Um Estudo Comparativo do Desempenho de Linguagens de Programação utilizando Técnicas Computacionais aplicadas ao Problema do Caixeiro-Viajante

Rodrigo M. Viana[1], **Bruno N. Gomes**[1], **Carlos A. Silva**[1]

[1]Departamento de Informática – Instituto Federal de Minas Gerais (IFMG)
CEP 34.590-390 – Sabará, MG – Brasil

rodrigomalaquias5@gmail.com, {bruno.nonato,carlos.silva}@ifmg.edu.br

Abstract. *The Traveling Salesman Problem (TSP) is a classic problem in Operations Research that arises in various practical contexts, such as vehicle routing problems, drilling printed circuit boards, maintenance of gas turbine engines, genome sequencing, among others. The effectiveness of the programming language used to address the TSP can directly influence the quality of the solution. This study aimed to analyze the performance of widely recognized programming languages in both academic and professional settings, including C, Python, C#, and Julia. Both heuristic and metaheuristic techniques were employed to solve the optimization problem represented by the TSP. The results demonstrate that the choice of language and technique directly impacts the final execution, where languages like C and Julia stood out significantly, with results that were at least twice as fast as those achieved with the other languages, and this advantage increased as the problem complexity grew.*

Keywords: *Traveling Salesman Problem, Programming languages, Optimization, Heuristic techniques, Performance analysis.*

Resumo. *O Problema do Caixeiro Viajante (PCV) é um clássico problema da Pesquisa Operacional que surge em diversos contextos práticos, como problemas de roteamento de veículos, perfuração de placas de circuito impresso, manutenção de motores de turbina a gás, sequenciamento de genoma, entre outros. A eficácia da linguagem de programação utilizada para abordar o PCV pode influenciar diretamente a qualidade da solução. Neste trabalho buscou-se analisar o desempenho de linguagens de programação amplamente reconhecidas no âmbito acadêmico e profissional, sendo elas C, Python, C# e Julia. Utilizaram-se tanto técnicas heurísticas quanto metaheurísticas para resolver o problema de otimização representado pelo PCV. Os resultados mostram que a escolha da linguagem e a técnica afeta diretamente na execução final, onde linguagens como C e Julia se destacaram significativamente, apresentando resultados que eram, no mínimo, duas vezes mais rápidos do que os obtidos com as demais linguagens, com essa vantagem aumentando à medida que o problema se tornava mais complexo.*

Palavras-chave: *Problema do Caixeiro Viajante, Linguagens de programação, Otimização, Técnicas heurísticas, Análise de desempenho.*

1. Introdução

Problemas de otimização estão frequentemente presentes em atividades cotidianas, como determinar o menor caminho para chegar ao trabalho ou escolher produtos em um supermercado de forma a suprir a necessidade de consumo ao menor custo possível. Em maior escala, esses problemas podem ser observados na área de logística, como na determinação de rotas de transportadoras e na minimização de custos de fábricas, entre outros. A busca pela eficiência e aprimoramento dos processos de otimização tem sido tema de inúmeras pesquisas acadêmicas, sobretudo por sua aplicabilidade em setores de grande relevância econômica e social.

Problemas combinatoriais comumente estão inseridos na classe de problemas denominados **NP** (*Non-Deterministic Polynomial Time*), ou seja, problemas que são **decidíveis** em tempo polinomial por uma máquina de Turing não-determinística, porém, em geral, não existe algoritmo para **resolver** o problema em um tempo que seja polinomial em relação ao tamanho da entrada. Entre os problemas mais clássicos da literatura de otimização combinatória está o Problema do Caixeiro-Viajante (PCV) [Biggs et al. 1986]. Trata-se de um problema que tem o objetivo de encontrar a rota mais curta ou econômica entre um conjunto de pontos (por exemplo, cidades) que devem ser visitados. O PCV consiste na visita única a cada ponto em um conjunto, sendo esses pontos caracterizados como cidades que um viajante pretende percorrer no menor caminho possível. Este problema é amplamente reconhecido no âmbito da matemática e pesquisa operacional, sendo classificado como um problema NP-difícil. O termo "NP" na teoria da complexidade computacional denota problemas que pertencem à classe de tempo polinomial não determinístico, indicando que obter resultados ótimos exigiria um tempo considerável, tornando impraticável uma solução rápida e eficiente.

A otimização, enquanto processo de busca por soluções ótimas ou subótimas para problemas complexos, contribui significativamente para o avanço científico e tecnológico. No contexto da computação, a escolha da linguagem pode ser um fator determinante na eficácia das técnicas de otimização. A literatura Palomares (2015) e Mendes (2016) indica que a otimização não apenas agiliza processos computacionais, mas também abre novas possibilidades para a criação de soluções inovadoras, impulsionando o desenvolvimento em diversas áreas do conhecimento.

Diversas técnicas computacionais são empregadas na tentativa de solucionar o PCV. Destacam-se entre elas as heurísticas e metaheurísticas, como o Vizinho Mais Próximo, Método da Descida, e técnicas avançadas como ILS (*Iterated Local Search*) e GRASP (*Greedy Randomized Adaptive Search Procedure*). Essas abordagens são amplamente utilizadas no processo de busca local, visando encontrar e refinar soluções. Cada uma dessas técnicas oferece uma perspectiva única para abordar o desafio, proporcionando diferentes formas de explorar o espaço de soluções e contribuindo para a diversidade de métodos aplicados na resolução do PCV.

O trabalho se propõe a explorar como diferentes linguagens de programação influenciam a eficácia na resolução de problemas de otimização, concentrando-se no PCV. A pesquisa visa explicar fatores que influenciam na escolha das linguagens de programação e como as linguagens afetam a eficiência, proporcionando perspectivas importantes para o aperfeiçoamento de estratégias de otimização, conforme evidenciado em estudos anteriores [Palomares 2016, Mendes 2015].

No contexto de buscar soluções eficientes para a classe de problemas de otimização combinatória e métodos computacionais aplicados na tentativa de solucionar esses problemas, os principais conceitos e desafios na área de problemas de otimização combinatória são discutidos no trabalho [Peres and Castelli 2021].

2. Referencial Bibliográfico

A literatura destaca a importância do PCV como um problema clássico na teoria da otimização combinatória, sendo amplamente estudado devido à sua complexidade e relevância. Além disso, estudos focados na influência das linguagens de programação na eficiência de algoritmos de otimização, revelam que a escolha da linguagem pode impactar significativamente o desempenho e a eficácia das abordagens computacionais. na solução de problemas combinatórios.

2.1. Problemas de otimização

Em virtude da complexidade dos problemas de PCV de grande porte, que são aquelas com um número expressivo de pontos a serem visitados, torna-se essencial o uso de métodos heurísticos na tentativa de solucioná-los. Em certos casos, pode-se optar por métodos exatos, mas estes podem se tornar impraticáveis devido ao aumento exponencial das combinações possíveis.

2.2. Desempenho de linguagens de programação

A escolha da linguagem de programação tem um impacto significativo no desempenho dos algoritmos e na eficiência da busca de resultados. Diferentes linguagens possuem características e estruturas distintas que afetam como os algoritmos são executados. Fatores como a eficiência de compilação, o gerenciamento de memória, e as bibliotecas disponíveis podem influenciar a velocidade e a eficácia dos processos computacionais.

Ao discutir sobre linguagens, é encontrado duas categorias principais: linguagens compiladas e interpretadas. Ambas convertem o código para binário, que é a linguagem compreensível para o computador. As linguagens compiladas traduzem todo o código de uma só vez, evitando a repetição de processos, o que resulta na maioria das vezes em uma execução mais rápida. Por outro lado, as linguagens interpretadas realizam essa tradução gradualmente. Por exemplo, um estudo de [Aruoba and Fernández-Villaverde 2017] revelou que o código Python, quando executado no interpretador CPython padrão, é entre 155 e 269 vezes mais lento que o código em C++, destacando a importância da escolha da linguagem quando se busca eficiência na resposta.

Busca-se comparar a eficiência de diferentes linguagens de programação executando algoritmos em diferentes cenários, variando o tamanho das instâncias de entrada. Foram escolhidas quatro linguagens de programação para executar o mesmo algoritmos. A seguir, estão as linguagens selecionadas para este estudo:

- Python, considerada por muitos uma das linguagens mais populares e influentes na programação atual devido à sua simplicidade e versatilidade, destaca-se por sua simplicidade e versatilidade. Estas características, que facilitam a curva de aprendizado e permitem a escrita de código conciso.
- C, criada por Dennis Ritchie em 1972, é reconhecida e tem seu lugar de destaque no mundo da programação até os dias de hoje, a linguagem C permite o desenvolvimento de aplicações de alto desempenho em uma variedade de contextos.
- C#, escolhida devido à sua maturidade e credibilidade. Com vários anos de presença no mercado e sua facilidade de aprendizado [Hsu 2012]. Além disso, a linguagem através de sucessivas atualizações e aprimoramentos em resposta às demandas do mercado e da comunidade de desenvolvedores, resulta em uma evolução ao longo do tempo. Vale ressaltar que o C# é originário da Microsoft, uma empresa de renome no mundo da tecnologia.
- Julia, é a mais recente das quatro linguagens. Ela foi desenvolvida em 2009 por um grupo de quatro pessoas com o objetivo de atender a processos que requerem alto desempenho numérico e científico [Edelman et al. 2023].

A escolha da linguagem de programação é fundamental em qualquer projeto de desenvolvimento para uma solução computacional, uma vez que diferentes linguagens possuem características e funcionalidades variadas que as tornam mais ou menos adequadas para determinados tipos de trabalho. Em projetos científicos, como os de pesquisa operacional, a escolha da linguagem de programação é crucial, pois a precisão, a estabilidade e a velocidade dos resultados são requisitos fundamentais.

Entre as linguagens de programação mais utilizadas em projetos de computação científica, é citado que C, C#, Python e Julia. C é uma linguagem de programação de baixo nível, conhecida por sua eficiência e velocidade de reposta, tornando-a adequada para projetos que exigem alto desempenho e que trabalham com grandes volumes de dados. C#, embora seja considerada de alto nível, também oferece desempenho robusto e é utilizada em diversos projetos científicos. Já Python e Julia são linguagens de programação de alto nível, que, à primeira vista, parecem mais simples devido ao uso de um conjunto enxuto de palavras reservadas e à possibilidade de escrever menos linhas de código para resolver tarefas e uma grande variedade de bibliotecas que facilitam o trabalho com dados científicos [Bezanson et al. 2017, Oliphant 2007].

Um exemplo de projeto científico que utilizou a linguagem C é o trabalho de [Breuer and Ziegler 2018], no qual os autores desenvolveram um modelo computacional para simulação de fluxo sanguíneo em válvulas cardíacas. O uso da linguagem C nesse projeto destaca sua capacidade de lidar com grandes volumes de dados e oferecer um alto controle sobre a execução do código.

Por outro lado, Python é uma das linguagens de programação mais utilizadas em projetos científicos atualmente. Um exemplo de seu uso é a biblioteca NumPy, amplamente empregada para computação científica e análise de dados. NumPy é frequentemente referenciada em trabalhos acadêmicos, como demonstrado em um estudo sobre suas aplicações na pesquisa científica [Harris et al. 2020].

Já a linguagem Julia tem ganhado destaque nos últimos anos, especialmente em projetos científicos que exigem alto desempenho. Um exemplo notável é o trabalho de [Koolen and Deits 2019], que explora a aplicação de Julia em simulações e controle em tempo real para robótica. Os autores optaram por Julia devido à sua capacidade de lidar com grandes volumes de dados e por ser uma linguagem de programação de alto nível que oferece um ótimo desemepenho.

Em resumo, a escolha da linguagem de programação é fundamental em projetos científicos, pois cada linguagem oferece características únicas que a tornam mais adequada para determinadas tarefas.

3. Heurísticas e metaheurísticas implementadas

Nesta seção são apresentadas as estratégias de resolução do PCV, com foco em heurísticas e metaheurísticas. A seção inclui a forma de representação da solução, heurísticas construtivas, heurísticas de busca local, estruturas de vizinhança e metaheurísticas. Inicialmente, é discutida a representação da solução, seguida pelos métodos construtivos, como o Vizinho Mais Próximo, e pelas metaheurísticas de busca local e de vizinhança variável. A análise abrange tanto a implementação quanto a avaliação dos resultados, permitindo comparar a eficácia de cada abordagem.

3.1. Representação da solução

O desafio do PCV envolve N pontos a serem visitados, sem repetição em nenhum deles, exceto pelo retorno final ao ponto de origem, visando minimizar o custo total gerado pelos deslocamentos.

Na codificação desta solução, foi escolhida a representação por meio de um vetor de $N+1$ posições, sendo N o número de nós na instância do problema. Cada uma das $N+1$ posições contém um valor inteiro no intervalo $[0, N]$.

3.2. Método de construção - Vizinho mais próximo

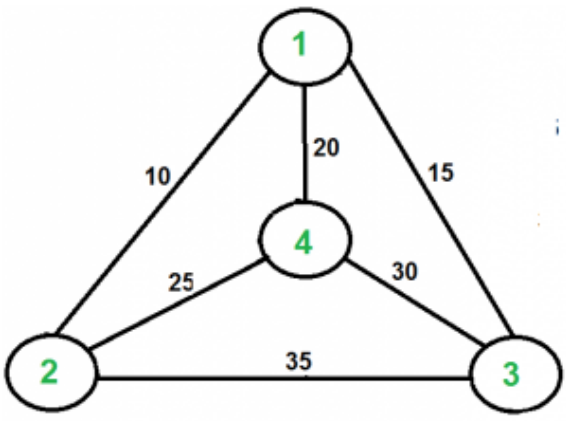
Heurísticas construtivas são métodos que visam construir uma solução inicial para um problema, a qual pode ser posteriormente refinada por métodos de busca local. O método do vizinho mais próximo é um exemplo de heurística construtiva. Como mostra o Algoritmo 1, a estratégia é escolher o melhor vizinho do ponto em análise e considerar o mesmo como o próximo caminho, repetindo esse processo até que todos os pontos tenham sido visitados e retornando ao ponto inicial.

Algoritmo 1 Vizinho mais próximo

1: Seja $i \leftarrow 1$ o vértice inicial do grafo
2: $V \leftarrow \{2, 3, ..., n\}$
3: **enquanto** $V \neq \emptyset$ **faça**
4: Seja $j \in V$ um vértice tal que $c_{i,j}$ é mínimo
5: Inserir a aresta (i, j) na solução
6: $V \leftarrow V \setminus \{j\}$
7: $i \leftarrow j$
8: **fim enquanto**
9: Inserir a aresta $(i, 1)$ na solução

A Figura 1 ilustra um exemplo de funcionamento do vizinho mais próximo para uma instância com 4 nós, que podemos exemplificar a técnica, considerando o ponto 1 como inicial, o algoritmo seleciona o vizinho mais próximo não visitado até que todos os pontos tenham sido visitados, e finalmente retorna ao ponto inicial, formando assim um ciclo fechado que representa uma possível solução para o PCV. O somatório das distâncias percorridas nesta instância é 10 (do ponto 1 ao 2) + 25 (do ponto 2 ao 4) + 30 (do ponto 4 ao 3) + 15 (do ponto 3 ao 1), totalizando 80, chegando na solução vetorizada $[1, 2, 4, 3, 1]$

Figura 1. Exemplo de construção pelo algoritmo de vizinho mais próximo



Fonte: www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation

3.3. Buscas locais

Heurísticas de refinamento representam uma abordagem crucial na otimização de soluções já existentes. Em contraste com as heurísticas construtivas, que iniciam com uma técnica de construção

inicial, as heurísticas de refinamento pressupõem uma solução inicial construída que será explorada em busca de melhorias.

As heurísticas de busca local podem ser exploradas de formas distintas, tais como "melhor aprimorante" ou "primeiro de melhora". Na estratégia melhor aprimorante, todas as possíveis soluções de uma determinada vizinhança são exploradas e a solução passa a ser aquela que teve o melhor valor de custo. Já na estratégia primeiro de melhora altera-se a solução corrente caso a busca encontre qualquer movimento de melhoria durante a exploração da vizinhança.

Nos algoritmos implementados neste trabalho adotou-se a abordagem de primeira melhora de forma a identificar a solução superior durante a busca e a considerá-la como a solução atual, onde o processo de busca é reiniciado considerando a solução corrente e finaliza quando toda a vizinhança é explorada e nenhuma solução melhor é encontrada.

Algoritmo 2 Descida com Primeira Melhora

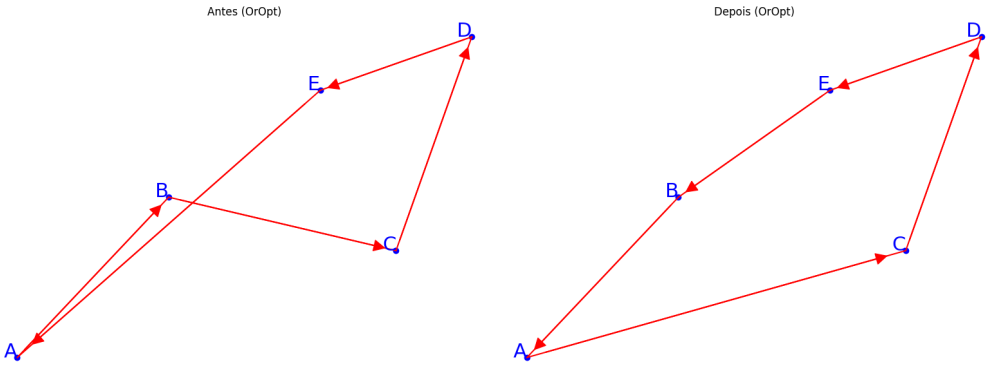
```
1: enquanto (|V| > 0) faça
2:     Selecione s' ∈ V tal que f(s') < f(s), se existir:
3:     se Existe s' ∈ V tal que f(s') < f(s) então
4:         s ← s'
5:         V = ∅                          ▷ Interrompe o loop, pois encontrou a primeira melhora
6:     fim se
7: fim enquanto
8: retorne s
```

A descida é uma estratégia de busca local que consiste em analisar todas as possíveis trocas de vizinhos nos vértices de uma solução. O objetivo é verificar se essas trocas resultam em uma melhoria na otimização, reduzindo o custo da solução. Quando uma troca benéfica é encontrada, ela é realizada e o processo recomeça até que nenhuma troca adicional possa otimizar a solução. Como detalhado no Algoritmo 2, exibindo a execução do importante refinamento para soluções já construídas.

3.3.1. Vizinhança OrOpt

A vizinhança OrOpt consiste em explorar as soluções por meio da troca da ordem de visitação de dois nós de uma determinada solução, tal como exemplificado na Figura 2. Nesse contexto, a permutação ordenada de nós representa uma estratégia eficaz para reconfigurar as rotas existentes, buscando aprimorar a solução corrente. Ao aplicar a movimentação OrOpt, os ajustes locais resultantes podem influenciar diretamente a distância percorrida e a eficiência da rota da vizinhança em questão.

Figura 2. Exemplo de movimento da vizinhança OrOpt.



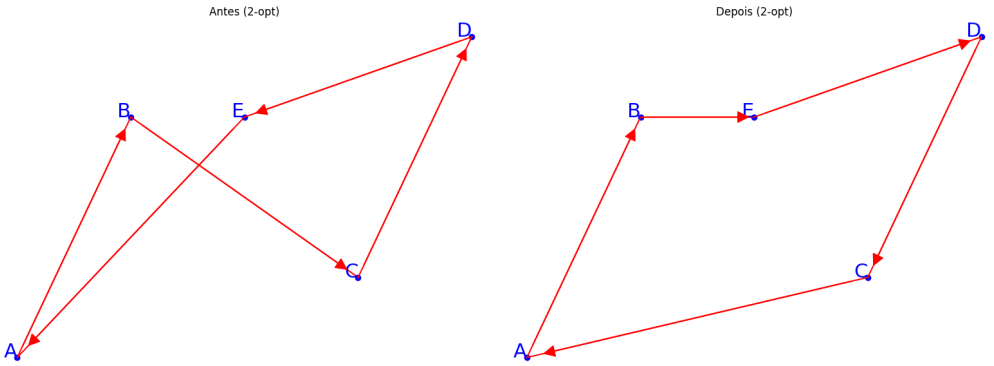
Fonte: Elaborado pelo autor.

A Figura 2 ilustra uma movimentação da vizinhança OrOpt. Na solução inicial, os pontos são visitados na ordem $[1, 2, 3, 4, 5, 6, 1]$. Após a aplicação da movimentação OrOpt, a ordem é alterada para $[1, 2, 3, 5, 4, 6, 1]$. Essa mudança reflete a troca da posição de dois nós consecutivos, que resulta em uma nova configuração de rota. A movimentação OrOpt, ao reordenar os nós, buscou encontrar uma solução mais eficiente ao ajustar a sequência de visitação, o que pode reduzir a distância total percorrida.

3.3.2. Vizinhança 2Opt

A vizinhança 2-opt consiste no movimento que envolve a remoção de duas arestas não adjacentes de uma rota existente. Ao realizar essa remoção, duas novas arestas são estrategicamente inseridas, conectando os nós de início e fim das arestas excluídas, Como destacado por [Esther and de la Mota Idalia 2011], o movimento é uma abordagem bem reconhecida para otimizar trajetos, sendo amplamente aplicado na resolução de problemas de otimização combinatória, como o PCV.

Figura 3. Exemplo de movimento do 2-opt



Fonte: Elaborado pelo autor.

A Figura 3 ilustra um exemplo de movimento 2-opt, utilizando um grafo com cinco vértices. O algoritmo 2-opt consiste em eliminar duas arestas do ciclo e inseri-las novamente de forma cruzada, caso isso melhore o ciclo. Neste exemplo, a troca envolve as conexões entre os

vértices $B \rightarrow C$ e $C \rightarrow D$. Após a troca, as conexões são alteradas para $B \rightarrow E$ e $C \rightarrow A$ consequentemente. A demonstração visual é que 2-opt reconfigura a rota para potencialmente reduzir a distância total percorrida.

Algoritmo 3 2-opt

```
1: s ← s_inicial
2: melhora ← verdadeiro
3: enquanto melhora = verdadeiro faça
4:     melhora ← falso
5:     para cada par de arestas (i, j) e (k, l) com i < j < k < l faça
6:         solucaoTemporaria ← troca(i, j, k, l, s)
7:         se custo(solucaoTemporaria) ¡ custo(s) então
8:             s ← solucaoTemporaria
9:             melhora ← verdadeiro
10:        fim se
11:    fim para
12: fim enquanto
13: retorne s
```

O Algoritmo 3 detalha o procedimento para realizar o movimento 2-opt em uma solução inicial s. O algoritmo começa com uma cópia da solução inicial s' e iterativamente tenta melhorar esta solução através de trocas 2-opt, até que nenhuma melhoria adicional possa ser encontrada.

3.4. Metaheurísticas

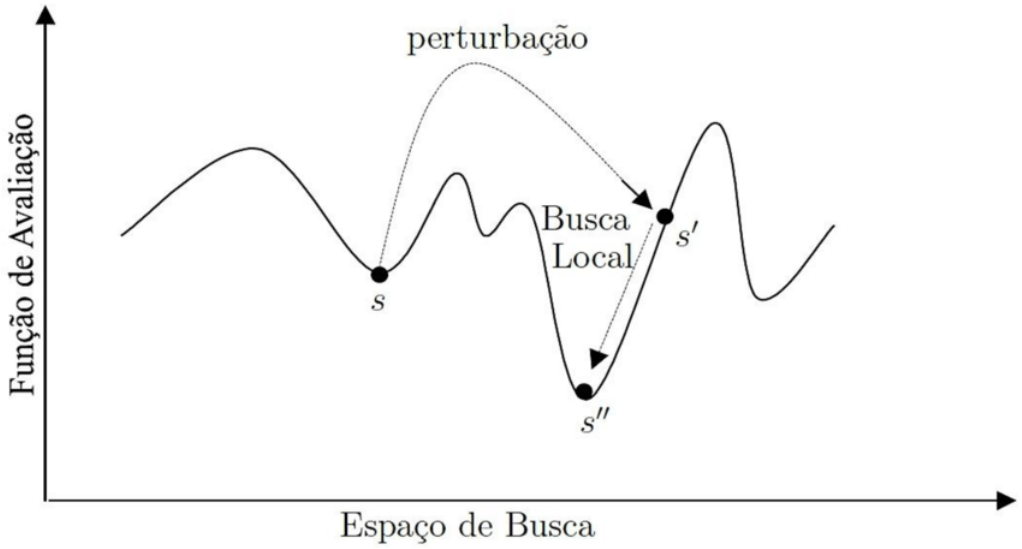
As metaheurísticas surgem em resposta à necessidade de resolver problemas de otimização combinatória com uma complexidade mais alta, para os quais encontrar a solução ótima em tempo razoável é computacionalmente inviável. Essas técnicas são especialmente úteis quando a busca exaustiva por todas as soluções possíveis é proibitivamente cara em termos computacionais. Ao combinar elementos de busca aleatória e determinística, as metaheurísticas oferecem um equilíbrio entre a qualidade da solução e o tempo de execução, tornando-as ferramentas poderosas para diversas aplicações.

3.4.1. Iteraded Local Search - ILS

A metaheurística *Iterated Local Search* (ILS), proposta por [Lourenço et al. 2003], tem como componentes uma construção inicial, uma busca local e uma pertubação, existe o critério de aceitação, usado para identificar qual o momento ideal para realizar a próxima pertubação necessário para que não fique preso em um determinado ”local”de otimização.

A pertubação não pode ser muito forte a ponto de atrapalhar a otimização realizada anteriormente, e também nem muito fraca para ser indiferente ao estado atual, ou seja, é necessário achar um equilibro entre o quanto deve per tubar ou não, levando isso em conta na execução, pode fazer come que encontre diferentes soluções ótimas, como é mostrado na figura 4.

Figura 4. Exemplo do ILS e aplicação da sua perturbação



Fonte: [Temponi 2007], p. 25.

Algoritmo 4 Iterated Local Search

1: $s0 \leftarrow GeraSolucaoInicial$
2: $s \leftarrow BuscaLocal(s0)$
3: **enquanto** (os critérios de parada não estiverem satisfeitos) **faça**
4: $s' \leftarrow$ Pertubação(histórico)
5: $s'' \leftarrow$ BuscaLocal (s')
6: $s' \leftarrow$ CriterioAceitacao(s,s'',histórico)
7: **fim enquanto**

A perturbação executada no algoritmo consiste em uma troca aleatória de vizinhos, seguindo uma abordagem similar a vizinhança OrOpt. No entanto, é essencial destacar que tanto a perturbação quanto as heurísticas empregadas no algoritmo podem ser adaptadas para atender às particularidades de diferentes problemas. A estratégia é escolhida, com o objetivo de evitar perturbações desequilibradas que possam prejudicar o processo de otimização, em vez de aprimorá-lo.

4. Experimentos Computacionais

Este estudo concentrou-se na análise de duas bases de dados amplamente reconhecidas e comumente empregadas em pesquisas sobre o PCV. A primeira, conhecida como "*Very-large-scale integration*" (VLSI), envolve a otimização de rotas em um chip, onde cada transistor representa um ponto e cada conexão entre eles, uma rota potencial, estes dados serão usados para testar a eficácia e a eficiência das diferentes linguagens de programação.

A coleção de instâncias analisadas pode ser encontrada em http://www.math.uwaterloo.ca/tsp/vlsi/, acesso em: 29 de outubro de 2024, fornecida por Andre Rohe. Essas instâncias são de vários tamanhos diferentes. As instâncias foram divididas empiricamente em três grupos: pequenas (xqf131, xqg237, pma343, pka379), médias (pbk411, pbn423, pbm436) e grandes (rbx711, dkg813, xit1083), como é mostrado na Figura 5. O número que segue os nomes das instâncias indica a quantidade de nós presentes na mesma. O objetivo dessa divisão é

PEQUENA
xqf131, xqg237, pma343, pka379

MÉDIA
pbk411, pbn423, pbm436

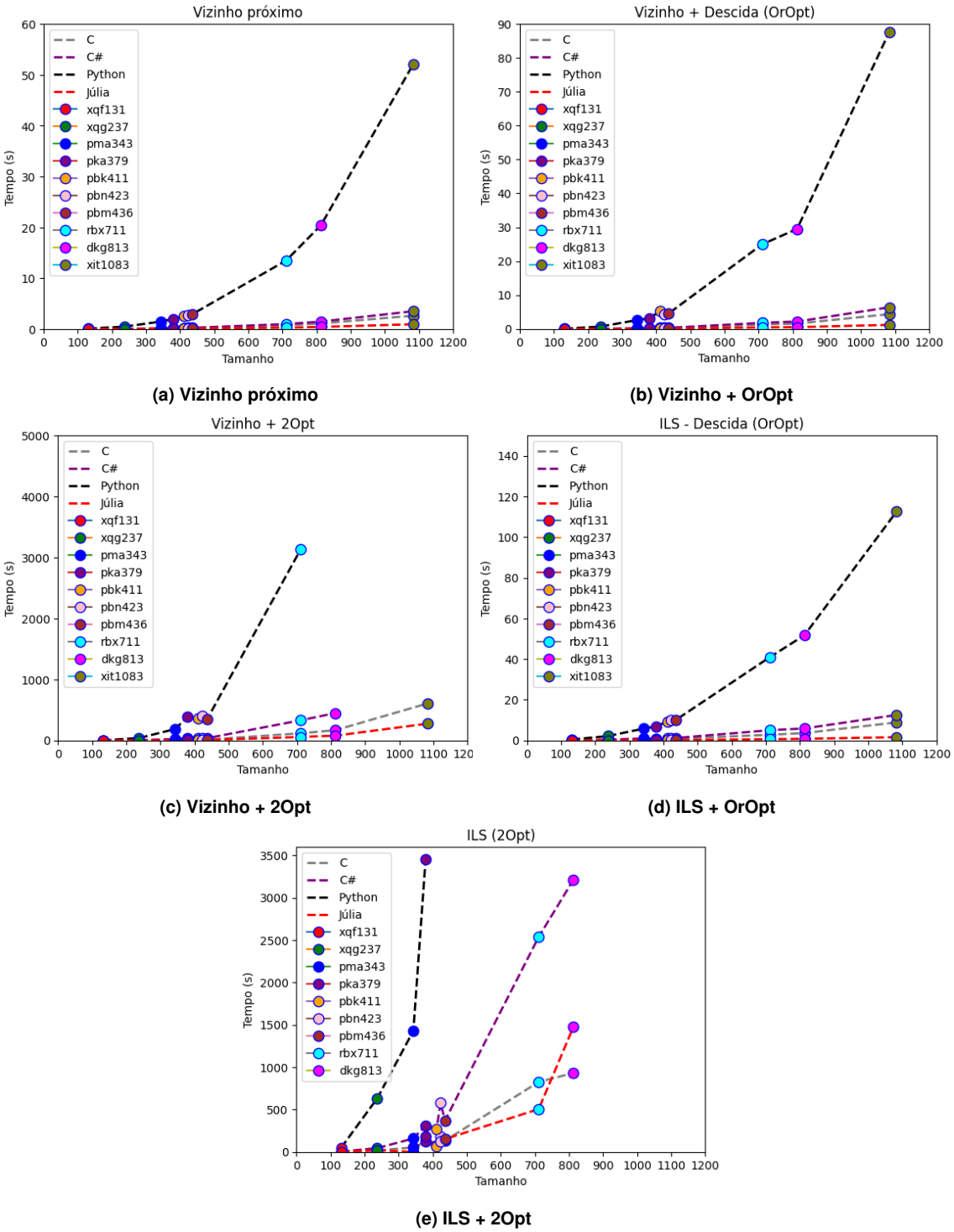
GRANDE
rbx711, dkg813, xit1083

Figura 5. Instâncias e seus grupos

analisar quais seriam os resultados e como seria o comportamento das linguagens de acordo com o aumento das instâncias, possuindo uma quantidade maior de nós a serem percorridos.

No decorrer dos testes de desempenho das linguagens de programação realizados neste estudo, todas as avaliações foram conduzidas em um ambiente computacional configurado para garantir a precisão e a reprodutibilidade dos resultados. O sistema de hardware utilizado para executar os testes consistiu em um processador AMD Ryzen 5 3600 6-Core, com uma frequência de clock de 3.59 GHz. A capacidade de memória RAM é de 16 GB. Além disso, o sistema operacional é o Windows 10 Pro de 64 bits, instalado em um HD de 1 terabyte.

Os resultados dos testes de desempenho das linguagens de programação neste estudo são apresentados na Figura 6. Cada subfigura representa as análises das técnicas de vizinhança, busca local e metaheurísticas entre elas, demonstrando o comportamento das linguagens em relação a diferentes métricas de desempenho, incluindo técnicas como o algoritmo do vizinho mais próximo (6a), busca vizinho com OrOpt (6b), busca vizinho com 2-opt (6c), e a aplicação da metaheurística ILS (*Iterated Local Search*) usando o vizinho mais próximo como construção e refinamento usando a técnica da descida considerando a vizinhança OrOpt (6d) e a 2-opt (6e). Cada gráfico ilustra o comportamento das linguagens de programação em relação ao tempo de execução, apresentando o tempo em segundos que cada linguagem demandou em relação ao tamanho da instância em todas as linguagens, vale a o observação de que técnicas que ultrapassaram um limite estabelecido nos testes de até uma hora, não é exibido no gráfico, esse limite foi colocado com o intuito dos testes não ficarem muito massivos e perderem o controle já que o segundos exibidos são uma média de dez repetições daquela técnica.



(a) Vizinho próximo

(b) Vizinho + OrOpt

(c) Vizinho + 2Opt

(d) ILS + OrOpt

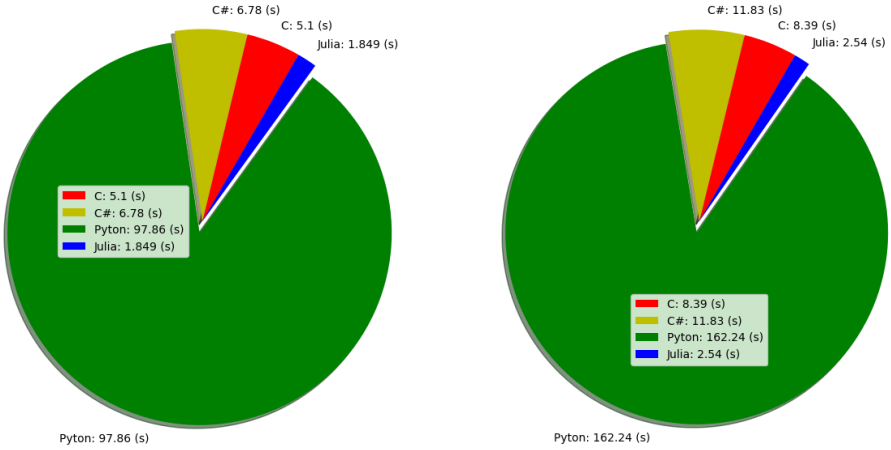
(e) ILS + 2Opt

Figura 6. Análises de todas as técnicas

Observa-se na figura que 6, conforme a escala das instâncias aumenta, o tempo de execução também tende a aumentar para todas as técnicas avaliadas. Nos algoritmos do vizinho mais próximo (6a), vizinho com descida (6b) e ILS com descida (6d), o aumento no tempo é menos acentuado e a variação dos resultados é suave, indicando que esta técnica possui uma complexidade computacional relativamente baixa.

Por outro lado, as técnicas que envolvem o 2-opt (6c) e (6e) mostram um crescimento mais significativo no tempo de execução, refletindo a maior complexidade dessas abordagens, o que pode ser atribuído ao esforço computacional adicional necessário para explorar múltiplas vizinhanças e realizar várias iterações de busca, no caso da meta-heurística ILS que demonstrou tempos de execução mais demorados pode ter relação com o fato da pertubação.

A análise comparativa dos resultados de desempenho das linguagens de programação C, Python, C# e Julia revelou diferenças significativas em termos de eficiência e velocidade em uma variedade de cenários. Os testes aplicados, utilizando técnicas no PCV em diferentes instâncias, mostraram consistentemente que tanto C quanto Julia apresentam desempenho de estarem sempre pelo menos duas vezes mais rápidos que os demais.



(a) Vizinho mais próximo **(b) Vizinho + OrOpt**

Figura 7. Comparação de diferentes métodos

Os dados exibidos na Figura 7 proporcionaram uma visão do desempenho das linguagens de programação em cenários com técnicas distintas de forma acumulada, ou seja, é exibido o somatório da porcentagem de tempo gasto em segundos das instâncias de acordo com cada linguagem. Julia, surpreendentemente, revelou-se como uma alternativa com um bom desempenho, especialmente considerando que é uma linguagem mais nova, lançada em 2012, em relação as outras, essa particularidade geralmente leva à expectativa de que uma linguagem mais jovem não tenha o mesmo nível de maturidade ou otimização que as demais, obtendo resultados melhores que o da linguagem C. Esse fenômeno pode ser associado à eficácia da linguagem em manipular operações complexas, destacando o trabalho de [Bezanson et al. 2017] sobre a eficiência e flexibilidade da Julia em tarefas computacionais intensivas. Em contrapartida, embora C# tenha apresentado um desempenho mediano, quando comparado a Júlia e C, foi notado que Python, demonstrou uma desvantagem em termos de eficiência temporal. Este resultado está alinhado com estudos anteriores, como o trabalho [Perkel 2017], que discute entre legibilidade e desempenho em Python.

Os tempos de execução obtidos na Figura 8 revelam diferenças consideráveis entre as linguagens analisadas e Python. A linguagem C apresentou o tempo de execução mais rápido, com apenas 0,0047 segundos. Sua natureza de baixo nível e eficiência de código contribuem para sua vantagem competitiva em termos de desempenho. A linguagem C#, embora um pouco mais lenta, ainda demonstrou uma desempenho competitivo, com um tempo de execução de 0,0061 segundos. Por outro lado, a linguagem Python mostrou-se mais lenta em comparação às demais, exigindo 0,0799 segundos para executar a mesma tarefa. Já a linguagem Julia, obteve um tempo

de execução de 0,0060 segundos.

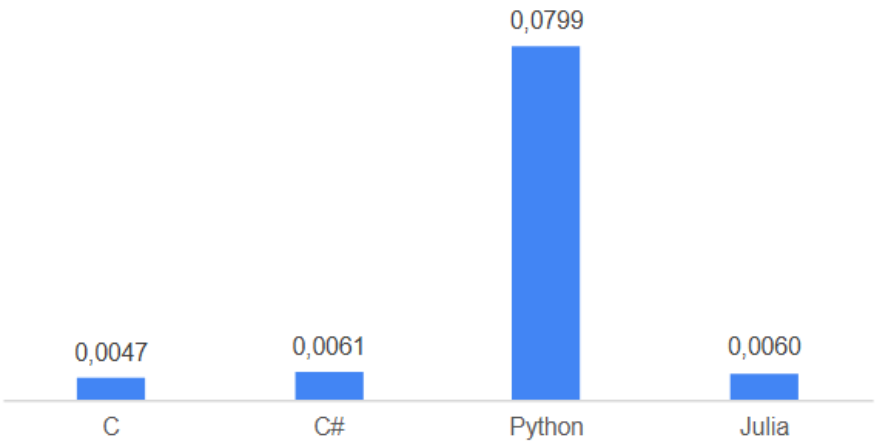


Figura 8. xqf131 - Vizinho mais próximo

A próxima análise busca mostrar o desempenho à medida que é aumentado a complexidade da instância para pelo menos o dobro da instância de 131 elementos, especificamente a instância 'pma343' com 343 elementos. Isso aumenta significativamente a demanda computacional da linguagem, exigindo um esforço computacional mais substancial.

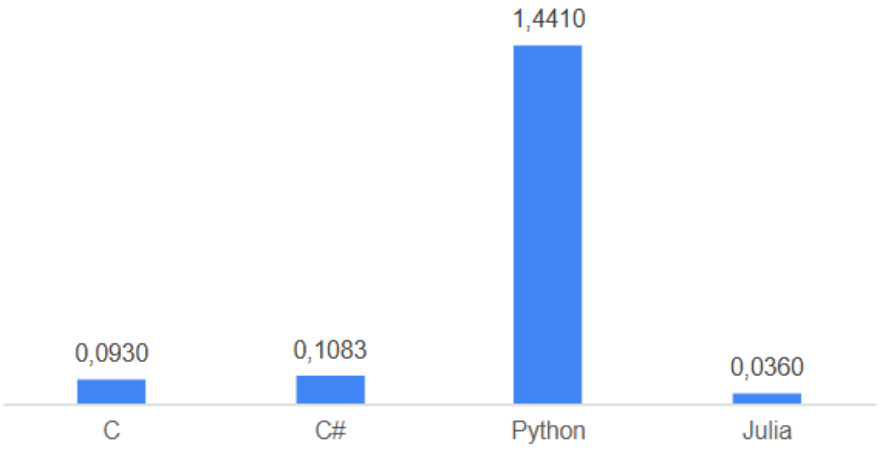


Figura 9. pma343 - Vizinho mais próximo

As técnicas apresentadas nas Figuras 8 e 9 são determinísticas, ou seja, são técnicas que não apresentam variações nos resultados e não envolvem aleatoriedade durante o processo de ajuste para diminuir o custo e encontrar uma solução para as instâncias. Esse caráter determinístico garante uma consistência nos resultados obtidos em cada execução das técnicas, proporcionando uma base sólida para a análise comparativa do desempenho das linguagens de programação.

Os resultados da Figura 9 revelam um aumento no tempo de execução em Python, com um valor de 1,4410 segundos. Esse aumento de mais de dez vezes maior em relação à análise anterior indica uma possível limitação no desempenho da linguagem Python para instâncias maiores. Por outro lado, a linguagem Julia continua demonstrando uma performance eficiente, exigindo apenas 0,0360 segundos para executar o processo. A linguagem C também se mantém eficiente, com

um tempo de execução de 0,0930 segundos já na execução do C# teve uma execução de 0,1083 segundos.

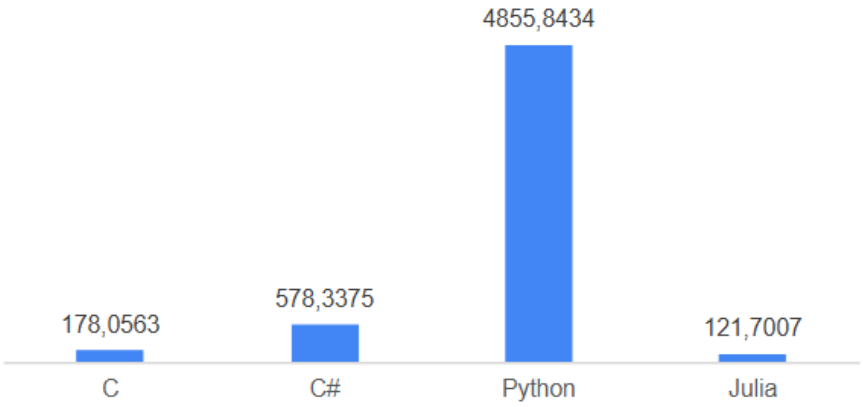


Figura 10. pbn423 - ILS (2opt)

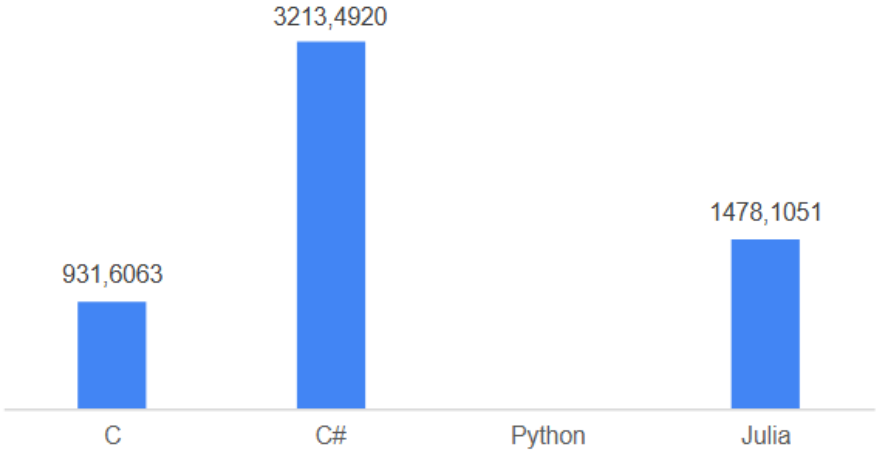


Figura 11. dkg813 - ILS (2opt)

Quando é considerado o *Iterated Local Search* (ILS) apresentados nas Figuras 10 e 11, a dinâmica se transforma significativamente. O ILS envolve a perturbação aleatória dos nós seguida por uma busca local, introduzindo assim um elemento de aleatoriedade no processo. Isso torna o ILS mais complexo em comparação com as técnicas determinísticas, pois os resultados podem variar entre diferentes execuções devido à natureza aleatória das perturbações. Nos experimentos conduzidos com o ILS, critérios de aceitação são adotados baseados em níveis de melhoria, é considerado o resultado como aceitável apenas quando certos limites são alcançados, proporcionando critérios adicionais para avaliar a qualidade das soluções obtidas, para tais experimentos adotou-se 60 minutos como critério de parada da metaheurística, portanto os valores vazios significam que a técnica excedeu o limite especificado.

Para os experimentos, adotou-se 90 minutos como critério de parada da metaheurística; assim, os valores vazios indicam que a técnica excedeu o limite especificado. Uma análise detalhada dos resultados, que inclui tempos de execução e métricas de desempenho para cada técnica aplicada, pode ser consultada na tabela completa apresentada na Figura 12, no final do artigo.

5. Conclusão

O presente trabalho explorou o desempenho de diferentes linguagens de programação C, Python, C#, e Julia, no contexto da otimização combinatória, com foco específico no Problema do Caixeiro Viajante (PCV). O objetivo principal foi analisar como a escolha da linguagem de programação pode influenciar a eficiência e a eficácia, através do tempo gasto de acordo com a instância em questão, juntamente da técnica aplicada para a resolução de problemas de otimização.

A comparação dos resultados e desempenho das linguagens de p rogramação C, Python, C# e Julia oferecem ideias e questionamentos valiosos sobre suas eficiências em uma de cenários de otimização. Durante os testes, tanto C quanto Julia demonstraram consistentemente um desempenho superior em relação a C# e Python, destacando suas vantagens em termos de eficiência computacional.

”É importante reconhecer que algumas linguagens, como C, enfatizam o controle direto sobre os recursos do sistema, o que as torna ideais para aplicações que exigem alta performance e eficiência. Por outro lado, linguagens como Python são valorizadas por sua sintaxe expressiva e pela vasta disponibilidade de bibliotecas, que facilitam o desenvolvimento rápido e a prototipagem de soluções, permitindo que os desenvolvedores se concentrem na lógica do problema em vez de nos detalhes de implementação.

O reconhecimento de que algumas linguagens, como C, enfatizam o controle direto sobre os recursos do sistema, enquanto outras, como Python, são valorizadas por sua sintaxe expressiva e disponibilidade de biblioteca, fica aberto o questionamento o quanto o nível do desenvolvimento pode influenciar no resultado, deixando claro que as implementações deste trabalho foi focado em sintaxes básicas. Essa diversidade de linguagens permite aos desenvolvedores escolherem a ferramenta mais adequada para cada projeto, levando em consideração os requisitos específicos de desempenho e funcionalidade.

Este estudo destaca a importância de selecionar a linguagem de programação adequada com base nos requisitos específicos de desempenho e funcionalidade do projeto. Embora a popularidade e a conveniência sejam fatores a considerar, a eficiência computacional e a adequação ao problema específico devem ser prioritárias.

Ao considerar o cenário amplo das linguagens de programação, é claro que cada uma delas possui seus pontos fortes e aplicações ideais. No entanto, é importante reconhecer que o desempenho relativo de cada linguagem pode variar significativamente dependendo do contexto de uso e das características específicas do problema a ser resolvido.

Portanto, ao fazer escolhas sobre quais linguagens adotar em um projeto de otimização combinatória, os desenvolvedores devem avaliar cuidadosamente as necessidades de desempenho e eficiência, garantindo assim a melhor solução para cada situação. As linguagens C e Julia, são particularmente eficazes para aplicações que exigem alta performance computacional, como a resolução do PCV.

	PEQUENA				MÉDIA				GRANDE			
TÉCNICAS	C	C#	Python	Julia	C	C#	Python	Julia	C	C#	Python	Julia
VIZINHO	0,005	0,006	0,080	0,006	0,144	0,184	2,508	0,052	0,712	0,956	13,457	0,258
	0,030	0,036	0,449	0,014	0,157	0,207	2,668	0,061	1,082	1,509	20,408	0,392
	0,093	0,108	1,441	0,036	0,172	0,222	2,900	0,071	2,599	3,476	52,000	0,919
	0,115	0,146	1,956	0,048								
VIZINHO+OROPT	0,008	0,010	0,080	0,006	0,267	0,351	4,925	0,084	1,329	1,808	24,989	0,463
	0,046	0,055	0,697	0,020	0,247	0,316	4,198	0,082	1,574	2,178	29,506	0,532
	0,149	0,188	2,588	0,042	0,245	0,319	4,568	0,072	4,356	6,380	87,597	1,191
	0,179	0,233	3,107	0,058								
VIZINHO+2OPT	0,087	0,231	1,855	0,043	13,761	39,108	346,966	6,964	116,824	330,486	3134,158	57,109
	1,816	4,790	38,825	0,803	14,896	42,317	4,198	7,392	168,328	448,523	4164,279	78,541
	7,535	21,370	186,430	3,953	13,001	36,237	325,593	6,673	607,245	-	-	275,039
	15,298	43,322	386,198	7,641								
VIZINHO+OROPT+2OPT	0,028	0,065	0,615	0,015	9,052	25,493	235,813	4,495	72,078	204,831	1979,342	36,436
	1,495	3,937	33,134	0,708	11,367	32,133	296,550	5,649	115,048	306,729	2866,299	51,420
	6,373	17,762	157,083	3,192	9,701	27,795	257,268	5,109	389,213	-	-	178,819
	12,440	34,898	322,645	6,400								
VIZINHO+2OPT+OROPT	0,092	0,230	1,788	0,059	13,670	38,887	341,789	7,138	115,758	341,388	3.175,720	58,434
	1,723	4,828	38,618	0,879	14,912	42,284	385,816	7,603	166,149	447,723	4.405,848	76,777
	7,464	21,385	181,940	3,824	12,758	36,087	331,319	6,530	621,676	-	-	272,935
	15,192	43,283	370,433	7,730								
ILS COM OROPT	0,065	0,122	0,469	0,013	0,781	0,979	9,304	0,141	2,774	5,015	40,878	0,598
	0,226	0,402	2,135	0,032	0,759	1,167	9,886	0,138	3,627	5,887	51,720	0,806
	0,544	0,947	5,842	0,085	0,830	1,191	10,066	0,156	8,874	12,531	112,549	1,609
	0,596	0,771	6,934	0,101								
ILS COM 20PT	1,842	4,823	48,526	0,013	76,026	270,275	-	70,282	825,978	2538,545	-	505,976
	18,792	44,704	632,345	21,112	178,056	578,338	-	121,701	931,606	3213,492	-	1478,105
	52,491	158,691	1428,841	57,973	133,291	371,298	-	151,794	4844,692	-	-	3681,996
	120,605	312,776	3456,103	181,408			-			-	-	-

Figura 12. Tabela completa de resultados (figura rotacionada)

Referências

Aruoba, S. B. and Fernández-Villaverde, J. (2017). A comparison of programming languages in economics. *Journal of Economic Dynamics and Control*, pages 1–19.

Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98.

Biggs, N., Lloyd, E. K., and Wilson, R. J. (1986). *Graph Theory, 1736-1936*. Oxford University Press.

Breuer, J. and Ziegler, E. (2018). Development of a Computational Model for Blood Flow Simulation in Cardiac Valves. *Journal of Computational Biology*, 25(6):558–567.

Edelman, A., Bezanson, J., Karpinski, S., and Shah, V. B. (2023). Julia: A high-performance programming language. *IEEE Computer*, 56(8):20–27.

Esther, S. P. and de la Mota Idalia, F. (2011). Experimental analysis of selective uncrossing as a modification to the 2-opt technique for solving the euclidean traveling salesman problem. *BALCOR 2011*, page 222.

Harris, C. R., Millman, K. J., Van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., and Oliphant, T. E. (2020). Array programming with numpy. *Nature*, 585(7825):357–362.

Hsu, D. K. (2012). Learning a new programming language? why not c#. net programming? In *2012 IEEE TCF Information Technology Professional Conference*, pages 1–5. IEEE.

Koolen, T. and Deits, R. (2019). Julia for robotics: simulation and real-time control in a high-level programming language. In *2019 International Conference on Robotics and Automation (ICRA)*.

Lourenço, H. R., Martin, O. C., and Stützle, T. (2003). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 320–353. Springer.

Mendes, N. F. M. (2015). Técnicas de otimização combinatória aplicadas a criação de estratégias de policiamento urbano. Master's thesis, Universidade Federal de Viçosa, Viçosa, MG.

Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.

Palomares, M. (2016). Alternativas de instalação de centros de distribuição de uma empresa multinacional de grande porte utilizando modelagem matemática. Monografia (Bacharel em Engenharia de Transportes e Logística), Universidade Federal de Santa Catarina, Joinville, SC.

Peres, F. and Castelli, M. (2021). Combinatorial optimization problems and metaheuristics: Review, challenges, design, and development. *Applied Sciences*, 11(14):6449.

Perkel, J. M. (2017). Why python is the next wave in earth sciences computing. *Nature*, 549(7674):451–453.

Temponi, J. (2007). Comportamento da meta-heurística ils. In *Anais do XXXVI Encontro Nacional de Engenharia de Produção*, pages 25–30. ABEPRO.