IDENTIFICAÇÃO DE PROGRAMAS SEMELHANTES

Ana Julia Faccioli Sordi (PIBIC/CNPq/FA/UEM), Anderson Faustino da Silva (Orientador). E-mail: ra115630@uem.br.

Universidade Estadual de Maringá, Centro de Tecnologia, Maringá, PR.

Ciência da Computação, Sistemas de Computação

Palavras-chave: sistemas inteligentes; representação; similaridade.

RESUMO

Sistemas inteligentes de geração de código empregam estratégias de aprendizagem de máquina para determinar quais transformações devem ser aplicadas ao programa a ser compilado. Tais estratégias utilizam experiências passadas para determinar um novo plano de compilação, para um programa não visto. Neste contexto, é essencial a identificação de programas semelhantes. Este resumo apresenta uma avaliação de diferentes representações de programas e como elas diferem na identificação de programas semelhantes. Os resultados indicam que uma representação baseada em histogramas é eficiente na identificação de tais programas.

INTRODUÇÃO

Um sistema inteligente de geração de código (ZANELLA, SILVA e PEREIRA, 2020) tenta descobrir, para um determinado programa: (1) quais transformações de código devem ser utilizadas, (2) a ordem dessas transformações e (3) suas parametrizações; ou seja, um plano de compilação. Sistemas deste porte criam um modelo de previsão em uma fase de treino, baseado no comportamento de programas treino. Posteriormente, o modelo de previsão, na fase de teste, encontra o melhor conjunto de transformações para um programa não visto pelo sistema.

Tais sistemas necessitam identificar programas semelhantes; pois estes partem do princípio que programas semelhantes podem utilizar o mesmo conjunto de transformações. Para alcançar esta identificação é realizado uma comparação entre a representação de dois programas. Desta forma, a questão que surge é: qual representação fornece uma melhor acurácia na determinação de programas semelhantes. Este é o objetivo deste resumo, avaliar diferentes representações e assim indicar aquela que fornece a maior precisão na identificação de programas semelhantes.

MATERIAIS E MÉTODOS

As representações de programas são divididas em duas classes: (1) representações gráficas (ZANELLA, SILVA e PEREIRA, 2020) e (2) representação vetoriais (LONG e O'BOYLE, 2004; TARTARA e REGHIZZI, 2013; QUEIROZ JUNIOR e SILVA, 2013; LIMA, SOUZA XAVIER, SILVA e RUIZ, 2013; ZANELLA, SILVA e PEREIRA, 2020). A primeira representa o programa com componentes interconectados entre si. Por outro lado, a segunda utiliza um vetor numérico ou de caracteres.

As representações gráficas são árvore e grafo. Uma árvore representa o programa por meio da sintaxe de suas estruturas e é conhecida como árvore sintática abstrata (ASA). Por sua vez, um grafo representa o programa por meio das relações entre suas estruturas. Existem basicamente duas representações baseadas em grafo: grafo de fluxo de controle (GFC) e grafo de fluxo de dados e controle (GFDC).

As representações vetoriais representam o programa como um DNA ou um vetor numérico. O primeiro transforma cada estrutura do programa em um gene. O segundo coleta a quantidade de estruturas contidas no programa e cria um vetor.

Tradicionalmente, o sistema de geração de código elege apenas uma representação e a implementa. Porém, o sistema YaCoS (ZANELLA, SILVA e PEREIRA, 2020) possui o diferencial de disponibilizar diversas representações.

YaCoS é uma infraestrutura que permite explorar planos de compilação. Tal infraestrutura consiste de quatro partes: (1) um extrator de características (representação) do programa; (2) um módulo que estabelece uma distância (similaridade) entre programas; (3) um módulo de busca por planos de compilação; e (4) um conjunto de programas que podem ser utilizados como treino e/ou teste.

Extrator de características. Este módulo fornece a necessária funcionalidade para extrair uma representação para o programa. Esta representação pode ser o histograma das instruções que compõem programa, ou um grafo. Para este último, YaCoS fornece as seguintes representações: um GFC, além de três variações GFDC (GFDC tradicional, GFDC contendo arestas que representam os blocos básicos e PrograML).

Identificador de similaridade. A distância entre dois programas é a medida de similaridade entre eles. Programas com a menor similaridade devem ser tratados como semelhantes. YaCoS fornece as seguintes métricas: (1) MCoeff, a qual mede a distância entre dois programas analisando o efeito de diferentes planos de compilação sobre eles; (2) alinhamento de DNA; (3) distância de coseno; (4) distância euclideana; (5) distância de Manhattan; (6) distância de edição.

Motor de planos de compilação. Este é um mecanismo de busca cujo objetivo é associar um bom plano de compilação a um determinado programa. YaCoS fornece os seguintes mecanismos: (1) algoritmo genético; (2) otimização por enxame de partículas; (3) eliminação em lote; (4) ordenação de fase; (5) aprendizagem ativa; (6) raciocínio baseado em caso; e (7) algoritmo aleatório.

Benchmarks. YaCoS também fornece uma coleção de programas, os quais foram coletados de diversas fontes; tais como: MiBench, SPEC CPU, LLVM Test Suite, entre outras.

RESULTADOS E DISCUSSÃO

Para identificar a representação que fornece a maior precisão na identificação de programas semelhantes são seguidos os passos a seguir.

1. Seleção de programas de treino. Este grupo é composto por 100 programas do LLVM Test Suite.
2. Seleção de programas de teste. Este grupo é composto por 10 programas do SPEC CPU.
3. Seleção de planos de compilação. Este grupo é composto por 100 planos de compilação utilizando o algoritmo de busca aleatória.
4. Identificação de pares de programas treino semelhantes. Para identificar tais programas é utilizada a métrica MCoeff, que indica quais programas possuem comportamento semelhante quando o mesmo plano de compilação é utilizado para ambos, com os 100 planos de compilação gerados no passo anterior. Tal métrica será utilizada como oráculo, ou seja, programas semelhantes.
5. Extração de representação para os programas treino e teste. Todas representações disponíveis no YaCoS são utilizadas.
6. Identificação do programa treino mais semelhante a um determinado programa teste. Para alcançar tal objetivo cada programa teste é comparado com todos programas treino. Tal comparação é feita utilizando um par (R, D), onde R indica a representação e D indica a métrica de distância aplicada a esta representação.
7. Por fim, é anotado quantos acertos cada representação obteve. Um acerto indica que o par (R, D) encontrou, na base de treino, o mesmo programa encontrada pelo oráculo.

A Tabela 1 apresenta os resultados obtidos após a execução do Passo 7. Como demonstrado pelos resultados obtidos, a representação que obteve o melhor desempenho é o histograma das instruções do programa. Tal fato indica que é possível obter resultados utilizando uma representação simples. Outro ponto a ser observado, é o fato das variações do GFDC não apresentarem resultados diferentes. Isto indica que aumentar o grafo com novas informações não ocasionam necessariamente uma melhora nos resultados.

Tabela 1: Acurácia Obtida por Cada Representação

Representação	***Indicador de Similaridade***	***Acurácia***
Histograma	*Distância Euclideana*	80 %
DNA	*Alinhamento*	45 %
GFC	*Distância de edição*	60 %
GFDC tradicional	*Distância de edição*	70 %
GFDC + blocos básicos	*Distância de edição*	70 %
Programl	*Distância de edição*	70 %

CONCLUSÕES

Este resumo apresentou uma avaliação de diferentes representações de programas, as quais são utilizadas por diferentes sistemas inteligentes de geração de código, com o objetivo de identificar programas semelhantes. Os resultados indicam que histograma é uma boa representação para alcançar tal objetivo.

AGRADECIMENTOS

Agradeço a CNPQ pelo financiamento a este projeto.

REFERÊNCIAS

LIMA, E.D., SOUZA XAVIER, T.C., SILVA, A.F., RUIZ, L.B. **Compiling for performance and power efficiency**. *In*: Anais de International Workshop on Power and Timing Modeling, Optimization and Simulation. 2013. pp.142-149.

QUEIROZ JUNIOR, N.L., SILVA, A.F. **Finding good compiler optimization sets — A case-based reasoning approach**. *In*: Anais de International Conference on Enterprise Information Systems. 2015, pp.504-515.

LONG, S., O'BOYLE, M. **Adaptive Java optimisation using instance-based learning**. *In*: Anais de International Conference on Supercomputing, 2004. pp. 237-246.

TARTARA M., REGHIZZI, S.C. **Continuous learning of compiler heuristics**. ACM Transactions on Architecture and Code Optimization, 2013, 9(4): 46:1-46:25.

ZANELLA, A.F., SILVA, A.F., PEREIRA, F.Q. **YaCoS: a Complete Infrastructure to the Design and Exploration of Code Optimization Sequences**. *In*: Anais do Simpósio Brasileiro de Linguagem de Programação, 2020. pp. 56-63.