Sequence to Graph alignments for *Pan-genomic* graphs

Souvadra Hati (Sr: 15551)

May 26, 2021

Problem with Reference Genome: The first genome was sequenced with the official completion of the **Human Genome Project** on April 14, 2003, after spending more than 3 billion USD in today's term. This task was a marvelous achievement for humanity as we finally started to look into the *source code* of our own species. The output of this project was basiclly a 3×10^9 base pairs long A, T, G, C sequence that became the standard reference for all the Genetics and Molecular Biology experiments based on human cells and tissues and within a very short time provided scientists all across the world a standard to understand what was wrong in the "*source code*" of diseased individuals and helped them come up with genetic basis of plethora of diseases. Within that very short period of time, sequencing technology also improved and the sequencing became cheaper and cheaper. With this, sequencing became more and more popular and we got to know that the genome of all individuals of *Homo sapiens* are not the same and that difference was not due to any disease or anything, but because of shear diversity present across all human beings on this planet. Still, most of the state-of-the art Bioinformatics pipelines uses a single string as the reference, which is causing **reference bias**[3] due to not being able to provide adequate information of the natural variations found in a diverse set of human population present on the face of this planet. This reference bias, although might look minor for us to ignore in many cases. But some important parts of our genome which is highly variable even within the same individual in the RNA and protein level and sometimes even in the DNA level due to some advanced mechanics persent in human cells. Hence, finding those variations are crucial in many scientific development. One such example is **Major Histocompatibility Complex** (MHC) that plays essential roles in proiding memory-based immunity in higher organisms, without which even common cold would have been as deadly as SARS CoV2 virus that causes COVID-19 disease. But, all these important properties are missing from a linear string based reference genome and hence it is high time to shift towards a more suitable reference for all Genomics research.

Pan-Genome and Graphs: The term 'pan-genome' refers to any collection of genomic sequences to be analyzed jointly or to be used as a reference. The goal of computational pan-genomics is to provide completeness to the genomics reference which will fully cover the total complexity of the genome by incorporating all the variations present in the population scale in a compact, efficient, scalable manner. One of the prime candidate for a proper data-structure for constructing a pan-genome due to its natural capability of expressing variation in the genome [4]. But, use of graph data-structures as a reference genome (*genome graph*) comes up with additional challenges when it comes to the problem of query alignment, which is one of the most fundamental operations in Bioinformatics pipelines [11].

Alignment Problem: Aligning sequence to sequence is one of the oldest problems in Bioinformatics research and a number of tools already exist to perform that operation smoothly, most of which are based on the classic Needleman and Wunsch algorithm [10] with additional heuristic functions to accomplish differnt types of alignment (glocal / local) with different cost functions (Hamming / Edit distance). But, until recently, there was not much work done in the realm of string-to-graph alignment. One of the earlier attempts in sequence-to-graph alignment has been studied by G. Navarro[9] in the year 2000 on the topic of hypertext seaching, outside of the realm of Bioinformatics.

Navarro's algorithm is a provably correct algorithm capable of performing approximate pattern matching on hypertext (both cyclic and acyclic graphs) with $O(n)$ extra space and asymptotic time complexity of $O(m(n+e))$, where m is the length of the pattern, n is the total size of the text ad e is the total number of edges [9] provided only pattern can have errors (and hence can accomodate edit operations).

Unlike strings, graphs don't necessarily come up with a natural way to traverse them from beginning to end (specially in case of cycles that frequently arises in Genome graphs as *indels*). Hence, the conventional DP method of strings are not possible to apply here directly. But, we can essentially apply the similar idea by using a slighly different approach to traverse the graph. To understand the algorithm, let's first define some variables:

- C_v denotes the state per node. At each iteration of the algorithm, a new version of the node, C'_v are computed, corresponding to the $O(n)$ additional space.

The recursion can be expressed as:

$$g(v,i) = \begin{cases} \min\left(\{C_u/(u,v) \in E\} \cup \{i-1\}\right) & \text{if } patt[i] = t[v] \\ 1 + \min\left(C_v, \min_{(u,v)\in E} C_u\right) & \text{otherwise} \end{cases}$$

And the pseudocode looks like:

Algorithm 1 Navarro's algorithm

```
function SEARCH(V, E, patt)
    for all v ∈ V do C_v ← 0
    for i = 1 to m do
        for all v ∈ V, C'_v ← g(v, i)
        for all v ∈ V, C_v ← C'_v
        for alll (u, v) ∈ E, Propagate(u, v)

function PROPAGATE(u, v)
    if C_v > 1 + C_u then
        C_v ← 1 + C_u
        for all z/(v, z) ∈ E do
            Propagate(v, z)
```

The **Propagate** function works only $O(1)$ time per edge whenever a reduction is made in its source node and the overall algorithm terminates after $O(e)$ amount of computation per character in the pattern [9]. If we extrapolate this algorithm to a simple linear graph to string alignment then $|e| = O(n)$ and hence the overall time complexity of the algorithms becomes $O(mn)$ which is as efficient as the classic Needleman and Wunsch algorithm.

Although the algorithm provided by Navarro provides a good intuition behind how the ideas of string to string alignment can be extrapolated to this task in a simple and elegant manner, it is not the most general and efficient algorithm for that job. After the inital work of Navarro, many Computer Scientists [5, 2] have come up with modified versions of that algorithm capable of performing the same operation in $O(|V|+m|E|)$ time when **edit distance** is considered.

Jain et al (2020) has finally showed a provably correct algorithm capable of performing the string to graph alignment using arbitrary linear gap penalty and affine gap penalty functions in $O(|V| + m|E|)$ time [8] and also provided a stronger argument than Amir et al (1997) (that assumed alphabet size $\geq |V|$) regarding the NP-completeness of the problem if we consider changes in both the query and the text for alphabet size ≥ 2 [8, 1].

I am going to discuss the algorithm for edit distance model here, but there is a natural extension to this algorithm to incorporate affine gap distance which can be found in the original article. So, for proposing the algorithm, we have to first define an *alignment graph*. Let's first focus on the formal definition of alignment graph from Jain et al (2020) itself.

"**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 $V_a = (\{1, ..., m\} \times (V \cup \{\delta\})) \cup \{s, t\}$ in the vertex set, and $\omega_a : E_a \to \mathbb{R}_{\geq 0}$

is the weight function defined as

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

”

Dummy vertices are required to account for the deletion operation.

Informally speaking, an alignment graph is a way to incorporate all possible ways to align the query sequence q to a sequence graph $G(V,E,\sigma)$ by taking into consideration of all the possible edit operations, such that any path that can connect source *'s'* to destination *'t'*, will correspond to one way of aligning the sequence to the graph. Now, the the job of the algorithm is to find the shortest possible path from *'s'* to *'t'*, which will indirectly give us the lowest amount of edit cost required to perform the alignment and also tell us how that alignment work in an intuitive manner. To make this statment clearer, I am using the example presented in Jain et al (2020):

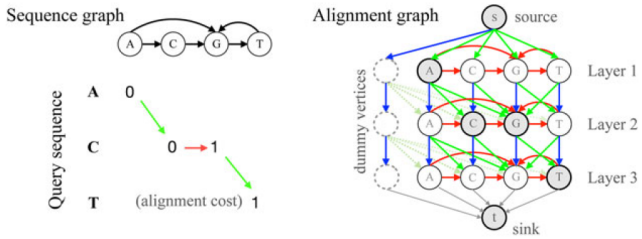


FIG. 3. An example to illustrate the construction of an alignment graph (right) from a given sequence graph and a query sequence (left). Multiple colors are used to show weighted edges of different categories in the alignment graph. The red, blue, and green edges are weighted as insertion, deletion,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

Although, the most obivous way to find the shortest path from s to t is to use Dijkstra's algorithm, but that will take $O(m|V|log(m|V|)+m|E|)$ time. Another way to find the shortes path is to use A^* search algorithm, with an efficient heuristic function (which has been done on a slightly different alignment graph by Ivanov et al (2020) [6], which we are gonig to briefly discuss later).

First, I am going to show the psudocode of the algorithm as given in the Jain et al (2020) and then explain the algorithm in simple terms.

Algorithm 2 Jain et al Algorithm

Result: The length of the shortes path from s to t
$PreviousLayer = [s]$;
$s.distance = 0$;
for i = 1 to m **do**
  CurrentLayer = $[(i, v_1), (i, v_2), ..., (i, v_n), (i, k)]$;
  $x.distance = \infty \ \ \forall x \in CurrentLayer$;
  InitializeDistance($PreviousLayer$, $CurrentLayer$);
  PropagateInsertion($CurrentLayer$);
  $PreviousLayer = CurrentLayer$;
Return Min(PreviousLayer.distance);

function InitializeDistance($PreviousLayer, CurrentLayer$)
  for each $x \in PreviousLayer$ **do**
    for each $y \in x.neighbory$ and $y \in CurrentLayer$ **do**
      if $y.distance > x.distance + \omega_a(x, y)$ **then**
        $y.distance = x.distance + \omega_a(x, y)$;
  Sort($CurrentLayer$);

function PropagateInsersation($CurrentLayer$)
  $x.resolved = false \ \ \forall x \in CurrentLayer$;
  Queue $q_1 = \phi, q_2 = \phi$;
  q_1.Enqueue($CurrentLayer$);
  $CurrentLayer = []$;
  while $q_1 \neq \phi$ or $q_2 \neq \phi$ **do**
    $q_{min} = q_1.Front() < q_2.Front()?q_1 : q_2$;
    $x = q_{min}.Dequeue()$;
    if $x.resolved = false$ **then**
      $x.resolved = true$;
      $CurrentLayer$.Append(x);
      for each $y \in x.neighbory$ and $y.layer = x.layer$ **do**
        if $y.distance > x.distance + \Delta_{ins}$ **then**
          $y.distance = x.distance + \Delta_{ins}$;
          q_2.Enqueue(y);

So, the idea behind this algorithm is to traverse the alignment graph *layer by layer*. As per our construction of the alignment graph, the vertices can be arranged into $|q| = m$ number of layers. While travelling one of the layers, we assume the vertices of the pervious layer is already sorted in increasing order of distances. We first use the 'InitializeDistance' function that computes the *tentative* distance of the vertices in the current layer, based on the shortest distances computed in the previous layer. This function basically computes the **deletion** and **substitution** operations as those edges traverse from previous layer to current layer. Once this operation is done, we perform the 'Propa-

gateInsersation' function (which is quite similar to the *Propagate* function used by Navarro (2000) in his algorithm) that then makes sure if there exits any **insertion** edge that can decrease the cost of any of the vertices in the current layer, it updates that accordingly. The 'Propagateinsersation' function finishes its operation in $O(|E| + |V|)$ time via exploiting the fact that Δ_{ins} is a constant. It uses two **Queue** data-structures to implement this operation, by first sorting the vertices in the current layer so far based on the output from the 'InitializeDistance' function and store it into Queue 1 and keep Queue 2 *empty*. In each step we take the queue with front node with minimum distance and dequeue the front vertex from that queue. If the distance of that vertex can be decreased via use of an insertion edge we perform that update and enqueue that node in the Queue 2. We keep doing this till one of the queues remain *non-empty*. So, after both this helper function, we essentially get the minimum distance of all the vertices till that layer in the alignment graph. We then sort the vertices according to their distance and then move on to the next layer and do the exact same operation, till we reach the target t.

Next, I am going to briefly discuss about the alignment algorithm proposed in a preprint by Ivanov et al (2020). The approach taken by the author was slightly different in its construction compared to what we have seen so far.

Here, we consider the edges of the *reference graph* to contain the letters (belonging to the alphabet) instead of the nodes. The fundamental idea here is to create a *edit graph* from the reference graph based on the edit distance model (under the assumption that)$0 \leq \Delta_{match} \leq \Delta_{sub}, \Delta_{ins}, \Delta_{del})$. Then we convert the edit graph into an *alignment graph* (this alignment graph is slightly different from that of Jain et al (2020)). The formulation of the graphs are:

- **Reference graph:** Graph $G_r = (V_r, E_r)$ of nodes V_r and directed, labelled edges $E_r \subseteq V_r \times V_r \times \Sigma$. Σ represents the alphabet which is $\{A, T, G, C\}$ in the realm of Genomics. Any sequence can be read as a path of this graph where we just concatenate all the letters we read in each edges as we traverse.

- **Edit graph:** Graph $G_e \coloneqq (V_e, E_e)$ has directed, labeled edges $E_e \subseteq V_e \times V_e \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ with cost associated with each edit operation. Here, $\Sigma_\epsilon \coloneqq \Sigma \cup \{\epsilon\}$ extends the alphabet to incorporate another letter for denoting *deleted characters*. Edit graph containt equal number of vertices as that of reference graph but the number of edges in edit graph is a constant multiple of that of the reference graph as in edit graph, we have separate edges to denote all the possible edit operations. The optimal alignment on an edit graph is the alignment with minimal edit cost.

- **Alignment graph:** Graph $G_a^q = (V_a^q, E_a^q)$, where each state $\langle v, i \rangle \in V_a^q$ corresponds a vertex $v \in V_e$ and represents the query position $i \in \{0, 1, ..., |q|\}$. The alignment of the i^{th} character of the query sequence and the node v of the reference/edit graph can be represented by the state $\langle v, i \rangle$. Hence, any path from the *source* $\langle u, 0 \rangle$ to a state $\langle v, i \rangle$ represents

an alignment of the first i letters of the query q to G_r. Basically the whole idea of constructing an alignment graph from an edit graph is to make sure, we only consider the paths of edit graph which are spelling the sequence q and reduce the search space.

A very simple example of all these three kinds of graphs can be found in Ivanov et al. (2020):

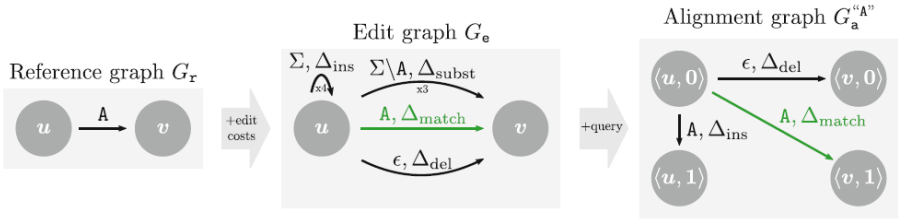
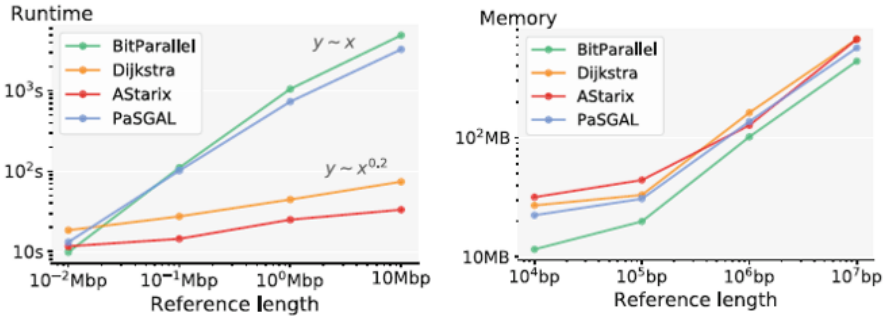


Fig. 1. Starting from the reference graph (left), we can construct the edit graph (middle) and the alignment graph G_a^q for query $q =$ "A" (right). Edges are annotated with labels and/or costs, where sets of labels represent multiple edges, one for each letter in the set (indicated by "x3" and "x4").

To keep it concise I am avoiding detailed discussion of the algorithm used here. So, in a nutshell their algorithm primarily performs two task. First, given a query and the reference graph, it construct the alignment graph of the problem. It then performs an A^* search algorithm on the alignment graph to find out the shortest distance from the source vertex $\langle u, 0 \rangle$ to the target vertex $\langle v, |q| \rangle$, which will essentially correspond to the sequence to graph alignment.

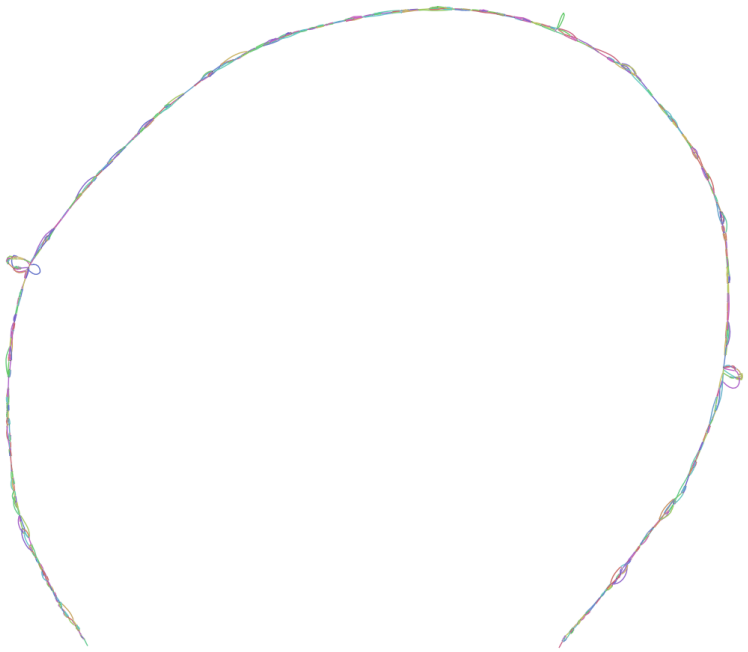
Now, the challenge here, is to use an efficient heuristic function h to let the A^* search algorith wisely choose the edges to avoid going through all the edges in the alignement graph (which can totally happen if we use Dijkstra's algorithm instead of A^* search). For a detailed account of the optimaility of the heuristic function used, the reader is adviced to go through Ivanov et al (2020).

Ivanov et al (2020) has also performed an experiment showing the time and space requirement of AStarix (both with A^* and $Dijkstra's$ algorithm), PaSGAL[7], and BitParallel[12] when ran on single thread (because some of these tools were highly optimized for parallel performance). [For implementing the $Dijkstra's$ algorithm, they just modified the heuristic function of A^* to be **Zero** all the time.]



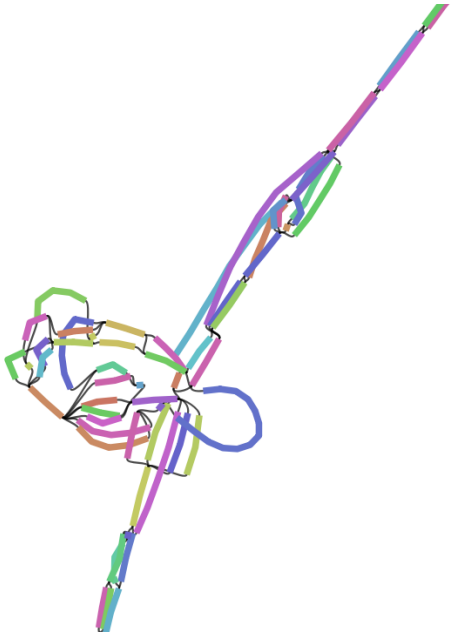
This experiment essentially shows that although, the theoretical complexity of all these algorithms are same, the formulation given by Ivanov et al (2020) experiences much lower run-time compared to others.

Experiment In order to show that graph is a better reference than that of string , I have used a small dataset of genomes. On top of that I constructed some more synthetic version of those genomes randomly and constructed a graph using the 'minigraph toolkit'. I unassisted 'Bandage' for visualization of the graphs. The graph generated looks like:

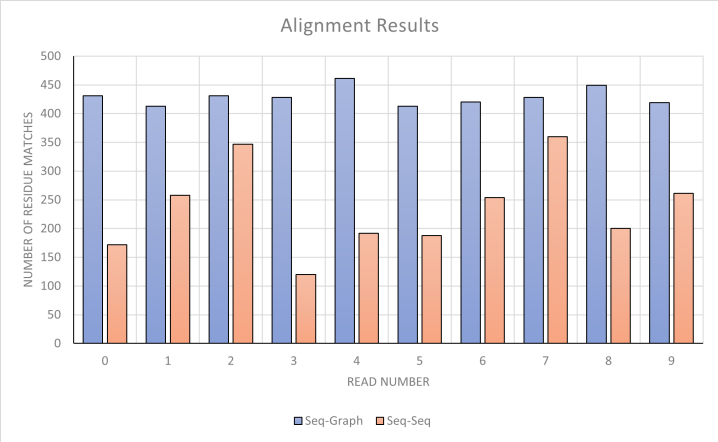


This looks little overwhelming and hence I am showing a small part of this

graph to show the underlying complexity:



Next, I used the built-in function of 'vg-toolkit' to simulate ten 500 nucleotide long reads and converted those reads to separate .fa files. Then I performed sequence-to-sequence alignment using one of the inital linear genomes and the reads followed by sequence-to-graph alignment using the generated graph and the reads, both using the built in functions of 'minigraph toolkit'. On completion of the alignment operations, I tabulated the number of *residue matches* in each of those alignment operations and plotted them. The results can be visualized using the given bar-chart.



From the above plot, we can very clearly see that there is a drastic difference in the number of residue matches when we use a sequence as the reference compared to a graph (because the sequence can't take into account the variation present in the genome). For all the reads, the graph based alignment ended up with significantly higher number of residue matches compared to the string based alignment.

This proves my hypothesis that graphs are better references compared to strings when it comes to Bioinformatics pipelines.

Github Repository: All the data and the codes used for the above experiment can be found here, with detailed instructions to reproduce the results.

References

[1] A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. *WADS 1997. Lecture Notes in Computer Science, Springer, Berlin*, 1272:160–173, 1997.

[2] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*, 32(7):1009–1015, 11 2015.

[3] S. Ballouz, A. Dobin, and J. A. Gillis. Is it time to change the reference genome? *Genome Biology*, 20(1), 2019.

[4] T. C. P.-G. Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 10 2016.

[5] S. Garg, M. Rautiainen, A. M. Novak, E. Garrison, R. Durbin, and T. Marschall. A graph-based approach to diploid genome assembly. *Bioinformatics*, 34(13):i105–i114, 06 2018.

[6] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, and M. Vechev. AStarix: Fast and Optimal Sequence-to-Graph Alignment. *Research in Computational Molecular Biology. RECOMB 2020, Springer, Cham*, 12074, 2020.

[7] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru. Accelerating sequence alignment to graphs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461, 2019.

[8] C. Jain, H. Zhang, Y. Gao, and S. Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020.

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

[10] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[11] M. Rautiainen and T. Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1), 2020.

[12] M. Rautiainen, V. Mäkinen, and T. Marschall. Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 35(19):3599–3607, 03 2019.