# Modelo para Detecção de Plágio em Exercícios de Programação na Linguagem Python

**Rafael C. Chaves**[1], **Kerven M. Albuquerque**[1], **Renato C. Chaves**[1], **Thiago Gouveia**[1]

[1]Instituto Federal da Paraíba (IFPB)
João Pessoa – PB – Brazil

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

***Abstract.*** *This article proposes an approach based on bipartite graphs with unidirectional edges to detect plagiarism in codes written in Python. The graph modeling is designed to allow the identification of the probability of plagiarism through solutions to the Maximal Reward Shortest Path problem.*

***Resumo.*** *Este artigo propõe uma abordagem baseada em grafos bipartidos com arcos unidirecionais para detectar plágio em códigos escritos em Python. A modelagem do grafo é projetada para permitir a identificação da probabilidade de plágio por meio de soluções para o problema do Menor Caminho com Premiação Máxima.*

## 1. Introdução

É possível observar que o ensino de programação vem ganhando crescente importância nos últimos anos, e isto se deve, em grande parte, ao papel cada vez mais central que a tecnologia desempenha na vida de todos. Por sua vez, segundo [Srinath 2017], Python é a linguagem de programação com o crescimento mais rápido nos últimos anos. Para [Leping et al. 2009] a linguagem Python tem sido reconhecida como uma escolha popular e eficaz para o ensino de programação.

No entanto, com a popularização do ensino de programação surge o sério problema da cópia de códigos. Os estudantes podem copiar códigos de fontes na Internet, de outros estudantes ou de amigos, em vez de escreverem seus próprios códigos. Tal comportamento pode ser extremamente prejudicial ao aprendizado. Dentre as estratégias de combate à cópia de código, podemos destacar o uso de ferramentas anti-plágio, capazes de ajudar a identificar possíveis casos de cópia de códigos e ajudar a reforçar a importância da originalidade.

Com o objetivo de auxiliar os professores no combate à cópia de código-fonte, este trabalho propõe um modelo para detecção de plágio em exercícios de programação escritos na linguagem Python. Dentre as características do modelo apresentado, destacam-se o seu comportamento determinístico, a sua capacidade de explicar as similaridades e dissimilaridades entre os códigos comparados e a sua saída: o grau de certeza quanto à cópia, em escala percentual. O restante deste trabalho está organizado como segue: a Seção 2 detalha o modelo proposto, a Seção 3 apresenta alguns resultados obtidos e uma pequena discussão acerca destes, e a Seção 4 traz as conclusões e propostas de trabalhos futuros.

## 2. Modelo Proposto

Sejam dados dois programas $A$ e $B$ escritos na linguagem Python, representados como sequências de instruções $A = (a_1, a_2, ..., a_n)$ e $B = (b_1, b_2, ..., b_m)$, respectivamente. Para resolver o problema de detecção de plágio, podemos considerar que há uma alta probabilidade de plágio quando é possível emparelhar os conjuntos de instruções $A$ e $B$ de modo que a maioria dos pares de instruções formados apresentem alta similaridade.

O problema de detecção de plágio pode ser abordado como um problema de grafos. Para isso, define-se o grafo bipartido $G$ com o conjunto de vértices $V$, que contém dois vértices especiais denominados $s$ e $t$, e um vértice para cada instrução contida em $A$ e $B$. O conjunto de arcos $E$ é composto por arcos que partem de cada $a_i$ em direção a $b_j$, indicando a similaridade entre as respectivas instruções. Além disso, há arcos que partem de cada $b_j$ em direção a cada $a_i$, para identificar sequências contíguas de código similar. Em relação aos vértices especiais, existem arcos com peso nulo que partem de $s$ em direção a $t$ e a cada $a_i$. Já quanto a $t$, não há nenhum arco partindo dele, mas há um conjunto de arcos que partem de $b_j$ e $s$ em direção a $t$, todos também com peso nulo.

Baseado no modelo dado, o problema da identificação de plágio pode ser visto como o Problema de Menor Caminho com Premiação Máxima (PMCPM), onde o objetivo é partir do vértice $s$ e utilizar o menor trajeto até $t$, de forma que a maior premiação seja atingida. Por tanto o objetivo é maximizar a equação 1, onde $V'$ é o conjunto dos vértices que compõe a solução, $E'$ o conjunto das arestas utilizadas no trajeto escolhido e $p$ a premiação fixa dada para cada vértice no conjunto $V'$.

$$|V'| \cdot p - \sum_{k=1}^{|E'|} e_k \tag{1}$$

Cada instrução $a_i$, por sua vez, é uma sequência de $k$ *tokens* $(t_1\ t_2\ \cdots\ t_{k-1}\ t_k)$. Um *token* pode ser definido como uma *substring* de uma instrução contida no programa, de acordo com as regras sintáticas da linguagem Python. Cada *token* possui um tipo associado, de acordo com sua função no programa, podendo ser considerado uma palavra-chave. Uma vez que é possível dividir uma instrução em *tokens*, podemos inicialmente calcular a diferença entre um *token* $t_a$ e um $t_b$. Esse procedimento é descrito abaixo.

1. **Se** $t_a$ é igual à $t_b$: Retorne distância mínima (0.0);
2. **Se** $t_a$ e $t_b$ possuem tipos diferentes: Retorne distância máxima (1.0);
3. **Se** $t_a$ ou $t_b$ é palavra-chave: Retorne distância máxima (1.0);
4. **Caso contrário**: Retorne a distância de edição entre dois *tokens* (Eq.2) normalizada no intervalo [0.0 - 0.5];

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

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 $c_a$ representa o conjunto de *tokens*

que compõem uma instrução $a_i$. Para chegar a isso, lança-se mão de uma segunda versão da distância de edição (Eq.3). Uma vez utilizando essa equação, o resultado deve ser normalizado no intervalo [0, 1000] e por fim obter o peso do arco que parte de $a_i$ para $b_j$.

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

Para finalizar a modelagem do grafo, restou o cálculo do peso de cada arco que parte de $b_j$ para $a_i$. Como dito anteriormente, essas arcos são utilizadas para influenciar na escolha do próximo par de instruções a serem pareados. Definindo $a_x$ e $b_j$ o último par de instruções pareadas e o próximo pareamento $a_i$ e $b_y$. Considere o seguinte procedimento para o cálculo do peso da arco de volta:

1. **Se** $i = i + 1$: Retorne 0;
2. **Se** $i > x + 1$: Retorne $min(50 + (i - x - 2) * 20, 200)$
3. **Caso contrário**: Retorne $min(100 + (i - x - 2) * 10, 400)$

## 3. Resultados

Nesta seção, será demonstrado o processo de modelagem do grafo a partir da comparação entre o Código A, apresentado na Figura 1, e o Código B, apresentado na Figura 2. As instruções do Código A serão denotadas por ($a_1$, $a_2$, $a_3$ e $a_4$), enquanto as do Código B serão denotadas por ($b_1$, $b_2$, $b_3$ e $b_4$).

```
1 n = int(input())
2 m = 2 * n + 1
3 for i in range(1, m, 2):
4     print(i)
```

**Figura 1.** Código A.

```
1 num = int(input())
2 for k in range(num):
3     x = 2 * k + 1
4     print(x)
```

**Figura 2.** Código B.

A Tabela 1 mostra o resultado da *tokenização* de $b_1$. Nela é possível notar que a instrução foi separada em pequenas *strings*, classificadas de acordo com o seu tipo e se são palavras-chave. A Tabela 2 mostra o resultado da aplicação do procedimento entre os *tokens* das instruções $a_1$ e $b_1$ é apresentado na tabela .

| Token | Tipo | Palavra-chave |
|---|---|---|
| num | name | False |
| = | op | False |
| int | name | False |
| ( | op | False |
| input | name | False |
| ( | op | False |
| ) | op | False |
| ) | op | False |

**Tabela 1.** Tokenização da instrução $b_1$.

| | **n** | **=** | **int** | **(** | **input** | **(** | **)** | **)** |
|---|---|---|---|---|---|---|---|---|
| **num** | 0.250 | 1.000 | 0.333 | 1.000 | 0.250 | 1.000 | 1.000 | 1.000 |
| **=** | 1.000 | 0.000 | 1.000 | 0.500 | 1.000 | 0.500 | 0.500 | 0.500 |
| **int** | 0.250 | 1.000 | 0.000 | 1.000 | 0.125 | 1.000 | 1.000 | 1.000 |
| **(** | 1.000 | 0.500 | 1.000 | 0.000 | 1.000 | 0.000 | 0.500 | 0.500 |
| **input** | 0.333 | 1.000 | 0.125 | 1.000 | 0.000 | 1.000 | 1.000 | 1.000 |
| **(** | 1.000 | 0.500 | 1.000 | 0.000 | 1.000 | 0.000 | 0.500 | 0.500 |
| **)** | 1.000 | 0.500 | 1.000 | 0.500 | 1.000 | 0.500 | 0.000 | 0.000 |
| **)** | 1.000 | 0.500 | 1.000 | 0.500 | 1.000 | 0.500 | 0.000 | 0.000 |

**Tabela 2.** Distância entre os *tokens* das instruções $a_1$ e $b_1$.

Na Tabela 3, são apresentados os pesos dos arcos que partem dos vértices que representam as instruções do Código A. Esses pesos são obtidos a partir dos algoritmos de distância de edição e indicam a diferença entre elas. A Tabela 4 apresenta os pesos dos Arcos que partem dos vértices que representam o Código B. Já na figura 3, é mostrado a solução que maximiza o resultado da equação 1 a partir da instância apresentada nesta seção utilizando uma premiação fixa $p$ de 250.

| | $b_1$ | $b_2$ | $b_3$ | $b_4$ |
|---|---|---|---|---|
| $a_1$ | 31 | 544 | 560 | 410 |
| $a_2$ | 485 | 719 | 250 | 508 |
| $a_3$ | 600 | 143 | 650 | 636 |
| $a_4$ | 438 | 636 | 624 | 125 |

**Tabela 3.** Arcos partindo de $a_i$.

| | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $s$ |
|---|---|---|---|---|---|
| $a_1$ | 100 | 120 | 140 | 160 | 0 |
| $a_2$ | 0 | 100 | 120 | 140 | 0 |
| $a_3$ | 50 | 0 | 100 | 120 | 0 |
| $a_4$ | 60 | 50 | 0 | 100 | 0 |
| $t$ | 0 | 0 | 0 | 0 | 0 |

**Tabela 4.** Arcos partindo de $b_j$ e $s$.

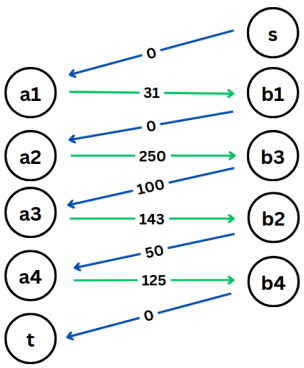


**Figura 3.** Solução ótima para a modelagem apresentada.

## 4. Conclusão

Com base nos resultados apresentados na última seção, é possível concluir que a modelagem de um grafo bipartido unidirecional é uma abordagem viável para o problema de identificação de plágio. Sendo assim, sugere-se como trabalhos futuros a resolução do problema do Menor Caminho com Premiação Máxima em instâncias geradas a partir da modelagem apresentada nesse trabalho utilizando programação Matemática e metaheurísticas.

## Referências

Leping, V., Lepp, M., Niitsoo, M., Tõnisson, E., Vene, V., and Villems, A. (2009). Python prevails. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 1–5.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357.