Universidade Federal de Pernambuco
Centro de Informatica

Bacharelado em Engenharia da Computação

Space-efficient representations for de Bruijn graph for data streams

Augusto Sales de Queiroz

Trabalho de Graduação

Recife
April 26, 2022

Universidade Federal de Pernambuco
Centro de Informatica

Augusto Sales de Queiroz

Space-efficient representations for de Bruijn graph for data streams

Trabalho apresentado ao Programa de Bacharelado em Engenharia da Computação do Centro de Informatica da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Paulo Gustavo Soares da Fonseca*

Recife
April 26, 2022

CHAPTER 1

Introduction

CHAPTER 2

State of the Art

CHAPTER 3

Fundamentação teórica e metodologia

3.1 de Bruijn graph

The de Bruijn graph is a directed graph for which each node represents a sequence of symbols, and each edge between two nodes represents the overlap between the two sequences. That is, given two nodes on the graph, they each represent a distinct sequence of symbols S_1 and S_2, and there is an edge between them if and only if the tail of S_1 is the head of S_2.

Within the context of genome sequencing, de Bruijn graphs are used in the assembly process by storing the distinct k-mers identified in the read sequences. In the ideal case (when each k-mer is present only once in the original sequence, and there are no sequencing errors), the complete traversal of this graph would produce the original sequence. In practice, such a straightforward approach is not feasible, but the de Bruijn graph can still be used to produce longer sequences, called *contigs*, which can then be processed to assemble the original genome. Figure 3.1 presents an example of the de Bruijn graph within this context.

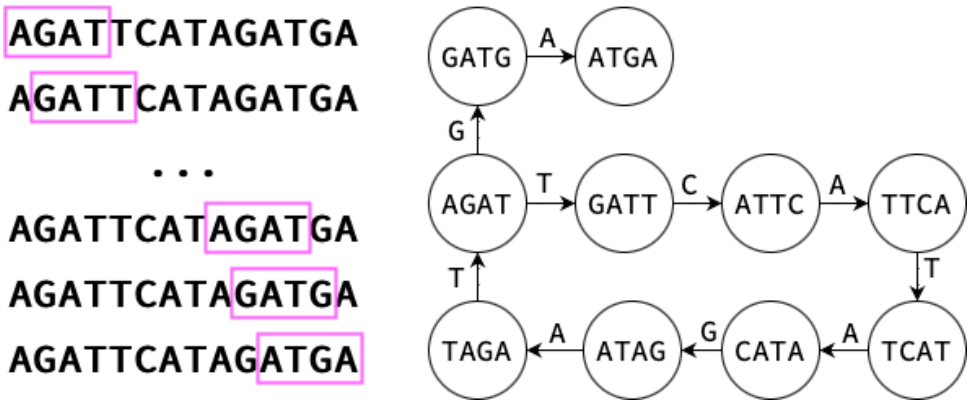

Figure 3.1 Example of a de Bruijn graph. $k = 4$

3.1.1 Representing a de Bruijn graph

Due to its nature, a de Bruijn graph can be represented by its set of nodes or edges independently, as one can be derived from the other. → asq says: Citation needed? ← As such, a structure that can answer queries about the presence of a given node on the graph is enough to represent the graph. In this regard, Conway & Bromage showed that a lower bound on the

number of bits required to *exactly* represent a de Bruijn graph exists, and is $\Omega(n \log n)$, for n being the number of distinct k-mers present in the graph, and $4^k > n$ [2].

In order to further improve space-efficiency, new forms of representation were created that XXX exactness for a probabilistic approach such as *Navigational Data Structures* (NDS), which have some probability of giving an erroneous answer to a membership query, but can be used to navigate the graph. This definition is useful due to the fact that a de Bruijn graph is not queried for the membership of randomly selected nodes, but rather only the neighborhood of a known member node is queried[1]. In the same paper where they introduce the idea of NDS, Chikhi *et al.* also present a lower bound for the number of bits needed to represent such a structure as $3.24n$. In sections 3.3 and 3.4 we will introduce two new NDS's.

3.1.2 Reverse Complements

One individuality of the genome sequencing context is the presence of *reverse complements*. When generating sequencing reads, a sequence of DNA can be read both in its forward form, or in its reverse complement. That is, the read is generated not from the original sequence S, but its complement $\overline{S}$, generated by swapping each base with its Watson-Crick complement ($\texttt{A} \leftrightarrow \texttt{T}$, $\texttt{C} \leftrightarrow \texttt{G}$). As in [2], this will be treated by processing all reads in both directions, without, however, merging nodes representing reverse complements. As noted by Conway & Bromage: "This makes the graph symmetric; a forward traversal corresponds to a backwards traversal on the reverse complement path, and vice versa."[2].

3.2 CountMin

The CountMin sketch, first introduced in [3], is a sub-linear data structure intended to allow for event frequency mapping. In this way, it must allow for the query of the frequency of a given event, as well as the update of that frequency, through the *query* and *update* operations. The sketch is composed of a W-wide, D-deep matrix of counters. With each row in this matrix is associated a hash function that map the possible events to the W positions in that row. All D hash functions must be pair-wise independent.

Updating the frequency of a given event is done by passing it through the hash functions for each row, and then updating the counter in the resulting position accordingly.

Querying the structure consists of, similarly, retrieving the value of the counter associated with the key in each row, and then returning the minimum value among them.

Figure 3.2 presents a visualization of the CountMin sketch.

3.2.1 As a representation for a de Bruijn graph

→ asq says: Vale colocar isso aqui, ou é mais interessante deixar apenas para falar disso no capítulo anterior (State of the Art), quando falando sobre o *FastEtch*? ←

A CountMin sketch can implement the membership query operation by querying the count for a given k-mer and comparing it to a presence threshold: if the count surpasses this threshold, the k-mer is considered to be present in the de Bruijn graph, and if the count is inferior to the

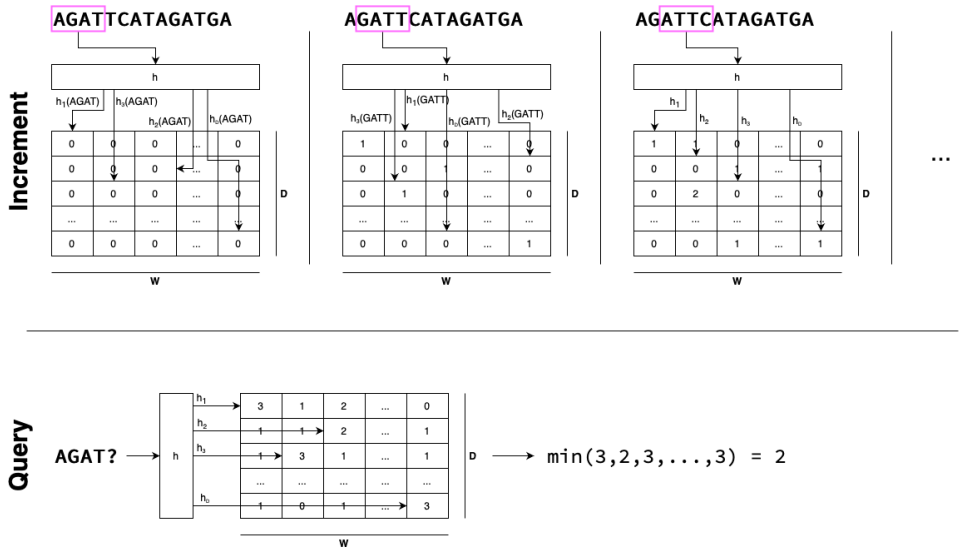

Figure 3.2 Example of a CountMin sketch being used to count k-mers. $k = 4$

threshold the k-mer is considerent absent from the graph. The MEMB function is defined in Algorithm 1.

Algorithm 1: MEMB

Data: k-mer; *sketch*, the CountMin sketch, *threshold*, the threshold for the presence of a k-mer in the de Bruijn graph

return $sketch.query(k\text{-}mer) \geq threshold$

3.3 de Bruijn CountMin

In order to leverage the benefits of using a CountMin representation of the de Bruijn graph while also improving its navigability, we introduce a modified version of the CountMin sketch, which we call the de Bruijn CountMin, that allows the sketch to be queried not only for k-mer counts, but also for the out-edges from the de Bruijn graph associated with that k-mer.

In order to store the additional information, we expand the CountMin sketch such that each cell in the matrix stores not only the counter, but also a set of out edges. The structure, then, provides an interface to increment the counters associated with a given key by one, and an interface to add an out edge to the sets associated with a given key. The increment operation is performed in the same way as in a regular CountMin sketch, and the algorithm for the add edge operation can be seen in Algorithm 2.

Furthermore, the de Bruijn CountMin must accomodate this new information in its query operation. In order to do this, the sketch returns not only the minimum value of the counters, but also the intersection of the sets of out edges. The algorithm for the updated query operation

is described in Algorithm 3.

Algorithm 2: *addOutEdge*(k-mer, *outEdge*)

Data: $outEdge \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$
for $i = 1, \ldots, D$ **do**
  $CountMin[i][h_i(\text{k-mer})].outEdges \leftarrow CountMin[i][h_i(\text{k-mer})].outEdges \cup outEdge$;
end

From a practical perspectie, due to a node only ever having 4 possible out edges (corresponding to the 4 bases $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$), the set of out edges can be represented by a bit vector indicating whether each of these possible edges is present. An edge is added by setting the corresponding bit, and the intersection is obtained by performing the bitwise AND operation. This allows both set of out edges and the counter to be stored together in a single integer.

Algorithm 3: *query*(k-mer)

$count \leftarrow inf$;
$outEdges \leftarrow \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$;
for $i = 1, \ldots, D$ **do**
  $count \leftarrow \min(count, CountMin[i][h_i(\text{k-mer})].count)$;
  $outEdges \leftarrow outEdges \cap CountMin[i][h_i(\text{k-mer})].outEdges$;
end
return $(count, outEdges)$

3.3.1 As a representation of a de Bruijn graph

As for the CountMin sketch, a de Bruijn CountMin sketch can be queried for membership by comparing the count associated with a given k-mer to a threshold for presence (i.e. the k-mer is considered present if it appeared more than T times in the reads). This structure can be navigated from an initial set of k-mers by querying for their out-edges and then querying each of their neighbors, repeating this when a neighbor is found to be in the graph. By storing the out-edges, we expect some neighbors to never be explored, reducing the chances of finding a false path on the graph.

3.4 Hashtable

We also propose a new hashtable-based representation for the de Bruijn graph that is made more efficient by not storing the k-mer. Instead, a fingerprint generated from the k-mer is stored, along with the set of out edges as described in Section 3.3. When a k-mer is inserted into the hashtable, or queried from it, a hash value and a fingerprint are calculated in parallel. In case of an insertion, the fingerprint is written at the desired position and, on a query, the fingerprints are compared. Collisions are resolved by linear probing, such that if a key tries

to insert in a position that is already occupied by a fingerprint that doesn't match its own, the k-mer is inserted in the next free position, unless its fingerprint is found before a free position is. During the query this process is repeated until the desired fingerprint is found, or a free position is reached (in which case the k-mer is considered to be absent from the structure).

This operation allows for the insertion of a node by adding the k-mer to the hashtable, and the insertion of an edge by updating the edge set associated with the given k-mer. When queried, the structure returns the edge set associated with the given k-mer, provided the k-mer has been added to the structure.

3.5 A pipeline using the de Bruijn CountMin and the de Bruijn Hashtable

Beyond the two isolated datastructures to represent the de Bruijn graph, we also propose a way to use both of them in tandem in order to obtain the benefits of both. In this pipeline, the sequencing reads are processed and inserted into the de Bruijn CountMin as they are made available, such that the de Bruijn graph can be constructed online. Once the reads are all processed in this manner, and a navigatable version of the graph has been constructed, it can be traversed, with all of its nodes being, then, inserted in a de Bruijn Hashtable. In this way, a de Bruijn CountMin is effectively compressed into a de Bruijn Hashtable.

3.6 Experiments

3.6.1 Metrics

Through the experiments described further in this section, we evaluate different metrics for the de Bruijn CountMin and the de Bruijn Hashtable. This is due to the fact that both these structures have different goals and are used in different contexts.

As both of these structures are probabilistic in nature, however, there are certain metrics that are used in the evaluation of both. One such metric is the *false positive rate*. In this work, we define this rate based on the k-mers that are visited during a traversal of the graph. Let S be a genetic sequence that contains the set of k-mers K, and let G be the graph, represented either by a de Bruijn CountMin or a de Bruijn Hashtable, constructed from the sequencing reads of S. Further, let Q be the set of k-mers that were queried from G during its traversal, and $P = k | k \in Q \wedge k \in G$ be the set of queried k-mers that were in G. As such, we can define the set of false positives as $F_P = P \setminus K$ (i.e.: the set of k-mers that were queried and found to be in G but are not actually in the original sequence S). Finally, the false positive rate is defined as $fp = \frac{|F_P|}{|Q|}$ (i.e.: the ratio of false positives to the total number of k-mers that were queried during traversal of G).

→ asq says: Realmente vale a pena tentar fazer isso ainda? Estamos na reta final do projeto já, e isso iria requerer a definição do que é uma mudança significativa nos resultados que justifique a mudança na estrutura ←

Further, as we posit that being able to answer neighborhood queries will improve navigabil-

ity of the graph by reducing the number of membership queries made and, thus, the number of false positives, we perform two traversals of the de Bruijn graph, the first without using the out edges information (i.e.: for every node that is in the graph, we query all possible neighboring nodes), and the second with that information (i.e.: only recorded out edges are used to expand the frontier). We, then, compare the different metrics for the two traversals.

3.6.1.1 de Bruijn CountMin

The de Bruijn CountMin was developed to be used directly with the sequencing reads without any pre-processing. It's goal is to build a reliable navigatable version of the de Bruijn graph as the reads are made available. In this context, not only do we expect a certain amount of false positives will appear, but as we must probabilistically filter the set of k-mers from the reads to remove all the spurious ones, we also expect the occurence of *false negatives*, which are defined as k-mers from the original sequence S that are not present in the graph G. I.e.: $F_N = K \setminus P$. As such, the *false negative rate* can be defined as the ratio of false negatives to the total number of k-mers in the original sequence S, or $fn = \frac{|F_N|}{|K|}$.

3.6.1.2 de Bruijn Hashtable

3.6.2 *E. Coli*

The *E. Coli* genome is an established benchmark for new assemblers to compare against.

We used the reference genome for the *E. Coli* bacterium available in `http://ftp.ensemblgenomes.org/pub/bacteria/release-52/fasta/bacteria_0_collection/escherichia_coli_str_k_12_substr_mg1655_gca_000005845/dna/`

Three different experiments were performed. In the first we generated simulated perfect reads from the genome by taking substrings of the original sequence at random. We then used the *ART Illumina* toolkit → asq says: Citation needed ← to simulate realistic reads from this genome, including read errors and reverse complements. Finally, a dataset of real-world reads was downloaded from SRA → asq says: Citation Needed ← and used.

3.6.2.1 Synthetic reads without errors

Given the reference genome S, the length of each read, L, and the desired coverage C, the synthetic reads were generated by picking $\frac{|S| \times C}{L}$ substrings of S at random. This is presented in algorithmic form in Algorithm 4.

3.6.2.2 Synthetic reads with errors

In order to simulate the reads as they would be produced by the sequencing process, we used the ART Illumina toolkit to generate synthetic reads from the *E. Coli* genome. The reads were generated using the following parameters:

1. **Sequencing System**: Illumina MiSeq v3
2. **Read length**: *250bp*

Algorithm 4: Generate Reads

Data: S, the reference genome, L the read length, C the coverage
$\#reads \leftarrow \frac{|S| \times C}{L}$;
$reads \leftarrow \emptyset$;
for $i \leftarrow 1, \ldots, \#reads$ **do**
    $j \leftarrow random(0, |S| - L)$;
    $reads.add(S[j : j + L])$;
end
return *reads*

3. **Coverage**: *80x*

CHAPTER 4

Results

CHAPTER 5

Conclusion

Bibliography

[1] R. Chikhi, A. Limasset, S. Jackman, J. Simpson, and P. Medvedev. On the representation of de bruijn graphs. 1 2014.

[2] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27:479–486, 2 2011.

[3] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55:58–75, 4 2005.