

Métodos de Otimização Aplicados à Detecção de Plágio em Exercícios de Programação na Linguagem C

Rafael Cavalcante Chaves¹, Kervén Maciel Monteiro de Albuquerque¹, João Pedro Oliveira Santos¹, Thiago Gouveia¹,

Instituto Federal de Educação Ciência e Tecnologia da Paraíba
João Pessoa — PB

Autor para correspondência: `chaves.rafael@academico.ifpb.edu.br`

RESUMO

Este trabalho apresenta uma extensão de um modelo de detecção de plágio em exercícios de programação baseado em grafos. O modelo original, que utiliza a resolução do Problema do Menor Caminho com Premiação Máxima (PMCPM) em um grafo bipartido completo para quantificar a similaridade entre códigos Python, foi adaptado para suportar a linguagem C. Para resolver o PMCPM, são propostos dois métodos: um exato, baseado em Programação Dinâmica, e um heurístico, combinando a Meta-heurística Multi-Start com uma busca local baseada na estratégia VND (*Variable Neighborhood Descent*). Além disso, foi construído um conjunto de dados de *benchmark* baseado em exercícios reais de programação para validar o modelo e os métodos propostos. Os experimentos computacionais demonstraram que o modelo é capaz de representar as similaridades entre códigos-fonte na linguagem C de forma consistente, abrindo caminho para futuras pesquisas.

PALAVRAS CHAVE. Detecção de plágio, Meta-heurísticas, Programação Dinâmica, Grafos.

Tópicos: Detecção de plágio. Similaridade entre códigos-fonte. Problema do Menor Caminho com Premiação Máxima. Meta-heurística. Multi-Start-VND. Programação Dinâmica, Grafos Bipartidos Completos.

ABSTRACT

This paper presents an extension of the plagiarism detection model in programming exercises based on graphs. The original model, which uses the resolution of the Prize Collecting Minimum Path Problem (PMCPM) in a complete bipartite graph to quantify the similarity between Python codes, was adapted to support the C language. To solve the PMCPM, two methods are proposed: an exact method based on Dynamic Programming, and a heuristic method combining the Multi-Start Meta-heuristic with a local search based on the Variable Neighborhood Descent (VND) strategy. Additionally, a benchmark dataset based on real programming exercises was constructed to validate the model and the proposed methods. Computational experiments demonstrated that the model is capable of consistently representing the similarities between source codes in the C language, paving the way for future research.

KEYWORDS. Plagiarism Detection, Metaheuristics, Dynamic Programming, Graphs.

Topics: Plagiarism Detection. Source Code Similarity. Prize-Collecting Shortest Path Problem. Metaheuristic. Multi-Start-VND. Dynamic Programming. Complete Bipartite Graphs.

1. Introdução

Atualmente, a tecnologia desempenha papel central nas nossas vidas, transformando, entre outros, comunicação, trabalho, aprendizado e entretenimento. Por um lado, a Internet e os dispositivos móveis facilitam conexões globais e trocas rápidas de informações. Por outro, automação, ferramentas digitais e Inteligência Artificial aumentam vertiginosamente a eficiência, enquanto plataformas online democratizam o acesso à cultura e ao conhecimento. Nesta esteira, é possível observar que o ensino de programação tem ganhando cada vez mais importância. Neste contexto, os Juízes Online, ferramentas para julgamento automático de códigos-fonte enviados como solução para determinado problema de programação, surgem como importante ambiente para auxiliar o ensino desta disciplina [Gouveia et al., 2023].

No entanto, a popularização do ensino da programação carrega consigo o sério problema do plágio de códigos-fonte. De acordo com Cheers et al. [2021], o plágio ocorre quando um estudante se apropria de código ou trechos de códigos obtidos a partir de outras fontes e os utiliza indevidamente como se fossem seus. Tal comportamento, além de antiético, pode ser extremamente prejudicial ao processo de ensino e aprendizado. Deste modo, se faz crucial que os professores disponham de métodos eficientes para detectar esse tipo de ação, uma vez que a verificação manual demandaria significativo esforço. Dentre as estratégias de combate à cópia de código disponíveis, destacam-se as três mais populares: JPlag¹, Sherlock² e MOSS³ [Horváth e Pietriková, 2024].

Por outro lado, diversos métodos para identificação de plágio de código têm sido propostos na literatura recente, como o modelo de Chaves et al. [2023], que representa a similaridade entre códigos como um grafo bipartido completo e permite identificar e quantificá-la a partir da resolução do Problema do Menor Caminho com Premiação Máxima (PMCPM) em um grafo bipartido completo. No entanto, uma das limitações deste modelo é que ele se aplica apenas a códigos escritos em linguagem de programação Python. Em contraste, de acordo com Rong et al. [2023], a linguagem de programação C tem sido amplamente empregada em diversas disciplinas de Introdução à Programação, tanto em cursos de nível Superior, quanto no nível Médio. Segundo Cui et al. [2023], isso se deve ao fato de que a linguagem C pode estabelecer uma base sólida de programação, permitindo aos alunos realizarem pesquisas e desenvolverem *software* de forma eficaz. Adicionalmente, esta linguagem é especialmente útil em cursos de Engenharia da Computação e Elétrica, devido à sua natureza mais próxima ao *hardware*.

Neste contexto, este trabalho estende o modelo de Chaves et al. [2023], adicionando suporte à linguagem de programação C. Adicionalmente, são propostos dois métodos para resolução do PMCPM gerado para comparação de dois códigos-fonte: um método exato baseado em Programação Dinâmica, e um método heurístico combinando a Meta-heurística Multi-Start com uma busca local baseada na estratégia VND (do inglês, *Variable, Neighbourhood Descent*), composta por quatro novas heurísticas de busca local. Adicionalmente, é construído um conjunto de dados de *benchmark* baseado em respostas para exercícios reais de programação, e são apresentados experimentos para validação do modelo e dos métodos propostos.

Para tal, o restante deste trabalho está organizado como segue: A seção 2 apresenta uma discussão acerca de trabalhos sobre detecção de plágio. A seção 3 detalha o modelo proposto, o algoritmo baseado em Programação Dinâmica, a heurística de construção, as quatro estratégias de busca local, e o método Multi-Start-VND. A seção 4 discute a estratégia para geração de instâncias, descreve os experimentos realizados, e os resultados obtidos. Por fim, a seção 5 traz as considerações finais, assim como sugestões para continuidade da pesquisa.

¹<https://helmholtz.software/software/jplag>

²<https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/>

³<https://theory.stanford.edu/aiken/moss/>

2. Trabalhos Relacionados

Esta seção apresenta uma pequena revisão da literatura sobre detecção de plágio e/ou identificação de similaridade em códigos-fonte. Por simplicidade, a revisão será baseada em dois dos surveys mais abrangentes da literatura. Por fim, será apresentada uma pequena discussão acerca dos trabalhos apresentados, comparando-os, ainda que superficialmente, com o modelo proposto.

Agrawal e Sharma [2016] realizam uma revisão da literatura com foco na detecção de plágio em códigos-fonte, listando ferramentas e técnicas comumente utilizadas. O autor também sugere a categorização de acordo com as abordagens utilizadas para verificar a similaridade, que são baseadas em: (1) Texto, os códigos são tratados como sequências de strings, e algoritmos de similaridade textual são aplicados para identificar semelhanças. (2) Tokens, os códigos são convertidos em sequências de tokens utilizando *lexers*, permitindo comparação sintática e textual. (3) Árvores, utilizando *parsers*, árvores sintáticas abstratas são geradas, possibilitando uma comparação estrutural e sintática entre os códigos. (4) Grafos, os códigos são frequentemente convertidos em grafos de dependência, permitindo a verificação tanto sintática quanto semântica dos programas. (5) Métricas, conjuntos de características são extraídos dos códigos, e a similaridade é estimada com base na semelhança desses conjuntos de características.

Zakeri-Nasrabadi et al. [2023] apresentam uma revisão extensiva da literatura sobre a detecção de similaridade em códigos-fonte, não se restringindo apenas à aplicação na detecção de plágio. Foram considerados trabalhos que utilizavam o conceito de similaridade em outras aplicações, como na identificação de *malwares*, predição de defeitos e recomendação de código. Nesse contexto, foram analisados 10.000 artigos, dos quais 136 foram selecionados como estudos primários. Com base nas análises, os autores propõem categorias semelhantes às citadas anteriormente, mas com o acréscimo de métodos baseados em aprendizado de máquina, processamento de imagem e testes, além dos métodos híbridos, que combinam diversas abordagens em um único método. Entre os estudos primários, os métodos mais comuns eram os híbridos (27,94%), seguidos pelo aprendizado de máquina (18,38%), tokens (16,91%), grafos (11,76%), árvores (11,03%) e textual (7,35%).

A partir da análise dos trabalhos apresentados nesta seção, pode-se verificar que o modelo proposto por Chaves et al. [2023] se encaixa na classe das soluções Híbridas. Dentre os trabalhos diretamente citados pelos *Surveys*, não foi possível encontrar nenhuma solução que realize a análise de *tokens* para geração de grafos que representem a similaridade ou que a quantifique a partir da resolução de problemas de otimização. Por tanto, podemos considerar o método proposto como uma nova abordagem para o estudo da detecção de similaridade entre códigos-fonte.

3. Metodologia

Esta seção é dedicada à descrição dos métodos propostos neste trabalho. Inicialmente, por motivo de auto-contenção, é apresentado o modelo de Chaves et al. [2023], assim como a definição do PMCPM derivado deste e sua extensão para a linguagem de programação C. Em sequência, são apresentados dois novos métodos para resolução do PMCPM, o primeiro utilizando Programação Dinâmica, e o segundo combinando a meta-heurística Multi-Start com uma estratégia VND.

3.1. Modelo de Chaves et al. [2023]

Dados dois códigos-fonte $A = (a_1, a_2, \dots, a_n)$, e $B = (b_1, b_2, \dots, b_m)$, representados como sequências de instruções, é possível compará-los afim de verificar o seu grau de similaridade, e assim, a possível ocorrência de plágio. Caso exista alguma permutação das sequências de modo que a maioria dos pares de instruções de mesmo índice apresentem alta similaridade, então podemos afirmar que existe grande probabilidade de que um destes códigos seja fruto de plágio.

Desta forma, é possível modelar a similaridade entre os códigos A e B como um problema de caminho mínimo com coleta de prêmios. Seja $G = (V, W, E)$ um grafo bipartido orientado e

completo, formado pelos conjuntos de vértices V e W , e por um conjunto de arcos E . Para cada instrução do código-fonte A (B) há um vértice associado em V (W). Por sua vez, o conjunto E é composto por arcos que partem de cada $a_i \in V$ em direção a cada $b_j \in W$, representando a similaridade entre as respectivas instruções, e arcos que partem de cada b_j em direção a cada a_i , utilizados para ajudar na identificação de sequências contíguas de código similar. Adicionalmente, seja $w : E \rightarrow \mathbb{R}$ uma função que retorna a distância entre dois vértices de G , e P um valor constante de premiação.

Uma vez definido o grafo bipartido completo $G = (V, W, E)$, a partir dos códigos-fonte A e B , o Problema do Menor Caminho com Premiação Máxima (PMCPM) tem como objetivo encontrar um caminho simples $C = \{e_1, e_2, \dots, e_s\} \subset E$, composto por s arcos, que parte de um vértice em V , termina em um vértice em W , e maximiza a função objetivo FO , definida na Equação 1. A primeira parte da FO premia cada vértice do caminho com um valor P , enquanto a segunda parte penaliza as diferenças entre cada instrução, assim como a sua distância relativa nos códigos.

$$FO = (s + 1) \cdot P - \sum_{e \in C} w(e) \quad (1)$$

Por fim, resta a definição formal da função w . Seja $w(e) = \vec{w}(e)$, caso e parta de V para W , e $w(e) = \overleftarrow{w}(e)$, caso contrário. Considere que cada instrução a_i é uma sequência de *tokens* $(t_1, t_2, \dots, t_k)$, sendo um *token* definido como uma *substring* de a_i , de acordo com as regras sintáticas da linguagem. Cada *token* possui um tipo associado, de acordo com sua função no programa, podendo ser considerado, entre outros, uma palavra-chave. O cálculo da distância entre dois *tokens* t_a e t_b é realizado segundo as condições: (1) Se apenas t_a ou apenas t_b é palavra-chave, a distância é 1; (2) Se t_a e t_b possuem tipos diferentes, a distância também é 1; (3) Se os tipos e as *strings* de t_a e t_b são iguais, então a distância é 0; (4) Caso contrário, este valor é dado pelo algoritmo distância de edição entre t_a e t_b (Equação 2), normalizado no intervalo $[0, 0.5]$ pelo tamanho da maior *string*.

$$\delta(t_a, t_b) = \begin{cases} |t_a|, & \text{se } |t_b| = 0 \\ |t_b|, & \text{se } |t_a| = 0 \\ \delta(t_a[1..], t_b[1..]), & \text{se } t_a[0] = t_b[0] \\ 1 + \min \begin{cases} \delta(t_a, t_b[1..]) \\ \delta(t_a[1..], t_b) \end{cases}, & \text{caso contrário} \end{cases} \quad (2)$$

Dado que é possível calcular a distância entre t_a e t_b , pode-se utilizar esses resultados para encontrar a distância entre c_a e c_b , onde c_a (c_b) representa o conjunto de *tokens* que compõem uma instrução a_i (b_j). Para tal, lança-se mão de uma segunda versão da distância de edição (Equação 3), que uma vez calculado, seu resultado deve ser normalizado no intervalo $[0, 1000]$ pelo maior valor entre $|c_a|$, $|c_b|$. Desta forma, é obtido o peso do arco que parte de $a_i \in V$ para $b_j \in W$, valor da função $\vec{w}(e)$.

$$\delta'(c_a, c_b) = \begin{cases} |c_a|, & \text{se } |c_b| = 0 \\ |c_b|, & \text{se } |c_a| = 0 \\ \delta'(c_a[1..], c_b[1..]), & \text{se } c_a[0] = c_b[0] \\ \min \begin{cases} 1 + \delta'(c_a, c_b[1..]) \\ 1 + \delta'(c_a[1..], c_b) \\ 2 \cdot \delta(c_a[0], c_b[0]) + \delta'(c_a[1..], c_b[1..]) \end{cases}, & \text{caso contrário} \end{cases} \quad (3)$$

Por fim, resta calcular $\overleftarrow{w}(e)$, o peso dos arcos que partem de $b_j \in W$ para $a_i \in V$. Estes arcos, por sua vez, auxiliam na quantificação da continuidade de uma solução do PMCPM, estabelecendo que quanto maior a distância em linhas entre b_j e a_i , maior será $\overleftarrow{w}(e)$. Tal quantificação é útil para identificar intervalos contíguos de código plagiado, já que um caminho com poucos "saltos" tende a acumular menos peso a partir destes arcos.

Para isso, dado um $b_j \in W$ e um $a_i \in V$, o cálculo é realizado obedecendo às seguintes condições: (1) Se $i = j + 1$, que indica que o próximo par de instruções está na linha seguinte a b_j , então $\overleftarrow{w}(e) = 0$; (2) Se $i > j + 1$, onde há quebra da continuidade com um "salto" para linhas posteriores, então utilize a Equação 4; (3) Caso contrário, onde há quebra da continuidade com um "salto" para linhas anteriores, considerada mais forte, então utilize a Equação 5.

$$\overleftarrow{w}(e) = \min(50 + (i - j - 2) \times 10, 200) \quad (4)$$

$$\overleftarrow{w}(e) = \min(100 + (j - i) \times 20, 400). \quad (5)$$

3.2. Extensão do Modelo para Linguagem C

Estender o modelo de Chaves et al. [2023] para linguagem de programação C requer uma ferramenta capaz de transformar um código-fonte escrito nesta linguagem em um conjunto de *tokens*, processo conhecido como tokenização, realizado por um software denominado *lexer*. Uma vez que o pacote *tokenizer*⁴, utilizado em Chaves et al. [2023], não oferece suporte à tokenização de códigos em C, foi escolhido o pacote *clang*⁵, um conjunto de *bindings* para as funcionalidades da *libclang*. Para integrar o *lexer* do *clang* ao *script*, foi necessário adaptar os parâmetros retornados (como *string*, tipo, etc.) por cada *token*, de modo a torná-los compatíveis com os procedimentos já implementados.

Antes da realização da tokenização, os códigos de entrada passam por uma fase de pré-processamento. Nesse estágio, removem-se os comentários e realiza-se a formatação dos códigos para desfazer alterações como a adição de espaços em branco, a quebra de instruções em várias linhas, múltiplas instruções em uma única linha, etc. Dessa forma, ao final, cada linha pode ser considerada uma instrução.

A Figura 1, mostra o grafo bipartido completo resultante da modelagem, com os pesos dos arcos não utilizados na solução ótima em cinza. Já em cores azuis, está o caminho ótimo obtido pela resolução do PMCPM, utilizando a premiação P fixada em 200. É importante ressaltar que todos os valores de parâmetros constantes do modelo foram obtidos mediante exaustivos testes empíricos, objetivando encontrar um equilíbrio entre os prêmios e as penalidades. Tais testes não puderam ser reportados por conta de limitações de espaço.

3.3. Programação Dinâmica para o PMCPM

Considerando o modelo apresentado na subseção 3.2, é possível resolver o PMCPM associado ao testar todas as permutações de pareamentos possíveis. A partir de cada pareamento, utilizamos os valores obtidos na modelagem do grafo bipartido para resolver a Equação 6. Esta abordagem nos permite determinar o valor máximo entre todas as permutações e, consequentemente, determinar a solução ótima do PMCPM para essa instância.

$$\max_{K \in [1, |V|]} \sum_{i=1}^K (P - w(e_{b_{i-1}, a_i}) - w(e_{a_i, b_i})) \quad (6)$$

Baseado na abordagem apresentada, é possível implementar um *Backtracking* com complexidade $O(|V|! \times |W|!)$. Devido a complexidade extremamente alta, as instâncias que podem ser

⁴<https://pypi.org/project/tokenizer/>

⁵<https://pypi.org/project/clang/>

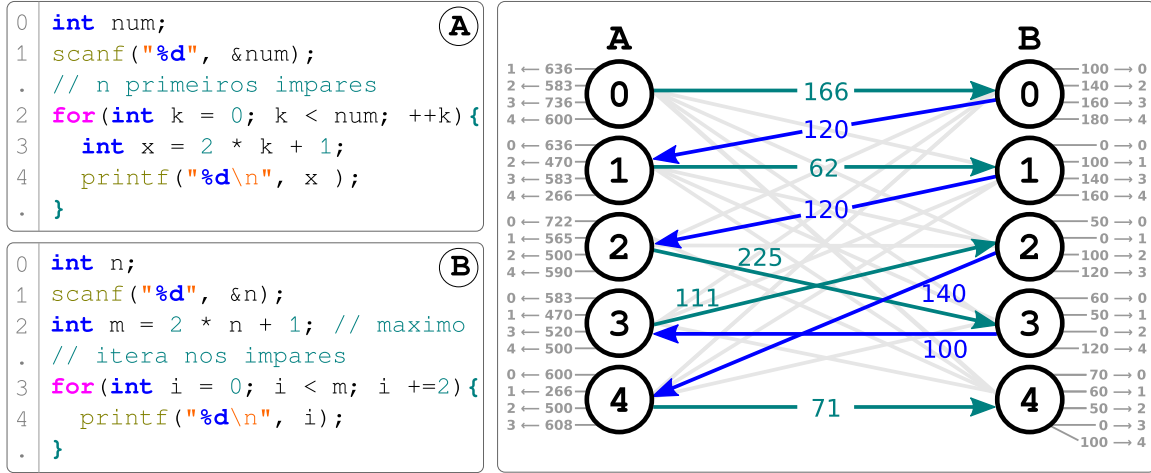


Figura 1: Exemplo de extensão do modelo proposto para linguagem C

resolvidas por essa abordagem em tempo útil são muito pequenas e acabam não sendo interessantes para a demonstração da identificação de plágio através do PMCPM. No entanto, é possível perceber que há espaço para a aplicação de técnicas de memoização e, por consequência, realizar a conversão do *Backtracking* em Programação Dinâmica *Top-Down*, como descrito no Algoritmo 1.

Algoritmo 1: Programação Dinâmica para o PMCPM

```

1 Procedimento PD ( $V', W', b, G = (V, W, E)$ )
2     se ( $|V'| = |V|$ ) ou ( $|W'| = |W|$ ) então retorna 0 ;
3     se VerificarEstado( $V', W', b$ )  $\neq$  Nil então retorna ResultadoEstado( $V', W', b$ ) ;
4      $best \leftarrow 0$ ;
5     para todo  $a' \in V \setminus V'$  faça
6         para todo  $b' \in W \setminus W'$  faça
7              $best \leftarrow \text{Max}(best, PD(V' + a', W' + b', b', G) - w(e_{a',b}) - w(e_{a',b'}) + P));$ 
8     SalvarEstado( $V', W', b, best$ );
9     retorna  $best$ ;
    
```

Para um determinado nó da árvore de recursão, os parâmetros que podem influenciar o resultado ótimo de suas subárvores são o conjunto de instruções de V pareadas (V'), o conjunto de instruções de W pareadas (W') e o último b_j pareado. Nesse sentido, podemos considerar que todos os nós que possuem os mesmos parâmetros estão no mesmo estado. Consequentemente, é possível reutilizar o resultado obtido para as suas subárvores. Como resultado das melhorias propostas, a nova abordagem consegue lidar com instâncias mais interessantes ao problema. No entanto, ele se torna inviável a medida que as instâncias crescem, devido a sua complexidade $O(2^{|V|} \times 2^{|W|} \times |W|)$ de espaço e $O(2^{|V|} \times 2^{|W|} \times |V| \times |W|^2)$ de tempo.

3.4. Método Multi-Start-VND

Devido as limitações do método de Programação Dinâmica para instâncias maiores, este trabalho propõe a resolução do PMCPM combinando a meta-heurística Multi-Start com uma estratégia de busca local VND, método que denominamos *Multistart-VND* (Algoritmo 2). A função objetivo utilizada é baseada na Equação 6, e cada solução S é representada como uma sequência de pares $(p_1, p_2, \dots)$, onde cada par é formado por dois vértices $a \in V$ e $b \in W$.

O *Multistart-VND* recebe três parâmetros: α , o fator de aleatoriedade para a heurística de construção; T , o tempo limite de execução; e G , o grafo bipartido modelado partir dos códigos A

e B . A partir desses parâmetros, o algoritmo gera uma solução e realiza tentativas de melhorias a partir de quatro buscas locais. Ao final da execução, a melhor entre todas as soluções geradas é retornada.

Algoritmo 2: Heurística Multi-Start-VND

```

1 Procedimento Multi-Start-VND ( $\alpha, T, G = (V, W, E)$ )
2    $S^* \leftarrow \emptyset$ ;
3   enquanto  $t < T$  faça
4      $S \leftarrow \text{ConstruirSolucao}(\alpha, G)$ ;
5      $S \leftarrow \text{TrocarVertices}(S, G)$ ;
6      $S \leftarrow \text{RemoverParesRuins}(S, G)$ ;
7      $S \leftarrow \text{TrocarPares}(S, G)$ ;
8      $S \leftarrow \text{InserirPares}(S, G)$ ;
9      $S^* \leftarrow \text{Melhor}(S^*, S)$ ;
10  retorna  $S^*$ 

```

O Algoritmo 3 descreve o processo de construção de uma solução S . O procedimento consiste na criação e inserção sucessiva de pares $p = (a, b)$ em S . O pareamento dos vértices a e b pode ser realizado com viés randômico ou guloso, sendo essa decisão realizada probabilisticamente com base no α escolhido. No caso do pareamento ter viés randômico, a e b são escolhidos aleatoriamente e inseridos na posição que maximize o impacto na solução S . Caso o pareamento seja realizado de maneira gulosa, um a é escolhido randomicamente e então procura-se o b que formará o p com melhor impacto na solução.

Algoritmo 3: Heurística de Construção

```

1 Procedimento Construir-Solução ( $\alpha, G = (V, W, E)$ )
2    $S \leftarrow \emptyset$ ,  $V' \leftarrow V$ ,  $W' \leftarrow W$ ;
3   enquanto  $(V' \neq \emptyset)$  e  $(W' \neq \emptyset)$  faça
4      $a' \leftarrow$  Retire um vértice aleatório de  $V'$ ;
5     se  $\text{Rand}(0, 1) < \alpha$  então
6        $b' \leftarrow$  Retire um vértice aleatório de  $W'$ ;
7        $S \leftarrow$  insira  $p = (a', b')$  onde maximize  $FO(S)$ ;
8     senão
9        $S' \leftarrow S$ ;
10      para todo  $b'' \in W'$  faça
11         $S'' \leftarrow$  insira  $p = (a', b'')$  onde maximize  $FO(S)$ ;
12        se  $S''$  é melhor que  $S'$  então  $S' \leftarrow S''$ ,  $b' \leftarrow b''$ ;
13      Remova  $b'$  de  $W'$ ,  $S \leftarrow S'$ ;
14  retorna  $S$ 

```

Após a geração de S , a primeira busca local, *TrocarVertices*, tem por objetivo realizar melhorias a partir de trocas sucessivas nas posições dos $b \in W$. Para manter algum grau de diversificação, o intervalo $I = \{1, 2, \dots, |S| - 1\}$ é iterado de maneira aleatória. Para cada $i \in I$, são testadas trocas das posições de $b_i \in p_i$ com todo $b \in W$. Caso uma troca seja benéfica, ela é efetivada.

A segunda busca local, *RemoverParesRuins*, é responsável por construir uma nova solução S' a partir de S , ignorando os pares considerados ruins da solução original. Para cada $p \in S$, verifica-se o impacto de sua remoção. Caso essa ação tenha um impacto positivo, p é considerado ruim e é ignorado. Caso contrário, adiciona-se p em S' . Ao final, uma nova solução é construída utilizando apenas pares considerados bons.

Após a eliminação dos pares ruins, aplica-se a busca local *TrocarPares*. Esse procedimento reposiciona os pares de maneira gulosa, visando reduzir a punição recebida pelos arcos de retorno. Para garantir algum grau de diversificação, itera-se sobre o intervalo $I = \{1, 2, \dots, |S| - 1\}$ de maneira aleatória. Para cada $i \in I$, testa-se a troca de p_i com cada $p \in S$. Caso uma troca tenha impacto positivo, ela é efetivada.

A última busca local, *InserirPares*, tem como objetivo melhorar a solução através da inserção de pares compostos pelos $a, b \notin S$. Para isso, utiliza-se um procedimento semelhante ao de inserção com viés guloso da heurística de construção (Algoritmo 3), com a diferença de que apenas inserções que melhorem S são efetivadas.

4. Experimentos Computacionais

Esta seção reporta os experimentos computacionais realizados para validação do método proposto em termos de eficácia, i.e. demonstrar que ele é capaz de detectar similaridades entre códigos-fonte na linguagem C, e eficiência. Nesse sentido, são descritos os procedimentos para geração das instâncias, e em seguida, são apresentados os resultados obtidos.

4.1. Instâncias

Com intuito de demonstrar a fidedignidade dos valores de similaridade obtidos através do modelo proposto, efetuamos a comparação entre códigos-fonte gerados por meio de modificações sucessivas a partir de uma versão original. Espera-se então, que a similaridade entre duas versões seja compatível com a quantidade de alterações realizadas entre elas. Wen et al. [2019] lista modificações feitas em códigos copiados e as ordena de acordo com o nível de complexidade. Utilizando tais alterações como referência, propomos as seguintes modificações: (1) Mudança na formatação; (2) Adição de comentários; (3) Modificação no nome de variáveis; (4) Substituição de Valores por Constantes ou *Defines*; (5) Alteração na ordem de termos em expressões matemáticas/lógicas; (6) Mudança na ordem entre instruções independentes; (7) Alteração na estrutura do código (Adicionar funções, modificar estruturas de repetição e decisão).

Após a aplicação sequencial das alterações listadas, obtém-se oito códigos no total (um original e sete novos). Em seguida, utilizamos o modelo proposto para gerar 64 grafos que representam a similaridade entre quaisquer duas versões do código. Para cada grafo, aplica-se os métodos de resolução do PMCPM apresentados neste trabalho e, por fim, é possível obter uma matriz 8x8 contendo o nível de similaridade entre os códigos de uma mesma instância.

Os códigos utilizados estão escritos na linguagem C, e as versões originais foram obtidas a partir da resolução de problemas presentes em listas de exercícios da disciplina de Algoritmos e Lógica de Programação do curso de Análise e Desenvolvimento de Sistemas do IFPB - Campus Cajazeiras. Para garantir a consistência dos resultados, assegurou-se que todos os códigos de uma mesma instância apresentem saídas iguais para as mesmas entradas.

4.2. Resultados

Os experimentos foram realizados em um computador com sistema operacional Ubuntu 20.04, com processador Intel Core i5 de 12ª geração (2,5GHz), 32GB de RAM e 1 TB de SSD. Foram utilizadas nove instâncias, identificadas como $I_1, I_2, \dots, I_9$, geradas a partir do procedimento descrito na Seção 4.1. Os códigos utilizados possuem entre 9 e 44 linhas, e após o pré-processamento e a tokenização, apresentaram entre 10 e 38 instruções.

Devido às suas limitações, o método de Programação Dinâmica foi aplicado exclusivamente na instância I_9 , cujos códigos continham entre 10 e 15 instruções. Para cada grafo gerado, houve apenas uma execução deste método. Por outro lado, o método heurístico foi aplicado em todas as instâncias, com 10 execuções realizadas para cada grafo gerado.

Os métodos exato e heurístico possuem parâmetros que podem ser ajustados para otimizar suas execuções. Para o presente experimento, esses parâmetros foram definidos como: a premiação

(P) dada a cada vértice adicionado à solução foi fixada em 200; o tempo limite (T) para o processamento do *Multistart-VND* foi estipulado em 1 segundo; e o fator de aleatoriedade (α) para a heurística de construção foi fixado em 20%.

A Tabela 1 apresenta uma comparação entre os resultados obtidos através da resolução da Instância I_9 usando *Programação Dinâmica* (Seção 3.3) e do *Multistart-VND* (Seção 3.4). Os campos A e B indicam os índices dos códigos utilizados para gerar o grafo, enquanto $|A|$ e $|B|$ representam a quantidade de instruções em cada código. O campo FN indica o fator de normalização para o PMCPM no grafo, cujo valor é obtido através da Equação 7.

$$FN = 2 \times P \times \min(|A|, |B|) \quad (7)$$

Para os campos referentes ao método exato, Rt representa a taxa de similaridade entre os códigos normalizada no intervalo $[0, 1]$, e $t(ms)$ refere-se ao tempo de execução. Entre os campos referentes ao método heurístico, temos a taxa de similaridade média normalizada Rt_m , o tempo médio para geração da melhor solução $t_m(ms)$ e BL , a razão média de melhoria de soluções obtidas respectivamente pelas buscas locais *TrocarVertices*, *RemoverParesRuins*, *TrocarPares* e *InserirPares*.

Sobre os resultados, pode-se notar que foi possível executar a *Programação Dinâmica* para todos os grafos da instância I_9 . Também é perceptível que o tempo de execução cresce exponencialmente à medida que o número de instruções aumenta. A diferença no tempo de processamento entre a execução para o menor grafo e o maior é de cerca de 343.615%, tornando sua utilização rapidamente inviável. Já em relação ao *Multistart-VND*, podemos perceber que o ótimo global foi atingido para todos os grafos utilizando, no máximo, 0,2% do tempo de processamento exigido pelo método exato. Também é importante pontuar que não foi necessário adicionar nenhuma medida de dispersão para o campo Rt_m , pois todas as execuções obtiveram o melhor valor, assim, não havendo variação. A respeito das buscas locais, percebe-se que tanto *TrocarVertices* como *TrocarPares* atuam bem em todos os grafos. Enquanto que a *RemoverParesRuins* performa bem apenas nas instâncias em que $|A|$ é maior que $|B|$, também é possível notar que *InserirPares* tem um desempenho melhor quando há diferença na quantidade de instruções entre os códigos.

Tabela 1: Resultados obtidos para a instância I_9

A	B	$ A $	$ B $	FN	Programação Dinâmica		Multistart-VND					
					Rt	$t(ms)$	Rt_m	$t_m(ms)$	BL_{tv}	BL_{rpr}	BL_{tp}	BL_{ip}
1	1	10	10	4000	1,000	215,59	1,000	0,02	0,821	0,057	0,713	0,050
1	2	10	10	4000	1,000	221,37	1,000	0,02	0,820	0,057	0,712	0,050
1	3	10	10	4000	1,000	221,14	1,000	0,01	0,821	0,057	0,713	0,050
1	4	10	10	4000	0,869	217,10	0,869	0,01	0,821	0,060	0,707	0,070
1	5	10	12	4000	0,733	1.056,03	0,733	0,11	0,878	0,152	0,657	0,437
1	6	10	12	4000	0,667	1.012,01	0,667	0,16	0,890	0,329	0,691	0,545
1	7	10	12	4000	0,667	1.028,17	0,667	0,10	0,887	0,346	0,670	0,525
1	8	10	15	4000	0,607	6.815,18	0,607	0,13	0,883	0,255	0,570	0,989
2	1	10	10	4000	1,000	216,96	1,000	0,02	0,821	0,057	0,713	0,050
2	2	10	10	4000	1,000	221,10	1,000	0,01	0,820	0,057	0,712	0,050
2	3	10	10	4000	1,000	216,59	1,000	0,02	0,821	0,057	0,713	0,050
2	4	10	10	4000	0,869	218,26	0,869	0,02	0,821	0,061	0,708	0,070
2	5	10	12	4000	0,733	1.021,45	0,733	0,17	0,878	0,153	0,657	0,437
2	6	10	12	4000	0,667	1.025,96	0,667	0,16	0,890	0,330	0,692	0,545
2	7	10	12	4000	0,667	1.023,53	0,667	0,05	0,887	0,344	0,669	0,526
2	8	10	15	4000	0,607	6.729,33	0,607	0,17	0,883	0,256	0,570	0,989
3	1	10	10	4000	1,000	216,91	1,000	0,01	0,821	0,057	0,713	0,050
3	2	10	10	4000	1,000	218,30	1,000	0,01	0,821	0,057	0,713	0,050
3	3	10	10	4000	1,000	227,45	1,000	0,03	0,820	0,057	0,713	0,050
3	4	10	10	4000	0,869	216,49	0,869	0,01	0,821	0,061	0,708	0,070
3	5	10	12	4000	0,733	1.015,83	0,733	0,17	0,878	0,152	0,657	0,438

Continua na próxima página

Tabela 1: Resultados obtidos para a instância J (continuação)

A	B	A	B	FN	Programação Dinâmica		Multistart					
					Rt	t(ms)	Rt _m	t _m (ms)	BL _{tv}	BL _{rpr}	BL _{tp}	BL _{ip}
3	6	10	12	4000	0,667	1.024,03	0,667	0,17	0,890	0,330	0,691	0,544
3	7	10	12	4000	0,667	1.025,96	0,667	0,10	0,887	0,345	0,670	0,525
3	8	10	15	4000	0,607	6.747,03	0,607	0,09	0,883	0,255	0,571	0,989
4	1	10	10	4000	0,869	217,00	0,869	0,02	0,820	0,061	0,708	0,070
4	2	10	10	4000	0,869	216,98	0,869	0,02	0,821	0,060	0,708	0,070
4	3	10	10	4000	0,869	217,51	0,869	0,01	0,820	0,060	0,708	0,070
4	4	10	10	4000	1,000	218,75	1,000	0,02	0,821	0,057	0,713	0,050
4	5	10	12	4000	0,864	1.038,45	0,864	0,22	0,879	0,154	0,670	0,153
4	6	10	12	4000	0,788	1.013,32	0,788	0,29	0,878	0,158	0,664	0,297
4	7	10	12	4000	0,788	1.013,49	0,788	0,06	0,878	0,160	0,651	0,299
4	8	10	15	4000	0,728	6.793,66	0,728	0,12	0,883	0,095	0,635	0,751
5	1	12	10	4000	0,823	1.011,91	0,823	1,77	0,948	0,971	0,751	0,964
5	2	12	10	4000	0,823	1.011,92	0,823	1,64	0,948	0,971	0,752	0,964
5	3	12	10	4000	0,823	1.004,60	0,823	1,59	0,948	0,971	0,752	0,964
5	4	12	10	4000	0,954	1.014,42	0,954	1,11	0,951	0,972	0,761	0,965
5	5	12	12	4800	1,000	6.201,62	1,000	0,03	0,883	0,098	0,797	0,088
5	6	12	12	4800	0,937	6.077,12	0,937	0,02	0,884	0,101	0,795	0,109
5	7	12	12	4800	0,906	6.060,65	0,906	0,04	0,883	0,100	0,829	0,106
5	8	12	15	4800	0,873	52.246,71	0,873	0,62	0,917	0,190	0,728	0,270
6	1	12	10	4000	0,757	1.100,46	0,757	2,57	0,948	0,978	0,714	0,980
6	2	12	10	4000	0,757	1.022,09	0,757	1,38	0,948	0,978	0,713	0,980
6	3	12	10	4000	0,757	1.024,25	0,757	1,93	0,948	0,978	0,713	0,980
6	4	12	10	4000	0,878	1.036,45	0,878	1,62	0,951	0,971	0,759	0,966
6	5	12	12	4800	0,937	6.035,34	0,937	0,02	0,884	0,100	0,795	0,107
6	6	12	12	4800	1,000	5.945,65	1,000	0,02	0,883	0,098	0,797	0,089
6	7	12	12	4800	0,969	5.991,45	0,969	0,03	0,884	0,098	0,833	0,088
6	8	12	15	4800	0,935	52.173,25	0,935	0,45	0,917	0,185	0,735	0,189
7	1	12	10	4000	0,742	1.038,74	0,742	0,33	0,950	0,979	0,721	0,981
7	2	12	10	4000	0,742	999,16	0,742	0,43	0,950	0,979	0,722	0,981
7	3	12	10	4000	0,742	985,99	0,742	0,55	0,950	0,979	0,722	0,981
7	4	12	10	4000	0,863	1.023,12	0,863	0,59	0,952	0,973	0,767	0,967
7	5	12	12	4800	0,906	6.066,57	0,906	0,06	0,883	0,098	0,831	0,104
7	6	12	12	4800	0,969	5.869,33	0,969	0,02	0,884	0,097	0,834	0,087
7	7	12	12	4800	1,000	6.069,73	1,000	0,03	0,884	0,099	0,799	0,089
7	8	12	15	4800	0,950	52.702,54	0,950	0,52	0,917	0,184	0,703	0,190
8	1	15	10	4000	0,712	7.056,54	0,712	2,66	0,959	0,971	0,711	0,981
8	2	15	10	4000	0,712	7.034,15	0,712	4,14	0,959	0,971	0,712	0,981
8	3	15	10	4000	0,712	7.087,03	0,712	5,71	0,959	0,971	0,712	0,981
8	4	15	10	4000	0,833	7.046,65	0,833	5,05	0,960	0,970	0,736	0,972
8	5	15	12	4800	0,894	55.190,89	0,894	0,41	0,971	0,916	0,859	0,937
8	6	15	12	4800	0,956	54.311,85	0,956	0,46	0,972	0,903	0,865	0,925
8	7	15	12	4800	0,948	54.531,80	0,948	0,30	0,972	0,887	0,857	0,912
8	8	15	15	6000	1,000	740.800,36	1,000	0,13	0,938	0,169	0,873	0,156

Na Figura 2, são apresentados os *Heatmaps* que contêm os valores médios de similaridade entre todos os códigos para cada uma das instâncias restantes. Observa-se que o mesmo padrão se repete em todas as instâncias. Os códigos 1, 2 e 3 são considerados equivalentes devido ao pré-processamento efetuado pelo modelo. As modificações realizadas nas demais versões afetam a similaridade, porém a intensidade dessa variação depende de cada instância. Outro aspecto importante a se destacar é a diferença consistente na similaridade calculada de *A* para *B* e de *B* para *A*, sendo necessário um aprofundamento sobre a causa dessas variações.

5. Conclusão

Este trabalho apresenta uma extensão para a linguagem C do modelo de detecção de plágio e similaridade de códigos proposto por Chaves et al. [2023], baseado na resolução do problema do menor caminho com premiação máxima. São propostas técnicas de pré-processamento, uma solução baseada em Programação Dinâmica e outra baseada na meta-heurística Multi-Start. Adicionalmente, foi construído um *benchmark* para validação dos métodos propostos.

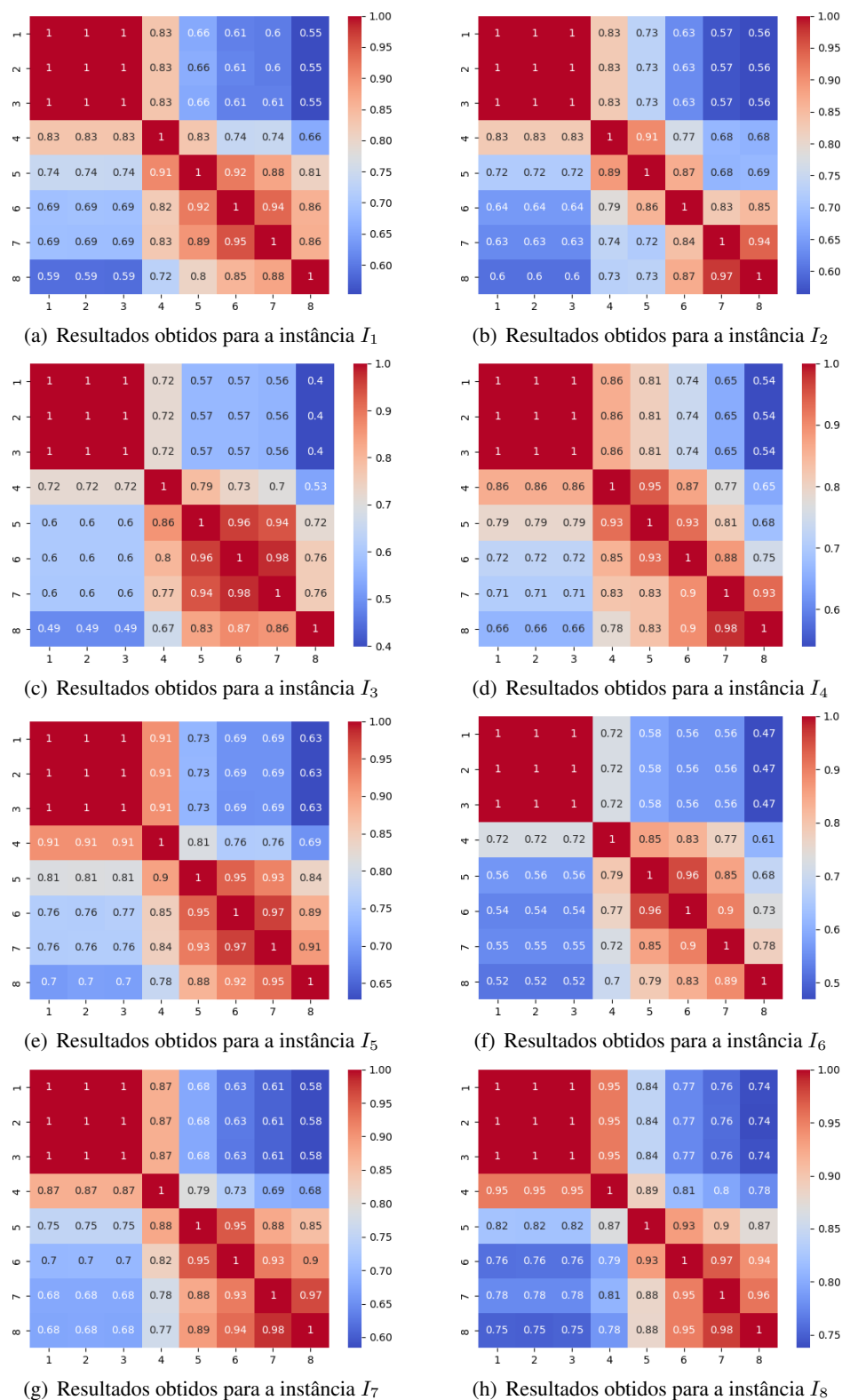


Figura 2: Heatmaps da similaridade média entre os códigos de cada instância

A partir dos experimentos realizados foi possível demonstrar que o modelo proposto é capaz de representar similaridade entre dois códigos escritos na linguagem C de forma consistente, abrindo espaço para futuros trabalhos a respeito da verificação de plágio em códigos reais e a comparação com ferramentas já consolidadas. Além disto, o algoritmo de Programação Dinâmica apresentado neste trabalho foi capaz de resolver o PMCPM em instâncias pequenas, porém, para a resolução exata de instâncias maiores, em trabalhos futuros, pretende-se utilizar formulações matemáticas e Programação Inteira [Maculan e Fampa, 2006], que comumente apresentam melhor performance que a Programação Dinâmica em problemas semelhantes. Por sua vez, o *Multistart-VND* conseguiu atingir consistentemente, e em tempo satisfatório, todos os ótimos globais conhecidos, permitindo futuras comparações com outras meta-heurísticas. As buscas locais propostas funcionaram bem e apresentaram algumas características de desempenho próprias podendo ser mais exploradas em trabalhos futuros.

Referências

- Agrawal, M. e Sharma, D. K. (2016). A state of art on source code plagiarism detection. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, p. 236–241. IEEE.
- Chaves, R. C., Albuquerque, K. M., Chaves, R. C., e Gouveia, T. (2023). Modelo para detecção de plágio em exercícios de programação na linguagem python. In *Anais do VIII Encontro de Teoria da Computação*, p. 196–200. SBC.
- Cheers, H., Lin, Y., e Smith, S. P. (2021). Academic source code plagiarism detection by measuring program behavioral similarity. *IEEE Access*, 9:50391–50412.
- Cui, Y., Tao, L., Bao, D., e Hai, R. (2023). Analysis and research on gamification teaching strategy used in c language programming course. In *2023 4th International Conference on Education, Knowledge and Information Management (ICEKIM 2023)*, p. 172–180. Atlantis Press.
- Gouveia, T., Albuquerque, K., Oliveira, J., e Maciel, V. M. (2023). C073: ferramenta para apoio ao ensino de programação usando a metodologia de aprendizagem baseada em problemas. *Revista Principia - Divulgação Científica e Tecnológica do IFPB*, 60(1):70–87. ISSN 2447-9187. URL <https://periodicos.ifpb.edu.br/index.php/principia/article/view/5942>.
- Horváth, M. e Pietriková, E. (2024). An experimental comparison of three code similarity tools on over 1,000 student projects. In *2024 IEEE 22nd World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, p. 000423–000428. IEEE.
- Maculan, N. e Fampa, M. H. C. (2006). Otimização linear. *Editora Universidade de Brasília: Brasília*.
- Rong, W., Xu, T., Sun, Z., Sun, Z., Ouyang, Y., e Xiong, Z. (2023). An object tuple model for understanding pointer and array in c language. *IEEE Transactions on Education*.
- Wen, W., Xue, X., Li, Y., Gu, P., e Xu, J. (2019). Code similarity detection using ast and textual information. *International Journal of Performability Engineering*, 15(10):2683.
- Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., e Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software*, p. 111796.