Dimensionado y Planificación de Redes
Tema 1 - Conceptos básicos de teoría de grafos

Ramón Agüero Calvo
ramon.agueroc@unican.es

Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Motivación

- Análisis de *Redes*
  - Concatenación de nodos y enlaces de comunicaciones
- Para llevar a cabo el análisis se utiliza la teoría de grafos
  - Tiene amplias aplicaciones
  - Es una rama matemática por sí misma
- Su aplicación principal es en las redes de conmutación de paquetes
  - Algoritmos/Protocolos de encaminamiento
  - Máximo flujo (Multi-path/Network coding)
  - Spanning Tree (Bridges)

Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Complejidad de algoritmos (1/7)

- Para establecer “la calidad” de cualquier algoritmo se “estiman/cuentan” los ciclos/pasos necesarios para completar la ejecución
  - Depende fuertemente de los datos concretos que se utilicen en cada caso
- Modelo RAM (Random Access Machine)
  - No hay operaciones concurrentes
  - Cada instrucción toma un tiempo constante
  - Simplificaciones: operaciones complejas (p.ej. Exponenciales), accesos a memoria
- Se podría hablar de complejidad del peor caso, mejor caso y caso promedio
  - El más empleado es el peor caso

Complejidad de algoritmos (2/7)

- Las funciones matemáticas que se podrían emplear son “poco predecibles” y con mucho detalle
- Notación Big-Oh
  - Reduce los detalles para “cuantificar” la complejidad de los algoritmos
- Definiciones
  - Cota superior: $f(n) = O(g(n))$. Existe una constante $c>0$, tal que $c \cdot g(n) \geq f(n)$
  - Cota inferior: $f(n) = \Omega(g(n))$. Existe una constante $c>0$, tal que $c \cdot g(n) \leq f(n)$
  - Ambas cotas: $f(n) = \Theta(g(n))$. Existen dos constantes c_1, c_2, tal que $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- Propiedades de la notación Big-Oh
  - $O(f(n)) + O(g(n)) \rightarrow O(\max(f(n),g(n))$
  - $O(c \cdot f(n)) \rightarrow O(f(n))$
  - $O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$

Complejidad de algoritmos (3/7)

- Tasas de crecimiento funciones típicas (n en nanosegundos)

n	$\log_2 n$	n	$n\log_2 n$	n^2	2^n	n!
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.15 años
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 s	8.4E+15 a.
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 días	
100	0.007 μs	0.1 μs	0.664 μs	10 μs	4E+13 años	
1000	0.010 μs	1 μs	9.966 μs	1 ms		
10000	0.013 μs	10 μs	133 μs	100 ms		
100000	0.017 μs	0.1 ms	1.66 ms	10 s		
1000000	0.020 μs	1 ms	19.93 ms	16.7 m		
10000000	0.023 μs	0.01 s	0.23 s	1.2 días		
100000000	0.027 μs	0.1 s	2.66 s	115.7 días		
1000000000	0.030 μs	1 s	29.90 s	31.7 años		

Complejidad de algoritmos (4/7)

- Funciones empleadas en el análisis de complejidad de algoritmos
  - Funciones constantes: operaciones aritméticas; funciones sencillas – no dependen de n
  - Funciones logarítmicas: f(n) = $\log_2(n)$. Búsqueda binaria
  - Funciones lineales: f(n) = n. Búsqueda de máximo o mínimo. Cálculo de la media. Analizan los elementos de un array de n componentes una (o más) vez (veces)
  - Funciones súperlineales: f(n) = $n \cdot \log_2(n)$. Quicksort.
  - Funciones cuadráticas: f(n) = n^2. Analizan todas las parejas posibles en un universo de n elementos. Inserción o selección ordenada.
  - Funciones cúbicas: f(n) = n^3. Enumeran todos los triples posibles en un universo de n elementos. Multiplicación matrices.
  - Funciones exponenciales: f(n) = c^n. Enumeración de subconjuntos de n elementos
  - Funciones factoriales: f(n) = n!. Permutaciones. Ordenación de n elementos
- Relaciones de dominancia

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Complejidad de algoritmos (5/7)

- Problemas “sencillos”
  - Se pueden desarrollar algoritmos que resuelvan cualquier instanciación del problema en tiempo polinomial: algoritmo *eficiente*
  - Se les denomina Problemas P
- Existe otro grupo de problemas para los que *aún* no se han desarrollado ningún algoritmo eficiente
  - ¿Son realmente complejos? ¿Nunca se desarrollará un algoritmo eficiente para resolverlos?
  - Problemas NP y problemas NP-completos
- Problemas NP
  - Aquellos que son “verificables” en tiempo polinomial; dada una solución, se puede “comprobar” que satisface las condiciones del problema
  - Los Problemas P pertenecen a los NP

Complejidad de algoritmos (6/7)

- Problemas NP Completos
  - Caso particular de los NP
  - No se ha encontrado un algoritmo eficiente para resolverlos
  - No se ha demostrado que dicho algoritmo no existe realmente
  - Un problema es NP-completo si es tan *complicado* como cualquier problema NP
    - Si un problema NP-completo se pudiera resolver en tiempo polinomial, entonces *todos* los problemas NP también tendrían solución en dicho tiempo
  - Si un problema es NP-completo (hay que demostrarlo) sería sensato buscar una aproximación diferente al mismo o resolver un caso específico
  - Muchos problemas en grafos son NP-completos

Complejidad de algoritmos (7/7)

- ¿Cómo se demuestra que un problema P_2 es NP completo?
  - P_2 es NP
  - El resto de problemas NP se transforman en tiempo polinomial a P_2
    - En la práctica es suficiente con comprobarlo con un problema P_1 que se sepa NP-completo: hay varios ejemplos que se emplean habitualmente
- Reducción/transformación de problemas
  - P_1 se reduce de manera polinomial a P_2 si algún algoritmo polinomial que resuelve P_1 usa como subrutina un algoritmo para resolver P_2 sin coste adicional
    - Si P_1 se reduce de manera polinomial a P_2 y existe un algoritmo polinomial para resolver P_2, entonces algún algoritmo polinomial resuelve P_1
  - P_1 se transforma de manera polinomial a P_2 si para cada instancia I_1 de P_1 existe una instancia I_2 de P_2 tal que I_1 es una instancia YES de P_1 si y sólo si I_2 es una instancia YES de P_2
    - Si P_1 se puede transformar de manera polinomial a P_2 entonces P_2 es tan complicado como P_1 (el inverso no es cierto)

Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Definiciones previas

- Un grafo G(N,E) se define sobre un conjunto de |N| vértices/nodos y tiene un conjunto de |E| enlaces, parejas ordenadas o no de nodos de N
  - En algunas ocasiones puede resultar interesante/necesario asignar atributos c(u,v) a los enlaces: costes, capacidades, etc
- Un grafo G'(N',E') es subgrafo de G(N,E) si N' $\subseteq$ N y E' $\subseteq$ E
  - Si N' = N y E' $\subseteq$ E, G' es un *spanning subgraph* (subgrafo cubierto) de G
- Concatenaciones de enlaces
  - Walk (paseo): secuencia de nodos (n_1,n_2,…n_l) tal que cada pareja (n_{i-1},n_i) es un enlace del grafo
  - Path (camino): es un walk en el que no hay nodos repetidos
    - Un camino se puede "registrar" manteniendo el predecesor de cada nodo
    - Dos nodos (u,v) están conectados si existe al menos un camino entre ellos
  - Cycle (ciclo o bucle): camino con más de un enlace y en el que $n_1 = n_l$

Clasificación

- Dirigidos frente a no dirigidos
  - Dirigidos: (u,v) ≠ (v,u): Los enlaces son pares dirigidos de nodos
  - No dirigidos: el enlace (u,v) implica que exista el enlace (v,u)
    - No es necesario establecer un criterio de ordenación a los nodos en cada enlace
- Conectados frente a no conectados
  - Se dice que un grafo es (fuertemente) conectado si cualquier par de nodos está conectado por un camino
- Densos frente a poco densos (sparse)
  - Califica los grafos en función del número de enlaces que tienen
    - Densos: |E| cuadrático frente a |N|
    - Sparse: |E| lineal frente a |N|
- Cíclicos frente a no-cíclicos (acíclicos)
  - Los grafos a-cíclicos no tienen ciclos

Definiciones adicionales (1/2)

- Grado de un nodo (grafo dirigido)
  - Grado entrante (δ_i^+): número de enlaces que entran a un nodo
  - Grado saliente (δ_i^-): número de enlaces que salen de un nodo
  - Se cumple que…

$$|E| = \sum_{\forall i \in N} \delta_i^+ = \sum_{\forall i \in N} \delta_i^-$$

- Grado de un nodo (grafo no dirigido)
  - Número de vecinos (nodos adyacentes)
  - Se cumple que…

$$|E| = \frac{\sum_{\forall i \in N} \delta_i}{2}$$

Definiciones adicionales (2/2)

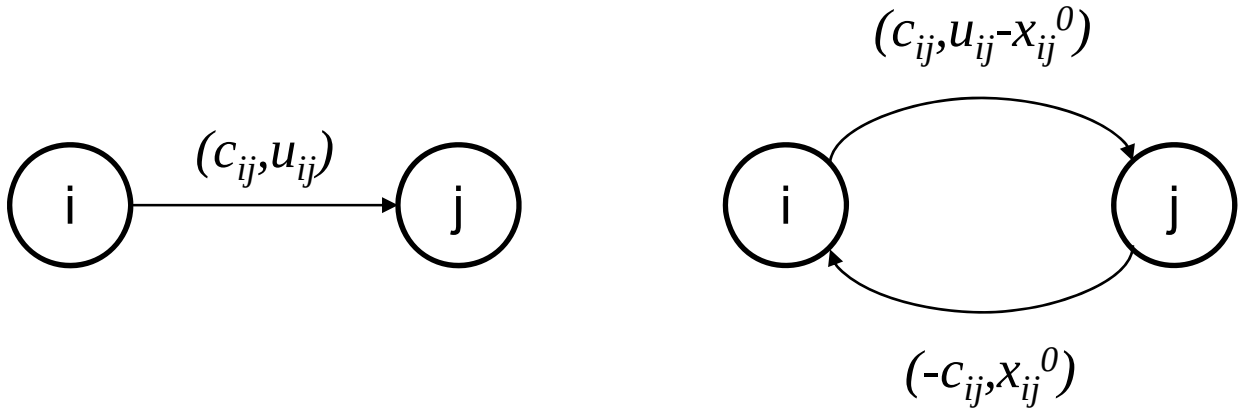
- Corte
  - Un corte "parte" (divide) el conjunto de nodos N en dos partes S y Ṡ = N – S
  - Se define por un conjunto de enlaces con un extremo en S y el otro en Ṡ
  - Un corte s-t cumple la propiedad que s $\in$ S y t $\in$ Ṡ
- Árbol
  - Grafo conectado sin ciclos
  - Propiedades
    - Un árbol de N nodos tiene N-1 enlaces
    - Cada par de nodos de un árbol está unido por un único camino
- Spanning Tree (Árbol cubierto)
  - T es un *spanning tree* de G si T es un *spanning subgraph* de G

Transformaciones de red (1/2)

- Enlaces no dirigidos a enlaces dirigidos
  - E{i,j} (no dirigido) se transforma en E(i,j) y E(j,i) (dirigidos)
- Eliminar cotas inferiores (mayores de 0) de flujos
- Revertir enlaces
  - Para eliminar enlaces con costes negativos
- Eliminar capacidades de enlaces
  - Transformar redes con capacidad a redes sin capacidad
  - Se añaden nodos adicionales
- Dividir nodos
- Remplazar costes con costes reducidos
  - Se “imputan” los costes asociados a los nodos extremos de un enlace (potencial)

Transformaciones de red (2/2)

- Redes residuales
  - En lugar de trabajar con flujos absolutos se emplean flujos incrementales en torno a una solución factible
  - Se supone que hay un flujo x_{ij}^0 en el enlace (i,j)
  - Se remplaza cada enlace (i,j) con dos: (i,j) y (j,i)
    - El enlace (i,j) tiene un coste c_{ij} y una capacidad residual $r_{ij} = u_{ij} – x_{ij}^0$
    - El enlace (j,i) tiene un coste $–c_{ij}$ y una capacidad residual $r_{ji} = x_{ij}^0$



Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Lista de adyacencia

- Consta de un array de N listas (una por nodo de la red), con punteros a cada nodo con el que tenga un enlace
- Memoria necesaria
  - En un grafo dirigido el número de punteros necesarios coincide con |E|
  - En un grafo no dirigido será 2 |E|
- Implementación como lista enlazada
- Ventajas
  - Se pueden asignar costes a los enlaces de manera sencilla
  - Requiere una cantidad menor de memoria, apropiada para grafos sin muchos enlaces (sparse)
- Desventajas
  - El proceso de búsqueda puede ser lento (se pueden diseñar algoritmos que eviten llevar a cabo esas búsquedas)

Matriz de adyacencia

- Matriz A de dimensión N x N
  - a(i,j) = 1 si (i,j) es un enlace de G
  - a(i,j) = 0 en caso contrario
- Con grafos no dirigidos, A es simétrica: $A^T = A$
- El tamaño de A es, para cualquier red, N^2
- Ventajas
  - La búsqueda es muy rápida
  - La inserción y eliminación de enlaces también son operaciones rápidas
  - Si no se necesitan costes, se puede usar un sólo bit para cada elemento de la matriz
- Desventajas
  - Suele requerir mayor memoria, se usa en grafos más pequeños

Matriz de incidencia [Nodo/Enlace]

- Matriz B de N x E elementos
  - Las filas representan los nodos y las columnas los enlaces
    - b(i,j) = 1 si el nodo i es origen del enlace j
    - b(i,j) = -1 si el nodo j es destino del enlace j
    - b(i,j) = 0 en caso contrario
  - Sólo hay 2E elementos no cero (1 ó -1)
  - Cada columna tiene un elemento 1 y un elemento -1
- Ventajas
  - Estructura matricial tiene varias propiedades teóricas interesantes
- Desventajas
  - Capacidad necesaria muy elevada (N · E)
  - La eliminación de enlaces es complicada
  - La localización de un enlace también es un procedimiento complejo

Estrellas

- Estrella hacia adelante
  - Representación similar a la lista de adyacencia
  - Los enlaces que parten de un nodo se guardan como un array, no como una lista enlazada
  - Ventajas
    - Escasa capacidad de almacenaje
    - Localización de enlaces sencilla
  - Desventajas
    - Eliminación e incorporación de enlaces
- Estrella hacia atrás
  - Estructura complementaria que determina los enlaces entrantes a un nodo

Lista Vs. Matriz de adyacencia

- Las dos representaciones más relevantes son la lista y la matriz de adyacencia
- En la mayoría de los casos la lista de adyacencia es la estructura de almacenamiento más apropiada

Aspecto	**Estructura "ganadora"**
Chequear la existencia de un enlace	Matriz
Encontrar el grado de un nodo	Lista
Memoria en grafos sparse	Lista (N+E) Vs. N^2
Memoria en grafos "densos" (E ~ N^2)	Matriz (escasa diferencia)
Incorporar/borrar enlaces	Matriz – O(1) Vs. O(d)
Recorrer el grafo	Lista - Θ(N+E) Vs. $\Theta(N^2)$
Mayoría de problemas	Lista

Lista adyacencia: Implementación (1/3)

```
#define MAXV 1000                   /* maximo numero de nodos */

typedef struct edgenode {
   int y;                           /* destino del enlace */
   int weight;                      /* coste del enlace */
   struct edgenode *next;           /* lista enlazada */
} edgenode_t;

main {
   ...                              /* variables */
   edgenode_t *edges[MAXV+1]; /* informacion adyacencia */
   int nvertices;                   /* numero de vertices */
   int nedges;                      /* numero de enlaces */
   bool directed;                   /* grafo dirigido? */
   ...
}
```

Lista adyacencia: Implementación (2/3)

```
initialize_graph(edgenode_t *edges[MAXV+1]) {
    int i;
    for( i=1 ; i<=MAXV ; i++ ) {
        edges[i]=NULL;
    }
}
main {
    ...
    initialize_graph(edges);
    scanf("%d %d",&nvertices, &nedges);      /* conocemos nedges */
    for( i=1 ; i<=nedges ; i++ ) {
        scanf("%d %d %d",&x,&y,&w);
        insert_edge(edges,x,y,w,directed);
    }
}
```

Lista adyacencia: Implementación (3/3)

```
insert_edge(edgenode_t *edges[MAXV+1], int x, int y, int w,
bool directed)
{
   edgenode_t *p;
   p = (edgenode_t *) malloc(sizeof(edgenode));
   p->weight = w;
   p->y = y;

   p->next = edges[x];
   edges[x] = p;                  /* insercion al principio */

   if(directed == FALSE) {
        insert_edge(edges,y,x,w,TRUE);
   }
}
```

Contenido

- Motivación
- Complejidad de algoritmos
- Conceptos básicos de grafos
- Representación de grafos
- Algoritmos básicos de grafos

Introducción

- Problema fundamental en teoría de grafos: visitar todos los nodos y enlaces de manera sistemática
  - Evitar “recorrer” de manera repetitiva nodos/enlaces ya procesados
- Estado de los nodos (se pueden utilizar códigos de colores)
  - No descubierto: el nodo no se ha “visitado” aún
  - Descubierto: el nodo se ha “visitado”, pero no se han explorado todos sus enlaces
  - Procesado: ya se han explorado todos los enlaces del nodo
- Algoritmos
  - Breadth-First Search: *“búsqueda en amplitud”*
  - Depth-First Search: *“búsqueda en profundidad”*

Breadth-First Search

- Dado un grafo G(N,E) y un nodo origen s, el algoritmo BFS sistemáticamente explora los enlaces de G para descubrir los nodos que son “alcanzables” desde s
- La búsqueda se hace de manera paulatina en función de la distancia a los posibles destinos (búsqueda en amplitud)
  - Los nodos a distancia k+1 sólo se descubren cuando ya se han visitado todos los que estén a distancia k
- Todos los nodos tendrán un único “padre” (menos el origen), por lo que el BFS genera un árbol del grafo
- Mecanismo básico empleado por otros algoritmos
  - Dijkstra – camino más corto
  - Prim – *minimum spanning tree*

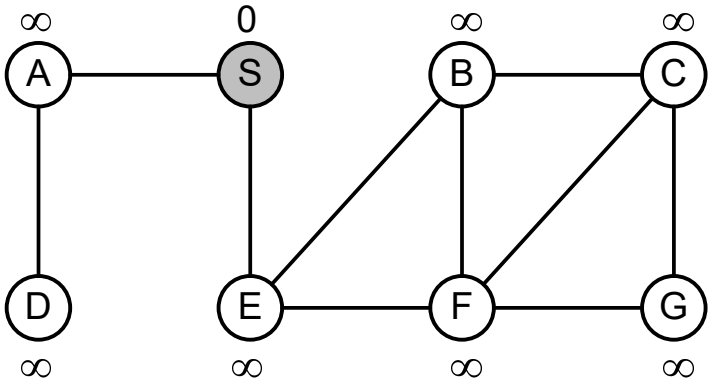
Breadth-First Search - Pseudocódigo

```
for each vertex u in N – {s} do
   state[u] = "undiscovered"          // color[u] = WHITE
   p[u] = nil                         // no predecesor (padre)
state[s] = "discovered"               // color[s] = GRAY
p[s] = nil                            // origen sin predecesor
Q = {s}
while Q ≠ ø
   u = dequeue(Q)                     // extrae primer nodo de Q
   // procesado de u
   for each v adj[u] do
      // procesado de enlace (u,v)
      if state[v] == "undiscovered"   // if state[v] == WHITE
         state[v] = "discovered"      // color[v] = GRAY
         p[v] = u                     // padre de v es u
         enqueue(Q,v)
   state[u] = "processed"             // color[u] = BLACK
```

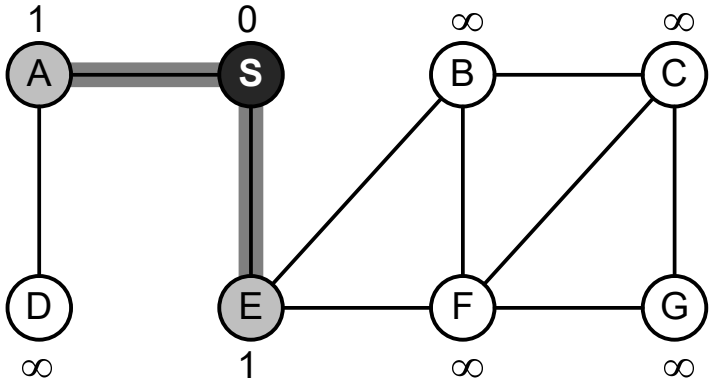
Breadth-First Search – Análisis

- Fases
  - Inicialización: O(N)
  - Enqueue/Dequeue: O(1)
    - Total de acciones enqueue/dequeue: O(N)
  - Cada lista de adyacencia se explora una única vez
    - Tiempo total: O(E)
- Tiempo total: O(N+E)
  - Lineal con el tamaño de la lista de adyacencia de G

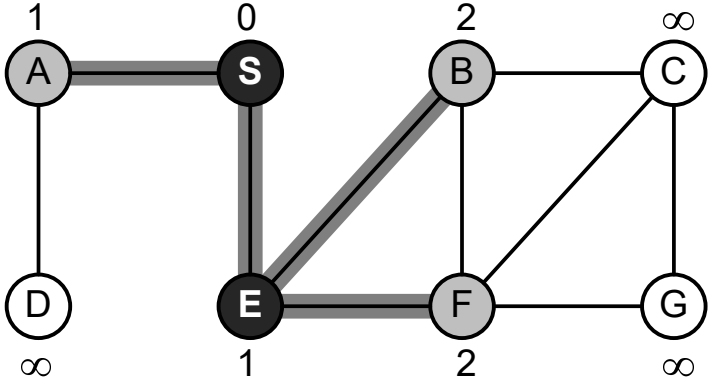
Breadth-First Search – Ejemplo (1/3)



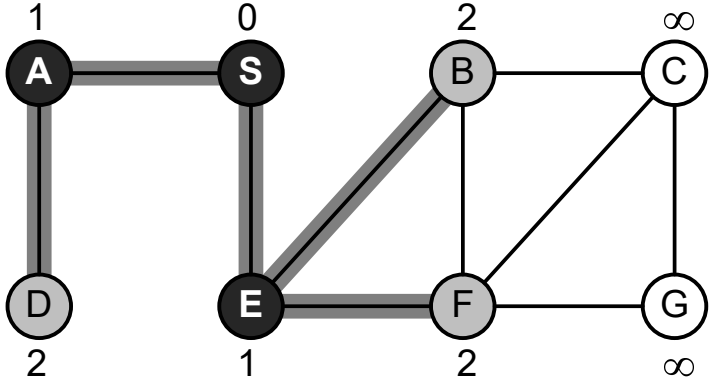
Q = {S}



Q = {E,A}

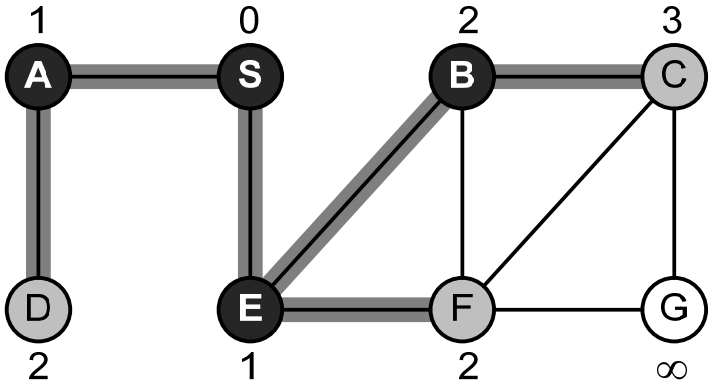


Q = {A,B,F}

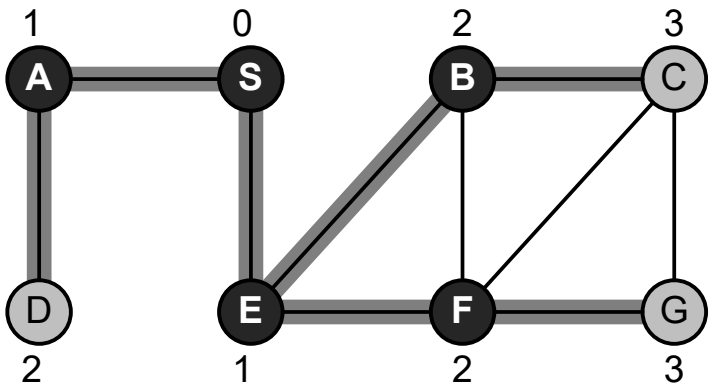


Q = {B,F,D}

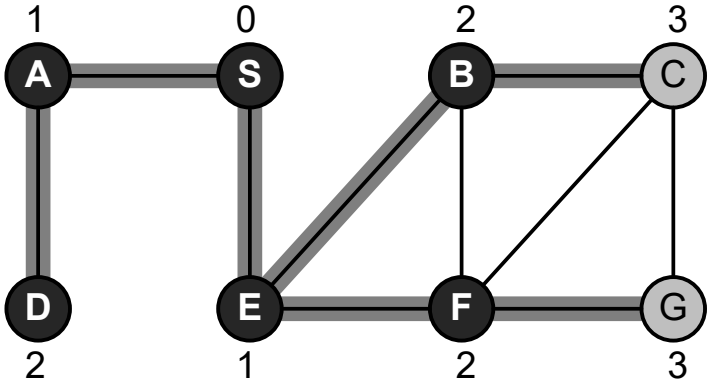
Breadth-First Search – Ejemplo (2/3)



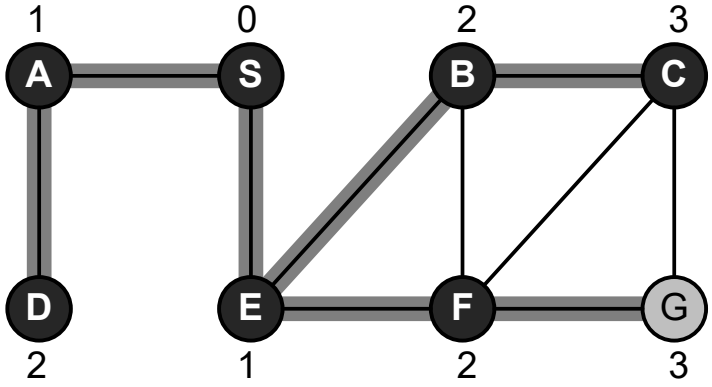
Q = {F,D,C}



Q = {D,C,G}

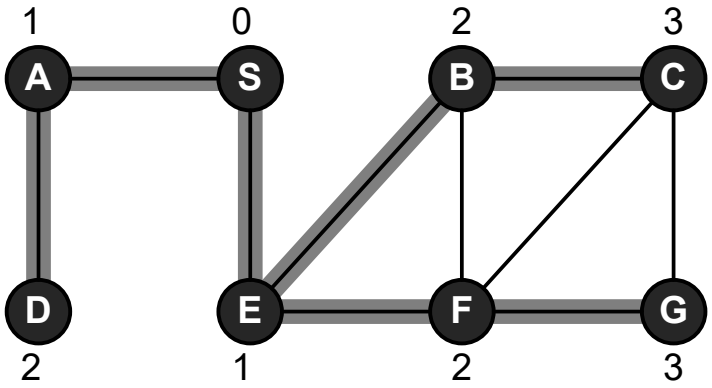


Q = {C,G}

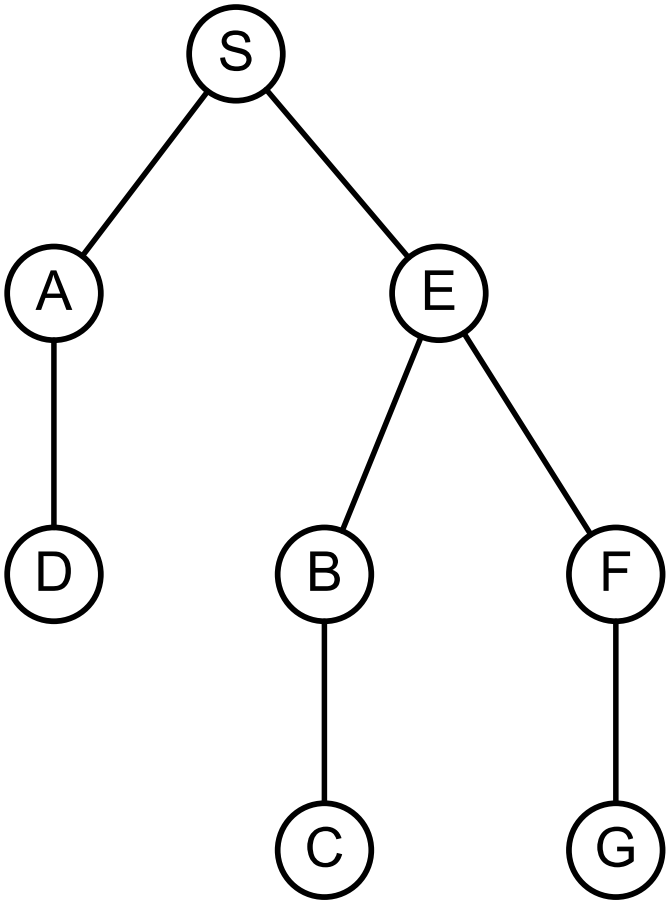


Q = {G}

Breadth-First Search – Ejemplo (3/3)



Q = {ø}



Árbol que construye BFS

Depth-First Search

- El algoritmo DFS recorre el grafo “profundizando” hasta que sea posible
  - Se recorren los enlaces de los nodos que se acaban de descubrir
  - Se vuelve hacia atrás cuando ya no se pueda “explorar” más lejos
- BFS mantiene la lista de nodos como una cola/queue (FIFO), mientras que en DFS se utiliza una pila/stack (LIFO)
- El DFS genera un subgrafo de predecesores que puede estar formado por varios árboles (a diferencia de BFS)
- Uso de marcas temporales para obtener información del grafo

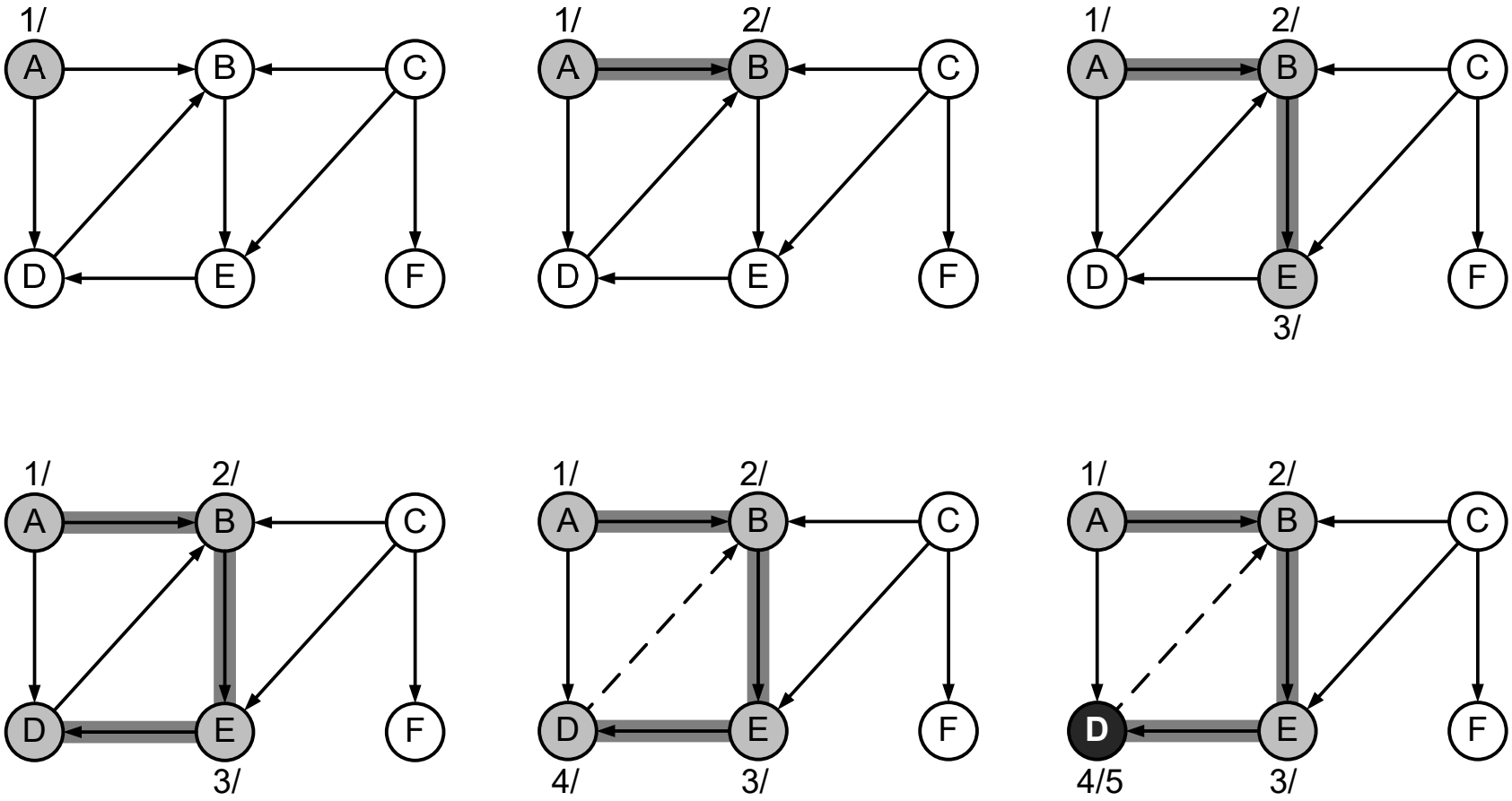
Depth-First Search - Pseudocódigo

```
DFS(G)
   for each vertex u in N
      state[u] = "undiscovered"          // color[u] = WHITE
      p[u] = nil
   time = 0                              // variable global timestamp
   for each vertex u in N
      dfs(G,u)                           // llamo a dfs para cada nodo
dfs(G,u)
   state[u] = "discovered"               // color[u] = GRAY
   entry[u] = time
   time = time + 1
   for each v adj(u)
      if state[v] == "undiscovered"      // if color[u] == WHITE
         p[v] = u                        // parent de v es u
         dfs(G,v)                        // llamo a dfs (RECURSIVIDAD)
   state[u] = "processed"                // color[u] = BLACK
   exit[u] = time
   time = time + 1
```

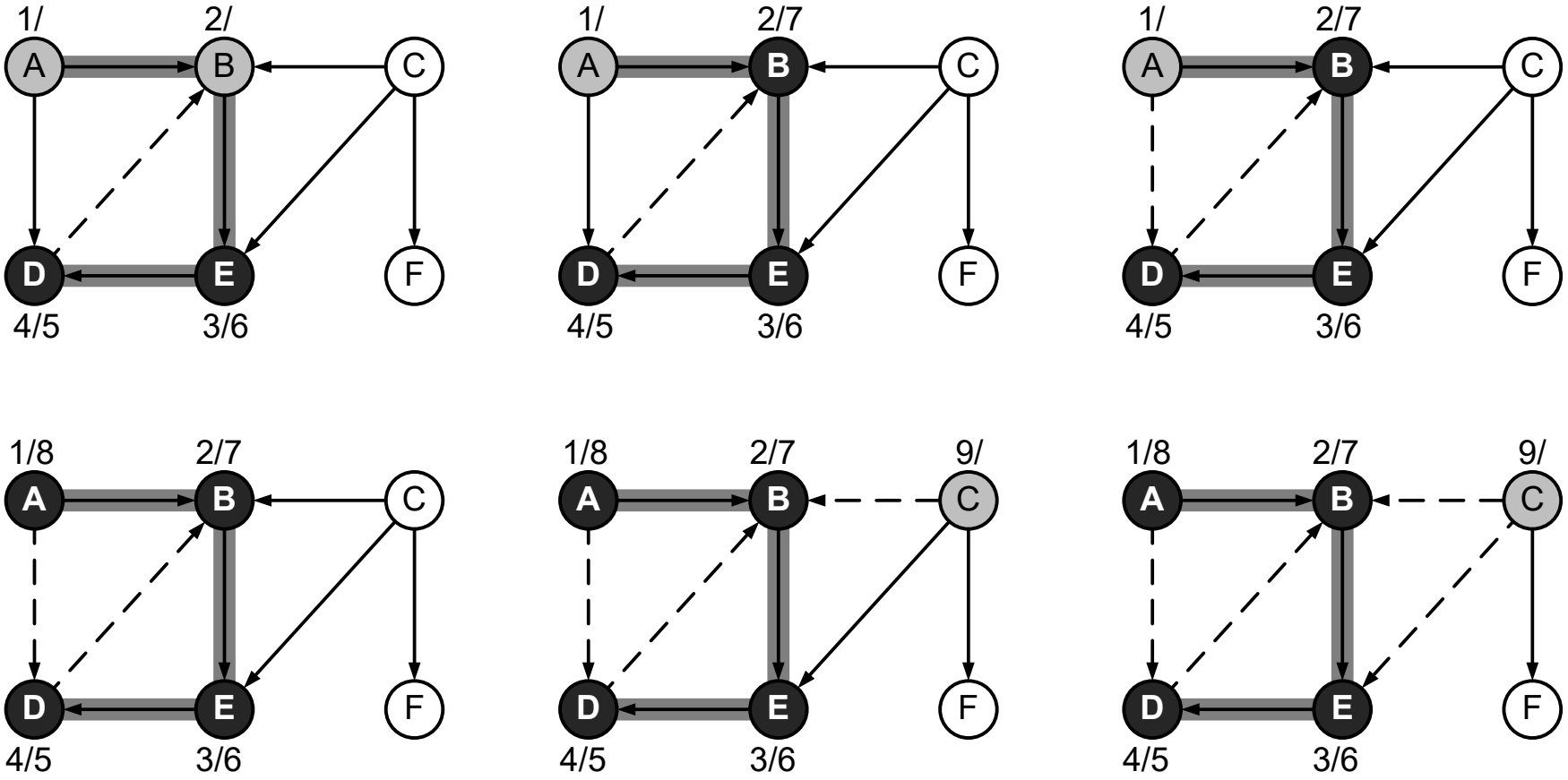
Depth-First Search - Propiedades

- Marca temporal
  - El nodo u es antecesor de v si
    - entry[u] < entry[v]: hay que empezar a procesar u antes que v
    - exit[v] < exit[u]: hay que acabar de procesar v antes que u
  - Número de descendientes de u
    - descendientes[u] = (entry[u] – exit[u])/2
- Análisis
  - DFS(G,u) se llama, como máximo N veces
  - Recorre la lista de adyacencia – E
  - Tiempo total O(N+E)
    - Comer lo describe como Θ(N+E)

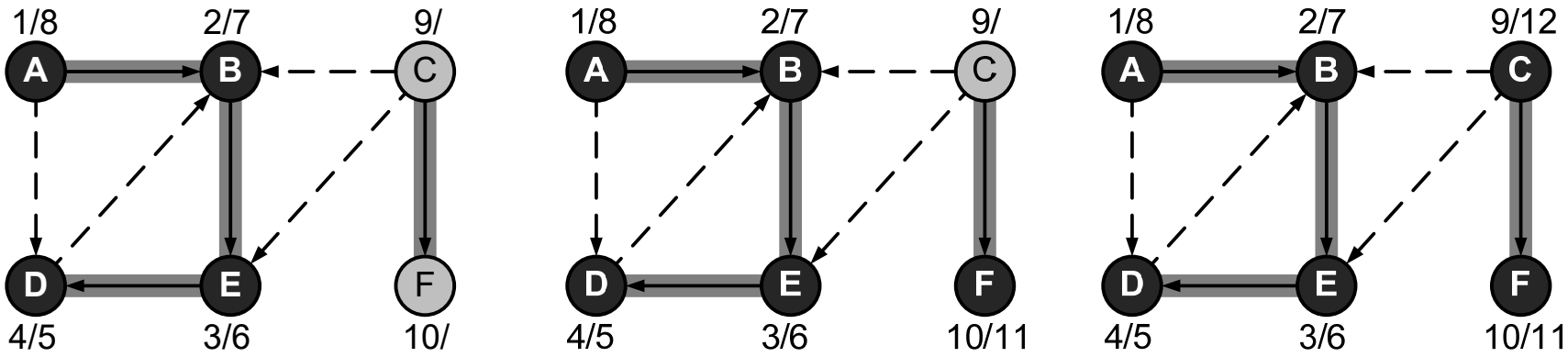
Depth-First Search – Ejemplo



Depth-First Search – Ejemplo



Depth-First Search – Ejemplo



Aplicaciones

- BFS
  - Búsqueda de componentes conectados (grafos no dirigidos)
  - Coloración de grafos (bipartitos)
    - Caso particular de un problema complejo (NP-completo)
- DFS
  - Búsqueda de componentes fuertemente conectados (grafos dirigidos)
  - Ordenación topológica
  - Encontrar ciclos
  - Vértices “articulación” – single point of failure