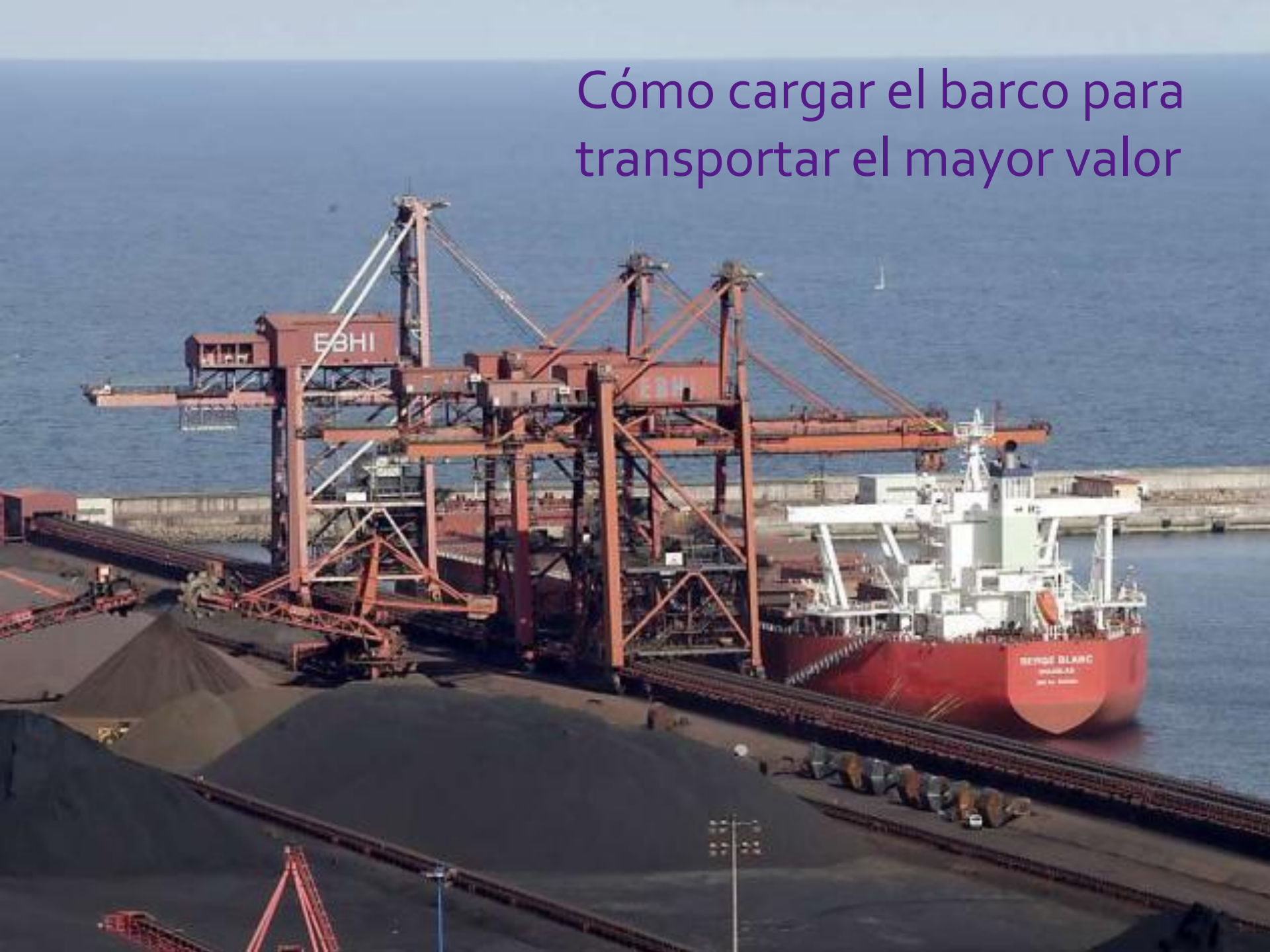
Algoritmia
Grado en Ingeniería Informática del Software
Escuela de Ingeniería Informática – Universidad de Oviedo

Algoritmos Voraces (Greedy)

Juan Ramón Pérez Pérez

jrpp@uniovi.es

Cómo cargar el barco para transportar el mayor valor



Problema de la mochila

n objetos / tipos y una "mochila" para transportarlos.

Para cada objeto $i = 1, 2, \ldots n$ tiene un peso w_i y un valor v_i.

La mochila puede llevar un peso que no sobrepase W.

Objetivo: maximizar valor de los objetos transportados respetando la limitación de peso.

Mochila con fragmentación

Suponemos que **se pueden dividir los objetos**.

Podemos asignar a la mochila una fracción x_i del objeto i, con $0 <= x_i <= 1$.

Correspondencia con la carga del barco

n tipos de minerales y el buque de carga es la "mochila" para transportarlos.

Para cada tipo $i = 1, 2, \ldots n$ tenemos un peso w_i y un valor v_i.

El carguero puede llevar un peso que no sobrepase *W*.

Objetivo: maximizar valor de los objetos transportados respetando la limitación de peso.

Algoritmos voraces

- Los algoritmos **voraces** se utilizan en problemas de **optimización**.
- La solución se *construye* paso a paso.
- Basan la búsqueda de la solución óptima al problema en buscar en cada paso la solución localmente óptima.
- Para buscarla suelen emplear **heurísticos**, reglas basadas en algún tipo de conocimiento sobre el problema.

Elementos que se deben identificar en el problema

Conjunto de **candidatos**, las *n* entradas del problema.

Función de selección que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.

Función que compruebe **subconjunto de candidatos es factible**: Que sea posible encontrar una solución. Si no lo es rechazaríamos al último candidato.

Función que compruebe si un subconjunto es **solución**.

Función objetivo que determine el valor de la solución hallada.

Toma de decisiones en algoritmos voraces

Estos algoritmos toman **decisiones** basándose en la información que tienen disponible **localmente**

- Confiando en que estas decisiones conduzcan a la solución óptima buscada.

Nunca se reconsidera una decisión

- No hay necesidad de emplear procedimientos de seguimiento para deshacer las decisiones.

Fáciles de diseñar y programar.

Concretando esquema general para este problema

- **Candidatos**: los n objetos.
- **Función de selección** de un nuevo elemento: en función de un heurístico.
  - Estado del problema: vector / tupla $X=(x_1,x_2,x_3..x_n)$.
- Función para comprobar si una **solución es factible**:

$$\sum_{i=1}^{n} w_i \leq W$$

Datos del problema concreto

Mineral	1	2	3	4	5
w (Tm)	10	20	30	40	50
v (x1000€)	20	30	66	40	60

n=5, W=100

¿Cuál es el mejor heurístico para la función de selección?

Mineral	1	2	3	4	5
w (Tm)	10	20	30	40	50
v (x1000€)	20	30	66	40	60

- Posibilidades:
  - Seleccionar objeto más valioso.
  - Seleccionar objeto menos pesado.

Objetos	1	2	3	4	5	Valor
Decreciente v_i	0	0	1	0,5	1	146
Creciente w_i	1	1	1	1	0	156

Mejorando el heurístico

- Heurístico que seleccione el objeto restante con mayor coeficiente *valor / peso*
- Este heurístico nos proporciona la solución óptima.
- Estos datos han servido de contraejemplo para demostrar que los dos primeros heurísticos no proporcionan la solución óptima.

Objetos	1	2	3	4	5	Valor
Valor / peso	2	1,5	2,2	1	1,2	
Decreciente *v/w*	1	1	1	0	0,8	164


Mochila 0/1

Mochila sin fragmentación

En la variante mochila 0/1 **NO se pueden dividir los objetos**.

Podemos asignar a la mochila un objeto *i* o no asignarlo,

con $x_i = (0 \mid 1)$.

¿Puede comportar este cambio en los requisitos, cambios en el comportamiento del heurístico, y por tanto del algoritmo?

Contraejemplo para demostrar algoritmo NO óptimo

Mineral	1	2	3
w (Tm)	10	10	12
v (x1000€)	20	20	36
v/w	2	2	3

n=3, W=20

- Heurístico v/w → resultado v= 36
- Valor óptimo v= 40

Esquema general algoritmos voraces

```
public Estado realizarVoraz(List candidatos)
{
    Estado estadoActual= new Estado();

    while (!candidatos.esVacio() && !estadoActual.esSolucion())
    {
        /* Elige la mejor componente
         * en función de un determinado heurístico */
        Object x= SeleccionarCandidato(candidatos);  // heurístico

         if (estadoActual.esFactible(x))
            estadoActual.add(x);
    }

    if (estadoActual.esSolucion())
        return estadoActual;
    else
        return null;  // No se ha encontrado una solución
}
```

Características algoritmos voraces

- La principal característica de estos algoritmos:
  - Son rápidos y fáciles de implementar,
  - Dependen totalmente de la calidad del heurístico
- Si iteramos sobre el conjunto de candidatos: $O(n^2)$
- Si se hace una ordenación previa del conjunto de candidatos, de tal forma que:
  - La función de selección no tiene que recorrer todo el array para buscar un nuevo candidato.
  - Con array ordenado seleccionamos o descartamos cada elemento avanzando cada vez una posición.
  - Complejidad: O(n) + la complejidad de la ordenación

Influencia del heurístico en este tipo de algoritmos

- Está técnica está orientada a problemas de optimización
- Un heurístico de baja calidad podría proporcionar soluciones no óptimas
- En el extremo podría no proporcionar soluciones, aunque el problema las tenga.
- Son adecuados para aquellos casos en que:
  - por la calidad del heurístico se encuentra la solución óptima,
  - no se dispone de tiempo para aplicar otras técnicas que la encuentren.
- Hay problemas que no se pueden resolver correctamente con este enfoque.

Ejemplos de algoritmos voraces

El problema del cambio (I)

- Diseñar un algoritmo para pagar una cierta cantidad, utilizando el menor número posible de monedas.
- Ejemplo:
  - Tenemos que pagar 2,89 €. Sistema monetario euro.
- ¿Heurístico?
  - **Función selección**: Dar la moneda de mayor valor posible
  - **Función factibilidad**: NO exceda lo que queda por devolver.
  - Solución: 1 moneda de 2 €, 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. → Solución óptima.

El problema del cambio (II)

- Dependiendo del sistema monetario y la disponibilidad de monedas, el heurístico **no siempre es óptimo**.
- Lo demostramos con un **contraejemplo**:
  - Pagar 0,60 €.
  - Monedas disponibles: 0,50 €, 0,20 €, 0,02 €.
  - Solución heurístico: 0,50 € + 5· 0,02 € → 6 monedas.
  - Solución óptima: 3· 0,20 € → 3 monedas.
- Este problema se podría resolver con *backtracking*, desarrollando todas las soluciones o cambios posibles.

Implementación del problema del cambio

```
public void calcularCambio(int[] monedas, int cantidad)
{
    int tipoMon= 0;
    while (cantidad>0 && tipoMon<monedas.length)
    {
        // Estado es factible / valido
        if (cantidad-monedas[tipoMon]>=0)
        {
            // nueva moneda en el conjunto solución
            solucion[tipoMon]++;
            cantidad-= monedas[tipoMon];
        }
        else
        // pasa a comprobar la moneda de valor inferior
            tipoMon++;
    }
}
```

Algoritmos voraces que trabajan con grafos

- Búsqueda del camino mínimo
  - **Dijkstra**
  - Simulación:
  - https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
- Árbol de búsqueda por algoritmo D.

Algoritmos voraces que trabajan con grafos

- Árboles de recubrimiento mínimo para grafos:
  - **Kruskal**.
    - Simulación:
    - https://www.cs.usfca.edu/~galles/visualization/Kruskal.html
  - **Prim**.
    - Simulación
    - https://www.cs.usfca.edu/~galles/visualization/Prim.html

El fontanero diligente

- Un fontanero necesita hacer *n* reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea *i*-ésima tardará t_i minutos
- Necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.
- Debemos diseñar un algoritmo voraz que resuelva el problema y probar su validez, bien mediante demostración formal o con un contraejemplo que la refute.
- Este es un problema típico de planificación.

El fontanero diligente (II)

- Si llamamos E_i a lo que espera el cliente *i*-ésimo hasta que le reparen su avería por completo, necesita minimizar la expresión:
- $E(n) = \sum_{i=1}^{n} E_i$
- El fontanero siempre tardará el mismo tiempo global $T = t1 + t2 + \ldots + tn$ en realizar todas las reparaciones, independientemente de la forma en que las ordene
- Los tiempos de espera de los clientes sí dependen de esta ordenación.

El fontanero diligente (III)

- Formalmente: debemos encontrar es una permutación de las tareas en donde se minimice la expresión de $E(n)$
- La permutación óptima es aquella en la que los avisos se atienden en orden creciente de sus tiempos de reparación.
- Heurístico:
  - **Función selección**: Escoger la tarea con menor tiempo de reparación
  - **Función factibilidad**: No hay ninguna condición en este problema ya que no hay tiempos límites y tenemos que realizar todas las tareas

Ejercicio propuesto

- Sea el algoritmo de **Prim** para la creación de un árbol de recubrimiento mínimo para un grafo.
- Este es un algoritmo voraz. Analizar el comportamiento y comprobar que cumple con las características de este tipo de algoritmos.
- ¿Que complejidad tiene si las aristas no están ordenadas? ¿Podríamos mejorar la complejidad si las ordenamos (en relación a qué las tenemos que ordenar)?
- Estudiar la implementación de este algoritmo.

Resuelto en el seminario 3

Ejercicio propuesto 2

Problema del viajante de comercio

Dado un grafo no dirigido, ponderado y completo (sin ciclos), encontrar un **ciclo simple que incluya todos los nodos** (**ciclo de Hamilton**) y cuyo **coste sea mínimo**.

Al ser completo se podría demostrar que el camino mínimo es un ciclo simple y, además, no depende del nodo de partida.

- Se supone que hay arista entre cualquier par de nodos.
- Cálculo del coste de cada arista: distancia euclídea entre los nodos.

Resuelto en el seminario 5

Resolver el problema del viajante de comercio

- 6 nodos situados en las coordenadas indicadas.