Pseudocódigo: Programación de Trabajos con Fuerza Bruta

Algoritmo de Fuerza Bruta

25 de septiembre de 2025

1. Descripción del Problema

El problema de programación de trabajos con deadline (Job Scheduling with Deadlines) consiste en seleccionar y programar un subconjunto de trabajos de manera que se maximice la ganancia total, respetando las restricciones de deadline. Cada trabajo tiene:

- Un identificador único
- Un deadline (tiempo límite para completar el trabajo)
- Una ganancia asociada

El objetivo es encontrar la secuencia de trabajos que maximice la ganancia total, donde cada trabajo debe completarse antes de su deadline y cada trabajo toma exactamente una unidad de tiempo.

2. Pseudocódigo

Algorithm 1 Programación de Trabajos con Fuerza Bruta

Require: Una lista *jobs* de n elementos donde cada elemento es una tupla $(id, deadline, profit)$
Ensure: La mejor programación y la ganancia máxima

```
max_profit ← 0
best_schedule ← lista vacía
                                        ▷ Probar todas las permutaciones posibles
for cada perm en permutations(jobs) do
    time ← 0                            ▷ Tiempo actual
    profit ← 0                          ▷ Ganancia acumulada
    schedule ← lista vacía              ▷ Programación actual
    for cada job en perm do
        if time < job.deadline then
            schedule.append(job)
            profit ← profit + job.profit
            time ← time + 1
        end if
    end for
    if profit > max_profit then
        max_profit ← profit
        best_schedule ← schedule
    end if
end forreturn best_schedule, max_profit
```

3. Explicación del Algoritmo

3.1. Enfoque de Fuerza Bruta

El algoritmo de fuerza bruta para programación de trabajos funciona de la siguiente manera:

- **Generación de permutaciones:** Se generan todas las posibles permutaciones del conjunto de trabajos
- **Evaluación de cada permutación:** Para cada permutación, se simula la ejecución de trabajos en orden
- **Verificación de deadlines:** Solo se incluyen trabajos que pueden completarse antes de su deadline
- **Optimización:** Se mantiene la mejor programación encontrada hasta el momento

La complejidad del algoritmo viene del hecho de que se prueban todas las $n!$ permutaciones posibles de los trabajos.

3.2. Restricciones del Problema

Para cada permutación de trabajos, se aplican las siguientes restricciones:

- **Deadline:** Un trabajo solo puede ser incluido si $tiempo\_actual < deadline$
- **Tiempo unitario:** Cada trabajo toma exactamente 1 unidad de tiempo
- **Secuencialidad:** Los trabajos se ejecutan en el orden de la permutación
- **Objetivo:** Maximizar la suma de ganancias de los trabajos incluidos

Matemáticamente, para una permutación $P = (j_1, j_2, ..., j_n)$:

$$\max \sum_{i=1}^{k} profit(j_i)$$

donde k es el número de trabajos que pueden completarse antes de sus deadlines.

3.3. Complejidad

- **Tiempo:** $O(n! \cdot n)$
  - $O(n!)$ para generar todas las permutaciones
  - $O(n)$ para evaluar cada permutación
- **Espacio:** $O(n)$ para almacenar la mejor programación

Esta complejidad factorial hace que el algoritmo sea impráctico para problemas grandes, pero garantiza encontrar la solución óptima.

4. Resolución Paso a Paso

4.1. Ejemplo: Trabajos = [(“a”,2,100), (“b”,1,19), (“c”,2,27), (“d”,1,25), (“e”,3,15)]

Trabajos de entrada:

ID	Deadline	Ganancia
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Paso 1: Identificar parámetros del problema

- Número de trabajos: $n = 5$
- Total de permutaciones posibles: $5! = 120$
- Objetivo: Encontrar la permutación que maximice la ganancia

Paso 2: Ejemplos de permutaciones y sus evaluaciones
Permutación 1: (a, b, c, d, e)

- Tiempo 0: Trabajo a (deadline=2) $\rightarrow$ Incluir, ganancia=100, tiempo=1
- Tiempo 1: Trabajo b (deadline=1) $\rightarrow$ $1 \not< 1$, NO incluir
- Tiempo 1: Trabajo c (deadline=2) $\rightarrow$ $1 < 2$, Incluir, ganancia=127, tiempo=2
- Tiempo 2: Trabajo d (deadline=1) $\rightarrow$ $2 \not< 1$, NO incluir
- Tiempo 2: Trabajo e (deadline=3) $\rightarrow$ $2 < 3$, Incluir, ganancia=142, tiempo=3
- **Resultado:** Ganancia = 142, Programación = [a, c, e]

Permutación 2: (b, a, c, d, e)

- Tiempo 0: Trabajo b (deadline=1) $\rightarrow$ $0 < 1$, Incluir, ganancia=19, tiempo=1
- Tiempo 1: Trabajo a (deadline=2) $\rightarrow$ $1 < 2$, Incluir, ganancia=119, tiempo=2
- Tiempo 2: Trabajo c (deadline=2) $\rightarrow$ $2 \not< 2$, NO incluir
- Tiempo 2: Trabajo d (deadline=1) $\rightarrow$ $2 \not< 1$, NO incluir
- Tiempo 2: Trabajo e (deadline=3) $\rightarrow$ $2 < 3$, Incluir, ganancia=134, tiempo=3
- **Resultado:** Ganancia = 134, Programación = [b, a, e]

Permutación 3: (d, a, c, b, e)

- Tiempo 0: Trabajo d (deadline=1) $\rightarrow$ $0 < 1$, Incluir, ganancia=25, tiempo=1
- Tiempo 1: Trabajo a (deadline=2) $\rightarrow$ $1 < 2$, Incluir, ganancia=125, tiempo=2
- Tiempo 2: Trabajo c (deadline=2) $\rightarrow$ $2 \not< 2$, NO incluir
- Tiempo 2: Trabajo b (deadline=1) $\rightarrow$ $2 \not< 1$, NO incluir
- Tiempo 2: Trabajo e (deadline=3) $\rightarrow$ $2 < 3$, Incluir, ganancia=140, tiempo=3
- **Resultado:** Ganancia = 140, Programación = [d, a, e]

Paso 3: Continuar evaluando permutaciones hasta encontrar la óptima
Después de evaluar múltiples permutaciones, se encuentra:
Permutación óptima: (a, c, d, b, e)

- Tiempo 0: Trabajo a (deadline=2) $\rightarrow$ $0 < 2$, Incluir, ganancia=100, tiempo=1

- Tiempo 1: Trabajo c (deadline=2) $\rightarrow 1 < 2$, Incluir, ganancia=127, tiempo=2
- Tiempo 2: Trabajo d (deadline=1) $\rightarrow 2 \not< 1$, NO incluir
- Tiempo 2: Trabajo b (deadline=1) $\rightarrow 2 \not< 1$, NO incluir
- Tiempo 2: Trabajo e (deadline=3) $\rightarrow 2 < 3$, Incluir, ganancia=142, tiempo=3
- **Resultado:** Ganancia = 142, Programación = [a, c, e]

Paso 4: Verificar si existe una mejor solución
Continuando la búsqueda exhaustiva:
Permutación mejorada: (a, c, e, b, d)

- Tiempo 0: Trabajo a (deadline=2) $\rightarrow 0 < 2$, Incluir, ganancia=100, tiempo=1
- Tiempo 1: Trabajo c (deadline=2) $\rightarrow 1 < 2$, Incluir, ganancia=127, tiempo=2
- Tiempo 2: Trabajo e (deadline=3) $\rightarrow 2 < 3$, Incluir, ganancia=142, tiempo=3
- Tiempo 3: Trabajo b (deadline=1) $\rightarrow 3 \not< 1$, NO incluir
- Tiempo 3: Trabajo d (deadline=1) $\rightarrow 3 \not< 1$, NO incluir
- **Resultado:** Ganancia = 142, Programación = [a, c, e]

Resultado Final: La ganancia máxima es 142, obtenida con la programación [a, c, e].

4.2. Visualización de la Solución Óptima

Tiempo	**Trabajo**	**Deadline**	**Ganancia**
0	a	2	100
1	c	2	27
2	e	3	15
Total			**142**

Justificación:

- El trabajo 'a' se ejecuta en tiempo 0 (antes de deadline 2)
- El trabajo 'c' se ejecuta en tiempo 1 (antes de deadline 2)
- El trabajo 'e' se ejecuta en tiempo 2 (antes de deadline 3)
- Los trabajos 'b' y 'd' no pueden incluirse porque sus deadlines (1) ya han pasado

5. Análisis de Complejidad

5.1. Comparación con Algoritmos Greedy

- **Fuerza Bruta:** $O(n! \cdot n)$ - Garantiza solución óptima
- **Algoritmo Greedy por Ganancia:** $O(n \log n)$ - Puede no ser óptimo
- **Algoritmo Greedy por Deadline:** $O(n \log n)$ - Puede no ser óptimo

Ventajas del enfoque de fuerza bruta:

- Garantiza encontrar la solución óptima
- Útil para verificar la corrección de algoritmos heurísticos

- Apropiado para problemas pequeños

Desventajas:

- Complejidad factorial hace que sea impráctico para $n > 10$
- No aprovecha la estructura del problema
- Consume mucho tiempo computacional

6. Conclusiones

El algoritmo de fuerza bruta para programación de trabajos garantiza encontrar la solución óptima pero tiene una complejidad factorial que lo hace impráctico para problemas grandes. Es útil para:

- Verificar la corrección de algoritmos más eficientes
- Resolver instancias pequeñas del problema
- Estudios teóricos sobre la optimalidad de soluciones

Para aplicaciones prácticas, se recomienda usar algoritmos greedy o de programación dinámica que, aunque pueden no garantizar optimalidad, ofrecen soluciones razonables en tiempo polinomial.