Algoritmos Greedy

Integrantes

- Christian Echeverría 221441
- Gustavo Cruz 22779
- Josué Say 22801
- Mathew Cordero 22982
- Pedro Guzmán 22111

Minimización del tiempo promedio de terminación en calendarización no-preemptiva

Veamos un nuevo problema de calendarizacion . Tenemos un conjunto de tareas S= {a1, …. an}, donde la letra a_i tarda en realizarse p_i de unidades de tiempo. Dependiendo de como calendaricemos su ejecucion, cada tarea tendra un tiempo de terminacion c_i (que es cuando la tarea se completa). Queremos hacer una calendarizacion non-preemptive (un tarea iniciada no se puede interrumpir) que minimice todos los tiempos de terminacion

a. Proponga un algoritmo greedy que provee esta calendarizacion
b. Determine el tiempo de ejecucion de su algoritmo
c. Demuestre que su algoritmo de la solucion optima (es decir que , demuestre presenta caracteristicas que permiten obtener una solucion optima mediante un algoritmo greedy o modele el problema como una matroide ponderada)

1. Proponga un algoritmo

Lo que queremos minimizar es todos los tiempos de terminacion de nuestro stack osea que minicemos el tiempo total de minimizacion entre ellos. En pocas palabras :

$$min(\sum_{i=1}^{n} c_i)$$

Asi mismo se sabe lo siguiente

- Una vez iniciado un proceso a_i no se puede pausar
- Se ordenan los procesos segun su p_i o tiempo de procesamiento.

- El tiempo de terminacion c_i de un proceso a_i es igual a $p_i + t$ osea es el procesamiento actual y t que es la sumatoria de todos los anteriores. $t =∑_{i=1}^i c_i $.

Ahora vamos a proponer el algoritmo, recordemos que estamos lidiando con una heuristica SPF o shortest processing time first debido a que vamos a realizar el procesamiento de los que tienen menos tiempo de procesamiento para minimizar el total de tiempos de terminacion.

Esto se puede observar si tenemos 2 procesos:

- A tarda 1 seg
- B tarda 10 seg

Si se ejecuta primero el tiempo de c de A sera 1, el c de B sera 1 + 10 osea 11. Y el total de terminacion de todos sera 1 + 11 = 12.

Pero si se ejecuta primero el de mas tiempo el tiempo de procesamiento de B sera 10 pero el de A sera de 11. Y el total sera de 10 + 11 osea 21

El pseudocodigo del algoritmo sera el siguiente

```
function merge_sort(tareas):
    if longitud(tareas) <= 1:
        return tareas

    medio ← longitud(tareas) // 2
    izquierda ← merge_sort(tareas[0:medio])
    derecha ← merge_sort(tareas[medio:])

    return merge(izquierda, derecha)

function merge(izq, der):
    resultado ← []
    mientras izq no vacío y der no vacío:
        si izq[0].p <= der[0].p:
            resultado.agregar(izq.eliminar_primero())
        sino:
            resultado.agregar(der.eliminar_primero())
    agregar el resto de izq y der a resultado
    return resultado

function calcular_suma_tiempos(tareas):
```

```
    tareas_ordenadas ← merge_sort(tareas)
    t ← 0
    suma_total ← 0

    para cada tarea en tareas_ordenadas:
        t ← t + tarea.p
        suma_total ← suma_total + t

    return suma_total
```

Como se puede ver lo que se hizo fue ordenar obtener la lista de tareas, ordenarlas por p_i de la menor a la mayor, y luego tener una variable t que es la suma de los c_i.

2. Tiempo de Ejecucion

El tiempo de ejecucion es el siguiente

De todos los tiempos de ejecucion realmente el de sumar es O(1) el de obtener la lista es de O(1) lo unico que afecta el rendimiento es que algoritmo de ordenamiento usemos, en este caso usamos merge sort entonces el analisis seria el siguiente:

Merge Sort

Sea $T(n)$ el tiempo que toma ordenar un arreglo de tamaño n:

Se divide el arreglo en dos partes de tamaño $n/2$.

Se hacen dos llamadas recursivas para ordenar cada mitad → $2T(n/2)$.

Luego se mezclan ambas mitades ordenadas en tiempo $O(n)$.

$$T(n) = 2T(n/2) + O(n)$$

Ahora resolvemos la recurrencia con la ecuacion de recurrencia

$$T(n) = aT(n/b) + f(n)$$

Una vez identificada la forma de la grafica sabemos que

- $a = 2$
- $b = 2$
- $f(n) = O(n)$

Resolviendo nos da $n^{log_b a} = n^{log_2 2}$ $f(n) = O(n^{lob_b a})$ estamos en el caso 2 por ende

$$T(n) = O(nlogn)$$

Por ende la complejidad o tiempo de ejecucion de nuestro algoritmo sera de :

$$O(nlogn)$$

3. Solucion Optima

Demostración (Intercambio por pares)

Sabemos que de la secuencia de tareas S, tenemos 2 tareas a_i y a_{i+1} entonces sus p_i seguiran el orden de $p_i > pi + 1$

Calculamos el impacto

Sea:

- t: el tiempo acumulado hasta antes de a_i
- c_i = $t + p_i$
- c_{i+1} = $t + p_i + p_{i+1}$

La suma parcial de tiempos

$$c_i + c_{i+1} = (t + p_i) + (t + p_i + p_{i+1}) = 2t + 2p_i + p_{i+1}$$

Después de intercambiar osea que primero se hagan los mas largos:

- $c' = t + p_{i+1}$
- $c'_{i+1} = t + p_{i+1}$

La sumatoria sera

$$c'_i + c'_{i+1} = (t + p_{i+1}) + (t + p_{i+1} + p_i) = 2t + 2p_{i+1} + p_i$$

Ahora vamos a comparar

$$(2t + 2p_i + p_{i+1}) - (2t + 2p_{i+1} + p_i)$$

Al operar nos da lo siguiente

$$2t - 2t + 2p_i - p_i + 2p_{i+1} - p_{i+1}$$

Lo que da

$$p_i + p_{i+1} > 0$$

Esto quieres decir que la suma antes del intercambio era mayor, por lo tanto el intercambio mejora la suma total de tiempos de finalización.

Conclusion

Intercambiar una tarea más larga que aparece antes por una más corta que aparece después reduce la suma de tiempos de terminación.

Lo que nos lleva a que debemos de ordenar los p_i de manera creciente.

Modelación de problemas con matroides ponderadas: Árboles de expansión mínima y hospedaje

Demuestre que el minimum-spanning-tree problem y el problema de sitios de hospedaje descritos en clase se pueden modelar como matroides ponderadas (es decir, identifique y describa el conjunto S, la función de pesos y la familia de conjuntos independientes). Demuestre que cumplen con las propiedades de herencia e intercambio.

1. Minimum-Spanning-Tree Problem (Árbol de expansión mínima)

Descripción del problema

Dado un grafo no dirigido y conexo $G = (V, E)$ con pesos positivos asignados a cada arista $e \in E$, el objetivo del problema de **minimum spanning tree (MST)** es encontrar un subconjunto de aristas $T \subseteq E$ que:

- Conecte todos los vértices (es decir, forme un árbol de expansión)
- Tenga peso total mínimo:

$$\sum_{e \in T} w(e)$$

Modelación como matroide ponderada

Queremos demostrar que este problema puede modelarse como una **matroide ponderada** $M = (S, I, w)$.

1.1 Conjunto base (S) El conjunto de **aristas del grafo**, es decir:

$$S := E$$

1.2 Función de pesos (w) Función que asigna un peso positivo a cada arista:

$$w : S \to \mathbb{R}^+, \quad w(e) \text{ representa el costo de la arista } e$$

1.3 Familia de conjuntos independientes (I) La familia I está compuesta por todos los subconjuntos de aristas que **no forman ciclos**, es decir, **subconjuntos acíclicos** o **bosques**:

$$I := \{A \subseteq E \mid A \text{ no contiene ciclos}\}$$

Verificación de las propiedades de matroide

1. No-vaciedad

$$\emptyset \in I \quad \text{(el conjunto vacío no forma ciclos)}$$

- **Se cumple.**

2. Herencia Si $A \in I$, entonces cualquier subconjunto $B \subseteq A$ también es acíclico, porque eliminar aristas no puede introducir ciclos.

- **Se cumple.**

3. Intercambio Si $A, B \in I$ con $|A| < |B|$, entonces existe una arista $e \in B \ \ A$ tal que $A \cup \{e\} \in I$.

Esto es **cierto para los grafos**: en dos bosques acíclicos con distinta cantidad de aristas, siempre podemos agregar alguna arista del más grande al más pequeño **sin formar un ciclo** (ver notas sobre "propiedad de intercambio").

- **Se cumple.**

2. Problema de Hospedaje (Unit Task Scheduling Problem)

Descripción del problema

Dado un conjunto de tareas $T = \{t_1, t_2, ..., t_n\}$, donde cada tarea t_i tiene:

- Un deadline $d_i \in \mathbb{Z}^+$: tiempo límite para completarla
- Una penalización $w_i \in \mathbb{R}^+$ si no se completa a tiempo

Todas las tareas requieren exactamente **una unidad de tiempo** para completarse y se ejecutan en una **única máquina** que procesa **una tarea a la vez.**

Objetivo

Encontrar un subconjunto de tareas y un orden de ejecución tal que **minimice la penalización total**, es decir:

$$\min \sum_{i=1}^{n} w_i \cdot I_i$$

donde la función indicadora I_i es:

$$I_i = \begin{cases} 0, & C_i \leq d_i \quad \text{(a tiempo)} \\ 1, & C_i > d_i \quad \text{(tarde)} \end{cases}$$

Modelación como matroide ponderada

2.1 Conjunto base (S) Conjunto de tareas:

$$S := \{t_1, t_2, ..., t_n\}$$

2.2 Función de pesos (w') Para aplicar estrategia greedy, transformamos los pesos con:

$$w'(t_i) = M - w_i \quad \text{donde} \quad M > \sum w_i$$

De esta forma, **maximizar** $w'(A)$ **equivale a minimizar** $\sum w_i \cdot I_i$**.**

2.3 Conjuntos independientes (I) Un subconjunto $A \subseteq S$ es **independiente** si existe un ordenamiento (schedule) de sus tareas que las completa **todas a tiempo.**

Formalmente:

$$A \in I \iff \exists\, \sigma : A \to \{1, ..., |A|\} \text{ tal que } C_i \leq d_i \quad \forall t_i \in A$$

Verificación de propiedades de matroide

1. No-vaciedad El conjunto vacío no tiene tareas → ninguna penalización.

- **Se cumple.**

2. Herencia Si A puede completarse a tiempo, cualquier subconjunto $B \subseteq A$ también.

- **Se cumple.**

3. Intercambio (Propiedad de aumento) Si $A, B \in I$ con $|A| < |B|$, se puede demostrar que **existe una tarea** $x \in B \ A$ tal que $A \cup \{x\}$ **también se puede completar a tiempo.**

Esto se demostró en clase por contradicción, considerando el ordenamiento de deadlines y argumentando que si **todas las combinaciones fallan**, entonces B tampoco sería agendable a tiempo, lo cual contradice $B \in I$.

- **Se cumple.**

Tiempo de ejecución del algoritmo greedy en una matroide

Enunciado

En una matroide $M = (S, I)$, donde S tiene n elementos, ¿qué puede asegurar sobre el tiempo de ejecución del algoritmo greedy correspondiente?

Solución

Algoritmo Greedy para Matroides

El algoritmo greedy para una matroide ponderada $M = (S, I)$ con función de peso $w : S \to \mathbb{R}$ se estructura de la siguiente manera:

1. Ordenar los elementos $e \in S$ según su peso $w(e)$
2. Inicializar un conjunto solución $A = \emptyset$
3. Para cada elemento $e \in S$ (en orden decreciente o creciente de peso, según el problema):
   - Si $A \cup \{e\} \in I$, entonces $A = A \cup \{e\}$
4. Retornar A

Análisis de Complejidad

1. Ordenamiento de los elementos Para ordenar n elementos, utilizando algoritmos eficientes como HeapSort, MergeSort o QuickSort, el tiempo requerido es:

$$O(n \log n)$$

2. Inicialización del conjunto solución Esta operación es constante:

$$O(1)$$

3. Verificación de independencia para cada elemento Para cada uno de los n elementos, debemos verificar si al agregarlo al conjunto actual se mantiene la

independencia. Si denotamos el costo de esta verificación como $T_I(n)$, entonces el costo total de este paso es:

$$O(n \cdot T_I(n))$$

El valor de $T_I(n)$ depende de la implementación específica de la matroide:

- **Matroide Gráfica**: Para verificar que un conjunto de aristas no forma ciclos, se puede utilizar un algoritmo de búsqueda en profundidad o unión-búsqueda (Union-Find), lo que da $T_I(n) = O(\alpha(n))$ donde α es la función inversa de Ackermann, que es prácticamente constante.
- **Matroide de Partición**: Para verificar que no se seleccionan más de un elemento de cada partición, puede ser $T_I(n) = O(1)$ usando estructuras adecuadas.
- **Caso General**: En el peor caso, la verificación podría requerir $T_I(n) = O(n)$.

Complejidad Total

Combinando los análisis anteriores, el tiempo de ejecución total del algoritmo greedy es:

$$O(n \log n + n \cdot T_I(n))$$

Podemos asegurar que el tiempo de ejecución del algoritmo greedy en una matroide con n elementos está acotado inferiormente por $\Omega(n \log n)$ debido al paso de ordenamiento, y superiormente por:

- $O(n \log n)$ si la verificación de independencia es constante o casi constante
- $O(n^2)$ si la verificación de independencia toma tiempo lineal

Selección óptima de equipo interdisciplinario bajo restricciones de diversidad

Parte a: Demostración de Matroide y Algoritmo Greedy

1. Demostración que el problema es una matroide ponderada

Para modelar este problema como una matroide ponderada, definimos:

- **Conjunto base (E)**: Todos los estudiantes disponibles `E = {estudiante_1, estudiante_2, ..., estudiante_n}`
- **Familia de conjuntos independientes (I)**: Todos los subconjuntos de E donde no hay dos estudiantes de la misma carrera

Propiedades que cumplen:

1. **Hereditaria:**
   - Si $B \in \mathcal{J}$ y $A \subseteq B \Rightarrow A \in \mathcal{J}$
   - Ejemplo: Si $\{A, D\}$ es válido (carreras diferentes), entonces $\{A\}$ también lo es.
2. **Propiedad de intercambio:**
   - Si $A, B \in \mathcal{J}$ y $|A| < |B| \Rightarrow \exists x \in B \quad A$ tal que $A \cup \{x\} \in \mathcal{J}$
   - Ejemplo: Si $A = \{A\}$ y $B = \{D, E\}$, podemos agregar D o E a A.

2. Algoritmo Greedy

```
def seleccionar_equipo(estudiantes):
    # Paso 1: Ordenar por promedio descendente
    estudiantes_ordenados = sorted(estudiantes, key=lambda x: x['promedio'], reverse=True)

    equipo = []
    carreras_seleccionadas = set()

    # Paso 2: Selección greedy
    for estudiante in estudiantes_ordenados:
        if estudiante['carrera'] not in carreras_seleccionadas:
            equipo.append(estudiante)
            carreras_seleccionadas.add(estudiante['carrera'])

    return equipo
```

Complejidad temporal: O(n log n) por el ordenamiento

Parte b: Instancia del Problema y Aplicación del Algoritmo

1. Datos de entrada

```
estudiantes = [
    {"nombre": "Ana", "carrera": "Ingeniería en Ciencias de la Computación", "promedio": 95},
    {"nombre": "Carlos", "carrera": "Matemática Aplicada", "promedio": 90},
    {"nombre": "Beatriz", "carrera": "Ingeniería en Ciencias de la Computación", "promedio":
↪   85},
    {"nombre": "David", "carrera": "Licenciatura en Química", "promedio": 88},
    {"nombre": "Elena", "carrera": "Data Science", "promedio": 92}
]
```

2. Ejecución paso a paso

#	Nombre	Carrera	Prom.	Acción	Equipo Actual
1	Ana	Ing. Comp.	95	Agregar	[Ana]
2	Elena	Data Science	92	Agregar	[Ana, Elena]
3	Carlos	Mat. Aplicada	90	Agregar	[Ana, Elena, Carlos]
4	David	Química	88	Agregar	[Ana, Elena, Carlos, David]
5	Beatriz	Ing. Comp.	85	Rechazar (repetida)	-

3. Resultado final

Equipo seleccionado:

1. Ana (Ing. Ciencias Computación) - 95
2. Elena (Data Science) - 92
3. Carlos (Matemática Aplicada) - 90
4. David (Licenciatura en Química) - 88

Suma total de promedios: 365

4. Análisis de optimalidad

Para demostrar que esta solución es óptima:

- No se puede incluir a Beatriz sin eliminar a Ana (mayor promedio)
- Todos los estudiantes seleccionados tienen los mayores promedios de sus respectivas carreras
- No existe ninguna combinación válida con mayor suma de promedios

Conclusión

El problema cumple con la estructura de matroide y el algoritmo greedy garantiza encontrar la solución óptima en tiempo O(n log n), seleccionando siempre los estudiantes con mayores promedios de carreras no repetidas.

Comparación de algoritmos para caminos más cortos: Dijkstra vs Bellman-Ford

Demostración de la Complejidad del Algoritmo de Dijkstra

Descripción del Algoritmo

El algoritmo de Dijkstra resuelve el problema del camino más corto entre dos vértices sobre un grafo ponderado positivamente y dirigido. Utiliza un enfoque codicioso (greedy) para seleccionar en cada paso el vértice no visitado con la menor distancia tentativa.

Pseudocódigo

```
función Dijkstra(Grafo, nodoInicial):
    // Inicialización
    para cada vértice v en Grafo:
        distancia[v] = infinito
        visitado[v] = falso
    distancia[nodoInicial] = 0

    // Proceso principal
    mientras existan nodos no visitados:
        u = vértice no visitado con menor distancia
        visitado[u] = verdadero

        para cada vecino v de u:
            distanciaTentativa = distancia[u] + peso(u, v)
            si distanciaTentativa < distancia[v]:
                distancia[v] = distanciaTentativa

    retornar distancia[]
```

Análisis de Complejidad

Implementación con Array Simple

Operaciones Críticas

1. **Inicialización**: O(V) - Asignamos valores iniciales a cada vértice
2. **Bucle principal**: Se ejecuta V veces (una por cada vértice)

- **Encontrar el vértice con distancia mínima**: O(V) - Recorrido lineal
- **Actualizar distancias de vecinos**: En el peor caso O(V)

Demostración Formal Sea G = (V, E) un grafo con |V| vértices y |E| aristas:

- El algoritmo realiza exactamente |V| extracciones del mínimo. Prueba: Cada vértice se marca como visitado exactamente una vez, y para cada marcado se realiza una extracción.
- Cada arista se examina a lo sumo una vez. Prueba: Una arista (u,v) se examina solo cuando u se extrae de la cola, lo que ocurre exactamente una vez.

Teorema: Con un array simple, la complejidad de Dijkstra es O(V²). Prueba:

- Extracciones del mínimo: $|V| * O(V) = O(V^2)$
- Examen de aristas: O(E), pero acotado por O(V²) ya que $E <= V^2$
- Por tanto, la complejidad total es O(V²)

Implementación con Cola de Prioridad (Min-Heap)

Operaciones Críticas

1. **Extracción del mínimo**: O(log V)
2. **Actualización de distancia (decrease-key)**: O(log V)
3. **Número de operaciones**:
    - **Extracciones**: V en total
    - **Actualizaciones potenciales**: Una por cada arista, O(E) en total

Demostración Formal **Teorema**: Con una cola de prioridad binaria, la complejidad de Dijkstra es $O((V + E) * logV)$. Prueba:

- Extracciones del mínimo: $|V| * O(logV) = O(V * logV)$
- Operaciones decrease-key: $O(E) * O(logV) = O(E * logV)$
- Por tanto, la complejidad total es $O((V + E) * logV)$

Cuando E > V, podemos simplificar a O(E·log V).

Implementación con Cola de Fibonacci

La implementación con cola de Fibonacci mejora la complejidad teórica:

- Extraer mínimo: O(log V) amortizado
- Operación decrease-key: O(1) amortizado
- Complejidad total: O(V·log V + E)

Comparación con Bellman-Ford

- **Dijkstra**: O(V²) con array simple, O(E·log V) con heap binario
- **Bellman-Ford**: O(n log n)

Conclusión

La complejidad del algoritmo de Dijkstra con enfoque greedy varía según la implementación:

- $O(V^2)$ con implementación básica
- $O(E * logV)$ con cola de prioridad binaria
- $O(V * logV + E)$ con cola de Fibonacci

Esta demostración confirma que la implementación con estructuras de datos adecuadas hace que Dijkstra tenga un rendimiento similar al algoritmo Bellman-Ford, ambos son igual de útiles en la misma instancia del problema ya que tienen una complejiad similar, en otra instancia del problema también tienen un rendimiento similar