Una plataforma de evaluación automática con una metodología efectiva para la enseñanza/aprendizaje en programación de computadores

An automatic evaluation platform with an effective methodology for teaching/learning computer programming

Jorge López Reguera[1] Cecilia Hernández Rivas[1] Yussef Farran Leiva[1]

Recibido 29 de marzo de 2010, aceptado 14 de abril de 2011
Received: March 29, 2010 Accepted: April 14, 2011

RESUMEN

Aprender a programar computadores es un proceso difícil para los estudiantes novatos y un desafío a las metodologías empleadas por los docentes. En este artículo se presenta una plataforma de evaluación automática que apoya el proceso de enseñanza/aprendizaje en cursos introductorios de programación de computadores para estudiantes de ingeniería de la Universidad de Concepción. Esta plataforma utiliza mecanismos que combinan análisis estático/dinámico y aplican evaluación de comprensión/análisis en línea, permitiendo una retroalimentación personalizada a los alumnos. En este trabajo se presentan los criterios usados para crear secuencias didácticas de problemas y una forma de aplicarlas efectivamente mediante la evaluación automática. Los resultados obtenidos después de seis años de aplicación muestran que la evaluación automática afecta positivamente la motivación, el desempeño y la autoeficacia de los alumnos. La metodología utilizada para estudiar la efectividad de la plataforma incluye análisis cualitativo y cuantitativo. Los aspectos cualitativos se extraen mediante la observación del comportamiento de los estudiantes durante el proceso de aprendizaje, mientras que el análisis cuantitativo está basado en los datos de los estudiantes registrados por la plataforma. Con el fin de acumular información cuantitativa, aplicamos distintos experimentos basados en grupos de estudiantes de control y grupos experimentales.

Palabras clave: Evaluación automática, aprendizaje activo, autoeficacia, modelos mentales, análisis estático.

ABSTRACT

Learning to program is a difficult process for novice students and challenging for teaching methods. In this paper we present an automatic evaluation platform and a methodology to support the teaching/learning process in computer programming introductory courses for engineering students at the Universidad de Concepción. This platform uses static/dynamic techniques and applies comprehension /analysis evaluation enabling personalized feedback to students. This work includes the criteria used to create didactic problems sequences and effective ways to apply them through automatic evaluation. The results obtained during six years of application show that automatic evaluation along with the proposed methodology improves student's motivation, performance and self-efficacy. The methodology used to study the effectiveness of the platform includes qualitative and quantitative analysis. The qualitative aspects are extracted by observing student behavior during the learning process, whereas the quantitative analysis is based on student data registered by the platform. In order to gather quantitative information, we applied different experiments based on control and experimental student groups.

Keywords: Automatic assessment, active learning, self efficacy, mental models, static analysis.

[1] Departamento de Ingeniería Informática y Ciencias de Computación. Facultad de Ingeniería. Universidad de Concepción. Concepción, Chile. E-mail: jorlopez@udec.cl; cecihernandez@udec.cl; yfarran@udec.cl

INTRODUCCIÓN

En los cursos introductorios de programación, los profesores inician a los estudiantes en la disciplina de computación y en el proceso de resolución de problemas algorítmicos. Aprender a programar es una tarea compleja que requiere, primero, entender el problema y luego diseñar un algoritmo que represente los pasos de solución para implementarlo en un lenguaje de programación. La construcción de la solución en un lenguaje de programación es un proceso cíclico que consiste en codificar, compilar, probar y depurar programas. Para comprobar que un programa está correcto, éste debe pasar exitosamente varios casos de prueba. Cada caso de prueba consiste en ejecutar el programa usando un conjunto de datos de entrada y luego analizar los resultados.

Para avanzar en el proceso de aprendizaje, los alumnos a menudo requieren apoyo de los docentes; sin embargo, con cursos numerosos, la realización de estas tareas impone una elevada demanda de tiempo, lo cual dificulta la retroalimentación oportuna a los estudiantes.

Evaluar el aprendizaje en programación es un proceso que involucra principalmente verificar que los programas construidos por los alumnos operan correctamente (ejecución de varios casos de prueba) y obtener información desde el código fuente de los programas (estilo de codificación, patrones de error, métricas de software, evaluar diseño, detectar plagio, etc.) [14]. Adicionalmente hay que considerar que para un problema particular es posible construir diferentes programas que lo resuelven y no existen reglas que permitan convertir uno en otro. Estos factores limitan la cantidad de tareas y exámenes que se pueden realizar en el contexto de un curso, produciendo escalones de complejidad que pueden impactar negativamente la ***autoeficacia*** (confianza en sí mismo para solucionar problemas) de los alumnos, y que los docentes puedan observar y apoyar adecuadamente la correcta formación del ***modelo mental*** (idea) del estudiante.

Actualmente existen varias líneas de trabajo destinadas a apoyar el aprendizaje en programación [14-15]. Algunas se orientan a los aspectos cognitivos del aprendizaje [7-11, 16] y otras a desarrollar herramientas de evaluación automática. Sistemas tales como TRY [4], PSGE [3], TRAKLA [12], CourseMaker [13], y AutoLEP [2] usan mecanismos para ejecutar los programas de los estudiantes sometiéndolos a diferentes casos de prueba, en un ciclo que permite varios intentos. Estos sistemas usan mecanismos destinados a forzar al alumno a pensar mejor sus respuestas evitando que pasen a trabajar en una modalidad de *prueba-y-error,* limitando el número máximo de intentos o disminuyendo la retroalimentación en la medida que sobrepasan cierto umbral. La retroalimentación es complementada mediante herramientas que realizan una *prueba estática*, que consiste en realizar una verificación del estilo y aplicación de métricas de la estructura de los programas fuente del alumno, o evaluar su similitud con programas modelo construidos por los profesores. Las descripciones de estas herramientas no profundizan en aspectos metodológicos de su aplicación y se han publicado muy pocas investigaciones que muestren cuán bien se han transferido las habilidades a los estudiantes [15]. Estas herramientas tienen los siguientes problemas:

- Las pruebas estáticas y dinámicas, así como los mecanismos usados para evitar la modalidad prueba-y-error, son todavía insuficientes para generar un nivel de retroalimentación personalizada y oportuna que favorezca la asimilación de conceptos y habilidades de programación por parte del alumno.
- No se orientan en forma explícita a la gestión de secuencias de problemas con complejidad gradual, limitando el pleno desarrollo de las metodologías que las aplican.

En este artículo se presenta una plataforma de evaluación automática y una metodología que incluye criterios de diseño de secuencias de problemas para apoyar el proceso de enseñanza/aprendizaje en cursos introductorios de programación. Este trabajo incorpora mecanismos a la plataforma, que permiten obtener información en línea de los patrones de comportamiento de los alumnos, con el fin de evaluar y retroalimentar a los estudiantes en forma personalizada.

En las siguientes secciones se discuten los trabajos relacionados, se describe el ambiente de aprendizaje y la metodología, se presentan los resultados obtenidos y finalmente las conclusiones.

TRABAJOS RELACIONADOS

Aspectos cognitivos y educacionales

Entre los factores que pueden influir en el éxito de los novatos en programación están: experiencia previa en computación, comodidad mientras se trabaja, sensación de estar jugando, ***autoeficacia***, conocimientos previos en matemáticas y ciencias, estilo de aprendizaje, y el ***modelo mental*** que el estudiante tiene de la programación [21]. Bandura [5] define la *autoeficacia* como "el juicio que una persona realiza de su propia habilidad para alcanzar un cierto nivel de desempeño en un cierto dominio". Influye en la cantidad de esfuerzo empleado, en la persistencia en el caso de fallar y en el rendimiento logrado. Es afectada por el rendimiento en los logros personales, el observar que sus semejantes pueden hacerlo, la persuasión verbal y el estado psicológico desde el cual el alumno juzga sus capacidades.

Vennila [6] postula que la *autoeficacia* es afectada positivamente si el ambiente de aprendizaje permite la *medición de superación individual*, la solución de problemas en *complejidad incremental*, la búsqueda de *soluciones en equipo*, el desarrollo de *modelos mentales* adecuados y la asignación de *tareas frecuentes y cortas*. Basándose en la teoría de adquisición de habilidades, Sweller [19] recomienda que la complejidad de los ejercicios debe ser siempre gradual, especialmente en la primera etapa del aprendizaje, conocida como etapa interpretativa, que es en la que se le impone al estudiante una mayor carga cognitiva.

Los *modelos mentales* son las representaciones cognitivas del conocimiento y el *modelo conceptual* es el estímulo que el profesor usa para enseñar conceptos. Los estudiantes novatos poseen *modelos mentales* que son una variante de los modelos conceptuales y no es raro que algunos resulten *no-viables* Kahney [8]. Por ejemplo, el modelo denominado 'copias' es viable y el modelo 'paso' es no viable, en el contexto de recursividad.

Götschi [7] y Sanders [16] identificaron modelos mentales para recursividad empleando una metodología basada en la inspección visual de exámenes manuscritos. La metodología de *análisis de protocolos verbales* [9-10] (grabaciones de voz mientras se realizan programas) ha permitido observar que si la intención al analizar un programa fuente es *leer-para-recordar* (documentar un programa), el enfoque es el texto y su forma, en cambio, si la intención es *leer-para-hacer* (reutilizar un programa), el enfoque es la secuencia de pasos para lograr las metas del programa y el modelo mental usado es más profundo. Lo anterior tiene directa incidencia sobre el diseño de la estrategia de enseñanza y el diseño de tareas de programación.

En tal sentido, las plataformas de evaluación automática son valiosas en la medida que generan un ambiente propicio para desarrollar metodologías de aprendizaje activo que incorporen de manera efectiva las recomendaciones de las teorías de carga cognitiva y de aprendizaje.

Herramientas de evaluación automática

Las herramientas creadas para la evaluación automática usualmente adoptan un mecanismo de prueba dinámica que ejecuta los programas de los alumnos a través de un conjunto de datos de prueba [1]. Por ejemplo, en el sistema TRY (Reek [4]), que proporciona retroalimentación inmediata al estudiante, el programa es probado comparando símbolo por símbolo los resultados generados con los resultados esperados. TRY permite al estudiante realizar varios intentos de respuesta, pero sólo los restringe a un número máximo para forzarlo a pensar exhaustivamente en la operación de su programa antes de someter su respuesta a evaluación.

PSGE [3], derivado de TRY, es un sistema pensado para incluir la adecuación de los conceptos de *prueba de programas* representados por el acrónimo SPRAE (Specification, Premeditation, Repeatability, Accountability, Economy), para especificar un ciclo de vida de tareas de programación que son susceptibles de ser calificadas automáticamente. El ciclo obtenido incluye tres etapas: (1) especificación de tareas (qué probar, cómo probar, plan de calificación); (2) distribución de tareas (publicar en repositorio público, los requisitos de aprobación, definir los probadores, definir las entradas y capturar las salidas, calcular los puntajes); (3) probar los programas de acuerdo al plan establecido.

TRAKLA [12] es una herramienta de evaluación automática aplicada a la simulación gráfica de algoritmos. El estudiante simula un algoritmo dado como ejercicio haciendo *drag-and-drop* sobre una representación gráfica de estructuras de datos (ej.:

árboles). Sin embargo, comprender los conceptos y principios no garantiza la habilidad de escribir programas para computadores. TRAKLA limita sólo la retroalimentación cuando el número de intentos sobrepasa un cierto umbral, para evitar modalidad de *prueba-y-error* y hacer pensar la respuesta. Al no tener código fuente, no puede generar retroalimentación mediante prueba estática.

CourseMaker [13] fue desarrollado en Java, usando todas las herramientas de diseño orientado a objetos y patrones para obtener modularidad y flexibilidad. Realiza pruebas dinámicas de funcionalidad y de eficiencia y pruebas estáticas que incluyen análisis de estilo y detección de plagio. Permite que el número de intentos y el nivel de retroalimentación sean definidos por el autor del problema. Este sistema registra la estadística de errores de compilación por alumno en una base de datos MySQL. Este registro es usado posteriormente para determinar los contenidos del curso que deben ser reforzados.

AutoLEP (Wang [2]) extiende el uso del análisis estático a programas para los cuales las pruebas dinámicas son insuficientes (no se ejecutan o no cumplen con un estilo requerido), pero que tienen en el programa fuente aspectos que deben ser considerados para realizar las marcas de calificación. Aunque este análisis estático mejora la calificación, no entrega herramientas en línea que permitan la identificación de modelos mentales.

Rössling [20] identifica el registro de información de los estudiantes como una ventaja muy importante de los sistemas de gestión del aprendizaje (LMS), porque habilita a los profesores para obtener perfiles tanto de uso de los sistemas como de aprendizaje, realizar proyecciones del desempeño, etc. Pero el problema más relevante con estos registros es que generalmente son datos ilegibles, demasiado simples y no estandarizados. Los profesores deben realizar inferencias fuera de línea, por lo tanto las acciones remediales son postergadas.

Del análisis de los trabajos relacionados podemos resumir las siguientes observaciones:

- Las pruebas estáticas y dinámicas sólo obtienen información relativa a lo que ocurre con el problema que está siendo resuelto en ese momento, y las inferencias del efecto de secuencias didácticas sobre la formación de modelos mentales son aprovechadas fuera de línea, restándole oportunidad y personalización a la retroalimentación.
- Los mecanismos usados que evitan la modalidad de prueba-y-error no recogen información de las causas que lo producen, ni aportan a una retroalimentación orientadora.

Para mejorar la adecuación entre la plataforma y la metodología, superando las limitaciones observadas en los trabajos descritos, esta propuesta ha incorporado características que profundizan la aplicación de las recomendaciones derivadas de las teorías de la carga cognitiva y adquisición de habilidades. Estas características incluyen:

- Análisis estático orientado a monitorear modelos mentales y ayudar al estudiante a establecer relaciones en las secuencias de problemas que resuelve.
- Complementación de la retroalimentación con cuestionarios (minitest) que obligan al alumno a realizar análisis relacionados con "conceptos transversales" de programación.
- Obtención de información sobre las causas que llevan al alumno a la modalidad de prueba y error, para mejorar la identificación de modelos mentales no viables mejorando la retroalimentación.

AMBIENTE DE APRENDIZAJE

En la Facultad de Ingeniería de la Universidad de Concepción, los cursos introductorios de programación tienen tres horas de clases expositivas y tres horas de práctica en laboratorio, con una evaluación que incluye exámenes escritos, tareas fuera del aula, actividad de aprendizaje en laboratorio con evaluación automática sumativa, y un examen en computador evaluado automáticamente al finalizar el curso. La actividad de aprendizaje en laboratorio se realiza sobre una plataforma diseñada para operar en una sala, con los computadores de los alumnos comunicándose en red con el computador en el que corre el Centro Evaluador, como se muestra en la Figura 1. El avance es desplegado al curso mediante proyector multimedia.

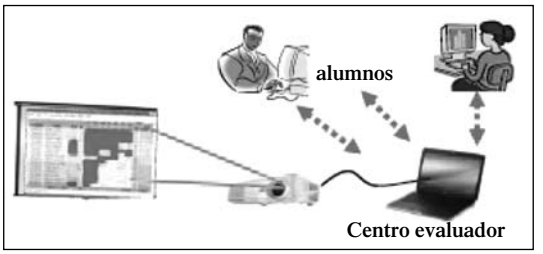


Figura 1. Sala típica, aislada de aprendizaje sincronizado.

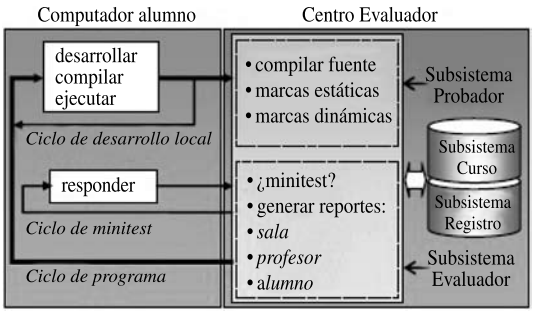


Figura 2. Ciclos de solución de un problema.

La plataforma se ha desarrollado y aplicado en forma conjunta con la metodología, a partir del año 2004, y está basada en la tecnología web, incluyendo PHP, lenguajes usados en los cursos (C, C++, MATLAB, Java), recursos Linux (filtros) y Mysql. Como se puede ver en la Figura 2, está compuesta por cinco subsistemas. El subsistema Curso maneja la información relativa a los alumnos, material didáctico, problemas, calificaciones, etc. El subsistema Registro almacena la información histórica de todos los eventos, marcas, patrones de error, intentos de respuestas, etc. El subsistema Probador ejecuta las pruebas dinámicas y estáticas y marca las metas que alcanza el programa del alumno. El subsistema Evaluador realiza las inferencias del nivel de aprendizaje de cada alumno y genera retroalimentaciones a partir de las marcas y registro histórico de comportamiento. El subsistema Acceso controla el acceso y la seguridad de las actividades que realizan los usuarios.

Para resolver un problema, el alumno interactúa desde su computador con el Centro Evaluador realizando *ciclos de programa*, que se repiten hasta que su programa respuesta cumple con las especificaciones del enunciado de un problema particular. Las etapas de un *ciclo de programa* incluyen: *ciclos de desarrollo local*, evaluación del programa respuesta, *ciclos de minitest* y retroalimentación al alumno. Cada ciclo de programa incluye un *intento* del alumno por lograr la aprobación de su respuesta al problema.

La incorporación de ciclos de minitest al ciclo de problema tiene por objetivo mejorar la obtención de información relativa al aprendizaje del alumno, para entregarle una retroalimentación más personalizada y evitar que realice muchos *intentos* pasando a la modalidad de *prueba-y-error*. Un minitest consiste en un pequeño cuestionario (verdadero/falso, selección de alternativas, ingreso de datos numéricos o de texto). Las preguntas son seleccionadas al azar desde un conjunto proporcionado por el profesor.

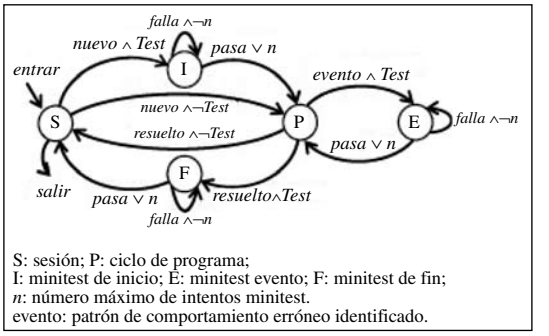


Figura 3. Autómata de transición entre ciclos.

Como se muestra en la Figura 3, el alumno entra a una sesión (S), selecciona un problema y pasa a realizar ciclos de programa (P), con la opción de que pase previamente a responder un minitest de inicio (I). Los ciclos de programa pueden ser configurados para ser interrumpidos pasando a responder un minitest de evento (E) al detectarse algún comportamiento particular (Ej.: muchos *intentos*). Al lograr la solución del problema puede pasar nuevamente por un minitest de fin (F) antes de continuar con el siguiente programa en S.

La inserción de cada tipo minitest es opcional y queda a criterio del profesor incluir el(los) que estime conveniente para cada problema. El minitest de inicio (I) tiene como objetivos orientar al alumno en la búsqueda de la solución, obtener un registro de su conocimiento de entrada y determinar si debe saltar a problemas de exigencia mayor. El minitest de evento (E) se realiza sólo cuando se detecta que el alumno exhibe algún patrón de error asociado (*modelo mental no viable)* o muchos *intentos*

(Douce [1]), y su propósito es forzarlo a pensar mejor su respuesta y registrar las posibles causas. El minitest de fin (F) tiene el objetivo de que el alumno reflexione sobre el modo en que logró su solución y la relacione con otros problemas.

La idea principal es sincronizar el uso de minitest con momentos significativos dentro del ciclo de programa para mejorar oportunamente los conceptos débiles. Conceptos tales como *rol de las variables, complejidad y eficiencia de algoritmos, alternativas de recorrido de las estructuras de datos, generalización, reutilización, identificación de invariantes, aplicación a problemas del mundo real, etc*., pueden ser explícitamente estimulados y monitoreados usando este mecanismo. La pregunta siguiente se usó en un minitest al finalizar un problema en el que los alumnos debían aprender a trabajar con listas encadenadas:

Si ejecuto p=p->prox->prox, entonces p apuntará: {a) al nodo siguiente, b) al subsiguiente, c) al mismo, d) al nodo anterior, e) a NULL}.

Figura 4. Reporte en sala al profesor.

Interfaces

El reporte al profesor incluye un indicador de estado de cada alumno que permite identificar alumnos que están con dificultades y su color es función del nivel de avance, del tiempo transcurrido, del número de intentos y otros parámetros (ver Figura 4). Entre los datos reportados se encuentran: marcas alcanzadas, número de intentos de respuesta, tiempos medios por meta, el puntaje de calificación, indicador de estado, alarmas de plagio, etc. El indicador de estado permite dar apoyo en sala a los alumnos que se encuentren en una situación más vulnerable.

La retroalimentación al alumno se muestra en el despliegue de resultados comparados, como lo muestra parcialmente la Figura 5. En este caso, la plataforma genera una representación visual de lo que hace una función construida por el alumno en contraste con lo que debería hacer. La retroalimentación incluye errores de compilación, posibles patrones de error típico, marcas de metas dinámicas, puntaje actual, sugerencias de material y/o preguntas de minitest que lo orientan a encontrar la solución.

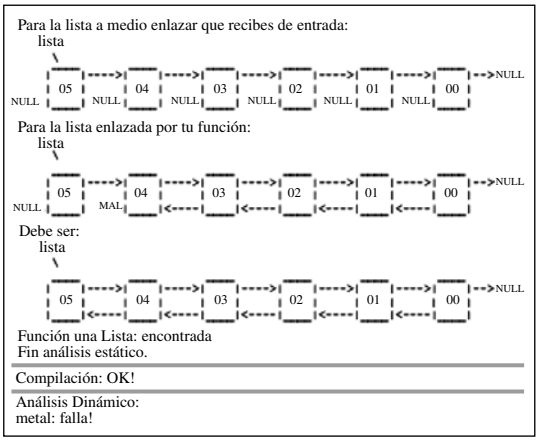


Figura 5. Retroalimentación al alumno.

Diseño de secuencias de problemas

Para seleccionar, diseñar y evaluar los problemas del aprendizaje sincronizado se usó un conjunto de criterios derivados de la teoría cognitiva y adquisición de habilidades y de la propia experiencia previa [17], que ha ido incorporando proposiciones como la de Scott [18], en la que se describen problemas de programación diseñados para alcanzar los objetivos de aprendizaje en cada nivel de la taxonomía revisada de Bloom.

El primer criterio consiste en que 'todos' los alumnos resuelvan 'todos' los problemas, para afectar positivamente su autoeficacia. Para diferenciar la calificación según el desempeño, ésta queda definida de la siguiente forma: $N = f(\boldsymbol{n, i, t, c})$, con ***n****:* número de marcas, ***i***: número de intentos, ***p***: tiempo empleado en resolverlos, ***t***: tiempo medio y ***c***: comportamiento. El comportamiento (***c***) incluye aspectos tales como la forma en que el alumno responde minitest: un tiempo demasiado breve y fallar en todos, puede significar que no quiere tomar tiempo para pensar. Que obtenga las marcas

dinámicas pero falle en cierto tipo de preguntas de minitest, puede ser síntoma de no haber obtenido el conocimiento al nivel esperado.

Un segundo criterio usado en el diseño de problemas consiste en lograr que los alumnos resuelvan muchos problemas relacionados con complejidad incremental, hasta lograr que su nivel de habilidad complete la *etapa interpretativa* [19]. Por ejemplo: imprimir {***1*** a ***n***, ***n*** a ***1***, ***–1*** a ***–m***, ***–m*** a ***-1***, ***n*** a ***1***, de ***–1*** a ***–m***, ***n*** *pares-impares*} da origen a varios problemas simples que refuerzan el control de iteraciones, los que pueden ser resueltos en una sola sesión de las primeras semanas del curso. El minitest es el mecanismo usado para que el alumno vaya descubriendo características y relaciones entre ellos.

Entre otros criterios usados están los siguientes: controlar el nivel de abstracción al que se forma el modelo mental, estimular la reutilización para comprender el diseño de algoritmos, inducir la vivencia del "descubrimiento" de algoritmos clásicos (búsqueda, ordenamiento, etc.), familiarizar con aspectos de complejidad y eficiencia, poner al estudiante frente a la tentación de resolver sólo el caso particular para inducirlo a mejorar su respuesta

	enunciado	**estimular mediante minitest**
I)	a) Intercambiar contenido de dos variables. b) Intercambiar condicionalmente contenido de dos variables (mayor a la derecha) .	número de variables auxiliares requeridas
II)	a) Intercambiar contenidos de variables x, y, z en los pasos: *x <-> y* *y <-> z* b) Intercambiar condicionalmente (menor a mayor) contenidos de variables x, y, z en los pasos: x <->> y y <->> z	posición del primero, después posición del mayor, después
III)	ídem II pero con cuatro variables	posición del mayor, después
IV)	a) Repetir los pasos III) con arreglo de 4 elementos pero con constantes en subíndice. *x[0] <->> x[1]* *x[1] <->> x[2]* *x[2] <->> x[3]* b) Ídem a) pero repitiendo dos veces el conjunto de pasos. c) Ídem a) pero repitiendo más veces sobre un arreglo de más elementos.	patrón a iterar efecto de traslación de intercambios efecto de repetir sobre arreglo y sub-arreglos número apropiado de subarreglos
V) – Debes escribir el código del método "ordena" de forma tal, que el contenido del arreglo "x", quede ordenado de menor a mayor: {1,2,3,4,5,6}. – Debes lograr las modificaciones al arreglo "x" haciendo que el método "ordena" llame repetidamente **dos** métodos que debes elegir desde la clase A, los cuales no debes modificar. – "Ordena" debe llamar los métodos elegidos repetidamente, de forma que los elementos mayores alcancen su posición definitiva primero. – Debes lograr un código que minimice el número de llamadas a los métodos, pero "no debes actuar directamente sobre el arreglo". Las marcas que puedes obtener son: **marca 1)**: {5,4,3,2,1,**6**} el número mayor debe quedar en la última posición del arreglo. **marca 2**): {4,3,2,1**,5,6**} el segundo número mayor debe quedar en la penúltima posición del arreglo. **marca 3**): {1,2,3,4,5,6} sin minimizar el número de llamadas a los métodos. **marca 4**): {1,2,3,4,5,6} minimizando el número de llamadas a los métodos.	public static void main(String[]args){ int x[]={6,5,4,3,2,1}; A a = new A(); a.ordena(x); } class A { public void ordena(int a[]){ **//tu código debe ir en este lugar y llamar dos** **//métodos más abajo que selecciones** } private boolean aaa(int a[], int i){ if (a[i]>a[i+1]) return true; else return false; } private void bbb(int a[], int i){ int r; r=a[i]; a[i]=a[i+1]; a[i+1]=r; } *//más métodos con código* *//que no sirve para ordenar* --------- }

Ejemplo 1. Secuencia didáctica para ordenamiento.

generalizándola, ponerlo frente a situaciones en las que tenga más posibilidades de cometer error, etc.

Ejemplo de aplicación

El Ejemplo 1 muestra una secuencia de problemas con complejidad incremental y los aspectos que se estimula a observar mediante un conjunto de preguntas de minitest. La secuencia tiene los siguientes objetivos didácticos:

- Forzar la intención del alumno: *leer-para-hacer* [9-10], para que identifique y reutilice algunos conceptos básicos previamente aprendidos, usando modelos mentales más profundos.
- Lograr que los alumnos "descubran" el algoritmo de ordenamiento de la Burbuja.
- Poner al estudiante frente a la posibilidad de resolver únicamente el caso particular de seis elementos, para inducirlo a generalizar su respuesta.
- Forzar el uso de funciones para encapsular código, facilitar la identificación de patrones invariantes y reforzar la forma de utilizarlos. En este caso, también facilita expresar lo que el alumno debe minimizar.
- Promover que el alumno desarrolle la perspectiva de observación del orden del algoritmo involucrado.

Un criterio cuya importancia se ha observado en la práctica es estimular que el alumno desarrolle el hábito de "verificar" que ha considerado todas las posibilidades y los efectos de visitar estructuras. Para el caso de arreglos, por ejemplo, se le somete a una mayor cantidad de ejercicios tendientes a visualizar un arreglo y visitar todos los posibles subarreglos mediante ciclos iterativos (ver Ejemplo 2). Este ejercicio se puede complementar con preguntas de minitest que pongan al alumno en una situación equivalente y/o lo interroguen acerca de haber considerado todos los posibles subarreglos. La reiteración de estas preguntas favorece el desarrollo del hábito.

```
Crear una función que sume elementos de un arreglo
unidimensional,  deja los resultados de la suma en
otro arreglo unidimensional, de la siguiente forma:

   si: int x[4]:{1,1,1,1} entonces el otro arreglo
   debe quedar: y[4]: {4,3,2,1}, donde:
                  y[0]=x[0]+x[1]+x[2]+x[3]
                  y[1]=x[1]+x[2]+x[3]
                  y[2]=x[2]+x[3]
                  y[3]=x[3]
```

Ejemplo 2. Visitas a arreglos y subarreglos.

RESULTADOS

Motivación y autoeficacia

La Tabla 1 muestra los promedios y desviación estándar de una encuesta aplicada a los estudiantes el año 2009. Las preguntas tienen el objetivo de conocer la percepción que los alumnos tienen de la metodología basada en evaluación automática. Los alumnos debieron elegir la opción que mejor representara su opinión. (Desde "completo desacuerdo" 1, hasta "completo acuerdo" 5). De los resultados de la encuesta se desprende que los alumnos desarrollaron una buena *autoeficacia* en programación, es decir, ellos creen que aprendieron (preguntas 10 a 19). Todos los alumnos consideran muy positivo que la plataforma les permita *obtener la respuesta de inmediato* (pregunta 6).

La motivación, medida por las preguntas 1, 2 y 4, muestra que los alumnos sienten que están jugando y les agrada aprender usando esta metodología. Les impulsa a trabajar el poder ver que sus compañeros avanzan y compararse con ellos. Independientemente de la causa que los impulsaba, el cien por ciento trabajó sin distraerse en otras actividades, lo cual no ocurre con cursos que no usan la plataforma, como se ha podido observar.

Desempeño de los alumnos

En las Figuras 6, 7 y 8 se muestran las diferencias en el desempeño entre dos grupos similares de alumnos, en la primera sesión de aprendizaje de un nuevo lenguaje (Java), después de haber aprobado un primer curso de programación (C). Ambos grupos pertenecen a la misma especialidad de ingeniería y cursan el mismo semestre de su carrera, por lo tanto las asignaturas paralelas son las mismas. El grupo A, con 66 alumnos, en el primer curso realizó las sesiones prácticas de aprendizaje en computador sin la plataforma, pero realizaron dos exámenes en ella. El grupo B, con 63 alumnos, en el primer curso realizó 12 sesiones de aprendizaje con apoyo de la plataforma en lenguaje C.

La sesión de comparación entre ambos grupos se realizó en la plataforma usando lenguaje Java. Ambos grupos debieron resolver, en dos horas, el

Tabla 1. Encuesta 2009 alumnos de programación.

	Promedio	Desviación estándar
01. Me agrada aprender con esta metodología basada en la plataforma.	4,8	0,44
02. Ver que mis compañeros avanzan me impulsa a trabajar.	4,1	1,24
03. Al terminar la sesión tengo la sensación de haber aprendido.	4,5	0,72
04. Con esta plataforma siento que estoy jugando y aprendiendo a la vez.	4,5	0,71
05. Esta forma de aprender me permite aprender más en menos tiempo.	4,4	0,79
06. La respuesta inmediata de la plataforma contribuye en mi aprendizaje.	5,0	0,0
07. El hecho de que la plataforma me calificara me produjo nerviosismo.	3,1	1,40
08. Al haber usado esta plataforma me siento más seguro de mis conocimientos.	4,4	0,74
09. Las sugerencias que entregaba la plataforma fueron acertadas.	4,0	0,88
10. Me siento capaz de hacer programas que lean desde teclado e impriman en pantalla.	4,8	0,56
11. Me siento capaz de hacer programas que operen sobre arreglos.	4,7	0,60
12. Me siento capaz de hacer funciones simples que usen parámetros de entrada por valor.	4,6	0,66
13. Me siento capaz de hacer funciones simples que usen parámetros de entrada por referencia.	4,5	0,71
14. Me siento capaz de elegir las estructuras de control de selección de acuerdo al enunciado.	4,5	0,71
15. Me considero capaz de trabajar con punteros.	4,4	0,83
16. Puedo crear programas y funciones que manejen listas encadenadas sencillas.	4,5	0,75
17. Me siento capaz de combinar ciclos iterativos con preguntas de control de selección.	4,5	0,87
18. Me siento capacitado para programar con archivos en forma básica.	4,2	1,12
19. Me siento capacitado para seguir mejorando por mí mismo y construir programas más complejos.	4,2	1,00

mismo conjunto de problemas, que variaban desde "hola mundo", pasando por arreglos, hasta listas encadenadas. Ellos debían aprender conceptos de orientación al objeto tales como clases, métodos constructores, herencia, etc. En el gráfico de la Figura 6 se muestra el número de problemas resueltos por cada grupo. Se puede observar que un 47% de los alumnos del grupo A resolvió cinco problemas, un 52% resolvió cuatro, con un promedio de 4,5 problemas en la sesión. Los alumnos del grupo B resolvieron 7,6 problemas en promedio y un 73% resolvió ocho problemas.

En las Figuras 7 y 8 se puede observar el desempeño de ambos grupos desde la perspectiva del tiempo. Se estimó que a un alumno debería tomarle unos 15 minutos resolver cada problema. La escala del eje horizontal representa el número de problemas que debería llevar resuelto en múltiplos de 15 minutos, por ejemplo: 3p/45m significa tres problemas en 45 minutos. Se puede observar que del grupo A, ningún alumno resolvió un primer problema en los primeros 15 minutos, recién a los 30 minutos un 3% logró tomar el ritmo con dos problemas, el máximo se produjo a los 45 minutos: un 33% con cuatro problemas. En el grupo B, un 75% de los alumnos resolvió al menos un problema en los primeros 15 minutos y un 43% estaba adelantado, tendencia que se mantuvo el resto de la sesión.

El hecho de que un alto porcentaje de alumnos del grupo B pudo resolver los problemas relacionados con listas encadenadas (problemas 6, 7 y 8), puede explicarse en parte a que en el primer curso contaban con una retroalimentación visual como la de la Figura 5, lo que les permitió familiarizarse mejor con este tipo de problemas.

En la Tabla 3 se muestra el efecto que ha tenido el curso de programación de la especialidad de informática, basado en la incorporación paulatina de la metodología que incluye evaluación automática, desde el año 2004 al 2009. El año 2004 se usó por primera vez la plataforma sólo para someter al curso a una prueba frente al computador. Esta prueba se ha usado cada año como una referencia

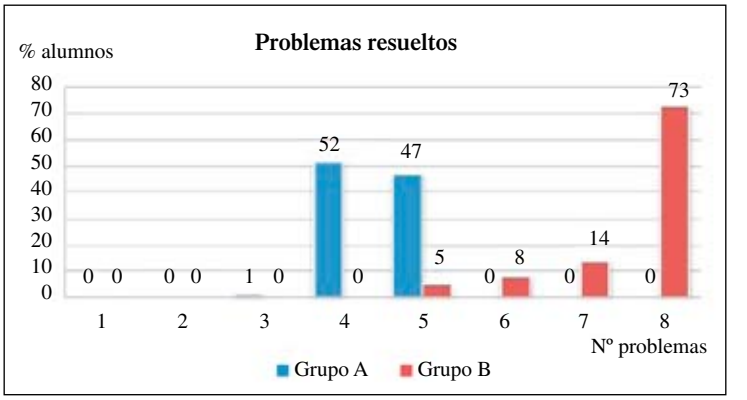


Figura 6. Problemas resueltos por cada grupo.

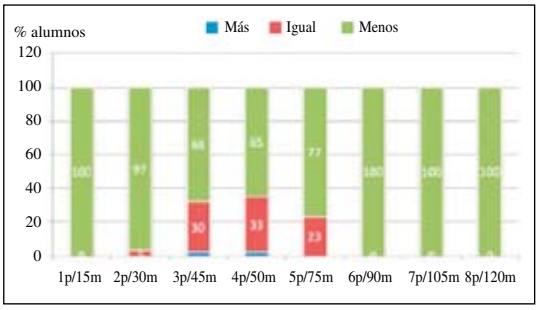


Figura 7. Rendimiento grupo A.

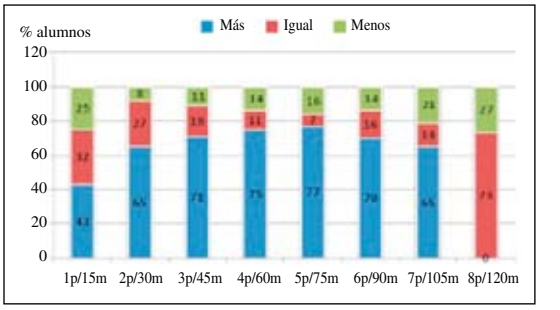


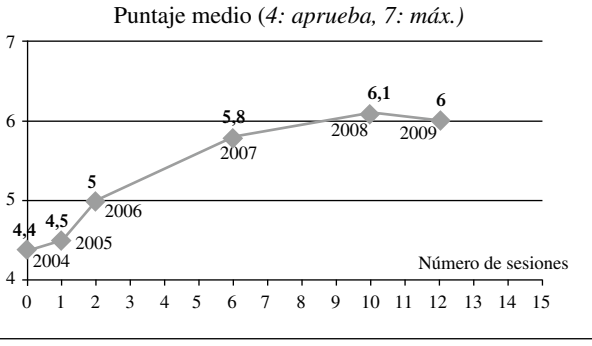
Figura 8. Rendimiento grupo B.

que permite observar el efecto de ir introduciendo en forma paulatina prácticas de aprendizaje con la plataforma. A partir del año 2006, la preparación previa a la prueba de referencia se va transformando en metodología de aprendizaje activo con evaluación automática sumativa, es decir, los problemas resueltos por el alumno son considerados en la calificación.

En el gráfico de la Tabla 2 se puede observar que el puntaje medio aumenta al aumentar el número de sesiones de aprendizaje, pero tiende a mantenerse en los tres últimos años. La duración de la prueba de referencia se fue reduciendo debido a que la mayoría de los alumnos terminaba en menos tiempo y se le aumentó el número de problemas.

Tabla 2. Prueba de referencia versus sesiones de aprendizaje con la plataforma.

año	Características	
	Problemas	Duración [hrs]
2004	**6**	**3, 5**
2005	**6**	**3**
2006	**6**	**3**
2007	**6**	**2,5**
2008	**7**	**2,5**
2009	**7**	**2**



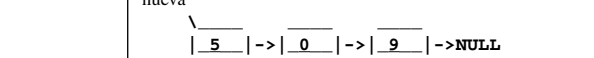
Construir una función que construya una lista nueva a partir de otra lista. La lista nueva debe copiar solo los nodos que tienen elementos mayores o iguales que cero. La lista original debe quedar intacta. Por ejemplo:

si la lista original es:

original

La lista resultante debe quedar:

nueva



una respuesta típica al problema:

```
struct nodo * cpNodos(struct nodo *original){
   struct nodo *aux, *nueva=NULL,*n;
   aux=original;
   while(aux!=NULL){
     if(aux->x>=0){
        n=(struct nodo *)malloc(sizeof(struct nodo));
        n->x=aux->x;
        n->p=nueva;
        nueva=n;
     }
     aux = aux->p;
   }
  return nueva;
}
```

Ejemplo 3. Copiar parcialmente una lista encadenada (C).

Tabla 3. Reconocimiento de patrones de error típicos.

Error identificado (modelo mental no viable)	% alumnos
"pedir memoria una sola vez al principio sólo para el puntero"	37
"la petición de memoria para nodo fuera del ciclo"	21
"falta return en la función para el puntero a la nueva lista"	13
"return dentro del ciclo, por cada nodo creado"	21
alguno de los casos anteriores	79
resolvieron el problema	84

Patrones de error (modelos mentales no viables) El propósito de la plataforma es facilitar el reconocimiento de modelos mentales erróneos, entregando a los docentes una herramienta que les permita configurar los mensajes y la extracción de muestras para identificar los patrones que se van descubriendo. Una vez identificado un patrón de error, se creaba el reconocedor correspondiente y era aplicado en las sesiones siguientes.

En las Tablas 3 y 4 se muestran diferencias encontradas después de que dos grupos de alumnos resolvieran el problema del Ejemplo 3. Los patrones de error fueron obtenidos en sesiones previas, a partir de problemas equivalentes. En la Tabla 3 se muestran algunos errores típicos y el porcentaje de alumnos que los manifestaron. Se puede observar que el 79% de los alumnos partió con alguno de estos errores que la plataforma pudo atender.

En la Tabla 4 se puede observar que en el caso con apoyo de reconocedor hay una disminución en el número de intentos y del tiempo medio para resolver. En algunas sesiones el número de intentos totales ha llegado a 1.000 en cursos de unos 60 alumnos.

Tabla 4. Efecto del reconocedor en el rendimiento.

	Sin reconocedor	*Con reconocedor*
Núm. intentos máx.	21	11
Núm. intentos medio	7,2	5,6
Tiempo medio solución	44 minutos	35 minutos

CONCLUSIONES Y TRABAJO FUTURO

En este artículo hemos presentado una plataforma de evaluación automática con mecanismos de reconocimiento de patrones de comportamiento y una metodología que incluye un conjunto de criterios de diseño de problemas, destinados a mejorar el proceso de enseñanza/aprendizaje en programación. Los resultados obtenidos muestran que los mecanismos incorporados a la plataforma y su modo de aplicación pueden mejorar la calidad del proceso de enseñanza/aprendizaje. La modalidad de uso utilizada permite garantizar que todos los

alumnos tienen un entrenamiento similar que nivela las habilidades para programar, aumenta el interés por la programación y mejora su autoeficacia. Por otra parte, la flexibilidad de la plataforma hace posible que las metodologías de apoyo al aprendizaje incorporen las recomendaciones derivadas de las teorías de las ciencias cognitivas y adquisición de habilidades, mejorando el desempeño de los alumnos en programación.

Como trabajo futuro se abordará la evaluación automática de programas con interfaz gráfica y la capacidad de evaluar automáticamente el aprendizaje de programación solucionando problemas en equipo.

AGRADECIMIENTOS

Este trabajo ha sido financiado por el proyecto DIUC 207.093.012.-1.0 de la Universidad de Concepción.

REFERENCIAS

[1] C. Douce, D. Livingstone and J. Orwell. "Automatic test-based assessment of programming: a review". Journal on Educational Resources in Computing (JERIC). Vol. 5, Issue 3. September, 2005.

[2] T. Wang, X. Su, P. Ma, Y. Wang and K. Wang. "Ability-training-oriented automated assessment in introductory programming course". Journal Computer & Education. Vol. 56, Issue 1. January, 2011.

[3] E.L. Jones. "Grading student programs-a software testing approach". Journal of Computing Science in Colleges. Vol. 16, Issue 2. January, 2001.

[4] K.A. Reek. "The TRY System or How to Avoid Testing Students Programs". Proceedings of SIGCSE, pp. 112-116. 1989.

[5] A. Bandura. "Social foundation of thought and action". Prentice Hall. Englewood Cliffs, NJ, USA. 1986.

[6] R. Vennila, D. Labelle and S. Wiendenbeck. "Self-efficacy and mental models in learning to program". ACM SGCSE Bulletin. Vol. 36, Issue 3. September, 2004.

[7] T. Götschi, I. Sanders and V. Galpin. "Mental Models of Recursion". ACM SIGCSE Bulletin. Vol. 35, Issue 1. January, 2003.

[8] H. Kahney. "What do a novice programmers know about recursion?". CHI'83 Proceedings of the SIGCHI conference on Human Factors in Computing Systems. December, 1983.

[9] A.J. Ko and B. Uttl. "Individual differences in program comprehension strategies in unfamiliar programming systems". Proceedings of the 11th IEEE International Workshop on Program Comprehension. IEEE Computer Society, Washington DC., EE.UU. 2003.

[10] S. Letowsky. "Cognitive process in program comprehension". In Empirical Studies of Programmers, pp. 58-79. IEEE Computer Society Press. 1986.

[11] R. Bednarik and M. Tukianinen. "An eye-tracking methodology for characterizing program comprehension processes". Proceedings of the 2006 symposium on Eye tracking research & applications (ETRA). New York, NY, USA. 2006.

[12] L. Malmi, V. Karavirta, A. Korhonen and J. Nikander. "Experiences on automatically assessed algorithm simulation exercises with different resubmission policies". Journal on Educational Resources in Computing (JERIC). Vol. 5, Issue 3. New York, NY, USA. September, 2005.

[13] C. Higgins, G. Gray, P. Symeonidis and A. Tsintfsias. "Automated assessment and experiences of teaching programming". ACM Journal on Educational Resources in Computing. Vol. 5, Issue 5, pp. 1-21. September, 2005.

[14] K.M. Ala-Mutka. "A Survey of automatic assessment approaches for programming assignments". Computer Science Education. Vol. 15, Issue 2, pp. 83-102. 2005.

[15] A. Pears, S. Seideman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin and J. Parteson. "A survey of literature on the teaching of introductory programming". ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education, pp. 204-222. December, 2007.

[16] I. Sanders, V. Galpin and T. Götschi. "Mental models of recursion revisited". ACM SIGCSE Bulletin. Vol. 38, Issue 3. September, 2006.

[17] J. López, C. Hernández y Y. Farran. "Plataforma de autoaprendizaje y evaluación automática".

Revista Electrónica de la Sociedad Chilena de Ciencia de la Computación ISSN: 0717-4276. Fecha de consulta: Diciembre 2010. http://www.sccc.cl

[18] T. Scott. "Bloom's taxonomy applied to testing computer science classes". Journal of Computer Sciences in Colleges. Vol. 19, Issue 1, pp. 267-274. October, 2003.

[19] J. Sweller. "Cognitive load during problem solving: effects on learning". Cognitive Science. Vol. 12, pp. 257-285. 1988.

[20] G. Rössling, M. Joy, A. Moreno, A. Radenski, L. Malmi, A. Kerren, T. Naps, R.J. Ross, M. Clancy, A. Korhonen, R. Oechsle and J.A. Velázquez. "Enhancing learning management systems to better support computer science education". ACM SIGCSE Bulletin. Vol. 40, Issue 4, pp. 142-166. December, 2008.

[21] P. Byrne and G. Lyons. "The effect of student attributes on success in programming". Proceedings of ITiCSE 2001, pp. 49-54. ACM Press. NY, NY, USA. 2001.