Grafos: ordenamiento topológico, conectividad y árboles de expansión mínima

UNIVERSIDAD DE LOS ANDES

Análisis y Diseño de Algoritmos

Prof. Isabel Besembel Carrera

Orden parcial

- Un orden parcial es una relación reflexiva, antisimétrica y transitiva.
- **Dominio:** es un conjunto de valores

  Ejm: D1 = {'rojo', 'verde', 'negro', 'azul'}

  D2 = {'ford', 'chevrolet', 'fiat`, 'toyota', 'renault'}
- **Relación:** es un subconjunto del producto cartesiano de una lista de dominios, no necesariamente disjuntos.

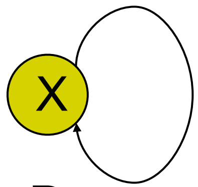
  Ejm: R1 = {('rojo', 'ford'), ('verde', 'ford'), ('negro', 'chevrolet'), ('azul', 'toyota')}

  R2 = {('fiat', 'verde')}

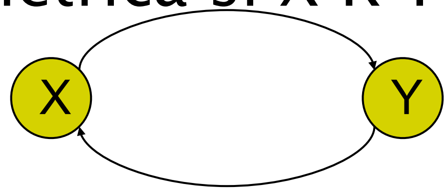
  R3 = { }

Propiedades de las relaciones

- Reflexividad: Una relación R es reflexiva si X R X para todo X en S.



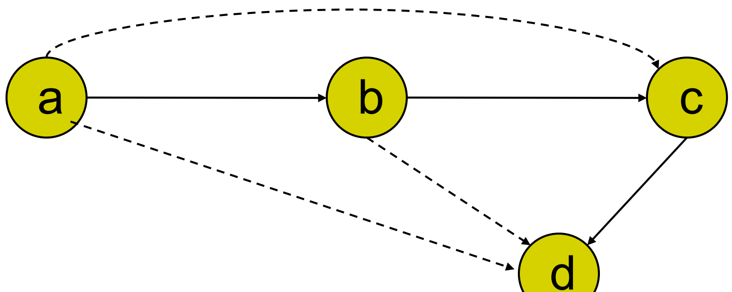
- Simetría: Una relación R es simétrica si X R Y implica Y R X para todo X y Y.



- Antisimetría: Una relación R es antisimétrica, si X R Y y Y R X implica X = Y, para todo X y Y.
  - La relación $\leq$ es antisimétrica, $x \leq p$ y $p \leq x$ implica que $x=p$

Propiedades de las relaciones

- Transitividad: Una relación R es transitiva si X R Y y Y R Z implica que X R Z, para todo X, Y, Z.

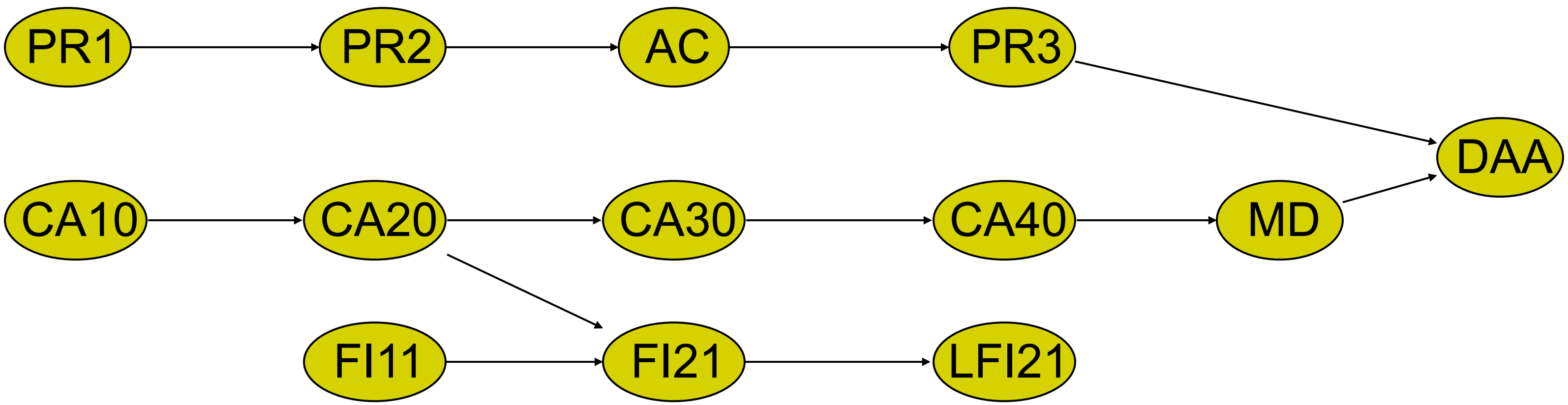


- La importancia de la teoría de las relaciones es que está relacionada con cualquier tipo de grafo.
- Un orden parcial (toda relación antisimétrica y transitiva) tiene un digrafo acíclico

Ordenamiento topológico

- Para un digrafo acíclico (dag) G = (N, A) el orden lineal de todos los nodos tal que si G contiene el arco (u, v), entonces u aparece antes de v en el orden dado.
- Si G tiene ciclos el orden lineal no es posible.
- Ordenamiento topológico de todos los nodos de un digrafo es una manera de visitar todos sus nodos, uno por uno, en una secuencia que satisface una restricción dada.

Ejemplo



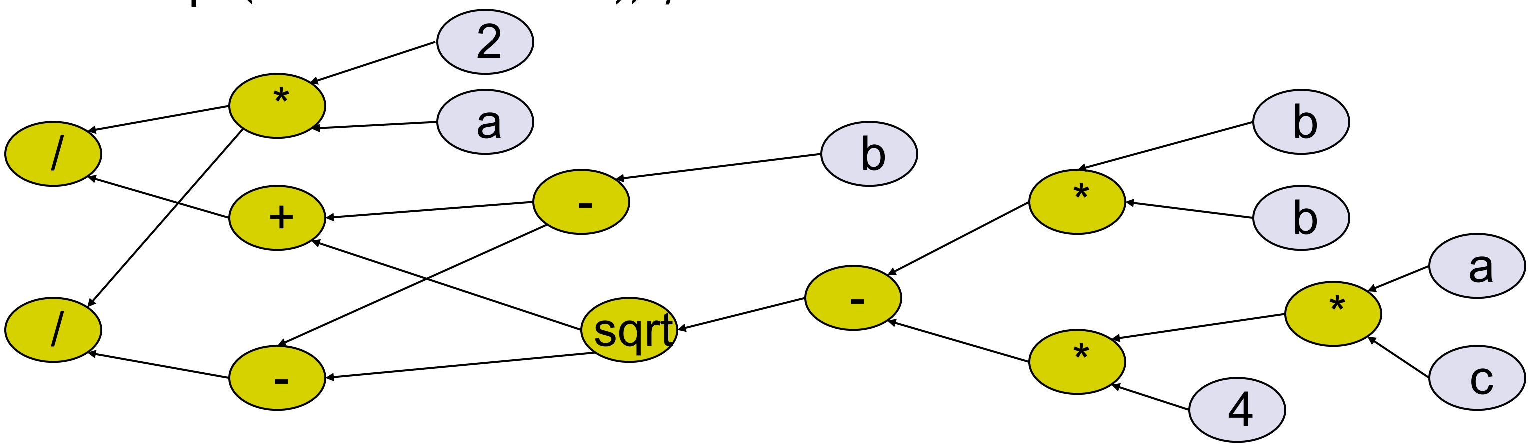
- Un orden topológico de los nodos de un digrafo G={N, A} es una secuencia (n1, n2, ..., nk) tal que N= {n1, n2, ..., nk} y para todo (ni, nj) en N, ni precede a nj en la secuencia.

Aplicaciones

- Modelado de actividades para resolver problemas de planificación o programación de las mismas siguiendo un criterio de minimización o maximización.
- Evaluación de expresiones aritméticas

(– b + sqrt(b * b – 4 * a * c)) / 2 * a

(– b – sqrt(b * b – 4 * a * c)) / 2 * a



Teorema (Szpilrajn, 1930)

- En cualquier grafo G con n≥1 nodos hay un nodo con grado de incidencia negativo (gin) igual a cero

Demostración: por contradicción

Si todos los nodos del digrafo G tienen un gin de al menos 1, entonces G contiene ciclos

Asuma que todos los nodos tienen un gin de al menos 1

Se comienza con cualquier nodo n1, se traza hacia atrás a lo largo de cualquier arista que incide en n2.

Desde n2 hasta n3 y así sucesivamente

Como todos los nodos tienen un gin de al menos 1, este proceso nunca acaba, pero como G es finito, eventualmente un nodo debe ser alcanzado cuando ya se ha pasado por él, por lo que G tiene ciclo

UNIVERSIDAD DE LOS ANDES

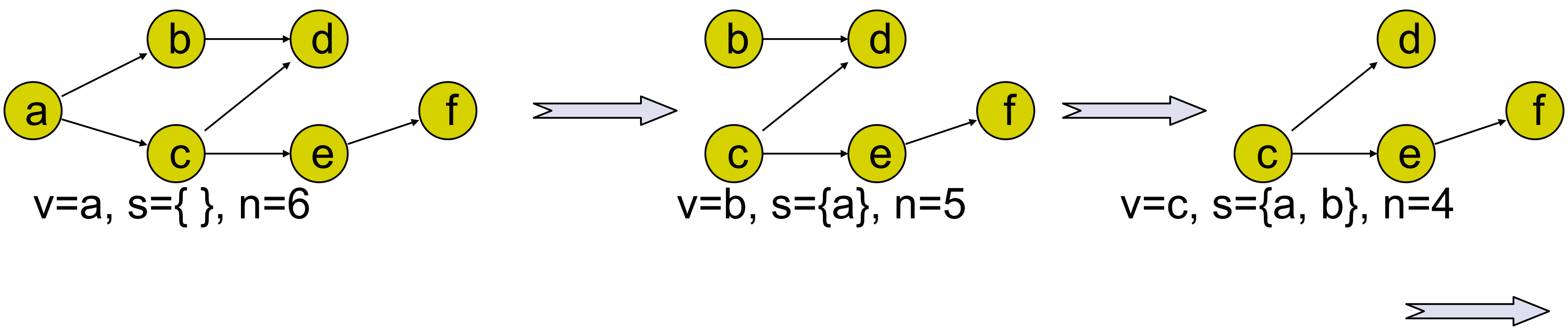
Algoritmo genérico de ordenamiento topológico

Abril/05

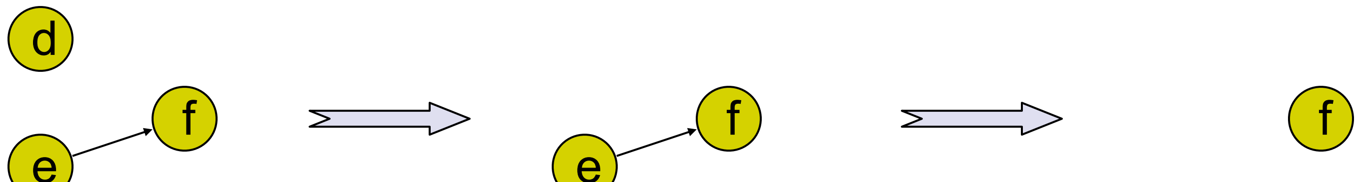
orden Topologico( ): Lista

{pre: n > 0 } {pos: n = 0 }

1	(n > 0) [v = nodo con gin = 0 elimine v y todas las aristas que salen de v s.inserLista(v)]	**-v:** Entero. Nodo del digrafo. **-inserLista().** Definida en Lista. **-s.** Lista. Lista que contiene el orden topológico de los nodos del grafo.
2	regrese s	



Algoritmo genérico de ordenamiento topológico



v=d, S={a, b, c}, n=3 v=e, S={a, b, c, d}, n=2 v=f, S={a, b, c, d, e}, n=1

- Al finalizar S={a, b, c, d, e, f} con n=0
- Teorema de la invariante del lazo: Sea un digrafo G={N, A}, entonces al comienzo de la k-ésima iteración del ordenTopologico( ), S=(n1, n2, …, nk-1) tiene la propiedad siguiente: para todo arco (ni, nj) en A, si nj está en S, entonces ni precede a nj en S.

  Demostración: por inducción sobre k

  Etapa base: k=1, S1={}, el teorema es cierto

  Etapa inductiva: hipótesis: Sea Sk={n1, n2, …, nk-1} el estado al comienzo de la k-ésima iteración y sea Gk el estado de G.

Algoritmo genérico de ordenamiento topológico

Por la hipótesis Sk+1={n1, n2,..., nk} donde nk es el nodo seleccionado por el algoritmo durante la k-ésima iteración.

Si nk existe, el teorema de Szpilrajn puede ser aplicado al dag Gk, como nk tiene gin=0 en Gk, en el digrafo original G, todos los predecesores de nk están en {n1, n2, …, nk-1}, lo que indica que para todo (ni, nj) en A, ni precede a nk en Sk+1.

Por lo que el teorema es cierto para Sk+1={n1, n2,...,nk}

- La invariante del lazo implica que Sn+1 es un orden topológico de los nodos de g y es correcto
- Todos los nodos de cualquier dag pueden ser ordenados topológicamente

Algoritmo de ordenamiento topológico

26/11/98

ordenTopologico( ): ListaDe [Entero]

{pre: $n > 0$ } {pos: $n > 0 \wedge G' = G \wedge$ padre $\supset$ bosque en profundidad }

1 2 3 4	[color(u), padre(u) = 'blanco', 0] u ∈ N t = 0 [Si (color(u) = 'blanco') entonces visitaBusPro(u, t, color, padre, d, f, lot) fsi] u ∈ N regrese lot	**-u, t:** Entero. Variable auxiliar y contador. **-color, padre, d, f:** Arreglo[n] De [Entero]. Contienen el color del nodo, el nodo predecesor inmediato, la etiqueta donde se empezó a procesar el nodo y donde se terminó de procesar cada nodo. **-visitaBusPro().** Rutina recursiva para hacer la búsqueda en profundidad. **-lot.** ListaDe [Entero]. Lista que contiene el orden topológico de los nodos del grafo.

Digrafo acíclico

- Lema 12: Un grafo G es acíclico si y solo si la búsqueda en profundidad de G no arroja arcos hacia atrás (arcos que conectan un nodo u a uno de sus ancestros v)

Algoritmo de búsqueda en profundidad

26/11/98

visitaBusPro(Entero: u, t, Arreglo[n]De [Entero]: color, padre, d, f, ListaDe [Entero]: lot)

{pre: $n > 0 \wedge u \in N$ } {pos: $n > 0 \wedge G' = G \wedge$ padre $\supset$ bosque en profundidad }

1	color(u), t, d(u) = 'gris', t + 1, t	**-u, v, t:** Entero. Variables auxiliares y contador.
2	[Si (color(v) = 'blanco') entonces padre(v) = u visitaBusPro(u, t, color, padre, d, f, lot) fsi] $\forall$ v $\in$ u.listaAdyacencia	**-color, padre, d, f:** Arreglo[n] De [Entero]. Contienen el color del nodo, el nodo predecesor inmediato, la etiqueta donde se empezó a procesar el nodo y donde se terminó de procesar cada nodo.
3	color(u), t, f(u) = 'negro', t + 1, t	**-insLista().** Función de la clase ListaDe [X].
4	lot.insLista(u)	
5	regrese	

$$T(n) = \Theta(N + A)$$

Componentes fuertemente conexas

- Un componente es fuertemente conexo en un digrafo G = (N, A) si el máximo conjunto de sus nodos $U \subseteq N$ tal que para cada par de nodos u y v en U, se tiene un camino desde u hasta v y viceversa, esto es u y v son alcanzables de uno a otro.
- Cálculo de los componentes fuertemente conexos de G: se realizan dos búsquedas en profundidad, la primera sobre G y la segunda sobre GT
- componentesFuertementeConexas( )
  1. G.busProf( )
  2. GT = G.transpuesto( )
  3. GT.busProf( ), pero en el lazo principal de busProf( ) considerar los nodos en orden decreciente de su f(u)
  4. Desplegar los nodos de cada árbol en profundidad de GT como una componente conexa para G.
- $T(n) = \Theta(N + A)$

Algoritmos voraces

- Características:
  - Para construir la solución del problema se tiene un conjunto de candidatos
  - Se van acumulando dos conjuntos: el de los considerados y seleccionados y el de los considerados y rechazados
  - Función de comprobación: si el conjunto de candidatos es una solución óptima o no
  - Función de factibilidad: si es posible o no completar el conjunto añadiendo otros candidatos para obtener una solución óptima o no
  - Función de selección: indica cuál es el más prometedor de todos los candidatos restantes, aquellos que no han sido ni seleccionados ni rechazados
  - Función objetivo: da el valor de la solución que se ha encontrado. No aparece explícita en el algoritmo

Algoritmos voraces

- Para resolver el problema, se busca un conjunto de candidatos que constituya una solución y que optimice la función objetivo
- Inicialmente el conjunto está vacío
- En cada paso, se considera añadir al conjunto el mejor candidato para la solución sin considerar los restantes utilizando la función de selección
- Si el conjunto ampliado ya no es factible, el candidato se pasa al conjunto de considerados y rechazados; sino se pasa al conjunto de seleccionados
- Cada vez que se amplía el conjunto de seleccionados se prueba si es una solución

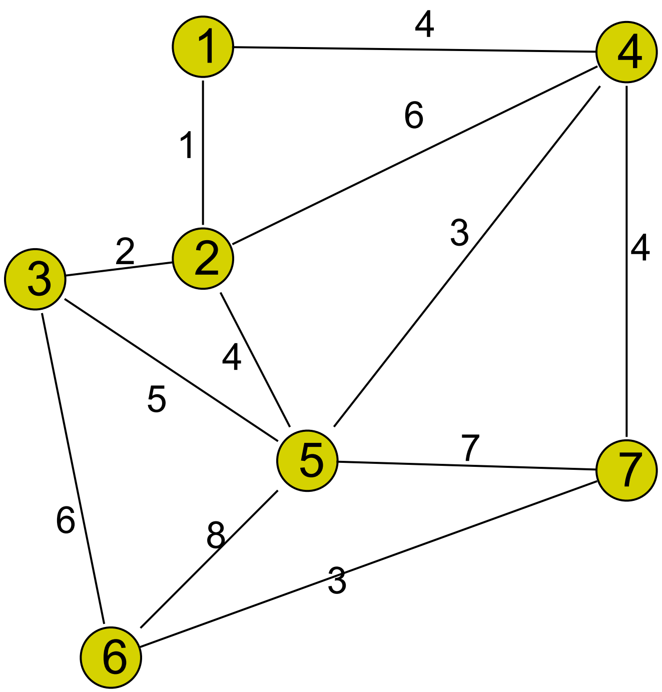
Algoritmos voraces

- La primera solución que encuentra es siempre óptima
- Algoritmo genérico

```
Voraz(Conjunto: c):Conjunto
   1  S={}
   2  ( C no esté vacío y ¬ solución(S) )[  X=seleccionar(C)
                                           C=C-X
                                           Si( factible(S∪X) entonces
                                                 S=S ∪X
                                           fsi   ]
   3  Si ( solucion(S) ) entonces
       regrese S
      sino
       regrese {}
      fsi
```

Grafos etiquetados

- Grafo etiquetado o grafo con peso: Es un grafo que tiene un valor entero o real asignado a cada arista



Grafo etiquetado

Árboles abarcadores mínimos

- Árboles de expansión mínima o de recubrimiento mínimo
- Aplicación:
  - G={N, A} N={ciudades} y A={costo de línea telefónica del nodo a al nodo b} El árbol abarcador mínimo T de G es la red más barata para conectar las ciudades utilizando conexiones directas
  - C={candidatos}=A
  - T={solución}
  - Conjunto de aristas es factible si no contiene ciclos
  - Función de selección varía según el algoritmo
  - Función objetivo: minimizar la longitud total de las aristas en T

Problema del árbol de expansión mínima

- Un grafo etiquetado es un grafo G = (N, A) donde sus aristas tienen asignada alguna información.
- Sea un grafo etiquetado no dirigido y conexo G = (N, A), el árbol de expansión (T) de G es un subconjunto acíclico de A, $T \subset A$, w(T) =   donde $w: A \rightarrow \Re$.
- Encontrar T es el denominado problema del árbol de expansión mínimo.
- Algoritmo general:

  X = { }

  (X no forme un árbol de expansión) [Encontrar una arista (u, v) que sea segura para X

  X = X U (u, v)   ]

  regrese X

Árboles abarcadores mínimos

- Conjunto de aristas factible es prometedor si se puede extender para producir la solución óptima
- Conjunto vacío es siempre prometedor
- Si el conjunto de aristas prometedor ya es una solución, la extensión requerida es irrelevante y esa es la solución óptima
- Lema CP: Sea G={N, A} un grafo conexo etiquetado, $B \subset N$ un subconjunto estricto, $T \subseteq A$ un conjunto prometedor de aristas tal que no haya ninguna arista de T que sale de B, v la arista más corta que salga de B, entonces $T \cup \{v\}$ es prometedor

o una de las más cortas si hay empates

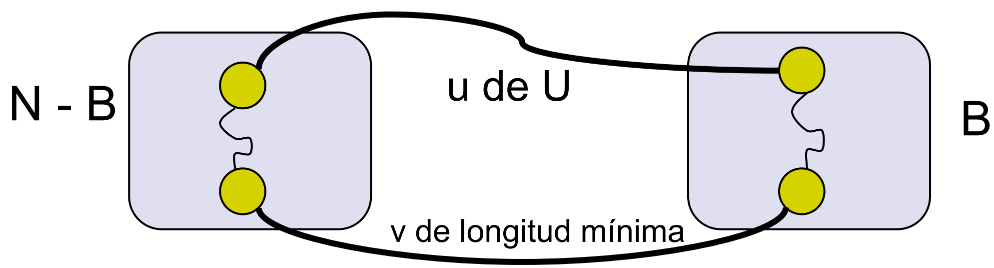
Demostración del lema CP

Sea U un árbol abarcador mínimo de G tal que T$\subseteq$ U, U debe existir, pues T es prometedor por la hipótesis.

Si v $\in$ U, entonces no hay nada que probar.

Si no, cuando se añada v a U se crea un ciclo, donde v sale de B y existe necesariamente al menos otra arista u que también sale de B, o bien el ciclo no se cerraría. Si se elimina u, el ciclo desaparece y se obtiene un nuevo árbol V. Sin embargo, la longitud de v, por definición no es mayor que la longitud de u y por ello, la longitud total de las aristas de V no sobrepasa la longitud total de las aristas de U. Por tanto, V es también un árbol abarcador y contiene a v.

Para completar la demostración, T $\subseteq$ V porque la arista u que se ha eliminado sale de B y por lo tanto no podría haber sido una arista de T.



Conceptos

- $\exists\ T\ /\ X \subseteq T \wedge$ si $\exists\ (u, v) \in T\ /\ (u, v) \notin X \Rightarrow (u, v)$ es segura para X
- **Corte:** Un corte (S, N–S) de un grafo no dirigido G = (N, A) es una partición de N.
- Una arista $(u, v) \in A$ **cruza** el corte (S, N–S) si uno de los nodos terminales de la arista está en S y el otro en N–S.
- Un corte **respeta** A si no hay aristas en A que crucen el corte.
- Una arista es **ligera cruzando el corte** si su peso es el mínimo de cualquier arista cruzando el corte.
- Una arista es ligera satisfaciendo una propiedad dada, si su peso es el mínimo para cualquier arista que satisfaga la propiedad.
- **Teorema:** Sea G =(N, A) un grafo no dirigido conexo etiquetado con una función real para los pesos w definida en A, sea $X \subseteq A$ que está incluido en algún árbol de expansión mínima para G, sea (S, N–S) cualquier corte de G que respete X y sea (u, v) una arista ligera cruzando (S, N–S), entonces la arista (u, v) es segura para X.
- **Corolario:** Sea G =(N, A) un grafo no dirigido conexo etiquetado con w, sea $X \subseteq A$ que está incluido en algún árbol de expansión mínima para G y sea C una componente conexa (árbol) en el bosque Gx = (N, X). Si (u, v) es una arista ligera conectando C a alguna componente de Gx entonces (u, v) es segura para X.

Algoritmo de Kruskal

- Sea C1 y C2 dos árboles que están conectados por (u, v), como (u, v) debe ser una arista ligera conectando C1 a otro árbol, el corolario implica que (u, v) es segura para C1.
- La implantación se basa en la clase ConjDisj (conjuntos disjuntos).

Especificación de la clase ConjDisj

UNIVERSIDAD DE LOS ANDES

2/12/98 Especificación ConjDisj[TipoClave]		
1	**Sintáctica** ConjDisj()→ConjDisj, incluir(ConjDisj,TipoClave) → ConjDisj, union(ConjDisj, ConjDisj) → ConjDisj, buscar(ConjDisj,TipoClave) →Conjunto, vacíoCD(ConjDisj) →Lógico, ConjDisj(ConjDisj)→.	**- ConjDisj():** Crea un conjunto de conjuntos vacío o lo destruye. **-incluir():** Crea un nuevo conjunto con la clave dada. **-union():** Une ambos conjuntos, destruyéndolos y creando el de la unión. **-buscar():** Regresa el conjunto que contiene al elemento. **-vacíoCD():** Regresa verdadero si está vacío.
2	**Declaraciones** TipoClave: c,{TipoNoDef}	
3	**Semántica** vacíoCD(ConjDisj()) = Verdadero vacíoCD(insertar(ConjDisj(), c) = Falso buscar(ConjDisj(), c) = ∅	

Conjuntos disjuntos

Un conjunto disjunto mantiene una colección de conjuntos dinámicos disjuntos, S = $\{S_1, S_2, \ldots, S_k\}$.

Cada conjunto en S contiene un miembro que lo representa y que lo identifica, denominado su representante X.

Operaciones:

crea(X): Crea un nuevo conjunto miembro con un solo elemento que es su representante X, el cual no pertenece a ningún otro conjunto miembro.

union(X, Y): Unifica los dos conjuntos dinámicos que contienen a X e Y, S_X, S_Y en un nuevo conjunto miembro con todos los miembros de S_X y de S_Y. El representante de la unión se escoge entre sus miembros, pero normalmente se selecciona uno de los dos representantes de los conjuntos unidos. Los conjuntos unidos son eliminados del conjunto disjunto, pues no pueden haber repetidos.

busca(X): Regresa una referencia al representante del conjunto que contiene X.

Análisis de algoritmos: se hace según: **n** número de crea(X) realizados y **m** número total de crea(X), union(X, Y) y busca(X) realizados.

Conjuntos disjuntos

Cada union() reduce S en un conjunto, luego de n - 1 uniones S solo contiene un conjunto miembro y $m \geq n$.

Aplicación: Representar un grafo no conexo.

Representación con listas enlazadas

Cada conjunto miembro está implantado como una lista enlazada, donde su primer nodo contiene su representante y el resto de los elementos está en los nodos enlazados.

Cada nodo se enlaza con el siguiente y con su representante.

La relación de orden es de libre escogencia.

Las operaciones crea(X) y busca(X) son simplemente crear una nueva lista con un único nuevo nodo con X y devolver la referencia al primer nodo, respectivamente.

Ambas en O(1).

La implementación de union(X, Y) más simple:

1. Anexar la lista X al final de la lista Y
2. El representante es el representante de Y
3. Actualizar las referencias de todos los elementos de X para que apunten al representante de Y

Conjuntos disjuntos

Sean m operaciones en S, $n = \lceil m/2 \rceil + 1$ y $q = m - n = \lfloor m/2 \rfloor - 1$.

Suponga que se tienen n objetos $x_1, x_2, \ldots, x_n$.

Cuando se ejecutan m = n + q operaciones, se usa $\Theta(n)$ en crea(), $\sum_{i=1}^{q-1} i = \Theta(q^2)$ uniones que actualizan i objetos en S.

El tiempo total es $\Theta(n + q^2)$ lo que implica $\Theta(m^2)$.

El tiempo amortizado de una operación es $\Theta(m)$.

Heurística que mejora el rendimiento:

- Colocar en la cabeza de las listas su número de elementos
- concatenar la lista más pequeña a la lista más grande

El nuevo tiempo amortizado con la mejora es $O(m + n \lg n)$

Conjuntos disjuntos

Representación con un bosque de conjuntos disjuntos

Cada conjunto miembro es un árbol, donde cada nodo referencia a su padre.

La raíz contiene el representante y la referencia a si mismo.

Su rendimiento es el mismo de las listas enlazadas, pero con las heurísticas de *union por rango* y *compresión de camino* se mejora su rendimiento.

- La operación crea() solo crea la raíz de un nuevo árbol con ese único nodo. El constructor se será ConjDisj().
- La operación busca() encuentra la raíz del árbol atravesando el camino de búsqueda desde el nodo hasta su raíz
- La operación union() solo hace que la raíz de un árbol apunte a la raíz del otro

Unión por rango

Union por rango: La raíz del árbol con menos nodos apunta a la raíz del árbol con más nodos.

Rango: valor almacenado en la raíz que se aproxime al logaritmo del tamaño del árbol y a su vez sea una cota superior de la altura del nodo.

Compresión de camino: Cada nodo del árbol apunta a su raíz en vez de apuntar a su padre.

Cada nodo contiene su rango que es un número entero igual al número de enlaces en el camino más largo entre él y una hoja.

Implementación de la clase ConjDisj

Mayo 2000		
ConjDisj()		
{**pre:** }		{**pos:** raiz = Nulo }
1	raiz = Nulo	
2	regrese	

Mayo 2000		
ConjDisj(TipoClave: x)		
{**pre:** }		{**pos:**nx.rango=0,nx.dato=x,nx.p=este }
1	Nodo nx(x)	**Nodo.** Constructor.
2	raiz = direccionDe nx	**-direccionDe.** Regresa la dirección de memoria de la variable.
3	regrese	

Mayo 2000		
buscaEle(TipoEle: cl): Lógico		
{**pre:** raiz≠Nulo }		{**pos:** }
1	pn = busca(raiz)	**-Clave().** Definido en Nodo.
2	regrese (pn→Clave() = cl)	**-pn.** ApuntadorA Nodo. Var. aux.

Mayo 2000		
busca(ApuntadorA Nodo: px): ApuntadorA Nodo		
{**pre:** px≠Nulo }		{**pos:**px→P()≠Nulo }
1	**Si** (px≠px→P()) **entonces** .. px→P(busca(px→P())) **fsi**	**-P().** Definido en Nodo. **-busca().** Definido en ConjDisj.
2	regrese (px→P())	

Implementación de la clase ConjDisj

Mayo 2000		
union(ConjDisj: cd2)		
{**pre:** }		{**pos:** raiz≠Nulo }
1	enlace(este→busca(raiz),cd2.busca(cd2.raiz))	**-raiz,** **busca().** Definido en ConjDisj.
2	regrese	

Mayo 2000		
enlace(ApuntadorA Nodo: px, py)		
{**pre:** px, py≠Nulo }		{**pos:** }
1	Si (px→Rango()>py→Rango()) entonces .. py→P(px) sino .. px→P(py) .. Si (px→Rango()=py→Rango()) entonces .... py→Rango(py→Rango()+1) .. fsi fsi	**-P(),** **Rango().** Definidos en Nodo.
2	regrese	

Análisis:

Solo con union por rango O(m lg n). Con n operaciones crea(), al menos n - 1 operaciones union() y f operaciones busca() se obtiene $W(n) = \Theta(f \log_{1+f/n} n)$ si $f \geq n$ y $\Theta(n + f \lg n)$ si f<n.

Con ambas heurísticas W(n) = O(m α(m,n)) donde α(m, n) es la función inversa de Ackermann que tiene un crecimiento muy lento. Una cota débil sobre ella es O(m lg* n).

Análisis de la clase ConjDisj

- La mejor implantación que se conoce para los conjuntos disjuntos es la que está basada en árboles con el uso de las heurísticas de unión por rango y compresión de los caminos. El análisis de tal implantación es de **O(m lg* n)**, donde **m** es el número de veces que se han invocado las operaciones del TAD y **n** es el número de conjuntos en el TAD.
- Cada conjunto contiene los nodos del bosque actual y la operación buscar se modifica para que regrese el elemento (clave) representativo del conjunto que contiene al solicitado.

Algoritmo de Kruskal

- T está vacío al inicio
- T va creciendo a medida que el algoritmo avanza.
- Mientras no haya encontrado la solución, el grafo parcial formado por los nodos de G y las aristas de T consta de varios componentes conexos
- Los elementos de T forman un árbol abarcador mínimo para los nodos del componente conexo
- Al final, solo queda un componente conexo, entonces T es el árbol abarcador mínimo de G

Algoritmo de Kruskal

26/11/98

KruskalAEM( ): Conjunto[Arista]

{pre: $n > 0$ }   {pos: $n > 0 \wedge G' = G \wedge X$ es el árbol de expansión mínima de G }

1	[cdis.incluir(v)] $v \in N$	**-X.** Conjunto[X]. Árbol de expansión mínima resultante para el grafo.
2	Ordene ascendente las aristas de G por sus pesos w	**-cdis:** ConjDisj[TipoClave]. Bosque de nodos del grafo.

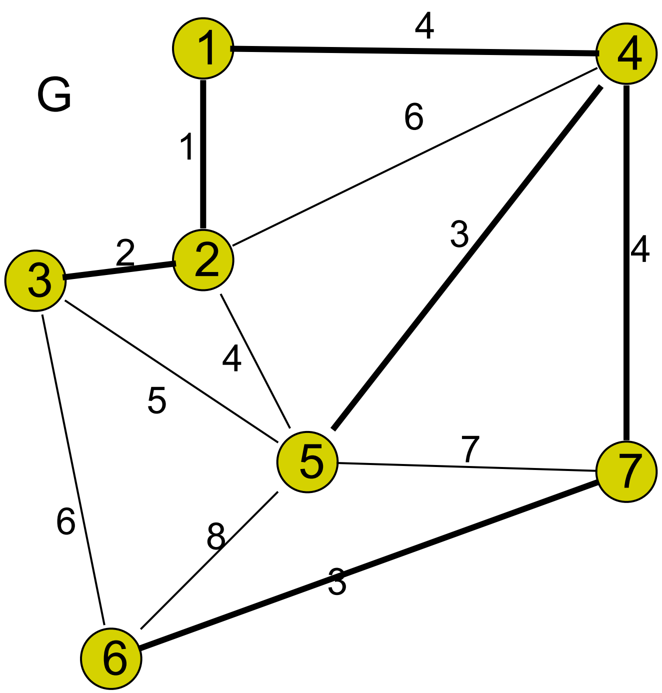
3	[Si (cdis.buscar(v) $\neq$ cdis.buscar(u)) entonces
	    $X = X \cup \{(u, v)\}$
	    cdis.union(u, v) // O(A lg A)
	  fsi] $(u, v) \in A$ por orden ascendente de w
		-incluir( ), buscar( ), union ( ). Definidas en la clase ConjDisj.

4	regrese X	**-** $\cup$**( ).** Función de la clase Conjunto [X].

T(n) = O(A lg A)

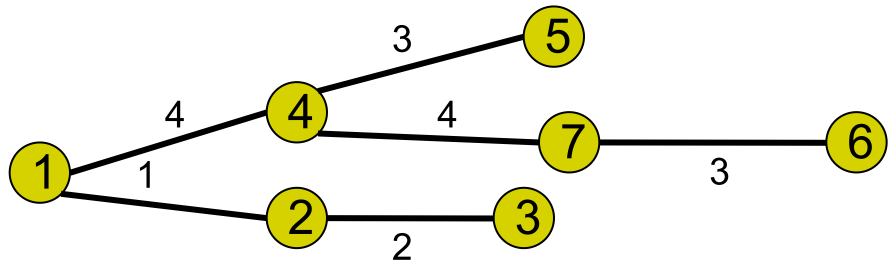
Algoritmo de Kruskal

G



Paso	arista considerada	componentes conexos
Inicio	-	{1}{2}{3}{4}{5}{6}{7}
1	(1,2)	{1,2}{3}{4}{5}{6}{7}
2	(2,3)	{1,2,3}{4}{5}{6}{7}
3	(4,5)	{1,2,3}{4,5}{6}{7}
4	(6,7)	{1,2,3}{4,5}{6,7}
5	(1,4)	{1,2,3,4,5}{6,7}
6	(2,5)	rechazado
7	(4,7)	{1,2,3,4,5,6,7}

T



Teorema

- El algoritmo de Kruskal halla un árbol abarcador mínimo

Demostración: por inducción sobre el número de aristas en T.

Se muestra que si T es prometedor entonces sigue siendo prometedor en cualquier paso del algoritmo cuando se le añade una arista adicional.

Base: T vacío es prometedor porque G es conexo

Inductiva: suponga que T es prometedor inmediatamente antes que el algoritmo añada una nueva arista h=(u, v). Las aristas de T dividen a los nodos de G en dos o más componentes conexos, el nodo u se encuentra en uno y v en otro distinto. Sea B el conjunto de nodos que contiene a u. Entonces:

- B es un subconjunto estricto de G, pues no incluye a v
- T es un conjunto prometedor tal que ninguna arista de T sale de B
- h es una de las aristas más cortas que salen de B

Se cumplen las condiciones del lema CP y se concluye que $T \cup \{h\}$ es también prometedor

T es prometedor en todas las fases del algoritmo, T es la solución óptima.

Colas por prioridad

- Características:
  - Su nombre viene de una aplicación de Sistemas Operativos: el mantenimiento de las colas internas de procesos, donde esos procesos son manejados según su prioridad asignada.
  - Es un conjunto de entradas
  - Cada entrada en la cola es un par [clave, valor]
  - Clave: es un campo especial para reconocer la entrada y puede tener un valor repetido en el conjunto de entradas (clave secundaria)
  - Las claves están siempre ordenadas obedeciendo a un orden total
  - Los valores asociados a las claves se pueden actualizar, pero no las claves
  - Normalmente se implementan con montículos

Clase ColasxPrioridad

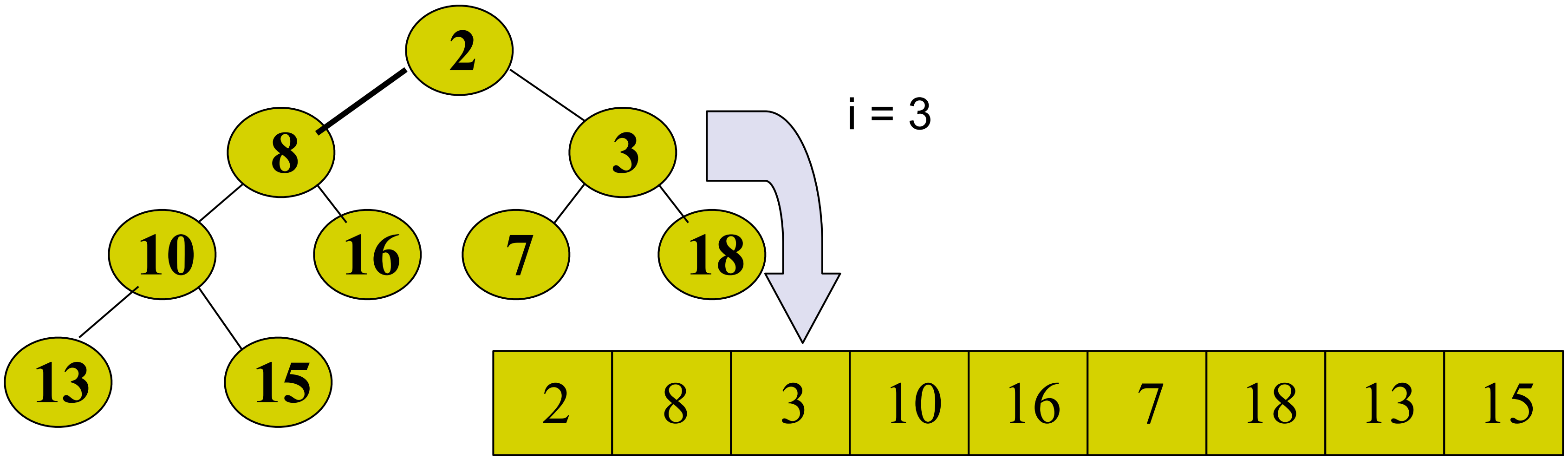
- Las operaciones más importantes en un TDA de colas por prioridad se refieren aquellas que permiten repetidamente seleccionar el elemento de la cola de prioridad que tiene como clave el valor mínimo (máximo).
- Esto conlleva a que una cola por prioridad P debe soportar las siguiente operaciones:
  - inserta(ent) crea()
  - min() union()
  - extMin() destruye()

Montículos binarios

- Implementaciones de un TDA de Colad de Prioridad
  - Arboles equilibrados (AVL, ROJO y NEGRO)
  - Montículos Binarios
  - Montículos a la izquierda
  - Montículos oblicuos
  - Colas binomiales, colas binomiales perezosas
  - Colas de Fibonacci
- Un montículo binario (o simplemente montículo) es un árbol binario semicompleto en el que el valor de la clave almacenada en cualquier nodo es menor o igual que los valores claves de sus hijos
- Propiedad de ordenamiento parcial: la clave almacenada en cualquier nodo es menor o igual que los valores claves de sus hijos.

Montículos binario

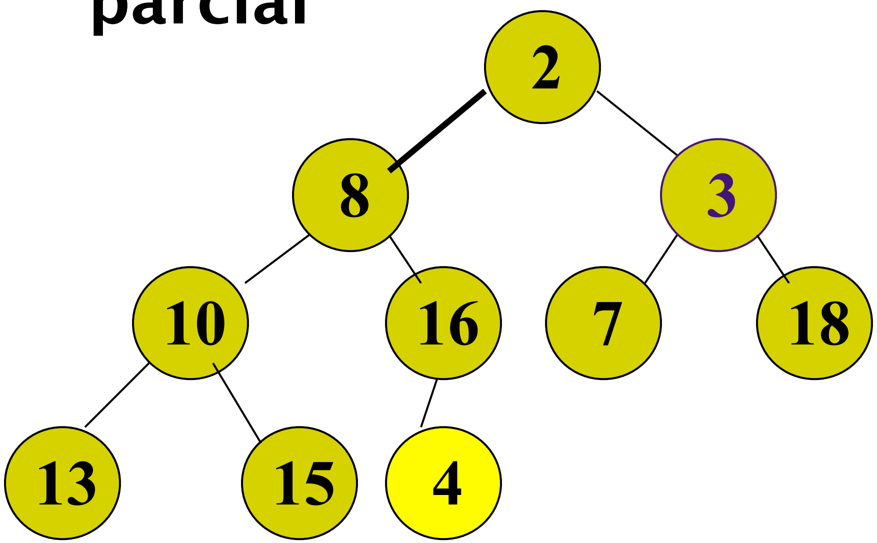
- Ventajas: el hecho de ser semicompleta hace que sea posible una representación secuencial
- **Si un nodo esta almacenado en la posición i.**
  - **Su hijo izquierdo si existe, se encuentra en la posición 2i.**
  - **Su hijo derecho si existe, se encuentra en la posición 2i+1.**
  - **El padre en la posición i/2**



Montículos binarios

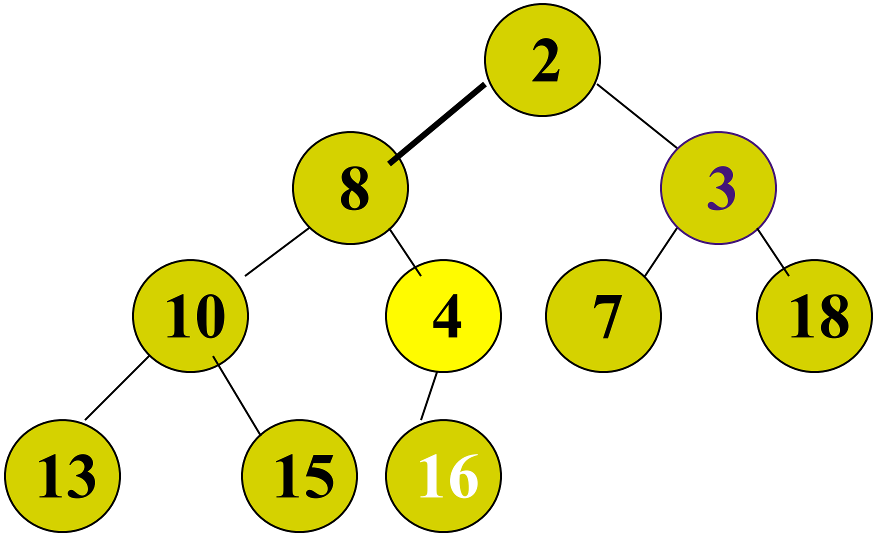
Inserta(ent):

- **El nodo se añade como una hoja extrema creciendo de izquierda a derecha (n+1). (garaztiza la propiedad de forma)**
- **Se garantiza la propiedad de ordenamiento parcial**



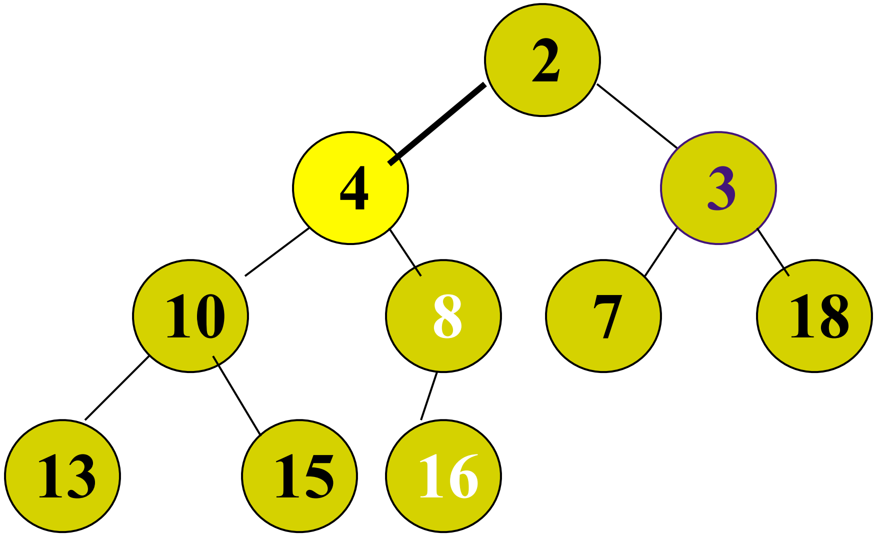
2	8	3	10	16	7	18	13	15	4

Montículos binarios



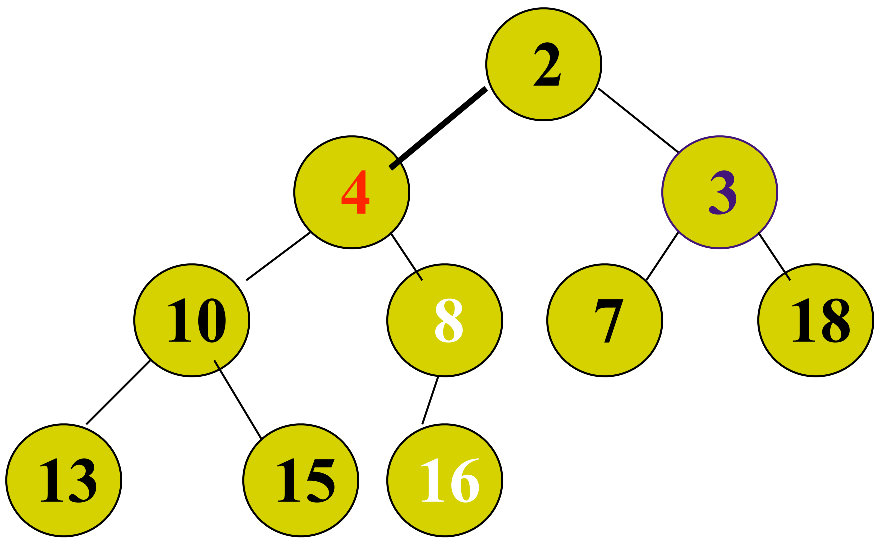
2	8	3	10	4	7	18	13	15	16

Montículos binarios



2	4	3	10	8	7	18	13	15	16

Montículos binarios



2	4	3	10	8	7	18	13	15	16

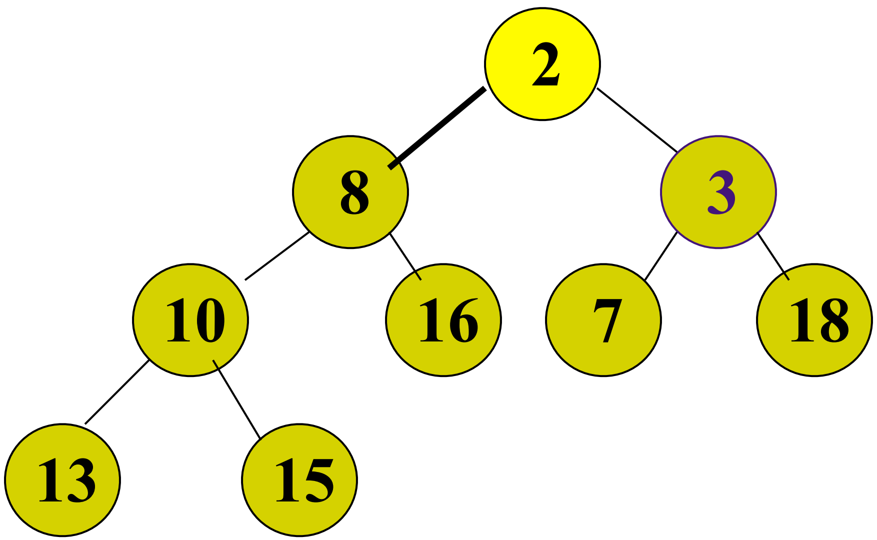
Montículos binarios

2	8	3	10	16	7	18	13	15	4	I=10
2	8	3	10	4	7	18	13	15	16	I=5
2	4	3	10	8	7	18	13	15	16	I=2
2	4	3	10	8	7	18	13	15	16	I=1

Montículos binarios

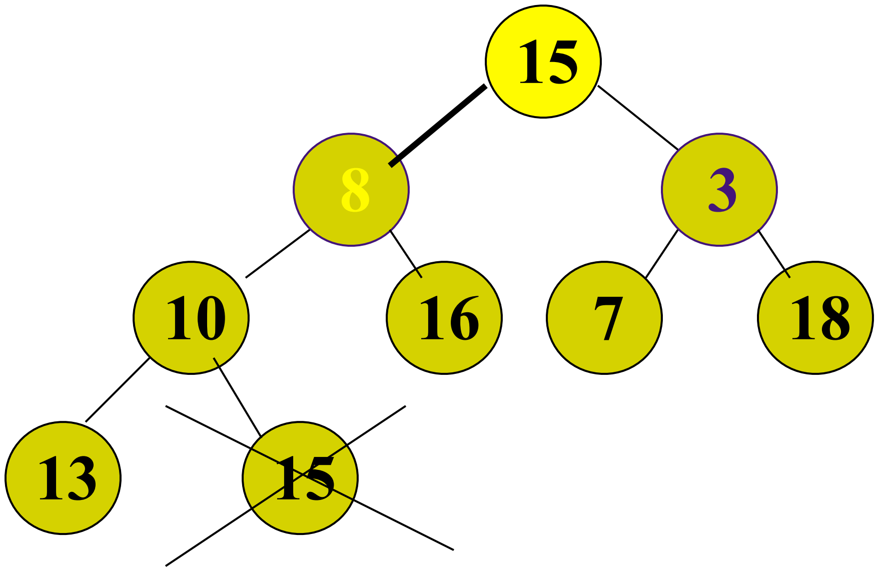
extMin(): se elimina el elemento de clave mínima, es decir el elemeto de la raíz.

Se debe garantizar la propiedad de ordenamiento parcial



2	8	3	10	16	7	18	13	15

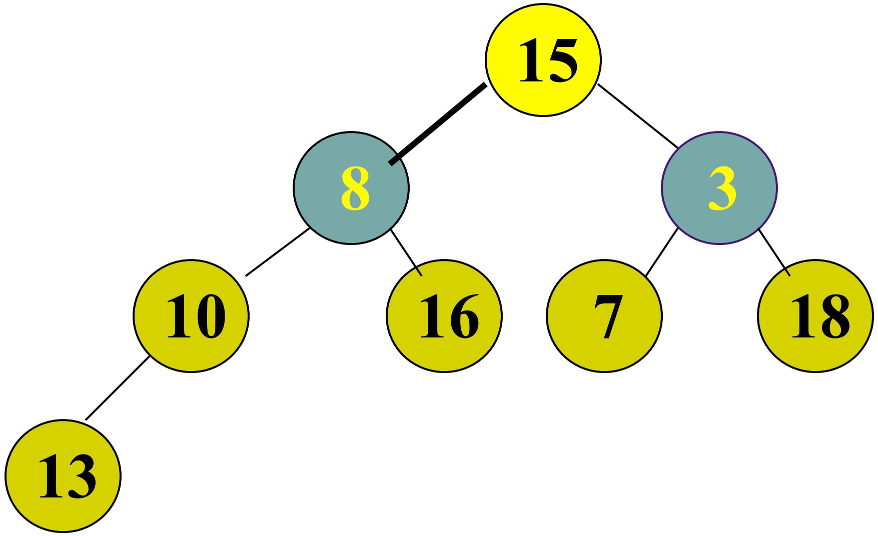
Montículos binarios



15	8	3	10	16	7	18	13

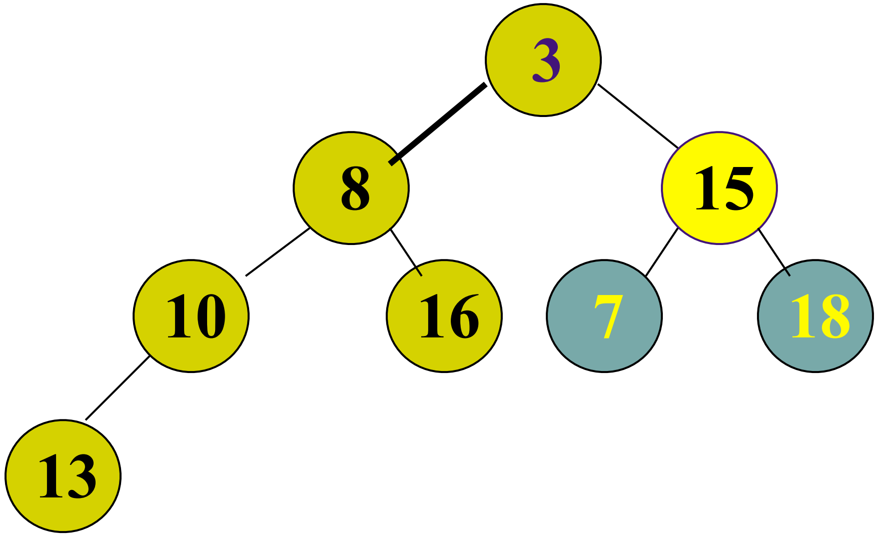
n=n-1

Montículos binarios



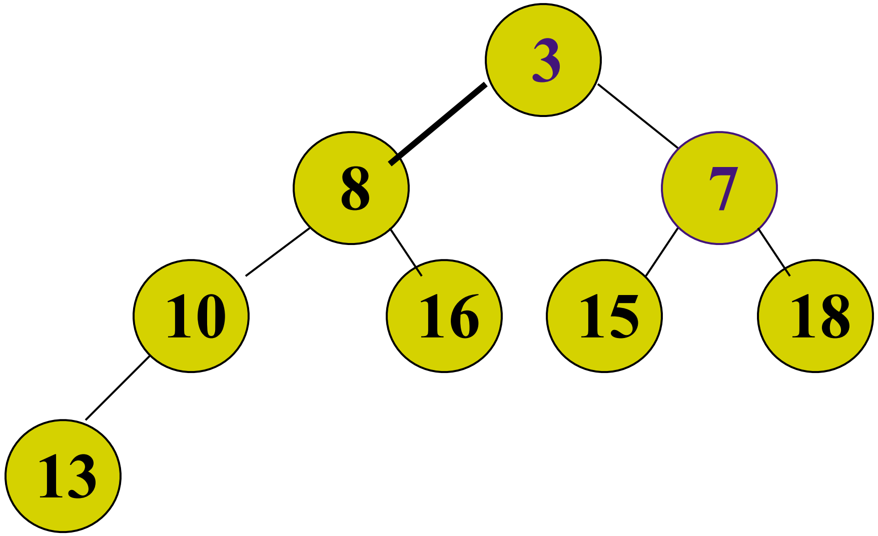
15	8	3	10	16	7	18	13

Montículos binarios



3	8	15	10	16	7	18	13

Montículos binarios



3	8	7	10	16	15	18	13

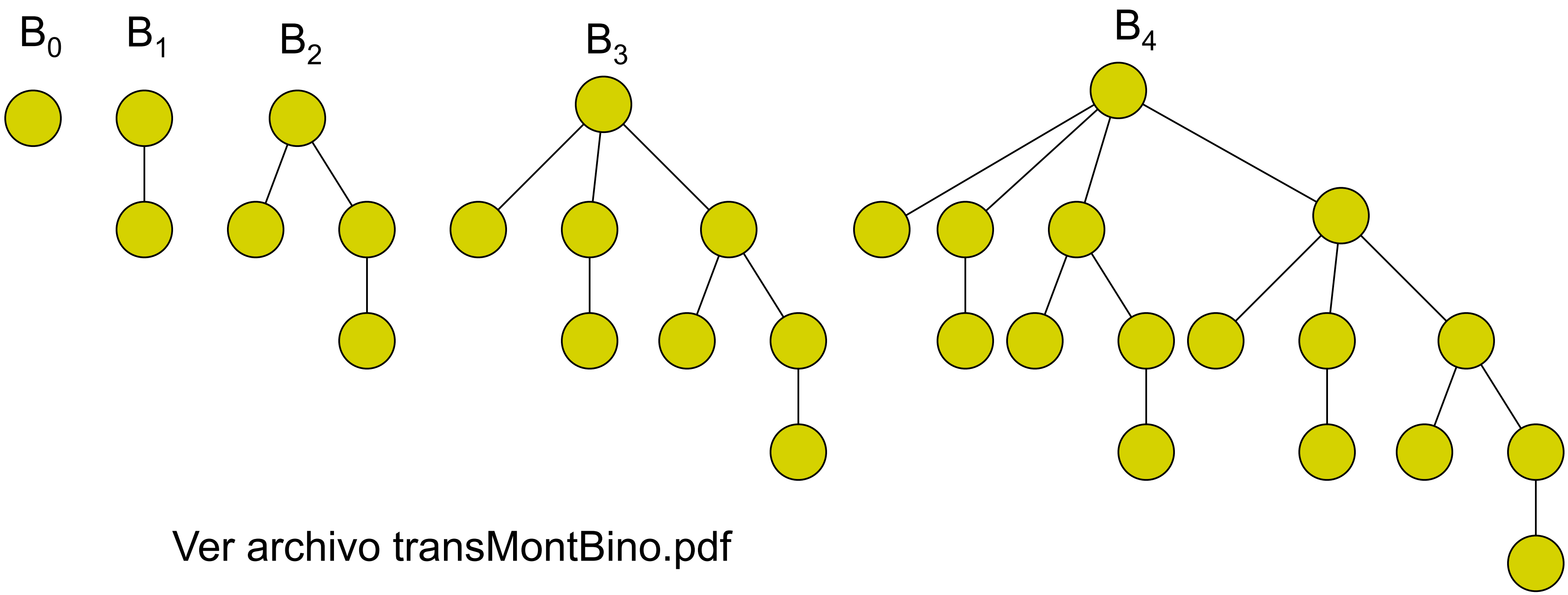
Montículos binarios

- En la operación de inserción es necesario realizar un **flotar** (filtrado ascendente) del nodo insertador para asegurar la propiedad de forma.
- En la operación de eliminación es necesario realizar un **hundir** (filtrado descendente) del nodo insertador para asegurar la propiedad de forma.

Operación	Montículo binario W(n)	Montículo binomial W(n)	Montículo Fibonacci Amort.
crear()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
inserta()	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
min()	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
extMin()	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
union()	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
decreClave()	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
elimina()	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Árboles binomiales

- El i-ésimo árbol binomial Bi, con i ≥ 0, es aquel que consta de un nodo raíz con i hijos, donde el j-ésimo hijo con 1 ≤ j ≤ i, es a su vez un árbol binomial Bj-1.



Ver archivo transMontBino.pdf

Montículos de Fibonacci

Fredman y Tarjan, 1987

Ver archivo transMontFibo.pdf

Es un montículo mezclable conformado por una colección de árboles, pero sin relación de orden entre ellos.

Se recomiendan cuando el número de extMin() y elimina() es menor en relación al número de ocurrencias de las otras operaciones.

Diferencia con los binomiales: los de fibonacci tienen una estructura menos rígida cuyo mantenimiento se retrasa hasta que sea conveniente realizarlo, permitiendo una menor complejidad en tiempo.

NodoMont: | p clave grado marca hijo izq der |

p: Apuntador al nodo padre

clave: Clave del nodo

grado: Número de hijos del nodo

marca: Es verdadero si el nodo ha perdido un hijo durante el periodo en que él fue hecho hijo de otro nodo. Los nodos recién creados tienen marca Falso.

hijo: Apuntador a un nodo hijo

izq: Apuntador al nodo hermano a la izquierda

der: Apuntador al nodo hermano a la derecha

Algoritmo de Prim

- Es un algoritmo incremental de tipo II.
- El árbol formado es un árbol simple, que se comienza a formar con un nodo arbitrario y crece hasta tener todos los nodos en X.
- Todos las aristas sumadas a X son seguras para X.
- El árbol se aumenta en cada etapa con una arista que contribuye con la mínima cantidad posible a los pesos del árbol.
- Se implementa con una cola por prioridad para seleccionar fácilmente la nueva arista a ser incluida en X.

Algoritmo de Prim

- Características:
  - El árbol abarcador mínimo crece en forma natural, desde un nodo seleccionado como raíz
  - En cada fase se añade una arista hasta que se alcanzan todos los nodos
  - Sea B el conjunto de nodos y T el conjunto de aristas
  - Inicialmente B tiene un único nodo arbitrario y T está vacío
  - En cada paso, busca la arista más corta posible (u, v) tal que $u \in B$ y $v \in N$, se añade v a B y (u, v) a T
  - Se continúa mientras $B \neq N$, las aristas en T siempre forman un árbol abarcador mínimo

Algoritmo de Prim

- Algoritmo genérico

Prim(Grafo: g):Conjunto

1 T={}, B = {un nodo cualquiera de g}

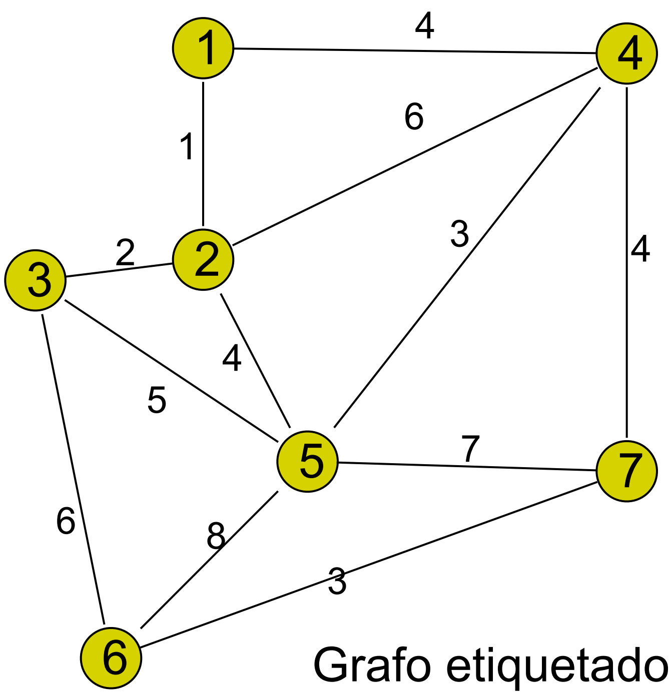
2 (B $\neq$ N)[buscar e = (u, v) de longitud mínima / u$\in$B y v$\in$N

T = T $\cup$ {e}

B = B $\cup$ {v}]

3 regrese T

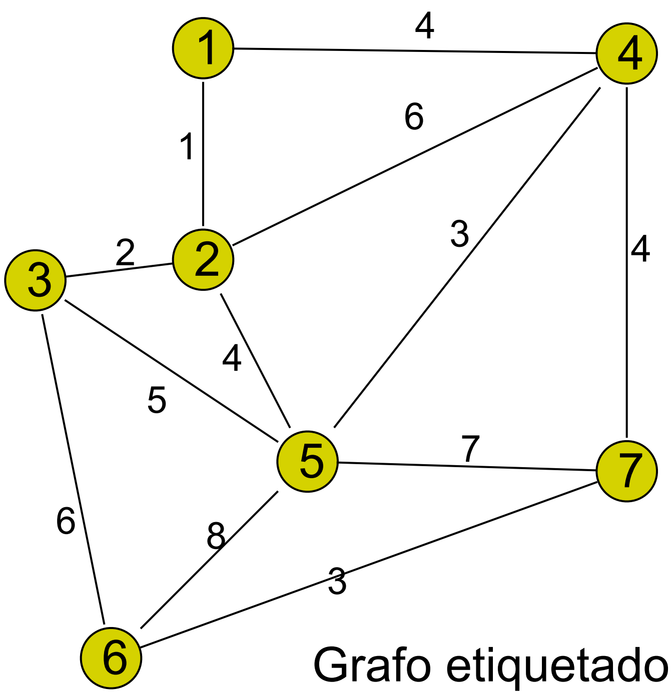
Algoritmo de Prim



Grafo etiquetado

Algoritmo de Prim

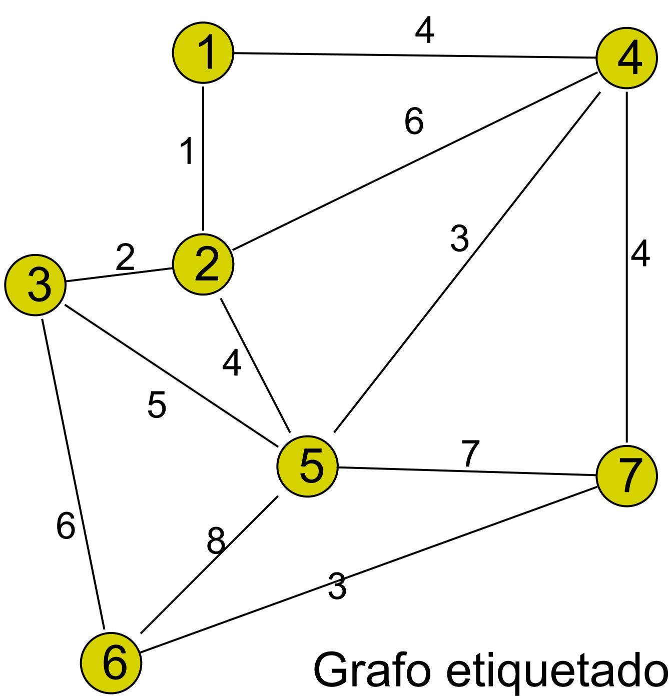
Paso (u, v) B



Grafo etiquetado

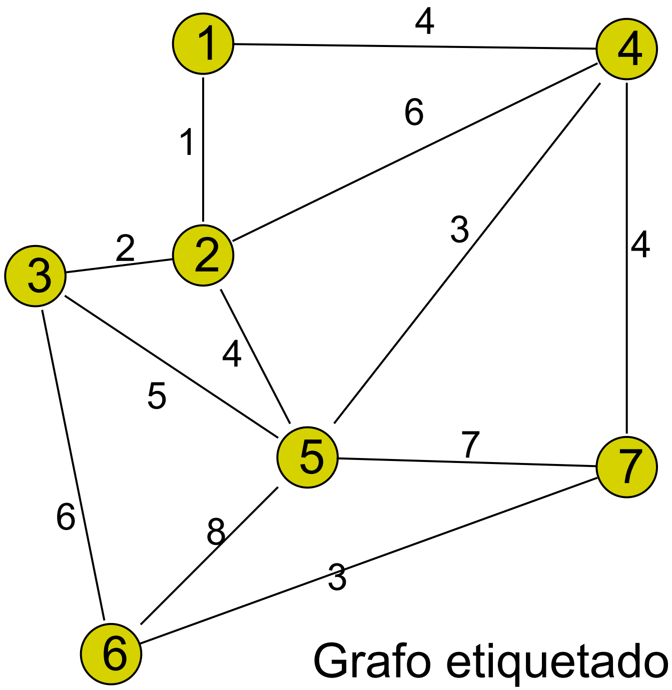
Algoritmo de Prim

Paso	(u, v)	B
Inicio	----	{1}



Grafo etiquetado

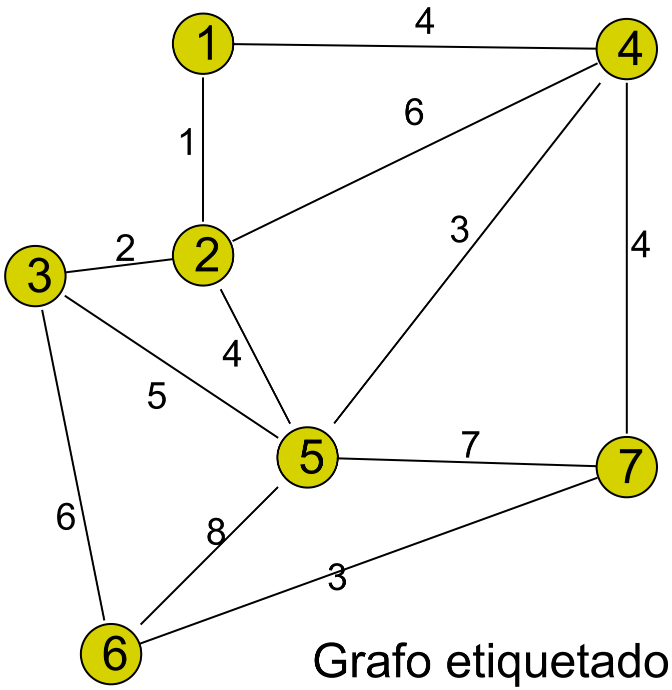
Algoritmo de Prim



Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}

Algoritmo de Prim

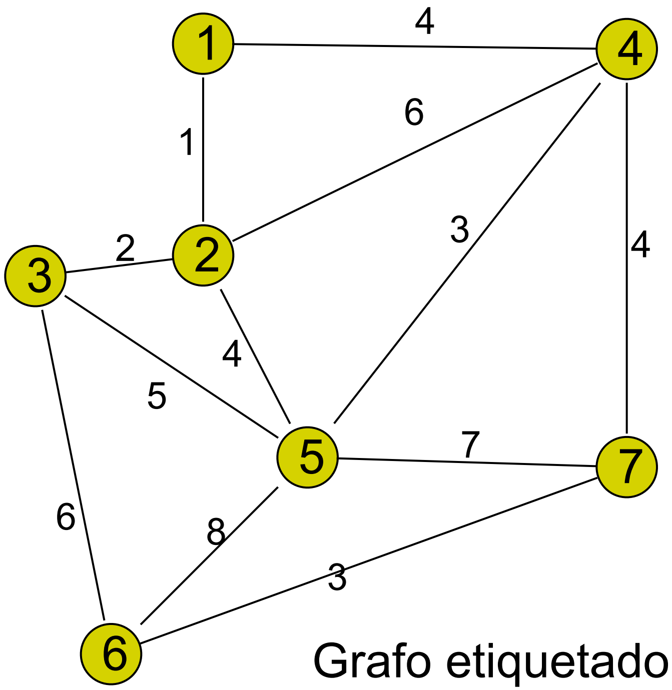


Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}

martes 12 de julio de 2011

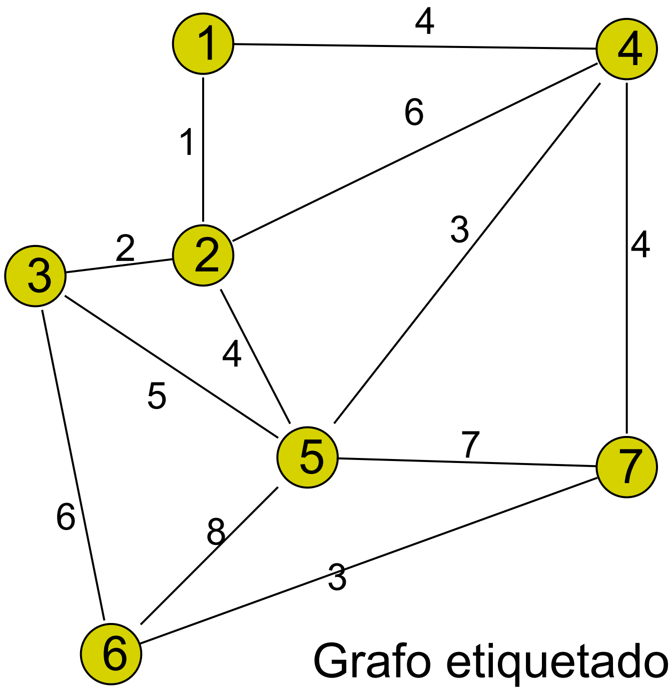
Algoritmo de Prim



Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}

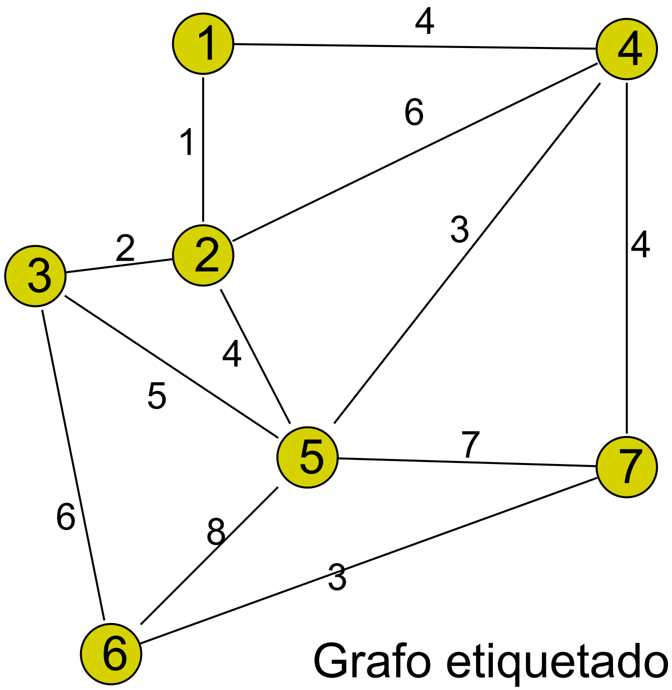
Algoritmo de Prim



Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}
4	(4, 5)	{1, 2, 3, 4, 5}

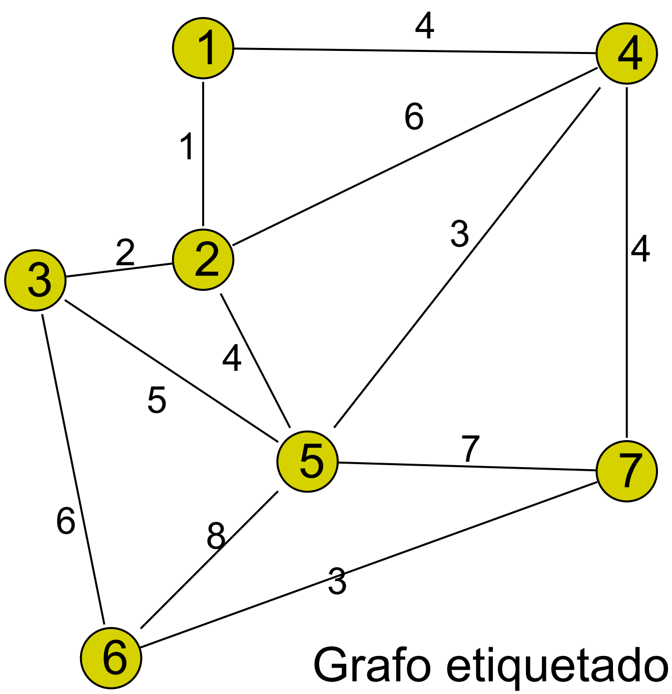
Algoritmo de Prim



Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}
4	(4, 5)	{1, 2, 3, 4, 5}
5	(4, 7)	{1, 2, 3, 4, 5, 7}

Algoritmo de Prim

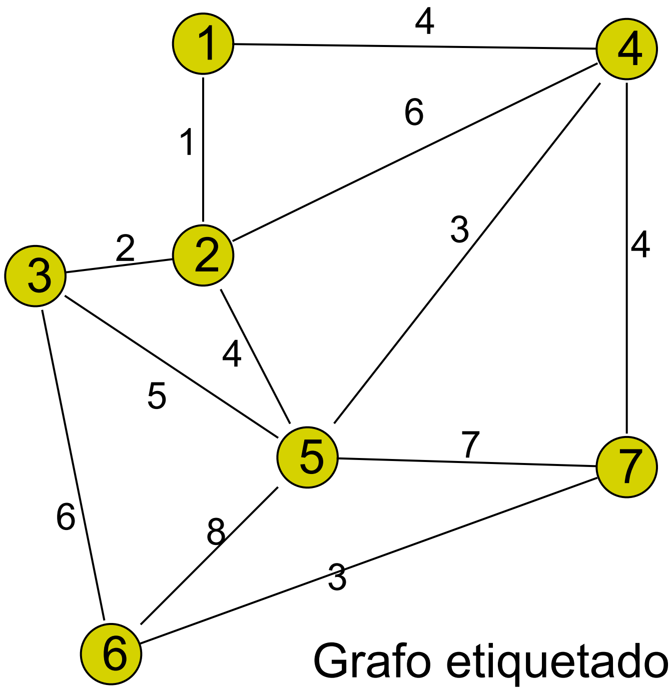


Grafo etiquetado

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}
4	(4, 5)	{1, 2, 3, 4, 5}
5	(4, 7)	{1, 2, 3, 4, 5, 7}
6	(7, 6)	{1, 2, 3, 4, 5, 7, 6}

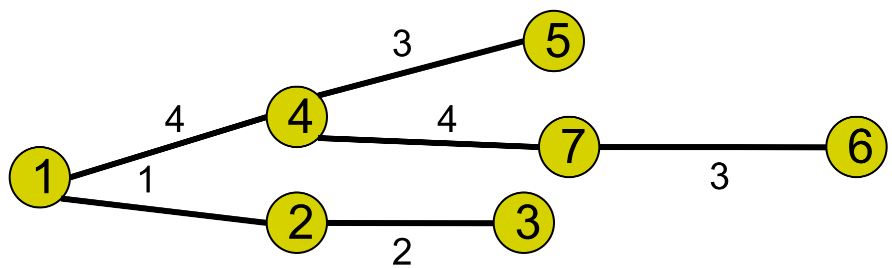
Algoritmo de Prim

Paso	(u, v)	B
Inicio	----	{1}
1	(1, 2)	{1, 2}
2	(2, 3)	{1, 2, 3}
3	(1, 4)	{1, 2, 3, 4}
4	(4, 5)	{1, 2, 3, 4, 5}
5	(4, 7)	{1, 2, 3, 4, 5, 7}
6	(7, 6)	{1, 2, 3, 4, 5, 7, 6}



Grafo etiquetado

T



Teorema

- El algoritmo de Prim halla un árbol abarcador mínimo

Demostración: por inducción sobre el número de aristas en T. Se muestra que si T es prometedor entonces sigue siendo prometedor en cualquier paso del algoritmo cuando se le añade una arista adicional.

Base: T vacío es prometedor

Inductiva: suponga que T es prometedor inmediatamente antes que el algoritmo añada una nueva arista e=(u, v). Entonces:

- B es un subconjunto estricto de N, pues no incluye a v
- T es un conjunto prometedor
- e es una de las aristas más cortas que salen de B

Se cumplen las condiciones del lema CP y se concluye que $T \cup \{e\}$ es también prometedor

T es prometedor en todas las fases del algoritmo, T es una solución óptima del problema.

Algoritmo de Prim

26/11/98

PrimAEM(Nodo: r, Arreglo[n]De Nodo: &clave): Arreglo[n]De Nodo

{pre: $n > 0 \wedge r \in N$ } {pos: $n > 0 \wedge G' = G \wedge$ }

1 [c.entrar(v, MV)] $v \in N$
2 clave(r), padre(r) = 0, Nulo
3 (¬c.vaciaCola( ))[u = c.min( )
  [Si ($v \in c \wedge w(u, v) <$ clave(v)) ent.
    padre(v) = u
    clave(v) = w(u, v)
  fsi] $\forall\, v \in$ u.listaAdyacencia
4 regrese padre

-c. ColaPrioridad[X]. Cola de nodos.
-clave: Arreglo[n]De Nodo. Variable auxiliar con los pesos.
-entrar(), vaciaCola(), min(). Definidas en la clase ColaPrioridad.
-padre. Arreglo[n]De Nodo. Variable auxiliar con el árbol de expansión mínima.

$$T(n) = O(A + N \lg N)$$

Análisis del algoritmo de Prim

- La clase ColaPrioridad debe ser implantada con montículos de Fibonacci para poder tener el tiempo asintótico dado.
- Si se implanta con un montículo binario su complejidad es igual a la del algoritmo de Kruskal.