UNIVERSIDADE
FEDERAL DE JUIZ DE FORA

Especialização em Gerência de Projetos de Software na Era de Dados de Sensores e IA

Trabalho Prático de Conclusão de Disciplina

Link para Github

https://github.com/wesleyteles/sbst_vs_llms_compartivo/

UFJF - 2026

COMPARANDO SBST VS LLMS
TRABALHO PRÁTICO

WESLEY DE SÁ TELES

PROF. LUCIANA C. D. CAMPOS

LINK PARA GITHUB:
https://github.com/wesleyteles/sbst_vs_llms_compartivo/

JUIZ DE FORA
01/2026

1. Análise do Resultado Gerado (SBST)

1. **Cobertura de Ramos (Branch Coverage):** O EvoSuite cumpriu o objetivo da atividade prática ao gerar inputs que exercitam:
   - A exceção (a=0).
   - O retorno vazio (Delta negativo).
   - O retorno único (Delta zero).
   - O retorno duplo (Delta positivo). Isso demonstra a capacidade dos algoritmos genéticos de encontrar "caminhos lógicos" através da evolução de valores de entrada.

2. **Fitness Function (Função de Aptidão):** O algoritmo guiou a busca. Por exemplo, para encontrar o caso delta == 0, o EvoSuite minimizou a distância entre o valor calculado de delta e zero até encontrar os parâmetros 1.0, 2.0, 1.0.

3. **Limitações de Legibilidade:** Embora o exemplo seja simples o código, em sistemas complexos, o EvoSuite tende a gerar nomes de variáveis genéricos (ex: double0, calculadoraBhaskara1) e valores numéricos que podem parecer aleatórios (ex: 1452.32), dificultando a compreensão humana sobre o porquê daquele teste existir.

4. **Teste de Regressão:** assume que o comportamento atual do código está correto. Ele captura esse comportamento em assertEquals. Se você introduzirmos um bug na lógica matemática original, o EvoSuite vai criar um teste que "protege" esse bug, pois ele não sabe a regra de negócio, apenas explora o código.

2. Relatório de Simulação: PITest (Mutation Coverage)

Classe Alvo: CalculadoraBhaskara

Testes Executados: CalculadoraBhaskara_ESTest (Gerados pelo EvoSuite)

Resumo da Execução

- Line Coverage (Cobertura de Linhas): 100% (4/4 ramos cobertos)
- Mutation Score (Eficácia): 100% (Todos os mutantes foram mortos)
- Mutantes Gerados: 7
- Mutantes Mortos: 7

O PITest aplicou operadores de mutação no bytecode da classe.

1. Mutante: Alteração de Condicional (Negated Conditional)

- Onde: Linha 10 if (a == 0) transformado em if (a != 0)
- Resultado: KILLED

• Por que morreu? O test0 do EvoSuite envia a=0 e espera uma IllegalArgumentException. Com a mutação, o código não lançaria a exceção (entraria no cálculo). O teste falhou por não receber a exceção esperada.

2. **Mutante:** Alteração de Operador Matemático (Math Mutator)
   • Onde: Linha 14 double delta = (b * b) - (4 * a * c) transformado em ... + (4 * a * c)
   • Resultado: KILLED
   • Por que morreu? O test1 (Delta Negativo) usa valores onde b^2 é menor que 4ac. Ao trocar - por +, o Delta vira positivo. O teste esperava um array vazio [] mas o código retornou um array com duas raízes. A asserção assertEquals(0, resultado.length) falhou.
3. **Mutante:** Alteração de Condicional de Borda (Conditionals Boundary)
   • Onde: Linha 17 if (delta < 0) transformado em if (delta <= 0)
   • Resultado: KILLED
   • Por que morreu? O test2 (Delta Zero) gera um cenário onde Delta é exatamente 0. O código original passaria para o próximo if. O código mutado entraria no return new double[]{}. O teste esperava 1 raiz e recebeu 0. Falha na asserção.
4. **Mutante:** Alteração de Valor de Retorno (Return Values)
   • Onde: Linha 24 return new double[]{raiz} transformado em return null
   • Resultado: KILLED
   • Por que morreu? O test2 tentou verificar o tamanho do array retornado e encontrou um null, gerando NullPointerException ou falha de asserção imediata.
5. **Mutante:** Inversão de Lógica Aritmética (Invert Negatives)
   • Onde: Linha 23 double raiz = -b / (2 * a) transformado em b / (2 * a) (removeu o sinal negativo)
   • Resultado: KILLED
   • Por que morreu? O test2 verifica o valor exato da raiz (assertEquals(-1.0, ...)). O mutante retornaria 1.0. A asserção de valor do EvoSuite pegou o erro.

3. Análise Crítica da Parte A (SBST)

Conforme os objetivos da Unidade 5, podemos concluir sobre a Automação Evolutiva (EvoSuite):

1. **Alta Cobertura:** O algoritmo genético foi extremamente eficaz em encontrar caminhos para matar todos os mutantes lógicos simples em um código matemático.

2. **Asserções Fortes:** O EvoSuite chamou os métodos (smoke testing) e capturou o valor de retorno (snapshot) e criou asserções rígidas (assertEquals). Isso foi importante para matar os mutantes 2 e

3. **Eficiência:** O processo gerou em segundos testes que cobrem 100% dos cenários, algo que manualmente

exigiria cálculo prévio dos valores de Delta.

4. O Prompt para a LLM

"Atue como um Engenheiro de Qualidade de Software Sênior. Eu tenho a seguinte classe Java CalculadoraBhaskara que calcula raízes de uma equação de segundo grau.
Sua tarefa é:
1. Gerar uma classe de teste JUnit 4 ou 5 completa.
2. Criar casos de teste que cubram 100% dos branches (ramos) lógicos do código (Branch Coverage).
3. Incluir testes para cenários de borda (ex: Delta negativo, Delta zero, exceções).
4. Adicionar comentários explicando o que cada teste valida."

5. Comparativo Crítico: SBST (EvoSuite) vs. LLM (ChatGPT)

Critério	SBST (EvoSuite) - Parte A	LLM (IA Generativa) - Parte B
Legibilidade	**Baixa.** Gera nomes de variáveis aleatórios (ex: double0) e valores difíceis de ler (ex: 1.4323). O foco é a máquina.	**Alta.** Gera nomes descritivos (deveRetornarDuasRaizes...) e usa exemplos matemáticos "limpos" (ex: raízes 2 e 3).
Cobertura	**Garantida.** O algoritmo matemático força a execução de todos os ramos (if/else) até atingir a meta.	**Dependente do Prompt.** A LLM "adivinha" os casos de teste baseada no texto. Pode esquecer um caso de borda se não for explicitamente pedida para cobrir "todos os branches".
Manutenção	**Difícil.** Se a regra de negócio mudar, o teste gerado é descartável, pois é difícil de entender e ajustar manualmente.	**Fácil.** O código parece escrito por um humano, facilitando ajustes futuros ou a adição de novos casos.
Confiança (Oráculo)	Assume que o código atual está certo. Se houver um bug, ele cria um teste que valida o bug.	Tenta inferir a intenção correta. Se o código tiver um erro lógico óbvio, a LLM às vezes "corrige" o teste esperando o valor certo, o que faria o teste falhar (o que é bom).

6. Conclusão da Atividade:

A LLM (Parte B) se mostrou mais eficaz para **documentação e produtividade inicial**, gerando testes que servem como especificação do sistema. A SBST (Parte A) é superior para **stress test e descoberta de falhas profundas** em lógicas complexas onde humanos (e LLMs) podem falhar em visualizar todas as combinações matemáticas.