Um Algoritmo Evolucionário para Encontrar Sequências de Transformações com Ordenação por Classe

João Fabrício Filho
Universidade Tecnológica Federal do Paraná Câmpus Campo Mourão
Via Rosalina Maria dos Santos, 1233 - Campo Mourão/PR
joaof@utfpr.edu.br

Anderson Faustino da Silva
Universidade Estadual de Maringá - Departamento de Informática
Av. Colombo, 5790 Bloco C56 - Maringá/PR
anderson@din.uem.br

RESUMO

Algoritmos Evolucionários são metaheurísticas amplamente utilizadas para solucionar problemas combinatórios e de otimização. Um problema combinatório, no contexto da Ciência da Computação, é a escolha da sequência de transformações que deve ser utilizada pelo compilador durante a geração de código final. O objetivo deste artigo é propor e avaliar um algoritmo evolucionário capaz de encontrar efetivas sequências de transformações, o qual é potencializado por uma classificação na qual seja respeitada uma ordem de aplicação de transformações. Os resultados com o algoritmo proposto indicam que este é eficiente em encontrar boas sequências de transformações, além de ser uma boa opção para a geração de bases de dados para sistemas de aprendizagem de máquina.

PALAVRAS CHAVE. Algoritmos Evolucionários. Transformações de Código. Compilação Iterativa.

Tópicos: Metaheurísticas

ABSTRACT

Evolutionary Algorithms are metaheuristics widely used to solve combinatorial problems and optimizational. A combinatorial problem, on the context of Computer Science, is the choose for a transformations sequence that is used by the compiler during the target code generation. This work aims to propose and evaluate an evolutionary algorithm capable of find effective transformations sequences, that is potentialized by a classification, which respect an application order of transformation classes. The results indicates that the proposed technique is efficient to find good transformations sequences, besides to be a good option to databases generation to machine learning systems.

KEYWORDS. Evolutionary Algorithms. Code Transformations. Iterative Compilation.

Paper topics: Metaheuristics

1. Introdução

Metaheurísticas são métodos não exatos formulados para resolver problemas de otimização, atuando como um *framework* estocástico [Luke, 2013; Meena et al., 2011; Rajakumar, 2013; Jia et al., 2016; Cheraitia e Haddadi, 2016; Bessedik et al., 2011].

Algoritmos Evolucionários (AE), um tipo de metaheurística, são inspirados na seleção natural e amplamente utilizados para solucionar problemas de otimização [Russell et al., 1996; Gu et al., 2016; Wang et al., 2017]. Geralmente, um AE consiste em iterações sobre um operador de *crossover*, uma mutação por uma probabilidade e um método de seleção por uma função de aptidão; operações que ocorrem sobre uma população de indivíduos, na qual cada indivíduo é composto por uma sequência de genes, chamada de cromossomo.

Um problema apropriado para se aplicar AE é a escolha de quais transformações de código o compilador deve aplicar ao código de entrada de forma a gerar um código final de boa qualidade [Aho et al., 2006] [Muchnick, 1997]. É importante destacar que a escolha de transformações não é um problema trivial, devido basicamente a dois fatores:

1. Quais transformações devem ser selecionadas; e
2. Em qual ordem as transformações selecionadas devem ser aplicadas.

Além disto, é importante ressaltar que uma sequência de transformações pode ser benéfica a um determinado programa mas não a outro. Isto indica, que tal problema é dependente de programa.

Este artigo tem por objetivo propor um AE para selecionar transformações a serem aplicadas a um código de entrada, de forma a aumentar o desempenho do código final em termos de tempo de execução. Isso significa que o compilador, ao utilizar o AE proposto, será capaz de escolher quais transformações utilizar e em qual ordem aplicá-las. Com isso se tem por objetivo gerar um código final cujo tempo de execução seja menor do que o obtido utilizando a sequência mais agressiva disponibilizada pelo compilador em questão.

As principais contribuições deste artigo são:

- Apresenta um AE aplicado ao problema de seleção de transformações;
- Descreve um mecanismo de *crossover* para cromossomos de tamanho variável, utilizando genes que possuem classificação com ordenação pré-determinada;
- Propõe uma classificação de transformações indicando em que ordem elas devem ser aplicadas;
- Apresenta um método para gerar sequências de transformações capazes de obter desempenho para os programas de entrada; e
- Apresenta uma avaliação experimental detalhada, descrevendo os resultados que podem ser alcançados pelo método apresentado.

Os experimentos realizados indicam que o AE proposto é capaz de alcançar um desempenho superior a outras técnicas aplicadas ao mesmo problema, além de ser uma boa opção para geração de base de dados para sistemas de aprendizagem de máquina.

2. Encontrando a Melhor Sequência de Transformações

Compiladores modernos disponibilizam dezenas de algoritmos de transformação. Contudo, devido à dependência existente nas estruturas que compõem o código fonte de um programa, é necessário estudar estratégias de explorar o espaço de busca de transformações com o objetivo de encontrar as melhores para se aplicar a um código de entrada. Uma boa estratégia é classificar as transformações disponibilizadas pelo compilador de uma forma que se mantenha uma ordem pré-estabelecida de aplicação de cada classe.

2.1. Classificação das Transformações

Considerando o contexto das transformações de código, Muchnick [1997] propõe que primeiro sejam aplicadas transformações que alterem as estruturas básicas do programa e somente ao final deste passo sejam aplicadas transformações que modificam estruturas mais complexas (como por exemplo estruturas de controle de fluxo).

Considerando as observações de Muchnick, este artigo propõe a seguinte classificação de transformações (a qual indica a ordem de aplicação das transformações):

A transformações fundamentais que não afetem estruturas complexas do programa.

B transformações que afetam estruturas de dados e procedimentos.

C transformações que afetam estruturas básicas do programa.

D transformações que reduzem a quantidade de código.

E transformações que eliminam o código morto e modificam estruturas de controle de fluxo.

F transformações que modificando estruturas complexas de controle de fluxo.

G transformações que modificam o acesso a dados e referências globais.

É importante destacar três pontos. Primeiro, a classificação proposta indica a ordem de aplicação de cada classe de transformação. Neste caso, uma transformação da classe A nunca pode aparecer após uma transformação da classe B. Segundo, duas transformações que pertencem a mesma classe pode aparecer em qualquer ordem entre elas. Terceiro, embora a classificação proposta siga as premissas descritas por Muchnick, ela difere da classificação por ele proposta no tocante às otimizações que pertencem a cada classe.

2.2. AE para Encontrar Sequências de Transformações

O AE proposto consiste em evoluir uma população de indivíduos, um número específico de gerações. Indivíduos são compostos por cromossomos, os quais são compostos por uma lista de genes (transformações). A cada geração, dois indivíduos são escolhidos e um operador de *crossover* é aplicado sobre eles. Após, uma mutação pode ocorrer em cada novo indivíduo. O Algoritmo 1 apresenta o AE proposto, cujo objetivo é encontrar sequências de transformações para um determinado programa.

Algoritmo 1: A estrutura do AE proposto.

```
Input: tamanhoPopulacao
Output: Melhor sequência encontrada
populacao ← inicializaPopulacao(tamanhoPopulacao)
while estagnacao < max_estagnacao do
    pai1, pai2 ← escolhePais(populacao)
    indiv1, indiv2 ← crossover(pai1, pai2)
    mutacaoProbabilidade(indiv1, indiv2)
    melhorAptidaoGerada ← max(indiv1.aptidao(), indiv2.aptidao())
    melhorIndividuo ← populacao.mehorAdaptado()
    if melhorIndividuo.aptidao() < melhorAptidaoGerada then
        estagnacao ← 0
    else
        estagnacao++
    populacao ← selecao(populacao, indiv1, indiv2)
return A sequência de populacao.mehorAdaptado()
```

Na proposta deste trabalho, um indivíduo representa uma possível sequência de transformações a ser aplicada para gerar um código final. Portanto, cada gene representa um algoritmo de transformação disponibilizado pelo compilador utilizado. Além disso, cada indivíduo tem codificado em seu cromossomo as classes de transformações, de forma que a ordenação de transformações seja respeitada em futuras modificações do cromossomo.

A função de aptidão retorna um valor inversamente proporcional ao tempo de execução do código gerado (após compilado e transformado pela sequência), e será a base para a seleção dos indivíduos da próxima geração.

A cada geração, dois pais são escolhidos por uma probabilidade baseada no valor de aptidão dos indivíduos da atual população. Indivíduos com maior aptidão possuem maiores chances de serem escolhidos.

A operação de mutação consiste em modificar um gene aleatório do cromossomo. Neste caso, uma classe é escolhida aleatoriamente para a mutação. Após, uma posição é escolhida na faixa dessa classe. Por fim, uma operação é selecionada entre (1) troca, (2) inserção e (3) remoção de um gene do cromossomo, para ser aplicada a esta posição. Se a operação for troca ou inserção, uma transformação da mesma classe é escolhida aleatoriamente para ser inserida na sequência (substituindo ou deslocando o gene da mesma posição). O critério de parada é a estagnação da solução, que vai dizer se o AE está melhorando a solução ou não.

2.3. O mecanismo de *crossover*

O *crossover* consiste em uma das operações principais em um AE, pois é ele que fará as principais modificações em cada indivíduo.

Embora, como proposto neste artigo seja necessário manter a ordenação de classes nos genes, é necessário propor um operador de *crossover* capaz de de modificar indivíduos consideravelmente. Assim, este artigo propõe uma operação de *crossover* em vários pontos dos cromossomos. Tal operação irá modificar, em cada geração, os cromossomos de cada classe separadamente. Assim, a ordenação de classes é mantida, porém o arranjo dos genes dentro das classes pode ser modificado.

Nos indivíduos, os cromossomos são compostos por genes divididos em classes, que possuem uma faixa de tamanho variável dentro da sequência. Dessa forma, o mecanismo de *crossover* escolhe uma posição aleatória em cada faixa das classes, gerando dois indivíduos conforme é mostrado na Figura 1.

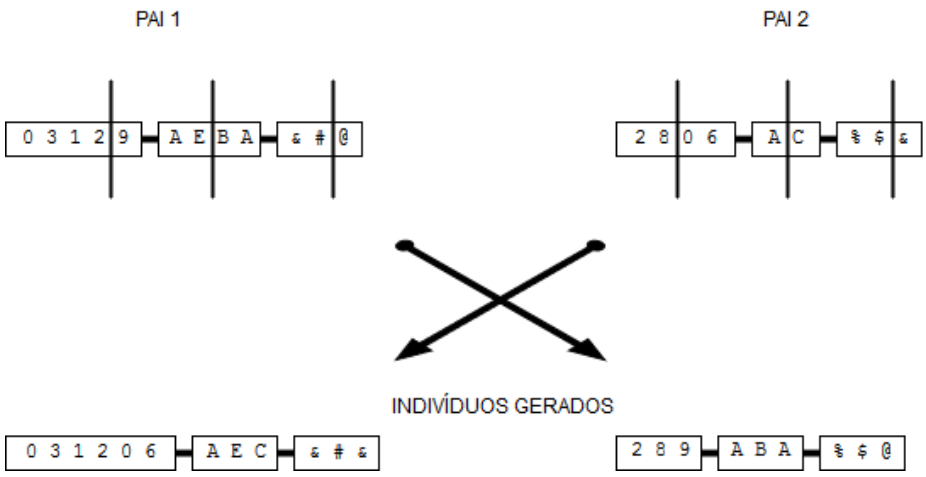


Figura 1: Exemplo de aplicação da operação de *crossover*.

Em cada faixa de classes, dois pontos são escolhidos - um para o primeiro pai, outro para o segundo pai. Assim, a operação é realizada gerando dois novos indivíduos. Como pode ser observado na Figura 1, o primeiro indivíduo gerado terá o início de cada faixa do primeiro pai e o final de cada faixa do segundo pai. Enquanto o segundo indivíduo gerado terá o início de cada faixa do segundo pai e o final de cada faixa do primeiro pai. O Algoritmo 2 apresenta o operador

de crossover.

Algoritmo 2: Mecanismo de *crossover* multi ponto para genes classificados com tamanho variável.

```
Input: pai1, pai2: Indivíduos selecionados da população
Output: Dois novos indivíduos gerados
indiv1 ← copy(pai1)
indiv2 ← copy(pai2)
ini1 ← 0
ini2 ← 0
for i ← 1 to N do
    fim1 ← indiv1.classes[i] - 1
    ponto1 ← aleatorio(ini1, fim1)
    fim2 ← indiv2.classes[i] - 1
    ponto2 ← aleatorio(ini2, fim2)
    indiv1, indiv2 ← do_crossover(indiv1, indiv2, ponto1, ponto2)
    ini1 ← indiv1.classes[i]
    ini2 ← indiv2.classes[i]
return indiv1, indiv2
```

Como pode ser observado, o *crossover* proposto mantém a ordenação de classes, modificando e potencializando os indivíduos em relação aos seus pais.

3. Avaliação Experimental

As seções seguintes descrevem as avaliações realizadas com o AE proposto, cuja finalidade é encontrar sequências de transformações para programas de entrada.

3.1. Configuração Experimental

Arquitetura Intel(R) Core(TM) i7-3770 CPU 3.4GHz com 8GB RAM executando sobre o sistema operacional Ubuntu 15.10 x64, com kernel 4.2.0-41.

Compilador A infraestrutura de compilação LLVM 4.0.0 [Lattner, 2017].

Programas A avaliação foi realizada com programas que pertencem à suíte de testes da LLVM [Lattner, 2017] e do *The Computer Language Benchmarks Game* [Gouy, 2017], os quais são apresentados na Tabela 1.

Tabela 1: Programas utilizados na avaliação da proposta.

Suíte de testes da LLVM			
ackermann (T00)	flops-3 (T14)	mandel (T28)	queens-mcgill (T42)
ary3 (T01)	flops-4 (T15)	mandel-2 (T29)	quicksort (T43)
bubblesort (T02)	flops-5 (T16)	matrix (T30)	random (T44)
chomp (T03)	flops-6 (T17)	methcall (T31)	realmm (T45)
dry (T04)	flops-7 (T18)	misr (T32)	recursive (T46)
dt (T05)	flops-8 (T19)	n-body (T33)	reedsolomon (T47)
fannkuch (T06)	fp-convert (T20)	nsieve-bits (T34)	richards_benchmark (T48)
fbench (T07)	hash (T21)	oourafft (T35)	salsa20 (T49)
ffbench (T08)	heapsort (T22)	oscar (T36)	sieve (T50)
fib2 (T09)	himenobmtxpa (T23)	partialsums (T37)	spectral-norm (T51)
fldry (T10)	huffbench (T24)	perlin (T38)	strcat (T52)
flops (T11)	intmm (T25)	perm (T39)	towers (T53)
flops-1 (T12)	lists (T26)	pi (T40)	treesort (T54)
flops-2 (T13)	lpbench (T27)	queens (T41)	whetstone (T55)
The Computer Language Benchmarks Game			
binary-trees (T56)	fasta-redux (T58)	pidigits (T60)	
fasta (T57)	mandelbrot (T59)	regex-dna (T61)	

População inicial A qualidade da população inicial pode ser essencial para obter bons resultados para um AE. Assim, a população contém as sequências -O1, -O2 e -O3 da LLVM na primeira geração. Os outros N indivíduos são gerados aleatoriamente sem repetição de transformações.

Critério de parada A estagnação é o critério de parada escolhido para o AE proposto. Após 30 gerações sem melhoria, o algoritmo retorna a melhor solução encontrada.

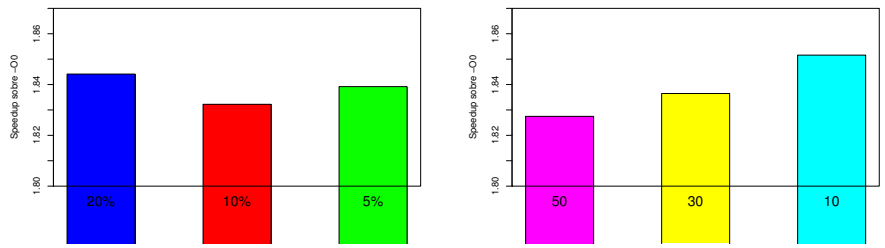
Métricas de avaliação A avaliação utiliza o *speedup* (relação entre o tempo de execução obtido pelo código final, quando gerado sem a aplicação de transformações, pelo tempo de execução obtido aplicando transformações no processo de geração de código final) e o tempo de execução para analisar o desempenho do algoritmo proposto. Baseado no *speedup* e no tempo de execução são utilizadas três métricas:

1. `MGS`: Média Geométrica de *Speedup*;
2. `NPS`: Número de programas com *speedup* maior do que o nível mais agressivo de transformação (`-O3`), ou seja, cobertura; e
3. `TR`: O tempo de resposta da técnica.

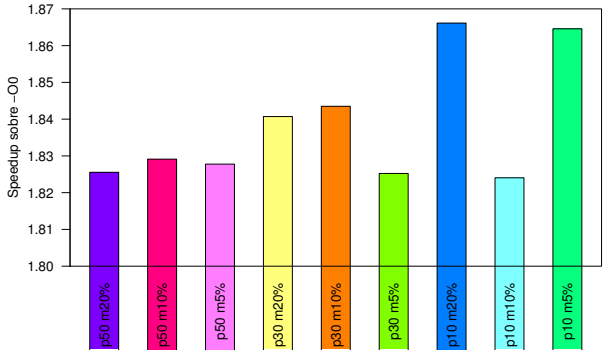
3.2. Calibragem

A calibragem do `AE` proposto considera dois parâmetros: tamanho da população e probabilidade de mutação. O tamanho da população deve ser suficiente para manter os melhores indivíduos gerados, para que a cada nova geração haja maiores chances de melhorar a solução. A probabilidade de mutação deve auxiliar para convergir a um ótimo global, escapando de possíveis ótimos locais.

Para calibrar a solução, um programa teste foi escolhido aleatoriamente. O tamanho da população foi avaliado com os valores 10, 30 e 50, enquanto a probabilidade de mutação foi avaliada com 5%, 10% e 20%. A medida de qualidade das soluções considera o desempenho alcançado obtido pela aplicação da estratégia com cada parâmetro e com a combinação dos resultados com todas as avaliações. Além dessa medida, se considera o tempo decorrido entre o início da execução do algoritmo e o retorno da solução encontrada. A Figura 2 apresenta os *speedups* obtidos pela combinação de cada parâmetro de calibragem.



(a) Probabilidade de Mutação (b) Tamanho da população



(c) Tamanho da População e Probabilidade de Mutação

Figura 2: Desempenho médio obtido considerando os parâmetros tamanho de população (p - 10, 30 e 50) e probabilidade de mutação (m - 5%, 10% e 2%).

Os resultados para diferentes probabilidades de mutação foram próximos, com uma diferença de 0,6% entre o pior resultado (m=10%) e o melhor resultado alcançado (m=20%). Quanto ao tamanho da população, com 10 indivíduos se obtivera os melhores resultados, com 0,8% melhor do que p=30 e 1,3% melhor do que p=50. Analisando uma combinação dos dois parâmetros, a combinação p=10 e m=20% obteve os melhores resultados, 1,5% melhor do que a média geométrica dos outros resultados.

O tempo médio decorrido para cada combinação é apresentado na Figura 3.

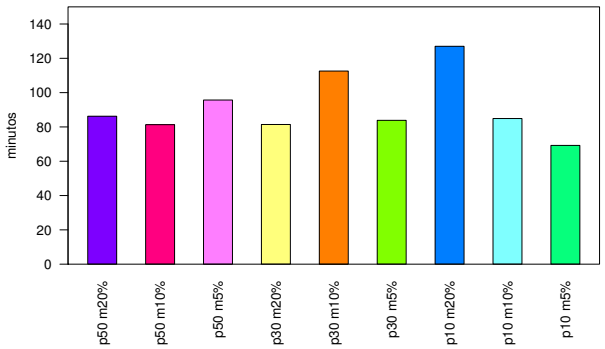


Figura 3: Tempo médio decorrido em combinação dos parâmetros de tamanho de população (10, 30 e 50) e probabilidade de mutação (5%, 10% e 20%).

Considerando o tempo dispendido em cada combinação, p=10 e m=5% obteve um tempo de resposta menor de que qualquer outra combinação avaliada, 23% menos do que a média das outras combinações. Por outro lado, a combinação p=10 e m=20% obteve os melhores desempenhos, mas precisou de um maior tempo de resposta.

Ponderando o tempo de resposta e o desempenho obtido, p=10 e m=5% foram os melhores valores para os parâmetros avaliados, pois tais valores propiciaram bons resultados em um menor espaço de tempo.

3.3. Avaliação da Classificação de Transformações Proposta

Para avaliar a efetividade classificação de transformações proposta neste artigo, o `AE` proposto foi avaliado utilizando três distintas abordagens para classificar transformações:

1. classificação proposta neste artigo (`Classificação1`);
2. classificação proposta por Muchnick [1997] (`Classificação2`);
3. sem classificação (`SemClassificação`).

Utilizando as classificações 1 ou 3, a população inicial contem as três sequências pré-definidas disponibilizadas pela infraestrutura `LLVM`. Porém utilizando a classificação 2, a população inicial é totalmente aleatória.

Por ser um método não exato, um `AE` pode apresentar diferentes resultados em cada execução. Por esse motivo, cada proposta de classificação foi executada 3 vezes para todos os programas avaliados, e os resultados analisados consideram a média das 3 execuções de cada programa. Além disso, foram utilizados os melhores parâmetros encontrados (população 10 e probabilidade de mutação 5%).

Os *speedups* médios alcançados para os `AE`s, utilizando diferentes classificações de transformações, são apresentados na Figura 4.

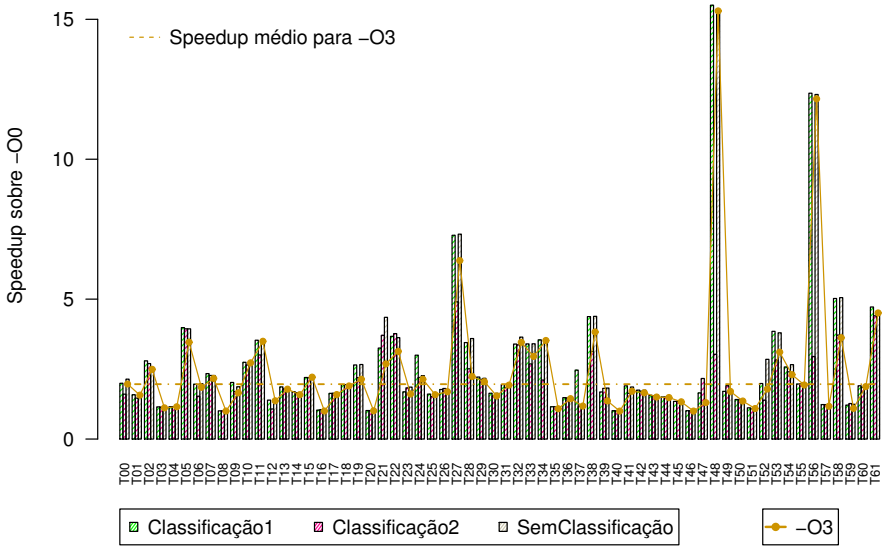


Figura 4: *Speedup* médio obtido para cada programa avaliado.

`MGS` Em relação ao *speedup* médio obtido, a `Classificação1` (`MGS` de 2,143) se mostrou mais eficiente, sendo 26,66% melhor do que `Classificação2` (`MGS` de 1,876) e 1,85% melhor do que `SemClassificação` (`MGS` de 2,125). Isso indica que para se obter desempenho é melhor que haja uma classificação que considere boas sequências já estabelecidas, que possam alimentar uma população inicial (`Classificação1` e `SemClassificação`) do que uma classificação com uma população inicial aleatória (`Classificação2`).

`NPS` Enquanto `Classificação1` conseguiu cobertura de 96,77% e `SemClassificação` 98,39% dos programas avaliados, `Classificação2` alcançou cobertura de somente 62,90%. Assim, embora `Classificação1` possua maior desempenho médio, `SemClassificação` alcançou desempenho para um número maior de programas avaliados.

`TR` Os tempos médios de resposta foram 01h:10m:46s, 01h:32m:40s e 01h:23m:29s para `Classificação1`, `Classificação2` e `SemClassificação`, respectivamente. Pode-se ressaltar então que `Classificação1` responde, em média, 23,63% mais rápido do que `Classificação2` e 15,21% mais rápido do que `SemClassificação`.

Os resultados apresentados mostram que a classificação proposta é eficiente para se alcançar desempenho para os programas avaliados, e, embora tenha obtido cobertura menor do que outra estrutura de classificação, alcançou desempenho médio maior com um tempo de resposta menor.

3.4. Comparação com Algoritmo Genético

É importante comparar o desempenho obtido pelo `AE` proposto com outra técnica. Para alcançar tal objetivo foi implementado um algoritmo genético, o qual é baseado nas estratégias dos trabalhos de Purini e Jain [2013] e Martins et al. [2016].

A estrutura do algoritmo genético é como segue. A população inicial é aleatória, cada indivíduo consiste em uma sequência de transformações, e a população é evoluída a cada geração. Dois indivíduos são escolhidos a cada iteração, por uma estratégia de torneio, para gerar novos indivíduos por meio de um operador de *crossover*. Além disso, uma mutação pode ocorrer. O *crossover* possui probabilidade de 60% de ocorrência, enquanto a mutação possui probabilidade de 40%. Os indivíduos iniciam com tamanhos de 1 até —Espaço de Transformação—, e assim

os operadores podem ser aplicados a indivíduos de diferentes tamanhos, sendo que o indivíduo de melhor desempenho sempre permanece para a geração seguinte.

A mutação pode modificar um indivíduo (1) inserindo uma nova transformação arbitrariamente; (2) removendo uma transformação da sequência em um ponto aleatório; (3) trocando duas transformações em pontos arbitrário; ou (4) alterando uma transformação da sequência por outra qualquer. Apenas uma dessas modificações é realizada, a qual é escolhida de forma aleatória.

O critério de parada é a estagnação da solução por três gerações ou a aptidão menor do que 0,01 do melhor indivíduo.

O AE proposto foi comparado com duas variações do algoritmo genético:

- AG10: evolui durante 10 gerações, com uma população de 20 indivíduos.
- AG100: evolui durante 100 gerações, com uma população de 50 indivíduos.

A Figura 5 apresenta os desempenhos alcançados por GA10 e GA100 comparados com o AE com a classificação proposta (Classificação1), população=10 e probabilidade de mutação=5%.

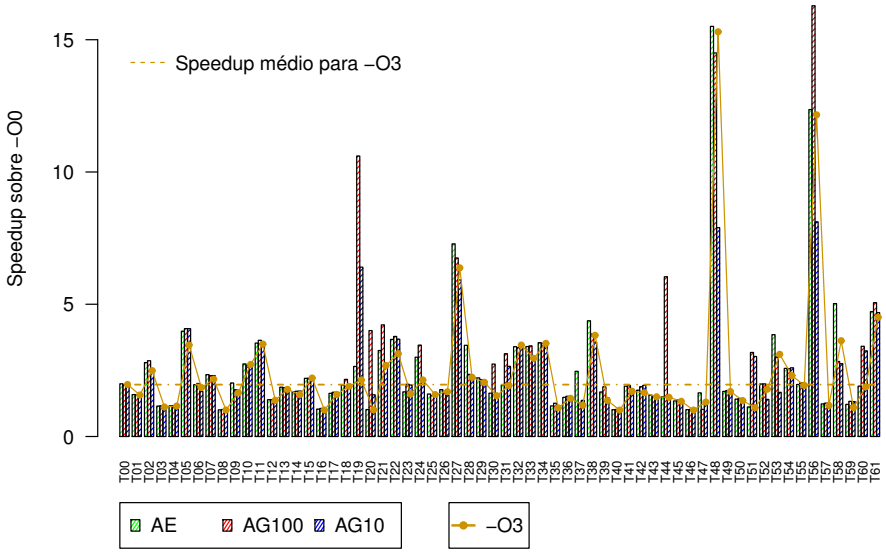


Figura 5: Desempenho alcançado pelo AE proposto e pelas técnicas AG10 e AG100.

MGS O desempenho médio obtido pelas estratégias foi de 2,143 pelo AE proposto e de 2,342 e 2,051 por AG100 e AG10, respectivamente. Assim, tem-se que o AE proposto foi capaz de superar AG10 em 9,19%, e foi superado por AG100 em 19,84%.

NPS Apesar de alcançar menor desempenho do que AG100, a estratégia proposta foi capaz de alcançar 96,77% de cobertura, enquanto AG100 obteve 82,26% e AG10 66,13%. Isso significa que AG100 consegue alto desempenho para alguns programas, enquanto a técnica proposta alcança desempenho para um número maior de programas.

TR O tempo médio decorrido de cada técnica foi de 1h:10m:40s para o AE proposto, 1h:17m:48s para AG10 e 32h:25m:15s para AG100. Dessa forma, a técnica proposta executou 9,18% menos tempo do que AG10 e 96,37% menos tempo do que AG100.

Os resultados alcançados evidenciam que o AE proposto é capaz de alcançar cobertura para um número maior de programas do que as outras técnicas comparadas, apesar de executar em

menos tempo. Com tempo de resposta 10% menor, é possível obter resultados melhores do que um algoritmo genético. Por outro lado, um algoritmo genético mais agressivo tende a obter um melhor desempenho, no tocante ao *speedup* médio alcançado. Contudo, isso ocorre ao custo de um alto tempo de resposta possivelmente uma menor cobertura.

4. Trabalhos Relacionados

Os primeiros trabalhos propostos para encontrar sequências de transformações se baseavam na compilação iterativa. Esta estratégia explora o espaço de busca seletivamente, avaliando N pontos do espaço. Atualmente, existem sistemas de compilação iterativa com busca parcial, aleatória e heurística.

Sistemas de compilação iterativa com busca parcial tentam explorar uma fração dos possíveis resultados [Kulkarni et al., 2009; Foleiss et al., 2011], enquanto os de busca aleatória realizam a busca por uma solução empregando técnicas estatísticas e/ou aleatórias, tentando reduzir o número de sequências avaliadas [Haneda et al., 2005; Shun e Fursin, 2005]. Em sistemas de busca heurística se tenta realizar tal redução por uma exploração intuitiva com algoritmos heurísticos [Kulkarni et al., 2005; Che e Wang, 2006].

O trabalho de Purini e Jain [2013] merece destaque no contexto da compilação iterativa. Apesar de ser classificado como compilação iterativa, esse trabalho reduz o tempo de resposta do sistema utilizando sequências de transformações capazes de conseguir desempenho para vários programas. O processo se dá com técnicas de busca híbridas, e ao final as 10 melhores sequências encontradas são extraídas segundo um critério de desempenho para os programas treino. Assim, essa estratégia reduziu o espaço de busca para essas 10 sequências, as quais são avaliadas para cada novo programa, retornando a que gerou o código de melhor desempenho.

A compilação iterativa é capaz de gerar boas sequências de transformações, porém ao custo de um alto tempo de resposta. Com o objetivo de reduzir tal custo, estratégias de aprendizagem de máquina foram propostas [de Lima et al., 2013; Tartara e Reghizzi, 2013; Queiroz Junior e da Silva, 2015]. Tais estratégias utilizam duas fases: uma de treino e outra de teste. A fase de treino tem por objetivo criar uma base de dados contendo informações que ajudarão na solução de futuros problemas. A fase de teste tem por objetivo inferir da base de dados uma possível solução para um novo problema. Estratégias baseadas em aprendizagem de máquina possuem dois diferenciais. Primeiro, elas são capazes de reduzir o tempo de resposta do sistema. Segundo, elas tomam decisões baseadas no conhecimento acumulado pelo sistema. Contudo, tais estratégias ainda necessitam de uma fase de treinamento a qual possui um tempo computacional considerável por serem estratégias de compilação iterativa.

de Lima et al. [2013] propuseram o uso de uma estratégia de aprendizagem de máquina para encontrar sequências de transformações. De fato, eles propuseram uma estratégia de raciocínio baseado em casos. Essa estratégia cria diversas sequências, utilizando um algoritmo aleatório, em uma fase de treino. Depois, em uma fase de teste, infere uma boa sequência para o programa teste baseada na similaridade entre o programa de entrada e cada programa da base.

O trabalho de Queiroz Junior e da Silva [2015] avaliou diferentes configurações de uma estratégia de aprendizagem de máquina, que visava encontrar sequências de transformações para programas de entrada. O objetivo desse trabalho foi avaliar o desempenho de uma estratégia de raciocínio baseado em casos utilizando diferentes base de dados, coeficientes e caracterizações de programas. Nesse trabalho, a base fase de treino é um algoritmo genético, o qual é classificado como compilação iterativa.

Tartara e Reghizzi [2013] propuseram uma estratégia híbrida de longo prazo, na qual o objetivo é eliminar a fase de treino. Nessa estratégia, o compilador é capaz de aprender, durante cada compilação, como gerar o melhor código alvo. De fato, os autores propuseram um algoritmo evolucionário que cria diversas heurísticas com base em características estáticas do programa teste.

Conforme exposto, estratégias de compilação iterativa demandam alto tempo de resposta, o que é uma desvantagem em relação a estratégias de aprendizagem de máquina. Por outro lado,

estratégias de aprendizagem máquina possuem um tempo de resposta menor, contudo ao custo de uma fase de treinamento que na prática é uma estratégia de compilação iterativa. Neste contexto, o AE proposto neste artigo possui duas vantagens. Primeiro, é uma estratégia de compilação iterativa cujo tempo de resposta é menor do que outras estratégias. Segundo, é capaz de reduzir o custo da fase de treinamento necessária a estratégias de aprendizagem de máquina.

5. Conclusões e Trabalhos Futuros

As estratégias de compilação iterativa, embora possuam alto tempo de resposta, ainda são utilizadas para potencializar o desempenho que pode ser obtido por estratégias de aprendizagem de máquina.

A estratégia apresentada neste artigo se mostrou eficiente para encontrar boas sequências de transformações para distintos programas. Tal estratégia é um Algoritmo Evolucionário, no qual a ordenação de classes de transformações deve ser mantida nos cromossomos. Assim, se pode classificar as transformações de compiladores e manter a ordem de aplicação, consequentemente maximizando o ganho de desempenho. Além disso, a classificação de transformações proposta mostrou-se ser uma excelente estratégia para potencializar o ganho de desempenho.

O Algoritmo Evolucionário proposto mostrou-se atraente para criar uma base de boas sequências para ser utilizada em uma estratégia de aprendizagem de máquina, na qual se escolhe boas sequências para um código de entrada com base em suas características. Essa estratégia conseguiu desempenhos até 9.73% superiores a outra estratégia da literatura em um tempo 23.22% menor, e além disso superou o nível de transformação {O3 da LLVM 4.0 para 75.85% dos casos avaliados.

Referências

Aho, A. V., Lam, M. S., Sethi, R., e Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Prentice Hall, Boston, MA, USA, 2 edition.

Bessedik, M., Toufik, B., e Drias, H. (2011). How can bees colour graphs. *Int. J. Bio-Inspired Comput.*, 3(1):67–76. ISSN 1758-0366. URL http://dx.doi.org/10.1504/IJBIC.2011.038705.

Che, Y. e Wang, Z. (2006). A Lightweight Iterative Compilation Approach for Optimization Parameter Selection. In *First International Multi-Symposiums on Computer and Computational Sciences*, volume 1, p. 318–325, Washington, DC, USA. IEEE Computer Society.

Cheraitia, M. e Haddadi, S. (2016). Simulated annealing for the uncapacitated exam scheduling problem. *Int. J. Metaheuristics*, 5(2):156–170. ISSN 1755-2176. URL https://doi.org/10.1504/IJMHEUR.2016.080266.

de Lima, E. D., de Souza Xavier, T. C., da Silva, A. F., e Ruiz, L. B. (2013). Compiling for performance and power efficiency. In *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, p. 142–149.

Foleiss, J. H., da Silva, A. F., e Ruiz, L. B. (2011). An Experimental Evaluation of Compiler Optimizations on Code Size. In *Proceedings of the Brazilian Symposium on Programming Languages*, p. 1–15, São Paulo, São Paulo, Brazil. EACH USP.

Gouy, I. (2017). The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/. Último acesso em 09/jan/2017.

Gu, F., lin Liu, H., e Li, X. (2016). A fast evolutionary algorithm with searching preference. *International Journal of Computational Science and Engineering*, 12(1):29–37.

Haneda, M., Knijnenburg, P. M. W., e Wijshoff, H. A. G. (2005). Generating New General Compiler Optimization Settings. In *Proceedings of the Annual International Conference on Supercomputing*, p. 161–168, New York, NY, USA. ACM.

Jia, L., Wang, Y., e Fan, L. (2016). An improved uniform design-based genetic algorithm for multi-objective bilevel convex programming. *Int. J. Comput. Sci. Eng.*, 12(1):38–46. ISSN 1742-7185. URL http://dx.doi.org/10.1504/IJCSE.2016.074562.

Kulkarni, P. A., Hines, S. R., Whalley, D. B., Hiser, J. D., Davidson, J. W., e Jones, D. L. (2005). Fast and Efficient Searches for Effective Optimization-Phase Sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198. ISSN 1544-3566.

Kulkarni, P. A., Whalley, D. B., Tyson, G. S., e Davidson, J. W. (2009). Practical Exhaustive Optimization Phase Order Exploration and Evaluation. *ACM Transactions on Architecture and Code Optimization*, 6(1):1–36. ISSN 1544-3566.

Lattner, C. (2017). The LLVM Compiler Infrastructure. http://llvm.org. Último acesso em 09/jan/2017.

Luke, S. (2013). *Essentials of Metaheuristics*. Lulu, second edition. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Martins, L. G. A., Nobre, R., Cardoso, J. a. M. P., Delbem, A. C. B., e Marques, E. (2016). Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization*, 13(1):8:1–8:28. ISSN 1544-3566.

Meena, M. J., Chandran, K. R., Karthik, A., e Samuel, A. V. (2011). A parallel aco algorithm to select terms to categorise longer documents. *Int. J. Comput. Sci. Eng.*, 6(4):238–248. ISSN 1742-7185. URL http://dx.doi.org/10.1504/IJCSE.2011.043923.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-320-4.

Purini, S. e Jain, L. (2013). Finding Good Optimization Sequences Covering Program Space. *ACM Transactions on Architecture and Code Optimization*, 9(4):56:1–56:23. ISSN 1544-3566.

Queiroz Junior, N. L. e da Silva, A. F. (2015). Finding Good Compiler Optimization Sets - A Case-Based Reasoning Approach. In *International Conference on Enterprise Information Systems*, p. 504–515. ISBN 978-989-758-096-3.

Rajakumar, B. R. (2013). Static and adaptive mutation techniques for genetic algorithm: A systematic comparative analysis. *Int. J. Comput. Sci. Eng.*, 8(2):180–193. ISSN 1742-7185. URL http://dx.doi.org/10.1504/IJCSE.2013.053087.

Russell, S. J., Norvig, P., Candy, J. F., Malik, J. M., e Edwards, D. D. (1996). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-103805-2.

Shun, L. e Fursin, G. (2005). A Heuristic Search Algorithm Based on Unified Transformation Framework. In *Proceedings of the International Conference Workshops on Parallel Processing*, p. 137–144, Oslo, Norway. IEEE Computer Society.

Tartara, M. e Reghizzi, S. C. (2013). Continuous Learning of Compiler Heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):46:1–46:25. ISSN 1544-3566.

Wang, Q., Liu, H.-L., cheng Li, J., e Li, Y. (2017). A resource allocation evolutionary algorithm for ofdm system. *International Journal of Computational Science and Engineering*, 14(1):55–633.