

Trabajo Práctico 1: Técnicas Algorítmicas

Compilado: 11 de abril de 2022

Fecha de entrega: 20 de abril de 2022

Fecha de reentrega: 27 de junio de 2022.

Este trabajo práctico consta de cuatro ejercicios más o menos independientes. Para la aprobación del trabajo práctico se debe entregar:

1. Un informe *de a lo sumo doce páginas* (sin contar las tablas de experimentación) que describa la solución de los ejercicios y responda a todas las consignas del enunciado.
2. El código fuente con los programas que implementan las soluciones propuestas a los ejercicios.

El informe debe ser autocontenido, lo que significa que cualquier estudiante potencial de AED3 que conozca los temas debe poder leerlo sin necesidad de conocer cuál fue el enunciado. En particular, todos los conceptos y notaciones utilizados que no sean comunes a la materia deben definirse, incluso aquellos que se encuentren en el presente enunciado. No es necesario explicar aquellos conceptos propios de la materia ni la teoría detrás de las distintas técnicas algorítmicas o de demostración de propiedades.

El informe debe estar estructurado usando una sección independiente por cada ejercicio resuelto. Esto no impide que un ejercicio haga referencias a la solución de otro ejercicio; las comparaciones son muchas veces bienvenidas o parte de la consigna. Cada sección debe dar una respuesta cabal a todas las preguntas del enunciado. Sin embargo, el informe no es una sucesión de respuestas a las distintas consignas, sino una elaboración única y coherente donde se pueden encontrar las respuestas a las preguntas. La idea es que el informe sea de agradable lectura.

El informe **no debe incluir código fuente**, ya que quienes evaluamos tenemos acceso al código fuente del programa. En caso de utilizar *pseudocódigo*, el mismo debe ser de alto nivel, evitando construcciones innecesariamente complicadas para problemas simples.¹ Todo pseudocódigo debe estar acompañado de un texto coloquial que explique lo que el pseudocódigo hace en términos conceptuales.

El código fuente entregado debe venir acompañado de un documento que explique cómo compilar y ejecutar el programa. El código fuente debe compilar y debe resolver correctamente **todos** los casos de test en un tiempo que se corresponda a la complejidad esperada. En todos los ejercicios, la instancia se leerá de la entrada estándar y la solución se imprimirá en la salida estándar. Junto a este enunciado se incluyen algunos casos de test, mientras que la complejidad temporal en peor caso forma parte del enunciado. Tener en cuenta que el programa puede ser probado en casos de test adicionales.

Para aprobar el trabajo práctico es necesario aprobar cada ejercicio en forma individual, ya sea en la primera entrega o en el recuperatorio. No es necesario reentregar aquellos ejercicios que sean aprobados en la primer entrega. Asimismo, para aprobar cada ejercicio es necesario que el informe describa correctamente la solución y que el programa propuesto sea correcto. La correcta escritura del informe forma parte de la evaluación.

Ejercicio 1: *Backtracking*

Consideremos una *red social* similar a *Facebook*, en la que cada *actor* (persona, organización, etc.) tiene asociado un valor que determina su *poder de influencia* sobre la red. Una *clique* es un subgrupo de

¹E.g., “tomar el máximo del vector V ” vs. “poner $\text{max} := V_0$; para cada $i = 1, \dots, n - 1$: poner $\text{max} := \text{maximo}(\text{max}, V_i)$;

actores que son todos amigos entre sí. Es sabido que las cliques forman subgrupos altamente cohesivos que tienden a tener opiniones similares. Cuanto más poder de influencia tengan los actores de una clique, más posibilidades hay de que su forma de pensar marque tendencia en la red. En este ejercicio resolvemos el problema de la *clique más influyente*: dada la descripción de una red social, queremos encontrar una clique cuya influencia sea máxima.

Formalmente, una red social es una tripla $G = (V, E, p)$ tal que:

- $V = \{1, \dots, n\}$ es el conjunto de actores, cada uno de los cuales se identifica con un número,
- E es el conjunto *amistades* y está formado por subconjuntos de V que tienen exactamente dos actores, y
- $p: V \rightarrow \mathbb{N}_{>0}$ es la función que denota el poder de influencia de cada actor.

Por simplicidad, vamos a escribir vw para denotar al conjunto $\{v, w\}$ para todo $v, w \in V$. Decimos que v y w son *amigos* cuando $vw \in E$ y que son *no-amigos* en caso contrario. Un conjunto $Q \subseteq V$ es una *clique* de G cuando v y w son amigos para todo $v, w \in Q$. El *poder de influencia* de una clique Q es $p(Q) = \sum_{v \in Q} p(v)$. Las cliques de influencia máxima de G son aquellas cliques de G cuyo poder de influencia es $\max\{p(Q) \mid Q \text{ es clique de } G\}$. Dada una red social $G = (V, E, p)$, el problema de la clique más influyente consiste en determinar una clique de influencia máxima.

1. Diseñar un algoritmo de backtracking que, dada una red social G , resuelva el problema de la clique más influyente de G . El algoritmo **debe** estar basado en las siguientes ideas y cada nodo **debe** procesarse en $O(n^2)$ tiempo:

- Todas las soluciones parciales del árbol de backtracking son cliques de G .
- Cada nodo del árbol de backtracking representa un conjunto de soluciones (cliques) \mathcal{Q} que se obtienen al visitar las hojas de su subárbol; todas las cliques de \mathcal{Q} contienen a la solución parcial (clique) Q . Intuitivamente, en el proceso de extensión ya se tomó la decisión de que cada actor de Q esté en las soluciones que se generan a partir de este nodo.
- Además de Q , cada nodo del árbol de backtracking tiene asociado un conjunto de actores K que es disjunto a Q . Intuitivamente, K contiene los actores sobre los que aún no se tomó ninguna decisión en el proceso de extensión. A partir de ahora, hablamos del nodo (Q, K) para referirnos a aquel nodo correspondiente a una solución parcial Q cuyo conjunto asociado es K .
- En cada nodo (Q, K) , el conjunto K satisface dos propiedades invariantes. Por un lado, cada $v \in K$ es amigo de todos los actores de Q , lo que implica que $Q \cup \{v\}$ es una clique de G . Por otro lado, cada $v \in K$ tiene un no-amigo en K , lo que implica que al menos una clique de G contiene a Q pero no a v .
- La extensión del nodo (Q, K) genera dos subárboles a partir de un actor $v \in K$. En uno de estos subárboles, cada hoja se corresponde a una clique que contienen a $Q \cup \{v\}$. En el otro subárbol, cada hoja representa una clique que contiene a Q y no a v .
- Para reducir el espacio de búsqueda, se poda cada nodo (Q, K) tal que $p(Q) + \sum_{v \in K} p(v) \leq p(S)$ para la clique S de mayor influencia encontrada hasta el momento.
- El árbol se recorre recursivamente, avanzando primero en profundidad; pensar qué hijo conviene visitar primero en cada nodo.

- Antes de empezar, los actores se ordenan por algún criterio (ver abajo). En cada extensión, el candidato v es el primer vértice del orden que pertenece a K .
 - **Ayuda:** recordar que al extender el nodo (Q, K) a partir de un vértice $v \in K$ se obtienen dos nodos hijo (Q_I, K_I) y (Q_D, K_D) . Sin pérdida de generalidad, supongamos que $v \in Q_I$, i.e., el subárbol de (Q_I, K_I) representa a cliques que contienen a $Q \cup \{v\}$. En cambio (Q_D, K_D) representa a las cliques que contienen a Q y no a v . Notar que, por invariante, ni Q_I ni K_I pueden contener actores que sean no-amigos de v . Por otra parte, K_I tampoco puede contener actores que sean amigos de todos los actores en $Q_I \cup K_I$; estos últimos pertenecen a todas las cliques que contienen a $Q \cup \{v\}$ y por lo tanto se pueden agregar a Q_I . Pensar cómo computar el resto de los parámetros de cada nodo, siguiendo un razonamiento similar.
2. Considerar al menos dos opciones para ordenar a los actores antes de ejecutar el algoritmo: de mayor a menor influencia y de menor a mayor influencia.
 3. Realizar un experimento con los casos de test de ambas opciones y reportar los resultados, justificando los mismos a través de analizar los efectos de la poda.
 4. La operación de extensión define una función recursiva cuya descripción formal puede ser compleja. Sin definir formalmente esta función, explicar por qué la misma no tiene la propiedad de superposición de subproblemas.

Descripción de una instancia. Las instancias para este problema son públicas² y por lo tanto respetan (una restricción de) el formato DIMACS que es un estándar de facto para problemas sobre redes (y grafos). La primer línea de cada instancia tiene el formato `p edge N M` donde N y M denotan la cantidad de actores y amistades. A continuación hay N líneas que representan a cada actor y tienen el formato `n V I` donde V es un número entre 1 y N e I es la influencia del actor V . **Por simplicidad, las líneas aparecen en orden creciente de valor V .** Finalmente, el archivo tiene M líneas con formato `e V W` donde V y W son amigos.

Descripción de la salida. El programa debe imprimir la influencia de la clique más influyente, seguida de una línea con los k valores que identifican a los actores de dicha clique.

| Entrada de ejemplo | Salida esperada de ejemplo |
|---|----------------------------|
| <p>p edge 4 4</p> <p>n 1 1</p> <p>n 2 4</p> <p>n 3 3</p> <p>n 4 8</p> <p>e 1 2</p> <p>e 1 3</p> <p>e 2 3</p> <p>e 2 4</p> | <p>12</p> <p>2 4</p> |

²<https://github.com/jamestrimble/max-weight-clique-instances>

Soluciones de referencia. El Cuadro 1 muestra el valor de las cliques más influyentes y el tiempo requerido para obtener la misma en una implementación de referencia³ (columna Tiempo Ej 1). La implementación considera el orden de mayor a menor influencia para seleccionar cada actor para la extensión. El programa fue compilado con la opción `-O3` y fue ejecutado en procesador AMD Ryzen 7 3700U (single thread)⁴. El tiempo de ejecución sirve únicamente de referencia.

Ejercicio 2: Podando el árbol de backtracking

Sea $G = (V, E, p)$ una red social. Un conjunto de actores $I \subseteq V$ de G es *independiente* cuando $vw \notin E$ para todo $v, w \in I$. El *poder de influencia* de un conjunto independiente I está definido como $\bar{p}(I) = \max\{p(v) \mid v \in I\}$. En este ejercicio vamos a implementar una poda para el problema de la clique más influyente a partir de las influencias de los conjuntos independientes.

1. Sea (Q, K) un nodo del árbol de backtracking y sea I_1, \dots, I_k una partición de K en conjuntos independientes. Observar que $p(Q) + \sum_{i=1}^k \bar{p}(I_i) \geq p(Q \cup Q')$ para toda clique $Q' \subseteq K$. Explicar por qué.
2. Diseñar un algoritmo goloso y eficiente (lineal o cuadrático) que, dado un conjunto K de actores de una red social G , compute una partición I_1, \dots, I_k de K tratando de minimizar $\sum_{i=1}^k \bar{p}(I_i)$. No es necesario que el algoritmo encuentre el mínimo valor posible, pero sí que lo intente, i.e., que cada paso del algoritmo goloso busque minimizar el poder de influencia total de la partición.
3. Implementar una poda en el árbol de backtracking basada en los incisos anteriores. Es decir, se poda el nodo (Q, K) cuando $p(Q) + \sum_{i=1}^k \bar{p}(I_i) \leq p(S)$ para la partición I_1, \dots, I_k de K y la mejor solución S conocida.
4. Comparar el tiempo de ejecución del algoritmo del ejercicio anterior versus el que resulta de implementar esta nueva poda.

Soluciones de referencia. El tiempo de ejecución de referencia se puede encontrar en la columna “Tiempo Ej 2” del Cuadro 1; la computadora usada es la misma.

Ejercicio 3: Programación dinámica

Dado un conjunto de actividades $\mathcal{A} = \{A_0, \dots, A_{n-1}\}$ y una función de beneficio $b: \mathcal{A} \rightarrow \mathbb{N}$, el *problema de selección de actividades* consiste en encontrar un subconjunto de actividades \mathcal{S} cuyo beneficio $b(\mathcal{S}) = \sum_{A \in \mathcal{S}} b(A)$ sea máximo de entre todos aquellos subconjuntos de actividades que no se solapan en el tiempo. Cada actividad A_i se realiza en algún intervalo de tiempo $[s_i, t_i]$, siendo $s_i \in \mathbb{N}$ su momento inicial y $t_i \in \mathbb{N}$ su momento final. Suponemos que $0 \leq s_i < t_i \leq 2n$ para todo $0 \leq i < n$.

A los subconjuntos \mathcal{S} de actividades no se solapan en el tiempo los llamamos *compatibles*.

1. Describir cómo se puede resolver el problema de selección de actividades utilizando cualquier solución al problema de la clique más influyente.

³La implementación es similar a la que esperamos que sea entregada, utilizando estructuras de datos cómodas para los tiempos solicitados y sin detalles de implementación sofisticados.

⁴Los tiempos no deberían ser muy distintos en una máquina razonablemente moderna. Usar <https://www.cpubenchmark.net/> para comparar procesadores en caso de dudas.



| Instancia | Optimo | Tiempo Ej 1(seg.) | Tiempo Ej 2 (seg.) |
|--------------------|--------|-------------------|--------------------|
| brock200_1.clq | 2821 | 20.27 | 0.98 |
| brock200_2.clq | 1428 | 0.14 | 0.03 |
| brock200_3.clq | 2062 | 0.57 | 0.10 |
| brock200_4.clq | 2107 | 1.62 | 0.17 |
| brock400_2.clq | 3350 | > 600 | 364.40 |
| brock400_3.clq | 3471 | > 600 | 316.32 |
| brock400_4.clq | 3626 | > 600 | 248.83 |
| C125.9.clq | 2529 | 122.10 | 1.01 |
| C250.9.clq | 5092 | > 600 | 304.52 |
| c-fat200-1.clq | 1284 | 0.00 | 0.00 |
| c-fat200-2.clq | 2411 | 0.00 | 0.00 |
| c-fat200-5.clq | 5887 | 0.01 | 0.00 |
| c-fat500-10.clq | 11586 | 0.07 | 0.05 |
| c-fat500-1.clq | 1354 | 0.00 | 0.00 |
| c-fat500-2.clq | 2628 | 0.01 | 0.01 |
| c-fat500-5.clq | 5841 | 0.02 | 0.01 |
| DSJC500_5.clq | 1725 | 12.45 | 2.46 |
| gen200_p0.9_44.clq | 5043 | > 600 | 62.21 |
| gen200_p0.9_55.clq | 5416 | > 600 | 20.48 |
| hamming6-2.clq | 1072 | 0.04 | 0.00 |
| hamming6-4.clq | 134 | 0.00 | 0.00 |
| hamming8-2.clq | 10976 | > 600 | 0.06 |
| hamming8-4.clq | 1472 | 3.34 | 0.30 |
| johnson16-2-4.clq | 548 | 3.91 | 0.75 |
| johnson8-2-4.clq | 66 | 0.00 | 0.00 |
| johnson8-4-4.clq | 511 | 0.06 | 0.00 |
| MANN_a9.clq | 372 | 0.71 | 0.04 |
| p_hat1000-1.clq | 1514 | 3.03 | 1.49 |
| p_hat1500-1.clq | 1619 | 27.90 | 10.99 |
| p_hat300-1.clq | 1057 | 0.05 | 0.05 |
| p_hat300-2.clq | 2487 | 4.48 | 0.32 |
| p_hat300-3.clq | 3774 | > 600 | 9.13 |
| p_hat500-1.clq | 1231 | 0.20 | 0.13 |
| p_hat500-2.clq | 3920 | > 600 | 6.50 |
| p_hat700-1.clq | 1441 | 0.69 | 0.35 |
| p_hat700-2.clq | 5290 | > 600 | 153.73 |
| san1000.clq | 1716 | > 600 | 8.31 |
| san200_0.9_2.clq | 6082 | > 600 | 13.94 |
| san200_0.9_3.clq | 4748 | > 600 | 120.23 |
| san400_0.7_1.clq | 3941 | > 600 | 14.83 |
| san400_0.7_2.clq | 3110 | > 600 | 20.57 |
| san400_0.7_3.clq | 2771 | > 600 | 21.90 |
| sanr200_0.7.clq | 2325 | 4.62 | 0.40 |
| sanr200_0.9.clq | 5126 | > 600 | 53.92 |
| sanr400_0.5.clq | 1835 | 3.88 | 0.82 |

Cuadro 1: Optimos y tiempos de ejecución de referencia para los Ejercicios 1 y 2.

2. Supongamos que \mathcal{A} está ordenado por orden de comienzo de la actividad, i.e., $s_i \leq s_{i+1}$ para todo $1 \leq i < n$. Escribir una función recursiva $b: \{0, \dots, n\} \rightarrow \mathbb{N}$ tal que:
 - $b(i)$ denota el máximo beneficio entre todos los subconjuntos de actividades compatibles incluidos en A_i, \dots, A_{n-1} . (Notar que $b(n) = 0$ por definición.)
 - b satisface la propiedad de superposición de subproblemas.
3. Demostrar que la función anterior es correcta. Para ello, formalice el problema de selección de actividades como un problema de optimización combinatoria.
4. Implementar un algoritmo de programación dinámica *top-down* para el problema de selección de actividades. El algoritmo debe computar el beneficio de la solución óptima en $O(n)$.
5. Implementar una versión bottom-up del algoritmo anterior, indicando claramente el orden de cómputo de cada subinstancia.
6. Implementar un algoritmo que reconstruya un conjunto de actividades compatibles de beneficio máximo.

Descripción de una instancia. Cada instancia consistirá de una primera línea con un entero N correspondiente a la cantidad de actividades. Luego le sucederán N líneas con tres enteros, S , T y B , correspondientes a los tiempos de inicio y final de cada actividad y a su beneficio asociado. La i -ésima línea representa la i -ésima actividad y, por simplicidad, consideramos que las instancias se numeran empezando en 0. **Notar que las actividades están ordenadas por sus tiempos de inicio.**

Descripción de la salida. La salida tendrá un número indicando el beneficio máximo que se puede obtener, seguido de una línea con las actividades que alcanzan dicho beneficio.

| Entrada de ejemplo | Salida esperada de ejemplo |
|--------------------|----------------------------|
| 4 | 6 |
| 1 2 2 | 0 3 |
| 1 3 3 | |
| 2 4 2 | |
| 3 6 4 | |

Soluciones de referencia. En el Cuadro 2 se muestra el valor óptimo para algunas instancias, junto al tiempo requerido para resolverlas, en base a una implementación de referencia. El programa fue compilado con la opción `-O3` y fue ejecutado en un procesador AMD Ryzen 7 3700U.

Ejercicio 4: Greedy

Considerar la siguiente estrategia golosa para resolver el problema de selección de actividades compatibles para el caso en que $b(A) = 1$ para toda actividad $A \in \mathcal{A}$: elegir la actividad cuyo momento final sea lo más temprano posible, de entre todas las actividades que no se solapan con las actividades ya elegidas.

1. Demostrar por inducción en la cantidad de elecciones que la estrategia anterior es correcta.

| Instancia | Tamaño | Óptimo | Tiempo (seg.) |
|------------------|--------|--------|---------------|
| instancia_1.txt | 5 | 15 | 0.00 |
| instancia_2.txt | 7 | 27 | 0.00 |
| instancia_3.txt | 10 | 30 | 0.00 |
| instancia_4.txt | 1 | 15 | 0.00 |
| instancia_5.txt | 10^4 | 1456 | 0.00 |
| instancia_6.txt | 10^4 | 1416 | 0.00 |
| instancia_7.txt | 10^4 | 1578 | 0.00 |
| instancia_8.txt | 10^5 | 5100 | 0.06 |
| instancia_9.txt | 10^5 | 5103 | 0.06 |
| instancia_10.txt | 10^5 | 4834 | 0.07 |
| instancia_11.txt | 10^6 | 15566 | 0.76 |
| instancia_12.txt | 10^6 | 15382 | 0.79 |
| instancia_13.txt | 10^6 | 15665 | 0.77 |

Cuadro 2: Óptimos y tiempos de ejecución de referencia para el Ejercicio 3

- Mostrar que la solución no es correcta cuando $b(A)$ no es necesariamente 1 para todo A .
- Implementar un algoritmo basado en la idea anterior cuya complejidad temporal sea $\mathcal{O}(n)$.
- Comparar este algoritmo con el de programación dinámica, marcando cuál es más simple de implementar.

Descripción de una instancia. Igual al ejercicio anterior, salvo porque:

- las actividades no tienen un beneficio asociado.
- Las actividades no están ordenadas por tiempo de inicio o fin.

Descripción de la salida. Como salida se espera la cantidad máxima k de actividades que se pueden seleccionar, seguida de una línea con k actividades que no se solapan.

| Entrada de ejemplo | Salida esperada de ejemplo |
|---|----------------------------|
| 7 1 3 1 5 2 4 8 11 3 6 1 2 5 9 | 3 3 4 5 |

Soluciones de referencia. En el Cuadro 3 se muestra el valor óptimo para algunas instancias, junto al tiempo requerido para resolverlas, en base a una implementación de referencia. El programa fue compilado con la opción `-O3` y fue ejecutado en un procesador AMD Ryzen 7 3700U.

| Instancia | Tamaño | Óptimo | Tiempo (seg.) |
|----------------------|-----------------|--------|---------------|
| interval_instance_1 | 5 | 3 | 0.00 |
| interval_instance_2 | 7 | 2 | 0.00 |
| interval_instance_3 | 10 | 2 | 0.00 |
| interval_instance_4 | 1 | 1 | 0.00 |
| interval_instance_5 | 10 ⁴ | 2747 | 0.00 |
| interval_instance_6 | 10 ⁴ | 1539 | 0.00 |
| interval_instance_7 | 10 ⁴ | 592 | 0.00 |
| interval_instance_8 | 10 ⁵ | 27776 | 0.03 |
| interval_instance_9 | 10 ⁵ | 15646 | 0.06 |
| interval_instance_10 | 10 ⁵ | 5708 | 0.05 |
| interval_instance_11 | 10 ⁶ | 276866 | 0.43 |
| interval_instance_12 | 10 ⁶ | 155455 | 0.49 |
| interval_instance_13 | 10 ⁶ | 57523 | 0.46 |

Cuadro 3: Optimos y tiempos de ejecución de referencia para el Ejercicio 4

Ayuda para experimentación

Usando Linux, se puede medir el tiempo de un algoritmo sobre un conjunto de instancias que se encuentran en un directorio `<instancias>` con un comando desde la consola. Para ello, ejecute:

```
for i in <instancias>
do
    echo "Procesando $i"
    /usr/bin/time -f "%e" timeout <limite> <ejecutable> < $i > $i.output
done
```

donde `<limite>` es el tiempo limite en segundos, `<ejecutable>` es el nombre del programa ejecutable. Por cada instancia se genera un archivo con el mismo nombre y el sufijo `.output`.

En caso de utilizar la biblioteca `iostream`, incluya las siguientes dos líneas al inicio de la función `main` para reducir los tiempos de ejecución para lectura y escritura:

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
```