Exposición: Problemas de Análisis y Diseño de Algoritmos

Brute Force vs Greedy

Delgado Romero, Gustavo
Torres Reategui, Joaquin
Quintanilla Quispe, Dylan

Universidad Nacional de Ingeniería
CC371 - Análisis y Diseño de Algoritmos

25 de septiembre de 2025

Tabla de Contenidos

Resumen de Problemas

Problemas a Desarrollar

1. **Activity Selection Problem** - Selección de actividades
2. **Graph Coloring** - Coloreo de grafos
3. **Job Scheduling with Deadlines** - Programación de trabajos
4. **Huffman Codes** - Códigos de Huffman

Enfoques Comparados

- **Fuerza Bruta**: Solución óptima garantizada, complejidad exponencial
- **Greedy**: Solución eficiente, no siempre óptima, complejidad polinomial

Tabla de Contenidos

Activity Selection Problem - Enunciado

Descripción del Problema

Seleccionar el máximo número de actividades mutuamente compatibles de un conjunto dado.
Cada actividad tiene:

- Tiempo de inicio y finalización
- Compatibilidad: No deben solaparse en el tiempo
- Objetivo: Maximizar el número de actividades seleccionadas

Ejemplo

Actividades: [(1,2), (3,4), (0,6), (5,7), (8,9), (5,9)]
Solución óptima: 4 actividades

Activity Selection - Fuerza Bruta

Estrategia

- Generar todos los subconjuntos posibles de actividades
- Verificar compatibilidad para cada subconjunto
- Seleccionar el subconjunto válido de mayor tamaño

Complejidad

$O(2^n \cdot n^2)$ - Exponencial

Activity Selection - Pseudocódigo Fuerza Bruta

```
[Entrada:]Un arreglo activities de n elementos donde cada elemento es una tupla (start, end)
[Salida:]El subconjunto de máximo tamaño de actividades mutuamente compatibles
n ← longitud de activities                                        ▷ Cantidad de actividades
best ← lista vacía        ▷ Mejor subconjunto encontrado        ▷ Probar todos los subconjuntos posibles
for r = 1 to n do
    for cada subset en combinations(activities, r) do
        if is_valid(subset) ∧ |subset| > |best| then                  ▷ Compatibilidad y tamaño
            best ← subset                                          ▷ Actualizar mejor solución
        end if
    end for
end forreturn best
```

Activity Selection - Ejemplo Fuerza Bruta I

Actividades: [(1,3), (2,5), (4,6)]

Paso 1: Identificar actividades A=(1,3), B=(2,5), C=(4,6)
Paso 2: Subconjuntos de tamaño 1

- {A}, {B}, {C}: Todos válidos

Mejor hasta ahora: tamaño 1
Paso 3: Subconjuntos de tamaño 2

- {A,B}: $3 > 2 \rightarrow$ Se solapan $\rightarrow$ Inválido
- {A,C}: $3 \leq 4 \rightarrow$ No se solapan $\rightarrow$ Válido
- {B,C}: $5 > 4 \rightarrow$ Se solapan $\rightarrow$ Inválido

Mejor hasta ahora: {A,C}
Paso 4: Subconjunto de tamaño 3

- {A,B,C}: Hay solapamientos $\rightarrow$ Inválido

Activity Selection - Ejemplo Fuerza Bruta II

Solución óptima

{A,C} con 2 actividades

Activity Selection - Enfoque Greedy

Estrategia Óptima

- Ordenar actividades por tiempo de finalización
- Seleccionar siempre la siguiente actividad que termina primero y no se solape

Complejidad

$O(n \log n)$ - Polinomial (óptimo)

Activity Selection - Pseudocódigo Greedy

[Entrada:] Arreglos *start* y *finish* de n elementos con tiempos de inicio y fin
[Salida:] El número máximo de actividades mutuamente compatibles

```
activities ← lista de tuplas (start[i], finish[i]) para i = 0..n − 1     ▷ Crear pares inicio-fin
Ordenar activities por tiempo de finalización ascendente                 ▷ Clave para optimalidad
count ← 1                                                                 ▷ Al menos una actividad puede realizarse
j ← 0                                                                     ▷ Índice de la última actividad seleccionada
for i = 1 to n − 1 do
    if activities[i].start > activities[j].finish then                   ▷ Verifica no solapamiento
        count ← count + 1                                                 ▷ Selecciona actividad
        j ← i                                                             ▷ Actualiza última seleccionada
    end if
end forreturn count
```

Activity Selection - Ejemplo Greedy I

Actividades = [(1, 2), (3, 4), (0, 6), (5, 7), (8, 9), (5, 9)]

Paso 1: Crear tuplas y ordenar por fin: (1,2), (3,4), (0,6), (5,7), (8,9), (5,9)

Iteraciones

- i=1: (3,4). ¿3 > 2? Sí → Seleccionar, count=2, j=1
- i=2: (0,6). ¿0 > 4? No → Omitir
- i=3: (5,7). ¿5 > 4? Sí → Seleccionar, count=3, j=3
- i=4: (8,9). ¿8 > 7? Sí → Seleccionar, count=4, j=4
- i=5: (5,9). ¿5 > 9? No → Omitir

Resultado Final

4 actividades: (1,2), (3,4), (5,7), (8,9)

Activity Selection - Análisis Comparativo

Enfoque	**Complejidad**	**Optimalidad**
Fuerza Bruta	$O(2^n \cdot n^2)$	✓
Greedy	$O(n \log n)$	✓

Activity Selection - Análisis Comparativo

Algoritmo	Actividades	Tiempo (segundos)
Greedy	4	0.000016
Greedy	5	0.000012
Greedy	6	0.000008
Greedy	6	0.000009
Brute Force	4	0.000065
Brute Force	5	0.001175
Brute Force	10	0.156779
Brute Force	12	76.587371

Cuadro: Tiempos de ejecución comparativos

Conclusión

- Greedy es óptimo y eficiente para este problema
- Fuerza bruta solo práctico para instancias pequeñas ($n < 20$)
- El ordenamiento por tiempo de finalización garantiza optimalidad

Tabla de Contenidos

Graph Coloring - Enunciado

Descripción del Problema

Asignar colores a vértices de un grafo tal que:

- Vértices adyacentes tengan colores diferentes
- Minimizar el número de colores usado (número cromático)
- Problema NP-completo en general

Aplicaciones

- Scheduling de procesos
- Asignación de registros
- Sudoku

Estrategia

- Probar todas las asignaciones de k colores a n vértices
- Verificar validez para cada asignación
- Encontrar el mínimo k que permita coloreo válido

Complejidad

$O(k^n \cdot n \cdot m)$ - Exponencial

Graph Coloring - Pseudocódigo Fuerza Bruta

```
[Entrada:]Grafo G (lista de adyacencia), número máximo de colores k
[Salida:]Asignación válida de colores o null
for cada assignment en product(range(k), repeat = n) do        ▷ Todas las asignaciones de k colores
    if is_valid(G, assignment) then        ▷ Verifica restricción de coloreo return assignment
    end if
end forreturn null
```

```
function IS_VALID(G, colors)
    for u = 0 to |G| − 1 do        ▷ Recorrer todos los vértices
        for cada v en G[u] do        ▷ Revisar vecinos
            if colors[u] = colors[v] then return false        ▷ Vértices adyacentes con mismo color
            end if
        end for
    end forreturn true
end function
```

Graph Coloring - Ejemplo Fuerza Bruta

Grafo con 3 vértices: Triángulo

- **k=2**: Probar $2^3 = 8$ asignaciones, ninguna válida
- **k=3**: Probar $3^3 = 27$ asignaciones
- Encontrar solución: [0,1,2] - 3 colores

Limitaciones

- 10 vértices con 3 colores: $3^{10} = 59,049$ pruebas
- 20 vértices: $3^{20} > 3.4$ billones de pruebas

Graph Coloring - Enfoque Greedy

Estrategia

- Procesar vértices en orden fijo
- Asignar el menor color disponible no usado por vecinos
- No garantiza número cromático mínimo

Complejidad

$O(V^2 + E)$ - Polinomial

Graph Coloring - Pseudocódigo Greedy

```
[Entrada:]Grafo G (lista de adyacencia) con V vértices
[Salida:]Asignación válida de colores
result ← arreglo de tamaño V inicializado en −1                  ▷ Colores por vértice
result[0] ← 0                                                   ▷ Color al primer vértice
available ← arreglo booleano de tamaño V en false               ▷ Colores ocupados por vecinos
for u = 1 to V − 1 do                        ▷ Marcar colores de vecinos como no disponibles
    for i = 0 to |adj[u]| − 1 do
        if result[adj[u][i]] ≠ −1 then
            available[result[adj[u][i]]] ← true
        end if
    end for                                           ▷ Encontrar el primer color disponible
    cr ← 0
    while cr < V do
        if available[cr] = false then
            break
        end if
        cr ← cr + 1
    end while
    result[u] ← cr        ▷ Asignar color encontrado    ▷ Resetear disponibilidad para el siguiente vértice
```

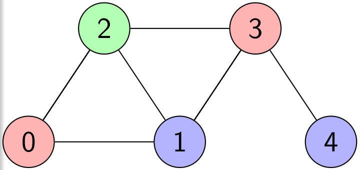
Graph Coloring - Pseudocódigo Greedy

$result[u] \leftarrow cr$ ▷ Asignar color encontrado ▷ Resetear disponibilidad para el siguiente vértice
for $i = 0$ to $|adj[u]| - 1$ **do**
  if $result[adj[u][i]] \neq -1$ **then**
    $available[result[adj[u][i]]] \leftarrow$ false
  end if
end for
end forreturn *result*

Graph Coloring - Ejemplo Greedy

Grafo de 5 vértices

- V0: Color 0
- V1: Color 1 (vecino de V0)
- V2: Color 2 (vecino de V0,1)
- V3: Color 0 (vecino de V1,2)
- V4: Color 1 (vecino de V3)



Resultado

3 colores usados (óptimo para este grafo)

Graph Coloring - Análisis Comparativo

Enfoque	**Complejidad**	**Optimalidad**
Fuerza Bruta	$O(k^n \cdot n \cdot m)$	✓
Greedy	$O(V^2 + E)$	$\times$

Graph Coloring - Análisis Comparativo

Grafo	Algoritmo	Colores	Tiempo (segundos)
Graph 1	Greedy	3	0.000007
Graph 1	Brute Force	3	0.000022
Graph 2	Greedy	4	0.000006
Graph 2	Brute Force	3	0.000025

Cuadro: Tiempos de ejecución comparativos

Conclusión

- Fuerza bruta garantiza optimalidad pero es impráctica
- Greedy es eficiente pero puede usar más colores de los necesarios
- En la práctica: Greedy para grafos grandes, heurísticas avanzadas para mejor aproximación

Tabla de Contenidos

Job Scheduling - Enunciado

Descripción del Problema

Programar trabajos con:

- Deadline: Tiempo límite para completar
- Profit: Ganancia por completar a tiempo
- Cada trabajo toma 1 unidad de tiempo
- Objetivo: Maximizar ganancia total

Aplicaciones

- Scheduling de procesos en CPU
- Planificación de producción
- Asignación de recursos limitados

Job Scheduling - Fuerza Bruta

Estrategia

- Generar todas las permutaciones de trabajos
- Para cada permutación, programar trabajos en orden
- Solo incluir trabajos que cumplan deadline
- Seleccionar permutación con máxima ganancia

Complejidad

$O(n! \cdot n)$ - Factorial

Job Scheduling - Pseudocódigo Fuerza Bruta

```
[Entrada:]Una lista jobs de n elementos donde cada elemento es (id, deadline, profit)
[Salida:]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                      ▷ Verifica deadline
            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
```

Job Scheduling - Ejemplo Fuerza Bruta

Trabajos: [(a,2,100), (b,1,19), (c,2,27), (d,1,25), (e,3,15)]

- **Permutación 1**: [a,b,c,d,e] $\rightarrow$ Ganancia: 142
- **Permutación 2**: [b,a,c,d,e] $\rightarrow$ Ganancia: 134
- **Permutación 3**: [d,a,c,b,e] $\rightarrow$ Ganancia: 140
- **Mejor**: [a,c,e] con ganancia 142

Limitaciones

5 trabajos = 120 permutaciones, 10 trabajos = 3.6 millones

Job Scheduling - Enfoque Greedy

Estrategia

- Ordenar trabajos por profit descendente
- Para cada trabajo, asignar al slot más tardío disponible dentro de su deadline

Complejidad

$O(n \log n + n \cdot d)$ - Polinomial

Job Scheduling - Pseudocódigo Greedy

[Entrada:]Lista *jobs* donde cada trabajo tiene (*id*, *deadline*, *profit*)
[Salida:]Lista de trabajos seleccionados y ganancia total
Ordenar *jobs* por ganancia en orden descendente ▷ Prioriza ganancias altas
$max\_deadline \leftarrow \text{máx}\{job.deadline\ \forall job \in jobs\}$
$slots \leftarrow$ arreglo de tamaño $max\_deadline + 1$ inicializado en -1 ▷ Slots de tiempo
$total\_profit \leftarrow 0$, $scheduled\_jobs \leftarrow$ lista vacía
for cada *job* en *jobs* **do**
    for $t \leftarrow job.deadline;\ t \geq 1;\ t \leftarrow t - 1$ **do** ▷ Buscar slot más tardío disponible
        if $slots[t] = -1$ **then**
            $slots[t] \leftarrow job.id$
            $total\_profit \leftarrow total\_profit + job.profit$
            $scheduled\_jobs.append(job.id)$
            break
        end if
    end for
end forreturn $scheduled\_jobs, total\_profit$

Job Scheduling - Ejemplo Greedy I

Trabajos = [A(2,100), B(1,19), C(2,27), D(1,25), E(3,15)]

Orden por ganancia: A(100), C(27), D(25), B(19), E(15)

Asignación

- A (d=2): buscar t=2..1 → t=2 libre → slots[2]=A, total=100
- C (d=2): t=2 ocupado, t=1 libre → slots[1]=C, total=127
- D (d=1): t=1 ocupado → no asignar
- B (d=1): t=1 ocupado → no asignar
- E (d=3): t=3 libre → slots[3]=E, total=142

Resultado Final

Programación: [C, A, E] con ganancia total 142

Job Scheduling - Análisis Comparativo

Enfoque	Complejidad	Optimalidad
Fuerza Bruta	$O(n! \cdot n)$	✓
Greedy	$O(n \log n + n \cdot d)$	×

Job Scheduling - Análisis Comparativo

Ejemplo	**Algoritmo**	**Ganancia**	**Tiempo (segundos)**
Ejemplo 1	Brute Force	142	0.000051
Ejemplo 1	Greedy	142	0.000017
Ejemplo 2	Brute Force	190	0.001998
Ejemplo 2	Greedy	190	0.000020

Cuadro: Tiempos de ejecución comparativos

Conclusión

- Greedy no siempre es óptimo pero es muy eficiente
- En la práctica funciona bien para la mayoría de casos
- Fuerza bruta solo para instancias muy pequeñas ($n < 10$)

Tabla de Contenidos

Huffman Codes - Enunciado

Descripción del Problema

Encontrar código de prefijo óptimo para compresión de datos:

- Símbolos con frecuencias dadas
- Código de longitud variable
- Objetivo: Minimizar longitud promedio
- Costo $= \sum f_i \cdot l_i$

Aplicaciones

- Compresión de archivos (ZIP, JPEG, MP3)
- Transmisión eficiente de datos

Huffman Codes - Fuerza Bruta

Estrategia

- Generar todos los árboles binarios completos con n hojas
- Para cada árbol, calcular longitudes de código
- Probar todas las asignaciones de símbolos a hojas
- Seleccionar asignación con mínimo costo

Complejidad

$O(C_{n-1} \cdot n! \cdot n)$ - Super-exponencial

Huffman Codes - Pseudocódigo Fuerza Bruta

```
[Entrada:]Símbolos y frecuencias
[Salida:]Asignación óptima de códigos
n ← número de símbolos
best_cost ← ∞, best_assignment ← null
for cada tree en generate_trees(n) do
    lengths ← code_lengths(tree)
    for cada perm en permutations(range(n)) do
        cost ← ∑ freqs[perm[i]] · lengths[i]
        if cost < best_cost then
            best_cost ← cost, best_assignment ← asignación
        end if
    end for
end forreturn best_cost, best_assignment
```

Huffman Codes - Ejemplo Fuerza Bruta

4 símbolos: [A:5, B:2, C:1, D:1]

- Número de árboles: $C_3 = 5$
- Permutaciones por árbol: $4! = 24$
- Total: $5 \times 24 = 120$ asignaciones
- **Mejor**: A→1, B→2, C→2, D→3 (Costo=14)

Limitaciones

6 símbolos: $C_5 = 42$ árboles $\times$ $6! = 720 \times 6 = 181{,}440$ operaciones

Huffman Codes - Enfoque Greedy

Estrategia Óptima

- Usar min-heap por frecuencias
- Combinar siempre los dos nodos de menor frecuencia
- Construir árbol de abajo hacia arriba

Complejidad

$O(n \log n)$ - Polinomial (óptimo)

Huffman Codes - Pseudocódigo Greedy

```
[Entrada:]Lista de símbolos s y frecuencias freq
[Salida:]Códigos Huffman óptimos
n ← longitud de s
pq ← cola de prioridad (min-heap) vacía     ▷ Menor frecuencia primero     ▷ Crear nodos hoja para cada símbolo
for i = 0 to n − 1 do
    tmp ← Node(freq[i])
    pq.push(tmp)
end for                                       ▷ Construir árbol de Huffman
while |pq| ≥ 2 do
    l ← pq.pop()                              ▷ Nodo con menor frecuencia
    r ← pq.pop()                              ▷ Siguiente menor
    new ← Node(l.freq + r.freq)
    new.left ← l, new.right ← r
    pq.push(new)
end while
root ← pq.pop()
Generar códigos con recorrido preorden desde root return lista de códigos por símbolo
```

Huffman Codes - Ejemplo Greedy

Símbolos: [a:5, b:9, c:12, d:13, e:16, f:45]

- Combinar a(5) + b(9) = 14
- Combinar c(12) + d(13) = 25
- Combinar 14 + e(16) = 30
- Combinar 25 + 30 = 55
- Combinar f(45) + 55 = 100

Resultado

Códigos óptimos: f→0, c→100, d→101, a→1100, b→1101, e→111

Huffman Codes - Análisis Comparativo

Enfoque	Complejidad	Optimalidad
Fuerza Bruta	$O(C_{n-1} \cdot n! \cdot n)$	✓
Greedy	$O(n \log n)$	✓

Conclusión

- Huffman greedy es óptimo y eficiente
- Raro caso donde greedy garantiza optimalidad
- Fuerza bruta completamente impráctica
- Algoritmo ampliamente usado en compresión

Huffman Codes - Análisis Comparativo

Ejemplo	Algoritmo	Costo	Tiempo (segundos)
Ejemplo 1	Brute Force	11	0.000039
Ejemplo 1	Greedy	11	0.000017
Ejemplo 2	Brute Force	15	0.000149
Ejemplo 2	Greedy	15	0.000019

Cuadro: Tiempos de ejecución comparativos

Conclusión

- Huffman greedy es óptimo y eficiente
- Raro caso donde greedy garantiza optimalidad
- Fuerza bruta completamente impráctica
- Algoritmo ampliamente usado en compresión

Tabla de Contenidos

Conclusiones Generales

Comparativa de Enfoques

Problema	**BF Optimal**	**Greedy Optimal**	**BF Complejidad**	**Greedy Complejidad**
Activity Selection	$\checkmark$	$\checkmark$	$O(2^n n^2)$	$O(n \log n)$
Graph Coloring	$\checkmark$	$\times$	$O(k^n nm)$	$O(V^2 + E)$
Job Scheduling	$\checkmark$	$\times$	$O(n!n)$	$O(n \log n + nd)$
Huffman Codes	$\checkmark$	$\checkmark$	$O(C_{n-1} n! n)$	$O(n \log n)$

Insights Clave

- Greedy es óptimo cuando el problema exhibe **subestructura óptima**
- La **elección greedy** debe preservar la posibilidad de solución óptima
- Fuerza bruta es invaluable para verificar correctitud de algoritmos

Recomendaciones Prácticas

Cuándo usar cada enfoque

- **Fuerza Bruta**:
  - Problemas pequeños ($n \leq 20$)
  - Verificación de algoritmos
  - Casos donde se necesita optimalidad garantizada
- **Greedy**:
  - Problemas grandes donde optimalidad no es crítica
  - Cuando el problema exhibe subestructura óptima
  - Aplicaciones en tiempo real que requieren rapidez

Estrategia Híbrida

En la práctica: Empezar con greedy, usar fuerza bruta para casos pequeños o como verificación

¿Preguntas?

Contacto

- Delgado Romero
- Torres Reategui
- Quintanilla Quispe

¡Gracias por su atención!