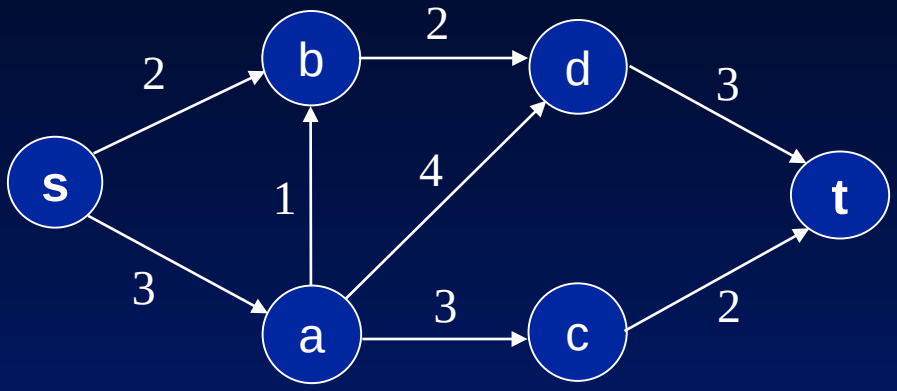
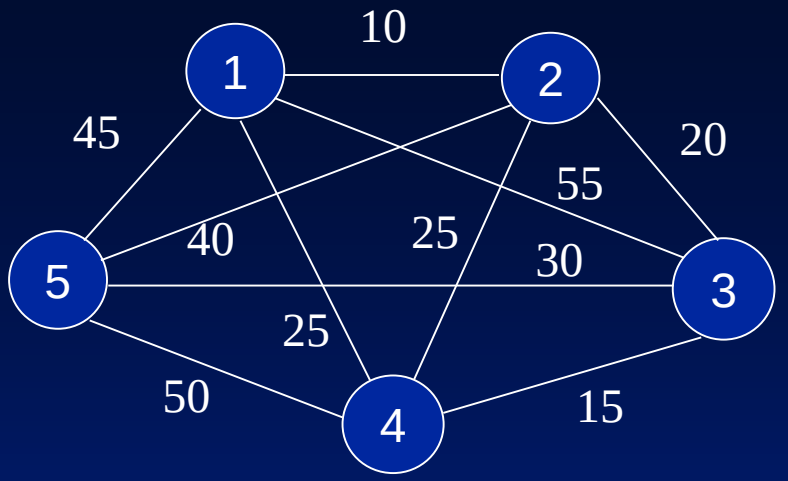
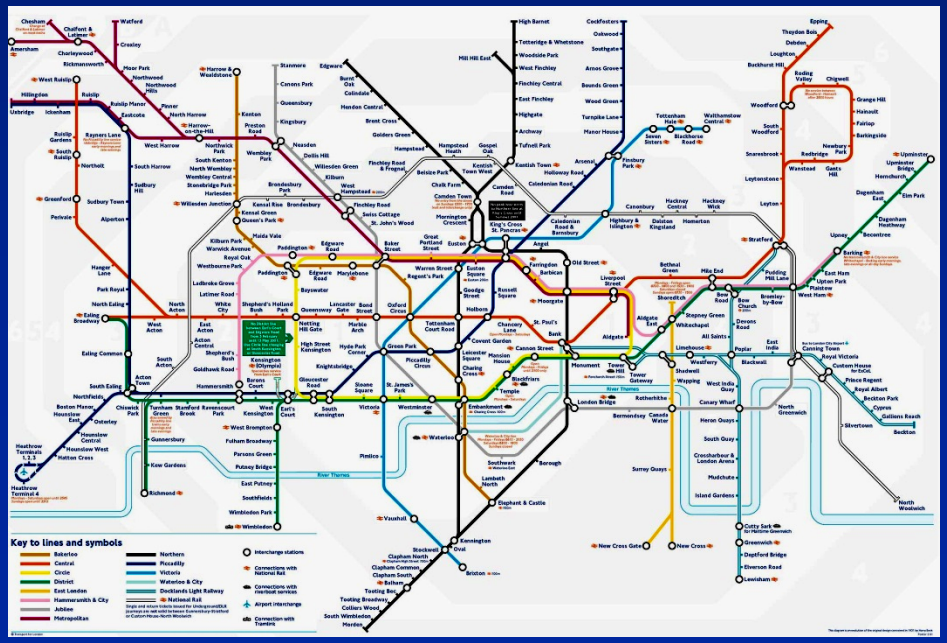

s
a
b
c
d
t
2
2
3
1
4
3
3
2


1
2
3
4
5
10
45
20
55
40
25
30
25
50
15

Grafos


Key to lines and symbols

Indice general

Indice

- Introducción.
- Definiciones.
- Tipo de dato abstracto grafo.
- Estructuras de datos para grafos.
  - Lista de aristas.
  - Lista de adyacencia.
  - Matriz de adyacencia.

Introducción

- Los grafos se usan para modelar problemas definidos en términos de relaciones o conexiones entre objetos.
- Tienen un amplio uso en ingeniería para representar redes de todo tipo:
  - transporte (tren, carretera, avión),
  - servicios (comunicación, eléctrica, gas, agua),
  - de actividades en el planeamiento de proyectos, etc.

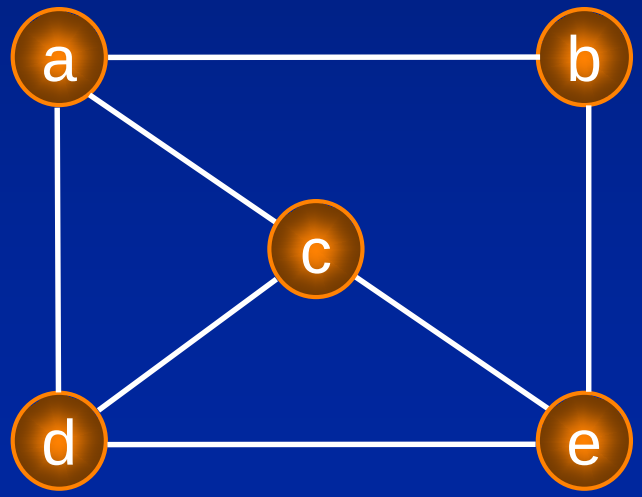
¿Qué es un grafo?

- Un grafo G = (V, E) está compuesto de:

  V : conjunto de *vértices o nodos*

  E : conjunto de *aristas o arcos* que conectan los vértices en V

- Una arista e = (*v*, *w*) es un par de vértices
- Ejemplo:

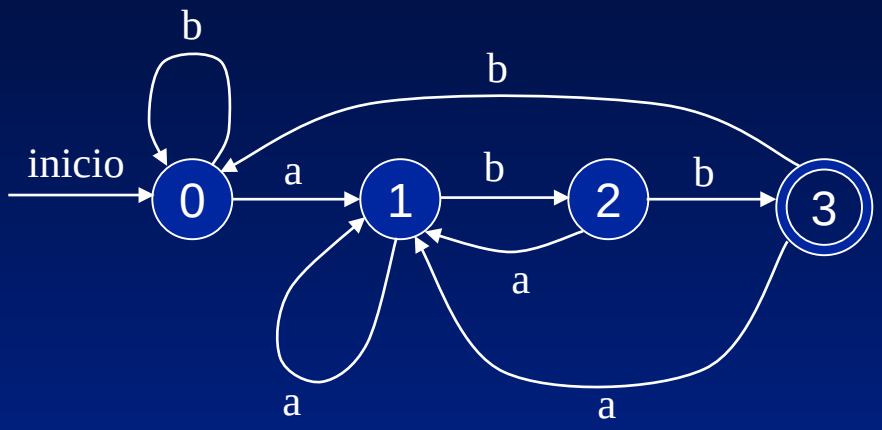


V = { a, b, c, d, e}

E = { (a, b), (a, c), (a,d), (b, e), (c, d), (c, e), (d, e) }

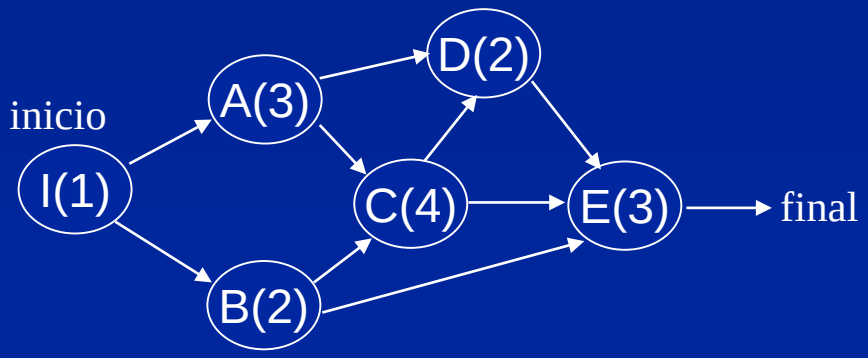
Aplicaciones

- Grafo de transiciones (AFD)

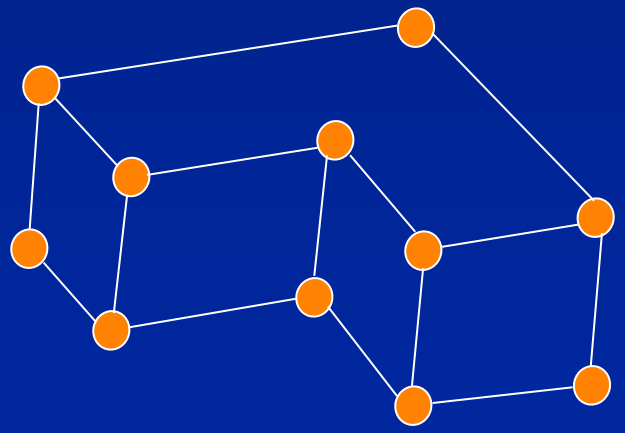


- Tiempo de vuelos aéreos

- Planificación de tareas (Pert/CPM)



- Grafo asociado a un dibujo de líneas (visión artificial)



Definiciones

- *Arista dirigida*: par ordenado (u, v)

- *Arista no dirigida*: par no ordenado (u, v)

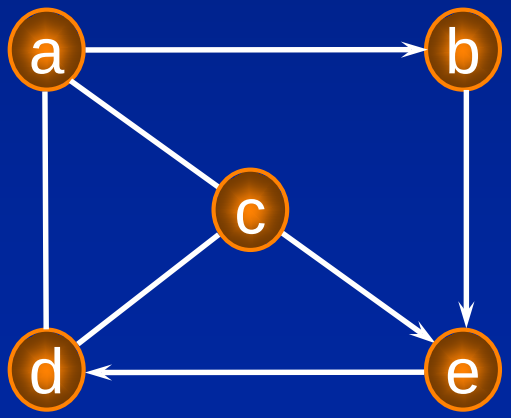
- *Grafo dirigido* o *digrafo*: grafo cuyas aristas son todas dirigidas.
- *Grafo no dirigido* o *grafo*: grafo cuyas aristas son todas no dirigidas.
- *Grafo mixto*: grafo con aristas dirigidas y no dirigidas.

Definiciones

- *Vértices finales* o *extremos de la arista*: vértices unidos por una arista.
  - *Vértice origen*: primer vértice de una arista dirigida.
  - *Vértice destino*: segundo vértice de una arista dirigida.
- *Arista incidente en un vértice*: si el vértice es uno de los vértices de la arista.
- *Aristas salientes de un vértice*: aristas dirigidas cuyo origen es ese vértice.
- *Aristas entrantes de un vértice*: aristas dirigidas cuyo destino es ese vértice.

Definiciones

- *Vértices adyacentes*: vértices finales de una arista.
  - Un vértice w es *adyacente* a v sí y sólo si (v, w) (ó (w, v)) pertenece a E.
  - En grafos no dirigidos la relación de adyacencia es *simétrica*.
  - En grafos dirigidos la relación de adyacencia no es simétrica.



Vértices adyacentes:
a = { b, c, d }
b = { e }
c = { a, d, e }
d = { a, c }
e = { d }

Definiciones

- *Grado de un vértice* v (*grado*(v)) en un grafo: número de aristas incidentes en v o número de vértices adyacentes.
  - En un digrafo:
    - *Grado entrante de un vértice* v (*graent*(v)): número de aristas entrantes a v.
    - *Grado saliente de un vértice* v (*grasal*(v)): número de aristas salientes de v.
  - Si G es un grafo con *m* aristas, entonces

$$\sum_{v \in G} grado(v) = 2m$$

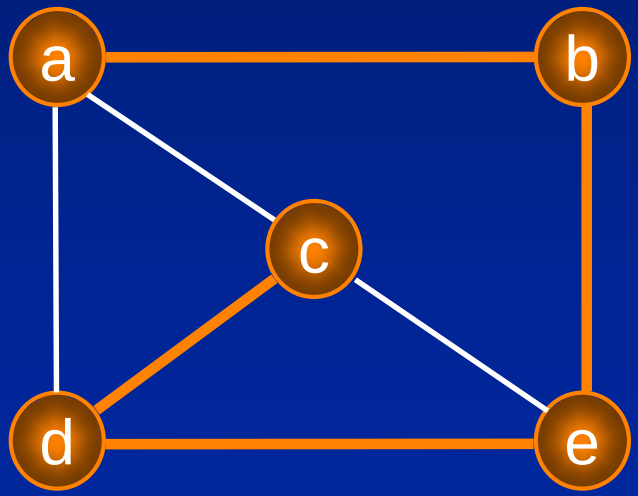
  - Si G es un digrafo con *m* aristas, entonces

$$\sum_{v \in G} graent(v) = \sum_{v \in G} grasal(v) = m$$

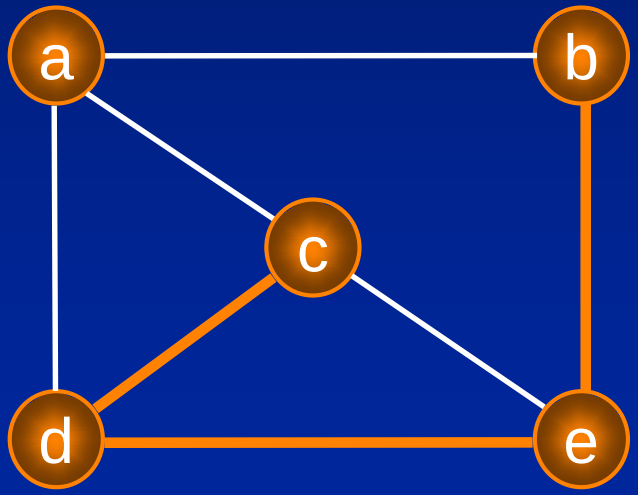
Definiciones

- Sea G es un grafo con *n* vértices y *m* aristas.
  - Si G es no dirigido, entonces $m \leq n(n-1)/2$.
  - Si G es dirigido, entonces $m \leq n(n-1)$.

- *Camino*: secuencia de vértices $<v_1, v_2, \ldots, v_n>$ tal que (v_i, v_{i+1}) son adyacentes.



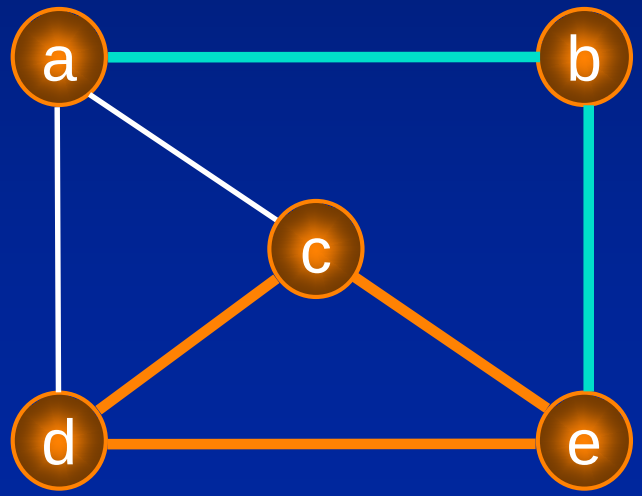
$C_1 = \{ a, b, e, d, c\}$



$C_2 = \{ b, e, d, c\}$

Definiciones

- *Camino simple*: todos los vértices son distintos.
- *Longitud de un camino*: número de aristas del camino = n – 1.
- *Ciclo*: camino simple que tiene el mismo vértice inicial y final.

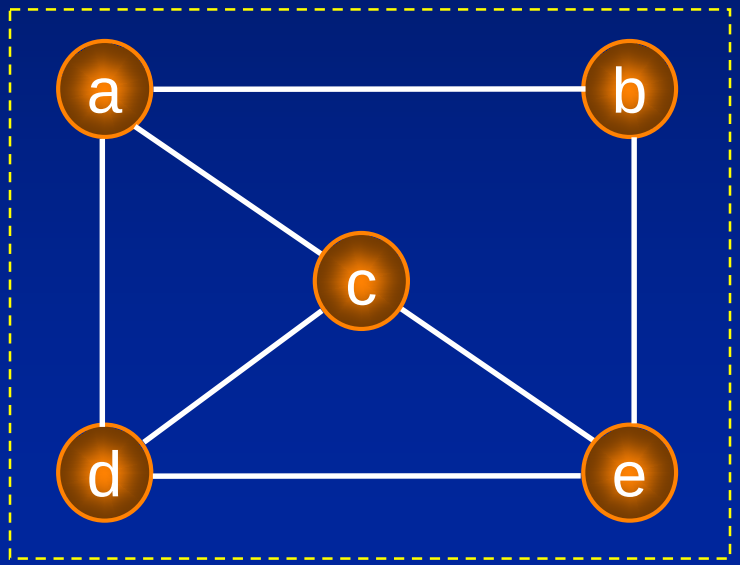


Camino simple = { a, b, e}

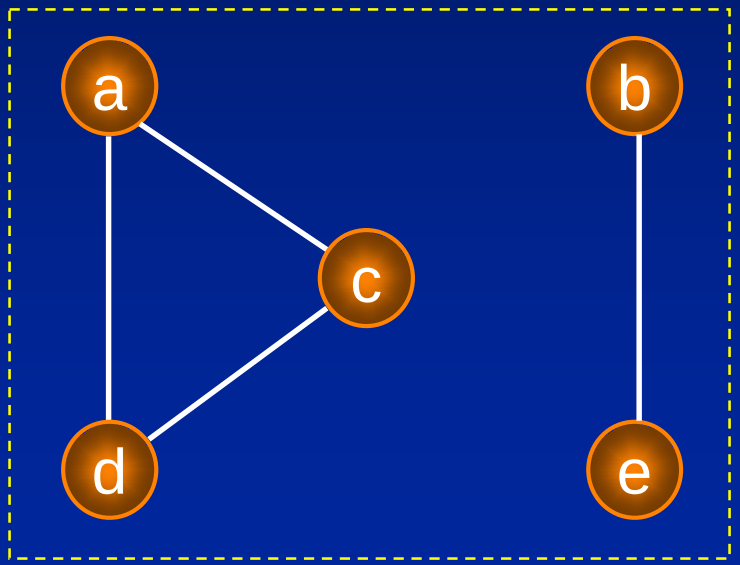
Ciclo = { c, e, d, c}

Definiciones

- Dos vértices v, w están *conectados* si existe un camino de v a w.
- *Grafo conectado* (*conexo*): si hay un camino entre cualquier par de vértices.
  - Si es un grafo dirigido se llama fuertemente conexo.



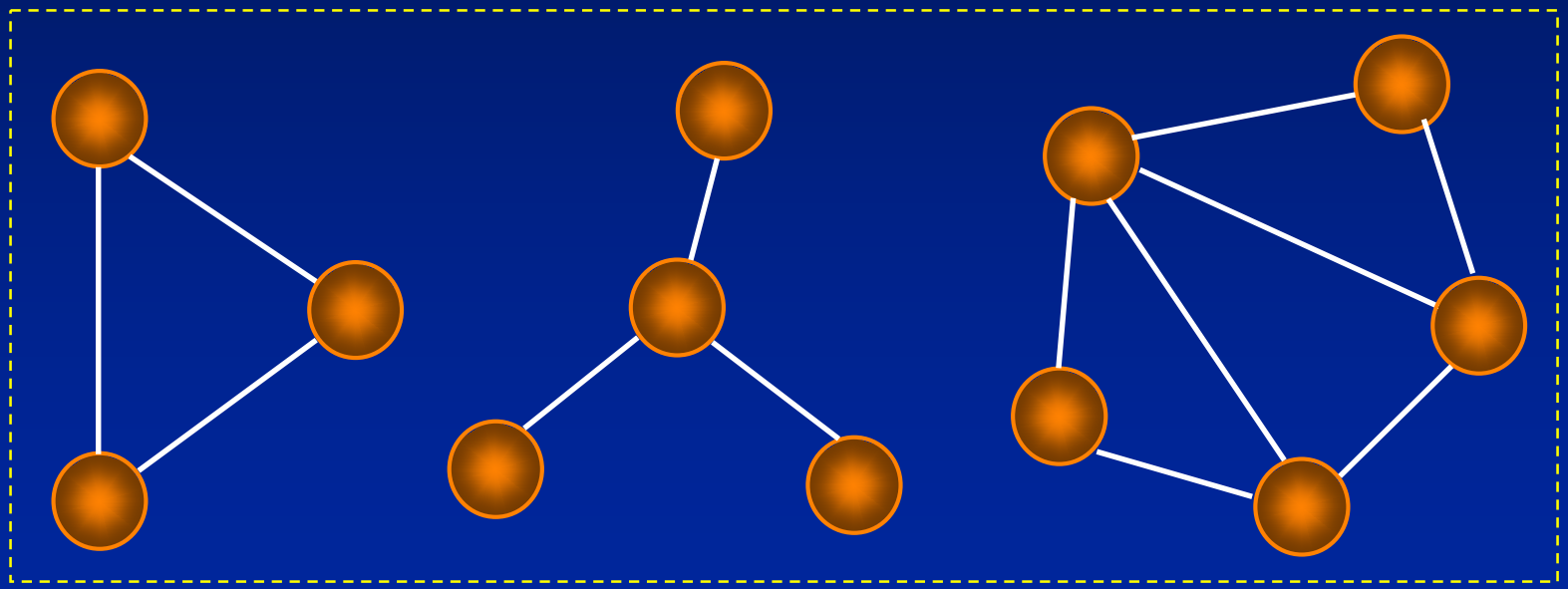
Conectado



No conectado

Definiciones

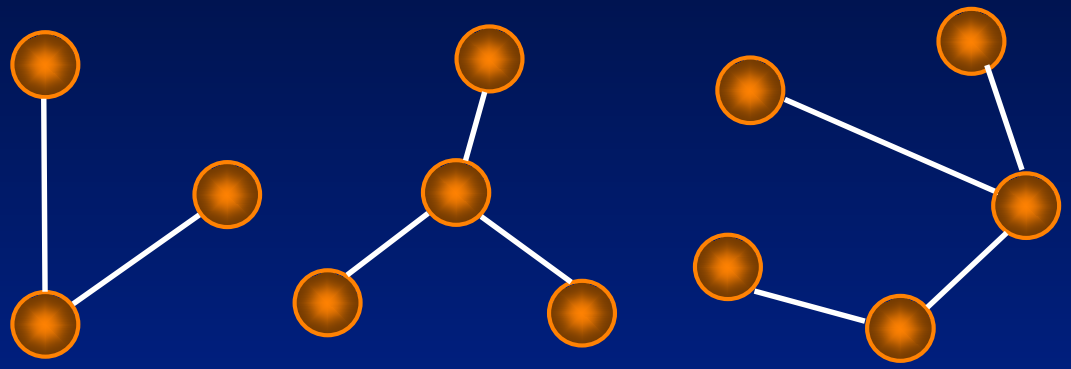
- *Subgrafo*: subconjunto de vértices y aristas que forman un grafo.
- *Componente conectado*: subgrafo conectado máximo.



3 componentes conectados

Definiciones

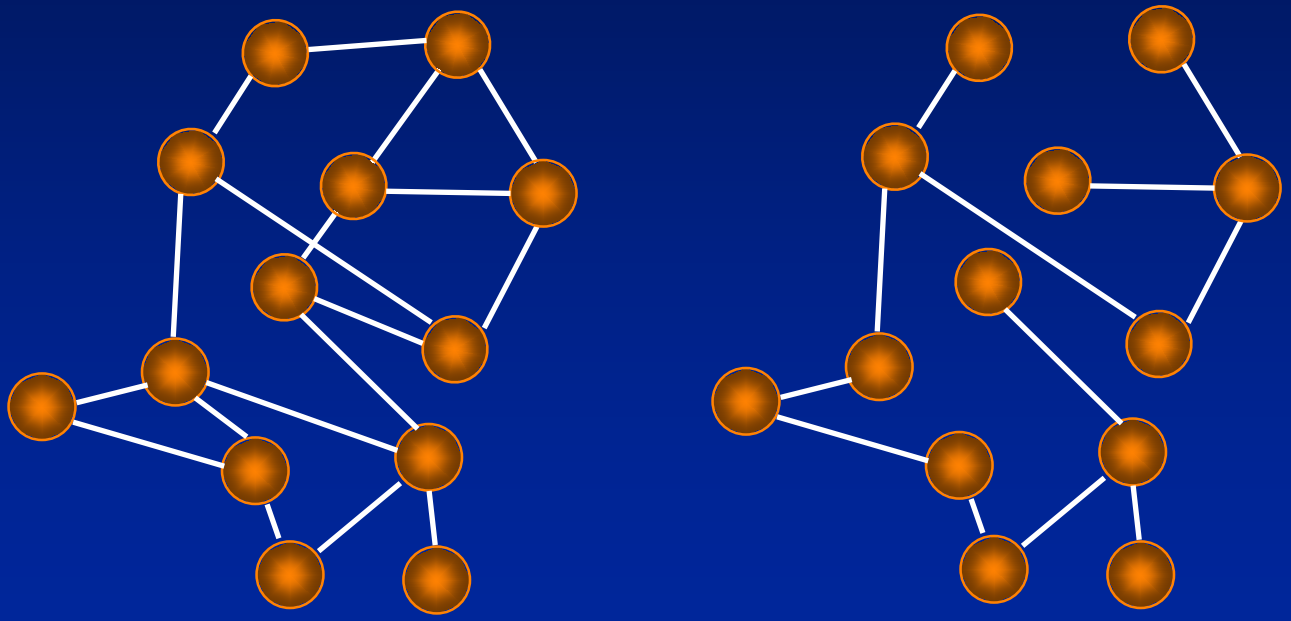
- *Árbol*: grafo conectado sin ciclos.
- *Bosque*: colección de árboles.



- *Grafo completo*: todos los pares de vértices son adyacentes. ($m = n*(n-1)/2$)
- En un grafo no dirigido G con n vértices y m aristas se cumple lo siguiente:
  - Si G es conectado, entonces $m \geq n - 1$
  - Si G es un árbol, entonces $m = n - 1$
  - Si G es un bosque, entonces $m \leq n - 1$

Definiciones

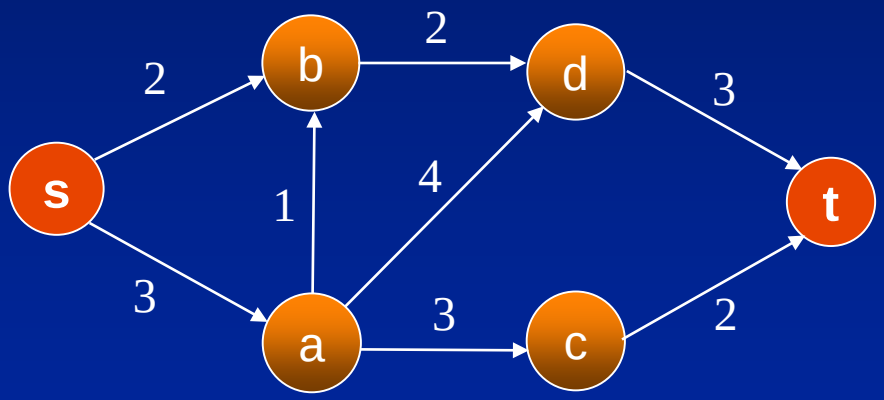
- *Árbol de cubrimiento de un grafo G*: subgrafo que
  - es un árbol.
  - contiene todos los vértices de G.



El fallo de una arista desconecta el sistema (menos tolerante a fallos).

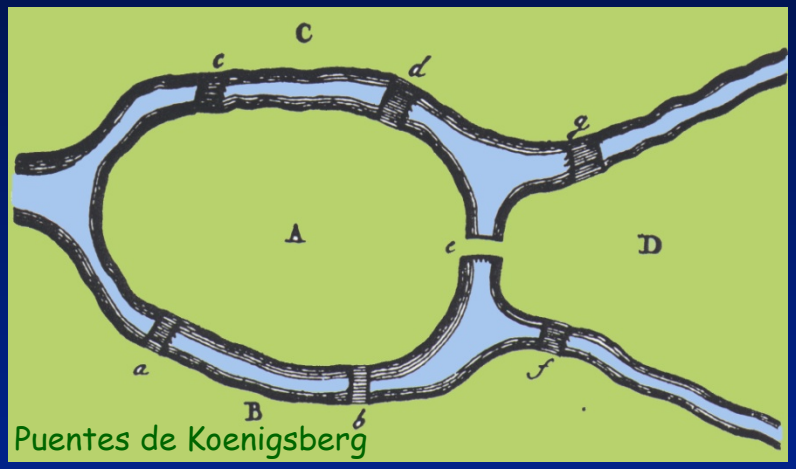
Definiciones

- Un *grafo* está *etiquetado* si asociamos a cada arista un peso o valor.
- *Grafo con pesos*: grafo etiquetado con valores numéricos.



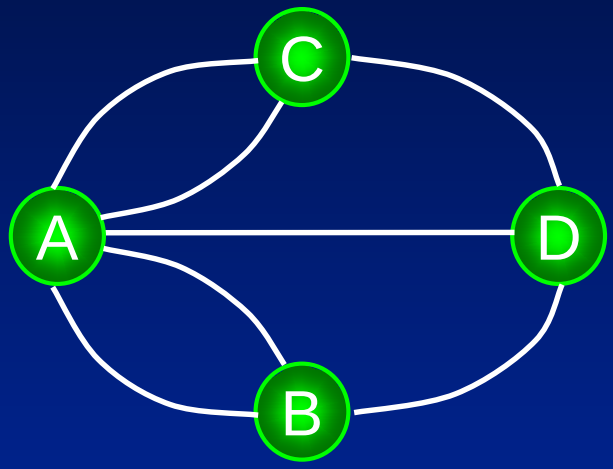
Definiciones

- *Circuito de Euler*: camino que recorre todas las aristas una vez y retorna al vértice de partida.



Puentes de Koenigsberg

grafo



- *Teorema de Euler (*1736*)*: un grafo tiene un circuito de Euler si y solo si todos los vértices tienen grado par.
- Más definiciones y teoremas en Teoría de Grafos.

El tipo de dato abstracto Grafo

- El TDA Grafo es un contenedor de posiciones que almacena los vértices y las aristas del grafo.
- Operaciones para la información posicional:
  - tamano(), devuelve el número de vértices más el número de aristas de G.
  - estaVacio()
  - elementos()
  - posiciones()
  - reemplazar(p, r)
  - intercambiar(p, q)

    donde p y q indican posiciones, y r indica un elemento de información.

El tipo de dato abstracto Grafo

- Operaciones generales: (v: vértice, e: arista, o: elemento de información).

numVertices()	Devuelve el número de vértices de G
numAristas()	Devuelve el número de aristas de G
vertices()	Devuelve una lista de los índices de los vértices de G
aristas()	Devuelve una lista de los índices de las aristas de G
grado(v)	Devuelve el grado de v
verticesAdyacentes(v)	Devuelve una lista de los vértices adyacentes a v
aristasIncidentes(v)	Devuelve una lista de las aristas incidentes en v
verticesFinales(e)	Devuelve un array de tamaño con los vértices finales de e
opuesto(v, e)	Devuelve los puntos extremos de la arista e diferente a v
esAdyacente(v, w)	Devuelve verdadero si los vértices v y w son adyacentes

El tipo de dato abstracto Grafo

- Operaciones con aristas dirigidas:

aristasDirigidas()	Devuelve una lista de todas las aristas dirigidas
aristasNodirigidas()	Devuelve una lista de todas las aristas no dirigidas
gradoEnt(v)	Devuelve el grado de entrada de v
gradoSalida(v)	Devuelve el grado de salida de v
aristasIncidentesEnt(v)	Devuelve una lista de todas las aristas de entrada a v
aristasIncidentesSal(v)	Devuelve una lista de todas las aristas de salida a v
verticesAdyacentesEnt(v)	Devuelve una lista de todas las aristas adyacentes a v a través de las aristas de entrada a v
verticesAdyacentesSal(v)	Devuelve una enumeración de todas las aristas adyacentes a v a través de las aristas de salida a v
destino(e)	Devuelve el destino de la arista dirigida e
origen(e)	Devuelve el origen de la arista dirigida e
esDirigida(e)	Devuelve verdadero si la arista e es dirigida

El tipo de dato abstracto Grafo

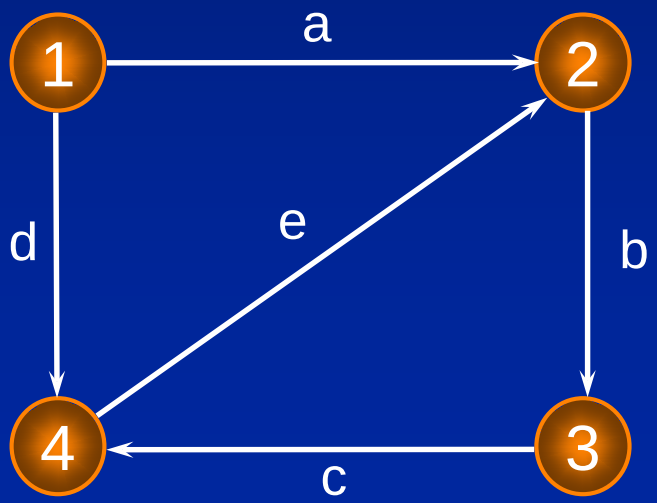
- Operaciones para actualizar grafos:
  - insertaArista(v, w, o) Inserta y devuelve una arista no dirigida entre los vértices v y w, almacenando el objeto o en esta posición
  - insertaAristaDirigida(v, w, o) Inserta y devuelve una arista dirigida entre los vértices v y w, almacenando el objeto o en esta posición
  - insertaVertice(o) Inserta y devuelve un nuevo vértice almacenando el objeto o en esta posición
  - eliminaVertice(v) Elimina vértice v y todas las aristas incidentes
  - eliminaArista(e) Elimina arista e
  - convierteNoDirigida(e) Convierte la arista e en no dirigida
  - invierteDireccion(e) Invierte la dirección de la arista dirigida e
  - asignaDireccionDesde(e, v) Produce arista dirigida e salga del vértice v
  - asignaDireccionA(e, v) Produce arista dirigida e entrante al vértice v

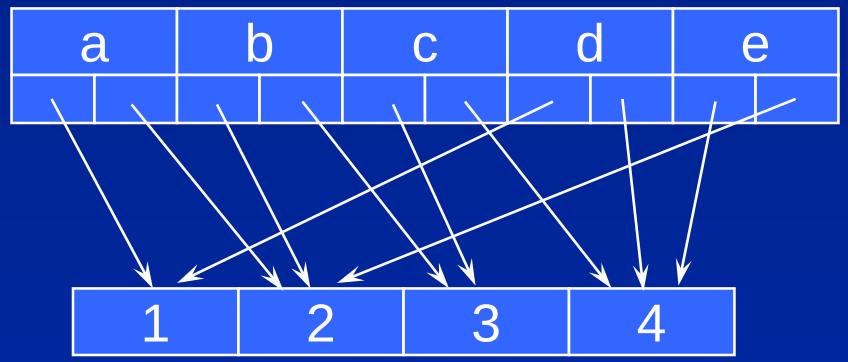
Estructuras de datos para Grafos

- Se necesita almacenar los vértices y las aristas del grafo y realizar eficientemente las operaciones del TDA Grafo.
- Las estructuras de datos usuales son:
  - Lista de aristas.
  - Lista de adyacencia.
  - Matriz de adyacencia.

Lista de Aristas

- La estructura lista de aristas almacena los vértices y las aristas en secuencias sin ordenar.
- Fácil de implementar.
- Hallar las aristas incidentes sobre un determinado vértice es ineficiente porque requiere el examen entero de la estructura que almacena las aristas.



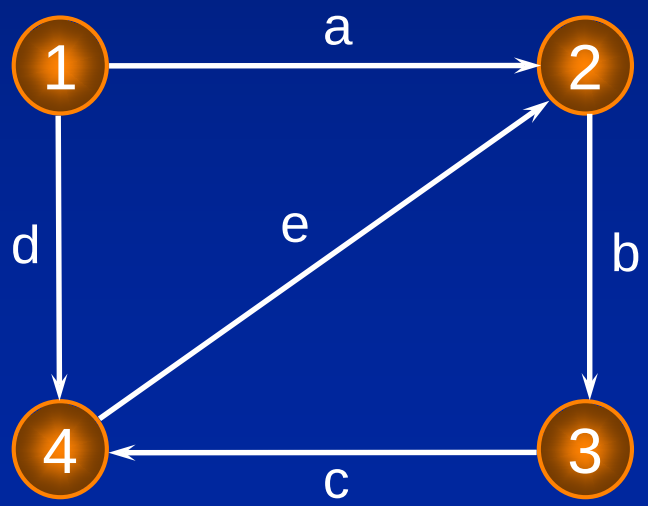


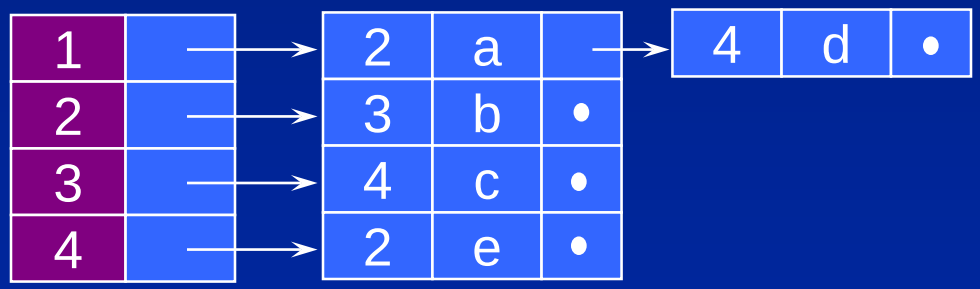
Eficiencia de la estructura Lista de Aristas

Operación	Tiempo
tamano, estaVacio, remplazarElemento, intercambiar	$O(1)$
numVertices, numAristas	$O(1)$
vertices	$O(n)$
aristas, aristasDirigidas, aristasNodirigidas	$O(m)$
elementos, posiciones	$O(n + m)$
verticesFinales, opuesto, origen, destino, esDirigida, grado, gradoEnt, gradoSalida	$O(1)$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(m)$
esAdyacente	$O(m)$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(1)$
insertaVertice	$O(1)$
eliminaVertice	$O(m)$
Espacio requerido	$O(n + m)$

Lista de Adyacencia

- Lista de adyacencia del vértice v: secuencia de vértices adyacentes a v.
- Representa el grafo por las listas de adyacencia de todos los vértices.
- Es la estructura más usada para representar grafos con pocas aristas (dispersos).



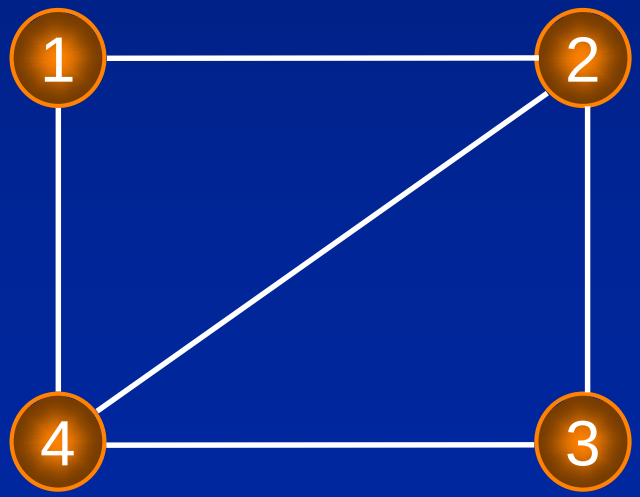


Eficiencia de la estructura Lista de Adyacencia

Operación	Tiempo
tamano, estaVacio, remplazarElemento, intercambiar	$O(1)$
numVertices, numAristas	$O(1)$
vertices	$O(n)$
aristas, aristasDirigidas, aristasNodirigidas	$O(m)$
elementos, posiciones	$O(n + m)$
verticesFinales, opuesto, origen, destino, esDirigida, grado, gradoEnt, gradoSalida	$O(1)$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(\text{grado}(v))$
esAdyacente	$O(\min(\text{grado}(u), \text{grado}(v))$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(1)$
insertaVertice	$O(1)$
eliminaVertice	$O(\text{grado}(v))$
Espacio requerido	$O(n + m)$

Matriz de Adyacencia

- Matriz M[i][j] con entradas para todos los pares de vértices.
  - En grafos no etiquetados:
    - M[i][j] = verdadero, si hay una arista (i, j) en el grafo.
    - M[i][j] = falso, si no hay una arista (i, j) en el grafo.
  - En grafos no dirigidos: M[i][j] = M[j][i]. La matriz es simétrica.

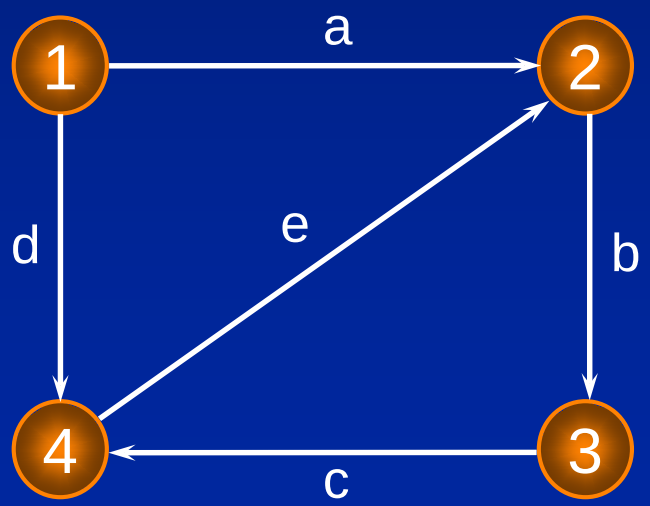


	1	2	3	4
1	F	V	F	V
2	V	F	V	V
3	F	V	F	V
4	V	V	V	F

Matriz de Adyacencia

- En grafos etiquetados:
  - M[i][j] = atributo de la arista (i, j) en el grafo, indicador especial si no hay una arista (i, j).

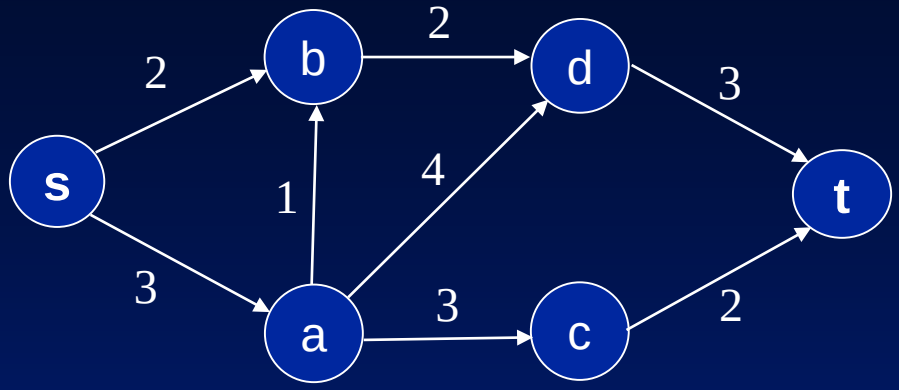
- Es la estructura más usada para representar grafos con muchas aristas (densos).

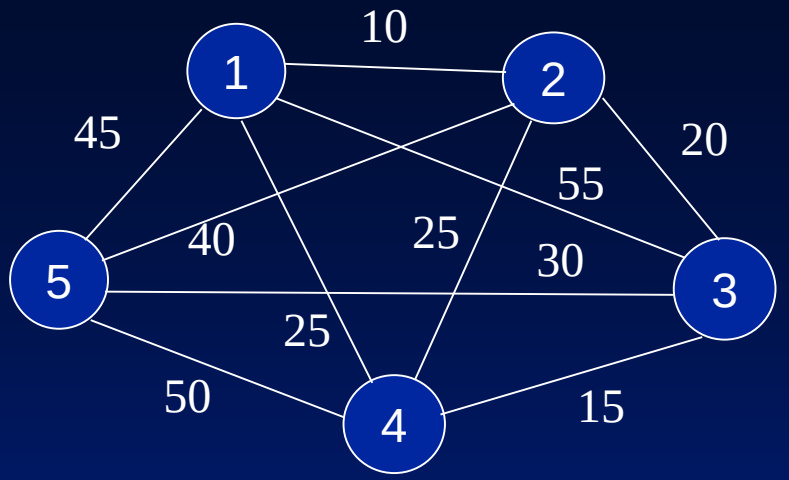


	1	2	3	4
1	-	a	-	d
2	-	-	b	-
3	-	-	-	c
4	-	e	-	-

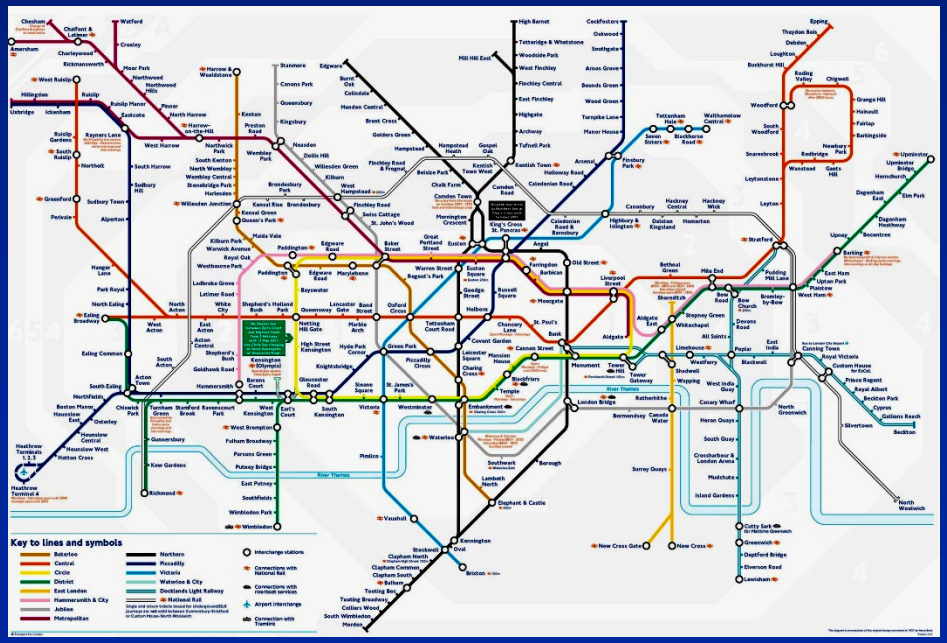
Eficiencia de la estructura Matriz de Adyacencia

Operación	Tiempo
tamano, estaVacio, remplazarElemento, intercambiar	$O(1)$
numVertices, numAristas	$O(1)$
vertices	$O(n)$
aristas, aristasDirigidas, aristasNodirigidas	$O(m)$
elementos, posiciones	$O(n + m)$
verticesFinales, opuesto, origen, destino, esDirigida, grado, gradoEnt, gradoSalida	$O(1)$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(n)$
esAdyacente	$O(1)$
aristasIncidentes, aristasIncidentesEnt, aristasIncidentesSal, verticesAdyacentes, verticesAdyacentesEnt, verticesdyacentesSal	$O(1)$
insertaVertice	$O(n^2)$
eliminaVertice	$O(n^2)$
Espacio requerido	$O(n^2)$





Recorridos en grafos



Indice

- Introducción.
- Búsqueda primero en profundidad.
- Búsqueda primero en anchura.
- Usos de los recorridos.
  - Digrafos acíclicos.
  - Orden topológico.

Introducción

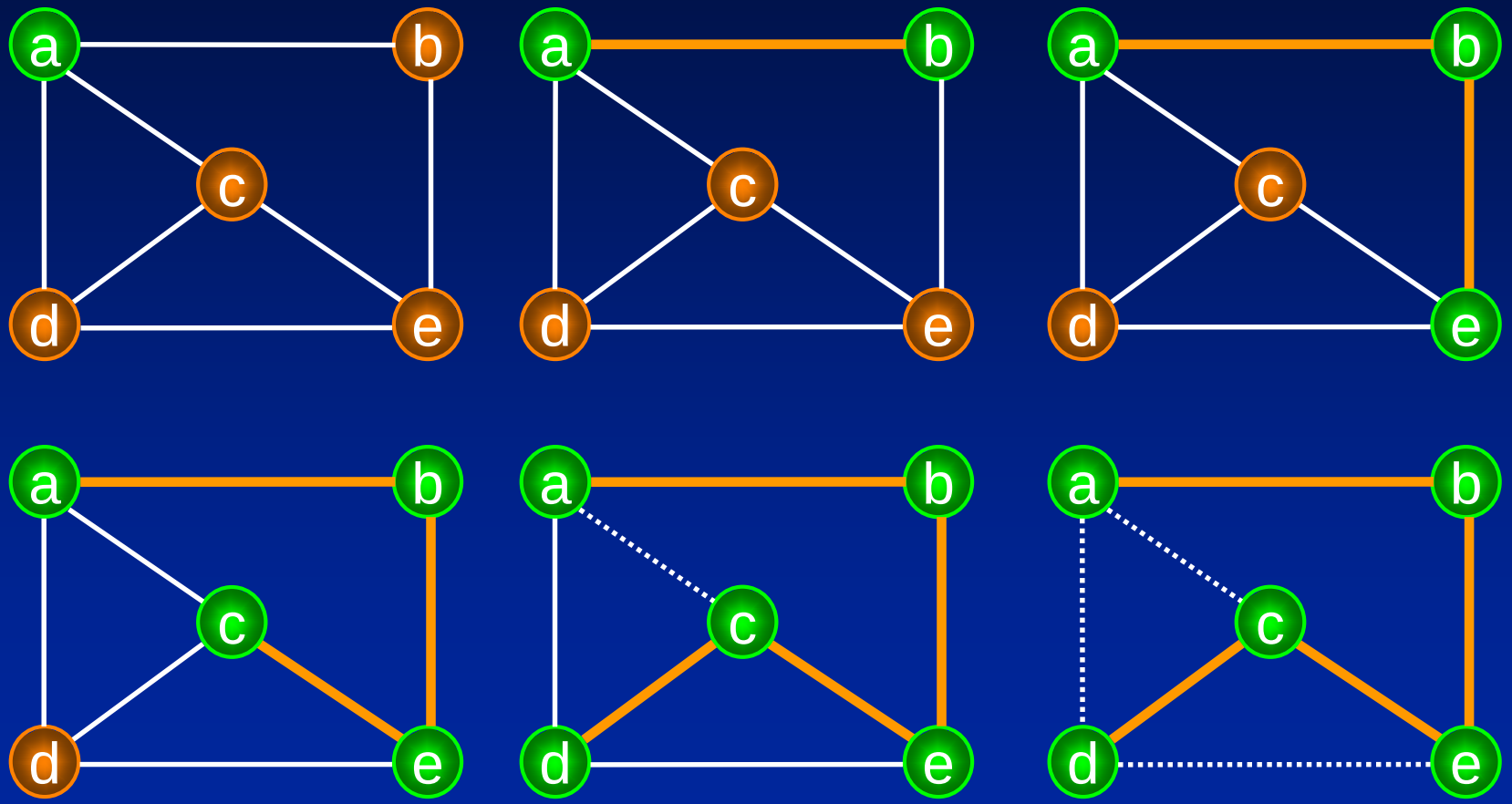
- *Recorrido*: procedimiento sistemático de exploración de un grafo mediante el examen de todos sus vértices y aristas.
- Un recorrido es eficiente si visita todos los vértices y aristas en un tiempo proporcional a su número, esto es, en tiempo lineal.

Búsqueda primero en profundidad

- *Analogía*: deambular en un laberinto con una cuerda y un bote de pintura, para no perderse.
  - Se selecciona un vértice inicial s de G, al que se fija un extremo de la cuerda y se pinta s como "visitado". Se asigna s a u (vértice en curso).
  - Se recorre G considerando una arista arbitraria (u, v). Si la arista (u, v) conduce a una vértice v ya visitado se retorna inmediatamente al vértice u.
  - Si (u, v) conduce a un vértice no visitado v, entonces se extiende la cuerda y se fija en v. Se pinta v como "visitado" y se asigna a u (vértice en curso), repitiendo el mismo procedimiento anterior.
  - Si todas las aristas incidentes a un vértice conducen a vértices ya visitados, se enrolla la cuerda vuelta atrás a la arista que condujo a u y se repite el procedimiento anterior para las aristas incidentes que no se han recorrido antes.
  - El proceso termina cuando la vuelta atrás conduce al vértice inicial s y no hay más aristas incidentes sin explorar desde s.

Búsqueda primero en profundidad

- *Animación*



Búsqueda primero en profundidad

- El recorrido BPP es una generalización del recorrido preorden de un árbol.
- Se pueden identificar cuatro tipos de aristas durante el recorrido:
  - *Aristas de descubrimiento*: son aquellas aristas que conducen al descubrimiento de nuevos vértices. También se les llama *aristas de árbol*.
  - *Aristas de retorno*: son las aristas que conducen a vértices antecesores ya visitados en el árbol.
  - *Aristas de avance*: son las aristas que conducen a vértices descendientes en el árbol.
  - *Aristas de cruce*: son aristas que conducen a un vértice que no es ancestro de otro o a vértices que están en diferentes árboles.
- Las aristas de descubrimiento forman un árbol de cubrimiento de los componentes conectados del vértice inicial s.

Algoritmo recursivo BPP

```
recorre_grafo_bpp()
{
  for cada vértice v
    marca[v]=SINVISITAR;
  for cada vértice v
    if (marca[v]==SINVISITAR)
          BPP(v);
}
```

```
BPP(u)
{
  marca[u]=VISITADO;
  for cada vértice v adyacente a u
    if (marca[v]==SINVISITAR)
          BPP(v);
}
```

- Orden de complejidad del recorrido en profundidad:
  - Con lista de adyacencia, se recorre cada elemento de lista una vez, $O(n + e)$.
  - Con matriz de adyacencia, para cada nodo se buscan sus adyacentes, $O(n^2)$.

Algoritmo recursivo BPP

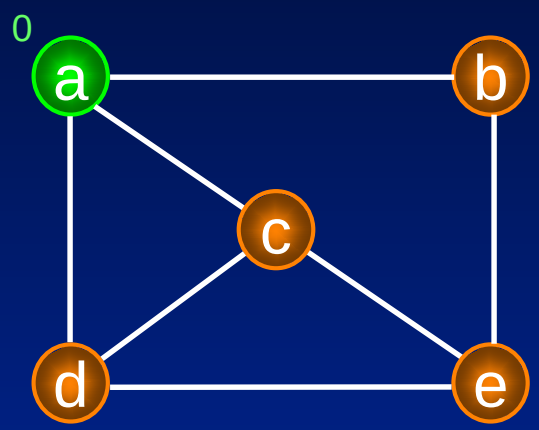
- Animación del recorrido BPP

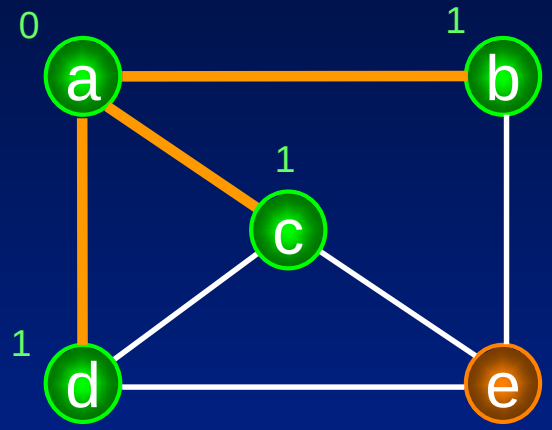
Búsqueda primero en anchura

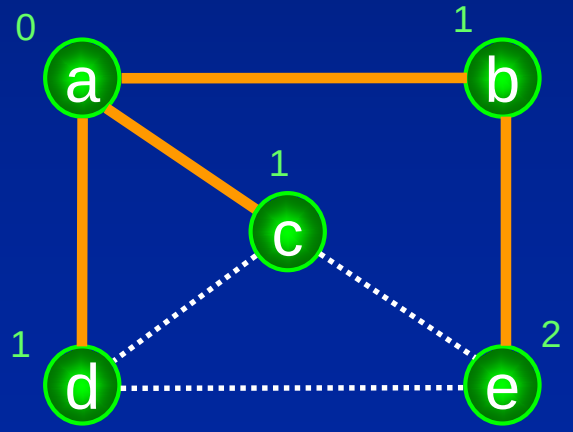
- *Analogía*: deambular en un laberinto con una cuerda y un bote de pintura, para no perderse.
  - Se selecciona un vértice inicial s de G, al que se fija inicialmente un extremo de la cuerda y se marca s con el nivel 0.
  - Se ajusta la longitud de la cuerda igual al de una arista. Se visitan y marcan con 1 todos los vértices adyacentes a s que se alcanzan con esa longitud.
  - Se repite el proceso anterior con una longitud de cuerda igual al de dos aristas. Todos los vértices adyacentes al nivel 1 se marcan con el nivel 2.
  - El recorrido termina cuando todos los vértices tienen asignado un nivel.

Búsqueda primero en anchura

- *Animación*







Búsqueda primero en anchura

- El recorrido BPA es una generalización del recorrido por niveles de un árbol.
- Se pueden identificar dos tipos de aristas durante el recorrido:
  - *Aristas de descubrimiento*: son aquellas aristas que conducen al descubrimiento de nuevos vértices.
  - *Aristas de cruce*: son aristas que conducen a un vértice ya visitado.
- Las aristas de descubrimiento forman un árbol de cubrimiento de los componentes conectados del vértice inicial s.

Algoritmo BPA

```
recorre_grafo_bpa()
{
  for cada vértice v
    marca[v]=SINVISITAR;
  for cada vértice v
    if (marca[v]==SINVISITAR)
          BPA(v);
}
```

```
BPA(v)
{
  marca[v] = VISITADO;
  InsertaCola(v, C)
  while not EsVacíaCola(C) {
    u = SuprimirCola(C);
    for cada nodo y adyacente a u {
          if (marca[y]==SINVISITAR) {
                marca[y] = VISITADO;
                InsertaCola(y, C);
          }
    }
  }
}
```

- Orden de complejidad del recorrido en anchura:
  - Con lista de adyacencia: O(n + e).
  - Con matriz de adyacencia: O(n^2).

Recorrido BPA

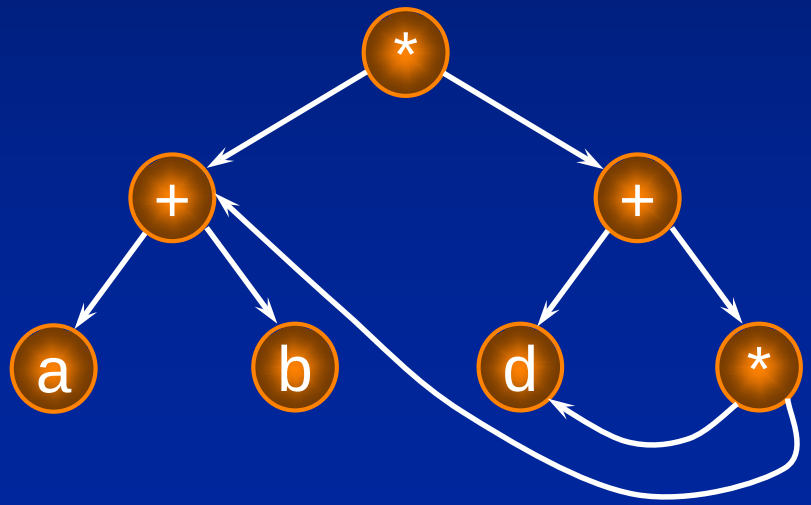
- Animación del recorrido BPA

Usos de los Recorridos

- Ambos recorridos se pueden usar para resolver los siguientes problemas:
  - Probar que G es conectado.
  - Obtener un árbol de expansión de G.
  - Obtener los componentes conectados de G.
  - Obtener un camino entre dos vértices dados de G, o indicar que no existe tal camino.
- El recorrido BPP se usa para:
  - Obtener un ciclo en G, o indicar que G no tiene ciclos.
- El recorrido BPA se usa para:
  - Obtener para cada vértice v de G, el número mínimo de aristas de cualquier camino entre s y v.

Digrafos acíclicos

- Es un grafo dirigido que no tiene ciclos.
- Representan relaciones más generales que los árboles pero menos generales que los digrafos.
- *Ejemplo*: representar estructuras sintácticas de expresiones aritméticas con subexpresiones comunes y el orden parcial de un conjunto.



(a+b)*(d+d*(a+b))

Orden parcial R en un conjunto S, relación binaria que cumple:

- ∀ elemento a de S, (a R a) es falso.
- ∀ a, b, c de S, si (a R b) y (b R c) entonces (a R c).

Digrafos acíclicos

- Un grafo es acíclico si durante un recorrido BPP no existen aristas de vuelta atrás o retorno.
- Algoritmo: recorrer el digrafo usando BPP y numerando los nodos nuevos en el recorrido. Si en algún momento en una arista de retorno un nodo descendiente tiene un nivel de profundidad menor que el antecesor, entonces existe un ciclo.

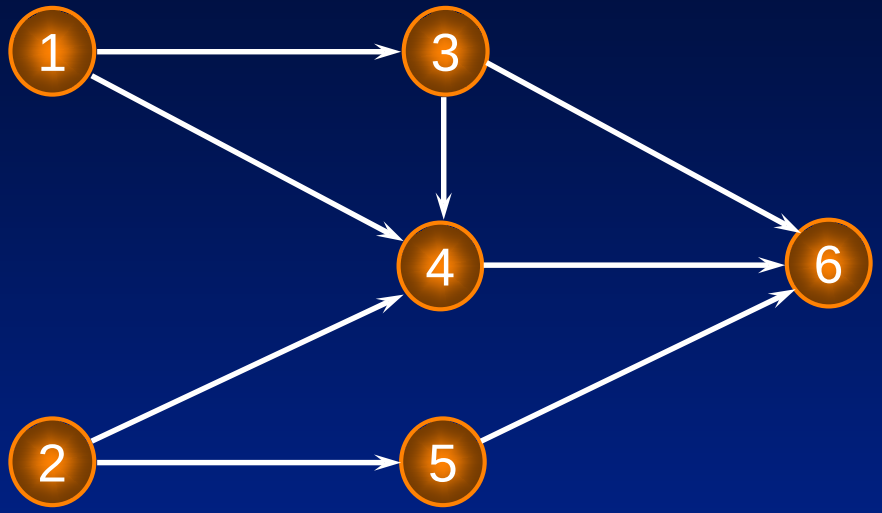
Orden topológico

- *Ordenamiento topológico de un digrafo acíclico*: orden lineal de los vértices colocándolos a lo largo de una línea horizontal de tal manera que todas las aristas tengan una dirección de izquierda a derecha.
- Ejemplo: las tareas de un proyecto de construcción.
- Algoritmo: usar una versión modificada de BPP.

```
orden_topologico(v)       /* orden inverso */
{
  marca[v]=VISITADO;
  for cada vértice w en lista_adyacencia(v)
        if (marca[w]==SINVISITAR)
              orden_topologico(w);
  imprime(v);
}
```

Orden topológico

- *Ejemplo*

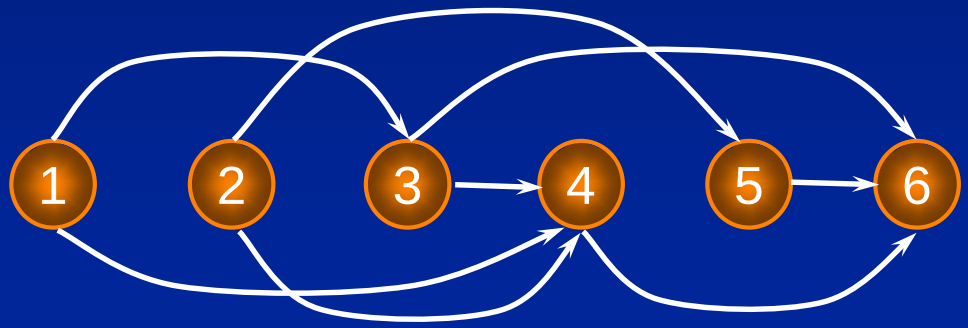


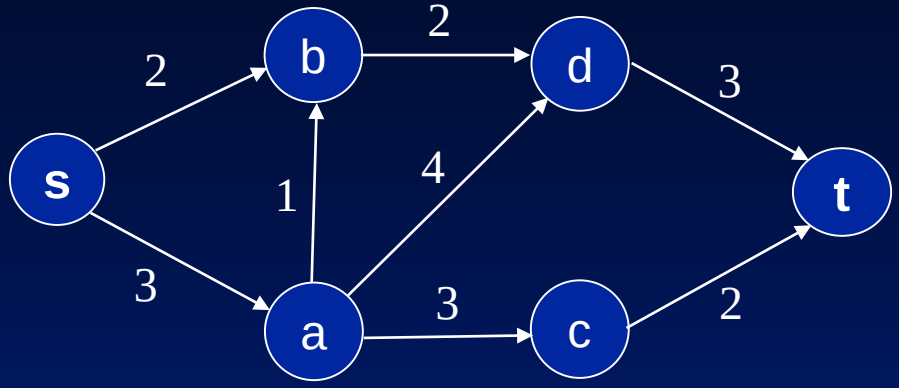
- *Orden topológico:*

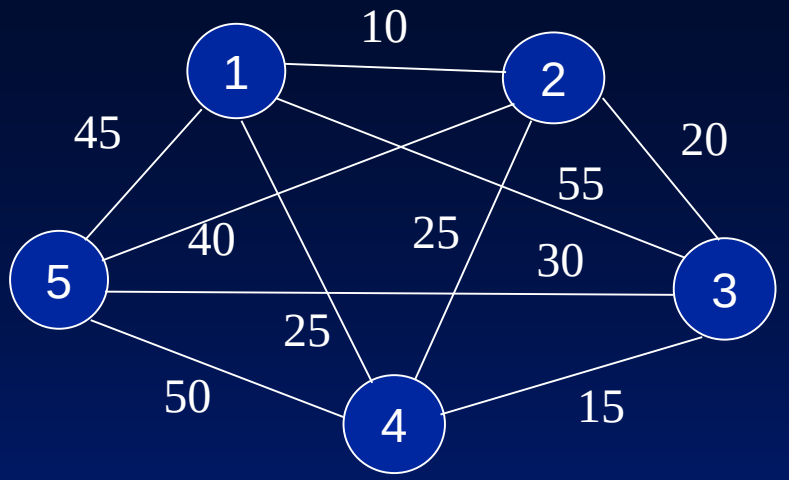
1 2 3 4 5 6

1 3 2 4 5 6

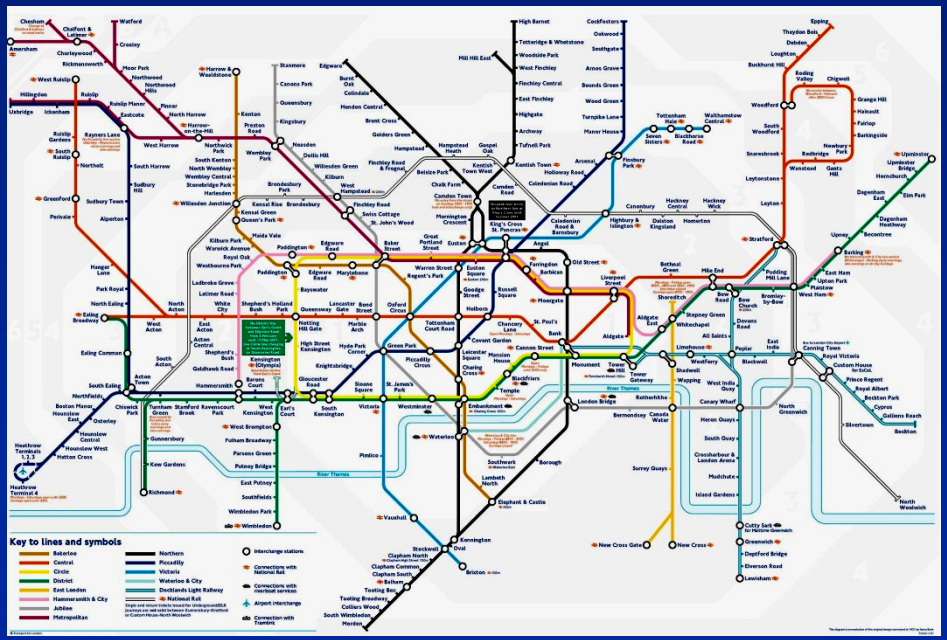
2 1 5 3 4 6







Algoritmos de caminos más cortos

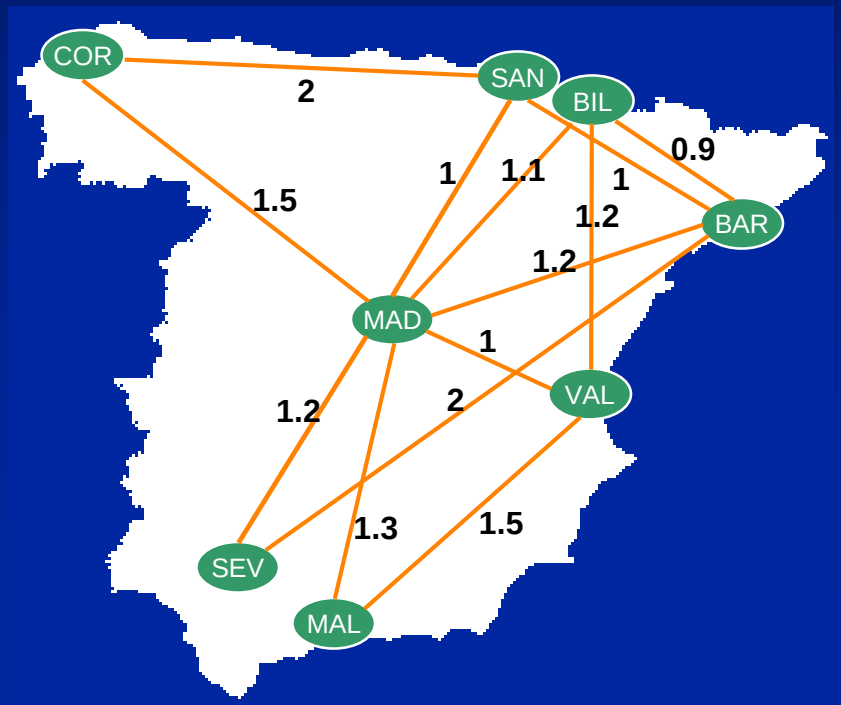


Indice

- El problema de los caminos más cortos desde un vértice.
  - Algoritmo de Dijkstra.
- El problema de los caminos más cortos entre todos los pares de vértices.
  - Algoritmo de Floyd.
- Cierre transitivo.
  - Algoritmo de Warshall

Grafos con pesos

- Cada arista lleva asociado un valor numérico *no negativo*, *w*(*e*), que representar un costo que varía linealmente a lo largo de la arista (distancia, tiempo).
- El costo de un camino es la suma de los costos de las aristas del camino.
- *Ejemplo*: grafo que representa las rutas entre ciudades de una aerolínea. El peso de las aristas es el tiempo de vuelo.



Caminos más cortos

- El recorrido BPA halla los caminos con el menor número de aristas desde el vértice inicial. Por tanto, BPA halla los caminos más cortos asumiendo que las aristas tienen el mismo peso.
- En muchas aplicaciones (p.e. redes de transporte) las aristas tienen peso diferentes.
- *Problema*: Hallar los caminos de peso total mínimo desde un vértice determinado (fuente) a todos los demás vértices.

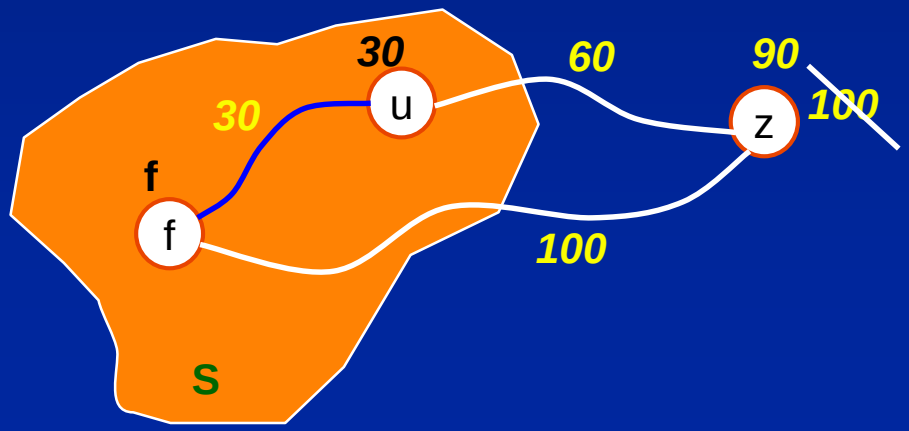
Algoritmo de Dijkstra

- La idea principal es realizar una búsqueda a lo ancho "ponderada" empezando por el vértice inicial *f*.
- De manera iterativa se construye un conjunto de vértices seleccionados S que se toman del conjunto de vértices candidatos C según el menor peso (distancia) desde *f*.
- El algoritmo termina cuando no hay más vértices de G fuera del conjunto formado.
- El paradigma usado corresponde al ***método voraz***, en el que se trata de optimizar una función sobre una colección de objetos (menor peso).
- Se usa un vector d[*v*] para almacenar la distancia de *v* a *f*.

Algoritmo de Dijkstra

- Cuando se añade un vértice al conjunto S, el valor de d[*v*] contiene la distancia de *f* a *v*.
- Cuando se añade un nuevo vértice *u* al conjunto S, es necesario comprobar si *u* es una mejor ruta para sus vértices adyacentes *z* que están en el conjunto C.
- Para ello se actualiza d con la ***relajación*** de la arista (*u*, *z*):

$$d[z] = \min(d[z], d[u] + w[u, z])$$



Algoritmo de Dijkstra

Dijkstra(G, f)

S = { *f* }, C = { V }

d[*f*] = 0

d[*u*] = ∞ $\quad\quad$ $\forall$u $\neq$ *f*

while (C $\neq \varnothing$) {

  seleccionar vértice $w \in$ C / d[w] es mínimo

  S = S $\cup$ {w}, C = C - {w}

  for cada vertex $v \in$ Adyacente[w] {

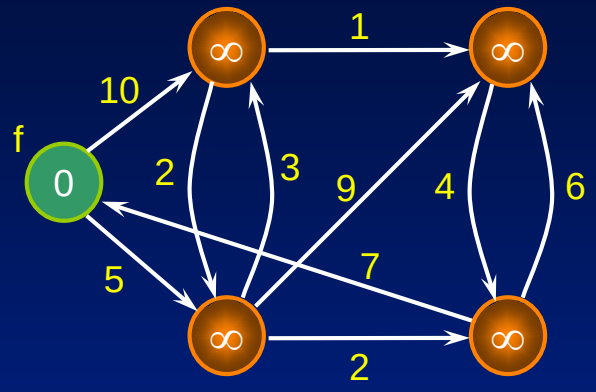
    if (d[v] > d[w] + w(w, v))

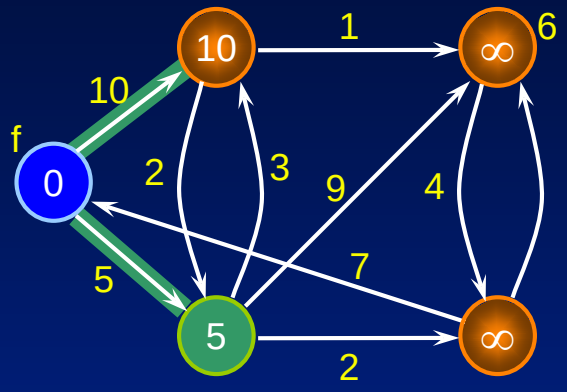
      d[v] = d[w] + w(w, v)

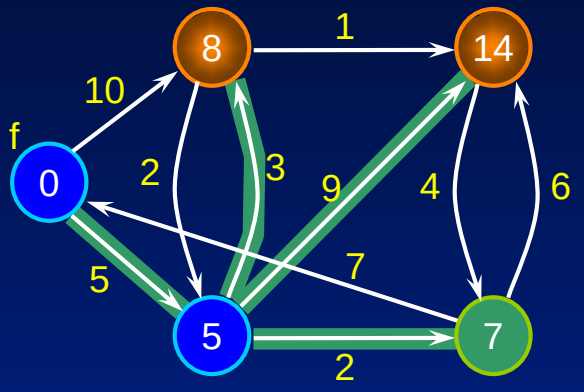
  }

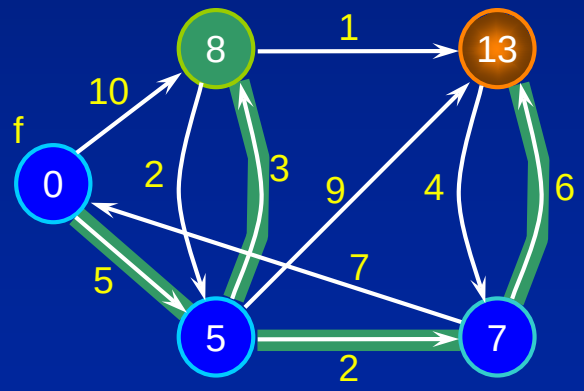
}

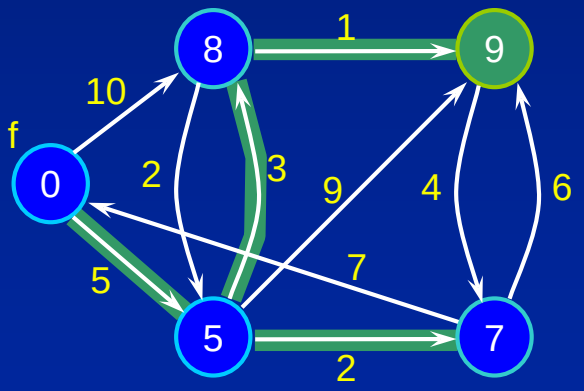
Algoritmo de Dijkstra

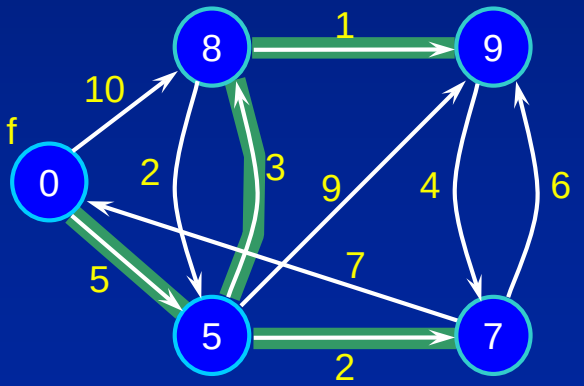








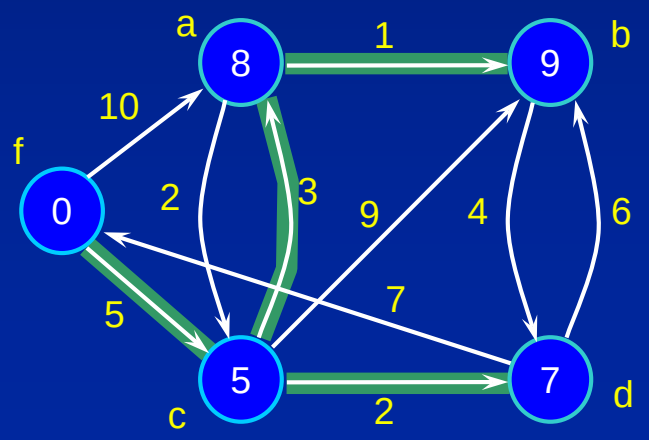




Algoritmo de Dijkstra

- Para reconstruir los vértices del camino más corto desde el vértice fuente a cada vértice:
  - Se usa otro array p[v] que contiene el vértice anterior a v en el camino.
  - Se inicializa p[v] = f para todo v $\neq f$
  - Se actualiza p siempre que d[v] > d[w] + w(w, v) a:

    p[v] = w.
  - El camino a cada vértice se halla mediante una traza hacia atrás en el array p.



p[a] = c p[b] = a p[c] = f p[d] = c

Algoritmo de Dijkstra

- Complejidad:
  - Con matrices de adyacencia: $O(n^2)$.
  - Con listas de adyacencia: $O(n^2)$.
  - Se puede usar estructuras de datos más eficientes, como una cola de prioridad para buscar el vértice con menor d[v] se consigue O(e*log n).

Ejemplo de aplicación Algoritmo de Dijkstra

- App para tiempo mínimo de viaje en avión
  - Determinación de la ruta más corta en tiempo del vuelo entre ciudades de EEUU.
  - La diferencia horaria entre las ciudades puede llegar a cuatro horas.
  - Fichero de vuelos entre ciudades.
  - Se toma la hora GMT para el cálculo del tiempo de vuelo.

Animación del Algoritmo de Dijkstra

- Animación del algoritmo de Dijkstra

Caminos más cortos

- *Problema*: Hallar los caminos mínimos entre cualquier par de nodos de un grafo.
- Se puede usar el algoritmo de Dijkstra tomando cada vértice como fuente.
- Existe una manera más directa: *algoritmo de Floyd*.

Algoritmo de Floyd

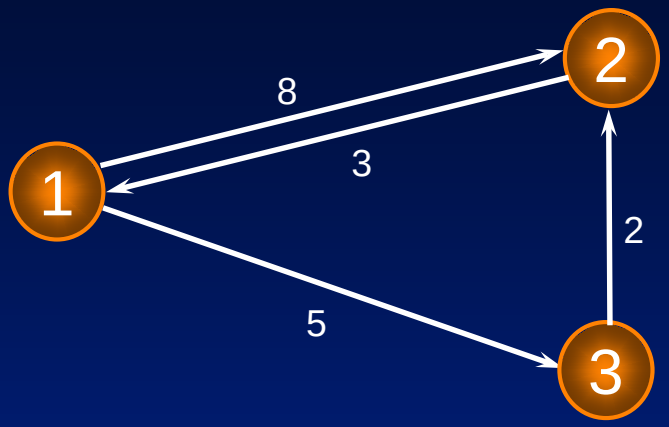
- Utiliza una matriz $A_k[i][j]$, que contiene el camino más corto que pasa por los primeros k primeros vértices.
- Inicialmente $A_k[i][j] = C[i][j] \ \forall \ i \neq j$. Si no hay arista de i a j $C[i][j] = \infty$ y los elementos diagonales se ponen a 0.
- En la iteración k (nodo k como pivote) se calcula, para cada camino de v a w, si es más corto pasando por k aplicando:

  $A_k[i][j] = \min (A_{k-1}[i][j] , A_{k-1}[i][k] + A_{k-1}[k][j]), \ \forall \ i \neq j.$
- Como no varía la fila y la columna k en la iteración k, sólo es necesario una matriz A.

Algoritmo de Floyd

```
Floyd (n, C, A)
{
    A[i][j] = C[i][j]
    A[i][j] = 0
    for (k =1; k <=n; k ++)
          for (i =1; i <=n; i ++)
                for (j =1; j <=n; j ++)
                      if (A[i][k] + A[k][j] < A[i][j] )
                            A[i][j] = A[i][k] + A[k][j]
}
```

Algoritmo de Floyd



C[*i*][*j*]	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

A_1[*i*][*j*]	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

A_2[*i*][*j*]	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

A_3[*i*][*j*]	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Algoritmo de Floyd

- Para obtener los caminos se procede como en Dijkstra. Se usa una matriz P[*i*][*j*] para almacenar el camino:
  - P[*i*][*j*] = 0 si el camino es directo.
  - En otro caso, si

    if (A[*i*][*k*] + A[*k*][*j*] < A[*i*][*j*])

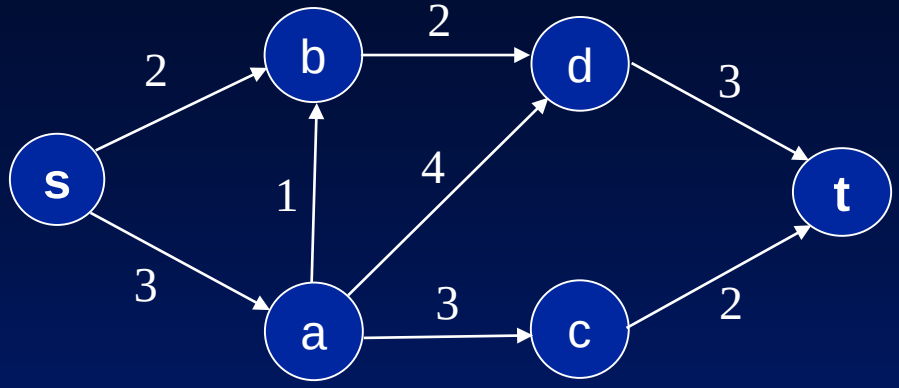
    P[*i*][*j*] = *k*
- La complejidad del algoritmo es:
  - Con matriz de adyacencia: $O(n^3)$.
  - Para grafos dispersos es mejor usar la versión Dijkstra con lista de adyacencia que toma

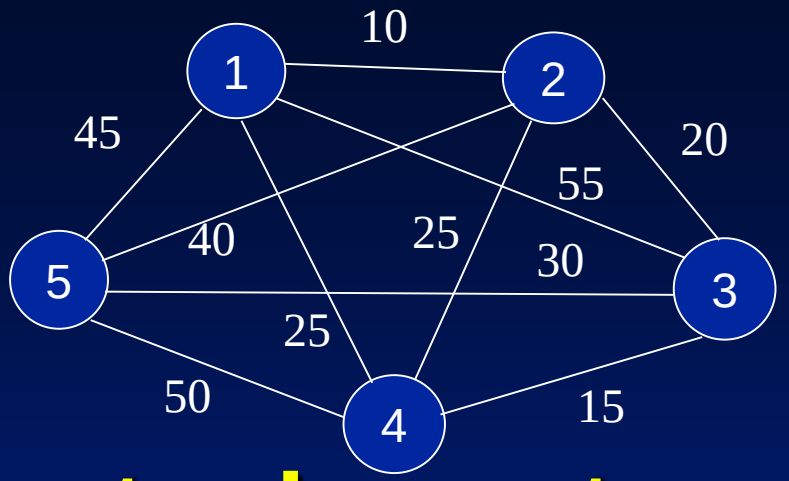
    $O(ne \log n)$.

Cierre transitivo

- Hay situaciones en las que sólo se desea determinar si existe un camino entre dos vértices.
- El algoritmo de Floyd se puede adaptar para resolver este problema; el algoritmo resultante se llama *algoritmo de Warshall*.
- Se usa la matriz de adyacencia A[i][j] = 1, si existe una arista entre los vértices i y j, sino se asigna el valor 0.
- La matriz resultante se calcula aplicando la siguiente fórmula en la k-ésima iteración sobre la matriz A que almacenará el resultado:

$$A_k[i][j] = A_{k-1}[i][j] \text{ or } (A_{k-1}[i][k] \text{ and } A_{k-1}[k][j])$$





Árbol de cubrimiento de costo mínimo

Indice

- Introducción.
- Algoritmo de Prim.
- Algoritmo de Kruskal.

Introducción

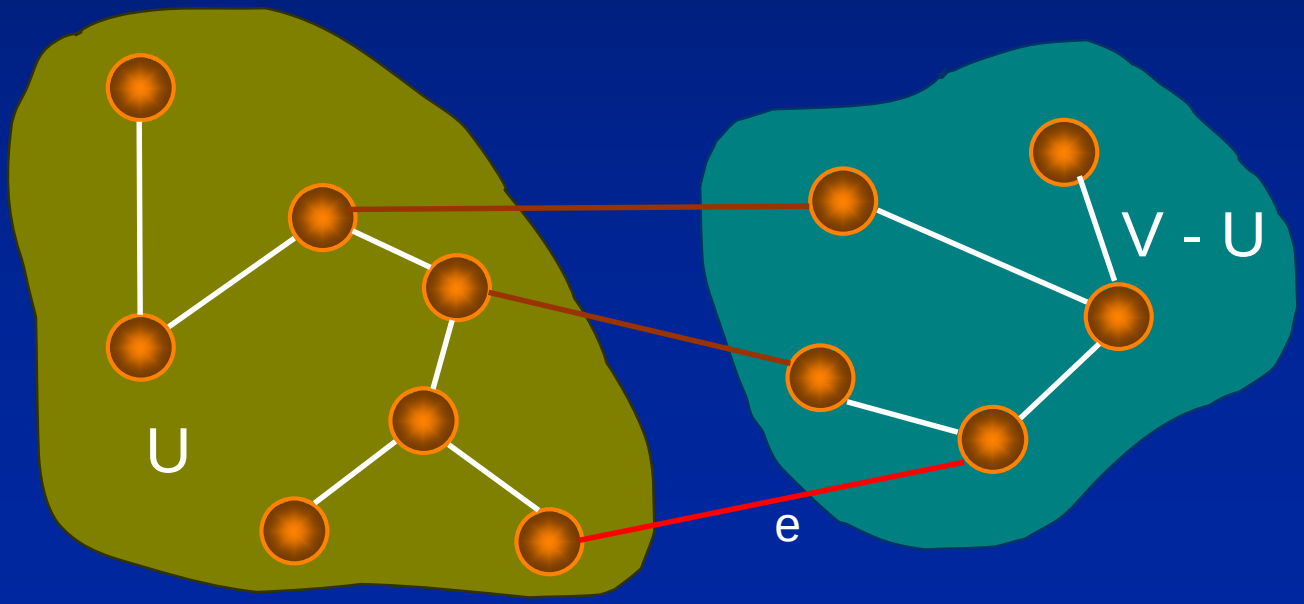
- Un *árbol de cubrimiento o expansión* para un grafo G = (V, E) no dirigido conectado con pesos es un árbol libre que conecta todos los vértices en V.
- El costo de un árbol de cubrimiento está determinado por la suma de los costos de las aristas en el árbol.
- *Problema*: hallar el árbol de cubrimiento de costo mínimo para G.
- Problema común en el planeamiento de redes de distribución y comunicación.

Introducción

- *Propiedad*: Sea G = (V, E) un grafo conectado con pesos.

  Sea U un subconjunto del conjunto de vértices V.

  Si e=(u, v) es la arista de menor costo considerando que $u \in$ U y $v \in$ V-U, entonces hay un árbol de cubrimiento mínimo que incluye (u, v) como arista.



Introducción

- Algoritmos comunes para resolver el problema:
  - Prim
  - Kruskal
- Ambos algoritmos
  - utilizan la propiedad anterior.
  - son de tipo voraz: se selecciona uno de los candidatos con el criterio que es mejor en cada momento (menor costo).

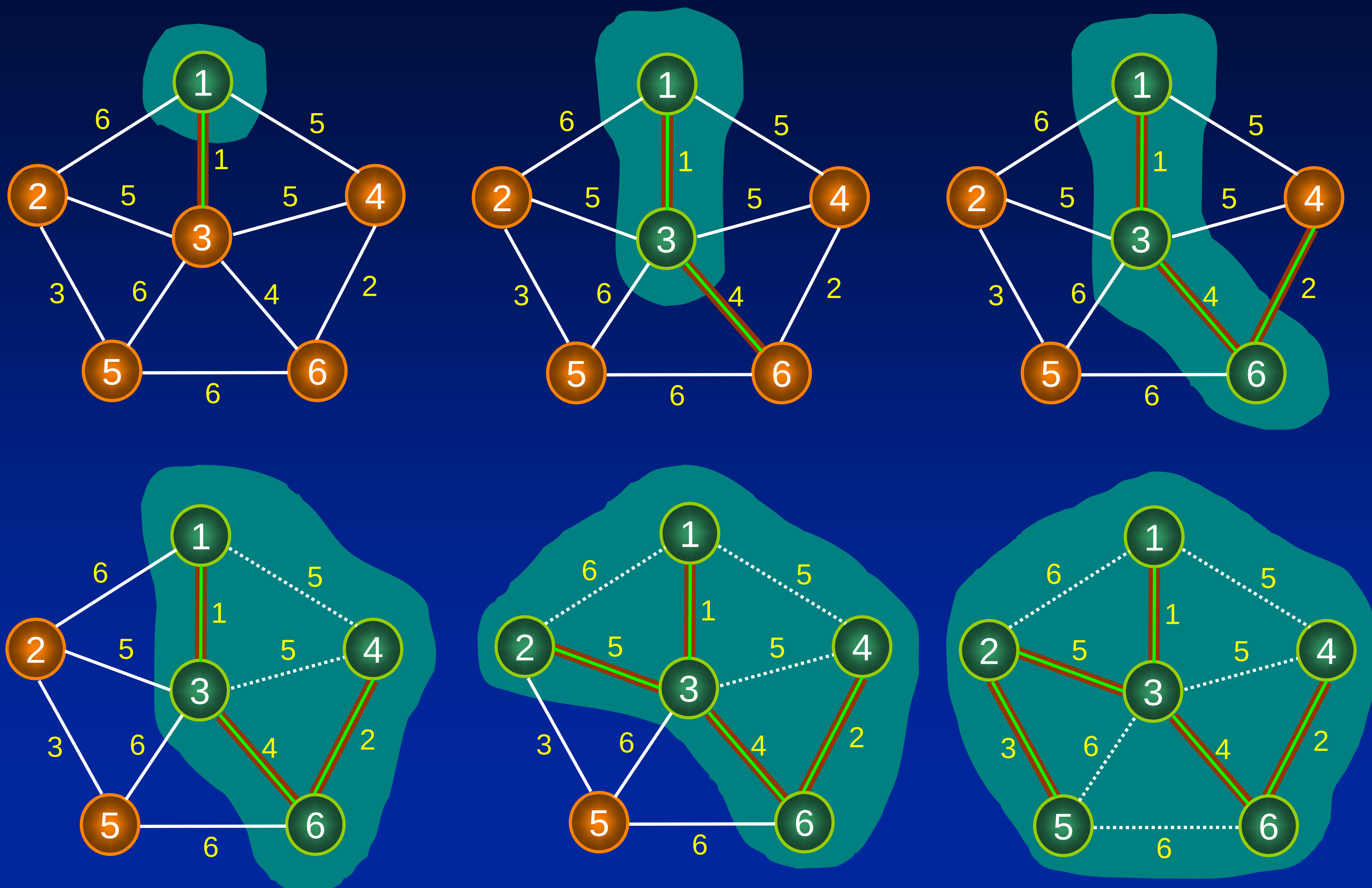
Algoritmo de Prim

- Similar al algoritmo de Dijkstra.
- Aumenta el árbol T un vértice cada vez.
- El array d[*v*] contiene el menor costo de la arista que conecta *v* con el árbol.
- Tiene una complejidad $O(n^2)$.

Algoritmo de Prim

```
Prim (G, T )
{
    T = ∅
    U = {1}
    while U ≠ V
    {
      seleccionar la arista (u, v) de menor costo
           tal que u ∈ U y v ∈ V-U
      T=T ∪ {(u, v)}
      U=U ∪ {v}
    }
}
```

Algoritmo de Prim



Costo Total = 15

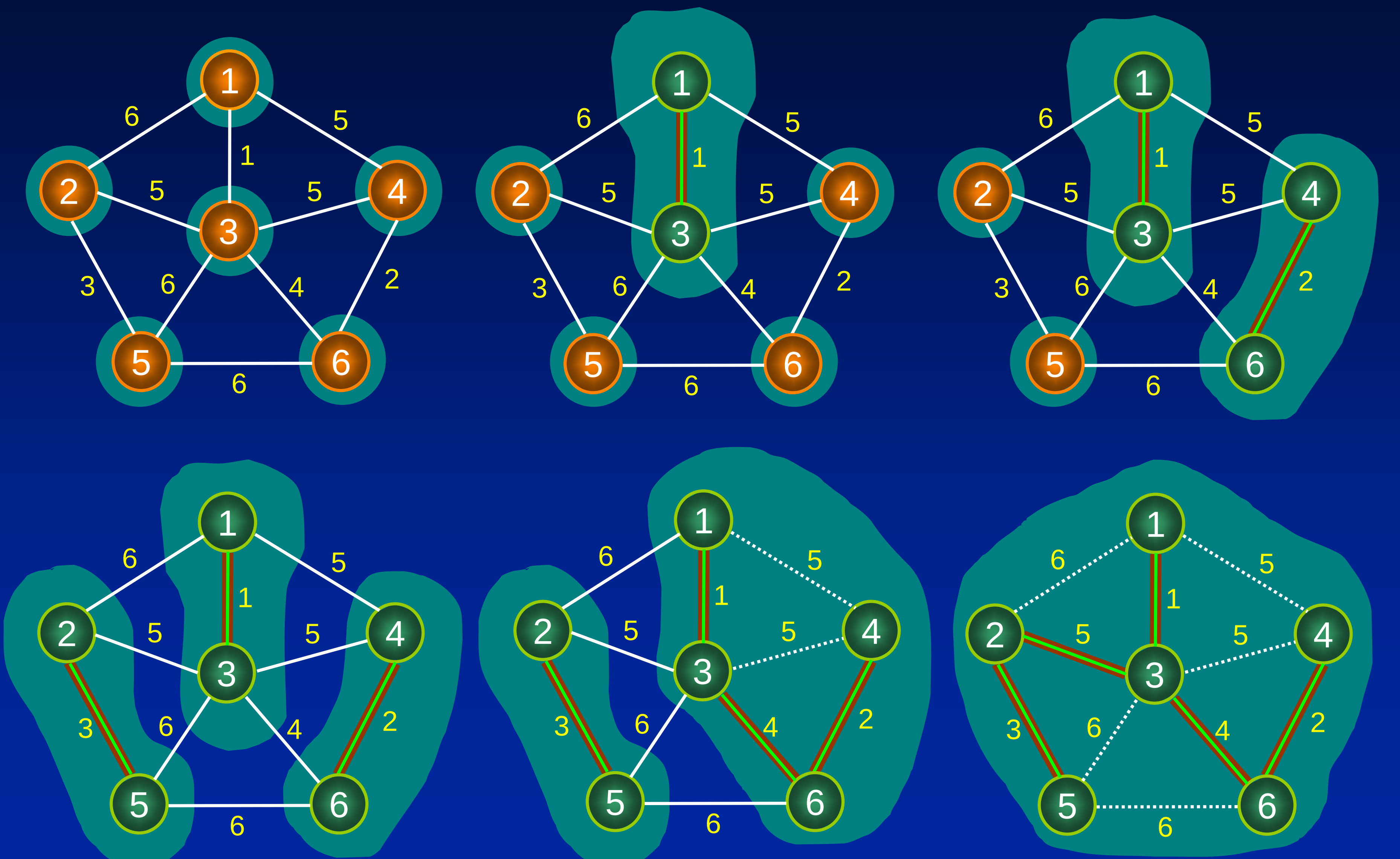
Algoritmo de Kruskal

- Añade una arista cada vez por orden de peso.
- Acepta una arista si no produce un ciclo.
- Se implementa usando una cola de prioridad.
- Tiene una complejidad $O(e \log e)$.

Algoritmo de Kruskal

```
Kruskal (G, T)
{
    for cada vértice v en G
        C(v) = {v}          /* grupo de vértices */
    Q = cola de prioridad { (u, v) ∈ G, clave = w(u, v) }
    T = ∅
    while Q ≠ 0
    {
      Extraer de Q la arista (v, u) con menor peso
      if C(v) ≠ C(u)
          T=T ∪ { (v, u) }
          C(v) = C(v) ∪ C(u)
    }
}
```

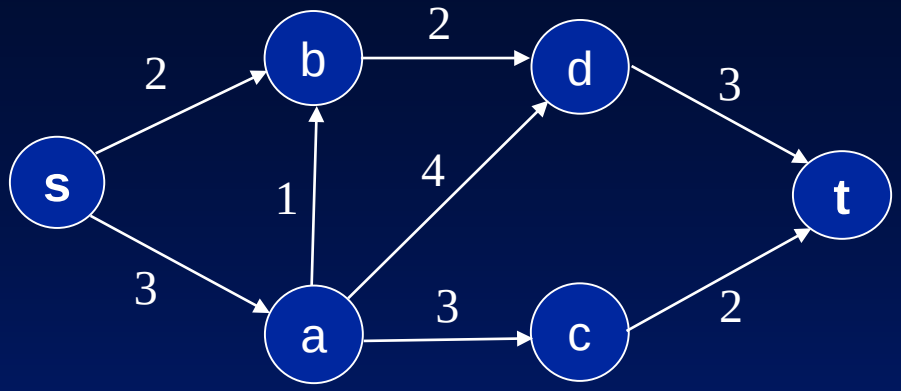
Algoritmo de Kruskal

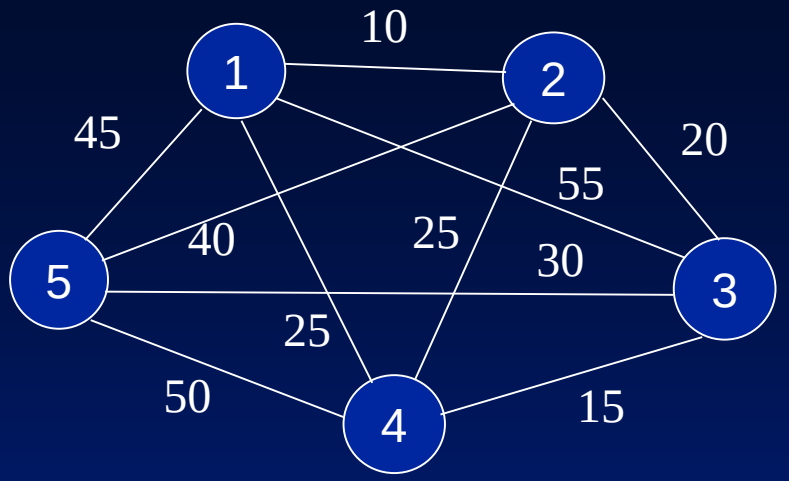


Costo Total = 15

Animación de los Algoritmos

- Árbol de cubrimiento mínimo





Flujo en Redes. Flujo máximo

Indice

- Introducción.
- Flujo en redes.
- El método de Ford Fulkerson. Flujo máximo.
- Redes residuales.
- Caminos aumentantes.
- Cortes en redes de flujos.
- Teorema de flujo-máximo mínimo-corte.
- El algoritmo de Ford Fulkerson.

Introducción

- Los digrafos se pueden usar para representar flujo en redes.
- Permiten modelar todo tipo de red, en particular las de transporte y distribución:
  - flujo de fluídos en tuberías, piezas en una línea de ensamblaje, corriente en circuitos eléctricos, información en redes de comunicación, etc.
- *Problema*: Maximizar la cantidad de flujo desde un vértice fuente a otro sumidero, sin superar las restricciones de capacidad.
  - Método de Ford-Fulkerson para resolver el problema de máximo flujo.

Redes de flujo

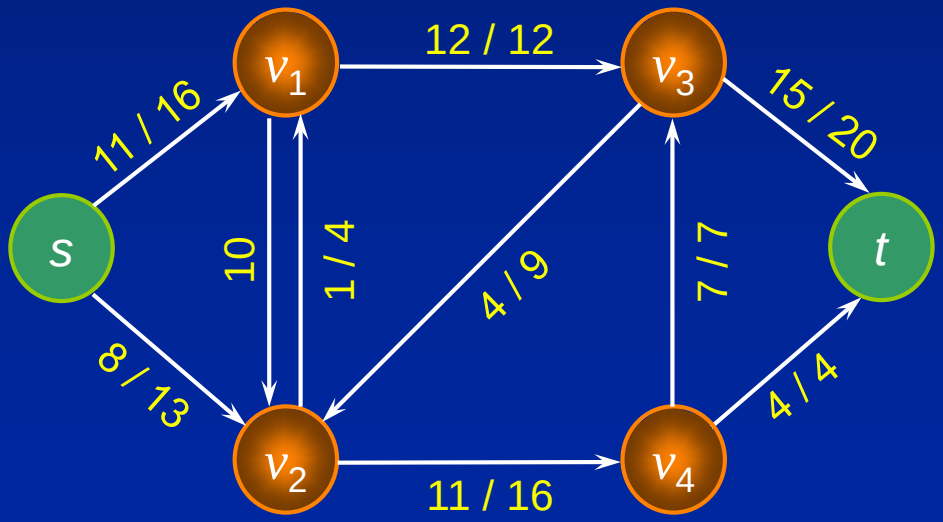
- Digrafo G=(V, E)
- Los pesos de las aristas representan capacidad (c(u, v)> 0). Si no hay aristas la capacidad es cero.
- Vértices especiales:

  fuente *s*, vértice sin aristas de entrada.

  sumidero *t*, vértice sin aristas de salida.
- El grafo es conectado: Hay un camino entre *s* y *t* por algún vértice intermedio del grafo.

Redes de flujo

- Un flujo en G es una función real $f : V \times V \rightarrow \Re$ que satisface las siguientes propiedades:
  - Restricción de capacidad: Para todo $u, v \in V$, $f(u, v) \leq c(u, v)$
  - Antisimetría: Para todo $u, v \in V$, $f(u, v) = -f(v, u)$
  - Conservación de flujo: Para todo $u \in V - \{s, t\}$, $\sum_{v \in V} f(u,v) = 0$
- Valor del flujo: $|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)$



Método de Ford-Fulkerson

- Método iterativo para resolver el problema de flujo máximo.
- Seudocódigo:

```
Método de Ford-Fulkerson (G, s, t)
Inicializar flujo f a 0
while exista un camino aumentante p {
        aumentar flujo f a través de p
}
return f
```

- El método depende de tres conceptos básicos:
  - Redes residuales.
  - Camino aumentante.
  - Cortes en redes de flujo.

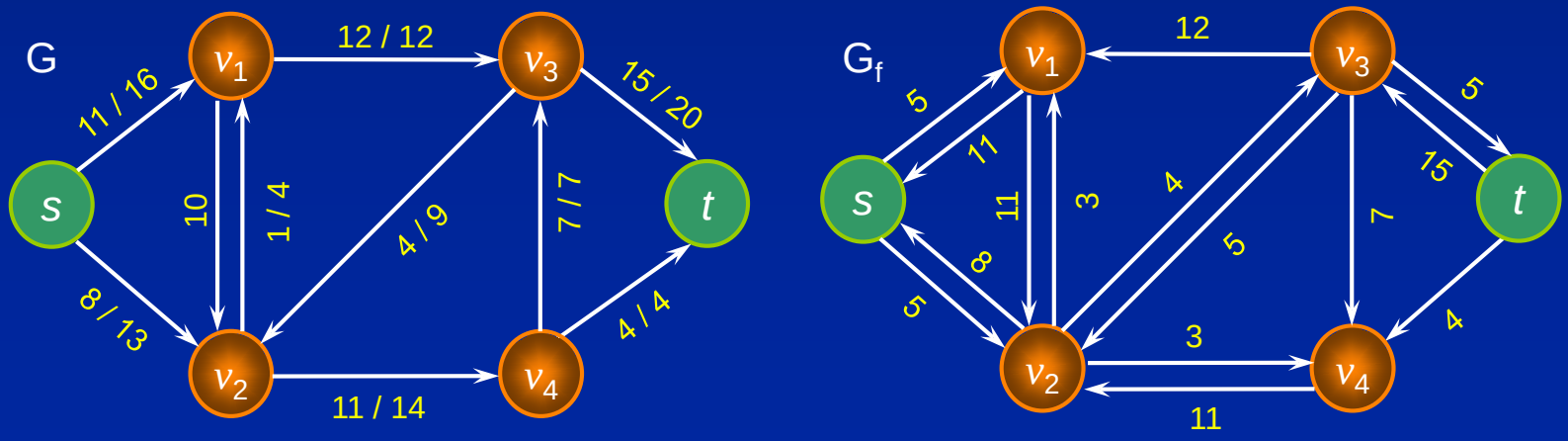
Redes residuales

- Para una red de flujo y un flujo, la red residual es el conjunto de aristas que pueden admitir más flujo.
- Sea una red de flujo G=(V, E) con fuente *s* y sumidero *t*. Sea *f* un flujo en G y un par de vértices *u*, *v* ∈ V. El flujo neto adicional desde *u* a *v* sin exceder la capacidad c(*u*, *v*) es la ***capacidad residual*** de (*u*, *v*), definida por:

$$c_f(u,v) = c(u,v) - f(u,v)$$

- La ***red residual*** de G inducida por *f* es $G_f = (V, E_f)$ donde

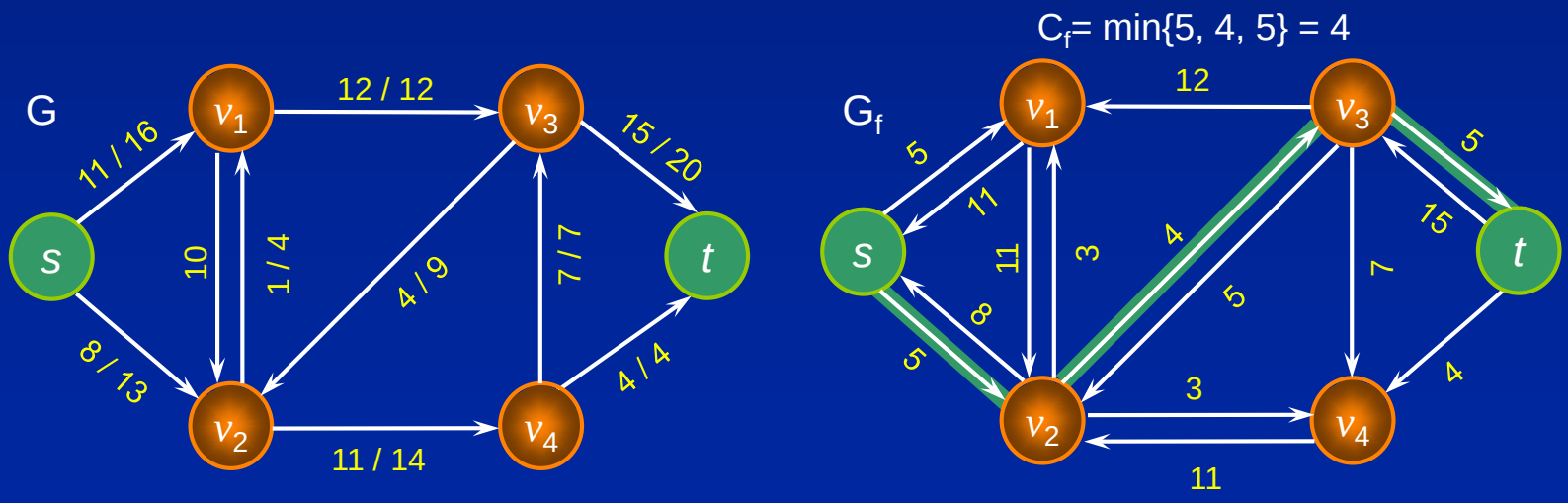
$$E_f = \{(u,v) \in V \times V : c_f(u, v) > 0\}$$



Caminos aumentantes

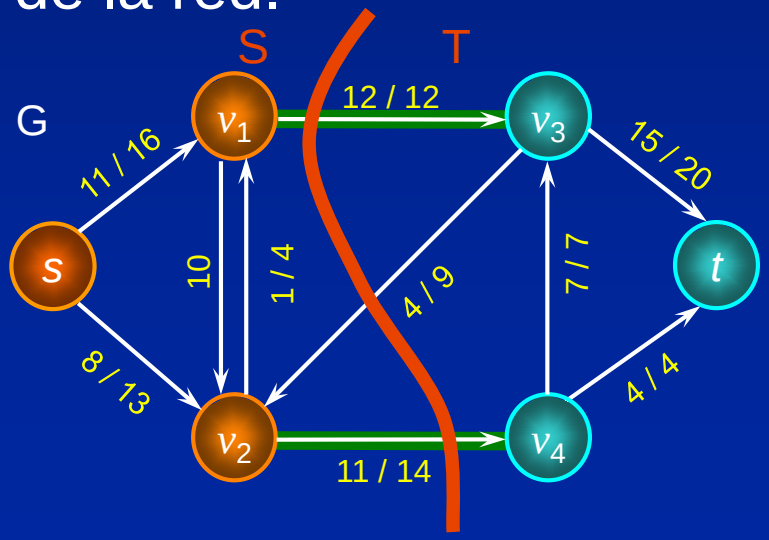
- Un camino aumentante p en una red de flujo G=(V, E) y flujo *f*, es un camino simple de *s* a *t* en la red residual G_f.
- Cada arista (*u*, *v*) del camino aumentante admite un flujo neto positivo adicional de *u* a *v* sin violar la restricción de capacidad de la arista.
- Capacidad residual: es la máxima cantidad de flujo neto que se puede enviar por las aristas de un camino aumentante. Se calcula por:

$$c_f(p) = \min\{c_f(u,v) \,/\, (u,v) \in p\}$$



Cortes en redes de flujo

- Un corte (S, T) de una red de flujo G=(V, E) es una partición del conjunto de vértices V en dos subconjuntos S y T = V–S tal que $s \in S$ y $t \in T$.
- Si *f* es un flujo:
  - *f*(S, T) es el flujo neto a través del corte (S,T).
  - c(S, T) es la capacidad del corte (S,T).
- Flujo en una red = flujo neto a través de cualquier corte de la red.



Corte = ({s, v_1, v_2}, {s, v_1, v_2})

f(s, t) = f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) =
12 + (-4) + 11 = 19

c(s, t) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26

Teorema flujo-máximo mínimo-corte

- Si *f* es un flujo en una red de flujo G = (V, E) con fuente *s* y sumidero *t*, entonces las siguientes condiciones son equivalentes:
  - *f* es un flujo máximo en G.
  - La red residual G_f no contiene caminos aumentantes.
  - | f | = c(S, T) para algún corte (S, T) de G.

Algoritmo de Ford-Fulkerson

Ford-Fulkerson (G, *s*, *t*)

for cada arista (*u*, *v*) $\in$ E[G] {

f [*u*, *v*] = 0

f [*v*, *u*] = 0

}

while exista un camino p de *s* a *t* en el grafo residual G_f {

$c_f(p) = \min\{ c_f(u, v) / (u, v) \in p\}$

for cada arista (*u*, *v*) $\in$ p

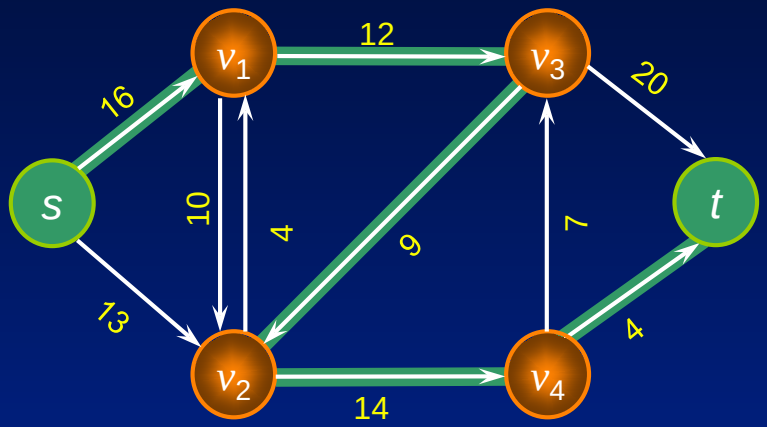
f [*u*, *v*] = *f* [*u*, *v*] + $c_f(p)$

f [*v*, *u*] = –*f* [*u*, *v*]

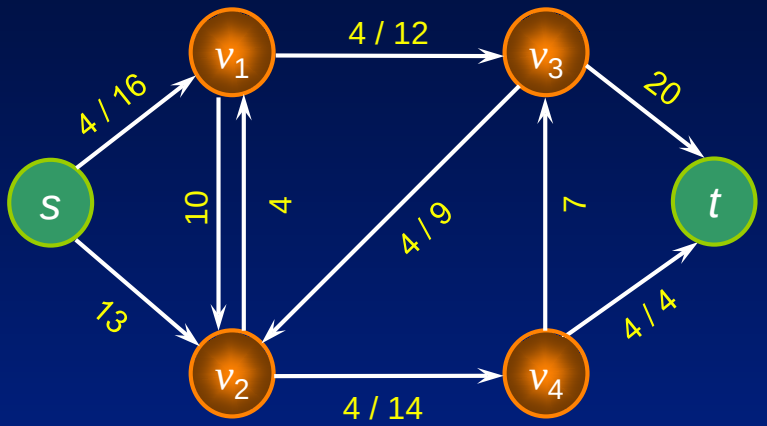
}

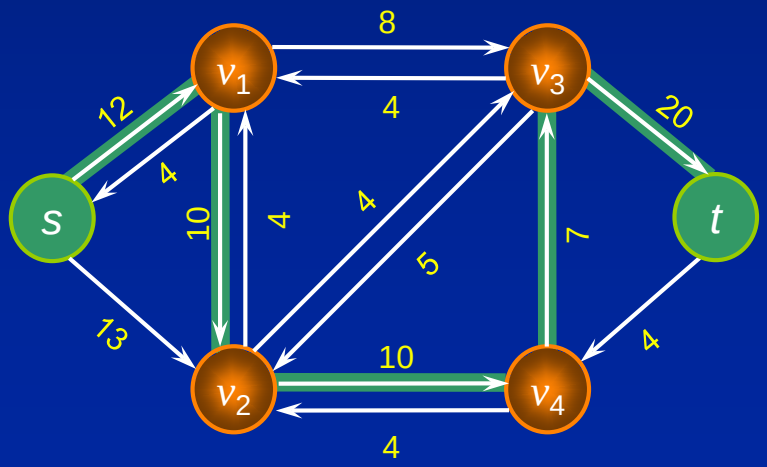
Ejemplo

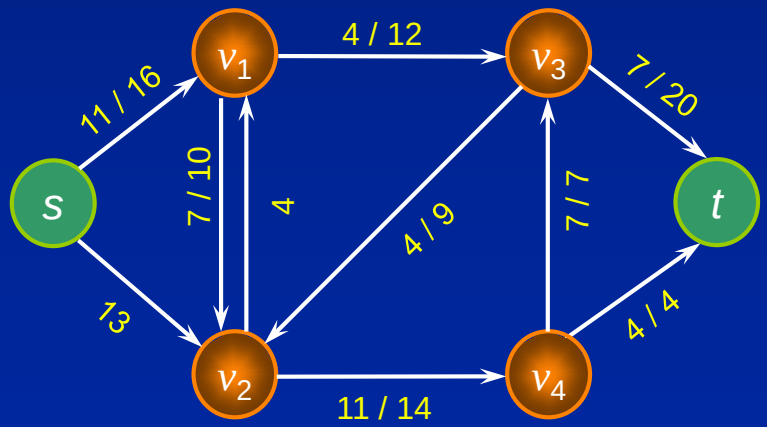
Grafo Residual



Flujo

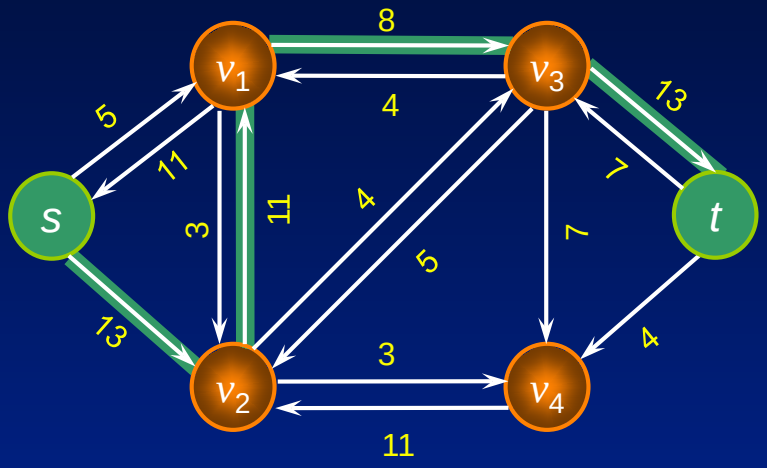




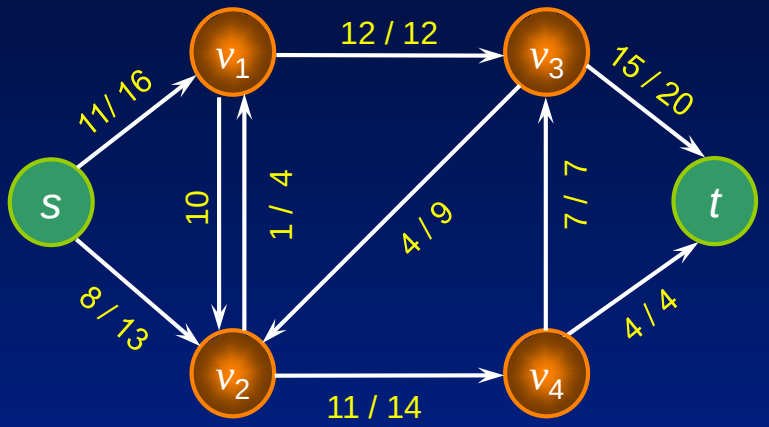


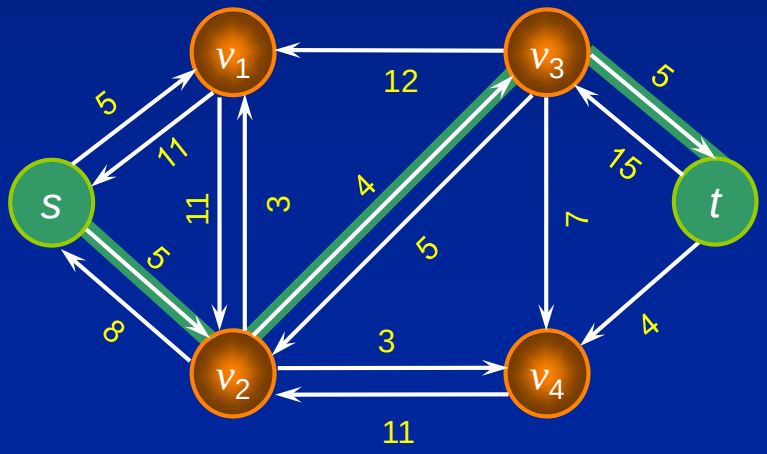
Ejemplo

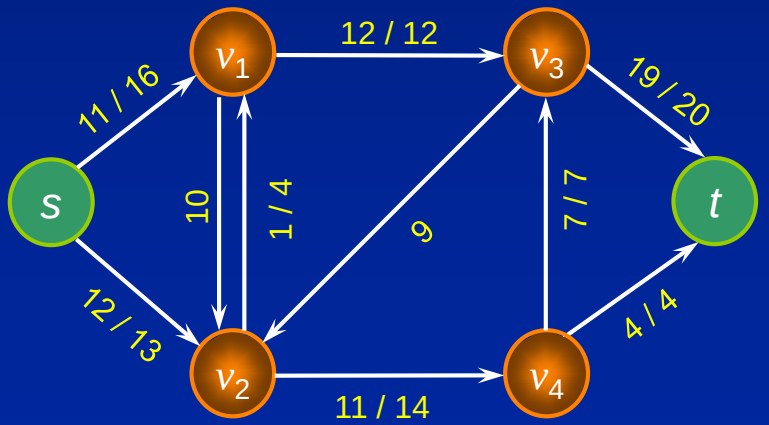
Grafo Residual



Flujo

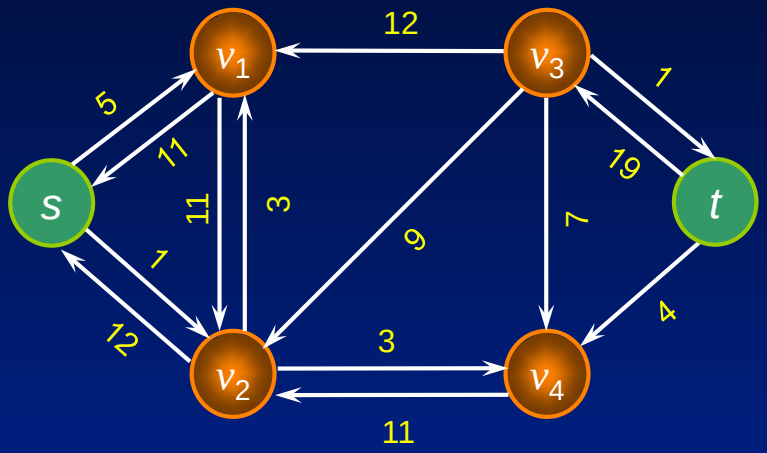




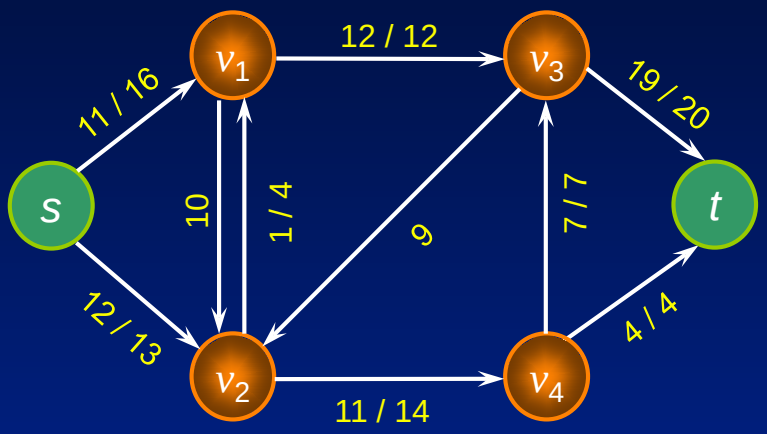


Ejemplo y complejidad

Grafo Residual



Flujo



- Para hallar el camino aumentante se puede usar cualquier tipo de recorrido (BPA o BPP).
- La capacidad de cada arista se puede multiplicar por un factor de escala para conseguir que sea entera.
- Bajo estas condiciones el algoritmo tiene una complejidad de O(E| f^* |), donde f^* es el máximo flujo obtenido por el algoritmo.

Animación del algoritmo flujo máximo

Algoritmo de flujo máximo