

TAREAS CON PLAZO FIJO, DURACIÓN Y COSTE

ALBERTO VERDEJO



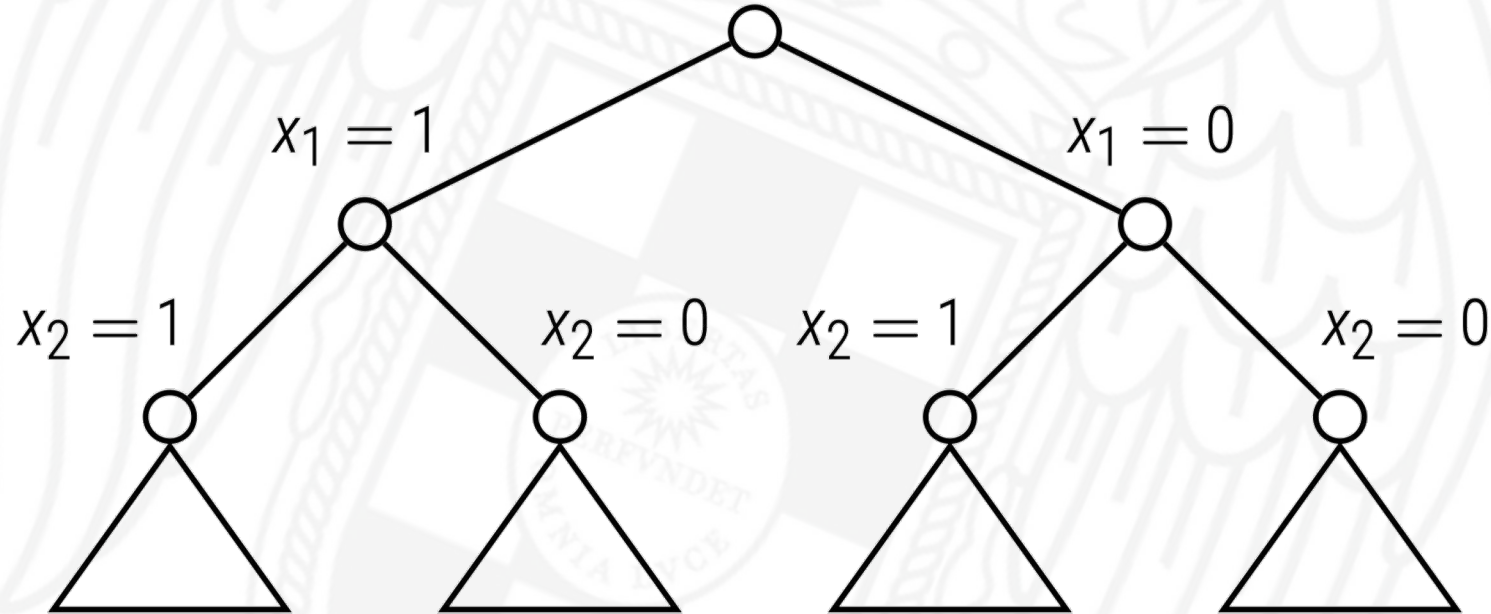
U N I V E R S I D A D
COMPLUTENSE
M A D R I D

Tareas con plazo fijo, duración y coste

- ▶ Tenemos n tareas y un procesador.
- ▶ Cada tarea i tiene asociada una terna (t_i, p_i, c_i) que indica que la tarea i tarda un tiempo t_i en ejecutarse, y que si su ejecución no está completa antes del plazo p_i , deberá pagarse una multa c_i por ella.
- ▶ El objetivo es seleccionar un subconjunto S de las n tareas que puedan realizarse antes de su plazo y que la multa impuesta por las tareas no realizadas sea mínima.

Representación de las soluciones y árbol de exploración

- ▶ Las soluciones son subconjuntos de tareas.
- ▶ Cada subconjunto se representa mediante su función característica, mediante una tupla (x_1, \dots, x_n) , donde $x_i = 1$ indica que la tarea i -ésima se realiza, mientras que $x_i = 0$ indica que no.



Test de factibilidad (o función de poda)

- ▶ Un subconjunto S de tareas es factible si y solo si la secuencia que ordena las tareas en S de forma no decreciente según plazo es admisible.
- ▶ Supongamos que $p_1 \leq p_2 \leq \dots \leq p_k$ y que existe una tarea r , con $1 \leq r \leq k$, tal que $p_r < \sum_{i=1}^r t_i = T_r$
- ▶ Pero entonces $\forall i : 1 \leq i \leq r : p_i < T_r$
- ▶ S no es factible.

Estimación

► Necesitamos una cota inferior:

1. Coste actual, $\sum_{i=1}^k (1 - x_i)c_i$.
2. Coste actual más el coste de las tareas pendientes (desde $k + 1$ hasta n) que ya no pueden hacerse antes de su plazo.

Implementación

```
struct Tarea {  
    double t, p, c;  
};  
  
struct Nodo {  
    vector<bool> sol;  
    int k;  
    double tiempo; // tiempo ocupado  
    double coste; // coste acumulado  
    double coste_estimado; // estimación, prioridad  
    bool operator<(Nodo const& otro) const {  
        return otro.coste_estimado < coste_estimado;  
    }  
};
```

Implementación

```
double calculo_estimacion(vector<Tarea> const& tareas, Nodo const& X) {  
    double estimacion = X.coste;  
    for (int i = X.k + 1; i < tareas.size(); ++i) {  
        if (X.tiempo + tareas[i].t > tareas[i].p) {  
            estimacion += tareas[i].c;  
        }  
    }  
    return estimacion;  
}
```

Implementación

```
// tareas ordenadas de menor a mayor plazo
void tareas_rp(vector<Tarea> const& tareas,
               vector<bool> & sol_mejor, double & coste_mejor) {
    int N = tareas.size();
    // se genera la raíz
    Nodo Y;
    Y.sol = vector<bool>(N, false);
    Y.k = -1;
    Y.tiempo = Y.coste = 0;
    Y.coste_estimado = calculo_estimacion(tareas, Y);
    priority_queue<Nodo> cola;
    cola.push(Y);
    coste_mejor = numeric_limits<double>::infinity();
```


Implementación

```
// búsqueda mientras pueda haber algo mejor
while (!cola.empty() && cola.top().coste_estimado < coste_mejor) {
    Y = cola.top(); cola.pop();
    Nodo X(Y);
    ++X.k;
    // hijo izquierdo, realizar la tarea
    if (Y.tiempo + tareas[X.k].t <= tareas[X.k].p) {
        X.sol[X.k] = true;
        X.tiempo = Y.tiempo + tareas[X.k].t; X.coste = Y.coste;
        X.coste_estimado = calculo_estimacion(tareas, X);
        if (X.coste_estimado < coste_mejor)
            if (X.k == N-1) {
                sol_mejor = X.sol; coste_mejor = X.coste;
            } else cola.push(X);
    }
}
```

Implementación

```
// hijo derecho, no realizar la tarea
X.sol[X.k] = false;
X.tiempo = Y.tiempo; X.coste = Y.coste + tareas[X.k].c;
X.coste_estimado = calculo_estimacion(tareas, X);
if (X.coste_estimado < coste_mejor) {
    if (X.k == N-1) {
        sol_mejor = X.sol; coste_mejor = X.coste;
    } else
        cola.push(X);
    }
}
```