Actas de las Jenui, vol. 5. 2020.
Páginas: 229-236

Interacciones entre tres tradiciones intelectuales en la docencia de la algoritmia

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

Resumen

La informática es una disciplina en la que confluyen tres tradiciones intelectuales distintas: matemáticas, ciencias experimentales e ingeniería. La algoritmia no es una excepción y contiene elementos de las tres tradiciones. La comunicación clasifica en alguna de las tres tradiciones contenidos que son comunes en las asignaturas de algoritmos. La aportación principal consiste en la presentación de varias actividades docentes novedosas, basadas en el uso conjunto de la experimentación y alguna de las otras dos tradiciones. Están inspiradas en la hipótesis de que podría mejorarse el aprendizaje de algunos contenidos tratándolos desde varios enfoques. Por ejemplo, los resultados de probar algoritmos heurísticos con respecto a su optimalidad pueden usarse para el diseño de contraejemplos de su optimalidad. Se completa la descripción de estas actividades con nuestra experiencia durante el curso académico 2019-20. Los resultados obtenidos para algunos contenidos ingenieriles son satisfactorios, pero no tanto para otros contenidos teóricos.

Abstract

Computing is a discipline which is at the cross of three intellectual approaches: mathematics, experimental sciences and engineering. Algorithmics is not an exception and it also contains elements from the three approaches. The paper classifies contents which are common in algorithms courses into any of these traditions. The main contribution consists in presenting several innovative instructional activities, which are based on the joint use of experimentation and one of the other two traditions. Their ultimate goal is to enhance the understanding of some contents by addressing them from several approaches. For instance, the outcomes of testing heuristic algorithms with respect to optimality can be used to design counterexamples of their optimality. The description of these activities is completed with our experience in the academic year 2019-20. The results obtained for some engineering contents are satisfactory, not so much for theoretical contents.

Palabras clave

Algoritmia, experimentación, optimalidad, casos de prueba, precondición, contraejemplos, depuración.

1. Introducción

Existen distintas formas de ver el mundo que nos rodea. Ya lo afirmaba C. P. Snow al hablar de “las dos culturas”: “los intelectuales literarios en un polo, y en el otro los científicos, y como más representativos, los físicos” [16]. Pueden existir varias culturas incluso dentro de una sola disciplina. Así, la informática se basa en tres “paradigmas” o “estilos culturales” [4]: matemáticas, ciencias experimentales e ingeniería.

La algoritmia no es una excepción y contiene elementos de las tres tradiciones. Así, se utilizan métodos formales para el análisis de complejidad o para las demostraciones de optimalidad en la técnica voraz o de acotación en algoritmos aproximados. Asimismo, las ciencias experimentales se utilizan para tomar medidas de tiempo en algoritmos de difícil o imposible análisis formal, como los algoritmos de vuelta atrás, o para comprobar las predicciones sobre tiempos de ejecución derivadas de los análisis de complejidad. Por último, las técnicas de diseño de algoritmos proporcionan patrones, una actividad común en las ingenierías, para el diseño y desarrollo de algoritmos.

La comunicación presenta diversos contenidos y actividades, cada uno clasificado en una tradición intelectual. La contribución principal de la comunicación consiste en la presentación de varias actividades docentes innovadoras, basadas en la experimentación y alguna otra tradición. Están inspiradas en la hipóte-

Este trabajo se ha financiado con los proyectos de investigación iProg (TIN2015-66731-C2-1-R) del Ministerio de Economía y Competitividad y e-Madrid-CM (S2018/TCS-4307) de la Comunidad Autónoma de Madrid. El proyecto e-Madrid-CM también está financiado con los fondos estructurales FSE y FEDER.

sis de que podría mejorarse el aprendizaje de algunos contenidos tratándolos desde varios enfoques.

La estructura de la comunicación es la siguiente. En la sección segunda se presentan las características principales de las tres tradiciones intelectuales, así como una relación no exhaustiva de contenidos de una asignatura de algoritmos, clasificados por tradición intelectual. La sección tercera presenta diversas actividades docentes, principalmente basadas en la experimentación. La sección cuarta presenta varias actividades docentes de carácter mixto nuestra experiencia en el curso académico 2019-20. Por último, terminamos con nuestras conclusiones.

2. Tres tradiciones intelectuales en la algoritmia

Las tres tradiciones presentes en la informática tienen una epistemología distinta [4]. Las matemáticas buscan desarrollar teorías válidas y coherentes mediante el uso de métodos formales. La forma de trabajar puede resumirse en la sucesiva realización de definiciones (axiomas), propuestas de propiedades (teoremas) y su demostración formal. Su forma de trabajar raramente es lineal, ya que pueden descubrirse errores o incompletitud en los elementos citados.

Las ciencias experimentales investigan fenómenos para comprenderlos con ayuda de métodos experimentales. El investigador propone hipótesis que permiten hacer predicciones, diseña experimentos para recoger datos y los analiza. Si las predicciones y las evidencias experimentales no coinciden, el científico debe reconsiderar sus hipótesis.

Las ingenierías tienen como objetivo desarrollar sistemas o dispositivos que resuelvan un problema de forma eficaz y eficiente. El ingeniero debe recoger requisitos, transformarlos en una especificación, desarrollar un sistema y probarlo. Si las pruebas muestran fallos, debe revisarse dicho desarrollo.

La algoritmia contiene elementos de las tres tradiciones intelectuales, aunque no siempre resulta fácil clasificarlos en una tradición concreta. Así, la realización de pruebas puede considerarse una actividad experimental (cuando se espera corroborar cierto comportamiento previsto para un algoritmo) o ingenieril (cuando se prueba su corrección). En estos casos, la clasificación en una u otra tradición dependerá de su papel en el proceso de diseño y análisis de algoritmos.

Veamos una clasificación de elementos de la algoritmia. Sin ánimo de ser exhaustivos, hemos seleccionado temas tratados en libros de algoritmia, identificando y clasificando los hechos, conceptos o métodos más destacados. Aunque es inevitable que la selección sea subjetiva, consideramos que es razonable porque pueden encontrarse en un alto porcentaje de los libros de texto de prestigio (p.ej. [9, 13]).

2.1. Análisis de rendimiento

El análisis de rendimiento de algoritmos busca determinar si un algoritmo es eficiente en el uso de los recursos del ordenador, es decir de memoria y, sobre todo, de tiempo. Se trata de una actividad de carácter principal pero no exclusivamente matemático. Destacan los siguientes elementos:

- Elementos matemáticos: fundamentos matemáticos, función de tiempo y de memoria, casos mejor, peor y medio, métodos de análisis de algoritmos iterativos, métodos de análisis de algoritmos recursivos, notaciones asintóticas.
- Elementos experimentales: medición de tiempos.
- Elementos ingenieriles: soluciones generales para ecuaciones de recurrencias, notaciones asintóticas más frecuentes.

2.2. Técnica de divide y vencerás

Es la técnica de diseño de algoritmos más sencilla y popular. Podemos distinguir:

- Elementos matemáticos: razonamiento inductivo.
- Elementos ingenieriles: patrón de código, solución general de análisis de complejidad.

2.3. Técnica voraz

Es la técnica de diseño de algoritmos más sencilla para problemas de optimización. Podemos distinguir los siguientes elementos:

- Elementos matemáticos: conceptos de optimización, demostraciones de optimalidad, contraejemplos de optimalidad.
- Elementos ingenieriles: criterio voraz, patrón de código voraz, patrón de código para ordenar candidatos [17], estructuras de datos para mejorar la eficiencia.

2.4. Algoritmos heurísticos y aproximados

Permiten resolver problemas de optimización de forma suboptimal. Podemos distinguir los siguientes elementos:

- Elementos matemáticos: algoritmos aproximados c-absolutos y $\in$-relativos, demostraciones de acotación.
- Elementos experimentales: medición de diferencias entre soluciones optimales y suboptimales.

2.5. Técnica de vuelta atrás

Es otra técnica muy popular, que permite resolver de forma exacta algoritmos combinatorios y de optimización. Podemos distinguir varios elementos:

- Elementos matemáticos: condiciones de validez.
- Elementos experimentales: medición de rendimiento.
- Elementos ingenieriles: árboles de búsqueda, patrones de código (para encontrar todas las soluciones válidas, una sola solución válida o una solución optimal), formas de reducir el tamaño del árbol de búsqueda, estructuras de datos para mejorar la eficiencia.

2.6. Técnica de ramificación y poda

Es una técnica de búsqueda alternativa a la técnica de vuelta atrás. Podemos distinguir varios elementos:

- Elementos matemáticos: conceptos de optimización.
- Elementos experimentales: medición de tiempos.
- Elementos ingenieriles: definición de funciones de cota, patrón de código.

2.7. Técnica de programación dinámica

Es una técnica alternativa para resolver de forma exacta algoritmos de optimización. Quizá sea la más difícil de aprender. Podemos distinguir los siguientes elementos:

- Elementos matemáticos: conceptos de optimización, principio de optimalidad de Bellman.
- Elementos ingenieriles: métodos de eliminación de la recursividad múltiple redundante, patrones de código.

3. Actividades docentes

Las actividades que los alumnos realizan en el curso de una asignatura de algoritmos son variadas y pueden corresponder a una u otra tradición. Probablemente, las tres actividades más frecuentes son:

- Diseñar un algoritmo que resuelva un problema, quizá utilizando una técnica de diseño concreta. Se trata de una actividad ingenieril (aunque también puede plantearse como una actividad creativa [7]).
- Realización de pruebas para tener cierta evidencia de la corrección de un algoritmo desarrollado. Las pruebas son una actividad ingenieril realizada por los propios alumnos, generalmente sin planificar. También las utilizan muchos profesores para corregir prácticas, bien de forma manual bien mediante sistemas de corrección automática [8, 22].
- Analizar la complejidad de un algoritmo, bien un algoritmo dado, bien diseñado por el propio alumno. Se trata de una actividad matemática.

Otra actividad común, aunque menos frecuente, consiste en medir tiempos de ejecución. Es una actividad experimental que puede plantearse con diversos objetivos: bien corroborar los resultados teóricos de análisis de complejidad de un algoritmo [10], bien inferir su orden de complejidad [5], bien evaluar la eficiencia de algoritmos que no pueden analizarse analíticamente [5, 12]. La mayor parte de las experiencias del primer tipo se basan en algoritmos de ordenación [10], aunque obviamente pueden usarse otros [1, 2]. Cabe destacar algunas publicaciones que ponen énfasis en el método científico [1, 11] o en detalles técnicos de la medición de tiempos de ejecución [13, 14, 15].

En el resto de la sección presentamos dos variantes de una actividad docente basada en experimentar con la optimalidad. Se usa como base para las actividades docentes presentadas en la sección siguiente.

3.1. Experimentación con la optimalidad de criterios voraces

En este apartado presentamos una actividad docente basada en experimentar con la optimalidad [18]. En su primera formulación, se diseñó como un método de aprendizaje activo de los algoritmos voraces. El objetivo era determinar qué criterios voraces para resolver un problema dado parecían ser buenos candidatos para construir algoritmos voraces exactos, es decir, que siempre calcularan soluciones optimales. (Obsérvese que un experimento solamente permite obtener evidencia sobre la probable exactitud de un algoritmo voraz, pero la certeza requiere una demostración formal.)

Sea un problema de optimización que se pretende resolver mediante algún algoritmo voraz. La forma de proceder es la siguiente, donde los pasos 2-4 pueden repetirse tantas veces como parezca conveniente:

1. Se proponen todos los criterios voraces "razonables" que se pueda.
2. Se genera aleatoriamente un juego de datos de entrada que sea válido para el problema.
3. Se ejecutan todos los criterios voraces con los datos de entrada generados y se almacenan sus resultados en tablas.
4. Se comparan los resultados obtenidos para determinar si algunos criterios voraces han dado resultados optimales en el 100% de los casos ejecutados hasta el momento.

La tarea experimental resulta más fácil de realizar si se dispone de un sistema que se ocupe de aquellos detalles del experimento que son ajenos a los propios algoritmos voraces, como son la generación de datos aleatorios, el almacenamiento en tablas y la presentación de resultados en formato numérico o diagramático. Para dar soporte computacional a estas tareas se desarrolló el sistema GreedEx [19]. Está concebido como introducción a los algoritmos voraces, dando soporte a seis problemas distintos. El sistema permite

realizar un experimento paso a paso, pero también ejecutando en una sola operación todos los criterios voraces y sobre numerosos juegos de datos. También permite visualizar el efecto de cada criterio voraz e incluye varias facilidades de carácter docente.

Dado que GreedEx permite ejecutar varios algoritmos con diversos juegos de datos, tiene similitudes con los sistemas de corrección automática. Sin embargo, también presenta diferencias destacadas, que se resumen en el Cuadro 1.

Característica	**GreedEx**	**Sistemas de corrección automática**
Propiedad a evaluar	Optimalidad	Corrección
Casos de prueba	Generados aleatoriamente	Diseñados por el alumno o el profesor
Número de casos de prueba	Hasta miles	Unos pocos
Valores de referencia	Calculados por los distintos algoritmos	Proporcionados por el alumno o el profesor

Cuadro 1: Diferencias principales entre GreedEx y los sistemas de corrección automática.

Podría pensarse que la actividad propuesta es fácil. Sin embargo, corresponde a los niveles de análisis y evaluación de la taxonomía de Bloom, que son categorías de procesos cognitivos situadas en la parte superior de la taxonomía. De hecho, al plantearse esta actividad por primera vez, solamente la realizaron bien el 28% de los alumnos, mientras que el 44% entregaron trabajos con malas concepciones graves [18]. En los cursos siguientes se realizaron diversas intervenciones didácticas, alcanzándose al tercer año un 87% de respuestas correctas y habiendo desaparecido las entregas con malas concepciones graves [18]. Las malas concepciones se referían a conceptos básicos de problemas, algoritmos y optimización.

También se evaluó la eficiencia educativa de GreedEx [6]. Todos los alumnos mejoraron (de forma estadísticamente significativa) al nivel de conocimiento de la taxonomía de Bloom, pero los que usaron GreedEx también mejoraron en la comprensión de los algoritmos voraces.

3.2. Experimentación con la optimalidad de algoritmos de optimización

Dados los buenos resultados docentes del método experimental, también se utilizó con otras técnicas de diseño de algoritmos de optimización. Asimismo, se modificó el sistema GreedEx para dar soporte a la experimentación con algoritmos de optimización cualesquiera. Un nuevo sistema, denominado OptimEx [19], permite experimentar con la optimalidad de algoritmos de optimización desarrollados en Java.

El sistema OptimEx puede utilizarse para actividades docentes con otras técnicas de diseño:

- Algoritmos heurísticos. Permite determinar el porcentaje de casos en que cada algoritmo calcula un valor optimal, así la diferencia media y máxima entre valores suboptimales y optimales. Se trata de una actividad de descubrimiento [21].
- Algoritmos aproximados. Permite determinar el porcentaje de casos en que cada algoritmo calcula un valor optimal y el valor medio calculado, así como comprobar que el valor extremo calculado está dentro de la cota demostrada formalmente para el algoritmo. Se trata de una actividad de descubrimiento y predictiva.

OptimEx se ha utilizado como instrumento para identificar malas concepciones de los alumnos sobre programación, optimización y algoritmia [21].

4. Actividades docentes mixtas

En esta sección presentamos algunas actividades docentes que se apoyan en el método experimental esbozado en la sección anterior, pero contienen elementos de las otras dos tradiciones intelectuales. Se analiza nuestra experiencia durante el curso académico 2019-20 en la asignatura optativa “Algoritmos Avanzados”, de cuarto curso del grado en Ingeniería Informática de la Universidad Rey Juan Carlos.

Presentamos de forma resumida la organización de las prácticas de la asignatura. Las prácticas se corregían pocos días después de su plazo de entrega, se notificaba la nota a sus autores junto a comentarios de carácter formativo, dándoles la oportunidad de corregir sus errores y realizar una segunda entrega.

Se propusieron seis prácticas, de las cuales esbozamos tres. Las tres prácticas se basaban en un mismo problema de optimización, que se enuncia después.

- Práctica 2: algoritmos heurísticos. Se describe un problema y se esboza un algoritmo heurístico. Se pide diseñar otro algoritmo heurístico al menos, programar todos y comprobar experimentalmente su optimalidad con OptimEx.
- Práctica 3: algoritmos de búsqueda en espacios de estados. Se pide desarrollar un algoritmo de vuelta atrás y otro de ramificación y poda para este problema y comparar experimentalmente la optimalidad y la eficiencia en tiempo de estos algoritmos y los heurísticos de la práctica 2.
- Práctica 5: programación dinámica. Se pide desarrollar un algoritmo de programación dinámica para este problema, analizar su eficiencia, y comparar experimentalmente la optimalidad y la eficiencia en tiempo de estos algoritmos y los desarrollados en las prácticas anteriores.

Estas tres prácticas se basaban en un mismo problema, de planificación de trabajos con beneficio

máximo [9, p. 323], que esbozamos a continuación. Un equipo de informáticos contrata y realiza trabajos con periodicidad semanal. Para mejorar su rendimiento, clasifica los contratos en trabajos de baja o de alta tensión (estrés). Si realizan un trabajo de baja tensión en la semana i-ésima, obtendrán un ingreso b_i>0, mientras que si realizan una tarea de alta tensión, ingresarán a_i>0. En general, las tareas de alta tensión proporcionan ingresos superiores a las de baja tensión, pero no tiene por qué cumplirse siempre.

Para poder atender satisfactoriamente sus compromisos, el equipo acuerda poder contratar un trabajo de baja tensión sin restricciones, pero solamente contratarán un trabajo de alto estrés si han descansado la semana anterior. La única excepción a esta restricción es la primera semana.

Un plan para *n* semanas consiste en una secuencia de decisiones, donde una decisión semanal consiste en realizar una tarea de baja tensión, realizar una tarea de alta tensión o descansar. Un plan es válido si se respeta la restricción sobre los trabajos de alta tensión. El beneficio de un plan válido es la suma de los ingresos obtenidos por los trabajos contratados. Dada una secuencia de *n* ingresos para tareas de baja y de alta tensión, el problema consiste en determinar un plan válido que proporcione un beneficio máximo.

Por ejemplo, sean unos ingresos para cinco semanas iguales a {10,15,15,15,15} por trabajos de alta tensión y a {10,10,10,10,10} por trabajos de baja tensión. El plan válido con máximo beneficio consiste en seleccionar las cinco tareas de baja tensión, con un ingreso igual a 10+10+10+10+10=50.

4.1. Experimentación y especificación de problemas

En esta subsección, se presenta una experiencia en la que intervienen las tradiciones experimental y matemática. Los experimentos esbozados en la sección anterior se basan en comparar algoritmos que resuelven un mismo problema. Un problema algorítmico se describe de forma precisa mediante su especificación, que consta de signatura, precondición y postcondición. En el caso de los problemas de optimización, la postcondición consta a su vez de condición de validez y de función objetivo. Veamos cómo obtiene OptimEx la información que necesita sobre el problema asociado a los algoritmos a comparar:

- Cabecera de los métodos: Dado que todos los algoritmos resuelven el mismo problema, sus datos de entrada y de salida son los mismos. Por tanto, todos los algoritmos deben tener la misma signatura. OptimEx determina a partir del código las cabeceras de los métodos públicos de la clase de Java cargada. En caso de que la clase contenga métodos con varias cabeceras, da al usuario a elegir una de ellas. Obsérvese que esta restricción de que los métodos a probar se ajusten de forma estricta a cierta cabecera también existe en los sistemas de corrección automática [8].
- Función objetivo: El problema de optimización puede ser de maximización o de minimización. El usuario debe seleccionar la opción adecuada.
- Generación aleatoria de casos de prueba. Los datos a generar deben satisfacer la precondición. La función de generación de datos ofrece un diálogo para que el usuario indique los rangos de valores permitidos para cada variable simple, así como el tamaño de los arrays.

La generación aleatoria de datos es una operación aparentemente sencilla. Una especificación adecuada de los rangos de valores de los datos a generar debe tener en cuenta las restricciones del enunciado del problema. Es decir, debe haber coherencia entre un aspecto formal del problema y un aspecto práctico del experimento, a saber, entre los rangos de valores de los datos de entrada y la precondición del problema.

En la práctica, el usuario indica rangos de valores que son más restrictivos que los especificados en la precondición, pero debe tenerse cuidado de no incumplir la precondición. Por ejemplo, si el problema en cuestión fuera alguna variante del problema de la mochila, tanto los pesos como los beneficios deben ser mayores que cero. Normalmente, el usuario restringirá el valor máximo a varias decenas (p.ej. 25, 50 ó 100), lo cual facilita la comprensión de los casos de prueba. Sin embargo, sería un error permitir la generación de un beneficio o un peso nulo.

Esta actividad aparentemente sencilla es realizada con dificultad por muchos alumnos. Si se les pregunta por el concepto de precondición, las respuestas serán satisfactorias. Sin embargo, encuentran dificultades para especificarla en problemas concretos.

Veamos la experiencia del curso 2019-20. Como parte de la práctica 2, se debía especificar el problema. En concreto, la precondición del problema debía indicar que: (a) los dos vectores de beneficios deben tener la misma longitud, y (b) los beneficios deben ser mayores que cero.

Se recogieron 34 prácticas. Un análisis de las mismas permite clasificar sus precondiciones en tres categorías: completas, parciales y vacías. Una contribución parcial solamente contiene una de las dos partes de la precondición. Los resultados se muestran en el Cuadro 2.

Precondición	Núm. grupos	% grupos
Completa	11	32%
Parcial	17	50%
Vacía	6	18%

Cuadro 2: Resultados de especificar precondiciones (N=34)

Obsérvese que el número de precondiciones completas apenas llega a una tercera parte de las entregas

y que casi una quinta parte de los alumnos dejó la precondición vacía o escribió elementos impropios de la misma. Estos resultados muestran un bajo dominio en la especificación de problemas algorítmicos.

También podemos preguntarnos por la coherencia entre la precondición especificada y la generación aleatoria de datos realizada. Todos los alumnos generaron dos vectores de la misma longitud. Falta saber si permitieron la generación de beneficios nulos.

El análisis de ambas especificaciones permite clasificar las contribuciones en tres categorías: coherentes, (parcialmente) incoherentes y desconocida. Una contribución es coherente cuando el alumno ha dado valores sobre los beneficios que son coherentes entre su precondición y en la generación aleatoria de datos. Incluimos en esta categoría los alumnos que especificaron parcialmente la precondición con la restricción sobre los beneficios. Una contribución es (parcialmente) incoherente cuando su precondición indica que los beneficios deben ser mayores que cero, pero la generación de datos admite beneficios nulos. Finalmente, se desconoce la coherencia de los grupos que no declararon los rangos usados para generar datos. Los resultados se muestran en el Cuadro 3.

Coherencia	Núm. grupos	% grupos
Coherente	16	47%
(Parcialmente) incoherente	2	6%
Desconocida	16	47%

Cuadro 3: Resultados de coherencia entre precondiciones y generación aleatoria de datos (N=34)

Obsérvese que casi la mitad de los grupos han sido coherentes entre su precondición y sus rangos de valores, pero de otros tantos grupos ignoramos su coherencia. En todo caso, el número de grupos incoherentes fue muy pequeño.

En definitiva, los alumnos no presentan problemas para aplicar coherentemente su especificación en la generación de juegos de datos, pero sí en el desarrollo de la propia especificación.

4.2. Experimentación y contraejemplos

En esta subsección, se presenta otra experiencia en la que, de nuevo, intervienen las tradiciones experimental y matemática. Un contraejemplo es un caso que desmiente una afirmación. En algoritmia es frecuente pedir un contraejemplo de que un algoritmo cumpla cierta propiedad, por ejemplo, un contraejemplo de que un criterio voraz sea exacto. Se trata de una actividad más sencilla que construir una demostración formal de que el algoritmo cumple esa propiedad. No obstante, exige una capacidad de abstracción suficiente como para analizar el dominio del problema e identificar propiedades relevantes, una actividad a veces difícil para muchos alumnos.

Proponemos una forma alternativa de construir contraejemplos, basada en la experimentación. Por ejemplo, para demostrar la inexactitud de un algoritmo, puede realizarse un experimento que incluya este y otros algoritmos. Si se encuentran casos en los que el algoritmo calcula un valor subóptimo, pueden analizarse para determinar situaciones en las que el algoritmo se comporta subóptimamente. A partir de esta información es relativamente fácil diseñar un contraejemplo de tamaño y con valores menores.

Se propuso esta tarea como parte de la práctica 2. El Cuadro 4 muestra las contribuciones de contraejemplos, clasificadas en tres categorías, similares a las de la subsección anterior: completas, incompletas y vacías. Una contribución completa aporta un contraejemplo para todos los algoritmos inexactos, mientras que una incompleta solo incluye contraejemplos para algunos algoritmos o incluye contraejemplos incompletos (p.ej. sin detallar su resultado). La segunda columna del cuadro presenta los contraejemplos aportados de forma experimental y la tercera, los diseñados por los propios alumnos.

Contraejemplos	Experimentales # (%)	Diseñados # (%)
Completos	13 (38%)	11 (32%)
Incompletos	13 (38%)	11 (32%)
Vacío	8 (24%)	12 (35%)

Cuadro 4: Resultados de contraejemplos obtenidos en la primera entrega (N=34)

Obsérvese que, a pesar de que el experimento lo realizaron 33 de los 34 grupos, una cuarta parte aproximadamente no aportan ningún contraejemplo. Los resultados son peores para los contraejemplos diseñados, donde una tercera parte de los grupos no aportan ningún contraejemplo.

Recordemos que los grupos podían realizar una segunda entrega de cualquier práctica. Nueve grupos la realizaron. El Cuadro 5 muestra una clasificación de las prácticas finales de cada grupo.

Contraejemplos	Experimentales # (%)	Diseñados # (%)
Completos	13 (38%)	14 (41%)
Incompletos	17 (51%)	12 (35%)
Vacío	4 (12%)	8 (24%)

Cuadro 5: Resultados de contraejemplos tras la segunda entrega (N=34)

De los nueve grupos que realizaron una segunda entrega, cinco que antes no aportaban contraejemplos experimentales, los aportaron, aunque incompletos. Un grupo que inicialmente incluía contraejemplos completos, no aportó nada (lo clasificamos como vacío, aunque probablemente sea un despiste). En conjunto, no mejoró el número de contraejemplos experimentales completos, pero sí el de incompletos.

En cuanto a los contraejemplos diseñados, tres grupos que antes no aportaban contraejemplos, los aportaron completos y otro grupo, incompleto. En resumen, aumentó el número de contraejemplos.

En conjunto, puede observarse que el número de grupos que, en una primera entrega, aportan contraejemplos incompletos o no los aportan oscila entre el 62 y el 68%. Es ligeramente mayor el rendimiento de los alumnos al identificar contraejemplos experimentales que al diseñarlos. En una segunda entrega, en ambos casos se reduce el porcentaje de grupos que no aportan contraejemplos, pero sigue siendo apreciable.

Algunas causas posibles de estos datos podrían ser incomprensión de qué es un contraejemplo o de cómo identificarlo, o incapacidad para diseñarlo. Incluso podría deberse a que es una pequeña parte de la práctica, a cuya elaboración quizá se le da poca importancia o pasa desapercibida. En el caso de los contraejemplos experimentales, también podría haber una familiarización insuficiente con OptimEx.

4.3. Experimentación y depuración

En esta subsección, se presenta una experiencia en la que intervienen las tradiciones experimental e ingenieril. Tal y como se ha concebido en las secciones 3.1 y 3.2, un experimento con la optimalidad implica a varios algoritmos y numerosos casos de prueba. Los resultados obtenidos deben ser coherentes con las predicciones de la teoría. En concreto, un algoritmo heurístico o aproximado será inexacto. Por el contrario, un algoritmo voraz (cuya optimalidad se haya demostrado), de vuelta atrás, de ramificación y poda o de programación dinámica debe ser exacto. La teoría permite predecir el comportamiento de cada algoritmo en función de su técnica de diseño.

Cuando los resultados obtenidos no coinciden con las predicciones teóricas, deberíamos indagar la razón de dichos resultados. Evidentemente, uno o más de los algoritmos presenta errores que debemos identificar y depurar. Puede haber algoritmos inexactos que produzcan resultados "mejores" que los optimales (por tanto, inválidos) o algoritmos exactos que calculen valores suboptimales. En todo caso, los resultados experimentales permiten encontrar casos de prueba que pueden usarse para encontrar las causas de los errores y depurar los algoritmos correspondientes.

En las prácticas 3 y 5 se pedía a los alumnos que informaran de cualquier incidencia con los experimentos. Un buen número de grupos informaron de modificaciones de algoritmos desarrollados debido a los resultados experimentales que estaban obteniendo.

El Cuadro 6 muestra el número de algoritmos modificados en cada práctica, clasificados por técnica de diseño. Las celdas marcadas con un guion corresponden a técnicas de diseño que no se habían visto en la asignatura en el momento de pedir la práctica. La práctica 3 se realizó en dos partes.

Técnica de diseño	Pr. 3a (N=32)	Pr 3b (N=31)	Pr. 5 (N=30)
Heurísticos	6 (19%)	0 (0%)	1 (3%)
Vuelta atrás	6 (19%)	4 (13%)	0 (0%)
Ramificación y poda	–	11 (23%)	1 (3%)
Pr. dinámica – v. recursiva	–	–	2 (6%)
Pr. dinámica – v. tabulada	–	–	1 (3%)

Cuadro 6: Resultados de algoritmos depurados

Puede comprobarse que en las prácticas 3a y 3b más de un tercio de los grupos informó de la necesidad de modificar algún algoritmo. En algunos casos, el error se encontraba en un algoritmo desarrollado en prácticas anteriores, como es el caso de los algoritmos heurísticos (en cualquiera de las tres prácticas), vuelta atrás (en la práctica 3b), o algún algoritmo de búsqueda (en la práctica 5).

En otros casos, vieron la necesidad de depurar los algoritmos que estaban desarrollando. Aparte de comportarse según las predicciones de la teoría, el experimento debía mostrar coherencia entre algoritmos equivalentes. Así, el algoritmo de ramificación y poda debía construirse modificando el algoritmo de vuelta atrás y el algoritmo de programación dinámica se obtiene eliminando la redundancia de un algoritmo recursivo diseñado previamente. Por tanto, si sus resultados no eran iguales, la transformación de un algoritmo en otro no había mantenido la equivalencia.

5. Conclusiones

Hemos realizado un recorrido por la materia de la algoritmia, mostrando que en ella confluyen tres tradiciones intelectuales e identificando la más directamente ligada a cada contenido y a actividades educativas frecuentes. La principal contribución de la comunicación ha sido presentar nuevas actividades basadas en la tradición experimental, pero relacionadas con las otras dos tradiciones. Esta interrelación es una oportunidad para una mejor comprensión de las tradiciones y de la materia por parte de los alumnos. Hemos incluido los datos recogidos en este curso académico en una asignatura de algoritmos de cuarto curso, que arrojan unos resultados medianos.

Tenemos dos conclusiones principales. En aquellos casos donde hay interacción entre teoría y experimentación, se observa que los alumnos presentan un dominio menor de lo deseable de los distintos enfoques y de la materia. Queda como reto para trabajos futuros realizar intervenciones docentes que mejoren estos resultados. Sin embargo, en el caso de la interacción entre experimentación e ingeniería, la experimentación resulta de una gran ayuda a los alumnos para depurar los algoritmos nuevos o, incluso, los algoritmos desarrollados en prácticas anteriores.

Referencias

[1] D. Baldwin. Using scientific experiments in early Computer Science laboratories. En *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1992)*, ACM DL, 1992, pp. 102-106.

[2] R. McCloskey. An analysis of algorithms laboratory utilizing the maximum segment sum problem. *ACM SIGCSE Bulletin*, 27(4):21-26, 1995.

[3] J. W. Coffey. Integrating theoretical and empirical computer science in a data structures course. En *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'13)*, ACM DL, 2013, pp. 23–27. DOI 10.1145/2445196.2445211.

[4] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. B. Tucker, A. J. Turner, y P. R. Young. Computing as a discipline. *Communications of the ACM*, *32*(1), 9–23, 1989. DOI 10.1145/63238.63239.

[5] E. C. Epp. Yet another analysis of algorithms laboratory. *ACM SIGCSE Bulletin*, 24(4):11-14, diciembre 1992.

[6] N. Esteban Sánchez, C. Pizarro y J. Á. Velázquez Iturbide. Evaluation of a didactic method for the active learning of greedy algorithms. *IEEE Transactions on Education*, 57(2):83-91, mayo 2014. DOI 10.1109/TE.2013.2275154.

[7] D. Ginat. Learning from wrong and creative algorithm design. En *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*, ACM DL, 2008, pp. 26–30. DOI 10.1145/1352135.1352148.

[8] P. Ihantola, T. Ahoniemi, V. Karavirta y O. Seppälä. Review of recent systems for automatic assessment of programming assignments. En *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling 2010)*, ACM DL, 2010, pp. 86-93. DOI 10.1145/1930464.1930480.

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

[10] C. Laxer. Treating computer science as science as: An experiment with sorting. En *Proceedings of the 32th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'01)*, ACM DL, 2001, p. 189. DOI 10.1145/507758.377710.

[11] J. Matocha. Laboratory experiments in an algorithms course: Technical writing and the scientific method. En *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference (FIE 2002)*, IEEE Xplore, 2002, pp. T1G 9-13.

[12] T. K. Moore, A. G. Rich y M. R. Vich. Scientific investigation in a breadth-first approach to introductory Computer Science. En *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'93)*, ACM DL, 1993, 63-67. DOI 10.1145/169073.169350.

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

[14] I. Sanders. Teaching empirical analysis of algorithms. En *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*, ACM DL, 2001, 321-325. DOI 10.1145/563517.563468.

[15] S. M. Sarwar, E. E. Parks y S. A. Sarwar. Laboratory exercises for practical performance of algorithms and data structures. *IEEE Transactions on Education,* 39(4):526-531, 1996. DOI 10.1109/13.544807.

[16] C. P. Snow. *The Two Cultures and a Second Look*. Cambridge University Press, 1964. (Existe traducción al español: *Las dos culturas y un segundo enfoque*. Alianza Editorial, 1977.)

[17] J. Á. Velázquez-Iturbide. The design and coding of greedy algorithms revisited. *Proceedings of the 16th Annual Conf. Innovation and Technology in Computer Science Education (ITiCSE 2011)*, ACM DL, 2011, 8-12, DOI 10.1145/1999747.1999753, 2011.

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

[19] J. Á. Velázquez-Iturbide. GreedEx and OptimEx: Two tools to experiment with optimization algorithms. *International Journal of Engineering Education*, 32(3A): 1.097-1.106, 2016.

[20] J. Á. Velázquez-Iturbide. Students' misconceptions of optimization problems. *Proceedings of the 24th Annual Conf. Innovation and Technology in Computer Science Education (ITiCSE 2019)*, ACM DL, 2019, 464-470, DOI 10.1145/3304221.3319749.

[21] J. Á. Velázquez Iturbide y O. Debdi. Experimentation with optimization problems in algorithm courses. En *Proceedings of the International Conference on Computer as a Tool (EUROCON'11)*. IEEE Xplore, 2011. DOI 10.1109/EUROCON.2011.5929294.

[22] S. Wasik, M. Antczak, J. Badura, A. Laskowski y T. Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys*, 51(1): artículo 3, 2018. DOI 10.1145/3143560.