Programación II

Tema 5. Algoritmos voraces, heurísticos y aproximados

5.1 Introducción

Los *algoritmos voraces* (*Greedy*) se utilizan típicamente para resolver *problemas de optimización*:

- minimizar o maximizar, bajo determinadas condiciones, el valor de una función del tipo: $f(x_1,x_2,..,x_n)=c_1x_1+c_2x_2+..+c_nx_n$

La solución se va construyendo en etapas:

- en cada etapa se añade un nuevo elemento a la solución parcial
  - el que nos parece el mejor candidato en ese momento
- las decisiones tomadas nunca se revisan
  - *voracidad*: se consume el mejor elemento lo antes posible sin pensar en futuras consecuencias
- al final de cada etapa se verifica si la solución parcial ya constituye una solución total para el problema

Ejemplo: “dar cambio utilizando el mínimo número de monedas”

Algoritmo voraz para "dar cambio"

Solución: vamos incluyendo secuencialmente la moneda de mayor valor posible de forma que todavía no superemos la cantidad a devolver

```
método daCambio(cent : entero) retorna monedas
   cambio := ∅
   suma := 0
   mientras suma != cent hacer
      x := mayor moneda que verifique suma+x ≤ cent
      si no existe x entonces
         retorna no hay solución
      fsi
      añade x a cambio
      suma := suma + x
   fhacer
   retorna cambio
fmétodo
```


5.2 Características generales

Los algoritmos voraces suelen tener las siguientes propiedades:

- **Se trata de resolver un problema de forma óptima**
  - **normalmente se imponen algunas restricciones**
- **Para construir la solución se dispone de un conjunto de candidatos**
- **Durante el desarrollo del algoritmo se van formando dos conjuntos: los candidatos seleccionados y los rechazados**
- **Función *solución*: comprueba si los candidatos seleccionados hasta el momento constituyen una solución al problema**
- **Función *factible*: comprueba si un conjunto de candidatos podría llegar a convertirse en solución**
- **Función de *selección*: selecciona en cada etapa el mejor candidato para incluirle en la solución parcial**

Esquema genérico de un algoritmo voraz

```
método voraz retorna conjunto
  C := candidatos
  S := ∅ // irá guardando la solución
  mientras no_es_solución(S) hacer
    si C == ∅ entonces
      retorna no hay solución
    fsi
    x := selecciona(C)
    elimina x de C // no se vuelve a considerar
    si factible(S+x) entonces
      añade x a S
    fsi
  fhacer
  retorna S // retorna la solución alcanzada
fmétodo
```

Análisis del algoritmo "dar cambio"

- Función a optimizar (en este caso minimizar):

  $f(x_1,x_2, ..,x_n)=x_1+x_2+..+x_n$

  Verificándose que $\Sigma x_i v_i=c$

  - donde: x_i es el número de monedas de la clase "i", v_i su valor y c la cantidad a devolver

- Candidatos:
  - Las monedas disponibles (p.e. 50, 20, 10, 5, 2 y 1 céntimos)
- Función solución: $\Sigma x_i v_i=c$
- Función de selección: la mayor de las monedas que verifican $\Sigma x_i v_i \leq c$
- Función factible: comprobar si la función de selección encuentra alguna moneda

5.3 Eficiencia de los algoritmos voraces

Por lo general los algoritmos voraces son *muy eficientes*

Su eficiencia depende de:

- el número de iteraciones, que viene determinado por el tamaño de la solución
- la eficiencia de las funciones "solución", "factible" y "selección"
  - "solución" y "factible" suelen ser operaciones de tiempo constante o dependientes de la longitud de la solución
  - "selección" depende de la longitud del conjunto de candidatos
    - muchas veces es la causante de la mayor parte de la complejidad del algoritmo
    - en ocasiones convendrá preordenar el conjunto de candidatos, para que la función de selección sea más rápida

Eficiencia del algoritmo "dar cambio"

		Complejidad
Iteraciones	cuando n es muy grande el número de monedas tiende a n	O(n)
solución	`suma == cent`	O(1)
factible	`existe x`	O(1)
selección	`mayor moneda que verifique suma+x ≤ cent`	O(m)

Donde:

- n es la cantidad que se desea cambiar
- `m` es el número de monedas distintas

Complejidad del algoritmo: $O(n)\cdot O(m) = O(n)$ (ya que n>>m)

5.4 Cuándo usar algoritmos voraces

Los algoritmos voraces suelen ser eficientes y fáciles de diseñar e implementar

Pero no todos los problemas se pueden resolver utilizando algoritmos voraces:

- **a veces no encuentran la solución óptima**
- **o incluso no encuentran ninguna solución cuando el problema sí la tiene**

Ejemplo de problema para él que no funciona el algoritmo voraz:

- **Dar cambio con el antiguo sistema monetario ingles**
  - **corona (30p), florín (24p), chelín (12p), 6p, 3p, penique**
- **Solución del algoritmo voraz para 48p: 30p+12p+6p**
  - **solución óptima: 24p+24p**

Necesidad de prueba matemática

Si queremos utilizar un algoritmo voraz para encontrar soluciones óptimas

- ***debe probarse* que el algoritmo encuentra la solución óptima en todos los casos**
- **la prueba puede ser muy complicada**

Un algoritmo que no garantice encontrar la solución óptima puede ser útil en algunos casos para encontrar una solución aproximada

- **los algoritmos voraces no óptimos a menudo encuentran soluciones bastante aproximadas a las óptimas**
- **hablaremos de ellos al tratar los algoritmos heurísticos y aproximados (apartado 5.8)**

Principio de optimalidad

Un problema verifica el principio de optimalidad si:

- **en una secuencia óptima de decisiones *toda subsecuencia ha de ser también óptima***
- **P.e. el cálculo del camino más corto entre ciudades:**
  - **el camino más corto Santander→Madrid pasa por Burgos**
    - **unión de los caminos más cortos Santander→Burgos, Burgos→Madrid**
- **No pasa igual con el camino más rápido:**
  - **ir rápido en un tramo puede obligar a repostar en el siguiente**

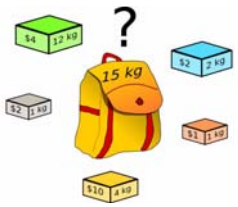
La mayoría de los problemas que verifican este principio pueden resolverse utilizando algoritmos voraces

- **puesto que es posible que exista una función de selección que conduzca a la solución óptima**
- **pero no hay garantía de que se vaya a encontrar esa función (ni siquiera de que exista)**

Dar el cambio cumple el principio de optimalidad

- **la vuelta óptima de 74 cent. es igual a la vuelta óptima de 50 cent. más la vuelta óptima de 24 cent.**
- **pero no es igual a la vuelta óptima de 43 cent. más la vuelta óptima de 31 cent.**
  - **de ahí la importancia de una buena función de selección**
- **el algoritmo para el sistema monetario inglés funcionaría correctamente con una función de selección apropiada**

5.5 El problema de la mochila



Descripción (*the knapsack problem*):

- **disponemos de una mochila que soporta un peso máximo P**
- **existen n objetos que podemos cargar en la mochila, cada objeto tiene un peso p_i y un valor v_i**
- **el objetivo es llenar la mochila maximizando el valor de los objetos que transporta**
  - **suponemos que los objetos se pueden romper, de forma que podemos llevar una fracción x_i ($0 \le x_i \le 1$) de un objeto**

Matemáticamente:

- **Maximizar $\Sigma x_i v_i$ con la restricción $\Sigma x_i p_i \le P$**
- **donde $v_i > 0$, $p_i > 0$ y $0 \le x_i \le 1$ para $1 \le i \le n$**

Aplicaciones: empresa de transporte, ...

El problema cumple el principio de optimalidad

- **intentaremos resolverlo utilizando un algoritmo voraz**
- **el problema será encontrar la función de selección adecuada**

El pseudocódigo de nuestro algoritmo será:

```
método mochila(real[1..n] p, real[1..n] v, real P)
                                    retorna real[1..n]
   peso := 0
   mientras peso < P y quedan objetos hacer
      i := seleccionaMejorObjeto
      si peso + p[i] <= P entonces
         x[i] := 1                  peso := peso + p[i]
      sino
         x[i] := (P-peso)/p[i]      peso := P
      fsi
   fhacer
   retorna x
fmétodo
```

Elección de la función de selección

Podríamos pensar en tres posibles funciones de selección:

- **Seleccionar primero los objetos más ligeros**
- **Seleccionar primero los objetos más valiosos**
- **Seleccionar primero los objetos con mejor relación valor/peso**

Se puede demostrar que la tercera función de selección lleva a una solución óptima (demostración informal):

- **consideramos los objetos ordenados de mayor a menor relación valor/peso**
- **la forma más eficiente de incluir el valor v_1 es incluyendo completamente el objeto `i=1`**
  - **cualquier otra combinación requerirá más peso**
- **el mismo razonamiento se puede aplicar para el objeto v_2 y una mochila de peso $P-p_1$ y así sucesivamente para los demás**

Cálculo del ritmo de crecimiento

Implementación directa:

- **El bucle requiere como máximo n (num. obj.) iteraciones: $O(n)$**
  - **una para cada posible objeto en el caso de que todos quepan en la mochila**
- **La función de selección requiere buscar el objeto con mejor relación valor/peso: $O(n)$**
- **Coste del algoritmo: $O(n)\cdot O(n)=O(n^2)$**

Implementación preordenando los objetos

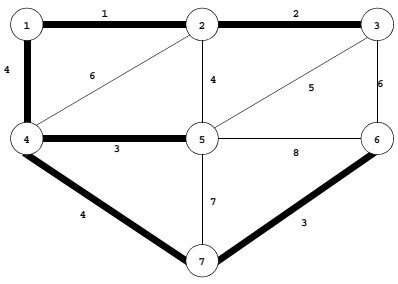
- **Ordena los objetos de mayor a menor relación valor/peso**
  - **existen algoritmos de ordenación $O(n \log n)$**
- **Con los objetos ordenados la función de selección es $O(1)$**
- **Coste del algoritmo: $O(\max(O(n \log n), O(n)\cdot O(1)))=O(n \log n)$**

5.6 Árbol de recubrimiento mínimo

Árbol: grafo sin ciclos y conexo

Árbol de recubrimiento mínimo (*minimum spanning tree*, MST)

- **árbol que interconecta todos los nodos el grafo**
- **de forma que la suma de los pesos de las aristas que lo forman sea lo menor posible**



Posibles aplicaciones:

- **tendido de líneas de teléfono que permita intercomunicar un conjunto de ciudades**

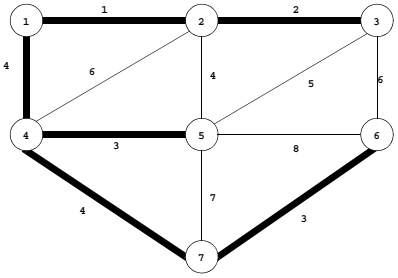
Algoritmo voraz:

- **Función a minimizar: longitud del árbol de recubrimiento**
- **Candidatos: las aristas del grafo**
- **Función solución: árbol de recubrimiento de longitud mínima**
- **Función factible: el conjunto de aristas no contiene ciclos**
- **Función de selección:**
  - **Seleccionar la arista de menor peso que aún no ha sido seleccionada y que no forme un ciclo (Algoritmo Kruskal)**
  - **Seleccionar la arista de menor peso que aún no ha sido seleccionada y que forme un árbol junto con el resto de aristas seleccionadas (Algoritmo Prim)**

Algoritmo de Kruskal

Seleccionar la arista de menor peso que:

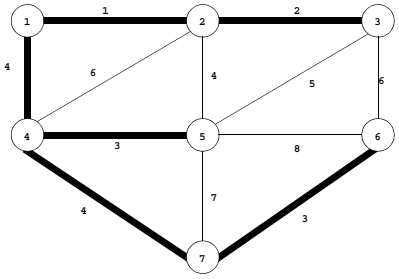
- **aún no haya sido seleccionada**
- **y que no conecte dos nudos de la misma componente conexa**
  - **es decir, que no forme un ciclo**



Paso	Arista considerada	Componentes conexas
-	-	{1} {2} {3} {4} {5} {6} {7}
1	(1,2)	{1,2} {3} {4} {5} {6} {7}
2	(2,3)	{1,2,3} {4} {5} {6} {7}
3	(4,5)	{1,2,3} {4,5} {6} {7}
4	(6,7)	{1,2,3} {4,5} {6,7}
5	(1,4)	{1,2,3,4,5} {6,7}
6	(2,5)	forma ciclo
7	(4,7)	{1,2,3,4,5,6,7}


```
método kruskal(Grafo g) retorna ConjuntoAristas
  {Pre: el grafo es conexo}
  ordena las aristas por pesos crecientes
  crea un conjunto distinto para cada nodo
  mst := conjunto vacío
  // bucle voraz
  hacer
    a := arista de menor peso aún no considerada
    conj1 := conjunto que contiene al nodo a.origen
    conj2 := conjunto que contiene al nodo a.destino
    si conj1 != conj2 entonces
      une conj1 con conj2
      mst.añade(a) // la arista forma parte del mst
    fsi
  mientras mst tiene menos de g.numNodos-1 aristas
  retorna mst
fmétodo
```

Algoritmo de Prim



Comenzando por un nodo cualquiera, seleccionar la arista de menor peso que:

- **aún no haya sido seleccionada**
- **y que forme un árbol junto con el resto de aristas seleccionadas**

Paso	Arista considerada	Nodos unidos (árbol)
-	-	{1}
1	(1,2)	{1,2}
2	(2,3)	{1,2,3}
3	(1,4)	{1,2,3,4}
4	(4,5)	{1,2,3,4,5}
5	(4,7)	{1,2,3,4,5,7}
6	(7,6)	{1,2,3,4,5,6,7}

Algoritmo de Prim (cont.)

```
método prim(Grafo g) retorna ConjuntoAristas
  {Pre: el grafo es conexo}
  mst := conjunto vacío
  B := conjunto con un nodo cualquiera de g.nodos
  // bucle voraz
  mientras B != g.nodos hacer
    a := arista de menor peso aún no considerada que
         cumple que a.origen ∈ B y
         a.destino ∈ g.nodos - B
    mst.añade(a)
    B.añade(a.destino)
  fhacer
  retorna mst
fmétodo
```


Complejidad de los algoritmos Kruskal y Prim

Para un grafo con n nodos y a aristas

- **Kruskal: $\Theta(a \log n)$**
- **Prim: $\Theta(n^2)$**

En un grafo denso se cumple que: $a \rightarrow n(n-1)/2$

- **Kruskal$\rightarrow\Theta(n^2 \log n)$**
- **Prim puede ser mejor (depende del valor de las constantes ocultas)**

En un grafo disperso se cumple que: $a \rightarrow n$

- **Kruskal$\rightarrow\Theta(n \log n)$**
- **Prim es menos eficiente**

5.7 Caminos mínimos (Dijkstra)

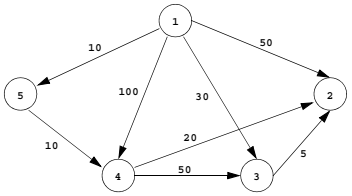
El algoritmo utiliza:

- un array `distancia` con las distancias de los nodos al origen
  - se inicializan todas a ∞ menos la del origen que lo hace a 0
- una `cola` con los nodos no visitados todavía
- un array `previo` que permite almacenar el nudo "previo" de cada nodo por el camino más corto
  - se inicializan todos a `null`

En cada iteración:

- se extrae de la `cola` el nodo u con menor `distancia` al origen
- para cada nodo v adyacente a u:
  - si `distancia[v]` > `distancia[u]` + distancia entre u y v entonces hacemos:
    `distancia[v]` = `distancia[u]` + distancia entre u y v

Ejemplo de desarrollo del algoritmo:



Nodo	Adyacencia	Cola nodo(distancia)	previo				
			1	2	3	4	5
-	-	1(0), 2(∞), 3(∞), 4(∞), 5(∞)	null	null	null	null	null
1	2, 3, 4, 5	5(10), 3(30), 2(50), 4(100)	null	1	1	1	1
5	4	4(20), 3(30), 2(50)	null	1	1	5	1
4	2, 3	3(30), 2(40)	null	4	1	5	1
3	2	2(35)	null	3	1	5	1
2	-	-	null	3	1	5	1


```
método dijkstra(Grafo g, Nodo s) retorna distancia[],previo[]
  para cada u ∈ g.nodos hacer
    distancia[u] = ∞     y    padre[u] = null
  fhacer
  distancia[s] = 0
  conjunto cola = g.nodos
  // bucle voraz
  mientras cola != ∅ hacer
    u = extraerNodoMinDistancia(cola, distancia)
    para cada nodo v ∈ adyacencia[u] hacer
      si distancia[v]>distancia[u]+arista(u,v).peso entonces
        distancia[v] = distancia[u]+arista(u,v).peso
        previo[v] = u
      fsi
    fhacer
  fhacer
fmétodo
```

5.8 Algoritmos heurísticos y aproximados

Existen problemas muy complicados para los que

- **no sabemos como encontrar su solución óptima, o bien,**
- **conocemos el algoritmo que permite encontrar la solución óptima, pero requiere una cantidad excesiva de tiempo**
  - **tiene un tiempo de ejecución no polinómico**

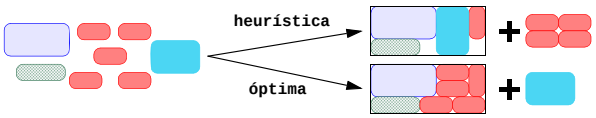
En estos casos utilizaremos algoritmos *heurísticos* o *aproximados*

- **proporcionan soluciones más o menos próximas a la óptima**
- **en un tiempo de ejecución polinómico**
- **permiten resolver ejemplares del problema de tamaños que serían inabordables utilizando el algoritmo óptimo**

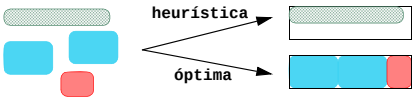
Muchos de los algoritmos heurísticos y aproximados son *algoritmos voraces*

Algoritmos heurísticos

- **Suelen estar basados en una *regla de "sentido común"***
  - **P.e.: para aprovechar al máximo el maletero del coche vamos colocando los objetos de mayor a menor tamaño**



- **Normalmente obtienen una *solución próxima a la óptima***
  - **incluso pueden llegar a obtener la óptima en algunos casos**
- **Pero puede haber *casos especiales (patológicos)* para los que la solución encontrada sea muy mala**
  - **o incluso no encontrar solución**



Algoritmos aproximados

- **Son un tipo especial de heurísticos que:**
  - **siempre encuentran una solución**
  - **el error máximo de la solución obtenida está acotado. Esta cota puede ser:**
    - **valor máximo de la diferencia (en valor absoluto) entre la solución obtenida y la óptima**
    - **valor máximo de la razón entre la solución óptima y la obtenida**
- **Debemos encontrar una demostración matemática de la existencia de la cota**
- **No existe un algoritmo aproximado para todos los problemas para los que existe un heurístico**

5.9 Heurístico para coloreado de grafos

Se trata de colorear los nodos de un grafo de forma que no haya dos nodos adyacentes del mismo color

- **utilizando el *mínimo número de colores* posible**
- **es uno de los problemas NP-completos clásicos (no existe ningún algoritmo de tiempo polinómico que le resuelva de forma óptima)**

Existe un algoritmo heurístico voraz muy sencillo:

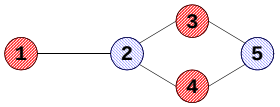
- **se toma un color no usado y un nodo no coloreado aún**
- **se recorren todos los nodos que quedan sin colorear, pintando del color actual todos los que sea posible**
  - **aquellos que no tengan ningún vecino del color actual**
- **se repiten los pasos hasta que se hayan coloreado todos los nodos**

Pseudocódigo

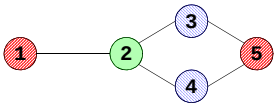
```
// coloreado de grafos utilizando una heurística
// voraz
método coloreaGrafoVoraz(Grafo g)
  Lista nodosNoColoreados=g.listaDeNodos()
  mientras nodosNoColoreados no está vacía hacer
    n=nodosNoColoreados.extraePrimero()
    c=un color no usado aún
    n.colorea(c)
    para cada nodo nnc en nodosNoColoreados hacer
      si nnc no tiene ningún vecino de color c
      entonces
        nodosNoColoreados.extrae(nnc)
        nnc.colorea(c)
      fsi
    fhacer
  fhacer // mientras nodosNoColoreados no está vacía
fmétodo
```


Bondad de la solución obtenida

El algoritmo obtiene diferentes soluciones dependiendo del orden de los nodos en `nodosNoColoreados`:

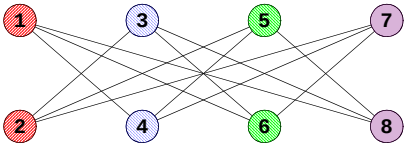


`Orden:1,2,3,4,5(dos colores)`

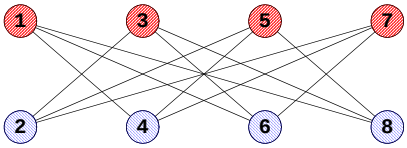


`Orden:1,5,2,3,4(tres colores)`

Para los grafos *bipartitos* la solución obtenida podría estar muy lejos de la óptima:



`Orden:1,2,3,4,5,6,7,8`



`Orden:1,3,5,7,2,4,6,8`

El algoritmo `coloreaGrafoVoraz` *NO* es un algoritmo aproximado

- **no podemos poner cota al error de la solución obtenida ya que crece con el número de nodos del grafo**
  - **para un grafo bipartito de n nodos la solución óptima es 2 colores, pero el algoritmo puede llegar a utilizar `n/2`**

Su eficiencia en el peor caso es $O(n^3)$

- **peor caso: grafo completo (todos los nodos están conectados entre sí)**
- **el lazo externo se realiza n veces ya que a cada pasada sólo se va a lograr colorear un único nodo: $O(n)$**
- **el lazo interno se realiza para todos los nodos no coloreados: $O(n)$**
- **dentro del lazo interno, para cada nodo puede haber que comprobar sus `n-1` vecinos: $O(n)$**

5.10 Heurístico para el problema del viajante

El problema del viajante es otro problema NP-completo clásico

Se trata de encontrar el *itinerario más corto* que permita a un viajante recorrer un conjunto de ciudades

- **pasando una única vez por cada una**
- **y regresando a la ciudad de partida**

Los algoritmos óptimos conocidos tienen un ritmo de crecimiento exponencial

- **no son prácticos a la hora de resolver ejemplares "grandes"**

El problema se suele representar mediante:

- **un grafo completo no dirigido con un nodo por cada ciudad**
- **o utilizando una matriz de distancias (lo normal es que se trate de una matriz simétrica)**

Una heurística voraz sencilla consiste en

- **ir desplazándose desde cada ciudad a la ciudad más próxima no visitada aún**
- **finalmente retornar a la ciudad desde la que se comenzó el viaje**

Ejemplo de matriz de distancias para 6 ciudades:

- **Recorrido óptimo: A→B→C→F→D→E→A=58**
- **Recorrido encontrado por el heurístico: A→B→C→E→D→F→A=60**

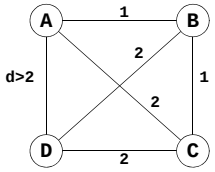
Desde	Hasta					
	A	**B**	**C**	**D**	**E**	**F**
A	-	3	10	11	7	25
B	3	-	8	12	9	26
C	10	8	-	9	4	20
D	11	12	9	-	5	15
E	7	9	4	5	-	18
F	25	26	20	15	18	-

Pseudocódigo

```
// problema del viajante usando heurística voraz
método viajanteVoraz(Grafo g)
  Lista ciudadesPorVisitar=g.listaDeNodos()
  c=ciudadesPorVisitar.extraePrimera()
  Lista itinerario={c} // comienza en c
  mientras ciudadesPorVisitar no está vacía hacer
    cprox=ciudad más próxima a c en ciudadesPorVisitar
    // elimina la más próxima de la lista a visitar
    // y la añade al itinerario
    ciudadesPorVisitar.extrae(cprox)
    itinerario.añadeAlFinal(cprox)
    c=cprox  // cprox pasa a ser la ciudad actual
  fhacer
  // vuelve al comienzo
  itinerario.añadeAlFinal(itinerario.primera())
fmétodo
```


Bondad de la solución obtenida

Se pueden encontrar casos para los que la solución del heurístico sea tan mala como se quiera



Heurístico: A→B→C→D→A=1+1+2+d

Óptimo: A→B→D→C→A=1+2+2+2

- **por consiguiente el algoritmo es un heurístico, pero no un aproximado (ya que no es posible acotar el error máximo)**

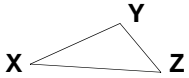
Eficiencia: $O(n^2)$ (donde n es el número de ciudades)

- **el lazo externo se realiza una vez para cada ciudad $O(n)$**
- **para buscar la ciudad más próxima hay que recorrer todas las ciudades que faltan de visitar $O(n)$**

5.11 Aproximado para el problema del viajante

Existe un algoritmo aproximado para el problema del viajante que puede aplicarse cuando:

- **es un grafo completo**
- **la matriz de distancias verifica que para cualquier trío de nodos X, Y y Z se cumple que $distancia(X,Z) \leq distancia(X,Y)+distancia(Y,Z)$**



- **se denomina propiedad *métrica***

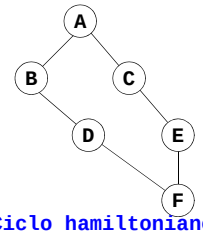
Esta condición NO se verifica en muchos problemas:

- **algunos casos de distancias entre ciudades**
- **tiempos de viaje**
- **precios de viajes**

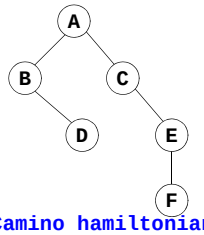
Demostración

El itinerario que buscamos es un ciclo (ciclo hamiltoniano)

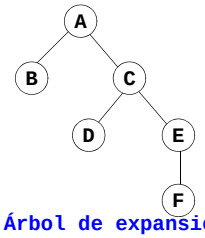
- **en cualquier ciclo si se elimina una de sus aristas se obtiene un camino (de longitud menor que la del ciclo)**
- **todo camino es un árbol de expansión, por consiguiente**
- **tendrá una longitud mayor que el árbol de expansión mínimo**



Ciclo hamiltoniano
longitud c



Camino hamiltoniano
(es un árbol de expansión)
longitud c' (c'<c)

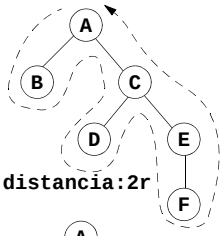


Árbol de expansión mínimo (MST)
longitud r (r≤c')

- **luego r<c: cualquier ciclo es más largo que el MST**

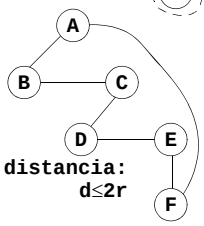
Si recorremos todos los nodos utilizando únicamente las aristas del MST

- **recorrido: A→B→A→C→D→C→E→F→E→C→A**
- **hay que pasar dos veces por cada arista**
- **distancia recorrida 2r**



Si tomamos "atajos":

- **recorrido: A→B→C→D→E→F→A**
- **la propiedad métrica permite asegurar que la distancia recorrida d es menor o igual que 2r**



Luego este itinerario verifica que:

- **$d \leq 2r < 2c$**
- **por consiguiente, será como máximo dos veces más largo que cualquier ciclo (incluido el itinerario óptimo para el viajante)**

Algoritmo

El algoritmo aproximado para el problema del viajante "métrico" consistirá en:

- **encontrar el árbol de expansión mínima del grafo**
  - **utilizando el algoritmo de Prim ($O(n^2)$)**
- **y posteriormente ordenar sus nodos en preorden ($O(n)$)**
- **luego su eficiencia es $O(n^2)$**

El itinerario obtenido por este algoritmo nunca será más de dos veces más largo que el que habríamos obtenido utilizando el algoritmo óptimo

- **$d_{aprox} < 2d_{óptima}$**

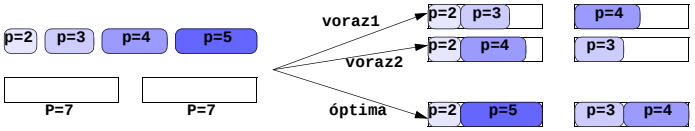
5.12 Aproximado para el llenado de cajas

Este problema es una variación del problema de la mochila:

- **disponemos de `N` objetos numerados del `0` al `N-1` y ordenados de menor a mayor peso ($p_i \leq p_j$ si `i<j`)**
- **y de `C` cajas idénticas cada una capaz de soportar un peso `P`**
- **se trata de meter el *máximo número de objetos* en las cajas**
- **es un problema NP-completo**

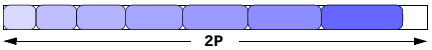
Existe una heurística voraz óptima si sólo hay una caja:

- **ir metiendo los objetos empezando por los de menor peso**
- **esta estrategia *deja de ser óptima cuando hay más de una caja***



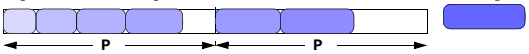
Aunque el heurístico no es óptimo para `C=2`, es posible demostrar que fallará, como máximo, en un objeto

- **suponemos que tenemos una caja que soporta un peso `2P`**
- **y la llenamos empezando por los objetos de menor peso**
  - **la solución obtenida es óptima para una caja de peso `2P` y mejor o igual que la solución óptima para 2 cajas de peso `P`**



Si ahora partimos la caja por la mitad, deberemos desplazar el objeto "partido" a la segunda caja

- **lo que puede provocar que, a lo sumo, el último objeto no quepa**



- **está sería la solución que habríamos obtenido con nuestro algoritmo aproximado**

El resultado anterior se puede generalizar:

- **para un número de cajas `C`, la solución aproximada tendrá, a lo sumo, `C-1` objetos menos que la solución óptima**

La eficiencia del algoritmo es:

- **ordenación de los objetos por pesos no decrecientes: `O(n log n)`**
- **algoritmo voraz que va tomando los objetos uno a uno y guardándoles en las cajas: `O(n)`**
- **eficiencia total: `O(n log n)`**

5.13 Bibliografía

[1] Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.

[2] Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[3] Entrada en la *Wikipedia* para “algoritmo voraz”

[4] Sartaj Sahni, *Data Structures, Algorithms, and Applications in Java*. Mc Graw Hill, 2000