INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
CAMPUS CAJAZEIRAS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

MÉTODOS DE OTIMIZAÇÃO APLICADOS À DETECÇÃO DE PLÁGIO EM EXERCÍCIOS DE PROGRAMAÇÃO NA LINGUAGEM C

JOÃO PEDRO OLIVEIRA SANTOS

Cajazeiras
24 de setembro de 2024

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA

JOÃO PEDRO OLIVEIRA SANTOS

MÉTODOS DE OTIMIZAÇÃO APLICADOS À DETECÇÃO DE PLÁGIO EM EXERCÍCIOS DE PROGRAMAÇÃO NA LINGUAGEM C

Trabalho de Conclusão de Curso apresentado junto ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - Campus Cajazeiras, como requisito à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador

Prof. Esp. João Igor Barros Rocha

Aprovada em: **13 de fevereiro de 2026.**

Prof. Esp. João Igor Barros Rocha - Orientador

Prof. Me. Diogo Dantas Moreira - Avaliador

IFPB - Campus Cajazeiras

Prof. Me. Francisco Paulo de Freitas Neto - Avaliador

IFPB - Campus Cajazeiras

Documento assinado eletronicamente por:

- **Fabio Abrantes Diniz**, **COORDENADOR(A) DE CURSOS - FUC1 - UNINFO-CZ**, em 13/02/2026 12:08:30.
- **Francisco Paulo de Freitas Neto**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 13/02/2026 13:10:57.
- **Joao Igor Barros Rocha**, **PROF ENS BAS TEC TECNOLOGICO-SUBSTITUTO**, em 13/02/2026 13:32:46.
- **Diogo Dantas Moreira**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 13/02/2026 14:19:58.

Este documento foi emitido pelo SUAP em 13/02/2026. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/autenticar-documento/ e forneça os dados abaixo:

Código 835828
Verificador: 62037a724e
Código de Autenticação:

IFPB / Campus Cajazeiras
Coordenação de Biblioteca
Biblioteca Prof. Ribamar da Silva
Catalogação na fonte: Cícero Luciano Félix CRB-15/750

S237m Santos, João Pedro Oliveira.

Métodos de otimização aplicados à detecção de plágio em exercícios de programação na linguagem C / João Pedro Oliveira Santos.– 2026.

44f. : il.

Trabalho de Conclusão de Curso (Tecnólogo em Análise e Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia da Paraíba, Cajazeiras, 2026.

Orientador(a): Prof. Esp. João Igor Barros Rocha.

1. Desenvolvimento de sistemas. 2. Programação. 3. Plágio - Detecção. 4. Linguagem C. I. Instituto Federal de Educação, Ciência e Tecnologia da Paraíba. II. Título.

IFPB/CZ CDU: 004.4(043.2)

À família, aos amigos e, em especial, à minha avó. Aos professores, orientadores e colegas de olimpíadas e maratonas de programação.

AGRADECIMENTOS

Agradeço a todos aqueles presentes nessa caminhada.

”Colorless green ideas sleep furiously”

Noam Chomsky

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.

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.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

ADS	Análise e Desenvolvimento de Sistemas
IFPB	Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
NBR	Norma Brasileira
TCC	Trabalho de Conclusão do Curso
PMCPM	Problema do Menor Caminho com Premiação Máxima
VND	Variable, Neighbourhood Descent

SUMÁRIO

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 (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: JPlag[1], Sherlock[2] e MOSS[3] (HORVÁTH; 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

[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/

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: O Capítulo 2 apresenta o referencial teórico, onde será explicado alguns conceitos importantes para o entendimento da ideia e uma discussão acerca de trabalhos sobre detecção de plágio. O Capítulo 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. O Capítulo 4 discute a estratégia para geração de instâncias, descreve os experimentos realizados e os resultados obtidos. Por fim, o Capítulo 5 traz as considerações finais, assim como sugestões para continuidade da pesquisa.

2 REFERENCIAL TEÓRICO

Neste capítulo, apresentaremos conceitos e definições fundamentais para o entendimento deste trabalho como um todo. Muitos dos tópicos abordados nas seções subsequentes dependem de um conhecimento prévio desses conceitos.

2.1 GRAFOS

Grafos são estruturas matemáticas usadas para modelar relacionamentos entre objetos. Eles consistem em dois componentes principais: os vértices, que representam os objetos ou entidades, e as arestas, que representam as conexões ou relações entre os vértices. Uma definição matemática seria: Um grafo G é definido como um par $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, que são pares de vértices. Abaixo seguem algumas definições importantes para o melhor entendimento do assunto.

Definição 2.1.1 (Orientação)**.** *Orientação ou direcionamento refere-se à direção das arestas. Em grafos não direcionados, a orientação das arestas não importa, uma vez que toda conexão entre quaisquer dois vértices é bidirecional. Em grafos direcionados, a orientação das arestas (ou arcos, ver Definição 2.1.2) indica como os vértices estão relacionados entre si, permitindo modelar cenários em que as relações possuem um fluxo ou sequência específica.*

Definição 2.1.2 (Arcos)**.** *Os arcos são elementos fundamentais em grafos direcionados, representando arestas que têm uma direção específica entre dois vértices. Em um grafo direcionado, cada arco indica uma relação unidirecional, significando que você pode ir de um vértice a outro em uma única direção, mas não necessariamente na direção oposta.*

Definição 2.1.3 (Grafos Bipartidos)**.** *Grafos bipartidos são um tipo especial de grafo em que os vértices podem ser divididos em dois conjuntos disjuntos, de forma que nenhuma aresta conecta dois vértices do mesmo conjunto. Ou seja, todas as arestas do grafo ligam vértices de um conjunto ao outro. Formalmente, pode-se observar que: Um grafo $G = (V, E)$ é bipartido se o conjunto de vértices V pode ser dividido em dois subconjuntos U e W, de modo que: $U \cup W = V$ (todos os vértices pertencem a um dos dois conjuntos), $U \cap W = \emptyset$ (os conjuntos são disjuntos, ou seja, não têm vértices em comum), e todas as arestas conectam um vértice de U a um vértice de W.*

Definição 2.1.4 (Grafos Completos)**.** *São grafos onde todos os pares de vértices distintos estão conectados diretamente entre si por uma aresta única. Um grafo completo com n vértices é definido como K_n.*

Definição 2.1.5 (Grafos Bipartidos Completos)**.** *São um tipo específico de grafo bipartido (Definição 2.1.3), onde há uma conexão entre todos os vértices de um conjunto e todos os vértices do outro conjunto. Um grafo bipartido completo com conjuntos de vértices V_1 e V_2 de tamanho $|V_1| = m$ e $|V_2| = n$, é definido como $K_{m,n}$, onde K indica que é um grafo bipartido completo e m e n indicam o número de vértices em cada conjunto.*

2.2 PROBLEMAS DE PROGRAMAÇÃO

Problemas de programação referem-se a desafios que envolvem a escrita de código para resolver questões específicas ou realizar tarefas. Esses problemas podem abranger uma ampla gama de tópicos e níveis de dificuldade, e são frequentemente utilizados para avaliar habilidades de programação, lógica e análise de algoritmos.

Tais problemas podem ser divididos em 4 componentes: A descrição do problema, que é um enunciado que define o que precisa ser resolvido, frequentemente incluindo os exemplos e as restrições. Entradas e Saídas, que são especificações sobre os dados de entrada que o algoritmo deve processar e as saídas que ele deve gerar. Restrições, que são limitações no tamanho dos dados, tempo de execução, ou recursos disponíveis que a solução deve respeitar. E testes (para *Online Judges*, Seção 2.3), que são casos de teste que a solução deve passar para ser considerada correta.

Os contextos de aplicação dos problemas de programação são diversos, estando presentes em alguns meios como: ambientes acadêmicos e educacionais, onde problemas de programação são usados para ensinar e avaliar a compreensão de algoritmos, estruturas de dados e técnicas de resolução de problemas. Entrevistas técnicas em processos seletivos para posições de desenvolvimento de software, nas quais os problemas de programação são frequentemente usados para avaliar as habilidades de codificação e resolução de problemas dos candidatos. E por fim, em ambientes de competição e treinamento, como as plataformas Codeforces e LeetCode, onde o uso de problemas de programação é essencial para competir contra outros programadores e aprimorar habilidades.

2.3 ONLINE JUDGES

Online Judges são plataformas que fornecem um *sistema de avaliação online* para problemas que podem ser resolvidos por meio de programação, geralmente de forma independente da linguagem utilizada. Essas plataformas fazem uso de julgamento automático de código, ou seja, possuem ferramentas que executam e testam o código para verificar se a saída está alinhada com os casos de teste definidos para cada problema.

Alguns *Online Judges* organizam competições (*contests*, como são mais conhecidos), que visam fazer com que competidores, geralmente de todo o mundo, mostrem e treinem suas habilidades de resolução de problemas (*problem solving*[1]). Plataformas que organizam tais *contests* geralmente oferecem premiações para os melhores colocados, desde premiações em produtos/aparelhos tecnológicos, até premiações em dinheiro, viagens, etc. Mas são poucos *Online Judges* que realmente os realizam de forma mais frequente e/ou com excelência.

Tais plataformas são importantes para o ensino de programação pois fornecem diversos *problemas* que treinam e desenvolvem diversas habilidades de resolução de problemas. E, se contabilizarmos também os *Online Judges* que contêm *contests*, essas habilidades são bem melhores desenvolvidas, além do treino sob pressão do tempo curto da competição, que forçam o competidor a lidar melhor com a mesma.

Dentre os principais *Online Judges*, destacam-se:

Beecrowd: É uma plataforma[2] brasileira que dispõe de diversos *problemas*[3] de diferentes categorias. Indo desde problemas com foco no público mais iniciante, até aqueles mais desafiadores, além de contar com uma categoria apenas para problemas de SQL. A plataforma também suporta soluções de mais de 20 linguagens de programação diferentes. Atualmente a plataforma aparenta estar almejando uma maior proximidade com as empresas da área de tecnologia, disponibilizando funcionalidades que visam criar uma conexão das mesmas com os usuários da plataforma.

LeetCode: É uma plataforma[4] que foca em *problemas*[5] frequentemente encontrados em entrevistas técnicas e é uma ferramenta para praticar e aprimorar habilidades de resolução de problemas, sendo amplamente utilizada por usuários do mundo inteiro, e bem vista pelas grandes empresas.

Codeforces: É uma plataforma[6] russa, que dispõe de inúmeros *problemas*[7] de diversas origens diferentes, muitos vindo dos seus *contests* [8], que são organizados com frequência na plataforma, que por sua vez, é a mais utilizada no contexto de programação competitiva. O codeforces também contêm um blog[9], onde muitos usuários podem compartilhar ideias e comentários.

---

1 *Problem solving* é o processo de resolver problemas de maneira eficiente e otimizada usando algoritmos e estruturas de dados.
2 https://judge.beecrowd.com/pt
3 https://judge.beecrowd.com/pt/categories
4 https://leetcode.com/
5 https://leetcode.com/problemset/
6 https://codeforces.com
7 https://codeforces.com/problemset
8 https://codeforces.com/contests
9 https://codeforces.com/top

2.4 PROGRAMAÇÃO DINÂMICA

Programação dinâmica é uma técnica de otimização usada em algoritmos para resolver problemas mais complexos, dividindo-os em subproblemas menores e reutilizando os resultados desses subproblemas para evitar computações repetidas. Essa técnica é aplicada a problemas que podem ser resolvidos de forma recursiva e possuem subestruturas otimizáveis e sobreposição de subproblemas, como veremos mais adiante.

A ideia central da programação dinâmica é evitar o cálculo repetido das mesmas subsoluções, armazenando-as para que possam ser reutilizadas posteriormente. E como já foi dito, isso é feito dividindo o problema em subproblemas menores, resolvendo cada subproblema uma vez e armazenando seus resultados em estruturas como *arrays* ou *matrizes*. Com isso, o algoritmo se torna mais eficiente, pois evita o trabalho redundante. Sendo assim, para que um problema possa ser resolvido utilizando programação dinâmica, é necessário que ele tenha essa capacidade de ser decomposto em subproblemas menores, e que esses subproblemas se repitam diversas vezes ao longo do processo de solução, junto com um ou mais *casos base* (Ver Definição 2.4.1). Além disso, o problema também precisa apresentar uma recorrência ou subestrutura ótima, o que indica que a solução ótima de um problema (maior) pode ser composta pelas soluções ótimas de seus subproblemas.

Definição 2.4.1 (Casos base)**.** *Casos base são componentes essenciais em algoritmos recursivos e, em particular, na programação dinâmica. Eles definem as condições de término de uma recursão, ou seja, os casos mais simples do problema, para os quais a solução é conhecida diretamente e não exige a decomposição em subproblemas. Esses casos servem como "ponto de parada" da recursão, evitando que o algoritmo continue indefinidamente.*

Problemas que são resolvíveis utilizando programação dinâmica geralmente podem ser resolvidos com qualquer uma das seguintes abordagens:

Top-down: Nessa abordagem o problema é resolvido de cima para baixo, começando pela versão maior do problema e dividindo-o recursivamente em subproblemas menores conforme necessário. À medida que cada subproblema é resolvido, seus resultados são armazenados (memoizados) para evitar a recomputação futura. Essa abordagem é frequentemente implementada de forma recursiva.

Bottom-up: Nessa abordagem o problema é resolvido de baixo para cima, começando pelos subproblemas menores e utilizando seus resultados para construir soluções para subproblemas maiores até chegar à solução do problema original. Essa abordagem é frequentemente implementada de forma iterativa.

2.5 ALGORITMOS DE OTIMIZAÇÃO

São métodos ou procedimentos utilizados para encontrar a melhor solução para um determinado problema, normalmente maximizando ou minimizando alguma função objetivo. Tais algoritmos podem ser categorizados em 3 tipos: Algoritmos Exatos (Subseção 2.5.1), Algoritmos Heurísticos (Subseção 2.5.2), e Algoritmos Metaheurísticos (Subseção 2.5.3).

Tais algoritmos otimização tem grande importância de modo geral, e podem ser aplicados em várias situações, como: logística e transporte, onde o objetivo é encontrar a rota mais eficiente, como no problema do caixeiro viajante (POP et al., 2024), alocação de recursos, em que o objetivo é otimizar a distribuição de recursos em uma empresa, minimizando custos e maximizando lucros, redes de telecomunicações, onde almeja-se a otimização do uso da largura de banda e melhorar a qualidade da comunicação, e machine learning, onde o objetivo é treinar modelos ajustando parâmetros para minimizar erros e maximizar a precisão.

Geralmente, problemas de otimização podem ser divididos em:

1. Função objetivo, que é a função que queremos otimizar, seja maximizando ou minimizando, por exemplo, maximizar lucro ou minimizar custo.
2. Variáveis de decisão, que são os elementos do problema que podem ser ajustados ou alterados para otimizar a função objetivo.
3. Restrições, que são condições ou limites que as variáveis de decisão devem obedecer, tais como limites de recursos, regras de operação ou requisitos do problema.

Os algoritmos de otimização, geralmente podem ter os seguintes conceitos atrelados aos mesmos, os quais irão aparecer com frequência neste trabalho:

Definição 2.5.1 (Espaço de Soluções)**.** *O "espaço de soluções" é o conjunto de todas as possíveis soluções para um determinado problema. Cada solução dentro desse espaço representa uma configuração que atende às condições do problema.*

Definição 2.5.2 (Vizinhança Local)**.** *São um conjunto de soluções próximas de uma solução atual, tentando melhorar iterativamente até que não seja possível encontrar soluções melhores nas proximidades imediatas.*

Definição 2.5.3 (Ótimo Local)**.** *Um ótimo local é alcançado quando, dentro da vizinhança de uma solução, não há solução melhor, mesmo que ela não seja a melhor solução global (ótimo global) do problema.*

Definição 2.5.4 (Diversificação e Intensificação)**.** *Diversificação refere-se a estratégias que aumentam a exploração do espaço de busca, forçando o algoritmo a investigar regiões amplamente diferentes. Intensificação é o oposto, focando a busca em uma região promissora com a intenção de melhorar soluções já boas. Em muitas metaheurísticas (Subseção 2.5.3), há um equilíbrio dinâmico entre esses dois processos.*

Definição 2.5.5 (Escapismo)**.** *Técnicas de escapismo são estratégias usadas para evitar ficar preso em ótimos locais. Alguns algoritmos de otimização possuem mecanismos embutidos para evitar ótimos locais, como aceitação probabilística de soluções piores ou saltos aleatórios no espaço de busca.*

Definição 2.5.6 (Perturbação)**.** *A perturbação é uma técnica usada para "sacudir" a solução atual, na esperança de levar o algoritmo a escapar de uma vizinhança local em busca de uma solução melhor.*

2.5.1 Algoritmos Exatos

São algoritmos que garantem encontrar a melhor solução possível para o problema, mas geralmente funcionam bem apenas em problemas menores ou de média complexidade. Exemplos: Algoritmo de Dijkstra, Algoritmo de Kruskal, Algoritmo de Bellman-Ford, etc.

2.5.2 Algoritmos Heurísticos

São algoritmos que utilizam abordagens aproximadas para encontrar soluções aceitáveis ou boas o suficiente, mas não necessariamente as melhores ou ótimas. Elas são amplamente utilizadas em situações onde é difícil ou impraticável encontrar a solução exata, seja devido à complexidade do problema, à falta de tempo ou à quantidade de recursos disponíveis. Existem alguns tipos de heurísticas, explicaremos dois dos mais comuns logo mais nas Subsubseções 2.5.2.1 e 2.5.2.2. Um dos exemplos mais comuns e o que, de longe, terá mais relevância para este trabalho, é a Busca Local, que será abordada na Subsubseção 2.5.2.3.

2.5.2.1 Heurísticas Construtivas

Conforme apresentado em (SOUZA, 2008), heurísticas construtivas são métodos que visam gerar uma solução, passo a passo, começando do zero. A escolha de cada elemento a ser adicionado em cada etapa depende da função de avaliação utilizada, que, por sua vez, está diretamente relacionada ao problema em questão. Nas heurísticas mais clássicas, os elementos candidatos costumam ser classificados de acordo com uma função gulosa, que avalia o benefício de inserir cada elemento. A cada iteração, apenas o "melhor" elemento, de acordo com essa avaliação, é selecionado e adicionado à solução.

2.5.2.2 Heurísticas de Refinamento

Heurísticas de refinamento são técnicas que partem de uma solução inicial, muitas vezes obtida de forma rápida e simples, e buscam melhorá-la gradativamente. O objetivo é ajustar ou modificar essa solução para encontrar uma versão mais eficiente ou próxima do ideal. A abordagem envolve a exploração do espaço de soluções, realizando pequenas alterações (ou refinamentos) na solução existente, como trocas de elementos, mudanças de posição ou ajustes em parâmetros. Em cada passo, avalia-se se a nova solução é melhor que a anterior, e, se for, ela é adotada como a nova base para continuar o processo.

2.5.2.3 Busca Local

Busca Local é uma heurística de refinamento que, em vez de explorar todo o espaço de soluções, se concentra em uma solução inicial e busca melhorias incrementais a partir dela. O processo envolve mover-se para vizinhos da solução atual (Definição 2.5.2), que são soluções que podem ser obtidas a partir de pequenas alterações. O principal objetivo da busca local é encontrar uma solução que seja "melhor" do que a atual, de acordo com um critério específico. No entanto, uma das limitações dessa abordagem é que ela pode ficar presa em ótimos locais.

2.5.3 Metaheurísticas

São algoritmos que combinam várias heurísticas para obter soluções mais robustas e eficientes para diferentes problemas, que, essencialmente, são custosos para encontrar uma solução ótima global. Exemplos: Algoritmo Genético, Algoritmo de Colônia de Formigas, Simulated Annealing, entre outros (ALORF, 2023). Cada metaheurística tem suas próprias características e mecanismos de funcionamento, mas todas compartilham o objetivo de melhorar gradualmente uma solução inicial por meio de processos iterativos, combinando exploração e aproveitamento de soluções já conhecidas. O uso de metaheurísticas é especialmente valioso em problemas onde a função objetivo é complexa, o espaço de busca é muito grande ou a solução exata demandaria um tempo computacional inviável. Dessa forma, tornaram-se uma ferramenta essencial em contextos que exigem soluções próximas da ótima global em tempo hábil.

As metaheurísticas que seguem abaixo serão de grande importância para o entendimento deste trabalho:

2.5.3.1 Multistart

É uma metaheurística utilizada para resolver problemas de otimização, em que a ideia central é executar várias vezes um procedimento de busca local, começando de diferentes soluções iniciais. Isso ajuda a explorar melhor o espaço de busca, evitando que o algoritmo fique preso em ótimos locais.

De maneira geral, o funcionamento do Multistart pode ser descrito a partir das seguintes etapas: Geração de soluções iniciais, em que várias soluções iniciais são geradas aleatoriamente ou de forma heurística, onde cada uma dessas soluções servirá como ponto de partida para uma busca local. A segunda etapa é a da Busca Local, em que para cada solução inicial, é aplicada uma técnica de busca local (ou seja, um algoritmo que tenta melhorar a solução iterativamente). Tal busca melhora a solução, movendo-se para vizinhanças (Definição 2.5.2) da solução atual até não ser mais possível encontrar melhorias. A terceira etapa é a repetição, na qual esse processo de gerar uma nova solução inicial e aplicar a busca local é repetido várias vezes, cada vez com uma solução inicial diferente. E a última etapa é a de selecionar a melhor solução dentre todas as obtidas ao longo das várias execuções.

A grande vantagem do Multistart é que, ao começar de diferentes pontos no espaço de busca, aumenta a chance de encontrar uma solução ótima global (a melhor possível), já que uma única execução de busca local pode ficar presa em um ótimo local (Definição 2.5.3). Assim, ao explorar várias partes diferentes do espaço de busca, o algoritmo pode superar essas limitações. O Multistart é simples de implementar e funciona bem em muitos problemas de otimização complexos. Ele pode ser usado sozinho ou como parte de uma estratégia maior, combinando-se com outras metaheurísticas.

2.5.3.2 VND

Busca de Vizinhança Variável (do inglês, *Variable Neighborhood Descent*), mais conhecido como *VND*, é um método metaheurístico, que faz parte da família de métodos de busca local (Subsubseção 2.5.2.3). Nesse método, a ideia principal é explorar diferentes "vizinhanças" (Definição 2.5.2) de uma solução inicial para encontrar melhorias. O VND começa com uma solução inicial e, em vez de se limitar a explorar apenas uma vizinhança (como é comum em buscas locais tradicionais), ele muda sistematicamente de uma vizinhança para outra, buscando superar limitações de métodos que ficam presos em *mínimos locais*. Se, ao explorar uma *vizinhança*, uma melhoria é encontrada, o algoritmo faz o movimento para a nova solução e retorna à primeira *vizinhança* para continuar a busca a partir daí. Caso contrário, ele passa para a próxima *vizinhança*. Isso aumenta as chances de encontrar soluções de melhor qualidade, pois a mudança entre diferentes vizinhanças amplia o espaço de busca e diminui a chance de ficar preso em ótimos locais (Definição 2.5.3).

2.6 PROBLEMA DO MENOR CAMINHO COM PREMIAÇÃO MÁXIMA

O problema do menor caminho com premiação máxima (PMCPM) é uma variação do problema clássico do menor caminho, onde o objetivo não é apenas encontrar o caminho mais curto ou eficiente, mas também maximizar algum tipo de "recompensa" ou "premiação" coletada ao longo do caminho.

Suponha que você tenha um conjunto de pontos ou vértices em um grafo, e cada ponto tenha uma certa premiação associada. O desafio é encontrar um caminho que satisfaça duas condições principais: a primeira é minimizar o custo do caminho, e isso pode ser a distância entre os vértices, o tempo necessário para viajar de um ponto a outro ou qualquer outra medida de custo. A segunda é maximizar a premiação coletada, onde ao longo do caminho escolhido onde você coleta "recompensas" ou "premiações" associadas a determinados pontos do grafo. Em outras palavras, a ideia é balancear a premiação máxima que você pode coletar com a restrição de que o caminho deve ser o mais eficiente possível em termos de custo (por exemplo, tempo ou distância). Isso traz uma complexidade extra, pois você não está apenas focado no menor caminho possível, mas também precisa levar em consideração os ganhos obtidos ao visitar certos nós.

Esse problema pode ser encontrado em diversas áreas, como logística, turismo ou até mesmo em jogos de videogame, onde o objetivo é alcançar um destino enquanto acumula a maior quantidade de benefícios, respeitando uma limitação de custo.

2.7 TRABALHOS RELACIONADOS

Esta seção apresenta uma breve 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; 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 (Seção 2.6) 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 (Seção 2.4), e o segundo combinando a meta-heurística MultiStart (Subsubseção 2.5.3.1) com uma estratégia VND (Subsubseção 2.5.3.2).

3.1 MODELO DE (CHAVES et al., 2023)

Dados dois códigos-fonte $A = (a_1, a_2, ..., a_n)$, e $B = (b_1, b_2, ..., b_m)$, 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 (Ver Definições 2.1.5 e 2.1.1), formado pelos conjuntos de vértices V e W, e por um conjunto de arcos E (Ver Definição 2.1.2). 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 $w : E \to \mathbb{R}$ 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 $C = \{e_1, e_2, \cdots, e_s\} \subset E$, 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.

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

Por fim, resta a definição formal da função w. Seja $w(e) = \overrightarrow{w}(e)$, 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* $(t_1, t_2, \cdots, t_k)$, 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*.

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

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 $\overrightarrow{w}(e)$.

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

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.

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

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

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*[1], utilizado em (CHAVES et al., 2023), não oferece suporte à tokenização de códigos em C, foi escolhido o pacote *clang*[2], 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.

3.3 PROGRAMAÇÃO DINÂMICA PARA O PMCPM

Considerando o modelo apresentado na Seção 3.2, é possível resolver o PMCPM (Seção 2.6) 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.

[1] https://pypi.org/project/tokenizer/
[2] https://pypi.org/project/clang/

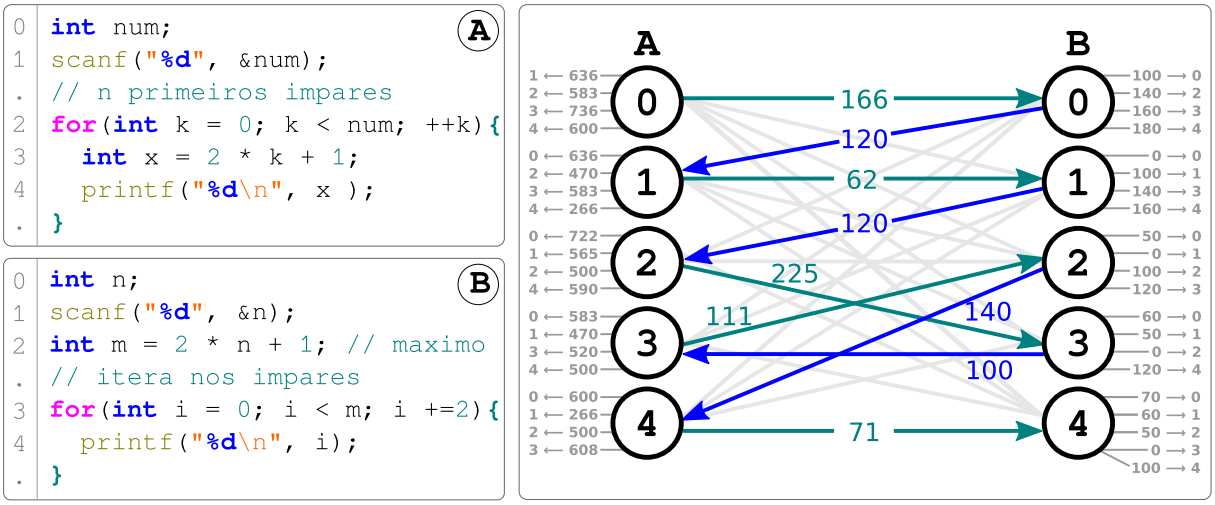


Figura 1 – Exemplo de extensão do modelo proposto para linguagem C

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

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 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) ≠ Nil então retorna
        ResultadoEstado(V', W', b) ;
4     best ← 0;
5     para todo a' ∈ V\V' faça
6         para todo b' ∈ W\W' faça
7             best ←
                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 MultiStart (Subsubseção 2.5.3.1) com uma estratégia de busca local VND (Subsubseção 2.5.3.2), 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 $(p_1, p_2, ...)$, 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(α, T, G = (V, W, E))
2      S* ← ∅;
3      enquanto t < T faça
4          S ← ConstruirSolucao(α, G);
5          S ← TrocarVertices(S, G);
6          S ← RemoverParesRuins(S, G);
7          S ← TrocarPares(S, G);
8          S ← InserirPares(S, G));
9          S* ← 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(α, G = (V, W, E))
2      S ← ∅,  V' ← V,  W' ← W;
3      enquanto (V' ≠ ∅) e (W' ≠ ∅) faça
4          a' ← Retire um vértice aleatório de V';
5          se Rand(0,1) < α então
6              b' ← Retire um vértice aleatório de W';
7              S ← insira p = (a', b') onde maximize FO(S);
8          senão
9              S' ← S ;
10             para todo b'' ∈ W' faça
11                 S'' ← insira p = (a', b'') onde maximize FO(S);
12                 se S'' é melhor que S' então S' ← S'',  b' ← b'' ;
13             Remova b' de W',  S ← 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 $I = \{1, 2, \ldots, |S| - 1\}$ 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 $I = \{1, 2, \ldots, |S| - 1\}$ 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 procedimento 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 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
6. Mudança na ordem entre instruções independentes
7. Alteração na estrutura do código, como adicionar funções, modificar estruturas de repetição e decisão, etc.

Após a aplicação sequencial das alterações listadas, obtém-se oito códigos no total (um original e sete novos). Pode-se encontrar exemplos desses códigos, junto com suas respectivas modificações no Apêndice A, Seção A.2. Em seguida, utilizamos o modelo proposto para gerar 64 grafos que representam a similaridade entre quaisquer duas versões do código (Conferir Apêndice A, Seção A.1). 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 para a criação das instâncias estão escritos na linguagem C, e as versões originais foram obtidas a partir de soluções de problemas presentes na lista 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 $I_1, I_2, \cdots, I_9$, 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.

$$FN = 2 \times P \times min(|A|, |B|) \tag{7}$$

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

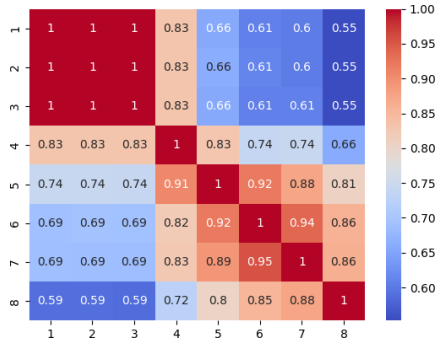
					Programação Dinâmica		**Multistart-VND**					
A	B	$\|A\|$	$\|B\|$	FN	Rt	$t(ms)$	Rt_m	$t_m(ms)$	BL_{tv}	BL_{rpr}	BL_{tp}	BL_{ip}
1	1	10	10	4000	1,000	215,59	1,000	0,02	0,821	0,057	0,713	0,050
1	2	10	10	4000	1,000	221,37	1,000	0,02	0,820	0,057	0,712	0,050
1	3	10	10	4000	1,000	221,14	1,000	0,01	0,821	0,057	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
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

Continua na próxima página

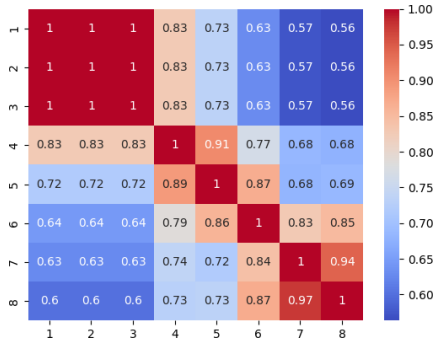
Tabela 1 – Resultados obtidos para a instância J (continuação)

					Programação Dinâmica		Multistart					
A	B	$\|A\|$	$\|B\|$	FN	Rt	$t(ms)$	Rt_m	$t_m(ms)$	BL_{tv}	BL_{rpr}	BL_{tp}	BL_{ip}
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

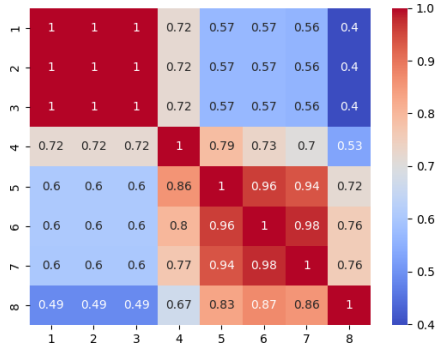
Na Figura 2, são apresentados os *Heatmaps* que contêm os valores médios de similaridade entre todos os códigos para cada uma das instâncias restantes. Observa-se que o mesmo padrão se repete em todas as instâncias. Os códigos 1, 2 e 3 são considerados equivalentes devido ao pré-processamento efetuado 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 varições.



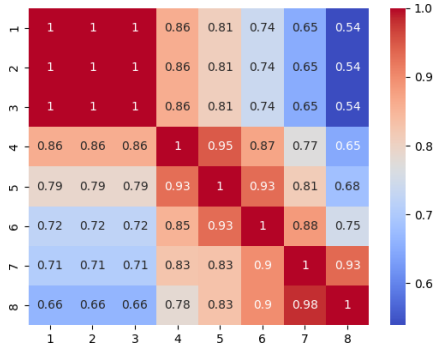
(a) Resultados obtidos para a instância I_1



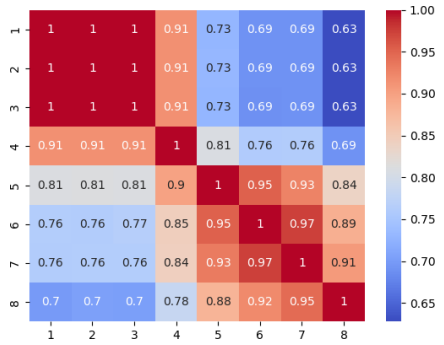
(b) Resultados obtidos para a instância I_2



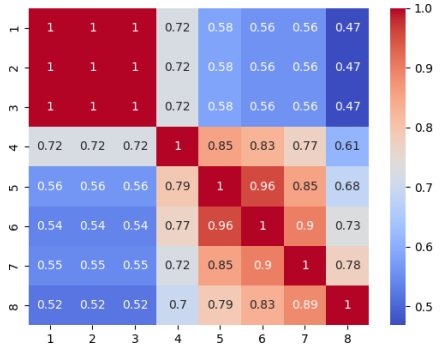
(c) Resultados obtidos para a instância I_3



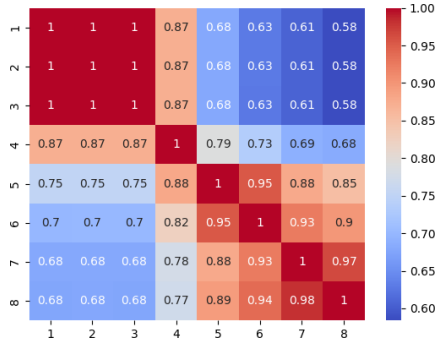
(d) Resultados obtidos para a instância I_4



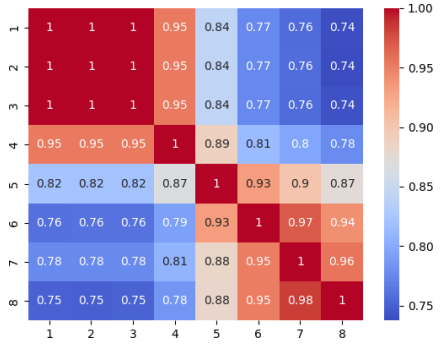
(e) Resultados obtidos para a instância I_5



(f) Resultados obtidos para a instância I_6



(g) Resultados obtidos para a instância I_7



(h) Resultados obtidos para a instância I_8

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

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 MultiStart. Adicionalmente, foi construído um *benchmark* para validação dos métodos propostos.

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; 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.; SHARMA, D. K. A state of art on source code plagiarism detection. In: IEEE. **2016 2nd International Conference on Next Generation Computing Technologies (NGCT)**. [S.l.], 2016. p. 236–241.

ALORF, A. A survey of recently developed metaheuristics and their comparative analysis. **Engineering Applications of Artificial Intelligence**, v. 117, p. 105622, 2023. ISSN 0952-1976. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0952197622006121>.

CHAVES, R. C.; ALBUQUERQUE, K. M.; CHAVES, R. C.; GOUVEIA, T. Modelo para detecção de plágio em exercícios de programação na linguagem python. In: SBC. **Anais do VIII Encontro de Teoria da Computação**. [S.l.], 2023. p. 196–200.

CHEERS, H.; LIN, Y.; SMITH, S. P. Academic source code plagiarism detection by measuring program behavioral similarity. **IEEE Access**, IEEE, v. 9, p. 50391–50412, 2021.

CUI, Y.; TAO, L.; BAO, D.; HAI, R. Analysis and research on gamification teaching strategy used in c language programming course. In: ATLANTIS PRESS. **2023 4th International Conference on Education, Knowledge and Information Management (ICEKIM 2023)**. [S.l.], 2023. p. 172–180.

GOUVEIA, T.; ALBUQUERQUE, K.; OLIVEIRA, J.; MACIEL, V. M. C073: ferramenta para apoio ao ensino de programação usando a metodologia de aprendizagem baseada em problemas. **Revista Principia - Divulgação Científica e Tecnológica do IFPB**, v. 60, n. 1, p. 70–87, 2023. ISSN 2447-9187. Disponível em: <https://periodicos.ifpb.edu.br/index.php/principia/article/view/5942>.

HORVÁTH, M.; PIETRIKOVÁ, E. An experimental comparison of three code similarity tools on over 1,000 student projects. In: IEEE. **2024 IEEE 22nd World Symposium on Applied Machine Intelligence and Informatics (SAMI)**. [S.l.], 2024. p. 000423–000428.

MACULAN, N.; FAMPA, M. H. C. Otimização linear. **Editora Universidade de Brasilia: Brasilia**, 2006.

POP, P. C.; COSMA, O.; SABO, C.; SITAR, C. P. A comprehensive survey on the generalized traveling salesman problem. **European Journal of Operational Research**, Elsevier, v. 314, n. 3, p. 819–835, 2024.

RONG, W.; XU, T.; SUN, Z.; SUN, Z.; OUYANG, Y.; XIONG, Z. An object tuple model for understanding pointer and array in c language. **IEEE Transactions on Education**, IEEE, 2023.

SOUZA, M. J. F. Inteligência computacional para otimização. **Notas de aula, Departamento de Computaçao, Universidade Federal de Ouro Preto, disponıvel em http://www. decom. ufop. br/prof/marcone/InteligenciaComputacional/InteligenciaComputacional. pdf**, v. 6, 2008.

WEN, W.; XUE, X.; LI, Y.; GU, P.; XU, J. Code similarity detection using ast and textual information. **International Journal of Performability Engineering**, v. 15, n. 10, p. 2683, 2019.

ZAKERI-NASRABADI, M.; PARSA, S.; RAMEZANI, M.; ROY, C.; EKHTIARZADEH, M. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. **Journal of Systems and Software**, Elsevier, p. 111796, 2023.

APÊNDICE A – GERAÇÃO DE INSTÂNCIAS E CÓDIGOS DE EXEMPLO

A.1 ALGORITMO DE GERAÇÃO DE INSTÂNCIAS

O código *python* abaixo cria uma instância a partir de dois programas em C. Onde primeiro, os dois códigos são transformados em *tokens*, conforme visto na Seção 3.2. E, após isso, é gerado um grafo seguindo o modelo de (CHAVES et al., 2023).

```
from src.code import Code  # código com a função de tokenizar dos programas
from src.graph import Graph  # código referente à criação do grafo seguindo
    o modelo apresentado
import sys

if __name__ == '__main__':
    # Processo de tokenização do código A
    a_name = sys.argv[1]
    a_file = open(a_name,'r')
    a = Code(a_file, language=a_name.split('.')[-1])
    a_file.close()

    # Processo de tokenização do código B
    b_name = sys.argv[2]
    b_file = open(b_name,'r')
    b = Code(b_file, language=b_name.split('.')[-1])
    b_file.close()

    graph = Graph(a, b) # geramos o grafo
    graph.printGraphData() # imprimimos o grafo
```

A.2 EXEMPLO 1

A.2.1 Programa Original

```
#include <stdio.h>
int main(){
  char c1[100],c2[100], c3[100];
  int v1,v2, v3;
  scanf("%s%d",c1, &v1);
  scanf("%s%d",c2, &v2);

  scanf("%s%d", c3, &v3);

  int t=v1+v2+v3;
  float p1 = v1 * 100.0 / t;
  float p2 = v2 * 100.0 / t;
  float p3 = v3 * 100.0 / t;


  printf("%s : %.1f %%\n", c1, p1);
  printf("%s : %.1f %%\n", c2, p2);
  printf("%s : %.1f %%\n", c3, p3);
}
```

A.2.2 Programa após Modificação 1

```c
#include <stdio.h>

int main() {
  char c1[100], c2[100], c3[100];
  int v1, v2, v3;
  scanf("%s%d", c1, &v1);
  scanf("%s%d", c2, &v2);

  scanf("%s%d", c3, &v3);

  int t = v1 + v2 + v3;

  float p1 = v1 * 100.0 / t;
  float p2 = v2 * 100.0 / t;
  float p3 = v3 * 100.0 / t;


  printf("%s : %.1f %%\n", c1, p1);
  printf("%s : %.1f %%\n", c2, p2);
  printf("%s : %.1f %%\n", c3, p3);
}
```

A.2.3 Programa após modificação 2

```c
#include <stdio.h>

int main() {
  // declarando variaveis dos
   candidatos
  char c1[100], c2[100], c3[100];
  // votos
  int v1, v2, v3;
  // leitura
  scanf("%s%d", c1, &v1);
  scanf("%s%d", c2, &v2);
  scanf("%s%d", c3, &v3);

  // total
  int t = v1 + v2 + v3;

  // calculando porcentagens
  float p1 = v1 * 100.0 / t;
  float p2 = v2 * 100.0 / t;
  float p3 = v3 * 100.0 / t;

  // imprimindo resultados
  printf("%s : %.1f %%\n", c1, p1);
  printf("%s : %.1f %%\n", c2, p2);
  printf("%s : %.1f %%\n", c3, p3);
}
```

A.2.4 Programa após modificação 3

```c
#include <stdio.h>

int main() {
  // declarando variaveis dos
   candidatos
  char cand1[100], cand2[100],
   cand3[100];
  // votos
  int votos1, votos2, votos3;
  // leitura
  scanf("%s%d", cand1, &votos1);
  scanf("%s%d", cand2, &votos2);
  scanf("%s%d", cand3, &votos3);

  // total
  int total = votos1 + votos2 +
   votos3;

  // calculando porcentagens
  float per1 = votos1 * 100.0 /
   total;
  float per2 = votos2 * 100.0 /
   total;
  float per3 = votos3 * 100.0 /
   total;

  // imprimindo resultados
  printf("%s : %.1f %%\n", cand1,
   per1);
  printf("%s : %.1f %%\n", cand2,
   per2);
  printf("%s : %.1f %%\n", cand3,
   per3);
}
```

A.2.5 Programa após modificação 4

```c
#include <stdio.h>
#define SIZE 100
```

```c
int main() {
  // declarando variaveis dos
   candidatos
  char cand1[SIZE], cand2[SIZE],
   cand3[SIZE];
  // votos
  int votos1, votos2, votos3;
  // leitura
  scanf("%s%d", cand1, &votos1);
  scanf("%s%d", cand2, &votos2);
  scanf("%s%d", cand3, &votos3);

  // total
  int total = votos1 + votos2 +
   votos3;

  // calculando porcentagens
  float per1 = votos1 * SIZE * 1.0
   / total;
  float per2 = votos2 * SIZE * 1.0
   / total;
  float per3 = votos3 * SIZE * 1.0
   / total;

  // imprimindo resultados
  printf("%s : %.1f %%\n", cand1,
   per1);
  printf("%s : %.1f %%\n", cand2,
   per2);
  printf("%s : %.1f %%\n", cand3,
   per3);
}
```

A.2.6 Programa após modificação 5

```c
#include <stdio.h>
#define SIZE 100

int main() {
  // declarando variaveis dos
   candidatos
  char cand1[SIZE], cand2[SIZE],
   cand3[SIZE];
  // votos
  int votos1, votos2, votos3;
  // leitura
  scanf("%s %d", cand1, &votos1);
  scanf("%s %d", cand2, &votos2);
  scanf("%s %d", cand3, &votos3);

  // total
  int total = votos1 + votos2 +
   votos3;

  // calculando porcentagens
  float per1 = votos1 * SIZE *
   01.00 / total;
  float per2 = votos2 * SIZE *
   01.00 / total;
  float per3 = votos3 * SIZE *
   01.00 / total;

  // imprimindo resultados
  printf("%s : %.1f %%\n", cand1,
   per1);
  printf("%s : %.1f %%\n", cand2,
   per2);
  printf("%s : %.1f %%\n", cand3,
   per3);
}
```

A.2.7 Programa após modificação 6

```c
#include <stdio.h>
#define SIZE 100

int main() {
  // votos
  int votos1, votos3, votos2;
  // declarando variaveis dos
   candidatos
  char cand2[SIZE], cand3[SIZE],
   cand1[SIZE];
  // leitura
  scanf("%s %d", cand1, &votos1);
  scanf("%s %d", cand2, &votos2);
  scanf("%s %d", cand3, &votos3);

  // total
  int total = votos1 + votos3 +
   votos2;

  // calculando porcentagens
```

```
  float per2 = votos2 * SIZE *
   01.00 / total;
  float per1 = votos1 * SIZE *
   01.00 / total;
  float per3 = votos3 * SIZE *
   01.00 / total;

  // imprimindo resultados
  printf("%s : %.1f %%\n", cand1,
   per1);
  printf("%s : %.1f %%\n", cand2,
   per2);
  printf("%s : %.1f %%\n", cand3,
   per3);
}
```

A.2.8 Programa após modificação 7

```
#include <stdio.h>
#define SIZE 100

// votos
int votos1, votos3, votos2;

// declarando variaveis dos
   candidatos
char cand2[SIZE], cand3[SIZE],
   cand1[SIZE];

float per1, per2, per3;

void read() {
  // leitura
  scanf("%s %d", cand1, &votos1);
  scanf("%s %d", cand2, &votos2);
  scanf("%s %d", cand3, &votos3);
}

void show() {
  // imprimindo resultados
  printf("%s : %.1f %%\n", cand1,
   per1);
  printf("%s : %.1f %%\n", cand2,
   per2);
  printf("%s : %.1f %%\n", cand3,
   per3);
}

int main() {
  read();

  // total
  int total = votos1 + votos3 +
   votos2;

  // calculando porcentagens
  per2 = votos2 * SIZE * 01.00 /
   total;
  per1 = votos1 * SIZE * 01.00 /
   total;
  per3 = votos3 * SIZE * 01.00 /
   total;

  show();
}
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Cajazeiras - Código INEP: 25008978
Rua José Antônio da Silva, 300, Jardim Oásis, CEP 58.900-000, Cajazeiras (PB)
CNPJ: 10.783.898/0005-07 - Telefone: (83) 3532-4100

Documento Digitalizado Ostensivo (Público)

TCC

Assunto:	TCC
Assinado por:	Joao Santos
Tipo do Documento:	Anexo
Situação:	Finalizado
Nível de Acesso:	Ostensivo (Público)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- **Joao Pedro Oliveira Santos**, **DISCENTE (202212010027) DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - CAJAZEIRAS**, em 28/07/2026 23:35:05.

Este documento foi armazenado no SUAP em 28/07/2026. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1928943
Código de Autenticação: aa0b0078b1