Um Operador de Mutação para Algoritmos Evolucionários na Seleção de Casos de Teste da Análise de Mutantes

Beatriz Proto Martins[1], André Assis Lôbo de Oliveira[1], Plínio de Sá Leitão Júnior[1], Celso Gonçalves Camilo-Junior[1], Auri Marcello Rizzo Vincenzi[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)

Caixa Postal 131 – 74.001-970 – Goiânia – GO – Brasil

{beatrizmartins,andreoliveira,plinio,celso,auri}@inf.ufg.br

Abstract. *This paper is situated in the field of Evolutionary Algorithms for Test Cases Selection on Mutation Analysis. It is proposed a hybrid mutation operator, aiming to achieve variability as well as effectiveness on the selected test cases subsets, by using both random and classificatory mechanisms. Experiments were performed in five algorithms, applied to real benchmarks, totaling 11520 executions. Althought there is a high computational cost in verifying each test case contribution on the selected subsets, the operator was sucessful when applied to most benchmarks, reducing costs up to 62%.*

Resumo. *Este artigo situa-se no campo dos Algoritmos Evolucionários para Seleção de Casos de Teste através da Análise de Mutantes. É proposto um operador de mutação híbrido, com o intuito de propiciar variabilidade e eficácia dos subconjuntos de casos de teste selecionados, ao utilizar tanto mecanismos aleatórios quanto mecanismos classificatórios. Foram realizados experimentos sobre cinco algoritmos, aplicados a benchmarks reais, totalizando 11520 execuções. Apesar do custo computacional de verificar a contribuição de cada caso de teste nos subconjuntos selecionados, o operador obteve êxito na maioria dos experimentos, ocasionando uma redução de custos de até 62%.*

1. Introdução

Durante a etapa de teste de um software, pode ser necessário um grande conjunto de casos de teste. O desafio consiste em encontrar um subconjunto capaz de representar os demais, sendo este um dos problemas mais complexos da área de Teste de Software. De acordo com Pezzè e Young (2008), provar que um subconjunto de casos de teste garante a corretude do programa é tão difícil quanto provar que o programa está correto e, para tanto, deveria ser possível provar a corretude sem teste algum.

Um bom caso de teste é aquele com grande possibilidade de revelar um defeito no software [Delamaro, Maldonado e Jino 2007], assim, informações a respeito da qualidade dos casos de teste podem ser obtidas analisando falhas anteriores do software. Essas especificações delineiam a técnica baseada em defeitos, na qual se destaca a Análise de Mutantes.

Apesar de não haver a garantia de se encontrar um subconjunto ótimo de casos de teste, é possível achar uma solução que melhor se aproxime da esperada. Tal artifício é utilizado pela *Search-Based Software Testing* (SBST), uma abordagem genérica que soluciona problemas de Teste de Software através de técnicas de busca e otimização meta-heurísticas [McMinn 2011].

A presente pesquisa propõe o Operador de Mutação Aleatório e Classificatório

(OMAC), um operador híbrido para Algoritmos Evolucionários (AEs) utilizados pela SBST na Seleção de Casos de Teste da Análise de Mutantes. Os AEs são então comparados com e sem o uso do operador, juntamente com o algoritmo aleatório, a fim de encontrar as melhores configurações possíveis. Entre os resultados, foi obtido uma redução de tempo de até 62% sobre um benchmark de grande porte utilizando o Algoritmo Genético canônico.

2. Revisão

2.1. Análise de Mutantes

A Análise de Mutantes ou Teste de Mutação foi primeiramente apresentada no artigo de DeMillo, Lipton, e Sayward (1978). A técnica baseia-se na "hipótese do programador competente", partindo do princípio de que programadores experientes escrevem programas corretos ou bem próximos do correto, e no "efeito de acoplamento" em que defeitos complexos são causados por defeitos simples.

Nessa técnica são aplicadas variações sintaticamente corretas no programa em teste para simular possíveis falhas, cada variação resulta em um programa chamado mutante. Um caso de teste deve provocar uma saída diferente em relação ao programa original, caso contrário o mutante é equivalente ou devem ser aplicados novos casos de testes, pois a falha pode estar no programa original. Quando as saídas são diferentes, o mutante é morto, caso contrário permanece vivo [DeMillo, Lipton, e Sayward, 1978].

Na Análise de Mutantes é possível avaliar um conjunto de casos de teste através do escore de mutação, calculado na Equação 1. O escore varia de 0 a 1 e quanto maior ele for, maior é capacidade do conjunto de casos de teste de matar mutantes [Delamaro, Maldonado e Jino, 2007].

$$ms(P,T) = \frac{\mathrm{DM(P,T)}}{\mathrm{M(P)} - EM(P)} \quad (1)$$

Onde:
- *ms*(*P*, *T*): escore de mutação do conjunto de testes T em relação ao programa P;
- *DM*(*P*, *T*): total de mutantes mortos pelo conjunto de casos de teste T;
- *M(P)*: total de mutantes gerados a partir do programa *P;*
- *EM*(*P*): total de mutantes equivalentes a P.

2.2. Trabalhos Correlatos

Os AEs formam uma subárea da Computação Evolucionária que, por ter sido criada recentemente nos anos 60, tem sido destaque em trabalhos acadêmicos. Os algoritmos genéticos são os mais estudados, suas várias vertentes e aplicações, como em aprendizagem e redes neurais, são apresentadas por Tanomaru (1995).

McMinn (2011) destaca as soluções oferecidas pela SBST na geração e seleção de casos de teste. Em seu trabalho, é possível notar que, ao contrário dos algoritmos aleatórios, os AEs oferecem uma solução razoável para o Teste de Software, especificamente para a Seleção de Casos de Teste, uma vez que as soluções para esse problema normalmente ocupam uma pequena fração de um grande espaço de busca.

Em seu trabalho, Oliveira (2013) expõe uma visão abrangente da aplicação de AEs na Análise de Mutantes e contribui para esse critério de teste com o conceito de

Efetividade Genética, ao propor um Algoritmo Genético Coevolucionário (AGC).

3. Abordagem Proposta

Dado um conjunto de casos de teste, surge a necessidade de reduzi-lo a um subconjunto capaz de representar os demais. A SBST resolve esse problema razoavelmente através de AEs em suas configurações convencionais. Entretanto, quando aplicados na Seleção de Casos de Teste, é possível guiar ainda mais a busca por meio da classificação dos casos de teste em termos de suas contribuições na matança de mutantes. Um caso de teste contribui para o conjunto quando é o único a matar pelo menos um mutante [Oliveira, Camilo-Júnior e Vincenzi 2013a].

Um subconjunto de casos de teste pode ser visto como um indivíduo nos AEs, enquanto que um caso de teste é representado por um cromossomo. Normalmente, durante a operação de mutação do AE, é alterado um cromossomo aleatoriamente. Nesse trabalho foi desenvolvido o OMAC que funciona da seguinte forma, dado um indivíduo a ser mutado: 1) sorteia-se um número; 2) se o número estiver acima da porcentagem de classificação escolhida, um caso de teste é mutado aleatoriamente; 3) senão, é feita a mutação dos casos de testes que não contribuem para o subconjunto.

O método classificatório do operador funciona do seguinte modo: 1) recupera-se os mutantes mortos por cada caso de teste do subconjunto; 2) realiza-se a ordenação decrescente do subconjunto de acordo com os casos de teste que matam mais mutantes; 3) armazena-se os números dos mutantes mortos pelo caso de teste que mata mais; 4) para cada caso de teste ordenado restante é feito o que segue; 4.1) analisa-se se o caso de teste mata algum mutante além dos que já foram mortos; 4.2) se sim, os números dos novos mutantes mortos são guardados; 4.3) senão, o caso de teste é mutado.

4. Metodologia

4.1. Algoritmos e Benchmarks

Eiben e Smith (2003) explicam o funcionamento dos AEs que, aplicados no Teste de Mutação, podem ser descritos da seguinte forma: 1) gera-se uma população composta por subconjuntos de casos de teste (indivíduos) aleatórios; 2) cada candidato é avaliado de acordo com a Equação 1 (função de aptidão); 3) enquanto o tempo ou escore de mutação máximos não são atingidos (condição de parada); 3.1) são selecionados subconjuntos (pais); 3.2) os subconjuntos são recombinados e/ou; 3.3) mutados e; 3.4) passados para a próxima geração (filhos).

Nos experimentos foram comparados cinco algoritmos, sendo quatro evolucionários e um aleatório a fim de verificar a eficácia do OMAC. São eles:

- AG: algoritmo genético canônico que utiliza os 3 operadores (seleção, recombinação e mutação) e na qual a seleção é aleatória [Tanomaru 1995];
- EE: estratégia evolutiva com seleções determinísticas (μ,λ)-EE e (μ+λ)-EE onde μ pais geram λ filhos, sendo que no primeiro caso os filhos substituem os pais e, no segundo caso, pais e filhos competem entre si [Eiben e Smith 2003].
- EST: algoritmo genético com reprodução de estado estável em que, normalmente, somente um indivíduo é trocado por geração [Tanomaru 1995].
- MI: algoritmo genético no modelo de ilhas em que subpopulações (ilhas) são evoluídas paralelamente e indivíduos migram entre elas, através de uma

topologia de comunicação, após cada época, isto é, após um certo número de gerações [Tanomaru 1995].

- AL: consiste no algoritmo de abordagem aleatória utilizando alguns conceitos dos AEs. O algoritmo pode ser descrito em 3 passos: 1) gera-se uma população aleatória; 2) calcula-se a aptidão dos indivíduos conforme a Equação 1; 3) escolhe-se o melhor indivíduo entre todos os avaliados; 4) se a condição de parada não for satisfeita, retorna-se ao passo 1.

Todos os experimentos foram executados em sistema desktop modelo Dell Precision T7500 de 64 bits com as configurações padrões de fábrica. Os AEs foram executados através do Watchmaker Framework[1], um Software Livre na linguagem Java desenvolvido por Daniel Dyer e disponibilizado sob a licença Apache 2.0.

Nos experimentos foram utilizados oito programas, dos quais quatro são utilitários UNIX escritos em linguagem C, são eles: a) *cal*; b) *comm*; c) *look* e; d) *uniq*. Para geração dos mutantes foi utilizada a ferramenta Proteum [Oliveira 2013 apud Delamaro 1996]. Os demais programas são escritos em linguagem Java e seus mutantes foram gerados pela ferramenta MuJava [Oliveira, Camilo-Júnior e Vincenzi 2013b apud Ma 1997], são eles: a) *bubcorrecto*; b) *fourballs*; c) *mid* e; d) *trityp*. Os benchmarks resultantes contêm os status dos mutantes quando executados com cada caso de teste, os detalhes se encontram na Tabela 1, onde o subconjunto mínimo corresponde ao subconjunto mínimo de casos de teste capaz de matar todos os mutantes.

Tabela 1. Informações sobre os benchmarks usados nos experimentos

	cal	comm	look	uniq	bubcorrecto	fourballs	mid	trityp
Nº de mutantes	4622	1869	1980	1618	80	212	181	309
Nº de equivalentes	344	222	233	224	12	44	43	70
Nº de casos de teste	2000	801	255	490	255	96	125	216
Escore máximo	0,99742	1,0	1,0	1,0	1,0	1,0	1,0	1,0
Suconjunto mínimo	20	22	22	4	1	5	5	17

4.2. Parâmetros dos algoritmos

Em cada experimento foram realizadas 30 execuções com, no máximo, 180 segundos cada uma (em alguns casos esse limite pode ser ultrapassado porque a verificação é feita após o final de uma geração que pode ser longa). No algoritmo AL foram aplicados os mesmos parâmetros dos AEs para tamanho da população, tamanho do indivíduo e condição de parada.

Os experimentos foram realizados com e sem o uso do OMAC. Dado um indivíduo a ser mutado com o operador, são aplicadas as seguintes porcentagens de uso dos mecanismos classificatórios: 10, 40 ou 70%. Nota-se que utilizar 0% de classificação seria equivalente a usar um operador de mutação tradicional.

Foram estabelecidos os seguintes parâmetros para os AEs: a) operador de seleção: torneio com 2 competidores; b) taxa de cruzamento: 95%; c) taxa de mutação: 5% e; d) tamanho do indivíduo: de acordo com o subconjunto mínimo da Tabela 1, porém respeitando o tamanho mínimo 2 para possibilitar a recombinação.

O tamanho da população está de acordo com a Equação 2 em que a quantidade

1 Disponível em: <http://watchmaker.uncommons.org/>.

total de casos de teste em uma população, isto é, o tamanho da população multiplicado pelo tamanho do indivíduo, corresponde a uma porcentagem do total de casos de testes do benchmark em questão. Tais variações foram realizadas para dificultar ou facilitar as disputas de forma a ocupar o intervalo de tempo disponível por execução. Nesse sentido k foi fixado em 0.05 quando $T < 300$; 0.3 quando $300 <= T < 1000$ e; 0.5 quando $T >= 1000$). Assim, o tamanho da população para cada benchmark é o seguinte: a) *cal*: 51; b) *comm*: 12; c) *look*: 5; d) *uniq*: 39; e) *bubcorrecto*: 8; f) *fourballs*: 2; g) *mid*: 3 e; h) *trityp*: 2.

$$pop(B) = k * T \div min + 1 \quad (2)$$

Onde:

- *pop(B)*: população do algoritmo utilizando o benchmark B;
- *k*: porcentual de casos de teste para disputas;
- *T*: quantidade total de casos de teste;
- *min*: subconjunto mínimo de acordo com a Tabela 1;

Para o algoritmo MI foram estabelecidas 2 ilhas com estratégia de migração em anel. Além disso, alguns parâmetros específicos por algoritmo foram variados:

- EE1: a) estratégia: (1, 7)-EE (valor recomendado por Eiben e Smith (2003));
- EE2: a) estratégia: (1+7)-EE;
- EE3: a) estratégia: (1+1)-EE;
- EST1: a) pais selecionados: 2 e; b) filhos gerados: 2;
- EST2: a) pais selecionados: 2 e; b) filhos gerados: 1;
- MI1: a) épocas: 25 e; b) migração: 5;
- MI2: a) épocas: 50 e; b) migração: 5;
- MI3: a) épocas: 50 e; b) migração: 1.

5. Resultados

O algoritmo AL se mostrou mais eficiente nos benchmarks da ferramenta MuJava com exceção do *Trityp*, quando comparado aos AEs (mostrados adiante), os resultados podem ser vistos na Tabela 2. Entretanto, todos os benchmarks serão ainda abordados neste artigo para fins de comparação.

A Tabela 3 apresenta as comparações do algoritmo EE2 para cada benchmark, em que a esquerda de cada coluna estão os resultados sem o uso do OMAC e a direita com 40% de uso da classificação. O operador obteve resultados melhores.

Tabela 2. Escore máximo alcançado (MAX), escore médio (MED), desvio padrão (DESVP) e tempo médio (TEMPO) em segundos de cada benchmark

	MAX	MED	DESVP	TEMPO
bubcorrecto	1	1	0	0,0056
fourballs	1	1	0	0,0283
mid	1	1	0	4,1999
trityp	0,9665272	0,9566248	0,0008222	180,0073
cal	0,973352	0,9664251	0,0004485	180,2266
uniq	1	0,9997848	0,0000032	104,3589
look	0,9828277	0,9615531	0,0010145	180,0127

Tabela 3. Comparação entre o uso e não uso do OMAC com o algoritmo EE2.

	MAX		MED		DESVP		TEMPO	
bubcorrecto	1	1	1	1	0	0	18,0996	17,3545
fourballs	1	1	0	0	0	0	14,9078	12,564
mid	1	1	0	0	0	0	7,4448	10,6777
trityp	1	1	1	1	0	0	18,5671	18,9456
cal	0,9974287	0,997429	0,995286	0,9940704	0,0001663	0,00037	176,2194	168,4677
uniq	1	1	0	0	0	0	19,362	20,4549
look	1	1	0,9999809	1	0,0000003	0	43,2228	37,2112
comm	1	1	0,999616	0,999717	0,0000107	0,0000094	139,6696	120,3308

Dentre os parâmetros apresentados na Seção 4.2, as menores médias de tempo em cada benchmark foram as dos seguintes algoritmos: a) EE2; b) EST1 e; c) MI1.

A Tabela 4 apresenta para cada benchmark a porcentagem de experimentos, na coluna SUCESSO, em que o uso do OMAC, independentemente da taxa de classificação, possui um tempo médio menor (ou escore de mutação médio maior, para os casos em que o tempo máximo é atingido) do que com o uso do operador tradicional. Também é mostrada a razão entre o tamanho do indivíduo e quantidade total de casos de teste (TAM_IND / QTD_CT), além da quantidade total de mutantes em relação ao tamanho do indivíduo (QTD_MUT / TAM_IND).

Observando-se a Tabela 4, nota-se uma forte influência do tamanho do indivíduo em relação a porcentagem de sucesso, principalmente para os benchmarks *cal* e *uniq* em que o indivíduo abrange menos de 1% do total de casos de teste e cada caso de teste deve matar, em média, uma grande quantidade de mutantes. Além disso, o custo computacional para se classificar muitos indivíduos é maior do que o custo para se classificar um indivíduo grande e, nesses experimentos, o tamanho da população é inversamente proporcional ao tamanho do indivíduo.

Tabela 4. Possíveis influências no sucesso obtido pelo OMAC.

	cal	comm	look	uniq	bubcorrecto	fourballs	mid	trityp
SUCESSO	20%	90%	90%	0%	80%	80%	40%	40%
TAM_IND / QTD_CT	0,01	0,027	0,086	0,008	0,0078	0,052	0,04	0,078
QTD_MUT / TAM_IND	231,1	84,9	90	404,5	40	42,4	36,2	18,2

O benchmark *cal* é o mais complexo para seleção de casos de teste dentre os benchmarks utilizados. Por isso, ele foi utilizado para realização de novas experimentações com os melhores parâmetros encontrados anteriormente. Os experimentos foram configurados aumentando o tamanho do indivíduo para 34 e diminuindo o tamanho da população para 31 visando manter, aproximadamente, a mesma quantidade de casos de teste em uma população. Obteve-se um resultado positivo, pois além de diminuir o tempo médio, tal configuração permitiu a visualização do aumento no escore de mutação médio provocada pelo OMAC (Gráficos 1 e 2). Observando-se a convergência da curva, nota-se que uma taxa de cerca de 70% na aplicação da classificação pode representar uma relação custo-benefício razoável, uma vez que aumentar a taxa de classificação, aumenta a quantidade de cálculos.

Gráfico 1. Tempo médio do benchmark *cal* com indivíduos maiores

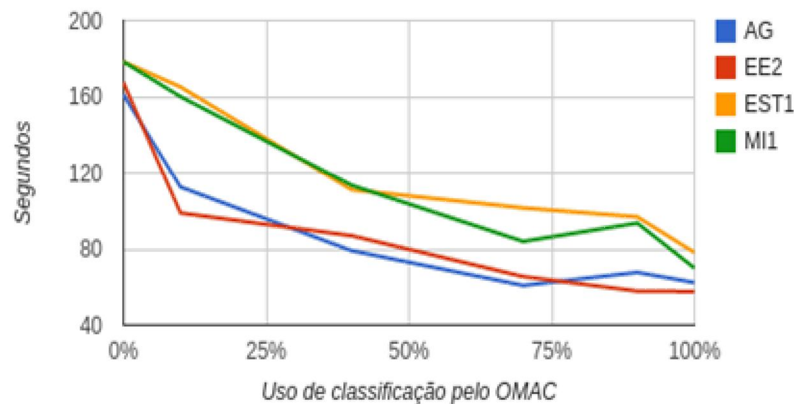
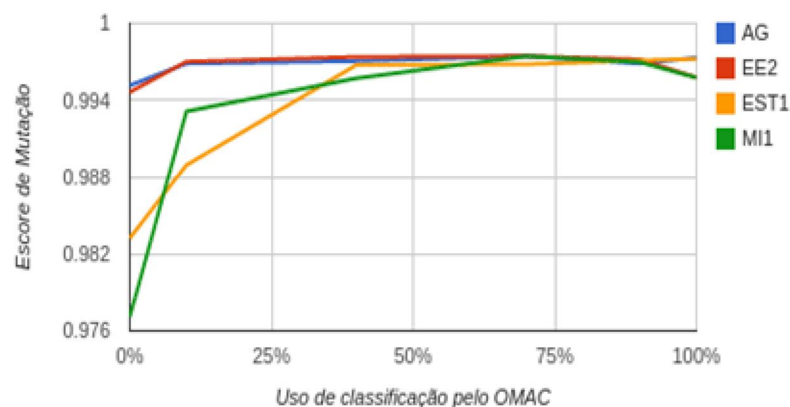


Gráfico 2. Escore de mutação médio do benchmark *cal* com indivíduos maiores



6. Conclusões e Trabalhos Futuros

Com a aplicação do OMAC, foi proporcionada uma alta variabilidade, isto é, foram evitadas estagnações em máximos locais, uma vez que são mutados aleatoriamente tanto casos de testes bons, quanto ruins, mas mutando principalmente os ruins através da classificação. Na maioria dos experimentos isso ocasionou tempos de execuções menores sem piorar o escore de mutação.

Frente a isso, acredita-se que seja vantajoso utilizar o OMAC, principalmente em benchmarks de grande porte, desde que sejam devidamente parametrizados, isto é, com o tamanho do indivíduo maior do que o tamanho da população em cada geração, os quais representam 50% dos casos de teste do benchmark, e aplicando a classificação pelo OMAC em 70% dos casos em que é requisitada a mutação. Observou-se também o bom desempenho dos algoritmos genéticos canônico e em ilhas, além das estratégias evolucionárias com os parâmetros previamente recomendados.

Como trabalhos futuros, pretende-se investigar quais fatores influenciam na escolha da taxa de classificação e comparar com diferentes taxas de mutação. Pretende-se, ainda, desenvolver um operador de seleção híbrido, análogo ao OMAC.

Referências

M. Pezzè, M. Young (2008) "Teste e análise de software: processo, princípios e técnicas". Ed. Bookman, Porto Alegre, RS.

M. Delamaro, J. Maldonado, M. Jino (2007) "Introdução ao teste de software". Rio de Janeiro: Elsevier Campus, 2007. 394p.

P. McMinn (2011) "Search-Based Software Testing: past, present and future. In: International Conference on Software Testing, Verication and Validation Workshops (ICSTW) p. 153-163.

R. A. DeMillo, R. J. Lipton, F. G. Sayward (1978) "Hints on test data selection: help for the practicing programmer". In: IEEE Computer Society Press Los Alamitos, p. 34-41.

J. Tanomaru (1995) "Motivação, Fundamentos e Aplicações de Algoritmos Genéticos" II Congresso Brasileiro de Redes Neurais.

A. Oliveira (2013) "Uma Abordagem Coevolucionária para seleção de Casos de Teste e Mutantes no Contexto do Teste de Mutação". Dissertação, 155 f. - Goiânia, Universidade Federal de Goiás, Instituto de Informática.

A. Oliveira, C. Camilo-Junior e A. Vincenzi (2013a) "Um Algoritmo Genético Coevolucionário com Classificação Genética Controlada aplicado ao Teste de Mutação". Congresso Brasileiro de Software: Teoria e Prática (CBSoft).

A. Oliveira, C. Camilo-Junior and Vincenzi (2013b) "A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing". In: IEEE Congress on Evolutionary Computation (CEC).