Avaliação de Desempenho e Escalabilidade do Algoritmo de Otimização de Colônia de Formigas em C++ e Python

João Marcos de Oliveira Magalhães, Ana Carolina Medeiros Gonçalves, Cristiane Neri Nobre, Henrique Cota de Freitas

Grupo de Arquitetura de Computadores e Processamento Paralelo (CArT)
Laboratório de Inteligência Computacional Aplicada (LICAP)
Instituto de Ciências Exatas e Informática (ICEI)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte, MG – Brasil

{jmomagalhaes,ana.medeiros}@sga.pucminas.br, {nobre,cota}@pucminas.br

Resumo. *O algoritmo de Otimização de Colônia de Formigas, do inglês Ant Colony Optimization (ACO), é um algoritmo inspirado no comportamento das formigas em busca de alimento. Implementado originalmente em Python para seleção de instâncias, o código desenvolvido apresenta desempenho lento em grandes bases de dados. Este artigo apresenta a avaliação da conversão do ACO desenvolvido em Python para C++, visando comparar o desempenho das duas versões. A hipótese está fundamentada na melhoria do tempo de execução em C++, mantendo a qualidade na seleção de instâncias. Os objetivos incluem converter o código ACO em Python para C++, comparar o desempenho entre as duas versões, e identificar razões para as diferenças. Os resultados mostram que o ACO em C++ possui maior desempenho e escalabilidade com qualidade na redução de instâncias, principalmente, para grandes bases de dados.*

1. Introdução

Na mineração de dados e aprendizado de máquina, é importante escolher um conjunto de dados representativo de exemplos de treinamento para construir modelos mais precisos e eficientes em termos de tempo de processamento e uso de recursos computacionais (Borovicka et al., 2012). Algoritmos de seleção de instâncias como o *K-Means* (Saha et al., 2022) e o *ReliefF* (Spolaôr et al., 2013) são utilizados para remover redundância e ruído dos conjuntos de dados.

O algoritmo de Otimização de Colônia de Formigas, do inglês *Ant Colony Optimization* (ACO) (Dorigo et al., 2006), utiliza o comportamento das formigas na busca

Este trabalho recebeu suporte da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - PROAP 88887.842889/2023-00–PUC/MG, PDPG 88887.708960/2022-00–PUC/MG, Código de Financiamento 001; do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) - 311573/2022-3, 311697/2022-4; da Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) - PCE-00437-24, APQ-03076-18, APQ-05058-23; e da Pontifícia Universidade Católica de Minas Gerais (PUC Minas) - FIP 2024/30947, PIBIC/PIBIT-2023/29487.

por alimento para selecionar um subconjunto ótimo de instâncias de treinamento. As formigas se separam, constroem caminhos e deixam rastros de feromônio, aumentando a probabilidade de outras seguirem o mesmo caminho, simulando a busca pelo melhor conjunto de instâncias.

Em um trabalho recente do grupo, o ACO foi implementado em Python (Medeiros, 2022) como um algoritmo que seleciona instâncias relevantes de maneira iterativa, utilizando heurísticas probabilísticas para formar um conjunto otimizado de dados. As formigas seguem caminhos pseudoaleatórios para encontrar um destino, e os caminhos mais eficazes formam a solução final para reduzir a base de dados. No entanto, devido à sua natureza probabilística, não há garantia de que a solução ótima será alcançada em cada execução, resultando em variações nos resultados. Além disso, o custo computacional do algoritmo aumenta com o número de instâncias na base de dados, o que pode ser significativo, degradando o desempenho e a escalabilidade de processamento.

Nesse sentido, o alto custo computacional do ACO (Medeiros, 2022) tem limitado sua escalabilidade, tornando-o inviável para grandes bases de dados devido ao tempo excessivo necessário para processar grandes conjuntos. Isso resulta em perda significativa de tempo e recursos, o que é crítico para aplicações de aprendizado de máquina e mineração de dados. Portanto, o objetivo desse artigo é propor e avaliar uma implementação em C++[1] para aumentar o desempenho e escalabilidade de processamento, sem comprometer a qualidade dos resultados. Foram avaliados e comparados os resultados das duas abordagens em diferentes bases de dados, focando em métricas de acurácia, precisão, recall, F-measure, tempo de execução, falhas de página (*page faults*), ciclos e *cache misses*.

A principal contribuição deste trabalho está na proposta eficaz e escalável do algoritmo ACO em C++ em alternativa à versão em Python, que é muito utilizada por comunidades científicas com aplicações em diversas áreas. Dessa forma, espera-se que a proposta ACO em C++ possa ser amplamente utilizada visando melhorar o desempenho ao remover ruídos e redundâncias de grandes bases de dados em várias áreas do conhecimento. As etapas realizadas para validar os resultados e contribuição científica são as seguintes:

- Converter a implementação do algoritmo ACO de Python para C++;
- Realizar reduções em diferentes *datasets* com o algoritmo ACO em Python e C++;
- Coletar métricas de aprendizado de máquina para validar a qualidade da seleção de instâncias;
- Avaliar o desempenho e a escalabilidade da seleção de instâncias entre as duas versões.

Este trabalho está organizado da seguinte forma: A Seção 2 apresenta uma revisão da literatura sobre seleção de instâncias, e *Ant Colony Optimization*; a Seção 3 aborda trabalhos correlatos; a Seção 4 descreve a metodologia utilizada para a conversão de implementação; a Seção 5 apresenta detalhes técnicos da implementação do ACO em C++; a Seção 6 discute os resultados obtidos e suas implicações; e por fim, a Seção 7 apresenta as conclusões.

[1]https://github.com/jmarcosjm/aco-cpp

2. Seleção de Instâncias e Colônia de Formigas

O estudo desenvolvido por Olvera-López et al. (2010) aborda o conceito da seleção de instâncias e seus diferentes métodos de aplicação. A seleção de instâncias é necessária para remover informações inúteis da base de dados, como ruídos ou instâncias redundantes. Para isso, existem diferentes métodos de seleção, que são divididos em dois grupos:

- *Wrapper*. O critério de seleção é baseado na precisão obtida por um classificador. Nesse grupo, as instâncias que não contribuem para a precisão da classificação são descartadas.
- *Filter*. O critério é baseado em alguma função de seleção para avaliar quais instâncias serão selecionadas, não possuindo relação com classificadores.

O algoritmo de otimização de colônia de formigas (ACO)[2] (Dorigo et al., 2006) utiliza uma abordagem relativamente nova para a resolução de problemas que se baseiam nos comportamentos sociais de insetos. O ACO se inspira no comportamento de forrageamento de algumas espécies de formigas, em que elas depositam feromônios no chão para marcar um caminho favorável, aumentando as chances dele ser seguido por outros membros da colônia.

Neste estudo, o ACO usa uma estratégia híbrida com elementos dos métodos *filter* e *wrapper*. O ACO adota uma abordagem *wrapper* ao escolher a melhor solução das formigas usando um classificador *K-Nearest Neighbors* (KNN). Simultaneamente, a seleção de instâncias é feita por uma função heurística probabilística independente de algoritmos de aprendizado de máquina, semelhante aos métodos *filter*. Assim, o ACO é um exemplo de método híbrido.

Os principais algoritmos (Dorigo et al., 2006) que implementam o conceito do ACO são os seguintes:

- *Ant System* (AS): O primeiro algoritmo proposto na literatura, sua principal característica é que em cada iteração o valor do feromônio é atualizado por todas as formigas que construíram uma solução na iteração.
- *MAX - MIN Ant System* (MMAS): É uma melhoria do AS que possui como característica principal o fato de que somente a melhor formiga atualiza a trilha de feromônio e o valor do feromônio é limitado a um valor máximo e mínimo pré-estabelecidos.
- *Ant Colony System* (ACS): A principal característica do ACS é a inclusão da atualização de feromônio local, além da atualização de feromônio que ocorre ao final de cada iteração, como no AS. A atualização de feromônio local é feita por todas as formigas ao final de cada passo do processo de construção de solução.

[2]Bio-inspirado: classe de algoritmos de otimização que se baseia em processos observados na natureza.

3. Trabalhos Correlatos

De forma semelhante ao abordado neste trabalho, Nanz e Furia (2015) compararam diferentes linguagens executando os mesmos algoritmos. Foram selecionadas 8 linguagens de paradigmas distintos: Procedural (C e Go), Orientação a Objetos (C# e Java), Funcional (F# e Haskell), e *Scripting* (Python e Ruby). Essas linguagens são comparadas em termos de código conciso, tamanho do executável, tempo de execução, uso de memória e propensão a falhas, alinhando-se ao escopo deste estudo na comparação de desempenho.

Em outro trabalho, Ueda e Ohara (2017) avaliaram o desempenho de linguagens compiladas dinamicamente e estaticamente. Isso se relaciona com este estudo, já que Python é interpretada dinamicamente e C++ é compilada estaticamente. O estudo testou o desempenho de uma aplicação web construída em Go (compilação estática) e em Java e JavaScript (compilação dinâmica), revelando que linguagens estaticamente compiladas, como Go, superam em desempenho as dinamicamente compiladas, como Java e JavaScript.

O artigo escrito por Gong et al. (2021) também apresenta o algoritmo de classificação utilizado como classificador para o ACO, o *K-nearest neighbor* (K-NN), e destaca suas limitações, como a necessidade de armazenar todos os exemplos de treinamento e baixa tolerância a objetos ruidosos. Para superar essas limitações, os autores propõem um algoritmo de seleção de instâncias chamado de *Evidential Instance Selection* (EIS) baseado na teoria da evidência, que retém as instâncias de fronteira e reduz o ruído. Também é apresentada uma versão distribuída do EIS (EIS-AS) para lidar com *big data* usando o paradigma *MapReduce* e o *Apache Spark*. Tanto o EIS quanto o EIS-AS foram testados em conjuntos de dados pequenos e grandes, mostrando bom desempenho na redução do tamanho dos dados de treinamento.

A seleção de instâncias visa obter um subconjunto de dados que mantém ou melhora o desempenho do original, mas com um tamanho menor, eliminando instâncias ruidosas e redundantes. Isso reduz o tempo e o custo computacional para trabalhar com os dados, sendo o foco deste estudo e do artigo de Bertoni e Pires (2017). O artigo busca equilibrar a redução dos dados e a precisão na classificação, avaliando os algoritmos *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) e *Strength Pareto Evolutionary Algorithm II* (SPEA-II). Conclui-se que a seleção de instâncias pode melhorar a precisão e reduzir o custo computacional dos algoritmos de classificação, com o NSGA-II apresentando melhores resultados.

4. Metodologia

Alguns materiais foram utilizados no desenvolvimento e avaliação do algoritmo ACO. Entre eles estão, e.g., linguagens de programação, bibliotecas, ferramentas de monitoramento e análise de desempenho. Foram escolhidas Python e C++ como linguagens de programação. Python, amplamente usada em aprendizado de máquina e mineração de dados, é valorizada pela facilidade de uso e pela disponibilidade de bibliotecas especializadas, mas não é usualmente indicada para alto desempenho. Por isso, a linguagem foi escolhida C++, conhecida por ser uma das principais linguagens em alto desempenho,

para a nova versão do ACO. C++ é uma linguagem de baixo nível que oferece mais controle sobre os recursos do sistema, manipulação direta da memória, menos abstrações e suporte a diversas bibliotecas, superando C nos critérios de escolha (Stroustrup, 1986). Além disso, a biblioteca NumCpp em C++ facilita a replicação de certas funções utilizadas em Python.

A partir dos conceitos de ambas as linguagens, existem algumas explicações para o Python ser menos eficiente que o C++. Primeiramente, C++ é compilado, com o código-fonte convertido para código de máquina e executado diretamente na CPU. Em contraste, o código-fonte em Python é compilado em *bytecode*, que é interpretado e executado na Máquina Virtual Python, linha a linha e em tempo de execução, gerando um *overhead* considerável de performance (Barany, 2014). Além disso, Python pode carecer de otimizações que linguagens compiladas previamente, como C++, recebem do compilador (Bacon et al., 1994).

Outro ponto é a tipagem estática de C++ contra a tipagem dinâmica de Python. Como C++ é tipado estaticamente, o compilador sabe previamente quais tipos de dados são esperados em cada parte do programa, permitindo que ele realize otimizações tornando o código menor ou mais rápido (Leroy, 1998), o que não ocorre em Python devido a sua natureza dinamicamente tipada.

Além disso, o gerenciamento de memória em Python é mais complexo do que em C++. Em C++, a gestão de memória é responsabilidade do desenvolvedor, enquanto em Python, um *garbage collector* automaticamente e regularmente libera objetos não utilizados da memória. Embora isso seja conveniente, tem um custo: o *garbage collection* pode consumir mais tempo do que o esperado, potencialmente tornando a execução do algoritmo mais lenta (Ismail e Suh, 2018).

A arquitetura de máquinas utilizada neste artigo consiste em um processador AMD Ryzen 7 3700X, com 32GB de memória RAM e o sistema operacional Ubuntu. As bibliotecas escolhidas incluem *NumCpp*[3] e algumas bibliotecas padrão, como a random. A biblioteca *NumCpp* é uma biblioteca de manipulação de arrays multidimensionais para C++ que reproduz as funcionalidades da biblioteca *NumPy*[4] do Python, utilizada no código ACO original. A biblioteca random de C++ é usada para gerar números pseudoaleatórios com qualidade estatística superior à função *rand* da *stdlib* de C, o que é crucial para a eficiência na seleção de instâncias. Além disso, foi utilizada a ferramenta Perf[5] para analisar os contadores de hardware durante a execução dos algoritmos em ambas as linguagens.

4.1. Método de Desenvolvimento e Avaliação

O método utilizado nesta pesquisa consiste de cinco etapas para implementar e avaliar o algoritmo ACO. A primeira etapa foi converter o algoritmo ACO de Python para C++, mantendo a lógica o mais próxima possível da original, para garantir a equivalência entre as versões. As estruturas de dados e de repetição foram preservadas: em Python, utilizaram-se Lists, Tuples e Arrays (NumPy), enquanto em C++ foram usados vectors, pairs e ndArrays (NumCpp). As estruturas de repetição incluíram laços de for e while em

[3]https://dpilger26.github.io/NumCpp/doxygen/html/index.html

[4]https://numpy.org/

[5]https://github.com/springmeyer/profiling-guide, https://docs.python.org/3/howto/perf_profiling.html

ambas as linguagens.

A segunda etapa do método envolveu realizar múltiplas reduções com *datasets* de diferentes tamanhos em Python e C++ para coletar métricas de eficiência na seleção de instâncias. Foram selecionados 9 *datasets*, divididos em 3 categorias baseadas em tamanho e tempo de redução: Pequenos ($<=$ 700 linhas), Médios ($>$ 700 e $<=$ 1500 linhas), e Grandes ($>$ 1500 linhas). Foram realizados testes com 3 *datasets* pequenos, 3 médios e 3 grandes. Nessa etapa, também foram catalogados os tempos de execução para reduzir o mesmo *dataset*. Foram executadas 6 reduções com os algoritmos em cada base de dados para obter valores médios nas métricas. O critério de parada foi o desvio padrão do tempo de execução, máximo de 0,61% da média de 6 execuções, garantindo a confiabilidade e baixa variabilidade das medições.

A terceira etapa consistiu em realizar uma redução em cada *dataset* utilizando a ferramenta de monitoramento Perf em ambos os algoritmos, avaliando o desempenho em tempo real por meio de contadores de hardware. Com isso é possível coletar métricas importantes na análise de desempenho, como *page faults* na memória principal, *load misses*, entre outros dados que ajudam a evidenciar a diferença entre as duas implementações.

A quarta etapa é comparar as métricas coletadas na etapa 2 e verificar se a eficiência na seleção de instâncias se manteve na conversão de Python para C++. A comparação por meio de gráficos permite avaliar de forma clara se houve ganho ou perda de eficácia na tentativa de conversão. Para isso um algoritmo em Python é implementado e utilizado para receber as métricas coletadas e gerar gráficos relevantes.

A quinta etapa consiste em correlacionar o aumento do tempo de execução com o tamanho dos *datasets* para os dois algoritmos e avaliar se houve ganhos de performance na versão em C++. Também são analisadas as métricas obtidas pelo Perf na etapa 3, afim de explicar a diferença de desempenho. Além disso, deve ser considerado o que os ganhos de desempenho implicam na prática para um processo de aprendizado de máquina e mineração de dados.

5. Desenvolvimento da versão em C++

Durante o desenvolvimento do algoritmo ACO para a linguagem C++, foi encontrado um desafio significativo ao tentar adaptar a função '*get_best_solution*' do Python para o C++. Essa função tem como objetivo identificar a melhor solução dentre as alternativas propostas pelas formigas. Essa tarefa é realizada por meio de treinamento com o classificador *KNeighborsClassifier* da biblioteca *scikit-learn*.

A função '*fit(X, Y)*' do *KNeighborsClassifier* em Python recebe o conjunto de treinamento 'X' e um vetor 'Y' representando os valores alvo. No contexto do algoritmo ACO, o vetor 'Y' consiste nos valores da coluna do atributo de classe do conjunto de dados, ou seja, um vetor contendo as classes da base de dados e 'X' o *dataset* sem a coluna de atributo classe.

Devido à dificuldade em encontrar uma biblioteca para C++ com implementação equivalente ao *KNeighborsClassifier* de Python, que aceitasse tanto o conjunto de treinamento 'X' quanto o vetor 'Y', foi efetuada uma modificação para registrar as soluções

identificadas por cada formiga em um arquivo chamado solutions.csv. Em paralelo, um código simplificado em Python que lê as soluções no arquivo *solutions.csv*, executa a função '*get_best_solution*' e retorna o *dataset* reduzido foi desenvolvido. Dito isso, é importante enfatizar que a medição de tempos em Python desconsiderou a execução dessa função.

Posteriormente foi implementada uma versão simples de um KNN em C++ para replicar a função '*get_best_solution*' e apesar do algoritmo funcionar efetivamente, o tempo de execução teve um aumento considerável. Por isso foi decidido que essa função deve ser implementada futuramente mediante sua otimização.

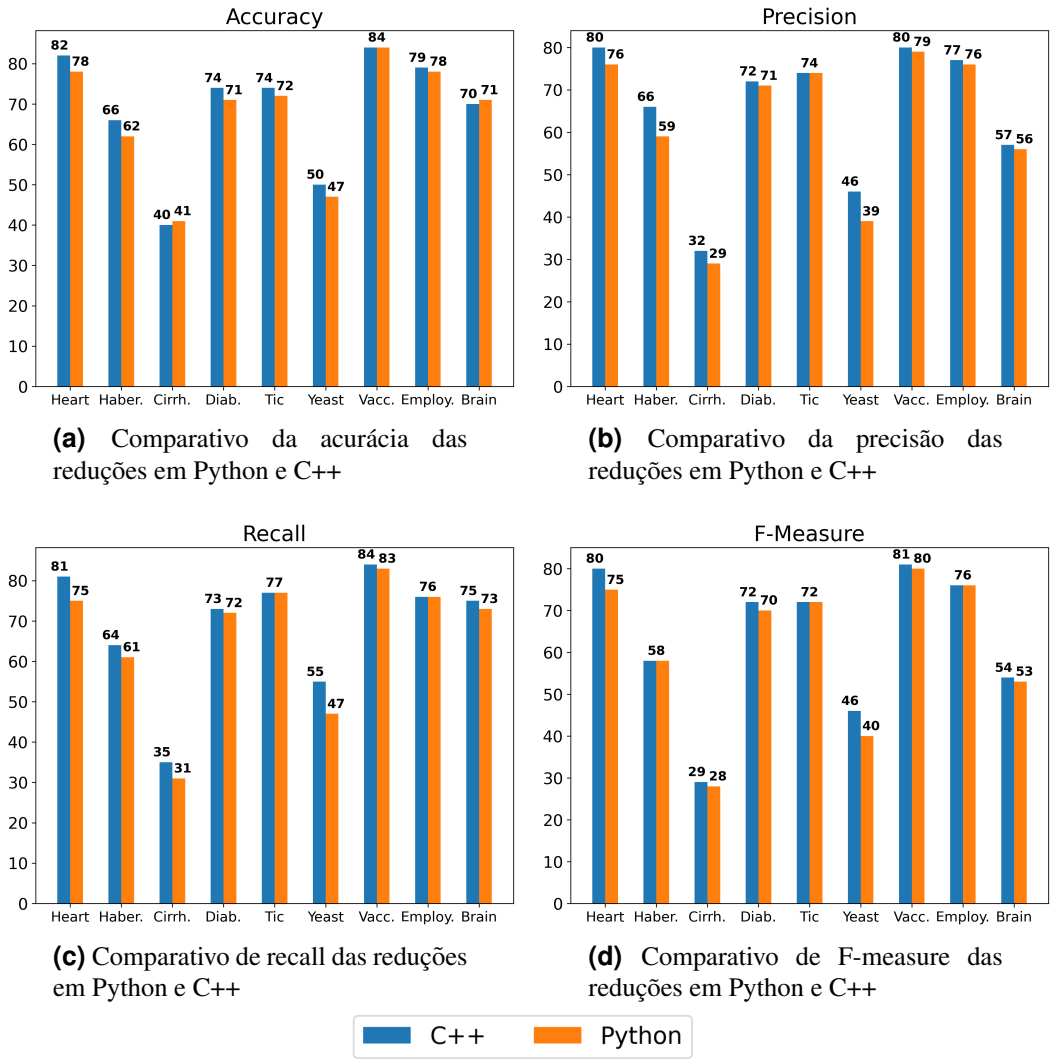
6. Avaliação dos Resultados

Os resultados apresentados nesta seção estão divididos da seguinte forma: i) inicialmente são analisadas as métricas da redução de instâncias, razão pela qual o algoritmo é utilizado neste artigo; ii) em seguida são analisados desempenho e escalabilidade, objetivo principal do artigo, com base nos tempos de execução de cada algoritmo para os diferentes *datasets*; e por fim, iii) contadores de hardware obtidos pela ferramenta Perf são avaliadas visando entender melhor os resultados de desempenho e escalabilidade.

6.1. Avaliação de Reduções de Instâncias

Para mostrar a eficácia do código em C++ e manter a mesma metodologia de validação, foi empregado um dos mesmos algoritmos de classificação utilizados durante o estudo de implementação do código original em Python, um *DecisionTreeClassifier* e, a partir das métricas obtidas, foram gerados alguns gráficos em Python para melhor análise. Para atestar a eficácia do código em C++, foram comparados os resultados de *accuracy*, *precision*, *recall* e *f-measure*.

Ao analisar as métricas da Figura 1, os *datasets* reduzidos em C++ mostram, em média, uma eficácia melhor ou equivalente às reduções realizadas pelo código Python original. Essa melhoria pode ser atribuída a dois fatores. Primeiro, a natureza pseudo-aleatória do ACO pode gerar resultados diferentes em cada execução, e os resultados obtidos pelo ACO em C++ foram superiores. Além disso, pode haver uma diferença na qualidade estatística dos números pseudoaleatórios gerados pelas linguagens. Inicialmente, utilizou-se a função *rand* da stdlib de C em C++, mas a substituição pela biblioteca *random*, que oferece melhor qualidade estatística, resultou em uma melhoria significativa nos resultados do ACO em C++. Isso sugere que o gerador de números pseudoaleatórios utilizado em Python pode ter uma qualidade estatística inferior.



(a) Comparativo da acurácia das reduções em Python e C++

(b) Comparativo da precisão das reduções em Python e C++

(c) Comparativo de recall das reduções em Python e C++

(d) Comparativo de F-measure das reduções em Python e C++

Figura 1. Comparativo de métricas C++ e Python

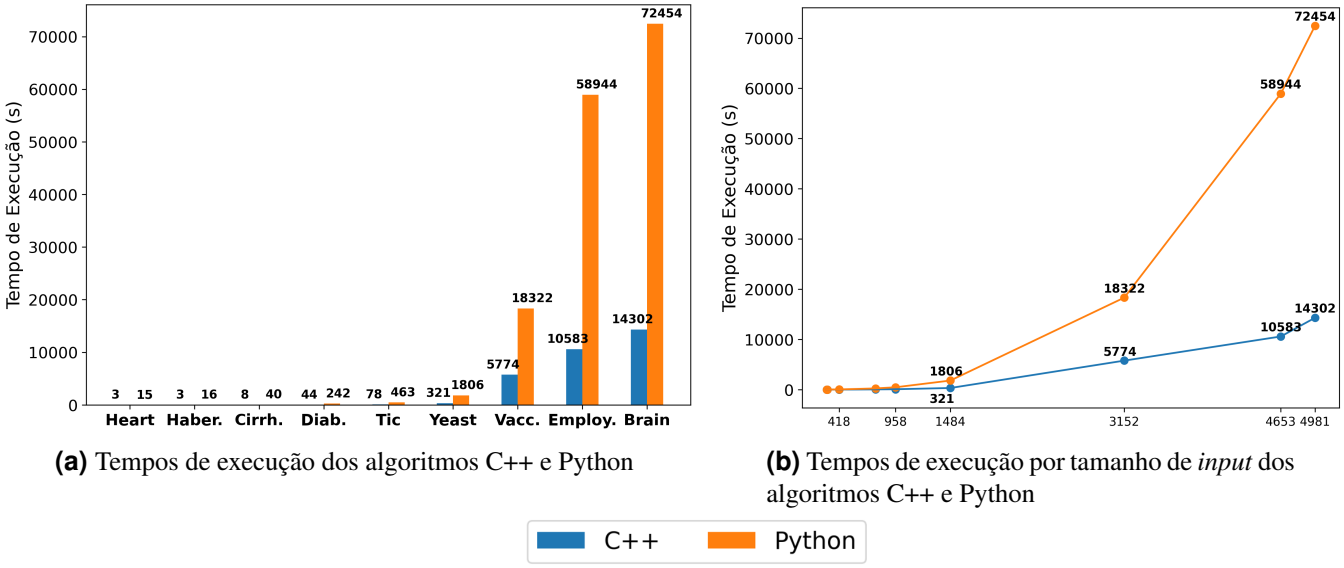
6.2. Avaliação de Desempenho e Escalabilidade

Os resultados dos tempos de execução foram divididos em 5 gráficos diferentes para uma melhor visualização, considerando a grande diferença existente entre os resultados de cada *dataset*.

Primeiro, na Figura 2a, é possível visualizar o tempo total que cada *dataset* levou para ser reduzido em ambos os algoritmos, sem considerar a análise do crescimento do tempo de execução em relação ao tamanho do *input*. Há diferença significativa no tempo de execução entre os códigos em C++ e Python.

Para analisar essa diferença de forma mais detalhada para os nove *datasets*, o gráfico da Figura 2b mostra o crescimento do tempo de execução em função do tamanho do *input* (dois primeiros *datasets* com valores sobrepostos nos pontos das curvas). Observa-se que, com o aumento da base de dados, o tempo de processamento no código em Python atinge valores absolutos muito maiores, destacando os benefícios do código em C++. No entanto, é importante considerar que, apesar dos valores absolutos inferiores

em C++, o tempo de processamento cresce proporcionalmente mais no ACO em C++. Ou seja, em bases de dados muito grandes, o tempo de execução pode se equiparar ou até superar o do ACO em Python. Vale ressaltar que para isso ocorrer, seriam necessárias bases extremamente grandes, o que pode tornar a execução inviável para ambos os algoritmos. De toda forma, o algoritmo em C++ se mostra mais escalável do que sua versão em Python, que com *inputs* menores já se torna inviável.

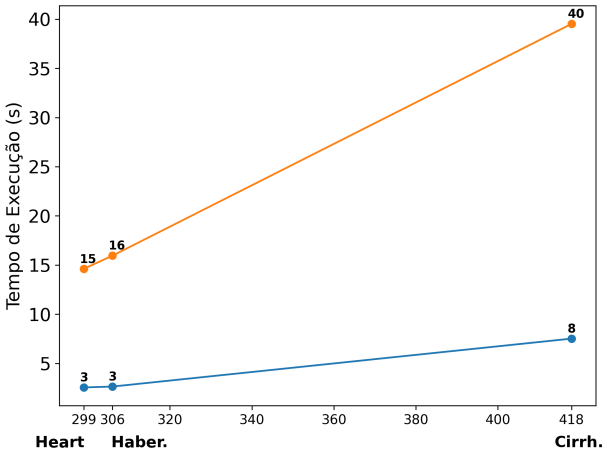


(a) Tempos de execução dos algoritmos C++ e Python

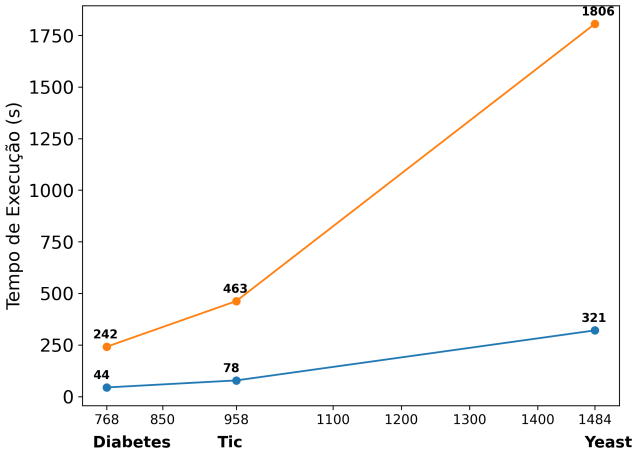
(b) Tempos de execução por tamanho de *input* dos algoritmos C++ e Python

Figura 2. Comparação de tempo de execução em C++ e Python

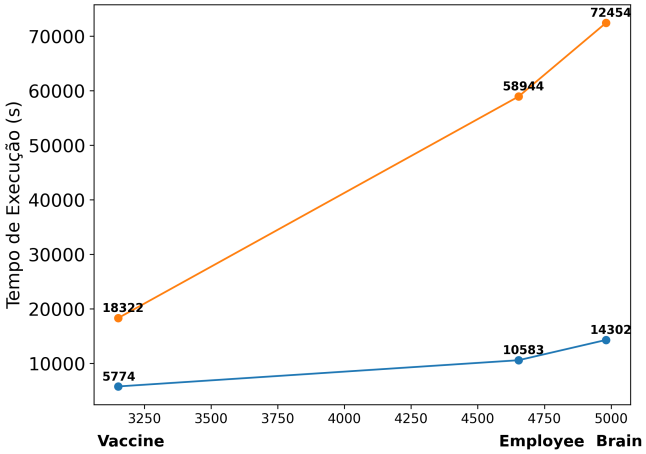
Para possibilitar uma visão mais específica da diferença de tempo em cada *dataset*, foram gerados 3 gráficos de tempo por *input*, cada um cobrindo *datasets* de uma categoria de tamanho, como apresentado pela Figura 3. Os gráficos revelam que o aumento do *input* resulta em um aumento muito mais intenso do tempo de execução no algoritmo Python em comparação ao C++. Em outras palavras, o algoritmo em C++ possui uma escalabilidade maior do que sua versão em Python. O crescimento do tempo de execução é mais lento em C++, o que garante sua viabilidade de execução à medida que há aumento no *input* de dados. Em *datasets* médios e pequenos, a diferença pode não ser suficiente para justificar a adoção do novo ACO. No entanto, considerando a natureza aleatória do ACO, que gera resultados variados a cada execução, a capacidade de executar o algoritmo mais vezes dentro de um período específico é uma vantagem significativa. Para *datasets* grandes, o ACO em C++ representa um avanço notável, permitindo realizar 5 reduções em um *dataset* de 4981 linhas, como o *brain_stroke*, no mesmo tempo necessário para uma única redução em Python.



(a) Tempo de execução por tamanho de *input* dos datasets pequenos



(b) Tempo de execução por tamanho de *input* dos datasets médios



(c) Tempo de execução por tamanho de *input* dos datasets grandes

Figura 3. Tempo de execução por categoria de *dataset* em C++ e Python

6.3. Avaliação de Contadores de Hardware

Considerando que os resultados obtidos pela ferramenta Perf apresentam um padrão consistente nos *datasets* usados neste artigo, foram selecionados 3 *datasets* — *cirrhosis*, *yeast* e *brain-stroke* — um de cada categoria, para análise das métricas coletadas. Para os três *datasets*, conforme Tabela 1, a diferença entre os resultados do Python e do C++ é expressiva. Observa-se que o Python apresenta um número significativamente maior de *page faults*, o que está diretamente relacionado o desempenho de execução do código e contribui para a grande diferença no tempo de execução entre as duas linguagens.

O maior número de *page faults* no Python também está associado a um maior número de *branch misses*, *L1 Load misses* e ciclos executados. Quando instruções e dados não estão presentes na memória principal, isso pode causar *branch misses* e *L1 Load*

Tabela 1. Contadores de Hardware

Contador	Pequeno (cirrhosis)		Médio (yeast)		Grande (brain-stroke)	
	C++	Python	C++	Python	C++	Python
Page faults	2823	54046	8519	63382	31032	366893
Cycles	30.322.516.329	199.016.522.657	1.340.207.895.244	6.953.297.110.566	58.101.882.858.150	270.056.171.740.948
Branch misses	78.301.956	565.309.388	3.789.472.861	10.612.258.271	169.013.978.359	397.557.586.290
L1 cache load misses	69.239.879	1.725.658.259	4.128.953.254	42.973.362.266	178.260.679.991	2.048.767.283.033

misses. Isso porque, após um *page fault*, a primeira tentativa de acessar um bloco da página recém carregada na memória principal gerará um *cache miss*. O maior número de ciclos no Python reflete diretamente o maior número de *page faults*, já que o processamento requer mais ciclos para concluir as operações quando os dados precisam ser recuperados da memória secundária.

Baseado na literatura e nos conceitos de arquitetura de computadores, é possível concluir que o Python pode apresentar mais *page faults* devido à sua natureza interpretada em vez de compilada. Um *page fault* ocorre quando um dado não é encontrado na memória principal. Como o Python é executado linha a linha por um interpretador em tempo de execução, isso pode resultar em operações de memória mais frequentes, aumentando a probabilidade de *page faults*. Além disso, o gerenciamento de memória do Python, com o *garbage collector* realizando operações constantes e automáticas, pode contribuir para um maior número de *page faults*, já que a memória é modificada com mais frequência.

7. Conclusão

O ACO original foi convertido para C++ com sucesso, apresentando um desempenho superior na nova versão. Isso pode ser explicado por diversos fatores, incluindo o fato de Python ser uma linguagem dinamicamente tipada e interpretada, enquanto C++ é compilado. Embora o algoritmo em C++ seja significativamente mais rápido, ainda há espaço para melhorias, dado que é sequencial. Além disso, é importante notar que a linguagem Python pode ser compilada, o que poderia potencialmente melhorar os resultados obtidos nesta pesquisa.

De acordo com os resultados encontrados nos experimentos, o ACO em C++ foi até 5 vezes mais rápido que a versão em Python, o que representa uma grande vantagem para a seleção de instâncias. Devido à natureza pseudo-aleatória do ACO, a capacidade de realizar múltiplas execuções em menos tempo pode levar a melhores resultados. Assim, espera-se que o meio acadêmico se beneficie dessa eficiência, permitindo realizar cinco reduções em C++ no tempo necessário para uma única redução em Python. Essa diferença de desempenho justifica pela escolha da versão em C++ do ACO.

Por fim, propõem-se alguns trabalhos futuros, com destaque para uma ampla caracterização de desempenho e a otimização da implementação da função *getbestsolution()* em C++ como prioridade. Também é importante realizar testes com a versão compilada do ACO em Python, pois isso pode impactar diretamente o desempenho do

algoritmo. Além disso, explorar a reorganização da lógica do código para permitir sua paralelização é um ponto interessante, o que poderia gerar um ganho ainda maior.

Referências

D. F. Bacon, S. L. Graham, e O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. doi: 10.1145/197405.197406.

G. Barany. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329163.

F. Bertoni e M. Pires. Aplicação de algoritmos evolutivos multiobjetivo na seleção de instâncias. In *Anais do XIII Simpósio Brasileiro de Sistemas de Informação*, pages 261–268, Porto Alegre, RS, Brasil, 2017. SBC. doi: 10.5753/sbsi.2017.6051.

T. Borovicka, M. J. Jr., P. Kordik, e M. Jirina. Selecting representative data sets. In A. Karahoca, editor, *Advances in Data Mining Knowledge Discovery and Applications*, chapter 2. IntechOpen, Rijeka, 2012. doi: 10.5772/50787.

M. Dorigo, M. Birattari, e T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006. doi: 10.1109/MCI.2006.329691.

C. Gong, Z. gang Su, P. hong Wang, Q. Wang, e Y. You. Evidential instance selection for k-nearest neighbor classification of big data. *International Journal of Approximate Reasoning*, 138:123–144, 2021. doi: https://doi.org/10.1016/j.ijar.2021.08.006.

M. Ismail e G. E. Suh. Quantitative overhead analysis for python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–47, 2018. doi: 10.1109/IISWC.2018.8573512.

X. Leroy. An overview of Types in Compilation. In *TIC 1998: workshop Types in Compilation*, volume 1473 of *LNCS*, pages 1–8. Springer, Kyoto, Japan, Mar. 1998. doi: 10.1007/BFb0055509.

A. C. Medeiros. Análise de desempenho do algoritmo colônia de formigas para seleção de instâncias. Trabalho de conclusão de graduação, PUC Minas, 2022.

S. Nanz e C. A. Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, 2015. doi: 10.1109/ICSE.2015.90.

J. A. Olvera-López, J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, e J. Kittler. A review of instance selection methods. *Artificial Intelligence Review*, 34(2):133–143, 2010. doi: 10.1007/s10462-010-9165-y.

S. Saha, P. S. Sarker, A. A. Saud, S. Shatabda, e M. Hakim Newton. Cluster-oriented instance selection for classification problems. *Information Sciences*, 602:143–158, 2022. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2022.04.036.

N. Spolaôr, E. A. Cherman, M. C. Monard, e H. D. Lee. Relieff for multi-label feature selection. In *2013 Brazilian Conference on Intelligent Systems*, pages 6–11, 2013. doi: 10.1109/BRACIS.2013.10.

B. Stroustrup. An overview of c++. *ACM SIGPLAN Notices*, 21(10):7–18, jun 1986. doi: 10.1145/323648.323736.

Y. Ueda e M. Ohara. Performance competitiveness of a statically compiled language for server-side web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, 2017. doi: 10.1109/ISPASS.2017.7975266.