DSK: k-mer counting with very low memory usage
Guillaume Rizk, Dominique Lavenier, Rayan Chikhi

To cite this version:

Guillaume Rizk, Dominique Lavenier, Rayan Chikhi. DSK: k-mer counting with very low memory usage. Bioinformatics, Oxford University Press (OUP), 2013, 29 (5), pp.652-653. 10.1093/bioinformatics/btt020. hal-00778473

HAL Id: hal-00778473
https://hal.archives-ouvertes.fr/hal-00778473
Submitted on 20 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
DSK: \(k\)-mer counting with very low memory usage

Guillaume Rizk\(^1\), Dominique Lavenier\(^2\) and Rayan Chikhi\(^2\)

\(^1\)Algorizk, 75013 Paris, France
\(^2\)ENS Cachan Brittany / IRISA, Campus de Beaulieu, 35700 Rennes, France

Received on XXXXX; revised on XXXXX; accepted on XXXXX

ABSTRACT

Summary: Counting all the \(k\)-mers (substrings of length \(k\)) in DNA/RNA sequencing reads is the preliminary step of many bioinformatics applications. However, state of the art \(k\)-mer counting methods require that a large data structure resides in memory. Such structure typically grows with the number of distinct \(k\)-mers to count.

We present a new streaming algorithm for \(k\)-mer counting, called DSK (disk streaming of \(k\)-mers), which only requires a fixed, user-defined amount of memory and disk space. This approach realizes a memory, time and disk trade-off. The multi-set of all \(k\)-mers present in the reads is partitioned and partitions are saved to disk. Then, each partition is separately loaded in memory in a temporary hash table. The \(k\)-mer counts are returned by traversing each hash table. Low-abundance \(k\)-mers are optionally filtered.

DSK is the first approach that is able to count all the \(27\)-mers of a human genome dataset using only 4.0 GB of memory and moderate disk space (160 GB), in 17.9 hours.

Availability: http://minia.genouest.org/dsk
Contact: rayan.chikhi@ens-cachan.org

1 INTRODUCTION

Determining the abundance of each distinct \(k\)-mer in a set of sequencing reads is a conceptually simple yet fundamental task. It is used in many bioinformatics applications related to sequencing, e.g. genome and transcriptome assembly, variants detection and read error correction. For de novo assembly, one is often interested in counting \(k\)-mers to discard those with low abundance, which likely stem from sequencing errors.

State of the art methods for \(k\)-mer counting rely on hash tables [1] and/or Bloom filters [2]. These structures need to reside in memory for random access. Sequencing errors induce erroneous \(k\)-mers, in a volume typically greater or comparable to that of correct \(k\)-mers. Hence, counting \(k\)-mers for a human dataset with either a hash table or a Bloom filter is a task that requires tens of gigabytes of memory [1, 2].

In the Methods section, we describe a fixed-memory and fixed-disk space streaming algorithm, DSK, and its worst-case complexity is analyzed in function of the desired memory and disk usage. In the Results section, DSK is used to count all the 27-mers of a whole-genome human dataset (Illumina reads). Furthermore, the trade-off between memory and disk space is analyzed on two smaller datasets. Finally, we discuss some advantages of DSK over related methods, and show a situation where a parallel implementation significantly improved the running time.

2 METHODS

Algorithm 1 The DSK algorithm

\[
\begin{align*}
1: & \quad \text{Input: The set } S \text{ of sequences, } k\text{-mer length } k, \text{ target memory usage } M (\text{bits}), \text{ target disk space } D (\text{bits}), \text{ hash function } h(\cdot) \\
2: & \quad v \leftarrow \sum_{s \in S} |s| - k + 1 \quad \{\text{Number of } k\text{-mers}\} \\
3: & \quad r_i \leftarrow \lfloor v \cdot 2^{\log_2(D/k)} \rfloor / D \quad \{\text{Number of iterations}\} \\
4: & \quad n_p \leftarrow \lfloor v/2^{\log_2(D/k)} + 32 \rfloor / 0.7 \quad \{\text{Number of partitions}\} \\
5: & \quad \text{for each iteration } i = 0..n_p \text{ do} \\
6: & \quad \quad \text{Initialize a set of empty lists } \{d_0, \ldots, d_{n_p}\} \text{ stored on disk} \\
7: & \quad \quad \text{for each sequence } s \in S \text{ do} \\
8: & \quad \quad \quad \text{for each } k\text{-mer } m \text{ in } s \text{ do} \\
9: & \quad \quad \quad \quad \text{if } \{h(m) \mod n_i = i\ \text{then} \\
10: & \quad \quad \quad \quad \quad j \leftarrow h(m) / n_i \mod n_p \\
11: & \quad \quad \quad \quad \text{Write } m \text{ to disk in } d_j \\
12: & \quad \quad \text{for each index } j = 0..n_p \text{ do} \\
13: & \quad \quad \quad \text{Initialize a hash table } T \text{ with } M \text{ bits of memory} \\
14: & \quad \quad \quad \text{for each } k\text{-mer } m \text{ in } d_j \text{ do} \\
15: & \quad \quad \quad \quad T[m] \leftarrow T[m] + 1, \quad \text{if } m \text{ is present in } T \\
16: & \quad \quad \quad \quad 1, \quad \text{otherwise} \\
17: & \quad \quad \text{output } \{m, T[m]\} \text{ for each } m \in T \\
18: & \quad \text{Delete } T \\
\end{align*}
\]

function \(h(\cdot)\) maps a \(k\)-mer to a numeric value in \([0; H]\), where \(H\) is a large integer (typically \(2^{64}\)). In the following analysis, we make a simplifying assumption. Let \(d\) be the total number of distinct \(k\)-mers in the input, we assume that the number of distinct \(k\)-mers having a given hash value \(x \in [0; H]\) is at most \([d/H]\). In other words, the set of distinct \(k\)-mer values can be uniformly partitioned by this hash function. Each \(k\)-mer is encoded using the classical 2 bits representation in the smallest available integer type, i.e. using \(2^\lceil\log_2(D/k)\rceil\) bits. The abundance of each \(k\)-mer is stored as a 32 bits integer. For convenience, let \(b = 2^\lceil\log_2(D/k)\rceil\). Each \(k\)-mer \(m\) present in \(S\) is examined \(n_i = \lfloor v/b \rfloor \) times (once per iteration), and written to disk only once, at the \(h(m) \mod n_i\)-th iteration. Using the uniform repartition hypothesis, a multi-set of \(v/n \leq [D/b] \) \(k\)-mers are written to disk at each iteration. Since each \(k\)-mer is encoded using \(b\) bits, the maximal disk usage of the algorithm is \(D\) bits.

The maximal memory usage of the algorithm is \(M\) bits, since Steps 7-11 require no memory, and Steps 12-17 load a single partition in \(T\) which requires exactly \(M\) bits. With an open-addressing mechanism, each distinct \(k\)-mer occupies exactly \((b + 32)\) bits in \(T\). To prove that the algorithm terminates, it suffices to show that \(T\) never overflows, i.e. that strictly
less than $M/(b+32)$ distinct $k$-mers are inserted in $T$. At each iteration, $(v/n_t)$ $k$-mers are split into $n_t$ partitions. Each partition contains at most $v/(n_t p) \leq \lceil 0.7M/(b+32) \rceil$ $k$-mers. In the worst case, all these $k$-mers are distinct, thus the load factor is upper-bounded by $0.7$ (a classical threshold above which hash table performance degrades).

The time complexity of Steps 7-11 (including the iteration loop) is $O(v^2 b/D)$. The algorithm creates $(v/n_t) p$ temporary hash tables, inserting at most $\lceil 0.7M/(b+32) \rceil$ elements in each. Hash tables accesses and insertions (Step 15) are done in constant expected time with open-addressing, as long as the load factor is strictly below $1$ (which was proved above). Hence, the expected time complexity of Steps 12-17 (including the iteration loop) is $O(v)$. Thus, Algorithm 1 runs in expected time $O(v^2 b/D)$. The algorithm runs in expected linear time with respect to $v$ when $D = \omega(v)$, e.g. setting $D$ equal to the reads file size. In practice, the simplifying assumption on the uniform partition of the hash function $h$ does not hold exactly. Some partitions contain a slightly larger number of distinct $k$-mers than $v/\lceil H \rceil$. Hence, the actual disk usage of the algorithm is slightly above $D$, and the load factor of $T$ could, in theory, be above $0.7$ (due to high $k$-mer redundancy, this is not the case in practice).

### 3 RESULTS

In Table 1, we compared the execution time and memory usage of DSK with Jellyfish (version 1.1.5) and BFCounter (version 0.2) on a human genome Illumina dataset. The target disk usage of DSK was set to 160 GB, equal to the size of the reads file. Since the algorithm relies heavily on I/O to the disk, we also tested DSK with a solid-state drive (DSK-SSD). The reads file was placed on a standard hard disk drive, and partitions of redundant kmers were written on a 256 GB SSD. In this configuration, we noticed the algorithm is no longer limited by disk I/O and could benefit from multi-threading. The two for loops lines 7 and 12 were parallelized using openMP (4 threads).

DSK-SSD ran in 3.5 hours using $4 \times 1$ GB of memory. Although this experiment required specific hardware, it is worth noting that the running time of DSK can be greatly reduced with a SSD and multi-core parallelism.

To further assess the trade-off between time, memory and disk usage, we executed DSK (using a standard hard drive) on two smaller $E. coli$ and $D. ananassae$ datasets, with various target memory and disk usage parameters. For the executions with 100 MB and 1 GB memory usage, the running time of DSK on both datasets decreases as the target disk space increases. This is a consequence of the decreasing number of iterations $n_t$. The running times reach a plateau at roughly the reads file size (where $n_t = 1$).

<table>
<thead>
<tr>
<th>Program</th>
<th>Time (hours)</th>
<th>Memory (GB)</th>
<th>Disk (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSK</td>
<td>17.9</td>
<td>4</td>
<td>160</td>
</tr>
<tr>
<td>DSK-SSD*</td>
<td>3.5</td>
<td>4</td>
<td>240</td>
</tr>
<tr>
<td>BFCounter</td>
<td>41.2</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>Jellyfish</td>
<td>3.5</td>
<td>70</td>
<td>211</td>
</tr>
</tbody>
</table>

The dataset used is the NA18507 human genome (SRX016231), unfiltered, consisting of 1.4 billion of reads of average length 100 bp (160 GB file size). Jellyfish used 8 threads, DSK-SSD used 4 threads, DSK and BFCounter are single-threaded. The disk column indicates the temporary amount of disk space used by each method. * Executed on a desktop computer equipped with two hard drives including a SSD.

The execution time generally appears to be unaffected by the target memory usage. However, at the smallest tested memory usage (10 MB), the execution time on both datasets is slightly higher, possibly due to consecutive disk writes to a large number of partitions. Note that in practice, the memory usage of DSK cannot be arbitrarily low: it is limited by the number of files that can be simultaneously opened on the system (the partitions $\{d_0, ..., d_{n_p}\}$ are all opened simultaneously). In the Drosophila dataset, DSK failed to run with 10 MB of memory and 6 GB disk space for this reason.

### 4 DISCUSSION

Compared to other methods, DSK has three strong points:

- **Low memory usage**: Only an arbitrarily small subset of $k$-mers is loaded in memory at any time. In contrast, BFCounter stores all the $k$-mers with count $\geq 2$ in a hash table. In principle, Jellyfish can use arbitrarily small hash tables, however storing the intermediate results requires a prohibitive amount of disk ($\geq 1$ TB for human genome reads using a hash table of size 5 GB).
- **Parameters are automatically inferred**: the only mandatory argument is the $k$-mer length. Optionally, target memory and disk usages can be specified. Jellyfish and BFCounter require the user to specify respectively a hash table size and an upper-bound on the number of distinct $k$-mers.
- **Supports arbitrarily large values of $k$**: as opposed to up to 32 for Jellyfish (unbounded for BFCounter).

### ACKNOWLEDGMENT

**Funding**: ANR MAPPI, ANR-10-COSI-0004

### REFERENCES
