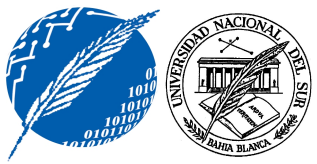
Algoritmos y Complejidad

Algoritmos sobre Grafos

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

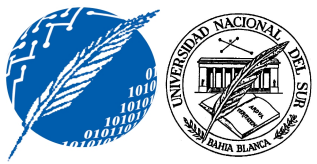
primer semestre 2024



Algoritmos sobre Grafos

Estructura de datos Grafos

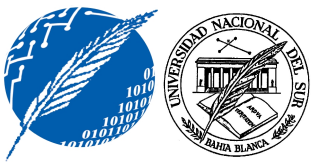
- los grafos constituyen una de las más importantes Estructuras de Datos en las Ciencias de Computación. Un inmensa variedad de problemas basan su solución en el uso de grafos
- en esta materia sólo nos interesaremos en aquellos problemas dónde es posible una representación total del grafo en la memoria de la máquina
- las heurísticas para el manejo de grafos implícitos se ven en Inteligencia Artificial



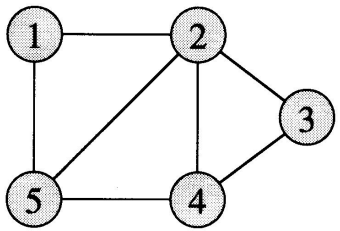
Representación de grafos

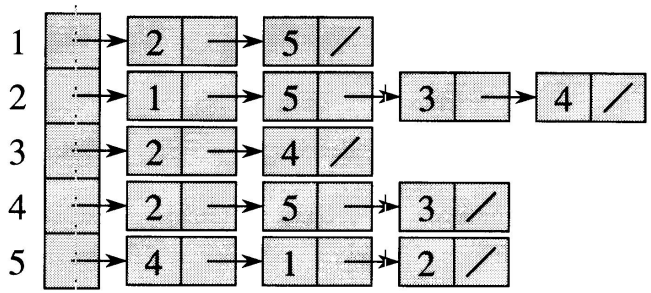
Representación de Grafos $\left\{ \begin{array}{l} \text{Matriz de Adyacencia} \\ \text{Lista de Adyacencia} \end{array} \right.$

- los arcos pueden o no tener peso y/o etiquetas
- los arcos pueden o no ser dirigidos
- se pueden permitir o no varios arcos entre los mismos nodos (multigrafos)
- se pueden permitir o no arcos hacia el mismo nodo



Representación grafos no dirigidos

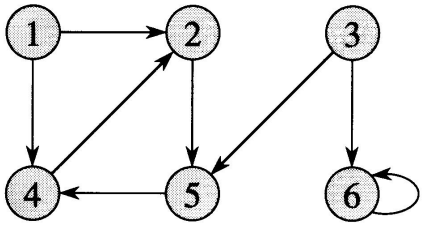


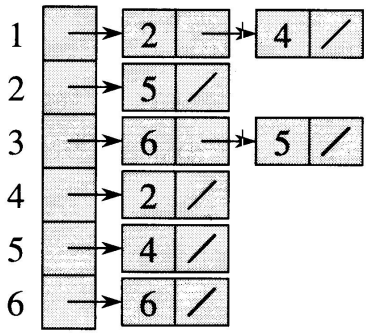


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Memoria
- matriz de adyacencia $\Theta(n^2)$
- lista de adyacencia $\Theta(n+2a) = \Theta(\max(n,a))$

Representación grafos dirigidos



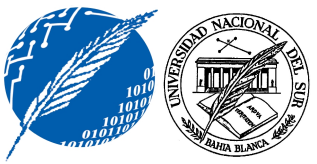


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Memoria
- matriz de adyacencia: $\Theta(n^2)$
- lista de adyacencia: $\Theta(n+a) = \Theta(\max(n,a))$

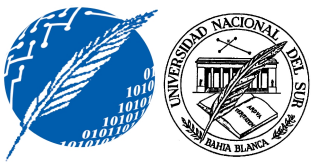
Recorrido de un grafo

- un recorrido es un algoritmo que visita todos los nodos de un grafo
- supondremos los grafos representados por matrices de adyacencia
- los algoritmos más usados para recorrer grafos generalizan los recorridos de árboles
- para el caso de grafo se necesita guardar información sobre los nodos que ya han sido visitados, de modo de no volver a visitarlos



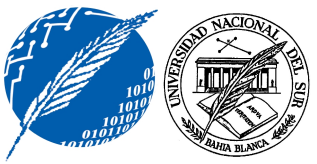
Cómo evitar volver a entrar a ciclos

- para recorrer grafos, se etiquetarán dinámicamente los nodos como:
  - nodos blancos: todavía no han sido visitados
  - nodos grises: ya han sido visitados, pero no se ha controlado la visita a todos sus adyacentes
  - nodos negros: ya han sido visitados, al igual que todos sus adyacentes
- esta caracterización implica que ningún nodo negro tiene un nodo blanco como adyacente



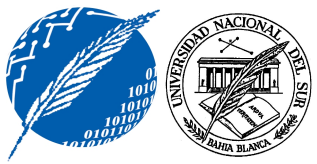
Recorrido por niveles

- el recorrido por niveles, o *breadth-first search* (BF), basa el orden de visita de los nodos del grafo en una E.D. Cola, incorporándole en cada paso los adyacentes al nodo actual
- esto implica que se visitarán todos los hijos de un nodo antes de proceder con sus demás descendientes
- las operaciones sobre la E.D. Cola se suponen implementadas en tiempo $\Theta(1)$



Recorrido por niveles

- el algoritmo que se presenta a continuación es no determinístico: la elección de los nodos en cada ciclo puede hacerse en forma arbitraria, o dependiente del contexto si es necesario
- puede aplicarse tanto a grafos dirigidos y como a no dirigidos
- está dividido en dos procedimientos
  - `bfs` inicializa las estructuras de datos, inicia el recorrido y controla que todos los nodos hayan sido visitados
  - `visitaBF` realiza el recorrido propio a partir de un nodo ya elegido



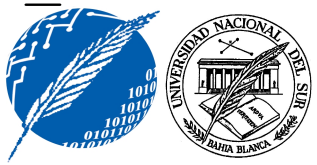
Algoritmo BFS

```
PROCEDURE bfs(G=N,A)
 FOR cada vértice v en N
   color[v]::=blanco
 ENDFOR
 Q.ColaVacía()
 FOR cada vértice v en N
   IF color[v]=blanco
     color[v]::=gris
     Q.insertar(v)
     visitarBF(G,Q)
   ENDIF
 ENDFOR
```

costo	veces
c	n
c	1
c	n
c	$\leq n$
c	$\leq n$
$T_{vBF}(n)$	$\leq n$


Algoritmo BFS

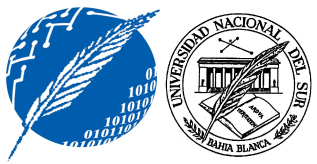
	costo	veces
`PROCEDURE visitarBF(G,Q)`		
`WHILE no Q.vacía()`		
`u::=Q.primero()`	b	$\leq n$
`IF existe (u,w) tq color[w]=blanco`	b	$\leq n$
`color[w]::=gris`		
`Q.insertar(w)`		
`ELSE`	b	$\leq a \times n$
`color[u]::=negro`		
`Q.sacarDeCola()`		
`ENDIF`	b	$\leq a \times n$
`ENDWHILE`		



Análisis del tiempo de ejecución

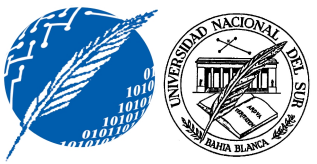
$$T_{BF}(n) \leq \sum_{i=1}^{n}(c + T_{visitarBF}(n)) = \sum_{i=1}^{n}(a + O(a \times n)) \in O(n^4)$$

- la cota obtenida es muy mala
- una análisis más detallado resulta en una cota mejor

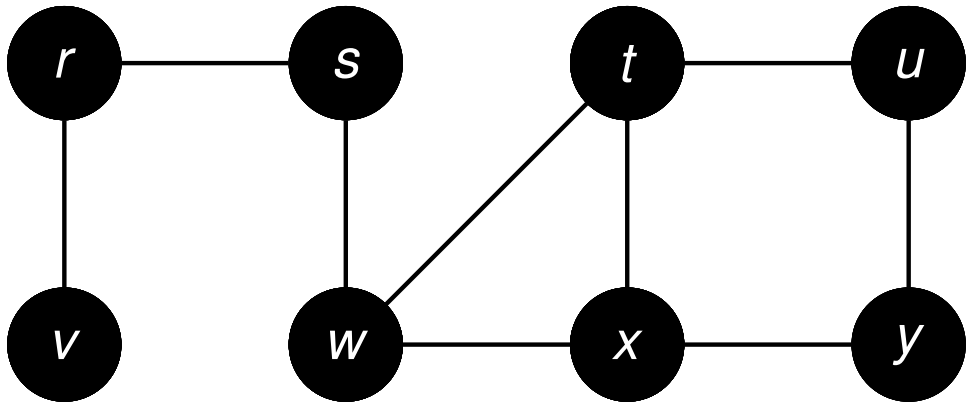


Análisis del tiempo de ejecución

- se puede probar por inducción sobre el número de iteraciones
  - que todo nodo es coloreado blanco, gris y negro exactamente una vez, y en ese orden
  - que todos los nodos en Q son grises y solamente estan una vez en Q
  - que los controles de adyacencia se hace exactamente una vez por cada arco.
- esto resulta $T_{BF}(n) = \Theta(\max(n, a))$, tomando como barómetros las operaciones sobre la E.D. Cola



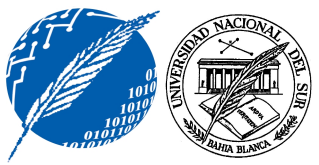
Ejemplo de ejecución de BFS



$Q = \langle\rangle$ $Q = \langle s \rangle$ $Q = \langle s, r \rangle$ $Q = \langle s, r, w \rangle$ $Q = \langle r, w \rangle$ $Q = \langle r, w, v \rangle$
$Q = \langle w, v \rangle$ $Q = \langle w, v, t \rangle$ $Q = \langle w, v, t, x \rangle$ $Q = \langle v, t, x \rangle$ $Q = \langle t, x \rangle$
$Q = \langle t, x, u \rangle$ $Q = \langle x, u \rangle$ $Q = \langle x, u, y \rangle$ $Q = \langle u, y \rangle$ $Q = \langle y \rangle$ $Q = \langle\rangle$

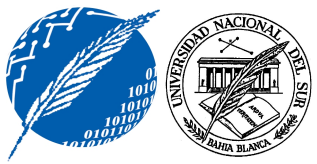
Propiedades de BFS

- al mismo tiempo que se hace un recorrido por niveles de un grafo se pueden calcular algunos datos adicionales que serán útiles para las aplicaciones de este algoritmo
- la foresta del recorrido es el subgrafo de G formado por todos los arcos utilizados en el recorrido
- se puede probar que siempre es una foresta (conjunto de árboles)
- se representa a través de un arreglo adicional $\texttt{padre}[1..n]$ donde cada nodo guarda su antecesor en la foresta; si $\texttt{padre}[i] = 0$ entonces el nodo es una raíz
- este arreglo se inicializa en 0



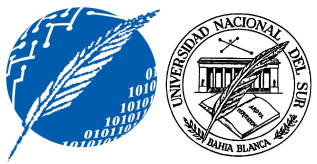
Propiedades de BFS

- el nivel de cada nodo es la distancia (en cantidad de arcos) que deben recorrerse en la foresta para llegar desde la raíz al nodo correspondiente
- se representa por un arreglo adicional `nivel`$[1..n]$, inicializado en 0
- se debe recordar que la raíz es en principio un nodo arbitrario
- se pueden calcular esta información adicional al mismo tiempo que se hace el recorrido; esto no agrega tiempo adicional al orden del algoritmo

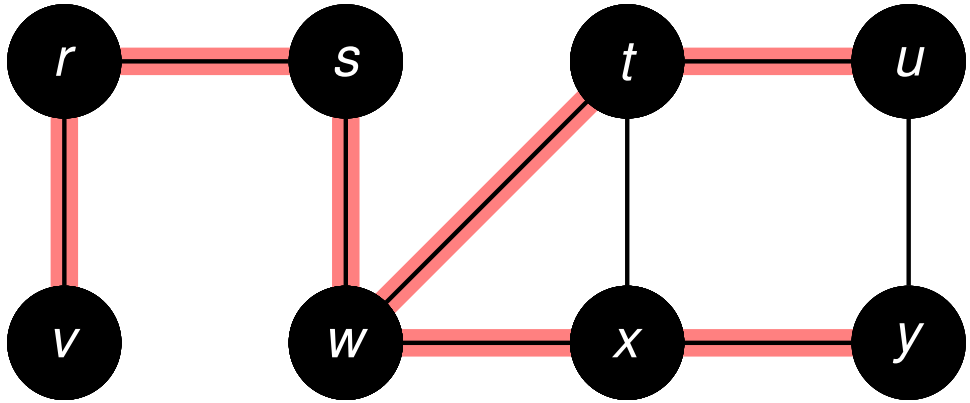


Algoritmo BFS - cálculo propiedades

```
PROCEDURE visitarBF(G,Q)
 WHILE no Q.esVacía()
  u::=Q.primero()
  IF existe (u,w) tq color[w]=blanco
       color[w]::=gris
       padre[w]::=u; nivel[w]::=nivel[u]+1
       Q.insertar(w)
  ELSE
       color[u]::=negro
       Q.sacarDeCola()
  ENDIF
 ENDWHILE
```



Ejemplo de ejecución de BFS - cálculo propiedades



$Q = \langle\rangle$ $Q = \langle s\rangle$ $Q = \langle s, r\rangle$ $Q = \langle s, r, w\rangle$ $Q = \langle r, w\rangle$ $Q = \langle r, w, v\rangle$
$Q = \langle w, v\rangle$ $Q = \langle w, v, t\rangle$ $Q = \langle w, v, t, x\rangle$ $Q = \langle v, t, x\rangle$ $Q = \langle t, x\rangle$
$Q = \langle t, x, u\rangle$ $Q = \langle x, u\rangle$ $Q = \langle x, u, y\rangle$ $Q = \langle u, y\rangle$ $Q = \langle y\rangle$ $Q = \langle\rangle$

	r	s	t	u	v	w	x	y
`nivel[]`	0 1	0	0 2	03	0 2	0 1	02	03
`padre[]`	0 s	0	0 w	0t	0 r	0s	0 w	0x

Propiedades de BFS

- se pueden probar las siguientes propiedades sobre los recorridos por nivel

Lema 1

Un grafo no dirigido es conexo si y solo si la foresta de recorrido es un árbol

- en el caso de un grafo dirigido, conexo significa que existe un camino de ida y de vuelta entre cada par de nodos, y entonces vale solamente el “solo si”

Lema 2

Un grafo dirigido es conexo solo si la foresta de recorrido es un árbol

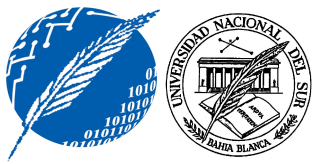
Propiedades de BFS

- la siguiente propiedad caraceriza los valores de `nivel`

Lema 3

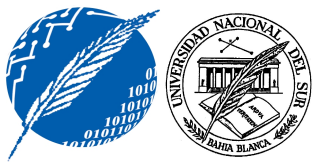
Al finalizar un recorrido BF, `nivel`$[v]$ *contiene la mínima distancia (en cantidad de arcos) desde la raíz del árbol de* v *en la foresta de recorrido hasta* v

- ejercicio: comparar con el algoritmo de Dijkstra

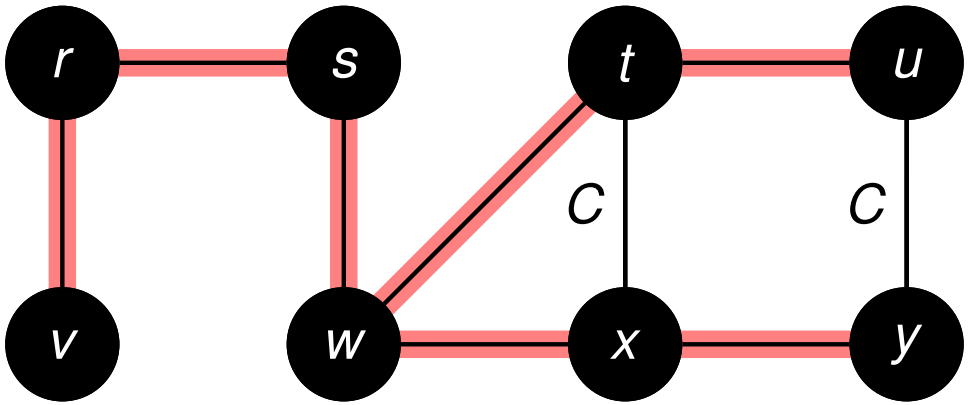


Clasificación de arcos

- el recorrido permite clasificar los arcos del grafo en las siguientes categorías:
  - arcos de la foresta son los arcos $(\texttt{padre}[v], v)$ utilizados en el recorrido
  - arcos hacia atrás (B) son arcos (u, v) en donde v es un ancestro de u en la foresta del recorrido
  - arcos hacia adelante (F) son arcos (u, v) en donde v es descendiente de u en la foresta del recorrido
  - arcos cruzados (C) son los demás arcos que no entran en las otras categorías. Los extremos pueden estar en el mismo árbol o en árboles diferentes



Ejemplo de clasificación de arcos en BFS



Clasificación de arcos

- si el grafo es no dirigido, como cada arco es considerado dos veces esta clasificación puede ser ambigua. Se toma la primera de las categorías posibles según el orden dado
- estas categorías pueden calcularse en el mismo algoritmo del recorrido, sin aumentar el orden del tiempo de ejecución
- ejercicio: adaptar el algoritmo de recorrido para clasificar todos los arcos

Propiedades de BFS

Lema 4

Si el grafo es no dirigido, entonces un recorrido BF sólo genera arcos de la foresta y cruzados

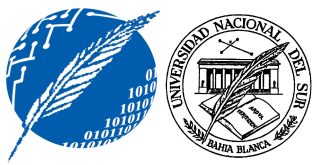
Lema 5

Si el grafo es dirigido, entonces un recorrido BF no genera arcos hacia adelante

- las demostraciones se hacen por el absurdo (ejercicio)

Recorrido por profundidad

- el recorrido por profundidad, o *depth-first search* (DF), basa el orden de visita de los nodos del grafo en una E.D. Pila, agregando en cada paso los adyacentes al nodo actual
- esto hace que agote los nodos accesibles desde un hijo antes de proceder con sus hermanos
- las operaciones sobre la E.D. Pila se suponen de tiempo en $\Theta(1)$
- al igual que el recorrido por profundidad, se presenta un algoritmo no determinístico que puede modificarse para incluir un orden en la elección de los nodos
- puede aplicarse tanto a grafos dirigidos como a grafos no dirigidos

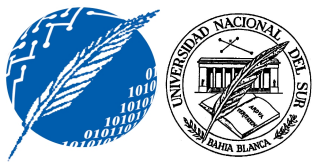


Algoritmo DFS

```
PROCEDURE dfs(G=N,A)
 FOR cada vértice v en N
   color[v]::=blanco
 ENDFOR
 P.pilaVacía()
 FOR cada vértice v en N
   IF color[v]=blanco
     color[v]::=gris
     P.apilar(v)
     visitarDF(G,P)
   ENDIF
 ENDFOR
```

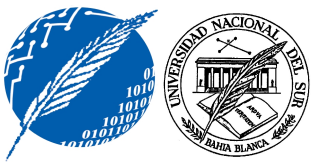
Algoritmo DFS

```
PROCEDURE visitarDF(G,P)
WHILE no P.esVacía()
 u::=P.tope()
 IF existe (u,w) tq color[w]=blanco
        color[w]::=gris
        P.apilar(w)
 ELSE
        color[u]::=negro
        P.desapilar()
 ENDIF
ENDWHILE
```

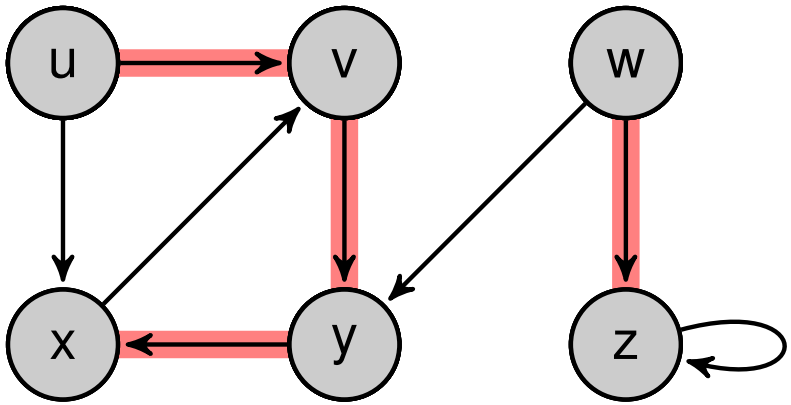


Propiedades

- en forma análoga al recorrido por niveles, se prueba por inducción que
  - todo nodo es coloreado blanco, gris y negro exactamente una vez, y en ese orden
  - todos los nodos en P son grises y solamente estan una vez en P
  - los controles de adyacencia se hace exactamente una vez por cada arco
- resultando en un algoritmo $\Theta(n+a)$



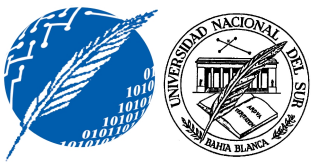
Ejemplo de ejecución de DFS



$P = \langle\rangle$ $P = \langle u \rangle$ $P = \langle v, u \rangle$ $P = \langle y, v, u \rangle$ $P = \langle x, y, v, u \rangle$ $P = \langle y, v, u \rangle$
$P = \langle v, u \rangle$ $P = \langle u \rangle$ $P = \langle\rangle$ $P = \langle w \rangle$ $P = \langle z, w \rangle$ $P = \langle w \rangle$ $P = \langle\rangle$

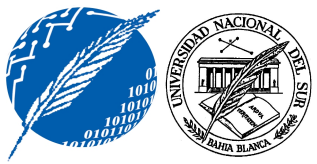
Propiedades

- al igual que en los recorridos por niveles, es posible obtener la foresta del recorrido de un recorrido por profundidad
- en cambio, la numeración `nivel`$[u]$ no tiene sentido
- sí es posible en este tipo de recorrido numerar a los nodos de acuerdo al tiempo del evento de descubrimiento (numeración preorden), o de finalización (numeración postorden)
- el descubrimiento coincide con la inserción del nodo en la pila; la finalización con su eliminación



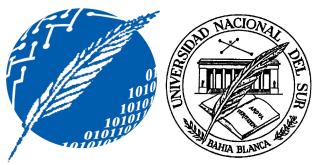
Propiedades

- es útil que la numeración sea de acuerdo al orden de los eventos, usando una estampilla de tiempo, que se guarda en `tiempo`
- la numeración en preorden se hace simultáneamente con la coloración en gris; y se guarda en un arreglo `d`$[1..n]$
- la numeración en postorden coincide con la colaración en negro; y se guarda en un arreglo `f`$[1..n]$
- en ambos casos siempre se actualiza la estampilla de tiempo

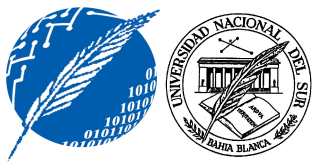


Algoritmo DFS - cálculo de propiedades

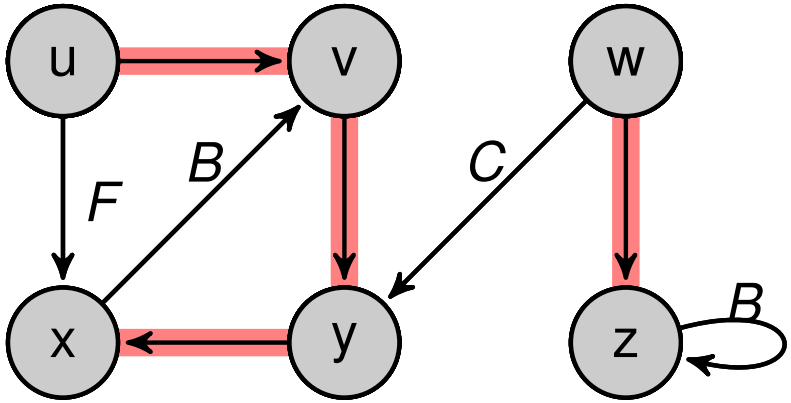
```
PROCEDURE dfs(G=N,A)
 FOR cada vértice v en N
   color[v]::=blanco
 ENDFOR
 P.pilaVacía(); tiempo::=0
 FOR cada vértice v en N
   IF color[v]=blanco
     color[v]::=gris; tiempo++; d[v]::=tiempo
     P.apilar(v)
     visitarDF(G,P)
   ENDIF
 ENDFOR
```



```
PROCEDURE visitarDF(G,P)
 WHILE no P.esVacía()
  u::=P.tope()
  IF existe (u,w) tq color[w]=blanco
        color[w]::=gris
        tiempo++; d[w]::=tiempo
        P.apilar(w)
  ELSE
        color[u]::=negro
        tiempo++; f[u]::=tiempo
        P.desapilar()
  ENDIF
 ENDWHILE
```

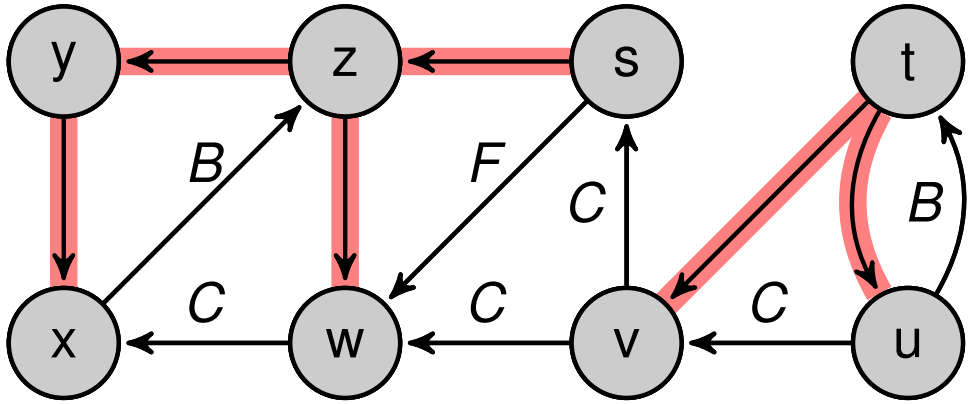


Ejemplo de ejecución de DFS



$\texttt{tiempo} = 0 \;\; P = \langle\rangle \; \texttt{tiempo} = 1 \;\; P = \langle u \rangle \; \texttt{tiempo} = 2 \;\; P = \langle v, u \rangle$
$\texttt{tiempo} = 3 \;\; P = \langle y, v, u \rangle \; \texttt{tiempo} = 4 \;\; P = \langle x, y, v, u \rangle$
$\texttt{tiempo} = 4 \;\; P = \langle x, y, v, u \rangle \; \texttt{tiempo} = 5 \;\; P = \langle y, v, u \rangle$
$\texttt{tiempo} = 6 \;\; P = \langle v, u \rangle \; \texttt{tiempo} = 7 \;\; P = \langle u \rangle$
$\texttt{tiempo} = 7 \;\; P = \langle u \rangle \; \texttt{tiempo} = 8 \;\; P = \langle\rangle \; \texttt{tiempo} = 9 \;\; P = \langle w \rangle$
$\texttt{tiempo} = 9 \;\; P = \langle w \rangle \; \texttt{tiempo} = 10 \;\; P = \langle z, w \rangle$
$\texttt{tiempo} = 10 \;\; P = \langle z, w \rangle \; \texttt{tiempo} = 11 \;\; P = \langle w \rangle$
$\texttt{tiempo} = 12 \;\; P = \langle\rangle$

Ejemplo de numeración y clasificación de arcos



$P = \langle\rangle$ $P = \langle s\rangle$ $P = \langle z, s\rangle$ $P = \langle y, z, s\rangle$ $P = \langle x, y, z, s\rangle$ $P = \langle x, y, z, s\rangle$
$P = \langle y, z, s\rangle$ $P = \langle z, s\rangle$ $P = \langle w, z, s\rangle$ $P = \langle w, z, s\rangle$ $P = \langle z, s\rangle$ $P = \langle s\rangle$
$P = \langle s\rangle$ $P = \langle\rangle$ $P = \langle t\rangle$ $P = \langle v, t\rangle$ $P = \langle v, t\rangle$ $P = \langle v, t\rangle$ $P = \langle t\rangle$
$P = \langle u, t\rangle$ $P = \langle u, t\rangle$ $P = \langle u, t\rangle$ $P = \langle t\rangle$ $P = \langle\rangle$

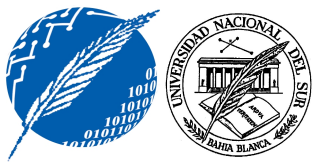
	s	t	u	v	w	x	y	z
d[]	0 1	0 11	0 14	0 12	07	04	0 3	0 2
f[]	010	016	0 15	0 13	08	05	06	09

Propiedades

Lema 6

Sea $G = \langle N, A \rangle$ *un grafo, y* $\mathtt{d}[], \mathtt{f}[]$ *la numeración de descubrimiento y finalización obtenida mediante un recorrido DF. Entonces*

- (u, v) *es de la foresta o F si y solo si* $\mathtt{d}[u] < \mathtt{d}[v] < \mathtt{f}[v] < \mathtt{f}[u]$
- (u, v) *es B si y solo si* $\mathtt{d}[v] < \mathtt{d}[u] < \mathtt{f}[u] < \mathtt{f}[v]$
- (u, v) *es C si y solo si* $\mathtt{d}[v] < \mathtt{f}[v] < \mathtt{d}[u] < \mathtt{f}[u]$



Propiedades

- muchas propiedades surgen a partir de la información obtenida en un recorrido DF

Teorema 7

En un recorrido DF de un grafo no dirigido G, todos sus arcos son de la foresta o hacia atrás

Demostración.

Sea (u, v) un arco del grafo, y supongamos que u es descubierto antes que v. Luego v es descubierto y finalizado antes de finalizar u. Si el nodo v tiene como padre a u entonces (u, v) es un arco de la foresta; si el nodo v tiene otro padre, entonces (u, v) es un arco hacia atrás. En forma similar se prueba si v es descubierto antes que u. □

Lema 8

En un recorrido DF de un grafo G, para cualquier par de nodos distintos u, v vale exactamente uno de:

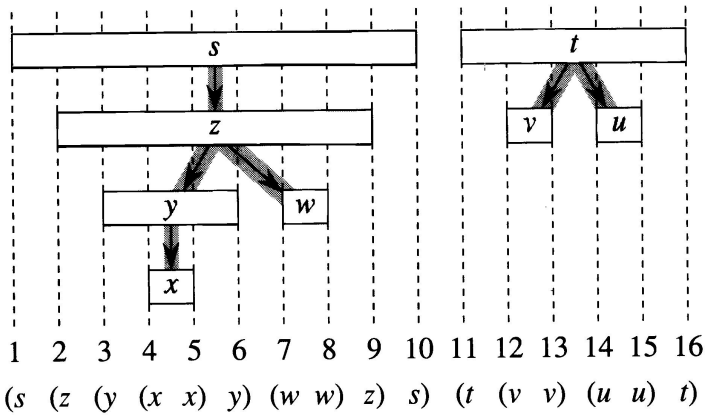
- *los intervalos* $[\mathtt{d}[u], \mathtt{f}[u]]$ *y* $[\mathtt{d}[v], \mathtt{f}[v]]$ *son totalmente disjuntos*
- $[\mathtt{d}[u], \mathtt{f}[u]] \subset [\mathtt{d}[v], \mathtt{f}[v]]$ *y u es descendiente de v*
- $[\mathtt{d}[v], \mathtt{f}[v]] \subset [\mathtt{d}[u], \mathtt{f}[u]]$ *y v es descendiente de u*

Demostración.

Analizando caso por caso si $\mathtt{d}[u] < \mathtt{d}[v] < \mathtt{f}[u]$, $\mathtt{d}[u] < \mathtt{f}[u] < \mathtt{d}[v]$, $\mathtt{d}[v] < \mathtt{d}[u] < \mathtt{f}[v]$ y $\mathtt{d}[v] < \mathtt{f}[v] < \mathtt{d}[u]$ □

Ejemplo de numeración y clasificación de arcos

	s	t	u	v	w	x	y	z
d[]	1	11	14	12	7	4	3	2
f[]	10	16	15	13	8	5	6	9



Corolario 9 (anidamiento de descendientes)

v es un descendiente propio de u en la foresta de recorrido si y solo si $\mathtt{d}[u] < \mathtt{d}[v] < \mathtt{f}[v] < \mathtt{f}[u]$

Demostración.

Inmediato del lema 8. □

Teorema 10 (caminos blancos)

En un recorrido DF de un grafo G, un nodo v es descendiente de un nodo u si y solo si en el momento $\mathtt{d}[u]$ *cuando u es descubierto existe un camino de u a v que consiste totalmente de nodos blancos.*

Demostración.

Para el *solo si* basta con aplicar el teorema 9 a todos los nodos intermedios. Para el *si* supongamos por el absurdo que *v* es alcanzable desde *u* por un camino de nodos blancos en el momento $\mathtt{d}[u]$, pero *v* no es descendiente de *u*. Se toma el primer nodo en el camino de *u* a *v* que no es descendiente de *u*, y se demuestra que su intervalo está contenido en $[\mathtt{d}[u], \mathtt{f}[u]]$, contradiciendo el teorema 9. □

Caracterización de grafos acíclicos

Lema 11

Un grafo dirigido G es acíclico si y solo si cualquier recorrido DF de G no produce arcos hacia atrás.

Demostración.

El *solo si* es inmediato. Para el *si* basta aplicar el teorema 10 al primer nodo del ciclo. □

Resumen de recorridos

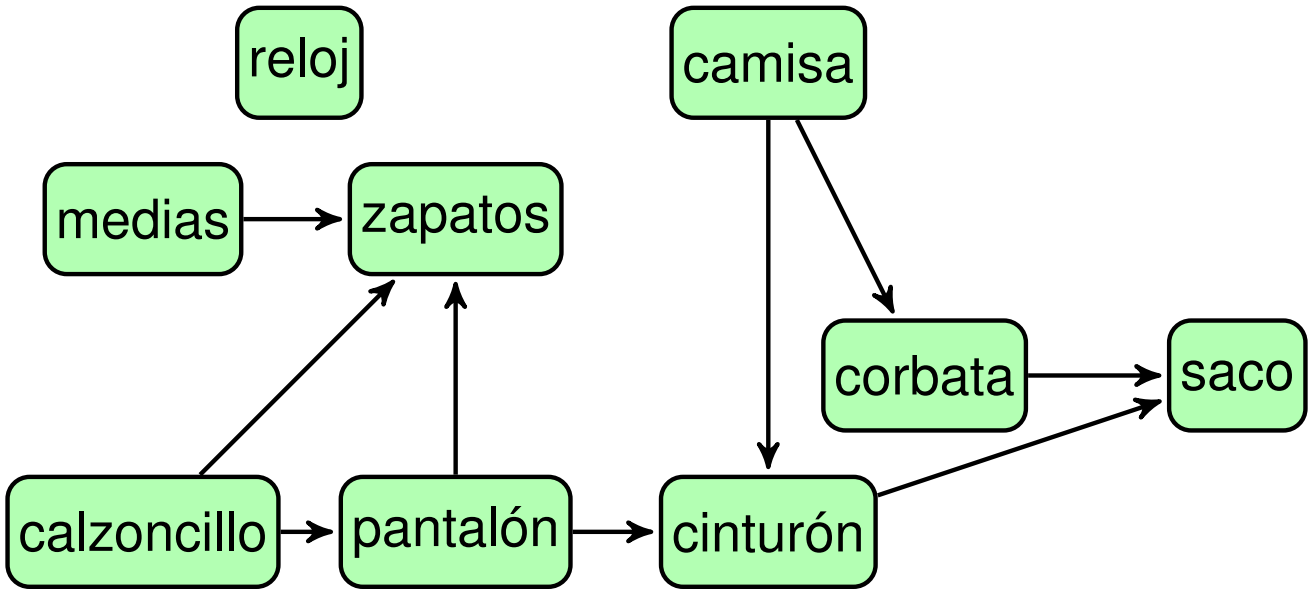
	calcula	arcos en g. no dir.	arcos en g. dir.	propiedades
BFS	`padre[]` `nivel[]`	de la foresta C	de la foresta B C	conectitud (no dir.) distancia mínima

DFS	`padre[]`
	`d[]`
	`f[]`
		de la foresta
		B
			de la foresta
			B
			F
			C
				conectitud (no dir.)
				caminos blancos
				caracteriz. dags

Definición del problema

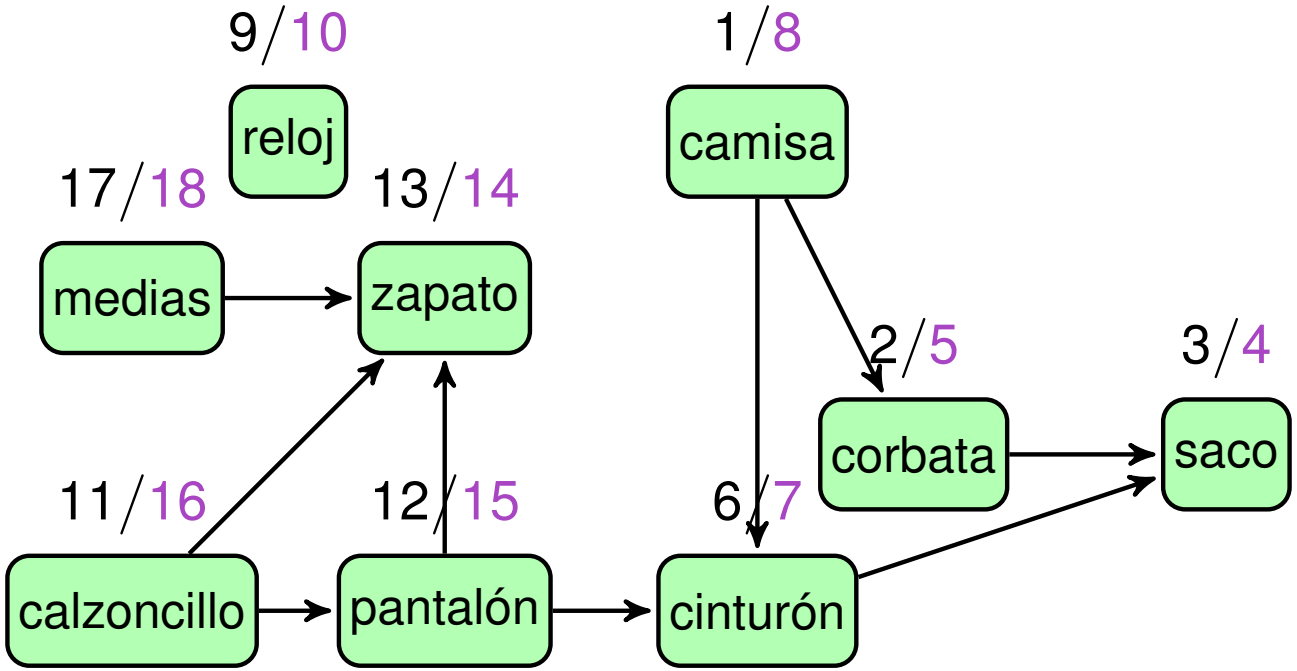
- Problema: dado un grafo dirigido acíclico G, un orden topológico es un ordenamiento lineal de sus nodos de forma que si el arco $(u, v) \in G$ entonces u aparece antes de v en el ordenamiento
- si el grafo tiene ciclos, entonces tal ordenamiento no existe
- el orden topológico es usado para planificar una serie de acciones que tienen precedencias: cada nodo representa una acción, y cada arco (u, v) significa que la acción u debe ejecutarse necesariamente antes de v

Ejemplo de DAG para actividades en un proyecto



Algoritmo

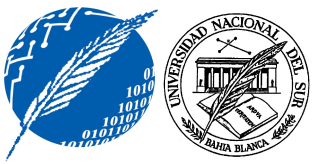
- la numeración de finalización (posorden) de los nodos en un recorrido DF da una idea del algoritmo para resolver el problema



Algoritmo

```
PROCEDURE OrdenTopológico(G)
  array f[1..n]
  DFS(G,f) % calculando f[v]
  L::= lista de nodos ordenada
       por f en forma decreciente
  RETURN L
```

- el tiempo de ejecución de este algoritmo claramente es $\Theta(n \log n)$, pero es posible reducirlo a $\Theta(n+a)$ cambiando levemente el algoritmo del recorrido (ejercicio)



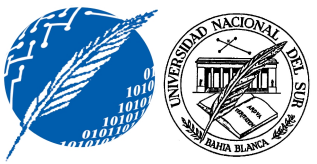
Correctitud

Teorema 12

El resultado del algoritmo anterior es un orden topológico.

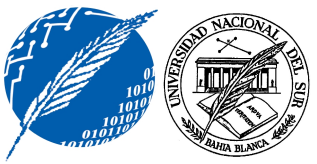
Demostración.

Se muestra que para todo arco (u, v) en G dirigido acíclico, $f[v] < f[u]$. Sólo hay tres tipos de arcos posible (por el lema 11): de la foresta, hacia adelante o cruzado. Usando el lema 6 siempre $f[v] < f[u]$. □

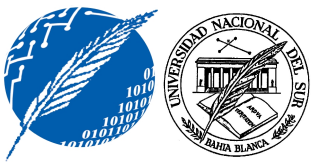


Definición del problema

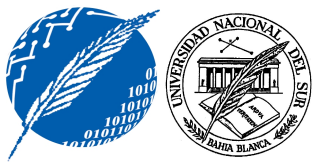
- dado un grafo dirigido $G = \langle N, A \rangle$ un componente fuertemente conexo (CFC) es un conjunto $U \subseteq N$ maximal tal que para todo $u, v \in U$ valen $u \rightsquigarrow_G v$ y $v \rightsquigarrow_G u$ (donde $a \rightsquigarrow_G b$ significa que existe en G un camino de a a b)
- Problema: encontrar todos los CFC de un grafo dirigido G
- para el caso de grafos no dirigidos, el problema se denomina componentes conexos y puede resolverse directamente a partir de cualquiera de los recorridos vistos



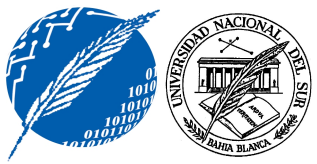
- sea G un grafo, su grafo traspuesto G^T se define como $G^T = \langle N, \{(u,v) : (v,u) \in A\}\rangle$
- es interesante observar que G y G^T tienen los mismos CFC (ejercicio)
- el algoritmo para CFC hace dos recorridos: uno del grafo G y otro del grafo G^T
- en el segundo recorrido, los nodos se consideran en orden decreciente de `f[]`



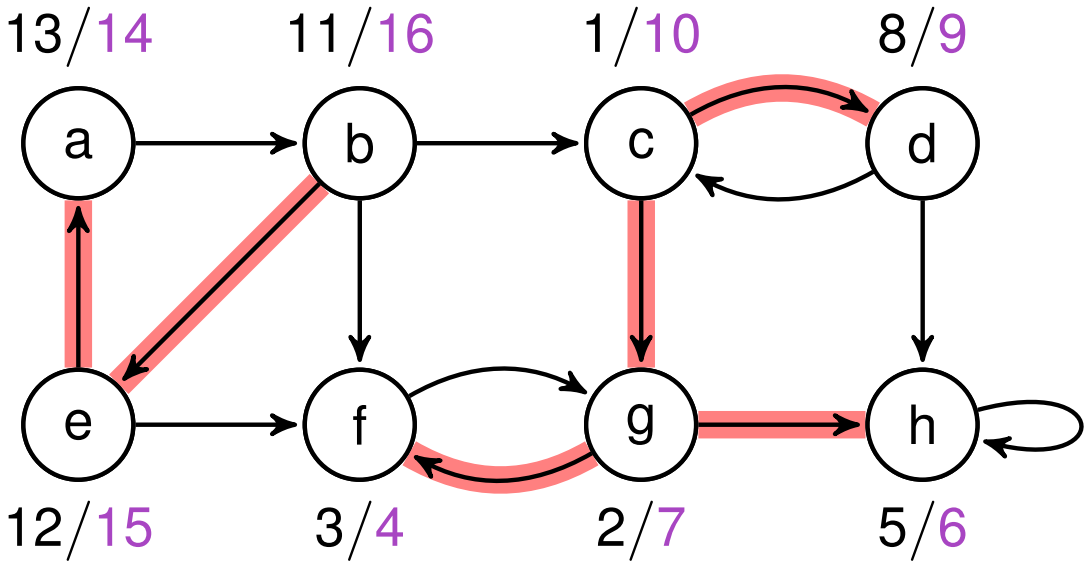
```
PROCEDURE CFC(G)
 array f[1..n]
 DFS(G,f)
 calcular GT % el traspuesto de G
 array padre[1..n]
 DFS(GT, padre, f)
      % padre es la foresta del recorrido,
      % tomando los nodos por f decreciente
 RETURN padre
```



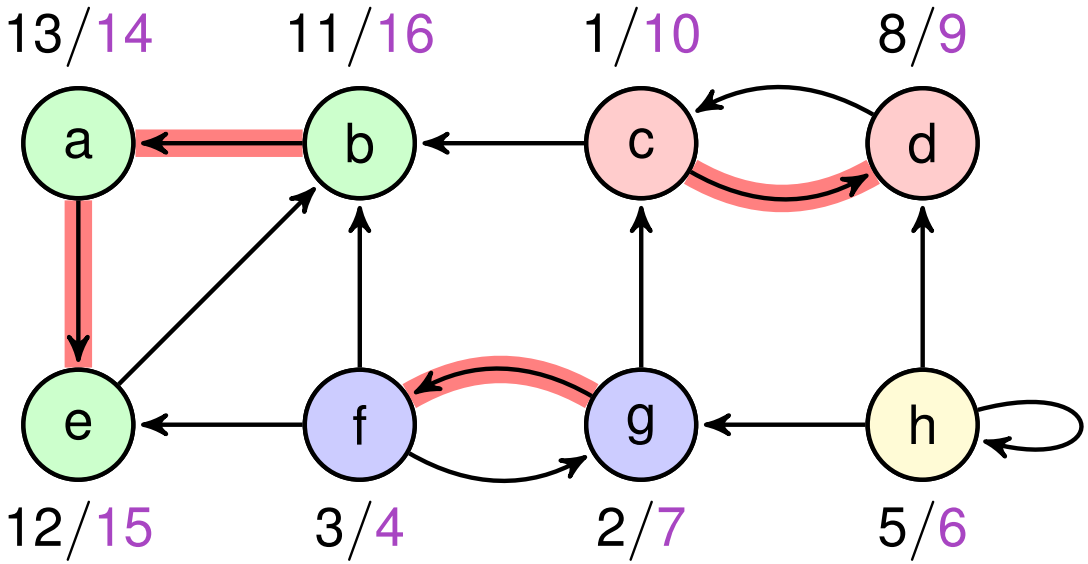
- cada árbol de la foresta representado en `padre` contiene exactamente los elementos de un CFC
- suponiendo una representación del grafo mediante lista de adyacencia, el tiempo y espacio del algoritmo anterior es de $\Theta(n+a)$ (ejercicio)



Ejemplo de ejecución del algoritmo (primer DFS)

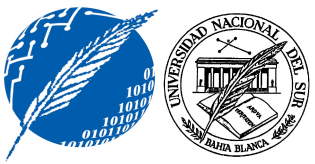


Ejemplo de ejecución del algoritmo (segundo DFS)



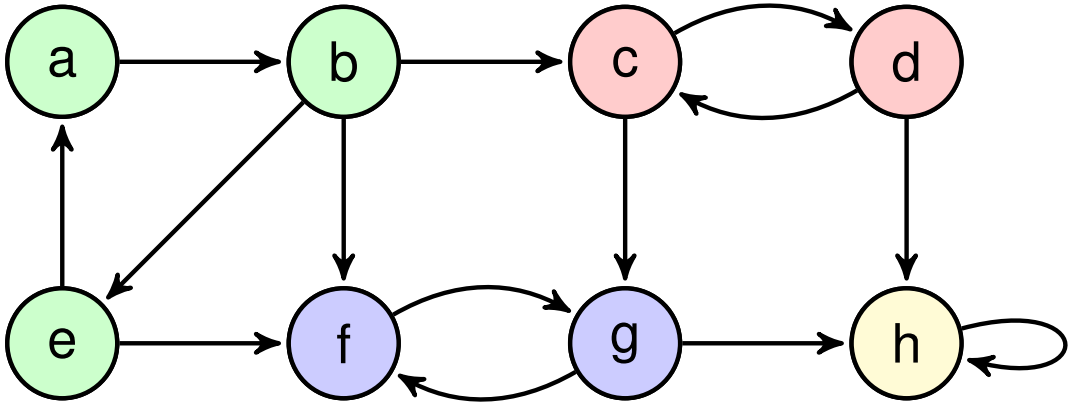
Correctitud

- para demostrar la correctitud del algoritmo, se usará el concepto de grafo de componentes
- sea $G = \langle N, A \rangle$ dirigido, entonces su grafo de componentes es $G^{CFC} = \langle N^{CFC}, A^{CFC} \rangle$, donde N^{CFC} está formado por un nodo que representa a cada CFC de G, y para cada par de CFC u, v se agrega un arco (u, v) en A^{CFC} si en G existe en el componente u un nodo que presenta un arco hacia otro nodo en el componente v

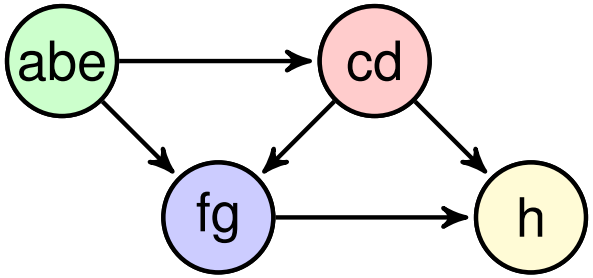


Ejemplo de grafo de componentes

G



G^{CFC}



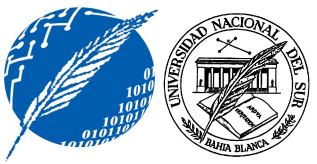
Propiedad del grafo de componentes

Lema 13

Sea G un grafo dirigido, entonces G^{CFC} es un dag.

Demostración.

Si hacemos un recorrido DF de G^{CFC}, no pueden existir arcos hacia atrás (de lo contrario los nodos extremos del arco hacia atrás estarían en el mismo componente). Por el lema 11, G^{CFC} es un dag. □

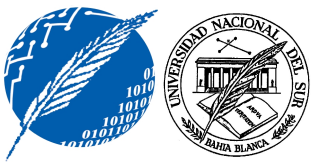


Propiedad del grafo de componentes

- se extiende la numeración de descubrimiento y finalización para conjuntos de nodos de la forma:

$$\mathtt{d}[C] = min_{u \in C}(\mathtt{d}[u])$$

$$\mathtt{f}[C] = max_{u \in C}(\mathtt{f}[u])$$



Correctitud del algoritmo para CFC

Lema 14

Sean $C, C' \in N^{CFC}$ *de* $G = \langle N, A \rangle$ *tal que* $(C, C') \in A^{CFC}$. *Entonces en un recorrido DF de G, vale* $\mathtt{f}[C] > \mathtt{f}[C']$.

Demostración.

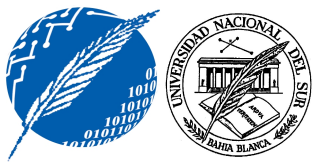
Se distinguen dos casos de acuerdo a cual componente C o C' es encontrado primero en el primer recorrido. Si primero se encuentra un nodo x de C entonces todos los nodos de C' son descendientes de x por el teorema 10. Si primero se encuentra un nodo y de C', dado que existe (C, C') entonces no existe (C', C) en G^{CFC} por el lema 13. Esto significa que en el momento $\mathtt{f}[C']$ todos los nodos en C son blancos e inalcanzables desde C', por lo que $\mathtt{f}[C] > \mathtt{f}[C']$. □

Correctitud del algoritmo para CFC

Corolario 15

Sea $C, C' \in N^{CFC}$ *dos CFC distintos de un grafo* $G = \langle N, A \rangle$*, entonces si existe* $(u, v) \in A^T$ *tal que* $u \in C$ *y* $v \in C'$ *vale que* $\mathtt{f}[C] < \mathtt{f}[C']$*.*

- esto significa que si comenzamos el recorrido de G^T a partir del nodo de mayor numeración $\mathtt{f}$, entonces no vamos a poder conectar con otro componente que no sea el de ese nodo
- así sucesivamente para los nodos restantes, cada árbol del segundo recorrido está compuesto por un único CFC



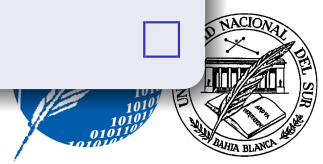
Correctitud del algoritmo para CFC

Teorema 16

El algoritmo calcula correctamente los CFC de cualquier grafo G.

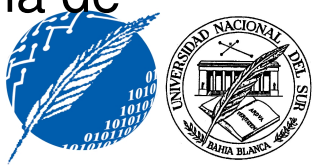
Demostración.

Por inducción sobre la cantidad de árboles en la foresta del segundo recorrido. Para el caso inductivo se muestra que si u es la raíz del k-ésimo árbol del segundo recorrido, entonces todos los nodos del CFC de u pertenecen a ese árbol. Además, por el corolario 15 cualquier arco que toca el CFC de u debe ser tal que conecta un descendiente de u con un nodo en un árbol anterior en la foresta. Luego el árbol con raíz u contiene exactamente el CFC de u. □



Caminos más cortos: problemas

- sea $G = \langle N, A \rangle$ un grafo dirigido, con pesos no negativos, en donde existe un nodo distinguido llamado *origen*
- se define el problema CAMINOS MÁS CORTOS CON ORIGEN ÚNICO al problema de hallar los caminos más cortos desde el origen hasta cada uno de los restantes nodos
- existen variantes de este problema, como CAMINOS MÁS CORTOS CON DESTINO ÚNICO, CAMINO MÁS CORTO PARA PAR ÚNICO y CAMINOS MÁS CORTOS PARA TODO PAR DE NODOS
- todos pueden ser resueltos mediante reducción al problema de CAMINOS MÁS CORTOS CON ORIGEN ÚNICO



Caminos más cortos con origen único

- consideraremos el grafo representado con una matriz de adyacencia G, con nodos $1..n$, y al nodo origen etiquetado con 1. $G[i,j]$ contiene el peso del arco (i,j) si $(i,j) \in A$, o $G[i,j] = \infty$ si el arco no existe
- atacaremos primero el problema de encontrar las distancias mínimas, y después el de los caminos que la implementan
- datos de entrada: `G[1..n, 1..n]` el grafo; se supone que el origen es el nodo 1
- datos de salida: `d[1..n]` un arreglo donde `d[i]` es la distancia más corta posible en G entre `1` y `i`
- veremos primero algunas propiedades, y luego el algoritmo greedy llamado de *Dijkstra*

Conceptos previos

- un camino en $G = \langle N, A \rangle$ es una secuencia $\langle n_1, n_2, \ldots, n_k \rangle$ de nodos $n_i \in N, 1 \leq i \leq k$ tales que $(n_i, n_{i+1}) \in A, 1 \leq i < k$. Se nota $n_1 \rightsquigarrow_G^p n_k$
- la distancia dist(p) de un camino $p = \langle n_1, n_2, \ldots, n_k \rangle$ es la suma de los pesos de los arcos intervinientes

$$\text{dist}(p) = \sum_{i=1}^{k} G[n_i, n_{i+1}]$$

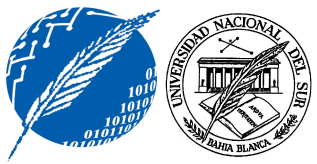
- dados dos nodos $u, v \in N$, la distancia mínima entre ellos $\delta(u, v)$ se define como

$$\delta(u,v) = \begin{cases} \min\{\text{dist}(p) : u \rightsquigarrow_G^p v\} & \text{si existe al menos un } p \\ \infty & \text{si no existe ningún } p \end{cases}$$

Relajación

- el arreglo `d` contendrá la menor distancia a cada nodo conocida en un momento dado de la ejecución; al finalizar el algortimo esa será la distancia mínima
- la relajación es una técnica que permite eventualmente reducir la distancia conocida hasta un momento dado, mediante la consideración de un posible nodo intermedio

```
Procedure RELAXATION(G,d[1..n],(u,v))
   d[v]::=min(d[v],d[u]+G[u,v])
RETURN
```

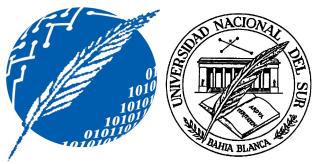


Propiedades de distancia mínima y relajación

- presentaremos algunas propiedades necesarias para las demostraciones de correctitud. Se asume que el grafo $G = \langle A, N \rangle$ y el arreglo d han sido inicializados, y que d sólo se modifica mediante el procedimiento de relajación

Lema 17 (Inecuación triangular)

para todo $(u, v) \in A$ *vale* $\delta(1, v) \leq \delta(1, u) + G[u, v]$



Propiedades de distancia mínima y relajación

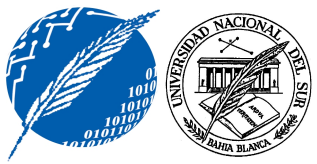
Lema 18 (Propiedad de la cota superior)

para todo $v \in N$ vale que $d[v] \geq \delta(1, v)$, y una vez que $d[v]$ alcanza el valor $\delta(1, v)$, nunca más cambia.

Lema 19 (Propiedad de convergencia)

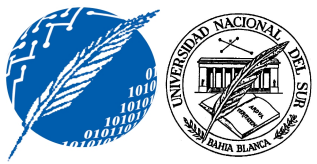
Para todo $(u, v) \in A$, si $1 \rightsquigarrow_G u \rightarrow v$ es un camino más corto y $d[u] = \delta(1, u)$ en algún momento anterior a relajar el arco (u, v), entonces $d[v] = \delta(1, v)$ en todo momento posterior

- las demostraciones quedan como ejercicios



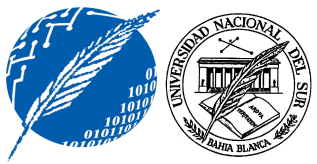
Algoritmo de Dijsktra

- el algoritmo de Dijsktra es un algoritmo *greedy* para resolver CAMINOS MÁS CORTOS CON ORIGEN ÚNICO
- consiste en mantener en S al conjunto de nodos cuyas distancias mínimas ya se conoce, y en C a aquellos que todavía falta calcular
- en cada paso se selecciona el elemento$u$ de C más cercano al origen, y se relajan los arcos que salen de u



Algoritmo de Dijsktra para distancias mínimas

```
array d[1..n]
FOR i ::= 1 TO n
   d[i] ::= G[1,i]
ENDFOR
S ::= {1}; C ::= {2, ..., n}
FOR j ::= 1 TO n-2
   u ::= el elemento de C que minimiza d[u]
   S ::= S + {u}; C ::= C-{u}
   FOR cada (u,v) en A
      RELAXATION(G,d,(u,v))
   ENDFOR
ENDFOR; RETURN d
```

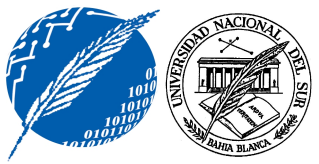


Análisis del tiempo de ejecución

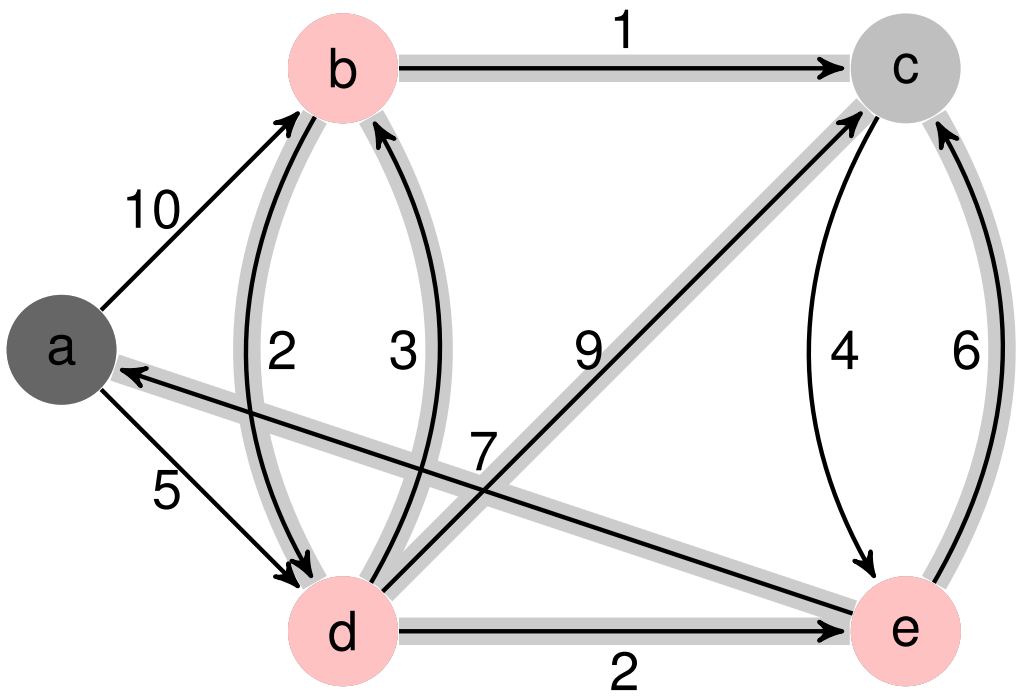
- se analizan las sentencias del ciclo greedy, y por separado la cantidad de llamadas a RELAXATION (como barómetro) que están acotadas en total por la cantidad de arcos

$$\begin{aligned} T_{\text{Dijsktra}}(n) &= an + b + \sum_{j=1}^{n-2} \sum_{k=1}^{n-1} c + \sum_{j=1}^{n-2} d + \\ &\quad + \sum_{j=1}^{a} e = \\ &\in \Theta(n^2) \end{aligned}$$

- sabiendo que siempre $a \leq n^2$



Ejemplo de ejecución de Dijsktra

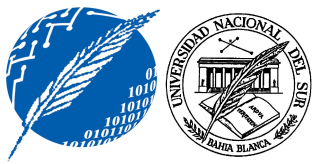


	a	b	c	d	e
d[]	0	10	8 ∞ 14	13	9 5 ∞ 7

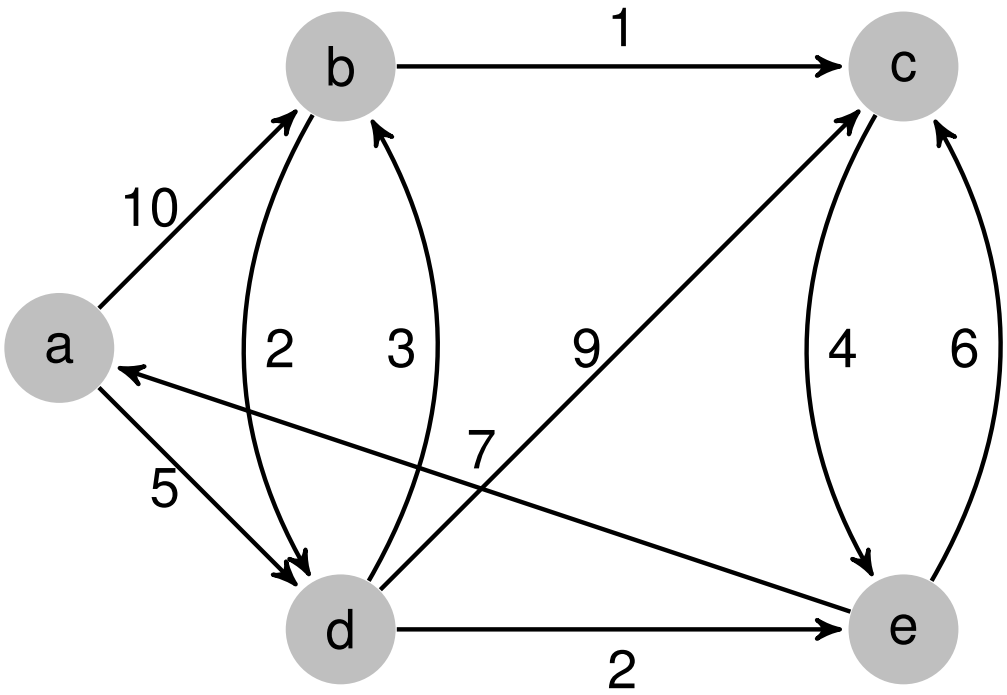
Algoritmo de Dijsktra para caminos más cortos

- para obtener los caminos más cortos es suficiente con mantener un arreglo auxiliar `p[1...n]`, inicializado en 1, donde `p[i]` contiene el último nodo en el camino más corto entre 1 y *i*
- luego se añade la incialización de `p` junto con la inicialización de `d`; y se agregra en la relajación su actualización

```
Procedure RELAXATION(G,d[1..n],(u,v))
 IF d[v]>d[u]+G[u,v]
    d[v] ::= d[u]+G[u,v]
    p[v] ::= u
 ENDIF
RETURN
```



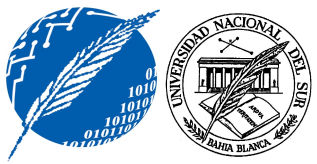
Algoritmo de Dijsktra para caminos más cortos



	a	b	c	d	e
d[]	0	8	9	5	7
p[]	0	d	b	a	d

Algoritmo de Dijsktra para caminos más cortos

- ¿cómo se obtienen los caminos más cortos a partir de p?
- ¿cambia el orden del tiempo de ejecución?



Alternativas de implementación

Ejercicios:

- se pueden evitar los excesivos accesos a `g[i,j]` cuando `G[i,j]`$=\infty$ si el grafo es ralo ($a << n^2$) representando al grafo con una lista de adyacencia.
- para seleccionar el próximo candidato se puede usar un heap; pero será necesario actualizarlo cada vez que se elimina el elemento y cada vez que se modifica alguna distancia en d.
- el tiempo total es de $\Theta((a+n)\log n) = \Theta(a \log n)$. ¿porqué?
- el algoritmo de Dijkstra se puede describir como un algoritmo de recorrido que usa la E.D. *Heap*

Correctitud del Algoritmo de Dijsktra

Teorema 20 (Correctitud del algoritmo de Dijsktra)

El Algoritmo de Dijsktra es siempre correcto si G no contiene arcos con peso negativo

Prueba.

Se prueba que el algoritmo termina (siempre se agotan los candidatos), y que al finalizar cada iteración del ciclo greedy, vale $d[v] = \delta[1, v]$ para el nodo v seleccionado en esa iteración (ver lema siguiente). Al terminar el algoritmo entonces $d[u] = \delta(1, u)$ para todo $u \in N$. □

Correctitud del Algoritmo de Dijsktra

Lema 21

Al finalizar cada iteración i, vale que $d[u]_i = \delta(1,u)$ para todo $u \in S_i$

Prueba.

Por inducción en i. Para $i = 0$ es trivial. Para $i > 0$, $S_i = S_{i-1} + \{v\}$. Si $u \in S_{i-1}$ vale por H.I. Si $u = v$ el nodo seleccionado en i, se supone por el absurdo $\delta(1,v) < d[v]_i$. Luego $\delta[1,v] < \infty$ y existe un camino más corto $1 \rightsquigarrow^{p}_{G} v$. Sea y el primer nodo de p tal que $y \notin S_{i-1}$, y x su predecesor. Como $x \in S_{i-1}$, $d[x]_{i-1} = \delta(1,x)$ y el arco (x,y) fue relajado cuando se agregó x a S. Luego $d[y]_{i-1} = \delta(1,y)$ (lema 19), y vale $d[y]_i \leq d[y]_{i-1} = \delta(1,y) \leq \delta(1,v) < d[v]_{i-1} = d[v]_i$ (lema 18) . Y según el algoritmo $d[v]_i \leq d[y]_i$, contradiciendo la suposición. □

Subgrafos de cubrimiento

- sea $G = \langle N, A \rangle$ un grafo no dirigido, conexo y con pesos.

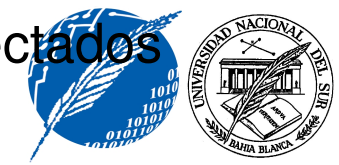
Definición 1

un subgrafo de cubrimiento es un subgrafo $G' = \langle N, A' \rangle, A' \subseteq A$, que también es conexo.

Definición 2

un subgrafo de cubrimiento es de cubrimiento mínimo si la suma de los arcos de A' es minimal entre todos los grafos de cubrimiento de G.

- los subgrafos de cubrimiento mínimo son interesantes de calcular porque representan una forma óptimal de mantener conectados todos los nodos de un grafo



Árboles de cubrimiento mínimo

Lema 22

Sea G un grafo no dirigido, conexo y con pesos, entonces todo subgrafo de cubrimiento mínimo de G es un árbol (no tiene ciclos).

Lema 23

Sea G un grafo no dirigido, conexo y con pesos, entonces todo árbol de cubrimiento mínimo de G tiene $n-1$ arcos, siendo $n = |N|$.

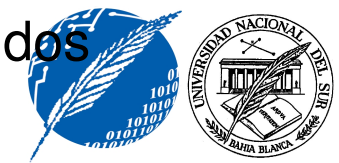
Lema 24

Sea G un grafo no dirigido, conexo y con pesos, entonces siempre posee al menos un árbol de cubrimiento mínimo.

- las demostraciones quedan como ejercicios

Árboles de cubrimiento mínimo

- ÁRBOL DE CUBRIMIENTO MÍNIMO es el problema computacional de, dado un tal G, encontrarle un árbol de cubrimiento mínimo.
- un algoritmo *greedy* para solucionar este problema tendría como conjunto de candidatos a A.
- un conjunto es una solución si contiene $n-1$ arcos (es un árbol y cubre todos los nodos).
- el control de viable se puede realizar controlando por la no existencia de ciclos (los candidatos deben siempre formar un árbol).
- de acuerdo a distintas funciones de selección se definen dos algoritmos para este problema: Kruskal y Prim.



- se puede demostrar que tanto el algoritmo de Kruskal como el de Prim son correctos (esto es muy inusual para algoritmos *greedy*!)
- para ambas demostraciones de correctitud se necesitará:

Definición 3

sea $T \subseteq A$, T es promisorio si está incluido en una solución optimal, ie *si está incluido en un árbol de cubrimiento mínimo.*

Definición 4

sea $B \subseteq N$, y $(u, v) \in A$. Se dice que (u, v) toca B si ($u \in B$ y $v \notin B$), o ($u \notin B$ y $v \in B$).

- también será necesario el siguiente lema:

Lema 25

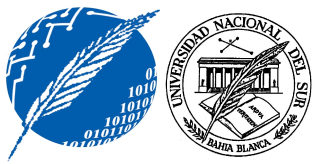
sea $G = \langle N, A \rangle$ un grafo no dirigido, conexo, con pesos; $B \subset N$ y $T \subseteq A$ un conjunto promisorio de arcos tal que ninguno de sus miembros toca B. Luego si (u, v) es uno de los arcos minimales que tocan B, entonces $T \cup \{(u, v)\}$ también es promisorio.

Prueba.

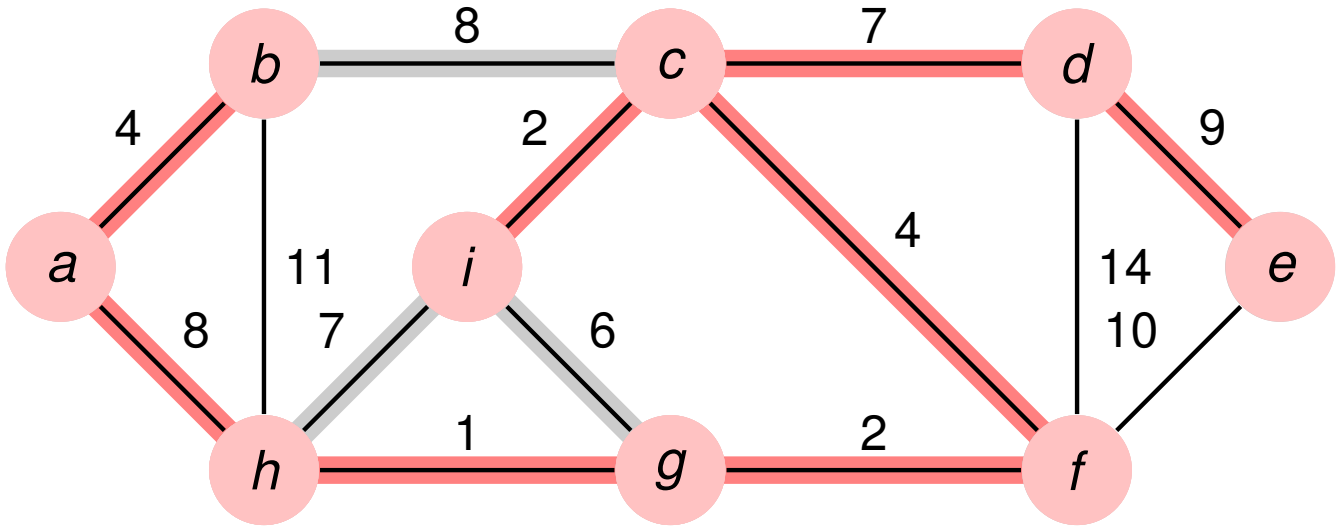
Sea G' el árbol de cubrimiento mínimo que contiene a T. Si $(u, v) \in G'$, entonces $T \cup \{(u, v)\}$ es promisorio. Si $(u, v) \notin G'$ y suponemos por el absurdo que $T \cup \{(u, v)\}$ no es promisorio, entonces existe G'' arbol de cubrimiento tal que $peso(G'') < peso(G')$. □

Algoritmo de Kruskal

- el Algoritmo de Kruskal es un algoritmo *greedy* que resuelve el problema de encontrar un árbol de cubrimiento mínimo.
- se caracteriza por seleccionar en cada iteración el menor de los arcos todavía no considerados.
- si el arco seleccionado junto con la solución parcial es viable, entonces se incluye en la solución parcial. En caso contrario, es descartado.
- se puede demostrar que es correcto, *ie* que siempre encuentra un árbol de cubrimiento mínimo para un grafo no dirigido, conexo y con pesos.



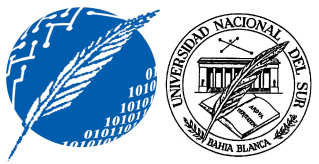
Ejemplo de ejecución del algoritmo de Kruskal



Componentes conexos: $\{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}\}$
Componentes conexos: $\{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}\}$
Componentes conexos: $\{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g,h\},\{i\}\}$
Componentes conexos: $\{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g,h\},\{i\}\}$
Componentes conexos: $\{\{a\},\{b\},\{c,i\},\{d\},\{e\},\{f\},\{g,h\}\}$
Componentes conexos: $\{\{a\},\{b\},\{c,i\},\{d\},\{e\},\{f\},\{g,h\}\}$

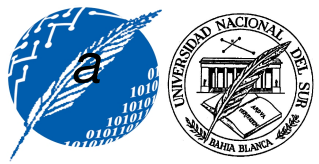
Implementación

- ordenar los arcos al principio para que la selección sea eficiente.
- usar conjuntos disjuntos para controlar si un nuevo arco conecta componentes distintos.



Pseudocódigo del algoritmo de Kruskal

	costo	veces
`ordenar A en L; n::= |N|; T::={}`	$\Theta(a \log a)$	1
`D.initiate(N)`	b	1
`REPEAT  //ciclo greedy`	$\Theta(n)$	1
`  (u,v)::= primero(L) // lo remueve`		
`  compu::= D.find(u)`	c	a
`  compv::= D.find(v)`	$\times$	a
`  IF (compu != compv)`	c	a
`     D.merge(u,v)`	$\times$	$n-1$
`     T::= T + {(u,v)}`	c	$n-1$
`  ENDIF`		
`UNTIL (|T|=n-1)`	c	
`RETURN T`		



Análisis del tiempo de ejecución

- el tiempo de ejecución resulta:

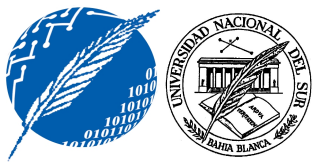
$$\begin{aligned} T(G) &= \Theta(a \log a) + \Theta(a) + \Theta(n) + O((2a + n - 1) \log^* n) \\ &= \Theta(a \log n) + O(a \log^* n) \\ &\in \Theta(a \log n) \end{aligned}$$

- considerando que:
  - si G es conexo $n - 1 \leq a \leq n(n-1)/2$.
  - K llamadas a operaciones *find*() y *merge*() en una E.D. de conjuntos disjuntos de n elementos lleva tiempo de $O(K \log^* n)$
  - $\log^* n \in O(\log n)$, pero $\log n \notin O(\log^* n)$.

Otra implementación:

(ejercicio)

- en lugar de ordenar los arcos al principio, se usa un *heap* invertido para obtener el arco minimal en cada iteración.
- disminuye el tiempo de inicialización, pero aumenta el del cuerpo del ciclo.
- el orden exacto del tiempo de ejecución obtenido es el mismo, pero las constantes ocultas por la notación asintótica serían menores.



Correctitud

Teorema 26 (Correctitud del algoritmo de Kruskal)

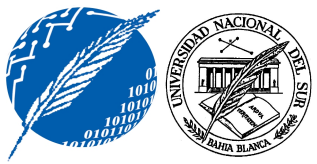
El algoritmo de Kruskal devuelve en T un árbol de cubrimiento mínimo para todo G conexo

Prueba.

Por inducción sobre i, se prueba que todo T_i es promisorio usando el lema 25, considerando como B al componente conexo de alguno de los extremos del (u, v) elegido en cada iteración. Luego el T_i final es una solución optimal porque no puede tener más arcos. □

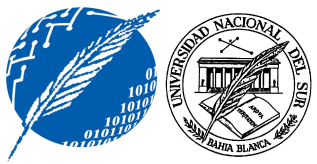
Características generales

- en Kruskal la función de selección elige arcos sin considerar la conexión con los arcos precedentes.
- el algoritmo de Prim se caracteriza por hacer la selección en forma local, partiendo de un nodo seleccionado y construyendo el árbol en forma ordenada.
- dado que el arco es seleccionado de aquellos que parten del árbol ya construído, la viabilidad está asegurada.
- también se puede demostrar que el algoritmo de Prim es correcto, es decir que devuelve un árbol de cubrimiento mínimo en todos los casos.



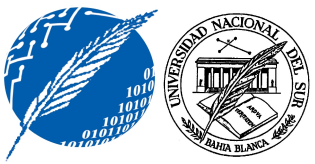
Esquema general del algoritmo de Prim

```
T ::= {}
B ::= {un nodo de N}
WHILE (B != N)      // Ciclo greedy
   encontrar (u,v) tal que peso((u,v)) sea
      mínimo  con u en B y v en N-B
   T ::= T+{(u,v)}
   B ::= B+{v}
ENDWHILE
RETURN T
```



Implementación

- suponer G representado por una matriz de adyacencia.
- usar un arreglo distB[] para conocer cuál es la distancia a B de cada nodo, o si el nodo ya está en B
- "relajar" el arreglo distB[] con los arcos adyacentes a un nodo cada vez que ese nodo se agrega a B
- usar un arreglo masCercano[] para conocer con cuál nodo de B se tiene esa distancia, y eventualmente recuperar al final el árbol de cubrimiento mínimo

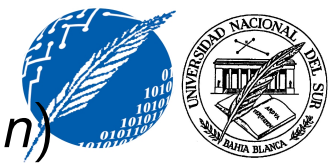


Pseudocódigo del algoritmo de Prim y tiempo de ejecución

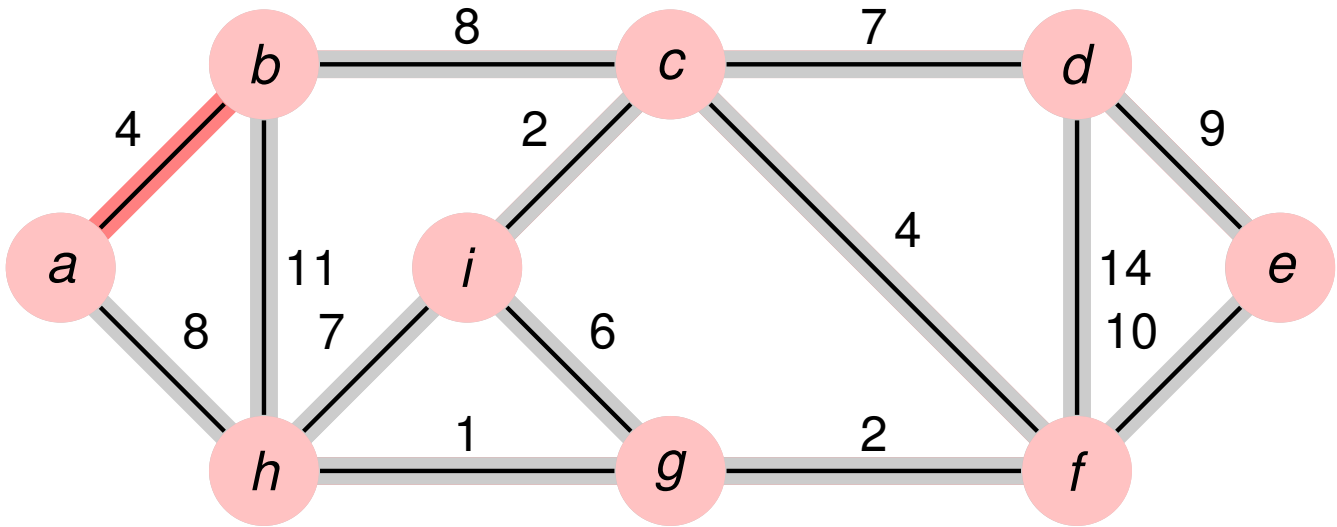
	costo	veces
`// Inicialización`		
`B ::= {1}; T ::= {}`	b	1
`FOR i ::=2 TO n`	b	$n-1$
`  distB[i] ::= G[1,i]`	b	$n-1$
`  IF (G[1,i]<MaxInt) THEN`	b	$n-1$
`  masCercano[i] ::= 1`	b	$n-1$
`  ELSE`		
`  masCercano[i] ::= 0`	b	$n-1$
`  ENDIF`		
`ENDFOR`		

Pseudocódigo del algoritmo de Prim

	costo	veces
`REPEAT // Ciclo greedy`		
`k::= nodo con distB[k]>0`	$\Theta(n)$	$n-1$
`y que minimice distB[k]`		
`T::= T+{(k,masCercano[k])}`	$\Theta(1)$	$n-1$
`B::= B+{k}; distB[k]::= 0`	$\Theta(1)$	$n-1$
`FOR cada j adyacente a k en G`	$\Theta(1)$	$\Theta(2a)$
`IF G[k,j]<distB[j] //RELAXATION`	$\Theta(1)$	$\Theta(2a)$
`dist[j]::= G[k,j]`	$\Theta(1)$	$\Theta(2a)$
`masCercano[j] ::= k`	$\Theta(1)$	$\Theta(2a)$
`ENDIF`		
`ENDFOR`		
`UNTIL |B|=n`	$\Theta(1)$	$\Theta(n)$



Ejemplo de ejecución del algoritmo de Prim



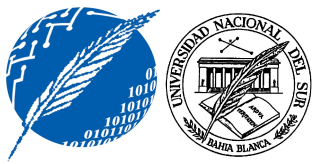
	a	b	c	d	e	f	g
distB[]	0	4 0	∞ 8 0	∞ 7 0	∞ 10 9 0	∞ 4 0	∞ 6 2 0
masCercano[]	0	a	0 b	0 c	0f d	0 c	0i

Comparación entre Kruskal y Prim

- el tiempo de ejecución de Prim resulta $T(n) \in \Theta(n^2)$, ya que $n-1 \leq a \leq n(n-1)/2$.
- entonces

Algoritmo de Kruskal	Algoritmo de Prim
$\Theta(a \log n)$	$\Theta(n^2)$

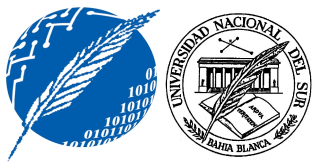
- en resumen
  - si $a \approx n$ conviene utilizar Kruskal
  - si $a \approx n^2$ conviene utilizar Prim



Otra implementación del algoritmo de Prim

(ejercicio)

- se usa un *heap* invertido para obtener el arco minimal en cada iteración.
- el orden del tiempo de ejecución en este caso es $\Theta(a \log n)$.
- ¿cómo cambia el tiempo de ejecución si se representa al grafo con una lista de adyacencia?



Correctitud

Teorema 27

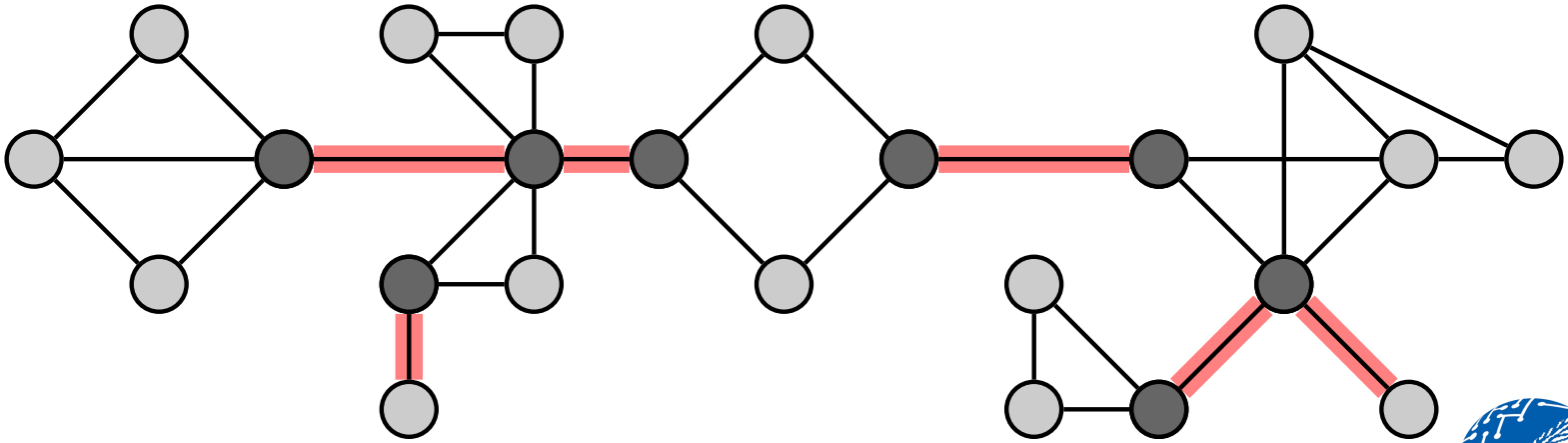
El algoritmo de Prim devuelve en T un árbol de cubrimiento mínimo para todo G conexo

Prueba.

Por inducción sobre i, se prueba que todo T_i es promisorio usando el lema 25, con el mismo B del algoritmo. Luego el T_i final es una solución optimal porque no puede tener más arcos. □

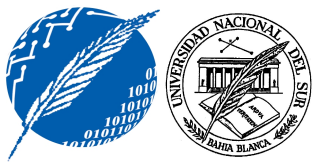

Definición de los problemas

- sea G no dirigido, conexo. Un puente es un arco de G cuya eliminación deja al grafo resultante disconexo
- un punto de articulación es un nodo de G cuya eliminación (junto con todos sus arcos incidentes) deja al grafo resultante disconexo



Definición de los problemas

- Problema: PUENTES es encontrar todos los puentes de un grafo no dirigido conexo G
- Problema: PUNTOS DE ARTICULACIÓN es encontrar todos los puntos de articulación en G
- un sólo recorrido DF servirá de punto de partida para solucionar los dos



Puentes

- para determinar los puentes es útil la siguiente propiedad

Lema 28

Sea $G = \langle N, A \rangle$ un grafo no dirigido. Un arco $(u, v) \in A$ es un puente en G si y solo si (u, v) no pertenece a ningún ciclo simple en G.

Demostración.

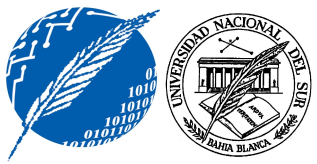
$\Rightarrow$): entonces removiendo (u, v) existiría un otro camino de u a v, con lo que el grafo resultante sería conexo, contradiciendo el hecho de que es puente. $\Leftarrow$): luego el grafo obtenido removiendo (u, v) es conexo por lo que existe un camino de u a v que no use (u, v). Tomando ese camino más (u, v) se forma un ciclo simple, con lo se obtiene una contradicción. □

Puentes

- se puede obtener entonces un algoritmo en tiempo $O(a)$ que encuentre los puentes, a partir de un único recorrido DF del grafo
- cada vez que se encuentra un arco hacia atrás (que equivale a un ciclo simple por el lema 11), se marcan todos los arcos del ciclo como “no puentes”. Al finalizar el recorrido, los arcos que quedan sin marcar son puentes
- ejercicio: especificar un algoritmo para encontrar los puentes en un sólo recorrido del grafo


Puntos de articulación

- las siguientes propiedades caracterizan puntos de articulación para los nodos iniciales e intermedios en un recorrido



Punto de articulación - nodo inicial del recorrido

Lema 29

Sea G un grafo no dirigido conexo, y consideremos $n \in N$ el nodo inicial de un recorrido DF. Entonces n es un punto de articulación si y solo tiene al menos dos hijos.

Demostración.

Es claro el si. Para el solo si, sean n_1, n_2 dos descendientes de n. Si n no fuera punto de articulación existiría camino blanco entre n_1 y n_2 en el momento de visitar el primero de ellos, con lo que el segundo sería descendiente del primero. □

Puntos de articulación - nodo intermedio del recorrido

Lema 30

Sea G un grafo no dirigido conexo, y consideremos $n \in N$ un nodo no inicial en un recorrido DF. Entonces $n \in N$ es punto de articulación si y solo existe un hijo u de n tal que ningún descendiente v de u tiene un arco (v, w) donde w es ancestro propio de n.

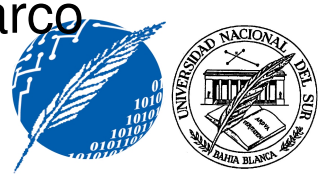
Demostración.

Es claro que si n es punto de articulación entonces tal arco no puede existir. Para el solo si, suponemos que existe u hijo de n tal que ninguno de sus descendientes v tiene un arco hacia un ancestro propio w de n. Entonces, removiendo n del grafo no existe ningún camino posible de u a w, por lo que n es punto de articulación. □

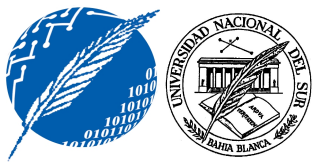
Punto de articulación - nodo intermedio del recorrido

- de acuerdo al lema anterior, para saber si un nodo n es un punto de articulación es suficiente con verificar si se puede acceder a un ancestro propio desde un descendiente de n por un arco fuera de la foresta
- teniendo en cuenta la numeración de descubrimiento generada por un recorrido DF, se puede computar para cada nodo el ancestro más viejo que se alcanza a través de arcos hacia atrás

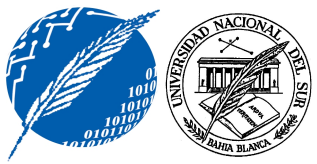
$$\texttt{masviejo}[v] = min \begin{cases} \texttt{d}[v] \\ \texttt{d}[w] : (v,w) \text{ es un arco hacia atrás} \\ \texttt{masviejo}[w] : (v,w) \text{ es un arco de la foresta} \end{cases}$$



```
PROCEDURE PuntosArticulación(G)
 array d[1..n], masviejo[1..n], padre[1..n]
 DFS(G,d)
 calcular masviejo en otro recorrido DFS de G
 pa[1]::= tiene más de un hijo en la foresta
 FOR cada v en  N-{1}
   pa[v]::= existe un hijo u de v
              tal que d[v]<=masviejo[u]
 ENDFOR
 RETURN pa
```

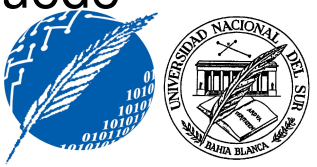


- el tiempo de ejecución es de $\Theta(a)$ (dos recorridos más una iteración sobre todos los nodos)
- se podrían mejorar las constantes realizando el cálculo completo en un sólo recorrido (ejercicio)



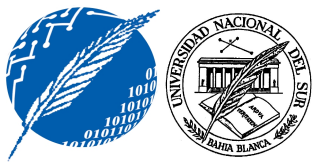
Definición del problema

- una red de flujo es un grafo dirigido $G = \langle N, A \rangle$ donde cada arco $(u, v) \in A$ tiene asociado una capacidad $c(u, v) \geq 0$, y se distinguen dos nodos s y t llamados fuente y destino, tal que ningún arco llega a la fuente, o sale del destino
- se supone que si $(u, v) \notin A$ entonces $c(u, v) = 0$
- además, por conveniencia, se requiere que $(u, u) \notin A$; si $(u, v) \in A$ entonces $(v, u) \notin A$; y que para todo $n \in N$, $s \rightsquigarrow_G n \rightsquigarrow_G t$
- estas restricciones facilitan la especificación de los algoritmos, y no son esenciales (cualquier grafo que no las cumpla se puede transformar fácilmente en uno que la cumple)

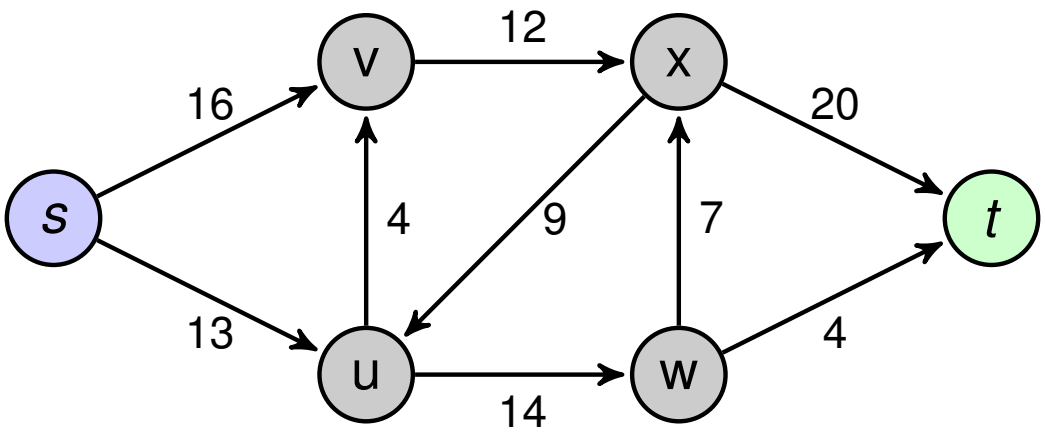


Definición del problema

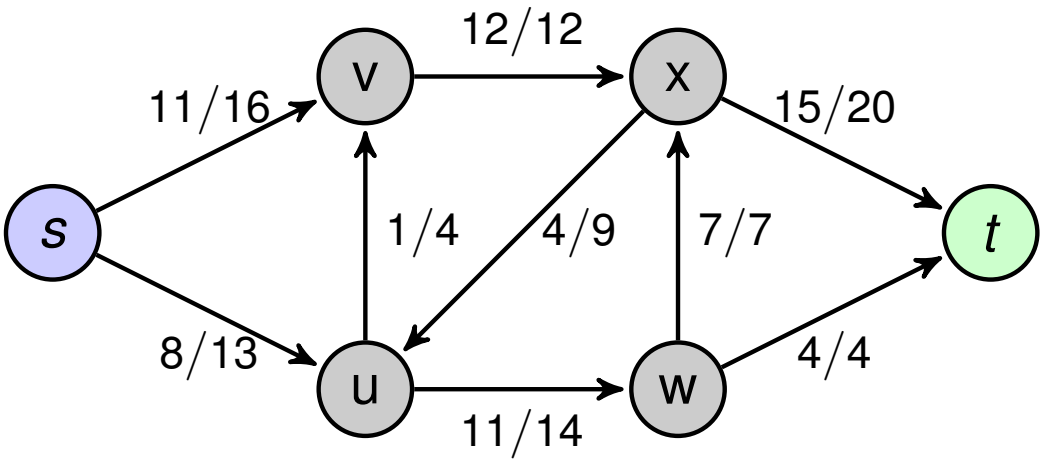
- un flujo en G es una función $f : N \times N \longrightarrow \mathbf{R}^+$ que satisface:
  - restricción de capacidad: $0 \leq f(u,v) \leq c(u,v)$ para todo $u, v \in N$
  - conservación de flujo: $\sum_{v \in N - \{s,t\}} f(u,v) = 0$
- el valor de un flujo $|f|$ se define como $|f| = \sum_{v \in N} f(s,v) - \sum_{v \in N} f(v,s)$
- Problema: dada una red de flujo G, el problema MAXFLUJO consiste en encontrar el máximo flujo que admite G



Ejemplo de red de flujo



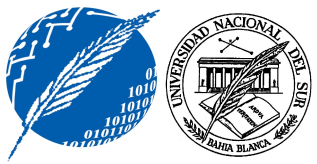
Ejemplo de flujo



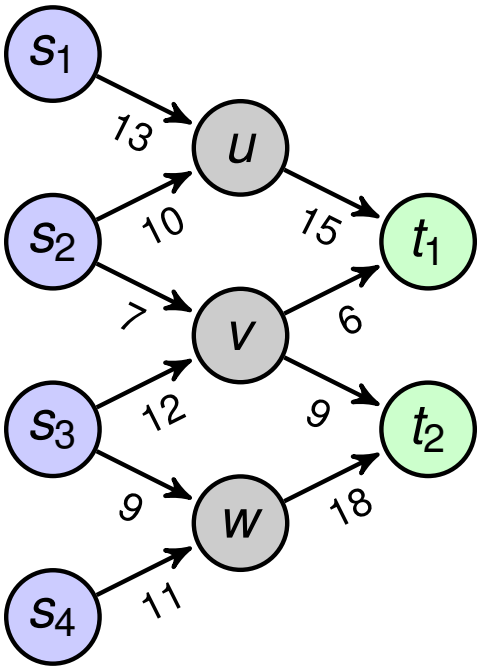
$|f| = 19$

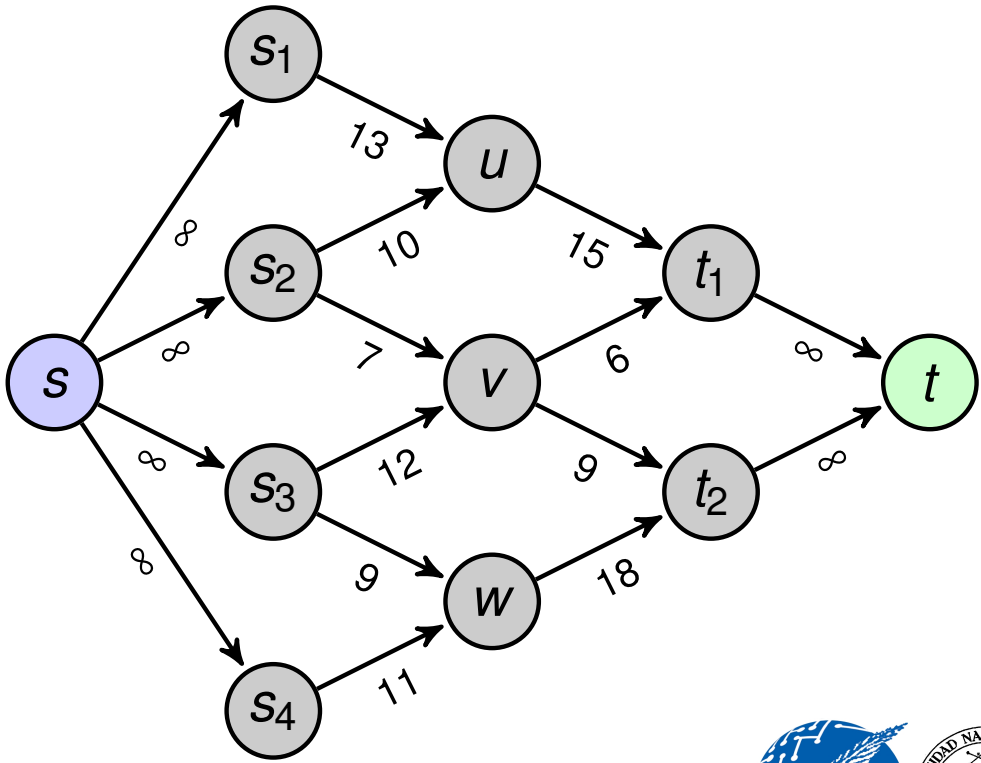
Otras restricciones

- se supondrá que todas las capacidades son enteros
- si las capacidades son racionales, se pueden escalar para que se consideren enteros
- la existencia de un único origen y un único destino también es una restricción que se puede levantar fácilmente



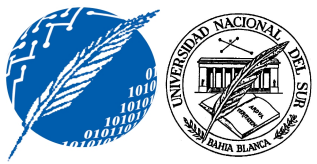
Otras restricciones





Método de Ford-Fulkerson

- el método de Ford-Fulkerson permite resolver el problema MAXFLUJO
- lo llamamos método porque produce distintos algoritmos, con distintos tiempos de ejecución de acuerdo a la implementación
- se basa en los conceptos de red residual, incremento, camino de aumento y corte

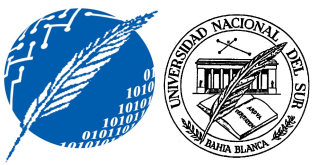


Red residual

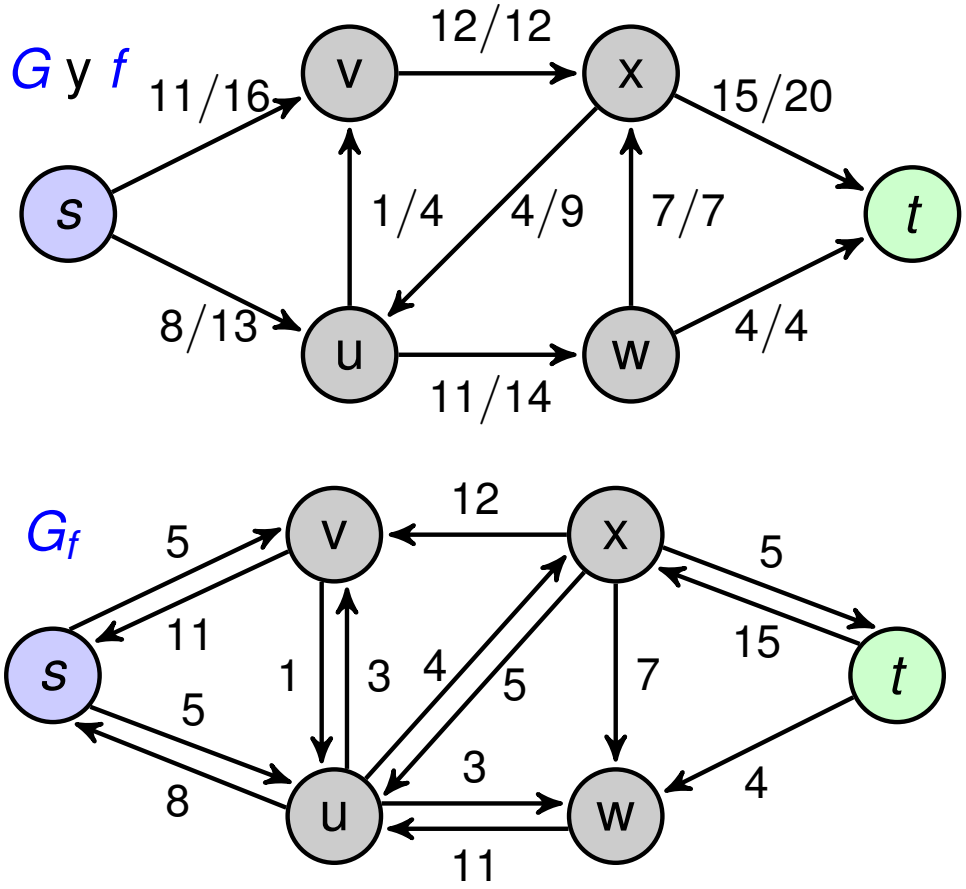
- sea $G = \langle N, A \rangle$ una red de flujo, y f un flujo en G. Se define para cada par de vértices $u, v \in N$ la capacidad residual $c_f(u, v)$ como

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{si } (u,v) \in A \\ f(v,u) & \text{si } (v,u) \in A \\ 0 & \text{si no} \end{cases}$$

- por la restricción en la definición de red de flujo, se puede ver que sólo una de estas opciones se aplica en cada caso
- una red residual es un grafo $G_f = \langle N, A_f \rangle$, donde los arcos son $A_f = \{(u,v) : c_f(u,v) > 0\}$ y la capacidad es precisamente $c(u,v) - f(u,v)$

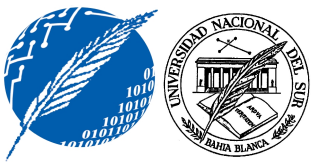


Ejemplo de red residual



Red de flujo vs. red residual

- no necesariamente una red residual es una red de flujo de acuerdo a nuestra definición, ya que puede haber arcos en ambos sentidos entre un mismo par de nodos
- aparte de esta diferencia, la red residual tiene las mismas propiedades que una red de flujo, y se puede definir un flujo en la red residual, que satisface las propiedades de flujo pero con respecto a las capacidades c_f definidas en G_f
- cualquier flujo en la red residual provee las bases para aumentar el flujo original f en G



Incremento

- sea f un flujo en una red de flujo G, y f' un flujo en G_f. Se define el incremento $f \uparrow f'$ como la función $N \times N \to \mathbf{R}$ definida

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{si } (u,v) \in A \\ 0 & \text{si no} \end{cases}$$

Lema 31

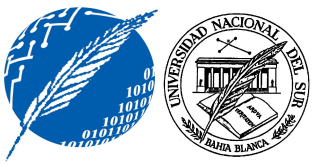
Sea G una red de flujo, f un flujo sobre G, G_f la red residual. Entonces el incremento $f \uparrow f'$ es un flujo sobre G tal que $| f \uparrow f' | = | f | + | f' |$.

Demostración.

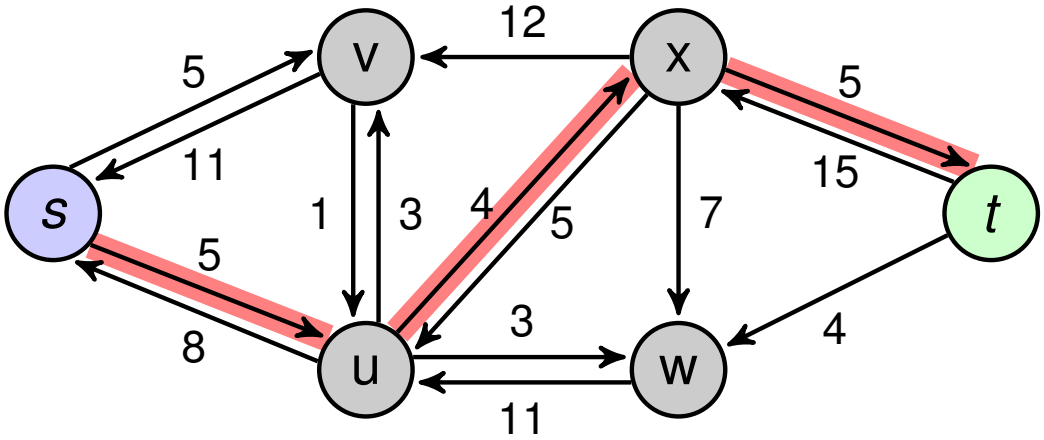
Se debe verificar que cumple con las dos propiedades de flujo, y que su valor es la suma de los valores de f y f' (ejercicio). □

Camino de aumento

- entonces, dado un flujo f sobre G, un incremento sobre la red residual G_f permite crear un nuevo flujo de mayor valor. El punto faltante es cómo encontrar un incremento
- un camino de aumento es un camino $p : s \rightsquigarrow_{G_f} t$ en la red residual G_f
- la capacidad residual de p es la mínima capacidad de los arcos que pertenecen al camino, y se nota $c_f(p)$



Ejemplo de camino de aumento



camino de aumento p con $c_f(p) = 4$

Lema 32

Sea G una red de flujo, f un flujo sobre G y p un camino de aumento en G_f. Entonces el siguiente f_p es un flujo en G_f tal que $\mid f_p \mid = c_f(p)$

$$f_p(u,v) = \begin{cases} c_f(p) & si\ (u,v)\ está\ en\ p \\ 0 & sino \end{cases}$$

Demostración.

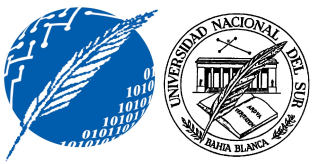
Es suficiente con probar las propiedades de flujo, y verificar el valor de f_p. □

Corolario 33

Sea f un flujo en G, y p un camino de aumento en G_f. Entonces $f \uparrow f_p$ es un flujo en G con valor $\mid f \uparrow f_p \mid$

Flujos incrementales

- el corolario anterior permite definir un algoritmo iterativo, a partir de un flujo nulo, que compute flujos cada vez más grandes en una red de flujo
- se denomina método de Ford-Fulkerson, y en cada iteración encuentra un camino de aumento, obtiene el flujo de ese camino y se lo suma al flujo anterior, resultando en un flujo con valor mayor
- se comienza a partir de un flujo vacío



Método de Ford-Fulkerson

```
PROCEDURE FordFulkerson (G,s,t,c)
  FOR cada arco (u,v) en A
    f[u,v]::=0
  WHILE existe un camino de aumento p en Gf
    cf(p)::=min{(c[u,v]-f[u,v] si (u,v) en p}
    FOR cada arco (u,v) en p
      IF (u,v) está en A
         f[u,v]::=f[u,v]+cf(p)
      ELSE
        f[v,u]::= f[v,u]-cf(p)
    ENDFOR
  ENDWHILE; RETURN f
```

Ejemplo de ejecución de Ford-Fulkerson

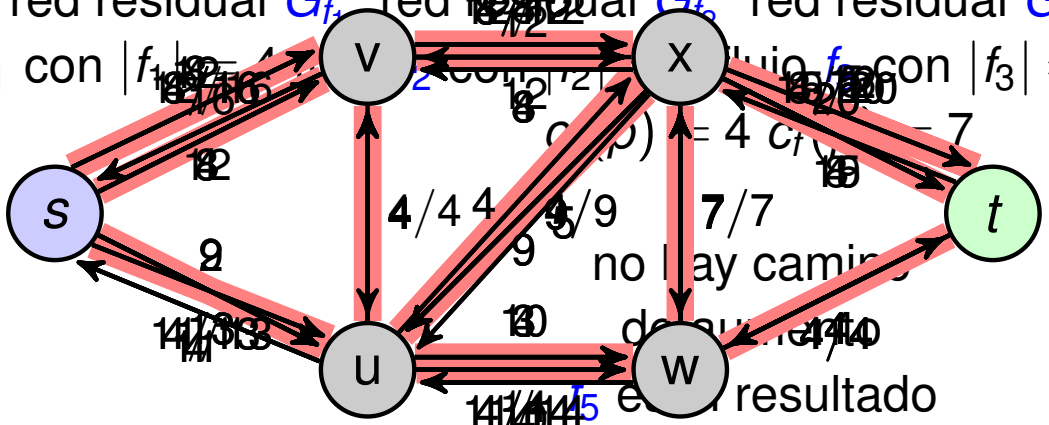
red de flujo G red residual G_{f_1} red residual G_{f_2} red residual G_{f_3} red residual G

flujo f_0 con $|f_0| = 0$ flujo f_1 con $|f_1|$ f_2 con $|f_2|$ f_3 con $|f_3| = 12$ flujo f_4 con

$c_f(p) = 4$ $c_f($ $= 7$

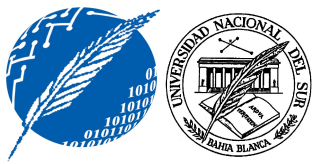
no hay camino de aumentante

el resultado f_5



Propiedades

- distintas implementaciones de cómo encontrar el camino de aumento determinan distintos tiempos de ejecución
- para afirmar que computa el flujo máximo, faltaría demostrar que cuando el algoritmo termina (es decir, cuando no hay más camino de aumento) entonces en ya se tiene el flujo máximo
- para probar esto es necesario el concepto de corte

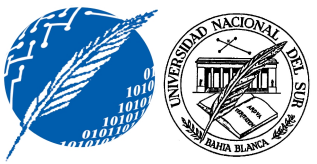


Corte de una red de flujo

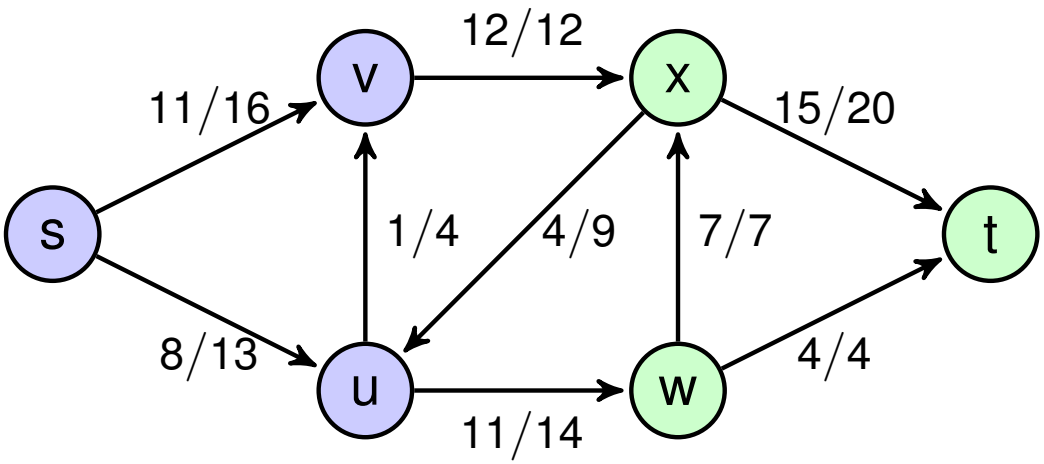
- G una red de flujo, y f un flujo sobre G. Un corte es una partición de N en S y $T = N - S$ tal que $s \in S$ y $t \in T$
- el flujo neto a través el corte (S, T) es

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u)$$

- la capacidad de un corte (S, T) es $c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$
- el corte mínimo de una red es un corte cuya capacidad es mínima entre todos los cortes



Ejemplo de corte



$$S = \{s, u, v\}$$
$$T = \{x, w, t\}$$
$$c(S, T) = 26$$
$$f(S, T) = 19$$

Propiedades de un corte

Lema 34

Sea f un flujo en una red de flujo G, y S, T un corte cualquiera en G. Entonces el flujo neto a través del corte $f(S,T) = |f|$.

Demostración.

A partir de la definición de $|f|$, y partiendo los nodos de N en S y T. □

Corolario 35

El valor de un flujo cualquiera f en una red G está acotado por la capacidad de un corte de G, ie $|f| \leq c(S,T)$ para cualquier corte S, T.

Correctitud de Ford-Fulkerson

Teorema 36 (Máximo flujo, mínimo corte)

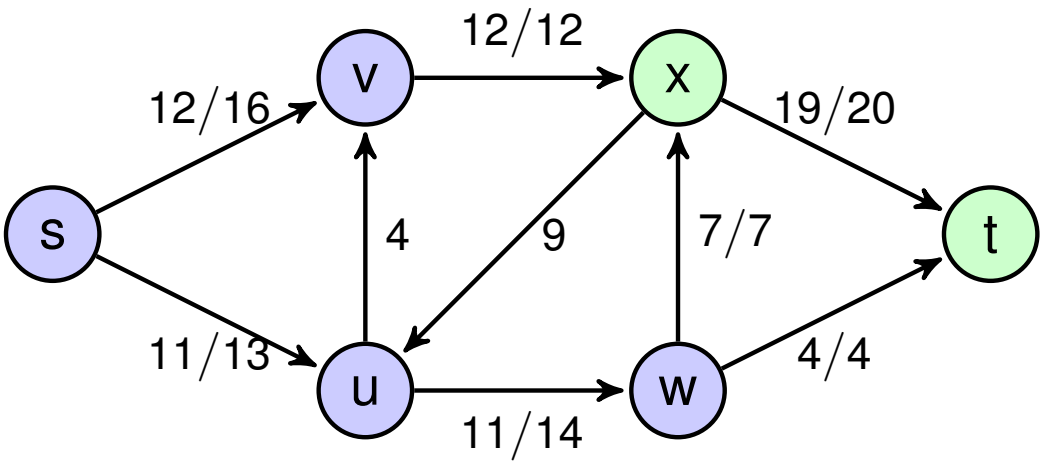
Sea G una red de flujo, y f un flujo en G. Entonces son equivalentes:

1. *f es un flujo máximo de G*
2. *la red residual G_f no contiene caminos de aumento*
3. *existe un corte (S,T) en G tal que $|f| = c(S,T)$*

Demostración.

1)$\Rightarrow$ 2): si f es el máximo flujo y G_f tiene camino de aumento p existe f_p (lema 32) y $f + f_p$ es un flujo en G de mayor valor. 2)$\Rightarrow$ 3): sean $S = \{v \in N : s \rightsquigarrow_{G_f} v\}$ y $T = N - S$. Luego (S,T) es un corte y $|f| = f(S,T) = c(S,T)$. 3) $\Rightarrow$ 1): $|f| \leq c(S,T)$ para todo corte (S,T) (lema 35). Pero $|f| = c(S,T)$ implica que f es un flujo máximo. $\square$

Ejemplo de máximo flujo-mínimo corte



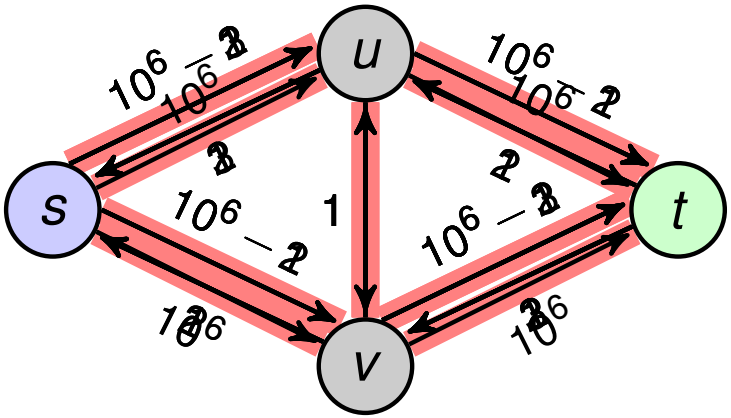
$$S = \{s, u, v, w\}$$
$$T = \{x, t\}$$
$$c(S, T) = 23$$
$$f(S, T) = 23$$

Análisis del tiempo de ejecución

	costo	veces
`PROCEDURE FordFulkerson (G,s,t,c)`		
`  FOR cada arco (u,v) en A`		
`    f[u,v]::=0; f[v,u]::=0`	b	a
`  ENDFOR`		
`  WHILE existe un camino de`	$\Theta(n+a)$	$O(\lvert f^* \rvert)$
`        aumento p en Gf`		
`   cf(p)::= capacidad mínima de p`	$O(a)$	$O(\lvert f^* \rvert)$
`   FOR cada arco (u,v) en p`		
`     f[u,v]::=f[u,v]+cf(p)`	b	$O(a\lvert f^* \rvert)$
`     f[v,u]::=-f[u,v]`	b	$O(a\lvert f^* \rvert)$
`   ENDFOR`		
`  ENDWHILE; RETURN f`	f^* es el resultado	


Análisis del tiempo de ejecución

- la dependencia del tiempo de ejecución en el valor del flujo, y no en la longitud de su representación, no es bueno



2×10^6 iteraciones!!

Algoritmo de Edmonds-Karp

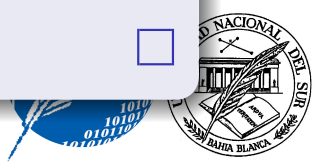
- el algoritmo de Edmonds-Karp es una instancia del método de Ford-Fulkerson en donde la búsqueda del camino de aumento p se hace mediante una búsqueda por niveles comenzando por s
- el recorrido por niveles permite encontrar los caminos mínimos (en cuanto a cantidad de arcos) desde el origen a cada nodo
- entonces en la red residual, el camino mínimo ya no existe. Esto permite ajustar la cantidad de iteraciones del ciclo WHILE


Lema 37

En el algoritmo EK, si $v \in N - \{s, t\}$ entonces la distancia mínima $\texttt{nivel}[v]$ *en G_f no disminuye con cada aumento de flujo.*

Demostración.

Supongamos que existen w tal que $\texttt{nivel}_{i+1}[w] < \texttt{nivel}_i[w]$. Sea v el nodo con menor nivel entre éstos, y $s \rightsquigarrow_{f_{i+1}} u \rightarrow v$ el camino de aumento, o sea $(u, v) \in A_{f_{i+1}}$. La elección de v se contradice tanto si $(u, v) \in A_{f_i}$, lo que implica $\texttt{nivel}_i[v] \leq \texttt{nivel}_i[u] + 1 \leq \texttt{nivel}_{i+1}[u] + 1 = \texttt{nivel}_{i+1}[v]$, como si $(u, v) \notin A_{f_i}$, en donde vale $\texttt{nivel}_i[v] = \texttt{nivel}_i[u] - 1 \leq \texttt{nivel}_{i+1}[u] - 1 = \texttt{nivel}_{i+1}[v] - 2$. □



Tiempo de ejecución

Teorema 38

El algoritmo EK en $G = \langle N, A \rangle$ toma $O(na)$ iteraciones.

Demostración.

Se muestra que cada arco (u, v) puede ser crítico (ie coincide con la capacidad residual) a lo sumo $n/2 - 1$ veces. Sea i la iteración donde (u, v) es crítico, y j la iteración donde (v, u) es crítico, luego $\texttt{nivel}_j[u] = \texttt{nivel}_j[v] + 1 \geq \texttt{nivel}_i[v] + 1 = \texttt{nivel}_i[u] + 2$, usando el lema 37. Con lo que la distancia mínima de u aumenta al menos en 2 entre cada par de iteraciones donde (u, v) es arco crítico. Y $(n-2)$ es una cota de la máxima distancia mínima. Entonces como hay $O(a)$ pares de vértices, no puede haber más de $O(na)$ iteraciones. □

Tiempo de ejecución

- como cada iteración (construir el grafo residual, hacer BFS, encontrar el camino de aumento y la capacidad residual, y actualizar el flujo) del algoritmo EK toma de $O(a)$, de acuerdo al teorema anterior el tiempo total de ejecución es de $O(na^2)$
- se elimina de esta forma la dependencia del tiempo de ejecución en f^*
- existen algoritmos asintóticamente mejores ($O(n^2a)$, $O(n^3)$) para el mismo problema