) {
         DFS(W);
         El archo (V,W) es de árbol;
      } else if( Marca[W] == VISITANDO )
         El archo (V,W) es hacia atrás;
      else if( Ini[V] < Ini[W])
         El archo (V,W) es hacia adelante;
      else
         El archo (V,W) es cruzado;
   }
   Marca[V] = VISITADO;
}
```


Grafo Dirigidos Acíclicos

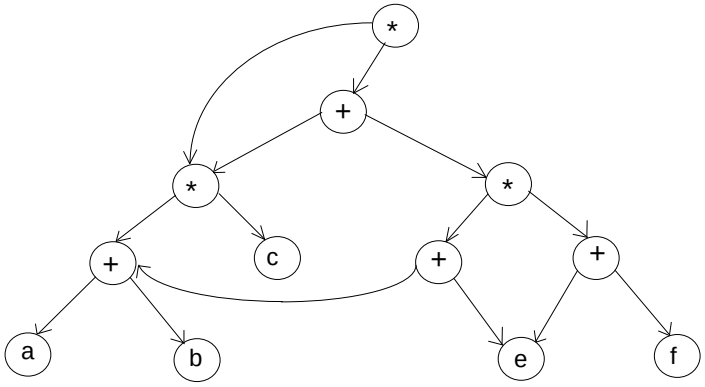
- Un **grafo dirigido acíclico** (DAG) es un grafo dirigido en el que no existen ciclos.
  - Los grafos dirigidos acíclicos son más generales que los árboles pero menos generales que los grafos dirigidos arbitrarios.

- Ejemplo
  - Representación de la estructura sintáctica de expresiones aritméticas con expresiones comúnes

    ((a+b)*c + ((a+b)+e)*(e+f)) * ((a+b)*c)

DAG Ejemplo

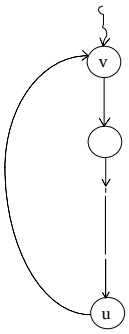
((a+b)*c + ((a+b)+e)*(e+f)) * ((a+b)*c)



Análisis y Diseño de Algoritmos *GraphAlg*-17

Prueba de Aciclidad

- Sea $G = (V, A)$ un grafo dirigido. G tiene un ciclo, si y solo si, se encuentra un arco hacia atrás en el recorrido en profundidad.
  - ⇐ Claramente, si se encuentra un arco hacia atrás en el recorrido en profundidad, ***G*** tiene un ciclo.

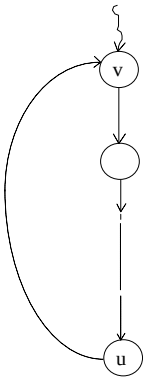


Análisis y Diseño de Algoritmos *GraphAlg*-18

Prueba de Aciclidad

⇒ Supongamos que ***G*** es cíclico.

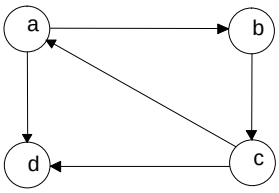
- Supongamos que al hacer el recorrido en profundidad de ***G***, los vértices se van numerando consecutivamente conforme ellos se van marcando.
- Sea *v* el vértice con la numeración más pequeña de los vértices que aparecen en un ciclo de ***G***.
- Ya que *v* está en un ciclo, considere un arco (*u*, *v)*, en el ciclo.
- *u* debe estar en el ciclo también y debe tener una numeración mayor.
- Por lo tanto, *u* debe ser un descendiente de *v* en el bosque de expansión del recorrido en profundidad.
- El arco (*u*, *v*) no puede ser un arco de árbol ni un arco hacia adelante.
- Tampoco puede ser un arco cruzado.
- Por lo tanto, (*u*, *v*) debe ser un arco hacia atrás.



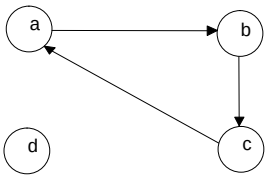
Componentes Fuertemente Conectadas

- Sea $G = (V, A)$ un grafo dirigido. Se define la relación R, tal que, $(a, b) \in R$, si y solo si, existe un camino de a a b y existe un camino de b a a.
  - ← R es una relación de equivalencia e induce una partición sobre V.
- Sean V_i, $i = 1, ..., k$ las clases de equivalencia de v.
- Se definen los conjuntos $A_i = \{ (a, b) \in A \mid a, b \in V_i \}$, esto es, los arcos que salen y llegan a miembros de la misma clase de equivalencia.
- Los grafos $G_i = (V_i, A_i)$, $i = 1, \ldots, k$ se llaman las componentes fuertemente conectadas de G.
- Sean $V_R = \{V_i \mid i = 1, ..., k\}$ y $A_R = \{ (V_i, V_j) \mid \exists a \in V_i$ y $\exists b \in V_j$ tales que $(a, b) \in A\}$
- A $G_R = (V_R, A_R)$ se le llama el grafo reducido de G.

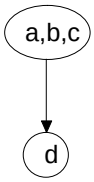
Componentes Fuertemente Conectadas



Grafo dirigido



Componentes fuertemente conectadas

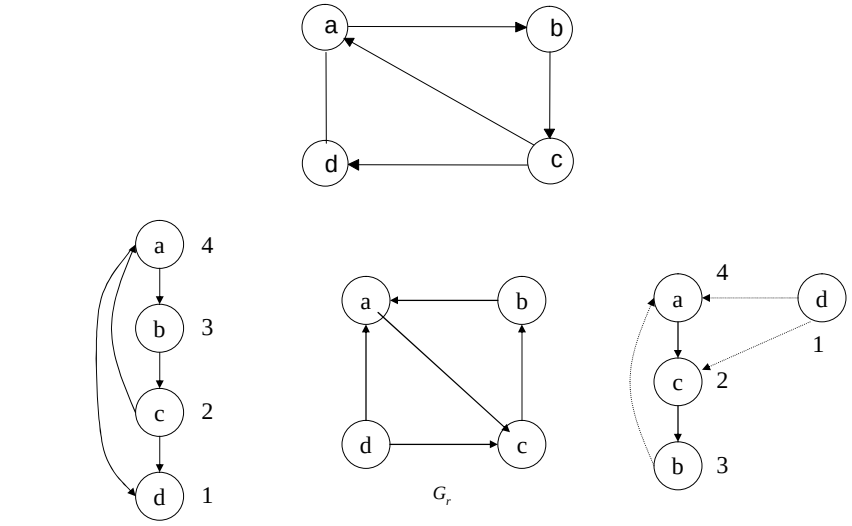


Grafo Reducido

Componentes Fuertemente Conectadas

- Sea $G = (V, A)$ un grafo dirigido.
  - Ejecutar el recorrido en profundidad de G y numerar los vértices en el orden en que se completan sus llamados recursivos.
  - Construir un nuevo grafo G_r invirtiendo la dirección de cada arco de G, esto es, $G_r = (V, A_r)$, donde $A_r = \{ (b, a) \mid (a, b) \in A \}$
  - Ejecutar el recorrido en profundidad de G_r empezando en el vértice con la numeración más alta obtenida en el primer paso.
  - Cada árbol en el bosque de expansión del recorrido en profundidad de G_r es una componente fuerte de G.

Componentes Fuertemente Conectadas

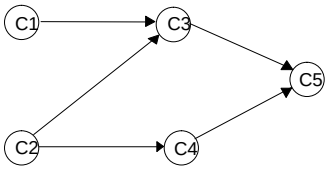


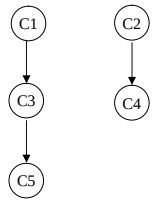
Orden Topológico

- Sea $G = (V, A)$ un grafo dirigido acíclico.
- El *orden topológico* es el proceso de asignar un ordenamiento lineal a los vértices de G, tal que, si $(i, j) \in A$, entonces, i aparece antes que j, en el ordenamiento lineal.
- Un orden topológico se puede obtener, en forma invertida, listando cada vértice cuando el recorrido en profundidad de sus descendientes ha terminado.

Orden Topológico

```
void  topsort ( VERTICE V )
{
    VERTICE  W;

    Marca[V] = VISITADO;
    for( Cada vértice W en L[V] ) {
        if( Marca [W] == NO_VISITADO )
            topsort( W)
    }
    printf(". . . .", V);
}
```





- C1 C2 C3 C4 C5
- C2 C4 C1 C3 C5

C5 C3 C1 C4 C2

Los Caminos Más Cortos Desde Un Origen

- Sea G = (V,A) un grafo en el cual cada arco tiene asociada una etiqueta la cual es un valor no negativo denominado costo.
  - En el grafo existe un vértice que se conoce como el origen.
  - El costo de un camino en el grafo se define como la suma de los costos en sus arcos.
  - El problema es encontrar los caminos más cortos (de menor costo) desde el origen a cualquier otro vértice del grafo.

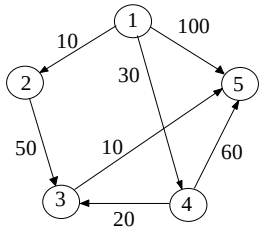
Algoritmo de Dijkstra

- Supongamos que $G = (V,A)$ es tal que $V = \{1, ..., n\}$ en donde 1 es el origen y C es tal que C[i,j] es el costo del arco que va de i a j.
- En cada paso del algoritmo se mantiene un conjunto de vértices, S, cuya distancia más corta desde el origen ya se conoce.
- En cada paso, se agrega a S uno de los vértices restantes, v, cuya distancia desde el origen es tan corta como sea posible.
- El algoritmo encuentra el camino más corto del origen a v que pasa únicamente por vértices en S.
- Termina cuando S incluye todos los vértices.

Algoritmo de Dijkstra

```
void Dijkstra( void )
{
  s = {1};
  for( i = 2; i <= n; i++ )
    D[i] = C[1,i];
  for( i = 1; i <= n-1; i++ )  {
    elegir el vértice w en V-S, tal que, D[w] es mínimo
    agregar w a S;
    for( cada vértice v en V-S )
      D[v] = min( D[v], D[w] + C[w,v] )
  }
}
```

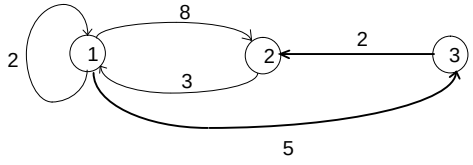
Algoritmo de Dijkstra



Iteración	*S*	*w*	*D*[2]	*D*[3]	*D*[4]	*D*[5]	*P*[2]	*P*[3]	*P*[4]	*P*[5]
	{1}	-	10	∞	30	100	-	-	-	-
1	{1,2}	2	10	60	30	100	-	2	-	-
2	{1,2,4}	4	10	50	30	90	-	4	-	4
3	{1,2,4,3}	3	10	50	30	60	-	4	-	3
4	{1,2,4,3,5}	5	10	50	30	60	-	4	-	3

Los Caminos Más Cortos Entre Cada Par

- Sea $G = (V,A)$ un digrafo en el cual cada arco tiene asociado un costo no negativo. El problema es hallar para cualquier par de vértices (v,w) el camino más corto de v a w.



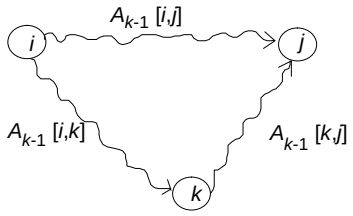
Algoritmo de Floyd

- $G = (V,A)$, $V = \{1, ..., n\}$, y C[i,j] es el costo del arco que va de i a j.
  - ← El algoritmo cálcula la serie de matrices

$$A_0(i,j) = \begin{cases} 0 & si \quad i = j \\ C[i,j] & si \quad i \neq j. \end{cases}$$

$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$$

  - ← $A_k[i,j]$ significa el costo del camino más corto que va de i a j y que no pasa por algún vértice mayor que k.

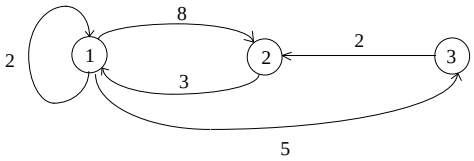


Algoritmo de Floyd

$$\boldsymbol{A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])}$$

- Significa el camino más corto que va de i a j sin pasar (entrar y salir) por un vértice con númeración mayor que k.
- El objetivo es calcular $\boldsymbol{A_n[i,j]}$

Algoritmo de Floyd: Ejemplo



A_0	1	2	3
1	0	8	5
2	3	0	α
3	α	2	0

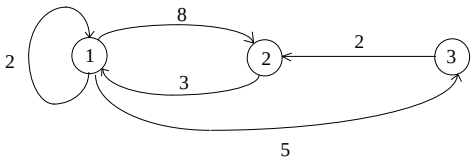
A_1	1	2	3
1	0	8	5
2	3	0	8
3	α	2	0

A_2	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

A_3	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Análisis y Diseño de Algoritmos *GraphAlg-33*

Recuperación de Caminos



A_0	1	2	3
1	0	8	5
2	3	0	α
3	α	2	0

A_1	1	2	3
1	0	8	5
2	3	0	8
3	α	2	0

A_2	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

A_3	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

P_0	1	2	3
1	0	0	0
2	0	0	-1
3	-1	0	0

P_1	1	2	3
1	0	0	0
2	0	0	1
3	-1	0	0

P_2	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

P_3	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Análisis y Diseño de Algoritmos *GraphAlg-34*

El Centro de un Grafo

- Sea $G = (V,A)$ un grafo dirigido con matriz de costos C.
- Sea $v \in V$ un vértice del digrafo.
- La excentricidad de v se define como

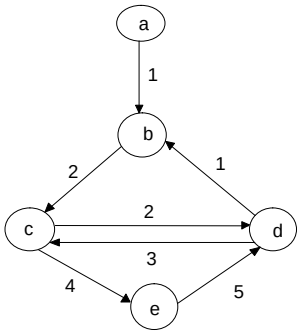
$$\max_{w}(la \quad longitud \quad del \quad camino \quad más \quad corto \quad de \quad w \quad a \quad v)$$

- El centro de un grafo es el vértice con la mínima excentricidad.

El Centro de un Grafo

- Encontrar el centro de un digrafo se puede realizar aplicando los pasos siguientes:
  - Aplicar el algoritmo de Floyd para encontrar la longitud de los caminos más cortos entre cualesquiera par de vértices.
    - El resultado se representa en la matriz *A*.
  - Hallar el costo máximo en cada columna *i*.
    - Esto proporciona la excentricidad del vértice *i*.
  - Hallar el vértice con la mínima excentricidad.
    - Esto proporciona el centro de *G*.

El Centro de un Grafo



A	a	b	c	d	e
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	∞	**6**	**8**	**5**	**7**

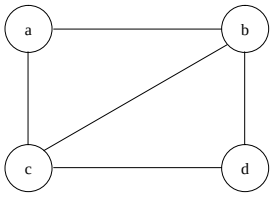
el centro el grafo es d

Grafos No Dirigidos

- Un grafo no dirigido es una estructura (V, A), donde, V es un conjunto de finito de elementos llamados vértices, y $A \subset V \text{x} V$ es un conjunto de pares ordenados (u, v) llamados arcos o aristas que cumple con la propiedad de simetría, esto es,

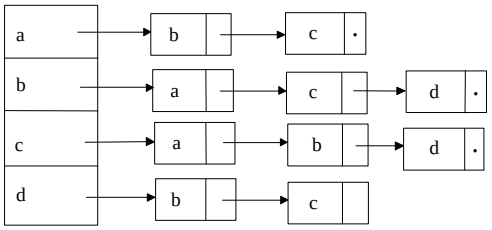
$$\text{Si } (u, v) \in A \Rightarrow (v, u) \in A$$

Grafos No Dirigidos: Representación



	a	b	c	d
a	0	1	1	0
b	1	0	1	1
c	1	0	1	1
d	0	1	1	0

Matriz de adyacencia



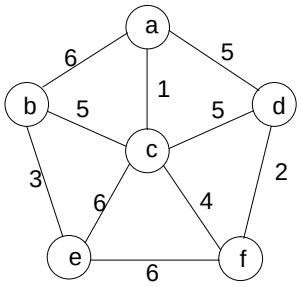
Listas de adyacencia

Arboles Generadores

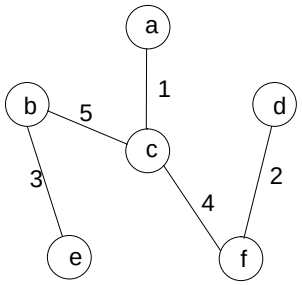
- Sea $G = (V, A)$ un grafo conectado en el cual cada arco (u, v) tiene asociado un costo $c(u, v)$.
- Un *árbol libre A* es un subgrafo de *G* que es acíclico.
- Un *árbol de generador* para *G* es un árbol libre que conecta todos los vértices en *V*.
- El *costo de un árbol generador* es la suma de los costos de arcos en el árbol.

- Un problema interesante es encontrar el árbol generador de una gráfica de costo mínimo.

Arboles Generadores



Grafo no dirigido



Arbol generador de costo mínimo

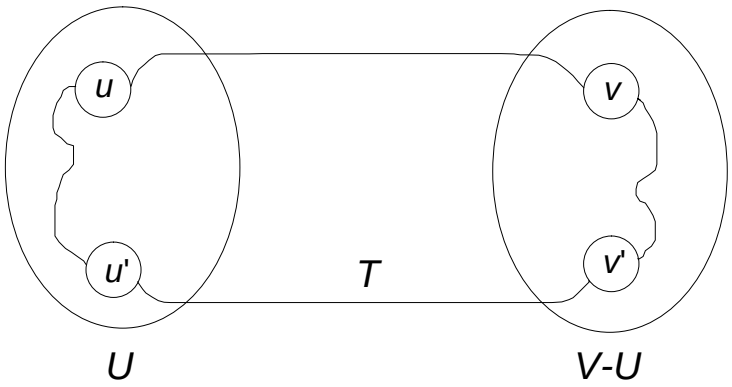
Arboles de Costo Mínimo: Propiedad

- Sea $G = (V, A)$ un grafo definido como antes. Sea U algún subconjunto propio del conjunto de vértices V.
- **Propiedad**: Si (u, v) es el arco de menor costo tal que $u \in U$ y $v \in V\text{-}U$, entonces, existe un árbol generador de costo mínimo que incluye a (u, v) como arco.

Arboles de Costo Mínimo: Propiedad

- Demostración
  - Suponga que no existe árbol generador de costo mínimo alguno para *G* que incluya a (*u*, *v*) como arco.
    - (*u*, *v*) el arco de menor costo
  - Sea *T* cualquier árbol generador de costo mínimo para *G*.
  - El agregar (*u*, *v*) a *T* debe introducir un ciclo ya que *T* es un árbol libre.
  - Este ciclo involucra al arco (*u*, *v*). Así debe existir otro arco (*u*', *v*') tal que *u*' Î *U* y *v*' Î *V-U*
  - Al borrar el arco (*u*', *v*') se rompe el ciclo y produce un árbol generador *T'* cuyo costo no es mayor que el costo de *T* ya que se supuso que *c*(*u*, *v*) <= *c*(*u*', *v*') .
  - Así, *T'* contradice la suposición que no existe árbol generador de costo mínimo que incluya a (*u*, *v*).

Arboles de Costo Mínimo



Algoritmo de Prim

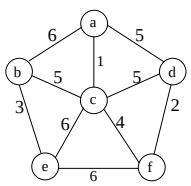
- Inicia con el conjunto *U* consistiendo de un solo vértice, cualquiera.
- Construye un árbol generador, un arco en cada paso.
  - En cada paso encuentra el arco de costo menor (*u*, *v*) que conecta *U* y *V-U*
  - Agrega *v*, el vértice en *V-U*, a *U*.
  - Se repite la secuencia de pasos hasta que *U*=*V*.

Algoritmo de Prim

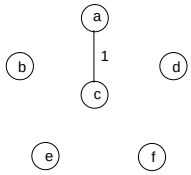
```
void Prim( GRAFO G, CONJUNTO T )
{
   CONJUNTO_V  U;
   VERTICE     v;

   T = ∅;
   U = { cualquier vértice de G };
   while( U != V ) {
     Sea (u,v) es el arco de menor costo tal que
         u ∈ U  y  v ∈ V-U;
     T = T ∪ { (u,v) };
     U = U ∪ { v };
   }
}
```

- La complejidad en tiempo del algoritmo de Prim es $O(n^2)$, donde, *n* es el número de vértices del grafo.

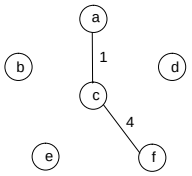
Algoritmo de Prim: Ejemplo



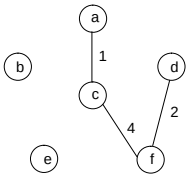
Grafo original



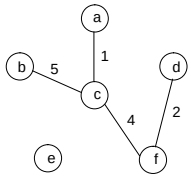
a)



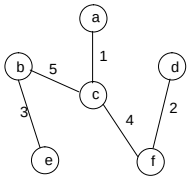
b)



c)



d)



e)

Análisis y Diseño de Algoritmos *GraphAlg*-47

Algoritmo de Kruskal

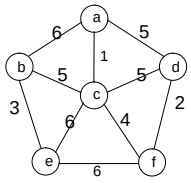
- Inicia con un bosque de árboles consistiendo de un vértice cada uno.
- Construye un árbol generador, un arco en cada paso.
  - En cada paso encuentra el arco de costo menor (*u*, *v*) que conecta un árbol con otro.
  - Mezcla los árboles de *u* y de *v* en uno solo
  - Agrega (*u*, *v*) al árbol generador
  - Se repite la secuencia de pasos hasta que el bosque consista de un solo árbol.

Análisis y Diseño de Algoritmos *GraphAlg*-48

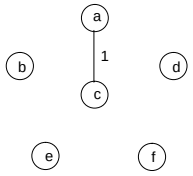
Algoritmo de Kruskal

```
void Kruskal( GRAFO G, CONJUNTO T )
{
   CONJUNTO_V  U;
   VERTICE     v;

   T = ∅;
   for( cada vértice v de G )
     construye un árbol con v;
   Ordena los arcos de G en orden no decreciente;
   while( Haya más de un árbol ) {
     Sea (u,v) es el arco de menor costo tal que el árbol
     de u es diferente al árbol de v;
     Mezcla los árboles de u y de v en uno solo;
     T = T ∪ { (u,v) };
   }
}
```

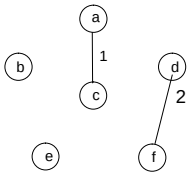

Algoritmo de Kruskal: Ejemplo



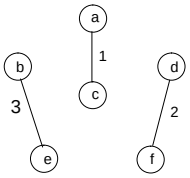
Grafo original



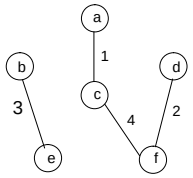
a)



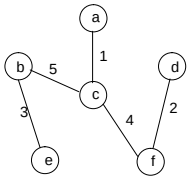
b)



c)



d)



e)