Actas de las Jenui, vol. 8. 2023.
Páginas: 407-414

Reflexiones sobre la impartición de asignaturas de algoritmos

J. Ángel Velázquez Iturbide
Departamento de Informática y Estadística
Universidad Rey Juan Carlos
28933 Móstoles, Madrid
angel.velazquez@urjc.es

Resumen

Ha habido más debate sobre la enseñanza de la introducción a la programación que sobre la enseñanza de cualquier otra materia informática. Sin embargo, es aconsejable, incluso necesario, debatir sobre la docencia de esta materia. Esta comunicación es una reflexión sobre la docencia de la algoritmia, abordando tres temas. Primero, se resalta que los problemas resueltos con varias técnicas destacadas de diseño son de optimización, comentando algunas implicaciones de este hecho. Segundo, se aboga por un enfoque experimental para el aprendizaje de los algoritmos, complementario del más extendido enfoque formal. En concreto, mostramos cómo pueden usarse la visualización y la comparación experimental para que los algoritmos sean más concretos para los alumnos. Tercero, se argumenta que algunos modelos conceptuales presentes en la mayoría de los libros de texto sobre algoritmos son imprecisos, dificultando el aprendizaje de los temas correspondientes. Analizamos esta cuestión en tres técnicas de diseño de algoritmos: la técnica voraz, programación dinámica y ramificación y poda.

Abstract

There has been more debate about the teaching of introductory programming than the teaching of any other informatics matter. However, it is advisable, even necessary, to discuss how to teach this subject matter. This position paper meditates on this concern by addressing three themes. Firstly, it is remarked that problems addressed by several common algorithm design techniques are optimization problems, and some implications are discussed. Secondly, it is advocated an experiential approach to learning algorithms, which would complement the more usual formal approach. In particular, we show how visualization and benchmarking can be used to make algorithms more concrete to students. Thirdly, it is argued that some conceptual models present in most algorithm textbooks are imprecise, making difficult to learn their corresponding topics. We elaborate on this concern for three design techniques, namely greedy algorithms, dynamic programming and branch-and-bound.

Palabras clave

Enseñanza de la algoritmia, problemas de optimización, visualización, medidas experimentales, modelos conceptuales, algoritmos voraces, programación dinámica, ramificación y poda.

1. Introducción

La investigación en la educación informática ha madurado como campo de investigación en las pasadas décadas [7]. La investigación ha abordado tanto áreas específicas de la informática como cuestiones transversales. Entre las áreas, la introducción a la programación es la que ha recibido mayor atención [13]. Dicha atención se debe al papel central de la programación dentro de la informática [1] y a que es una de las primeras asignaturas de cualquier grado en informática (o la primera en el caso de los EEUU, donde se conoce por las siglas CS1).

La educación de otras áreas informáticas no ha recibido tanta atención. Sean, por ejemplo, los algoritmos [14]. Ciertamente, hay numerosas y variadas contribuciones en este área, pero no tan informativas o tan consolidadas como en CS1. Por ejemplo, un hito en la investigación sobre la enseñanza de CS1 fue comprobar que las dificultades de los alumnos eran universales, no limitadas de ningún país, institución o profesor. Este hallazgo dio lugar a diferentes iniciativas, desde argumentar que CS1 debía tener unos objetivos de aprendizaje menos ambiciosos (p.ej., [12]) a métodos didácticos basados en que los aprendices construyan su conocimiento incrementalmente (p.ej. [11]).

Este trabajo se ha financiado con el proyecto de investigación e-Madrid-CM (S2018/TCS-4307) de la Comunidad Autónoma de Madrid y los proyectos-puente PROGRAMA de la Universidad Juan Carlos (M2614 y M3035). El proyecto e-Madrid-CM también está financiado con los fondos estructurales FSE y FEDER.

Esta comunicación de la categoría de reflexión se centra en la enseñanza de los algoritmos, basándose en nuestra experiencia de investigación y de docencia de la materia. Presentamos tres cuestiones que consideramos relevantes. Se resumen aquí y posteriormente se desarrollan en las tres secciones siguientes.

En primer lugar, una forma ampliamente aceptada de enfocar la docencia de los algoritmos es alrededor de técnicas de diseño. Puede considerarse un enfoque ingenieril, ya que se presentan a los alumnos esquemas de código o metodologías de amplia aplicabilidad. Ahora bien, muchas técnicas de diseño resuelven exclusivamente problemas de optimización: algoritmos voraces, algoritmos heurísticos y aproximados, programación dinámica, y ramificación y poda. Esta observación conlleva varios corolarios educativos.

En segundo lugar, si las asignaturas de algoritmos se inspiran en los principales libros de texto, tienen un fuerte componente matemático. Siendo necesario este énfasis, argumentamos que los alumnos deberían tener una mayor experiencia directa con los algoritmos que actualmente. Nos centramos en dos formas experienciales: la visualización y la comparación experimental. Tener estas experiencias constituye una forma más suave de abordar diversos temas y proporciona a los alumnos una experiencia de aprendizaje más vívida, contribuyendo así a un aprendizaje más profundo.

Finalmente, los modelos conceptuales [17] de varias cuestiones algorítmicas no están tan maduros como en otras áreas. En concreto, argumentamos que algunos modelos conceptuales ofrecen poco o ningún detalle. Lo ilustramos con tres técnicas de diseño de algoritmos: la técnica voraz, programación dinámica, y ramificación y poda.

2. Problemas de optimización

Como se comentó en la introducción, muchas de las técnicas de diseño más usadas tratan problemas de optimización: algoritmos voraces, algoritmos heurísticos y aproximados, programación dinámica, ramificación y poda, y (parcialmente) vuelta atrás. Esta observación tiene varias consecuencias.

En primer lugar, deberían detallarse las características de los problemas de optimización. En concreto, la postcondición de estos problemas consta de dos partes diferenciadas: la condición de validez y la función objetivo. Hemos revisado varios libros de texto destacados y ninguno hace mención explícita a este hecho, salvo Baase y van Gelder [2], sección 13.4 sobre algoritmos de aproximación. En general, los libros de texto solamente presentan la posibilidad de hallar soluciones "cercanas a la optimal" [2], sin explicar cómo encaja esta afirmación con las nociones convencionales de problema y de corrección de un algoritmo.

Una segunda observación, relacionada con la anterior, es que la distinción entre condición de validez y función objetivo tiene una implicación importante sobre la corrección de los algoritmos. Los problemas resueltos en asignaturas introductorias a la programación o a la algoritmia, como ordenar un vector o determinar si un número natural es primo, suele tener una única solución válida. Esta situación se relaja en los problemas combinatorios, como el problema de las *n* reinas, que en general admiten varias soluciones válidas. La situación se vuelve aún más compleja en los problemas de optimización, donde debemos distinguir entre soluciones válidas y optimales. Una solución es válida si satisface la condición de validez y es optimal si además optimiza el valor expresado por la función objetivo. En general, hay varias soluciones, válidas y optimales, siendo éstas últimas un subconjunto de las primeras. Por tanto, el concepto de corrección algorítmica cambia, lo cual debería hacerse explícito a los alumnos.

En tercer lugar, el planteamiento y resolución de problemas de optimización debería hacerse sobre el conocimiento de algunos preliminares matemáticos, como las relaciones de orden, las operaciones max y min y sus propiedades, y optimal vs. óptimo. No son muchos conceptos, por lo cual resulta sorprendente que no se incluyan en los prerrequisitos matemáticos de los Computing Curricula [1]. Algunos libros de texto contienen una introducción o un anexo con preliminares matemáticos para la algoritmia [4, 5], que incluyen series, sumatorios y límites, enumeración, etc. Sin embargo, no se incluyen los conceptos de orden citados, salvo las cotas.

3. Experimentando con los algoritmos

La explicación de un algoritmo en los libros de texto suele presentarse de forma abstracta, aunque probablemente ilustrada con algún ejemplo. Mostramos dos formas de proporcionar a los alumnos experiencia con el comportamiento o rendimiento de los algoritmos, lo que facilita su comprensión: la visualización y la comparación experimental.

3.1. Visualización

La animación de algoritmos es una forma conocida de ayudar a la comprensión de los algoritmos, consistente en mostrar su comportamiento, sin que sus visualizaciones correspondan a una codificación concreta de los mismos. Fue un área de investigación intensiva a partir de los años 80 [20], con un impulso educativo en la década del 2000 [16]. De hecho, las ilustraciones son un recurso frecuente en los libros de texto, que no sólo se usan para mostrar el comportamiento de un algoritmo con unos datos concretos, sino también para mostrar su comportamiento de forma genérica, para aclarar el enunciado del problema o incluso para analizar la eficiencia de un algoritmo.

La visualización de programas es una tecnología distinta pero relacionada con la animación de algoritmos, que genera vistas del programa o de sus estructuras de datos a partir del código fuente. Aunque pueden ser menos expresivas y tener menor nivel de abstracción que las animaciones de algoritmos, los sistemas de visualización de programas tienen la ventaja de generar automáticamente visualizaciones para cualquier algoritmo (dentro del ámbito del sistema). Por tanto, pueden ser usados por profesores y alumnos para la exploración libre de sus propios algoritmos, en el primer caso para la enseñanza, en el segundo para autoestudio o para resolver prácticas.

La Figura 1 muestra un ejemplo de una visualización de programa para el problema de la subsecuencia común más larga (SCML) [5]. Por falta de espacio, no explicamos el problema aquí, que es frecuente en el capítulo de los libros de texto dedicado a la programación dinámica. La figura muestra el árbol de recursión obtenido tras implementar las ecuaciones recursivas "hacia adelante" que resuelven el problema, y usarlas para determinar la SCML de las secuencias "aaaa" y "bba". La figura muestra una interfaz de vista global + detalle [27], en la que la parte inferior presenta el árbol completo a tamaño reducido mientras la parte superior sólo muestra una parte del árbol (enmarcada en la vista global) pero a un tamañor, que permite su visión. El árbol de recursión muestra un estado cercano a la terminación del algoritmo, en el que la llamada activa está enmarcada en verde (a la derecha), las llamadas pendientes están enmarcadas en azul claro y las llamadas terminadas están difuminadas.

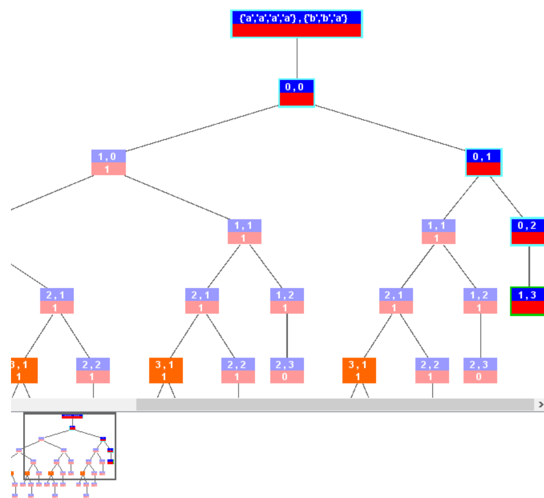

Figura 1: Árbol de recursión generado durante el cálculo de SCML(aaaa,bba).

La figura permite ilustrar la potencia de la visualización de programas para distintos propósitos. El sistema se configuró para que los nodos del árbol tuvieran distintos colores. Hemos sobrecargado la visualización para ilustrar simultáneamente varios usos:

- Comportamiento de los algoritmos. Es el uso más frecuente de las visualizaciones. El usuario puede generar libremente la visualización correspondiente a unos datos de entrada válidos, navegar por la animación, e interactuar con la visualización. Existen diversas variantes de este esquema básico, como mostrar "múltiples vistas" del estado del cómputo [20].
- Complejidad en tiempo. Los árboles de recursión ayudan a hacer visible el crecimiento del tiempo de ejecución. Por ejemplo, sea el algoritmo recursivo para el problema de la SCML, de complejidad exponencial. La Figura 1 muestra un árbol de recursión con 38 nodos generado para dos secuencias de longitudes respectivas 4 y 3. Si incrementamos la longitud de cada secuencia en uno, el tamaño del árbol aumenta a 96 (esta cifra puede variar, ya que la complejidad del algoritmo no sólo depende de la longitud de las secuencias sino también de su valor).
- Complejidad en espacio. El árbol de recursión también permite ilustrar la reserva y liberación de memoria durante el proceso recursivo. Una diferencia clave entre la complejidad en tiempo y en espacio es que el tiempo es acumulativo, mientras que el espacio se reutiliza. Por tanto, la complejidad en tiempo es proporcional al número de nodos del árbol de recursión, mientras que la complejidad en espacio es proporcional a la longitud de la rama más larga del árbol. Esta diferencia se ha resaltado en la visualización generada al mostrar difuminadas las llamadas terminadas, cuya memoria ha liberado y reutilizado el sistema operativo. En realidad, en cada momento sólo está reservada la memoria correspondiente a las llamadas que hay entre la raíz del árbol y el nodo activo (a la derecha de la figura).
- Redundancia. La clase de los algoritmos recursivos múltiples diseñados para resolver problemas de optimización mediante programación dinámica siempre son redundantes, es decir, muchas llamadas recursivas se calculan repetidamente. En la figura, se han coloreado en marrón claro las llamadas con parámetros i=3, j=1. La visualización permite determinar el número total de nodos y de nodos distintos, así como identificar los nodos redundantes.

Aunque es menos frecuente, la visualización de programas también pueden usarse para ayudar a la mejora de algoritmos [28]. Cualquier algoritmo recursivo puede optimizarse si se elimina su redundancia. Un método bien conocido [3] comienza transformando el árbol de recursión en un grafo de dependencia (es decir, un grafo dirigido acíclico) mediante la fusión de los nodos iguales en un único nodo, preservando sus arcos. Si se hacen corresponder los nodos del grafo de dependencia con las celdas de una tabla, obtenemos

tanto una representación visual de la tabla a usar en el algoritmo de programación dinámica como las dependencias de cálculo que debe respetar dicho algoritmo.

La Figura 2 muestra el resultado de hacer corresponder los nodos de la Figura 1 con celdas de una tabla bidimensional, asociando los valores del primer parámetro con columnas y los del segundo parámetro con filas. Obsérvese que solamente hay 18 nodos distintos para las 38 llamadas mostradas en el árbol de recursión de la Figura 1. Es sencillo desarrollar un algoritmo iterativo que calcule los valores mostrados, respetando las dependencias [3]. Para este ejemplo, puede consistir en dos bucles anidados que calcule los valores a almacenar en las celdas de la tabla por columnas de derecha a izquierda y, en cada columna, de abajo a arriba.

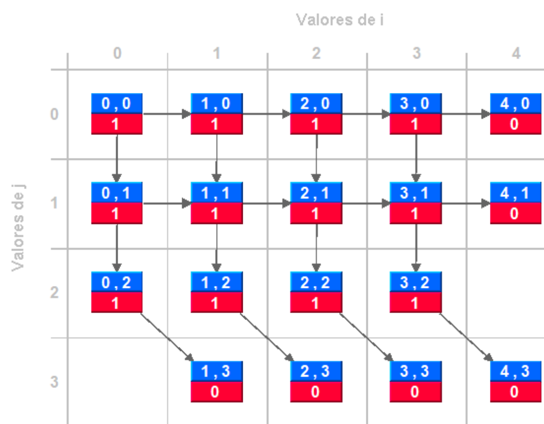

Figura 2: Grafo de dependencia derivado de la Figura 1 y dispuesto sobre una matriz.

3.2. Comparación experimental

Otra forma de tener una experiencia vívida con un algoritmo consiste en experimentar con su comportamiento mediante la recogida de medidas de su rendimiento. Mejor aún, pueden recogerse estas medidas para algoritmos distintos que resuelven un mismo problema, comparándolas entre sí y mostrando claramente las diferencias de rendimiento entre los algoritmos. Obviamente, conviene disponer de un sistema que ayude interactivamente en las tareas auxiliares de generación de juegos de datos, ejecución, almacenamiento, comparación y exportación de resultados [24].

La comparación experimental (*benchmarking*) se ha propuesto para comparar los tiempos de ejecución de distintos algoritmos [15], con frecuencia de ordenación. La comparación experimental también puede realizarse con otras medidas [15, 24], en concreto con la calidad de la solución en problemas de optimización. Recordemos que un algoritmo exacto para un problema de optimización siempre produce una solución optimal para datos de entrada válidos, mientras que un algoritmo inexacto puede producir soluciones suboptimales. Los algoritmos de programación dinámica, de vuelta atrás y de ramificación y poda son exactos por construcción, mientras que los algoritmos heurísticos y aproximados son inexactos.

Comparar experimentalmente algoritmos exactos e inexactos proporciona una experiencia sobre su rendimiento. Si nos centramos en los algoritmos heurísticos, sabemos que sus soluciones no están acotadas con respecto a las soluciones optimales. Sin embargo, ¿con que frecuencia calculan resultados suboptimales?, ¿cuál es la desviación media de sus resultados suboptimales? Incluso para algoritmos aproximados, conocemos la desviación máxima respecto a los resultados optimales, pero no su frecuencia ni su desviación media. Comparar experimentalmente algoritmos exactos e inexactos proporciona una evidencia empírica [25] de la calidad de los algoritmos en observación, más vívida que una demostración formal de la existencia de una cota para la calidad de sus soluciones.

Por ejemplo, sea la versión de minimización del problema de empaquetado de objetos (o llenado de cajas) [2, 4, 8, 19], y cuatro algoritmos que lo resuelven: dos algoritmos aproximados (*first-fit* y *next-fit*), un algoritmo de vuelta atrás y otro de ramificación y poda. Supongamos que generamos aleatoriamente 100 juegos de datos con 12 objetos, de pesos comprendidos entre 1 y 7, y cajas de capacidad comprendida entre 7 y 20. La Figura 3 resume [24] el resultado de ejecutar estos cuatro algoritmos sobre estos juegos de datos y comparar la calidad de sus resultados (mostrados de izquierda a derecha en el mismo orden en que se mencionaron).

La Figura 3(a) muestra que el algoritmo *first-fit* calcula resultados minimales en el 55% de los casos, que el *next-fit* lo hace en el 99% de los casos y que los algoritmos de vuelta atrás y de ramificación y poda calculan resultados minimales en todas las ejecuciones (como se esperaba). La Figura 3(b) muestra la desviación de las soluciones suboptimales. El primer algoritmo aproximado produce menos desviación que el segundo en la mayoría de los casos, pero su desviación máxima es mayor (50% frente a 33’33% por encima de la solución optimal). Obviamente, los resultados varían para distintos tamaños y rangos de valores de los datos de entrada, pero con la misma tendencia.

La comparación experimental es más completa si también medimos y comparamos tiempos de ejecución. Si hacemos esto con los mismos datos, obtenemos los resultados de la Figura 4 (presentamos los resultados en formato numérico en lugar de gráfico porque difieren en varios órdenes de magnitud). Los resultados se presentan con los algoritmos en el mismo orden que en la Figura 3. Puede verse que los algoritmos aproximados son los más rápidos, pero también que el algoritmo de ramificación y poda mejora drásticamente la eficiencia del algoritmo de vuelta atrás.

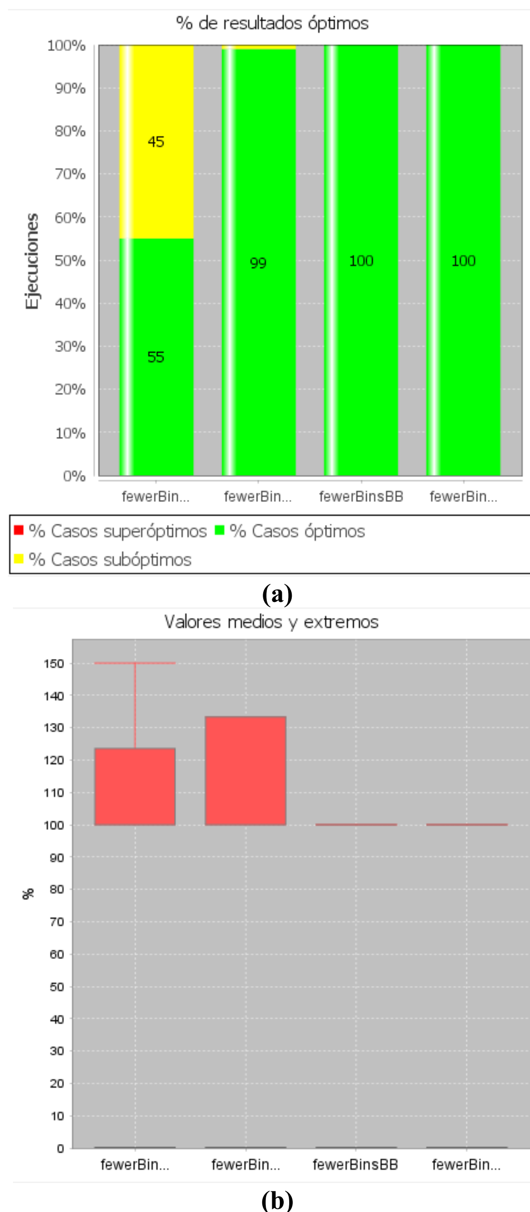

Figura 3: Diagramas con: (a) el porcentaje de ejecuciones con resultados optimales, y (b) las desviaciones máxima y media de las soluciones suboptimales respecto a las optimales.

4. Modelos conceptuales

Los modelos mentales y los conceptuales son dos modelos complementarios en educación [17]. Un modelo mental es la representación que un estudiante construye de un modelo conceptual, es decir, es su comprensión del mismo. Por tanto, un modelo mental frecuentemente es parcial y ambiguo. Por otro lado, un modelo conceptual es una representación del conocimiento, creada por el profesor para transmitirlo a los alumnos. Por tanto, un modelo conceptual debe ser preciso, completo y coherente.

Se han realizado numerosos estudios que han permitido conocer las dificultades y malas concepciones de los estudiantes [10], fuertemente ligados a sus modelos mentales. Sin embargo, existe menos discusión sobre los modelos conceptuales usados en asignaturas de algoritmos. No tenemos certeza sobre las razones de esta situación, aunque podemos conjeturar varias causas, entre ellas que la investigación sobre las dificultades de los alumnos se valora, mientras que no sucede así con la discusión de modelos conceptuales.

En esta sección, presentamos algunos modelos conceptuales sobre tres técnicas de diseño de algoritmos. Se han extraído de la investigación de otros autores, y los hemos encontrado útiles bien para mantener la coherencia en la materia que tratan bien para proporcionar guías útiles a los alumnos en su uso de la técnica de diseño.

4.1. Algoritmos voraces

Un principio básico en desarrollo de software es la separación clara entre la interfaz y la implementación de un módulo. En los algoritmos, la interfaz es la cabecera del método que resuelve un problema. Dicha cabecera debe deducirse del enunciado del problema, independientemente de las características del algoritmo usado para resolverlo. Por ejemplo, según el enunciado del problema de la mochila 0/1 [4, 5, 8, 9, 19], la cabecera de cualquier método que lo resuelva puede ser la siguiente:

```
public static int mochila
                   (int[] p, int[] b, int c)
```

donde *p* es el vector de pesos de los objetos, *b* el vector de beneficios y *c* la capacidad de la mochila.

Asimismo, los datos usados para cualquier invocación del método deben satisfacer las condiciones del enunciado del problema y ninguna más (en términos formales, deben satisfacer su precondición). Sin embargo, esta separación entre interfaz e implementación se incumple frecuentemente en los algoritmos voraces.

Aunque algunos libros proponen un esquema de código para los algoritmos voraces [4], suele usarse una variante más eficiente del esquema, aplicable con aquellos problemas en que se conocen los candidatos y sus valores a partir de los datos de entrada. El esquema optimizado consta de una primera fase en la que los candidatos se ordenan según el criterio voraz, seguida por el bucle voraz. La primera fase es necesaria porque la precondición de estos problemas no declara que los datos de entrada estén ordenados. Sin embargo, los libros de texto que presentan código de algoritmos voraces bien ignoran dicha fase (p.ej. [4] para el problema de la mochila) bien asumen que ya están ordenados (p.ej. [19]). En este último caso, esta asunción no respeta la precondición.

Medidas	fewerBinsAprox1	fewerBinsAprox2	fewerBinsBB	fewerBinsBack
Núm. ejecuciones	100	100	100	100
Núm. ejec. válidas del método	100	100	100	49
Núm. ejec. válidas en total	49	49	49	49
Tiempo máximo	0.096 ms	0.065 ms	5.626 ms	201.225 ms
Tiempo medio	0.004 ms	0.005 ms	0.578 ms	51.519 ms
Tiempo mínimo	0.001 ms	0.002 ms	0.009 ms	0.098 ms

Figura 4: Tabla de resume los tiempos de ejecución de los algoritmos dados para los juegos de datos generados.

Curiosamente, algunos libros hacen explícita la necesidad de usar un algoritmo de ordenación, haciendo uso de su complejidad para calcular la complejidad global del algoritmo voraz, pero no dedican atención a dicha fase. El problema que surge es que una "ordenación directa" de los datos de entrada produce efectos laterales indeseados, ya que los datos de entrada quedan modificados al terminar la ejecución del algoritmo voraz o se pierde el orden inicial de los objetos y no puede informarse de su posición. Por tanto, los datos de entrada deben ordenarse sin producir efectos colaterales ni perder el orden inicial de los candidatos. Una buena solución consiste en realizar una "ordenación indirecta", cuyos detalles pueden encontrarse en [21].

4.2. Programación dinámica

La programación dinámica suele etiquetarse como una de las técnicas de diseño más difíciles de aprender [19, p. 797]. De hecho, hay estudios de las malas concepciones de los alumnos sobre la programación dinámica. Sin embargo, sorprende que el método usado en los libros de texto para desarrollar algoritmos de programación dinámica es, en general, extremadamente indefinido. Por tanto, es una pregunta oportuna plantear si las malas concepciones de los alumnos se deben a modelos conceptuales imprecisos.

La técnica de la programación dinámica puede caracterizarse mediante un método de desarrollo [3]. Algunos libros de texto ni siquiera proporcionan una descripción general de la técnica en ninguna parte del capítulo dedicado a ella. Otros lo hacen, pero dan detalle insuficiente a los aprendices sobre cómo realizar algunos pasos del método. De hecho, algunos libros reconocen la imprecisión de su descripción del método. Kleinberg y Tardos presentan un " basic outline" de la técnica [9, p. 260], pero enseguida explican que "these are informal guidelines". Horowitz y Sahni [8, p. 257] confían en que "these examples should help you understand the method better".

Reproducimos aquí la descripción dada por Cormen *et al.*, que es la más completa [5, p. 359]:

1. Caracterizar la estructura de una solución optimal.
2. Definir recursivamente el valor de una solución optimal.
3. Calcular el valor de una solución optimal, generalmente en modo ascendente.
4. Construir una solución optimal a partir de la información calculada.

Obviamente, cada libro de texto presenta mejor algún paso del método. Por ejemplo, Horowitz y Sahni enfatizan la caracterización de los problemas, a veces proponiendo dos caracterizaciones para el mismo problema, una "hacia atrás" y otra "hacia adelante".

Sin embargo, la mayoría de los libros apenas dan ningún detalle sobre el paso 3. Leemos en la pág. 291 de Brassard y Bratley [4]: "La idea que subyace a la programación dinámica es, por tanto, bastante sencilla: evitar calcular dos veces una misma cosa, normalmente manteniendo una tabla de resultados conocidos que se vaya llenando a medida que se resuelven los subcasos." El capítulo contiene varios problemas que se resuelven por programación dinámica, pero nunca aclara por qué la tabla tiene cierta forma y tamaño ni por qué la tabla se rellena en un orden concreto y no en otro. Sahni *et al.* describen la fase de eliminación de la recursividad de la siguiente forma [8, p. 256; 18, p. 800]: "solve the dynamic-programming recurrence equations". Sin embargo, la eliminación de esta recursividad no suele estudiarse en ninguna asignatura, por lo que los alumnos deben hacerlo de manera intuitiva.

Encontramos descripciones más detalladas del paso 3 en Baase y van Gelder [2, pp. 474-5] y en Cormen *et al.* [5, pp. 377-8]. Estos autores dejan claro que la herramienta que debe usarse para determinar el número total de subproblemas y sus relaciones es el grafo de dependencia ("grafo del subproblema"). También señalan que el algoritmo final de programación dinámica recorrerá el grafo de dependencia siguiendo algún orden topológico inverso. Sin embargo, su discusión de varios algoritmos apenas ilustra la forma y tamaño de la tabla, ni el orden de recorrido usado.

El uso de una visualización hace visibles estos pasos y decisiones. Para el problema de la SCML, incluido en la sección 3.1, una sencilla correspondencia entre los dos parámetros i y j del algoritmo recursivo con las columnas y filas de la tabla permite generar la Figura 2. En ella se aprecia el tamaño necesario para la tabla. También permite derivar órdenes topológicos inversos, fácilmente de codificar con dos bucles anidados:

- Por filas de abajo arriba, rellenando cada fila de derecha a izquierda.
- Por columnas de derecha a izquierda, rellenando cada columna de abajo arriba.

También se puede rellenar la tabla por diagonales, pero es más difícil de programar.

4.3. Ramificación y poda

Las técnicas de búsqueda, como la de vuelta atrás o la de ramificación y poda, no son tratadas en algunos libros de texto [2, 5, 9] o se hace con menos profundidad que otras técnicas de diseño de algoritmos [4]. Probablemente Howrowitz y Sahni son los autores que han presentado estas técnicas con más detalle [8, 19].

La técnica de ramificación y poda puede caracterizarse con varios elementos, entre ellos cotas sobre las soluciones en construcción. Cuando se trata de un problema de maximización, se maneja una cota superior, mientras que si es de minimización, es una cota inferior. Los problemas de minimización admiten el uso simultáneo de una cota superior y una otra inferior [8].

Sin embargo, los libros no dan ninguna pauta a los alumnos para la definición de las funciones de cota. Afortunadamente, encontramos pautas en los libros de inteligencia artificial, ya que se usan para técnicas de búsqueda, como la búsqueda A*. Una función de cota $f(j)$ frecuentemente tiene el siguiente formato [18]:

$$f(j) = g(j) + h(j)$$

donde $g(j)$ calcula el valor asociado a las primeras j decisiones tomadas (es decir, la función objetivo limitada a estos objetos), y $h(j)$ hace una estimación optimista sobre las restantes decisiones.

Por ejemplo, sea el problema de la mochila 0/1 de nuevo [4, 5, 8, 9, 19]. La función objetivo a maximizar es max $\sum_{j=1}^{n} p_j \cdot x_j$, donde x_j representa la decisión tomada sobre el objeto j, siendo 0 su exclusión de la mochila y 1 su inclusión. Es decir, la función objetivo es igual a la suma de los objetos que se introduzcan en la mochila. Al ser un problema de maximización, la función de cota debe definir una cota superior sobre una solución optimal. Por tanto, una función de cota adecuada para el problema de la mochila 0/1 es:

$$f(j) = \sum_{i=1}^{j} p_i x_i + \sum_{i=j+1}^{n} p_i$$

en la que la estimación optimista supone que podrían meterse en la mochila los objetos aún no considerados.

5. Conclusiones

En la comunicación, hemos realizado una reflexión sobre tres aspectos de la docencia de los algoritmos: el papel destacado de los problemas de optimización y la conveniencia de más experiencia directa con los algoritmos y de mayor precisión en algunos modelos conceptuales. Las reflexiones son el resultado de muchos años de docencia de la materia y esperamos que ayuden a otros profesores en su labor docente.

En nuestra experiencia, las soluciones presentadas han contribuido a un mayor aprendizaje de los alumnos. La atención a conceptos básicos de optimización sienta una base que incluso se refleja en un mejor uso del lenguaje (p.ej. es incorrecto decir “más óptimo”). Los modelos conceptuales presentados proporcionan esquemas de código o metodologías que ayudan a los alumnos en el desarrollo de algoritmos basados en las técnicas de diseño correspondientes.

Los sistemas de visualización y comparación experimental usados (SRec [27, 28], AlgorEx [24] y GreedEx [6, 26]) permiten acceder a facetas de los algoritmos que de otra forma serían invisibles. Como resultado, los alumnos desarrollan menos malas concepciones [22][23] y menos graves. Obviamente, cada sistema es adecuado para temas distintos. La visualización de programas es más útil con problemas de grafos y con las técnicas de divide y vencerás, programación dinámica y búsquedas. La experimentación es más útil con algoritmos heurísticos, aproximados y criterios voraces inexactos. Hay que destacar el uso continuado de los sistemas a lo largo de los años, que ha permitido su evaluación, ajuste y ampliación.

Las cuestiones tratadas se han presentado sin tener en cuenta su integración en la asignatura, pero es un aspecto clave para su éxito educativo. Los conceptos de optimización y los modelos conceptuales deben integrarse en el temario. En nuestra experiencia, es conveniente presentar los algoritmos heurísticos antes que otras técnicas para familiarizar desde el principio a los alumnos con la existencia de soluciones válidas pero suboptimales. También resulta útil organizar una sesión de familiarización para cada herramienta, de forma coordinada con el contenido de algún tema.

Asimismo, conviene que existan hilos conductores que estructuren la asignatura y no sea un simple conjunto de temas. Algunos elementos que dan dicha continuidad son el énfasis en la optimalidad, la resolución de los mismos problemas con distintas técnicas de diseño y el uso recurrente de los sistemas de visualización o experimentación, tanto en clase como para realizar las prácticas.

Referencias

[1] Association for Computing Machinery and IEEE Computer Society, The Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*. DOI: 10.1145/2534860.

[2] Sara Baase y Allen van Gelder. *Computer Algorithms*, 3ª ed. Addison Wesley Longman, 2000.

[3] Richard S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4), 403-417, diciembre 1980. DOI: 10.1145/356827.356831.

[4] Gilles Brassard y Paul Bratley. *Fundamentos de Algoritmia*. Prentice-Hall, 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein. *Introduction to Algorithms*, 3ª ed. MIT Press, 2009.

[6] Ouafae Debdi, Maximiliano Paredes Velasco y J. Ángel Velázquez Iturbide. GreedExCol, a CSCL tool for experimenting with greedy algorithms. *Computer Applications in Engineering Education*, 23(5), 790-804, septiembre 2015.

[7] Sally A. Fincher y Anthony V. Robins, eds. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press, 2019.

[8] Ellis Horowitz, Sartaj Sahni y Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1997.

[9] Jon Kleinberg y Éva Tardos. *Algorithm Design*. Pearson Addison-Wesley, 2006.

[10] Colleen M. Lewis, Michael J. Clancy y Jan Vahrenhold. Student knowledge and misconceptions. En *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher y Anthony V. Robins, eds. Cambridge University Press, pp. 773-800, 2019.

[11] Raymond Lister. Toward a developmental epistemology of computer programming. En *Proc. 11th Workshop Primary and Secondary Computing Education, WiPSCE'16*, pp. 5-16.

[12] Andrew Luxton-Reilly. Learning to program is easy. En *Proc. 21st Annual Conf. Innovation and Technology in Computer Science Education, ITiCSE'16*, pp. 284-289.

[13] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett Becker, Michail Giannakos, Amruth Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, y Claudia Szabo. Introductory programming: A systematic literature review. En *Proc. Companion 23th Annual Conf. Innovation and Technology in Computer Science Education, ITiCSE'18*, pp. 55-106.

[14] Michael Luu, Matthew Ferland, Varun Nagaraj Rao, Arushi Arora, Randy Huynh, Frederick Reiber y Jennifer Wong-Ma. What is an algorithms course? Survey results of introductory undergraduate algorithms courses in the U.S. En *Proc. 54th ACM Technical Symp. Computer Science Education, SIGCSE'23*, pp. 284-290.

[15] Jeff Matocha. Laboratory experiments in an algorithms course: Technical writing and the scientific method. En *Proc. 32nd ASEE/IEEE Frontiers in Education Conf., FIE '02*, pp. T1G 9-13.

[16] Thomas L. Naps, Guido Roessling, Vicki Almstrum, Wand Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger y J. Ángel Velázquez Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin* 35(2), 131-152, 2003. DOI: 10.1145/960568.782998.

[17] Donald Norman. Some observations on mental models. En *Mental Models*, Dedre Gentner y Albert Stevens, eds. Erlbaum, pp. 7-14, 1983.

[18] Stuart J. Russell y Peter Norvig. *Artificial Intelligence: A Modern Approach*, 3ª ed. Pearson Education, 2010.

[19] Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java*, 2ª ed. Silicon Press, 2004.

[20] John Stasko, John Domingue, Marc H. Brown y Blaine A. Price, eds. *Software Visualization*. MIT Press, 1998.

[21] J. Ángel Velázquez Iturbide. The design and coding of greedy algorithms revisited. En *Proc. 16th Annual Conf. Innovation and Technology in Computer Science Education, ITiCSE'11*, pp. 8-12. DOI: 10.1145/1999747.1999753.

[22] J. Ángel Velázquez Iturbide. An experimental method for the active learning of greedy algorithms. *ACM Trans. Computing Education*, 13(4), artículo 18, 2013. DOI: 10.1145/2534972.

[23] J. Ángel Velázquez Iturbide. Students' misconceptions of optimization problems. *Proc. 24th Annual Conf. Innovation and Technology in Computer Science Education*, *ITiCSE'19*, pp. 464-470. DOI 10.1145/3304221.3319749

[24] J. Ángel Velázquez Iturbide. A unified framework to experiment with algorithm optimality and efficiency. *Computer Applications in Engineering Education*, 29(6), 1.793-1.810, 2021. DOI: 10.1002/cae.22423.

[25] J. Ángel Velázquez Iturbide y Ouafae Debdi. Experimentation with optimization problems in algorithm courses. En *Proc. International Conf. Computer as a Tool, EUROCON'11*. DOI: 10.1109/EUROCON.2011.5929294.

[26] J. Ángel Velázquez Iturbide, Ouafae Debdi, Natalia Esteban Sánchez y Celeste Pizarro. GreedEx: A visualization tool for experimentation and discovery learning of greedy algorithms. *IEEE Trans. Learning Technologies*, 6(2), 130-143, 2013. DOI: 10.1109/TLT.2013.8.

[27] J. Ángel Velázquez Iturbide y Antonio Pérez Carrasco. InfoVis interaction techniques in animation of recursive programs. *Algorithms*, 3(1), 76-91, 2010. DOI: 10.3390/a3010076.

[28] J. Ángel Velázquez Iturbide y Antonio Pérez Carrasco. Systematic development of dynamic programming algorithms assisted by interactive visualization. En *Proc. 21st Annual Conf. Innovation and Technology in Computer Science Education, ITiCSE'16*, pp. 71-76. DOI: 10.1145/2899415.2899450.