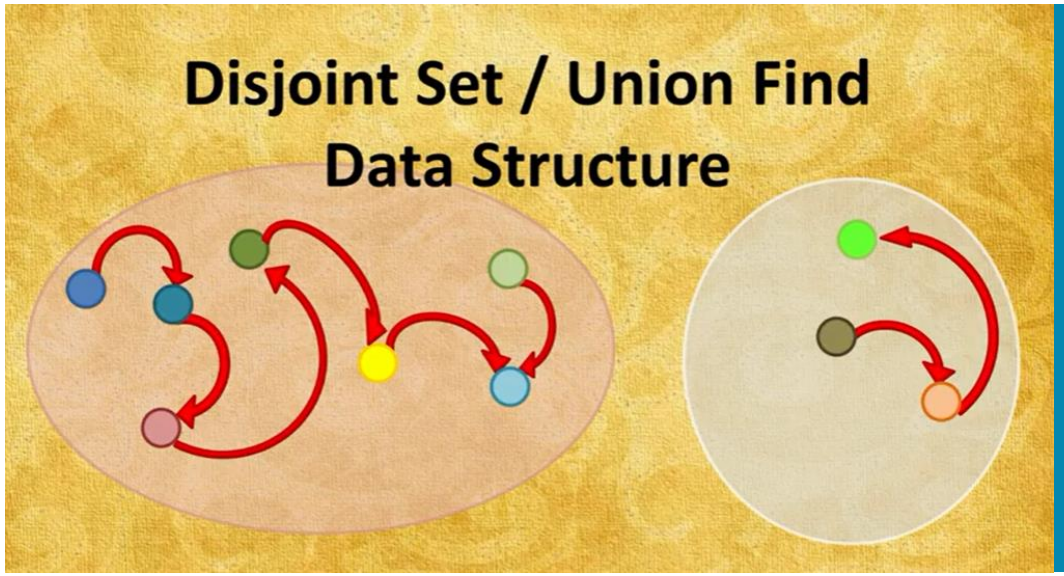


Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 9: Estructura de datos para conjuntos disjuntos (UFDS)

Complejidad Algorítmica

Semana 9 / Sesión 1

MÓDULO 9: Estructura de datos para conjuntos disjuntos (UFDS)

Contenido

1. Definición de Union-Find Disjoint Sets (UFDS)
2. Algoritmos o Estrategias Union-Find
   - 2.1. Quick-Find
   - 2.2. Quick-Union
   - 2.3. Quick-Union Ponderado
3. Aplicaciones de UFDS

Preguntas

1. Definición de UFDS

¿Qué es UFDS?

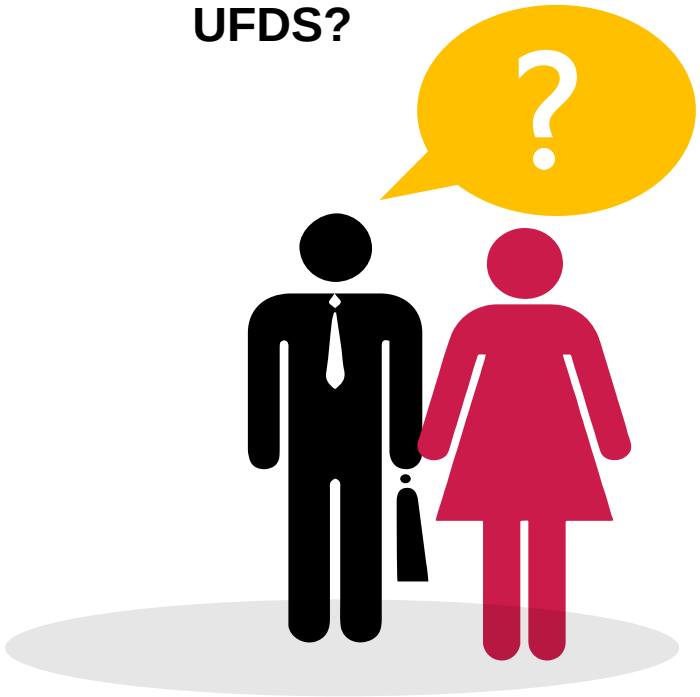
- Al hacer búsquedas y resolver problemas en grafos, hemos utilizado previamente técnicas como DFS o BFS. Ahora **UFDS** la utilizaremos para resolver un problema muy específico: **los componentes conectados**.
- No confundir con la detección de los componentes fuertemente conectados, que se resuelven con algoritmos DFS más complejos (algoritmo de Kosaraju).
- **Union-Find Disjoint Sets** (UFDS) por sus siglas en ingles, significa:

> **Estructura de datos de conjuntos disjuntos** es una estructura de datos que almacena una colección de conjuntos disjuntos (no superpuestos).

- En otras palabras, un conjunto disjunto es un grupo de conjuntos donde ningún elemento puede estar en más de un conjunto.

1. Definición de UFDS

¿Para qué nos sirve UFDS?



- Para identificar o rastrear elementos particionados en diferentes grupos o “sets”.
- Para detectar ciclos dentro de un grafo.
- Para mostrar conectividad

Para su implementación, veremos:

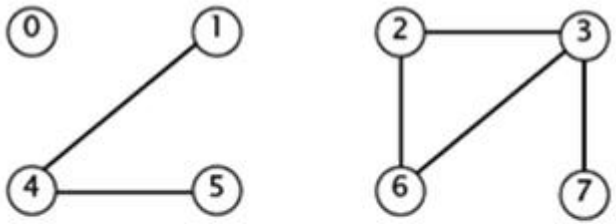
- Internamente utiliza un arreglo de padres.

1. Definición de UFDS

UN POCO DE LO BASICO

Vamos a asumir que la conexión es una relación de equivalencia. Es decir:

- **Es reflexiva: p** está conectado con **p**
- **Es simétrica:** si **p** está conectado con **q**, **q** está conectado con **p**
- **Es transitiva:** si **p** está conectado con **q** y **q** está conectado con **r**, entonces **p** está conectado con **r**

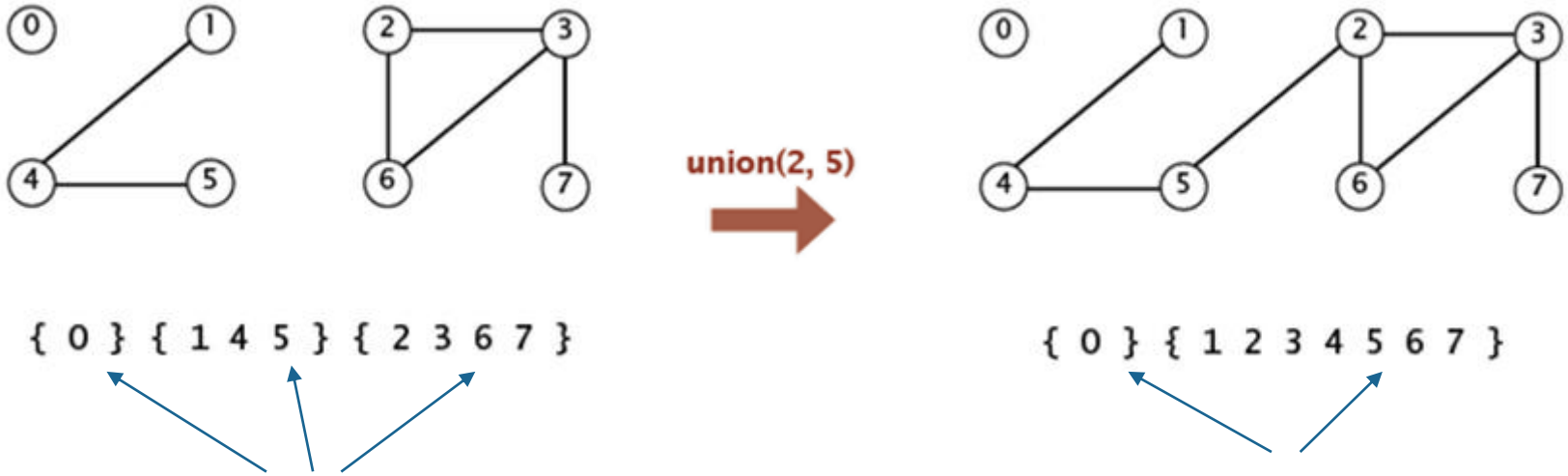


{ 0 } { 1 4 5 } { 2 3 6 7 }

3 componentes conexos

1. Definición de UFDS

UN POCO DE LO BASICO



3 componentes conexos

2 componentes conexos

2. Algoritmos o Estrategias Union-Find

- El **algoritmo de Union-Find**, también llamado algoritmo de búsqueda de unión, es un algoritmo que realiza dos operaciones útiles en esta estructura de datos:

1. **Buscar (Find):**
   - Determinar en qué subconjunto se encuentra un elemento en particular.
   - Esto se puede usar para determinar si dos elementos están en el mismo subconjunto.

2. **Unir (Union):**
   - Unir dos subconjuntos en un solo subconjunto.
   - Primero tenemos que verificar si los dos subconjuntos pertenecen al mismo conjunto. Si no, entonces no podemos realizar la unión.

2. Algoritmos o Estrategias Union-Find

Ejemplo

A partir de un Conjunto de objetos: 0 1 2 3 4 5 6 7 8 9

Elementos particionados en conjuntos disjuntos (Disjoint Sets):

0 1 {2 3 9} {5 6} 7 {4 8}

Consultar para encontrar (Find):

0 1 {**2** 3 **9**} {5 6} 7 {4 8}

Están los objetos 2 y 9 conectados?

Comando de unión (Union):

0 1 {2 3 4 8 9} {5 6} 7
Agregar una conexión entre los objetos 3 y 8.

2. Algoritmos o Estrategias Union-Find

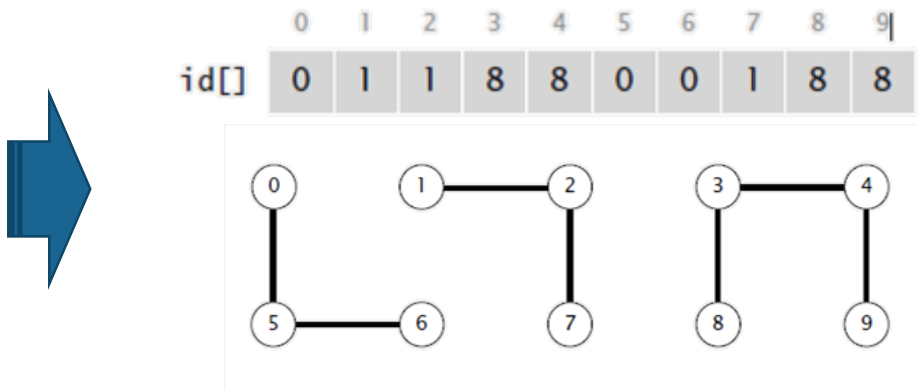
2.1. Algoritmo Quick-Find

OBJETIVO: El objetivo de este algoritmo es encontrar si dos elementos están conectados. Si no están conectados, los conectaremos.

- Este algoritmo, llamado también algoritmo entusiasta, para resolver el problema denominado: **problema de conectividad dinámica**.

Consideraciones de Quick-Find:

1. Arreglo id[] de tamaño N
2. Interpretación: **p** y **q** están conectados si tienen el mismo id.



2. Algoritmos o Estrategias Union-Find

2.1. Algoritmo Quick-Find

Reglas de QUICK FIND:

- La estructura de los datos de este algoritmo incluye:

  - Una matriz de enteros **id[]** de tamaño **N** (donde N es cualquier entero).

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

  - Suponemos que la matriz de enteros **id[]** es un rango de 0 a N-1.

  - Los elementos **p** y **q** son 2 enteros en la matriz **id[]**.

  - Los elementos **p** y **q** están conectados si tienen el mismo id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	**9**	**9**	**9**	**6**	**6**	7	8	**9**

5 y **6** están conectados.

2, **3**, **4** y **9** están conectados.

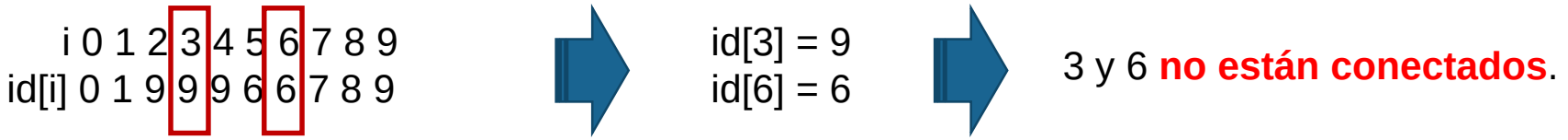
} Tienen el mismo id

2. Algoritmos o Estrategias Union-Find

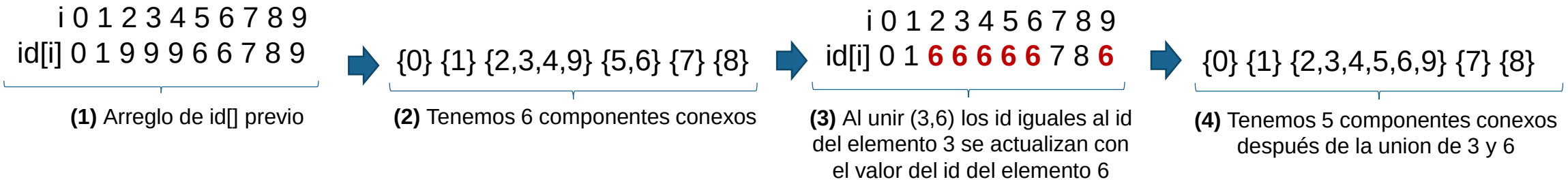
2.1. Algoritmo Quick-Find

Ejemplo:

Paso 1 -> Find: Verificar si **p** y **q** tienen el mismo id ⇨ ¿Los elementos 3 y 6 están conectados?



Paso 2 -> Union: Unir componentes conteniendo **p** y **q** ⇨ Unir 3 y 6



Los elementos 2, 3, 4, 5, 6 y 9 están ahora conectados.

2. Algoritmos o Estrategias Union-Find

2.1. Algoritmo Quick-Find

```
class QuickFind(object):
    def __init__(self, N):
        self.lst = list(range(N))


def find(self, p, q):
    return self.lst[p] == self.lst[q]


def union(self, p, q):
    pid = self.lst[p]

    qid = self.lst[q]

    for ind, x in enumerate(self.lst):
      if x == pid:
        self.lst[ind] = qid

    return self.lst
```

Complejidad:

- El algoritmo Quick-Find puede tomar M × N pasos para procesar M comandos de unión sobre N objetos.

Ejemplo

➥10^{10} aristas conectando 10^{9} nodos.
➥Quick-Find tomará mas de 10^{19} operaciones.
➥300+ años de tiempo computacional.

Algoritmo	Inicialización	Union	Consulta
QuickFind	N	N	1

Desventajas de QuickFind

- Vemos que es muy lento en las uniones!
- Tenemos que agregar N elementos: $O(n^2)$

2. Algoritmos o Estrategias Union-Find

2.2. Algoritmo Quick-Union

- En el algoritmo de búsqueda rápida (Quick-Find), cada vez que hacíamos una unión, teníamos que iterar a través de toda la matriz. Eso no pasará en Quick-Union porque solo cambiaremos una identificación.

OBJETIVO: Mejorar el algoritmo de búsqueda rápida Quick-Find para que sea más eficiente.

- El enfoque principal estará en el método de 'unión'. Ese fue el método más ineficiente en Quick-Find
- Aquí ayudará un enfoque perezoso para el método de unión.

2. Algoritmos o Estrategias Union-Find

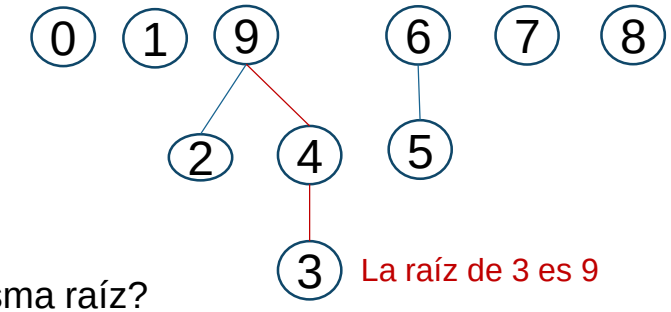
2.2. Algoritmo Quick-Union

Reglas de QUICK UNION:

- Arreglo id[] de tamaño N
- Interpretación: **id[i]** es padre de **i**
- La **Raíz** de **i** es **id**[id[id[… id[i] …]]] (sigue hasta que no cambie, el algoritmo asegura que no haya ciclos)

Interpretación:

i 0 1 2 3 4 5 6 7 8 9
Raíz = id[i] 0 1 9 4 9 6 6 7 8 9



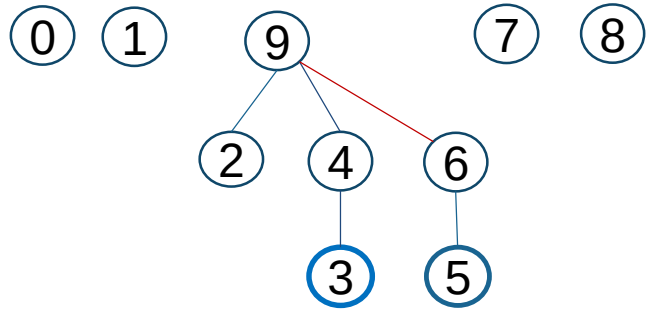
Find: p(3) y q(5) ¿Tienen la misma raíz?

Union: Unir componentes conteniendo p y q

Unir 3 y 5 = Unir **Raíz** de 3 y 5 = Unir 9 y 6

i 0 1 2 3 4 5 6 7 8 9
Raíz = id[i] 0 1 9 4 9 6 **9** 7 8 9

Solo un valor cambia



2. Algoritmos o Estrategias Union-Find

2.2. Algoritmo Quick-Union

```
class QuickUnion(object):
    def __init__(self, N):
      self.lst = list(range(N))

    def find(self, ind):
      while ind != self.lst[ind]:
        ind = self.lst[ind]
      return ind

    def connect(self, p, q):
      return self.find(p) == self.find(q)

    def union(self, p, q):
      pid = self.find(p)
      self.lst[pid] = self.find(q)
```

Complejidad:

- La unión puede ser muy costosa (N pasos).
- Los arboles pueden volverse muy altos Find puede ser también muy costoso (N pasos).
- Necesario hacer Find para hacer Union.

Algoritmo	Inicialización	Union	Consulta
QuickUnion	N	N+	N

Desventajas de QuickUnion:

- Los árboles podrían quedar muy grandes
- La consulta puede ser muy cara

2. Algoritmos o Estrategias Union-Find

2.3. Algoritmo Quick-Union Ponderado

OBJETIVOS:

- Modificar el algoritmo Quick-Union para evitar arboles altos.
- Mantener registro del tamaño de cada componente.
- Balancear conectando los arboles pequeños debajo del más grande.

Se incorporan dos tipos de mejoras:

Mejora #1: **Árbol Ponderado**

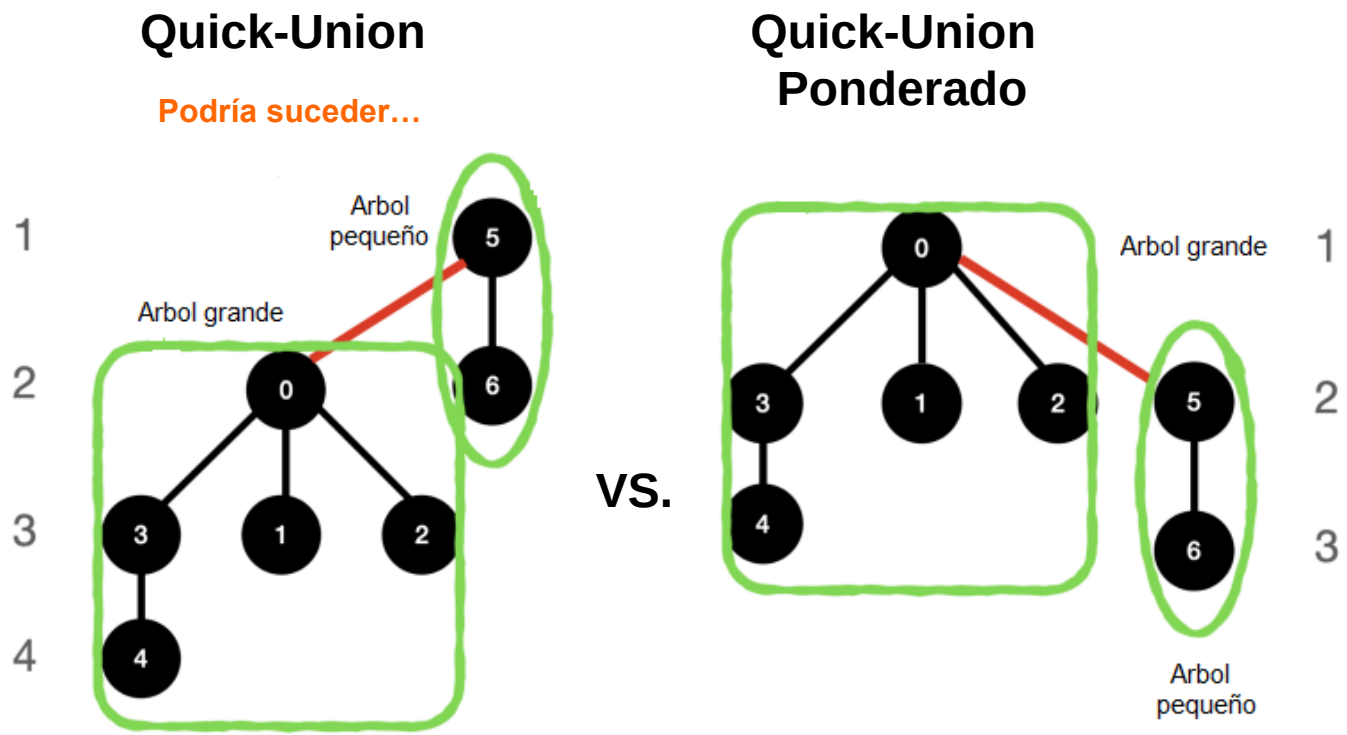
Mejora #2: Compresión de la ruta

2. Algoritmos o Estrategias Union-Find

2.3. Algoritmo Quick-Union Ponderado

Mejora #1: Árbol Ponderado

- La idea básica del árbol ponderado es que siempre coloca el árbol más pequeño (el árbol con menos nodos) debajo del árbol más grande (el árbol con más nodos).
- En **Quick-Union**, cuando vinculamos dos árboles de diferentes tamaños, podemos vincular el árbol más grande debajo del árbol más pequeño, creando más capas.
- En **Quick-Union Ponderado**, examinaremos el tamaño de dos árboles y nos aseguraremos de unir solo el árbol más pequeño debajo del árbol más grande.
- Al aplicar este método, podemos garantizar que el árbol no crecerá demasiado alto.

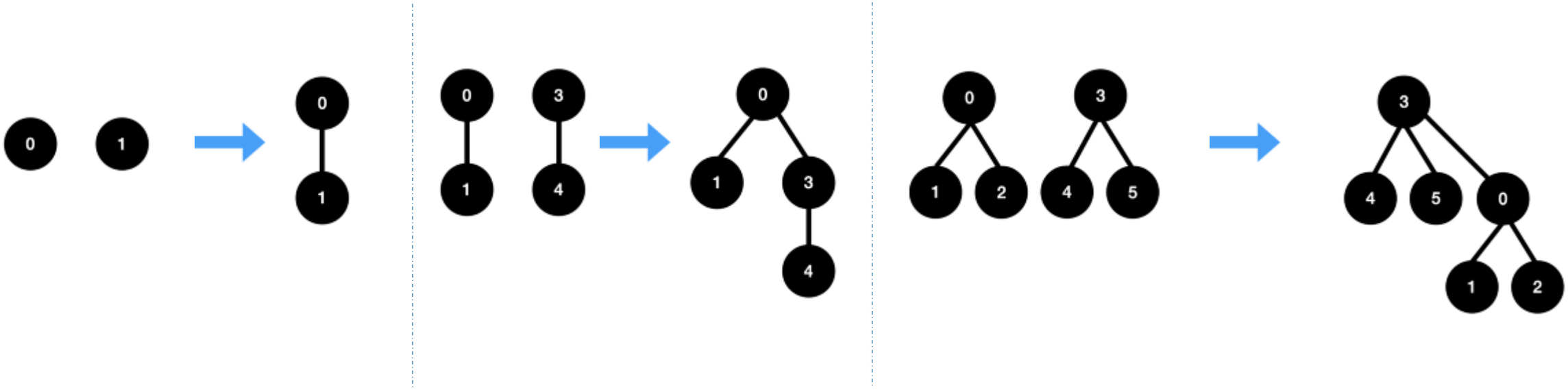


2. Algoritmos o Estrategias Union-Find

2.3. Algoritmo Quick-Union Ponderado

Mejora #1: Árbol Ponderado

- La estructura de árbol de **Quick-Union Ponderado** aumentará una capa más cuando el número de dos capas de árbol sea el mismo .



2. Algoritmos o Estrategias Union-Find

2.3. Algoritmo Quick-Union Ponderado

```
def find(s, a):
    i = a
    while s[i] >= 0:
        i = s[i]
    return i

def union(s, a, b):
    pa = find(s, a)
    pb = find(s, b)
    if pa == pb: return
    if s[pa] < s[pb]:
        s[pa] += s[pb]
        s[pb] = pa
    elif s[pb] < s[pa]:
        s[pb] += s[pa]
        s[pa] = pb
    else:
        s[pa] += s[pb]
        s[pb] = pa
```

Mejora en la función Union

Find: Si **p** y **q** están conectados no hacemos nada. Si no lo están los unimos.

Union: Al unir componentes, revisaremos su ponderación en cuanto a tamaño, recalculando la ponderación y asignando un valor nuevo de raíz según los siguientes casos:

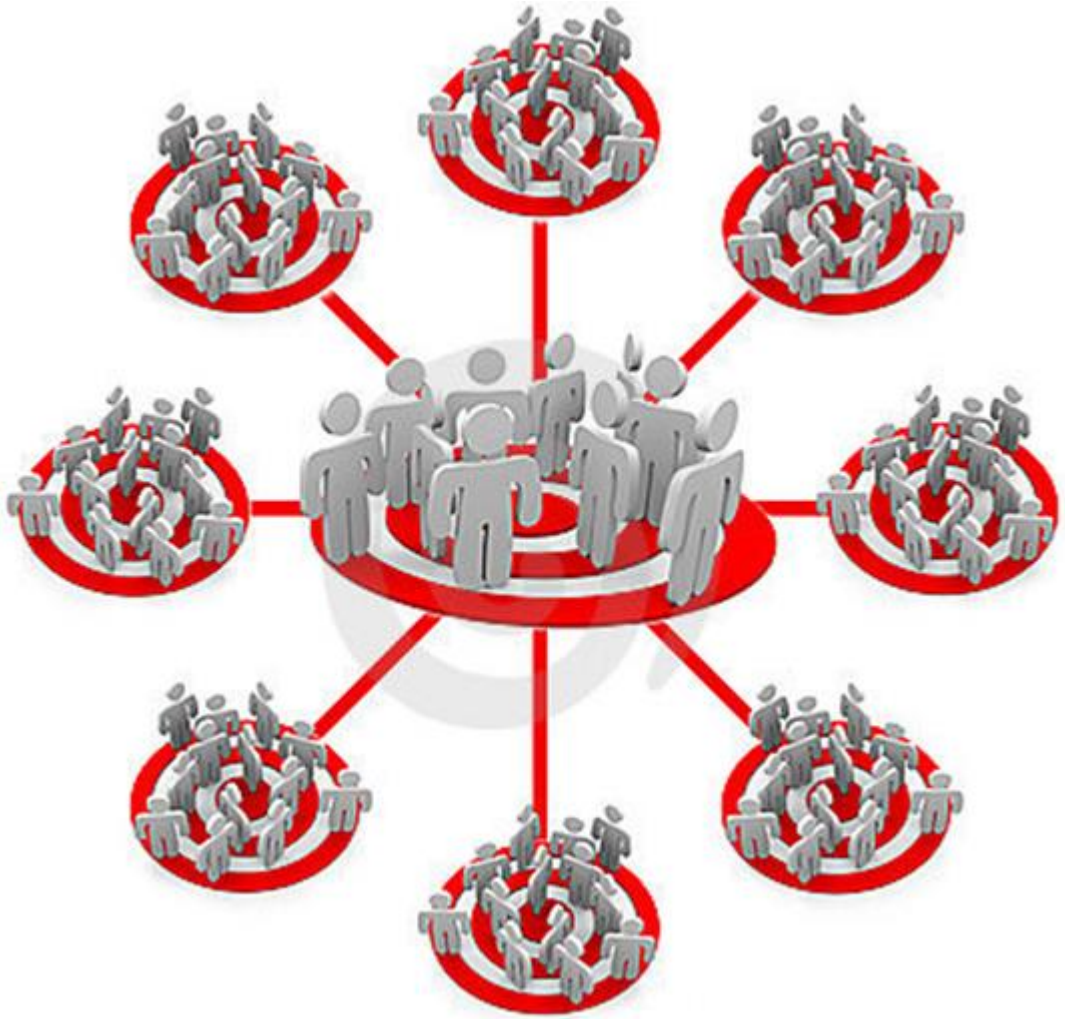
- **Ponderación de p < ponderación de q**, recalculamos ponderación para p y actualizamos la raíz de q con la de p.
- **Ponderación de p > ponderación de q**, recalculamos ponderación para q y actualizamos la raíz de p con la de q.
- **Igual ponderación de p y q**, recalculamos ponderación para p y actualizamos la raíz de q con la de p (igual que en el primer caso).

Complejidad

Algoritmo	Inicialización	Union	Consulta
QuickUnion Ponderado	N	Lg *N*	Lg *N*

- **Find:** toma un tiempo proporcional a la profundidad de **p** y **q**.
- **Union:** toma un tiempo constante, dadas las raíces.
- La profundidad es a lo sumo de log N

3. Aplicaciones de UFDS



Manejar grupos eficientemente

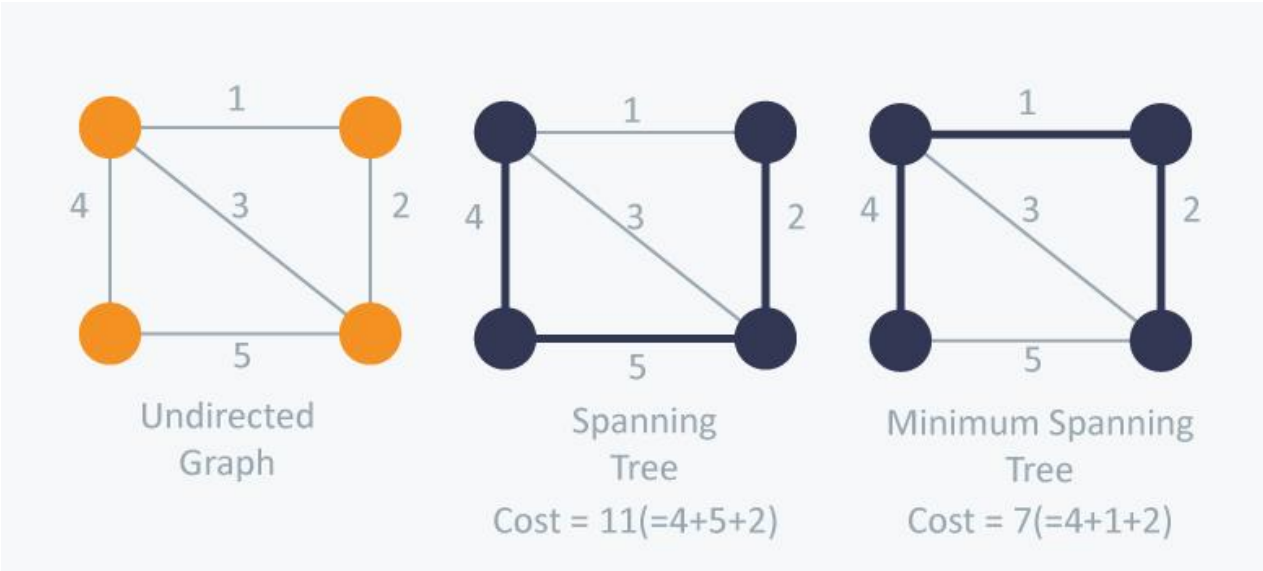
3. Aplicaciones de UFDS

Estos tipos de algoritmos ayudan a manipular los objetos de todo tipo:

- Computadoras en una red.
- Elementos en un conjunto matemático.
- Sitios metálicos en un sistema compuesto.
- Píxeles en una foto digital.
- Amigos en una red social.
- Transistores en un chip de computadora.

PREGUNTAS

Dudas y opiniones



Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 10: Árbol de Expansión Mínima (MST)

Ing. Patricia Reyes Silva
pcsiprey@upc.edu.pe

Complejidad Algorítmica

Semana 10 / Sesión 1

MÓDULO 10: Arboles de Expansión Mínima (MST)

Contenido

1. Conceptos básicos y MST
2. Algoritmos MST
   - 2.1. Algoritmo Kruskal
   - 2.2. Algoritmo PRIM
3. Aplicaciones del MST

Preguntas

1. Conceptos básicos y MST

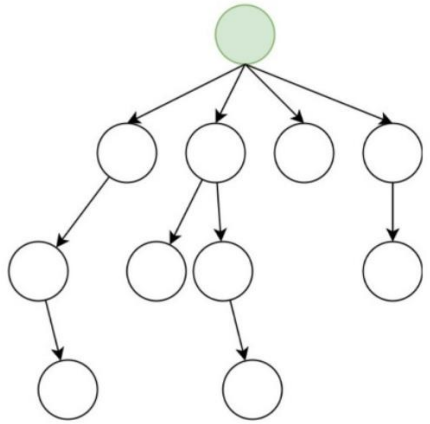
¿Qué es un árbol dentro de la teoría de grafos?

- Antes de responder, formalicemos algunos conceptos que ya conocemos.

Árbol: es una estructura de datos que simula una estructura de árbol jerárquica compuesta por un conjunto finito de uno o más nodos tales que:

- Hay un nodo especialmente designado llamado **raíz**.
- Los nodos restantes se dividen en n>=0 **conjuntos disjuntos** T 1 , T 2 , T 3 , …, T n

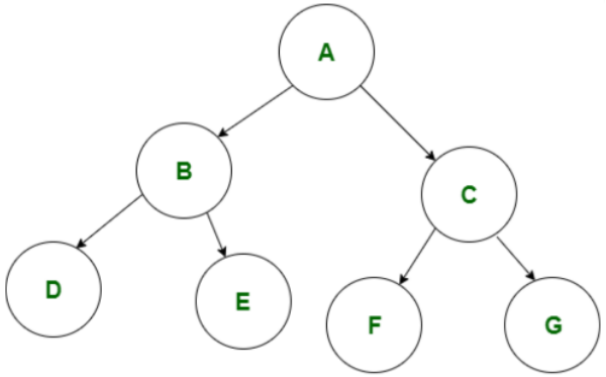
  donde T 1 , T 2 , T 3 , …, T n se denomina subárboles o hijos de la raíz.



Árbol General

Puede tener cero o muchos subárboles secundarios desordenados

VS

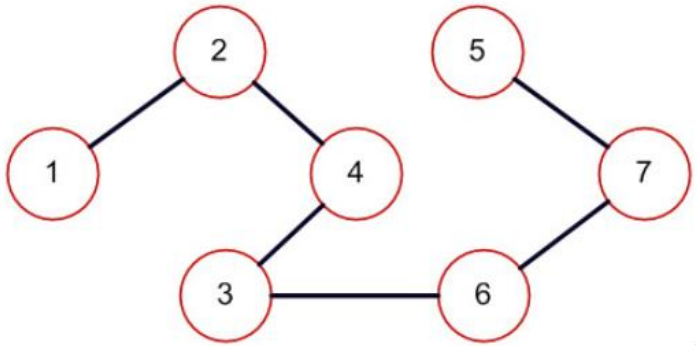


Árbol binario

Cada nodo puede tener como máximo dos nodos (subárbol izquierdo y derecho), ordenados.

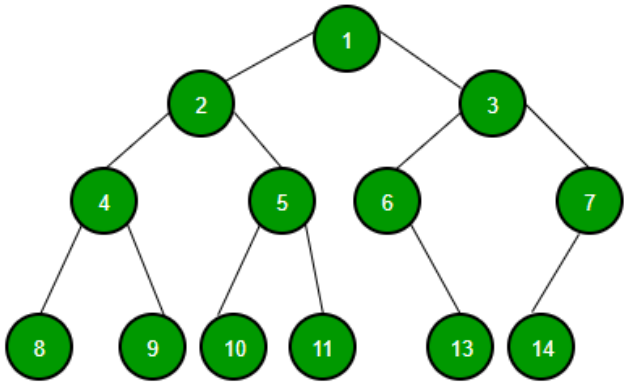
1. Conceptos básicos y MST

¿Todo grafo es un árbol?



Grafo

vs



Árbol binario

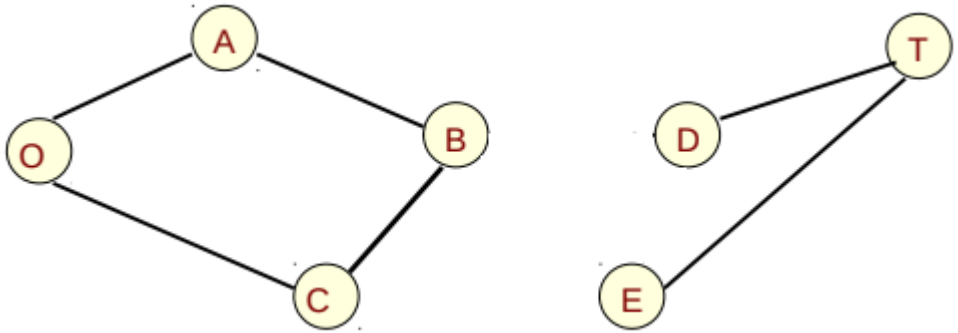
1. Conceptos básicos y MST

¿Hay diferencias entre un grafo y un árbol?

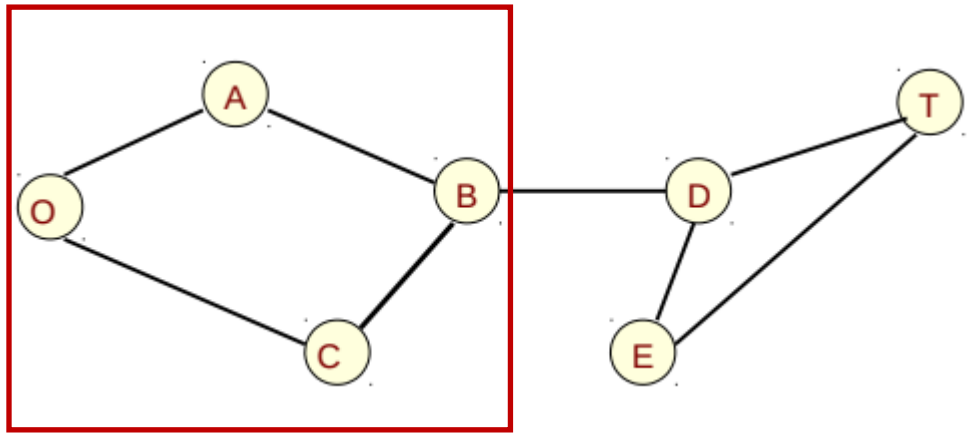
GRAFO	ARBOL
1. El Grafo es una estructura de datos no lineal.	1. Igual.
2. Es una colección de vértices/nodos y aristas.	2. Igual.
3. Cada nodo puede tener cualquier número de aristas.	3. Pueden contener cualquier número de nodos secundarios (subárboles). Pero en el caso de los árboles binarios, cada nodo puede tener como máximo dos nodos secundarios.
4. No hay un nodo único llamado raíz en el grafo.	**4. Hay un nodo único llamado raíz** en los árboles.
5. Se puede formar un ciclo.	5. No existe ningún ciclo.
6. Aplicación: Encontrar la ruta más corta.	6. Aplicación: árboles de juego, árboles de decisión.

1. Conceptos básicos y MST

Sea el siguiente grafo: ¿Es un árbol?

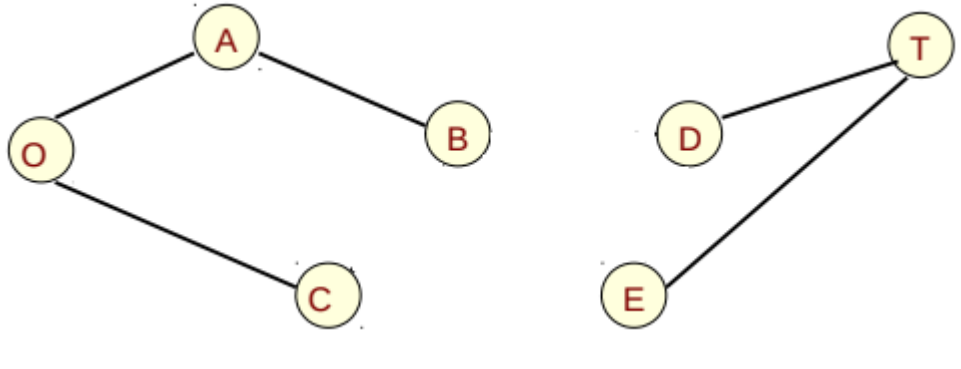


NO, es un árbol porque una red o grafo con ciclo, ¡no es un árbol!

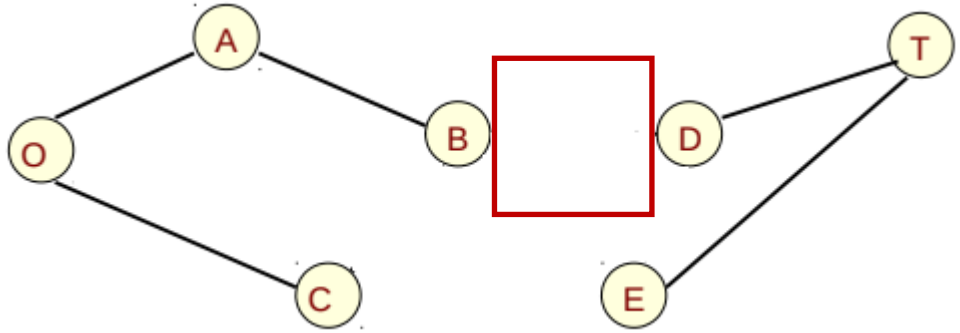


1. Conceptos básicos y MST

Sea el siguiente grafo: ¿Es un árbol?



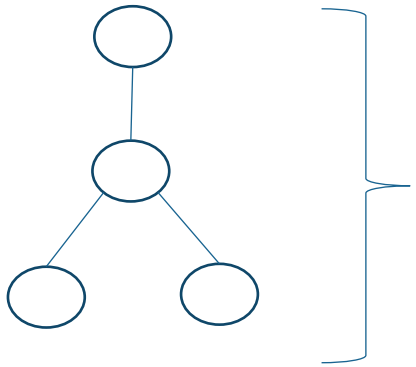
- **NO**, es un árbol porque una red NO CONEXA, ¡no es un árbol! Es un bosque.



- Un bosque es una colección disjunta de árboles.

CARACTERISTICAS DE UN ARBOL

- **Es un grafo acíclico conexo**, en otras palabras, una árbol es un grafo conexo sin ciclos.
- Los bordes de un árbol se conocen como ramas.
- Los elementos de los árboles se llaman sus nodos.
- Los nodos sin nodos secundarios se denominan nodos hoja.
- Un árbol con 'n' vértices tiene aristas 'n-1'.



ARBOL:

Grafo acíclico conexo con nodos-1 aristas

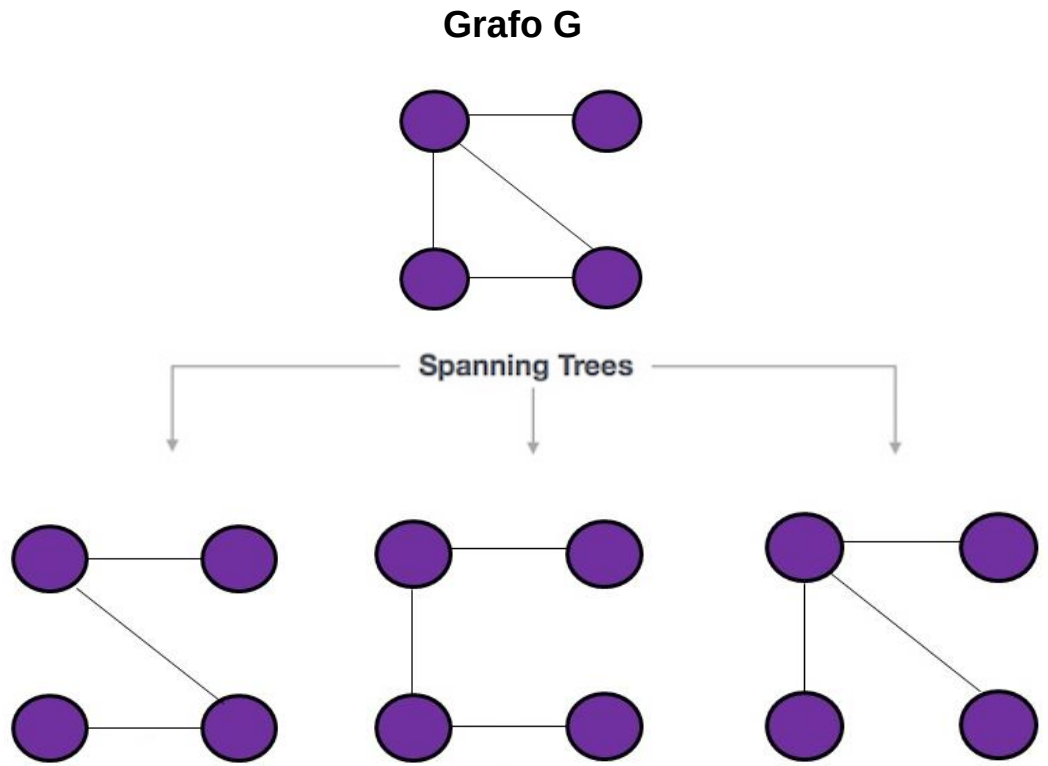
1. Conceptos básicos y MST

ARBOL DE EXPANSION (Spanning Tree)

Dado un grafo no dirigido y conexo G = (V,A), un árbol de expansión del grafo G :

1. Es un árbol que se extiende de G, es decir, incluye todos los vértices de G.

2. Es un subgrafo de G, donde cada borde en el árbol pertenece a G.

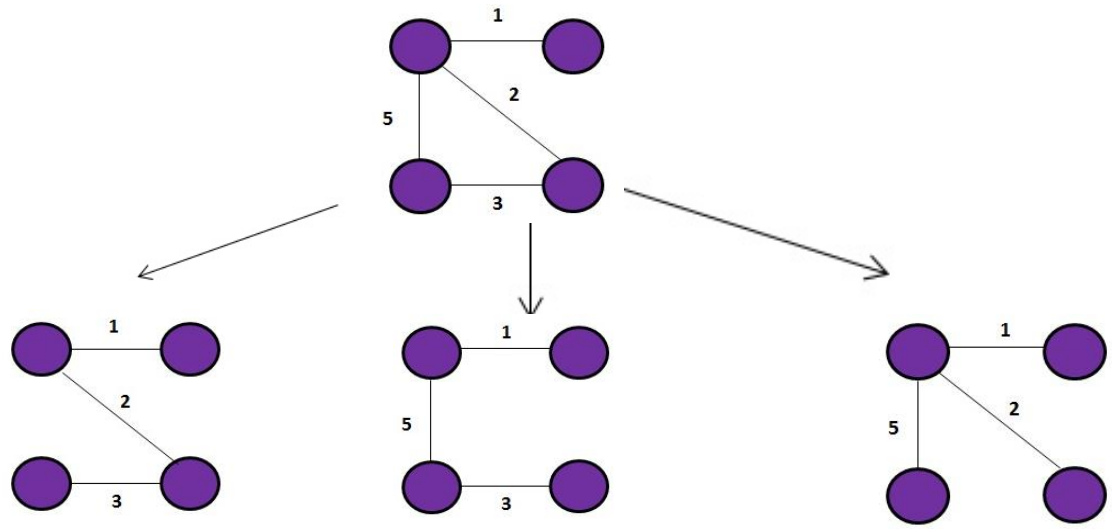


1. Conceptos básicos y MST

- **Minimum spanning tree (MST)** o **Minimum weight spanning tree**, significa árbol de expansión mínimo o árbol de expansión de peso mínimo**,** y es un modelo de optimización de redes.
- Un árbol de expansión mínima es un **subconjunto de los bordes ponderados de un grafo no dirigido** que conecta todos los vértices entre sí, **sin ningún ciclo** y con el objetivo que el **peso total de los bordes sea el mínimo posible**.

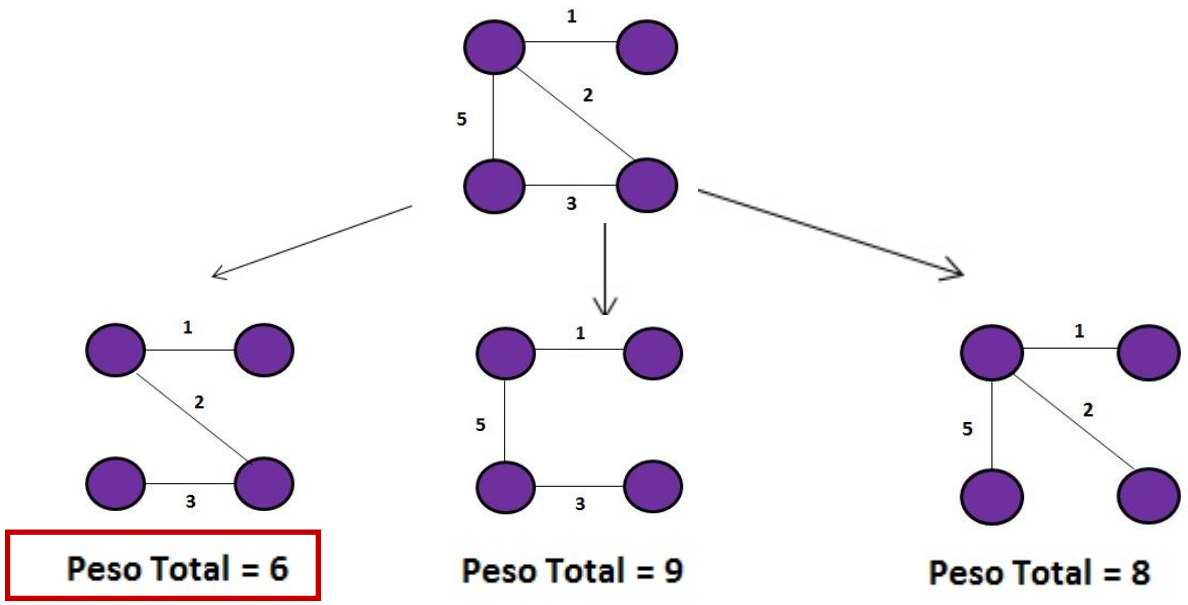
¿Qué es un MST?
(Árbol de Expansión Mínimo)

Sea G (V,A) ➡ Grafo G ponderado con 4 vértices y 4 aristas



1. Conceptos básicos y MST

Sea G (V,A) ➡ Grafo G ponderado con 4 vértices y 4 aristas



<u>Árbol de expansión mínimo (o de peso mínimo)</u>

- El <u>costo del árbol de expansión es la suma de los pesos de todos los bordes del árbol</u>.
- Puede haber muchos árboles de expansión (a partir de un grafo).
- El **árbol de expansión mínimo** es el **árbol de expansión donde el costo es mínimo** entre todos los árboles de expansión.
- También puede haber muchos árboles de expansión mínimos.

2. Algoritmos MST

¿Qué algoritmos solucionan el problema del MST?

Existen dos Algoritmos famosos para resolver este problema de calculo del árbol de expansión mínimo:

- **Algoritmo Kruskal**
- **Algoritmo Prim**

Es común en ambos algoritmos:

- Utilizar la propiedad anterior.
- Ser de tipo voraz (codicioso o greedy), es decir, que seleccionan uno de los candidatos con el criterio que es mejor en cada momento (menor costo).
- Giran en torno a verificar si al agregar un borde o un vértice se crea un ciclo o no.

2. Algoritmos MST

2.1. Algoritmo de KRUSKAL

- El **algoritmo de Kruskal** construye el árbol de expansión agregando aristas una por una en un árbol de expansión en crecimiento.
- El **algoritmo de Kruskal** sigue un enfoque codicioso (voraz o greedy), ya que en cada iteración encuentra un borde que tiene el menor peso y lo agrega al árbol de expansión en crecimiento.

Pasos del algoritmo

- Ordenar los bordes del gráfico con respecto a sus pesos.
- Agregar bordes al MST desde el borde con el peso más pequeño hasta el borde con el peso más grande.
- Solo agregar bordes que no formen un ciclo, bordes que conecten solo componentes desconectados.
- La forma más común de averiguar si dos componentes están desconectados, es aplicando el algoritmo **Union Find**. Este divide los vértices en grupos y nos permite verificar si dos vértices pertenecen al mismo grupo o no y, por lo tanto, decidir si agregar un borde crea un ciclo.

Veamos un ejemplo…

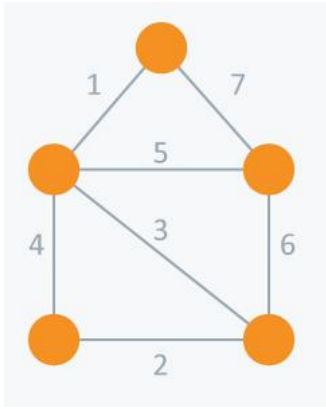
2. Algoritmos MST

2.1. Algoritmo de KRUSKAL

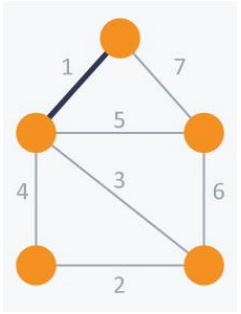
En el algoritmo de Kruskal, en cada iteración seleccionaremos la arista con el peso más bajo sin que forme un ciclo.

Ejemplo # 1:

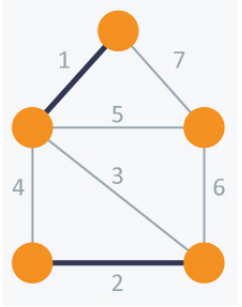
G(V,A)



1. Comenzaremos primero con el borde de peso más bajo, es decir, los bordes con peso 1.

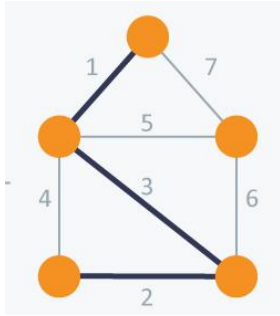


2. Después de eso, seleccionaremos el segundo borde de peso más bajo, es decir, el borde con peso 2.

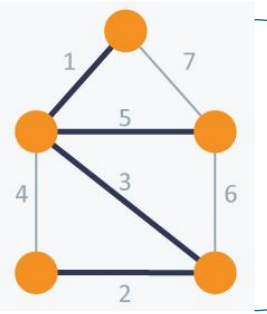


Observamos que los bordes 1 y 2 son totalmente disjuntos.

3. El próximo borde será el tercer borde ponderado más bajo, el borde con peso 3, que conecta las dos partes separadas del grafo



4. No podemos elegir el borde con peso 4, porque con eso se creará un ciclo y no podemos tener ningún ciclo. Por tanto elegimos el borde con peso 5.



Ignoramos también los bordes con pesos 6 y 7 porque también estarían formando ciclos.

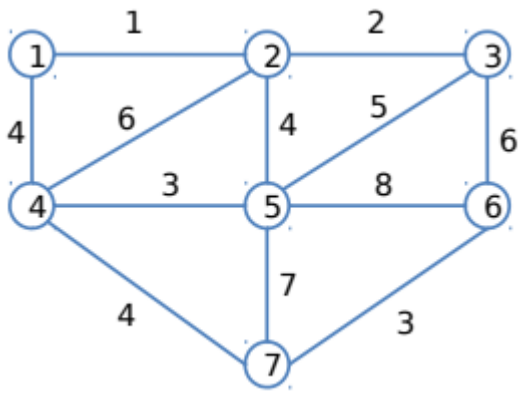
Finalmente, terminamos con un **árbol de expansión mínimo** con un **costo total del MST = 11** (suma de los costos de los bordes 1 + 2 + 3 + 5)

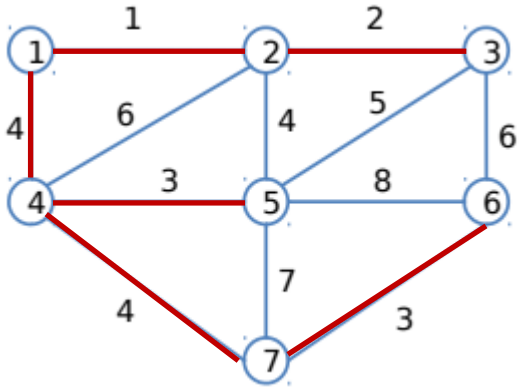
2. Algoritmos MST

2.1. Algoritmo de KRUSKAL

Ejemplo #2: Encontrar el árbol de expansión mínima por el algoritmo de Kruskal.

G(V,A)

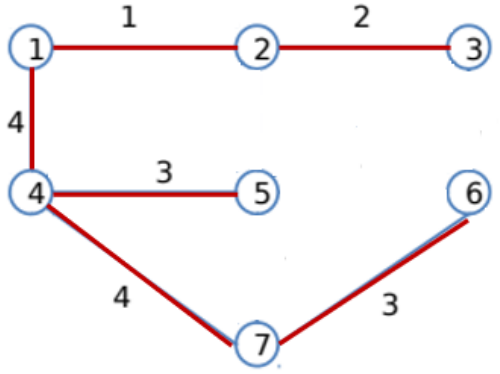




Costos de las aristas

1-2 = 1
2-3 = 2
4-5 = 3
6-7 = 3
1-4 = 4
4-7 = 4

Árbol de expansión mínimo



Costo total del MST = 17

2. Algoritmos MST

2.1. Algoritmo de KRUSKAL

Implementación

Necesitamos:

- Ordenar las aristas de G, de menor a mayor: O(a log a).
- Saber si una arista dada (v, w) provocará un ciclo.
  - ➢ ¿Cómo comprobar rápidamente si (v, w) forma un ciclo?
    - ✓ Una arista (v, w) forma un ciclo si v y w están en el mismo **componente conexo**.
    - ✓ La relación “están en el mismo componente conexo” es una relación de equivalencia.

Pseudocodigo

Sea el grafo G= (V, A)

1. Empezar con un grafo sin aristas: G’= (V, Ø)
2. Seleccionar la arista de menor coste de A.
   - ➢ Si la arista seleccionada forma un ciclo en G’, eliminarla.
   - ➢ Si no, añadirla a G’.
3. Repetir el paso 2. hasta tener n-1 aristas

```
KRUSKAL(G):
A = Ø
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

2. Algoritmos MST

2.2. Algoritmo PRIM

- El algoritmo de Prim también usa el enfoque Greedy (voraz o codicioso) para encontrar el árbol de expansión mínimo.
- En el Algoritmo de Prim hacemos crecer el árbol de expansión desde una posición inicial.
- A diferencia de un borde en Kruskal, agregamos un vértice al árbol de expansión creciente en Prim.

Pasos del algoritmo

- Mantener dos conjuntos disjuntos de vértices. Uno que contiene vértices que están en el árbol de expansión en crecimiento y el segundo que no está en el árbol de expansión en crecimiento.
- Seleccionar el vértice menos costoso que esté conectado al primer árbol de expansión en crecimiento y que no esté en el segundo árbol de expansión en crecimiento.
- Insertar los vértices, que están conectados al primer árbol de expansión en crecimiento, en la cola de prioridad.
- Consultar por los ciclos. Marcar los nodos que ya han sido seleccionados e insertar solo aquellos nodos en la cola de prioridad que no estén marcados.

Veamos a través de un ejemplo…

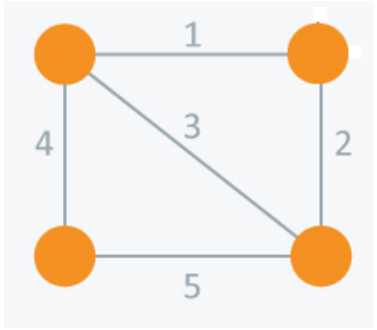
2. Algoritmos MST

2.2. Algoritmo PRIM

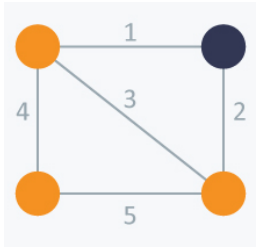
En el algoritmo de PRIM, al igual que en Kruskal, en cada iteración seleccionaremos la arista con el peso más bajo sin que forme un ciclo.

Ejemplo # 1:

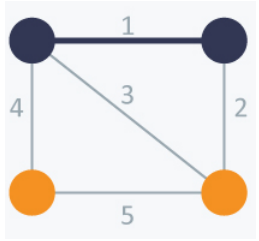
G(V,A)



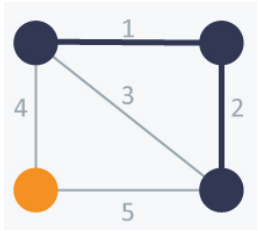
1. Comenzaremos con un nodo arbitrario (no importa cuál) y lo marcaremos.



2. En cada iteración marcaremos un nuevo vértice que sea contiguo al que ya hemos marcado (primero la arista con borde 1 y luego 2).

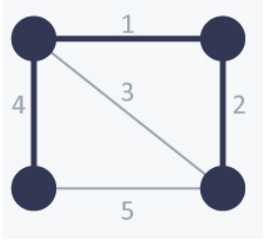


Costo mst= 1



Costo mst= 1+2

3. No elegimos el borde con peso 3 porque estaría formando un ciclo, en cambio, elegimos 4 por ser el de menor valor.



Costo mst= 1+2 + 4

Finalmente, terminamos con un **árbol de expansión mínimo**

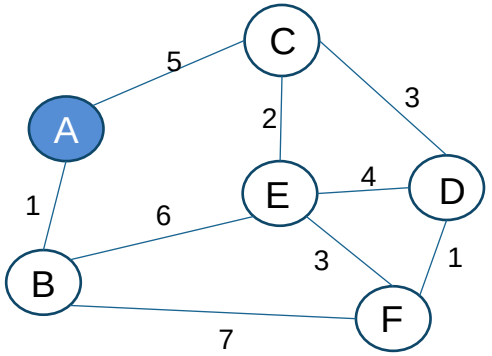
Costo total del MST = 7
(suma de los costos de los bordes 1 + 2 + 4)

2. Algoritmos MST

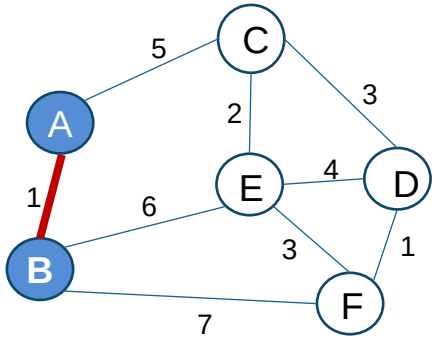
2.2. Algoritmo PRIM

Ejemplo # 2: Encontrar el árbol de expansión mínima por el algoritmo de Prim.

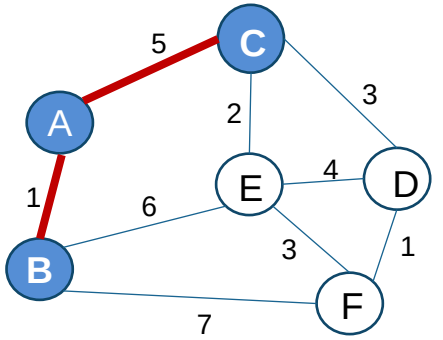
G(V,A)



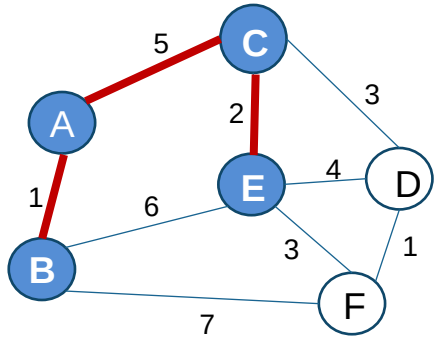
Aleatoriamente, elegimos el nodo A para iniciar.



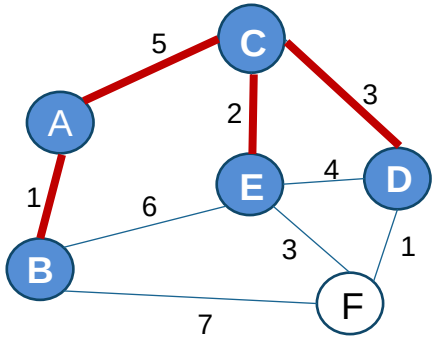
(1) A-B = 1



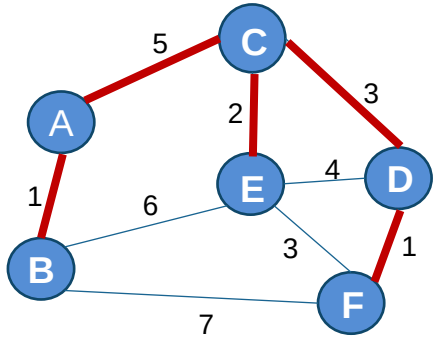
(2) A-C = 5 (porque de B-E y B-F el costo > 5)



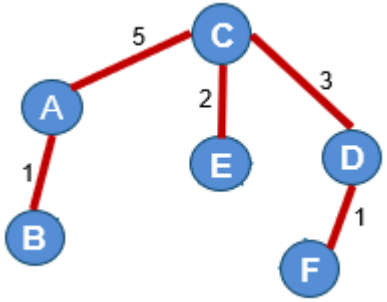
(3) C-E = 2



(4) C-D = 3



(5) D-F = 1



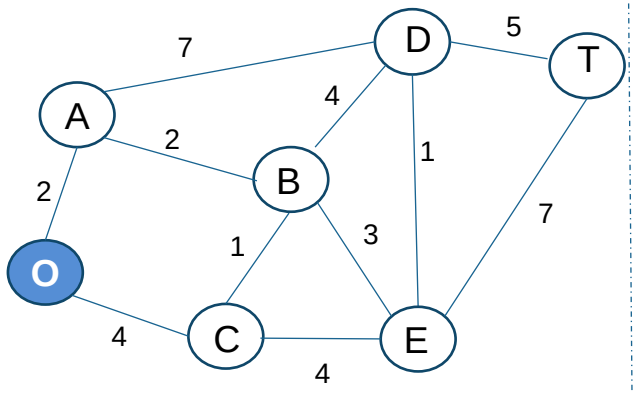
Costo total del MST = 12

2. Algoritmos MST

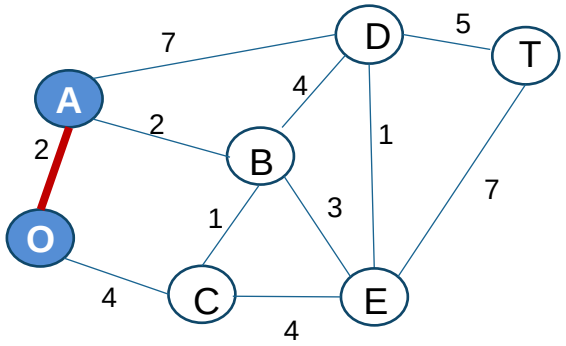
2.2. Algoritmo PRIM

Ejemplo # 3: Encontrar el árbol de expansión mínima por el algoritmo de Prim.

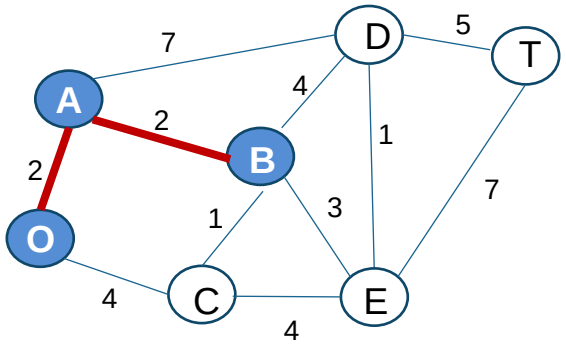
G(V,A)



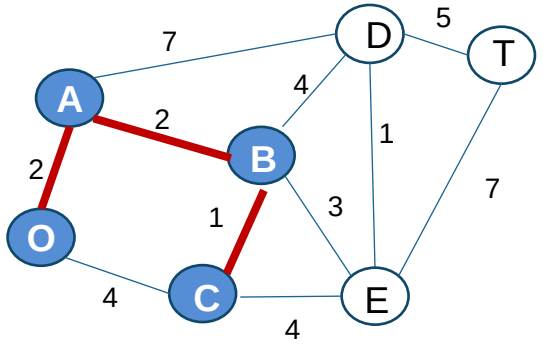
Aleatoriamente, elegimos el nodo "O" para iniciar.



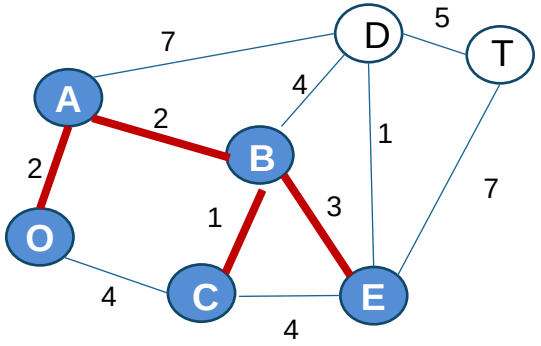
(1) O-A = 2



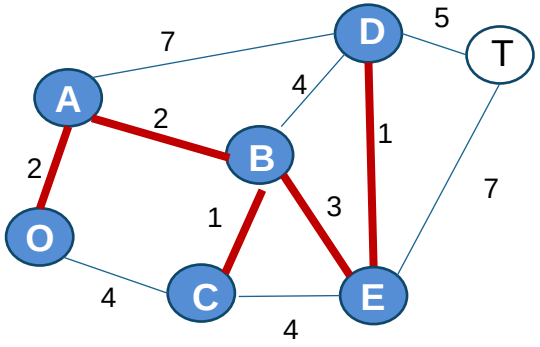
(2) A-B = 2



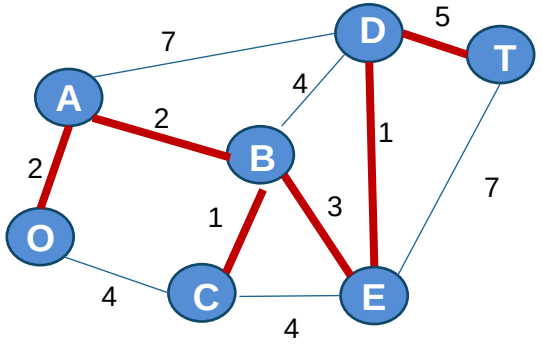
(3) B-C = 1



(4) B-E = 3



(5) E-D = 1



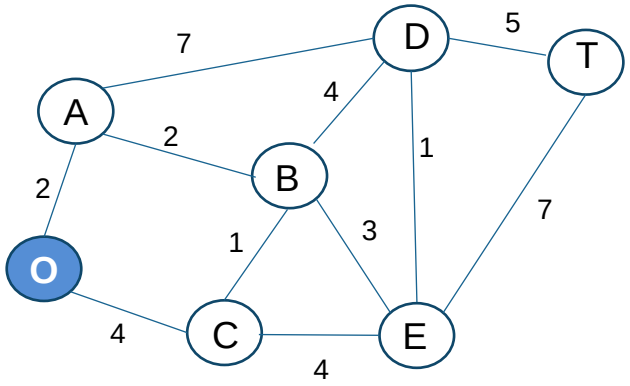
(6) D-T = 5

2. Algoritmos MST

2.2. Algoritmo PRIM

Ejemplo # 3: Encontrar el árbol de expansión mínima por el algoritmo de Prim.

G(V,A)



O-A = 2

A-C = 5

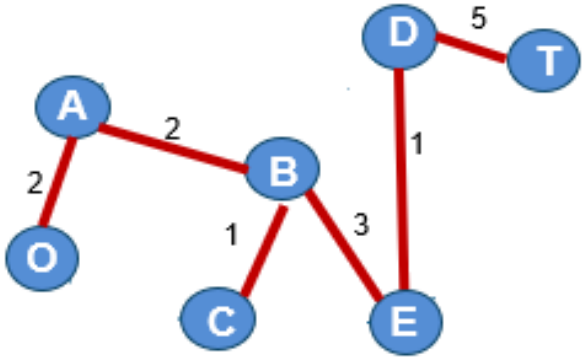
B-C = 1

B-E = 3

E-D = 1

D-T = 5

Finalmente, terminamos con un **árbol de expansión mínimo**



Costo total del MST = 17

(2+5+1+3+1+5)

2. Algoritmos MST

2.2. Algoritmo PRIM

Implementación

1. Empezar en un vértice cualquiera v. El árbol consta inicialmente sólo del nodo v.
2. En el resto de vértices, buscar el que esté más próximo a v (es decir, con la arista (v, w) de coste mínimo). Añadir w y la arista (v, w) al árbol.
3. Buscar el vértice más próximo a cualquiera de estos dos. Añadir ese vértice y la arista al árbol de expansión.
4. Repetir sucesivamente hasta añadir los n vértices.

- El árbol T aumenta 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(n2).

Pseudocodigo

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

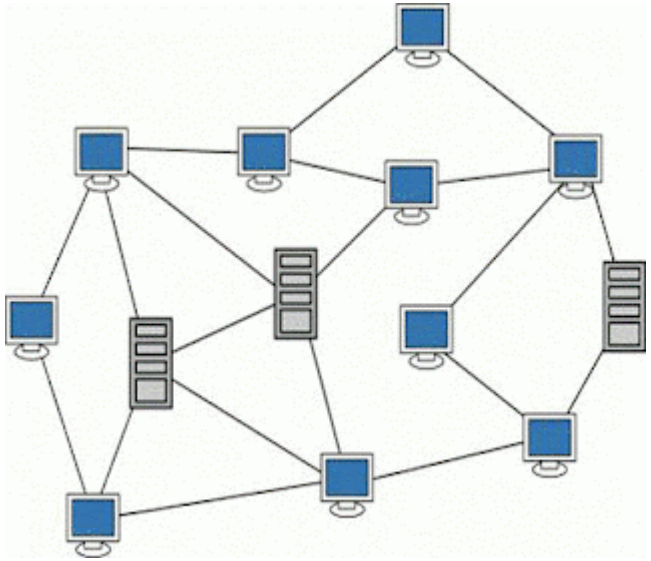
- La solución se construye poco a poco, empezando con una solución “vacía”.
- Implícitamente, el algoritmo maneja los conjuntos:
  - **V:** Vértices del grafo.
  - **U:** Vértices añadidos a la solución.
  - **V-U:** Vértices que quedan por añadir.

3. Aplicaciones de MST

- Varios algoritmos de búsqueda de rutas, entre ellos, el **algoritmo de Dijkstra** y el **algoritmo de búsqueda A***, construyen internamente un árbol de expansión como paso intermedio para resolver el problema.
- Las personas a menudo utilizamos algoritmos que construyen gradualmente un árbol de expansión (o muchos árboles similares) como pasos intermedios en el proceso de encontrar el árbol de expansión mínimo,

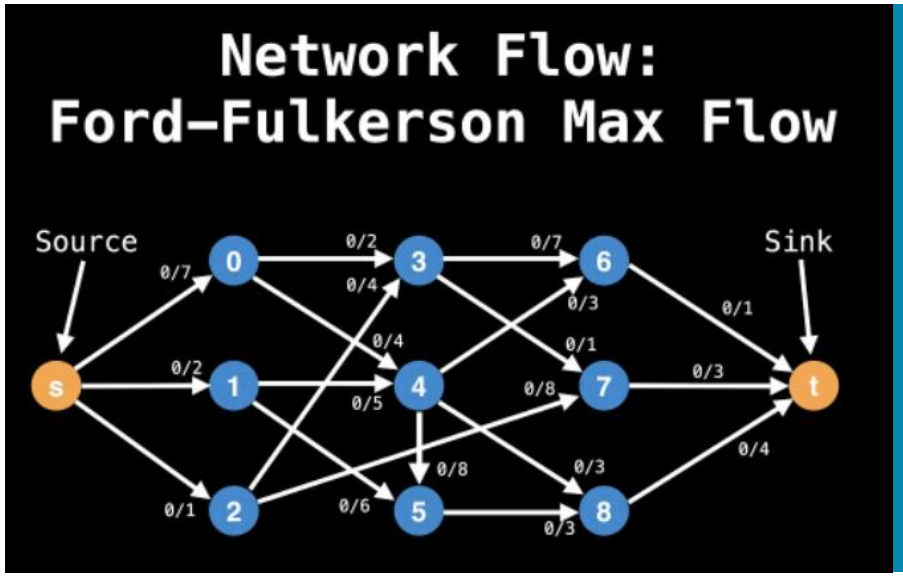
Principales aplicaciones

- Minimizar:
  - El costo de las redes eléctricas (para la transmisión de energía eléctrica de alto voltaje).
  - Las conexiones de cableado de equipos eléctricos.
  - Las conexiones de tuberías para conectar diferentes localidades.
  - El costo total de los trayectos en las redes de transporte.
  - Las conexiones de cableado en las redes de telecomunicaciones.
  - Las rutas terrestres, para conectar un grupo de computadoras en una red cableada que se encuentran a distancias variables.
  - Las rutas aéreas. Siendo los vértices del grafo las ciudades y los bordes las rutas entre las ciudades, cuanto más se tenga que viajar, más costará.



PREGUNTAS

Dudas y opiniones



Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 11: Flujo Máximo en Redes

Ing. Patricia Reyes Silva
pcsiprey@upc.edu.pe

Complejidad Algorítmica

Semana 11 / Sesión 1

MÓDULO 11: Flujo Máximo en Redes

Contenido

1. Conceptos básicos – Flujos en redes
2. El problema del Flujo Máximo en redes
3. Algoritmo Ford – Fulkerson
4. Aplicaciones del Flujo en Redes

Preguntas

1. Conceptos básicos – Flujos en redes

En teoría de grafos:
¿Qué es **una red** y un **flujo de red**?

Red: En teoría de grafos, **una red** es un **grafo dirigido con pesos.**

Flujo de Red: es la cantidad de "algún contenido" que circula dentro de una red.

- Debemos entender que en una red, las aristas representan canales por los que puede circular cualquier elemento. Por ejemplo:
  - Datos
  - Agua
  - Autos
  - Corriente eléctrica, etc.

- Los pesos de las aristas representan la capacidad máxima de un canal:
  - Velocidad de una conexión
  - Volumen máximo de agua
  - Cantidad máxima de tráfico
  - Voltaje de una línea eléctrica, etc.

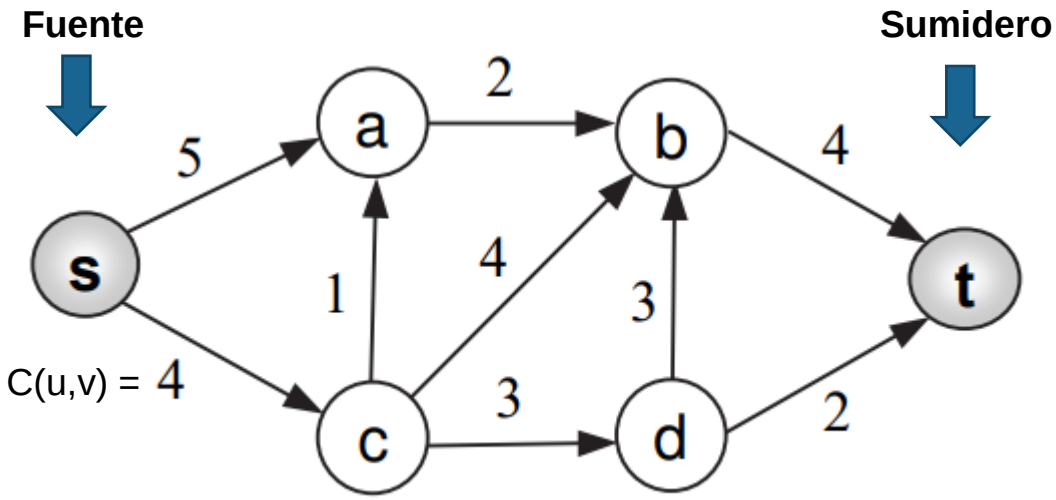
- Pero muchas veces es posible que la cantidad real de flujo de un canal sea menor a la considerada como valor en la arista.

1. Conceptos básicos – Flujos en redes

Características de flujo de una red

- Existe un **nodo inicial** llamado **Origen/Fuente (s)** y un **nodo final** llamado **Destino/Sumidero (t)**, ambos nodos pertenecen a V.
- Entre el origen y el destino existe una cantidad determinada de nodos interconectados entre sí a través de aristas.
- Estos nodos o vértices solo son uniones. No consumen ni crean flujo.
- Cada arista tiene una Capacidad definida no negativa C(u,v).
- La capacidad máxima que puede transportar cada arista entre los nodos que conecta, puede variar de un arista a otra.
- Una arista solo podrá soportar un flujo menor o igual a su capacidad, así, si un flujo mayor quiere discurrir a través de una arista, solo una parte de dicho flujo (de valor igual a la capacidad) viajará a través de ella, y el resto deberá ir por otra arista que salga del mismo nodo. De no haber otra arista, entonces el flujo se verá reducido.

Grafo **G(V,A,W)**



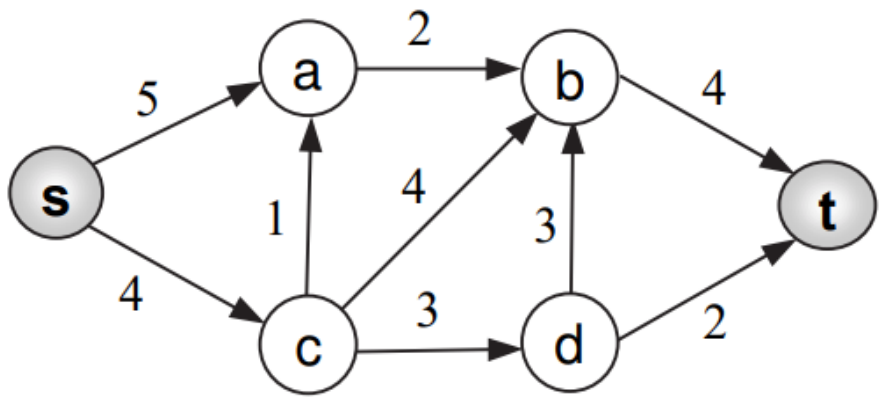
El flujo se preserva en todo momento.

"Todo flujo que entra debe salir"

1. Conceptos básicos – Flujos en redes

Representación del flujo de una red

- El siguiente grafo **G(V,A,W)** representa la circulación de algún elemento, que a través de cada una de sus aristas con una capacidad máxima de canal podrá transportar (fluir) algún contenido.



Tener en cuenta:

- Un nodo de origen **S** tiene todos los bordes salientes y ningún borde entrante.
- El nodo receptor o sumidero **T** tiene todos los bordes entrantes y ningún borde saliente.

Para cualquier red de flujo se debe cumplir:

- Para todos los nodos (excepto el fuente y sumidero), el flujo de entrada debe ser igual al flujo de salida.
- Existe una función F de flujo la cual indica la cantidad de flujo que transcurre por dicha arista (no podemos enviar más flujo a través de un borde que su capacidad).

$$\mathbf{f(u,v)} <= c(u,v)$$

- El nodo de origen es el único que produce flujo. Por lo tanto el flujo total de salida es:

$$\mathbf{|f|} = \text{Sum } f(s, v)$$

La salida total desde el vértice fuente debe ser igual a la entrada total al vértice receptor.

2. El problema del Flujo Máximo en redes

¿En que consiste el problema del flujo máximo?

El **problema del flujo máximo** consiste en **encontrar la máxima cantidad de flujo que puede ser llevado desde el nodo s al nodo t**.

Objetivo

Determinar la máxima cantidad de material u objetos (flujo) que puede fluir en la red desde la fuente (s) hacia el sumidero (t).

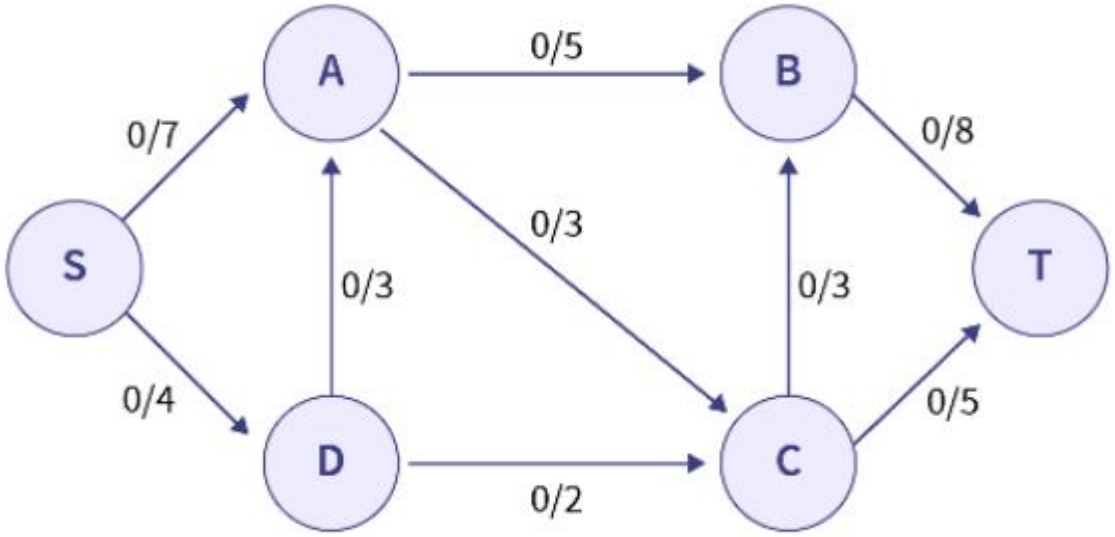
- Será muy importante conocer el valor del flujo máximo porque permite a la fuente saber exactamente cuánto producir y enviar a través de una ruta sin generar desperdicios.

Flujo Máximo

El flujo máximo es el valor máximo posible del flujo de la red donde el flujo se define como "suma de todo el flujo que se produce en la fuentes **S**, o suma de todo el caudal que se consume en el sumidero **t**.

2. El problema del Flujo Máximo en redes

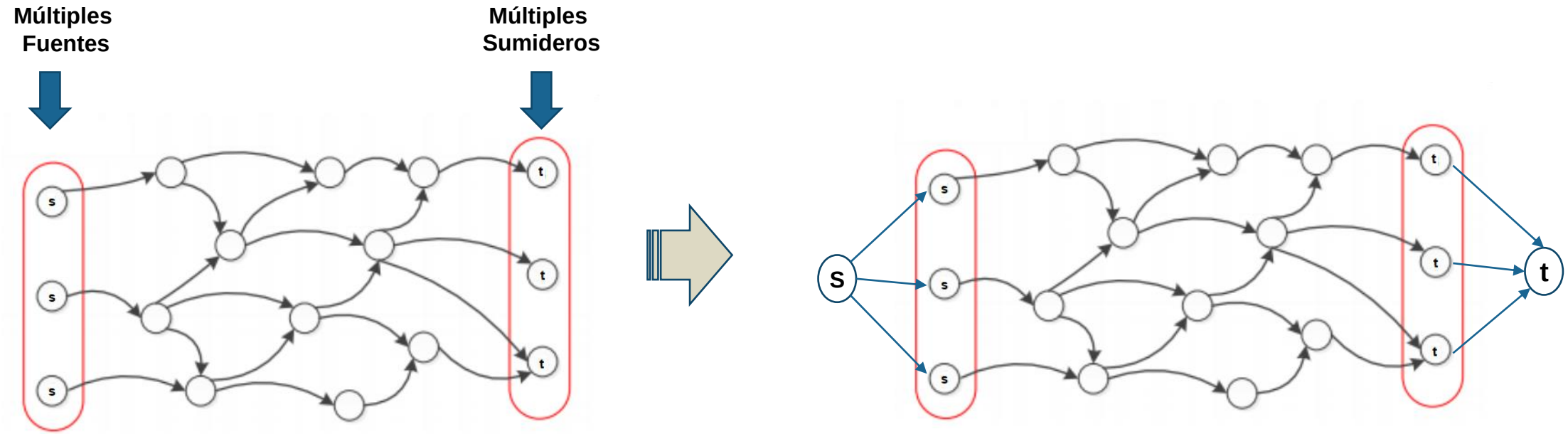
Representación de una red de flujo



- El primer valor de cada borde o arista representa la cantidad de flujo que lo atraviesa (que inicialmente se establece en 0).
- El segundo valor representa la capacidad real del borde o arista.

2. El problema del Flujo Máximo en redes

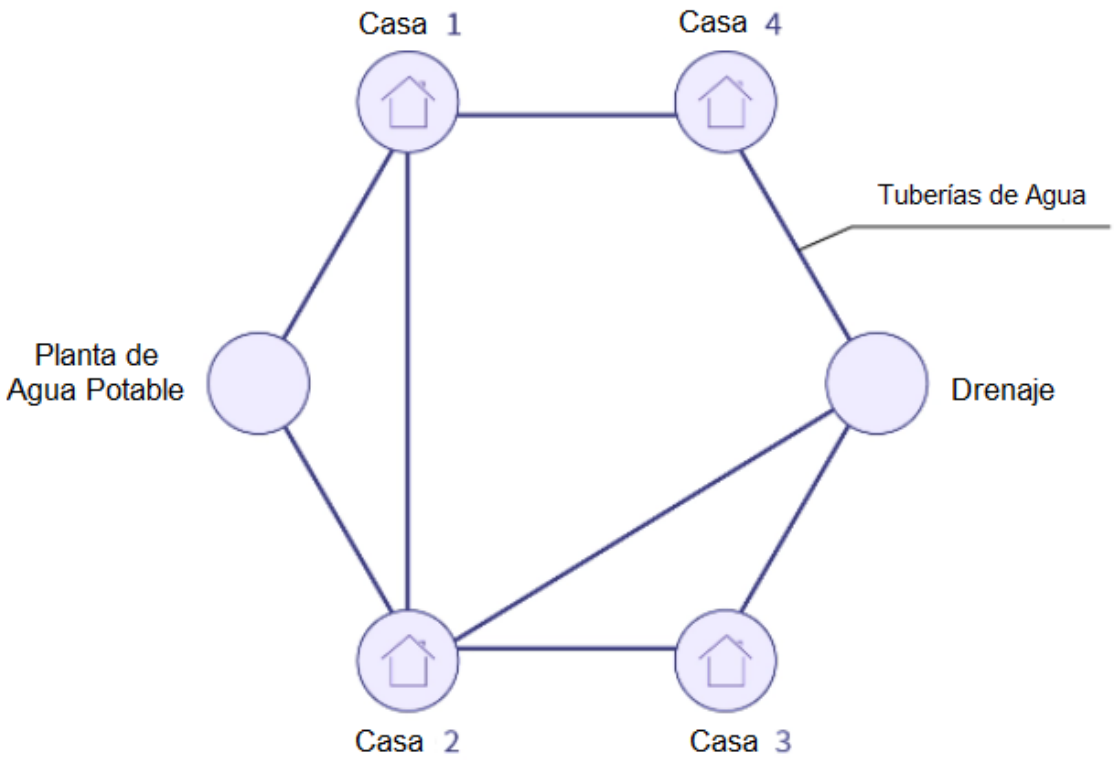
Representación de una red de flujo



- Cualquiera que sea la cantidad de orígenes o destinos, esta situación se puede reducir a un problema de flujo máximo ordinario (una sola fuente y un único sumidero).

2. El problema del Flujo Máximo en redes

Problema de la vida real de una Red de Flujo



- Podemos visualizar todos los bordes o aristas como tuberías de agua, donde su capacidad (aquí tubería) es la cantidad máxima de agua que puede fluir a través de la arista por unidad de tiempo.
- Aquí el caudal máximo es análogo a la planta de agua donde **S** es análoga a la planta de drenaje.

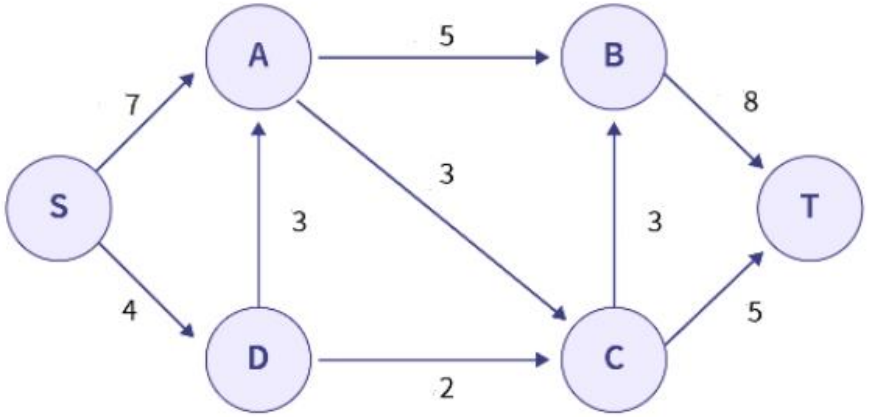
Cumple las condiciones:

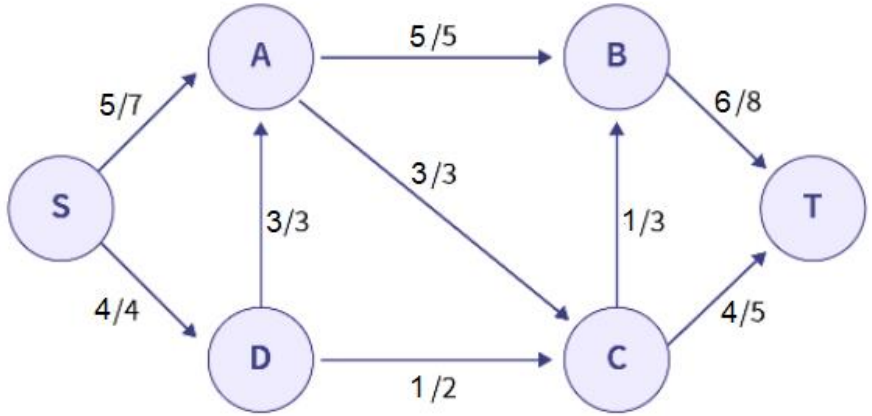
- Más agua que la capacidad de la tubería no puede fluir a través de ella.
- La entrada y salida de agua en cada nodo (casa) debe ser igual ya que el agua no puede desaparecer o aparecer mágicamente.

2. El problema del Flujo Máximo en redes

Ejemplo de representación del flujo máximo en una red

Grafo G(V,A)





Donde:

- G (V, A) es una red de transporte.
- Cada arista (u, v) tiene capacidad q y flujo f.
- Asumimos siempre que el flujo es compatible.

Flujo/Capacidad

$f(u,v) <= c(u,v)$

3. Algoritmo Ford – Fulkerson

¿Qué algoritmos solucionan el problema del Flujo Máximo?

- El principal método de solución del problema del flujo máximo es el **Algoritmo de Ford-Fulkerson.**

En 1956, con el problema de flujo máximo Ford-Fulkerson establecieron el famoso teorema del flujo máximo - mínimo corte.

Lester Randolph Ford Jr.

Delbert Ray Fulkerson

Es un algoritmo codicioso (greedy) que implica:

- Encontrar una ruta desde el nodo de origen (s) al nodo sumidero (t)
- Y si el camino existe entonces aumentamos el flujo tanto como sea posible para que a lo largo de cualquier borde o arista no exceda su capacidad.

3. Algoritmo Ford – Fulkerson

Veamos el algoritmo

E = número de aristas en la red de flujo.

f (e)= flujo a lo largo del borde o arista.

c (e)= Capacidad del borde o arista.

- Inicializar maxFlow con 0, es decir, **maxFlow=0**
- Usar un algoritmo **BFS/DFS** para buscar un camino **P** entre los nodos **s** y **t**. La existencia de **P**, significaría que por cada arista **e**, se cumple que f(e) < c (e). Si el camino **P** existe entonces:
  - **flowCanBeAdded** = min(c(e)−f(e)) para cada arista **e** en **P**.
  - **maxFlow** = maxFlow + flowCanBeAdded
  - Por cada arista e en P, f(e)=f(e)+flowCanBeAdded
- En el otro caso (si no existe ningún camino entre s y t) devolver el maxFlow.

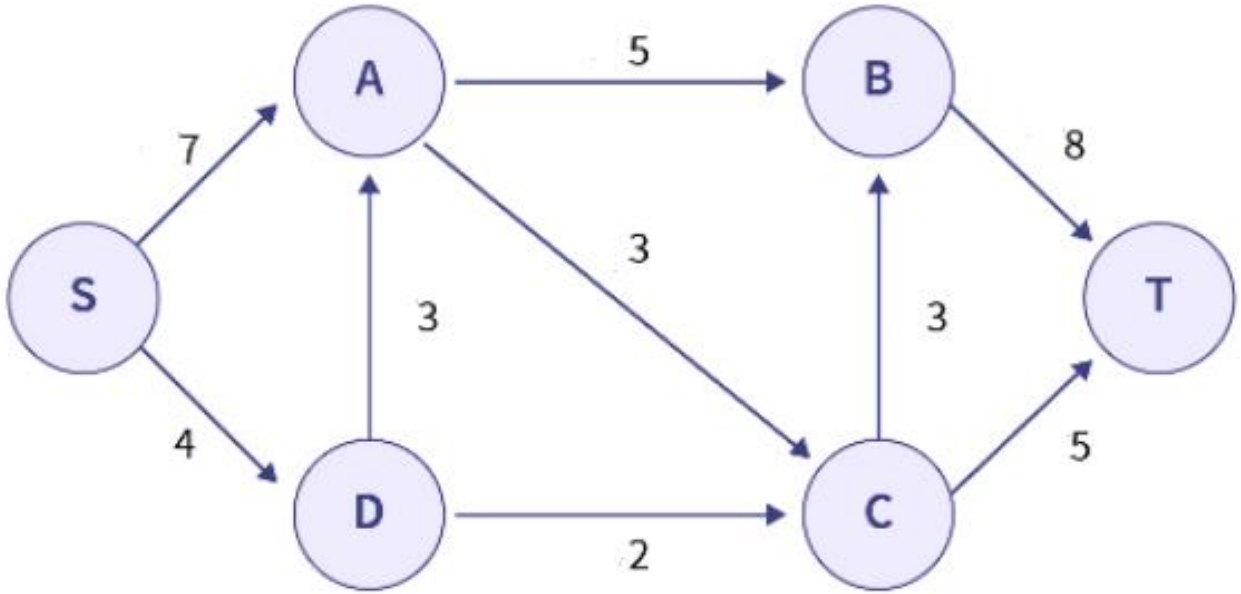
Pseudocodigo

```
FordFulkerson(Graph G, Node s, Node t):
    Initialize flow of all edges e to 0.
    while(there is augmenting path(P) from s to t
    in the residual graph):
        Find augmenting path between s and t.
        Update the residual graph.
        Increase the flow.
    return
```

Veamos el algoritmo en un ejemplo

3. Algoritmo Ford – Fulkerson

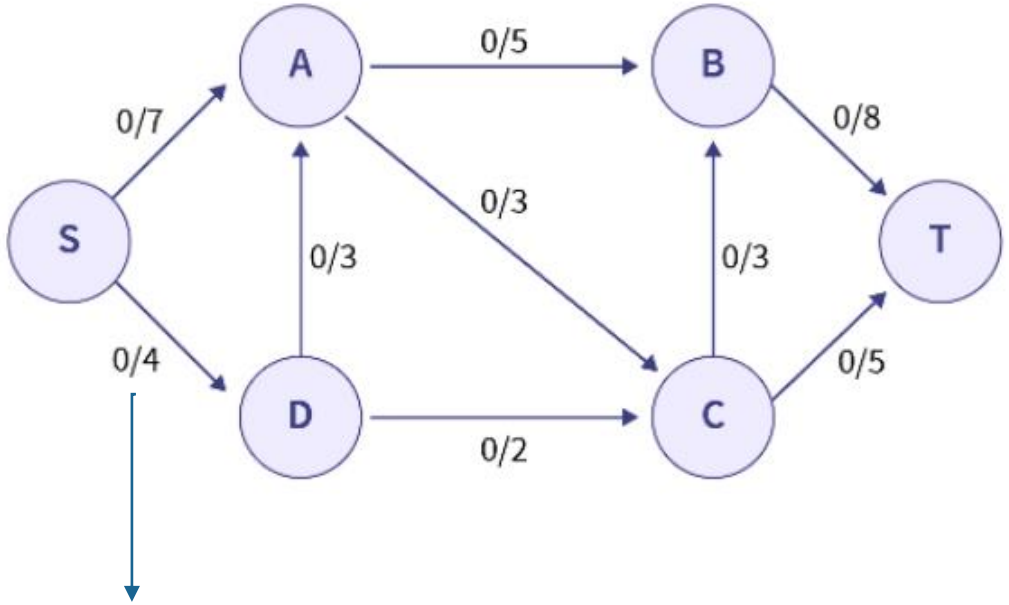
EJEMPLO #1: Encuentre el máximo flujo de **s** a **t** en el siguiente gráfico.



3. Algoritmo Ford – Fulkerson

EJEMPLO #1: Encuentre el máximo flujo de **s** a **t** en el siguiente gráfico.

Grafo residual G'



c(e)−f(e) = Capacidad residual = lo que falta por fluir

- Podemos ver que el **flujo inicial** de todos los caminos es 0.
- Ahora buscaremos una ruta (camino) de aumento en la red.

Camino de aumento

Es una trayectoria desde el nodo fuente **s** al nodo sumidero **t** que puede conducir más flujo.

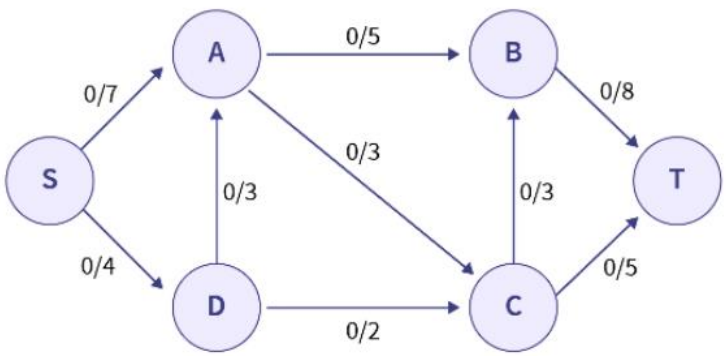
Capacidad residual

Es la capacidad restante después de llevar el flujo

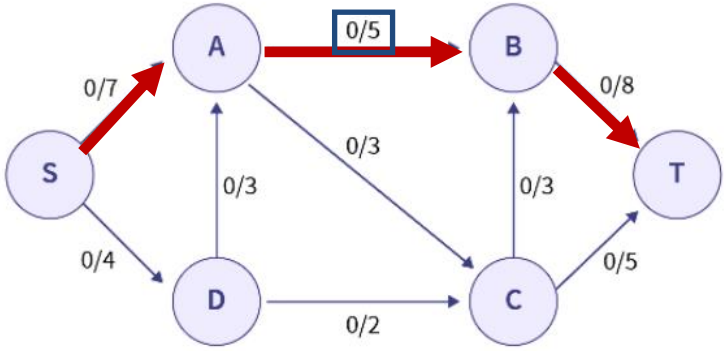
3. Algoritmo Ford – Fulkerson

Uno de esos caminos es el que presenta capacidades residuales como 7, 5, y 8. Su mínimo capacidad residual es 5, por lo que según el método de **Ford-Fulkerson** aumentaremos un flujo de 5 a lo largo de todas las aristas del camino.

Grafo residual G'



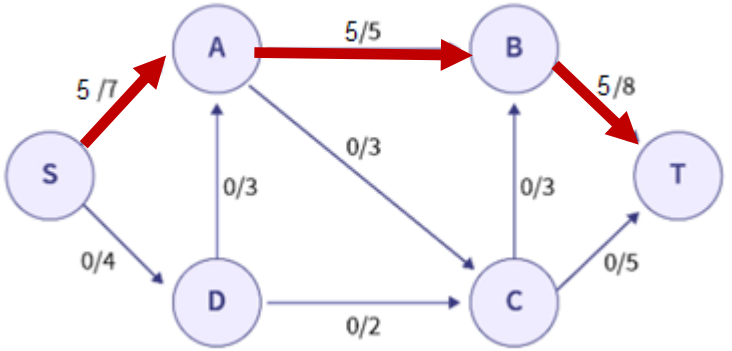
1. Se comienza con f(u,v) =0 para cada par de nodos.



2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar un C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e))**

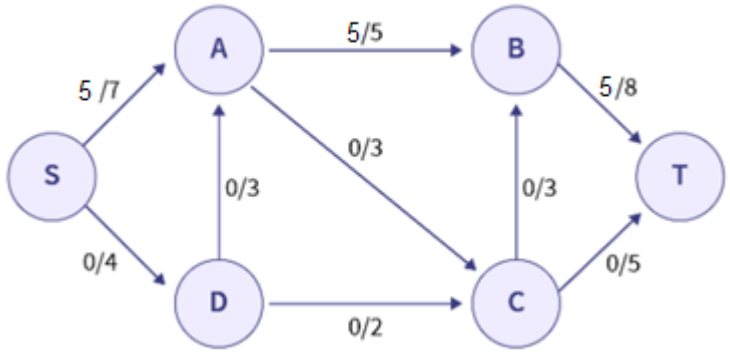


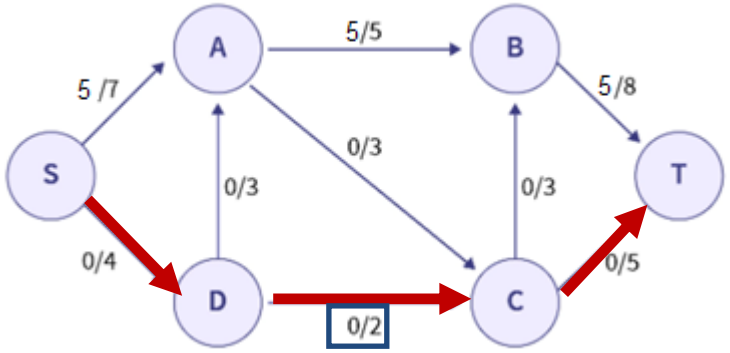
C(p) = 5 acumulamos este flujo en cada arista del camino.

3. Algoritmo Ford – Fulkerson

- Ahora, buscaremos otras posibles rutas (caminos) de aumento, una de esas rutas puede ser: **s→D→C→t** con capacidades residuales como 4, 2, y 5de los cuales 2es el mínimo.
- Entonces aumentaremos un flujo de 2 a lo largo del camino.

Grafo residual G'

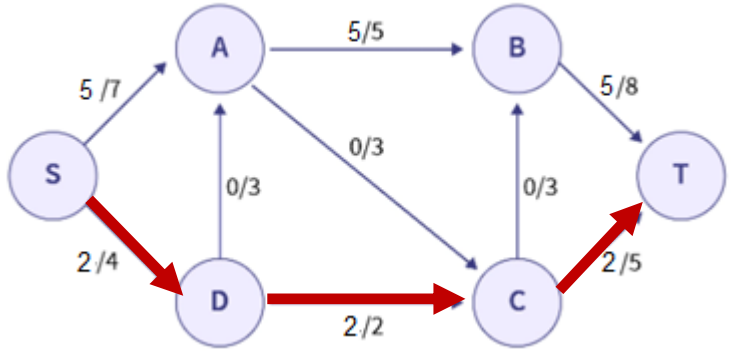




2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar un C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e))**

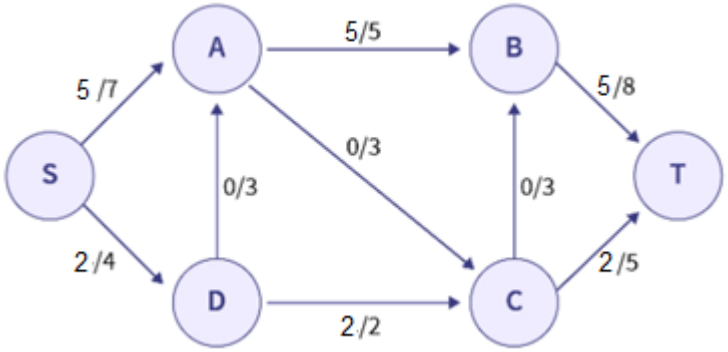


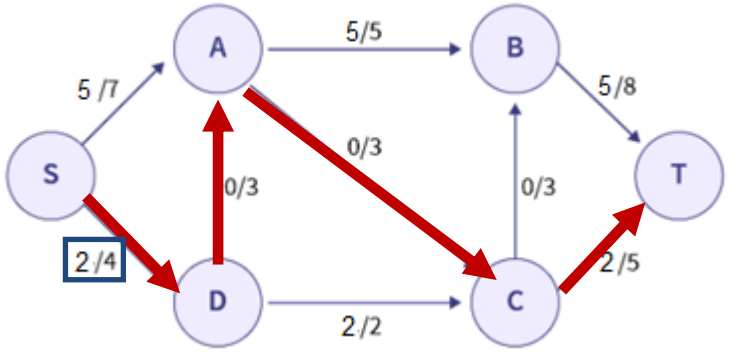
C(p) = 2 acumulamos este flujo en cada arista del camino.

3. Algoritmo Ford – Fulkerson

- Uno de esos caminos puede ser **s→D→A→C→t** con capacidades 2,3,3, y 3 de los cuales 2 es el mínimo por lo que aumentará el flujo en 2 a lo largo del camino.

Grafo residual G'

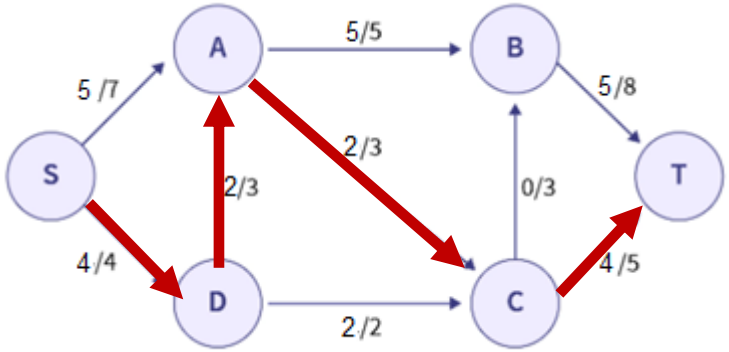




2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar un C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e))**

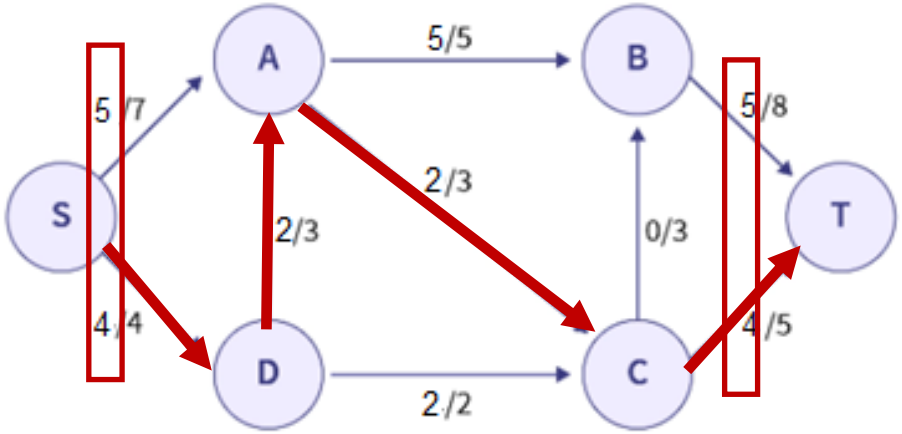


C(p) = 2 acumulamos este flujo en cada arista del camino.

3. Algoritmo Ford – Fulkerson

- En este ultimo grafo residual, podemos observar que hemos alcanzado un flujo máximo en las aristas que unen los nodos s->t.

Grafo residual G'

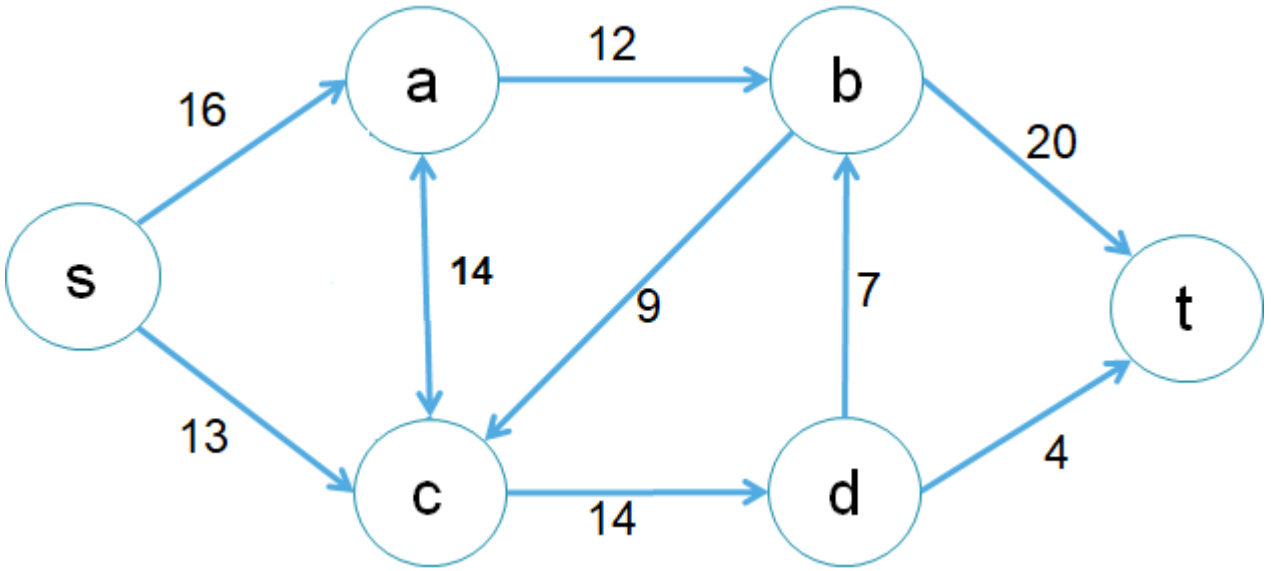


Ahora no hay más rutas de aumento posibles en la red, por lo que el flujo máximo es el flujo total que sale de la fuente o el flujo total que entra en el sumidero, es decir

Flujo máximo = 5 + 4 = 9 = Flujo total 5+4 (que sale de la fuente **s**) = Flujo total 5 + 4 (que entra al sumidero **t**)

3. Algoritmo Ford – Fulkerson

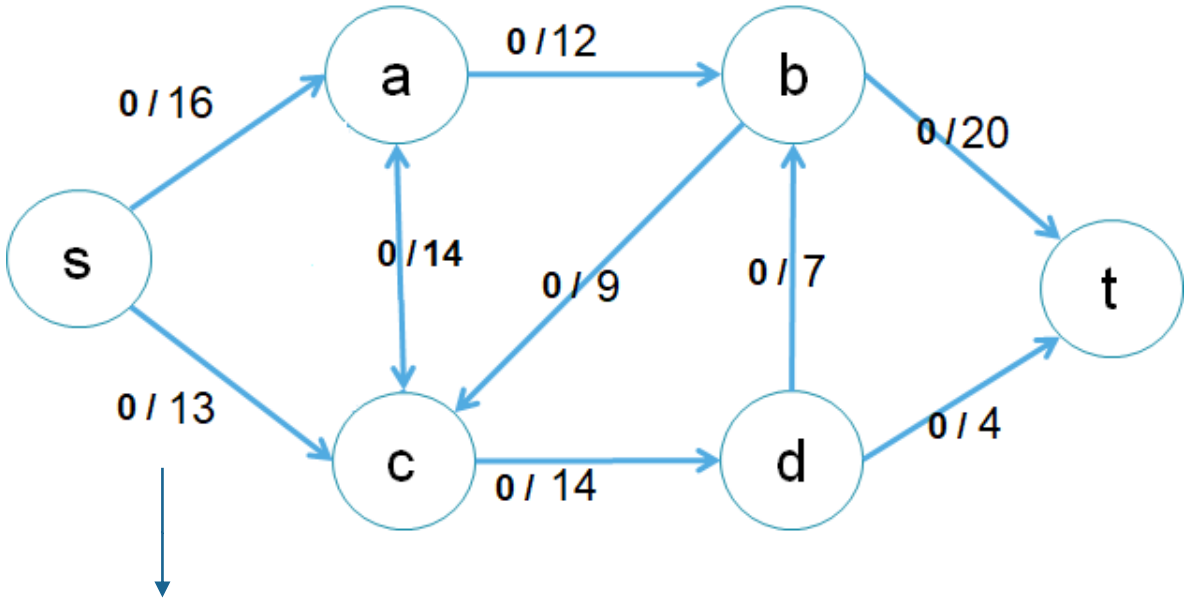
EJEMPLO #2: Encuentre el máximo flujo de **s** a **t** en el siguiente gráfico.



3. Algoritmo Ford – Fulkerson

EJEMPLO #2: Encuentre el máximo flujo de **s** a **t** en el siguiente gráfico.

Grafo residual G'



c(e)−f(e) = Capacidad residual = lo que falta por fluir

- Podemos ver que el **flujo inicial** de todos los caminos es 0.
- Ahora buscaremos una ruta (camino) de aumento en la red.

Camino de aumento

Es una trayectoria desde el nodo fuente **s** al nodo sumidero **t** que puede conducir más flujo.

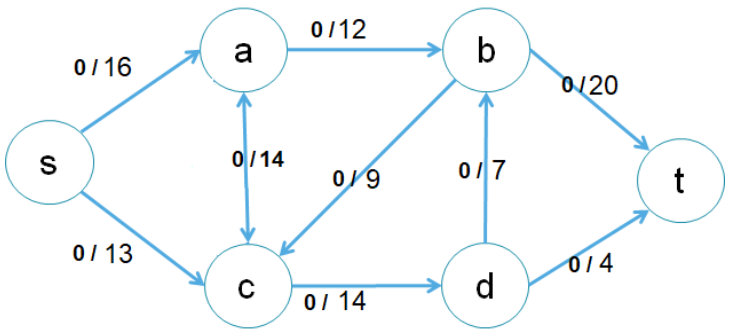
Capacidad residual

Es la capacidad restante después de llevar el flujo

3. Algoritmo Ford – Fulkerson

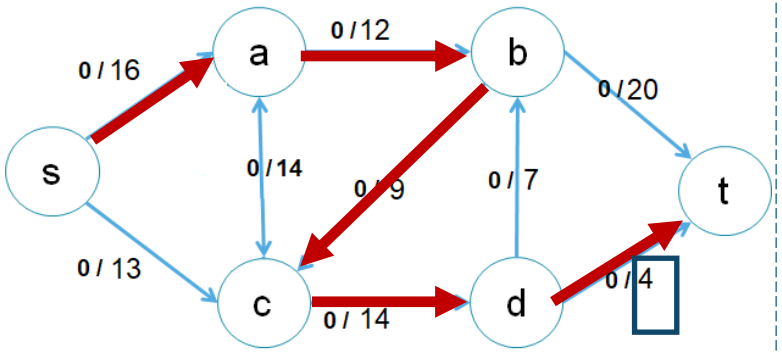
EJEMPLO #2: Encuentre el máximo flujo de **s** a **t** en el siguiente gráfico.

Grafo residual G'



1. Se comienza con f(u,v) =0 para cada par de nodos.

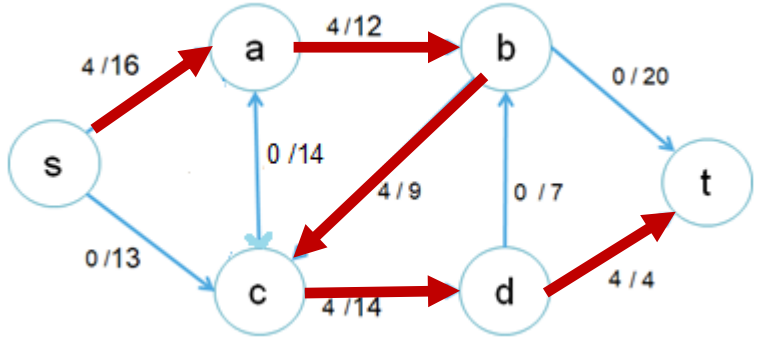
- Encontrar un camino de **s** a **t**



2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar un C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e))**

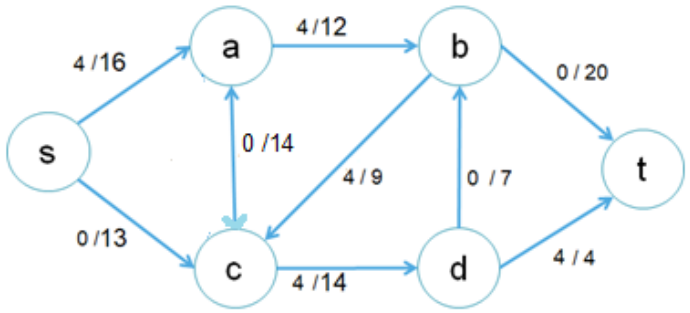


C(p) = 4 acumulamos este flujo en cada arista del camino.

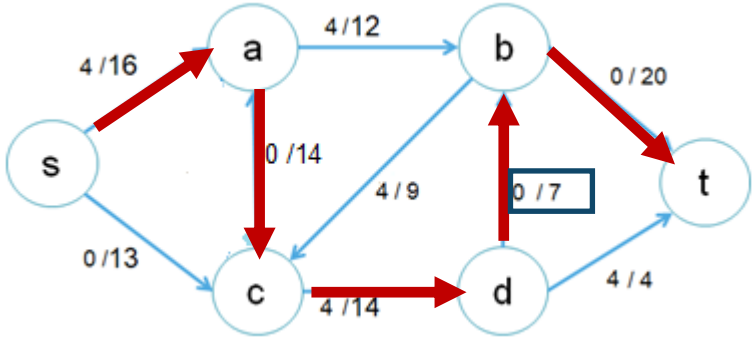
3. Algoritmo Ford – Fulkerson

3. Se repite el proceso previo hasta no encontrar un camino de aumento.

Grafo residual G'



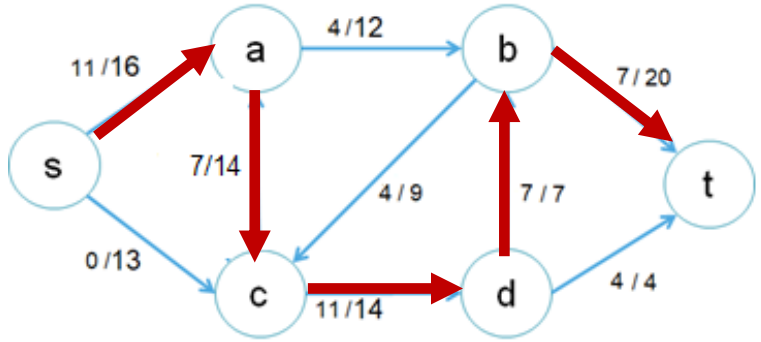
- Encontrar un camino de **s** a **t**



2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e)) = 7 – 0 = 7**

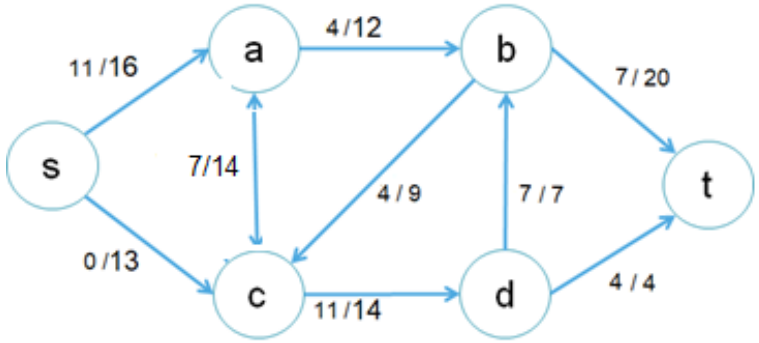


C(p) = 7 acumulamos este flujo en cada arista del camino.

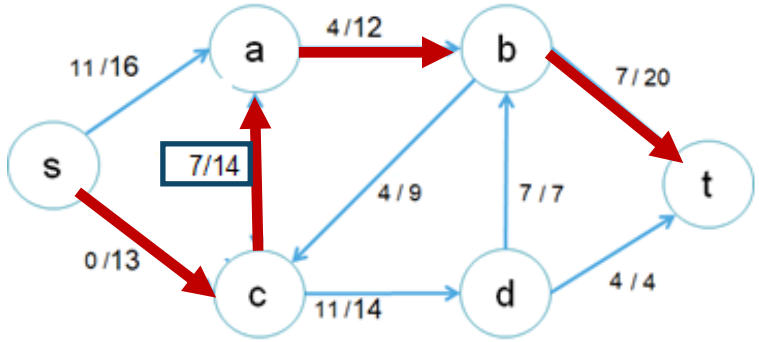
3. Algoritmo Ford – Fulkerson

3. Se repite el proceso previo hasta no encontrar un camino de aumento.

Grafo residual G'



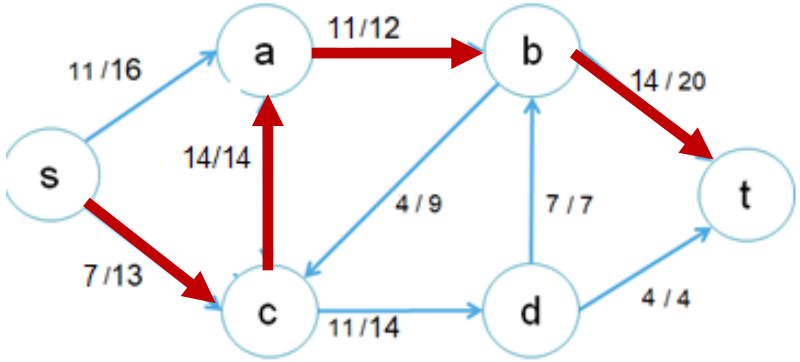
- Encontrar un camino de **s** a **t**



2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e)) = 14 – 7 = 7**

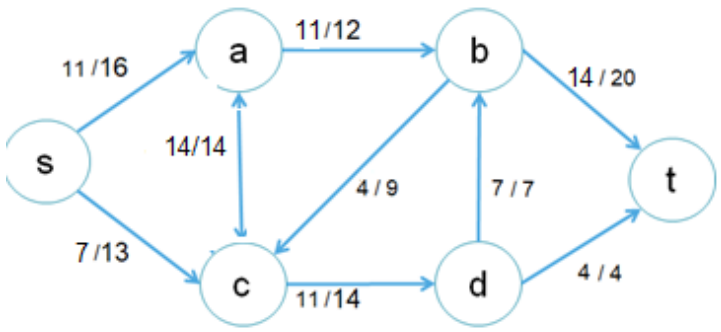


C(p) = 7 acumulamos este flujo en cada arista del camino.

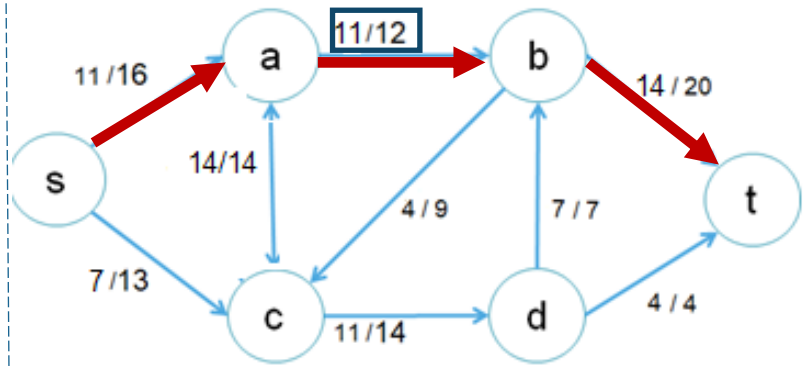
3. Algoritmo Ford – Fulkerson

3. Se repite el proceso previo hasta no encontrar un camino de aumento.

Grafo residual G'



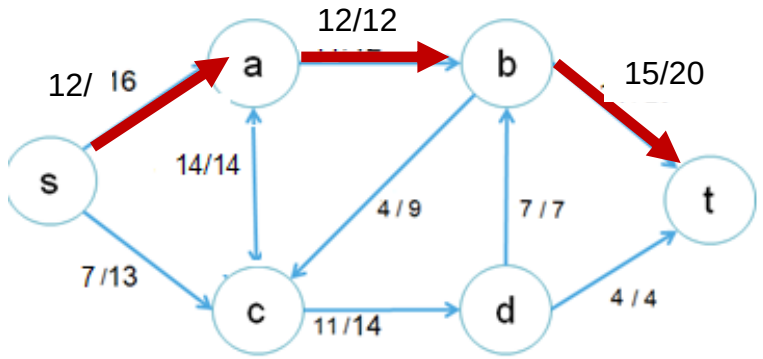
- Encontrar un camino de **s** a **t**



2. En cada iteración se incrementa el valor del flujo buscando un **camino de aumento** (seleccionando el camino que lleve mas flujo).

Hallar C(p)

donde **C(p)** sea el mínimo valor en las aristas del camino **min(c(e)−f(e)) = 12 – 11 = 1**



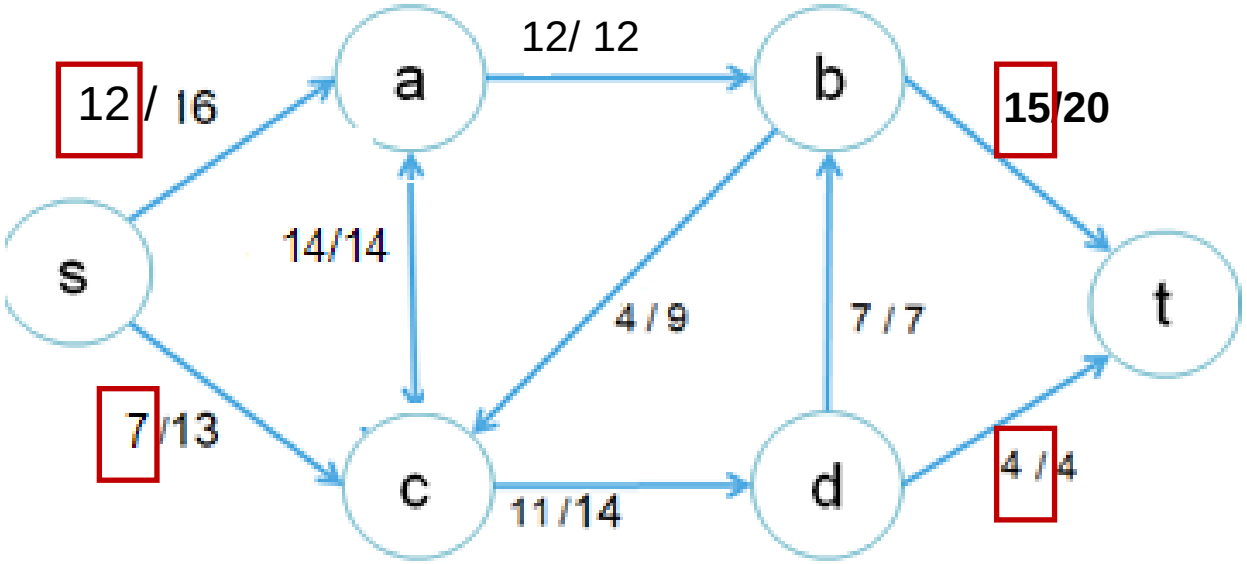
S-> a = 12/16 b-> t = 15/20

a-> b = 12/12

C(p) = 1 acumulamos este flujo en cada arista del camino.

3. Algoritmo Ford – Fulkerson

Grafo residual G'



Flujo máximo = 15 + 4 = 19

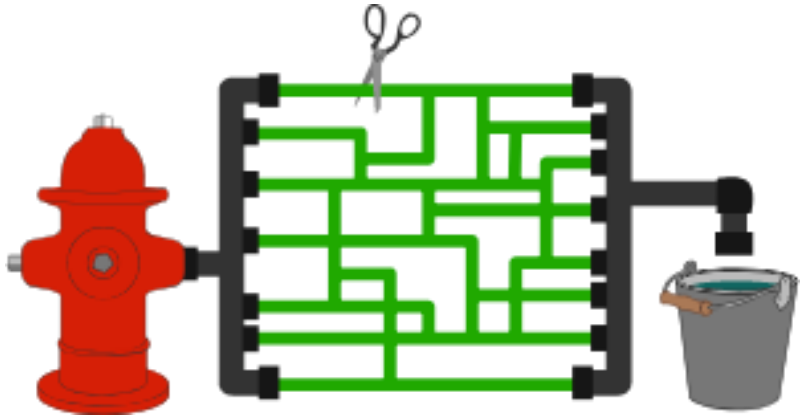
Ahora no hay más rutas de aumento posibles en la red, por lo que el flujo máximo es el flujo total que sale de la fuente o el flujo total que entra en el sumidero, es decir

Flujo máximo = 15 + 4 = 19 = Flujo total 12+7 (que sale de la fuente **s**) = Flujo total 15 + 4 (que entra al sumidero **t**)

3. Aplicaciones del Flujo Máximo en Redes

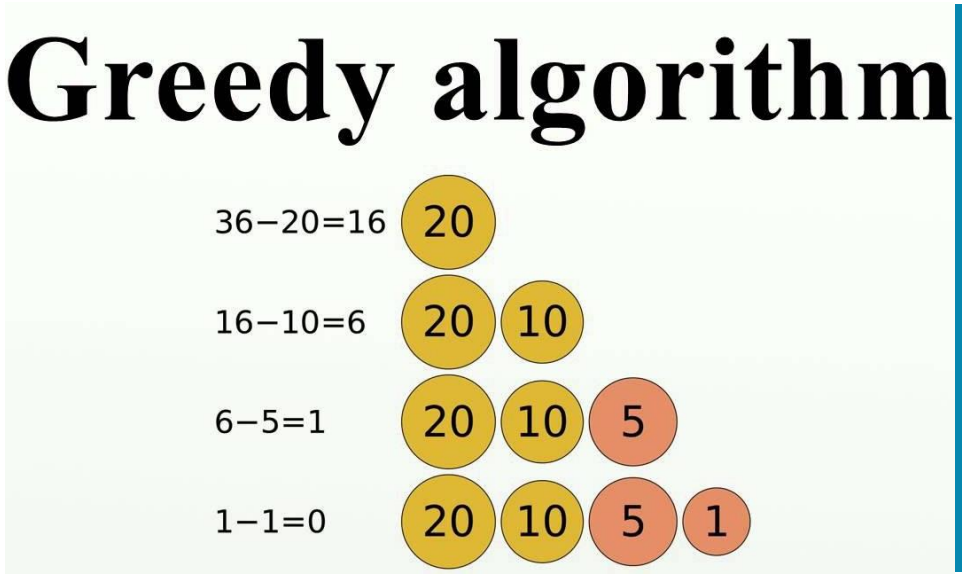
Principales aplicaciones

- En movimientos de tráfico, para encontrar cuánto tráfico puede moverse de una ciudad A a la ciudad B.
- En los sistemas eléctricos, para encontrar la corriente máxima que podría circular por los cables sin provocar ningún cortocircuito.
- En el sistema de gestión de agua/alcantarillado, para encontrar la capacidad máxima de la red.



PREGUNTAS

Dudas y opiniones



Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 12: Algoritmos Voraces y Programación Dinámica

Complejidad Algorítmica

Semana 12

Objetivos

- Entender como trabajan los Algoritmos Voraces
- Manejar y desarrollar los principios del paradigma de programación dinámica.

MÓDULO 12: Algoritmos Voraces y Programación Dinámica

Contenido

1. Conceptos básicos – Algoritmos Voraces
2. Ventajas y desventajas del enfoque codicioso
3. El enfoque codicioso en un ejemplo
4. Pasos de un algoritmo voraz
5. Tipos de Algoritmos Voraces
6. Programación Dinámica
7. Diferencia entre Programación Dinámica y Codiciosa

Preguntas / Conclusiones

1. Conceptos básicos – Algoritmos Voraces

¿Por qué un **algoritmo** se considera **voraz o codicioso**?

Algoritmo: secuencia de pasos que conducen a la solución de un problema.

Voraz o codicioso: un enfoque que para resolver un problema selecciona la mejor opción disponible en ese momento.

Características de los Algoritmos Codiciosos

- No se preocupan si el mejor resultado actual traerá luego el resultado óptimo general.
- Un algoritmo voraz nunca revierte la decisión anterior, incluso si la elección es incorrecta. Funciona en un enfoque de arriba hacia abajo.
- Es posible que este algoritmo no produzca el mejor resultado para todos los problemas. Esto sucede porque siempre busca la mejor opción local para producir el mejor resultado global.

¿Es posible utilizar un Algoritmo Voraz para solucionar cualquier tipo de problema?

1. Conceptos básicos – Algoritmos Voraces

¿Cuándo aplicar un **algoritmo voraz o codicioso** en la solución de un problema?

Podremos determinar si un algoritmo codicioso se puede usar con cualquier problema si el problema tiene las siguientes dos propiedades:

1. **Propiedad de elección codiciosa**

Si se puede encontrar una solución óptima al problema eligiendo la mejor opción en cada paso sin reconsiderar los pasos anteriores una vez elegidos, **entonces** el problema se puede resolver utilizando un enfoque codicioso.

2. **Subestructura óptima**

Si **la solución general óptima del problema corresponde a la solución óptima de sus subproblemas**, **entonces** el problema se puede resolver utilizando un enfoque codicioso.

2. Ventajas y desventajas del enfoque codicioso

VENTAJAS

- El algoritmo es más fácil de describir.
- Este algoritmo puede funcionar mejor que otros algoritmos (pero no en todos los casos).

DESVENTAJAS

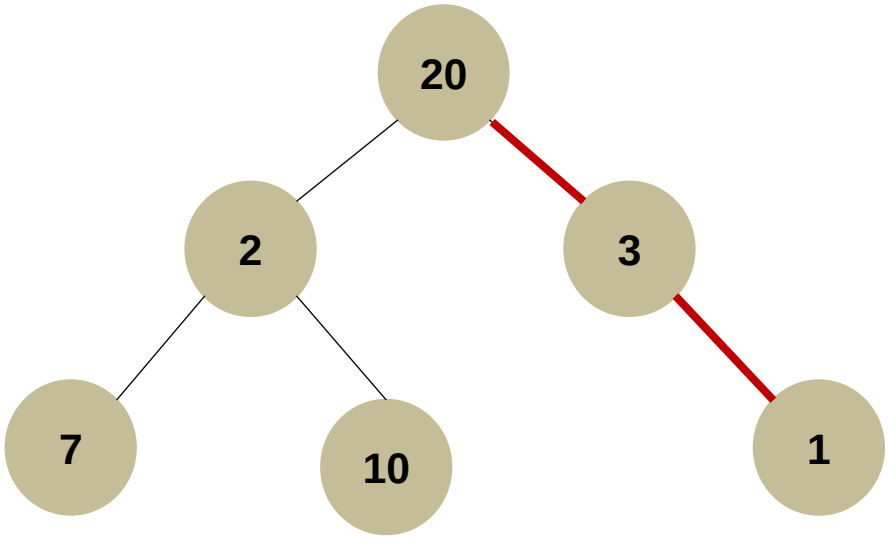
- El algoritmo codicioso no siempre produce la solución óptima. Esta es la principal desventaja del algoritmo.

Veamos el enfoque codicioso en uno ejemplos

3. El enfoque codicioso en un ejemplo

Ejemplo #1

- Aplicar el enfoque codicioso al siguiente árbol desde la raíz hasta la hoja para encontrar la ruta mas larga.



Enfoque codicioso:

- Comencemos con el nodo raíz 20 . El peso del nodo derecho es 3 y el peso del nodo izquierdo es 2.
- Nuestro problema es encontrar el **camino más largo**. Y, la solución óptima en este momento es 3 . Entonces, el algoritmo codicioso elegirá 3 .
- Finalmente el peso del nodo/hijo único de 3 es 1 . Esto nos da un resultado final 20 + 3 + 1 = 24.

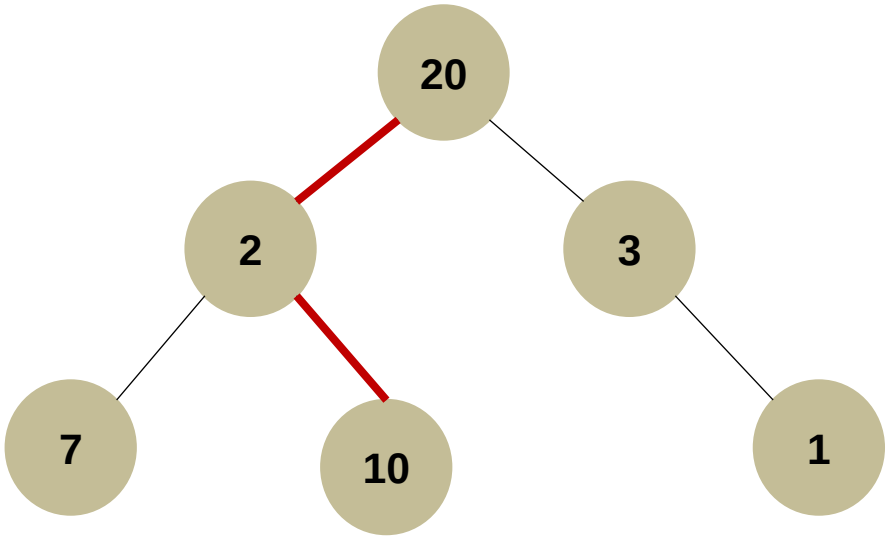
Sin embargo, no es la solución óptima... ¿Verdad?

Hay otro camino que tiene más peso, ¿Cuál es?

3. El enfoque codicioso en un ejemplo

Ejemplo #1

- Aplicar el enfoque codicioso al siguiente árbol desde la raíz hasta la hoja para encontrar la ruta mas larga.



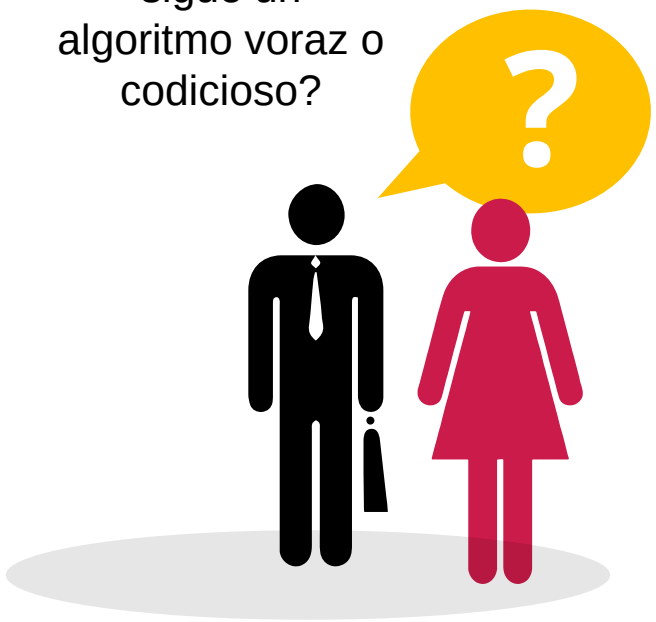
Existe un camino con mayor peso

- 20 + 2 + 10 = 32 como se muestra en la imagen.

Por lo tanto, los algoritmos codiciosos no siempre dan una solución óptima/factible.

4. Pasos de un algoritmo voraz

¿Qué pasos sigue un algoritmo voraz o codicioso?



Algoritmo codicioso

1. Para empezar, el conjunto de soluciones (que contiene las respuestas) está vacío.
2. En cada paso, se agrega un elemento al conjunto de soluciones hasta que se llega a una solución.
3. Si el conjunto de soluciones es factible, se mantiene el elemento actual.
4. De lo contrario, el elemento se rechaza y no se vuelve a considerar nunca más.

Ahora usemos este algoritmo para resolver un problema!

4. Pasos de un algoritmo voraz

PROBLEMA:

- ❑ Hacer un cambio de una cantidad utilizando el menor número de monedas posible.

Monto: S/.18

- ❑ Las monedas disponibles son monedas de:
  - S/. 5
  - S/. 2
  - S/. 1

- ❑ No hay límite para el número de cada moneda que puede usar.

4. Pasos de un algoritmo voraz

SOLUCION:

1. Se crea un conjunto-solución vacío **solution-set = { }**. Y las monedas disponibles son **{5, 2, 1}**.
2. Debemos encontrar el sum = 18. Entonces, comencemos con sum = 0.
3. Se selecciona siempre la moneda con el mayor valor (es decir, 5) hasta cuya suma no sea sum > 18. Cuando seleccionamos el valor más grande en cada paso, esperamos llegar al destino más rápido. Recordemos que este concepto se llama **propiedad de elección codiciosa**.
   - En la primera iteración, **solution-set = {5}** y sum = 5.
   - En la segunda iteración, **solution-set = {5, 5}** y sum = 10.
   - En la tercera iteración, **solution-set = {5, 5, 5}** y sum = 15.
   - En la cuarta iteración, **solution-set = {5, 5, 5, 2}** y sum = 17
     (no podemos seleccionar 5 aquí porque si lo hacemos, sum = 20 y seria mayor que 18, entonces, seleccionamos el segundo elemento más grande que es 2).
   - De manera similar, en la quinta iteración, se selecciona 1.
4. Ahora sum = 18 y **solution-set = {5, 5, 5, 2, 1}**.

5. Tipos de Algoritmos Voraces

¿Qué algoritmos son de tipo voraz o codicioso?

Existen diferentes tipos de algoritmos codiciosos. Entre ellos se pueden señalar los siguientes:

Algoritmos codiciosos estándar:

- Clasificación de selección.
- Problema de programación de trabajos.
- Codificación de Huffman.

Algoritmos codiciosos en grafos:

- **Árbol de expansión mínimo (MST)**
- **Algoritmo de árbol de expansión mínimo de Prim**
- **Algoritmo de árbol de expansión mínimo de Kruskal**
- Algoritmo de árbol de expansión mínimo de Dijkstra
- **Algoritmo de Ford-Fulkerson (Flujo Máximo en Redes)**

Algoritmos codiciosos para casos especiales de problemas de Programación Dinámica:

- Problema de mochila fraccionada.
- Número mínimo de monedas requeridas.

6. Programación Dinámica

¿De qué se trata la **Programación Dinámica** o DP?

Programación Dinámica (DP) como Divide y Vencerás, combinan soluciones a subproblemas para hallar una solución.

La programación dinámica es el algoritmo que divide el problema en varios subproblemas para encontrar la solución óptima.

Divide y Vencerás:

- Divide un problema en partes independientes.
- Resuelve cada parte.
- Combina las soluciones para resolver el problema original.

Algoritmo Divide y Vencerás

- **Divide el problema en subproblemas independientes** para resolver los subproblemas recursivamente y luego combinan sus soluciones para resolver el problema original.

Algoritmo de Programación Dinámica

- Es **aplicable cuando el subproblema no es independiente**, es decir, cuando los subproblemas comparten subproblemas.
- Por lo tanto, **un algoritmo de programación dinámica resuelve cada subproblema solo una vez** y luego guarda sus respuestas en una tabla, evitando así el trabajo de volver a calcular la respuesta cada vez que se encuentra el subproblema.

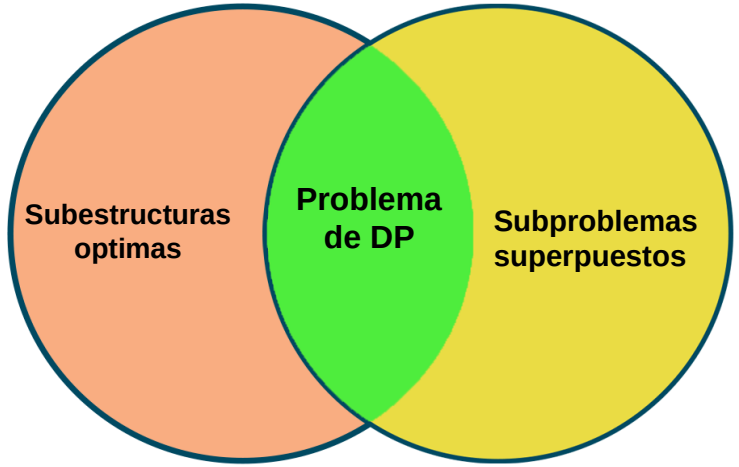
6. Programación Dinámica

¿De qué se trata la **Programación Dinámica** o DP?

A continuación, entenderemos sus características y elementos.

CARACTERÍSTICAS DE LA PROGRAMACIÓN DINÁMICA

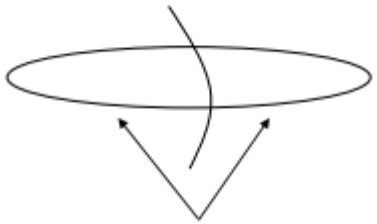
- **Utiliza dos enfoques para resolver un problema de optimización**:



6. Programación Dinámica

CARACTERÍSTICA DP #1

1. **Subestructuras optimas**



Cada subestructura es óptima
(Principio de optimalidad)

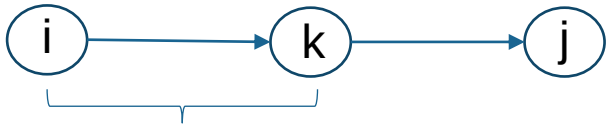
El **algoritmo de programación dinámica** obtiene la solución utilizando el principio de optimalidad.

¿Qué es el Principio de Optimalidad?

- El **principio de optimalidad** establece que **"en una secuencia óptima de decisiones o elecciones, cada subsecuencia también debe ser óptima"**.
- Cuando no es posible aplicar el principio de optimización, es casi imposible obtener la solución utilizando el enfoque de programación dinámica.

El **principio de optimización**:

"Si **k** es un nodo en el camino más corto para ir de **i** a **j**, entonces la parte del camino **de i a k**, y la parte **de k a j**, también deben ser óptimas".

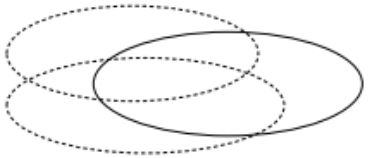


Parte del camino optimo

6. Programación Dinámica

CARACTERÍSTICA DP #2

2. Problemas que se sobreponen



Los subproblemas son dependientes o Subproblemas superpuestos
(Si no fuera así, se trata de Divide y Vencerás - D&C porque los subproblemas que resuelve son independientes)

¿Qué son los subproblemas superpuestos?

- El problema se llama **subproblemas superpuestos** si el algoritmo recursivo visita repetidamente los mismos subproblemas.

- Si algún problema tiene **subproblemas superpuestos**, podemos mejorar la implementación repetitiva de los subproblemas calculándolos solo una vez.

- Si el problema no tiene subproblemas superpuestos, entonces no es factible usar un algoritmo de programación dinámica para resolver el problema.

6. Programación Dinámica

ELEMENTOS DE LA PROGRAMACION DINAMICA

- Como ya mencionamos, el algoritmo DP divide el problema en varios subproblemas para encontrar la solución óptima.
- En DP, el **problema se divide en 3 elementos en total para obtener el resultado final**. Estos elementos son:

1. Subestructura

- La sub estructuración es el proceso de dividir el enunciado del problema dado en **subproblemas más pequeños**.
- Aquí logramos identificar la solución del problema original en términos de la solución de subproblemas.

2. Estructura de la tabla

- Es necesario almacenar la solución del subproblema en una tabla después de resolverlo.
- La programación dinámica **reutiliza** las soluciones de los subproblemas muchas veces para que no tengamos que resolver repetidamente el mismo problema, una y otra vez.

3. Enfoque de abajo hacia arriba

- Es el proceso de combinar las soluciones de los subproblemas para lograr el resultado final utilizando la tabla.
- El proceso comienza resolviendo el subproblema más pequeño y luego combinando su solución con los subproblemas de tamaño creciente hasta obtener la solución final del problema original.

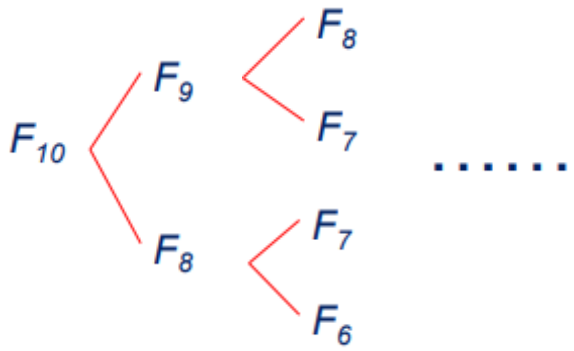
6. Programación Dinámica

EJEMPLO: PROBLEMA NÚMEROS FIBONACCI

- Se define a los números Fibonacci como:

$$\mathbf{F_0 = 0}$$
$$\mathbf{F_1 = 1}$$
$$\mathbf{F_i = F_{i-1} + F_{i-2}} \quad \boldsymbol{para\ i > 0}$$

- ¿Cómo se calcula $\mathbf{F_{10}}$?



¿Se puede aplicar DP en este problema?

- ✓ DP es aplicable cuando los problemas **NO SON INDEPENDIENTES**.
  - Los problemas comparten subproblemas.
- ✓ Números Fibonacci:
  - Recurrencia: $F_i = F_{i-1} + F_{i-2} \quad para\ i > 0$
  - Límites: $F_0 = 0\ F_1 = 1$

- ❑ Un acercamiento D&C resolvería repetidamente los subproblemas comunes.
- ✓ **DP resuelve cada problema 1 vez y lo guarda en una tabla**
- ✓ El uso de la tabla evita el tener que ejecutar re - computaciones.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

6. Programación Dinámica

¿Diferencias entre **Programación Dinámica** y Programación Codiciosa?

Programación dinámica	Programación codiciosa
Tome una decisión en cada paso considerando el problema actual y la solución al problema resuelto previamente para calcular la solución óptima	Hacer la elección que sea mejor en un momento determinado con la esperanza de que conduzca a soluciones óptimas.
Garantía de conseguir la solución óptima	No existe tal garantía de obtener una solución óptima
Más lento en comparación con la programación codiciosa	Más rápido en comparación con la programación dinámica
Basado en la naturaleza recursiva para usar los resultados calculados previamente	Hacer uso de opciones localmente óptimas en cada paso

6. Programación Dinámica

VENTAJAS

- Mediante la programación dinámica se puede obtener una solución óptima tanto local como total (global).
- La programación dinámica reduce las líneas del código.

DESVENTAJAS

- La programación dinámica hace que la utilización de la memoria sea necesaria.
- A medida que utiliza la tabla mientras resuelve el problema de programación dinámica, necesita mucho espacio de memoria para fines de almacenamiento.

6. Programación Dinámica

APLICACIONES

1. En el problema de la mochila 0/1, que se plantea con más frecuencia en las entrevistas técnicas.
2. Como ayuda para almacenar el problema de la ruta más corta.
3. Se usa ampliamente al resolver un problema de optimización matemática.
4. Se utiliza en control de vuelo y robótica.
5. Se utiliza en algoritmos de enrutamiento.
6. Se utiliza principalmente en redes informáticas y problemas gráficos.

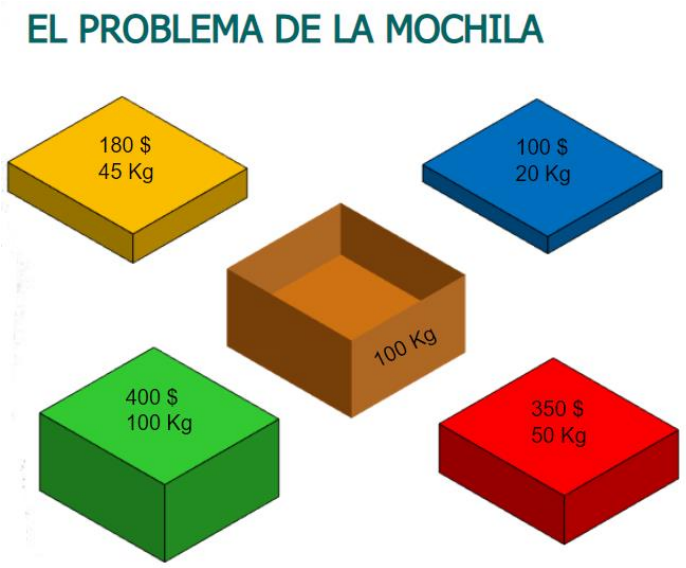
6. Programación Dinámica

CONCLUSIONES

1. La programación dinámica es más importante para optimizar las soluciones del problema en comparación con el enfoque recursivo.
2. Ayuda a reducir las llamadas a funciones repetidas y, por lo tanto, demuestra ser más rápido y efectivo que el enfoque recursivo y divide y vencerás.
3. La idea de simplemente almacenar los resultados de los subproblemas y utilizarlos para calcular el resto del problema hace que el algoritmo de programación dinámica sea un tema diferente e importante en el dominio de la estructura de datos y el algoritmo.
4. Es usual, que en una entrevista técnica se pregunte sobre las ventajas y desventajas de la DP, por ello es muy recomendable aprender y comprender.

PREGUNTAS

Dudas y opiniones



Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 12: Programación Dinámica - Casos de Uso

Ing. Patricia Reyes Silva
pcsiprey@upc.edu.pe

Complejidad Algorítmica

Semana 12

Objetivos

- Examinar casos de uso de la Programación Dinámica

MÓDULO 12: Programación Dinámica - Casos de Uso

Contenido

1. Aplicaciones de la Programación Dinámica
    - **Caso de la Mochila**
    - Caso del Cambio Mínimo de Monedas

Preguntas / Conclusiones

1. Aplicaciones de la Programación Dinámica

¿Qué problemas solucionamos con DP?

Existen muchos problemas que se resuelven utilizando un enfoque de programación dinámica para encontrar la solución óptima.

A continuación, conoceremos los siguientes casos de uso:

- **El problema de la Mochila.**
- El problema del Cambio Mínimo de Monedas.

6. Programación Dinámica

¿Cómo resolver problemas de programación dinámica?

Cuando se trata de encontrar la solución al problema utilizando la programación dinámica, a continuación se detallan algunos pasos que debe considerar seguir:

1) Reconocer el problema de DP

2) Identificar las variables del problema

3) Expresar la relación de recurrencia

4) Identificar el caso base

5) Decidir el enfoque iterativo o recursivo para resolver el problema

6) Añadir memorización

1. Aplicaciones de la Programación Dinámica

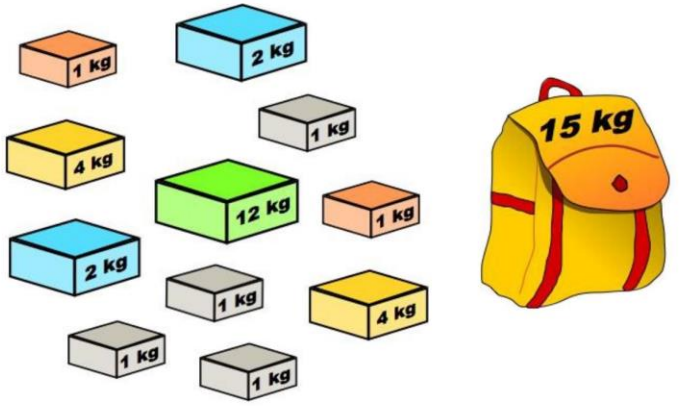
El problema de la Mochila

- El problema de la mochila es el ejemplo perfecto de un **algoritmo de programación dinámica** y la pregunta más frecuente en una entrevista técnica de empresas basadas en productos.

- Como datos tenemos **las ganancias** y **los pesos de N artículos**, y debemos colocar estos artículos en una mochila con la capacidad 'W', por tanto, debemos encontrar (seleccionar) la cantidad de artículos que sea menor o igual a la capacidad de la mochila.

PLANTEAMIENTO DEL PROBLEMA

Dada una bolsa con capacidad **W**, y una lista de artículos junto con sus pesos y ganancias asociadas con ellos.
La tarea es llenar la mochila de manera eficiente, de modo que se logre el máximo beneficio.



1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

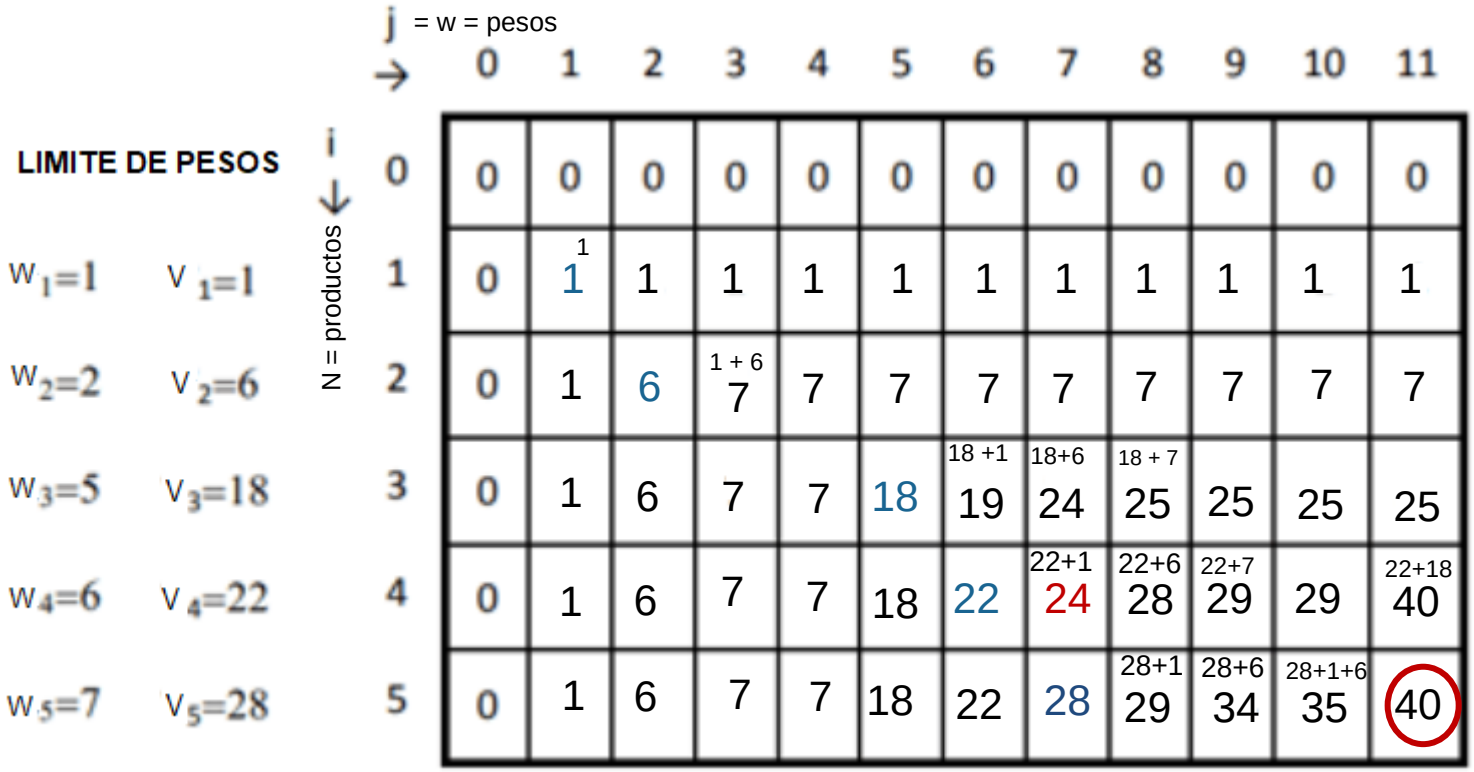
EJEMPLO:

Capacidad de la mochila: 11kg
Nro. Productos: 5

Producto	1	2	3	4	5
Pesos	**1**	**2**	**5**	**6**	**7**
Precios	1	6	18	22	28

Donde:

Capacidad de la Mochila = W
Lista de pesos : wt = []
Lista de precios : pr = []
No. De productos = N



i \ j	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	1 + 6 = 7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	18 +1 = 19	18+6 = 24	18 + 7 = 25	25	25	25
4	0	1	6	7	7	18	22	22+1 = 24	22+6 = 28	22+7 = 29	29	22+18 = 40
5	0	1	6	7	7	18	22	28	28+1 = 29	28+6 = 34	28+1+6 = 35	40

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

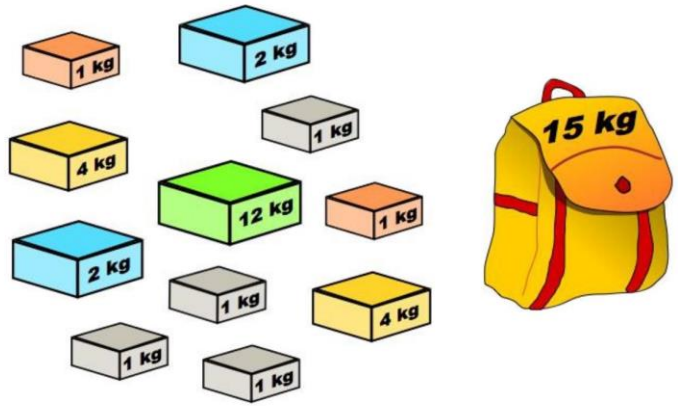
PASOS A SEGUIR

1. Crear una tabla **dp[ ][ ]** y considerar todos los pesos posibles de 1 a W como columnas y pesos posibles a elegir como filas.

2. El estado /celda **dp[i][j]** en la tabla representa la ganancia máxima alcanzable si **'j'** es la capacidad de la mochila y los primeros elementos **'i'** se incluyen en la matriz peso/artículo.
   Por lo tanto, la última celda representará el estado de respuesta.

3. Solo se pueden incluir artículos si su peso es inferior a la capacidad de la mochila.

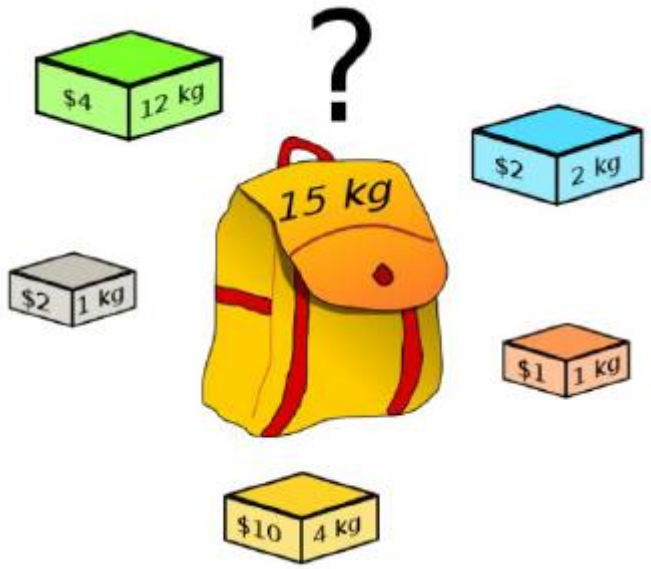
4. Existen dos posibilidades para
   la condición en la que puede completar todas las columnas que tienen 'peso> peso [i-1]'.



1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

EJEMPLO #1: El problema de la Mochila



Estado Inicial: mochila vacía

Estado Final: cualquier combinación de objetos en la mochila

Operadores: meter o sacar objetos de la mochila

Heurística: $max \sum_i Valor_i$ o $max \sum_i \frac{Valor_i}{Peso_i}$

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

EJEMPLO #1: El problema de la Mochila

Capacidad de la mochila = 8kg
Artículos = 6

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	Artículo	Dimensión (Kg)	Ganancia (€)	0	1	2	3	4	5	6	7	8
0	Estado inicial			0	0	0	0	0	0	0	0	0
1	A	1	2									
2	B	2	5									
3	C	4	6									
4	D	5	10									
5	E	7	13									
6	F	8	16									

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #0: El estado de la mochila esta vacía y la ganancia es 0.
Evaluamos 6 etapas porque solo hay 6 objetos (de A-F, existe solo 1 und. x cada objeto).
Los colocamos en la tabla en orden creciente en dimensión con su respectiva ganancia.

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	Artículo	Dimensión (Kg)	Ganancia (€)	0	1	2	3	4	5	6	7	8
0	Estado inicial			0	0	0	0	0	0	0	0	0
1	A	1	2	0	2 A	2 A	2 A	2 A	2 A	2 A	2 A	2 A
2	B	2	5	0	2 A	5 B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #1: Introducimos el objeto A en el casillero de 1kg con la ganancia asociada igual a 2.
Como no hay otro objeto a introducir perteneciente a etapas previas, copiamos esta casilla en las restantes (de la casilla 2 a la de 8kg).

Etapa #2:

- Introducimos el objeto B en el casillero de 2kg con la ganancia asociada igual a 5, manteniendo los valores de las casilla 0-1 de la etapa #1.
- En el siguiente casillero, verificamos si podemos introducir uno o mas objetos que totalicen 3kg => A + B cumplen la condición con una ganancia de 2+5 = 7.
- Colocamos 7 en el casillero 3(kg) haciendo referencia a los objetos A+B y copiamos este valor al resto de casillas porque ya no hay mas objetos que introducir.

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	Artículo	Dimensión (Kg)	Ganancia (€)	0	1	2	3	4	5	6	7	8
0	Estado inicial			0	0	0	0	0	0	0	0	0
1	A	1	2	0	2 A	2 A	2 A	2 A	2 A	2 A	2 A	2 A
2	B	2	5	0	2 A	5 B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B
3	C	4	6	0	2 A	5 B	7 A+B	7 A+B	8 A+C	11 B+C	13 A+B+C	13 A+B+C

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #3: No introducimos el objeto C en el casillero de 4kg con la ganancia asociada igual a 6 porque la ganancia de la etapa #2 para el casillero 4 fue de 7 (mayor a 6, por tanto, lo dejamos en 7).

Evaluamos que otros objetos podemos introducir en los siguientes casilleros mayores a 4kg:

- En la casilla de 5kg podemos colocar los objetos A y C (5Kg) que totalizan una ganancia 8 (mayor a 7 de la etapa anterior)
- En la casilla de 6kg podemos colocar los objetos B y C (6kg) que totalizan una ganancia de 11 (mayor a 7 de la etapa anterior)
- En la casilla de 7kg podemos colocar los objetos A, B y C (7kg) que totalizan una ganancia de 13 (mayor a 7 de la etapa anterior)
- En la casilla de 8kg ya no podemos colocar mas objetos, por tanto copiamos el resultado de la casilla de 7kg.

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	**Artículo**	**Dimensión (Kg)**	**Ganancia (€)**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**Estado inicial**			0	0	0	0	0	0	0	0	0
1	A	1	2	0	2 A	2 A	2 A	2 A	2 A	2 A	2 A	2 A
2	B	2	5	0	2 A	5 B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B
3	C	4	6	0	2 A	5 B	7 A+B	7 A+B	8 A+C	11 B+C	13 A+B+C	13 A+B+C
4	D	5	10	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #4: Introducimos el objeto D en el casillero de 5kg con la ganancia asociada igual a 10, manteniendo los valores de las casilla 0-4 de la etapa #3.

Evaluamos que otros objetos podemos introducir en los siguientes casilleros mayores a 5kg:

- En la casilla de 6kg podemos colocar los objetos A y D (6Kg) que totalizan una ganancia 12 (mayor a 11 de la etapa anterior)
- En la casilla de 7kg podemos colocar los objetos B y D (7kg) que totalizan una ganancia de 15 (mayor a 13 de la etapa anterior)
- En la casilla de 8kg podemos colocar los objetos A, B y D (8kg) que totalizan una ganancia de 17 (mayor a 13 de la etapa anterior)

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	**Artículo**	**Dimensión (Kg)**	**Ganancia (€)**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**Estado inicial**			0	0	0	0	0	0	0	0	0
1	A	1	2	0	2 A	2 A	2 A	2 A	2 A	2 A	2 A	2 A
2	B	2	5	0	2 A	5 B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B
3	C	4	6	0	2 A	5 B	7 A+B	7 A+B	8 A+C	11 B+C	13 A+B+C	13 A+B+C
4	D	5	10	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D
5	E	7	13	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #5: No introducimos el objeto E en el casillero de 7kg con la ganancia asociada igual a 13 porque la ganancia de la etapa #4 para el casillero 7 fue de 15 (mayor a 13, por tanto, lo dejamos en 15).
Evaluamos que otros objetos podemos introducir en los siguientes casilleros mayores a 7kg:

- En la casilla de 8kg podemos colocar los objetos A y E que totalizan una ganancia de 15 (menor a 17 de la etapa anterior). Lo dejamos con 17.

1. Aplicaciones de la Programación Dinámica

El problema de la Mochila

1kg - 2€
2kg - 5€
4kg - 6€
5kg - 10€
7kg - 13€
8kg - 16€

				GANANCIAS EN DIMENSION (Kg.)								
Etapa	**Artículo**	**Dimensión (Kg)**	**Ganancia (€)**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**Estado inicial**			0	0	0	0	0	0	0	0	0
1	A	1	2	0	2 A	2 A	2 A	2 A	2 A	2 A	2 A	2 A
2	B	2	5	0	2 A	5 B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B	7 A+B
3	C	4	6	0	2 A	5 B	7 A+B	7 A+B	8 A+C	11 B+C	13 A+B+C	13 A+B+C
4	D	5	10	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D
5	E	7	13	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D
6	F	8	16	0	2 A	5 B	7 A+B	7 A+B	10 D	12 A+D	15 B+D	17 A+B+D

0 – 8 kg. (máximo)

Estado inicial: Mochila vacía

Etapa #6: No introducimos el objeto F en el casillero de 8kg con la ganancia asociada igual a 16 porque la ganancia de la etapa #5 para el casillero 8 fue de 17 (mayor a 16, por tanto, lo dejamos en 17).

Hemos llegado al **Estado final = Mochila llena = 8kg => Maximizando las ganancias a 17 €**

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

- Este es uno de los famosos problemas de programación dinámica que se pregunta principalmente en las entrevistas técnicas para ingresar a las principales empresas.
- El problema consiste en hacer un cambio del valor dado de centavos donde se tiene un suministro infinito de cada una de las monedas valoradas en C = {c1, c2,….cm}.

PLANTEAMIENTO DEL PROBLEMA

Dado un conjunto de denominaciones de monedas disponibles y un precio objetivo. Encuentre la cantidad mínima de monedas requeridas para pagar lo mismo.

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

PASOS A SEGUIR

1. Podemos comenzar la solución con suma = N centavos.

2. En cada iteración, encontramos las monedas mínimas requeridas dividiendo el problema original en subproblemas.

3. Consideramos una moneda de { 1, c2,…cm} y reducimos la suma repetidamente dependiendo de la moneda de la denominación que se elija.

4. Repetir el mismo proceso hasta que N se convierta en 0, y en este punto, encontramos la solución.

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Ejemplo: Analicemos el caso de Perú que tiene las monedas de céntimos.

Ahora imaginemos que existiera una moneda más de 25 céntimos:

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Ejemplo: Analicemos el caso de Perú que tiene las monedas de céntimos.

Ahora imaginemos que existiera una moneda más de 25 céntimos:

La solución no consiste

- En tomar siempre las monedas de mayor valor hasta llegar al número (como lo hicimos en el enfoque codicioso).

En DP debemos generalizar el problema

- Las denominaciones serán $d_1, d_2, d_3, \ldots, d_k$
- Estas denominaciones estarán ordenadas, es decir:

$$d_1 < d_2 < d_3 < \ldots < d_k$$

- Así, el último ejemplo tiene:

$$d_1 = 1,\ d_2 = 5,\ d_3 = 10,\ d_4 = 20,\ d_5 = 25,\ d_6 = 50$$

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Resolveremos este problema en 4 pasos.

Pasos a seguir

Paso #1: Describir la estructura de una solución optima

- La mejor solución al problema tiene la mejor solución a sus subproblemas.
- Por ejemplo, para 85 céntimos, el óptimo es (10,25,50):

Ambas soluciones son optimas

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Pasos a seguir

Paso #2: Definir recursivamente el valor de una solución

- Voy a almacenar en C[p], la cantidad mínima de monedas para cambiar p centavos.
- ¿Cuál es mi caso base?
  C[0] = 0

- ¿Cómo calculamos otros C[p] para otros p?
  C[p] = min $_{i: d_i \leq p}$ {1+C[p - d_i]}
- Luego, para construir la solución, guardamos en S[p] la moneda escogida.

- **Ejemplo:** Si quiero calcular el mínimo de monedas para n = 85 con las monedas:

• Dijimos que: C[p] = min $_{i: d_i \leq p}$ {1+C[p - d_i]}

C[85] = min{1+d[84], 1+d[80], 1+d[75], 1+d[65], 1+d[60], 1+d[35]}

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Pasos a seguir

Paso #2: Definir recursivamente el valor de una solución

- La GENERALIZACION quedaría así:

$$C[p] \begin{cases} 0, \text{ si } p = 0 \\ \min_{i:d_i \leq p}\{1+C[p-d_i]\} \text{ , si } p \neq 0 \end{cases}$$

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Pasos a seguir

Paso #3: Hallar el valor de una solución optima.

d[ ] = lista de denominaciones de monedas
n = cantidad a sencillar
k = cantidad de denominaciones de monedas

```
Sencillar(d[], n)
   k ← length(d)
   C[0] ← 0                    //caso base
   for p ←1 to n
     min ← infinito
     for i ←1 to k
        if d[i] <=p then              //filtro monedas ≤ p
           if 1 + C[p-d[i]] < min then
              min ← 1 + C[p-d[i]]
              moneda ← i
     C[p] ← min                  //guardo el mínimo y
     Sol[p] ← moneda             //la moneda escogida
   Retornar C y Sol
```

1. Aplicaciones de la Programación Dinámica

El problema del Cambio mínimo de Monedas

Pasos a seguir

Paso #4: Construir la solución optima.

- Ya tenemos un valor óptimo en C[p], ahora debemos construir la solución con S[]

```
Reportar(S, d, n)
   Imprimir "Número mínimo de monedas: " + C[n] + " y son: "
   While n > 0 do
      Imprimir d[Sol[n]] + ", "
      n ← n - d[Sol[n]]
```

Resolvimos el problema del cambio mínimo de monedas con la construcción de una solución optima.

1. Aplicaciones de la Programación Dinámica

CONCLUSIONES

1. La complejidad del tiempo de ejecución del problema de la mochila es **O(N*W)** donde N es el número de artículos dados y W es la capacidad de la mochila.

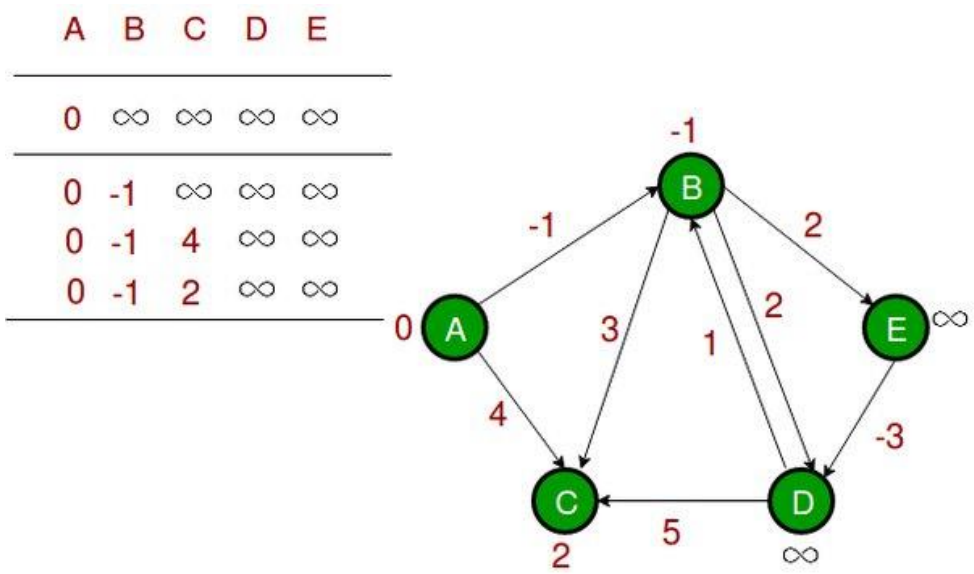
2. La complejidad del tiempo de ejecución del problema mínimo de cambio de moneda es **O(m*n)**, donde m es el número de monedas y n es el cambio requerido.

3. El registro tabular de los resultados simplifican realizar cálculos desde el inicio

4. Hemos aprendido a manejar y desarrollar los principios del paradigma de programación dinámica.

PREGUNTAS

Dudas y opiniones



Complejidad Algorítmica

Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP

Módulo 13: Programación Dinámica en Grafos

Ing. Patricia Reyes Silva
pcsiprey@upc.edu.pe

Complejidad Algorítmica

Semana 13 / Sesión 1

MÓDULO 13: Programación Dinámica en Grafos

Contenido

1. Objetivo de la DP en Grafos
2. Principales Algoritmos de DP en grafos
   - Algoritmo Bellman Ford
   - Algoritmo Floyd-Warshall

Preguntas / Conclusiones

1. Objetivo de la DP en Grafos

¿Qué algoritmos de DP podemos aplicar en grafos?

El objetivo de esta sesión es:

> Manejar y desarrollar los principios del paradigma de **programación dinámica para encontrar caminos mínimos en grafos**.

Los siguientes algoritmos son ejemplos de Programación Dinamica:

- Algoritmo **Bellman-Ford**
- Algoritmo **Floy Warshall**

Aprendamos en que consisten…

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Objetivo: Encontrar el camino más corto desde un vértice a todos los demás vértices de un grafo ponderado.

Características

- Es similar al algoritmo de Dijkstra pero puede funcionar con gráficos en los que **los bordes pueden tener pesos negativos**.

- Los bordes de peso negativo pueden parecer inútiles al principio, pero pueden explicar muchos fenómenos, como:

  - El flujo de caja,
  - El calor liberado (pesos positivos)/absorbido (pesos negativos) en una reacción química, etc.

Hay que tener cuidado con los bordes con pesos negativos…

¿Pesos negativos?
¿Por qué? ¿Para qué?

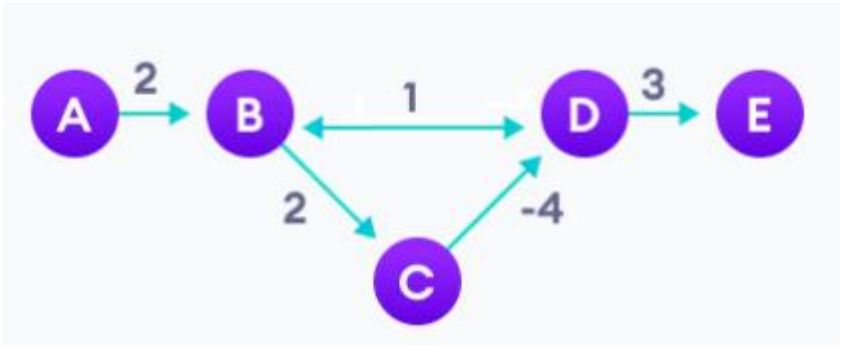
2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

¿Por qué debemos tener cuidado con los pesos negativos?

- Los bordes de peso negativo pueden crear ciclos de peso negativo, es decir, un ciclo que reducirá la distancia total de la ruta al regresar al mismo punto.

- Los ciclos de peso negativos pueden dar un resultado incorrecto al intentar encontrar el camino más corto. Ver el siguiente grafo:



- Los algoritmos de ruta más corta, como el algoritmo de Dijkstra, que no pueden detectar un ciclo de este tipo, pueden dar un resultado incorrecto porque pueden pasar por un ciclo de peso negativo y reducir la longitud de la ruta.

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

- **Bellman Ford** es muy similar al Algoritmo de Dijkstra.
- A diferencia del algoritmo de Dijkstra, el algoritmo de **Bellman-Ford** puede funcionar en grafos con bordes de ponderación negativa.
- Esta capacidad hace que el algoritmo **Bellman-Ford** sea una opción popular.

¿Por qué es importante tener en cuenta los bordes negativos?

En la teoría de grafos, los bordes negativos son más importantes ya que pueden crear un ciclo negativo en un grafo dado.

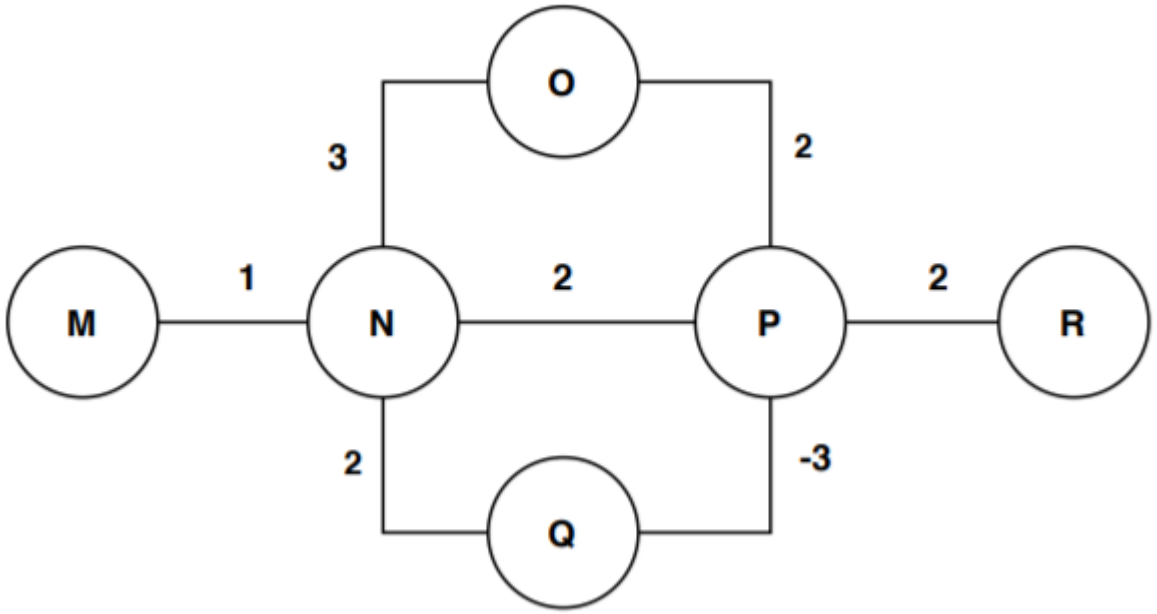
Veamos su funcionamiento en un ejemplo…

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

EJEMPLO

Comencemos con un grafo ponderado simple con un ciclo negativo e intentemos encontrar la distancia más corta de un nodo a otro.

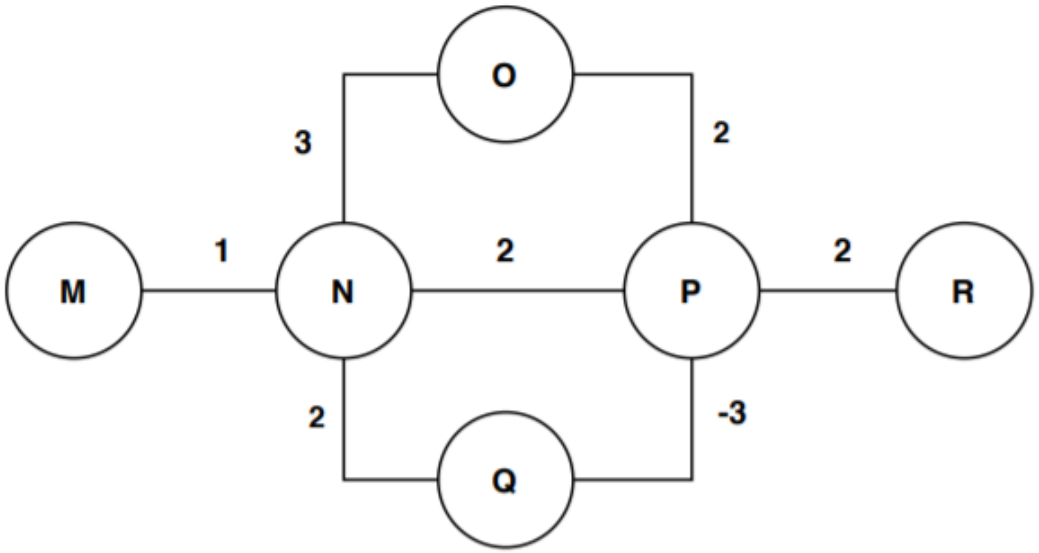


2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

EJEMPLO #1

Comencemos con un grafo ponderado simple con un ciclo negativo e intentemos encontrar la distancia más corta de un nodo a otro.



- Estamos considerando cada nodo como una ciudad. Queremos ir a la **ciudad R** desde la **ciudad M**.
- Hay tres caminos desde la ciudad M para llegar a la ciudad R: **MNPR**, **MNQPR**, **MNOPR**. Las longitudes de los caminos son 5, 2 y 8.
- Vemos que hay un ciclo negativo: **NQP**, que tiene una longitud de **-1**. Entonces, cada vez que cubrimos el camino **NQP**, la longitud de nuestro camino disminuirá.
- Esto hace que no podamos obtener una respuesta exacta sobre el camino más corto, ya que repetir infinitas veces el camino **NQP** sería, por definición, el camino menos costoso.

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Pasos del algoritmo Bellman-Ford

- Este algoritmo toma como entrada un **grafo ponderado dirigido** y **un vértice inicial**.
- Produce todos los caminos más cortos desde el vértice inicial hasta todos los demás vértices.

1. **El primer paso** es inicializar los vértices.
    - El algoritmo inicialmente estableció la distancia desde el vértice inicial hasta todos los demás vértices hasta el infinito.
    - La distancia entre el vértice inicial y sí mismo es 0.
    - La variable D[] denota las distancias en este algoritmo.

2. **El segundo paso**, el algoritmo comienza a calcular la distancia más corta desde el vértice inicial hasta todos los demás vértices. Este paso corre (|V|-1)tiempos.
    - El algoritmo intenta explorar diferentes caminos para llegar a otros vértices y calcula las distancias.
    - Si el algoritmo encuentra una distancia de un vértice que es más pequeña que el valor previamente almacenado, relaja el borde y almacena el nuevo valor.

Algorithm 1: Bellman-Ford Algorithm

```
Data: Given a directed graph G(V, E) , the starting vertex S, and
      the weight W of each edge
Result: Shortest path from S to all other vertices in G
D[S] = 0;
R = V - S;
C = cardinality(V);
for each vertex k ∈ R do
|  D[k] = ∞;
end
for each vertex i = 1 to (C - 1) do
|  for each edge (e1, e2) ∈ E do
|  |  Relax(e1, e2);
|  end
end
for each edge (e1, e2) ∈ E do
|  if D[e2] > D[e1] + W[e1, e2] then
|  |  Print("Graph contains negative weight cycle");
|  end
end
Procedure Relax (e1, e2)
for each edge (e1, e2) in E do
|  if D[e2] > D[e1] + W[e1, e2] then
|  |  D[e2] = D[e1] + W[e1, e2];
|  end
end
```

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Pasos del algoritmo Bellman-Ford (continuación)

3. En el **tercer paso**, después que el algoritmo itera el (|V|-1)tiempo y relaja todos los bordes requeridos, el algoritmo realiza una última verificación para averiguar si hay algún ciclo negativo en el gráfico.

   - Si existe un ciclo negativo, las distancias seguirán disminuyendo.
     - ➢ El algoritmo termina y da como resultado que el grafo contiene un ciclo negativo, por lo que el algoritmo no puede calcular la ruta más corta.

   - Si no se encuentra ningún ciclo negativo
     - ➢ El algoritmo devuelve las distancias más cortas.

¡Apliquemos estos pasos en un ejemplo!

Algorithm 1: Bellman-Ford Algorithm

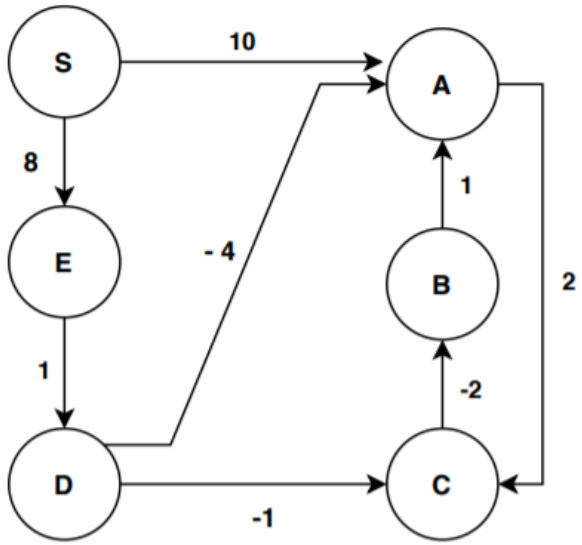
```
Data: Given a directed graph G(V, E) , the starting vertex S, and
      the weight W of each edge
Result: Shortest path from S to all other vertices in G
D[S] = 0;
R = V - S;
C = cardinality(V);
for each vertex k ∈ R do
    D[k] = ∞;
end
for each vertex i = 1 to (C - 1) do
    for each edge (e1, e2) ∈ E do
        Relax(e1, e2);
    end
end
for each edge (e1, e2) ∈ E do
    if D[e2] > D[e1] + W[e1, e2] then
        Print("Graph contains negative weight cycle");
    end
end
Procedure Relax (e1, e2)
for each edge (e1, e2) in E do
    if D[e2] > D[e1] + W[e1, e2] then
        D[e2] = D[e1] + W[e1, e2];
    end
end
```

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo

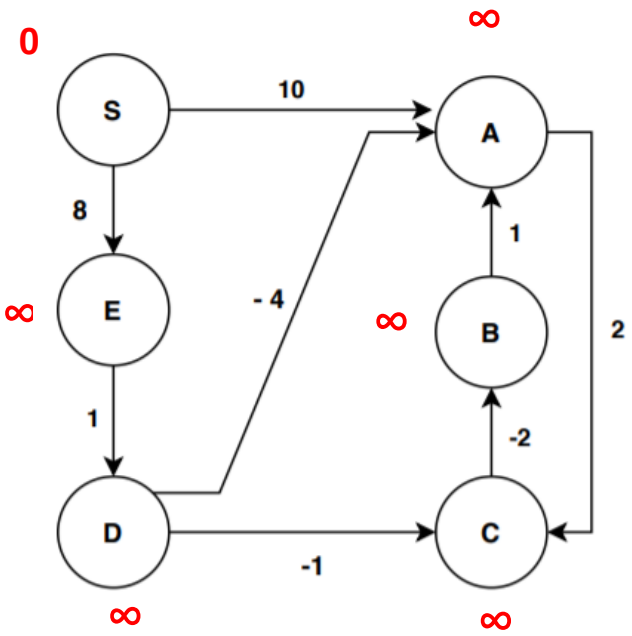
Suponemos que S es nuestro vértice inicial. Ahora estamos listos para comenzar con los pasos del algoritmo:



Paso #1: Inicialización

- Como comentamos, la distancia desde el nodo inicial al nodo inicial es 0.
- La distancia de todos los demás vértices es **infinita** para el paso de inicialización.
- Los valores en rojo denotan las distancias.
- Como tenemos seis vértices, el algoritmo ejecutará cinco iteraciones (|V|-1 veces) para encontrar el camino más corto y una iteración para encontrar un ciclo negativo (si existe alguno).

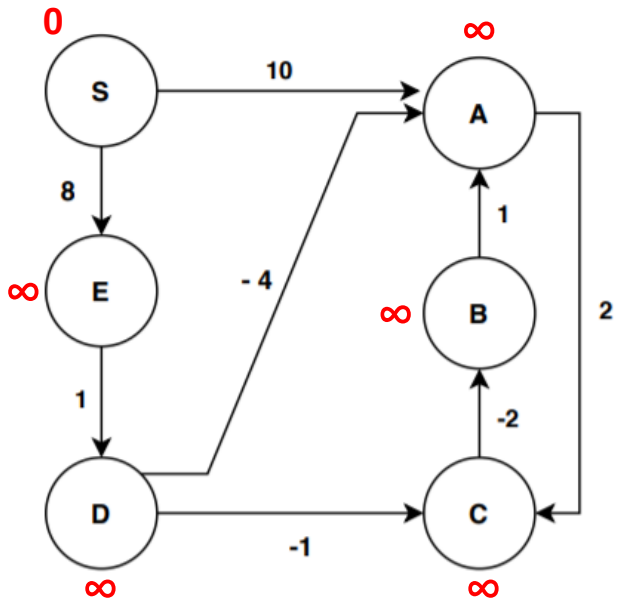
Iteración	S	S -> A	C -> B	A -> C	E -> D	S -> E	B -> A	D->A	D->C
	D(S)	D(A)	D(B)	D(C)	D(D)	D(E)	D(A)	D(A)	D(C)
0	0	∞	∞	∞	∞	∞	∞	∞	∞



2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



Paso #2: Iteración (|V|-1) veces

- Después de inicializar el gráfico, ahora podemos proceder a la primera iteración:
- Actualizamos los valores de distancia de cada vértice:

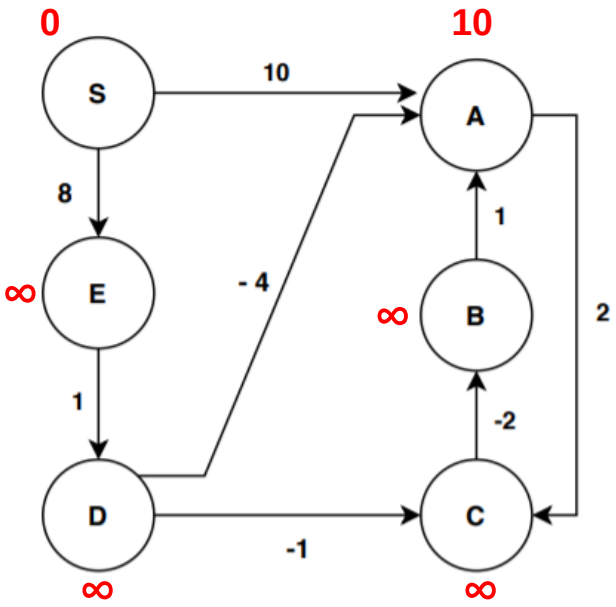
➢ **Iteración #1:** El algoritmo selecciona cada borde y lo pasa a la función Relax(). Primero, para el borde (S, A), veamos cómo funciona la función Relax(S, A). Primero verifica la condición:

$$D[A] > D[S] + W[S, A] \implies \infty > 0 + 10$$
$$\implies \infty > 10 \implies True$$

➢ La arista (S, A) satisface la condición de verificación, por lo tanto, el vértice A obtiene una nueva distancia:

$$D[A] = D[S] + W[S, A] \implies D[A] = 0 + 10 = 10$$

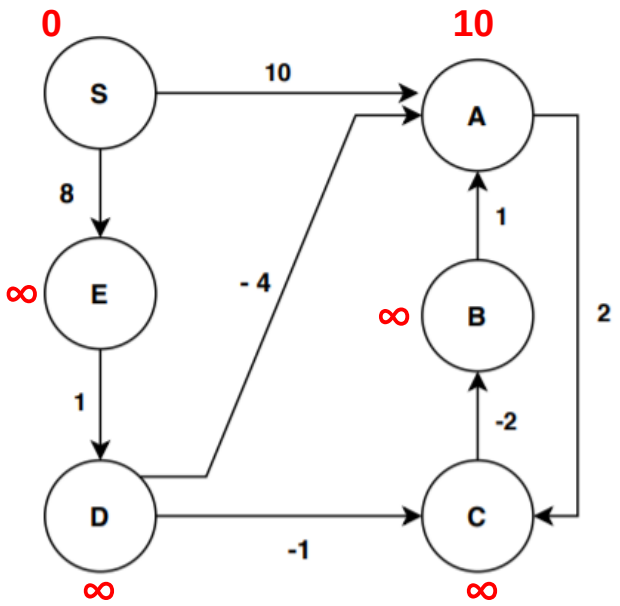
Ahora el nuevo valor de distancia del vértice A es 10



2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



Paso #2: Iteración (|V|-1) veces (continuación)

- Seguimos en la **iteración #1** por los siguientes bordes en este orden: (S, A) -> (A, C) -> (S, E) -> (C, B) -> (B, A) -> (E, D)

(A,C): **D[C]** > D[A] + W [A,C] = 10 + 2 => ∞ > 12 => **si** => **D[C] = 12**

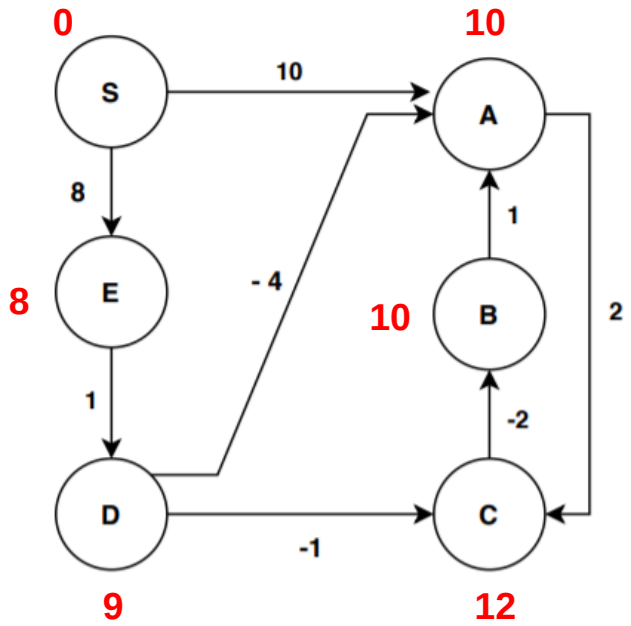
(S,E): **D[E]** > D[S] + W [S,E] = 0 + 8 => ∞ > 8 => **si** => **D[E] = 8**

(C,B): **D[B]** > D[C] + W [C,B] = 12 + (-2) => ∞ > 10 => **si** => **D[B] = 10**

(B,A): **D[A]** > D[B] + W [B,A] = 10 + 1 => **10** > 11 => no => **D[A] =10**

(E,D): **D[D]** > D[E] + W [E,D] = 8 + 1 = 9 => ∞ > 9 => **si** => **D[D] = 9**

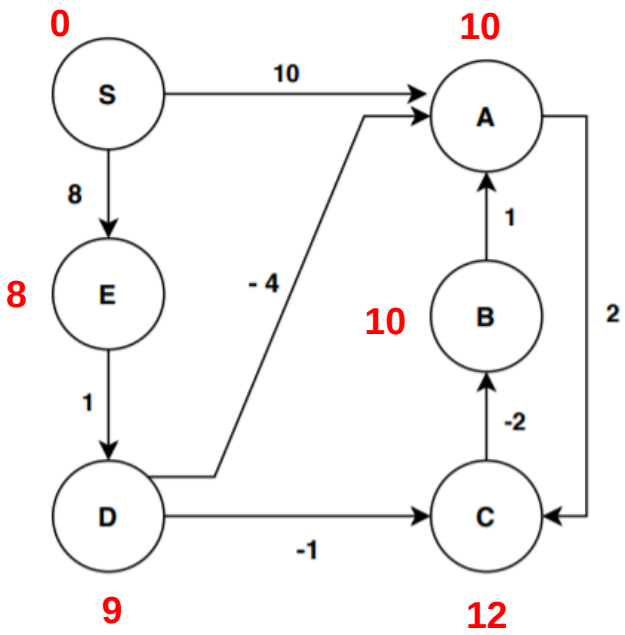
Iteración	S	S -> A	C -> B	A -> C	E -> D	S -> E	B -> A	D->A	D->C
	D(S)	D(A)	D(B)	D(C)	D(D)	D(E)	D(A)	D(A)	D(C)
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	10	10	12	9	8	∞	∞	∞



2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



Paso #2: Iteración (|V|-1) veces (continuación)

- **Iteración #2:** Seguimos iterando por los siguientes bordes en este orden: (S, A) -> (S, E) -> (A, C) -> (B, A) -> (C, B) -> (D, C) -> (D, A)

(S,A) -> **D[A]** > D[S] + W [S,A] = 0 + 10 => **10** > 10 => no => **D[A] = 10**

(S,E) -> **D[E]** > D[S] + W [S,E] = 0 + 8 => **8** > 8 => no => **D[E] = 8**

(A,C) -> **D[C]** > D[A] + W [A,C] = 10 + 2 => **12** > 12 => no => **D[C] = 12**

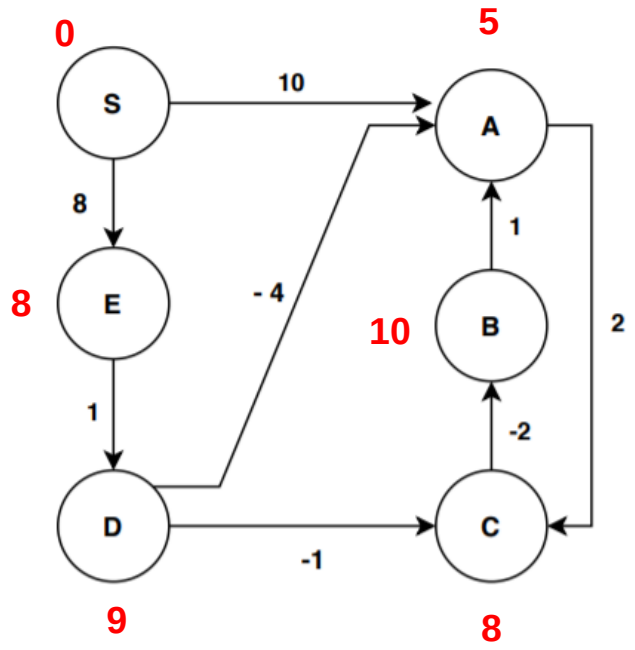
(B,A) -> **D[A]** > D[B] + W [B,A] = 10 + 1 => **10** > 11 => no => **D[A] =10**

(C,B) -> **D[B]** > D[C] + W [C,B] = 12 + (-2) => **10** > 10 => no => **D[B] = 10**

Si cumplen la condición:

(DC) -> **D[C]** > D[D] + W [D,C] = 9 + (-1) = 8 => **9** > 8 => SI => **D[C] = 8**

(DA) -> **D[A]** > D[D] + W [D,A] = 9 + (-4) = 5 => **10** > 5 => SI => **D[A] = 5**

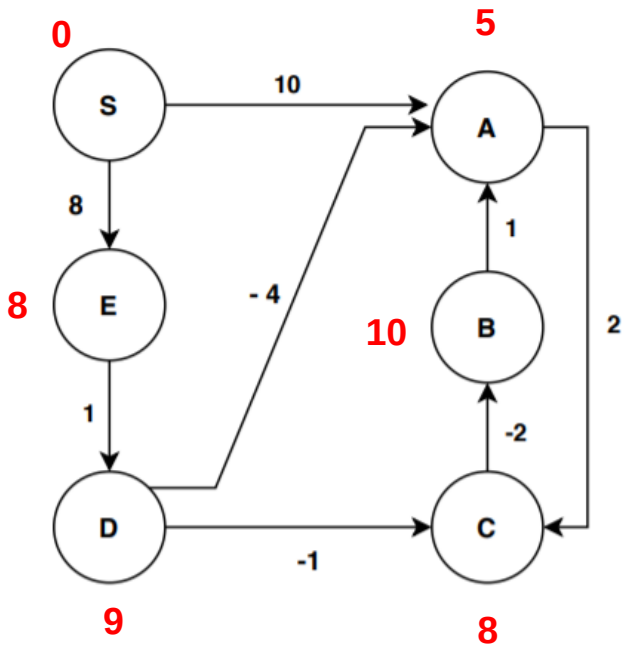


Iteración	S	S -> A	C -> B	A -> C	E -> D	S -> E	B -> A	D->A	D->C
	D(S)	D(A)	D(B)	D(C)	D(D)	D(E)	D(A)	D(A)	D(C)
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	10	10	12	9	8	∞	∞	∞
2	0	5	10	8	9	8	10	5	8

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



Paso #2: Iteración (|V|-1) veces (continuación)

- **Iteración #3:** Seguimos iterando por los siguientes bordes en este orden: (S, A) -> (S, E) -> (A, C) -> (B, A) -> (C, B) -> (D, C) -> (D, A)

(S,A) -> **D[A]** > D[S] + W [S,A] = 0 + 10 => **5** > 10 => no => **D[A] = 5**

(S,E) -> **D[E]** > D[S] + W [S,E] = 0 + 8 => **8** > 8 => no => **D[E] = 8**

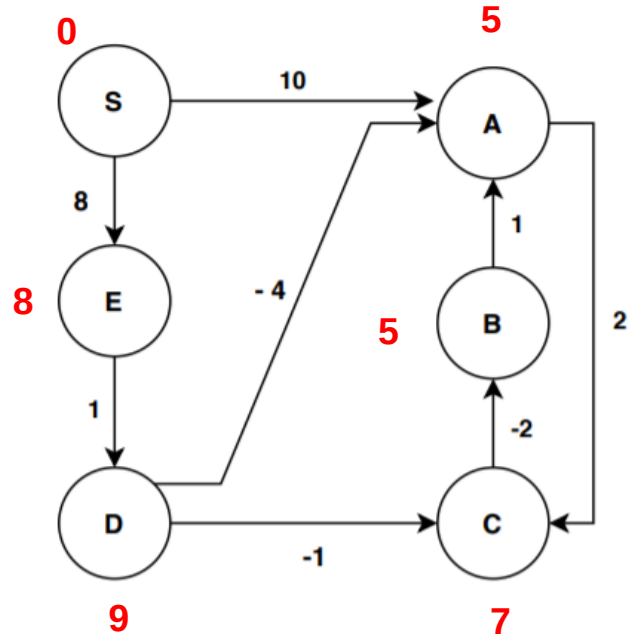
(A,C) -> **D[C]** > D[A] + W [A,C] = 5 + 2 => **8** > 7 => **si** => **D[C] = 7**

(B,A) -> **D[A]** > D[B] + W [B,A] = 10 + 1 => **5** > 11 => no => **D[A] =5**

(C,B) -> **D[B]** > D[C] + W [C,B] = 7 + (-2) => **10** > 5 => **si** => **D[B] = 5**

(DC) -> **D[C]** > D[D] + W [D,C] = 9 + (-1) = 8 => **7** > 8 => no => **D[C] = 7**

(DA) -> **D[A]** > D[D] + W [D,A] = 9 + (-4) = 5 => **5** > 5 => no => **D[A] = 5**

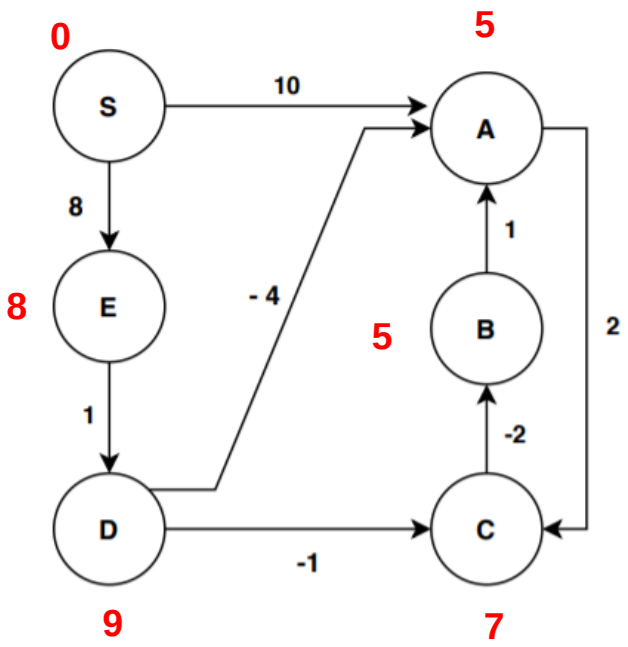


Iteración	S	S -> A	C -> B	A -> C	E -> D	S -> E	B -> A	D->A	D->C
	D(S)	D(A)	D(B)	D(C)	D(D)	D(E)	D(A)	D(A)	D(C)
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	10	10	12	9	8	∞	∞	∞
2	0	5	10	8	9	8	10	5	8
3	0	5	5	7	9	8	5	5	7

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



Paso #2: Iteración (|V|-1) veces (continuación)

- **Iteración #4:** Seguimos iterando por los siguientes bordes en este orden: (S, A) -> (S, E) -> (A, C) -> (B, A) -> (C, B) -> (D, C) -> (D, A)

(S,A) -> **D[A]** > D[S] + W [S,A] = 0 + 10 => **5** > 10 => no => **D[A] = 5**

(S,E) -> **D[E]** > D[S] + W [S,E] = 0 + 8 => **8** > 8 => no => **D[E] = 8**

(A,C) -> **D[C]** > D[A] + W [A,C] = 5 + 2 => **7** > 7 => no => **D[C] = 7**

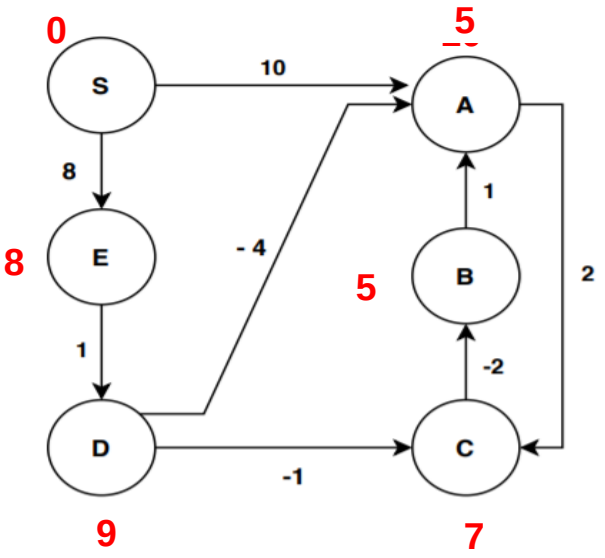
(B,A) -> **D[A]** > D[B] + W [B,A] = 5 + 1 => **5** > 6 => no => **D[A] =5**

(C,B) -> **D[B]** > D[C] + W [C,B] = 7 + (-2) => **5** > 5 => no => **D[B] = 5**

(DC) -> **D[C]** > D[D] + W [D,C] = 9 + (-1) = 8 => **7** > 8 => no => **D[C] = 7**

(DA) -> **D[A]** > D[D] + W [D,A] = 9 + (-4) = 5 => **5** > 5 => no => **D[A] = 5**

Ninguna actualización en la iteración #4

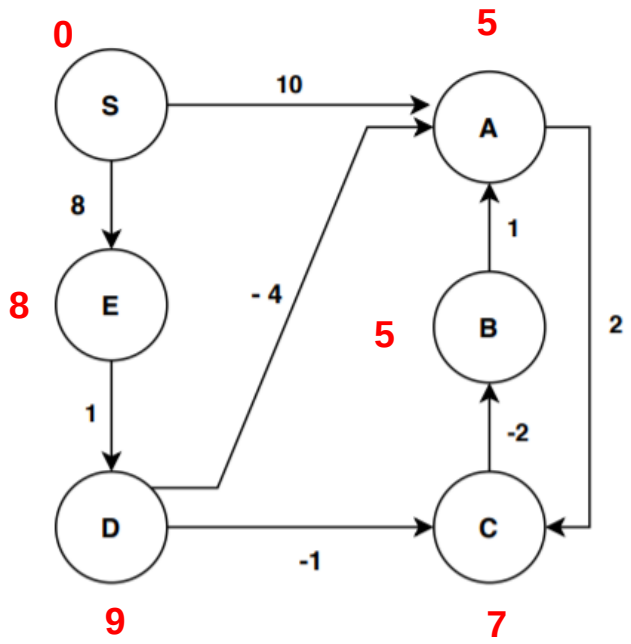


Iteración	S	S -> A	C -> B	A -> C	E -> D	S -> E	B -> A	D->A	D->C
	D(S)	D(A)	D(B)	D(C)	D(D)	D(E)	D(A)	D(A)	D(C)
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	10	10	12	9	8	∞	∞	∞
2	0	5	10	8	9	8	10	5	8
3	0	5	5	7	9	8	5	5	7
4	0	5	5	7	9	8	5	5	7

2. Principales Algoritmos de DP en grafos

ALGORITMO BELLMAN FORD

Ejemplo #1: Un grafo sin ciclo negativo



- No hay actualizaciones en los valores de distancia de los vértices después de la cuarta iteración.
- Esto significa que el algoritmo ha producido el resultado final (a pesar que mencionamos que necesitamos ejecutar este algoritmo |V|-1 veces, es decir para 5 interacciones).
- En este caso, obtuvimos nuestro resultado después de 4 iteraciones.
- En general (|V| - 1) es el mayor número de iteraciones que necesitamos ejecutar en caso de que los valores de distancia de las iteraciones consecutivas no sean estables.
- En este caso, obtuvimos los mismos valores para dos iteraciones consecutivas, por lo que el algoritmo termina.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Objetivo: Encontrar el camino más corto para cada par de vértices en un grafo dirigido ponderado.

Características

- En el problema del camino más corto de todos los pares, necesitamos encontrar todos los caminos más cortos desde cada vértice a todos los demás vértices en el grafo.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Pasos del algoritmo FLOYD-WARSHALL

- Este algoritmo toma como entrada un **grafo ponderado dirigido G (V, E)**.
- Produce todos los caminos más cortos por cada par de vértices en G.

1. **El primer paso:**
   - El algoritmo construye una matriz gráfica a partir del grafo inicial dado.
   - Esta matriz incluye los pesos de los bordes en el gráfico.
   - El resto de las posiciones se rellenan con los respectivos pesos de los bordes del gráfico de entrada.
2. **El segundo paso**, encontrar la distancia entre dos vértices
   - Al encontrar la distancia, también verificamos si hay algún vértice intermedio entre dos vértices seleccionados.
   - Si existe un vértice intermedio, comprobamos la distancia entre el par de vértices seleccionados que pasa por este vértice intermedio.
   - Si esta distancia al atravesar el vértice intermedio es menor que la distancia entre dos vértices seleccionados sin pasar por el vértice intermedio, actualizamos el valor de la distancia más corta en la matriz.

- El número de iteraciones es igual a la cardinalidad del conjunto de vértices.
- El algoritmo **devuelve la distancia más corta de cada vértice a otro en el grafo dado**.

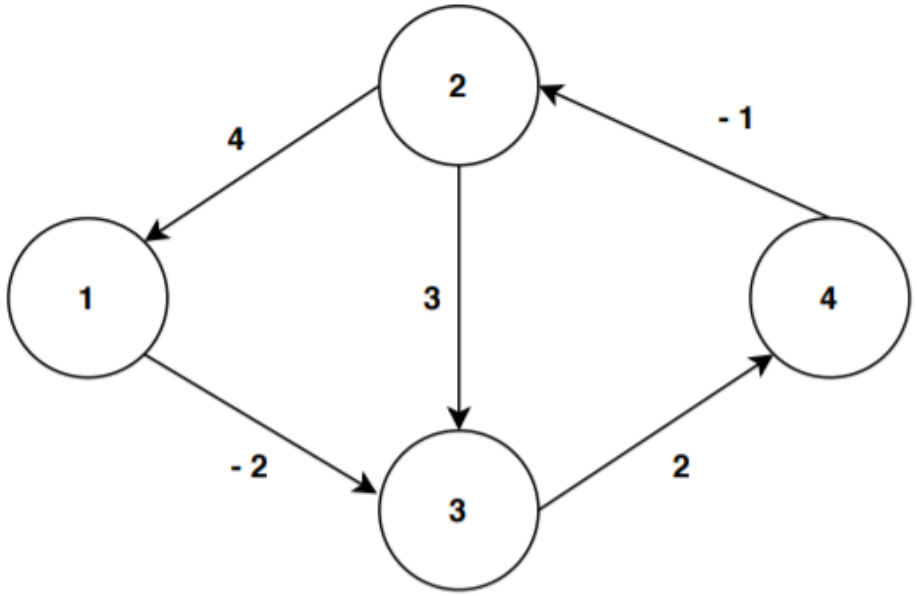
Algorithm 1: Pseudocode of Floyd-Warshall Algorithm

```
Data: A directed weighted graph G(V, E)
Result: Shortest path between each pair of vertices in G
for each d ∈ V do
    distance[d][d] ← 0;
end
for each edge (s, p) ∈ E do
    distance[s][p] ← weight(s, p);
end
n = cardinality(V);
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if distance[i][j] > distance[i][k] + distance[k][j] then
                distance[i][j] ← distance[i][k] + distance[k][j];
            end
        end
    end
end
```

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #1:

- Construimos una matriz a partir del grafo de entrada G.

$$\begin{pmatrix} \infty & \infty & -2 & \infty \\ 4 & \infty & 3 & \infty \\ \infty & \infty & \infty & 2 \\ \infty & -1 & \infty & \infty \end{pmatrix}$$

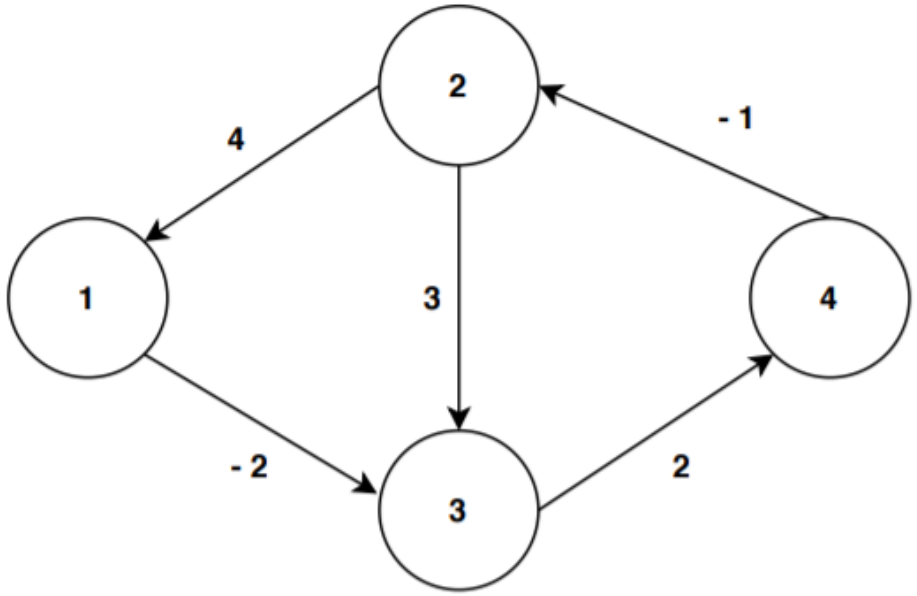
- Insertamos **"0"** en las posiciones diagonales en la matriz, y el resto de las posiciones se mantienen con los pesos de los bordes del grafo de entrada.

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #2:

- Como la cardinalidad del conjunto de vértices es 4, tendremos 4 iteraciones.

Iteración #1: $k = 1, i = 1, j = 1$, verificaremos si debemos actualizar la matriz

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \propto & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

$$distance[i][j] > distance[i][k] + distance[k][j]$$

$$distance[1][1] > distance[1][1] + distance[1][1]$$

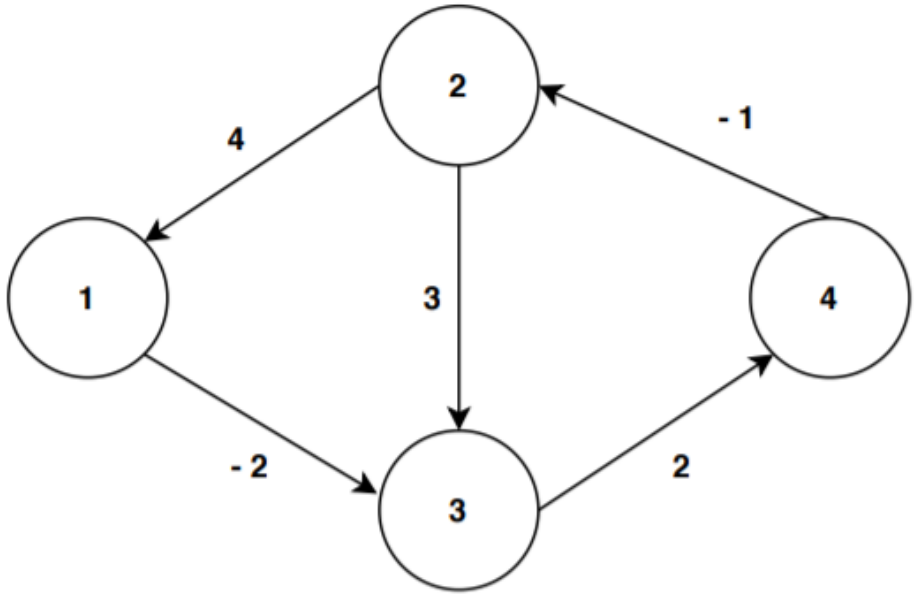
$$0 > 0 + 0 \implies \text{FALSE}$$

- Como ningún valor de la iteración cumple con la condición, no hay actualización en la matriz.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #2:

Iteración #1: *k = 1, i = 1, j = 2 verificamos nuevamente en la matriz*

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

$$distance[i][j] > distance[i][k] + distance[k][j]$$

$$distance[1][2] > distance[1][1] + distance[1][2]$$

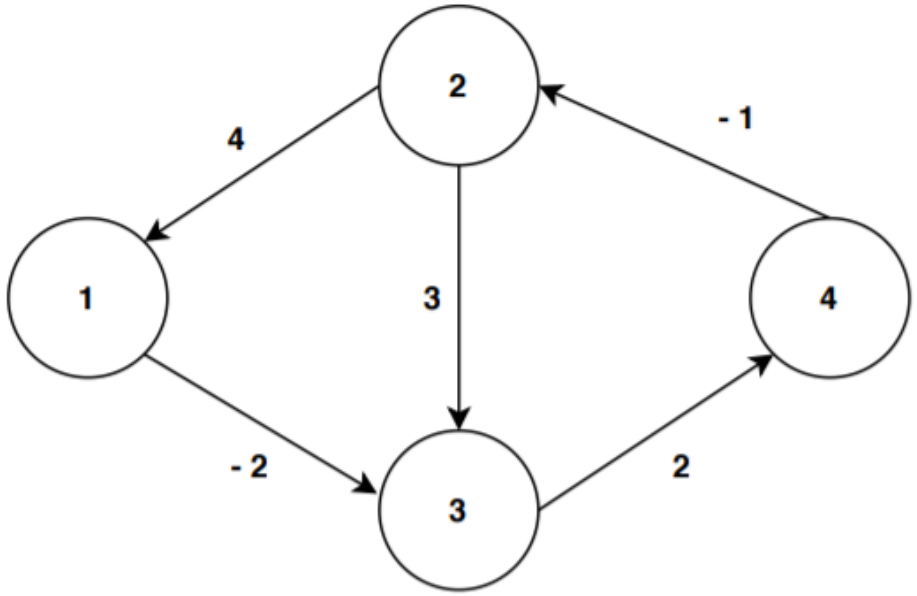
$$0 > 0 + \infty \implies \text{FALSE}$$

- Por lo tanto, no habrá cambios en la matriz. De esta forma, continuaremos y comprobaremos todos los pares de vértices.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #2:
Avancemos rápidamente a algunos valores que satisfagan la condición de distancia.

Iteración #1: Veremos que se cumple la condición para los valores del ciclo k =1, i=2, j= 3

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

$$distance[i][j] > distance[i][k] + distance[k][j]$$

$$distance[2][3] > distance[2][1] + distance[1][3]$$

$$3 > 4 + -2$$

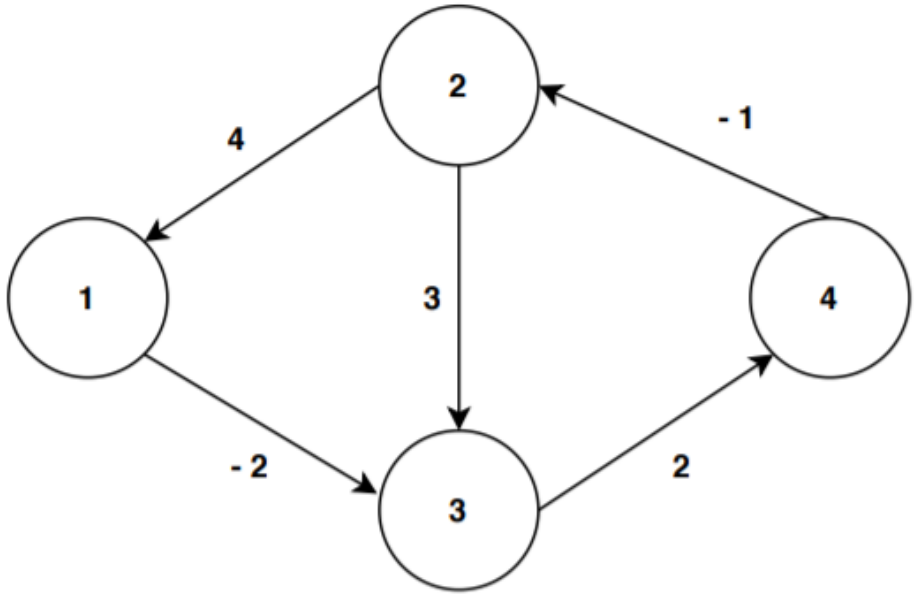
$$3 > 2 \implies \text{TRUE}$$

- Por lo tanto, la condición se cumple para el par de **vértices (2,3)**. Al principio, la distancia entre el vértice 2 a 3 **era 3**. Sin embargo, encontramos **una nueva distancia más corta 2**. Por eso, actualizamos la matriz con esta nueva distancia de ruta más corta.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #2:
Avancemos rápidamente a algunos valores que satisfagan la condición de distancia.

Iteración #1: Veremos que se cumple la condición para los valores del ciclo k=2, i= 4, j= 1

$$\begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ 3 & -1 & \infty & 0 \end{pmatrix}$$

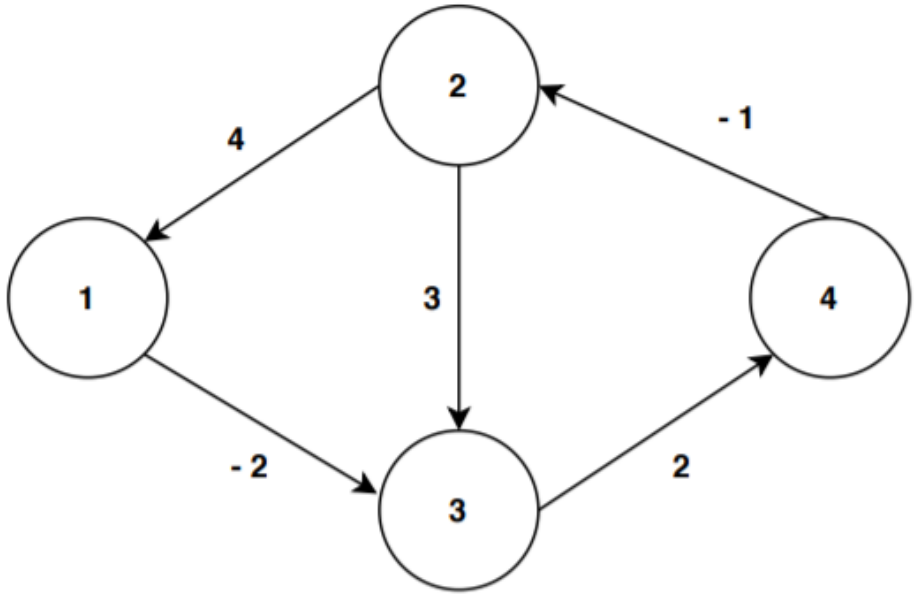
$$distance[i][j] > distance[i][k] + distance[k][j]$$
$$distance[4][1] > distance[4][2] + distance[2][1]$$
$$\infty > -1 + 4$$
$$\infty > 3 \implies \text{TRUE}$$

- Por lo tanto, la condición se cumple para el par de **vértices (4,1)**. Al principio, la distancia entre el vértice 4 a 1 era ∞. Sin embargo, encontramos **una nueva distancia más corta 3**. Por eso, actualizamos la matriz con esta nueva distancia de ruta más corta.

2. Principales Algoritmos de DP en grafos

ALGORITMO FLOYD-WARSHALL

Ejemplo: Ejecutemos el algoritmo de **Floyd-Warshall** en un grafo dirigido ponderado:



Paso #2:
Avancemos rápidamente a algunos valores que satisfagan la condición de distancia.

Iteración #n: continuamos y verificamos diferentes valores de bucle. Finalmente, después de que termine el algoritmo, obtendremos la matriz de salida que contiene las distancias más cortas de todos los pares:

$$\begin{pmatrix} 0 & -1 & -2 & 0 \\ 4 & 0 & 2 & 4 \\ 5 & 1 & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$

- NO OLVIDAR que el número de iteraciones es igual a la cardinalidad del conjunto de vértices (en este ejemplo es 4).

Programación Dinámica en Grafos

CONCLUSIONES

1. El algoritmo **Bellman-Ford** es un ejemplo de programación dinámica.
2. Comienza con un vértice inicial y calcula las distancias de otros vértices que se pueden alcanzar por un borde. Luego continúa encontrando un camino con dos aristas y así sucesivamente. El algoritmo **Bellman-Ford** sigue el enfoque de abajo hacia arriba.
3. Complejidad de **Bellman-Ford**
   - Primero, el paso de inicialización toma O(V).
   - Luego, el algoritmo itera (|V| - 1)veces y cada iteración toma O(1)tiempo.
   - Después (|V| - 1)de las interacciones, el algoritmo elige todos los bordes y luego pasa los bordes a Relax() . Elegir todos los bordes lleva O(E)tiempo y la función Relax() lleva O(1)tiempo.
   - Por lo tanto , la complejidad de hacer todas las operaciones lleva O(VE)tiempo.
   - Dentro de la función Relax() , el algoritmo toma un par de aristas, realiza un paso de verificación y asigna el nuevo peso si está satisfecho. Todas estas operaciones toman O(E)tiempo.
   - Por lo tanto, el tiempo total del algoritmo Bellman-Ford es la suma del tiempo de inicialización, el tiempo de ciclo y el tiempo de la función Relax . En total, la complejidad temporal del algoritmo Bellman-Ford es O(VE).
4. El algoritmo **Floyd-Warshall** también es un ejemplo de programación dinámica.
   - Primero, insertamos los pesos de los bordes en la matriz. Hacemos esto usando un bucle **for** que visita todos los vértices del grafo. Esto se puede realizar en un tiempo O(n).
   - Luego, realizamos tres bucles anidados, cada uno de los cuales va desde uno hasta el número total de vértices del grafo. Por lo tanto, la complejidad temporal total de este algoritmo es O (n^3)

PREGUNTAS

Dudas y opiniones