Compact universal *k*-mer hitting sets

Yaron Orenstein[1], David Pellow[2], Guillaume Marçais[3], Ron Shamir[2], and Carl Kingsford[3]

[1] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA
[2] Blavatnik School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel
[3] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
yaronore@mit.edu, dpellow@tau.ac.il, gmarcais@cs.cmu.edu, rshamir@tau.ac.il, carlk@cs.cmu.edu

Abstract. We address the problem of finding a minimum-size set of k-mers that hits L-long sequences. The problem arises in the design of compact hash functions and other data structures for efficient handling of large sequencing datasets. We prove that the problem of hitting a given set of L-long sequences is NP-hard and give a heuristic solution that finds a compact universal k-mer set that hits *any* set of L-long sequences. The algorithm, called DOCKS (design of compact k-mer sets), works in two phases: (i) finding a minimum-size k-mer set that hits every infinite sequence; (ii) greedily adding k-mers such that together they hit all remaining L-long sequences. We show that DOCKS works well in practice and produces a set of k-mers that is much smaller than a random choice of k-mers. We present results for various values of k and sequence lengths L and by applying them to two bacterial genomes show that universal hitting k-mers improve on minimizers. The software and exemplary sets are freely available at `acgt.cs.tau.ac.il/docks/`.

1 Introduction

Inspired by Grabowski and Raniszewski's sampled suffix array using minimizers [1], we consider the following problem involving covering strings by selecting short k-mer substrings:

Problem 1. Given integers k and L, find a smallest set U_{kL} of k-mers such that any string of length L or longer must contain at least one k-mer from U_{kL}.

The set U_{kL} is called a *universal* set of hitting k-mers, and we call each k-mer in the set *universal*. Such a set has a number of applications in speeding up genomic analyses since it can often be used in places where minimizers have been used in the past [2]. For example:

1. **Hashing for read overlapping.** A naïve read overlapper must test $O(n^2)$ pairs of reads to see whether they overlap. If we require an overlap of length L, any pair of reads with such an overlap must contain a k-mer from set U_{kL} in this overlapped region. By bucketing reads into bins according to the universal k-mers they contain, we need only test pairs of reads in the same bucket. The number of buckets is limited by $|U_{kL}|$.

2. **Sparse suffix arrays.** A sparse suffix array of a string S saves memory by storing an index for only every sth position in S [3, 1]. To query a sparse suffix array for string q, we perform at most s queries starting from indices $0, \ldots, s-1$ in q; one of these queries will intersect a position stored in the suffix array. Using U_{kL}, we can instead store only positions in S that start with a k-mer in U_{kL}. Any query with $|q| \geq L$ must contain one of these selected k-mers and will be matched when searching the suffix array.
3. **Bloom filters to speed up sequence search.** Bloom filters have been used to speed up sequence search by storing k-mers present in a read set for quick testing [4]. In current implementations, all k-mers present in a read set are stored in these filters. If, instead, only the set of k-mers in a U_{kL} is stored, any window of length $\geq L$ is still guaranteed to contain one of these representative queries, potentially reducing the size of Bloom filters that must be maintained.

Minimizers have been used for some of these and similar applications [5–7]. Minimizers are the lexicographically first k-mer within a window of length L, which were introduced by Roberts *et al.* [2] for genome assembly. MSP [8] compresses k-mers by hashing them to their 4-mer minimizer to efficiently construct a de Bruijn graph for assembly. SparseAssembler [9] represents the de Bruijn graph using only every g-th k-mer in the sequence (and has also been implemented using minimizers). Kraken [10] uses minimizers to speed up database queries for k-mers during metagenome sequence classification. The Locally Consistent Parsing (LCP) [11] provides the concept of "core substrings" which, like minimizers, are guaranteed to be shared by long enough identical strings. The SCALCE software package [12] uses core substrings to compress DNA sequences.

A universal set U_{kL}, if it can be found, has a number of advantages over minimizers for these applications. First, the set of minimizers for a given collection of reads may be as dense as the complete set of k-mers, whereas we show below that U_{kL} is often smaller by a factor of k. Second, for any k and L, the set of universal k-mers needs to be computed only once and not recomputed for every dataset. Third, the hash buckets, sparse suffix arrays, and Bloom filters created for different datasets will contain a comparable set of k-mers if they are sampled according to U_{kL}. The universal set of k-mers also has the advantage over dataset-specific sets because one does not need to look at all the reads before deciding on the k-mers to use, and one does not need to build a dataset-specific de Bruijn graph to select covering k-mers.

The need for faster and more memory efficient genomic analysis methods is rapidly increasing as fast as the size and depth of sequencing data is increasing. The NIH Sequence Read Archive, for example, contains over 3.5 petabytes of sequence data and is growing at a fast pace. Increased use of RNA-seq in many conditions and in clinical settings leads to high processing burdens. Metagenomic sampling at increasing depth to quantify and assemble microbes from environmental samples leads to even larger sequencing datasets. New ideas in indexing, data structures, and algorithms are essential to keep computational pace with this data generation. The minimizer idea has been extremely successful in re-

ducing computational requirements. The universal set of k-mers proposed here will lead to further improvements in speed and memory.

The problem is also of theoretical interest as it can be rephrased as an equivalent problem on the complete (original) de Bruijn graph (see Definition 1 below). This is the viewpoint we take for most of this article:

Problem 2. Given a de Bruijn graph D_k of order k and an integer L, find a smallest set of vertices U_{kL} such that any path in D_k of length $L-k$ passes through at least one vertex of U_{kL}.

A solution to this problem may reveal additional hidden structure contained within the class of de Bruijn graphs.

We show that the problem of finding a minimum-size k-mer set that hits every string in a given set of L-long strings is NP-hard, further motivating the need for a universal k-mer set. We provide a heuristic called DOCKS that is based on the combination of three ideas. First, we use a decycling algorithm [13] to convert a complete de Bruijn graph into a directed acyclic graph (DAG) by removing a minimum number of k-mers, building off an implementation by Knuth [14]. We then supply a novel dynamic program to score remaining k-mers by the number of remaining length-ℓ paths that they hit. Finally, we use that dynamic program in a greedy heuristic to select the additional k-mers and produce a small universal set $\hat{U}_{kL}$, which we show empirically to often be close to the optimal size. Our use of a greedy heuristic is motivated by providing a proof that finding a small ℓ-path cover in a graph G is NP-hard even when G is a DAG.

DOCKS provides the first practical solution to the identification of universal sets of k-mers. The software is freely available on `acgt.cs.tau.ac.il/docks/`, as are universal sets of k-mers over a range of values of L and k. We report on the size of the universal k-mer hitting set produced by DOCKS and demonstrate on two datasets that we can better cover sequences with a smaller set of k-mers than is possible using minimizers. Our results also provide a starting point for additional theoretical investigation of these path coverings of de Bruijn graphs.

2 Definitions

Definition 1 (de Bruijn Graph). *A de Bruijn graph of order k over alphabet Σ is a directed graph in which every vertex has an associated label (a string over Σ) of length k (k-mer) and every edge has an associated label of length $k+1$. There are exactly $|\Sigma|^k$ vertices in a de Bruijn graph, each representing a unique k-mer. If an edge (u,v) has a label l, then the label of u must be a k-prefix of l and the label of v must be a k-suffix of l.*

A *complete* de Bruijn graph contains all possible edges, which represent together all $(k+1)$-mers over Σ. Every path in a de Bruijn graph represents a sequence. A path $v_0, e_0, v_1, e_1, v_2, \ldots, v_n$ of length n spells a sequence s of length $n+k$ such that the label of v_i occurs in s starting at position i for all $0 \leq i \leq n$,

and the label of e_i occurs in s starting at position i for all $0 \leq i \leq n-1$. Note, that vertices may repeat in a path.

We will need two bits of terminology involving k-mers intersecting and not intersecting sequences over an alphabet Σ:

Definition 2 (hits). *K-mer w* hits *sequence S if $w \subseteq S$, i.e. w appears as a contiguous substring in string S. K-mer set X* hits *sequence S if $\exists w \in X$ s.t. $w \subseteq S$. Denote $hit(w, L) = \{S \in \Sigma^L \mid w \subseteq S\}$ for k-mer w and length L. Denote $hit(X, L) = \bigcup_{w \in X} hit(w, L)$.*

Definition 3 (avoids). *Sequence S* avoids *k-mer w if $w \not\subseteq S$. Sequence S* avoids *k-mer set X if $\forall w \in X, w \not\subseteq S$. Denote $avoid(w, L) = \Sigma^L \setminus hit(w, L)$ for k-mer w and similarly $avoid(X, L) = \Sigma^L \setminus hit(X, L)$ for k-mer set X.*

3 Methods

It is not known how to efficiently find a minimum universal (k, L)-hitting set. As we show in Section 4, the corresponding problem when restricted to a given set of input sequences is NP-hard (Sec. 4.1). Here, we give a practical heuristic to find small (but non-optimal) universal k-mer sets. This algorithm works in two steps: first it finds and removes a minimum-size k-mer set hitting all infinite sequences, and then it removes additional k-mers to hit all remaining L-long sequences. We now describe these two steps in detail.

3.1 Finding a minimum k-mer set hitting all infinite sequences

The problem of finding a minimum-size k-mer set hitting all infinite sequences is known in the literature as finding an 'unavoidable set' of constant length [15]. This is a set of words of the same length k that hits any infinite word (but finite words may avoid the set). The problem of finding an unavoidable set for a given k can be solved in time polynomial in the output size [15]. The original algorithm is due to Mykkeltveit [13]. Its running time is $O(kM(k))$, where $M(k)$ is the size of the minimum unavoidable set. $M(k)$ converges to $|\Sigma|^k/k$ (an exact formula is given in Section 5.1, Equation 5), so the running time is $O(|\Sigma|^k)$.

An unavoidable set of constant length k is equivalent to a set of vertices in a complete de Bruijn graph of order k whose removal turn it into a DAG. Each k-mer in the set corresponds to a vertex, and the removal of vertices from every cycle guarantees that no infinite sequence is represented as a path in the graph. This set is known as a *decycling set*.

3.2 A greedy algorithm to hit all remaining L-long sequences

Unfortunately, finding an unavoidable set is not enough, as there may be L-long sequences that avoid that set. Thus, we need additional k-mers to hit those. If we consider the graph formulation, after removal of the unavoidable set from the

graph, we are left with a directed acyclic graph, which may contain $(L-k)$-long paths representing L-long sequences. We need to remove additional vertices, so that there is no path of length $\ell = L-k$. The problem of finding a minimum-size set of vertices that hit all ℓ-long paths in a directed acyclic graph is NP-hard, as we prove in Subsection 4.2. Therefore, we give a heuristic algorithm.

Our algorithm is based on the greedy algorithm for the minimum hitting set [16]. We define the *hitting number* $T_\ell(v)$ of a vertex v as the number of paths of length ℓ that contain it. The main observation is that we can calculate the hitting number of each vertex efficiently using dynamic programming. The solution is based on calculating the number of paths of length i that terminate at vertex v, and the number of paths of length i that start at vertex v, for all $v \in V$ and $0 \leq i \leq \ell$. Then, the number of ℓ-long paths through v is directly computable from these values by breaking any path into a i-long path ending at v and a $(\ell - i)$-long path starting at v, for all possible values of i. We set $\ell = L - k$ to get the hitting number of each vertex.

Specifically, let $G' = (V', E')$ be the directed acyclic graph, after removing the decycling set. Denote by D and F matrices of size $|V'| \times (\ell+1)$, where $D(v,i)$ is the number of i-long paths in G' starting at vertex v and $F(v,i)$ is the number of i-long paths ending at vertex v.

The calculation of D and F is as follows:

$$D(v,0) = F(v,0) = 1, \forall v \in V' \tag{1}$$

$$D(v,i) = \sum_{(v,u)\in E'} D(u, i-1) \tag{2}$$

$$F(v,i) = \sum_{(u,v)\in E'} F(u, i-1) \tag{3}$$

To get the number of ℓ-long paths vertex v participates in, we sum:

$$T_\ell(v) = \sum_{i=0}^{\ell} F(v,i) \times D(v, \ell - i) \tag{4}$$

The running time is proportional to the sum of all vertex degrees (which is $\Theta(|E|)$) times ℓ, giving a running time of $O(|\Sigma|^k \cdot \ell)$ for $\ell = L - k$.

3.3 The complete DOCKS algorithm

To get the complete algorithm, we combine the two steps. First, we find a decycling set in a complete de Bruijn graph of order k and remove it from the graph. Then, we repeatedly remove a vertex v with the largest hitting number $T_\ell(v)$ until there are no ℓ-long paths, recomputing $T_\ell(u)$ for all remaining u after each removal. This is summarized below (Algorithm DOCKS).

Finding the decycling set takes $O(|\Sigma|^k)$, as the size of the set is $\Theta(|\Sigma|^k/k)$ and the running time for finding each k-mer is $O(k)$ [13]. In the second phase, each iteration calculates the hitting number of all vertices using dynamic programming in time $O(|\Sigma|^k L)$. The number of iterations is $1+p$, where p is the number

Algorithm 1 DOCKS: Find a small k-mer set hitting all L-long sequences

1: Generate a complete de Bruijn graph G of order k, set $\ell = L - k$.
2: Find a decycling vertex set X using Mykkeltveit's algorithm.
3: Remove all vertices in X from graph G, resulting in G'.
4: **while** there are still paths of length ℓ **do**
5: Calculate the number of starting and ending i-long paths at each vertex, for $0 \leq i \leq \ell$.
6: Calculate the hitting number for each vertex.
7: Remove a vertex with maximum hitting number from G', and add it to set X.
8: **end while**
9: Output set X.

of vertices removed. Thus, the total running time is dominated by steps 4–8 and is $O((1+p)|\Sigma|^k L)$.

4 Complexity

4.1 NP-hardness of MINIMUM (k, L)-HITTING SET

The problem of finding a dataset-specific hitting set is NP-hard, further motivating the need for the design of a universal k-mer set:

MINIMUM (k, L)-HITTING SET
INSTANCE: Set S of L-long sequences over Σ and k.
VALID SOLUTION: Set X of k-mers s.t. $S \subseteq hit(X, L)$.
GOAL: Minimize $|X|$.

We prove that MINIMUM (k, L)-HITTING SET is NP-hard. For simplicity, we study the problem on DNA alphabet, but it can be easily generalized to any finite alphabet Σ. We show a reduction from HITTING SET [17]. While the problems look similar, HITTING SET is not a special case of the other since in HITTING SET the subsets are arbitrary, while in MINIMUM (k, L)-HITTING SET problem each subset is made of overlapping k-mers.

Theorem 1. *MINIMUM (k, L)-HITTING SET is NP-hard.*

Proof. Given an input to HITTING SET, a set S of subsets of $E = \{e_1 \dots e_n\}$, we generate an input to MINIMUM (k, L)-HITTING SET problem as follows: Denote by m the size of the maximum cardinality set, i.e. $m = \max_{S_i \in S} |S_i|$. We choose $\ell = \lceil \log_2(\max(m, n)) \rceil$, $L = 3\ell m$ and $k = 2\ell$. We map each set $S_i \in S$ to a k-long binary representation of i, where instead of bits we use nucleotides `C` and `G`. We map each element $e_j \in E$ to a k-long binary representation of j, where instead of bits we use nucleotides `A` and `T`. We call these representations the set's $\{C, G\}$-representation and the element's $\{A, T\}$-representation and denote them by $f_{CG}(S_i)$ and $f_{AT}(e_j)$.

We generate a sequence set T, which is the input to MINIMUM (k, L)-HITTING SET. For each set $S_i \in S$ we generate a sequence that contains

all of its elements' $\{A,T\}$-representations, each appearing twice consecutively and buffered by the set's $\{C,G\}$-representation. Formally, for the set $S_i = \{e_{i_1}, \ldots, e_{i_{|S_i|}}\}$ we create the sequence: $T_i := (\prod_{j=1}^{|S_i|} f_{AT}(e_{i_j}) \cdot f_{AT}(e_{i_j}) \cdot f_{CG}(S_i)) \cdot (f_{AT}(e_{i_1}) \cdot f_{AT}(e_{i_1}) \cdot f_{CG}(S_i))^{m-|S_i|}$ (here $\prod$ indicates concatenation). The new instance T is $\{T_1, \ldots, T_{|S|}\}$.

Denote by T^{OPT} an optimal solution to MINIMUM (k,L)-HITTING SET. If a k-mer contains a complete $\{A,T\}$-representation w, then the element $f_{AT}^{-1}(w)$ is in the optimal solution to HITTING SET. If a k-mer contains a complete $\{C,G\}$-representation w, then any element from the set $f_{CG}^{-1}(w)$ can be part of the optimal solution. The running time of the reduction is bounded by $O(|S| \times L)$ to generate the input sequence set T. In terms of m and n the running time is $O(|S| \cdot m \cdot (log(m) + log(n)))$.

We now prove the correctness of the reduction. We start with proving several properties of the solution.

Lemma 1. *A k-mer that contains a complete $\{A,T\}$-representation w can be replaced by k-mer ww to produce a hitting set of the same cardinality.*

Proof. The k-mer contains a complete $\{A,T\}$-representation w. Thus, it can only hit sequences that contain w. Since the sequences were constructed to contain two adjacent $\{A,T\}$-representations per element, and since this representation is unique, k-mer ww hits the same set of sequences. □

Lemma 2. *A k-mer that contains a complete $\{C,G\}$-representation can be replaced by a k-mer that contains two adjacent occurrences of any $\{A,T\}$-representation from this sequence to produce a hitting set of the same cardinality.*

Proof. A $\{C,G\}$-representation is unique to each sequence. Thus, it can only hit one sequence, and replacing it by any other k-mer from that sequence preserves the hitting properties of the set. □

We now prove the two sides of the reduction:

1. MINIMUM (k,L)-HITTING SET $\Rightarrow$ HITTING SET: all L-long sequences in T are hit by k-mers in T^{OPT}. By Lemmas 1 and 2 we can transform any hitting set to a hitting set of the same cardinality, but containing only k-mers over $\{A,T\}$. These correspond to elements in an optimal solution of HITTING SET. Assume contrary that there is a smaller solution U to HITTING SET. Then, the set $\{f_{AT}(w) \cdot f_{AT}(w) \mid w \in U\}$ hits all sequences in the k-mer hitting problem, and by that producing a smaller solution, contrary to its optimality.
2. HITTING SET $\Rightarrow$ MINIMUM (k,L)-HITTING SET: denote S^{OPT} an optimal solution to HITTING SET. Then, a set of k-mers $\{f_{AT}(w) \cdot f_{AT}(w) \mid w \in S^{OPT}\}$ is an optimal solution to MINIMUM (k,L)-HITTING SET. Assume contrary that there is a smaller solution U to MINIMUM (k,L)-HITTING SET. By Lemmas 1 and 2 there is a solution composed of k-mers over $\{A,T\}$. The set of element $\{f_{AT}^{-1}(w_{1:k/2}) \mid w \in U\}$ is a smaller hitting set in HITTING SET, contrary to its optimality. □

4.2 NP-hardness of MINIMUM ℓ-PATH COVER IN A DAG

Our heuristic to find U_{kL} searches for a minimum ℓ-path cover in the DAG created after removing a decycling set (Sec. 3.1). We show now that this problem is in general NP-hard (by a reduction from VERTEX COVER [17]) — motivating our use of a greedy heuristic to solve this subproblem.

MINIMUM ℓ-PATH VERTEX COVER IN A DAG
INSTANCE: A directed acyclic graph $G = (V, E)$ and integer ℓ.
VALID SOLUTION: Vertex set X s.t. $G' = (V \setminus X, E)$ contains no ℓ-long paths.
GOAL: Minimize $|X|$.

Theorem 2. *MINIMUM ℓ-PATH COVER IN A DAG is NP-hard.*

Proof. Given a graph $G = (V, E)$ as input to VERTEX COVER, we construct an instance to MINIMUM ℓ-PATH COVER IN A DAG as follows. We first remove from G any vertices that are incident to self-loop edges, since these must be part of any vertex cover. We then transform the remaining graph into a DAG by arbitrarily ordering the vertices of G, and directing the edges from lower-index to higher-index vertices. Since there are no self-loops, the result is a DAG $D = (V, A)$. The input to the ℓ-path cover is $I = (D, 1)$. The running time of the reduction is linear in the size of the graph.

A set of vertices $U \subseteq V$ is a vertex cover in G iff it intersects every edge in E. But this is true iff it hits every path of length 1 in D. Hence, U is a minimum vertex cover iff it is a minimum 1-path cover in D. □

5 Results

5.1 A theoretical lower bound for the number of k-mers

For a given k-mer w, its conjugacy class is the set of k-mers obtained by rotation of w. Conjugacy classes form cycles in the de Bruijn graph and form a partition of the k-mers. The number of conjugacy classes over all k-mers is given by [15]

$$C(|\Sigma|, k) = \sum_{i=1}^{k} |\Sigma|^{\gcd(i,k)}/k. \tag{5}$$

A decycling set necessarily contains a k-mer in each conjugacy class. Golomb's conjecture, proved by Mykkeltveit [13], states that the smallest decycling set has cardinality $C(|\Sigma|, k)$. Consequently, a (k, L)-hitting set has a size $\geq C(|\Sigma|, k)$.

Table 1 reports $L_{max} = \ell + k$, the length of the longest sequence in a complete de Bruijn graph after the decycling set is removed. The length of sequences avoiding the decycling set is too long for most applications. Additional k-mers must be selected to obtain a hitting set for smaller longest path.

Table 1. Maximum length of longest sequence avoiding an unaviodable set for different k. For each value k, a decycling set was removed from a complete de Bruijn graph, and the length L_{max} of the longest sequence, represented as a longest path, was calculated.

k	2	3	4	5	6	7	8	9	10	11	12	13	14
L_{max}	5	11	20	45	70	117	148	239	311	413	570	697	931

5.2 Computational results

We implemented and ran DOCKS over a range of k and L: $5 \leq k \leq 9$ with $20 \leq L \leq 200$, in increments of 10. These are typical values used for minimizers of longer k-mers and read lengths of short read sequences. We also implemented two random procedures that we compare to as baselines. One, termed "random", removes random vertices until no $\ell = L - k$ paths remains. The second, termed "decycling+random" (DR), first removes a minimum-size decycling set and then randomly removes vertices until no path of length $\ell = L - k$ exists. In both cases checking the termination condition is done by first testing if there are any cycles, and if there are no cycles, computing the maximum-length path, which takes linear time in a DAG.

The results are summarized in Figure 1. Our method outputs a set of k-mers that is much smaller than both random procedures. The results also show that there is a significant benefit in removing a minimum-size decycling set first and then additional vertices if we wish to hit all ℓ-long paths, as the random procedure that starts from the complete graph performs far worse than the one that is applied to the graph after removing an optimal decycling set. Note that random sometimes removes the same number for different values of L, since by the time it gets an acyclic graph, only short paths remain. As expected, the ratio compared to the lower bound decreases with L. It is easier to hit longer sequences as they contain more k-mers.

Table 2. Running times of the DOCKS algorithm for different k and L values. The user run time is in seconds (s) or minutes (m).

k / L	100	110	120	130	140	150	160	170	180	190	200
7	0.7s	0.5s	0.4s	0.4s	0.4s	0.4s	0.4s	0.4s	0.4s	0.4s	0.4s
8	11.1s	7.6s	4.3s	2.6s	1.3s	0.7s	0.7s	0.7s	0.7s	0.7s	0.7s
9	8.8m	6.9m	5.4m	4.2m	3.2m	2.5m	1.8m	1.4m	1.0m	0.7m	0.5m

Table 2 reports the running times for different values of k and L. For all instances, the dominant running time is of the second step, greedily finding an ℓ-path cover. This computation needs to be done only once per (k, L) pair. Running times were benchmarked on a high-performance cluster of nodes, where each combination of (k, L) was run serially on a separate IBM BladeCenter HS22 node with 20 GB of allocated RAM.

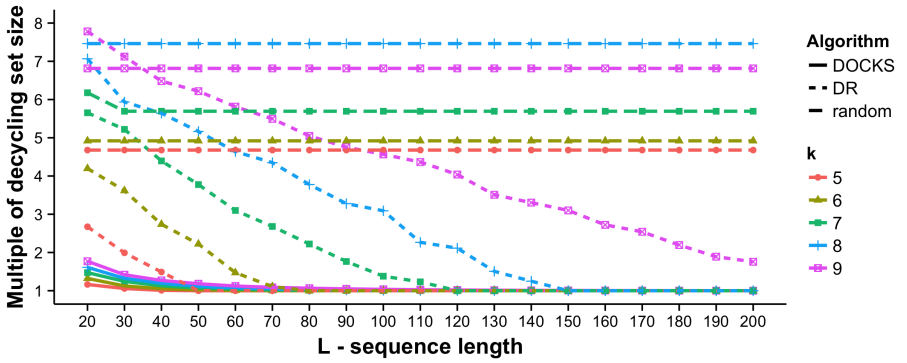

Fig. 1. Performance of DOCKS. For different combinations of k and L we ran DOCKS and two random procedures over the DNA alphabet. The results are shown in comparison to the size of the decycling set. When the ratio is 1, all the sequences avoiding the decycling set were of length shorter than L. DR: decycling+random.

5.3 Comparison to minimizers on bacterial genomes

Although the number of universal hitting k-mers for a given path length can be a significant proportion of all k-mers (around $|\Sigma|^k/k$), the actual number of k-mers hitting a given sequence is much less, even less than the number of minimizers. In Table 3, we compare the distribution of the universal hitting k-mers and the minimizers in two bacterial genomes. *Acetobacter tropicalis* (RefSeq NZ_CP011120) has a genome of 2.8 Mbp and a GC content of 47.8%. *Caulobacter vibriodes* (RefSeq NC_002696) is larger at 4.0 Mbp and has a higher GC content of 67.2%. For each genome, we computed the number of minimizers using $k = 8$ and a window length of 100. Also, for each window of 100 bases we found a k-mer from the set $\hat{U}_{kL}$ for $k = 8, L = 100$, computed by DOCKS. Each such window is guaranteed to contain at least one universal k-mer, and usually more than one. In each window, we select only one of the universal k-mers, the smallest one in lexicographic order. In addition, we measured the distances between the selected k-mers (minimizers or universal k-mers) and computed the mean and standard

Table 3. Comparison of the number of selected minimizers and universal k-mers, for $k = 8, L = 100$, and their distribution, in bacterial genomes. We report the mean distance ($\pm$std) between positions at which consecutive selected k-mers appear in the sequence.

	minimizers		universal k-mers	
	selected	mean distance	selected	mean distance
Acetobacter	3119	44.1 $\pm$ 33.6	2439	50.8 $\pm$ 29.2
Caulobacter	7315	47.2 $\pm$ 31.0	4585	51.2 $\pm$ 28.4

deviation of the distances. Using universal hitting k-mers instead of minimizers gives a smaller set of selected k-mers, which is also sparser in the sequence and more evenly distributed.

6 Conclusion

In this work, we presented the DOCKS algorithm, which generates a compact set of k-mers that together hit all L-long DNA sequences. DOCKS's good performance can be attributed to its two components. It first optimally removes a minimum-size set that hits all infinite sequences, which takes care of most L-long sequences. It then greedily removes vertices that hit remaining L-long sequences. Its feasibility stems from the first step, which runs in time $O(k)$ times the size of the output, and the second step, which uses dynamic programming to bound the running time to be quadratic in the output size times L.

A limitation of our approach is its heuristic nature, which does not guarantee any ratio over the optimal solution. Unfortunately, as we show, the general problem of finding a minimum (k, L)-hitting set is NP-hard. On top of that, even after removing a decycling edge set, the problem of finding a minimum set that hits all L-long sequences in a directed acyclic graph is NP-hard.

Some problems from this work remain open. First, is the problem of the universal (k, L)-hitting set polynomial in $O(|\Sigma|^k)$? The size of the output $\Theta(|\Sigma|^k/k)$ is doubly exponential in the size of the input (the parameters k and L), but the computational complexity remains open. Second, is the problem of ℓ-path cover in a DAG polynomial in the special case of directed acyclic subgraphs of de Bruijn graphs? Third, since the dominant run time is the second phase, which re-calculates the vertex hitting numbers on each iteration, can we update this number more efficiently after the removal of one vertex? Fourth, is there a tight upper bound on the number p of vertices that will be removed by the greedy heuristic? Fifth, can we give an upper bound or a tighter lower bound on the size of U_{kL}? Sixth, is the ℓ-path cover problem polynomial for $L > 1$?

In conclusion, we demonstrated the ability of DOCKS to generate compact sets of k-mers that hit all L-long sequences. These k-mer sets can be generated once for any desired value of k and L and then used easily for many different purposes. For example, there is a set of only 700 6-mers out of a total of 4096 that hits every sequence longer than 70 bases — a typical read length for many sequencing experiments — enabling efficient binning of reads. These sets of k-mers could improve many of the applications that use minimizers, as we showed that they are both smaller and more evenly distributed across typical sequences.

Acknowledgments

R.S. was supported in part by the Israel Science Foundation as part of the ISF-NSFC joint program 2015-2018. D.P. was supported in part by a Ph.D. fellowship from the Edmond J. Safra Center for Bioinformatics at Tel-Aviv University. This research is funded in part by the Gordon and Betty Moore Foundation's

Data-Driven Discovery Initiative through Grant GBMF4554 to C.K., by the US National Science Foundation (CCF-1256087, CCF-1319998) and by the US National Institutes of Health (R01HG007104). C.K. received support as an Alfred P. Sloan Research Fellow. Part of this work was done while Y.O., R.S. and C.K. were visiting the Simons Institute for the Theory of Computing.

References

1. Grabowski, S., Raniszewski, M.: Sampling the suffix array with minimizers. In: String Processing and Information Retrieval, Springer (2015) 287–298
2. Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., Yorke, J.A.: Reducing storage requirements for biological sequence comparison. Bioinformatics **20** (2004) 3363–3369
3. Karkkainen, J., Ukkonen, E.: Sparse suffix trees. In: Computing and Combinatorics: 2nd Annual International Conference, COCOON'96. Volume 2., Springer (1996) 219–230
4. Solomon, B., Kingsford, C.: Fast search of thousands of short-read sequencing experiments. Nature Biotech **34** (2016) 300–302
5. Movahedi, N.S., Forouzmand, E., Chitsaz, H.: De novo co-assembly of bacterial genomes from multiple single cells. In: 2012 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). (2012) 1–5
6. Deorowicz, S., Kokot, M., Grabowski, S., Debudaj-Grabysz, A.: KMC 2: Fast and resource-frugal k-mer counting. arXiv:1407.1507 [cs, q-bio] (2014)
7. Chikhi, R., Limasset, A., Jackman, S., Simpson, J.T., Medvedev, P.: On the representation of de Bruijn graphs. Journal of Computational Biology **22** (2015) 336–352
8. Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., Suri, S.: Memory efficient minimum substring partitioning. In: Proceedings of the VLDB Endowment. Volume 6., VLDB Endowment (2013) 169–180
9. Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., Douglas, W.Y.: Exploiting sparseness in de novo genome assembly. BMC Bioinformatics **13** (2012) S1
10. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biology **15** (2014) R46
11. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm. In: 37th Annual Symposium on Foundations of Computer Science. Proceedings. (1996) 320–328
12. Hach, F., Numanagi, I., Alkan, C., Sahinalp, S.C.: SCALCE: boosting sequence compression algorithms using locally consistent encoding. Bioinformatics **28** (2012) 3051–3057
13. Mykkeltveit, J.: A proof of Golomb's conjecture for the de Bruijn graph. Journal of Combinatorial Theory, Series B **13** (1972) 40–45
14. Knuth, D.E.: Unavoidable2. `http://www-cs-faculty.stanford.edu/~uno/programs/unavoidable2.w` (2003)
15. Champarnaud, J.M., Hansel, G., Perrin, D.: Unavoidable sets of constant length. International Journal of Algebra and Computation **14** (2004) 241–251
16. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of Operations Research **4** (1979) 233–235
17. Karp, R.M.: Reducibility among combinatorial problems. In: 50 Years of Integer Programming 1958-2008. Springer (2010) 219–241