RESEARCH

Dashing: Fast and Accurate Genomic Distances with HyperLogLog

Daniel N Baker* and Ben Langmead

*Correspondence: dnb@cs.jhu.edu
Department of Computer Science, Johns Hopkins University, 3400 N Charles St, 21218 Baltimore, USA
Full list of author information is available at the end of the article

Abstract

Dashing is a fast and accurate software tool for estimating similarities of genomes or sequencing datasets. It uses the HyperLogLog sketch together with cardinality estimation methods that specialize in set unions and intersections. Dashing sketches genomes more rapidly than previous MinHash-based methods while providing greater accuracy across a wide range of input sizes and sketch sizes. It can sketch and calculate pairwise distances for over 87K genomes in under 6 minutes. Dashing is open source and available at https://github.com/dnbaker/dashing.

Keywords: sketch data structures; hyperloglog; metagenomics; alignment; sequencing; genomic distance

Background

Since the release of the seminal Mash tool [1], data sketches such as MinHash have become instrumental in comparative genomics. They are used to cluster genomes from large databases [1], search in databases for datasets with certain sequence content [2], accelerate the overlapping step in genome assembly tools [3, 4], map sequencing reads [5], and find similarity thresholds characterizing species-level distinctions [6].

Mash's advantages come from its use of the MinHash sketch, first developed for finding similar web pages among vast numbers of candidates [7]. MinHash can summarize a large genomic sequence collection as a small set of constituent k-mers, in turn stored as a list of integers. The summary is much smaller than the original data but can be used to estimate relevant set cardinalities such as the size of the union or the intersection between the k-mer contents of two genomes. From these set cardinalities one can obtain a Jaccard-coefficient (J) or a "Mash distance," which is a proxy for Average Nucleotide Identity (ANI) [1]. These make it possible to cluster sequences and otherwise solve massive genomic nearest-neighbor problems.

MinHash is related to other computational ideas gaining traction in bioinformatics. Minimizers, which can be thought of as a special case of MinHash, are widely used in metagenomics classification [8] and alignment and assembly [9]. More generally, MinHash can be seen as a kind of Locality-Sensitive Hashing (LSH), which involves hash functions designed to map similar inputs the same value. LSH has also been used in bioinformatics, including in homology search [10] and metagenomics classification [11]. Here we focus on MinHash and other methods that are geared toward cardinality estimation, which (via the Jaccard coefficient) can measure sequence distance or similarity.

Spurred by MinHash's utility, other groups have proposed alternatives using new ideas from search and data mining. BinDash [12] uses a b-bit one-permutation rolling MinHash to achieve greater accuracy and speed compared to Mash at a smaller memory footprint. More theoretical improvements are proposed in the HyperMinHash [13] and SuperMinHash [14] studies.

Some studies have also pointed out shortcomings of Mash and MinHash. Koslicki and Zabeti argue that MinHash cardinality estimates suffer when the sets are very different sizes [15]. This is not an uncommon scenario, e.g. when finding the distance between two genomes of very different lengths or when finding the similarity between a short sequence (say, a bacterial genome) and a large collection (say, deep-coverage metagenomics datasets).

Here we propose to use the HyperLogLog (HLL) sketch [16] as an alternative to MinHash that exhibits excellent accuracy and speed across a range of scenarios, including when the input sets are very different sizes. HLL has been applied in other areas of bioinformatics, e.g. to count the number of distinct k-mers in a genome or data collection [17, 18, 19]. Our approach additionally builds on recent theoretical improvements in cardinality estimates for set unions and intersections [20], the crucial components needed to estimate J and other similarity measures. These improvements, together with the HLL's inherent advantages, yield greater accuracy and speed compared to both Mash and BinDash in many situations.

We implemented the HLL sketch in the Dashing software tool (`https://github.com/dnbaker/dashing`), which is free and open source under the GPLv3 license. Dashing supports the functions available in similar tools like Mash [1], BinDash [12] and Sourmash [21]. Dashing can build a sketch of an input sequence set (`dashing sketch`), including FASTA files (for assembled genomes) or FASTQ files (for sequencing datasets). Like Mash, Dashing has a sketch-based facility for removing k-mers that likely contain sequencing errors prior to sketching. The `dashing dist` function performs all-pairwise distance comparisons between pairs of datasets in a large collection, e.g. all the complete genomes from the RefSeq database. This is similar to Mash's `mash dist` and `mash triangle` functions. Since Dashing's `sketch` function is extremely fast, Dashing can perform both sketching and all-pairs distance calculations in the same command, saving the user from having to store sketches on disk between steps. Dashing is parallelized and we show that it scales efficiently to 100 threads. Dashing also uses Single Instruction Multiple Data (SIMD or "vector") instructions on modern general-purpose computer processors to exploit the finer-grained parallelism inherent in calculating the HLL estimate.

Results

Here we briefly discuss of the design of the Dashing software tool, then present simulation results demonstrating HLL's accuracy relative to other sketch data structures. We then describe experiments demonstrating Dashing's improved accuracy relative to Mash and BinDash in a range of scenarios. Finally, we discuss Dashing's computational efficiency relative to Mash and BinDash.

Unless otherwise noted, experiments were performed on a Lenovo x3650 M5 system with 4 2.2Ghz Intel E5-2650 CPUs with 12 cores each and 512 GB of DDR4 RAM. Input genomes and sketches were all stored on a SAS-attached Lenovo Storage E1000 disk array with 12 8TB 7,200-RPM disks combined using RAID5.

All experiments were conducted using scripts available in the `dashing-experiments` repository at https://github.com/langmead-lab/dashing-experiments.

Design

Dashing is a software tool that uses the HyperLogLog (HLL) sketch to solve genomic distance problems. Similarly to tools like Mash, Dashing takes one or more sequence collections as input. These could be assembled genomes in FASTA format or sequencing datasets in FASTQ format. It then builds an HLL sketch for each input collection based on its k-mer content. The sketch can be written to disk or simply forwarded to the next phase, which performs a distance comparison between one or more pairs of sketches. In one usage scenario, the user provides two FASTA files and Dashing sketches them and emits a set of similarity estimates, including an estimate of the Jaccard-coefficient J. This is similar to Mash's `mash dist` mode. In another scenario, the user provides a large collection of FASTA files (e.g. all the complete bacterial genome assemblies in Refseq) and Dashing sketches them and emits estimates for all pairwise distances, similarly to `mash triangle`.

Dashing is written in C++. It can use many threads in parallel, with both the sketching and distance phases readily scaling to 100 threads. It also makes effective use of data-parallel SIMD instructions, including the recent AVX512-BW extensions that have been effective at accelerating other bioinformatics software [22]. Dashing also provides Python bindings that enable other developers to easily make use of our HLL implementation.

Sketch accuracy

To assess HLL's accuracy, we measured Jaccard-coefficient estimation error across a range of set and sketch sizes. We implemented and compared three structures in Dashing v0.1.1: HLL [16], Bloom filters [23] and MinHash [7]. For HLL, we used Ertl's Maximum Likelihood Estimator (MLE) method for estimating set cardinalities [20], though we explore alternate methods in later sections. For the Bloom filter, we used both a naive (collision-agnostic) and a collision-aware method [24] for estimating set cardinalities. For the MinHash structure, we used a k-bottom sketch with a single hash function, following Mash's strategy [1]. In all cases, we used Thomas Wang's 64-bit reversible hash function [25]. In all cases, the tools used canonicalized k-mers, so that a k-mer and its reverse complement are treated as equal when sketching.

While it can improve accuracy to use more than one hash function in a Bloom filter, we used just one hash function here. This is a reasonable choice in the absence of any foreknowledge of the set cardinality. Additionally, because our aim is high accuracy in reduced space, the optimal number of hash functions for a Bloom Filter $(\frac{m}{n}\ln(2)) \leq 1$ for any experiments with cardinality greater than sketch size.

We performed sets of experiments where we first fixed the two input sets — their sizes and the true Jaccard-coefficient between them — as well as the size of the data structures. Though the structures differ in character, with the HLL storing an array of narrow integers, the MinHash storing an array of wider integers and the Bloom filter storing an array of individual bits, we can nonetheless parameterize them to use exactly the same amount of storage. (A practical detail is that the

width of the integers stored in the MinHash sketch depend on the selected k-mer length; we account for this so as to maintain equal size across tools.) We populated each structure using its natural insert operation; for the HLL this involves hashing the item and using the resulting value to identify the target register and possibly update it according to the leading zero count of the remainder bits. This is discussed in detail in Methods. For the Bloom filter, inserting involves hashing the item and setting the corresponding array bits to 1. For the bottom-k MinHash, inserting involves hashing the item and updating the sketch if the hash is less than the current greatest sketch element.

We populated the input sets with random numbers, thereby simulating an ideal hash function with uniformly distributed outputs. Sets were constructed in order to achieve a target Jaccard coefficients for a set of predetermined Js ranging from 0.00022 to 0.818. A range of set-size pairs were evaluated ranging from equal-size sets to sets with sizes differing by a factor of 2^{12}. In total, 36 set-size / J pairs were evaluated, with full results presented in Supplementary Table 1. Note that set sizes and Jaccard-coefficient are dependent; if set A has cardinality c times greater than set B, $J(A, B) \leq \frac{1}{c}$.

Figure 1 shows results for two values of the true Jaccard coefficient, 0.0465 and 0.111, for five pairs of unequal-cardinality input sets, and for various sketch sizes. The vertical axis shows the difference in absolute error between the alternate structure and the HLL. Points appearing below $y = 0$ indicate where the HLL sketch had higher absolute error than the alternative. The three rows show successively tighter zoom-ins on the y axis.

We first observed that HLL exhibits lower absolute error in most circumstances, especially for smaller sketches and larger sets. For the two Js tested here, HLL had lower error compared to MinMash as seen most clearly in the second row of Figure 1. The Bloom-filter-based methods, especially the collision aware method (Bloom+) achieved slightly lower error in some scenarios. Bloom+ had lower error (by < 0.0025) for sketches of size 2^{13} bytes and higher for the smallest sets. For larger sets, the point where Bloom+ began to have lower error than HLL moved rightward. The naive Bloom method also eventually outperformed HLL, though only at the largest sketch size and smallest input sets. That the Bloom filters outperform at large sketch sizes is not surprising; as the number of filter bits increases far past the set cardinality, collisions become rare and the method converges on error-free linear counting.

Though Figure 1 plots error differences between pairs of structures, full results, including absolute and squared errors, can be found in Supplementary Table 1. There we observed that even in the most adverse scenarios (small data structures and very different set sizes) HLL's absolute error never exceeded 3% (compared to 8% for MinHash). Overall, the results recommend HLL has an accurate and memory-economical sketch requiring no major assumptions about input set sizes.

Accuracy for complete genomes

Encouraged by HLL's accuracy, we measured the accuracy of Dashing v0.1.1's HLL-based Jaccard-coefficient estimates versus Mash v2.1 [1] and BinDash v0.2.1 [12].

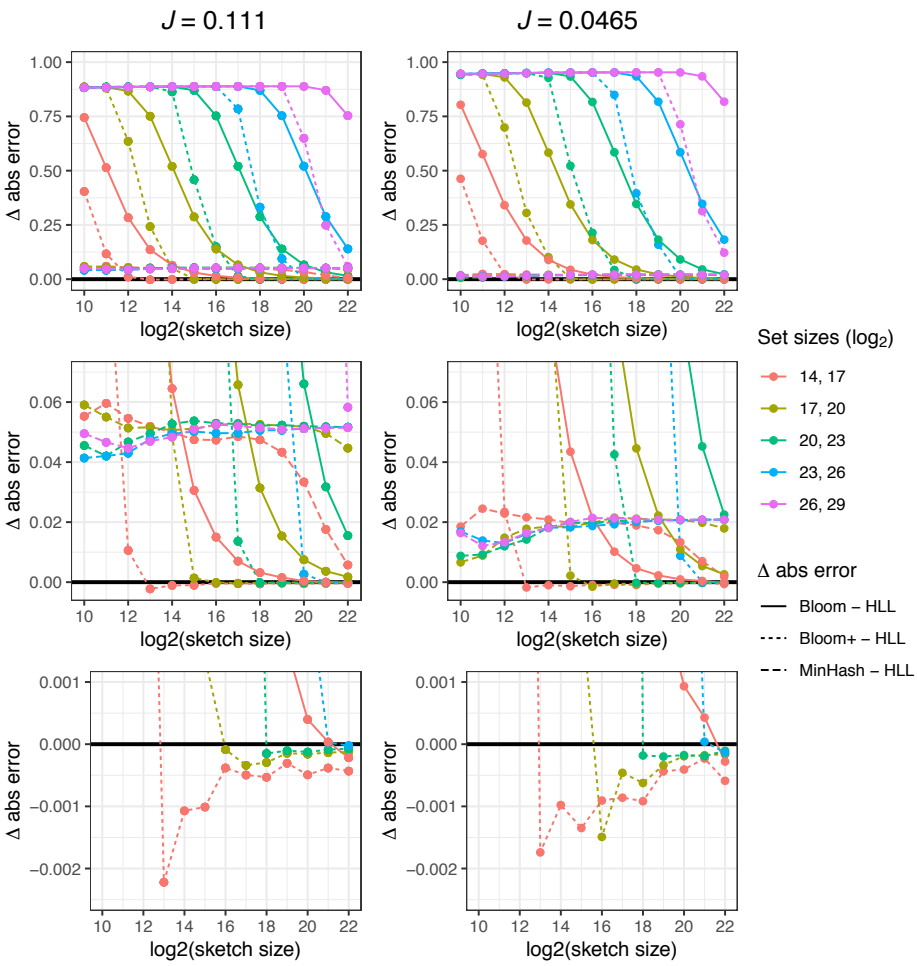

Figure 1 Jaccard-coefficient estimation error for different-size sets using HLL, Bloom filters and MinHash. Left column shows a set of experiments with various set sizes but with the true Jaccard-coefficient fixed at 0.111. Right shows the same but for a true coefficient of 0.0465. All pairs of input sets differ in size by a factor of $2^3 = 8$. The second and third rows zoom further in with respect to the y-axis so as to highlight the relative performance of MinHash (2nd row) and the Bloom+ filter (3rd row).

For HLL, we repeated the experiment for three HLL cardinality estimation methods: Flajolet's canonical method using harmonic mean [16], and two maximum-likelihood-based methods (MLE and JMLE) proposed by Ertl [20]. We selected 400 pairs of bacterial genomes from RefSeq [26] covering a range of Jaccard-coefficient values. To select the pairs, we first used `dashing dist` with $s = 16$, $k = 31$ and the MLE estimation method on the full set of 87,113 complete RefSeq assemblies (latest versions). We then selected a subset such that we kept 4 distinct genome pairs per Jaccard-coefficient percentile. In this way, we started out with an even spread of Jaccard-coefficient values, though some unevenness emerges later due to differences between data structures and different selections of k. Of the genomes included in these pairs, the maximum, minimum and mean lengths were 11.7 Mbp, 308 Kbp, and 4.00 Mbp respectively.

We ran the three tools to obtain Jaccard-coefficient estimates for the 400 pairs and plotted the results versus true Jaccard-coefficients, as determined using a full hash-table-based k-mer counter. Results for $k = 16$ and $k = 21$ and for sketches of size 2^{10} and 2^{14} bytes are shown in Figure 2. The horizontal axis is the true Jaccard-coefficient while the vertical is the difference between the tool-estimated Jaccard-coefficient and the true Jaccard-coefficient . We plot a smooth (loess) curve for each tool to make J-dependent biases more evident. For Dashing we used the MLE estimation method. We made a minor change to the Mash software to allow it to output estimated Jaccard coefficient, as it typically emits only Mash distance.

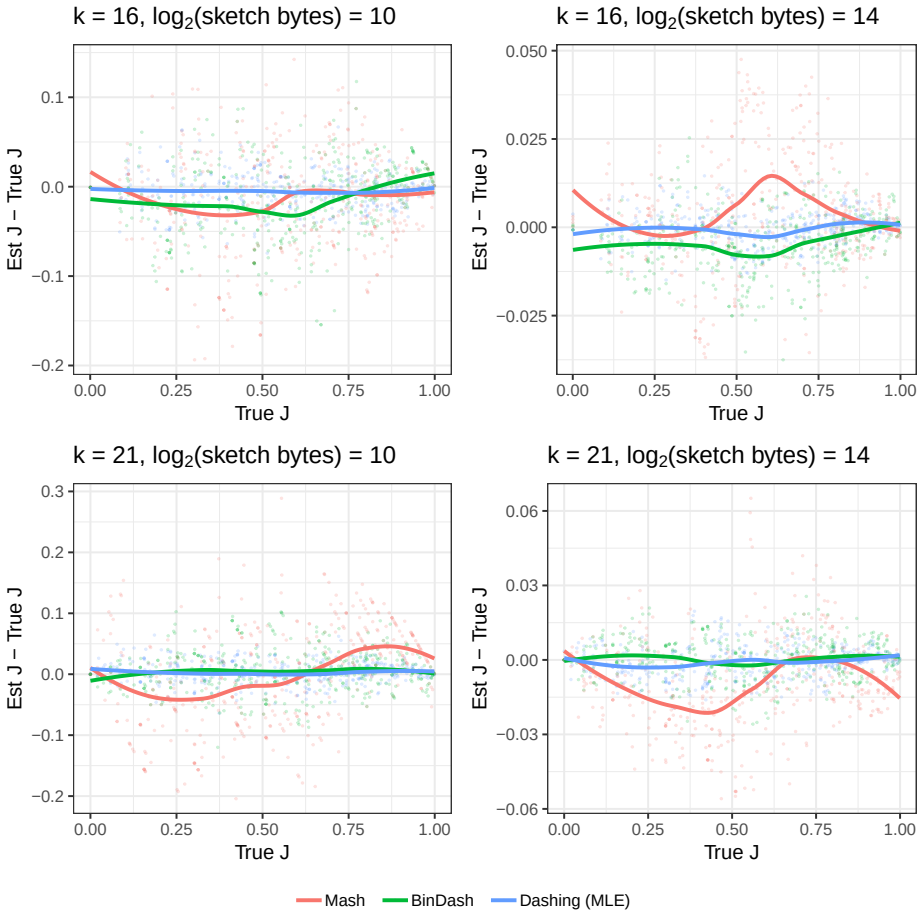

Figure 2 Estimated versus true Jaccard coefficients (Js) for various methods across a range of true J. Each point is one pair from an overall set of 400 pairs of genomes, selected to evenly cover the range of true Js. Smooth loess curves are drawn to highlight systematic bias as a function of J.

Dashing's estimates were consistently near the true J. Mash shows a pattern of bias whereby its estimates are somewhat too low at low Jaccard-coefficients then too high at higher coefficients, causing the smooth curve to bend down and then up as it moves to the right. This is sometimes combined with an overall bias shifting estimates too high (in the case of $k = 16$, sketch size $= 2^{14}$) or low (in the case of $k = 21$, sketch size $= 2^{14}$). BinDash and Dashing exhibit less J-specific bias, but with Dashing staying noticeably closer to $y = 0$ compared to BinDash.

Table 1 shows mean squared Jaccard-coefficient estimation error (MSE) for a range of sketch sizes and for $k = 31$, and also including the two alternate cardinality estimation methods for Dashing (Original and JMLE). The lowest two MSEs are highlighted in each row. While JMLE consistently has the lowest MSE, we show later that it is also quite time consuming; Dashing's default estimation method is the MLE, which performed similarly to (but somewhat worse than) JMLE. The HLL methods performed better than Mash and BinDash in every J decile except the lowest, where BinDash had the lowest MSE across all sketch sizes. Supplementary Table 2 shows a fuller set of results for $k \in \{16, 21, 31\}$ and sketch sizes of 2^i bytes for $i \in \{10, 11, 12, 13, 14, 15\}$.

Table 1 Jaccard-coefficient estimation accuracy for Mash, BinDash and Dashing for $k = 31$ and a range of sketch sizes. Results are stratified by Jaccard-coefficient decile (first column), and within-decile results are mean squared errors across all genome pairs having Jaccard-coefficient in the decile; the number of such pairs is shown in the n column. The lowest two errors in each row are bolded.

J bin	k	$\log_2(size)$	Mash	BinDash	HLL Original	HLL Ertl-MLE	HLL Ertl-JMLE	n
0.0 – 0.1	31	12	269.0	**71.7**	120.8	116.3	**97.3**	36
		13	155.8	**33.1**	95.6	89.6	**72.7**	36
		14	120.7	**25.4**	42.4	41.2	**31.8**	36
		15	75.9	**14.5**	30.0	27.4	**23.5**	36
0.1 – 0.2	31	12	1,323.5	200.0	179.8	**177.7**	**135.2**	46
		13	655.8	138.6	92.3	**88.6**	**76.2**	46
		14	443.5	85.0	**57.6**	58.4	**47.6**	46
		15	228.7	28.8	17.1	**15.8**	**14.0**	46
0.2 – 0.3	31	12	1,025.7	277.4	118.6	**113.4**	**91.3**	38
		13	631.0	145.7	70.4	**57.8**	**57.8**	38
		14	478.8	57.5	31.6	**30.0**	**27.7**	38
		15	145.5	70.8	21.8	**19.8**	**20.3**	38
0.3 – 0.4	31	12	1,518.9	333.1	90.3	**85.6**	**82.3**	40
		13	532.2	164.8	63.9	**61.1**	**59.5**	40
		14	377.7	77.6	**39.8**	41.4	**36.5**	40
		15	260.5	73.8	**5.6**	5.8	**5.7**	40
0.4 – 0.5	31	12	2,017.8	509.7	130.4	**114.4**	**100.0**	41
		13	773.8	162.5	39.6	**36.2**	**39.2**	41
		14	220.4	102.4	**29.3**	**29.5**	30.7	41
		15	99.2	72.3	15.3	**14.8**	**14.1**	41
0.5 – 0.6	31	12	2,126.7	586.3	**131.5**	132.2	**124.4**	39
		13	997.0	265.4	52.3	**52.1**	**49.1**	39
		14	379.2	138.5	**17.3**	17.6	**16.0**	39
		15	165.4	75.9	13.2	**12.0**	**11.5**	39
0.6 – 0.7	31	12	2,950.5	544.2	85.8	**80.4**	**80.7**	40
		13	899.1	239.4	48.5	**47.9**	**46.0**	40
		14	387.3	96.6	37.4	**36.8**	**34.9**	40
		15	206.0	42.3	18.0	**17.8**	**17.2**	40
0.7 – 0.8	31	12	2,584.8	383.5	99.9	**87.9**	**85.5**	40
		13	405.5	178.1	67.1	**63.0**	**61.6**	40
		14	166.3	99.4	**32.2**	32.9	**32.6**	40
		15	121.3	41.1	**12.2**	**12.4**	12.6	40
0.8 – 0.9	31	12	1,641.6	285.0	69.2	**65.2**	**64.7**	39
		13	145.9	131.0	19.4	**19.0**	**18.9**	39
		14	222.9	87.4	12.0	**10.4**	**10.4**	39
		15	157.8	43.6	6.6	**5.8**	**5.7**	39
0.9 – 1.0	31	12	178.5	110.3	19.0	**14.9**	**14.8**	41
		13	110.9	79.6	10.7	**9.1**	**9.0**	41
		14	70.9	35.9	5.1	**4.3**	**4.3**	41
		15	34.9	7.7	1.6	**1.4**	**1.3**	41

Computational efficiency

To assess computational efficiency and scalability in a realistic context, we used Dashing v0.1.1, Mash v2.1 and BinDash v0.2.1 to sketch and perform all-pairs

distance calculations for 87,113 complete genome assemblies (marked "latest" and "Complete genome" and without "contig" in the name) from RefSeq [26], requiring over 3.7 billion pairwise comparisons. The set included genomes from various taxa, spanning viral, archaeal, bacterial and eukaryotic. Genome lengths varied from 288 bases to 4,502,951,408 bases with mean and median lengths of 9.8Mb and 3.8Mb, respectively. We repeated the experiment for a range of sketch sizes and k-mer lengths. All experiments were performed on a Lenovo x3850 X6 system with 4 2.0Ghz Intel E7-4830 CPUs with 14 cores each and 1 TB of DDR4 RAM. After hyperthreading, the system supports up to 112 simultaneous hardware threads. The system runs CentOS 7.5 Linux, kernel v3.10.0, and is located at the Maryland Advanced Research Computing Center (MARCC).

For Dashing, we used the `dashing sketch` command for the sketching phase and `dashing dist` for the pairwise distance calculation. In a separate experiment, we ran `dashing dist` such that both sketching and all-pairwise distance calculations were performed in a single invocation, without writing the sketches to disk. For Mash, we used `mash sketch` and `mash triangle` for the two stages respectively. Likewise for BinDash we used `bindash sketch` and `bindash dist`. Though Mash and BinDash lack a mode that combines sketching and distance calculations in a single invocation, we simulated this by combining the results from the separate invocations. Specifically, we summed wall clock times, took the maximum of the peak memory footprints, and took a wall-clock-time-weighted average of the CPU utilization measurements. Combined results are labeled "Both."

All tools were configured to use up to 100 simultaneous threads of execution (Dashing: `-p 100`, Mash: `-p 100`, BinDash: `--nthreads=100`). Since the system supports a maximum of 112 simultaneous threads, 100 was chosen to achieve high utilization while avoiding excessive contention. We report average number of CPUs, user time, wall time, system time, and peak memory footprint utilizated in each phase of each tool, as measured with the GNU *time* utility.

For Dashing, we repeated the experiment for each of its three cardinality estimation methods: Flajolet's canonical method ("Original"), Ertl's Maximum Likelihood Estimator ("Ertl-MLE") and Ertl's joint MLE ("Ertl-JMLE").

Results for $k = 31$ are summarized in Figure 3 and Table 2, while Supplementary Table 3 additionally shows results for $k = 21$. We observed that Dashing is the fastest tool in the Sketch phase, running 3.5–3.9 times faster than BinDash and 4.0–4.7 times faster than Mash. Though Dashing achieves the greatest CPU utilization in the Sketch phase, the tools all perform similarly well on this measure, with Mash utilizing about 85 CPUs on average, Bindash about 79 and Dashing about 85–97 (depending on estimation method). Thus, Dashing's speed is explained mostly by per-thread efficiency.

BinDash achieves the lowest memory footprint among the tools in the Sketch phase, requiring 80 MB for the 1-KB sketch and 1.4 GB for the 64-KB sketch. By contrast, Dashing required about 12 GB across all sketch sizes. This is largely because of how Dashing is parallelized; Dashing threads simultaneously work on separate sequence collections, each filling a buffer of size sufficient to hold the largest sequence yet parsed by that thread. Mash had the highest memory footprint, ranging from 16–37 GB.

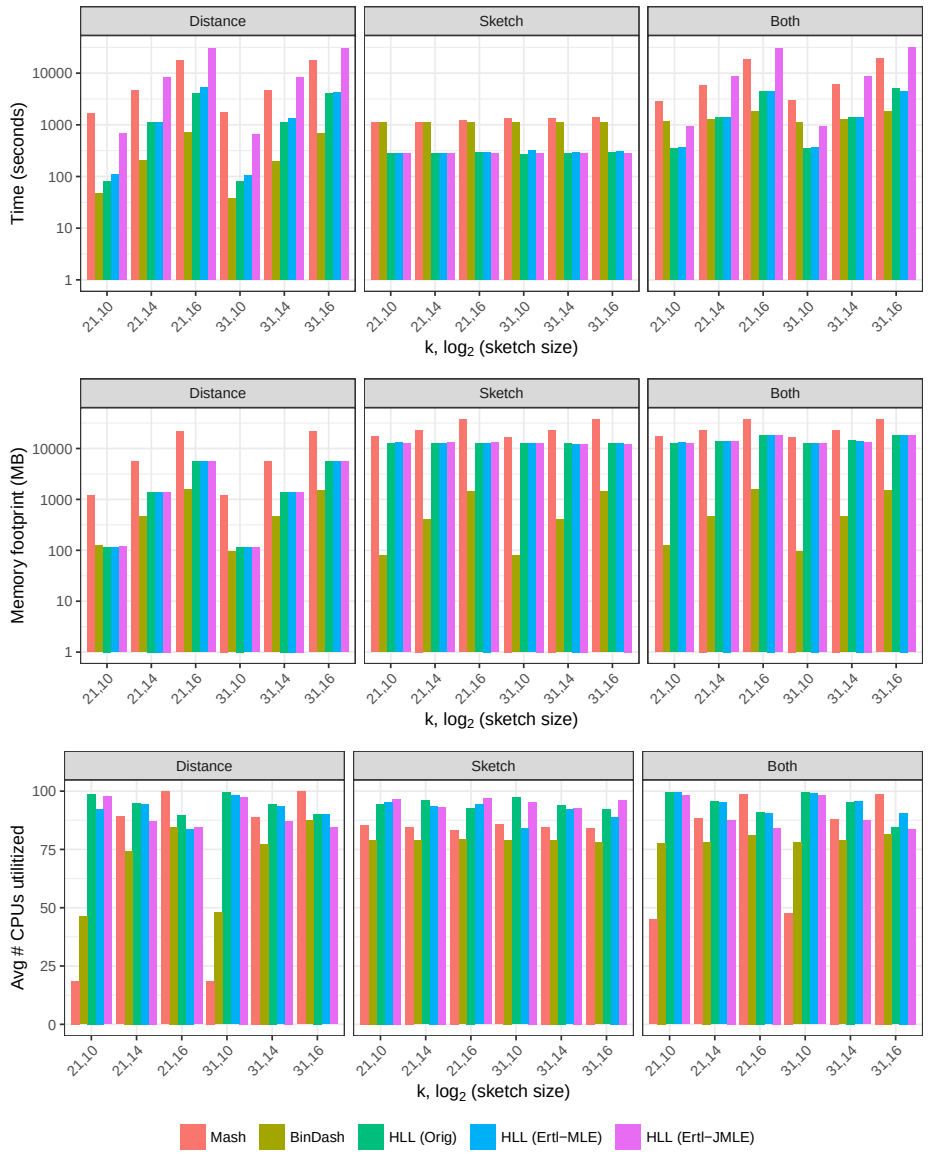

Figure 3 Computational efficiency of Mash, BinDash and Dashing. Results for $k = 21$, $k = 31$ and sketches of size 2^{10} (1KB) 2^{14} (16KB) and 2^{16} (64KB). "Both" results obtained either by using a combined Sketch+Distance mode (for Dashing) or by combining results from the separate Sketch and Distance invocations (for Mash and BinDash). Dashing was assessed using three estimation methods: Flajolet's method using the harmonic mean ("Orig") and Ertl's MLE and JMLE methods.

In the Distance phase, we noted that the estimation method had a major influence on Dashing's speed, with JMLE performing about 6–7.5 times slower than MLE. This is because the JMLE perfors significantly more calculations, as described in Methods. This result, together with the relatively small accuracy difference noted earlier, led us to chose the Ertl-MLE method as Dashing's default.

BinDash was the fastest tool in the Distance Phase, running up to 2–7 times faster than Dashing's MLE mode, with the largest speed gap observed at the largest

Table 2 Comparison of computational efficiency of Mash, BinDash and Dashing. Results for $k = 31$ shown here, with $k = 21$ results shown in Supplementary Table 3. "Both" results obtained either by using a combined Sketch+Distance mode (for Dashing) or by combining results from the separate Sketch and Distance invocations (for Mash and BinDash). Dashing was assessed using three estimation methods: Flajolet's method using the harmonic mean ("Original") and Ertl's MLE and JMLE methods.

Phase	Measure	k	$\log 2(size)$	Mash	BinDash	Dashing Original	Dashing Ertl-MLE	Dashing Ertl-JMLE
Sketch	Wall clock (s)	31	10	1,313	1,098	**271**	316	277
			14	1,349	1,097	**283**	287	287
			16	1,425	1,127	295	305	**283**
	Avg # CPUs	31	10	85.7	78.99	**97.33**	84.0	95.32
			14	84.59	79.03	**94.09**	92.15	92.87
			16	84.23	78.04	92.17	89.03	**96.32**
	Peak mem (MB)	31	10	16,753	**79**	12,436	12,378	12,835
			14	23,058	**399**	12,412	12,269	12,291
			16	37,393	**1,428**	12,728	12,702	12,269
Distance	Wall clock (s)	31	10	1,714	**37**	80	103	661
			14	4,704	**196**	1,118	1,315	8,338
			16	17,779	**687**	4,119	4,171	30,360
	Avg # CPUs	31	10	18.52	47.97	**99.36**	98.08	97.48
			14	88.97	77.45	**94.39**	93.48	87.26
			16	**99.79**	87.41	90.1	89.92	84.42
	Peak mem (MB)	31	10	1,217	**95**	116	115	115
			14	5,473	**461**	1,392	1,391	1,391
			16	21,394	**1,492**	5,477	5,477	5,477
Both	Wall clock (s)	31	10	3,027	1,135	**348**	370	926
			14	6,053	**1,293**	1,386	1,398	8,555
			16	19,204	**1,814**	4,991	4,427	31,920
	Avg # CPUs	31	10	47.66	77.97	**99.48**	99.02	98.07
			14	87.99	78.79	95.45	**95.62**	87.41
			16	**98.64**	81.59	84.44	90.49	83.85
	Peak mem (MB)	31	10	16,753	**95**	12,604	12,763	12,591
			14	23,058	**461**	14,113	14,043	13,458
			16	37,393	**1,492**	17,998	18,394	18,326

sketch size. But Dashing is 3–16 times faster than Mash, with the largest speed gap observed at the smallest sketch size.

When we compared tools based on combined performance across both phases, BinDash again had the lowest memory footprint (always below 1.5GB), with Dashing's footprint in the 12–18 GB range and Mash's in the 16–38GB range. Regarding speed, we found that Dashing was the fastest at the 1-KB sketch size, slightly (8%) slower than BinDash at the 16-KB sketch size, and substantially slower at the largest (64-KB) size. Mash was the slowest of the tools in all cases. Considering Dashing's accuracy at small sketch sizes, the fact that Dashing is fastest for those sizes fits well with the scenarios we expect users to encounter.

Discussion

Genomics methods increasingly use MinHash and other locality-sensitive hashing approaches as their computational engines. We showed that the HyperLogLog sketch, combined with recent advances in cardinality estimation, offers a superior combination of efficiency and accuracy compared to MinHash. This is true even for small sketches and for the challenging case where the input sets have very different sizes. While HLL has been used in bioinformatics tools before [17, 18, 19], this is the first application to the problem of estimating genomic distances, the first implementation of the highly accurate MLE and Joint-MLE estimators [20], and the first comprehensive comparison to MinHash and similar methods.

We implemented HLL-based sketching and distance calculations in the Dashing software tool. Dashing can sketch and calculate pairwise distances for over 87K Refseq [26] genomes in under 6 minutes using 100 threads. The speed advantage

is clearest in the sketching step. Notably, re-sketching from scratch is not much slower than loading pre-made sketches from disk. Thus, Dashing users can forgo the typical practice of saving sketches to disk between steps. Dashing's accuracy at smaller sketch sizes also allows us to set the default sketch size (1KB) substantially below that of other tools (Mash: either 4KB or 8KB depending on k-mer length, BinDash: 3.5–4KB).

HLLs also come with drawbacks relative to other sketches. As shown in Figure 3 and Table 2, distance calculations with Dashing is substantially slower than the MinHash-based method implemented in BinDash. This is expected; the MinHash approach for finding the distance between two sketches involves simple counting and merging of sorted lists of integers. By contrast, a distance calculation between two HLL sketches involves more computation, including exponentiations, divisions, harmonic means, and — for the MLE-based methods — iterative procedures for finding roots of functions. So the increased accuracy comes at a computational cost. This was clearest for the most accurate method we tested, Ertl's joint MLE [20], which was the slowest (even compared to MinHash) for sketches sized 16K and greater. It will be important to continue to refine and accelerate the cardinality-estimation algorithms at the core of `dashing dist`.

HLL lacks another advantage of MinHash; when MinHash is used in conjunction with a reversible hash function, it can be used not only to calculate the relevant set cardinalities but also to report the k-mers common between the sets. This can provide crucial hints when the eventual goal is to map a read to (or near) its point of origin with respect to the reference, as is the goal for tools like MashMap [5].

Multiple efforts have considered how to extend MinHash to include information about element multiplicities, essentially allowing it to sketch a multiset rather than a set. This can improve accuracy of genomic distance measurements, especially in the presence of repetitive DNA. Finch [27] works by capturing more sketch items than strictly needed for the k-bottom sketch, then tallying them into a multiset. More theoretical studies have proposed ways to store multiplicities, including BagMinHash [28], and SuperMinHash [14]. In the future it will be important to seek similar multiplicity-preserving extensions — and related extensions like *tf-idf* weighting [3, 29] — for HLL as well.

As we consider how HLL can be extended to improve accuracy and handle multiplicities, an asset is that our current design uses only 6 out of the 8 bits that make up each HLL register. (The LZC of our hash cannot exceed 63 and therefore fits in 6 bits.) Thus, 25% of the structure is waiting for an appropriate use. One idea would be to use the bits to store a kind of striped, auxiliary Bloom filter. This would add an alternate sketch whose strength lies in estimating low-cardinality sets. Since we showed that a Bloom filter has superior accuracy at sketch sizes that are large enough to simulate linear counting (Figure 1), we could populate the auxiliary filter with a well calibrated subsample of the input items and thereby recover some of the accuracy advantage enjoyed by Bloom filters.

HLL's accuracy even at low sketch sizes also recommends it as a tool for search and indexing. It can be seen as performing a similar function as the Sequence Bloom Tree [30]. Additionally, because any items which can be hashed can be inserted in a HyperLogLog, dashing could be generalized or extended to other applications, such as comparing text documents by their n-grams, or images by extracted features.

Methods

HyperLogLog

The HyperLogLog sketch builds on prior work on approximate counting in $\mathcal{O}(\log_2 \log_2(n))$ space. Originally proposed by Morris [31] and analyzed by Flajolet [32], this method estimates a count by probabilistically incrementing a counter for each item with exponentially decaying probability. The probability is typically halved after each increment, in which case we are approximately tracking the $\log_2$ of the true count. While the estimator is unbiased, it has high variance. The hope is that needing only $\log_2 \log_2(n)$ bits to store a summary — compared to the $\log_2(n)$ needed for a simple minimum hash — will allow us to store more summaries total and, with averaging, will yield a better overall estimate.

The HLL combines many counters of this kind into one sketch, leveraging "stochastic averaging" [33]. Given a stream of data items, we partition them according to the most significant bits ("prefix") of their hash values. That is, if o is an input item and h is the hash function, the value $h(o)$ is partitioned so that $h(o) = p \oplus q$ for bit-string prefix p and suffix q. To insert the item, we use p as an offset into an array of 8-bit "registers." We set the register to the maximum of its current value and the leading zero count (LZC) of the suffix q (Figure 4) + 1. Note that the LZC of a bit string x of length q is related to $\log_2(x)$:

$$\mathrm{LZC}(x) = \begin{cases} q, & x = 0 \\ q - 1 - \lfloor \log_2(x) \rfloor & x > 0 \end{cases}$$

Each register ultimately stores a value related to $\min_{q \in Q} \log_2(q)$ where Q is the set of suffixes mapping to the register (Figure 5). We then combine estimates across registers by taking their harmonic mean and applying a correction factor. The harmonic mean dampens the effect of outlier registers. The HLL estimator has a standard error of $\frac{1.03896}{\sqrt{m}}$ [16].

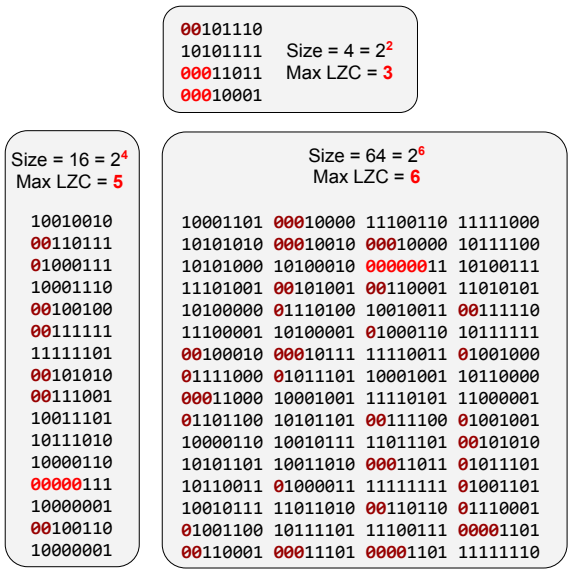

Figure 4 Relationship between maximum leading zero count (Max LZC) and set size for three randomly-generated sets of 8-bit numbers. The Max LZC roughly estimates the $\log_2$ of the set size, though with high variance; here, two of the three estimates are off by 2-fold.

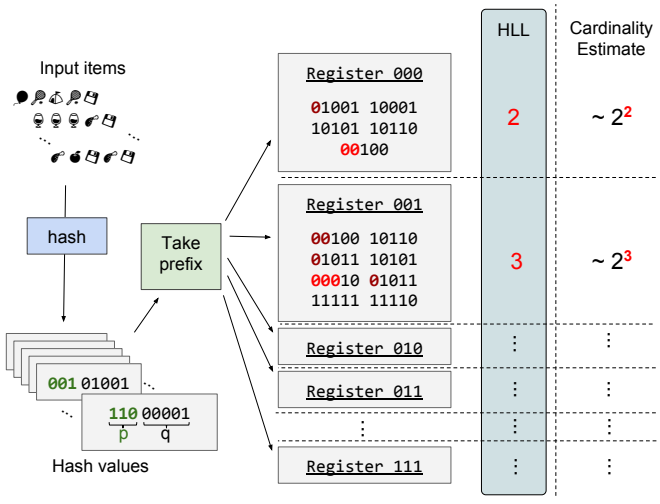

Figure 5 Schematic of HyperLogLog sketch. Input items are hashed and the hash value is divided into prefix p and suffix q. p is used as an index into an array of registers. A register contains the maximum leading zero count among all suffixes q that mapped there. This in turn can be used to estimate the cardinality of each register. Register-level estimates are then averaged (by harmonic mean) and corrected to obtain an overall cardinality estimate. Though hash values shown here are 8 bits, Dashing uses a 64-bit hash. While the figure shows 8 registers, a typical Dashing sketch will have on the order of thousands to millions of registers.

While the HLL is conceptually distinct from MinHash sketches and Bloom filters, it is related to both. As an intuitive example, an HLL with a single register (i.e. using a prefix length of 0) and where the summary statistic is a simple minimum (rather than the minimum $\log_2$) approaches a MinHash. Similarly, a Bloom filter with a single hash function and 2^x bits resembles an HLL with an x-bit hash prefix and registers consisting of a single bit each.

Estimation methods

The original HLL cardinality estimation method [16] combines the estimates in the registers by taking a corrected harmonic mean:

$$E = \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-M_j}}$$

Where α_m is a corrective factor equal to $\frac{1}{2 \ln 2} = 0.72134$ and M_j is the $1 +$ the maximum LZC stored in register j. But the accuracy of this estimator suffers at both low and high extremes of cardinality. This has spurred various refinements starting with the original HLL publication [16], where linear counting is used to improve estimates for low cardinalities and careful treatment of saturated counters improves high-cardinality estimates.

Ertl proposed further refinements [20]. The "improved estimator" uses the assumptions that (a) the hash functions produces uniformly distributed outputs, and (b) register values are independent. This allows it to model register counts as a Poisson random variable. Ertl shows that estimating the Poisson parameter yields an estimate for the cardinality.

Ertl's MLE method again uses the uniformity and Poisson assumptions of the Improved method, but the MLE method proceeds by finding the roots — e.g. using Newton's method or the secant method — of the derivative of the log-likelihood of the Poisson parameter given the register values. Ertl shows that the estimate is lower- and upper-bounded by harmonic means of the per-register estimates (which retrospectively validated use of harmonic mean in the original HyperLogLog estimator). Ertl suggests using the secant method, which allows the root to be found using primarily inexpensive instructions and no derivative calculations. We following this suggestion in Dashing. Ertl also argues that the MLE generally converges in a small number of steps, and we confirm that our implementation converges in ≤ 3 steps in every case we have tested.

Unlike the methods discussed up to here, the Joint MLE method can directly estimate cardinalities of set intersections and differences. It does so without using the inclusion-exclusion principle. The JMLE method again adopts the Poisson model but now the two sketches, A and B, are modeled as a mixture of three components, one with elements unique to A, another with elements unique to B and a third with elements in their intersection $A \cap B$. The method then jointly estimates three Poisson parameters. Inputs to the procedure include tallies for how often registers in A are less than, equal to, or greater than their counterparts in B.

As discussed in Results, the JMLE method as implemented in Dashing is substantially slower than the MLE. This is only partly because of the increased complexity

of the numerical optimization, as there are more optimization problems and each requires roughly twice as many iterations as for MLE. However, our profiling indicates the added time is chiefly spent on tallying the $<, =, >$ relationships between the sketch registers. Further, the tallying cost is linear with the sketch size whereas the optimization costs stay relatively fixed in our experiments. This highlights the importance of efficient, SIMD-ized inner loops for comparing HLLs.

We considered but did not include Ertl's Improved Estimator or the HyperLogLog++ estimator [34] in this study as they performed worse than Ertl's MLE in preliminary comparisons.

Optimizing Hardware Utilization

We designed Dashing to take advantage of (a) highly parallel general-purpose processors supporting dozens of simultaneous threads, as well as (b) inherent data parallelism within the HLL, allowing use of efficient SIMD instructions. Since distance calculations are functions of union and intersection cardinalities, and since the most common method (besides JMLE) for estimating intersection cardinality uses the inclusion-exclusion principle ($|A \cap B| = |A| + |B| - |A \cup B|$.), union cardinalities tend to be the key to overall efficiency. For two HyperLogLog sketches having the same size and hash function, a sketch of their union is simply the elementwise maximum of their registers. Thus, the important inner loops of the cardinality estimation methods involve such elementwise maximums.

Modern general-purpose processors support single-instruction multiple data (SIMD) instructions that can perform arithmetic and bitwise operations on vectors that are substantially wider (up to 512 bits) than a typical machine word (64 bits). Thus, substantial speedups can be attained by converting important loops can be made to use only or mostly SIMD instructions. The more operands that can be packed into the SIMD words, the greater the benefit, as was observed recently in bioinformatics tools that can operate on 64 8-bit values packed into a 512-bit word using Intel's AVX-512BW instruction set [22]. Therefore, we use 8-bit registers, even though we could hold all values in $[0, q - p + 1]$ with 6 bits when $p > 1$.

Besides these vectorized comparisons, tallying the frequencies the leading zero counts is the other chief computational cost. We mitigated this with manual loop unrolling, which reduced the cost by approximately 30%.

Parallelization

Use of simultaneous threads of execution was added using OpenMP v4.5. The `dashing sketch` function is parallelized across input files, with distinct threads reading, sketching, and writing sketches for distinct inputs simultaneously. The `dashing dist` function is parallelized such that distinct threads work on distinct rows of the upper-triangular all-pairs matrix simultaneously.

To minimize global locks on memory allocation, each thread is provided its own memory buffers. The all-pairs distance calculation uses multiple output buffers and asynchronous I/O to avoid needless blocking and contention on the output lock.

Another concern is load balance; having many threads is only useful if we can avoid having any "straggler" threads that run long after the others have finished. We eliminated one such source of stragglers by ordering the inputs to be sketched

from large to small. Choosing the largest genome first minimizes the chance that the thread with the largest genome will still be working when the others are finishing. Since input sequences are buffered in memory, this also causes each thread to reach its peak memory footprint more quickly, allowing overly memory-intensive jobs to fail immediately rather than after a long delay.

Sketching sequencing data

While Dashing supports both FASTA and FASTQ inputs, input data from sequencing experiments require special consideration due to the presence of sequencing errors. Following the strategy of Mash [1], Dashing uses an auxiliary data structure at sketching time to attempt o filter out k-mers that occur very infrequently, and which are therefore likely to contain errors. Dashing does this in a single pass. Each k-mer in a sequencing experiment is added to a Count-min Sketch [35], and only if the estimated count for that k-mer is sufficiently high is it added to the HLL. The advantage of a Count-Min sketch is accurate counting estimates in sublinear space.

Hash function

We compared clhash, Murmur3's finalizer, and the Wang hash across a set of synthetic Jaccard index estimates, and found that Wang's had the lowest error (8.20e-3) and bias (-2.14e-4), compared to 8.27e-3 and 2.30e-4 for Murmur3 and 8.21e-3 and -2.66e-4 for clhash. In addition to providing the best results, the Wang hash was also much faster than clhash, which is meant for string inputs rather than specialized for 64-bit integers.

Competing interests

The authors declare that they have no competing interests.

Author's contributions

DNB conceived the method and implemented the software. DNB and BL designed the experiments and wrote the paper. All authors read and approved the manuscript.

Acknowledgements

We thank Florian Breitwieser for HLL implementation discussions, and Nikita Ivkin for insights with regard to sketch data structure theory and implementation.

Grant support

BL and DNB were supported by National Science Foundation grant IIS-1349906 to BL and National Institutes of Health/National Institute of General Medical Sciences grant R01GM118568 to BL.

Data and software availability

- Dashing source code is available under the open source GPLv3 license at https://github.com/dnbaker/dashing.
- The particular version of Dashing evaluated here is tagged at https://github.com/dnbaker/dashing/releases/tag/v0.1.1.
- Scripts and code used to perform the experiments described in this study are available under the open source GPLv3 license at https://github.com/langmead-lab/dashing-experiments.
- The particular version of the scripts and code used to perform the experiments described in this study is tagged at: https://github.com/langmead-lab/dashing-experiments/releases/tag/v0.1.
- Accessions of genomes compared in the "Accuracy for complete genomes" subsection of the "Results" section are listed at: https://github.com/langmead-lab/dashing-experiments/blob/master/accuracy/genomes_for_exp.txt.
- Accessions of genomes compared in the "Computational efficiency" subsection of the "Results" section are listed at: https://github.com/langmead-lab/dashing-experiments/blob/master/timing/filenames.txt.

References

1. Ondov, B.D., Treangen, T.J., Melsted, P., Mallonee, A.B., Bergman, N.H., Koren, S., Phillippy, A.M.: Mash: fast genome and metagenome distance estimation using MinHash. Genome Biol. **17**(1), 132 (2016)
2. Schaeffer, L., Pimentel, H., Bray, N., Melsted, P., Pachter, L.: Pseudoalignment for metagenomic read assignment. Bioinformatics **33**(14), 2082–2088 (2017)
3. Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H., Phillippy, A.M.: Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Res. **27**(5), 722–736 (2017)
4. Berlin, K., Koren, S., Chin, C.S., Drake, J.P., Landolin, J.M., Phillippy, A.M.: Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. Nat. Biotechnol. **33**(6), 623–630 (2015)
5. Jain, C., Koren, S., Dilthey, A., Phillippy, A.M., Aluru, S.: A fast adaptive algorithm for computing whole-genome homology maps. Bioinformatics **34**(17), 748–756 (2018)
6. Jain, C., Rodriguez-R, L.M., Phillippy, A.M., Konstantinidis, K.T., Aluru, S.: High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries. Nat Commun **9**(1), 5114 (2018)
7. Broder, A.Z.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings, pp. 21–29 (1997). IEEE
8. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biol. **15**(3), 46 (2014)
9. Li, H.: Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. Bioinformatics **32**(14), 2103–2110 (2016)
10. Buhler, J.: Efficient large-scale sequence comparison by locality-sensitive hashing. Bioinformatics **17**(5), 419–428 (2001)
11. Luo, Y., Yu, Y.W., Zeng, J., Berger, B., Peng, J.: Metagenomic binning through low-density hashing. Bioinformatics (2018)
12. Zhao, X.: Bindash, software for fast genome distance estimation on a typical personal laptop. Bioinformatics, 651 (2018)
13. Yu, Y.W., Weber, G.: Hyperminhash: Jaccard index sketching in loglog space. CoRR **abs/1710.08436** (2017). 1710.08436
14. Ertl, O.: Superminhash - A new minwise hashing algorithm for jaccard similarity estimation. CoRR **abs/1706.05698** (2017). 1706.05698
15. Koslicki, D., Zabeti, H.: Improving min hash via the containment index with applications to metagenomic analysis (2017). doi:10.1101/184150
16. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Jacquet, P. (ed.) AofA: Analysis of Algorithms. DMTCS Proceedings, vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), pp. 137–156. Discrete Mathematics and Theoretical Computer Science, Juan les Pins, France (2007). https://hal.inria.fr/hal-00406166
17. Breitwieser, F.P., Baker, D.N., Salzberg, S.L.: KrakenUniq: confident and fast metagenomics classification using unique k-mer counts. Genome Biol. **19**(1), 198 (2018)
18. Crusoe, M.R., Alameldin, H.F., Awad, S., Boucher, E., Caldwell, A., Cartwright, R., Charbonneau, A., Constantinides, B., Edvenson, G., Fay, S.e.a.: The khmer software package: enabling efficient nucleotide sequence analysis. F1000Res **4**, 900 (2015)
19. Georganas, E., Buluç, A., Chapman, J., Oliker, L., Rokhsar, D., Yelick, K.: Parallel de bruijn graph construction and traversal for de novo genome assembly. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '14, pp. 437–448 (2014)
20. Ertl, O.: New cardinality estimation algorithms for hyperloglog sketches. CoRR **abs/1702.01284** (2017). 1702.01284
21. Brown, C.T., Irber, L.: sourmash: a library for minhash sketching of dna. The Journal of Open Source Software **1**(5) (2016)
22. Rahn, R., Budach, S., Costanza, P., Ehrhardt, M., Hancox, J., Reinert, K.: Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading. Bioinformatics **34**(20), 3437–3445 (2018)
23. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
24. Swamidass, S.J., Baldi, P.: Mathematical correction for fingerprint similarity measures to improve chemical retrieval. J Chem Inf Model **47**(3), 952–964 (2007)
25. Wang, T.: Integer Hash Function . http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm. [Online; accessed by archive July 2017] (1997)
26. O'Leary, N.A., Wright, M.W., Brister, J.R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D.e.a.: Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. Nucleic Acids Res. **44**(D1), 733–745 (2016)
27. Bovee, R., Greenfield, N.: Finch: a tool adding dynamic abundance filtering to genomic minhashing. Journal of Open Source Software **3(22)** (2018)
28. Ertl, O.: Bagminhash - minwise hashing algorithm for weighted sets. CoRR **abs/1802.03914** (2018). 1802.03914
29. Chum, O., Philbin, J., Zisserman, A., *et al.*: Near duplicate image detection: min-hash and tf-idf weighting. In: BMVC, vol. 810, pp. 812–815 (2008)
30. Solomon, B., Kingsford, C.: Fast search of thousands of short-read sequencing experiments. Nat. Biotechnol. **34**(3), 300–302 (2016)
31. Morris, R.: Counting large numbers of events in small registers. Commun. ACM **21**(10), 840–842 (1978)
32. Flajolet, P.: Approximate counting: A detailed analysis. BIT Numerical Mathematics **25**(1), 113–134 (1985)
33. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. Journal of computer and system sciences **31**(2), 182–209 (1985)

34. Heule, S., Nunkesser, M., Hall, A.: Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In: Proceedings of the 16th International Conference on Extending Database Technology. EDBT '13, pp. 683–692 (2013)
35. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms **55**(1), 58–75 (2005). doi:10.1016/j.jalgor.2003.12.001

Additional Files

Full results for sketch Accuracy
Experimental comparison of MinHash, Bloom, Bloom+, and HyperLogLog for Jaccard-coefficient estimation on synthetic data in tabular format.

Full results for accuracy for complete genomes
Jaccard coefficient estimation accuracy across a range of true Jaccard values for BinDash, Mash and 3 HyperLogLog estimation algorithms in tabular format. Experiments were repeated for all combinations of $k \in \{16, 21, 31\}$ and $\log_2$ sketch size $\in \{10, 11, 12, 13, 14, 15\}$.

Full results for computational efficiency
Space and time efficiency benchmark for all pairwise comparisons between 87,113 genomes for $k \in \{16, 21, 31\}$ and $\log_2$ sketch size $\in \{10, 14, 16\}$ between BinDash, Mash, and 3 HyperLogLog estimation algorithms in tabular format.