ETH zürich

On the Complexity of Sequence-to-Graph Alignment

CHIRAG JAIN, HAOWEN ZHANG, YU GAO, and SRINIVAS ALURU

Mirko De Vita

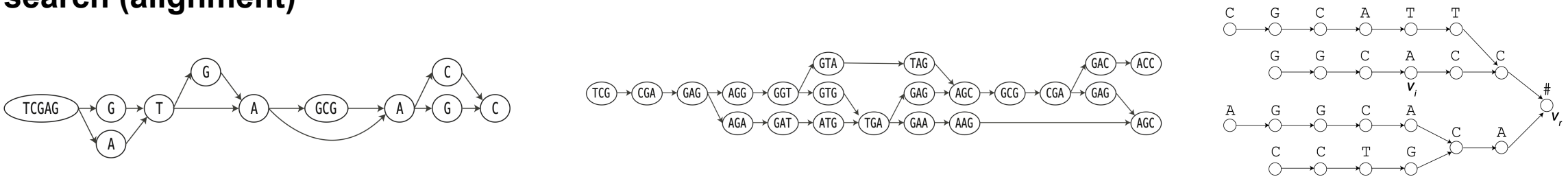
15th March, 2021

Background

- Strings can represent the genomic sequence of an individual or an aggregate of individuals

```
...TGGCTGTACCGAATAGCTAGATAGGCGCCTAGCTCGTGTGCTAGGATCGCATAGCTTGCTCTAATAGCTAGATCGCGCTAAA...
```

- They are useful for applications where a reference genome is needed. If there are several individuals to represent without taking a consensus, a lot of redundant information is created. How the redundancy could be reduced?
- Solution: **Genome graphs compress redundant information and allow for efficient exact and inexact search (alignment)**



String Graph, K-Mer Graph, BWT on Tree from the Computational Biomedicine course

- Aligning sequences to graphs has the potential to increase the number of aligned reads in the context of several applications:
    1. Variant calling: identifying single nucleotide polymorphisms (SNPs) and small insertions and deletion (indels)
    2. Genome assembly: process of taking a large number of short DNA sequences and putting them back together to create a representation of the original chromosomes from which the DNA originated.
    3. RNA - Seq data analysis: identification of splicing events; quantification of expression levels of genes, transcripts, and exons; differential analysis of gene expression.

Introduction

- In the sequence-to-graph approach the input graphs can be cyclic or acyclic and edits can be allowed in the graph or in the query, or in both.
- In this article, the proposed results hold for general directed graphs, i.e they 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. The following contributions are made:

Variants of the problem where changes to the graph labels are allowed were previously showed to be NP-complete assuming $|\Sigma| \geq |V|$.
The authors extend this result to constant sized alphabets ($|\Sigma| \geq 2$) in variants characterised by changes to graph alone or both graph and query, under the Hamming or edit distance models.

Allowing changes only to the query sequence makes the problem polynomially solvable.
The authors propose an $O(|V| + m|E|)$ algorithm for both linear and affine gap penalty cases, which is the same time required for the easier problem of sequence alignment to DAGs.

Preliminaries

Let Σ denote an alphabet, and x and y be two strings over Σ:

1. $x[i]$ denotes the i^{th} character of x, and $|x|$ denotes its length.
2. $x[i, j]$ denotes the substring of x beginning at the i^{th} position and ending at the j^{th} position
3. Concatenation of x and y is denoted as xy.
4. x^k denotes string x concatenated with itself k times.

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

Path $p = v_i, v_{i+1}, \ldots, v_j$ spells the sequence $\sigma(v_i)\sigma(v_{i+1}) \ldots \sigma(v_j)$.

Motivation

Given a query sequence q, the optimal alignment is its best matching path sequence in the graph.

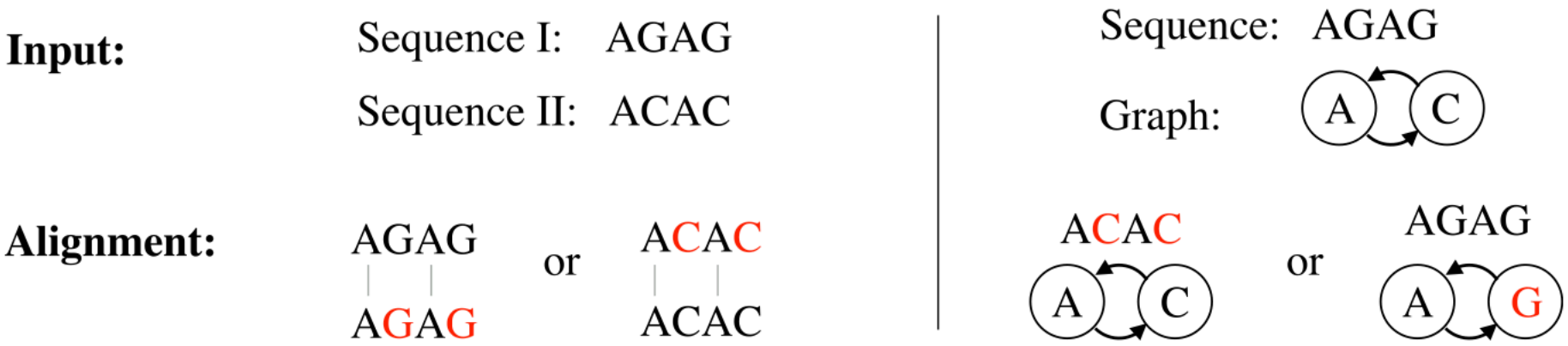
Optimal alignment problem: the distance between the computed path and q is minimised, subject to a specified distance metric such as Hamming or Edit distance:

The Hamming distance between two strings of equal length measures the minimum number of substitutions required to change one string into the other.

The Edit distance is a way of counting the minimum number of operations (insertions, deletions and substitutions) required to transform one string into the other.

An alignment under the Edit distance is scored using either a linear or an affine gap penalty function that imposes an additional cost to initiate a gap.

Motivation cont.



An alignment between two sequences describes the possible changes (e.g substitutions, insertions and deletions) that can be applied **either** to the first or the second sequences.

Symmetry is no longer valid when aligning sequences to graphs because alignments can occur along cyclic paths.

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.

As such, optimal alignment scores vary depending on whether changes are permitted in just the sequence, just in the graph or in both $\Longrightarrow$ three different problems $\Longrightarrow$ three different optimal distances.

Complexity analysis

For each distance metric, there are three versions of the problem depending on whether changes are allowed:

1. Query alone $\Longrightarrow$ Polynomial time solutions
2. Graph alone
3. Both in the query and graph

} Changes to graph are NP-complete.

NP-completeness was previously proved assuming $|\Sigma| \geq |V|$.

In what follows, the authors show that those problems remain NP-complete for any alphabet of size at least 2.

$|\Sigma| \geq 2$ is a tight bound since for $|\Sigma| = 1$ all the problem instances can be decided in polynomial time using standard graph algorithms that are able to find an exact matching path in the graph given the query sequence.

Proving NP-completeness

The problem belongs to class P if it can be solved in polynomial time.

The problem belongs to class NP if the solution can be verified in polynomial time but technically the problem cannot be solved in polynomial time.

The problem is NP-Complete if:

1. It belongs to NP
2. Every problem from NP polynomially reduces to it (NP-Hardness)

Reduction of a problem X to problem Y is a conversion of inputs of problem X to the inputs of problem Y. This conversion is a polynomial-time algorithm itself. The complexity depends on the length of the input.

1. In other words, given some information, it is possible to create a polynomial time algorithm that will verify for every possible input whether the answer to the input is ”Yes” or “No”.

2. To show that the problem is NP-Hard, choose another NP-Hard problem and reduce the chosen problem to the considered one.

Proving NP-completeness cont.

Finding the best matching path is an optimisation problem.

Unlike decision problems, for which there is only one correct answer for each input, optimisation problems are concerned with finding the best answer to a particular input.

Since there are standard techniques for transforming optimisation problems into decision problems,

consider the **decision version** of these problems:

Does an alignment with $\leq d$ modifications (substitutions or edits) exist?

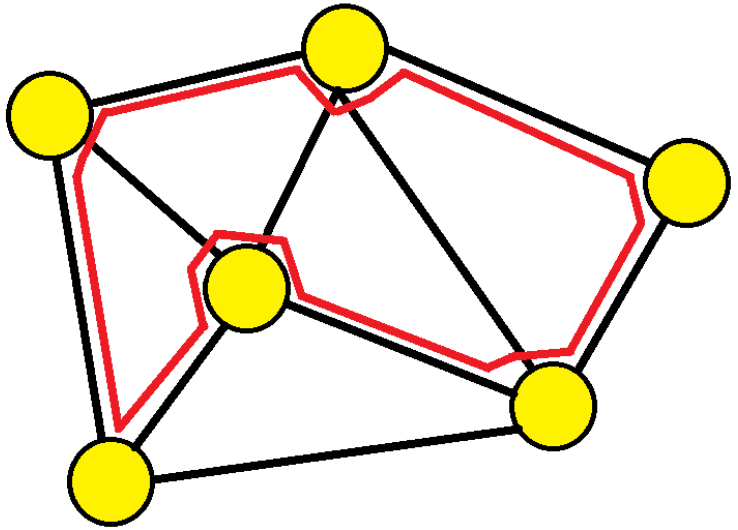
1. Is this decision problem in NP?

   Given a solution, it is possible to use any polynomial time algorithm to verify if q matches a path in the corrected graph.

2. To show that the problem is NP-hard, the authors perform a reduction using the directed Hamiltonian cycle problem.

Definition of Hamiltonian cycle

- A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once.
- A Hamiltonian cycle is a Hamiltonian path that is a cycle.
- Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.



Definitions and example from Wikipedia

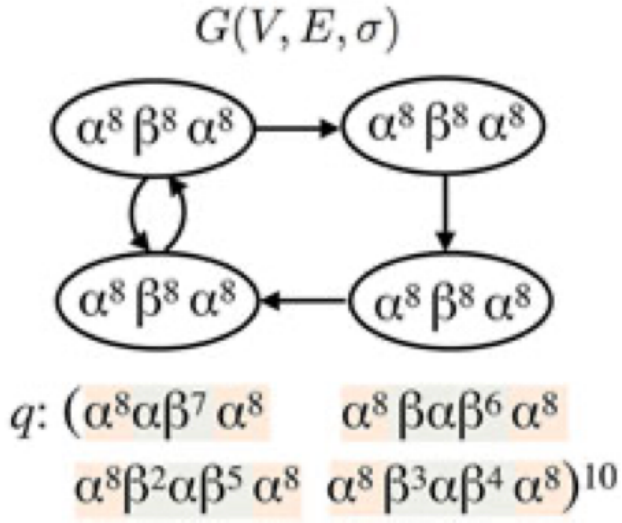
Alignment using Edit distance

Since the score under the Edit distance is upper-bounded by the one under the Hamming distance, the former is usually of practical interests. The Hamming distance case has a separate proof in the article.

Theorem 1: 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 NP-complete for $|\Sigma| \geq 2$.

- Given $G'(V, E)$, $G(V, E, \sigma)$ is designed using $\Sigma = \{\alpha, \beta\}$, $n = |V|$.
- Each vertex in G is a sequence of $6n$ characters $\alpha^{2n}\beta^{2n}\alpha^{2n}$.
- The query sequence $q = (t_o t_1 \dots t_{n-1})^{2n+2}$ such that

  $t_i = \alpha^{2n}\beta^i\alpha\beta^{2n-1-i}\alpha^{2n}$ of length $6n$.



Prove that a Hamiltonian cycle exists in $G'(V, E)$ if only if the sequence q can be matched to the graph $G(V, E, \sigma)$ using $\leq n$ total edits.

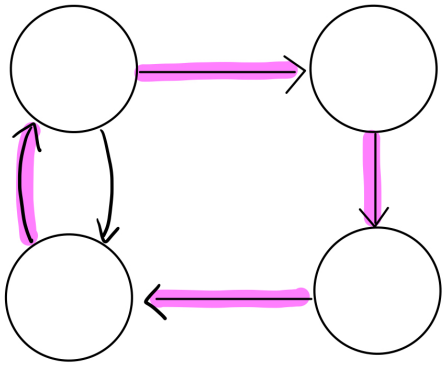
Proof for Theorem 1

A Hamiltonian cycle exists in $G'(V, E)$ if only if the sequence q can be matched to the graph $G(V, E, \sigma)$ using $\leq n$ total edits.

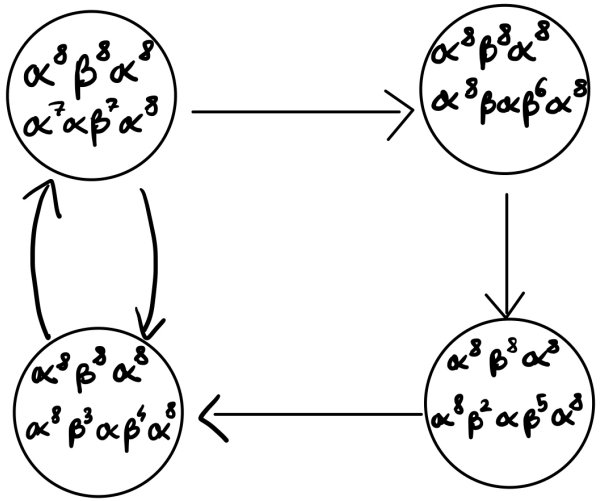
Hamiltonian cycle in $G'(V, E) \implies$ Sequence q can be matched to $G(V, E, \sigma)$ using $\leq n$ total edits.

Just follow the same loop in $G(V, E, \sigma)$ to align q and the alignment need one substitution per vertex.

Hamiltonian cycle in $G'(V, E)$



$G(V, E, \sigma)$ with substitutions



q: $(\alpha^8\alpha\beta^7\,\alpha^8 \quad \alpha^8\,\beta\alpha\beta^6\,\alpha^8$
$\alpha^8\beta^2\alpha\beta^5\,\alpha^8 \quad \alpha^8\,\beta^3\alpha\beta^4\,\alpha^8)^{10}$

Proof for Theorem 1 cont.

Sequence q can be matched to $G(V, E, \sigma)$ using $\leq n$ total edits $\Longrightarrow$ Hamiltonian cycle in $G'(V, E)$

To prove the conserve, 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 $q_{sub} = t_0 t_1 \ldots t_{n-1} t_0$

There are $n + 1$ non overlapping instances of q_{sub} in q:

For $n = 2$: $(t_0 t_1)^{2n+2} = (t_0 t_1)^6 = \underbrace{t_0 t_1 t_0}\ t_1\ \underbrace{t_0 t_1 t_0}\ t_1\ \underbrace{t_0 t_1 t_0}\ t_1 \rightarrow$ 3 instances of q_{sub}.

A total of $\leq n$ edits are allowed and even if all the n substitutions occur in the query, at least one instance of q_{sub} must remain unchanged.

With $n = 2$ substitutions $\underbrace{t_0 t_1 t_0}\ t_1\ \underbrace{t_0 t_1 t_0}\ t_1 t_0 t_1 t_0 t_1$ - $t_0 t_1 t_0 t_1\ \underbrace{t_0 t_1 t_0}\ t_1\ \underbrace{t_0 t_1 t_0}\ t_1$ - $\underbrace{t_0 t_1 t_0}\ t_1 t_0 t_1 t_0 t_1\ \underbrace{t_0 t_1 t_0}\ t_1$

$\Longrightarrow$ at least one instance must remain unchanged

$\Longrightarrow$ q_{sub} must match a path in the corrected $G(V, E, \sigma)$

Proof for Theorem 1 cont.

For the token t_i, let $k_i = \beta^i \alpha \beta^{2n-1-i}$ be its **kernel** sequence of length $2n$ such that $t_i = \alpha^{2n} k_i \alpha^{2n}$

A total of $\leq n$ edits is allowed

$\Longrightarrow$ any vertex label in G cannot shrink from $6n$ to $< 5n$ characters

$\Longrightarrow$ a kernel cannot be matched to an entire vertex after the edits

The kernel has size $2n$ and the vertex has minimum size $5n$

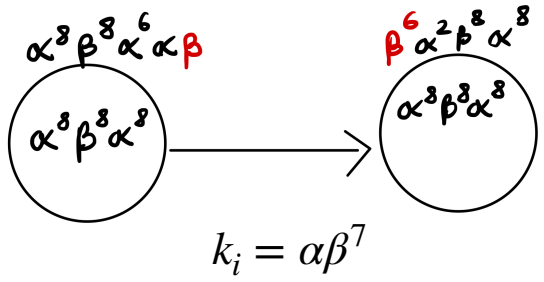
$\Longrightarrow$ **a kernel must match to ≤ 2 vertices**

Can it be aligned across two vertices?

A kernel aligning across two vertices requires $(2n-1)$ β's in place of α's at the two vertex ends

$\Longrightarrow$ $> n$ edits

$\Longrightarrow$ **the kernel cannot be aligned across two vertices**



Proof for Theorem 1 cont.

The two arguments show that **a kernel can only be matched within a single vertex label.**

Moreover, a single vertex label after $\leq n$ edits cannot be matched to more than one kernel.

Combining these statements with the fact that all n consecutive kernels in q_{sub} are unique,

the authors establish that the alignment path of q_{sub} must follow a Hamiltonian cycle in $G(V, E, \sigma)$

$\Longrightarrow$ **There is a Hamiltonian cycle in** $G'(V, E)$**.**

Edits in Graph alone

Corollary: The problem “Can we perform $\leq d$ edits in graph G so that q will match in G?” is NP-complete for $|\Sigma| \geq 2$.

Showing that the variant problem using only edits in $G(V, E, \sigma)$ is NP-complete is simple:

Define $q = (t_0 t_1 \dots t_{n-1})^2$ since substitutions in the query sequence are not allowed and therefore only one substring q_{sub} in q is needed.

Similarly to Theorem 1 proof, it is sufficient to show the existence of a Hamiltonian cycle in $G(V, E, \sigma)$.

Edits in Sequence alone

The sequence-to-graph alignment problem is polynomially solvable when changes are allowed in the query sequence alone.

TABLE 1. COMPARISON OF RUNTIME 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

	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	$O(\|V\|+m\|E\|)$	$O(\|V\|+m\|E\|)$	$O(\|V\|+m\|E\|)$

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 authors begin with the algorithm for the case of a linear gap penalty function and later they generalise it to affine gap penalty.

Without loss of generality:

From hereon, the sequence graph $G(V, E, \sigma)$ is a character-labeled graph, that is, $\sigma(v) \in \Sigma, v \in V$.

Alignment Graph

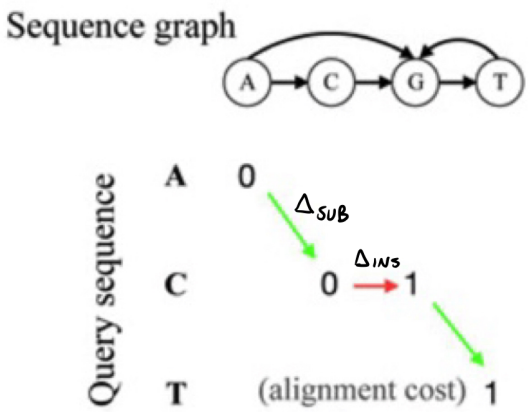
The alignment graph is a weighted directed graph, which is constructed such that every path from source vertex s to sink vertex t corresponds to a valid alignment.

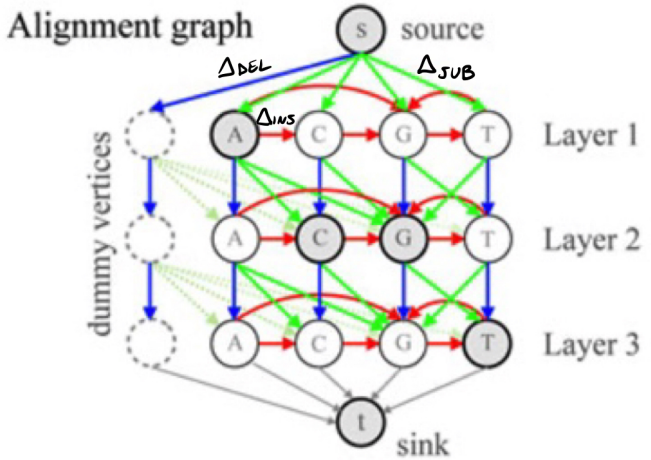
The alignment cost is equal to the corresponding path distance from s to t.

Definition: Given a query sequence q, a sequence graph $G(V, E, \sigma)$, linear gap penalty parameters Δ_{del},Δ_{ins}, and a substitution cost parameter Δ_{sub}, the corresponding alignment graph is a weighted graph $G_a(V_a, E_a, \omega_a)$, where $V_a = (\{1,...,m\} \times (V \cup \{\delta\})) \cup \{s, t\}$ is the vertex set and $\omega_a : E_a \rightarrow \mathbb{R}^+$ is the weight function.

Edges $(x, y) \in E_a$ are pairs (x, y) for which ω_a is defined.

$\Delta_{i,v} = \Delta_{sub}$ if $q[i] \neq \sigma(v), v \in V$, 0 otherwise.





$\Delta_{del}, \Delta_{ins}, \Delta_{sub} > 0$

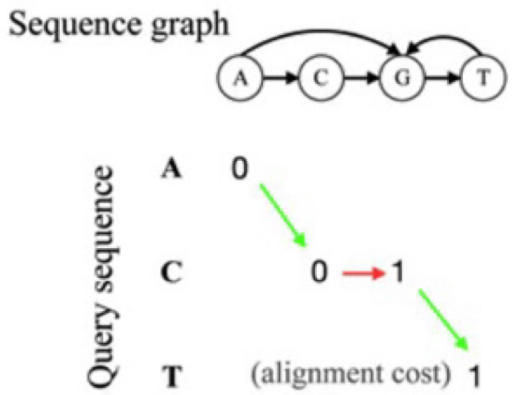
A column of dummy vertices is required to accommodate the possibility of deleting a prefix of the query sequence.

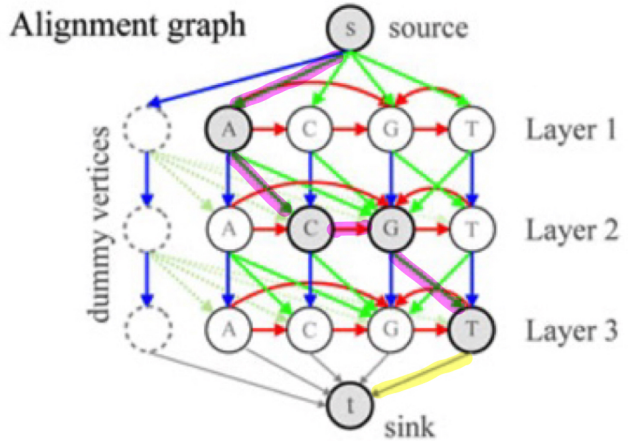
Alignment Graph cont.

Lemma [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 the cost of optimal alignment between query q and the sequence graph $G(V, E, \sigma)$.

- A is a match, (s, A) cost is 0
- C is a match, (A, C) cost is 0
- G is not present in q, (C, G) is an insertion with cost $\Delta_{ins} = 1$
- T is a match, (G, T) cost is 1

Optimal cost $= 1$





Previous work by Antipov et. al. (2015) $O(m|V|\log(m|V|) + |E|)$

The new proposed algorithm achieves a runtime complexity $O(|V| + m|E|)$ without using a standard priority queue.

Changes in the sequence alone algorithm

1. ***InitializeDistance****:* any path from s 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.

INPUT: an array of the shortest path distances of the vertices in previous layer sorted in non-decreasing order (by distance)

This stage computes the “tentative” distances of all vertices in the current layer because it ignores the insertion cost weighted edges during the computation.

OUTPUT: sorted tentative distances as an input for the second stage.

2. ***PropagateInsertion****:* this stage returns the optimal distances of all vertices in the current layer while maintaining the sorted order for a subsequent iteration.

Algorithm 1: Algorithm for sequence-to-graph alignment

Result: The length of shortest path from s to t

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

The InitializeDistance stage

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

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

```
1 Function InitializeDistance (PreviousLayer, CurrentLayer)
2 |  foreach x ∈ PreviousLayer do
3 |  |  foreach y ∈ x.neighbory & y ∈ CurrentLayer do
4 |  |  |  if y.distance > x.distance + ω_a(x, y) then
5 |  |  |  |  y.distance = x.distance + ω_a(x, y);
6 |  |  |  end
7 |  |  end
8 |  end
9 |  Sort (CurrentLayer);
```

All deletion and substitution cost weighted edges are directed from the previous layer toward the current one

$\Longrightarrow$ linear $O(|V| + |E|)$ time traversal.

A vertex $v \in$ *PreviousLayer* can only assign three possible distance values (*v.distance, v.distance* + Δ_{sub} *or v.distance* + Δ_{del}*)* to a neighbour in the *CurrentLayer.*

One list for each possibility (created in sorted order) $\Longrightarrow$ merge in $O(|V|)$

$\Longrightarrow$ Order of vertices in the *CurrentLayer* determined in linear time by tracking positions of their distance values in the merged list

The PropagateInsertion stage

When processing vertex v, the distance of its neighbour should be adjusted such that it is no more than *v.distance* + Δ_{ins}.

Selecting vertices with minimum scores:

- Fibonacci Heap: $O(|E| + |V| \log |V|)$ time per layer.
- All considered edges have uniform weights

$\Longrightarrow$ **two First-in-First-Out queues**

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

q_1 always maintains its non-decreasing sorted order since there are no new enqueued elements after the init.

A. q_1 initialised with sorted vertices in the current layer

   q_2 initialised as empty

B. Minimum distance vertex dequeued from q_1 or q_2

C. Vertex enqueued in q_2 when its distance is updated

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         qmin = q1. Front() < q2. Front() ? q1 : q2;
 8         x = qmin. Dequeue ();
 9         if x.resolved = false then
10             x.resolved = true;
11             CurrentLayer. Append (x);
12             foreach y ∈ x.neighbory & y.layer = x.layer do
13                 if y.distance > x.distance + Δins then
14                     y.distance = x.distance + Δins;
15                     q2.y Enqueue (y);
16                 end
17             end
18         end
19     end
```

PropagateInsertion complexity analysis

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

Proof:

Each vertex in the current layer enqueues its updated neighbours into q_2 at most once.

There are no enqueue operations in q_1.

The distance of a vertex can be updated at most once $\Longrightarrow$ maximum number of enqueue operations is $|V|$

Bounded by $O(|V|)$

Executed at most once per vertex, amortised runtime is $O(|V| + |E|)$

The above claims yields $O(m(|V| + |E|))$, tightened to $O(|V| + m|E|)$ using a preprocessing step that merges all the vertices with 0 in-degree into $\leq \Sigma$ vertices $\Longrightarrow$ no more than $|E| + |\Sigma|$ vertices.

```
Function PropagateInsertion (CurrentLayer)
    x.resolved = false ∀x ∈ CurrentLayer;
    Queue q1 = ∅, q2 = ∅;
    q1.Enqueue(CurrentLayer);
    CurrentLayer = [ ];
    while q1 ≠ ∅ or q2 ≠ ∅ do
        qmin = q1. Front() < q2. Front() ? q1 : q2;
        x = qmin. Dequeue ();
        if x.resolved = false then
            x.resolved = true;
            CurrentLayer. Append (x);
            foreach y ∈ x.neighbory & y.layer = x.layer do
                if y.distance > x.distance + Δins then
                    y.distance = x.distance + Δins;
                    q2.y Enqueue (y);
                end
            end
        end
    end
```

Affine Gap Penalty

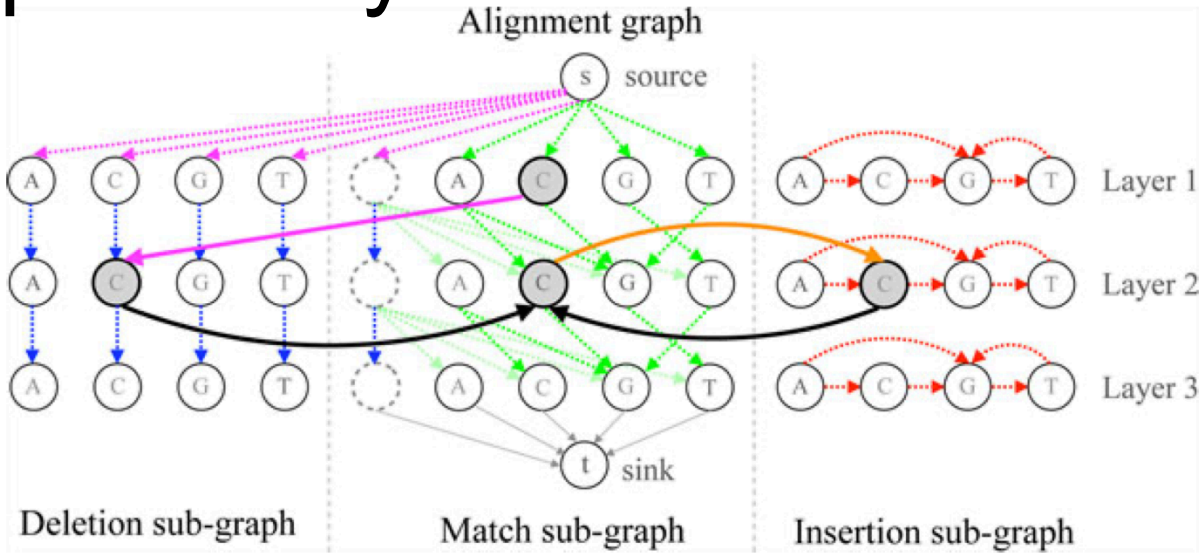
Dynamic programming algorithm for sequence-to-sequence alignment with affine gap penalty functions uses three scoring matrices instead of just one (Gotoh, 1982).

Similarly, the alignment graph can be extended to contain **three subgraphs** with substitution, deletion, and insertion cost weighted edges

The cost for opening a gap is incurred whenever a path leaves the match subgraph to either the insertion or the deletion subgraph

Previous algorithm with m iterations is adapted with five stages:

1. Initialisation of optimal distances of vertices in the deletion layer using distances of vertices in the above match and deletion layers



2. Initialisation of distances of vertices in the current match layer using distances of vertices in the current deletion layer and the above match layer

3. Distances of vertices in the current match layer utilised to initialise the current insertion layer

4. Distances in the current insertion layer are resolved using the insertion propagation algorithm

5. Distances of vertices in the current insertion layer are used to make a final update to the current match layer.

Lower Bounds

Does a faster algorithm for solving the sequence-to-graph alignment problem exist?

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. [Rautiainen and Marschall (2017)]

The 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 is unlikely:

Arturs Backurs and Piotr Indyk. 2015. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false).

The above article provides evidence that the near-quadratic running time bounds known for the problem of computing edit distance might be tight. Specifically, if such algorithm exists it would violate the Strong Exponential Time Hypothesis, which is believed to be true but not formally proved.

Applicability of the proposed algorithm

Is the algorithm faster if the number of edits is constrained (Banded Alignment)?

Changes in the graph : exact and approximate matching to graphs were proved to be equally hard problems [Equi et al. (2019)]

Changes in the query alone: the banded alignment strategy would only reduce the m factor, which is less significant than the $|E|$ factor and the overall time-complexity is still quadratic in the input size.
Overall, it is difficult to scale the algorithm for high throughput DNA sequencing data.

C. Jain, S. Misra, H. Zhang, A. Dilthey and S. Aluru, "Accelerating Sequence Alignment to Graphs," *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, Brazil, 2019

Given a **variation graph** in the form of a directed acyclic string graph, the above article proposes the first parallel algorithm for computing sequence to graph alignments that exploits multiple cores and single-instruction multiple-data (SIMD) operations.

Future work will try to extend this framework to accelerate the alignment to **general** sequence graphs, which could finally lead to the practical use of the proposed algorithm.

Conclusion and open problems

The sequence-to-graph alignment problem has several applications in genomics, pangenomics, and transcriptomics.

The article shows that the problem is NP- complete when changes are allowed in the sequence graph for any alphabet of size ≥ 2.

When changes are allowed in the query sequence alone, the authors provide an asymptotically faster polynomial time algorithm that generalises to linear gap penalty and affine gap penalty functions.

The alignment problem for sequence graphs is a rich area, especially for the intractable problem variants the development of fast exact and approximate algorithms is fertile ground 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).

It will be also useful to explore better algorithms when a substitution matrix (e.g., PAM, BLOSUM)-based scoring is preferred.

Supplementary material

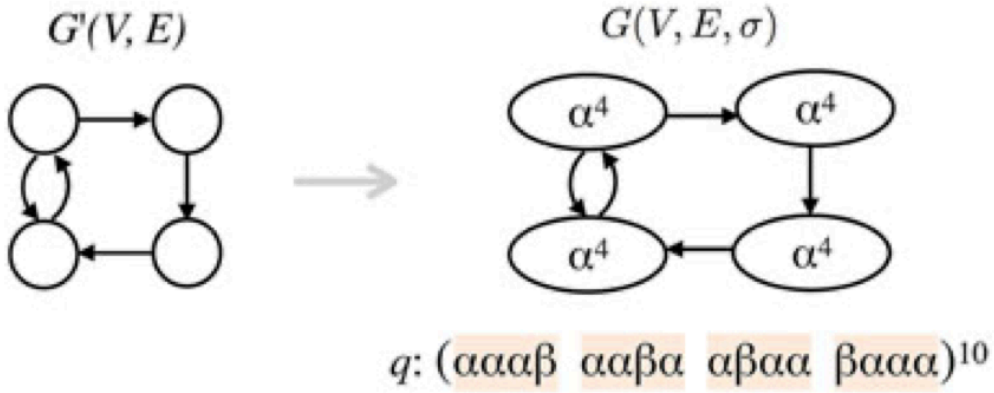
Alignment using Hamming distance

Theorem: 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 NP-complete for $|\Sigma| \geq 2$.

$G'(V, E)$ = directed graph in which we seek a Hamiltonian cycle

$n = |V|$

$G(V, E, \sigma)$ = sequence graph over $\Sigma = \{\alpha, \beta\}$ obtained from $G'(V, E)$ by labelling each vertex $v \in V$ with α^n.



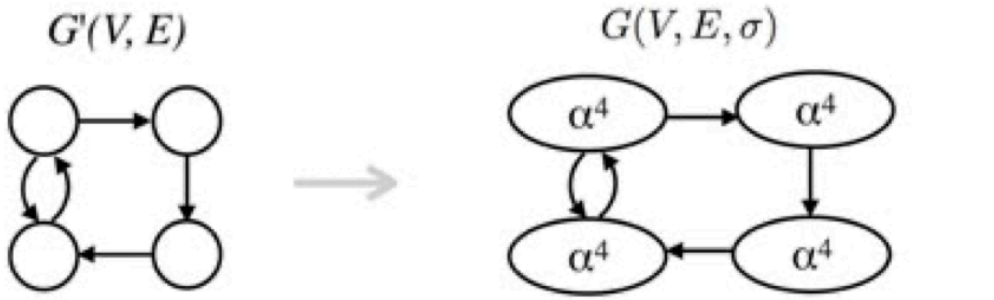
Constructing the query sequence q:

$t_i = \underbrace{\alpha^{n-i-1}\beta\alpha^{i}}_{n}$ such that $q = (t_o t_1 \ldots t_{n-1})^{2n+2}$ is $n^2(2n+2)$ characters long.

Claim: A Hamiltonian cycle exists in $G'(V, E)$ iff q can be matched after substituting a total of $\leq n$ characters in $G(V, E, \sigma)$ and q.

Hamiltonian Cycle in $G'(V, E) \Longrightarrow q$ can be matched with a total of $\leq n$ substitutions in $G(V, E, \sigma)$ and q:

We can follow the corresponding loop in $G(V, E, \sigma)$ from the first character of any vertex label. To match each query q, we require one $\alpha \rightarrow \beta$ substitution per vertex. This means that we are 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 $q_{sub} = t_0 t_1 \ldots t_{n-1} t_0 t_1$. Note there are $n + 1$ non-overlapping instances of q_{sub} in q.

For $n = 2$: $(t_0 t_1)^{2n+2} = (t_0 t_1)^6 = \underbrace{t_0 t_1 t_0 t_1}\ \underbrace{t_0 t_1 t_0 t_1}\ \underbrace{t_0 t_1 t_0 t_1}$ $\rightarrow$ 3 instances of q_{sub}.

Even if all the n substitutions occur in the query, at least one instance of q_{sub} must remain unchanged.

With $n = 2$ substitutions $\underbrace{t_0 t_1 t_0 t_1}\ \underbrace{t_0 t_1 t_0 t_1}\ t_0 t_1 t_0 t_1$ / $t_0 t_1 t_0 t_1\ \underbrace{t_0 t_1 t_0 t_1}\ \underbrace{t_0 t_1 t_0 t_1}$ / $\underbrace{t_0 t_1 t_0 t_1}\ t_0 t_1 t_0 t_1\ \underbrace{t_0 t_1 t_0 t_1}$

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.

$q_{sub}[1,n] = t_0, q_{sub}[n+1,2n] = t_1, \ldots, q_{sub}[n^2-n+1,n^2] = t_{n-1}$ are all unique followed by:
$q_{sub}[n^2+1,n^2+n] = t_0 \rightarrow$ Hamiltonian cycle in $G(V,E,\sigma) \rightarrow$ Hamiltonian cycle in $G'(V,E)$.

Case 2: q_{sub} starts somewhere other than the starting position within a vertex label.

Let $q_{sub}[k]$ $(1 \leq k \leq n)$ be the first character that matches at the beginning of the next vertex on the path matching q.

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

Example with $n = 2$: $\alpha\beta\beta\alpha\alpha\beta\beta\alpha\alpha\beta\beta\alpha$

$k = 1 : q_{sub}[k, n+k-1] = \alpha\beta, q_{sub}[n+k,2n+k-1] = q_{sub}[n^2-n+k, n^2+k-1] = \beta\alpha$

$k = 2: q_{sub}[k, n+k-1] = \beta\beta, q_{sub}[n+k,2n+k-1] = q_{sub}[n^2-n+k, n^2+k-1] = \alpha\alpha$

Corollary: The problem “Can we substitute $\leq d$ characters in graph G such that q will have a matching path in G?” is NP-complete for $|\Sigma| \geq 2$. (changes to graph alone)

The proof is simple:

We define $q = (t_0 t_1 \ldots t_{n-1})^2$ since only one substring q_{sub} in q is needed. This is because substitutions in the query sequence are not allowed. Therefore, we need to show that a Hamiltonian cycle in G exists like we did for proving the theorem.

PropagateInsertion stage proof of correctness

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

q_1 always maintains its non-decreasing sorted order since we never enqueue new elements after the init.

Proof by contradiction that q_2 also maintains the order:

Let $i > 1$ be the first iteration where the order is lost.

All the new vertices $y_1, y_2, \ldots, y_k$ enqueued in q_2 and the vertex x that caused the additions dequeued.

The distance of the y_i's equals *x.distance* + Δ_{ins}.

The vertex before y_1 (say y_{pre}) must have a distance higher than y_1.

Contradiction, when y_{pre} was enqueued to q_2 the distance of the vertex that caused the addition could not be higher than the distance of vertex x.

Thus, the algorithm mimics the choices made by Dijkstra's algorithm.