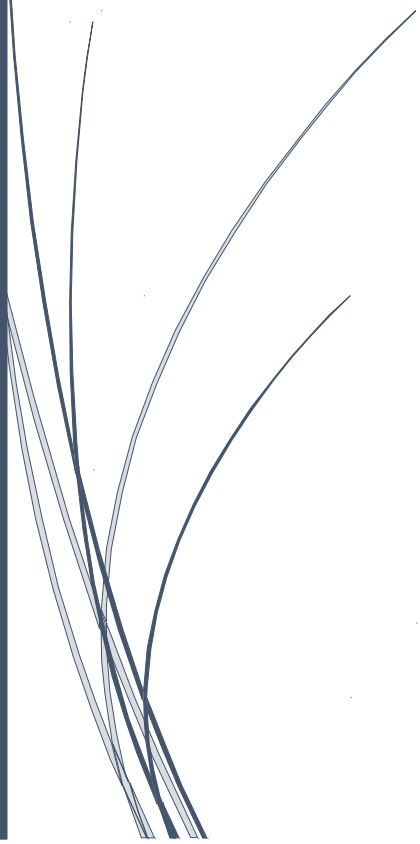
26-5-2017

Práctica 3

Algoritmos Greedy



Pablo Álvarez Cabrera
Johanna Capote Robayna
Cristina de la Torre Villaverde
Guille Galindo Ortuño
Adolfo Soto Werner

Índice

- Análisis del problema.
- Diseño de la solución.
- Pseudocódigo del algortimo.
- Explicación del funcionamiento del algoritmo.
- Ejemplo real donde aplicar el algoritmo.
- Cálculo de la eficiencia teórica.
- Compilación y ejecución del algoritmo.

Análisis del problema.

El objetivo de esta práctica implementar un algoritmo Greedy que solucione el problema del embajaje (bin packing problem).

Este problema consiste en buscar la mejor forma de almacenar un número determinado de elementos en cajas de un mismo volumen, de forma que las cajas estén lo mas llenas posible y, por tanto, se use el menor número posible de estas.

Para ello, partimos de un número indefinido de cajas con un volumen V y de un conjunto de N objetos a empaquetar, cada uno de ellos con un volumen v_i, de forma que debemos empaquetar los distintos elementos de manera que la suma de todos los volúmenes de los objetos que hay en cada caja no supere al volumen total de la caja en la que se encuentran. Es decir:

$$\sum v_i \leqslant v$$

El algoritmo debe proporcionar una solución que implique usar el menor número de cajas posible, aunque la solución dada no sea la única posible.

Diseño de la solución.

Para implementar el algoritmo se han usado dos clases, “*Problema*” y “*Solucion*”, que facilitan el funcionamiento del mismo.

La clase “*Problema*” representa el problema que queremos resolver. Para ello, usamos un vector indexado (*v*) donde los índices se corresponden con los distintos objetos a empaquetar y el valor con el volumen de cada uno de ellos. Así, *v[i]* contendrá el volumen del objeto *i*. Además, la clase almacena el número de objetos a empaquetar y el volumen de las cajas, y proporciona métodos para consultar estas variables y para leer un problema determinado desde un fichero.

Por otro lado, la clase “*Solucion*” se usa para representar la solución que se obtiene al usar el algoritmo. Esta solución se almacena en un vector de vectores de pares <*int,double*>, es decir, un vector dinámico de embalajes donde cada embalaje es un vector con los objetos que contiene, y cada objeto está representado por un par de números (número de objeto y volumen). Si llamamos “*posiciones*” a dicho vector, *posiciones[i]* contendrá todos los objetos almacenados en la caja *i.* La clase proporciona además métodos para ir modificando la solución conforme se va obteniendo, y métodos para consultarla.

En cuanto al algoritmo, hemos identificado las siguientes componentes greedy:

- Lista de candidatos: Los N objetos a empaquetar
- Lista de candidatos utilizados: Los objetos que ya se han empaquetado
- Función solución: Se han empaquetado los N objetos
- Criterio de factibilidad: El volumen de los objetos contenidos en una caja es menor o igual que el volumen de la caja
- Función de selección: Se selecciona el objeto de mayor volumen
- Función objetivo: Minimizar el número de cajas necesarias

Pseudocódigo del algoritmo.

El pseudocódigo del algoritmo es el siguiente:

```
AlgorimoGreedyBinPacking(Problema p)

S ← Ø      // S es la solución a construir
P= conjunto de N elementos a empaquetar
Añadimos una caja vacía a S
Mientras P != Ø hacer:
    x= mayor elemento de P
    P= P\{x}
   //C es el vector de cajas de la solución S
   //k= número de cajas en S
    Si {x} cabe en C[i] para i=0,…,k-1
        C[i]= C[i] U {x}
    Si no
        Añadimos una caja vacía (C[k]) a S
        C[k]= {x}
Fin-mientras
Devuelve S
```

La lista de candidatos *P* se corresponde con el vector *v* de la clase “*Problema*” y la lista de cajas con el vector *posiciones* de la clase “*Solucion*”.

El funcionamiento se explica de forma más detallada en la siguiente sección.

Explicación del funcionamiento del algoritmo.

El funcionamiento de nuestro algoritmo se basa en buscar el elemento de mayor volumen en la lista de candidatos, mientras esta no esté vacía, para introducirlo en una caja. Una vez seleccionado se quita de la lista de candidatos y se introduce en la caja correspondiente, siguiendo el siguiente criterio:

Si resulta que, por su volumen, no cabe en ninguna caja de las que tenemos hasta el momento, se añade otra nueva y se introduce en ella. En caso contrario, se introduce en la primera caja disponible.

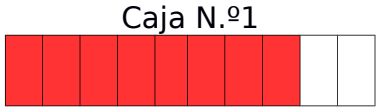
Después de esto se pasa a buscar el siguiente más grande y se va repitiendo el mismo proceso, hasta que la lista de candidatos esté vacía, lo que significa que se han empaquetado todos los elementos.

Para ilustrar el funcionamiento, proponemos el siguiente ejemplo:

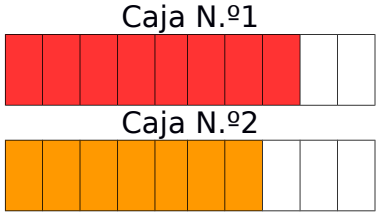
Sea P={8,5,4,7,3,1} un vector donde P[i] representa el volumen del objeto i. Supongamos que contamos con cajas suficientes para embalar dichos objetos, cada una de ellas con un volumen V=10.

Al principio contamos con una sola caja (caja nº1).

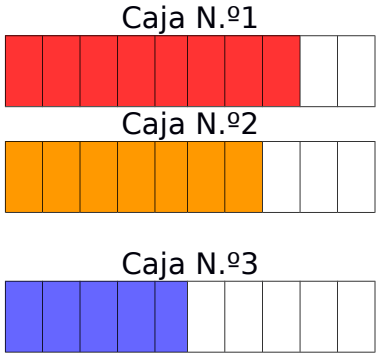
El primer paso del algoritmo es buscar el elemento mas grande del vector (en nuestro caso 8) y quitarlo del vector de candidatos. Como cabe en la única caja disponible lo introducimos. Nuestra lista de candidatos sería ahora P = {5,4,7,3,1}.



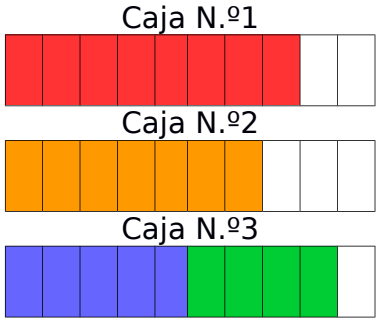
El siguiente elemento que encontramos es el que tiene volumen 7. Como no cabe en la primera caja que tenemos parcialmente llena, añadimos otra e introducimos este objeto. Nuestra lista de candidatos sería ahora P = {5,4,3,1}.



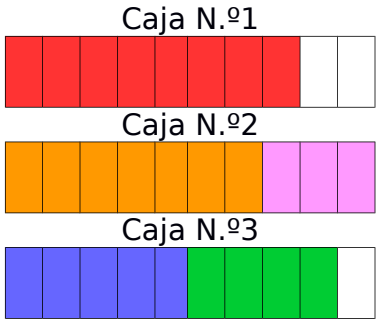
El siguiente elemento que encontramos es el que tiene volumen 5. Como no cabe en la primera caja ni tampoco en la segunda, añadimos otra e introducimos este objeto. Nuestra lista de candidatos sería ahora P = {4,3,1}.



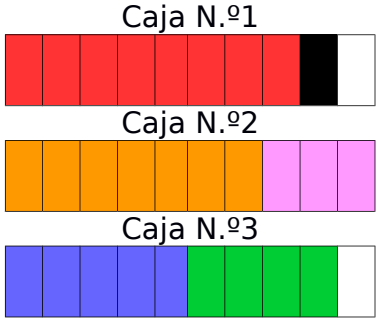
El siguiente elemento que encontramos es el que tiene volumen 4. No cabe en la primera caja, tampoco en la segunda, pero sí en la tercera. Introducimos entonces este elemento en la caja N.º 3. Nuestra lista de candidatos sería ahora P = {3,1}.



El siguiente elemento que encontramos es el que tiene volumen 3. No cabe en la primera caja que tenemos parcialmente llena pero si en la segunda. Introducimos entonces este elemento en la caja N.º2. Nuestra lista de candidatos sería ahora P = {1}.



El último elemento que nos queda por empaquetar es el que tiene volumen 1. Como cabe en la caja N.º1, lo introducimos en ella. Nuestra lista de candidatos sería ahora P = {}.



Como nuestra lista de candidatos se ha quedado vacía significa que el algoritmo ha terminado y tenemos un vector (al que denotaremos por V) de vectores (que denotaremos por V_i) de pares con 3 entradas (3 cajas). En V_1 estarán los pares <8,0> y <1,5>. En V_2 estarán los pares <7,3> y <3,4>. Por último, en V_3 estarán los pares <5,1> y <4,2>.

Ejemplo real donde aplicar el algoritmo.

Una situación en la que este algoritmo se podría aplicar sería, por ejemplo, en una empresa que necesite transportar paquetes, como puede ser la empresa cervecera Alhambra.
En esta empresa, a la hora de realizar el transporte de las cajas de cerveza que van saliendo de la fábrica hacia el almacén correspondiente, se necesita optimizar estos trayectos, ya que cuantos más desplazamientos se hagan mayor será el coste de estos y se necesitará más tiempo.

Suponiendo que todos los camiones tienen la misma capacidad y que todas las cervezas se empaquetan en cajas de un volumen determinado dependiendo de su tipo, aplicar este algoritmo supondría encontrar la manera de guardar las cajas de cerveza en los camiones de manera que se necesiten el menor número de camiones posible y, de esta forma, la empresa ahorre en el gasto de gasolina que supone el transporte.

Eficiencia teórica del algoritmo.

En este apartado, se analiza la eficiencia del algoritmo implementado, cuyo código es el siguiente:

```
Solucion AlgorimoGreedyBinPacking(Problema p){
  Solucion S;
  vector<pair<int,double> > candidatos(p.getNumObjetos());
  for (int i = 0; i < p.getNumObjetos(); i++) { //O(n)
    candidatos[i].first = i;
    candidatos[i].second = p.getVolumen(i);
  }
  S.addCaja();                                  //O(1)
  while (!candidatos.empty()) {                 //O(n)
    vector<pair<int, double> >::iterator it, sol;
    pair<int, double> max(-1,0);

    for (it = candidatos.begin(); it != candidatos.end(); it++) { //O(n)
      if (max.second < it->second) {
        max = *it;
        sol = it;
      }
    }
    candidatos.erase(sol);
    bool add = false;
    for (int i = 0; i < S.getNumCajas() && !add; i++) {
      if (S.getVolumen(i) + max.second <= p.getVolumen()){
        S.addObjeto(i,max);
        add = true;
      }
    }
    if(!add){
      S.addCaja();
      S.addObjeto(S.getNumCajas()-1, max);
    }
  }
  return S;
}
```

Como vemos, el número de elementos que tenemos que evaluar es el tamaño del conjunto de candidatos, es decir, ***p.getNumObjetos().***

Nos fijamos en que el algoritmo tiene tres bucles “principales”. El primero es un ***for*** que inicializa el vector de pares con el índice y volumen de cada elemento. El segundo es un bucle ***while*** que se recorre todos los elementos ya que en cada iteración borra un elemento de la lista. Dentro de este bucle tenemos otro bucle ***for*** que se recorre todos los posibles candidatos buscando el de mayor volumen. Finalmente tenemos otro bucle ***for*** que se recorre el vector de cajas para buscar en cual de ellas encaja el candidato de mayor volumen que ha encontrado previamente el anterior bucle.

Por tanto, tenemos que el primer bucle recorre todos los elementos del conjunto de candidatos y su eficiencia es:

$$\sum 1 = n \Rightarrow O(n)$$

En el segundo bucle tanto el ***while*** como el ***for*** que tiene dentro se recorren todos los elementos del conjunto de candidatos así que su eficiencia es:

$$\sum \sum 1 = \sum n = n \sum 1 = n^2 \Rightarrow O\left(n^2\right)$$

El tercer bucle se recorre el vector de cajas. Tomando que el número de elementos de este vector es n_c que es menor o igual que el número de candidatos tenemos que su eficiencia es:

$$\sum 1 = n \Rightarrow O\left(n_c\right)$$

Luego tenemos que la eficiencia total del algoritmo será:

$$O(n) + O\left(n^2\right) + O\left(n_c\right) = O\left(n + n^2 + n_c\right) = O\left(n^2\right)$$

Compilación y ejecución del algoritmo.

Para compilar el programa que incluye el algoritmo hemos creado un Makefile que compila el fichero ***main.cpp:***

```
SRC = src

INC = include

OBJ = obj

BIN = bin

CXX = g++

CPPFLAGS = -O2 -Wall -g -I./$(INC) -std=c++11


all: $(BIN)/binPacking


$(BIN)/binPacking : $(SRC)/main.cpp $(INC)/Problema.h $(SRC)/Problema.cpp $
(INC)/Solucion.h $(SRC)/Solucion.cpp $(INC)/Algoritmos.h $(SRC)/Algoritmos.cpp

    $(CXX) $(CPPFLAGS) -o $(BIN)/binPacking $(SRC)/main.cpp

# ************ Limpieza ************

clean :

    -rm $(OBJ)/* $(SRC)/*~ $(INC)/*~ ./*~


mrproper : clean

    -rm $(BIN)/*
```

La ejecución se realiza sin parámetros ya que por simplicidad solo lee datos de un archivo por defecto llamado ***Problema.dat*** que tiene el siguiente formato:

```
10      //Volumen máximo de cada caja

7       //Número de elementos que se van a introducir

1       //A partir de este número representan el volumen de cada objeto

2

8

3

6

5

2
```