Heurísticas para o Problema do Alinhamento de Sequências em Grafos de Sequência sob a Distância de Edição

– Trabalho de Conclusão de Curso –

João Victor da Silva Santos[1], **Lucas de Souza Villar**[1], **Said Sadique Adi**[1]

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS), Campo Grande – MS – Brasil

victor_silva@ufms.br, lucas_villar@ufms.br, said.sadique@ufms.br

Abstract. In Computational Biology, the alignment and comparison of genetic sequences is a recurring problem. In the traditional approach, a reference sequence is used, however, this methodology can introduce significant biases, given that such a reference is inherently incapable of representing all possible variations. In this scenario, sequence graphs emerge as a robust data structure capable of consolidating multiple sequences and their respective variations into a single representation. Aligning a sequence to a sequence graph consists of identifying a path in the graph that induces a sequence that is as close as possible to the input sequence. This problem is known in the literature as the Sequence Graph Alignment Problem (SGAP) and it is known to be NP-complete. In the present study, two heuristics for this problem are proposed, implemented in C++ and evaluated on artificial test cases. The results obtained corroborate the effectiveness of the approach and indicate its potential for application in real world scenarios.

Resumo. Na Biologia Computacional, o alinhamento e a comparação de sequências genéticas constituem um problema recorrente. Na abordagem tradicional, se emprega uma sequência de referência, no entanto, esta metodologia pode introduzir vieses significativos, visto que essa referência é inerentemente incapaz de representar as variabilidades genéticas possíveis. Como alternativa, os grafos de sequência surgem como uma estrutura de dados robusta, capaz de consolidar múltiplas sequências e suas respectivas variações em uma única representação. Alinhar uma sequência com um grafo de sequência consiste em identificar um caminho no grafo que induza uma sequência que seja a mais próxima possível da sequência de entrada. Este problema é conhecido na literatura como Problema do Mapeamento de Sequências em Grafos de Sequência (PMSGS) e é sabidamente NP-completo. No presente estudo, propõe-se duas heurísticas para esse problema, que foram implementadas em C++ e avaliadas sobre casos de testes artificiais. Os resultados obtidos corroboram a eficácia da abordagem e indicam seu potencial para aplicação em cenários reais.

1 Introdução

Na Biologia Computacional, o alinhamento e a comparação de sequências genéticas constituem um problema recorrente. Na abordagem tradicional, se emprega uma sequência de

referência, no entanto, esta metodologia pode introduzir vieses significativos, visto que essa referência é inerentemente incapaz de representar as variabilidades genéticas possíveis. Como alternativa, os grafos de sequência surgem como uma estrutura de dados robusta, capaz de consolidar múltiplas sequências e suas respectivas variações em uma única representação. Com a introdução dessa nova estrutura, a tarefa de comparação e alinhamento de sequências se transforma em um novo problema computacional centrado no alinhamento de sequências no grafo de sequência.

É nesse contexto que se insere o **Problema do Mapeamento de Sequências em Grafos de Sequência (PMSGS)** que consiste em, dados um grafo de sequência G e uma sequência s, encontrar um caminho p em G que minimize a distância entre a sequência induzida por p e s, baseado em algum critério de distância. No presente trabalho, teremos como foco o estudo quando o critério escolhido é a **distância de Levenshtein**.

No artigo *"Read Mapping on De Bruijn Graphs"*[Limasset et al., 2016], foi abordado o problema de mapear sequências em grafos de sequência, utilizando a distância de Hamming como medida de distância entre a sequência de entrada e a sequência induzida no grafo. No mesmo artigo, esse problema foi provado ser NP-completo. Em razão da semelhança do problema abordado no artigo com o estudado no presente trabalho, conjectura-se que o PMSGS, também é NP-completo. Dado a complexidade do problema, propõe-se duas heurísticas que visam abordar o PMSGS.

O presente trabalho está organizado da seguinte forma. Na Seção 2, é apresentado alguns conceitos preliminares necessários para melhor entendimento do problema abordado. Após isso, na Seção 3, é explicado e contextualizado o Problema do Mapeamento de Sequências em Grafos de Sequência (PMSGS). Continuando, na Seção 4 é conjecturada a dificuldade do problema abordado baseado em outros trabalhos e sua equivalência a problemas conhecidamente NP-completo. Na Seção 5, é desenvolvida, definida e estruturada as heurísticas propostas. Na Seção 6, são apresentados os resultados obtidos em testes artificias e analisado a eficácia da abordagem proposta. Por fim, é feita a conclusão deste trabalho apresentando possíveis melhorias à proposta, orientações futuras e a necessidade de avaliação em casos reais.

2 Conceitos preliminares

Para compreender adequadamente o problema abordado neste estudo, esta seção apresenta alguns conceitos preliminares essenciais. Na Seção 2.1, é apresentado o conceito de sequência e outros conceitos relacionados, como o conceito de distância, com foco em distância de edição. Na Seção 2.2, é introduzido o conceito de grafo e conceitos relacionados, como o de caminhos e circuitos em grafos, além de introduzir o conceito de componentes fortemente conexas e algoritmos relacionados. Por fim, na Seção 2.3 é abordado o conceito de grafo de sequencia.

2.1 Sequência

Um **alfabeto** Σ corresponde a um conjunto não vazio e finito de símbolos. Uma **sequência** s construída sobre Σ é um conjunto ordenado de símbolos de Σ. Denota-se uma sequencia

s por $s = s[1]s[2]s[3]...s[|s|]$ em que $s[i]$ é o i-ésimo símbolo de s para $1 \leq i \leq |s|$, onde $|s|$ representa o tamanho da sequência s. Uma sequência s tal que $|s| = 0$ é chamada de sequência **vazia**.

Um **segmento** de uma sequência s consiste em um conjunto de símbolos contínuos de s. Dessa forma, um segmento é denotado, dado uma posição i de início e outra posição j de fim, por $s[i, j]$. Formalmente, $s[i, j] = s[i]s[i+1]...s[j]$ para $1 \leq i \leq j \leq |s|$. O segmento é chamado de **prefixo** quando começa desde o início da sequência, ou seja $i = 1$.

Por exemplo, dado uma sequencia $s = ACTTG$, construída sobre o alfabeto $\Sigma = \{A, C, G, T\}$ temos que $|s| = 5$, $s[1] = A$ e $s[4] = T$. Também, temos que o segmento $s[1, 3] = ACT$ é prefixo de s.

2.1.1 Distância entre sequências

A distância entre duas sequências s e t pode ser entendida como uma maneira de mensurar a diferença entre elas baseado em algum critério. Dentre as principais medidas de distância propostas na literatura, destaca-se a distância de Hamming [Hamming, 1950], que calcula o número de posições correspondentes cujos símbolos são distintos em duas sequências de mesmo tamanho, e a distância de Levenshtein [Levenshtein, 1966], que será o foco deste estudo.

A distância de Levenshtein, ou **distância de edição**, é definida como o menor número de operações de inserção, substituição ou remoção de caracteres necessárias para transformar uma sequência em outra.

O **Problema de Determinar a Distância de Edição**, dado duas sequências s e t, é classicamente resolvido utilizando programação dinâmica. Esse algoritmo, também introduzido por Levenshtein [Levenshtein, 1966], possui complexidade quadrática, sendo expressa como $O(|s| \times |t|)$. Por fim, a distância de edição entre s e t é denotada por $D(s, t)$.

Por exemplo, considere as sequências $s = ACTTG$ e $r = CTATA$ sobre o alfabeto $\Sigma = \{A, C, T, G\}$. Podemos realizar a seguinte ordem de operações para transformar s em r:

1. Remoção de $s[1]$: $ACTTG$ -> $CTTG$
2. Inserção de 'A' após $s[2]$: $CTTG$ -> $CTATG$
3. Substituição de $s[5]$ por 'A': $CTATG$ -> $CTATA$

Dessa forma, a distância de edição $D(s, r)$ para transformar s em r é 3.

2.2 Grafos

Um **grafo** G é um par ordenado (V, A) em que V é um conjunto qualquer cujos elementos representam os **vértices** do grafo e A é um conjunto de pares de vértices que representam os **arcos** do grafo. Também, é chamado de $V(G)$ e $A(G)$ o conjunto de vértices e arcos do grafo G.

Um arco é representado por um par ordenado de vértices (u, v). Dado um arco (u, v) de um grafo G, se diz que o arco começa em u e termina em v e também que v é **vizinho**

ou **adjacente** a u. Um grafo composto por arcos também pode ser chamado de **grafo dirigido** ou **direcionado**[1].

O **peso** ou **custo** de um arco é um número real que pode ser obtido por meio de uma função que o associa a cada arco de um grafo. Formalmente, seja P uma função, temos que o peso de um arco pode ser obtido por $P((u,v)) \to \mathbb{R}$. Um grafo cujos arcos possuem peso é chamado de **grafo ponderado**. Uma ilustração dos conceitos citados pode ser visto na Figura 1.

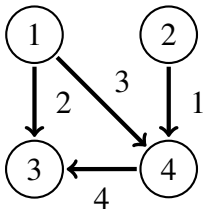

Figura 1: Nesse exemplo é ilustrado um grafo G em que $V(G) = \{1,2,3,4\}$ e $A(G) = \{(1,3),(1,4),(2,4),(4,3)\}$. Também, temos que G é ponderado e o peso do arco $(4,3)$ é 4, assim, $P((4,3)) = 4 \in \mathbb{R}$.

Um grafo H é **subgrafo** de um grafo G quando se obtém H pela remoção de alguns vértices e arcos de G. Formalmente, H é subgrafo de G ($H \subseteq G$) se e somente se $V(H) \subseteq V(G)$ e $A(H) \subseteq A(G)$.

Um exemplo de subgrafo pode ser visto na Figura 2.

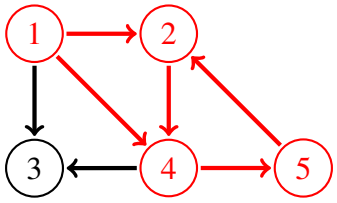

Figura 2: Nesse exemplo, seja um grafo G em que $V(G) = \{1,2,3,4,5\}$ e $A(G) = \{(1,2),(1,3),(1,4),(2,4),(4,3),(4,5),(5,2)\}$. Temos, em vermelho, o subgrafo H de G obtido ao remover o vértice 3 e os arcos $(1,3)$ e $(4,3)$. Dessa forma, é fácil notar que $V(H) \subseteq V(G)$ e $A(H) \subseteq A(G)$.

[1]Normalmente, define-se um grafo com arcos não ordenados, chamados de **arestas**. Por essa definição não ter relevância no presente trabalho, tomou-se a liberdade de apenas considerar os grafos dirigidos.

2.2.1 Passeios, circuitos e caminhos

Dado um grafo $G(V,A)$, um **passeio** em G é uma sequência ordenada de vértices $p = v_1, v_2, v_3, ..., v_{|p|}$ tal que para cada par (v_i, v_{i+1}) temos que arco $(v_i, v_{i+1}) \in A$ para todo $1 \leq i \leq |p| - 1$. Um passeio se diz fechado quando se tem pelo menos dois arcos e seu primeiro vértice é igual a seu último, ou seja, $v_1 = v_{|p|}$.

Um passeio fechado em que não se repete nenhum arco é chamado de **ciclo** ou **circuito**. Um **caminho** é um passeio que não repete vértices. Por fim, se um grafo não possui nenhum circuito, ele é chamado de grafo **acíclico** ou *DAG* (*Directed Acyclic Graph*). Intuitivamente, um grafo que possui algum ciclo é chamado de grafo **cíclico**.

O **custo de um passeio** é o somatório do peso de todos os arcos que conectam os vértices do passeio. Isso permite comparar diferentes trajetos entre dois vértices distintos. Formalmente, o custo de um passeio p é $C(p) = \sum_{i=1}^{|p|-1} P((v_i, v_{i+1}))$. Dentre todos os caminhos entre dois vértices de um grafo, aquele de menor custo é chamado de **caminho mínimo**.

O **Problema do Caminho Mínimo** consiste em encontrar, dados um grafo ponderado G e dois vértices s e t de G, um caminho mínimo entre s e t. O problema do caminho mínimo é amplamente estudado na literatura. Um exemplo de algoritmo que resolve este problema quando o peso das arestas é não negativo é o algoritmo de Dijkstra [Dijkstra, 1959], com complexidade $O(|A| \log |V|)$, onde $|A|$ e $|V|$ representam o tamanho dos conjunto das arestas e vértices do grafo, respectivamente.

Por exemplo, na Figura 1, os caminhos $p_1 = \{1,4,3\}$ e $p_2 = \{1,3\}$ de G possuem custo $C(p_1) = 7$ e $C(p_2) = 2$. Também, p_2 é o caminho mínimo entre os vértices 1 e 3 já que não existe nenhum outro caminho com custo menor.

2.2.2 Componentes Fortemente Conexos

Um **componente fortemente conexo**, ou *SCC* (*Strongly Connected Component*), de um grafo dirigido $G(V,A)$ é um subgrafo de G formado por um conjunto maximal de vértices $C \subseteq V(G)$ tal que, para todo par $(u,v) \in C$, u alcança v e v alcança u, ou seja, existe um caminho no grafo que conecta u a v e vice-versa. Um exemplo desse conceito é ilustrado na Figura 3.

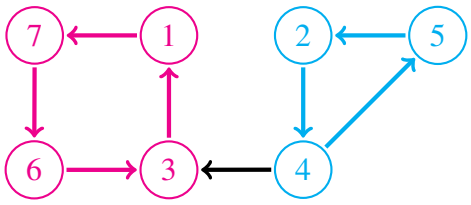

Figura 3: Nesse exemplo, é possível ver dois diferentes componentes fortemente conexos presentes no grafo. Sendo o componente magenta $C_M = \{1,7,6,3\}$ e o componente ciano $C_C = \{2,5,4\}$.

Uma característica marcante de um componente fortemente conexo é a presença de ciclos já que sempre é possível encontrar algum passeio fechado que começa em u passa por v e retorna à u para algum par (u,v), em que u atinge v e vice-versa. Por exemplo,

na Figura 3, ao juntar o caminho $p_1 = \{1,7,6\}$ e $p_2 = \{6,3,1\}$, sem repetir o vértice 6, é encontrado o circuito $p_3 = \{1,7,6,3,1\}$.

O **Problema de Encontrar Componentes Fortemente Conexos de um Grafo** pode ser resolvido em tempo linear utilizando o algoritmo de Kosaraju [Sharir, 1981]. Também, ao identificar cada componente fortemente conexo é possível realizar a condensação do grafo, isto é, substituir toda *SCC* por um único vértice no grafo. A condensação permite a transformação do grafo cíclico estudado em um grafo acíclico ou *DAG*. Na Figura 4 é visto o resultado da condensação do grafo representado na Figura 3.

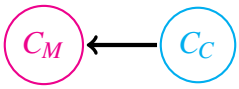

Figura 4: Nessa exemplo é ilustrado o resultado da condensação do grafo representado na **Figura 3**.

2.3 Grafo de sequência

Um **grafo de sequência** é um grafo em que cada vértice possui uma sequência associada construída sobre um certo alfabeto Σ [Braga et al., 2000]. Essa sequência vinculada a cada vértice é denominada **rótulo** do vértice. Um **grafo de sequência simples**, ou *SSG* (*Simple Sequence Graph*), é aquela em que cada vértice é rotulado com apenas um único símbolo do alfabeto [Amir et al., 1997, Rautiainen and Marschall, 2017, Jain et al., 2020]. Grafos de sequência simples serão o foco deste estudo. Um exemplo desse tipo de grafo pode ser visto na Figura 5.

Dado um passeio p em um grafo de sequência, a **sequência induzida** pelo passeio, denotada por $i(p)$, é a sequência obtida pela concatenação de cada sequência vinculada aos vértices do passeio. Um exemplo dessa definição também encontra-se ilustrado na Figura 5.

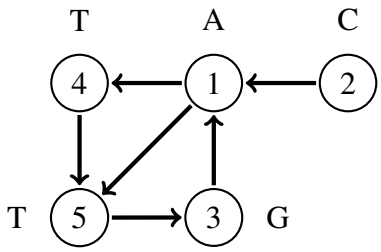

Figura 5: Nesse exemplo é ilustrado um grafo de sequência simples. Também, dado um passeio $p = \{2,1,5,3\}$, podemos induzir a sequência $i(p) = CATG$ de tamanho 4.

3 Problema do Mapeamento de Sequências em Grafos de Sequência (PMSGS)

O **Problema do Mapeamento de Sequências em Grafos de Sequência**, restrito a grafos de sequência simples, consiste em, dados um grafo de sequência simples $G(V,A)$, cujos rótulos são sequências sobre um alfabeto Σ, e uma sequência de entrada s, também construída sobre Σ, encontrar um caminho p em G tal que a distância de edição $D(s,i(p))$ é a menor possível. Uma instância do problema pode ser vista no Exemplo 1.

Exemplo 1. Considerando o grafo G apresentado na Figura 6 e a sequência $s = GAATGC$, analisam-se dois possíveis caminhos, $p_1 = \{7,2,1,5,3,6\}$ e $p_2 = \{3,1,4,5\}$, que induzem, respectivamente, as sequências $i(p_1) = GCATGA$ e $i(p_2) = GATT$. Assim, é obtido, pela distância de edição, $D(s,i(p_1)) = 2$ e $D(s,i(p_2)) = 3$. Isso mostra que $i(p_1)$ é mais próxima de s do que $i(p_2)$. Além disso, pode-se verificar que não existem outros caminhos em G que induzem uma sequência tal que a distancia de edição seja menor em relação a s que a de $i(p_1)$. Logo, $i(p_1)$ se mostra como uma possível resposta ao problema.

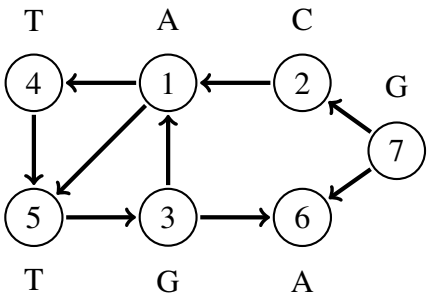

Figura 6: Exemplo de um Grafo de Sequência Simples.

4 Dificuldade

No artigo intitulado *"Read Mapping on De Bruijn Graphs"* [Limasset et al., 2016] é abordado o **Problema de Mapeamento de *Reads* em Grafos** (*Graph Read Mapping Problem* ou *GRMP*) e provado ser NP-completo.

Formalmente, o *GRMP* consiste em, dados um grafo de sequência G, onde os arcos são rotulados por símbolos de um alfabeto Σ, uma sequência s, construída sobre o mesmo alfabeto, e um limite superior de custo k, determinar se existe algum caminho c em G com tamanho $|s|$ que induza uma sequência tal que a distância de Hamming entre $i(c)$ e s seja menor ou igual a k.

Dada a definição do *GRMP*, entende-se que o problema abordado nesse estudo é computacionalmente semelhante a ele. Isso, em razão da equivalência entre as métricas de distância e entre a estrutura dos grafos. Observe que a distância de Hamming é uma versão restrita da distância de edição que faz uso apenas da operação de substituição. Além disso, os grafos de sequência com rótulos nos arcos e grafos com rótulos nos vértices são estruturas equivalentes. A conversão entre eles pode ser feita por transformações triviais

no grafo de entrada. O procedimento consiste em criar um novo vértice para cada arco e atribuir a esse novo vértice o rótulo do arco. Após isso, são definidas novas conexões intermediárias entre os vértices originais, preservando a estrutura do grafo.

Por meio das semelhanças apresentadas, conjectura-se que o problema abordado no presente trabalho também é NP-completo.

5 Heurísticas para o problema

5.1 Grafo de alinhamento

O Problema do Mapeamento de Sequências em Grafos de Sequência (PMSGS) possui uma solução em tempo polinomial quando se tem o objetivo de encontrar um *passeio* que induza uma sequência mais próxima possível da sequência de entrada. Essa solução é construída reduzindo o problema ao de se encontrar um caminho mínimo em um tipo específico de grafo, chamado de **grafo de alinhamento** [Rautiainen and Marschall, 2017, Jain et al., 2020].

O grafo de alinhamento é um grafo direcionado e ponderado que representa o conjunto de todos os alinhamentos possíveis entre uma sequência r e um grafo de sequência $G(V,A)$. Esse grafo é construído como um grafo com várias "camadas", composto por $|r|$ cópias do grafo de sequência original G. A i-ésima camada contém o conjunto de todos os vértices e arcos do grafo de sequência e corresponde ao alinhamento do i-ésimo símbolo da sequência r, representado por $r[i]$. Os arcos entre vértices distintos do grafo são definidos com base nas operações inserção, substituição e remoção da distância de edição:

- **Arcos de Inserção:** conectando vértices u e v dentro de uma mesma camada representando a operação de inserção na sequência de entrada, esses arcos são representados por arcos horizontais e possuem peso 1.

- **Arcos de Remoção:** conectando vértices u e v de camadas adjacentes, que representam o mesmo vértice do grafo de sequência, esses arcos representam a remoção de um símbolo na sequência de entrada; esses arcos são representado por arcos verticais e também possuem peso 1.

- **Arcos de substituição:** conectando vértices u e v também de camadas adjacentes representando a substituição de um símbolo na sequência de entrada, esses arcos apenas existem se e somente se o arco (u,v) também está presente no grafo de sequência; seja i a camada do vértice v no grafo de alinhamento, o peso de cada um desses arcos é dado pela função $f(r[i], rot_v)$, em que rot_v representa o rótulo de v, definida como:

$$f(r[i], rot_v) = \begin{cases} 0, & \text{se } r[i] = rot_v \\ 1, & \text{se } r[i] \neq rot_v \end{cases} \tag{1}$$

Ademais, também definimos o vértice s, chamado de ***source***, o qual se conecta a todos os vértices da primeira camada com arcos de substituição e o vértice t, chamado de ***sink***,

ao qual todos os vértices da última camada se conectam com peso 0. Isso garante que todo alinhamento possível possua o mesmo ponto final e inicial.

Por fim, temos os vértices chamados de ***dummy***, presente em todas as camadas, que se conectam com outros vértices dummy e todos os vértices de camadas adjacentes. Esses vértices permitem representar a remoção de um prefixo da sequência r.

Com essa estrutura estabelecida, temos que o caminho mínimo entre os vértices s e t, no grafo de alinhamento, induz a sequência mais próxima da sequência r sob a distância de edição. Desse modo, o custo desse caminho mínimo corresponde diretamente à distância de edição entre as duas sequências.

Um exemplo de grafo de alinhamento com as noções e notações observadas acima pode ser visto na Figura 7.

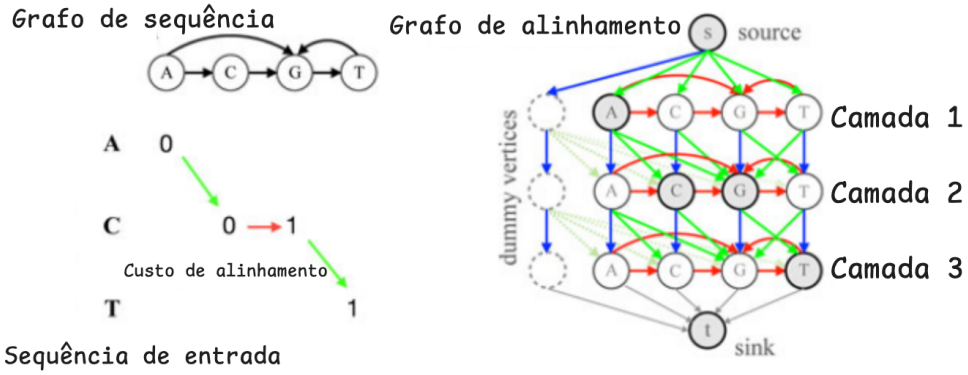

Figura 7: Exemplo de construção de um grafo de alinhamento dado uma sequência $r = ACT$ e um grafo de sequência G. Por meio deste, podemos ver os vértices *dummy*, *source*, *sink* e as várias camadas do grafo de alinhamento. Os arcos coloridos representam as operações da distância de edição, com a inserção, substituição e remoção sendo representadas, respectivamente, pelas cores vermelho, verde e azul. Por fim, temos que o alinhamento da sequência r no grafo de sequência G é um caminho no grafo de alinhamento que induz a sequência $ACGT$ e possui custo 1, esse é equivalente a distância à edição das duas sequências [Jain et al., 2020].

5.2 Heurísticas

Inicialmente, dados uma sequência s e um grafo de sequência G, é construído o grafo de alinhamento G' no qual cada caminho entre os vértices *source* e *sink* do grafo representam um possível alinhamento entre a sequência e o grafo de sequência.

Em seguida, é determinado o caminho mínimo entre os vértices *source* e *sink* de G', chamado de w, por meio do Algoritmo de Dijkstra. Com o caminho mínimo, determina-se o passeio p correspondente em G, o qual é utilizado para construir um subgrafo H de G formado pelos vértices e arestas que compõem o passeio.

Assim, é obtido o grafo H, subgrafo de G, composto apenas dos vértices e arestas necessários para encontrar o passeio que induz a sequência com menor distância de edição da sequência s.

Nessa etapa, o objetivo é encontrar, no grafo H, um caminho c que possua o maior número de vértices, partindo do vértice inicial q, onde q é o primeiro vértice do passeio p, encontrado anteriormente em G, ou seja, $q = p[1]$. Formalmente, dados o grafo H e o vértice q, buscamos o caminho c que inicia em q e cujo tamanho seja o maior possível. Dessa forma, conjectura-se que esse caminho c no grafo H induz a sequência mais próxima da sequência s sob distância de edição.

As duas heurísticas presentes no trabalho exploram abordagens distintas para essa etapa. A primeira, chamada de *Heurística DFS*, aborda o problema por encontrar o caminho mais longo que se inicia no vértice q utilizando busca em profundidade. A segunda, denominada *Heurística DAG*, aborda o problema transformando o grafo H em um *DAG* e encontrando o maior caminho nele a partir do vértice q.

5.2.1 Heurística DFS

A heurística DFS (*Depth First Search*) consiste em, dado o grafo H e o primeiro vértice do passeio q, extrair um caminho com a maior quantidade de vértices aplicando uma busca em profundidade sobre o grafo H a partir do vértice q. Essa busca é realizada computando recursivamente para cada vértice do grafo o tamanho do caminho de q até ele. A maneira que isso é feito é representado no **Pseudocódigo 1**

Pseudocódigo 1 buscaProfundidade

Entrada: vértice **u**, grafo de sequência **H**, conjunto de vértices processados **L**, vetor **distancia**
Resultado: Atualiza as distâncias dos vértices do grafo H
$L \leftarrow L \cup \{u\}$
para cada vértice v adjacente a u em H **faça**
  se $v \notin L$ **então**
    $distancia[v] \leftarrow distancia[u] + 1$
    buscaProfundidade$(v, H, L, distancia)$
  fim se
fim para

Com esse processo estabelecido, é possível construir a heurística DFS proposta. Então, dado o grafo H e a sequência de entrada s, é computado a partir do vértice q a distância para todos os vértices do grafo H utilizando busca em profundidade. Com isso, é possível recuperar o maior caminho que começa em q trivialmente. Por fim, é computado a distância da sequência induzida pelo caminho e a sequência de entrada s sob a distância de edição. A maneira que isso é feito é representado no **Pseudocódigo 2**.

Pseudocódigo 2 Heurística DFS

Entrada: grafo de sequência $\mathbf{G}(V,A)$, sequência $\mathbf{s}$
Resultado: distância de edição d
// Inicializa estruturas e vetores auxiliares //
$G' \leftarrow$ Grafo de Alinhamento (s,G)
$w \leftarrow$ Caminho mínimo (G')
$p \leftarrow$ passeio correspondente em G dado w
$H \leftarrow$ grafo induzido pelo passeio p
$q \leftarrow p[1]$
$distancia[q] = 1$
buscaProfundidade$(q,H,L,distancia)$
$c \leftarrow$ encontra maior caminho de H
$s' \leftarrow$ sequência induzida por c
devolva Distancia de Edição(s,s')

A complexidade da heurística DFS é predominantemente determinada pela execução do algoritmo de Dijkstra no grafo de alinhamento (G'). Devido a necessidade de construir esse grafo, que possui $|V(G')| = |s| \times |V(G)|$ vértices e $|A(G')| = (|s| \times (|V(G)| + |A(G)|)$ arcos, a complexidade total é dada por $O((|s| \times (|V(G)| + |A(G)|) \times log(|s| \times |V(G)|))$, onde $|s|$ é o comprimento da sequência de entrada e G é o grafo de sequência. Por fim, observa-se que a busca em profundidade no grafo H possui complexidade inferior ao do grafo de alinhamento e de tempo linear $O(|V(G)| + |A(G)|)$, pois cada vértice e arco é processado uma única vez e $H \subseteq G$.

5.2.2 Heurística DAG

Dado o grafo H, utilizando o Algoritmo de Kosaraju, são identificadas as componentes fortemente conexas de H e é feita a condensação do grafo resultando em um *DAG* (*Directed Acyclic Graph*) representado pelo grafo H_{DAG}, no qual cada vértice representa uma componente fortemente conexa.

Uma vez construído o H_{DAG}, o cálculo da maior distância entre seus vértices é realizado percorrendo-os em ordem topológica. No entanto, antes de calcular as distâncias entre as componentes representadas pelos vértices de H_{DAG}, é essencial computar a distância entre os vértices que estão na mesma componente conexa.

Nesse sentido, para computar essa distância, é importante notar que, por se tratar de componentes fortemente conexas, garante-se, por definição, que qualquer vértice dentro da componente alcança todos os outros vértices da mesma componente, o que assegura a existência de caminhos entre eles.

Dessa forma, escolhemos inicialmente um vértice arbitrário u como referência. A distância de q à u, denotada por $distancia[u]$, representa o número de vértices no caminho de q a u no grafo H. A partir disso, é determinado o caminho com o maior número de vértices dentro da mesma componente que começa em u. A maneira que isso é feito é representada no **Pseudocódigo 3**.

Pseudocódigo 3 maiorCaminhoSCC

Entrada:
vértice **u**
grafo de sequência simples $\mathbf{G}(V,A)$
componente conexa **C**,
conjunto de vértices visitados **L**,
vetor **tamanhoCaminho**
vetor **proximoVertice**
Resultado: Encontra o maior caminho que começa em u na componente C
$L \leftarrow L \cup \{u\}$
$tamanhoCaminho[u] \leftarrow 0$
$proximoVertice[u] \leftarrow u$
para cada vértice v adjacente a u em G **faça**
  se $u, v \in C$ **então**
    se $v \notin L$ **então**
      maiorCaminhoSSG$(v, G, C, L, tamanhoCaminho)$
    fim se
    se $tamanhoCaminho[v] > tamanhoCaminho[u]$ **então**
      $tamanhoCaminho[u] \leftarrow tamanhoCaminho[v]$
      $proximoVertice[u] \leftarrow v$
    fim se
  fim se
fim para
$tamanhoCaminho[u] \leftarrow tamanhoCaminho[u] + 1$

Após a execução do **Pseudocódigo 3**, é necessário propagar a distância do vértice u para os demais da mesma componente conexa. Assim, a componente é processada novamente priorizando primeiro os vértices que estão contidos no caminho encontrado pelo **Pseudocódigo 3**. A maneira que isso é feito é representado no **Pseudocódigo 4**.

Com esse processo estabelecido, a escolha do vértice de referência u para cada componente conexa é feita de duas formas. Na primeira componente conexa da ordem topológica o vértice u é o próprio vértice q e $distancia[u] = 1$, já que um caminho de um vértice para ele mesmo tem tamanho 1. Já para as demais componentes conexas, a escolha é baseada na distância de q até os vértices da própria componente.

Com isso temos que, para uma componente conexa C, é necessário encontrar para cada $e \in C$ o maior caminho c tal que $c[1] = q$, $c[|c|] = e$ e $c[i] \notin C$ para todo $1 < i < |c|$. Dessa forma, é fácil notar que, é preciso apenas encontrar os vértices $d \notin C$ tal que $(d, e) \subseteq A(H)$, já que a diferença do maior caminho de q até d é a adição do vértice e no final do caminho. Isso pode ser feito criando o grafo H_{REV}, que é o próprio grafo H mas com os vértices dos arcos invertidos. Com essa informação, o vértice $e \in C$ que apresentar o maior caminho c é escolhido como o vértice de referência u e portanto $distancia[u] = |c|$.

Dessa forma, esse procedimento é representado no **Pseudocódigo 5**.

Pseudocódigo 4 atualizaDistanciaSCC

Entrada: vértice **u**, grafo de sequência **H**, componente conexa de **C**, conjunto de vértices com vetor *distancia* atualizado $\mathbf{L_{dist}}$, vetor **tamanhoCaminho**, vetor **distancia**, vetor **proximoVertice**
Resultado: Atualiza as distâncias dos vértices da componente conexa de u
se $proximoVertice[u] \neq u$ e $proximoVertice[u] \notin L_{dist}$ **então**
    $x \leftarrow proximoVertice[u]$
    $L_{dist} \leftarrow L_{dist} \cup \{x\}$
    $distancia[x] \leftarrow distancia[u] + 1$
    atualizaDistanciaSSG$(x, G, C, L_{dist}, tamanhoCaminho, distancia, proximoVertice)$
fim se
para cada vértice v adjacente a u em G **faça**
    se $u, v \in C$ **então**
        se $v \notin L_{dist}$ **então**
            $L_{dist} \leftarrow L_{dist} \cup \{v\}$
            $distancia[v] \leftarrow distancia[u] + 1$
            atualizaDistanciaSSG$(v, G, C, L_{dist}, tamanhoCaminho, distancia, proximoVertice)$
        fim se
    fim se
fim para

Pseudocódigo 5 encontraVerticeReferencia

Entrada: grafo de sequência $\mathbf{H_{REV}}$, componente conexa **C**, vetor **distancia**
Resultado: Encontra e computa a distância do vértice escolhido como referência da componente C
$melhorVertice \leftarrow -1$
$melhorDistancia \leftarrow -1$
para cada vértice e em C **faça**
    para cada vértice d adjacente a e em H_{REV} **faça**
        se $d \notin C$ **então**
            se $melhorVertice = -1$ **então**
                $melhorVertice \leftarrow e$
                $melhorDistancia \leftarrow distancia[d]$
            fim se
            se $distancia[d] > melhorDistancia$ **então**
                $melhorVertice \leftarrow e$
                $melhorDistancia \leftarrow distancia[d]$
            fim se
        fim se
    fim para
fim para
$distancia[melhorVertice] \leftarrow melhorDistancia + 1$
devolva $melhorVertice$

Com esses outros processos estabelecidos é possível construir a heurística *DAG* proposta. Então, dado o grafo H e a sequência de entrada s, é, primeiro, identificado as *SCC*s e feito a condensação de H, criando o grafo H_{DAG}. Após, é processado em ordem topoló-

gica os vértices, que representam componentes conexas, de H_{DAG} e para cada componente conexa é escolhido um vértice de referência u, encontrado o maior caminho a partir dele dentro da componente conexa e propagado a $distancia[u]$ para os vértices de mesma componente. Ao fim desse processo terá sido computada a distância de q a todos os vértices do grafo H, sendo possível recuperar o maior caminho que começa em q trivialmente. Por fim, é computada a distância da sequência induzida pelo caminho e a sequência de entrada s sob a distância de edição. A maneira que isso é feito está representada no **Pseudocódigo 6**.

Pseudocódigo 6 Heurística *DAG*

Entrada: grafo de sequência $\mathbf{G}(V,A)$, sequência **s**
Resultado: distância de edição d
// Inicializa estruturas e vetores auxiliares //
$G' \leftarrow$ Grafo de Alinhamento (s,G)
$w \leftarrow$ Caminho mínimo (G')
$p \leftarrow$ passeio correspondente em G dado w
$H \leftarrow$ grafo induzido pelo passeio p
$H_{REV} \leftarrow$ grafo H com arcos invertidos
$C_H \leftarrow$ identifica *SCC*s de H em ordem topológica
$primeiraComponente \leftarrow Verdadeiro$
para cada componente C em ordem topológica de C_H **faça**
  se C não foi visitado **então**
    se $primeiraComponente = Verdadeiro$ **então**
      $u \leftarrow p[1]$
      $distancia[u] \leftarrow 1$
      $primeiraComponente \leftarrow Falso$
    senão
      $u \leftarrow$ **encontraVerticeReferencia**$(H_{REV}, C, distancia)$
    fim se
    $L_{dist} \leftarrow L_{dist} \cup \{u\}$
    maiorCaminhoSCC$(u,H,C,L,tamanhoCaminho,proximoVertice)$
    atualizaDistanciaSCC$(u,H,C,L_{dist},tamanhoCaminho,distancia,proximoVertice)$
    visita C
  fim se
fim para
$c \leftarrow$ encontra maior caminho de H
$s' \leftarrow$ sequência induzida por c
devolva Distancia de Edição(s,s')

A complexidade da heurística *DAG* é bastante semelhante à heurística *DFS* sendo, também, predominantemente determinada pela execução do algoritmo de Dijkstra no grafo de alinhamento (G'), resultando na mesma complexidade total $O((|s| \times (|V(G)| + |A(G)|) \times log(|s| \times |V(G)|))$, onde $|s|$ é o comprimento da sequência de entrada e G é o grafo de sequência. Ademais, observa-se que a execução das demais funções possuem complexidade inferior. Por fim, o processamento utilizando o grafo H é de tempo linear com complexidade $O(|V(G)| + |A(G)|)$ já que cada vértice e arco é processado um número constante de vezes e $H \subseteq G$.

6 Resultados

A fim de avaliar as heurísticas quanto à exatidão dos resultados, ambas foram implementadas em C++ (https://github.com/jvdss/tcc-ufms) e testadas sobre casos de testes artificiais.

A geração dos grafos foi realizada de forma aleatória, mas estruturada para assegurar a conectividade dos vértices. Cada grafo tinha uma quantidade aleatória de arcos que se encontrava no intervalo $[(n-1), 5n]$, sendo n a quantidade de vértices do grafo. Ademais, os vértices foram rotulados e as sequências geradas com caracteres do alfabeto $\Sigma = \{A, C, T, G\}$. Finalmente, para cada tamanho de grafo avaliado, foram gerados 150 casos de teste distintos, visando garantir a representatividade dos resultados para cada classe de tamanho. No total foram executados 1050 casos de teste.

Na ausência de um algoritmo ótimo para solução do Problema do Mapeamento de Sequências em Grafos de Sequência, as heurísticas foram avaliadas em comparação com o custo do passeio determinado pelo caminho mínimo no grafo de alinhamento. Esse custo se compara à um limite inferior para o problema, uma vez que a restrição de se formar um caminho, sem repetir vértices e arcos, implica que o custo do caminho será igual ou superior ao custo do passeio, que pode repetir vértices e arcos.

Assim, a métrica utilizada consistiu na diferença entre a distância de edição obtida a partir da sequência induzida pela heurística *DAG* e aquela resultante da sequência induzida pelo passeio no grafo de sequência. De forma análoga, também foi considerada a diferença entre a distância de edição calculada para a sequência produzida pela heurística DFS e a distância de edição da sequência derivada do passeio no grafo. Essa com o objetivo de mensurar quanto cada abordagem está longe do ótimo, a distância de edição da sequência induzida pelo passeio. Dessa forma, temos a seguinte métrica para cada heurística:

$$\begin{aligned} \Delta_{DAG} &= C_{DAG} - C_{passeio} \\ \Delta_{DFS} &= C_{DFS} - C_{passeio} \end{aligned} \tag{2}$$

Sendo C_{DAG}, C_{DFS} e C_{paseio}, respectivamente, o custo da distância de edição da heurística *DAG*, da heurística DFS e do passeio no grafo de sequência, encontrado a partir do caminho mínimo no grafo de alinhamento.

Esta metodologia permitiu, portanto, uma avaliação sistemática do desempenho das heurísticas propostas. Ao abranger diversos tamanhos de entrada e comparar os resultados com o limite inferior, foi possível verificar a eficácia e a escalabilidade das soluções propostas.

6.1 Resultados práticos

Os resultados encontrados após a execução das heurísticas citadas podem ser vistos na Tabela 1 e na Tabela 2. Na Tabela 1, é representado o número de vértices, tamanho da sequência utilizada, quantidade de casos de teste realizado, as médias das métricas Δ_{DAG} e Δ_{DFS} e, por fim, o tempo médio de execução por caso de teste.

Tabela 1: Análise de desempenho das heurísticas por tamanho de entrada.

Número de Vértices (n)	Tamanho da Sequência	Casos de Teste	Média Δ_{DAG}	Média Δ_{DFS}	Tempo Médio de Execução (s)*
10	10	150	2,72	2,77	0,09
50	50	150	25,27	26,55	0,14
100	100	150	54,77	56,84	0,28
250	250	150	154,58	158,99	1,25
500	500	150	325,93	332,38	4,71
750	750	150	508,03	517,03	12,02
1000	1000	150	681,90	692,37	21,24

***Tempo médio de execução por caso de teste.**

Conforme os dados da Tabela 1, os valores médios de ambas as métricas, Δ_{DAG} e Δ_{DFS}, mantêm-se relativamente próximos conforme o número de vértices e o tamanho da sequência aumentam. Apesar disso, a heurística Δ_{DAG} sempre permanece em vantagem, o que fica mais evidente com o aumento do tamanho da entrada.

Ademais, é fácil notar o rápido crescimento do tempo de execução, já que ao duplicar o tamanho de entrada, por exemplo de $n = 500$ para $n = 1000$, o tempo de execução aumenta em mais de 4 vezes, indo de $4.71s$ para $21.24s$. Isso se justifica pela alta complexidade de construção do grafo de alinhamento, já que ao realizar as $|s|$ cópias, sendo $|s|$ o tamanho da sequência de entrada, do grafo de sequência G, a complexidade de tempo se torna quadrática. Isso, também, é facilmente notado pelo crescimento acelerado da curva do tempo de execução ilustrado na Figura 8.

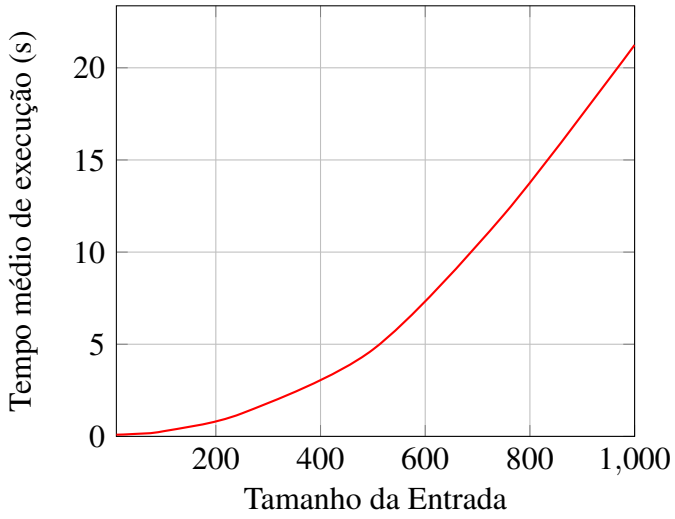

Figura 8: Curva de complexidade das heurísticas propostas.

Essa complexidade cria uma barreira considerável quando lidamos com versões ampliadas do problema, evidenciando a necessidade de maiores otimizações. Acredita-se que maiores otimizações na complexidade do tempo de execução sejam um passo vital para garantir a escalabilidade e a viabilidade da heurística em cenários de larga escala.

Para complementar a análise das médias da Tabela 1, também foi realizada uma com-

paração direta das abordagens. Para cada caso de teste executado, foi comparado qual heurística ficou mais próximo do limite inferior, o custo do passeio. Esses resultados foram classificados da seguinte forma:

- **Vitória:** $\Delta_{DAG} < \Delta_{DFS}$
- **Empate:** $\Delta_{DAG} = \Delta_{DFS}$
- **Derrota:** $\Delta_{DAG} > \Delta_{DFS}$

Os resultados encontrados são ilustrados na Tabela 2, que representa o número de vértices, quantidade de casos de teste, quantidade de vitórias, empates e derrotas, respectivamente.

Tabela 2: Resultados da comparação direta entre as heurísticas *DAG* e *DFS*.

Número de Vértices (n)	**Casos de Teste**	$\Delta_{DAG} < \Delta_{DFS}$ ***(Vitória)***	$\Delta_{DAG} = \Delta_{DFS}$ ***(Empate)***	$\Delta_{DAG} > \Delta_{DFS}$ ***(Derrota)***
10	150	9	138	3
50	150	75	75	0
100	150	83	66	1
250	150	106	44	0
500	150	117	33	0
750	150	125	25	0
1000	150	131	19	0

Conforme os dados da Tabela 2, se afirma a eficácia da heurística *DAG*, cuja vantagem sobre a abordagem de *DFS* apresenta a mesma correlação positiva com o aumento do tamanho da entrada já vista anteriormente. Essa visualização fica ainda mais evidente na Figura 9.

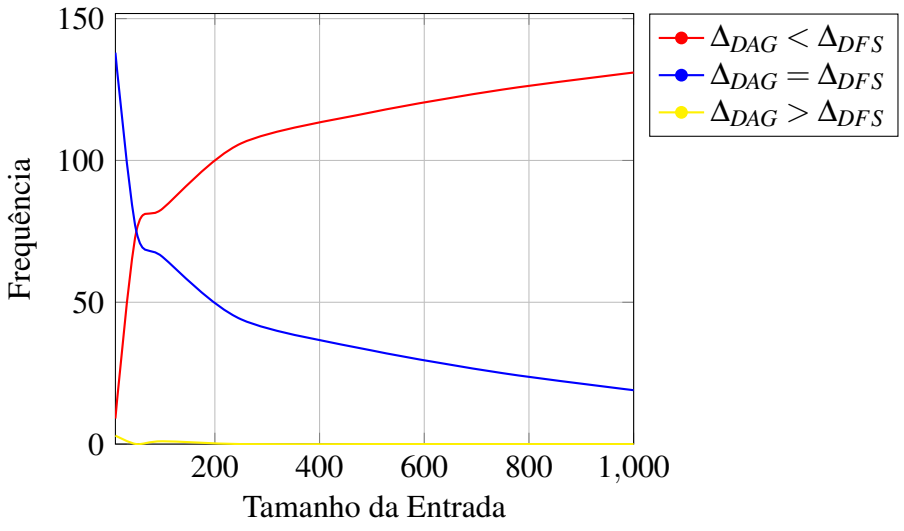

Figura 9: Gráfico linha comparando os resultados das heurísticas pelo tamanho da entrada.

Também se observa que enquanto o número de vitórias da heurística aumenta com o tamanho de entrada, o número de empates diminui. Acredita-se que o alto número de empates em grafos menores ocorra quando a própria estrutura do grafo faz com que o passeio ótimo já constitua um caminho. Nesses casos, onde há poucos caminhos possíveis, ambas as abordagens se tornam equivalentes, apresentando o mesmo resultado. Por fim, a quantidade de derrotas não apresentou significância estatística, logo, foi considerada irrelevante para a análise geral de desempenho.

7 Conclusão

Este trabalho desenvolveu e avaliou duas heurísticas em C++ para abordar o Problema do Mapeamento de Sequências em Grafos de Sequência com base na distância de edição. A eficácia e a escalabilidade da solução foram validadas utilizando uma metodologia de avaliação que empregou 1050 casos de teste artificiais com grafos e sequências variando de 10 a 1000 vértices e símbolos, utilizando um alfabeto $\Sigma = \{A, C, T, G\}$.

A análise das heurísticas foi realizada em comparação com um limite inferior, sendo esse o custo do passeio no grafo de sequência induzido a partir do caminho mínimo encontrado pelo Algoritmo de Dijkstra no grafo de alinhamento. Os resultados demonstraram que a heurística *DAG* proposta obteve distâncias consistentemente melhores aos da abordagem DFS, indicando uma maior capacidade e eficácia da heurística.

Apesar da robustez dos resultados, a análise de desempenho revelou uma limitação crítica em termos de escalabilidade computacional. O tempo médio de execução encontrado se mostrou de complexidade quadrático, criando uma barreira considerável para a aplicação da heurística em cenários de larga escala.

Para trabalhos futuros, a exploração de otimizações algorítmicas que reduzam a complexidade de tempo ou adaptações de outros algoritmos serão passos essenciais para superar o complexidade de desempenho encontrado no presente trabalho. Ademais, a integração de técnicas de aprendizado de máquina para aprimorar a previsão de caminhos pode melhorar significativamente a precisão e eficácia dos resultados encontrados. Por fim, a validação da heurística em conjuntos de dados biológicos reais será de grande relevância para validar sua utilidade prática e robustez fora dos cenários de teste artificiais.

Referências

[Amir et al., 1997] Amir, A., Lewenstein, M., and Lewenstein, N. (1997). Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99.

[Braga et al., 2000] Braga, M. D. V. et al. (2000). *Grafos de sequências de DNA*. UNICAMP.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160.

[Jain et al., 2020] Jain, C., Misra, S., Zhang, K., and Paten, B. (2020). On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, pages 707–710.

[Limasset et al., 2016] Limasset, A., Holub, J., Flot, J.-F., and Peterlongo, P. (2016). Read mapping on de bruijn graphs. *BMC Bioinformatics*, 17(1):237.

[Rautiainen and Marschall, 2017] Rautiainen, M. and Marschall, T. (2017). Graph alignment and querying using genome graphs. *Nature Communications*, 8:1–10.

[Sharir, 1981] Sharir, M. (1981). A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72.