ALGORITMOS VORACES

Aránzazu Jurío

ALGORITMIA

2018/2019

Índice

Índice

Qué son los algoritmos voraces

- Métodos sencillos
- Aplicable a numerosos problemas, especialmente de optimización

- Dado un problema con n entradas, el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida por el problema. A cada subconjunto que cumpla las restricciones, le llamaremos solución prometedora. A la solución prometedora que maximice o minimice la función objetivo, le llamaremos solución óptima.

Qué son los algoritmos voraces

- Elementos
  - Conjunto de candidatos
  - Función de selección: escoge el candidato no seleccionado más prometedor
  - Función para comprobar si un subconjunto de candidatos es prometedor
  - Función objetivo: es la que queremos maximizar o minimizar
  - Función para comprobar si un subconjunto de candidatos es solución (óptima o no)

Qué son los algoritmos voraces

- Tratan de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituyan la solución óptima
- Trabajan por etapas. En cada una de ellas toman la decisión que consideran mejor en ese momento, sin tener en cuenta las consecuencias futuras
- Antes de añadir un candidato, comprueban si produce una solución prometedora. En caso afirmativo, lo añaden. En caso negativo, lo descartan para siempre
- Cada vez que añade un candidato, comprueban si el conjunto obtenido es solución al problema

Qué son los algoritmos voraces

```
función voraz (ENT: conjunto; SAL: conjunto)
var
    x: elemento; solución: conjunto; encontrada: booleano
principio
    encontrada = falso
    solución = { }
    mientras entrada no vacía y no encontrada hacer
        x=seleccionarCandidato (entrada)
        si esPrometedor (x, solución) entonces
            incluir (x, solución)
            si esSolución (solución) entonces
                encontrada = verdadero
            fin si
        fin si
    fin mientras
    devolver solución
fin
```

Qué son los algoritmos voraces

- Son muy fáciles de implementar y producen soluciones muy eficientes
- ¿Por qué no utilizarlos siempre?
  - No todos los problemas se pueden expresar en forma de algoritmo voraz
  - Hay problemas en que las mejores soluciones locales no forman parte de la mejor solución global

Índice

El problema del cambio

- Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1, v_2, \ldots, v_n$ el problema consiste en descomponer cualquier cantidad dada M en monedas de ese país utilizando el menor número posible de monedas

El problema del cambio

- Moneda: EUROS
- Cantidad 8,43€

El problema del cambio

- Elementos
  - Conjunto de candidatos: infinitas monedas de cada uno de los valores disponibles
  - Función para comprobar si un subconjunto de candidatos es prometedor: comprobar que la suma actual sea menor o igual que la cantidad a obtener
  - Función objetivo (es la que queremos maximizar o minimizar): número de monedas
  - Función para comprobar si es un subconjunto de candidatos es solución (óptima o no): la suma de las monedas es la cantidad buscada
  - Función de selección (escoge el candidato no seleccionado más prometedor):

El problema del cambio

- Elementos
  - Conjunto de candidatos: infinitas monedas de cada uno de los valores disponibles
  - Función para comprobar si un subconjunto de candidatos es prometedor: comprobar que la suma actual sea menor o igual que la cantidad a obtener
  - Función objetivo (es la que queremos maximizar o minimizar): número de monedas
  - Función para comprobar si es un subconjunto de candidatos es solución (óptima o no): la suma de las monedas es la cantidad buscada
  - Función de selección (escoge el candidato no seleccionado más prometedor): escoger la moneda de mayor valor

El problema del cambio

- Ejemplo
  - Monedas: 100, 25, 10, 5, 1
  - Cantidad: 386

El problema del cambio

- Ejemplo
  - Monedas: 100, 25, 10, 5, 1
  - Cantidad: 386

  - 100 + 100 + 100 + 25 + 25 + 25 + 10 + 1

El problema del cambio

- Si las monedas nos llegan ordenadas, complejidad lineal con respecto al número de monedas
- Muy eficiente

El problema del cambio

- ¿Este problema tiene solución para cualquier conjunto de monedas y cualquier cantidad a sumar?

El problema del cambio

- ¿Este problema tiene solución para cualquier conjunto de monedas y cualquier cantidad a sumar?
  - Monedas: 2, 4, 8, 16, 25
  - Cantidad: 21

- Necesitamos una moneda unidad

El problema del cambio

- Sabiendo que tenemos una moneda unidad, ¿la solución encontrada por el algoritmo voraz es siempre óptima?

El problema del cambio

- Sabiendo que tenemos una moneda unidad, ¿la solución encontrada por el algoritmo voraz es siempre óptima?
  - NO
- ¿Qué propiedad puedo exigir a las monedas, para conseguir siempre la solución óptima con el algoritmo voraz?

El problema del cambio

- Sabiendo que tenemos una moneda unidad, ¿la solución encontrada por el algoritmo voraz es siempre óptima?
  - NO
- ¿Qué propiedad puedo exigir a las monedas, para conseguir siempre la solución óptima con el algoritmo voraz?
  - todo número natural k se puede descomponer como

$$k = a + bp + cp^2 + \cdots + zp^n$$

  - por tanto, si las monedas son de la forma $1, p, p^2, \ldots, p^n$ la solución obtenida es óptima

El problema del cambio

- Demostración
  - Por simplicidad, suponemos que las monedas son del estilo $\{1, 2, 2^2, 2^3, 2^4, \dots\}$
  - Solución obtenida por el algoritmo voraz
  - $x = r_0 + r_1 p + r_2 p^2 + \dots + r_n p^n$
  - Solución distinta
  - $x = s_0 + s_1 p + s_2 p^2 + \dots + s_m p^m$
  - Por nuestro algoritmo voraz, sabemos que $x < 2^{n+1}$, luego $m \leq n$
  - Si x es par, entonces $r_0 = 0$. Como $s_0 \geq 0$ y $r_0 \neq s_0$, entonces $r_0 < s_0$
  - Si x es impar, entonces $r_0 = 1$. La solución 2 también debe tener al menos una moneda unidad, luego $s_0 \geq 1$. Como son distintos $r_0 < s_0$
  - Por tanto, $s_0 - r_0 > 0$ y par. Al menos dos monedas pueden pasar a ser una moneda de cantidad mayor

Índice

El fontanero diligente

- Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes

El fontanero diligente

- Cada tarea i tarda un tiempo t_i en ser ejecutada
- El tiempo que el fontanero va a tardar en ejecutar todas las tareas es fijo, independientemente del orden en que las haga

$$\sum_{i=1}^{n} t_i$$

- ¿Qué función es la que quiero minimizar?

El fontanero diligente

- Cada tarea i tarda un tiempo t_i en ser ejecutada
- El tiempo que el fontanero va a tardar en ejecutar todas las tareas es fijo, independientemente del orden en que las haga

$$\sum_{i=1}^{n} t_i$$

- ¿Qué función es la que quiero minimizar?
  - El tiempo que tarda cada cliente en ver reparada su avería
    - $E_1 = t_1$
    - $E_2 = t_1 + t_2$
    - $\cdots$
    - $E_n = t_1 + t_2 + \cdots + t_n$

$$\sum_{i=1}^{n} E_i$$

El fontanero diligente

- Ejemplo
  - Tarea 1: $t_1 = 15$
  - Tarea 2: $t_2 = 8$
  - Tarea 3: $t_3 = 10$

Orden	Tiempo
Tarea 1, tarea 2, tarea 3	15 + 23 + 33 = 71
Tarea 1, tarea 3, tarea 2	15 + 25 + 33 = 73
Tarea 2, tarea 1, tarea 3	8 + 23 + 33 = 64
Tarea 2, tarea 3, tarea 1	8 + 18 + 33 = 59
Tarea 3, tarea 1, tarea 2	10 + 25 + 33 = 68
Tarea 3, tarea 2, tarea 1	10 + 18 + 33 = 61

El fontanero diligente

- Elementos
  - Conjunto de candidatos: los diferentes clientes
  - Función para comprobar si un subconjunto de candidatos es prometedor: comprobar que no se repiten clientes
  - Función objetivo (es la que queremos maximizar o minimizar): tiempo de espera de los clientes
  - Función para comprobar si es un subconjunto de candidatos es solución (óptima o no): todos los clientes son parte de la solución
  - Función de selección (escoge el candidato no seleccionado más prometedor): ¿?

El fontanero diligente

- ¿Qué función es la que quiero minimizar?
  - El tiempo que tarda cada cliente en ver reparada su avería
    - $E_1 = t_1$
    - $E_2 = t_1 + t_2$
    - $\cdots$
    - $E_n = t_1 + t_2 + \cdots + t_n$

$$\sum_{i=1}^{n} E_i$$

$$\sum_{i=1}^{n} E_i = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + \cdots + (t_1 + t_2 + \cdots + t_n) =$$
$$= nt_1 + (n-1)t_2 + (n-2)t_3 + \cdots + t_n$$

El fontanero diligente

- ¿Qué función es la que quiero minimizar?
  - El tiempo que tarda cada cliente en ver reparada su avería
    - $E_1 = t_1$
    - $E_2 = t_1 + t_2$
    - $\cdots$
    - $E_n = t_1 + t_2 + \cdots + t_n$

$$\sum_{i=1}^{n} E_i$$

    - $\sum_{i=1}^{n} E_i = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + \cdots + (t_1 + t_2 + \cdots + t_n) =$
    - $= nt_1 + (n-1)t_2 + (n-2)t_3 + \cdots + t_n$

    - Minimizamos haciendo que $t_i \leq t_j \; \forall i < j$

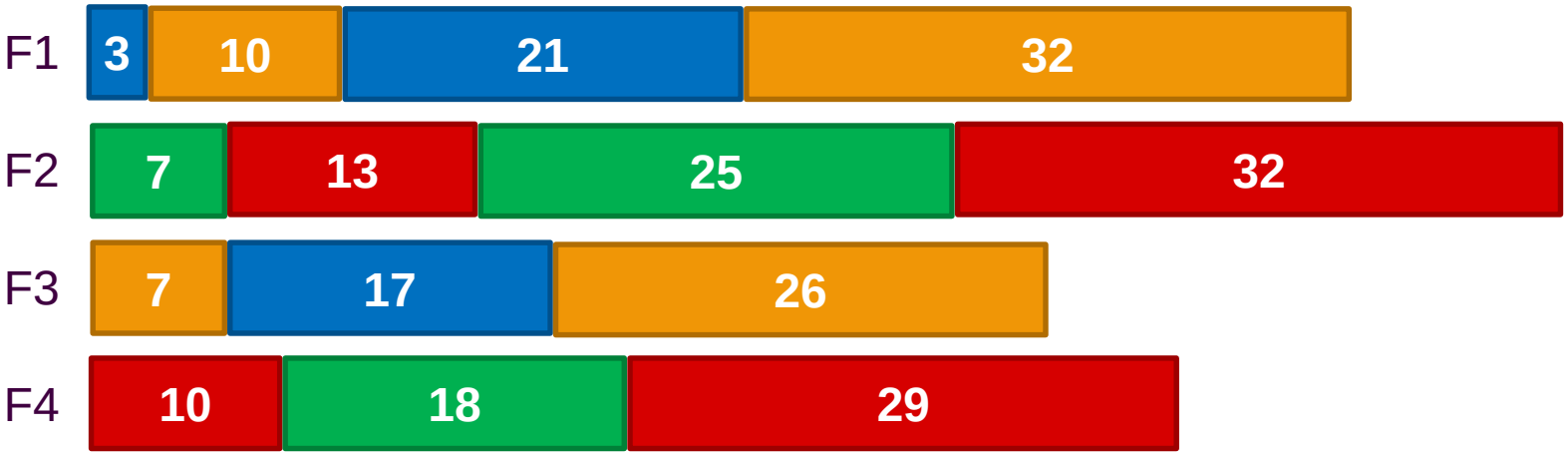
Índice

Más fontaneros

- En este caso, la empresa ha decidido contratar a más personal, y contamos con F fontaneros para realizar n tareas
- ¿Qué criterio se debe seguir para asignar cada tarea a uno de los fontaneros? ¿En qué orden debe realizar cada fontanero sus tareas?
- Ejemplo
  - 4 fontaneros
  - 14 tareas con tiempos:
    - 3, 7, 7, 10, 10, 13, 17, 18, 21, 25, 26, 29, 32, 32

Más fontaneros

- 4 fontaneros
- 14 tareas con tiempos:
  - 3, 7, 7, 10, 10, 13, 17, 18, 21, 25, 26, 29, 32, 32



Índice

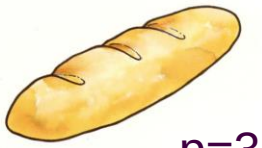
El problema de la mochila

- Dados n elementos $e_1, e_2, \ldots, e_n$ con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$, y dada una mochila con capacidad de albergar hasta un máximo de peso M, queremos encontrar las porciones de los n elementos $x_1, x_2, \ldots, x_n$ $(0 \leq x_i \leq 1)$ que tenemos que introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima
- Esto es, tenemos que encontrar los valores $(x_1, x_2, \ldots, x_n)$ de forma que se maximice la cantidad $\sum_{i=1}^{n} b_i x_i$, sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \leq M$

El problema de la mochila

M=8

p=2
b=4



p=3
b=5

p=5
b=11

p=1
b=3

p=2
b=3

- Es obvio que tengo que llenar la mochila entera para obtener el máximo beneficio
  - Si tengo una combinación que deja parte del peso de la mochila sin llenar, siempre puedo añadir una fracción de un producto, que va a aumentar el beneficio

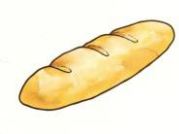
El problema de la mochila

- Elementos
  - Conjunto de candidatos: los diferentes objetos
  - Función para comprobar si un subconjunto de candidatos es prometedor: el peso de los objetos es menor o igual que la capacidad de la mochila
  - Función objetivo (es la que queremos maximizar o minimizar): maximizar el beneficio de los objetos introducidos
  - Función para comprobar si es un subconjunto de candidatos es solución (óptima o no): el peso de los objetos igual a la capacidad de la mochila
  - Función de selección (escoge el candidato no seleccionado más prometedor): ¿?

El problema de la mochila

M=8

p=2
b=4



p=3
b=5

p=5
b=11

p=1
b=3

p=2
b=3

- La misma proporción de cada objeto
  - 8/13 harina + 8/13 pan + 8/13 chocolate + 8/13 pipas + 8/13 refresco → beneficio = 16
- Objetos que menos pesen
  - Pipas + harina + refresco + pan → beneficio = 15
- Objetos que den más beneficio
  - Chocolate + pan → beneficio = 15

El problema de la mochila

M=8

p=2
b=4
b/p=2

p=3
b=5
b/p=1,67

p=5
b=11
b/p=2,2

p=1
b=3
b/p=3

p=2
b=3
b/p=1,5

- Objetos que den más beneficio por unidad de peso
  - Pipas + chocolate + harina → beneficio = 18

El problema de la mochila

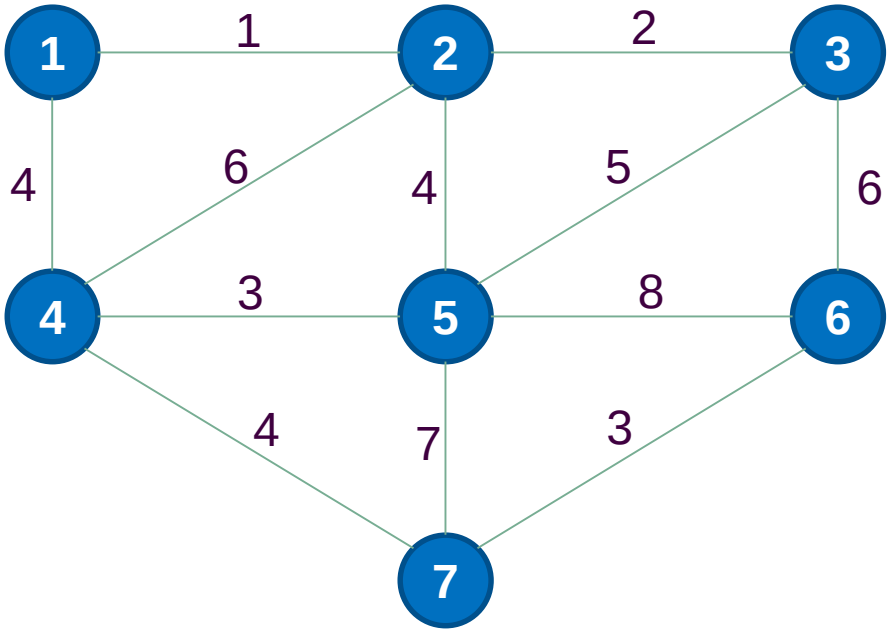
- ¿Es esta solución siempre óptima?

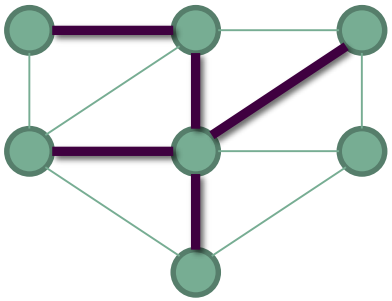
Índice

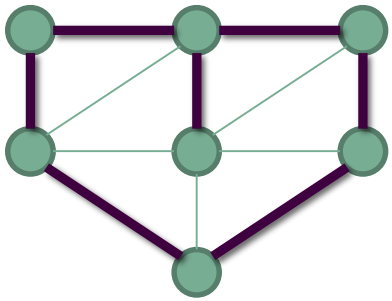
Árboles recubridores de coste mínimo

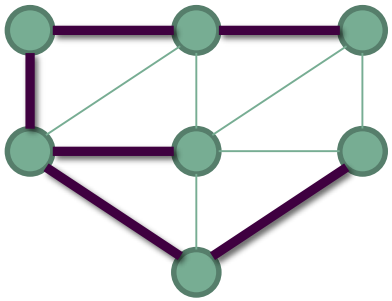
- Sea G = (N, A) un grafo ponderado conexo no dirigido donde N es el conjunto de vértices y A el conjunto de arcos. Nuestro objetivo es encontrar un subconjunto T de arcos que conecte todos los vértices del grafo y tal que la suma de costes sea mínima
- Al grafo (N, T) lo llamaremos árbol recubridor de coste mínimo de G

Árboles recubridores de coste mínimo









Índice

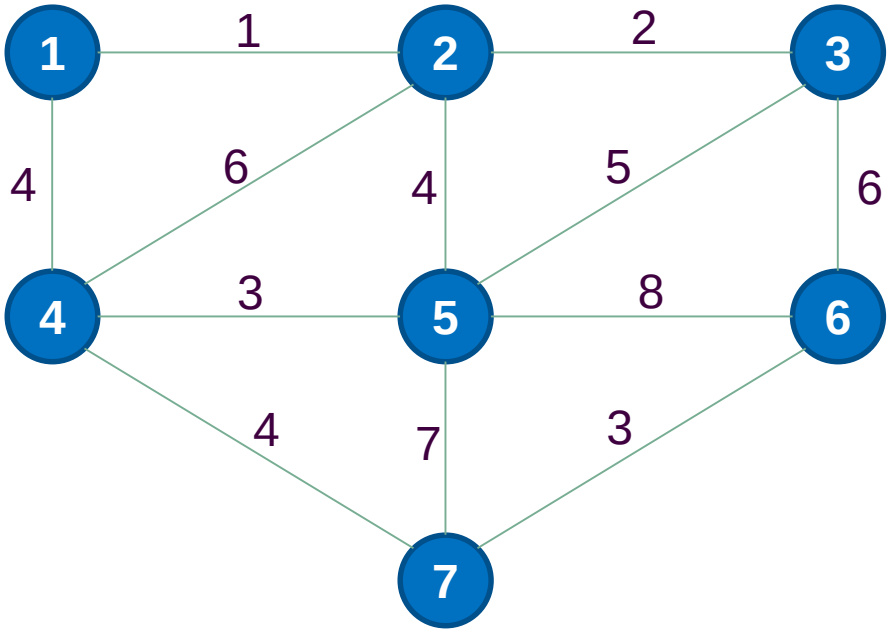
Algoritmo de Kruskal

- Inicialmente el conjunto de arcos T es el conjunto vacío
- En todo instante, el grafo parcial formado por los vértices de G y los arcos de T contiene varias componentes conexas (inicialmente cada vértice de G es una componente conexa)
- Los arcos de cada componente conexa de T forman un árbol recubridor de coste mínimo para los vértices de dicha componente conexa
- Al terminar, en T existirá una única componente conexa que enlazará todos los vértices de G formando un árbol recubridor de coste mínimo para G

Algoritmo de Kruskal

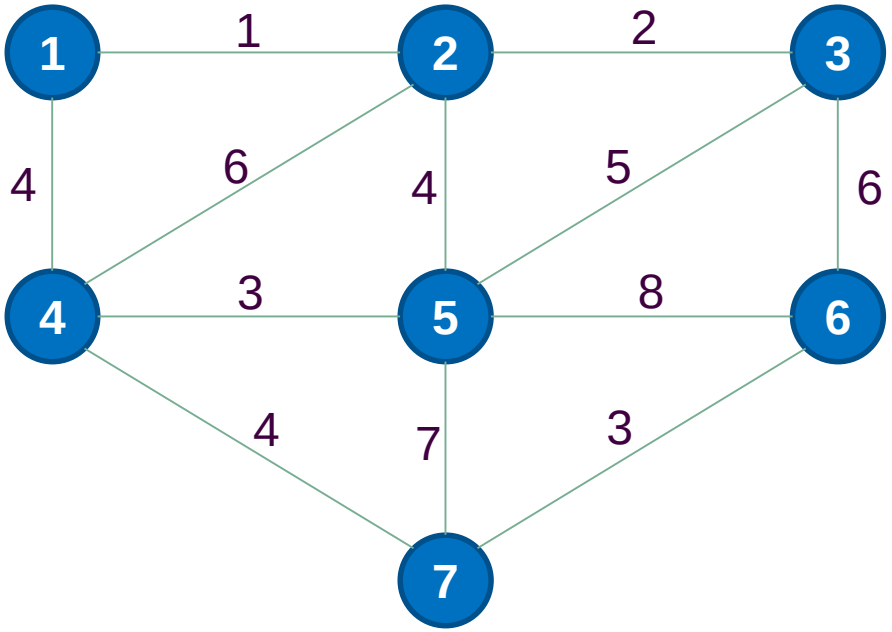
- Para construir componentes conexas cada vez mayores, se consideran los arcos de G en orden creciente de coste
  - Si un arco une dos componentes conexas de T, lo incorporamos a T y las dos componentes conexas quedan fusionadas en una sola
  - En caso contrario, el arco es rechazado porque uniría dos vértices de una misma componente conexa y se formaría un ciclo
- Cuando sólo queda una única componente conexa que une todos los vértices de G el algoritmo termina

Algoritmo de Kruskal



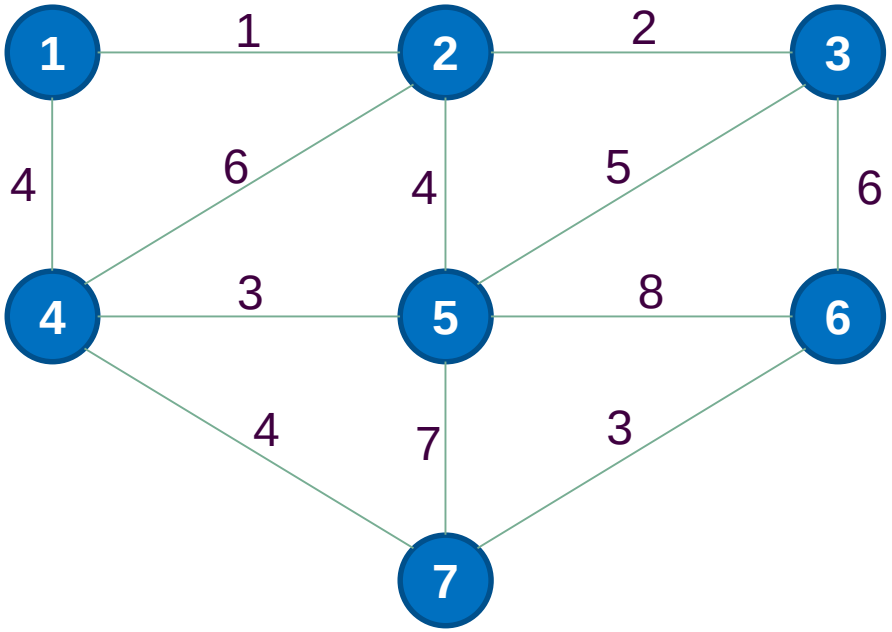
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - 4 – 5
  - 6 – 7
  - 1 – 4
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



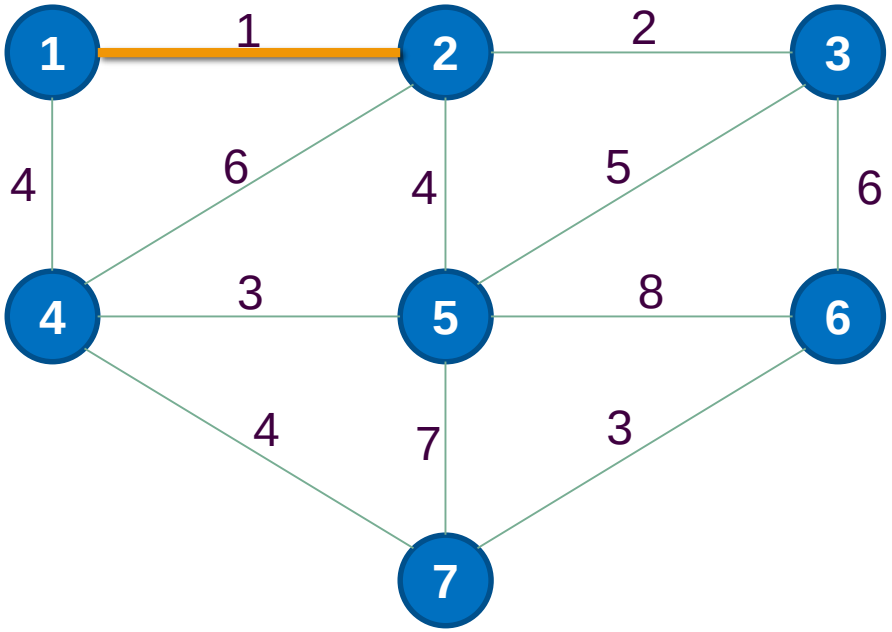
- {1} {2} {3} {4} {5} {6} {7}

Algoritmo de Kruskal



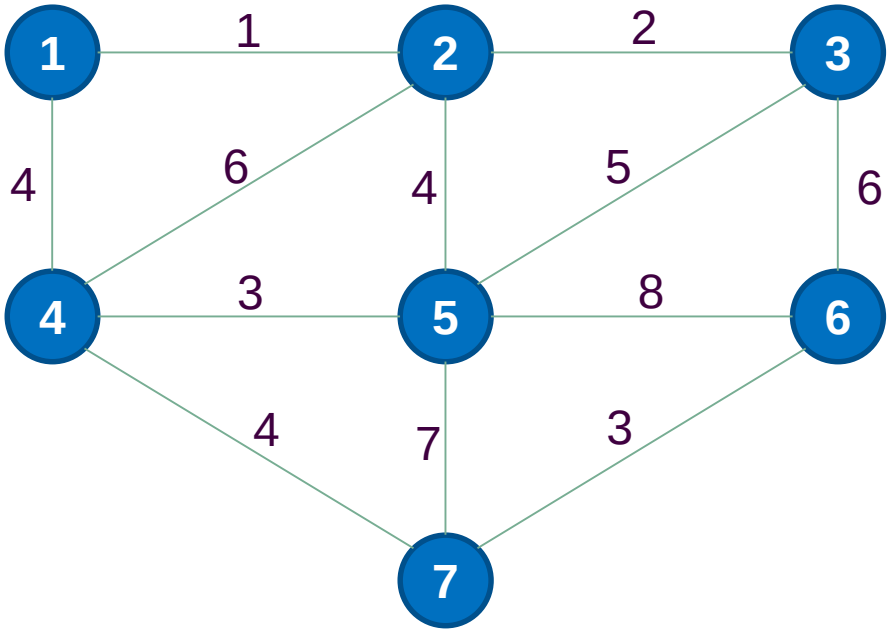
- Ordenamos todas las aristas en orden creciente
  - **1 – 2**
  - 2 – 3
  - 4 – 5
  - 6 – 7
  - 1 – 4
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



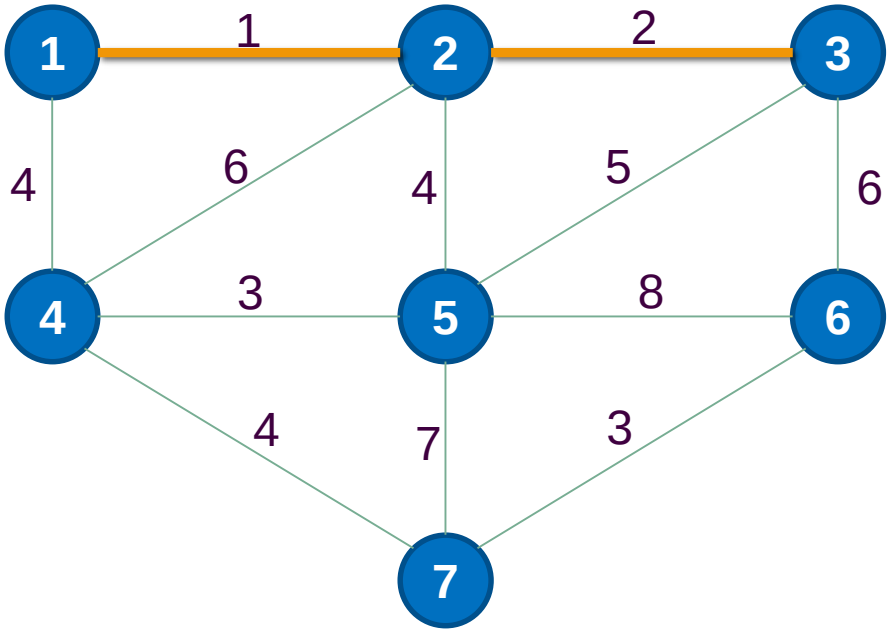
- {1} {2} {3} {4} {5} {6} {7}
- {1, 2} {3} {4} {5} {6} {7}

Algoritmo de Kruskal



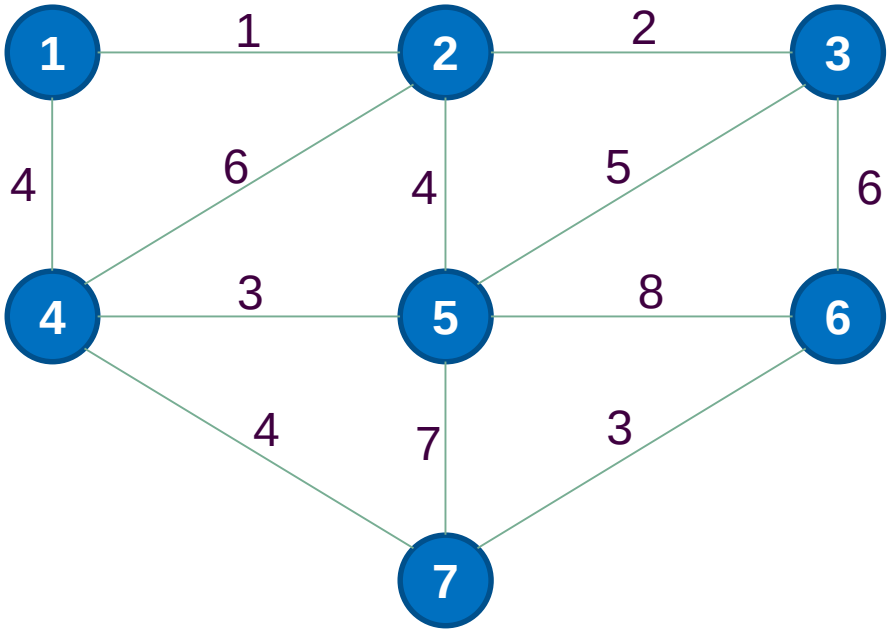
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - **2 – 3**
  - 4 – 5
  - 6 – 7
  - 1 – 4
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



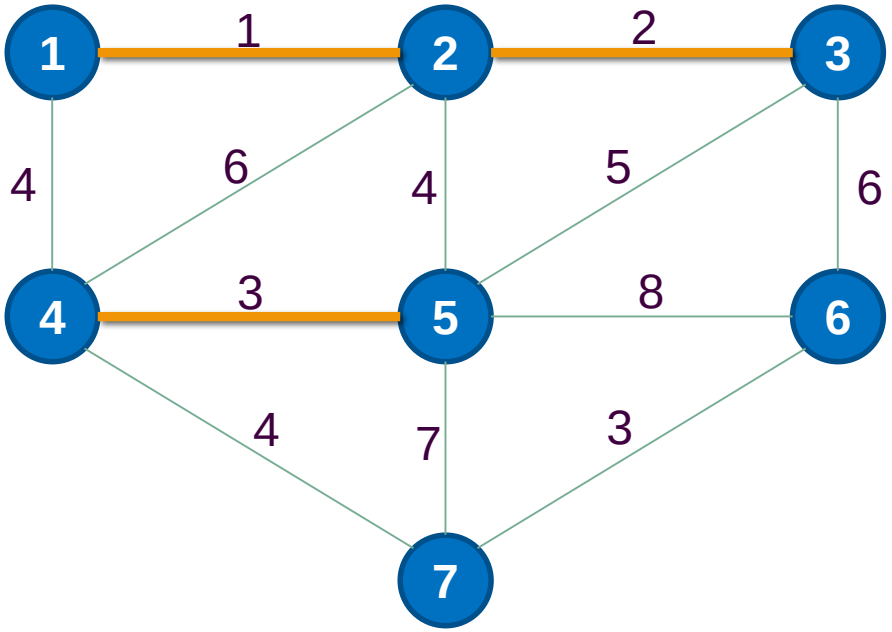
- {1, 2} {3} {4} {5} {6} {7}
- {1, 2, 3} {4} {5} {6} {7}

Algoritmo de Kruskal



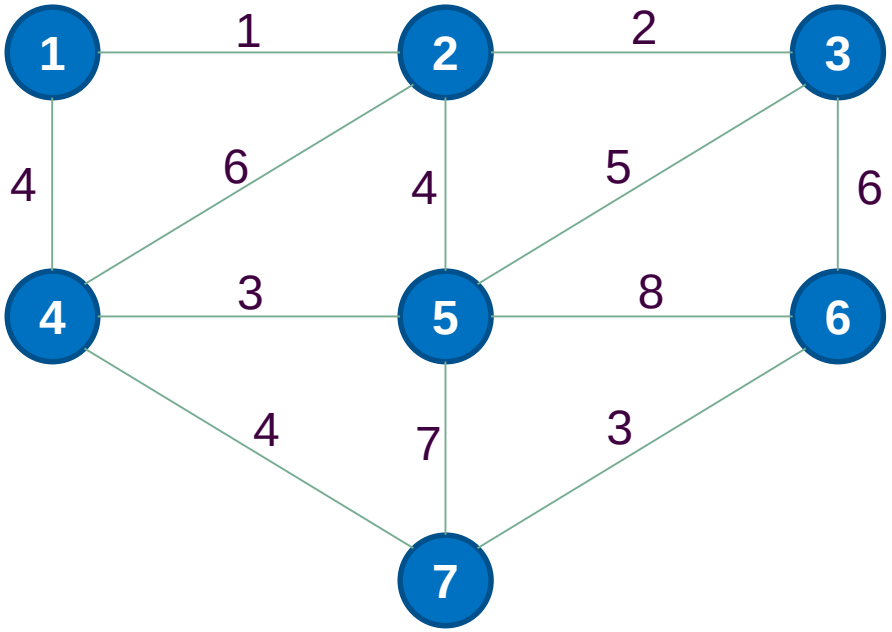
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - **4 – 5**
  - 6 – 7
  - 1 – 4
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



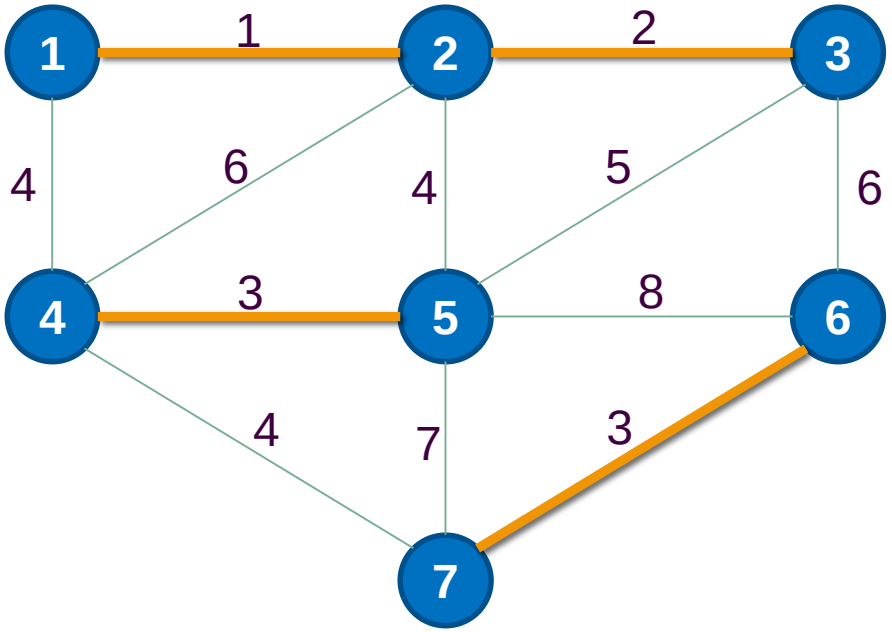
- {1, 2, 3} {4} {5} {6} {7}
- {1, 2, 3} {4, 5} {6} {7}

Algoritmo de Kruskal



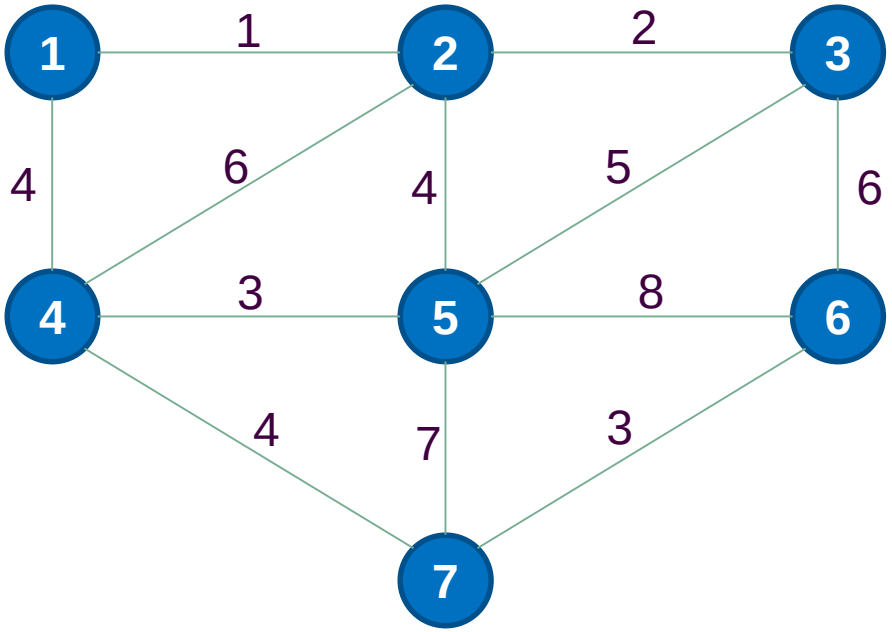
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - 4 – 5
  - **6 – 7**
  - 1 – 4
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



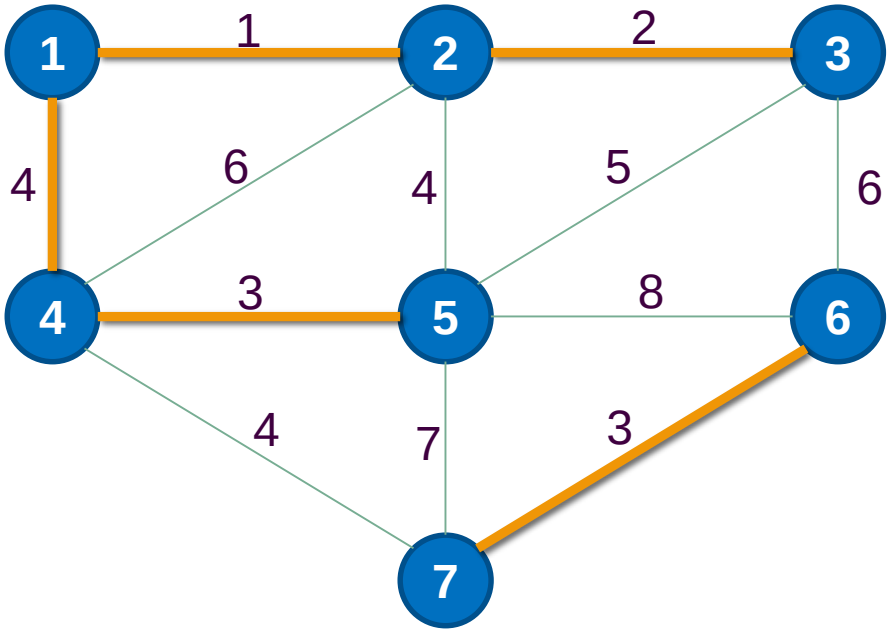
- {1, 2, 3} {4, 5} {6} {7}
- {1, 2, 3} {4, 5} {6, 7}

Algoritmo de Kruskal



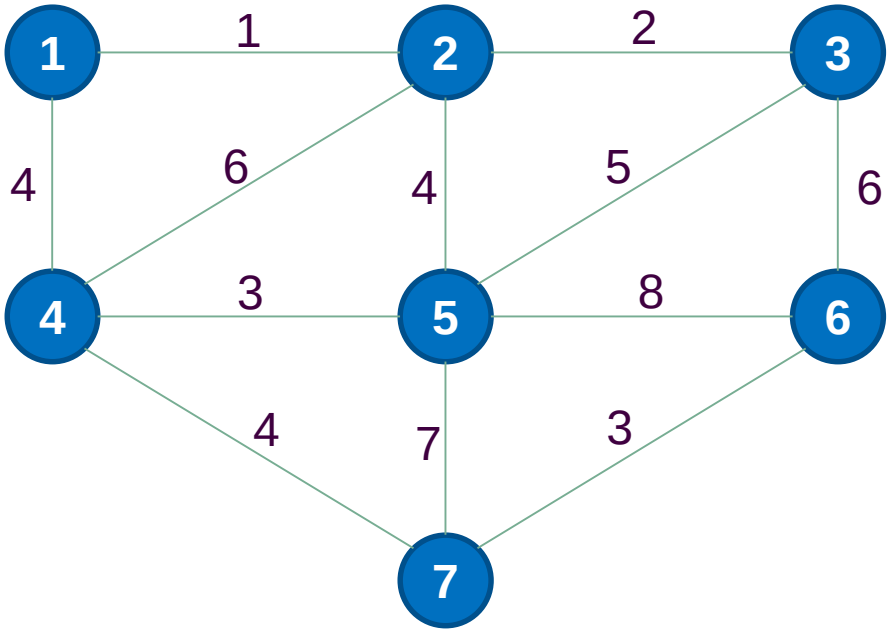
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - 4 – 5
  - 6 – 7
  - **1 – 4**
  - 2 – 5
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



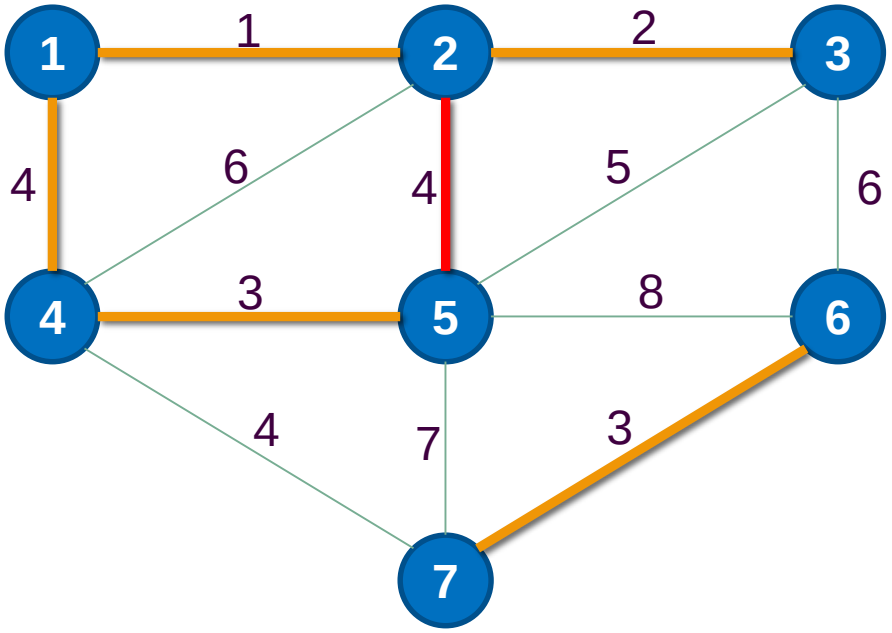
- {1, 2, 3} {4, 5} {6, 7}
- {1, 2, 3, 4, 5} {6, 7}

Algoritmo de Kruskal



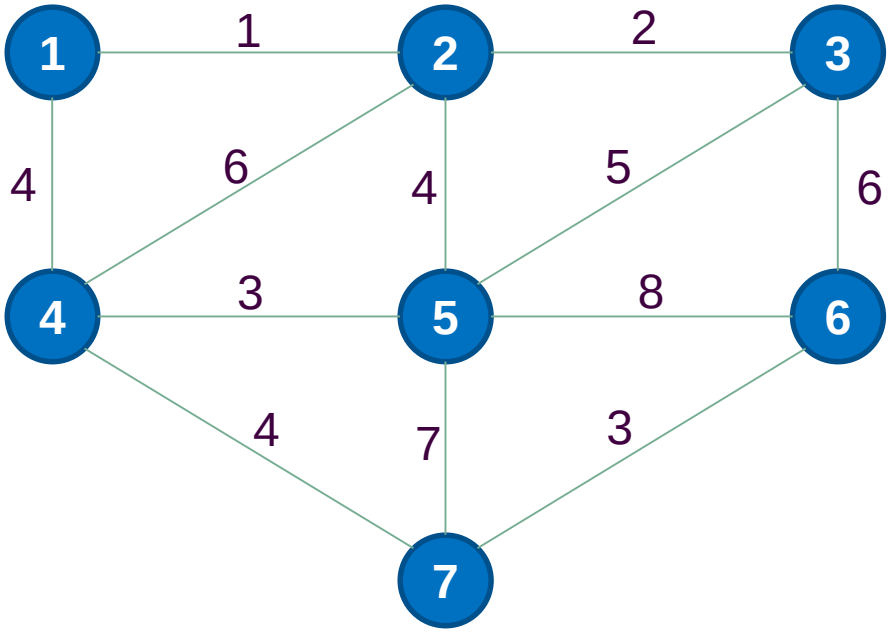
- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - 4 – 5
  - 6 – 7
  - 1 – 4
  - **2 – 5**
  - 4 – 7
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



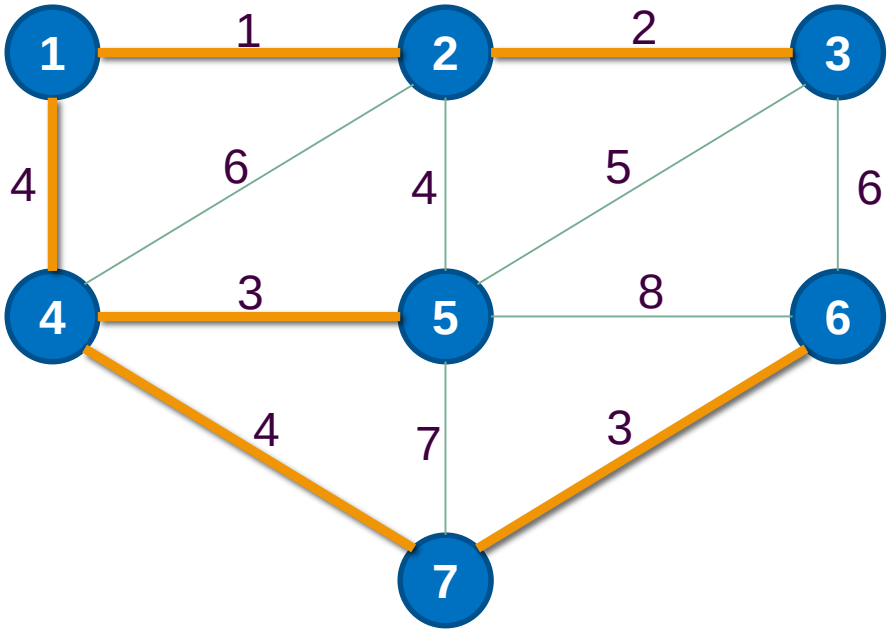
- {1, 2, 3, 4, 5} {6, 7}

Algoritmo de Kruskal



- Ordenamos todas las aristas en orden creciente
  - 1 – 2
  - 2 – 3
  - 4 – 5
  - 6 – 7
  - 1 – 4
  - 2 – 5
  - **4 – 7**
  - 3 – 5
  - 2 – 4
  - 3 – 6
  - 5 – 7
  - 5 – 6

Algoritmo de Kruskal



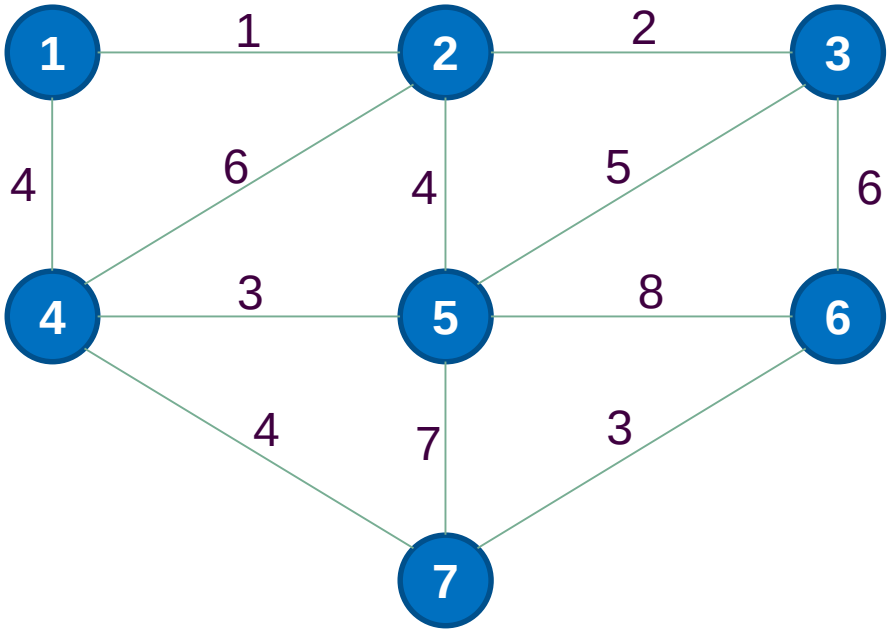
- {1, 2, 3, 4, 5} {6, 7}
- {1, 2, 3, 4, 5, 6, 7}

Índice

Algoritmo de Prim

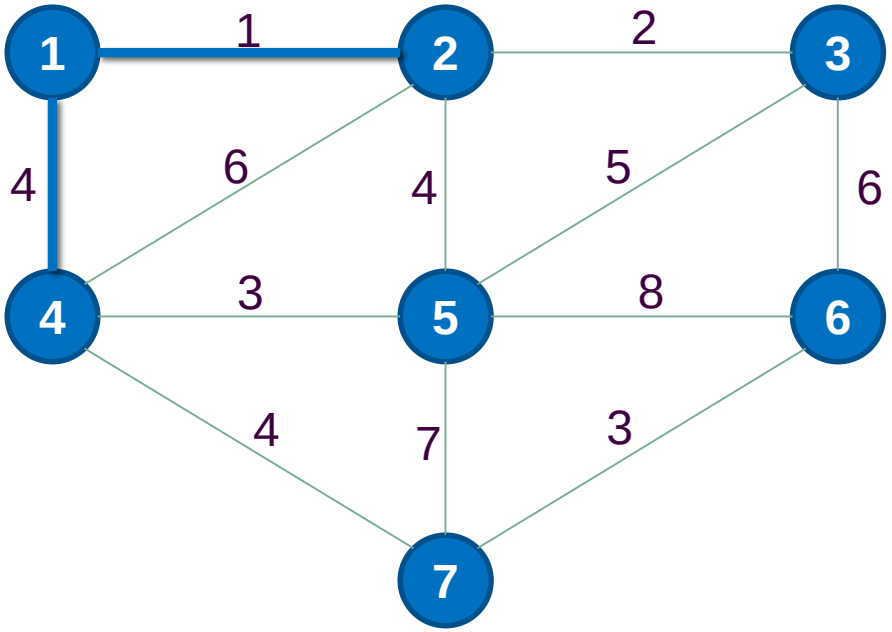
- El árbol recubridor de coste mínimo crece de forma natural a partir de una raíz arbitraria de forma que en cada iteración se añade una rama más al árbol en curso y el algoritmo termina cuando se alcanzan todos los vértices
- Inicializaremos el conjunto de vértices B con uno cualquiera, y el conjunto de arcos T con el conjunto vacío. En cada paso el algoritmo busca un arco {u, v} de coste mínimo tal que u ∈ N \B y v ∈ B. Así, los arcos de T forman en todo momento un árbol de expansión mínimo para todos los vértices de B. Continuaremos el proceso hasta que B sea igual a N.

Algoritmo de Prim



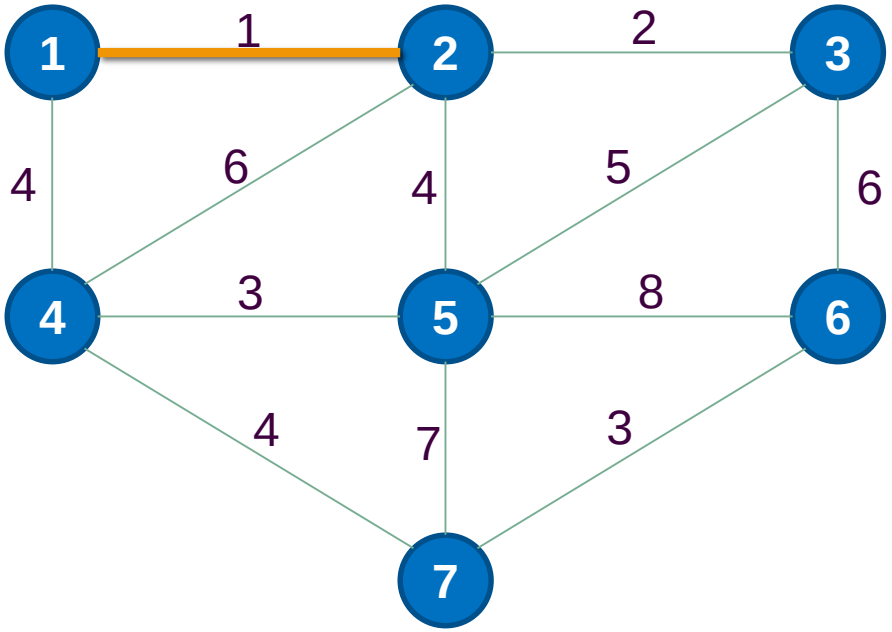
- Empezamos por un vértice aleatorio
  - Seleccionamos el 1

Algoritmo de Prim



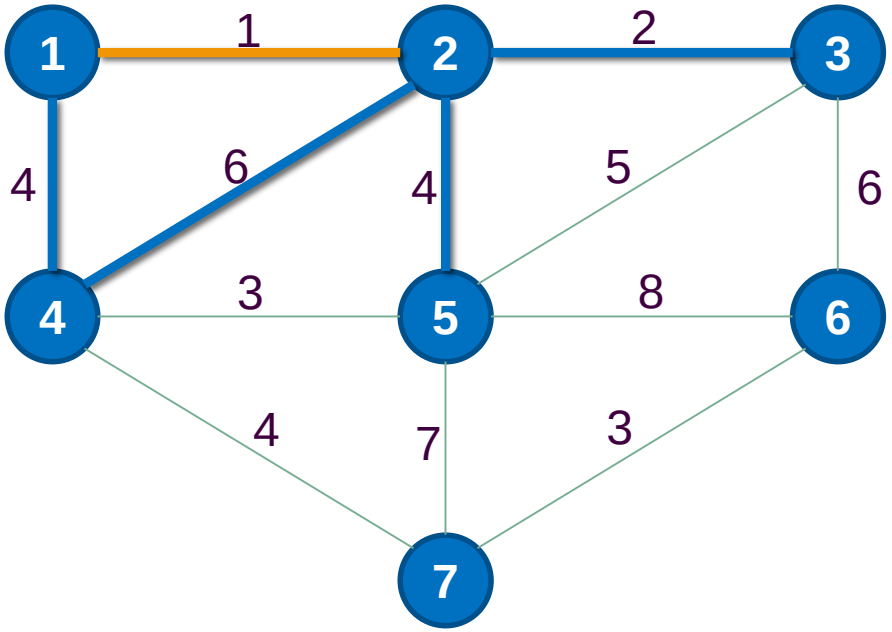
- {1}

Algoritmo de Prim



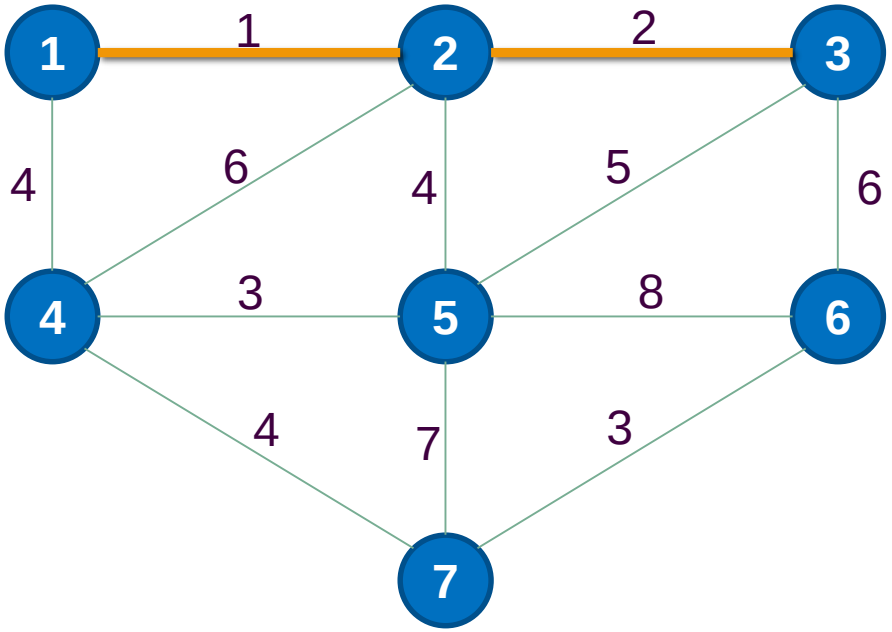
- {1} → (1, 2)
- {1, 2}

Algoritmo de Prim



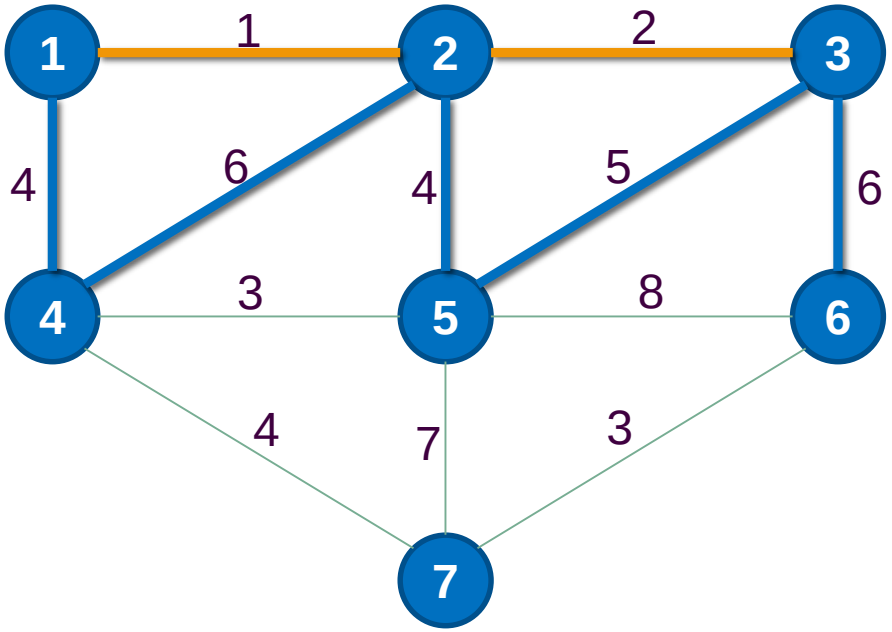
- {1, 2}

Algoritmo de Prim



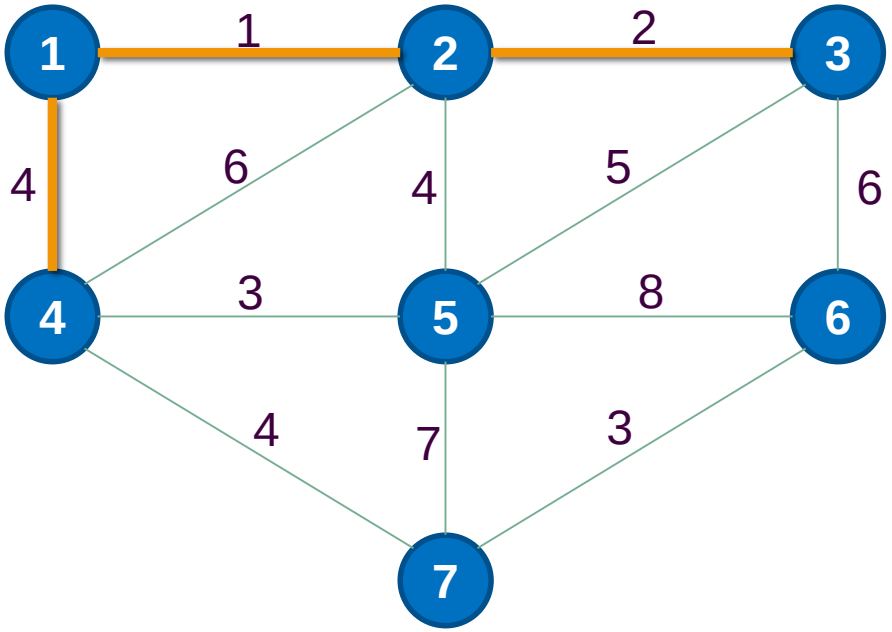
- {1, 2} → (2, 3)
- {1, 2, 3}

Algoritmo de Prim



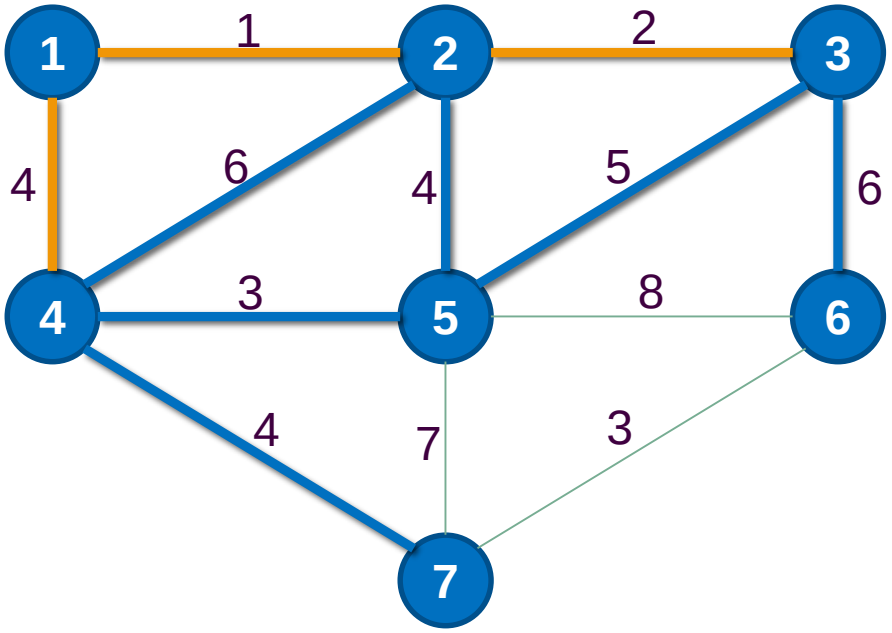
- {1, 2, 3}

Algoritmo de Prim



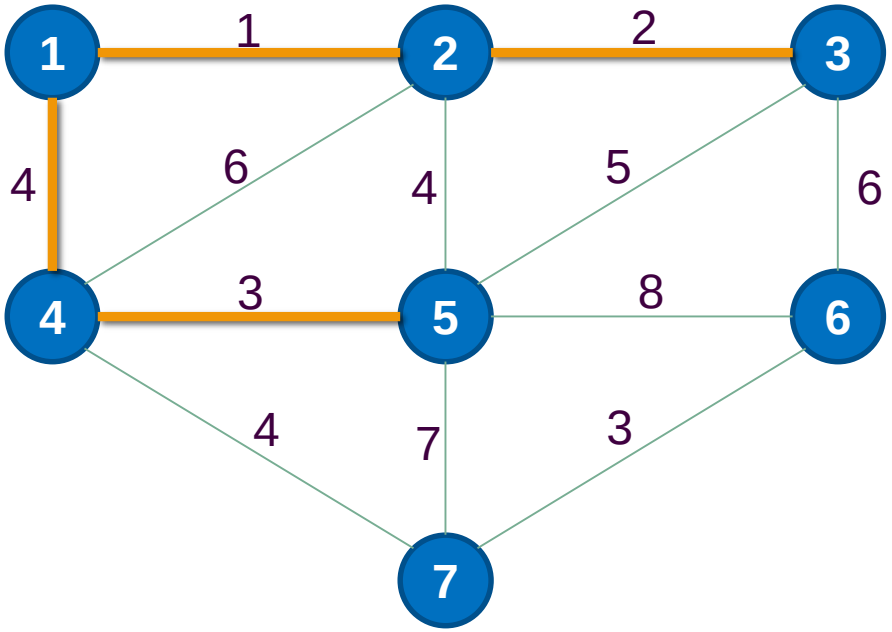
- {1, 2, 3} → (1, 4)
- {1, 2, 3, 4}

Algoritmo de Prim



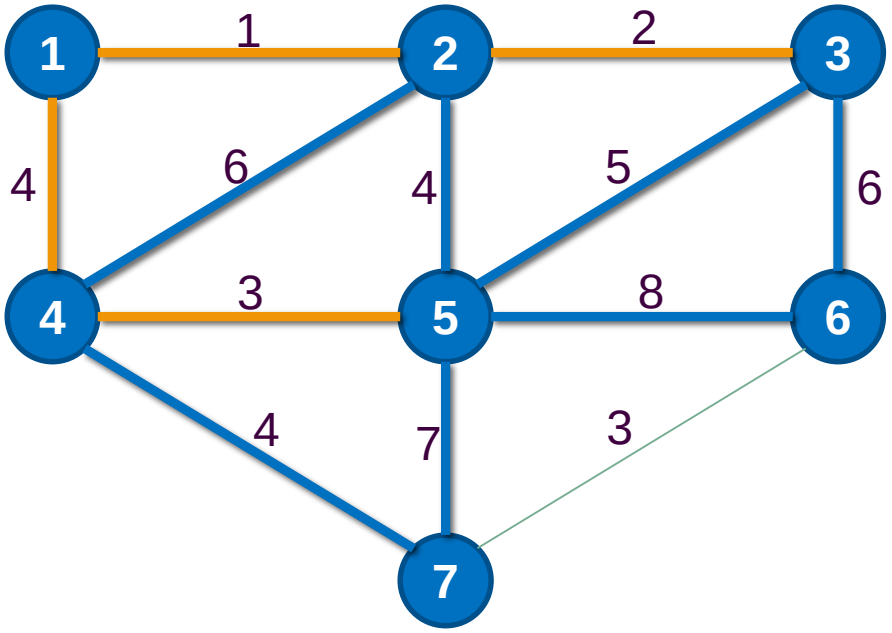
- {1, 2, 3, 4}

Algoritmo de Prim



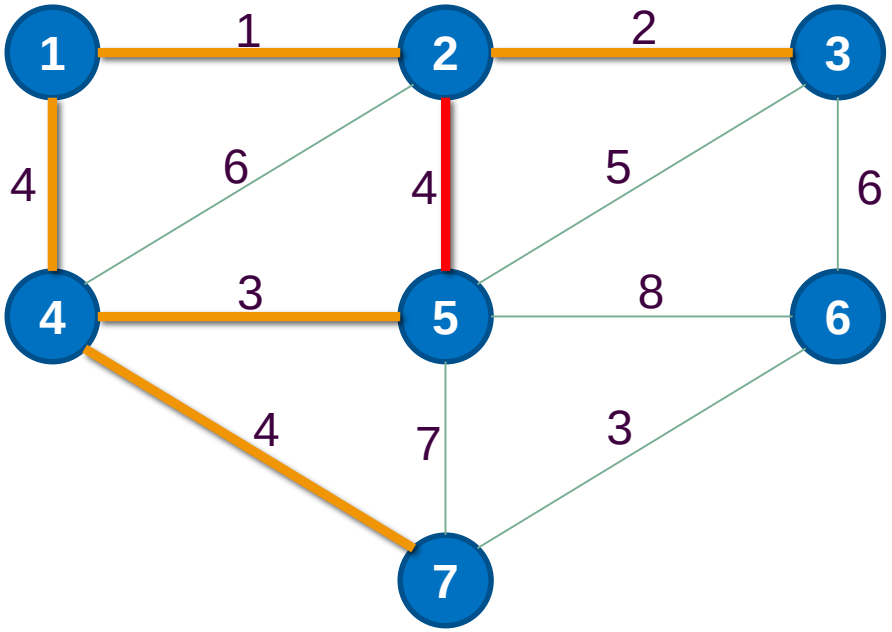
- {1, 2, 3, 4} → (4, 5)
- {1, 2, 3, 4, 5}

Algoritmo de Prim



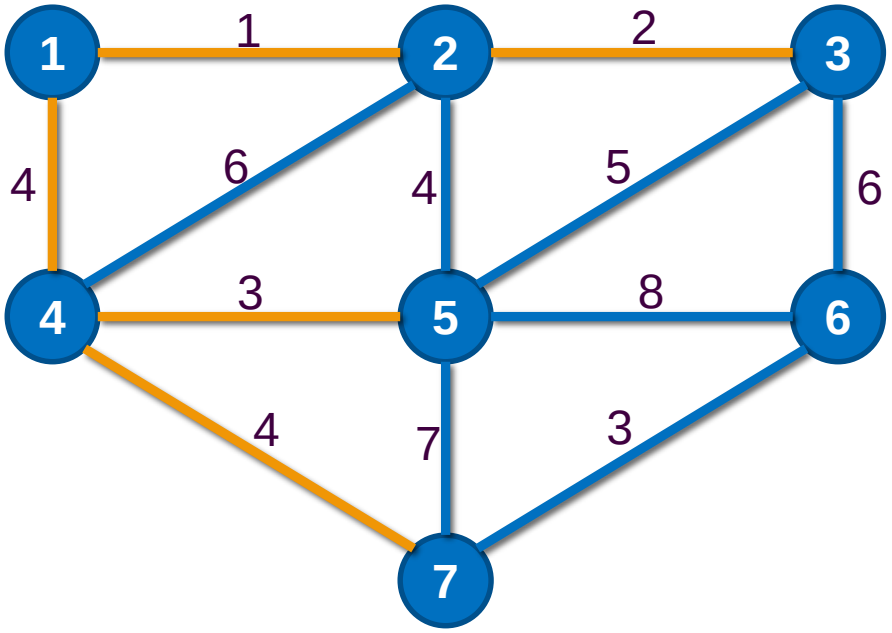
- {1, 2, 3, 4, 5}

Algoritmo de Prim



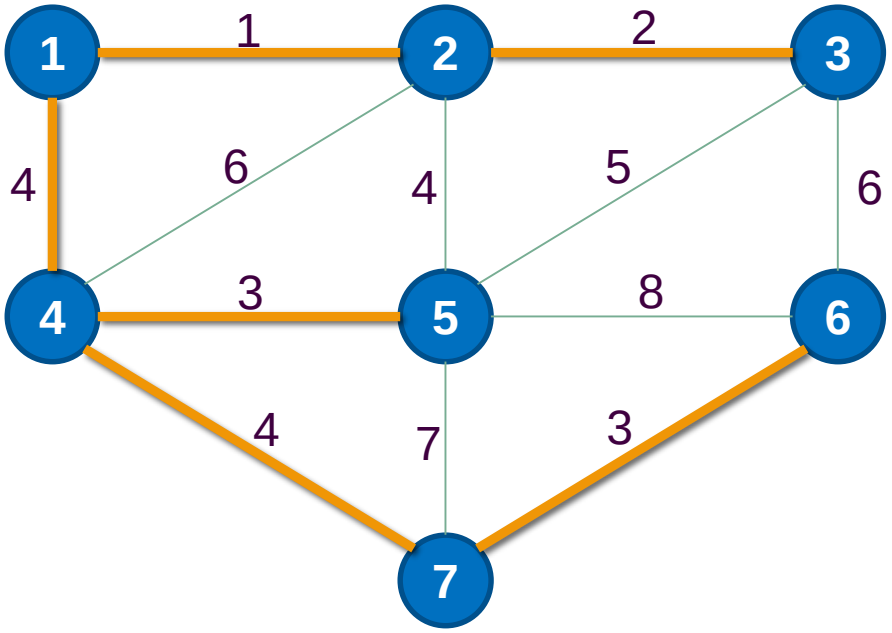
- {1, 2, 3, 4, 5} → (2, 5) ¡CICLO!
- → (4, 7)
- {1, 2, 3, 4, 5, 7}

Algoritmo de Prim



- {1, 2, 3, 4, 5, 7}

Algoritmo de Prim



- {1, 2, 3, 4, 5, 7} → (6, 7)
- {1, 2, 3, 4, 5, 6, 7}

Kruskal vs. Prim

- Complejidades
  - Kruskal: $O(a log_2 n)$ (a = nº aristas; n = nº vértices)
  - Prim: $O(n^2)$

  - ¿De qué depende que un algoritmo sea más eficiente que otro?

Kruskal vs. Prim

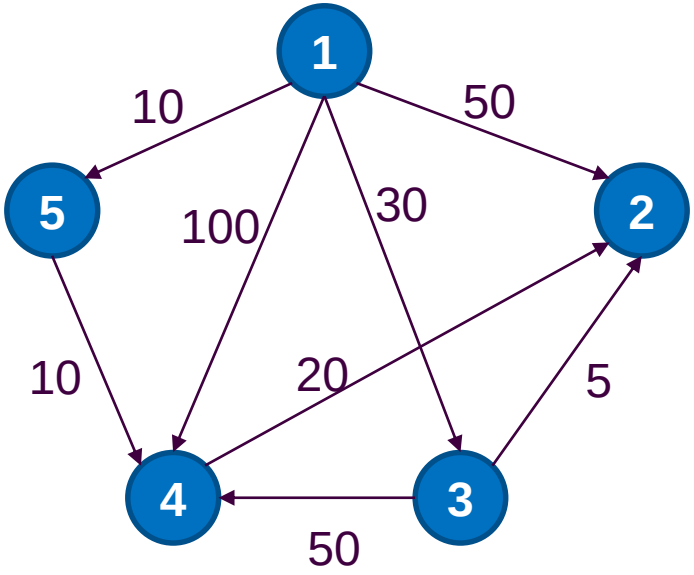
- Complejidades
  - Kruskal: $O(alog_2 n)$ (a = nº aristas; n = nº vértices)
  - Prim: $O(n^2)$

  - ¿De qué depende que un algoritmo sea más eficiente que otro?
    - Si el grafo es muy denso
      - $a \sim \frac{n(n-1)}{2}$
      - Kruskal: $O(n^2 \log_2 n)$
      - Prim: $O(n^2)$
    - Si el grafo es poco denso
      - $a \sim n$
      - Kruskal: $O(n \log_2 n)$
      - Prim: $O(n^2)$

Índice

Caminos mínimos

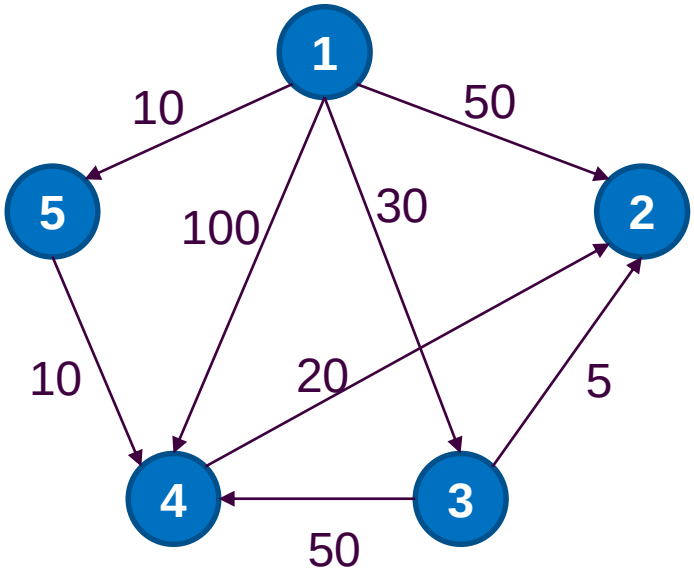
- En este caso trabajamos con grafos dirigidos G = (N, A), donde N es el conjunto de vértices y A el conjunto de arcos, cada uno de los cuales tiene asociado un coste no negativo.
- Identificando uno de los vértices como origen, se pretende determinar la longitud de los caminos mínimos desde el origen a cada vértice del grafo.



Algoritmo de Dijkstra

- Sean C y S respectivamente los conjuntos de vértices candidatos y escogidos
- S contiene en todo momento el conjunto de vértices cuya distancia mínima al origen es conocida y C contiene los vértices restantes.
- Al principio, S contendrá únicamente el origen, y al final, todos los vértices de G con lo que el problema queda resuelto.
- En cada iteración, cogemos el vértice de C cuya distancia al origen es mínima y lo incorporamos a S.

Algoritmo de Dijkstra

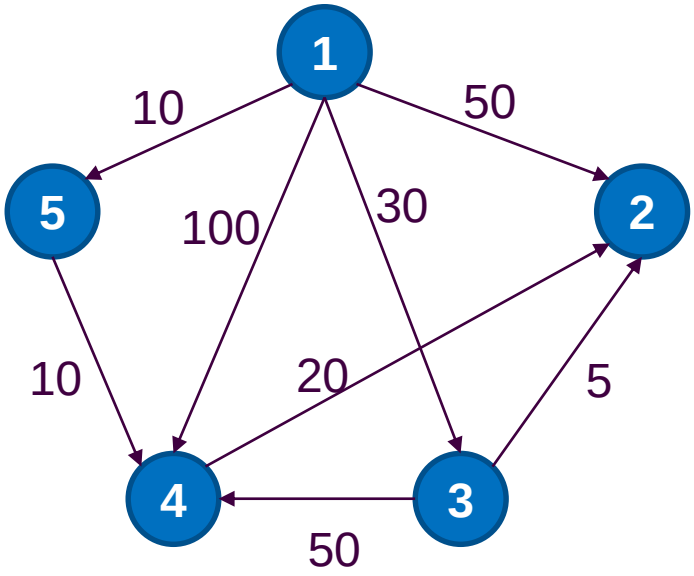


Etapa	v	C	S	D
1		{2, 3, 4, 5}	{ }	(-, 50, 30, 100, 10)
2	5	{2, 3, 4}	{5}	(-, 50, 30, 20, 10)
3	4	{2, 3}	{4, 5}	(-, 40, 30, 20, 10)
4	3	{2}	{3, 4, 5}	(-, 35, 30, 20, 10)
5	2	{ }	{2, 3, 4, 5}	

Algoritmo de Dijkstra

- Si además de la longitud del camino mínimo queremos saber cuál es ese camino, necesitamos almacenar otra tabla en la que se indica, para nodo, desde qué nodo se llega a él

Algoritmo de Dijkstra



Etapa	v	C	S	D	P
1		{2, 3, 4, 5}	{ }	(-, 50, 30, 100, 10)	(-, 1, 1, 1, 1)
2	5	{2, 3, 4}	{5}	(-, 50, 30, 20, 10)	(-, 1, 1, 5, 1)
3	4	{2, 3}	{4, 5}	(-, 40, 30, 20, 10)	(-, 4, 1, 5, 1)
4	3	{2}	{3, 4, 5}	(-, 35, 30, 20, 10)	(-, 3, 1, 5, 1)
5	2	{ }	{2, 3, 4, 5}		

Algoritmo de Dijkstra

- Complejidad: depende de la implementación utilizada
  - $O(n^2)$
  - $O((a+n)\log_2 n)$ utilizando montículos

  - Si el grafo es muy denso, es mejor la primera implementación (tal cual visto en clase)
  - Si el grafo es poco denso es mejor utilizar montículos