Universidade Federal de Goiás
Instituto de Informática

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Goiânia
2025

Universidade Federal de Goiás

Instituto de Informática

Autorização para Publicação de Trabalho de Conclusão de Curso em Formato Eletrônico

Na qualidade de titular dos direitos de autor, **AUTORIZO** a Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos I e VI, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulte, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Autor(a): Gabriel Porto do Carmo

Goiânia, de Novembro de 2025.

Gabriel Porto do Carmo – Autor

Gilmar Ferreira Arantes – Orientador

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Bacharelado em Ciência da Computação.

Área de concentração: Qualidade de Software.

Orientador: Prof. Gilmar Ferreira Arantes

Goiânia
2025

Gabriel Porto do Carmo

Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial

Trabalho de Conclusão apresentado à Coordenação do Curso de Bacharelado em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Bacharel em Bacharelado em Ciência da Computação, aprovada em de Novembro de 2025, pela Banca Examinadora constituída pelos professores:

Prof. Gilmar Ferreira Arantes
Instituto de Informática – UFG
Presidente da Banca

Prof. <Nome do membro da banca>
<Unidade acadêmica> – <Sigla da universidade>

Profa. <Nome do membro da banca>
<Unidade acadêmica> – <Sigla da universidade>

Gabriel Porto do Carmo

<Texto com um perfil resumido do autor do trabalho. Por exemplo: (Graduou–se em Artes Cênicas na UFG - Universidade Federal de Goiás. Durante sua graduação, foi monitor no departamento de Filosofia da UFG e pesquisador do CNPq em um trabalho de iniciação científica no departamento de Biologia. Durante o Mestrado, na USP - Universidade de São Paulo, foi bolsista da FAPESP e desenvolveu um trabalho teórico na resolução do Problema das Torres de Hanói. Atualmente desenvolve soluções para problemas de balanceamento de ração para a pecuária de corte.)>

<Dedicatória do trabalho a alguma pessoa, entidade, etc.>

Agradecimentos

<Texto com agradecimentos àquelas pessoas/entidades que, na opinião do autor, deram alguma contribuíção relevante para o desenvolvimento do trabalho.>

<Epígrafe é uma citação relacionada com o tópico do texto>

<Nome do autor da citação>,
<Título da referência à qual a citação pertence>.

Resumo

Carmo, Gabriel Porto do. **Geração Automática de Casos de Teste para Robot Framework: Uma Abordagem Baseada em Inteligência Artificial**. Goiânia, 2025. 44p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

<Resumo do trabalho>

Palavras–chave

<Palavra chave 1, palavra chave 2, etc.>

Abstract

Carmo, Gabriel Porto do. **<Work title>**. Goiânia, 2025. 44p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

A sketchy summary of the main points of the text.

Keywords

<Keyword 1, keyword 2, etc.>

Contents

List of Figures

List of Tables

CHAPTER 1

Introdução

Os testes de software são essenciais para a garantia e validação das funcionalidades de um projeto, desde a concepção inicial até sua versão final. Eles têm como objetivo validar funcionalidades, identificar falhas e assegurar a qualidade do software, evitando que erros cheguem ao usuário.

No âmbito dos testes, existem diversos tipos e técnicas que podem ser aplicados de acordo com as funcionalidades e a estrutura do projeto. Os testes podem ser manuais ou automatizados, abrangendo desde testes unitários, que validam individualmente uma parte do código, até testes de integração e de sistema, que avaliam a interação entre módulos e o funcionamento completo do software. Cada abordagem tem seu papel na construção de um processo confiável de desenvolvimento e entrega ao usuário.

Em alguns casos, os testes tornam-se ainda mais indispensáveis, especialmente quando envolvem funcionalidades críticas que podem comprometer completamente o software e causar falhas catastróficas no mundo real. Nesse cenário, a validação contínua da qualidade e da segurança torna-se uma exigência constante. Para isso, os processos e as técnicas de teste devem ser cuidadosamente planejados, priorizando as melhores estratégias para cobrir todas as funcionalidades essenciais do software.

Durante o desenvolvimento e a manutenção do software, mudanças e melhorias são necessárias, e os testes devem acompanhar essas alterações. Uma das alternativas para esse monitoramento e aperfeiçoamento é a aplicação de testes automatizados, que surgem como uma solução eficaz para manter os testes sempre atualizados.

Porém, para que essa técnica seja realmente eficaz, os testes não podem ser uma etapa isolada do desenvolvimento; eles devem estar presentes desde o início. É essencial que façam parte de cada momento do ciclo de vida do software, tornando-se parte natural do trabalho da equipe. Essa abordagem tem sido cada vez mais adotada em metodologias ágeis e na cultura DevOps, promovendo uma integração contínua da qualidade ao longo de todo o processo de desenvolvimento.

[é necessário enriquecer um pouco mais esta introdução]

1.1 Justificativa

Este trabalho surgiu do interesse pessoal do autor, pela área de testes de software e pela busca por soluções que automatizem estes testes. A validação automatizada exige uma abordagem constante e personalizada. O grande desafio é garantir que todas as funcionalidades do software continuem funcionando corretamente, especialmente após alterações no seu código-fonte, o que é fundamental para garantir a qualidade e a segurança do software.

Com foco principal na melhoria dos testes de software por meio da automação, este trabalho também utiliza Inteligência Artificial (IA) para validar e apoiar a tomada de decisões. Isso não apenas reduz o risco de problemas não detectados, mas também contribui para um ciclo de desenvolvimento mais ágil e seguro.

Além disso, este trabalho também busca fomentar discussões sobre como a automação de testes e o uso de IA podem melhorar a validação de sistemas de software. Ao detectar defeitos, o quanto antes, agiliza o ciclo de desenvolvimento, beneficiando os envolvidos no processo de desenvolvimento de software.

Em resumo, a ideia é o aprimoramento contínuo e a ampliação do conhecimento sobre todos os aspectos relacionados à automação de testes, beneficiando assim a comunidade de profissionais e entusiastas da área.

1.2 Objetivos

1.2.1 Objetivo Geral

Esse estudo tem como objetivo analisar como a automação de teste, combinado com o uso de inteligencia artificial podem melhorar o teste de software, com foco na validação contínua das funcionalidades do software, promovendo maior qualidade, segurança e eficiência no ciclo de desenvolvimento.

1.2.2 Objetivos Específicos

- Compreender como a automação de testes pode ser aplicada de forma eficaz em todas as partes de um software;
- Analisar como a Inteligência Artificial pode ser utilizada para identificar falhas e auxiliar na tomada de decisões durante os ciclos de testes;

- Apresentar casos práticos onde o uso do *Robot Framework*[1] e do *GitHub Actions* [2] contribui para a detecção de defeitos e a integração contínua do código;
- Avaliar a eficácia da validação automatizada.

1.3 Metodologia

Este trabalho se caracteriza como uma pesquisa aplicada, de natureza qualitativa e exploratória. O principal objetivo [tem que tirar este objetivo daqui, pois tem seção específica para os objetivos] é avaliar uma abordagem para detectar e propor soluções para falhas em sistemas web, utilizando ferramentas de automação de testes e plataformas de versionamento de código

Na fase inicial da pesquisa, realizou-se uma revisão bibliográfica abrangente, com análise das publicações de (MYERS; SANDLER; BADGETT, 2011) sobre testes de software, complementadas por livros, artigos e materiais técnicos atuais sobre testes automatizados

Em seguida, foram desenvolvidos testes automatizados utilizando Robot Framework e *Selenium WebDriver*. Os primeiros testes foram aplicados em um site com cenários de erro proposital[3]. Esse ambiente foi ideal para validar o funcionamento da estratégia de testes automatizados.

Depois dessa fase inicial, o projeto será implementada em um sistema real para testar sua eficácia em um ambiente de produção. Os testes foram configurados para rodar automaticamente sempre que um novo *commit* é feito no repositório do projeto no GitHub, ou em horários definidos, com o uso do *GitHub Actions*. Quando uma falha é detectada, o sistema gera automaticamente uma *issue* no próprio *GitHub*, contendo *prints* da tela, *logs* e uma descrição resumida com a ajuda de uma inteligência artificial que interpreta o erro.

Por fim, os resultados são analisados, observando os tipos de erros mais frequentes, o tempo ate encontrar a falha, e a utilidade das informações geradas para os desenvolvedores.

1.4 Organização do Trabalho

Para facilitar o alcance dos objetivos especificados, o restante deste trabalho está organzado da seguinte forma:

[1] https://robotframework.org/
[2] https://github.com/features/actions
[3] Disponível em: https://the-internet.herokuapp.com/login. Acesso em: 22 jun. 2025.

- No Capítulo 2 é apresentada a fundamentação teórica, que embasa todo este trabalho.
- No Capítulo 3 é apresentado o projeto propriamente dito...
- No Capítulo 4 são apresentadas as considerações finais e os possíveis desdobramentos futuros deste trabalho.

CHAPTER 2

Fundamentação Teórica

Para esta dissertação, é fundamental apresentar os conceitos essenciais que baseiam o processo de teste de software, destacando sua relevância para assegurar a qualidade em projetos de desenvolvimento. Os testes assumem um papel central na verificação de requisitos e na validação de funcionalidades, permitindo a identificação de falhas e a prevenção de erros que poderiam comprometer tanto a experiência do usuário quanto a segurança do software.

Com base nisso, diferentes técnicas de teste foram desenvolvidas e aprimoradas ao longo do tempo. Entre elas, destacam-se as técnicas funcionais e estruturais, que oferecem visões complementares para a avaliação do software. Enquanto os testes funcionais verificam se o sistema atende às especificações definidas, os estruturais exploram a lógica interna do código, permitindo uma análise mais detalhada do seu comportamento.

Já os níveis de teste representam as diferentes etapas em que o software é avaliado ao longo do desenvolvimento. Eles vão desde a verificação de partes menores do sistema até a análise do produto final, pronto para uso. A ideia é que cada nível contribua para aumentar a confiança no software, começando com verificações mais específicas e chegando a testes mais amplos, que simulam o uso real.

2.1 Técnicas de Teste de Software

Com o avanço nas práticas de desenvolvimento de software, diversas novas abordagens de teste foram surgindo. No entanto, algumas técnicas clássicas, desenvolvidas ainda nas primeiras décadas da computação, permanecem fundamentais até os dias de hoje, e com o mesmo propósito, de revelar a presença de defeitos no software.

As principais técnicas de teste ganharam destaque a partir das publicações de Glenford Myers (MYERS; SANDLER; BADGETT, 2011), que propôs a divisão entre testes funcionais e estruturais. Essas duas abordagens analisam de forma diferente o software. Os testes funcionais verificam se o software atende aos seus

requisitos, enquanto os testes estruturais avaliam o código por dentro. Dessa forma, é possível validar o software de maneira mais completa.

Essa relação demonstra que a aplicação de uma única técnica não é suficiente para garantir a segurança e a qualidade do software. É justamente a combinação entre diferentes abordagens que amplia a cobertura dos testes e fortalece a detecção de falhas. Além disso, a consolidação desses conceitos possibilitou o surgimento de métodos derivados e de ferramentas que são amplamente utilizadas em todo o ecossistema de testes.

2.1.1 Técnicas de Teste Funcional

A técnica de teste funcional ou teste caixa-preta, Figura 2.1[1], se baseia em verificar o funcionamento do software, sem ter que testar necessariamente o código que foi implementado. Nesse método, são definidos dados de entrada e os resultados esperados para cada situação. O teste é aprovado quando as respostas obtidas correspondem as respostas esperadas.

Da forma que a técnica foi desenvolvida, ela consegue ser aplicada em diferentes níveis do software, desde um método isolado até a aplicação completa. Além disso, esse modelo permite ao testador ter uma perspectiva mais próxima da experiência do usuário final, avaliando o comportamento do software como ele seria percebido na prática.

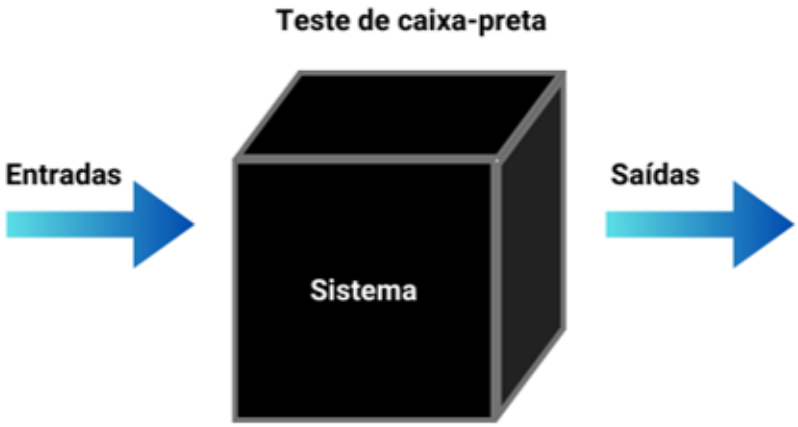


Figure 2.1: *Teste de Caixa Preta*

[1] <https://www.linkedin.com/pulse/testes-de-caixa-preta-anselmo-n-de-oliveira-ygdgf/>

2.1.2 Técnicas de Teste Estrutural

A técnica de teste estrutural, Figura 2.2, também conhecida como teste de caixa-branca, foca na validação do software a partir da análise direta do código-fonte. Diferentemente da abordagem funcional, esse modelo exige que o testador examine as estruturas internas, a lógica de execução e os métodos implementados, garantindo que todos os caminhos possíveis do código sejam devidamente testados.

Com base nesse código, são elaborados casos de teste que buscam cobrir toda a execução do requisito avaliado.

Os testes de unidade e os testes de integração são exemplos dessa abordagem, sendo amplamente aplicados pelos desenvolvedores ao longo do processo de desenvolvimento. Em um teste de unidade, por exemplo, pode-se verificar se um método responsável por calcular descontos e retorna o valor esperado para diferentes entradas. Já em um teste de integração, a atenção pode estar na interação entre módulos, como a comunicação entre o módulo de autenticação e o módulo de geração de relatórios. Esses testes permitem revelar a presença de defeitos em partes específicas do software e assegurar que cada elemento funcione corretamente em diferentes cenários.

É importante lembrar que a técnica estrutural não substitui a técnica funcional, elas devem se complementar. Enquanto o teste funcional valida o software a partir da visão do usuário, o estrutural garante confiabilidade e segurança do código, contribuindo para uma cobertura de testes mais completa e eficaz.

[é necessário informar qua é a fonte da imagem do teste caixa branca. Adicionar ao arquivo bib e referenciar aqui.]

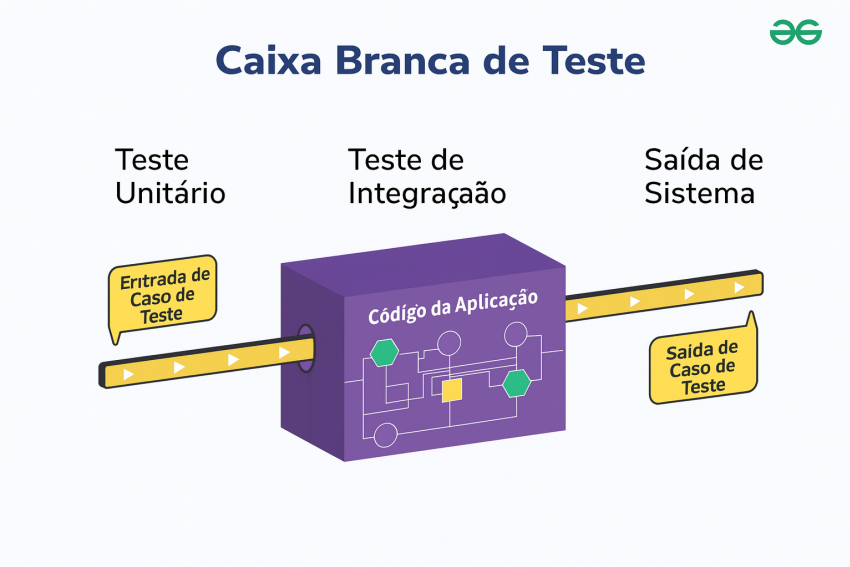


Figure 2.2: *Teste de Caixa Branca*

2.1.3 Técnicas de Teste Baseada em Defeitos

A técnica baseada em defeitos é uma abordagem que visa identificar vulnerabilidades no software por meio da introdução intencional de falhas. Em vez de se basear apenas em requisitos funcionais, essa técnica busca avaliar como o software se comporta diante de erros, explorando suas fragilidades e limites. Para isso, são utilizadas técnicas como *Mutation Testing*[2], conforme Figura 2.3, que realiza pequenas alterações no código, simulando erros comuns e *Fuzz Testing*, que fornece entradas aleatórias e inesperadas ao software. O objetivo é verificar se os testes conseguem detectar esses problemas e garantir que o softaware continue operando corretamente mesmo em situações adversas.

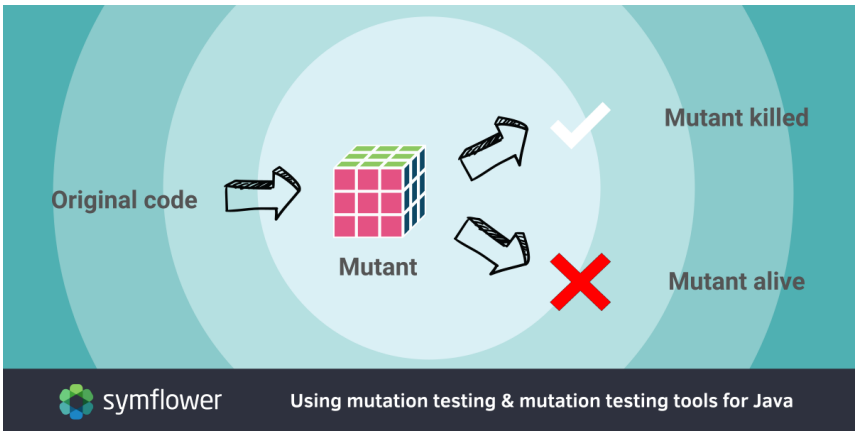


Figure 2.3: *Teste de Mutantes.*

2.1.4 Critérios de Teste

Os critérios de testes tem um papel fundamental no processo de validação, sendo responsáveis por determinar as regras e diretrizes que serão utilizadas no processo de analise. Dessa forma, o testador consegue determinar se o conjunto de testes elaborado é suficiente para verificar adequadamente o componente ou funcionalidade em questão, aumentando assim a chance de revelar a presença de defeitos no software.

Existem diferentes tipos de critérios, de acordo com a técnica que foi aplicada. Os principais são, o critérios de teste funcional, que focam principalmente no comportamento externo da aplicação, e os critérios de teste estrutural, que considera a lógica interna do código-fonte. Cada tipo possui técnicas específicas para auxiliar no planejamento e execução dos testes.

[2] <https://symflower.com/en/company/blog/2023/using-mutation-testing/>

Critérios de Teste Funcional

Para validar se o software atende aos requisitos, os critérios de testes funcional exercem papel fundamental. Essa abordagem permite que o testador simule diferentes cenários de uso real, avaliando se as funcionalidades respondem corretamente de acordo com o que foi especificado

1. **Particionamento por Equivalência**;
   O particionamento por equivalência é uma técnica que busca simplificar os testes, dividindo os dados de entrada em grupos que devem ser tratados da mesma forma pelo software. A ideia é que, se um valor de um grupo especifico funciona corretamente, os demais também funcionarão, esses grupos são conhecidos como classe de equivalência, e podem conter dados validos e inválidos.

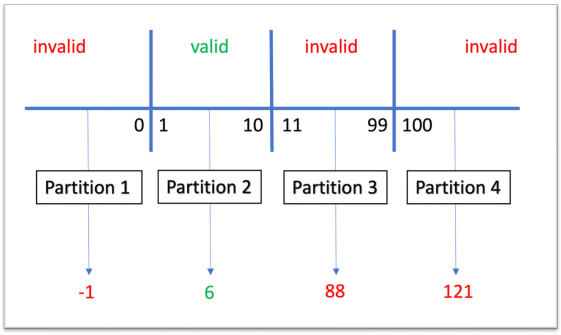


Figure 2.4: *Partição de equivalência*

[é necessário informar qua é a fonte da imagem do critério e teste funcional, particionamento por equivalência. Adicionar ao arquivo bib e referenciar aqui. Ou pode ser também uma nota de rodapé. Isso se aplica a todas as imagens que estão sem referência da fonte onde foram obtidas.]

2. **Análise do Valor Limite**;
   A analise do valor limite, é uma técnica que se baseia em pontos limites de uma faixa de dados, apresentando um abordagem simples, onde os erros costumam ocorrer nos pontos extremos das entradas. A partir disso, em vez de testar os valores típicos, o testador analisa os valores mais altos e baixos da amostra de dados. Essa técnica pode ser considerada um complemento para o particionamento por equivalência.
3. **Grafo de Causa e Efeito**
   A analise do valor limite, é uma técnica que se baseia em pontos limites de uma faixa de dados, apresentando um abordagem simples, onde os erros costumam ocorrer nos pontos extremos das entradas. A partir disso, em vez de testar os valores típicos, o testador analisa os valores mais altos e baixos da

amostra de dados. Essa técnica pode ser considerada um complemento para o particionamento por equivalência. (ARAUJO, 2015)

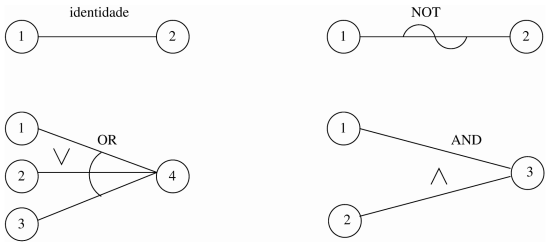


Figure 2.5: *Grafo de Causa e Efeito*

4. **Tabela de Decisão**.
   A Tabela de decisão é um técnica conhecida por testar softwares que envolvem varias condições lógicas e regras de negocio mais complexas. Ela ajuda a organizar e visualizar de maneira clara como o software deve se comportar com diferentes combinação de entradas.
   A técnica consiste em apresentar as ações e condições de entrada em um tabela, onde cada linha corresponde a um regra especifica, e as condições são avaliadas como verdadeiras ou falsas. Essa estrutura é bastante útil em cenários com diversas variações de comportamento, pois auxilia na identificação de todos os casos relevantes a serem testados.

Conditions	**R1**	**R2**	**R3**
Withdrawal Amount <= Balance	T	F	F
Credit granted	-	T	F
Actions			
Withdrawal granted	T	T	F

Figure 2.6: *Tabela de Decisão (ARAUJO, 2015)*

Critérios de Teste Estrutural

Para uma análise mais precisa do código-fonte, são aplicados os critérios de teste estrutural. Eles são utilizados quando o objetivo é verificar o funcionamento interno do software, analisando diretamente a lógica implementada, como condições, fluxos e estruturas de decisão presentes no código.

Alguns exemplos comuns desses critérios, referencia são:

- **Cobertura de instruções** , que garante que cada linha de código seja executada ao menos uma vez. Por exemplo, em um método que soma dois números, o critério assegura que a linha de retorno seja testada em diferentes situações.
- **Cobertura de caminhos** , que busca percorrer todos os caminhos possíveis de execução do código. Esse critério é útil em sistemas com múltiplas regras de negócio encadeadas.
- **Cobertura de decisões** , que verifica se todas as ramificações de um comando condicional, como if ou switch, foram testadas, e se os resultados estão de acordo com o esperado.

Para viabilizar a aplicação desses critérios, utiliza-se o Grafo de Fluxo de Controle (GFC) constitui uma representação gráfica que ilustra o fluxo de execução de programas ou aplicações, permitindo a análise detalhada do comportamento interno do software. Desenvolvido originalmente por Frances E. Allen, esse grafo é direcionado e orientado a processos, no qual os nós representam blocos básicos de instruções e as arestas indicam os possíveis caminhos de fluxo de controle. Essa estrutura facilita a compreensão, otimização e verificação da lógica de programas, sendo essencial para o desenvolvimento de software mais seguro e eficiente.

Dentre essas métricas, a Complexidade Ciclomática é uma medida fundamental em Engenharia de Software, introduzida por Thomas McCabe na década de 1970. Seu objetivo principal é quantificar a complexidade de um módulo de software por meio da análise de seus pontos de decisão, os quais determinam a quantidade de caminhos de execução possíveis no código.

Quanto maior o valor da complexidade ciclomática, maior a quantidade de ramificações lógicas (como estruturas condicionais e loops), o que tende a aumentar a dificuldade de compreensão, teste e manutenção do software. Por outro lado, valores mais baixos indicam código mais simples, estruturado e confiável, reduzindo a probabilidade de defeitos e facilitando a evolução do sistema.

Considerando essas características, os testes estruturais são geralmente aplicados nos níveis de teste unitário e de integração. Sua implementação requer conhecimento detalhado do código-fonte e o uso de ferramentas específicas, sendo portanto de responsabilidade primária do desenvolvedor. Essa abordagem assegura que a verificação da qualidade interna do software ocorra desde as etapas iniciais de desenvolvimento, contribuindo significativamente para a construção de sistemas mais robustos e confiáveis.

2.1.5 Níveis de Teste

Exemplo de Citação de Imagem. Figura 2.7

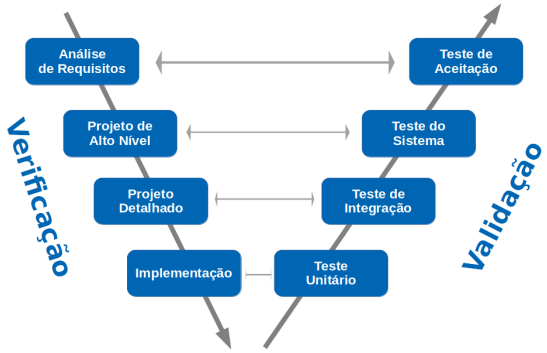


Figure 2.7: *Modelo V*

Teste Unitário

O testes unitários representam o nível básicos no processo de software, eles são responsáveis por verificar pequenos trechos de código, como métodos, funções e módulos isolados. A principal função é encontrar erros lógicos ou de implementação ainda nas etapas iniciais do desenvolvimento

Teste de Integração

Após a validação individual dos módulos, o teste de integração busca encontrar falhas que podem surgir quando esses módulos são integrados. Essa etapa é essencial para garantir que a estrutura do projeto esteja funcional e bem acoplada.

Teste de Sistema

No teste de sistema, o software como um todo é validado, já em sua versão integrada. Os testes são executados em condições semelhantes às do ambiente real de uso, simulando o comportamento de um usuário final e validando os requisitos de forma geral.

Teste de Aceitação

Os testes de aceitação são realizados por um grupo restrito de usuários finais, buscando confirma se o software esta pronto para ser colocado em produção. Esses devem simular tarefas e rotinas que serão realizadas quando o projeto estiver entregue

2.1.6 Tipos de Teste

Teste de Performance

O teste de performance tem como objetivo avaliar a eficiência de um software em termos de tempo de resposta, uso de recursos e estabilidade sob diferentes condições de operação. Esse tipo de teste é fundamental para assegurar que o sistema atenda a requisitos de qualidade relacionados a rapidez, confiabilidade e escalabilidade, fatores determinantes para a experiência do usuário e para a competitividade da aplicação no mercado.

Por meio do teste de performance, é possível identificar gargalos que impactam diretamente o funcionamento do sistema, como consultas lentas ao banco de dados, consumo excessivo de memória ou problemas de rede. Além disso, esses testes permitem definir parâmetros de aceitação, tais como o tempo máximo aceitável de resposta, a quantidade de transações suportadas por segundo e a eficiência na utilização de recursos do servidor.

Teste de Carga

O teste de carga consiste em simular a demanda real de usuários em um software, com o objetivo de analisar seu comportamento sob diferentes condições de tráfego, desde situações de uso leve até picos de acesso intenso. Esse método é geralmente aplicado nas fases finais do ciclo de desenvolvimento, permitindo avaliar a robustez e a estabilidade do sistema perante cenários operacionais previsíveis. Por meio dessa técnica, é possível assegurar que a aplicação suportará o volume esperado de usuários, além de identificar e corrigir possíveis problemas de desempenho antes que o software seja disponibilizado para o público geral

Teste de Estresse

O teste de estresse tem como objetivo avaliar até onde um sistema consegue suportar situações extremas, indo além das condições normais de uso. Ele é essencial porque não basta que um software funcione bem em cenários comuns, é preciso garantir também que ele se mantenha estável quando submetido a sobrecargas ou pressões inesperadas. Esse tipo de teste mostra o quão robusto e confiável o sistema realmente é, revelando se ele resiste sem travar ou falhar quando colocado em seu limite.

Teste de Regressão

O teste de regressão é uma técnica essencial no desenvolvimento de software, aplicada sempre que uma nova versão do sistema é lançada ou quando ele passa por ciclos de evolução contínua. O objetivo principal dessa abordagem é assegurar que as modificações implementadas não introduzam falhas em funcionalidades que já estavam sólidas.

Para isso, todos os testes que foram previamente executados nas versões ou ciclos anteriores são reaplicados à nova versão em avaliação. Além disso, leva-se em consideração as fases e técnicas de teste mais adequadas, de acordo com o impacto causado pelas alterações recentes.

2.2 Ferramentas de Teste

À medida que sistemas se tornam mais complexos e exigem maior rapidez em sua entrega, o uso de ferramentas de teste passa a ser essencial para apoiar equipes na criação, execução e gestão das atividades de verificação e validação.

Essas ferramentas oferecem recursos que vão desde a elaboração de casos de teste até a automação da execução e o acompanhamento dos resultados, possibilitando maior controle sobre o processo e reduzindo falhas que poderiam comprometer a experiência do usuário final. Além disso, permitem padronizar práticas, otimizar o tempo das equipes e fornecer métricas importantes para a tomada de decisão.

Outro aspecto importante é a capacidade dessas ferramentas de se integrarem a diferentes fases do ciclo de desenvolvimento, especialmente em metodologias ágeis e em práticas de integração e entrega contínuas (CI/CD). Isso possibilita que a verificação de qualidade seja incorporada de forma contínua e incremental, acompanhando a evolução do software e prevenindo problemas em estágios mais avançados do projeto.

2.3 Teste de Software e Inteligência Artificial

A inteligência artificial está revolucionando a área de testes de software, trazendo um novo patamar de eficiência para processos que antes dependiam quase que exclusivamente do trabalho manual e demandavam tempo significativo. O uso de técnicas de aprendizado de máquina, mineração de dados e análise preditiva permite acelerar atividades fundamentais como a geração de casos de teste, a priorização de cenários e a detecção de falhas, e ate mesmo propor soluções para os erros encontrados .

Entre os benefícios mais relevantes está a capacidade da IA em automatizar a criação de casos de teste, a partir da análise de requisitos, históricos de execução e até do comportamento dos usuários em produção. Isso amplia a cobertura dos testes e reduz a possibilidade de lacunas que poderiam comprometer a confiabilidade do sistema. Além disso, algoritmos inteligentes podem identificar padrões de falhas recorrentes e priorizar a execução dos testes mais críticos, direcionando os recursos para as áreas de maior risco.

Outro ponto de destaque é o suporte da IA na gestão de resultados de teste. Em projetos de grande porte, a quantidade de dados gerados pode dificultar a análise manual. Nesse cenário, sistemas baseados em IA auxiliam na categorização automática de falhas, na identificação de falsos positivos e na sugestão de correções mais prováveis, agilizando a tomada de decisão das equipes de desenvolvimento e qualidade.

CHAPTER 3

Projeto de Teste Automatizado

O projeto em questão, visa demonstrar a aplicação pratica do *Robot Framework*[1] na automação de testes funcionais em uma aplicação *web* real. A base do trabalho é avaliar como a utilização de um *framework* de automação baseado em *keywords* pode contribuir para a melhoria do processo de garantia da qualidade de software, ao proporcionar maior organização, reutilização de componentes e geração automática de evidências de execução.

3.1 Descrição do Projeto

Para a execução dos testes de interface, o projeto faz uso da *SeleniumLibrary*[2], que integra o *Robot Framework* ao *Selenium WebDriver*[3]. Essa combinação permite simular o comportamento real de um usuário no navegador. O *Selenium* é um dos *frameworks* mais consolidados para automação de navegadores, oferecendo suporte a diferentes navegadores e sistemas operacionais, o que garante flexibilidade e maior realismo nos cenários de teste.

Além do foco na automação de testes, o trabalho busca através da integração com a inteligencia artificial (IA) com a *api* do *gemini*[4], gerar um ambiente muito eficiente onde as falhas encontradas durante a execução dos testes não apenas são detectadas, mas também analisadas automaticamente. A IA sugere hipóteses sobre as possíveis causas e até caminhos para uma depuração mais rápida, servindo como um apoio extra para as equipes de *quality assurance* (QA) e de desenvolvimento.

Outro ponto importante é a conexão com o *GitHub Issues*[5]. Sempre que um defeito é identificado e diagnosticado, ele pode ser registrado automaticamente em um repositório de versionamento. Isso torna o fluxo de trabalho mais ágil e

[1] <https://robotframework.org/>
[2] <SeleniumLibrary>
[3] <https://www.selenium.dev/documentation/webdriver/>
[4] <https://ai.google.dev/gemini-api/docs>
[5] <https://github.com/features/issues>

colaborativo, reduzindo atividades manuais e garantindo a rastreabilidade entre o defeito, sua análise e o acompanhamento da correção.

Sendo assim, o objetivo deste estudo de caso vai além de apenas validar a execução de casos de teste. Ele busca mostrar o potencial da automação inteligente, unindo o *Robot Framework*, o *Selenium* e recursos de IA para apoiar diagnósticos e tornar o desenvolvimento de software ainda mais orientado à qualidade.

3.2 Contexto da Aplicação de Testes

Os testes automatizados desenvolvidos neste projeto foram aplicados sobre uma aplicação web pública e amplamente utilizada em treinamentos de automação: o portal *The Internet Herokuapp*[6]. Essa plataforma foi escolhida por oferecer diversos cenários controlados de teste, como formulários de login, *upload* de arquivos, botões dinâmicos e elementos interativos, que permitem validar diferentes aspectos do comportamento de uma aplicação *web*.

Durante a execução, o *Robot Framework*, em conjunto com a *SeleniumLibrary*, foi responsável por controlar o navegador e reproduzir as ações do usuário, como preencher campos, clicar em botões e validar mensagens de erro. Esse processo possibilitou avaliar a estabilidade da aplicação e a capacidade do *framework* em detectar e registrar falhas de interface de forma confiável.

Além disso, o contexto do teste foi ampliado com a integração de ferramentas complementares. A *API Gemini* foi utilizada para interpretar automaticamente as falhas registradas, fornecendo análises e hipóteses sobre suas possíveis causas, enquanto a integração com o *GitHub Issues* garantiu que cada erro detectado fosse documentado diretamente em um repositório de versionamento, permitindo o acompanhamento e rastreabilidade do ciclo de correção.

Dessa forma, o contexto da aplicação de testes reflete um ambiente que simula situações reais de uso e falhas comuns em sistemas web, demonstrando como a automação aliada à inteligência artificial pode contribuir para tornar o processo de validação mais inteligente, ágil e integrado ao fluxo de desenvolvimento de software.

[6] https://the-internet.herokuapp.com

3.3 Arquitetura e Estrutura do Projeto

3.4 Tecnologias Utilizadas

O desenvolvimento deste projeto envolveu o uso de diversas tecnologias que, em conjunto, permitiram construir um ambiente de automação robusto, integrado e inteligente. Cada uma delas desempenha um papel específico na execução dos testes e na análise automatizada dos resultados.

3.4.1 *Robot Framework*

O **Robot Framework** é a base do projeto, sendo um *framework* de automação de testes de código aberto amplamente utilizado para testes funcionais, de aceitação e de integração. Seu principal diferencial é o uso de uma linguagem baseada em *keywords*, que torna a escrita dos testes mais legível e acessível, mesmo para profissionais que não possuem conhecimento avançado em programação. Além disso, sua arquitetura modular permite a integração com diferentes bibliotecas e ferramentas externas, o que facilita a expansão das funcionalidades.

3.4.2 *Selenium WebDriver* e *SeleniumLibrary*

A automação da interface foi implementada por meio da ***SeleniumLibrary***, que faz a ponte entre o *Robot Framework* e o ***Selenium WebDriver***. Essa integração permite controlar o navegador da mesma forma que um usuário real, simulando ações como cliques, preenchimento de campos e navegação entre páginas. O *Selenium* é uma das tecnologias mais consolidadas para automação de testes em aplicações *web*, oferecendo suporte a múltiplos navegadores e sistemas operacionais, o que garante flexibilidade e portabilidade aos testes.

3.4.3 API Gemini

A integração com a **API Gemini**, da Google, representa o elemento de IA do projeto. Sempre que um defeito é detectado durante a execução dos testes, a IA é acionada para interpretar o contexto do defeito e gerar uma análise automática, sugerindo causas técnicas prováveis e ações de depuração. Essa abordagem agrega um nível de automação cognitiva ao processo, reduzindo o tempo de diagnóstico e apoiando as equipes de QA e desenvolvimento na identificação de problemas.

3.4.4 GitHub e GitHub Issues

O **GitHub** foi utilizado tanto como repositório para controle de versão do código quanto como ferramenta de rastreabilidade de falhas. A integração direta com o **GitHub Issues** permite que cada defeito identificado seja automaticamente registrado como uma ocorrência documentada, contendo detalhes do teste, a mensagem de erro e a análise gerada pela IA. Esse processo automatizado facilita o acompanhamento das correções e promove maior colaboração entre os times envolvidos.

Dessa forma, a combinação dessas tecnologias possibilitou a criação de um ecossistema de automação completo, no qual a execução de testes, a detecção de falhas, a análise inteligente e o registro de ocorrências estão integrados em um único fluxo automatizado e eficiente.

3.5 Fluxo de Execução dos Testes com IA

O fluxo de execução dos testes automatizados foi projetado para demonstrar como a integração entre o *Robot Framework*, a IA (API Gemini) e o *GitHub Issues* pode criar um ciclo de validação e análise contínua, tornando o processo de garantia da qualidade mais inteligente e autônomo.

3.5.1 Etapa 1: Inicialização do Teste

A execução se inicia no *Robot Framework*, que orquestra todos os componentes do processo. O ambiente de testes é configurado com as bibliotecas necessárias, como *SeleniumLibrary* (para controle do navegador), *RequestsLibrary* (para comunicação HTTP) e *AllureLibrary* (para geração de relatórios visuais). Em seguida, o *framework* abre o navegador definido e acessa a aplicação alvo — neste caso, o portal *The Internet Herokuapp* — para iniciar os cenários de teste funcionais.

3.5.2 Etapa 2: Execução das Ações e Validação

Durante os testes, o *Selenium* simula as interações de um usuário real, realizando ações como preencher formulários, submeter dados e validar respostas da interface. Caso a aplicação se comporte conforme o esperado, o teste é marcado como bem-sucedido e registrado no relatório de execução. No entanto, se uma inconsistência for identificada (como um erro de autenticação ou falha de carregamento), o *framework* aciona automaticamente o módulo de análise com IA.

3.5.3 Etapa 3: Análise Automática da Falha com IA

Quando um defeito é detectado, o erro é capturado e enviado à *API Gemini*, da Google. Essa API recebe como entrada o contexto do erro — incluindo a mensagem retornada, o nome do teste, o *commit* em execução e o autor — e gera uma resposta textual com: [É preciso deixar claro que o teste é disparado a partir de uma ação de *commit* do usuário]. Foi a primeira vez que isso foi citado aqui no projeto.

- três possíveis causas técnicas para a falha; [você está usando de forma intercambiada as palavras 'erro' e 'falha', confira se estão no contexto correto. Algumas anteriores já corrigi.]
- duas sugestões de ações de depuração rápida.

Essa análise automática permite identificar a origem do problema de forma imediata, reduzindo o tempo necessário para diagnóstico manual e fornecendo insights técnicos diretamente para a equipe de desenvolvimento.

3.5.4 Etapa 4: Registro da Falha no GitHub Issues

Após o retorno da análise da IA, o sistema utiliza a **API do GitHub** para registrar automaticamente uma *issue* no repositório do projeto. Este registro contém:

- título descritivo da falha;
- mensagem de erro capturada;
- diagnóstico gerado pela IA (causas e ações recomendadas);
- informações sobre o commit e o responsável pela execução.

Essa integração garante rastreabilidade entre o erro ocorrido, sua análise e o acompanhamento da correção, permitindo que o fluxo de trabalho entre QA e desenvolvimento se torne mais colaborativo e eficiente.

3.5.5 Etapa 5: Relatórios e Evidências

Por fim, os resultados completos são consolidados pelo **Allure Report**, que gera relatórios visuais contendo o status de cada teste, as evidências de execução (como capturas de tela) e o histórico das execuções anteriores. Dessa forma, é possível acompanhar a evolução da qualidade da aplicação ao longo do tempo.

Em síntese, o fluxo de execução dos testes combina automação funcional, análise inteligente e rastreabilidade contínua, demonstrando como a união entre **Robot Framework**, **Selenium**, **IA Gemini** e **GitHub Issues** pode criar um ecossistema de validação moderno, ágil e orientado à qualidade.

CHAPTER 4

Considerações Finais

4.1 Figuras

Rótulos de figuras e tabelas devem ser centralizados se tiverem até uma linha (Figura 4.1), caso contrário devem estar justificados e identados em ambas as margens, como mostrado na Figura 4.2. Essa formatação já é realizada automaticamente pela classe inf-ufg.

Os compiladores LaTeX provêem um mecanismo bastante simples para inclusão de figuras, o que pode ser feito com o auxílio de várias classes auxiliares (as mais comuns são `graphic` e `graphicx`). A classe `inf-ufg` usa o comando `\includegraphics`, da classe `graphicx`, para a inclusão de figuras e não é necessário você colocar a extensão do arquivo neste comando. Por exemplo, para a figura 4.1 os comandos usados foram:

```
\begin{figure}[htb]
 \centering
 \includegraphics[width=0.40\textwidth]{./fig/exemploFig1}
 \caption{Uma figura típica.}
 \label{fig:exemploFig1}
\end{figure}
```

Ao se usar o compilador LaTeX, as figuras podem estar nos formatos *eps* e *ps*. Ao se usar o PDFLaTeX, as figuras podem estar nos formatos *png*, *jpg*, *pdf* e *mps*. A classe `graphicx` também pode ser usada para a inclusão de figuras, nos formatos listados, ao se usar o PDFLaTeX. Os comandos necessários são os mesmos ao se incluir figuras ao se usar o compilador LaTeX. O uso do comando `\includegraphics` faz com com que PDFLaTeX procure primeiro por figuras com extensão *pdf*, depois *jpg*, depois *mps* e por último *png*. Aqui também não é necessário especificar a extensão do arquivo.

Para a inclusão das figuras 4.1 à 4.3 os comandos usados, tanto no LaTeX quanto no PDFLaTeX, seriam os mesmos. É claro que em cada caso devem estar

disponíveis as figuras nos formatos suportados por cada compilador. Por exemplo, para a inclusão da figura 4.3 foram usados:

```
 \begin{figure}[H]
  \centering
  \includegraphics[width=0.40\textwidth]{./fig/exemploFig3}
  \caption{Figura incluída no texto com a classe graphicx.}
  \label{fig:exemploFig3}
\end{figure}
```

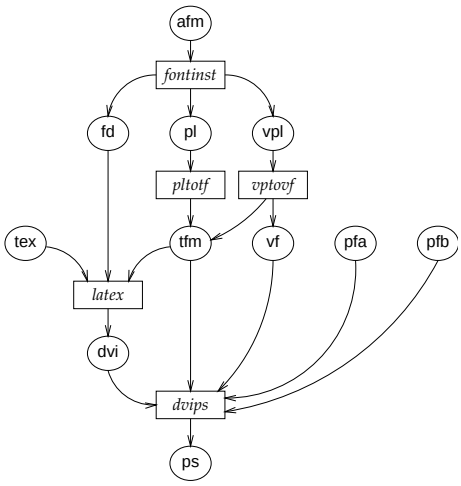


Figure 4.1: *Uma figura típica.*

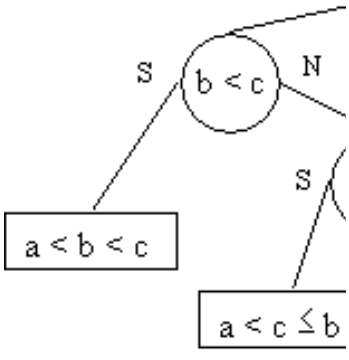


Figure 4.2: *Esta figura é um exemplo de um rótulo de figura que ocupa mais de uma linha, devendo ser identado e justificado.*

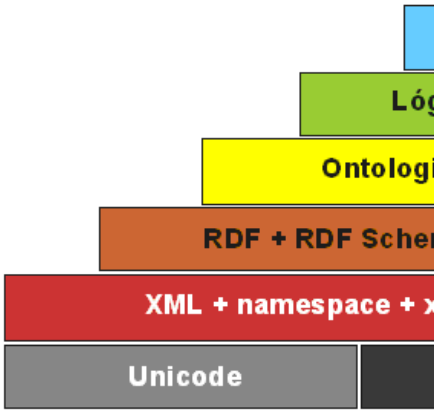


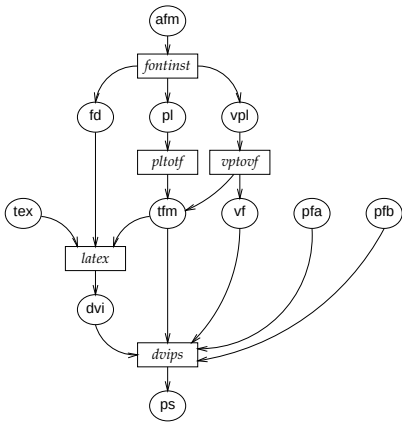
Figure 4.3: *Figura incluída no texto com a classe graphicx.*

4.1.1 Subfiguras

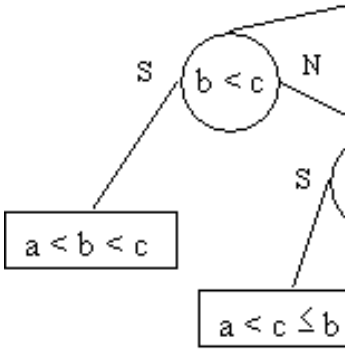
A classe `subfigure` pode ser usada para a inclusão de figuras dentro de figuras (consulte a documentação da classe para maiores detalhes). Por exemplo, a Figura 4.4 contém duas subfiguras. Estas podem ser referencidas por rótulos independentes, ou seja, podem ser referenciadas como Figuras 4.4(a) e 4.4(b) ou Subfiguras (a) e (b).

A figura 4.4 foi incluída com os comandos listados a seguir. Observe que há rótulos independentes para cada uma das subfiguras e um rótulo geral para a figura, os quais podem ser todos referenciados.

```
\begin{figure}[h]
 \centering
  \subfigure[Primeira subfigura.]
   {
    \includegraphics[width=0.35\textwidth]{./fig/exemploFig1}
    \label{subfig:ex1}
   } \qquad
  \subfigure[Segunda subfigura (um pedaço).]
   {
    \includegraphics[width=0.30\textwidth]{./fig/exemploFig2}
    \label{subfig:ex2}
   }
```



(a) *Primeira subfigura.*



(b) *Segunda subfigura (um pedaço).*

Figure 4.4: *(a) e (b) representam dois exemplos do uso de subfiguras dentro de uma única figura.*

```
  \caption{{\subref{subfig:ex1}} e {\subref{subfig:ex2}} representam
           dois exemplos do uso de subfiguras dentro de uma única
           figura.}
  \label{fig:subfiguras}
\end{figure}
```

Caso uma subfiguras não tenha rótulo, para evitar que o apenas o número da mesma apareça na Lista de Figuras, use o comando `\subfigure[][]`. Caso uma subfigura tenha rótulo e deseja-se evitar que a mesma apareça na Lista de Figuras, use o comando `\subfigure[][Rótulo]`.

4.2 Tabelas

Em tabelas, deve-se evitar usar cor de fundo diferente do branco e o uso de linhas grossas ou duplas. Ao relatar dados empíricos, não se deve usar mais dígitos decimais do aqueles que possam ser garantidos pela sua precisão e reprodutibilidade. Rótulos de tabelas devem ser colocados antes das mesmas (veja a Tabela 4.1).

4.3 Algoritmos

Algoritmos devem ser representados no formato do Algoritmo 4.1, que foi descrito com o uso da classe algorithm2e. A rigor não é obrigatório o uso dessa

classe, contudo o uso da mesma permite que seja gerada automaticamente uma lista de algoritmos logo após o sumário.

Algoritmo 4.1: $MSR(A, i, j)$

Entrada: vetor $A[i\,..\,j]$, inteiros não negativos i e j.
Saída: vetor $A[i\,..\,j]$ ordenado.

```
1 n ← j − i.
2 se (n < 4) então
3 |   Ordene com ≤ 3 comparações.
4 senão
5 |   Divida A em ⌈√n⌉ subvetores de comprimento máximo ⌊√n⌋.
6 |   Aplique MSR a cada um dos subvetores.
7 |   Intercale os subvetores.
8 fim
```

4.4 Códigos de Programa

Códigos de programa podem ser importados, mantendo-se a formatação original, conforme se pode ver no exemplo do Código 4.1. Este exemplo usa o ambiente codigo, definido na classe inf-ufg, que permite que uma lista de programas seja gerada automaticamente logo após o sumário.

Código 4.1 `insertionsort()`

```
1  void insertionSort( int* v, int n )
2  {
3    int i   = 0;
4    int j   = 1;
5    int aux = 0;
6
7    while (j < n)
8    {
9      aux = v[j];
10     i   = j - 1;
11     while ((i >= 0) && (v[i] > aux))
12     {
13       v[i + 1] = v[i];
14       i = i - 1;
15     }
16     v[i + 1] = aux;
17     j = j + 1;
18   }
19 }
```

4.5 Teoremas, Corolários e Demonstrações

O uso do ambiente theorem permite a escrita de teoremas, como no exemplo a seguir:

```
\begin{theorem}[Pitágoras]
Em todo triângulo retângulo o quadrado do comprimento
da hipotenusa é igual a soma dos quadrados dos
comprimentos dos catetos.
\end{theorem}
```

O resultado é o mostrado a seguir:

Teorema 4.1 (Pitágoras) *Em todo triângulo retângulo o quadrado do comprimento da hipotenusa é igual a soma dos quadrados dos comprimentos dos catetos.*

Da mesma forma pode-se usar o ambiente proof para demonstrações de teoremas:

```
\begin{proof}
Para demonstrar o Teorema de Pitágoras \dots
\end{proof}
```

Neste caso, o resultado é:

Prova. Para demonstrar o Teorema de Pitágoras ... □

Além desses dois ambientes, estão definidos os ambientes definition (Definição), corollary (Corolário), lemma (Lema)), proposition (Proposição), comment (Observação).

4.6 Citações Longas

Segundo as normas da ABNT, uma citação longa (mais de 3 linhas) deve seguir uma formação especial. Para tanto foi criado o ambiente `citacao`, o qual é baseado no ambiente de mesmo nome definido pelo grupo ABNTex (**??**):

> Uma citação longa (mais de 3 linhas) deve vir em parágrafo separado, com recuo de 4cm da margem esquerda, em fonte menor, sem as aspas (**??**, 4.4) e com espaçamento simples (**??**, 5.3). Uma regra de como fazer citações em geral não é simples. É prudente ler (**??**) se você optar for fazer uso freqüente de citações. Para satisfazer às exigências tipográficas que a norma pede para citações longas, use o ambiente citacao.

Este exemplo de citação longa foi produzido com o uso do ambiente `citacao`, como descrito logo a seguir:

```
\begin{citacao}
Uma citação longa (mais de 3 linhas) deve vir em parágrafo
separado, com recuo de 4cm da margem esquerda, em fonte menor,
sem as aspas \cite[4.4]{NBR10520:2001} e com espaçamento
simples \cite[5.3]{NBR14724:2001}. Uma regra de como fazer
citações em geral não é simples. É prudente ler
\cite{NBR10520:2001} se você optar for fazer uso freqüente
de citações. Para satisfazer às exigências tipográficas que a
norma pede para citações longas, use o ambiente citacao.
\end{citacao}
```

4.7 Referências Bibliográficas

Esta seção mostra exemplos de uso de referências bibliográficas com BibTeX e do comando `\cite`. Muitas das entradas listadas na página 44

foram obtidas de: http://liinwww.ira.uka.de/bibliography/index.html. Outro grande repositório de referências já em formato BibTeX está disponível em: http://www.math.utah.edu/ beebe/bibliographies.html.

As referências bibliográficas devem ser não ambíguas e uniformes. Recomenda-se usar números entre colchetes, como por exemplo (**??**), (**??**) e (**??**). O comando `\nocite` não produz texto, mas permite que a entrada seja incluída nas referências. Por exemplo, o comando `\nocite{Ber1970}` gera na lista de referências bibliográficas a entrada referente à chave Ber1970, mas não inclui nenhuma referência no texto. O comando `\nocite{*}` faz com que todas as entradas do arquivo de dados do BibTeX sejam incluídas nas referências.

Existem vários livros sobre LaTeX, como (**??????**), embora os mais famosos sejam sem dúvida (**??**) e (**??**). Para converter documentos LaTeX para HTML veja (**??**, pg.1–10).

Table 4.1: *Conteúdo do diretório (??)*

Tag	Comprimento	Início		Tag	Comprimento	Início
001	0020	00000		100	0032	00235
003	0004	00020		245	0087	00267
005	0017	00024		246	0036	00354
008	0041	00041		250	0012	00390
010	0024	00082		260	0037	00402
020	0025	00106		300	0029	00439
020	0044	00131		500	0042	00468
040	0018	00175		520	0220	00510
050	0024	00193		650	0033	00730
082	0018	00217		650	0012	00763

Referências

ARAUJO, L. da S. *Gerenciamento e Controle de Mudanças de Requisitos*. 2015. Disponível em <https://www.devmedia.com.br/gerenciamento-e-controle-de-mudancas-de-requisitos/32278>. Acesso em 18/04/2021.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The Art of Software Testing*. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.