A Heuristic to the Sequence Graph Alignment Problem under the Hamming Distance

– Trabalho de Conclusão de Curso –

Raphaella B. Jacques[1], **Said S. Adi** [1]

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79070-900 – Campo Grande – MS – Brazil

raphaella.jacques@ufms.br, said.sadique@ufms.br

Abstract. *A relevant problem in Computational Biology is mapping and comparing sequences, often using a high-quality reference sequence. However, such reference sequences are often biased, representing only a subset of all possibilities. To address this, multiple sequences are represented using more robust structures, such as sequence graphs, onto which sequences can be mapped. More formally, given as input a sequence s and a sequence graph G, the goal is to find a path in G that induces a sequence as similar as possible to s. This definition leads to the Sequence Graph Alignment Problem (SGAP). In this study, we propose a heuristic for the SGAP, under the Hamming Distance, designed to solve a variant of the problem where the goal is to find a path in the sequence graph G. The proposed heuristic demonstrated efficiency and yielded promising results on an artificial data set.*

Resumo. *Um problema relevante em Biologia Computacional é o mapeamento e a comparação de sequências, frequentemente utilizando uma sequência de referência de alta qualidade. Contudo, tais sequências de referência muitas vezes são enviesadas, representando apenas um subconjunto das possibilidades existentes. Para resolver esse problema, múltiplas sequências são representadas por meio de estruturas mais robustas, como os grafos de sequência, nos quais as sequências podem ser mapeadas. De forma mais formal, dado uma sequência s de entrada e um grafo de sequência G, o objetivo é encontrar um caminho em G que induza uma sequência o mais similar possível a s. Esta definição leva ao Problema de Alinhamento de Grafos de Sequência (SGAP), sob a Distância de Hamming. Neste estudo, propomos uma heurística para o SGAP, projetada para resolver uma variante do problema, onde o objetivo é encontrar um caminho no grafo de sequência G. A heurística proposta demonstrou eficiência e obteve resultados promissores em um conjunto de dados artificial.*

1. Introduction

In Computer Science Theory, a problem is considered intractable when it cannot be solved in polynomial time. To handle such problems, heuristics are often used, providing approximate solutions efficiently. In Computational Biology, where many problems share these characteristics, heuristics play a crucial role in delivering sufficiently good solutions within polynomial time, even if they are not always optimal. A common problem in this

field is to map one sequence onto another for comparison. This process typically uses a reference sequence; however because it represents only a restricted subset of possible sequences, it is biased. For example, the Human Genome Project was based on a restricted pool of human genomes, and comparing your genome with the resulting reference could lead to unexpected results. One way to address this problem is to align the sequence in a graph that represents multiple sequences simultaneously; therefore, it is increasingly important to use graphs to provide a more comprehensive representation nowadays.

A sequence graph is a graph in which each vertex is labeled with one or more characters. Among the types of sequence graph, the De Bruijn graph [De Bruijn 1946] and the simple sequence graph are particularly noteworthy, with our study focusing primarily on the latter. In a simple sequence graph, vertices are labeled with a single character, connected by edges. In contrast, a De Bruijn graph of order k has vertices labeled by a string of character of length k, with an edge between two vertices if there is an overlap of $k-1$ characters between the suffix of the first vertex and the prefix of the second. In these structures, a walk w spells a sequence s' corresponding to the label of vertices connected by arcs in w. Given a sequence graph G and a sequence s, the goal of the Sequence Graph Alignment Problem (SGAP) is to find a path c in G that induces a sequence s' as close as possible to s.

Finding a path c that induces a sequence s' as close as possible to a target sequence s in a De Bruijn graph is known to be a NP-hard problem [Limasset et al. 2016], making exact solutions impractical as the size of the graph increases. In simple sequence graphs, this problem is also computationally challenging, as we will conjecture later in this study. Given the NP-hardness of this problem, we developed a heuristic inspired by Marschall's work [Rautiainen and Marschall 2017] on sequence-to-graph alignment. In their study, Rautiainen and Marschall proposed a polynomial time algorithm to a variant of the SGAP where the objective is to find a walk instead of a path in the sequence graph, under the edit distance. Their approach maps a sequence onto a simple sequence graph in $O(|A| \cdot m + |V| \cdot m \cdot log(|A| \cdot m))$ time. In our work, we propose an implementation of Marschall's algorithm, adapting it to address the SGAP under the hamming distance.

This work is organized as follows: Section 2 presents preliminary concepts and essential computational definitions for understanding the Sequence Graph Alignment Problem (SGAP). In Section 3, we carried out a bibliographical review of the problem addressed in this study. In Section 4, we investigate the possibility of converting De Bruijn graphs into simple sequence graphs and analyze the structural equivalence between these two representations. In Section 5, we demonstrate the process of the proposed heuristic, outlining its functionality and methodology. Section 6 presents the results, accompanied by tables and graphs, to illustrate the heuristic's efficiency and accuracy. Finally, Section 7 provides the conclusion of this research, along with future research directions, discussing potential improvements and the broader applications of SGAP in biological contexts.

2. Preliminary Concepts

For a clearer understanding of this study, some preliminary concepts are introduced in this section. In Section 2.1, we provide an overview of sequences and related concepts; in Section 2.2, we introduce the concept of Hamming distance; and in Section 2.3, we discuss fundamental graph concepts.

2.1. Sequences

An **alphabet** Σ is a finite set of **characters** (or **symbols**). A **sequence** or **string** s of Σ is any group of 0 or more characters denoted by $s = s[1]s[2]..s[n]$, with $s[i]$ being the i-th symbol of s, and $1 \leq i \leq n$. The **size** of s, represented by $|s|$, indicates the quantity of symbols is s. For a $k \in \mathbb{N}$, the set Σ^k represents all sequences of length k that can be formed using the symbols in Σ. The set $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$ thus includes all possible sequences over Σ of any length.

Given a sequence $s = s[1]s[2] \dots s[n]$, a **substring** of s is any sequence of consecutive characters of s. A substring $s[i]s[i+1] \dots s[j]$ with $i \geq 1$ and $j \leq |s|$ is denoted by $s[i, j]$. In other words, a substring of s is any part of s that can be obtained by selecting one or more characters in sequence, without rearranging their order. A substring of length k of a sequence s is called a k-mer of s. For a given i where $1 \leq i \leq |s|$, the substring $x = s[1, i]$ ($s[i, |s|]$) is called a **prefix** (**suffix**) of s.

Given two distinct sequences s and t, where $s = uw$ (with u as a prefix and w as a suffix of s) and $t = wv$ (with w as a prefix and v as a suffix of t), the substring w common to both s and t is called an **overlap** of these sequences. For two sequences s and t, we denote their concatenation by st, which consists of joining these two sequences into a single continuous sequence by placing all characters of t, in their original order, at the end of s.

For example, consider $s =$ ABBCA and $t =$ BCABB, two sequences built over the alphabet $\Sigma = \{A, B, C\}$ and $k = 2$. Thus, $|s| = 5$, $s[1] = A$, $s[4] = C$, $\Sigma^k = \{$AA, AB, AC, BA, BB, BC, CA, CB, CC$\}$, the substring $s[1, 3] =$ ABB is a prefix of s, $s[3, 5] =$ BCA is a suffix of s, $w =$ BCA is common to both strings, so it's an overlap of s and t of length 3, and $s\underline{t} =$ ABBCA$\underline{BCABB}$ is the concatenation of s and t.

2.2. Distance

The **distance** between two sequences s and t is a non-negative value intended to measure how similar the two sequences are. It can represent, for example, the minimum number of **editing operations** (deletion, substitution and insertion of a character) described by the Russian mathematician Vladimir Levenshtein [Levenshtein 1966], required for make one sequence identical to the other. Among distance measures, the Hamming distance is particularly relevant to our study.

The **Hamming distance**, introduced by the American mathematician Richard Hamming [Hamming 1950], considers only substitutions as the only allowed editing operation to transform s into t. This implies that the **calculation** is only possible when both sequences have the same length. Formally, for two sequences s and t with length n, the Hamming distance $d_h(s, t)$ between the two sequences is defined as

$$d_h(s, t) = \sum_{i=1}^{n} f(s[i], t[i]), \tag{1}$$

where $f(s[i], t[i]) = 1$ if $s[i] \neq t[i]$ and 0 otherwise. This measure represents the substitution cost for aligning $s[i]$ with $t[i]$, and vice versa. This measure can be computed in $O(n)$ time. For example, consider the sequences $s = ACAGT$ and $t = GTACT$. Table

1 shows the differences between s and t, which serves as the basis for the calculation of the Hamming distance.

A	C	A	G	T
G	T	A	C	T
1	1	0	1	0

Table 1. Example of calculating the Hamming distance between the same length sequences $s = ACAGT$ and $t = GTACT$. The Hamming distance $d_h(s,t) = 3$ due to differing characters at positions $s[1] \neq t[1]$, $s[2] \neq t[2]$ and $s[4] \neq t[4]$, while the other characters are identical.

2.3. Graphs

A graph G is defined as an ordered pair (V, A), composed of two sets: V, representing the elements called **vertices** (of the graph), and A, representing the elements called **edges** or **arcs** (of the graph). Each edge of a graph corresponds to a non-ordered pair of vertices, while each arc corresponds to an ordered pair of vertices of a graph. When the graphs have arcs, they are called **directed** graphs or **digraphs**. However, if it contains only edges, they are called **undirected** graphs. For example,
$G" = (\{v_1\, v_2, v_2, v_4\}, \{(v_1, v_2), (v_2, v_3), (v_4, v_1), (v_2, v_4), (v_4, v_2)\}$ is a directed graph, while $G = (\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_2.v_3\}, \{v_2, v_4\}, \{v_4, v_1\}\})$ is an undirected graph.

Graphs are called this because they can be represented graphically, with circles as vertices and line segments as edges. For digraphs, the line segments representing the arcs have arrows indicating their direction. Given two vertices u and v in an undirected graph G, they are called **adjacent** if there is an edge between them. In a directed graph D, u and v are adjacent if there is an arc from u to v $(u \to v)$. In this case, u is adjacent to v, while v could not be adjacent to u. For a vertex v in a undirected graph G, the **degree** of v $(deg(v))$ is the number of edges incident to it. In a directed graph D, the **in-degree** of v $(deg^-(v))$ is the number of arcs coming to the vertex, and the **out-degree** of v $(deg^+(v))$ is the number of arcs coming out from the vertex. A graphical representation of an undirected graph G and a directed graph D can be seen in Figure 1.

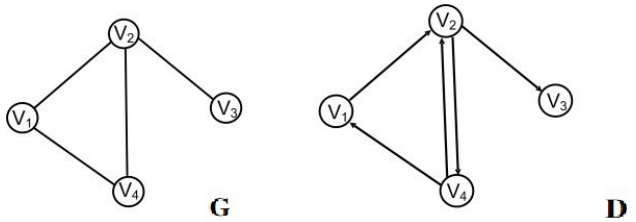

Figure 1. In G, the vertices v_1 and v_2 are adjacent while in D v_1 is adjacent to v_2 but not vice versa. In D the in-degree of v_4 is $deg^-(v_4) = 1$ and the out-degree is $deg^+(v_4) = 2$. In G the degree of v_4 is $deg(v_4) = 2$.

Each edge $\{u, v\}$ of an undirected graph G (or arc (u, v) of a directed graph D) can be associated with a real number through a function $c : A \to \mathbb{R}$ (or $c((u, v)) \to \mathbb{R}$). This number associated to an edge (or arc) of a graph is called the **cost** (or **weight**) of the edge (or arc). A weighted graph is a graph in which each edge (or arc) is assigned a cost (weight). In a graphical representation, these costs are displayed together with their corresponding edges (or arcs), as demonstrated in the Figure 2.

2.3.1. Graph paths

From this point onward, we will refer to digraphs simply as graphs.

Given a graph $G = (V, A)$, a **walk** in G is a sequence of vertices $p = v_1, v_2, \ldots, v_k$, such that for each pair of vertices v_i, v_{i+1} in p there is an arc $(v_i, v_{i+1}) \in A$. The length of a walk is defined as the number of arcs in the sequence. A walk is **closed** if it's first vertex coincides with the last one. A **cycle** is a closed walk without repeating arcs. Graphs without cycles as called **acyclic** graphs - DAG (Directed Acyclic Graph). A **path** in G is a walk without repeating vertices.

Given a weighted graph G, the **cost** of the path $p = v_1, \ldots, v_n$ in G is $c(p)$, the sum of the costs of the arcs of p ($c(p) = \sum_{j=1}^{n-1} c(v_j, v_{j+1})$). For graphs with cost on the arcs, the **shortest path** from u to v is the one with the minimum cost. If a path between u to v doesn't exist, the cost of the path is infinite. For example, in the graph G from the Figure 2, the shortest path from v_1 to v_8 is v_1, v_2, v_3, v_5, v_8 with length 4 and cost 10, although there is a path v_1, v_4, v_7, v_8, with length 3 but total cost of 13.

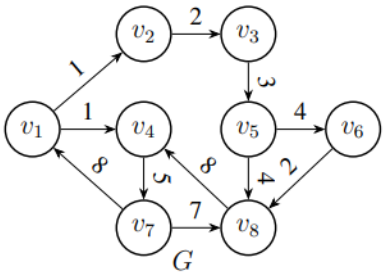

Figure 2. : Example of weighted graph , adapted from [Rocha 2024].

The Shortest Path Problem can be defined as:

Problem 1. *Given a weighted graph G and two vertices u and v in G, find the shortest path from u to v.*

This problem is well-studied in the literature and can be solved in $O(|A|+|V|log|V|)$ time using the Dijkstra's algorithm [Dijkstra 1959] where A is the set of arcs and V is the set of vertices of the input graph. Dijkstra's algorithm is used to calculate the shortest path between vertices in a graph. It begins by selecting a vertex as the source, marking it with a distance of zero and all others with an infinite distance, as well as marking them as unvisited. At each iteration, the unvisited vertex with the shortest distance to the source vertex is selected, and the distances to its neighbors are recalculated. If the new distance to each neighbor is smaller, it is updated. This process repeats until all vertices have been visited. It is important to note that the algorithm does not support arcs with negative costs.

2.3.2. Sequence Graph

A **sequence graph** is a graph $G = (V, A)$, where each vertex has an associated sequence of characters constructed over an alphabet Σ [Braga 2000]. A sequence of symbols associated with a vertex in a sequence graph is called the vertex **label**. A **simple**

sequence graph (SSG) is a graph in which each vertex is labeled with only one character [Amir et al. 1997, Rautiainen and Marschall 2017, Jain et al. 2020]. An example of a (simple) sequence graph is presented in Figure 3.

Given a walk $p = v_1, v_2, \ldots, v_n$ in a sequence graph G, the **sequence induced** by p is determined by concatenating each sequence associated with each vertex in $p = v_1, v_2, \ldots, v_n$ in the order that the vertices appear in the walk. For example, in the graph G in Figure 3, the sequence induced by $p = v_3, v_5, v_6, v_7, v_4, v_1$, of length 5, is $GTCGTA$.

Given a set of sequences $S = \{r_1, \ldots, r_m\}$ constructed over an alphabet Σ and an integer $k \geq 2$, a **De Bruijn graph** (DBG) [De Bruijn 1946] of order k is defined as a graph $G_k = (V, A)$ such that:

- $V = \{d \in \Sigma^k | d$ is a sequence of length $k(k$-mer) of $r \in S\}$;
- $A = \{(d, d') |$ the suffix of length $k-1$ of d is a prefix of $d'\}$.

Note that in a De Bruijn graph, the arcs are **induced** by the set of vertices; therefore, we can define a De Bruijn graph G_k solely by its set of vertices. For example, consider the set $S = \{$TTCTG, TGATA, TCATA$\}$ and $k = 3$. An example of a De Bruijn graph G_3 along with its induced arcs for the set S is presented in Figure 3.

Given a walk $p = v_1, v_2, \ldots, v_n$ in a De Bruijn graph, the sequence induced by p is $v_1 v_2[k] \ldots v_n[k]$, given by concatenating the k-mer v_1 with the last character of each k-mer $v_2, \ldots, v_n$. For example, in the graph G_3 in Figure 3, the sequence induced by $p = v_1, v_5, v_8, v_9$ is $TTCATA$.

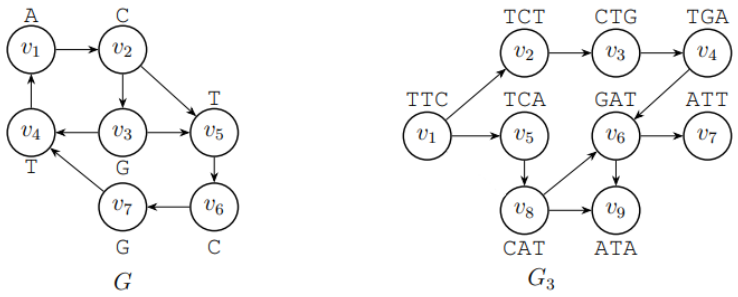

Figure 3. G represents a simple sequence graph and G_3 a De Bruijn graph with $k = 3$.

3. SGAP - Sequence Graph Alignment Problem

The Sequence Graph Alignment Problem (SGAP) models the challenge of mapping sequences onto graphs. Informally, the problem consists of finding a path in a sequence graph G that induces a sequence s' as similar as possible of another sequence s given a measure distance.

3.1. Problem Formulation and Example

Formally, SGAP is defined as follows. Given:

- A sequence graph $G = (V, A)$;
- An input sequence $s \in \Sigma^*$, of length m.

- A cost function $F : \Sigma \times \Sigma \to \mathbb{N}$, which measures the "distance" between characters.

The goal is to find a path p in G such that the induced sequence s' minimizes the total cost $d_h(s, s') = \sum_{i=1}^{m} f(s[i], s'[i])$.

Example 1: Given the DBG of Figure 4 and s = GTTCTACG, an output for the SGAP would be the path $p_1 = v_1, v_2, v_3, v_4, v_6, v_7$ that induces the sequence s_1 = GTTCGGAC, whose distance to s is $d_h(s, s_1) = 4$.

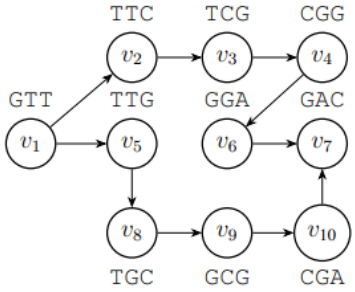

Figure 4. : Example of a De Bruijn Graph G_k , adapted from [Rocha 2024].

Example 2: Given the SSG of Figure 5 and s = ATGCTAG, an output for the SGAP would be the path $p_1 = v_1, v_4, v_2, v_3, v_5, v_8, v_7$ that induces the sequence s_1 = ATCGTAG, whose distance to s is $d_h(s, s_1) = 2$.

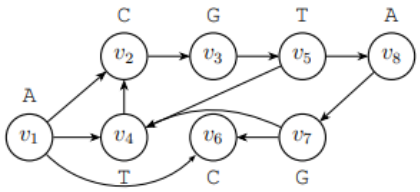

Figure 5. : Example of a Simple Sequence Graph G , adapted from [Rocha 2024].

The Sequence Graph Alignment problem was proven to be NP-complete for De Bruijn graphs by Limasset *et al.* [Limasset et al. 2016] under the Hamming Distance. In the next section, we propose that this computational complexity persists even for simple sequence graphs, further emphasizing the problem's general difficulty.

4. Difficulty of the Problem in Simple Sequence Graphs

The equivalence between DBG and SSG forms the basis for analyzing the complexity of the SGAP in the context of simple sequence graphs. The representation of a DBG, particularly as an SSG, is often treated abstractly in theoretical studies, as highlighted by Gibney [Gibney et al. 2022]. In their work, the authors use the DBG by assigning a character to each vertex, with k-mers generated through walks of length k. Building on this premise, we describe a naive algorithm for converting a DBG G_k into an SSG G, showing that both graphs induce the same set of sequences.

The conversion involves two steps:

1. for each vertex (k-mer) u in G_k, the first step consists of dividing it into k other vertices $v_{u_1}, \ldots, v_{u_k}$, where the label v_{u_i} corresponds to the i-th symbol of the k-mer of u. These new vertices form the vertex set of G.
2. the second step involves inserting arcs into G to connect its vertices. In general, for each vertex u of G_k that was divided, we insert the arcs (v_{u_1}, v_{u_2}),(v_{u_2}, v_{u_3}),...,$(v_{u_{k-1}}, v_{u_k})$. Furthermore, for each pair of adjacent vertices u and u' in G_k, we add an arc connecting the vertex v_{u_k} from the division of u to the vertex $v_{u'_k}$ from the division of u' in G.

An example of the conversion is illustrated in Figure 6. In this process, each vertex u in DBG G_k, with $k = 3$, is subdivided into k vertices in SSG G. Arcs are added to connect consecutive vertices: the first to the second, the second to the third, and so on, until the $(k-1)$-th vertex connects to the k-th vertex.

For example, the arcs $(v_{u_{11}}, v_{u_{12}})$ and $(v_{u_{12}}, v_{u_{13}})$ in G represent the subdivision of the k-mer ACA from the vertex u_1 in G_k. If a vertex u in G_k has adjacent vertices u', an arc is added between the k-th vertex of u's subdivision and the vertex representing the k-th character of each adjacency u'. For example, the arc $(v_{u_{13}}, v_{u_{23}})$ in G reflects the adjacency between the k-mers ACA and CAC in G_k.

This rule also applies to loops in G_k. In such cases, the arc from the k-th vertex of u's subdivision to the vertex representing the k-th character of the adjacency u'. In this case, the arc connects back to the k-th vertex of u's subdivision, as illustrated by the vertex $v_{u_{43}}$ in G for the k-mer AAA in G_k.

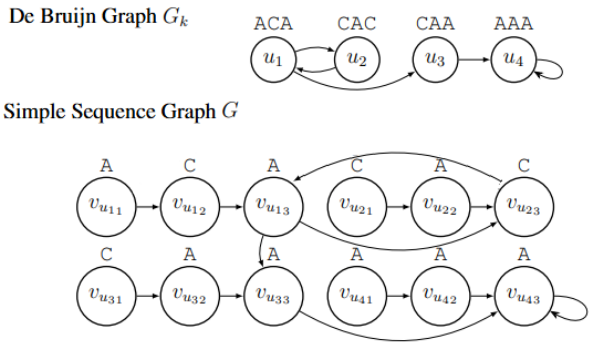

Figure 6. Example of the conversion of a DBG G_k, with $k = 3$, into an equivalent SSG G. Each vertex in G_k is subdivided into k vertices, with arcs added to represent both internal connections and adjacencies between k-mers.

Theorem 4.1. *All sequences induced in the De Bruijn graph G_k are also induced in the simple sequence graph G.*

Proof. Given a De Bruijn graph G_k and its equivalent simple sequence graph G, G contains all the k-mers of G_k, divided and correctly connected by arcs. For any walk $p = u_1, \ldots, u_n$ in G_k, there exists a walk $qp' = v_{u_{11}}, \ldots, v_{u_{1(k-1)}}, v_{u_{1k}}, v_{u_{2k}}, \ldots, v_{u_{nk}}$ in G, where $q = v_{u_{11}}, \ldots, v_{u_{1k}}$ and $p' = v_{u_{2k}}, \ldots, v_{u_{nk}}$. Since $p = \underline{u_1}, \ldots, u_n = \underline{v_{u_{11}}, \ldots, v_{u_{1k}}}, v_{u_{2k}}, \ldots, v_{u_{nk}} = qp'$, it can be observed that $q = v_{u_{11}}, \ldots, v_{u_{1k}}$ induces the same sequence as the vertex u_1, and each character is equivalently induced in $v_{u_{2k}} = u_2[k], \ldots, v_{u_{nk}} = u_n[k]$. Therefore, p and qp' induce the same sequences. □

While a formal proof that all sequences induced in the SSG are also induced in the DBG are missing, we conjecture that this holds for sequences of length $n \geq k$ in SSG that can be constructed as paths in the DBG. This conjecture is based on the structural equivalence observed in the conversion process, where each k-mer in the DBG corresponds to a valid walk in the SSG, preserving the original sequence characteristics.

This equivalence highlights that the complexity of aligning sequences in SSGs is inherited from DBGs. Since the SGAP in DBGs is known to be NP-complete, and we have demonstrated the equivalence of sequence induction between DBGs and SSGs, it follows that the analogous sequence alignment problem in SSGs is also NP-complete. This conclusion underscores the inherent computational challenges of sequence alignment in graphs and motivates the development of efficient algorithms to address this problem in the context of SSGs.

5. Our Heuristic for the Problem

5.1. Alignment graph

For the SGAP, dynamic programming is an effective approach for directed acyclic graphs (DAGs), as it leverages topological ordering to efficiently compute optimal alignments. However, this method becomes challenging to apply when the graph contains cycles, as the lack of a clear hierarchical structure complicates the alignment process. To address this, the problem can be reformulated as a shortest-path search in a specific type of graph, called here *alignment graph*. This approach is particularly useful because it enables shortest-path algorithms, such as Dijkstra's, to find optimal alignments even in graphs with cycles. The alignment graph, described below, is constructed using the query sequence, the sequence graph, and scoring parameters.

The alignment graph is a weighted directed graph that represents all possible alignments of a query sequence to a sequence graph. It is structured as a multilayer graph with m "copies" of the sequence graph, where m is the length of the query sequence. Each layer represents a character in the query sequence, and each layer contains all vertices of the sequence graph. Arcs between vertices in different layers are defined based on the alignment possibilities:

- Horizontal arcs/Insertion arcs: connect vertices v and u with a single layer to account for insertions in the query sequence; these arcs have weight 1;

- Vertical arcs/Deletions arcs: connect vertices v and u within the same layer to account for skipping characters in the query sequence; these arcs also have weight 1;

- Substitution arcs: connect vertices v and u between adjacent layers if v and u are connected in the sequence graph, and the character c at v can align with the corresponding character c' in the query sequence; these arcs have weight corresponding to the function $f(c, c')$.

With this structure, the alignment graph is a weighted directed graph in which a valid alignment corresponds to a path through the graph. At the boundaries of the graph, we define a source vertex s, which connects to all vertices in the first layer with arcs of zero weight, and a sink vertex t, connected to all vertices in the last layer with arcs of zero weight. This ensures that every possible alignment has a well-defined starting and ending

point. The alignment cost is equal to the weight of the corresponding path, as described in Figure 7.

In our problem, however, we simplify the alignment graph by excluding horizontal and vertical arcs. This is because we use the Hamming distance instead of the edit distance to score alignments, which only considers substitutions and ignores insertions or deletions. As a result, our alignment graph contains only the arcs that correspond to direct matches or mismatches between characters of the query sequence and vertices in the sequence graph. This formulation both simplifies the graph structure and focuses on the alignment properties relevant to our specific problem.

The path found in the alignment graph represents a walk in the SSG as it can be seen in Figure 7, where the path $ACGT$ in the alignment graph correspond to the walk $ACGT$ in the input graph. More generally, each layer in the alignment graph corresponds to a step in the walk, ensuring the sequence induced by the walk aligns as closely as possible to the query sequence under the Hamming distance metric.

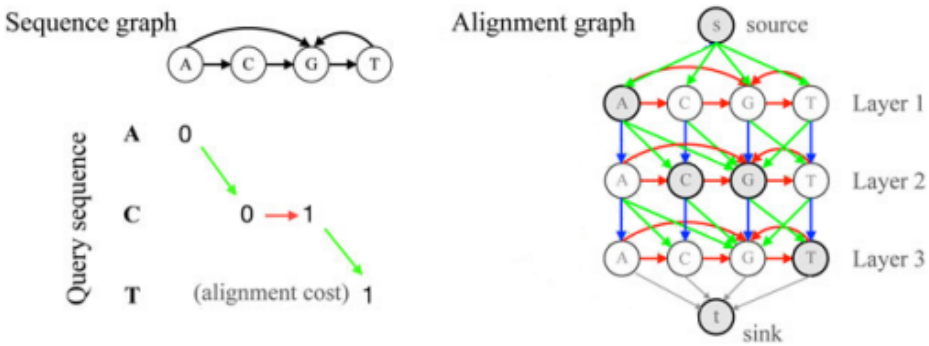

Figure 7. An example, adapted from [Jain et al. 2020], to illustrate the construction of an alignment graph (right) from a given sequence graph and a query sequence (left). Multiple colors are used to show weighted arcs of different categories in the alignment graph. The red, blue, and green arcs are weighted as insertion, deletion, and substitution costs, respectively. Optimal alignment between the query and the sequence graph is computed by finding the shortest path from source to sink vertex in the alignment graph.

5.2. Heuristic Development

Given a sequence s and a SSG G, our heuristic for solving the SGAP operates in four main steps. First, we build the alignment graph G' upon the sequence s and the SSG G; in the second step, we apply the Dijkstra's algorithm to identify a shortest path in G'. If the walk in G corresponding to the path in G' contains cycles, they are detected and removed, transforming the walk into a path. During this process, segments of the sequence s' corresponding to cycles are replaced with an arbitrary character (e.g., '-'). Finally, the Hamming Distance between s and s' is computed, excluding the arbitrary characters. A pseudocode of our heuristic is presented in Algorithm 1.

Data: sequence s and SSG G
Result: hamming distance c
$G' \leftarrow$ AlignmentGraph(s, G);
$p \leftarrow$ optimal path given by Dijkstra(G');
$p' \leftarrow$ corresponding walk in G given p;
while p' has cycles **do**
  $p' \leftarrow removeCycles(p')$;
end
$p \leftarrow p'$;
$s' \leftarrow$ sequence induced by the path p;
$c \leftarrow hammingDistance(s, s')$;
write(c);

Algorithm 1: Heuristic's Psudocode

6. Results

To evaluate the accuracy of our heuristic, it was implemented in C++ language (https://github.com/RaphaLella/TCC) and tested on 1200 artificially generated test cases. All tests were conducted using a set of sequence graphs with 2000 vertices and 3999 arcs. The graphs were randomly generated, with arcs being created to ensure the connectivity of the graph. Finally, the vertices were labeled by randomly assigning characters from a predefined alphabet.

For each graph, we generated 10 different input sequences, each of length 100, using the given alphabet. The first sequence was generated by following a valid path, ensuring a cost of 0 for this sequence. Subsequent sequences were created by introducing random modifications to the original sequence. More specifically, two characters in the sequence were modified randomly in each iteration, until a maximum of 18 changes had been made to the original sequence.

This setup allowed us to systematically evaluate the heuristic's performance across varying levels of sequence error while maintaining control over the graph's structure and connectivity.

The accuracy of the heuristic was assessed using the following metric:

$$d_h(s, s') = \sum_{i=1}^{n} f(s[i], s'[i]),$$

where $f(s[i], s'[i]) = 1$ if $s[i] \neq s'[i]$ and $s'[i] \neq$ '-' , and $f(s[i], s'[i]) = 0$ otherwise.

6.1. Test Results

The results of the execution of our heuristic on the proposed data set are shown in Tables 2, 3, 4, and 5. For each table, the first column represents the total number of tests conducted, the second column shows the number of errors introduced into the initial path, the third column provides the average Hamming distance between the input sequence and the sequence corresponding to the path identified by the heuristic, and the last column reports the average execution time (in seconds).

Table 2. First 300 tests: conducted with an alphabet of size 4.

Qty	Errors	Avg. Ham. Dist.	Avg. Time Exec. (s)
30	0	0.0	217.798
30	2	2.0	217.027
30	4	4.0	219.759
30	6	5.5	217.673
30	8	7.0	218.857
30	10	9.0	220.318
30	12	9.0	218.608
30	14	10.5	218.849
30	16	10.0	217.065
30	18	12.0	218.887

Table 3. Next 300 tests: conducted with an alphabet of size 6.

Qty	Errors	Avg. Ham. Dist.	Avg. Time Exec. (s)
30	0	0.0	218.770
30	2	2.0	216.507
30	4	4.0	216.642
30	6	6.0	215.358
30	8	8.0	216.656
30	10	10.0	218.149
30	12	12.0	218.227
30	14	13.5	219.849
30	16	15.0	217.890
30	18	16.0	218.164

Table 4. Next 300 tests: conducted with an alphabet of size 8.

Qty	Errors	Avg. Ham. Dist.	Avg. Time Exec. (s)
30	0	0.0	217.905
30	2	2.0	218.977
30	4	4.0	217.677
30	6	6.0	218.613
30	8	8.0	217.865
30	10	10.0	216.928
30	12	11.0	217.521
30	14	12.0	216.571
30	16	14.0	218.482
30	18	15.5	219.232

As shown in Tables 2, 3, 4, and 5, the paths identified by the heuristic are not significantly distant from the number of errors introduced. In many cases, the Hamming distance between the input sequence and the sequence corresponding to the path found by the heuristic aligns with the expected error count. However, interestingly, there are instances where the Hamming distance is lower than anticipated. This discrepancy can be attributed to the heuristic's ability to find alternative paths that minimize the mismatch.

Table 5. Final 300 tests: conducted with an alphabet of size 10

Qty	Errors	Avg. Ham. Dist.	Avg. Time Exec. (s)
30	0	0.0	218.770
30	2	2.0	216.507
30	4	4.0	216.642
30	6	6.0	215.358
30	8	8.0	216.656
30	10	10.0	218.149
30	12	12.0	218.227
30	14	13.5	219.849
30	16	15.0	217.890
30	18	16.0	218.164

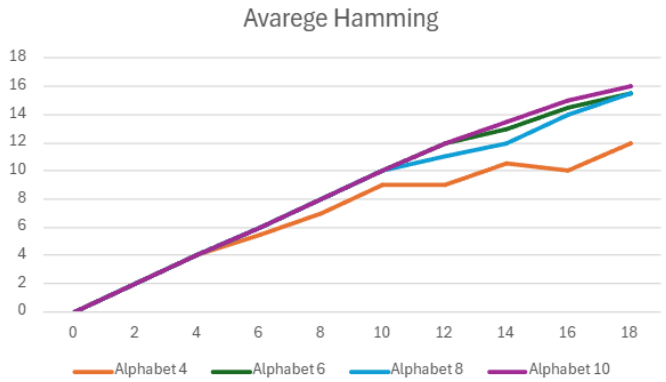

Figure 8. The axis x represents the number of error introduced in s and the axis y represents the hamming distance.

For example, consider an initial input sequence `AGTGT`, which has zero errors and corresponds to a valid path in the graph, and a modified input sequence `AGATT`, with two errors. For the second sequence, the heuristic might identify a path that aligns with the original path `AGTGT`, resulting in a Hamming distance of 2, matching the number of errors introduced in the input sequence. However, the heuristic could also find a different path, such as `AGACT`, which has a lower Hamming distance of 1 to `AGATT`, resulting in fewer mismatches compared to the errors introduced. This behavior demonstrates the adaptability of the heuristic in identifying paths close to the input sequence, even when the sequence altered by the introduction of errors has no direct path in the graph.

Figures 8 further highlight the relationship between alphabet size and the average Hamming distance. As the alphabet size increases, the average Hamming distance also tends to grow. This trend is rooted in the reduced frequency of repeated patterns within the graph for larger alphabets. When the alphabet is small, such as with a size of 4, repeated patterns are more common, increasing the likelihood of identifying paths that closely align with the input sequence. In these cases, higher alignment accuracy is achieved and the resulting Hamming distances are lower. In contrast, as the alphabet expands to sizes 6, 8, or 10, the diversity of vertex labels within the graph increases. This added diversity reduces the probability of encountering vertices with matching labels along potential paths,

making it more difficult to align closely with the input sequence. Consequently, accuracy declines, and Hamming distances grow larger for larger alphabets, illustrating the impact of graph structure and input complexity on alignment performance.

In terms of computational efficiency, the heuristic demonstrates robust and relatively fast performance under varying input conditions. The average run-time for all tests conducted was approximately 218 seconds, showing remarkable consistency irrespective of the number of errors introduced or the alphabet size. This stability underscores a key strength of the algorithm: its ability to maintain computational efficiency while processing diverse and complex input scenarios. The graph traversal mechanisms, combined with efficient cycle detection and path selection processes, contribute to this consistent and rapid performance, even as the size and complexity of the graph and input sequences grow. Such responsiveness makes the heuristic well-suited for practical applications where quick results are essential.

However, the heuristic faces a significant limitation in terms of memory usage. While precise memory consumption measurements were not taken, practical testing with larger sequences and more extensive graphs revealed a critical bottleneck. The program's memory requirements led to swapping operations, which severely impacted its efficiency in such scenarios. This issue arises due to the inherent design of the heuristic, which involves replicating the graph for each character in the input sequence. This replication process causes memory consumption to scale quadratically with respect to the sequence length and the number of vertices in the graph. Such quadratic growth becomes increasingly problematic when dealing with larger datasets, underscoring the need for further optimization. Addressing this limitation will be essential for enhancing the scalability of the heuristic and ensuring its applicability to even larger and more complex graphs in practical scenarios.

7. Conclusion

This study addressed the SGAP is shwon as NP-complete when alignment is restricted to paths in the sequence graph. To tackle this problem, we developed a heuristic using SSGs, leveraging Dijkstra's algorithm to identify the shortest walks and transforming these into paths by detecting and removing cycles.

The heuristic was implemented in C++ language and evaluated on 1200 test cases using a graph with 2000 vertices and 4999 edges. Input sequences were systematically varied, introducing errors incrementally from 0 to 18. Results showed that the heuristic is adaptable to different alphabet sizes (4, 6, 8, and 10) and error levels. Notably, as the alphabet size decreased, the accuracy improved. This is because smaller alphabets increase the probability of encountering repeated patterns in the graph, making it easier to find paths that align more closely with the input sequence. However, accuracy tends to decrease as the number of errors increases or as the alphabet size grows, which aligns with the expected trend.

The heuristic exhibited consistent runtime performance, averaging 218 seconds across all tests, regardless of the error rate or alphabet size. This consistency underscores the algorithm's efficiency and its ability to handle diverse input conditions without substantial performance degradation. However, a notable limitation lies in the memory consumption of the heuristic. Although formal memory usage measurements were not

conducted, practical tests with larger sequences and graphs revealed that swapping occurred, significantly impacting performance. This challenge arises from the heuristic's graph replication process for each character in the sequence, leading to quadratic memory consumption relative to the sequence length and the number of vertices in the graph.

Future work will explore algorithmic optimizations, such as those proposed by Jain et al. in "On the Complexity of Sequence-to-Graph Alignment" [Jain et al. 2020] which could reduce the runtime complexity to $O(|V| + m|A|)$. Additionally, adapting the algorithm to consume linear memory, inspired by Hirschberg's method in "A Linear Space Algorithm for Computing Maximal Common Subsequences" [Hirschberg 1975], would address the current memory limitations. Testing the heuristic on larger datasets and applying it to real-world biological problems will be crucial for validating its robustness and practical utility. Finally, integrating machine learning techniques to improve path prediction and enhance generalization to diverse input scenarios represents an exciting avenue for future development.

References

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

Braga, M. D. V. e. a. (2000). Grafos de sequências de DNA. UNICAMP.

De Bruijn, N. G. (1946). A combinatorial problem. In Proc. Koninklijke Nederlandse Academie van Wetenschappen, volume 49, pages 758–764.

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

Gibney, D., Thankachan, S. V., and Aluru, S. (2022). On the hardness of sequence alignment on de bruijn graphs. Journal of Computational Biology, 29(12):1377–1396.

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

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. Communications of the ACM, 18(6):341–343.

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.

Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., et al. (2001). Initial sequencing and analysis of the human genome. Nature, 409(6822):860–921.

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

Limasset, A., Holub, J., Flot, J.-F., and Peterlongo, P. (2016). Read mapping on de Bruijn graphs. BMC Bioinformatics, 17(1):237.

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

Rocha, L. B. (2024). O Problema do Mapeamento de Sequências em Grafos de De Bruijn. Tese de doutorado, Universidade Federal de Mato Grosso do Sul, Campo Grande, MS, Brasil.