Relatório: Engenharia de Software Baseada em Busca

Bruna Mariana F. de Souza[1], **Débora da C. Medeiros**[1], **Valber C. Tavares**[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brazil

bruna.mariana@icomp.ufam.edu.br, debora.medeiros@icomp.ufam.edu.br

valber.tavares@icomp.ufam.edu.br

Abstract. *Software Engineering Based on Search (SBSE) employs optimization techniques, such as evolutionary algorithms, to enhance the efficiency of software testing. Tools like EvoSuite and COLEMAN represent an effective integration of SBSE into software testing practice, aiming to improve its efficiency and effectiveness. This report presents a case study on the use of these tools applied to open-source projects, detailing how these tools were implemented. The results obtained demonstrate that the application of EvoSuite increased code coverage and the MAB Thompson Sampling policy can be an efficient choice in terms of time when resources are limited.*

Resumo. *A Engenharia de Software Baseada em Busca (SBSE) emprega técnicas de otimização, como algoritmos evolutivos, para melhorar a eficiência dos testes de software. Ferramentas como o EvoSuite e o COLEMAN representam uma integração eficaz da SBSE na prática dos testes de software, visando aprimorar sua eficiência e eficácia. Este relatório apresenta um estudo de caso sobre o uso destas ferramentas aplicadas a projetos open source, detalhando como essas ferramentas foram implementadas. Os resultados obtidos demonstram que a aplicação do EvoSuite elevou a cobertura do código e a politica MAB Thompson Sampling pode ser uma escolha eficiente em termos de tempo quando os recursos são limitados.*

1. Introdução

De forma ampla, pode-se definir um software como um conjunto de instruções que direcionam o computador na realização de tarefas específicas. Ele não existe de forma isolada, ele é criado por pessoas e para pessoas nas mais distintas áreas. A cada necessidade atendida, novas necessidades e demandas surgem, naturalmente, para atender as novas necessidades são necessários novos conjuntos de instruções, e isto aumenta a complexidade do software. De acordo com [TERCEIRO 2013], a complexidade de um sistema de software está relacionada aos elementos que influenciam o custo envolvido em sua criação e manutenção. Em outras palavras, a elevação na complexidade dos softwares também eleva os custos no seu desenvolvimento e manutenção.

Desenvolver um software é um processo que envolve muitas etapas, para manter a qualidade do software é indispensável que uma destas etapas seja a de testes. Testar possibilita encontrar erros e permite analisar se o software se comporta como esperado em diferentes cenários [Teixeira 2022]. Contudo, a realização de testes não é trivial, pois

envolve uma complexa interação de variáveis e restrições. Mesmo os testes automatizados que aceleram o processo de testes, não estão imunes aos problemas do aumento da complexidade [Oliveira 2024].

Neste contexto a utilização de técnicas de inteligência artificial pode ser de grande ajuda no enfrentamento da crescente complexidade. Uma área de pesquisa que se concentra nesse desafio é a Engenharia de Software Baseada em Busca (SBSE), que explora técnicas de otimização, como algoritmos evolutivos, para melhorar o desenvolvimento de softwares [Pinheiro 2022]. Para lidar com esses desafios, surgiram ferramentas de geração de testes unitários que empregam técnicas de inteligência artificial, como algoritmos genéticos e geração aleatória direcionada por feedback.

Inicialmente este relatório aborda um estudo de caso da automatização de geração de testes unitários com a ferramenta EvoSuite, para analisar como seu uso pode influenciar na maior cobertura dos testes. Complementarmente, a segunda parte deste relatório trata da aplicação da abordagem COLEMAN, propondo alterações com o objetivo de melhorar os resultados para o problema que a abordagem se propõe a solucionar, como a priorização de casos de teste em ambiente de integração contínua. São dois os problemas estudados neste relatório, o primeiro diz respeito a necessidade de maior cobertura do código por testes dentro de um tempo viável enquanto que o segundo trata da volitividade dos casos de testes em um ambiente de integração contínua.

2. EvoSuite

Diferentes estratégias de teste envolvem a seleção cuidadosa de valores de entrada. Estes devem incluir tanto valores de borda — que simulam os limites em determinadas condições — quanto valores intermediários, que representam o funcionamento típico esperado do sistema. Escolher estes valores influencia diretamente na cobertura do código, esta que trata-se de uma métrica quantitativa dada em porcentagem, que indica quanto do código foi efetivamente testado, possibilitando saber se aquele conjunto de testes é pertinente [Pinheiro 2022].No entanto, um conceito crucial na tomada de decisões surge: o trade-off, que pode ser aplicado a essa situação: é desejável que o software seja confiável para não acarretar em prejuízos com manutenção, e isto é possível quanto maior for a cobertura do código. Por outro lado, aumentar a cobertura do código implica em um maior tempo e, consequentemente, em mais recursos.

É preciso alcançar um equilíbrio viável nessa situação, o EvoSuite entra nesse cenário como uma ferramenta que faz uso de técnicas de otimização, ele atua com a geração automática de testes unitários para classes escritas em Java. Segundo [Vogl et al. 2021] ele faz uso de algoritmos de busca meta-heurística para criar suítes de testes em JUnit, com objetivo principal de aumentar a cobertura do código. Porém aqui entra a relevância do uso da abordagem meta-heurística, pois essas abordagens buscam o equilíbrio no problema disposto. Elas encontram a solução ótima em relação a um tempo viável. Não se busca apenas atingir uma alta cobertura, mas sim alcançar a máxima cobertura possível dentro de um tempo viável [Pinheiro 2022].

A versão atual do EvoSuite é a 1.2.1, ela possui segundo sua documentação oficial as seguintes alterações:

- Suporte Java 9 para DSE

- Configuração atualizada do Docker
- Suporte para saída JUnit 5
- Refatoração do código relacionado a genéricos
- Suporte Java 9 para plug-in Eclipse
- Refatoração e reformatação gerais
- Correções de bugs para vários travamentos e problemas menores

Desde a versão 1.1.0 o EvoSuite utiliza como algoritmo padrão o DynaMOSA [Vogl et al. 2021]. O DynaMOSA é considerada uma das técnicas mais avançadas atualmente para SBST, ele é um algoritmo de evolução que atua no domínio dos testes. Utiliza-se de funções especiais para avaliar casos de testes a partir de um grupo de testes gerados aleatoriamente partindo dos limites estabelecidos [Datskiv 2023]. Os algoritmos evolutivos são baseados na seleção natural, o que eles fazem é simular a evolução de uma população, dando um valor de aptidão para cada individuo afim de que os detentores dos maiores valores tenham maior probabilidade de se tornarem pais e passarem suas características adiante. O DynaMOSA é um algoritmo evolutivo multiobjetivo, ele se propõe a otimizar situações onde há diversos objetivos que conflitam entre si simultaneamente [Li 2023]. Nessa situação não há como levar em conta somente o valor de aptidão, o que é considerado é o quanto um individuo/solução domina uma outra solução e não é dominada por nenhuma outra.

O DynaMOSA é uma evolução do MOSA, focado na seleção eficiente de casos de teste ao considerar dependências estruturais entre alvos de teste. Ambos os algoritmos utilizam dados de entrada e declarações de programa como genes para avaliar a eficácia dos testes. O DynaMOSA se destaca por sua seleção dinâmica, que busca otimizar a cobertura de código de maneira mais eficiente e direcionada. Ele foca em cobrir primeiro as condições externas, como as de declarações if aninhadas, antes de avançar para as internas. Isso permite uma seleção de teste mais eficiente e lógica. [Datskiv 2023].

O EvoSuite, com seu algoritmo padrão DynaMOSA, tem sido utilizado em diversos projetos de software demonstrando uma melhoria significativa na eficiência e eficácia dos testes unitários. Anualmente ocorre uma competição que visa avaliar e impulsionar ferramentas de geração de testes, no SBST 2021, o EvoSuite alcançou a maior pontuação geral entre as ferramentas concorrentes. De acordo com [Vogl et al. 2021] o EvoSuite alcançou "71% de cobertura de ramificação, 77% de cobertura de linha e uma pontuação de mutação de 53%, usando um orçamento de busca de 120 segundos nas 25 classes consideradas para a competição."

2.1. Estudo de Caso - EvoSuite

Um estudo de caso foi realizado para analisar o quão impactante é a geração automática de testes unitários utilizando o EvoSuite. Para o estudo, foi escolhido o repositório do projeto MyRobotLab hospedado no GitHub. O MyRobotLab é um framework de código aberto desenvolvido para projetos de robótica e controle de máquinas criativas. Como critérios para a escolha, foi levado em conta ele ser um projeto desenvolvido em Java, o que é fundamental a geração de testes com o EvoSuite. Tratando-se de um projeto que ainda é mantido e constantemente atualizado por seus desenvolvedores além de ser um projeto robusto o que possibilita um amplo campo de aplicação para os algoritmos de busca do EvoSuite. A tabela 1 mostra descreve de forma sucinta as principais características do repositório.

Table 1. Dados descritivos do repositório selecionado

Repositório	Commits	Contribuidores	Branches	Linguagem Principal	Licença
MyRobotLab	14,391	24	42	Java	Apache License 2.0

2.1.1. Configuração do ambiente

Antes de sua utilização, é necessário garantir a presença e a compatibilidade do Java Development Kit (JDK) na máquina. Isso pode ser verificado executando o comando:

```
javac –version
```

Além do JDK, outra dependência crucial para a geração de testes é o Apache Maven. Para confirmar se o Maven está corretamente instalado, verifica-se com o comando:

```
mvn –version
```

Uma vez que as ferramentas necessárias estejam configuradas, o próximo passo é obter o código-fonte do repositório que será testado, sendo necessário posteriormente navegar até o diretório do repositório clonado, o que pode ser feito com os comandos:

```
git clone https://github.com/MyRobotLab/myrobotlab
cd myrobotlab
```

É necessário compilar as classes do projeto, os comandos abaixo invocam o Maven para compilar o código-fonte do projeto em arquivos .class, seguindo as instruções definidas no arquivo pom.xml do projeto.

```
mvn package
mvn install
```

Para analisar o sucesso da Geração Automática de Testes, foram utilizadas as seguintes métricas [Marques and Fernandes 2020]: Cobertura de Código: Comparação entre a cobertura de código antes e após a implementação dos testes automáticos. Testes de Mutação: Análise da eficácia dos testes em identificar e eliminar mutantes (erros inseridos propositalmente) antes e após a geração dos testes.

Logo, antes de gerar os testes com o EvoSuite, foi necessário obter a cobertura atual para comparação posteriori. Para isso é preciso primeiro executar os testes:

```
mvn clean test
```

E em seguida utilizou-se o JaCoCo para gerar o relatório de cobertura do código do projeto Java:

```
mvn jacoco:report
```

Para a comparação posteriori com a segunda métrica escolhida: Testes de Mutação. Foi selecionada a ferramenta Pitest (PIT), pois ela pode ser integrada ao processo de build usando Maven, sendo para isso necessário adicionar o plugin PIT ao arquivo pom.xml que está na raiz do projeto, o que foi ser feito inserindo os comandos a seguir:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>1.16.0</version>
            <configuration>
                <threads>16</threads>
            </configuration>
        </plugin>
    </plugins>
</build>
```

O PIT gera uma série de mutações no código-fonte do programa e executa os testes existentes contra esses mutantes para determinar a cobertura dos testes e a eficácia em detectar mutações. Após a inclusão do plugin, para a execução do PIT com o Maven foi preciso no terminal fornecer o comando a seguir para obter os relatórios.

```
mvn test-compile org.pitest:pitest-maven:mutationCoverage
```

2.1.2. Instalação do EvoSuite

Para garantir a compatibilidade com as versões mais atuais do java, foi optado por instalar a versão mais atual do EvoSuite, a versão 1.2.1. A instalação desta versão demanda o clone do repositório do projeto, seguido pela disponibilização local do mesmo para ser usado como dependência em outros projetos. Os passos a seguir detalham o processo.

```
git clone https://github.com/EvoSuite/evosuite.git
cd evosuite
mvn package
mvn install
```

Com o EvoSuite devidamente instalado e o projeto disponível localmente, o próximo passo consistiu em executar o EvoSuite para gerar os testes. Foi usado o seguinte comando para executar o EvoSuite dez vezes:

```
for I in {1..10}; do sudo java -jar ~/myrobotlab/evosuite/master
   /target/evosuite-master-1.2.1-SNAPSHOT.jar -target ./target/
   classes -Doutput_variables=TARGET_CLASS,criterion,Coverage,
   Total_Goals,Covered_Goals,Size,Length,MutationScore ; done
```

Por trata-se de um projeto robusto, foi criado um script bash para executar os testes gerados pelo Evosuite: para que o JUnit alcançasse a todos os arquivos .class.

Listing 1. Script Bash

```
#!/bin/bash

# Diret rio onde est o localizadas as classes de teste
TEST_DIR="./evosuite-tests"

# Comando a ser executado para cada classe de teste
COMMAND="sudo␣java␣--add-opens␣java.desktop/java.awt=ALL-UNNAMED
   ␣-cp␣.:\$HOME/dados_evosuite_myrobotlab/myrobotlab/target/
   classes:\$HOME/dados_evosuite_myrobotlab/myrobotlab/evosuite-
   tests-compiled:\$HOME/dados_evosuite_myrobotlab/myrobotlab/
   evosuite/master/target/evosuite-master-1.2.1-SNAPSHOT.jar:\
   $HOME/Tutorial_Stack/target/dependency/junit-4.12.jar␣org.
   junit.runner.JUnitCore"

# Arquivo para salvar a sa da dos comandos
LOG_FILE="test_results.log"
# Arquivo para salvar os nomes das classes com falhas
FAILURE_FILE="failures.log"

# Cria ou limpa os arquivos de log
> "\$LOG_FILE"
> "\$FAILURE_FILE"

# Vari vel para contar os testes OK
tests_ok=0
# Vari vel para contar o total de testes
total_tests=0

# Loop recursivo para percorrer todas as classes de teste
for file in \$(find "\$TEST_DIR" -type f -name '*.java'); do
    # Verifica se o nome do arquivo cont m a palavra "
       scaffolding"
```

```
    if [[ "\$file" == *scaffolding* ]]; then
        continue  # Pula o arquivo se contiver "scaffolding" no
            nome
    fi

    # Extrai o caminho relativo da classe de teste
    relative_path="\${file#\$TEST_DIR/}"
    # Remove a extens o .java do nome do arquivo
    class_name="\${relative_path%.java}"
    # Substitui '/' por '.' para obter o nome do pacote completo
    class_name="\${class_name//\//.}"

    # Executa o comando para a classe de teste
    \$COMMAND "\$class_name" >> "\$LOG_FILE" 2>&1

    # Verifica se o teste foi bem-sucedido (exit code 0)
    if [ \$? -eq 0 ]; then
        ((tests_ok++))
    else
        # Salva o nome da classe com falha no arquivo
        echo "\$class_name" >> "\$FAILURE_FILE"
    fi
done

# Extrai a quantidade de testes do arquivo de log e incrementa
    no total de testes
total_tests=\$(grep -a -o 'OK␣([0-9]\+␣test[s]*' "\$LOG_FILE" |
    sed 's/OK␣(\([0-9]\+\)␣test[s]*.*/\1/' | paste -sd+ - | bc)

echo "Quantidade␣de␣classes␣OK:␣\$tests_ok"
echo "Classes␣com␣falhas:␣\$(cat␣\$FAILURE_FILE)"
echo "Total␣de␣testes␣executados:␣\$total_tests"
```

O EvoSuite é uma ferramenta projetada para gerar casos de teste para programas Java, utilizando os arquivos .class (bytecodes Java). Os arquivos .class do projeto estão distribuídos em diversos diretórios do projeto, sendo preciso criar uma lógica que capturasse todos estes arquivos e adicionalmente executasse o EvoSuite dez vezes neste projeto.

Seguindo estes passos o ambiente de testes foi configurado com as seguintes especificações:

- Sistema Operacional: Linux
- JDK (Java Development Kit): java 11.0.22
- Apache Maven: Apache Maven 3.6.3
- EvoSuite: Versão 1.2.1

2.2. Análise da Aplicação do EvoSuite

Antes da geração dos testes com o EvoSuite, foi realizado uma análise prévia do projeto e seus respectivos testes. Antes de obter as métricas para comparação a posteriori, os testes

do projeto foram executados mais de uma vez para melhor análise. Na primeira rodada de execuções foi obtido o resultado em resumo:

- Total de testes executados: 303
- Falhas: 0
- Erros: 0
- Ignorados: 26
- Tempo decorrido: 6 minutos e 10 segundos

O primeiro resultado promissor se deu após a execução dos testes gerados:

- Total de testes executados: 2707
- Falhas: 0

De forma comparativa, o EvoSuite conseguiu gerar um total de 2.007 testes, o que representa um acréscimo de 2.404 em relação aos testes inicialmente existentes no projeto. Contudo, apenas esse aumento quantitativo não é, por si só, significativo. São necessárias análises adicionais para determinar se essa quantidade adicional de testes é verdadeiramente relevante para o projeto.

2.2.1. Cobertura do Código

Para mitigar a natureza estocástica dos algoritmos de busca usados pelo EvoSuite e para garantir que os resultados não sejam influenciados por flutuações aleatórias, o EvoSuite foi executado dez vezes no projeto. A execução repetida permite avaliar a consistência dos resultados e identificar padrões ou tendências ao longo das iterações [Vogl et al. 2021]. O ideal seria executar o EvoSuite ao menos trinta vezes, porém devido a limitações de tempo e hardware foi escolhido o valor mínimo de dez vezes para a sua execução. Devido a estas limitações, não foi executado o EvoSuite para diferentes configurações, sendo apenas utilizada a sua configuração padrão.

O projeto MyRoboLab contém em seu repositório myrobotlab/src/test 164 arquivos de testes, estes contendo cerca de 303 testes como foi mostrado no momento da execução. Com auxilio da ferramenta JaCoCo, foi possível comparar e obter a cobertura do código por aqueles testes. A figura 1 mostra o gráfico onde a cobertura do código foi de apenas 29%, uma cobertura que pode ser considerada baixa.

Após as dez execuções do EvoSuite, foram obtidos 212 arquivos de testes. A figura 2 mostra com o gráfico, uma melhora significativa na cobertura do código após a geração de testes com o EvoSuite. Passando do valor de 29% para 85%. Os testes gerados de forma automática aumentaram em 56% a cobertura.

Afim de entender como as dez execuções do EvoSuite foram relevantes para um melhor resultado, o gráfico 3 mostra a distribuição da cobertura de código nas dez repetições executadas. Nele é possível observar que a cobertura média está entre 0,4 e 0,6 o que significa em termos práticos que metade das repetições do EvoSuite teve uma cobertura de código igual ou superior a 40% e metade teve uma cobertura igual ou inferior a 60%. Uma mediana entre 0,4 e 0,6 indica que, em geral, as repetições não alcançaram uma cobertura de código muito alta. O ideal, seria que esse valor estivesse mais próximo de 1,0, o que indicaria uma cobertura de código próxima de 100%. Porém isso ajuda a

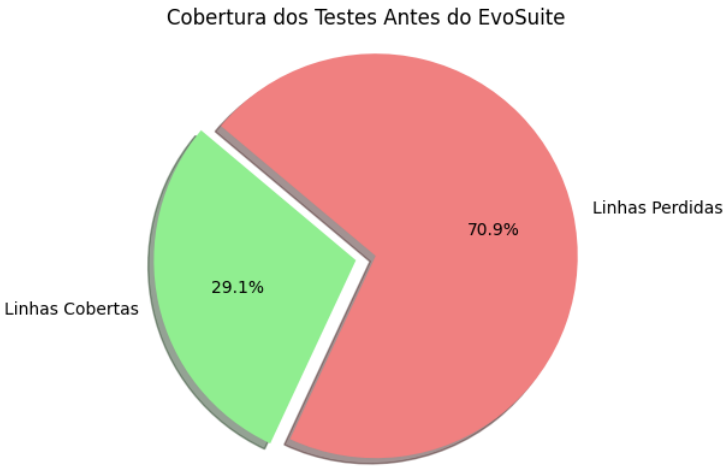

Figure 1. Cobertura antes da aplicação do EvoSuite.

entender a eficácia do EvoSuite em gerar testes que cobrem o código. Estando a cobertura média mais baixa do que o esperado; é necessário ajustar as configurações do EvoSuite para melhores resultados.

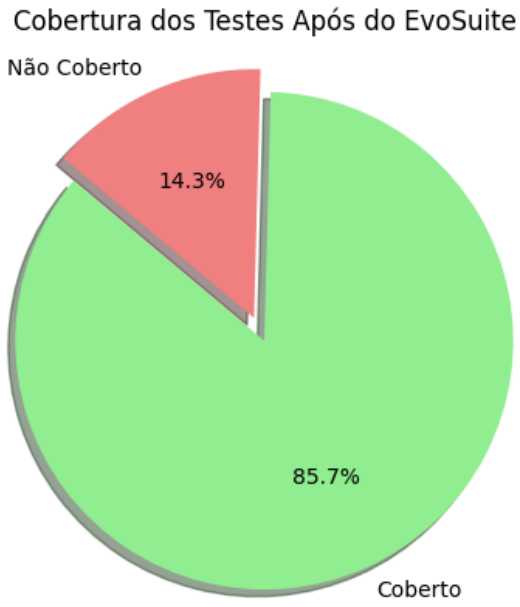

Figure 2. Cobertura antes da aplicação do EvoSuite.

Para uma comparação mais clara sobre como os testes gerados com o EvoSuite tiveram um melhor desempenho em questão de cobertura, o gráfico 3 mostra a cobertura em função das classes para os dois suítes de teste. Ele mostra como os testes gerados alcançaram valores de até 100% de cobertura para um número superior de classes. Outra análise relevante é que o número de classes com até 0% de cobertura diminuiu consideravelmente.

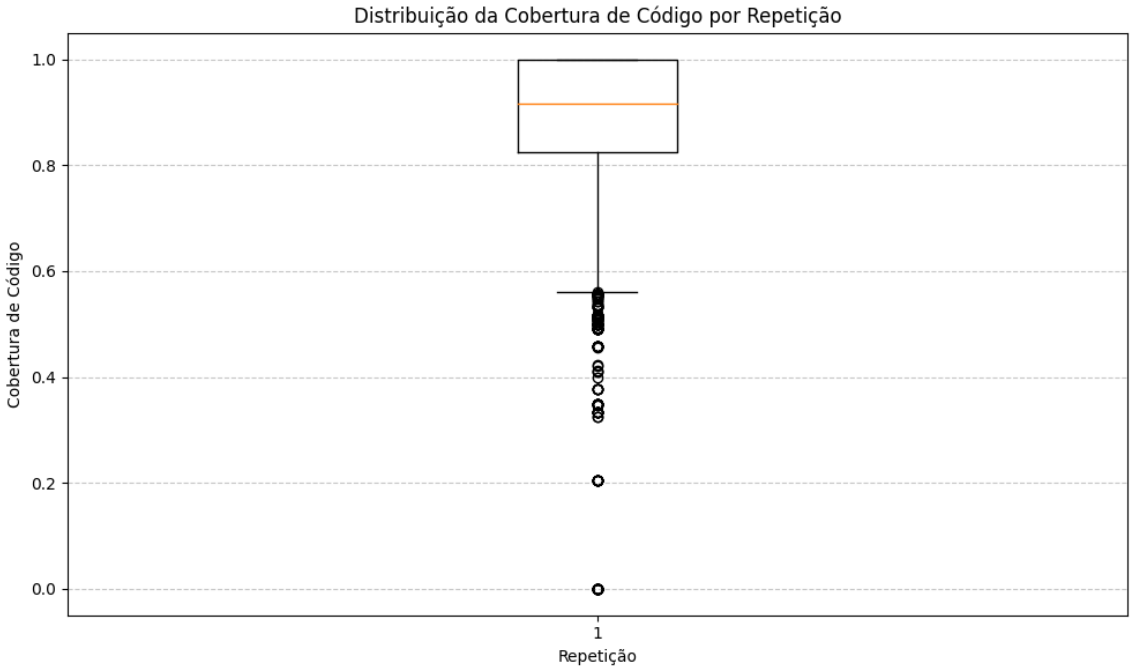

Figure 3. Cobertura por repetição

2.2.2. Testes de Mutação

Os resultados do PIT antes da geração de testes com o EvoSuite revelaram que, um teste não passou sem mutação ao calcular a cobertura de linha. Isso resultou em uma falha na cobertura de mutação do Maven. O erro indicou que a execução do objetivo default-cli falhou devido a um teste que não passou sem mutação ao calcular a cobertura de linha. Essa situação é problemática porque, para que o teste de mutação seja considerado bem-sucedido, todos os testes devem passar sem nenhuma mutação, formando uma ”suíte verde”. Devido a isso, o PIT não conseguiu gerar o arquivo com as estatísticas, o que impossibilitou a comparação que seria feita com os pontos de mutação dos testes gerados com o EvoSuite.

Embora seja uma situação indesejada, é importante ressaltar que a falha nos testes de mutação não invalida a utilidade dessa técnica. Pelo contrário, ela evidencia que os testes de mutação estão desempenhando seu papel ao identificar falhas nos testes unitários. O teste de mutação é uma ferramenta poderosa para avaliar a eficácia dos conjuntos de testes existentes, revelando lacunas na cobertura de teste ou fragilidades nos testes em capturar certos tipos de falhas.

Para além disso, o arquivo de estatísticas disponibilizado pelo EvoSuite, forneceu os dados referente aos pontos de mutação. O gráfico 5 mostra que a maioria dos testes tem uma pontuação mediana de cerca de 0,6, o que sugere uma eficácia razoável na detecção de mutações. No entanto, alguns testes têm pontuações mais baixas, indicando áreas de possível melhoria. O ideal é que tanto a mediana quanto os quartis estejam mais próximos de 1, indicando uma alta eficácia geral nos testes. Este gráfico ajuda a identificar áreas específicas para focar em melhorias nos testes.

Mas há ainda que se verificar a qualidade dos testes. Como métrica foi selecionado

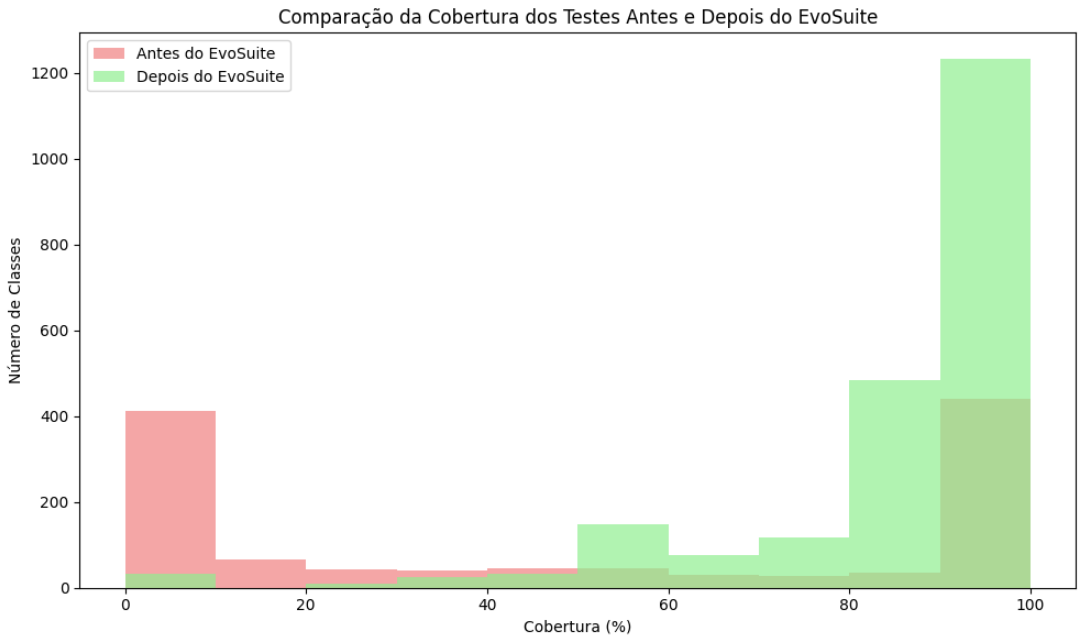

Figure 4. Comparação das coberturas

o tamanho dos testes gerados, pois testes menores, significam facilidade de manutenção, por serem mais fáceis de entender e manter. Eles também executam de forma mais rápida e facilitam até o diagnóstico do erro quando ocorre alguma falha.

O gráfico representado na figura 6 mostra que a maioria dos testes são curtos, com uma concentração notável de testes de comprimento menor. A linha vermelha mostra que, em média, os testes têm cerca de 41 caracteres de comprimento, enquanto a linha verde indica que metade dos testes tem 13 caracteres ou menos. Isso sugere que os testes são eficientes e focados, o que é ótimo para garantir uma boa cobertura de testes e identificar problemas no código de forma rápida. Em resumo, tem-se uma quantidade significativa de testes curtos e concentrados, o que é positivo para a eficácia geral dos seus testes de software.

De forma sucinta, os resultados indicaram que os testes gerados pelo EvoSuite alcançaram uma cobertura significativamente maior do que os testes originais disponíveis no projeto. Além disso, em uma análise quantitativa, observou-se que esses testes são geralmente curtos, o que sugere que são facilmente mantidos. Por fim, em relação à capacidade de identificar falhas no código, os testes alcançaram uma taxa de eliminação de ”mutantes” de 60%, demonstrando uma eficácia razoável nesse aspecto.

3. COLEMAN

Os testes unitários desempenham um papel crucial no ciclo de testes, avaliando a qualidade do software em nível de unidades do código, como funções e métodos. No entanto, é importante ressaltar que, por si só, os testes unitários não garantem a qualidade do software como um todo. A complexidade inerente aos softwares, juntamente com a necessidade de flexibilidade para acomodar alterações decorrentes de erros detectados ou demandas do cliente [Coutinho 2022], pode resultar na ocorrência de bugs no sistema. A prática da Integração Contínua no desenvolvimento de um software permite que ao

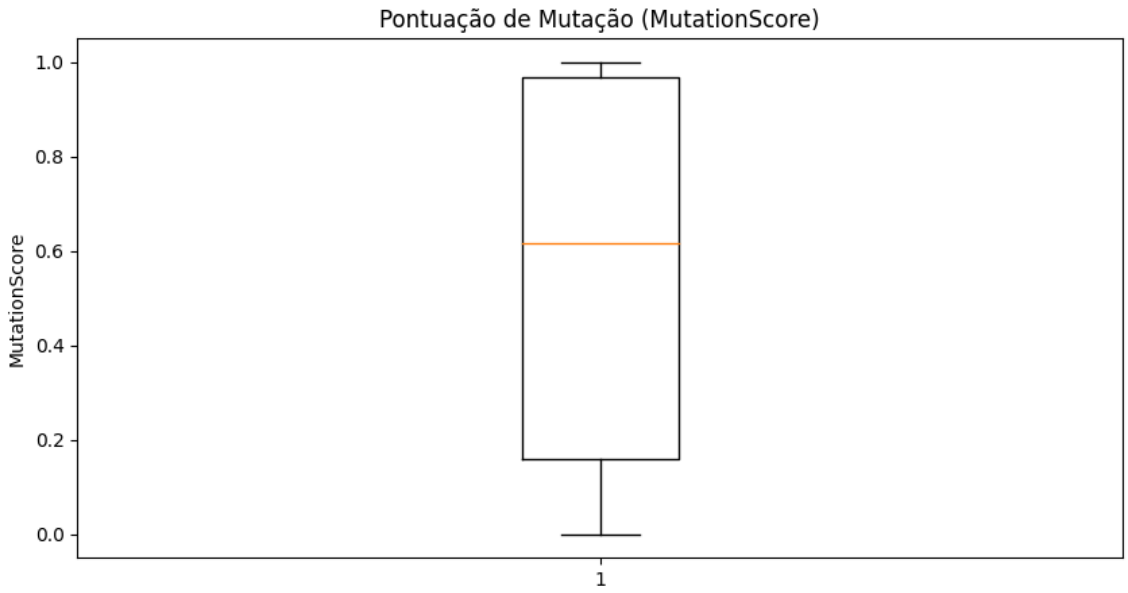

Figure 5. Pontuação de Mutação

alterar o código se tenha como retorno imediato a resposta se ele irá funcionar ou não [Pizzaia and MALARA 2022]. Os testes de regressão nessa conjuntura são essenciais para verificar o impacto de mudanças no código como um todo [Cavalcante 2020].

O teste de regressão envolve a repetição de um conjunto específico de testes realizados antes de alguma alteração no sistema, seja ela uma nova versão do software ou a integração de um novo módulo. Isso ocorre porque a inserção de novo código pode criar novos caminhos de fluxo de dados. Portanto, o teste de regressão é realizado para garantir que as mudanças não tenham causado impactos secundários indesejados [Alberto 2020]. A depender do tamanho do sistema, esse processo de reexecutar todos os testes pode ser custoso em questão de tempo e recursos.

Para lidar com este problema, foi desenvolvida a estratégia de Priorização de Casos de Testes (TCP). Ela trata-se do rearranjo dos testes de forma a focar em objetivos específicos, como a detecção rápida de falhas. Isso significa colocar os testes que identificam problemas no início da fila para que sejam executados primeiro. Com essa organização, os desenvolvedores podem realizar os testes mais cruciais, ou uma quantidade viável deles devido a limitações de recursos, sem perder eficácia na detecção de problemas [Cavalcante 2020].

No entanto, a volatilidade dos casos de teste em um ambiente de Integração Contínua, introduz o problema de equilibrar a diversidade do conjunto de testes com a quantidade de novos casos e aqueles propensos a erros ou com alta capacidade de detecção de falhas. A abordagem COLEMAN (Combinatorial VOlatiLE Multi-Armed BANDit) [Lima and Vergilio 2020] propõe utilizar um algoritmo baseado em Multi-Armed Bandit (MAB) para lidar com este problema. Isso permite otimizar a seleção de casos de teste de forma adaptativa, considerando a volatilidade do ambiente, sem depender excessivamente de informações estáticas como histórico de falhas ou cobertura de testes, que podem ser difíceis e demoradas de manter atualizadas. Isso torna a abordagem mais escalável e efi-

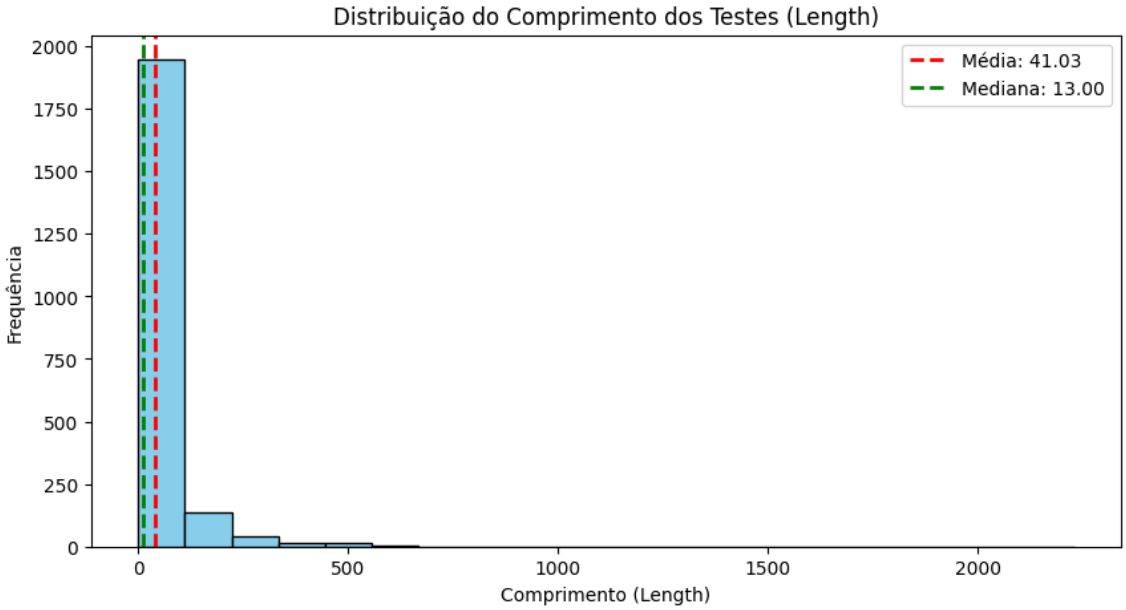

Figure 6. Comprimento dos Testes

caz, especialmente em ambientes de integração contínua (CI) com orçamentos limitados para testes [Lima and VERGILIO 2020].

Para uma melhor compreensão de como funciona o COLEMAN é preciso entender o problema dos Bandidos Multi-Armados (MAB). No contexto deste problema existe um agente que pode ser por exemplo: um algoritmo de aprendizado por reforço, que necessita em cada instante de tempo, escolher uma ação de um conjunto de ações possíveis e assim poder receber uma recompensa com base nessa ação. O objetivo do agente é aprender a melhor política de ação para acumular a maior recompensa possível, que neste caso é o aumento da precisão da classificação [Heskebeck et al. 2022]. Os algoritmos que tentam resolver o problema do MAB são chamados de politicas, estas por sua vez visam equilibrar a exploration (exploração) e a exploitation (exploração das ações). A exploration envolve tentar novas ações para descobrir suas recompensas, enquanto a exploitation se concentra em maximizar a recompensa com base no conhecimento atual.

3.1. Estudo de Caso - COLEMAN

A politica MAB escolhida foi a: Thompson Sampling (TS) [Thompson 1933], que trata-se de um algoritmo para o MAB com recompensas binárias, introduzido em 1933 por Thompson e também conhecido como probability matching. A política de TS é descrita como uma técnica que iguala a probabilidade de escolher uma ação à probabilidade dessa ação ser a melhor, [Heskebeck et al. 2022].

O limite teórico do arrependimento para o algoritmo de Thompson Sampling é logarítmico. Isso significa que, à medida que fazemos mais tentativas, o quanto nos arrependemos por não ter escolhido a melhor opção possível aumenta, mas não de forma muito rápida. Em outras palavras, o Thompson Sampling tende a minimizar a quantidade de arrependimento à medida que mais decisões são tomadas, o que é uma característica valiosa em situações onde é importante balancear a exploração de novas opções com a exploração das melhores opções conhecidas [Meidani et al. 2022]. O Thompson Sam-

pling é amplamente usado por sua capacidade de encontrar um equilíbrio entre explorar e aproveitar oportunidades, e tem sido eficaz em muitas situações práticas e teóricas. Ele é visto como uma alternativa aos métodos tradicionais, como Upper Confidence Bound (UCB), e pode melhorar a seleção de operadores em algoritmos evolutivos. Isso é particularmente útil em situações onde é necessário explorar continuamente as opções disponíveis, como em sistemas dinâmicos. O Thompson Sampling permite que diferentes abordagens sejam testadas e avaliadas com base no feedback recebido ao longo do tempo. Ele funciona atualizando um conjunto de dados com o feedback obtido e selecionando uma ação com base nesses dados. A ação selecionada é então comparada com a melhor ação possível, e o arrependimento é calculado com base nessa diferença. Essa abordagem dinâmica é útil em situações onde a melhor opção pode mudar ao longo do tempo.

Em um contexto de integração contínua, a política MAB Thompson Sampling é uma abordagem promissora pois possui um mecanismo adaptativo que equilibra habilmente a exploração de novos testes para aprendizado com a exploração de testes com desempenho conhecido para maximizar a qualidade do software entregue. Além disso, a TS modela a incerteza em torno das recompensas dos testes, permitindo uma seleção mais inteligente e informada. Sua capacidade de convergir rapidamente para uma política de priorização ótima facilita a detecção eficiente de falhas durante o processo de integração contínua. Além disso, a TS é relativamente simples de implementar e entender, tornando-se acessível mesmo para equipes com recursos limitados em termos de especialização em aprendizado de máquina ou otimização de políticas de priorização [Riquelme et al. 2018].

Ela foi escolhida como a política a ser aplicada pois, assim como a Fully Randomized Round Robin Multi-Armed Bandit (FRRMAB), política que obteve o melhor desempenho no experimento do COLEMAN [Lima and Vergilio 2020], o TS busca um equilíbrio entre exploração e exploitation, permitindo que o agente aprenda e adapte suas escolhas com base nas recompensas observadas. A natureza adaptativa do TS o torna particularmente eficaz em ambientes dinâmicos ou em situações em que as distribuições de recompensa das ações podem mudar ao longo do tempo.

Além disso, o TS oferece uma abordagem mais refinada para a seleção de ações, baseada em amostras de uma distribuição beta que é atualizada continuamente com base nas observações. Isso permite uma exploração mais inteligente das ações, levando em consideração tanto as recompensas médias quanto a incerteza associada a elas. Como resultado, o TS tende a se destacar em cenários mais complexos ou em problemas onde a eficiência na maximização da recompensa é crucial.

Em resumo, o Thompson Sampling deve seguir a seguinte fórmula para ser implementado:

- Atualização da Crença (Bayesiana): Para cada braço, mantemos uma distribuição de probabilidade sobre sua recompensa média e a atualização da crença ocorre sempre que um novo resultado é observado para um braço específico.
- Escolha do Braço: Para escolher um braço, amostramos aleatoriamente de cada distribuição de crença e o braço com a maior amostra é selecionado.
- Regret: O regret mede a diferença entre a recompensa cumulativa obtida pelo Thompson Sampling e a recompensa cumulativa ótima (se soubéssemos qual braço é o melhor). O objetivo é minimizar o regret ao longo do tempo.

3.1.1. Instalação e Configuração do Ambiente

Para Instalar o COLEMAN o primeiro passo necessário foi clonar seu repositório e após isso foram seguidas as instruções obtidas na Documentação oficial do COLEMAN.

3.2. Análise da Aplicação do COLEMAN

Foi implementada a política de Thompson Sampling utilizando o princípio geral da politica que é manter uma crença probabilística sobre a recompensa média de cada ação e, em seguida, escolher a ação com base em amostras aleatórias dessa distribuição de crença. A classe descrita a seguir mostra a implementação:

Listing 2. Implementação da Política de Thompson Sampling

```
class ThompsonSamplingPolicy(Policy):
    def __str__(self):
        return 'Thompson Sampling'

    def credit_assignment(self, agent: Agent):
        action_attempts = np.array(agent.actions['ActionAttempts
            '].tolist())
        value_estimates = np.array(agent.actions['ValueEstimates
            '].tolist())

        # Quality estimate: average of reward
        agent.actions['Q'] = value_estimates / action_attempts

        # Remove negative estimates and NaN (if any)
        value_estimates = np.maximum(value_estimates, 0)
        value_estimates[np.isnan(value_estimates)] = 0

        # Make sure action_attempts is always positive
        action_attempts = np.maximum(action_attempts, 1)

        # Thompson Sampling
        samples = []
        for s, f in zip(value_estimates, action_attempts):
            if f - s + 1 <= 0:
                # Handle the case where f - s + 1 is not
                    positive
                b = 1  # Set b to a default positive value
            else:
                b = f - s + 1
            samples.append(np.random.beta(s + 1, b))

        # Pick the arm with the highest sampled estimate
        best_arm = np.argmax(samples)

        # Play with the best arm
        actual_prob = np.random.uniform()
```

```
        if actual_prob < agent.actions['ValueEstimates'][
           best_arm] / agent.actions['ActionAttempts'][best_arm
           ]:
            # If we win with this arm
            agent.actions['ActionAttempts'][best_arm] += 1
            agent.actions['ValueEstimates'][best_arm] += 1
        else:
            # If we lose with this arm
            agent.actions['ActionAttempts'][best_arm] += 1
```

O método `credit_assignment(self, agent: Agent)` é responsável por aplicar a política Multi-Armed Bandit (MAB). Ele realiza as seguintes etapas:

- Atribuição de Crédito:
  - Atualiza as estimativas de valor do agente com base nas recompensas observadas.
- Obtenção de Dados:
  - Obtém os dados do agente relacionados às tentativas de ação e às estimativas de valor.
- Cálculo das Estimativas de Qualidade (Q):
  - Calcula as estimativas de qualidade (Q) para cada ação, que é a média das recompensas observadas até o momento.
- Remoção de Valores Negativos e NaNs:
  - Remove valores negativos e valores NaN (se houver) das estimativas de valor.
- Garantia de que o Número de Tentativas de Ação seja Positivo:
  - Garante que o número de tentativas de ação seja sempre positivo, substituindo qualquer valor negativo por 1.

O que a politica MAB Thompson Sampling faz é calcular uma amostra de Thompson, que é uma amostra de uma distribuição Beta com base nas estimativas de valor e no número de tentativas de ação. Isso reflete a incerteza sobre as estimativas de valor. Nesse processo é selecionado o braço com a maior estimativa amostral de acordo com a politica Thompson Sampling. Além de simular a execução da ação selecionada e atualizar as tentativas de ação e as estimativas de valor do agente com base no resultado observado.

Em resumo, este algoritmo implementa a política de Thompson Sampling para selecionar a próxima ação de um agente em um problema de multi-armed bandit, atualizando dinamicamente as estimativas de valor com base nas recompensas observadas. Isso permite que o agente explore diferentes ações enquanto explora aquelas que parecem ser mais promissoras com base nas informações disponíveis.

3.3. Análise da Aplicação do COLEMAN

Para a implementação do COLEMAN, optou-se pela utilização dos seguintes datasets: i) Druida - Um sistema de banco de dados projetado para análises instantâneas, capaz de realizar consultas em frações de segundo em vastos volumes de dados contínuos e acumulados. ii) Fastjson - Esta é uma ferramenta Java destinada à conversão bidirecional entre objetos Java e sua forma JSON. iii) LexisNexis - Fornece uma coleção extensa de

dados para análise e pesquisa, bem como para uso em sistemas de inteligência artificial, disponibilizados pelo serviço Nexis Data as a Service (DaaS). iv) Deeplearning4j (DL4J) - Representa uma biblioteca de aprendizado de máquina que facilita a construção de redes neurais de forma simplificada, utilizando recursos avançados e eficientes.

3.3.1. Distribuição Normalizada da Redução de Tempo

O gráfico da Figura 7 representam a Distribuição Normalizada da Redução de Tempo em um contexto de políticas MAB para priorização de testes em integração contínua. Ao focar na estratégia Thompson Sampling e compará-la com outras, observamos que apresenta uma distribuição mais consistente de redução de tempo, especialmente notável em orçamentos mais generosos, como 80%. Sua variabilidade de resultados é relativamente baixa em comparação com estratégias como Random e E-greedy. No entanto, Thompson Sampling pode não ser tão eficaz em orçamentos mais restritos, como 10%, e pode não ser ideal para cenários com severas restrições de tempo ou recursos. Concluindo, enquanto Thompson Sampling se mostra promissora para a priorização de testes em integração contínua, sua eficácia depende do contexto e das restrições específicas de tempo. Recomenda-se uma avaliação cuidadosa dos trade-offs entre eficiência e recursos ao escolher uma estratégia de MAB para tomada de decisões.

Figure 7. Gráficos sobre a Distribuição Normalizada da Redução de Tempo do dataset 'druid'.

Os gráficos da Figura 8 ilustram a distribuição da redução do tempo normalizado para diversas políticas de Multi-Armed Bandit (MAB) utilizadas na priorização de testes em integração contínua, com base em um conjunto de dados específico. Ao focar na estratégia Thompson Sampling e compará-la com outras políticas apresentadas no gráfico, observamos alguns padrões interessantes. Em primeiro lugar, nota-se que a Thompson Sampling mantém uma performance consistente em diferentes orçamentos de tempo (10%, 50% e 80%), o que sugere uma adaptabilidade notável às variações nas condições. No entanto, em termos de eficiência na redução do tempo, outras estratégias como FRRMAB se destacam, especialmente em orçamentos mais apertados. Além disso, embora a Thompson Sampling apresente consistência, também é evidente uma certa variação nos resultados, principalmente em cenários de orçamento de 10%, indicando alguma incerteza ou inconsistência na sua aplicação prática. Em suma, a Thompson Sampling emerge como uma escolha sólida e confiável, mas talvez não seja a mais eficiente em todas as situações, sugerindo a necessidade de uma avaliação cuidadosa dos trade-offs entre consistência e eficiência ao selecionar uma estratégia de MAB para a tomada de decisões em integração contínua.

Figure 8. Gráficos sobre a Distribuição Normalizada da Redução de Tempo do dataset 'fatjson'.

Com base nos gráficos da Figura 9, foi possível realizar uma análise comparativa da estratégia Thompson Sampling em relação a outras estratégias (FRMAB, Greedy, Random e -greedy) no contexto da priorização de testes em integração contínua. Os pontos positivos da Thompson Sampling incluem sua consistência, evidenciada pela distribuição mais estável da Redução de Tempo Normalizado em todos os três orçamentos de tempo, e sua eficiência, destacada pela alta mediana da redução do tempo normalizado nos orçamentos de 50% e 80%. No entanto, alguns pontos negativos também foram identificados, como seu desempenho inferior em relação à estratégia FRMAB no orçamento de tempo de 10%, o que pode ser problemático em situações de tempo extremamente limitado, e a possibilidade de menos exploratividade devido à sua distribuição compacta, o que pode ser uma desvantagem em cenários onde a exploração é crucial. Em suma, a Thompson Sampling demonstra consistência e eficiência, mas pode ser menos exploratória em comparação com outras estratégias. A escolha entre essas abordagens dependerá das prioridades específicas do contexto de integração contínua e dos recursos disponíveis.

Figure 9. Gráficos sobre a Distribuição Normalizada da Redução de Tempo do dataset 'lexisNexis'.

Observou-se, na Figura 10, que a Thompson Sampling demonstra eficiência em orçamentos maiores (80%), aproximando-se das outras estratégias em termos de redução do tempo normalizado nesses cenários. Além disso, apresenta menor variabilidade nos resultados, evidenciada pelas barras de erro menores em comparação com outras estratégias. No entanto, foram identificados alguns pontos negativos da Thompson Sampling, como sua ineficácia em orçamentos menores, onde não alcança o mesmo nível de eficácia na redução do tempo normalizado em comparação com outras estratégias. Além disso, em situações críticas de tempo, onde a rapidez e eficiência são cruciais, a Thompson Sam-

pling pode não ser a melhor escolha. Vale ressaltar que essa análise se baseia apenas no gráfico e nos dados apresentados, sendo importante considerar outros fatores antes de tomar uma decisão final sobre a escolha da estratégia mais adequada para um determinado contexto de integração contínua.

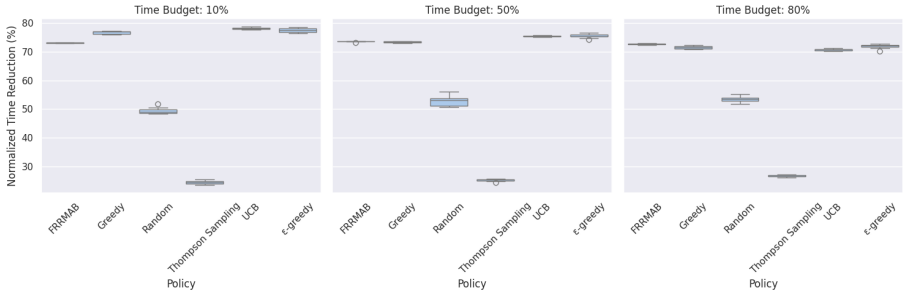

Figure 10. Gráficos sobre a Distribuição Normalizada da Redução de Tempo do dataset 'deeplearning4j'.

3.3.2. Priorização da Distribuição de Tempo

Os pontos positivos da Thompson Sampling, na Figura 11 incluem sua eficiência de tempo, evidenciada pelo tempo médio de priorização consistentemente mais baixo em todos os três orçamentos de tempo, indicando uma maior rapidez na priorização dos testes. Além disso, sua consistência na performance, refletida pela baixa variação nos resultados, sugere uma estabilidade na estratégia. No entanto, alguns pontos negativos também foram identificados, como a incerteza quanto à qualidade ou precisão da priorização realizada pela Thompson Sampling, uma vez que ser rápido não garante necessariamente precisão ou eficácia. Outra questão observada são os outliers nos dados da Thompson Sampling, especialmente no orçamento de tempo de 10%, o que pode indicar instabilidade ocasional ou resultados anômalos. Vale ressaltar que essa análise é baseada apenas no gráfico fornecido e não considera outros fatores além do tempo de priorização. Portanto, enquanto a Thompson Sampling parece ser uma boa opção em termos de eficiência, é importante também avaliar a qualidade das decisões tomadas por essa estratégia antes de adotá-la em um contexto específico de integração contínua.

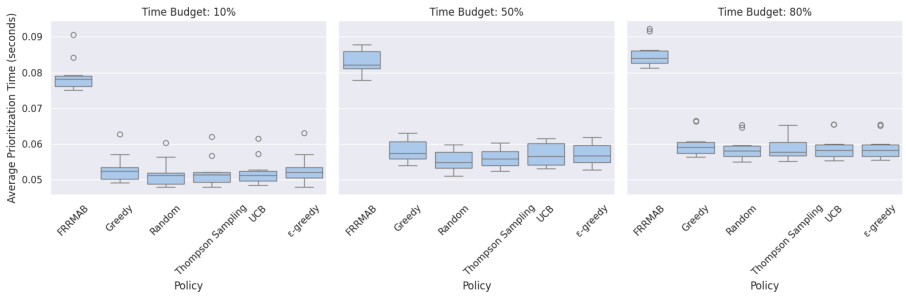

Figure 11. Gráficos sobre a Priorização da Distribuição de Tempo do dataset 'druid'.

Na Figura 12, podemos observar que os resultados do Thompson Sampling incluem sua consistente eficiência, mantendo um tempo de priorização baixo em difer-

entes orçamentos de tempo, sugerindo que é uma opção robusta para diversos cenários. No entanto, alguns pontos negativos também foram observados, como a falta de uma diferença significativa em sua performance em comparação com a estratégia UCB quando o orçamento de tempo é aumentado para 80%, e a ausência de um destaque drástico em relação às outras políticas, indicando que suas vantagens podem ser mais notáveis em situações específicas ou sob certas condições. O gráfico compara o ”Average Prioritization Time” entre cinco políticas diferentes (FRAMAB, Greedy, Random, Thompson Sampling e UCB) sob três ”Time Budgets” diferentes, utilizando box plots para representar a variação do tempo médio de priorização. Essas observações fornecem insights sobre o desempenho relativo das estratégias em diferentes contextos de orçamento de tempo. No entanto, é importante ressaltar que essa análise é uma aproximação e não uma medida cientificamente precisa.

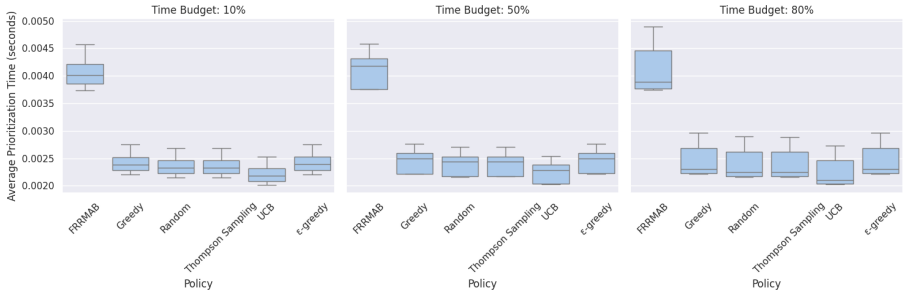

Figure 12. Gráficos sobre a Priorização da Distribuição de Tempo do dataset ’deeplearning4j’.

Examinando a Figura 13, vemos que a politica Thompson Sampling demonstrou eficiência, com um tempo médio de priorização consistentemente baixo em todos os três orçamentos de tempo e consistência, mantendo tempos de priorização baixos independentemente do orçamento de tempo. No entanto, alguns pontos negativos também foram observados, como o fato de não ser a estratégia mais rápida em todos os cenários, sendo superada pelo FRMAB em um orçamento de tempo de 80%, e a maior variação nos resultados, especialmente evidente no orçamento de tempo de 10%, indicando possível inconsistência na performance quando o tempo disponível é limitado. Quanto às outras estratégias (FRMAB, Greedy, Random, UCB e -Greedy), o gráfico revela que têm tempos médios de priorização variados em diferentes orçamentos de tempo, com o FRMAB superando todas as outras políticas ao mostrar o menor tempo médio no orçamento de tempo de 80%. Em resumo, enquanto a Thompson Sampling é eficiente e consistente, ela não é necessariamente a mais rápida em todos os cenários.

Com base na análise da Figura 14, a estratégia Thompson Sampling revela pontos positivos e negativos em comparação com outras estratégias. Entre os aspectos positivos, destaca-se sua consistência, com um tempo médio de priorização mais estável em comparação com outras estratégias, especialmente em orçamentos de tempo de 50% e 80%. Além disso, a Thompson Sampling oferece um equilíbrio entre eficiência e consistência, mesmo que não seja a estratégia mais rápida em todos os cenários. No entanto, em orçamentos de tempo mais baixos, como 10%, a Thompson Sampling não se destaca tanto quanto outras estratégias, como Greedy ou FRAMAB. Ademais, se o objetivo for minimizar estritamente o tempo de priorização, a Thompson Sampling pode não ser a

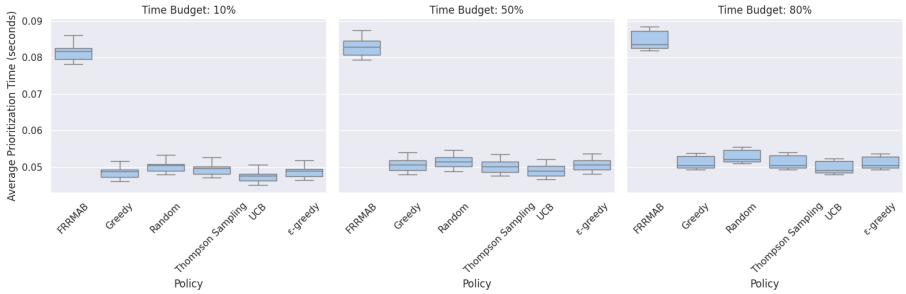

Figure 13. Gráficos sobre a Priorização da Distribuição de Tempo do dataset 'fatjson'.

melhor opção. As outras estratégias apresentam desempenhos diversos: FRAMAB é consistente e eficiente, Greedy é a mais rápida, mas perde consistência com orçamentos de tempo maiores, Random não se destaca em nenhum orçamento específico, enquanto UCB e -Greedy têm desempenho variável e menor consistência em comparação com Thompson Sampling. Em suma, a Thompson Sampling é uma escolha sólida para equilibrar eficiência e consistência na priorização, mas é crucial considerar o contexto e os objetivos específicos ao escolher uma estratégia de priorização.

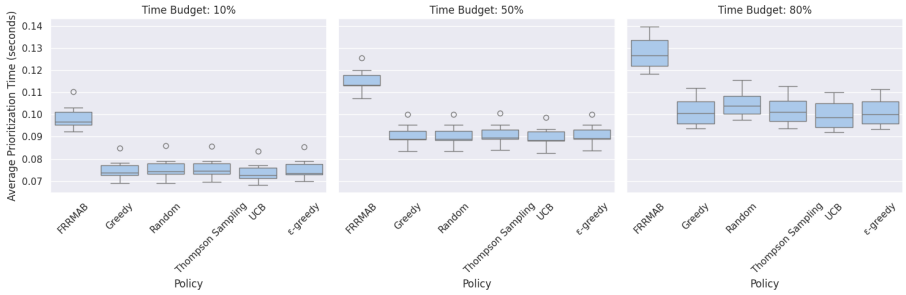

Figure 14. Gráficos sobre a Priorização da Distribuição de Tempo do dataset 'lexisNexis'.

3.3.3. Distribuição NAPFD usando avg_fitness_time

Com base na Figura 15, observamos o desempenho das diferentes estratégias de priorização de testes em integração contínua, incluindo a Thompson Sampling, e analisamos seus pontos positivos e negativos. Entre os pontos positivos da Thompson Sampling, destaca-se seu tempo médio de priorização mais baixo em relação às estratégias FRAMAB e -greedy no espaço de tempo de 10%, indicando eficiência. Além disso, sua variação nos resultados é relativamente baixa no orçamento de tempo de 50%, sugerindo consistência. No entanto, alguns pontos negativos foram observados, como o desempenho inferior nos orçamentos de tempo de 50% e 80% em comparação com outras estratégias como FRAMAB e Greedy, e uma maior variação nos resultados nos orçamentos de tempo de 10% e 80%, indicando possíveis inconsistências. Cada uma das outras estratégias (FRAMAB, Greedy, Random, UCB e -greedy) possui seu próprio perfil de desempenho, com vantagens e desvantagens específicas. Por exemplo, a FRAMAB pode ser mais robusta em determinados cenários, enquanto a Greedy pode ser mais rápida, mas menos

exploratória. Em resumo, a Thompson Sampling parece ser uma escolha eficiente em termos de tempo em orçamentos menores, mas pode não ser a melhor opção quando o tempo é mais abundante. A escolha da estratégia ideal depende das prioridades específicas do contexto e dos trade-offs entre eficiência e exploração.

Figure 15. Gráficos sobre a Distribuição NAPFD do dataset 'druid'.

Com base na Figura 16, que representa a Distribuição NAPFD no dataset 'deeplearning4j', podemos analisar a estratégia Thompson Sampling em comparação com outras estratégias apresentadas. Entre os pontos positivos da Thompson Sampling, destaca-se seu desempenho superior ao Random e comparável ao Greedy no orçamento de tempo de 10%, enquanto supera todas as outras estratégias no orçamento de tempo de 50%, indicando eficiência em cenários de médio prazo. No entanto, seu desempenho é inferior ao FRAMAB e Greedy no orçamento de tempo de 80%, sugerindo que pode não ser a melhor opção para prazos mais longos ou recursos mais amplos. As outras estratégias (FRAMAB, Greedy, Random e UCB) demonstram seu próprio desempenho em diferentes orçamentos de tempo, com FRAMAB e Greedy parecendo mais eficazes em cenários de maior orçamento de tempo, enquanto Random é menos consistente e UCB está em algum lugar intermediário. Em suma, a Thompson Sampling mostra-se uma estratégia promissora, especialmente em cenários de médio prazo, mas sua eficácia pode variar dependendo do contexto e do orçamento de tempo disponível.

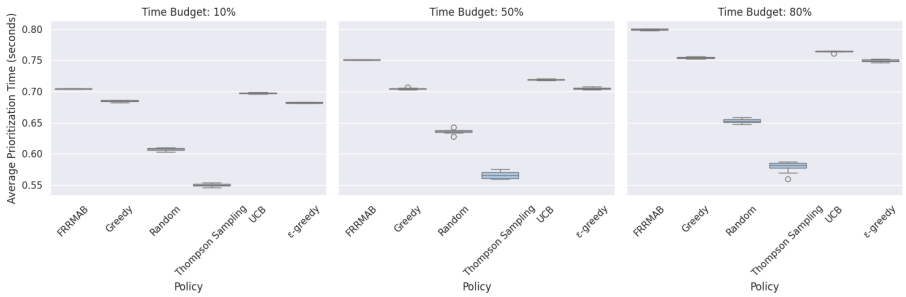

Figure 16. Gráficos sobre a Distribuição NAPFD do dataset 'deeplearning4j'.

Com base na análise da Figura 17, entre seus pontos positivos, destaca-se seu desempenho consistente em diferentes orçamentos de tempo, com um tempo de priorização mais baixo do que o FRAMAB e o Random em todos os cenários. Além disso, mantém-se relativamente estável à medida que o orçamento de tempo aumenta. No entanto, alguns pontos negativos foram observados, como ser superado pelo Greedy e -greedy em todos os cenários de orçamento de tempo e não mostrar uma melhoria significativa à medida

que o orçamento de tempo aumenta, ao contrário do Greedy e -greedy. Entre as outras estratégias, o Greedy tende a ter o menor tempo médio de priorização em todos os três cenários, seguido por -greedy, que mostra um desempenho superior em todos os cenários de orçamento de tempo. FRAMAB e Random também são superados pela Thompson Sampling e pelo Greedy. No entanto, o UCB não é tão eficaz quanto a Thompson Sampling e o Greedy. Em resumo, embora a Thompson Sampling seja uma estratégia promissora, não é a melhor em todos os cenários, com Greedy e -greedy ainda sendo opções competitivas para a priorização de testes em integração contínua.

Figure 17. Gráficos sobre a Distribuição NAPFD do dataset 'fatjson'.

Com base na Figura 18, a análise da politica Thompson Sampling revela que sua variação nos resultados é menor em comparação com outras estratégias, indicando uma maior estabilidade. No entanto, alguns pontos negativos foram observados, como o fato de que, embora consistente, a Thompson Sampling não é a estratégia mais rápida. Especialmente quando o orçamento de tempo é aumentado para 50% e 80%, outras estratégias como Random e UCB superam a Thompson Sampling em termos de tempo médio de priorização. Em resumo, a Thompson Sampling é uma escolha sólida para integração contínua, oferecendo consistência e estabilidade. No entanto, é importante considerar o equilíbrio entre tempo e precisão ao selecionar a estratégia ideal para o contexto específico.

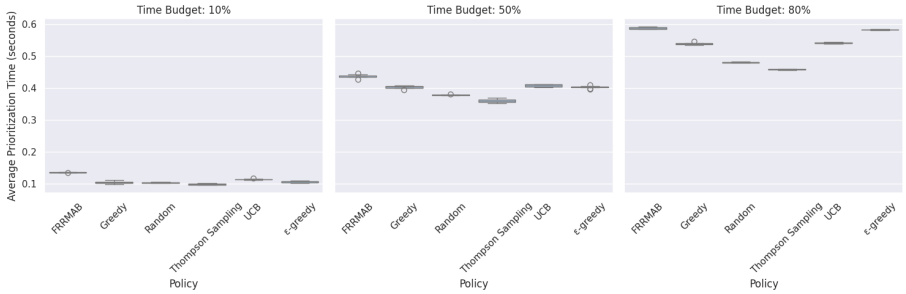

Figure 18. Gráficos sobre a Distribuição NAPFD do dataset 'lexisNexis'.

3.3.4. Distribuição APFDc - usando avg_cost_time

Segundo a Figura 19 a política Thompson Sampling, destaca-se sua superioridade sobre as estratégias Greedy e Random no espaço de tempo de 10%, e um desempenho significativamente melhor do que todas as outras estratégias no espaço de tempo de 50%.

No entanto, seu desempenho cai drasticamente no orçamento de tempo de 80%, sendo a pior entre todas as estratégias, e a variabilidade nos resultados sugere que pode não ser a estratégia mais confiável em todos os contextos ou níveis de orçamento de tempo. A estratégia FRAMAB apresenta um desempenho consistente em todos os orçamentos de tempo, mas sem se destacar em nenhum deles, enquanto o Greedy tem um desempenho mediano em todos os orçamentos de tempo, sem grandes variações. O Random mostra resultados aleatórios, sem uma tendência clara de melhoria com o aumento do orçamento de tempo, e o UCB tem um desempenho razoável, mas não tão bom quanto o Thompson Sampling no orçamento de tempo de 50%. Em resumo, o Thompson Sampling é uma estratégia promissora para a prioritização de testes em integração contínua, especialmente em cenários com orçamentos de tempo limitados. No entanto, sua eficácia pode variar dependendo do contexto específico e das condições de orçamento de tempo.

Figure 19. Gráficos sobre a Distribuição APFDc do dataset 'druid'.

Observou-se na Figura 20 que, com um orçamento de tempo de 10%, o Thompson Sampling apresenta um desempenho comparável ao Random e superior ao Greedy e -Greedy. No entanto, ao considerar um orçamento de tempo de 50%, o Thompson Sampling supera todas as outras estratégias, destacando sua eficácia na prioritização dos testes. Por outro lado, com um orçamento de tempo de 80%, o desempenho do Thompson Sampling cai drasticamente, tornando-se a pior entre as estratégias apresentadas. A inconsistência nos resultados pode representar um fator limitante para sua aplicação em diferentes contextos. Essas observações ressaltam a importância de considerar o contexto específico e as características de cada estratégia ao priorizar testes em integração contínua.

Figure 20. Gráficos sobre a Distribuição APFDc do dataset 'deeplearning4j'.

Com base na Figura 21 nos espaços de tempo de 10% e 50%, a Thompson Sampling não se destaca como a estratégia mais eficaz, ficando atrás do FRAMAB e do Greedy. No entanto, é importante notar que, no orçamento de tempo de 80%, a Thompson Sampling supera todas as outras estratégias em termos de APFDc. A variação nos

resultados sugere uma certa inconsistência na eficácia da estratégia em diferentes contextos ou configurações, o que destaca a necessidade de considerar cuidadosamente os prós e contras ao aplicar a Thompson Sampling em um contexto específico. Essa análise, entretanto, é baseada exclusivamente no gráfico e não leva em conta outros fatores externos que podem influenciar a escolha da estratégia mais apropriada.

Figure 21. Gráficos sobre a Distribuição APFDc do dataset 'fatjson'.

Analisando a Figura 22 a política Thompson Sampling mantém uma priorização média estável em diferentes orçamentos de tempo, superando as estratégias Greedy e Random em todos os cenários apresentados. Além disso, sua capacidade de exploração adaptativa permite uma rápida correção e adaptação às mudanças no ambiente. No entanto, é importante notar que a Thompson Sampling pode ser computacionalmente custosa devido à necessidade de manter e amostrar de uma distribuição posterior completa sobre modelos. Além disso, em orçamentos de tempo maiores (50% e 80%), estratégias como UCB e -greedy podem ter um desempenho ligeiramente melhor. Em suma, enquanto a Thompson Sampling se destaca em estabilidade e superioridade em certos cenários, sua eficiência computacional e desempenho relativo em comparação com outras estratégias devem ser considerados ao decidir sua aplicação.

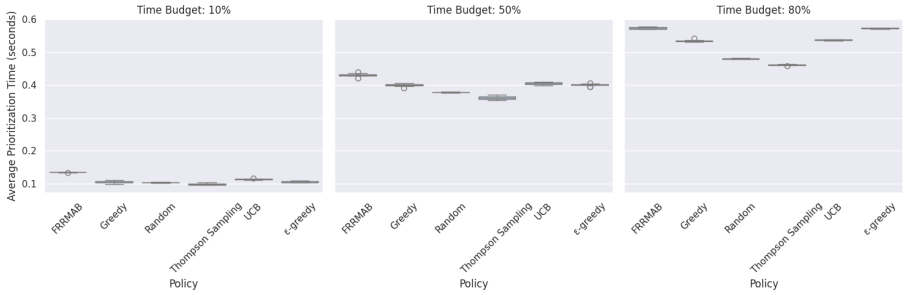

Figure 22. Gráficos sobre a Distribuição APFDc do dataset 'lexisNexis'.

3.3.5. Tabela - Priorização de Tempo

Com base na Figura 23 concluimos que os tempos de priorização médios da Thompson Sampling são comparáveis aos das outras estratégias em todos os três espaços de tempo. Isso sugere que a Thompson Sampling pode estar explorando mais opções para encontrar a melhor política de MAB para priorização de testes, tornando-a potencialmente mais eficaz em cenários complexos onde a exploração é crucial. No entanto, como ponto negativo, a Thompson Sampling leva mais tempo para priorizar os testes em comparação com

TimeBudget	Random	UCB	Greedy	ε-greedy	FRRMAB	Thompson Sampling
0.1	0.0460	0.0452	0.0461	0.0461	0.0681	0.0461
0.5	0.0507	0.0507	0.0516	0.0512	0.0732	0.0508
0.8	0.0552	0.0536	0.0547	0.0542	0.0769	0.0543

Figure 23. Tabela Priorização de Tempo.

outras estratégias, o que pode não ser ideal quando o tempo é uma restrição crítica. Em resumo, embora a Thompson Sampling apresente tempos de priorização semelhantes, suas vantagens em termos de exploração e eficácia em cenários desafiadores podem torná-la uma escolha valiosa em determinados contextos.

3.3.6. Tabela - NAPFD

Com base nos dados apresentados na Figura 24, podemos observar que o Thompson Sampling oferece algumas vantagens significativas. Em comparação com outras estratégias como Random, UCB, Greedy e -greedy, o Thompson Sampling demonstra tempos médios de aptidão (avg_fitness_time) comparáveis em todos os três orçamentos de tempo analisados: 0.1, 0.5 e 0.8. Essa estratégia se destaca por ser uma abordagem baseada em probabilidades que equilibra efetivamente a exploração e a exploração de ações, adaptando suas crenças de forma adaptativa com base em observações anteriores. Além disso, é eficaz em cenários onde as recompensas são incertas e mudam ao longo do tempo. No entanto, o Thompson Sampling pode apresentar algumas limitações, como a potencialmente custosa amostragem de parâmetros da distribuição posterior e a dependência da qualidade das estimativas iniciais e da distribuição priori. Por outro lado, outras estratégias como Random, UCB, Greedy e -greedy têm suas próprias características e podem ser mais adequadas dependendo do contexto específico e dos objetivos do problema. Em suma, embora o Thompson Sampling ofereça benefícios importantes, a escolha da estratégia mais apropriada deve ser feita considerando cuidadosamente o cenário em questão.

TimeBudget	Random	UCB	Greedy	ε-greedy	FRRMAB	Thompson Sampling	Metric
0.1	0.5789	0.6281	0.6109	0.6135	0.6574	0.5642	avg_fitness_time
0.5	0.6859	0.7542	0.7414	0.7564	0.7832	0.6662	avg_fitness_time
0.8	0.7251	0.7980	0.7972	0.8104	0.8370	0.6932	avg_fitness_time

Figure 24. Tabela NAPFD.

3.3.7. Tabela - APFDc

A análise dos dados da Figura 25 revela que a política Thompson Sampling em comparação com as outras, demonstra um desempenho superior em termos de tempo médio de custo, especialmente em cenários com maiores valores de TimeBudget (0.5 e 0.8). Sua abordagem adaptativa, levando em conta a incerteza sobre as recompensas das ações, mostra-se vantajosa em ambientes dinâmicos. No entanto, a Thompson Sampling pode ser mais intensiva computacionalmente devido à necessidade de amostragem de distribuições, e sua eficácia depende da qualidade das estimativas das distribuições de recompensa. Em resumo, embora promissora, a Thompson Sampling requer uma

TimeBudget	Random	UCB	Greedy	ε-greedy	FRRMAB	Thompson Sampling	Métrica
0.1	0.5794	0.6488	0.6354	0.6355	0.6821	0.5556	avg_cost_time
0.5	0.6881	0.7824	0.7703	0.7805	0.8021	0.6611	avg_cost_time
0.8	0.7265	0.8158	0.8179	0.8251	0.8467	0.6834	avg_cost_time

Figure 25. Tabela APFDc

consideração cuidadosa dos trade-offs e da complexidade computacional ao ser implementada.

Em resumo, com base na análise dos resultados, as políticas de priorização de testes em integração contínua: Thompson Sampling demonstra eficiência ao apresentar tempos médios de priorização mais baixos em comparação com políticas como FRAMAB e -greedy em orçamentos de tempo mais restritos, indicando sua capacidade de tomar decisões rápidas. Além disso, sua variabilidade nos resultados é relativamente baixa em certos contextos, sugerindo consistência.

Entretanto, alguns aspectos negativos também são observados. Em comparação com outras estratégias, como FRAMAB e Greedy, a Thompson Sampling mostra desempenho inferior em orçamentos de tempo mais amplos, indicando possível falta de eficiência em cenários onde o tempo é mais abundante. Além disso, há uma maior variabilidade nos resultados em certos cenários de tempo, sugerindo possíveis inconsistências na performance.

Cada política tem seu próprio perfil de desempenho, com vantagens e desvantagens específicas. Por exemplo, enquanto FRAMAB pode ser mais robusta em determinados cenários, Greedy pode ser mais rápida, porém menos exploratória. Em suma, a Thompson Sampling parece ser uma escolha eficiente em termos de tempo em orçamentos menores, mas pode não ser a melhor opção quando o tempo é mais abundante. A escolha da estratégia ideal depende das prioridades específicas do contexto e dos trade-offs entre eficiência e explorabilidade.

3.3.8. NAPFD

Os gráficos a seguir apresentam o resultado utilizando a métrica NAPFD *Average Percentage of Faults Detected cumulative* (Média Percentual Acumulada de Falhas Detectadas) com os algoritmos *cum_fitness* e *fitness_variation*. Os três gráficos mostram o desempenho de diferentes algoritmos (*FRRMAB*, *Greedy*, *Random*, *Thompson Sampling* e ε-greedy) em três cenários de tempo (10%, 50% e 80% do tempo total). O eixo Y é o desempenho acumulado (*NAPFD*) e o eixo X é o ciclo CI. Todos os algoritmos têm melhor desempenho com mais tempo disponível. Em resumo, quanto mais tempo disponível, melhor o desempenho dos algoritmos.

3.3.8.1. NAPFD com algoritmo cum_fitness Na Figura 26, ao analisar os dados usando o algoritmo *fitness_variation* com o conjunto de dados *druid*, podemos ver como diferentes estratégias de amostragem se comportam em diferentes orçamentos de tempo (10%, 50% e 80%). Entre essas estratégias, o Thompson Sampling se destaca com um desempenho notável. Comparado com Greedy, Random, UCB e ϵ-greedy, o Thompson

Sampling mostra resultados relativamente melhores. No entanto, é o método FRRMAB que consistentemente supera todas as outras estratégias.

Em detalhes, o Thompson Sampling supera Random e ϵ-greedy, mas fica abaixo do UCB quando o orçamento de tempo é de 10%. Contudo, com orçamentos de tempo de 50% e 80%, o Thompson Sampling supera o UCB em desempenho, indicando sua eficácia em cenários de teste com mais recursos disponíveis.

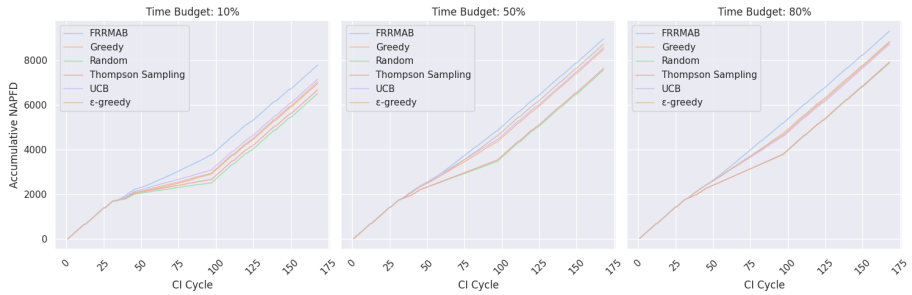

Figure 26. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'druid'

Na Figura 27, os resultados da análise de dados utilizando o algoritmo

fitness_variation com o conjunto de dados 'deeplearning4j' são apresentados. Observou-se que, dentre as estratégias avaliadas, FRRMAB demonstrou consistentemente o melhor desempenho, seguida pelo Greedy. As estratégias UCB e ϵ-greedy exibiram padrões semelhantes, com ϵ-greedy mostrando um desempenho ligeiramente superior. Ambas as estratégias superaram a abordagem aleatória, mas ficaram abaixo de outras estratégias em termos de eficácia na identificação da melhor ação.

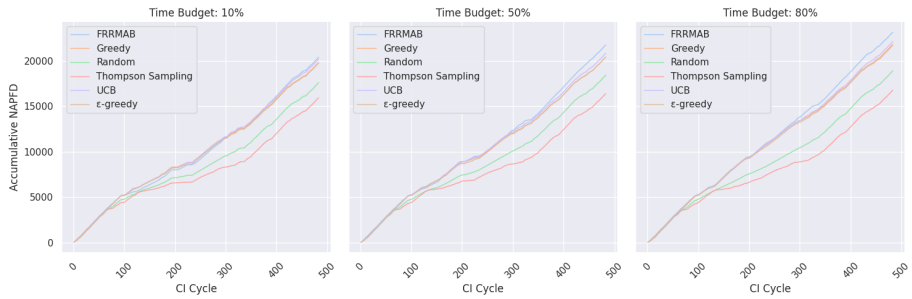

Figure 27. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'deeplearning4j'.

Na Figura 28, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'fatjson', foram observadas tendências significativas em relação ao desempenho das estratégias de seleção. Para 10% do Orçamento de Tempo, FRRMAB destacou-se como a estratégia mais eficaz, seguida por Greedy, UCB, Thompson Sampling, ϵ-greedy e Random. À medida que o orçamento de tempo aumenta para 80%, a ordem de desempenho permanece relativamente constante em comparação com o segundo gráfico analisado.

Na Figura 29, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'lexisNexis', observamos que o Thompson Sampling supera a es-

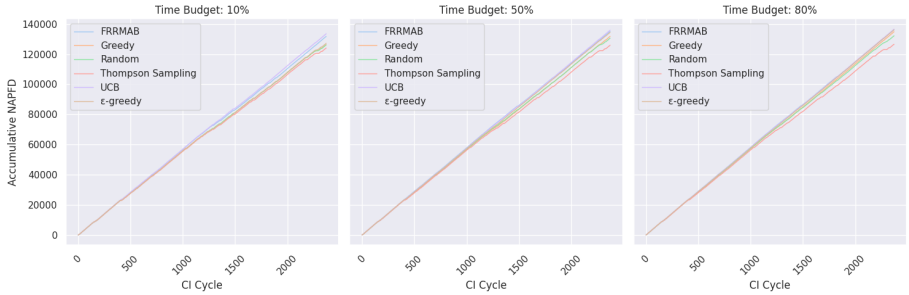

Figure 28. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'fatjson'.

tratégia aleatória, embora seja superado por outras abordagens, como o Greedy e o UCB. No entanto, é notável que o Thompson Sampling demonstrou ser ainda mais eficiente na identificação da melhor ação, exigindo 35% menos tentativas para alcançar o mesmo resultado. Isso destaca sua capacidade de oferecer um bom desempenho, apesar de não ser a estratégia líder em todas as circunstâncias.

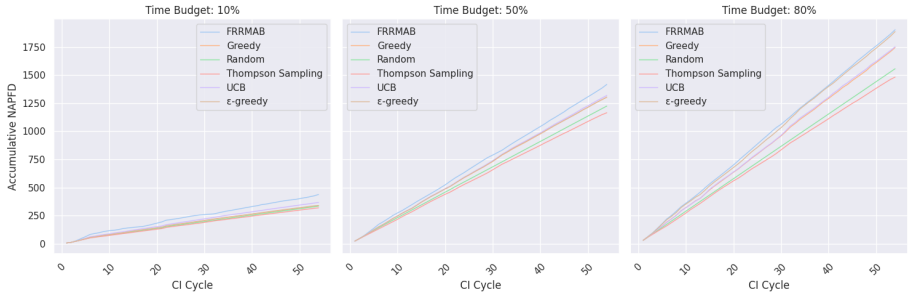

Figure 29. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'lexisNexis'.

3.3.8.2. NAPFD com algoritmo fitness_variation Na Figura 30, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'druid', destacam-se três principais estratégias de seleção de braço em problemas de bandit multi-armed. A estratégia Greedy, embora simples, tende a explorar apenas o braço atualmente considerado o melhor, o que pode levar à convergência rápida para uma solução subótima. Por outro lado, o UCB (Upper Confidence Bound) emprega intervalos de confiança para equilibrar a exploração e a exploração, demonstrando ser potencialmente mais eficiente do que o Thompson Sampling em determinados cenários. Além disso, a estratégia ϵ-greedy, que alterna entre exploração e exploração com probabilidades específicas, oferece uma abordagem mais simplificada, mas pode não ser tão eficaz quanto o Thompson Sampling em termos de identificação da melhor ação. Essas análises ressaltam a importância de considerar diferentes estratégias e suas características ao enfrentar problemas de bandit multi-armed.

Na Figura 31, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'fatjson', observa-se que o Thompson Sampling apresenta vantagens significativas em comparação com outras estratégias, como Greedy, UCB, ϵ-greedy e Ran-

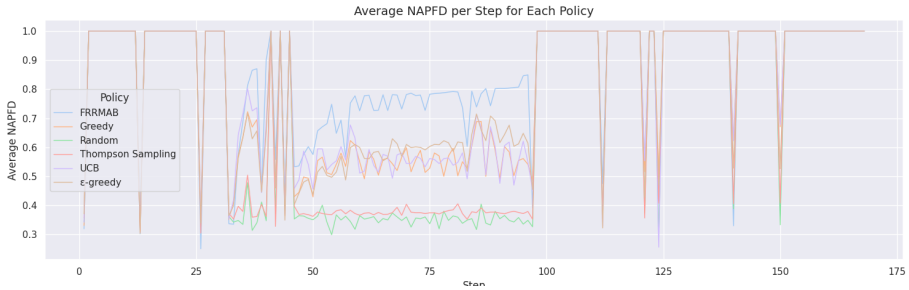

Figure 30. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'druid'.

dom, em termos de minimização do arrependimento cumulativo. De fato, o Thompson Sampling tende a superar essas outras estratégias nesse aspecto.

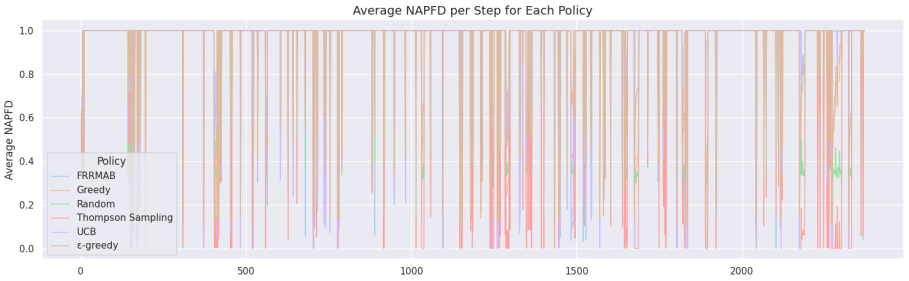

Figure 31. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'fatjson'.

Na Figura 32, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'deeplearning4j', observamos o desempenho de diferentes estratégias de seleção de ação. A estratégia Greedy tende a ser menos exploratória, potencialmente perdendo oportunidades de encontrar ações melhores. Por sua vez, a estratégia ϵ-greedy busca equilibrar exploração e exploração, mas pode ser menos adaptável do que o Thompson Sampling. Enquanto isso, o UCB (Upper Confidence Bound), embora popular, pode ser sensível a parâmetros e menos adaptável. Por fim, a abordagem aleatória mostra-se ineficaz em cenários complexos.

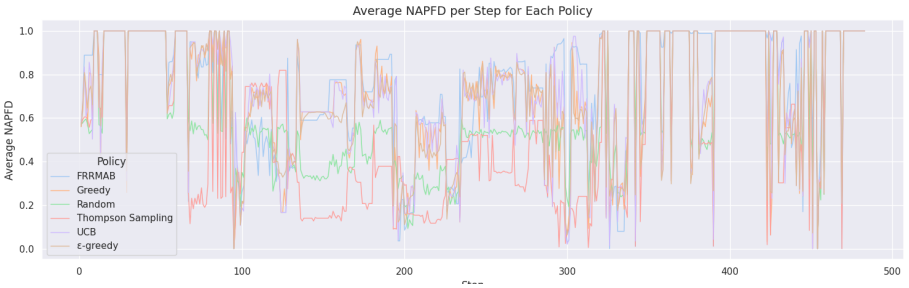

Figure 32. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'deeplearning4j'.

Na Figura 33, ao analisar os dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'lexisNexis', observamos o desempenho de diversas estratégias de

seleção de ação. A estratégia FRRMAB mostra um pico inicial, mas diminui rapidamente, sendo sensível a mudanças nas recompensas. Por outro lado, o Greedy não explora adequadamente e pode ficar preso em ações subótimas, enquanto o Random é totalmente exploratório, sem usar informações sobre recompensas. O UCB busca balancear exploração e exploração, mas pode ser menos adaptativo, assim como o ϵ-greedy, que combina exploração e exploração, mas pode ser sensível à escolha de ϵ. Por fim, a estratégia Thompson Sampling parece adotar uma abordagem equilibrada entre essas características.

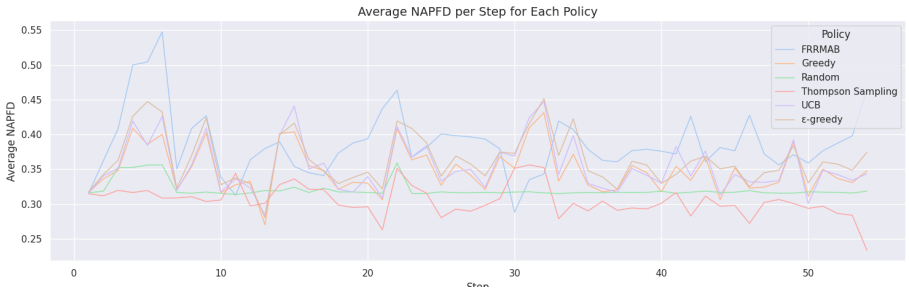

Figure 33. Análise de dados utilizando o algoritmo *fitness_variation* com o conjunto de dados 'lexisNexis'.

3.3.9. APFDc

Os gráficos abaixo apresentam o desempenho de diferentes algoritmos (*FRRMAB*, *Greedy*, *Random*, *Thompson Sampling* e ε-greedy) em três cenários de tempo (10%, 50% e 80% do tempo total), usando a métrica APFDc (*Average Percentage of Faults Detected cumulative*). A APFDc avalia a eficácia de um conjunto de testes em detectar defeitos com base na ordem de execução dos casos de teste. Quanto maior o valor da APFDc, mais eficaz é o conjunto de testes em encontrar defeitos rapidamente.

3.3.9.1. APFDc com algoritmo cum_cost Na Figura 34, ao analisar os dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'druid', observamos o desempenho de diversas estratégias de seleção de ação. A estratégia FRRMAB se destaca, apresentando o melhor desempenho em todos os três orçamentos de tempo. Ela aprende rapidamente e acumula mais recompensas ao longo do tempo. Por outro lado, o Greedy também tem um desempenho sólido, embora fique atrás do FRRMAB. No entanto, supera o Thompson Sampling, especialmente nos estágios iniciais. Enquanto isso, as estratégias Random, UCB e ϵ-greedy apresentam desempenhos semelhantes ou inferiores ao Thompson Sampling.

Na Figura 35, ao analisar os dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'fatjson', observamos que o Thompson Sampling pode superar outras estratégias apresentadas no gráfico, especialmente em cenários onde a incerteza é alta e a adaptação dinâmica é crucial.

Na Figura 36, ao analisar os dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'deeplearning4j', observamos o desempenho de diversas estratégias de seleção de ação. A estratégia FRRMAB supera consistentemente o Thompson Sampling

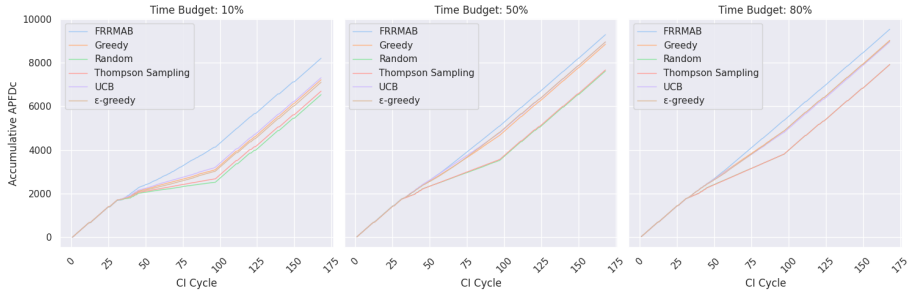

Figure 34. Análise de dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'druid'.

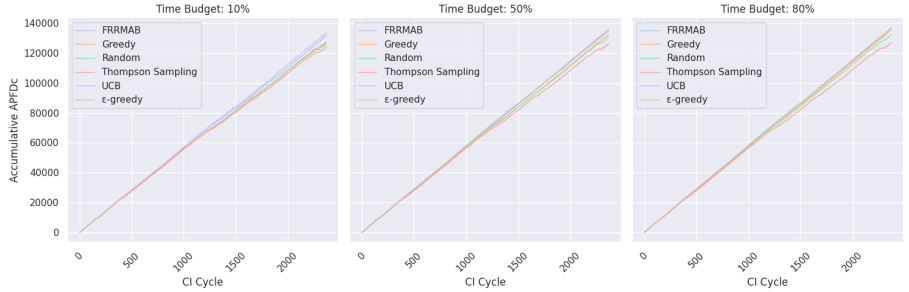

Figure 35. Análise de dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'fatjson'.

em todos os três orçamentos de tempo, enquanto o Greedy e o ϵ-greedy têm desempenhos inferiores. O Random é claramente a estratégia de pior desempenho, enquanto o UCB apresenta resultados semelhantes ao Thompson Sampling, mas ligeiramente inferiores na maioria dos casos. Em resumo, o Thompson Sampling é uma estratégia valiosa, mas sua eficácia depende do contexto e da natureza das recompensas das ações.

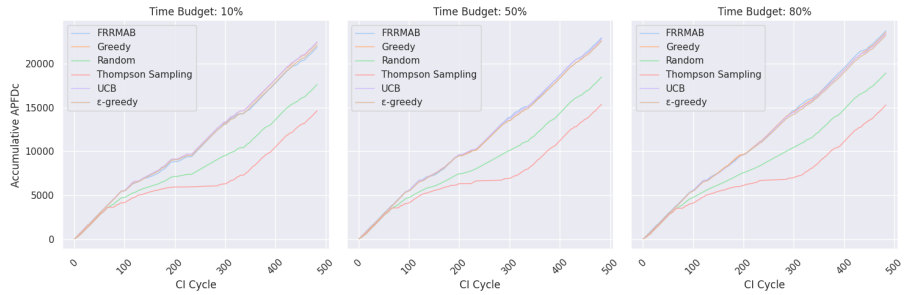

Figure 36. Análise de dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'deeplearning4j'.

Na Figura 37, ao analisar os dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'lexisNexis', observamos o desempenho de diversas estratégias de seleção de ação. A estratégia FRMMAB se destaca, apresentando o melhor desempenho em todos os três orçamentos de tempo, com um crescimento rápido e estável do APFDc acumulado. Por outro lado, as estratégias Greedy e ϵ-greedy apresentam desempenhos semelhantes ao Thompson Sampling, mas ligeiramente inferiores na maioria dos casos. Enquanto isso, as estratégias Random e UCB têm o pior desempenho, com o Random consistentemente

apresentando resultados mais baixos. Em resumo, o Thompson Sampling é uma estratégia sólida, mas pode não ser a melhor escolha para identificar o melhor braço em termos de desempenho após a fase de exploração.

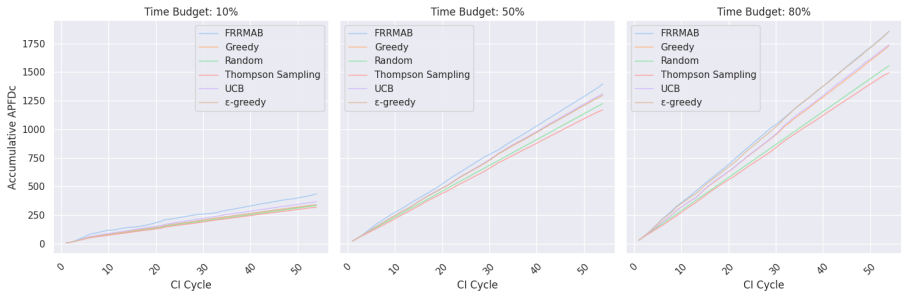

Figure 37. Análise de dados utilizando o algoritmo *cum_cost* com o conjunto de dados 'lexisNexis'.

3.3.9.2. APFDc com algoritmo cost_variation Na Figura 38, ao analisar os dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'druid', observamos o desempenho de diferentes políticas de seleção de ação. A política FRRMAB apresenta picos significativos de desempenho, mas também mostra quedas acentuadas, sendo mais volátil em comparação com o Thompson Sampling. O comportamento do Greedy é semelhante ao FRRMAB, porém não atinge valores tão altos de APFDc. Tanto o Random quanto o ϵ-greedy têm desempenhos inferiores em comparação com o Thompson Sampling, com o Random apresentando o menor valor de APFDc. O gráfico representa o desempenho médio das diferentes políticas ao longo do tempo ou etapas. As linhas para FRRMAB e Greedy mostram picos significativos, enquanto Random e ϵ-greedy têm desempenho inferior em comparação com o Thompson Sampling.

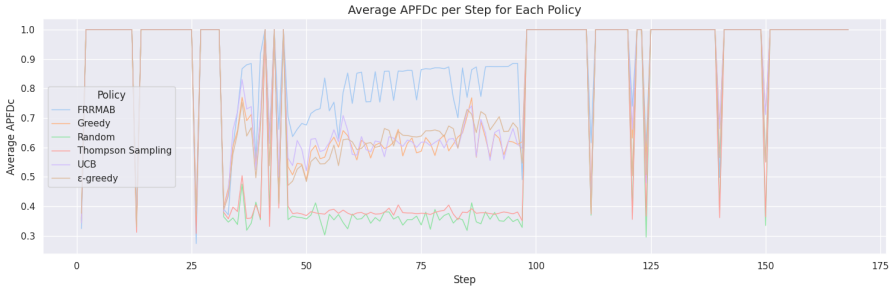

Figure 38. Análise de dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'druid'.

Na Figura 39, ao analisar os dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'deeplearning4j', observamos diversos aspectos relacionados à variabilidade e desempenho das estratégias de seleção de ação. A estratégia Thompson Sampling mostra uma variabilidade significativa nos resultados ao longo do tempo, como evidenciado pela flutuação considerável na linha correspondente no gráfico. Durante a fase de exploração, o Thompson Sampling demonstra um desempenho sólido em relação ao Simple Regret. No entanto, após essa fase, pode não otimizar a identificação do melhor braço para jogar durante a fase de exploração 1. Comparando com outras estratégias

no gráfico, observamos que a FRRMAB apresenta picos agudos de desempenho, mas também tem quedas significativas, indicando inconsistência. O Greedy mostra um desempenho estável, mas não atinge os picos mais altos observados em algumas outras estratégias. O Random exibe um desempenho muito variado, sem uma tendência clara de melhoria, enquanto o UCB apresenta um padrão semelhante ao Thompson Sampling, mas com menos variabilidade. Por fim, o ϵ-greedy mostra melhorias incrementais, mas não alcança os picos de desempenho observados em estratégias como FRRMAB ou Thompson Sampling.

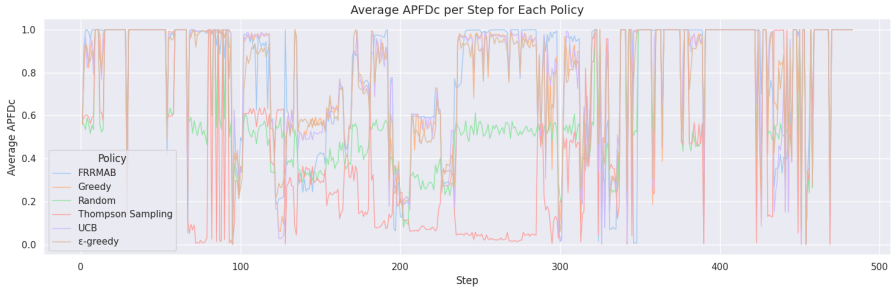

Figure 39. Análise de dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'deeplearning4j'.

Na Figura 40, ao analisar os dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'fatjson', observamos o desempenho de diversas estratégias de seleção de ação. O Thompson Sampling geralmente supera o FRRMAB em termos de desempenho, especialmente quando as recompensas são incertas ou variáveis. Enquanto o Thompson Sampling equilibra exploração e exploração, o Greedy é puramente exploratório ou puramente exploratório. Comparado com o Random, o Thompson Sampling é mais eficiente, pois leva em consideração informações anteriores. O Thompson Sampling e o UCB têm abordagens diferentes para a exploração, com o UCB usando intervalos de confiança, enquanto o Thompson Sampling incorpora incerteza probabilística. Embora o Thompson Sampling supere o ϵ-greedy em cenários incertos, o ϵ-greedy é mais simples de implementar.

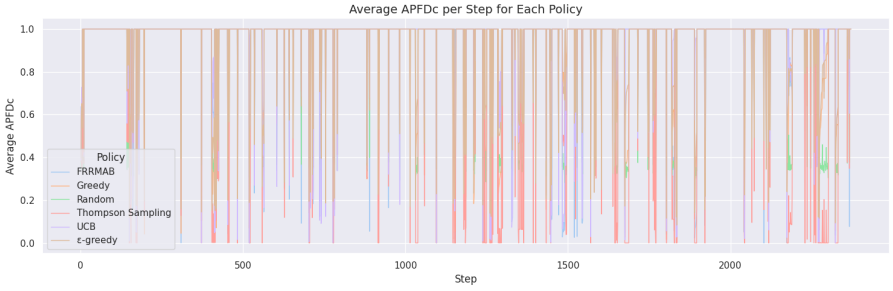

Figure 40. Análise de dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'fatjson'.

Na Figura 41, ao analisar os dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'lexisNexis', observamos o desempenho de várias estratégias de seleção de ação. A estratégia FRRMAB mostra um pico inicial alto, mas depois se estabiliza e performa de forma similar ao Thompson Sampling. O Greedy exibe flutuações

significativas e não é muito estável ao longo do tempo. Como esperado, o Random tem um desempenho inconsistente. O UCB apresenta um desempenho relativamente bom e estável, superando o Thompson Sampling em alguns pontos. Já o ϵ-greedy mostra flutuações, mas tende a ter um desempenho melhor que o Thompson Sampling ao longo do tempo. Em resumo, o Thompson Sampling é uma estratégia sólida, mas sua eficácia depende do contexto e das prioridades específicas do problema.

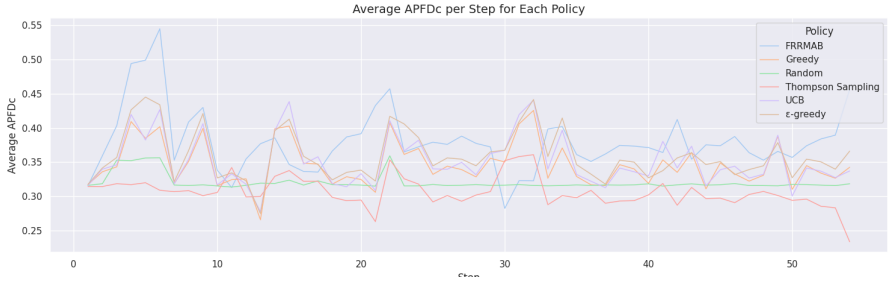

Figure 41. Análise de dados utilizando o algoritmo *cost_variation* com o conjunto de dados 'lexisNexis'.

3.3.9.3. Considerações da Análise Os gráficos apresentam o resultado utilizando as métricas APFDc (*Average Percentage of Faults Detected cumulative*) e NAPFD (*Normalized Average Percentage of Faults Detected*) com os algoritmos *cum fitness* e *fitness variation*. Os três gráficos mostram o desempenho de diferentes algoritmos (FRRMAB, Greedy, Random, Thompson Sampling e ϵ-greedy) em três cenários de tempo (10%, 50% e 80% do tempo total). O eixo Y é o desempenho acumulado (APFDc ou NAPFD) e o eixo X é o ciclo CI. Todos os algoritmos têm melhor desempenho com mais tempo disponível. Em resumo, quanto mais tempo disponível, melhor o desempenho dos algoritmos.

3.4. Conclusão

Em resumo, os gráficos destacam a eficácia do Thompson Sampling e FRRMAB em termos de APFDc e NAPFD em vários cenários. No entanto, é importante considerar as características específicas de cada problema ao escolher a estratégia de seleção de ação mais adequada.

4. Dificuldades

A primeira dificuldade encontrada pela equipe foi a instalação da versão atual do EvoSuite. A documentação oficial que foi estudada para aplicação posterior da ferramenta disponibilizava apenas como obter a versão 1.0.6. A equipe notou que ao utilizar esta versão eram necessárias muitas modificações para a utilizar em projetos com java 9 ou superior, fazendo assim opção por utilizar a sua versão atual. Foi preciso uma busca criteriosa por fóruns para encontrar meios de obter a nova versão. Após obter o projeto da forma mais simples possível que foi clonando o repositório oficial do EvoSuite, foi necessário encontrar o arquivo .jar responsável por sua execução. Por não existir um arquivo com o nome equivalente ao evosuite-1.0.6.jar modificando apenas o número da versão, foi necessário o teste manual dos arquivos .jar disponíveis no projeto para encontrar o correto, encontrando o arquivo de nome evosuite-master-1.2.1-SNAPSHOT.jar.

Outra dificuldade encontrada pela equipe foi a limitação do hardware disponível para o estudo da aplicação das ferramentas. Os computadores disponibilizados são de configurações básicas. A exemplo: o tempo das dez execuções do EvoSuite se estenderam por três dias. Quanto ao COLEMAN foi necessário utilizar a busca de uma máquina com mais recursos de hardware para possibilitar sua execução. Logo as limitações com o hardware se entrelaçaram com as limitações de tempo e não foi possível realizar mais etapas como a a execução de diferentes configurações do EvoSuite.

5. Contribuições

A tabela 2 mostra como a equipe se dividiu inicialmente para a realização dos trabalhos, a ideia era de cada um ficasse responsável por realizar uma parte importante da atividade simultaneamente.

Table 2. Divisão inicial das atividades

Atividade	RESPONSÁVEL
Bruna Mariana F. de Souza	EvoSuite
Débora da C. Medeiros	Relatório
Valber C. Tavares	COLEMAN

Porém algumas dificuldades foram sendo encontradas no decorrer da realização das atividades, resultando em uma nova divisão como mostra a tabela 3. As discentes Bruna e Débora, ficaram responsáveis por pesquisar como instalar a versão atual do EvoSuite, além de pesquisar repositórios robustos de código aberto que possuíam como linguagem principal o Java. Posteriormente enquanto a discente Débora começou a escrever a fundamentação teórica do projeto, a discente Bruna se ocupou na execução da geração de testes com o EvoSuite. De forma simultaneamente o discente Valber se ocupou na instalação, configuração e estudo do COLEMAN, porém ele encontrou uma série de dificuldades e como o tempo estava se esgotando, a equipe de comum acordo se reorganizou.

Os discentes Bruna e Valber ficaram responsáveis por continuar em conjunto pela pesquisa da politica MAB e pelos experimentos com o COLEMAN. Enquanto que a discente Débora ficou responsável pela análise dos resultados do EvoSuite para a escrita do relatório.

6. Conclusão

Após analisar os resultados, fica evidente que as políticas de priorização de testes em integração contínua apresentam perfis de desempenho distintos. A abordagem da Thompson Sampling se destaca ao oferecer tempos médios de priorização mais baixos em cenários com orçamentos de tempo restritos, demonstrando sua habilidade para tomar decisões rápidas. No entanto, em situações com orçamentos de tempo mais amplos, outras estratégias, como FRAMAB e -greedy, superam a Thompson Sampling. Além disso, a variabilidade nos resultados da Thompson Sampling é relativamente baixa em

Table 3. Divisão final das atividades

Atividade	RESPONSÁVEL
Bruna Mariana F. de Souza	EvoSuite e COLEMAN
Débora da C. Medeiros	EvoSuite e Relatório
Valber C. Tavares	COLEMAN

certos contextos, indicando consistência. Em relação aos testes criados pelo EvoSuite, eles conseguiram obter uma cobertura significativamente maior do que os testes originais do projeto. Esses testes são geralmente concisos, o que facilita a sua manutenção. Em suma, a escolha da estratégia ideal depende das prioridades específicas do contexto e dos trade-offs entre eficiência e explorabilidade. A Thompson Sampling pode ser uma escolha eficiente em termos de tempo quando os recursos são limitados, mas outras abordagens podem ser mais adequadas em cenários com mais tempo disponível. Utilizar a Thompson Sampling e o EvoSuite em um contexto de testes de uma empresa pode significar um ganho de tempo e economia de recursos significativo.

References

Alberto, J. e. a. (2020). Automação de testes: Um estudo de caso.

Cavalcante, I. S. e. a. (2020). Test case prioritization: a case study in the evolution of a real system.

Coutinho, J. C. d. S. e. a. (2022). Agile eteasy: um método para aplicação de testes exploratórios em contextos ágeis.

Datskiv, S. (2023). Can rvea with dynamosa features perform well at generating test cases?

Heskebeck, F., BERGELING, C., and BERNHARDSSON, B. (2022). Multi-armed bandits in brain-computer interfaces. *Frontiers in Human Neuroscience*, 16:931085.

Li, E. (2023). Investigating the performance of spea-ii on automatic test case generation.

Lima, J. A. P. and Vergilio, S. R. (2020). A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, 48(2):453–465.

Lima, J. A. P. and VERGILIO, S. R. (2020). A multi-armed bandit approach for test case prioritization in continuous integration environments. *IEEE Transactions on Software Engineering*, 48(2):453–465.

Marques, N. N. and Fernandes, R. A. (2020). Um arcabouço para a geração automatizada de testes funcionais a partir de cenários bdd.

Meidani, K., , et al. (2022). Mab-os: multi-armed bandits metaheuristic optimizer selection. *Applied Soft Computing*, 128:109452.

Oliveira, C. M. d. (2024). Utilização de chat bots baseados em llms para automação de testes de software.

Pinheiro, I. G. C. (2022). Algoritmo meta-heurístico para o problema de geração de dados de teste. *BS thesis*.

Pizzaia, V. H. and MALARA, R. D. (2022). Garantia da qualidade de software com devops. *RECIMA21-Revista Científica Multidisciplinar*, 3(11):e3112193–e3112193.

Riquelme, C., Tucker, G., and Snoek, J. (2018). Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127*.

Teixeira, E. (2022). Análise de ferramentas de automação de testes de software. *Manancial - Repositório Digital da UFSM*.

TERCEIRO, A. S. d. A. (2013). *Caracterização da Complexidade Estrutural em Sistemas de Software*. PhD thesis.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294.

Vogl, S. et al. (2021). Encoding the certainty of boolean variables to improve the guidance for search-based test generation. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1088–1096.