Laboratorio #3
Selección de Actividades

(Ingeniería de Sistemas)
Miguel Ángel Márquez Posso

(Ingeniería de Sistemas e Ingeniería Electrónica)
Ivan Dario Orozco Ibañez

Facultad de Ingeniería, Pontificia Universidad Javeriana

SISTEMAS 4800-6 (13107): Análisis de Algoritmos (Teo-Prac)

Andrés Dario Moreno Barbosa

13 de abril de 2025

Contents

Part I
Definición del problema

En el problema de selección de actividades de forma optima, se define para cada una de ellas un tiempo de inicio y un tiempo de finalización, además, se busca seleccionar el mayor número posible de actividades asegurándose que no se solapen entre si. Es decir, se pretende maximizar el número de actividades que pueden realizarse sin interferencias entre ellas. Para abordar este problema se pueden utilizar diferentes técnicas algorítmicas: una solución voraz (greedy) que ordena las actividades por su tiempo de finalización y selecciona secuencialmente las siguientes actividades compatibles, y otra basada en programación dinámica, la cual explora de forma recursiva todas las combinaciones posibles utilizando una matriz de memoización para evitar cálculos repetidos.

Part II
Algoritmos

1 Solución Voraz

1.1 Algoritmo

El algoritmo utiliza una estrategia voraz para construir una secuencia óptima de actividades compatibles. Primero, ordena todas las actividades en orden creciente según su tiempo de finalización. Luego, selecciona la primera actividad (la que finaliza más temprano) como punto inicial de la secuencia. A partir de allí, itera sobre las actividades restantes y añade a la secuencia aquella cuya hora de inicio sea compatible, es decir, que no se solape con la actividad previamente seleccionada, priorizando siempre la actividad con el menor tiempo de finalización disponible.

Algorithm 1 Algoritmo Voraz para Selección de Actividades

```
1:  procedure GREEDY(A)
2:      C ← OrdenarPorTiempoFinalizacion(A)
3:      R ← [C[0]]
4:      k ← 0
5:      for m ← 1 to |C| − 1 do
6:          if C[m].inicio ≥ C[k].fin then
7:              R.append(C[m])
8:              k ← m
9:          end if
10:     end for
11:     return R
12: end procedure
```

1.2 Complejidad Temporal

Peor Caso: $O(n \log n)$

El algoritmo Greedy para selección de actividades presenta una complejidad temporal de $O(n \log n)$ determinada por las siguientes operaciones:

1. **Ordenamiento inicial**: $O(n \log n)$
   - La secuencia A de n actividades se ordena según tiempos de finalización
   - Particularmente, en el codigo fuente se implementa el algoritmo Timsort mediante la función sorted con una función lambda como clave de ordenamiento mediante el argumento key. El algoritmo Timsort es una combinación eficiente de merge sort e insertion sort, optimizado para manejar listas parcialmente ordenadas. Su complejidad temporal en el mejor caso es de $O(n)$ cuando los datos ya están ordenados, mientras que en el promedio y peor de los casos alcanza $O(n \log n)$.
   - Esta operación domina la complejidad total del algoritmo
2. **Inicialización del resultado**: $O(1)$
   - Se añade la primera actividad de la secuencia ordenada al resultado
3. **Recorrido y selección**: $O(n)$
   - Un único recorrido lineal por la secuencia ordenada ($n - 1$ iteraciones)
   - En cada iteración se realiza una comparación y posiblemente una inserción, ambas en $O(1)$
4. **Retorno del resultado**: $O(1)$

Complejidad total: $T(n) = O(n \log n) + O(1) + O(n) + O(1) = O(n \log n)$

Esta complejidad se mantiene constante para mejor caso, caso promedio y peor caso, ya que el ordenamiento siempre requiere $O(n \log n)$ operaciones y el recorrido examina todos los elementos exactamente una vez.

1.3 Complejidad Espacial

Peor Caso: $O(n)$

El algoritmo requiere espacio adicional proporcional al tamaño de la entrada, distribuido de la siguiente manera:

1. **Secuencia ordenada** C: $O(n)$
   - Almacena las n actividades ordenadas por tiempo de finalización
   - El algoritmo de ordenamiento Timsort utiliza buffers temporales adicionales de tamaño $O(n)$

2. **Secuencia resultado** R: $O(n)$
   - Inicialmente contiene solo la primera actividad
   - En el peor caso, todas las actividades podrían ser compatibles y formar parte de la solución
3. **Variables auxiliares**: $O(1)$
   - Índices k y m para controlar el recorrido y la selección

Complejidad total: $S(n) = O(n) + O(n) + O(1) = O(n)$

Esta complejidad espacial también se mantiene constante en el mejor caso, caso promedio y peor caso, independientemente de las características específicas de los datos de entrada.

2 Solución Dinámica

2.1 Algoritmo

El algoritmo propuesto se basa en un enfoque Top-down de Programación Dinámica, el cual se encarga de resolver el problema de selección de actividades dividiéndolo en subproblemas cada vez más pequeños, hasta alcanzar casos base triviales. A continuación se enumeran los pasos fundamentales que describen cómo se desarrolla el algoritmo:

1. **Pre-procesamiento:**
   - Se ordena el conjunto de actividades A según su hora de finalización.
   - Se añaden dos elementos sentinela, uno al inicio y otro al final, para simplificar el manejo de las fronteras y casos base.
   - Se inicializa una matriz M de dimensiones $(n \times n)$, donde n es el número de actividades, llenándola con el valor $-\infty$ (ya que se busca maximizar la cantidad de actividades) para indicar que el subproblema aún no ha sido resuelto.
2. **Inicialización recurrente:** Se plantea el problema completo como la búsqueda de la solución óptima para el intervalo $[0, n-1]$ que abarca todas las actividades.
3. **División en subproblemas:** Para resolver el problema en el intervalo $[i, j]$, se identifica un conjunto S de índices k tales que la actividad $A[k]$ es compatible con la actividad representada por el límite izquierdo $A[i]$ y el derecho $A[j]$. Esto corresponde a descomponer el problema en dos subproblemas:
   - El subproblema que abarca el intervalo $[i, k]$.
   - El subproblema que abarca el intervalo $[k, j]$.
4. **Caso Base:** Si para un intervalo $[i, j]$ no existen índices k compatibles (es decir, el conjunto S es vacío), se define que la solución para ese subproblema es 0.

5. **Recurrencia:** En el caso contrario, para cada k en el conjunto S, el algoritmo calcula:

$$v = 1 + \text{solución}(i, k) + \text{solución}(k, j)$$

y selecciona el valor máximo q obtenido entre todas las opciones. Este valor se almacena en $M[i][j]$ para reutilizarlo en llamadas futuras, evitando cálculos repetidos.

Algorithm 2 ActivitySelectorDP

```
1: procedure ActivitySelectorDP(A)
2:     C ← sort(A, key = λ a : a[2])
3:     C.insert("__not_valid__" with (−∞, 0) at the beginning of C)
4:     C.append("__not_valid__" with (C[last][2], +∞) to C)
5:     M ← < 0, ..., |C| > < 0, ..., |C| >
6:     M ← < −∞ >
7:     return ActivitySelectorDP_Aux(C, 0, n − 1, M)
8: end procedure
```

Algorithm 3 ActivitySelectorDP_Aux

```
1: procedure ActivitySelectorDP_Aux(A, i, j, M)
2:     if M[i][j] = −∞ then
3:         S ← [∅]
4:         for k ← i to j do
5:             if A[k][1] ≥ A[i][2] and A[k][2] ≤ A[j][1] then
6:                 S.append(k)
7:             end if
8:         end for
9:         if |S| = 0 then
10:            M[i][j] ← 0
11:        else
12:            q ← −∞
13:            for all k ∈ S do
14:                v ← 1 + ActivitySelectorDP_Aux(A, i, k, M)
15:                v ← v + ActivitySelectorDP_Aux(A, k, j, M)
16:                if q < v then
17:                    q ← v
18:                end if
19:            end for
20:            M[i][j] ← q
21:        end if
22:    end if
23:    return M[i][j]
24: end procedure
```

2.2 Complejidad Temporal

El algoritmo basado en programación dinámica Top-down se puede analizar mediante la siguiente fórmula:

$$T(n) = (\#\text{ de subproblemas}) \times (\text{costo por subproblema})$$

Peor Caso:

Para cada subproblema definido por un par (i, j), se recorre el intervalo de i a j en un bucle, lo que tiene un coste en el peor caso de $\mathcal{O}(n)$. Dado que existen $\mathcal{O}(n^2)$ subproblemas, se tiene:

$$T(n) = \mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$$

Caso Promedio:

En el caso promedio se asume que el conjunto S (de índices compatibles) tiene un tamaño constante en promedio; sin embargo, para determinar S es necesario iterar a lo largo del rango $[i, j]$, lo que supone un coste de $\mathcal{O}(n)$ por subproblema. Así:

$$T(n) = \mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$$

Mejor Caso:

Incluso en el mejor escenario, cuando la condición en el bucle interna falla frecuentemente (resultando en un conjunto S vacío), el algoritmo debe examinar todos los posibles k entre i y j, incurriendo en un coste de $\mathcal{O}(n)$ por subproblema. De este modo:

$$T(n) = \mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$$

Por lo tanto, la complejidad temporal del algoritmo, tanto en el peor, promedio y mejor de los casos, es:

$$T(n) = \Theta(n^3)$$

2.3 Complejidad Espacial

El algoritmo utiliza una matriz de memoización M de dimensiones $n \times n$, donde n representa el número total de actividades (incluyendo los sentinelas). Esta estructura de datos requiere un espacio del orden de $\mathcal{O}(n^2)$. Adicionalmente, la profundidad de la recursión es del orden de $\mathcal{O}(n)$, pero este uso de la pila es despreciable comparado con el requerimiento principal de la matriz de memoización. Así, la complejidad espacial total del algoritmo es:

$$S(n) = \mathcal{O}(n^2)$$

Part III
Comparación de los Algoritmos

3 Complejidad Temporal y Espacial

La comparación entre el algoritmo voraz (Greedy) y el enfoque de programación dinámica para el problema de selección de actividades revela diferencias significativas tanto en términos de complejidad temporal y espacial, así como en la aplicabilidad a diferentes escenarios. El algoritmo voraz exhibe una complejidad temporal de $O(n \log n)$, dominada principalmente por la operación inicial de ordenamiento de las actividades según sus tiempos de finalización, seguida de un recorrido lineal $O(n)$ para seleccionar las actividades compatibles; mientras que su complejidad espacial es estrictamente $O(n)$ debido a las estructuras de datos necesarias para almacenar la secuencia ordenada y la solución resultante. En contraste, la implementación de programación dinámica presenta una complejidad temporal significativamente mayor de $O(n^3)$ en el peor caso, donde para cada par de índices (i, j) se realiza una búsqueda a través de todos los índices intermedios k, generando un patrón de triple anidamiento; adicionalmente, la programación dinámica requiere una matriz de memoización M de dimensión $n \times n$ para almacenar los resultados parciales, resultando en una complejidad espacial de $O(n^2)$, sustancialmente mayor que la del enfoque voraz.

4 Calidad de la Solución

Es crucial destacar que para el problema específico de selección de actividades con el objetivo de maximizar el número de actividades seleccionadas, el algoritmo voraz garantiza encontrar una solución óptima debido a la estructura matemática particular del problema que satisface las propiedades de subestructura óptima y elección voraz óptima, mientras que la programación dinámica, aunque es computacionalmente más costosa, también garantiza ser una solución óptima con el beneficio adicional de poder adaptarse fácilmente a variantes del problema como la maximización del valor de las actividades cuando éstas tienen pesos o valores diferenciados, escenario donde el algoritmo voraz podría no encontrar la solución óptima.

La eficiencia relativa de estos enfoques sugiere que el algoritmo voraz es preferible para instancias grandes del problema estándar de selección de actividades, mientras que la programación dinámica podría ser más adecuada para variantes del problema cuando la flexibilidad para adaptarse a cambios en las especificaciones del problema es más importante que la eficiencia computacional bruta. Adicionalmente, es importante considerar que, mientras el algoritmo voraz siempre selecciona inmediatamente la actividad con el tiempo de finalización más temprano entre las compatibles con la última actividad seleccionada, el enfoque de programación dinámica explora recursivamente todos los posibles subproblemas, lo que significa que este último podría adaptarse más fácilmente para incorporar restricciones adicionales como dependencias entre actividades, interrupciones permitidas, o recursos múltiples, situaciones donde el simple criterio voraz podría fallar en

encontrar la solución óptima, ilustrando así que la elección del algoritmo óptimo depende no solo de consideraciones de complejidad, sino también de las características específicas del problema y los requisitos del sistema en el que se implementará la solución.

Archivo	Actividades	Rango de slots	Greedy	Prog. Dinámica
activities_1	100	0-48	13	13
activities_2	100	0-48	15	15
activities_3	100	0-48	14	14
activities_4	100	0-48	11	11
activities_5	100	0-48	15	15
activities_6	100	0-48	12	12
activities_7	100	0-48	15	15
activities_8	100	0-48	11	11
activities_9	100	0-48	15	15
activities_10	100	0-48	15	15
activities_11	100	0-48	15	15
activities_12	100	0-48	14	14
activities_13	100	0-48	12	12
activities_14	100	0-48	13	13
activities_15	100	0-48	12	12
activities_16	100	0-48	12	12
activities_17	100	0-48	11	11
activities_18	100	0-48	14	14
activities_19	100	0-48	15	15
activities_20	100	0-48	17	17
activities_21	100	0-48	15	15
activities_22	100	0-48	18	18
activities_23	100	0-48	13	13
activities_24	100	1-48	15	15

Figure 4.1: Tabla de resultados de seleccionar actividades de 100 archivos, usando el algoritmo voraz y el de enfoque de programación dinámica (Parte 1).

activities_25	100	0-48	19	19
activities_26	100	0-48	14	14
activities_27	100	0-48	15	15
activities_28	100	0-48	13	13
activities_29	100	0-48	12	12
activities_30	100	0-48	15	15
activities_31	100	0-48	13	13
activities_32	100	0-48	15	15
activities_33	100	0-48	13	13
activities_34	100	1-48	13	13
activities_35	100	0-48	14	14
activities_36	100	0-48	15	15
activities_37	100	0-48	11	11
activities_38	100	1-48	13	13
activities_39	100	1-48	11	11
activities_40	100	0-48	15	15
activities_41	100	0-48	13	13
activities_42	100	0-48	13	13
activities_43	100	0-48	11	11
activities_44	100	0-48	14	14
activities_45	100	0-48	15	15
activities_46	100	1-48	14	14
activities_47	100	0-48	14	14
activities_48	100	1-48	15	15
activities_49	100	0-48	16	16
activities_50	100	0-48	13	13
activities_51	100	0-48	14	14
activities_52	100	0-48	14	14

Figure 4.2: Tabla de resultados de seleccionar actividades de 100 archivos, usando el algoritmo voraz y el de enfoque de programación dinámica (Parte 2).

activities_53	100	0-48	10	10
activities_54	100	0-48	15	15
activities_55	100	0-48	13	13
activities_56	100	0-48	13	13
activities_57	100	0-48	15	15
activities_58	100	0-48	13	13
activities_59	100	0-48	11	11
activities_60	100	0-48	15	15
activities_61	100	0-48	13	13
activities_62	100	0-48	13	13
activities_63	100	0-48	14	14
activities_64	100	0-48	14	14
activities_65	100	0-48	14	14
activities_66	100	0-48	13	13
activities_67	100	0-48	16	16
activities_68	100	0-48	12	12
activities_69	100	0-48	15	15
activities_70	100	0-48	13	13
activities_71	100	0-48	14	14
activities_72	100	0-48	17	17
activities_73	100	0-48	12	12
activities_74	100	0-48	13	13
activities_75	100	0-48	14	14
activities_76	100	0-48	13	13
activities_77	100	0-48	17	17
activities_78	100	1-48	16	16
activities_79	100	0-48	16	16
activities_80	100	1-48	13	13
activities_81	100	0-48	13	13

Figure 4.3: Tabla de resultados de seleccionar actividades de 100 archivos, usando el algoritmo voraz y el de enfoque de programación dinámica (Parte 3).

activities_82	100	0-48	14	14
activities_83	100	0-48	12	12
activities_84	100	0-48	13	13
activities_85	100	0-48	14	14
activities_86	100	1-48	16	16
activities_87	100	0-48	15	15
activities_88	100	0-48	13	13
activities_89	100	0-48	13	13
activities_90	100	0-48	13	13
activities_91	100	0-48	12	12
activities_92	100	0-48	16	16
activities_93	100	0-48	12	12
activities_94	100	0-48	12	12
activities_95	100	0-48	15	15
activities_96	100	2-48	11	11
activities_97	100	0-48	12	12
activities_98	100	0-48	12	12
activities_99	100	0-48	14	14
activities_100	100	0-48	12	12

Figure 4.4: Tabla de resultados de seleccionar actividades de 100 archivos, usando el algoritmo voraz y el de enfoque de programación dinámica (Parte 4).