BMC Bioinformatics

METHODOLOGY ARTICLE Open Access

Compact representation of *k*-mer de Bruijn graphs for genome read assembly

Einar Andreas Rødland

Abstract

Background: Processing of reads from high throughput sequencing is often done in terms of edges in the de Bruijn graph representing all *k*-mers from the reads. The memory requirements for storing all *k*-mers in a lookup table can be demanding, even after removal of read errors, but can be alleviated by using a memory efficient data structure.

Results: The FM-index, which is based on the Burrows–Wheeler transform, provides an efficient data structure providing a searchable index of all substrings from a set of strings, and is used to compactly represent full genomes for use in mapping reads to a genome: the memory required to store this is in the same order of magnitude as the strings themselves. However, reads from high throughput sequences mostly have high coverage and so contain the same substrings multiple times from different reads. I here present a modification of the FM-index, which I call the kFM-index, for indexing the set of *k*-mers from the reads. For DNA sequences, this requires 5 bit of information for each vertex of the corresponding de Bruijn subgraph, i.e. for each different $k-1$-mer, plus some additional overhead, typically 0.5 to 1 bit per vertex, for storing the equivalent of the FM-index for walking the underlying de Bruijn graph and reproducing the actual *k*-mers efficiently.

Conclusions: The kFM-index could replace more memory demanding data structures for storing the de Bruijn *k*-mer graph representation of sequence reads. A Java implementation with additional technical documentation is provided which demonstrates the applicability of the data structure (http://folk.uio.no/einarro/Projects/KFM-index/).

Background

High throughput sequencing is generating huge amounts of sequence data even from single experiments. The raw sequence data will typically be too much to keep in the memory of most off-the-shelf computers, and with sequencing technologies progressing faster than the improvements in computer memory, the memory challenge is likely to increase in the future.

One key property of the raw sequencing data is that it is highly redundant. Genomes are usually sequenced at high coverage, which means there will frequently be at least 30–50 reads covering the same region of the genome, differing primarily by sequencing errors. Processing of sequencing reads for genome assembly usually involves two crucial steps: error correction to remove or correct sequencing errors, and assembly of overlapping reads to produce a smaller number of assembled sequences.

A common approach for simplifying the processing of the sequence data is to consider all the *k*-mers of the reads: i.e. all the *k*-substrings of the reads if we view them as strings. This set of *k*-strings is then thought of as a subgraph of the de Bruijn graph of order $k-1$: i.e. one which has vertices corresponding to all $k-1$-substrings and edges corresponding to the *k*-substrings. Even if sequenced at high coverage, each *k*-mer is thus represented only once, reducing the redundancy of the sequence data considerably. However, direct storage of all *k*-mers in a single list will require *k* letters per *k*-mer, i.e. $2k$ bit of information for DNA sequences, which can be quite memory consuming when *k* is large.

Naively, one might expect that this could be greately improved. From each vertex in the graph, there may be 4

Correspondence: einarro@ifi.uio.no
[1]Center for Cancer Biomedicine & Departement of Informatics, University of Oslo, 0316 Oslo, Norway
[2]Department of Tumor Biology, Institute for Cancer Research, The Norwegian Radium Hospital, Oslo University Hospital, 0424 Oslo, Norway

BioMed Central

© 2013 Rødland; licensee BioMed Central Ltd. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

possible out-going (or in-coming) edges if the graph represents DNA sequences: one for each of the nucleotides. Encoding which of these exist in the graph should require only 4 bit of information per vertex; if most vertices have only one out-edge, this might even be reduced towards 2 bit of information per vertex by only encoding which of the 4 possible edges is actually found. Of course, this approach requires that the vertices be known, but one might envision that the information about the vertices could be reconstructed when walking the graph: when walking $k-1$ steps, all $k-1$ letters of the resulting vertex will be known.

A traversable representation of the de Bruijn subgraph is equivalent to storing a searchable index of all the k-substrings. By traversable, I mean that it is possible to efficiently walk the graph starting at any vertex, to check if any k-mer or $k-1$-mer is present as an edge or vertex in the graph, and preferably also to be able to retrieve the k-mers and $k-1$-mers represented by the graph. Thus, it is not only important that the data structure be compact, but efficient algorithms for using it are just as important.

A number of data structures exist that provide more compact storage of the de Bruijn subgraph than naive k-mer lists or maps. Conway et al. [1] were able to represent a de Bruijn subgraph with 12 G edges in 40.8 GB, i.e. 28.5 bit per edge, by using a compressed array. Other approaches reduce memory by storing only a subset of the k-mers [2-4].

An entirely different approach uses a Bloom filter to store a hashed set of k-mers [5] using only 4 bit per k-mer. This is a probabilistic data structure with a known false positive rate, but where false positive edges can be identified by not being part of longer paths. However, while this data structure is effective for checking if a k-mer is contained in the graph, it does not easily allow listing of all vertices or edges. An enhancement of this method, Minia [6], avoids critical false positives and also allows retrieval of all vertices, but at the cost of higher memory consumption.

Another memory-efficient solution uses the FM-index [7], which is based on the Burrows–Wheeler transform [8] used to represent a suffix array [9], to store the collection of reads in a compressed form [10]. The Burrows–Wheeler transform was originally developed for text compression and has the property that recurrent substrings in the text before the transform result in single-letter repeats in the transformed string. The FM-index adds auxiliary information on top of the Burrows–Wheeler transformed sequence that effectively turns it into a compactly stored suffix array. When concatenating the reads, the coverage makes the Burrows–Wheeler transformed sequence dominated by single-letter repeats which are highly compressible [10]. Effectively, it requires 2 bit per edge to store the nucleotide, which corresponds to specifying the in-edge (or out-edge) of a vertex, and additional memory to store the run-length of the nucleotide, which corresponds to the k-mer count. At least up to 50 times coverage, this data structure should be able to store one edge per byte if used to represent the de Bruijn subgraph of k-mers.

It should be noted that the ability of different methods to handle read errors varies. Some of the cited methods are intended to perform error correction by filtering k-mers by their frequency, while other methods assume that read errors for the most part have been corrected or excluded in advance.

I here provide a data structure with strong similarities to the FM-index, but which stores the de Bruijn subgraph representing the k-mer substrings rather than entire sequences. It is based on the idea of storing for each vertex which of the possible in-coming edges are actually present. For each vertex it thus needs one bit of information per letter in the alphabet, i.e. 4 bit per vertex for DNA sequences, plus some additional data. The additional data consists of a grouping of vertices which requires one extra bit per vertex, plus the equivalent to the FM-index for mapping in-coming edges to their parent vertices. This version of the FM-index, which I call the *kFM-index* since it applies to an index of k-substrings, can be generated from the stored data, but for computational speed a subset of the index is kept in memory. All in all, a de Bruijn subgraph for DNA sequences, including the stored subset if the index, can be stored using 5–6 bits per vertex if memory consumption is critical. In the case where most vertices are of degree 1, i.e. have one in-edge and one out-edge, the stored data may be compressed down to approximately half the size.

Like the FM-index, the kFM-index stores only one strand of DNA sequences, and is suitable for walking the graph in one direction. For genome assembly, one does not know in advance which strand the read is on, and so normally are required to ensure that both the k-mers of the reads and their reverse complements are added to the graph. Some data structures, e.g. most hashing strategies, can combine k-mers and their reverse complements, and thus require roughly half the number of items. For the kFM-index, however, it is necessary to add both the reads and their reverse complements. In doing this, one may walk in the opposite direction by switching to the reverse complement, although there will be some computational overhead in doing so.

The basic operations available on the kFM-index are similar to those of the FM-index. Each vertex is identified by it's index position, $i = 0, 1, \ldots, n-1$ where n is the number of vertices in the de Bruijn subgraph and the vertices are lexicographically ordered. For a given

string, the vertices having that string as a prefix, identified by the interval of index positions, can be found efficiently: the computational time is proportional to the length of the string. Given a vertex, identified by it's index position i, one can look up directly in the stored data which in-coming edges exist for that vertex. The index positions of the vertices from which the in-edges come can be computed efficiently. Thus, checking if a string exists as a path in the de Bruijn subgraph can be done. The reverse operation of identifying the string representation of a given vertex identified by index position i also exists, but is slower: time complexity is $O(k \lg n)$.

The kFM-index can be generated directly from a sorted list of in-edges, which is appropriate for amounts of sequence data that fit into the computer memory, although it should also be feasible to extend this by sorting the in-edges on disk: the time complexity is $O(Nk \lg \sigma \lg N)$ where N is the total length of the sequence data, and thus the number of items to be sorted, and $Nk \lg \sigma$ is the amount of data being sorted. Generation of kFM-indexes in memory from sequentially read sequence data can be done by splitting the raw sequence data into parts, generate kFM-indexes for each part, and then perform pairwise merges of these kFM-indexes. The time complexity of generating the kFM-index in this manner is essentially $O(Nk\sigma \lg(nm))$, where n is the number of vertices in the final de Bruijn graph (i.e. not counting identical k-mers), σ is the alphabet size, and m is the number of parts the initial sequence data is partitioned into. This has proven to be quite time consuming: in part because of the time complexity of the provided merge algorithm, but probably also in part due to an inefficient implementation. I expect that there is room for major improvements. In addition, these operations are all open to parallelisation.

Readers familiar with the FM-index will see the similarities to it, despite the fact that the FM-index represents all suffixes while this new data structure only stores information about k-substrings. Not only is the data structure very similar, but the functions and algorithms are also similar, or at least analogous, to those used with the FM-index. I therefore refer to this data structure as the kFM-index: an FM-index for k-substrings. And instead of pointing out the similarities throughout the article, I will point out differences where these are noteworthy.

A Java implementation of the data structure is provided as a demonstration.

Methods

Notation

Let Σ denote an *alphabet* of size $\sigma = |\Sigma|$, i.e. an arbitrary set whose elements we refer to as *letters*: for DNA sequences, $\Sigma = \{A, C, G, T\}$ and $\sigma = 4$. A string of length l, or an l-string, is an element of $x \in \Sigma^l$. Let $\Sigma^* = \cup_{l=0}^{\infty} \Sigma^l$ denote the set of all strings, including the empty string denoted ϵ. We denote the length of the string by $|x|$. If x and y are strings, xy denotes the concatenated string of length $|x| + |y|$; for sets U and V of strings, the set of concatenated strings is denoted $U \circ V = \{uv | u \in U, v \in V\}$.

We write $x < y$ to indicate that string x sorts lexicographically before y based on an ordering of the letters in Σ. In addition to the letters in Σ, we have two special characters \$ and ∞ with the properties that $\$ < a < \infty$ for all $a \in \Sigma$.

If x is an l-string, we write $x = x_1 \ldots x_l$ where $x_i \in \Sigma$ are the letters. For $p \leq q$, the $[p, q]$ substring $x_{[p,q]} = x_p \ldots x_q$ is a string of length $q - p + 1$: $x_{[p,p-1]}$ is the empty string. A substring $x_{[1,p]}$ at the start is referred to as a *prefix*, while a substring $x_{[p,l]}$ at the end is referred to as a *suffix*. The operation of trimming away the last letter is denoted $x^- = x_{[1,l-1]} = x_1 \ldots x_{l-1}$.

For $\mathcal{S} = (s_1, \ldots, s_N)$ a list of strings, i.e. $s_i \in \Sigma^*$, let $\mathcal{S}^{[k]} \subset \Sigma^k$ denote the set of length k substrings: i.e. $x \in \Sigma^k$ is contained in $\mathcal{S}^{[k]}$ if and only if there is a string $s \in \mathcal{S}$ with $x = s_{[p,p+k-1]}$ for some position p.

We denote the base 2 logarithm by $\lg x = \log_2 x$ which is convenient for quantifying information. Thus, the information required to specify one out of n options is $\lg n$ bit.

Problem description

Given a set $\mathcal{S}$ of strings, e.g. a set of sequencing reads, we will construct a compact representation of $\mathcal{S}^{[k]}$, i.e. the set of length k substrings, suitable for quickly checking if any particular k-string is present.

The data structure is best understood in terms of the *de Bruijn subgraph* representation of $\mathcal{S}^{[k]}$. This has vertices $V = \mathcal{S}^{[k-1]}$ and edges $E = \mathcal{S}^{[k]}$ where $e \in E$ is an edge from $e_{[1,k-1]}$ to $e_{[2,k]}$. It is a subgraph of the de Bruijn graph of order $k - 1$, i.e. with vertices Σ^{k-1} and edges Σ^k. Some authors may refer to this as a *word graph*, or even just a *de Bruijn graph*. Since the set of vertices can be deduced from the edges, storing $\mathcal{S}^{[k]}$ is effectively the same as storing the information encoded in the de Bruijn subgraph. However, the graph structure highlights the overlap between edges meeting at vertices.

While some authors focus on k as the length of the strings represented by the edges, others focus on the order of the graph which is the $k - 1$ length of the strings represented by the vertices. Since our purpose is to represent the k-mer composition of the sequences, it is natural to focus on k as the k-mer length. However, the

implementation of the algorithms is more naturally centered around the vertices, and so the Java implementation focuses on the order of the de Bruijn subgraph which is $k-1$.

The kFM-index data structure

The data structure for storing the k-substrings $\mathcal{S}^{[k]}$ from a set of strings $\mathcal{S}$ has similarities to the FM-index and the Burrows–Wheeler transformation. One similarity is that the data structure stores the prefixing letters, which represent the in-edges to vertices, and backtracks the de Bruijn subgraph through these in-coming edges rather than walking paths from beginning to end; the sequences, including the strings the vertices and edges represent, are thus reconstructed from the in-edge data when backtracking through the graph.

The initial de Bruijn subgraph representing the k-string composition $\mathcal{S}^{[k]}$ may contain any number of *final vertices*: i.e. vertices for which there are no out-going edges. These final vertices correspond to $k-1$-strings found only as suffixes of the strings in $\mathcal{S}$, and represent a problem as they cannot be reached by backtracking the de Bruijn subgraph. As the data structure does not store the $k-1$-strings for each vertex, but instead reconstructs these strings when walking the graph, these final vertices cannot be thus reconstructed. The solution is to add extra vertices and edges leading from these final vertices to a special final vertex from which we may start the reconstruction. See Figure 1 for an example.

Let $V_{\text{final}} \subset \mathcal{S}^{[k-1]}$ be a set that includes all $k-1$-strings which are final vertices in the graph with edge set $\mathcal{S}^{[k]}$: i.e. if $v \in \mathcal{S}^{[k-1]}$ and v is not a prefix of any string in $\mathcal{S}^{[k]}$, then v has to be in V_{final}. Ideally, in order to get the most compact representation of $\mathcal{S}^{[k]}$, we want V_{final} to contain only these strings. However, we might start off by letting V_{final} contain all $k-1$-suffixes of the strings in $\mathcal{S}$, knowing that the vertices required to be in V_{final} have to be a subset of these, and then later prune away superfluous edges and vertices. Hence, we permit V_{final} to be bigger than strictly required.

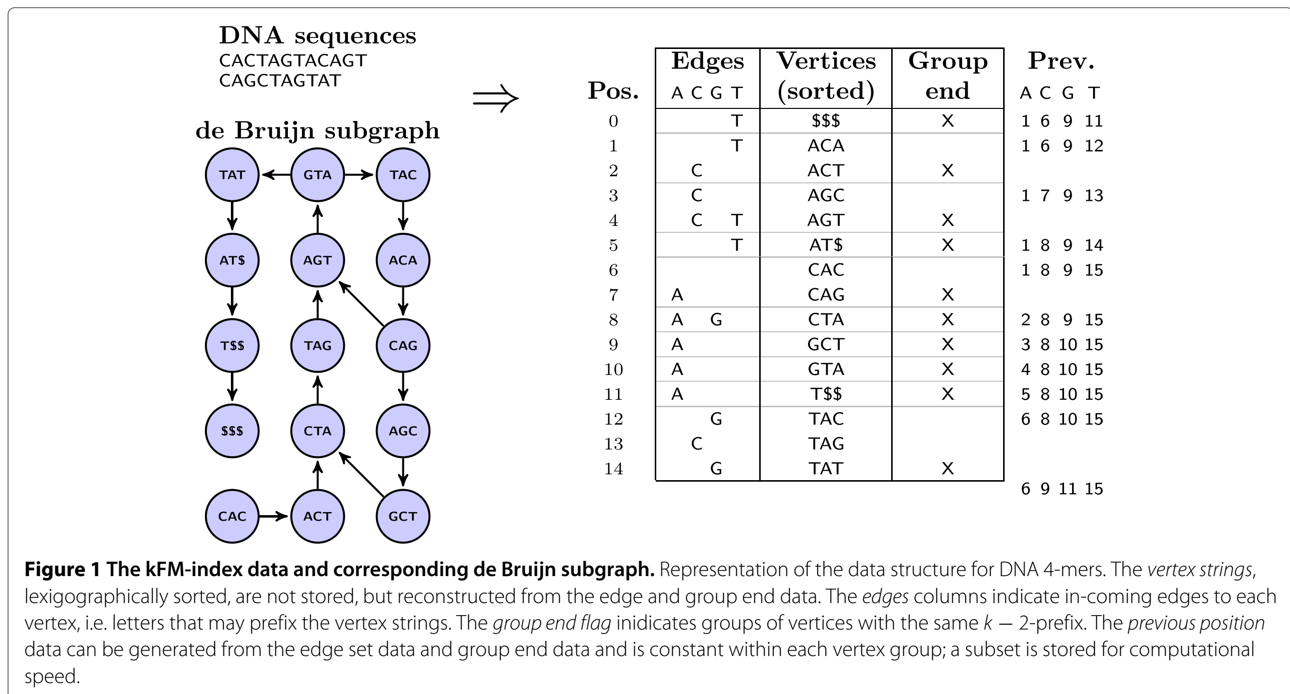
We now define the *final-completed* de Bruijn subgraph with paths added from each $v \in V_{\text{final}}$ to a special vertex, $\$^{k-1} = \$\ldots\$$ which we refer to as *the final vertex*, having vertices

$$V = \mathcal{S}^{[k-1]} \cup \left(V_{\text{final}} \circ \$^{k-1}\right)^{[k-1]} \cup \{\$^{k-1}\} \tag{1}$$

and edges

$$E = \mathcal{S}^{[k]} \cup \left(V_{\text{final}} \circ \$^{k-1}\right)^{[k]} \tag{2}$$

where $V_{\text{final}} \circ \$^{k-1} = \{v\$\ldots\$ \mid v \in V_{\text{final}}\}$ denotes the strings from which these additional paths are constructed. These are strings over an extended alphabet $\Sigma \cup \{\$\}$, where $\$$ is a special character that is sorted before any of the let-



Pos.	Edges A	Edges C	Edges G	Edges T	Vertices (sorted)	Group end	Prev. A	Prev. C	Prev. G	Prev. T
0				T	$$$	X	1	6	9	11
1				T	ACA		1	6	9	12
2		C			ACT	X				
3		C			AGC		1	7	9	13
4		C		T	AGT	X				
5				T	AT$	X	1	8	9	14
6					CAC		1	8	9	15
7	A				CAG	X				
8	A		G		CTA	X	2	8	9	15
9	A				GCT	X	3	8	10	15
10	A				GTA	X	4	8	10	15
11	A				T$$	X	5	8	10	15
12			G		TAC		6	8	10	15
13		C			TAG					
14			G		TAT	X				
							6	9	11	15

Figure 1 The kFM-index data and corresponding de Bruijn subgraph. Representation of the data structure for DNA 4-mers. The *vertex strings*, lexigographically sorted, are not stored, but reconstructed from the edge and group end data. The *edges* columns indicate in-coming edges to each vertex, i.e. letters that may prefix the vertex strings. The *group end flag* inidicates groups of vertices with the same $k-2$-prefix. The *previous position* data can be generated from the edge set data and group end data and is constant within each vertex group; a subset is stored for computational speed.

ters of Σ. The added vertices, i.e. those containing one or more \$ at the end, are referred to as *final-completing vertices* and are parts of paths leading to the final vertex. In fact, the final-completing vertices form a tree with the final vertex, $\$^{k-1}$, as the root. Note that, for the case where V_{final} is empty, we explicitly add the special vertex $\$^{k-1}$: this is purely a matter of convenience.

By this extension of the de Bruijn subgraph, we have ensured that there is exactly one final vertex that cannot be reached by backtracking the graph, namely the final vertex $\$^{k-1}$. When sorting the vertices lexicographically, this will always come first. Note that we do not require that the final vertex be reachable from the rest of the graph. If the original graph had V_{final} empty, this would be the case.

We may identify E with a subset of $\Sigma \times V$ describing the set of in-coming edges to each vertex, and will by abuse of notation say that the pair $(a, v) \in \Sigma \times V$ is an edge if the concatenation $av \in E$. We denote the in-coming edges to v by $E_v \subset \Sigma$: i.e. $E_v = \{a \in \Sigma \mid av \in E\}$.

Note that backtracking through this de Bruijn subgraph corresponds to reading the strings in the backwards direction, from the end of the string towards the beginning, just as with the FM-index. A variant of the data structure which naturally reads the strings in the forward direction can be obtained by performing the construction on the reversed strings, the only effect of which is on the sorting of strings in the index which would then be based on the reversed string.

Main data

Let $n = |V|$ be the number of vertices of the final-completed de Bruijn subgraph, and let $v_0, \ldots, v_{n-1}$ denote the vertices of V in lexicographic order; in particular, $v_0 = \$^{k-1}$, which is the only final vertex of the final-completed de Bruijn subgraph. The basic information required to store the final-completed de Bruijn subgraph of $\mathcal{S}^{[k]}$ is:

Edges: The set $E_{v_i} \subset \Sigma$ of edges from each vertex v_i is stored; i.e. the edge set E identified as a subset of $\Sigma \times V$. This may be encoded as a $\sigma \times n$ array, $\eta(a, i)$, with binary values: i.e. $\eta(a, i) \in \{\mathit{false}, \mathit{true}\}$ indicates if $av_i \in E$. The in-edges $E_i = \{a \mid av_i \in E\}$ to vertex v_i may be represented as a bit-mapped number on which set operations correspond to binary operations.

Group end flags: We group vertices $v \in V$ with the same $k-2$-prefix together: i.e. u and v are grouped together if $u^- = u_{[1,k-2]}$ and $v^- = v_{[1,k-2]}$ are identical. We indicate the group end by a flag f_i which is *true* if v_i is the last vertex in its group, *false* otherwise. This requires one bit of information per vertex.

More formally, these binary arrays take logical values, *true* or *false*, as defined by

$$\eta(a, i) \stackrel{\text{def}}{\iff} a \in E_{v_i} \tag{3}$$

and

$$f_i \stackrel{\text{def}}{\iff} v_i^- \neq v_{i+1}^- \text{ or } i = n-1 \tag{4}$$

where $v_i < v_{i+1}$ follows from the lexicographic sorting of the vertices. We could have added as a convention that $v_n = \infty^{k-1}$, in which case special handling of the final position would not have been required.

The grouping of vertices with the same $k-2$-prefix allows us to check which in-edges originate from the same vertices: for $a, b \in \Sigma$, $u, v \in V$, edges au and bv originate from vertices au^- and bv^- respectively, which is the same vertex if $a = b$ and $u^- = v^-$, which corresponds to checking if u and v are in the same vertex group.

Index to previous vertex position

In addition to the main data, there is an indexing function which may be generated from the main data. These are values per vertex group, i.e. constant within the groups of vertices with the same $k-2$-prefix. Storing the entire indexing function as auxiliary data would take a lot of memory, while computing everything from scratch would take a lot of time. Instead, a balance between memory and speed is obtained by storing a sparse subset, e.g. at regular intervals, and recompute the values in-between on demand. This index corresponds to the FM-index and provides a map from a vertex to the origin vertex of its in-edges.

For each letter $a \in \Sigma$, let $\tau(a)$ denote the number of vertex groups that contain an a in-edge. For $i = 0, \ldots, n$, let $c(a, i)$ denote the number of vertex groups prior to position i that contain an a in-edge: the vertex group containing v_i is not included in this sum. This makes $\tau(a) = c(a, n)$, since the vertices all have positions $i < n$.

For $a \in \Sigma$, $i = 0, \ldots, n$, let

$$\rho(a, i) = 1 + \sum_{b<a} \tau(b) + c(a, i) \tag{5}$$

where the summation of $b < a$ is for all $b \in \Sigma$ lexicographically prior to a, and the 1 corresponds to skipping the vertex $v_0 = \$^{k-1}$. Note that if a' is the letter following a in the alphabet Σ, i.e. $a' = a + 1$ if we consider the letters to be enumerated $0, \ldots, \sigma - 1$, then $\rho(a', 0) = \rho(a, n)$; and for a the last letter of Σ, i.e. $a = \sigma - 1$, we have $\rho(a, n) = n$.

The position array, $\rho(a, i)$, has the property that if vertex v_i has an in-coming edge $av_i \in E$, i.e. $a \in E_{v_i}$, this edge comes from vertex $v_{\rho(a,i)}$. A more general definition is that

$$\rho(a, i) = \min\{j \mid v_j \geq av_i^- \text{ or } j = n\} \tag{6}$$

where, as before, $v^- = v_{[1,k-2]}$ for $k-1$-strings $v \in V$ and $u \geq v$ refers to the lexicographic ordering of strings.

We may also note that, if we represent letters a by integers $0, \ldots, \sigma - 1$, we have $\rho(a, n) = \rho(a + 1, 0)$, and may define $\rho(an + i) = \rho(a, i)$, where $\rho(j)$ is again a non-decreasing function for $j = 0, \ldots, n \times \sigma$ with $\rho(0) = 1$ and $\rho(n \cdot \sigma) = n$. Representing $\rho(a, i)$ in terms of $\rho(an + i)$ is sometimes convenient, e.g. when we want to compute the inverse, and similarly we may use $an + i$ to represent an in-edge, or potential in-edge, (a, v_i).

As may be noted, the role of $\rho(a, i)$ in mapping from one vertex to another is essentially the same as the FM-index for mapping from one suffix to another. The main difference is in the grouping of vertices into groups, where counts are over vertex groups rather than individual vertices. The reason this vertex grouping is required is that the de Bruijn subgraph allows branching, i.e. for vertices to have more than one out-edge; the vertices in a vertex group share the same set of in-edges. The traditional FM-index could be envisioned as a de Bruijn subgraph with no branching, where each vertex has exactly one in-edge and one out-edge, and so the vertex groups would all consist of just one vertex.

Auxiliary data stored for computation speed

Storing $\rho(a, i)$ (or equivalently $c(a, i)$) as an array of integers requires $\sigma \times \lg n$ bit for each vertex. However, if $\rho(a, j)$ are known for some nearby j, the number of computational steps to compute $\rho(a, i)$ from $\rho(a, j)$ is proportional to $|i - j|$. So by storing only a subset of the $\rho(a, i)$, e.g. every qth position, the memory required for storing the auxiliary data is greatly reduced, but at the cost of computational time for determining $\rho(a, i)$. The partial storing of $\rho(a, i)$ is essentially the same as for the FM-index, and can be done in a number of different ways.

Let $0 = i_0 < \cdots < i_\zeta = n$ be the position for which $\rho(a, i)$ is to be stored, with $i_r - i_{r-1} \leq q$ for some chosen q. The stored values then consist of an array $\kappa(a\zeta + r) = \rho(a, i_r)$ where $a = 0, \ldots, \sigma - 1$ represent the letters. Thus, we have $\kappa(j)$ for $0 \leq j \leq \sigma\zeta$ with increments $0 \leq \kappa(j) - \kappa(j-1) \leq q$. Storing the entire κ array is, however, still space consuming unless q is allowed to be big, in which case computing $\rho(a, i)$ will take time.

Knowing that $\kappa(j)$ increases by values between 0 and q, we can write $\kappa(j) = u_j + qU_j$ where $U_j = \lfloor \kappa(j)/q \rfloor$ and $0 \leq u_j < q$ and store the u_j in a bit-packed array. The values U_j now have increments $\Delta U_j = U_j - U_{j-1} \in \{0, 1\}$. We store the increments ΔU_j as an array of bits, and a subset of the U_j from which the remaining U_j can then be computed efficiently. This allows us to select a much smaller value for q than would otherwise be feasible.

Java implementation

The Java implementation stores $\eta(a, i)$ and f_i as a $\sigma + 1$ bit block for each vertex i. These blocks are then packed into 64 bit words (long integers). For DNA sequences, each 64 bit word thus stores 12 vertices, each using 5 bit.

The stored values of $\rho(a, i)$ are expressed in terms of $\kappa(j) = u_j + qU_j$. The u_j are bit-packed into an array to preserve memory. For reconstruction of the U_j, every 64th U_j is stored, i.e. U_{64r}, and the increments ΔU_j are stored in blocks of 64 bits. Any U_j can then be computed efficiently from U_{64r}, $r = \lfloor j/64 \rfloor$, and the increments $\Delta U_{64r+1}, \ldots, \Delta U_j$ using operations on 64 bit words.

Fundamental functions for utilising the data structure

For a string x and $a \in \Sigma$, we define the function $\gamma(x)$ recursively by

$$\begin{aligned} &\gamma(\epsilon) = \gamma(\$) = 0, \quad \gamma(\infty) = n, \\ &\gamma(a) = \rho(a, 0), \quad \gamma(ax) = \rho(a, \gamma(x)). \end{aligned} \tag{7}$$

This function has a natural interpretation. If $|x| < k, \gamma(x)$ is the smallest non-negative integer i for which $x \leq v_i$, or $i = n$ if none such exists. If $|x| \geq k, \gamma(x)$ is the smallest integer i for which $x \leq v_i y$ for some string $y \in \Sigma^*$ so that $v_i y$ can be realised as a path in the de Bruijn subgraph (V, E), i.e. all k-substrings of $v_i y$ are in E, or $i = n$ if no such string exists. This property is formally proved in Lemma 2 in the Proofs of results appendix.

In order to utilise the data structure, for strings x with length $|x| < k$, we define two utility functions: the first vertex $v \geq x$ is

$$\alpha(x) = \gamma(x) = \min\{i \mid v_i \geq x \text{ or } i = n\} \tag{8}$$

while the first vertex with $v > x\infty$ is

$$\beta(x) = \gamma(x\infty) = \min\{i \mid v_i > x\infty \text{ or } i = n\} \tag{9}$$

Recall that $\infty > a$ for all $a \in \Sigma$, so $\beta(x)$ for $x \in \Sigma^l$, $l < k$, finds the first $v \in V$ for which $v_{[1,l]} > x$, while $\alpha(x)$ finds the first v for which $v_{[1,l]} \geq x$. Thus, α and β are defined by the property

$$\{v \in V \mid v_{[1,l]} = x\} = \{v_i \mid \alpha(x) \leq i < \beta(x)\} \tag{10}$$

for all $x \in \Sigma^l$, $l < k$, making them exactly the functions we need to identify all vertices starting with a given prefix.

Note that if we add the vertex $v_n = \infty^{k-1}$ to the list, we wouldn't have to specify the $i = n$ case in the above definitions. Again, this addition would purely be a matter of convenience and not have any practical impact.

Algorithms for using the data structure

The above described data structure encodes a de Bruijn subgraph representation of the k-substring composition of the strings $\mathcal{S}$. However, to utilise this representation, we need efficient algorithms.

Throughout the algorithms, vertices of V will be identified by their position $i \in \{0, \ldots, n-1\}$ in the lexicographically sorted list $v_0, \ldots, v_{n-1}$ where $n = |V|$. The string that each vertex represents will generally not be known.

The alphabet Σ is known from the start and the letters ordered. Computationally, it is natural to represent the letters by numbers $0, \ldots, \sigma - 1$ (ignoring the letter \$) since they are to be used as array indexes. However, for added readability, I will denote them as letters $a \in \Sigma$ in the algorithms rather as numerical indexes.

Computing the previous position $\rho(a, i)$ for arbitrary positions

A first step is to be able to compute $\rho(a, i)$ for arbitrary positions based on the stored data.

Let $0 = i_0 < \ldots < i_\zeta = n$ be the values for which $\rho(a, i)$ is stored, i.e. $\rho_{\text{store}}(a, i_r) = \rho(a, i_r)$, and define functions

$$\iota^+(i) = \min\{i_r | i_r \geq i\}, \quad \iota^-(i) = \max\{i_r | i_r \leq i\} \tag{11}$$

for pointing to the next or previous stored value. We may then compute $\rho(a, i)$ by aggregating from the vertex group containing $\rho(a, \iota^-(i))$ as in Algorithm 1.

Algorithm 1 Compute arbitrary $\rho(a, i)$ from previous stored value

```
function ρ(a, i)                              ▷ a ∈ Σ, i ∈ {0, ..., n}
    if i = 0 then return ρ_store(a, 0) end
    j ← ι⁻(i)                                 ▷ stored position j = i_r ≤ i
    p ← ρ_store(a, j)                         ▷ stored value
    while not f_{j−1} do j ← j − 1 end
    hasEdge ← false
    while j < i do
        if a ∈ E_j then hashEdge ← true end
        if f_j then                           ▷ end of vertex group
            if hasEdge then p ← p + 1 end
            hasEdge ← false
        end if
        j ← j + 1
    end while
    return p
end function
```

An alternative is to start at $\rho(a, \iota^+(i))$ and subtract contributions from vertex groups prior to this position, which can be done with a similar algorithm (see Additional file 1). To speed up the procedure, one may identify the nearest stored value, either previous or later, and use whichever of the two algorithms is appropriate. This will on average double the speed, and is done in the Java implementation.

Find all vertices with a particular prefix

The functions $\alpha(x)$ and $\beta(x)$ for strings x with length $|x| < k$ are both naturally expressed in terms of the the function γ. We note that γ gets called either as $\gamma(x) = \gamma(x\$)$ or as $\gamma(x\infty)$, and so we can express these as $\gamma(x) = \gamma(x\$) = \gamma(x, 0)$ and $\gamma(x\infty) = \gamma(x, n)$ where $\gamma(ax, i) = \rho(a, \gamma(x, i))$ and for $x = \epsilon$ the empty string $\gamma(\epsilon, i) = i$. Algorithm 2 details the computations.

Algorithm 2 Algorithm for computing $\gamma(x, i)$: then, $\alpha(x) = \gamma(x) = \gamma(x, 0)$ and $\beta(x) = \gamma(x\infty) = \gamma(x, n)$

```
function γ(x, i)
    for p = |x| to 1 step −1 do
        i ← ρ(x_p, i)
    end for
    return i
end function
```

By combining the call to $\alpha(x)$ and $\beta(x)$ into one function (Algorithm 3), it is possible to exploit the fact that when the interval is empty the two computations become identical. The return value $[\alpha(x), \beta(x)\rangle$ represents the interval $\alpha(x), \ldots, \beta(x) - 1$, and may be represented by a pair (α, β). If $\alpha(x) = \beta(x)$, the interval is empty: the function could be

modified to abort once it is clear that the resulting interval will be empty, or at least reduce computations by half once it is clear that $\alpha(x) = \beta(x)$.

Algorithm 3 Compute $[\alpha(x), \beta(x)\rangle$

```
function INTERVAL(x)
    i ← 0; j ← n                    ▷ start with [i, j⟩ = [0, n⟩
    for p = |x| downto 1 do
        i ← ρ(x_p, i); j ← ρ(x_p, j)
        — may abort returing null if i = j —
    end for
    return [i, j⟩
end function
```

When merging two kFM-indexes, $[\alpha(x), \beta(x)\rangle$ is computed numerous times on one kFM-index with x being vertices (or vertex prefixes) from the other kFM-index. When the result maps a vertex in one to an vertex in the other kFM-index, the two are merged; when a vertex v is mapped to an empty interval in the other kFM-index, the position $\alpha(v) = \beta(v)$ tells the merge procedure into which position it should be merged.

Backtracking through the de Bruijn subgraph

For a string x of length $|x| = l \geq k$, walking the de Bruijn subgraph path corresponding to x is most easily done by starting at the end of x and backtracking the graph towards the start of x. Algorithm 4 provides the algorithm for doing this: it will start at the $k-1$-suffix $x_{[l-k+2,l]}$ and backtrack one step at a time, exiting if the string leaves the graph.

Algorithm 4 Backtrack the de Bruijn subgraph for a string x of length $\geq k$

```
l ← |x|                          ▷ x string of length l = |x| ≥ k
v ← x_[l−k+2,l]                  ▷ ending vertex, i.e. k − 1-suffix
[i, j⟩ ← [α(x), β(x)⟩            ▷ INTERVAL(x)
if i = j then exit end           ▷ v does not exist
— i now points to the vertex x_[l−k+2,l] —
for p = |l − k + 1| downto 1 do
    if x_p ∉ E_i then exit end   ▷ no in-edge
    i ← ρ(x_p, i)                ▷ previous vertex
    — i now points to the vertex x_[p,p+k−2] —
end for
```

Identifying the string value of a vertex

If we start with a vertex identified by its position i, we can determine the string that vertex represents. In order to do so, we need a function $\rho^{\text{inv}} : \{0, \ldots, n-1\} \rightarrow \Sigma \times \{0, \ldots, n-1\}$ which has the property that

$$\rho^{\text{inv}}(i) = (a, j) \iff \rho(a, j) = i, \rho(a, j+1) = i + 1 \tag{12}$$

and where $\rho^{\text{inv}}(0) = (\$, 0)$. The pair (a, j) can be found through a binary search. However, since the computation of ρ is done in a stepwise manner starting at one of the stored values, a binary search should be performed on the stored values to identify the interval that contains the solution, and the stepwise procedure then followed until the solution is found.

The interpretation of $\rho^{\text{inv}}(i) = (a, j)$ is that a is the first letter of v_i, while j is the last vertex in the vertex group with $k-2$-prefix equal to the $k-2$-suffix of v_i: i.e. there is an edge from v_i to a vertex in the same vertex group as v_j. However, since the vertices in the same vertex group only differ by the last letter, and we don't have to determine this letter, we do not have to determine which vertex (or vertices) in this vertex group has an edge from v_i. By iterating this procedure $k-1$ times as in Algorithm 5, we can find the string v_i.

Algorithm 5 Return string v_i for position i

```
function VERTEX(i)
    x = string[k − 1]
    for p = 1 to k − 1 do
        (a, i) ← ρ^inv(i)
        x_p ← a
    end for
    return x
end function
```

Generating the kFM-index from a set of strings

The simplest way to generate the main data, i.e. the in-edge list and vertex group end flags, from a set of strings, is to generate the set of all k-substrings, including the final-completing strings with one or more more \$ attached at the end, sort and group them by their $k-1$-suffix, and then generate the in-edge list and group end flags directly from this. Once the main data, i.e. the binary arrays η and f_i representing the edge set E and the group end flags, have been generated, the auxiliary data can be generated from these.

This brute force approach requires a fair amount of memory since all k letters need to be stored for all k-substrings. The list of k-substrings thus takes up k times as much memory than the original strings from which they

are generated, and so is only feasible when the original string data is moderate in size.

If the string data is large, so that not all k-substrings of $\mathcal{S}^{[k]}$ can be kept in memory, the job may be split up. The set $\mathcal{S}$ of strings may be split up into smaller subsets, the kFM-index generated for each subset, and pairwise merging of kFM-indexes may then be performed to combine the subset based indexes into a kFM-index for the whole set.

Note that this procedure will generate a full set of final-completing vertices, i.e. those containing \$ at the end, even when they are not required by the kFM-index. We may reduce the kFM-index by checking which final-completing vertices are actually required in order to be able to reach the entire graph by backtracking from the final vertex. However, even if we do this for the initially generated kFM-indexes so as to ensure these contain minimal sets of final-completing vertices, when we merge the kFM-indexes for the string subsets, superfluous final-completing vertices may again occur when final-completing vertices required in one kFM-index are rendered superfluous by the edges of the other kFM-index. Hence, we may wish to prune away these final-completing vertices at the end, or in some of the intermediary merges.

Merging two kFM-indexes

If we have two kFM-indexes denoted A and B, one with n_A elements and the other with n_B elements, these can be merged in two steps. First, we merge the two lists into a list of length $n_A + n_B$. In the process of merging the two lists, rows representing the same vertex are not combined, but instead we mark the occurences where one vertex merged list is identical to the next one so that these may later be combined. In addition, vertex groups found in both A and B must be merged into one vertex group, which involves removing the group end flag from any vertex not at the end of the merged vertex group. After that, we sequentially pass through the $n_A + n_B$-length merged list, combining identical vertices into one vertex.

Instead of generating the $n_A + n_B$-length list in full, which would require the same amount of memory as the two original kFM-indexes of length n_A and n_B, we can represent the merge by a $n_A + n_B$ bit array indicating which of the two lists go into each position. Another bit array is used to mark identical vertices, which only needs n_A bit since we only need to store which vertices in A are also found in B. Finally, we use an $n_A + n_B$ bit array to mark vertices in the merged list that should not keep whatever group end flag it might have: this is required for vertex groups found in both A and B to ensure that only the final vertex in the group retains it group end flag.

Algorithm 6 Merge two kFM-indices

```
function MERGE(A, B)
    isA ← array[n_A + n_B] (false, ..., false)
    same ← array[n_A] (false, ..., false)
    group ← array[n_A + n_B] (false, ..., false)
    for l = 0 to k − 2 do
        PREMERGE(l, 0, 1, 0, 1)
    end for
    PREMERGE(k − 1, 0, n_A, 0, n_B)
    merged ← PERFORMMERGE()
    — compute and store ρ_store for merged —
    return merged
end function
```

It is sufficient to find the positions in the $n_A + n_B$-list of the n_A vertices from the kFM-index A. We may assume $n_A \leq n_B$ since that will require only n_A lookups; the n_B vertices from the kFM-index B will then be in the remaining positions. To do this, for all items $i = 0, \ldots, n_A - 1$ in the A list, compute the string v_i^A of that vertex. We then look up the position of v_i^A in the kFM-index B by computing $[\alpha_B(v_i^A), \beta_B(v_i^A)\rangle$. If the interval is empty, i.e. $\alpha_B(v_i^A) = \beta_B(v_i^A)$, the vertex goes into position $i + \alpha_B(v_i^A)$ and so we mark this position as containing a vertex from A. If the interval is not empty, the vertex v_i^A is found in position $\alpha_B(v_i^A)$ in the B list (and $\beta_B(v_i^A) = \alpha_B(v_i^A) + 1$), and so again we mark position $i + \alpha_B(v_i^A)$ as containing an A vertex, while the vertex from B takes position $i + \alpha_B(v_i^A) + 1$; in addition, we mark vertex i in the A list as having a duplicate in the B list.

We similarly need to iterate over all vertex groups in A, i.e. all non-empty $[\alpha_A(u), \beta_A(u)\rangle$ for $u \in \Sigma^{k-2}$. If the vertex groups $[\alpha_A(u), \beta_A(u)\rangle$ and $[\alpha_B(u), \beta_B(u)\rangle$ are both non-empty, we know that the vertices in the interval $[\alpha_A(u) + \alpha_B(u), \beta_A(u) + \beta_B(u)\rangle$ correspond to the u vertex group in the preliminary merged list. The last vertex, i.e. the one in position $\beta_A(u) + \beta_B(u) - 1$, will have its group end flag set from either list A or B. However, there will be another vertex with its group end flag set from the other list which may be in any of the other positions in the interval, and to ensure that this is unflagged, we mark positions $\alpha_A(u) + \alpha_B(u), \ldots, \beta_A(u) + \beta_B(u) - 2$ for group end flag removal.

Rather than process the items $i = 0, \ldots, n_A - 1$ sequentially, which requires computing VERTEX(i) for each and then looking these up in B, it is more efficient to recurse over all p-mers for $p = 1, \ldots, k-1$, each recursion adding all possible one letter prefixes. Thus, $l = 0$ corresponds to vertices, $l = 1$ to vertex groups, $[\alpha, \beta\rangle$ refers to index intervals corresponding to a given p-mer prefix in A or B. The function PREMERGE performs this recursion. It is called from MERGE where $l = k - 1 - p$ is the number of trailing \$.

Algorithm 7 Prepare merge: recurse over A intervals

```
function PREMERGE(l, α_A, β_A, α_B, β_B)
    if α_A = β_A then exit end                         ▷ empty
    if l = 0 then                                      ▷ vertex
        isA[α_A + α_B] ← true
        if α_B < β_B then same[α_A] ← true end
        exit
    end if
    if l = 1 and α_B < β_B then                        ▷ vertex group
        group[α_A + α_B], ..., group[β_A + β_B − 2] ← true
    end if
    for a = 0 to σ − 1 do                              ▷ prefixing letter
        PREMERGE(l − 1, ρ_A(a, α_A), ρ_A(a, β_A),
                 ρ_B(a, α_B), ρ_B(a, β_B))
    end for
end function
```

Once we have the $n_A + n_B$ bit array indicating which positions in the $n_A + n_B$ merge comes from A or B, another n_A bit array telling which vertices in A are also found in B, and a third $n_A + n_B$ bit array marking vertices for group end flag removal, we can merge the two lists sequentially using PERFORMMERGE.

Algorithm 8 Merge subroutine: perform merge based on merge information

```
function PERFORMMERGE
    merged ← empty list of kFM-data
    i_A, i_B ← 0
    while i_A + i_B < n_A + n_B do
        if not isA[i_A + i_B] then                     ▷ vertex from B
            E_new ← E^B_{i_B}                          ▷ in-edges from B
            f_new ← f^B_{i_B} and not group[i_A + i_B]
            i_B ← i_B + 1
        else if same[i_A] then                         ▷ vertex in A and B
            E_new ← E^A_{i_A} ∪ E^B_{i_B}              ▷ combined in-edges
            f_new ← f^B_{i_B} and not group[i_A + i_B + 1]
            i_A ← i_A + 1; i_B ← i_B + 1
        else                                           ▷ vertex from A
            E_new ← E^A_{i_A}                          ▷ in-edges from A
            f_new ← f^A_{i_A} and not group[i_A + i_B]
            i_A ← i_A + 1
        end if
        push merged ← (E_new, f_new)                   ▷ add to list
        — may remove data prior to i_A and i_B —
    end while
    return merged
end function
```

The three bit arrays used to facilitate the merge require $3n_A + 2n_B$ bit of extra data. The merge requires n_A lookups to find v_i^A and then $[\alpha_B(v_i^A), \beta_B(v_i^A)\rangle$ followed by the copying of all $n_A + n_B$ elements combining them into one when they represent the same vertex. In addition, the creation of the target list requires a temporary duplication of all the vertex and edge data, but this could be avoided by using a data structure in which the merged list is being created gradually as needed while memory used by A and B is gradually released as they are being merged. The Java implementation provided does this.

Pruning away superfluous final-completing vertices

The removal of superfluous final-completing vertices, i.e. vertices ending with one or more $ characters that are not required in order to avoid final vertices that have no out-edge, can be done by a few simple rules. We can perform these checks by a recursive approach, exploiting that the final-completing vertices form a tree with the final vertex, $\$^{k-1}$, as the root. We start at the final vertex, which is in position 0 of the kFM-index, and recursively backtrack through all in-edges at most $k-2$ steps to reach all final-completing vertices in V, performing the tests depth-first. We then identify edges and vertices that can be removed. After all final-completing vertices have been processed in this manner, we condense the list by removing the superfluous edges and vertices from the list.

Algorithm 9 Prune the index of unneeded final-completing vertices

```
function PRUNEFINALCOMPLETIONS
    V_prune ← empty list of vertices
    𝓔_prune ← empty list of pairs (position,edge set)
    CHECKUNUSED(k − 1, 0)                ▷ start at final vertex
    for (i, E_new) ∈ 𝓔_prune do E_i ← E_new end
    V ← V \ V_prune
    — recompute and store ρ_store —
end function
```

If $v \in V$ is a final-completing vertex, it is superfluous and can be removed if it has no in-edges. By having no in-edges, I include cases where the in-edges from vertices marked for exclusion have been removed: this is the reason why the tests must be done depth-first. Since ρ depends on which in-edges exist in each vertex group, superfluous in-edges can be removed immediately only for final-completing vertices where there is another vertex in the same vertex group with an in-edge from the same vertex (i.e. the same in-edge prefix). In general, edges and vertices must be marked for removal while the final-completing vertices are checked, and only removed after the checking is finished.

If $v = u\$ \in V$ is a final-completing vertex ending with a single $ and $e = av$ is an in-edge to v for some $a \in \Sigma$, the edge av can be removed if there is an a-in-edge to

another vertex in the same vertex group as v: i.e. if there is another vertex $ub \in V$ which has an in-edge $aub \in E$, the in-edge $av = au\$$ can be removed from the in-edges to v. The edge $au\$$ is superfluous since au can be reached by backtracking from ub. If all in-edges to v can be removed by this rule, the previous rule then allows v to be removed.

Algorithm 10 Prune recursal from vertex i ending in $\$^l$, return *true* if unneeded

```
function CHECKUNUSED(l, i)
    E_del ← ∅
    if l > 1 then                    ▷ only vertex in group when l > 1
        for a ∈ E_i do
            j ← ρ(a, i)
            if CHECKUNUSED(l − 1, j) then
                push V_prune ← j             ▷ remove vertex v_j
                E_del ← E_del ∪ {a}          ▷ remove in-edge a
            end if
        end for
        if E_del ≠ ∅ then
            push ℰ_prune ← (i, E_i \ E_del)
        end if
    else if not f_i then             ▷ other vertices in group
        j ← i
        while not f_j do
            j ← j + 1
            E_del ← E_del ∪ E_j
        end while
        E_i ← E_i \ E_del            ▷ can replace E_i immediately
    end if
    return (E_i \ E_del = ∅)         ▷ true if unused
end function
```

The pruning away of superfluous final-completing vertices is not required for the kFM-index to work, but it can reduce the memory required in cases where the number of such vertices is big. As such, depending on the number of final-completing vertices at any point of the processing or merging of kFM-indexes, one may choose to perform this pruning at the end or at some of the intermediary merges.

Pre-assembly

As a first step of sequence assembly, and a good task for assessing both the resulting de Bruijn subgraph and the efficiency of its use, uniquely determined paths of the graph are determined. This consists of two steps. First, the list of vertices are checked to identify all vertices that have in-degree or out-degree different from one:an algorithm is provided in the Additional file 1. The remaining vertices are simple, non-branching vertices that paths just pass through. Iterating over all branching/ending vertices, all possible paths passing through degree-one vertices are generated. When generating the sequence corresponding to a path, the sequence of the last vertex of the path needs to be found using Algorithm 5, while the rest is determined when backtracking through the in-edges.

The result is a list of non-simple vertices, and a list of all uniquely determined paths between these. An estimate of the number of such paths can be found simply from summing over the in-degrees of all branching vertices, but this may include paths consisting entirely of final-completing vertices. When generating the list of uniquely determined paths, those containing only final-completing vertices are excluded.

Results

Memory usage

The main data kept in memory is the $\sigma \times n$ binary array $\eta(a,i)$ and the binary flags f_i for marking the end of each vertex group. Direct storage of these in bit-packed arrays requires $\sigma + 1$ bit of information per node, so the total memory for storing the in-edge list and vertex group flags is

$$\mathfrak{Mem}\left[E,f\right] = n \times (\sigma + 1) \text{ bit.} \tag{13}$$

For efficient computation of the previous vertex position $\rho(a,i)$, a subset $\rho(a,i_r)$ is permanently stored for positions $0 = i_0 < \cdots < i_\zeta = n$. Direct storage of $\kappa(\zeta a + r) = \rho(a,i_r)$, where $a = 0,\ldots,\sigma - 1$ represents the letters, would require $\sigma\zeta \lg n$ bit since each position requires $\lg n$ bit. However, decomposing $\kappa(j) = u_j + qU_j$ where $i_r - i_{r-1} \le q$, and storing the u_i and $\Delta U_j = U_j - U_{j-1} \in \{0,1\}$, requires only $\lg q + 1$ bit for each value in κ: i.e. $\sigma\zeta(\lg q + 1)$ bit where $\zeta \approx n/q$ if the i_r are evenly spaced.

In order to efficiently compute arbitrary U_j, the Java implementation stores U_{64r}. Since each of these values require $\lg(n/q)$ bit of memory, the total memory requirement for storing the in-edge list, vertex group flags, and the data used to compute the previous vertex position, is

$$\mathfrak{Mem}\left[\rho_{\text{store}}\right] \approx \frac{n\sigma}{q}\left(\lg q + 1 + \frac{\lg(n/q)}{64}\right) \text{ bit.} \tag{14}$$

The memory saving construction used to compress κ could be repeated for the stored values $\kappa'_r = U_{\omega r}$ by writing this as $\kappa'_r = u'_r + 64U'_r$, but at an additional computational cost. However, for most practical cases, the term $\lg(n/q)/64$ is already a very minor part of the memory cost and not worth the computational overhead. By the time $n > 2^{64}$ becomes an issue, we have probably moved beyond 64 bit computers, in which case increas-

ing the block size of 64 to the higher word size ω changes the memory term to $\lg(n/q)/\omega$ without increasing the computational time.

Under the assumption that $n < 2^{\omega}$, where $\omega = 64$ is the word size of a 64 bit computer, the total memory requirement is

$$\mathfrak{Mem}\left[E, f, \rho\right] < n \times \left[\sigma + 1 + \frac{\sigma(\lg q + 2)}{q}\right] \text{ bit} \quad (15)$$

although there may be some additional overhead depending on how the data is bit-packed into arrays.

For DNA, $\sigma = 4$, which requires 5 bit of data per vertex. However, storing 12 vertices packed into a single 64 bit word leaves 4 unused bits, and so it effectively consumes $\approx$ 5.333 bit per vertex in the present Java implementation. For stored previous vertex positions, ρ_{store}, natural step sizes q between stored values are $q = 16$, 32 and 64, which adds 1.5 bit, 0.875 bit and 0.5 bit of memory usage per vertex, respectively.

Computational speed

Estimates of computational speeds are based on the assumption that $n < 2^{\omega}$ where ω is the word size: i.e. $\omega = 64$ on a 64 bit computer. This means that a number in the range 0 to n can be read from memory in one operation: for arbitrarily large n, this would require at least $(\lg n)/\omega$ operations. It also means single operations can operate on ω bits at the same time, although this is largely unexploited by the implementation.

The central algorithm that influences most kFM-index computations is that of computing arbitrary $\rho(a, i)$: Algorithm 1, or the extension of this provided in the Additional file 1 and implemented in the Java program. A subset of the values are stored in a compressed form and can be retrieved in constant time. If every qth value is stored, the time required to reconstruct and arbitrary $\rho(a, i)$ will on average be of order $O(q)$. However, since we will in practice select a fixed q of moderate size, which is sufficient to keep memory costs of the auxiliary data at a low level, and a fixed computational time remains even as we let q drop towards 1, we may consider the computation of $\rho(a, i)$ to be of constant time.

Algorithm 2 for computing $\gamma(x, i)$ for any string x requires $|x|$ calls to ρ, and thus has time complexity $O(|x|)$. Consequently, identifying the interval of vertices with prefix x (Algorithm 3) has time complexity $O(|x|)$. Algorithm 4 for backtracking the graph from vertex i along edges provided by the string x is essentially the same as the computation of γ, just with checks that the edges exist, and also has time complexity $O(|x|)$. The reverse computation of finding the string representation of a given vertex index, provided in Algorithm 5, requires solving for $\rho^{\text{inv}}(i)$ using a binary search, and is thus of time complexity $O(k \lg(n\sigma))$.

Constructing a kFM-index in memory, provided the memory is sufficient to hold a complete list of k-mers from the strings, has time complexity $O(Nk \lg \sigma \lg N)$ where $N = \|\mathcal{S}\|$ is the total length of the string data. The time is primarily required for sorting the list of in-edges generated from the strings, while construction of the kFM-index from the sorted list is linear in N.

Merging two kFM-indexes of sizes p and q using Algorithm 6 has worst case time complexity $O((p+q)k\sigma)$. This is due to Algorithm 7. The factor σ stems from checking sequentially for in-edges and could most likely be replaced by something more efficient should large alphabets be of interest. There is also room for improvement, as detailed in hte Additional file 1, e.g. by reducing the number of computations once conditions like $\alpha_B = \beta_B$ are met.

Construction of a kFM-index in memory by dividing the initial string data into m parts, generating kFM-indexes from each, and then merging these pairwise until a single kFM-index remains, has time complexity $O(Nk\sigma \lg(nm))$ using the present algorithms. At the lowest level, m kFM-indexes are generated, each from sorting approximately N/m k-words, which takes $O(Nk \lg \sigma \lg(N/m))$ time and is generally fast. During the first roughly $\lg(nm/N)$ rounds of pairwise merges, while the number of partitions is higher than the coverage, the total sizes of the kFM-indexes may still be $\approx N$, and so each round requires time $O(Nk\sigma)$. After that, since none of the kFM-indexes have more than n vertices, which is the size of the final de Bruijn subgraph, the time complexity drops by a factor of two for each new round of pairwise merges until a single kFM-index remains. The main part of the computations are the first $\lg(nm/N)$ or so rounds of pairwise merges, and so the final kFM-index takes $O(Nk\sigma \lg(nm/N)$. As N cannot increase without m increasing in proportion, since N/m in-edge k-words must be kept in memory, but m can increase independently if less memory is to be used, it is more natural to write this $O(Nk\sigma \lg(nm))$.

Benchmarking of the Java implementation

The Java implementation has not been optimised for speed: it runs on a single core, and prioritises memory consumption and code generality and readability over speed. However, it can still give a fair indication of the computational speeds, and indicate which are the bottlenecks.

The benchmarks on *E. coli* and simulated data were run on Java 6 under 64 bit Windows 7 on a standard office laptop: Dell Latitude E6320 with Intel Core i7-2620 2.70 GHz CPU and 8 GiB RAM. For *C. elegans* and the soil sample, it was run on a server with more memory and roughly twice the computational speed. The amount of RAM available

to Java was set with the option -Xmx. Note that for specifying computer memory, I use IEC prefixes ki-, Mi-, and Gi- which represent powers of 1024, while SI prefixes k-, M-, and G- represent powers of 1000.

All kFM-index constructions from read data added both reads and their reverse complements, discarding pairing information. Quality filtering consisted of removing bases with quality score less than 30, splitting the reads into fragments with higher quality bases. Unless otherwise stated, $k = 23$ were used: note that the Java implementation specifies the order $k - 1$, i.e. length of the vertex strings. The distance between stored values ρ_{store} was $q = 32$. This should require 6.2 bit per vertex as the actual memory usage on the data, including 0.33 bit due to the 4 unused bits in the pack of 12 vertices stored in a 64 bit Java long integer, although there would be some additional memory overhead from the program itself.

Memory usage during kFM-index construction was largely determined by the size of partitions, i.e. the maximal number of k-mers processed in each partition, which was set to different values to assess the time required to generate kFM-indexes by merging smaller indexes. For runs on the laptop, it was set to process at most 250 M words in each partition, which for $k \leq 28$ would require 2 GiB of memory with each word using 2×32 bit integers; on the server, the partition size was limited by implementation of the buffer as a Java array with at most 2 G 32 bit values, allowing at most 1 G words in each partition when $k \leq 28$. However, the peak memory usage reported here includes memory used and released by Java, but not yet garbage collected, and may reflect available memory more than actual use.

E. coli str. K-12 substr. MG1655

The implementation was evaluated on *E. coli* str. K-12 substr. MG1655 (SRA accession SRX000429, SRR001665, http://www.ncbi.nlm.nih.gov/sra/SRX000429) with 21 M 36 nt reads after discarding pairing information.

The quality filtered graph contained 13.4 M vertices and 13.4 M edges. This was processed in 2 parts and then merged, taking 7.7 minutes, with roughly a fifth of the time spent on merging kFM-indexes. Without quality filtering, the graph contained 81.7 M vertices and 83.7 M edges. For kFM-index construction, this was divided in 5 parts, which were then merged, taking 38 minutes, half of which was spent on merging the kFM-indexes together.

Once the final kFM-indexes had been constructed and the temporary memory freed up, the Java program used 18 MiB holding the quality filtered graph, and 69 MiB holding the unfiltered graph. At startup, before adding data, the Java program used 6–10 MiB of memory.

When given access to 6 GiB of RAM, the peak usage was a bit over 3 GiB. However, by reducing the available memory, peak usage could be reduced to just over 2 GiB, with no substantial change in computational time. The difference is due to memory that has been used and released, but not garbage collected. The main limitation was Java's ability to allocate the approximately 2 GiB block required to collect and sort in-edges for kFM-index construction.

In the quality filtered graph, 96.3% of the vertices were simple, i.e. had in- and out-degree one. Pre-assembly took 12.9 seconds and produced 427 k uniquely determined paths. In the unfiltered graph, only 89.9% of the vertices were simple, and as a consequence it produced 8.3 M uniquely determined paths using 227 seconds.

Simulated read data

Simulated sequence data were generated from two 1 Mnt DNA sequences, which were identical random sequences except from 0.1% random differences, intended to simulate a diploid organism with SNPs. Random 300 k 100 nt reads were generated with error rates 0%, 0.1%, and 1%, intended to represent the true sequences, reads with partial error correction, and raw reads at 30 times coverage.

The reads with no errors resulted in a graph with 2.05 M vertices and 2.05 M edges. This corresponds to 2×1 Mnt plus 22 extra vertices from each strand of the 0.1% SNPs. Accordingly, pre-assembly produced 6358 uniquely determined paths, which corresponds to the sequence between the SNPs and two variants for each SNP.

When the reads were given 0.1% error rate, which may be a realistic error rate after mild error correction, the graph size increased to 3.26 M vertices and 3.31 M edges. Pre-assembly resulted in 153 k uniquely determined paths

Including the full 1% read errors, as is common in uncorrected reads, the size increased to 15.7 M vertices and 16.1 M edges. Pre-assembly resulted in 1.37 M uniquely determined paths.

The time for constructing the kFM-index was 47, 49, and 64 seconds, respectively, for the three cases when enough memory was allocated to process all the data in one part. When the reads were spilt in 6 partitions and then merged, this time increased to 109, 139, and 327 seconds, respectively. The pre-assembly time was proportional to the size of the graph, and was 0.64, 3.7, and 35 seconds, respectively.

C. elegans str. N2

A kFM-index was generated from 67.6 M 100 nt reads on *C. elegans* str. N2 (SRA accession SRX02594, SRR065390, http://www.ncbi.nlm.nih.gov/sra/SRX026594).

With $k = 23$, this resulted in a graph with 255 M vertices and 259 M edges. With reads processed in 8 parts and merged, this took 5.4 hours on the server. Pre-assembly took 7.3 minutes and resulted in 12.8 M uniquely determined paths.

After completion of the kFM-index, the program holding the index used 290 MiB, some of which is unreleased memory after pruning away 61 M final-completing vertices at the end. The buffer used to store each of the 8 partitions took 8 GiB, while peak memory usage was 12 GiB.

Soil sample

An additional run was made on a soil sample (SRA accession SRX128885, SRR444039, http://www.ncbi.nlm.nih.gov/sra/SRX128885) with 37 M 76 nt reads.

Quality filtering left 2.59 G 23-mers to be processed, including the reverse complements. The resulting graph consisted of 2.86 G vertices and 2.87 G edges: the increase relative to the number of words added is due to final-completing vertices which represent read suffixes. This took 7.1 hours to generate, processing the data in 4 partitions before merging them. Most of this time was spent merging the kFM-indexes.

After completion, the program only occupied 2.2 GiB of memory, i.e. 6.6 bit per vertex including all overhead, indicating approximately 6% memory overhead relative to the 6.2 bit per vertex required by the data structure. During kFM-index construction, peak memory usage was 14 GiB.

Pre-assembly took 1.08 hours and resulted in 85.6 M uniquely determined paths. These appeared to be mostly from single reads.

Discussion

Memory requirements

The memory required to store $E = \mathcal{S}^{[k]}$ as a list of strings would be $|E| \times k \lg \sigma$ bit. If the edge set had been an arbitrary subset $E \subset \Sigma^k$, optimal storage would require

$$\begin{aligned} \mathfrak{Mem}\left[E \subset \Sigma^k\right] &= \lg \binom{\sigma^k}{|E|} \text{ bit} \\ &\approx |E| \times \left(k \lg \sigma - \lg \frac{|E|}{e}\right) \text{ bit} \end{aligned} \quad (16)$$

where the approximation assumes that E is a sparse subset of Σ^k. This is slightly better than storing E as a list of strings since it takes into account that the list of edges is unordered and contains each edge at most once. However, the claim that this is minimal required memory [1], is not strictly true, as $E = \mathcal{S}^{[k]}$ is not an arbitrary subset of Σ^k: it is induced by the sequences in $\mathcal{S}$. If most of the strings in $\mathcal{S}$ are much longer than k, this gives us ample room for reducing the memory usage.

Storing the sequences of $\mathcal{S}$, i.e. the data from which $\mathcal{S}^{[k]}$ is generated, requires

$$\mathfrak{Mem}[\mathcal{S}] = \|\mathcal{S}\| \times \lg \sigma \text{ bit} \quad (17)$$

where $\|\mathcal{S}\| = \sum_{x \in \mathcal{S}} |x|$ is the total length of the strings, assuming we do not have to store information about the lengths of the strings: this is true if all strings $x \in \mathcal{S}$ have the same length $|x| = l$, and a good approximation if the average length of the strings is much greater than the alphabet size.

If the strings of $\mathcal{S}$ are very different, in the sense that they do not share k-substrings to any particular extent, it will be more memory efficient to store the strings of $\mathcal{S}$ directly than storing $\mathcal{S}^{[k]}$, and little can be done to reduce this memory requirement. This was the case with the soil sample data analysed. In this case, the FM-index [11], which is based on the Burrows–Wheeler transform, provides a compact index for representing all substrings of $\mathcal{S}$: the Burrows–Wheeler transform requires $\|\mathcal{S}\| \times \lg \sigma$ bit, i.e. no more than the raw sequences.

When there is substantial overlap between the strings in $\mathcal{S}$, there are more compact representations of $\mathcal{S}^{[k]}$. For example, if the strings of $\mathcal{S}$ can be assembled into a smaller number of strings, $\mathcal{S}'$, of which the strings of $\mathcal{S}$ are substrings, we could use these strings instead to represent $\mathcal{S}^{[k]}$: if each k-string in $\mathcal{S}^{[k]}$ is found on average ν times, this would approximately reduce the required memory by a factor of ν. However, since assembly is a difficult problem, this is not a practical approach. In addition, the strings may not assemble well, e.g. due to read sequencing errors.

One fairly direct way to represent a de Bruijn subgraph, without storing a complete list of all k-mers, is to represent edges as pointers between vertices: e.g. an out-edge from a vertex may be stored as a pointer to the target vertex. Storing edges as pointers as well as the letters they correspond to requires $|E| \times (\lg |V| + \lg \sigma)$ bit of memory, which can be quite demanding for large graphs. For the 13.4 M graph representing the quality filtered *E. coli* reads, this would be 25.7 bit per vertex: more than four times as much as the kFM-index.

If most vertices are simple, i.e. have only one in-edge and out-edge, the number of pointers required may be drastically reduced by combining them into uniquely determined paths in the graph, as is done in Velvet [12]. Only one pointer would then be required for each such path, which for the quality filtered *E. coli* graph would be 427 k pointers each requiring 18.7 bit, resulting in $2 + 0.6$ bit per vertex for storing the nucleotide and the pointers. By combining vertices representing reverse complements into duplex vertices, the number of vertices and paths is halved, but pointers in both directions must be maintained, making this $2 + 2 \times 0.6$ bit per duplex vertex.

Even if merging non-branching vertices allows the graph itself to be stored more compactly, a map from arbitrary $k-1$-mers to vertices of the graph is required, and storing these pointers would cost at least $|V| \times \lg |V|$ bit of memory. A regular hash map would use additional memory

to identify $V \subset \Sigma^{[k-1]}$, but that is not strictly needed and could be avoided by smart hashing schemes. Still, for the quality filtered *E. coli* reads, a full map for the 13.4 M 22-mer vertices would require at least 23 bit per duplex vertex, and this would increase for larger graphs. If the 22-mers are mapped to the uniquely determined paths rather than to the individual vertices, this could be reduced to 18.7 bit per duplex vertex, which with the 3.1 bit per duplex vertex for storing the graph, still adds up to twice as much as required by the kFM-index, and would increase with the size and complexity of the graph. Many genome assemblers, such as ABySS [13], hash either vertices or edges, and are thus subject to this requirement. In fact, for any data structure that does not throughout make use of the fact that the vertices and edges tend to form long paths, the memory bound of equation (16) applies.

A natural method to compare the kFM-index against is the compressed Burrows–Wheeler transform of the concatenated reads used by SGA [10], due to the similarity between the kFM-index and the Burrows–Wheeler based FM-index. The Burrows–Wheeler transform of the reads would result in runs of identical bases corresponding to the coverage of the reads (unless broken up due to sequencing errors), and the SGA stores each run by its base and run length in a single byte. A naive comparison of memory requirements could be made against the in-edge data of the kFM-index which requires 5 bit per vertex; if most vertices have degree one, the in-edge data can be stored more compactly using only 2–3 bit per vertex since most vertices only require the base of the single in-edge to be stored. However, this is an unfair comparison since the kFM-index only stores the k-mers for a specific k, while the Burrows–Wheeler transform used by SGA stores all substrings, and thus allows k-mer frequencies to be found for all k. While the kFM-index specifically stores the de Bruijn k-mer subgraph, SGA uses overlap-based assembly and was not made with de Bruijn graphs in mind. SGA also does frequency based read error correction.

Memorywise, the kFM-index is comparable to the probabilistic de Bruijn graph using a Bloom filter [5], which requires approximately 4 bit of data per vertex. However, this can merge k-mers with their reverse complements, and this reduces the memory requirement by a factor of two relative to data structures like the kFM-index which has to store both. On the other hand, the Bloom filter is probabilistic, with a risk of introducing false vertices. While it can be used for checking if an arbitrary vertiex is present in the graph, although this requires a lower error rate and thus a little more memory spent per vertex, additional information is required to actually reproduce the graph. Certain removal of false vertices as well information required to reproduce the graph can be added [6], but requires more memory. In comparison, on the same *E. coli* read data, Minia [6] represented 4.7 M "solid" 23-mers (frequency at least 3) using 13.62+0.49 bit per duplex vertex, where each duplex vertex represents both a k-mer and its reverse complement. The 0.49 bit were used to store the marking structure required to reconstruct the graph. As such, the amount of memory per vertex is just slightly above the kFM-index, given that kFM-index needs two vertices to represent reverse complements where Minia only needs one.

As can be seen, the different data structures have different strength and weaknesses. Reductions in memory consumption tends to come at the expense of accessibilty of the stored information, computational speed and simplicity. Choice of a suitable data structure for any given problem thus depends on the computational needs, and there is no single best data structure for storing k-mer data from high throughput sequencing reads. The kFM-index is particularly made for storing the k-mer de Bruijn subgraph representation of sequence reads in a compact manner, yet allowing efficient random access of vertices and edges.

Further reduction in memory usage

The main memory usage of the kFM-index is the bit arrays $\eta(a, i)$ and f_i representing the in-edges and group end flags. These contain $\sigma + 1$ binary values per vertex: 5 bit per vertex for DNA sequence data. For arbitrary de Bruijn subgraphs, little can be done to reduce this substantially.

De Bruijn subgraphs constructed from real sequence data will, however, tend to be dominated by vertices with exactly one in-edge and no other vertices in its vertex group. In this case, for the majority of vertices, the data could simply be summarised by one letter, $a_i \in \Sigma$, representing the letter of the in-edge to vertex i: i.e. $E_i = \{a\}$ and $f_i = true$. This would reduce the required memory from $\sigma + 1$ bit per vertex, and potentially towards $\lg(\sigma)$ bit per vertex. The current implementation targets cases with small alphabets, like DNA which has $\sigma = 4$, and little emphasis has been placed on handling large σ where $\lg(\sigma)$ bit per vertex would make a big difference.

Since not all vertices can be thus represented, there would have to be some way of representing more general vertices as well, which would require both some memory overhead as well as computational overhead. For DNA sequencing reads, the conditions might be met where this approach could be useful, and could possibly reduce the memory consumption for storing the kFM-index by a factor of 2. The entropy of the vertex data, in-edge sets and group end flags, for typical kFM-indexes on DNA sequencing read data tends to be 2 to 2.5 bit per vertex, even for sequence data with read errors included.

Effects of read errors

I have not explicitly addressed to problem of read errors. Each read error may result in up to $k-1$ additional vertices, one for each of the $k-1$-substrings containing the read error. If the error rate or coverage is high, this may result in a substantial increase in the number of vertices. For example, if the error rate is ϵ, the chance that a particular k-word from a sequence will contain an error is approximately $k\epsilon$. If the average coverage is γ, this means there will tend to be $\gamma k\epsilon$ incorrect vertices for each correct vertex: i.e. the number of vertices in the de Bruijn subgraph when incorrect vertices are included will be $n \approx (1+\gamma k\epsilon)n_{\text{correct}}$ where n_{correct} is the number of correct vertices. E.g. with 1% error rate, 30 times coverage, and $k = 35$, this would make ten times as many incorrect vertices than correct vertices. The simulated read data illustrate this trend.

By also storing vertex counts, the kFM-index could be used for error correction based on $k-1$-mer frequencies. However, there already exist multiple algorithms and applications for excluding infrequent k-mers [10,14-18]. Furthermore, much of the advantage of the kFM-index over the compressed Burrows–Wheeler transform of the concatenated reads [10] is lost if the frequency count is to be stored, which is needed for error correction. Hence, the basic assumption has been that the primary usefulness of the kFM-index occurs when the majority of read errors have been removed or corrected in advance, and that error correction is more efficiently done prior to kFM-index construction.

The Java implementation allows filtering by base quality scores, removing bases with low reliability and splitting the sequence accordingly. On a number of benchmarking runs, this simple approach seemed to be able to remove the majority of read errors, and reduce the size of the resulting de Bruijn graph substantially. However, for genome assembly, further error correction would be required.

Effect of adding final-completing vertices

The vertices V consist of $\mathcal{S}^{[k-1]}$ with vertices added to allow paths from V_{final} to $\$^{k-1}$. There may be up to $k-1$ vertices added to V for each vertex in V_{final}. However, as long as $|V_{\text{final}}|$ is small compared to $|\mathcal{S}^{[k-1]}|$, this has little impact on the memory requirements. We may note that we can always get $|V_{\text{final}}| \leq |\mathcal{S}|$, so if the strings on average are much longer than k letters, the contribution of V_{final} is likely to be small.

For final-completing vertices to contribute substantially to the memory usage, there would have to be a substantial portion of read ends not found internally in other reads. This could happen for read data consisting of short reads with low coverage. It could also happen if there are frequent read errors at the flanks of the reads which are not filtered out or corrected.

Computational speed

The central operation in most uses of the kFM-index is the computation of the in-coming vertex position, $\rho(a,i)$. Algorithm 1 gives an implementation whose time complexity on average is $O(q)$, where q is the distance between the stored values ρ_{store}.

The Java implementation combines algorithm 1 with a similar algorithm provided in the Additional file 1 for computing $\rho(a,i)$ from the next stored value rather than the previous one, and selects the closest stored value, effectively halving the number of steps needed. A good balance between speed and memory usage is then to use $q=32$.

In the Java implementation, the average computational time of $\rho(a,i)$ is linear in q as could be expected. In the benchmarking, time per call to ρ during large kFM-index was estimated to $75.8 + 2.8q$ ns. Although some of this is likely overhead related to the merge operations, and the numbers will depend on the computer and implementation, it does give an indication that reducing q much below 30 is of limited use. One likely reason for this is the time required for random memory access is high, while repeated accessed to the same block of memory is fast due to caching.

I will treat calls to ρ as constant time in the subsequent analyses, although the present implementation does depend on q, under the assumption that q will remain fixed, typically between 30 and 64, and that in this range the memory overhead of the stored values ρ_{store} is moderate.

Low-level parallel computing of $\rho(a,i)$ using 64 bit words

Algorithm 1 works by processing one bit at a time. When doing this, the average number of steps required to compute $\rho(a,i)$ is proportional to the distance, q, between the stored values. It is, however, possible to parallell process these bit operations and thus exploit that e.g. a 64 bit processor can process a 64 bit word in one operation.

One such method is described in the Additional file 1, in which the number of steps for processing one word of data is proportional to the alphabet size σ. This method requires that the flags indicating if $a \in E_j$ for consecutive j are stored in one word, and the same for the group end flags f_j, so that the data for an entire interval of j positions can be retrieved from memory in a single operation. A replacement for Algorithm 1, provided in the Additional file 1, will then compute arbitrary $\rho(a,i)$ from one word of containing in-edge data for the letter a and one word with group end flags, allowing a distance $q = 64 - \sigma + 1$

between stored values: reducing the distance below this has no benefit.

The present Java implementation, however, does not use this approach.

Construction of the kFM-index

The main bottleneck at present consist of constructing the kFM-index from the strings. The provided algorithm in which the strings are partitioned into subsets, each subset converted to kFM-indices, and these kFM-indices recursively merged together, introduces substantial overhead in terms of memory and in computational time during construction.

In the FM-index setting, where $n = N$, a similar approach would have required time $O(n \ln n)$: if we split the data into 2^r sets, each iteration merges these pairwise using time $O(n)$ per round, with r rounds required. However, for the kFM-index, in the initial steps when the sequences are partitioned, the total number of vertices may be much greater due to $k - 1$-words present multiple times in the reads and thus entering into a large number of the subset kFM-indexes. The time consumption may therefore increase by a factor proportional to the coverage and become of order $O(Nk\sigma \lg(nm))$ where $N = \|\mathcal{S}\|$ is the total size of the sequence data and m is the number of partitions into which it is divided.

The algorithm for merging kFM-indexes could most likely be improved substantially, and the Java implementation provided is far from optimal: neither in terms of speed, nor in memory usage. Some minor improvements have been made in the implementation, reducing the number of calls to ρ when certain conditions are met (see Additional file 1 for details). However, even with further improvements, the method of partitioning and pairwise merging is inherently slow.

The construction of the kFM-index may be split up and run on separate CPUs; even the final mergers of two kFM-indexes can be split up, e.g. into σ^r different threads based on the first r levels of recursion. However, if the reads have high coverage, each of the subset kFM-indexes may already contain many of the same high-coverage vertices, and thus require almost as much memory and processing power as the final kFM-index.

For constructing the Burrows–Wheeler transform and the FM-index, there are more efficient algorithms [19,20] which rely on reformulating the Burrows–Wheeler transform in terms of a shorter sequence over a larger alphabet. These do not easily generalise to the kFM-index. In particular, a k-word in the original kFM-index may correspond to multiple different sequence locations, and these may appear as several different k-words in the the reformulated sequences. However, I have some hope that the induced sorting approach [19,21] may be adapted, although perhaps not quite as successfully as for the FM-index.

An alternative approach to kFM-index construction in memory is to split the reads up into edges representing the k-words of the sequences, and use the disk to help make a sorted list. A simple way to do this can be to partition the read data into parts that are small enough that the in-edge set can be stored as k-words and sorted in memory, write each partition to disk as a single file, and then merge these files to construct the final kFM-index. While this may require substantial temporary disk space, particularly for high coverage sequence data, it will most likely be much faster than the in-memory construction of the kFM-index presently implemented. This approach could also be divided between several computers.

Use with large alphabets

The algorithms, and the Java implementation, have been written with small alphabets in mind: in particular, DNA with $\sigma = 4$. Technically, they work for general alphabets, although the Java implementation cannot at present handle alphabets with $\sigma > 63$ since the vertex data are packed into a 64 bit word. However, the routines for merging two kFM-indexes involve iterating over the entire alphabet, adding a time factor σ to the merge procedure. For large alphabets, one might store the in-edge data in a more compact form than a bit vector, and could identify the relevant letters without having to check them one by one.

Conclusions

The kFM-index is a data structure that stores the k-words corresponding to the edges of a de Bruijn subgraph in a compact manner, while allowing efficient random access to vertices and edges. The data structure is made compact by avoiding the direct storage of k-words and pointers, which often are the main memory expense for storing de Bruijn subgraphs. The vertex and edge information is stored in a direct manner with each line in the data table representing a vertex, and each bit set in the in-edge bit array representing an edge. Thus, the compactness of the kFM-index data structure does not rely on compactification of the graph or compression of the data, and additional compression of the index is feasible.

The presently implemented method for in-memory construction of the kFM-index is uncompetitive for large data sets. However, there are multiple ways in which this could be improved. Also, as with the FM-index used by SGA [10], after the index has been constructed, the user can reuse it to try different assembly options and parameters.

One of the main approaches to de novo genome assembly using high throughput sequencing is to generate the de

Bruijn subgraph representing the k-mers of the reads, and multiple applications exist already for doing this. For large genomes and in meta-genomics, the memory required for representing the de Bruijn subgraph is one of the limiting factors. The kFM-index could replace existing, more memory demanding, data structures in existing genome assembly applications to allow them to process larger genomes or to run on off-the-shelf hardware where special, high-RAM computers have previously been required.

Availability

A Java implementation of the data structure and algorithms, together with additional technical documentation of the implementation, are freely available from http://folk.uio.no/einarro/Projects/KFM-index/. Improvements to the implementation, removing memory limitations and introducing parallel processing, and updated benchmarks, are provided on this web site.

Appendix

Mathematical approximations

Based on **Stirling's approximation**, $\ln n! \approx n \cdot \ln \frac{n}{e}$, we can approximate binomials by

$$\ln \binom{n}{x} = \ln \frac{n \cdots (n-x+1)}{x!} \lesssim x \cdot \ln \frac{n}{x/e} \tag{18}$$

which is a good approximation as long as $x \ll n$. Another bound is

$$\ln \binom{n}{x} \leq x \ln \frac{n}{x} + (n-x) \ln \frac{n}{n-x} \leq n \ln 2 \tag{19}$$

where the first inequality follows from $\binom{n}{x} p^x (1-p)^{n-x} \leq 1$ upon entering $p = x/n$, where $\binom{n}{x} (\frac{x}{n})^x (1-\frac{x}{n})^{n-x} \approx \sqrt{n/2\pi x(n-x)}$ indicating that the inequality is fairly tight, while the second inequality follows from $\sum_{x=0}^{n} \binom{n}{x} = 2^n$ and is only tight for $x/n \approx 1/2$.

Proofs of results

Definition 1. *For a de Bruijn subgraph $G = (V, E)$ as defined in equations (1) and (2), we say that a string x corresponds to a path in G if $|x| \geq k$ and all k-substrings of x are contained in E: i.e. $x = x_1 \ldots x_l$ corresponds to the path $e_i = x_{[i,i+k-1]}$ for $i = 1, \ldots, l-k+1$. For a string x of any length, we say that x is compatible with G if either x corresponds to a path in G (for $|x| \geq k$) or x is a substring of a vertex $v \in V$ (for $|x| < k$). We write $x \sim G$ to indicate that x is compatible with G.*

For convenience, we will include $v_n = \infty^{k-1}$ in the vertex list, although it is strictly speaking not part of the graph, and apply the convention that ∞^k corresponds to a path in G. The main effect of this is that the $\min\{i \mid \ldots\}$ expressions below take the value n if criteria for i are not otherwise met.

Lemma 2. *Let $\rho(a, i)$ be defined as in equation (6) and γ as in (7). For x a string in Σ^* or $\Sigma^* \circ \{\$, \infty\}$,*

$$\gamma(x) = \min\{i \mid x \leq v_i z \sim G \text{ for some } z \in \Sigma^*\}. \tag{20}$$

For $|x| < k$, this is just $\min\{i \mid x \leq v_i\}$, while for $|x| \geq k$ it requires a path starting at v_i (unless $x \leq v_i$).

Proof. For the empty string, $\gamma(\epsilon) = 0$, making equation (20) true. For $a \in \Sigma$, $\gamma(a) = \rho(a, 0)$, and equation (20) is essentially the definition of ρ; for $\gamma(\$) = 0$ the result follows as all $v_i \geq \$$, while for $\gamma(\infty) = n$ only $v_n \geq \infty$. We will then complete the proof by induction on $|x|$ using (20) as the induction hypothesis.

For $2 \leq |x| < k$, write $x = ay$ where $a \in \Sigma$. By the induction hypothesis, we know that $y \leq v_j$ when $j \geq \gamma(y)$. Since $|y| \leq k-2$, this is the same as $y \leq v_j^-$, and thus equivalent to $x = ay \leq av_j^-$. If $x = ay \leq v_i$, then either $v_i = av_j^-$ for some j in which case $x \leq v_i = av_j^-$ and $i \geq \rho(a, j)$, or no such v_j exists in which case the first letter $v_{i,1} > a$ and $i \geq \rho(a, n)$. The smallest i for which this makes $x \leq v_i$ in either case corresponds to $\rho(a, j)$ where $j = \gamma(y)$.

For $|x| \geq k$, again write $x = ay$ where we know that $y \leq v_j z \sim G$ when $j \geq \gamma(y)$. This means that $x = ay \leq av_j z = v_i bz \sim G$ for $i = \rho(a, j)$ and where the condition $j \geq \gamma(y)$ becomes $i \geq \rho(a, j)$. □

Additional file

Additional file 1: Supplementary information on the kFM-index.

Competing interests

The author declares that he has no competing interests.

Acknowledgements

I am most grateful to Prof. Wen-Lian Hsu and his research group at the Institute of Information Science at Academia Sinica for hosting me during my stays in Taipei, Taiwan, during which parts of this work was conducted.

Received: 2 May 2013 Accepted: 14 October 2013
Published: 23 October 2013

References

1. Conway TC, Bromage AJ: **Succinct data structures for assembling large genomes.** *Bioinformatics* 2011, **27**(4):479–486.
2. Ning Z, Cox AJ, Mullikin JC: **SSAHA: a fast search method for large DNA databases.** *Genome Res* 2001, **11**(10):1725–1729.
3. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA: **Reducing storage requirements for biological sequence comparison.** *Bioinformatics* 2004, **20**(18):3363–3369.
4. Ye C, Ma ZS, Cannon CH, Pop M, Yu DW: **Exploiting sparseness in de novo genome assembly.** *BMC Bioinformatics* 2012, **13**(Suppl 6):S1.

5. Pell J, Hintze A, Canino-Koning R, Howe A, Tiedje JM, Brown CT: **Scaling metagenome sequence assembly with probabilistic de Bruijn graphs.** *Proc Natl Acad Sci U S A* 2012, **109**(33):13272–13277.
6. Chikhi R, Rizk G: **Space-efficient and exact de Bruijn graph representation based on a Bloom filter.** In *Algorithms in Bioinformatics, Lecture Notes in Computer Science*. Edited by Raphel B, Tang J: Springer; 2012:236–248.
7. Ferragina P, Manzini G: **Opportunistic data structures with applications.** In *41st Annual Symposium on Foundations of Computer Science*; 2000:390–398.
8. Burrows M, Wheeler DJ: **A block-sorting lossless data compression algorithm.** Tech. Rep. May 10, SRC Research Report 1994.
9. Grossi R, Vitter JS: **Compressed suffix arrays and suffix trees with applications to text indexing and string matching.** In *STOC '00 Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*. New York: ACM; 2000:397–406.
10. Simpson JT, Durbin R: **Efficient de novo assembly of large genomes using compressed data structures.** *Genome Res* 2012, **22**(3):549–556.
11. Ferragina P, Manzini G: **Indexing compressed text.** *J ACM* 2005, **52**(4):552–581.
12. Zerbino DR, Birney E: **Velvet: algorithms for de novo short read assembly using de Bruijn graphs.** *Genome Res* 2008, **18**(5):821–829.
13. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJM, Birol I: **ABySS: a parallel assembler for short read sequence data.** *Genome Res* 2009, **19**(6):1117–1123.
14. Kelley DR, Schatz MC, Salzberg SL: **Quake: quality-aware detection and correction of sequencing errors.** *Genome Biol* 2010, **11**(11):R116.
15. Yang X, Dorman KS, Aluru S: **Reptile: representative tiling for short read error correction.** *Bioinformatics* 2010, **26**(20):2526–2533.
16. Melsted P, Pritchard JK: **Efficient counting of k-mers in DNA sequences using a bloom filter.** *BMC Bioinformatics* 2011, **12:**333.
17. Marçais G, Kingsford C: **A fast, lock-free approach for efficient parallel counting of occurrences of k-mers.** *Bioinformatics* 2011, **27**(6):764–770.
18. Liu Y, Schröder J, Schmidt B: **Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data.** *Bioinformatics* 2013, **29**(3):308–315.
19. Nong G, Zhang S, Chan WH: **Linear suffix array construction by almost pure induced-sorting.** In *2009 Data Compression Conference*: IEEE; 2009:193–202.
20. Hon WK, Sadakane K, Sung WK: **Breaking a time-and-space barrier in constructing full-text indices.** *SIAM J Comput* 2009, **38**(6):2162–2178.
21. Okanohara D, Sadakane K: **A linear-time burrows-wheeler transform using induced sorting.** *Lecture Notes Comput Sci: String Process Inf Retrieval* 2009, **5721:**90–101.

doi:10.1186/1471-2105-14-313
Cite this article as: Rødland: Compact representation of *k*-mer de Bruijn graphs for genome read assembly. *BMC Bioinformatics* 2013 **14**:313.

Submit your next manuscript to BioMed Central and take full advantage of:

- **Convenient online submission**
- **Thorough peer review**
- **No space constraints or color figure charges**
- **Immediate publication on acceptance**
- **Inclusion in PubMed, CAS, Scopus and Google Scholar**
- **Research which is freely available for redistribution**

Submit your manuscript at
www.biomedcentral.com/submit