On the Complexity of Sequence to Graph Alignment

Chirag Jain*[1], Haowen Zhang*[1], Yu Gao[1], and Srinivas Aluru†[1]

[1]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

Abstract

Availability of extensive genetics data across multiple individuals and populations is driving the growing importance of graph based reference representations. Aligning sequences to graphs is a fundamental operation on several types of sequence graphs (variation graphs, assembly graphs, pan-genomes, etc.) and their biological applications. Though research on sequence to graph alignments is nascent, it can draw from related work on pattern matching in hypertext. In this paper, we study sequence to graph alignment problems under Hamming and edit distance models, and linear and affine gap penalty functions, for multiple variants of the problem that allow changes in query alone, graph alone, or in both. We prove that when changes are permitted in graphs either standalone or in conjunction with changes in the query, the sequence to graph alignment problem is $\mathcal{NP}$-complete under both Hamming and edit distance models for alphabets of size ≥ 2. For the case where only changes to the sequence are permitted, we present an $O(|V| + m|E|)$ time algorithm, where m denotes the query size, and V and E denote the vertex and edge sets of the graph, respectively. Our result is generalizable to both linear and affine gap penalty functions, and improves upon the run-time complexity of existing algorithms.

Keywords: genomics, graph search, sequence alignment, approximate pattern matching, computational complexity

1 Introduction

Aligning sequences to graphs is becoming increasingly important in the context of several applications in computational biology, including variant calling (Nguyen et al., 2015; Dilthey

*These authors contributed equally to this work

†Email: aluru@cc.gatech.edu

‡A preliminary version of this article appeared in RECOMB 2019

et al., 2015; Eggertsson et al., 2017; Garrison et al., 2018), genome assembly (Antipov et al., 2015; Wick et al., 2017; Garg et al., 2018), read error-correction (Zhang et al., 2019; Salmela and Rivals, 2014; Wang et al., 2018; Limasset et al., 2019), RNA-seq data analysis (Beretta et al., 2017; Kuosmanen et al., 2018), and more recently, antimicrobial resistance profiling (Rowe and Winn, 2018). Much of this has been driven by the growing ease and ubiquity of sequencing at personal, population, and environmental-scale, leading to significant growth in availability of datasets. Graph based representations provide a natural mechanism for compact representation of related sequences and variations among them. Some of the most useful graph based data structures are de-Bruijn graphs (Pevzner et al., 2001; Iqbal et al., 2012), variation graphs (Novak et al., 2017), string graphs (Myers, 2005), partial order graphs (Lee et al., 2002) and Wheeler graphs (Gagie et al., 2017).

Decades of progress made towards designing provably good algorithms for the classic sequence to sequence alignment problems serves as the foundation for mapping tools currently used in genomics, and similar efforts are necessary for sequence to graph alignment. To address the growing list of biological applications that require aligning sequences to a graph, several heuristics (Huang et al., 2013; Liu et al., 2016; Limasset et al., 2016; Heydari et al., 2018; Garrison et al., 2018; Guo et al., 2018; Rakocevic et al., 2019) and a few provably good algorithms (Sirén et al., 2014; Rautiainen and Marschall, 2017; Kavya et al., 2019) have been developed in recent years. In addition, sequence to graph alignment has been studied much earlier in the string literature through its counterpart, approximate pattern matching to hypertext (Manber and Wu, 1992). Since then, important complexity results and algorithms have been obtained for different variants of this problem (Amir et al., 2000; Navarro, 2000; Thachuk, 2013).

Many versions of the classic sequence to sequence alignment problem were considered in the literature, e.g., different alignment modes – local/global, scoring functions – linear/affine/arbitrary gap penalty, and so on (Navarro, 2001). The list further proliferates when considering a graph-based reference. This is because the nature of the problem changes depending on whether the input graphs are cyclic or acyclic (Navarro, 2000), and whether edits are allowed in the graph, or query, or both (Amir et al., 2000). The alignment routine to directed acyclic graphs (DAGs) is often referred to as *partial order alignment* (Lee et al., 2002) in bioinformatics.

In this paper, we present new complexity results and improved algorithms for multiple variants of the sequence to graph alignment problem. The proposed results hold for general directed graphs, i.e., the graphs can contain cycles. Consider a query sequence of length m and a directed graph $G(V, E)$ with string-labeled vertices, over the alphabet Σ. We make the following contributions:

- The problem variants that allow changes to the graph labels are known to be $\mathcal{NP}$-complete, via proofs by Amir et al. (2000) that assume $|\Sigma| \geq |V|$. To date, tractability of these problems remains unknown for the case of constant sized alphabets, which is an important consideration when aligning DNA, RNA, or protein sequences to corresponding graphs. We close this knowledge gap by proving that four variants of the problem, characterized by changes to graph alone or both graph and query, under the

Hamming or edit distance models, remain $\mathcal{NP}$-complete for $|\Sigma| \geq 2$.

- Allowing changes to the query sequence alone makes the problem polynomially solvable. For graphs with character-labeled vertices, we propose an algorithm that achieves $O(|V| + m|E|)$ time bound for both linear and affine gap penalty cases, superior to the best existing algorithms (Table 1). An important attribute of the proposed algorithm is that it achieves the same time and space complexity as required for the easier problem of sequence alignment to DAGs (Lee et al., 2002), under both scoring models.

	Linear gap penalty		Affine gap penalty
	Edit distance	Arbitrary costs	
Amir et al. (2000)	$O(m(\|V\| \log \|V\| + \|E\|))$	$O(m(\|V\| \log \|V\| + \|E\|))$	-
Navarro (2000)	$O(m(\|V\| + \|E\|))$	-	-
Antipov et al. (2015)	$O(m(\|V\| \log(m\|V\|) + \|E\|))$	$O(m(\|V\| \log(m\|V\|) + \|E\|))$	-
Kavya et al. (2019)	$O(m\|V\|\|E\|)$	$O(m\|V\|\|E\|)$	$O(m\|V\|\|E\|)$
Rautiainen and Marschall (2017)	$O(\|V\| + m\|E\|)$	$O(m(\|V\| \log \|V\| + \|E\|))$	$O(m(\|V\| \log \|V\| + \|E\|))$
This work	$O(\|V\| + m\|E\|)$	$O(\|V\| + m\|E\|)$	$O(\|V\| + m\|E\|)$

Table 1: Comparison of run-time complexity achieved by different algorithms for the sequence to graph alignment problem when changes are allowed in the query sequence alone, using different scoring models. In this table, m denotes the query length, and V, E denote the vertex and edge sets in a graph with character-labeled vertices respectively.

The paper is organized as follows. We begin by defining the notations and definitions used throughout the paper (Section 2). Proofs for the hardness results are provided in Section 3. Later in Section 4, we present an improved algorithm for the polynomially solvable variant of the problem. As part of it, we also argue that significant improvement in the runtime is unlikely, and generalize the algorithm for other graph-based data structures typically used in bioinformatics. Conclusions and open problems are listed in Section 5.

2 Preliminaries

Let Σ denote an alphabet, and x and y be two strings over Σ. We use $x[i]$ to denote the i^{th} character of x, and $|x|$ to denote its length. Let $x[i, j]$ $(1 \leq i \leq j \leq |x|)$ denote $x[i]x[i+1] \dots x[j]$, the substring of x beginning at the i^{th} position and ending at the j^{th} position. Concatenation of x and y is denoted as xy. Let x^k denote string x concatenated with itself k times.

Definition 2.1. Sequence Graph: A sequence graph $G(V, E, \sigma)$ is a directed graph with vertices V and edges E. Function $\sigma : V \to \Sigma^+$ labels each vertex $v \in V$ with string $\sigma(v)$ over the alphabet Σ.

Naturally, path $p = v_i, v_{i+1}, \dots, v_j$ in $G(V, E, \sigma)$ spells the sequence $\sigma(v_i)\, \sigma(v_{i+1}) \dots \sigma(v_j)$. The above definition of sequence graph generalizes various graph data structures typically

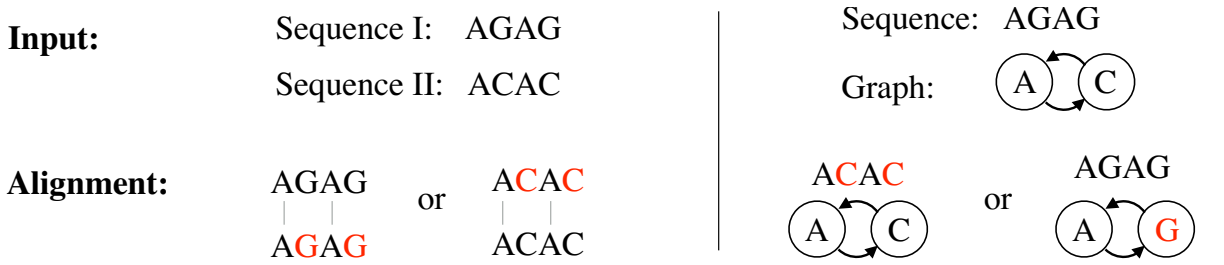

Figure 1: Asymmetry w.r.t. the location of changes in sequence to graph alignment illustrated using Hamming distance. For sequence to sequence alignment (left), two substitutions are required for a match, and can be made on either sequence. However, for sequence alignment to graph (right), two substitutions are required in the sequence, whereas just one is sufficient if made in the graph.

used in genomics (further discussed later in Section 4.5). Given a query sequence q, we seek its best matching path sequence in the graph. Alignment problems are formulated such that distance between the computed path and the query sequence is minimized, subject to a specified distance metric such as Hamming or edit distance. Typically, an alignment is scored using either a linear or an affine gap penalty function. The cost of a gap is proportional to its length, when using a linear gap penalty function. An affine gap penalty function imposes an additional constant cost to initiate a gap.

3 Complexity Analysis

3.1 Asymmetry of Edit Locations

An alignment between two sequences also specifies possible changes to the sequences (e.g. substitutions, insertions, deletions) to make them identical, with alignment distance specifying the cumulative penalty for the changes. The changes can be individually applied either to the first or the second sequence, or any combination thereof. Such a symmetry is no longer valid when aligning sequences to graphs (Amir et al., 2000). This is because alignments can occur along cyclic paths in the graph. If the label of a vertex in the graph is changed, then an alignment path visiting that vertex k times reflects the same change at k different positions in the alignment. On the other hand, a change in one position of the sequence only reflects that change in the corresponding position in the alignment. As such, optimal alignment scores vary depending on whether changes are permitted in just the sequence, just the graph, or both (see Figure 1 for an illustration). This characteristic leads to *three different problems*, with each potentially resulting in a different optimal distance.

Consider the sequence to graph alignment problem under the Hamming or edit distance metrics. For each distance metric, there are three versions of the problem depending on whether changes are allowed in query alone, graph alone, or both in the query and graph. Consider the decision versions of these problems, which ask whether there exists an alignment

with $\leq d$ modifications (substitutions or edits), as per the distance metric. Restricting substitutions or edits to the query sequence alone admits polynomial time solutions (Amir et al., 2000; Navarro, 2000; Rautiainen and Marschall, 2017). In the pioneering work of Amir et al. (2000) in the domain of string to hypertext matching, it has been proved that the other problem variants which permit changes to graph are $\mathcal{NP}$-complete. The proofs provided in their work assume an alphabet size $\geq |V|$. To date, tractability of these problems remains unknown for the case of constant sized alphabets (e.g., for DNA, RNA, or protein sequences). In what follows, we close this gap by showing that the problems remain $\mathcal{NP}$-complete for any alphabet of size at least 2.

3.2 Alignment using Hamming Distance

Theorem 3.1. The problem "Can we substitute a total of $\leq d$ characters in graph G and query q such that q will have a matching path in G?" is $\mathcal{NP}$-complete for $|\Sigma| \geq 2$.

Proof. The problem is in $\mathcal{NP}$. Given a solution, the set of substitutions can be used to obtain the corrected graph and query. Next, we can leverage any polynomial time algorithm (Amir et al., 2000; Navarro, 2000; Park and Kim, 1995) to verify if the corrected query matches a path in the corrected graph.

To show that the problem is $\mathcal{NP}$-hard, we perform a reduction using the directed Hamiltonian cycle problem. Suppose $G'(V, E)$ is a directed graph in which we seek a Hamiltonian cycle. Let $n = |V|$. We transform it into a sequence graph $G(V, E, \sigma)$ over the alphabet $\Sigma = \{\alpha, \beta\}$ by simply labeling each vertex $v \in V$ with α^n (Figure 2). Note that the graph structure remains unchanged. Next, we construct query sequence q. Let token t_i be the sequence of n characters $\alpha^{n-i-1}\beta\alpha^i$. We choose query q to be the $n^2(2n+2)$ long sequence: $(t_0 t_1 \dots t_{n-1})^{2n+2}$. We claim that a Hamiltonian cycle exists in $G'(V, E)$ if and only if q can be matched after substituting a total of $\leq n$ characters in $G(V, E, \sigma)$ and q.

Suppose there is a Hamiltonian cycle in $G'(V, E)$. We can follow the corresponding loop in $G(V, E, \sigma)$ from the first character of any vertex label. To match each token in the query q, we require one $\alpha \to \beta$ substitution per vertex. Thus, the query q matches $G(V, E, \sigma)$ after making exactly n substitutions in the graph.

Conversely, suppose the query q matches the graph $G(V, E, \sigma)$ after making $\leq n$ substitutions in the query and the graph. Consider the following substring q_{sub} of q: $t_0 t_1 \dots t_{n-1} t_0 t_1$. Note that there are $n+1$ non-overlapping instances of q_{sub} in q. Even if all the n substitutions occur in the query, at least one instance of q_{sub} must remain unchanged. As a result, q_{sub} must match to a path in the corrected $G(V, E, \sigma)$.

Case 1: q_{sub} starts matching from the first character of a vertex label. Note that the first n tokens $q_{sub}[1, n] = t_0$, $q_{sub}[n+1, 2n] = t_1, \dots$, $q_{sub}[n^2 - n + 1, n^2] = t_{n-1}$ are all unique followed by $q_{sub}[n^2+1, n^2+n] = t_0$. Therefore, this requires a Hamiltonian cycle in $G(V, E, \sigma)$. Accordingly, there is a Hamiltonian cycle in $G'(V, E)$.

Case 2: q_{sub} starts somewhere other than the starting position within a vertex label. Let $q_{sub}[k]$ $(1 < k \leq n)$ be the first character that matches at the beginning of the next vertex on the path matching q. Similar to the previous case, the following n sequences $q_{sub}[k, n+k-1]$,

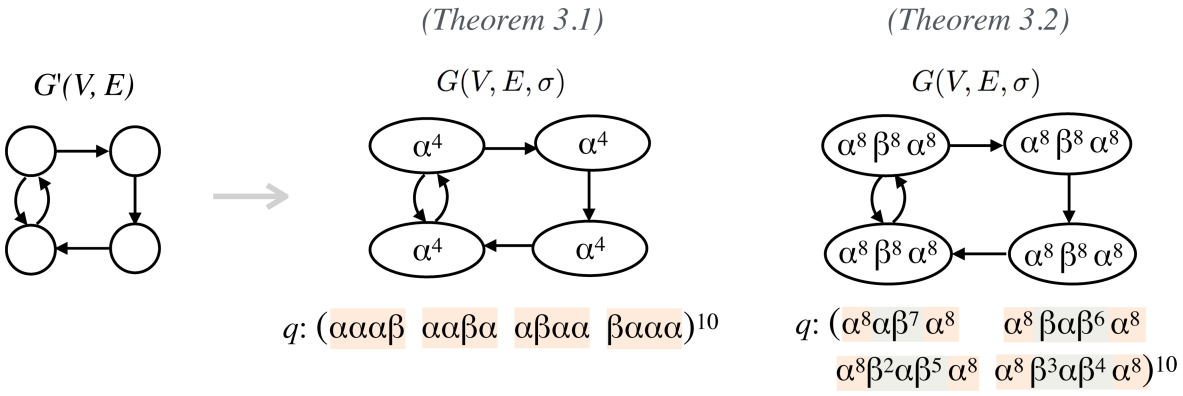

Figure 2: Illustration of the two constructs used for reductions in proofs of Theorems 3.1 and 3.2. In both cases, we are able to argue that Hamiltonian cycle exists in $G'(V, E)$ if and only if the query sequence can be matched to the sequence graph using $\leq |V|$ substitutions (Theorem 3.1) or $\leq |V|$ edits (Theorem 3.2).

$q_{sub}[n+k, 2n+k-1], \ldots, q_{sub}[n^2-n+k, n^2+k-1]$ are unique due to the spacing between β characters in q_{sub}. Therefore, the matching path must yield a Hamiltonian cycle. □

Corollary 3.1.1. The problem "Can we substitute $\leq d$ characters in graph G such that q will have a matching path in G?" is $\mathcal{NP}$-complete for $|\Sigma| \geq 2$.

Proof. The setup used in the proof of Theorem 3.1 can be trivially extended to prove the above claim. Alternatively, we can simplify the proof by using the query sequence $q = (t_0 t_1 \ldots t_{n-1})^2$ since only one instance of the substring q_{sub} in q is needed for the subsequent arguments. This is because substitutions in the query sequence are not permitted. □

Using the above two results, we conclude that Hamming-distance based decision formulations of sequence to graph alignment problems are $\mathcal{NP}$-complete when substitutions are allowed in graph labels, for $|\Sigma| \geq 2$. In fact, it can be easily shown that $|\Sigma| \geq 2$ reflects a tight bound. Using $|\Sigma| = 1$, all the problem instances can be decided in polynomial time using straightforward application of standard graph algorithms.

3.3 Alignment using Edit Distance

We next show that edit distance based decision problems that permit changes in graph labels are $\mathcal{NP}$-complete if $|\Sigma| \geq 2$. Similar to our previous claims, allowing edits in the graph makes the sequence to graph alignment problem intractable. Proofs used for Hamming distance do not apply here as edits also permit insertions and deletions. Length of vertex labels can grow or shrink using insertion and deletion edits respectively.

Theorem 3.2. The problem "Can we perform a total of $\leq d$ edits in graph G and query q so that q will match in G?" is $\mathcal{NP}$-complete for $|\Sigma| \geq 2$.

Proof. Clearly the problem is in $\mathcal{NP}$. We again use the directed Hamiltonian cycle problem for reduction. Given an instance $G'(V, E)$ of the directed Hamiltonian cycle problem, we design an instance $G(V, E, \sigma)$ using $\Sigma = \{\alpha, \beta\}$. Let $n = |V|$. Label each vertex v in V using a sequence of $6n$ characters $\alpha^{2n}\beta^{2n}\alpha^{2n}$ (Figure 2). Let token t_i be a sequence of length $6n$: $\alpha^{2n}\ \beta^i\alpha\beta^{2n-1-i}\ \alpha^{2n}$. Using such tokens, we build a query sequence q of length $6n^2(2n+2)$ as $(t_0t_1 \dots t_{n-1})^{2n+2}$. We claim that a Hamiltonian cycle exists in $G'(V, E)$ if and only if we can match the sequence q to the graph $G(V, E, \sigma)$ using $\leq n$ total edits.

If there is a Hamiltonian cycle in $G'(V, E)$, we can follow the same loop in $G(V, E, \sigma)$ to align q. The alignment requires one substitution per vertex. To prove the converse, suppose query q matches graph $G(V, E, \sigma)$ after making a total of $\leq n$ edits in q and $G(V, E, \sigma)$. Consider the substring q_{sub} of q: $t_0t_1 \dots t_{n-1}t_0$. Note that there are $n+1$ non-overlapping instances of q_{sub} in q, at least one of which must remain unchanged. Accordingly, the substring q_{sub} must match corrected $G(V, E, \sigma)$.

For the token t_i, let $k_i = \beta^i\alpha\beta^{2n-1-i}$ be its *kernel* sequence of length $2n$. It follows that $t_i = \alpha^{2n}k_i\alpha^{2n}$. We show that a kernel must be matched entirely within a vertex in $G(V, E, \sigma)$ using the following two arguments. First, since any vertex label cannot shrink from length $6n$ to $< 5n$, a kernel cannot be matched to an entire vertex after the edits. It implies that a kernel must match to ≤ 2 vertices. Second, if a kernel aligns across two vertices, $(2n-1)$ β's must be required in place of α's at the two vertex ends, thus requiring $> n$ edits. Therefore, a kernel can only be matched within a single vertex label. Finally, it is easy to observe that any vertex label after $\leq n$ edits cannot be matched to more than one kernel. When combining these arguments with the fact that all n consecutive kernels in q_{sub} are unique, we establish that the alignment path of q_{sub} must follow a Hamiltonian cycle in $G(V, E, \sigma)$. Accordingly, there is a Hamiltonian cycle in $G'(V, E)$. □

Corollary 3.2.1. The problem "Can we perform $\leq d$ edits in graph G so that q will match in G?" is $\mathcal{NP}$-complete for $|\Sigma| \geq 2$.

Proof. The setup used to prove Theorem 3.2 can be trivially extended to prove the above claim. □

It is straightforward to prove that other problem variants, e.g., with linear gap penalty or affine gap penalty scoring functions are at least as hard as the edit-distance based formulations. Therefore, the sequence to graph alignment problem remains $\mathcal{NP}$-complete even on constant sized alphabets for these classes of scoring functions also if changes are permitted in the graph. Finally, we note that all the above problems remain equally hard even for planar sequence graphs of max-degree 3, as is true for the Hamiltonian cycle problem (Plesn et al., 1979).

4 Sequence-to-Graph Alignment with Edits in Sequence

The sequence to graph alignment problem is polynomially solvable when changes are allowed in the query sequence alone (Amir et al., 2000; Navarro, 2000). Here, we improve upon the

state-of-the-art by presenting an algorithm with $O(|V| + m|E|)$ run-time. Our algorithm matches the run-time complexity achieved previously by Rautiainen and Marschall (2017) for edit distance, while improving that for linear and affine gap penalty functions. In addition, it is simpler to implement because it only uses elementary queue data structures. A prototype implementation of the algorithm is available at `https://github.com/haowenz/SGA`.

Edit distance is a special case of linear gap penalty when cost per unit length of the gap is 1, and substitution penalty is also 1. We begin by presenting our algorithm for the case of a linear gap penalty function, and later show its generalization to affine gap penalty in Section 4.3. From hereon, we assume that the sequence graph $G(V, E, \sigma)$ is a character labeled graph, i.e., $\sigma(v) \in \Sigma, v \in V$. This assumption simplifies the description of the algorithm. Note that it is straightforward to transform a graph from string-labeled form to character-labeled form, and vice versa.

4.1 Alignment Graph

In the literature on the classic sequence to sequence alignment problem, the problem is either formulated as a dynamic programming problem or an equivalent graph shortest-path problem in an appropriately constructed edge-weighted *edit graph* or *alignment graph* (Myers, 1991). However, formulating the sequence to graph alignment problem as a dynamic programming recursion, while easy for DAGs through the use of topological ordering, is difficult for general graphs due to the possibility of cycles. As it turns out, formulation as a shortest-path problem in an alignment graph is still rather convenient, even for graphs with cycles (Amir et al., 2000). The alignment graph, described below, is constructed using the given query sequence, the sequence graph, and the scoring parameters.

The alignment graph is a weighted directed graph which is constructed such that each valid alignment of the query sequence to the sequence graph corresponds to a path from source vertex s to sink vertex t in the alignment graph, and vice versa (Figure 3). The alignment cost is equal to the corresponding path distance from the source to the sink. Note that the alignment graph is a multi-layer graph containing m 'copies' of the sequence graph, one in each layer. A column of dummy vertices is required in addition to accommodate the possibility of deleting a prefix of the query sequence. Edges that emanate from a vertex are equivalent to the choices available while solving the alignment problem. A formal definition of the alignment graph follows:

Definition 4.1. Alignment graph: Given a query sequence q, a sequence graph $G(V, E, \sigma)$, linear gap penalty parameters $\Delta_{del}, \Delta_{ins}$, and a substitution cost parameter Δ_{sub}, the corresponding alignment graph is a weighted directed graph $G_a(V_a, E_a, \omega_a)$, where $V_a = \big(\{1, \ldots, m\} \times$

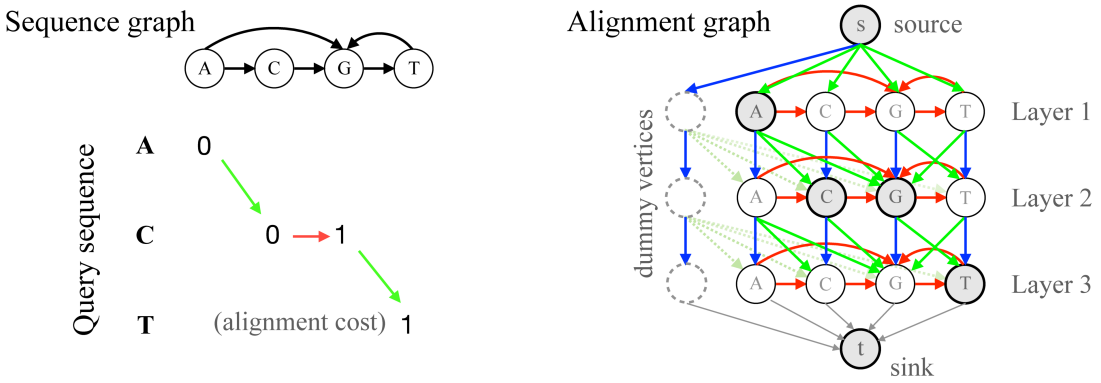

Figure 3: An example to illustrate the construction of an alignment graph (right) from a given sequence graph and a query sequence (left). Multiple colors are used to show weighted edges of different categories in the alignment graph. The red, blue and green edges are weighted as insertion, deletion 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.

$(V \cup \{\delta\})) \cup \{s, t\}$ is the vertex set, and $\omega_a : E_a \to \mathbb{R}_{\geq 0}$ is the weight function defined as

$$\omega_a(x,y) = \begin{cases} \Delta_{i,v} & x = (i-1,u), y = (i,v) \quad 1 < i \leq m \ \ \& \ \ (u,v) \in E \\ \Delta_{ins} & x = (i,u), y = (i,v) \quad 1 \leq i \leq m \ \ \& \ \ (u,v) \in E \\ \Delta_{del} & x = (i-1,v), y = (i,v) \quad 1 < i \leq m \ \ \& \ \ v \in V \cup \{\delta\} \\ \text{for source and sink vertices:} & \\ \Delta_{1,v} & x = s, y = (1,v) \quad v \in V \\ \Delta_{del} & x = s, y = (1,\delta) \\ 0 & x = (m,v), y = t \quad v \in V \cup \{\delta\} \\ \text{for dummy vertices:} & \\ \Delta_{i,v} & x = (i-1,\delta), y = (i,v) \quad 1 < i \leq m \ \ \& \ \ v \in V \end{cases}$$

Edges $(x, y) \in E_a$ are defined implicitly, as those pairs (x, y) for which ω_a is defined above. $\Delta_{i,v} = \Delta_{sub}$ if $q[i] \neq \sigma(v), v \in V$, and 0 otherwise. Δ_{sub} denotes the cost of substituting $q[i]$ with $\sigma(v)$.

Existing definitions of the alignment graph (Amir et al., 2000; Rautiainen and Marschall, 2017) did not incorporate dummy vertices, which are needed to account for the deletions correctly. Using the alignment graph, we reformulate the problem of computing an optimal alignment to that of finding the shortest path in the alignment graph. Even though the alignment graph defined by Amir et al. (2000) has minor differences, proof in their work can be easily adapted to state the following claim:

Lemma 4.1 (Amir et al. (2000))**.** Shortest distance from the source vertex s to the sink vertex t in the alignment graph $G_a(V_a, E_a, \omega_a)$ equals cost of optimal alignment between the query q and the sequence graph $G(V, E, \sigma)$.

One way of solving the above shortest path problem is to directly apply Dijkstra's algorithm (Amir et al., 2000; Antipov et al., 2015). However, it results in an $O\big(m|V|\log(m|V|) + m|E|\big)$ time algorithm. We next show how to solve this problem in $O(|V| + m|E|)$ time.

4.2 Proposed Algorithm

While searching for a shortest path from the source to the sink vertex, we compute the shortest distances from the source to intermediate vertices $V_a \backslash \{s, t\}$ in the alignment graph. An edge from a vertex in layer i is either directed to a vertex in the same layer or to a vertex in the next layer. As a result, the shortest distances to vertices in a layer can be computed once the distances for the previous layer are known. This also makes it feasible to solve for the layers 1 to m, one by one (Navarro, 2000). We use a two-stage strategy to achieve linear $O(|V| + |E|)$ run-time per layer. Before describing the details, we give an outline of the algorithm and its two stages.

Any path from the source vertex to a vertex v in a layer must extend a path ending in the previous layer using either a deletion or a substitution cost weighted edge. Afterwards, a path that ends in the same layer but not at v can be further extended to v using the insertion cost weighted edges if it results in the shortest path to the source. Roughly speaking, the first stage executes the former task, while the second takes care of the latter. The two stages together are invoked m times during the algorithm until the optimal distances are known for the last layer (Algorithm 1). Input to the first stage *InitializeDistance* is an array of the shortest distances of the vertices in previous layer sorted in non-decreasing order. This stage computes the 'tentative' distances of all vertices in the current layer because it ignores the insertion cost weighted edges during the computation. It outputs the sorted tentative distances as an input to the second stage *PropagateInsertion*. The PropagateInsertion stage returns the optimal distances of all vertices in the current layer while maintaining the sorted order for a subsequent iteration.

The following are two important aspects of our algorithm. First, we are able to maintain the sorted order of vertices by spending $O(|V|)$ time per layer during the first stage (Lemma 4.2). Secondly, we propagate insertion costs through the edges in $O(|V| + |E|)$ time per layer during the second stage by eluding the need for standard priority queue implementations (Lemmas 4.3-4.5). Both of these features exploit characteristics specific to the alignment graphs.

The InitializeDistance stage We compute tentative distances for each vertex in the current layer by using shortest distances computed for the previous layer (Algorithm 2). Because all deletion and substitution cost weighted edges are directed from the previous layer towards the current, this only requires a straightforward linear $O(|V| + |E|)$ time traversal (lines 2-8). In addition, we are required to maintain the current layer as per sorted order of distances. Note that vertices in the previous layer are already available in sorted

Algorithm 1: Algorithm for sequence to graph alignment

Result: The length of shortest path from s to t

```
PreviousLayer = [s];
s.distance = 0;
for i = 1 to m do                    /* Do the computation layer by layer */
    CurrentLayer = [(i, v_1), (i, v_2), ..., (i, v_n), (i, k)];
    x.distance = ∞ ∀x ∈ CurrentLayer;
    InitializeDistance(PreviousLayer,CurrentLayer);
    PropagateInsertion(CurrentLayer);
    PreviousLayer = CurrentLayer;
end
return Min(PreviousLayer.distance);
```

Algorithm 2: Algorithm to initialize and sort layer before insertion propagation

Result: A sorted layer *CurrentLayer* with distances initialized using *PreviousLayer*

```
Function InitializeDistance(PreviousLayer, CurrentLayer)
    foreach x ∈ PreviousLayer do
        foreach y ∈ x.neighbor & y ∈ CurrentLayer do
            if y.distance > x.distance + ω_a(x, y) then
                y.distance = x.distance + ω_a(x, y);
            end
        end
    end
    Sort(CurrentLayer);
```

order of their shortest distances from s. A vertex v in the previous layer can assign only three possible distance values ($v.distance, v.distance + \Delta_{sub}$, or $v.distance + \Delta_{del}$) to a neighbor in the current layer. By maintaining three separate lists for each of the three possibilities, we can create the three lists in sorted order and merge them in $O(|V|)$ time. The relative order of vertices in the current layer can be easily determined in linear time by tracking the positions of their distance values in the merged list. As a result, the current layer can be obtained in sorted form in $O(|V|)$ time and $O(|V|)$ space, leading to the following claim.

Lemma 4.2. Time and space complexity of the sorting procedure in Algorithm 2 is $O(|V|)$.

The PropagateInsertion Stage Note that the tentative distance computed for a vertex is sub-optimal if its shortest path from the source vertex traverses any insertion cost weighted edge in the current layer. One approach to compute optimal distance values is to process vertices in their sorted distance order (minimum first) and update the neighbor vertices, similar to Dijkstra's algorithm. When processing vertex v, the distance of its neigh-

Algorithm 3: Algorithm to propagate insertions in the same layer

Result: A sorted layer *CurrentLayer* with optimal distance values

```
1  Function PropagateInsertion(CurrentLayer)
2      x.resolved = false ∀x ∈ CurrentLayer;
3      Queue q1 = ∅, q2 = ∅;
4      q1.Enqueue(CurrentLayer);
5      CurrentLayer = [ ];
6      while q1 ≠ ∅ or q2 ≠ ∅ do
7          q_min = q1.Front() < q2.Front() ? q1 : q2;
8          x = q_min.Dequeue();
9          if x.resolved = false then
10             x.resolved = true;
11             CurrentLayer.Append(x);
12             foreach y ∈ x.neighbor & y.layer = x.layer do
13                 if y.distance > x.distance + Δ_ins then
14                     y.distance = x.distance + Δ_ins;
15                     q2.Enqueue(y);
16                 end
17             end
18         end
19     end
```

bor should be adjusted such that it is no more than $v.distance + \Delta_{ins}$. Selecting vertices with minimum scores can be achieved using a standard priority queue implementation (e.g., Fibonacci heap); however, it would require $O(|E| + |V| \log |V|)$ time per layer. A key property that can be leveraged here is that all edges being considered in this stage have uniform weights (Δ_{ins}). Therefore, we propose a simpler and faster algorithm using two First-In-First-Out queues (Algorithm 3). The first queue q_1 is initialized with sorted vertices in the current layer, and the second queue q_2 is initialized as empty (line 4). The minimum distance vertex is always dequeued from either of the two queues (line 8). As and when distance of a vertex is updated by its neighbor, it is enqueued to q_2 (line 15). Following lemmas establish the correctness and an $O(|E| + |V|)$ time bound for the PropagateInsertion stage in the algorithm.

Lemma 4.3. In each iteration at line 8, Algorithm 3 dequeues a vertex with the minimum overall distance in q_1 and q_2.

Proof. The queue q_1 always maintains its non-decreasing sorted order at the beginning of each loop iteration (line 6) in Algorithm 3 as we never enqueue new elements into q_1. We prove by contradiction that q_2 also maintains the order. Maintaining this invariant would immediately imply the above claim. Let i be the first iteration where q_2 lost the order. Clearly $i > 1$. Because i is the first such iteration, new vertices (say $y_1, y_2, \ldots, y_k$) must

have been enqueued to q_2 in the previous iteration (line 15), and the vertex (say x) which caused these additions must have been dequeued (line 8). Note that the distance of all the new vertices, the y_i's, equals $x.distance + \Delta_{ins}$. Therefore, the vertex prior to y_1 (say y_{pre}) must have a distance higher than y_1. However, this leads to a contradiction because if we consider the iteration when y_{pre} was enqueued to q_2, the distance of the vertex that caused addition of y_{pre} could not be higher than the distance of the vertex x. □

Lemma 4.4. Once a vertex is dequeued in Algorithm 3, its computed distance equals the shortest distance from the source vertex.

Proof. Lemma 4.3 establishes that Algorithm 3 processes all vertices that belong to the current layer in sorted order. Therefore, it mimics the choices made by Dijkstra's algorithm (Cormen et al., 2009). □

Lemma 4.5. Algorithm 3 uses $O(|V| + |E|)$ time and $O(|V|)$ space to compute shortest distances in a layer.

Proof. Each vertex in the current layer enqueues its updated neighbor vertices into q_2 at most once. Note that distance of a vertex can be updated at most once, therefore the maximum number of enqueue operations into q_2 is $|V|$. In addition, enqueue operations are never performed in q_1. Accordingly, the number of outer loop iterations (line 6) is bounded by $O(|V|)$. The inner loop (line 12) is executed at most once per vertex, therefore the amortized run-time of the inner loop is $O(|V| + |E|)$. □

The above claims yield an $O(m(|V|+|E|))$ time algorithm for aligning the query sequence to sequence graph. Assuming a constant alphabet, we can further tighten the bound to $O(|V| + m|E|)$ by using a simple preprocessing step suggested in Rautiainen and Marschall (2017). This step transforms the sequence graph by merging all vertices with 0 in-degree into $\leq |\Sigma|$ vertices. As a result, the preprocessing ensures that the count of vertices in the new graph is no more than $|E| + |\Sigma|$ without affecting the correctness. Summary of the above claims is presented as a following theorem:

Theorem 4.6. Algorithm 1 computes the optimal cost of aligning a query sequence of length m to graph $G(V, E, \sigma)$ in $O(|V| + m|E|)$ time and $O(|V|)$ space using a linear gap penalty cost function.

Traceback $O(|V|)$ space is required using the proposed algorithm if just the optimal alignment score is desired. This is because we are able to process the alignment graph row by row. However, an additional traceback stage is required to compute base-to-base alignments. As is typical for the classic sequence to sequence alignment, we need to save intermediate values or decisions to recover the alignment path. A naive solution to this problem requires $O(m|V|)$ space by storing the distances corresponding to all m layers in memory. It turns out that the classic linear time algorithms (Hirschberg, 1975) do not apply for directed graphs. However, an algorithm that uses sub-quadratic space can be designed using 'checkpoint-and-recalculate' strategy (Grice et al., 1997; Rautiainen and Marschall, 2017), where we only save

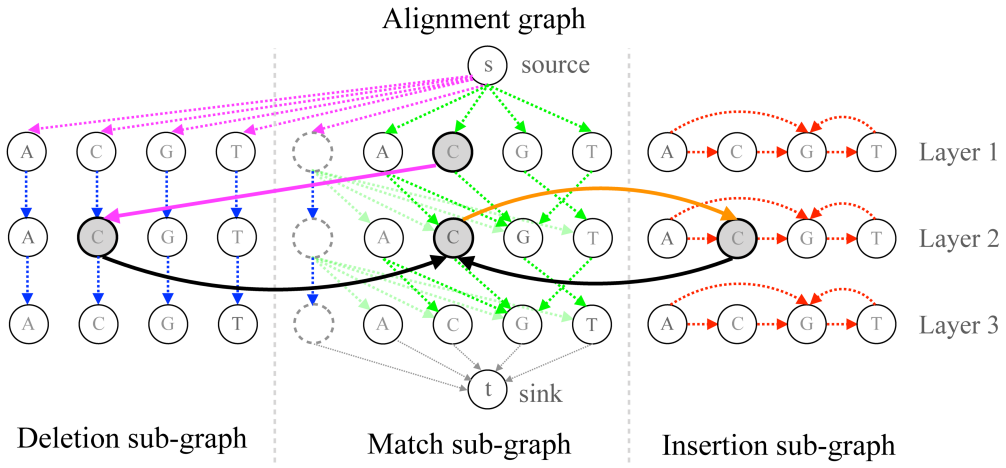

Figure 4: The above figure shows how to extend the construction of an alignment graph for sequence to graph alignment from linear gap penalty (Figure 3) to affine gap penalty. The alignment graph now contains three sub-graphs separated by the gray dashed lines. The deletion and insertion weighted edges in the alignment graph for linear gap penalty are shifted to deletion sub-graph and insertion sub-graph, respectively. Their weights are also changed to the gap extension penalty. Besides, more edges are added to connect the sub-graphs with each other. To keep the figure legible, we only use the highlighted vertices as an example to illustrate the edges required to initiate or end any gap. The weight of magenta colored edges is the sum of gap open penalty and gap extension penalty, the weight of orange colored edges is the gap open penalty, and the weight of the black colored edges is 0.

every $\sqrt{m}th$ layer, resulting in an $O(\sqrt{m}|V|)$ space algorithm without incurring additional an asymptotic cost in time. Space requirement can be further reduced, but at the cost of increased time complexity (Grice et al., 1997).

4.3 Generalization to Affine Gap Penalty

In the dynamic programming algorithm for sequence to sequence alignment, affine gap penalty functions are typically supported by using three scoring matrices instead of just one (Gotoh, 1982). Similarly, the alignment graph can be extended to contain three sub-graphs with substitution, deletion, and insertion cost weighted edges respectively (Rautiainen and Marschall, 2017). The edge weights are adjusted for the affine gap penalty model such that cost for opening a gap is incurred whenever a path leaves the match sub-graph to either the insertion or the deletion sub-graph (Figure 4). Lemma 4.1 continues to hold true for the alignment graph built for sequence to graph alignment using affine gap penalty.

Our previous two stage algorithm using InitilizeDistance and PropagateInsertion stages can be extended to a five stage algorithm for solving the shortest path problem in the new alignment graph. The alignment graph is still processed one level at a time such that there are m iterations in total. The five stages leverage the InitilizeDistance-based procedure four

times, and PropagateInsertion procedure once in each iteration. At a particular level in an iteration, we refer to vertices in the match, deletion, and insertion sub-graphs as match, deletion, and insertion layers, respectively.

In each iteration, we have the following five stages: 1) begin by initializing optimal distances of vertices in the deletion layer by using distances of vertices in the above match and deletion layers, 2) initialize distances of vertices in the current match layer using distances of vertices in the current deletion layer and the above match layer, 3) the distances of vertices in the current match layer are then utilized to initialize the current insertion layer, 4) resolve distances in the current insertion layer by using the insertion propagation algorithm (Algorithm 3), and 5) the distances of vertices in the current insertion layer are used to make a final update to the current match layer. It can be easily shown that distances of all vertices at the current level are now optimal.

The properties that were leveraged to design faster algorithm for linear gap penalty functions continue to hold in the new alignment graph. In particular, the sorting still requires linear time during the InitializeDistance stage, and insertion propagation is still executed over uniformly weighted edges in the insertion sub-graph. As a result, the two-stage algorithm can be extended to operate using affine gap penalty function in the same asymptotic time and space as with the linear gap penalty function.

4.4 Lower Bounds

It is natural to wonder whether there exist faster algorithms for solving the sequence to graph alignment problem. As noted by Rautiainen and Marschall (2017), the sequence to sequence alignment problem is a special case of the sequence to graph alignment problem because a sequence can be represented as a directed chain graph with character labels. As a result, existence of either $O(m^{1-\epsilon}|E|)$ or $O(m|E|^{1-\epsilon}), \epsilon > 0$ time algorithm for solving the sequence to graph alignment problem (for both acyclic or cyclic graphs) is unlikely because it would also yield a strongly sub-quadratic algorithm for solving the sequence to sequence alignment problem, further contradicting SETH (Backurs and Indyk, 2015). Notably, Equi et al. (2019) prove that exact and approximate matching to graphs are equally hard problems under SETH. An implication of this result is that the sequence to graph alignment is unlikely to have a faster 'banded alignment' solution (Myers, 1986), for the problem variant where count of edits allowed is an input parameter.

4.5 Generalization to Commonly Used Graphs

A sequence graph with character labeled vertices provides a good abstraction for solving the alignment problem on various types of graphs used in bioinformatics. Commonly used graphs can be converted into an *equivalent* sequence graph. In the context of solving the alignment problem, equivalence implies that any sequence (i.e., concatenation of vertex labels in a path) in the first graph exists if and only if it exists in the second graph.

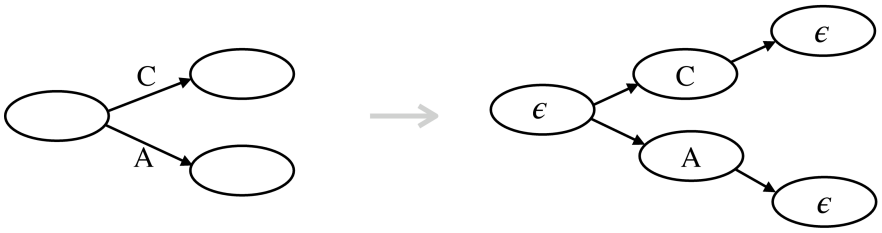

Figure 5: An example to illustrate the conversion of an edge labeled graph into the corresponding vertex labeled graph.

Directed graphs with labeled vertices Splicing graphs (Heber et al., 2002), partial order graphs (Lee et al., 2002), or variation graphs (Paten et al., 2017) with labeled vertices and directed edges are equivalent to our definition of the sequence graphs. A vertex with a string label can be split into a chain of character labeled vertices to execute our algorithm.

Directed graphs with labeled edges An alternative representation used for graphs is to put labels on edges instead of vertices (Dilthey et al., 2015). The following two ways can be used to convert an edge labeled graph $G(V, E)$ into an equivalent vertex labeled graph $G'(V', E')$. The first approach is to represent each edge in G as a vertex in G'. The out-neighbors of a vertex in G' are defined by the immediately reachable edges from its corresponding edge in G. Here $|V'| = |E|$, but $|E'| = O(|E|^2)$, though it is much more manageable in practice as $|E'| = O(|E| * d_{max})$ where d_{max} is the maximum out-degree of a vertex in G. The second approach allows conversion while restricting the graph size to within a small constant factor of the original graph. We split each edge into a pair of edges with a new vertex in the middle, where the new vertex holds the label (Figure 5). The two end points are left unlabeled or empty. In this case, $|V'| = |E| + |V|$, and $|E'| = 2|E|$. A minor challenge that remains to be addressed is how to handle the empty labels (denoted as ϵ). Our previously proposed sequence to graph alignment algorithm assumes non-empty labels in the graph. However, the alignment graph and the algorithm can be easily adapted to handle ϵ-labeled vertices using the following modifications. While defining the alignment graph, the cost of insertion weighted edges between vertices (i, u) and (i, v) $(1 \leq i \leq m)$ is modified to 0 if $\sigma(v) = \epsilon$, and the substitution cost $\Delta_{i,v}$ is set to ∞ whenever $\sigma(v) = \epsilon$. The PropagateInsertion stage of the algorithm can be adjusted to handle the 0-weighted edges without affecting the time and space complexity of the algorithm.

Regular expressions Alignment of sequences against regular expressions is useful to locate specific patterns (e.g., repeats, activation sites) in public databases, or text mining. In this context, we seek a minimum-cost set of edits that converts a sequence to match a regular expression. Myers and Miller (1989) discuss how to convert a regular expression R into an ϵ-labeled non-deterministic automaton whose size, measured in vertices or edges is linear in $|R|$ (Figure 6). This automaton, likewise, can be considered as a sequence graph, while allowing for empty labels ϵ. Additionally, the edges from source or sink vertices in the

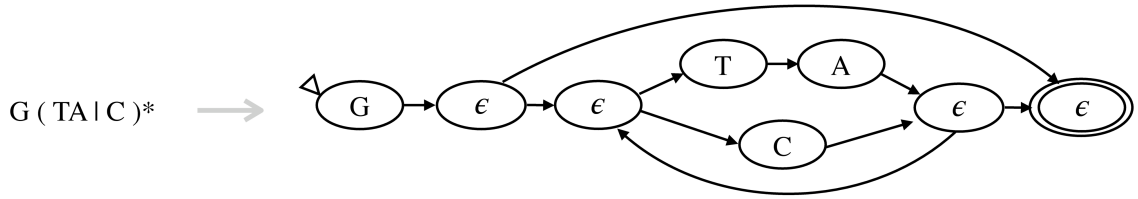

Figure 6: An example to illustrate the conversion of a regular expression into a state-labeled non-deterministic automaton using the construction rules provided by Myers and Miller (1989).

alignment graph can be adjusted to force the alignment to begin or end at selected vertices respectively. This way, it is possible to solve the sequence to regular expression alignment problem in $O(m|R|)$ time. Our quadratic time complexity in this context matches with Myers and Miller (1989), but our algorithm has the advantage of being simpler and generic.

Directed assembly graphs Alignment to assembly graphs is useful for read error correction (Salmela and Rivals, 2014; Wang et al., 2018; Zhang et al., 2019; Limasset et al., 2019) and genome assembly (Antipov et al., 2015; Wick et al., 2017; Garg et al., 2018) applications. De Bruijn graphs and overlap graphs, likewise, can be converted into a sequence graph for computing the alignments. Each vertex in a de Bruijn graph represents a k-mer, and two vertices are connected if they have $k-1$ base long suffix-prefix overlap. Transforming a de Bruijn graph to a vertex labeled sequence graph requires labeling the vertices by using their k-mer value. For all vertices with in-degree 0, we split the k-mer string into a chain of k characters, whereas for all vertices with in-degree ≥ 1, we define the k^{th} character of its k-mer as its label (Figure 7). This procedure remains correct even in the presence of self-loops, as a vertex with a self-loop in a de Bruijn graph must be a homopolymer vertex. Overlap graphs, on the contrary, are vertex labeled graphs, where an edge signifies a suffix-prefix match. This graph also can be converted similarly, by defining the edges and labels such that we avoid redundant overlaps along any path.

Bidirected graphs Assembly graphs and variation graphs are often used in their bidirectional form (Medvedev et al., 2007; Garrison et al., 2018; Novak et al., 2017) to incorporate the strand orientation within the graph. We can process these by first converting the bidirectional graph into its uni-directional form by splitting each vertex into two (one for each strand), followed by converting it into a directed graph.

5 Conclusions and Open Problems

The sequence to graph alignment problem is useful in the context of several applications in genomics, pan-genomics and transcriptomics. In this paper, we show that the problem is $\mathcal{NP}$-complete when changes are allowed in the sequence graph, for any alphabet of size ≥ 2.

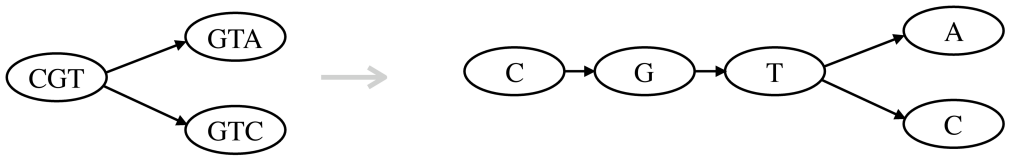

Figure 7: An example to illustrate the conversion of a de Bruijn graph (k=3) into the corresponding character labeled sequence graph.

When changes are allowed in the query sequence alone, we provide a asymptotically faster polynomial time algorithm that generalizes to linear gap penalty and affine gap penalty functions. The proposed algorithms use elementary data structures, therefore are simple to implement. Overall, the theoretical results presented in this work enhance the fundamental understanding of the problem, and will aid the development of faster tools for mapping to graphs.

The alignment problem for sequence graphs is a rich area with several unsolved problems. For the intractable problem variants, development of fast exact and approximate algorithms are fertile grounds for future research. The presented hardness proofs hold for general labeled graphs. As such, the problem complexity remains open for special instances (e.g., de Bruijn graphs). For the polynomially solvable problem variant, empirical evaluation of the proposed as well as existing approaches will help evaluate their practical utility. It will be useful to explore better algorithms when a substitution matrix (e.g., PAM, BLOSUM) based scoring is desired. Finally, working towards robust indexing schemes and heuristics that scale to large input graphs and different sequencing technologies is an active subject of research.

Acknowledgments

This work is supported in part by US National Science Foundation grant CCF-1816027. Yu Gao was supported by the ACO Program at Georgia Institute of Technology.

References

Amir, A., Lewenstein, M., and Lewenstein, N. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.

Antipov, D., Korobeynikov, A., McLean, J. S., and Pevzner, P. A. hybridspades: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*, 32(7):1009–1015, 2015.

Backurs, A. and Indyk, P. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 51–58. ACM, 2015.

Beretta, S., Bonizzoni, P., Denti, L., Previtali, M., and Rizzi, R. Mapping RNA-seq data to a transcript graph via approximate pattern matching to a hypertext. In *International Conference on Algorithms for Computational Biology*, pages 49–61. Springer, 2017.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

Dilthey, A., Cox, C., Iqbal, Z., Nelson, M. R., and McVean, G. Improved genome inference in the MHC using a population reference graph. *Nature genetics*, 47(6):682, 2015.

Eggertsson, H. P., Jonsson, H., Kristmundsdottir, S., Hjartarson, E., Kehr, B., Masson, G., Zink, F., Hjorleifsson, K. E., Jonasdottir, A., Jonasdottir, A., et al. Graphtyper enables population-scale genotyping using pangenome graphs. *Nature genetics*, 49(11):1654, 2017.

Equi, M., Grossi, R., Mäkinen, V., and Tomescu, A. I. On the complexity of string matching for graphs. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

Gagie, T., Manzini, G., and Sirén, J. Wheeler graphs: A framework for bwt-based data structures. *Theoretical computer science*, 698:67–78, 2017.

Garg, S., Rautiainen, M., Novak, A. M., Garrison, E., Durbin, R., and Marschall, T. A graph-based approach to diploid genome assembly. *Bioinformatics*, 34(13):i105–i114, 2018.

Garrison, E., Sirén, J., Novak, A. M., Hickey, G., Eizenga, J. M., Dawson, E. T., Jones, W., Garg, S., Markello, C., Lin, M. F., et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature biotechnology*, 2018.

Gotoh, O. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

Grice, J. A., Hughey, R., and Speck, D. Reduced space sequence alignment. *Bioinformatics*, 13(1):45–53, 1997.

Guo, H., Liu, B., Guan, D., Fu, Y., and Wang, Y. Fast variation-aware read alignment with debga-vara. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 227–233. IEEE, 2018.

Heber, S., Alekseyev, M., Sze, S.-H., Tang, H., and Pevzner, P. A. Splicing graphs and EST assembly problem. *Bioinformatics*, 18(suppl_1):S181–S188, 2002.

Heydari, M., Miclotte, G., Van de Peer, Y., and Fostier, J. Browniealigner: accurate alignment of illumina sequencing data to de bruijn graphs. *BMC bioinformatics*, 19(1):311, 2018.

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

Huang, L., Popic, V., and Batzoglou, S. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013.

Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature genetics*, 44(2):226, 2012.

Kavya, V. N. S., Tayal, K., Srinivasan, R., and Sivadasan, N. Sequence alignment on directed graphs. *Journal of Computational Biology*, 26(1):53–67, 2019.

Kuosmanen, A., Paavilainen, T., Gagie, T., Chikhi, R., Tomescu, A., and Mäkinen, V. Using minimum path cover to boost dynamic programming on DAGs: co-linear chaining extended. In *International Conference on Research in Computational Molecular Biology*, pages 105–121. Springer, 2018.

Lee, C., Grasso, C., and Sharlow, M. F. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

Limasset, A., Cazaux, B., Rivals, E., and Peterlongo, P. Read mapping on de bruijn graphs. *BMC bioinformatics*, 17(1):237, 2016.

Limasset, A., Flot, J.-F., and Peterlongo, P. Toward perfect reads: short reads correction via mapping on compacted de bruijn graphs. *bioRxiv*, 2019. doi: 10.1101/558395. URL `https://www.biorxiv.org/content/early/2019/02/28/558395`.

Liu, B., Guo, H., Brudno, M., and Wang, Y. debga: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32(21):3224–3232, 2016.

Manber, U. and Wu, S. Approximate string matching with arbitrary costs for text and hypertext. In *Advances In Structural And Syntactic Pattern Recognition*, pages 22–33. World Scientific, 1992.

Medvedev, P., Georgiou, K., Myers, G., and Brudno, M. Computability of models for sequence assembly. In *International Workshop on Algorithms in Bioinformatics*, pages 289–301. Springer, 2007.

Myers, E. W. An O(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

Myers, E. W. *An overview of sequence comparison algorithms in molecular biology*. University of Arizona. Department of Computer Science, 1991.

Myers, E. W. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.

Myers, E. W. and Miller, W. Approximate matching of regular expressions. *Bulletin of mathematical biology*, 51(1):5–37, 1989.

Navarro, G. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

Navarro, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

Nguyen, N., Hickey, G., Zerbino, D. R., Raney, B., Earl, D., Armstrong, J., Kent, W. J., Haussler, D., and Paten, B. Building a pan-genome reference for a population. *Journal of Computational Biology*, 22(5):387–401, 2015.

Novak, A. M., Hickey, G., Garrison, E., Blum, S., Connelly, A., Dilthey, A., Eizenga, J., Elmohamed, M. A. S., Guthrie, S., Kahles, A., et al. Genome graphs. *bioRxiv*, 2017. doi: 10.1101/101378. URL `https://www.biorxiv.org/content/early/2017/01/18/101378`.

Park, K. and Kim, D. K. String matching in hypertext. In *Annual Symposium on Combinatorial Pattern Matching*, pages 318–329. Springer, 1995.

Paten, B., Novak, A. M., Eizenga, J. M., and Garrison, E. Genome graphs and the evolution of genome inference. *Genome research*, 27(5):665–676, 2017.

Pevzner, P. A., Tang, H., and Waterman, M. S. An eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

Plesn, J. et al. The np-completeness of the hamiltonian cycle problem in planar diagraphs with degree bound two. *Information Processing Letters*, 8(4):199–201, 1979.

Rakocevic, G., Semenyuk, V., Lee, W.-P., Spencer, J., Browning, J., Johnson, I. J., Arsenijevic, V., Nadj, J., Ghose, K., Suciu, M. C., et al. Fast and accurate genomic analyses using genome graphs. Technical report, Nature Publishing Group, 2019.

Rautiainen, M. and Marschall, T. Aligning sequences to general graphs in O(V + mE) time. *bioRxiv*, 2017. doi: 10.1101/216127. URL `https://www.biorxiv.org/content/early/2017/11/08/216127`.

Rowe, W. P. and Winn, M. D. Indexed variation graphs for efficient and accurate resistome profiling. *Bioinformatics*, 34(21):3601–3608, 2018.

Salmela, L. and Rivals, E. Lordec: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

Sirén, J., Välimäki, N., and Mäkinen, V. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(2):375–388, 2014.

Thachuk, C. Indexing hypertext. *Journal of Discrete Algorithms*, 18:113–122, 2013.

Wang, J. R., Holt, J., McMillan, L., and Jones, C. D. Fmlrc: Hybrid long read error correction using an FM-index. *BMC bioinformatics*, 19(1):50, 2018.

Wick, R. R., Judd, L. M., Gorrie, C. L., and Holt, K. E. Unicycler: resolving bacterial genome assemblies from short and long sequencing reads. *PLoS computational biology*, 13 (6):e1005595, 2017.

Zhang, H., Jain, C., and Aluru, S. A comprehensive evaluation of long read error correction methods. *bioRxiv*, 2019. doi: 10.1101/519330. URL `https://www.biorxiv.org/content/early/2019/01/13/519330`.