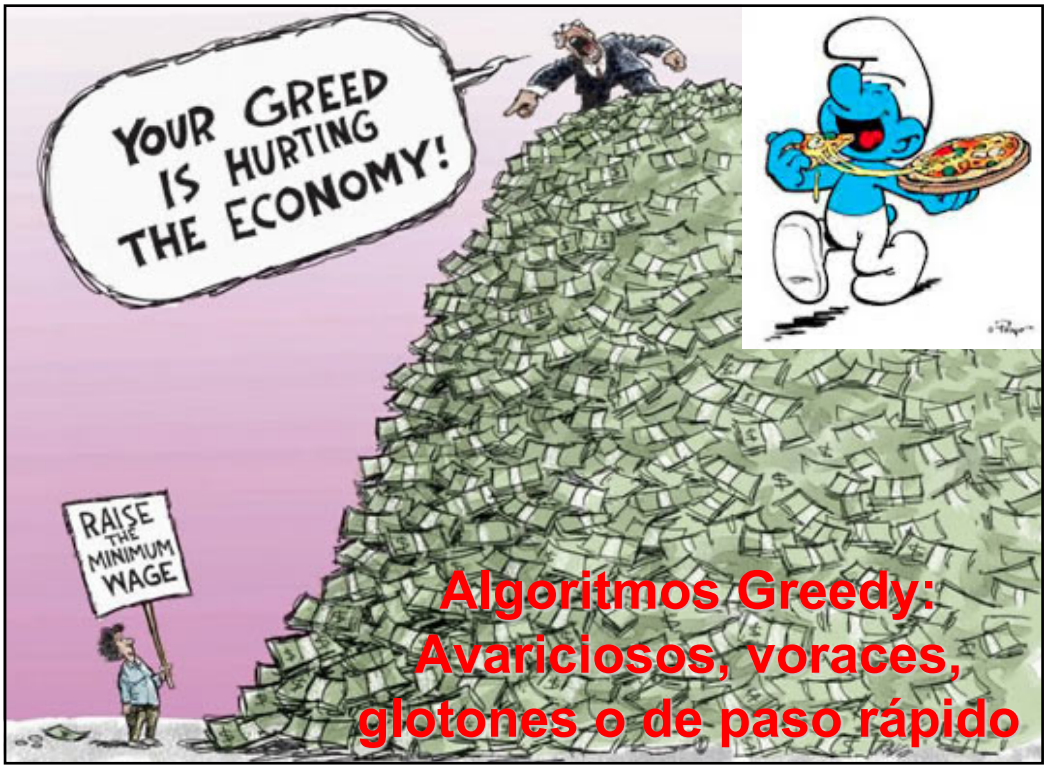
Tema 3 – Algoritmos Voraces (Greedy)

- *3.1 Introducción.*
- *3.2 Esquema general.*
- *3.3 Análisis de tiempos de ejecución.*
- *3.4 Ejemplos.*
  - *3.4.1 Caminos mínimos. Algoritmo de Dijkstra.*
  - *3.4.2 Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.*
  - *3.4.3 El problema de la Mochila 0/1.*
  - *3.4.4 El problema del Viajante.*
  - *3.4.5 Planificación de tareas.*
  - *3.4.6 Otros ejemplos.*



1 – Introducción.

Enfoque:

- Simple.
- Eficiente.
- Muy utilizado

En cada etapa "toman todo lo que pueden" sin analizar consecuencias, es decir, son avariciosos por naturaleza.

Nombres:

- Algoritmos voraces.
- Algoritmos ávidos.
- Algoritmos de avance rápido.
- Algoritmos glotones.
- Greedy algorithms.

1 – Introducción.

Características:

- Típicamente se emplean para resolver problemas de optimización, es decir, minimizar o maximizar, bajo determinadas condiciones, el valor de una función del tipo:

$$f(x1, x2, ..., xn) = c1x1 + c2x2 + ... + cnxn$$

- La solución se va construyendo en etapas.

- Toman decisiones en función de la información que está disponible en cada momento.

- Una vez tomada la decisión, ésta no vuelve a replantearse en el futuro.

- Suelen ser algoritmos rápidos y fáciles de implementar.

1 – Introducción.

Características:

- Existe una entrada de tamaño n que son los candidatos para formar parte de la solución. La naturaleza de los candidatos depende del problema.

- Existe un subconjunto de esos n candidatos que satisface ciertas restricciones: se llama solución parcial, factible o prometedora.

- En cada etapa, se añade un candidato (el mejor posible) a la solución parcial.

- Durante el desarrollo del algoritmo se van formando dos conjuntos: los candidatos seleccionados y los rechazados.

- El orden en que se toman los candidatos, puede ser importante o no serlo.

1 – Introducción.

Características:

- Al final de cada etapa, se verifica si la solución parcial ya es la solución total para el problema.

- Hay que obtener la solución parcial que maximice o minimice una cierta función objetivo: será la solución óptima.

- Si funciona bien, la primera solución es la óptima. No hay marcha atrás.

- No siempre garantizan alcanzar la solución óptima. Por lo tanto, siempre habrá que estudiar la corrección del algoritmo, para demostrar si las soluciones obtenidas son óptimas o no.

1 – Introducción.

Elementos de una solución "greedy":

1. **Conjunto de candidatos** (elementos seleccionables, corresponden a las n entradas del problema)
   (C:conjunto_candidatos)
2. **Solución parcial** (candidatos seleccionados).
   (S:conjunto_solución_etapa)
3. **Función de selección** (determina en cada etapa el mejor candidato del conjunto de candidatos seleccionables, es decir, que aún no han sido seleccionados ni rechazados). (x es un candidato)
   x = selecciona (C)
   a) La función de selección suele estar relacionada con la función objetivo. Ej: Si se quiere maximizar beneficios, se escoge el candidato con mayor valor individual. Si se quiere minimizar costes, se selecciona el candidato más barato.
   b) Pueden definirse varias funciones de selección, se escoge la más adecuada en función de los objetivos

1 – Introducción.

Elementos de una solución "greedy":

4. **Función de factibilidad** (determina si es posible completar la solución parcial, añadiendo candidatos, para alcanzar una solución del problema).
   boolean factible (S U {x})
5. **Función de solución, o criterio que define lo que es una solución** (indica si la solución parcial obtenida resuelve el problema).
   boolean solución (S)
6. **Función objetivo** (valor de la solución).
   tipo objetivo (S) // ***maximizado/minimizado***

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio

✓ Se trata de devolver una cantidad de dinero con el menor número posible de monedas.

✓Se parte de un conjunto de tipos de monedas válidas, de las que se supone que hay cantidad suficiente para realizar el desglose, y de un importe a devolver.

✓Veamos un ejemplo: con las monedas de curso legal del Euro, devolver 3,89 €.

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio

➢ El método intuitivo se puede entender como un algoritmo voraz: en cada paso añadir una moneda nueva a la solución actual, del mayor valor posible, hasta llegar a la cantidad pedida, y sin superar dicha cantidad. Además, la decisión se toma sin pensar en consecuencias posteriores, y una vez tomada, no se modifica.

➢ La respuesta es: (3,89€)

- 1 moneda de 2€
- 1 moneda de 1€
- 1 moneda de 50 cent €
- 1 moneda de 20 cent €
- 1 moneda de 10 cent €
- 1 moneda de 5 cent €
- 2 monedas de 2 cent €

Total: 8 monedas.

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio
Elementos del problema

- **Conjunto de candidatos**: cada una de las monedas de los diferentes tipos que se pueden usar para realizar el desglose del importe a devolver (ej: monedas de los valores de curso legal del Euro). Supondremos una cantidad ilimitada de cada tipo.

$$C = \{2€, 1€, 0{,}50€, 0{,}20€, 0{,}10€, 0{,}05€, 0{,}02€, 0{,}01€\}$$

- **Solución**: conjunto de monedas devuelto tras el desglose y cuyo valor total es igual al importe a desglosar.

$$S = \{x_i / \Sigma\, x_i \cdot v_i = c\ \}$$

Donde: x_i es el número de monedas de la clase i (en el caso del Euro, i=0..7); v_i su valor; y c la cantidad a devolver.

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio
Elementos del problema

- **Condición de factibilidad**: la suma de los valores de las monedas escogidas en un momento dado no superará el importe a desglosar.

$$\Sigma\, x_i \cdot v_i \leq c$$

- **Función de selección**: elegir en cada paso la moneda de mayor valor de entre las candidatas, sin sobrepasar la cantidad que queda por devolver.

$$v_i = \max(\{v_j\}) / \{v_j\} = \{v_k \in C / (v_k < c_i)\}$$

Función objetivo: número total de monedas utilizadas en la solución (debe minimizarse).

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

Verificándose que $\Sigma\, x_i \cdot v_i = c$

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio
Algoritmo inicial (simple)

método **daCambio (cant : real)** retorna **monedas**
  cambio = ∅
  suma = 0
  mientras **suma ≠ cant** hacer
    x = mayor moneda que verifique suma+x ≤ cant
    si **no existe x** entonces
      retorna **no hay solución**
    fsi
    añade x a cambio
    suma = suma + x
  fmientras
  retorna **cambio**
fmétodo

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio
Algoritmo inicial (simple) eficiencia

		Complejidad
Iteraciones	Cuando n es muy grande el número de monedas tiende a n	O(n)
Solución	suma = cant	O(1)
Factibilidad	Existe x	O(1)
Selección	Mayor moneda que verifique suma+x ≤ cant	O(m)

Donde n es la cantidad que se desea cambiar, m es el número de monedas distintas.

* Discutir n *

Complejidad del algoritmo: O(n)·O(m) = O(n) (ya que n>>m)

1 – Introducción.

Ejemplo elemental algoritmo voraz: problema del cambio
Algoritmo mejorado

```
método daCambio2 (real cant; real[8] v) retorna monedas
        int x[8]
        suma = 0; j = 7
        para i = 0, ... ,7 hacer
                x[i] = 0
        fpara
        mientras suma ≠ cant hacer
                mientras (v[j] > (cant - suma)) AND (j>=0) hacer
                        j = j – 1
                fmientras
                si j== -1 entonces
                        retorna no hay solución
                fsi
                x[j] = int((cant - suma)*100) DIV int(v[j]*100)   // ajuste enteros
                suma = suma + v[j] * x[j]
        fmientras
        retorna x
fmétodo
```

En lugar de seleccionar monedas de una en una, usamos la división entera y cogemos todas las monedas posibles de mayor valor.

Los valores de las monedas están ordenados en el array v, de menor a mayor

1 – Introducción.

Aplicabilidad del esquema voraz:

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.
- Incluso pueden no encontrar ninguna solución aunque el problema sí la tenga.

Ejemplo de problema en que no funciona el algoritmo voraz:
Dar cambio con el antiguo sistema monetario inglés (simplificado):
- Libra (20 chelines = 240 p) - Corona (30p) - Florín (24p)
- Chelín (12p)
- 6p - 3p
- Penique (1p)

Solución del algoritmo voraz para 48p: 30p+12p+6p

Solución óptima: 24p+24p (2 florines)

1 – Introducción.

Principio de optimalidad (Bellman, 1957):

Principio de optimalidad: toda sub-secuencia de una secuencia óptima de decisiones, también es óptima.

Ejemplo: cálculo del camino más corto entre ciudades; el camino más corto de Santander a Madrid pasa por Burgos, siendo unión de los caminos más cortos de Santander a Burgos y de Burgos a Madrid.

Una gran parte de los problemas que verifican el principio de optimalidad pueden resolverse utilizando algoritmos voraces, si puesto que en cada etapa escogen, de entre todos los candidatos, el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema (esto último no es seguro). Es posible que exista una función de selección que conduzca a la solución óptima, aunque no hay garantía de que se vaya a encontrar esa función (ni siquiera de que exista).

1 – Introducción.

Principio de optimalidad:

No todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global.

El principio de optimalidad se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas.

Pero…. el principio de optimalidad no nos dice que, si tenemos las soluciones óptimas de los subproblemas, entonces podamos combinarlas para obtener la solución óptima del problema original…

1 – Introducción.

Determinación de la función de selección:

Problema: encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima, sin posibilidad de reconsiderar dicha decisión.

Demostración formal de que la función de selección consigue óptimos globales parar cualquier entrada del algoritmo.

No basta con diseñar un procedimiento ávido, que seguro que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema. Si queremos utilizar un algoritmo voraz para encontrar soluciones óptimas, debe probarse que el algoritmo encuentra la solución óptima en todos los casos.

No son fáciles estas demostraciones

1 – Introducción.

Soluciones aproximadas:

Un algoritmo que no garantice encontrar la solución óptima, puede ser útil en algunos casos para encontrar una solución aproximada, que nos permita resolver el problema objetivo.

Los algoritmos voraces no óptimos a menudo encuentran soluciones bastante aproximadas a las óptimas.

Debido a su eficiencia, este tipo de algoritmos se utiliza aun en los casos donde se sabe que no necesariamente encuentra la solución óptima.

En ocasiones, la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada (p.e. detección misil, control central nuclear).

1 – Introducción.

Soluciones aproximadas: algoritmos heurísticos

- 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.

Heurística: Procedimiento que proporciona una solución aceptable a un problema mediante métodos que carecen de justificación formal (ej: por tanteo, usando reglas empíricas…).

DRAE, Heurística: Arte de inventar.

1 – Introducción.

Soluciones aproximadas, algoritmos heurísticos: tipos

- Heurísticas para problemas concretos. Se desarrollan específicamente para tipos de problemas concretos. (En muchos casos se utilizan las técnicas de "Modelado de conocimiento" propias de la Ingeniería del Conocimiento – CommonKADS -).

-Metaheurísticas (heurísticas de propósito general): Son técnicas genéricas que permiten su uso en diversos campos de aplicación.

- ✓ Enfriamiento simulado.
- ✓ Búsqueda tabú.
- ✓ GRASP [Greedy Randomized Adaptive Search Procedures].
- ✓ Algoritmos bioinspirados:
  - algoritmos genéticos
  - algoritmos meméticos
  - colonias de hormigas
  - …

1 – Introducción.

Ejemplo de esquema de algoritmo memético

No entra en la materia. Es un simple ejemplo!

```
Algoritmo Memético
Entrada: una instancia I de un problema P.
Salida: una solución sol.

// generar población inicial
1 :   para j = 1:popsize hacer
2 :       sea ind = GenerarSoluciónHeuristica (I)
3 :       sea pop[j] = MejoraLocal (ind, I)
4 :   finpara
5 :   repetir // bucle generacional
          // Selección
6 :       sea criadores = SeleccionarDePoblación (pop)
          // Reproducción segmentada
7 :       sea auxpop[0] = pop
8 :       para j = 1:#op hacer
9 :           sea auxpop[j] = AplicarOperador (op[j], auxpop[j ¡ 1], I)
10:       finpara
11:       sea newpop = auxpop[#op]
          // Reemplazo
12:       sea pop = ActualizarProblación (pop, newpop)
          // Comprobar convergencia
13:       si Convergencia (pop) entonces
14:           sea pop = RefrescarPoblación (pop, I)
15:       finsi
16:   hasta CriterioTerminación
17:   devolver Mejor(pop,I)
```


1 – Introducción.

Soluciones aproximadas, algoritmos heurísticos: características

- Suelen estar basados en el conocimiento “intuitivo” o “experto” del diseñador sobre determinado problema, y en reglas de “sentido común”. Ej: Para aprovechar el maletero del coche, guardamos las piezas del equipaje 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 encontrarse solución.

1 – Introducción.

Soluciones aproximadas: algoritmos aproximados

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 uno heurístico.

1 – Introducción.

Soluciones aproximadas: uso del esquema "greedy"

Muchos de los algoritmos heurísticos y aproximados son algoritmos voraces (en su forma):

- Cuando el tiempo que se tarda en resolver un problema es un factor clave, un algoritmo "greedy" puede utilizarse como criterio heurístico.
- También puede utilizarse un algoritmo "greedy" para encontrar una primera solución (como punto de partida para otra heurística).
- La estructura de los algoritmos voraces se puede utilizar para construir procedimientos heurísticos: hablamos de heurísticas voraces. La clave: diseñar buenas funciones de selección.

2 – Esquema general.

```
método voraz (C:conjunto_candidatos) retorna conjunto_solución
  S = ∅                    // irá guardando la solución
  mientras no_es_solución (S) hacer
     si C = ∅ entonces
        retorna no hay solución
     fsi
     x = selecciona (C)
     C = C - {x}          // no se vuelve a considerar
     si factible (S U {x}) entonces
          S = S U {x}
     fsi
   fmientras
   retorna S     // retorna la solución alcanzada
fmétodo
```


2 – Esquema general.

Pasos:

1. Inicialmente el conjunto de candidatos escogidos es vacío: S = Ø.

2.En cada paso, se intenta añadir al conjunto de los escogidos el mejor de los no escogidos aún (sin pensar en el futuro), utilizando una función de selección basada en algún criterio de optimización (puede ser o no la función objetivo).

   VORACIDAD: se consume el mejor elemento lo antes posible sin pensar en futuras consecuencias.

3.Eliminamos el candidato de la lista de candidatos posibles (si hay un único uso posible), o se reduce la disponibilidad (si hay un número finito de usos posibles). Si hay una cantidad ilimitada de unidades disponibles, no se modifica la lista de candidatos.

2 – Esquema general.

Pasos:

4. Con la función de factibilidad (factible) se comprueba si la incorporación del candidato es viable (genera una solución parcial), más adelante se comprueba si el conjunto resultante (solución parcial) es ya una solución total para el problema (con la función de no_es_solución). Si es así, pasaremos a dar la solución final.

5. Si aún no hemos llegado a una solución final, seleccionamos otro candidato y repetimos el proceso, hasta llegar a una solución (o quedarnos sin posibles candidatos).

6. El algoritmo es correcto si la solución encontrada es siempre óptima.

3 – Análisis de tiempos de ejecución.

• Por lo general, los algoritmos voraces son muy eficientes.

• El orden de complejidad depende de diversos factores, como son:

- El número de candidatos (n).
- El número de iteraciones, que viene determinado por el tamaño de la solución (m).
- 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 pre-ordenar el conjunto de candidatos, para que la función de selección sea más rápida.

3 – Análisis de tiempos de ejecución.

El mecanismo general de cálculo de la complejidad podría resumirse:

1.Sea n: número de elementos de C; m: número de elementos de una solución.

2.Repetir, como máximo n veces y como mínimo m:

- Comprobar si el valor actual es solución: f(m). Normalmente O(1) ó O(m).

- Selección de un elemento entre los candidatos: g(n). Entre O(1) y O(n).

- La función factible es parecida a solución, pero con una solución parcial h(m).

- La unión de un nuevo elemento a la solución puede requerir otras operaciones de cálculo, j(n, m).

3 – Análisis de tiempos de ejecución.

Tiempo de ejecución genérico:

$$t(n,m) \in O(n*(f(m)+g(n)+h(m)) + m*j(n, m))$$

No obstante, el análisis depende de cada algoritmo concreto.

En la práctica, los algoritmos voraces suelen ser bastante rápidos, encontrándose dentro de órdenes de complejidad polinomiales.

Valores muy deseables…

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

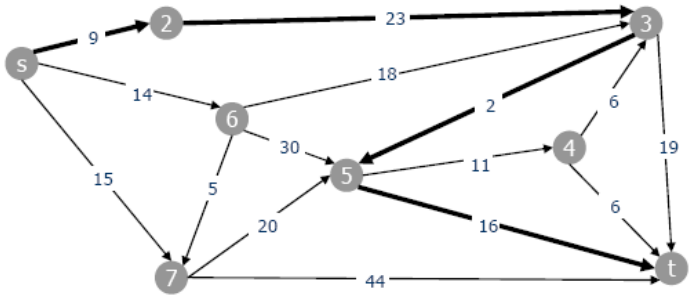
Sea un grafo G = <N, A> donde N es el conjunto de nodos y A el de aristas.

El algoritmo de Dijkstra sirve para resolver el problema de encontrar los caminos de coste mínimo, desde un vértice origen dado a cada uno de los restantes, en grafos dirigidos y ponderados (con pesos no negativos en sus aristas).

Coste (peso, longitud) de un camino: suma de los pesos de las aristas que lo componen.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Utilidad: El grafo puede representar una distribución geográfica, donde las aristas dan el coste (precio, distancia...) de la conexión entre dos lugares, y se desea averiguar el camino más corto (barato...) para llegar a diversos puntos (a todos), partiendo de uno dado.



Camino mínimo del nodo s al nodo t: s – 2 – 3 - 5 – t
Coste = 9 + 23 + 2 + 16 = 50

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (elementos)

- Conjunto de candidatos: Vértices del grafo para los que hay que resolver el problema (inicialmente todos menos el propio origen).

- Solución parcial s: Vértices a los cuales ya sabemos llegar usando el camino mínimo desde el vértice origen (inicialmente Ø).

- Función de selección: hallar el vértice w del conjunto de candidatos (vértices - s) que está a menor distancia del vértice origen.

- Función de factibilidad: el nuevo candidato es siempre factible si su distancia al origen es la mínima.

- Función objetivo: caminos mínimos, es decir, con la menor distancia (coste), del origen a cada nodo.

- Función de solución: no se utiliza.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (mecanismo)

- Genera uno a uno los caminos del nodo origen al resto, por orden creciente de longitud.

- En cada paso, se guardan los nodos para los que ya se sabe el camino mínimo desde el origen. En dicho camino, todos los nodos intermedios pertenecen a s (la solución).

- Cada vez que se incorpora un nodo a la solución s, se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él.

- Se supone que el camino mínimo de un nodo a sí mismo tiene coste nulo.

- Devuelve un vector de distancias del origen a cada uno de los nodos (o vértices), indexado por éstos: en cada posición w se guarda el coste del camino mínimo que conecta el origen con w.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (mecanismo)

- Devuelve un vector de vértices previos: en cada posición w se guarda el vértice previo de w en el camino mínimo del origen a w, de modo que se pueda reconstruir por dónde pasan los caminos mínimos.

- Un valor ∞ en la posición w del vector de distancias indica que no hay ningún camino desde el origen hasta w.

- Al comienzo del algoritmo, las posiciones del vector de distancias se inicializan al valor de la arista del origen al vértice, si existe, y a ∞, si no existe dicha arista. La distancia del origen a sí mismo se inicializa a 0.

- Las posiciones del vector de vértices previos, se inicializan al valor del origen, si existe la arista del origen al nodo, y a nulo si no existe la arista, y para el propio origen.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (esquema)

```
{Pre: g es un grafo dirigido ponderado no negativo.}
método dijkstra (Grafo g, Nodo origen) retorna distancia[ ], previo[ ]
    ConjuntoVértices vertices, s ; s = {origen}
    para cada u ∈ g.nodos hacer
        Inicializar distancia [u]   // Origen 0; peso (origen,u) si existe, ∞ si no existe
        Inicializar previo [u]      // Origen si existe arista (origen,u), nulo otros casos
    fpara
    vertices = g.nodos
    para i desde 1 hasta n-2 hacer   // bucle voraz
            Extraer w ∈ (vertices – s) tal que distancia [w] sea mínima
            s = s U {w}
            para cada v ∈ (vertices – s)
                si ∃ arista(w, v) entonces
                        si distancia[w] + arista(w, v).peso < distancia[v] entonces
                            distancia [v] = distancia [w] + arista(w, v).peso
                            previo [v] = w
                        fsi
                fsi
            fpara
    fpara
    retorna distancia, previo
fmétodo
```

{Post: distancia contiene los costes de los caminos mínimos para (g, origen); previo contiene los vértices previos, que permiten reconstruir los caminos mínimos.}

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (tiempo ejecución)

El bucle principal se ejecuta n-2 veces porque el último camino queda calculado después de éstas (no quedan vértices para encontrar caminos mejores).

Se supone que las operaciones sobre conjuntos están implementadas en tiempo constante, excepto la creación.

a) Fase de inicialización:

- Creación del conjunto y ejecución n veces de diversas operaciones constantes: O(n).

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (tiempo ejecución)

b) Fase de selección:

- Las instrucciones del interior del bucle son O(1).

-Nº de ejecuciones del bucle: 1ª vuelta, se consultan n-1 vértices; 2ª vuelta, n-2; etc... (el cardinal de vértices – s decrece en 1 en cada paso).

Nº de ejecuciones: $n(n-1)/2-1 \in O(n^2)$.

c) Fase de contabilizar como tratado:

- n supresiones a lo largo del algoritmo: O(n).

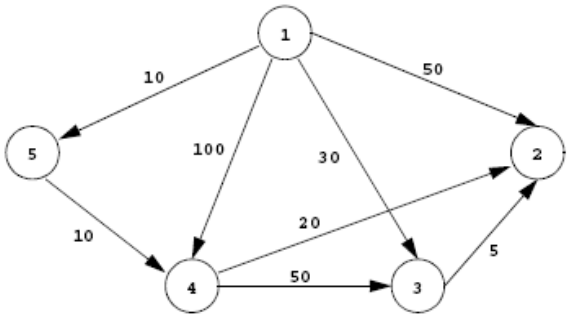
d) Fase de re-cálculo de las distancias mínimas:

- Queda $O(n^2)$ por igual razón que la selección.

Coste total: $O(n^2)$.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra (ejemplo simple)



Iteración	Nodo elegido	d[1]	d[2]	d[3]	d[4]	d[5]	p[1]	p[2]	p[3]	p[4]	p[5]

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado

- Si la representación del grafo es por listas de adyacencia, la fase de re-cálculo se ejecuta sólo m veces, con m < n2 (sustituyendo el bucle sobre los nodos de vertices – s por otro bucle sobre los vértices adyacentes del vértice elegido w).

- Si el conjunto vertices – s se sustituye por una cola de prioridad, con criterio del mínimo para la distancia al origen, se rebaja también el coste de la fase de selección (puesto que se selecciona el vértice con mínima distancia al origen). Problema: la fase de re-cálculo puede exigir cambiar la prioridad de un elemento cualquiera de la cola. Solución: nuevo tipo de cola de prioridad que permita el acceso directo a cualquier elemento.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado

En cada iteración:

1. Se extrae de la cola el nodo w con menor distancia al origen. Se asigna a distancia [w] el valor de dicha distancia.
2. Para cada nodo v adyacente a w:

Si distancia [v] > distancia [w] + valor arista entre w y v entonces:
distancia [v] = distancia [w] + valor arista entre w y v

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado

- Como en el caso anterior, se utiliza un array distancia, con las distancias de los nodos (o vértices) al origen. Se inicializan todas a ∞ menos la del origen que lo hace a 0.

- También se usa un array previo, que almacena el vértice previo de cada nodo, por el camino más corto que va del origen a dicho nodo. Se inicializan todos a nulo.

- En la cola de prioridad, están los nodos no tratados todavía y sus distancias al origen.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado (esquema)

```
{Pre: g es un grafo dirigido ponderado no negativo.}
método dijkstra (Grafo g, Nodo origen) retorna distancia[ ], previo[ ]
    para cada u ∈ g.nodos hacer        // inicialización
        (distancia [u] = ∞) y (previo [u] = nulo)
    fpara
    distancia[origen] = 0
    Colaprioridad colap = g.nodos
    mientras colap ≠ Ø hacer           // bucle voraz
        w = extraerNodoMinDistancia (colap)
        para cada nodo v ∈ adyacentes de w hacer
            si distancia [v] > distancia [w] + arista(w,v).peso entonces
                distancia [v] = distancia [w] + arista(w,v).peso
                previo [v] = w
            fsi
        fpara
    fmientras
    retorna distancia, previo
fmétodo
```

{Post: distancia contiene los costes de los caminos mínimos para (g, origen); previo contiene los vértices previos, que permiten reconstruir los caminos mínimos.}

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado (tiempo ejecución)

Eficiencia temporal:

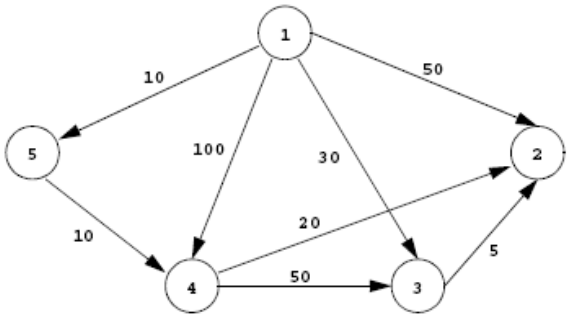
- Inicialización: O(n log n)
- Selección y supresión: O(n log n)
- Bucle interno: examina todas las aristas del grafo y en el caso peor efectúa una sustitución por arista, por tanto: O(m log n)

El coste total es: O((m+n) * log n), luego es mejor que la versión anterior si el grafo es disperso.

Si el grafo es denso, el algoritmo original es mejor.

4 – Ejemplos: Caminos mínimos. Algoritmo de Dijkstra.

Solución voraz: Algoritmo de Dijkstra mejorado (ejemplo simple)



Nodo elegido	Adya-centes	Cola Prioridad (nodo, distancia origen)	d[1]	d[2]	d[3]	d[4]	d[5]	p[1]	p[2]	p[3]	p[4]	p[5]

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo

Objetivo: dado un grafo G = <N, A>, no dirigido, ponderado con pesos positivos y conexo, obtener el subgrafo conexo T $\subseteq$ G, que sólo contenga las aristas imprescindibles para interconectar todos los nodos, con coste mínimo.

Aplicación: problemas relacionados con distribuciones geográficas, como el tendido de líneas de teléfono que permita intercomunicar un conjunto de ciudades, o el diseño de redes (eléctricas, hidráulicas, de ordenadores, de carreteras...), minimizando el precio y hallando enrutamientos sin ciclos.

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo

Árbol libre: grafo no dirigido, sin ciclos y conexo.

Árbol de recubrimiento mínimo (minimum spanning tree, MST): árbol libre que interconecta todos los nodos del grafo, tal que la suma de los pesos de las aristas que lo forman sea lo menor posible.

Propiedad fundamental de los MSTs:

Sea G = <N, A> un grafo no dirigido conexo y ponderado no negativamente.

- Sea U un subconjunto de vértices, $U \subset N$, $U \neq \emptyset$.
- Si (u, v) es la arista más pequeña de A tal que $u \in U$ y $v \in N-U$, entonces existe algún árbol de recubrimiento de coste mínimo de G que contiene a dicha arista.

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces:

Existen al menos dos algoritmos muy conocidos, voraces, que resuelven este problema. En ambos se va construyendo el árbol por etapas, y en cada una se añade una arista (o arco). La forma en la que se realiza esa elección es la que distingue a ambos algoritmos.

✓El algoritmo de Prim comienza por un vértice y escoge en cada etapa la arista de menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya conectados y el otro no.

✓En el algoritmo Kruskal, se ordenan primero las aristas por orden creciente de peso, y en cada etapa se decide qué hacer con cada una de ellas. Si la arista no forma un ciclo con las ya seleccionadas para la solución, se incluye en ella; si no, se descarta.

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: funciones

- Función objetivo a minimizar: longitud (coste) del árbol de recubrimiento.
- Candidatos: las aristas del grafo.
- Función solución: el conjunto de aristas seleccionado es 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 forme un árbol junto con el resto de aristas seleccionadas (Algoritmo de Prim).
  - Seleccionar la arista de menor peso que aún no ha sido seleccionada y que no forme un ciclo (Algoritmo de Kruskal).

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Prim

- Aplica reiteradamente la propiedad de los árboles de recubrimiento de coste mínimo, incorporando en cada paso una arista.
- Se usa un conjunto U de vértices tratados y se selecciona en cada paso la arista mínima que une un vértice de U con otro de su complementario.

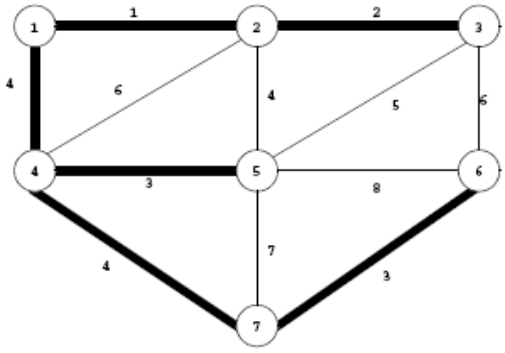
4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Prim
Esquema algorítmico

{Pre: El grafo es no dirigido, conexo y ponderado no negativamente.}
método **prim (Grafo g)** retorna **ConjuntoAristas**
  mst = Ø
  U = conjunto con un nodo cualquiera de g.nodos
  mientras **U ≠ g.nodos** hacer *// bucle voraz*
    a = arista de menor peso aún no considerada tal que
    (a.origen ∈ U) y (a.destino ∈ g.nodos – U)
    mst.añade(a)
    U.añade(a.destino)
  fmientras
  retorna **mst**
fmétodo
{Post: mst es árbol de recubrimiento mínimo de g}

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Prim
Ejemplo.



Etapa	Arista considerada	Nodos conectados por el árbol de recubrimiento mínimo

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Kruskal

✓ Inicialmente disponemos de todos los vértices del grafo y ninguna arista seleccionada, por lo cual cada uno de los vértices está asignado a una componente conexa distinta (el propio vértice aislado).

✓ Conforme se van añadiendo aristas en cada paso del algoritmo, el número de componentes conexas va disminuyendo, y los vértices van siendo asignados a éstas.

✓ Al igual que el de Prim, se basa también en la propiedad de los árboles de recubrimiento de coste mínimo: partiendo del árbol vacío, se selecciona en cada paso la arista de menor peso que aún no haya sido seleccionada y que no conecte dos nodos de la misma componente conexa (es decir, que no forme un ciclo).

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Kruskal

✓ En cada momento, los vértices que están dentro de una componente conexa en la solución forman una clase de equivalencia, y el algoritmo se puede considerar como la fusión continuada de clases hasta obtener una única componente con todos los vértices del grafo.

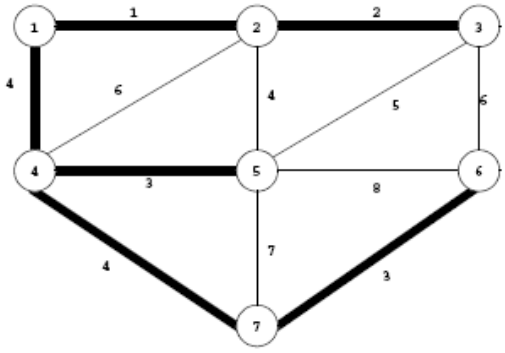
4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Kruskal
Esquema algorítmico

```
{Pre: El grafo es no dirigido, conexo y ponderado no negativamente.}
método kruskal (Grafo g) retorna ConjuntoAristas
  Ordenar las aristas por pesos crecientes
  Crear un conjunto distinto para cada nodo de g.nodos
  mst = Ø
  hacer                          // bucle voraz
        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
            Unir conj1 con conj2
            mst.añade(a)
        fsi
  mientras mst tiene menos de (g.numNodos – 1) aristas
  retorna mst
fmétodo
{Post: mst es árbol de recubrimiento mínimo de g}
```


4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

Árboles de recubrimiento mínimo, soluciones voraces: Kruskal
Ejemplo.



Etapa	Arista considerada	Componentes conexas

4 – Ejemplos: Árboles de recubrimiento mínimo. Algoritmos de Prim y Kruskal.

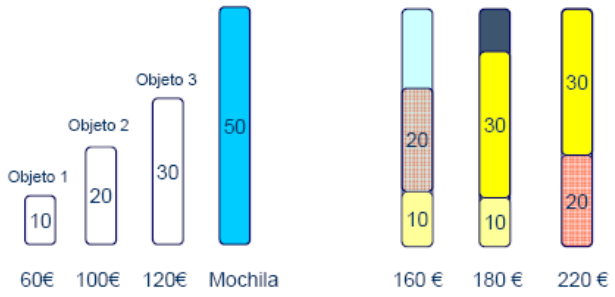
Complejidad de los algoritmos de Prim y Kruskal:

1. Para un grafo con n nodos y a aristas:
   - Prim: $O(n^2)$
   - Kruskal: $O(a \log n)$
2. En un grafo denso se cumple que: $a \rightarrow n(n-1)/2$
   - Prim puede ser mejor (depende del valor de las constantes ocultas).
   - Kruskal $\rightarrow O(n^2 \log n)$
3. En un grafo disperso se cumple que: $a \rightarrow n$
   - Prim es menos eficiente.
   - Kruskal $\rightarrow O(n \log n)$

4 – Ejemplos: El problema de la Mochila 0/1.

Planteamiento del problema: carga de la mochila.

Disponemos de una mochila que soporta hasta un peso dado. Existe una serie de objetos que podemos cargar en la mochila, cada uno con un peso y un valor asociados. El objetivo es seleccionar los objetos, para aprovechar lo mejor posible la mochila, maximizando el beneficio debido a los objetos que transporta.



Aplicación:

Empresas de transporte

4 – Ejemplos: El problema de la Mochila 0/1.

Planteamiento del problema: Carga de la mochila, opciones

1. Los objetos no se pueden fraccionar (mochila 0/1).

   - Problema muy complejo. Clase NP de computabilidad.

   - Solución voraz heurística. "No siempre solución óptima".

2. Los objetos se pueden fraccionar, transportando sólo parte (mochila 0-1).

   - Solución algoritmo voraz. Cumple principio de optimalidad.

   - Dificultades en el mundo real. Situaciones intermedias.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0/1: Planteamiento formal

$$\max \sum_{1\le i\le n} x_i b_i \quad sujeto\ a \sum_{1\le i\le n} x_i p_i \le P \quad con\ x_i \in \{0,1\}$$

donde:

a)n es el número de objetos;
b)b_i ($b_i > 0$) es el beneficio (o valor) asociado al objeto i;
c)p_i ($p_i > 0$) es el peso del objeto i;
d)P es el peso máximo soportado por la mochila;
e)x_i es la presencia/ausencia del objeto i, su valor puede ser 0 o 1.

Extensiones: problema mochila acotado y no acotado

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0/1: heurística "greedy"

Ordenar los objetos por densidad (razón beneficio/peso) decreciente.

$$b_i / p_i \geq b_{i+1} / p_{i+1} \text{ para } 1 \leq i < n$$

Si M es el valor máximo de los objetos que se pueden llevar en la mochila, la heurística garantiza obtener, al menos, un valor de M/2 (George Dantzig).

Sin embargo, la heurística puede proporcionar resultados mucho peores (muy alejados del óptimo) para instancias particulares del problema.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0/1: esquema algorítmico voraz

```
{Prec. Pesos y valores de los objetos reales positivos, Pe peso mochila}
método mochila0/1 (real [1..n] p, real [1..n] v, real Pe) retorna boolean [1..n]
  boolean[] x
  peso = 0
  para todo i inicializar x[i] a false
  mientras (peso < Pe y quedan objetos) hacer
    i = seleccionaMejorObjeto
    si (peso + p[i] ≤ P) entonces
          x[i] = true
          peso = peso + p[i]
    fsi
    eliminar objeto
  fmientras
  retorna x
 fmétodo
{Post. Devuelve a true los elementos a cargar en la mochila}
```

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0/1: eficiencia esquema algorítmico voraz

a) Implementación directa:

• El bucle requiere como máximo n (número de objetos) iteraciones, una para cada posible objeto en el caso de que todos quepan en la mochila: O(n).

• La función de selección requiere buscar el objeto con mejor relación valor/peso: O(n).

Tiempo del algoritmo: $O(n) \cdot O(n) = O(n^2)$.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0/1: eficiencia esquema algorítmico voraz

b) 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).

Tiempo del algoritmo: $O(\max (O(n \log n), O(n) \cdot O(1))) = O(n \log n)$.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0-1: Planteamiento formal

✓ La mochila puede soportar como máximo un peso Pe.

✓ El conjunto de candidatos son n objetos fraccionables, de los cuales podemos guardar una fracción x_i ($0 \leq x_i \leq 1$). Salida.

✓ En cada paso, se añade a la mochila todo lo que se pueda: si el objeto i, elegido como el mejor, no cabe entero, se rellena el espacio disponible con la fracción de dicho objeto que quepa, hasta completar la capacidad de la mochila.

✓ Cada objeto i tiene un peso p_i ($p_i > 0$) y proporciona un beneficio b_i ($b_i > 0$).

✓Objetivo: maximizar el beneficio de los objetos transportados.

$$\max \sum_{1 \leq i \leq n} x_i b_i \quad sujeto\ a \sum_{1 \leq i \leq n} x_i p_i \leq P$$

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0-1: Planteamiento formal

✓ Datos entrada: n tupla (p_1, …, p_n) de pesos, n tupla (b_1, …, b_n) de beneficios. Pe, peso que admite la mochila.

✓Solución factible: cualquier n-tupla (x_1, …, x_n), con ($0 \leq x_i \leq 1$), donde x_i es la fracción escogida del objeto i, que cumpla la restricción a la que está sujeto el objetivo, es decir, que la sumatoria de $x_i \cdot p_i$ sea $\leq P$.

✓ Función de factibilidad: será siempre cierta, ya que se pueden añadir fracciones de objetos, y hacer de ese modo que se cumpla la restricción sobre el objetivo.

✓Solución óptima: solución factible para la que se cumpla el objetivo, es decir, que la sumatoria de $x_i \cdot b_i$ sea máxima.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0-1: Planteamiento formal

✓Criterio de solución: Toda solución óptima ocupará por completo la mochila. Si $p_1 + ... + p_n \leq Pe$, entonces obviamente $x_i = 1$, $1 \leq i \leq n$, es una solución óptima. En este caso, estaríamos guardando todos los objetos completos, con la posibilidad de no haber alcanzado el peso máximo de la mochila.

✓El problema cumple el principio de *optimalidad*. La clave será encontrar la función de selección adecuada.

1. Seleccionar primero los objetos más ligeros, para acumular beneficios de un número mayor de objetos, y agotar la capacidad más lentamente.
2. Seleccionar primero los objetos más valiosos, para incrementar el valor de la carga rápidamente.
3. Seleccionar primero los objetos con mejor relación valor/peso.

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0-1: Planteamiento formal

1. Seleccionar primero los objetos más ligeros, para acumular beneficios de un número mayor de objetos, y agotar la capacidad más lentamente.
2. Seleccionar primero los objetos más valiosos, para incrementar el valor de la carga rápidamente.
3. Seleccionar primero los objetos con mejor relación valor/peso.

Ejemplo de prueba: mochila 100 kg.

Objeto	1	2	3	4	5
Beneficio (€)	20	30	65	40	60
Peso (kg.)	10	20	30	40	50

4 – Ejemplos: El problema de la Mochila 0/1.

Mochila 0-1: Planteamiento formal, función de selección óptima

- Definimos la densidad del objeto i como b_i / p_i y seleccionamos los objetos en orden decreciente de densidad:

$$b_i / p_i \geq b_{i+1} / p_{i+1} \text{ para } 1 \leq i < n$$

- Consideramos los objetos ordenados de mayor a menor relación valor/peso.
- La forma más eficiente de incluir el valor b_i , con mayor relación b_i / p_i, es incluyendo completamente el objeto i, cualquier otra combinación requerirá más peso.
- El mismo razonamiento se puede aplicar para el objeto j, con la segunda relación b_j / p_j y una mochila de peso Pe-p_i, y así sucesivamente para los demás.

4 – Ejemplos: El problema del Viajante.

Planteamiento del problema:

Dado un grafo G = <N, A>, totalmente conexo (completo), no dirigido y ponderado, encontrar un camino que empiece en un vértice v ∈ N y acabe en ese mismo vértice, pasando una única vez por todos los vértices (nodos) de N (es decir, un circuito hamiltoniano), con el objetivo de que sea el circuito hamiltoniano de coste mínimo.

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. Se suponen conocidas las distancias entre cada par de ciudades.

El problema del viajante es otro problema NP-completo (clase de problemas complejos, definida en Teoría de la Computación) clásico.

4 – Ejemplos: El problema del Viajante.

Planteamiento del problema: solución inicial

El algoritmo de *fuerza bruta* para resolver el problema consiste en intentar todas las posibilidades, es decir, calcular las longitudes de todos los recorridos posibles, y seleccionar el de longitud mínima.

(Existe una solución algo mejor mediante la técnica de programación dinámica, se verá más adelante.)

Obviamente, el coste de tal algoritmo crece exponencialmente con el número de ciudades a visitar.

Los algoritmos óptimos conocidos también tienen un ritmo de crecimiento exponencial, luego no son prácticos a la hora de resolver casos "grandes".

4 – Ejemplos: El problema del Viajante.

Planteamiento del problema: datos y aplicación del esquema voraz

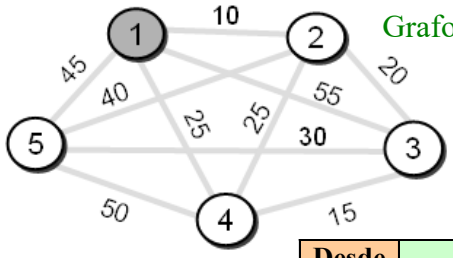
El problema se suele representar mediante dos posibles estructuras de datos:

- Un grafo completo, no dirigido y ponderado, con un nodo por cada ciudad.
- Una matriz de distancias (lo normal es que se trate de una matriz simétrica).

Se trata de un problema de optimización, la solución está formada por un conjunto de elementos en cierto orden: podemos aplicar el esquema voraz.

4 – Ejemplos: El problema del Viajante.

Planteamiento del problema: Estructuras de datos



Grafo completo, no dirigido y ponderado

Matriz de distancias

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	-

4 – Ejemplos: El problema del Viajante.

Heurísticas voraces:

a) Heurística Greedy 1: Nodos como candidatos. Simple.

- Se va desplazando de cada ciudad a la más próxima no visitada aún.
- Cuando se han visitado todas las ciudades se retorna a la de origen.

b) Heurística Greedy 2: Aristas como candidatos.

- Se analizan todas las parejas de puntos posibles.
- Se incorporan al camino las parejas con menos distancia, de forma coherente, es decir, no generando ciclos ni incorporando un punto más de dos veces (generando por tanto más de dos visitas) o cerrando el circuito antes de terminar.
- Se revisan todas las aristas hasta completar el circuito.

4 – Ejemplos: El problema del Viajante.

Heurística voraz a), selección de nodos:

Escoger, en cada momento, el vértice más cercano al último nodo añadido al circuito, siempre que no se haya seleccionado previamente y que no cierre el camino antes de pasar por todos los vértices.

Solución: será un cierto orden en el conjunto de nodos, **s** = (v_1, v_2, ..., v_n), donde v_i es el nodo visitado en el lugar i-ésimo.

Inicialización: empezar en un nodo cualquiera.

Función de selección: de los nodos candidatos, seleccionar el más próximo al último de la secuencia actual (v_1, v_2, ..., v_{actual}).

Objetivo: minimizar el coste del circuito que une los n nodos.

Acabamos cuando tengamos n nodos.

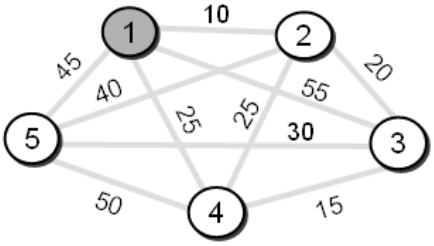
4 – Ejemplos: El problema del Viajante.

Heurística voraz a), esquema algorítmico:

```
{Prec. g grafo completo, no dirigido y ponderado}
método viajanteVoraz(Grafo g) retorna Lista
    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
    fmientras
    // vuelve a la ciudad de partida
    itinerario.añadeAlFinal(itinerario.primera)
    retorna itinerario
fmétodo
{Postc. Lista incluye todos los nodos del grafo, ordenados y sin repetición}
```

4 – Ejemplos: El problema del Viajante.

Heurística voraz a), solución:



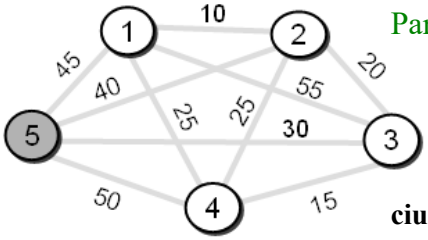
Solución hallada por la heurística greedy 1, partiendo del nodo 1:
(1, 2, 3, 4, 5, 1). Coste total: 10+20+15+50+45=140

Podemos encontrar una solución mejor: (1, 2, 4, 3, 5, 1)
Coste total: 10+25+15+30+45=125

Vemos que la heurística no asegura encontrar la solución óptima.

4 – Ejemplos: El problema del Viajante.

Heurística voraz a), mejora:



Partir de otro origen.

Cambiar el método de obtención del primer nodo:

ciudadesPorVisitar.seleccionarPrimera

??

Solución dada por la heurística 1, partiendo del nodo 5:
(5, 3, 4, 1, 2, 5) Coste total: 30+15+25+10+40 = 120

Vemos que mejora el resultado de la heurística 1, con respecto al obtenido partiendo del nodo 1.

4 – Ejemplos: El problema del Viajante.

Heurística voraz a), eficiencia:

El tiempo de ejecución es $O(n^2)$, donde n es el número de ciudades:

✓El bucle externo se realiza una vez para cada ciudad, luego es O(n).

✓Para buscar la ciudad más próxima hay que recorrer todas las ciudades que faltan por visitar, por tanto O(n).

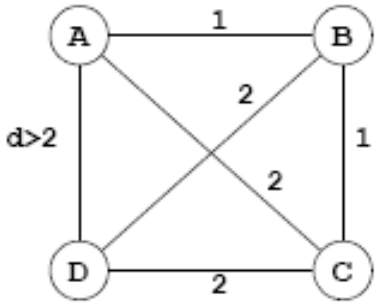
¿Cuál es el efecto de la mejora en la selección del nodo?

4 – Ejemplos: El problema del Viajante.

Heurística voraz a), bondad solución:

Se pueden encontrar casos para los que la solución de la heurística voraz sea tan mala como se quiera.

Se trata de una heurística, no de un algoritmo aproximado, ya que no es posible acotar el error máximo.



Heurística a) base:
$A \rightarrow B \rightarrow C \rightarrow D \rightarrow A = 1+1+2+d$

Óptimo:
$A \rightarrow B \rightarrow D \rightarrow C \rightarrow A = 1+2+2+2$

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Idea general

Seleccionar parejas de puntos que serán visitados de forma consecutiva.

- Se seleccionará primero aquella pareja de puntos entre los que la distancia sea mínima.
- A continuación, se selecciona la siguiente pareja separada por una distancia mínima, siempre que esta nueva elección no haga que:
  - Se visite un punto dos veces o más (es decir, que el punto aparezca tres o más veces en las parejas de puntos seleccionadas.)
  - Se cierre un recorrido antes de haber visitado todos los puntos.

De esta forma, si hay que visitar *n* puntos (incluido el origen), se selecciona un conjunto de *n* parejas de puntos (que serán visitados de forma consecutiva) y la solución consiste en reordenar todas esas parejas de forma que constituyan un recorrido.

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Mecanismo

Se ordenan las aristas y se procede como en el algoritmo de Kruskal, pero garantizando que al final se forme un circuito. En este caso, la eficiencia depende del algoritmo de ordenación que se use.

En cada etapa, se elige la arista más corta de las restantes, teniendo en cuenta que:

1. No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completará el recorrido del viajante.)
2. La elegida no sea la tercera arista incidente a un nodo cualquiera (formando así una estrella.)

Solución: conjunto de aristas $s = (a_1, a_2, ..., a_n)$ que formen un ciclo hamiltoniano, sin importar el orden, donde cada $a_i = (v_i, w_i) \in A$.

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Mecanismo

Solución: conjunto de aristas $s = (a_1, a_2, ..., a_n)$ que formen un ciclo hamiltoniano, sin importar el orden, donde cada $a_i = (v_i, w_i) \in A$.

Inicialización: empezar con un grafo sin aristas.

Selección: seleccionar la arista candidata de menor coste.

Función factible: una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que dos (que sean incididos por más de dos aristas).

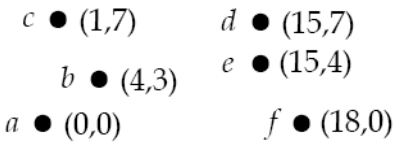
Objetivo: minimizar el coste del circuito formado por las n aristas.

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Ejemplo

Desde	Hasta					
	a	b	c	d	e	f
a	-	5	7,07	16,55	15,52	18
b	5	-	5	11,70	11,05	14,32
c	7,07	5	-	14	14,32	18,38
d	16,55	11,70	14	-	3	7,6
e	15,52	11,05	14,32	3	-	5
f	18	14,32	18,38	7,6	5	-

Puntos en el plano



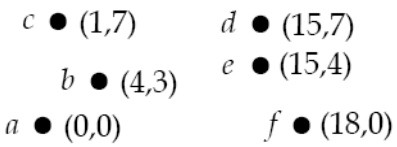
1. Las parejas ordenadas por distancia entre sus puntos son: (*d*,*e*), (*a*,*b*), (*b*,*c*), (*e*,*f*), (*a*,*c*), (*d*,*f*), (*b*,*e*), (*b*,*d*), (*c*,*d*), (*b*,*f*), (*c*,*e*), (*a*,*e*), (*a*,*d*), (*a*,*f*) y (*c*,*f*).
2. Se selecciona primero la pareja (*d*,*e*) pues la distancia entre ambos puntos es mínima (3 unidades). Lista de parejas: (*d*,*e*)
3. Después se seleccionan las parejas (*a*,*b*), (*b*,*c*) y (*e*,*f*). La distancia es igual para todas ellas (5 unidades). Lista de parejas: (*d*,*e*), (*a*,*b*), (*b*,*c*), (*e*,*f*)
4. La siguiente pareja de distancia mínima es (*a*,*c*), con longitud 7,07. Sin embargo, esta pareja cierra un recorrido junto con otras dos ya seleccionadas, (*b*,*c*) y (*a*,*b*), por lo que se descarta.
5. La siguiente pareja es (*d*,*f*) y se descarta también por motivos similares.

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Ejemplo

Desde	Hasta					
	a	b	c	d	e	f
a	-	5	7,07	16,55	15,52	18
b	5	-	5	11,70	11,05	14,32
c	7,07	5	-	14	14,32	18,38
d	16,55	11,70	14	-	3	7,6
e	15,52	11,05	14,32	3	-	5
f	18	14,32	18,38	7,6	5	-

Puntos en el plano



(*d*,*e*), (*a*,*b*), (*b*,*c*), (*e*,*f*) -- (*a*,*c*),(*d*,*f*), (*b*,*e*), (*b*,*d*), (*c*,*d*), (*b*,*f*), (*c*,*e*), (*a*,*e*), (*a*,*d*), (*a*,*f*), (*c*,*f*)

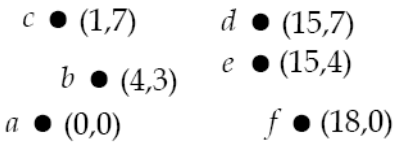
6. La siguiente pareja de distancia mínima es (*a*,*c*), con longitud 7,07. Sin embargo, esta pareja cierra un recorrido junto con otras dos ya seleccionadas, (*b*,*c*) y (*a*,*b*), por lo que se descarta.
7. La siguiente pareja es (*d*,*f*) y se descarta también por motivos similares.
8. La siguiente es la pareja (*b*,*e*), pero debe rechazarse también porque su inclusión haría visitar más de una vez los puntos *b* y *e*.
9. La pareja (*b*,*d*) se rechaza por motivos similares (el punto *b* habría que visitarlo dos veces).
10. La siguiente pareja es (*c*,*d*), y se selecciona. Lista de parejas: (*d*,*e*), (*a*,*b*), (*b*,*c*), (*e*,*f*), (*c*,*d*)

4 – Ejemplos: El problema del Viajante.

Heurística voraz b), selección de aristas: Ejemplo

Desde	Hasta					
	a	b	c	d	e	f
a	-	5	7,07	16,55	15,52	18
b	5	-	5	11,70	11,05	14,32
c	7,07	5	-	14	14,32	18,38
d	16,55	11,70	14	-	3	7,6
e	15,52	11,05	14,32	3	-	5
f	18	14,32	18,38	7,6	5	-

Puntos en el plano



(*d*,*e*), (*a*,*b*), (*b*,*c*), (*e*,*f*) (*c*,*d*) -- (*b*,*f*), (*c*,*e*), (*a*,*e*), (*a*,*d*), (*a*,*f*), (*c*,*f*)

11. Las parejas (*b*,*f*), (*c*,*e*), (*a*,*e*) y (*a*,*d*) no son aceptables.
12. Finalmente, la pareja (*a*,*f*) cierra el recorrido:

Este recorrido no es el óptimo pues su longitud es de 50 unidades. No obstante, es el cuarto mejor recorrido de entre los sesenta (esencialmente distintos) posibles y es más costoso que el óptimo en sólo un 3,3%.

4 – Ejemplos: Planificación de tareas.

Conjunto de problemas:

a) Selección de tareas.

Dado un conjunto de tareas posibles (no compatibles en el mismo tiempo), realizar tantas como sea posible/ ocupar máximo tiempo.

b) Minimización del tiempo de espera.

Atender un conjunto de tareas con tiempos de ejecución distintos minimizando el tiempo conjunto de espera.

c) Planificación con fechas límite.

Seleccionar un conjunto de tareas factibles con fecha límite de ejecución maximizando el beneficio.

4 – Ejemplos: Planificación de tareas.

Selección de tareas:

Se tiene un conjunto C = {1, 2, …, n} de actividades que deben usar un recurso que sólo puede ser usado por una actividad en cada instante.

Cada actividad i tiene asociado un instante de comienzo c_i y un instante de finalización f_i, tales que $c_i \leq f_i$, de forma que la actividad i, si se realiza, debe transcurrir durante el intervalo $[c_i, f_i)$.

Ejemplos:

a) Hay un conjunto de clases (actividades) que se han de dar en un mismo aula (recurso). Hay que asignar las clases en el tiempo de forma que se utilice al máximo el aula.
b) Tenemos que elegir de entre un conjunto de actividades (conferencias). Dan puntos y queremos asistir a todas las posibles. Sin embargo, no se pueden solapar en el tiempo, ya que no podemos estar en dos sitios a la vez.

4 – Ejemplos: Planificación de tareas.

Selección de tareas:

Dos actividades i, j se dicen compatibles si los intervalos $[c_i, f_i)$ y $[c_j, f_j)$ no se superponen (por ej., $(c_i \geq f_j)$ o $(c_j \geq f_i)$).

El problema de selección de actividades consiste en seleccionar un subconjunto S de C, con actividades mutuamente compatibles, buscando dos posibles objetivos:

a) Que S tenga cardinal máximo. Ej. Se imparte el mayor número de clases posible.

b) Que se minimice el tiempo en que no se realiza ninguna actividad. Ej. El aula está el mínimo tiempo desocupada.

4 – Ejemplos: Planificación de tareas.

Selección de tareas: elementos del esquema voraz

Conjunto de candidatos: C = {actividades propuestas}.

Solución parcial: S = {actividades seleccionadas por el momento}. (inicialmente, S=Ø).

Función de selección: Distintas posibilidades: terminación más temprana, menor duración…

Función de factibilidad: $x \in C$ es factible si es compatible (esto es, no se solapa) con las actividades de S.

Criterio que define lo que es una solución: haber considerado todas las actividades del conjunto C.

Función objetivo: a) Cardinal de S, que debe ser máximo.
b) $\sum \forall\, i \in \{1 .. N\}. (f_j - c_i)$ máximo.

4 – Ejemplos: Planificación de tareas.

Selección de tareas: elementos del esquema voraz

Estrategias greedy (alternativas con respecto al orden en el que se pueden considerar las actividades), función de selección:

✓Orden creciente de tiempo de comienzo.

✓Orden creciente de tiempo de finalización.

✓Orden creciente de duración: $f_i - c_i$.

✓Orden creciente de conflictos con otras actividades.

Discutir el efecto de cada una…

4 – Ejemplos: Planificación de tareas.

Selección de tareas: esquema voraz (orden creciente finalización)

{ Pre: $\forall i \in 1 .. n : c[i] \leq f[i] \wedge \forall i \in 1 .. n-1: f[i] \leq f[i+1]$ }

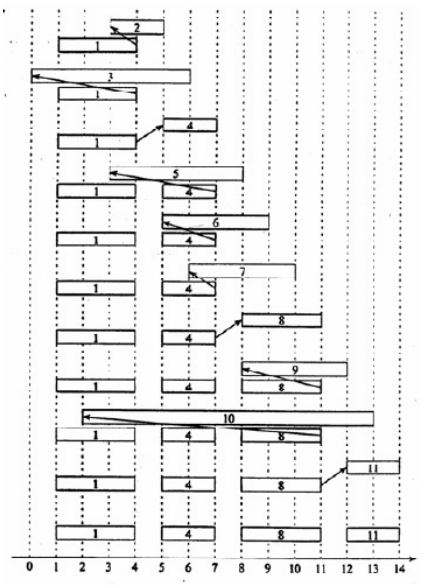
```
método seleccionaActividades (c, f: array [1..n] de real) retorna conjunto
  conjunto s
  s = { 1 }          // Selección voraz: actividad primer instante de finalización.
  j = 1              // j es la última actividad seleccionada.
  para i = 2 hasta n hacer
     si c [ i ] ≥ f [ j ] entonces     // Si comienza después actividad previa
          s = s ∪ { i }                // Seleccionar
          j = i
     fsi
  fpara
  retorna s
fmétodo
```

{ Post: s es solución óptima del problema de la selección de actividades. }

4 – Ejemplos: Planificación de tareas.

Selección de tareas: ejemplo

i	c_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



4 – Ejemplos: Planificación de tareas.

Selección de tareas: eficiencia y corrección de la solución

✓ El coste del bucle es O(n), pero dado que hay que ordenar previamente los vectores, queda **O(n log n)**.

✓Corrección de la solución:

- Después de cada selección voraz, nos encontramos con otro problema de optimización con igual forma que el problema original.
- Por inducción sobre el número de selecciones, y realizando una selección voraz en cada paso, se obtiene una solución óptima.

4 – Ejemplos: Planificación de tareas.

Minimización del tiempo de espera:

Supongamos un servidor (por ej., un procesador, un cajero automático, un surtidor de gasolina, etc.) que tiene que atender n clientes, que llegan todos juntos al sistema.

El tiempo de servicio para cada cliente i es t_i, con i=1,2, …, n.

Teniendo en cuenta que en un instante dado sólo se puede estar atendiendo a un cliente, se quiere minimizar el tiempo total que los n clientes están en el sistema:

$$T = \sum \forall\, i \in \{1 \,..\, N\}.\, T_i$$

$$T_i = \sum \forall\, j \in \{1 \,..\, i\}.\, t_j$$

4 – Ejemplos: Planificación de tareas.

Minimización del tiempo de espera: Solución voraz

Atender en cada paso al cliente no atendido con menor tiempo de servicio.

Ejemplo: Sean tres clientes, con t_1=5, t_2=10, t_3=3.

1, 2, 3:	5 + (5+10) + (5+10+3) = 38	
1, 3, 2:	5 + (5+3) + (5+3+10) = 31	
2, 1, 3:	10 + (10+5) + (10+5+3) = 43	
2, 3, 1:	10 + (10+3) + (10+3+5) = 41	
3, 1, 2:	3 + (3+5) + (3+5+10) = 29	→ Secuencia óptima
3, 2, 1:	3 + (3+10) + (3+10+5) = 34	

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite:

- Tenemos un procesador y n posibles tareas para ejecutar.
- Cada tarea i ($1 \leq i \leq n$) tiene una fecha límite t_i, es decir, la tarea i tiene que estar finalizada antes, o en el propio instante de tiempo t_i, o sea, en un instante $t \leq t_i$.
- Todas las tareas consumen la misma cantidad de tiempo: una unidad (time slot).
- En cada instante (time slot), sólo se puede ejecutar una tarea, bajo la suposición de que en ese instante se realiza la tarea completa.
- Suponemos también que la tarea i produce una ganancia g_i ($g_i > 0$) si y sólo si se ejecuta sin sobrepasar su fecha límite, es decir, sin sobrepasar el instante de tiempo t_i, en un instante $t \leq t_i$.
- Puede que no sea posible ejecutar todas las tareas.

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite:

- Objetivo: establecer una secuencia posible de tareas, respetando las fechas límite, que permita maximizar la ganancia obtenida (suma de las ganancias asociadas a las tareas de la secuencia).

Ejemplo: Sean las cuatro tareas de la tabla dada, con las ganancias y tiempos límites señalados.

i	**1**	**2**	**3**	**4**
g_i	50	10	15	30
t_i	2	1	2	1

Analizamos todas las secuencias (para dos slots de tiempo):
1 / 2 / 3 / 4 / 1, 2 / 1, 3 / 1, 4 / 2, 1 / 2, 3 / 2, 4 /
3, 1 / 3, 2 / 3, 4 / 4, 1 (óptima) / 4, 2 / 4,3

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: solución

Una solución estará formada por un conjunto de candidatos, junto con un orden de ejecución de los mismos.

Un conjunto de tareas es factible si existe al menos una secuencia (también factible) que permite que todas las tareas en el conjunto se ejecuten en el tiempo de sus respectivas fechas límite.

Planificación con fechas límite: fuerza bruta

Comprobar todos los posibles órdenes de las tareas y quedarse con el mejor (y que sea factible).

Complejidad de **Ω(n!)** ...

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite:

Algoritmo voraz: construir la secuencia etapa por etapa (partiendo de la secuencia vacía), añadiendo en cada etapa la tarea con el mayor valor de g_i entre aquéllas que aún no hayan sido consideradas (ésa es la función de selección).

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz

Construir la secuencia etapa por etapa (partiendo de la secuencia vacía), añadiendo en cada etapa la tarea con el mayor valor de g_i entre aquéllas que aún no hayan sido consideradas (ésa es la función de selección).

Para el esquema algorítmico de la implementación, suponemos que las tareas están enumeradas de modo que se verifique la siguiente precondición:

{ Pre: $g_1 \geq g_2 \geq \ldots \geq g_n$, con $n > 0$, y $t_i > 0 \; \forall \; i \in \{1, ..., n\}$ }

Reordenar ejemplo:

i	**1**	**2**	**3**	**4**
$\mathbf{g_i}$	50	30	15	10
$\mathbf{t_i}$	2	1	2	1

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz

Entradas: vector de tiempos

i	0	1	2	3	4
t	0	2	1	2	1

La posición i=0 es para un valor centinela

Salidas: vector secuencia tareas

m	0	1	2=k
s	0	2	1

La posición m=0 es para un valor centinela.

k es el número de tareas en la secuencia de solución.

En cada posición m, con $1 \leq m \leq k$, se guarda el número de la tarea que se ejecuta en el instante de tiempo m, en la secuencia de solución.

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema algorítmico voraz

```
método secuencia ( t[0 .. n] ): k, array [1 .. k]
  t[0] = 0          // Inicialización, centinela en el vector t de fechas límite
  s[0] = 0          // centinela en el vector s con la secuencia solución
  k = 1             // k representa el número de tareas en la secuencia de solución
  s[1] = 1          // La tarea 1, con la mayor ganancia, siempre se elige.
  para i = 2 hasta n hacer  // Bucle voraz, orden decreciente de g_i
    r = k// parte de la secuencia de solución fijada en cada etapa
    mientras ( t[s[r]] > max (t[i], r ) ) hacer
          r = r-1
    fmientras
    si ( t[s[r]] ≤ t[i] ) y ( t[i] > r ) entonces
          para m = k decremento 1 hasta r+1 hacer
                s[m+1] = s[m]
          fpara
          s[r+1] = i; k = k+1
    fsi
  fpara
  retorna k, s[1..k]
fmétodo
```

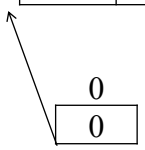

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz, ejemplo

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3
t_i	3	1	1	3	1	3

Entrada:

i	1	2	3	4	5	6
t	3	1	1	3	1	3



4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz, eficiencia

a) Ordenar las tareas por orden creciente de plazo: **O(n·log n)**

b) Repetir para i desde 1 hasta n:

- Elegir el próximo candidato: **O(1)**

- Comprobar si la nueva planificación es factible, y añadir la tarea a la solución en caso afirmativo: **O(i)** en el peor caso.

En total, el algoritmo es un **$O(n^2)$**

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz mejorado

Un conjunto de tareas J es factible si y sólo si podemos construir una sucesión factible que incluya a las tareas en J como sigue: para cada tarea $i \in J$, ejecutar i (aún no seleccionada) en el instante t, donde t es el mayor entero que verifica que $0 \leq t \leq \min(n, t_i)$, donde n es el número de tareas propuestas y t_i es la fecha límite para la tarea i.

En otras palabras: añadir cada tarea $i \in J$ a la sucesión que se está construyendo tan atrás como sea posible, pero no detrás de su fecha límite.

Si una tarea no puede ejecutarse a tiempo por su fecha límite, entonces no se añade a J, pues J no sería factible.

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz mejorado
Ejemplo:

i	**1**	**2**	**3**	**4**	**5**	**6**
$\mathbf{g_i}$	20	15	10	7	5	3
$\mathbf{t_i}$	3	1	1	3	1	3

Precondicion:

{ Pre: $g_1 \geq g_2 \geq \ldots \geq g_n$, con $n > 0$, y $t_i > 0 \; \forall \; i \in \{1, ..., n\}$ }

Entrada:

i	1	2	3	4	5	6
t	3	1	1	3	1	3

No incorporamos centinela

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz mejorado
Salida (básica):

k – Número de tareas de la solución

m	1	2	3=k
s	2	4	1

s – Solución, secuencia de tareas (2, 4, 1), indica posiciones.
m – Orden de las tareas
l – Número de tareas de la solución (tras comprimir = k)

Siendo l inicial:

l = mínimo (n, máximo (t_i, con $1 \leq i \leq n$))

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz mejorado

Estructuras auxiliares:

m	1	2	3=l
f	1	2	3

Valores iniciales

f – Indica la posición disponible para una tarea con fecha límite el valor de índice.

Conjuntos:

C_i = valores de tiempo que se pueden colocar como más tarde en la posición i, es decir f(c∈ C_i) = i
Inicialmente C_i = {i}

K = Conjunto de valores límite de tiempo asociados a la posición que se está considerando.

L = Conjunto de valores límite de tiempo asociados a la siguiente posición libre…

4 – Ejemplos: Planificación de tareas.

Planificación con fechas límite: esquema voraz mejorado

```
método secuencia2 ( t [ 1..n ] ): k, array [ 1..k ]
   l = min ( n, max { t [i] con 1 ≤ i ≤ n } )      // Inicialización
   para i = 1 hasta l hacer
      s[i] = 0;   f[i] = i
      Inicializar el conjunto { i }
   fpara
   para i = 1 hasta n hacer // Bucle voraz,  en orden decreciente de g_i
      k = min ( n, t[i] )      // k representa el límite para la tarea i
      K = conjunto que contiene el valor k
      m = f[k]                 // Posición donde posible colocar tarea i
      si m ≠ 0 entonces        // Si m = 0 se rechaza la tarea
         s[m] = i              // Colocamos la tarea en su posición de la solución.
         L = conjunto que contiene el valor m-1
         para todo elemento e ∈ K hacer
            f[e] = f[m-1]      // Posiciones de f en K, valor posición de f de L
         fpara
         Unir los conjuntos K y L // Valor posiciones de f nuevo conjunto las de L.
      fsi
   fpara
fmétodo
```

// Queda comprimir la solución.

4 – Ejemplos: Planificación de tareas.

```
método secuencia2 ( t [ 1..n ] ): l, array [ 1..l ]
   l = min ( n, max { t [i] con 1 ≤ i ≤ n } )      // Inicialización
   ……
   para i = 1 hasta n hacer // Bucle voraz,  en orden decreciente de g_i
      k = min ( n, t[i] )      // k representa el límite para la tarea i
      K = conjunto que contiene el valor k
      m = f[k]                 // Posición donde posible colocar tarea i
      ……..
   fpara
   // Comprimir la solución.
   k = 0
   para i = 1 hasta l hacer
      si j [ i ] > 0 entonces
         k = k+1
         s[k] = s[i]
      fsi
   fpara
   retorna k, j [1..k]
fmétodo
```

Planificación con fechas límite: esquema voraz mejorado

4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: definición del problema

Se trata de colorear los nodos de un grafo (no dirigido) G = <N, A> de forma que no haya dos nodos adyacentes del mismo color, con el objetivo de utilizar el *mínimo número de colores* posible.

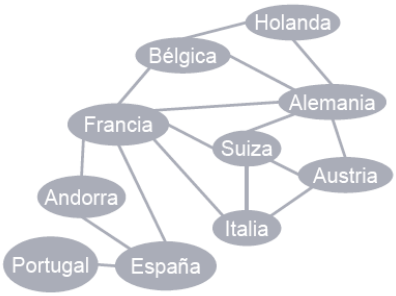
Si no se utilizan más de k colores diferentes, entonces es un *coloreado k.*

El menor k tal que existe un coloreado k del grafo se llama *número cromático del grafo* y cualquier coloreado k de éstos será un coloreado óptimo.

Es uno de los problemas NP-completos clásicos (no existe ningún algoritmo de tiempo polinómico que lo resuelva de forma óptima, asegurándonos haber usado el mínimo número de colores).

4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: ejemplo, colorado de mapas

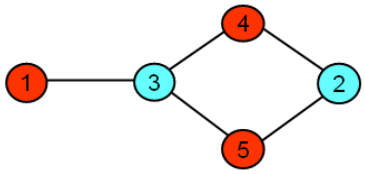


4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: planteamiento formal

Una solución S será un conjunto de pares de la forma (nodo, color), cumpliendo que para todo (n_i, c_i) y (n_j, c_j), si (n_i, n_j) es una arista del grafo, entonces $c_i \neq c_j$.

La solución es válida si para toda arista $(v, w) \in A$, se cumple que $c_v \neq c_w$.



S = ((1, 1) (2, 2) (3, 2) (4, 1) (5, 1))

Número de colores = 2 (solución óptima)

4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: algoritmo heurístico voraz

Inicialmente, ningún nodo tiene color asignado.

✓ Se toma un color no usado y un nodo (arbitrario) no coloreado aún.

✓ Se recorren todos los nodos que quedan sin colorear, pintando del color actual todos los que sea posible: aquéllos que no tengan ningún adyacente del color actual.

✓Se repiten los pasos hasta que se hayan coloreado todos los nodos.

La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los nodos estén coloreados.

➢ Función de *selección*: cualquier candidato restante.

➢ Función *factible*: se puede asignar un color al nodo actual si ninguno de sus adyacentes tiene ese mismo color.

4 – Ejemplos: Otros ejemplos.

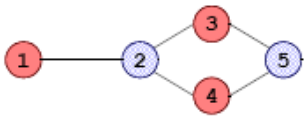
Coloreado de grafos: esquema voraz

```
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 adyacente de color c entonces
           nodosNoColoreados.extrae(nnc)
           nnc.colorea(c)
        fsi
     fpara
   fmientras
fmétodo
```

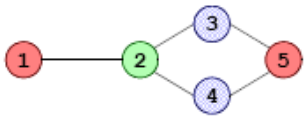
4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: discusión resultados

El algoritmo no garantiza la solución óptima (en el ejemplo, 2 colores).

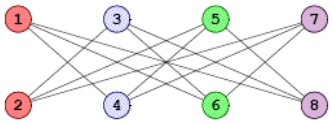


Orden: 1, 2, 3, 4, 5
2 colores

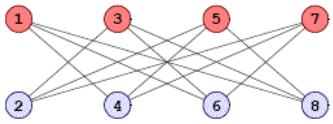


Orden: 1, 5, 2, 3, 4
3 colores

Los grafos bipartitos demuestran que la solución no es aproximada.



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



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

4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: discusión resultados

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.

Es un algoritmo *heurístico*, que tiene la posibilidad, pero no la certeza, de encontrar la solución óptima. Por ejemplo, para un grafo bipartito de n nodos la solución óptima es 2 colores, pero el algoritmo puede llegar a utilizar n/2.

No obstante, como todos los algoritmos exactos conocidos para el problema del coloreado de un grafo emplean un tiempo exponencial, es indudable la utilidad de la heurística voraz.

4 – Ejemplos: Otros ejemplos.

Coloreado de grafos: eficiencia

Sea n el número de nodos del grafo.

- Su eficiencia en el mejor caso es O(n).
- 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 ser necesario comprobar sus n-1 vecinos: O(n).

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: definición del problema

Los códigos de Huffman son una técnica muy útil para comprimir ficheros.

El algoritmo "greedy" de Huffman utiliza una tabla de frecuencias de aparición de cada carácter para construir una forma óptima de representar los caracteres con códigos binarios.

Ejemplo:

Se tiene un fichero con 100 000 caracteres que se desea compactar. Las frecuencias de aparición de caracteres en el fichero son las siguientes:

Carácter	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5

El archivo original ocuparía 800 000 bits (suponiendo ASCII.)

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: definición del problema

Se puede reducir el espacio que ocupa la codificación de los distintos caracteres.

Puede usarse un código de longitud fija (supongamos de 3 bits, para nuestro ejemplo). El fichero requeriría en ese caso 300.000 bits.

Carácter	a	b	c	d	e	f
Código (longitud fija 3)	000	001	010	011	100	101

Puede hacerse mejor con un *código de longitud variable*, dando codificaciones cortas a los caracteres más frecuentes, y más largas a los menos frecuentes:

Carácter	a	b	c	d	e	f
Código (longitud variable)	0	101	100	111	1101	1100

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: definición del problema

Código de longitud variable:

Carácter	a	b	c	d	e	f
Código (longitud variable)	0	101	100	111	1101	1100

Este código ahorra algo más del 25% sobre el de longitud fija, (requiere 224.000 bits en lugar de 300.000).

Se precisa un *código libre de prefijos*:

- ✓ Ninguna codificación puede ser prefijo de otra.
- ✓ De esta forma, la decodificación es inmediata, pues no hay ambigüedades.

Por ejemplo: ‘001011101’ sólo puede ser ‘aabe’.

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: definición del problema

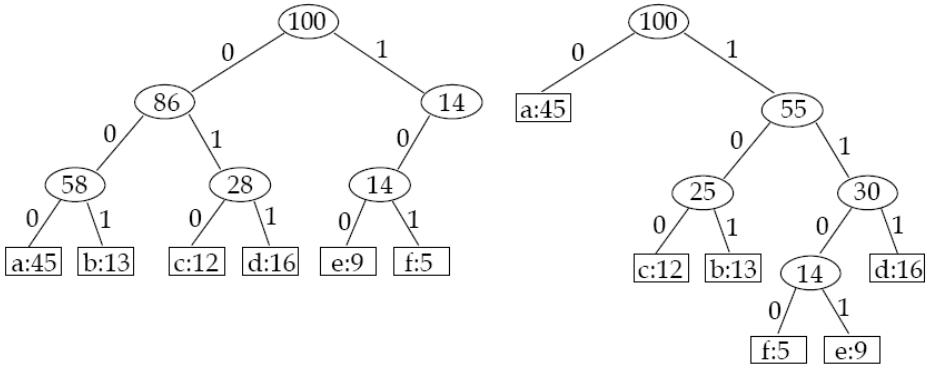
El código se representa mediante un *trie* (árbol lexicográfico):

- ✓Árbol binario cuyas hojas son los caracteres codificados.
- ✓El código de cada carácter es el camino desde la raíz hasta la hoja, donde '0' significa ir al hijo izquierdo, y '1' significa ir hacia el derecho.
- ✓Cada hoja está etiquetada con un carácter y su frecuencia.
- ✓Cada nodo interno está etiquetado con la suma de los pesos (frecuencias) de las hojas de su subárbol.
- ✓Un código *óptimo* siempre está representado por un árbol *lleno*: cada nodo interno tiene dos hijos. En ese caso, si el alfabeto que se quiere codificar es C, entonces el árbol del código óptimo tiene |C| hojas y |C|-1 nodos internos.

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: definición del problema

Ejemplo, Árboles de los dos códigos anteriores:



4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: algoritmo "greedy"

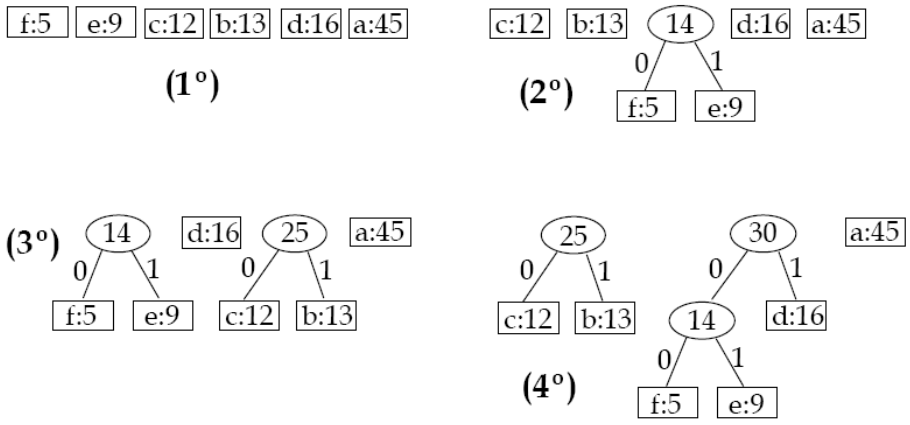
El algoritmo *voraz* de Huffman construye el árbol A de un código óptimo, desde abajo hacia arriba (desde las hojas hacia la raíz):

- Utiliza una cola de prioridad Q de árboles (las frecuencias hacen de prioridades).
- Empieza con un conjunto de |C| hojas en Q y realiza una secuencia de |C|-1 mezclas, hasta crear el árbol final.
- En cada paso, se mezclan los dos árboles de Q que tienen menos frecuencia, y el resultado es un nuevo árbol, cuya frecuencia es la suma de las frecuencias de los dos árboles mezclados.

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: algoritmo "greedy"

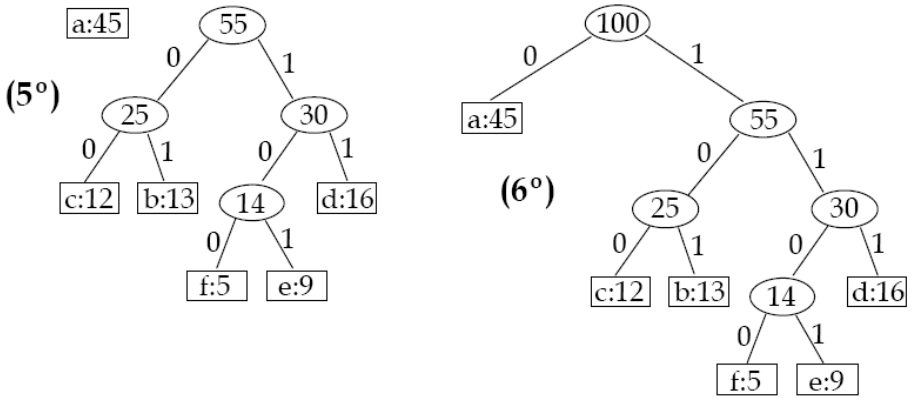
Ejemplo de construcción del árbol:



4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: algoritmo "greedy"

Ejemplo de construcción del árbol:



4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: esquema algoritmo "greedy"

{Pre: c conjunto de caracteres, f vector de frecuencias de esos caracteres.}

método **huffman(conjunto c, vectorFrec f): arbol**
  colaPrioridad q *// Cola de prioridad de árboles con criterio del mínimo.*
  entero i, fx, fy, fz
  arbol z, x, y
  colaPrioridad.creaVacía(q)
  para **todo x en c** hacer *// Se insertan los árboles*
    colaPrioridad.inserta(q, <x,f[x]>)
  fpara
  para **i=1 hasta |C|-1** hacer
    <x,fx>=colaPrioridad.extraePrimero(q) // Se extrae árbol mínimo
    <y,fy>=colaPrioridad.extraePrimero(q)
    fz = fx + fy
    z=arbol.crearArbol(fz, x, y) *// Crea un árbol de raíz z*
    colaPrioridad.inserta(q, <z,fz>)
  fpara
  <z,fz>=colaPrioridad.extraePrimero(q)
  retorna z
fmétodo

{Post: z es el árbol de un código libre de prefijos óptimo para (c, f).}

4 – Ejemplos: Otros ejemplos.

Códigos de Huffman: algoritmo “greedy”, eficiencia

Coste temporal:

- Inicialización de Q: O(|C|).
- El interior del bucle: O(log |C|).
- Coste total: O(|C| log |C|).

n, el tamaño del problema, es |C|

4 – Ejemplos: Otros ejemplos.

Asignación de parejas: definición del problema

El *problema de la asignación de parejas*, utilizando tablas de afinidad, busca la creación de parejas de elementos maximizando la afinidad total del conjunto de parejas.

	0	**1**	**2**	**3**
0	0	85	88	47
1	13	0	54	4
2	34	6	0	78
3	48	69	73	0

Cada posible candidato a formar una pareja muestra una afinidad con cada uno de los otros posibles candidatos.

Se trata de un problema NP-completo, que se puede resolver con una heurística voraz.

4 – Ejemplos: Otros ejemplos.

Asignación de parejas: definición del problema

El caso de la asignación de compañeros de trabajo, puede ser un ejemplo de este tipo de problema. Una empresa de reparación de averías telefónicas divide a sus operarios por parejas para realizar las reparaciones.

La dirección de la empresa considera fundamental que los componentes de una pareja se lleven lo mejor posible, por lo que solicita a cada operario que indique su afinidad con cada uno de sus compañeros.

La afinidad se indica de forma numérica, siendo 100 la máxima afinidad y 0 la mínima. Con los resultados obtenidos en la encuesta, la dirección construye la matriz de afinidades.

4 – Ejemplos: Otros ejemplos.

Asignación de parejas: definición del problema

La afinidad se indica de forma numérica, siendo 100 la máxima afinidad y 0 la mínima. Con los resultados obtenidos en la encuesta, la dirección construye la matriz de afinidades. Por ejemplo, la matriz de afinidades para una empresa con 4 operarios podría ser:

	0	**1**	**2**	**3**
0	0	85	88	47
1	13	0	54	4
2	34	6	0	78
3	48	69	73	0

Cada posible candidato a formar una pareja muestra una afinidad con cada uno de los otros posibles candidatos.

4 – Ejemplos: Otros ejemplos.

Asignación de parejas: definición del problema

	0	1	2	3
0	0	85	88	47
1	13	0	54	4
2	34	6	0	78
3	48	69	73	0

Cada posible candidato a formar una pareja de trabajadores muestra una afinidad con cada uno de los otros posibles candidatos.

En la matriz, cada celda de afinidades[i][j] indica la afinidad indicada por el operario i con respecto al operario j.

La afinidad de la pareja (i, j) será la suma de las afinidades indicadas por ambos operarios: afinidades[i][j]+afinidades[j][i].

El algoritmo voraz heurístico encontrará soluciones sub-optimas para el problema en tiempo polinómico.

4 – Ejemplos: Otros ejemplos.

Asignación de parejas: heurística voraz

Se puede definir una solución voraz sub-óptima, siguiendo el esquema voraz, con las siguientes características:

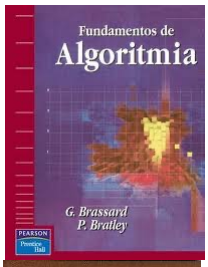
- La entrada es una matriz de afinidades. La salida es un vector en que se indica la pareja que se le asigna a cada elemento. La salida se inicializa a -1, que significa que el candidato no está emparejado.

- Se van recorriendo los candidatos no emparejados, bucle voraz. Este bucle voraz se repetirá n/2 veces.

- La selección de la pareja de un candidato libre se realiza asignándole aquel posible partner libre que ofrece la mayor afinidad compuesta.

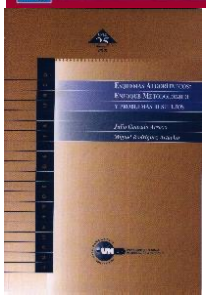
- La afinidad compuesta es la suma de las afinidades de una pareja.

Material a estudiar



Fundamentos de algoritmia
G. Brassard, T. Bratley
Prentice Hall, D.L. 2004

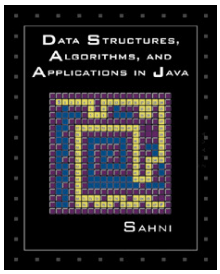
Biblioteca: 519 BRA fun
Capítulo 6. Algoritmos voraces.

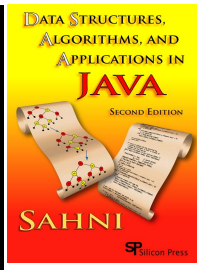


Esquemas algorítmicos. Enfoque metodológico y problemas resueltos
Julio Gonzalo Arroyo y Miguel Rodríguez Artacho)
ISBN: 84-362-3622-X

Biblioteca: 519 GON esq
Capítulo 2. Algoritmos voraces.

Material a estudiar





Data Structures, Algorithms, and Applications in Java
Sartaj Sahni
McGraw Hill, 2000 / Silicom Press, 2005
Biblioteca: Agotado – Esperando 2ª edición. Capítulo 18.

Ejercicios recomendados - Libres

10. Aplicar Dikstra para evaluar la mejor localización de un Centro Logístico, aquella ubicación en que el coste de distribuir los productos es mínimo. Dikstra nos da los caminos mínimos para alcanzar todas las otras ubicaciones desde un punto dado; nosotros deseamos encontrar el punto desde dónde es mejor realizar la distribución. Se ha de:

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba generados aleatoriamente (dado el tamaño del problema).
- Almacenar juegos de prueba, resultados y tiempos de ejecución.
- Analizar la eficiencia obtenida empíricamente frente a la teórica.

Ejercicios recomendados - Libres

11. Proponer una solución con el esquema voraz al problema de la mochila acotado (0/1/../m). De cada producto i, hay m_i unidades, las unidades no son fraccionables, pero la partida si lo es. Ha de:

- Presentar el esquema algorítmico detallado (pseudocódigo).
- Seleccionar las estructuras de datos adecuadas para almacenar los datos.
- Implementar el correspondiente código en Java.
- Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba generados aleatoriamente (dado el tamaño del problema).
- Almacenar juegos de prueba, resultados y tiempos de ejecución.
- Analizar la eficiencia obtenida empíricamente frente a la teórica.

Ejercicios recomendados - Libres

2. Realizar un estudio comparativo de las distintas funciones heurísticas de selección que se propusieron para el problema de la "Selección de tareas". Se ha de analizar el efecto utilizando 5 grupos juegos de prueba con el 10, 25, 40, 65 y 80 por ciento de superposición de tareas (las tareas se superponen un X% si durante un X% del tiempo se puede ejecutar más de una tarea). Probar con 3 tamaños (5, 10 y 15 tareas).
   - Presentar el esquema algorítmico detallado (pseudocódigo).
   - Seleccionar las estructuras de datos adecuadas para almacenar los datos.
   - Implementar el correspondiente código en Java.
   - Almacenar juegos de prueba, resultados y tiempos de ejecución.
   - Analizar la eficiencia obtenida empíricamente frente a la teórica.
   - Analizar el valor obtenido para las distintas funciones de optimización vistas.

Ejercicios recomendados - Libres

3. Desarrollar con más detalle el problema de asignación de parejas. ¿Qué ocurre si se le da un cierto valor a la similitud de afinidades? En todos los casos se ha de:
   - Presentar el esquema algorítmico detallado (pseudocódigo).
   - Seleccionar las estructuras de datos adecuadas para almacenar los datos.
   - Implementar el correspondiente código en Java.
   - Evaluar experimentalmente la eficiencia utilizando diversos juegos de prueba.
   - Almacenar juegos de prueba, resultados (indicando quiénes juegan) y tiempos de ejecución.
   - Analizar la eficiencia obtenida empíricamente frente a la teórica.
   - Hacer una crítica a la solución, buscando ejemplo problemáticos (problema de la desigualdad de afinidades).