CENTRO UNIVERSITÁRIO HERMINIO OMETTO
UNIARARAS

GUILHERME CAVENAGHI

Avaliação Comparativa de Algoritmos em Diferentes Linguagens de Programação: Impacto no Desempenho e Eficiência Computacional

Projeto de Trabalho de Conclusão de Curso apresentado ao curso de Engenharia da Computação da Uniararas, como parte dos requisitos para obtenção do grau de Bacharel.

Orientador: Prof. Renato Luciano Cagnin

ARARAS
2025

Resumo

Este trabalho tem como objetivo realizar uma análise comparativa de desempenho de algoritmos clássicos implementados em diferentes linguagens de programação. Com base nas linguagens mais utilizadas na indústria e na comunidade acadêmica, serão avaliadas métricas como tempo de execução, uso de memória e complexidade de implementação. A proposta visa identificar como a escolha da linguagem pode influenciar na eficiência computacional de uma aplicação.

Palavras-chave: algoritmos, linguagens de programação, desempenho, eficiência computacional.

1 Introdução

O desenvolvimento de software eficiente depende de diversos fatores, e entre eles destaca-se a linguagem de programação utilizada. Embora a lógica dos algoritmos possa ser universal, a forma como eles são implementados e executados varia significativamente entre as linguagens. Neste contexto, torna-se relevante compreender de que forma essa escolha impacta o desempenho e a eficiência computacional.

O conceito central deste trabalho é a análise comparativa entre diferentes linguagens, com foco em algoritmos clássicos. Parte-se do princípio de que, embora o algoritmo em si permaneça constante, o ambiente de execução, o modelo de compilação e a forma de gerenciamento de memória diferem entre linguagens, afetando diretamente os resultados. O problema (GAP) reside na carência de estudos empíricos que abordem essa comparação de maneira prática, considerando dados objetivos como tempo e memória.

A justificativa para esta pesquisa se baseia na crescente demanda por aplicações de alto desempenho e na necessidade de tomar decisões mais embasadas na escolha da linguagem, tanto na academia quanto na indústria. Entender esses impactos pode auxiliar desenvolvedores a optarem por ferramentas mais adequadas ao seu contexto de projeto.

A motivação deste trabalho surge da curiosidade em saber se determinadas linguagens, vistas como mais modernas ou mais populares, realmente oferecem vantagens práticas em termos de desempenho, ou se essa percepção está mais ligada a fatores culturais e de mercado do que técnicos.

Os objetivos desta pesquisa incluem: avaliar o desempenho de algoritmos clássicos implementados em diferentes linguagens; identificar os fatores técnicos que afetam esse desempenho; e oferecer uma síntese clara de quais linguagens se destacam em cada aspecto analisado.

Por fim, será apresentado um resumo dos resultados obtidos com os testes comparativos, destacando os pontos fortes e fracos de cada linguagem no contexto dos algoritmos utilizados.

2 Fundamentação Teórica

2.1 Algoritmos Clássicos

Algoritmos são sequências finitas de instruções bem definidas que, partindo de um estado inicial, visam resolver um problema ou realizar uma tarefa específica. Entre os algoritmos clássicos, destacam-se:

- **Algoritmos de ordenação**: como o QuickSort e o MergeSort, fundamentais para organizar dados de maneira eficiente.
- **Algoritmos de busca**: como a Busca Binária, que permite localizar elementos em listas ordenadas com complexidade logarítmica.
- **Estruturas recursivas**: padrões de implementação onde a solução de um problema envolve a chamada do próprio algoritmo em subproblemas menores.

2.2 Complexidade Computacional

A análise dos algoritmos considerados neste trabalho será baseada nas classes de complexidade da Teoria da Computação. Estas classes descrevem o comportamento dos algoritmos em termos de tempo e espaço exigidos conforme o tamanho da entrada cresce.

A classe **P** representa os problemas que podem ser resolvidos em tempo polinomial por uma máquina determinística. Já a classe **NP** representa os problemas cujas soluções podem ser verificadas em tempo polinomial, mesmo que não saibamos resolvê-los de forma eficiente.

Problemas **NP-completos** são os mais difíceis dentro da classe NP, e um algoritmo eficiente para qualquer um deles implicaria em uma solução eficiente para todos os problemas NP. Já os **NP-difíceis** não pertencem necessariamente à classe NP, mas são ao menos tão difíceis quanto os problemas NP-completos.

As definições e análises dos algoritmos implementados neste trabalho serão fundamentadas nessas categorias de complexidade, permitindo compreender não apenas a eficiência empírica, mas também os limites teóricos esperados para cada abordagem em diferentes linguagens de programação.

2.2.1 Exemplo de Algoritmo da Classe P

Um exemplo clássico de algoritmo da classe **P** é o **MergeSort**, um algoritmo de ordenação baseado no paradigma "dividir para conquistar", cuja complexidade é $O(n \log n)$. Este algoritmo sempre resolve o problema de ordenar uma lista em tempo polinomial.

2.2.2 Exemplo de Problema da Classe NP

Um exemplo de problema da classe **NP** é o **Problema de Satisfatibilidade Booleana (SAT)**. Dada uma fórmula booleana composta por variáveis e operadores lógicos, o objetivo é determinar se existe uma atribuição de valores que satisfaça todas as cláusulas. Embora verificar uma solução seja rápido, encontrar uma solução não possui, até o momento, algoritmo determinístico em tempo polinomial.

2.2.3 Exemplo de Problema NP-Completo

O **Problema do Caixeiro Viajante (TSP — Traveling Salesman Problem)** é um exemplo clássico de problema **NP-completo**. Nele, dada uma lista de cidades e as distâncias entre elas, busca-se o menor caminho possível que visite cada cidade exatamente uma vez e retorne à cidade de origem. Apesar de ser fácil verificar uma solução proposta, encontrar a melhor solução é um desafio computacional ainda sem solução eficiente conhecida.

2.2.4 Exemplo de Problema NP-Difícil

O **Problema da Parada (Halting Problem)** é um exemplo clássico de problema **NP-difícil**. Este problema consiste em determinar se um programa de computador, dado um input específico, terminará sua execução ou continuará indefinidamente. Alan Turing demonstrou que não existe algoritmo capaz de resolver esse problema para todos os casos possíveis, caracterizando-o como indecidível.

2.2.5 Resumo das Classes e Exemplos

A seguir, são apresentados exemplos representativos de algoritmos e problemas relacionados a cada uma das principais classes de complexidade computacional:

- **Classe P**: O **MergeSort** é um algoritmo de ordenação com tempo polinomial $O(n \log n)$. Ele resolve de forma eficiente o problema de ordenar uma lista de elementos.
- **Classe NP**: O **Problema de Satisfatibilidade Booleana (SAT)** consiste em determinar se existe uma atribuição de valores que satisfaça todas as cláusulas de uma fórmula booleana. Embora a verificação de uma solução seja rápida, encontrar a solução pode ser computacionalmente difícil.
- **Classe NP-completo**: O **Problema do Caixeiro Viajante (TSP)** busca o menor percurso que visita todas as cidades exatamente uma vez e retorna à cidade de origem. Embora seja fácil verificar se um percurso proposto é válido, encontrar o percurso ótimo é um problema notoriamente difícil.
- **Classe NP-difícil**: O **Problema da Parada (Halting Problem)** questiona se, dado um programa e uma entrada, é possível determinar se o programa terminará ou continuará executando indefinidamente. Alan Turing provou que este problema é indecidível, ou seja, não existe um algoritmo geral que resolva essa questão para todos os casos.

Classe	Exemplo e Descrição
P	**MergeSort**: algoritmo de ordenação com tempo polinomial $O(n \log n)$.
NP	**SAT**: verificar se uma fórmula booleana pode ser satisfeita.
NP-completo	**TSP**: encontrar o menor percurso que visita todas as cidades e retorna à origem.
NP-difícil	**Problema da Parada**: determinar se um programa finaliza sua execução; problema indecidível.

Tabela 2.1: Resumo das classes de complexidade e exemplos.

2.3 Linguagens de Programação

Serão consideradas as linguagens mais populares de acordo com rankings do TIOBE tiobe [2024], GitHub Octoverse octoverse [2022] e Stack Overflow Developer Survey stackoverflow [2024]:

- Python
- C
- C++
- Java
- JavaScript
- Go
- Rust
- TypeScript
- C#
- Kotlin

Estas linguagens foram escolhidas devido à sua ampla adoção na indústria e na comunidade acadêmica, proporcionando uma base representativa para a análise comparativa do desempenho dos algoritmos clássicos implementados neste trabalho.

3 Materiais e Métodos

3.1 Materiais

Para a realização deste estudo, foram utilizados os seguintes recursos:

- **Hardware**: Computador pessoal com processador multi-core, arquitetura x64, 16GB de memória RAM, e sistema operacional Linux Ubuntu 22.04.
- **Ambientes de desenvolvimento**:
  - Compiladores e interpretadores específicos de cada linguagem: *GCC* para C e C++, *OpenJDK* para Java, *Python 3.x*, *Node.js* para JavaScript e TypeScript, *Go Compiler*, *Rust Compiler*, *.NET SDK* para C# e *Kotlin Compiler*.
  - IDEs e editores: Visual Studio Code, IntelliJ IDEA e terminal de linha de comando.
- **Ferramentas de medição**:
  - Ferramentas internas das linguagens para medição de tempo (por exemplo, `time`, `System.nanoTime`, `performance.now`).
  - Monitoramento de uso de memória via `/usr/bin/time` e ferramentas nativas do sistema.

Os algoritmos selecionados para este estudo foram implementados de forma equivalente em cada linguagem de programação, respeitando suas características sintáticas, mas preservando a lógica algorítmica original.

3.2 Métodos

3.2.1 Seleção dos Algoritmos

Foram selecionados algoritmos clássicos amplamente reconhecidos na literatura de ciência da computação, com diferentes propósitos e níveis de complexidade:

- **Ordenação**: QuickSort e MergeSort.
- **Busca**: Busca Binária.
- **Estruturas Recursivas**: cálculo do fatorial e sequência de Fibonacci.

3.2.2 Implementação

Cada algoritmo foi implementado nas seguintes linguagens de programação: Python, C,C++,Java, JavaScript, Go, Rust, TypeScript, C# e Kotlin. As implementações buscaram manter a estrutura algorítmica o mais fiel possível, evitando otimizações específicas ou uso de bibliotecas externas que pudessem alterar significativamente o desempenho.

3.2.3 Procedimentos de Teste

Os algoritmos foram executados sobre conjuntos de dados sintéticos, com tamanhos variados para simular diferentes níveis de carga computacional:

- Pequeno: 1.000 elementos.
- Médio: 10.000 elementos.
- Grande: 100.000 elementos.

Cada teste foi repetido 10 vezes para reduzir o impacto de variações esporádicas do sistema. Os seguintes parâmetros foram registrados:

- **Tempo de execução**: medido em milissegundos.
- **Uso de memória**: medido em megabytes.
- **Facilidade de implementação**: avaliada qualitativamente com base na extensão e complexidade do código fonte.

3.2.4 Análise dos Resultados

Os dados coletados foram organizados em tabelas e gráficos para facilitar a análise comparativa. A avaliação focou em:

- Identificação da linguagem com melhor desempenho em cada categoria.
- Comparação da eficiência entre linguagens compiladas e interpretadas.
- Discussão sobre a relação entre características da linguagem (paradigma, tipagem, compilação) e o desempenho obtido.

Os resultados serviram de base para as conclusões acerca do impacto da escolha da linguagem de programação na eficiência computacional dos algoritmos clássicos.

4 Resultados Esperados

Espera-se identificar quais linguagens oferecem melhor desempenho para tarefas específicas e como características como paradigma, tipagem, compilação influenciam o comportamento dos algoritmos. Esses dados poderão auxiliar desenvolvedores e pesquisadores na escolha mais adequada de linguagem para projetos com requisitos de desempenho.

5 Justificativa

Com a crescente demanda por sistemas eficientes e responsivos, a escolha da linguagem de programação tornou-se um fator crítico no desenvolvimento de software. Embora existam inúmeros estudos sobre algoritmos, poucos se concentram em comparações práticas entre linguagens populares no contexto da eficiência computacional. Este trabalho se justifica pela necessidade de fornecer uma base empírica que auxilie desenvolvedores, estudantes e pesquisadores na tomada de decisões mais conscientes, considerando o impacto direto da linguagem no desempenho final das aplicações.

Referências Bibliográficas

octoverse. GitHub Octoverse 2022. https://octoverse.github.com/2022/top-programming-languages, 2022. Acesso em: abril 2025.

stackoverflow. Stack Overflow Developer Survey 2024. https://survey.stackoverflow.co/2024/technology, 2024. Acesso em: abril 2025.

tiobe. TIOBE Index for April 2025. https://www.tiobe.com/tiobe-index/, 2024. Acesso em: abril 2025.