FUERZA BRUTA, BACKTRACKING Y PROGRAMACIÓN DINÁMICA

Tecnología Digital V: Diseño de Algoritmos

Universidad Torcuato Di Tella

- **Problema:** Dadas $M_1, \ldots, M_n$, calcular

$$M = M_1 \times M_2 \times \ldots M_n$$

realizando la menor cantidad de multiplicaciones entre números de punto flotante.

- Por ejemplo, si $A \in \mathbb{R}^{13\times 5}$, $B \in \mathbb{R}^{5\times 89}$, $C \in \mathbb{R}^{89\times 3}$ y $D \in \mathbb{R}^{3\times 34}$, tenemos que

  1. $((AB)C)D$ requiere 10582 multiplicaciones,
  2. $(AB)(CD)$ requiere 54201 multiplicaciones,
  3. $(A(BC))D$ requiere 2856 multiplicaciones,
  4. $A((BC)D)$ requiere 4055 multiplicaciones,
  5. $A(B(CD))$ requiere 26418 multiplicaciones.

- Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i por un lado y las matrices $i+1$ a n por otro lado y luego multiplicar estos dos resultados, para algún $1 \leq i \leq n-1$, que es justamente lo que queremos determinar.
- Estos dos subproblemas, $M_1 \times M_2 \times \ldots M_i$ y $M_{i+1} \times M_{i+2} \times \ldots M_n$ deben estar resueltos, a su vez, de forma óptima, es decir realizando la mínima cantidad de operaciones.

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas, para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \ldots, n-1$, $i = 1, 2, \ldots, n-s$,

$$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

La solución del problema es $m[1][n]$.

- **Problema:** Dado un conjunto de n trabajos $J = \{1, 2, \ldots, n\}$, cada uno con
  - una **fecha límite** $d_j \in \mathbb{N}$,
  - una **ganancia** $p_j \in \mathbb{R}_+$,

  programar los trabajos en ranuras unitarias de tiempo, de modo que:
  - como máximo un trabajo se ejecute en cada unidad de tiempo,
  - el trabajo j debe completarse antes de d_j para obtener ganancia p_j.

- **Objetivo:** maximizar la suma de las ganancias obtenidas.

Idea: Probar todas las permutaciones y acumular la ganancia.

Fuerza Bruta:

$\textsc{JobScheduling}(S \in \{1, 2, \ldots, n\}^k,\ k \in \mathbb{N})$

if $k = n$ **then**
    if beneficio(S) $>$ beneficio(B) **then**
        $B \leftarrow S$
    end if
else
    for all $j \in \{1, \ldots, n\} \setminus S$ **do**
        $S' \leftarrow S + [j]$
        $\textsc{JobScheduling}(S',\ k + 1)$
    end for
end if

Job Scheduling – Backtracking

Idea: Podar ramas usando una cota superior de la ganancia.

$\textsc{JobScheduling}(S \in \{1, 2, \ldots, n\}^k,\ k \in \mathbb{N})$

---

if $k = n$ **then**
    if beneficio(S) $>$ beneficio(B) **then**
        $B \leftarrow S$
    end if
else
    if beneficio(S) $+ \sum_{i: i \notin S,\ d_i > |S|} p_i >$ beneficio(B) **then**
        for all $j \in \{1, \ldots, n\} \setminus S$ **do**
            $S' \leftarrow S + [j]$
            $\textsc{JobScheduling}(S',\ k+1)$
        end for
    end if
end if

Job Scheduling – Observación clave

- Siempre existe una solución óptima en la que los trabajos programados antes de su fecha límite están ordenados por **fecha límite creciente**, y se realizan antes de los trabjos entregados después de su límite.
- Esto permite dividir los trabajos en dos conjuntos:
  - **On-time:** trabajos programados antes de su fecha límite, generando ganancia.
  - **Late:** trabajos que no alcanzan su fecha límite, sin ganancia.
- Por lo tanto, podemos considerar recursivamente cada trabajo k en este orden, y decidir:
  - 1: programar k a tiempo,
  - 0: no programarlo a tiempo.

Idea: Probar todos los subconjuntos (ordenados) de jobs.

Fuerza Bruta:

$\textsc{JobScheduling}(S \in \{0,1\}^k,\ k \in \mathbb{N})$

```
if k = n then
    if esFactible(S) ∧ beneficio(S) > beneficio(B) then
        B ← S
    end if
else
    S′ ← S + [0]
    JobScheduling(S′, k + 1)
    S′ ← S + [1]
    JobScheduling(S′, k + 1)
end if
```

Idea: Podas por factibilidad y optimalidad.

$\textsc{JobScheduling}(S \in \{1, 2, \ldots, n\}^k,\ k \in \mathbb{N})$

---

if $k = n$ **then**
  if esFactible(S) $\wedge$ beneficio(S) $>$ beneficio(B) **then**
    $B \leftarrow S$
  end if
else
  if esFactible(S) $\wedge$ beneficio(S) $+ \sum_{\substack{j \in \{k+1, \ldots, n\} \\ d_j > |S|}} p_j >$ beneficio(B) **then**
    $S' \leftarrow S + [0]$
    $\textsc{JobScheduling}(S',\ k+1)$
    $S' \leftarrow S + [1]$
    $\textsc{JobScheduling}(S',\ k+1)$
  end if
end if

Definición de estados:

- Sea $z^*(k,t)$ = máxima ganancia considerando que se programaron t trabajos previamente entre los primeros k trabajos (ordenados por fecha límite).

Recurrencia: $k = 1, \ldots, n-1,\ t = 0, \ldots, n$

$$z^*(k,t) \quad = \quad \begin{cases} \text{máx}\Big(f(k+1,t), p_k + z^*(k+1,t+1)\Big) & \text{si } d_k > t \\ z^*(k+1,t) & \text{si } d_k \leq t \end{cases}$$

Caso base: $k = n, t = 0, 1, \ldots, n$:

$$z^*(n,t) \quad = \quad \begin{cases} p_n & \text{si } d_n > t \\ 0 & \text{si } d_n \leq t \end{cases}$$

TSP – Definición del Problema

Problema: Dado un conjunto de n ciudades y una matriz de distancias $d[i][j]$, encontrar el ciclo hamiltoniano de costo mínimo.

Formulación:

- Cada ciudad debe visitarse exactamente una vez.
- El ciclo debe regresar a la ciudad de origen.
- Objetivo: minimizar $\sum_{k=1}^{n} d[c_k, c_{k+1}]$ donde $c_{n+1} = c_1$.

Idea: Probar todas las permutaciones de ciudades y acumular la distancia total.

```
TSP(S,k):
    if k == n:
        if distancia(S+[$n_0$]) < distancia(B):
            B = S
    else:
            for city in cities:
                if city not in S:
                    S' = S + [city]
                    TSP(S',k+1)
            return best
```

Podemos elegir cualquier nodo n_0 como nodo inicial. Comenzamos con TSP([],0),

Idea: Poda por optimalidad.

```
TSP(S,k):
    if k == n:
        if distancia(S+[$n_0$]) < distancia(B):
            B = S
    else:
            if distancia(S) < distancia(B):
                for city in cities:
                    if city not in S:
                        S' = S + [city]
                        TSP(S',k+1)
            return best
```

Podemos elegir cualquier nodo n_0 como nodo inicial. Comenzamos con TSP([],0),

Podemos asumir que empezamos desde el nodo 0, visitamos las ciudades 1,2,...,n en cualquier orden, y retornamos al nodo 0.

Definición de estados:

- Estado = (i, S), $S \subset \{1, 2, \ldots, n\}$ es el conjunto de nodos a percorrer y i es el nodo inicial.
- $z^*(i, S)$ = distancia mínima para percorrer todos los nodos en S, empezando en ciudad i y terminando en ciudad n_0.

Recursión:

$$
\begin{aligned}
z^*(i, S) &= \text{mín}_{j \in S} \left(d[i][j] + z^*(j, S \setminus \{j\}) \right) \\
z^*(i, \{\}) &= d[i][0]
\end{aligned}
$$

$z^*(0, \{1, \ldots, n\})$ **es la distancia óptima.**

Cuántos estados hay en total?