# Análise Comparativa de Linguagens de Programação a partir de Problemas Clássicos da Computação

## *Comparative Analysis of Programming Languages from Classical Computer Problems*

**Rodrigo Duarte Seabra[1], Isabela Neves Drummond[2], Fernando Coelho Gomes[3]**

Universidade Federal de Itajubá, Minas Gerais – Brasil[1]

rodrigo@unifei.edu.br, isadrummond@unifei.edu.br, fcgomes.92@gmail.com

***Abstract.*** *In computational studies regarding programming languages, a common topic is which one is the most useful to a specific project or which language has a better performance when applied to certain software. This research aims to contribute towards this, presenting a qualitative and quantitative analysis about the Python, Java and C languages using the execution time, the amount of code lines, the file size, the time taken to produce the algorithm, the amount of documentation access and the number of native functions used. Six classical computer problems were implemented and, from the results, comparative analyses have been performed, which allowed drawing some conclusions about the use of each language selected for the study. The main result showed that the C language has the best performance among the three languages, while Python presented the worst time performance; yet it had a positive result in the other analyses. Finally, Java language did not have the best or worst results in any case. It was also noticeable that the time taken to produce the algorithm, which was analyzed using the Pomodoro technique, did not present concrete results.*

***Resumo.*** *Nos estudos computacionais envolvendo linguagens de programação, muito se discute sobre qual a melhor opção para se utilizar em um projeto específico ou qual linguagem possui melhor desempenho em certa aplicação. O trabalho proposto busca contribuir nesse sentido, apresentando uma análise qualitativa e quantitativa das linguagens Python, Java e C utilizando parâmetros de tempo de execução, quantidade de linhas de código, tamanho do arquivo, tempo de produção, quantidade de acessos à documentação e uso de funções nativas. Foram implementados seis problemas considerados clássicos na área de computação e, a partir dos resultados alcançados, foram realizadas análises comparativas que possibilitaram a obtenção de algumas conclusões sobre o uso de cada linguagem selecionada para estudo. O principal resultado obtido mostrou que a linguagem C possui o melhor desempenho entre as três linguagens analisadas, enquanto Python apresentou o pior resultado de tempo de execução, mas se destacou positivamente nos demais aspectos analisados. Finalmente, a linguagem Java não apresentou grandes resultados positivos ou negativos. Também foi possível observar que o parâmetro de análise de tempo de produção, em que se utilizou da técnica Pomodoro, não apresentou resultados concretos para análise.*

## 1. Introdução

Desde o advento do exercício da função de "programador" existe um fator importante que o define: a linguagem de programação. Outros fatores e qualidades também podem definir este profissional, mas o que normalmente desperta atenção no mercado profissional são as linguagens computacionais dominadas pelo programador. Com o avanço da tecnologia e da computação, este fator foi sendo cada vez mais evidenciado, pois, com a rápida criação e incorporação de novas linguagens, muitas se tornaram obsoletas, sendo necessário o aprendizado de linguagens mais atualizadas e adequadas às novas situações exigidas nas soluções computacionais dos variados problemas.

Considerando a diversidade de linguagens de programação atualmente disponíveis para uso, um profissional recém-ingresso no mercado de trabalho, ou até mesmo que esteja há mais tempo e deseja se atualizar, pode se deparar com um grande impasse: qual linguagem de programação deve ser utilizada ou aprendida agora? Mesmo conhecendo o objetivo e o paradigma de cada linguagem considerada como candidata a uso, existe uma grande dúvida em como se especializar e em qual delas se aprofundar. Na Internet, podem ser encontrados artigos que apresentam as linguagens que estão sendo mais utilizadas, ou qual delas escolher quando se é novo neste cenário; mas são raros os artigos que fazem uma análise mais detalhada dos problemas que uma linguagem pode apresentar durante seu uso e como esta pode se comportar perante algumas situações que lhes são apresentadas. Além disso, aqueles que desenvolvem suas aplicações em certa linguagem por muito tempo, normalmente se identificam tanto com ela que se estabelece uma "zona de conforto", acreditando-se fielmente que esta linguagem será a "bala de prata" (Sommerville 2007) na computação.

Para facilitar a seleção de qual linguagem de programação deve-se começar a aprender ou se aprofundar, é necessário ter conhecimento se ela é compatível com os projetos e situações nas quais será empregada, e se as soluções para os problemas que o programador possui interesse em resolver podem ser alcançadas por tal linguagem. Uma análise que envolva variadas áreas de problemas da computação e utilize parâmetros tais como a facilidade de codificação do problema, o tempo de execução da linguagem, a velocidade de solução do problema e o tamanho de código gerado são informações relevantes para um profissional ao selecionar determinada linguagem de programação. Ademais, a divisão por paradigmas, seguindo a classificação apresentada por Sebesta (2012), contribui nas decisões de modelagem da solução.

Com base no exposto, essa pesquisa tem como objetivo analisar, pelo método AHP (*Analytic Hierarchy Process*) e métricas propostas no estudo, códigos implementados nas linguagens de programação C, Java e Python. Os algoritmos produzidos nessas linguagens buscam solucionar problemas clássicos computacionais. A escolha de diferentes paradigmas de programação para a implementação dos algoritmos clássicos é o diferencial desta pesquisa, pois os variados problemas são comparados em função das métricas estabelecidas. Mais ainda, as situações selecionadas para fins de implementação se constituem em problemas geralmente utilizados como exemplos em livros didáticos explorados em cursos ou aulas de programação. Muitos dos problemas apresentados versam sobre abordagens matemáticas devido aos fundamentos da computação. Pode-se observar, por exemplo, que o estudo de algoritmos inicialmente era realizado por matemáticos; assim, a ciência da computação

pode ser identificada como uma parte dos estudos matemáticos. Mas também, é passível de argumentação que o estudo de algoritmos faz parte da computação, e estes são estudados e utilizados por matemáticos. A partir dessas duas afirmações e considerando o Teorema de Cantor-Bernstein-Schroeder ambas as áreas poderiam ser consideradas equipolentes, segundo Knuth (1974). Não é possível afirmar que, ao iniciar o estudo de uma nova linguagem de programação, todo profissional direcionará sua atenção às situações que envolvam implementações matemáticas. No entanto, dado o histórico da computação, é possível encontrar exemplos matemáticos sendo utilizados em várias explicações computacionais.

Após a obtenção das soluções, a análise foi realizada por meio de uma comparação entre as três linguagens selecionadas para o estudo, utilizando os seguintes parâmetros: (i) facilidade de codificação do problema; (ii) tamanho de código gerado; (iii) tempo de execução do algoritmo; e (iv) tamanho do arquivo que armazena o programa. Os dados que compõem o insumo para a análise foram obtidos durante a execução dos códigos de cada linguagem e serão apresentados e discutidos no decorrer do trabalho.

Apesar de a seleção da linguagem a ser utilizada em um projeto depender de variados fatores, este trabalho se propõe a realizar uma análise, não definitiva, de alguns aspectos subjetivos das linguagens tratadas no estudo com base em uma proposta de metodologia de análise. Assim, procurou-se apresentar uma metodologia diferenciada para a análise de linguagens por meio de algoritmos comuns a engenheiros e cientistas. O foco do estudo não se restringe aos *benchmarks* tradicionalmente disponíveis, já que estes se baseiam em consumo de recursos computacionais. Este trabalho tenta abordar o consumo de recursos humanos ao utilizar métricas mais subjetivas, tais como tempo de produção, uso de funções nativas, dentre outras. Desse modo, a comparação com outros *benchmarks* de tempo e consumo de recursos não se adequa a todas as métricas abordadas na análise proposta. Ademais, para que esta comparação possa ser realizada seria necessário produzir os códigos para analisar algumas métricas, todavia essas informações não foram encontradas em outros *benchmarks*. O uso do AHP e de métricas fora do escopo puramente computacional foram as propostas principais empregadas no estudo, ao tentar diminuir a subjetividade para uma análise quantitativa.

## 2. Referencial Teórico

Considera-se que a primeira linguagem de programação surgiu antes mesmo do primeiro computador moderno. Augusta Ada Byron, filha de Annabella Milbanke e do poeta Lord Byron, também conhecida como Ada Lovelace, é considerada, por muitos, como a primeira programadora da história. Em 1843, Lovelace publicou uma tradução do artigo de Luigi Menabrea sobre a Máquina Analítica de Charles Babbage. Nesta tradução, foram adicionadas muitas notas da própria tradutora, nas quais ela descreve comandos, passo-a-passo, para calcular os números de Bernoulli. Este, considerado o primeiro software, seria utilizado na Máquina Analítica de Babbage (Kim e Toole 1999). Este primeiro software idealizado por Ada foi considerado um programa de baixo nível. A primeira linguagem de alto nível reconhecida, chamada de Plankalkul, foi criada em 1942, pelo alemão Konrad Zuse, que também foi o idealizador, projetista e criador do primeiro computador eletromecânico (Guimarães 2012).

Já na década de 50, surgiu uma das primeiras tentativas bem-sucedidas de facilitar a programação. Anterior a este marco, a programação era muito custosa e suscetível a diversos erros, principalmente pelo fato de ser realizada utilizando perfuração de cartões e trocas mecânicas nos grandes computadores da época. O FORTRAN, criado por John Backus em 1954, teve como conceito inicial "aceitar uma formulação concisa de um problema em termos de notação matemática e produzir um programa" (Backus 1978). O nome FORTRAN deriva da junção das palavras FORmula e TRANslation (Miesel 2012). Desde a época da criação do FORTRAN, a programação vem se desenvolvendo de modo expressivo. Foram criados campos de pesquisa e estudo, tais como: a Inteligência Artificial, que foi implementada, inicialmente, na linguagem Lisp, de John McCarthy, dando início às linguagens funcionais; a programação estruturada que teve início com Haskell Brooks Curry e linguagens como ALGOL, Ada e Pascal; a programação modular que se derivou da programação estruturada; a programação orientada a objetos que se iniciou com a linguagem Simula 67, mas que foi realmente implementada como orientada a objetos com Alan Kay, na linguagem SmallTalk; e muitas outras que surgiram e continuam sendo utilizadas até hoje (Miesel 2012).

O paradigma imperativo define que um software deve se basear em mudanças de seu próprio estado e delegações de mudança de estado. Uma linguagem definida como imperativa possui, em seu cerne, a sequência e a execução de comandos. Para que um programa seja feito sob o paradigma imperativo, ou também conhecido como procedural, é necessário que este seja focado na pura resolução do problema. Este paradigma foi o primeiro concebido e despertou grande motivação, já que por volta da década de 50, eram procuradas formas de facilitar certos cálculos que possuíam uma solução sequencial (Baranauskas 1993). Por sua vez, o paradigma da orientação a objetos tem sua origem relacionada às pesquisas de Alan Kay no ramo de interfaces de computadores (Sebesta 2012).

As pesquisas de Kay também podem ser contextualizadas aos problemas da época, os quais estão relacionados às linguagens orientadas a procedimentos, como Pascal e C. As linguagens procedimentais geram dificuldade na produção de grandes softwares. Essas dificuldades, segundo Ziviani (2007), podem ser divididas em dois tipos: falta de coesão entre o software e o mundo real, e a estrutura do código produzido. A falta de coesão entre os elementos do programa e dos elementos responsáveis por regerem o mundo real e auxiliarem uma atividade fora do ambiente virtual não forma uma relação única entre o controle e o controlado. O modo como o programa escrito em uma linguagem procedimental é estruturado faz com que a comunicação entre elementos do próprio programa seja comprometida, já que qualquer elemento pode acessar as diversas partes e funções do software que o compõe (Ziviani 2007). Com base nesses problemas, foi criado o paradigma da orientação a objetos. Os conceitos de interface, objetos e classes foram criados por Kay no final da década de 60, sendo que a primeira linguagem a utilizar o paradigma foi a SmallTalk-72 em uma estação de trabalho da Xerox Alto, que foi baseada no Dynabook, projeto em que Kay desenvolveu a ideia de uma linguagem orientada a objetos (Sebesta 2012).

Um dos paradigmas mais recentes é o da linguagem *scriptada*. O conceito de *script* se iniciou como o ato de criar uma coleção de comandos executados pelo sistema e fazer com que o computador interpretasse o conjunto como um algoritmo.

Inicialmente, os comandos dentro de uma coleção eram tratados como simples chamadas de sistema. Utilizando esse conceito e adicionando variáveis, controladores de fluxo e gerenciadores de funções, foi criada a linguagem Shell (Sebesta 2012). A primeira linguagem Shell foi implementada na sétima versão do sistema UNIX, em 1979, e foi chamada de Bourne Shell, uma homenagem realizada ao seu criador, Steven Bourne. Esta não chegou a se popularizar, já que a linguagem ficou amplamente conhecida quando foi lançada a derivação C Shell, criada por Bill Joy (Newham e Rosenblatt 2005).

Em todos estes campos, e até mesmo na época de criação de cada paradigma de programação ou linguagem, sempre existiu a comparação de métodos e suas facilidades. As linguagens surgiram de maneira a suprir necessidades onde suas predecessoras não obtinham resultado, ou para apresentar novos conceitos que hoje são inerentes a qualquer linguagem de alto nível. Além de buscar novos meios de se programar e novas linguagens, sempre existiram tentativas de se criar parâmetros de comparação entre elas. Estes parâmetros não são bem estabelecidos até hoje, já que algumas linguagens foram criadas para fins tão específicos que elas somente possuem um resultado positivo em seu ambiente de foco. Contudo, mesmo com parâmetros que se aplicam somente a certas linguagens, ainda se podem aplicar análises mais específicas envolvendo parâmetros tais como tamanho de código, tempo de execução, tamanho do software, portabilidade etc., para se realizar uma comparação concisa e bem fundamentada.

## 2.1 Trabalhos Correlatos

No trabalho de Zapalowski (2011), o autor aborda variadas métricas quantitativas e qualitativas para comparar diferentes linguagens. Os parâmetros utilizados foram: quantidade de linhas de código, o tempo médio de execução de cada algoritmo e a eficiência relativa dos programas. Na pesquisa, as métricas de comparação possuem abordagem comercial e mercadológica, já que o autor do estudo demonstra o valor e a escolha de cada uma por meio de explicações voltadas a aplicações no mercado de trabalho. As linguagens que geraram dados de comparação foram C, C++, Java, PHP, Python e Ruby, sendo que elas foram indicadas como as escolhidas devido à popularidade no ramo de desenvolvimento e do conhecimento prévio do autor. Por meio dos resultados gerados na pesquisa, é possível visualizar as diferenças e as vantagens e desvantagens que o autor apresenta em sua análise final. A principal conclusão que pode ser vista na análise de Zapalowski é a extrema eficiência que as linguagens como C e C++ oferecem em termos de execução, enquanto as linguagens de mais alto nível, como Python e Ruby, comprovam ter uma eficiência próxima, mas com complexidade de desenvolvimento muito menor.

Utilizando-se de um único problema focado na busca e processamento de strings, Prechelt (2000) faz uma comparação de sete linguagens de programação utilizando diversos programas escritos por diferentes programadores em condições de trabalho e produção diversas. As linguagens utilizadas para produção de uma solução ao problema apresentado pelo autor foram: C, C++, Java, Perl, Python, Rexx e Tcl. Dada a explicação, pelo autor do artigo, que as comparações realizadas na maioria das vezes são muito generalizadas, ou devido ao baixo fator de comparação, muito limitadas, a pesquisa apresentada foca em aspectos mais específicos e busca chegar a um grande número de resultados e conclusões provenientes do estudo de somente um problema.

Como as soluções do problema proposto foram realizadas em diferentes linguagens e em diferentes circunstâncias, uma margem de erro foi adicionada às métricas e também foi considerado como um ponto plausível de estar presente nos resultados, já que o ambiente de desenvolvimento pode variar para cada desenvolvedor. A busca por resultados mais precisos fez com que a análise dos programas fosse mais aprofundada e cautelosa; diversos fatores tais como, consumo de memória, confiabilidade do código executado, tempo de execução total e tempo decorrido da execução de partes específicas e comuns aos programas foram tomados como base de comparação. Os resultados obtidos, juntamente da dissertação do seu significado, apresentam um conjunto de informações confiáveis e embasadas para que trabalhos futuros possam usufruir de uma métrica precisa e detalhada como a tomada pelo autor. Tais dados alcançados, por se basear em códigos de diferentes indivíduos, apresentaram uma grande diferenciação entre algoritmos tanto entre as linguagens como entre implementações da mesma linguagem. Este fator contribuiu para a conclusão que apesar do melhor desempenho em linguagens de "baixo nível" como C e C++, o maior consumo de memória e tempo que algumas outras linguagens podem requerer são aceitos devido ao fator de produtividade de desenvolvedores. Além disso, é possível concluir que a linguagem a ser caracterizada como pertencente ao paradigma de *script* não garante que seu desempenho será o pior e que seu tempo de produção será o melhor.

Na pesquisa apresentada por Miglioranza (2009), é conduzida uma abordagem de comparação de diversas linguagens em um ambiente mais específico, as plataformas de telefonia móvel. Além de definir um escopo de plataforma de menor abrangência, o autor apresenta como campo de estudo a manipulação de vídeo em tempo real nas plataformas por meio do sistema operacional e APIs (*Application Programming Interface*). As métricas abordadas têm grande peso no quesito quantitativo e qualitativo, visto que diversas métricas de caráter não quantificável foram selecionadas como parâmetro do estudo. Um exemplo de métrica qualitativa se refere à legibilidade, a qual se relaciona ao nível de proximidade da sintaxe da linguagem de programação à linguagem escrita comum. Na finalização da pesquisa é possível perceber uma análise minuciosa sobre os aspectos abordados e sobre os componentes de análise utilizados. Os diversos parâmetros analisados apresentam ao leitor boa quantidade de informações e demonstram alguns ângulos mais específicos das plataformas móveis. Os resultados obtidos por Miglioranza revelam que, apesar da melhor legibilidade, menor complexidade de produção e adaptação do Java à plataforma de implementação, no caso foram escolhidos dispositivos móveis, a linguagem C++ apresenta melhor desempenho e algumas funcionalidades que podem garantir maior controle sobre a aplicação. Uma análise final revela que a escolha de uma ou outra linguagem depende do projeto a ser aplicado, ou seja, tanto o desempenho da aplicação quanto o do desenvolvedor na implementação final devem ser levados em conta.

A comparação realizada por Barnabé (2010) é voltada para a área da Internet, ou seja, as linguagens utilizadas de base para comparação são geralmente utilizadas em servidores *web* ou *intraweb*. A comparação entre PHP (PHP Hypertext Processor), ASP (Active Server Pages) e JSP (Java Server Pages) traz uma boa abrangência e visão de linguagens de programação *web server side*. A abordagem ao analisar as sintaxes, preparação de ambiente de desenvolvimento e criação de um aplicativo utilizando um banco de dados simples, confere um aspecto didático ao trabalho. Fazendo uma comparação a partir dos requisitos para começar a trabalhar com uma linguagem até

como manipulá-la, apresenta um lado pouco explorado desta área. Outro ponto que se destaca, apresentado pela autora, é o de custo monetário para produção de software em dadas linguagens. Além disso, é realizado um estudo sobre a competência de cada plataforma em quesitos voltados a um servidor, tais como: concorrência, requisições bem-sucedidas e tempo de resposta. Os resultados apresentados por Barnabé indicam que, após diversos testes, a linguagem proprietária ASP foi a que obteve melhor resultado, apesar de algumas falhas de resposta, enquanto o PHP obteve o segundo melhor desempenho, com praticamente nenhuma falha. A autora também aponta que os resultados obtidos podem variar muito com o projeto e com os outros componentes necessários para a utilização de qualquer uma das três linguagens analisadas.

## 3. Tratamento e Codificação dos Problemas

São muitos os métodos para se analisar um algoritmo e uma linguagem. O código produzido pode ser analisado utilizando algumas técnicas, tais como a complexidade do algoritmo e seu comportamento assintótico. Outra abordagem consiste na utilização de técnicas baseadas em autômatos e a produção de gramáticas representativas para as linguagens a serem analisadas. O enfoque deste trabalho, no entanto, é a relação entre o código produzido, a linguagem e o programador. As métricas escolhidas e explanadas, a seguir, visam a verificar o impacto de cada variável na codificação em uma linguagem específica. Como dito por Ziviani (2007), sobre o que deve ser realizado após as análises e decisões em um projeto de um algoritmo, "neste momento o projetista tem que estudar as várias opções de algoritmos a serem utilizados, onde os aspectos de tempo de execução e espaço ocupado são considerações importantes", se for necessário selecionar uma linguagem para o projeto, as métricas apresentadas a seguir podem ser consideradas de grande importância.

### 3.1 Tempo de Execução

O tempo de execução de um código trata da velocidade com que o programa será executado, e em quanto tempo este apresentará uma resposta ou solução para o problema proposto. Em um projeto de software, a métrica relacionada ao tempo de espera por uma resposta é crucial, principalmente em sistemas de operação em tempo real, em que é necessário que as respostas sejam entregues em um tempo determinado e de forma consistente. Tanenbaum (2010) demonstra a necessidade do tempo de resposta ao falar sobre Sistemas Operacionais de Tempo Real (SOTR). É realizada a divisão entre sistemas de tempo real críticos e não críticos, onde o primeiro exige que falhas não ocorram, caso contrário as consequências são drásticas. A segunda categorização aceita, de forma não desejada, algumas falhas para a entrega de uma resposta. Em SOTR, como o próprio nome indica, o tempo é um parâmetro fundamental que os caracteriza, sendo que estes possuem prazos extremamente rígidos e que devem ser seguidos e mantidos para não se tornarem alvos de falhas. É muito comum ver SOTR críticos sendo aplicados em processos de linhas de montagens, enquanto os não críticos são implementados, em sua maioria, em sistemas multimídia.

Dada a importância da métrica, este trabalho visa apresentar o tempo de execução das soluções de cada problema e realizar esta medição por meio de cinco testes para verificar também a consistência temporal que a linguagem proporciona ao apresentar uma dada solução. A consistência está relacionada ao modo que a linguagem

se comunica com o sistema operacional onde é executada. Mesmo tendo algoritmos determinísticos, é possível que se observe uma diferença no tempo entre execuções. Para se obter o parâmetro de medição em questão foi selecionada a função Time, um método básico presente nas bibliotecas padrões do GNU/Linux. Esta ferramenta, ao ser utilizada, executa certo comando determinado pelo usuário, com seus devidos parâmetros, e quando o código executado é finalizado, três estatísticas de tempo são apresentadas, a saber: (i) **o tempo real de execução**, do início ao fim do programa; (ii) **o tempo de uso de CPU pelo usuário**, soma do *tms_utime* (tempo total de uso do processador) e *tms_cutime* (tempo total de uso do processador por tarefas filhas); (iii) **o tempo de CPU do sistema**, soma do *tms_stime* (tempo do processador utilizado a favor do processo) e *tms_cstime* (tempo do processador utilizado a favor de tarefas filhas). Cada um destes itens será um fator de comparação para a análise do tempo de execução do código.

## 3.2 Tamanho do Código

O tamanho do código gerado trata da quantidade de linhas necessárias para que o programa, criado em determinada linguagem de programação, solucione o problema proposto. Mesmo com a evolução que as linguagens de programação vêm sofrendo ao longo dos últimos 50 anos, a métrica utilizada na análise do tamanho de um código, também conhecida como "*lines of code*" (LOC), continua sendo utilizada amplamente por engenheiros de software. A origem deste parâmetro de análise data dos anos 60, quando o desenvolvimento de um software era dado em 90% pela escrita de código. Com a evolução do planejamento do desenvolvimento e das linguagens em si, este parâmetro foi se tornando cada vez mais impreciso. O surgimento de novos paradigmas, de novos métodos de programação e até mesmo de estilos de endentação específicos (como o Comma-first[1]*),* fazem com que o mesmo código possa ser escrito com mais ou menos linhas. Além de ser impreciso, este parâmetro de escalabilidade de software pode ser prejudicial nas decisões do projeto, tanto para a contratação de pessoas como na escolha de tecnologias a serem utilizadas. Somente utilizando este parâmetro não é possível chegar a nenhuma conclusão, já que não se pode mensurar a capacidade de um programador de acordo com a quantidade de linhas que ele já escreveu, assim como não é possível determinar se o projeto a ser implementado é mais barato por possuir menos linhas de código (Jones 2009).

No livro Software Engineering Best Practices (Jones 2009), apesar de apresentar que o International Software Benchmarking Standards Group (ISBSG) possui um grande número de software analisados, muitas empresas e programadores se restringem ao sistema LOC de análise. Por este motivo, este trabalho aborda o conceito de análise de quantidade de linhas de código. No entanto, considerando a afirmação de Jones sobre o conceito de contar linhas de código não ser mais um parâmetro relevante, o peso adotado para este parâmetro será reduzido.

## 3.3 Tamanho do Arquivo

O tamanho do arquivo que armazena o código está relacionado ao espaço em disco que certo algoritmo ocupará, para que armazene um modo de solucionar o problema

1 http://nomadev.com.br/comma-first-por-que-usar/

proposto. Apesar da grande evolução que a área de armazenamento digital vem sofrendo, o espaço ocupado por um software ou por um projeto sempre é avaliado quando se deseja desenvolver um novo programa. Diversas questões sobre esta métrica são abordadas, com grande peso de decisão, quando se envolve *backups*, distribuição e versionamento de projetos. Uma das áreas que corrobora para a compressão e utilização de pequenos arquivos que consigam manter a qualidade original é a da Computação Gráfica. São encontrados vários estudos e citações em relação aos algoritmos de compressão e redução de imagens digitais. Scuri (1999) cita que o motivo do uso destes algoritmos e do investimento em suas melhorias, no caso da computação gráfica, ocorre devido à necessidade de armazenar imagens que, se persistidas em sua forma pura, seriam quase que inviáveis de se manter em um disco rígido. Tanto esta citação como a de Lopes (2008), em "Cor e Luz", podem demonstrar como o espaço de armazenamento ocupado por um arquivo é algo considerável na computação. É valido também comparar que uma imagem não tem o mesmo peso de um arquivo de texto simples; mas se for contabilizado um projeto como todo, o qual pode possuir imagens, quanto menor o espaço ocupado pelos algoritmos, maior será a economia nos aspectos de armazenamento.

Para a medição de tamanho do código é levado em conta o arquivo final gerado para execução do programa. A medição de seu tamanho será realizada utilizando o gerenciador de arquivo Nemo[2], que possui uma opção de visualização da quantidade de *bytes* ocupados por cada arquivo. As bibliotecas padrões e arquivos do ambiente de produção não foram inseridos na métrica, dado que o ambiente de produção utilizado para desenvolvimento é o mínimo necessário para a linguagem de programação.

## 3.4 Facilidade de Codificação

O aspecto relativo à facilidade de codificação sempre é tratado quando se discute o uso de uma linguagem. Devido a sua subjetividade e dependência em opiniões previamente formadas e não em fatos, acaba-se gerando diversos conflitos no quesito de definição deste parâmetro de análise. Além da complexa forma de quantificação, existe a forma de interpretação sobre o que consiste a "facilidade de programar". O conceito abordado por este trabalho é o de uso da linguagem em um primeiro momento de desenvolvimento, já que o quesito de escalabilidade de sistemas necessita de uma estrutura básica maior e de problemas que permitam a demonstração de escalabilidade mais visível. Como forma de quantificar o parâmetro de facilidade de codificação são propostos três novos aspectos a serem analisados: (i) **tempo de produção**; (ii) **uso de funções nativas**; (iii) **acesso à documentação**. A análise realizada em cada um destes novos parâmetros, juntamente da combinação dos três, será tomada como base representativa à determinação da facilidade de se codificar um algoritmo em cada uma das linguagens estudadas.

## 3.5 Seleção dos Problemas

Visando uma seleção imparcial dos problemas a serem analisados, foi realizada uma investigação com diversos professores da área de computação, e uma busca em variados aspectos da literatura de computação. Foi decidido que a opinião de outros estudiosos da área seria de grande valia, assim como a busca de algoritmos capazes de serem

2 https://github.com/linuxmint/nemo

aplicados na solução de problemas práticos, sendo selecionados os seguintes problemas: Problema 1 – Cálculo de número fatorial de forma iterativa e recursiva; Problema 2 – Cálculo de números primos; Problema 3 – Multiplicação de matrizes; Problema 4 – Fatoração de um número em números primos; Problema 5 – O caixeiro viajante; Problema 6 – A mochila.

- Problema 1: foram implementadas duas formas de solução do problema para a comparação do desempenho de cada linguagem quanto ao tratamento de cada tipo de codificação;

- Problema 2: os códigos produzidos têm como foco descobrir os primeiros 1000 números primos. Após a obtenção dos números, é realizada uma validação do resultado, utilizando uma listagem já conhecida de números primos;

- Problema 3: O código envolvido no Problema 3 recebe as matrizes a serem multiplicadas por um arquivo de texto, previamente gerado, e após a leitura e inicialização de cada parâmetro a multiplicação é realizada. Foram escolhidas matrizes quadráticas de ordem 30 e para evitar que os números fossem escolhidos pelos autores da pesquisa, cada elemento da matriz foi determinado por um programa auxiliar que tem como objetivo gerar números aleatórios dentro da faixa -1000 a 1000;

- Problema 4: utiliza uma lista de números primos previamente definidos e, após a leitura de todos os valores, o programa realiza multiplicações entre elementos da lista para verificar se algum deles pode ser validado como fatoração de um número em números primos. O valor a ser fatorado é previamente definido e proveniente da multiplicação de elementos primos aleatórios;

- Problema 5: o código envolvido lê os nomes das cidades e uma matriz de conexões entre elas. A partir das distâncias entre cada cidade, representadas em uma matriz, são realizados os cálculos necessários para que um caminho que passe por todos os nós seja apresentado;

- Problema 6: a implementação definida nesta pesquisa foi a mochila binária, em virtude da complexidade e variedade dos algoritmos da mochila fracionária. O algoritmo produzido realiza uma verificação binária dos elementos que podem estar na mochila e define as combinações que podem ser realizadas, já que cada elemento a ser inserido na mochila possui uma característica própria. Para definir a entrada na mochila são analisados o peso e o valor de cada possível objeto.

## 3.6 Parâmetros Iniciais para a Aplicação do AHP

Algumas premissas foram selecionadas para que o AHP fosse aplicado na análise de cada um dos problemas propostos. Primeiramente, foi definido o uso de somente um nível do AHP, ou seja, os parâmetros de análise são tratados no mesmo nível com o intuito de mostrar que todos são parte primordial na escolha da linguagem. A Figura 1 apresenta um diagrama dos níveis de análise, de forma generalizada, para os problemas a serem verificados. Inicialmente, é apresentada a meta geral da análise: seleção da

linguagem a ser utilizada. No nível dois tem-se a apresentação dos critérios de análise; e, no último nível, têm-se os problemas que serão analisados.

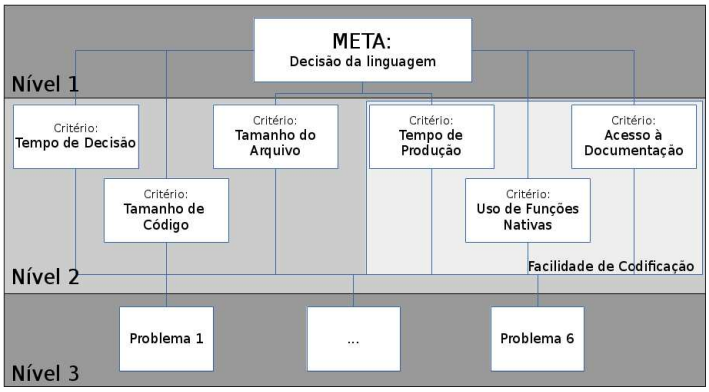


**Figura 1. Diagrama de níveis do AHP. Fonte: Os autores.**

A partir do diagrama de níveis foi selecionada a tabela de pesos, que define os pesos que serão inseridos nas tabelas de decisão a serem geradas. Existem diversas formas de se fazer uma tabela de pesos e variadas interpretações que podem ser seguidas. Como o método adotado é o AHP, foi selecionada a alternativa da escala relativa proposta por Saaty (1987). A Tabela 1 apresenta os pesos e relações escolhidas entre cada um dos parâmetros de análise. A definição desta tabela ocorre previamente à análise por questões do método AHP, que utiliza esta como uma matriz de pesos.

**Tabela 1. Relação de pesos para cada métrica analisada**

| | Tempo de execução | Tamanho de código | Tamanho de arquivo | Tempo de produção | Uso de funções | Acesso à documentação |
|---|---|---|---|---|---|---|
| **Tempo de execução** | 1 | 9 | 6 | 2 | 6 | 6 |
| **Tamanho de código** | 0,11 | 1 | 0,17 | 0,11 | 0,2 | 0,2 |
| **Tamanho de arquivo** | 0,17 | 6 | 1 | 0,2 | 4 | 4 |
| **Tempo de produção** | 0,5 | 9 | 5 | 1 | 6 | 6 |
| **Uso de funções nativas** | 0,17 | 5 | 0,25 | 0,17 | 1 | 1 |
| **Acesso à documentação** | 0,17 | 5 | 0,25 | 0,17 | 1 | 1 |

Os valores apresentados na tabela e definidos pelos autores do estudo, além de seguirem os padrões e significados apresentados por Saaty (1987)**Erro! Fonte de referência não encontrada.**, se valem de experiência pessoal e de pesquisas realizadas na definição de cada uma das métricas. Somente o parâmetro **tempo de execução** prevalece sobre todos os outros, já que este tende a ser um fator decisivo em projetos e que muitas aplicações são completamente dependentes do desempenho da execução e da velocidade de entrega de resposta. Contrariamente à ideia discutida sobre o parâmetro anterior, o **tamanho de código** teve seu uso e significado alterado ao longo dos tempos, fazendo com que, apesar de ainda utilizado, seu valor fosse reduzido. Já o **tamanho de**

**arquivo** gerado ao final da implementação, ficou muito próximo à igualdade com os demais, já que, apesar de relevante, o custo de armazenamento, na atualidade, vem sendo cada vez mais reduzido. O **tempo de produção** foi considerado como o segundo parâmetro mais importante, somente inferior ao **tempo de execução**. Essa consideração se deve aos mesmos quesitos de seu antecessor (**tempo de execução**), ou seja, devido a fatores de custo e de tempo o período que se leva em produção é muito relevante a decisões de projeto. Os outros dois parâmetros que, juntamente ao tempo de produção, incorporam as métricas subjetivas deste estudo, são os valores de **uso de funções nativas** e **acesso à documentação**; tais valores foram considerados de igual valia entre eles e foram adotados parâmetros da escala AHP bem próximos à igualdade, já que tanto o **acesso à documentação** quanto o **uso de funções nativas** podem ser diferenciais em um projeto, mas, dependendo da experiência de um programador, podem ser considerados praticamente irrelevantes.

## 4. Método

A fim de manter a máxima precisão possível nas medições, principalmente na medição de produtividade de codificação de cada um dos algoritmos, foi criado um roteiro que dividiu o processo de codificação em três etapas: **estudo dirigido**, **execução de validações** e **implementação**. As três atividades foram realizadas durante o período de uma semana para cada um dos problemas propostos. O tempo dedicado a cada conjunto de três algoritmos foi determinado com o propósito de uma análise técnica mais aprofundada dos autores da pesquisa sobre cada um dos problemas, para que assim, durante o período de implementação, fosse possível a apresentação de uma solução com maior embasamento e consolidação.

Durante a tarefa de **estudo dirigido**, foram realizadas diversas pesquisas sobre o problema, focando sempre em formas de implementação e métodos de aplicação do problema em questão. Os resultados obtidos foram utilizados para realizar o embasamento do código a ser validado, e da forma de solução a ser apresentada nas outras linguagens. O contato com implementações realizadas nas linguagens propostas por esta pesquisa foi evitado para prevenir possíveis influências. Foram analisadas, principalmente, soluções teóricas, ou seja, propostas de estruturas de solução para os problemas.

A **validação** foi utilizada como método de auxiliar o estudo dirigido. Como foram encontradas diversas soluções, foi necessária a escolha de uma forma de implementação. Utilizando a linguagem de programação PHP, a qual os autores têm conhecimento básico, foram feitos testes de estruturas de implementação a serem utilizadas. Esta validação foi realizada em uma linguagem de programação diferente das propostas no estudo para evitar, também, uma possível influência no processo de implementação.

Para o processo de **implementação** do problema foram seguidos alguns passos sempre que se dava início à implementação em uma das linguagens. Os passos seguidos visavam a maior eficiência e o menor número de interrupções possíveis durante esta tarefa. Nesse contexto, os passos seguidos foram: (i) iniciar todos os programas necessários para começar o desenvolvimento. No caso das três linguagens, foram utilizados o SublimeText3 e o gnome-terminal; (ii) preparação da documentação da

linguagem que seria utilizada naquele momento; (iii) preparação do ambiente de implementação. Em todos os casos, o ambiente consistiu em sua mesa de trabalho; (iv) preparação do contador Pomodoro[3] utilizando um aplicativo simples desenvolvido pelos autores (https://goo.gl/Alkatn) para o controle do tempo decorrido na implementação; (v) preparação de um gravador de áudio utilizando o aplicativo Voice Recorder. Foi realizada uma gravação do período de implementação para que quaisquer comentários relevantes pudessem ser transcritos no processo de análise; (vi) início da implementação.

A implementação de cada problema seguiu sempre a mesma ordem de linguagens, começando a tarefa com C, seguindo em Java e finalizando com Python. Além disso, foram efetuadas no mesmo dia, com intervalo de uma hora ao final de cada implementação. É importante salientar que durante o processo de implementação, outras métricas, além do **tempo de produção**, foram coletadas: (i) a métrica de **uso de funções nativas**, obtida por meio das notas de áudio do programador; (*ii*) o **acesso à documentação**, essa aquisição de dados se tornou possível em função das gravações realizadas durante a seção de codificação do algoritmo, já que a cada acesso da documentação foi realizada uma notação em áudio para marcar este evento. Ainda no ambiente de desenvolvimento, foram coletados os seguintes dados: (*i*) **quantidade de linhas**, após a finalização de todos os problemas foram abertos os arquivos fonte e anotadas as quantidades de linhas demarcadas pelo SublimeText3; (*ii*) **tamanho do arquivo**, utilizando o gerenciador de arquivos Nemo, tomou-se nota da quantidade de espaço ocupado por cada arquivo fonte.

Para realizar a coleta dos dados de tempo de execução necessários para a análise, foi gerada uma máquina virtual, utilizando o software VirtualBox, e preparado um ambiente de execução utilizando o Sistema Operacional (SO) Arch Linux. Esta distribuição Linux foi selecionada pela familiaridade dos autores e por se tratar de uma versão mais limpa, quando comparada no quesito de software pré-instalados. A versão do Arch Linux utilizada foi a 2015.08.01. Após a configuração básica do sistema, foram executados todos os algoritmos, previamente gerados, e coletados os dados. Para a aquisição do tempo de execução, utilizou-se a função *time* do sistema operacional. A escolha de um sistema diferente do utilizado diariamente pelo autor foi realizada como uma tentativa de ampliar a estabilidade entre as execuções. Almejando atingir um resultado mais significativo no período de execução dos algoritmos, estes foram executados repetidamente 10, 100 e 1000 vezes, sendo contabilizado o tempo médio total da execução, utilizando um *script* desenvolvido pelos autores, em cada conjunto de repetições. A diferença de tempo se torna mais clara quando abordada desta forma. Apesar de demonstrado o resultado de múltiplas execuções, este resultado não foi contabilizado juntamente do *framework* de análise, sendo apresentado somente para fins de melhoria da percepção dos resultados.

## 5. Análise

O problema de **cálculo de números fatoriais** foi dividido em iterativo e recursivo para as execuções, todavia, no período de produção do algoritmo, foi considerado como um único problema. A separação para execução teve como objetivo a verificação da

[3] http://pomodorotechnique.com/

diferença de tempo entre a versão recursiva e iterativa de um algoritmo muito comum na área da computação. Durante a produção do código, não houve acesso à documentação de nenhuma das linguagens. O número escolhido como teste de execução das implementações foi 20. Tal valor foi escolhido por limitações da simplicidade do algoritmo. O maior valor armazenado por uma variável em C, um *unsigned long long*, é extrapolado a partir do valor do fatorial de 21. Assim como em Java, este valor limite é atingido após o fatorial de 70. Essas características não são consideradas como falhas da linguagem, mas limitações da sua estrutura básica. No comparativo de linguagens, como Python está no nível mais alto, a estrutura de *overflow* de uma variável é tratada automaticamente, enquanto em C ou Java, deve-se preparar o código para estas eventualidades. Com isso, o limite máximo de uma estrutura simples em C foi escolhido para manter o código similar e para demonstrar como cada uma das linguagens trata seus tipos básicos.

Para o **cálculo de números primos**, durante a produção do código, não houve acesso à documentação de nenhuma das linguagens. Como parâmetro de execução e teste deste problema, os primeiros 1000 números primos foram calculados. No caso do problema de **multiplicação de matrizes**, durante a produção do código, foi realizado um acesso à documentação da linguagem C. Para os testes executados, foi gerada uma matriz quadrada de ordem 30 com valores aleatórios de -1000 a 1000. A matriz escolhida foi transcrita em um arquivo texto para que os mesmos dados fossem inseridos de forma mais simples para todos os algoritmos.

Para o problema de **fatoração em números primos**, durante a produção do código, não foi realizado qualquer acesso à documentação das linguagens. A lista contendo os 1000 primeiros números primos e o número 33770797 foram utilizados como dados de entrada para o algoritmo de fatoração em números primos. Este valor foi escolhido por representar a fatoração de dois números primos contidos na listagem. O método de escolha de quais valores da lista seriam utilizados foi baseado em um método aleatório, no qual foram gerados dois valores ente 1 e 1000, e, assim, selecionado o valor da lista referente ao resultado da metodologia de escolha. Durante a produção do código do problema do **caixeiro viajante**, não foi realizado qualquer acesso à documentação das linguagens. Para aplicação dos testes, foi gerado um arquivo contendo 15 cidades, nomeadas de A a U, e uma matriz de distâncias entre as cidades. A forma de entrada de dados matricial foi escolhida devido à solução desenvolvida e proposta. Para o último problema, **mochila booleana,** durante a produção do código, não foi realizado qualquer acesso à documentação das linguagens. Diferentemente dos algoritmos previamente apresentados, o problema da mochila tem seus valores de teste nativamente dentro do código. Sendo assim, foi selecionado um peso total para a mochila e gerados dois vetores, um referente ao peso e outro relativo ao valor de cada item. Foi escolhida uma capacidade total de 400 para a mochila e 22 itens com pesos e valores distintos.

A Tabela 2 apresenta os resultados numéricos obtidos para os parâmetros de tamanho de código, tamanho de arquivo e tempo de produção para cada um dos problemas e linguagens.

**Tabela 2. Resultados gerais para cada um dos problemas e linguagens**

| Problema | Linguagem | Tamanho de código (linhas) | Tamanho do arquivo (kB) | Tempo de produção (ciclos) |
|---|---|---|---|---|
| Cálculo de números fatoriais | C | 22 | 7,2 + 0,445 kB | 0,5 ciclo |
| | Java | 20 | 0,467 + 0,732 kB | 0,5 ciclo |
| | Python | 20 | 0,72 + 0,372 kB | 0,5 ciclo |
| Cálculo de números primos | C | 31 | 7,3 + 0,647 kB | 0,5 ciclo |
| | Java | 30 | 1 + 0,708 kB | 0,5 ciclo |
| | Python | 26 | 0,64 + 0,483 kB | 0,5 ciclo |
| Multiplicação de matrizes | C | 54 | 9,4 + 1,1 kB | 0,5 ciclo |
| | Java | 58 | 1,5 + 2,2 kB | 0,5 ciclo |
| | Python | 52 | 1,1 + 1,2 kB | 0,5 ciclo |
| Fatoração em números primos | C | 30 | 7,7 + 0,597 kB | 0,5 ciclo |
| | Java | 37 | 0,866 + 1,7 kB | 0,5 ciclo |
| | Python | 30 | 0,549 + 0,773 kB | 0,5 ciclo |
| Mochila booleana | C | 53 | 1,1 + 8,4 kB | 1,5 ciclo |
| | Java | 54 | 1,2 + 1,6 kB | 1,5 ciclo |
| | Python | 40 | 0,938 + 2 kB | 1,5 ciclo |
| Caixeiro viajante | C | 79 | 9,6 + 1,8 kB | 1 ciclo |
| | Java | 81 | 2,1 + 2,4 kB | 1 ciclo |
| | Python | 58 | 1,5 + 1,4 kB | 1 ciclo |

Para que a análise entre as linguagens seja realizada, é necessário que alguns valores referentes à matriz comparativa do AHP sejam calculados. A Tabela 3 mostra tanto os valores obtidos na matriz comparativa normalizada quanto a matriz comparativa original. Em seguida, a Tabela 4 expõe os valores referentes ao vetor de Eigen, ou seja, o valor de influência de cada um dos parâmetros de análise. Também é importante observar que o coeficiente de consistência (CR) obtido ficou abaixo de 10%, assim, segundo a teoria AHP, a tabela comparativa está apta para uso. O valor de Eigen apresentado na Tabela 4 se relaciona diretamente com a Tabela 3, dado que os valores referentes a cada um dos problemas são calculados utilizando os valores normalizados. Por fim, o coeficiente de consistência, obtido por meio dos valores de Eigen, demonstra o balanceamento da Tabela 4. Este coeficiente valida os pesos escolhidos na Tabela 3.

**Tabela 3. Normalização da tabela comparativa**

| | Tempo de execução | Tamanho de código | Tamanho de arquivo | Tempo de produção | Uso de funções nativas | Acesso à documentação |
|---|---|---|---|---|---|---|
| **Tempo de execução** | 1 | 9 | 6 | 2 | 6 | 6 |
| **Tamanho de código** | 0,11 | 1 | 0,17 | 0,11 | 0,2 | 0,2 |
| **Tamanho de arquivo** | 0,17 | 6 | 1 | 0,2 | 4 | 4 |
| **Tempo de produção** | 0,5 | 9 | 5 | 1 | 6 | 6 |
| **Uso de funções nativas** | 0,17 | 5 | 0,25 | 0,17 | 1 | 1 |
| **Acesso à documentação** | 0,17 | 5 | 0,25 | 0,17 | 1 | 1 |
| **Total** | 2,11 | 35 | 12,67 | 3,64 | 18,2 | 18,2 |
| **Tempo de execução** | 0,47 | 0,26 | 0,47 | 0,55 | 0,33 | 0,33 |
| **Tamanho de código** | 0,05 | 0,03 | 0,01 | 0,03 | 0,01 | 0,01 |
| **Tamanho de arquivo** | 0,08 | 0,17 | 0,08 | 0,05 | 0,22 | 0,22 |
| **Tempo de produção** | 0,24 | 0,26 | 0,39 | 0,27 | 0,33 | 0,33 |
| **Uso de funções nativas** | 0,08 | 0,14 | 0,02 | 0,05 | 0,05 | 0,05 |
| **Acesso à documentação** | 0,08 | 0,14 | 0,02 | 0,05 | 0,05 | 0,05 |

**Tabela 4. Valores de Eigen por comparativo e valor CR**

| Comparativos | Vetor de Eigen |
|---|---|
| **Tempo de execução** | 40,21% |
| **Tamanho de código** | 2,45% |
| **Tamanho de arquivo** | 13,73% |
| **Tempo de produção** | 30,37% |
| **Uso de funções nativas** | 6,62% |
| **Acesso à documentação** | 6,62% |
| **Total** | 100% |
| **CR ( < 10%)** | 9,5% |

Na comparação entre os dados de tempo de execução, é facilmente observado que a linguagem C obteve grande vantagem e resultados muito mais eficientes quando comparada a Python e Java. Não somente o tempo total de execução, mas como cada um dos tempos individuais são bem menores quando executados em C. Ao comparar Java e Python, individualmente, tem-se que a primeira linguagem somente se sobressai nos problemas que envolvem números primos e, apesar disso, o tempo final das execuções em Java é menor do que em Python. Analisando os resultados juntamente do método de implementação, a diferença entre os tempos pode ser ocasionada tanto pelo nível da linguagem quanto pela forma escolhida para implementar os problemas. O paradigma para qual a linguagem foi desenvolvida pode, muitas vezes, influenciar na forma em que o código é executado e compilado. Os tempos de execução de cada problema em cada uma das linguagens são apresentados na Tabela 5.

**Tabela 5 – Resultado do tempo de execução de cada linguagem.**

| TEMPO DE EXECUÇÃO | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | PROBLEMA | | | | | | | TOTAL |
| LINGUAGEM | P1 ITER | P1 REC | P2 | P3 | P4 | P5 | P6 | |
| C | 0,001 | 0,001 | 0,04 | 0,002 | 0,007 | 0,001 | 0,001 | 0,053 |
| JAVA | 0,074 | 0,06 | 0,132 | 0,1 | 0,081 | 0,068 | 0,067 | 0,582 |
| PYTHON | 0,049 | 0,05 | 0,457 | 0,049 | 0,153 | 0,032 | 0,026 | 0,816 |

Ao analisar a quantidade de linhas que cada implementação obteve, observou-se que, em ambos os casos, na comparação total e parcial, Python se sobressaiu. Assim como os resultados do tempo de execução, o paradigma sobre o qual a linguagem foi desenvolvida apresenta influência na quantidade de código necessária para executar uma ação. Um exemplo é Python, que não necessita de chaves para funções, métodos, blocos condicionais ou blocos de repetição, fazendo com que algumas linhas sejam economizadas. As quantidades de linhas de cada problema em cada uma das linguagens são apresentadas na Tabela 6.

**Tabela 6. Resultado do tamanho de código de cada linguagem**

| TAMANHO DE CÓDIGO (LINHAS) | | | | | | | |
|---|---|---|---|---|---|---|---|
| LINGUAGEM | PROBLEMA | | | | | | TOTAL |
| | P1 | P2 | P3 | P4 | P5 | P6 | |
| C | 22 | 31 | 54 | 30 | 53 | 79 | 269 |
| JAVA | 20 | 30 | 58 | 37 | 54 | 81 | 280 |
| PYTHON | 20 | 26 | 52 | 30 | 40 | 58 | 226 |

O tamanho de arquivo resultante, soma entre o código fonte e versão compilada, apresentou resultados nos quais os algoritmos em Python, com exceção do problema da mochila booleana, apresentaram resultado muito melhor. No geral, a diferença entre Python e Java não foi elevada, ao contrário da linguagem C, que se destoou das outras. O espaço ocupado pela versão compilada foi o maior diferencial entre as linguagens, dado que este fator depende do compilador. Os arquivos fonte não apresentaram muita diferença, visto que podem ser considerados como somente arquivos texto normais. Os tamanhos de arquivo de cada problema em cada uma das linguagens são apresentados na Tabela 7.

Para o tempo de produção, obtiveram-se algumas conclusões diferenciadas. Todos os resultados foram aproximadamente iguais quando analisados no quesito tempo real. O método utilizado foi o Pomodoro, que consiste em um conjunto de regras e técnicas para que seja ampliada a produtividade e qualidade do trabalho por meio de um melhor gerenciamento de foco e tempo. As regras são simples e objetivas, de fácil aplicação, além de apresentar resultados palpáveis de melhoria em projetos de software como discutido no estudo de Patrício, Macedo e França (2011). Para a aplicação do método, é necessário manter o foco, sem qualquer distração, em um trabalho a ser executado durante um ciclo de 25 minutos. Após um ciclo, deve-se interromper o trabalho por cinco minutos, e somente descansar. Ao concluir um conjunto de quatro ciclos de trabalho é recomendado um tempo de 30 minutos de "*long break*" para o descanso. Segundo os apoiadores do projeto, ao seguir este procedimento de execução de tarefas, há melhoria no foco, na agilidade de pensamento e, até mesmo, na saúde física.

**Tabela 7. Resultado do tamanho de arquivo de cada linguagem**

| TAMANHO DE ARQUIVO (kB) | | | | | | | |
|---|---|---|---|---|---|---|---|
| LINGUAGEM | PROBLEMA | | | | | | TOTAL |
| | P1 | P2 | P3 | P4 | P5 | P6 | |
| C | 7,65 | 7,95 | 10,5 | 8,3 | 9,5 | 11,4 | 55,29 |
| JAVA | 1,2 | 1,71 | 3,7 | 2,57 | 2,8 | 4,5 | 16,47 |
| PYTHON | 1,09 | 1,12 | 2,3 | 1,32 | 2,94 | 2,9 | 11,68 |

Ao utilizar a métrica e a técnica Pomodoro, foi possível perceber que os tempos se igualaram. Isso pode ser justificado por alguns fatores: (i) o estudo prévio de cada programa com uma definição de estrutura pode ter ocasionado uma normalizado do tempo de produção; (ii) os problemas selecionados para desenvolvimento podem não ser totalmente adequados para a métrica Pomodoro; (iii) a métrica utilizada, a técnica Pomodoro, por trabalhar em ciclos de 30 minutos, pode não ser a técnica mais precisa e adequada para aplicar nesta situação. Os tempos de produção de cada problema em cada uma das linguagens são apresentados na Tabela 8.

**Tabela 8. Resultado do tempo de produção de cada linguagem**

| TEMPO DE PRODUÇÃO (CICLOS) | | | | | | | |
|---|---|---|---|---|---|---|---|
| LINGUAGEM | PROBLEMA | | | | | | TOTAL |
| | P1 | P2 | P3 | P4 | P5 | P6 | |
| C | 0,5 | 0,5 | 0,5 | 0,5 | 1,5 | 1 | 4,5 |
| JAVA | 0,5 | 0,5 | 0,5 | 0,5 | 1,5 | 1 | 4,5 |
| PYTHON | 0,5 | 0,5 | 0,5 | 0,5 | 1,5 | 1 | 4,5 |

O uso de funções nativas também foi influenciado pelo paradigma e pelo nível da linguagem implementada. Pode-se observar que Python fez uso de um maior número de funções prontas. A linguagem C ficou em segundo lugar neste quesito, pois a leitura de arquivos em Java se utiliza de conceitos do paradigma orientado a objetos. Mas é interessante observar também o momento de uso das funções nativas. Um exemplo é a linguagem Python, que utiliza o método *range* para gerar um vetor de valores sequenciais e, assim, iterar entre esses valores para gerar um "*looping*". Os números de chamadas a funções nativas de cada problema em cada uma das linguagens são apresentados na Tabela 9.

A última métrica, acesso à documentação, foi a mais difícil de mensurar, já que esta depende da memória do programador e de sua aptidão com a linguagem. O resultado apresentado, somente um acesso à documentação da linguagem C durante a produção do problema de multiplicação de matrizes, ocorreu, pois, o método de leitura de caracteres de um arquivo não é tão trivial como em outras linguagens. Mesmo não realizando tantos acessos, alguns comentários sobre cada forma de acesso à documentação são válidos: (i) documentação C, utilizando um sistema Linux os acessos às funções e bibliotecas se tornam mais fáceis por meio da linha de comando; (ii) documentação Java, é necessário conhecer o local de instalação da documentação *off-line*. Apesar disso, o acesso utilizando o *browser* se torna bem intuitivo; (iii) documentação Python, a utilização do *shell* interativo torna o acesso à documentação simplificado como em C e ainda permite que testes sejam realizados mais facilmente.

**Tabela 9. Resultado final de chamadas de funções nativas de cada linguagem**

| FUNÇÕES NATIVAS | | | | | | | |
|---|---|---|---|---|---|---|---|
| LINGUAGEM | PROBLEMA | | | | | | TOTAL |
| | P1 | P2 | P3 | P4 | P5 | P6 | |
| C | 1 | 1 | 5 | 5 | 2 | 5 | 19 |
| JAVA | 1 | 1 | 3 | 3 | 1 | 3 | 12 |
| PYTHON | 2 | 3 | 5 | 5 | 2 | 5 | 22 |

Uma conclusão geral dos resultados é apresentada na Tabela 10. É possível observar que C obteve melhor desempenho somente no tempo de execução, mas o valor e o peso desta métrica fazem com que, dentre as análises realizadas, C tenha o melhor desempenho geral. Os resultados gerais de Python e Java obtiveram uma grande variedade. Enquanto Python se destacou nos demais parâmetros de análise, a linguagem obteve problemas de tempo, fazendo com que sua preferência fosse afetada. Apesar de Java somente apresentar pior desempenho na métrica relativa ao tamanho do código, os resultados da linguagem com relação ao tempo se aproximam mais dos resultados obtidos em Python.

**Tabela 10. Resultados por parâmetro de análise**

| | Prioridade | Melhor desempenho | Pior desempenho |
|---|---|---|---|
| **Tempo de execução** | 40,21% | C | Python |
| **Tamanho de código** | 2,45% | Python | Java |
| **Tamanho de arquivo** | 13,73% | Python | C |
| **Tempo de produção** | 30,37% | - | - |
| **Uso de funções nativas** | 6,62% | Python | C |
| **Acesso à documentação** | 6,62% | Python/Java | C |

## 6. Considerações Finais

A busca por melhor desempenho em todos os aspectos de produção sempre foi um atributo muito presente no desenvolvimento da tecnologia. Os desenvolvedores e apoiadores da tecnologia sempre almejam hardware e software que consigam atingir o máximo potencial disponível. Um aspecto claro nesse ponto é a comparação. Em todo ambiente em que se buscam melhorias, principalmente relativas a desempenho, faz-se necessário que dados, informações e conhecimentos sejam analisados de forma concorrente. Na área de desenvolvimento de software, uma das comparações que mais tende a ser realizada é a relacionada às linguagens de programação. Uma boa parte se deve ao custo de operação, desenvolvimento e manutenção que certa linguagem pode exigir.

Nesta pesquisa, os resultados obtidos para cada parâmetro analisado por linguagem demonstraram que apesar de não se destacar em nenhum dos outros parâmetros, somente no tempo de execução, a linguagem C seria a melhor opção a se escolher. Apesar de esta linguagem apresentar o pior resultado em tamanho de arquivo, e por se tratar de uma linguagem mais complexa, o que exigiria mais acessos à documentação e pouco uso de métodos nativos, seu desempenho seria de grande valia para um projeto. A linguagem Python, apesar de apresentar pior desempenho quando comparada às demais, seria uma boa escolha para implementações nas quais se necessita de um resultado quase que imediato. A escolha do Java se daria mais por conta de experiência e aptidão do desenvolvedor, já que seus resultados gerais não se demonstraram sobressair em nenhum dos aspectos. Além disso, é importante observar que linguagens de mais alto nível, como Python, tendem a apresentar estruturas básicas em um formato mais complexo. Um exemplo apontado neste trabalho foi o *overflow* de tipos, ou até mesmo listas encadeadas que, em Python, representam os vetores. Também pode-se observar uma correlação entre o nível de abstração de cada linguagem, a eficiência computacional e o parâmetro de facilidade de codificação. Apesar de a análise ser um pouco imprecisa, por utilizar medidas subjetivas, ainda sim é possível identificar que as linguagens de mais alto nível tendem a perder em desempenho por conta da abstração. Ou seja, quando se é realizada a decisão de abstrair um processo em uma linguagem de programação, pode-se resultar na queda de performance computacional juntamente do ganho no tempo de produção.

É importante observar também que os aspectos de afinidade, capacidade e experiência do programador podem influenciar na análise. Esta influência se deve ao caráter da análise, alguns parâmetros subjetivos e até mesmo as diferenças entre os paradigmas utilizados. Apesar disso, foi realizada uma tentativa de quantificar alguns parâmetros qualitativos de forma que, utilizando o AHP, fosse possível a realização de

uma decisão concreta quando se trata da seleção de linguagens de programação. Outro aspecto importante de ser observado é o fato de o consumo de memória volátil não ser considerado como parâmetro. O principal fator que levou a esta decisão, não incluir o consumo de memória como métrica, se relaciona ao fato que a complexidade dos problemas e as entradas escolhidas não proporcionaram resultados conclusivos.

Como conclusão final desta pesquisa, considerando os resultados e as avaliações realizadas, o objetivo de definição de linguagens de programação para projetos, apesar de ainda nebuloso, pode ser tratado de forma quantitativa e bem definido pelo AHP. Apesar de não se poder afirmar uma solução final para as decisões envolvendo linguagens de programação, este trabalho pode contribuir para futuros métodos de decisão ou, até mesmo, como forma de aquisição de dados sobre as linguagens abordadas.

## Referências


Bonardi, G.; Souza, V. B. A.; de Moraes, J. F. D. (2007). Incapacidade funcional e idosos: um desafio para os profissionais. *Scientia Medica*, v. 17, n. 3, p. 138-144.

Backus, J. The history of Fortran I, II, and III. In: *History of programming languages I*. ACM, p. 25-74, 1978.

Baranauskas, M. C. C. Procedimento, função, objeto ou lógica? Linguagens de programação vistas pelos seus paradigmas. *Computadores e Conhecimento: repensando a educação*. Campinas, SP, Gráfica Central da UNICAMP, 1993.

Barnabé, G. Um Estudo Comparativo entre as Linguagens de Programação PHP, ASP e JSP, 2010. 78f. *Monografia* (Especialização em Formação Didático-Pedagógica) – Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí, Rio do Sul, 2010.

Guimarães, G. *História da Computação*. 2012. Disponível em: <http://www.dsc.ufcg.edu.br/~pet/jornal/novembro2012/materias/historia_da_computacao.html>. Acesso em: 5 mai. 2016.

Jones, C. *Software Engineering Best Practices*. McGraw-Hill, Inc., 2009.

Kim, E. E.; Toole, B. A. Ada and the first computer. *Scientific American*, v. 280, p. 76-81, 1999.

Knuth, Donald E., Computer science and its relation to mathematics. *The American Mathematical Monthly,* v. 81.4, p. 323-343, 1974.

Lopes, J. M. B. *Cor e Luz*. Departamento de Engenharia Informática. Instituto Superior Técnico. Lisboa, 2008.

Miesel, D. *FORTRAN The Pioneering Language*. 2012. Disponível em: <http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/fortran/>. Acesso em: 10 mai. 2016.

Miglioranza, E. F. Estudo Comparativo de Linguagens de Programação para Manipulação de Vídeo em Telefones Móveis, 2009. 78f. *Trabalho Final de Graduação* (Graduação em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2009.

Newham, C.; Rosenblatt, B. *Learning the bash shell: Unix shell programming*. O'Reilly Media, Inc., 2005.

Patrício, R. G; Macedo, N. C. C.; França, C. T. P. L. Pomodoro aliado a SCRUM para aumento da produtividade: um estudo de caso, *INFOBRASIL*, Fortaleza, 2011.

Prechelt, L. An Empirical Comparison of Seven Programming Languages. *Computer*, v. 33, n. 10, p. 23-29, 2000.

Saaty, R. W. The analytic hierarchy process – what it is and how it is used. *Mathematical Modelling*, v. 9, n. 3, p. 161-176, 1987.

Scuri, A. E. *Fundamentos da Imagem Digital*. Pontifícia Universidade Católica do Rio de Janeiro, 1999.

Sebesta, W. R. *Concepts of Programming Languages*. 10th edition, New Jersey: Pearson Education Inc., 2012.

Sommerville, I. *Software Engineering*. 8th edition, England: Pearson Education Limited, 2007.

Tanembaum, A. S. *Sistemas Operacionais Modernos*. 3a. edição, São Paulo: Pearson Education Inc., 2010.

Zapalowski, V. Análise Quantitativa e Comparativa de Linguagens de Programação, 2011. 45f. *Trabalho Final de Graduação* (Graduação em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011.

Ziviani, N. *Projeto de Algoritmos com Implementações em Java e C++*. 1ª. edição, São Paulo: Cengage Learning, 2007.