Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Docentes: Paulo Mejía; Tomás Gálvez P.
Semestre 1, 2023

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Algoritmos *Greedy*

Para algunos problemas de optimización podemos emplear un acercamiento más económico que el de programación dinámica. Este es el acercamiento ***greedy*, voraz** o **codicioso**, llamado así por construir soluciones "en el camino", tomando cada decisión según lo que parezca mejor en el momento. Es decir, los algoritmos *greedy* simplemente cambian la forma en la que se toman las decisiones durante la construcción de una solución óptima. Ya no nos metemos a resolver subproblemas antes de tomar una decisión, sino que tomamos la decisión basándonos en la información que el problema actual provee.

La subestructura óptima también es un requerimiento de los problemas de optimización a los que se aplique el acercamiento *greedy* porque este acercamiento producirá, en cada paso, subproblemas que serán resueltos con base en un mismo criterio. Lo distintivo del acercamiento es que no busca cuál de todas las alternativas es la mejor antes de tomar una decisión. Más bien, define un **procedimiento de selección**, que es una idea de cómo elegir qué agregar a la solución final, y lo aplica, encomendándose a los dioses. Es por esto que el acercamiento *greedy* no nos garantiza una solución óptima, como la programación dinámica, a menos que se cumplan ciertas condiciones. Lo que sí nos garantiza es proveer una solución de forma más eficiente.

1. El procedimiento de selección, por sí solo, no permite construir una solución óptima. ¿Qué otros pasos o, más bien, precauciones, debe tomar un algoritmo *greedy* buscando resolver un problema de optimización arbitrario, luego de ejecutar el procedimiento de selección "en el camino"?

Una vez elegimos algo que incluir a nuestra solución, debemos hacer dos chequeos:

- **Chequeo de factibilidad**: lo que sea que hayamos seleccionado no debe modificar lo que llevamos armado de la solución de forma que las restricciones del problema sean violadas. De lo contrario, nuestra solución se vuelve inválida.
- **Chequeo de solución**: si es seguro agregar lo seleccionado a nuestra solución, agreguémoslo. ¿Será que ya está completa la solución?

El acercamiento *greedy* evita el trabajo de resolver algunos subproblemas, por lo que produce algoritmos más eficientes que los de programación dinámica. Es por ello que algunos problemas se resuelven de esta forma, aunque no se alcance una solución óptima. Se tolera la falta de optimalidad a cambio obtener una solución rápidamente. Pero, dadas ciertas condiciones, *greedy* produce soluciones eficientes y óptimas.

La forma en la que *greedy* toma su decisión es comparando las alternativas actuales en función de cómo contribuyen a los objetivos de la solución. El problema debe tener la propiedad de que dicha forma de decidir resulte en una solución óptima. Esta propiedad es llamada **propiedad de la elección codiciosa** (***greedy-choice property***), y se puede expresar, en otras palabras, de la siguiente forma: la combinación de una decisión *greedy* y la solución óptima al subproblema que produce debería resultar en la solución óptima al problema general. La manera en la que un problema exhiba esta propiedad (y la forma en la que se demuestre) variará de problema en problema, pues depende del procedimiento de selección.

La subestructura óptima y la *greedy-choice property* son dos características que nos guían hacia el diseño de un algoritmo *greedy*, pero no hay un procedimiento universal con el que podamos demostrar que un algoritmo *greedy* produce un resultado óptimo. El fundamento de la metodología *greedy* es **heurístico**: no es un método formal de solución de problemas, sino una técnica deducida de forma empírica que ha funcionado para otros problemas.

Como primer ejemplo, consideremos el ***knapsack problem*** **fraccionado**. Este problema plantea lo siguiente: un caco entra conspicuamente a una tienda, con una bolsa vacía al hombro. La bolsa aguanta un peso máximo W. El caco se para frente a un anaquel con n artículos que piensa robar, y para cada artículo i el valor es v_i y el peso es w_i. Obviamente, el ladrón desea meter a su bolsa todos los artículos que aguante, de forma que salga de la tienda con el mayor valor acumulado posible. Esta versión del problema permite tomar porciones de un artículo.

2. El ladrón se siente observado y le agarra la perseguidora. ¿Qué artículo toma de primero? ¿Qué información y qué cálculos necesita para tomar esa decisión?

El ladrón tomaría de primero el artículo de mayor valor, pero para saber cuál es el artículo de mayor valor necesita conocer cuál es el que presenta la mayor proporción de valor por unidad de peso. Además, necesita saber cuánto espacio le queda en la bolsa para saber cuánto de dicho artículo puede tomar.

3. Demuestre que este problema presenta subestructura óptima. ¿Qué forma debe tener la solución al problema?

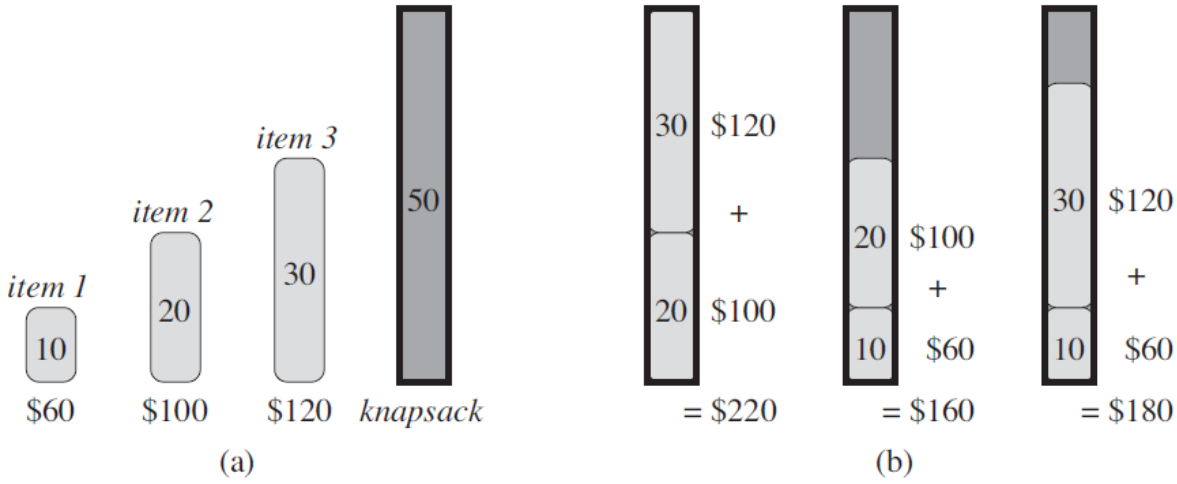
Necesitamos obtener una secuencia de elementos y cantidades de cada elemento, tal que su valor combinado sea el más alto posible para cualquier carga de peso W. Si visualizamos esta solución como $c_1, c_2, \dots, c_n$, donde c_i es la cantidad que robará del artículo i, sabemos que esta solución debe construirse a partir de soluciones óptimas a subproblemas. Si la pregunta original es cuánto robar de cada artículo, y la respuesta hipotética es $c_1, c_2, \dots, c_n$, cuando quitamos c_n de la solución queda la pregunta, ¿cuánto debemos robar de cada artículo de forma que, al agregarle c_n, obtengamos la máxima ganancia posible? Las decisiones son cuánto de qué artículo, de entre todo lo disponible, vamos a agregar a nuestra mochila, y el subproblema que queda es el de optimizar el espacio restante en la mochila con los artículos y cantidades restantes.

Con esto en cuenta, la necesaria optimalidad se manifiesta al pensar que, quitando c_n, podríamos elegir llenar el espacio restante $W_0 = W - c_n$ con una combinación diferente de cantidades por artículo, como $k_1, \dots, k_{n-1}$, que produzca una ganancia subóptima G_s para el peso W_0, y presentar la solución óptima como $k_1, \dots, k_{n-1}, c_n$. Si hacemos esto, la ganancia obtenida al agregar c_n sería $G_s + p(c_n)$, donde p es la función que calcula el valor de una cantidad de artículo dada. Esta ganancia sabemos que sería menor a $G_o + p(c_n)$, donde G_o es la ganancia óptima con la que ocupamos el peso W_0, porque $G_o > G_s$. Por tanto, se contradice la optimalidad de $k_1, \dots, k_{n-1}, c_n$, demostrando la necesaria optimalidad en las soluciones a los subproblemas.

4. Explique por qué este problema presenta la *greedy-choice property*.

El argumento es similar al de la necesaria optimalidad. La idea es que la demostración de necesaria optimalidad ahora se apoye en una elección codiciosa, que en nuestro caso es elegir el artículo con mayor valor por peso sin pensar en lo que tomará después. Sin embargo, la contradicción la buscamos sobre el artículo elegido. Es decir, ¿qué pasa si no tomamos la decisión *greedy*? En nuestro caso, ¿qué pasa si tomamos menos de lo que debemos del artículo de mayor valor, o si tomamos de otro artículo con menor valor? Significa que la cantidad c_n que habríamos tomado ahora es k_n, $0 \leq k_n < c_n$, requiriéndonos que el espacio $c_n - k_n$ restante sea ocupado con otro artículo (digamos, el artículo j) para alcanzar el peso W (de lo contrario, queda espacio vacío y la solución no es óptima). Es evidente que, como el artículo n tiene un mayor valor por peso que el artículo j, podríamos reemplazar el peso $c_n - k_n$ que tomamos del artículo j por el mismo peso del artículo n, lo que produciría una mayor ganancia que con el acercamiento de tomar una decisión distinta a la *greedy*.

Para comparar, consideremos la versión binaria del *knapsack problem*, a veces llamada ***0-1 knapsack problem***. En esta versión no se permite tomar porciones de un artículo. Se lleva el artículo completo o no se lleva. El siguiente ejemplo demuestra que esta versión del problema no presenta la *greedy-choice property*:



Para demostrar que el aceramiento *greedy* no garantiza la solución óptima a un problema basta un contraejemplo. Esto lo obtenemos fácilmente si, al tomar este acercamiento, la solución producida es inválida. Otra posibilidad es que la solución sea subóptima, lo que ocurriría en este caso si llenamos nuestra bolsa con los artículos incorrectos o si nos queda espacio vacío que se podría aprovechar mejor. Supongamos, para este problema, que el primer artículo tomado es el a_n, cuyo valor por peso es el más grande de todos. ¿Qué nos garantiza que podremos llenar con exactitud el espacio que queda con los artículos que quedan? En la versión fraccionada del problema esto está garantizado porque siempre podremos ajustar lo que tomemos a lo que nos cabe en la mochila. En esta versión del problema no podemos garantizar que cualquier espacio podrá llenarse con objetos de valor.

5. Describa la recurrencia que fundamenta un algoritmo de programación dinámica para resolver el *0-1 knapsack problem*. *Hint*: uno de los aspectos importantes a notar es que los subproblemas se definen en función de dos cosas: lo que tenemos disponible para meter a la mochila y el espacio que queda en la misma.

Esto sugiere que el arreglo donde guardaremos las soluciones a los subproblemas que vayamos resolviendo tendrá dos dimensiones en lugar de una. Por tanto, consideremos los artículos robables en un orden arbitrario, de modo que podemos enumerarlos como $a_1, \dots, a_n$. Supongamos que la solución óptima es el conjunto de artículos A. Sabemos, por la definición del problema, que cada artículo a_i estará o no estará en la solución óptima. Si a_i pertenece a A, sabemos que $W - w_i$ (donde W es el peso límite y w_i el peso del artículo a_i) será un peso límite que debemos optimizar con los $n - 1$ artículos que nos quedan. Si $a_i \notin A$ sabemos que, de todos modos, A debe contener la solución óptima formada a partir de los otros $n - 1$ artículos que no son a_i, aunque la diferencia es que dicha solución tiene la restricción de peso W en lugar de $W - w_i$. Así, lo que nos interesa es calcular la ganancia máxima que se obtendría en ambos casos, y por tanto proponemos la siguiente recurrencia:

$$P[i][w] = \max\{P[i-1][w], P[i-1][w-w_i] + p_i\}$$

Hay que considerar que algunas veces no podremos incluir un artículo porque no cabe en el espacio disponible. Por tanto, incluimos este caso en nuestra recurrencia de la siguiente forma:

$$P[i][w] = \begin{cases} \max\{P[i-1][w], P[i-1][w-w_i] + p_i\}, si\ w_i \leq w \\ P[i-1][w], si\ w_i > w \end{cases}$$

6. Suponga que, dada una selección de artículos para robar, consideramos el problema con peso límite W cada vez más grande. ¿Qué sucedería con el tiempo de ejecución del algoritmo que implementa la recurrencia de la pregunta anterior?

Conforme crece el límite de peso tenemos posibilidad de probar más combinaciones válidas de artículos en nuestra solución. Desde una perspectiva técnica, el arreglo de soluciones que llenaríamos incrementa el número de columnas, y por tanto incrementa en múltiplos de n la cantidad de casos que se tendrían que computar. Si $W \gg n$, este tiempo de ejecución podría ser peor que $O(2^n)$, que es correspondiente a la solución por fuerza bruta (*i.e.*, computar todas las posibles combinaciones y elegir la mejor).

El acercamiento para desarrollar el algoritmo es similar a lo que hemos realizado con otros algoritmos de programación dinámica. Si las filas del arreglo identifican al artículo que incluimos en nuestra solución óptima; y las columnas identifican limitantes de peso por unidad (*i.e.*, van de 0 unidades de peso a W unidades), computamos el valor de toda una fila, que efectivamente es probar incluir el elemento correspondiente a esa fila en la solución óptima para el peso límite correspondiente a cada columna. Luego cambiamos de fila y repetimos el proceso.

Para ilustrar la motivación de uso de algoritmos *greedy*, resolvamos el *0-1 knapsack problem* con los siguientes parámetros: $n = 3, W = 30$. Los pesos y precios de cada artículo son, respectivamente:

	a_1	a_2	a_3
w_i	5	10	20
p_i	50	60	140

Si procedemos como discutido, tendríamos un arreglo de 3×30 que llenar. Considerando que la fila y la columna 0 tendrán valores de 0 por defecto (ya que sin artículos o sin espacio no se puede robar nada), calcular la primera fila completa conllevaría probar robar el artículo 1 cuando nuestra bolsa tiene un peso límite de 1, 2, 3, etc.

Claramente, hasta la columna $w \geq 5$ tendremos valores diferentes a cero, ya que hasta en esas columnas tendremos espacio suficiente para el artículo a_1. A partir de esa columna, el valor de $P[1][w]$ será siempre 50, ya que no estaremos considerando incluir ningún otro artículo en la solución. Al considerar la segunda fila estaremos probando incluir el artículo a_2 en la solución óptima. La primera diferencia será que a partir de $w \geq 5$ las celdas de esa fila presentarán valores distintos a 0, ya que aunque no quepa el artículo a_2 sabemos que cabe el artículo a_1 y que, por tanto, hay una mejor forma de aprovechar el espacio disponible. Esto refleja la recurrencia en acción. Podemos apreciar un esbozo de este desarrollo en la siguiente tabla parcial:

	1	2	…	5	6	…	10	11	…	20	…
a_3	0	0	…	50	50	…	60	60	…	140	…
a_2	0	0	…	50	50	…	60	60	…	110	…
a_1	0	0	…	50	50	…	50	50	…	50	…

7. Considerando el ejemplo anterior, ¿cómo cambiaría el desarrollo de la solución si se emplea un acercamiento *top-down*? Estime el tiempo de ejecución para el *worst-case scenario* de este algoritmo.

Notemos que el acercamiento *top-down* irá calculando las subsoluciones que necesita conforme las necesita. Podemos observar que el caso raíz dependerá del resultado para dos subproblemas (aquel donde se incluyó determinado artículo y aquel donde no). Cada uno de estos resultados dependerá, a su vez, de dos subproblemas. Así, en el i-ésimo nivel del árbol calcularemos el valor de 2^i posibilidades, con $0 \leq i \leq n$, lo que nos da un tiempo de ejecución $\Omega(2^n)$.

Algunas soluciones de programación dinámica incurren en tiempos de ejecución peores que polinomiales. Es por esto que se justifica el empleo de soluciones *greedy* aunque esto no siempre produzca la mejor solución posible… al menos se produce más rápido. Sin embargo, hay algunos problemas para los que podemos garantizar que las soluciones *greedy* producirán la solución óptima. Ilustraremos esta categoría de problemas a continuación.

Considérese un grafo no dirigido. Dicho grafo es llamado un **árbol** si es acíclico, **conexo** (*i.e.*, se puede llegar a cualquier vértice partiendo de cualquier vértice) y **simple** (*i.e.*, que no tiene bucles ni aristas paralelas). Si un grafo no dirigido cualquiera presenta un árbol que toca a todos sus vértices, el árbol es llamado **abarcador** (mi traducción) o ***spanning tree***. El ***minimum-spanning-tree problem*** toma un grafo **ponderado** (*i.e.*, existe una función de peso que asigna números positivos a las aristas), conexo y no dirigido como entrada, y produce un *spanning tree* con el menor peso posible.

Supongamos, entonces, un grafo ponderado, conexo y no dirigido $G(V, E)$ donde V es el conjunto de vértices y E son las aristas. Luego, nombremos I a un conjunto vacío de árboles en el grafo (es decir, un **bosque** vacío). Puesto que el grafo ya es conexo y no dirigido, cualquier subconjunto de sus aristas que sea acíclico será un bosque, *i.e.*, un conjunto de árboles posiblemente disconexo. El *minimum-spanning-tree problem* sobre este grafo se puede resolver si ordenamos las aristas ascendentemente por peso y, recorriendo este orden, vamos armando un conjunto con cada arista que encontremos que se pueda incluir sin que el conjunto presente ciclos. Éste es un acercamiento *greedy* porque para construir el *minimum spanning tree* debemos decidir repetidamente qué arista incluir, y el método descrito siempre toma la arista que no forme ciclos y que tenga el menor peso posible. Eventualmente se hallará el árbol abarcador de menor peso o, si el "árbol" no es conexo, el bosque abarcador de menor peso. Este es el **algoritmo de Kruskal**.

Consideremos un problema más sencillo. Imaginemos una página *web* para encontrar hospedaje, donde los lugares tienen una calificación calculada a partir de retroalimentación de antiguos clientes. La página desea proveer una funcionalidad que permita al usuario determinar un número k y obtener los k lugares de hospedaje cuya suma de calificaciones sea la más alta. En otras palabras, este problema desea el conjunto de k lugares con calificaciones más altas (pues esto resultará en la suma más alta) de entre todos los disponibles.

8. Describa brevemente un algoritmo para resolver el problema de búsqueda de hospedajes recién planteado.

La solución a este problema es inmediata: ordenemos todos los lugares descendentemente por calificación y tomemos los primeros k elementos.

Nótese que, para armar el conjunto resultante, cada paso que tomamos conforma la decisión de qué lugar agregar, y en cada paso la decisión es el lugar con calificación más alta entre los restantes (y sin superar la cantidad de lugares solicitada). Es decir, se trata de un algoritmo *greedy*.

¿Qué similitud tienen estos problemas, que los hace solucionables con un algoritmo *greedy* tan similar? Es la estructura. Ambos problemas se pueden modelar como una **matroide ponderada**, que es una pareja $M = (S, I)$ donde S es un conjunto no vacío para el cual hay una función w que asigna valores (**pesos**) positivos a cada elemento; e I es una familia de subconjuntos de S, llamados subconjuntos **independientes**. El conjunto I debe cumplir con las siguientes características:

1. I es **hereditario**, que significa que todo subconjunto de un conjunto independiente es independiente. En símbolos, si $A \in I$, $B \subset A \Rightarrow B \in I$.
2. M cumple con la **propiedad de intercambio**. Esta propiedad dice que si B es un subconjunto independiente más grande que otro independiente A, existe algún elemento x de $(B - A)$ que podemos transferir a A de modo que $A \cup \{x\}$ sea independiente. En símbolos, si $A, B \in I$ tales que $|B| > |A|$ entonces $\exists x \in (B - A)$ tal que $(A \cup \{x\}) \in I$.

En el *minimum-spanning-tree problem*, las aristas del grafo son S y el bosque inicialmente vacío es I, los subconjuntos acíclicos de aristas. En el problema de los hospedajes, el conjunto de todos los n lugares es S y, para un $0 \leq k \leq n$, todos los subconjuntos con k lugares o menos son independientes.

A continuación, veremos y justificaremos el algoritmo *greedy* que garantiza una solución óptima para problemas que se pueden modelar como matroides ponderadas.

```
GREEDY(M, w)
1  A = ∅
2  sort M.S into monotonically decreasing order by weight w
3  for each x ∈ M.S, taken in monotonically decreasing order by weight w(x)
4      if A ∪ {x} ∈ M.I
5          A = A ∪ {x}
6  return A
```

Este algoritmo provee un subconjunto independiente **óptimo**, que es el que tiene el máximo peso combinado posible. En una matroide ponderada, un subconjunto independiente óptimo es necesariamente **maximal**, que significa que no es subconjunto de ningún otro independiente. Esto sucede porque los pesos son siempre positivos y mientras más elementos incluyamos tendremos más peso total. La decisión *greedy* se manifiesta al incluir el elemento con el mayor peso posible cada vez, donde la posibilidad de inclusión se determina por las condiciones de independencia.

La familia de subconjuntos independientes identifica los subconjuntos que cumplen con las restricciones que nos interesan para nuestro problema. Si sólo se tratara de tomar los elementos con el mayor peso posible, la solución sería inmediata. Pero lo normal es que debamos elegir estos elementos sujetos a condiciones específicas, como aristas que formen un árbol o conjuntos de sitios de hospedaje que no superen el tamaño k dado. Para el *minimum-spanning-tree problem* no podemos proponer como solución a un conjunto de aristas que no formen un árbol. Sabemos que cualquier subárbol de un árbol también es un árbol, incluso cuando consta de una única arista. Entonces sabemos que cualquier *minimum spanning tree* se forma de aristas que son, por sí solas, un árbol. El algoritmo *greedy* irá eligiendo arista por arista, comenzando por la de menor peso que forme un árbol unitario. Esto se manifiesta formalmente en las matroides.

9. Demuestre que, dado cualquier conjunto independiente $A \in I$, cada elemento $x \in A$ forma un conjunto unitario independiente.

Como todo independiente pertenece a I e I es hereditario, si $x \in A \in I \Rightarrow \{x\} \in I$. Esto, en otras palabras, dice que ningún independiente podrá contener elementos que no forman, por su propia cuenta, subconjuntos independientes unitarios, lo que justifica la condición en el algoritmo bajo la que se incluye $\{x\}$ en A.

A continuación, se demostrará que en las matroides ponderadas la decisión *greedy* siempre es la correcta para armar un subconjunto independiente óptimo (es decir, se demostrará que las matroides ponderadas cumplen con la *greedy choice property*).

Al ordenar los elementos de S descendientemente por peso buscaremos el primer elemento en ese orden con el que se pueda armar un subconjunto independiente. Este elemento, forzosamente, pertenecerá a un subconjunto independiente óptimo. Supongamos que existe un subconjunto independiente óptimo B que no incluye a x, el primer elemento en el orden descendiente tal que $A = \{x\}$ es independiente. Por el orden que hicimos, ningún elemento $y \in B$ tendrá peso mayor al de x. Luego, aprovechamos la propiedad de intercambio para pasar, uno por uno, los elementos de B hacia A hasta que A tenga el mismo tamaño que B.

La misma propiedad de intercambio asegura que, para ser óptimo, A no podrá ser más grande que B, porque todos los independientes maximales deben tener el mismo tamaño. Como A y B tienen el mismo tamaño, pero B tiene un elemento y en lugar de x tal que $w(y) \leq w(x)$, entonces $w(B) = w(A) - w(x) + w(y) \leq w(A)$. Sin embargo, $w(A) \leq w(B)$ porque supusimos que B era óptimo, entonces $w(A) = w(B)$, lo que hace a A un independiente óptimo que incluye a x.

10. Demuestre que todos los conjuntos independientes maximales tienen el mismo tamaño.

Supongamos que $A, B \in I$ son conjuntos independientes maximales tales que $|B| > |A|$. Entonces existen elementos $x_1, \ldots, x_n \in B$ tales que $B - A = \{x_1, \ldots, x_n\}$. Aplicando la propiedad de intercambio podemos formar $A^* = A \cup \{x_i\}$ con el x_i que lo permita. De este modo, tenemos $A \subset A^*$ y $A, A^* \in I$, lo que contradice la maximalidad de A. Siempre podremos agregar elementos de un conjunto independiente maximal a cualquier conjunto independiente no maximal, excepto cuando ambos conjuntos sean de igual tamaño.

Ahora debemos recalcar que cualquier problema que podamos modelar como una matroide ponderada se podrá resolver óptimamente con un algoritmo *greedy*. Ya vimos que esto es posible porque las matroides ponderadas exhiben la *greedy choice property*, pero nos falta la otra propiedad de problemas a los que aplica el acercamiento *greedy*: subestructura óptima.

11. Demuestre que las matroides ponderadas exhiben subestructura óptima.

La decisión que se toma en cada paso del algoritmo es cuál elemento agregar al independiente óptimo resultante. Si agregamos el elemento x, esta decisión produce un subproblema: una matroide $M' = (S', I')$ donde S' se conforma de elementos de S que pueden unirse a $\{x\}$ preservando la independencia; y donde cada subconjunto $B \in I'$ debe preservar la independencia si se le agrega x. En símbolos, estas definiciones se ven así:

$$S' = \{y \in S : \{x\} \cup \{y\} \in I\}$$

$$I' = \{B \subset S - \{x\} : B \cup \{x\} \in I\}$$

Observamos que si A es un independiente óptimo de la matroide original M entonces $A' = A - \{x\}$ es un independiente de M'. Por tanto, $w(A) = w(A') + w(\{x\})$. En esta fórmula, maximizar el valor de $w(A)$ requiere maximizar el valor de $w(A')$ porque no podemos manipular el valor de $w(\{x\})$, y x no nos lo quitamos de encima porque sabemos que, al haber sido elegido, debe ir en el resultado final. Entonces, maximizar el peso de un independiente en la matroide original requiere maximizar el peso de un independiente en la matroide subproblema, lo que describe una subestructura óptima.

Para concretar el concepto de matroide veremos un último ejemplo a detalle. Se trata del **problema de calendarización de tareas unitarias**. Este problema se ilustra con un procesador que trabaja por unidades de tiempo (como ciclos de reloj) y un conjunto S de tareas que toman una unidad de tiempo cada una.

Una **calendarización** será cualquier permutación de S cuyo orden indique la secuencia en la que se realizarán las tareas. Cada tarea tiene una ***deadline*** o **tiempo límite** al cabo del cual tendrá que haberse completado para no incurrir en una penalización, donde las penalizaciones son definidas por tarea. El objetivo para resolver el problema es proveer una calendarización que minimice la penalización total.

Para formalizar el problema, sea $S = \{a_1, \dots, a_n\}$ el conjunto de tareas unitarias, $\{d_1, \dots, d_n\}$ el conjunto de tiempos límite para cada tarea (d_i es el tiempo límite para la tarea a_i) y $\{w_1, \dots, w_n\}$ el conjunto de penalizaciones por tarea, con la misma correspondencia que las *deadlines*. Además, una calendarización en **forma canónica** será una calendarización donde las tareas que terminan temprano (llamémosles "tempraneras") preceden a las que terminan tarde (llamémosles "tardías") y las tareas tempraneras están ordenadas ascendentemente por *deadline*.

Cualquier calendarización se puede transformar a forma canónica, ya que anticipar las tareas tempraneras a las tardías sólo las hace terminar todavía más temprano. Además, si en la calendarización hay dos tareas tempraneras consecutivas a_i, a_j cuyas *deadlines* cumplen $d_j \leq d_i$, intercambiarlas en la calendarización no provocará penalizaciones porque a_j terminará aún más temprano y, como a_j era tempranera, el nuevo tiempo de finalización de a_i será $\leq d_j \leq d_i$. Un conjunto de tareas se considerará independiente si todas sus tareas se pueden calendarizar como tempraneras.

Para identificar un conjunto de tareas independiente podemos emplear un sencillo criterio: no debe haber más tareas que tiempo. Es decir que no contiene más que una tarea con *deadline* ≤ 1, no hay más que dos tareas con *deadline* ≤ 2, etc. Si expresamos con $N_t(A)$ el número de tareas de A cuya *deadline* es t o menor, el criterio se expresa así: para $t = 0,1, \dots, n$, $N_t(A) \leq t$. Debemos demostrar que, en efecto, esta definición de conjunto independiente forma una matroide.

Claramente, cualquier subconjunto de un conjunto de tareas tempraneras también contendrá solo tareas tempraneras, lo que asegura que I es hereditario. Luego, consideremos dos conjuntos independientes de tareas, A, B, tales que $|B| > |A|$. Para que ocurra esa diferencia de tamaño podría darse que $N_t(B) > N_t(A)$ para todo $t = 0,1, \dots n$. De lo contrario, habrá algún valor máximo $k < n$ tal que $N_i(B) \leq N_i(A)$ para $i = 0, \dots, k$, lo que implica que para $i = k + 1, \dots, n$ se cumplirá $N_i(B) > N_i(A)$. En otras palabras, A puede tener más tareas, con *deadlines* de k o menos, que B, pero no puede ocurrir esto para todas las *deadlines* porque entonces A sería más grande que B.

12. ¿Por qué necesitamos demostrar la propiedad de intercambio con un elemento de B cuya *deadline* sea mayor o igual a $k + 1$?

Esto nos servirá porque $A \cup \{b\}$ podría no ser independiente si tomamos una tarea $b \in B$ con *deadline* $c \leq k$, considerando el caso en que $N_c(A) = c$. Para evitar estos casos, demostraremos la propiedad de intercambio usando tareas de B con *deadlines* posteriores a k.

Entonces, sea $\beta \in (B - A)$ una tarea con *deadline* $k + 1$. En el conjunto $A' = A \cup \{\beta\}$ tendremos que $N_j(A') = N_j(A) \leq j$ para $j = 0, \dots, k$, partiendo de que A es originalmente independiente. Por el intercambio realizado ya no podemos asegurar que $N_j(A')$ sea estrictamente menor a $N_j(B)$ para $j = k + 1, \dots, n$, pero sí podemos asegurar que $N_j(A') \leq N_j(B)$. Como B es independiente, $N_j(B) \leq j$ para todo $j = 0, \dots, n$, con lo que concluimos que $N_j(A') \leq j$ para todo $j = 0, \dots, n$, lo que hace a A' independiente.

Con esta demostración podemos concluir que este modelo cumple con la propiedad de intercambio, por lo que el conjunto de tareas y esta definición de independencia de subconjuntos forman una matroide.

13. Con el algoritmo *greedy* para matroides ponderadas, provea la calendarización óptima (que minimice las penalizaciones incurridas) para las tareas con las siguientes propiedades:

Tarea	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Deadline	**3**	**1**	**4**	**2**	**4**	**4**	**6**
Penalización	**40**	**30**	**50**	**60**	**70**	**20**	**10**

Como sólo cuenta la penalización de las tareas tardías, lo que buscamos es una calendarización donde las tareas tempraneras tengan la máxima penalización. Tendríamos que ordenar el conjunto de tareas descendientemente por penalización. Entonces, nuestro algoritmo *greedy* construirá un conjunto independiente de tareas con la máxima penalización posible. A ese conjunto se le agregan las tareas tardías y se le convierte a forma canónica. El resultado es la siguiente calendarización: $(a_4, a_1, a_5, a_3, a_7, a_2, a_6)$.

Ejercicios

1. Veamos un nuevo problema de calendarización. Tenemos un conjunto de tareas $S = \{a_1, \dots, a_n\}$ donde la tarea a_i tarda en realizarse p_i de unidades de tiempo. Dependiendo de cómo calendaricemos su ejecución, cada tarea tendrá un tiempo de terminación c_i (que es cuando la tarea se completa). Queremos hacer una calendarización *non-preemptive* (*i.e.*, una tarea iniciada no se puede interrumpir) que minimice el promedio de todos los tiempos de terminación.
    a. Proponga un algoritmo *greedy* que provea esta calendarización.
    b. Determine el tiempo de ejecución de su algoritmo.
    c. Demuestre que su algoritmo da la solución óptima (es decir, demuestre que el problema presenta las características que permiten obtener una solución óptima mediante un algoritmo *greedy* o, alternativamente, modele el problema como una matroide ponderada).
2. 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.
3. En una matroide $M = (S, I)$, donde S tiene n elementos, ¿qué puede asegurar sobre el tiempo de ejecución del algoritmo *greedy* correspondiente?
4. Usted es catedrátic@ de un curso libre en la UVG, y en su curso hay estudiantes de muchas carreras diferentes (cada estudiante, suponemos, perteneciente a una única carrera). La decanatura le ha solicitado seleccionar un equipo de estudiantes para enviar a una olimpiada de ciencias, con el requerimiento de no enviar a más de un estudiante de cada carrera. Los estudiantes participarán como equipo, y usted ha determinado que el equipo tendrá mejores posibilidades de ganar conforme más alta sea la suma de promedios de todos los miembros.
    a. Demuestre que este problema se puede modelar como una matroide ponderada y provea el algoritmo *greedy* que lo resuelve.

b. Desarrolle una instancia de este problema y úsela para ilustrar la aplicación de su algoritmo.

5. El **algoritmo de Dijkstra** resuelve el problema del camino más corto entre dos vértices sobre un grafo ponderado positivamente y dirigido. Lo hace con un acercamiento *greedy*, mientras que el algoritmo de Bellman-Ford, discutido en los ejercicios del tema anterior, resuelve el mismo problema aplicando programación dinámica. Provea y compare (dicho de otra forma, investigue y discuta), con notación asintótica, cotas superiores a las tasas de crecimiento de los algoritmos de Dijkstra y Bellman-Ford. Suponiendo que ambos algoritmos se enfrentan a la misma instancia del problema, ¿cuál es mejor? Si no necesariamente se enfrentan a la misma instancia, ¿cuál es mejor?

Fuentes:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Massachusetts: The MIT Press.
- Erickson, J. (2014). *Matroids.* Retrieved from Algorithms, Etc.: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/08-matroids.pdf
- Soltys, M. (2012). *An Introduction to the Analysis of Algorithms.* Singapore: World Scientific Publishing.