# Resumen para Algoritmos y Estructuras de Datos III

Basado en esta guía de estudio.

Tomás Spognardi

18 de agosto de 2022

# Índice general

# Capítulo 1

# Técnicas de Diseño de Algoritmos

## 1.1. Complejidad

### 1.1.1. Repaso de complejidad computacional

La complejidad computacional es una técnica de análisis de algoritmos, en particular, de su tiempo de ejecución. Es de carácter *teórico*: se basa en determinar matemáticamente la cantidad de operaciones que llevará a cabo el algoritmo para una instancia de tamaño dado, indendientemente de la máquina sobre la cuál se implementa y el lenguaje utilizado.

#### Definición informal

La complejidad de un algoritmo es una función $T_A : \mathbb{N} \longrightarrow \mathbb{R}_{>0}$ que representa el tiempo de ejecución en función del tamaño de la entrada. Para distinguir entre entradas de un mismo tamaño, se pueden considerar:

- **Complejidad de peor caso**:[1]

$$T_{\text{peor}}(n) = \text{máx}\left\{t_A(i) \mid I \in I_A, |I| = n\right\}$$

- **Complejidad de mejor caso**:

$$T_{\text{mejor}}(n) = \text{mín}\left\{t_A(i) \mid I \in I_A, |I| = n\right\}$$

- **Complejidad de caso promedio**:[2]

$$T_{\text{prom}}(n) = \sum_{I \in I_A, |I| = n} P(I) \cdot t_A(I)$$

Para ciertos algoritmos, conviene hacer un análisis más profundo que distingue entre tipos de instancias particulares al problema.

[1] $t_A : I_A \longrightarrow \mathbb{R}_{>0}$ devuelve el tiempo de ejecución para una instancia particular del problema $A$.
[2] $P(I)$ es la probabilidad de que la entrada sea la instancia $I$.

Estas definiciones no son rigurosas: no contienen ninguna indicación sobre cómo determinar $T_A$ para un algoritmo $A$, y el "tiempo de ejecución" ni siquiera tiene unidades de medida. Para formalizarlas, es necesario definir un *modelo de cómputo*.

### Modelo de cómputo: Máquina RAM

La *Máquina RAM* es una máquina abstracta que funciona como modelo de cómputo. Nos permite modelar computadoras en las que la memoria es suficiente y los enteros involucrados en los cálculos entran en una palabra[3]. Este modelo cuenta con:

- **Memoria Principal**: Una sucesión de celdas numeradas (tantas como se necesiten). Cada una puede guardar un entero de tamaño arbitrario.
- **Registro Acumulador**: un registro especiel se usa como (generalmente primer) operando en las operaciones.
- **Acceso Aleatorio**: Acceso directo a cualquier celda en tiempo constante. También cuenta con direccionamiento indirecto: la dirección accedida puede ser el valor de una celda (un *puntero*).
- **Programa**: Se codifica en una serie de instrucciones, y se almacena en una memoria aparte de la principal. Hay un *contador de programa*, que identifica la próxima a ser ejecutada y puede ser manipulado a través de ciertas instrucciones (*jumps*).

Tanto la entrada como la salida son representadas como una sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario. Para codificar un programa, es necesario definir un *set de instrucciones*. Un ejemplo posible sería el siguiente[4]:

- LOAD `valor` – Carga un valor en el acumulador.
- STORE `valor` – Carga el acumulador en un registro.
- ADD `valor` –Suma el operando al acumulador
- SUB `valor` – Resta el operando al acumulador
- MULT `valor` – Multiplica el operando por el acumulador
- DIV `valor` – Divide el acumulador por el operando
- READ `valor` – Lee un nuevo dato de entrada → operando
- WRITE `valor` – Escribe el operando a la salida
- JUMP `label` – Salto incondicional
- JGTZ `label` – Salta si el acumulador es positivo
- JZERO `label` – Salta si el acumulador es cero

[3] Una *palabra* es el tamaño de una celda de memoria

[4] Este ejemplo no es de ninguna forma minimal, pero es similar a un set de instrucciones RISC para una computadora real.

- HALT – Termina el programa

Para calcular la complejidad de un programa, se asume que cada instrucción tiene un tiempo de ejecución constante. En ese caso, se puede definir $t_A(I)$ = suma de los tiempos de ejecución de las instrucciones ejecutadas por el algoritmo $A$ para la instancia $I$. Esto es casi suficiente para calcular $T_A(n)$: solo resta definir $|I|$, el tamaño de la instacia.

**Modelo uniforme**

En este modelo, cada **dato individual** ocupa una celda de memoria, y cada operación básica tiene tiempo de ejecución constante. Esto resulta razonable cuando la entrada es una estructura de datos y cada dato entra en una palabra de memoria. Bajo esta suposición, el tamaño de entrada se define como la cantidad de datos individuales de la instancia.

Sin embargo, para algoritmos que operan sobre un entero particular, esta definición no resulta adecuada. Por ejemplo, se puede tomar el siguiente algoritmo, que determina si un número es o no primo:

```
ES-PRIMO(n)
1  for i = 2 to ⌈√n⌉
2       if n ≡ 0  mód i
3            return FALSE
4  return TRUE
```

Según la definición anterior, el tamaño de la entrada de ES-PRIMO es siempre 1, lo cual es anti-intuitivo: sería conveniente poder definir la complejidad de este algoritmo en función del tamaño de $n$.

**Modelo logarítmico**

En este caso, el tamaño de la instancia se define como la cantidad de símbolos de un **alfabeto** necesaria para representarla, y el tiempo de ejecución de cada operación elemental depende del tamaño de los operandos (definido de la misma manera). Esto es apropiado para algoritmos que toman como input un número fijo de datos individuales.

Para representar los datos, se suele tomar como alfabeto $\mathbb{B} = \{0, 1\}$, los dígitos binarios. En tal caso, el tamaño de un entero $n \in \mathbb{Z}$ es $L(n) = \lceil \log_2 n \rceil + 1$ bits, mientras que para almacenar una lista de $m$ enteros se necesitan $L(m) + mL(N)$, donde $N$ es el valor máximo posible en la lista.

### 1.1.2. Notación O

Para comparar tiempos de ejecución entre distintos algoritmos, es conveniente obviar constantes de proporcionalidad y enfocarse en el comportamiento asintótico de las complejidades. Con eso en mente, se definen las clases:

$$
\begin{aligned}
f \in \mathcal{O}(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0,\ f(n) \leq c \cdot g(n) \\
f \in \Omega(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0,\ f(n) \geq c \cdot g(n) \\
f \in \Theta(g) &\iff f \in \mathcal{O}(g) \wedge f \in \Omega(g)
\end{aligned}
$$

Informalmente, $f \in \mathcal{O}(g)$ implica que $f$ crece a lo sumo tan rápido como $g$.

**Complejidades comunes**

- Si un algoritmo es$\mathcal{O}(\log n)$, se dice **logarítmico**.
- Si un algoritmo es$\mathcal{O}(n)$, se dice **lineal**.
- Si un algoritmo es$\mathcal{O}(n^2)$, se dice **cuadrático**.
- Si un algoritmo es$\mathcal{O}(n^3)$, se dice **cúbico**.
- Si un algoritmo es$\mathcal{O}(n^k)$, se dice **polinomial**.
- Si un algoritmo es $\mathcal{O}(k^n)$ $(k > 1)$, se dice **exponencial**.

Además, se tiene:

$$\mathcal{O}(n^k) \subsetneq \mathcal{O}(d^n)\ \forall k, d \in \mathbb{N}$$
$$\mathcal{O}(\log n) \subsetneq \mathcal{O}(n^k)\ \forall k \in \mathbb{R}_{>0}$$

### 1.1.3. Problemas "bien resueltos" e intractabilidad

Un problema se denomina *bien resuelto* si existe un algoritmo de tiempo polinomial que lo resuelve. Esto se debe a que el tiempo de ejecución de los algoritmos exponenciales crece demasiado rápido: su ejecución puede resultar infactible para valores de $n$ pequeños.

| | $n = 10$ | $n = 20$ | $n = 30$ | $n = 40$ | $n = 50$ |
|---|---|---|---|---|---|
| $O(n)$ | 0.01 ms | 0.02 ms | 0.03 ms | 0.04 ms | 0.05 ms |
| $O(n^2)$ | 0.10 ms | 0.40 ms | 0.90 ms | 0.16 ms | 0.25 ms |
| $O(n^3)$ | 1.00 ms | 8.00 ms | 2.70 ms | 6.40 ms | 0.12 sg |
| $O(n^5)$ | 0.10 sg | 3.20 sg | 24.30 sg | 1.70 min | 5.20 min |
| $O(2^n)$ | 1.00 ms | 1.00 sg | 17.90 min | 12 días | 35 años |
| $O(3^n)$ | 0.59 sg | 58 min | 6 años | 3855 siglos | $2 \times 10^8$ siglos! |

Tabla de comparaciones de los posibles tiempos de ejecución para distintas clases de complejidad.

Sin embargo, cabe destacar que:

- Si los tamaños de instancias no son muy grandes, un algoritmo exponencial puede ser apropiados.
- Un algoritmo puede ser polinomial, pero con un exponente o una constante demasiado grande para ser aplicado en la práctica.
- Existen ciertos algoritmos con complejidad de peor caso exponencial, pero que en la práctica son muy eficientes (como el método *simplex*).

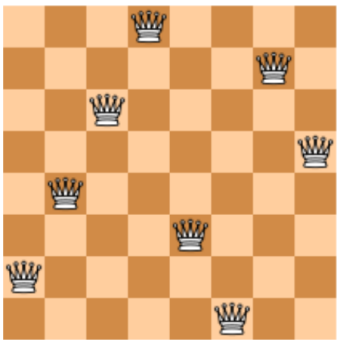
## 1.2. Backtracking

### 1.2.1. Fuerza bruta

Un algoritmo de *fuerza bruta* (también llamado de *búsqueda exhaustiva*) analiza todas las posibles configuraciones de la salida, hasta encontrar una que cumple con los requerimientos del problema.

**Ejemplo: Problema de las $n$ damas**

> **Problema:**
> Ubicar $n$ damas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna dama amenace a otra.



Solución posible para el caso $n = 8$

Un posible algoritmo de fuerza bruta sería recorrer todos los posibles subconjuntos de $n$ casillas, verificando si algún par de reinas se amenaza en caso de ser ubicarlas en las casillas del subconjunto. Sin embargo, esto no es muy eficiente: para $n = 8$, implicaría recorrer $\binom{64}{8} =$ 4,426,165,368 combinaciones.

Se pueden lograr mejoras aprovechando la estructura del problema: como cada columna debe tener exactamente 1 dama, las configuraciones exploradas pueden representarse como un vector $(a_1, ..., a_n)$, con $a_i \in \{1, ..., n\}$ indicando la fila de la dama que está en la columna $i$. Además, cada fila tiene exactamente una reina, así que los elementos del vector no se repiten. Por ende, la cantidad de combinaciones se reduce a $n!$, que para el caso $n = 8$ es $8! = 40{,}320$. No obstante, esto puede mejorarse.

### 1.2.2. Backtracking

El backtracking es una técnica general de diseño de algoritmos que consiste de extender las soluciones parciales $a = (a_1, ..., a_k), k < n$, agregando un elemento $a_{k+1}$ al final del mismo. Si se detecta que $S_{k+1}$, el conjunto de soluciones que tienen al vector como prefijo, es vacío, se retrocede a la solución anterior. Esto permite descartar configuraciones parciales apenas se determina que no pueden llevar a una solución. Los algoritmos de backtracking siguen el siguiente esquema general:

```
BT(a)
1  if ¬Es-Válida(a)
2       return
3  if Es-Solución(a)
4       Procesar(a)
5       return
6  for a' ∈ Sucesores(a)
7       BT(a')
```

Si solo se busca una solución, esto se puede volver más eficiente usando una variable global *encontró*:

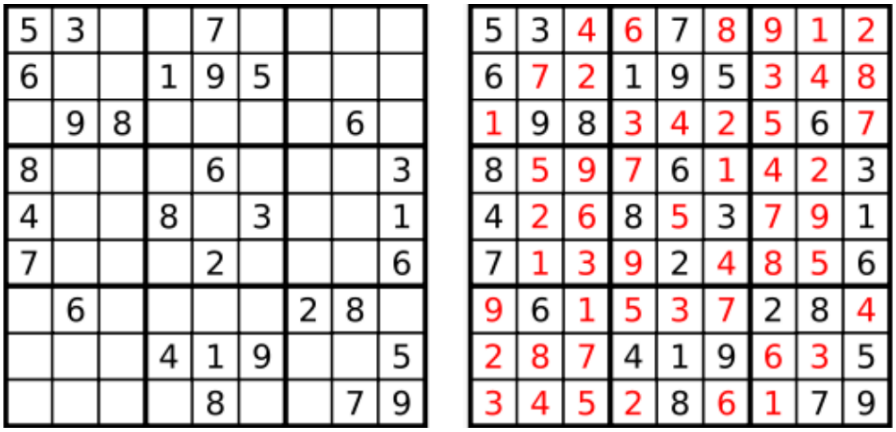
```
BT(a)
 1  if ¬Es-Válida(a)
 2       return
 3  if Es-Solución(a)
 4       sol = a
 5       encontró = true
 6       return
 7  for a' ∈ Sucesores(a)
 8       BT(a')
 9       if encontró
10            return
```

Para que el backtracking sea eficiente, el procedimiento Es-Válida debe ser capaz de identificar algún conjunto de instancias inválidas, y no puede tener una complejidad demasiado grande.

En el problema de las $n$ damas, se puede chequear en cada paso si alguna reina amenaza a la recién agregada. Esto se puede realizar en tiempo lineal, y por la construcción de las soluciones solo hace falta comprobar las amenazas diagonales. Utilizando este algoritmo, cualquier configuración de $n$ elementos que no haya sido rechazada es una solución válida.

**Ejemplo: Resolución de Sudokus**

> **Problema:**
> Encontrar una asignación de números a casillas que resuelve un Sudoku particular.



Ejemplo de un Sudoku y su solución.

Los Sudokus pueden ser resueltos con un algoritmo de backtracking: las soluciones son extendidas agregando un número a algún casillero vacío. Cuando el nuevo número no cumple alguna de las restricciones del sudoku, la solución es rechazada. A pesar de ser exponencial, este algoritmo es muy eficiente en la práctica.

## 1.3. Programación Dinámica

### 1.3.1. Definición

La *programación dinámica* (PD/DP) es otra técnica de diseño de algoritmos. Es similar al *Divide & Conquer*, ya que se basa en dividir el problema en sub-problemas de menor tamaño, resolverlos recursivamente, y combinar las sub-soluciones para formar una solución. La diferencia con este método es que PD se utiliza en casos donde estos sub-problemas suelen superponerse, y aprovecha este hecho al resolverlos una única vez.

Para evitar repetir la resolución de sub-problemas equivalentes, los algoritmos de programación dinámica siguen alguno de estos dos esquemas:

- **Enfoque "top-down"**: Se implementa el algoritmo tradicionalmente, pero los resultados se guardan en una estructura de datos indexada por los parámetros de la llamada (*memoización*). Luego, antes de resolver ejecutar el algoritmo para una llamada, se chequea si sus parámetros están en esta estructura, y en tal caso se devuelve la solución previamente calculada.

- **Enfoque "bottom-up"**: Se resuelven los sub-problemas en un orden que asegura que las llamadas recursivas de cada uno son calculadas antes que este[5], guardando los resultados de cada llamada en una tabla.

[5]Esto representa un ordenamiento topológico del árbol de llamadas de la función (en realidad, del árbol invertido, donde cada nodo tiene una arista apuntando hacia aquellos que lo tienen como sub-problema).

**Ejemplo: Cálculo de coeficientes binomiales**

**Problema:**
Calcular el valor del coeficiente $\binom{n}{k}$, definido como:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

El problema se podría resolver calculando $\binom{n}{k}$ directamente, pero esto se dificulta para valores grandes. Por ejemplo, a pesar de que $\binom{100}{99} = 100$, el valor 100! es un número de 157 cifras, muy por encima del límite de 64 bits utilizados para representar enteros.

Una forma alternativa de realizar la operación sería haciendo uso del siguiente teorema:

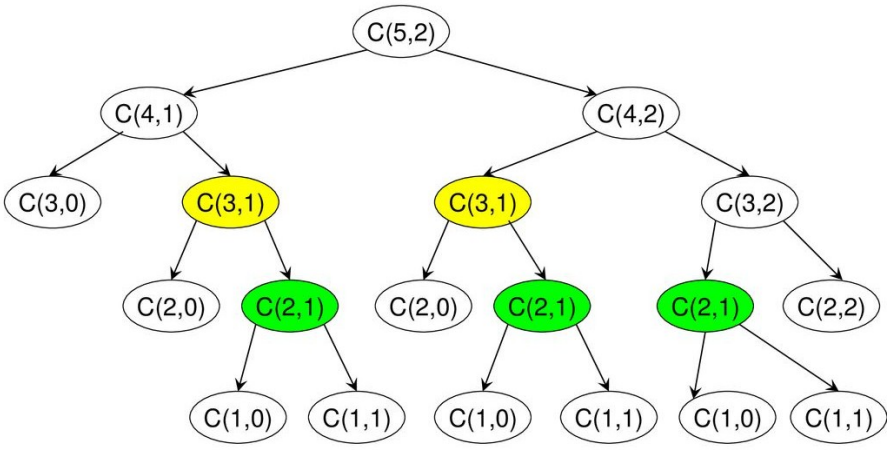
**Teorema.** *Si* $n \geq 0$ *y* $0 \leq k \leq n$*, entonces:*

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \end{cases}$$

Esta fórmula recursiva se puede implementar directamente:

```
COMBINATORIO(n, k)
1  if k == 0 ∨ k == n
2       return 1
3  else
4       return COMBINATORIO(n − 1, k − 1) + COMBINATORIO(n − 1, k)
```

Este método tiene una complejidad de$\Omega(\binom{n}{k})$, y evita calcular factoriales, pero podría ser más eficiente, ya que al ejecutarlo se repiten llamadas con los mismos parámetros.




Árbol de llamadas de COMBINATORIO para la instancia
$n = 5, k = 2$

Acá es donde entra en juego la programación dinámica. El siguiente algoritmo bottom-up calcula una única vez cada coeficiente necesario:

COMBINATORIO-PD$(n, k)$
1 Inicializar matriz $A \in \mathbb{N}^{n \times k}$
2 **for** $i = 1$ **to** $n$
3 $\quad A[i][0] = 1$
4 **for** $j = 0$ **to** $k$
5 $\quad A[j][j] = 1$
6 **for** $i = 2$ **to** $n$
7 $\quad$ **for** $j = 2$ **to** mín $\{i - 1, k\}$
8 $\quad\quad A[i][j] = A[i - 1][j - 1] + A[i - 1][j]$

La complejidad de este método es$\mathcal{O}(nk)$, y $\mathcal{O}(nk) \subseteq \mathcal{O}(n^2)$, ya que $k \leq n$. Además, se puede implementar con una complejidad espacial de$\mathcal{O}(k)$ almacenando solo la fila actual y la anterior de la tabla en el ciclo.

### 1.3.2. Principio de optimalidad de Bellman

Un problema satisface el *principio de optimalidad de Bellman* cuando para cualquier *sucesión óptima* de decisiones, cada *subsucesión* es a su vez óptima para el subproblema asociado. Esta es una condición necesaria para que aplicar PD sea eficiente.

**Ejemplo: Problema de la mochila**

**Problema:**

Dados

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ del objeto $i$.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto $i$.

Determinar qué objetos se deben incluir en la mochila para **maximizar** el beneficio total, sin **excederse** del peso máximo $C$. Formalmente, encontrar:

$$\arg\max \left\{ \sum_{s \in S} b_s \mid S \subseteq \{1, ..., n\},\ \sum_{s \in S} p_s \leq C \right\}$$

Para resolver este problema utilizando PD, se puede definir la función $m(k, D)$ como el valor óptimo para el problema considerando solo los primeros $k$ objetos y una mochila con capacidad $D$. Los valores de esta función pueden ser guardados en una tabla de $n \times C$ posiciones. Los valores se pueden calcular de manera recursiva:

$$m(k, D) = \begin{cases} 0 & \text{si } k = 0 \vee D \leq 0 \\ \max\{m(k-1, D), b_k + m(k-1, D-p_k)\} & \text{en caso contrario} \end{cases}$$

Esto contempla, para cada $k$ dos posibilidades: o bien el objeto de índice $k$ está en la solución óptima, y entonces $m(k, D) = b_k + m(k-1, D-p_k)$, o bien no, en cuyo caso $m(k, D) = m(k-1, D)$.

Si esta función se implementa directamente en un algoritmo de PD (ya sea top-down o bottom-up) utilizando una matriz como estructura de memoización, tanto la complejidad temporal como la espacial son$\mathcal{O}(nC)$. Esta complejidad es *pseudopolinomial*: está acotada por un polinomio, pero este incluye valores númericos del input, no solo el tamaño del mismo.

**Solución óptima**

Calcular $m(k, D)$ nos da el *valor óptimo*, pero no la *solución óptima*. Para obtener el conjunto de objetos que resulta en ese valor se debe reconstruir a partir de la tabla calculada. El esquema general para la reconstrucción se basa en la observación anterior: recorriendo los índices de atrás para adelante, si $m(k, D) = b_k + m(k-1, D-p_k)$, entonces el valor $k$ está en (alguna) solución. Si no, es porque $m(k-1, D) > b_k + m(k-1, D-p_k)$, es decir, ignorar el objeto $k$ resulta en un mayor beneficio total. Este procedimiento permite obtener el conjunto solución en tiempo lineal (una vez que ya se ejecutó el algoritmo anterior).

**Ejemplo: Multiplicación de matrices**

> **Problema:**
> Dadas $M_1, M_2, ..., M_n$, calcular:
>
> $$M = M_1 \times M_2 \times ... \times M_n$$
>
> Realizando la menor cantidad de multiplicaciones entre números de punto flotante.

La dificultad de este problema radica en que la cantidad de operaciones realizadas depende de la forma en la que se asocie el producto. Para resolverlo, se puede observar que alguna de las multiplicaciones tiene que ser la última realizada, es decir, para algún $i$, se deben multiplicar primero las matrices de 1 a $i$ por un lado y las de $i+1$ a $n$ por el otro, y finalmente multiplicar estos 2 resultados. Estos dos sub-problemas ($M_1 \times M_2 \times ... \times M_i$ y $M_{i+1} \times M_{i+2} \times ... \times M_n$) deben ser resueltos, a su vez, de forma óptima.

Luego, suponiendo que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$ tal que $M_i \in \mathbb{R}^{d[i-1] \times d[i]}$, se puede implementar el siguiente algoritmo bottom-up:

MIN-OPERACIONES($d$)
1 Inicializar la matriz $m \in \mathbb{N}^{n \times n}$
2 **for** $i = 1$ **to** $n$
3 $\quad m[i][i] = 0$
4 **for** $i = 1$ **to** $n - 1$
5 $\quad m[i][i+1] = d[i-1]d[i]d[i+1]$
6 **for** $s = 2$ **to** $n - 2$
7 $\quad$ **for** $i = 1$ **to** $n - s$
8 $\quad\quad m[i][i+s] = \text{mín}\{m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s] \mid i \leq k < i+s\}$

En este caso, $m[i][j]$ representa la cantidad mínima de operaciones necesarias para calcular $M_i \times M_{i+1} \times ... \times M_j$, y por ende el valor óptimo es $m[1][n]$. Para obtener la secuencia de multiplicaciones, se puede emplear un procedimiento similar al del ejemplo anterior.

## 1.4. Algoritmos Golosos

### 1.4.1. Heurísticas

Una *heurística* para un problema dado es un procedimiento computacional que intenta obtener soluciones de "buena calidad" para el mismo. Por ejemplo, para un problema de optimización, una heurística obtendría una solución con un valor cercano al óptimo.

Un algoritmo $A$ es $\epsilon$-aproximado cuando:

$$\left| \frac{x_A - x^*}{x^*} \right| \leq \epsilon$$

Donde $x^*$ es el valor óptimo, y $x_A$ es el resultado del algoritmo.

Un ejemplo práctico es el algoritmo de Christofides y Serdykuov, un algoritmo $\frac{1}{2}$-aproximado para instancias del problema del viajante de comercio que forman un espacio métrico (las distancias son simétricas y obedecen la desigualdad triangular). Lo notable de este algoritmo es que tiene complejidad polinómica, siendo el TSP un problema NP-Completo.

### 1.4.2. Algoritmos golosos

Los *algoritmos golosos* se basan en construir una solución para un problema seleccionando en cada la "mejor" alternativa, sin considerar (o haciéndolo débilmente) las implicancias posteriores de esa selección. Habitualmente, proporcionan heurísitcas sencillas para los problemas de optimización, produciendo soluciones razonables (aunque subóptimas) en tiempos eficientes. Sin embargo, existen casos donde la solución que generan es óptima.

#### Ejemplo: Problema de la mochila

A pesar de haber resuelto el problema anteriormente, un enfoque goloso puede proveer soluciones (subóptimas) con mayor eficiencia temporal. El esquema general es agregar a la mochila el objeto $i$ que...

1. ...tenga el mayor beneficio $b_i$.
2. ...tenga el menor peso $p_i$.
3. ...maximice $\frac{b_i}{p_i}$ (la "densidad").

Se puede demostrar que, si se corre el algoritmo goloso 2 veces, una con el primer criterio y otra con el segundo, alguno de los resultados tiene un valor de al menos la mitad de la solución óptima. Esto hace al procedimiento un algoritmo $\frac{1}{2}$-aproximado, y se puede implementar en tiempo$\mathcal{O}(n \log n)$ si se ordenan los elementos previamente (es aún más eficiente usar una colas de prioridad implementadas con heap).

Por otro lado, si cambia el problema, permitiendo poner una fracción de cada elemento en la mochila, el algoritmo goloso que utiliza el tercer criterio devuelve soluciones óptimas.

#### Ejemplo: Problema del cambio

> **Problema:**
> Dado un monto $m$ y un conjunto de denominaciones $d_1, ..., d_k$, encontrar la mínima cantidad de monedas necesarias para obtener el valor $m$.

Para encontrar soluciones (no necesariamente óptimas) de este problema, se puede emplear un algoritmo goloso simple: en cada paso, seleccionar la moneda de mayor valor que no exceda el monto restante.

DAR-CAMBIO$(D, m)$
1 $suma = 0$
2 $M = \{\}$
3 **while** $suma < m$
4 $\quad pr\acute{o}xima = \max\{d \mid d \in D, d \leq m\}$
5 $\quad M = M \cup \{pr\acute{o}xima\}$
6 $\quad suma = suma + pr\acute{o}xima$
7 **return** $M$

Para ciertos conjuntos de denominaciones, como el tradicional ($\{1, 5, 10, 25, 50\}$), este algoritmo siempre devuelve soluciones óptimas, mientras que para otros no (en $D = \{1, 5, 10, 12\}, m = 21$, el algoritmo devuelve un conjunto de 6 monedas cuando la solución óptima tiene 3).

El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

**Ejemplo: Tiempo de espera total en un sistema**

> **Problema:**
> Un servidor tiene $n$ clientes que puede atender en cualquier orden, y el tiempo necesario para atender al cliente $i$ es $t_i \in \mathbb{R}_+$. Encontrar un orden de atención que minimice el tiempo de espera total de todos los clientes.

Si se denota $I = (i_1, ..., i_n)$ al orden de atención, el tiempo de espera total $T$ se puede calcular de la siguiente manera:

$$T = t_{i_1} + (t_{i_1} + t_{i_2}) + ... = \sum_{k=1}^{n} (n - k) t_{i_k}$$

Se puede plantear el siguiente algoritmo goloso: En cada paso, atender al cliente pendiente que tenga el menor tiempo de atención. La idea detrás de ese criterio es que el tiempo de los clientes que son atendidos primero tendrá que ser esperado por todos los demás, así que lo ideal es que sea el mínimo. Formalmente, la solución $I = (i_1, ..., i_n)$ es una que cumple $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, ..., n - 1$.

En este caso, la solución que proporciona el algoritmo resulta ser óptima. Por otro lado, la complejidad temporal es$\mathcal{O}(n \log n)$, ya que el procedimiento es equivalente a ordenar a los clientes por tiempo de espera.

## 1.5. Algoritmos Probabilísticos

Un *algoritmo probabilístico* es uno que emplea un grado de aleatoriedad en su ejecución. Los efectos de esta aleatoriedad pueden variar: en algunos casos solo varía el tiempo de ejecución, mientras que en otros la salida tiene una probabilidad de ser incorrecta.

### 1.5.1. Algoritmos numéricos

Un *algoritmo numérico probabilístico* es uno que aproxima la solución a un problema matemático. Estos algoritmos suelen ser adaptaciones aleatorizadas de algoritmos clásicos, como el método de cuadratura bayesiana para la integración numérica, o el de optimización bayesiana para problemas de optimización.

### 1.5.2. Algoritmos de Monte Carlo

Los *algoritmos de Monte Carlo* son aquellos que proporcionan una respuesta que tiene cierta probabilidad (típicamente baja) de ser incorrecta. En general, si estos algoritmos se corren varias veces, la probabilidad de que la respuesta obtenida sea correcta aumenta (asumiendo independencia entre las distintas ejecuciones). Un ejemplo de estos algoritmos sería el test de primalidad de Solovay-Strassen, que siempre identifica a números primos correctamente, pero tiene una probabilidad menor a $\frac{1}{2}$ de devolver una respuesta falsa para los compuestos.

### 1.5.3. Algoritmos de Las Vegas

Los *algoritmos de Las Vegas* siempre devuelven una respuesta cuando terminan, pero su tiempo de ejecución es aleatorio (potencialmente infinito). Un ejemplo podría ser un algoritmo para el problema de $n$ damas que chequea configuraciones aleatorias hasta encontrar una que satisface las restricciones.

### 1.5.4. Algoritmos de Sherwood

Los *algoritmos de Sherwood* son algoritmos que aleatorizan procesos determinísticos, habitualmente aquellos que tienen una gran diferencia entre el peor caso y el promedio. El ejemplo clásico es el algoritmo de quicksort con pivote seleccionado aleatoriamente.

# Capítulo 2

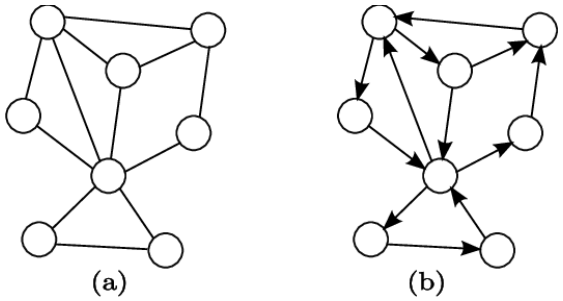
# Introducción a Teoría de Grafos

## 2.1. Grafos

### 2.1.1. Definición

Un *grafo* es un par ordenado $G = (V, E)$: el conjunto $V$, de *vértices* o *nodos* , y el de aristas/arcos $E$, que relacionan a esos nodos.

En el caso de los grafos *no dirigidos* (o simplemente grafos), $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V, v_1 \neq v_2\}$ es un conjunto de pares no ordenados de los elementos de $V$, conocidos como *aristas*. Por otro lado, para los *grafos dirigidos* (también llamados *digrafos*), $E \subseteq V \times V$ tiene pares ordenados de nodos, y sus elementos se denominan arcos. Se suele realizar un abuso de notación menor, utilizando $(v, w)$ para referirse tanto a aristas como arcos.




Representaciones gráficas de un grafo (a) y un digrafo (b). Los círculos son los vértices, y las líneas/flechas son las aristas/arcos.

En general, se denota $n_G = |V|$ y $m_G = |E|$ para referirse a las cantidades de vértices y aristas. Cuando el grafo referido es inambigüo, se omite el subíndice[1].

[1] Lo mismo vale para el resto de las definiciones en esta sección.

### 2.1.2. Vecinos

Dados $v, w \in V$, se denominan *adyacentes* cuando $e = (v, w) \in E$, y que $e$ es *incidente* a $v$ y $w$. Similarmente, la *vecindad* de $v$, denotada por $N_G(v)$ es el conjunto de vértices adyacentes a $v$, es decir:

$$N_G(v) = \{w \in V \mid (v, w) \in E\}$$

Por otro lado, la cantidad de aristas incidentes a un vértice $v$ se llama *grado*, definida como:

$$d_G(v) = |N_G(v)|$$

**Teorema.** *Dado un grafo de $G = (V, E)$, la suma de los grados de sus vértices es el doble de la cantidad de aristas. Es decir,*

$$\sum_{v \in V} d(v) = 2m$$

*Demostración.* Se puede demostrar por inducción en $m$, la cantidad de aristas.

**Caso base:** Se puede tomar como caso base $m = 0$. En un grafo sin aristas, todos los vértices tienen grado 0, y por ende:

$$\sum_{v \in V} d(v) = 0 = 2m$$

**Paso inductivo:** Asumiendo que la propiedad se cumple para $m = k$, tomemos un grafo cualquiera $G = (V, E)$ con $|E| = k + 1$ aristas. Se puede elegir una arista cualquiera $e = (v, w) \in E$, y construir el grafo $G' = (V, E - e)$ que resulta de quitar una de sus aristas. Como $m_{G'} = k$, se cumple la hipótesis inductiva:

$$\sum_{u \in V} d_{G'}(u) = 2m_{G'} = 2k$$

Luego, para el grafo original $G$, la adición de la arista $e$ solo incrementea el grado de los vértices $v$ y $w$. Concretamente:

$$d_G(u) = \begin{cases} d_{G'}(u) + 1 & \text{si } u = v \vee u = w \\ d_{G'}(u) & \text{en caso contrario} \end{cases}$$

Por lo tanto, se tiene:

$$\begin{aligned} \sum_{u \in V} d_G(u) &= \sum_{v \in V - \{v,w\}} d_{G'}(u) + (d_{G'}(v) + 1) + (d_{G'}(w) + 1) \\ &= \sum_{u \in V} d_{G'}(u) + 2 \overset{HI}{=} 2k + 2 = 2(k + 1) = 2m_G \end{aligned}$$

Lo cual, por inducción, implica que la propiedad vale para todo $m \in \mathbb{N}$.

□

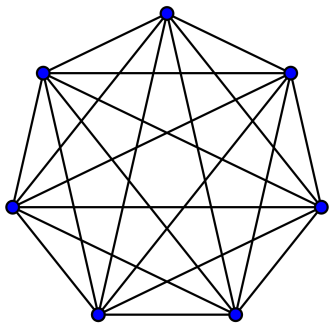
**Complemento**

Dado un grafo $G = (V, E)$, su *grafo complemento*, denotado como $\bar{G} = (V, \bar{E})$, tiene el mismo conjunto de vértices, pero cada par de vértices es adyacente en $\bar{G}$ si y solo si no lo es en $G$. Es decir,

$$\bar{E} = (V \times V) - E$$

**Grafos Completos**

El grafo $K_n$ es el *grafo completo* de $n$ vértices, los cuales son todos adyacentes entre sí. Este grafo tiene $m_{K_n} = \frac{n(n-1)}{2}$.



Representación gráfica del grafo completo $K_7$.

### 2.1.3. Generalizaciones

Algunas generalizaciones[2] de los grafos son:

- **Multigrafos**: En un multigrafo, $E$ pasa a ser un multiconjunto, es decir, pueden haber varias aristas entre un mismo par de vértices.
- **Pseudografo**: Los pseudografos pueden tiene varias aristas entre un mismo par de vértices, y también puede haber aristas que unan a un mismo par de vértices (llamadas *loops*).

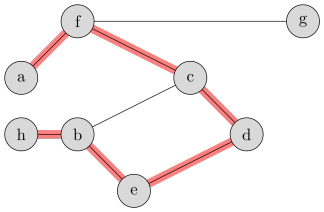
### 2.1.4. Recorridos

- Un *recorrido* en un grafo es una secuencia de vértices $P = v_0 v_1 \cdots v_k$ tal que todos los pares consecutivos son adyacentes, es decir, $(v_i, v_{i+1}) \in E \ \forall i = 0, ..., k-1$. Para multi- y pseudo-grafos, se debe especificar entre qué aristas se pasa.
- Un *camino*[3] es un recorrido que no pasa por el mismo vértice 2 veces.
- Una *sección* de un recorrido $P$ es una subsecuencia $S = v_i v_{i+1} \cdots v_j$ de vértices consecutivos de $P$, y se denota $P_{v_i v_j}$.
- Un *circuito* es un recorrido que empieza y termina en el mismo vértice.

[2] No se estudian mucho en la materia.

[3] Hay ambigüedad en el término: a veces se llama camino a los recorridos, en cuyo caso a los recorridos sin vértices repetidos se les dice *camino simple*.

- Un *ciclo* o *circuito simple* es un circuito de 3 o más vértices que no pasa 2 veces por el mismo vértice (salvo por el principio y fin).




Un ejemplo de un camino entre los vértices $a$ y $h$.

### 2.1.5. Distancia

Dado un recorrido $P$, su *longitud*, $l(P)$, es la cantidad de aristas que tiene. Luego, la *distancia* entre $v$ y $w$ se define como la longitud del camino más corto entre $v$ y $w$, y se llama $d(v, w)$. Si no hay recorrido entre $v$ y $w$, se define que $d(v, w) = \infty$, mientras que $d(v, v) = 0$ para cualquier $v$.

**Teorema.** *Si un recorrido $P$ entre $v$ y $w$ cumple $l(P) = d(v, w)$, entonces es un camino.*

*Demostración.* Se puede demostrar por el absurdo: si $P$ no fuera un camino, tendría algún vértice $u$ por el que se pasa 2 veces: $P = v \cdots u \cdots u \cdots w$. Si se forma un nuevo recorrido $P' = P_{vu} + P_{uw}$ (excluyendo el recorrido de $u$ a sí mismo), este tendría una longitud estrictamente menor que $P$, y por ende $l(P') < d(v, w)$ (**Absurdo**).

□

**Teorema.** *Para cualquier grafo $G = (V, E)$, la función de distancia $d : V \times V \longrightarrow \mathbb{N}$ es una métrica, es decir, cumple las siguientes propiedades para todo $u, v, w \in V$:*

- $d(u, v) = 0 \iff u = v$
- $d(u, v) = d(v, u)$
- $d(u, w) \leq d(u, v) + d(v, w)$ *(desigualdad triangular)*

*Demostración.* Se demuestra por separado:

- La ida vale por definición, y la vuelta vale porque cualquier camino entre un par de vértices tiene al menos 1 arista (y por ende $d(u, v) \geq 1$).
- En un grafo las aristas no tienen sentido, así que cualquier camino puede ser invertido para formar un camino válido. Por ende, la longitud del camino más corto entre $u$ y $v$ debe ser la misma que entre $v$ y $u$.
- Si $P_{uv}$ y $P_{vw}$ son caminos tales que $l(P_{uv}) = d(u, v)$ y $l(P_{vw}) = d(v, w)$, se pueden concatenar para formar un recorrido $P_{uv} + P_{vw}$ entre $u$ y $w$. Como la distancia es la longitud

mínima entre todos los recorridos, se tiene $d(u,w) \leq l(P_{uv} + P_{vw}) = d(u,v) + d(v,w)$.

□

### 2.1.6. Subgrafos

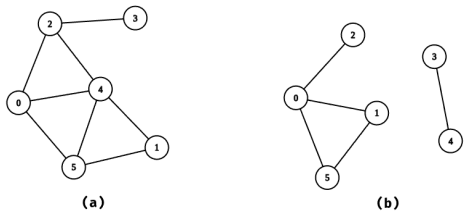
Dado un grafo $G = (V_G, E_G)$,

- Un *subgrafo* de $G$ es un grafo $H = (V_H, E_H)$ tal que $V_H \subseteq V_G$ y $E_H \subseteq E_G \cap (V_H \times V_H)$. Los notamos como $H \subseteq G$.
- $H$ es un *subgrafo propio* cuando $H \subseteq G$ y $H \neq G$.
- $H$ es un *subgrafo generador* cuando $H \subseteq G$ y $V_H = V_G$.
- $H$ es un *subgrafo inducido* cuando $(v,w) \in E_H \iff v,w \in V_H \wedge (v,w) \in E_H$. Estos subgrafos pueden definirse únicamente por su conjunto de vértices, y se denota como $G_{[V_H]}$.

### 2.1.7. Conectividad

Un grafo se denomina *conexo* cuando existe un camino entre todo par de vértices. Una *componente conexa* de un grafo es un subgrafo inducido conexo maximal (no se pueden agregar más vértices y mantenerlo conexo) de $G$.

Por otro lado, una arista de $G$ es *puente* si $G - e$ tiene más componentes conexas que $G$.




Un grafo conexo (a) y uno disconexo (b).

### 2.1.8. Representación de Grafos

Existen distintas alternativas para representar grafos en un algoritmo, que proveen ventajas y desventajas a la hora de realizar diversas operaciones.

#### Lista de aristas

El grafo se almacena como una lista de pares de vértices, que representan sus aristas. Esta es la forma más simple de representarlo, y es el formato que se asume que tiene la entrada de cualquier algoritmo de grafos. Debido a su falta de estructura, realizar la mayoría de las operaciones resulta costoso, con la excepción de agregar nodos o aristas.

Esta estructura tiene ciertas variaciones. Por ejemplo, se pueden ordenar los vértices dentro de cada lista, lo cual permite usar búsqueda binaria para comprobar la pertenencia de un vértice a ellas, pero aumenta la complejidad de construir la estructura y la de agregar vértices (porque hay que mantener el orden).

#### Listas de adyacencia

Se mantienen $n$ listas, donde cada lista $L_i$ contiene todos los vértices de $N(v_i)$. Esto permite realizar algunas operaciones más rápidamente, y la estructura se puede construir a partir de la lista de aristas en tiempo lineal.

#### Matriz de adyacencia

En este caso, se tiene una matriz $M \in \{0,1\}^{n \times n}$, donde cada posición está determinada por:

$$M_i j = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en caso contrario} \end{cases}$$

La matriz es simétrica para grafos, pero no necesariamente para digrafos.

La estructura permite comprobar si dos vértices son adyacentes en tiempo constante. Sin embargo, construirla a partir de una lista de adyacencia es una operación de complejidad cuadrática, y la estructura es muy rígida (para agregar un vértice se debe armar una nueva matriz). Además, la complejidad espacial es también$\mathcal{O}(|V|^2)$, lo cual es problemático para guardar grafos ralos[4].

#### Matriz de incidencia

Esta estructura es una matriz $I \in \{0,1\}^{m \times n}$ donde las filas representan los vértices y las columnas las aristas. Una posición $i, j$ tiene uno cuando la arista de la columna $j$ es incidente al vértice de la fila $i$.

#### Complejidades

### 2.1.9. Isomorfismo

Dos grafos $G = (V, E)$ y $G' = (V', E')$ son *isomorfos* cuando existe una función biyectiva $f : V \longrightarrow V'$ tal que:

$$\forall v, w \in V,\ (v,w) \in E \iff (f(v), f(w)) \in E'$$

A la función $f$ se la llama isomorfismo, y se denota $G \cong G'$ o (por abuso de notación) $G = G'$.

**Teorema.** *Si dos grafos $G \cong G'$ son isomorfos.*

- *Tienen el mismo número de vértices.*
- *Tiene el mismo número de aristas.*

[4] Un grafo *ralo* es uno con "pocas" aristas.

- $\forall 0 \leq k \leq n-1$, *tienen el mismo número de vértices de grado* $k$.
- *Tienen el mismo número de componentes conexas.*
- $\forall 0 \leq k \leq n-1$, *tienen el mismo número de caminos simples de longitud* $k$.

*Demostración.* □

### 2.1.10. Definiciones en digrafos

**Vecinos**

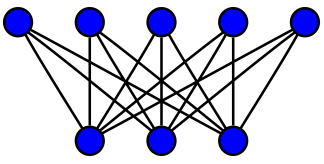
- Para un arco $e = (v, w) = v \to w$, se llama *cola* de $e$ a $v$ y *cabeza* de $e$ a $w$.
- El *grado de entrada* $d_-(v)$ es la cantidad de arcos que tienen a $v$ como cabeza.
- El *grado de salida* $d_+(v)$ es la cantidad de arcos que tienen a $v$ como cola.
- El *grafo subyacente* de $G$ es el grafo que resulta de ignorar las direcciones de sus arcos.

**Recorridos**

- Un *recorrido/camino orientado* en un digrafo es una sucesión de vértices que están conectados apropiadamente por arcos (sin repetidos en el caso del camino).
- Un *circuito/ciclo orientado* es un recorrido/camino orientado que empieza y termina en el mismo vértice.
- Un digrafo es *fuertemente conexo* si para todo par de vértices $v, u$ existen caminos orientados de $u$ a $v$ y de $v$ a $u$.

## 2.2. Grafos Bipartitos

Un grafo $G = (V, E)$ es *bipartito* cuando existe una *bipartición* de sus vértices $(V_1, V_2)$ tal que todas las aristas de $G$ tienen un extremo en $V_1$ y el otro en $V_2$. Por otro lado, $G$ es *bipartito completo* cuando todo vértice de $V_1$ es adyacente a todo vértice de $V_2$, y se denota $G = K_{|V_1|,|V_2|}$.



El grafo bipartito completo $K_{3,5}$.

**Teorema.** *Un grafo $G$ es bipartito $\iff$ no tiene ciclos de longitud impar.*

*Demostración.* Como un grafo es bipartito si y solo si cada una de sus componentes conexas es

bipartita, y un grafo no tiene ciclos impares si y solo si ninguna de sus componentes conexas tiene ciclos impares, alcanza con demostrar el teorema para grafos conexos.

$\Longrightarrow$ ) Sea $(V_1, V_2)$ la bipartición de $G$.

Si $G$ tiene algún ciclo $C = v_1 v_2 \cdots v_k v_1$, se puede asumir sin pérdida de generalidad que $v_1 \in V_1$. Luego, como $v_1 v_2 \in E$ (y $G$ es bipartito), $v_2 \in V_2$. En general, $v_{2i+1} \in V_1$ y $v_{2i} \in V_2$. Como $v_1 \in V_1$ y $v_k v_1 \in E$, se debe cumplir $v_k \in V_2$. Por ende $k = 2i$ así que $l(C)$ es par.

$\Longleftarrow$ ) Sea $u$ cualquier vértice de $V$. Se definen los siguientes conjuntos:

$$V_1 = \{v \in V \mid 2 \mid d(u, v)\} \cup \{u\}$$
$$V_2 = \{v \in V \mid 2 \nmid d(u, v)\}$$

$(V_1, V_2)$ definen una partición de $V$. Se puede demostrar que es una bipartición de $G$ por el absurdo.

Supongamos que no es una bipartición, entonces existen $v, w \in V_1$ (s.p.g.) tales que $vw \in E$. Si $v = u$, entonces $d(v, u) = 1$, que es absurdo porque $d(v, w)$ es par. Lo mismo vale para $w$, así que $v \neq u$ y $v \neq w$.

Sea $P$ un camino mínimo entre $v$ y $u$ y $Q$ uno entre $v$ y $w$. Como $u, w \in V_1$ $P$ y $Q$ tienen longitud par. Luego, sea $z$ el vértice común a $P$ y $Q$ tal que $P_{zv}$ y $Q_{zw}$ son disjuntos (ignorando $z$).

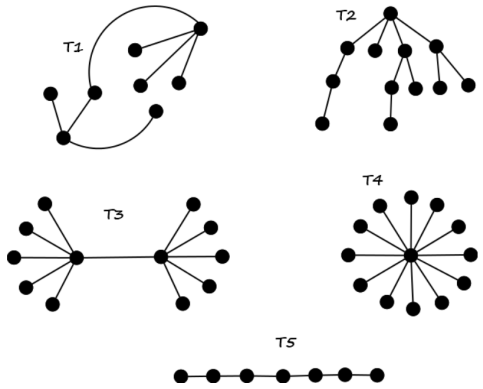
Se debe cumplir $d(u, z) = l(P_{uz}) = l(Q_{uz})$, porque del contrario $P$ y $Q$ no serían caminos mínimos. Esto implica que $l(P_{zv})$ y $l(Q_{zw})$ tienen la misma paridad, porque $l(P)$ y $l(Q)$ son ambos pares y la diferencia entre las longitudes totales y las de los subcaminos es la misma. Por ende, el ciclo $P_{zv}(v, w)Q_{wz}$ tiene longitud impar (**Absurdo**).

□

## 2.3. Árboles

### 2.3.1. Definición

Un *árbol* es un grafo conexo acíclico.




Ejemplos de grafos que son árboles.

Existen caracterizaciones alternativas:

**Teorema.** *Dado un grafo $G = (V, E)$, son equivalentes:*

1. *$G$ es un árbol (un grafo conexo acíclico).*
2. *$G$ es un grafo acíclico y $\forall e \notin X$, $G + e = (V, E \cup \{e\})$ tiene exáctamente un ciclo, y ese ciclo pasa por $e$.*
3. *Existe exactamente un camino simple entre todo par de vértices de $G$.*
4. *$G$ es conexo, pero si se quita cualquier arista de $G$, queda un grafo disconexo (toda arista es puente).*
5. *$G$ es un grafo conexo con $|E| = |V| - 1$*
6. *$G$ es un grafo acíclico con $|E| = |V| - 1$.*

#### Hojas

Una *hoja* en un árbol es un vértice de grado 1. Todo árbol *no trivial* (con al menos 2 vértices) tiene al menos 2 hojas.

#### Bosques

Un *bosque* es un grafo sin ciclos. Sus componentes conexas forman árboles, y se cumple $m = n - c$, donde $c$ es la cantidad de componentes conexas del bosque.

### 2.3.2. Árboles enraizados

Un *árbol enraizado* es un árbol con un vértice especial $r$ designado *raíz*. Luego, queda definido un árbol dirigido, donde los arcos van desde vértices más cercanos a la raíz hacia los más lejanos. En tal caso, está la siguiente terminología:

- Los vértices *internos* son aquellos que no son ni hojas ni raíz.

- El *nivel* de un vértice $v$ es la distancia de la raíz a ese vértice ($d(r, v)$).
- Para cada arco $v \to w$, $v$ es el *padre* de $w$, y $w$ es el *hijo* de $v$.
- La *altura* $h$ de un árbol enraizado es la distancia desde la raíz al vértice más lejano ($\max\{d(r, v) \mid v \in V\}$).
- Un árbol se dice *$m$-ario* si todos sus nodos internos tiene grado a lo sumo $m + 1$ y la raíz tiene grado a lo sumo $m$.
- Un árbol es *balanceado* cuando la diferencia entre el nivel de cada par de hojas es a lo sumo 1.

**Teorema.** *Dado un árbol enraizado $m$-ario con altura $h$ y $l$ hojas, $l \leq m^h$ ($\iff h \geq \lceil \log_m l \rceil$).*

### 2.3.3. Representación de árboles

Además de las representaciones de grafos anteriormente mencionadas, los árboles enraizados tienen una alternativa particular: debido a que todos los nodos tienen un único padre, un árbol puede ser definido por la correspondencia entre cada nodo y su antecesor. Esto se puede lograr usando solamente un arreglo "`prev`", en el que cada posición $i$ contiene al padre del nodo $v_i$. La única excepción es la raíz $r$ que, al no tener antecesor, se puede marcar utilizando un valor especial $\perp$, o como su propio padre ($d[r] = r$)[5].

### 2.3.4. Árbol generador

Un *árbol generador* (AG) de un grafo $G$ es un subgrafo generador que además es un árbol. En la práctica, los árboles generadores son utilizados cuando se busca conectar (con la cantidad mínima posible de conexiones) a $n$ puntos (ciudades, centrales eléctricas, servidores).

**Teorema.** *Dado un grafo conexo $G = (V, E)$.*

- *$G$ tiene (al menos) un árbol generador*
- *$G$ tiene un único árbol generador $\iff$ $G$ es un árbol.*
- *Sea $T = (V, E_T)$ un AG de $G$ y $e \in E - E_T$. Luego, para toda arista $f \neq e$ contenida en el único ciclo de $T + e$, $T + e - f = (V, E_T \cup \{e\} - \{f\})$ es un AG de $G$.*

## 2.4. Recorridos

Es común querer pasar por todos los vérices de un grafo una única vez. Existen distintos métodos de hacerlo de forma sistemática y ordenada, y en este caso nos vamos a enfocar en los 2 más comunes: *BFS* y *DFS*. En ambos casos, se mantiene una *frontera* con los vértices que se están por explorar, y cada vez que se pasa por uno de ellos sus vecinos son agregados a la misma. Además, se mantiene un conjunto de los vértices explorados (generalmente implementado con un *bitset*) para evitar su repetición. El esquema general es el siguiente:

[5] Esto se puede extender para guardar bosques: basta con tener varias raíces.

```
RECORRER(G, s)
1  Inicializar la frontera F = {s}.
2  while la frontera no esté vacía
3      Extraer un v de la frontera.
4      PROCESAR(v)
5      for each u ∈ N(v)
6          if u no fue visitado
7              Marcar a u como visitado.
8              Agregar u a la frontera.
```

### 2.4.1. BFS

El *Breadth-First Search* (BFS) es un algoritmo que, dado un grafo $G$ y un vértice inicial $s$, recorre todos los vértices nivel por nivel, es decir, los vértices más cercanos al inicial son visitados primero. Formalmente, si $\langle v_1, ..., v_n \rangle$ es la secuencia de vértices en el orden en que son recorridos, entonces se cumple:

$$d(s, v_i) \leq d(s, v_j) \; \forall \, 1 \leq i \leq j \leq n$$

Para lograr esto, BFS utiliza como frontera una cola, donde los elementos son procesados en orden de llegada (FIFO). El algoritmo se puede implementar iterativamente de la siguiente manera:

```
BFS(G = (V, E), s)
 1  visitados = ∅
 2  Inicializar árbol T con raíz en s.
 3  Inicializar arreglo de distancias d.
 4  Inicializar cola Q.
 5  d[s] = 0
 6  ENCOLAR(Q, s)
 7  while ¬VACÍO?(Q)
 8      v = DESENCOLAR(Q)
 9      PROCESAR(v)
10      for each u ∈ N(v)
11          if u ∉ visitados
12              visitados = visitados ∪ {u}
13              d[u] = d[v] + 1
14              Agregar u a T como hijo de v.
15              ENCOLAR(Q, u)
16  return (T, d)
```

El algoritmo devuelve 2 valores: un árbol generador $T$, que se denomina *árbol BFS* y contiene las aristas transitadas por el recorrido, junto con la función $d : V \longrightarrow \mathbb{N}_0$, que indica las distancias de $s$ a cada vértice del grafo.

Si el grafo se representa utilizando listas de adyacencia, el vecindario $N(v)$ se puede recorrer fácilmente. El algoritmo pasa una sola vez por cada vértice y, asumiendo que tanto *visitados*, $T$ y $d$ se representan a través arreglos, realiza una operación de tiempo constante en cada uno de

sus vecinos. Por ende, la complejidad de este algoritmo es:

$$\mathcal{O}(|V| + \sum_{v \in V} d(v)) = \mathcal{O}(|V| + 2|E|) = \mathcal{O}(|V| + |E|)$$

**Árboles geodésicos**

Un árbol generador $T$ de un grafo $G$ se llama $v$-geodésico cuando $d_G(v, w) = d_T(v, w)\ \forall w \in V$.

**Teorema.** *Si se corre el algoritmo BFS en un grafo $G$ empezando en un vértice $s$, el árbol generador resultante $T$ es $s$-geodésico, y las distancias que devuelve son las mínimas entre $s$ y cada vértice del árbol.*

*Demostración.* □

### 2.4.2. DFS

La estrategia que sigue el *Depth-First Search* (DFS) es buscar "en profundidad" siempre que sea posible. Esto significa que al llegar a $v$, se recorren todos los vértices no visitados alcanzables desde este. Este procedimiento se realiza hasta que todos los nodos hayan sido explorados.

El algoritmo de DFS se puede implementar recursivamente de la siguiente manera (*visitados*, $T$, *principio*, *fin* y *contador* son variables globales):

```
DFS(G, s)
1  visitados = ∅
2  contador = 0
3  Inicializar arreglos principio y fin.
4  Inicializar T como árbol vacío.
5  VISITAR-DFS(G, s)
```

```
VISITAR-DFS(G, v)
1  contador = contador + 1
2  principio[v] = contador
3  visitados = visitados ∪ v
4  for each u ∈ N(v)
5       if u ∉ visitados
6       Agregar u como hijos de v en el árbol T.
7       VISITAR-DFS(G, u)
8  contador = contador + 1
9  fin[v] = contador
```

El tiempo de ejecución del algoritmo, al igual que BFS, es lineal[6]: hay una llamada por cada nodo, y cada llamada tiene un tiempo de ejecución proporcional al grado del nodo, así que la complejidad es $\mathcal{O}(|V| + |E|)$.

[6]La linealidad de estas complejidades se refiere a que, como los grafos se pasan como listas de adyacencia, el tamaño de la entrada es $|E|$. Si se considerara la cantidad de vértices, una complejidad de$\mathcal{O}(|E|)$ sería cuadrática, ya que $|E| \in \mathcal{O}(|V|^2)$.

Al terminar, DFS no solo devuelve el árbol generado $T$, sino que también un par de arreglos *principio* y *fin*. El primero guarda el orden en el que se empieza a explorar el subárbol de cada nodo (también llamado *pre-order*), mientras que el segundo guarda el orden en el que se termina dicha exploración (también llamado *post-order*). Estos valores son muy útiles para analizar la estructura del árbol.

**Teorema.** *Dado grafo $G$ y un árbol DFS $T_G$ y un par de vértices $v$, $u$, se cumple alguna de las siguientes:*

1. $[principio[v], fin[v]] \cap [principio[u], fin[u]] = \emptyset$*, y en tal caso $v$ y $u$ están en ramas distintas (ninguno es descendiente del otro).*
2. $[principio[v], fin[v]] \subseteq [principio[u], fin[u]]$*, y entonces el vértice $v$ es descendiente de $u$.*
3. $[principio[v], fin[v]] \supseteq [principio[u], fin[u]]$*, y entonces el vértice $u$ es descendiente de $v$.*

### Tipos de aristas

Dado un grafo $G$ y un árbol DFS $T$, las aristas de $G$ se pueden dividir en las siguientes categorías:

- *Tree edges*: son aquellas que están en $E_T$.
- *Back edges*: son aquellas que no están en $E_T$, y que conectan a un nodo con un antecesor en $T$.
- *Forward edges*: son aquellas que no están en $E_T$, y que conectan a un nodo con un descendiente en $T$.
- *Cross edges*: son aquellas que no están en $E_T$, y que conectan a nodos de distintas ramas del árbol.

La categoría de cualquier arista puede ser identificada en tiempo constante utilizando los resultados del DFS: la pertenencia a $E_T$ se puede chequear revisando los padres de los vértices incidentes a la arista en el árbol, mientras que las otras condiciones se pueden verificar a través de los arreglos *principio* y *fin*.

**Teorema.** *Dado un grafo no dirigido $G$, todas las aristas de cualquier árbol DFS son Tree Edges o back edges.*

### Detección de ciclos

**Teorema.** *Un grafo $G$ es acíclico $\iff$ ningún árbol DFS de $G$ tiene back edges.*

*Demostración.*

$\implies$ ) Se demuestra por el contrarrecíproco: si $T$ es un árbol DFS con una backedge $e = u \to v$, se puede tomar el ciclo $C = P + e$, donde $P$ es el camino que une a $v$ y $u$ en $T$ (debe existir, ya que $v$ es antecesor de $u$ por ser $e$ back edge).

$\impliedby$ ) También se demuestra el contrarrecíproco: supongamos que $C$ es un ciclo de $G$.

Dado un recorrido DFS cualquiera, sea $v$ el primer vértice de $C$ que se encuentra y $u$ el vértice "anterior"[7] en el ciclo. Luego, como $u$ es alcanzable desde $v$, será uno de sus descendientes, así que la arista $u \to v$ es una back edge.

□

El algoritmo DFS puede ser utilizado para encontrar ciclos de un grafo, ya que todas las back edges forman parte de al menos un ciclo. Existen varias opciones:

- En el caso de los grafos no dirigidos, basta con adaptar DFS para devolver el ciclo cuando un vértice $u$ adyacente al actual $v$ ya fue visitado. En ese caso, el ciclo es $C = uT_{uv}vu$, donde $T_{uv}$ es el camino entre $u$ y $v$ en el árbol. Esto funciona porque cuando se visita un vértice por segunda en un grafo no dirigido vez siempre se hace a través de una Back Edge.
- Otra opción que también sirve para grafos dirigidos es pasar por todas las aristas que no están en el árbol hasta identificar una Back Edge (usando las condiciones establecidas anteriormente)

### Versión iterativa

La versión iterativa de DFS sigue el esquema general establecido previamente. En este caso, la frontera se implementa utilizando un stack (FILO), lo cual garantiza que un vértice se deja de explorar solo cuando todos los vértices alcanzables desde ese fueron visitados.

### Bosques

Si el grafo recorrido $G$ no es conexo, se puede formar un bosque donde cada componente conexa es un árbol DFS. Esto se logra corriendo DFS iterativamente, cada vez empezando en uno de los vértices que aún no fue recorrido por las iteraciones anteriores. El algoritmo es el siguiente, donde VISITAR-DFS es el procedimiento definido anteriormente:

```
DFS(G)
1  visitados = ∅
2  contador = 0
3  Inicializar arreglos principio y fin.
4  Inicializar T como árbol vacío.
5  for each v ∈ V
6      if v ∉ visitados
7          VISITAR-DFS(G, s)
```

[7] En grafos no dirigidos hay dos: se puede tomar sin pérdida de generalidad.

## 2.5. Orden Topológico

### 2.5.1. Definición

**Problema:**
Dado un digrafo acíclico (un "DAG") $D = (V, E)$, encontrar un ordenamiento $\langle v_1, v_2, ..., v_n \rangle$ de sus vértices de forma tal que, para todo $v \in V$ los vértices alcanzables desde $v$ aparezcan después en el orden. Formalmente,

$$d(v_i, v_j) < \infty \implies \forall\ 1 \leq i \leq j \leq n$$

Esto se denomina un *orden/ordenamiento topológico*, y tiene múltiples aplicaciones prácticas, que surgen cada vez que se debe ordenar un conjunto de cosas con precedencias entre sí. Notar que solo es posible para digrafos acíclicos: en un ciclo, todos los vértices son alcanzables desde los demás, así que ninguno podría ir antes de los demás.

### 2.5.2. Algoritmo

Habiendo implementado y analizado DFS, el algoritmo para encontrar un ordenamiento topológico es simple, gracias al siguiente teorema:

**Teorema.** *Para un DAG $D$, el ordenamiento inverso al post-order de cualquier DFS es un orden topológico.*

*Demostración.* Como el valor de $fin[v]$ se asigna después de haber explorado por todos los vértices $u$ alcanzables desde $v$, se cumple que $fin[u] < fin[v]$. Por ende, si se ordenan los vértices por valor $fin$ decreciente, los vértices que se pueden alcanzar desde otros aparecerán después. □

Para implementar este algoritmo, no es necesario ordenar directamente los vértices a través de $fin$ (lo tendría complejidad$\mathcal{O}(|V| \log |V|)$), sino que basta con modificar DFS: después de terminar de explorar el subárbol de cada vértice, se lo agrega al principio de una secuencia. Esto produce un ordenamiento topológico de $D$.

## 2.6. Algoritmo de Kosaraju

El algoritmo de Kosaraju es un algoritmo lineal que encuentra las componentes fuertemente conexas de un digrafo $G$. Se basa en hacer 2 recorridos DFS: el primero se utiliza para obtener un post-order de los vértices del grafo, mientras que el segundo lo recorre de manera inversa para armar los componentes. Este segundo DFS se hace sobre $G^T$, el digrafo cuyas aristas tienen el sentido opuesto al de las de $G$. El algoritmo es el siguiente:

KOSARAJU($G$)
1 Llamar a DFS($G$) para obtener un post-order inverso.
2 Computar $G^T$
3 Llamar a DFS($G^T$), pero en el loop principal explorar los vértices en el post-order inverso. Cada vez que se visita un vértice nuevo, agregarlo a la CFC de la raíz actual.

El loop principal de DFS se refiere al de la versión que arma bosques iterando por cada vértice sin explorar.

Para ver por qué este algoritmo funciona, primero se debe observar el siguiente lema:

**Lema.** *Dadas CFCs*[8] $C$ *y* $C'$ *distintas en* $G = (V, E)$*, ningún vértice de* $C$ *es alcanzable desde* $C'$*, o viceversa.*

*Demostración.* Esto se puede demostar por el absurdo: si existen caminos $u \rightsquigarrow v$ y $u' \rightsquigarrow v'$ tales que $u, v \in C$ y $u', v' \in C'$, entonces cualquier vértice de $C$ puede llegar a cualquiera de $C'$ a pasando por $u \rightsquigarrow v$, y cualquiera de $C'$ puede a su vez llegar a uno de $C$ a través de $u' \rightsquigarrow v'$. Esto implica que todos están en la misma CFC (**Absurdo**).

□

Esto permite visualizar las componentes conexas de un digrafo como un DAG: $G^{CFC}$ se puede definir como el grafo donde cada vértice es una versión compactada de un CFC de $G$. No puede haber ciclos porque violarían el lema anterior.

Luego, el post-order se puede emplear gracias a que:

**Lema.** *Si* $C$ *y* $C'$ *son CFCs distintas de* $G$ *y existe un arco* $u \to v$ *tal que* $u \in C$ *y* $v \in C'$*, entonces* $\mathit{fin}[C] > \mathit{fin}[C']$[9].

*Demostración.* Se puede separar en dos casos dependiendo de qué CFC se visita antes:

- Si algún vértice $w$ de $C$ se explora primero, entonces esta exploración va a alcanzar el vértice $u$ (porque están en la misma CFC), y por ende visitar todos los vértices de $C'$ antes de $\mathit{fin}[w]$, y por ende $\mathit{fin}[C] > \mathit{fin}[C']$.
- Si $C'$ se visita primero, la exploración va a concluir sin pasar por ningún vértice de $C$ ya que, gracias al lema anterior, no existen caminos entre vértices de $C'$ y los de $C$. Esto implica que $\mathit{fin}[C] > \mathit{fin}[C']$.

□

Esto tiene como corolario que, en el grafo $G^T$, un par de CFCs $C$ y $C'$ con un arco $u \to v$ tal que $u \in C$ y $v \in C'$ cumplen $\mathit{fin}[C] < \mathit{fin}[C']$. Eso se debe a que $G^T$ tiene los mismos CFCs que $G$, y $u \to v \in E^T \implies v \to u \in E$.

Finalmente, se puede demostrar la correctitud del algoritmo:

**Teorema.** *El algoritmo de Kosaraju devuelve las componentes conexas de un digrafo* $G$*.*

*Demostración.* Como el segundo DFS empieza por un vértice $v$ de la CFC con $\mathit{fin}[C]$ máximo, ninguno de sus vértices contiene una arista hacia otra CFC $C'$ (porque en ese caso $\mathit{fin}[C] < \mathit{fin}[C']$). Además, como todos los vértices de $C$ son alcanzables desde $v$, la CFC se construye

[8] Componentes Fuertemente Conexas

[9] El *fin* de una CFC se define como $\max\{\mathit{fin}[v] \mid v \in C\}$.

correctamente (se visitan todos sus vértices, y ningún otro). Luego, para las demás componentes, sucederá algo similar: las únicas aristas que las conectan con otras CFCs serán a aquellas con un mayor valor de *fin*, y por ende ya habrán sido exploradas. Esto implica que el algoritmo arma cada componente de forma correcta.

□

# Capítulo 3

# Árbol Generador Mínimo

## 3.1. Definición

### Grafos pesados

Un *grafo pesado* es un grafo $G = (V, E)$ junto con una función $w : E \longrightarrow \mathbb{N}$, donde $w(e)$ es el peso de la arista $e$. El peso suele usarse para modelar diversas magnitudes, como el costo de una arista, o tiempo que se tarda en pasar por ella.

### AGM

Un *árbol generador mínimo* (AGM) es un árbol generador cuyo peso total es menor al de todos los demás AGs, donde peso total se define como la suma de los pesos de sus aristas. Formalmente, el problema es el siguiente:

**Problema:**
Dado un grafo $G = (V, E)$ y una función de peso $w : E \longrightarrow \mathbb{N}$, encontrar un AG $T = (V, E_T)$ que cumpla:

$$w(T) = \sum_{e \in E_T} w(e) \leq \sum_{e \in E_{T'}} w(e) = w(T')$$

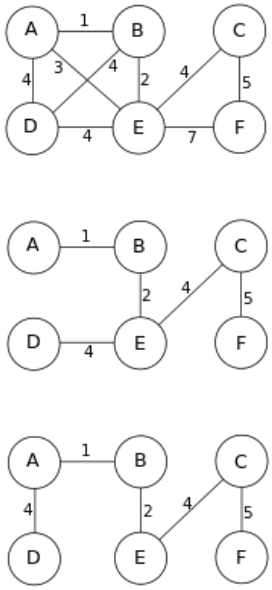
Para cualquier árbol generador $T' = (V, E_{T'})$ en $G$.

También se puede definir el árbol generador máximo, que maximiza el peso total de $T$, pero este problema se puede reducir al anterior: si se toman los pesos $w'(e) = -w(e)$, el peso de cualquier AG $T$ será $w'(T) = \sum_{e \in E_T} w'(e) = \sum_{e \in E_T} -w(e) = -w(T)$, y por ende:

$$\begin{aligned} w'(AGM) \leq w'(T) &\iff -w'(AGM) \geq -w'(T) \\ &\iff w(AGM) \geq w(T) \end{aligned}$$

### Propiedades

El AGM de un grafo no necesariamente es único: pueden varios grafos con el mismo peso, que es menor al de todos los demás.




Un grafo y 2 de sus AGMs, ambos con peso total 16.

### Algoritmos

En las siguientes secciones analizaremos 2 algoritmos para encontrar un AGM: el algoritmo de Prim y el de Kruskal. Ambos son golosos: siguen estrategias que toman en cada paso la mejor decisión a corto plazo, y en este caso devuelven soluciones óptimas. Logran esto armando progresivamente un bosque (en el caso de Prim un árbol) en un ciclo que mantiene el invariante de que sus aristas forman un subconjunto de las de algún AGM.

ARBOL-GENERADOR-MINIMO$(G, w)$
1 Inicializar grafo $T = (V, \emptyset)$
2 **while** $T$ no es un AG (es disconexo):
3 Encontrar una arista segura $e$.
4 Agregar $e$ a $E_T$.
5 **return** $T$

En este contexto una arista es *segura* cuando se puede agregar al bosque $T$ y mantener el invariante (es un subconjunto de algún AGM). Encontrar una es donde radica la dificultad del problema, y es en donde difieren los algoritmos.

## 3.2. Algoritmo de Prim

### 3.2.1. Introducción

El *algoritmo de Prim* mantiene un árbol $T = (V_T, E_T)$, y en cada iteración agrega un nuevo vértice de $V - V_T$ al mismo (junto con una arista a $E_T$), hasta que $T$ pasa por todos los vértices de $G$ (y se vuelve un AG). La arista seleccionada en cada paso es la de mínimo peso entre todas las $(u, v) \in V_T \times (V - V_T)$ (las que conectan vértices dentro del árbol con los de afuera).

Se puede implementar de la siguiente manera:

```
PRIM(G, w)
1  V_T = {s} (cualquier vértice)
2  E_T = ∅
3  for i = 1 to n − 1
4        e = arg mín {w(e) | e = uv ∈ E, u ∈ V_T, v ∈ V − V_T}
5        E_T = E_T ∪ e
6        V_T = V_T ∪ v
7  return T = (V_T, E_T)
```

El algoritmo asume que el grafo es conexo. Si esto no se cumple, se puede modificar el algoritmo levemente[1] para que devuelva el AGM de la componente conexa que contiene al vértice inicial.

### 3.2.2. Correctitud

**Teorema.** *El árbol que devuelve el algoritmo de Prim es un AGM.*

*Demostración.* Se demuestra por inducción. La hipótesis inductiva será la siguiente: $\forall\, 0 \leq k \leq n-1, T_k = (V_k, E_k)$, el grafo mantenido por Prim en la $k$-ésima iteración del ciclo, es un árbol y es subgrafo de algún AGM de $G$.

**Caso Base**: En el paso $k = 0$, todavía no se entró al ciclo, así que $T_0 = (\{s\}, \emptyset)$, que es un árbol (no tiene ciclos) y es subgrafo de cualquier AG (y por lo tanto de cualquier AGM).

**Paso Inductivo**: Sea $e = vw$ agregada en la iteración. Luego, $T_k = (V_k, E_k) = (V_{k-1} \cup \{w\}, E_{k-1} \cup \{e\})$. Por hipótesis inductiva, se sabe que $T_{k-1}$ es subgrafo de algún árbol generador $T$. Existen dos posibilidades:

- $e \in E_T$, en cuyo caso $T_k$ es también subgrafo de $T$.
- $e \notin E_T$. Aún así, sabemos que $V_{k-1}$ forma una componente conexa en $T$, porque $T_{k-1}$ es un subgrafo suyo y es conexo (es un árbol). Entonces, debe haber alguna arista $e' = v'w' \in E_T$ que conecta $V_{k-1}$ y $V - V_{k-1}$. Si se saca esa arista de $T$, las componentes se vuelven disconexas, pero se pueden volver a conectar usando la arista $e$.

  Por ende, $T' = T + e - e' = (V, (E_T \cup \{e\}) - \{e'\})$ es un árbol (y es generador, ya que

[1] El único problema que tiene esa implementación es que asume que el conjunto del que se toma el arg mín no está vacío. Si esto se chequea en cada iteración, se puede terminar la ejecución cuando no quedan vértices por agregar.

contiene a todos los vértices). Además, como $e'$ era una de las aristas que se podían elegir al principio de la iteración (conecta a $V_{k-1}$ con $V - V_{k-1}$), se debe cumplir que $w(e) \leq w(e')$, así que:

$$w(T') = w(T) + \underbrace{w(e) - w(e')}_{\leq 0} \leq w(T)$$

Por lo tanto, el peso total de $T'$ es menor o igual al de un AGM, así que este también es AGM. Como $T_k$ es subgrafo de $T'$ ($E_k = E_{k-1} \cup \{e\} \subseteq E_T \cup \{e\}, e' \notin E_k$), se cumple la propiedad.

Dado que la propiedad vale al terminar cualquier iteración, también se cumple en $k = n-1$, donde $T_{k-1}$ tiene $n$ vértices, y por lo tanto es un árbol generador (y mínimo).

□

### 3.2.3. Complejidad

Para implementar el algoritmo de Prim, es necesario determinar un método para encontrar la arista de menor peso entre $V_T$ y $V - V_T$. Esto se puede lograr utilizando una cola de prioridad, pero la forma en la que esta se represente puede variar:

- Un arreglo de $|V|$ posiciones: En este caso, el peso de la arista que conecta a cada vértice con el árbol se guarda en una posición del arreglo (los vértices que ya se agregaron se marcan con un valor especial). Para determinar el siguiente vértice a agregar, se toma el mínimo del arreglo, y una vez que se agrega se actualizan las posiciones correspondientes (en caso de que uno de sus vecinos se pueda alcanzar con peso menor al anterior). Ambas operaciones son $\mathcal{O}(|V|)$, y como se realizan $|V| - 1$ operaciones, la complejidad total $\mathcal{O}(|V|^2)$.

- Un heap: En este caso, se mantiene un mín-heap de los nodos que no fueron explorados, donde la clave es el peso mínimo entre las aristas que lo conectan al árbol. En cada paso se desencola un vértice, y se actualizan las claves de sus vecinos[2]. Esto implica que se realizan $1 + d(v)$ operaciones logarítmicas en cada iteración, lo cual resulta en una complejidad total de $\mathcal{O}((|V| + |E|) \log |V|)$. Como se asume que $G$ es conexo, $|E| \geq |V| - 1$, y por ende $\mathcal{O}((|V| + |E|) \log |V|) \subseteq \mathcal{O}(|E| \log |V|)$.

En el caso de los grafos *densos*, que se podrían definir como aquellos en los que $|E| \in \Omega(|V|^2)$, la segunda alternativa tendría complejidad $\mathcal{O}(|V|^2 \log |V|)$, mientras que la primera solo $\mathcal{O}(|V|^2)$. Por otro lado, si el grafo es *ralo*, que sería cuando $|E| \in \mathcal{O}(|V|)$, la segunda sería tan solo $\mathcal{O}(|V| \log |V|)$, mejor que $\mathcal{O}(|V^2|)$.

Hay una tercera opción para representar la cola de prioridad: a través de Fibonacci heaps. Estos permiten que el algoritmo corra en $\mathcal{O}(|E| + |V| \log |V|)$ (asintóticamente mejor que las dos anteriores), pero es más complejo de implementar que las otras opciones.

[2] Para lograr esto en tiempo logarítmico, es preferible usar un árbol balanceado (como los AVLs) antes que un heap.

## 3.3. Algoritmo de Kruskal

### 3.3.1. Disjoint-Set/Union-Find

Antes de ver el algoritmo en sí, es necesario una de las estructuras que usa: el *Disjoint-Set* o *Union-Find*. Esta mantiene una partición $P_1, ..., P_n$ de un conjunto $C$, asignándole a todos los elementos de cada $P_i$ un mismo representante $r_i$. Permite realizar las siguientes operaciones de forma eficiente:

- MAKE-SET($x$): crea un nuevo conjunto $\{x\}$ dentro de la partición. Como precondición, $x$ no puede estar en ninguno de los conjuntos anteriores.
- UNION($x$, $y$): Reemplaza los conjuntos que contenían a $x$ e $y$, $S_x, S_y$ por su unión $S_x \cup S_y$.
- FIND($x$): Devuelve el representante $r_x$ de $x$ dentro de la partición. Esto puede ser utilizado para comprobar si dos elementos pertenecen al mismo conjunto.

#### Implementación

Internamente, la estructura se implementa como un bosque, donde cada árbol se corresponde con el conjunto de la partición que contiene a sus vértices, y la raíz es el representante. Implementar la operación UNION es simple: solo es necesario agregar a la raíz de uno de los conjuntos como hijo de la raíz del otro. FIND también resulta trivial: solo es necesario recorrer los antecesores de cada nodo hasta llegar a la raíz.

Esto alcanza para implementar a la estructura, pero la complejidad no es ideal: los árboles podrían ser degenerados (una línea donde cada nodo tiene 1 solo hijo), lo cual haría que FIND sea $\mathcal{O}(n)$. Para evitar que suceda esto, se emplean 2 heurísticas:

- **Union by rank**[3]: Esto implica mantener la altura de cada árbol guardado, lo cual en este caso no es costoso: la única forma en la que puede cambiar es cuando se agrega otro árbol $T$ a la raíz como hijo, y en tal caso la altura pasa a ser el mínimo entre la anterior y $h(T)+1$. Esto se utiliza a la hora de decidir qué raíz utilizar para el nuevo árbol de UNION: se elige la que resulte en menor altura (queda como raíz el representante de mayor rango).
- **Path compression**: Esta optimización se basa en el hecho de que la estructura interna del árbol no es rígida: solo se debe cumplir que la raíz sea el representante de todos los nodos. Por ende, cada vez que se realiza un FIND, todos los vértices transitados se pueden colocar como hijos directos del árbol.

Los algoritmos se pueden implementar de la siguiente manera, donde *prev* es el arreglo que guarda los antecesores de cada nodo (representa al bosque) y *rank*[$v$] guarda la altura del árbol con raíz en $v$:

```
MAKE-SET(x)
1  prev[x] = x
2  rank[x] = 0
```

[3]Es equivalente usar la alternativa **union by size**, que usa la cantidad de nodos en el árbol en lugar de la altura.

```
UNION(x, y)
1  r_x = FIND(x), r_y = FIND(y)
2  if rank[r_x] > rank[r_y]
3        prev[y] = x
4  else
5        prev[x] = y
6        if rank[x] == rank[y]
7              rank[y] = rank[y] + 1
```

```
FIND(x)
1  if x ≠ prev[x]
2        prev[x] = FIND(prev[x])
3  return prev[x]
```

Haciendo uso de ambas heurísticas, la complejidad de peor caso amortizada de la operación FIND (y por ende también la de UNION) baja a $\mathcal{O}(\alpha(n))$, donde $\alpha$ es la función de Ackermann inversa[4].

### 3.3.2. Algoritmo

Teniendo esta estructura, se puede definir el *algoritmo de Kruskal*: se construye un bosque, recorriendo todas las aristas en orden de peso creciente, y mientras tanto se mantiene un Union-Find con las componentes conexas del bosque. Cada vez que se visita una arista, se chequea si conecta dos vértices de componentes conexas distintas. Si es así, se corre UNION sobre los conjuntos de las componentes, y la arista es agregada al árbol. Esto se hace hasta que queda una sola componente, y queda formado un árbol generador (que además es mínimo).

```
KRUSKAL(G, w)
1  T = (V, E_T = ∅)
2  for each v ∈ V
3        MAKE-SET(v)
4  for each uv ∈ E_T, en orden de w creciente
5        if FIND(u) ≠ FIND(v)
6              E_T = E_T ∪ uv
7              UNION(u, v)
8        return T
```

### 3.3.3. Correctitud

La correctitud de este algoritmo se puede demostrar de forma similar al anterior:

**Teorema.** *El árbol que devuelve el algoritmo de Kruskal es un AGM.*

*Demostración.* Se demuestra por inducción con la siguiente hipótesis inductiva: $\forall\, 0 \leq k \leq n-1, T_k = (V_k, E_k)$, el grafo mantenido por Kruskal después de agregar la $k$-ésima arista, es un bosque y subgrafo de algún AGM de $G$.

[4] A efectos prácticos, $\alpha(n) \leq 5$

**Caso Base**: En $k = 0$, el grafo es $T_0 = (V, \emptyset)$, que es subgrafo de cualquier AG, y no tiene ciclos (así que es un bosque).

**Paso Inductivo**: Sea $uv \in E$ la $k$-ésima arista agregada al grafo. Por hipótesis inductiva, sabemos que $T_{k-1}$ es un bosque, y como las aristas deben conectar componentes distintas, la nueva arista no forma parte de ningún ciclo, así que $T_k$ sigue siendo un bosque.

Por otro lado, la HI también implica que $T_{k-1}$ es subgrafo de algún AGM $T$. Como los árboles son conexos, debe haber algún camino de $T$ entre los vértices $u$ y $v$. Si este camino es solo la arista $uv$, $T_k$ también es un subgrafo del árbol. Por otro lado, si el camino es distinto, seguro contiene una arista $u'v'$ que conectaría el componente de $u$ en $T_{k-1}$ con algún otro (porque $v$ no es alcanzable desde $u$ en $T_{k-1}$). Eso implica que $w(uv) \leq w(u'v')$, ya que $uv$ es la arista de menor peso que cumple esa propiedad.

Si se toma $T' = (V, (E \cup \{uv\} - \{u'v'\}))$, $T'$ es un árbol porque agregar $uv$ genera un único ciclo que contiene a $u'v'$, y sacar esta arista lo rompe. Por otro lado, su peso es:

$$w(T') = w(T) + \underbrace{w(uv) - w(u'v')}_{\leq 0} \leq w(T)$$

Por ende, $T'$ es un AGM y, como $T_k$ es un subgrafo de $T'$, se cumple la propiedad.

□

### 3.3.4. Complejidad

El algoritmo de Kruskal tiene dos pasos generales: ordenar las aristas, y recorrerlas hasta armar el árbol. El ordenamiento es una operación $\mathcal{O}(|E| \log |E|)$, mientras que el ciclo realiza a lo sumo $|E|$ iteraciones, y en cada se hace una llamada a UNION, que tiene complejidad $\mathcal{O}(\alpha|V|)$. Aprovechando que $|E| \in \mathcal{O}(|V|^2)$, se tiene $\mathcal{O}(|E| \log |E|) \subseteq \mathcal{O}(|E| \log (|V|^2)) = \mathcal{O}(|E| \log |V|)$. La complejidad final entonces es:

$$\mathcal{O}(|E| \log |V| + |E|\alpha(|V|)) = \mathcal{O}(|E| \log |V|)$$

## 3.4. Camino Minimáx

### 3.4.1. Definición

Dado un grafo pesado $G = (V, E)$ con $w : E \longrightarrow \mathbb{N}$, el *camino minimáx* entre un par de vértices $u, v \in V$ es aquel que minimiza el peso de la arista más pesada del camino. Formalmente, es el $P_m$ que cumple:

$$P_m = \arg\min \{\max \{w(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Análogamente, el camino maximín es aquél que maximiza el peso de la arista menos pesada del camino, es decir, el $P_M$ tal que:

$$P_M = \arg\max \{\min \{w(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Esta noción tiene varias aplicaciones: en general se la conoce como "ancho de banda". Podría utilizarse para modelar la cantidad de datos que se pueden transmitir entre dos terminales pasando por una red con ciertas capacidades[5], o la cantidad de peso que puede pasar por una red de puentes con límites dados.

### 3.4.2. Solución

El método para encontrar un camino minimax entre dos vértices es simple:

**Teorema.** *Dado un grafo pesado $G = (V, E)$ con $w : E \longrightarrow \mathbb{N}$ y un AGM $T$, para cualquier par de vértices $u, v \in V$, el camino que conecta a $u$ con $v$ en $T$ es minimax.*

*Demostración.* Supongamos que existe un camino minimax $P$ que pasa por aristas que no están en el árbol $T$. Sea $uv \in E \cap P - E_T$, y $T_{uv}$ el camino que conecta sus vértices en el árbol. Luego, si se toma $e' = \arg\max\{w(e) \mid e \in T_{uv}\}$, se puede ver que $w(uv) \geq w(e')$, ya que si no $e'$ podría ser reemplazada por $uv$, formando un AG $T \cup \{uv\} - \{e'\}$ con un peso estrictamente menor, lo cual es absurdo porque $T$ es AGM.

Por ende, cualquier arista fuera del árbol puede ser reemplazada por un camino que pasa por el árbol y tiene aristas de menor o igual peso (así que el peso máximo del camino se mantiene igual). Entonces, queda demostrado que siempre existe un camino minimáx que pasa por el árbol.

□

Esto significa que, para encontrar el camino minimáx entre cualquier par de vértices, se puede calcular el AGM del grafo, y tomar el camino que los une dentro de este. Para el camino maximín, se puede tomar el árbol generador máximo (la demostración es análoga).

[5] Con la restricción de que todos los datos pasan por un mismo camino; si se pueden "dividir", es un problema de flujo.

# Capítulo 4

# Camino Mínimo

## 4.1. Definición

Dado un digrafo[1] pesado $G = (V, E, w)$, el peso de un camino $P$ entre sus vértices se define como la suma de los pesos de sus aristas:

$$w(P) = \sum_{e \in P} w(e)$$

Entonces, dados dos vértices $u$ y $v$, existe un camino de peso mínimo, es decir uno con el peso:

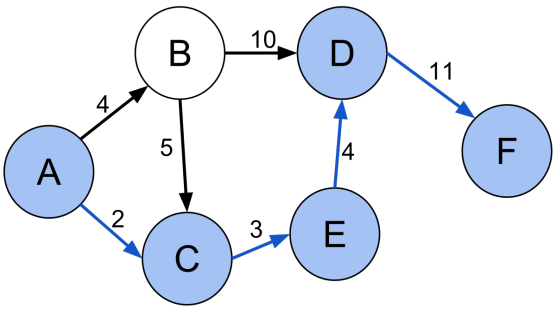
$$\delta(v, w) = \text{mín}\,\{w(P) \mid P \text{ es un camino entre } v \text{ y } w\}$$

La función $\delta(v, w)$ sigue reglas similares a $d(v, w)$, es decir:

- $\delta(v, v) = 0 \ \forall v \in V$
- $\delta(u, v) = \infty \iff u$ no es alcanzable desde $v$

$P$ es un *camino mínimo* cuando $w(P) = \delta(v, w)$, y no necesariamente es único.

[1] Todos los algoritmos de camino mínimo que analizamos pueden ser fácilmente adaptados para operar en grafos no dirigidos.




El camino mínimo entre el par de vértices $A$ y $F$.

Este mínimo existe porque el conjunto de caminos es finito: a lo sumo hay $n!$, uno por cada permutación de los vértices de $G$. Esto no necesariamente vale en el caso de recorridos, que pueden tener vértices repetidos.

Un problema análogo es el de *camino máximo*, que es el camino de peso máximo entre un par de vértices.

### 4.1.1. Existencia

A pesar de que un **camino** mínimo entre $u$ y $v$ siempre existen (asumiendo que están conectados), no es el caso para los recorridos: si un grafo tiene algún *ciclo de peso negativo* (o simplemente *ciclo negativo*) alcanzable desde $u$ y $v$, este puede ser transitado múltiples veces para obtener un peso tan bajo como se desee.

### 4.1.2. Camino mínimo elemental

El problema de esta índole más simple de definir (y más general) es el de *camino mínimo elemental*:

> **Problema:**
> Dado un (di)grafo $G = (V, E, w)$ y un par de vértices $v, u \in V$, encontrar el camino (sin vértices repetidos) de peso mínimo.

Como establecimos antes, este problema esta bien definido para cualquier grafo y cualquier par de vértices. El problema de *camino máximo elemental*, donde se busca el camino de peso máximo, se puede reducir fácilmente a este: basta con tomar una función de peso $w'(e) = -w(e)$[2].

Para el caso general, este problema es NP-hard, así que se suelen estudiar casos restringidos. La restricción más importante para nuestros algoritmos es que el grafo no tenga ciclos negativos. En ese caso, encontrar un camino mínimo es equivalente a encontrar un recorrido mínimo.

[2] A pesar de que los problemas son equivalentes, las aplicaciones de camino mínimo suelen ser en casos donde los pesos son no negativos (y por ende el grafo no tiene ciclos negativos). En tal caso, encontrar un camino máximo se vuelve más difícil, ya que negar los pesos hace que casi cualquier ciclo sea negativo. Es por esto que camino máximo se considera "más difícil".

**Teorema.** *Si un (di)grafo $G = (V, E, w)$ no tiene ningún ciclo negativo, existe un camino entre cualquier par de vértices $v, u \in V$ con peso total menor al de cualquier recorrido.*

*Demostración.* Supongamos que existe algún recorrido $R = v \cdots u$ que no es un camino y cuyo peso $w(R)$ es menor a la longitud de todos los caminos entre $v$ y $u$. Debido a que no es camino, tiene que pasar por dos veces por algún vértice $z$, definiendo un ciclo $R_{z_1 z_2}$[3]. Como el grafo no tiene ciclos negativos, $w(P_{z_1 z_2}) \geq 0$, y entonces se puede definir un nuevo recorrido $R' = R_{vz_1} + R_{z_2 u}$ que "recorta" el ciclo, y cumple $w(R') = w(R) - w(R_{z_1 z_2}) \leq w(R)$. Este proceso puede repetirse hasta conseguir un camino de peso menor o igual al recorrido original, lo cual contradice la suposición inicial. □

**Subestructura óptima**

El problema de encontrar un camino mínimo $P_{uv}$ entre un par de vértices $u$ y $v$ cumple el princpio de optimalidad de Bellman: Cualquier subcamino $P_{u'v'} \subseteq P_{uv}$ es a su vez un camino mínimo entre $u'$ y $v'$.

**Teorema.** *Dado un digrafo pesado $G = (V, E, w)$ sin ciclos negativos y un camino mínimo $P = v_1 \cdots v_k$, el subcamino $P_{v_i v_j}$ es un camino mínimo entre $v_i$ y $v_j$.*

*Demostración.* Supongamos que $P_{v_i v_j}$ no es un camino mínimo. Eso implica que existe un camino alternativo $P' = v_i \cdots v_j$ tal que $w(P') < w(P_{v_i v_j})$. En tal caso, se podría construir el recorrido $R = P_{v_1 v_i} + P' + P_{v_j v_k}$, para el cual existen dos posibilidades:

- Si el recorrido $R$ es un camino, entonces $w(R) = w(P_{v_1 v_i}) + w(P') + w(P_{v_j v_k}) < w(P_{v_1 v_i}) + w(P_{v_i v_j}) + w(P_{v_j v_k}) = w(P)$, lo cual es absurdo porque $P$ es un camino mínimo.
- Si el recorrido $R$ no es un camino, hay algún vértice $v_a$ por el que pasa 2 veces. Esto forma un ciclo $C = R_{v_{a1} v_{a2}}$ dentro del recorrido, que por hipótesis debe tener peso no negativo. Por lo tanto, si se "recorta" el ciclo, se obtiene un recorrido $R' = R_{v_1 v_{a1}} R_{v_{a2} v_k}$ de peso menor o igual. Estos recortes se pueden repetir hasta obtener un camino que, al igual que en el caso anterior, tiene peso estrictamente menor a $P$ (**Absurdo**).

Como ambas alternativas llevan a una contradicción, la suposición inicial ($P_{v_i v_j}$ no es camino mínimo) es falsa. □

### 4.1.3. Árbol de caminos mínimos

Ahora que definimos las restricciones necesarias (no hay ciclos negativos), se pueden empezar a analizar los métodos para encontrar caminos mínimos. Sorprendentemente, el problema de encontrar un camino mínimo entre $v$ y $u$ parece ser igual de difícil que el de encontrar el camino mínimo entre $v$ y todos los vértices del grafo, ya que no se conoce ningún algoritmo que resuelva el primero con una complejidad de peor caso estrictamente menor a los métodos para resolver el otro.

Este nuevo problema se llama *camino mínimo con un origen y múltiples destinos* (SSSP[4]), y los caminos definen un árbol enraizado en $v$, llamado el *árbol de caminos mínimos* (ACM). Un $v$-

[3] $z_1$ y $z_2$ no son vértices distintos, sino formas de distinguir las apariciones de $z$ en $R$.
[4] *Single Source Shortest Path.*

ACM es análogo a un árbol $v$-geodésico, solo que en este caso se cumple $\delta_T(v,u) = \delta_G(v,u)\ \forall u \in V$.

## 4.2. Algoritmo de Dijkstra

El *algoritmo de Dijkstra* es uno de los más famosos en todo el área de computación. Este resuelve el problema de camino mínimo uno a todos, bajo la restricción de que los pesos de las aristas son no negativos.

### 4.2.1. Definición

El procedimiento general es similar al de Prim: se mantiene un árbol enraizado en el vértice inicial $v$ junto con una "frontera" de vértices adyacentes a los del árbol. En cada paso, se agrega el vértice de la frontera más cercano a la raíz. Esto puede ser clasificado como goloso, ya que en cada iteración se toma la decisión más ventajosa a corto plazo. Extendiendo la comparación, mientras que Prim agrega la arista segura de costo mínimo, Dijkstra agrega la que conecta al vértice con menor distancia a $v$.

DIJKSTRA$(G, w, s)$
1 $T = (V_T = \{s\}, E_T = \emptyset)$
2 Inicializar arreglo $d$ con $+\infty$ en cada posición.
3 $d[s] = 0$
4 **for** $i = 1$ **to** $n - 1$
5 $\quad u \to v = \arg\min \{d[x] + w(x \to y) \mid x \in V_T, y \in V - V_T\}$
6 $\quad d[v] = d[u] + w(u \to v)$
7 $\quad V_T = V_T \cup \{v\}$
8 $\quad E_T = E_T \cup \{u \to v\}$
9 **return** $(T, d)$

El diccionario $d$ contiene la distancia mínima entre $s$ y cada vértice. Esto se puede utilizar, entre otras cosas, para obtener el camino mínimo a cualquier vértice $u$ en tiempo lineal: se puede revisar el vecindario de entrada $N^-(u)$ hasta encontrar un vértice $v \in N^-(u)$ tal que $d[v] + w(v \to u) = d[u]$. Esto significa que hay un camino mínimo que pasa por la arista $v \to u$, y el proceso se puede repetir hasta llegar a $s$.

El algoritmo puede modificarse para grafos disconexos: debe mantener un conjunto de vértices conectados, y frenar cuando no quedan nuevos vértices por explorar. El comportamiento en ese caso es devolver los caminos mínimos a los vértices alcanzables (los demás están a distancia $\infty$).

#### Generalización

El procedimiento se puede generalizar para cualquier función $\bullet : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ no decreciente de forma que el costo $w_\bullet(P)$ de un camino $P = v_1 \cdots v_k$ sea:

$$w_\bullet(v_1 \cdots v_k) = \begin{cases} 0 & \text{si } k = 1 \\ w_\bullet(v_1 \cdots v_{k-1}) \bullet c(v_{k-1}v_k) & \text{en caso contrario} \end{cases}$$

La restricción de ser *no decreciente* significa que $w_\bullet(P + y) \geq w_\bullet(P)$ para todo camino $P$ y vértice $y$. La suma cumple esto solo cuando los pesos son no negativos.

Si se define $\bullet$-peso de un camino $P$ al valor $w_\bullet(P)$, se puede llamar camino $\bullet$-mínimo al camino entre $v$ y $u$ con menor $\bullet$-peso (y $\delta_\bullet(v, u)$ al peso de ese camino). Más aún, el árbol de caminos $\bullet$-mínimos de $v$ es uno en el cual $\delta_{G,\bullet}(v, u) = \delta_{T,\bullet}(v, u)$ para todo $u$.

Finalmente, si se tiene un $\bullet$-ACM $T$ enraizado en $v$, se puede definir $w_\bullet(x \to v) = w_\bullet(P) \bullet w(x \to y)$, donde $P$ es el camino de $T$ entre $v$ y $x$. Esto permite diseñar el siguiente algoritmo general:

DIJKSTRA-$\bullet(G, w, s)$
1 $T = (V_T = \{s\}, E_T = \emptyset)$
2 **for** $i = 1$ **to** $n - 1$
3 $\quad u \to v = \arg\min \{w_\bullet(x \to y) \mid x \in V_T, y \in V - V_T\}$
4 $\quad V_T = V_T \cup \{v\}$
5 $\quad E_T = E_T \cup \{u \to v\}$
6 **return** $(T, d)$

Bajo esta definición, el algoritmo de Prim es solo una versión de DIJKSTRA-$\bullet$, donde la función $\bullet(x, y)$ es $\max\{x, y\}$.

### 4.2.2. Correctitud

Se podría demostrar la correctitud de la función DIJKSTRA-$\bullet$, pero para el final es más útil el caso particular de camino mínimo.

**Teorema.** *Dado un (di)grafo pesado sin ciclos negativos $G = (V, E, w)$ y un origen $s \in V$, el algoritmo de Dijkstra devuelve un $s$-ACM $T$, y $d[u] = \delta(v, u)$ $\forall u \in V$.*

*Demostración.* Se procede demostrando la siguiente hipótesis inductiva: para cada $0 \leq k \leq n-1$, el grafo $T_k = (V_k, E_k)$ mantenido por Dijkstra en la $k$-ésima iteración es un árbol que cumple $\delta_{T_k}(s, u) = \delta_G(s, u)$ $\forall u \in V_k$, y además $d[u] = \delta_G(s, u)$ $\forall u \in V_k$.

**Caso Base**: Antes de la primera iteración, $T_0 = (\{s\}, \emptyset)$, y se cumple que $d[s] = 0 = \delta(s, s)$ (por definición).

**Paso Inductivo**: Llamemos $u \to v$ el arco que se agrega al árbol en la $k$-ésima iteración, es decir, $T_k = (V_k, E_k) = (V_{k-1} \cup \{u\}, E_{k-1} \cup \{u \to v\})$. Es claro que $T_k$ sigue siendo un árbol, ya que la nueva arista conecta a un vértice que no estaba en el grafo (y por ende no puede formar un ciclo).

Sea $P = s \cdots y$ un camino mínimo entre $s$ e $y$, es decir, $w(P) = \delta(s, y)$. Tomamos el último vértice $x$ del camino que pertenece a $V_{k-1}$, y sea $y$ el que lo sigue (estos deben existir, porque $x$ está en el conjunto, e $y$ no). Como se demostró anteriormente, $w(P_{su}) = \delta(s, u)$ (es subcamino de un camino mínimo). Como la arista $x \to y$ conecta un vértice de $V_{k-1}$ con uno de $V - V_{k-1}$, se debe cumplir que:

$$d[u] + w(u \to v) \leq d[x] + w(x \to y) \qquad (u \to v \text{ fue elegida por Dijkstra})$$
$$\delta(s, u) + w(u \to v) \leq \delta(s, x) + w(x \to y) \qquad (\text{Por HI, ya que } u, x \in T_{k-1})$$

Entonces, se tiene que el camino une a $s$ con $v$ en $T_k$, $P'$ tiene un peso

$$w(P') = \delta(s,u) + w(u \to v) \leq \delta(s,x) + w(x \to y) \underset{w \geq 0}{\leq} \underbrace{\delta(s,x) + w(x \to y) + w(P_{yv})}_{= w(P),\text{ que es un CM}} = \delta(s,v)$$

Así que el camino $P'$ es mínimo, y $T_k$ cumple que las distancias de sus vértices son las mínimas en el grafo. Por otro lado, el nuevo valor de $d[v] = d[u] + w(u,v)$ es la distancia mínima de $s$ a $v$.

□

### 4.2.3. Complejidad

La complejidad de Dijkstra es análoga a la de Prim: solo es necesario mantener un arreglo adicional con las distancias a cada nodo, lo cual no impone ningún costo en términos de tiempo. Por ende, el tiempo de ejecución dependerá de como se implemente la cola de prioridad de vértices a agregar:

- Arreglo: $\mathcal{O}(|V|^2)$.
- Binary Heap o AVL: $\mathcal{O}(|E| \log |V|)$.
- Fibonacci Heap: $\mathcal{O}(|E| + |V| \log |V|)$.

## 4.3. Bellman-Ford

El *algoritmo de Bellman-Ford*, al igual que el de Dijkstra, resuelve el problema de camino mínimo de uno a todos. Sin embargo, a diferencia de este, Bellman-Ford puede ser utilizado en (di)grafos con pesos negativos (pero sin ciclos negativos). Si el grafo tiene algún ciclo alcanzable desde el vértice de origen, Bellman-Ford es capaz de detectarlo.

### 4.3.1. Definición

Este algoritmo se basa en "*relajar*" las aristas del grafo. La relajación de $u \to v$ implica comprobar si el camino mínimo que va del origen a $v$ puede ser mejorado pasando por la arista. Si las distancias mínimas $\delta(s,v)$ se guardan en $d[v]$, se peude expresar de la siguiente manera:

RELAJAR$(u \to v, w)$
1 $\quad d[v] = \text{mín}\,\{d[u] + w(u \to v), d[v]\}$

Bellman-Ford repite esta operación para cada arista, $|V| - 1$ veces. En cada iteración $i$ ciclo, los vértices se conectan a $s$ por caminos de a lo sumo $i$ aristas.

BELLMAN-FORD$(G, w, s)$

```
1  Inicializar arreglo de distancias d con +∞ en cada posición
2  d[s] = 0
3  for i = 0 to n − 1
4       for each e ∈ E
5            RELAJAR(e, w)
6  for each u → v ∈ E
7       if d[u] + w(u → v) < d[v]
8            return Hay un ciclo negativo
9  return d
```

El último ciclo chequea si existe algún ciclo negativo. Esto se debe a que, si alguna arista se puede relajar después de $i$ iteraciones, el recorrido más corto entre $s$ y algún vértice tiene $|V|$ aristas, lo cual implica que contiene un ciclo negativo (se demuestra más adelante).

El algoritmo puede adaptarse para devolver un $s$-ACM: basta con actualizar el padre de $v$ en RELAJAR, es decir, asignar $prev[v] = u$.

### 4.3.2. Correctitud

Primero analizamos el caso de grafos sin ciclos negativos:

**Teorema.** *Dado un (di)grafo pesado $G = (V, E, w)$ sin ciclos negativos, el algoritmo de Bellman-Ford devuelve en $d$ las distancias $d[v] = \delta(s, v)$.*

*Demostración.* Para demostrar esto, se demuestra la siguiente propiedad hipótesis inductiva: para cada iteración $k$ del ciclo exterior de Bellman-Ford, si un vértice $v$ está conectado a $s$ por un camino mínimo $P$ tal que $|P| \leq k$, se tiene que $d_k[v] = \delta(s, v)$, y $d_k[u] \geq \delta(s, u)\ \forall v \in V$.

**Caso Base**: Para $k = 0$, el único vértice conectado por un camino de longitud 0 es $s$, y se cumple que $d[s] = \delta(s, s) = 0$

**Paso Inductivo**: Primero, demuestro la parte auxiliar de la hipótesis inductiva: sabiendo que $d_{k-1}[u] \geq \delta(s, u)\ \forall v \in V$, se puede analizar la asignación de RELAJAR (es el único procedimiento que cambia $d$ en el algoritmo):

$$d_k[u] = \text{mín}\left\{d_{k-1}[u], d_{k-1}[u'] + w(u' \to u)\right\}$$

Podemos ver que ambas opciones cumplen la propiedad: $d_{k-1}[u] \geq \delta(s, u)$ por HI, mientras que: $d_{k-1}[u'] + w(u' \to u) \geq \delta(s, u') + w(u' \to u) \geq \delta(s, u)$ (desigualdad triangular).

Luego, por hipótesis sabemos que en el paso $k-1$-ésimo se cumplía que $d_{k-1}[v] = \delta(s, v)$ para cualquier vértice $v$ con camino mínimo de longitud menor o igual a $k - 1$. Esto se sigue cumpliendo en $d_k$ para esos mismos vértices gracias a que, como RELAJAR asigna:

$$d_k[v] = \text{mín}\left\{d_{k-1}[v], d_{k-1}[u] + w(u \to v)\right\}$$

Se tiene que $d_k[v] \leq d_{k-1}[v] = \delta(v, s)$. Como demostramos previamente, $d_k[v] \geq \delta(v, s)$, así que $d_k[v] = \delta(v, s)$.

Por otro lado, para aquellos vértices $z$ tales que el camino mínimo de $s$ a $z$ es $P = s \cdots z$ de longitud $|P| = k$, sea $z'$ el anteúltimo vértice. Debido a que $P_{sz'} = s \cdots z'$ es también camino mínimo (es subcamino de un camino mínimo) y tiene longitud $k-1$, sabemos por $HI$ que $d_{k-1}[z'] = \delta(s, z')$. Esto, implica que $d_{k-1}[z'] + w(z' \to z) = \delta(s, z') + w(z' \to z) = w(P) = \delta(s, z)$. Por lo tanto, en la asignación:

$$d_k[z] = \text{mín}\{d_{k-1}[z], d_{k-1}[z'] + w(z' \to z)\}$$

Como $d_{k-1}[z] \geq \delta(s, z)$, se tiene $d_k = \delta(s, z)$.

Finalmente, habiendo demostrado la inducción, nos sirve un caso particular de la misma: después de la última iteración, todos los vértices conectados al origen por un camino mínimo de longitud máxima $|V| - 1$ tendrán los pesos asignados de forma correcta. Estos son pesos son las distancias $\delta(s, v)$, porque en caso contrario el recorrido mínimo tendría que tener longitud mayor a $|V| - 1$, y por ende repetir algún vértice (esto no sucede en grafos sin ciclos negativos).

□

El comportamiento cuando hay ciclos negativos también es correcto:

**Teorema.** *Si un (di)grafo pesado $G = (V, E, w)$ tiene algún ciclo de peso negativo $C$ alcanzable desde un origen $s$, el algoritmo de Bellman-Ford lo detecta.*

*Demostración.* Primero, demuestro la siguiente hipótesis inductiva: en la $k$-ésima iteración de relajaciones, cualquier vértice $u$ con distancia[5] a $s$ menor o igual a $k$ cumple $d_k[v] < \infty$

**Caso Base**: El único vértice alcanzable desde $s$ a través de 0 aristas es $s$ en sí, y $d[s] = 0 < \infty$.

**Paso inductivo**: Si la propiedad vale para $k-1$, entonces $d_{k-1}[u] < \infty$ para los $u$ a distancia $\leq k-1$ de $s$. Después de la iteración $k$, eso se sigue cumpliendo para esos vértices, ya que Relajar asigna un mínimo entre dos valores, uno de los cuales es $d_{k-1}[u]$, así que $d_k[u] \leq d_{k-1}[u] < \infty$.

Por otro lado, si un vértice $u$ está a distancia $k$ de $s$, existe un camino $P = s \cdots u$ tal que $|P| = k$. Si llamamos $u'$ al anteúltimo vértice de $P$, se puede ver que $d_{k-1}[u] < \infty$, ya que el camino $P_{su'}$ tiene longitud $k-1$ (hipótesis inductiva). Luego, el valor $d_k[u]$ está dado por un mínimo entre dos valores, uno de los cuales es $d_{k-1}[u'] + w(u' \to u) < \infty$ (es una suma de valores finitos), así que $d_k[u] < \infty$. Entonces, queda demostrado el caso $k$.

En el caso particular de $k = |V| - 1$, cualquier vértice $v$ alcanzable desde $s$ tiene $d[v] < \infty$, ya que si existe un recorrido entre ambos, se pueden recortar los ciclos del mismo para obtener un camino (que tiene una longitud de a lo sumo $|V| - 1$).

Sea $C = v_0 \cdots v_l$ el ciclo de peso negativo alcanzable desde $v$. Se tiene que:

$$w(C) = \sum_{i=1}^{l} w(v_{i-1} \to v_i) < 0$$

[5] Distancia es la longitud del camino más corto, no el peso del camino mínimo.

Supongamos que el algoritmo no detecta ningún ciclo. Esto implica que $d[u] + w(u \to v) < d[v]$ para todas las aristas del grafo. En particular, $d[v_i] \leq d[v_{i-1}] + w(v_{i+1} \to v_i)\ \forall i = 1...l$. Sumando estas desigualdades:

$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + w(v_{i-1} \to v_i)$$
$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$

Por otro lado, como $C$ es un ciclo, $v_0 = v_l$, y por ende:

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=0}^{k-1} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Como los vértices son alcanzables desde el origen, sus valores $d[v_i]$ son finitos, así que:

$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$
$$\sum_{i=1}^{l} d[v_i] - \sum_{i=1}^{l} d[v_{i-1}] \leq \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$
$$0 \leq \sum_{i=1}^{l} w(v_{i-1} \to v_i) = w(C)$$

Esto es absurdo, así que el algoritmo tiene que haber detectado el ciclo.

□

### 4.3.3. Complejidad

La complejidad de Bellman-Ford es fácil de calcular: RELAJAR es una operación de tiempo constante, y corre $|E|$ veces en cada iteración (una por cada arista). Como hay $|V|-1$ iteraciones, la complejidad es $\mathcal{O}(|V||E|)$ (el último ciclo para comprobar la existencia de ciclos negativos es $\mathcal{O}(|E|)$, lo cual no cambia la complejidad).

### 4.3.4. Mejoras

### 4.3.5. Aplicación: Sistema de Restricciones de Diferencias

Un problema que se puede resolver a través de Bellman-Ford es el de *sistemas de restricciones de diferencias* (SRD). El enunciado es el siguiente:

**Problema:**
Encontrar un conjunto de valores $x_1, ..., x_n$, que respeten un sistema de $m$ inecuaciones de la forma:

$$x_i - x_j \leq c_{ij}$$

El problema se puede reducir a camino mínimo el digrafo $D$, que tiene las siguiente características:

- Para cada $i = 1, ..., n$, hay un vértice $v_i \in V_D$ que se corresponde con la incógnita $x_i$.
- Para cada inecuación $x_i - x_j \leq c_{ij}$, hay un arco $v_j \to v_i \in E_D$ de peso $w(v_j \to v_i) = c_{ij}$.
- Hay un vértice adicional $v_0$, que cuenta con arcos $v_0 \to v_i$ hacia todos los vértices $v_i$, todos de peso $w(v_0 \to v_i) = 0$.

El sistema tiene solución solo cuando el digrafo no tiene ciclos negativos

**Teorema.** *Un sistema SRD tiene solución cuando el digrafo correspondiente $D$ no tiene ciclos negativos, y en tal caso la solución es* $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$.

*Demostración.* Primero, veamos que pasa si $D$ tiene algún ciclo negativo $C = v_{i_0} \cdots v_{i_l}$. Supongamos que existe una solución $\{x_1, ..., x_n\}$. En tal caso, se tiene:

$$\sum_{j=1}^{l} w(v_{i_{j-1}} \to v_{i_j}) = \sum_{j=1}^{l} c_{i_j i_{j-1}} < 0$$

Si se suman las inecuaciones correspondientes del sistema, se llega a:

$$\underbrace{\sum_{j=1}^{l} v_{i_j} - v_{i_{j-1}}}_{\text{Suma telescópica}} \leq \sum_{j=1}^{l} c_{i_j i_{j-1}} < 0$$

$$v_{i_l} - v_{i_0} < 0$$

Como $C$ es un ciclo, $v_{i_l} = v_{i_0}$, así que $0 < 0$ (**Absurdo**). Esto implica que no puede existir una solución para el sistema.

Por otro lado, si no existe ningún ciclo de peso negativo, tomemos la asignación de valores $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$. Entonces, para cada arco del grafo $v_j \to v_i$, se cumple la desigualdad triangular:

$$\delta(v_0, v_i) \leq \delta(v_0, v_j) + w(v_j \to v_i)$$

Reemplazando por los valores representados en el sistema,

$$x_i \leq x_j + c_{ij}$$
$$x_i - x_j \leq c_{ij}$$

Es decir, se cumplen todas las desigualdades, así que $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$ es una solución al sistema.

□

Entonces, el sistema de ecuaciones se puede resolver encontrando el camino mínimo de un origen a todos los vértices o detectando algún ciclo de peso negativo alcanzable, que es precisamente lo que logra el algoritmo Bellman-Ford.

## 4.4. Algoritmo de Floyd-Warshall

### 4.4.1. Definición

El *algoritmo de Floyd-Warshall* resuelve el problema de caminos mínimos todos a todos (APSP[6]). Esto también se puede lograr corriendo alguno de los algoritmos anteriores (que devuelve el camino mínimo de uno a todos) para cada vértice, pero tendría desventajas: Dijkstra solo funciona cuando los pesos son no negativos, y $|V|$ ejecuciones de Bellman-Ford sería (en general) más lento que este nuevo algoritmo. Al igual que antes, el grafo de entrada no puede tener ciclos de peso negativo.

Dado un grafo pesado $G = (V, E, w)$, se puede definir una matriz $L \in \mathbb{R}^{|V|\times|V|}$[7], donde cada posición es el peso de la arista entre un par de vértices (si no existe una arista, el peso se considera infinito). Es decir,

$$l_{ij} = \begin{cases} 0 & \text{si } i = j \\ w(v_i \to v_j) & \text{si } v_i \to v_j \in E \\ \infty & \text{en caso contrario} \end{cases}$$

El método calcula la *matriz de distancias*, esto es, una matriz $D \in \mathbb{R}^{|V|\times|V|}$ tal que:

$$d_{ij} = \begin{cases} \delta(v_i, v_j) & \text{si } v_i \text{ y } v_j \text{ están conectados} \\ \infty & \text{en caso contrario} \end{cases}$$

Para lograrlo, emplea la técnica de programación dinámica, empezando por $D^0 = L$, y calculando la siguiente matriz a través de la recurrencia:

$$d_{ij}^k = \text{mín}\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\}$$

La matriz $D^k$ puede interpretarse como la matriz de distancias mínimas cuando se consideran solo los caminos que pasan por los vértices intermedios $\{v_1, ..., v_k\}$. Luego, después de $n$ iteraciones, la matriz $D^n$ es la matriz de distancias $D$.

El algoritmo se puede expresar de la siguiente manera:

[6] All Pairs Shortest Paths

[7] Esto se puede considerar una extensión de la representación matriz de adyacencia, donde ahora cuando dos vértices están conectados por una arista se guarda su peso, y si no se guarda $\infty$.

FLOYD-WARSHALL($G$)
1 $D = L$
2 **for** $k = 1$ **to** $|V|$
3 **for** $i = 1$ **to** $|V|$
4 **for** $j = 1$ **to** $|V|$
5 $d_{ij} = \min\{d_{ij}, d_{ik} + d_{kj}\}$
6 **return** $D$

En realidad, este algoritmo no implementa la recurrencia directamente: como utiliza una única matriz, hay casos en los que se asigna $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^k + d_{kj}^k\}$. Esto no afecta el resultado, solo tiene el efecto de converger más rápido a la matriz de distancias mínimas.

### 4.4.2. Correctitud

**Teorema.** *Dado un (di)grafo pesado $G = (V, E, w)$ sin ciclos negativos, el algoritmo de Floyd-Warshall devuelve la matriz de distancias del mismo.*

*Demostración.* Para demostrar eso, primero demostramos por inducción que, en la iteración $k$ del ciclo exterior de FW, se cumple $d_{ij}^k = \delta(v_i, v_j)$ para cualquier par de vértices $v_i, v_j \in V$ conectados por un camino mínimo $P$ cuyos vértices intermedios están contenidos en $\{v_1, ..., v_k\}$ (es decir, $P \subseteq \{v_1, ..., v_k\} \cup \{v_i, v_j\}$). Además, se cumple para todos los vértices que $d_{ij}^k \geq \delta(v_i, v_j)$ (en todas las iteraciones).

**Caso Base**: El caso base es $k = 0$. El conjunto de vértices intermedios sería $\emptyset$, y los únicos caminos mínimos sin vértices intermedios son aquellos constituidos por una sola arista. En esos casos, se cumple que $d_{ij}^0 = w(v_i \to v_j) = \delta(v_i, v_j)$, gracias a que se asigna $D^0 = L$.

Por otro lado, como asumimos que no hay ciclos de peso negativo, $\delta(v_i, v_i) = 0 = d_{ii}^0 \forall i$, y para una arista $v_i \to v_j$ se tiene por desigualdad triangular:

$$d_{ij}^0 = w(v_i \to v_j) = \underbrace{\delta(v_i, v_i)}_{=0} + w(v_i \to v_j) \geq \delta(v_i, v_j)$$

**Paso Inductivo**: Supongamos que la propiedad vale en el caso $k - 1$. Recordemos que la asignación que se realiza en la iteración $k$ es:

$$d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$$

Para demostrar que $d_{ij}^k \geq \delta(v_i, v_j)$, notemos dos cosas:

- $d_{ij}^{k-1} \geq \delta(v_i, v_j)$, por HI.
- $d_{ik}^{k-1} + d_{kj}^{k-1} \geq \delta(v_i, v_k) + \delta(v_k, v_j) \geq \delta(v_i, v_j)$, por HI y desigualdad triangular, respectivamente.

En todo caso, se cumple que $d_{ij}^k \geq \delta(v_i, v_j)$.

Para los pares de vértices con caminos mínimos formados por vértices intermedios en el conjunto $\{v_1, ..., v_{k-1}\}$, se sigue cumpliendo la propiedad, ya que la HI implica que $d_{ij}^{k-1} =$

$\delta(v_i, v_j)$ (así que $d_{ij}^k \leq \delta(v_i, v_j)$), y como recién demostramos que $d_{ij}^k \geq \delta(v_i, v_j)$, se tiene $d_{ij}^k = \delta(v_i, v_j)$.

Por otro lado, sea $P = v_x \cdots v_y$ algún camino mínimo entre $v_x$ y $v_y$ que contiene al vértice $v_k$ y cuyos vértices intermedios están contenidos en $\{v_1, ..., v_k\}$. $P$ se puede descomponer como la concatenación $P_{v_x v_k} + P_{v_k v_y}$, y estos dos subcaminos (que son mínimos) están formados por vértices intermedios del conjunto $\{v_1, ..., v_{k-1}\}$ (si estuviera $v_k$ en el medio, aparecería dos veces, y no serían caminos simples). Esto, por HI, implica que $d_{xk}^{k-1} = \delta(v_x, v_k) = w(P_{v_x v_k})$ y $d_{ky}^{k-1} = \delta(v_k, v_y) = w(P_{v_k v_y})$, así que cuando se asigna:

$$d_{ij}^k = \text{mín}\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\}$$

Se obtiene $d_{xy}^k \leq d_{xk}^{k-1} + d_{ky}^{k-1} = w(P_{v_x v_k}) + w(P_{v_k v_y}) = w(P) = \delta(v_x, v_y)$ que, junto con $d_{xy}^k \geq \delta(v_x, v_y)$ (se demostró previamente), significa que $d_{xy}^k = \delta(v_x, v_y)$. Queda demostrado entonces que la propiedad vale para el caso $k$.

Como, por inducción, la propiedad se cumple después de cualquier iteración $k$, se cumple en particular para después de la última, lo cual significa que todo par de vértices $v_i, v_j$ conectado por un camino mínimo con vértices intermedios en $\{1, ..., v_{|V|}\} = V$ cumple $d_{ij}^k = \delta(v_i, v_j)$. Esto incluye a cualquier par de vértices conectados, y para los vértices que no son alcanzables entre sí, la segunda propiedad garantiza que $d_{ij}^k \geq \delta(v_i, v_j) = \infty \implies d_{ij}^k = \infty$. Por ende, todas las distancias de $D^{|V|}$ son correctas.

□

### 4.4.3. Complejidad

La complejidad de Floyd-Warshall es fácil de calcular: tiene 3 ciclos anidados, de $|V|$ iteraciones cada uno, en cuyo interior se realizan operaciones de tiempo constante, así que el algoritmo es $\mathcal{O}(|V|^3)$. Como se mencionó previamente, esto es mejor que correr $n$ veces Bellman-Ford, siempre que el grafo sea conexo (porque en ese caso $|E| \geq |V| - 1 \implies |E| \in \Omega(|V|)$).

Una posible mejora sería chequear en cada iteración si $d_{ii} < 0$ para algún $i$, ya que esto implica que el grafo tiene un ciclo negativo (en el teorema anterior se demuestra $d_{ii} \geq \delta(v_i, v_i) = 0$ cuando no hay ciclos negativos), y cortar la ejecución.

## 4.5. Camino Mínimo en DAGs

### 4.5.1. Definición

Para el caso de digrafos acíclicos, camino mínimo uno a muchos puede resolverse utilizando un algoritmo más eficiente que los anteriores. Este se basa en la siguiente fórmula para la distancia entre dos nodos:

$$\delta(u, v) = \begin{cases} 0 & \text{si } u = v \\ \text{mín}\left\{\delta(u, z) + w(z \to v) \mid z \in N^-(v)\right\} & \text{en caso contrario} \end{cases}$$

Esto es intuitivo: la distancia mínima entre un par de nodos $u, v$ es el peso de algún camino

mínimo $P = u \cdots v$. El anteúltimo vértice del camino $z$ está en el vecindario de entrada de $v$ (ya que tiene una arista que apunta hacia él), mientras que el subcamino $P_{uz}$ también es mínimo.

Esta relación no se puede utilizar para computar las distancias en un grafo con un ciclo $C = v \cdots v$, ya que la llamada $\delta(u, v)$ llevaría a un bucle de recursión infinita (porque existe un camino de $v$ a $v$ a través de los vecindarios de entrada).

#### Versión Top-Down

La fórmula se puede implementar como un algoritmo de programación dinámica top-down. Asumiendo que el arreglo de distancias (que funciona estructura de memoización) $D$ se inicializa con $\perp$ en todas las posiciones, el código es el siguiente:

SP-DAG-Top-Down$(G, s, v)$

```
1  if s == v
2      return 0
3  if D[v] == ⊥
4      D[v] = mín {SP-DAG-Top-Down(G, s, z) + w(z → v) | z ∈ N⁻(v)}
5  return D[v]
```

#### Versión Bottom-Up

Esto también se puede implementar de forma bottom-up: las distancias son calculadas en orden topológico, lo cual asegura que $D[z]$ es calculado antes de $D[v]$ para cualquier $z \in N^-(v)$[8].

SP-DAG-Bottom-Up$(G, s)$

```
 1  Inicializar arreglo de distancias D.
 2  Calcular un orden topológico de G.
 3  for each v ∈ V, en orden topológico
 4      if s == v
 5          D[v] = 0
 6      else
 7          if N⁻(v) ≠ ∅
 8              D[v] = mín {D[z] + w(z → v) | z ∈ N⁻(v)}
 9          else
10              D[v] = ∞
11          return D
```

### 4.5.2. Complejidad

Ambos métodos tienen una complejidad $\mathcal{O}(|V| + |E|)$ ya que pasan 1 vez por cada nodo $v$, realizando una operación $\mathcal{O}(d^-(v))$. En el caso bottom-up, se realiza el paso adicional de computar un orden topológico, pero esto también se puede realizar en tiempo lineal a través de DFS.

[8] Esto se debe a que, en un orden topológico, los vértices $v$ que son alcanzables desde $z$ aparecen después en la secuencia

## 4.6. Algoritmo de Johnson

### 4.6.1. Definición

El algoritmo de Johnson resuelve el problema de camino mínimo todos a todos. Es una alternativa al algoritmo de Floyd-Warshall, y es más eficiente para grafos ralos. La idea general del método es "repesar" las aristas del grafo, de forma que todos los pesos nuevos sean no negativos y se preserven los caminos mínimos, para luego aplicar el algoritmo de Dijkstra en cada nodo.

El proceso de repesar se basa en el siguiente teorema:

**Teorema.** *Dado un (di)grafo pesado $G = (V, E, w)$ y una función $h : V \longrightarrow \mathbb{R}$, se puede definir una nueva función de peso:*

$$\hat{w}(u \to v) = w(u \to v) + h(u) - h(v)$$

*Luego, si se toma el grafo "repesado" $\hat{G} = (V, E, \hat{w})$, cualquier camino mínimo $P$ en $G$ sigue siendo mínimo en $\hat{G}$, y $G$ tiene un ciclo negativo $\iff$ $\hat{G}$ tiene un ciclo negativo.*

*Demostración.* Dado un recorrido cualquiera $P = \langle v_0, ..., v_k \rangle$ se puede ver que:

$$\begin{aligned}
\hat{w}(P) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \underbrace{\sum_{i=1}^{k} h(v_{i-1}) - h(v_i)}_{\text{Suma telescópica}} \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(P) + h(v_0) - h(v_k)
\end{aligned}$$

Por ende, si $P'$ es un camino mínimo entre $u$ y $v$ en $G$, se tiene:

$$\begin{aligned}
w(P') &\leq w(P) && \forall P \text{ camino entre } u \text{ y } v \\
w(P') + h(u) - h(v) &\leq w(P) + h(u) - h(v) && \forall P \text{ camino entre } u \text{ y } v \\
\hat{w}(P') &\leq \hat{w}(P) && \forall P \text{ camino entre } u \text{ y } v
\end{aligned}$$

Entonces, $P'$ sigue siendo camino mínimo en $\hat{G}$.

Por otro lado, dado un ciclo $C = u \cdots u$, su peso en $\hat{G}$ es $\hat{w}(C) = w(C) + h(u) - h(u) = w(C)$, así que $\hat{G}$ tiene ciclos negativos si y solo si $G$ los tiene.

□

Para aprovechar este resultado, hace falta encontrar una función $h : V \longrightarrow \mathbb{R}$ que permita asignar valores no negativos a todos los pesos. El algoritmo de Johnson logra esto agregando un vértice adicional $s$ a $G$, conectado a todos los vértices con aristas de peso 0, y calculando las distancias mínimas de $s$ a los demás (a través del algoritmo de Bellman-Ford, acá es donde el procedimiento falla si hay un ciclo negativo). Luego, toma $h(v) = \delta(s, v)$, que resulta en pesos no negativos ya que:

$$\begin{aligned} \delta(s,v) &\leq \delta(s,u) + w(u \to v) && \text{(Desigualdad triangular)} \\ h(v) &\leq h(u) + w(u \to v) \\ 0 &\leq w(u \to v) + h(u) - h(v) \\ 0 &\leq \hat{w}(u \to v) \end{aligned}$$

Luego, se puede correr Dijkstra en cada vértice para obtener las distancias de todos a todos. El código es el siguiente:

JOHNSON$(G, w)$
1 $G' = (V \cup \{s\}, E \cup \{s \to v \mid s \in E\})$, $w(s, v) = 0\ \forall v \in V$
2 **if** BELLMAN-FORD$(G', w, s)$ encuentra un ciclo negativo
3 **return** Hay un ciclo negativo
4 **else** BELLMAN-FORD devolvió las distancias $\delta(s, v)\ \forall v \in V$.
5 Definir $\hat{w}(u \to v) = w(u \to v) + \delta(s, u) - \delta(s, v)\ \forall u \to v \in E$.
6 Inicializar matriz $D \in \mathbb{R}^{|V| \times |V|}$
7 **for each** $u \in V$
8 Correr DIJKSTRA$(G, \hat{w}, u)$ para obtener $\hat{\delta}(u, v)\ \forall v \in V$
9 Asignar $d_{uv} = \hat{\delta}(u, v) - \delta(s, v) + \delta(s, u)\ \forall v \in V$
10 **return** $D$

### 4.6.2. Complejidad

La ejecución del algoritmo de Johnson se divide en dos partes. Primero, se ejecuta Bellman-Ford en el vértice agregado $s$, lo cual tiene una complejidad de $\mathcal{O}(|V||E|)$. Luego, se corre el algoritmo de Dijkstra una vez en cada vértice, y el tiempo de ejeución de eso depende de la cola de prioridad que utilice ese procedimiento. Si se utiliza un Fibonacci heap, la complejidad de Dijkstra es $\mathcal{O}(|V| \log |V| + |E|)$, así que la complejidad total de Johnson es $\mathcal{O}(|V|^2 \log |V| + |V||E|)$.

# Capítulo 5

# Flujo en Redes

## 5.1. Flujo máximo

Una *red de flujo* es un digrafo $N = (V, E)$ junto con una función de *capacidad* $u : E \longrightarrow \mathbb{Z}_+$ y un par de nodos $s, t \in V$, llamados origen y destino. Un *flujo factible* $x : E \longrightarrow \mathbb{Z}_+$ en esa red es una asignación de valores a las aristas, de forma que se cumpla:

- $0 \leq x(v_i \to v_j) \leq u(v_i \to v_j)$ (Respeta las capacidades).
- $\forall v_i \in V - \{s, t\}$,

$$\sum_{u \in N^+(v)} x(u \to v) = \sum_{u \in N^-(v)} x(v \to u) \qquad \textit{(Conservación)}$$

El *valor del flujo* $x$ es la cantidad neta que sale del vértice $s$, es decir:

$$F = \sum_{u \in N^+(s)} x(u \to s) - \sum_{u \in N^-(s)} x(s \to u)$$

Entonces, se puede definir el problema de flujo máximo:

**Problema:**
Dada una red de flujo $N = (V, E, u)$ con origen $s$ y destino $t$, encontrar un flujo factible $x$ que tenga valor $F$ máximo. Es decir, maximizar:

$$F = \sum_{u \in N^+(v)} x(u \to s) - \sum_{u \in N^-(v)} x(s \to u)$$

Sujeto a:
- $0 \leq x(e) \leq u(e)\ \forall e \in E$
- $\forall v_i \in V - \{s, t\}$,

$$\sum_{u \in N^+(v)} x(u \to v) = \sum_{u \in N^-(v)} x(v \to u)$$

Las unidades del flujo se pueden considerar "paquetes" que viajan del origen $s$ al destino $t$, donde cada paquete puede tomar cualquier ruta que tenga la capacidad necesaria.

### 5.1.1. Corte

Un *corte* en una red de flujo $N = (V, E, u)$ es un subconjunto $S \subseteq V$ tal que $s \in S$ y $t \notin S$. Por otro lado, dado un par de subconjuntos $S, T \subseteq V$, se define:

$$ST = \{u \to v \mid u \to v \in E, u \in S, v \in T\}$$

**Teorema.**

*Sea $x$ un flujo definido en una red $N = (V, E, u)$ y sea $S$ un corte. Entonces:*

$$F = \sum_{e \in S\bar{S}} x(e) - \sum_{e \in \bar{S}S} x(e)$$

*Donde $\bar{S} = V - S$.*

*Demostración.* Sumando los flujos netos que pasan por los vértices de $S$, se obtiene:

$$\sum_{v \in S} \underbrace{\left( \sum_{u \in N^+(v)} x(u \to v) - \sum_{u \in N^-(v)} x(v \to u) \right)}_{\text{Siempre 0, excepto para } v = s} = F + 0$$

Por otro lado, los términos de la sumatoria representan todas las aristas que conectan un nodo de $S$ con uno de $V$, así que:

$$F = \sum_{e \in SV} x(e) - \sum_{e \in VS} x(e)$$

Las aristas de $SS$ aparecen en ambas sumatorias, así que se cancelan. Los únicos términos que quedan los las aristas que están en $S(V-S) = S\bar{S}$:

$$F = \sum_{e \in S\bar{S}} x(e) - \sum_{e \in \bar{S}S} x(e)$$

□

Por otro lado, la capacidad de un corte $S$ es la cantidad de flujo que puede salir del mismo, es decir:

$$u(S) = \sum_{e \in S\bar{S}} u(e)$$

Luego, se puede definir otro problema de flujo, el de corte mínimo:

> **Problema:**
> Dada una red de flujo $N = (V, E, u)$ con origen $s$ y destino $t$, encontrar el corte $S$ con capacidad mínima.

A partir del último teorema, se puede ver que:

$$F = \sum_{e \in S\bar{S}} \underbrace{x(e)}_{\leq u(e)} - \sum_{e \in \bar{S}S} \underbrace{x(e)}_{\geq 0} \leq u(S)$$

Esto vale para cualquier valor de flujo $F$, y cualquier corte $S$. Esto es intuitivo: como las unidades de flujos empiezan en el origen $s$, todas deben pasar por alguna arista de $S\bar{S}$ para llegar al destino $t$.

### 5.1.2. Certificado de optimalidad

Un *certificado de optimalidad* para una instancia de un problema de optimización es un valor que permite verificar que la solución una solución dada es efectivamente óptima (y, generalmente, hacerlo en menor tiempo que el que toma resolver el problema). El flujo máximo cuenta con uno: si se encuentra un flujo factible $x^*$ y un corte $S^*$ tales que $F^* = u(S)$, se tiene que, para cualquier valor de flujo $F$,

$$F \leq u(S) = F^*$$

Así que $x^*$ es un flujo máximo.

### 5.1.3. Red residual y camino de aumento

Dada una red de flujo $N = (V, E, u)$ y un flujo factible $x$, la *red residual* $R(N, x) = (V, E_R)$ es un digrafo[1] definido por:

[1] En realidad, $R(N, x)$ es un multigrafo, debido a que cuando $N$ contiene aristas $v_i \to v_j$ y $v_j \to v_i$, y estas cumplen $0 < x(e) < u(e)$, la red residual tiene 4 aristas entre $v_i$ y $v_j$.

- $x(v \to w) < u(v \to w) \iff v \to w \in E_R$
- $x(v \to w) > 0 \iff w \to v \in E_R$

Además, se puede definir la *capacidad residual* de una arista de $R(N, x)$ de la siguiente manera:

$$\Delta(v \to w) = \begin{cases} u(v \to w) - x(v \to w) & \text{si } v \to w \in E \\ x(v \to w) & \text{en caso contrario} \end{cases}$$

Luego, un *camino de aumento* es un camino orientado en el grafo $R(N, x)$ que va de $s$ a $t$. Para un camino $P$ se define $\Delta(P) = \min_{e \in P} \{\Delta(e)\}$.

**Teorema.** *Dada una red de flujo $N = (V, E, u)$ junto con un flujo factible $x$, si $P$ es un camino entre $s$ y $t$ en la red residual $R(N, x)$, se puede definir un nuevo flujo $\bar{x}$:*

$$\bar{x}(v \to w) = \begin{cases} x(v \to w) + \Delta(P) & \textit{si } v \to w \in P \\ x(w \to v) - \Delta(P) & \textit{si } w \to v \in P \\ x(v \to w) & \textit{en caso contrario} \end{cases}$$

*Luego, $\bar{x}$ es un flujo factible, y tiene un valor $\bar{F} = F + \Delta(P)$.*

*Demostración.* Para facilitar la demostración, introduzco una notación no estándar: si $e = u \to v$, $e^T := v \to u$.

Primero, verifiquemos que $\bar{x}$ es un flujo factible.

- **Respeta las capacidades**: Solo hace falta verificar las capacidades de las aristas de $P$, ya que el flujo que pasa por las demás no se modifica (y $\bar{x}$ es un flujo factible). Entonces, veamos que pasa para las aristas $e \in P$.

  Si $e \in E$, entonces $x(e) < u(e)$ (ya que la arista también está en la red residual). Además, por la definición de $\Delta(P)$, se tiene que:

  $$\Delta(P) \leq \Delta(e) = u(e) - x(e)$$

  Esto implica que el nuevo flujo de la arista respeta la capacidad:

  $$\bar{x}(e) = x(e) + \Delta(P) \leq x(e) + u(e) - x(e) = u(e)$$

  Por otro lado, si $e^T \in E$, eso implica que $x(e^T) > 0$ (ya que la arista inversa está en la red residual). En cuanto a la capacidad residual de la arista,

  $$\Delta(P) \leq \Delta(e) = x(e^T)$$

  Entonces, el nuevo flujo es no negativo:

  $$\bar{x}(e^T) = x(e^T) - \Delta(P) \geq x(e^T) - x(e^T) = 0$$

- **Respeta la conservación**: De nuevo, para vértices por los que no pasa el camino, el flujo no cambia, así que la conservación se mantiene.

  Para cada vértice $v \in P - \{s, t\}$, sean $e_1 = a \to v$ y $e_2 = v \to b$ las dos[2] aristas de $P$ que contienen a $v$. Observemos que el flujo neto que pasa por $v$ es:

$$\sum_{e \in N^+(v)} \bar{x}(e) - \sum_{e \in N^-(v)} \bar{x}(e)$$

  Para los valores de flujo de $e_1, e_2$, existen 4 posibilidades:

  1. $e_1, e_2 \in E$: el flujo neto es:

$$\begin{aligned}\sum_{e \in N^+(v)} \bar{x}(e) - \sum_{e \in N^-(v)} \bar{x}(e) &= \sum_{e \in N^+(v) - \{e_1\}} x(e) - \sum_{e \in N^-(v) - \{e_2\}} x(e) + \bar{x}(e_1) - \bar{x}(e_2) \\ &= \sum_{e \in N^+(v) - \{e_1\}} x(e) - \sum_{e \in N^-(v) - \{e_2\}} x(e) + x(e_1) + \Delta(P) - x(e_2) - \Delta(P) \\ &= \sum_{e \in N^+(v)} x(e) - \sum_{e \in N^-(v)} x(e) = 0\end{aligned}$$

  2. $e_1^T, e_2^T \in E$:
  3. $e_1, e_2^T \in E$:
  4. $e_1^T, e_2 \in E$:

Por otro lado, para calcular el valor del nuevo flujo, vamos a asumir que el vértice $s$ no tiene aristas entrantes ($d^-(s) = 0$). Esto es razonable: si hay flujo entrante a $s$, se forma un ciclo que se puede "cancelar", así que no es necesario modelarlo. Bajo esta suposición, el valor del flujo $\bar{x}$ es:

$$\bar{F} = \sum_{w \in N^-(s)} x(w \to s) - \sum_{w \in N^+(s)} x(s \to w)$$

□

## 5.2. Método de Ford-Fulkerson

El *método de Ford-Fulkerson* es un esquema para resolver el problema de flujo máximo en una red de flujo basado en encontrar caminos de aumento dentro de la red residual. Esto funciona gracias al siguiente teorema:

**Teorema.** *Sea $N = (V, E, u)$ una red de flujo y $x$ un flujo factible en esa red. Luego, $x$ es un flujo máximo $\iff$ no existe camino de aumento en $R(G, x)$.*

*Demostración.*

[2] Son dos aristas porque, por ser camino, no se pasa dos veces por $v$.

$\Longrightarrow$ ) Se demuestra el contrarrecíproco: por el teorema anterior, si existe un camino de aumento $P$, se puede definir el flujo $\bar{x}$ que tiene como valor $\bar{F} = F + \Delta(P)$. Como $\Delta(P) = \text{mín}_{e \in P} \{\Delta(e)\}$ y, por definición $\Delta(e) > 0 \ \forall e \in E_R$, se tiene que $\Delta(P) > 0$, así que $\bar{F} > F$. Por lo tanto, $x$ no es un flujo máximo.

$\Longleftarrow$ ) Supongamos que no existe camino de aumento en $R(G, x)$. Entonces, sea $S$ el corte que contiene a todos los vértices alcanzables desde $s$ en la red residual (es un corte porque, si contuviera a $t$, habría un camino de aumento). Entonces, si se consideran las aristas $v \to w \in S\bar{S}$, sabemos que $v \to w \notin E_R$, así que $x(v \to w) = u(v \to w)$. Además, para las aristas $v' \to w' \in S\bar{S}$, también sabemos que $w' \to v \notin E_R$, así que $x(v' \to w') = 0$. Por ende, se tiene:

$$F = \sum_{e \in S\bar{S}} x(e) - \sum_{e \in \bar{S}S} x(e) = \sum_{e \in S\bar{S}} u(e) - 0 = u(S)$$

Como se demostró anteriormente, si se encuentra un flujo $x$ de valor $F$ y un corte $S$ tal que $F = u(S)$, el flujo es máximo y el corte mínimo.

□

Por ende, si se incrementa el flujo a través de caminos de aumento hasta que la red residual no tenga más, se llega a un flujo máximo. El esquema es el siguiente:

FORD-FULKERSON($N = (V, E, u)$)
1 Inicializar flujo inicial $x$ con $x(e)$ $\forall e \in E$.
2 **while** $\exists P$ camino de aumento en $R(N, x)$
3 **for each** $(v \to w) \in P$
4 **if** $(v \to w) \in E$
5 $x(v \to w) = x(v \to w) + \Delta(P)$
6 **else**
7 $x(w \to v) = x(w \to v) - \Delta(P)$
8 **return** $x$

Ford-Fulkerson se denomina un esquema, y no un algoritmo[3], porque el criterio con el cuál encuentra el camino de aumento $P$ en cada iteración no se especifica. Existen algoritmos que implementan este esquema con una elección particular de camino de aumento, como el de Edmonds-Karp.

### 5.2.1. Complejidad

Para analizar la complejidad del esquema de Ford-Fulkerson, es necesario acotar la cantidad de iteraciones que hace el ciclo exterior. Para ello, nos valemos del siguiente teorema:

**Teorema.** *Si las capacidades de una red $N = (V, E, u)$ cumplen $u(e) \in \mathbb{Z}$, el flujo máximo de $N$ tiene también valores enteros, y el esquema de Ford-Fulkerson puede encontrar siempre un camino de aumento $P$ con $\Delta(P) \in \mathbb{Z}$.*

*Demostración.* Se puede demostrar haciendo inducción en las iteraciones de F&F: en el paso

[3] A veces sí se lo llama algoritmo, y en ese caso no se especifica el camino que se elige.

$k$-ésimo, el flujo $x_k$ tiene valores enteros, y se puede encontrar un camino de aumento con valor entero.

**Caso Base**: En el paso 0, el flujo $x$ vale $x(e) = 0 \in \mathbb{Z}$ para todas las aristas $e$.

**Paso Inductivo**: Asumiendo que en el paso $k-1$-ésimo vale la propiedad, se tiene que $x_{k-1}(e) \in \mathbb{Z}\ \forall e \in E$. Sabemos que si hay algún camino de aumento $P$, su valor $\Delta(P)$ se define como:

$$\Delta(P) = \text{mín}\,\{\Delta(e) \mid e \in P\}$$

Luego, como $\Delta(e) \in \{u(e) - x_{k-1}(e), x_{k-1}(e)\} \subseteq \mathbb{Z}$, se tiene que $\Delta(P) \in \mathbb{Z}$. Esto además implica que $x_k$ tiene valores enteros, porque $x_k$ se define como:

$$x_k(v \to w) = \begin{cases} x_{k-1}(v \to w) + \Delta(P) & \text{si } v \to w \in P \\ x_{k-1}(w \to v) - \Delta(P) & \text{si } w \to v \in P \\ x_{k-1}(v \to w) & \text{en caso contrario} \end{cases}$$

En todos los casos, los valores son enteros, así que la propiedad vale para el caso $k$.

□

A partir de este teorema se puede ver que, cuando las capacidades son enteros, el esquema de F&F realiza como máximo $F$ iteraciones: en cada paso, se encuentra un camino de aumento $P$ que cumple $\Delta(P) \geq 0 \wedge \Delta(P) \in \mathbb{Z} \implies \Delta(P) \geq 1$, así que en a lo sumo $F$ pasos se llega a un flujo de valor $F$. Esto implica que la complejidad total del algoritmo es $\mathcal{O}(|V|F)$, porque en cada iteración se realiza una operación constante para cada vértice del camino $P$ (que puede contener a todos los vértices).

Para expresar la complejidad sin saber el flujo máximo, se pueden encontrar cotas superiores para el mismo, como la suma de las capacidades $\sum_{e \in E} u(E)$, o el máximo máx $\{u(e) \mid e \in E\}$.

### 5.2.2. Algoritmo de Edmonds-Karp

El *algoritmo de Edmonds-Karp* es una versión del de Ford-Fulkerson. En este caso, el camino de aumento es encontrado a través de BFS. Esto permite asegurar una cota de $\mathcal{O}(|V||E|^2)$.

## 5.3. Matching Máximo en Grafo Bipartitos

### 5.3.1. Definición

Dado un grafo bipartito $G = (V, E)$ con bipartición $(V_1, V_2)$, un *matching* $M \subseteq E$ es un conjunto de aristas tal que:

$$\forall u \to v \in M,\ (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$$

Otra definición equivalente para $M$ sería que cualquier vértice $v \in V$ es incidente a a lo sumo 1 arista de $M$. El problema de matching máximo en grafos bipartitos es:

**Problema:**
Dado grafo bipartito $G = (V, E)$, encontrar el matching $M^*$ de cardinal máximo. Es decir,

$$M^* = \arg\max \{\#M \mid M \text{ es matching en } G\}$$

Este problema se puede resolver reduciéndolo a flujo máximo.

### 5.3.2. Solución

Para resolver matching máximo en el grafo bipartito $G = (V, E)$, se puede definir la red de flujo $N = (V', E', u)$ de la siguiente forma:

- $V' = V \cup \{s, t\}$
- $E' = E \cup \{s \to v \mid v \in V_1\} \cup \{w \to t \mid w \in V_2\}$
- $u(e) = 1 \ \forall e \in E'$

Luego, el flujo máximo de la red pasará por un conjunto de aristas que conforman un matching $M$ de cardinal máximo.

## 5.4. Flujo de costo mínimo

**Problema:**
Dada una red de flujo $N = (V, E, u)$ que tiene además:
- Una función de costo unitario (por cada unidad de flujo) en cada arista $c : E \to \mathbb{Z}_+$.
- Un *imbalance* para cada nodo $b : V \to \mathbb{Z}_+$.

Encontrar el *flujo de costo mínimo* $x : E \to \mathbb{Z}_+$, esto es, un flujo que:
- Respeta el imbalance de cada nodo $v \in V$:

$$b(v) = \sum_{w \in N^+(v)} x(v \to w) - \sum_{w \in N^-(v)} x(w \to v)$$

- Respeta las capacidades de las aristas $e \in E$:

$$0 \leq x(e) \leq u(e)$$

- Tiene costo $T = \sum_{e \in E} c(e) \cdot x(e)$ es mínimo.

En este caso, vamos a asumir que los valores son enteros, y que los imbalances se balancean ($\sum_{v \in V} b(v) = 0$).

Cuando los imbalances son todos nulos, el flujo se denomina una *circulación*. Las circulaciones cumplen la siguiente propiedad:

**Teorema.** *Dada una red de flujo $N = (V, E, u)$, una circulación $x$ puede ser descompuesta en combinación de circulaciones $x_1, ..., x_k$ tal que:*

$$x(e) = \sum_{i=1}^{k} x_i(e)$$

*Y cada circulación $x_i$ está compuesta por un único ciclo $C$ (por el resto de las aristas $e' \in E - C$, el flujo $x_i(e) = 0$).*

*Demostración.* Esto se puede demostrar de forma constructiva: tomemos un nodo cualquiera $v$ que sea cola de un arco con flujo no nulo. Como no hay imbalances en la red, es claro que $v$ tiene que ser parte de un ciclo con flujo positivo: si no, se podría tomar un camino $P$ maximal que se extiende a izquierda y derecha por aristas de flujo positivo. Si no se forma un ciclo, el vértice al principio del camino tendría flujo neto positivo, y el del final flujo neto negativo.

Sabiendo que $v$ forma parte de algún ciclo $C$ compuesto por aristas con flujo no nulo, se puede definir $\Delta(C) = \text{mín}\{x(e) \mid e \in C\}$. Luego, podemos tomar un flujo alternativo $\bar{x}$ definido como

$$\bar{x}(e) = \begin{cases} x(e) - \Delta(C) & \text{si } e \in C \\ x(e) & \text{en caso contrario} \end{cases}$$

El flujo que pasa por los nodos fuera de $C$ no se ve afectado, mientras que los que están en el ciclo mantienen la conservación, ya que cada $\Delta(C)$ unidades que pierden por una arista de entrada son compensadas por una pérdida equivalente en una arista de salida. Esto significa que $\bar{x}$ es una circulación válida, y además sabemos que el flujo que pasa por las aristas de $C$ es estrictamente menor.

El proceso anterior puede repetirse hasta que no queden aristas con flujo no nulo, en cuyo caso la circulación es nula. En cada paso $i$, se obtuvo una circulación nueva restando un ciclo $C_i$, que se puede expresar como circulación $x_i$ donde:

$$x_i = \begin{cases} \Delta(C_i) & \text{si } e \in C_i \\ 0 & \text{en caso contrario} \end{cases}$$

Así que $x$ puede expresarse como la sumatoria:

$$x(e) = \sum_{i=1}^{k} x_i(e)$$

□

### 5.4.1. Red Residual

Análogamente al caso de flujo máximo, se puede definir una *red residual* $R(N, x) = (V, E_R)$ para una red de flujo pesada $N = (V, E, u, c, b)$ y un flujo válido $x$. Al igual que antes, las aristas de $E_R$ están definidas por:

- $x(v \to w) < u(v \to w) \iff v \to w \in E_R$
- $0 < x(v \to w) \iff w \to v \in E_R$

Cada arista de $E_R$ cuenta con una *capacidad residual*:

$$r(v \to w) = \begin{cases} u(v \to w) - x(v \to w) & \text{si } v \to w \in E \\ x(w \to v) & \text{si } w \to v \in E \end{cases}$$

Y además tienen los costos dados por:

$$c_R(v \to w) = \begin{cases} c(v \to w) & \text{si } v \to w \in E \\ -c(v \to w) & \text{si } w \to v \in E \end{cases}$$

Esta red residual nos permite establecer una *condición de optimalidad* análoga a la de "no hay camino de aumento" en flujo máximo.

**Teorema.** *Dada una red de flujo pesada $N = (V, E, u, c, b)$, un flujo factible $x$ es de costo mínimo $\iff$ la red residual $R(N, x)$ no cuenta con ningún ciclo de costo negativo.*

*Demostración.*

$\implies$ ) Se demuestra por el contrarrecíproco: supongamos que $R(N, x)$ cuenta con un ciclo de costo negativo $C$. Definamos entonces $r(C) = \text{mín}\,\{r(e) \mid e \in C\}$, y a partir de eso un nuevo flujo $\bar{x}$ de la siguiente forma:

$$\bar{x}(v \to w) = \begin{cases} x(v \to w) + r(C) & \text{si } v \to w \in C \\ x(w \to v) - r(C) & \text{si } w \to v \in C \\ x(v \to w) & \text{en caso contrario} \end{cases}$$

Como se demostró para un caso análogo en la sección anterior, este flujo es factible. Además, si se calcula el costo total de $\bar{x}$, se tiene:

$$\begin{aligned} \bar{T} = \sum_{e \in E} c(e) \cdot \bar{x}(e) &= \sum_{e \notin C, e^T \notin C} c(e)x(e) + \sum_{e \in C} c(e)(x(e) + r(C)) + \sum_{e^T \in C} c(e)(x(e) - r(C)) \\ &= \sum_{e \in E} c(e) \cdot x(e) + r(C) \left( \sum_{e \in C} c(e) - \sum_{e^T \in C} c(e) \right) \\ &= T + r(C) \left( \sum_{e \in C} c(e) - \sum_{e^T \in C} c(e) \right) \end{aligned}$$

Sabemos que $C$ es un ciclo negativo, con un costo definido por:

$$\begin{aligned} c_R(C) = \sum_{e \in C} c_R(e) &= \sum_{e \in C} c(e) + \sum_{e^T \in C} -c(e) \\ &= \sum_{e \in C} c(e) - \sum_{e^T \in C} c(e) < 0 \end{aligned}$$

Por ende, el costo de $\bar{x}$ es:

$$\bar{T} = T + \underbrace{r(C)}_{\geq 0}\underbrace{c_R(C)}_{<0} < T$$

$\Longleftarrow$ )

□

### 5.4.2. Algoritmo de Klein

El *algoritmo de Klein* o *algoritmo de cancelación de ciclos* se basa en el teorema anterior: empieza desde un flujo factible $x$ y, mientras exista un ciclo negativo en su red residual, aumenta el flujo a lo largo de ese ciclo. Esto garantiza que:

- En cada paso se obtiene un flujo de costo estrictamente menor.
- Al terminar, como no existe ningún ciclo negativo en la red residual, el flujo es óptimo.

Puede expresarse de la siguiente manera:

```
KLEIN(N)
1  Encontrar un flujo factible x.
2  while R(N, x) tenga algún ciclo negativo C
3      for each (v → w) ∈ C
4          if v → w ∈ E
5              x(v → w) = x(v → w) + r(C)
6          else
7              x(w → v) = x(w → v) − r(C) return x
```

Para encontrar un flujo inicial factible, se puede aplicar flujo máximo: se define la red $N' = (V', E', u')$, con:

- $V' = V \cup \{s, t\}$
- $E' = E \cup \{s \to v \mid v \in V, b(v) > 0\} \cup \{w \to t \mid w \in V, b(w) < 0\}$
- Las capacidades $u'$ están dadas por:

$$u'(v \to w) = \begin{cases} u(v \to w) & \text{si } v \to w \in E \\ b(w) & \text{si } v = s \\ -b(v) & \text{si } w = t \end{cases}$$

Entonces, un flujo máximo debe saturar todas las capacidades de los arcos de la fuente y el destino. En caso contrario, un flujo factible no es posible en la red inicial.

#### Complejidad

Para calcular la complejidad del algoritmo de cancelación de ciclos, se puede aprovechar el siguiente teorema:

**Teorema.** *Si todos los imbalances y capacidades de una red de flujo pesada $N = (V, E, u, c, b)$, entonces el problema de flujo de costo mínimo tiene una solución óptima entera.*

Si además todos los costos son enteros, siempre se puede encontrar un ciclo negativo de costo entero. Sabiendo que en cada paso el nuevo flujo obtenido tiene un costo estrictamente menor al anterior, se debe cumplir $\bar{T} \leq T - 1$ (por ser todos los valores involucrados enteros).

Como el costo inicial no puede ser más de $|E|C_{\text{máx}}U_{\text{máx}}$ (con $C_{\text{máx}} = \text{máx}\{c(e) \mid e \in E\}$ y $U_{\text{máx}} = \text{máx}\{u(e) \mid e \in E\}$), el costo final es mayor o igual a 0, y en cada iteración el costo actual se reduce en al menos 1 unidad, no se pueden realizar más de $|E|C_{\text{máx}}U_{\text{máx}}$ iteraciones. Dentro de cada iteración, se busca un ciclo negativo ($\mathcal{O}(|V||E|)$ con el algoritmo de Bellman-Ford) y se actualiza el flujo de las aristas de ese ciclo ($\mathcal{O}(|V|)$). Por ende, la complejidad total está acotada por $\mathcal{O}(|V||E|^2 C_{\text{máx}}U_{\text{máx}})$.

#### Mejoras

Una mejora al algoritmo de cancelación consiste de seleccionar en cada paso el ciclo de *costo promedio mínimo* (lo cual se puede realizar en $\mathcal{O}(|V||E|)$). Esto resulta en una complejidad de a lo sumo $\mathcal{O}(\text{mín}\{|V|^2|E|^2 \log(|V|C_{\text{máx}}), |V|^2|E|^3 \log|V|\})$.

### 5.4.3. Relación con otros problemas

El flujo de costo mínimo se puede utilizar para resolver otros problemas. Primero, analicemos una formulación alternativa, pero equivalente, del problema:

**Problema:**

Dada una red de flujo pesada $N = (V, E, u, c)$, un par de vértices $s$ y $t$ y un valor $d$, encontrar un flujo $x$ tal que:

- Tenga valor neto $d$:

$$\sum_{v \in N^+(s)} x(s \to v) - \sum_{v \in N^-(s)} x(v \to s) = d$$

- Respeta las capacidades de las aristas $e \in E$:

$$0 \leq x(e) \leq u(e)$$

- Tiene costo $T = \sum_{e \in E} c(e) \cdot x(e)$ es mínimo.

Es claro que esta formulación se puede reducir a la anterior ($b(s) = d, b(t) = -d$). Para la reducción inversa, se puede conectar cada nodo con imbalance positivo a $s$, y cada uno con imbalance negativo a $t$. Luego, si las capacidades de esas aristas son $|b(v)|$, un flujo de valor $d =$ suma de los valores positivos respetará los imbalances.

#### Camino mínimo

Para resolver el problema de camino mínimo entre dos vértices $s$ y $t$ en un digrafo pesado $G = (V, E, w)$, se puede tomar una red de flujo pesada con los mismos costos que $G$ y capacidades $u(e) = 1\ \forall e \in E$, y buscar el flujo de costo mínimo entre $s$ y $t$ con valor $d = 1$.

**Flujo máximo**

El problema de flujo máximo también se puede reducir a flujo de costo mínimo: se toma la misma red de flujo $N = (V, E, u)$ y se asigna un peso nulo a todas las aristas de $E$ y un imbalance nulo a todos los nodos. Luego, se coloca una arista $t \to s$ de costo $-1$ y capacidad infinita. En tal caso, el flujo (en este caso una circulación) de costo mínimo tendrá un valor de $-F$, ya que $F$ es lo máximo que puede fluir de $s$ a $t$.

**Cotas inferiores**

Una versión más general del problema de flujo de costo mínimo consiste en imponer cotas inferiores $l(v \to w)$ para los flujos de las aristas.

## 5.5. Programación Lineal

### 5.5.1. Definición

Un problema de *programación lineal* es uno en el que se maximiza una combinación lineal de variables sujeta a un conjunto de restricciones. La formulación general es:

**Problema:**
Dados una matriz $A \in \mathbb{R}^{m \times n}$ y vectores $b \in \mathbb{R}^m$ y $c \in \mathbb{R}^n$, encontrar el vector $x^* \in \mathbb{R}^n$ que maximice el producto $c^T x^*$ y respete $Ax^* \leq b$ (la desigualdad es coeficiente a coeficiente). Formalmente:

$$x^* = \arg\max \{c^T x \mid x \in \mathbb{R}^n, Ax \leq b\}$$

Cada inecuación $i$ del sistema $Ax \leq b$ representa la desigualdad:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

Es habitual agregar la restricción de que $x \geq 0$, aunque eso se puede incorporar al sistema $Ax \leq b$.

Esta formulación es sumamente general, ya que permite:

- Minimizar en vez de maximizar, si se toma $-c$ en lugar de $c$, se tiene que:

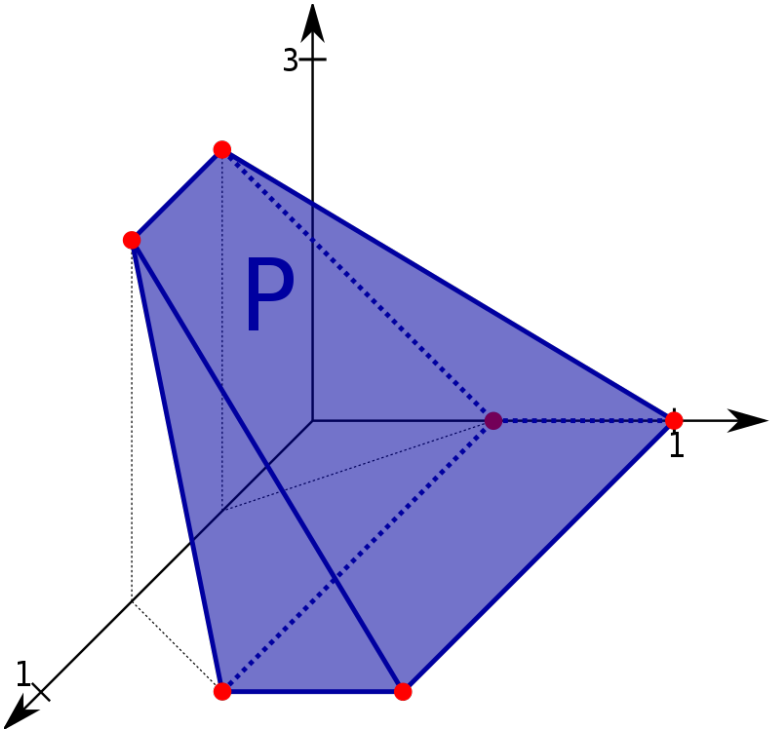
$$-c^T x^* \geq -c^T x \ \forall x \iff c^T x^* \leq c^T x \ \forall x$$

- Agregar cotas inferiores, si se invierten los coeficientes de la fila $i$ de $A$ y $b_i$, se obtiene:

$$-a_{i1}x_1 - a_{i2}x_2 - \cdots - a_{in}x_n \leq -b_i \iff a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \geq b_i$$

- Agregar igualdades, a través de las restricciones:

$$\left.\begin{array}{l} a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i \\ a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \geq b_i \end{array}\right\} \iff a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$




Una ilustración de una posible región factible en $\mathbb{R}^3$

Para una instancia de programación linear (también llamada un *programa lineal*), una *solución factible* es un vector $x$ que respeta $Ax \leq b$ pero no necesariamente maximiza la función objetivo $c^T x$. En base a esto, se puede definir una *región factible*: la sección del espacio vectorial $\mathbb{R}^n$ que contiene a todas las soluciones factibles.

Los bordes de la región factible están dados por los hiperplanos de la ecuación $Ax = b$. También es posible que la región no esté acotada: en cuyo caso puede pasar que el problema no tenga solución, y se pueda tomar valores de la función objetivo tan chicos como se desee.

### 5.5.2. Método Simplex

El *método simplex* es un algoritmo para resolver problemas de programación lineal. Se basa en recorrer los vértices de la región factible (también llamada *simplex*), ya que la solución óptima se debe encontrar en uno de ellos. En cada movimiento que realiza, la función objetivo no crece (e, idealmente, decrece). Como la cantidad de vértices crece exponencialmente con respecto a la cantidad de restricciones, el algoritmo es exponencial, y efectivamente corre lentamente para ciertos casos patológicos. No obstante, este algortimo es muy eficiente, y ampliamente utilizado, en la práctica.

### 5.5.3. Reducciones

Varios de los problemas de grafos que vimos en la materia se pueden reducir a programas lineales

### Flujo de costo mínimo

Las restricciones de flujo de costo mínimo se dan en formato de programa lineal, así que la reducción es directa. Para una red de flujo $N = (V, E, u, b, c)$ Las ecuaciones e inecuaciones son:

$$\begin{aligned} \text{mín} \sum_{vw \in E} c_{vw} x_{vw} & \\ \sum_{w \in N^+(v)} x_{vw} - \sum_{w \in N^-(v)} x_{vw} = b_v & \qquad \forall v \in V \\ x_{vw} \geq l_{vw} & \qquad \forall vw \in E \\ x_{vw} \leq u_{vw} & \qquad \forall vw \in E \end{aligned}$$

### Flujo máximo

Similarmente, se puede reducir el problema de flujo máximo a programación lineal[4]. Esto se logra de la siguiente manera:

$$\begin{aligned} \text{máx} \sum_{v \in N^+(s)} x_{sv} & \\ \sum_{w \in N^+(v)} x_{vw} - \sum_{w \in N^-(v)} x_{vw} = 0 & \qquad \forall v \in V - \{s, t\} \\ x_{vw} \geq 0 & \qquad \forall vw \in E \\ x_{vw} \leq u_{vw} & \qquad \forall vw \in E \end{aligned}$$

### Camino mínimo

Para un grafo pesado $V = (V, E, w)$ y un par de vértices $s, t \in V$, encontrar el camino mínimo entre $s$ y $t$ también se puede expresar como programa lineal, donde la variable $x_{uv}$ indica si una arista se utiliza ($x_{uv} = 1$) o no ($x_{uv} = 0$):

$$\begin{aligned} \text{máx} \sum_{uv \in E} x_{uv} w_{uv} & \\ \sum_{w \in N^-(t)} x_{vw} = 1 \qquad \sum_{w \in N^+(v)} x_{vw} - \sum_{w \in N^-(v)} x_{vw} = 0 & \qquad \forall v \in V - \{s, t\} \end{aligned}$$

Esta reducción es equivalente a la reducción anterior de camino mínimo a flujo de costo mínimo. Las propiedades de la matriz de coeficientes garantizan que hay una solución óptima con $x_{ij} \in \{0, 1\}$.

[4] También se puede reducir "pasando" por flujo de costo mínimo, es decir, reducirlo primero a flujo de costo mínimo y luego a PL.

**Árbol Generador Mínimo**

El problema de encontrar un AGM para un grafo pesado $G = (V, E, w)$ se puede expresar como el siguiente progama lineal:

$$\begin{aligned} &\text{máx} \sum_{uv \in E} x_{uv} w_{uv} \\ \sum_{uv \in E(S)} x_{vw} &\leq |S| - 1 && \forall S \subseteq V \\ \sum_{uv \in E} x_{vw} &= n - 1 && \forall v \in V - \{s, t\} \\ x_{vw} &\geq 0 && \forall vw \in E \end{aligned}$$

También se puede demostrar que esta formulación tiene un óptimo entero. A pesar de que esta formulación tiene una cantidad exponencial de restricciones con respecto al tamaño del grafo original, se puede resolver en tiempo polinomial.

### 5.5.4. P-Completitud

No solo son los problemas anteriores los que se pueden reducir a PL: se puede demostrar que la programación lineal es P-Completo, esto es, cualquier problema resoluble en tiempo polinomial se puede reducir a él. En este sentido, es el problema "más general" entre ellos.

La teoría de clases de complejidad se estudia más en el siguiente capítulo.

### 5.5.5. Dualidad

Para un problema de optimización que implica minimizar un valor, el *problema dual* es uno que busca maximizarlo, y cuya solución es al menos tan grande como cualquier solución factible del primer problema (y viceversa para los problemas de maximización). Esto implica que si se encuentra una solución en la que coinciden, esta es mínima para el problema original.

Para la programación lineal, el problema dual de cualquier instancia es otro programa lineal. Si la instancia es:

$$\text{máx} \{c^T x \mid x \in \mathbb{R}^n, Ax \leq b\}$$

El programa lineal dual es:

$$\text{mín} \{y^T b \mid y \in \mathbb{R}^m, y^T A \geq c\}$$

**Teorema.** *(Teorema de Dualidad Fuerte) Si un problema tiene una solución óptima, entonces el problema dual también la tiene, y sus valores coinciden.*

# Capítulo 6

# NP-Completitud

## 6.1. Problemas

### 6.1.1. Versiones

Dada una instancia $I$ del problema de optimización $\Pi$ con función objetivo $f$, se pueden derivar varias versiones:

- Versión de **optimización**: Encontrar una solución óptima $S^*$ del problema $\Pi$ para $I$ (con $f(S^*) \geq f\ \forall S$ en el caso de maximización y $f(S^*) \leq f\ \forall S$ para la minimización).
- Versión de **evaluación**: Determinar el valor $f(S^*)$ de una solución óptima de $\Pi$ para $I$.
- Versión de **localización**: Dado un número $k$, determinar una solución factible $S$ de $\Pi$ para $I$ tal que $f(S) \leq k$ si el problema es de minimización, o $f(S) \geq k$ si es de maximización.
- Versión de **decisión**: Dado un número $k$, ¿Existe una solución factible $S$ de $\Pi$ para $I$ tal que $f(S) \leq k$ si el problema es de minimización, o $f(S) \geq k$ si es de maximización?

Tomemos el ejemplo del problema de viajante de comercio:

**Problema:**
Dado un grafo pesado $G = (V, E, w)$, encontrar un *circuito hamiltoniano* de peso mínimo, es decir, un circuito simple $C^*$ que pase por todos los vértices y minimice su peso total:

$$C^* = \arg\min \left\{ \sum_{e \in C} w(e) \mid C \text{ es circuito hamiltoniano en } G \right\}$$

Para este problema, las versiones serían:

- Versión de **optimización**: Encontrar un circuito hamiltoniano en $G$ de peso mínimo.
- Versión de **evaluación**: Determinar la longitud mínima que puede tener un circuito hamiltoniano en $G$.
- Versión de **localización**: Dado un número $k$, encontrar un circuito hamiltoniano en $G$ que tenga peso menor o igual a $k$.
- Versión de **decisión**: Dado un número $k$, ¿Existe un circuito hamiltoniano en $G$ que tenga peso menor o igual a $k$?

Para muchos problemas de optimización combinatoria, las cuatro versiones son equivalentes: si una de ellas puede resolverse eficientemente, las demás también. El estudio se realiza en la versión más simple: la de decisión (y hay problemas que no tienen versión de optimización). Estos tienen dos respuestas posibles: **SÍ** o **NO**.

### 6.1.2. Definición

Un problema Π se define dando su entrada y su salida:

> **Problema:**
> Satisfactibilidad (SAT)
>
> Dada una *fórmula proposicional* $f$ en forma normal conjuntiva, ¿Existe una asignación de valores de verdad ($\{True, False\}$) a las proposiciones de $f$ que hace que $f$ sea verdadera?

Una *instancia* $I$ de un problema es una especificación de sus parámetros. En el ejemplo de SAT, sería una fórmula CNF.

Un problema de decisión Π tiene asociado un conjunto $I_\Pi$ de instancias, y un subconjunto de ellas $Y_\Pi \subseteq I_\Pi$ cuya respuesta es **SÍ**.

## 6.2. Máquinas de Turing

### 6.2.1. Máquina de Turing Determinística

Una *máquina de Turing determinística* (MTD) es un modelo de cómputo simple. La máquina cuenta con:

- Una *cabeza lecto-escritora*, que se puede mover 1 unidad en ambas direcciones (Definido como $M = \{+1, -1\}$).
- Una *cinta infinita* en ambas direcciones con celdas indexadas en $\mathbb{Z}$.
- Una celda inicial con índice 0.

Esas son las características generales de cualquier MTD. Para una máquina particular, se tiene:

- Un *alfabeto* de *símbolos* finito $\Sigma$ y un símbolo especial $*$ llamado "*blanco*". Se define $\Gamma = \Sigma \cup \{*\}$. La celda puede tomar valores en $\Gamma$.
- Un conjunto finito $Q$ de *estados*.
- Un *estado inicial* $q_0 \in Q$.
- Un conjunto de *estados finales* $Q_f \subseteq Q$ (en el caso de problemas de optimización $Q_f = \{q_{\text{sí}}, q_{\text{no}}\}$).
- Un *programa*, definido como un conjunto finito de *instrucciones*. Estas, a su vez, están definidas como quíntuplas $S \in Q \times \Gamma \times Q \times \Gamma \times M$, y una instrucción $(q, s, q', s', m)$ se interpreta como: "Si la máquina está en el estado $q$ y la cabeza lee el símbolo $s$, entonces escribe $s'$ en esa celda, realiza el movimiento $m$, y pasa al estado $q'$".

La entrada al programa se carga utilizando los símbolos de $\Sigma$, y las demás celdas en la cinta empiezan con el valor $*$.

El modelo es capaz de simular cualquier otra máquina Turing-equivalente, lo cual incluye a la máquina de RAM analizada anteriormente.

#### Determinismo

Las MTDs son *determinísticas*: para todo par $(q, s) \in Q \times \Gamma$, existe en el programa a lo sumo una quíntupla que comienza con ese prefijo. Esto implica que, en cada paso, no hay ambigüedad respecto a qué hacer (porque solo una instrucción aplica).

#### Resolución y Complejidad

La máquina *resuelve* el problema $\Pi$ si para toda instancia $I \in I_\Pi$ esta alcanza un estado final y es el correcto.

La *complejidad* de una MTD está dada por la cantidad de movimientos de la cabeza que se realizan entre el estado inicial y el final, en función del tamaño de entrada:

$$T_M(n) = \text{máx}\{m \mid M \text{ realiza } m \text{ movimientos para la entrada } I \in I_{@P}, |I| = n\}$$

Una MTD $M$ es *polinomial* para $\Pi$ cuando $T_M(n) \in \mathcal{O}(n^k)$ para algún $k$.

### 6.2.2. La clase P

Un problema de decisión $\Pi$ pertenece a la clase P si existe una MTD polinomial que lo resuelve. Esto equivalente a que exista un algoritmo polinomial (en la máquina RAM) que resuelve $\Pi$. Hay problemas (como la programación lineal) que son P-completos: cualquier problema de P se puede reducir a una instancia de ellos

### 6.2.3. Máquina de Turing no Determinística

Una máquina de Turing no determinística (MTND) es una máquina de Turing en la que no se requiere unicidad para el prefijo $(q, s)$ con el que empieza cada instrucción. En el caso de estar en un estado y símbolo donde múltiples instrucciones aplican, la máquina elige cualquiera de ellas (no está especificado cuál).

**Resolución y Complejidad**

Una MTND resuelve el problema de decisión Π cuando existe una secuencia de ejecución de instrucciones que lleva a un estado de aceptación si y solo si la respuesta correcta es sí. La ejecución se puede interpretar como un *árbol de alternativas*, donde cada nodo representa un punto en el que se puede "elegir" entre 1 o más instrucciones válidas.

Una definición equivalente sería que para toda instancia de $Y_\Pi$ existe **al menos una** rama de ejecución que termina en $q_{\text{sí}}$, mientras que para toda instancia de $I_\Pi - Y_\Pi$, ninguna lo hace.

Por otro lado, la complejidad temporal deuna MTND $M$ se define como el máximo número de pasos que toma la rama de ejecución más corta en llegar a $q_{\text{sí}}$ para una instancia de $Y_\Pi$ en función de su tamaño:

$$T_M(n) = \text{máx}\{m \mid \text{La rama más corta de } M \text{ termina en } m \text{ movimientos para } x \in Y_\Pi, |x| = n\}$$

Una MTND $M$ es polinomial para Π cuando $T_M(n) \in \mathcal{O}(n^k)$ para algún $k$.

### 6.2.4. La clase NP

Un problema de decisión Π pertenece a la clase NP (polinomial no-detereminístico) cuando existe una MTND polinomial que lo resuelve.

Una definición equivalente es que, dada una instancia $I \in Y_\Pi$, se puede dar un *certificado* de longitud polinomial (con respecto al tamaño de $I$) que garantiza que la respuesta es **SÍ**, y esto se puede verificar en tiempo polinomial.

Una observación es que $P \subseteq NP$: para un problema polinomial, una instancia $I \in Y_\Pi$ es su propio certificado, ya que tiene tamaño lineal con respecto a sí misma y se puede verificar aplicando el algoritmo que resuelve el problema que, por definición, es polinomial.

**Teorema.** *Si* Π *es un problema de decisión que pertenece a la clase NP, entonces* Π *puede ser resuelto por un algoritmo determinístico en tiempo exponencial con respecto al tamaño de la entrada.*

*Demostración.* Para resolver Π, se puede simular la ejecución de la MTND que lo resuelve: cada vez que hay ambigüedad con respecto a la instrucción a aplicar se exploran todos los "súbarboles de ejecución". Esto implica una complejidad exponencial: si hay a lo sumo $A$ estados distintos para cada punto de no-determinismo, se realizan $\mathcal{O}(A^{T_M(n)})$, y $T_M(n)$ es una función exponencial (porque Π ∈ NP). □

**P vs. NP**

El problema abierto más famoso en la ciencia de la computación es el de P vs. NP, esto es, es la clase P la misma que NP. Se sabe que $P \subseteq NP$, así que lo que se demostraría es si $NP \subseteq P$ o $NP \subsetneq P$. Si P fuera igual a NP, entonces cualquier problema que se pueda certificar en tiempo polinomial se podría resolver también en tiempo polinomial.

### 6.2.5. Reducciones Polinomiales

Una *transformación* o *reducción polinomial* de un problema de decisión $\Pi'$ a otro $\Pi$ es una función $f : I_{\Pi'} \longrightarrow I_{\Pi}$ que puede ser computada en tiempo polinomial y cumple que:

$$I' \in Y_{\Pi'} \iff f(I') \in Y_{\Pi} \forall I' \in I_{\Pi'}$$

Es decir, una instancia de $\Pi'$ tiene respuesta sí si y solo si la instancia correspondiente $f(I')$ tiene respuesta sí.

Se dice que un problema de decisión $\Pi'$ se *reduce polinomialmente* a otro $\Pi$ cuando existe una reducción polinomial entre $\Pi'$ y $\Pi$. Se denota $\Pi' \leq_p \Pi$.

Las reducciones polinomiales son **transitivas**: si $\Pi_1 \leq_p \Pi_2$ y $\Pi_2 \leq_p \Pi_3$ entonces $\Pi_1 \leq_p \Pi_3$. Esto es porque, si las reducciones correspondientes son $f_{12}$ y $f_{23}$, se puede construir una función $f_{13}(I) = f_{23}(f_{12}(I))$.

### 6.2.6. La clase NP-completo

Un problema de decisión $\Pi$ es *NP-completo* cuando:

1. $\Pi \in NP$
2. $\forall \bar{\Pi} \in NP, \bar{\Pi} \leq_p \Pi$

Si un problema cumple la condición 2, se lo llama *NP-difícil*.

**Teorema.** *(Teorema de Cook-Levin) SAT es NP-Completo.*

*Demostración.* Sabemos que SAT $\in NP$: para certificar una instancia $f \in Y_{\text{SAT}}$, se puede tomar la asignación de valores de verdad a las proposiciones de $f$. Tanto el tamaño de este certificado como verificarlo (que implica evaluar la fórmula en con los valores) son lineales.

Para ver que es NP-hard, tomemos un problema cualquiera $\Pi \in NP$. Como está en NP, hay una MTND que lo resuelve. Llamamos:

- $\Gamma = \Sigma \cup \{*\}$ a su alfabeto.
- $Q$ a su conjunto de estados.
- $s \in Q$ al estado inicial.
- $F \subseteq Q$ al conjunto de estados finales.
- $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times M$ al conjunto de instrucciones.

Asumimos $M = \{+1, 0, -1\}$ (el cabezal se puede quedar quieto).

Reducimos $\Pi$ a SAT de la siguiente manera: dado un input $I$ de $\Pi$, construimos una fórmula proposicional $f$ tal que $f$ es satisfactible si y solo si $I \in Y_{\Pi}$. Sea $p(n)$ la función de complejidad de la MTND. La fórmula contiene las siguientes propocisiones:

- $T_{ijk} \equiv$ "la celda $i$ contiene el símbolo $j$ en el paso $k$ de la ejecución de la MTND"
- $H_{ik} \equiv$ "el cabezal de la MTND está ubicado en la celda $i$ en el paso $k$"
- $Q_{qk} \equiv$ "la MTND está en estado $q$ en el paso $k$"

Para $i = \{-p(n), ..., p(n)\}$ (el cabezal no se puede mover más que la cantidad máxima de pasos), $k = \{0, ..., p(n)\}$, $j \in \Gamma$ y $q \in Q$. Además, llamamos $I = (j_0, ..., j_{n-1})$ a la entrada, que empieza en la celda 0.

Las cláusulas de la fórmula son las siguientes:

- $T_{ij_i0}$: la celda $i$ contiene el valor $j_i$ en tiempo 0, para $i = 0, ..., n-1$.
- $T_{i*0}$: la celda $i$ contiene el valor blanco en tiempo 0, para $i \in \{-p(n), ..., p(n)\} - \{0, ..., n-1\}$
- $H_{00}$: el cabezal comienza en la celda 0.
- $Q_{s0}$: la máquina comienza en el estado $s$.
- $\neg(T_{ijk} \wedge T_{ij'k})$: la celda $i$ contiene a lo sumo un símbolo en el paso $k$ ($j \neq j'$).
- $\bigvee_{j \in \Gamma} T_{ijk}$: la celda $i$ contiene al menos un símbolo en el paso $k$.
- $T_{ijk} \wedge T_{ij'(k+1)} \implies H_{ik}$: las celdas no apuntadas por el cabezal no cambian ($j \neq j', k < p(n)$).
- $\neq (Q_{qk} \wedge Q_{q'k})$: la máquina está en a lo sumo un estado en el paso $k$ ($q \neq q'$).
- $\neq (H_{ik} \wedge H_{i'k})$: el cabezal apunta a lo sumo a una celda en el celda paso $k$ ($i \neq i'$).
- $H_{ik} \wedge Q_{qk} \wedge T_{i\sigma k} \implies \bigvee_{(q,\sigma,q',\sigma',m) \in \Delta} (H_{(i+m)(k+1)} \wedge Q_{q'(k+1)} \wedge T_{i\sigma(k+1)})$: en cada paso $k < p(n)$, se da una de las transiciones válidas.
- $\bigvee_{k=0}^{p(n)} \bigvee_{f \in F} Q_{fk}$: la máquina termina en un estado final.

Si hay un cómputo de la MTND con $I$ que termina en un estado $F$, entonces $f$ es satisfacible asignando las proposiciones correspondientes a la ejecución. Recíprocamente, si $f$ es satisfacible, entonces existe un cómputo de la MTND que, a partir de $I$ y siguiendo los pasos especificados por las proposiciones verdaderas, llega a un estado de $F$. Como la fórmula tiene $\mathcal{O}(p(n)^2)$ proposiciones y $\mathcal{O}(n^3)$ cláusulas, esta reducción es polinomial.

Dado que esta reducción es posible para cualquier problema $\Pi \in NP$, SAT es NP-completo. □

Para demostrar la NP-completitud de otros problemas $\Pi' \in NP$, se puede reducir problemas conocidos NP-completos a ellos: si $\Pi$ es un problema NP-completo, entonces

$$\Pi \leq_p \Pi' \implies \forall \bar{\Pi} \in NP, \bar{\pi} \leq_p \Pi'$$

Esto vale por la transitividad de las reducciones polinomiales.

Para demostrar el caso $P = NP$, bastaría con encontrar un problema en $NP$-completo $\cap P$, ya que los demás problemas de NP podrían ser reducidos a ese problema y resueltos, todo en tiempo polinomial.

## 6.3. Problemas NP-completos

Como demuestra el teorema de Cook-Levin, SAT es NP-completo. Este hecho puede ser utilizado para demostrar la NP-completitud de múltiples problemas: en 1972, Karp demostró que otros 21 problemas son NP-completos. En la actualidad se conocen más de 3000 problemas problemas NP-completos.

### 6.3.1. 3-SAT

3-SAT es una versión restringida de SAT, donde las fórmulas CNF $f$ tienen exactamente 3 literales. A pesar de ser aparentemente menos general, este problema también es NP-completo.

**Teorema.** *3-SAT es NP-completo*

*Demostración.* Para demostrar que está en NP, se puede tomar el mismo certificado y verificador que para SAT: la asignación de valores que hace a $f$ verdadera.

Para demostrar que es NP-hard, se puede reducir SAT a 3-SAT. Esto se logra tomando cada cláusula $(p_{i1} \vee \cdots \vee p_{ik})$ y agregando distintas cláusulas dependiendo de su tamaño:

- Si $k = 1$, la cláusula tiene una sola proposición: $p_{i1}$. En ese caso se puede agregar el conjunto de cláusulas

$$\begin{aligned}(p_{i1} \vee q_{i1} \vee q_{i2}) \wedge \\ (p_{i1} \vee \neg q_{i1} \vee q_{i2}) \wedge \\ (p_{i1} \vee q_{i1} \vee \neg q_{i2}) \wedge \\ (p_{i1} \vee \neg q_{i1} \vee \neg q_{i2})\end{aligned}$$

  Se puede ver que la única forma de que este grupo de cláusulas sea verdadero es que $p_{i1}$, ya que cualquier asignación de los parámetros $q_{i1}, q_{i2}$ hace que alguna de ellas sea falsa.

- Si $k = 2$, la cláusula tiene 2 proposiciones: $(p_{i1} \vee p_{i2})$. En su lugar, se agregan dos cláusulas:

$$(p_{i1} \vee p_{i2} \vee q_i) \wedge (p_{i1} \vee p_{i2} \vee \neg q_i)$$

  Este par de cláusulas es verdadero si y solo si la cláusula original es verdadera: el valor de la variable $q_i$ no afecta esto.

- Si $k = 3$, se pone la misma cláusula en la fórmula.

- Si $k > 3$, se puede usar el siguiente conjunto de cláusulas:

$$\begin{aligned} & (p_{i1} \vee p_{i2} \vee q_{i1}) \\ \wedge\ & (\neg q_{i1} \vee p_{i3} \vee q_{i2}) \\ & \vdots \\ \wedge\ & (\neg q_{i(j-2)} \vee p_{ij} \vee q_{i(k-1)}) \\ & \vdots \\ \wedge\ & (\neg q_{i(k-3)} \vee p_{i(k-1)} \vee q_{ik}) \end{aligned}$$

Este conjunto de cláusulas es verdadero si y solo si $(p_{i1} \vee \cdots \vee p_{ik})$ es verdadero.

Como cada componente de la nueva conjunción $f'$ es satisfactible si y solo si el término correspondiente de $f$ lo es, $f \iff f'$. Además, el tamaño de la nueva fórmula es a lo sumo 3[1] veces más grande que la anterior, así que la reducción es polinomial. Queda demostrado $3 - \text{SAT} \leq_p \text{SAT}$.

□

### 6.3.2. Clique máxima

**Problema:**
CLIQUE

**Entrada**: Un grafo $G = (V, E)$
**Salida**: Un subgrafo inducido completo (también llamado clique) $K_m \subseteq G$ tal que

$$|K_m| \geq |S| \ \forall S \subseteq G, S \text{ es clique en } G$$

La versión de decisión del problema de clique máxima es determinar si existe alguna clique en $G$ con tamaño mayor o igual $k$ para un $k$ dado.

**Teorema.** CLIQUE *es NP-completo*

*Demostración.* □

## 6.4. Restricción y Extensión

### 6.4.1. Definición

El problema $\Pi$ es una *restricción* de otro problema $\bar{\Pi}$ si el dominio de $\Pi$ está incluido en el dominio de $\bar{\Pi}$, y la salida de ambos problemas es la misma (es decir, son el mismo problema, pero $\Pi$ tiene una entrada restringida). Cuando sucede esto, se dice que $\bar{\Pi}$ es una *extensión* o *generalización* de $\Pi$.

[1] No estoy seguro

Es habitual que una restricción de un problema de NP-completo esté en P, pero que la situación recíproca no pueda suceder (a menos que P = NP).

### 6.4.2. Ejemplos

- Como vimos en la sección anterior, 3-SAT es una restricción de SAT, pero ambos problemas son NP-completos. Por otro lado, 2-SAT (también una restricción de SAT) es polinomial.
- Coloreo es un problema NP-completo, y lo sigue siendo para:
  - Grafos arco-circulares.
  - Grafos que no contienen a $P_5$ como subgrafo inducido.
  - Grafos planares.
  - Grafos sin triángulos.

  Sin embargo, se vuelve polinomial si se restringe a:
  - Grafos arco-circulares propios.
  - Cografos (grafos sin $P_4$ inducido).
  - Grafos de intervalos.
  - Grafos cordales.
  - Grafos perfectos.
  - Grafos sin subgrafos inducidos $K_{1,3}$.
- $k$-Clique, una restricción de Clique en la cual el tamaño de la clique es fijo, es polinomial para cualquier $k$. Esto se debe a que se pueden ennumerar todas las combinaciones de vértices de tamaño $k$ en tiempo $\mathcal{O}(n^k)$ (lo cual, para $k$ fijo, es polinomial) y verificar si son cliques en tiempo cuadrático. Esto funciona porque cualquier grafo que tiene una clique de tamaño mayor a $k$ tiene otra(s) de tamaño exactamente $k$ como subgrafo de esta. El resultado implica que el problema Clique en grafos planares es polinomial, porque los grafos planares no pueden tener a $K_5$ como subgrafo (así que se puede reducir a 4-Clique).

## 6.5. La clase co-NP

### 6.5.1. Problema complemento

El *problema complemento* de un problema de decisión $\Pi$, $\Pi^c$, es un problema de decisión con el mismo conjunto de instancias, pero cuya respuesta es **SÍ** para todas las instancias **NO** de $\Pi$, y viceversa. Es decir, $\Pi^c$ cumple:

- $I_{\Pi^c} = I_\Pi$
- $Y_{\Pi^c} = I_\Pi - Y_\Pi$

Un ejemplo posible sería TAUTOLOGY: Dada una fórmula $f$ en DNF, es $f$ una tautología. Esto es cierto si y solo si $\neg f$, una fórmula CNF, es una contradicción, es decir, no se puede satisfacer, así que el problema responde **SÍ** para las instancias de SAT que no son satisfactibles.

**Teorema.** *Si un problema* $\Pi$ *está en* $P$*, entonces* $\Pi^c \in P$.

*Demostración.* Si existe un algoritmo polinomial para resolver $\Pi$ en tiempo polinomial, la respuesta de ese algoritmo se puede invertir, obteniendo un algoritmo polinomial para $\Pi^c$. □

Este argumento no sirve para la clase NP: que $\Pi$ esté en NP significa que hay un certificado y verificador polinomiales para las instancias de $Y_\Pi$ (no necesariamente para las de $Y_{\Pi^c} = I_\Pi - Y_\Pi$).

### 6.5.2. co-NP

La clase co-NP está definida como aquellos problemas cuyo complemento está en NP, es decir:

$$\Pi \in NP \iff \Pi^c \in co - NP$$

Una definición alternativa sería que son aquellos problemas donde existen certificado y verificador polinomiales para instancias de respuesta **NO**. El problema TAUTOLOGY, mencionado anteriormente, es un ejemplo de un miembro de esa clase y, más aún, es co-NP-completo: cualquier problema de co-NP se puede reducir polinomialmente a este.

**Teorema.** *Si* $\Pi \in$ *NP-completo,* $\Pi^c \in$ *co-NP-completo*

*Demostración.* Si $\Pi$ es NP-completo existe, para cualquier problema $\bar{\Pi} \in$ NP, una reducción polinomial a $\Pi$. Esta misma reducción se puede aplicar entre problemas de co-NP y $\Pi^c$. □