Mestrado em Ciência da Computação - Universidade Federal de Goiás

Genetic Improvement of Software: A Comprehensive Survey

Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward

Apresentado por Sávio Sampaio
saviosampaio@inf.ufg.br

2018

I4Soft
Intelligence for Software

UFG

INSTITUTO DE INFORMÁTICA
UFG

Agenda

- Introdução
- História do *Genetic Improvement*
- Metodologia utilizada no *Survey*
- Trabalhos existentes sobre *Genetic Improvement*
- Trabalhos relacionados
- Conclusões
- Referências

O **melhoramento genético, ou Genetic Improvement (GI),** usa pesquisa automatizada para encontrar versões aprimoradas de um software existente.

- Abstract

Genetic Improvement of Software: A Comprehensive Survey

Introdução

Introdução

- IEEE **TRANSACTIONS ON EVOLUTIONARY COMPUTATION**, VOL. 22, NO. 3, JUNE 2018
- Foco nos principais artigos **entre 1995 e 2015**
- 96% dos trabalhos usam **algoritmos evolucionários**, **programação genética em particular**
- 3.132 títulos distintos, **66 artigos básicos** sobre GI
- Aumento significativo neste campo, **desde 2012**
- **59,70% publicados** nos últimos três anos (2013–2015)

Introdução

- Trabalhos recentes receberam **prêmios notáveis**
- Aceitação e **sucesso dentro das comunidades** de engenharia de software e computação evolucionária
- **Atenção da mídia** de transmissão, bem como de revistas, sites e blogs populares de desenvolvedores
- **Influência e alcance** além da comunidade de pesquisa, comunidade de desenvolvedores e o público em geral

Introdução

- O GI resultou em **melhorias drásticas de desempenho** para um conjunto diversificado de propriedades, como:
  - **tempo de execução**
  - **consumo de energia**
  - **consumo de memória**
- Bem como:
  - resultados para correção
  - ampliação da funcionalidade

Introdução

- Exemplos de trabalhos **na fronteira** entre GI e outras áreas:
  - transformação de programas
  - computação aproximada
  - reparo de software

Introdução

- A **computação evolutiva** é de longe a técnica de busca computacional mais difundida usada na literatura
- GI é um **campo na interseção intelectual** entre áreas:
  - Computação evolucionária
  - Engenharia de software
  - Otimização
  - Análise e manipulação do código fonte

Introdução

- No artigo também revisaram a **relação entre o GI e**:
  - síntese de programas
  - transformação de programa
  - ajuste de parâmetros
  - computação aproximada
  - fatiamento (slicing)
  - avaliação parcial e outros

História do *Genetic Improvement*

História do *Genetic Improvement*

- O IG baseia-se e desenvolve pesquisas em vários tópicos, que **fazem parte da sua história**, como:
  - transformação de programas
  - síntese de programas
  - programação genética
  - teste de software
  - engenharia de software baseada em pesquisa (SBSE)

Antes dos computadores eletrônicos

Antes dos computadores eletrônicos

- A **primeira menção à otimização de software** deve-se a **Ada Lovelace**, em 1842 sobre o “mecanismo analítico”
- Quase 200 anos depois, fica bem claro que Ada estava ciente da **necessidade de otimizar programas**

> Em quase todos os cálculos **é possível uma grande variedade de arranjos para a sucessão dos processos**, e várias considerações devem influenciar a seleção entre eles para os propósitos de um Mecanismo de Cálculo.
>
> Um objetivo essencial é **escolher aquele arranjo que tenderá a reduzir ao mínimo** o tempo necessário para completar o cálculo.

- Ada Lovelace (Note D)

"Sketch of the Analytical Engine invented by Charles Babbage, with notes by the translator."

Antes dos computadores eletrônicos

- Ada estava ciente de que os programas caíam em **classes de equivalência** e via a possibilidade de **otimizar a escolha de um programa** dentro de uma classe de equivalência

Transformação de Programas

Transformação de Programas

- 2 linhas de pesquisa que abordam a manipulação de programas e que cresceram nos anos 60 e 70:
  - transformação de programas
  - síntese de programas
- Exploram as **classes de equivalência** citadas por Ada
- Uma procurava aplicar **transformações com preservação de significado** para refinar um programa existente
- Outra procurava **construir um novo código** de programa.

Transformação de Programas

- ➔ Primeiros trabalhos sobre compiladores: **converter automaticamente** cálculos em formas canônicas minimizadas, **para eficiência de espaço ou tempo**
- ➔ Sheridan (1959) fez clara distinção entre **transformações gerais e específicas**
- ➔ Observou que as **transformações gerais** permitiram que uma expressão de programa arbitrária fosse "**embaralhada em uma ordem diferente sem perturbar o algoritmo**"

Transformação de Programas

- *The Arithmetic Translator Compiler of the IBM Fortran Automatic Coding System*, Peter B. Sheridan (1959)
- ***Optimization (General):***
  - Eliminação de parênteses redundantes
  - Eliminação de sub-expressões comuns, para evitar computação redundante
- ***Optimization (Special):***
  - Minimizar acesso à memória

Transformação de Programas

- ➔ Nos 60’s e 70’s, foco nos princípios que permitiam que uma representação sintática fosse transformada em outra, **preservando a correção semântica**
- ➔ Pesquisadores procuraram definir a **semântica das linguagens de programação**, fornecendo assim uma base matemática sólida para a teoria e prática do desenvolvimento de software

Transformação de Programas

- Nos 80’s, sistemas de **transformação de linguagem declarativa de propósito geral** começaram a aparecer
- Sistemas cada vez mais sofisticados foram desenvolvidos para a **transformação geral**, como o sistema CIP de Munique na década de 1980, sistemas de avaliação parcial, como Tempo, na década de 1990 e a linguagem e sistema de transformação TXL na década de 2000

Síntese de Programas

Síntese de Programas

- Procurou **construir um novo código** de programa, de tal forma que o programa resultante seria **correto por construção**.
- Uma das primeiras implementações (de 1961) foi o trabalho sobre o "compilador heurístico" de Simon.

Síntese de Programas

- Os primeiros sistemas de **síntese de programas** e de **transformação de programas** foram desenvolvidos e inspirados sobre os **compiladores da época**.
- A **síntese de programas** permaneceu um tópico de interesse e desenvolvimento contínuos, ao longo dos anos 1970, 1980 e 1990, até o trabalho recente sobre **síntese de macros de planilhas** por Gulwani et al.

Síntese de Programas

- O IG está intimamente relacionado à síntese e à transformação, **mas difere de cada um**:
  - O IG nem sempre é guiado pela motivação da **correção por construção**
  - Geralmente, o **teste de software é usado como um oráculo** para o comportamento correto do sistema.
- O IG baseia-se na **rica herança da programação genética**, que também é geralmente **guiada por testes de software**, **e não reivindica correção por construção**.

Programação Genética

Programação Genética

- ➔ O **primeiro registro da proposta para evoluir programas** é provavelmente de Turing (1950's)
- ➔ A ideia de programas em evolução estava presente **entre os estudantes de John Holland (criador do *GA*)**
- ➔ Seus alunos organizaram a **primeira conferência de Algoritmos Genéticos** em Pittsburgh (1985), onde Cramer publicou programas evoluídos em duas línguas especialmente concebidas para isso

Programação Genética

- Em 1988, **John Koza** (aluno de John Holland) patenteou sua invenção de **um GA para a evolução de programas**
- Trabalho publicado no ano seguinte, na ***International Joint Conference on Artificial Intelligence*** - IJCAI-89
- Koza seguiu isto com **205 publicações sobre "Genetic Programming" (GP)**, nome cunhado por David Goldberg, também aluno de doutorado de John Holland

Programação Genética

- Foi a **série de 4 livros de John Koza**, começando em 1992, com vídeos, que realmente estabeleceram a GP.
- Enorme expansão do número de publicações sobre Programação Genética, **superando 10.000 entradas**.
- Em 2010, Koza listou 77 resultados onde a Programação Genética era "***human competitive***".

Programação Genética

- Excluindo GI, a GP continua concentrada na **modelagem preditiva**, particularmente **mineração de dados** e **modelagem financeira**.
- Evolução de sensores suaves (indústria química), design e processamento de imagens, finanças.
- Também usada na bioinformática e indústria siderúrgica.
- GP também pode ser encontrada em projetos artísticos.

Programação Genética

- A **GP** compartilha com a **síntese do programa** seu objetivo de construir um programa do zero.
- A **GI** geralmente começa a partir de um programa existente (mais como a **transformação de programa**).
- A **GI** pode lidar com programas muito maiores do que a **síntese de programas** ou a **programação genética**.
- Como a **GP**, o **teste de software** é frequentemente usado para guiar o **GI** para a variante do software.

Teste e validação de software

Teste e validação de software

- O teste é importante para o GI, pois pode ser usado como um **guia para a fidelidade semântica**, e para **avaliar o grau em que a melhoria foi alcançada**.
- Na década de 1940, Turing delineou os papeis de "**testador de programas**" de "**programador**".
- 1960's: a automação é essencial para **gerenciar a escala do desafio de teste de software**; e aparecem os primeiros **sistemas de geração de entrada de teste automatizado**.
- Tais sistemas são melhorados ao longo dos anos 1970.

Teste e validação de software

- Recente **desenvolvimento significativo** em muitas áreas de testes, como:
  - **Execução simbólica dinâmica**
  - **Testes de software baseados em pesquisa**
  - **Testes de mutação**

Teste e validação de software

- ➔ Muitos pesquisadores poderiam ser “perdoados por acreditar” que **o teste nunca seria suficiente para garantir fidelidade à semântica** do programa original.
- ➔ Um “aforismo” (máxima ou sentença) de Dijkstra (1969), altamente citável, tornou-se um **"artigo de fé" em uma infeliz batalha entre testes e verificação** que só recentemente diminuiu.

> O **número de entradas diferentes**, ou seja, o **número de cálculos diferentes** para os quais as afirmações indicam, **é tão fantasticamente alto**, que a demonstração de correção por amostragem está completamente fora de questão.
>
> **O teste do programa pode ser usado para mostrar a presença de bugs**, mas nunca para mostrar sua ausência! Portanto, **a exatidão do programa deve ser provada por causa do texto do programa**.

- E. W. Dijkstra (1969)
"Structured Programming". [Online]. Available: Google EWD268.PDF

Teste e validação de software

- ➔ É verdade que **o teste nunca pode mostrar a ausência de todos os bugs**.
- ➔ Mas **é questionável se qualquer abordagem** de correção do programa pode ou poderia mostrar a essa ausência.
- ➔ Já há técnicas que podem provar a ausência de bugs **com relação a determinadas suposições**.
- ➔ Mas **os testes sempre terão um papel**, mesmo que apenas para verificar se tais suposições são razoáveis.

Teste e validação de software

- Em GI **não se pressupõe que apenas o teste seja usado**.
- Há outras **técnicas de verificação para complementar** as abordagens existentes, baseadas em testes.
- Mas houve um grande progresso usando apenas o teste:
  - tanto para avaliar **fidelidade à semântica** a ser retida
  - quanto para **medir o grau de melhoria alcançado**

Teste e validação de software

Como o GI poderia ser tão bem-sucedido, usando uma combinação de técnicas que pareceriam tão equivocadas do ponto de vista de ilustres pesquisadores?

- ➔ A resposta pode estar nos **recentes resultados empíricos**.
- ➔ Esses resultados empíricos **confundem algumas das suposições amplamente estabelecidas**, baseadas em inferências plausíveis da natureza teórica.

Teste e validação de software

Resultados empíricos recentes **desafiam os pressupostos**:

- ➔ O número de **programas possíveis** em uma determinada língua é "tão inconcebivelmente grande" que o GI certamente não poderia esperar encontrar soluções no **"material genético" do programa existente**.
- ➔ O espaço de entrada de teste também é "tão fantasticamente alto" que, com certeza, as entradas de amostragem **nunca poderiam ser suficientes para capturar verdades estáticas** sobre a computação.

Teste e validação de software

- ➔ O código que ocorre naturalmente **é surpreendentemente repetitivo** (Gabel e Su, 2010, “uniqueness of source code”).
- ➔ Um programador **teria que escrever mais de 6 linhas de código para criar um fragmento de código original**, não localizado em algum lugar no sourceforge.
- ➔ Muitas das intervenções exploradas pelo GI **consistiam em menos de 6 linhas de código** (remendos, correções e pequenas modificações).

Teste e validação de software

- **43% dos commits** para um grande repositório de projetos Java poderiam ser reconstituídos a partir do código existente (Barr et al., 2014)
- Ou seja, um número grande de mudanças feitas por humanos **já poderia ser fabricado por GI**, ou técnicas semelhantes **que reutilizem o código já existente** como "**mero material genético**" a ser manipulado.

Teste e validação de software

Esses 2 estudos forneceram evidências empíricas de que:

➔ Embora o espaço teórico dos programas seja extraordinariamente grande, **o espaço prático habitado pelo código desenvolvido pelo homem é muito mais restrito**, tornando-o potencialmente mais acessível ao GI do que se poderia supor de um ponto de vista puramente teórico.

Teste e validação de software

- Trabalho de 2001 demonstrou que **verdade estática** sobre computação de programa **pode ser inferida de uma amostra pequena** de pares de entrada-saída, **em um grande número de casos**.
- A observação de que **uma pequena quantidade de informação dinâmica pode gerar verdade estática** também foi encontrada em outros domínios de engenharia de software (2015).

Search-Based Software Engineering - SBSE

Search-Based Software Engineering - SBSE

- ➔ Os sistemas de testes automatizados iniciais (1962) formularam **a geração de dados de teste como um problema de pesquisa**.
- ➔ 1975 e 1976: primeira aplicação da **pesquisa automatizada** em problemas de engenharia de software.
- ➔ Em 2001, o termo "SBSE" foi cunhado por Harman e Jones, em um manifesto para a **aplicação de busca automatizada a problemas em engenharia de software** (Primeiro artigo a defender uma disciplina de SBSE).

> A tese subjacente ao presente artigo é que **as técnicas de otimização metaheurística baseadas em pesquisa são altamente aplicáveis à Engenharia de Software** e que sua investigação e aplicação à Engenharia de Software está muito atrasada. É hora de a Engenharia de Software alcançar as suas contrapartes mais maduras nos campos tradicionais de engenharia.

- M. Harman and B. F. Jones
"Search-based software engineering", 2001

Search-Based Software Engineering - SBSE

- Trabalhos “pré-SBSE”, que contribuíram para a SBSE:
  - **problemas no gerenciamento de projetos de software** (1994 e 2000)
  - **testes de software** (1992 e 1998)
  - **novas formas de GP** para problemas de engenharia de software (1998 e 1999).
- Idéias iniciais **associadas ao GI** em trabalho de 1998.

Search-Based Software Engineering - SBSE

- **Melhorar o software com vários objetivos em mente é muito poderoso**, e computação evolutiva é boa para encontrar compensações entre objetivos conflitantes.
- Lakhotia (2007, GECCO) foram os primeiros a usar a **otimização multiobjetivo** para a geração de dados de teste.
- Kalboussi (2013, SSBSE) consideraram **sete objetivos ao gerar casos de teste** (com NSGA-III).
- Mkaouer (2015) e Ramírez (2016) consideraram o **conflito entre objetivos ao reorganizar código-fonte** em Java.

Search-Based Software Engineering - SBSE

- Desde 2001 houve um **aumento na atividade da SBSE**.
- Pesquisas em muitas subáreas da SBSE, incluindo:
  - requisitos, modelagem preditiva, gerenciamento de projetos de software, projeto, testes, linhas de produtos de software, reparo, entre outras.
- Mas não houve trabalhos na área de GI, que procurem **aplicar a abordagem SBSE ao próprio código-fonte**.

Genetic Improvement e o futuro

Genetic Improvement e o futuro

- GI remonta à **síntese de programas**, **transformação de programas** e **programação genética** (anos 90), e apenas recentemente o GI surge como uma área de pesquisa.
- **O termo "GI"** surgiu a partir de estudos anteriores (2011 e 2012): "melhoramento evolutivo" e "GI de programas".
- GI **expandiu com trabalhos de reparo automatizado**: A. Arcuri e X. Yao, "A novel co-evolutionary approach to **automatic software bug fixing**", 2008, e Claire Le Goues et al. "Current challenges in **automatic software repair**", 2013.

Genetic Improvement e o futuro

➔ Por que o GI emergiu apenas recentemente **como uma área de pesquisa separada**?

Genetic Improvement e o futuro

- ➔ Por que o GI emergiu apenas recentemente **como uma área de pesquisa separada**?
- ➔ Só recentemente os **“ingredientes” do GI** se juntaram em áreas suficientemente maduras de atividade:
  - ◆ Poderosas **técnicas de geração de dados de teste**
  - ◆ Abundância de **código-fonte publicamente disponível**
  - ◆ A importância de **propriedades não-funcionais**

Genetic Improvement e o futuro

- ➔ Por que o GI emergiu apenas recentemente **como uma área de pesquisa separada**?
  - Durante a maior parte dos anos anteriores, se preocuparam com a correção do programa.
  - Muito trabalho tem sido dedicado a transformações de programas que preservam a semântica.
  - Autores influentes consideraram desafio inatingível.
  - Em 1988, Dijkstra afirmou que a programação automatizada era uma contradição.

> A ciência da computação está, e sempre estará, preocupada com **a interação entre a manipulação de símbolos mecanizada e humana**, geralmente referida como "computação" e "programação", respectivamente.
>
> Um benefício imediato desse *insight* é que **ele revela "programação automática" como uma contradição** ...

- E. W. Dijkstra

"On the Cruelty of Really Teaching Computing Science", 1988

Genetic Improvement e o futuro

- A tendência atual em GI não tem buscado apenas a **programação automática**, mas tem produzindo para a máquina uma grande parte do território **anteriormente ocupado por seres humanos**.
- Com a **abundância de softwares para "reuso genético"**, a síntese do zero parece cada vez mais sub-ótima.
- O poder da **geração de entrada de teste automatizada**, o uso do **programa original como um oráculo** e a importância das **propriedades não-funcionais** tornam o GI oportuno para a melhoria automatizada do software.

Genetic Improvement e o futuro

- ➔ No futuro poderemos ver **trabalhos híbridos** que se baseiam em **transformação, síntese, GP** e outras técnicas de análise e manipulação de código fonte.
- ➔ Trabalhos recentes sobre o **reparo automatizado** de programas (uma forma de GI) já usam uma combinação de técnicas, inspiradas em síntese de código e em GP.

Metodologia utilizada no Survey

Metodologia utilizada no Survey

- GI utiliza a pesquisa automatizada para navegar no espaço de pesquisa “3D”:
  - Quantidade de melhorias (sobre o código original)
  - Uso de software existente
  - Preservação da funcionalidade do código original
- Vários trabalhos que se encontram nos extremos desse espectro situam-se na fronteira entre o IG e outras áreas.

Metodologia utilizada no Survey

- ➔ Mrazek et al. usaram **programação genética cartesiana** para otimizar a eficiência e o consumo de energia (“Evolutionary approximation of software for embedded systems: Median function”, 2015, GECCO).
- ➔ Avaliaram aproximações das funções medianas de 9 entradas e de 25 entradas **por meio de testes, como é típico no trabalho GI**.
- ➔ **Evoluíram as funções a partir do zero**, gerando a população inicial de forma aleatória (típico na GP).

Metodologia utilizada no Survey

- ➔ Kocsis et al. (2014, SSBSE) e Burles et al. (2015, SSBSE) propuseram a utilização de **transformações que preservam a semântica**, a fim de manter a funcionalidade completa do código original (transformação de programa).
- ➔ Orlov e Sipper (2009, GECCO) aprimoraram o software existente aplicando um **operador de crossover de preservação de semântica**.

Metodologia utilizada no Survey

- ➔ Walsh e Ryan (1996, GECCO) usaram **árvores GP padrão**, e desenvolveram **sequências de transformações que preservam a semântica**, para paralelização automática.
- ➔ Williams (1998) usou **6 algoritmos evolutivos** para paralelizar o código existente. Usou o conhecimento do **fluxo de dados** do programa e da **análise de dependência** para evitar transformações que quebram a funcionalidade.

Metodologia utilizada no Survey

- ➔ Mesmo que as alterações de código sejam restritas a transformações com preservação de semântica, **o espaço de pesquisa de variantes ainda é enorme**.
- ➔ Por isso, **Metaheurísticas têm sido aplicadas** para encontrar soluções ótimas ou quase ótimas.

Metodologia utilizada no Survey

- Outras abordagens envolvem **busca determinística**, como:
  - Exploração do espaço com algoritmos de busca **exaustivos e gananciosos** (gulosos).
  - Uso de **todos os seus operadores de mutação** até que **todos os casos de teste** fossem aprovados ou que um tempo limite fosse atingido.
  - Uso de **busca exaustiva sobre pequenas mudanças** no nível de código para melhorar o consumo de energia.

Metodologia utilizada no Survey

➔ Identificaram quatro critérios segundo os quais consideraram uma publicação como um **artigo GI básico**:

- **Busca Metaheurística é usada**;
- Variantes de software **que não preservam a semântica podem ser produzidas** durante a pesquisa;
- **O software existente é reutilizado como entrada** para a estrutura de melhoria fornecida;
- O software modificado **é aprimorado em relação ao software existente** com relação a critérios fornecidos.

Metodologia utilizada no Survey

Fontes de pesquisa:

- ➔ *Collection of Computer Science Bibliographies**; e
- ➔ Bibliotecas on-line de quatro grandes editoras em engenharia de software:
  - ◆ ACM (Biblioteca Digital ACM)
  - ◆ IEEE (IEEE Xplore)
  - ◆ Springer (SpringerLink)
  - ◆ Elsevier (ScienceDirect) (*)

Metodologia utilizada no Survey

- Usaram as seguintes frases exatas como **palavras-chave**:
  - “genetic improvement”
  - “software improvement”
  - “evolutionary improvement”
- Consideraram **artigos de conferências e workshops**, **artigos de periódicos e teses de doutorado** que foram publicadas até o final de 2015.
- Chamaram esta etapa de **pesquisa primária**.

Resultados da pesquisa primária

Keyword:	**“genetic improvement”**		
Source	**Filters**	**Papers Found**	**Papers on GI**
ACM	Title OR Abstract	28	12
IEEE	Metadata	12	4
Springer	Full Text, Computer Science, language: English	69	11
Elsevier	Title OR Abstract OR Keywords, Computer Science	5	0
Collection	Default	165	41

Resultados da pesquisa primária

Keyword:	**“evolutionary improvement”**		
Source	**Filters**	**Papers Found**	**Papers on GI**
ACM	Title OR Abstract	5	1
IEEE	Metadata	21	1
Springer	Full Text, Computer Science, language: English	133	4
Elsevier	Title OR Abstract OR Keywords, Computer Science	3	0
Collection	Default	32	1

Resultados da pesquisa primária

Keyword:	**“software improvement”**		
Source	**Filters**	**Papers Found**	**Papers on GI**
ACM	Title OR Abstract	45	0
IEEE	Metadata	83	0
Springer	Full Text, Computer Science, language: English	421	5
Elsevier	Title OR Abstract OR Keywords, Computer Science	9	0
Collection	Default	100	2

Resultados da pesquisa primária

	Papers Found	Papers on GI
Total:	1131	82
Distinct papers on GI found:		54
Distinct core papers on GI found:		**40**

Metodologia utilizada no Survey

- ➔ **Examinaram bibliografias dos selecionados**, para incluir outras publicações relevantes com base nos critérios.
- ➔ Dos 40 artigos da pesquisa primária, **mais de 1000 artigos citados** em suas bibliografias.
- ➔ Foi criado um **procedimento automatizado para remover trabalhos duplicados** (já considerados anteriormente).
- ➔ Com isso, **reduziram de 1000 para 862 novos títulos**, indicados nas bibliografias das 40 publicações iniciais.

Metodologia utilizada no Survey

- ➔ Os 862 **foram filtrados para 34**, que atendem aos critérios, com base em resumo, título e palavras-chave.
- ➔ **Repetiram esse procedimento** até não encontrarem novos documentos relevantes nas bibliografias.
- ➔ Pesquisa foi repetida em 03/05/2016, **para tentar garantir a inclusão de todas as publicações de 2015** (“secondary step”), e detectou uma nova publicação, mas sem novas referências bibliográficas.

Resumo das pesquisas sobre bibliografias

Search step	New titles found	Core papers on GI
Primary	-	40
Step 2	862	34
Step 3	279	27
Step 4	47	8
Step 5	10	0
Secondary	-	1
Step 6	9	0
Total (based on abstract, title or keywords)		110
Distinct core papers on GI found (based on manual inspection of the full text of the 110 selected papers)		66

Metodologia utilizada no Survey

- ➔ A busca foi concluída com a **inspeção manual do texto completo** de cada trabalho, confirmando que **não há duplicatas** e que **cumprem os 4 critérios iniciais**.
- ➔ Assim, **identificaram 66 artigos principais sobre GI**: 40 na busca primária e 26 na busca bibliográfica recursiva.

Metodologia utilizada no Survey

- ➔ Este artigo possui um **material suplementar:**
  - https://ieeexplore.ieee.org/document/7911210/media
- ➔ Para cada um dos **66 trabalhos**, material suplem. indica:
  - **Critério de melhoria**
  - **Técnica de busca** utilizada
  - **Representação** de software
  - Características da **função de adequação (Fitness)**
  - **Linguagem de programação** do software modificado

Trabalhos existentes sobre *Genetic Improvement*

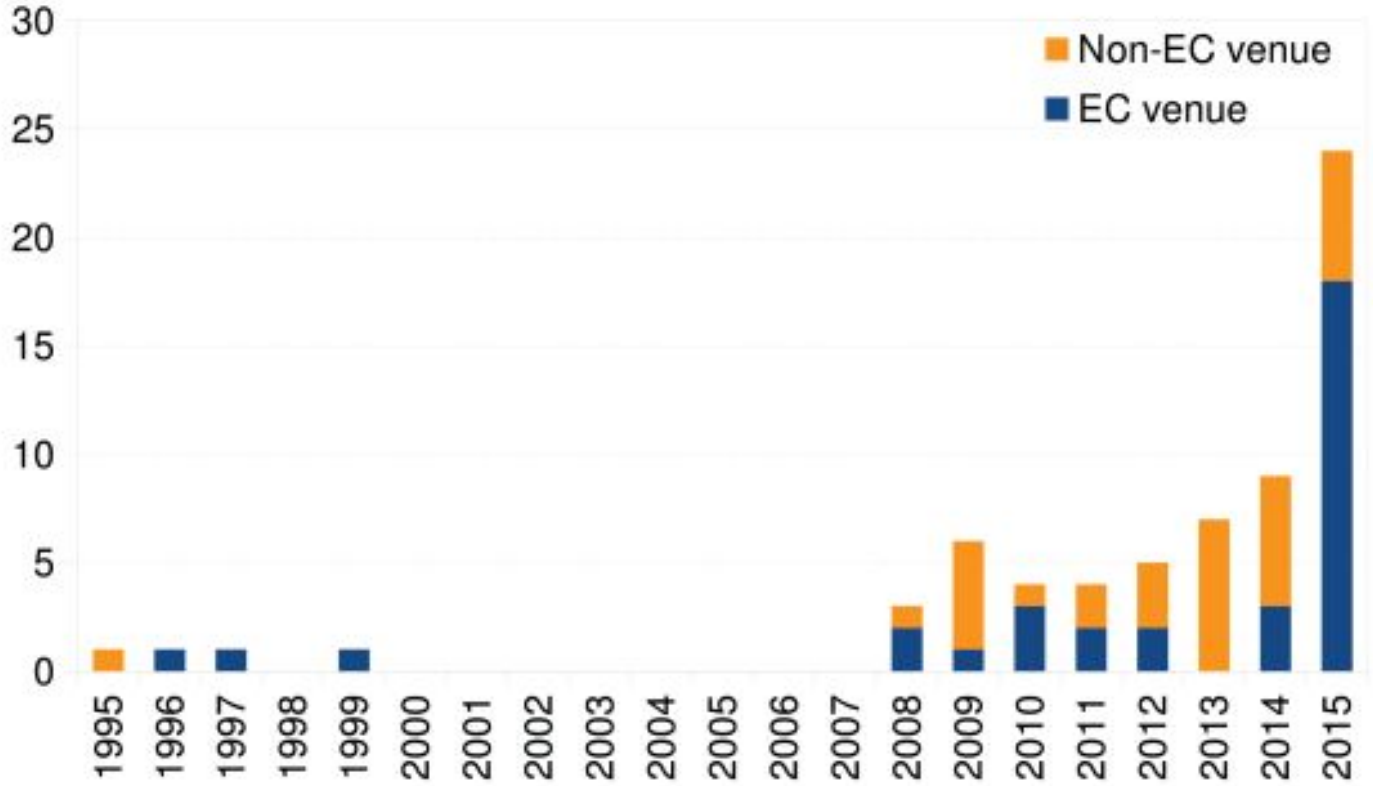
Trabalhos existentes sobre GI

- A necessidade de **otimização automatizada de software** é reconhecida há muito tempo e gerou várias abordagens.
- O que diferencia o GI das abordagens anteriores é a sua **generalidade e adaptabilidade**.
- O GI tira proveito da **abundância de código-fonte disponível**, reutilizando-o, em vez de partir do zero.
- O GI aplia o espaço de pesquisa de variantes de software, **diminuindo as restrições à correção do programa**.

Trabalhos existentes sobre GI

- ➔ O mais antigo trabalho de GI básico é sobre **paralelização de software**, por Walsh e Ryan (1995).
- ➔ No final dos anos 2000 o tema ressurge, com Arcuri e Yao (2008) e Arcuri (2009) sobre **reparo automatizado de software** e trabalho de White et al. (2008) e White (2009) sobre **redução de consumo de energia**.
- ➔ O sucesso desses estudos levou a uma rápida absorção do GI, com **diversas publicações a partir de 2008**.

Trabalhos existentes sobre GI



Trabalhos existentes sobre GI

- ➔ As seções a seguir **descrevem detalhadamente o processo GI típico**:
  - ◆ Preservação de propriedades
  - ◆ Uso de software existente
  - ◆ Critérios de melhoria (fitness; mono ou multiobjetivo)
  - ◆ Técnica de Busca
    - Operador de busca
    - Representação de software

Preservação de propriedades

Preservação de propriedades

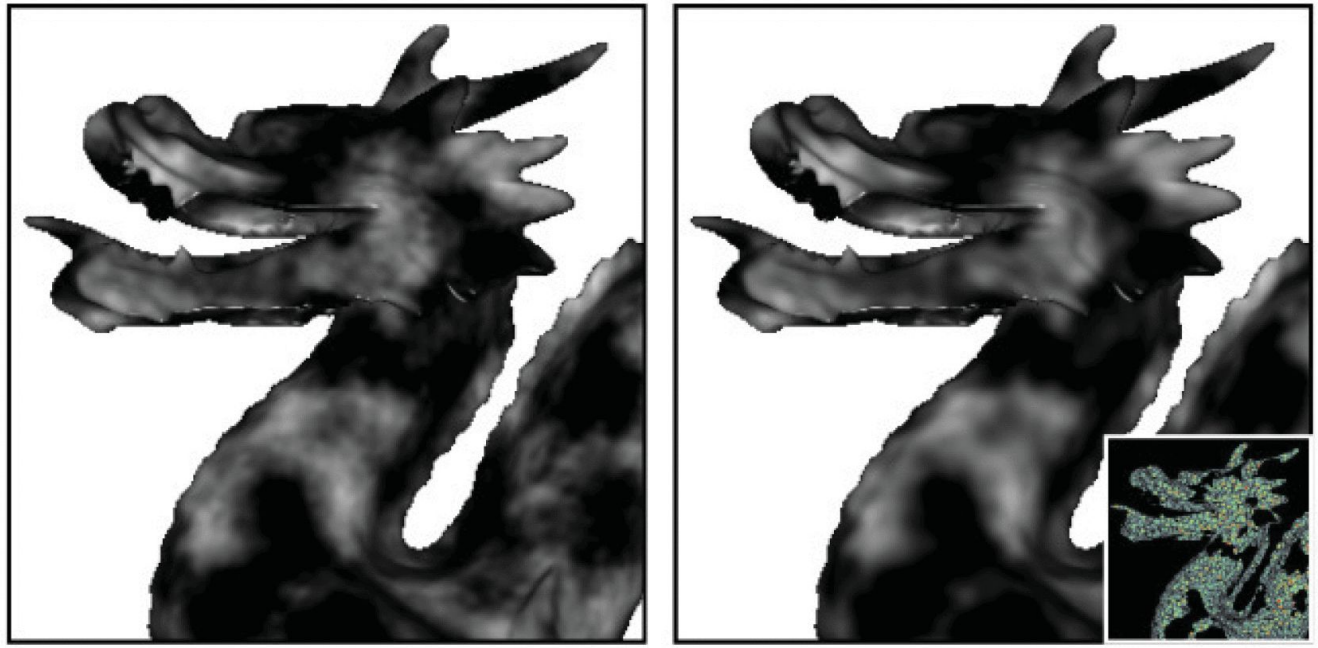
- ➔ A melhoria do software **implica que alguns aspectos mudam**, enquanto outros permanecem inalterados.
- ➔ Um sistema pode se tornar **mais rápido** através do GI, enquanto oferece o mesmo comportamento.
- ➔ Um **bug pode ser corrigido** enquanto mantém a funcionalidade “correta” existente (sem bug).
- ➔ Para **avaliar os aspectos funcionais inalterados**, precisamos de uma maneira de capturar a funcionalidade do software que precisa ser preservada.

Preservação de propriedades

- **Restrições relaxadas de fidelidade funcional** permitem uma troca entre propriedades de software.
- Negociar propriedades de software **pode ser benéfico**, especialmente em **ambientes com recursos limitados**.
- Sitthi-Amorn (2011), **trocou eficiência pela precisão**.
- Eles obtiveram uma **redução de 67% no tempo de execução**, permitindo flexibilidade na fidelidade da imagem em relação à saída do software original.

Preservação de propriedades

Simplificação de sombreamento: original e GI-modificada



Preservação de propriedades

- ➔ A questão permanece: **como capturar propriedades de software que precisam ser retidas?**

- ➔ Em todos os trabalhos empíricos examinados, **o teste de software foi usado como um proxy para capturar propriedades de software que precisavam ser retidas**.

Preservação de propriedades

- ➔ Se o conjunto de testes for todos casos possíveis, a equivalência de teste se tornará a **equivalência funcional**.
- ➔ O conjunto de todos os testes poderia ser quase infinito.
- ➔ **Relaxar essa noção, com um conjunto finito de testes, torna essa equivalência computável e tratável**.
- ➔ **Técnicas de seleção e de priorização de casos de teste** podem reduzir o custo de tempo de execução de testes.

Preservação de propriedades

- No caso mais simples, **o número de casos de teste aprovados serve como uma medida de adequação**.
- Dois trabalhos de 2015 usaram a seguinte função para transplante de software automatizado:

$$\text{fitness}(i) = \begin{cases} 1/3 \times (1 + |TX_i|/|T| + |TP_i|/|T|), & i \in I_C \\ 0, & i \notin I_C \end{cases}$$

- Onde i é a variante de software; IC é o conjunto de programas compatíveis; test suite T é o conjunto de funcionalidades desejadas; e TXi e TPi são os conjuntos de casos de teste não recortados e aprovados, respectivamente.

Preservação de propriedades

- **GenProg** (2012), ferramenta popular para reparo automatizado de software, usa **ponderação simples** na avaliação de adequação de programas modificados

$$\text{fitness}(C) = W_{\text{PosT}} \times \left|\left\{t \in \text{PosT} \mid P' \text{ passes } t\right\}\right| + W_{\text{NegT}} \times \left|\left\{t \in \text{NegT} \mid P' \text{ passes } t\right\}\right|$$

- Onde C representa a correção candidata, que produz o programa P´ ; WPosT atribui um peso aos casos de teste positivos (que o programa P original passa); WNegT é o peso atribuído ao número de casos de teste em que o programa P original falha, mas passam quando executados em P´. Os testes negativos são ponderados duas vezes mais do que os testes positivos.

Preservação de propriedades

- Arcuri e Yao (2008) usaram **função de distância ou erro**, que mede a diferença entre a saída obtida e a esperada.
- Arcuri (2009) **gerou casos de teste automaticamente**, outra vantagem do teste como captura de funcionalidade.
- Arcuri e Yao introduziram a ideia de co-evoluir casos de teste, **gerando testes unitários, com GA**, que passam quando executados no original e falham nos candidatos.
- **O programa original também serve como um oráculo** ao ser usado para se comparar as variantes de software.

Preservação de propriedades (GAP)

- ➔ Smith et al. (2015) concluíram que "***a qualidade dos patches é proporcional à cobertura do conjunto de testes usado durante o reparo***".
- ➔ Fast et al. (2010) usaram "dynamic program invariants", predicados, com testes para avaliar candidatos. **Resultados mais precisos do que com "soma ponderada"**.
- ➔ **Não está claro como caracterizar os conjuntos de testes** que melhor orientariam a busca por variantes de software aprimoradas. **Mais pesquisas são necessárias**. (GAP)

Uso de código existente

Uso de código existente

- O poder do GI reside na sua **aplicabilidade a uma infinidade de sistemas** do mundo real.
- Normalmente, o GI não começa do zero. Nos trabalhos principais, **parte-se de um sistema existente**.
- No campo da computação evolucionária, a **reutilização de código existente** corresponde à "transferência genética".

Uso de código existente

Fonte de Material Genético para GI

- ➔ Usar o código existente é fundamental no GI.
- ➔ A "hipótese da cirurgia plástica" (2014) supõe que o novo código pode frequentemente ser montado a partir de fragmentos de código que já existem.
- ➔ Estudos indicam que as alterações são 43% enxertáveis a partir da versão exata do software a ser melhorado.

Uso de código existente (GAP)

Fonte de Material Genético para GI

- Pode-se encontrar **três opções para a escolha do código**:
  - **o próprio programa que está sendo aprimorado**
  - **um programa diferente escrito na mesma linguagem**
  - **um pedaço de código gerado a partir do zero**
- Uma opção atualmente inexplorada é a **importação de uma linguagem diferente** do software a ser melhorado.

Uso de código existente

Transplante automatizado de código

- ➔ Outra área do GI, sobre reutilização de software.
- ➔ Destacado por Harman et al.(2013), citando **práticas e idéias do GI e do GP, aplicáveis à engenharia reversa**.
- ➔ Petke et al.(2014) foram os primeiros a usar o conceito no contexto GI. Com variantes do mesmo programa (**MiniSAT**), **obtiveram velocidades até 17% melhores**.

Uso de código existente (GAP)

Transplante automatizado de código

- Barr et al.(2015), usando GP, **extraíram um recurso do programa doador para o programa host** (alvo). Um codec de vídeo foi transplantado para o media player VLC.
- Marginean et al.(2015) **transferiram um recurso de visualização de gráfico** do CFLOW para o editor KATE.
- Sidiroglou et al. (2015) criaram **transplante sistemático para reparo de software**, com correções disponíveis online.
  * Ainda precisa ser tentada com buscas metaheurísticas.

Uso de código existente

Transplante automatizado de código

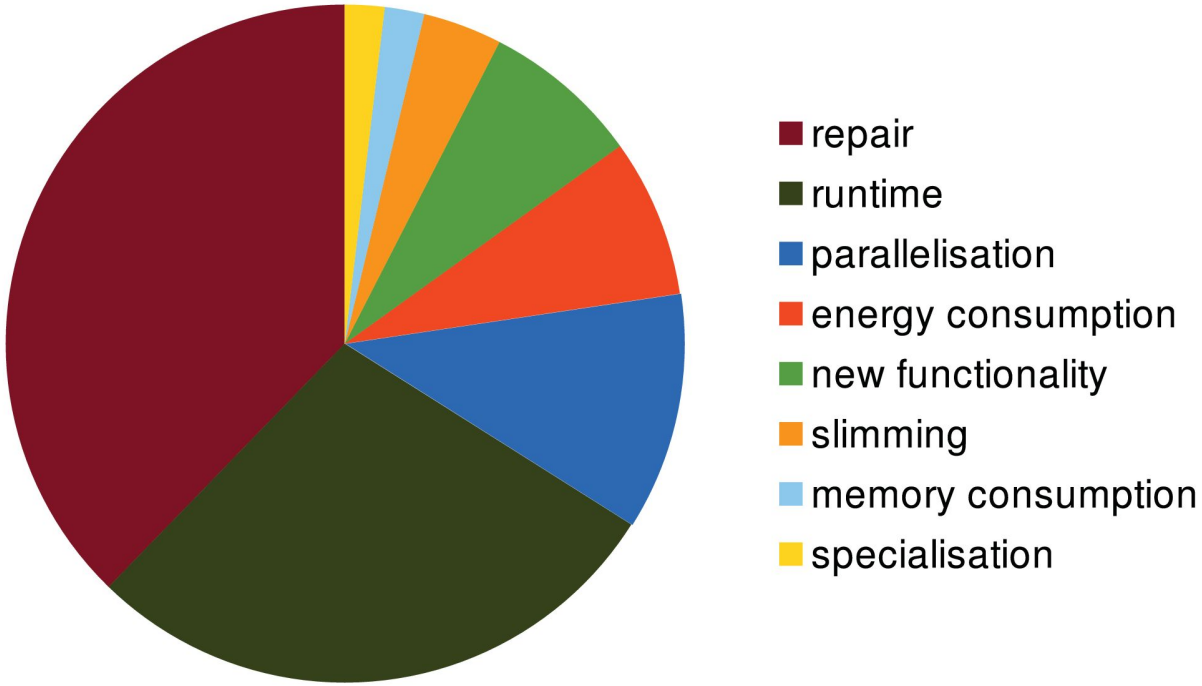
- **Na ausência da funcionalidade** no software existente, alguns **usaram GP para evoluir o recurso desejado do zero**.
- Harman (2014) criou recurso de tradução de idiomas, **transplantado para um sistema de mensagem instantânea**.
- Jia et al. (2015) ampliou um serviço de citação **e enxertou-o em um sistema** de desenvolvimento Web.
- Langdon e Harman (2015) criaram um recurso paralelo que **melhorou 10.000 vezes o software original extertado**.

Critérios de melhoria

Critérios de melhoria

- ➔ Critérios **Funcionais** e Critérios **Não Funcionais**.
- ➔ Alguns trabalhos usaram algoritmos evolutivos para **paralelizar o software**.
- ➔ White (2009) focou na **redução do consumo de energia**
- ➔ White e Arcuri (2009) defenderam o uso do GI para **melhorar as propriedades não-funcionais** do software.
- ➔ Grande parte deste *survey* concentra-se em **propriedades não funcionais**.

Critérios de melhoria



Aplicações dos estudos empíricos em trabalhos básicos sobre GI

Critérios de melhoria

- ➔ Uma dificuldade reside na **medição da propriedade não funcional** desejada.
- ➔ Por exemplo, para consumo de energia, **medições precisas podem simplesmente ser inviáveis**.
- ➔ No entanto, elas não são necessárias para a abordagem GI; **nós só precisamos de precisão relativa**, não absoluta.
- ➔ **O GI requer apenas uma função fitness que orientará a busca** de variantes de software desejáveis.

Critérios de melhoria

“Teste” como medida de Fitness

- ➔ Grande interesse pelo **reparo automatizado**, melhorando a correção dos programas conforme **medidos pelo “teste”**.
- ➔ **GenProg** é muito citado por trabalhos que **contribuem** com seu desenvolvimento, ou que o utilizam para **comparação**.
- ➔ Há trabalhos que **antecedem o GepProg**: Acuri (2008) e Wilkerson (2012), com otimização multiobjetivo.
- ➔ Ackling et al. (2011) **reparou software em linguagem Python**, com os mesmos princípios que o **GenProg** (2012).

Critérios de melhoria

“Teste” como medida de Fitness

- ➔ Além do risco óbvio de **criar novos bugs**, há também o perigo de introduzir um **comportamento prejudicial**.
- ➔ Schulte et al (2010, 2013 e 2015) **usaram "sandboxing", com testes em VM's**, para evitar danos ao ambiente.
- ➔ Muitos trabalhos de bugfix (GI) assumem que o programa defeituoso **contém suas próprias correções potenciais**, e assumem uma **liberdade de erros tipográficos e nomes de variáveis incorretos**.

Critérios de melhoria

Outras funções Fitness

- ➔ A maioria das funções de fitness que não contabilizam "passar" e/ou "falhar" nos casos de teste, **mede alguma propriedade não funcional do software**.
- ➔ **Dependem do hardware** e alguns deles podem ser manipulados **em parte por compiladores** (Ex: memória).
- ➔ Apesar de tratada anteriormente, há **um único exemplo de otimização do uso da memória com o GI** (Wu et al. 2015).

Critérios de melhoria

Outras funções Fitness

- ➔ A **propriedade não funcional** aprimorada com mais freqüência é o **tempo de execução**.
- ➔ Mas pode **variar entre sistemas e hardware**.
- ➔ **Número de linhas ou de instruções** foi considerado como um proxy independente do sistema **para o tempo de execução**.

Critérios de melhoria

Outras funções Fitness

- ➔ GI também pode ser aplicado **por sistema e hardware**, especializando-se em classes de programa ou dispositivos.
- ➔ Smarphones possuem **maior poder computacional** MAS tiveram um **aumento no consumo de baterias (energia)**.
- ➔ Problema na otimização de energia: como o uso é medido?

Critérios de melhoria

Outras funções Fitness

- ➔ Vários **resultados promissores para otimização de energia**, usando vários **métodos para aproximar seu consumo**.
- ➔ White (2009) usou simulação e um modelo linear para avaliar o consumo de energia; Bruce et al. (2015) usaram a *Intel Power Gadget API* **para aproximar o uso**.
- ➔ Para ambos os métodos, o software é executado isoladamente **para reduzir o ruído nas leituras de energia** devido a outros processos.

Critérios de melhoria

Outras funções Fitness

- Dada a dificuldade de fornecer uma medida de energia correta para a avaliação da aptidão, Harman e Petke (2015) **propuseram usar GI para evoluir a própria função de fitness** para um processo GI subsequente (***GI4GI***)
- Essa ideia (***GI4GI***) generaliza **para qualquer propriedade não funcional (ou funcional)** do software.

Critérios de melhoria

Melhoria Multiobjetivo

- A **otimização de propriedades não funcionais** pode, às vezes, significar **a degradação de outras propriedades** (funcionais ou não funcionais).
- Ex: **reduzir o tempo** de execução, **excluindo uma certa funcionalidade** do software
- Ex: **reduzir o consumo de memória**, em detrimento do **aumento do tempo de execução**.

Critérios de melhoria (GAP)

Melhoria Multiobjetivo

- ➔ Dados os muitos **critérios de melhoria conflitantes**, Arcuri (2009), White et al. (2011 e 2009) e Harman et al. (2012) **sugeriram a aplicação de algoritmos multiobjetivo**.
- ➔ Wu et al. (2015) aplicou essa abordagem para **otimizar o tempo de execução e o consumo de memória**.
- ➔ Entretanto, **o GI multiobjetivo de várias propriedades não funcionais ainda é pouco explorado**. (GAP)

Técnica de Busca

- ➔ **GI é poderoso por avaliar automaticamente várias versões de software**, buscando atender critérios de melhoria, preservando as propriedades desejadas.
- ➔ **GI pode avaliar milhares de programas** de software candidatos, enquanto humanos avaliam alguns poucos.
- ➔ Para explorar o enorme espaço de busca, **um algoritmo de busca eficiente precisa ser usado**.
- ➔ **GP é a abordagem heurística mais utilizada no GI.**

Técnica de Busca: Operadores

- ➔ Dado que a GP é o algoritmo evolutivo mais utilizada no GI, **seus operadores e similares são herdados na GI**.
- ➔ Operações básicas de remoção, substituição, adição, usadas em muitos trabalhos.
- ➔ Claire Le Goues et al. (GenProg) ampliou abordagem e **usou a localização de falhas na pesquisa de reparo**.
- ➔ Arcuri e Yao (2008) **co-evoluíram casos de teste** para melhorar capacidade de produzir reparos de erro válidos.
- ➔ Wu et al. (2015) usaram o teste de mutação.

Técnica de Busca: Representação no GI

- **Várias opções** para representar modificações em códigos:
  - AST (árvore de sintaxe abstrata), bytecode, o próprio código (em arquivo texto)
- Devido a limite de memória, ao invés da população conter códigos completos, ela **pode ser um conjunto de "patches"**.
- **Em GI é mais comum códigos em C e C++**, com representação gramatical do código BNF (Backus-Naur Form).
- **As ASTs são uma abordagem natural** para representar programas para fins de **programação genética**.

Técnica de Busca: “GI Binário”

- ➔ O GI foi aplicado diretamente a **binários, bem como ao código-fonte de alto nível**.
- ➔ Schulte et al. (2010) afirma que “**o GI binário**” tem os seguintes benefícios:
  - ◆ Técnica aplicável a **linguagens compiláveis**.
  - ◆ Reparos no **nível de instrução** podem ser executados (alterar declarações de tipo, operadores de comparação e atribuições a variáveis).
  - ◆ Assembly consiste em **pequeno conjunto de instruções**.

Técnica de Busca: “GI Binário”

- ➔ Schulte et al. (2013) **reparou defeitos no ARM**, com melhorando 86% no uso de memória e 95% no uso de disco.
- ➔ **Redução de 62% no tempo** para reparar os binários, **em comparação** com reparo em nível de código fonte.
- ➔ Técnica **aplicável a diferentes linguagens** (Java, C e Haskell)
- ➔ Consertaram **duas vulnerabilidades de segurança**.
- ➔ Sua abordagem **não requer acesso** ao código-fonte.

Técnica de Busca: *GI offline e GI online*

- GI on-line **modifica o software à medida que é executado**.
- Abordagens on-line incluem ECSELR e Gen-O-Fix (2014).
- ECSELR **incorpora adaptação de maneira autônoma**.
- Gen-O-Fix sugere que **pontos de verificação e escopo de mudança sejam definidos pelo programador**.
- **Abordagens híbridas** usam melhorias offline com base em monitoração obtida online, para posterior reimplantação.

Trabalhos Relacionados

Trabalhos Relacionados

- ➔ **A força do IG está na sua aplicabilidade geral.**
- ➔ **Outras abordagens** sacrificam essa generalidade, a fim de obter garantias de correção ou maiores taxas de sucesso dentro de uma faixa limitada de aplicação.
- ➔ O GI pode se beneficiar da incorporação de algumas dessas abordagens, com metaheurísticas.
- ➔ Alguns trabalhos relacionados muitas vezes empregam metaheurísticas e os métodos desenvolvidos no GI.

Relacionados: Síntese de código

- ➔ Balzer (1985) substituiu as especificações formais por linguagem natural.
- ➔ Gulwani et al. (2012) exploraram a síntese de programas baseada em exemplos. Sintetizaram funções de planilha do Microsoft Excel relativamente pequenas, mas úteis.
- ➔ "Projeto automatizado de algoritmos" (2014) usa busca computacional para descobrir e melhorar algoritmos para problemas específicos.
- ➔ O termo "síntese" tem sido usado também para se referir à adição de novas funcionalidades.

Relacionados: Reparo automatizado

- **Fontes úteis:** Monperrus (2014) e Claire Le Goues (2013).
- Tipicamente, abordagens não-GI para reparos fazem suposições sobre as especificações disponíveis, limitam os tipos de bug sob consideração, ou restringem as transformações que podem ser aplicadas.
- **Perdem em generalização, mas possibilitam explorar exaustivamente os possíveis reparos**.
- Em testes de mutação, os mutantes são usados para medir a eficiência dos conjuntos de testes na detecção de programas defeituosos

Relacionados: Transformação de programa

- ➔ Aplicação determinística de transformações de preservação semântica.
- ➔ Refatoração de código é uma forma de transformação.
- ➔ Transformação de programa tradicional também pode ser alcançada usando busca metaheuristic.
- ➔ Normalmente, essas transformações baseadas em busca são restritas a operações de preservação de semântica.

Relacionados: Ajuste de parâmetros

- ➔ Parâmetros podem **ajustar o desempenho de um programa**.
- ➔ Exemplo: compiladores modernos.
- ➔ Projetos automatizados onde **a busca é usada para selecionar o melhor conjunto** de parâmetros para um determinado problema.

Relacionados: Computação aproximada

- ➔ A melhoria das propriedades não-funcionais usando o GI **pode levar à exploração de uma frente de Pareto no espaço objetivo**, trocando algumas funcionalidades em troca de ganhos não-funcionais.
- ➔ Isso empurra o GI para a **fronteira da computação aproximada**.
- ➔ Trabalhos usam **interações ou feedbacks automáticos ou por influência do usuário**, para ajustes e compensações de desempenho.

Relacionados: Reparo de estrutura de dados

- Programas errados podem resultar em **estruturas de dados inválidas**.
- Uma função “repOK” pode verificar violações de restrições de integridade estrutural.
- Ferramenta Juzi (2008) usa a execução simbólica da função “repOK” para sugerir reparos na estrutura de dados.
- Juzi realiza uma busca sistemática através das variáveis fornecidas.

Relacionados: Estudos do código existente

- ➔ É útil ajudar tanto pesquisadores como desenvolvedores a entender o espaço de busca.
- ➔ Uma suposição de ferramentas de reparo de software GI é que o material necessário para corrigir a falha está dentro do código existente.
- ➔ Estudos identificaram redundância em nível de token e em nível de linha, em repositórios de códigos abertos.

Relacionados

Fatiamento, Avaliação Parcial e Especialização:

- ➔ O fatiamento reduz um programa a uma forma mínima que retém um subconjunto desejado de seu comportamento.
- ➔ Pode ser usado tanto para otimizar o tamanho do software existente quanto para extrair uma funcionalidade desejada.
- ➔ A avaliação parcial procura otimizar um programa, especializando-o em relação a alguns insumos conhecidos.
- ➔ Os programas podem ainda ser especializados para o ambiente em que são executados.

Conclusões

Conclusões

- ➔ A **crescente quantidade e tamanho de software** sendo desenvolvido, **requer técnicas automatizadas** para a melhoria de software.
- ➔ Devido à abundância de código, as abordagens de otimização disponíveis **não precisam ser iniciadas do zero**.
- ➔ **Metaheurísticas, como os algoritmos evolutivos**, demonstram ser bem-sucedidas na **exploração de grandes espaços** de pesquisa (Ex: espaço de variantes de software).
- ➔ **O GI combina esses insights** para melhorar o software por meio da aplicação da pesquisa automatizada.

Referências

Referências

- Petke J, Haraldsson S, Harman M, Langdon WB, White D & Woodward J (2018) Genetic Improvement of Software: a Comprehensive Survey, **IEEE Transactions on Evolutionary Computation**, 22 (3), pp. 415-432. https://doi.org/10.1109/TEVC.2017.2693219.

Obrigado

Dúvidas ou sugestões?

saviosampaio@inf.ufg.br