Actas de las Jenui, vol. 4. 2019.
Páginas: 271-278

Pistas en lenguaje natural en jueces en línea

Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid
{pedrop,marcoa}@fdi.ucm.es

Resumen

Con la proliferación del uso de los jueces en línea para el aprendizaje curricular, cada vez es más habitual que los alumnos tengan contacto con este tipo de sistemas en etapas tempranas de su aprendizaje. Por desgracia, los jueces en línea sufren una endémica ausencia de retroalimentación, que puede ocasionar frustración en usuarios noveles. En *¡Acepta el reto!*, un juez en línea con problemas en español, se ha incorporado un sistema piloto de pistas que proporciona ayuda a los usuarios que cometen los errores identificados como más habituales en esos problemas. La ayuda suministrada es en lenguaje natural y, aunque no utiliza análisis de código ni técnicas avanzadas de comparación, puede ser muy efectiva especialmente en aquellos problemas con muchos envíos.

Abstract

Online judges are used more and more for *curricular learning*, so students have contact with this kind of system earlier during their learning process. Unfortunately, online judges suffer a persistent lack of feedback, which can cause frustration in novice users. In *¡Acepta el reto!*, an online judge with problems in Spanish, a new system of hints has been incorporated in order to provide feedback to users who make some common mistakes. Hints are provided in natural language and, although it does not use advanced techniques, the system can be very effective, especially for those problems with many submissions.

Palabras clave

Jueces en línea, enseñanza de la programación, programación competitiva.

1. Introducción

Los jueces en línea son plataformas que publican problemas de programación y a los que los usuarios pueden enviar sus soluciones, que son evaluadas por el sistema en tiempo real para suministrar un veredicto sobre su corrección.

Debido al amplio espectro de problemas que proporcionan, a que permiten soluciones en múltiples lenguajes, y a que utilizan un modelo de evaluación de los envíos basado en análisis dinámico del código, los jueces en línea han sido tradicionalmente muy opacos a la hora de proporcionar retroalimentación a sus usuarios.

Esto no supone un problema para el uso original de este tipo de jueces, que se enfocaban en el entrenamiento de la llamada *programación competitiva*. Sin embargo los últimos años han sido testigos de un incremento en el uso de los jueces en línea en contextos educativos, ya sea como ejercicios de apoyo o como problemas de evaluación continua. Especialmente en el primero de los dos casos, la ausencia de retroalimentación se convierte en un hándicap para su uso, dado que los estudiantes se sienten a menudo perdidos ante los veredictos negativos.

Para paliar este problema este artículo describe un nuevo *sistema de pistas* incorporado en *¡Acepta el reto!*, un juez en línea con problemas en español, que permite proporcionar ayuda a los usuarios cuando cometen errores identificados como habituales.

La sección siguiente describe lo que es un juez en línea y su funcionamiento básico. A continuación se enumeran los diferentes tipos de ayuda proporcionados en ellos y por qué resulta complejo incorporar sistemas de retroalimentación más elaborados. Tras esto, las secciones 4 y 5 describen el juez *¡Acepta el reto!* y su sistema de pistas. El artículo termina con una breve evaluación y unas conclusiones.

2. Trabajo relacionado

La programación y la algoritmia son dos de los puntos clave de los grados de Informática, y constituyen

también un aspecto importante de otras titulaciones como Matemáticas o Ciencias Físicas. Incluso si no se considera su uso profesional, el *pensamiento computacional* [12] empieza a ser considerado lo suficientemente importante como para ser incorporado en los primeros niveles educativos.

Hoy son universalmente conocidas herramientas como Scracth [10] y Alice [4] que han generado a su alrededor comunidades de usuarios a lo largo y ancho de todo el mundo, en las que se involucran no solo alumnos, sino también profesores, padres y pedagogos. Y es que aprender programación no es fácil; requiere de ayuda y monitorización constante. Con el tiempo, han surgido diferentes sistemas que han buscado proporcionar ayuda de forma automática al estudiante en función de sus errores. Éste es el campo de los llamados *Tutores Inteligentes* (ITS, por sus siglas en inglés, *Intelligent Tutoring Systems*), como LISP Tutor [2].

El principal problema de este tipo de sistemas es que requieren un gran esfuerzo de autoría. El resultado es una construcción muy focalizada, a través de tareas de pequeña magnitud para los que existe una solución de referencia hacia la que se guía al alumno.

Una aproximación opuesta es abrir el abanico de ejercicios disponibles, a costa de reducir, o incluso eliminar completamente, la ayuda proporcionada a los estudiantes. Es la tónica general en los *jueces en línea* [9] (*online judges*, OJ), muchos de ellos nacidos al calor de los concursos de programación.

Un juez en línea es una plataforma software que dispone de una batería de problemas de programación, junto con un mecanismo para probar soluciones enviadas por los usuarios y emitir un veredicto sobre su corrección. El modo de hacer esa evaluación se basa en la existencia de baterías de pruebas, normalmente secretas, a las que se hace pasar a las soluciones enviadas. En particular, los problemas propuestos deben leer de la entrada estándar un conjunto de *casos de prueba*, y generar, por la salida estándar, un resultado para cada uno de ellos. El sistema de análisis de la solución realiza *análisis dinámico* del código, de tal forma que lo compila y ejecuta enviando los casos de prueba al proceso, y recibiendo su salida. Ésta es comparada con el resultado esperado, conseguido con una solución correcta de referencia, y dependiendo del resultado de la comparación se emite un veredicto u otro.

Este mecanismo de evaluación es también el utilizado en muchos de los concursos de programación existentes. El más antiguo es el ACM/ICPC (*International Collegiate Programming Contest*), enfocado en alumnos universitarios y que en 2017 superó los 50.000 participantes [1]. Para secundaria, existe desde 1989 la Olimpiada Internacional de Informática (*International Olympiad in Informatics*, IOI), impulsada por la UNESCO, en la que en 2015 participaron equipos de 83 países distintos [8].

Ambos concursos utilizan el mismo mecanismo de evaluación, aunque difieren en la riqueza de los veredictos yen el sistema de puntuación. En el caso del ACM/ICPC, el resultado del envío de una solución es binario en el sentido de que o bien es totalmente correcta, o bien se considera inválida. En este último caso, el sistema de evaluación proporciona un veredicto u otro en función del resultado de la ejecución, que puede dar una pista a los participantes sobre lo que está ocurriendo. Los veredictos más habituales son:

- *Aceptado* (AC): la solución enviada genera la salida esperada, por lo que se considera correcta.
- *Respuesta incorrecta* (WA, *Wrong Answer*): la salida no encaja con la esperada.
- *Error de ejecución* (RTE, *Runtime Error*): el programa falló durante la ejecución y no llegó a terminar. Es el veredicto habitual cuando un programa en C/C++ realiza una operación inválida, o uno en Java genera una excepción no controlada.
- *Límite de tiempo excedido* (TLE, *Time Limit Exceeded*): el programa estaba tardando demasiado tiempo en ejecutarse, y fue detenido antes de que pudiera terminar.
- *Límite de memoria excedido* (MLE, *Memory Limit Exceeded*): el programa solicitó demasiada memoria, y fue detenido antes de terminar.
- *Error de compilación* (CE, *Compilation Error*): el código enviado ni siquiera ha llegado a compilar.

Los mencionados jueces en línea proporcionan veredictos similares; no en vano estos sistemas nacieron a la estela de los propios concursos para conservar los problemas que se creaban para ellos y permitir a futuros participantes entrenar. Es ese el caso del primer juez en línea de fama mundial, UVa OJ[1], nacido en la Universidad de Valladolid y que fue gestionado durante muchos años por el profesor Miguel A. Revilla [11]. Su objetivo principal es el de la *programación competitiva*, poniendo más el foco en el entrenamiento para concursos y no tanto en el aprendizaje.

Con el tiempo han surgido muchos otros jueces como URI OJ [3], *TopCoder*[2] o *Codeforces*[3]. Algunos de ellos intentan separarse de la programación competitiva y acercarse a la enseñanza curricular. Para eso, incorporan recomendadores de ejercicios, información sobre la dificultad, o incluso módulos para gestionar clases que dan privilegios especiales a los profesores. Pero, en la práctica, mantienen una fuerte dependencia con el sistema de evaluación que hace difícil proporcionar ayuda personalizada, que sigue siendo una de sus grandes carencias [5].

[1] `http://uva.onlinejudge.org/`
[2] `https://www.topcoder.com/`
[3] `http://codeforces.com/`

3. Ayuda en jueces en línea

Por lo general, los jueces en línea son muy opacos con respecto a las pruebas que realizan, de modo que cuando un usuario hace un envío y recibe un veredicto de no aceptación, no cuenta con demasiada información que le ayude en la búsqueda del error. Para paliar esta carencia, se pueden seguir varias aproximaciones:

- *Uso de foros:* este mecanismo es seguramente el más habitual. Asociado a cada problema se crea un hilo de discusión en el que los usuarios pueden pedir y proporcionar ayuda.
- *Ayuda general de los autores:* los creadores de los problemas pueden proporcionar ayuda adicional, más allá del enunciado del problema, que guíe hacia la solución a aquellos usuarios que no sean capaces ni siquiera de vislumbrar una.
- *Pistas de la comunidad*: equivalente al anterior, pero creados por la propia comunidad de usuarios del juez en línea. A menudo ocurre a través de *categorías* o *etiquetas*, pues el mero hecho de saber que un problema pertenece, por ejemplo, a la categoría de programación dinámica puede ser lo bastante aclaratorio para los usuarios.
- *Ejecución de la solución de referencia:* en ocasiones, la causa del error no está en una implementación incorrecta, sino en una comprensión equivocada de lo que se está pidiendo. Algunos sistemas disponen de la posibilidad de enviar al juez *casos de prueba personalizados* que son ejecutados por una solución de referencia. La salida es enviada de vuelta al usuario, que puede comprobar si lo recibido coincide con sus expectativas.
- *Acceso a los casos de prueba secretos:* normalmente los jueces en línea mantienen secretos los casos de prueba. Solo unos pocos los publican, permitiendo a los usuarios analizar en qué punto está fallando su solución. Pese a su valor como fuente de ayuda, no puede en modo alguno considerarse ayuda personalizada. Además, su uso es muy tedioso, porque si bien es fácil detectar que la salida de la solución no encaja con la esperada, puede llegar a ser muy complicado averiguar cuál es la entrada asociada a la salida que falla.

Para conseguir que un juez en línea proporcione *ayuda personalizada* para las soluciones incorrectas, se puede intentar hacer uso de las técnicas de análisis estático de código utilizadas en los tutores inteligentes. Por desgracia, esto no es viable por varias razones:

- Los jueces en línea permiten múltiples lenguajes de programación. Crear herramientas de análisis para todos ellos suele ser impráctico. Incluso aunque los lenguajes fueran lo suficientemente parecidos como para que pudiera reutilizarse parte del trabajo (por ejemplo C++ y Java), lo habitual es que las *bibliotecas* asociadas a cada lenguaje sean muy diferentes, e incorporar todo ese conocimiento multiplica la dificultad.
- Los usuarios a menudo envían soluciones con código poco estructurado. El objetivo principal de estos sistemas es la algoritmia y no la programación limpia y estructurada. Muchos usuarios dejan estos aspectos de lado para favorecer o bien la velocidad de programación, o bien la velocidad de ejecución, dificultando más el análisis estático.
- Los problemas pueden carecer de solución de referencia con la que comparar las enviadas por los usuarios. En su mínima expresión, un problema para un juez en línea está compuesto de un enunciado, y una batería de casos de prueba especificados a través de ficheros de entrada y de salida. Si bien para conseguir los ficheros de salida debió usarse, en el momento de su creación, una solución de referencia, ésta puede no estar disponible, especialmente para aquellos problemas más antiguos. Incluso si existiera, un mismo problema puede tener diversas soluciones igualmente válidas. Por ejemplo, la detección de la palindromía de una frase puede implementarse con un bucle o de forma recursiva, y ninguna de las dos soluciones es necesariamente mejor que la otra.

4. El juez en línea *¡Acepta el reto!*

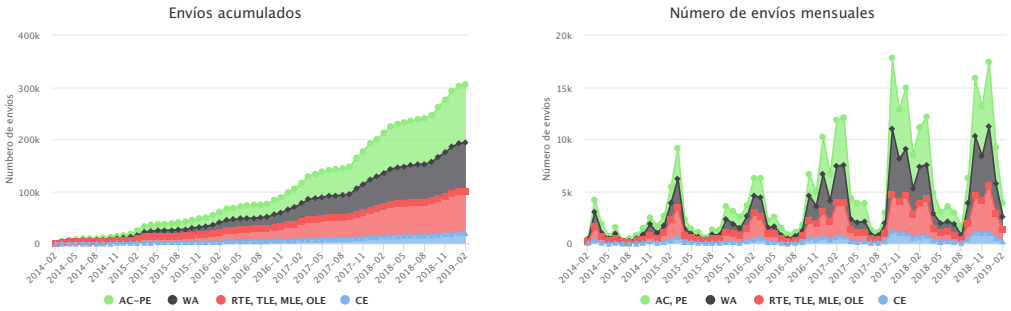
¡Acepta el reto![4] [7] es un juez en línea con más de 400 problemas de programación en español. Nació en 2014 para servir como *archivo vivo* de los problemas de programación creados por sus autores para diferentes actividades, como evaluación continua en clase [6] o ProgramaMe[5], el concurso español de programación para Ciclos de Formación Profesional.

El abanico del nivel de dificultad de los problemas es amplio, desde problemas simples de variables y expresiones para los primeros días de clase, hasta problemas que requieren algoritmos sobre grafos o teoría de números. Admite soluciones en C, C++ y Java y para cada uno informa sobre el tiempo y la memoria total consumida.

Las características distintivas de *¡Acepta el reto!* frente a otros jueces son:

- Los enunciados de los problemas están en español. Esto elimina la barrera del idioma para muchos estudiantes de los primeros cursos.
- Los problemas están *categorizados* en función de diferentes ejes. Algunos tienen una relación directa con el currículo de las asignaturas de programa-

[4] `https://www.aceptaelreto.com`
[5] `http://www.programa-me.com`



(a) Envíos acumulados por veredicto.

(b) Número de envíos mensuales por veredicto.

Figura 1: Número de envíos de *¡Acepta el reto!*

ción o de algoritmia, lo que facilita a profesores y alumnos encontrar problemas para poner en práctica los conceptos que están viendo en clase.

- Muchos problemas tienen objetivos pedagógicos específicos de modo que los límites de tiempo y memoria se afinan para ellos. Los jueces en línea tienen restricciones en tiempo que permiten limitar la complejidad asintótica de sus soluciones. No obstante, dado que los autores de los problemas y los administradores de los jueces no son los mismos, no es extraño que los límites no se ajusten con finura y terminen siendo válidas soluciones que los autores habrían preferido dejar fuera. En el caso de *¡Acepta el reto!* se presta especial cuidado a este asunto. Además, se pueden forzar también soluciones con complejidades *en memoria* acotadas, algo que no ocurre en otros jueces.

Los veredictos proporcionados por el juez son los descritos en la sección 2, con la inclusión de dos más:

- *Error de presentación* (PE, *Presentation Error*): la salida es correcta salvo por los separadores. Ocurre, por ejemplo, si el programa tiene espacios o saltos de línea sobrantes.
- *Límite de salida excedida* (OLE, *Output Limit Exceeded*): el programa generó demasiada salida y fue detenido antes de terminar.

El juez recibió su primer envío el 17 de febrero de 2014, y desde entonces el número de problemas, usuarios y envíos no ha dejado de crecer. En enero de 2019 recibió su envío número 300.000 de un total que supera los 11.000 usuarios de diferentes países hispanohablantes. La figura 1a muestra la evolución a lo largo del tiempo del número de envíos recibidos y sus veredictos. La figura 1b muestra la misma información, pero sin acumular los envíos anteriores, lo que muestra una clara temporalidad en el uso del juez, con muchos más envíos durante el primer semestre escolar.

Para evitar sobrecargar las figuras, los veredictos de

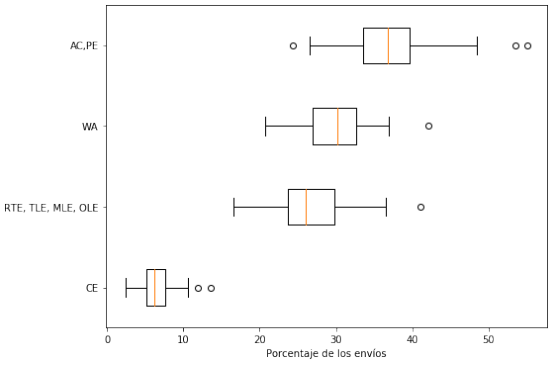


Figura 2: Distribución de los veredictos mensuales

aceptación y error de presentación (muy minoritarios) se muestran agrupados. También se agrupan todos los errores ocasionados por una finalización abrupta de la ejecución, ya sea por un error de programación o por la superación de alguno de los límites impuestos a la ejecución.

Es interesante hacer un análisis sobre la frecuencia de aparición de cada conjunto de veredictos para conocer el perfil de los errores. La figura 2 muestra un diagrama de cajas que pone de manifiesto que el veredicto de respuesta incorrecta (WA) es el más habitual de los veredictos de error, máxime cuando el siguiente bloque agrupa cuatro veredictos en su interior.

Para probar las soluciones, los problemas incorporan baterías de casos de prueba extensas, con algunos problemas superando los 100.000 casos. Para que sea viable la ejecución de las soluciones con esas cifras, prácticamente todos los problemas son *multicaso*, es decir las soluciones deben ser capaces de responder a múltiples casos de prueba *en una única ejecución*, algo muy habitual también en otros jueces y concursos.

Aun así, en *¡Acepta el reto!* las soluciones son ejecutadas, al menos, tres veces, con tres *ficheros de casos de prueba* distintos:

- Casos de prueba de ejemplo: todos los enunciados incorporan un ejemplo de entrada y de salida, para que el lector confirme que ha comprendido bien lo que se le está pidiendo. No pretenden servir para probar casos límite o confirmar que las soluciones funcionan, pues el número de casos que incluye es muy reducido (entre 2 y 5). En la primera ejecución de una solución, el juez confirma que el programa supera esos casos de prueba de ejemplo.
- Entrada vacía: como las soluciones son multicaso, una configuración particular de la entrada es que no incorpore ningún caso. Todos los problemas son probados bajo esta premisa, y es relativamente habitual la recepción de soluciones que fallan únicamente con esta entrada.
- Primera batería de casos de prueba secreta: normalmente incluye un juego de casos de prueba exhaustivo, que prueba con insistencia la solución con casos de reducido tamaño.

En la mayoría de los problemas existe más de una batería de casos secreta, por lo que las soluciones recibidas son ejecutadas 4, 5 o hasta 8 veces con casos de diferente tamaño o persiguiendo detectar soluciones poco eficientes en tiempo o en espacio.

5. Sistema de pistas

A la vista de los datos del histórico de envíos, parece adecuado centrar los primeros esfuerzos en la inclusión de ayuda en los veredictos de *Respuesta incorrecta* (WA) por ser los más habituales. Una primera posibilidad sería intentar identificar los *casos de prueba* concretos que han generado una discrepancia entre la salida esperada y la obtenida, y mostrar al usuario un subconjunto de ellas.

Si bien es relativamente habitual que los usuarios soliciten esta característica, tiene varios puntos débiles:

- Conseguir los casos de prueba que fallan es difícil técnicamente. Dado que las soluciones enviadas procesan múltiples casos de prueba en una única ejecución, identificar cuáles de todos ellos han generado salida incorrecta no es trivial, porque es necesario crear la relación entre la entrada de cada caso y la salida generada por el programa para ella. La existencia de *buffers en memoria* para la salida hace complicado averiguar esa relación desde el exterior del propio proceso. Incluso en problemas de salida previsible donde cada caso de prueba genera una única línea de salida, asociar cada línea a un caso no es completamente fiable, pues es relativamente habitual que las soluciones rompan el formato de la salida (olvidando escribir el salto de línea) o que incluso se salten casos.
- Ante esta situación, una alternativa por "fuerza bruta" para averiguar los casos fallidos es ejecutar el programa una vez por cada caso individual. Por desgracia, esto no es viable en juegos de problemas en los que el número de casos de prueba es tan numeroso como en *¡Acepta el reto!*. De hecho, los casos de prueba se agrupan en una única ejecución precisamente por razones de eficiencia, para evitar tener que hacer esto. Además, tampoco sería una solución completamente fiable. Es relativamente habitual la llegada de soluciones que tienen *dependencias entre casos*, en las que la salida proporcionada ante un caso de prueba depende, incorrectamente, de los que se hayan procesado anteriormente. Con soluciones así, la ejecución independiente de cada caso de prueba podría no encontrar un caso de prueba incorrecto y la mera idea de proporcionarlo es inútil fuera del contexto de los casos de prueba previos.
- Incluso si fuera viable averiguar los casos de prueba fallidos, su utilidad no siempre estaría asegurada. Algunos problemas tienen casos de prueba muy grandes como para ser "procesados por un humano". Esto ocurre en problemas que reciben en la entrada listas de números muy largas, o grafos muy grandes.
- La supuesta ayuda personalizada a partir de los casos de prueba fallidos es, en muchos casos, inútil desde su concepción. Sorprendentemente, muchas de las soluciones incorrectas que recibe el juez fallan *en el ejemplo del enunciado*. Los usuarios *tienen acceso* al primer caso de prueba que falla y, aun así, envían la solución incorrecta.

A la vista de todo esto, se necesita una aproximación diferente, que trate el resultado de la ejecución en conjunto en lugar de comprobar los detalles de cada caso de prueba por separado. Lo que hacemos en *¡Acepta el reto!* es considerar que dos soluciones son iguales si *generan exactamente la misma salida* para todos y cada uno de los casos de prueba del problema. Esto evita entrar en el análisis estático de código esbozado en la sección 3, manteniendose así aséptico al lenguaje de programación, optimizaciones e incluso el algoritmo particular usado.

La hipótesis del sistema es que dos soluciones que generan exactamente la misma salida (incorrecta) para la misma entrada (casos de prueba) sufren el mismo *error conceptual*. Las soluciones quedan agrupadas por la salida independientemente de sus detalles de implementación, pues podrían estar incluso programadas en lenguajes distintos. Asumiendo un juego de casos de prueba lo suficientemente rico, la coincidencia en la salida debería ser una indicación lo bastante fiable de que la razón de fondo del error es la misma.

Bajo esta hipótesis, tiene sentido analizar una úni-

ca solución de todas las que compartan salida, y crear, manualmente, una *pista* para ella, que será reutilizable para todas las demás. Esa ayuda debe escribirse en términos del error cometido *desde el punto de vista del problema*, no desde el punto de vista del código concreto. De otro modo, la ayuda sería solo aplicable a aquellas soluciones que tengan ese error de programación particular, reduciendo mucho su utilidad. A modo de ejemplo, una buena pista asociada a una salida particular sería "Tu solución falla cuando el denominador es negativo" en lugar de "Tienes mal el `if`".

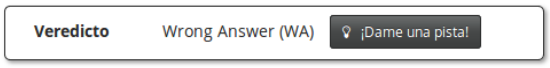
Las ventajas de este sistema de pistas son:

- Un mismo error de programación puede afectar a múltiples casos de prueba. En el modelo de buscar y mostrar los casos de prueba que fallan se puede terminar dedicando un gran esfuerzo computacional a esa búsqueda, que luego el usuario podría no necesitar.
- Reduce la necesidad de depuración. El texto de las pistas suele poner el foco en las características de los casos de prueba fallidos, por lo que el usuario puede analizar su código pensando en ellos, sin necesidad de ejecutarlo. Esto es especialmente interesante cuando los casos que fallan son largos.
- Al ser en lenguaje natural, son mucho más fáciles de comprender que los casos de prueba fallidos. Esto es particularmente cierto entre los estudiantes más noveles, que tienen dificultades incluso para probar sus propias soluciones como pone de manifiesto el alto porcentaje de envíos en los que falla el ejemplo.

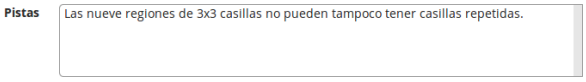
Naturalmente, no está exento de desventajas:

- Hay una fuerte dependencia con los casos de prueba. Si la batería de casos de prueba es débil, soluciones con errores conceptuales diferentes podrían acabar en la misma categoría por tener la misma salida, dificultando la escritura del texto asociado a la pista.
- Al ser manual, la creación de las pistas es tediosa.
- La ayuda proporcionada *no* es específica del código escrito por el usuario. Si bien esto también se considera una virtud, los usuarios noveles suelen requerir ayuda más particular sobre su código que el sistema es incapaz de dar.

A nivel de implementación, hay que tener en cuenta que la salida generada por las soluciones puede ser arbitrariamente larga. Guardar la salida completa generada por todas las soluciones recibidas, y compararlas con aquellas para las que se ha creado una pista es inviable tanto en tiempo como en espacio. La solución adoptada en *¡Acepta el reto!* es hacer uso de *funciones hash*, como *SHA256* o la más modesta y antigua *MD5*. Esta última ha mostrado algunas deficiencias en ciertos contextos por culpa de la posibilidad de forzar una colisión, si bien es algo improbable en el contexto de un juez en línea donde la batería de los casos de prueba es secreta.



(a) Botón para solicitar una pista.



(b) Pista proporcionada.

Figura 3: Interfaz del sistema de pistas

Como ya se ha comentado, es relativamente habitual que las soluciones recibidas fallen en la escritura de los separadores, olvidando cambiar de línea entre casos de prueba, o escribiendo espacios sobrantes al final del último valor de una lista, por ejemplo. Debido a ello, la función *hash* no la aplicamos directamente sobre la salida generada por los envíos, sino por una versión manipulada en la que han sido eliminados los espacios, tabuladores y saltos de línea. Salvo contadas ocasiones, una pista es útil para dos soluciones que generen exactamente la misma salida salvo por los separadores, e ignorando esos separadores en la agrupación de las soluciones en función de la salida conseguimos que todas reciban la misma pista. Posteriormente, cuando los usuarios arreglen el error indicado y reenvíen su solución, algunos mantendrán el error por culpa de los separadores en cuyo caso antes de conseguir el veredicto de éxito pasarán por el de *Error de Presentación* (PE) que es una pista en sí mismo.

Cuando un usuario envía una solución que genera una salida coincidente con una en la que se ha incorporado una pista, el sistema le da la opción de solicitar una pista (figura 3a). Si la solicita, junto a los datos de su envío aparece una nueva sección. En la figura 3b se muestra como ejemplo una de las pistas asociadas a un problema que pide comprobar si un sudoku es correcto.

6. Evaluación

Para probar el sistema, en *¡Acepta el reto!* se han añadido pistas en 37 problemas de diferente nivel de dificultad, de los más de 400 disponibles. El número es reducido por varias razones:

- Incorporar pistas es costoso en tiempo: analizar un envío hasta saber la razón que lo hace fallar es sencillo en los problemas más simples, pero resulta cada vez más peliagudo al ir subiendo el nivel de complejidad de los poblemas.
- Para que el tiempo invertido en plantear una pista sea rentable, es preferible hacerlo en los errores

"más populares". Como autores de problemas, anticipar los errores más comunes que los usuarios cometerán en ellos ha demostrado ser sorprendentemente difícil; los alumnos tienen unas maneras muy imaginativas de equivocarse. Según nuestra experiencia, es mejor esperar a que un problema reciba envíos reales para detectar esos errores más comunes, y poner pistas a esos. Por tanto, los problemas que tienen pocos envíos incorrectos, o están muy segmentados, no son (aún) buenos candidatos para recibir pistas.

- *¡Acepta el reto!* es utilizado, por otros profesores y por nosotros mismos, dentro del proceso de evaluación continua de nuestros alumnos. Incorporar pistas de forma indiscriminada a todos los problemas puede dañar ese caso de uso del juez. Intentamos incorporar pistas en aquellos problemas que llevan tiempo publicados en el juez, entendiendo que por ello será menos probable que sea utilizado para evaluar, y tenderá a un uso personal o para trabajo no evaluado. Esto se alinea con el punto anterior, pues cuanto más tiempo lleve un problema en el sistema más probabilidad hay de que el número de envíos erróneos sea lo suficientemente grande como para sacar a la luz los errores más habituales.

Para la creación de las pistas, hacemos un análisis previo de las *hashes* de los envíos incorrectos más habituales de los problemas más antiguos, y nos centramos en ellos. Para cada uno, buscamos los envíos de usuarios que han generado esa salida y han sido seguidos inmediatamente después por un envío correcto de ese mismo usuario. Cuando eso ocurre, las diferencias entre esos dos envíos consecutivos son de gran ayuda para saber, sin tener que analizarlo, dónde estaba el error de la solución incorrecta.

Si el número de envíos similares es suficientemente alto, para aumentar nuestra confianza en la pista hacemos esta revisión con otras parejas de envíos equivalentes de otros usuarios. Esto nos permite confirmar que el texto de la pista es, en efecto, lo suficiente general como para que sirva para múltiples envíos de los que han generado la misma salida. De otro modo, se corre el riesgo de que quede demasiado sesgado a una materialización particular.

Como ejemplo de las diferencias entre las soluciones que pueden ocasionar la misma salida, se muestran a continuación un par de envíos a un problema de comprobar la palindromía de una frase de forma insensible a separadores y mayúsculas. Se muestra una versión recortada y adaptada, eliminando por brevedad la parte del procesamiento multicaso y resto de "infraestructura" del código, como la definición de la clase.

```
private static boolean casoDePrueba() {
  String t = entrada.nextLine();
  if (t.equals("XXX"))
    return false;

  t = t.toUpperCase().replace(" "," ");
  String textoInverso = "";

  for (int i = 0; i < t.length(); i++)
    textoInverso += t.charAt(i);

  if (t.equals(textoInverso))
    System.out.println("SI");
  else
    System.out.println("NO");

  return true;
}
```

Tras leer la cadena y confirmar que no es la que marca el final de la entrada, elimina los espacios y convierte a mayúsculas. Después "da la vuelta" a la frase resultante, acumulando en una variable auxiliar el resultado. Un lector acostumbrado a revisar código, habrá apreciado que el bucle es incorrecto, porque se realiza de izquierda a derecha, por lo que en la práctica la cadena `tInv` es la misma que `t` y por tanto la solución dice que todas las palabras son palíndromas.

A continuación se muestra la versión recortada de otra solución al mismo problema. En este caso, el código está en C++ y se omite, por brevedad, incluso la lectura de la frase:

```
bool palAux(string v, int ini, int fin)
{
  if (ini <= fin)
    return true;
  else {
    v[ini] = tolower(v[ini]);
    v[fin] = tolower(v[fin]);
    if (v[ini] == v[fin])
      return palAux(v, ini+1, fin-1);
    if (v[ini] == ' ')
      return palAux(v, ini+1, fin);
    if (v[fin] == ' ')
      return palAux(v, ini, fin-1);
    else
      return false;
  }

}

bool palindroma(string v) {
  return palAux(v, 0, (v.length() - 1));
}
```

En este caso, el alumno se decantó por una solución recursiva. Ahora el error es más sutil. La comprobación del caso base (`ini <= fin`) es incorrecta, pues debería ser mayor o igual. Tal y como está escrita, la condición resulta ser siempre cierta y por tanto, de nuevo, la solución cataloga como palíndromas todas las frases.

El ejemplo demuestra que dos soluciones diametralmente distintas sufren errores particulares de programación que, a pesar de ello, llevan a ambas a generar la misma salida. En esta situación, la pista incorporada fue "Algo falla en tu comprobación. ¡Para ti todas las palabras son palíndromas!".

El análisis preliminar realizado sobre el uso de las pistas por parte del usuario muestra resultados prometedores. Para ser considerada acertada, una pista debe guiar correctamente al usuario hacia la solución. Hemos analizado para cada tipo de error que ha recibido una pista cuánto tardan los usuarios en alcanzar la solución correcta en el caso de consultar la pista y en el caso de no consultarla. Los resultados obtenidos indican que un gran porcentaje de los usuarios que consultan las pistas alcanzan la solución correcta en el envío siguiente, mientras que entre aquellos que no la consultan el número de pasos requeridos es mayor, llegando incluso a situaciones extremas de usuarios que necesitan hasta 25 envíos demostrando, no solo que las pistas acortan la frustración, sino que hay algunos usuarios que no desesperan frente a la adversidad.

7. Conclusiones y trabajo futuro

En este artículo se ha presentado un sistema de pistas incorporado en *¡Acepta el reto!* que ha proporcionado resultados prometedores. Este sistema es lo suficientemente general como para que pueda ser incorporado sin demasiada dificultad en otros jueces existentes.

No obstante, se halla aún en una etapa preliminar de la implantación, y todavía deben hacerse pruebas más exhaustivas para confirmar su validez. En particular, sería interesante hacer un análisis sobre la utilidad en problemas con diferentes niveles de dificultad, distintas temáticas, o diferente grado de dispersión en las soluciones. Esto último es especialmente importante, porque, en su estado actual, las pistas no son transferibles entre distintos tipos de error y deben hacerse individualmente para cada uno.

Una opción que está abierta es permitir que sean los propios usuarios los que escriban las pistas. Desde hace algunos meses, los usuarios que consiguen un envío correcto (en problemas con pistas) tienen la posibilidad de *proponer pistas* para sus envíos erróneos anteriores. El número de pistas recolectadas no es demasiado alto y no siempre están redactadas de manera lo suficientemente aséptica del código como para que sean extrapolables a usuarios futuros. No obstante, tenemos pendiente un análisis más elaborado de todas ellas.

Por último, se han dejado de lado completamente los envíos que sufren un veredicto de error diferente al de *Respuesta incorrecta*. Para algunos de ellos, la búsqueda del último caso de prueba ejecutado, descrito en la sección 5 es una opción muy interesante a ser explorada.

Referencias

[1] ACM/ICPC. *Fact Sheet – The 43st Annual World Finals of the ACM Internacional Collegiate Programming Contest (ICPC)*, Diciembre 2018.

[2] John R. Anderson y Brian J. Reiser. The LISP Tutor: It approaches the effectiveness of a human tutor. *BYTE*, 10(4):159–175, Abril 1985.

[3] Jean Luca Bez, Neilor A. Tonin, y Paulo R. Rodegheri. URI Online Judge Academic: A tool for algorithms and programming classes. *2014 9th International Conference on Computer Science & Education*, páginas 149–152, 2014.

[4] Stephen Cooper, Wanda Dann, y Randy Pausch. Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, Abril 2000.

[5] Ágnes Erdősné y László Zsakó. Grading systems for algorithmic contests. *Olympiads in Informatics*, 12, 2018.

[6] Marco Antonio Gómez Martín y Pedro Pablo Gómez Martín. Uso de software de gestión de concursos de programación para evaluación continua. In *Actas de las XIX Jornadas de la Enseñanza Universitaria de la Informática*, Jenui 2013, 2013.

[7] Pedro Pablo Gómez-Martín y Marco Antonio Gómez-Martín. ¡Acepta el reto!: juez online para docencia en español. In *Actas de las XXIII Jornadas de la Enseñanza Universitaria de la Informática*, Jenui 2017, páginas 289–296, 2017.

[8] Artem Iglikov, Mansur Kutybayev, y Bakhyt Matkarimo. IOI 2015 report. *Olympiads in Informatics*, 10, 2016.

[9] Andy Kurnia, Andrew Lim, y Brenda Cheang. Online judge. *Comput. Educ.*, 36(4), Mayo 2001.

[10] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, y Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Noviembre 2009.

[11] Miguel Á. Revilla, Shahriar Manzoor, y Rujia Liu. Competitive learning in informatics: the UVa online judge experience. En Elena Verdú, Rubén M. Lorenzo, Miguel Á. Revilla, y Luisa M. Regueras, editores, *A New Learning Paradigm: Competition Supported By Technology*. Sello Editorial, 2010.

[12] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, Marzo 2006.