

Análisis y Diseño de Algoritmos

Tema 5: Algoritmos Voraces



Contenido

- Introducción
 - El enfoque voraz/greedy
 - Elementos necesarios
 - Esquema de una solución
- Programación Dinámica vs Algoritmos greedy
 - Planificación de actividades
 - Solución con programación dinámica
 - Solución con una estrategia greedy
- Árbol de Recubrimiento mínimo
 - Algoritmo de Prim
 - Algoritmo de Kruskal
- Caminos más cortos desde un origen
 - Algoritmo de Dijkstra
- El problema de la mochila: versiones discreta y continua
- El problema del viajante de comercio
- Referencias

2

Introducción

- La **programación dinámica** se aplica a **problemas de optimización**, que tienen **subproblemas**, cuya **solución óptima** lleva a la **solución óptima del problema original**.
- Un algoritmo basado en programación dinámica construye la solución óptima final paso a paso.
- En cada paso, decide cual de entre todas las posibles soluciones, **es la que produce el mejor resultado**.
- El hecho de que tenga que explorar todas las alternativas hace que, en muchos caso, la **complejidad** sea **elevada**.

3

Introducción

- La **técnica greedy o voraz se parece a la programación dinámica** en que construye soluciones óptimas de problemas, mediante una secuencia de soluciones a subproblemas que son óptimas.
- La diferencia entre ambas técnicas es que **un algoritmo voraz**, en cada paso, no explora todas las alternativas, sino que **escoge la solución que en ese momento le parece la mejor**
- **Esta técnica puede fallar** en algunos casos, **pero** sorprendentemente, para muchos problemas funciona bien, y **tiene una complejidad menor que las soluciones que siguen el esquema dinámico**.

4

El enfoque voraz

- El **enfoque voraz es una técnica constructiva por pasos**, en la que en cada paso, se escoge una solución más cercana a la solución final.
- En cada paso se decide cual de entre todas las soluciones posibles es la mejor. Esta solución parcial debe satisfacer las siguientes condiciones:
 - **factible**: debe satisfacer las restricciones del problema.
 - **localmente óptima**: es la mejor solución de entre todas las posibles, con el conocimiento que se tiene en ese momento.
 - **irrevocable**: una vez seleccionadas la solución, no puede modificarse más adelante.

5

Elementos necesarios

- Para construir un algoritmo voraz necesitamos:
 - Una función de terminación, que nos dice si la solución parcial actual es ya la definitiva
 - Una función que genere soluciones candidatas
 - Una función de selección, que escoja la mejor solución
 - Una función objetivo, que mida la calidad de la solución final encontrada

6

Esquema de una solución

```

import java.util.*;
public static TSolucion voraz(TProblema p){
    private List<TSolucion > lista;
    private boolean factible=true;
    private TSolucion sol;

    sol = solucionVacia(p);
    while (!completa(sol,P) && factible){
        lista = generarSoluciones(sol);
        if (lista.isEmpty()) factible = false;
        else{
            sol = extender(sol,p,seleccionar(lista,p));
        }
    }
    if (!factible) return null;
    else return sol;
}
            
```

primera solución caso base

función de terminación

función de generación

función de selección

construcción de la siguiente solución óptima

Programación dinámica vs algoritmos voraces: planificación de actividades

Supongamos que varias actividades $S = \{a_1, \dots, a_n\}$ hacen uso de un recurso común (pista deportiva, sala de presentaciones, etc.) que no pueden utilizar simultáneamente. El objetivo del problema es encontrar el máximo número de actividades que pueden realizarse sin que se interfieran unas a otras.

1. Cada actividad comienza en el tiempo s_i , y termina en el tiempo f_i , tal que $0 \leq s_i < f_i < \infty$
2. Si se selecciona la actividad a_i , tiene lugar en el intervalo $[s_i, f_i)$.
3. Las actividades a_i y a_j son *compatibles* sii los intervalos $[s_i, f_i)$ y $[s_j, f_j)$ no se solapan.

Planificación de actividades

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

9

Planificación de actividades utilizando programación dinámica

Sea S_{ij} el conjunto de actividades que

- empiezan después de que termine a_i
- terminan antes de que empiece a_j

Supongamos que queremos encontrar el número máximo de actividades dentro de S_{ij} que no se interfieren. Llamamos a este conjunto óptimo A_{ij} . Veamos que posee la propiedad de *subproblema óptimo*.

Supongamos que A_{ij} contiene una actividad a_k . Entonces el problema se subdivide en dos subproblemas:

- encontrar una solución óptima para S_{ik} : A_{ik}
- encontrar una solución óptima para S_{kj} : A_{kj}

Entonces $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ y $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

10

Planificación de actividades utilizando programación dinámica

Si $c[i, j]$ es el tamaño de una solución óptima para S_{ij} , entonces $C[i, j] = 1 + c[i, k] + c[k, j]$.
 Como la actividad a_k es desconocida, entonces tenemos la siguiente relación:

$$c[i, j] = \begin{cases} 0 & \text{si } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{si } S_{ij} \neq \emptyset \end{cases}$$

El array c se puede rellenar de manera bottom-up, o mediante un algoritmo recursivo utilizando memoización.

11

Planificación de actividades utilizando una estrategia voraz

Supongamos que utilizamos una estrategia diferente. Si tuviéramos que escoger la primera tarea para ser planificada, ¿cuál escogeríamos?

La *estrategia voraz*, la que nos sugiere la intuición, consiste en escoger primero aquella tarea que deje más tiempo para planificar otras diferentes, o lo que es lo mismo escogeríamos la que termina antes, a_1 . Si hubiera varias que terminan a la vez, se escogería cualquiera de ellas.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

12

Planificación de actividades utilizando una estrategia voraz

Una vez escogida a_1 , ¿cómo seleccionamos el resto de tareas?

1. Todas deben comenzar después de f_1 .
2. Definimos $S_1 = \{a_k | s_k \geq f_1\}$
3. Escogemos la tarea que termine antes del conjunto S_1 , por ejemplo, a_m .
4. Tomamos $S_m = \{a_k | s_k \geq f_m\}$ y seguimos seleccionando la tarea que termine antes, hasta que encontremos un conjunto S_n que sea vacío, en cuyo caso hemos terminado.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

13

Planificación de actividades utilizando una estrategia voraz

¿Cómo sabemos que esta estrategia nos lleva a una solución óptima?

Teorema Sea un subproblema no vacío S_k , y sea a_m una de las actividades de S_k que termina antes. Entonces a_m está incluida en algún subconjunto de tamaño máximo actividades compatibles de S_k

Luego nuestra intuición es correcta, y para encontrar una solución óptima para este problema no hace falta utilizar **programación dinámica**.

14

Demostración del Teorema (1 de 3)

- Demostración por inducción sobre el numero de actividades seleccionadas.
 - Consideremos las actividades numeradas de 1 a n .
- **Base de la inducción:**
 - Sea $\text{cardinal}(S)=1$. Entonces $S = \{1\}$.
 - Si OPT es una solución optimal cualquiera y k es la “primera” actividad de OPT (o sea $f_k = \min\{f_j : j \in \text{OPT}\}$), pueden ocurrir dos casos: $k = 1$ ó $k \neq 1$.

15

Demostración del Teorema (2 de 3)

- Si $k = 1$, FIN, pues $S \subseteq \text{OPT}$.
- Si $k \neq 1$, $(\text{OPT}-\{k\}) \cup \{1\}$ es también una solución optimal ya que tiene el mismo numero de actividades que OPT, y además son actividades compatibles puesto que $f_1 \leq f_k$.
- **Hipótesis de inducción:** Sea $\text{cardinal}(S) = m - 1$ y sea S parte de una solución optimal OPT y construido según el criterio de selección.
 - Imaginemos OPT ordenado por tiempo de terminación de sus actividades: $\text{OPT} = \{a_1, \dots, a_{m-1}\} \cup \{b_m, \dots, b_n\}$
 - Por el criterio de selección, las actividades de S han de ser las primeras, es decir $S = \{a_1, \dots, a_{m-1}\}$

16

Demostración del Teorema (3 de 3)

- Sea b_m la primera de OPT-S (o sea, $f_{b_m} = \min\{f_j : j \in \text{OPT-S}\}$).
- Sea i la actividad elegida por el criterio de selección tras haber seleccionado el conjunto S , formalmente

$$f_i = \min\{f_j : j \text{ es compatible con } S\}.$$
- $f_i \leq f_{b_m}$, puesto que todas las actividades de $\text{OPT-S} = \{b_m, \dots, b_n\}$ son compatibles con S .
- Entonces, $(\text{OPT-S}) \cup \{i\}$ es solución optimal pues tiene el mismo cardinal que OPT y la actividad i es compatible además con $(\text{OPT-S}) - \{b_m\}$.
- Por tanto, la selección voraz m -ésima nos lleva a un conjunto de seleccionados incluido en una solución optimal.

17

Planificación de actividades algoritmo voraz recursivo

```

public static
List<Integer> planActividades(int[] s, int[] f, int k, int n) {
    // s y f son los arrays con el inicio y final
    // de cada actividad. Se añade una actividad 0
    // con fin igual a 0. Por lo que inicialmente hay
    // que considerar todas las actividades
    int m = k+1;
    while (m <= n && s[m] < f[k])
        m++;
    if (m <= n) {
        List<Integer> lista = planActividades(s, f, m, n);
        lista.add(m);
        return lista;
    } else
        return new ArrayList<Integer>();
}

La llamada inicial es planActividades(s, f, 0, n)

```

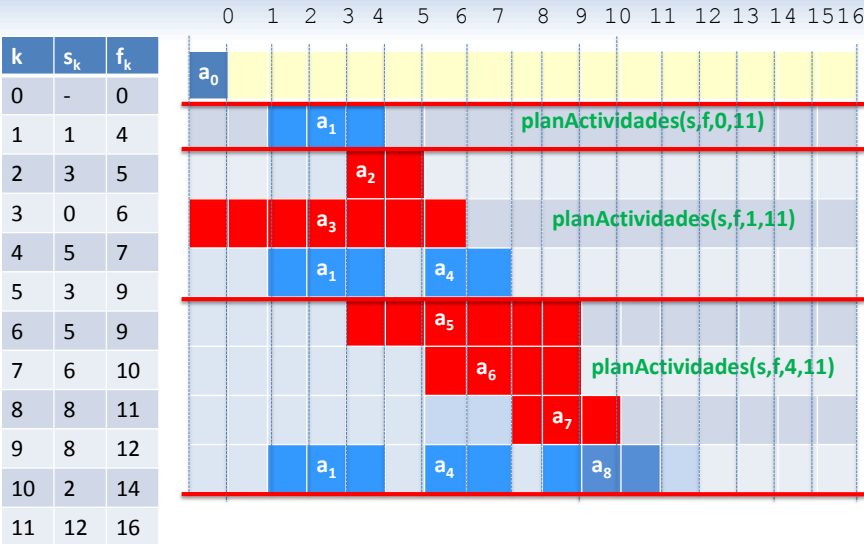
18

Algoritmo voraz recursivo Complejidad

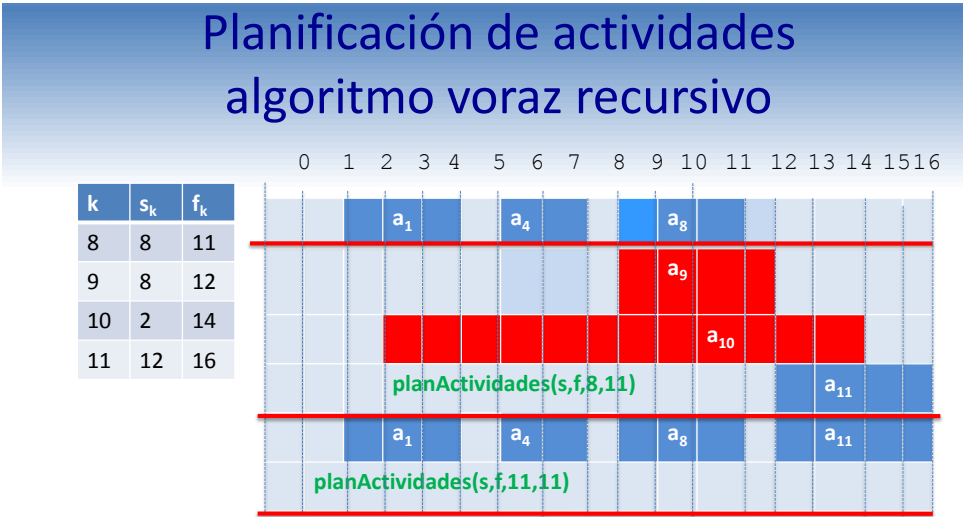
- Suponiendo que las actividades están inicialmente ordenadas en función de su terminación, la ejecución de *planActividades(s,f,0,n)* es de orden $\Theta(n)$
- En cada llamada recursiva, cada actividad es examinada exactamente una vez.

19

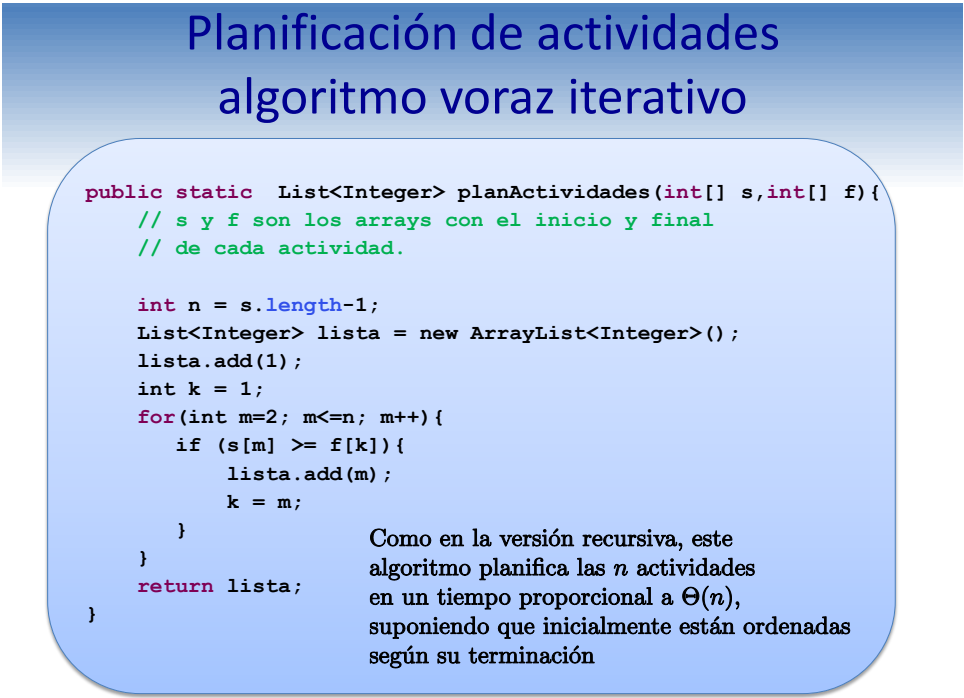
Planificación de actividades algoritmo voraz recursivo



20



21



22

Árbol de recubrimiento mínimo (MST)

Minimun spanning tree

Definición: Árbol de recubrimiento

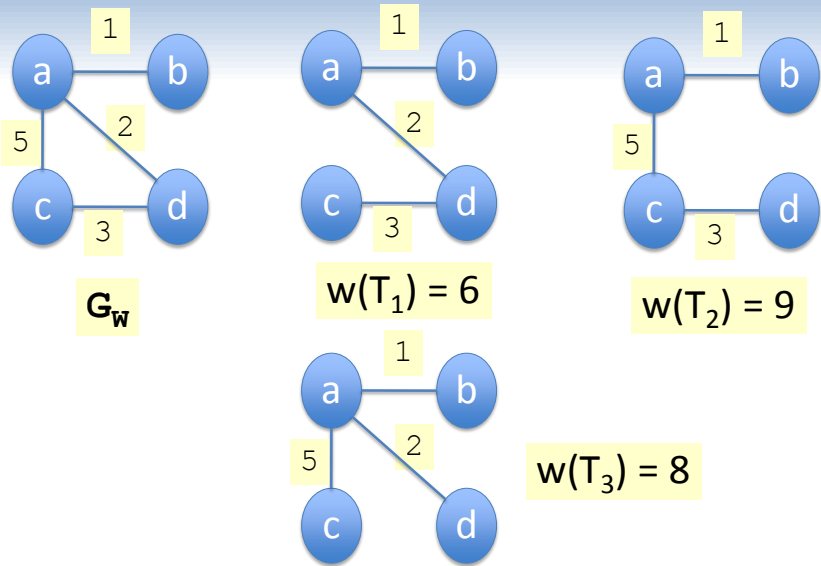
Sea $G = (V, E)$ un grafo conexo no dirigido.
 Un **árbol de recubrimiento** de G es un árbol (no cíclico) que contiene todos los vértices de G .

Definición: Árbol de recubrimiento mínimo

Sea $G_W = (V, E)$ un grafo conexo, *etiquetado* y no dirigido.
 Supongamos que para toda arista $e \in E$, $w_e \in W$ es el peso (o etiqueta) de e .
 Un **árbol de recubrimiento mínimo** de G_W es un árbol de recubrimiento de G_W tal que la suma de los pesos de las aristas es *mínimo*.

23

Ejemplo de MTS



24

Búsqueda de un MST

- Si intentamos utilizar una **estrategia exhaustiva** para construir un MST nos encontramos con dos problemas:
 - El **número de árboles de recubrimiento** de un grafo **crece exponencialmente** con el tamaño del grafo.
 - **Generar todos los árboles** de recubrimiento de un grafo **es** más **difícil** que encontrar un MST.
- Veremos dos algoritmos:
 - El algoritmo de Prim (1957)
 - El algoritmo de Kruskal (1956)

25

Algoritmo de Prim

- **Construye un MST de forma incremental**, empezando con un árbol mínimo (un solo vértice) y expandiéndolo hasta que contenga todos los vértices del grafo.
 - Escogemos **un vértice de forma aleatoria**
 - El árbol actual se expande de manera voraz añadiéndole el vértice más cercano de entre los no incluidos. El **vértice más cercano** es el que se une a uno de los del grafo con una arista de peso mínimo
 - El proceso se repite hasta que el árbol contiene todos los vértices del grafo.

26

Algoritmo de Prim (pseudocódigo)

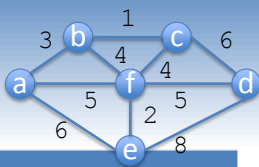
```

Algoritmo Prim(G)
    // G=(V,E) es el grafo de entrada
    // la salida es el conjunto de aristas
    // ET que constituyen un árbol de recubrimiento
    // mínimo de G.
    VT={v0}
    ET=∅
    para i = 1 hasta |V|-1 hacer
        encontrar la arista con peso mínimo
        e* = (v*,u*) de entre todas las aristas
        (v,u) tales que v∈VT, y u∈V-VT

        VT = VT ∪ {u*}
        ET = ET ∪ {e*}
    Devolver ET
    
```

27

Algoritmo de Prim Ejemplo de ejecución



Vértices del árbol	Vértices que faltan por añadir	Ilustración
a	b(a,3) , c(-,∞), d(-,∞), e(a,6), f(a,5)	
a, b(a,3)	c(b,1) , d(-,∞), e(a,6), f(b,4)	

28

Algoritmo de Prim Ejemplo de ejecución

Vértices del árbol	Vértices que faltan por añadir	Ilustración
a, b(a,3) c(b,1)	d(c,6), e(a,6), f(b,4)	
a, b(a,3) c(b,1) f(b,4)	d(f,5), e(f,2)	

29

Algoritmo de Prim Ejemplo de ejecución

Vértices del árbol	Vértices que faltan por añadir	Ilustración
a, b(a,3) c(b,1) f(b,4) e(f,2)	d(f,5)	
a, b(a,3) c(b,1) f(b,4) e(f,2) d(f,5)		

30

Algoritmo de Prim: Corrección

Teorema Dado G un grafo conexo no dirigido, el árbol T de recubrimiento generado por el **algoritmo de Prim** es un árbol de recubrimiento mínimo.

Demostración Veamos por inducción que cada subárbol $T_i, 0 \leq i \leq n - 1$, generado por el algoritmo de Prim es parte de algún MST.
Si demostramos esto, ya tenemos demostrado el teorema, ya que T_{n-1} es un MST al contener todos los vértices de G .

- T_0 tiene sólo un vértice, luego forma parte de cualquier MST.
- Supongamos que T_{i-1} es parte de un MST, tenemos que probar que T_i es también parte de un MST.
(sigue...)

31

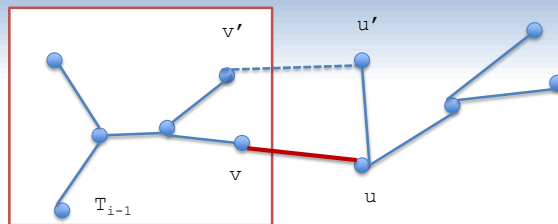
Algoritmo de Prim: Corrección

Demostración (sigue...)

- Por reducción al absurdo, supongamos que ningún MST contiene a T_i . En este caso, existe una arista con peso mínimo $e_i = (v, u)$, que el algoritmo de Prim ha añadido a T_{i-1} para construir T_i , que no está en ningún MST.
Sea T un MST de G . Como e_i tiene peso mínimo y no está en T , podemos deducir que v y u no están directamente conectados en T . Pero como T es un árbol de recubrimiento, debe existir un camino de v a u a través de otros vértices.
Por lo tanto, si añadimos la arista e_i a T obtenemos un grafo con un ciclo.
(sigue...)

32

Algoritmo de Prim: Corrección



Demostración (sigue...)

- Debe existir además otra arista (v', u') que conecta un vértice de T_{i-1} con otro que no está en T_{i-1} . Como el algoritmo de Prim añade la arista con menor peso en cada iteración, el peso de (v, u) tiene que ser menor o igual que el de (v', u') .
Por lo tanto, si eliminamos (v', u') del ciclo obtenemos un MST de G , con peso menor o igual que T , que contiene a e_i , lo que contradice la hipótesis.

33

Algoritmo de Prim: Complejidad

- La complejidad del algoritmo de Prim depende de las estructuras de datos empleadas para almacenar el grafo.
- En cada iteración del algoritmo, siempre se escoge la arista más cercana al árbol construido, por lo tanto, es conveniente almacenar el grafo considerando estas distancias como prioridades.
- La estructura más adecuada para ello, es el **montículo** o heap.

34

Algoritmo de Prim: Complejidad

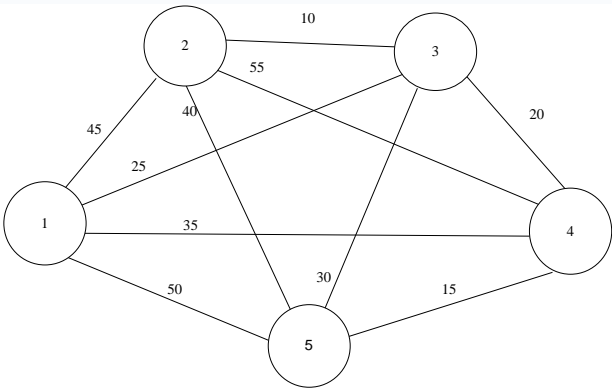
- Un montículo es un árbol binario, en el que cada nodo es menor (mayor) o igual que sus hijos.
 - Las operaciones de inserción, extracción y modificación de la prioridad de un elemento son de complejidad $O(\log n)$ (n es el número de vértices)
 - La construcción inicial tiene complejidad $O(n \log n)$ ya que hay que insertar $n-1$ nodos
 - Posteriormente, se hacen $n-1$ extracciones y $|E|$ modificaciones de la prioridad de los elementos
 - Por lo tanto, la complejidad es $O((n+|E|)\log n)$, y como $|E| \geq n$, equivale a $O(|E|\log n)$

35

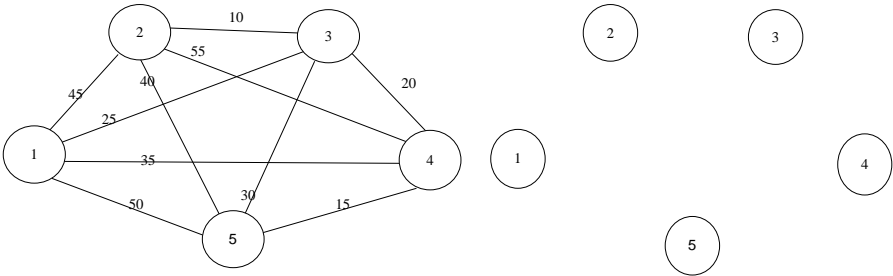
Algoritmo de Kruskal

- **Filosofía del algoritmo.**
- **Funciones:**
 - **Candidatos.** Cada una de las aristas de E .
 - **Función Selección.** Selecciona aquella arista de mínimo coste.
 - **Función factible.** Será factible aquella arista que une dos componentes conexas distintas.
 - **Función solución.** Se tendrá una solución si queda una sola componente conexa.
 - **Añadir a la solución.** Añadir la arista seleccionada.

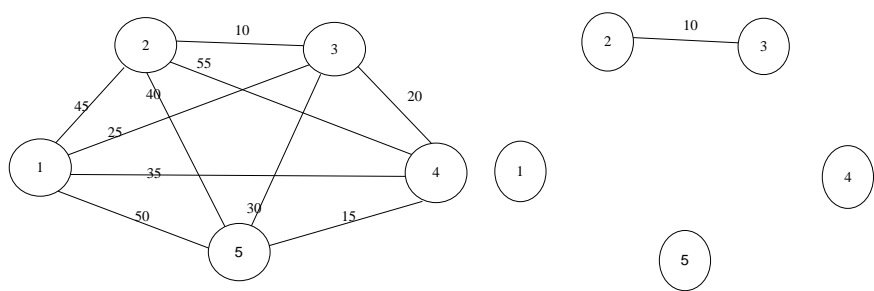
Algoritmo de Kruskal



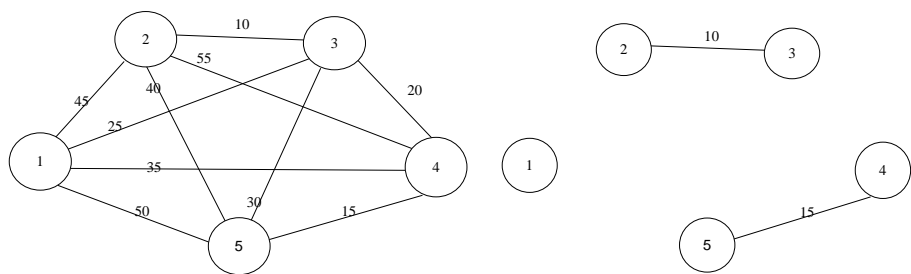
Algoritmo de Kruskal



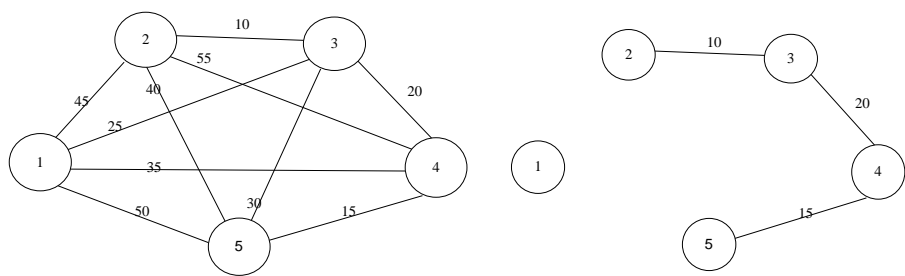
Algoritmo de Kruskal



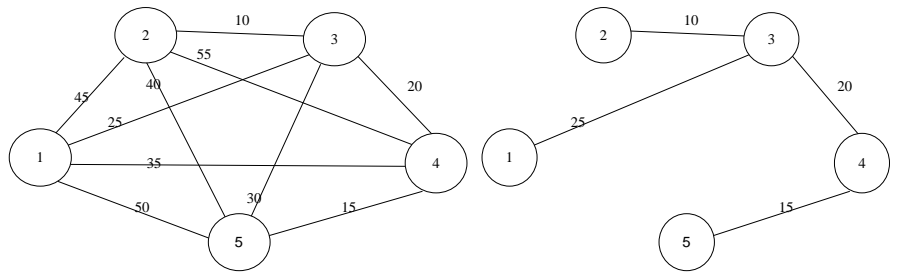
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Esquema del Algoritmo de Kruskal

```

función Kruskal( $G=\langle N, E \rangle$ : grafo; peso:  $E \rightarrow \mathbb{R}^*$ ): conjunto_aristas
    // inicialización
    ordenar E según peso creciente
     $n \leftarrow \#N$            // n es el número de elementos (vértices) de N
     $T \leftarrow \emptyset$     // T mantiene las aristas del árbol de recubrimiento mínimo
    iniciarpartición (P); // crea n particiones en P, cada uno con un elemento
                           distinto de N
    // bucle voraz
    repetir
         $\{u, v\} \leftarrow$  la arista mas corta aún no considerada
         $uconj \leftarrow$  obtenerparticiónP(u)
         $vconj \leftarrow$  obtenerparticiónP(v)
        si  $uconj \neq vconj$  entonces
            fusionar( $uconj, vconj, P$ )
             $T \leftarrow T \cup \{\{u, v\}\}$ 
        endsi
    hasta  $\#T = n - 1$ ;
    devolver T
finfunción

```

Detalles de implementación

*Complejidad (en un grafo con n nodos y e aristas):

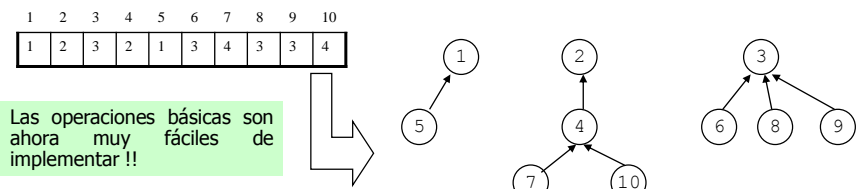
-Seleccionar es de orden $O(1)$ una vez ordenadas las aristas, y tanto iniciarparticion como obtenerparticion son de orden $O(n)$.

-Fusionar es de orden $O(1)$.

-La ordenación de los arcos puede realizarse en un tiempo del orden de $O(e \log e)$, siendo e el número de arcos del grafo. Como se verifica que $(n-1) \leq e \leq n(n-1)/2$ por tratarse de un grafo conexo, su orden es $O(e \log n)$.

Detalles de implementación

- Representación de componentes conexas: podemos representar cada componente como un árbol con raíz, en el cual cada nodo contiene un único puntero que apunta a su padre
- Se puede usar un vector de n posibles nodos donde $comp[i]=i$ si el nodo i es la raíz y la partición del árbol. Si $comp[i] = j$, y $j \neq i$, entonces j es el padre de i en algún árbol.



Detalles de implementación

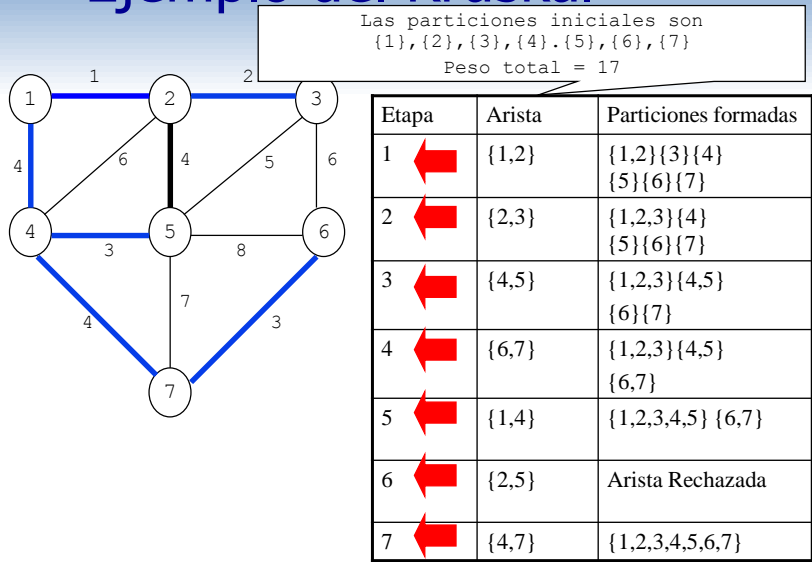
```

función obtenerpartición(x: vertice)
// Busca la partición que contiene al elemento x
  r ← x
  mientras comp [r] ≠ r hacer r ← comp [r]
finmientras
  devolver r
finproc
    
```

```

Proc fusionar(a,b: comp)
// fusiona los conjuntos cuyas
particiones son a y b, asumimos a ≠ b
  Si a < b entonces comp[b] ← a
  En otro caso comp [a] ← b
  Finsi
finproc
    
```

Ejemplo del Kruskal



Caminos más cortos desde un origen

- Dado un **grafo conexo y etiquetado con pesos**, y un vértice **s**, el problema consiste en **encontrar los caminos más cortos que conectan a s con el resto de vértices del grafo**.
- Hay varias soluciones a este problema, entre ellas **el algoritmo de Floyd**, que encuentra el camino más corto entre cada par de vértices del grafo.
- La mejor solución conocida hasta ahora se llama **el algoritmo de Dijkstra**, que puede aplicarse sólo cuando los **pesos son no negativos**.

El algoritmo de Dijkstra

- Intuitivamente, el algoritmo de Dijkstra realiza $|V|-1$ iteraciones:
 - En la primera iteración encuentra el vértice más cercano a s
 - En la segunda, el vértice segundo más cercano a s
 - ...
 - En la i -ésima iteración, encuentra el i -ésimo vértice más cercano a s .
 - ...

49

El algoritmo de Dijkstra

- Supongamos que hemos realizado i iteraciones, y hemos construido el árbol T_i con los i vértices más cercanos a s , para construir T_{i+1} :
 - Creamos el conjunto F de todos los vértices adyacentes a T_i , es decir, los que no están en T_i pero que comparten una arista con algún vértice de T_i
 - Para cada $u \in F$, calculamos la distancia de u a s , y escogemos el que esté más cerca
 - si hay varios con la misma distancia más corta, cualquiera de ellos

50

El algoritmo de Dijkstra

- Para calcular la distancia de cada $u \in F$ a s mantenemos una cola de prioridades Q con los vértices que quedan por encontrar, ordenados por su distancia a s . Q se actualiza en cada iteración.
- Los elementos de Q tienen la siguiente forma $v(w,d)$, donde
 - v es uno de los vértices que falta por encontrar
 - w es un vértice cuya distancia mínima a s ya se ha calculado
 - d es la distancia mínima de v a s , que se ha calculado como la distancia de v a w , más la distancia de w a s .
 - la clave del algoritmo está en que esta última distancia ya ha sido calculada por el algoritmo en la iteración anterior, y no hay que volver a calcularla.

51

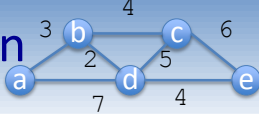
Algoritmo de Dijkstra(pseudocódigo)

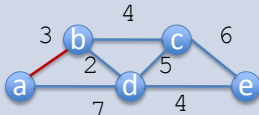
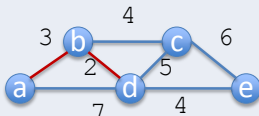
```
Algoritmo Dijkstra(G,s)
  // G=(V,E) es el grafo de entrada
  // s es uno de los vértices de V
  // la salida es un conjunto de aristas
  //  $E_T$  con el árbol de las distancias mínimas a s
   $E_T = \{s(-,0)\}$ 
   $S = V - \{s\}$ 
   $Q = \{v(s,d) \mid (s,v) \in E, w(s,v)=d\}$ 
  while (S  $\neq \emptyset$ ) {
     $u(v,d) = \text{extrae\_min}(Q)$ ;
     $E_T = E_T \cup \{u(v,d)\}$ ;  $S = S - \{u\}$ ;
    para cada  $x \in S$  tal que  $(u,x) \in E_T$ 
      si  $(x(-,d') \in Q, d + w(u,x) < d')$ 
         $Q = Q - \{x(-,d')\} \cup \{x(u, d + w(u,x))\}$ 
  }
  Devolver  $E_T$ 
```

52

Algoritmo de Dijkstra

Ejemplo de ejecución

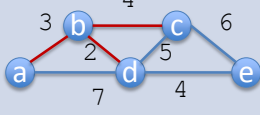
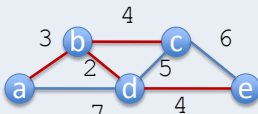


Vértices del árbol	Cola de prioridades	Ilustración
a(-,0)	b(a,3),d(a,7),c(-,∞),e(-,∞)	
a(-,0), b(a,3)	d(b,5),c(b,7),e(-,∞)	

53

Algoritmo de Dijkstra

Ejemplo de ejecución

Aristas del árbol	Lista ordenada de aristas	Ilustración
a(-,0), b(a,3), d(b,5)	c(b,7),e(d,9)	
a(-,0), b(a,3), d(b,5), c(b,7)	e(d,9)	

54

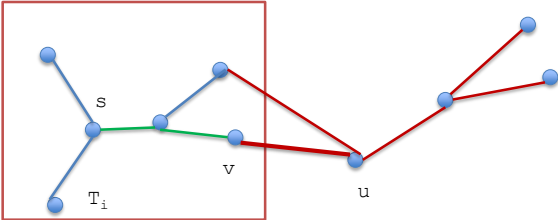
Algoritmo de Dijkstra Ejemplo de ejecución

Aristas del árbol	Lista ordenada de aristas	Ilustración
a(-,0), b(a,3), d(b,5), c(b,7), e(d,9)		

55

El algoritmo de Dijkstra: corrección

- Si se ordenan los caminos óptimos desde **s** hacia los otros **n-1** nodos, el **i**-ésimo camino menor se encuentra en la iteración **i**. Puede demostrarse por inducción:
 - En la primera iteración se encuentra el camino más corto
 - Suponiendo que en la iteración **i-1** se han encontrado los **i-1** caminos más cortos, el **i**-ésimo vértice **u** es el más cercano a **s** a través de los vértices explorados. Cualquier otro vértice es más lejano, por lo que no puede haber un camino más corto a **u**.



56

El algoritmo de Dijkstra: complejidad

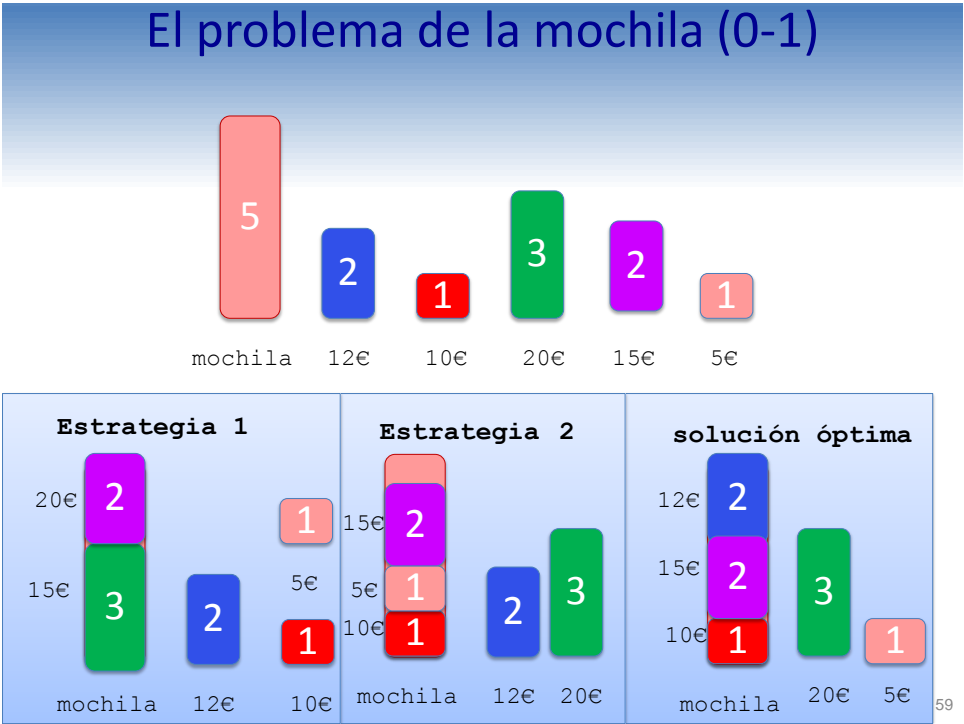
- Dada la similitud con el algoritmo de **Prim**, pueden emplearse estructuras de datos parecidas para la implementación del algoritmo de **Dijkstra**, por ejemplo, un montículo para mantener ordenados los vértices por la distancia al origen.
- Por lo tanto, **Dijkstra** tiene una complejidad del orden $O(|E| \log n)$. Si se desea calcular el camino óptimo entre cualquier par de nodos, puede repetirse el proceso n veces desde todos los orígenes posibles. La complejidad es, entonces, $O(|E| n \log n)$. Si el grafo no es muy denso, este algoritmo es mejor que el de **Floyd** de complejidad $O(n^3)$

57

El problema de la mochila (0-1) ¿Qué estrategia?

- Supongamos que tenemos una mochila de capacidad W , y n objetos de peso y valor (p_i, v_i) .
- Podemos utilizar dos estrategias para seleccionar los objetos a introducir en la mochila:
 - Seleccionar siempre el objeto de **mayor valor**
 - Seleccionar siempre el objeto de **menor peso**

58



El problema de la mochila (0-1) ¿Qué estrategia?

- Seleccionar siempre el objeto de **mayor valor**
- Seleccionar siempre el objeto de **menor peso**

NINGUNA DE LAS DOS FUNCIONA

Esta sería la respuesta a este problema, mediante los contraejemplos anteriores.

El problema de la mochila

Versión continua

- Consideremos la variante del problema de la mochila en la que se puede introducir **una fracción de cada objeto**, es decir, las soluciones son vectores $\langle x_1, \dots, x_n \rangle$ donde $x_i \in [0, 1]$.
- Esta versión satisface el principio de subestructura óptima (la demostración es similar a la del caso discreto).

61

El problema de la mochila

Versión continua

- El algoritmo voraz que ordena decrecientemente los objetos en función de la razón valor/peso y los incluye en orden hasta llenar la mochila es óptimo.
- Por ejemplo, suponiendo una mochila de capacidad $W = 5 \text{ kg}$

Objeto	Peso	Valor	Razón
1	1	6€	6€/kg
2	2	10€	5€/kg
3	3	12€	4€/kg

- Se incluye o1, $V = 6\text{€}$, $W = 4$
- Se incluye o2, $V = 16\text{€}$, $W = 2$
- Se incluyen 2/3 de o3, $V = 24\text{€}$, $W = 0$

62

Teorema para mochila continua (1 de 5)

Calcular un vector (x_1, \dots, x_n) de valores $0 \leq x_i \leq 1$ tal que maximice $\sum_{i=1}^n x_i v_i$ restringido a que $\sum_{i=1}^n x_i p_i \leq M$

Entonces, una solución optimal es de la forma siguiente $(1, \dots, 1, x_k, 0, \dots, 0)$ con $x_k \neq 1$, supuesto que hemos ordenado los artículos de manera que

$$v_1/p_1 \geq \dots \geq v_n/p_n$$

(No obstante, puede haber soluciones igualmente óptimas que no estén construidas así)

63

Teorema para mochila continua (2 de 5)

Suponemos que $v_1/p_1 \geq \dots \geq v_n/p_n$

CASO (1) El artículo 1 no cabe completo en la mochila: $q_1 p_1 = M$ con $q_1 < 1$.

Supongamos otra solución $OPT = (x_1, \dots, x_n)$ con $x_1 < q_1$. Entonces la solución Voraz = $(q_1, 0, \dots, 0)$, aporta tanto valor como OPT o más.

Demostración:

$$q_1 p_1 = M = \sum_{i=1}^n x_i p_i$$

$$\text{- Entonces, } (q_1 - x_1) p_1 = \sum_{i=2}^n x_i p_i$$

Ahora multiplicamos los dos miembros por v_1/p_1 :

$$(q_1 - x_1) v_1 = \left(\sum_{i=2}^n x_i p_i \right) v_1/p_1 \geq \sum_{i=2}^n (x_i p_i) v_i/p_i = \sum_{i=2}^n x_i v_i$$

Despejando: $q_1 v_1 \geq \sum_{i=1}^n x_i v_i$, luego Voraz \geq OPT

64

Teorema para mochila continua (3 de 5)

CASO (2) El artículo 1 cabe completo en la mochila.

Supongamos una solución optimal $OPT = (x_1, \dots, x_n)$.

Sea Voraz = $(1, y_2, \dots, y_n)$ con $\sum_{i=2}^n y_i \leq \sum_{i=2}^n x_i$

$$p_1 + \sum_{i=2}^n y_i p_i = M = \sum_{i=1}^n x_i p_i$$

$$\text{Entonces } (1 - x_1)p_1 = \sum_{i=2}^n (x_i - y_i) p_i$$

- Ahora multiplico todo por (v_1/p_1) y queda:
- $(1 - x_1)v_1 = (\sum_{i=2}^n (x_i - y_i)p_i) (v_1/p_1) \geq \sum_{i=2}^n (x_i - y_i)p_i (v_i/p_i)$

65

Teorema para mochila continua (4 de 5)

$$(1 - x_1)v_1 \geq \sum_{i=2}^n (x_i - y_i) v_i$$

Entonces resulta :

$$v_1 + \sum_{i=2}^n y_i v_i \geq x_1 v_1 + \sum_{i=2}^n x_i v_i$$

Luego,

$$\text{valor (VORAZ)} \geq \text{valor(OPT)}$$

66

Teorema para mochila continua (5 de 5)

Repitiendo el proceso (previa ordenación descendente de los objetos por su relación (valor/peso)) siempre tendremos una solución optimal tomando los objetos completos que podamos y la fracción necesaria del último.

Está es la solución optimal voraz que tiene la forma $(1, \dots, 1, x_k, 0, \dots, 0)$ con $x_k \neq 1$.

67

Algoritmo mochila continua

- función Mochila Continua ($A(1..n, 1..3), M$) return vector
// en la columna 1 de A están el valor, en la 2 el peso y en la 3 el cociente
// Primer paso: calcular los cocientes
// Segundo paso: ordenar A decreciente por la tercera columna
peso= 0
i= 1 //representa el numero del artículo a tratar
WHILE $i \leq n$ && $(\text{peso} + A[2,i]) \leq M$ DO
 $x(i)= 1$
 peso= peso + p(i)
 $i= i + 1$
END-WHILE
// SI $i < n$ ENTONCES $\text{peso} + A[2,i] > M$ Y $x(k) = 1, k=1,\dots,i-1$
IF $i \leq n$ THEN $x(i)= (M - \text{peso})/A[2,i]$
PARA $k= i + 1 \dots n$ HACER $x(k)= 0$
RETURN x

68

El problema de la mochila

Criterio del cociente en versión (0,1)

No funciona

- En el ejemplo de debajo, con mochila de $M= 5$ kg, la técnica voraz no funciona. Tomaríamos el objeto 1, y la mochila tiene valor 44, pero el óptimo es tomar los objetos 2 y 3, que dan un valor de 50.

Objeto	Peso	Valor	Razón
1	4	44€	11€/kg
2	2	20€	10€/kg
3	3	30€	10€/kg

69

El problema del viajante de comercio

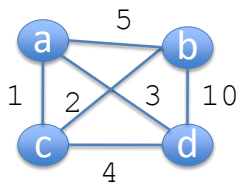
(TSP: travelling salesman problem)

- Supón n ciudades y una matriz $D_{n \times n} = \{d_{ij}\}$ que representa la distancia entre la ciudad i , y la j . TSP consiste en buscar un ciclo hamiltoniano (un camino que pase por todas las ciudades una única vez y vuelva al origen) de longitud mínima.
- TSP es un problema NP-hard clásico. No existen métodos generales de resolución óptima con garantías de eficiencia.
- Para este tipo de problemas, las heurísticas son el único mecanismo práctico de resolución para instancias de gran tamaño.

70

El problema del viajante de comercio (TSP: travelling salesman problem)

- Podemos considerar dos heurísticas
 - seleccionar siempre **el vecino más cercano**: se comienza en una ciudad cualquiera, y nos desplazamos sucesivamente a la ciudad no visitada más cercana en cada momento.

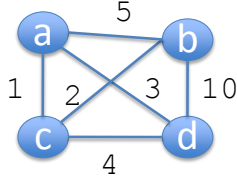


- En este ejemplo, empezando por a, obtendríamos la secuencia <a,c,b,d,a> con una distancia de 16, y la óptima es <a,d,c,b,a> que mide 14,

71

El problema del viajante de comercio (TSP: travelling salesman problem)

- Podemos considerar dos heurísticas
 - seleccionar siempre **el camino más corto**: Se ordenan los caminos por orden y se van añadiendo mientras que no se violen las restricciones.



- En este ejemplo, se añadiría (a,c), (c,b), (d,a) y finalmente (b,a) que no da el camino más corto

72

Referencias

- *Estructuras de datos y métodos algorítmicos. Ejercicios resueltos.* Martí Oliet, Narciso; Ortega Mallén, Yolanda; Verdejo López, José Alberto. **Editorial:** Pearson Alhambra
- *Enlaces:*
 - <https://www.geeksforgeeks.org/greedy-algorithms/>
 - <https://www.techiedelight.com/Tags/greedy/>
- *The Design & Analysis of Algorithms.* A. Levitin. Ed. Adison-Wesley
- *Introduction to Algorithms.* T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press
- *Técnicas de Diseño de Algoritmos.* Rosa Guerequeta y Antonio Vallecillo. Servicio de Publicaciones de la UMA

73

El problema de la mochila

Versión continua

- *Si resolvemos el problema suponiendo que los objetos son discretos, y utilizamos la heurística, “menor peso” nos saldría una solución de valor 16, cuando la solución óptima tiene valor 22*

Objeto	Peso	Valor	Razón
1	1	6€	6€/kg
2	2	10€	5€/kg
3	3	12€	4€/kg

74

El problema de la mochila

Versión continua

- En general, la heurística voraz no proporciona la solución óptima a la versión discreta del problema.
- Además, no hay ninguna garantía de que la solución obtenida esté dentro de un margen de error.

75