Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Engenharia da Computação

Space-efficient representation of de Bruijn graphs for data streams

Augusto Sales de Queiroz

Trabalho de Graduação

Recife
May 24, 2022

Universidade Federal de Pernambuco
Centro de Informática

Augusto Sales de Queiroz

Space-efficient representation of de Bruijn graphs for data streams

Trabalho apresentado ao Programa de Bacharelado em Engenharia da Computação do Centro de Informática 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
May 24, 2022

Acknowledgements

I am grateful to my parents, José Antônio and Andrea, who have always shown me love and support, and for all the help I received in the course of my studies, not only as parents but also as professors with a lifetime of experience. To my brother, Antônio, as well, who has been a good friend, specially in pandemic times, and has been an inspiration in achieving one's goals and dreams. I also thank my girlfriend, Rafaella, who has been a loving supporter for the past five and a half years, always by my side and helping me get through life, academic and otherwise. I also thank my family, in particular my godparents Amaro and Isabel and my uncle Fernando, who have also always been by my side. I am also grateful to my grandfather, Fernando, who didn't get to see me progress through university but was always a strong supporter of my academic efforts and intellectual pursuits, and would surely be among the readers of this work.

I thank the friends I made through the university, in special Igor, Rodrigo, and Tancredo, with whom I have shared many of the experiences of higher education and professional development, and who have helped me countless times when studying, coming up with projects, and debugging code. As well as Guilherme, whose help in finding an internship put me on the path to choosing this subject matter for my graduation project.

Beyond university I've also been fortunate to have a group of close friends who have been close to me and shared in many of the important moments in life, Bruno, Amanda, Túlio, Carol, Humberto, Mariana, Ruben, and Juliana.

I am grateful to my advisor, Professor Paulo Gustavo da Fonseca, who has been a great guide and teacher both in the subject of genome assembly and succint datastructures, as well as in the general matter of scientific research, and who has worked with me closely, including hours-long virtual meetings and on weekends, to help this work come to fruition.

I also thank Professor Edna Barros, who was a mentor through 2017 and 2018 as I worked on the embedded systems competition with Igor and Rodrigo, and who helped me to get the BRAFITEC exchange.

I am also grateful to the Federal University of Pernambuco (UFPE), that has been a second home since 2008, when I joined Colégio de Aplicação and since then has been an integral part of my personal, academic, and professional development.

Finally, I would like to thank Frédéric Cazals and Dorian Mazauric, from Inria Sophia Antipolis, thanks to whom I've had my first contact with scientific reseach and the field of bioinformatics, and who thaught me so much over the course of a year of internship.

I thank CAPES for the financial support offered for my exchange to Polytech Nice-Sophia through the BRAFITEC program.

Mini cupcakes? As in the mini version of regular cupcakes? Which is already a mini version of cake? Honestly, where does it end with you people?

—KEVIN MALONE (The Office, Season 9 Episode 9)

Abstract

Background Since the introduction of the second generation of DNA sequencing technologies, improving assembly efficiency has become critical to match the high-throughput of data. A major bottleneck is the memory needed to represent de Bruijn graphs, an essential structure in modern assemblers. Improving this representation involves finding a memory-efficient way of representing a set of *k*-mers, which are sequences of a fixed length *k*, as well as filtering out sequencing errors from the input reads generated by the sequencers.
Objectives In this work we address these two problems to develop a de Bruijn graph representation that faithfully represents the *k*-mers in the sequenced genome while leaving out spurious parts affected by sequencing errors. Not only we want our representation to have a good sensitivity to specificity ratio, but we also strive to make it space and time-efficient.
Methods We propose and implement a pipeline based on two new probabilistic representations of the de Bruijn graph, the CountMin-based DBCM and the hashtable-based DBHT, which aims at filtering *k*-mers obtained from the reads based on frequency and connectivity to other *k*-mers, and representing only high-quality *k*-mers in a succint manner. We then applied out method to a realistic dataset based on a microbial genome.
Results We show that DBCM can be used to effectively filter over 80% of spurious *k*-mers from the reads based on count alone, and can then be traversed to further remove erroneous *k*-mers based on connectivity, filtering over 95% of spurious *k*-mers in total. We then represent the remaining *k*-mers in a DBHT, requiring between 9 and 16 bits per *k*-mer, while allowing for constant time insertion and query. We show that even with 9 bits per *k*-mer, the number of false positive *k*-mers is still less than 16% of the total graph.
Conclusion We conclude that connectivity-based filtering through traversal is very effective in filtering spurious *k*-mers from the reads beyond frequency-based filtering and can efficiently be done by the DBCM. We also achieve a very good compromise of memory usage and insertion and query time with the DBHT by using a single hashing function.

Keywords: de Bruijn graph, *k*-mer, genome assembly, sequencing, filtering, sketch, CountMin, Hashtable

Resumo

Contexto Desde a introdução das tecnologias de sequenciamento de DNA de segunda geração, melhorar a eficiência da montagem de genomas tornou-se crítico para fazer face ao alto volume de dados. Um relevante gargalo é a quantidade de memória necessária para representar um Grafo de de Bruijn (GdB), uma estrutura essencial dos montadores de genomas modernos. Melhorar essa representação envolve encontrar uma maneira de representar um conjuntos de *k*-mers, que são sequências de comprimento fixo *k*, assim como filtrar os erros de sequenciamento das leituras geradas pelos sequenciadores.
Objectivos Neste trabalho nós abordamos estes dois problemas para desenvolver uma representação de GdB que representa fielmente os *k*-mers do genoma sequenciado, à medida que deixa fora trechos espúrios afetados por erros de sequenciamento. Não apenas queremos que nossa representação tenha uma boa relação entre sensibilidade e especificidade, como também esforçamo-nos para fazê-la computacionalmente eficiente em tempo e memória.
Métodos Nós propomos e implementamos uma *pipeline* baseada em duas novas representações probabilisticas para GdB's, o DBCM baseado em um *sketch* CountMin e o DBHT baseado em uma hashtable, que tem como alvo filtrar os *k*-mers obtidos das leituras baseado em frequência e conectividade com outros *k*-mers, e representar apenas os *k*-mers de alta qualidade de forma sucinta. Nós então aplicamos nosso método a um conjunto de dados realistas baseado no genoma microbiano.
Resultados Nós mostramos que o DBCM pode ser usada para filtrar efetivamente mais de 80% dos *k*-mers espúrios das leituras baseado em contagem apenas, e que pode então ser navegada para remover parte dos *k*-mers errôneos restantes, filtrando mais de 95% dos *k*-mers espúrios no total. Nós então representamos os *k*-mers restantes em uma DBHT, que requer entre 9 e 16 bits por *k*-mer, dependendo da taxa tolerada de falsos positivos, enquanto permite a inserção e consulta em tempo constante. Nós mostramos que, mesmo com 9 bits por *k*-mer, o número de nós falsos positivos constitui menos do que 16% do grafo resultante.
Conclusão Nós concluímos que a filtragem baseada em conectividade através do percurso do grafo é muito efetiva para filtrar *k*-mers espúrios das leituras para além da filtragem baseada em frequência, e pode ser realizada eficientemente na DBCM. Também alcançamos um bom compromisso entre uso de memória e tempo de inserção e consulta com o DBHT mediante o uso de apenas uma função de hashing.

Palavras-chave: Grafos de de Bruijn, *k*-mer, Sequenciamento Genômico, filtragem, *sketch*, CountMin, Hashtable

Contents

List of Figures

CHAPTER 1

Introduction

Determining the genomic sequence of a given organism is of interest to many biological sciences domains, such as Medicine and Agriculture [15, 13]. However translating a DNA molecule into the sequence of nucleic acids that constitute is not straightforward. Sequencing technologies can, at best, produce overlapping subsequences of the original genome, known as sequencing ***reads***, whose sizes vary according to the protocol used. For Whole Genome Sequencing (WGS), those sizes are much smaller than the genome being sequenced [20]. We need therefore to 'stitch' them together to reconstruct the original sequence. Hence the ***de novo genomic sequence assembly*** is the problem of reconstructing the shortest string that contains the reads as substrings. This problem, however, is known to be *NP-Hard* [9]. Furthermore, because of the presence of repeats in the original sequence, i.e. subsequences that appear more than once in different positions, the success of this method of assembly is limited by the size of the reads. This limitation became even more pronounced with the move from first generation sequencing technologies, that produced reads of 1000bp[1] in length, to second generation sequencing, producing reads with length between 25 and 400bp [26]. The second generation sequencing also introduced a higher chance of error associated with reading any given base, making the assembly problem even more difficult. For instance Illumina technologies were reported to have a per-base error rate between 0.1% and 1% in 2010 [19] *versus* first generation technologies achieving 99.999% per base accuracy [26]. We thus have to filter the reads to remove erroneous pieces.

In 2001, Pevzner *et al.* [22] introduced an assembler that tackled all of these difficulties by further breaking the reads into smaller k-length pieces, called k-mers and using them to construct a special kind of subsequence graph, named ***de Bruijn graph*** after its creator. Now, because the reads were divided into smaller pieces, the ones containing sequencing errors could be removed without losing the entire read. Moreover, by using a de Bruijn graph, the authors showed that the assembly process could be transformed from finding the shortest super string, known to be computationally difficult, to finding an Eulerian superpath on the graph that contained the paths representing the different reads. Because the Eulerian *path* problem can be solved in polinomial time, it was hoped that such a solution would also exist for the *superpath* problem. Medvedev *et al.* later showed, however, that the Eulerian superpath approach on the de Bruijn graph is, itself, *NP-hard* as well [17, 14].

The approach introduced by Pevzner *et al.* has proven to be, in practice, an efficient framework for genome assembly, such that the de Bruijn graph became a standard part of many assemblers, and reducing its memory space to make it fit entirely in memory became a pressing issue [5]. Since the publication of the original paper by Pevzner *et al.* [22], many succinct

[1]bp = base pairs. The number of bases $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ that compose the genome or genome fragment.

representations of the de Bruijn graph have been developed, aiming at reducing the number of bits per represented k-mer. In 2011, Conway & Bromage showed that any de Bruijn graph representation capable of answering a k-mer query exactly needs at least $O(n \log n)$ bits, with n being the number of distinct nodes in the graph [7]. By foregoing deterministic exactness and allowing some proportion of k-mers to be erroneously considered as represented in the graph, Pell *et al.* created a representation that needed only 4 bits per k-mer [21]. Shortly afterwards, Bowe *et al.* and Chikhi & Rizk independently observed that, because of the limited way in which de Bruijn graphs are used in assembly, they could represent the graph using a structure that, albeit not capable of exactly deciding if a given k-mer is represented or not, can exactly determine the neighbors of a given node known to be in the graph [3, 6].

Besides reducing the bits needed to represent a k-mer in the de Bruijn graph, it also became important to process the reads to remove the k-mers created by sequencing errors, effectively reducing the number of distinct k-mers represented in the graph. One ubiquitous approach consists in counting the number of times each distinct k-mer appears in the reads [28]. Any piece of the original genomic sequence is expected to be represented a number of times in the reads, called the sequencing ***coverage***, usually in the range of tens to a few hundred, while erroneous fragments are expected to appear only a few times. Thus the k-mers obtained from the reads can be categorized as high- or low-frequency, with the latter likely being spurious [7, 10]. Counting k-mers presents its own difficulty in terms of memory usage, as the number of spurious k-mers can be much larger than that of real k-mers [7, 18, 28, 10].

In this work, we will introduce a methodology for tackling the problem of improving the space-efficiency of de Bruijn graph from two sides. First we introduce a new representation based on a CountMin sketch [8] that can be constructed directly from the unprocessed reads, filtering out low-frequency k-mers. By traversing this structure, we can further identify and filter spurious k-mers. We also introduce a mechanism to improve traversal effectiveness by reducing the chance of false k-mers being visited. We show that, although the CountMin-based representation introduces count errors, it is still successful in removing most (over 80%) of the spurious k-mers from the reads based on frequency alone. We also show that the remaining 20% are further filtered out by traversing the graph, leading to only a relatively small number of false k-mers being represented in the graph, less than 2% of the graph. This allows us to insert almost exclusively high-quality k-mers into a novel probabilistic hashtable-based representation of a de Bruijn graph, using as few as 9 bits per k-mer. We then show that this new data structure introduces some new false k-mers, but they never exceeded 16% of the k-mers represented in the graph in our experiments.

1.1 Manuscript outline

The rest of this monograph is organized as follows.

In Chapter 2 we present the basic theoretical foundations underlying our work. We also review relevant related work that motivated this project.

In Chapter 3 we describe our implemented method in detail, including each algorithm and data structure in separate, and how they fit together.

In Chapter 4 we present an experimental evaluation of our method applied to a realistic synthetic dataset based on a microbial genome.

In Chapter 5 we recapitulate and summarize our findings, and indicate a few directions for future developments.

CHAPTER 2

Theoretical Foundation and Related Work

In this chapter we begin by setting the theoretical groundwork for understanding de Bruijn graphs in the context of genome assembly. We then discuss some related work on k-mer counting as well as succinct representations of the de Bruijn graph.

2.1 Notation

Throughout this work, we use lower-case symbols, such as t and c, to represent scalar values. Uppercase symbols, such as X and Y, represent strings. Finally, calligraphic uppercase symbols, such as $\mathscr{X}$ and $\mathscr{S}$, are used to represent sets. Strings and sets are 0-indexed, and the notation $X[i:j]$ represents the substring of X from position i up to, but not including, position j. The concatenation of two strings X and Y, or a string X and a character a, are respectively denoted by $X \cdot Y$ and $X \cdot a$.

2.2 de Bruijn graphs

The ***de Bruijn graph of order*** k of a string $X = x_0 \cdots x_{n-1}$, $G(X;k) = (V,E)$, is defined as the directed graph whose nodes V represent all distinct k-mers (i.e. substrings of length k) of X, and such that any two nodes representing $(k-1)$-overlapping k-mers are connected by an edge labeled by the last character of the second k-mer. For example if $k = 3$ and X contains the substring `ACGT` then the consecutive triplets `ACG` and `CGT` will originate the edge $\texttt{ACG} \xrightarrow{\texttt{C}} \texttt{CGT}$. Hence the edges E represent all distinct $(k+1)$-mers of X.

In genome sequencing, de Bruijn graphs are used in the assembly process to represent the distinct k-mers in a set $\mathscr{X}$ of randomly distributed fragments of the source DNA S, called ***reads***, generated by the sequencing machines. The amount and total length of the reads is determined by the ***coverage*** of the sequencing process, i.e. the number of times S was cloned and sequenced. Ideally, S could be obtained from an Eulerian traversal of $G(\mathscr{X},k)$. Unfortunately however, due to sequencing errors and repeats, such a straightforward approach is not feasible, but the de Bruijn graph can still be used to produce a collection of partial assemblies, called ***contigs***, which can then be further combined to form the original genome [22]. Figure 2.1 presents an example of the de Bruijn graph in this context.

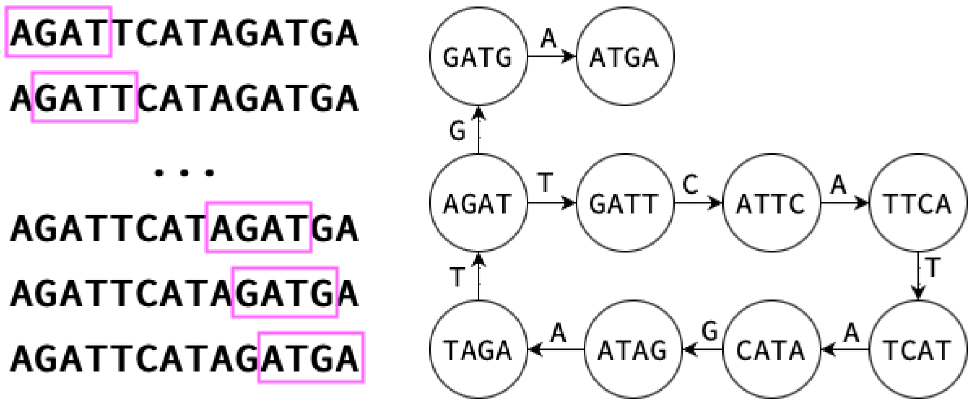


Figure 2.1: Example of a de Bruijn graph. $k = 4$

2.3 Reverse Complements

One difficulty of using de Bruijn graphs to represent DNA data is the presence of ***reverse complements***. When generating sequencing reads, the machine reads either of the two complementary strands of a fragment of the input DNA. That is, the output read may correspond to the sequence S in the forward (5'-3') direction, or its reverse complement $\overline{S}$ in the backward (3'-5') direction, with $\overline{S}$ being obtained from S by swapping each base with its Watson-Crick complement (A $\leftrightarrow$ T, C $\leftrightarrow$ G) and then reversing the string, and vice versa. For example, the reverse complement of the sequence $S =$ AGTACGGTC is $\overline{S} =$ GACCGTACT and vice versa.

To deal with reverse complements, reads are processed twice, once in each direction. Nodes representing k-mers that are reverse complements of each other are merged, with edges made bidirectional. Alternatively, those nodes can be kept distinct, resulting in a symmetric graph in which, as noted by Conway & Bromage, “a forward traversal corresponds to the backwards traversal on the reverse complement path, and vice versa.” [7]

2.4 Selecting the k-mers for the de Bruijn graph

Another difficulty of using a de Bruijn graph to represent DNA data is dealing with sequencing errors. During the sequencing process, there is a small error rate associated with reading any base from S (0.1%–1% per base in Illumina [19]). As a result, many of the k-mers extracted from the reads in $\mathscr{X}$ are erroneous. As discussed by Conway & Bromage, this causes the number of spurious k-mers to grow proportionally to the number of bases in the reads $|\mathscr{X}| = \sum_{X_i \in \mathscr{X}} |X_i|$, determined by the coverage, while the number of true k-mers is proportional to the size of the genome $|S|$ [7].

Although modern sequencing machines generate a quality score associated with each base, it is not possible to know exactly what parts of the reads are erroneous, and so we cannot tell whether a k-mer should be added to the de Bruijn graph without more information. However, real k-mers are expected to appear a number of times close to the sequencing coverage c, or a multiple of c in case of repeats, whereas spurious k-mers resulting from sequencing errors are commonly low-frequency or even unique [7, 28, 10]. Therefore, a natural way of filtering the k-mers obtained from the reads is to discard those that have a low frequency. Hence we consider a k-mer to be a real k-mer *iff* it occurs more than some threshold t, in which case we add it to the de Bruijn graph.

2.5 Counting k-mers

Counting k-mers in a set of reads in a time- and space-efficient manner is a challenging task, commonly performed as a preprocessing step to the de Bruijn graph construction [28], and for which many different approaches have been proposed.

Marçais & Kingsford introduced *Jellyfish* in 2001, a hashtable-based approach for k-mer counting that controls memory usage by identifying when the hashtable will grow beyond the available memory, in which case its contents are written to disk and the table cleared. It can, then, merge the partial hashtables saved in disk storage [16].

BFCounter, introduced by Melsted & Pritchard, also uses a hashtable to track k-mer counts, but it uses a Bloom Filter (BF) to skip unique k-mers. It does two passes over the sequencing reads, identifying the k-mers that appear more than once by verifying if they were already introduced into a BF on the first pass, and then using a hashtable to count only the nonunique k-mers on the second [18].

Rizk *et al.* later introduced *DSK*, which makes use of disk storage to track k-mer counts thus saving main memory. The tool allows for controlling how much disk storage is used by sacrificing some processing time. It separates the k-mers from the reads into a set of files whose number and size is determined by the target memory and disk usage given as an input. The k-mers in each file are then counted using a hashtable [23].

Similar to *BFCounter*, *Turtle* uses a BF to track k-mers appearing at least twice, adding k-mer occurrences to fixed-length array. Once the array is filled, it is sorted such that occurrences of the same k-mer appear next to each other. These occurrences can then be merged into a single array entry containing the number of times that k-mer occurred, freeing memory for the process to continue. For cases where the number of frequent k-mers is too large to fit into the array, they also introduce a probabilistic approach using counting BF's [24].

Zhang *et al.* use a CountMin sketch (discussed in Section 3.2.1) as a probabilistic approach to k-mer counting that incurs in some miscount rate. They show that this rate can be controlled by increasing the memory requirements of the sketch. They also show that, despite the miscount rate, the abundance profiles generated by *khmer* could be used for measuring sequencing error profiles and for performing read trimming [28].

It is also possible to introduce k-mer counting directly as a part of the de Bruijn graph construction. *FastEtch* is an assembler which uses a small CountMin sketch to allow for the online construction of the de Bruijn graph by filtering spurious k-mers based on abundance. As k-mers are taken from the reads, they are used to update counters on the CountMin sketch. Once a threshold is reached, the k-mer is added to a hashtable-based exact representation of the de Bruijn graph. Because the authors use a small CountMin sketch, many collisions are expected to occur and many false positives are expected to go through. They propose two methods for reducing these occurrences which are based on keeping track of how many k-mers have been added to the de Bruijn graph due the count of any individual bucket of the CountMin. Once a bucket has caused too many k-mers to be inserted into the graph, it is reset. They present two variations of an assembler that offer a good time-memory-quality tradeoff [10].

2.6 de Bruijn graph representation

A de Bruijn graph can be represented either by its set of nodes (k-mers) or edges ($(k+1)$-mers) equivalently, as one can be derived from the other. Therefore, a structure that can determine if a given node x is a member of G can represent the de Bruijn graph. Conway & Bromage showed that the lower bound on the space required to *exactly* represent these ***membership data structures*** (MDS's) is $\Omega(n \log n)$, with $n = |V(G)|$ [7]. They achieve a succinct representation of the graph by storing its edges in a compressed bitmap, allowing the graph to be represented using approximately 28 bits per edge [7].

In order to further improve space-efficiency, new representations were created that trade deterministic exactness for a probabilistic approach. Pell *et al.* showed that a probabilistic representation based on a BF could accuratly represent a de Bruijn graph with as little as 4 bits per k-mer while keeping false positive rates at 15% or lower [21].

However, Bowe *et al.* [3] and Chikhi & Rizk [6] independently observed that giving up deterministic exactness isn't the only option for obtaining better space-efficiency, introducing two new representations that use $O(n)$ and $O(n \log k)$ bits, respectively, and allow for an exact traversal of the de Bruijn graph from a starting set of known member nodes. This is possible due to the fact that a de Bruijn graph is not queried for membership of random nodes, but rather for potential neighbors of nodes already known to be in the graph. As such, the structures designed by the two groups do not offer a deterministically exact membership query operation, instead offering a neighborhood query operation that is exact for the members of the graph [3, 6]. Chikhi *et al.* later named this new form of representation a ***Navigational Data Structure*** (NDS) and showed that the lower bound on the number of bits required to exactly represent a de Bruijn graph with an NDS is $3.24n$ [5].

Chikhi & Rizk's implementation follows Pell *et al.*'s approach [21] by using a BF to store the nodes in the graph. However, they use a separate standard set to exactly store only the false positive neighbors of true k-mers, which they call *cFP* for critical false positives, such that a k-mer is considered to be represented in the graph iff it is considered present in the BF, but not in the *cFP* set. This allows them to construct an exact representation of the de Bruijn graph in 13.2 bits per k-mer [6].

Salikhov *et al.* expanded on the work of Chikhi & Rizk by storing the *cFP*'s in a BF. Because this can incur in some true positives being considered false positives, they use a second BF to store these 'false false positives'. They show that, in theory, they could achieve a representation of the de Bruijn graph using 8.45 bits per k-mer using an infinite set of BF's. In practice, however, they use a finite set of filters, with a set of *cFP*'s that are not resolved by those filters. They show that using just two filters is enough to resolve 95% of queries without using the set, and four filters resolve over 99% of queries. This allows them to improve on the results by Chikhi & Rizk [6] by constructing a representation that needs only between 8.5 and

9 bits per k-mer [25].

In the same paper where they establish the lower bound on space-efficiency for NDS's, Chikhi *et al.* introduced *DBGFM*, which represents the de Bruijn graph not as the set of k-mers individually, but through the set of maximal simple paths on the graph. A naive approach for storing the set of paths would provide an extremelly succint representation, but at the cost of query time. The authors, however, achieve a good tradeoff between the two resources by using an FM-index to speed up queries. The resulting structure can still represent a de Bruijn graph in as few as 3.53 bits per k-mer. To avoid using a larger intermediate representation of the de Bruijn graph to perform the construction of the maximal simple paths (called compaction) on, the authors introduce a new algorithm, *BCALM*, to produce the maximal simple paths using low-memory. This incurs in an upfront time cost for the compaction of the de Bruijn graph. Furthermore, this compaction is achieved in low-memory by making use of disk storage [5].

Beyond the distinction between probabilistic and exact, and MDSs and NDSs, there is also a distinction between ***static*** and ***dynamic*** representations for de Bruijn graph. Static representations are constructed once and never altered, whereas dynamic representations support operations such as insertion and deletion of nodes, useful in population-scale studies due to their everchanging nature [2]. Such structures fall beyond the scope of this work, however.

2.7 de Bruijn graph operations

Chikhi *et al.* [4] present the following set of operations common to many data structures for representing a de Bruijn graph.

1. *Construction* of the data structure
2. *Insertion* of a new k-mer
3. *Deletion* of an existing k-mer.
4. *Membership query*: Given a k-mer X, returns true *iff* $X \in G$
5. *Forward neighbor query*: Given a k-mer X and a base a, returns true *iff* $X[1:k] \cdot a \in G$
6. *Backward neighbor query*: Given a k-mer X and a base a, return true *iff* $a \cdot X[0:k-1] \in G$.

Note that a representation that implements the *construction* and *membeship query* operation can use those to implement some version of the others (e.g. by reconstructing the graph with the desired insertion or deletion, or by querying for the membership of the desired neighbor). However, dynamic representations implement *insertion* and *deletion* operations that do not require the reconstruction of the graph, while NDSs don't rely on the membership query to establish if a given forward or backward neighbor is represented in the graph. For example, the neighborhood query described by Chikhi *et al.* [5] as defining NDSs can be implemented by performing *forward* and *backward neighbor queries* for all extensions of the given k-mer and then returning only those for which the result was *true*.

CHAPTER 3

Methods

In this chapter, we detail a novel pipeline for constructing a succint de Bruijn graph representation directly from the unprocessed sequencing reads, and two new de Bruijn graph representations. The first is capable of performing k-mer counting and count-based filtering by sacrificing space efficiency, while the other is a succint hashtable-based representation. Both these data structures offer probabilistic membership query operation and, by associating each k-mer with a set of outedges, a probabilistic neighborhood query that is expected to perform better than using the membership query for all possible neighboring k-mers.

3.1 A navigational data structure-based pipeline for online construction of de Bruijn graphs

We propose two new probabilistic NDS-based representations for the de Bruijn graph: the ***de Bruijn CountMin*** DBCM (Section 3.2), and the ***de Bruijn Hashtable*** DBHT (Section 3.3). The DBCM aims at performing online k-mer counting in a structure that can also be traversed through the use of a probabilistic neighborhood query in a space-efficient manner. It is constructed online as the reads are processed, and can be navigated, allowing for further processing of the de Bruijn graph. The DBHT forgoes k-mer counting and any form of filtering in exchange for improved space-efficiency in a hashtable-based representation. Both structures are designed for improved navigability on the graph by storing a set of outgoing edges (outedges, for short) with each k-mer, allowing for constant time neighborhood queries.

These two structures can be used in tandem to leverage the benefits of each one individually. We construct the DBCM directly from the reads, without preprocessing, such that the reads can be treated as a data stream and added into the structure as soon as they are made available. During processing, we store in disk a set of possible starting k-mers $\mathscr{S}$ taken from the reads. Once all the reads have been processed, we traverse the DBCM starting from the k-mers in $\mathscr{S}$, inserting the visited nodes and their outedges into a DBHT. This pipeline can be seen in Algorithm 1.

Because spurious k-mers happen in sequence (e.g. a single erroneous base is likely to cause the k k-mers that include it to become erroneous), they create branches off of the graph generated by the real k-mers. If a single k-mer of such a branch is identified as spurious and not included in the graph, however, it causes all subsequent k-mers to be unreachable through traversal. As such, k-mers that would be included in the de Bruijn graph based on frequency alone may be excluded when constructing the DBHT from traversal due to a predecessor being correctly filtered out.

Algorithm 1: Pipeline

```
Input  : $\mathscr{X}$, a set of sequencing reads; $t$, the frequency threshold for a $k$-mer to be
         considered true; $n$ the number of starting $k$-mers for traversal to take from the
         reads
Output: The DBHT representation of the de Bruijn graph
$C \leftarrow$ an empty DBCM
$\mathscr{S} \leftarrow C.construct(\mathscr{X}, t)$                    ▷ see Section 3.2.4
$T \leftarrow$ an empty DBHT
$T.construct(C)$                                  ▷ see Section 3.3.3
return $T$
```

3.2 The de Bruijn CountMin sketch

In order to leverage the benefits of an NDS in reducing the impact of the false positive rate of the membership operation, we introduce a modified version of the original CountMin sketch, presented below, called ***de Bruijn CountMin*** (DBCM) allowing for querying not only for k-mer counts, but also for the outedges from the corresponding nodes in the de Bruijn graph. As such, the DBCM implements not only a membership query operation as detailed in Section 3.2.3, but also a neighborhood query.

3.2.1 The CountMin sketch

The CountMin sketch [8] is a sublinear data structure for event count estimation. Given a stream of events $\mathscr{X} = \{(x_i, c_i) | i \in \mathbb{N}\}$, where $c_i \geq 0$ is the number of occurrences of x_i, it offers two basic operations

1. $update(x, c)$, and
2. $query(x)$,

respectively for informing c occurrences of an event x, and for obtaining an estimate of the accumulated frequency of event x up to that point.

The sketch is composed of a $d \times w$ matrix of counters C, and a set of d pairwise-independent hash functions $h_0 \dots h_{d-1}$, such that each h_i maps an event to one of the w cells in row i. Updating an event is done by passing it through the hash functions for each row, and then incrementing the counters in the mapped positions $h_i(x)$ accordingly. We consider the simpler case where we only report individual event occurrences, so that counters are always incremented by one. Querying the structure consists in retrieving the value of the counters associated with the key event in each row, and then returning the minimum value among them, that is $\min\{C[i, h_i(x)]; i = 0 \dots d-1\}$. Figure 3.1 presents a visualization of the CountMin sketch.

It is important to notice that the CountMin might overestimate the frequency of an event due to hash collisions, since more than one event might be mapped to the same cell of the matrix. However, the pairwise independence requirement ensures that, for any row i and any two distinct events $x \neq x'$,

$$\Pr[h_i(x) = h_i(x')] = \frac{1}{w},$$

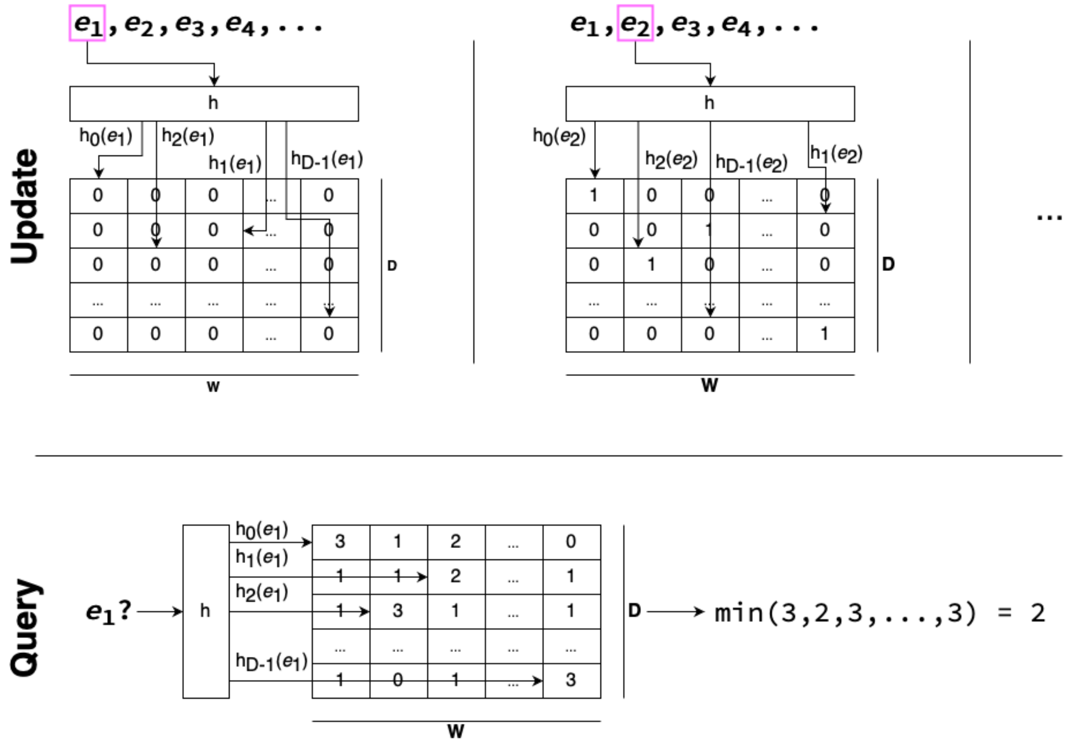


Figure 3.1: Example of a CountMin sketch

with this probability being computed over all possible choices of the hash function. Hence, by setting w appropriately, we can control the expected number of collisions per row, and likewise, by choosing a sufficiently large d, we can control for the probability of having at least one row without collisions for a given event. In general, for two given parameters $\varepsilon, \delta \in (0,1)$, by setting $w = O(2/\varepsilon)$ and $d = O(\log 1/\delta)$, we have

$$\Pr[C.query(x) - c(x) > \varepsilon \|\mathscr{X}\|_1] < \delta, \tag{3.1}$$

where $c(x)$ represents the true frequency of x, and $\|\mathscr{X}\|_1 = \sum_{x_i} c(x_i)$ is the true total event count. Put another way, we can have the estimate error relative to the total event count to be ‘small’ (no more than ε) with ‘good’ (at least $1-\delta$) probability.

3.2.2 Using the CountMin for counting k-mers

Zhang *et al.* proposed using a CountMin sketch to count k-mers [28]. Each k-mer is treated as an event, with reads being processed as a stream of k-mers. The k-mers can be hashed by interpreting them not as a string, but as a base-4 integer, with $\texttt{A} = 0$, $\texttt{C} = 1$, $\texttt{G} = 2$, $\texttt{T} = 3$. For example, the k-mer `CGATA` can be interpreted as the integer $12030_{(4)} = 396_{(10)}$. Whenever we refer to hashing a k-mer, we use its integer interpretation. Using a CountMin introduces the chance that any reported count will be an overestimate, the probability of which is approximately $(1 - e^{-\frac{N}{w}})^d$, with N the number of distinct k-mers [28].

Because the k-mer counts are used to determine if a k-mer should be added to the de Bruijn graph, as discussed in Section 2.4, a structure that can answer count queries can be extended to implement the membership query operation, with a k-mer X being considered to be represented in the graph if $C.query(X).c \geq t$. Assuming that the vast majority of low-frequency k-mers occur no more than once or twice in the input reads, in order for such a k-mer to be unduly considered as real (false positive), its CountMin estimate would have to be off by about t or more. Because the CountMin overestimation error is at most εF with probability at least $1-\delta$, according to Eq 3.1, by controlling this maximum error to be at most t, we have that the expected false positive rate is no more than δ. This amounts to setting the CountMin dimensions to $w = O(2/\varepsilon) = 2F/t$ and $d = 1/\delta$.

3.2.3 CountMin as a probabilistic NDS representation for de Bruijn graphs

Finally, in order to make the CountMin more easily navigable, we expand the sketch such that each cell $C[i,j]$ in the matrix stores not only the counter $C[i,j].c$, but also a set of outedges $C[i,j].E$. The structure then provides an operation to *update* the counters $C[i,h_i(X)].c$ associated with a given k-mer X by one, and an operation to *add an outedge* to the sets $C[i,h_i(X)].E$. The update operation is the same as in a regular CountMin, and the procedure for adding an edge can be seen in Algorithm 2.

Algorithm 2: $C.add\_outedge(X,a)$

```
Input  : C, the DBCM; $X$, the $k$-mer; $a \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$, the outedge label
for $i \leftarrow 0,\dots,d-1$ do
    $C[i,h_i(X)].E \leftarrow C[i,h_i(X)].E \cup \{a\}$
end
```

Besides the query operation, used for obtaining the count for a given k-mer X, done in the same way as in a regular CountMin, the DBCM must also implement an operation for retrieving the outedges for X. This new operation is presented in Algorithm 3.

Algorithm 3: $C.outedges(X)$

```
Input  : $C$, the DBCM; $X$, the $k$-mer
Output: The outedge set $E$
$E \leftarrow \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$
for $i = 1,\dots,d$ do
    $E \leftarrow E \cap C[i,h_i(X)].E$
end
return $E$
```

In practice, due to a node having at most four outedges corresponding to the bases $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$, the set storing them can be represented with four bits indicating whether each of them is present. An edge is added by setting the corresponding bit, and the intersection is obtained by performing the bitwise AND operation. Moreover the set of outedges and the counter are packed together in a single 16-bit integer. This layout can be visualized in Figure 3.2. When using a k such that $4^k > |S|$, then each k-mer in S is expected to appear only once in S, and a number of times equals to the coverage c in the reads. Considering that c is not a very high value (commonly below 200), each counter can be represented by a 12-bit integer, accounting for the expected count for all real k-mers, as well as possible collisions.

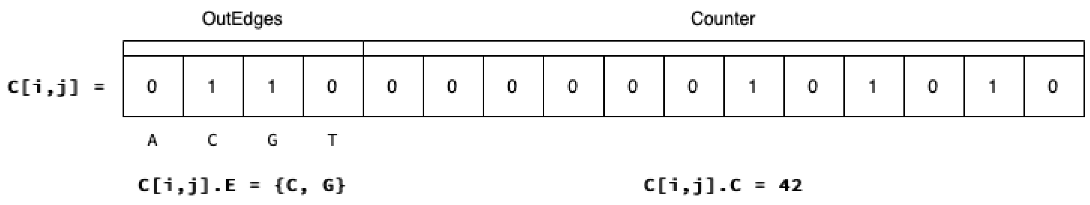


Figure 3.2: A DBCM cell

3.2.4 DBCM Construction

The DBCM is constructed as the reads are processed by incrementing the counters for each k-mer as they are read. Once a k-mer's counter reaches the threshold for presence t, an outedge is added from the k-mer that preceded it to itself. That is if k-mers X and Y happen consecutively in a read, and $C.query(Y).c \geq t$, then the operation $C.add\_outedge(X, Y[k-1])$ is performed. By only adding the edge $X \xrightarrow{a} Y$ when Y is considered to be high-frequency we avoid adding edges to nodes that we do not yet know to exist. Such edges would point, in the best case, to a node that would not be found to be on the graph, resulting in unnecessary operations before the path is considered finished, and in the worst case would be interpreted, in a collision, as a diferent edge, resulting in a new branch on the graph. Conversely, the source node X does not need to be considered present on the graph for the edge to be added. This avoids the scenario where both X and Y are real k-mers, but Y shows up for the last time before X is considered to be present, in which case the edge would never be added. Moreover, adding an outedge to a k-mer that is not considered to be present in the graph is irrelevant, as the k-mer will be ignored during traversal. Because the resulting DBCM is dependent on the threshold t used during construction, we store this value and denote it by $C.t$.

Besides updating the de Bruijn graph represented by the DBCM by counting the k-mers and tracking the outedges, the construction step also generates a set of k-mers known to be represented in the graph that can be used for traversal, as further discussed in Section 3.2.6. The procedure for constructing a DBCM can be seen in Algorithm 4. Like Conway & Bromage [7], we process each read in the forward and in the reverse complement direction, without merging k-mers corresponding to reverse complements of each other.

3.2.5 DBCM Operations

We can implement the operations listed in Section 2.7 using the DBCM as follows.

Algorithm 4: $C.construct(\mathscr{X}, t, n)$

```
Input  : $C$, the DBCM; $\mathscr{X}$, the set of reads; $t$, the frequency threshold for a $k$-mer to
         be considered true; $n$, the number of $k$-mers from each read to be included in
         $\mathscr{S}$
Output: $\mathscr{S}$ the set of starting $k$-mers
$C.t \leftarrow t$
$\mathscr{S} \leftarrow \emptyset$
$\overline{\mathscr{X}} \leftarrow \{\overline{R}; R \in \mathscr{X}\}$          ▷ The set of the reverse complements of the reads
for $R \in \mathscr{X} \cup \overline{\mathscr{X}}$ do
    $X \leftarrow \varnothing$
    $i \leftarrow 0$
    for $Y \in k\text{-}mers(R)$ do
        $C.update(Y)$
        if $C.is\_member(Y)$ then                                  ▷ see Section 3.2.5
            if $i \leq n$ then
                $\mathscr{S} \leftarrow \mathscr{S} \cup \{Y\}$
            end
            if $i > 0$ then
                $C.add\_outedge(X, Y[i-1])$
            end
        end
        $i \leftarrow i + 1$
        $X \leftarrow Y$
    end
end
return $\mathscr{S}$
```

Membership query Given the desired presence threshold t, we can implement the membership query operation in the DBCM as $C.is\_member(X) \equiv C.query(X).c \geq C.t$.

Forward neighbor query $C.forward\_neighbor(X, a) \equiv a \in C.query(X).E$.

Neighborhood query We can transform the set of outedges $C[i, j].E$ into a set of k-mers by extending the k-mer with the outedges from the set. As such, a procedure for generating the set of neighbors for a given k-mer X is found in Algorithm 5.

Algorithm 5: $C.neighbors(X)$

```
    Input  : $C$, the DBCM; $X$, a $k$-mer
    Output: $N$, a set of $k$-mers that are neighbors of $X$
    $N \leftarrow \emptyset$
    for $a \in C.query(X).E$ do
        $N \leftarrow N \cup \{X[1 : k-1] \cdot a\}$
    end
    return $N$
```

3.2.6 DBCM Traversal

This structure can be traversed in a breadth-first order from an initial set of k-mers $\mathscr{S}$ by querying for their out-edges and then querying each of their neighbors, repeating this for each neighbor found to be in the graph. This procedure can be observed in Algorithm 6.

Selecting the starting set $\mathscr{S}$ of k-mers for traversal Ideally, only the first k-mer from the original sequence S is needed to perform the traversal of the graph, but unfortunately it is not possible to determine where in the original sequence a read was taken from. One option is to use all the k-mers from the reads that are found to be in the graph, but we argue that this set is unnecessarily large. We can instead reduce the size of $\mathscr{S}$ by only considering the k-mers that are at the start of their reads since the following k-mers would be reachable from it anyway.

3.3 The de Bruijn Hashtable

We also propose a new hashtable-based representation for the de Bruijn graph, called ***de Bruijn Hashtable*** DBHT, that is made more space-efficient by not storing the k-mer directly. Instead, the slot $T[i]$ containing the k-mer X stores a ***fingerprint*** $T[i].f = f_X$, computed from X as described in Section 3.3.2, along with the set of outgoing edges from X, $T[i].E$, as described in Section 3.2.

When inserting a k-mer X into a DBHT of capacity m, a hash value h_X and the fingerprint f_X are calculated in parallel. The hash value, computed as described in Section 3.3.1, determines the initial position p_0 in which the k-mer should be stored. If this slot is empty, then the fingerprint is written there. Otherwise, collisions are resolved by linear probing, that is the

Algorithm 6: *$C.traverse(\mathscr{S}, t)$*

```
Input  : $C$, the DBCM; $\mathscr{S}$, the set of starting $k$-mers; $t$, the threshold of presence
Output: $V$, the set of $k$-mers queried from the DBCM during traversal
$F \leftarrow$ empty Queue
for $s \in \mathscr{S}$ do
    enqueue($F$, $s$)
end
$Q \leftarrow \{\}$
while $F$ is not empty do
    $u \leftarrow$ dequeue($F$)
    $Q \leftarrow Q \cup \{u\}$
    if $C.is\_member(u,t)$ then
        for $v \in C.neighbors(u)$ do
            if $v \notin Q$ then
                enqueue($F$, $v$)
            end
        end
    end
end
return $Q$
```

subsequent positions $(p_0 + j) \mod m$, for $j = 0 \dots m-1$, are checked until a free slot is found. If, however, a slot containing f_X is found before, then X is considered to be already represented in H and the insertion aborts. This process is detailed in Algorithm 7. Notice that this requires always having some unused slots, that is, having a *load factor* $0 < \alpha = n/m < 1$, where n is the number of used positions. Higher values of α will result in higher space-efficiency, but also‘ more collisions. Lower values have the opposite effect.

To query the hashtable for a k-mer/node X, the same sequence of positions as in the insertion are probed until the desired fingerprint f_X is found, or a free position is reached, in which case we can conclude that the k-mer is absent from the structure. This operation is presented in Algorithm 8.

Adding an edge $X \xrightarrow{a} Y$ to the DBHT is similar to adding a k-mer node, and breaks down to, first, locating the slot $T[i]$ of the source node X, and then adding the corresponding edge label a to its set of outedges. As before, this is done by setting the appropriate bit of the 4-bit pattern $T[i].E$. This procedure is detailed in Algorithm 9. Note that, because two k-mers can be mapped to the same cell, the outedges stored for both of them will be the same and will contain the true outedges of each one individually. This further makes it important that edges only be added to nodes known to be in the graph, which the DBHT does not verify.

3.3.1 Modular Fibonacci Hashing

The DBHT uses the Fibonacci hashing algorithm [27] to hash the k-mers, which leverages the property of the golden ratio ϕ that, given a range r, the set $\Gamma = \{i \cdot \frac{r}{\phi} \mod r;\ i \in \mathbb{N}\}$ is evenly

Algorithm 7: $T.insert(X)$

```
Input   : $T$: a DBHT with capacity $m$; $X$: the $k$-mer to be inserted
$h_X \leftarrow fibhash(X, m)$                                ▷ see Section 3.3.1
$f_X \leftarrow fingerprint(X)$                               ▷ see Section 3.3.2
$i \leftarrow h_X$
while $T[i]$ is not empty do
    if $T[i].f = f_X$ then
        return                                                ▷ $X$ already in $T$
    else
        $i \leftarrow (i+1) \mod m$
    end
end
$T[i].f \leftarrow f_X$
$T[i].E \leftarrow \varnothing$
```

Algorithm 8: $T.query(X)$

```
Input   : $T$: a DBHT with capacity $m$; $X$: the $k$-mer to be queried
$h_X \leftarrow fibhash(X, m)$
$f_X \leftarrow fingerprint(X)$
$i \leftarrow h_X$
while $T[i]$ is not empty do
    if $T[i].f = f_X$ then
        return $T[i].E$
    else
        $i \leftarrow (i+1) \mod m$
    end
end
return $\varnothing$
```

Algorithm 9: $T.add\_outedge(X, a)$

```
Input   : $T$: a DBHT with capacity $m$; $X \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}^k$: the $k$-mer to be queried;
          $a \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$: the edge label to be added
$h_X \leftarrow fibhash(X, m)$
$f_X \leftarrow fingerprint(X)$
$i \leftarrow h_X$
while $T[i]$ is not empty do
    if $T[i].f = f_X$ then
        $T[i].E \leftarrow T[i].E \cup \{a\}$
    else
        $i \leftarrow (i+1) \mod m$
    end
end
```

distributed over r, and has no period. Note that the modulo operator here means the remainder of a real division, so that the elements of Γ are real numbers. The integer hashing algorithm that maps a number to the range $[0,m]$ works by multiplying the key X by the closest integer to the value m/ϕ taking the result modulo m. The *fibhash* function can be seen in Algorithm 10.

```
Algorithm 10: $fibhash(X,m)$
    Input  : $X$, the key to be hashed; $m$, the range into which it should be hashed
    return $(round(m/\phi) \cdot X) \mod m$
```

3.3.2 k-mer Fingerprinting

We obtain a 3-bit fingerprint of a k-mer by performing a modified version of the Fibonacci hashing algorithm. We use the hashing function described above for the range addressable by a 64-bit integer ($r = 2^{64}$), with the modulo operation being performed implicitly by using integer multiplication with a 64-bit int type. Then we take the three most significant bits of the result as the fingerprint. We choose the most significant bits because, as observed by Skarupke, they present the best avalanche behavior, i.e. flipping any one bit individually from the input causes each bit in the output to flip with close to 50% probability [27]. In practice this procedure, shown in Algorithm 11, is very fast due to it foregoing the modulo operation and performing only one multiplication by a constant and a bitwise shift operation to extract the most significant bits. Note that, although 11400714819323198486 is actually closer to $2^{64}/\phi$, we prefer an odd number so as not to waste the last bit.

```
Algorithm 11: $fingerprint(X)$
    Input  : $X$, the key to fingerprint
    Let $\psi$ be the 64-bit constant $round(2^{64}/\phi) = 11400714819323198485$
    return $(X \cdot \psi) >> 61$
```

3.3.3 DBHT Construction

We construct the DBHT by traversing the DBCM representation of the graph starting from the set of initial nodes $\mathscr{S}$, which we insert into the DBHT, then inserting all of their true neighbors via the corresponding outedges, and then repeating the process from those neighbors. This procedure can be seen in Algorithm 12.

3.3.4 DBHT Operations

Membership query The membership query is just the evaluation of if the query operation found the k-mer: $T.is\_member(X) \equiv T.query(X) \neq \varnothing$.

Forward neighbor query As in the DBCM, the forward neighbor query just checks if the given base a is in the set of outedges of X: $T.forward\_neighbor(X,a) \equiv a \in T.query(X)$.

Algorithm 12: $T.construct(C)$

```
Input : $T$, the DBHT; $C$, a DBCM; $t$, the frequency threshold for a $k$-mer to be
        considered to be represented in $C$
$F \leftarrow$ empty Queue
for $s \in \mathscr{S}$ do
    enqueue($F$, $s$)
end
$Q \leftarrow \{\}$
while $F$ is not empty do
    $u \leftarrow$ dequeue($F$)
    $Q \leftarrow Q \cup \{u\}$
    if $C.is\_member(u,t)$ then
        $T.insert(u)$
        for $v \in C.neighbors(u)$ do
            if $C.is\_member(v,t)$ then
                $T.add\_outedge(u, v[k-1])$
                if $v \notin Q$ then
                    enqueue($F$, $v$)
                end
            end
        end
    end
end
```

Neighborhood query Again, similarly to the DBCM, the neighbors of X can be obtained by extending it with its outedges, as presented in Algorithm 13.

Algorithm 13: $T.neighbors(X)$

```
Input  : $T$, the DBHT; $X$, a $k$-mer
Output: $N$, a set of $k$-mers that are neighbors of $X$
$N \leftarrow \emptyset$
for $a \in T.query(X)$ do
    $N \leftarrow N \cup \{X[1:k-1] \cdot a\}$
end
return $N$
```

3.3.5 DBHT Traversal

We traverse DBHT in a breadth-first manner from a starting set of k-mers $\mathscr{S}$ by adding them to a queue. Then we dequeue one node and query for its neighbors, adding them to the queue if they weren't visited yet. We repeat this process until the queue is empty. This procedure can be seen in Algorithm 14. As in the traversal of the DBCM, the result is the set of k-mers that were queried during traversal.

Algorithm 14: $T.traverse(\mathscr{S}, t)$

```
Input  : $C$, the DBCM; $\mathscr{S}$, the set of starting $k$-mers
Output: $V$, the set of $k$-mers queried from the DBCM during traversal
$F \leftarrow$ empty Queue
for $s \in \mathscr{S}$ do
    enqueue($F$, $s$)
end
$Q \leftarrow \{\}$
while $F$ is not empty do
    $u \leftarrow$ dequeue($F$)
    $Q \leftarrow Q \cup \{u\}$
    if $C.is\_member(u)$ then
        for $v \in C.neighbors(u)$ do
            if $v \notin Q$ then
                enqueue($F$, $v$)
            end
        end
    end
end
return $Q$
```

CHAPTER 4

Results

We evaluate the DBCM based on how well it can identify true k-mers in the unprocessed reads (Section 4.1), as well as on how it can further filter out spurious k-mers through traversal of the graph (Section 4.2). We then evaluate how the false positive rate is affected when traversing the DBHT built from the traversal of the DBCM (Section 4.3). All the experiments described herein were carried out with synthetic reads generated using the *ART Illumina* simulation toolkit [12] of the *E. Coli* chromosomal DNA [1] with coverage $c = 80\times$ and reads of 250 bases, resulting in approximately 1.5M distinct reads. The experiments were conducted on a Dell XPS 15 9575, with a 4.1GHz Intel Core i7-80705G CPU and 16GB of RAM, plus 8GB of SSD swap.

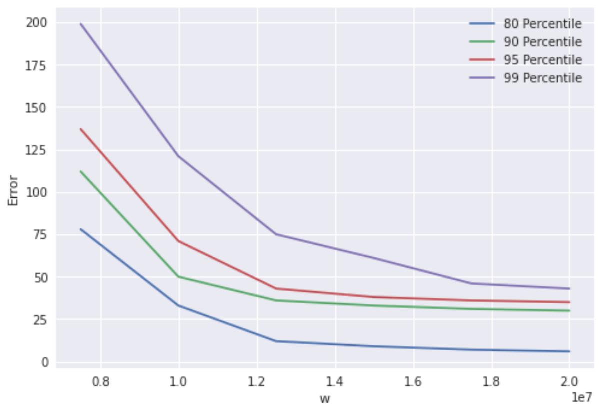
We show that exact counts are not needed to filter out the majority of spurious k-mers from the reads, such that a DBCM using 120MB of memory can still filter over 80% of the erroneous k-mers, leaving only twice as many as if the exact counts had been used, in which case $\approx$ 500MB of memory would have been required. Furthermore, traversing the DBCM and considering only the visited nodes as true k-mers causes the number of false positives to decrease by over 98%. Finally, a DBHT constructed from all the nodes and edges found during traversal of the DBCM can still be succesfully traversed, with an increase in the number of false k-mers by a factor of 2, but still keeping the percentage of false nodes on the graph below 5% with a load factor $\alpha = 0.5$, and even with $\alpha = 0.9$ only around 15% of the nodes visited during traversal represented k-mers not found in the original sequence. It also incurs in no loss of sensitivity, with over 99% of true k-mers being represented in the resulting de Bruijn graph. The DBHT needs between 9 bits per k-mer, when $\alpha = 0.9$, and 16 bits per k-mer, when $\alpha = 0.5$. This is a comparable result to other representations such as the one presented by Conway & Bromage [7] which, although exact, requires 24 bits per k-mer [6], as well as the representation introduced by Chikhi & Rizk, which requires just slightly less than 13.5 bits per k-mer [6]. It is also close to the results for the representation proposed by Salikhov *et al.*, which, in theory, would require 7.93 bits per k-mer and, in practice, needs around 8.45 bits per k-mer [25]. The *DBGFM* achieves a considerably more succint representation of the graph, requiring less than 5 bit per kmer, but sacrifices construction and query times [5].

4.1 The DBCM as a de Bruijn graph representation

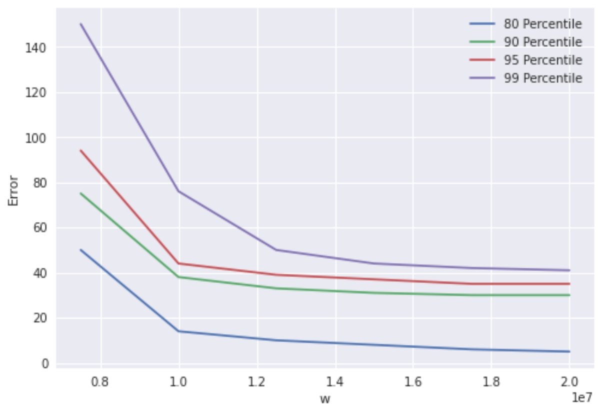
To assess how well the DBCM can represent a de Bruijn graph when constructed from the raw sequencing reads, we first analyze how accurately it performs k-mer counting. We constructed a set of DBCM instances with varying dimensions $w \in \{7.5\text{M}, 10\text{M}, \dots 17.5\text{M}, 20\text{M}\}$[1] and $d \in$

[1] $1\text{M} = 10^6$

$\{6,8\}$ from the set of synthetic reads of the *E. Coli* genome. For each instance, we calculated the 80, 90, 95, and 99 percentiles of the individual count errors $\Delta(X) = C.query(X) - c(X)$. In Figure 4.1 we plot these metrics as a function of w for the two values of d.



(a) $d = 6$

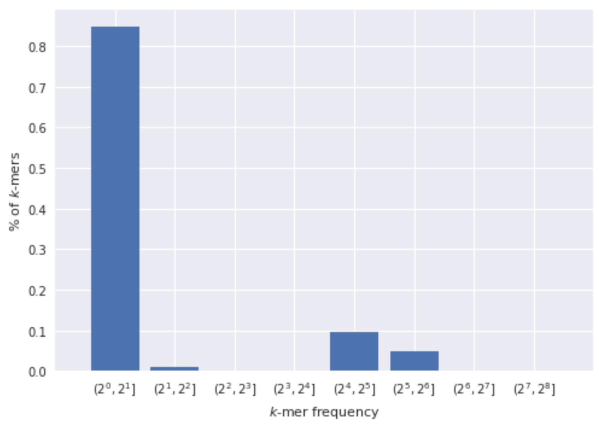


(b) $d = 8$

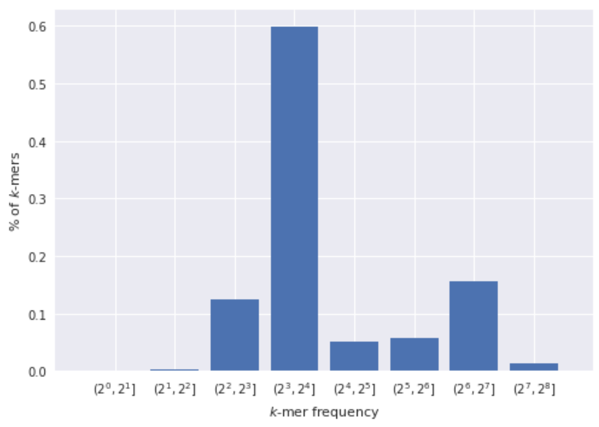
Figure 4.1: DBCM miscount percentiles as a function of w for $d = 6$ and $d = 8$

First, we observe that there is only a slight change for $w \geq 10$M on both scenarios, but one could still argue that the errors can still be too high. However, we are not so much interested in the absolute counts per se, but rather as way of discerning real k-mers occurring in original sequence from spurious k-mers appearing only in the reads due to sequencing errors. In order to save memory, we would like to use the smallest values allowing for this separation. To verify that the DBCM can effectively differentiate between these two categories, we select the conservative values $w = 10$M and $d = 6$ and compare the distribution of the count estimates with the actual values for the k-mers in the reads. This comparison can be observed in Figure 4.2. Note that, in the exact count distribution, close to 9M k-mers have frequency between 2^4 and 2^6. This matches the expected number of real k-mers from the sequence in forward and reverse complement forms. Although the estimated counts distribution is shifted right due to overcount, we see two peaks, one between 2^3 and 2^4, and the other between 2^6 and 2^7, indicating the high- and low-frequency categories may still be discernible although a threshold is not as clear as with the exact counts. Combining these observations, we see that for $w \geq 10$M, 90% of k-mers have an error of no more than 50 regardless of d, such that we expect the majority of erroneous k-mers to have an estimated count not much greater than that, while real k-mers are expected to have a count not much lower than 80 (the coverage). This suggests that high- and low-frequency k-mers are likely to still be discernible based on a threshold in the 30–60 range.

Next we evaluate possible thresholds t in the range $[1, 2c]$ by analyzing the ***true positive rate***, or ***sensitivity***, of the DBCM, i.e. percentage of true k-mers correctly identified, as well as its ***false positive rate***, i.e. percentage of false k-mers erroneously identified as true, using each one of those values. The results are presented in Figure 4.3b, where we can see that, without excluding any true k-mers from the de Bruijn graph, we can filter out over 80% of the spurious k-mers from the reads by using $20 \leq t \leq 40$. However, comparing these results with the sensitivity and false positive rate if using the exact counts (Figure 4.3a), we can see



(a) Exact

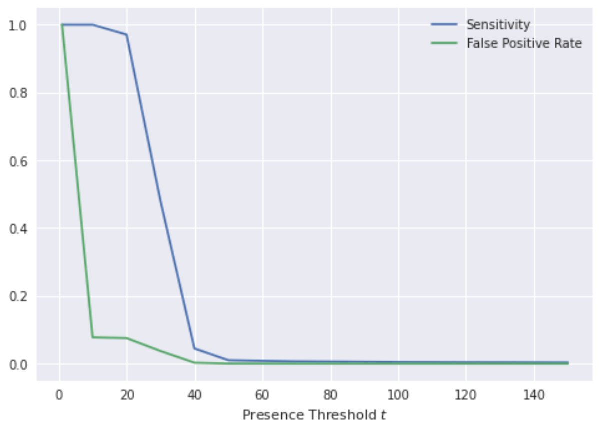


(b) Estimated ($w = 10\text{M}, d = 6$)

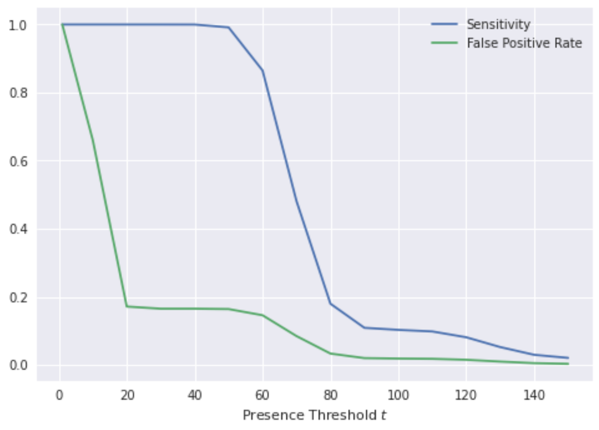
Figure 4.2: k-mer count distribution

that the number of false positives nearly doubles when keeping sensitivity maximized. In this dataset, this equates to nearly 9.7M spurious k-mers being added to the de Bruijn graph, over 5M more than would be added using exact counts. In the following section we show that, despite this increase in number of spurious k-mers represented in the DBCM, only a fraction of them are reachable in traversal, such that this representation still succeeds in filtering out erroneous k-mers.

A DBCM with $w = 10\text{M}$ and $d = 6$ requires $16 \times 10\text{M} \times 6 = 960\text{M}$ bits, or 120MB. The time needed for construction of the DBCM is linear on the total number of bases in the reads, and took about 6 minutes in our experimental environment.



(a) Exact counts



(b) Estimated by DBCM ($w = 10\text{M}, d = 6$)

Figure 4.3: Sensitivity and False Positive Rate of a DBCM with $w = 10\text{M}$ and $d = 6$ as a function of the presence threshold t

4.2 Filtering through traversal

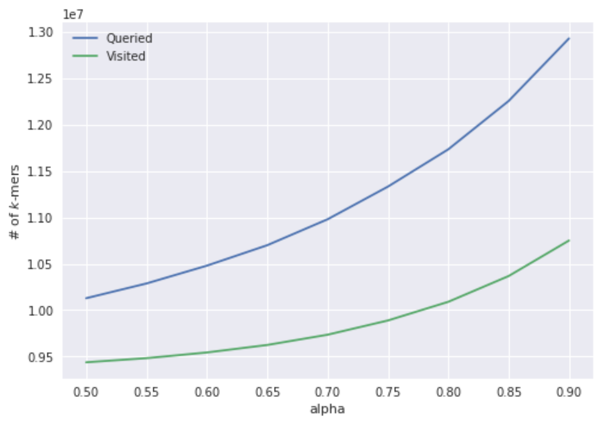
As discussed in Section 3.1, we expect many of the spurious k-mers inserted into the de Bruijn graph due to overcoming the frequency threshold to be isolated from the main components of the graph, containing the real k-mers. By traversing the graph from a small sample of these high-frequency k-mers, we expect never to visit most of these disconnected components, such that using the set of k-mers visited during a full traversal, rather than the set of high-frequency k-mers, to construct a representation of the de Bruijn graph should lead to a more succinct representation.

To evaluate this hypothesis, we constructed a DBCM with $w = 10$M, $d = 6$, and $t = 40$, from the unprocessed reads. We then traversed the graph from a subset of high-frequency k-mers at the beginning of the reads. Approximately 9.62M k-mers were queried in the process, of which ≈ 9.58M were effectively visited. Of those visited, only 173K were not found in the original DNA sequence in either forward or reverse complement form, or approximately 1.8% of the number of spurious k-mers that would have been added to a de Bruijn graph from the reads alone, based on the count estimated by the same DBCM, and 3.8% of the spurious k-mers from the reads that would have been added based on their exact counts considering the optimal threshold. In a hashtable-based representation, such as the DBHT, this would reduce the number of cells needed to represent the de Bruijn graph for this organism by *one third* when compared with using the exact counts as the requirement for insertion.

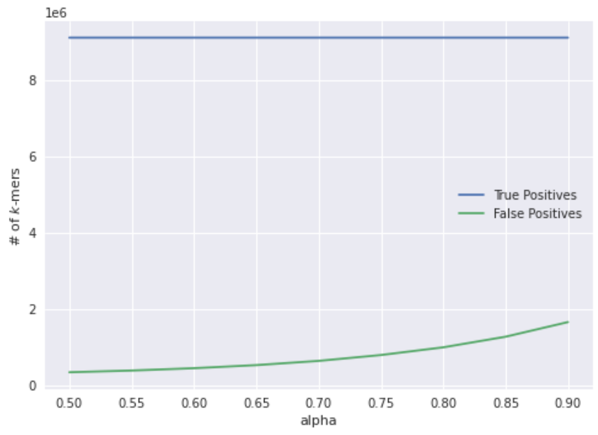
4.3 The DBHT as a NDS

Finally, we want to evaluate how different are the graphs represented by the DBCM and the DBHT generated from it. Note that, contrary to the DBCM, any node inserted into the DBHT found when queried, and likewise the outedges, such that there should be no loss in sensitivity when going from one representation to the other. However the DBHT may introduce errors due to collisions, because the k-mer is not directly saved in the hashtable, but only its fingerprint. If two k-mers with the same fingerprints are mapped to the same cell, we may take one for the other. This might lead to distinct k-mers sharing the same set of outedges which, in turn, can result in false k-mers being queried and incorrectly determined to be represented in the graph. The chance of collision, however, is controlled by the choice of load factor α. Therefore, we evaluate the traversal performance of DBHTs with varying load factors $\alpha \in \{0.5, 0.55, \ldots, 0.85, 0.9\}$, all constructed from the same DBCM with $w = 10$M, $d = 6$, and $t = 40$.

In Figure 4.4a, we can see that both the number of k-mers queried during the traversal, and number of k-mers actually visited, grow with α, as the table size shrinks and collisions become more frequent. This naturally results more false positives, as evidenced in Figure 4.4b. Note that, compared to the DBCM, the false positive rate is about twice as high, even in the best case with $\alpha = 0.5$. However, even in the worst case, with $\alpha = 0.9$, the number of spurious k-mers is less than 40% of what would be *explicitly inserted* from the reads based on *exact* counts alone. These spurious k-mers also represent only a small portion, $\approx 15\%$, of the the resulting graph. Our experiments also do confirm that there is no sensitivity loss in going from the DBCM to the DBHT in the sense that nearly all (99.5%) real k-mers are still queried and visited in the traversal in all tested scenarios.



(a) Queried (blue) and visited (green) counts



(b) True (blue) and false (green) positive counts

Figure 4.4: Traversal results for DBHT according to load factor α

With a load factor of $\alpha = 0.5$, the DBHT requires 16 bits per k-mer represented in it. In the experiments executed, this translates to approximately 19MB, 15% of the memory taken by the DBCM used to construct it. This can be further improved by using a bigger load factor. With $\alpha = 0.7$, only approximately 14MB are required, while only 6% of the k-mers visited in the traversal of the DBHT were spurious, and with $\alpha = 0.9$ the de Bruijn graph would be represented in approximately 11MB, with 15% of the nodes reachable through traversal being erroneous.

CHAPTER 5

Conclusion

We conclude this work by summarizing our main findings, and presenting a brief list of topics for further development.

5.1 Main results and contributions

The experimental analysis described in Chapter 4 allows us to conclude the following.

CountMin-based k-mer counting can be used to filter spurious k-mers from sequencing reads We start by noticing that our DBCM variation of the CountMin sketch can be used to filter spurious *k*-mers resulting from sequencing errors. We expand on the results presented by Zhang *et al.*, with *khmer*, [28] to show that, despite overcounting, we can still filter out most (over 80%) of spurious *k*-mers based on the count estimate from a CountMin sketch that has fewer cells than the number of distinct *k*-mers in the reads, and uses only 16 bits per cell. These results are similar to what *khmer* achieves through iteratively truncating the reads at the first low-frequency *k*-mer [28], i.e. *read trimming*. However, filtering directly from the counts only requires that the reads be processed once.

Filtering through traversal is very effective We have also shown that traversing the de Bruijn graph generated from the high-frequency *k*-mers further improves filtering of spurious *k*-mers without affecting the number of real *k*-mers represented. Therefore, although Ghosh & Kalyanaraman [10] present a framework for using the CountMin sketch for online filtering of spurious *k*-mers that allows the sketch to be considerably smaller, reducing the size of the CountMin sketch much further than presented here hinders its ability to be successfully navigated, losing a very efficient second filter that removed nearly 98% of the remaining spurious *k*-mers after count-based filtering. This would decrease memory requirement during processing of the reads, but results in a less succinct final representation of the de Bruijn graph.

Storing outedges improves traversal when compared to querying all possible neighbors We also see that, for both data structures used, the number of outedges stored for each node in the de Bruijn graph is low, with the vast majority having only one recorded outedge. Without using the outedges, and querying all four possible neighbors of a given node, we would need to query between two to four times as many *k*-mers, just from the nodes represented in the de Bruijn graph. Storing the outedges, even with some chance of false positives, can therefore decrease the impact of the false positive rate associated with the membership query of either structure by preventing false k-mers from even being queried. Moreover, by reducing the number of queries, storing the set of outedges also improves time performance of traversals or related operations.

The DBHT can succesfully represent a de Bruijn graph in as few as 9 *bits per k-mer* Finally, we have shown that the DBHT can represent a de Bruijn graph in as few as 9 bits per k-mer with, at most, 16% of the graph being composed of false k-mers introduced by the probabilistic nature of the representation. Based on the exploration by Pell *et al.* [21], we expect this number of false positives not to affect usability of the graph for assembly. We can reduce it, however, by increasing the size of the DBHT. With 16 bits per k-mer, we achieved a number of false positive nodes that is below 4% of the total number of k-mers represented in the graph. Moreover, our final DBHT representation is very efficient, with node and edge queries performed in constant time through a few simple operations. These results are comparable to current de Bruijn graph representations, that span from 4 bits per k-mer to 24 bits per k-mer [6, 11].

5.2 Future Work

Beyond successfully achieving the main goals for this work, we have also identified a few points that would benefit from further work.

Comparing our methods to similar work It would be important to compare the results obtained in this project to related work more directly. In particular, we hypothesize that the CountMin-based filter used in *FastEtch* [10], because of its very small size, is less effective in filtering spurious k-mers than the pipeline we introduce, based on count and traversal. We also hypothesize that despite providing a slightly less succinct de Bruijn graph representation in the form of a DBHT, this structure has better performance than Bloom Filter-based representations, e.g. the one introduced by Pell *et al.* [21], due to reducing the number of potential neighbor queries, and the number of hashing functions to only one in practice.

Generating contigs and N50 score for the DBHT A natural next step is to use the DBHT to generate maximal contigs and, from them, compute the ***N50 score***, defined as the length l_0 of the shortest contig such that all contigs with length $l \geq l_0$ cover at least 50% of the assembled sequences. The N50 score is used to determine the quality of an assembly in terms of contiguity. Based on the distribution of outedges observed during traversal of the DBHT, we expect this representation to perform fairly well.

Optimizing parts of the implementation We have implemented the methods described herein in C-language for maximum performance. However, a few things can still be streamlined and optimized. For instance, we have used a queue and a set to store respectively the nodes to be visited and the ones already visited during the DBCM traversal. In practice, this approach is not interesting as it ultimately requires representing the de Bruijn graph in memory as a set, in parallel to the succinct representation being constructed, which is prohibitive for larger genomes. It would be interesting to see how using the DBHT itself, as it is being constructed, as the set of visited nodes would affect its final result. Because of false positives, this could result in true nodes not being visited, and therefore not represented in the graph.

All the findings outlined above suggest that our approach is a viable alternative and offers interesting tradeoffs for the representation of de Bruijn graphs for genomic data, with challenging and promising opportunities for further development.

Bibliography

[1] http://ftp.ensemblgenomes.org/pub/bacteria/release-52/fasta/bacteria_0_collection/escherichia_coli_str_k_12_substr_mg1655_gca_000005845/dna/. Accessed: 2022-05-12. 4

[2] B. Alipanahi, A. Kuhnle, S. J. Puglisi, L. Salmela, and C. Boucher. Succinct dynamic de Bruijn graphs. *Bioinformatics*, 37(14):1946–1952, 07 2021. 2.6

[3] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012. 1, 2.6

[4] R. Chikhi, J. Holub, and P. Medvedev. Data structures to represent a set of k-long dna sequences. *ACM Comput. Surv.*, 54(1), mar 2021. 2.7

[5] R. Chikhi, A. Limasset, S. Jackman, J. Simpson, and P. Medvedev. On the representation of de bruijn graphs. 1 2014. 1, 2.6, 2.7, 4

[6] R. Chikhi and G. Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter, 2013. 1, 2.6, 4, 5.1

[7] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27:479–486, 2 2011. 1, 2.3, 2.4, 2.6, 3.2.4, 4

[8] 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. 1, 3.2.1

[9] J. Gallant, D. Maier, and J. Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980. 1

[10] P. Ghosh and A. Kalyanaraman. Fastetch: A fast sketch-based assembler for genomes. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16:1091–1106, 7 2019. 1, 2.4, 2.5, 5.1, 5.2

[11] A. M. Giani, G. R. Gallo, L. Gianfranceschi, and G. Formenti. Long walk to genomics: History and current approaches to genome sequencing and assembly. *Computational and Structural Biotechnology Journal*, 18:9–19, 2020. 5.1

[12] W. Huang, L. Li, J. R. Myers, and G. T. Marth. ART: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 12 2011. 4

[13] M. Imelfort and D. Edwards. De novo sequencing of plant genomes using second-generation technologies, 2009. 1

[14] E. Kapun and F. Tsarev. De bruijn superwalk with multiplicities problem is np-hard. *BMC Bioinformatics*, 14:S7, 2013. 1

[15] K. Lohmann and C. Klein. Next generation sequencing and the future of genetic diagnosis. *Neurotherapeutics*, 11:699–707, 2014. 1

[16] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 01 2011. 2.5

[17] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. In *Proceedings of the 7th International Conference on Algorithms in Bioinformatics*, WABI'07, page 289–301, Berlin, Heidelberg, 2007. Springer-Verlag. 1

[18] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12, 8 2011. 1, 2.5

[19] M. L. Metzker. Sequencing technologies the next generation, 1 2010. 1, 2.4

[20] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data, 6 2010. 1

[21] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences*, 109:13272–13277, 2012. 1, 2.6, 5.1, 5.2

[22] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001. 1, 2.2

[23] G. Rizk, D. Lavenier, and R. Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, pages 652–653, 2013. 2.5

[24] R. S. Roy, D. Bhattacharya, and A. Schliep. Turtle: Identifying frequent k -mers with cache-efficient algorithms . *Bioinformatics*, 30(14):1950–1957, 03 2014. 2.5

[25] K. Salikhov, G. Sacomoto, and G. Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs, 2014. 2.6, 4

[26] J. Shendure and H. Ji. Next-generation dna sequencing, 10 2008. 1

[27] M. Skarupke. Fibonacci hashing: The optimization that the world forgot (or: a better alternative to integer modulo). https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-amp/, 2018. Accessed: 2022-05-08. 3.3.1, 3.3.2

[28] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure. *PLoS ONE*, 9, 7 2014. 1, 2.4, 2.5, 3.2.2, 5.1