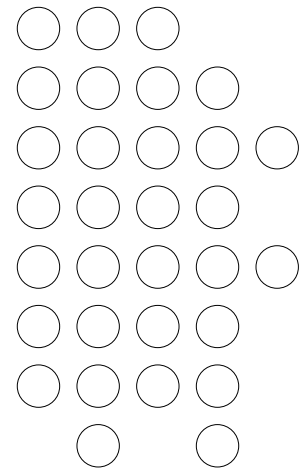


Grafos: Árboles abarcadores mínimos

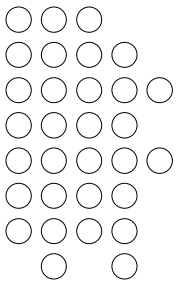


UNIVERSIDAD
DE LOS ANDES

Diseño y Análisis de Algoritmos
Cátedra de Programación
Carrera de Ingeniería de Sistemas
Prof. Isabel Besembel Carrera



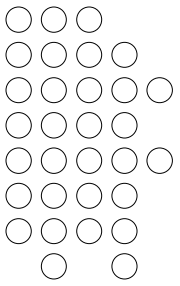
Algoritmos voraces



➤ Características:

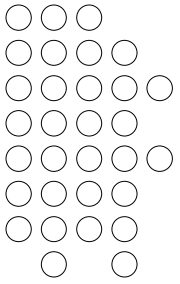
- ❖ Para construir la solución del problema se tiene un conjunto de candidatos
- ❖ Se van acumulando dos conjuntos: el de los considerados y seleccionados y el de los considerados y rechazados
- ❖ Función de comprobación: si el conjunto de candidatos es una solución óptima o no
- ❖ Función de factibilidad: 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 \neg solución(S)) [

$X = \text{seleccionar}(C)$

$C = C - X$

Si(factible($S \cup X$) entonces

$S = S \cup X$

fsi]

3 Si (solución(S)) entonces

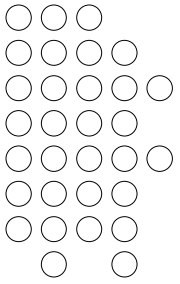
regrese S

sino

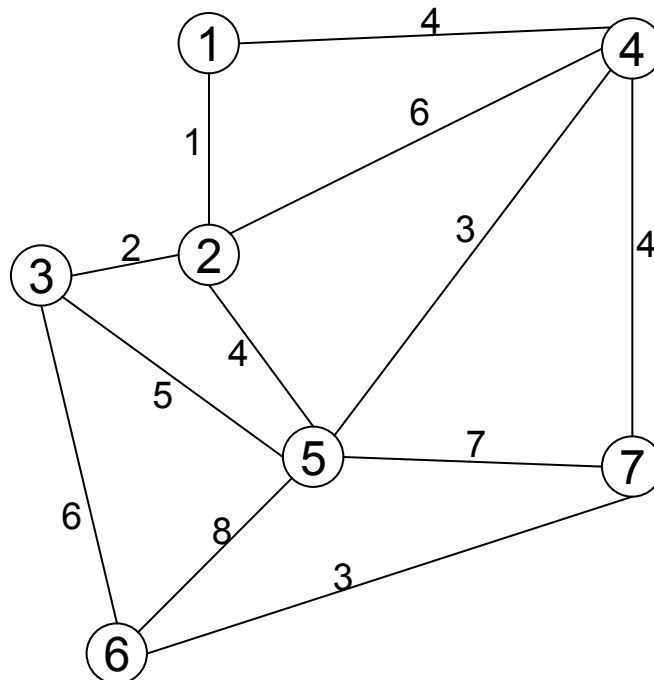
regrese $\{\}$

fsi

Grafos etiquetados

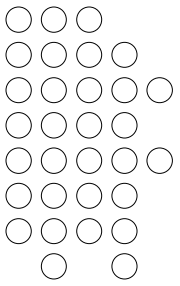


- Grafo etiquetado o grafo con peso: Es un grafo que tiene un valor entero o real asignado a cada arista



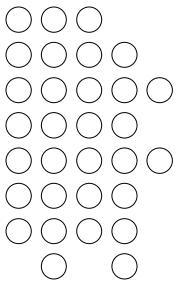
Grafo etiquetado

Árboles abarcadores mínimos



- Árboles de expansión mínima o de recubrimiento mínimo
- Aplicación:
 - ❖ $G = \{N, A\}$ $N = \{\text{ciudades}\}$ y $A = \{\text{costo de línea telefónica del nodo a al nodo b}\}$ El árbol abarcador mínimo T de G es la red más barata para conectar las ciudades utilizando conexiones directas
 - ❖ $C = \{\text{candidatos}\} = A$
 - ❖ $T = \{\text{solución}\}$
 - ❖ 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

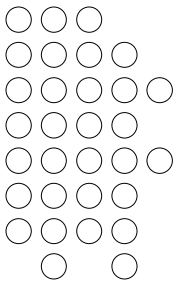
Problema del árbol de expansión mínima



- 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 , $T \subset A$, $w(T) = \sum_{e \in T} w(e)$ donde $w: A \rightarrow \mathcal{R}$.
- Encontrar T es el denominado *problema del árbol de expansión mínimo*.
- Algoritmo general:

$X = \{ \}$	[Encontrar una arista (u, v) que
(X no forme un árbol de expansión)		sea segura para X
$X = X \cup (u, v)$]	
regrese X		

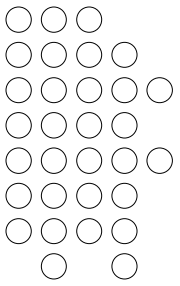
Árboles abarcadores mínimos



- 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

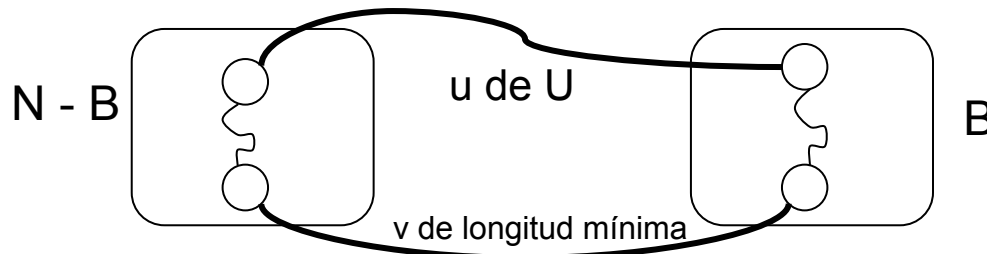


Sea U un árbol abarcador mínimo de G tal que $T \subseteq U$, U debe existir, pues T es prometedor por la hipótesis.

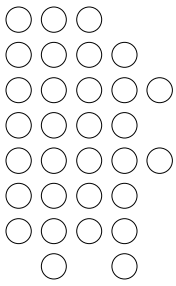
Si $v \in U$, entonces no hay nada que probar.

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 .

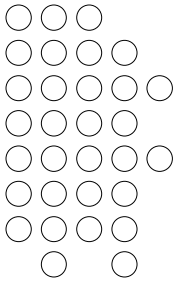


Conceptos



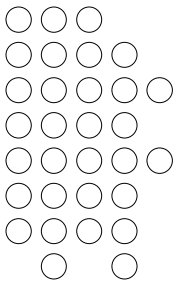
- $\exists T / X \subseteq T \wedge \text{si } \exists (u, v) \in T / (u, v) \notin X \Rightarrow (u, v)$ es segura para X
- **Corte:** Un corte $(S, N-S)$ de un grafo no dirigido $G = (N, A)$ es una partición de N .
- Una arista $(u, v) \in A$ **crusa** el corte $(S, N-S)$ si uno de los nodos terminales de la arista está en S y el otro en $N-S$.
- Un corte **respeto** 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 $G_x = (N, X)$. Si (u, v) es una arista ligera conectando C a alguna componente de G_x 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

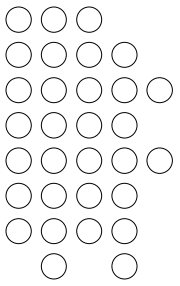


2/12/98

Especificación ConjDisj[TipoClave]

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

Conjuntos disjuntos



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

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

Operaciones:

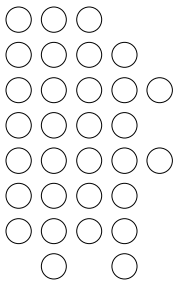
crea(X): Crea un nuevo conjunto miembro con un solo elemento que es su representante X , el cual no pertenece a ningún otro conjunto miembro.

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 $\text{union}()$ 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.

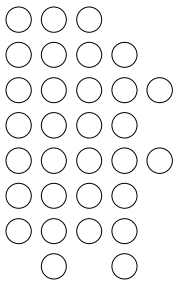
Las operaciones $\text{crea}(X)$ y $\text{busca}(X)$ son simplemente crear una nueva lista con un único nuevo nodo con X y devolver la referencia al primer nodo, respectivamente.

Ambas en $O(1)$.

La implementación de $\text{union}(X, Y)$ 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, \dots, x_n .

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

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

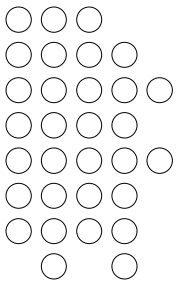
El tiempo amortizado de una operación es $\Theta(m)$.

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

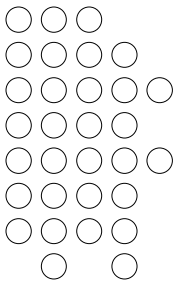
Cada conjunto miembro es un árbol, donde cada nodo referencia a su padre.

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



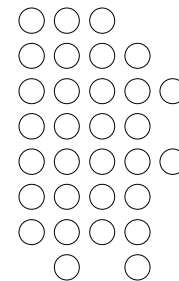
Unión por rango: La raíz del árbol con menos nodos apunta a la raíz del árbol con más nodos.

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



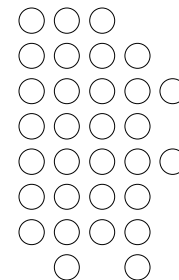
Mayo 2000		ConjDisj()
{pre: }		{pos: raiz = Nulo }
1	raiz = Nulo	
2	regrese	

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
3	regrese	la dirección de memoria de la variable.

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())) fisi	-P() . Definido en Nodo.
2	regrese (px→P())	-busca() . Definido en ConjDisj.

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		

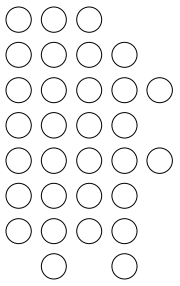
Mayo 2000		enlace(ApuntadorA Nodo: px, py)	
{pre: px, py≠Nulo }		{pos: }	
1	Si (px→Rango()>py→Rango()) entonces .. py→P(px) sino .. px→P(py) .. Si (px→Rango()==py→Rango()) entonces ... py→Rango(py→Rango()+1) .. fsi fsi	-P(), Rango(). Definidos en Nodo.	
2	regrese		

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 $f < n$.

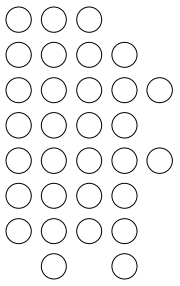
Con ambas heurísticas $W(n) = O(m \alpha(m, n))$ donde $\alpha(m, n)$ es la función inversa de Ackermann que tiene un crecimiento muy lento. Una cota débil sobre ella es $O(m \lg^* n)$.

Análisis de la clase ConjDisj

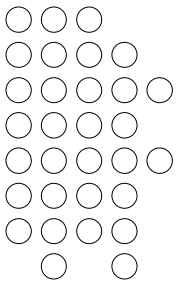


- La mejor implantación que se conoce para los conjuntos disjuntos es la que está basada en árboles con el uso de las heurísticas de unión por rango y compresión de los caminos. El análisis de tal implantación es de $O(m \lg^* n)$, donde m es el número de veces que se han invocado las operaciones del 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

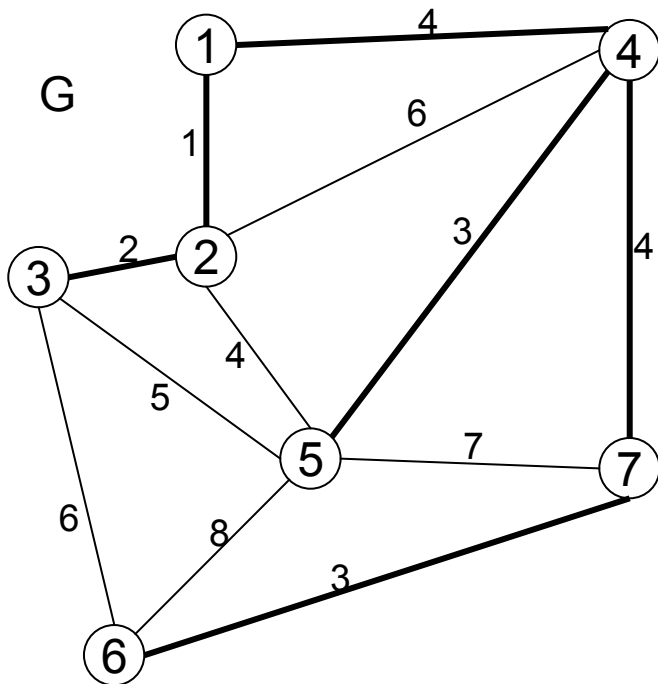
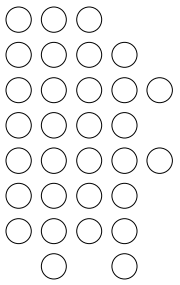
KruskalAEM(): Conjunto[Arista]

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

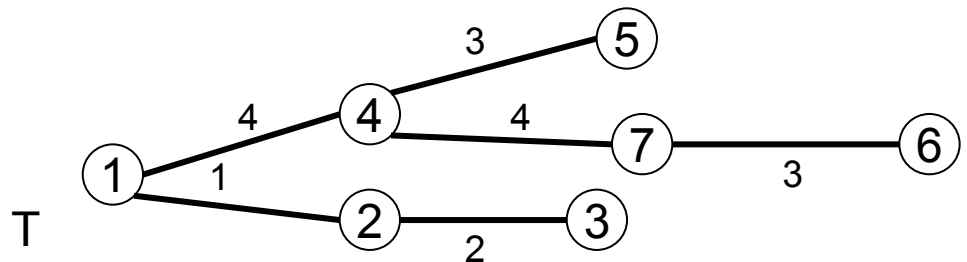
<p>1 [cdis.incluir(v)] $v \in N$ 2 Ordene ascendente las aristas de G por sus pesos w 3 [Si (cdis.buscar(v) \neq cdis.buscar(u)) entonces $X = X \cup \{(u, v)\}$ cdis.union(u, v) // $O(A \lg A)$ fsi] $(u, v) \in A$ por orden ascendente de w 4 regrese X</p>	<p>-X. Conjunto[X]. Árbol de expansión mínima resultante para el grafo. -cdis: ConjDisj[TipoClave]. Bosque de nodos del grafo. -incluir(), buscar(), union(). Definidas en la clase ConjDisj. - $\cup()$. Función de la clase Conjunto [X].</p>
--	--

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

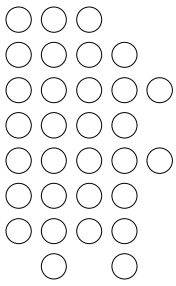
Algoritmo de Kruskal



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)	$\{1,2,3,4,5,6,7\}$



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.