



Análisis y Diseño de Algoritmos

Parcial 6 (Diciembre 2025)

Nombre:

Grupo:

1.(10 puntos) Parte teórica.

Una empresa dispone de un presupuesto P para inversiones y quiere decidir en qué proyectos invertirlo para obtener el **máximo beneficio total**. Cada uno de los n proyectos distintos requiere una inversión inicial c_i y producirá un beneficio esperado b_i , y no se puede financiar parcialmente, es decir, o se decide financiar completamente o no se financiará.

Se desea **diseñar** un algoritmo de **Ramificación y Poda** que una lista de ceros y unos indicando **qué proyectos se financiarán y cuáles no**. El algoritmo debe trabajar con los siguientes datos de entrada:

- Presupuesto P .
- Array C , donde c_i es el coste del proyecto i -ésimo
- Array B , donde b_i es el beneficio obtenido al financiar el proyecto i -ésimo.

- a) (10 ptos.) Diseñar un algoritmo de Ramificación y Poda para resolver el problema. Se debe indicar claramente la estructura de la solución, el estado inicial de la misma, la función de terminación, la función objetivo, la política de ramificación y la función de cota.

2. (10 puntos) Parte práctica.

Implementa el algoritmo de Ramificación y Poda diseñado en el ejercicio anterior.

Estructura de la solución -> Lista $S=[s_1,...,s_n]$, donde $s_i \in \{0,1\}$ indica si el objeto i -ésimo se incluyó (1) o no (0).

Estado inicial -> $S = []$

Política de ramificación-> Sea la solución parcial $S=[s_1,...,s_k]$, ¿qué valores son correctos para s_{i+1} ?

Rest. Explícitas (ramas posibles): $s_{i+1} \in \{0,1\}$

Rest, Implícitas(validez): el coste del proyecto s_{i+1} sumado al coste de los proyectos en S no debe superar P .

Es decir, el valor de s_{i+1} será correcto cuando $\sum_{i=1}^{k+1} s_i \cdot c_i \leq P$

F. Terminación-> $S.length = n$

F.objetivo-> $calidad(S) = \sum_{i=1}^n s_i \cdot b_i$ Hay que maximizarla.

F. cota-> $f(S) = g(S) + h(S)$

$$g(S) = \sum_{i=1}^k s_i \cdot b_i$$



$$h(S) = \left(P - \sum_{i=0}^k s_i \cdot c_i \right) \cdot \max_{k+1 \leq i \leq n} \{b_i/c_i\}$$

La implementación sería la siguiente:

```
Estado gestionarInversiones(int P, int[] c, int[] b) {
    ColaPrioridad cola = new ColaPrioridad();

    Estado inicial = new Estado(P,c,b); //Estado inicial

    Estado mejor = null;
    double cotaMejor = -1;

    cola.insertar(inicial);
    while (!cola.estaVacía()) {
        Estado actual = cola.extraer();

        if (actual.esCompleta()) { //Solución completa

            double cotaActual = actual.cota();
            if (cotaActual > cotaMejor) { // Actualizamos mejor
                mejor = actual;
                cotaMejor = cotaActual;
                cola.eliminar(cotaMejor); // Podamos
            }
        } else {
            List<Integer> opciones = actual.ramificar(); //ramas
            for (Integer opcion : opciones) {
                Estado sig = actual.extender(opcion);
                if (sig.cota() > cotaMejor) {
                    cola.insertar(sig);
                } //if cota
            } // for opcion
        } //if-else
    } //while

    return mejor;
}
```

La clase Estado estaría implementada de la siguiente manera:

```
public class Estado implements Comparable<Estado>{
    private List<Integer> sol; //solución contenida en el estado
    private int P; //presupuesto
    private int coste; //coste total de los proyectos financiados
    actualmente
}
```



```
private int beneficio; //beneficio total de los proyectos
financiados actualmente
private int n; //número de proyectos
private int[] c; //coste de cada proyecto
private int[] b; //beneficio de cada proyecto

/** Constructor para crear el estado inicial de la solución
**/
public Estado(int P, int[] c, int[] b) {
    this.P = P;
    this.c = c;
    this.b = b;
    this.n = c.length;
    sol = new ArrayList<Integer>();
    coste = 0;
    beneficio = 0;
}

//Constructor de copia. Crea una copia del estado "otro"
public Estado(Estado otro) {
    this(otro.P, otro.c, otro.b);
    sol.addAll(otro.getSol());
    this.coste = otro.coste();
    this.beneficio = otro.beneficio();
}

public List<Integer> getSol() {
    return sol;
}

/** Devuelve el coste total de los proyectos incluidos
    actualmente en la solución*/
public int coste() {
    return coste;
}

/** Devuelve el beneficio total proporcionado por los
    proyectos incluidos
    * actualmente en la solución
    */
public int beneficio() {
    return beneficio;
}

/** esCompleta devuelve true si la solución contenida en el
    estado es completa
    **/
public boolean esCompleta() {
    return sol.size() == n;
}
```



```
/**Devuelve la lista de candidatos válidos para extender el
estado actual de
* la solución.
* Precondición: el estado de la solución es parcial (no
completo)
**/
public List<Integer> ramificar(){
    List<Integer> candidatosValidos = new ArrayList<>();

    for (int i = 0; i<= 1; i++) {
        if (valido(i)) {
            candidatosValidos.add(i);
        }
    }

    return candidatosValidos;
}

//Comprueba si opcion es válida para ampliar la solución
contenida en el estado
private boolean valido(int opcion){
    return coste() + opcion * c[sol.size()] <= P;
}

//Crea una copia del estado actual de la solución y le añade
el elemento "opcion"
public Estado extender(int opcion){
    Estado nuevo = new Estado(this);

    int etapa = sol.size();

    nuevo.getSol().add(opcion);
    nuevo.coste += opcion * c[etapa];
    nuevo.beneficio += opcion * b[etapa];

    return nuevo;
}
```



```
/**
 * Devuelve el valor de cota del estado.
 */
public double cota() {
    double g = beneficio();
    double h = 0; //si es completa la estimación es 0.
    if (sol.size() < c.length) {
        //h = (P - coste()) * mejorRazonRestante();
        h = (P - coste()) *
            (double)b[sol.size()]/(double)c[sol.size()];
    }
    return g+h;
}
/**Precondición: quedan uno o más elementos por asignar a la
solución**/
private double mejorRazonRestante() {
    int etapa = sol.size();
    double mejor = Integer.MIN_VALUE;
    for(int i = etapa; i < c.length; i++) {
        double alternativa = (double)b[i]/(double)c[i];
        if (alternativa > mejor) {
            mejor = alternativa;
        }
    }
    return mejor;
}
@Override
public String toString(){
    return beneficio() + " " + getSol();
}
@Override
public int compareTo(Estado o) {
    return Double.compare(o.cota(), this.cota());
}
}
```