Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 13

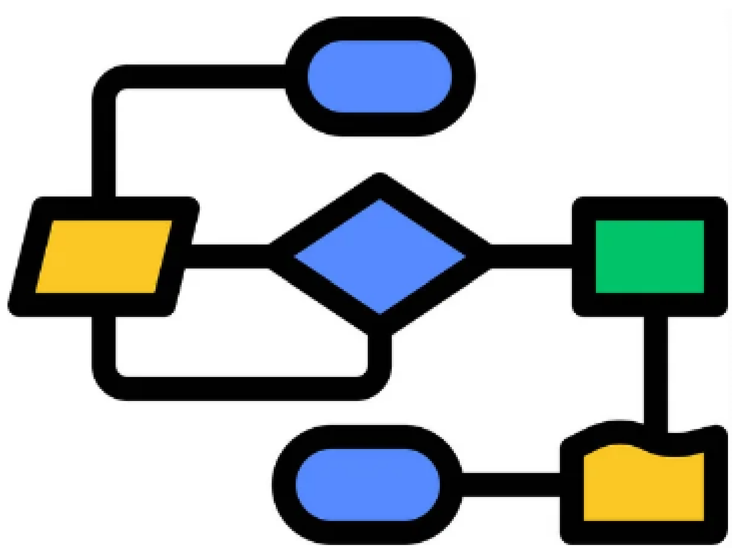
Curso 2023-2024

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T4. Técnicas Algorítmicas

- Divide y Vencerás
- Algoritmos Voraces
- Programación Dinámica
- Backtracking

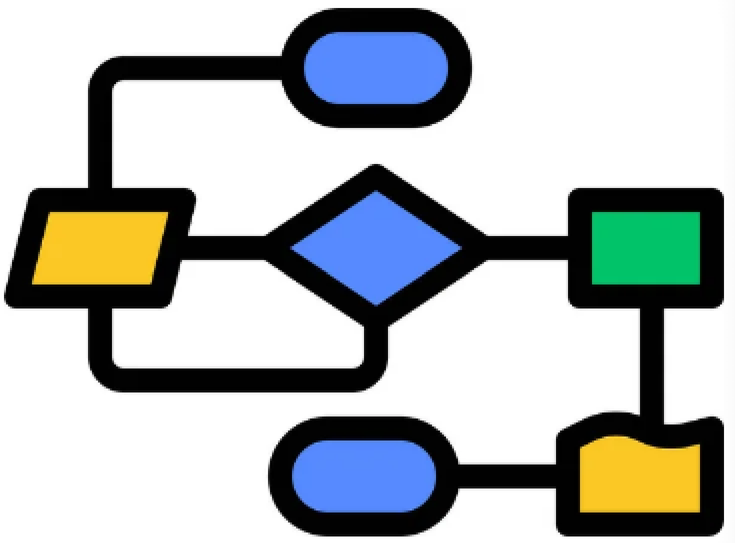


UNIVERSIDAD NEBRIJA

ÍNDICE

T4. Técnicas Algorítmicas

- Divide y Vencerás
- **Algoritmos Voraces**
- Programación Dinámica
- Backtracking



UNIVERSIDAD NEBRIJA

ÍNDICE

El nombre de los **algoritmos Voraces (Greedy o ávidos)** describen muy bien lo que hacen:

- Cada **etapa** se consume una parte de los datos.
- Su objetivo es que, **bajo ciertas condiciones**, la parte consumida sea lo **mayor/menor posible**.

La técnica de los **algoritmos Voraces (Greedy o ávidos)** consiste en resolver un problema de manera **progresiva** tomando **decisiones** o realizando **acciones** que permiten ir obteniendo la **solución del problema poco a poco**.

Cada decisión tomada permite obtener una **solución parcial** del problema, disminuyendo la dimensión del problema.

Habitualmente, se utiliza en **problemas de optimización** (obtener un **máximo**, un **mínimo**, etc.).

Condiciones de los **algoritmos Voraces:**

- Existe una **función objetivo** que se quiere **maximizar** o **minimizar**.
- La **función objetivo** depende de varias variables que toman valores de en un **Dominio (conjunto de valores candidatos)**.
- Existe un **función solución** que permite saber si ciertos valores de las variables forman parte de la solución del problema. La asignación de un valor a una variable se llama **Decisión**.
- Existen **restricciones** que se imponen a los valores de las **variables de la función objetivo**.
- Existe una **función de factibilidad** para determinar si las **decisiones tomadas** hasta el momento **cumplen las restricciones**.
- Las **decisiones factibles** tomadas hasta el momento forman la **Solución en Curso**.

Los **algoritmos Voraces** son básicamente **iterativos** y tienen una **serie de etapas**.

Cada **etapa** **consume** una **parte de los datos o candidatos (*C*)** y **construye** una **parte de la solución** (***S***).

Finalizan cuando se han consumido **todos los datos/candidatos o se alcanza una solución**.

Cada parte consumida se evalúa una única vez, siendo **descartada** o **seleccionada**:

- Si es **seleccionada**, forma **parte de la solución** (***S***).
- Mientras que si es **descartada**, **no forma parte de la solución ni volverá a ser considerada para la misma**.

Los **algoritmos Voraces** funcionan de la siguiente forma:

1. **Inicialmente** se parte de una **solución vacía** (***S***).
2. En **cada etapa**, se analiza en **conjunto de candidatos** (***C***) para **elegir el elemento** que se añade a la **solución** (***S***).
3. Se **termina** cuando el conjunto de **elementos seleccionados** constituyen una **solución** (***S***).

```
cjtoSolucion algVoraz (cjtoCandidatos &C)  }
    cjtoSolución S = ∅;
    while (!esSolucion(S) && C ≠ ∅) {
      x = seleccionar(C);
      C = C – {x}
      if (esFactible (S, x)) {
            insertar(S, x); }
    }
    if (esSolucion(S)) {
      return S; }
    else
      return { ∅; } // No se encontró la solución
}
```

Funciones:

esSolucion(S): Comprueba si el conjunto de candidatos seleccionado hasta el momento es una solución (independientemente de que sea óptima o no).

Seleccionar(C): Selecciona el elemento más adecuado del conjunto de candidatos pendientes (no seleccionados ni rechazados).

esFactible(S, x): Indica si a partir del conjunto S y añadiendo x, es posible construir una solución (posiblemente añadiendo otros elementos).

Insertar(S, x): Añade el elemento x al conjunto solución S. También puede ser necesario hacer otras operaciones.

El coste de los algoritmos **Voraces** depende de:

1. El **número de iteraciones del bucle**, que depende del **tamaño de la solución construida** y del **tamaño del conjunto de candidatos**.
2. El **coste de las funciones selección** y **factible**:
   a) La **función factible** suele tener un **coste constante**.
   b) La **función selección** tiene que **explorar el conjunto de candidatos**.

Los algoritmos **Voraces** son **bastante eficientes**: ***O(n log n), O(n²)***.

Al tomar **decisiones localmente óptimas** y no reconsiderar dichas decisiones, **no garantizan** que la **solución sea la óptima** del problema, ni siquiera que **se obtenga una solución**, aunque exista. Esto se debe a que las decisiones locales no garantizan la obtención de una **solución óptima global.**

Por este motivo, los problemas resueltos mediante la técnica voraz deben de cumplir:

"Dentro de una secuencia óptima de decisiones, toda subsecuencia ha de ser también óptima"

Descomposición en factores primos:

Sea un número natural ***n***, sus factores primos: $300 = 2^2\, 3^1 5^2$

Conjunto de candidatos: todos los **factores primos** posibles para el número (menores que su mitad).

Conjunto Solución: todos los **factores que dividen al número**.

Función objetivo: obtener el cociente 1.

EJEMPLO 1: Algoritmos Voraces
Descomposición en factores primos

Sea un número natural ***n***, sus factores primos: $300 = 2^2\,3^1 5^2$

Proceso de resolución: eliminar en cada etapa un divisor del número ***n***, cuantas veces sea posible:

1. Se **elimina el 2** tantas veces como sea posible, y se considera el **cociente final**,
2. Se **elimina el 3**, de igual manera.
3. Se **continua de igual forma con el resto de factores** hasta que el **cociente sea 1**.

EJEMPLO 1: Algoritmos Voraces

Descomposición en factores primos

```
vector<termino> calcularFactores (int N) {
    vector<termino> factores;
    termino ter;
    for (int i{2}; i <= (sqrt(N)); i++) {
        ter.veces=0;
        ter.f = i;
        while (N % i == 0) {
            ter.veces++;
            N /=i;
        }
        if (ter.veces > 0) {
            factores.push_back(ter);
        }
    }

    return factores;
}
```

```
struct termino{
    int f, veces;
};
```

Dar cambio con el menor número de billetes/monedas:

Se pide crear un algoritmo que permita a una máquina expendedora **devolver el cambio mediante el menor número de billetes posible**, considerando que el número de billetes es limitado, es decir, se tiene un número concreto de billetes de cada tipo.

Dar cambio con el menor número de billetes/monedas:

Se pide crear un algoritmo que permita a una máquina expendedora **devolver el cambio mediante el menor número de billetes posible**, considerando que el número de billetes es limitado, es decir, se tiene un número concreto de billetes de cada tipo.

La estrategia a seguir consiste en **seleccionar los billetes/monedas de mayor valor** que no superen la cantidad de cambio a devolver, en cada etapa.

Conjunto de candidatos: todos los tipos de monedas disponibles, suponiendo que de cada tipo hay una cantidad limitada.

Conjunto Solución: las monedas que suman el importe.

Función objetivo: minimizar el número de monedas utilizadas ($\sum_i x_i$).

$$\sum_{i=1..n} x_i \, c_i \qquad x_i \geq 0$$

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

Supongamos que hay que devolver **128 euros** y se tiene la siguiente disponibilidad de billetes y monedas:

- **3 billetes de 50 euros**
- **2 billetes de 20 euros**
- **1 billete de 5 euros**
- **6 monedas de 1 euro**

1. Se divide **128 entre 50** y se obtiene como **cociente 2**. Esto representa el **número de billetes de 50**, quedando una cantidad de cambio a devolver de **28 euros**.
2. Se divide **28 entre 20** y se obtiene como **cociente 1**, que es el **número de billetes de 20** que se pueden utilizar sin pasarse. Ahora queda pendiente la cantidad de **8 euros**.
3. Se **divide 8 entre 5** y se obtiene **1**, luego seleccionamos **1 billete de 5**.
4. La cantidad de cambio a devolver ahora es **3 euros**.
5. Se **divide 3 entre 1** y se obtiene como **cociente 3**, que son las **monedas de 1 euro** necesarias para terminar el problema de forma correcta.

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

```
array <int, TMonedas> obtenerCambio
                (int  T, const array <int, TMonedas> &C) {
    array <int, TMonedas> S;  // array solución
    int cambio{0}, i{0};
    S.fill(0);
    while (cambio != T) {
        while (C.at(i) > (T - cambio) && i < TMonedas){
            i++;
        }

        if (i==TMonedas) {
            cout << "No existe solución";
            S.fill(0);
        }
        else {
           S.at(i)=(T - cambio) / C.at(i);
           cambio = cambio + C.at(i) * S.at(i);
        }
    }
    return S;
}
```

Inconvenientes

- Si tenemos **5 monedas** de valores C={50, 25, 20, 5, 1} y se tiene que **obtener 42**, la solución que se obtendría **S = {25, 5, 5, 5, 1, 1}**, mientras que la **solución óptima sería S = {20, 20, 1, 1}**.
- Además, si **no se incluye la unidad, tampoco se garantiza la solución**.
- Para obtener una **solución óptima** es necesario que el conjunto de candidatos esté formado por **valores que sean potencia de un tipo básico**. Por ejemplo, C={125, 25, 5, 1}. Así, sólo hay que encontrar la descomposición de la cantidad en base a ese valor, que es única y mínima.

- Para que la función de selección funcione de forma adecuada, el **vector de valores de billetes/monedas debe estar <u>ordenado en orden decreciente</u>**.
- El coste del algoritmo es de $O(max(n \log n, m)$ donde n es el **número de valores de billetes/monedas** de C, y m el **número de iteraciones del bucle exterior**.

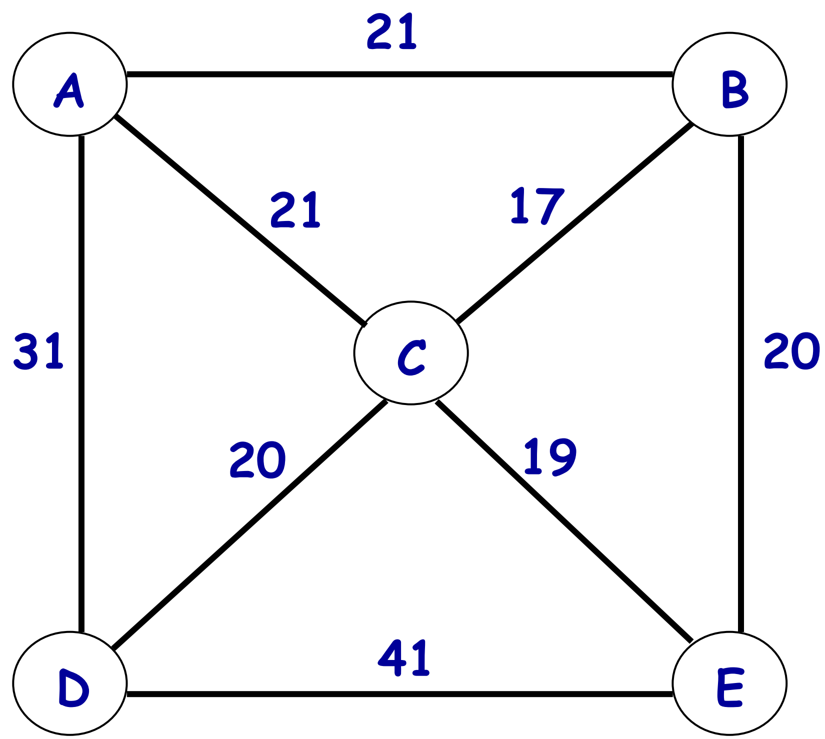
Árbol de Recubrimiento Mínimo

Sea **G es un grafo valorado y conexo**.

Su **árbol de recubrimiento mínimo** es un **árbol de recubrimiento que cumple que la suma de las etiquetas de sus aristas es la menor posible**.

Se obtiene a través de los algoritmos: **Kruskal** y **Prim**

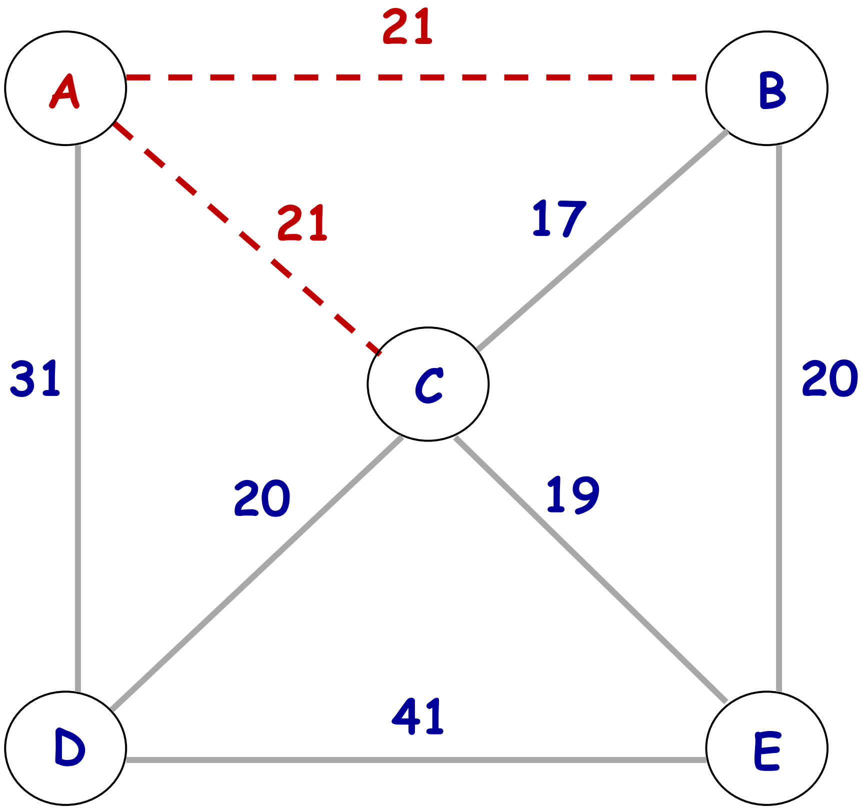
Supongamos el siguiente **grafo conexo valorado**

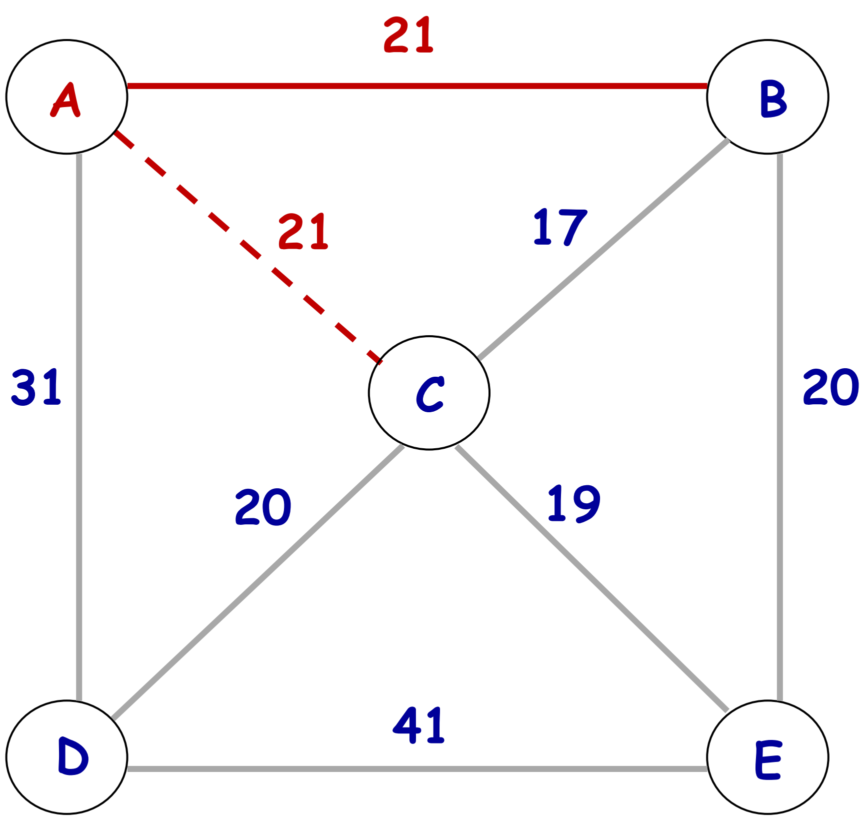


Algoritmo PRIM: Consiste en **añadir, en cada paso, una arista de peso mínimo** a un **árbol** previamente construido y siempre haciendo que el **subgrafo** que se va formando sea **conexo**.

EJEMPLO 3: Algoritmos Voraces

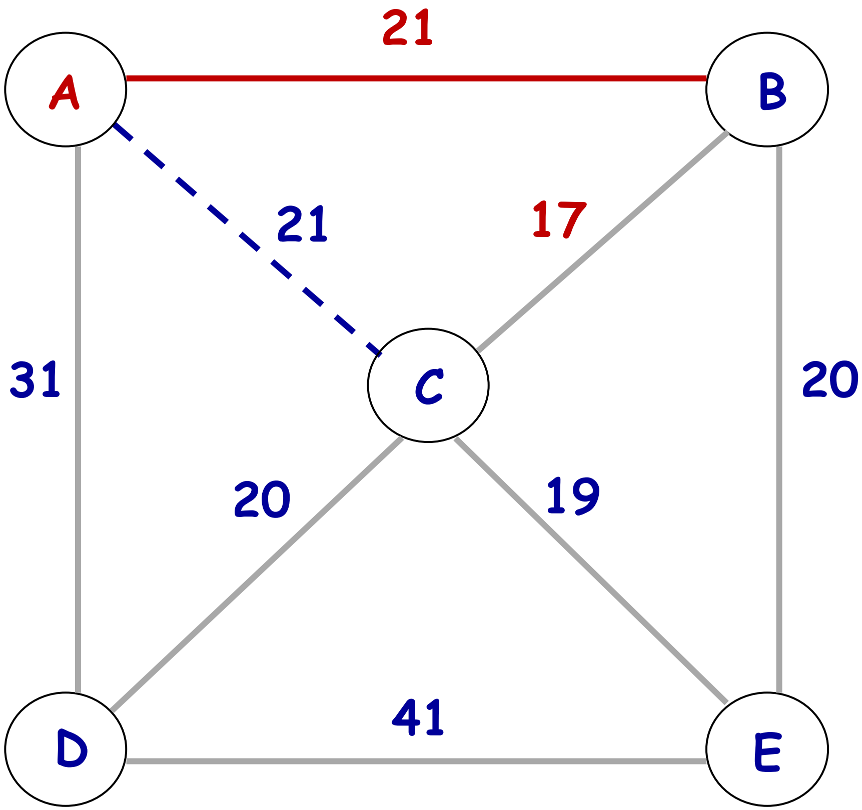
Algoritmo de PRIM

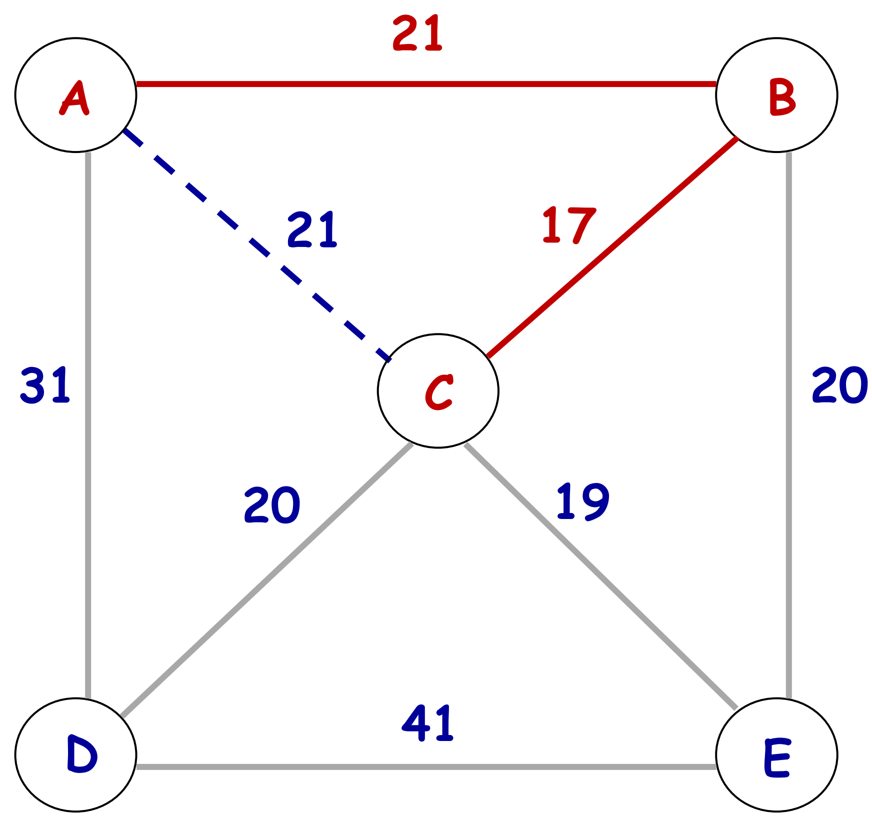




EJEMPLO 3: Algoritmos Voraces

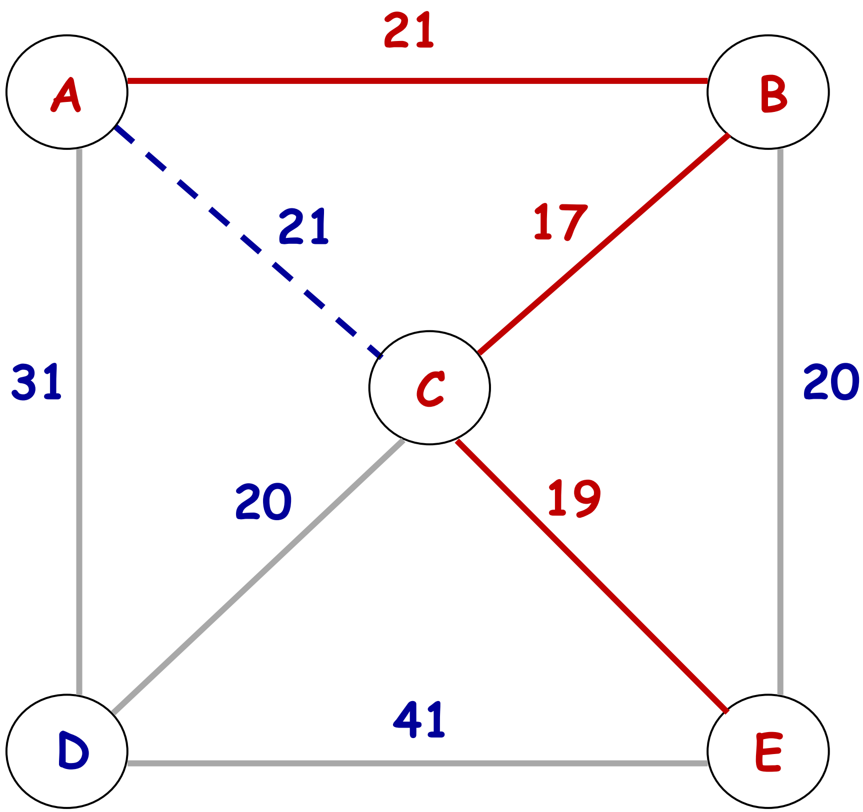
Algoritmo de PRIM



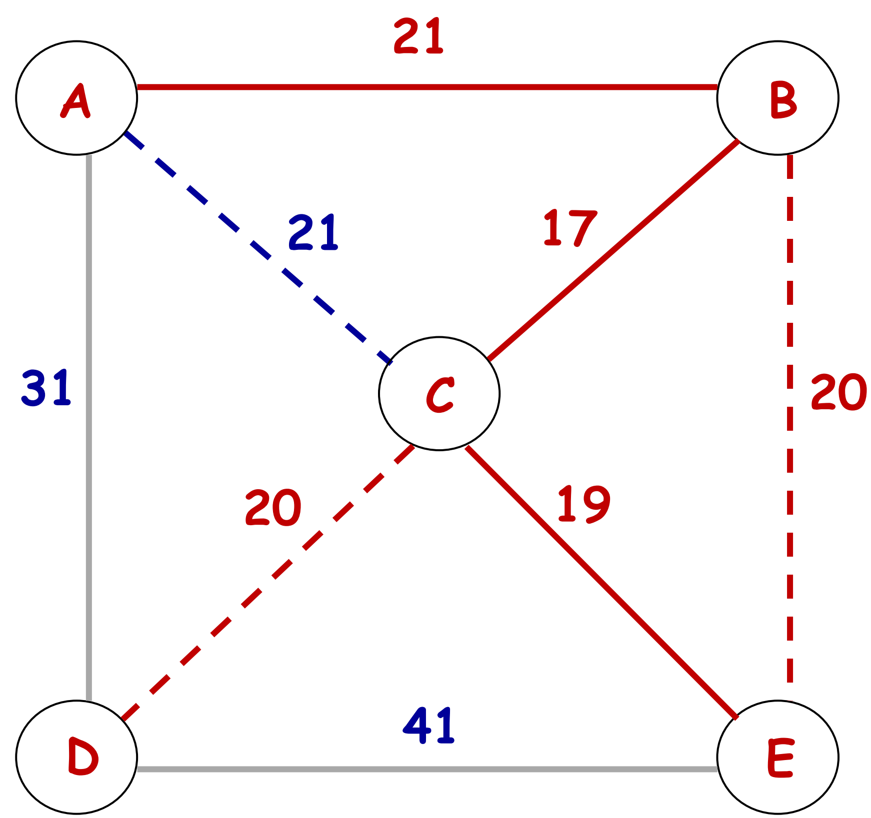


EJEMPLO 3: Algoritmos Voraces

Algoritmo de PRIM

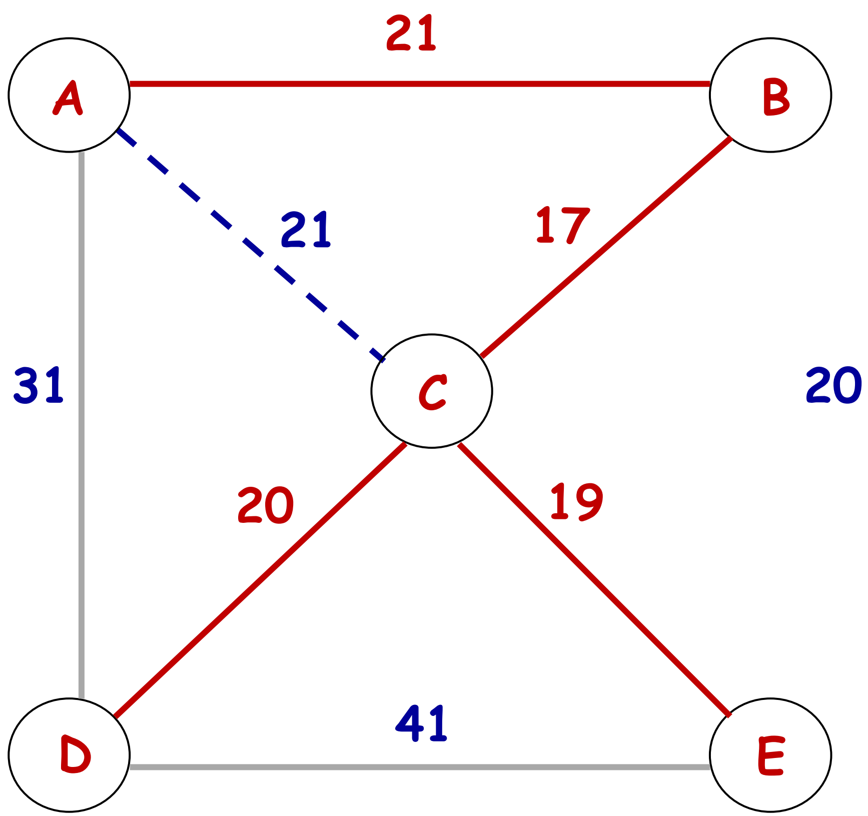


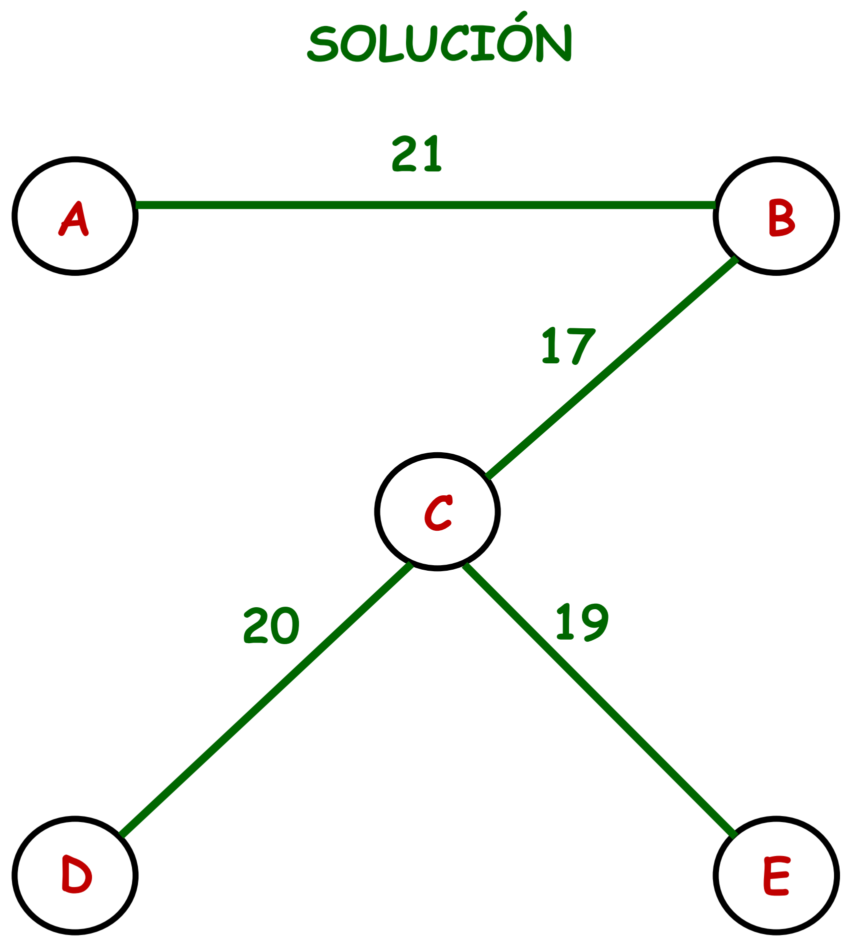
No vale porque FORMA CICLO



EJEMPLO 3: Algoritmos Voraces

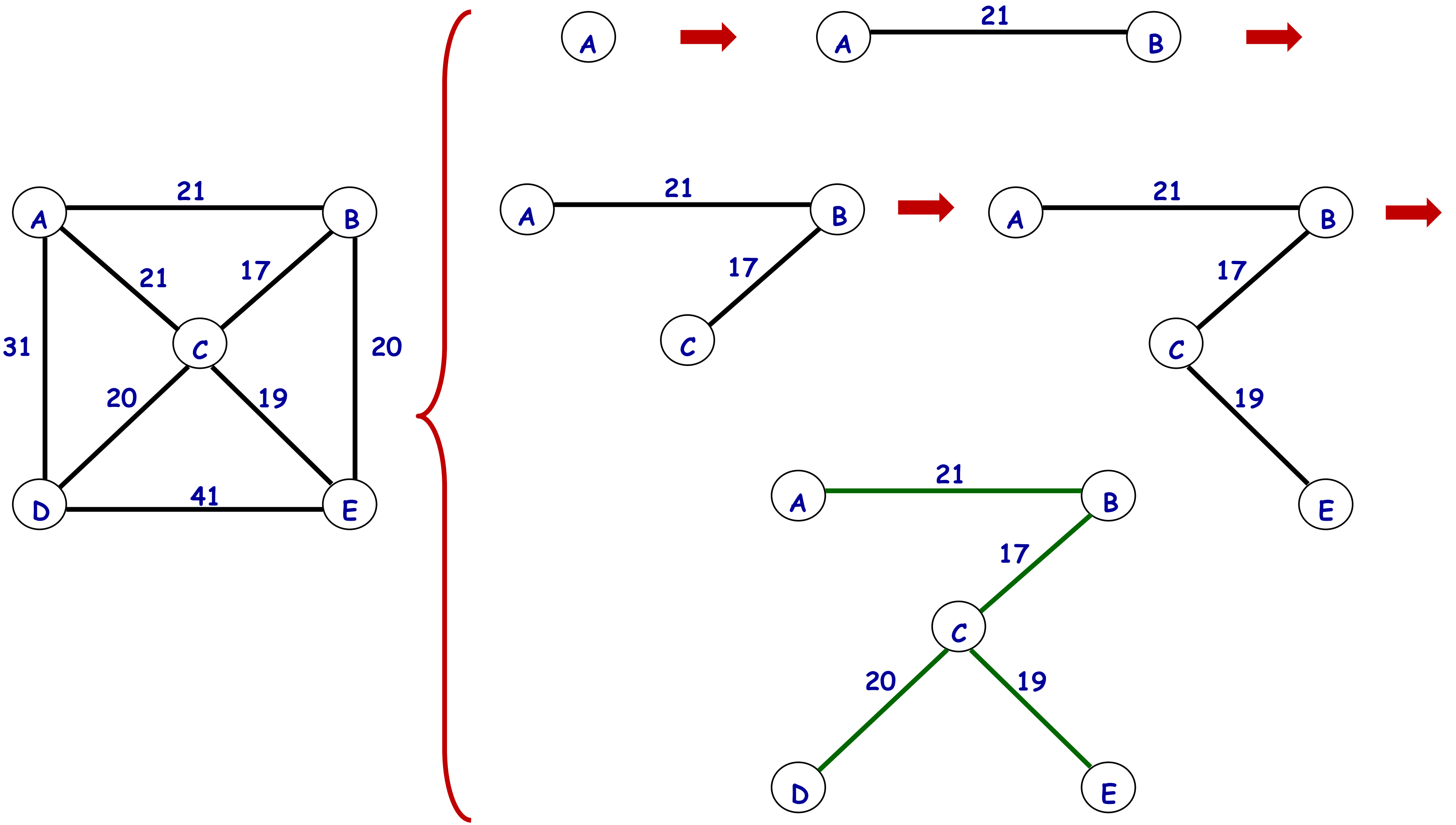
Algoritmo de PRIM



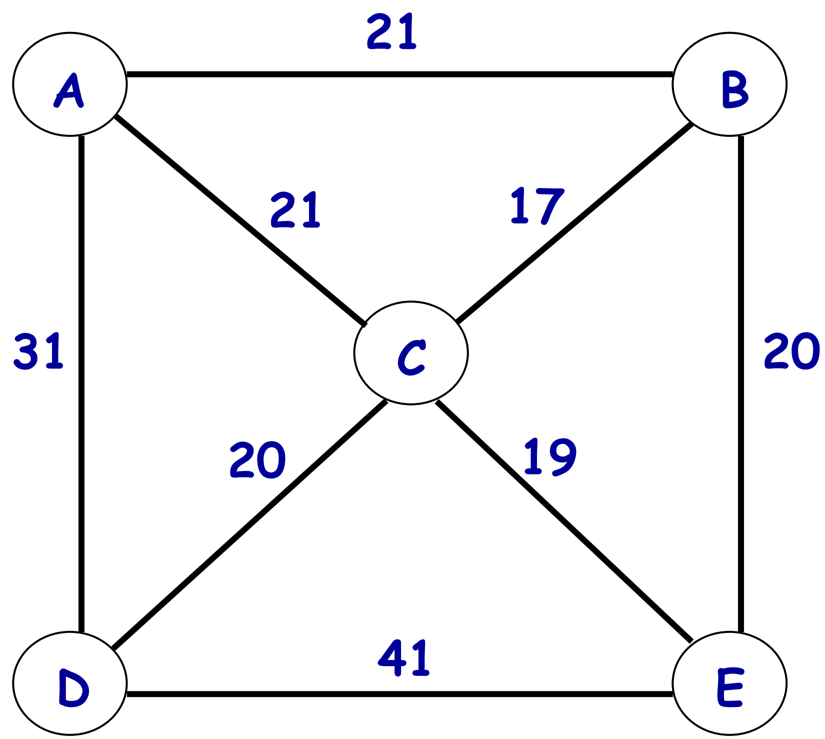


EJEMPLO 3: Algoritmos Voraces

Algoritmo de PRIM

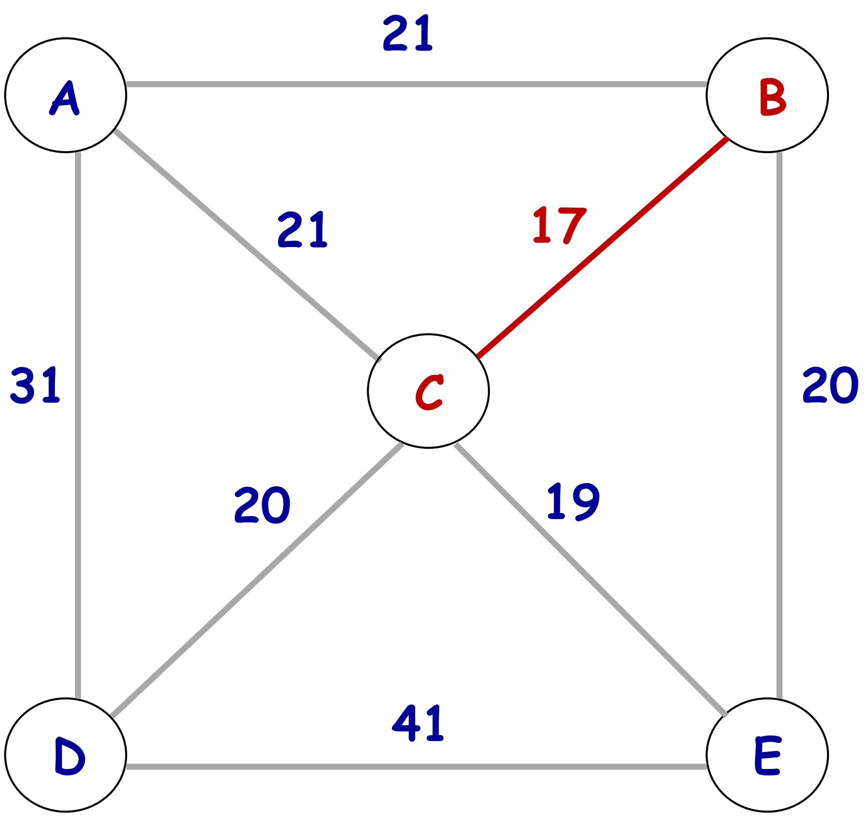


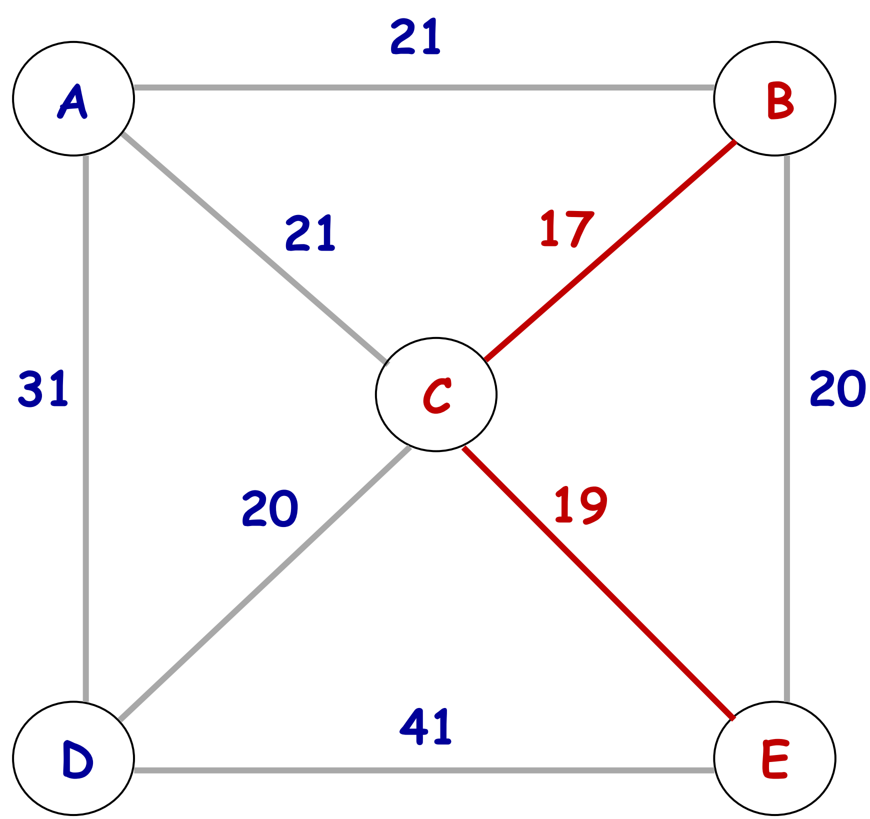
Supongamos el siguiente **grafo conexo valorado**



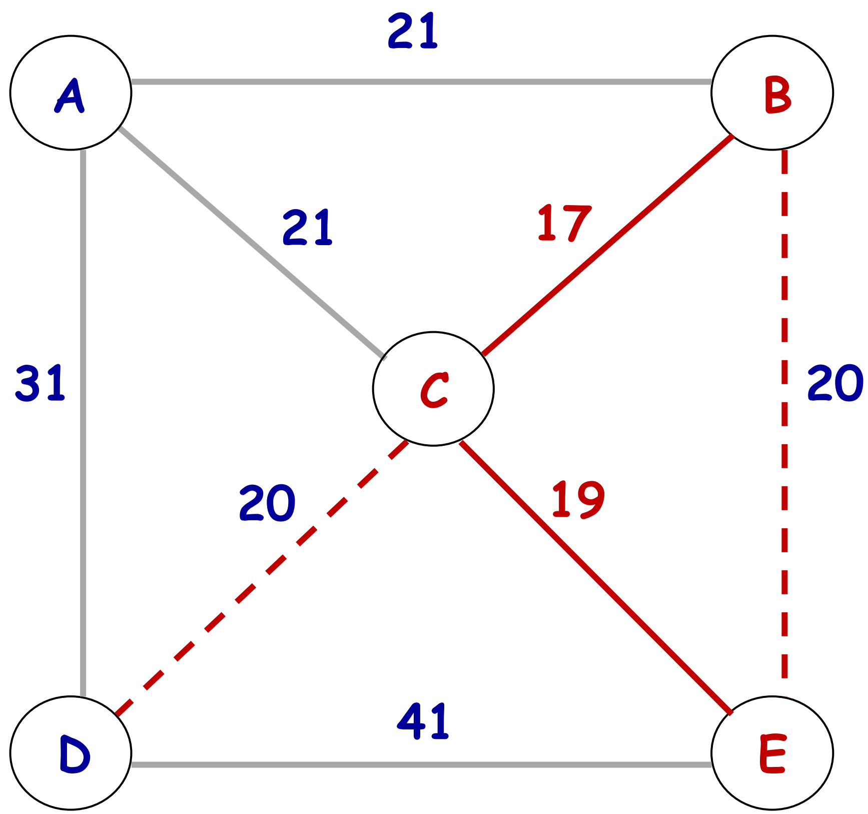
Algoritmo KRUSKAL: Consiste en elegir las **aristas de menor peso que no forman ciclos.** Para poder elegir dichas aristas es necesario **ordenarlas de menor a mayor peso**.

EJEMPLO 4: Algoritmos Voraces
Algoritmo de KRUSKAL

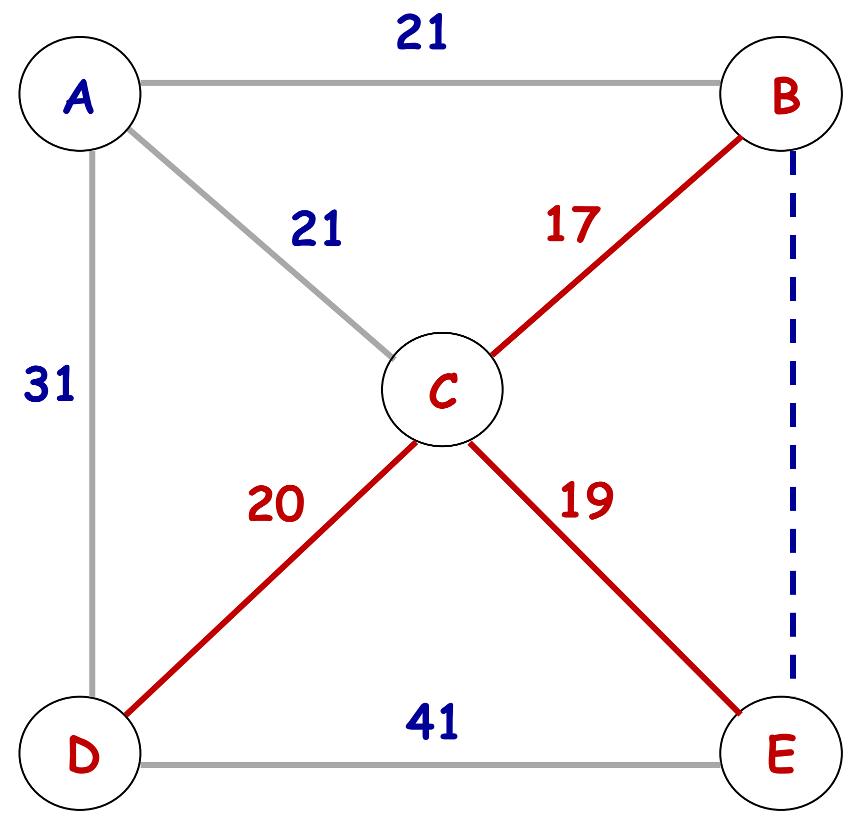




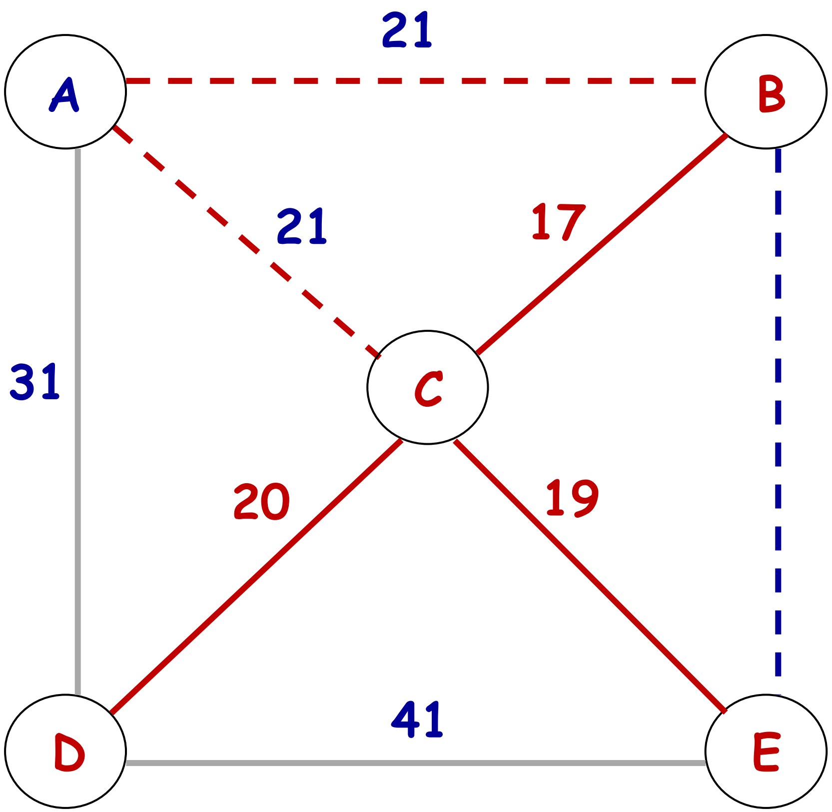
EJEMPLO 4: Algoritmos Voraces
Algoritmo de KRUSKAL



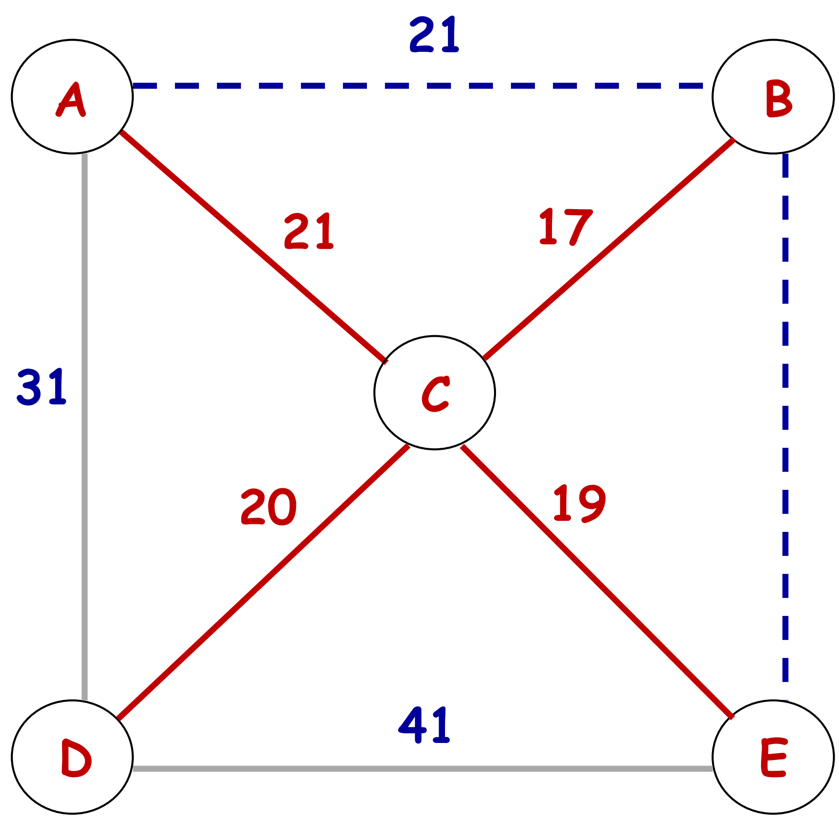
No vale porque FORMA CICLO



Algoritmo de KRUSKAL

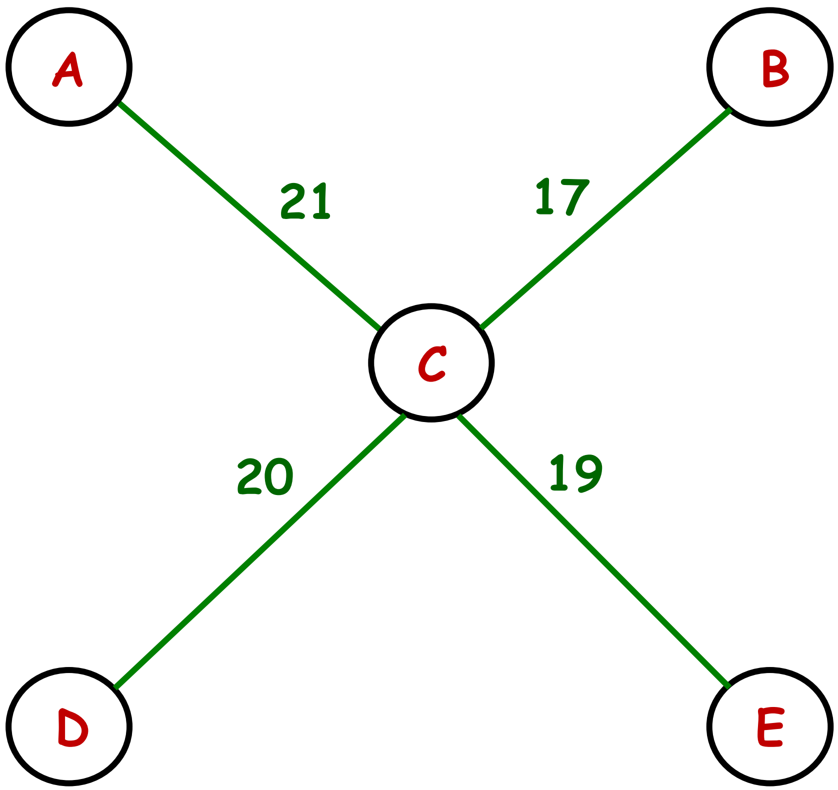


No vale porque FORMA CICLO



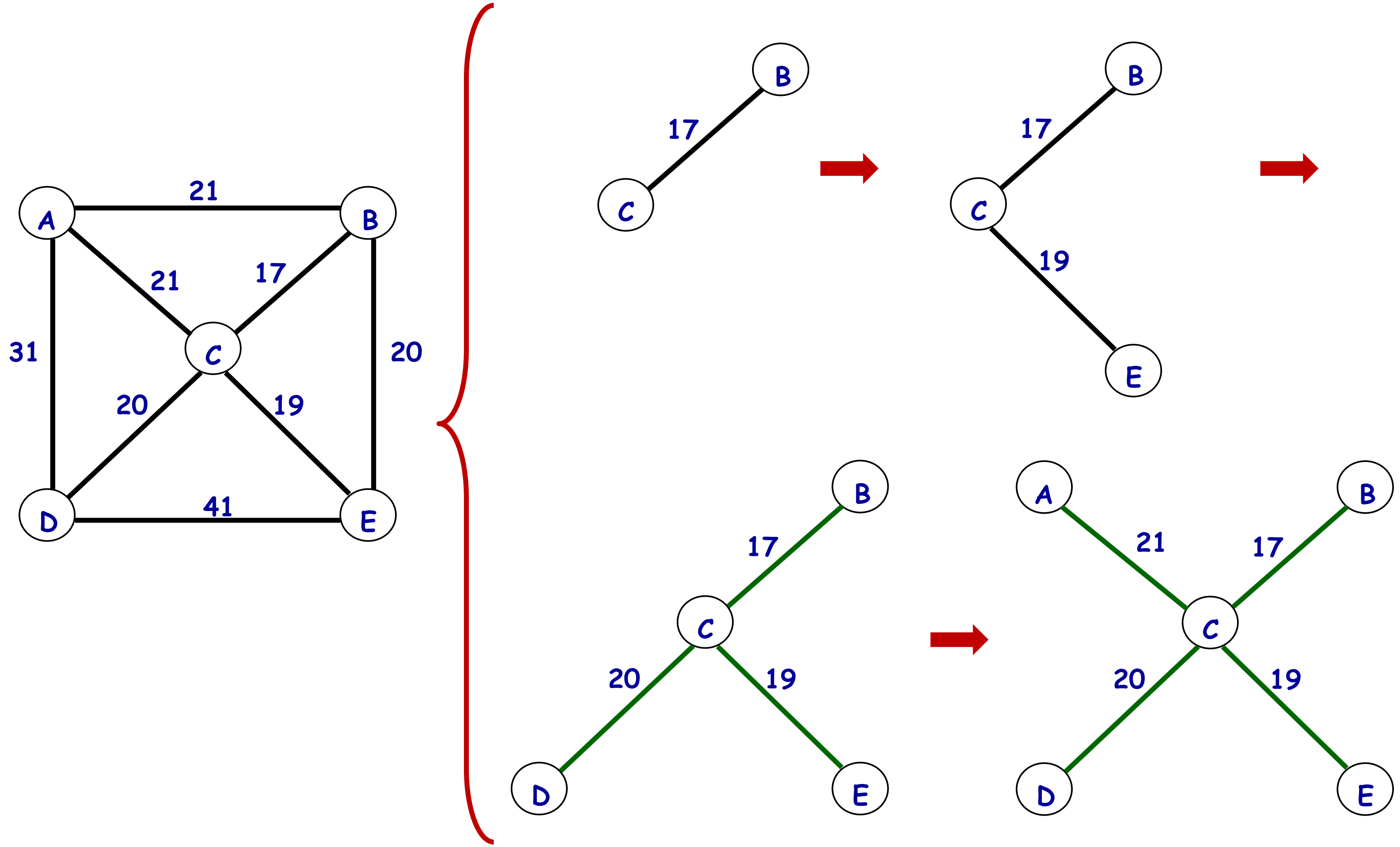
EJEMPLO 4: Algoritmos Voraces
Algoritmo de KRUSKAL

SOLUCIÓN



EJEMPLO 4: Algoritmos Voraces

Algoritmo de KRUSKAL



EJEMPLO 5: Algoritmos Voraces - El problema de la Mochila

- Se dispone de una colección de $\boldsymbol{n}$ objetos $\boldsymbol{o_1, o_2, \ldots, o_n}$, cada uno de ellos con un peso $\boldsymbol{p_i}$ y un valor asociado $\boldsymbol{v_i}$.
- Se tiene una mochila capaz de soportar un peso máximo $\boldsymbol{p_{max}}$.
- **El problema consiste en maximizar el valor de los objetos que se guardan en la mochila, pero sin superar el peso $\boldsymbol{p_{max}}$.**
- Los objetos se pueden o no fraccionar, existiendo dos variantes del problema:
  - **Mochila fraccionada**: los objetos se pueden dividir, luego se pueden introducir objetos fraccionado en la mochila. **Solución a través de la técnica de los Algoritmos Voraces.**
  - **Mochila entera**: los objetos no se pueden dividir, por tanto, la mochila solo puede contener objetos enteros.

EJEMPLO 5: Algoritmos Voraces - El problema de la Mochila

- Los **objetos se pueden fraccionar** y cada trozo del objeto $\boldsymbol{o_i}$ se llama $\boldsymbol{x_i}$. Así, el valor de $\boldsymbol{x_i}$ será 0 o 1 en función de que no esté o esté en la mochila.
- La **función objetivo** que hay que maximizar son el valor de los objetos incluidos en la mochila: $\sum_i \boldsymbol{x_i v_i}$
- El **dominio o conjunto de valores candidatos** será la colección de objetos que se quieren incluir en la mochila.
- **Función solución**: S={x_i,…, x_k} siempre que $\sum_i x_i p_i = p_{max}$
- **Función factible**: S={x_i,…, x_k} siempre que $\sum_i x_i p_i \leq p_{max}$
- **Función selección**: hay varias estrategias:
  - Seleccionar los objetos en orden decreciente del valor.
  - Seleccionar los objetos en orden creciente del peso.
  - Seleccionar los objetos por orden decreciente de relación valor/peso. Esta es la única estrategia que lleva a lo solución óptima.

EJEMPLO 5: Algoritmos Voraces - El problema de la Mochila

```
float cargarMochila (const array<float,DIM> &V, const array<float,DIM> &P,
                     array<float,DIM> &X, float pmax){
    array<objeto,DIM> VP;

    for(int i{0}; i < DIM; i++){
        VP.at(i).coste = V.at(i)/P.at(i);
        VP.at(i).posicion = i;
        X.at(i) = 0;
    }
    ordenarDecrecienteCoste(VP);
    float peso{0}, valor{0};
    int i{0}, j;
    while(peso <= pmax && i < DIM){
        j = VP.at(i).posicion;
        if ((peso + P.at(j)) <= pmax){
            X.at(j) = 1;
            valor += V.at(j);
            peso += P.at(j);
        }
        else{
            X.at(j) = (pmax-peso)/P.at(j);
            valor += V.at(j)*X.at(j);
            peso = pmax;
        }
        i++;
    }
    return valor;
}
```

```
struct objeto{
    float coste;
    int posicion;
}
```

- El tamaño del problema es el número de objetos ***n***.
- El primer bucle hace ***n*** iteraciones y el coste es constante. Por tanto, el coste de todo el bucle es ***O(n)***.
- El coste de la función ordenación decreciente de los objetos por valor/peso es el coste del algoritmo de ordenación utilizado (en el mejor caso ***O(nlogn)***).
- El segundo bucle realiza, a lo sumo ***n*** iteraciones, y el coste de su cuerpo es constante, por tanto, el coste total del segundo bucle también de ***O(n)***.
- Luego, el coste total del algoritmo utilizado será:

$$O(n) + O(nlogn) + O(n) = O(nlogn)$$