Algoritmos y Estructuras de Datos II Clase 21

Carreras:

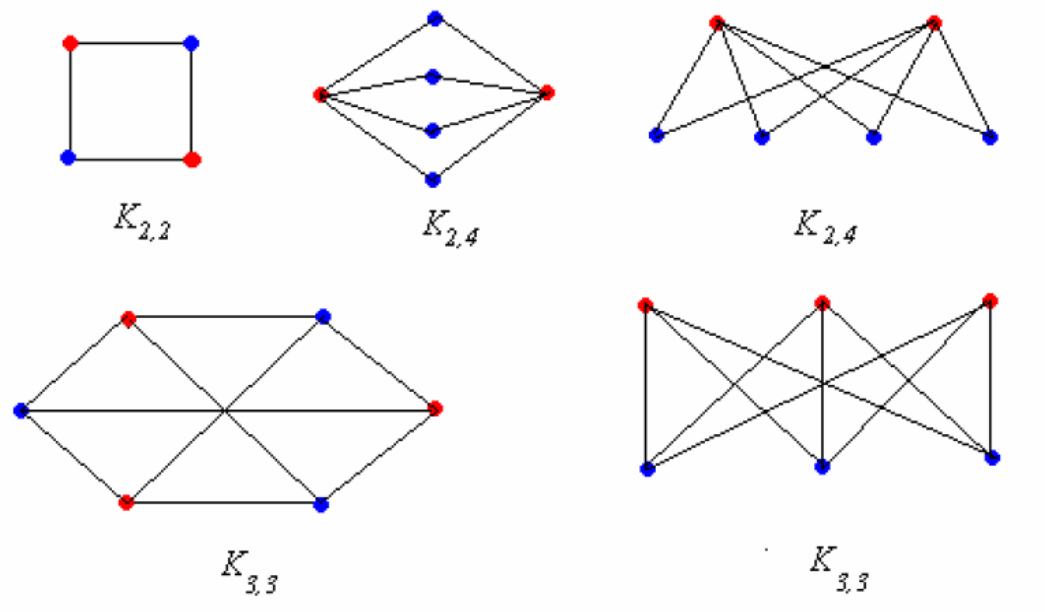
Licenciatura en Informática

Ingeniería en Informática

2024

Unidad IV

Algoritmos en Grafos(1)



Unidad IV

Bibliografia

- *Handbook of graph theory.* Edited by Jonathan l. Gross, Jay Yellen. C RC Press, 2004.
- *Graph Theory and its applications*. Jonathan Gross. Jay Yellen. CRC Press, 1999.
- *Introducción a la teoría de Grafos*. Fausto A. Toranzos. Universidad de Buenos Aires, Argentina, 1976.
- *Introductory Graph Theory*. Gary Chartrand. Dover Publications. New York, 1977.

Definición

Un grafo es una colección de vértices unidos por un conjunto de arcos que denotaremos como:

$$G = (V , E)$$

Donde

V: es un conjunto finito de elementos llamados **vértices, nodos o puntos.**

E $\subseteq$ VxV: es un conjunto finito de pares (ordenados o no ordenados) de vértices llamados **arcos, aristas o ejes**.

Cada arco tiene la forma (v_1,v_2) donde v_1 y $v_2 \in V$

- ***Grafo no dirigido*:** el par de vértices que representa una arista no tiene orden, de modo que (v_1,v_2) y (v_2,v_1) se refieren a la misma arista.
- ***Grafo dirigido o digrafo*:** cada arco se representa por un par ordenado (v_1,v_2).

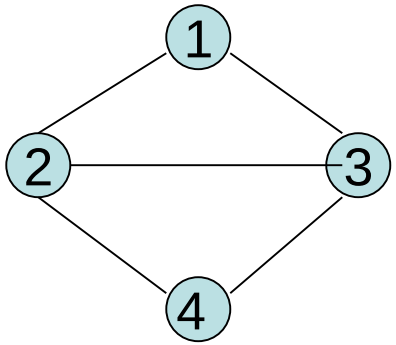
Implementación

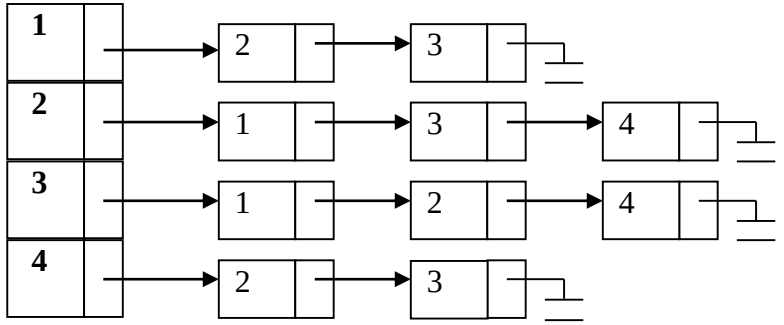
Existen dos formas de implementar un grafo:

- Con matriz de adyacencia
- Con estructura de adyacencia

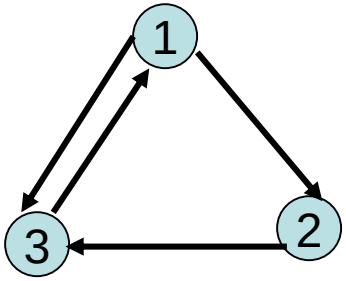
Ejemplos:

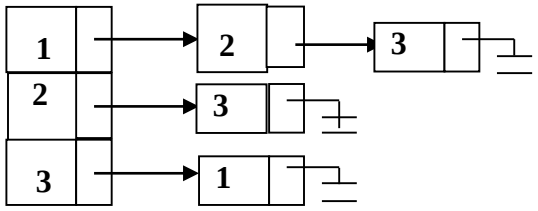
$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$





$$A_2 = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$





Implementación

La **matriz de adyacencia** se puede usar para almacenar un valor de otro tipo en lugar de un valor booleano.

Con esta representación se puede usar una ***matriz de adyacencia con costo***, donde A(i,j) es el ***rótulo*** o ***valor*** o ***costo*** del arco que une el vértice v_i con el vértice v_j. Si no existe tal arco se hace la convención de usar un valor especial constante por ejemplo infinito.

Sea G=(V,E) un grafo donde V= {v_1,v_2,...,v_n } , la matriz de adyacencia con costo de G es una ***matriz A cuadrada de orden n***, donde:

A(i,j) = costo si y solo si existe un arco de v_i a v_j.

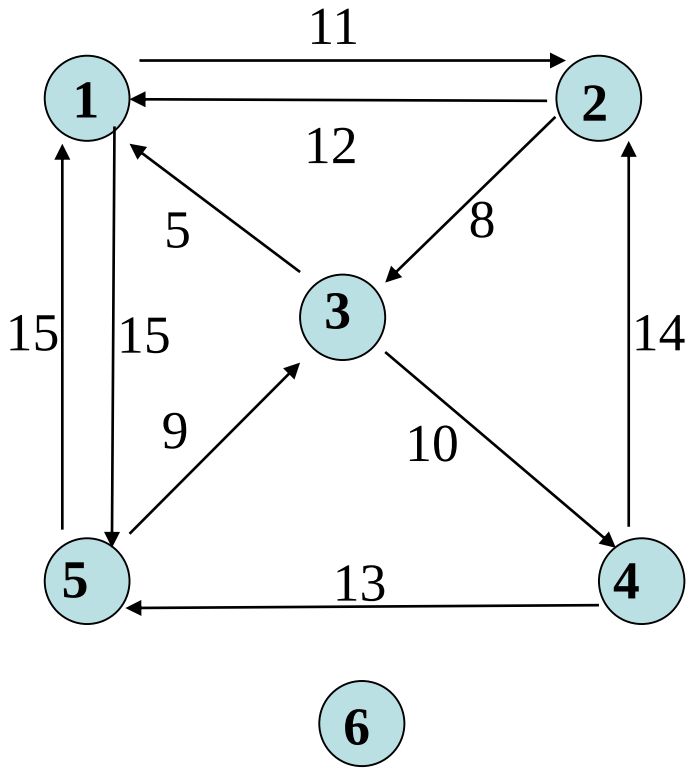
A(i,j) = ∞ si y solo si no existe un arco de v_i a v_j.

Implementación

Ejemplo:

n=6

$$A = \begin{pmatrix} \infty & 11 & \infty & \infty & 15 & \infty \\ 12 & \infty & 8 & \infty & \infty & \infty \\ 5 & \infty & \infty & 10 & \infty & \infty \\ \infty & 14 & \infty & \infty & 13 & \infty \\ 15 & \infty & 9 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$



Aplicaciones de grafos

- Problema de recubrimiento mínimo.
- Problema del camino mínimo.
- Problema del viajante/Ciclo Hamiltoniano.
- Problema de Coloreado.

Árbol de recubrimiento
Spanning Tree

PROBLEMA:

Dado un grafo conexo, no dirigido $g = (V,E)$, encontrar el árbol de recubrimiento de *g.*

- El ***árbol de recubrimiento*** también se denomina: *árbol generador, árbol de expansión o de extensión.*

- Un ***árbol de recubrimiento de un grafo*** es un subgrafo sin ciclos ***s=(V,T),*** que contiene a todos sus vértices **V**, donde ***T*** es un subconjunto de *E.*

- El problema consiste entonces en encontrar el conjunto de aristas ***T*** de modo que todos los nodos queden conectados.

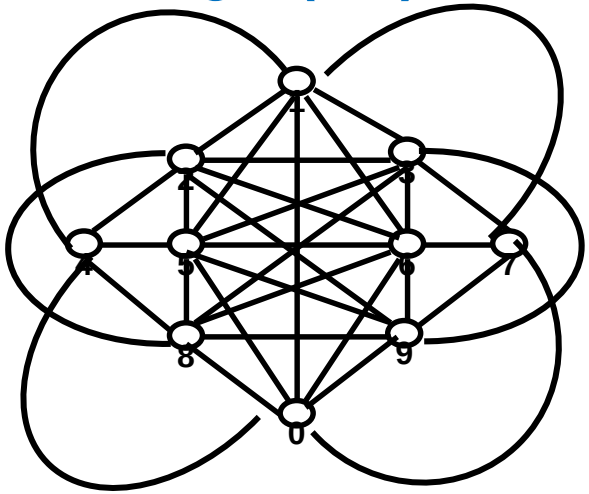
- Como *g* es conexo existe al menos una solución.

Árbol de recubrimiento

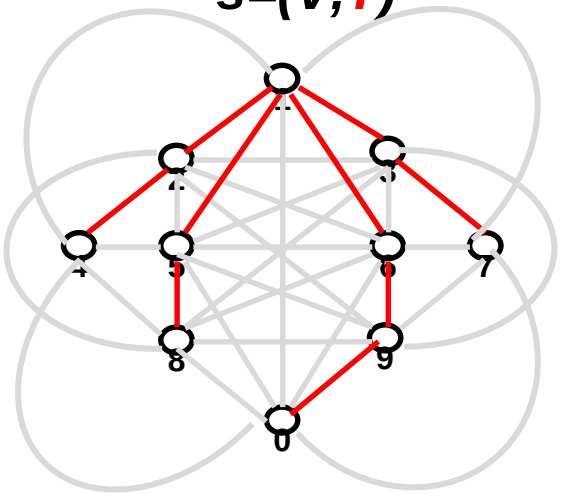
- Un grafo conexo con *n* nodos debe tener por lo menos *n-1* aristas.
- Un grafo conexo con *n* nodos y más de *n-1* aristas contiene al menos un ciclo.
- Como el subgrafo *s* tiene el mismo conjunto de vértices de *g*, si en *V* hay *n* nodos, el subgrafo sin ciclos *s* debe tener exactamente *n-1* aristas en *T*.
- Además como *s* es conexo entonces tiene que ser un árbol.

Ejemplo:

***g* = (*V*,*E*)**

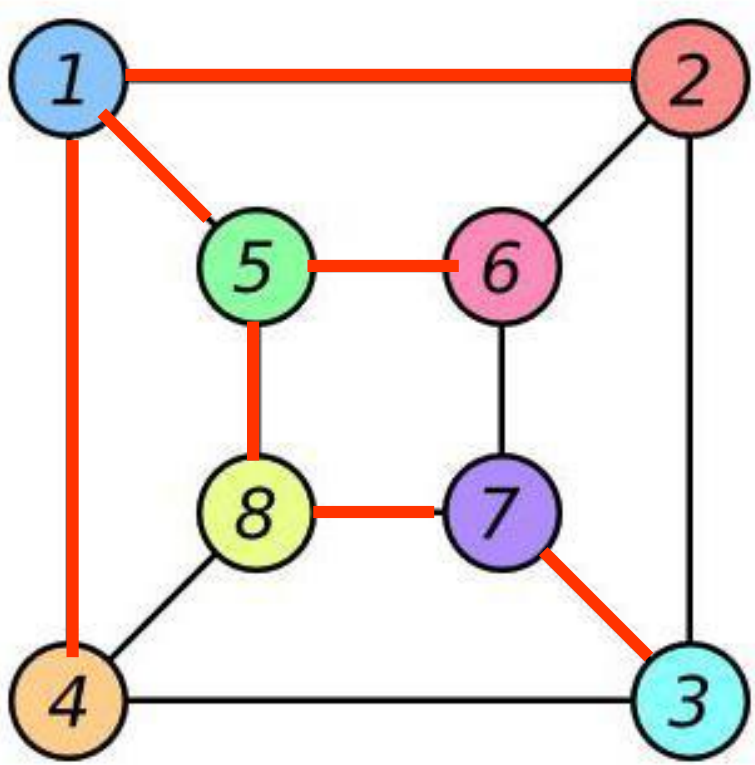


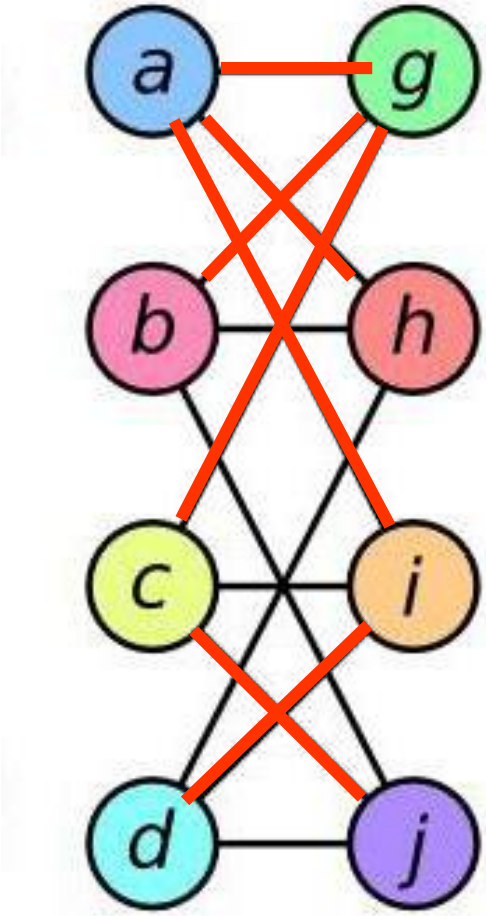
***s*=(*V*,*T*)**



Árbol de recubrimiento
Ejemplo 1

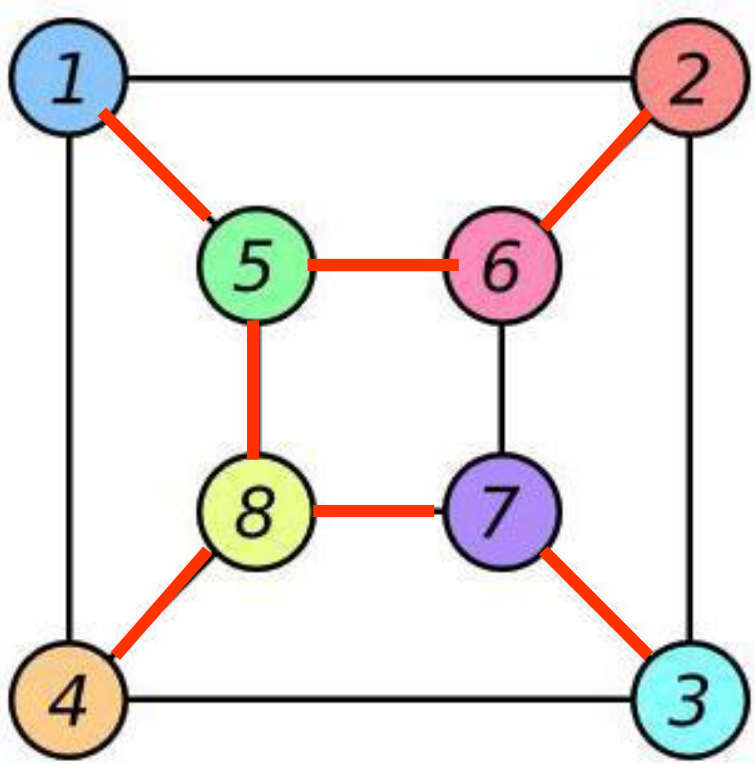
.

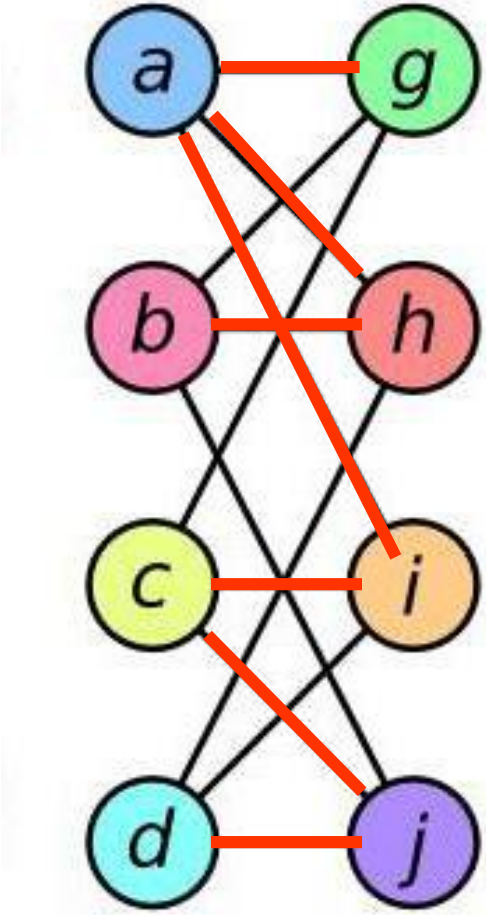




Árbol de recubrimiento
Ejemplo 2

.





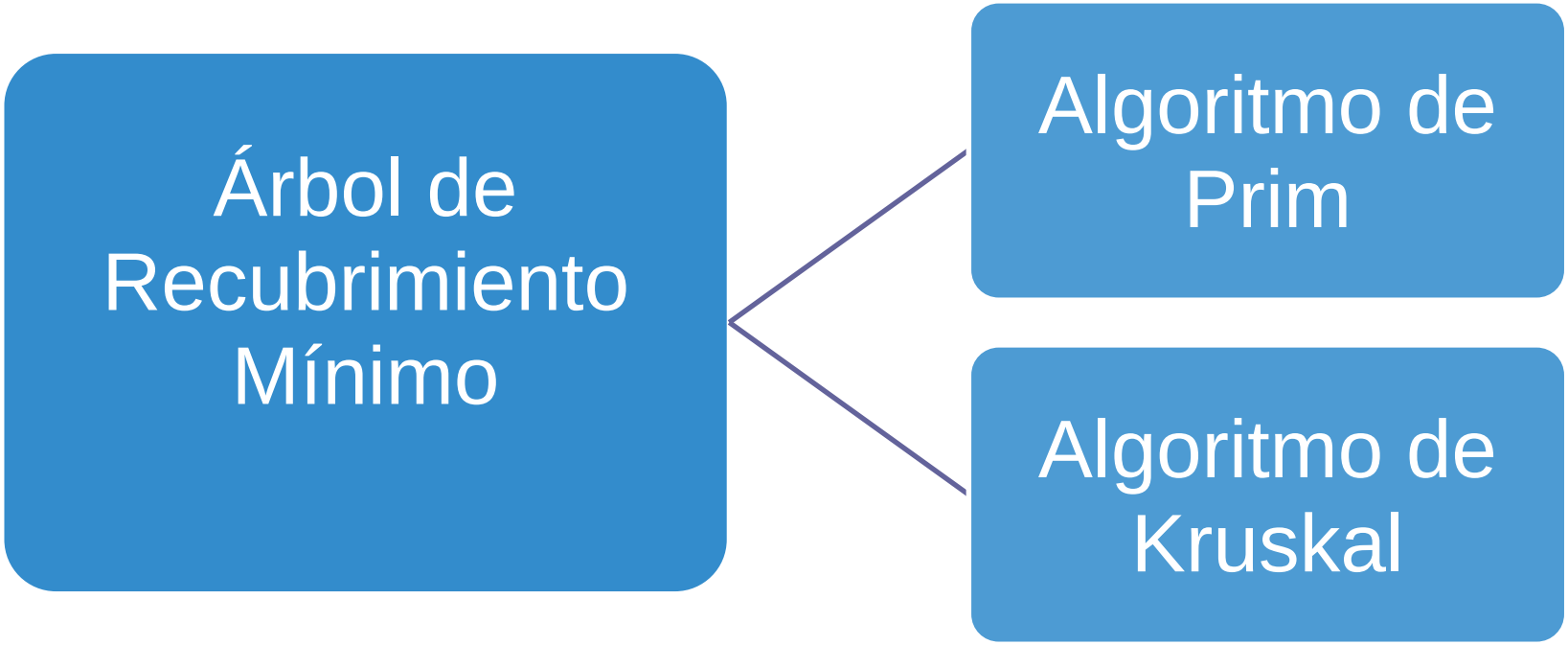
Árbol de recubrimiento mínimo
MST – Minimun Spanning Tree

PROBLEMA:

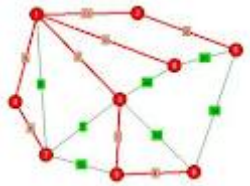
Dado un grafo conexo, ponderado y no dirigido $g = (V,E)$ con costos de arcos no negativos, encontrar el *árbol de recubrimiento* de *g* de *costo mínimo*.

- Un ***árbol de recubrimiento mínimo de un grafo*** es aquel árbol de recubrimiento ***s=(V,T),*** en que la suma de las longitudes de las aristas del conjunto *T* tenga costo mínimo.

Algoritmos Greedy



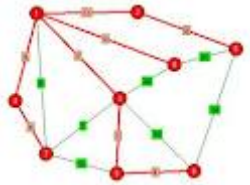
Árbol de recubrimiento mínimo



Existen dos algoritmos greedy muy conocidos que resuelven este problema:

- Algoritmo de Prim (1957) , descubierto por Jarnik (1930) y redescubierto por Prim y Dijkstra.
- Algoritmo de Kruskal (1956).
- En ambos se va construyendo el árbol por etapas, y en cada una se agrega un arco. La forma en la que se realiza esa elección es la que distingue a ambos algoritmos greedy.
- Algoritmos más sofisticados: Yao(1975), Cheriton y Tarjan (1976) y Tarjan (1983).

Árbol de recubrimiento mínimo



- Aplicar el esquema general de los algoritmos Greedy, en este caso:

- Los candidatos: el conjunto de aristas del grafo con sus costos: E

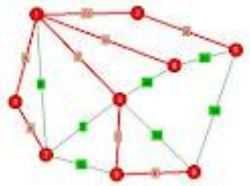
- Un conjunto de aristas es una solución si constituye un árbol de recubrimiento mínimo para los nodos de V.

- Un conjunto de aristas es factible si no contiene ningún ciclo.

- La función de selección se elegirá de acuerdo al algoritmo.

- La función objetivo es: minimizar el costo total de las aristas de la solución.

Árbol de recubrimiento mínimo



El algoritmo de Prim:

- Comienza por un vértice y elige en cada etapa el arco el menor costo que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya seleccionados y el otro afuera del conjunto.
- Al incluir un nuevo arco a la solución, se agrega su otro extremo al conjunto de vértices seleccionados.

En el algoritmo de Kruskal:

- Se arma el árbol de expansión mínima agregando de a una las aristas usando en cada paso la arista de menor costo que no forme ciclo.
- Para ello en cada etapa se elige una arista por su costo en orden creciente y se decide qué hacer con cada una de ellas.
- Si la arista no forma un ciclo con las ya seleccionados se incluye en la solución; sino, se descarta.

Árbol de recubrimiento mínimo

- En el algoritmo de Prim el árbol de recubrimiento mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada paso se agrega una rama al árbol ya construido y el algoritmo se detiene cuando se han alcanzado todos los nodos.

- En el algoritmo de Kruskal, la función de selección elige las aristas por orden creciente de longitud, sin preocuparse por su conexión con las aristas seleccionadas anteriormente, sólo se tiene cuidado de no formar un ciclo con ellas.

Árbol de recubrimiento mínimo

La demostración que ambos algoritmos dan una solución optima se basa en lo siguiente:

Lema: Sea **G = (V , E)** un grafo conexo no dirigido, con costos de arcos no negativos. Sea $S \subset V$ un subconjunto de los vértices de G. Sea $T \subseteq E$ un conjunto prometedor de aristas tal que no haya ninguna arista de T que sale de S. Sea $e \in E$ la arista de longitud mínima que nace en un vértice de S. Entonces T U {e} es un conjunto de aristas prometedor.

Definición: un conjunto de aristas es prometedor si se puede extender para producir no solo una solución del problema sino una solución óptima del problema.

Algoritmo de Prim

El algoritmo de Prim encuentra así un árbol de recubrimiento mínimo del grafo.

(ver demostración: Fundamentos de algoritmia - Brassard & Bratley, Teorema 6-3-3)

- Sea S un conjunto de nodos y sea T un conjunto de aristas.
- Inicialmente S contiene un nodo arbitrario, y T está vacío.
- En cada paso, el algoritmo busca la arista más corta (u,v) tal que u esté en el conjunto S y v esté fuera de él. Entonces agrega ese arco a T y el vértice v a S.
- Entonces en cada paso las aristas de T forman un árbol de recubrimiento mínimo para los nodos de S.
- Se continua así hasta que S coincide con el conjunto de vértices.

Algoritmo de Prim

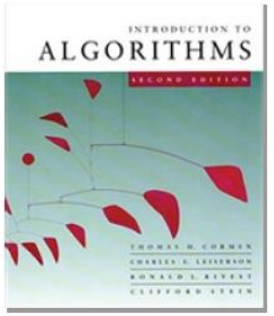
ALGORITMO PRIM (g): grafo→arbol

Obtiene el árbol de recubrimiento mínimo de un grafo

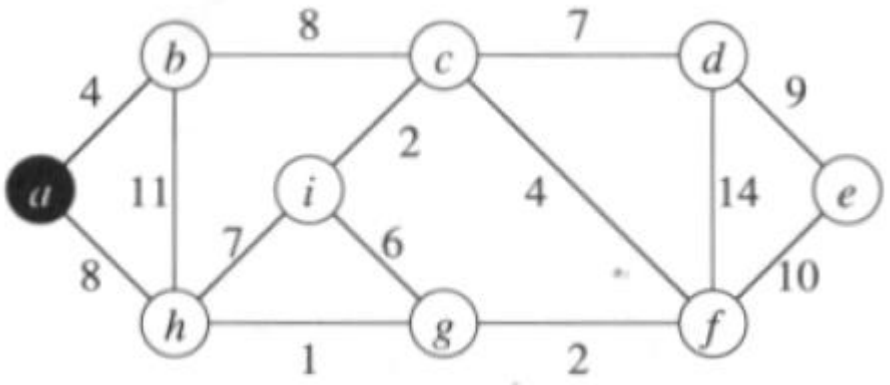
ENTRADA: g, un grafo no dirigido con costos g=(V,E)
SALIDA: T: conjunto de arcos
AUXILIARES: S: conjunto de vértices

P1. S ← { 1 } ; T=∅
P2. Mientras S≠V hacer
Elegir en E un arco e=(v1,v2) de longitud mínima
tal que v1∈S y v2∈V-S
T ← T U {e}
S ← S U {v2}
P3. Retorna T
P4. Fin.

Algoritmo de Prim
Ejemplo (*)



Inicial:



Conjunto de vértices seleccionados

S={a}

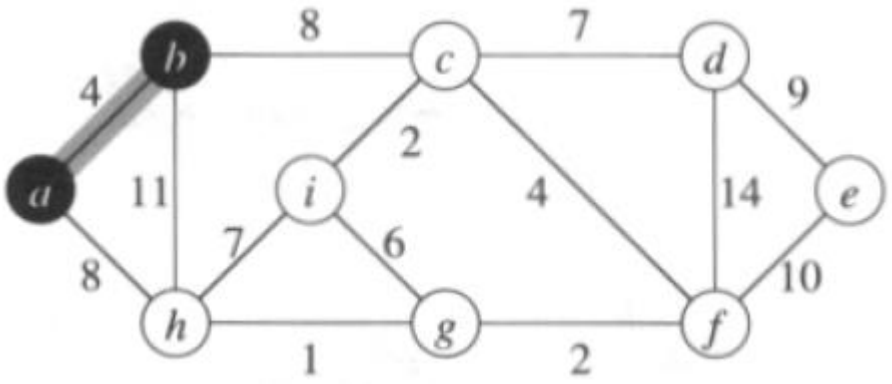
Árbol de recubrimiento

T=∅

(*) Extraído del libro *Introduction to Algorithms,* Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (MIT Press – 2nd Ed. -2001)

Algoritmo de Prim
Ejemplo

Iteración 1: elige (a,b)



Conjunto de vértices seleccionados

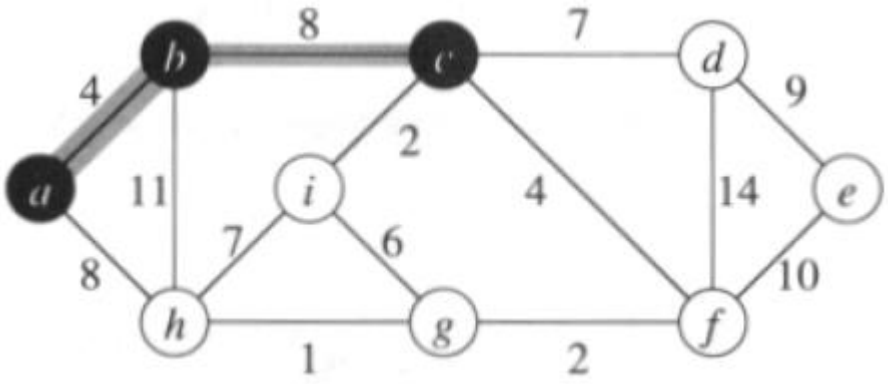
S={a,b}

Árbol de recubrimiento

T= {(a,b)}

Algoritmo de Prim
Ejemplo

Iteración 2: elige (b,c)



Conjunto de vértices seleccionados

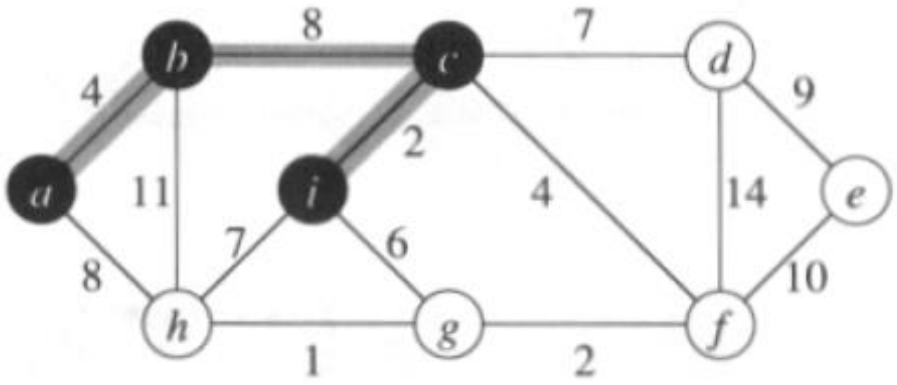
S={a,b,c}

Árbol de recubrimiento

T= {(a,b) (b,c)}

Algoritmo de Prim
Ejemplo

Iteración 3: elige (c,i)



Conjunto de vértices seleccionados

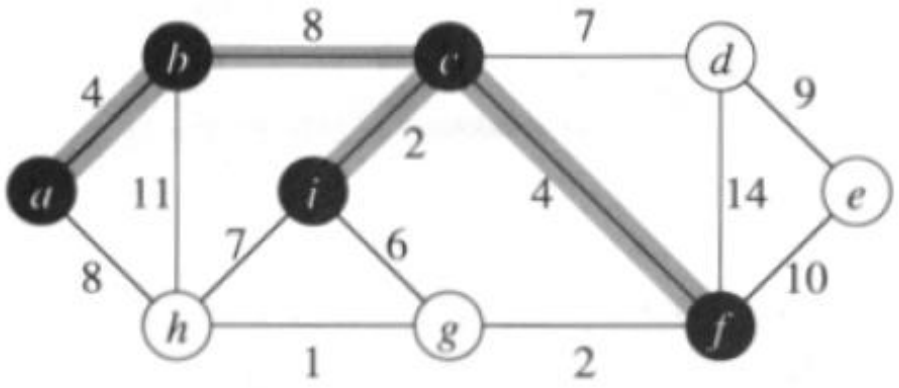
S={a,b,c,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i)}

Algoritmo de Prim
Ejemplo

Iteración 4: elige (c,f)



Conjunto de vértices seleccionados

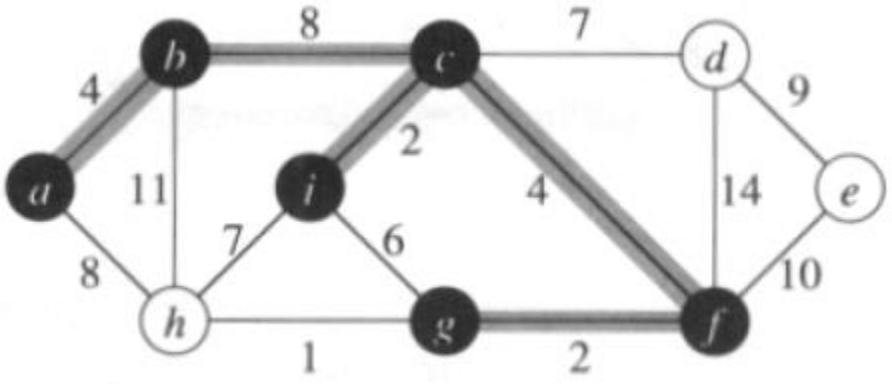
S={a,b,c,f,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i) (c,f)}

Algoritmo de Prim
Ejemplo

Iteración 5: elige (f,g)



Conjunto de vértices seleccionados

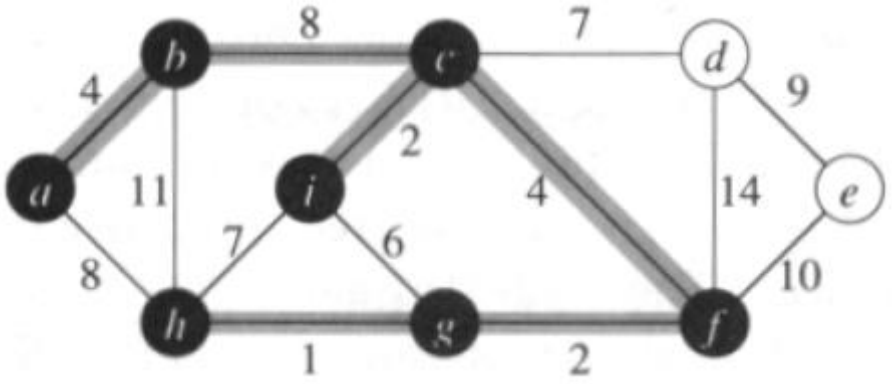
S={a,b,c,f,g,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i) (c,f) (f,g)}

Algoritmo de Prim
Ejemplo

Iteración 6: elige (g,h)



Conjunto de vértices seleccionados

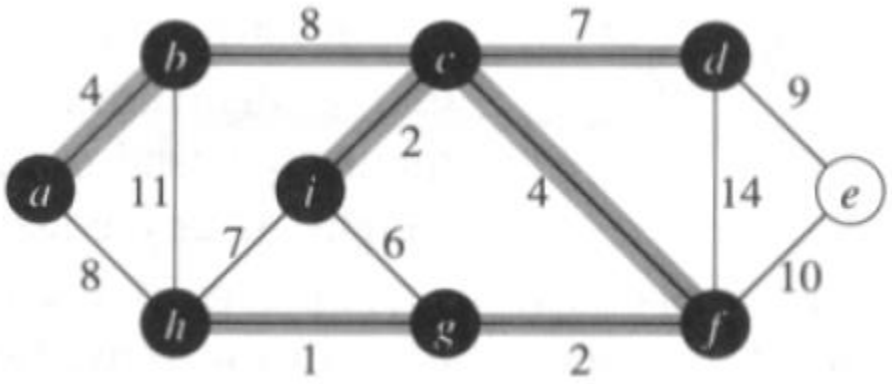
S={a,b,c,f,g,h,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i) (c,f) (f,g) (g,h)}

Algoritmo de Prim
Ejemplo

Iteración 7: elige (c,d)



Conjunto de vértices seleccionados

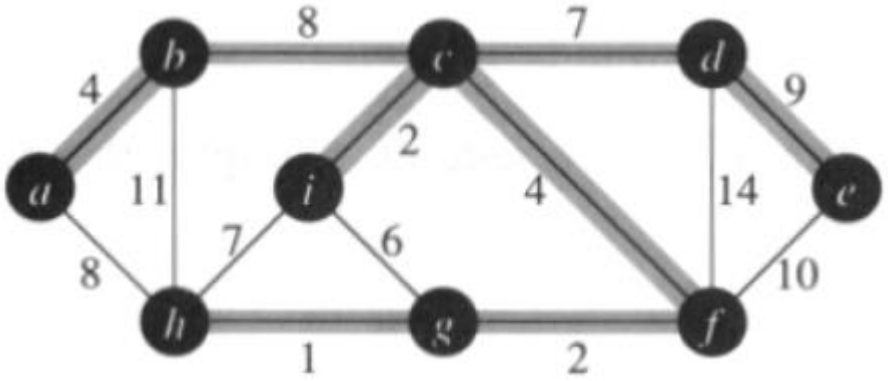
S={a,b,c,d,f,g,h,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i) (c,f) (f,g) (g,h) (c,d)}

Algoritmo de Prim
Ejemplo

Iteración 8: elige (d,e)



Conjunto de vértices seleccionados

S={a,b,c,d,e,f,g,h,i}

Árbol de recubrimiento

T= {(a,b) (b,c) (c,i) (c,f) (f,g) (g,h) (c,d) (d,e)}

Fin del algoritmo

Complejidad Algoritmo de Prim

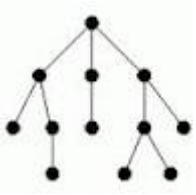
Complejidad temporal:

- Para estimar el ***tiempo de ejecución*** se considera que el algoritmo dado tiene una iteración de n-1 veces, en el cuerpo de la iteración se requiere O(n). Por lo tanto se requiere **O(n^2)**.
- Si se implementa con un montículo es O($a \log n$).
- Si se implementa con Fibonacci heap es O($a + n \log n$).

Complejidad espacial:

- Si se implementa con matriz de adyacencia, el espacio ocupado es de orden **O(n^2)** porque de esta complejidad es la matriz que representa el grafo.

Algoritmo de Kruskal



El algoritmo de Kruskal encuentra un árbol de recubrimiento mínimo del grafo.

(ver demostración: Fundamentos de algoritmia - Brassard & Bratley, Teorema 6-3-2)

- Comienza con el conjunto T de aristas vacío.
- A medida que progresa el algoritmo se van agregando aristas a T.
- En cada paso se selecciona la arista de menor costo.
- Si al elegir esta arista y agregarla a T no se produce un ciclo entonces se agrega a T sino se descarta.

Algoritmo de Kruskal

ALGORITMO KRUSKAL(g): grafo→arbol

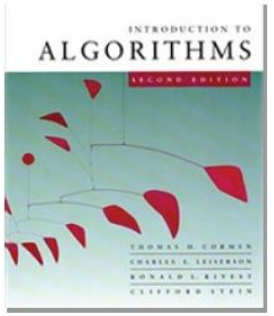
Obtiene el árbol de recubrimiento mínimo de un grafo

ENTRADA: g, un grafo no dirigido con costos g=(V,E)

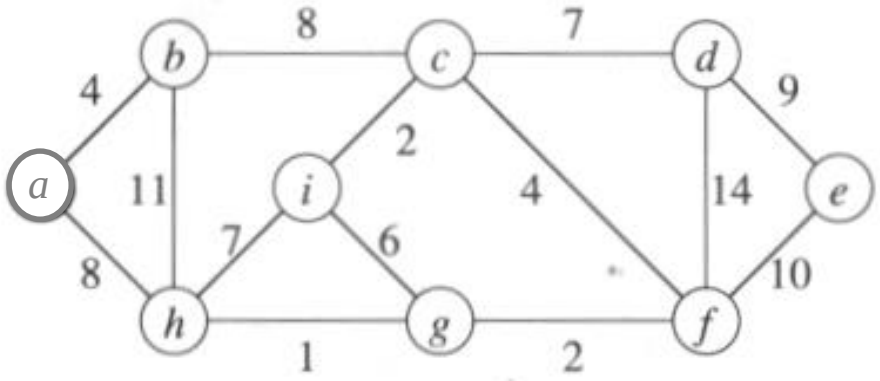
SALIDA: T, un árbol de expansión mínima para g

```
T ← ∅
Mientras T contiene menos de n-1 aristas hacer
        Elegir de E la arista (v,w) de menor costo
        E =  E- {(v,w)}
        Si T U {(v,w)} no tiene ningún ciclo entonces
                T = T U {(v,w)}
                Fusionar la comp. conexas a la que pertenecen v y w
Retorna T
Fin
```

Algoritmo de Kruskal
Ejemplo(*)



Inicial:



Árbol de recubrimiento

T=∅

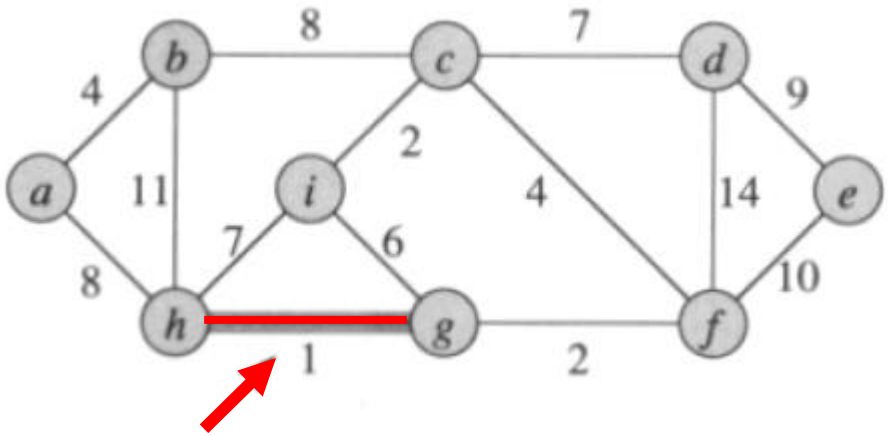
Componentes conexas

{a} {b} {c} {d} {e} {f} {g} {h} {i}

(*) Extraído del libro *Introduction to Algorithms,* Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (MIT Press – 2nd Ed. -2001)

Algoritmo de Kruskal
Ejemplo

Iteración 1: elige (g,h)



Árbol de recubrimiento

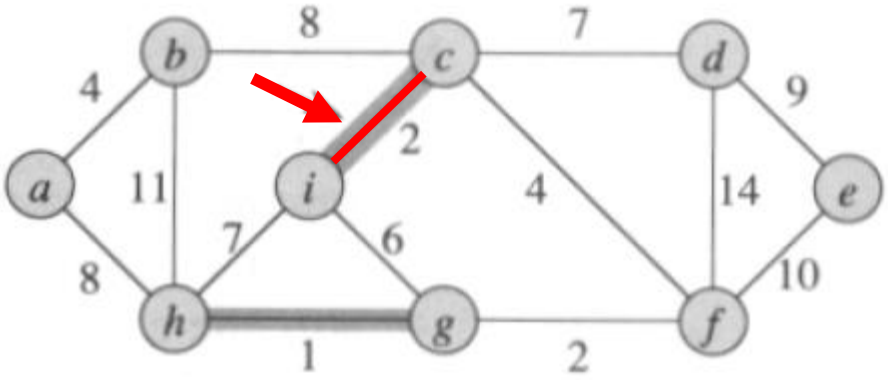
T= {(g,h)}

Componentes conexas

{a} {b} {c} {d} {e} {f} {g,h} {i}

Algoritmo de Kruskal
Ejemplo

Iteración 2: elige (c,i)



Árbol de recubrimiento

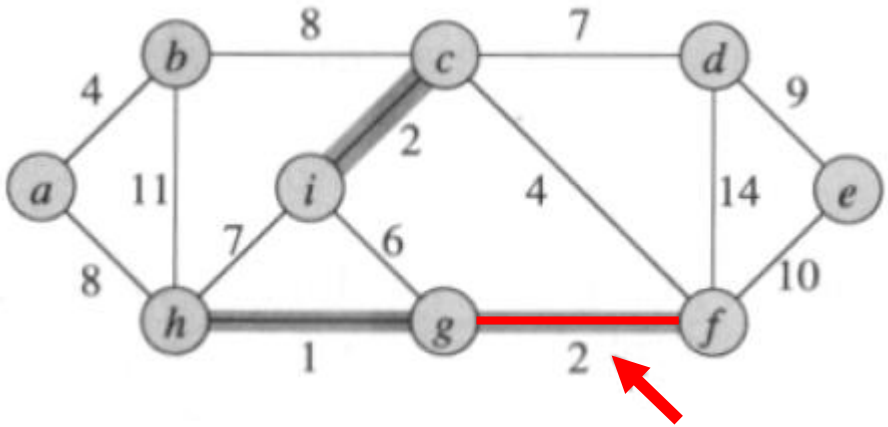
T= {(g,h) (c,i)}

Componentes conexas

{a} {b} {c,i} {d} {e} {f} {g,h}

Algoritmo de Kruskal
Ejemplo

Iteración 3: elige (f,g)



Árbol de recubrimiento

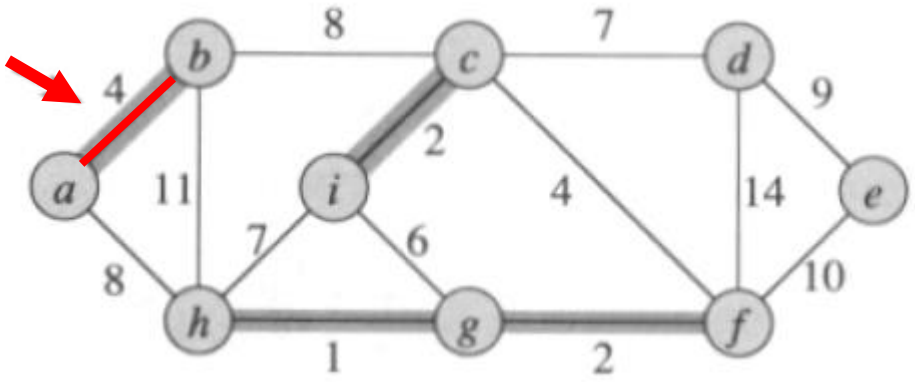
T= {(g,h) (c,i) (f,g)}

Componentes conexas

{a} {b} {c,i} {d} {e} {f,g,h}

Algoritmo de Kruskal
Ejemplo

Iteración 4: elige (a,b)



Árbol de recubrimiento:

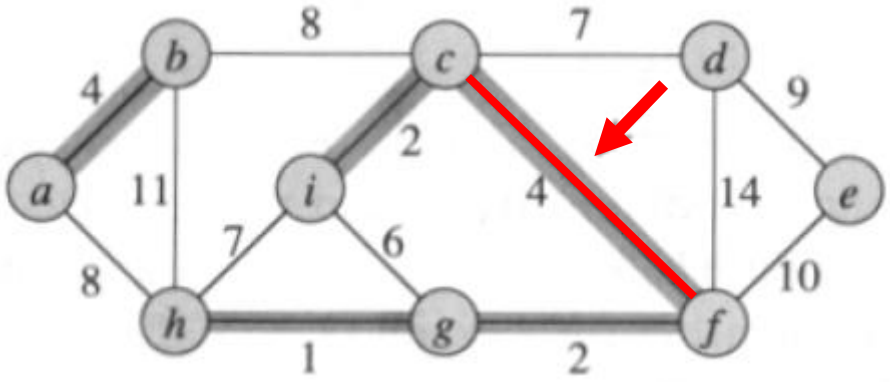
T= {(g,h) (c,i) (f,g) (a,b)}

Componentes conexas:

{a,b} {c,i} {d} {e} {f,g,h}

Algoritmo de Kruskal
Ejemplo

Iteración 5: elige (c,f)



Árbol de recubrimiento:

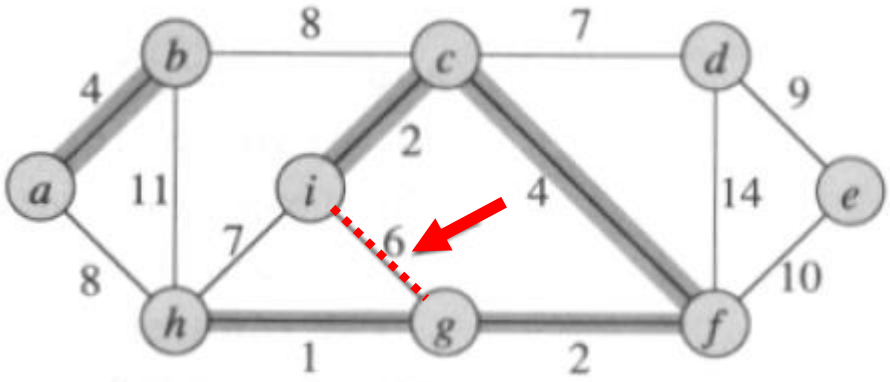
T= {(g,h) (c,i) (f,g) (a,b) (c,f)}

Componentes conexas:

{a,b} {d} {e} {c,f,g,h,i}

Algoritmo de Kruskal
Ejemplo

Iteración 6: elige (g,i) , se descarta porque forma ciclo



Árbol de recubrimiento:

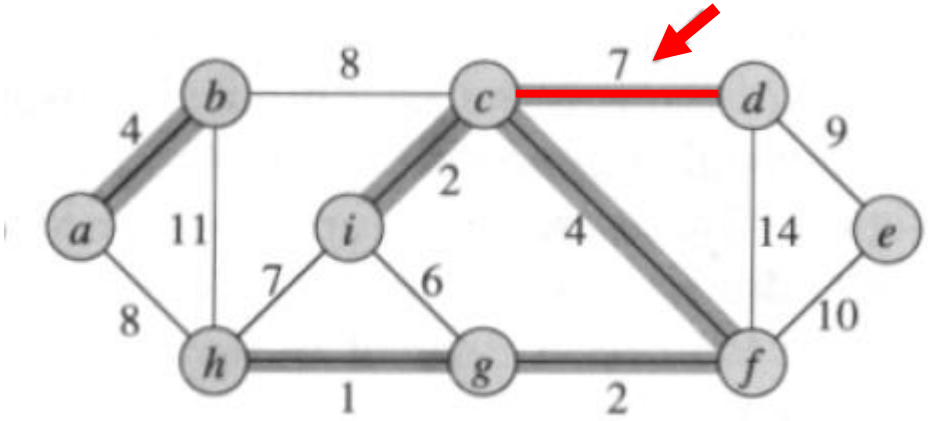
T= {(g,h) (c,i) (f,g) (a,b) (c,f)}

Componentes conexas:

{a,b} {d} {e} {c,f,g,h,i}

Algoritmo de Kruskal
Ejemplo

Iteración 7: elige (c,d)



Árbol de recubrimiento:

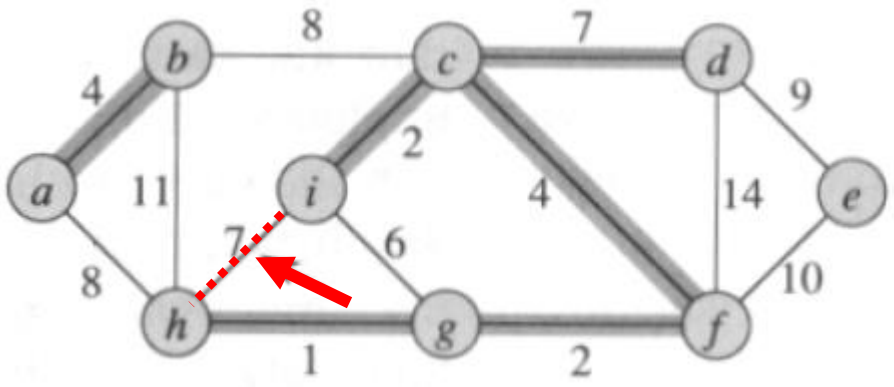
T= {(g,h) (c,i) (f,g) (a,b) (c,f) (c,d)}

Componentes conexas:

{a,b} {e} {c,d,f,g,h,i}

Algoritmo de Kruskal
Ejemplo

Iteración 8: elige (h,i) y se descarta porque forma ciclo



Árbol de recubrimiento:

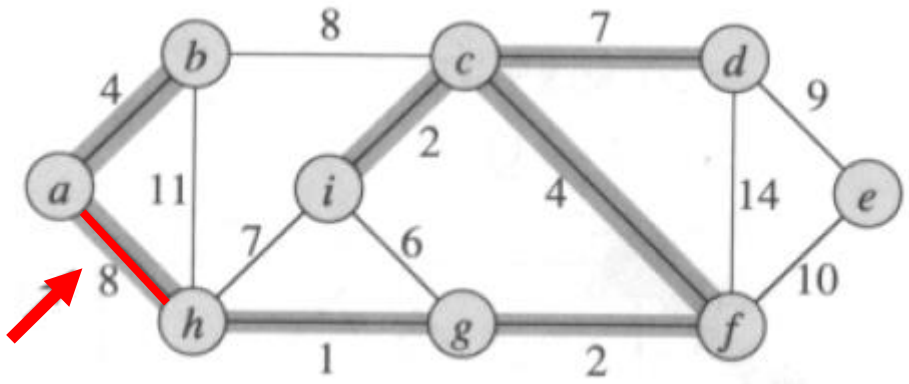
T= {(g,h) (c,i) (f,g) (a,b) (c,f) (c,d)}

Componentes conexas:

{a,b} {e} {c,d,f,g,h,i}

Algoritmo de Kruskal
Ejemplo

Iteración 9: elige (a,h)



Árbol de recubrimiento:

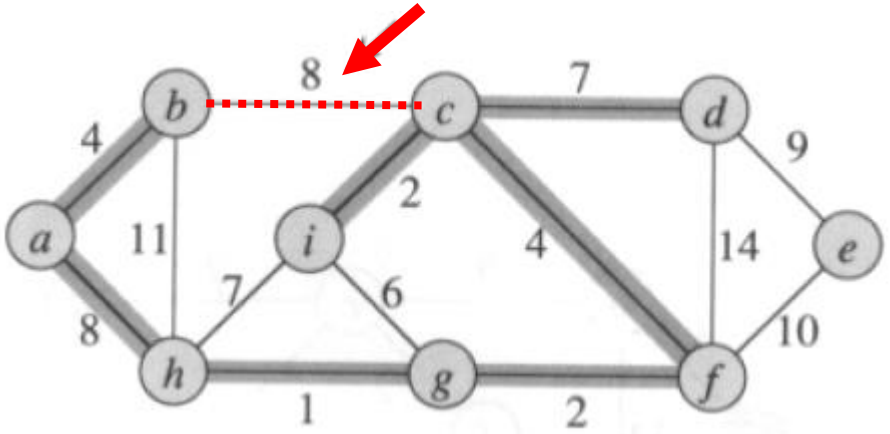
T= {(g,h) (c,i) (f,g) (a,b) (c,f) (c,d) (a,h)}

Componentes conexas:

{a,b,c,d,f,g,h,i} {e}

Algoritmo de Kruskal
Ejemplo

Iteración 10: elige (b,c) y se descarta porque forma ciclo



Árbol de recubrimiento:

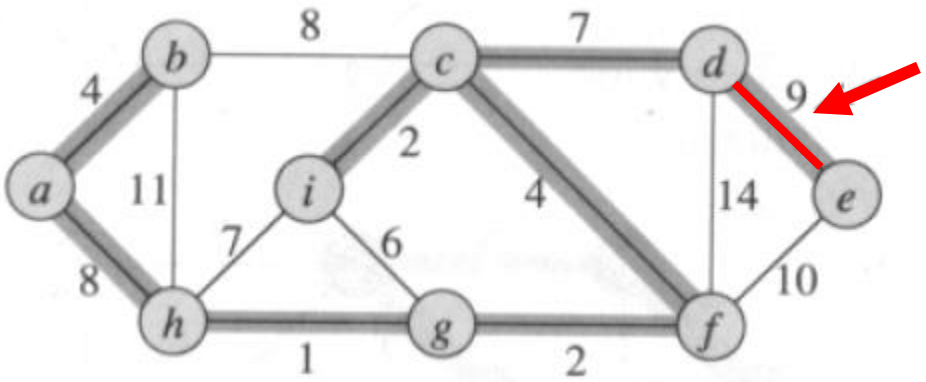
T= {(g,h) (c,i) (f,g) (a,b) (c,f) (c,d) (a,h)}

Componentes conexas:

{a,b,c,d,f,g,h,i} {e}

Algoritmo de Kruskal
Ejemplo

Iteración 11: elige (d,e)



Árbol de recubrimiento:

T= {(g,h) (c,i) (f,g) (a,b) (c,f) (c,d) (a,h) (d,e)}

Componentes conexas:

{a,b,c,d,e,f,g,h,i}

Fin del algoritmo

Complejidad del Algoritmo de Kruskal

Complejidad temporal, si el grafo tiene *n* nodos y *a* aristas, entonces el orden de complejidad de cada parte es:

- Para ordenar las aristas se necesita θ(*a* log *a*) equivalente a θ (*a* log *n*) porque $n-1 \leq a \leq n(n-1)/2$ por ser un grafo conexo.
- θ(*n*) para inicializar las n componentes conexas
- Las restantes operaciones ϵ θ(log *n*) dentro de un lazo O(*n*)

- Por lo tanto: su complejidad temporal es de orden **O(*a* log *n*).**

Complejidad espacial, es de orden **O(n^2)** porque de esta complejidad es la matriz que representa el grafo ordenado.

Si en vez de utilizar matrices de adyacencia se utiliza lista enlazada, la complejidad espacial del algoritmo es de orden **O(*a*)** (puesto que sólo hay que almacenar los arcos, y por ser un grafo conexo se sabe que: $n-1 \leq a \leq n(n-1)/2$),

Complejidad Algoritmo de Prim/Kruskal

- El algoritmo de Prim es siempre de orden $O(n^2)$ mientras que el orden de complejidad del algoritmo de Kruskal $O(a \log n)$ no sólo depende del número de vértices, sino también del número de arcos.

- Para *grafos densos* el número de arcos a es cercano a: $n(n-1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2 \log n)$, peor que la complejidad $O(n^2)$ de Prim.

- Sin embargo, para *grafos dispersos* en los que a es próximo a n, el algoritmo de Kruskal es de orden $O(n \log n)$, comportándose probablemente de forma más eficiente que el de Prim.

Definición de camino

Un ***camino en un digrafo*** (cadena en un grafo) es una secuencia finita de vértices $< v_0 , v_1, v_2, ..., v_k >$ unidos por arcos (aristas).

La ***longitud del camino*** es el número de arcos en ese camino: k

El ***camino es simple*** si todos sus vértices, excepto el primero y el último son distintos.

Un ***circuito (ciclo)*** es un camino en el que coinciden el primero y el último vértice.

Un ***circuito simple (ciclo simple)*** es un camino simple en el que coinciden el primero y el último vértice.

Problema del camino mínimo

PROBLEMA: Dado un digrafo G(V,E), en el cual los arcos tienen costo no negativo y dado un vértice origen v_0, el problema es ***determinar el camino de costo mínimo*** desde v_0 a los demás vértices del grafo.

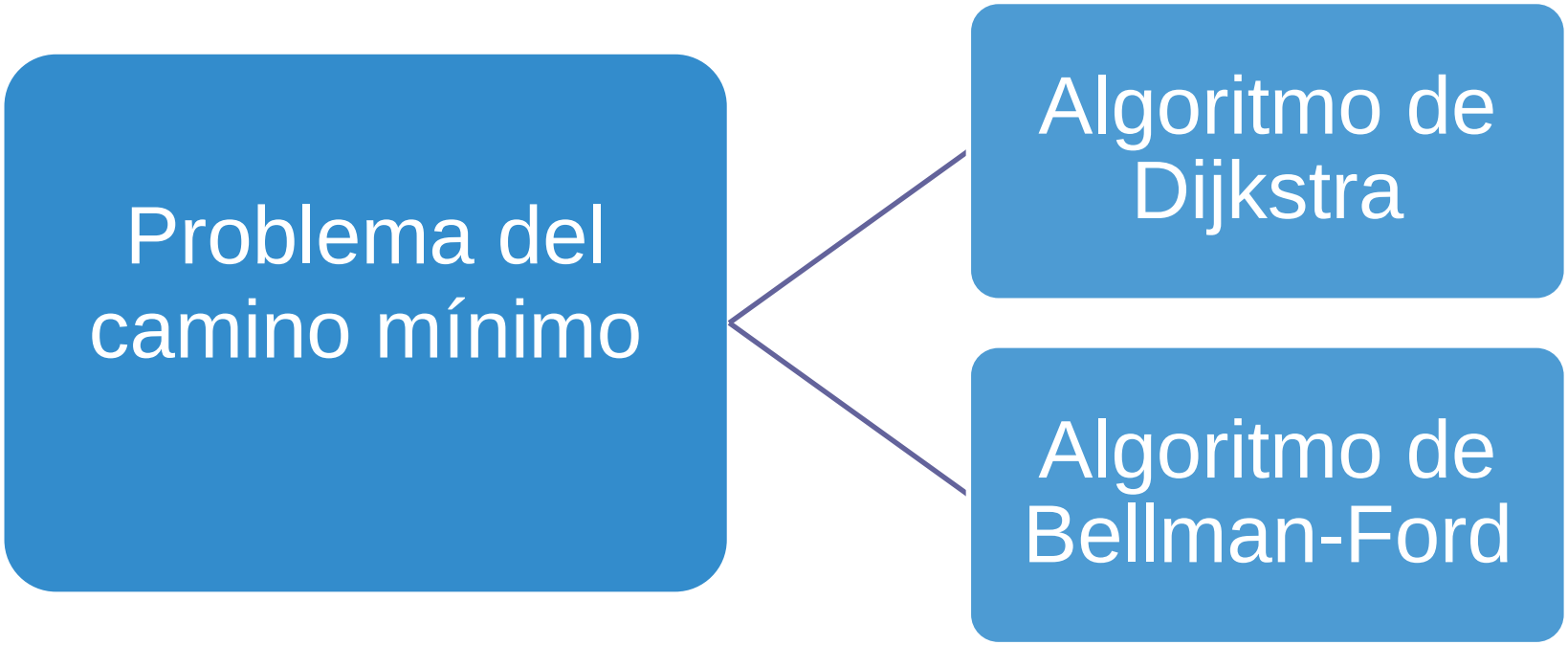
En un grafo rotulado, el ***costo del camino*** es la suma de los costos de los arcos que lo forman.

Este problema puede resolverse por distintos algoritmos.

Uno de los primeros algoritmos Greedy llamado ***ALGORITMO DE DIJKSTRA*** (1956) obtiene siempre una solución óptima (si es que existe).

(ver demostración: Fundamentos de algoritmia - Brassard & Bratley, Teorema 6-4-1)

Algoritmos Greedy



Algoritmo de Dijkstra

ALGORITMO DIJKSTRA (A,n,D,P)

Calcula el costo del camino de menor costo desde el vértice 1 a cada vértice del grafo.

ENTRADA: n: número de vértices.
A: matriz de adyacencia con los costos positivos.

SALIDA: D: vector de distancias especiales.
P: vector del camino.

AUXILIARES: S: conjunto de vértices elegidos.
C: conjunto de vértices candidatos

Algoritmo de Dijkstra

ALGORITMO DIJKSTRA (A,n,D,P)

P1. S ← { 1 } C ← { 2, 3, ..., n } P(1) ← 0

P2. Para i desde 2 hasta n hacer

D(i) ← A (1, i)

P(i) ← 1

P3. Repetir n-2 veces

Elegir en C un vértice w tal que D (w) sea mínimo

Agregar w a S

Borrar w de C

Para cada vértice v en C hacer

Si (D (w) + A (w , v)) < D (v) entonces

D (v) ← D (w) + A (w , v)

P (v) ← w

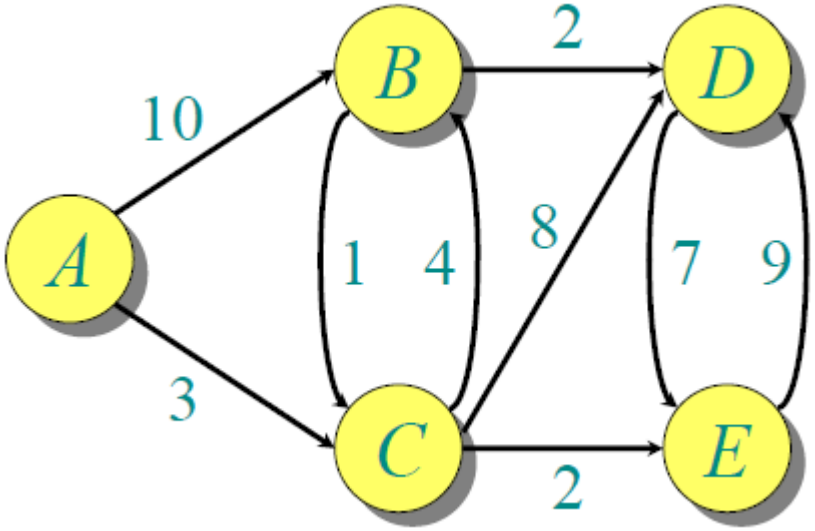
P4. Fin.

Algoritmo de Dijkstra
Ejemplo(*)

Datos del Problema:

Grafo: n=5, con Vértices : V={A, B, C, D, E}

Arcos: Matriz de Adyacencia con costos positivos:

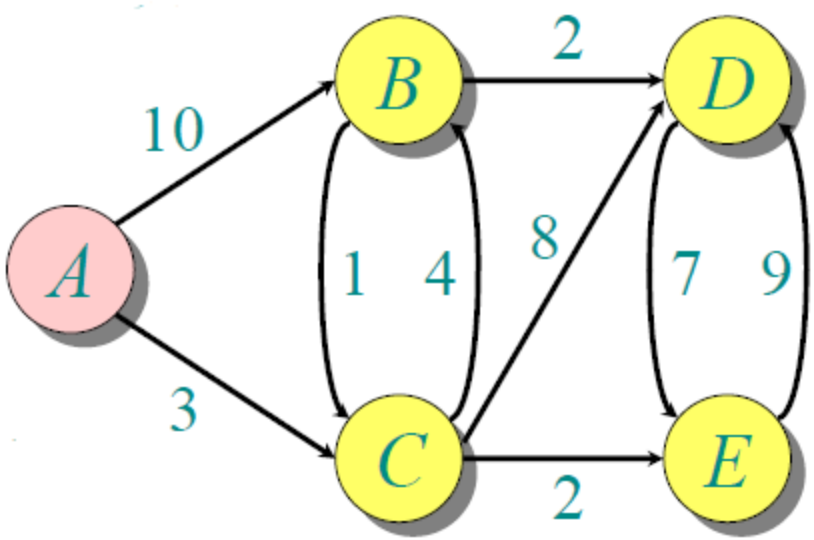


	A	B	C	D	E
A	0	10	3	∞	∞
B	∞	0	1	2	∞
C	∞	4	0	8	2
D	∞	∞	∞	0	7
E	∞	∞	∞	9	0

S={ } C={A, B, C, D, E}

(*) Extraído del curso *Introduction to Algorithms,* Erik Demaine and Charles Leiserson, *(Massachusetts Institute of Technology-2005).*

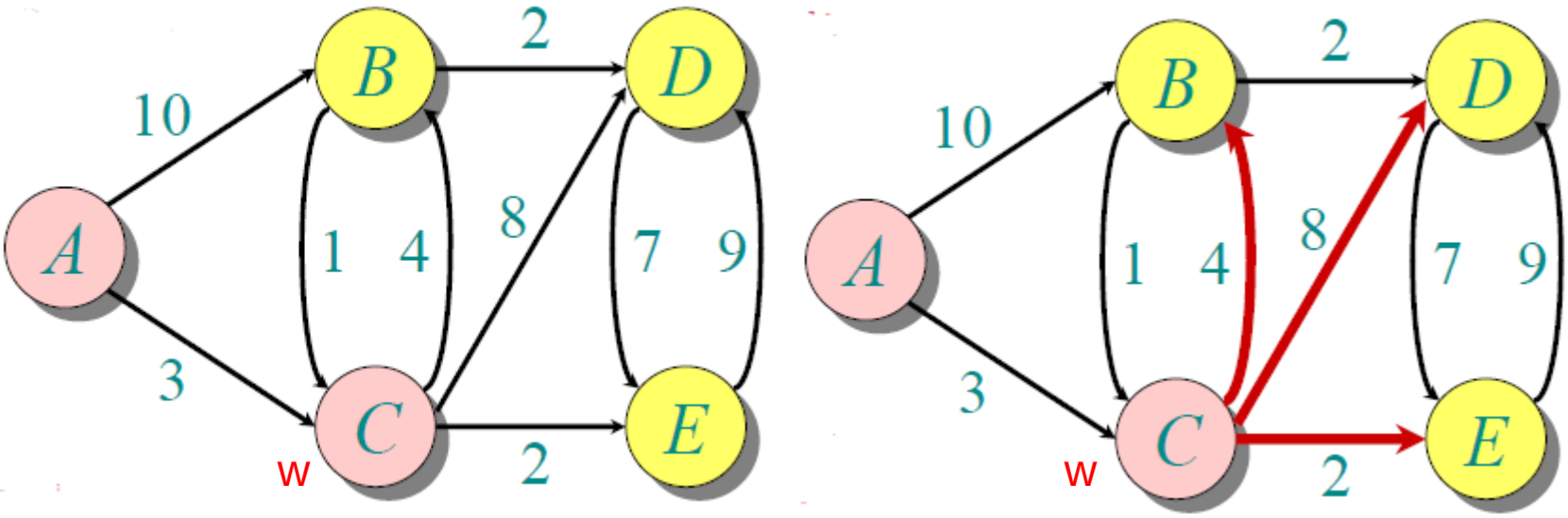
Algoritmo de Dijkstra
Ejemplo



	A	B	C	D	E
A	0	10	3	∞	∞
B	∞	0	1	2	∞
C	∞	4	0	8	2
D	∞	∞	∞	0	7
E	∞	∞	∞	9	0

Inicial:
Se elige el vértice de partida: **A**
S={ **A** } C={B, C, D, E}
Se inicializa los vectores: D=(**0 10 3** ∞ ∞) P=(A A A A A)

Algoritmo de Dijkstra
Ejemplo



S={ A} C={B, C, D, E} D=(0 10 **3** ∞ ∞)

Iteración 1:

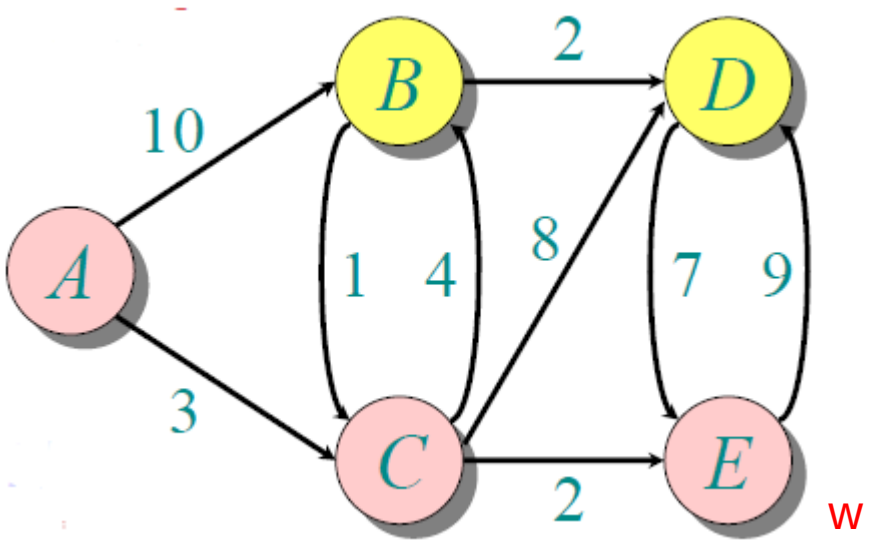
Se elige el vértice w de menor distancia al origen: w=**C** d(w) =3

Se agrega el vértice C al conjunto S={ A,**C**} C={B, D, E}

Se redefinen las distancias de los demás vértices al origen pasando por C:

D=(0 **7** 3 **11** **5**) P=(A C A C C)

Algoritmo de Dijkstra
Ejemplo



S={ A,C} C={B, D, E} D=(0 7 3 11 **5**)

Iteración 2:

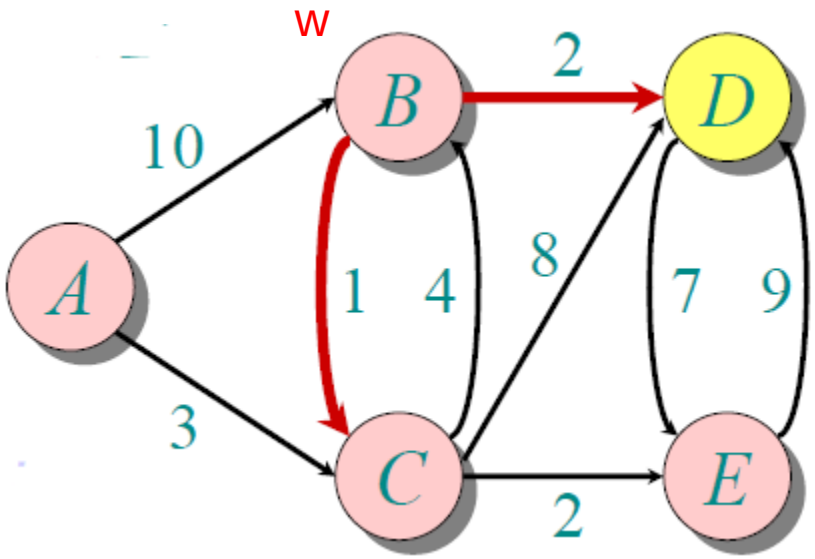
Se elige el vértice de menor distancia al origen: w=**E** d(w)=5

Se agrega al conjunto S={ A,C**,** **E**} C={B, D}

Se redefinen las distancias al origen pasando por E que no cambian:

D=(0 7 3 11 5) P=(A C A C C)

Algoritmo de Dijkstra
Ejemplo



S={ A,C**,** E} C={B, D} D=(0 **7** 3 11 5)

Iteración 3:

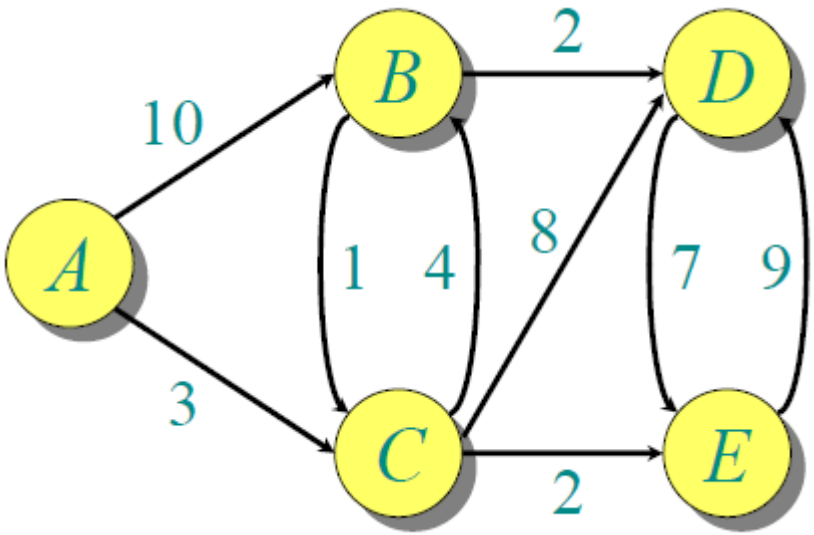
Se elige el vértice de menor distancia al origen: w=**B**, d(w)=7

Se agrega al conjunto S={ A,C**,** E, **B**} C={D}

Se redefine las distancias al origen pasando por E:

D=(0 7 3 **9** 5) P=(A C A B C)

Algoritmo de Dijkstra
Ejemplo

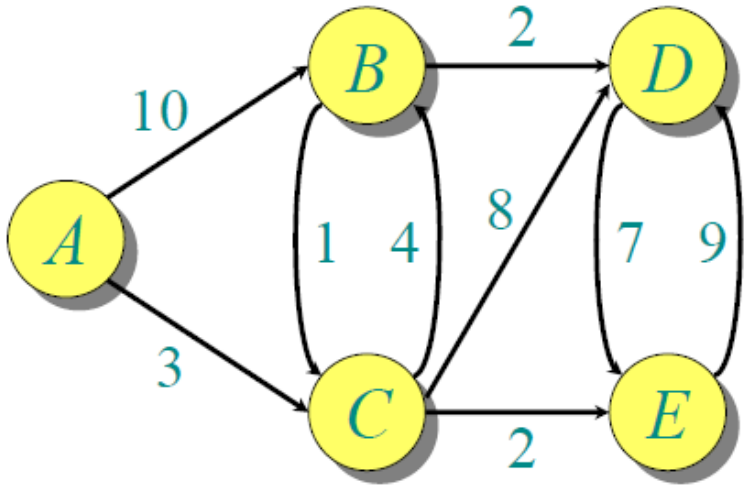


	A	B	C	D	E
A	0	10	3	∞	∞
B	∞	0	1	2	∞
C	∞	4	0	8	2
D	∞	∞	∞	0	7
E	∞	∞	∞	9	0

Inicial: S={ **A** } C={B, C, D, E} D=(0 10 3 ∞ ∞) P=(A A A A A)
it 1: w=**C** d(w)=3 S={ A,**C**} C={B, D, E} D=(0 7 3 11 5) P=(A C A C C)
it 2: w=**E** d(w)=5 S={ A,C**,** **E**} C={B, D} D=(0 7 3 11 5) P=(A C A C C)
it 3: w=**B**, d(w)=7 S={ A,C**,** E, **B**} C={D} D=(0 7 3 9 5) P=(A C A B C)

Resultados: D=(**0 7 3 9 5**) P=(**A C A B C**)

Algoritmo de Dijkstra
Ejemplo



En este ejemplo resulta:

D=(**0 7 3 9 5**)

P=(**A C A B C**)

Con el vector P se puede recuperar cuales son los vértices del camino de mínimo costo:

El camino de menor costo a cada vértice será:

De A a B : P(B) = C , P(C)=A, el camino es: A→C→B de costo 7

De A a C : P(C) = A, el camino es: A→C de costo 3

De A a D : P(D)=B, P(B) = C, P(C)=A , el camino es: A→C→B→D, costo 9

De A a E : P(E) = C, P(C)=A el camino es: A→C→E de costo 5

Algoritmo de Dijkstra
Costo

Dado un digrafo G con n nodos y con a arcos, el costo del algoritmo de Dijkstra depende de la implementación del grafo:

- Matriz de adyacencia:

  Algoritmo de Dijkstra ϵ $O(n^2)$

- Con una cola de prioridad implementada con heap:

  Algoritmo de Dijkstra ϵ $O((a+n) \log n)$

- Si todos los vértices son alcanzables desde el origen, se reduce a:

  Algoritmo de Dijkstra ϵ $O(a \log n)$

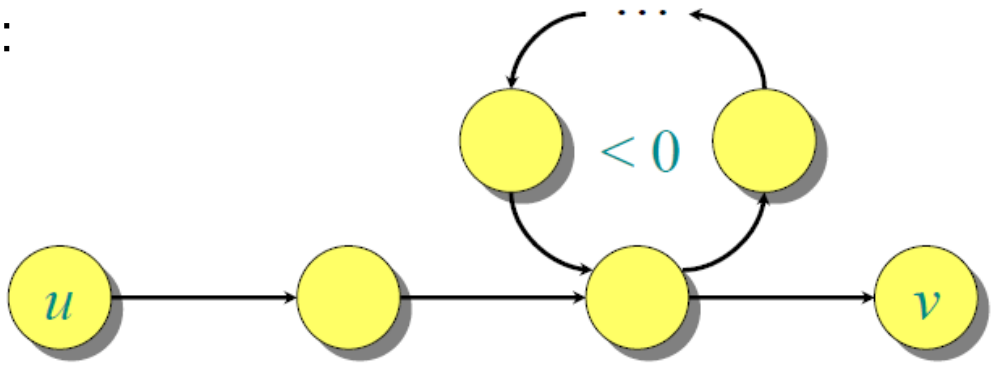
Algoritmo de Bellman-Ford

PROBLEMA: obtener el camino mínimo desde un vértice origen al resto de los vértices del grafo.

El Algoritmo de Bellman-Ford resuelve el problema, aunque el grafo tenga aristas con costo negativo.

- Detecta si el grafo tiene un ciclo de costo negativo.
- Si no hay ciclos negativos, devuelve la respuesta.
- Si hay ciclos negativos el problema no se puede resolver.

Ejemplo:



Algoritmo de Bellman-Ford

El algoritmo se puede escribir en tres pasos fundamentales:

- Inicializar
- Relajar aristas
- Detectar ciclos negativos

Relajar:

- Relajar la arista (*u*,*v*) significa chequear si se mejora el camino mínimo desde el vértice *u* al vértice *v* atravesando la arista (*u*, *v*).
- Si el camino pasando por (u,v) es de menor costo entonces actualizar el camino mínimo desde *u* hasta *v*.
- De este modo, en el paso *i* se asegura de tener el camino mínimo desde *u* a *v*, usando a lo sumo *i* aristas.

Algoritmo Bellman-Ford (G,A,n,D,C): grafo x matriz x ent → vector x bool

ENTRADA: G: grafo G(V,E)
A: matriz de adyacencia con los costos.
n: número de vértices.

SALIDA: D: vector de distancias especiales.
C: bool, true: indica que no existen ciclos de costo negativo

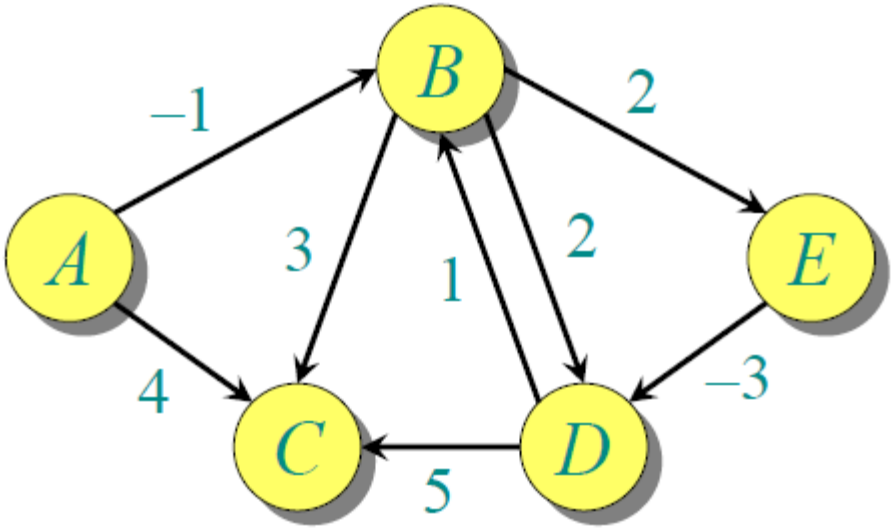
```
D(1) ← 0          // origen vértice 1
C ← True
Para i desde 2 hasta n hacer
    D( i ) ← ∞                                  // inicializar
Para i desde 1  hasta n-1 hacer
        Para cada arista (u,v) en E hacer       // relajar
                Si   D (v)  >  (D ( u ) + A ( u , v )) entonces
                        D (v) ←   D ( u ) + A ( u , v )
Para cada arista (u,v) en E hacer               // detectar ciclo de costo negativo
            Si   D (v)  >  (D ( u ) + A ( u , v )) entonces
                        C ←   False             // hay ciclo de costo negativo
Fin.
```

Complejidad del Algoritmo Bellman-Ford ϵ O(nxa)

Algoritmo de Bellman-Ford

EJEMPLO(*): dado el grafo G

V={A,B,C,D,E}



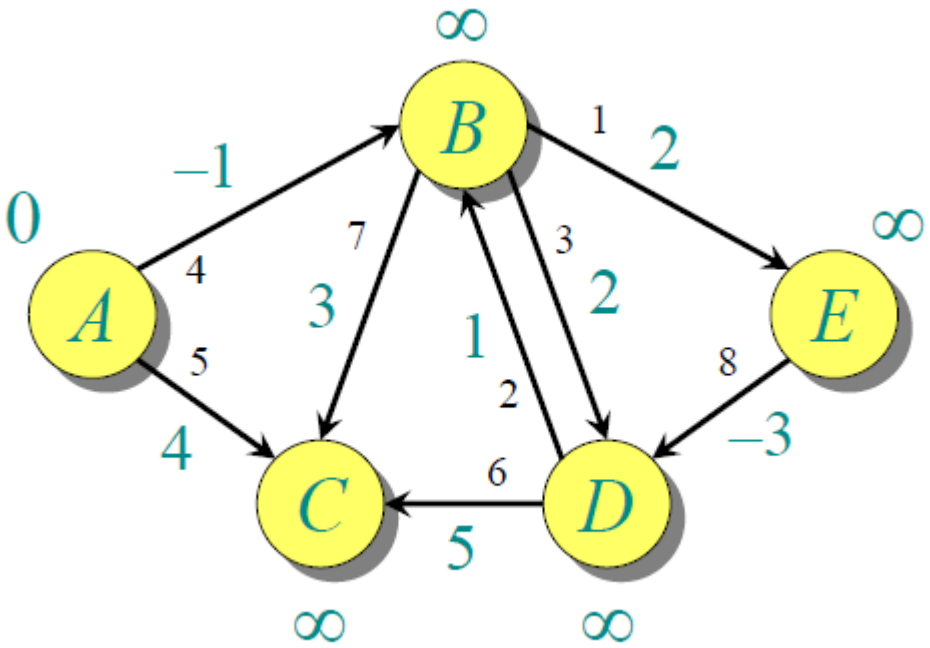
	A	B	C	D	E
A	0	-1	4	∞	∞
B	∞	0	3	2	2
C	∞	∞	0	∞	∞
D	∞	1	5	0	∞
E	∞	∞	∞	-3	0

(*) Extraído del curso *Introduction to Algorithms,* Erik Demaine and Charles Leiserson, *(Massachusetts Institute of Technology-2005).*

Algoritmo de Bellman-Ford

Orden para relajar las 8 aristas:

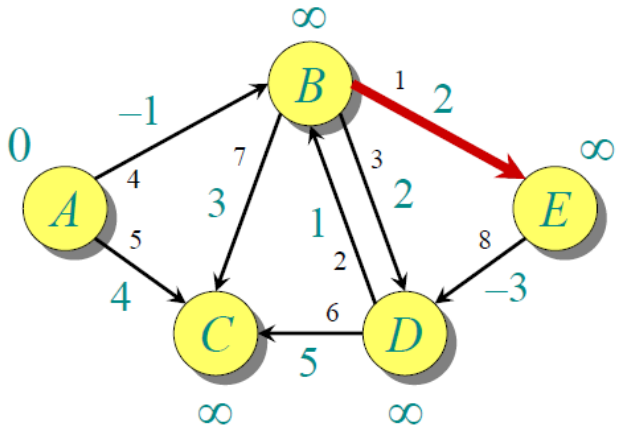
orden	1	2	3	4	5	6	7	8
arista	(B,E)	(D,B)	(B,D)	(A,B)	(A,C)	(D,C)	(B,C)	(E,D)

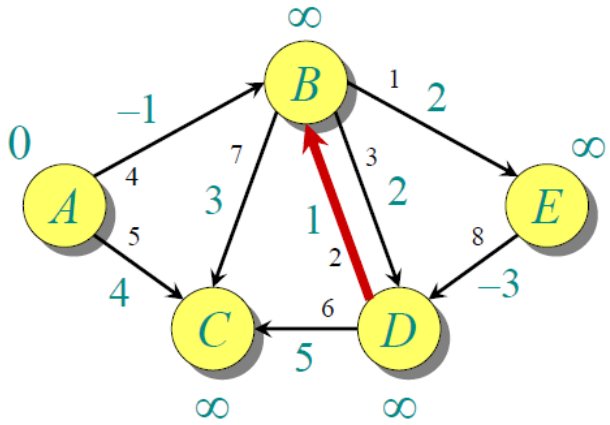


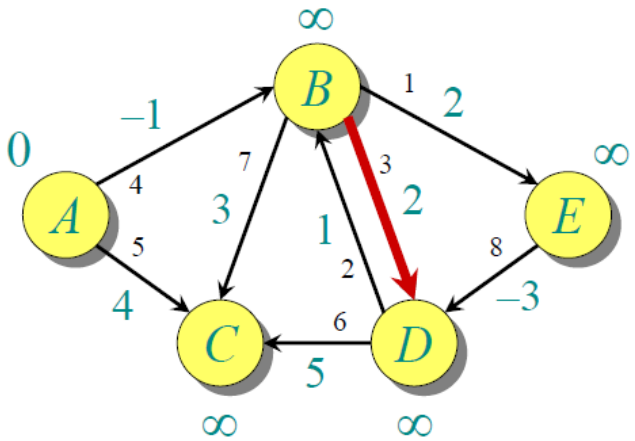
Inicial: D=(0, ∞, ∞, ∞,∞)

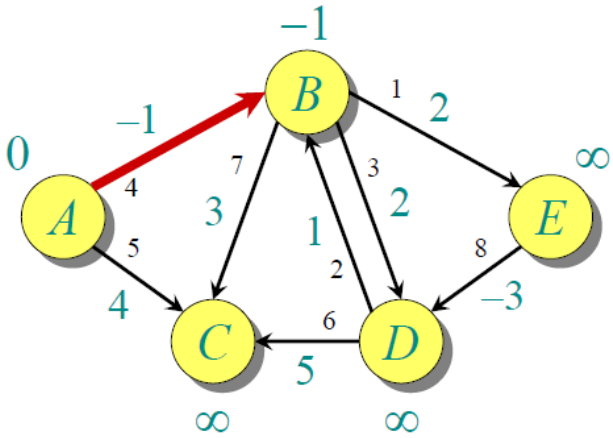
Algoritmo de Bellman-Ford

Paso 1: aristas 1, 2, 3 y 4



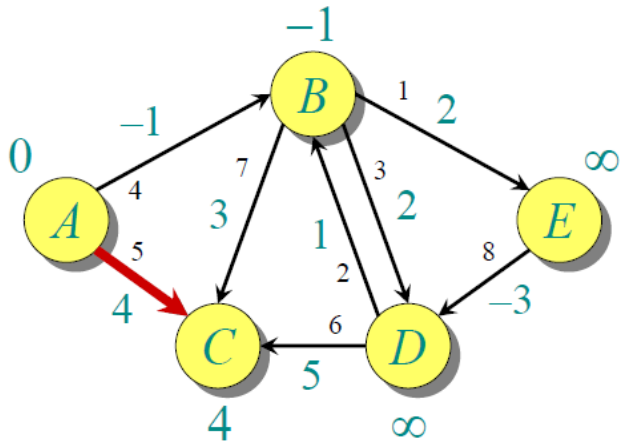


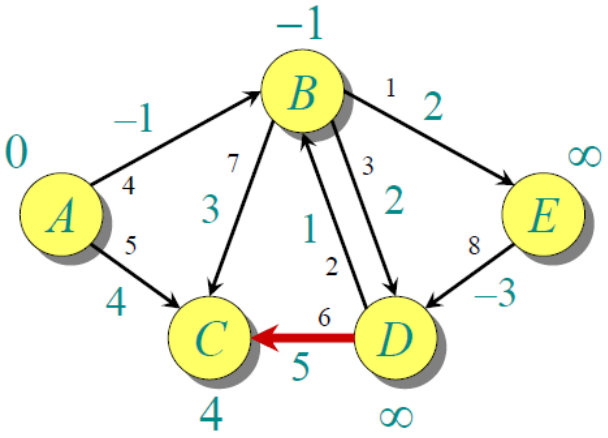


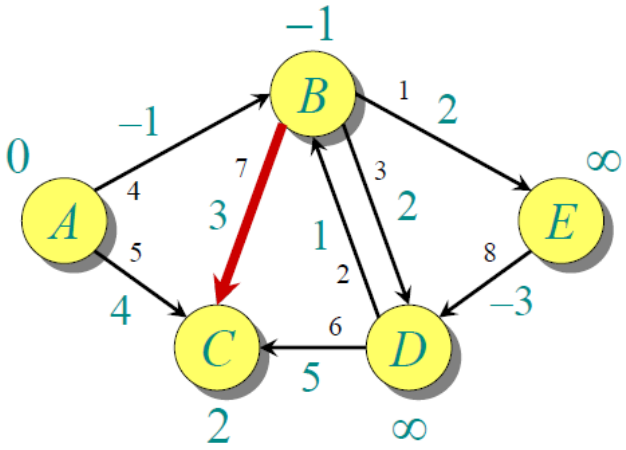


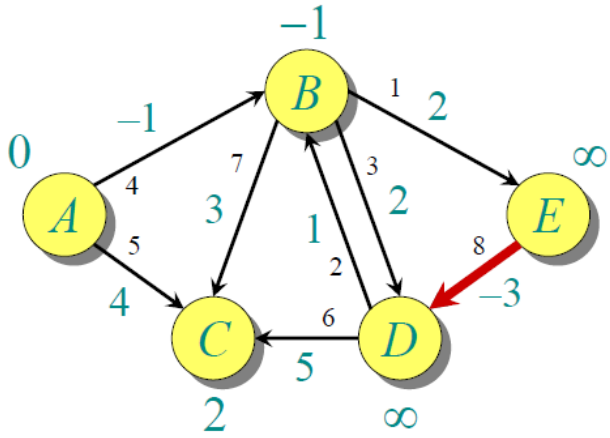
Algoritmo de Bellman-Ford

Paso 1: aristas 5, 6, 7 y 8



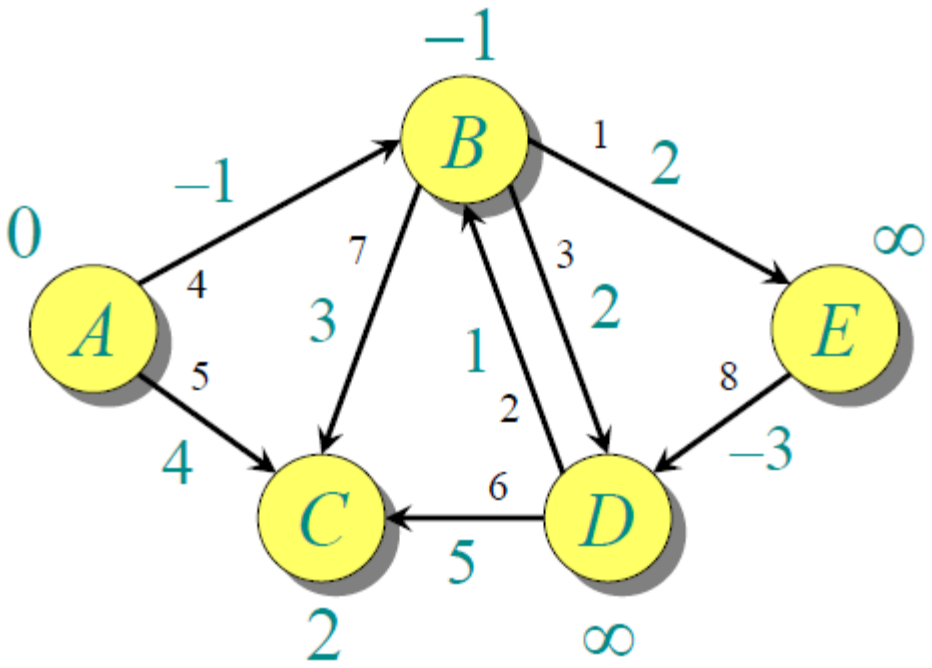






Algoritmo de Bellman-Ford

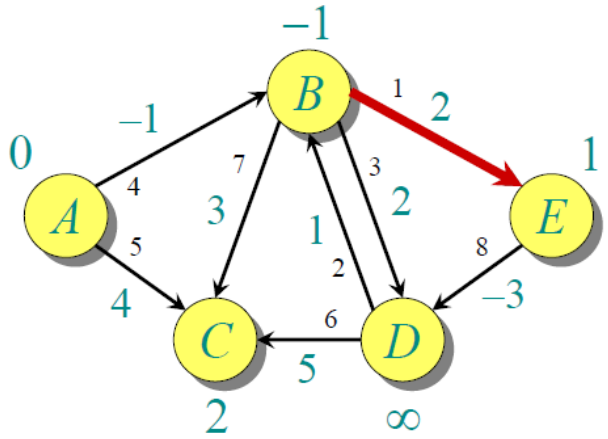
Final del paso 1:

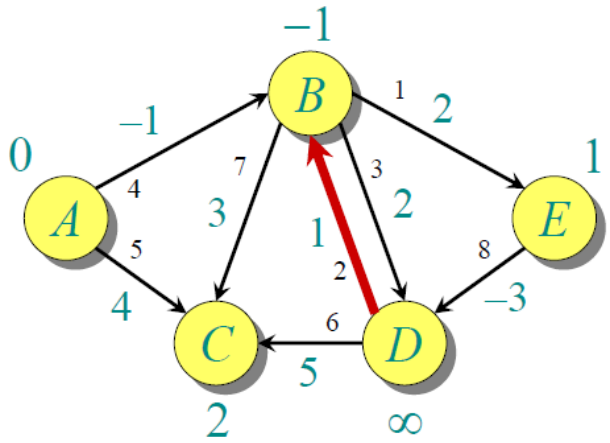


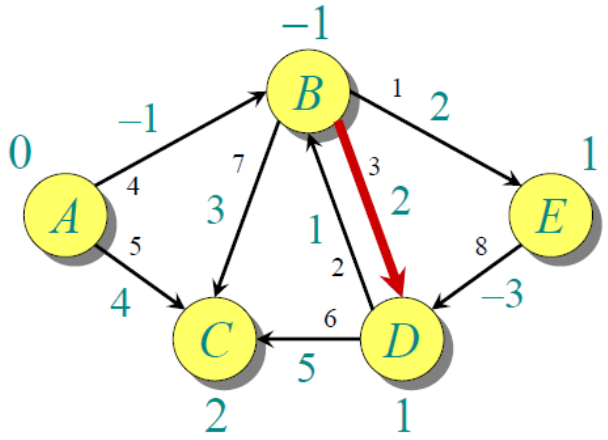
D=(0, -1, 2, ∞,∞)

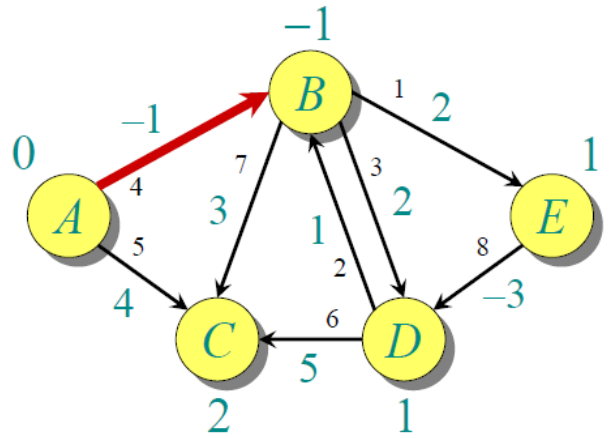
Algoritmo de Bellman-Ford

Paso 2: aristas 1, 2, 3 y 4



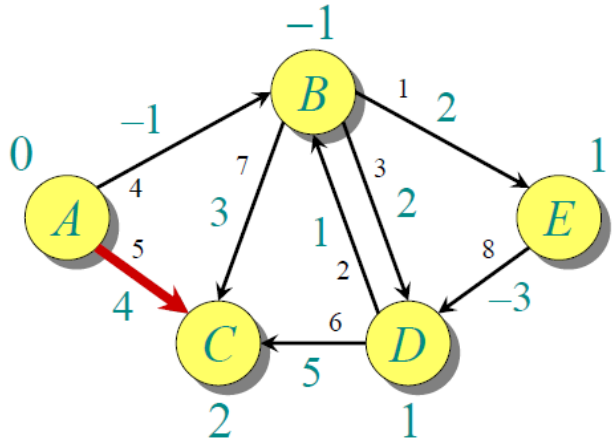


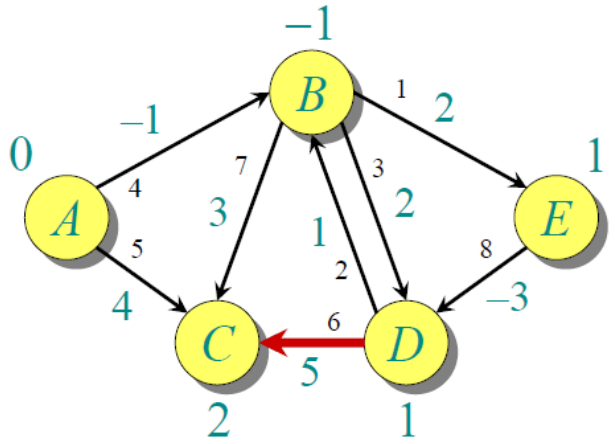


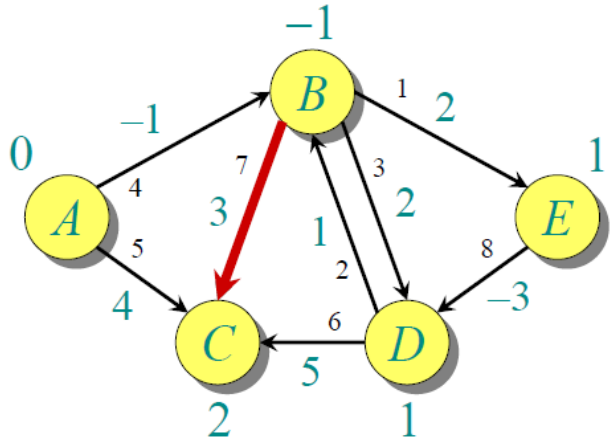


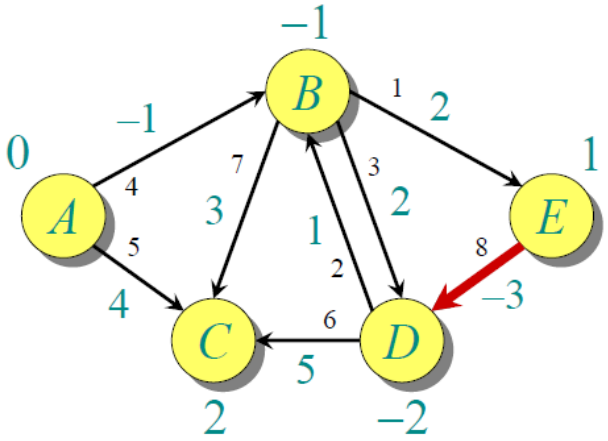
Algoritmo de Bellman-Ford

Paso 2: aristas 5, 6, 7 y 8



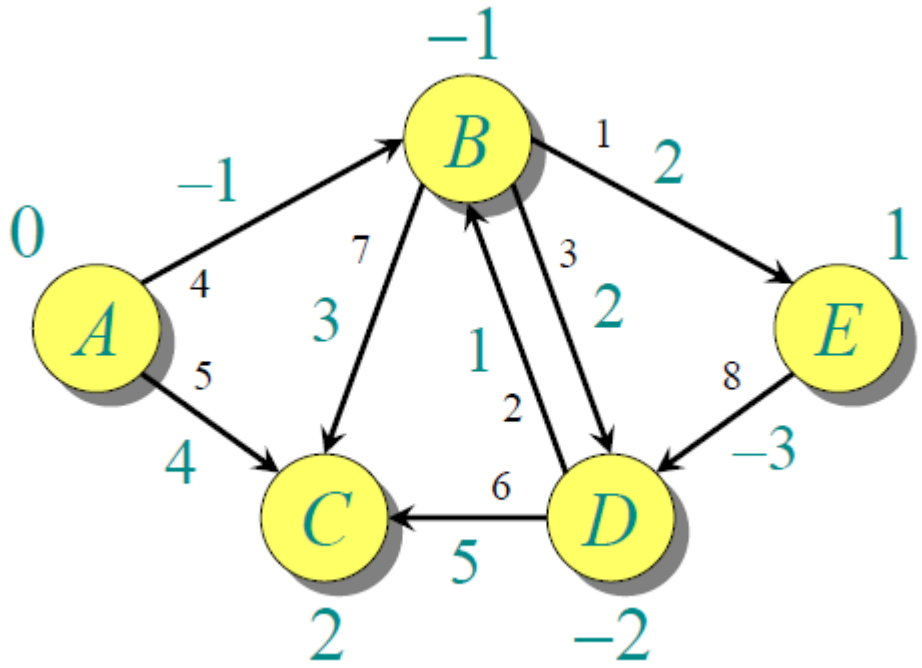






Algoritmo de Bellman-Ford

Final del paso 2:



D=(0, -1, 2, -2,1)

Igual para Paso 3 y Paso 4

Algoritmos en Grafos

Trabajo Práctico no. 10

(el último,…, al fin!!!)