Búsqueda "*Primero a lo Ancho*" (*Breadth-First Search*: BFS)

Bibliografía: "Introduction to Algorithms". Second Edition.

The MIT Press. Massachusetts Institute of Technology. Cambridge, Massachusetts 02142.

http://mitpress.mit.edu

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

© Departamento de Programación
Facultad de Matemática y Computación
Universidad de La Habana

Longitud mínima entre dos vértices

- Se define **longitud mínima** entre dos vértices $\boldsymbol{L(s,v)}$ como el **número mínimo** de aristas existentes en cualquier camino de s a v. Si no existe camino entre s y v, entonces $\boldsymbol{L(s,v)} = \infty$

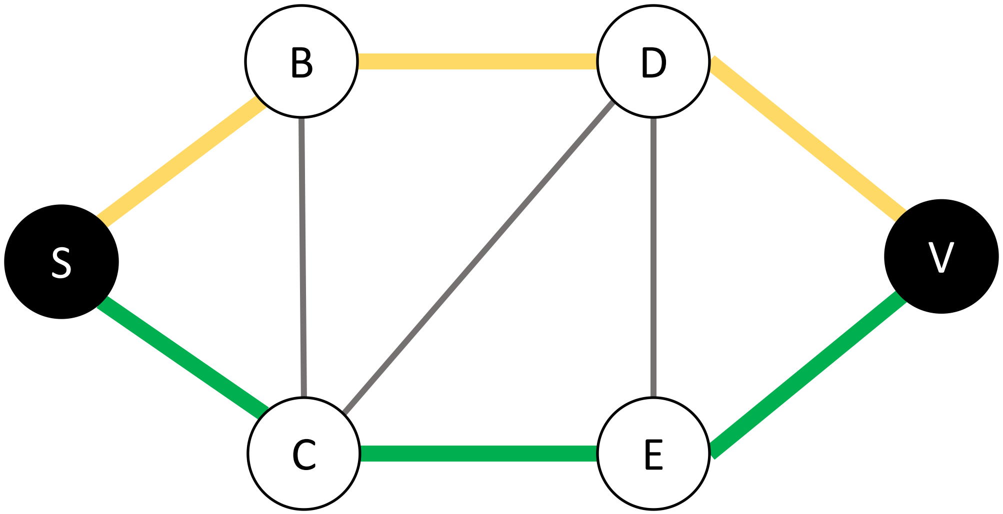
- Se le llama **camino de longitud mínima** al camino de longitud $\boldsymbol{L(s,v)}$

PROBLEMA

Dado dos vértices s y v calcule $\boldsymbol{L(s,v)}$ y devuelva el **camino de longitud mínima** entre ellos

Longitud mínima entre dos vértices

Sea $G =< V, E >$ grafo no dirigido. Dado dos vértices $s, v \in V$, calcule $L(s, v)$ y devuelva el **camino de longitud mínima**



NOTAS

El camino de longitud mínima no necesariamente es único

- (S, C, E, V)
- (S, B, D, V)

$$L(s, v) = 3$$

Algoritmo BFS Simple

INICIALIZACIÓN

$$\boldsymbol{Q} = [\boldsymbol{s}]$$
$$\mathrm{d[v]} = \infty \quad \forall \boldsymbol{v} \in \boldsymbol{V}$$

d[s] = 0

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] + 1
          ENQUEUE(Q, v)
```

Si $\boldsymbol{d}[\boldsymbol{v}] \neq \infty$ entonces el **BFS** ya visitó a $\boldsymbol{v}$ por lo que no se adiciona a la **COLA**

- Se utiliza una **COLA**-*QUEUE***(FIFO)**
- Inicialmente la **COLA** contiene, únicamente, al vértice de partida $\boldsymbol{s}$
- Por cada vértice que se extrae de la **Cola**, se ***insertan*** en ella sus **vértices adyacentes**
- Se lleva un *array* $\boldsymbol{d}[\boldsymbol{v}]$ que representa la menor distancia de ***s*** a ***v*** calculada hasta un momento dado de la ejecución del algoritmo. Al acabar la misma, ***d[v]=*** $\boldsymbol{L}(\boldsymbol{s},\boldsymbol{v})$; $\forall v \in V$ tal que; **exista un camino de *s* a *v***

Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
   do u ← DEQUEUE(Q)
      for each v ∈ Adj[u]
         do if d[v] = ∞
            then d[v] ← d[u] +
1
                 ENQUEUE(Q, v)
```

ESTADO INICIAL

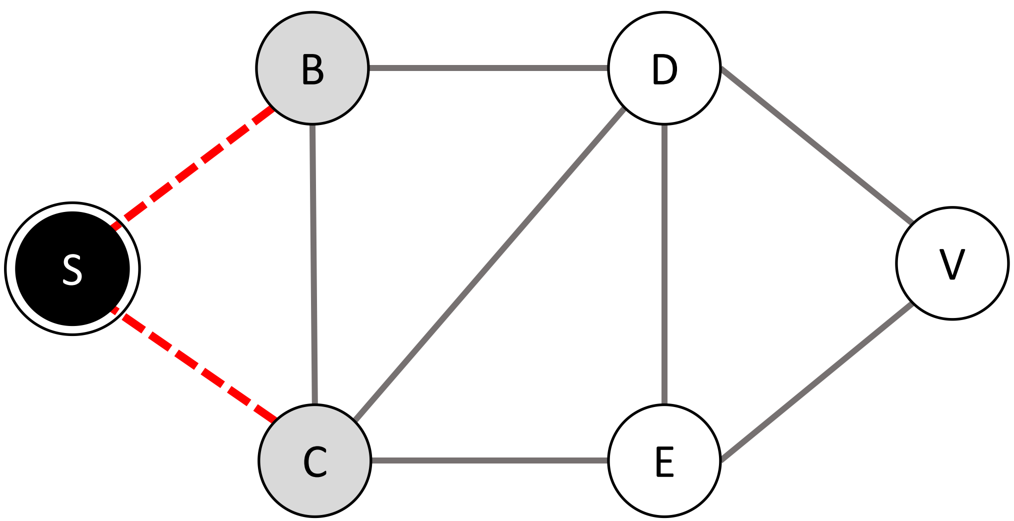
$$Q = [S]$$

	S,	B,	C,	D,	E,	V
d =	0	∞	∞	∞	∞	∞

ESTADO FINAL

$$Q = [B, C]$$

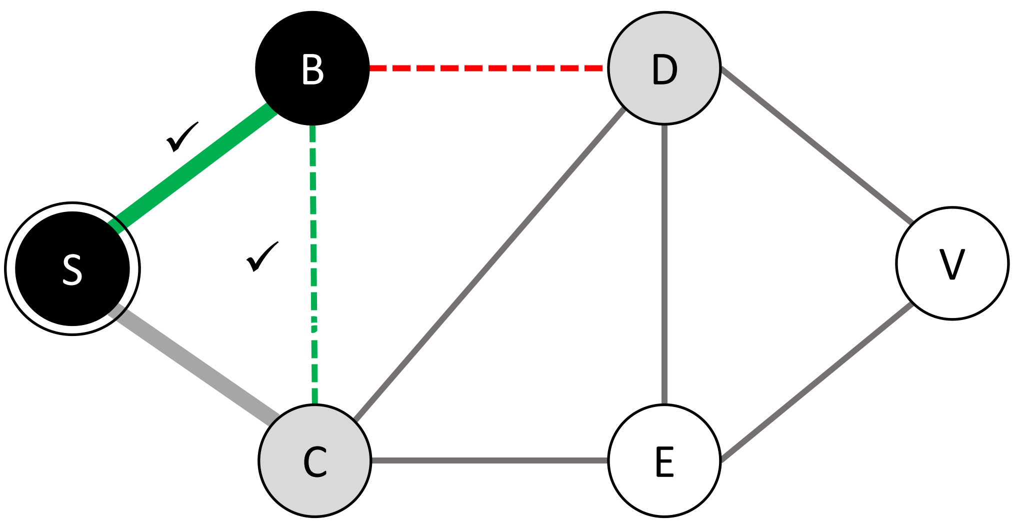
	S,	**B**,	**C**,	D,	E,	V
d =	0	1	1	∞	∞	∞



No ha sido procesado   Está en la cola   Salió de la cola

Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] +
1
             ENQUEUE(Q, v)
```



ESTADO INICIAL

$$Q = [B, C]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	∞	∞	∞

ESTADO FINAL

$$Q = [C, D]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	∞	∞

No ha sido procesado

Está en la cola

Salió de la cola

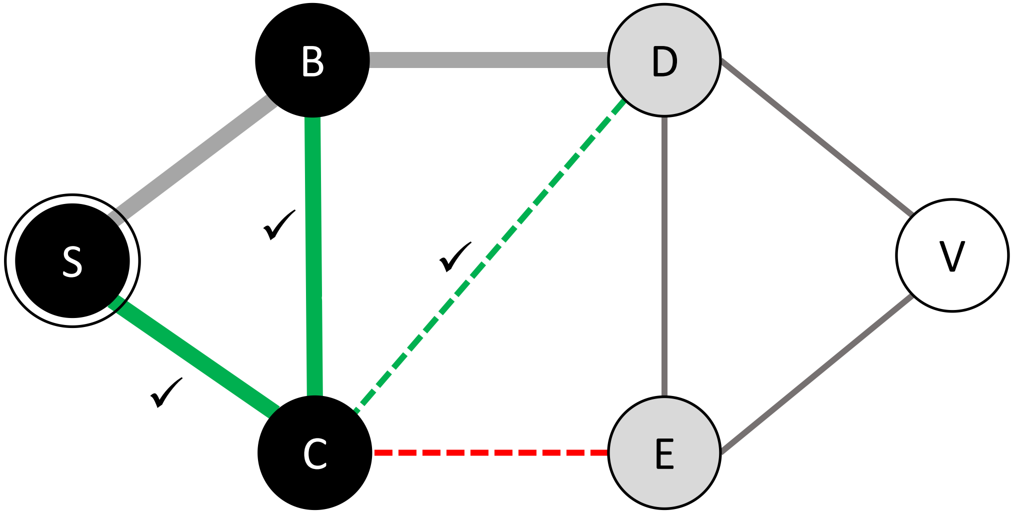
Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] +
1
             ENQUEUE(Q, v)
```

ESTADO INICIAL

$$Q = [C, D]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	∞	∞



ESTADO FINAL

$$Q = [D, E]$$

	S,	**B,**	**C,**	**D,**	**E,**	V
d =	0	1	1	2	2	∞

No ha sido procesado Está en la cola Salió de la cola

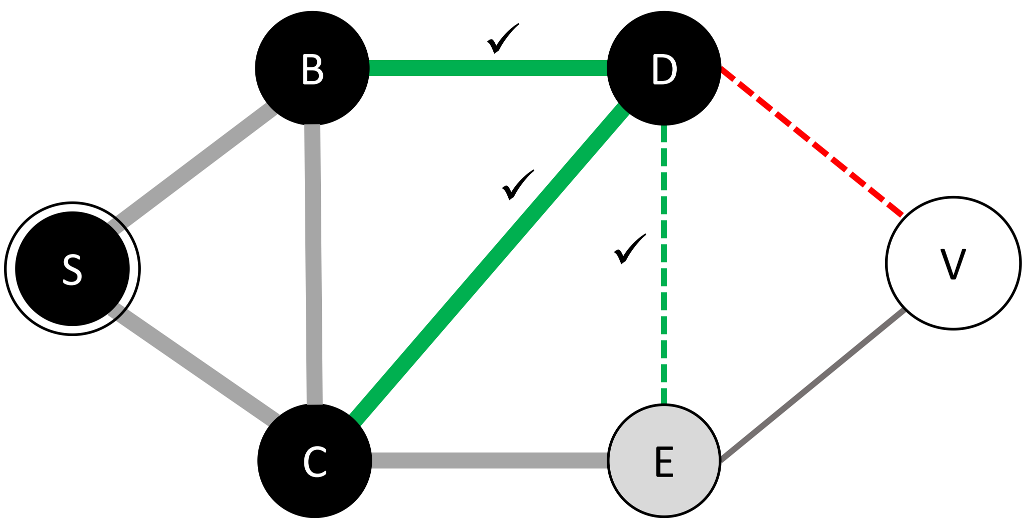
Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] + 1
             ENQUEUE(Q, v)
```

ESTADO INICIAL

$$Q = [D, E]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	2	∞



ESTADO FINAL

$$Q = [E, V]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	2	3

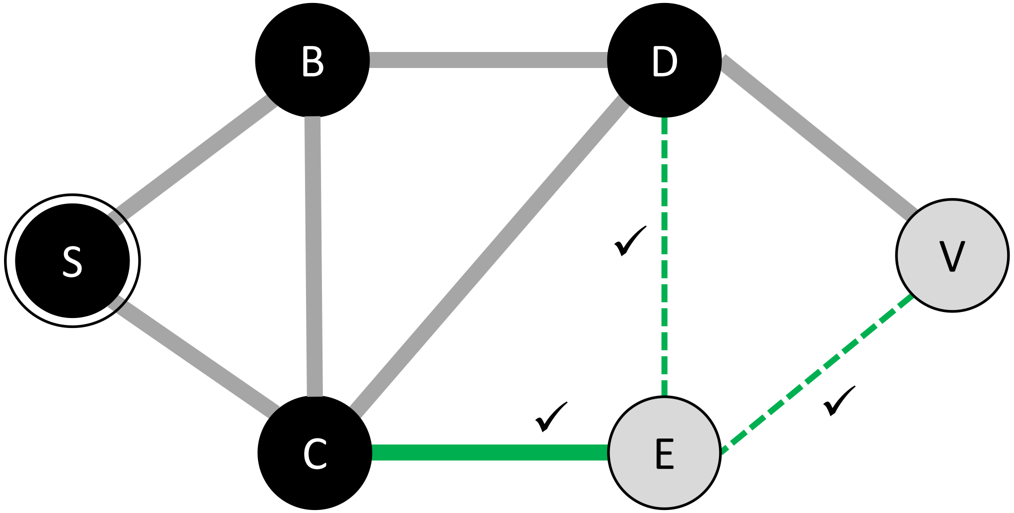
Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] +
1
             ENQUEUE(Q, v)
```

ESTADO INICIAL

$$Q = [E, V]$$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	2	∞



ESTADO FINAL

$$Q = [V]$$

	S,	**B,**	**C,**	**D,**	**E,**	**V**
d =	0	1	1	2	2	3

No ha sido procesado Está en la cola Salió de la cola

Algoritmo BFS Simple - Ejemplo

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
        then d[v] ← d[u] + 
1
          ENQUEUE(Q, v)
```

ESTADO INICIAL

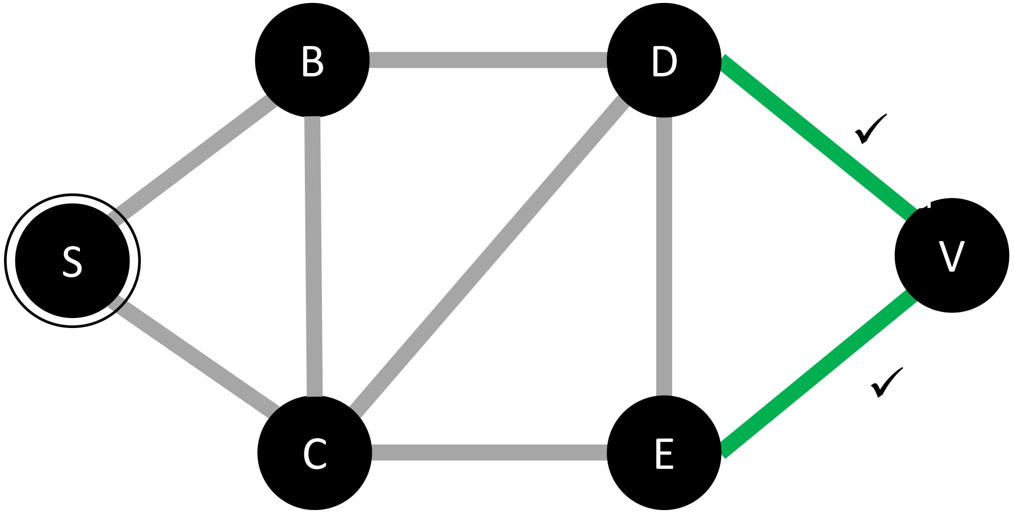
$Q = [\,V\,]$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	2	3

ESTADO FINAL

$Q = \phi$

	S,	B,	C,	D,	E,	V
d =	0	1	1	2	2	3

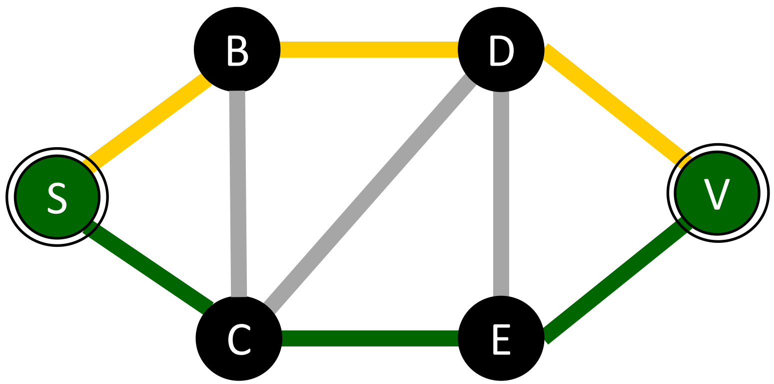


¿Cómo hallar el camino de *s* a *v*?

No ha sido procesado Está en la cola Salió de la cola

Algoritmo BFS Simple – Hallar camino

```
while Q ≠ ∅
  do u ← DEQUEUE(Q)
    for each v ∈ Adj[u]
      do if d[v] = ∞
       then
          d[v] ← d[u] + 1
          π[v] ← u
          ENQUEUE(Q, v)
```



S, B, C, D, E, V
d = [0, 1, 1, 2, 2, 3]
π = [-, S, S, B, C, D]

$L(s, v) = 3$:
1) → (S, B, D, V)
2) → (S, C, E, V)

Dos caminos diferentes y ambos de longitud mínima

Árbol que contiene a todos los vértices de G

- Se añade un ***array π*** que representará un ***árbol de cubrimiento o abarcador* (generado por el BFS)** con raíz ***s*** utilizando la técnica de ***referencia al padre***. En él, estarán todos los vértices alcanzables desde ***s***
- El **array *π*** se inicializa con todos sus valores = **null**
- Note que $\pi[s] = \text{null}$ siempre, ya que $d[s] = 0$

Algoritmo BFS - Propiedades

- Para todo vértice v alcanzable desde s, el camino en el **árbol abarcador** producido por el **BFS**, desde s hasta dicho vértice, es el **camino de longitud mínima** desde s hasta v en G

- El **árbol abarcador** que produce un recorrido **BFS no es único**

- El algoritmo **descubre todos los vértices** que se encuentran a una **longitud** ***k*** de s **antes** de descubrir algún vértice que está a una **longitud** ***k+1*** de dicho vértice

… Búsqueda primero a lo ancho …

Algoritmo BFS con Colores

En el ***BFS Simple*** para saber si un vértice ha sido visitado o no basta con comprobar que $\boldsymbol{d}[\boldsymbol{v}] \neq \infty$

Para mantener una traza de por dónde ha transitado el algoritmo, existe una variante del mismo donde se "**colorean**" los vértices utilizando tres colores:

- **blanco:** color de todos los vértices **al inicio**, excepto ***s: s.color = gris***
- **gris** : color al que pasan los vértices cuando son *descubiertos* (o sea, **la primera vez que son alcanzados por el algoritmo y se añaden a la Cola Q**)
- **negro**: un vértice pasa de ***gris*** a ***negro*** **cuando el algoritmo ya descubrió a todos sus adyacentes (el vértice sale de la Cola Q)**

Si ***(u, v)*** $\in$ ***E*** y el vértice ***u*** es **negro** $\Rightarrow$ el vértice ***v*** es **negro** o **gris**

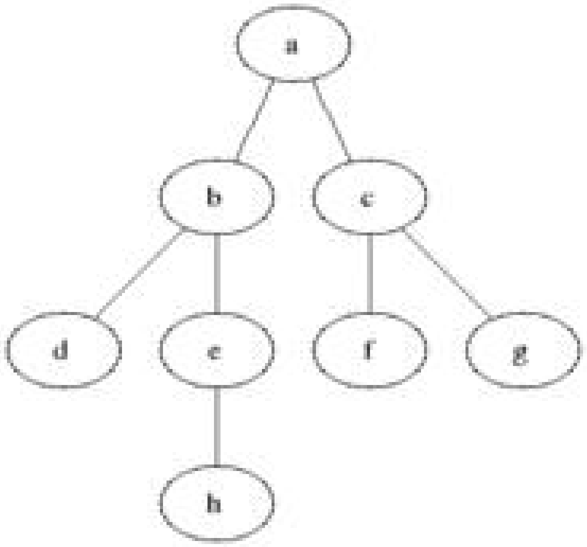
Algoritmo BFS con Colores

Analogía entre ***array d*** y ***colores***

 ⇔ ***v*** no está en la **Cola** y ***d[v] = ∞***

 ⇔ ***v*** está en la **Cola** y ***d[v] ≠∞***

 ⇔ ***v*** no está en la **Cola** y ***d[v] ≠ ∞***



d[] adquiere relevancia para saber con exactitud el ***tamaño*** (**cantidad de aristas**) del ***camino de longitud* mínima** entre el origen y cada vértice, una vez que concluye el ***BFS***

Algoritmo BFS con Colores

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

$\forall v \in V$; $v \neq$ origen, hacer:
color[v]=blanco, d[v]=∞; Π[v]=nil

color[origen]=gris; d[origen]=0; Π[origen]=nil

inicialmente, *Queue* vacía

insertar el *origen* en *Queue*

mientras que *Queue* $\neq \phi$:

- **eliminar un elemento de *Queue***
- **inspeccionar la *lista de adyacencia* del elemento eliminado:**
  - **para cada vértice adyacente al mismo, no descubierto (color blanco): colorearlo de gris, incrementar en 1 su distancia al origen, indicar que se llegó a *v* desde *u*, e insertarlo en Q**

- Después de inspeccionar toda su *lista de adyacencia* , colorearlo de negro

Algoritmo BFS – *Complejidad temporal*

- Tras las inicializaciones, ningún vértice volverá a colorearse de **blanco** $\Rightarrow$

la condición de la **línea 13** garantiza que:

> **cada vértice será almacenado en *Q*, a lo sumo, una sola vez y por tanto se extraerá de la misma una sola vez también**

- Las operaciones "***enqueue***" **y** "***dequeue***" son *O(1)*,

$\Rightarrow$ la complejidad temporal de las operaciones sobre la ***Queue*** son *O(V)*

O(*V*)

- La **lista de adyacencia** de cada vértice es analizada en el momento en que éste es sacado de ***Q*** y dicho análisis se hace **una sola vez**
- La suma del tamaño de todas las **listas de adyacencia** es O(*E*)

O(*E*)

- La sobrecarga de tiempo por las inicializaciones es O(*V*)

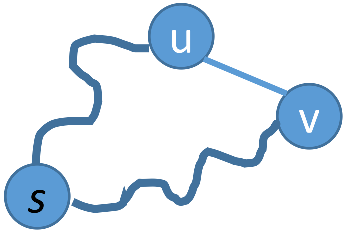
O(*V*)

Por tanto, el tiempo de ejecución del algoritmo ***BFS*** es **O(*V+E*)** para una representación del grafo *G=(V, E)* por **Lista de Adyacencia**

Sea ***G* = (*V*, *E*)** un grafo **no dirigido** y sea ***s*** **∈*V*** un vértice arbitrario, entonces, para toda arista ***(u, v)*∈*E*:**

Lema 1 → El valor del *camino de longitud mínima* entre *s* y *v* es, a lo sumo, 1 más que el valor del <u>*camino de longitud mínima*</u> entre *s* y *u*

$L(s, v) \le L(s, u) + 1$



Lema 2 → *"d[v]* acota superiormente a *L(s, v)*"

$d[v] \ge L(s, v)$

En esencia, este Lema demuestra que ***d[v]*** nunca será **menor** que ***L(s, v)*** y al final del algoritmo, si ***v*** se alcanza desde ***s***, entonces se cumple exactamente la igualdad. O sea, durante la ejecución del algoritmo, ***d[v]*** toma solo dos valores, *∞, al inicio, y si existe camino de **s** a **v**, entonces, al ser descubierto toma de inmediato el valor **L(s, v)** y ese valor se mantiene invariante mientras dure la ejecución del algoritmo BFS*

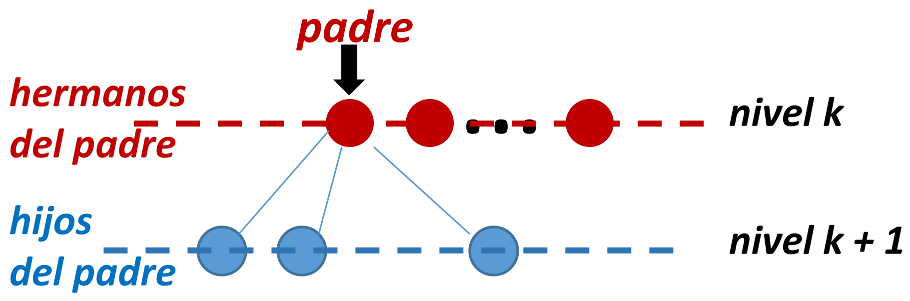
Lema 3 → **En *Q* hay, en cada momento, vértices con, a lo sumo, dos valores de *d* diferentes**

Supongamos que durante la ejecución del **BFS** , **Q** contiene los vértices:

$<v_1, v_2, v_3, \ldots, v_r>$

v_1: *frente* de **Q**

v_r : *fondo* de **Q**



entonces, $\boxed{d[v_r] \leq d[v_1] + 1}$ $\Rightarrow d[v_r] - d[v_1] \leq 1$ y además, se cumple $d[v_i] \leq d[v_{i+1}]$ $i=1, 2, \ldots, r-1$,

Justificación:

Un ***padre***, manda a sus ***hijos*** a ***Q***, por tanto, ***d[hijos] = d[padre] + 1*** y las ***d[hermanos]*** son iguales entre si

Corolario 4 → **Los valores de *d*, a medida que los vértices se van insertando en *Q*, crecen monótonamente en el decursar del tiempo**

Supongamos que los vértices $\boldsymbol{v_i}$ y $\boldsymbol{v_j}$ se insertan en **Q** durante la ejecución del **BFS** y que $\boldsymbol{v_i}$ se inserta antes que $\boldsymbol{v_j}$. Entonces $\boldsymbol{d[v_i] \leq d[v_j]}$ en el momento en que $\boldsymbol{v_j}$ es insertado

RESUMIENDO:

Sea $G = (V, E)$ un grafo **no dirigido** y sea $s \in V$ un vértice arbitrario, entonces, para toda arista $(u, v) \in E$:

Lema 1 → El valor del *camino de longitud mínima* entre *s* y *v* es, a lo sumo, 1 más que el valor del *camino de longitud mínima* entre *s* y *u*

Demostración en I.A (3rd. Edition) Lemma 22.1 pp 598

Lema 2 → *d[v]* acota superiormente a *L(s, v)*

Demostración en I.A (3rd. Edition) Lemma 22.2 pp 598

Lema 3 → En *Q* hay, en cada momento, vértices con, a lo sumo, dos valores de *d* diferentes

Demostración en I.A (3rd. Edition) Lemma 22.3 pp 599

Corolario 4 → Los valores de *d*, a medida que los vértices se van insertando en *Q*, crecen monótonamente en el decursar del tiempo

Demostración en I.A (3rd. Edition) Corollary 22.4 pp 599

Correctitud del BFS (Teorema 5)

El ***BFS*** calcula los ***caminos de longitud mínima*** entre el ***origen*** y los restantes vértices del grafo que son alcanzables desde él, o sea: $\forall v \in V$, $v \neq s$, entonces, si existe camino de s a v, ***BFS*** calcula $d[v]$: $d[v]=L(s, v)$

Demostración

Supongamos lo contrario:

Para, al menos, un ***vértice*** se cumple: ***d[vértice]≠L(s, vértice)***

De todos los que pueden cumplir lo anterior, **sea *v* el que tiene el mínimo valor de *L(s, v)*** (con ***v ≠ s***)

Tenemos entonces, ***d[v]≠L(s, v)*** y ***d[v] ≥ L(s, v)*** (por **Lema 2)**

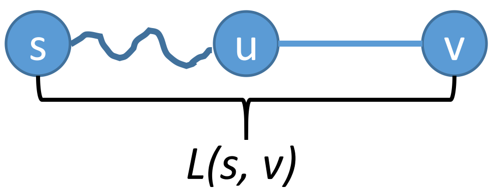
$\Rightarrow$ ***d[v] > L(s, v)*** **(1)** $\Rightarrow$ ***v*** es alcanzable desde ***s***

si no fuera así,
L(s, v) = ∞ ≥ d[v]
contradice **(1)**

Correctitud del BFS (Teorema 5)

Sea ***u*** el vértice que inmediatamente precede a ***v*** en el ***camino de longitud mínima*** de ***s*** a ***v***, por tanto,

$$L(s, v) = L(s, u) + 1 \quad \Rightarrow \quad \boxed{L(s, u) < L(s, v) \quad \Rightarrow \quad d[u] = L(s, u)}$$



Si se cumple esta desigualdad, no puede cumplirse ***d[u]≠L(s, u)*** ya que de todos los vértices que cumplen esto, ***L(s, v)*** es mínimo $\therefore$ ***d[u] = L(s, u)***

Poniendo todas estas propiedades juntas, tenemos:

$$\underbrace{d[v] > L(s, v)}_{(1)} = L(s, u) + 1 = d[u] + 1$$

$$\Rightarrow \boxed{d[v] > d[u] + 1} \quad \textbf{(2)}$$

Correctitud del BFS (Teorema 5)

<u>En el **momento en que el BFS decide extraer a *u* de Q**</u> **(l-11)**, ***v*** puede ser de cualquiera de estos colores: ***blanco***, ***gris*** o ***negro***

Veamos que, en cada caso se llega a una contradicción con la desigualdad (2)

$d[v] > d[u] + 1$

- Si ***v*** es **blanco**:
  En la (**l-15**) se hace $d[v] = d[u] + 1$ $\Rightarrow$ contradicción con (2)

- Si ***v*** es **negro**:
  Entonces ***v*** había sido extraído de **Q** con anterioridad y por el **Corolario 4**, se tiene $d[v] \leq d[u]$ $\Rightarrow$ contradicción con (2)

Correctitud del BFS (Teorema 5)

- Si ***v*** es **gris**:

  Entonces, un vértice ***w* ≠ *u***, adyacente también a ***v***, que se extrajo de la cola antes que ***u***, fue quien provoco que ***v*** se pintara de ***gris*** y que se insertara en la **Cola** ∴ $\boldsymbol{d[v] = d[w] + 1}$

  Además, ***w*** se extrajo de la misma antes que ***u***

  Por el **Corolario 4**, sin embargo, $\boldsymbol{d[w] \leq d[u]}$ y tendremos

  $\boldsymbol{d[v] = d[w] + 1 \leq d[u] + 1}$,

  $\Rightarrow \boldsymbol{d[v] \leq d[u] + 1} \Rightarrow$ contradicción con (2)

  $d[v] > d[u] + 1$

Finalmente, podemos concluir que tras acabar BFS: d[v] = L(s, v) para todo $\boldsymbol{v \in V}$

$\Rightarrow$ **el valor *d[v]* es igual a la *distancia mínima del origen* al vértice *v***

Correctitud del BFS (Teorema 5)

Para finalizar la prueba del **Teorema**, veamos cómo podemos obtener el ***camino de longitud mínima*** entre ***s*** y ***v***:

Observemos que:

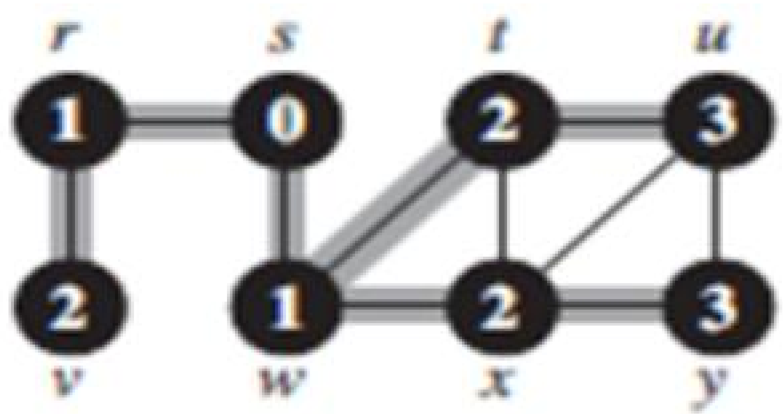
Si ***π[v] = u***, entonces ***d[v] = d[u] + 1***

Por tanto, podremos obtener el ***camino de longitud mínima*** para ir de ***s*** a ***v***:

transitando por un ***camino de longitud mínima,*** *de* ***s a π [v]*** *y posteriormente atravesar el arco* ***(π [v], v)***

Árbol de cubrimiento primero a lo ancho- BFS

El ***BFS***, a través de su recorrido por el grafo, forma un ***árbol de cubrimiento primero a lo ancho*** como se observa en la siguiente figura

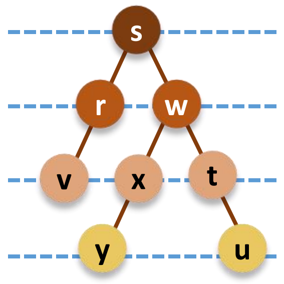


El ***árbol*** se forma a partir del campo ***π*** asociado a cada vértice de ***G***

árbol ***libre****: conexo y acíclico*

de recubrimiento*: en él están s y todos los vertices alcanzables desde s*

primero a lo ancho*: en el nivel k+1 solo hay vértices adyacentes a los vértices que están en el nivel k*   *k=0…Alt.-1*



Árbol de cubrimiento primero a lo ancho- BFS

- Inicialmente, el árbol solo contiene a su **raíz *(s: vértice origen)***

- Cada vez que se **descubre** un vértice ***v blanco, adyacente*** a algún vértice ***u*** que ya ha sido descubierto, entonces, ***v*** y el **arco (*u*, *v*)** se añaden al ***árbol***:

    - ***u*** es el ***predecesor***, o el ***padre***, de ***v*** en el ***árbol***

Cada vértice se descubre una sola vez, por tanto, tendrá, a lo sumo, **un solo *padre***

La relación ***ancestro–descendiente*** en el ***árbol*** se define tomando como referencia para ello a la raíz del árbol, o sea, ***s:***

$s \rightarrow \ldots \rightarrow u \rightarrow \ldots \rightarrow v$

ancestro de ***v*** ***descendiente*** de ***u***

u está, **en el árbol**, en el camino entre el vértice ***s*** y el vértice ***v***

Búsqueda "*Primero en profundidad*" (*Depth-First Search*: DFS)

Bibliografía: "Introduction to Algorithms". Second Edition.

The MIT Press. Massachusetts Institute of Technology. Cambridge, Massachusetts 02142.

http://mitpress.mit.edu

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

© Departamento de Programación
Facultad de Matemática y Computación
Universidad de La Habana

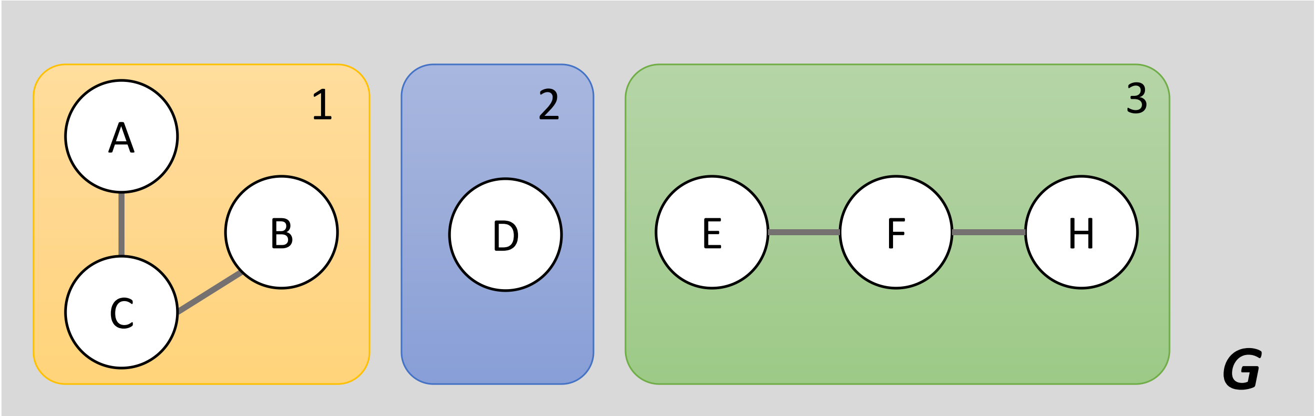
Componente conexa

Componente conexa de G:
Es un **subgrafo conexo MAXIMAL** de G

Se le llama **componente conexa** en el grafo no dirigido $G =< V, E >$ a un **conjunto maximal** de vértices $V_c \subset V$ tal que $\forall u, v \in V_c$ existe un camino entre u y v

PROBLEMA

Dado un grafo $G =< V, E >$no dirigido, determinar todas sus componentes conexas

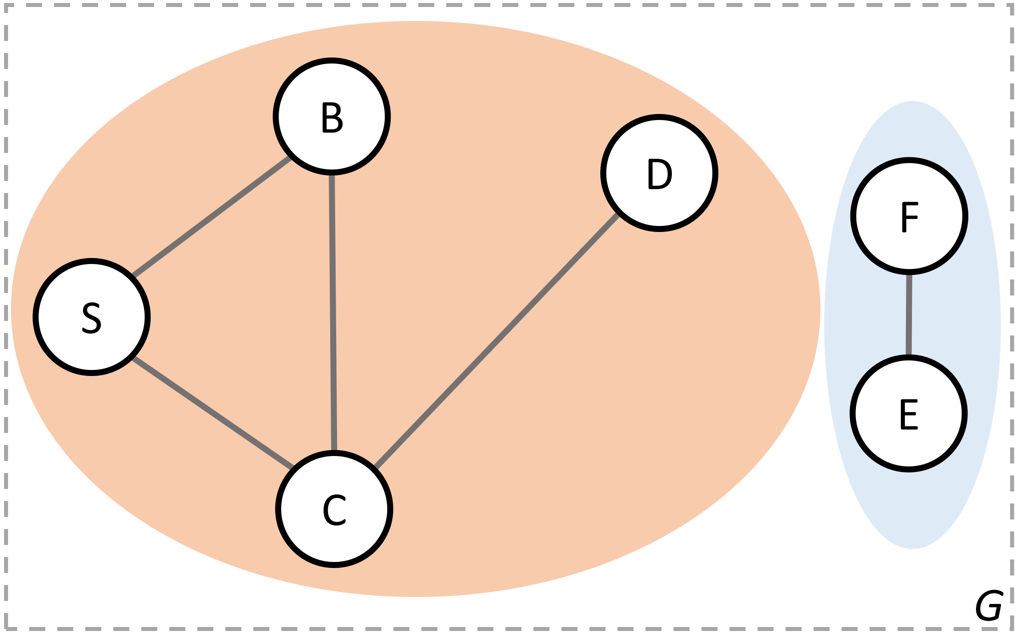


V={A, B, C, D, E, F, H }

E={ (A, C),
(C, B),
(E, F),
(F, H) }

Componente conexa

Los vértices E y F **no son alcanzables** desde los vértices S, B, C, D



NOTAS

En el grafo *G* existen dos (2) componentes conexas:

- *C1*= (S, B, C, D)
- *C2*= (E, F)

Algoritmo DFS simple

```
DFS(G)
   for each u ∈ V
      do if u not visited
            DFS-VISIT(G, u)
```

Garantiza visitar todos los vértices del grafo

```
DFS-VISIT(G, u)
   u ← visited
   ------------------------------------
   for each v ∈ Adj[u]
      do if v not visited
            DFS-VISIT(G, v)
   ------------------------------------
   return
   ------------------------------------
```

//se marca el vértice u como descubierto

//se analizan los adyacentes a u, que aun no han sido descubiertos

//se finaliza el análisis del vértice u (todos los vertices alcanzables desde u fueron descubiertos)

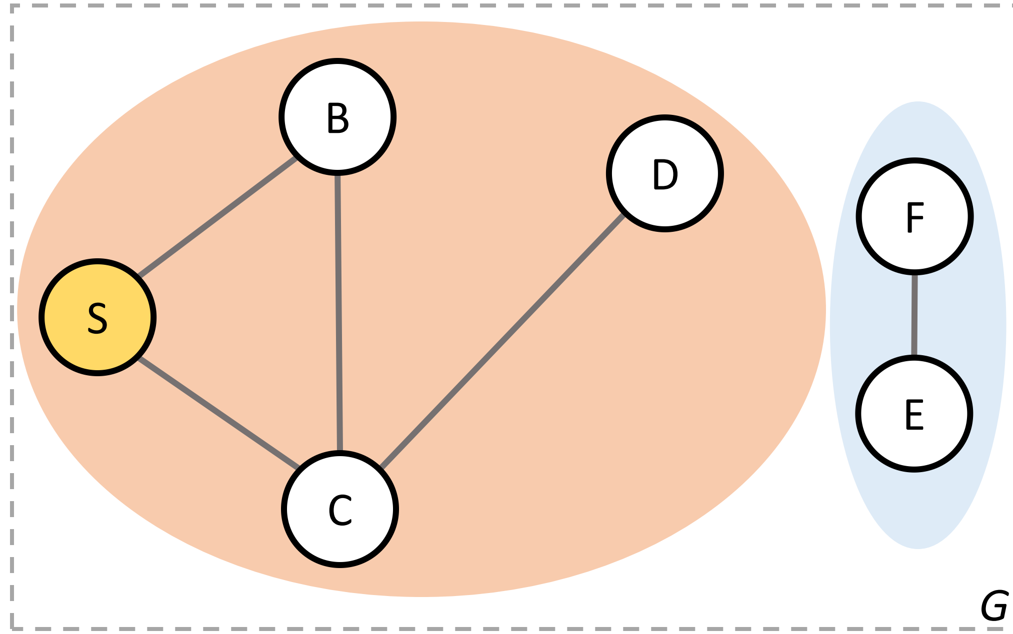
NOTAS

- DFS(G) visita todos los vértices del grafo
- DFS-VISIT(G,u), recursivamente, visita todos los vértices que son alcanzables desde u.
- Se lleva un *array booleano* **visited** para ir marcando los vértices visitados. **Inicialmente, se asumen todos como NO visitados**

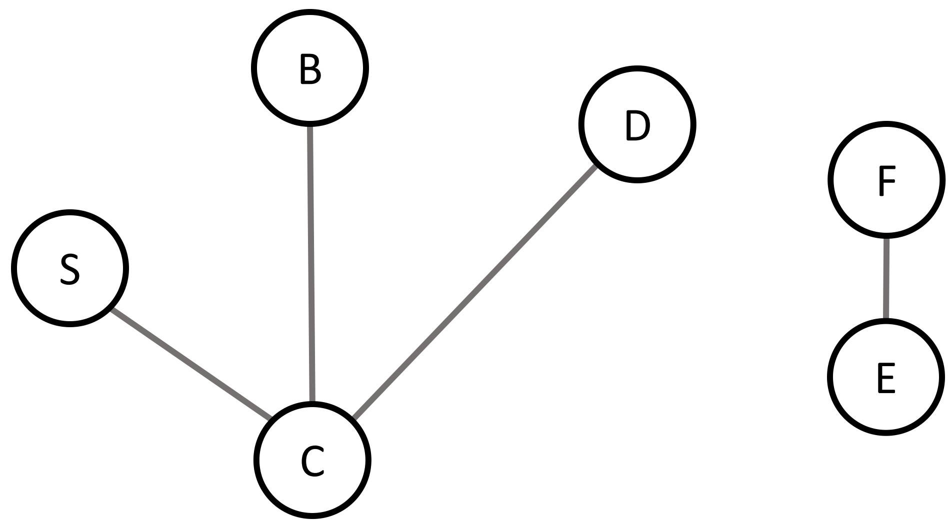
ALGUNAS MODIFICACIONES

- Array π [árbol de cubrim.]
- Cambiar ***visita*** por ***colores***
- Introducir ***tiempos*** de ***descubrimiento*** y ***finalización***

```
DFS(G)
   for each u ∈ V
      do if u not visited
           DFS-VISIT(G, u)

DFS-VISIT(G, u)
  u ← visited

  for each v ∈ Adj[u]
     do if v not visited
          DFS-VISIT(G, v)

  return
```



RESULTADO DEL DFS



Algoritmo DFS simple

```
DFS(G)
   for each u ∈ V
      do if u not visited
         π[u] ← null
         DFS-VISIT(G, u)
```

Garantiza visitar todos los vértices del grafo.

```
DFS-VISIT(G, u)
   u ← visited
```

// se marca el vértice u como descubierto

```
   for each v ∈ Adj[u]
      do if v not visited
         π[v] ← u
         DFS-VISIT(G, v)
```

// se analizan los adyacentes a u, que aun no han sido descubiertos

```
   return
```

// se finaliza el análisis del vértice u (todos los vértices alcanzables desde u fueron descubiertos)

NOTAS

- DFS(G) visita todos los vértices del grafo
- DFS-VISIT(G,u) recursivamente, visita todos los vértices que son alcanzables desde u.
- Se lleva un *array booleano* **visited** para ir marcando los vértices visitados. **Inicialmente, se asumen todos como NO visitados**

ALGUNAS MODIFICACIONES

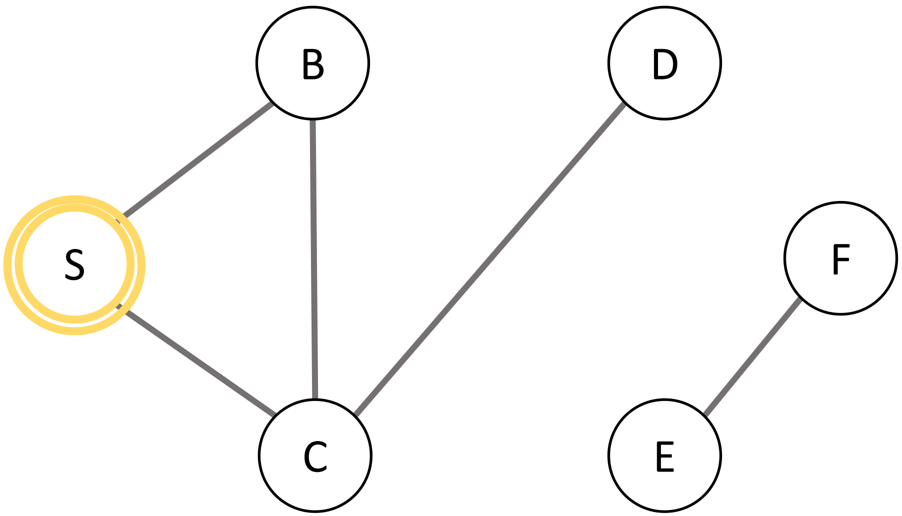
✓ Array π [árbol de cubrim.]
- Cambiar ***visita*** por ***colores***
- Introducir ***tiempos*** de ***descubrimiento*** y ***finalización***

Algoritmo DFS simple para Detección de CC

```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
       i++

DFS-VISIT(G, u, i)
  u ← visited

  for each v ∈ Adj[u]
    do if v not visited
       π[v] ← u
       DFS-VISIT(G, v)

  CC[u]=i
  return
```

DETECCIÓN COMP. CONEXAS

La presente modificación asigna el mismo valor entero a los vértices que pertenecen a una misma componente conexa



Algoritmo DFS simple – Detección CC

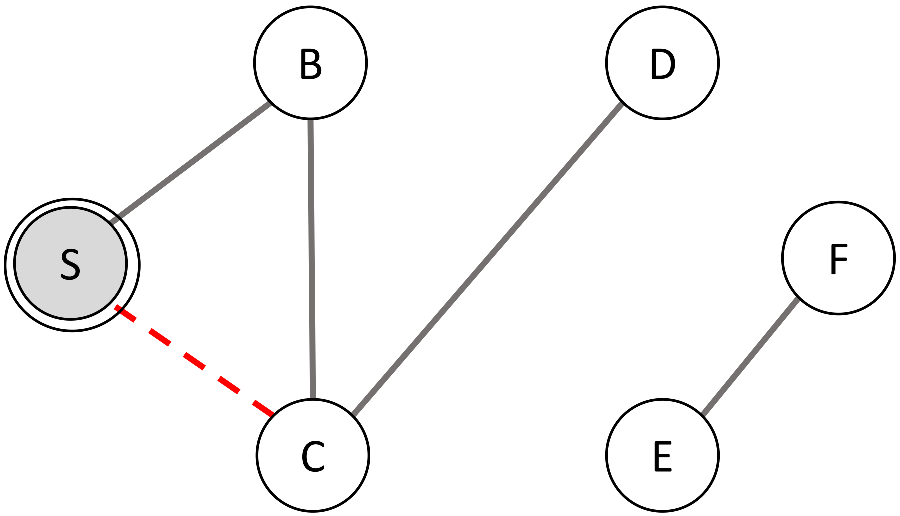
```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
      π[u] ← null
      DFS-VISIT(G, u , i)
      i++
```

```
DFS-VISIT(G, u, i)
  u ← visited
  - - - - - - - - - - - - - - - - - - - -
  for each v ∈ Adj[u]
    do if v not visited
      π[v] ← u
      DFS-VISIT(G, v)
  - - - - - - - - - - - - - - - - - - - -
  CC[u]=i
  return
  - - - - - - - - - - - - - - - - - - - -
```

ESTADO

Stack= [S]

	S,	B,	C,	D,	E,	F
π =	[-,	-,	-,	-,	-,	-]
VISIT=	[X,	-,	-,	-,	-,	-]
CC =	[-,	-,	-,	-,	-,	-]



Algoritmo DFS simple – Detección CC

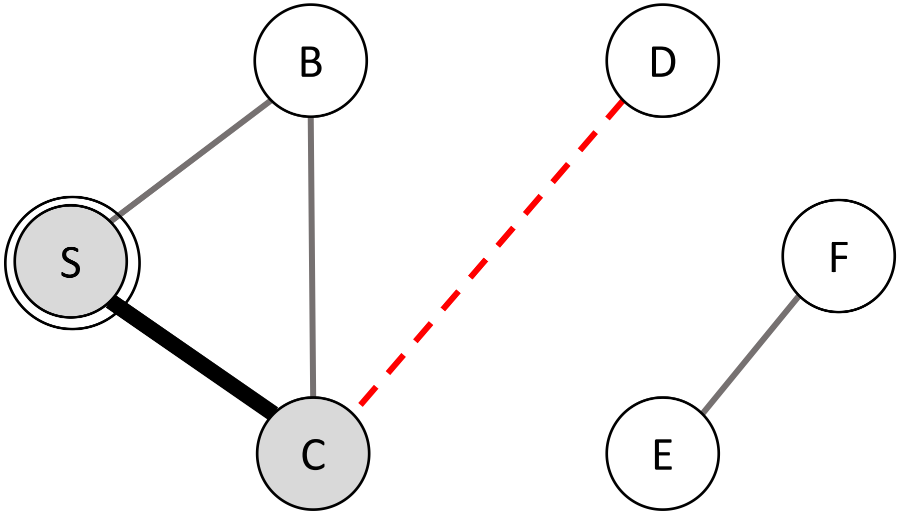
```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
      i++
```

```
DFS-VISIT(G, u, i)
  u ← visited
  - - - - - - - - - - - - - - - - - - - -
  for each v ∈ Adj[u]
    do if v not visited
       π[v] ← u
       DFS-VISIT(G, v)
  - - - - - - - - - - - - - - - - - - - -
  CC[u]=i
  return
  - - - - - - - - - - - - - - - - - - - -
```

ESTADO

Stack= [S,C]

	S,	B,	C,	D,	E,	F
π =	[-,	-,	S,	-,	-,	-]
VISIT=	[X,	-,	X,	-,	-,	-]
CC =	[-,	-,	-,	-,	-,	-]



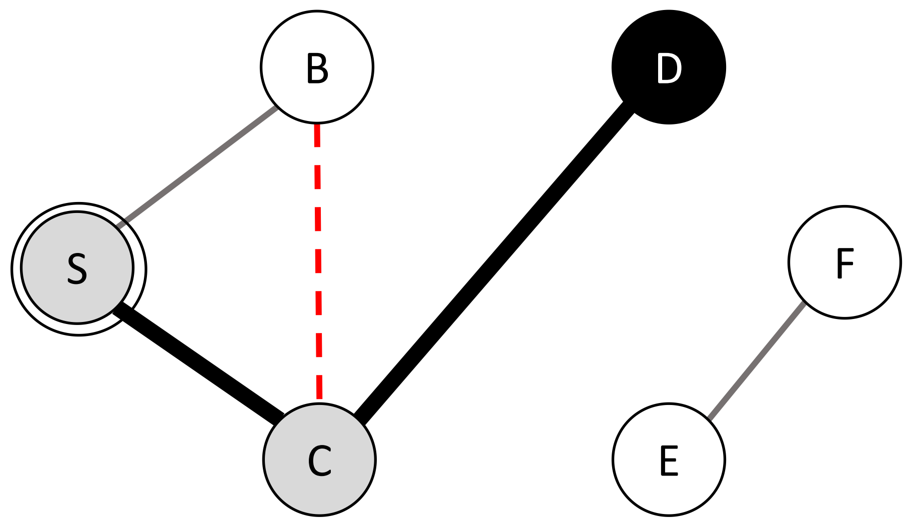
Algoritmo DFS simple – Detección CC

```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
      i++

DFS-VISIT(G, u, i)
  u ← visited
  ---------------------------------
  for each v ∈ Adj[u]
    do if u not visited
       π[v] ← u
       DFS-VISIT(G, v)
  ---------------------------------
  CC[u]=i
  return
  ---------------------------------
```

ESTADO

Stack= [S,C,D]

	S,	B,	C,	D,	E,	F
π =	[-,	-,	S,	C,	-,	-]
VISIT=	[X,	-,	X,	X,	-,	-]
CC =	[-,	-,	-,	1,	-,	-]



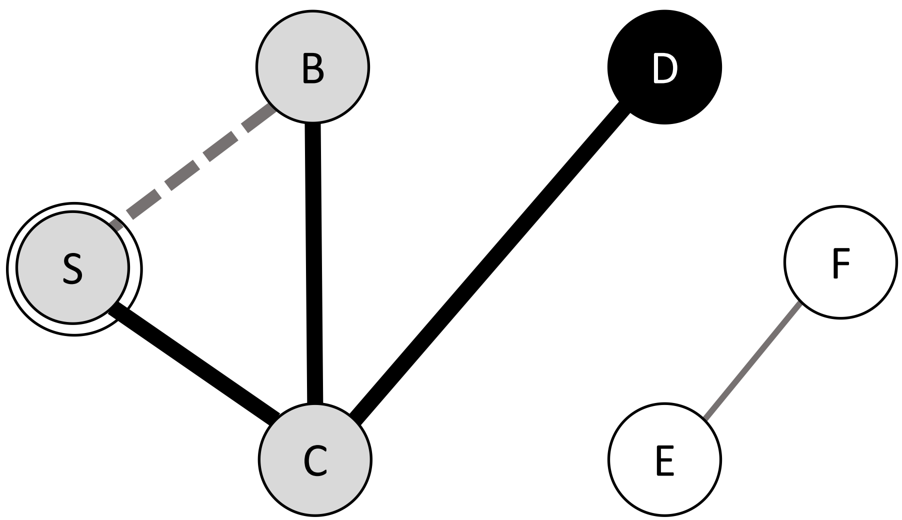
Algoritmo DFS simple – Detección CC

```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
      i++
DFS-VISIT(G, u, i)
  u ← visited
  - - - - - - - - - - - - - - - - - - - - -
  for each v ∈ Adj[u]
    do if v not visited
       π[v] ← u
       DFS-VISIT(G, v)
  - - - - - - - - - - - - - - - - - - - - -
  CC[u]=i
  return
  - - - - - - - - - - - - - - - - - - - - -
```

ESTADO

Stack= [S,C,B] Stack= [S,C] Stack= [S] Stack= []

	S,	B,	C,	D,	E,	F
π =	[-,	C,	S,	C,	-,	-]
VISIT=	[X,	X,	X,	X,	-,	-]
CC =	[-,	-,	-,	1,	-,	-]



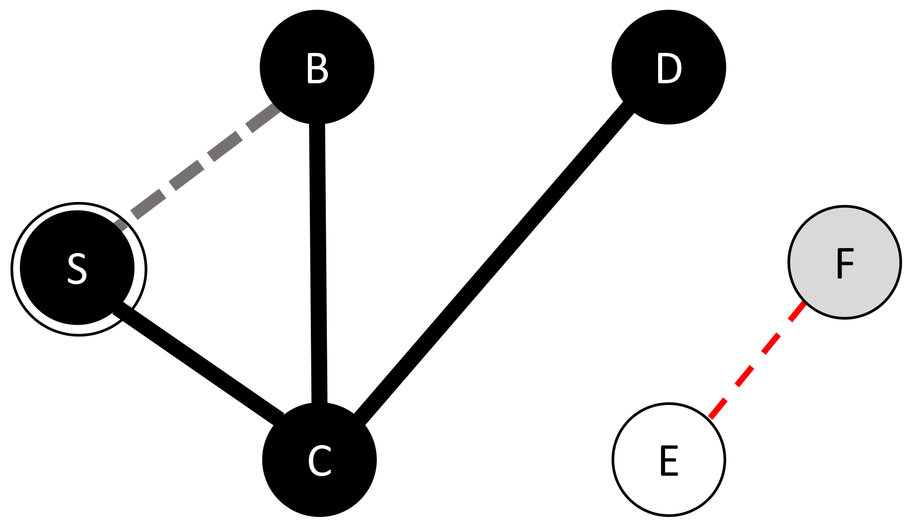
Algoritmo DFS simple – Detección CC

```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
       i++
DFS-VISIT(G, u, i)
  u ← visited
  ------------------------------
  for each v ∈ Adj[u]
    do if v not visited
       π[v] ← u
       DFS-VISIT(G, v)
  ------------------------------
  CC[u]=i
  return
  ------------------------------
```

ESTADO

Stack= [F]

	S,	B,	C,	D,	E,	F
π =	[-,	C,	S,	C,	-,	-]
VISIT=	[X,	X,	X,	X,	-,	X]
CC =	[1,	1,	1,	1,	-,	-]



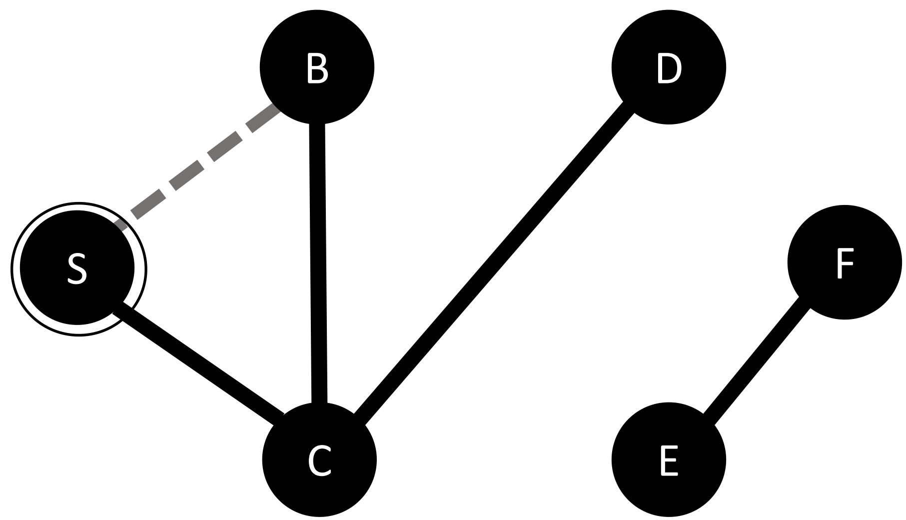
Algoritmo DFS simple – Detección CC

```
DFS(G)
  i=1
  for each u ∈ V
    do if u not visited
       π[u] ← null
       DFS-VISIT(G, u , i)
       i++

DFS-VISIT(G, u, i)
  u ← visited
  - - - - - - - - - - - - - - - -
  for each v ∈ Adj[u]
    do if v not visited
       π[v] ← u
       DFS-VISIT(G, v)
  - - - - - - - - - - - - - - - -
  CC[u]=i
  return
  - - - - - - - - - - - - - - - -
```

ESTADO

Stack= [F, E] Stack= [F] Stack= []

	S	B	C	D	E	F
π =	-	C	S	C	F	-
VISIT=	X	X	X	X	X	X
CC =	1	1	1	1	2	2



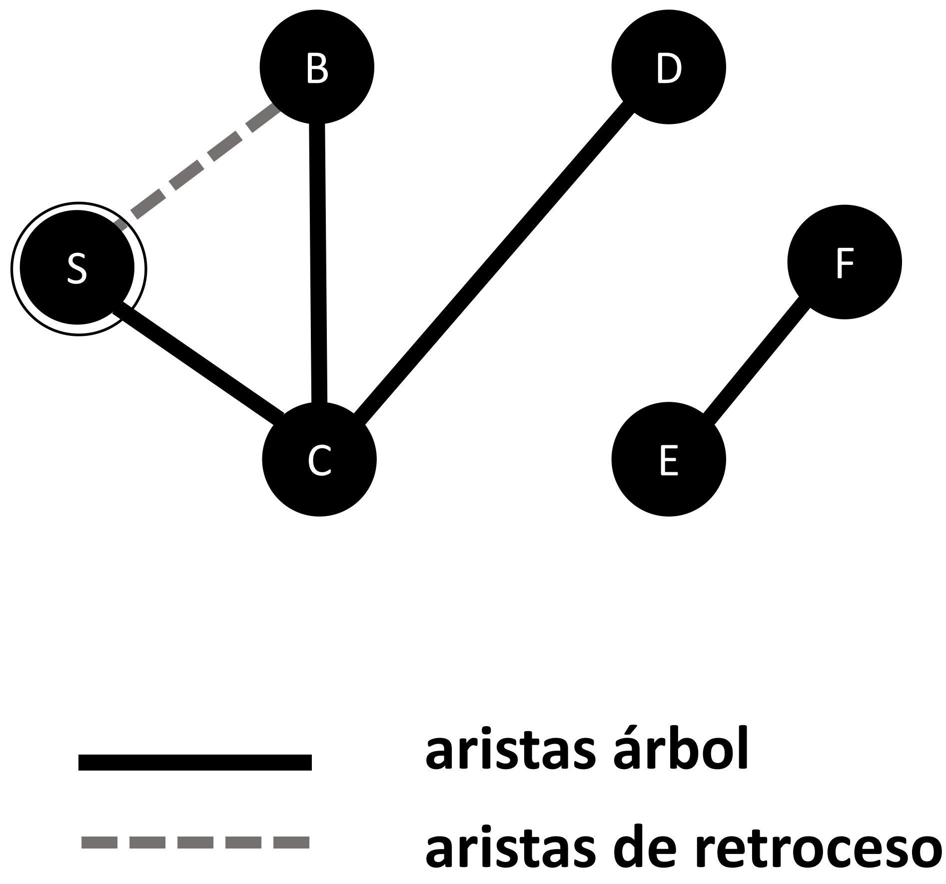
Algoritmo DFS - Propiedades

El ***array*** $\boldsymbol{\pi}$, en este caso, en vez de representar un solo ***árbol*** como en el BFS puede estar formado por ***varios árboles (bosque***) donde las respectivas *raíces* serán los valores ***null*** del ***array***

Observación: aunque conceptualmente, el **BFS** podría trabajar también desde múltiples orígenes, se utiliza típicamente para determinar **caminos de longitud mínima** a partir de un **vértice inicial**, por tanto, **recorre, puntualmente, los vértices de una** ***componente conexa del Grafo*** **o de un** ***Grafo Conexo*** **en general**

Se definen 2 tipos de aristas en términos del ***bosque abarcador en profundidad*** para el grafo **G no dirigido**:

- ***Aristas de árbol*** (*Tree edges*): Son las aristas de **G** que forman parte del bosque $\boldsymbol{\pi}$
- ***Aristas de retroceso*** (*Back edges*): Son aquellas aristas ***(u, v)*** de **G** que conectan un vértice ***u*** con un **ancestro** ***v*** en un **árbol abarcador** del bosque $\boldsymbol{\pi}$


B
D
S
F
C
E
aristas árbol
aristas de retroceso

DFS con colores

- Los vértices se van coloreando a medida que el proceso de búsqueda se va llevando a cabo. Esto permite identificar el estado en que se encuentra cada vértice en un momento dado del recorrido

- Al inicio del recorrido, todos los vértices son ***blancos***

- Cuando un vértice se alcanza en el recorrido, su color pasa de ***blanco*** a ***gris***

- Cuando todos los **vértices adyacentes,** de un determinado vértice, se analizan completamente entonces éste vértice se colorea de ***negro***

Esta técnica garantiza que, al final, cada vértice queda, exactamente, en un solo ***árbol libre abarcador en profundidad*** del ***bosque*** y obviamente, se garantiza que estos ***árboles*** sean disjuntos

DFS con tiempos

El **DFS** puede registrar momentos importantes para cada vértice, lo cual se hace **simulando una función de *tiempo* que se inicializa en 0 antes de comenzar el recorrido**

Para cada vértice ***v*, se registran dos "instantes de tiempo" relevantes**:

- *d[v]*: momento en que dicho vértice *se alcanza* o *se descubre* (momento en que el vértice se colorea a ***gris***) por el **DFS**

- *f[v]*: momento en que los adyacentes a dicho vértice se terminan de analizar (momento en que el vértice se colorea a ***negro***)

Estos valores se usan con frecuencia en varios algoritmos de grafos y son muy útiles a la hora de razonar en relación a cómo se comporta la ***búsqueda en profundidad*** en los mismos

DFS con tiempos

Para todo vértice ***u*** se cumplirá

d[u]<***f[u]***

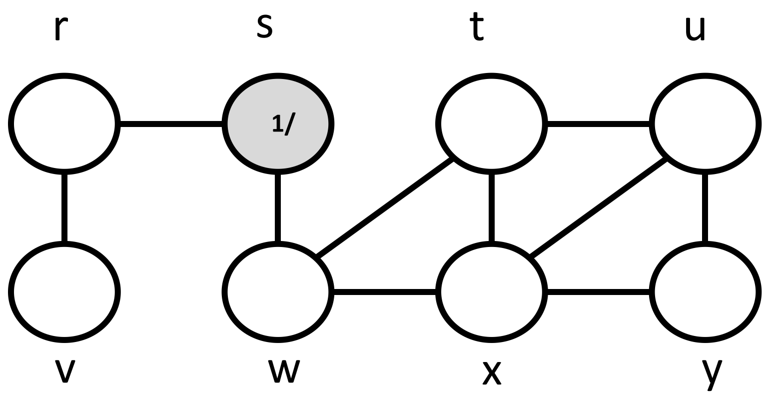
El vértice ***u:***

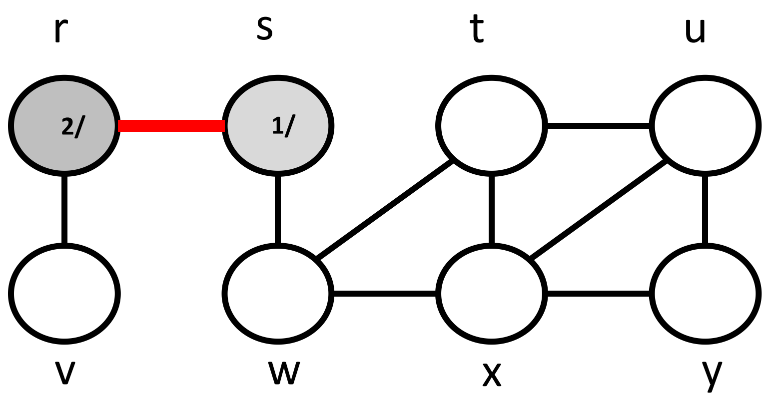
- será **blanco** antes del *instante de tiempo* ***d[u]***
- será **gris** entre el *instante de tiempo* ***d[u]*** y ***f[u]***
- será **negro** en lo sucesivo, o sea, a partir del *instante de tiempo* ***f[u]***

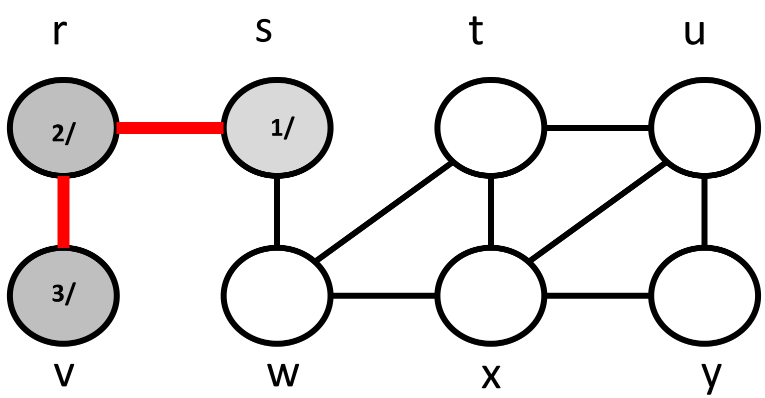
Para este algoritmo, en particular, el grafo de entrada *G* puede ser *dirigido* o *no-dirigido*. La variable *time* es global y sobre ella se simula el transcurso del *tiempo*

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

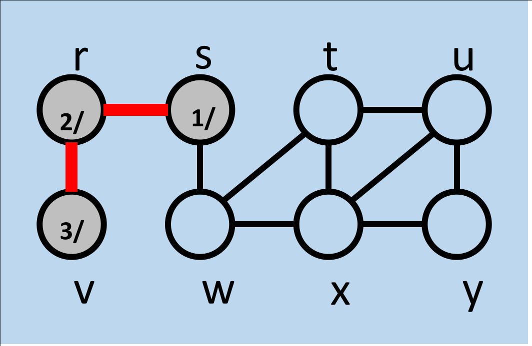
```
DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

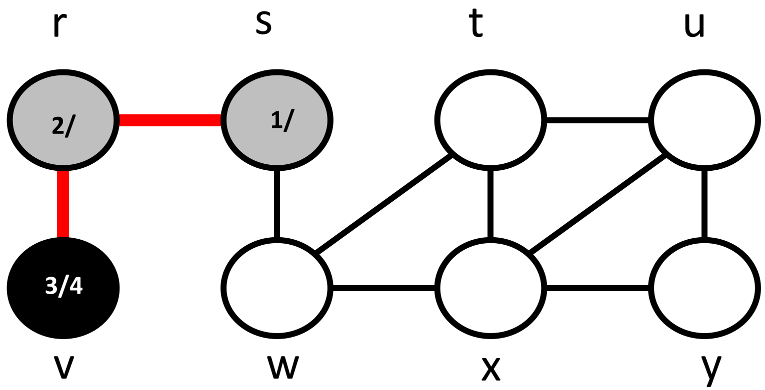


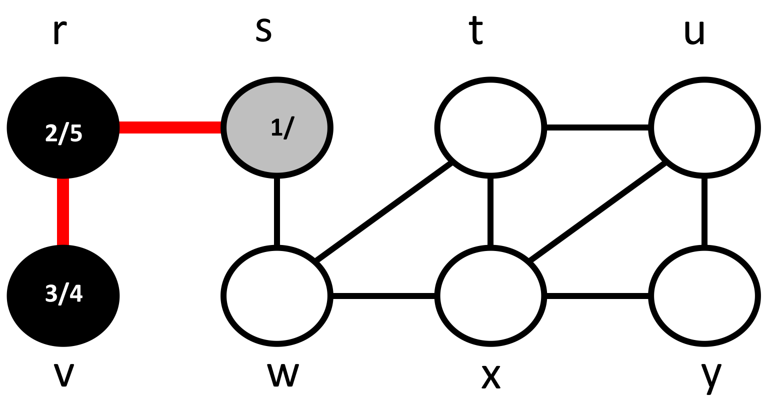


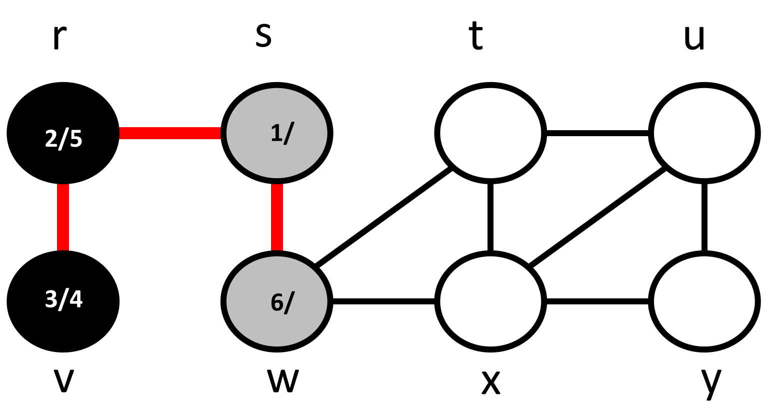


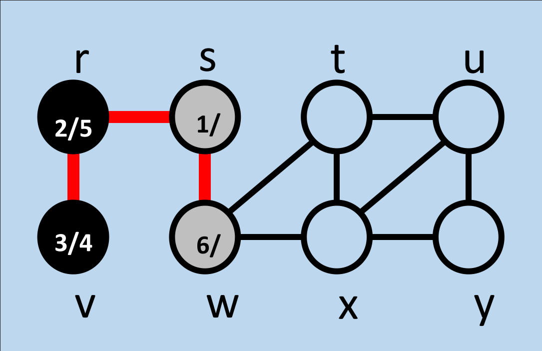
Tiempos: **d/f**

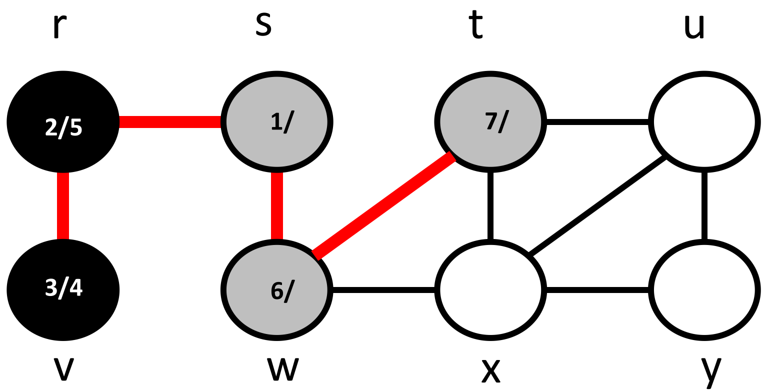

r
s
t
u
2/
1/
3/
v
w
x
y

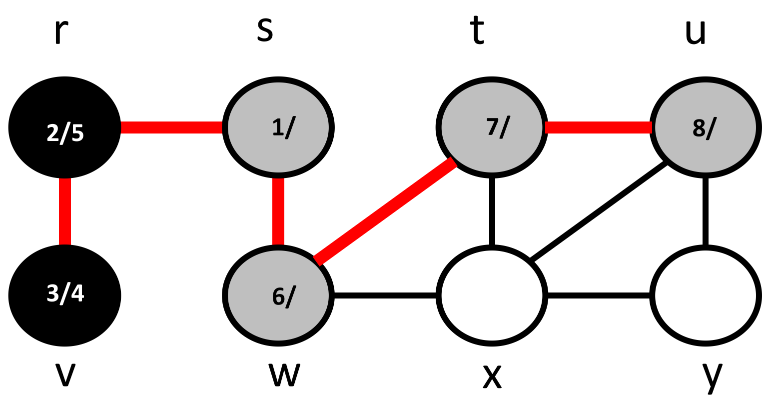

r
s
t
u
2/
1/
3/4
v
w
x
y

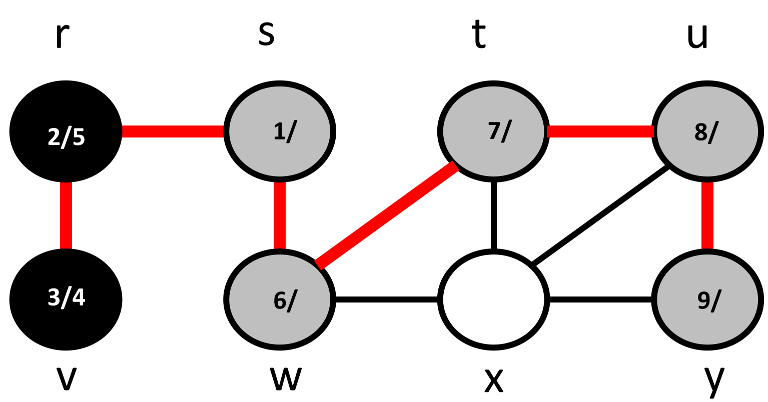

r
s
t
u
2/5
1/
3/4
v
w
x
y

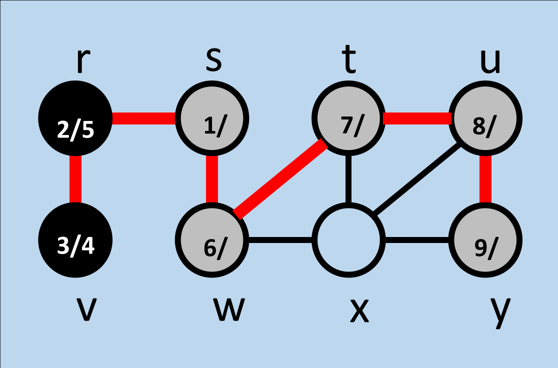

r
s
t
u
2/5
1/
3/4
6/
v
w
x
y

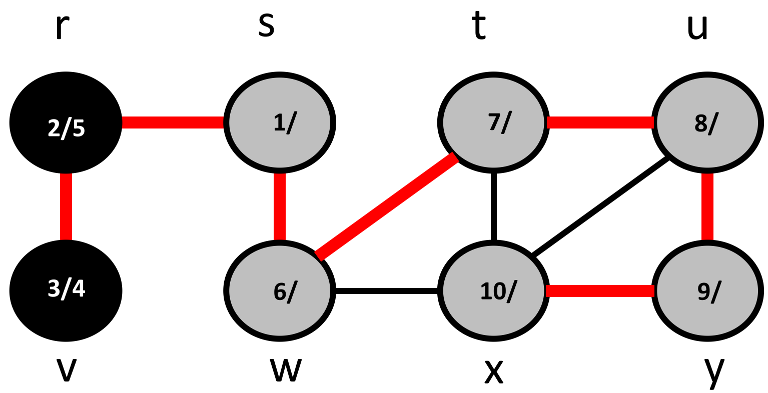

r s t u
2/5 1/
3/4 6/
v w x y

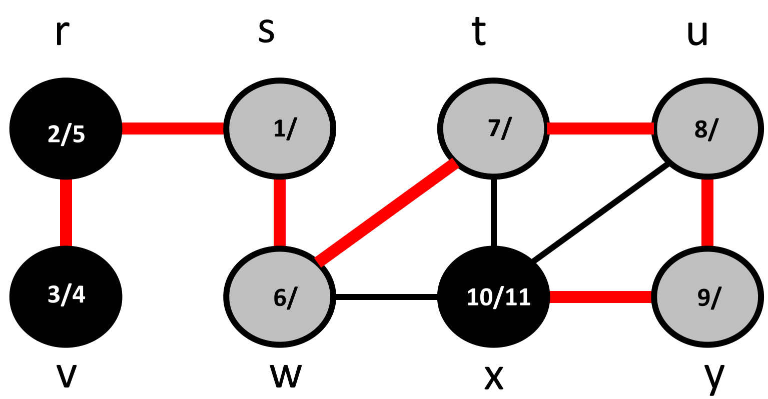

r s t u
2/5 1/ 7/
3/4 6/
v w x y

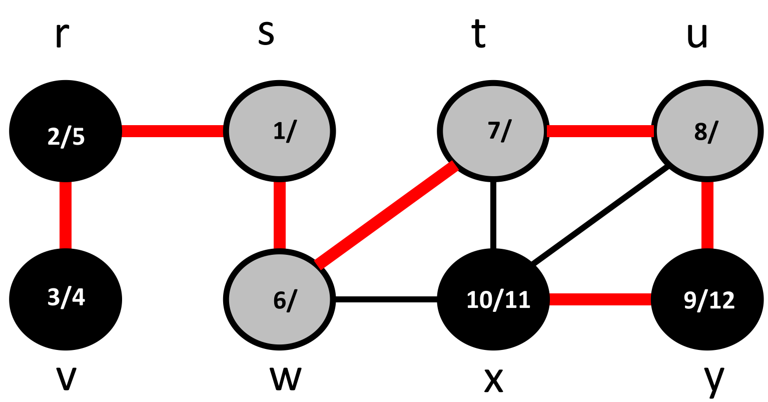

r s t u
2/5 1/ 7/ 8/
3/4 6/
v w x y

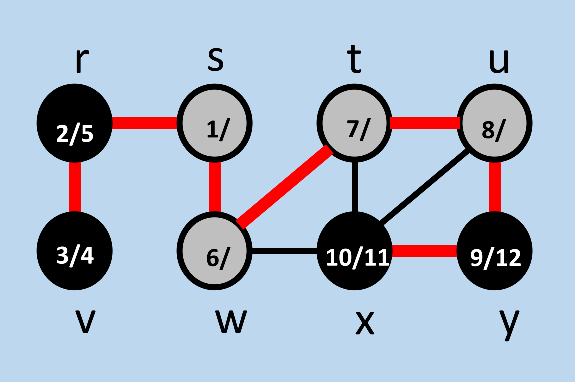

r s t u
2/5 1/ 7/ 8/
3/4 6/ 9/
v w x y

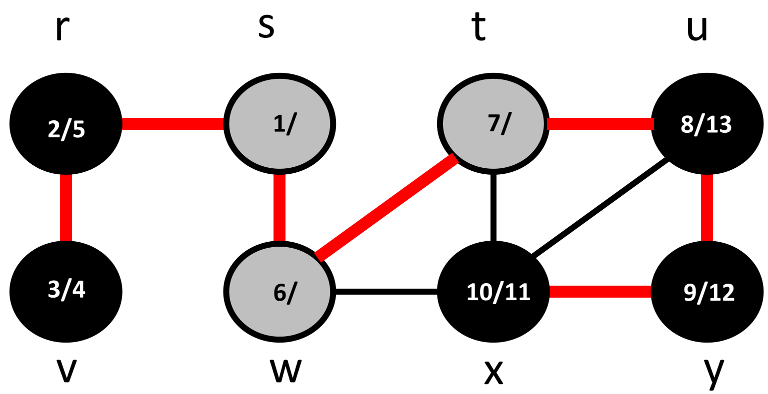

r
s
t
u
2/5
1/
7/
8/
3/4
6/
9/
v
w
x
y

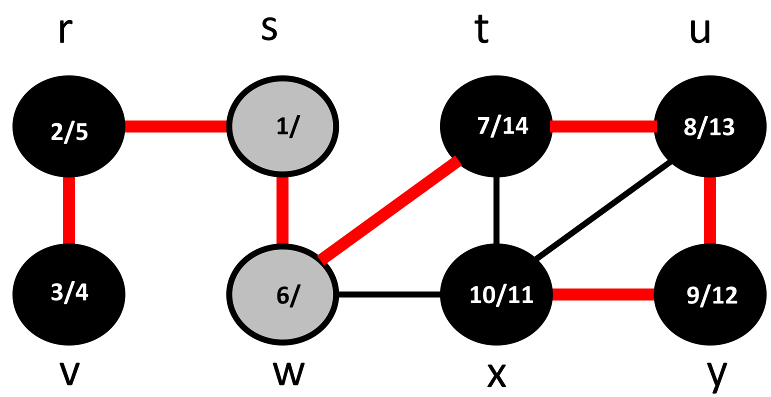

r
s
t
u
2/5
1/
7/
8/
3/4
6/
10/
9/
v
w
x
y

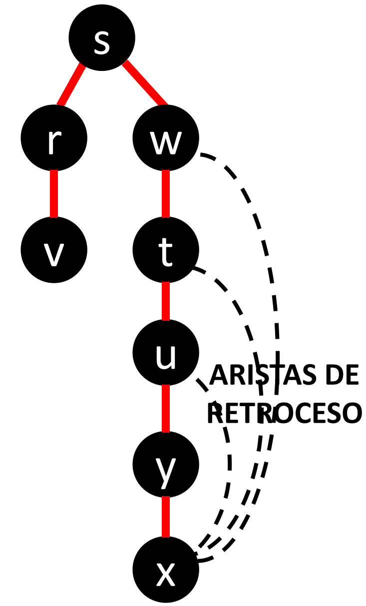

r
s
t
u
2/5
1/
7/
8/
3/4
6/
10/11
9/
v
w
x
y


r
s
t
u
2/5
1/
7/
8/
3/4
6/
10/11
9/12
v
w
x
y

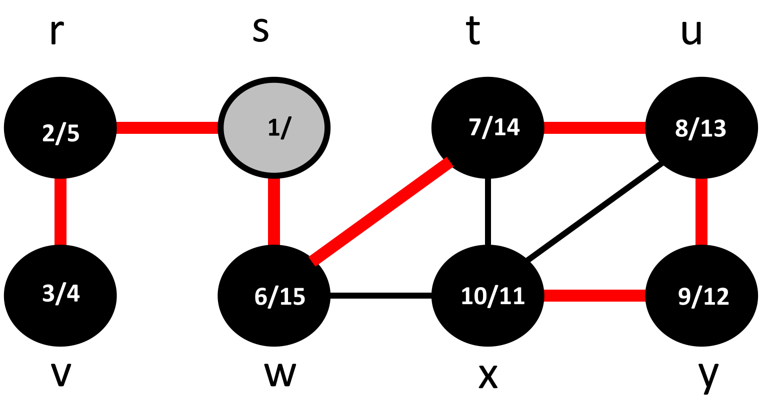


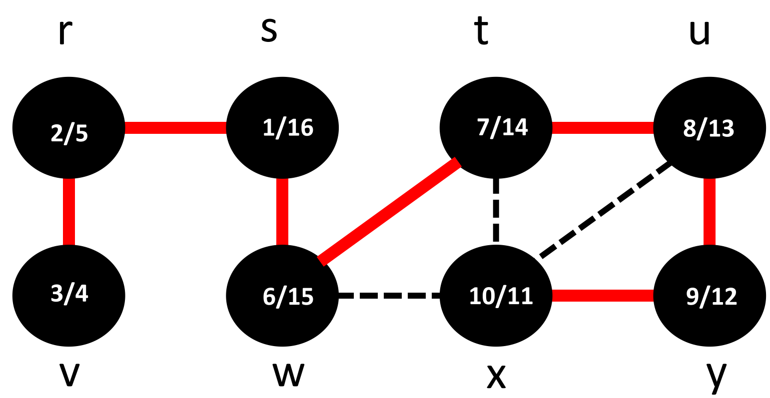






Árbol abarcador





DFS – Complejidad temporal

•Los ciclos que aparecen en las líneas **1-3** y **5-7** son *Θ(V)* excluyendo el tiempo requerido para ejecutar los llamados a **DFS-VISIT**

•El **DFS-VISIT** es llamado exactamente una vez para cada vértice ***v*** del grafo debido a que el mismo se invoca solo sobre los vértices **blancos** y ellos, en cuanto son descubiertos, se colorean a **gris**

•Durante la ejecución de **DFS-VISIT*(v)*** el ciclo de las líneas **4-7** es ejecutado |***Adj[v]***| veces, por tanto, el costo total de ejecutar dichas líneas es $\Theta(E)$ y en correspondencia,

$\Downarrow$

La complejidad temporal de *DFS* es:

$\boldsymbol{\Theta(V + E)} \rightarrow$ G representado en una **Lista de Adyacencia**.
$\boldsymbol{\Theta(|V|^2)} \rightarrow$ G representado en una **Matriz de Adyacencia**.