

DIJKSTRA

Es un **algoritmo** para **encontrar el camino más corto desde un nodo origen a todos los demás nodos** (o hasta un nodo) en un **grafo** (dirigido o no dirigido) **ponderado con pesos no negativos** (si hay pesos negativos, Dijkstra no funciona correctamente). Por ejemplo, en un GPS, Dijkstra puede decirte cuál es la ruta más corta desde tu ubicación a todos los puntos posibles.

💡 El algoritmo de Dijkstra es bastante parecido al de Bellman-Ford con la diferencia que en lugar de usar una cola, utilizamos una cola de prioridad, donde los primero elementos que sacamos son los que tienen menor distancia hacia el nodo inicial. Y además todos los pesos son positivos, sino debemos usar Bellman-Ford.

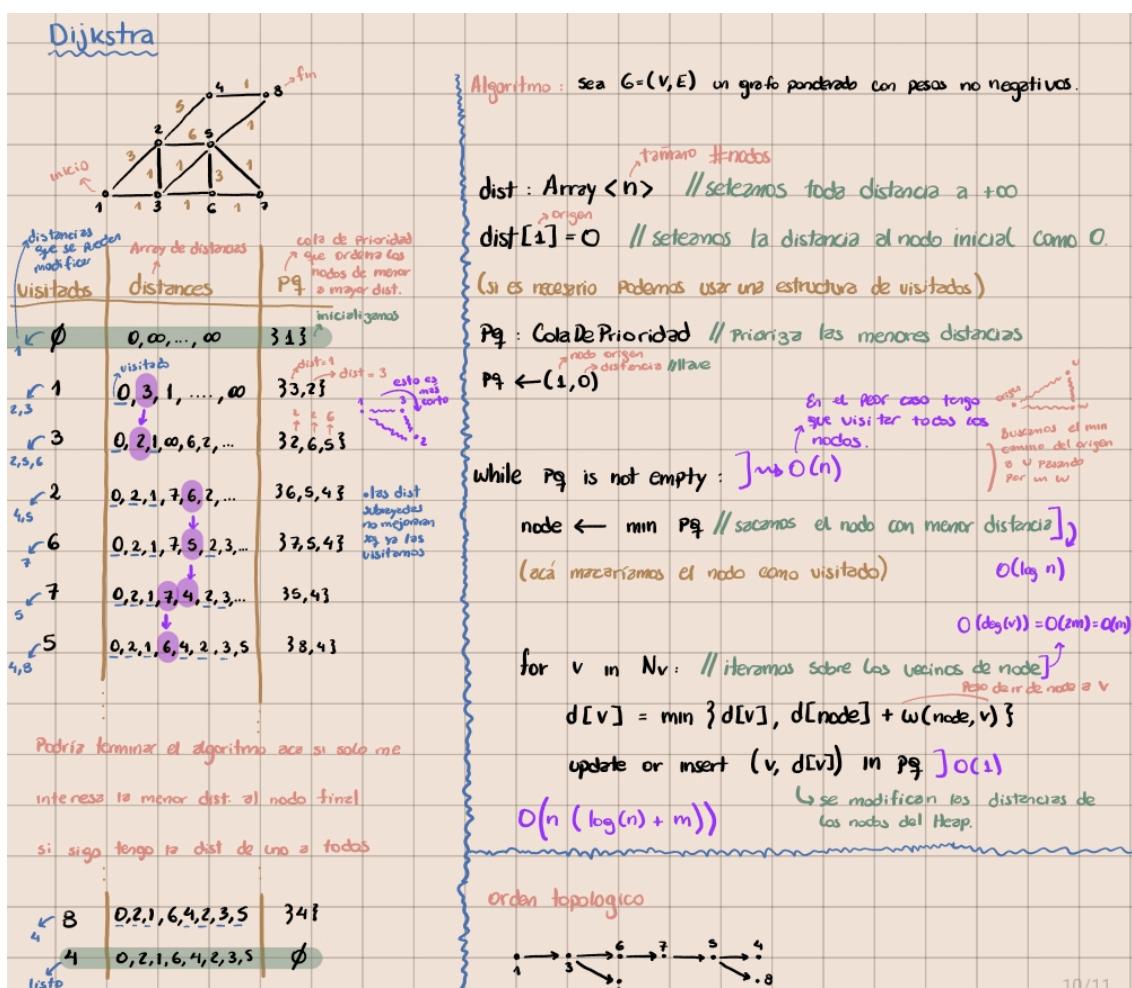
<https://www.datacamp.com/es/tutorial/dijkstra-algorithm-in-python>

▶ Dijkstra's algorithm in 3 minutes ▶ Dijkstras Shortest Path Algorithm Explained | With Example | Graph Th...

ALGORITMO

Estando parado en un nodo (el origen), si queremos visitar todos los nodos vecinos de la forma más barata posible, debemos seguir los siguientes pasos:

- Inicialización** → Se asigna una distancia inicial infinita a todos los nodos del grafo, excepto al nodo inicial, que se asigna una distancia de 0.
- Iteración** → El algoritmo selecciona el nodo no visitado con la distancia más pequeña y lo marca como visitado. Luego, revisa las aristas adyacentes a ese nodo y actualiza la distancia de los nodos vecinos si se encuentra un camino más corto. Desde ese nodo, se intenta relajar las distancias a sus vecinos (ver si hay un camino más corto a través de ese nodo).
- Repetición** → Se repite el paso 2 hasta que todos los nodos hayan sido visitados.



💡 El profe dijo de usar un fibo-heap como cola de prioridad, pero podría ser un min-Heap (pero cambia la complejidad).

Observación:

Si v_1 se visita antes de v_2 , entonces $d(v_1) \leq d(v_2)$ y si visito v_2 justo después de v_1 , entonces $d(v_1) \leq d(v_2)$.

• Al no haber pesos negativos :
 PASO 1 ~~~ tomo $d(v_1) \leq d(v_2)$ la menor dist.
 PASO 2 ~~~ $d(v_2) = \min \{ d(v_2), d(v_1) + w(v_1, v_2) \} \Rightarrow d(v_1) \leq d(v_2)$

- Si los pesos son enteros y acotados podemos usar el algoritmo dial usando buckets, y mejora la complejidad

Fibo-Heap

Un **Fibonacci Heap** es una estructura de datos avanzada para colas de prioridad. Es más compleja que un binario heap, pero permite algunas operaciones en **tiempo amortizado** mucho más rápido.

Operación	Binary Heap	Fibonacci Heap
Insertar elemento	$O(\log n)$	$O(1)$ amortizado
Extraer mínimo (pop min)	$O(\log n)$	$O(\log n)$ amortizado
Disminuir clave (decrease-key)	$O(\log n)$	$O(1)$ amortizado

💡 ¿Vale la pena usar Fibonacci Heap? Teóricamente: sí, mejora el tiempo en grafos densos. Pero en la práctica, no siempre, porque es más complejo de implementar y tiene constantes ocultas grandes.

ST-EFICIENTES

En un grafo ponderado (dirigido o no), dada una fuente s y un destino t , una arista ($v \rightarrow w$) se dice **st-eficiente**, si forma parte de algún camino de longitud mínima (camino más corto) entre el nodo s y el nodo t .

Si estás buscando el camino más corto desde s a t , por ejemplo con BFS (si el grafo no tiene pesos) o con Dijkstra (si tiene pesos), una arista st-eficiente es una de las **aristas que puede aparecer en ese camino mínimo**.



💡 Si existen múltiples caminos mínimos pueden haber múltiples aristas st-eficientes.

BFS & DFS

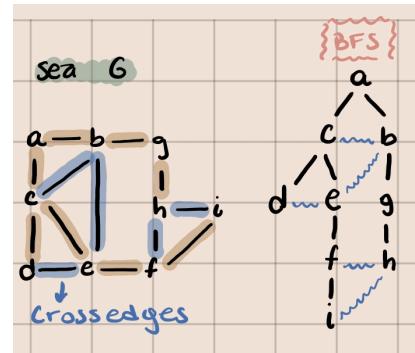
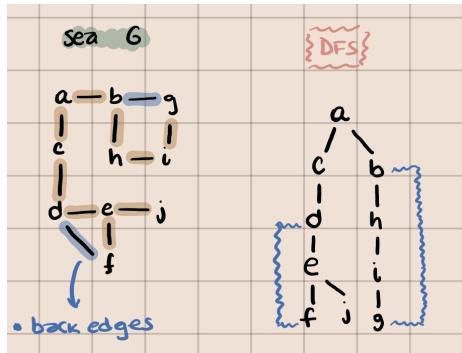
A la hora de construir árboles usando los algoritmos BFS y DFS, existen dos conceptos importantes dentro de la aplicación de estos algoritmos que son back edges (en DFS) y crossedges (en BFS).

Cuando corrés DFS sobre un grafo, generás un **árbol** (o bosque si hay distintas componentes conexas) de expansión, y clasificás todas las aristas del grafo original según cómo aparecen durante el recorrido DFS.

- **Tree Edges** → Arista descubierta por primera vez, va del nodo actual a un nuevo nodo.
- **Back Edges** → Arista que conecta un nodo con uno de sus **ancestros** en el árbol DFS.
- **Forward Edges** → Va de un nodo a un descendiente (ya visitado). Solo en grafos dirigidos.

Cuando corrés BFS sobre un grafo, generas un **árbol** (o bosque si hay distintas componentes conexas) que se recorre por niveles.

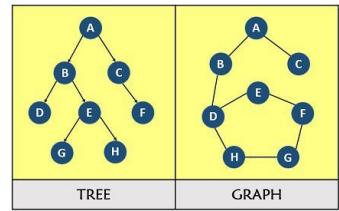
- **Tree Edges** → Arista descubierta por primera vez, va del nodo actual a un nuevo nodo.
- **Cross Edges** → Conecta dos nodos en el mismo nivel o en niveles consecutivos. Además puede conectar dos nodos en diferentes ramas del árbol cumpliendo la diferencia de a lo sumo 1 nivel.



GRAFO ÁRBOL

Un árbol es un grafo que cumple las siguientes propiedades:

- **Conexo** → Existe un camino entre cualquier par de vértices.
- **Acíclico** → No contiene ciclos.
- **Número de aristas** → Tiene exactamente $n - 1$ aristas, donde n es el número de vértices.

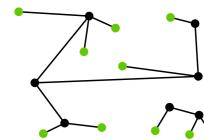


💡 Si $m_G > n_G - 1$, con m la cantidad de aristas y n la cantidad de nodos del grafo G , entonces G tiene un ciclo.

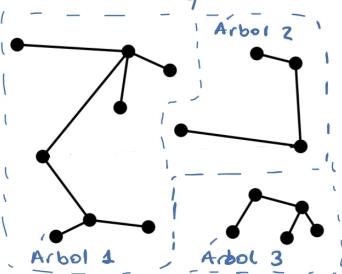
💡 Si G es conexo, entonces $m_G \geq n_G - 1$

💡 En inducciones sobre árboles solemos sacar las hojas como vértices, ya que no rompen el árbol.

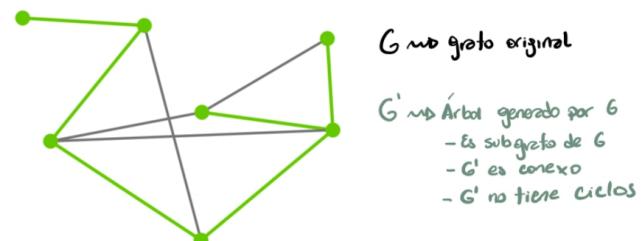
Definimos las **hojas** en un grafo como los **vértices de grado 1**.



G es un bosque



Definimos un **bosque** como un grafo acíclico, o sea, la unión disjunta de árboles.



Definimos un **árbol generador de un grafo G** como un subgrafo generador de G que es un árbol, es decir, el **subgrafo G' contiene todos los vértices del grafo original, y además, G' es un árbol (G' es conexo y G' no tiene ciclos)**

💡 Si el grafo no es conexo, entonces no admite un árbol generador.

Teorema: Un grafo es conexo si y solo si admite un árbol generador.

EQUIVALENCIAS DE ÁRBOL

EQUIVALENCIAS :

1. Un grafo T se dice **árbol** si es **conexo** y **no tiene ciclos**.

existe un único

2. $\forall v, w \in T \quad \exists!$ **camino simple** $p = (v, \dots, w)$ (T es conexo)

Todo par de vértices de T está unido por un único camino

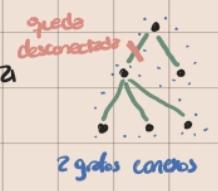
3. T **acíclico** y $m = n - 1 \rightarrow$ **cualquier arista que agregue**

sin ciclos

forma un ciclo



4. T **conexo** y $Ae \in E$, $G' = (V, E - \{e\})$ es **disconexa**



5. T **conexo** y $m = n - 1$

2 grafos conexos

Teorema

Son equivalentes:

1. G es un árbol.
2. G es un grafo sin ciclos, pero si se agrega una arista e a G resulta un grafo con exactamente un ciclo, y ese ciclo contiene a e (es acíclico *maximal* respecto de aristas).
3. G es conexo, pero si se quita cualquier arista a G queda un grafo no conexo (es conexo *minimal* respecto de aristas).



"maximal" y "minimal" se refieren a elementos que son "más grandes" o "más pequeños" en relación a los demás elementos del conjunto, pero no necesariamente son el mayor o menor valor absoluto del conjunto.

Las demostraciones están en las diapositivas de la teórica de la clase 12/05.

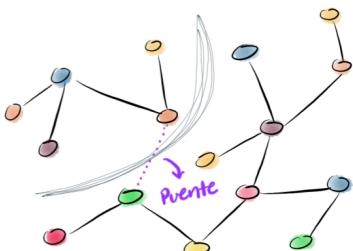
Teorema: Todo árbol no trivial (o sea con 2 o más nodos) tiene al menos dos hojas.

Esto sale por inducción, dado dos nodos y una arista, si le agregamos un nodo conectado a una punta finitas veces, llegamos a que por lo menos tiene dos hojas.

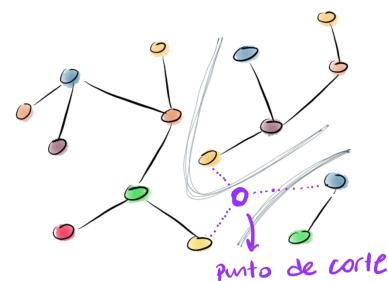
De este teorema también podemos deducir que al menos dos vértices (las hojas), tales que al sacar alguno de ellos, sigue siendo conexo.



Teorema: Toda arista de un árbol es un puente. Deja a lo sumo 2 componentes conexos.



Teorema : Un vértice de un árbol no trivial es un punto de corte si y sólo si no es una hoja. Deja al menos 2 componentes conexos.



Lema

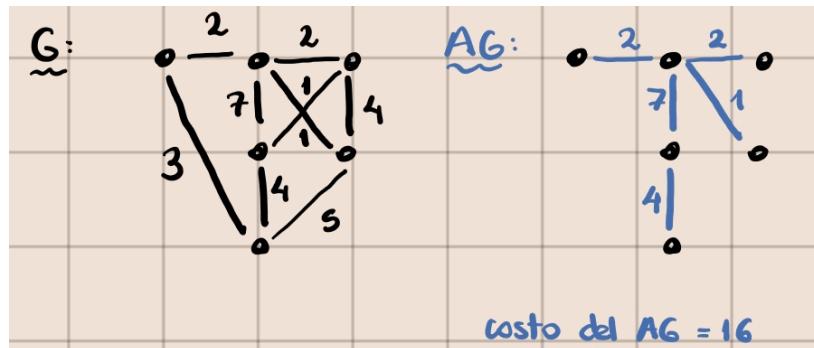
Sea $G = (V, E)$ un grafo conexo y $e \in E$. $G - e$ es conexo si y sólo si e pertenece a un ciclo de G .

AG (ÁRBOL GENERADOR)

Sea G un grafo. Decimos que T es un **árbol generador de G** si se cumplen las siguientes condiciones:

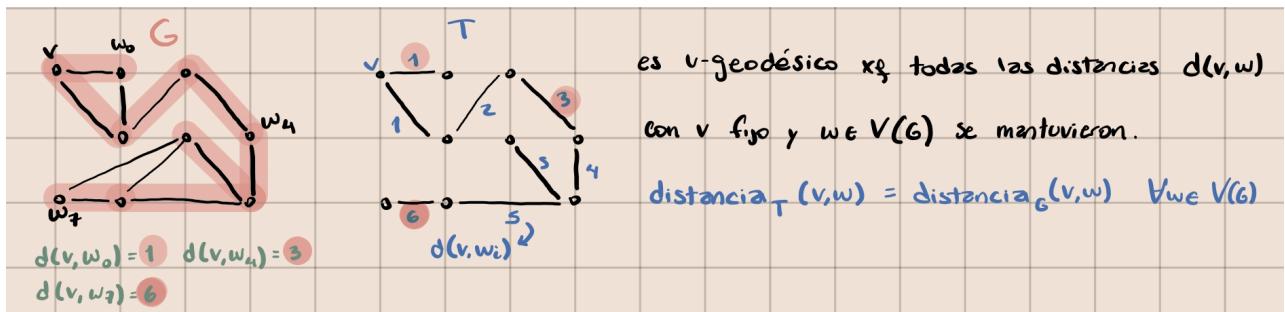
- T es **subgrafo generador de G** (tienen los mismos nodos y las aristas de $T \subseteq G$)
- T es un **árbol**

Costo de un AG → Sea G un grafo ponderado. Si T es un árbol generador de G , definimos el costo de T como la suma de los pesos de las aristas de T .



V-Geodésico

Un árbol generador T de un grafo G es v -geodésico si la distancia entre v y w en T es igual a la distancia entre v y w en G para todo $w \in V(G)$.



Camino Maximin y Minimax

Estas definiciones se usan típicamente en grafos ponderados y se refieren a caminos entre dos vértices donde se quiere optimizar el peso de las aristas más “críticas” del camino.

- **Camino Minimax** → Queremos ir de un nodo u a v , y entre todos los caminos posibles entre ellos, elegimos el que minimiza la arista más pesada del camino. En todos los caminos entre u y v , tomás el más “seguro”, es decir, aquel donde la arista más pesada sea lo más pequeña posible.

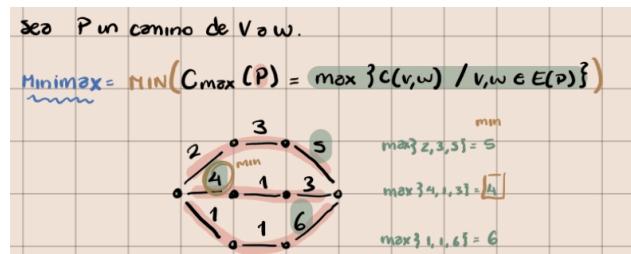
Si entre A y D tenés tres caminos:

A-B-D: pesos 3, 8 → máx = 8, peso total = 11

A-C-D: pesos 4, 5 → máx = 5, peso total = 9

A-E-D: pesos 2, 6 → máx = 6, peso total = 8

El camino minimax es A-C-D, porque su arista más pesada (5) es la mínima de los máximos.



- **Camino Maximin** → Queremos el **camino donde la arista de menor peso sea lo más grande posible**. De

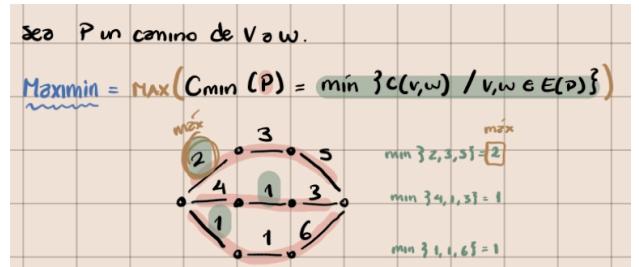
Supongamos tres caminos de A a D:

A-B-D: pesos 3, 8 → mín = 3, peso total = 11

A-C-D: pesos 4, 5 → mín = 4, peso total = 9

A-E-D: pesos 2, 6 → mín = 2, peso total = 8

El camino maximin es A-C-D, porque su arista más débil (4) es la mayor entre las más débiles de todos los caminos.

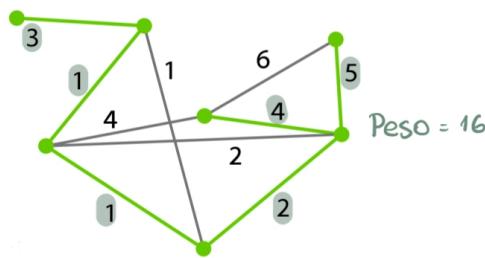


Aplicaciones típicas:

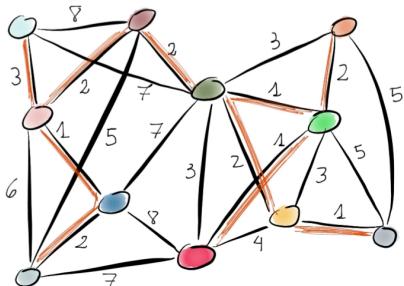
- Minimax: planificación en redes donde hay que evitar cuellos de botella (por ejemplo, logística, transporte).
- Maximin: rutas más seguras y robustas, muy usado en redes de comunicación o diseño de caminos alternativos.

MST (ÁRBOL GENERADOR MÍNIMO)

En un grafo pesado (grafo cuyas aristas tienen asociados un peso) definimos el **peso de un subgrafo** como la suma de los pesos de sus aristas.



Llamamos **árbol generador mínimo** en un grafo pesado, al árbol generador de peso mínimo.



* Conecta todos los vértices del gráfico

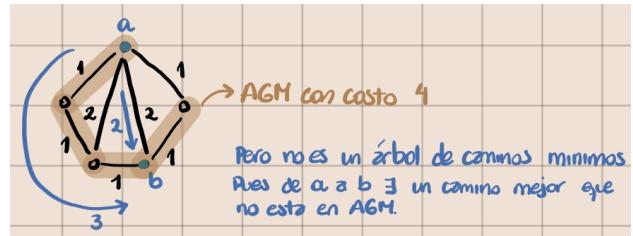
* No contiene ciclos

* Tiene peso total mínimo

$$\min \left\{ \sum_{e \in T} w(e) : T \text{ es un árbol generador} \right\}$$

El MST no es único si hay aristas con pesos iguales.

MST y Floyd-Warshall no siempre coinciden.



Los árboles generadores mínimos se construyen usando **algoritmos greedy**, ya que la solución óptima local termina siendo la **solución óptima global**.

How Do You Calculate a Minimum Spanning Tree? (demostración de MST very easy to understand visually)

Ser AGM \Rightarrow camino Minimax

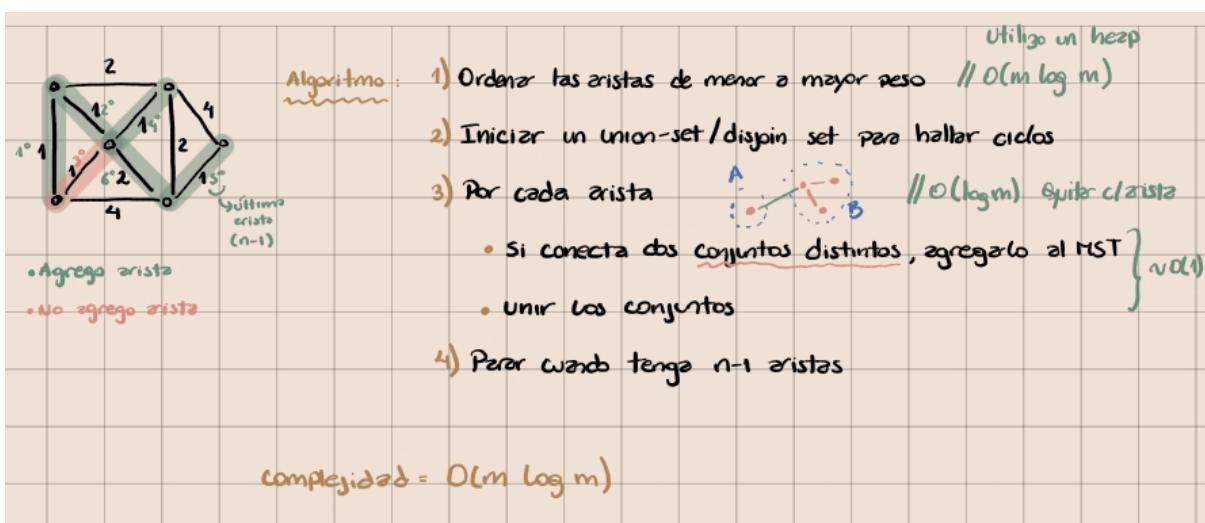
Algoritmo Kruskal

La idea es **construir el árbol generador mínimo (MST)** agregando aristas de menor peso, evitando formar ciclos.

Idea: Partimos de un subgrafo generador con 0 aristas, y en cada paso agregamos una arista de peso mínimo que no forme ciclos con las demás aristas del conjunto, hasta haber agregado $n-1$ aristas.

Kruskal's algorithm in 2 minutes

Union Find in 5 minutes — Data Structures & Algorithms



```

MST-KRUSKAL( $G, w$ )
1  $A = \emptyset$ 
2 for each vertex  $v \in G.V$ 
3   MAKE-SET( $v$ )
4 create a single list of the edges in  $G.E$ 
5 sort the list of edges into monotonically increasing order by weight  $w$ 
6 for each edge  $(u, v)$  taken from the sorted list in order
7   if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
8      $A = A \cup \{(u, v)\}$ 
9     UNION( $u, v$ )
10 return  $A$ 

```

Complejidad:

Usando Disjoint Set $\rightarrow O(m \log m + m \text{ arch}(v)^{-1}) = O(m \log m)$ donde $\text{arch}(v)^{-1}$ se asemeja a $O(1)$

Usando implementación estándar $\rightarrow O(m * n)$

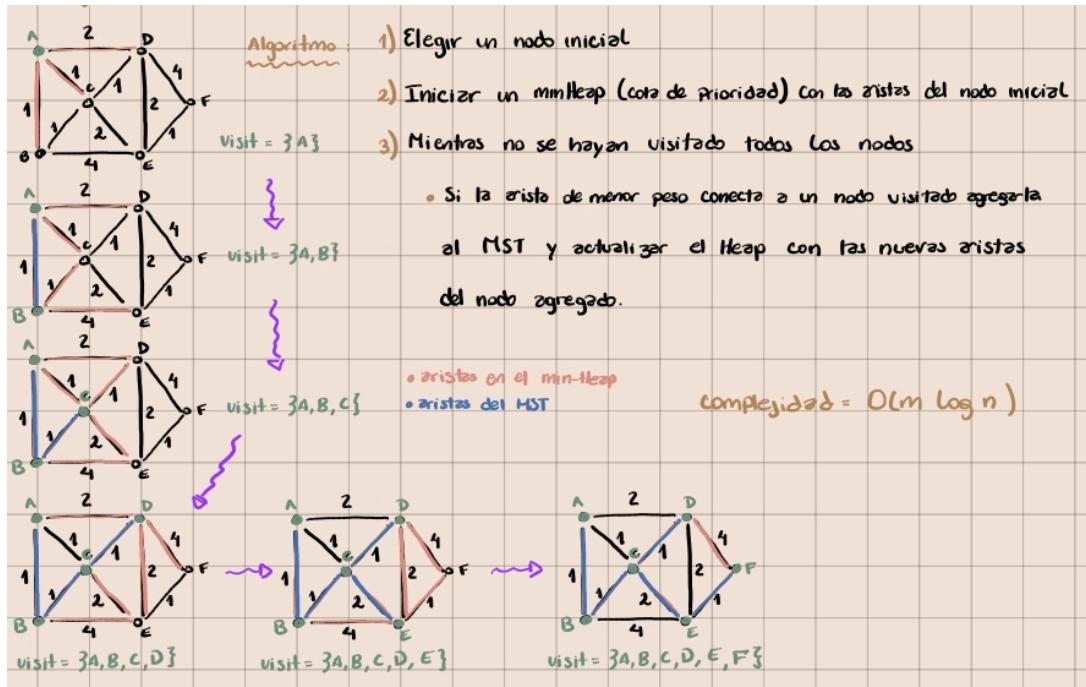
Algoritmo Prim

Construye el MST **expandiendo un árbol desde un nodo**, eligiendo siempre la **arista más barata** que lo conecta con un nodo no visitado.

Idea: Partir de un conjunto de aristas $A = \{e\}$ y un conjunto de vértices $= \{v, w\}$, donde e es una arista de peso mínimo en G , con v y w sus extremos. En cada paso, agregar a A una arista f de peso mínimo con un extremo en W y el otro en $V_G - W$. Agregar a W el extremo de f que no estaba en W , hasta que $W = V(G)$.

[▶ Prim's algorithm in 2 minutes](#)

[▶ Prim's Minimum Spanning Tree Algorithm | Graph Theory](#)



MST-PRIM(G, w, r)

```

1 for each vertex  $u \in G.V$ 
2    $u.key = \infty$ 
3    $u.\pi = \text{NIL}$ 
4    $r.key = 0$ 
5    $Q = \emptyset$ 
6   for each vertex  $u \in G.V$ 
7     INSERT( $Q, u$ )
8   while  $Q \neq \emptyset$ 
9      $u = \text{EXTRACT-MIN}(Q)$  // add  $u$  to the tree
10    for each vertex  $v$  in  $G.\text{Adj}[u]$  // update keys of  $u$ 's non-tree neighbors
11      if  $v \in Q$  and  $w(u, v) < v.key$ 
12         $v.\pi = u$ 
13         $v.key = w(u, v)$ 
14        DECREASE-KEY( $Q, v, w(u, v)$ )

```

Complejidad:

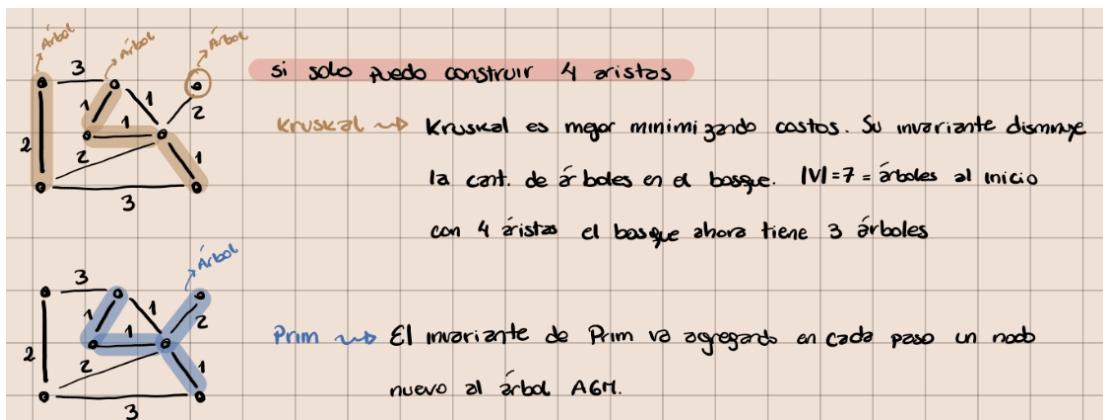
$O(n^2)$ si el grafo es denso

Usando Heap Binario $\rightarrow O((m + n) \log(n))$

Usando Heap Fibonacci $\rightarrow O(m + n \log(n))$

Prim VS Kruskal

Invariantes: Una diferencia entre el invariante de Prim y Kruskal, es que **Prim va armando un árbol AGM nodo a nodo, mientras que Kruskal va disminuyendo sus componentes conexas**, o sea cada vez que agrega una arista disminuye la cantidad de árboles en el bosque.



Prim mantiene un solo árbol en crecimiento, empezando desde un nodo. En cada paso, agrega un nodo al árbol actual, el más cercano que aún no está incluido.

Invariante: "Siempre tengo un solo árbol conexo que crece hacia los nodos no visitados con la menor arista posible."

Kruskal comienza con todos los nodos aislados (bosque de n árboles). En cada paso, agrega una arista mínima que conecta dos árboles distintos, evitando ciclos. Cada vez que Kruskal agrega una arista válida, disminuye la cantidad de componentes conexas del bosque (de k árboles pasa a $k-1$). Cuando llega a un solo árbol, terminó.

Invariante: "Siempre tengo un bosque de árboles disjuntos, y en cada paso, uno dos componentes distintos con la arista más barata posible."

Criterio	Kruskal	Prim
Tipo de grafo	Disperso	Denso
Estructura base	Conjunto de aristas	Grafo con listas de adyacencia
Ordenamiento necesario	Sí (ordenar aristas)	No (se usa heap)
Evita ciclos usando	Union-Find (componentes)	Visitados + cola de prioridad
Más similar a...	Greedy por componentes	Greedy desde un nodo

Cuándo usar Kruskal

- **El grafo es disperso** (pocas aristas comparado con la cantidad de vértices).
- Las aristas ya están disponibles en una lista o se pueden ordenar fácilmente.

Cuándo usar Prim

- **El grafo es denso** (muchas aristas, cercano a completo).
- Tenés una representación con listas de adyacencia.
- **Querés crecer el MST de a un nodo.**

GREEDY

La técnica de diseño greedy **consiste en construir una solución óptima paso a paso, eligiendo en cada paso la opción localmente óptima** (la mejor decisión en ese momento) con la esperanza de que esta lleve a una solución global óptima. Este enfoque prioriza la velocidad y la simplicidad, eligiendo generalmente el beneficio más inmediato o la mejor opción en el momento actual, por lo que **no siempre garantizan la mejor solución posible**.

Heurística → Una **heurística** es una **estrategia o método** (algoritmo) que busca una solución rápida y razonablemente a un problema, sin garantizar que sea la mejor.

- **Algoritmo greedy exacto:** Es aquel en el que la heurística greedy sí garantiza que la solución encontrada es óptima para todos los casos del problema.
- **Algoritmo greedy aproximado:** Usa una heurística que no garantiza óptimo, pero da una solución "razonablemente buena" y eficiente.

 Muchos problemas no pueden ser resueltos mediante esta técnica. En esos casos, proporcionan heurísticas sencillas que en general permiten construir soluciones razonables, pero subóptimas.

 La idea es construir una solución paso por paso, de manera de hacer la mejor elección posible localmente según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección. Es decir, en cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.

Se utiliza para resolver **problemas de optimización**, es decir, aquellos donde buscamos minimizar o maximizar alguna cantidad (como costo, tiempo, valor, etc.).

↑ Fácil de implementar	↓ Difícil de probar correctitud	↓ No siempre funciona
↑ Buena complejidad		

Técnica	¿Explora todas las soluciones?	¿Recuerda decisiones pasadas?	¿Corrige si se equivoca?	Tiempo típico
Backtracking	Sí (fuerza bruta inteligente)	Sí	Sí (backtrack)	Exponencial
Programación Dinámica (PD)	No, pero guarda subproblemas	Sí (memoización o tabla)	No (reutiliza resultados)	Polinomial
Greedy	No	No	No	Muy eficiente (lineal o logarítmico a menudo)

3. Greedy Method - Introduction

Garantía de Optimalidad: Cuando decimos que un algoritmo greedy tiene garantía de optimalidad, estamos diciendo que **si seguimos su estrategia, elegir el mejor paso local, vamos a obtener la mejor solución global**.

Para que esto pase, el problema debe cumplir dos propiedades:

- El problema tiene **subestructura óptima** (una solución óptima contiene soluciones óptimas en sus subproblemas).
- Tiene la propiedad de elección greedy, **siempre existe una solución óptima que empieza por la mejor opción local**.

Cuando estas dos condiciones se cumplen, Greedy es correcto y eficiente. Ahora si alguna de estas condiciones no se cumple, Greedy puede fallar. Por eso no siempre sirve.

Sistema canónico: No importa cuánto quieras cambiar las entradas al problema, greedy siempre te da la mejor combinación. Entonces, si un problema tiene propiedades de sistema canónico, entonces Greedy garantiza una solución óptima.

Sistema no canónico: Hay casos donde Greedy se equivoca. Por ejemplo, en el ejemplo del cambio, si cambio las monedas que puedo usar, puede pasar que se usen más monedas de las necesarias.

Un ejemplo de un algoritmo greedy es el Algoritmo de caminos más cortos (Dijkstra): encuentra el camino más corto entre nodos seleccionando siempre el nodo más cercano no visitado.

Ejemplo 2. Problema de la mochila continuo: Contamos con una mochila con una capacidad máxima C (peso máximo), donde podemos cargar n los objetos. Puede ser todo el objeto o una fracción de él, por eso lo de «continuo» en el nombre del problema. Cada elemento i , pesa un determinado peso p_i y llevarlo nos brinda un beneficio b_i .

Queremos determinar qué objetos debemos incluir en la mochila, sin excedernos del peso máximo C , de modo tal de maximizar el beneficio total entre los objetos seleccionados. En la versión más simple de este problema vamos a suponer que podemos poner parte de un objeto en la mochila (continuo).

Un algoritmo goloso para este problema, agregaría objetos a la mochila mientras esta tenga capacidad libre. En líneas generales:

```
llenarMochila( $C, n, P, B$ )
    entrada:  $C$  capacidad,  $n$  cantidad de elementos,
              $P$  arreglo con los pesos,  $B$  arreglo con los beneficios
    salida: arreglo  $X$ , donde  $X[i]$  proporción del elemento  $i$  colocado en la mochila,
             $benef$  beneficio obtenido

    ordenar los elementos según algún criterio
    mientras  $C > 0$  hacer
         $i \leftarrow$  siguiente elemento
         $X[i] \leftarrow \min(1, C/P[i])$ 
         $C \leftarrow C - P[i] \times X[i]$ 
         $benef \leftarrow benef + B[i] \times X[i]$ 
    fin mientras
    retornar  $X, benef$ 
```

Algunos criterios posibles para ordenar los objetos son:

- ordenar los objetos en orden decreciente de su beneficio
- ordenar los objetos en orden creciente de su peso
- ordenar los objetos en orden decreciente según ganancia por unidad de peso (b_i/p_i)

Supongamos que $n = 5$, $C = 100$ y los beneficios y pesos están dados en la tabla

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

Los resultados que obtendríamos ordenando los ítems según cada criterio son:

- mayor beneficio b_i : $66 + 60 + 40/2 = 146$.
- menor peso p_i : $20 + 30 + 66 + 40 = 156$.
- maximice b_i/p_i : $66 + 20 + 30 + 0,8 \cdot 60 = 164$.

Se puede demostrar que la selección según beneficio por unidad de peso, b_i/p_i , da una solución óptima.

Si los elementos deben ponerse completos en la mochila la situación es muy diferente. Eso lo veremos más adelante.

i La demo esta en la diapo 15 de la clase 5

Ejemplo 3. Problema del cambio: Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es \$0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y 4 de un centavo.

Un algoritmo goloso para resolver este problema es seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y sustraer la cantidad correspondiente a la cantidad que resta por devolver (hasta que sea 0).

```
darCambio(cambio)
    entrada: cambio ∈ ℑ
    salida: M multiconjunto de enteros

    suma ← 0
    M ← {}
    mientras suma < cambio hacer
        proxima ← masgrande(cambio, suma)
        M ← M ∪ {proxima}
        suma ← suma + proxima
    fin mientras
    retornar M
```

Este algoritmo siempre produce la mejor solución, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor *cambio*. Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima; si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y 4 de 1 centavos, mientras que la solución óptima es retornar 2 monedas de 10 centavos y una de 1 centavo.

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

CARACTERÍSTICAS

Características clave de los algoritmos greedy

- **Decisiones localmente óptimas:** En cada paso, el algoritmo selecciona la opción que parece mejor (decisión óptima local) según la situación actual, sin considerar las consecuencias futuras.
- **Simplicidad y velocidad:** Suelen ser fáciles de entender e implementar, lo que lleva a tiempos de ejecución rápidos, especialmente en problemas pequeños o bien estructurados.
- **No siempre óptimos:** Aunque apuntan a una solución óptima global, pero los algoritmos greedy no siempre la alcanzan.
- **Aplicables a problemas de optimización:** Son adecuados para problemas que buscan maximizar o minimizar un valor, dentro de ciertas restricciones.
- **Propiedad de elección greedy:** El problema debe tener la propiedad de que **hacer la mejor elección local en cada etapa garantiza una solución global óptima**. (Según GeeksforGeeks)
- **Subestructura óptima:** El problema debe tener subestructura óptima, es decir, **una solución óptima del problema completo contiene soluciones óptimas de sus subproblemas**. (Según GeeksforGeeks)4

DEMOSTRACIONES

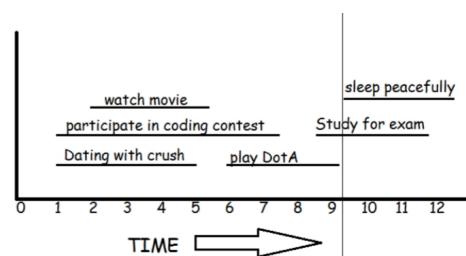
Para demostrar que un algoritmo greedy (goloso) es óptimo, existen principalmente **dos enfoques comunes de demostración: por contradicción (absurdo) y por inducción.**

- Ir por el absurdo sirve cuando sólo existe una solución óptima. La idea es suponer que la solución que obtengo con mi algoritmo greedy no es óptima (o que es distinta a la óptima) y si la comparo con la óptima debo llegar a un **absurdo**, lo que significa que mi solución es óptima, es decir, que no puede existir una solución mejor que la greedy, y por tanto la greedy es óptima.
- Hacer inducción se utiliza cuando hay varias soluciones óptimas, y por lo general la inducción abarca más casos. La idea es demostrar que, al aplicar la regla greedy paso a paso se mantiene la optimalidad en cada etapa.

Algoritmos Golosos

Chimi es una persona muy ocupada, pero como es muy ordenado tiene un cronograma con todas las actividades que podría realizar. Sin embargo, muchas de estas actividades se superponen entre sí, por lo que debe decidir cuidadosamente cuáles hacer y cuáles no. Su objetivo es realizar la mayor cantidad posible de actividades, siempre que ninguna se solape con otra.

Dado un conjunto de actividades, cada una con un horario de inicio y fin, ¿cuál es el máximo número de actividades que Chimi puede llevar a cabo sin que se superpongan?



- ① Ordenar las actividades por su tiempo de finalización en orden creciente: $\{(1, 2), (3, 5), (3, 6), (6, 7), (6, 9), (5, 12)\}$
- ② Inicializar un conjunto vacío B para las actividades seleccionadas.
- ③ Inicializar una variable $t \leftarrow -\infty$ que representa el final de la última actividad seleccionada.
- ④ Para cada actividad (t_i, t_f) en el orden dado:
 - Si $t_i > t$, entonces:
 - Agregar (t_i, t_f) al conjunto B .
 - Actualizar $t \leftarrow t_f$.
- ⑤ Devolver el conjunto B .

Vamos a hacerla por inducción. Usaremos la siguiente notación:

- Sea $B = (B_1, B_2, \dots, B_k)$ la solución construida por el algoritmo goloso, donde B_i es la i -ésima actividad seleccionada.
- Sea $O = (O_1, O_2, \dots, O_k)$ una solución óptima (pueden existir varias).

Caso base ($i = 1$):

Nuestro algoritmo elige como primera actividad B_1 , la que termina antes que todas las otras.

- Sabemos que $\text{fin}(B_1) \leq \text{fin}(O_1)$, entonces si definimos $O' = (B_1, O_2, \dots, O_k)$ tenemos una nueva solución óptima, ya que si O_1 no se solapaba con ninguna otra, entonces B_1 tampoco.
- Por lo tanto, el algoritmo comienza con una elección que puede ser parte de una solución óptima

Paso inductivo:

Queremos probar que podemos extender la secuencia B_1, \dots, B_i agregando B_{i+1} y que a partir de esta se puede obtener una solución óptima.

- Por H.I. sabemos que existe la solución óptima $O = (B_1, \dots, B_i, O_{i+1}, \dots, O_k)$
- Por cómo elige el algoritmo sabemos que B_{i+1} es la actividad compatible con B_i que termina más temprano.
- Luego, O_{i+1} también es compatible con B_i pero por cómo elige nuestro algoritmo sabemos que $\text{fin}(B_{i+1}) \leq \text{fin}(O_{i+1})$.
- Entonces podemos construir una nueva solución óptima $O' = (B_1, \dots, B_i, B_{i+1}, O_{i+2}, \dots, O_k)$.
- Por inducción, podemos extender paso a paso la solución parcial obtenida por el algoritmo goloso hacia una solución completa que es óptima. Esto demuestra que el algoritmo selecciona el **máximo de actividades sin que estas se superpongan**.

Hipótesis inductiva:

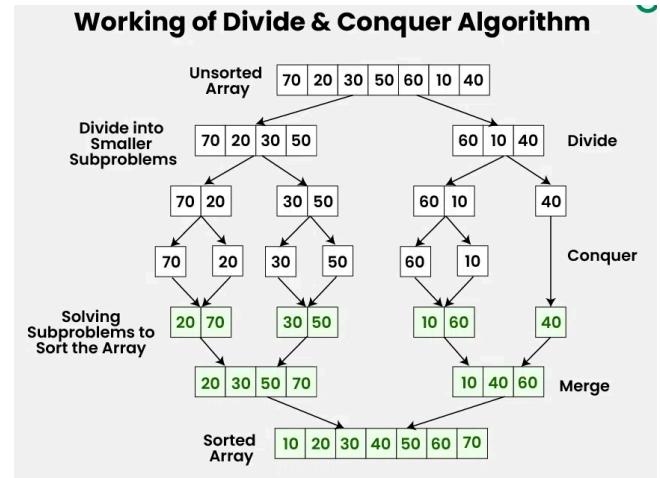
Suponemos que luego de i pasos, el algoritmo ha seleccionado una secuencia B_1, \dots, B_i , que puede extenderse a una solución óptima $O = (B_1, \dots, B_i, O_{i+1}, \dots, O_k)$

DIVIDE AND CONQUER

Divide and Conquer es una **técnica algorítmica** en la que un problema se divide recursivamente en subproblemas más pequeños, luego se resuelven independientemente, y finalmente se combinan las soluciones de los subproblemas para obtener la solución del problema original.

💡 Independencia de los subproblemas : Cada subproblema debe ser independiente de los demás, lo que significa que la solución de uno no depende de la de otro. Esto permite el procesamiento en paralelo o la ejecución concurrente de subproblemas, lo que puede generar mejoras de eficiencia.

2 Divide And Conquer



Lo que **diferencia Divide and Conquer de otras técnicas como Backtracking y Programación Dinámica**, es que en D&C los subproblemas suelen ser **independientes y no se repiten**, por lo que **no es necesario memorizar** resultados como en PD (aunque en algunos casos puede hacerse para optimizar, lo que se conoce como **memorización o DP top-down**).

En cuanto a la estructura, D&C suele dividir los problemas en subproblemas de tamaño n / c (como la mitad en cada llamada recursiva), lo que típicamente lleva a una **complejidad logarítmica o polinomial**, como $O(n \log(n))$. En contraste, Backtracking resuelve subproblemas de forma más "exploratoria", avanzando en pasos de $n - 1$ (por ejemplo, decisiones sí/no, combinaciones, etc.), lo cual lleva muchas veces a una **explosión combinatoria** y una **complejidad exponencial**, especialmente si no se aplican podas.

ESTRUCTURA

1) Dividir (Divide) → Dividir el problema original en subproblemas de menor tamaño **del mismo tipo que el problema original**, hasta que no sea posible realizar más divisiones.

2) Conquistar (Conquer) → Resolver cada subproblema recursivamente o directamente si son suficientemente pequeños (**caso base**). El objetivo es encontrar soluciones para estos subproblemas de forma independiente.

3) Combinar (Combine) → Unir las soluciones de los subproblemas para obtener una solución global del problema.

Dado un **problema a resolver para una entrada de tamaño n** , se divide la entrada en r subproblemas. Estos subproblemas se resuelven de forma independiente y después se combinan sus soluciones parciales para obtener la solución del problema original. En esta técnica, **los subproblemas deben ser de la misma clase que el problema original**, permitiendo una resolución recursiva. Por supuesto, deben existir algunos casos sencillos, cuya solución pueda calcularse directamente.

```
d&c(x)
  entrada: x
  salida: y
  si x es suficientemente fácil entonces
    y ← calcular directamente
  sino
    descomponer x en instancias más chicas x1, ..., xr
    para i = 1 hasta r hacer
      yi ← d&c(xi)
    fin para
    y ← combinar los yi
  fin si
  retornar y
```

COMPLEJIDAD

Estos algoritmos son de índole recursiva, por lo que vamos a seguir el mismo razonamiento para el cálculo de su complejidad. Generalmente las instancias que son caso base toman tiempo constante c (es $O(1)$) y para los casos recursivos podemos identificar tres puntos críticos:

- $r \rightarrow$ Cantidad de llamadas o subproblemas a resolver.
- $n / b \rightarrow$ Tamaño de cada subproblema, para alguna constante b . Se supone que todos los subproblemas tienen el mismo tamaño.
- $g(n) \rightarrow$ Coste del trabajo realizado fuera de la llamada recursiva, que incluye el coste de dividir el problema y el coste de fusionar las soluciones para una instancia de tamaño n .

Entonces, **tiempo total $T(n)$ consumido por el algoritmo** está definido por la siguiente **ecuación de recurrencia**:



💡 Siempre el costo del conquistar va a ser constante porque vamos a achicar el problema hasta entradas de algún tamaño particular que sepamos resolver rápido.

Para calcular el tiempo de ejecución de definiciones de recursivas hay distintas estrategias, como armar el árbol de recursiones (**forma manual**), buscar una fórmula no recursiva o usando el teorema maestro.

Complejidad del Merge Sort

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2 T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

subproblemas a resolver
Tamaño del Subproblema

* Resolución por búsqueda de una fórmula cerrada

$$\begin{aligned} T(n) &= 2 T(n/2) + \Theta(n) \\ &= 2 [2 T(n/2^2) + \Theta(n/2)] + \Theta(n) = 2^2 T(n/2^2) + 2\Theta(n) \\ &= 2^2 [2 T(n/2^3) + \Theta(n/2^3)] + \Theta(n) = 2^3 T(n/2^3) + 3\Theta(n) \\ &\vdots \\ &= 2^i T(n/2^i) + i\Theta(n) \end{aligned}$$

(esto se va a repetir de forma recursiva hasta que $\frac{n}{2^i} = 1 \iff i = \log_2(n)$)

$$\begin{aligned} \text{iteración } i = \log_2(n) &= 2^{\log_2(n)} T(n/2^{\log_2(n)}) + \log_2(n) \Theta(n) = \underbrace{2^{\log_2(n)}}_n + \underbrace{\log_2(n) \Theta(n)}_{\Theta(\log(n) + n)} = \Theta(n \cdot \log(n)) \\ &= 1 \Rightarrow T(1) = \Theta(1) \end{aligned}$$

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    medio = len(arr) // 2
    mitad_izq = merge_sort(arr[:medio])
    mitad_der = merge_sort(arr[medio:])
    return merge(mitad_izq, mitad_der)

def merge(izq, der):
    mergeados = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            mergeados.append(izq[i])
            i += 1
        else:
            mergeados.append(der[j])
            j += 1
    mergeados.extend(izq[i:])
    mergeados.extend(der[j:])
    return mergeados
```

▶ 2.1.1 Recurrence Relation ($T(n)=T(n-1)+1$) #1

▶ 2.1.2 Recurrence Relation ($T(n)=T(n-1)+n$) #2

▶ 2.1.3 Recurrence Relation ($T(n)=T(n-1)+\log n$) #3

▶ 2.1.4 Recurrence Relation $T(n)=2 T(n-1)+1$ #4

Teorema Maestro

Otra forma de resolver la complejidad del algoritmo usando la ecuación de recurrencia es mediante el **teorema maestro**. El **Teorema del Maestro** es una herramienta clave para analizar la complejidad de algoritmos de tipo **Divide and Conquer**, especialmente cuando un problema se resuelve dividiéndolo recursivamente en subproblemas de igual tamaño.

tercios). Si partimos en a subproblemas de tamaño $1/c$ del problema original, nuestra función tiempo nos va a quedar de la forma:

$$T(n) = \begin{cases} a \cdot T(n/c) + f(n) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$$

Para las funciones de tiempo de este tipo, en la mayoría de los casos vamos a poder calcularles el tiempo usando el teorema maestro:

- Si $f(n) = O(n^{\log_c a - \varepsilon})$ para $\varepsilon > 0$, entonces $T(n) = \Theta(n^{\log_c a})$
- Si $f(n) = \Theta(n^{\log_c a})$, entonces $T(n) = \Theta(n^{\log_c a} \log n)$
- Generalizando el caso anterior,
Si $f(n) = \Theta(n^{\log_c a} \log^k n)$ para algún $k \geq 0$, entonces $T(n) = \Theta(n^{\log_c a} \log^{k+1} n)$
- Si $f(n) = \Omega(n^{\log_c a + \varepsilon})$ para $\varepsilon > 0$ y $af(n/c) < kf(n)$ para $k < 1$ y n suficientemente grandes, entonces $T(n) = \Theta(f(n))$

El Teorema del Maestro te permite determinar la complejidad temporal (en notación O) de algoritmos recursivos cuya recurrencia tiene la forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Donde:

- $a \geq 1$: número de subproblemas.
- $b > 1$: factor por el cual se reduce el tamaño del problema.
- $f(n)$: trabajo que se hace fuera de las llamadas recursivas (ej: combinar soluciones, dividir, etc.).

comparar el crecimiento de $f(n)$ contra $n^{\log_b(a)}$

referencia

Caso 1: si $f(n) = O(n^{\log_b(a)-\varepsilon})$ para algún $\varepsilon > 0 \rightarrow T(n) = \Theta(n^{\log_b(a)})$

Caso 2: si $f(n) = \Theta(n^{\log_b(a)} \cdot [\log(n)]^k)$ $\rightarrow T(n) = \Theta(n^{\log_b(a)} \cdot [\log(n)]^{k+1})$

Caso 3: si $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ para algún $\varepsilon > 0$ y $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ para algún $c < 1 \rightarrow T(n) = \Theta(f(n))$

<https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/>

Complejidad de búsqueda binaria

subproblemas a resolver: $1 \cdot T(n/2) + O(1)$ Costo de dividir y Unir

$T(n) = \begin{cases} 1 \cdot T(n/2) + O(1) & \text{si } n > 1 \\ 1 & \text{si } n = 1 \end{cases}$

```
def busqueda_binaria(arr, objetivo, izq=0, der=len(arr)-1):
    if izq > der:
        return False # Elemento no encontrado O(1)
    medio = (izq + der) // 2
    if arr[medio] == objetivo:
        return medio O(1)
    elif arr[medio] > objetivo:
        return busqueda_binaria(arr, objetivo, izq, medio - 1) O(1)
    else:
        return busqueda_binaria(arr, objetivo, medio + 1, der) O(1)
```

Dividir (hay que dividir)

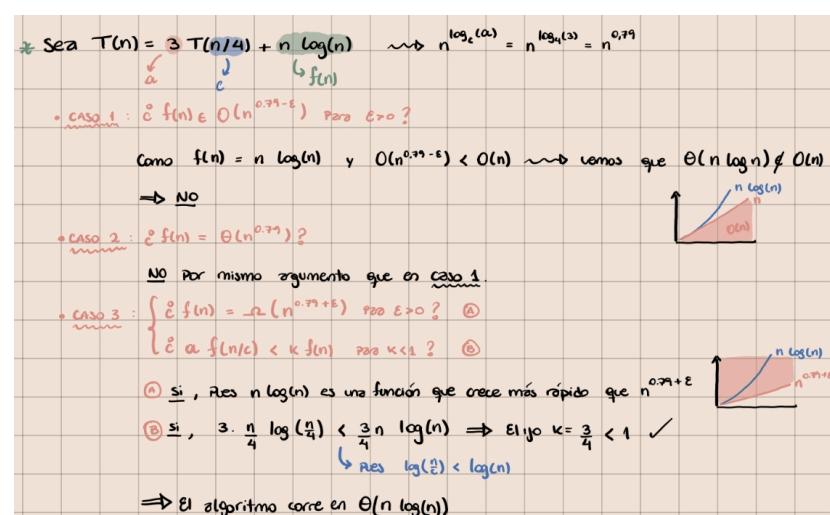
% Resolución por Teorema Maestro

observamos que: $a=1$, $c=2$, $f(n) = O(1)$ $\rightarrow n^{\log_2(1)} = n^0 = 1$

Caso 1: $\because f(n) = O(n^{\log_2(1)-\varepsilon})$ para $\varepsilon > 0$?
 $\text{Como } n^{\log_2(1)-\varepsilon} = n^{-\varepsilon} = \frac{1}{n^\varepsilon} \Rightarrow O(1) = O\left(\frac{1}{n^\varepsilon}\right) \text{ NO} \rightarrow O(1)$

Caso 2: $\because f(n) = \Theta(n^{\log_2(1)})$?
 $\text{Como } n^{\log_2(1)} = 1 \Rightarrow O(1) = O(1) \text{ si entonces } T(n) = \Theta(n^{\log_2(1)} \cdot \log(n)) = \Theta(\log(n))$

Caso 3: No hace falta verlo porque ya se cumplió el caso 2.



Otra regla es la siguiente:

Si existe un entero k tal que $g(n)$ es $O(n^k)$ se puede demostrar que:

$$T(n) \text{ es } \begin{cases} O(n^k) & \text{si } r < b^k \\ O(n^k \log n) & \text{si } r = b^k \\ O(n^{\log_b r}) & \text{si } r > b^k \end{cases}$$

 Las condiciones del teorema maestro son excluyentes, esto quiere decir que si se satisface un caso los otros no se satisfacerá, entonces solo hay una complejidad para el algoritmo.

UNIDAD 6 - FLUJO

RED DE FLUJO

https://cp-algorithms.com/graph/edmonds_karp.html (good summary)

Flow Networks - Georgia Tech - Computability, Complexity, Theory: Algorithms

Una red de flujo es un grafo dirigido conexo donde cada arista tiene una capacidad y transporta un flujo, desde un nodo fuente s (con grado positivo de salida) a un nodo sumidero t (con grado positivo de entrada). Se utiliza para modelar situaciones donde algo (agua, datos, bienes, etc.) fluye de un origen a un destino, sin exceder las capacidades de las conexiones.

 El objetivo es mover alguna entidad (líquido a través de cañería, electricidad en una red eléctrica, datos en una red de comunicación, etc) desde un punto a otro de la red de la forma más eficiente posible.

Una red de flujo es un grafo orientado conexo que tiene dos vértices distinguidos, una fuente s , con grado de salida positivo, y un sumidero t , con grado de entrada positivo.

Representamos una red de flujo como una tupla: $G = (V, E, c, s, t)$, a veces llamamos a la red N (network)

- V es el conjunto de vértices.
- $E \subseteq V \times V$ es el conjunto de aristas dirigidas.
- $c: E \rightarrow \mathbb{R}^+$ es la función de capacidad, donde $c(u, v)$ es la capacidad máxima de la arista (u, v) .
- $s \in V$ es el nodo fuente (source).
- $t \in V$ es el nodo sumidero (sink).

FLUJO FACTIBLE

* Un **flujo factible** en la red es una función $f: E \rightarrow \mathbb{R}_{\geq 0}$ que verifica que :

1) **Capacidad limitada** : $0 \leq f(u, v) \leq c(u, v) \quad \forall (u, v) \in E$ no se puede enviar más de lo que permite la capacidad.

2) **Conservación de flujo** : $\sum_{w \in \text{In}(v)} f(w, v) = \sum_{w \in \text{Out}(v)} f(v, w) \quad \forall v \in V - \{s, t\}$ lo que entra a un nudo debe salir

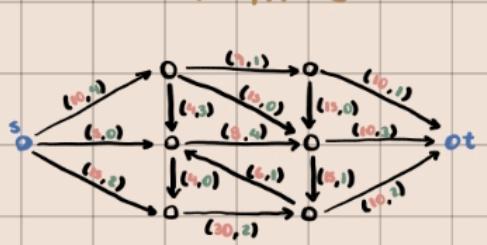
3) **Valor del flujo** : $|f| = \sum_{v \in V} f(s, v)$ es la cantidad total que sale del origen y entra al sumidero.
ó (puedo usar sat)

$$F = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v) = \sum_{v \in V} f(v, t)$$

$\cancel{\sum_{v \in V} f(t, v)}$ \rightarrow verdad (?)

$$F = |f| = 6$$

Ejemplo: En cada arista, la primera coordenada del par ordenado representa su capacidad, mientras que la segunda coord. representa el flujo sobre la arista.



FLUJO MÁXIMO

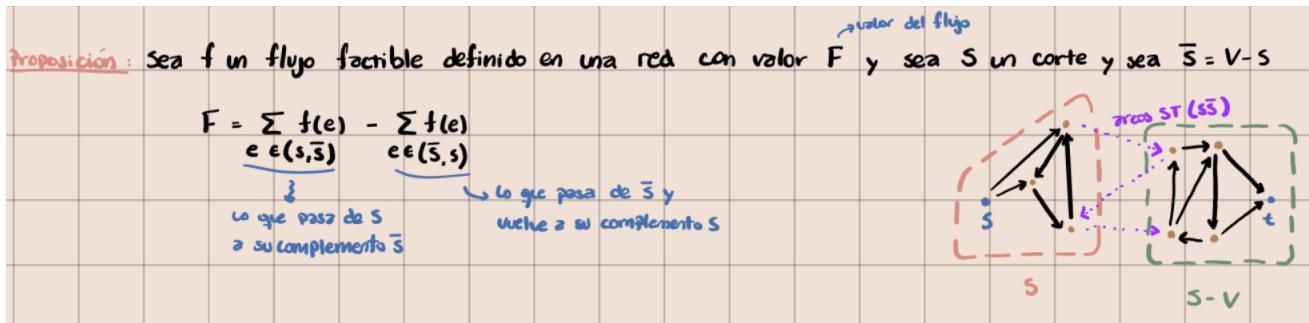
El flujo máximo es el flujo más grande posible que se puede mandar desde s hasta t , respetando las capacidades y la conservación. Formalmente, queremos: $\max \{f\} = \max\{F\}$ sujeto a las restricciones de flujo.

 Imagina una red de tuberías donde querés mandar la máxima cantidad de agua desde un depósito (fuente) a una ciudad (sumidero), pero cada caño (arista) tiene un límite de litros por minuto (capacidad). El problema del flujo máximo busca la mejor manera de usar esas tuberías sin reventarlas para mandar la mayor cantidad posible.

Cortes

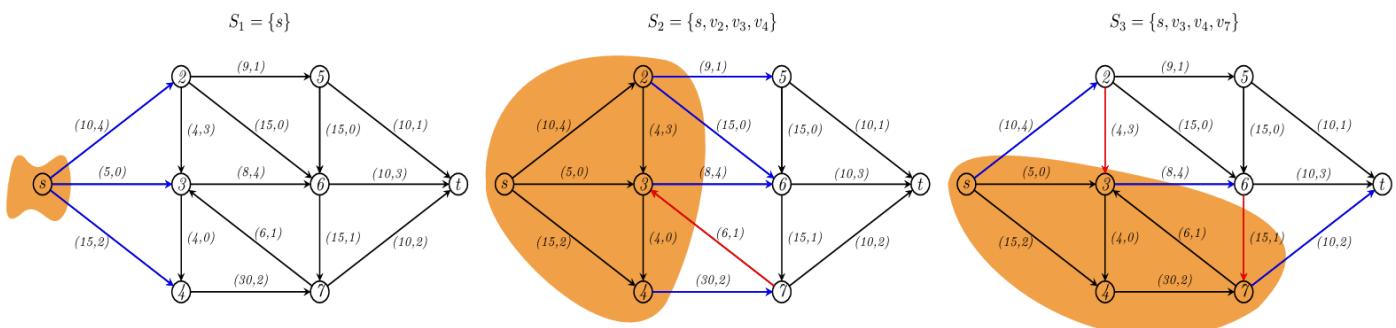
Un **corte en la red** es un subconjunto $S \subseteq V - \{t\}$, tal que s (vértice de origen) $\in S$.

Notación: Dados $S, T \subseteq V$, llamaremos $ST = \{(u \rightarrow v) \in X : u \in S \text{ y } v \in T\}$. (arcos que salen de S y llegan a T)



Ejemplo 3. Se muestran los gráficos para tres posibles cortes, S_1 , S_2 y S_3 . El valor del flujo F es igual a la suma de los flujos sobre los arcos azules menos la suma de los flujos sobre los arcos rojos.

$$F = 6$$



* La capacidad de un corte S se define como $c(S) = \sum_{u \in S, v \in \bar{S}} c(u, v)$ (suma de las capacidades de todas las aristas que van de un nodo en S a uno en \bar{S})

(esto representa la máxima cantidad de flujo que podría pasar del conjunto S a \bar{S})

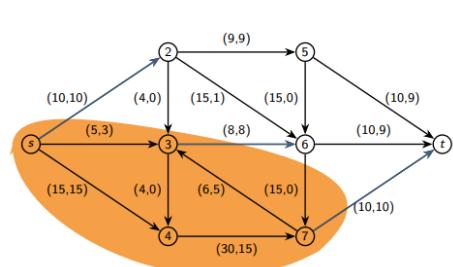
 La utilidad de los cortes es que me permiten acotar el valor máximo del flujo F , entonces podemos determinar la complejidad gracias a las cotas de los cortes en la red.

- **Lema:** Si f es una función de flujo con valor F y S es un corte en N , entonces $F \leq c(S)$.

Flujo en Redes

$$C = 28$$

$$F = 28$$



- **Corolario:** Si F es el valor de un flujo f y S un corte en N tal que $F = c(S)$ entonces f define un flujo máximo y S un **corte de capacidad mínima**.

 Cualquier otro corte que hubiera elegido en el ejemplo, tiene $F \leq c(S)$, entonces el único flujo máximo (corte de capacidad mínima) es cuando $F = c(S)$.

• Problema de Corte Mínimo (Max-Flow Min-Cut)

El problema de corte mínimo consiste en **encontrar el corte S que minimiza la capacidad total de las aristas que van de S a \bar{S}** . El valor del flujo máximo en una red es igual a la capacidad del corte mínimo entre S y $(V-S)$. https://cp-algorithms.com/graph/edmonds_karp.html

Si encontrás el corte más “estrecho” (el de menor capacidad), sabés automáticamente cuál es el flujo máximo posible que puede pasar de s a t.

* Buscarnos determinar un **corte con capacidad mínima**, es decir, buscarnos S , corte de la red G , tal que:

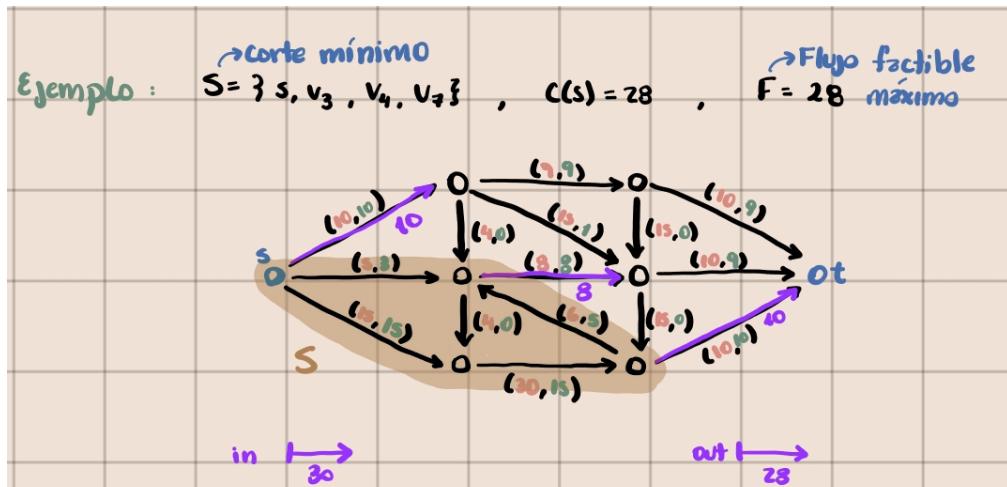
$$c(S) = \min \{ c(\bar{S}) \mid \bar{S} \text{ es un corte de } G \}$$

Lema 2. Dados una red $N = (V, X)$ con función de capacidad c , una función de flujo factible con valor F y un corte S , se cumple que:

$$F \leq c(S).$$

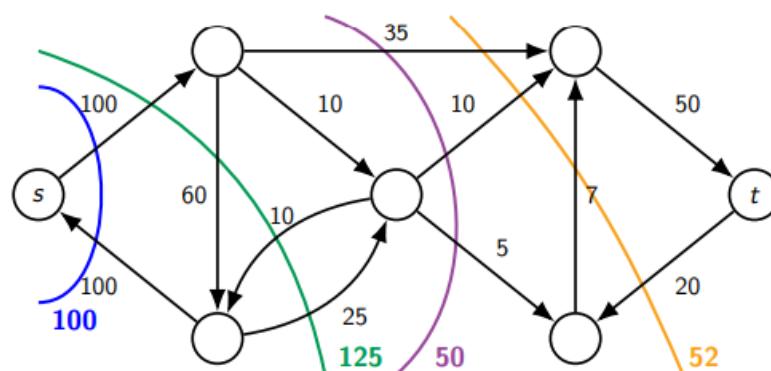
Corolario 1 (Certificado de optimalidad). Si F es el valor de un flujo factible f y S un corte en una red N tal que $F = c(S)$ entonces:

- f define un flujo factible máximo y S es un corte de capacidad mínima.



▶ Maximum flow Minimum Cut Algorithm

Ejemplo



Dualidad flujo máximo-corte mínimo

- **Débil:** Si f es un flujo y S un corte $\Rightarrow v(f) \leq c(S)$
- **Fuerte:** f es máximo y S mínimo $\Leftrightarrow v(f) = c(S)$

RED RESIDUAL & CAMINO EN AUMENTO

- * Dada una red N con función de capacidad c y un flujo factible f .

- Definimos la Red Residual $R(N, f) = (V, X_R)$ donde $V(v \rightarrow w) \in E$.

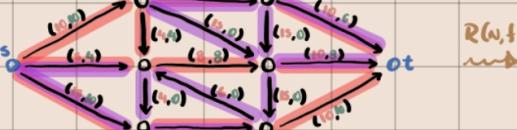
$$\begin{cases} \mapsto (v \rightarrow w) \in X_R & \text{si } f(v \rightarrow w) < c(v \rightarrow w) \\ \mapsto (w \rightarrow v) \in X_R & \text{si } f(v \rightarrow w) > 0 \end{cases}$$

} la red residual representa las capacidades restantes que se pueden aprovechar para seguir enviando flujo

- Un **camino en aumento** es un camino orientado de s a t en $R(N, f)$.

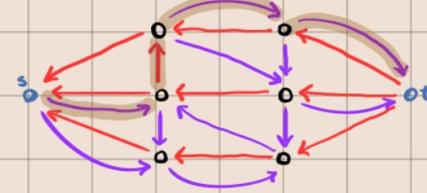
} Camino por el que todavía se puede mandar más flujo desde s hasta t

Ejemplo :



$R(N, f)$

$$\begin{cases} \mapsto (w \rightarrow v) \in X_R & \text{si } f(v \rightarrow w) > 0 \\ \mapsto (v \rightarrow w) \in X_R & \text{si } f(v \rightarrow w) < c(v \rightarrow w) \end{cases}$$



💡 Estos conceptos son clave en el algoritmo de Ford-Fulkerson para encontrar el flujo máximo.

- Algoritmo encuentra un camino de aumento si existe o, en caso contrario, determina un corte

```
caminoAumento( $N, f, R$ )
    entrada:  $N = (V, X)$ , función de flujo  $f$ , red residual  $R(N, f) = (V, X_R)$ 
    salida:  $P$  camino de aumento o  $S$  corte (que sera minimo)
```

```

 $S \leftarrow \{s\}$ 
mientras  $t \notin S$  y  $\exists (v \rightarrow w) \in X_R$  y  $v \in S$  y  $w \notin S$  hacer
     $ant[w] \leftarrow v$ 
     $S \leftarrow S \cup \{w\}$ 
fin mientras
si  $t \notin S$  entonces
    retornar  $S$  corte de  $N$ 
si no
    reconstruir  $P$  entre  $s$  y  $t$  usando  $ant$  a partir de  $t$ 
    retornar  $P$  camino de aumento
fin si
```

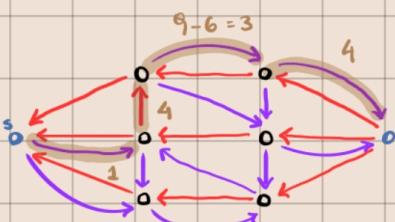
- Dada una red N , con función de capacidad c , un flujo factible f y un camino de aumento

P en $R(N, f)$:

Para cada arco $(v \rightarrow w)$ de P definimos: $\Delta((v \rightarrow w)) = \begin{cases} c(v \rightarrow w) - f(v \rightarrow w) & \text{si } (v \rightarrow w) \in E \\ f(w \rightarrow v) & \text{si } (w \rightarrow v) \in E \end{cases}$

$$\Delta(P) = \min_{e \in P} \{\Delta(e)\}$$

Ejemplo :



$$\Delta(P) = \min \{1, 4, 3, 4\} = 1$$

Proposición 3. Sea f un flujo factible definido sobre una red N con valor F y sea P un camino de aumento en $R(N, f)$. Entonces el flujo \bar{f} , definido por

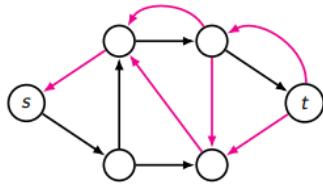
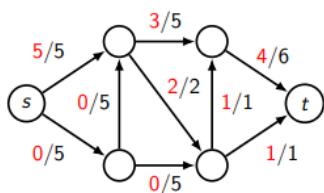
$$\bar{f}((v \rightarrow w)) = \begin{cases} f((v \rightarrow w)) & \text{si } (v \rightarrow w) \notin P \\ f((v \rightarrow w)) + \Delta(P) & \text{si } (v \rightarrow w) \in P \\ f((v \rightarrow w)) - \Delta(P) & \text{si } (w \rightarrow v) \in P \end{cases}$$

es un flujo factible sobre N con valor $\bar{F} = F + \Delta(P)$.

Definición

Dada una red $D = (V, E)$ con función de capacidad c y un flujo f , definimos la red residual $R = (V, E_R)$ donde:

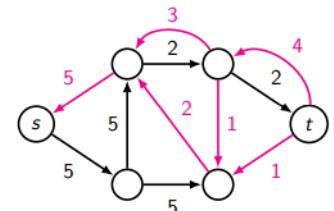
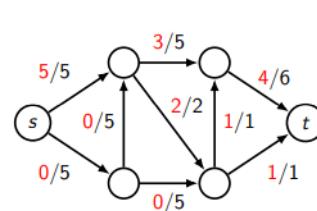
- $(v \rightarrow w) \in E_R$ si $f(v \rightarrow w) < c(v \rightarrow w)$
- $(w \rightarrow v) \in E_R$ si $f(v \rightarrow w) > 0$



Función de costos

Dada una red, $D = (V, E)$ residual $R = (V, E_R)$, un flujo f y una función de capacidad c . La función de peso p para cada arista es:

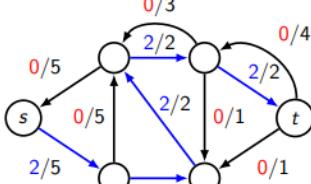
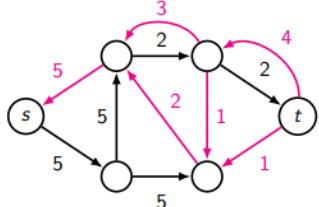
$$p(e = (v, w)) = \begin{cases} c(e) - f(e) & \text{si } (v \rightarrow w) \in E \\ f(e) & \text{si } (w \rightarrow v) \in E \end{cases}$$



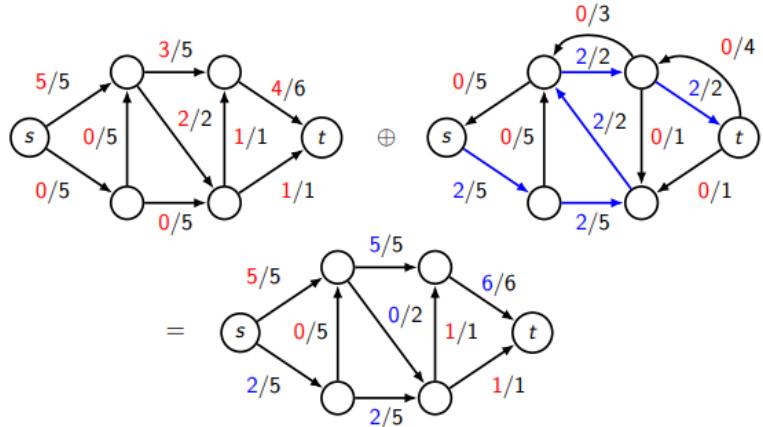
Definición

Dada una red residual $R = (V, E_R)$, cualquier camino P de s a t es un camino de aumento. El flujo asociado al camino es:

$$f^P(e) = \begin{cases} \min\{p(e) \mid e \in P\} & \text{si } e \in P \\ 0 & \text{cc} \end{cases}$$



Sumando el Camino de Aumento al Flujo



Algoritmos para Flujos Máximos

Dada una red con capacidades de flujo en los arcos, ¿cómo podemos enviar la máxima cantidad de flujo posible desde un origen a un sumidero determinados respetando las capacidades máximas de los arcos?

https://cp-algorithms.com/graph/edmonds_karp.html

• Ford-Fulkerson

- ▶ Maximum flow Minimum Cut Algorithm
- ▶ Ford-Fulkerson in 5 minutes
- ▶ Max Flow Ford Fulkerson | Network Flow | Graph Theory

La idea es **mientras haya un camino aumentante desde s a t con capacidad libre, lo uso para aumentar el flujo**. Complejidad depende del número de aumentos: si las capacidades son enteras $O(m * F)$, con F el valor máximo del flujo).

```
Ford&Fulkerson(N)
    entrada: N = (V, X) con función de capacidad c
    salida: f flujo maximo
    definir un flujo inicial en N
    (por ejemplo  $f(e) \leftarrow 0$  para todo  $e \in X$ )
    mientras existe P camino de aumento en  $R(N, f)$  hacer
        para cada arco  $(v \rightarrow w)$  de P hacer
            si  $(v \rightarrow w) \in X$  entonces
                 $f((v \rightarrow w)) \leftarrow f((v \rightarrow w)) + \Delta(P)$ 
            si no     comentario:  $((w \rightarrow v) \in X)$ 
                 $f((w \rightarrow v)) \leftarrow f((w \rightarrow v)) - \Delta(P)$ 
            fin si
        fin para
    fin mientras
```

Inicializa el flujo en 0.

Mientras haya un camino en aumento en la red residual:

Encontrás ese camino, le mandás todo el flujo posible (el mínimo de las capacidades residuales).

Actualizá la red residual (se reducen capacidades hacia adelante, aumentan hacia atrás).

Cuando ya no hay caminos en aumento, terminaste: el flujo actual es máximo.

Proposición 6. Si los valores del flujo inicial y las capacidades de los arcos de la red son enteros, el método de Ford y Fulkerson es $\mathcal{O}(nmU)$, donde U es una cota superior finita para el valor de las capacidades.

Ejemplo de F&F y algunos teoremas que pueden ser de interés en última diapositivas (34 - 46)

Teorema: Si las capacidades de los arcos de la red son enteras el problema de flujo máximo tiene un flujo máximo entero.

Teorema: Si los valores del flujo inicial y las capacidades de los arcos de la red son enteras el método de Ford y Fulkerson realiza a lo sumo nU iteraciones, siendo entonces $\mathcal{O}(nmU)$, donde U es una cota superior finita para el valor de las capacidades.

Si las capacidades o el flujo inicial son números irracionales, el método de Ford y Fulkerson puede no parar (realizar un número infinito de pasos).

• Edmonds-Karp (versión de Ford-Fulkerson)

Edmonds y Karp definieron una implementación del método de F&F utilizando BFS para el cálculo del camino de aumento (elige siempre el camino aumentante más corto en número de aristas). El procedimiento encontrará el camino entre s y t en la red residual de menor cantidad de arcos. La complejidad de esta implementación es $O(n * m^2)$, garantiza terminación en tiempo polinomial, incluso si las capacidades son enteras grandes.



Definimos la complejidad de FFEK como $O(\min\{mF, nm^2\})$, si acotamos F con cortes podemos ver cual es mejor.

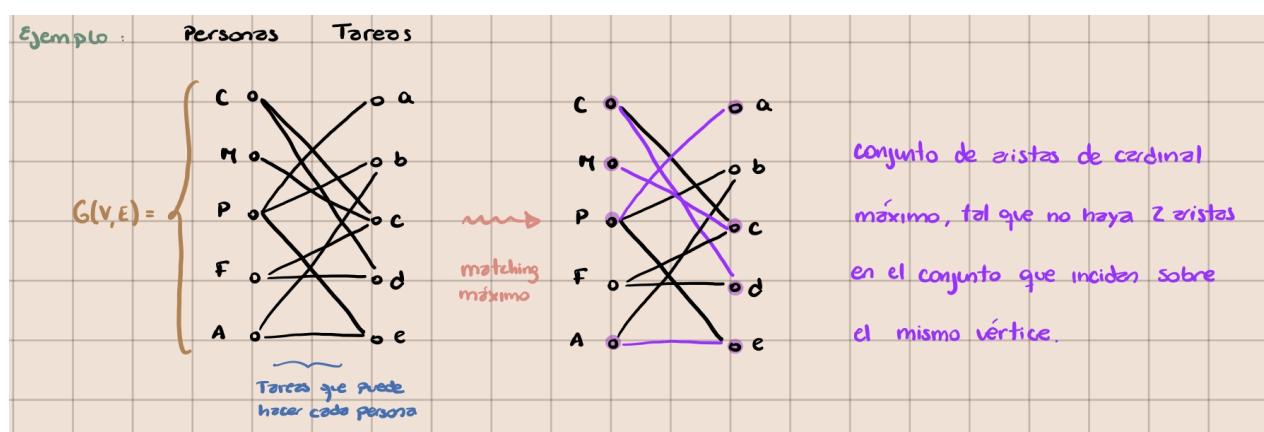
💡 El algoritmo de EK también se puede usar para obtener el corte mínimo. Cuando el algoritmo termina, el corte mínimo son los vértices alcanzables desde s en el grafo residual resultante. Si lo que buscamos es tomar una serie de decisiones que minimicen una suma de costos positivos, podemos representar cada posible decisión como un vértice, y tomarla o no como incluirla o no en el corte.

MATCHING MÁXIMO EN GRAFOS BIPARTITOS

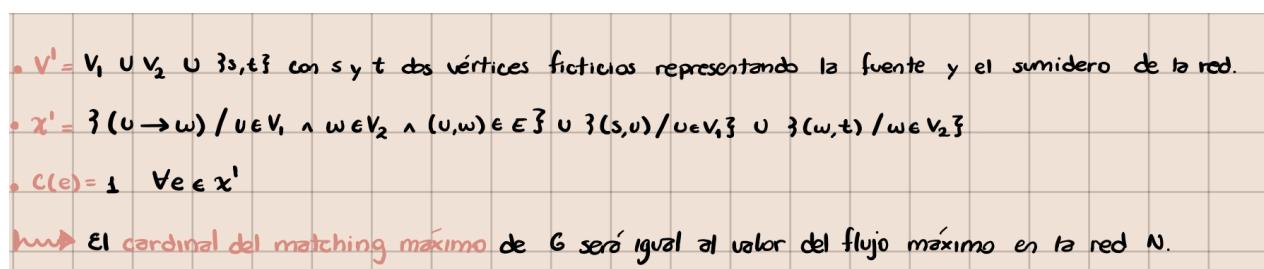
Dado un grafo $G = (V, E)$, un **matching** o correspondencia entre los vértices de G , es un conjunto $M \subseteq E$ de aristas de G tal que para todo $v \in V$, v es incidente a lo sumo a una arista $e \in M$ (Ningún vértice está en más de una arista del matching, o sea cada vértice se empareja con a lo sumo un vértice)

- El **matching máximo** consiste en encontrar aquel matching con la mayor cantidad de aristas posibles (cardinal máximo).

💡 Ejemplo: Supongamos que con un matching de 2 aristas ya no podés agregar más aristas sin repetir nodos. Entonces ese es un **matching máximo**.



El problema de **matching máximo** es computacionalmente fácil (se conocen algoritmos polinomiales) para grafos en general. Pero en el caso de grafo bipartito, podemos enunciar un algoritmo más simple transformándolo en un problema de flujo máximo en una red. Para esto, dado el grafo bipartito $G = (V_1 \cup V_2, E)$ definimos la siguiente red $N(V', X')$:



💡 Al poner capacidad 1, estamos obligando a que cada nodo se use a lo sumo una vez, igual que en un matching, el flujo que va de s a t corresponde a una asignación válida entre nodos de V_1 y V_2 . Entonces, el flujo máximo en esta red te da el matching máximo en el grafo.

💡 Como las capacidades son enteras, el flujo máximo va a ser un entero y como las aristas son 1, el flujo máximo es 1 o 0.

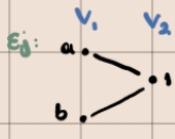
* Teorema de Hall: Si G es bipartito con partición (V_1, V_2) , entonces existe un matching que empareja a todos los vértices de $V_1 \Leftrightarrow \forall s \subseteq V_1, |s| \leq |N(s)|$

vecinos del conjunto s en V_2

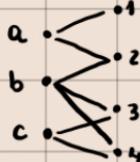
\hookrightarrow subconjunto de V_1

El teo dice que podemos emparejar a todos los V_1 con nodos de V_2 si el subconj. de V_1 tiene suficientes candidatos del otro lado para emparejarse

13/14



$$Ej: \quad S = \{a, b\} \rightsquigarrow N(S) = 1 \rightarrow |N(S)| = 1 < 2 = |S| \rightarrow \text{No se cumple Hall}$$



$$\left. \begin{array}{l} S = \{a\} \rightsquigarrow N(S) = 2 > 1 = |S| \\ S = \{a, b, c\} \rightsquigarrow N(S) = 4 > 3 = |S| \\ S = \{b, c\} \rightsquigarrow N(S) = 3 > 2 = |S| \end{array} \right\} \rightarrow \text{Se cumple Hall y el matching máximo es 3.}$$

(se cumple para todo S)

¿Cómo demostramos en flujo?

- Dado un problema P que se modela como una red de flujo N, queremos ver que si F es el valor del flujo máximo en N (encontrado con alguno de los métodos/algoritmos), coincide con la solución de P.
- Tenemos que probar el siguiente *si y solo si*: **Existe un flujo válido en N de valor F \iff Una solución posible para el problema P modelado con la red N tiene valor F.**
- Si probamos esta doble implicación entonces va a quedar probado que F es solución factible (y cuando es máximo es óptima), para lograr esto tenemos que probar ambos lados de la implicancia.

Existe un flujo válido en N de valor F \iff Una solución posible para el problema P modelado con la red N tiene valor F.

- **Restricciones de capacidad:** $f(e) \leq c(e) \quad \forall e \in E(N)$
- **Restricciones de conservación de flujo:**

$$\sum_{w \in N^-(v)} f(w \rightarrow v) = \sum_{w \in N^+(v)} f(v \rightarrow w) \quad \forall v \in V(D) - \{s, t\}$$

- f es un flujo de valor F : $\sum_{w \in N^+(s)} f(w) = \#\{i \mid s \in c_i\} = F$.
- Observación: para probar que un flujo F es máximo (es lo que queremos maximizar) basta con ver que en el corte mínimo el flujo f es exactamente igual a la suma de las capacidades del corte, ¿por qué?