

L Resumen TD V⁷

Un grafo es un par $G(V, E)$ tal que

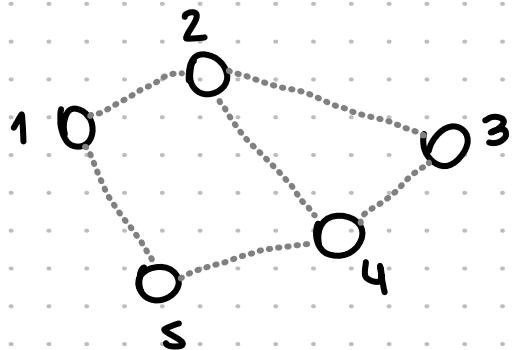
Vértices

aristas

\checkmark V es el conjunto (finito) de vértices

E es el conjunto de aristas

Las aristas van de un vértice a otro (sin bucles)



$$V = \{1, 2, 3, 4, 5\}$$

no van las repeticiones

$$E = \{(1,2); (1,3); (2,1); (2,3); (2,4); (3,2); (3,4); (4,2); (4,3); (4,5); (5,1); (5,4)\}$$

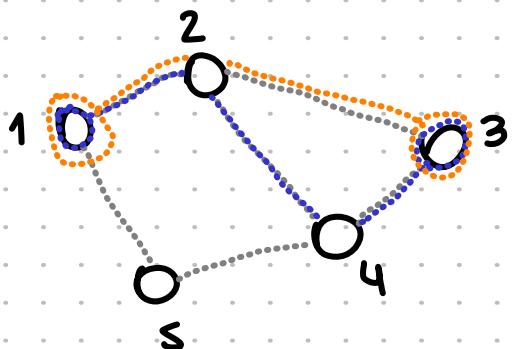
(Nodos = vértices)

Dos nodos conectados por una arista son **vecinos**.

El **vecindario** de un nodo es el conjunto de sus vecinos.

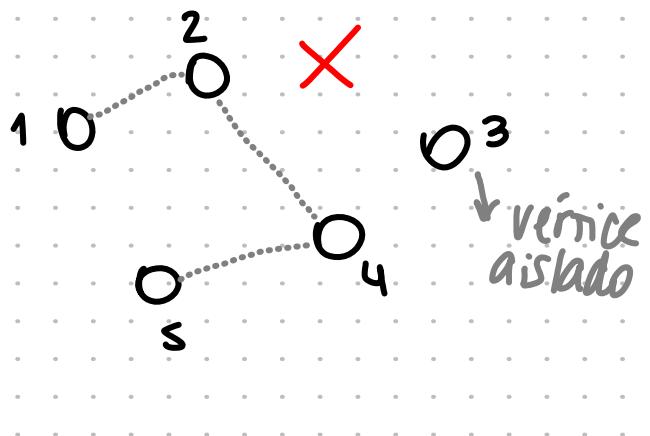
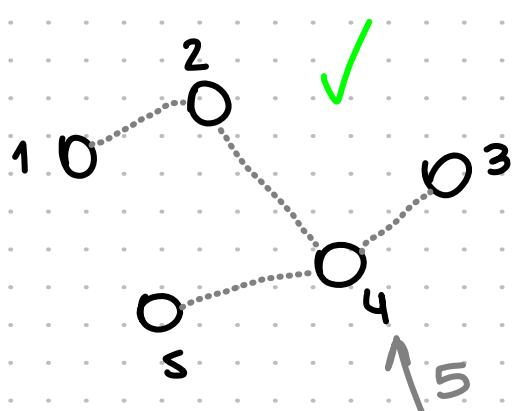
El **grado** de un nodo es cuantas aristas tiene (o vecinos).

Un **camino** entre dos nodos es la secuencia de aristas que los unen.



La **distancia** es la cantidad de aristas en el camino más corto.

Un grafo es **conexo** si existe un camino entre todo par de vértices.

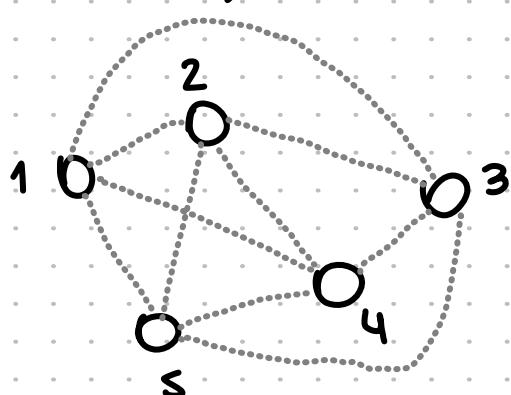


La **componente conexa** es el subconjunto más grande conexo de un grafo.

Un **vértice aislado** es aquél que no tiene vecinos.

La distancia de cualquier nodo a uno aislado es infinito.

En un grafo **completo**, todos los nodos son adyacentes (unidos).



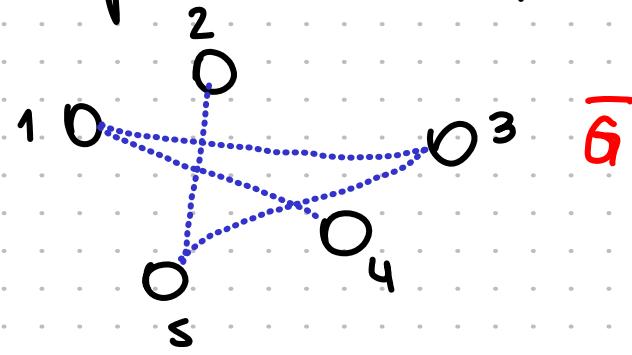
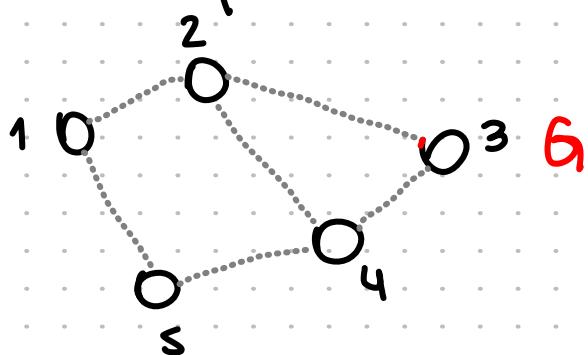
aristas de un grafo completo:

$$M = \binom{n}{2} = \frac{n(n-1)}{2}$$

n: vértices

M: aristas

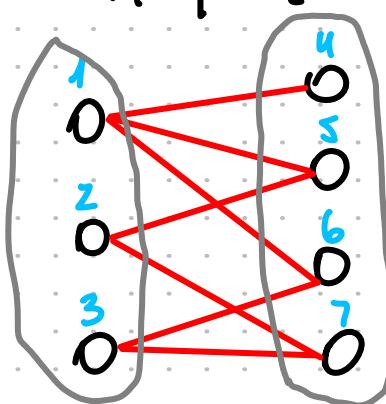
Un **grafo complemento** \bar{G} de G contiene las aristas que le faltan a G para ser completo.



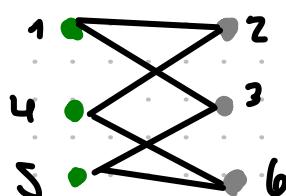
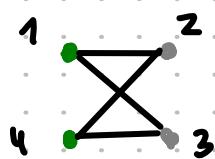
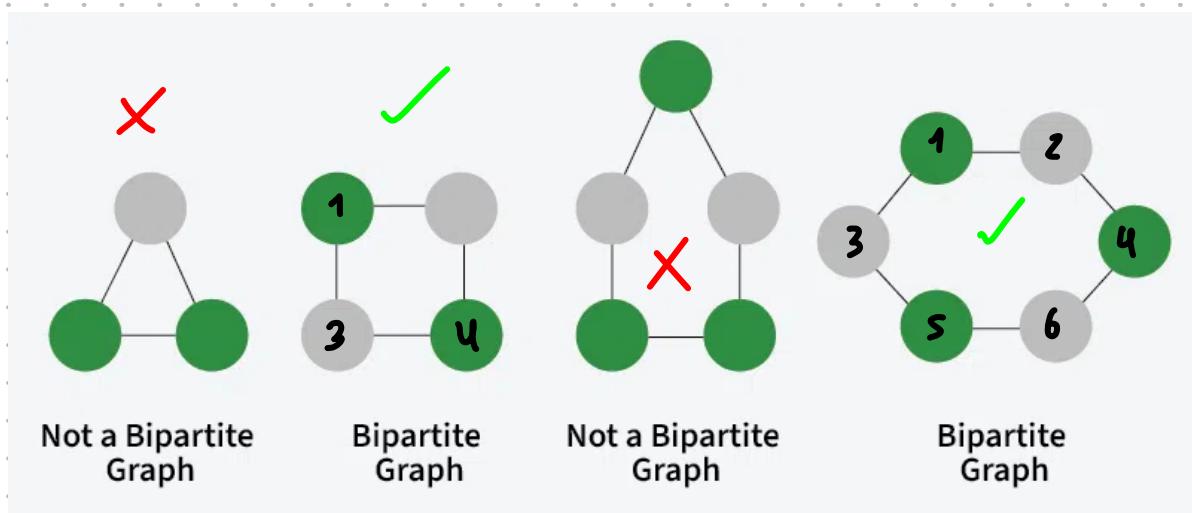
Va a tener $\bar{m} = \binom{n}{2} - m = \frac{n(n-1)}{2} - m$ aristas.

Un **grafo bipartito** cumple las siguientes propiedades:

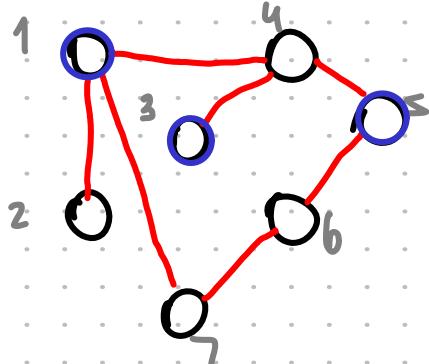
- * Existe una bipartición del conjunto de vértices V .
- * $V_1 \cup V_2 = V$
- * $V_1 \cap V_2 = \emptyset$
- * $V_1 \neq \emptyset \wedge V_2 \neq \emptyset$
- * todas las aristas del grafo tienen un extremo en V_1 y V_2 .



? No existen aristas que salgan y vuelven al mismo conjunto



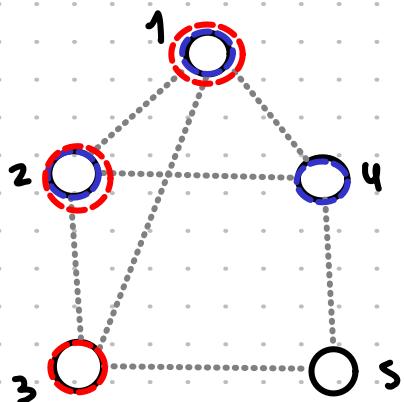
Un **conjunto independiente** es el conjunto de vértices que no están unidos por una arista.



$$\text{! } \mathbb{I} \subseteq V \\ \mathbb{I} = \{1, 3, 5\}$$

$$\alpha(G) = \mathbb{I} \text{ Máximo}$$

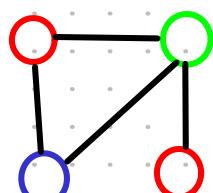
Una **clique** es un subconjunto de vértices unidos por aristas. El subconjunto es $K \subseteq V$.
Todo vector $\in K$ es adyacente a los demás.



$$K_1 = \{1, 2, 4\} \\ K_2 = \{1, 2, 3\}$$

$$\omega(G) = K \text{ Máximo}$$

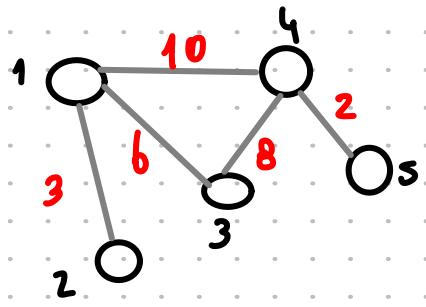
El **coloreo** es un método que implica que dos vértices no comparten el mismo color si son adyacentes.



Idealmente, el coloreo debería ser mínimo
 $\chi(G)$ es la función del número cromático.

$\chi(G) \leq \Delta(G) + 1 \rightarrow$ cota superior.
→ grado máximo del gráfico

las aristas pueden tener **pesos** asignados



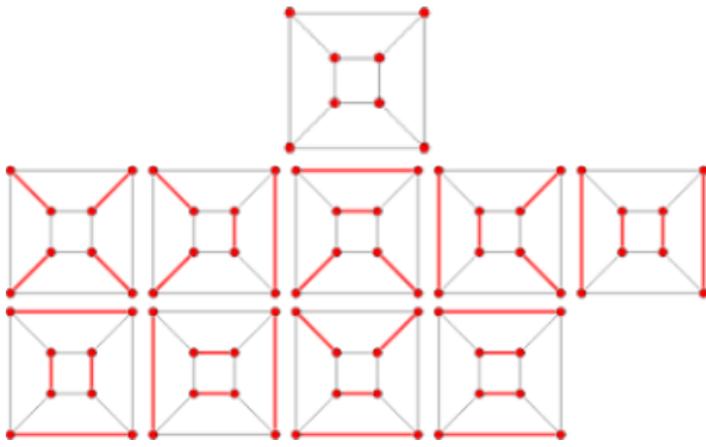
los vértices también (no está graficado)

$$V = \{1, 2, 3, u, s\}$$

$$E = \{12, 13, 14, 34, 4s\}$$

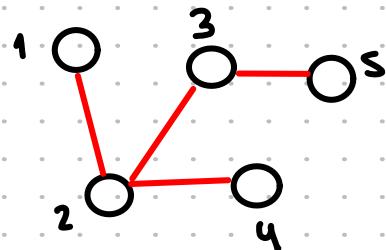
$$W_{12} = 3; W_{13} = 6; W_{14} = 10; W_{34} = 8; W_{4s} = 2; \dots$$

El **matching** es un subconjunto de aristas que incide sobre todos los vértices de un grafo. Ningún par de aristas comparte un vértice.



Falta lo de isomorfismo.

los grafos se pueden representar con una **Matriz de adyacencia**.



=>

$$A = \begin{vmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 1 & 0 & 0 \end{vmatrix}$$

A es una matriz
espejada respecto de
la diagonal.

Diagonal
con ceros
sí o sí.

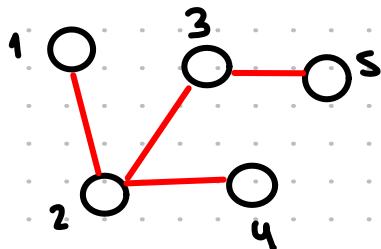
LO BUENO

Operaciones: Agregar/eliminar arista: $O(1)$
¿Existe la arista?: $O(1)$

LO MALO

Agregar/eliminar vértice: $O(n^2)$
Obtener vecinos de un vértice: $O(n)$

Otra representación es la **lista de vecinos**



$$\Rightarrow \begin{array}{l} 1 \rightarrow 2 \\ 2 \rightarrow 1, 3, 4 \\ 3 \rightarrow 2, 5 \\ 4 \rightarrow 2 \\ 5 \rightarrow 3 \end{array}$$

LO BUENO:

Obtener vecinos: $O(1)$

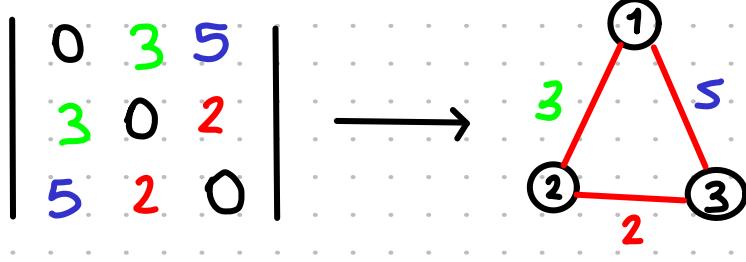
Agregar vértice: $O(n)$ peor caso
 $O(1)$ amortizado

LO MALO:

Agregar/quitar lados.
arista: $O(\log n)$
elim. vértice: $O(n^2)$

Operación	Matriz de adyacencia	Conjunto de Vecinos
Complejidad espacial	$O(n^2)$	$O(E)$
Agregar vértice	$O(n^2)$	$O(n)/O(1)$ amortizado
Eliminar vértice	$O(n^2)$	$O(n^2)$
Agregar arista	$O(1)$	$O(\log n)$
Eliminar arista	$O(1)$	$O(\log n)$
Consultar arista	$O(1)$	$O(\log n)$
Obtener vecinos	$O(n)$	$O(1)$

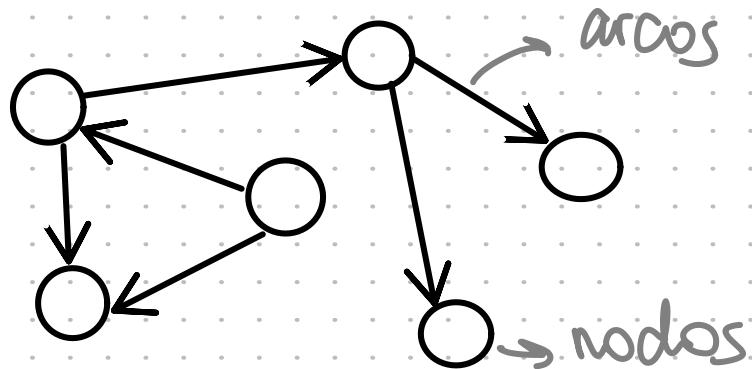
Si los grafos fuesen pesados, una matriz de adyacencia se vería así:



y lista de vecinos:

$$\begin{aligned} 1 &\rightarrow (2, 3); (3, 5) \\ 2 &\rightarrow (1, 3); (3, 2) \\ 3 &\rightarrow (1, 5); (2, 2) \end{aligned}$$

En un **grafo dirigido**, las flechas tienen sentidos



los arcos son pares ordenados ahora:

(Origen, destino)

Un problema de **optimización** tiene la forma:

CLASE 05/06

$$z^* = \max_{x \in S} f(x) \quad \text{ó} \quad z^* = \min_{x \in S} f(x)$$

↓
valor óptimo

f(x) → función objetivo
x ∈ S → región factible

solución factible

z* → óptimo del problema

z* es el resultado de f(x*)

Un algoritmo de **fuerza bruta** genera todas las soluciones factibles y se queda con el óptimo del problema x^* .

Si existe una solución, este algoritmo la encuentra.
Lo malo: la complejidad.

Objetos: $N = \{1, \dots, n\}$

Peso del objeto i: p_i

Beneficio del objeto i: b_i

Capacidad máx de la mochila: C

} Una instancia cualquiera del problema tiene estas cuatro variables.

Queremos averiguar qué objetos poner en la mochila, sin excedernos del peso y con el mayor beneficio posible.

Instancia

$$\sum = 280$$

- $n = 8$
- $b = (b_i) = (15, 100, 90, 60, 40, 15, 10, 1)$
- $p = (p_i) = (2, 20, 20, 30, 40, 30, 60, 10)$
- $C = 102$

$$\sum = 102 (\text{todo})$$

Solución factible

- $S = \{1, 2, 3, 4, 6\}$
- $z^* = 280$
- Capacidad usada: 102

Un objeto i con $b_i < 0$ ó $p_i < 0$ no es válido para la instancia.

con fuerza bruta, vamos a probar todas las combinaciones posibles de objetos para meter.

Mochila ($S \in [0,1]^K$, $K \in \mathbb{N}$)

S es un vector con valores binarios

$S[K] = 0 \rightarrow$ no ponemos el objeto K

$S[K] = 1 \rightarrow$ ponemos el objeto K

K es el índice del objeto. Hay n objetos.

} IF $K=n$: Si ya consideramos todos los objetos

IF peso(s) $\leq C$ \wedge beneficio(s) > beneficio(B):

NO nos excedemos y encontramos una combinación mejor

$B=s$ Redefinimos la mejor solución

else: caso recursivo

Rama ponemos el objeto K

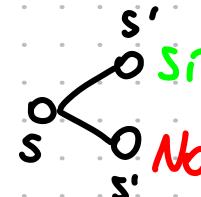
$s' = s + \{1\}$ se agrega

Mochila (s' , $K+1$) llamado recursivo con un objeto más visto

Rama NO ponemos el objeto K

$s' = s + \{0\}$

Mochila (s' , $K+1$)



}

Se hace el primer llamado con Mochila([], 1)

y $B = []$.

Se puede hacer la **poda** de ramas que cumplen ciertas propiedades.

Factibilidad: si una solución parcial excede la capacidad de la mochila.

Optimalidad: si una solución parcial no mejora a nuestro mejor candidato.

FACTIBILIDAD

Mochila($s \in \{0,1\}^k, k \in \mathbb{N}$):

IF $k=n$:

IF $\text{peso}(s) \leq C \wedge \text{ben}(s) > \text{ben}(B)$:

$B = s$

else IF $\text{peso}(s) \leq C$:

$s' = s + \{1\}$

Mochila($s', k+1$) } Agregar

$s' = s + \{0\}$ } No

Mochila($s', k+1$) } No Agregar

Condición AZUL: **backtracking**.

El **backtracking** es una generalización nada más:

backtracking($S \in D_1 \times \dots \times D_k$)

→ vector de decisiones D

IF $k=n$:

IF **esfactible**(s):

print(s)

encontramos = true

else:

for s in S → para cada continuación posible

$s' = s + \{s\}$ → se agrega la continuación

backtracking(s') → llamado recursivo

IF encontramos: return

}

}

En un problema de clique Máxima la secuencia de decisiones puede ser si incluimos los vértices.

OPTIMALIDAD

Mochila($s \in \{0,1\}^k, k \in \mathbb{N}$):

IF $k=n$:

IF $\text{peso}(s) \leq C \wedge \text{ben}(s) > \text{ben}(B)$:

$B = s$

else IF $\text{peso}(s) \leq C \wedge$

$\text{ben}(s) + \sum_{i=k+1}^n b_i > \text{ben}(B)$

$s' = s + \{1\}$

Mochila($s', k+1$) } Agregar

$s' = s + \{0\}$ } No

Mochila($s', k+1$) } No Agregar

MaxClique ($S \in \{0,1\}^K$) sin podas

} IF $K=n$: recorrimos todos los nodos.
IF $\text{esClique}(S) \wedge |S| > |B|$: si es clique y es mayor en tamaño
 $B = S$
else: crear ramas por cada posibilidad. Podemos NO conseguir una clique.
 $S' = S + \{0\}$ | no se agrega el nodo
 $\text{MaxClique}(S')$
 $S' = S + \{1\}$ | se agrega el nodo
 $\text{MaxClique}(S')$

{

MaxClique ($S \in \{0,1\}^K$) con backtracking

} IF $K=n$:
IF $\text{esClique}(S) \wedge |S| > |B|$:
 $B = S$
else if $\text{esClique}(S)$: se agrega un backtracking[⊗]
 $S' = S + \{0\}$ | no se agrega el nodo
 $\text{MaxClique}(S')$
 $S' = S + \{1\}$ | se agrega el nodo
 $\text{MaxClique}(S')$

{

[⊗] Se puede agregar la condición de
 $\dots \wedge |S| + (n - K) > |B|$ para hacerlo branch and bound
Es como que BT piensa si vale la pena calcular algo
que por ahí no cumple, mientras que B&B quiere calcular
si se puede mejorar el mejor candidato B.

Hay un problema con las llamadas recursivas: Muchas veces un programa resuelve el mismo problema muchas veces. Se repiten muchos cálculos.

Para evitar que pase esto, se usa la programación dinámica.

Una función recursiva f se llama muchas veces a sí misma con los mismos parámetros p .

$f(p)$

⇒ Almacenar el resultado de $f(p)$ así no se calcula de nuevo.

⇒ $p \rightarrow$ estado.

⇒ $f(p)$ llamada muchas veces con el mismo p
→ superposición de estados.

	0	1	2	3	4	...	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
⋮								
$k-1$	1						1	
k	1						1	
⋮								
$n-1$	1							
n	1							

{ se repiten
} estos cálculos!

Ejemplo: cálculo de coef. binomiales $\binom{n}{k}$.

Complejidad exponencial.

Con programación dinámica la complejidad es $O(nk)$.

Se guardará el resultado de cada llamado (memoización)

Si se repite un llamado, se recurre al valor almacenado.

memoización

Top-down: se comienza desde el problema más grande.

Bottom-up: se comienza desde el problema más pequeño.

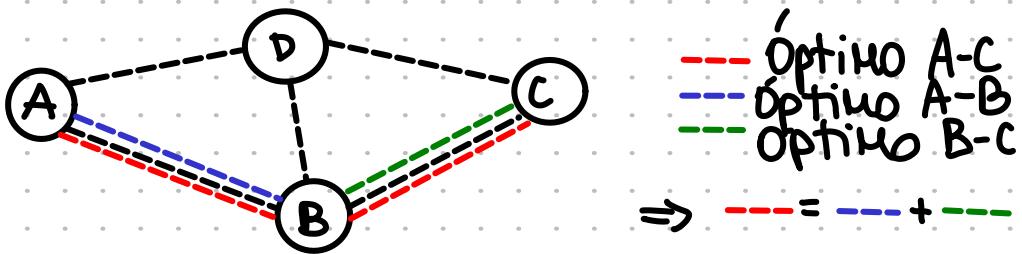
tableau-filling.

PD = Recursión con sup. de estados + $\left\{ \begin{array}{l} \text{memoización} \\ \text{or} \\ \text{tableau-filling} \end{array} \right\}$

¿Cómo se resuelve un problema de optimización con PD?

Principio de optimalidad:

Un problema tiene una **subestructura óptima** si la solución óptima se puede expresar a partir de las soluciones óptimas de los subproblemas.



- Si $v_i, v_1, v_2, \dots, v_k, v_f$ es el camino más corto entre v_i y v_f , entonces $v_1, v_2, \dots, v_k, v_f$ es el camino más corto entre v_2 y v_f .
 - De hecho cada subcamino de $v_i, v_1, v_2, \dots, v_k, v_f$ es el camino más corto entre el primer y último nodo de ese subcamino!

$Z^*(v)$ = distancia mínima entre cada nodo v y v_f .

Definición recursiva: $Z^*(v) = \min_{\substack{\text{costo/} \\ \text{dist óptima}}} \{ 1 + Z^*(w) \}$

WEV,
 $w \neq v$

primer paso
 (cada arista tiene costo 1)

Parámetro/estado: nodo inicial v .

Camino mínimo $v_i, v_1, v_2, \dots, v_k, v_f \rightarrow$ solución óptima.

Problema del cambio

- ▷ Oqueremos dar el vuelto usando el Mínimo de Monedas.
- ▷ Monedas: $\$1, \$5, \$10, \25 .
denominaciones
 $a_1 \ a_2 \ a_3 \ a_4$
- ▷ Vuelto a dar: $\$69$.
Objetivo $t = \$69$

Formalmente:

- ▷ Denominaciones a_1, \dots, a_k en orden decreciente.
- ▷ Objetivo t
- ▷ Cantidad de monedas de la denominación i : x_i a entregar en el vuelto.

$$\therefore \min_{x_1, \dots, x_k} \sum_{i=1}^k x_i \quad \text{Mínimo número de monedas.}$$

Sujeto a:

$$t = \sum_{i=1}^k x_i \cdot a_i \quad \text{Para devolver el vuelto exacto.}$$

Definición recursiva:

$$s = 0, \dots, t$$

$Z^*(s) \rightarrow$ Mínimo de monedas para devolver s centavos.

$$\Rightarrow Z^*(s) = \begin{cases} 0 & s=0 \\ \min_{i: a_i \leq s} 1 + Z^*(s - a_i) & s \neq 0 \end{cases}$$

Parámetro: Vuelto s

Sol. óptima: x_1^*, \dots, x_k^*

monedas de c/u
denominación.

Valor óptimo: $Z^*(s)$

Sumamos una moneda para devolver más las que faltan para el vuelto que queda devolver.

RECUSIÓN NORMAL

```
int cambio(int s)
{
    if (s == 0)      No hay más vuelto que dar
                    (ya le dimos todo el vuelto)
                    return 0;

    int ret = infinito;  Valor inicial de cuántas monedas hubo que dar
    for (int i=0; i<k; ++i) Recorremos todas las monedas que tenemos
    {
        if (a[i] <= s) Si la moneda entra en el vuelto
            ret = min(ret, 1 + cambio(s-a[i]));
    }           Sumamos la moneda al contador y calculamos la solución óptima del
                vuelto restante por devolver.

    return ret;
}
```

```

int cambio(int s, int* Z)
{
    Caso base
    if (s == 0)
        return 0;

```

Memoización

if ($Z[s] \geq 0$) // Inicializado con -1^{ss} .

return $Z[s]$;

Valor cualquiera

```

int ret = infinito;
for (int i=0; i<k; ++i)
{

```

Resolución

if ($a[i] \leq s$) → si se puede usar la moneda para devolver el cambio

$ret = \min(ret, 1 + \text{cambio}(s-a[i], Z))$

Almacenar sol. encontrada para s .

$Z[s] = ret$;

return ret;

}

RECUSIÓN TOP-DOWN + MEMOIZACIÓN

Memoización: guardar el número mínimo de monedas para sumar s .

Si ya lo calculamos ($\neq -1$) lo devolvemos así no lo calculamos de nuevo

BOTTOM-UP + MEMOIZACIÓN

int cambio(int t) → objetivo final

```

{
    int *Z = new int[t+1];
    Z[0] = 0; → devolver 0 => 0.

```

```

for (int s=1; s<=t; ++s)
{
    int ret = infinito;
    for (int i=0; i<k; ++i)
    {
        if ( $a[i] \leq s$ ) Si se puede
            ret = min(ret, 1 +  $Z[s-a[i]]$ );
    }
    Z[s] = ret;
}

```

return $Z[t]$;

se itera desde $1 \rightarrow t$.

Calcula cada $Z[s]$ usando valores ya conocidos ($Z*[s-a[i]]$)

Resolución de subproblemas

SIN RECUSIÓN!!

PROBLEMA DE LA MOCHILA

Definición recursiva:

$Z^*(K, c) \rightarrow$ óptimo con los primeros K objetos y capacidad restante c .

★ $Z^*(K, c) = 0$ si $K=0$ ó $c=0$

★ $Z^*(K, c) = Z^*(K-1, c)$, si $K>0$ y $P_k > c$

★ $Z^*(K, c) = \max\{Z^*(K-1, c), b_k + Z^*(K-1, c - P_k)\}$
en caso contrario.

$Z^*(K, c)$ es el valor óptimo del problema con K objetos y capacidad c .

→ Parámetro: tupla (K, c) .

→ Solución óptima: subconjunto

$S^* \subset \{1, 2, \dots, n\}$ de objetos a meter en la mochila.

Algoritmo recursivo:

```

MOCHILA( $k : \mathbb{Z}, c : \mathbb{Z}$ )
  if  $k == 0 \text{ || } c == 0$  then  Sin ítems o sin capacidad    ▷ Estamos en una hoja
    return 0
  else
    ▷ Falta considerar más elementos
     $v_{\text{without}} = \text{MOCHILA}(k - 1, c)$   NO tomar el ítem  $k$ 
     $v_{\text{with}} = -\infty$ 
    if  $p_k \leq c$  then  Si entra, puedo evaluar tomarlo
       $v_{\text{with}} = b_k + \text{MOCHILA}(k - 1, c - p_k)$ ;  Tomo  $k$  (tengo una opción menos
        objetos), resto su peso a la
        capacidad disponible.
    end if
    return  $\max\{v_{\text{with}}, v_{\text{without}}\}$   ¿Tomamos o no tomamos el objeto  $k$ ?
  end if

```

▷ Retorna valor óptimo

de ▷ No devuelve S^* .

Algoritmo DP top-down + Memoización:

- Asumimos $Z(k, c) = \text{null}$.
- La función es invocada inicialmente con $\text{MOCHILA}(n, C)$.

```
MOCHILA( $k : \mathbb{Z}, c : \mathbb{Z}, Z$ )  
if  $k == 0 \parallel c == 0$  then
```

$Z(k, c) = 0$

return 0

else \triangleright Casos base

if $Z(k, c) \neq \text{null}$ then

return $Z(k, c)$

else

$v_{\text{without}} = \text{MOCHILA}(k - 1, c, m)$

$v_{\text{with}} = -\infty$ Valor símbólico.

if $p_k \leq c$ then

$v_{\text{with}} = b_k + \text{MOCHILA}(k - 1, c - p_k, m)$; → SUMA los el beneficio y

end if

1º calc. | $Z(k, c) = \max\{v_{\text{with}}, v_{\text{without}}\}$ Guarda los la recursión con menos capacidad

vez | return $Z(k, c)$

end if

end if

Memoización

▷ Falta considerar más elementos

Si ya lo calculamos. → No usamos el objeto uno menos en consideración.

Entrá el objeto?

Guarda los la mejor decisión en Z.

vez

Algoritmo DP bottom-up (table filling):

	\dots	$c - p_k$	\dots	c	\dots
\vdots					
$k - 1$		$z^*(k - 1, c - p_k)$		$z^*(k - 1, c)$	
k				$z^*(k, c)$	
\vdots					

Sólo existe DP T-D + Memoización y DP B-U (table filling)

```
int knapsack(int n, int C, int* p, int* b)
{
    int **Z = crearMatriz(n+1, C+1);
    for (int i=0; i<=n; ++i)
        Z[i][0] = 0;
    for (int c=0; c<=C; ++c)
        Z[0][c] = 0;
    for (int k=1; k<=n; ++k)
        for (int c=1; c<=C; ++c)
            if (p[k] > c) → Si no entra
                Z[k][c] = Z[k-1][c];
            else → Si entra
                Z[k][c] = max(Z[k-1][c], b[k] + Z[k-1][c - p[k]]);
    return Z[n][C];
}
```

Complejidad cuadrática!

Acceso y modificación: $O(1)$.

Pero habíais dicho que $Z^*(K, C)$ no es la solución óptima. sino que es el valor óptimo.

Para conseguir el conjunto de objetos que realiza el valor óptimo, hay que reconstruir la solución.

Queremos conseguir el S^* .

Para la instancia con objetos $\{1, 2, \dots, K\}$ y capacidad restante C , entonces K estará en la solución óptima $\Leftrightarrow Z^*(K, C) = Z^*(K-1, C - p_K)$.

SUBSECUENCIA COMÚN MÁS

FB:

→ Una subsecuencia se obtiene quitando cero o más símbolos de A.

→ Queremos encontrar la SCML entre dos secuencias, A y B.

$$A = [a_1, \dots, a_r] \text{ y } B = [b_1, \dots, b_s]$$

Primer caso: $a_r = b_s$. Últimos coinciden

la SCML entre A y B se obtiene poniendo al final de la SCML entre $[a_1, \dots, a_{r-1}]$ y $[b_1, \dots, b_{s-1}]$ al elemento $a_r (= b_s)$.

$$\Rightarrow \text{SCML}(A, B) = \text{SCML}(A', B') + a_r$$

Segundo caso: $a_r \neq b_s$ últimos no coinciden

[a SCML entre A y B será la más larga entre:

1. SCML entre $[a_1, \dots, a_{r-1}]$ y $[b_1, \dots, b_s]$

2. SCML entre $[a_1, \dots, a_r]$ y $[b_1, \dots, b_{s-1}]$

Se calcula el problema para 1 y 2. Nos quedamos con la SCML más larga.

$$\Rightarrow \text{SCML}(A, B) = \max \{ \text{SCML}(A^1, B); \text{SCML}(A, B') \}$$

Definición recursiva:

→ $Z^*(i, j)$ es la longitud de la SCML entre $A = [a_1, \dots, a_i]$ y $B = [b_1, \dots, b_j]$.

* $Z^*(0, 0) = 0$.

* Para $j = 1, \dots, s \Rightarrow Z^*(0, j) = 0$.

* Para $i = 1, \dots, r \Rightarrow Z^*(i, 0) = 0$.

* Para $i = 1, \dots, r; j = 1, \dots, s$:

Si $a_i = b_j \Rightarrow Z^*(i, j) = Z^*(i-1, j-1) + 1$

Si $a_i \neq b_j \Rightarrow Z^*(i, j) = \underbrace{\max \{ Z^*(i-1, j), Z^*(i, j-1) \}}$

Valor óptimo del problema
(i, j) parámetro

SCML $\rightarrow [a_{k_1}, \dots, a_{k_{Z^*(r,s)}}]$ sol. óptima

Algoritmo DP B-U (table filling):

$scml(A, B)$

entrada: A, B secuencias

salida: longitud de la scml entre A y B

$Z[0][0] \leftarrow 0$ Caso base

para $i = 1$ hasta r hacer $Z[i][0] \leftarrow 0$ } Llenar con

para $j = 1$ hasta s hacer $Z[0][j] \leftarrow 0$ } Ceros

para $i = 1$ hasta r hacer

para $j = 1$ hasta s hacer

si $A[i] = B[j]$ Si coinciden los ítems

$Z[i][j] \leftarrow Z[i - 1][j - 1] + 1$ extendemos

sino

$Z[i][j] \leftarrow \max\{Z[i - 1][j], Z[i][j - 1]\}$

fin si te quedas con el antepenúltimo de

alguno

fin para

fin para

retornar $Z[r][s]$ Esta acá la SCML.

PROBLEMA DE MATRICES:

CLASE 09

Queremos multiplicar n matrices.

$$M_1 \times M_2 \times \dots \times M_n = M$$

Queremos hacer la menor cantidad de multiplicaciones.

Para hacerlo de manera óptima, tenemos que multiplicar $M_1 \times \dots \times M_i$ de un lado y $M_{i+1} \times \dots \times M_n$ del otro, y luego multiplicar los dos resultados. Generaremos dos subproblemas.

Pero estos dos subproblemas deben resolverse también de manera óptima.

Entonces:

* $M[i][j]$ es el número mínimo de multiplicaciones para calcular $M_i \times M_{i+1} \times \dots \times M_j$.

* Si M_i es de tamaño $d[i-1] \times d[i]$:

$$\rightarrow M_1: d[0] \times [1]$$

Vector con dimensiones de las matrices.

$$\rightarrow M_2: d[1] \times [2]$$

\rightarrow Y así...

* K es un punto de partición donde se separa el problema en dos subproblemas

★ Casos base:

→ $M[i][i] = 0$ Multiplicar una matriz sola no requiere ninguna op. (ya resuelta)

→ $M[i][i+1] = d[i-1] \cdot d[i] \cdot d[i+1]$

para multiplicar dos matrices consecutivas el costo es el de multiplicarlas directamente.

Resolver prob.

Mult. los dos
Operaciones

★ Recurrencia:

$$M[i][j] = \min \left\{ M[i][k] + M[k+1][j] + d[i-1] \cdot d[k] \cdot d[j] \right\}$$

Queremos probar todas las combinaciones de paréntesis y elegir la que haga menos multiplicaciones.

JOB SCHEDULING:

★ Tenemos un conjunto de trabajos J

$$J = \{1, \dots, n\}$$

★ Cada trabajo tiene fecha límite d_j y ganancia P_j

★ Queremos programar los trabajos para maximizar la ganancia total, cumpliendo las fechas límite.

Por fuerza bruta:

Podríamos probar todas las combinaciones posibles de trabajos y calcular la ganancia factible.

Por backtracking:

lo mismo que FB, sólo que se poda la rama si ya no es factible (~~violó una Fecha límite~~) o si la ganancia máxima posible no puede superar la mejor actual.

Programación dinámica:

Estado: $Z^*(k, t) \rightarrow$ Máxima ganancia al considerar los primeros k trabajos, con t trabajos ya programados.

Recurrencia:

$$Z^*(k, t) = \begin{cases} \max \left\{ Z^*(k+1, t); P_k + Z^*(k+1, t+1) \right\} & d_k > t \\ Z^*(k+1, t) & d_k \leq t \end{cases}$$

Recordar que:

k : índice del trabajo actual.

t : cantidad de trabajos ya programados.

d_k : deadline del trabajo k .

P_k : ganancia del trabajo k .

$Z^*(k, t)$: Máx ganancia posible considerando los trabajos de k en adelante, ya ocupado t slots.

$$\max \{ (z^*(K+1, t)); P_k + z^*(K+1, t+1) \} \}$$



No tomamos el trabajo K . Avanzamos al siguiente

Sí tomamos el trabajo K . Ganamos su ganancia P_k y seguimos considerando los otros, pero con un slot ocupado

$t+1$.

$$z^*(K+1, t)$$

si

$$d_K \leq t$$

No hay tiempo disponible para completarlo antes de su deadline.

Entonces, no consideramos ese trabajo y seguimos con el siguiente.

Tenemos t trabajos programados y este trabajo K vencería antes o al mismo tiempo que los ya ocupados.

si $d_K > t$

todavía hay un slot libre antes del deadline del trabajo K .

Queremos verificar si un grafo es conexo o no.

1. Partimos de un nodo inicial s
2. Marcamos sus vecinos alcanzables.
3. Repetir proceso con los vecinos descubiertos.
4. Si todos los vértices fueron visitados, el grafo es conexo.

Estados de un nodo:

- no descubierto → no visitado
- descubierto → detectado (no procesado)
- procesado → nos paramos ahí y VEMOS SUS vecinos.

↑ inicial

Recorrido(G, s):

```

para todo  $v$  en  $V$ : estado[ $v$ ] = no descubierto → init
estado[ $s$ ] = descubierto → inicial. No procesamos sus vecinos aún.
 $L = [s]$  # lista de vértices a procesar
mientras  $L$  no esté vacía: → Mientras haya vértices para procesar
     $v = \text{siguiente}(L)$  → tomamos el próx. vértice de la lista.
    para todo  $w$  en  $N(v)$ : → Vecinos de  $v$ 
        si estado[ $w$ ] == no descubierto:
            estado[ $w$ ] = descubierto → los marcamos como vistos
            agregar  $w$  a  $L$  → agregar vecino a  $L$  (para procesar)
        estado[ $v$ ] = procesado → ya vistos
        eliminar  $v$  de  $L$  → sacamos el ya visto
    
```

(COMPLEJIDADES)

Matriz de adyacencia

$$n = |V|, m = |E|$$

Obtener vecinos: $O(n)$

Algoritmo completo: $O(n^2)$

Lista de vecinos

Obtener vecinos: $O(1)$

Recorrer vecinos: $2M$

Algoritmo completo: $O(n+M)$

BFS(G, r):

la lista L se implementa como una queue
 \Rightarrow se recorre el grafo a lo ancho.
 Queue: FIFO.

BFS(G, s):

para cada v en $V(G)$: \rightarrow recorre todos los vértices.
 estado[v] = no descubierto \rightarrow Imputa estado.
 distancia[v] = ∞ \rightarrow Imputa distancia simbólica.
 estado[s] = descubierto \rightarrow nodo inicial.
 distancia[s] = 0 \rightarrow distancia a uno mismo = 0.
 Q = cola vacía \rightarrow inicializar Q.
 encolar(Q, s) \rightarrow añadir a Q.
 mientras Q no esté vacía: \rightarrow mientras queden nodos
 $v = \text{desencolar}(Q) \rightarrow$ sacamos el más antiguo
 para cada w en $N(v)$: \rightarrow para cada vecino suyo
 si estado[w] == no descubierto: \rightarrow si no se alcanzó antes
 estado[w] = descubierto \rightarrow actualizar
 distancia[w] = distancia[v] + 1 \rightarrow vecino \Rightarrow está a un hop
 encolar(Q, w) \rightarrow mandas sus vecinos a procesar
 estado[v] = procesado \rightarrow actualizar

PROFUNDIDAD.

DFS(G, r):

PILA \rightarrow LIFO

DFS(G, s):

para cada v en $V(G)$: \rightarrow recorrer todos los vértices
 estado[v] = no descubierto \rightarrow estado inicial
 estado[s] = descubierto \rightarrow estado del nodo inicial
 P = pila vacía \rightarrow iniciar stack vacío
 apilar(P, s) \rightarrow agregar inicial al stack
 mientras P no esté vacía: \rightarrow mientras queden nodos por ver
 $v = \text{desapilar}(P) \rightarrow$ sacamos el último añadido
 si estado[v] != procesado: \rightarrow Si no se proceso
 para cada w en $N(v)$: \rightarrow descubrimos sus vecinos
 si estado[w] == no descubierto:
 estado[w] = descubierto
 apilar(P, w) \rightarrow añadimos sus vecinos al stack
 estado[v] = procesado \rightarrow ya se process

ÁRBOLES:

- ! Grafo conexo y sin ciclos
- ! Hoja: Nodo de grado 1
- ! Todo árbol no trivial tiene más de dos hojas
- ! Características:

- Conexo y sin circuitos.
- Tiene $M = n - 1$ aristas.
 ↗ nodos
 ↘ aristas
- Si le sacas una arista ya no es conexo.
- Si le agregas una arista se genera un ciclo.
- Hay solo un camino simple entre cada par de vértices.

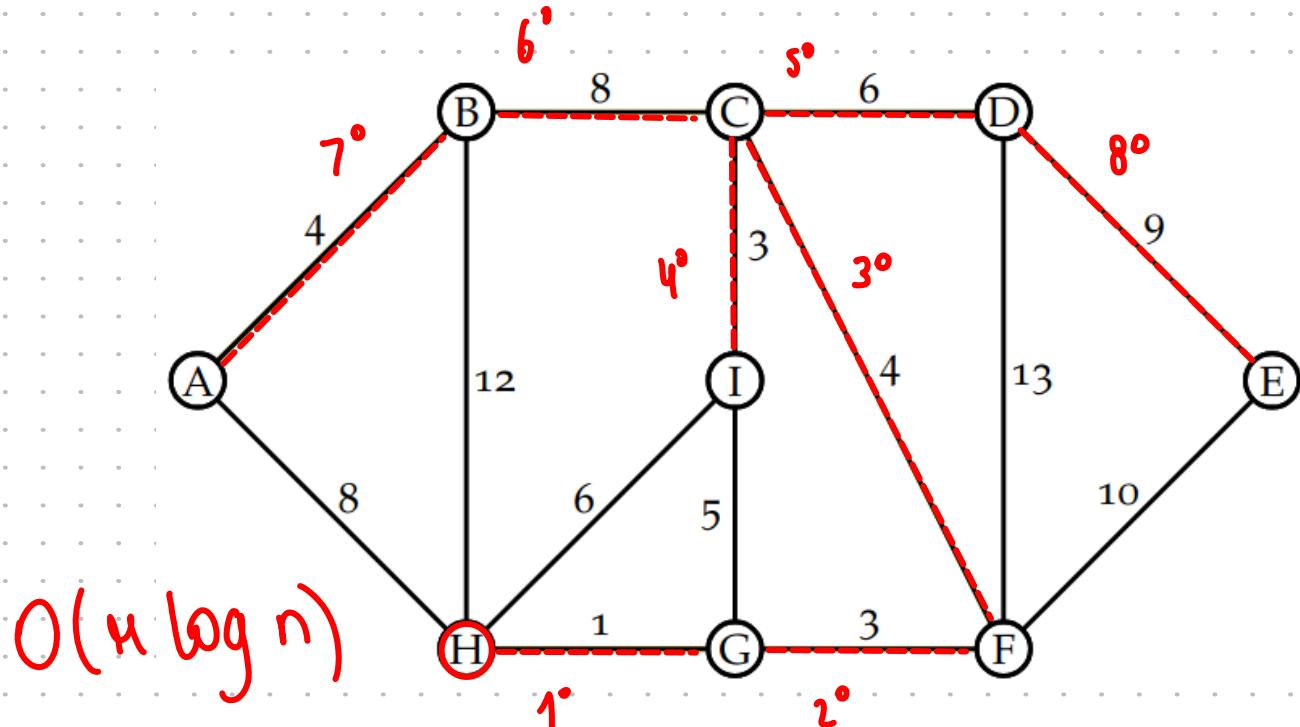
ÁRBOL GENERADOR y ÁRBOL GEN. MÍNIMO:

Un **árbol generador** es un subgrafo que es árbol y contiene todos los vértices del grafo original.

La **longitud de un árbol** (τ) es la suma de todos los pesos de sus aristas. $\sum_{e \in T} l(e)$

Un **AGM** es el árbol generador de mínima longitud.

ALGORITMO DE PRIM:



```
Algoritmo Prim(G)
Entrada:
    G = (V, E) grafo conexo con pesos  $l(e)$  en las aristas
Salida:
    T árbol generador mínimo de G

Elegir un vértice inicial u cualquiera
VT  $\leftarrow$  {u}
ET  $\leftarrow$   $\emptyset$ 

Mientras tamaño(VT) < tamaño(V) hacer
    Elegir la arista  $e = (x, y)$  de peso mínimo
        tal que  $x \in VT$  y  $y \notin VT$ 
    Agregar  $e$  al conjunto ET
    Agregar  $y$  al conjunto VT
Fin Mientras

Retornar  $T = (VT, ET)$ 
Fin Algoritmo
```

Siempre se expande el conjunto de vértices conectados con la arista más barata posible.

ALGORITMO DE KRUSKAL:

$E_T := \emptyset$

$i := 1$

mientras $i \leq n - 1$ hacer

 elegir $e \in E$ tal que $l(e)$ sea mínima entre las aristas que no forman circuito con las aristas que ya están en E_T

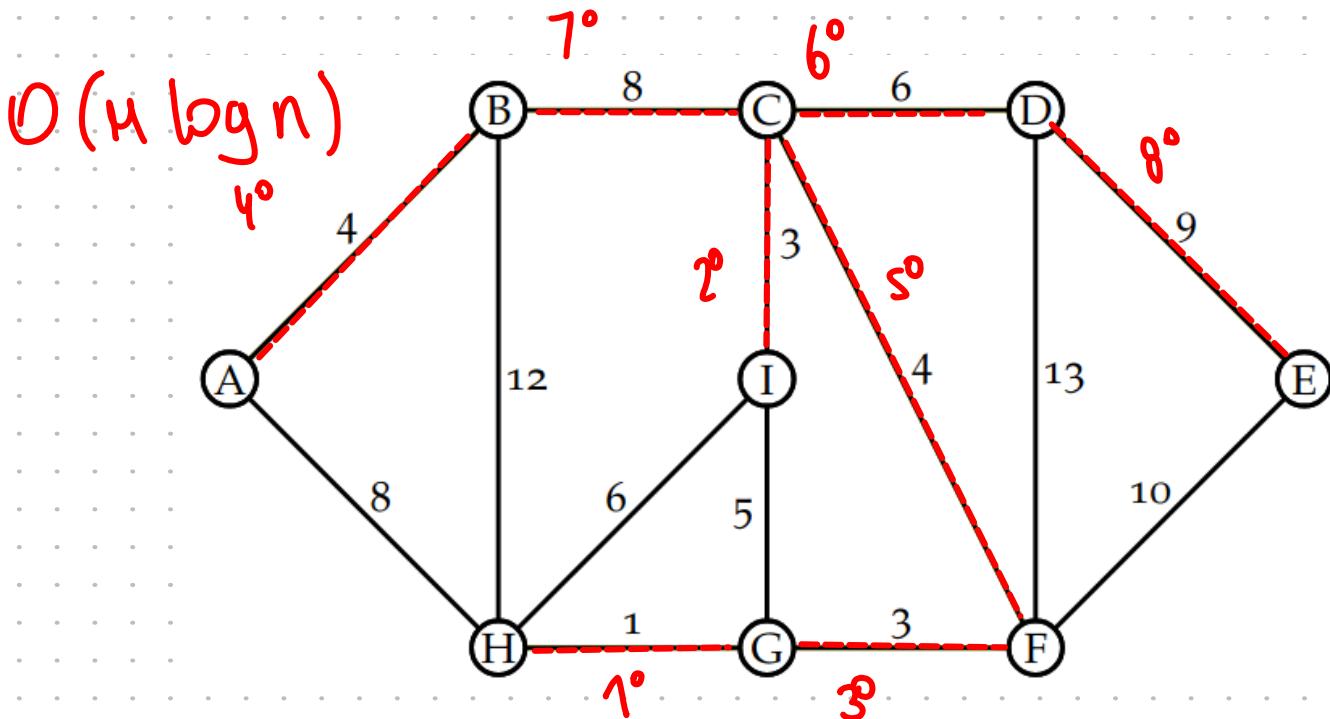
$E_T := E_T \cup \{e\}$

$i := i + 1$

retornar $T = (V, E_T)$

Selecciona aristas en orden creciente de peso, evitando ciclos.

Se puede mejorar la complejidad de Kruskal usando la estructura Union-Find.



Union find permite verificar si dos vértices están en la misma componente. Es para saber si una arista forma un ciclo.

USAMOS UN ARREGLO P DE PADRES.
CADA VÉRTICE i TIENE UN VALOR $P[i] \rightarrow$ EL PADRE DE ESE VÉRTICE.

SI $P[i] = i \Rightarrow i$ ES LA RAÍZ DE SU CONJUNTO.

TIENE TRES OPERACIONES BÁSICAS:

1) $\text{root}(i)$

ENCUENTRA EL REPRESENTANTE DEL CONJUNTO AL QUE PERTENECE i .

```
Procedimiento Root(i)
  Mientras  $P[i] \neq i$  hacer
     $i \leftarrow P[i]$ 
  Fin Mientras
  Retornar  $i$ 
Fin Procedimiento
```

$[0, 1, 1, 2, 2] \rightarrow \text{Root}(3) = 2$

2) $\text{find}(i, j) \rightarrow O(n) / O(\log n)$

COMPRUEBA SI DOS VÉRTICES ESTÁN EN EL MISMO CONJUNTO.

```
Procedimiento Find(i, j)
  Si  $\text{Root}(i) = \text{Root}(j)$  entonces
    Retornar verdadero
  Sino
    Retornar falso
  Fin Si
Fin Procedimiento
```

"SI AGREGO LA ARISTA (i, j) : SE FORMA UN CICLO?"

3) $\text{Union}(i, j) \rightarrow O(n) / O(\log n)$
Une los conjuntos de i y j .

```
Procedimiento Union(i, j)
    ri ← Root(i)
    rj ← Root(j)
    Si ri ≠ rj entonces
        P[ri] ← rj
    Fin Si
Fin Procedimiento
```

La raíz del conjunto i apuntará al conjunto j .

CAMINO MÍNIMO

CLASE 12

Tenemos un grafo dirigido y pesado $G(N, A)$, con una función de distancia $d: A \rightarrow \mathbb{R}^+$ y nodos $s, t \in N$, se busca el **camino mínimo** entre s y t .

→ más corto

Si hay ciclos de peso negativo alcanzables desde s , el problema no está bien definido. Se recorre infinitamente y se asigna un costo $-\infty$.

Si hay pesos negativos pero sin ciclos negativos, el problema sigue siendo válido.

Propiedad: una **subestructura óptima** marca que todo subcamino de un camino mínimo también será un camino mínimo.

DIJKSTRA

ASUMIMOS que todas las pesos de las aristas son positivos. No funciona con negativas.

1. Inicializar:

$d[s] = 0 \rightarrow$ dist. a uno mismo
 $d[i] = \infty$ para todo $i \neq s \rightarrow$ dist. al resto

2. Mientras existan nodos no visitados:

- a. Elegir i no visitado con menor $d[i]$
- b. Marcar i como visitado
- c. Para cada vecino j no visitado de i :

si $d[i] + dij < d[j]$ entonces
 $d[j] \leftarrow d[i] + dij \rightarrow$ act.

$O(n^2) \rightarrow$ normal
 $O(n \log n) \rightarrow$ heap

3. Retornar $d[t]$

si el nuevo camino por i es mejor que el que ya tenía

Luego de cada iteración, los nodos visitados tienen su distancia mínima definitiva.

Determina el CM desde s hacia todos los nodos.

BELLMAN-FORD

Admite pesos negativos y detecta ciclos negativos.

1. $d[s] = 0, d[i] = \infty$ para $i \neq s$

2. Para $k = 1$ hasta $|V| - 1$:

Para cada arista (i, j) :

si $d[i] + dij < d[j]$ entonces

$d[j] \leftarrow d[i] + dij$

3. (Opcional) Si en la iteración $|V|$ aún hay cambios \rightarrow existe un ciclo negativo.

$O(n \cdot m)$ peor caso

A*

Extensión de Dijkstra agregando una heurística ℓ_i de la distancia al destino.

Usa una distancia ajustada:

$$d_{ij}^* = d_{ij} + \ell_j - \ell_i$$

$$\ell_i \leq \ell_j + d_{ij}$$

Resumen:

Algoritmo	Pesos negativos	Calcula todos los caminos	Complejidad	Observaciones
Dijkstra	✗	No	$O(m \log n)$	Rápido con pesos positivos
Bellman-Ford	✓	No	$O(n \cdot m)$	Detecta ciclos negativos
Floyd-Warshall	✓	✓	$O(n^3)$	Todos los pares
A*	✗ / Heurístico	No	$O(m \log n)$	Usa heurística (p. ej. distancia euclíadiana)