Um Mecanismo para Correção Automática de Exercícios Práticos de Programação Introdutória

Fillipi D. Pelz, Elieser A. de Jesus, André L. A. Raabe

Universidade do Vale do Itajaí - UNIVALI
Itajaí – Santa Catarina – Brasil
{fillipi,elieser,raabe}@univali.br

Abstract. *In this paper we propose a mechanism for automatic assessment of small programming exercises by making the following checks: 1) syntactic verification, 2) verifying the presence of mandatory commands, 3) verification of the adequacy of program structure, and 4) program execution to test its outputs. The mechanism of automatic assessment was used in two distinct research and presented good results in the generation of feedback to programming learners. Some limitations and implications of the correction mechanism in student learning are discussed throughout the article.*

Resumo. *Neste artigo propõe-se um mecanismo para a correção automática de pequenos exercícios práticos de programação onde se faz: 1) a verificação sintática; 2) a verificação da presença de comandos obrigatórios; 3) a verificação da adequação da estrutura do programa; e 4) a execução do programa para testar suas saídas. Este mecanismo de correção automática foi utilizado em duas pesquisas distintas e apresentou bons resultados na geração de feedback para os aprendizes de programação. Algumas limitações e implicações do mecanismo de correção na aprendizagem dos alunos são discutidas ao longo do artigo.*

1. Introdução

A aprendizagem de programação introdutória é entendida como algo fundamental na formação em computação e áreas afins (CRISTÓVÃO, 2008). Entretanto, pesquisadores como Beauboef e Mason (2005) dizem que muitas universidades estimam taxas de reprovação e evasão entre 30 e 40% para as disciplinas de programação introdutória.

Sabe-se que os índices de reprovação e evasão em disciplinas de programação introdutória são influenciados por diversos fatores. Alguns trabalhos vêm procurando delinear a origem dos problemas na aprendizagem de programação e fornecer pistas para as possíveis soluções, como por exemplo, os trabalho de Castro *et al* (2002) e de Raabe e Silva (2005).

Neste artigo, parte-se do pressuposto que a falta de *feedback* sobre o resultado dos exercícios práticos de programação é um dos fatores que podem contribuir para os índices mencionados anteriormente, uma vez que sem *feedback* adequado o aluno pode acabar desistindo da disciplina ou até mesmo de todo o curso por conta de uma série de experiências frustrantes na tentativa de aprender os princípios da programação de computadores.

Uma forma de fornecer *feedback* para os alunos durante os exercícios práticos de programação é possibilitar a correção automatizada, permitindo que os estudantes rapidamente saibam se atingiram ou não o resultado esperado em cada atividade. Sem ferramentas automatizadas de correção torna-se muito difícil para um professor fornecer *feedback* adequado para todos os seus alunos, sendo que esta dificuldade pode ampliar-se ainda mais no caso da Educação à Distância (EAD) em função da grande quantidade de alunos matriculados.

Neste artigo propõe-se um mecanismo de correção automática de exercícios práticos de programação introdutória composto de quatro etapas sequenciais, sendo elas: 1) verificação sintática do programa construído pelo aluno; 2) verificação da presença de comandos obrigatórios, tais como desvios condicionais ou laços de repetição; 3) verificação da similaridade entre a estrutura do programa do aluno e as estruturas consideradas pelo professor como soluções gabarito; e 4) execução do programa para verificar se as saídas apresentadas correspondem às saídas esperadas. A estrutura do mecanismo de correção é mostrada em pseudocódigo no quadro abaixo:

```
Se não existem erros sintáticos então {
   Se existem comandos obrigatórios e eles foram utilizados então {
      Se programa têm similaridade suficiente com os gabaritos então {
         Se saídas do programa são iguais as saídas esperadas então {
            Retorna 'programa correto'
         }
      }
   }
}
Retorna programa incorreto
Retorna erros encontrados
```

Quadro 1. Algoritmo em pseudocódigo para a correção automática dos programas.

O mecanismo de correção foi utilizado em duas pesquisas distintas, sendo que uma delas foi publicada anteriormente em Jesus e Raabe (2010). Cabe ressaltar que mesmo referenciando material publicado anteriormente este artigo explora resultados inéditos e complementares aos já publicados. Primeiramente utilizou-se este mecanismo de correção em um experimento envolvendo três turmas de alunos ingressantes em um curso de Ciência da Computação. Em um segundo momento, uma variação do mecanismo de correção foi utilizada para melhorar o *feedback* fornecido pelo ambiente de construção de programas denominado PortugolStudio. Os resultados da utilização do mecanismo de correção nos dois casos mencionados anteriormente são descritos mais detalhadamente ao longo deste artigo.

A seguir apresenta-se o embasamento teórico necessário para a compreensão do mecanismo de correção automática e os detalhes sobre o seu funcionamento. Na sequencia são discutidos os experimentos realizados com um total de 93 alunos, as implicações do uso do mecanismo de correção na aprendizagem, as suas limitações e por fim as conclusões e recomendações para futuros utilizadores.

2. Correção Automática de Programas

Segundo Rahman e Nordin (2007) existem duas abordagens mais comuns na correção automática de exercícios práticos de programação: a dinâmica e a estática. Na

abordagem dinâmica a correção é realizada através da execução do código fonte. Neste caso, o programa é considerado correto se para uma entrada específica a saída é igual a um determinado valor esperado. Já na abordagem estática a estrutura do código é analisada, sem a necessidade da execução do mesmo. Ambas as abordagens mencionadas anteriormente podem ser observadas, por exemplo, na arquitetura proposta por Castro *et al* (2002). A seguir discute-se mais detalhadamente cada uma destas principais abordagens.

2.1 Técnicas Dinâmicas de Avaliação de Programas

As técnicas dinâmicas caracterizam-se pela comparação das saídas do programa com um conjunto de valores considerados corretos. Douce (2005) chama a atenção para o fato de nesta abordagem é possível que mesmo um programa mal construído produza os resultados corretos, o que diminui a qualidade da correção automatizada.

Daly e Horgan (2004) utilizam os testes dinâmicos ao comparar as saídas do programa do aluno com as saídas esperadas usando um conjunto de dados de teste pré-definido para cada exercício ou um conjunto de teste aleatório. Conforme ressalta Douce (2005), o problema mais significativo deste tipo de abordagem é o fato de que as entradas e saídas para todos os exercícios devem ser cuidadosamente especificadas, sob o risco de comprometer a qualidade do mecanismo de correção automática.

No trabalho de Motta *et al* (2009) os autores também utilizam os testes dinâmicos como parte de um mecanismo de avaliação automática de programas. Segundo os autores os resultados obtidos com os testes dinâmicos foram insuficientes para realizar inferências sobre o aprendizado de habilidades avançadas de programação.

O mecanismo de correção proposto neste artigo faz uso de testes dinâmicos em uma de suas etapas de correção. Porém, as limitações deste tipo de abordagem, mencionadas por Douce (2005), são superadas através de etapas adicionais de avaliação dos programas.

2.2 Técnicas Estáticas de Avaliação de Programas

Truong, Roe, e Bancroft (2004) mencionam que as técnicas para a análise estática de programas variam desde comparação de *strings* que representam o código fonte (a abordagem mais simples) até a comparação de grafos que representam programas (a abordagem mais complexa).

Rahman e Nordin (2007) citam algumas análises estáticas que podem ser feitas em exercícios práticos de programação: análise do estilo de programação, análise dos erros sintáticos ou semânticos, avaliação de métricas de software, análise de similaridade estrutural e não-estrutural, análise de palavras chave, detecção de plágio, entre outras. No trabalho de Saikkonen, Malmi, e Korhonen (2001) também são utilizadas técnicas de análise estática, tais como a verificação da estrutura e a detecção de plágio. A seguir discute-se mais detalhadamente a análise da estrutura de programas, uma vez que esta técnica é parte central no mecanismo de correção automática proposto neste artigo.

2.2.1 Análise da Estrutura de Programas

Rahman e Nordin (2007) descrevem a análise estrutural, utilizada para determinar a similaridade entre a estrutura do programa do aluno e a solução gabarito, sendo esta última criada pelo professor. Esquemas da estrutura tanto do programa do professor quanto do aluno são gerados e então comparados. Segundo os autores, dependendo da complexidade do exercício vários esquemas alternativos deverão ser criados pelo professor, de maneira que um conjunto razoável de soluções seja contemplado pelo mecanismo de correção automática.

No trabalho de Sager *et al* (2006) a comparação da estrutura do programa do aluno com a estrutura pré-definida pelo professor é feita por algoritmos de comparação de árvores sintáticas abstratas (ASAs). Uma das técnicas mencionadas é a *Tree Edit Distance*, utilizada para determinar quantos passos são necessários para transformar uma árvore em outra aplicando um conjunto de operações de edição, tais como inserção, substituição e remoção de nós.

Uma alternativa para a técnica mencionada anteriormente é transformar as ASAs que representam os programas em *strings*, ou seja, serializá-las, e então compará-las utilizando técnicas de *String Edit Distance*. Segundo Ristad e Yianilos (1998), as técnicas de *String Edit Distance* permitem que se calcule a similaridade entre duas *strings* por meio do número mínimo de inserções, deleções e substituições de caracteres necessárias para transformar uma *string* em outra. Entre as técnicas para comparação de *strings* Navarro (2001) destaca a *Levenshtein distance* ou *edit distance*.

O método de correção automática proposto neste artigo faz uso da versão serializada da árvore sintática abstrata dos programas e a compara com soluções gabarito através do cálculo da distância de Levenshtein. Os detalhes das etapas do mecanismo de correção são apresentados a seguir.

3. Detalhes do Método de Correção Automática

O mecanismo de correção automática proposto neste artigo é composto de quatro etapas, a saber: 1) a verificação sintática; 2) a verificação da presensa de comandos de programação considerados como obrigatórios; 3) a verificação da similaridade estrutural entre o programa do aluno e um conjunto de soluções gabarito; e 4) a comparação entre as saídas apresentadas pelo programa do aluno e as saídas definidas como corretas pelo professor.

A verificação sintática do programa consiste em avaliar se é possível executar o programa sem erros, ou seja, se cada um dos elementos do código (*loops*, desvios condicionais, etc.) pode ser executado corretamente. Se nenhum problema é encontrado na verificação sintática do programa então as etapas seguintes da correção são realizadas. A verificação sintática não será discutida em maiores detalhes neste artigo por ser um assunto amplamente coberto pela literatura sobre compiladores e interpretadores. Sendo assim, a seguir são apresentados os detalhes das três últimas etapas do mecanismo de correção automática proposto neste artigo. A ordem de apresentação das etapas será um pouco diferente da ordem em que elas foram mencionadas anteriormente apenas para facilitar a compreensão das mesmas.

3.1 Verificação da Similaridade Estrutural

Nesta etapa do mecanismo de correção é necessário determinar a similaridade entre a estrutura do programa do aluno e as estruturas consideradas como solução para o problema, sendo que estas últimas devem ser previamente definidas pelo professor.

Para que a similaridade entre as estruturas seja determinada propõe-se que seja feita a serialização da estrutura do programa do aluno, normalmente representada em uma árvore sintática abstrata (ASA). A serialização neste caso consiste em obter uma *string* que representa de alguma maneira a estrutura do programa. Assim, o processo de comparação das estruturas dos programas pode ser resumido a uma comparação de *strings*, e quanto mais similares as *strings* tanto mais similares serão as estruturas dos programas que elas representam. A seguir discute-se mais detalhadamente o procedimento de serialização proposto.

3.1.1 Detalhes Sobre o Processo de Serialização das Estruturas dos Programas

O padrão que se utilizou para serializar as ASAs, ou seja, transformá-las em *strings*, foi baseado no trabalho de Truong, Roe, e Bancroft (2004), onde os autores utilizam as ASAs "normalizadas". Esta normalização se dá pela utilização de nós genéricos que representam classes de comandos de programação. Por exemplo, todos os laços de repetição são representados pelo mesmo tipo de nó, ao invés de um tipo específico para cada variedade de laço. Dessa forma, evita-se a necessidade de uma solução gabarito diferente para cada tipo de laço de repetição disponível em uma linguagem de programação.

Já a ordem em que os nós da ASA são concatenados no processo de serialização é baseada no trabalho de Baxter *el al* (1998), onde os nós das sub árvores são visitados na ordem: raiz, e filhos da esquerda para direita. Inicialmente pensou-se em utilizar palavras para representar cada nó da ASA no momento da serialização, tais como: SE, SE_SENAO, ENQUANTO, etc. Entretanto, ao comparar as *strings* isto faria com que o algoritmo de comparação encontrasse similaridade estrutural onde na verdade haveria apenas caracteres coincidentes nos nomes utilizados para serializar os nós. Sendo assim, propõe-se que cada tipo de nó da ASA seja serializado como um número (decimal ou hexadecimal). Por exemplo, todos os *loops* podem ser serializados como o número 1, todas as declarações de variáveis como o número 2, e assim por diante. Ao final deste processo de serialização a estrutura de um programa estará resumida em uma sequência de números que lhe representarão.

3.1.2 Determinação da Similaridade entre as Estruturas dos Programas

Para a obtenção do grau de similaridade entre as *strings* que representam as estruturas dos programas propõe-se a utilização da distância de Levenshtein com uma adaptação para determinar um *coeficiente de similaridade*. Este coeficiente pode assumir valores entre 0 e 1, representando respectivamente os extremos *nenhuma similaridade* e *similaridade total* entre as *strings*. Tal coeficiente pode ser calculado a partir da Equação (1).

$$Similaridade = 1 - \frac{\mathrm{DL}}{Max(|\mathrm{EPA}|, |\mathrm{ES}|)} \qquad \text{Equação 1}$$

Na equação 1 *DL* representa o valor calculado com a distância de Levenshtein, *EPA* representa a quantidade de caracteres na *Estrutura do Programa do Aluno*, e *ES* representa a quantidade de caracteres na *Estrutura da Solução* (o gabarito).

Propõe-se que para cada problema submetido ao mecanismo de correção automática existam várias soluções gabarito e que a estrutura do programa do aluno seja comparada com todas estas soluções. Para cada uma destas comparações será obtido um coeficiente de similaridade diferente. Então, pode-se utilizar o maior coeficiente de similaridade obtido e compará-lo com um limiar de similaridade mínima. Nos experimentos realizados com alunos utilizou-se o limiar de 0.75 com bons resultados.

3.2 Verificação da Presença de Comandos Obrigatórios

Nesta etapa do mecanismo de correção a estrutura do programa do aluno é analisada em busca de comandos considerados como obrigatórios pelo professor. Recomenda-se que a verificação da presença de comandos obrigatórios seja feita após a etapa de serialização da ASA, descrita anteriormente. Neste caso, se um exercício deve ser obrigatoriamente resolvido com desvios condicionais, e o código dos comandos deste tipo é o número 3, então, descobrir se existe um desvio condicional na estrutura do programa do aluno equivale a descobrir se o número 3 está presente na *string* que representa a estrutura serializada do programa.

3.3 Verificação das Saídas

No mecanismo de correção automática proposto neste artigo, uma vez que a compatibilidade estrutural mínima tenha sido observada no programa do aluno executa-se a última etapa da correção: a verificação das saídas do programa quando executado sobre determinadas condições de entrada.

Neste caso, são colocados valores previamente definidos pelo professor nas variáveis do programa, o código do aluno é executado com estas entradas de teste e a saída é comparada com valores pré-determinados pelo professor. Se nenhuma discrepância for encontrada entre as saídas apresentadas pelo programa do aluno e as saídas esperadas então o programa do aluno é considerado correto, pois nesse ponto do processo de correção todas as verificações anteriores já teriam sido realizadas.

4. Possíveis Implicações da Correção Automática na Aprendizagem

Em um primeiro momento o mecanismo de correção automática proposto neste artigo foi utilizado por três turmas de alunos ingressantes em Ciência da Computação, totalizando 88 alunos. Nesta pesquisa, publicada em Jesus e Raabe (2010), os alunos utilizaram três softwares diferentes onde resolveram um mesmo conjunto de exercícios práticos de programação. Nos dois primeiros softwares a cada desafio completado o aluno tinha acesso ao próximo, até que todos os desafios fossem completados. Um desafio só era considerado completado quando o aluno executava o programa e nenhum erro era encontrado pelo mecanismo de correção automática. Já no terceiro software, um jogo, os alunos puderam resolver os desafios na ordem em que desejaram e puderam visualizar o seu desempenho e o dos colegas através de um *ranque* ordenado pela quantidade de desafios resolvidos corretamente, o que adicionou competição ao jogo.

Em um segundo momento o mecanismo de correção apresentado neste artigo foi adaptado e reutilizado em um ambiente chamado PortugolStudio, construído para auxiliar alunos iniciantes em programação. O mecanismo de correção foi utilizado para melhorar o *feedback* fornecido aos alunos sobre os resultados de exercícios pré-determinados. Neste estudo foram realizados testes com cinco alunos de programação introdutória.

Em ambos os experimentos onde o mecanismo de correção automática foi utilizado os alunos podiam a qualquer momento executar o mecanismo de correção e verificar se seus programas estavam ou não corretos. Além disso, em uma das pesquisas o mecanismo de correção foi implementado de maneira que os alunos recebiam *feedback* sobre o grau de similaridade entre a estrutura do seu programa e as soluções gabarito, e também recebiam *feedback* sobre as estruturas faltantes no programa.

Nos experimentos percebeu-se alguns casos onde alunos construíam seus programas em um processo de tentativa e erro, usando o *feedback* do corretor automático para orientar-lhes sobre o que estava faltando na estrutura do programa. Ou seja, nestes casos os alunos deixaram de pensar na solução do problema e passaram a fazer apenas o que era necessário para atender as expectativas mínimas do mecanismo de correção automática.

Durante a realização dos experimentos com os alunos os testes dinâmicos (execução do programa) apresentaram benefícios que não haviam sido imaginados até então. Em algumas situações os alunos chamaram os professores alegando que seus programas estavam corretos e que o mecanismo de correção apontava erro em um determinado teste. Em cada exercício os testes dinâmicos foram pensados para testar inclusive as situações limite, como por exemplo: o que acontece quando todas as variáveis assumem valores muito pequenos? E valores muito grandes? O que se verificou é que alguns alunos construíam um programa que resolvia o caso médio ou o melhor caso do problema apresentado, mas não tratava o pior caso. Uma vez que os alunos foram alertados sobre a necessidade de refinarem suas soluções para que cobrissem todos os casos possíveis os professores não foram mais chamados para resolver tais situações. Neste momento, observou-se que a utilização dos testes dinâmicos promoveu a reflexão sobre como refinar os programas construídos, objetivo que nem sempre é atingido sem mecanismos automáticos de correção por conta da impossibilidade do professor testar todas as situações possíveis com cada um dos alunos.

5. Nível de Robustez e Limitações do Mecanismo de Correção Proposto

De maneira geral, nos experimentos realizados com alunos o mecanismo de correção automática mostrou bons resultados. O mecanismo consegue evitar muitas das soluções "ilegais", aquelas soluções que são apenas truques utilizados para atender as expectativas do mecanismo de correção. Sobre isto, cabe ressaltar que nem sempre a solução "ilegal" deve ser entendida como uma solução que prejudica a aprendizagem, pois o aluno que constrói uma destas "soluções ilegais" certamente precisou adquirir boas *doses* de conhecimento em programação introdutória para construí-la.

Uma das formas mais diretas de se burlar um mecanismo de correção automática seria atribuir diretamente nas variáveis os valores que o mecanismo espera como saída.

O mecanismo de correção evita este tipo de "solução ilegal" por meio da verificação da estrutura do programa e da verificação da presença de comandos obrigatórios. O aluno ainda poderia inserir comandos inúteis em seu programa para tentar "enganar" a etapa de verificação estrutural. Neste caso, alguns testes dinâmicos provavelmente falhariam, já que os valores atribuídos para as variáveis dificilmente atenderiam a todos os testes.

Quanto às limitações do mecanismo de correção, talvez a principal delas seja o fato de que a aplicabilidade do mecanismo limita-se a problemas onde tanto o número de variáveis quanto o tamanho das estruturas das soluções sejam pequenos. Em problemas maiores o professor teria que pensar em um número muito grande de variáveis no momento de especificar os testes dinâmicos, e a especificação das estruturas das várias soluções possíveis seria uma tarefa bastante árdua, possivelmente tomando mais tempo do professor do que uma correção manual dos exercícios.

Por outro lado, uma das limitações acima (a necessidade da especificação das variáveis em testes dinâmicos) pode ser parcialmente contornada sorteando-se valores para as variáveis ao invés de especificá-los manualmente. Uma consequência desta abordagem é que com valores sorteados não se pode garantir que tais valores testarão adequadamente as situações limite de um dado problema (ver discussão na seção anterior).

Apesar de estar limitado a pequenos problemas, cabe ressaltar que o mecanismo de correção automática está sendo proposto para exercícios práticos de programação introdutória, mais especificamente para aqueles exercícios trabalhados nas primeiras semanas de aula, onde tradicionalmente os problemas são suficientemente pequenos para enquadrarem-se nas restrições mencionadas anteriormente.

5. Conclusões

O mecanismo de correção automática de programas apresentado neste artigo foi concebido como um processo de quatro etapas, sendo elas a verificação: 1) da sintaxe; 2) da presença de comandos obrigatórios; 3) da adequação da estrutura do programa do aluno; e 4) dos valores de saída do programa diante de um conjunto de testes.

Em um primeiro momento o mecanismo de correção foi utilizado em experimentos com três turmas de alunos ingressantes em Ciência da Computação. Em um segundo momento uma versão adaptada do mecanismo de correção foi utilizada por cinco alunos em outra pesquisa. Nesta segunda pesquisa validou-se a integração do mecanismo de correção com a IDE de programação denominada PortugolStudio, onde o mecanismo foi utilizado para prover melhorias na apresentação de *feedback*. No total, 93 alunos utilizaram o mecanismo de correção automática descrito neste artigo.

Durante os experimentos realizados com os alunos foi possível observar que o *feedback* oferecido pelo mecanismo de correção automática tem um grande potencial pedagógico. Destaca-se aqui o fato de que os testes dinâmicos podem ser utilizados pelo professor para verificar as situações limite dos programas dos alunos, e com isto promover a reflexão sobre como os programas podem ser refinados.

Cabe ressaltar que o primeiro experimento realizado com alunos seria inviável sem a presença do mecanismo de correção, pois não seria possível para o professor corrigir todos os exercícios de todos os alunos (3 turmas com 30 alunos em média) na

medida em que estes terminavam suas atividades. Com a utilização do mecanismo de correção automática os alunos puderam avançar na realização dos exercícios práticos, sendo que a cada novo exercício o aluno tinha a certeza de que o exercício anterior estava correto. Ressalta-se também que neste experimento com os alunos uma das três turmas utilizou um jogo onde era necessário resolver exercícios práticos de programação para progredir. Também neste caso, a utilização do jogo só foi possível graças a presença do mecanismo de correção automática, e o fato dos alunos conseguirem chegar até as últimas fases do jogo (um total de 9 fases) mostra que o mecanismo cumpriu com o seu papel.

A partir das observações feitas durante os experimentos com os alunos, sugere-se que o *feedback* sobre a falta de comandos obrigatórios na estrutura do programa seja utilizado com cuidado em implementações futuras do mecanismo de correção. Quando este tipo de *feedback* é apresentado ao aluno sem nenhuma precaução é possível que este descubra as estruturas da solução do problema simplesmente invocando o mecanismo de correção, antes mesmo de começar a refletir sobre como solucionar o problema. Além disso, é importante ressaltar que o mecanismo de correção automática dificilmente seria útil para avaliação de exercícios de programação mais complexos do que aqueles apresentados aos alunos nas primeiras semanas de aula, já que neste caso a quantidade de variáveis e a complexidade das estruturas dos programas inviabilizariam a definição prévia dos esquemas de correção automática da maneira como estão propostos neste artigo.

O mecanismo de correção poderia, por exemplo, ser utilizado em contextos como jogos, ambientes integrados de desenvolvimento e ferramentas para a EAD. No caso dos jogos, o mecanismo de correção automática pode ser utilizado para determinar se o jogador avança ou não nas fases em função do resultado de seus programas. Já no caso dos ambientes integrados de desenvolvimento (conhecidos como IDEs) é possível utilizar o mecanismo de correção para prover *feedback* além daquele tradicionalmente oferecido, como completamento de código. Acredita-se que o mecanismo de correção automática pode ser de grande valia na EAD em virtude do grande número de alunos e da impossibilidade de um professor oferecer *feedback* adequado para todos. Neste caso, o mecanismo de correção poderia ser implementado em objetos de aprendizagem que por sua vez poderiam ser distribuídos nos ambientes virtuais de aprendizagem.

De maneira geral, conclui-se que a correção automática de pequenos exercícios práticos de programação é possível através da utilização do mecanismo proposto neste artigo. Apesar de não se ter mensurado especificamente o impacto do mecanismo de correção automática na aprendizagem dos alunos, as observações realizadas durante os experimentos permitem sugerir que o mecanismo pode ser de grande valia no processo de aprendizagem de programação, possivelmente diminuindo as frustrações durante a realização de exercícios práticos, e eventualmente contribuindo para a diminuição dos índices de evasão e reprovação em disciplinas de programação introdutória.

Referências

BAXTER, Ira D. et al. Clone Detection Using Abstract Syntax Trees. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 1998. Proceedings… Washington: IEEE Computer Society, 1998. p. 368.

BEAUBOUEF, Theresa; MASON, John. Why the high attrition rate for computer science students: some thoughts and observations. In: ACM SIGCSE Bulletin, USA, v. 37, n. 2, p. 103-106, jun. 2005.

CASTRO, T. H. C. et al. "Arquitetura SAAP - Sistema de Apoio à Aprendizagem de Programação". XXII Congresso da Sociedade Brasileira de Computação, volume 5 - VIII Workshop de Informática na Escola, Florianópolis, 2002.

CRISTÓVÃO, Henrique Monteiro. Aprendizagem de Algoritmos num Contexto Significativo e Motivador: um relato de experiência. In: CONGRESSO DA SBC – WEI – WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO, 18., Belém do Pará, Pará. Anais... 2008.

DALY, Charlie; HORGAN, Jane M. An automated learning system for Java programming. IEEE Transactions on Education, USA, p. 10-17, feb. 2004.

DOUCE, Christopher; LIVINGSTONE, David; ORWELL, James. Automatic test-based assessment of programming: a review. Journal on Educational Resources in Computing (JERIC), USA, v. 5, n. 3, p. 4-es, sep. 2005.

JESUS, Elieser A. de; RAABE, André L. A. Avaliação Empírica da Utilização de um Jogo para Auxiliar a Aprendizagem de Programação. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 2010, João Pessoa. Anais, 2010.

MOTA, Marcelle. P.; BRITO, Silvana R. de; MIREILLE, Pinheiro Moreira; FAVERO, Eloi Luiz. Ambiente de Aprendizagem de Programação com Visualização e Avaliação Automática de Programas. XX Simpósio Brasileiro de Informática na Educação, Florianópolis, SC, 2009.

NAVARRO, Gonzalo. A guided tour to approximate string matching. ACM Computing Surveys (CSUR), USA, v. 33, n. 1, p. 31-88, mar. 2001.

RAABE, André L. A.; SILVA, Júlia M. Carvalho da. Um ambiente para atendimento as dificuldades de aprendizagem de algoritmos. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 25., São Leopoldo, Rio Grande do Sul. Anais... RS:SBC, 2005.

RAHMAN, Khirulnizam A.; NORDIN, Md. Jan. A Review on the Static Analysis Approach in the Automated Programming Assessment Systems. In: NATIONAL CONFERENCE ON PROGRAMMING 07, Kuala Lumpur, Malaysia. Proceedings… dez. 2007.

RISTAD, Eric Sven; YIANILOS, Peter N. Learning string edit distance. IEEE Transactions on Pattern Analysis and Machine Intelligence, USA, v. 20, n. 5, p. 522-532, 1998.

SAGER, Tobias et al. Detecting similar Java classes using tree algorithms. In: INTERNATIONAL WORKSHOP ON MINING SOFTWARE REPOSITORIES, Shanghai, China. Proceedings… USA:ACM, 2006. p. 65-71.

SAIKKONEN, Riku; MALMI, Lauri; KORHONEN, Ari. Fully automatic assessment of programming exercises. ACM SIGCSE Bulletin, USA, v. 33, n. 3, p. 133-136, 2001.

TRUONG, Nghi; ROE Paul; BANCROFT, Peter. Static analysis of students' Java programs. In: CONFERENCE ON AUSTRALASIAN COMPUTING EDUCATION, 6., Dunedin, New Zealand. Proceedings… Darlinghurst: Australian Computer Society, jan. 2004. p. 317-325.