Complejidad computacional:

Problema: La descripción de los datos de entrada y la Rta a proporcionar para cada dato de entrada

Instancia de un problema: Juego valido de datos de entrada

Suponemos una maquina RAM, es decir, memoria dada por una sucesión de celdas enumeradas donde cada celda puede almacenar b bits (y tamaño fijo, donde todos nuestros datos entran en esos b bits), también hay un programa imperativo no almacenado en memoria compuesto por asignaciones, y estructuras de control.

Costos:

- Acceder a cualquier celda de memoria, asignar y manejar estructuras de control, y operaciones lógicas cuestan O(1)

Las operaciones sobre enteros dependen de b:

- Sumas y restas O(b)
- Multiplicaciones y divisiones son O(b*log(b))

Obs: Si b esta fijo entonces las operaciones son O(1)

Tiempo de ejecución de un algoritmo A: El máximo de la suma de tiempos de ejecución de las instrucciones realizadas por el algoritmo.

Notación O:

Dadas f, g: $\mathbb{N} \to \mathbb{R}$

F(n) = O(g(n)) si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ / f(n) <= c* g(n) para todo n >= n_0

Luego F(n) = Ω(g(n)) si pasa lo mismo pero f(n) >= c * g(n)

Y F(n) = θ(g(n)) si F(n) = O(g(n)) = Ω(g(n))

Convención: Algoritmos polinomiales -> satisfactorios.

Caso contrario no satisfactorios (supra-polinomiales)

Problemas de optimización:

Consiste en encontrar la mejor solución dentro de un conjunto:

Z* = max f(x) con $x \in S$ (o el min)

La función f : S -> $\mathbb{R}$ es la función objetivo

El conjunto S es la región factible, y los elementos x de S son las soluciones factibles

El valor $Z^* \in \mathbb{R}$ es el valor optimo del problema, y cualquier solución factible x* que devuelva Z* es el óptimo del problema

Un problema de optimización combinatoria es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias, siendo la combinatoria una rama de la matemática que estudia las configuraciones de objetos finitos que satisfacen ciertas propiedades.

Algoritmo de fuerza bruta:

También llamado búsqueda exhaustiva o generate and test. Genera todas las soluciones factibles y se queda con la mejor. Suele ser fácil de implementar y es un algoritmo exacto: si hay solución, siempre la encuentra.

Lo negativo es su complejidad (suele ser exponencial)

Backtracking:

Dicha técnica hace una exploración ordenada del espacio de soluciones por medio de la extensión de soluciones parciales. Cada posible extensión de la sol parcial se explora haciendo recursión sobre la nueva sol extendida, esto se puede ver como un árbol y podemos aplicarle podas (descartar configuraciones antes de explorarlas) al árbol para reducir el espacio de búsqueda.

Podas por factibilidad: Evito explorar nodos no factibles

Podas por optimalidad: Evito explorar nodos subóptimos

Branch and bound: Uso la solución más óptima para comprar y ver si exploro eso o no (Backtracking y podas por optimalidad)

Habitualmente se utiliza un vector para representar la solución candidata donde en cada paso se va extendiendo

Programación dinámica:

En el caso de que haya superposición de problemas, por llamar a la función con los mismos parámetros, conviene guardar dichos resultados, dos enfoques:

Top down: Implementado recursivamente, donde se guarda el res en una estructura de datos (memorización), si se requiere del mismo dato, lo agarra de dicha estructura. Permite reconstruir la solución

Bottom up: Se realiza iterativamente, en este resolvemos todos los subproblemas más pequeños y guardamos todos los resultados, generalmente nos puede dejar utilizar menos memoria y no permitir reconstruir la solución.

La complejidad se reduce muchísimo, generalmente a algo pseudo-polinomial (acotado por un polinomio en los valores numéricos del input, en lugar de la longitud del mismo)

Divide and conquer:

Se basa en dividir un problema en problemas más pequeños del mismo tipo que el original, resolver dichos subproblemas y combinar las soluciones. Teorema maestro:

- Permite resolver relaciones de recurrencia de la forma:
  $T(n) = \begin{cases} a\ T(n/c)+f(n) & \text{si} \quad n>1 \\ 1 & \text{si} \quad n=1 \end{cases}$
- Si $f(n) = O(n^{\log_c a-\epsilon})$ para $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_c a})$
- Si $f(n) = \Theta(n^{\log_c a})$, entonces $T(n) = \Theta(n^{\log_c a} \log n)$
- Si $f(n) = \Omega(n^{\log_c a+\epsilon})$ para $\epsilon > 0$ y $af(n/c) < kf(n)$ para $k < 1$ y n suficientemente grandes, entonces $T(n) = \Theta(f(n))$

Ejemplos:

T(n) = 3* T(n/2) + θ(n). Caso 1: $\theta(n^{\log2(3)})$

T(n) = 2* T(n/2) + θ(n). Caso 2: θ(n*log n)

T(n) = T(n/2) + θ(n). Caso 3: θ(n)

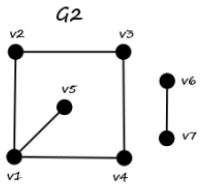
Algoritmos golosos:

Heurística: Proc computacional que intenta obtener sols precisas, de buena calidad. Ejemplo obtener un valor cercano al optimo en optimización. Algoritmo A es e-aproximado (e >= 0): $\left|\frac{x_A - x_{OPT}}{x_{OPT}}\right| \leq \epsilon.$

En golosos construís una solución seleccionando en cada paso la mejor alternativa, considerando nada o casi nada las implicancias de la misma. Estos proporcionan heurísticas sencillas para problemas de optimización, generando soluciones razonables (pero subóptimas) en tiempos eficientes

Grafos:

Un grafo G = (V, X) es un lar de conjuntos, donde V es un conjunto de puntos/nodos/vértices, y X es un subconjunto del conjunto de pares NO ordenados de elementos distintos de V.



$V_{G2} = \{v1, v2, v3, v4, v5, v6, v7\}$

$X_{G2} = \{(v1,v2),(v2,v3),(v3,v4),(v4,v1),(v1,v5),(v6,v7)\}$

Aristas/ejes: Elementos de X

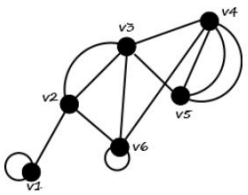
v y w son adyacentes si estos pertenecen a V, y su par pertenece a X. Se dice que el par (v, w) es incidente a v y w

Vecindad de v: N(v) es el conjunto de los vértices adyacentes a v

Notación: n = |V|. m = |X|

Multígrafo: Grafo donde puede haber varias aristas entre un mismo par de vértices distintos

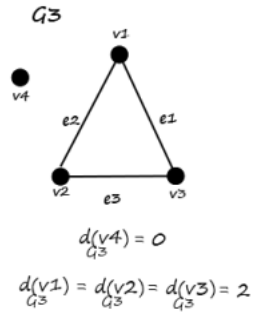
Pseudografo: Multígrafo donde también puede haber aristas que unan a un vértice consigo mismo (loop)



$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

$X = \{(v_1, v_1), (v_1, v_2), (v_2, v_3), (v_2, v_3), (v_2, v_6), (v_3, v_4), (v_3, v_5), (v_3, v_6), (v_4, v_5), (v_4, v_5), (v_4, v_5), (v_4, v_6), (v_6, v_6)\}$

Grado de un vértice v es la cantidad de aristas incidentes a v en G (cantidad de líneas conectadas a v)



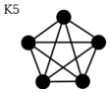
Notación:

Δ(G): Máximo grado de los vértices de G

δ(G): Mínimo grado de los vértices de G

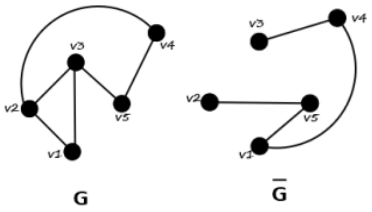
Teorema: La suma de los grados de los vértices de un grafo es igual a 2 $\sum_{i=1}^{n} d(v_i) = 2m$ veces su número de aristas

Corolario: La cantidad de vértices que tienen grado impar, es par



Grafo completo: Todos los vértices son adyacentes entre sí y se nota: K_n (grafo completo de n vértices)

Complemento de G = (V, X): $\bar{G}$/G^c = (V, X') tienen el mismo conjunto de vértices y: Un par de vértices son adyacentes en G^c <-> no lo son en G.



Recorrido: Secuencia alternada de vértices y aristas / los vértices deben estar conectados entre sí. En los grafos (no multi ni pseudo) está definido por la secuencia de vértices (conectados entre sí)

$P_1 = v_2 v_3 v_1 v_2 v_4$
es recorrido pero no camino

Camino: Es un recorrido que no pasa dos veces por el mismo vértice

$P_2 = v_1 v_2 v_5 v_4$
es camino

Sección de un camino: Una subsecuencia del mismo

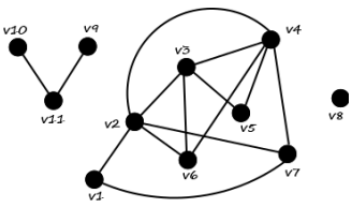
$C_1 = v_1 v_3 v_2 v_4 v_5 v_2 v_1$
es circuito, pero no ciclo

Circuito: Es un recorrido que empieza y termina en el mismo vértice.

Ciclo/circuito simple: Un circuito de 3 o más vértices que no pasa dos veces por el mismo vértice

$C_2 = v_2 v_3 v_5 v_4 v_2$
es ciclo

Longitud de recorrido P: l(P) = la cantidad de aristas que tiene



Distancia entre vértices c y w: d(v, w) es la longitud del recorrido más corto entre v y w. Si no existe = infinito. ∀ vértice d(v, v) = 0

Prop: Si un recorrido P entre v y w tiene longitud d(v, w), P es un camino

Props de la distancia para u, v, w pertenecientes a V:

- d(u, v) >= 0
- d(u, v) = 0 <-> u = v
- d(u, v) = d(v, u)
- d(u, w) <= d(u, v) + d(v, w)

$$d(v_1, v_2) = 1$$
$$d(v_1, v_3) = 2$$
$$d(v_1, v_5) = 3$$
$$d(v_3, v_7) = 2$$
$$d(v_3, v_9) = \infty$$
$$d(v_7, v_7) = 0$$
$$d(v_1, v_8) = \infty$$
$$d(v_{10}, v_9) = 2$$

Subgrafo:

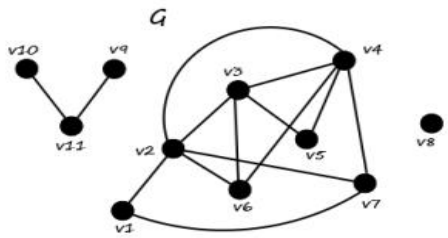
Dado G = (V_g, X_g), un subgrafo de G es un grafo H = (V_h, X_h) / $V_h \subseteq V_g$ y $X_h \subseteq X_g \cap (V_h \times V_h)$

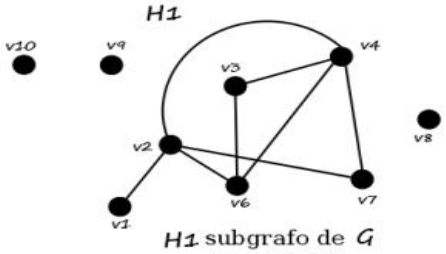
Si $H \subseteq G$ y $H \neq G$ entonces H es un subgrafo propio de G, $H \subset G$

H es un subgrafo generador de G si $H \subseteq G$ y $V_g = V_h$

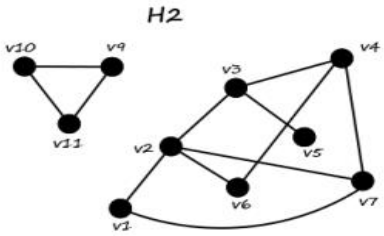
Un subgrafo H de G, es un subgrafo inducido si $\forall$ par u, v $\in V_h$ con (u, v) $\in X_g$ entonces también (u, v) $\in X_h$

Un subgrafo inducido de G = (V_g, X_g) por un conjunto de vértices V' ⊆ V_g, se denota como $G_{[V']}$

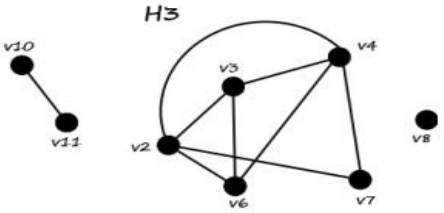




H1 subgrafo de G no inducido



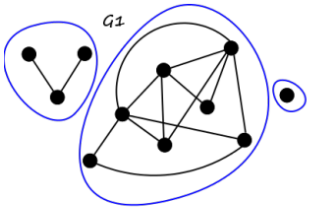
H2 no es subgrafo de G



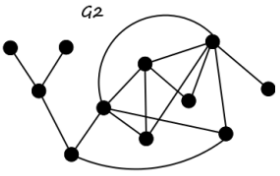
H3 subgrafo inducido de G por {v2,v3,v4,v6,v7,v8,v10,v11}

Grafo conexo: Si existe camino entre todo par de vértices

Componente conexa de G: Es un subgrafo conexo maximal de G



Grafo no conexo con 3 componentes conexas



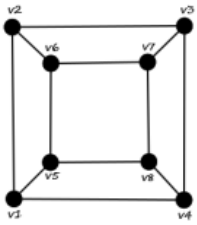
Grafo conexo

Grafo bipartito:

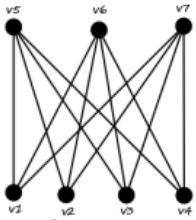
Si existen dos subconjuntos V_1, V_2 del conjunto de vértices V / V = V_1 U V_2 y que V_1 ∩ V_2 = Vacío

Se dice bipartito completo si todo vértice de V_1 es adyacente a todo vértice de V_2

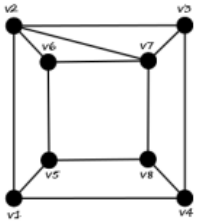
No bipartito



$V_1 = \{v_1, v_3, v_6, v_8\}$
$V_2 = \{v_2, v_4, v_5, v_7\}$



$V_1 = \{v_1, v_2, v_3, v_4\}$
$V_2 = \{v_5, v_6, v_7\}$



No bipartito

Teorema: Un grafo G es bipartito <-> no tiene ciclos de longitud impar

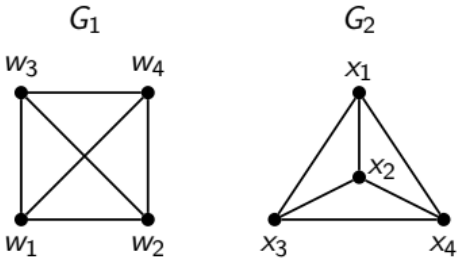
Isomorfismo de grafos:

Dos grafos G y G' son isomorfos si existe una función biyectiva f: V -> V' / ∀ v, w ∈ V:

(v, w) ∈ X <-> (f(v), f(w)) ∈ X'

f se le llama función de isomorfismo

Notación: G ≅ G'



$f(w_1) = x_1$ $f(w_2) = x_2$

$f(w_3) = x_3$ $f(w_4) = x_4$

Proposición:

Si G y G' son isomorfos entonces:

- Tienen mismo número de vértices y de aristas

- ∀k entre 0 y n-1 tienen el mismo número de vértices de grado k, y tienen el mismo número de caminos de longitud k

- Tienen el mismo número de componentes conexas

Obs: Hay grafos que cumplen todos los puntos y no son isomorfos

Representación de grafos:

Matriz de adyacencia:

$S \in \{0, 1\}^{nxn}$ donde los elems a_{ij} de A se definen como:

a_{ij} = 1 si G tiene una arista entre los vértices v_i y v_j. 0 en caso contrario

Props;

- La suma de los elems de la columna i de A (o l fila i, puesto que es simétrica) = $d(v_i)$

- Los elems de la diagonal de A^2 indican los grados de los vértices: $a_{ii}^2 = d(v_i)$

Dígrafos:

Un dígrafo G es un par de conjuntos V y X, donde V es el conjunto de nodos, y X es un subconjunto del conjunto de pares ordenados de elementos distintos de V. A los elems de X los llamaremos arcos

Dado el arco e = (u, w) llamamos al primer elem u, la cola de e, y al segundo w, la cabeza de e

Grado de entrada: $d_{in}(v)$ es la cantidad de arcos que llegan a v (tienen a v como cabeza)

Grado de salida $d_{out}(v)$ es la cantidad de arcos que salen de v (tienen a v como cola)

Grafo subyacente de G: se nota G^s y resulta de remover las direcciones

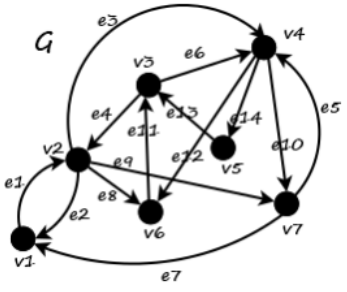
Matriz de adyacencia de un dígrafo:

aij = 1 si G tiene un arco de v_i a v_j. 0 si no.

Props:

- La suma de los elems de la fila i de A es igual a $d_{out}(vi)$

- La suma de los elems de la columna i de A es igual a $d_{in}(vi)$

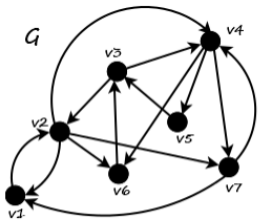


$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Recorrido/camino orientado: Sucesión de arcos / recorres en el sentido de las flechas

Circuito/ciclo orientado: Es un recorrido/camino orientado que comienza y termina en el mismo vértice

Un dígrafo fuertemente conexo si $\forall$ par de vértices u, v existen caminos orientados de u a v y de v a u



Es fuertemente conexo

$P_1 = v_1v_2v_4v_5v_3$ es camino orientado

$P_2 = v_2v_3v_4$ NO es camino orientado

$C_1 = v_1v_2v_4v_7v_1$ es ciclo orientado

$C_2 = v_2v_1v_7v_2$ NO es ciclo orientado

Grafo ralo: Un grafo con pocas aristas respecto a la cantidad de nodos

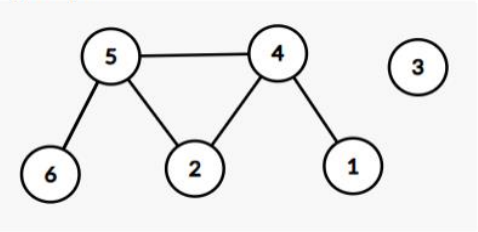
Grafo denso: un grafo con muchas aristas respecto a la cantidad de nodos

Representación de grafos:

lista de aristas

El conjunto de aristas como una secuencia (lista).

Tomemos el siguiente grafo como ejemplo:



Lista de aristas:

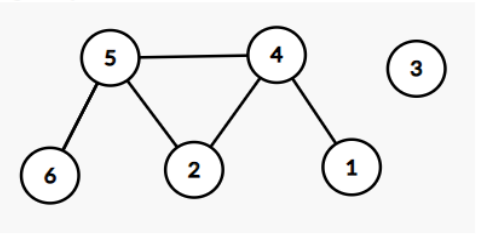
$\{(6,5),(5,2),(2,4),(5,4),(4,1)\}$

Nota 1: normalmente esta va a ser la única representación con la que se van a expresar los inputs de grafos.
Nota 2: si junto con la lista se pasa el tamaño del grafo, como por convención las etiquetas de los vértices están numeradas 1...n podemos deducir qué vértices no tienen vecinos.

Lista de adyacencia

El diccionario es un vector y los vecindarios son listas de tamaño d(v) conteniendo a los nodos vecinos.

Tomemos el siguiente grafo como ejemplo:



Lista de adyacencia:
Nodo : lista de vecinos

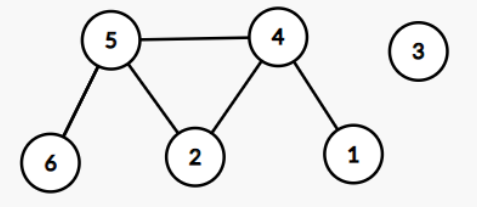
1 : 4
2 : 4 $\longrightarrow$ 5
3 :
4 : 5 $\longrightarrow$ 1 $\longrightarrow$ 2
5 : 2 $\longrightarrow$ 6 $\longrightarrow$ 4
6 : 5

Nota: Se pueden hacer cosas como ordenar los vecindarios, pero es caro mantenerlo si el grafo cambia.

Matriz de adyacencia

El diccionario y los vecindarios son vectores de tamaño n. Resultando así en una matriz de nxn donde $M_{ij} = 1$ si los vértices i y j son adyacentes y $M_{ij} = 0$ si no.

Tomemos el siguiente grafo como ejemplo:



Matriz de adyacencia:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Las complejidades para las representaciones vistas quedarían:

	lista de aristas	matriz de ady.	listas de ady.
construcción	$O(m)$	$O(n^2)$	$O(n+m)$
adyacentes	$O(m)$	$O(1)$	$O(d(v))$
vecinos	$O(m)$	$O(n)$	$O(d(v))$
agregarArista	$O(m)$	$O(1)$	$O(d(u)+d(v))$
removerArista	$O(m)$	$O(1)$	$O(d(u)+d(v))$
agregarVértice	$O(1)$	$O(n^2)$	$O(n)$
removerVértice	$O(m)$	$O(n^2)$	$O(n+m)$

Nota: como el tamaño del grafo está dado por su cantidad de nodos y aristas, una complejidad O(n+m) es lineal respecto al tamaño del grafo.

- La complejidad de pedir los vecinos queda en O(d(v)) porque, si bien podríamos devolver el puntero al inicio de la lista en O(1), eso rompe el encapsulamiento. Sería mejor devolver una copia del vecindario para garantizar que no se rompa la estructura externamente (ej. si le quitan elementos la lista perderíamos adyacencias o si es un grafo no dirigido se rompen invariantes).
- Si bien decimos que la complejidad de agregar vértices de una lista de adyacencia es O(n) por el costo de reinicializar el array, si se usaran arrays dinámicos quedaría en O(1) amortizado.

Arboles:

Un árbol es un grafo conexo sin circuitos simples.

Una arista e de G es puente si G − e tiene más componentes conexas que G.

Un vértice v de G es punto de corte o punto de articulación si G − v tiene más componentes conexas que G.

Teorema: Dado un grafo G = (V, X) son equivalentes:

1. G es un árbol.

2. G es un grafo sin circuitos simples, pero si se agrega una arista e a G resulta un grafo con exactamente un circuito simple, y ese circuito contiene a e.

3. Existe exactamente un camino entre todo par de nodos. 4. G es conexo, pero si se quita cualquier arista a G queda un grafo no conexo (toda arista es puente).

Lema 1: Sea G = (V, X) un grafo conexo y e $\in$ X. G − e es conexo si y solo si e pertenece a un circuito simple de G.

Definición: Una hoja es un nodo de grado 1.

Lema 2: Todo ´árbol no trivial tiene al menos dos hojas.

Lema 3: Sea G = (V, X) ´árbol. Entonces m = n − 1.

Corolario 1: Sea G = (V, X) sin circuitos simples y c componentes conexas. Entonces m = n − c.

Corolario 2: Sea G = (V, X) con c componentes conexas. Entonces m ≥ n − c.

Teorema: Dado un grafo G son equivalentes:

1. G es un árbol.

2. G es un grafo sin circuitos simples y m = n − 1.

3. G es conexo y m = n − 1.

Arboles enraizados

Un árbol enraizado es un árbol que tiene un vértice distinguido que llamamos raíz.

Explícitamente queda definido un árbol dirigido.

El nivel de un vértice es la distancia de la raíz a ese vértice.

La altura h de un árbol enraizado es el máximo nivel de sus vértices.

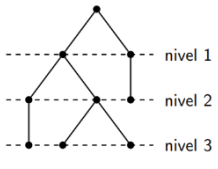
Los vértices internos de un árbol enraizado son aquellos que no son ni hojas ni la raíz.

Un árbol enraizado se dice m-ario si todos sus vértices internos tienen grado a lo sumo m + 1 y su raíz a lo sumo m.

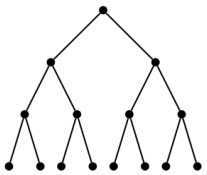
Un árbol enraizado se dice exactamente m-ario si todos sus vértices internos tienen grado m + 1 y su raíz m.

Un árbol se dice balanceado si todas sus hojas están a nivel h o h − 1.

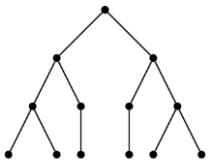
Un árbol se dice balanceado completo si todas sus hojas están a nivel h.



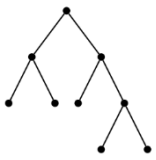
altura 3
2-ario
no exáctamente 2-ario



exáctamente 2-ario
balanceado completo



2-ario
balanceado completo



exáctamente 2-ario
balanceado
no balanceado completo

Decimos que dos vértices adyacentes tienen relación padre-hijo, siendo el padre el vértice de menor nivel.

Teorema:

- Un árbol m-ario de altura h tiene a lo sumo m^h hojas. Alcanza esta cota si es un árbol exactamente m-ario balanceado completo con $h \geq 1$.
- Un árbol m-ario con l hojas tiene $h \geq \lceil \log_m l \rceil$.
- Si T es un árbol exactamente m-ario balanceado no trivial entonces $h = \lceil \log_m l \rceil$.

Definición:

- Un **árbol generador** (AG) de un grafo G es un subgrafo generador (que tiene el mismo conjunto de vértices) de G que es árbol.

Teorema:

- Todo grafo conexo tiene (al menos) un árbol generador.
- G conexo. G tiene un único árbol generador $\iff$ G es árbol.
- Sea $T = (V, X_T)$ un AG de $G = (V, X)$ y $e \in X \setminus X_T$.
  Entonces $T' = T + e - f = (V, X_T \cup \{e\} \setminus \{f\})$, con f una arista del único circuito de $T + e$, T' es árbol generador de G.

Recorrido de árboles/grafos

A lo ancho (Breadth-First Search - BFS): se comienza por el nivel 0 (la raíz) y se visita cada vértice en un nivel antes de pasar al siguiente nivel.

En profundidad (Depth-First Search - DFS): se comienza por la raíz y se explora cada rama lo más profundo posible antes de retroceder.

```
recorrer(G)
    salida: pred[i] = padre de v_i, orden[i] = numero asignado a v_i
    next ← 1
    r ← elegir un vertice como raiz
    pred[r] ← 0                                   (marcar vertice r)
    orden[r] ← next
    LISTA ← {r}
    mientras LISTA ≠ ∅ hacer
        elegir un nodo i de LISTA
        si existe un arco (i,j) tal que j ∉ LISTA entonces
            pred[j] ← i                           (marcar vertice j)
            next ← next + 1
            orden[j] ← next
            LISTA ← LISTA ∪ {j}
        sino
            LISTA ← LISTA \ {i}
        fin si
    fin mientras
    retornar pred y orden
```

BFS y **DFS** difieren en el elegir:

- **BFS**: *LISTA* implementada como cola.
- **DFS**: *LISTA* implementada como pila.

El BFS sirve para calcular distancia:

- Agregar una matriz *dist* de tamaño $n \times n$.
- Inicializar $dist[i,j] \leftarrow \infty$ si $i \neq j$ sino $dist[i,i] \leftarrow 0$ para $1 \leq i \leq n$.
- Después de $pred[j] \leftarrow i$ dentro de la función $recorrer(G)$, se debe agregar $dist[r,j] \leftarrow dist[r,i] + 1$ (en caso de grafos, agregar también $dist[j,r] \leftarrow dist[r,j]$).
- Aplicar BFS para cada raíz $r : 1 \leq r \leq n-1$ en caso de grafos y $1 \leq r \leq n$ para digrafos.

DFS con más info:

Si aplicamos DFS para enumerar todos los vértices de un digrafo, se pueden clasificar sus arcos en 4 tipos:

- **tree edges**: arcos que forman el bosque DFS.
- **backward edges**: van hacia un ancestro.
- **forward edges**: van hacia un descendiente.
- **cross-edges**: van hacia a otro árbol (anterior) del bosque o al otra rama (anterior) del árbol.

Para grafos, solamente existen aristas tree edges y back edges !!!

Incorporamos algunos elementos adicionales para brindar más info.:

- una variable timer que se inicializa en 0.
- 2 arreglos: *desde* y *hasta*, ambos de dimensión n y se inicializan con -1.
- cada vez que un nodo nuevo i ingresa a *lista* que es una pila, se incrementa 1 el timer y se asigna dicho valor a $desde[i]$.
- cada vez que un nodo i sale de *lista*, se incrementa 1 el timer y se asigna dicho valor a $hasta[i]$.

El intervalo $(desde[i], hasta[i])$ representa el lapso de tiempo que el nodo i estuvo en la $lista - pila$. Para cualquier par de estos intervalos pasa una de las dos situaciones siguientes:

- hay una relación de contención entre ellos
- son disjuntos

Similitud con una secuencias de paréntesis !!!

Podemos usar esto y el arreglo $pred$ para determinar el tipo de un arco $i \to j$

- tree edges: si $i = pred[j]$.
- backward edge: si $(desde[j], hasta[j])$ contiene a $(desde[i], hasta[i])$.
- **forward edges**: si $(desde[i], hasta[i])$ contiene a $(desde[j], hasta[j])$ y $i \neq pred[j]$.
- **cross-edges**: si $hasta[j] < desde[i]$.

DFS puede servir para:

Detección de ciclos

Teorema: Dado un grafo (dígrafo) G. G tiene un ciclo (ciclo orientado) ⇔ existe un backward edge en G.

Sort topológico

Ordenar los nodos de acuerdo a su valor en el arreglo hasta de mayor a menor. ¿Funciona? ¿Realmente hay que ordenar?

Determinar las componentes fuertemente conexas de un dígrafo

Determinar los puntos de cortes y las aristas puentes de un grafo

Observaciones:

DFS es un algoritmo recursivo que sigue la idea de backtracking para poder recorrer todos los nodos. Lo podemos usar tanto para grafos dirigidos como para no dirigidos. Vamos a recorrer en profundidad (Depth): siempre vamos hasta el final de la rama y de ahí subimos.

La complejidad de DFS es O(m+n) ya que recorro todos los nodos una sola vez y reviso las aristas también una sola vez (aunque puede que no recorra todas). Sin agregarle complejidad al algoritmo de DFS podemos pedirle que nos devuelva el árbol o bosque (como un vector de padres) y la lista de backedges, que son las aristas que generan ciclos.

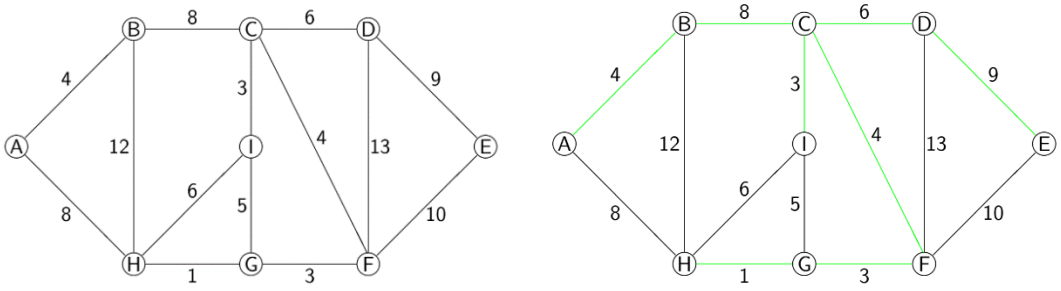
Con BFS también vamos a recorrer todos los nodos, pero en lugar de recorrer en profundidad va recorriendo a lo ancho. Se suele implementar iterativo. La idea del algoritmo es arrancar en algún nodo y recorrer todos ´ sus vecinos, luego los vecinos de sus vecinos...

La complejidad de BFS también es O(m+n), al igual que DFS. Nos devuelve un árbol v-geodésico, siendo v el nodo desde el cual corremos el algoritmo. Puede devolver también las distancias de v a todos en el grafo (¡que son iguales que el árbol ya que es v-geodésico!)

Árbol generador mínimo

Sea T = (V, X) un árbol, y I : X -> R una función que asigna pesos (longitudes a las aristas de T). Se define la longitud de T como $l(T) = \sum_{e \in T} l(e)$.

Dado un grafo conexo G = (V X) con una función I que asigna pesos a sus aristas, un árbol generador mínimo de G, T, es un árbol generador de G de mínima longitud, $l(T) \leq l(T') \; \forall T'$ árbol generador de G.

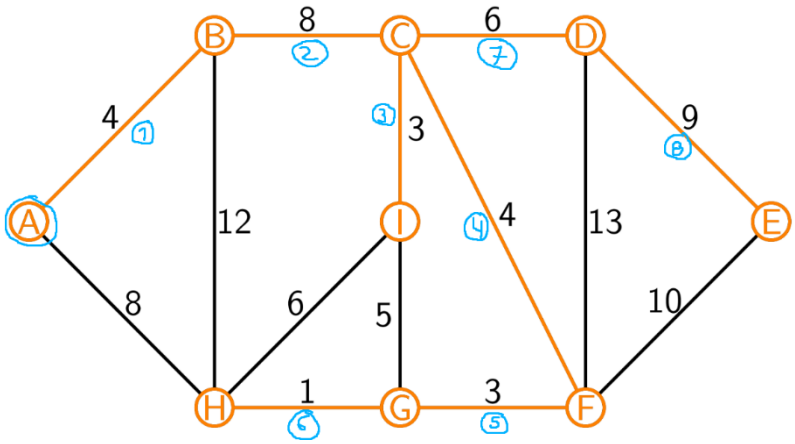


Algoritmo de Prim:

Entrada: $G = (V, X)$ grafo conexo con una función $l : X \to R$.

V_T := $\{u\}$ (u cualquier nodo de G)
X_T := $\emptyset$
i := 1
mientras $i \leq n - 1$ **hacer**
  elegir $e = (u, v) \in X$ tal que $l(e)$ sea mínima
    entre las aristas que tienen un extremo
    $u \in V_T$ y el otro $v \in V \setminus V_T$
  X_T := $X_T \cup \{e\}$
  V_T := $V_T \cup \{v\}$
  i := $i + 1$
retornar $T = (V_T, X_T)$

Empieza desde cierto nodo, y va eligiendo las aristas que tienen peso menor (y no genere ciclos con las que ya tengo), las cuales conectan a partir de los nodos en los cuales me voy moviendo



Empezando en “A”

En la iteración k se consigue un subgrafo de un AGM conexo con k aristas sin ciclos

Teorema: El algoritmo de Prim es correcto, y dado un grafo G conexo determina un árbol generador mínimo de G. A parte, es un algoritmo goloso.

Proposición: Sea $T_k = (V_{T_k}, X_{T_k})$ el árbol que genera Prim en la iteración K, para 0 <= K < n. T_K es un subárbol de un árbol generador mínimo de G.

Complejidad:

$O(n^2)$ implementación estándar (Esta se suele usar)

O((m+n) * log n) usando heap binario (esta tambien) Pero generalmente n<<m => O(m*log(n))

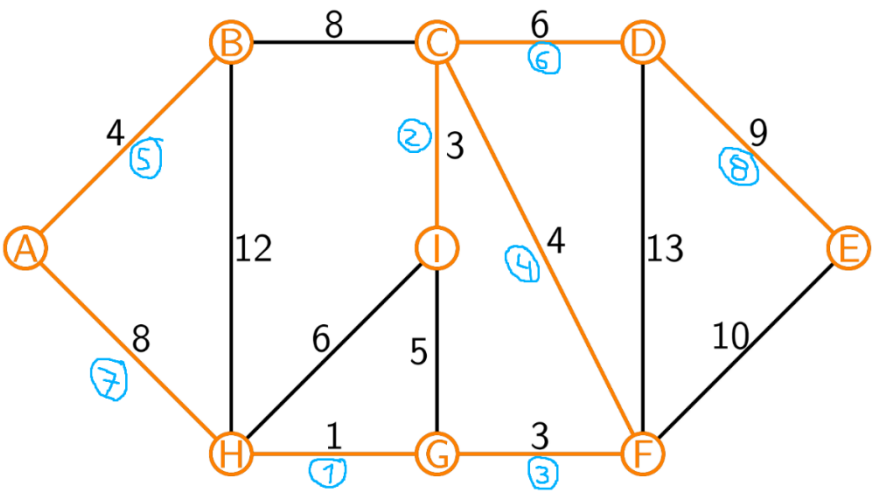
O(m + n * log n) usando heap Fibonacci

Algoritmo de Kruskal:

Entrada: $G = (V, X)$ grafo conexo con una función $l : X \to R$.

$X_T := \emptyset$
$i := 1$
mientras $i \leq n - 1$ **hacer**
  elegir $e \in X$ tal que $l(e)$ sea mínima entre las aristas que no forman circuito con las aristas que ya están en X_T
  $X_T := X_T \cup \{e\}$
  $i := i + 1$
retornar $T = (V, X_T)$

Va tomando siempre las aristas que tienen menor peso, las cuales no hagan que se genere un ciclo



Teorema: El algoritmo de Kruskall es correcto, Dado un grafo G conexo determina un AGM de G. A parte es un algoritmo goloso

Proposición: Sea G un grafo conexo. Sea $B_k = (V, X_{T_k})$ el bosque que el algoritmo de Kruskal genera en algún momento con k aristas, 0 <= k < n. B_K es un subgrafo generador sin ciclos de un árbol generador mínimo de G

Complejidad:

O(m*n) implementación trivial

O(m log n) con Union and find (por rango) (Esta se suele usar)

O(m log n + m * a(n)) con Union and find (por rango + compresión de camino)

Observación: Un problema AGM podría decirse que se resuelve en O(min{n², m * log n}) Es decir, si tengo un grafo denso, conviene usar O(n²), ya que m se va acercando a n².

Disjoint Set

Es una estructura de datos que nos provee las siguientes operaciones eficientemente:

- *find-set(u)*: Dado un vértice nos dice a qué componente conexa pertenece.
- *union(u,v)*: Une las componentes conexas a las que pertenecen u y v.

Disjoint Set Union

Todos los vértices empiezan teniendo a sí mismos como padres con tamaño de árbol 1.

Cuando queremos ver si dos vértices pertenecen al mismo subconjunto, seguimos la cadena de sus padres hasta llegar a algún nodo que sea padre de sí mismo (una raíz) y vemos si llegamos al mismo lugar.

Cuando unimos los conjuntos de un vértice u y otro v:

1. Encontramos las raíces de u y v (sean r_u y r_v)
2. Sea r_u la de mayor tamaño.
3. Ahora el padre de r_v es r_u y el tamaño de r_u se incrementó en el tamaño de r_v.

Uniendo con estos algoritmos, encontrar siempre es $O(log\ n)$.

Camino mínimo:

La longitud de un recorrido entre dos nodos v y u es la suma de las longitudes de las aristas del recorrido.

Un recorrido mínimo entre u y v es un recorrido entre u y v tal que tiene la mínima longitud

Si un recorrido mínimo entre dos nodos es un camino, entonces es un camino mínimo.

La distancia entre u y v, es la longitud de un camino mínimo entre u y v. En caso de no existir es ∞.

Tres variantes del problema de camino mínimo:

- Único origen – único destino: Camino mínimo entre dos vértices específicos (1)
- Único origen – múltiples destinos: Camino mínimo entre un vértice con todos los vértices (2)
- Múltiples orígenes – múltiples destinos: Camino mínimo entre todo par de vértices (3)

Obs: Si el grafo G no tiene ciclos de pesos negativos alcanzables desde v, entonces el problema está bien definido. En caso contrario el concepto de recorrido mínimo no está definido.

Propiedad: Todo subcamino de un camino mínimo entre dos nodos, también es camino mínimo entre los nodos que lo conforman.

Camino mínimo del tipo 1 y 2:

- Algoritmo de Dijkstra:

Este mantiene un array π de tamaño n, el cual, al finalizar la última iteración tendrá la longitud del camino mínimo entre un nodo v con el resto de los nodos.

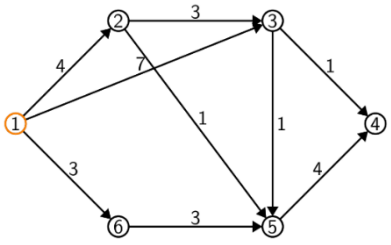
Este empieza en el nodo v, con π[u] = ∞ si u ≠ v, y π[u] = 0 en caso contrario. Luego completa los π[u] tales que (v, u) pertenece a X. Fija la distancia entre v y w tal que π[w] es mínima (es decir π[w] no volverá a cambiar), y nuevamente para todo u tal que (w, u) pertenece a X, π[u] = if (π[u] >= π[w] + peso((w, u))) then π[u] = π[w] + peso((w, u)) else skip. Finalmente, fijara la distancia entre v y z tal que π[z] es mínima y no es w. Repetirá el proceso hasta llegar a la última iteración.

Teorema: El algoritmo de Dijkstra es correcto <-> G no tiene aristas con peso negativo, y encuentra el camino mínimo entre todos los nodos y un nodo v fijo.

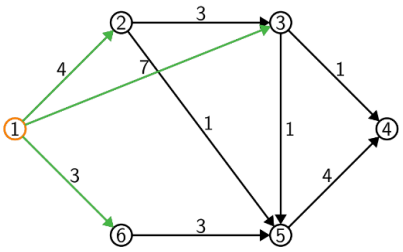
$S := \{v\}$, $\pi(v) := 0$, $pred(v) := 0$, $w := v$
para todo $u \in V$ **hacer**
  si $(v,u) \in X$ **entonces**
    $\pi(u) := l(v,u)$, $pred(u) := v$
  si no
    $\pi(u) := \infty$, $pred(u) := \infty$
  fin si
fin para
mientras $S \neq V$ **y** $\pi(w) < \infty$ **hacer**
  `elegir` $w \in V \setminus S$ `tal que` $\pi(w) = \min_{u \in V \setminus S} \pi(u)$
  $S := S \cup w$
  para todo $u \in V \setminus S$ **y** $(w,u) \in X$ **hacer**
    si $\pi(u) > \pi(w) + l(w,u)$ **entonces**
      $\pi(u) := \pi(w) + l(w,u)$
      $pred(u) := w$
    fin si
  fin para
fin mientras
retornar π, $pred$

Lema: Al finalizar la iteración k, Dijkstra determina el camino mínimo entre el nodo v y los nodos de S_K (Siendo S_K el conjunto S al finalizar la iteración k).

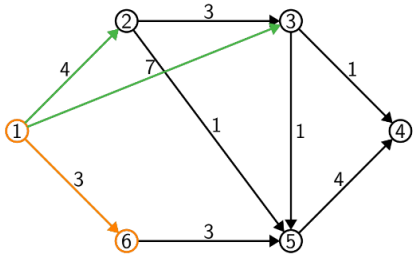
$S = \{1\}$ $\pi = (0, ?, ?, ?, ?, ?)$



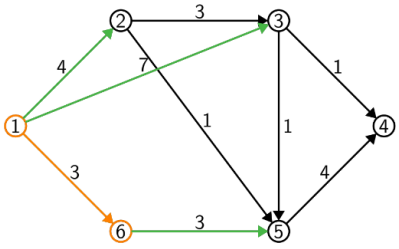
$S = \{1\}$ $\pi = (0, 4, 7, \infty, \infty, 3)$



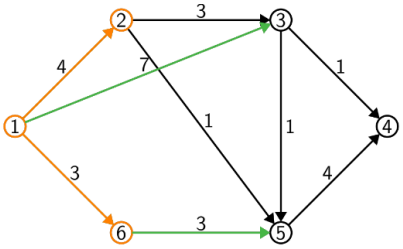
$S = \{1, 6\}$ $\pi = (0, 4, 7, \infty, \infty, 3)$



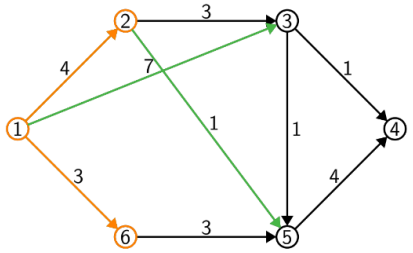
$S = \{1, 6\}$ $\pi = (0, 4, 7, \infty, 6, 3)$



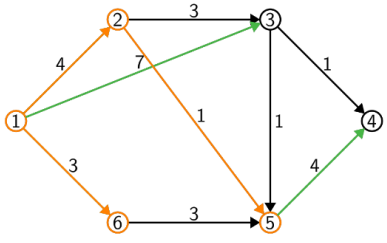
$S = \{1, 6, 2\}$ $\pi = (0, 4, 7, \infty, 6, 3)$



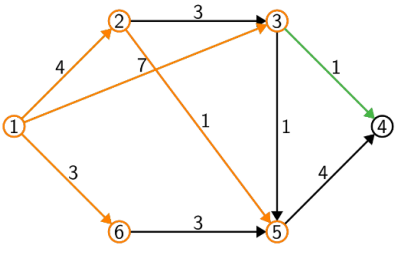
$S = \{1, 6, 2\}$ $\pi = (0, 4, 7, \infty, 5, 3)$



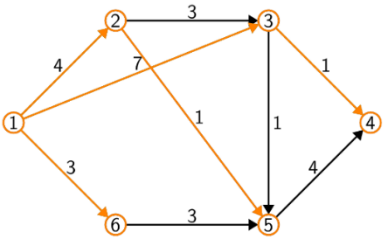
$S = \{1, 6, 2, 5\}$ $\pi = (0, 4, 7, 9, 5, 3)$



$S = \{1, 6, 2, 5, 3\}$ $\pi = (0, 4, 7, 8, 5, 3)$



$S = \{1, 6, 2, 5, 3, 4\}$ $\pi = (0, 4, 7, 8, 5, 3)$



Complejidad: (Igual a Prim) :

O(n²) implementación estándar

O((m+n) * log n) usando heap binario. Pero generalmente n<<m => O(m*log(n))

O(m + n * log n) usando heap Fibonacci

- Algoritmo de Bellman-Ford

Este algoritmo utiliza nuevamente un array π, en donde, en una primera iteración setea en infinito los índices donde dicho índice es distinto de v, y 0 en caso de que sea v.

Luego, mantiene un segundo array π' (que refleja un π antiguo), el cual será igual a π en primera instancia. Después mientras π ≠ π' (salvo apenas entra), para todo u ≠ v, y para todo w / (w, u) pertenece a X, π[u] = mínimo entre π[u] y los π[w] + peso((w, u))

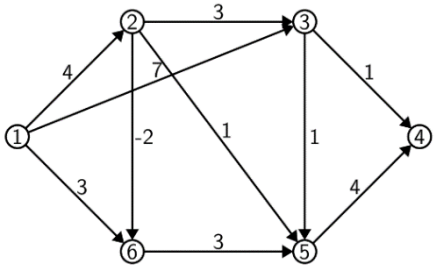
Teorema: El algoritmo de Bellman-Ford es correcto para un grafo orientado G sin ciclos de longitud negativa alcanzables desde v, determinando un camino mínimo entre v y cada nodo alcanzable desde v.

Complejidad: O(n*m)

$\pi(v) := 0$
para todo $u \in V \setminus \{v\}$ **hacer**
    $\pi(u) := \infty$
fin para
mientras `hay cambios en` π **hacer**
    $\pi' := \pi$
    para todo $u \in V \setminus \{v\}$ **hacer**
        $\pi(u) := \min(\pi(u), \min_{(w,u)\in X} \pi'(w) + l(w,u))$
    fin para
fin mientras
retornar π

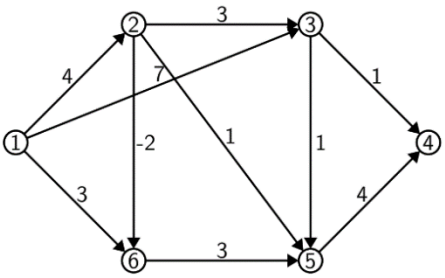
Iteración 1 $\pi = (0, \infty, \infty, \infty, \infty, \infty)$

$\pi' = (0, \infty, \infty, \infty, \infty, \infty)$



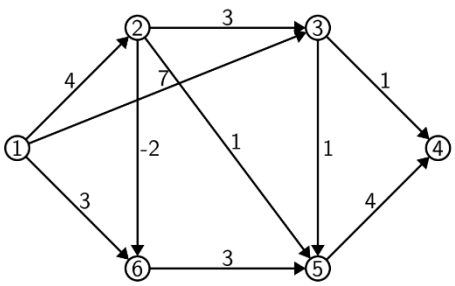
Iteración 1 $\pi = (0, 4, 7, \infty, \infty, 3)$

$\pi' = (0, \infty, \infty, \infty, \infty, \infty)$



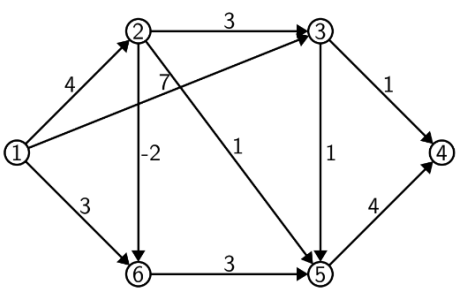
Iteración 2 $\pi = (0, 4, 7, \infty, \infty, 3)$

$\pi' = (0, 4, 7, \infty, \infty, 3)$



Iteración 2 $\pi = (0, 4, 7, 8, 5, 2)$

$\pi' = (0, 4, 7, \infty, \infty, 3)$



Lema 1: En todo momento de la ejecución de Ford:

- Si π(w) < ∞ para algún nodo w entonces existe un recorrido que conecta v con w y π(w) = Longitud de dicho recorrido.

- Si existe un camino mínimo entre v y w entonces π(w) >= distancia(v, w)

Lema 2: Si C es un camino entre v y w con k aristas entonces al finalizar la iteración k del algoritmo de Ford (puede ocurrir antes): π(w) <= L(C)

Corolario 1: Al finalizar la iteración k del algoritmo de Ford, este determina un camino mínimo entre v y w si existe un camino mínimo de v a w con a lo sumo k aristas.

Corolario 2: Si hubo cambio de π hasta la iteración n inclusive en la ejecución, entonces existe un ciclo de longitud negativa alcanzable desde v.

Proposición 1: Si existe un ciclo de longitud negativa alcanzable desde v entonces hay cambio de π en toda iteración del algoritmo de Ford.

Modificación para detectar ciclos de peso negativo: Va pisando los valores de π, no necesita un π'. También pongo un iterador i, que corte el ciclo si i == n. Luego si hay cambios en π devuelvo que hay ciclos de longitud negativo.

```
    π(v) := 0, i := 0
    para todo u ∈ V \ {v} hacer
        π(u) := ∞
    fin para
    mientras hay cambios en π e i < n hacer
        i := i + 1
        para todo u ∈ V hacer
            π(u) := mín(π(u), mín_{(w,u)∈X} π(w) + l(w, u))
        fin para
    fin mientras
    si hay cambios en π entonces
        retornar "Hay ciclos de longitud negativa
alcanzable desde v."
    si no
        retornar π
    fin si
```

Modificaciones que Ford me genere el árbol de caminos mínimos: Mantengo un arreglo como en Dijkstra, donde se actualiza pred(u) = w cada vez que π(u) es mejorado al valor de π(w) + l(w, u)

Modificaciones para generar el ciclo de longitud negativa si hubiera uno: Implemento lo anterior, y en caso de detectar dicho ciclo, considerar solamente las aristas (pred(u), u) siempre que pred(u) sea nodo del grafo (hay a lo sumo n aristas). Aplicar DFS a partir de v, en caso de encontrar un back-edge, localizamos el ciclo y necesariamente es de longitud negativa.

Si no hay back-edge, entonces DFS devolvió un árbol con raíz v, entonces tomamos una arista (pred(w), w) fuera de este árbol, y aplicamos DFS a partir de w pero invirtiendo las orientaciones de las aristas ((u, pred(u))). Necesariamente hay un backedge y localizamos el ciclo (volver a tomar las orientaciones originales y sigue siendo ciclo, el cual es de longitud negativa necesariamente).

Proposición 2: G tiene un ciclo de longitud negativo alcanzable desde v <-> el subgrafo G' de G con solamente aristas de (pred(u), u) tiene un ciclo con al menos 2 aristas.

Corolario 3: Después de aplicar la última versión del algoritmo de Ford a un grafo pesado G y un nodo de origen v.

G* = Un subgrafo de G' (el de prop 2) con los nodos alcanzables desde v

1- dist(v, w) = ∞ <-> π(w) = ∞ (w inalcanzable desde v)

2- dist(v, w) = -∞ (No existe recorrido mínimo entre v y w) <-> π(w) < ∞ y se cumple alguna de las siguientes condiciones:

a- pred(v) es un nodo de G

b- w es inalcanzable desde v en inalcanzable desde v en G* (v y w están en diferentes componentes conexas del grafo subyacente de G*)

c- w es alcanzable en G desde un nodo u ≠ w / dist(v, u) = -∞

Implementación

1- Trivial en O(n)

2- Buscar los nodos w de G* / dist(v, w) = -∞

2.1- si pred(v) ≠ ∞ entonces para cualquier nodo w de G* dist(v, w) = -∞ ya que w está en un ciclo de longitud negativa alcanzable desde v o es alcanzable desde un nodo que si está en dicho ciclo (aplico pred iterativamente)

2.2- Si pred(v) = ∞. Los nodos que no estén en la componente conexa de v (fuera del árbol T que puede generar v) están en un ciclo de longitud negativa alcanzable desde v o desde otro nodo que si lo está (aplicando pred iterativamente). Entonces dist(v, w) = -∞. Usar DFS desde v para calcular T con aristas de G*. O(n)

2.3 Para hallar los nodos w de T tal que dist(v,w) = −∞, hay que ver cuales nodos de T son alcanzables en G desde nodos de G* fuera de T. Esta condición también es

necesaria ya que los valores de π de nodos de T no sufrieron cambios en la iteración n. Para que vuelva a cambiar π(w) para algún nodo w de T, ese cambio debe ser causado (directamente o no) por el cambio de π(u) de un nodo u de G* fuera de T y en tal caso u alcanza a w en G. Para simplificar este cálculo, agregamos un nodo nuevo v* y agregar una arista (v* , u) para cada nodo u de G* fuera de T. Este grafo resultante tiene un nodo extra y a lo sumo n − 1 aristas más que G en comparación. Luego, aplicar DFS desde v* para ver que nodos de T se pueden alcanzar y esos nodos son los que buscamos. O(m + n)

Camino mínimo tipo 3 (Algoritmos matriciales)

Sea $G = (\{1, \ldots, n\}, X)$ un digrafo y $l : X \to R$ una función de longitud/peso para las aristas de G. Definimos las siguientes matrices:

- $L \in R^{n \times n}$, donde los elementos l_{ij} de L se definen como:

$$l_{ij} = \begin{cases} 0 & \text{si } i = j \\ l(i \to j) & \text{si } i \to j \in X \\ \infty & \text{si } i \to j \notin X \end{cases}$$

- $D \in R^{n \times n}$, donde los elementos d_{ij} de D se definene como:

$$d_{ij} = \begin{cases} \text{longitud del camino mínimo orientado de } i \text{ a } j & \text{si existe alguno} \\ \infty & \text{si no} \end{cases}$$

D es llamada matriz de distancias de G.

Algoritmo de Floyd

1- Si $L^0 = L$ y calculamos L^1 como:

$l_{ij}^1 = \min(l_{ij}^0, l_{i1}^0, l_{1j}^0)$

l_{ij}^1 es la longitud de un camino mínimo de i a j con nodo intermedio v_1 o directo

2- Si calculamos L^k a partir de L^{k-1} como:

$L_{ij}^k = \min(l_{ij}^{k-1}, l_{ik}^{k-1} + l_{kj}^{k-1})$

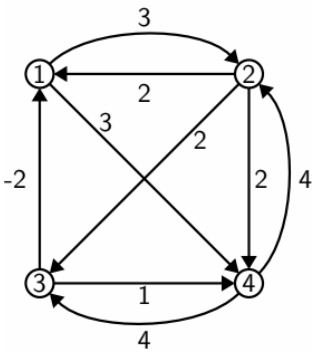
l_{ij}^k es la longitud de un camino mínimo de i a j cuyos nodos intermedios están en $\{v_1, \ldots, v_k\}$

3- $D = L^n$

Si no hay ciclos de longitud negativa:

$L^0 := L$
para k **desde** 1 **a** n **hacer**
  para i **desde** 1 **a** n **hacer**
    para j **desde** 1 **a** n **hacer**
      $l_{ij}^k := \min(l_{ij}^{k-1}, l_{ik}^{k-1} + l_{kj}^{k-1})$
    fin para
  fin para
fin para
retornar L^n

Ejemplo:



$L^0 =$

	1	2	3	4
1	0	3	∞	3
2	2	0	2	2
3	-2	∞	0	1
4	∞	4	4	0

$L^1 =$

	1	2	3	4
1	0	3	∞	3
2	2	0	2	2
3	-2	1	0	1
4	∞	4	4	0

$L^2 =$

	1	2	3	4
1	0	3	5	3
2	2	0	2	2
3	-2	1	0	1
4	6	4	4	0

$L^3 =$

	1	2	3	4
1	0	3	5	3
2	0	0	2	2
3	-2	1	0	1
4	2	4	4	0

$D = L^4 =$

	1	2	3	4
1	0	3	5	3
2	0	0	2	2
3	-2	1	0	1
4	2	4	4	0

Lema: Al finalizar la iteración k, l_{ij} es la longitud de los caminos mínimos de v_i a v_j cuyos nodos intermedios son elementos de $V_k = \{v_1, \ldots, v_k\}$, si no existe ciclo de longitud negativa con todos sus vértices en V_K

Teorema: El algoritmo de Floyd es correcto entre todos los pares de nodos en un grafo orientado sin ciclos negativos.

Complejidad temporal: $O(n^3)$

Complejidad espacial: $O(n^2)$ se puede

Lema: si G tiene ciclo de longitud negativa entonces $d_{ii}^n < 0$ para algún i.

Sin embargo, conviene utilizar Bellman-Ford para ver si tiene ciclos de longitud negativa, puesto que solamente coloco un vértice fantasma que vaya a todos con costo 0 y corro Bellman Ford, es más barato.

Otras implementaciones: (next para reconstruir dicho camino mínimo)

Input: $G = (V, E)$
Output: d
for $i, j = 1$ ***to*** n ´ **do**
  $d[i][j][0] := w_{ij}$;
for $k \leftarrow 1$ ***to*** n **do**
  for $i \leftarrow 1$ ***to*** n **do**
    for $j \leftarrow 1$ ***to*** n **do**
      if $d[i][j][k-1] > d[i][k][k-1] + d[k][j][k-1]$ **then**
        $d[i][j][k] := d[i][k][k-1] + d[k][j][k-1]$;
        $next[i][j] := k$;

$L^0 := L$
para k **desde** 1 **a** n **hacer**
  para i **desde** 1 **a** n **hacer**
    si $l_{ik}^{k-1} \neq \infty$ **entonces**
      si $l_{ik}^{k-1} + l_{ki}^{k-1} < 0$ **entonces**
        retornar "Hay ciclos negativos."
      fin si
      para j **desde** 1 **a** n **hacer**
        $l_{ij}^{k} := \min(l_{ij}^{k-1}, l_{ik}^{k-1} + l_{kj}^{k-1})$
      fin para
    fin si
  fin para
fin para
retornar L

Algoritmo de Dantzig

Lema: Al finalizar la iteración k-1, el algoritmo de Dantzig genera una matriz de k x k de caminos mínimos en el subgrafo inducido por los vértices $\{v_1, \ldots, v_k\}$

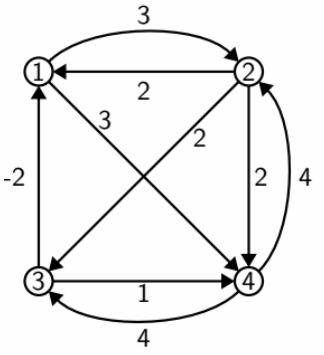
Cada vez que agrego otro k, puede caer en tres casos. Que tenga que buscar el camino mínimo de alguien que ya está en el subgrafo inducido con alguien que no está, luego otro caso, y finalmente que los dos estén en el subgrafo inducido y agregue un nuevo.

Calcula la matriz L^{k+1} a partir de la matriz L^k para $1 \leq i, j \leq k$ como:

- $L^{k+1}_{i,k+1} = \text{mín}_{1 \leq j \leq k}(L^k_{i,j} + L^k_{j,k+1})$
- $L^{k+1}_{k+1,i} = \text{mín}_{1 \leq j \leq k}(L^k_{k+1,j} + L^k_{j,i})$
- $L^{k+1}_{i,j} = \text{mín}(L^k_{i,j}, L^k_{i,k+1} + L^k_{k+1,j})$

- Distancia de $i \in \{1, \ldots, k\}$ a $k+1$: hay que ir hasta un j y luego tomar el eje $j \to k+1$. Es decir, $D^{k+1}_{i,k+1} = \text{mín}_{1 \leq j \leq k}(D^k_{i,j} + w_{j,k+1})$.
- Distancia de $k+1$ a $i \in \{1, \ldots, k\}$: análogo.
- Distancia de i a j, ambos menores a $k+1$: uso el camino viejo que no pasaba por $k+1$, o bien paso por $k+1$ combinando dos caminos óptimos. Es decir, $D^{k+1}_{i,j} = \text{mín}(D^k_{i,j}, D^{k+1}_{i,k+1} + D^{k+1}_{k+1,j})$.

Ejemplo:



$L =$

	1	2	3	4
1	0	3	∞	3
2	2	0	2	2
3	-2	∞	0	1
4	∞	4	4	0

$k = 1$

$L =$

	1	2	3	4
1	0	3	∞	3
2	2	0	2	2
3	-2	∞	0	1
4	∞	4	4	0

$k = 2$

$L =$

	1	2	3	4
1	0	3	5	3
2	2	0	2	2
3	-2	1	0	1
4	∞	4	4	0

$k = 3$

$L =$

	1	2	3	4
1	0	3	5	3
2	0	0	2	2
3	-2	1	0	1
4	2	4	4	0

$k = 4$

$L =$

	1	2	3	4
1	0	3	5	3
2	0	0	2	2
3	-2	1	0	1
4	2	4	4	0

Teorema: El algoritmo de Dantzig determina los caminos mínimos entre todos los pares de nodos de un grafo orientado sin ciclos de longitud negativa

Complejidad temporal: $O(n^3)$

Complejidad espacial: $O(n^2)$ se puede

```
para k desde 1 a n − 1 hacer
    para i desde 1 a k hacer
        L_{i,k+1} := mín_{1≤j≤k}(L_{i,j} + L_{j,k+1})
        L_{k+1,i} := mín_{1≤j≤k}(L_{k+1,j} + L_{j,i})
    fin para
    t := mín_{1≤i≤k}(L_{k+1,i} + L_{i,k+1})
    si t < 0 entonces
        retornar "Hay ciclos de longitud negativa"
    fin si
    para i desde 1 a k hacer
        para j desde 1 a k hacer
            L_{i,j} := mín(L_{i,j}, L_{i,k+1} + L_{k+1,j})
        fin para
    fin para
fin para
retornar L
```

Obs: Si quiero obtener camino mínimo todos contra todos puedo usar los algoritmos de camino mínimo uno contra todos n veces:

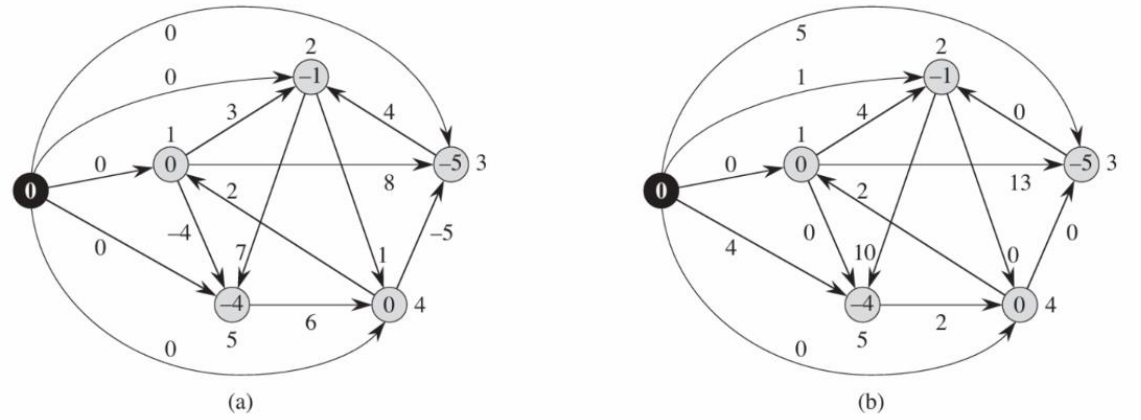
Dijkstra: O(n*m*log(n))

Bellman-Ford: $O(n^2*m)$

El segundo siempre es malo, pero el primero le gana a $O(n^3)$ en grafos ralos. Pero solo se puede usar si no hay aristas negativas.

Algoritmo de Johnson

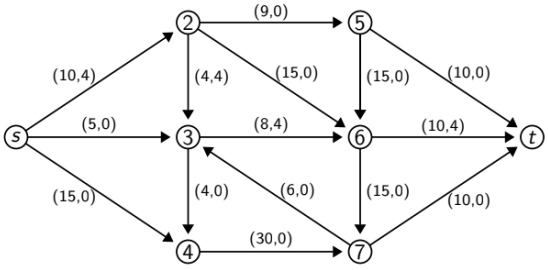
Modifico el costo en los nodos de tal forma que los pesas sean todos positivos, y aplico Dijkstra n veces. Para saber por qué valor modificarlo, pongo un nodo fantasma con valor 0 conectado a todos, aplico Bellman-Ford obteniendo las distancias mínimas h(v), modificación: w'(u,v)=w(u,v)+h(u)−h(v).



Flujo

Dado un grafo dirigido G = (N, A), nodos s, t ∈ N de origen y destino, y una función de capacidad u asociada a los arcos. Se desea encontrar un flujo (cantidad a enviar por cada arco) entre s y t de mayor valor posible.

$F = 4$



Obs: Salvo s y t, la cantidad de flujo que entra a un nodo debe ser igual a la que sale. La cantidad de flujo enviada por un arco debe estar <= 0, y >= capacidad de dicho arco, y el valor de un flujo es la cantidad de flujo neto que sale de s.

Un corte en la red G = (N, A) es un subconjunto S ⊆ N \ {t} tal que s ∈ S.

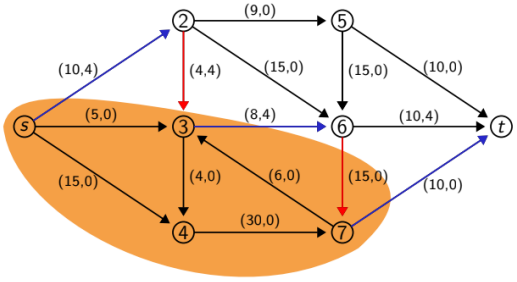
Dados dos cortes S y C ⊆ N, defino SC = {ij ∈ A : i ∈ S y j ∈ C}

Proposicion: Sea x un flujo definido en una red, y S un corte entonces:

$$F = \sum_{ij \in S\bar{S}} x_{ij} - \sum_{ij \in \bar{S}S} x_{ij} \quad \text{donde } \bar{S} = N \setminus S.$$

Es decir F es la suma del flujo que sale del corte, menos la suma de lo que entra.

$F = 4$



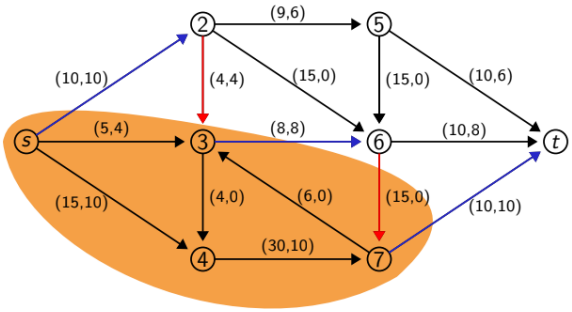
La capacidad de un corte S es: $u(S) = \sum_{ij \in S\bar{S}} u_{ij}.$

Proposicion: Si x es un flujo con valor F, y S es un corte en N, entonces F <= u(S)

Corolario: Si F = u(S) entonces x es un flujo maximo y S un corte de capacidad minima.

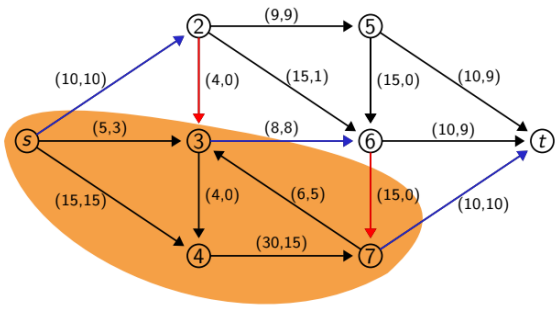
$U = 28$

$F = 24$



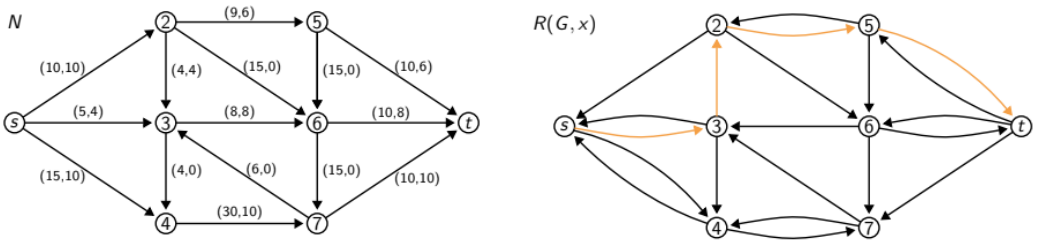
$U = 28$

$F = 28$



Dada una red G = (N, A) con función de capacidad u, y un flujo factible x, defino la red residual R(G, x) = (N, A_R) con: ij ∈ A_R si $x_{ij} < u_{ij}$, y ji ∈ A_R si $x_{ij} > 0$. Es decir, pongo las aristas donde el flujo que mando por ahí es menor que la capacidad, y además pongo una arista apuntando hacia el otro lado si el flujo que pasa en el grafo original es > 0.

Un camino de aumento es un camino orientado de s a t en la red residual.

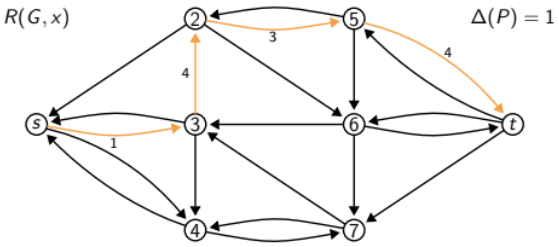


Luego, dado un camino de aumento P, para el arco ij ∈ P:

$$\Delta(ij) = \begin{cases} u_{ij} - x_{ij} & \text{si } ij \in A \\ x_{ji} & \text{si } ji \in A \end{cases}$$

Además defino: $\Delta(P) = \min_{ij \in P}\{\Delta(ij)\}$.

Podemos encontrar un camino de aumento en P en la red residual en O(m), y calcular Δ(P) en O(n).



Proposición: Sea x un flujo definido sobre una red N con valor F y sea P un camino de aumento en la red residual. Entonces el flujo $\bar{x}$, definido por:

$$\bar{x}(ij) = \begin{cases} x_{ij} & \text{si } ij \notin P \wedge ji \notin P \\ x_{ij} + \Delta(P) & \text{si } ij \in P \\ x_{ij} - \Delta(P) & \text{si } ji \in P \end{cases}$$

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

Es decir, si dejamos el flujo como esta si no pertenece al camino de aumento. Le sumamos la capacidad del camino si esa arista ij pertenece al camino, y finalmente le restamos la capacidad del camino si la arista ji pertenece a dicho camino

Teorema: Sea x un flujo definido sobre una red N. Entonces x es un flujo máximo <-> no existe camino de aumento en la red residual

Teorema: Dada una red N, el valor del flujo máximo = capacidad del corte mínimo.

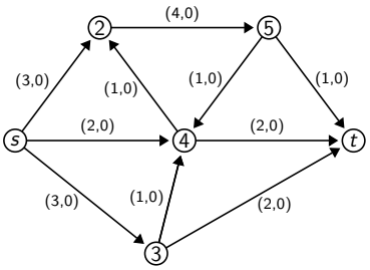
Algoritmo de Ford y Fulkerson:

Obtiene un flujo máximo en O(m*n*U) con U la arista con capacidad máxima. También se puede acotar más por: O(m*F) con F el flujo máximo (ya que se incrementa 1 en cada iteración en el peor caso).

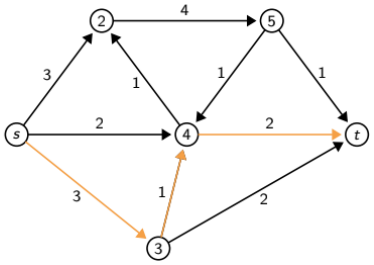
Y se tiene que F <= U*n

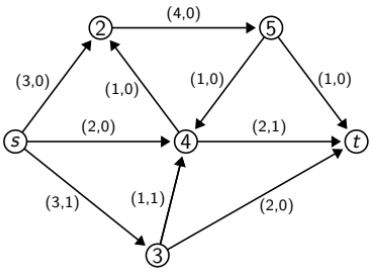
(Es m porque depende como se implemente la búsqueda de caminos de aumento)

Definir un flujo inicial en N (por ejemplo, $x = 0$)
mientras exista $P :=$ camino de aumento en $R(G, x)$ **hacer**
  para cada arco $ij \in P$ **hacer**
    si $ij \in A$ **entonces**
      $x_{ij} := x_{ij} + \Delta(P)$
    si no $(ji \in A)$
      $x_{ji} := x_{ji} - \Delta(P)$
    fin si
  fin para
fin mientras

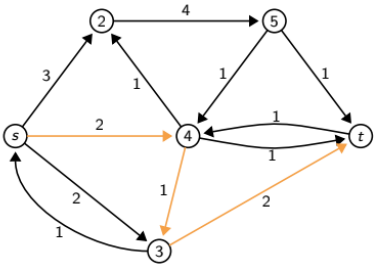


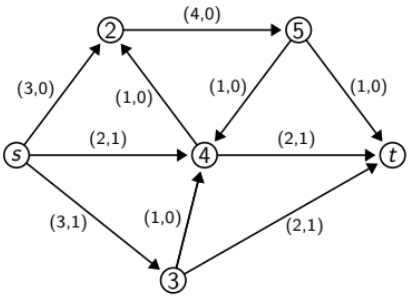
$R(G,x)$



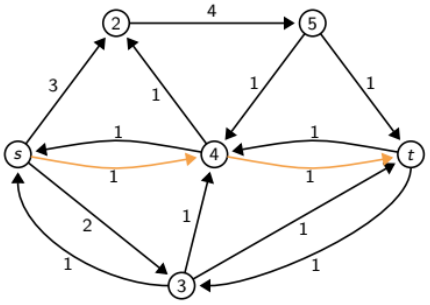


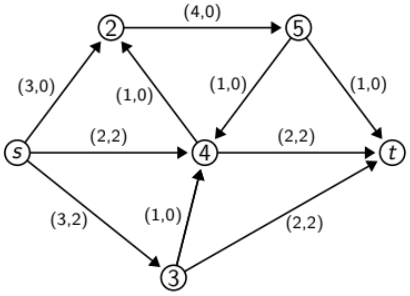
$R(G,x)$



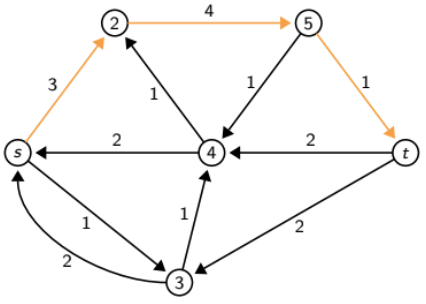


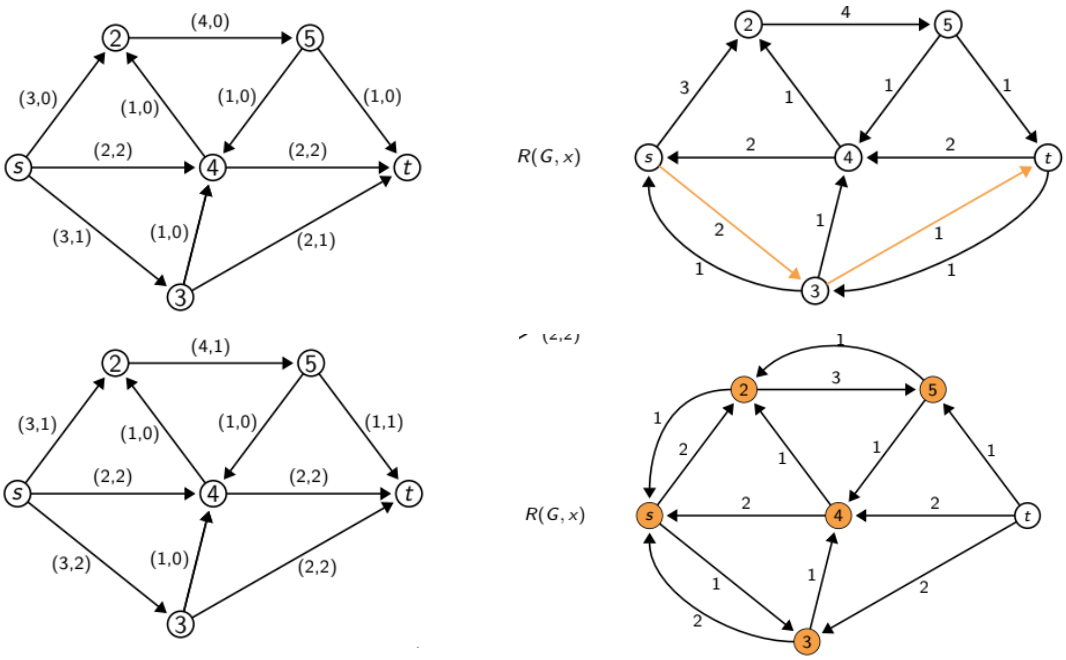
$R(G,x)$





$R(G,x)$





Teorema: Si las capacidad de los arcos de la red son enteras, entonces el problema de flujo máximo tiene un flujo entero

Teorema: Si los valores del flujo inicial y las capacidades de los arcos son enteras, entonces hará a lo sumo n*U iteraciones, o más acotadamente F iteraciones.

Si las capacidades o el flujo inicial son números irracionales, dicho método puede no terminar.

Algoritmo de Edmonds y Karp:

Es una modificación del algoritmo anterior, la cual consiste en utilizar BFS para buscar caminos de aumento.

Esto lo resuelve con complejidad $O(n*m^2)$, aunque en realidad podemos decir que lo resuelve en $\min\{n*m^2, m*F\}$.

Matching máximo en grafos bipartitos

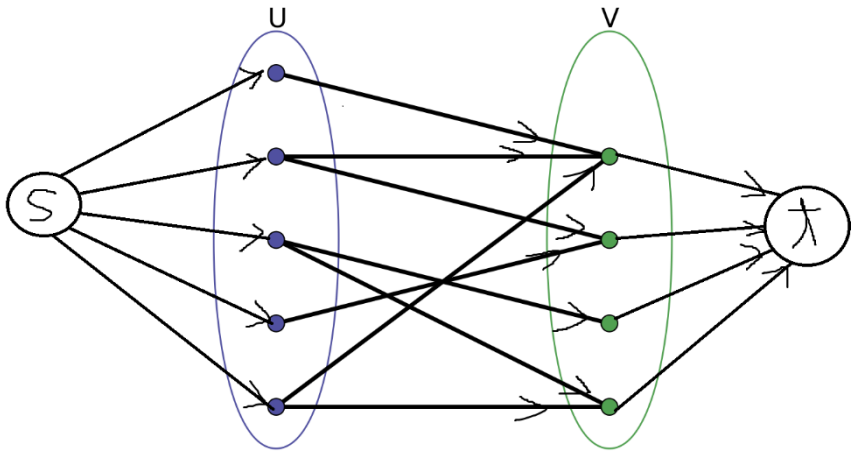
Un matching entre los vértices de G, es un conjunto M ⊆ E de aristas de G / para todo v ∈ V, v es incidente a lo sumo a una arista de M

El problema de matching máximo consiste en encontrar un matching de cardinal máximo entre todos los matchings de G. Este es resoluble en tiempo polinomial para todo grafo, pero nos centraremos en los grafos bipartitos donde es más simple, ya que lo transformamos a un problema de flujo máximo en una red.

Dado el grafo bipartito $G = (V_1 \cup V_2, E)$ definimos la siguiente red $N = (V', E')$:

- $V' = V_1 \cup V_2 \cup \{s, t\}$, con s y t dos vértices ficticios representando la fuente y el sumidero de la red.
- $E' = \{(i,j) : i \in V_1, j \in V_2, ij \in E\}$
  $\cup \{(s,i) : i \in V_1\}$
  $\cup \{(j,t) : j \in V_2\}$.
- $u_{ij} = 1$ para todo $ij \in E$.

El cardinal del matching máximo de G será igual al valor del flujo máximo en la red N.



By Damy