APPLIED RANDOMIZED ALGORITHMS FOR EFFICIENT GENOMIC ANALYSIS

by

Daniel N. Baker

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for the degree of
Doctor of Philosophy

Baltimore, Maryland
January, 2022

Abstract

The scope and scale of biological data continues to grow at an exponential clip, driven by advances in genetic sequencing, annotation and widespread adoption of surveillance efforts. For instance, the Sequence Read Archive (SRA) now contains more than 25 petabases of public data, while RefSeq, a collection of reference genomes, recently surpassed 100,000 complete genomes. In the process, it has outgrown the practical reach of many traditional algorithmic approaches in both time and space.

Motivated by this extreme scale, this thesis details efficient methods for clustering and summarizing large collections of sequence data. While our primary area of interest is biological sequences, these approaches largely apply to sequence collections of any type, including natural language, software source code, and graph structured data.

We applied recent advances in randomized algorithms to practical problems. We used MinHash and HyperLogLog, both examples of Locality-Sensitive Hashing, as well as coresets, which are approximate representations for finite sum problems, to build methods capable of scaling to billions of items. Ultimately, these are all derived from variations on sampling.

We combined these advances with hardware-based optimizations and

incorporated into free and open-source software libraries (sketch, frp, libsimdsampling) and practical software tools built on these libraries (Dashing, Minicore, Dashing 2), empowering users to interact practically with colossal datasets on commodity hardware.

Thesis Committee

Primary Readers

Ben Langmead (Primary Advisor)
Associate Professor
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Vladimir Braverman
Associate Professor
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Michael Schatz
Bloomberg Distinguished Professor
Department of Biology
Johns Hopkins Krieger School of Arts and Sciences
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Alex Szalay
Bloomberg Distinguished Professor
Department of Physics and Astronomy
Johns Hopkins Krieger School of Arts and Sciences
Department of Computer Science
Johns Hopkins Whiting School of Engineering
Director, Institute for Data-Intensive Engineering and Sciences

Alternate Readers

Stephanie Hicks
Assistant Professor
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Acknowledgments

Otmar Ertl, whose theoretical contributions form a foundation for much of this work, including personal communications. Nikita Ivkin, Anton Belyy, and Vladimir Braverman, for theoretical discussions and practical insights. Florian Breitwieser, for his work with HyperLogLog and sketching applications.

My advisor, Ben Langmead, for both guiding and supporting, while still giving me the freedom to explore a range of methods and applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Biological data is big data. As raw sequencing data continues to accelerate, and the set of reference sequences continues to grow, developing methods that can scale to match it becomes both more rewarding and more challenging.

Sequence similarity search is a core utility in analyzing biological data. You might use similarity search to classify new assemblies, high-throughput sequencing reads, de-duplicate a collection of genomes or sequences, or to identify homologous regions in large collections. Methods that can efficiently summarize and index large sets of this high-dimensional data are widely applicable.

In this thesis, I will explore randomized algorithms for similarity search and data summarization, apply them to large problems in computational biology and structured data, tailor solutions to commodity hardware, and discuss future applications and open problems.

1.1 Sketching

Sketching algorithms allow analysis of unbounded stream size in near-linear time with approximation guarantees. Designed for massive streams for from telecommunications and other big data sources, they provide a set of methods for accelerating downstream analysis. Specifically, a **streaming algorithm** is an algorithm which requires only one pass through the data and uses at most polylog space. Both the space constraint - sublinear memory - and the single-pass constraint provide speed and flexibility advantages.

Motivated by both rapidly accelerating data accumulation and the increased capabilities afforded by this information, many analytical methods supporting almost-linear time and sublinear space have become key methods for big data analysis. Approximate Counting [1] [2] serves as a precursor to modern streaming algorithms.

High-dimensional data can be summarized with sampling techniques as well as general dimension reduction techniques. *sketching*, which reduces dimensionality while providing some guarantees with regard to approximation, uses a form of sampling and preserves some desired quantity with high probability. For instance, the Johnson-Lindenstrauss (JL) transform approximately preserves ℓ_2 distances between pairs of points by pseudorandom matrix multiplication, and MinHash sampling approximately preserves set similarity between pairs of sets by sampling. A substantial number of these techniques are **Linear Sketches**.

1.1.1 Linear Sketching

There are many useful "linear sketches" - sketches which can be described by a matrix multiplication. These include some of the most useful sketching algorithms - The Count-Min Sketch ([3]), which provides approximate counts which only over-estimates, the Count Sketch [4], which is unbiased but tends to preserve the values for high-count items, and the many variations of the Johnson-Lindenstrauss (JL) transform [5], which approximately preserve ℓ_2 distances.

The matrix formalism also comes with several useful properties for distributed computation.

First, they are composable - the union of two sketches is the sketch of their unions. For instance, if the data is distributed across machines, it can be summarized and reduced using a merge tree of $\log_n$ iterations. This will also be true for other kinds of sketches.

Second, this means that they support deletions, at least in this form. This can then be used to perform turnstile updates to generate sketches for sliding windows, or they can be dynamic structures.

And lastly, they can be used for approximate linear algebra problems, such as compressed sensing and linear programming [6, 7].

1.1.1.1 Count-Min

Count-Min sketch [3] is a linear sketch in which each input feature is hashed to a pseudorandom register for each subtable. This can be described by a large matrix with one ‘1‘ for each range of columns corresponding to each

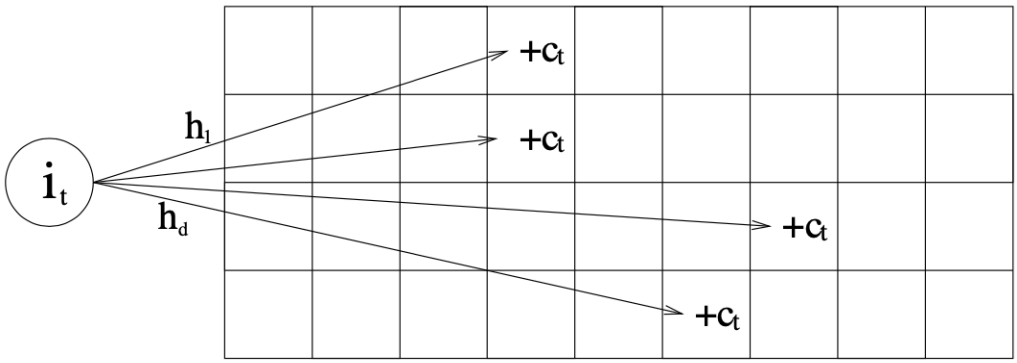


Figure 1.1: Count-Min Sketch Visualization by Cormode and Muthukrishnan

subtable. See figure 1.1, which has been taken from the original Cormode and Muthukrishnan paper.

At query time, the minimum value across all subtables yields an estimate which never under-estimates.

There are small improvements (conservative-update, in particular, improves accuracy), but they often come at the expense of reversibility or composability.

The popular "feature hashing" technique is a special case of the Count-Min sketch of a single row. It can be used to cap the space requirement of counting algorithms, and is particularly common in NLP applications.

1.1.1.2 Count Sketch

The Count sketch ([4]) is a linear sketch in which each input feature is hashed to a pseudorandom register for each subtable. This can be described by a large matrix with one Rademacher random variable (± 1, each 50%-50% probability) for each range of columns corresponding to each subtable. (This has been

generalized to higher-order tensors in works such as the higher-order Count Sketch and the Tensor Sketch [**hocs**, **ahle-tensor-sketch**].)

Instead of returning the minimum estimate like the Count-Min sketch (which only over-estimates), the Count-Sketch estimate is the median of subtable values; this pools information across the subtables in a way that supports two-sided errors.

The Count-Min sketch satisfies the JL property because its matrix is subgaussian, and while its error tends to be higher than random Gaussian matrices, it still provides the same asymptotic guarantees. The fact that its generation is only $\mathcal{O}(\text{nnz})$ makes it particularly attractive, as full random matrix-vector multiplication is $\mathcal{O}(\text{nnz} \times \text{sketchdim})$. For more details, see the excellent book by David Woodruff, Sketching as a Tool for Numerical Linear Algebra [8].

1.1.1.3 p-stable sketching & JL Transform

Another example of linear sketches are the [9] families of locality-sensitive hashing (LSH) functions. Given $p > 1$, ℓ_p distances can be preserved with some expectation using random matrices whose values are sampled from specific probability distributions. If $p = 2$, this simplifies to Gaussian-distributed data with mean 0 and unit variance. For $p = 1$, this is the Cauchy distribution, although truncating extreme values from this sampling may be necessary for good behavior due to its extreme variance.

Further, since these are linear sketches, they can be used to estimate ℓ_p distances between points in ambient space from sketches by taking the distance

between two points and estimating its ℓ_p norm.

Hyperplane LSH and cross-polytope LSH use similar techniques to build sketches for cosine distance and inner product search. Efficient implementations of these methods often use structured and/or sparse matrices, as they can be evaluated quickly. Our software library [10] implements a several of these using structured matrices.

1.2 MinHash & HyperLogLog

The family of sketching methods at the center of this thesis are MinHash derivatives. Developed for the AltaVista search engine by Broder [11] for indexing text, MinHash is the foundation of locality-sensitive hashing. Most if not all LSH families derive from a reduction to set MinHash. At its core, MinHash is **consistent sampling**. By applying the same sampling method to all inputs, similar inputs yield similar samples.

Specifically, given a pseudorandom ranking function H applied to all items from two sets of items S_1 and S_2, let $MH(S) = \min_{x \in S} H(x)$. (Note: the argmin is equivalent.) The probability that $MH(S_1) = MH(S_2) = \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$, the Jaccard coefficient, or similarity of the sets. This simple concept extends to faster methods of sketch generation [12, 13] yielding equivalent statistical properties.

One particular variant, the HyperLogLog is an example of the one-permutation [13] MinHash sketch with $\log_2$-truncated signatures. This is very space-efficient while still yielding high-accuracy set comparisons, and has applications in privacy as well. I applied used this data structure in the Dashing

software tool and accompanying paper.

Extended variants can account for counts of items [14, 15], or be more specific for edit distance [16] or probability distributions [17]. I implemented these, along with the generalized logarithmic truncation, in the Dashing2 software tool with its forthcoming accompanying paper.

I build LSH indexes over input collections, allowing us to search and index enormous genomic collections in practical time. This core utility facilitates wide ranges of biological inquiry, and I demonstrate its usefulness and accuracy.

1.2.1 MinHash Applications

MinHash and minimizers (original k-mers corresponding to the MinHash register for a window) typically serve a core initial step in recent efficient sequence aligners. While hash indexes have been common components of mapping software for over a decade, minimizer indexes were brought into mainstream aligners by Jain et al. in MashMap [18] and improved upon by both Li and and Jain, et al. [19, 20].

They also play a key role in performance for k-mer classification software - notably in Kraken, KrakenUniq and Kraken2 [21, 22, 23]. For Kraken and KrakenUniq, they simply affect locality when accessing a hash table, but for Kraken2 (and our work, bonsai [24]) on the minimizers are indexed. In essence, by indexing only minimizers, the database can be shrunk while maintaining high accuracy.

Minimizer sequence transduction also plays an important role in recent

assembly software. Introduced by ntJoin [25] and improved with MBG [26]. By replacing a sequence of characters with a sequence of minimizers, individual characters become more specific and less numerous. This comes at the expense of some accuracy but with extreme performance gains.

Lastly, the use of minimizers for pre-filtering plays a key role in linear-time clustering of large databases. Similar approaches have been part of the CD-HIT (Cluster Database with High-Identity with Tolerance) [27] method for a long time, and it plays an important step in the linear-time protein clustering algorithm Linclust by Steinegger, et al. [28]. In this thesis, I primarily focus on improvements to sketching, comparing, and neighbor queries.

1.3 Coresets

While sketching as above reduces the dimensionality of data, one might instead want to reduce the number of items under consideration. Coresets [29], developed by computational geometry, provide a framework for generating approximate representations for datasets which preserve the overall shape of it.

This provides an orthogonal tool for compressing data; indeed, it can be performed jointly [30] with dimensionality reduction. And it is especially important for cases where the dimensionality of the data can not be reduced.

Most practical coreset methods are derived from importance sampling. They can be used to accelerate gradient descent [31], summarize road networks or point clouds [32, 33].

For this thesis, I implemented efficient methods for coreset generation

in the Minicore [34] software library, and expose them with both C++ and Python front-ends. These include discrete optimization algorithms, methods for graph-structured data, numerical linear algebra techniques, fast sampling software, and hardware-tailored optimizations for these problems.

1.3.1 Coresets for Single-cell Applications

Droplet Single-cell RNA-Seq datasets are often considered to be derived from a multinomial [35] distribution. This allows us to imbue algorithms with a richer notion of distance than the typical Euclidean or squared Euclidean distance, the latter of which corresponds to a log-likelihood ratio test using a Gaussian model with uniform variance. However, this was both slower than Euclidean distances and more complex, as typical distance calculations may be undefined or infinite when working with sparse data.

Typical acceleration methods for Euclidean-space data did not apply; I could not use p-stable locality-sensitive hashing methods to avoid exhaustive all-points to all-centers distance calculation. And dimensionality reduction techniques designed for ℓ_p spaces, such as the JL transform [5] or Count Sketch [4], would both violate nonnegativity constraints, and provide no approximation guarantees for Bregman divergences besides squared Euclidean distances.

The 2015 paper by Lucic, Bachem and Krause [36] provided an algorithm for generating coresets for clustering with Bregman divergences by first building an approximate solution to k-means clustering with μ-similar Bregman divergences. It reduces to the kmeans++ [37] algorithm, using the relevant

distance. Lucic, et al., [36] refers to it as D^2 sampling. This means that an initial approximate solution can be generated in $\mathcal{O}(ndk)$ time - k for the number of iterations, d for the dimension, a limit on the time distances can require, and n for the total number of items in the dataset.

There are methods for dimensionality reduction which can satisfy distributional assumptions, specifically generalized PCA. The 2001 Collins method [38] provides a framework and algorithm for performing principle component analysis using Bregman divergences of choice. However, this method is so slow as to be impractical on large datasets. Related, variational autoencoders, which are themselves neural network analogs of PCA, can be parameterized using relevant distributions, but are typically impractical without specialized co-processor acceleration, such as GPUs and TPUs.

Importantly, our method's speed improvements are most dramatic for the case of dense data, yielding improvements of over 700x single-threaded, allowing us to accelerate methods by factors of many thousands with multi-threaded processing. Our software would be well-applied for processing and summarizing previously dimension-reduced data.

References

[1] Robert Morris. "Counting Large Numbers of Events in Small Registers". In: *Commun. ACM* 21.10 (1978), 840–842. ISSN: 0001-0782. DOI: 10.1145/359619.359627. URL: https://doi.org/10.1145/359619.359627.

[2] Philippe Flajolet. "Approximate Counting: A Detailed Analysis". In: *BIT* 25.1 (1985), 113–134. ISSN: 0006-3835. DOI: 10.1007/BF01934993. URL: https://doi.org/10.1007/BF01934993.

[3] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58 –75. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0196677403001913.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding Frequent Items in Data Streams". In: *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*. ICALP '02. 2002, 693–703. ISBN: 3540438645.

[5] William B. Johnson. "Extensions of Lipschitz mappings into Hilbert space". In: *Contemporary mathematics* 26 (1984), pp. 189–206.

[6] Graham Cormode and S. Muthukrishnan. "Combinatorial Algorithms for Compressed Sensing". In: *Structural Information and Communication Complexity*. Ed. by Paola Flocchini and Leszek Gąsieniec. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 280–294. ISBN: 978-3-540-35475-8.

[7] Ping Li, Cun-Hui Zhang, and Tong Zhang. *Compressed Counting Meets Compressed Sensing*. 2013. arXiv: 1310.1076 [stat.ME].

[8] David P. Woodruff. "Sketching as a Tool for Numerical Linear Algebra". In: *CoRR* abs/1411.4357 (2014). arXiv: 1411.4357. URL: http://arxiv.org/abs/1411.4357.

[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. "Locality-Sensitive Hashing Scheme Based on p-Stable Distributions". In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. SCG '04. Brooklyn, New York, USA: Association for Computing Machinery, 2004, 253–262. ISBN: 1581138857. DOI: 10.1145/997817.997857. URL: https://doi.org/10.1145/997817.997857.

[10] Daniel Baker. *Fast Random Projections*. https://github.com/dnbaker/frp. 2017.

[11] Andrei Z Broder. "On the resemblance and containment of documents". In: *Compression and complexity of sequences 1997. proceedings*. IEEE. 1997, pp. 21–29.

[12] Otmar Ertl. "SetSketch: Filling the Gap between MinHash and HyperLogLog". In: *CoRR* abs/2101.00314 (2021). arXiv: 2101.00314. URL: https://arxiv.org/abs/2101.00314.

[13] Ping Li, Art Owen, and Cun-hui Zhang. "One Permutation Hashing". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/eaa32c96f620053cf442ad32258076b9-Paper.pdf.

[14] Otmar Ertl. "BagMinHash - Minwise Hashing Algorithm for Weighted Sets". In: *CoRR* abs/1802.03914 (2018). arXiv: 1802.03914. URL: http://arxiv.org/abs/1802.03914.

[15] Otmar Ertl. "ProbMinHash - A Class of Locality-Sensitive Hash Algorithms for the (Probability) Jaccard Similarity". In: *CoRR* abs/1911.00675 (2019). arXiv: 1911.00675. URL: http://arxiv.org/abs/1911.00675.

[16] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford. "Locality-sensitive hashing for the edit distance". In: *Bioinformatics* 35.14 (2019), pp. i127–i135.

[17] Lin Chen, Hossein Esfandiari, Thomas Fu, and Vahab S. Mirrokni. *Locality-Sensitive Hashing for f-Divergences: Mutual Information Loss and Beyond*. 2019. arXiv: 1910.12414 [cs.LG].

[18] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru. "A fast adaptive algorithm for computing whole-genome homology maps". In: *Bioinformatics* 34.17 (2018), pp. i748–i756.

[19] H. Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (2018), pp. 3094–3100.

[20] C. Jain, A. Rhie, H. Zhang, C. Chu, B. P. Walenz, S. Koren, and A. M. Phillippy. "Weighted minimizer sampling improves long read mapping". In: *Bioinformatics* 36.Suppl_1 (2020), pp. i111–i118.

[21] D. E. Wood and S. L. Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biol.* 15.3 (2014), R46.

[22] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. "KrakenUniq: confident and fast metagenomics classification using unique k-mer counts". In: *Genome Biol.* 19.1 (2018), p. 198.

[23] D. E. Wood, J. Lu, and B. Langmead. "Improved metagenomic analysis with Kraken 2". In: *Genome Biol* 20.1 (2019), p. 257.

[24] Daniel Baker. *Bonsai - Fast, flexible taxonomic analysis and classification*. https://github.com/dnbaker/bonsai. 2016-2021.

[25] L. Coombe, V. Nikolić, J. Chu, I. Birol, and R. L. Warren. "ntJoin: Fast and lightweight assembly-guided scaffolding using minimizer graphs". In: *Bioinformatics* 36.12 (2020), pp. 3885–3887.

[26] M. Rautiainen and T. Marschall. "MBG: Minimizer-based Sparse de Bruijn Graph Construction". In: *Bioinformatics* (2021).

[27] L. Fu, B. Niu, Z. Zhu, S. Wu, and W. Li. "CD-HIT: accelerated for clustering the next-generation sequencing data". In: *Bioinformatics* 28.23 (2012), pp. 3150–3152.

[28] M. Steinegger and J. Söding. "Clustering huge protein sequence sets in linear time". In: *Nat Commun* 9.1 (2018), p. 2542.

[29] Sariel Har-Peled. "Clustering Motion". In: *Discrete & Computational Geometry* 31.4 (2004), pp. 545–565.

[30] Dan Feldman and Michael Langberg. "A unified framework for approximating and clustering data". In: *43rd Annual ACM Symposium on Theory of computing*. ACM. 2011, pp. 569–578.

[31] Baharan Mirzasoleiman, Jeff A. Bilmes, and Jure Leskovec. "Data Sketching for Faster Training of Machine Learning Models". In: *CoRR* abs/1906.01827 (2019). arXiv: 1906.01827. URL: http://arxiv.org/abs/1906.01827.

[32] Daniel N. Baker, Vladimir Braverman, Lingxiao Huang, Shaofeng H.-C. Jiang, Robert Krauthgamer, and Xuan Wu. "Coresets for Clustering in Graphs of Bounded Treewidth". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 569–579. URL: http://proceedings.mlr.press/v119/baker20a.html.

[33] Lingxiao Huang and Nisheeth K. Vishnoi. "Coresets for clustering in Euclidean spaces: importance sampling is nearly optimal". In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*. Ed. by Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy. ACM, 2020, pp. 1416–1429. DOI: 10.1145/3357713.3384296. URL: https://doi.org/10.1145/3357713.3384296.

[34] D. N. Baker, N. Dyjack, V. Braverman, S. C. Hicks, and B. Langmead. "k-means clustering with various distances". In: *ACM BCB* 2021 (2021).

[35] F. W. Townes, S. C. Hicks, M. J. Aryee, and R. A. Irizarry. "Feature selection and dimension reduction for single-cell RNA-Seq based on a multinomial model". In: *Genome Biol* 20.1 (2019), p. 295.

[36] Mario Lucic, Olivier Bachem, and Andreas Krause. "Strong Coresets for Hard and Soft Bregman Clustering with Applications to Exponential Family Mixtures". In: *CoRR* (2016). arXiv: 1508.05243 [stat.ML].

[37] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: *SODA*. SODA '07 (2007), 1027–1035.

[38] Michael Collins, Sanjoy Dasgupta, and Robert E. Schapire. "A Generalization of Principal Component Analysis to the Exponential Family". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. NIPS'01. Vancouver, British Columbia, Canada: MIT Press, 2001, 617–624.

Chapter 2

Dashing: Fast and Accurate Genomic Distances using HyperLogLog

2.1 Context

This thesis chapter is comprised of the Dashing paper, published in Genome Biology in 2019 [1]. Its core contribution consists of using the HyperLogLog as an compact, approximate representation of *k*-mer sets allowing rapid summarization and comparison of sequencing datasets. We also leverage hardware intrinsics to accelerate this process, and provide a considerable runtime improvement to genomic analyses while also improving their accuracy.

2.2 Background

Since the release of the seminal Mash tool [2], data sketches such as MinHash have become instrumental in comparative genomics. They are used to cluster genomes from large databases [2], search for datasets with certain sequence

content [3], accelerate the overlapping step in genome assemblers [4, 5], map sequencing reads [6], and find similarity thresholds characterizing species-level distinctions [7]. Whereas MinHash was originally developed to find similar web pages [8], here it is being used to summarize large genomic sequence collections such as reference genomes or sequencing datasets. A collection is reduced to a set of representative k-mers and ultimately 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 cardinalities one can obtain a Jaccard coefficient (J) or a "Mash distance," which is a proxy for Average Nucleotide Identity (ANI) [2]. These make it possible to cluster sequences and otherwise solve massive genomic nearest-neighbor problems.

MinHash is related to other core methods in bioinformatics. Minimizers, which can be thought of as a special case of MinHash, are widely used in metagenomics classification [9] and alignment and assembly [10]. 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 [11] and metagenomics classification [12].

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

proposed in the HyperMinHash [14] and SuperMinHash [15] studies.

Some studies have pointed out shortcomings of MinHash. Koslicki and Zabeti argue that MinHash cardinality estimates suffer when the sets are very different sizes [16]. 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 use the HyperLogLog (HLL) sketch [17] 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 and when the sketch data structures are quite small. 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 [18, 19, 20]. We additionally use recent theoretical improvements in cardinality estimation for set unions and intersections [21], the components needed to estimate J and other similarity measures.

We implemented the HLL in the Dashing software tool [22] (`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 [2], BinDash [13] and Sourmash [23]. Dashing can build a sketch of an input sequence set (`dashing sketch`), including FASTA files (for assembled genomes) or FASTQ files (for sequencing datasets). 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. Since Dashing's `sketch` function is extremely fast, Dashing can perform both sketching and all-pairs distance calculations in the same command, obviating the need 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) instructions on modern general-purpose computer processors to exploit the finer-grained parallelism inherent in HLL computations.

2.3 Results

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

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 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.

2.3.1 Design

Dashing uses the HyperLogLog (HLL) sketch to solve genomic distance problems. 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. Dashing prints a set of similarity estimates, including estimates for Jaccard coefficient and ANI. It can operate on a given pair of datasets, or can perform all-pairs comparisons across many datasets in a single invocation of the tool.

Dashing is written in C++14. It uses OpenMP for multithreading, with both the sketching and distance phases readily scaling to 100 simultaneous threads. It also uses data-parallel SIMD instructions, including the recent AVX512-BW extensions that have been effective at accelerating other bioinformatics software [24]. Dashing has Python bindings that enable other developers to use the HLL implementation.

2.3.2 Accuracy for complete genomes

Encouraged by HLL's accuracy, we measured the accuracy of Dashing v0.1.2's HLL-based Jaccard Coefficientestimates versus those of Mash v2.1 [2] and BinDash v0.2.1 [13]. We repeated the HLL experiments for three HLL cardinality estimation methods: Flajolet's canonical method using harmonic mean [17], and two maximum-likelihood-based methods (MLE and JMLE) proposed by Ertl [21]. We selected 400 pairs of bacterial genomes from RefSeq [25] covering a range of Jaccard Coefficientvalues. To select the pairs, we first used `dashing dist` with $s = 16$, $k = 31$ and the MLE estimation method on the full set of

complete bacterial RefSeq assemblies (latest versions). We then selected a subset such that we kept 4 distinct genome pairs per Jaccard Coefficientpercentile. Our goal was to test an even spread of Jaccard Coefficientvalues, though some unevenness emerged 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 Coefficientestimates for the 400 pairs and plotted the results versus true J, 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 2.1. The horizontal axis is divided into 10 J partitions, each containing about 40 pairs. The vertical axis shows the difference between tool-estimated and true Jaccard coefficient. 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.

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. 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.

shows mean squared Jaccard Coefficientestimation error (meanSE) for a range of sketch sizes and for $k \in \{16, 21, 31\}$, also including the two alternate

cardinality estimation methods for Dashing (Original and JMLE). In short, BinDash and Dashing consistently achieve lower meanSE than Mash, with BinDash achieving the lowest meanSE at smaller J's and both BinDash and Dashing achieving similar meanSE at intermediate and larger J's. Among the Dashing estimation methods, JMLE consistently achieves the lowest meanSE. For computational efficiency reasons (discussed later), Dashing's default estimation method is the MLE, which had only slightly higher error than JMLE.

2.3.3 Computational efficiency

To assess computational efficiency and scalability in a realistic context, we used Dashing v0.1.2, Mash v2.1 and BinDash v0.2.1 to sketch and perform all-pairs distance calculations for 87,113 complete genome assemblies. We obtained the assemblies from Refseq, filtering to include only assemblies marked "latest" and "Complete genome" and without "contig" in the name. 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. The total number of genome-pair distance calculations required for 87,113 assemblies was over 3.79 billion. 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, each with 14 processor cores. After hyperthreading, the system supports up to 112 simultaneous hardware threads. The system had 1 TB of DDR4 RAM, and ran CentOS 7.5 Linux, kernel v3.10.0. The system was located at and maintained by the Maryland Advanced Research Computing

Center (MARCC).

For Dashing, we used `dashing sketch` for sketching and `dashing dist` for pairwise distance calculations. For Mash, we used `mash sketch` and `mash triangle` for the two stages respectively. For BinDash we used `bindash sketch` and `bindash dist`. We also ran each tool in a way that performed sketching immediately followed by all-pairs distance calculations. For Mash, this involves running its `dist` and `triangle` commands but specifying the sequence files (rather than their sketches) as input. In the case of `dashing dist`, this combined invocation avoids writing any sketches to disk. Mash provides support for this functionality as well, but we were unable to run it successfully for our large experiment.

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 used the GNU `time` utility to measure the average number of CPUs utilized, wall time and peak memory footprint for each tool invocation.

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 $k = 21$ and $k = 31$ are summarized in Figure 2.2 and a tabular version of the results for $k = 31$ is shown in Table 2.1.

We observed that Dashing is the fastest tool in the Sketch phase, running 3.3–4.3 times faster than BinDash and 3.8–5.0 times faster than Mash.

Table 2.1: Comparison of computational efficiency of Mash, BinDash and Dashing at $k = 31$ and various sketch sizes. The $\log 2(size)$ column reports the $\log_2$ of the sketch size in bytes. "Both" results obtained either by using a combined Sketch+Distance mode (for Dashing) or by combining results from separate sketching and distance-calculation 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	1,345	1,157	273	**271**	277
			12	1,349	1,157	273	274	**270**
			14	1,356	1,159	286	289	**278**
			16	1,400	1,226	359	367	**299**
	Peak mem (MB)	31	10	17,720	**141**	12,683	12,721	12,644
			12	18,296	**399**	12,723	12,430	12,726
			14	19,706	**1,426**	12,630	12,877	12,853
			16	25,127	**5,542**	12,888	12,412	12,933
Distance	Wall clock (s)	31	10	1,901	**74**	80	100	601
			12	2,368	**188**	286	308	2,139
			14	3,446	**672**	1,113	1,137	8,308
			16	8,777	**3,603**	6,172	4,251	30,506
	Peak mem (MB)	31	10	1,120	409	**116**	**116**	**116**
			12	1,380	673	**371**	**371**	372
			14	2,785	1,709	**1,392**	**1,392**	**1,392**
			16	10,776	5,816	**5,476**	**5,476**	**5,476**
Both	Wall clock (s)	31	10	3,246	1,231	**345**	365	870
			12	3,717	1,345	**557**	579	2,407
			14	4,801	1,831	**1,390**	1,408	8,574
			16	10,177	4,829	**4,394**	4,453	30,433
	Peak mem (MB)	31	10	17,720	**409**	12,468	12,950	12,988
			12	18,296	**673**	12,958	13,042	13,020
			14	19,706	**1,709**	13,951	13,782	14,205
			16	25,127	**5,816**	18,320	18,081	18,011

BinDash achieves the lowest memory footprint among the tools in the Sketch phase, requiring 140 MB for the 1-KB sketch and 5.5 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 17–25 GB.

In the Distance phase, we noted that the estimation method had a major effect on Dashing's speed, with JMLE performing 5.9–7.4 times slower than MLE. This is because the JMLE performs 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. (In a separate experiment, we found that the JMLE inner loop could be made about 20% faster using AVX512BW instructions.)

BinDash was the fastest tool in the Distance phase, running 25–70% faster than Dashing's MLE mode. But Dashing is 2–19 times faster than Mash, with the largest speed gap observed at the smallest (1KB) sketch size.

When we compared tools based on combined performance in both the Sketch and Distance phases, BinDash again had the lowest memory footprint (always below 6GB), with Dashing's footprint at 12–18 GB and Mash's at 17–25GB. Dashing was the fastest among the three tools at all sketch sizes, though BinDash achieves similar speed at the largest (64KB) sketch size. Mash was the slowest of the tools in all cases. Since small sketch sizes tend to be used in practice (Mash's default is 4KB or 2^{12} bytes), we expect Dashing to be the fastest overall tool — certainly for sketching, but also combined sketching and distance calculations — in typical situations.

2.3.4 Thread scaling

We also compared the tools' speed and memory footprint when run with 4, 8 and 16 threads. We found that all three tools achieved excellent thread scaling in the sketching phase, where Dashing achieves the highest throughput. We

also found that, for the distance estimation phase, Dashing exhibited better thread scaling compared to Mash and BinDash.

2.4 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 [18, 19, 20], this is the first application to the problem of estimating genomic distances, the first implementation of the highly accurate MLE and Joint-MLE estimators [21], and the first comprehensive comparison to MinHash and similar methods. The combination of HLL and JMLE is also notable since it directly estimates the cardinality of an intersection, a meaningful quantity independent of its use in the Jaccard coefficient.

We implemented HLL-based sketching and distance calculations in the Dashing software tool. Dashing can sketch and calculate pairwise distances for over 87K Refseq [25] genomes in around 6 minutes using its MLE estimation method, 1KB sketch size, and 100 simultaneous threads of execution (Table 2.1).

Dashing's 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 with smaller sketches justifies a lower default sketch size (1KB) compared to Mash's default of 4KB (or 8KB for long k-mers).

It is interesting to observe that Dashing's accuracy is comparable to that of BinDash across the Jaccard-index deciles in Table 2.1. Though Dashing is faster — both at sketching and at combined sketching-and-distance — BinDash's speed approaches that of Dashing at the highest sketch size tested. As we continue to investigate the HyperLogLog sketch, the b-Bit Minwise Hashing technique underlying BinDash is clearly a close competitor, and it will be important to continue to study it as well. In particular, b-Bit Minwise Hashing is also more amenable to SIMD acceleration, providing a trade-off between resolution as runtime as vector size grows.

Because the HLL can be used to estimate intersections and unions directly, it can be applied to readily estimate not just Jaccard coefficient but containment ($|A \cap B|/|A|$) or overlap ($|A \cap B|/\min(|A|, |B|)$) coefficients.

The Dashing software also supports several features not supported by Mash or BinDash, including spaced seeds, PHYLIP-based output format, TSV, binary output, asymmetric distances, and a hash-set-based mode that can calculate exact Jaccard coefficients (as we did in one of our experiments) at the cost of memory footprint. Further, Dashing contains its own implementation of MinHash and b-Bit, and so is a flexible tool for future situations where a combination of approaches is warranted.

HLL also comes with drawbacks. As shown in Figure 2.2 and Table 2.1, Dashing is slower than BinDash at distance calculations. This is expected;

the b-bit Minwise Hashing approach consists primarily of comparisons of bit-packed suffixes of minimizers, which can be effectively vectorized. By contrast, the distance calculation between two HLL sketches is relatively expensive, requiring exponentiations, divisions, harmonic means, and — for the MLE-based methods — iterative procedures for finding roots of functions. The trade-off between accuracy and computational cost is underlined by Ertl's Joint MLE [21] method, which is both the slowest (even compared to MinHash) but the most accurate of the HLL-based methods. 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 [6].

Past efforts have considered how to extend MinHash to include information about 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 [26] 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 [27], and SuperMinHash [15]. In the future it will be important to seek similar multiplicity-preserving extensions —

and related extensions like *tf-idf* weighting [4, 28] — 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 observed that Bloom filters have superior accuracy when the bitvector is large enough to simulate linear counting, we could potentially populate the auxiliary filter with the input items (or a sample thereof) and recover some of the accuracy advantage enjoyed by Bloom filters.

While HLL was used by the KrakenUniq [18] tool for metagenomics read classification, KrakenUniq's implementation allows for a sparse representation of the registers, with 0-count registers omitted and non-0-count registers stored in a sparse array. Sparsity is a reasonable assumption in KrakenUniq, since some taxa have few associated k-mers due to relatedness of the genomes at the leaves. The sparsity assumption is less valid in Dashing's typical usage scenarios, though it can be valid if one input set has few elements compared to the number of HLL registers. In the future it will be important to investigate whether Dashing can be extended to exploit sparsity where it exists.

Though we compared to Mash and BinDash here, an alternative approach is used by the Kmer-db software [29]. Kmer-db's data structure captures the k-mer content of many input datasets at once. The underlying data structure

is a compressed bit matrix with bits indicating membership relationships between k-mers (rows) and input datasets (columns). Once a matrix is built, a second phase can perform individual or all-pairwise distance calculations over the samples. Since distinct k-mers are represented explicitly — which can take considerable space — the tool gives the option of subsampling the input k-mers using a MinHash-based method.

HLL's accuracy even when using a small sketch makes it appropriate 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.

2.5 Methods

2.5.1 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 possibly incrementing a counter with exponentially decaying probability. The probability is typically halved after each increment, so the counter approximates 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 MinHash — allows us to store more summaries total and, after averaging, achieve a better estimate.

The HLL combines many such counters into one sketch using 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 update the register to equal either its current value or the leading zero count (LZC) of suffix q, whichever is greater (Figure 2.3a). Note that the LZC of a bit string x of length q is related to $\log_2(x)$:

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

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 2.3b). We can combine estimates across registers by taking their harmonic mean and applying a correction factor, as detailed below. The estimator has a standard error of $\frac{1.03896}{\sqrt{m}}$ [17].

While the HLL is conceptually distinct from MinHash sketches and Bloom filters, it is related to both. Informally, an HLL modified so that the summary stored in each register is a simple minimum (without the $\log_2$) is similar to a MinHash sketch. Similarly, a Bloom filter with a single hash function and 2^x bits is essentially an HLL with an x-bit hash prefix and with registers consisting of a single bit each.

2.5.2 Estimation methods

The original HLL cardinality estimation method [17] combines the register-level estimates by taking a corrected harmonic mean:

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

Where α_m is a correction factor equal to $\frac{1}{2\ln 2}$ and M_j is 1 + the maximum LZC stored in register j. But the estimator's accuracy suffers at low and high extremes of cardinality. This has spurred various refinements starting with the original HLL publication [17], 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 [21]. The "improved estimator" uses the assumptions that (a) the hash function produces uniformly distributed outputs, and (b) register values are independent. It then models register count as a Poisson random variable. 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. Ertl suggests using the secant method, which uses inexpensive instructions and avoids derivative calculations. We follow this

suggestion in Dashing. Ertl also argues that the MLE generally converges in a small number of steps; we confirm that our implementation converges in at most 3 steps in every case we have tested.

Ertl's Joint MLE method, unlike those described so far, can directly estimate the cardinality of set intersections. We say "directly" to contrast it with methods that use the inclusion-exclusion principle to estimate intersection cardinality indirectly via cardinalities of sets (Figure 2.3c) and their unions (Figure 2.3d). The JMLE method again adopts the Poisson model but 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 the Poisson parameters for the three components. The procedure operates on a set of tallies of how often registers having a certain value in A are less than, equal to, or greater than their counterparts in B (and vice versa) (Figure 2.3e).

As discussed in Results, the JMLE as implemented in Dashing is substantially slower than MLE. This is 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. This tallying work grows linearly with the sketch size. 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.

2.5.3 Optimizing speed

Dashing takes advantage of the fine-grained parallelism inherent in HLLs. Union and intersection cardinalities are the key components of similarity measures like the Jaccard coefficient. For two HLLs having the same number of registers and the same hash function, a sketch of their union is simply the element-wise maximum of their registers. Thus, one fundamental need is to perform element-wise maximum over long vectors of 8-bit registers. Finding the cardinatlity of an individual set — or of the intersection of two sets using the JMLE — requires tallying statistics over the register array. Thus, another need is to perform tallies (e.g. counting the registers having a particular value) over long vectors of 8-bit registers.

For set unions, Dashing's inner loops use Single-Instruction Multiple Data (SIMD) instructions, which are capable of performing fast arithmetic and bitwise operations on vectors of many adjacent operands. These vectors are substantially wider (up to 512 bits) than the typical 32-bit or 64-bit machine words used to store scalar operands. Speedups can be attained by converting important loops to use only or mostly SIMD instructions and to avoid loops with scalar instructions. The more operands per SIMD vector, the greater the potential benefit [24]. The ideal would be to use vectors consisting of 8-bit operands, since this matches the HLL register width. While past iterations of the SIMD instruction set operated on 128- and 256-bit vectors of 8-bit operands, only with the recent introduction of Intel's AVX-512BW instruction

set did it become possible to operate on 512-bit vectors of 8-bit operands. We created AVX-512BW versions of inner set-union loops and confirmed that these deliver the greatest distance-estimation throughput, providing 20% speed boost compared to loops based on the older SSE2 SIMD instruction set. For compatibility with older systems, Dashing supports older SIMD instruction sets back to SSE2.

The process of tallying statistics for set cardinalities and set intersection cardinalities is harder to SIMD-ize in this way. Dashing uses manual loop unrolling to speed up these inner loops, but no SIMD instructions. A question for future work is whether these loops can be rewritten using, for example, a combination of SIMD gather, increment and scatter operations.

Dashing also supports use of many simultaneous threads of execution using the OpenMP v4.5 library. The `dashing sketch` function is parallelized across input files, with distinct threads reading, sketching, and writing sketches for distinct inputs. In `dashing dist`, threads work in parallel on elements in a row of the upper-triangular matrix while a distinct thread writes out the results. To minimize the overhead associated with global memory-allocation locks, each thread allocates from a private memory buffer. The all-pairs distance calculation uses multiple output buffers and asynchronous I/O to avoid blocking and output-lock contention.

Another concern is load balance; having many simultaneous threads is beneficial only if we can avoid "straggler" threads that run long after the others have finished. We eliminated an important source of stragglers by performing an up-front large-to-small ordering of the inputs to be sketched.

This minimizes the chance that the thread with the largest genome will still be working when others are finishing.

2.5.4 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 [2], Dashing uses an auxiliary data structure at sketching time to remove infrequent k-mers that are 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 (CMS) [35], and only if the estimated count for the k-mer is sufficiently high is it added to the HLL. The CMS can provide count estimates using an amount of space that grows sublinearly with the number of items.

2.5.5 Hash function

We compared clhash [36], Murmur3's finalizer [37], and the Wang hash [38] across a set of synthetic Jaccard index estimates, and found that Wang's had the lowest error (8.20×10^{-3}) and bias (-2.14×10^{-4}), compared to 8.27×10^{-3} and 2.30×10^{-4} for Murmur3 and 8.21×10^{-3} and $-2.66e \times 10^{-4}$ 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.

2.6 Availability of Data and Materials

- Dashing source code is available under the open source GPLv3 license [22]
- The particular version of Dashing evaluated here is included in this permanent archive: [39]
- Scripts and code used to perform the experiments described in this study are available under the open source GPLv3 license [40].
- The particular version of the scripts and code used to perform the experiments described in this study is included in this permanent archive [39].
- 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`.

Figures

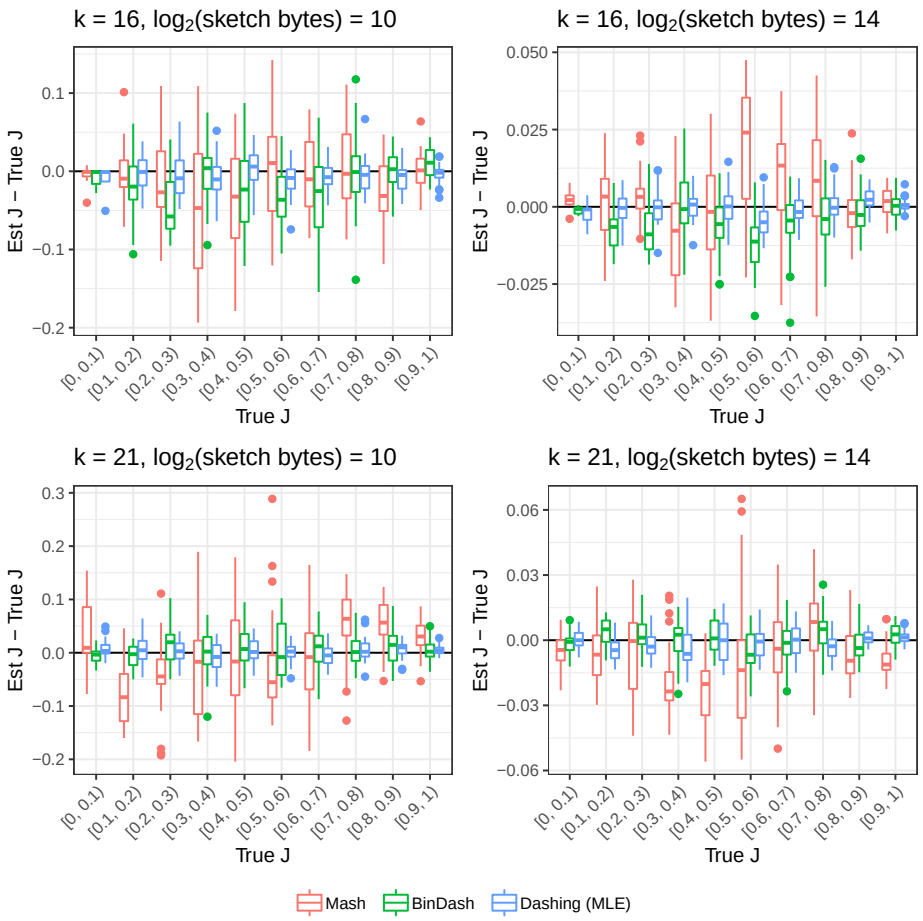


Figure 2.1: Estimated versus true Jaccard coefficients (Js) for various methods across a range of true J. Each point is one pair from an overall set of 400 pairs of genomes, selected to evenly cover the range of true Js.

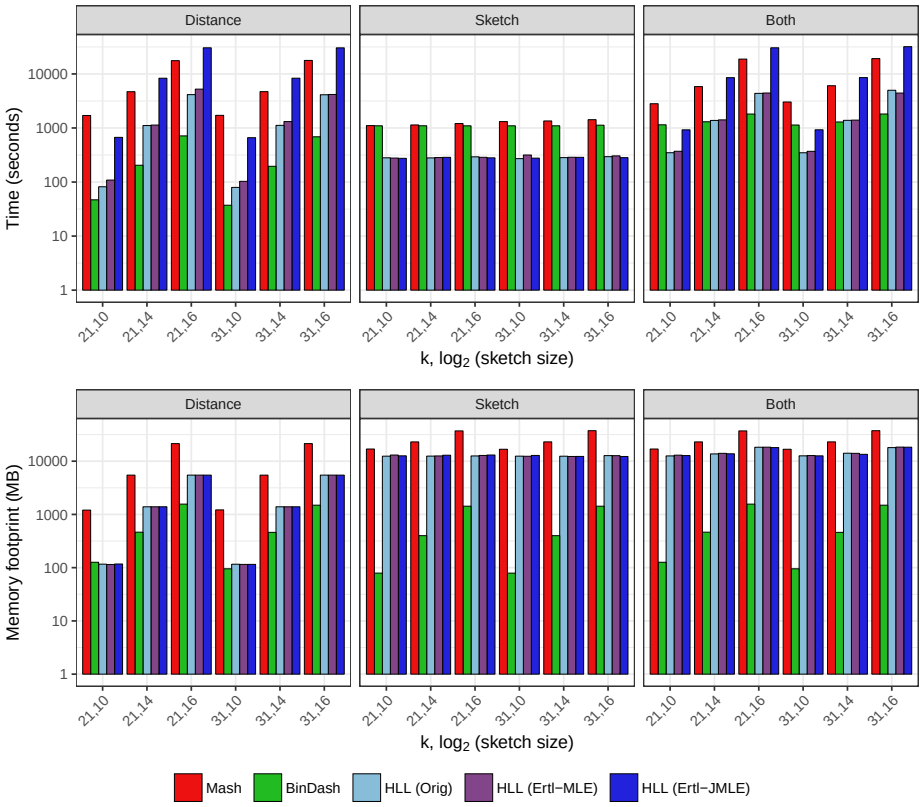


Figure 2.2: Computational efficiency of Mash, BinDash and Dashing. Results for $k = 21$, $k = 31$ and sketches of size 2^{10} (1KB), 2^{12} (4KB), 2^{14} (16KB) and 2^{16} (64KB). "Both" results obtained either by using a combined Sketch+Distance mode (for Dashing) or by combining results from separate sketching and distance-calculation 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.

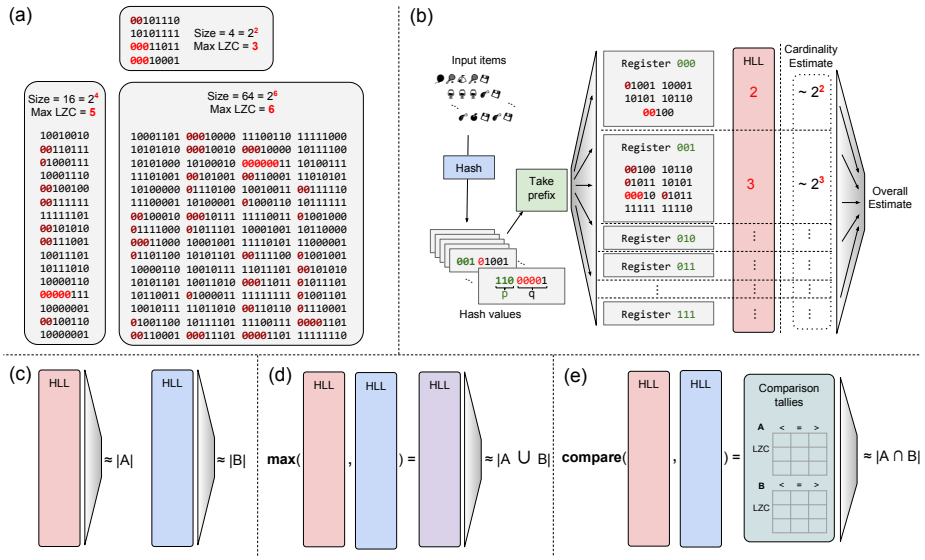


Figure 2.3: (a) Relationship between maximum leading zero count (Max LZC) and set size for three randomly-generated sets of 8-bit numbers. The Max LZC roughly estimates the $\log_2$ of the set size, though with high variance; here, two of three estimates are off by 2-fold. (b) Schematic of HyperLogLog sketch. Input items are hashed and hash value is partitioned into prefix p and suffix q. p indexes into the array of HLL registers. A register contains the maximum leading zero count among all suffixes q that mapped there. Register-level estimates are then combined to obtain an overall cardinality estimate. (c) Estimating cardinalities of sets A and B, and (d) estimating the cardinality of their union. For intersection cardinalities using inclusion-exclusion principle, estimated set and union cardinalities are combined. (e) Direct estimation of intersection cardinality with Ertl's JMLE.

References

[1] D. N. Baker and B. Langmead. "Dashing: fast and accurate genomic distances with HyperLogLog". In: *Genome Biol* 20.1 (2019), p. 265.

[2] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. "Mash: fast genome and metagenome distance estimation using MinHash". In: *Genome Biol.* 17.1 (2016), p. 132.

[3] L. Schaeffer, H. Pimentel, N. Bray, P. Melsted, and L. Pachter. "Pseudoalignment for metagenomic read assignment". In: *Bioinformatics* 33.14 (2017), pp. 2082–2088.

[4] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy. "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation". In: *Genome Res.* 27.5 (2017), pp. 722–736.

[5] K. Berlin, S. Koren, C. S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". In: *Nat. Biotechnol.* 33.6 (2015), pp. 623–630.

[6] C. Jain, S. Koren, A. Dilthey, A. M. Phillippy, and S. Aluru. "A fast adaptive algorithm for computing whole-genome homology maps". In: *Bioinformatics* 34.17 (2018), pp. i748–i756.

[7] C. Jain, L. M. Rodriguez-R, A. M. Phillippy, K. T. Konstantinidis, and S. Aluru. "High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries". In: *Nat Commun* 9.1 (2018), p. 5114.

[8] Andrei Z Broder. "On the resemblance and containment of documents". In: *Compression and complexity of sequences 1997. proceedings*. IEEE. 1997, pp. 21–29.

[9] D. E. Wood and S. L. Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome Biol.* 15.3 (2014), R46.

[10] H. Li. "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences". In: *Bioinformatics* 32.14 (2016), pp. 2103–2110.

[11] J. Buhler. "Efficient large-scale sequence comparison by locality-sensitive hashing". In: *Bioinformatics* 17.5 (2001), pp. 419–428.

[12] Y. Luo, Y. W. Yu, J. Zeng, B. Berger, and J. Peng. "Metagenomic binning through low-density hashing". In: *Bioinformatics* (2018).

[13] XiaoFei Zhao. "BinDash, software for fast genome distance estimation on a typical personal laptop". In: *Bioinformatics* (2018), bty651.

[14] Y. William Yu and Griffin Weber. "HyperMinHash: Jaccard index sketching in LogLog space". In: *CoRR* abs/1710.08436 (2017). arXiv: 1710.08436. URL: http://arxiv.org/abs/1710.08436.

[15] Otmar Ertl. "SuperMinHash - A New Minwise Hashing Algorithm for Jaccard Similarity Estimation". In: *CoRR* abs/1706.05698 (2017). arXiv: 1706.05698. URL: http://arxiv.org/abs/1706.05698.

[16] D. Koslicki and H. Zabeti. "Improving Min Hash via the Containment Index with applications to Metagenomic Analysis". In: *bioRxiv* (2017). DOI: 10.1101/184150.

[17] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *AofA: Analysis of Algorithms*. Ed. by Philippe Jacquet. Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). DMTCS Proceedings. Juan les Pins, France: Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156. URL: https://hal.inria.fr/hal-00406166.

[18] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. "KrakenUniq: confident and fast metagenomics classification using unique k-mer counts". In: *Genome Biol.* 19.1 (2018), p. 198.

[19] M. R. Crusoe, H. F. Alameldin, S. Awad, E. Boucher, A. Caldwell, R. Cartwright, A. Charbonneau, B. Constantinides, G. Edvenson, and S. et al Fay. "The khmer software package: enabling efficient nucleotide sequence analysis". In: *F1000Res* 4 (2015), p. 900.

[20] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. "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. New Orleans, Louisana: IEEE Press, 2014, pp. 437–448. ISBN: 978-1-4799-5500-8.

[21] Otmar Ertl. "New cardinality estimation algorithms for HyperLogLog sketches". In: *CoRR* abs/1702.01284 (2017). arXiv: 1702.01284. URL: http://arxiv.org/abs/1702.01284.

[22] Daniel N Baker. *Dashing: Fast and accurate genomic distances using HyperLogLog*. 2019. URL: https://github.com/dnbaker/dashing.

[23] C Titus Brown and Luiz Irber. "sourmash: a library for MinHash sketching of DNA". In: *The Journal of Open Source Software* 1.5 (2016).

[24] R. Rahn, S. Budach, P. Costanza, M. Ehrhardt, J. Hancox, and K. Reinert. "Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading". In: *Bioinformatics* 34.20 (2018), pp. 3437–3445.

[25] N. A. O'Leary, M. W. Wright, J. R. Brister, S. Ciufo, D. Haddad, R. McVeigh, B. Rajput, B. Robbertse, B. Smith-White, and D. et al Ako-Adjei. "Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation". In: *Nucleic Acids Res.* 44.D1 (2016), pp. D733–745.

[26] Roderick Bovee and Nick Greenfield. "Finch: a tool adding dynamic abundance filtering to genomic MinHashing". In: *Journal of Open Source Software* 3(22) (2018).

[27] Otmar Ertl. "BagMinHash - Minwise Hashing Algorithm for Weighted Sets". In: *CoRR* abs/1802.03914 (2018). arXiv: 1802.03914. URL: http://arxiv.org/abs/1802.03914.

[28] Ondrej Chum, James Philbin, Andrew Zisserman, et al. "Near Duplicate Image Detection: min-Hash and tf-idf Weighting." In: *BMVC*. Vol. 810. 2008, pp. 812–815.

[29] S. Deorowicz, A. Gudys, M. Dlugosz, M. Kokot, and A. Danek. "Kmer-db: instant evolutionary distance estimation". In: *Bioinformatics* 35.1 (2019), pp. 133–136.

[30] B. Solomon and C. Kingsford. "Fast search of thousands of short-read sequencing experiments". In: *Nat. Biotechnol.* 34.3 (2016), pp. 300–302.

[31] Robert Morris. "Counting Large Numbers of Events in Small Registers". In: *Commun. ACM* 21.10 (1978), 840–842. ISSN: 0001-0782. DOI: 10.1145/359619.359627. URL: https://doi.org/10.1145/359619.359627.

[32] Philippe Flajolet. "Approximate counting: A detailed analysis". In: *BIT Numerical Mathematics* 25.1 (1985), pp. 113–134. ISSN: 1572-9125.

[33] Philippe Flajolet and G Nigel Martin. "Probabilistic counting algorithms for data base applications". In: *Journal of computer and system sciences* 31.2 (1985), pp. 182–209.

[34] Stefan Heule, Marc Nunkesser, and Alexander Hall. "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. Genoa, Italy: ACM, 2013, pp. 683–692. ISBN: 978-1-4503-1597-5.

[35] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58 –75. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0196677403001913.

[36] Daniel Lemire and Owen Kaser. "Faster 64-bit universal hashing using carry-less multiplications". In: *CoRR* abs/1503.03465 (2015). arXiv: 1503.03465. URL: http://arxiv.org/abs/1503.03465.

[37] A Appleby. *MurmurHash3*. 2011. URL: https://github.com/aappleby/smhasher.

[38] Thomas Wang. *Integer Hash Function*. http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm. 1997.

[39] Daniel N Baker and Ben Langmead. *Dashing software used in manuscript experiments*. 2019. DOI: 10.5281/zenodo.3402234. URL: https://zenodo.org/record/3402234.

[40] Daniel N Baker and Ben Langmead. *Dashing software used in manuscript experiments*. 2019. URL: https://github.com/langmead-lab/dashing-experiments.

Chapter 3

Dashing 2: fast and flexible sketching with multiplicities and Locality-Sensitive Hashing filtering

3.1 Context

This thesis chapter is comprised of the Dashing 2 paper, which is in preparation and will be submitted for publication in the near future. In it, we correct for varying weaknesses of the original Dashing method by bringing in recent theory [1, 2, 3], accounting for *k*-mer multiplicities, and eliminated wasted space. We show that the logarithmic truncation and fast similarity estimators used in the SetSketch paper also apply to weighted MinHash variations, yielding fast and accurate sketch comparisons.

Finally, to broaden its applicability, we leverage the locality-sensitive hashing (LSH) property of the MinHash registers to build an index facilitating near-linear time analysis. Using this index, we can very rapidly de-duplicate, and generate K-Nearest Neighbor- and Similarity Thresholded Neighbor-graphs for various genomic collections.

3.2 Abstract

Sketching algorithms, including MinHash and HyperLogLog have become crucial building blocks for scalable computational biology solutions, from de-duplication, clustering, indexing and summarization of genomic collections.

While prior tools, incl. Mash [4] and Dashing [5], could efficiently scale to hundreds of thousands of entities, many important biological datasets number in the millions (UniProt) or billions (BFD), at which point quadratic all-pairs comparisons become infeasible in both space and time.

Addressing this, we use the Locality-Sensitive Hashing (LSH) properties of MinHash to build LSH tables supporting nearest-neighbor queries. Using this table, we can accurately generate "top-k" nearest neighbor tables, perform clustering in linear time, and accelerate sparse Jaccard-thresholded results, making it feasible to scale into the millions and beyond.

We have also improved the accuracy and comparison speed of our sketches. When using the HyperLogLog as the core structure for Dashing 1, there was the matter of wasted space, wherein only 6 bits out of an 8-bit register were used in order to maintain parallelized comparisons. Further, with such small registers, the HyperLogLog could yield small false positive estimates due to register collisions.

With the theoretical developments with the SetSketch, varying the base of the sketch became formalized, allowing us to tailor the log-base for higher-accuracy with faster sketch comparisons, use registers as small as 4 bits or as large as 64-bits. Further, we explore the tradeoff between space and accuracy

and demonstrate that for many cases, even 4-bit registers are sufficient.

Dashing 2 [**dashing2-src**] also supports multiplicity-aware sketching and comparisons, using BagMinHash to compute multiset Jaccard sketches and ProbMinHash to compute sketches over discrete probability distributions. These additional modes extends Dashing 2's relevance to datasets sensitive to multiplicities such as RNA-sequencing, splicing data, and coverage vectors.

Dashing 2 also adds support for sketching additional file types, including the BED interval format, the BigWig coverage vector format, and a more general compressed sparse row (CSR)-format weighted set representation.

Dashing 2 is open-source and freely available at https://github.com/dnbaker/dashing2.

3.3 Background

MinHash [6, 4] and the HyperLogLog [5, 7] are core utilities for efficiently supporting similarity search and nearest-neighbor functionality from huge sequencing databases. They have applications in phylogeny reconstruction [4, 8], deduplication [7], and clustering [9]. While existing tools, such as Mash, Dashing, and Bindash [10] can quickly cluster 10s of thousands of genomes, many relevant biological datasets are much larger, reaching millions or billions of sequences. Dashing can be a suitable substitute for Average Nucleotide Identity (ANI) [11]. Typical pairwise comparison methods, with their quadratic complexity, cannot scale to these problems. We apply the Locality-Sensitive Hashing property of MinHash sketches to build indexes supporting efficient generation of nearest-neighbor lists in near-linear time.

Further, these approaches typically fail to adapt to the weighted Jaccard case, where *k*-mers are treated as multisets instead of sets, making them less useful in situations where weights factor into similarity, such as RNA-sequencing or epigenetic modifications.

In this work, we move beyond the HLL-based method in Dashing, and adopt both a new sketching approach and data structure – the SetSketch –as well as a locality-sensitive hashing scheme that can drastically reduce the number of dataset pairs we compare. The SetSketch solidifies the theory around choosing a flexible log base (and offset), which in turn fixes a major issue in Dashing 1, which was its unused space per register. It also brings with it a simple, accurate similarity estimation method which allows us to substantially improve comparison speed. Further, its similarity to weight-aware sketching methods, such as ProbMinHash and BagMinHash [2, 3], this allows us support weights and adapt the fast comparison algorithm to the weighted case.

We find that the SetSketch is the most accurate for calculating the standard (unweighted) Jaccard coefficient in a variety of scenarios, outperforming standard MinHash, HLL, and b-bit Minhash. Further, we demonstrate that sketching methods accounting for similarity provide superior average nucleotide identity (ANI) estimates over weighted *k*-mer methods.

Finally, we use a locality-sensitive hashing approach to greatly narrow the space of all-pairs comparisons we need to do when providing sparsified analysis. This builds on the statistical independence of registers in the SetSketch and its weighted relatives, and allows us to very rapidly group items at

different levels of similarity. We evaluate its accuracy, in relation to exhaustive all-pairs comparisons, and demonstrate immense runtime improvements with unchanged results. And we further demonstrate our method's ability to scale to millions of items, which would be out of the reach of competing methods without an LSH index.

3.4 Results

We did three kinds of experiments. (a) Experiments using a pre-collected a set of genome pairs, selected to cover a range of true Jaccard similarities, and mainly for comparison the accuracy of different methods, (b) Experiments using a large number of assembled genomes, mainly for comparison the computational performance and scalability of different methods, and (c) Experiments using an LSH index to generate approximate K-Nearest Neighbor graphs and measure their accuracy.

Software for reproducing these results can be found at Dashing2-Experiments.

3.4.1 Sketching Improvements

3.4.1.1 Use of SetSketch

We previously implemented the HyperLogLog sketch in the Dashing 1 tool, which was extremely efficient at sketching and more accurate in estimates than Mash sketches of the same sizes. However, there were some weaknesses.

First, because the expected variance of the estimator was fixed with respect to the cardinality of the data ($\mathcal{O}(\frac{1}{\sqrt{\#\text{Registers}}})$), this led to a tendency to return unreliable estimates of low similarity. This corresponded to Bindash's slightly

higher accuracy at low similarities, while Dashing 1's method was more accurate at higher similarities.

Second, because the log base used in the HyperLogLog is fixed at 2, our previous approach was wasting the upper 2-3 bits of each one-byte register. Using a log base of 2 and a 64-bit hash, the maximum leading zero count of a table with p signature bits and table size 2^p is '64 - p' with a maximum number of bits $\lceil \log_2 64 - p \rceil$. For typical parameterizations with fewer than 2^32 registers, this requires 6 bits per signature. All register space greater than $64 - p$ was unused, as packing into bytes allowed us to exploit SIMD computation. Relatedly, we also could not modify the log-base to fit a specific number of bits. Wanting to use all available space, we expected we could achieve higher accuracy with fixed-size registers, and that there would be a trade-off between more registers and more precise registers.

And finally, because both the original HyperLogLog estimation algorithm [12] and the Ertl enhanced algorithms [13] required a sum of register counts, while vectorized computation provided an importance performance improvement, there was a serial final summation loop in that algorithm, limiting how much it could be vectorized. Additionally, several HyperLogLog algorithms retained some special handling for extreme cases.

The recent SetSketch [1] method provided us the mathematical framework to address these concerns and expand our applicability. With the SetSketch, this allowed us to change the log-base arbitrarily, which we then mapped to registers of 4, 8, 16, 32, and 64 bits. We also by default lazily truncate the signatures, which allows us to minimize the number of logarithms calculated

and match the parameters to the data to maximally exploit register value space. The SetSketch also permits a fast, simple estimator which requires on the counts of $>$ and $<$ between registers, for which we developed fast SIMD implementations for this estimator which are substantially faster than the HyperLogLog algorithms. In addition to its performance advantages, it is also consistently more accurate.

3.4.1.2 Use of locality-sensitive hashing

Lastly, previous methods using sketches of k-mer sets primarily suffered from a quadratic barrier - exhaustive pairwise comparisons were $\mathcal{O}(n^2)$. To allow our methods to scale, we use an inverted index, where combinations of input MinHash registers are used as keys mapping to entities containing those values.

Because the SetSketch registers are independent, we can sample from them with replacement to generate stronger MinHash keys with collision probability J^p, where p is the number of MinHash registers per combined hash register. We build multiple tables and query from most specific to most sensitive.

3.4.1.3 Practical implementation

We introduced some implementation details which improved the algorithm and expanded its usefulness. First, we delay the logarithm by dfault, maintaining the full floating-point random variates during sketching. Once complete, we can then tune the log-base to the maximum that will yield all-valid register values, allowing use to maintain as much precision as possible in a fixed

number of bits. We call this the Continuous SetSketch. (If expected cardinalities are known in advance, one can specify the parameters a and b from the SetSketch to reduce peak memory requirements, at the expense of slightly reduced precision.)

Second, we introduce a stochastically-averaged SetSketch we call the One-Permutation SetSketch. The HyperLogLog sketching update loop is very fast, as it requires only one register update. While the SetSketch adds each item to all buckets with an increasing exponential draw that can stop early, the HyperLogLog uses stochastic averaging for constant-time updates, which can be substantially faster for larger sketches. We use this technique in the Continuous SetSketch. To maintain accuracy for sketches with empty registers, we use densification [14] of the One-Permutation Continuous SetSketch [1]. In our experiments, we demonstrate that the one-permutation setsketch has effectively identical accuracy as the full setsketch, while being significantly more efficient to produce.

Third, when building the Continuous SetSketch without stochastic averaging, the log calculation is a significant contributor to the runtime. We found that many of these exponential random draws are above the current sketch maximum. This led us to realize that we could only perform log if an item was likely to be inserted into the sketch, we could significantly speed up sketching. We use the fast approximate-log trick, which relies on the floating-point number's integral representation's similarity to its $\log_2$. C/C++ code is provided in figures fig. 3.7 and **??**. It may overestimate, but never by more than a factor of 1.42. By dividing the estimate by this number, we can reject most candidates

without the log function call, which significantly accelerated sketching.

We also wanted to be able to perform multiplicity-aware sketching, to improve our applicability to applications sensitive to counts, such as coverage vectors, expression counts, and weighted interval sets. We applied Ertl's recent work with ProbMinHash and BagMinHash [2, 3] to generated weighted sketches. These sketching algorithms require a mapping from items to counts to generate iterates. For large k-mer sets, the size of the structure this data requires can be extremely large. We found that using feature hashing [15], equivalent to a single-row Count-Min Sketch [16], gave us a way to quickly estimate count vectors without risking out-of-memory errors. This approach only can only increase similarity estimates, which leads to small but consistent over-estimation.

We also applied the logarithmic-truncation technique from the SetSketch to the weighted cases. Since both of the weighted sketching approaches are derived from exponential draws, we can apply the same tailored logarithmic truncation and fast SIMD $>/<$-count based comparison for accelerated similarity estimates.

3.4.1.4 Rare Event Filtering

To support filtering of rare elements when sketching read sets, we provide two mechanisms. First, Dashing 2 can receive a parameter "–count-threshold <int>" which causes it to only sketch items above a minimum observed frequency. For this approach, *k*-mers whose abundance is below this threshold but whose hashes could update the sketch are maintained in a temporary count structure.

This method is inspired by that used in the Mash method/paper [4]. Once the item's frequency passes this threshold, it is inserted into the sketch, and this count structure is pruned periodically to account for updates to the sketch's inclusion threshold.

Second, we also provide a "–downsample-frac <float> " argument, which randomly discards k-mers with probability $1 -$ <float>. If typical depth of coverage is > 10, then selecting a value such as $\frac{1}{10}$ would discard the majority of k-mers occurring once, while tending to preserve most higher-frequency items. This fails with some probability, but it comes at no space and effectively no runtime cost.

3.4.2 Scaling to millions: Sparse Similarity and Applications

In addition to improving the speed and accuracy of our comparisons, the exhaustive comparison approach taken by Dashing, Mash, and BinDash came with downsides. While Mash and BinDash have methods filtering by similarity thresholds, they still suffer superlinear runtime with respect to the number of sequences by performing exhaustive comparisons.

Considering that MinHash is a locality-sensitive hash (LSH) hash function, we decided to build LSH indexes built over the sketches to faciliate several downstream applications in near-linear time.

This index generates near-neighbor candidates in order of decreasing similarity in linear time.

This technique combines multiple registers into fewer but stronger hash registers with greater discriminatory power. While a MinHash register is

identical between two sets with probability $J = \frac{\text{Intersection}}{\text{Union}}$, by performing this grouping approach, the probability of P independent registers matching is J^P. This allows us to "spread out" candidates with the exponent of Jaccard similarity.

We build a set of N LSH tables, each of which groups registers of size $\min 2^i, 2i$ into composite hashes. After building the tables, we begin querying from the most specific table (comprised of hashes from the largest groups of registers), to the least specific table (comprised of single MinHash registers) until we have reached the nuber of desired candidates or until we have queried all tables. This allows us to limit the number of queries required while both preferentially selecting nearer neighbors and while returning all entities matching even a single register if there are few enough near neighbors. Using this index to eliminate quadratic comparisons allows us to scale analysis to hundreds of thousands or millions of entities while maintaining accuracy.

First, we use this to generate fast K-Nearest Neighbor ("KNN") graphs. After building the LSH index, we can query it for all items to generate candidates, followed by using sketches among those candidates to select a final set of K-Nearest Neighbors. This is evaluated in 3.6.

We also provide thresholded result, which is similar in design. While querying the index, we maintain a heap of all neighbors with similarity above/distance below a given threshold.

And finally, we apply the CD-HIT [17] algorithm, similar to Linclust [9], building an LSH index over representative entities. We sort inputs by decreasing size, similar these other tools' sorting of proteins by decreasing length.

This allows us to quickly de-duplicate a collection of sequences by grouping sets separated by a similarity threshold.

3.4.3 Other Improvements: Exact mode, minimizers, & Alphabets

We also provide exact modes - both using sorted k-mer hash sets and k-mer count dictionaries. While these are substantially slower, they can be practically evaluated on selected subsets selected from nearest-neighbor lists. Our k-mer encoding also supports minimizers, which transduces the k-mer sets/dictionaries into minimizer sets, which can accelerate processing.

To reduce the likelihood of selecting low-complexity k-mers, we also provide minimizer weighting by Shannon entropy $H = -\sum_{c \in A} f_c \log f_c$, where f_c is the frequency of the character c. Instead of assigning $W_{k\text{-mer}} = \text{Hash}_{k\text{-mer}}$, as in standard minimizer selection, we assign ranks by $W^{\text{entropy}}_{k\text{-mer}} = \frac{\text{Hash}_{k\text{-mer}}}{H+\epsilon}$, using $\epsilon = 1e-4$ to avoid infinite values if H is 0. This decreases the likelihood that a low-complexity k-mer will be selected as a minimizer.

We use a sliding window technique to maintain the entropy sums with constant-time updates, irrespective of k-mer length. We accomplish this by decrementing the count for the character leaving the window and incrementing the count for the character entering the k-mer.

We use a technique known as Compensated Addition or Kahan Summation, developmented independently by Kahan [18] and Babuska [19], in the SetSketch algorithm to reduce errors in summation due to the floating-point

approximation. This comes at a the cost of some additional arithmetic operations, but better preserves the precision of the exponential draws. Notably, however, we only have to perform compensated addition for increments which pass our fastlog-prefilter.

3.4.3.1 Iteration Order

We also find that iteration order plays a substantial role in efficient calculation. In Dashing 2, we store all sketches in a matrix for locality, and we group computations during multithreading so that these methods tend to work with the same comparison data. This provides speed improvements of 50-250%. This also naturally comes with synchronization, as threads that get out of sync in advance of the other threads are held up by memory latency, and threads that are behind benefit from the pre-fetching performed by the threads which were ahead.

3.4.3.2 Memory Management

For exceptionally large data, we exploit file memory-mapping to lazily deposit sketches to disk. Since most of our access patters are sequential, this allows us to exceed random access memory (RAM) thresholds without significant performance penalties. Since these are represented as one large matrix, many memory accesses are shared across sketches.

If memory-mapping is insufficient, we can further extend the applicable scale by generated already-truncated signatures with user-set a, b parameterizations instead of tuning them to the data. This can increase computation costs slighty, but it reduces the signature matrix in proportion to the reduction

in register size. For instance, a byte-sized register yields a memory reduction factor of 8, and a nibble-sized register by 16. The nearest neighbor index is built on the reduced registers. Fortunately, the SetSketch paper provides guarantees on locale-sensitivity for these reduced representations.

3.4.3.3 Input Data Types

We support FASTA and FASTQ data formats, for the typical expected use cases. We also support interval sets and coverage vectors - specifically, BED files, and BigWigs. To expand the generality of our method, we also provide sketching methods working directly from compressed-sparse row (CSR) format sparse matrices.

3.4.4 Sketch accuracy

We wanted to investigate the accuracy of these sketches, both at estimating k-mer similarity and inferring average nucleotide identity. For this experiment, we selected a set of 51,200 pairs of genomes spanning the range of similarity from 0 to 1. We first computed all-pairs similarities using Dashing 2 with a sketch size of $65536 = 2^{16}$ registers with $k = 31$. Using this full distance matrix, we evenly partitioned the input similarity space into 2048 buckets and selected the first 25 pairs of genomes in each bucket.

After selecting these pairs, we computed the exact weighted and unweighted Jaccard similarities between all these pairs of genomes with Dashing 2, estimated the ANI between the genomes with fastANI [20], and compared the using various parameterizations of Dashing, Mash, BinDash, and Dashing

k	kbits	Mash	SS8	Dash1	SS1
21	32	1.36	1.33	1.31	0.922
	64	1.32	1.18	1.13	1.01
	128	1.94	0.991	1.00	1.00
	256	1.53	0.933	1.08	0.960
	512	1.24	1.02	1.05	0.959
31	32	1.25	1.37	0.874	0.689
	64	1.00	0.930	0.820	0.727
	128	0.663	0.752	0.689	0.697
	256	0.581	0.703	0.681	0.656
	512	0.556	0.719	0.656	0.645

Table 3.1: Sum of squared error between estimated and true Jaccard coefficients for several methods. Bright red indicates the lowest error in each row. Dark red indicates second-lowest error.

2. This allowed us to compare both direct similarity estimation and to evaluate whether accounting for k-mer multiplicities can improve ANI estimation.

The code for generating these results can be found in the Dashing2 experiments repository [21] in the `jirange` directory, and the final data is in the `jirange/jirsplit/` directory, split due to GitHub file-size limits.

First, we wanted to see if the SetSketch could be more accurate than the HyperLogLog. We can see in Table 3.1 for the JI Range Experiment that SetSketch with byte-sized registers (SS1) improves on the accuracy of Dashing, which itself is consistently more accurate than Mash. The 8-byte register SetSketch method (SS8) is also comparable to Mash, which uses 8-byte registers. For very large sketches (> 256 Kb/sketch), Mash seems to improve relative to SS8, but only marginally.

We also wanted to see how well these methods matched ANI estimation. Using the fastANI method as an efficient ANI estimator, we compared Mash, Dashing 1, Dashing 2, and a weighted version based on ProbMinHash [2]

which we call D2W in terms of accuracy in estimating ANI. The results in the Table 3.2 of sum of squared error of ANI estimation show that while both Dashing and Dashing 2 improve on Mash in ANi estimation, incorporating weights via ProbMinHash yields some substantial improvements over its unweighted equivalents. This higher accuracy, in conjunction with efficient construction, ability to scale to many genomes, and the large runtime improvements in comparisons afforded by our methods, makes Dashing 2 a very efficient and scalable method for similarity search across sequence collections.

For shorter k-mers (21 and 31), exact multiset Jaccard calculations are the most accurate for ANI estimation. With long k-mers (71), we find that weighted sketching is even more accurate than exact multiset Jaccard. This may be because 71 is too specific and the bias introduced by feature hashing corrects for this over-precision. Complete results, including results for $k = 71$, stochastically-averaged and standard setsketch, BagMinHash, ProbMinHash, and varied feature hashing parameterizations are available in the Dashing 2 experiment repository [21].

3.4.5 Performance: All-Pairs

We performed exhaustive pairwise comparisons across 118,467 genomes for a range of tools, using 96 threads in Figure ??. Separately, we performed the same experiment with a single-thread on all available fungal and archaeal genomes (307 and 665, respectively) for a total of 974, with results in Figure ??. For our invocation of Dashing 2 in this experiment, we used the similarity

k	kbits	Mash	Dashing1	D2	D2W
21	32	90.4	51.9	27.3	14.4
	64	46.9	17.8	19.2	14.4
	128	31.6	9.04	24.7	14.5
	256	25.9	7.11	23.5	14.6
	512	18.5	9.23	15.6	14.6
31	32	129.	102.	37.9	28.6
	64	55.5	115.	32.0	28.5
	128	44.0	59.7	28.5	28.4
	256	37.5	43.1	24.8	28.5
	512	33.4	29.7	26.0	28.5

Table 3.2: Sum of squared error between estimated and true ANI for several methods. Bright red indicates the lowest error in each row. Dark red indicates second-lowest error.

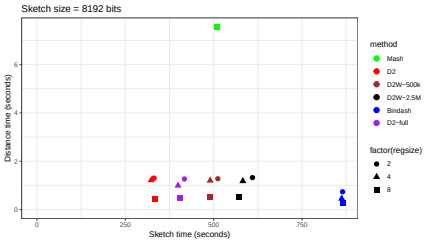


Figure 3.1: All-pairs comparisons across 974 fungal and archaeal genomes in RefSeq, with sketch size = 8192 bytes

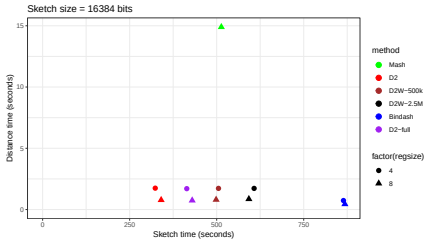


Figure 3.2: All-pairs comparisons across 974 fungal and archaeal genomes in RefSeq, with sketch size = 16384 bytes

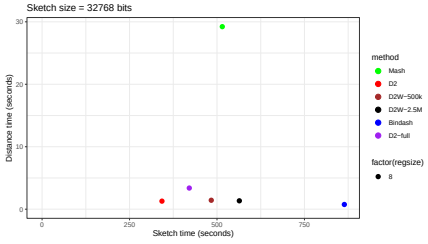
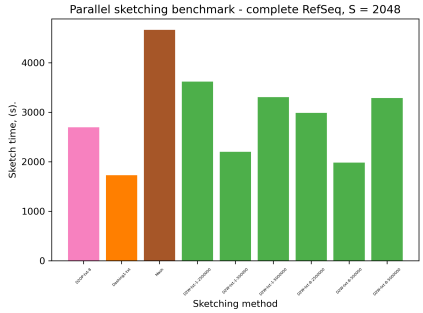


Figure 3.3: All-pairs comparisons across 974 fungal and archaeal genomes in RefSeq, with sketch size = 32768 bytes



t

Figure 3.4: Sketch time for 118,467 reference genomes from RefSeq, S = 2048 with 96 threads

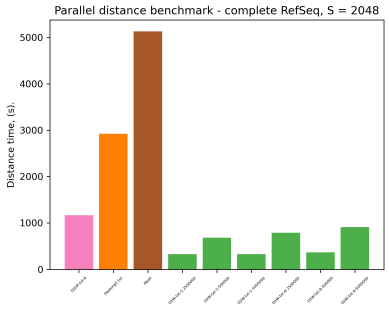


Figure 3.5: Comparison time for all-pairs comparisons across 118,467 reference genomes from RefSeq, S = 2048 with 96 threads

estimation algorithm from the SetSketch paper.[2]

Consistently over these results, we see that Dashing 2's sketching time is faster than Mash's while being slightly slower than Dashing 1.

However, the biggest advantage is that the comparison speed is 2-70x that of Mash and Dashing 1. This comes from the simplicity of the SetSketch comparison method and our fast handwritten SIMD distance code. This gives us massive speed improvements over prior methods, whether using a single thread of execution or many.

3.4.6 All-pairs comparisons using LSH

We evaluated our method over the the UniProt-SwissProt collection of 565,254 proteins reduced to 12−mer sketches of size 256 using a 6-letter reduced amino acid alphabet for long-range homology [22]. We then compared exhaustive top-256, all-pairs sketch-based calculations against those generated by our

[2]In our experiments, the runtime performance of this SetSketch comparison algorithm was equivalent to that of b-bit MinHash signatures, and we therefore omit those results.

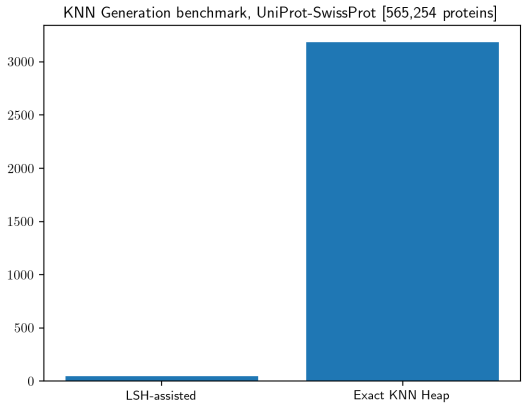


Figure 3.6: Benchmark, comparing LSH-assisted KNN graph generation time against exhaustive all-pairs comparisons with a heap.

LSH-assisted approach. Using this, we achieved 100% recall, precision and F_1. Generating this list took less than a minute, compared to an hour for exhaustive calculations. This is 159,755,759,631 pairwise comparisons - 42 times as many as in the exhaustive all-pairs comparisons in Dashing, and 23 times the number of comparisons as the larger experiment in the previous section 3.4.5. This runtime result is reported in the KNN figure 3.6. We also applied this approach to the much larger UniRef50 dataset, delineated by protein similarity thresholds of 50%. After sketching, the KNN graph was generated in less than 10 minutes. By contrast, the exhaustive all-pairs comparisons approach timed out after 2 days; this was expected; this calculation would have taken over a year to complete.

3.5 Methods

3.5.1 Sketching sequencing data

We've introduced several significant changes relevant to sketching read sets, as opposed to assembled genomes or sequence collections.

First, we've adapted a technique from the Mash paper [4] for eliminating k-mers below a specific threshold for the set-based sketching methods. If the inclusion threshold is > 1, both SetSketch implementations (One-Permutation and Full) maintain a dictionary of potentially-relevant items whose counts are below the thresholds, and these items are added to the final sketches once they pass the count threshold.

This eliminates rare k-mers from read sets without requiring excessive storage. Dashing 1's approach was to use a count-min sketch, but this was substantially slower than this alternative method. On the other hand, the memory requirement for the count-min sketch approach is fixed.

Second, we've added a downsampling approach ("–downsample <float>"), which causes Dashing 2 to randomly ignore a specified fraction of k-mers F. Frequent k-mers, occurring substantially more commonly than the $\frac{1}{F}$ will be expected to be included, but rarer k-mers will tend to be eliminated without any storage costs.

Most importantly, we've introduced feature hashing [15], equivalent to a one-row count-min sketch [23], for use in conjunction with weighted variations of the SetSketch (ProbMinhash and BagMinHash).

This allows us to bound our memory usage when generating weighted sketches at the expense of some approximation.

Dashing 2's default counting method is exact before generating these weighted sketches, but when feature hashing is enabled, the input k-mers are hashed to a pseudorandom feature index before incrementing. This introduces a systematic bias toward over-estimating similarity, as feature hashing can only increase estimated similarity between two weighted sets, but the magnitude of this bias is relatively small, as demonstrated by our experiments.

3.6 Discussion

Dashing 2 addresses all of the motivating needs: (a) ability to handle much more than 10,000s of pairs, (b) able to handle multiplicities, and (c) better accuracy than before, partly through efficient use of the "wasted" bits in the original.

The future will have even more assembled genomes, especially as it becomes easier to assemble genomes directly from metagenomes. E.g., practical improvements from Martin Steinegger's work and improvements in high-accuracy long-read sequencing technology.

We may try to compete with Linclust, CD-HIT, and similar tools more specifically designed for protein sequence clustering. This would be a big, separate effort, and beyond the scope of this paper.

3.6.1 Future Improvements

While our LSH index is efficient and accurate, it may be possible to reduce its space requirements or improve runtime. The PUFFINN [24] storage method, which uses both tensoring (re-using LSH registers) and sorted substring tables

for flexible matching. This allows for trie matching of LSH strings without the memory alocality that a pointer-based trie implementation would cause. This solves the a-priori selection of subtable key length selection. This also improves the efficiency for batched queries, which can share memory latency costs across queries. We may adapt this approach for our use cases, along with different heuristics designed for our particular use-case. We suspect that for these dense trie indexes, 4-8 are likely sufficient for typical genomic applications.

We may also replace the feature-hashing approach for weighted sketching with a Counting Quotient Filter [25] or a variant thereof. If it can reduce collisions in comparable space, perhaps it can improve downstream weighted *k*-mer set comparisons.

Clustering results

Since exhaustive pairwise distance matrices become impractical with millions of items, we suggest graph-based approaches such as Leiden [26]. Other options would include HDBSCAN [27] or UMAP-based methods [28], both of which can work efficiently with sparse nearest-neighbor graphs.

These methods can deal with global structure efficiently by leveraging the sparsity that comes from K-nearest neighbor- or similarity thresholded- graphs. UMAP, in particular, also provides continuous embeddings and a visualization method for genome similarity, which can be useful for visualizing the structure of collections.

Figure 3.7: C/C++ code for fast approximate $\log_2$

Using the representation of floating-point numbers of significand length L_S as an approximate log2 after, then (1) dividing by the 2^{L_S} and (2) subtracting the bias. Note that this description may violate strict aliasing; using `memcpy` avoids this.
We provide code for both versions below, separated by a `NO_STRICT_ALIASING` macro.

```
static inline float flog2_float(float x) {
#ifndef NO_STRICT_ALIASING
    x = (float)*(uint32_t *)&x * 0x1p-23f - 127.f;
#else
    uint32_t y;
    memcpy(&y, &x, sizeof(y));
    x = 0x1p-23f * y - 127.f;
#endif
    return x;
}
static inline double fastlog2_double(double x) {
    return (double)*(uint64_t *)&x * 0x1p-52 - 1023.;
#ifndef NO_STRICT_ALIASING
    x = (double)*(uint64_t *)&x * 0x1p-52 - 1023.;
#else
    uint64_t y;
    memcpy(&y, &x, sizeof(y));
    x = 0x1p-23f * y - 127.f;
#endif
    return x;
}
```

Figure 3.8: C/C++ code for fast approximate $\log_e$

Using the representation of floating-point numbers of significand length L_S as an approximate log2 after, then (1) dividing by the 2^{L_s} and (2) subtracting the bias. It over-estimates, but by a factor ≤ 1.42, allowing its use for filtering. These methods are equivalent to the flog2 methods, but scaled by $\log_e 2$.

```
static inline double flog_double(double x) {
    return (double)*(uint64_t *)&x * 1.539095918623324e-16
           - 709.0895657128241;
}

static inline float flog_float(float x) {
    return (float)*(uint32_t *)&x * 8.262958e-8f - 88.02969f;
}
```

References

[1] Otmar Ertl. "SetSketch: Filling the Gap between MinHash and HyperLogLog". In: *CoRR* abs/2101.00314 (2021). arXiv: 2101.00314. URL: https://arxiv.org/abs/2101.00314.

[2] Otmar Ertl. "ProbMinHash - A Class of Locality-Sensitive Hash Algorithms for the (Probability) Jaccard Similarity". In: *CoRR* abs/1911.00675 (2019). arXiv: 1911.00675. URL: http://arxiv.org/abs/1911.00675.

[3] Otmar Ertl. "BagMinHash - Minwise Hashing Algorithm for Weighted Sets". In: *CoRR* abs/1802.03914 (2018). arXiv: 1802.03914. URL: http://arxiv.org/abs/1802.03914.

[4] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. "Mash: fast genome and metagenome distance estimation using MinHash". In: *Genome Biol.* 17.1 (2016), p. 132.

[5] D. N. Baker and B. Langmead. "Dashing: fast and accurate genomic distances with HyperLogLog". In: *Genome Biol* 20.1 (2019), p. 265.

[6] Andrei Z Broder. "On the resemblance and containment of documents". In: *Compression and complexity of sequences 1997. proceedings*. IEEE. 1997, pp. 21–29.

[7] Ben Woodcroft. *CoverM*. 2018. URL: https://github.com/wwood/CoverM.

[8] A. Criscuolo. "On the transformation of MinHash-based uncorrected distances into proper evolutionary distances for phylogenetic inference". In: *F1000Res* 9 (2020), p. 1309.

[9] M. Steinegger and J. Söding. "Clustering huge protein sequence sets in linear time". In: *Nat Commun* 9.1 (2018), p. 2542.

[10] XiaoFei Zhao. "BinDash, software for fast genome distance estimation on a typical personal laptop". In: *Bioinformatics* (2018), bty651.

[11] Julie E. Hernández-Salmerón and Gabriel Moreno-Hagelsieb. "ANI, Mash and Dashing equally differentiate between Klebsiella species". In: *bioRxiv* (2021). DOI: 10.1101/2021.11.05.467470. eprint: https://www.biorxiv.org/content/early/2021/11/05/2021.11.05.467470.full.pdf. URL: https://www.biorxiv.org/content/early/2021/11/05/2021.11.05.467470.

[12] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *AofA: Analysis of Algorithms*. Ed. by Philippe Jacquet. Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). DMTCS Proceedings. Juan les Pins, France: Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156. URL: https://hal.inria.fr/hal-00406166.

[13] Otmar Ertl. "New cardinality estimation algorithms for HyperLogLog sketches". In: *CoRR* abs/1702.01284 (2017). arXiv: 1702.01284. URL: http://arxiv.org/abs/1702.01284.

[14] Anshumali Shrivastava. "Optimal Densification for Fast and Accurate Minwise Hashing". In: *CoRR* abs/1703.04664 (2017). arXiv: 1703.04664. URL: http://arxiv.org/abs/1703.04664.

[15] John Moody. ""Fast Learning in Multi-Resolution Hierarchies"". In: *Proceedings of the 1st International Conference on Neural Information Processing Systems*. NIPS'88. Cambridge, MA, USA: MIT Press, 1988, 29–39.

[16] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58 –75. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001. URL: http://www.sciencedirect.com/science/article/pii/S0196677403001913.

[17] L. Fu, B. Niu, Z. Zhu, S. Wu, and W. Li. "CD-HIT: accelerated for clustering the next-generation sequencing data". In: *Bioinformatics* 28.23 (2012), pp. 3150–3152.

[18] W. Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". In: *Commun. ACM* 8.1 (1965), p. 40. ISSN: 0001-0782. DOI: 10.1145/363707.363723. URL: https://doi.org/10.1145/363707.363723.

[19] I. Babuska. "Numerical stability in mathematical analysis". In: IFIP 1968. North-Holland, Amsterdan: IFIP Congress, 1969, pp. 11–23.

[20] C. Jain, L. M. Rodriguez-R, A. M. Phillippy, K. T. Konstantinidis, and S. Aluru. "High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries". In: *Nat Commun* 9.1 (2018), p. 5114.

[21] Daniel N Baker and Ben Langmead. *Dashing software used in manuscript experiments*. 2021. URL: https://github.com/dnbaker/dashing2-experiments.

[22] Robert C. Edgar. "Local homology recognition and distance measures in linear time using compressed amino acid alphabets". In: *Nucleic Acids Research* 32.1 (2004), pp. 380–385. ISSN: 0305-1048. DOI: 10.1093/nar/gkh180. eprint: https://academic.oup.com/nar/article-pdf/32/1/380/4027416/gkh180.pdf. URL: https://doi.org/10.1093/nar/gkh180.

[23] Graham Cormode and Shan Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75.

[24] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. "PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors". In: *CoRR* abs/1906.12211 (2019). arXiv: 1906.12211. URL: http://arxiv.org/abs/1906.12211.

[25] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. "A General-Purpose Counting Filter: Making Every Bit Count". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 775–787. ISBN: 9781450341974. DOI: 10.1145/3035918.3035963. URL: https://doi.org/10.1145/3035918.3035963.

[26] V. A. Traag, L. Waltman, and N. J. van Eck. "From Louvain to Leiden: guaranteeing well-connected communities". In: *Sci Rep* 9.1 (2019), p. 5233.

[27] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. "PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors". In: *CoRR* abs/1906.12211 (2019). arXiv: 1906.12211. URL: http://arxiv.org/abs/1906.12211.

[28] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML].

Chapter 4

Coresets for Clustering in Graphs of Bounded Treewidth

4.1 Context

This thesis chapter is comprised of the 2020 ICML paper, *Coresets for Clustering in Graphs of Bounded Treewidth* [1], which was a joint effort with Xuan Wu, Shaofeng Jiang, Robert Krauthgamer, Lingxiao Huang, and Vladimir Braverman.

Theoretical insights were provided by Xuan Wu, Lingxiao Huang, Vladimir Braverman, and Shaofeng Jiang, and Robert Krauthgamer, of which proof derivations are primarily the work of Xuan Wu.

Experimental design, algorithm selection and design were the work of Xuan Wu, Vladimir Braverman, Shaofeng Jiang, Robert Krauthgamer, and Daniel Baker.

Practical implementation was prepared by Daniel Baker and experiments were performed by Daniel Baker.

For this work, my contributions were in providing the experimental section

of the paper. With guidance and assistance primarily from Shaofeng Jiang and Xuan Wu, I implemented efficient methods as necessary to support the theoretical findings of the group.

The theoretical contributions of this work, developed by the author co-authors, were to use the treewidth of graphs to derive bounds for accuracy for coreset generation for the graph shortest-paths distance metric and their zth powers, where $z \geq 1$. Since many important graphs, including road networks, biological systems, and telecommunications networks, satisfy this property, these findings apply to many practical problems [2].

However, the coreset construction algorithm requires an approximate solution to the k, z-clustering problem, where k is the number of center nodes, and z is the power of the shortest-paths distance metric where $z \geq 1$. I developed software for extracting paths from OpenStreetMap databases [3], clips it to bounding boxes, selects the largest connected component, converts the OSM database connections to actual distances using the Haversine formula, and then performed our graph experiments on this real data.

To perform these experiments, I implemented the Gonzalez, k-center algorithm, which provides an $\in\backslash-$ approximate solution to the k-clustering problem. Then, I developed a SIMD-acclerated and massively parallelized implementation of the local-search algorithm [4], which yields a 5-approximate solution to the k-clustering problem using swaps of size 1. (This may be improved by exploring multi-swaps at polynomial increases in runtime.)

In addition, in order to make this search problem practical on millions of nodes, we applied the iterated graph-sampling approach from [5]. This

algorithms as described in the paper yielded impractically large subsets still. Shaofeng introduced an repeated, iterated sampling approach which preserved approximation accuracy while also allowing us to reduce the graph search space to the point that this problem became practical, 1 2.

Placing all of this together, we demonstrated the practical usefulness of these developments, yielding runtime improvements in the millions while nearly preserving the fidelity of our approximation.

4.2 Abstract

We initiate the study of coresets for clustering in graph metrics, i.e., the shortest-path metric of edge-weighted graphs. Such clustering problems are essential to data analysis and used for example in road networks and data visualization. A coreset is a compact summary of the data that approximately preserves the clustering objective for every possible center set, and it offers significant efficiency improvements in terms of running time, storage, and communication, including in streaming and distributed settings. Our main result is a near-linear time construction of a coreset for k-MEDIAN in a general graph G, with size $O_{\epsilon,k}(\mathrm{tw}(G))$ where $\mathrm{tw}(G)$ is the treewidth of G, and we complement the construction with a nearly-tight size lower bound. The construction is based on the framework of Feldman and Langberg [STOC 2011], and our main technical contribution, as required by this framework, is a uniform bound of $O(\mathrm{tw}(G))$ on the shattering dimension under any point weights. We validate our coreset on real-world road networks, and our scalable algorithm constructs tiny coresets with high accuracy, which

translates to a massive speedup of existing approximation algorithms such as local search for graph k-MEDIAN.

4.3 Introduction

We initiate the study of coresets for clustering in *graph metrics*, i.e., the shortest-path metrics of graphs. As usual in these contexts, the focus is on edge-weighted graphs $G = (V, E)$ with a restricted topology, and in our case bounded treewidth. Previously, coresets were studied extensively but mostly under geometric restrictions, e.g., for Euclidean metrics.

4.3.1 Coresets for k-Clustering

We consider the *metric* k-MEDIAN problem, whose input is a metric space $M = (V, d)$ and an n-point data set $X \subseteq V$, and the goal is to find a set $C \subseteq V$ of k points, called *center set*, that minimizes the objective function

$$\text{cost}(X, C) := \sum_{x \in X} d(x, C),$$

where $d(x, C) := \min\{d(x, c) : c \in C\}$. The metric k-MEDIAN generalizes the well-known Euclidean case, in which $V = \mathbb{R}^d$ and $d(x, y) = \|x - y\|_2$. k-MEDIAN problem and related k-clustering problems (like k-MEANS, whose objective is $\sum_{x \in X} (d(x, C))^2$), are essential tools in data analysis and are used in many application domains, such as genetics, information retrieval, and pattern recognition. However, finding an optimal clustering is a nontrivial task, and even in settings where polynomial-time algorithms are known, it is often challenging in practice because data sets are huge, and potentially

distributed or arriving over time. To this end, a powerful data-reduction technique, called *coresets*, is of key importance.

Roughly speaking, a coreset is a compact summary of the data points by weighted points, that approximates the clustering objective for every possible choice of the center set. Formally, an ϵ-coreset for k-MEDIAN is a subset $D \subseteq V$ with weight $w : D \to \mathbb{R}_+$, such that for every k-subset $C \subseteq V$,

$$\sum_{x \in D} w(x) \cdot d(x, C) \in (1 \pm \epsilon) \cdot \mathrm{cost}(X, C).$$

This notion, sometimes called a strong coreset, was proposed in [6], following a weaker notion of [7]. Small-size coresets (where size is defined as $|D|$) often translate to faster algorithms, more efficient storage/communication of data, and streaming/distributed algorithms via the merge-and-reduce framework, see e.g. [6, 8, 9, 10, 11] and recent surveys [12, 13, 14].

Coresets for k-MEDIAN were studied extensively in Euclidean spaces, i.e., when $V = \mathbb{R}^{\mathsf{d}}$ and $d(x, y) = \|x - y\|_2$. The size of the first ϵ-coreset for k-MEDIAN, when they were first proposed [6], was $O(k(\frac{1}{\epsilon})^{\mathsf{d}} \cdot \log n)$, and it was improved to $O(k(\frac{1}{\epsilon})^{\mathsf{d}})$, which is independent of n, in [15]. Feldman and Langberg [16] drastically improved the dependence on the dimension d, from exponential to linear, achieving an ϵ-coreset of size $O(\frac{k}{\epsilon^2} \cdot \mathsf{d})$, and this bound was recently generalized to doubling metrics [10]. Recently, coresets of size *independent* of d and polynomial in $\frac{k}{\epsilon}$ were devised by [17].

4.3.2 Clustering in Graph Metrics

While clustering in Euclidean spaces is very common and well studied, clustering in graph metrics is also of great importance and has many applications. For instance, clustering is widely used for community detection in social networks [18], and is an important technique for the visualization of graph data [19]. Moreover, k-clustering on graph metrics is one of the central tasks in data mining of spatial (e.g., road) networks [20, 21], and it has been applied in various data analysis methods [22, 23], and many other applications can be found in a survey [24].

Despite the importance of graph k-MEDIAN, coresets for this problem were not studied before, and to the best of our knowledge, the only known constructions applicable to graph metrics are coresets for general n-point metrics $M = (V, d)$ with $X = V$ [25, 16], that have size $\text{poly} \log n$. In contrast, as mentioned above, coresets for Euclidean spaces usually have size independent of $n = |V|$ and sometimes even independent of the dimension d. Moreover, this generic construction assumes efficient access to the distance function, which is expensive in graphs and requires to compute all-pairs shortest paths.

To fill this gap, we study coresets for k-MEDIAN on the shortest-path metric of an edge-weighted graph G. As a baseline, we confirm that the $O(\log n)$ factor in coreset size is really necessary for general graphs, which motivates us to explore whether structured graphs admit smaller coresets. We achieve this by designing coresets whose size are independent of n when G has a bounded *treewidth* (see Definition **??**), which is a special yet common graph family. Moreover, our algorithm for constructing the coresets runs in *near-linear time*

(for every graph regardless of treewidth).

Indeed, treewidth is a well-studied parameter that measures how close a graph is to a tree [26, 27], and intuitively it guarantees a (small) vertex separator in every subgraph. Several important graph families have bounded treewidth: trees have treewidth at most 1, series-parallel graphs have treewidth at most 2, and k-outerplanar graphs, which are an important special case of planar graphs, have treewidth $O(k)$. In practice, treewidth is a good complexity measure for many types of graph data. A recent experimental study showed that real data sets in various domains including road networks of the US power grid networks and social networks such as an ego-network of Facebook, have small to moderate treewidth [28].

4.4 Results

Our main result is a near-linear time construction of a coreset for k-MEDIAN whose size depends linearly on the treewidth of G and is completely independent of $|X|$ (the size of the data set). This significantly improves the generic $O(\frac{k}{\epsilon^2} \cdot \log n)$ size bound from [16] whenever the graph has small treewidth.

Theorem 4.4.1 (Fast Coresets for Graph k-MEDIAN; see Theorem **??**)**.** *For every edge-weighted graph $G = (V, E)$, $0 < \epsilon < 1$, and integer $k \geq 1$, k-MEDIAN of every data set $X \subseteq V$ (with respect to the shortest-path metric of G) admits an ϵ-coreset of size $\tilde{O}(\frac{k^3}{\epsilon^2}) \cdot \mathrm{tw}(G)$.*[1] *Furthermore, the coreset can be computed in time $\tilde{O}(|E|)$ with high probability.*[2]

[1] Throughout, we use $\tilde{O}(f)$ to denote $O(f \cdot \mathrm{polylog}(f))$.

[2] We note that our size bound can be improved to $\tilde{O}(\frac{k^2}{\epsilon^2}) \cdot \mathrm{tw}(G)$, by replacing Lemma **??** with Theorem 31 of a recent work [29].

We complement our coreset construction with a size lower bound, which is information-theoretic, i.e., regardless of computational power.

Theorem 4.4.2 (Coreset Size Lower Bound; proved in full version)**.** *For every* $0 < \epsilon < 1$ *and integers* $k, t \geq 1$*, there exists a graph* $G = (V, E)$ *with* $\mathrm{tw}(G) \leq t$*, such that every* ϵ*-coreset for* k-MEDIAN *on* $X = V$ *in* G *has size* $\Omega(\frac{k}{\epsilon} \cdot t)$*.*

This matches the linear dependence on $\mathrm{tw}(G)$ in our coreset construction, and we show in a corollary (in full version) that the same hard instance actually implies for the first time that the $O(\log n)$ factor is optimal for general metrics, which justifies considering restricted graph families.

4.4.1 Experiments

We evaluate our coreset on real-world road networks. Thanks to our new near-linear time algorithm, the coreset construction scales well even on data sets with millions of points. Our coreset consistently achieves $< 5\%$ error using only 1000 points on various distributions of data points X, and the small size of the coreset results in a 100x-1000x speedup of local search approximation algorithm for graph k-MEDIAN. When experimenting with our coreset on different data sets X, we observe that coresets of similar size yield similar error, which confirms our theoretical bounds (for structured graphs) where the coreset size is independent of the data set.

In fact, our experiments demonstrate that the algorithm performs well even without knowing the treewidth of the graph G. More precisely, the algorithm can be executed on an arbitrary graph G, and the treewidth parameter is needed only to tune the coreset size. We do not know the treewidth

of the graphs used in the experiments (we made no attempt to compute it, even approximately). Our experiments validate the algorithm's effectiveness in practice, with coreset size much smaller than our worst-case theoretical guarantees. In fact, it is also plausible that while the graphs have moderate treewidth, they are actually "close" to having an even smaller treewidth. Another possible explanation is that the algorithm actually works well on a wider family of graphs than bounded treewidth, hence it is an interesting open question to analyze our construction for graphs that are planar or excluding a fixed minor.

4.4.2 Related Work

Approximation algorithms have been extensively studied for k-MEDIAN in graph metrics, and here we only mention a small selection of results. In general graphs (which is equivalent to general metrics), it is NP-hard to approximate k-MEDIAN within $1 + \frac{2}{e}$ factor [30], and the state-of-art is a 2.675-approximation [31]. For planar graphs and more generally graphs excluding a fixed minor, a PTAS for k-MEDIAN was obtained in [4] based on local search, and it has been improved to be FPT (i.e. the running time is of the form $f(k, \epsilon) \cdot n^{O(1)}$) recently [32]. For general graphs, [5] proposed an $O(1)$-approximation that runs in near-linear time.

Coresets have been studied for many problems in addition to k-MEDIAN, such as PCA [29] and regression [33], but in our context we focus on discussing results for other clustering problems only. For k-CENTER clustering in Euclidean space $\mathbb{R}^d$, an ϵ-coreset of size $O(\frac{k}{\epsilon^d})$ can be constructed in near-linear

time [34, 35]. Recently, coreset for generalized clustering objective receives attention from the research community, for example, [36] obtained simultaneous coreset for ORDERED k-MEDIAN, [37, 38] gave a coresets for k-clustering with fairness constraints, and [39] presented a coreset for k-MEANS clustering on lines in Euclidean spaces where inputs are lines in $\mathbb{R}^d$ while the centers are points.

4.5 Coresets for k-MEDIAN in Graph Metrics

4.5.1 Referral

We would now be in position to conclude our main result. However, the details of this are the contributions and expertise of the other authors. I will instead provided a summary of these details but refer interested parties to the ICML paper itself [40]. We use sampling algorithms and discrete optimization techniques to generate constant-factor approximate solution to the k-clustering problem. Treewidth is a property of graphs which formalizes a notion of complexity with regard to its structure. If this is taken to be bounded, the other authors prove that one can bound the shattering dimension of the metric space corresponding to the graph shortest-paths distance metric. This, in turn, yields a bound on the accuracy of coresets generated using the Feldman-Langberg framework, explaining why the techniques we applied in the experiment efficiently yielded high-accuracy approximations.

Algorithm 1: ITERATEDTHORUPSAMPLING

0: **Input** edge-weighted graph $G = (V, E)$, data set $X \subseteq V$, number of centers k, parameters n and m that control the number of iterations
0: **Output** bicriteria solution $F \subseteq X$
0: let $X_0 \leftarrow X$, and $\forall u \in X_0$ let $w_{X_0}(u) \leftarrow 1$
1: **for** $i = 1$ to n **do**
2: expand X_{i-1} into a multi-set X', such that each $u \in X'$ has multiplicity $w_{X_{i-1}}(u)$
3: let $F_i \leftarrow$ THOSAMPLEBEST(G, X', k, m)
4: let $X_i \leftarrow F_i$ and $\forall u \in X_i$, let

$$w_{X_i}(u) \leftarrow |\{v \in X : \mathrm{NN}_{F_i}(v) = u\}|$$

// $\mathrm{NN}_{F_i}(v)$ is the nearest point in F_i from v; this step implements the projection of X to F_i
5: **end for**return F_n

4.6 Experiments

We implement our algorithm and evaluate its performance on real-world road networks. Our implementation generally follows the importance sampling algorithm as in the Upper-Bound section of the paper. We observe that the running time is dominated by computing an $O(1)$-approximation for k-MEDIAN (used to assign importance σ_x), for which we use Thorup's $\tilde{O}(|E|)$-time algorithm. However, the straightforward implementation of Thorup's algorithm is very complicated and scales with $k \log^3 n$ which is already near the size of our data set, and thus we employ an optimized implementation based on it.

4.6.1 Optimized Implementation

Thorup's algorithm starts with an $O(|E| \log |E|)$-time procedure to find a bicriteria solution F (Algorithm D in [5]), namely, $|F| = O(k \log^2 n)$ such that

Algorithm 2: THOSAMPLEBEST

0: **Input** edge-weighted graph $G = (V, E)$, data set $X \subseteq V$, number of centers k, number of iterations m
0: **Output** bicriteria solution $F \subseteq X$
1: **for** $i = 1$ to m **do**
2: let $F_i \leftarrow \text{THOSAMPLE}(G, X, k)$
// THOSAMPLE is Algorithm D of [5]
3: **end for**
4: return F_i such that $i = \arg\min_{1 \le j \le m} \text{cost}(F_i, X)$

$\text{cost}(F, X) = O(1) \cdot \text{OPT}$. Then a modified Jain-Vazirani algorithm [41] is applied on F to produce the final $O(1)$-approximation in $\tilde{O}(|E|)$ time. However, the modified Jain-Vazirani algorithm is complicated to implement, and the hidden polylogarithmic factor in its running time is quite large. Thus, we replace the Jain-Vazirani algorithm with a simple local search algorithm [42] to find an $O(1)$-approximation on F. The performance of the local search relies heavily on $|F|$, but $|F| = O(k \log^2 n)$ is not much smaller than n for our data set. Therefore, we run the bicriteria approximation *iteratively* to further reduce $|F|$. Specifically, after we obtain F_i, we project X to F_i (i.e., map each $x \in X$ to its nearest point in F_i) to form X_i, and run the bicriteria algorithm again on X_i to form F_{i+1}. We use a parameter to control the number of iterations, and we observe that F reduces significantly in our data set with only a few iterations.

The procedure for finding F iteratively is described in Algorithm 1, which uses Algorithm 2 as a subroutine. Algorithm 2 essentially corresponds to the above-mentioned Thorup's bicriteria approximation algorithm THOSAMPLE (Algorithm D in [5]), except that we execute it multiple times (m times in Algorithm 1) to boost the success probability. As can be seen in our experiments,

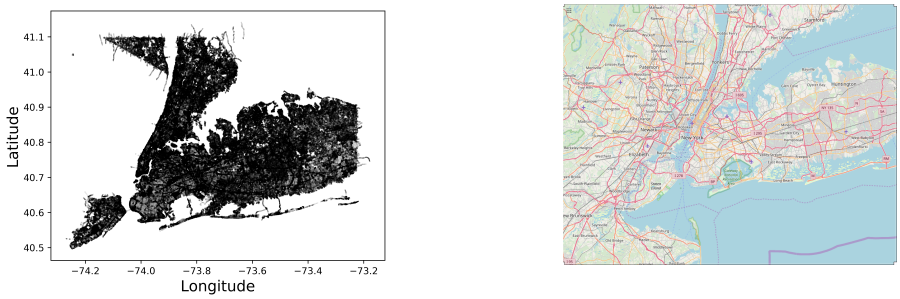


Figure 4.1: Illustration of our graph G, plotting (on left) the vertices according to their geographic coordinates, and showing (on right) a map, taken from OpenStreetMap, of the bounding box used to form G.

the improved implementation scales very well on road networks and achieves high accuracy.

4.6.2 Experimental Setup

Throughout the experiments the graph G is a road network of New York STATE extracted from OpenStreetMap [43] and clipped by bounding box to enclose New York City (NYC). This graph consists of 1 million vertices and 1.2 million edges whose weight are the distances calculated using the Haversine formula between the endpoints. It is illustrated in Figure 4.1. Our software is open source and freely available, and implemented in C++17. All experiments were performed on a Lenovo x3850 X6 system with 4 2.0 GHz Intel E7-4830 CPUs, each with 14 processor cores with hyperthreading enabled. The system had 1 TB of RAM.

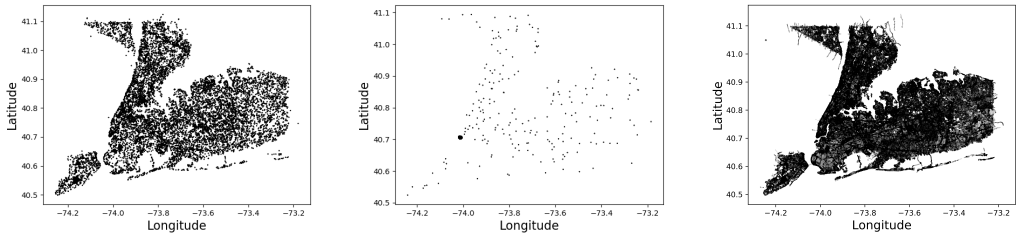


Figure 4.2: Illustration of data set X used in the accuracy-vs-size experiment. The left plot is a uniform data X_{uni}, the middle is X_{man} that is highly concentrated in Manhattan, where in both cases $|X| \approx 14000$, and the right plot is all of V which is the full NYC.

4.6.3 Performance of Coresets

Our first experiments evaluate how the accuracy of our coresets depends on their size. Here, the data X may be interpreted as a set of customers to be clustered, and their distribution could have interesting geographical patterns. We experiment with X chosen uniformly at random from V (all of NYC), mostly for completeness as it is less likely in practice, and denote this scenario as X_{uni}. We also experiment with a "concentrated" scenario where X is highly concentrated in Manhattan but also has much fewer points picked uniformly from other parts of NYC, denoted as X_{man}. We demonstrate the two types of data sets X in Figure 4.2.

We define the *empirical error* of a coreset D and a center set $C \subseteq V$ as $\text{err}(D, C) := \left| \frac{\text{cost}(D,C)}{\text{cost}(X,C)} - 1 \right|$ (corresponding to ϵ in the definition of a coreset). Since by definition a coreset preserves the objective for all center sets, we evaluate the empirical error by randomly picking 2000 center sets $C \subseteq V$ from V, and reporting the *maximum* empirical error $\text{err}(D, C)$ over all these C. For the sake of evaluation, we compare the maximum empirical error of

Table 4.1: Comparison of empirical error of our coreset with the baseline of uniform sampling when $k = 25$ and varying coreset sizes, for both data sets X_{uni} and X_{man}.

SIZE	X_{uni}		X_{man}	
	OURS	UNI.	OURS	UNI.
25	32.1%	35.8%	32.1%	151.6%
50	26.6%	23.0%	22.1%	90.3%
75	17.8%	23.2%	23.2%	62.3%
100	17.2%	17.2%	15.2%	49.9%
500	7.72%	8.53%	8.34%	31.7%
1250	4.57%	5.32%	4.87%	21.2%
2500	4.14%	4.03%	3.29%	9.53%
3750	2.49%	3.21%	2.89%	14.39%
6561	2.00%	2.11%	2.38%	5.83%
13122	1.50%	1.70%	1.53%	6.53%
19683	1.27%	1.36%	1.39%	3.73%

our coreset with a baseline of a uniform sample, where points are drawn uniformly at random from X and assigned equal weight (that sums to $|X|$). To reduce the variance introduced by the randomness in the coreset construction, we repeat each construction 10 times and report the average of their maximum empirical error.

4.6.3.1 Results

We report the empirical error of our coresets and that of the uniform sampling baseline in Table 4.1. Our coreset performs consistently well and quite similarly on the two data sets X, achieving for example 5% error using only about 1000 points. Compared to the uniform sampling baseline, our coreset is $3 - 5$ times more accurate on the Manhattan-concentrated data X_{man}, and (as expected) is comparable to the baseline on the uniform data X_{uni}.

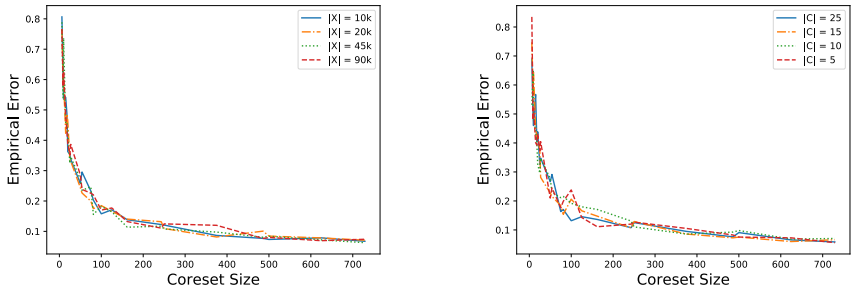


Figure 4.3: The left plot shows the accuracy of coresets ($k = 25$) on uniform X's with varying sizes. Each line is labeled with the size of each respective $X \subseteq V$. The right plot shows the accuracy of coresets constructed with $k = 25$ but evaluated with smaller center sets C on the same uniform X with $|X| = 10^4$.

In addition, we show the accuracy of our coresets with respect to varying sizes of data sets X in Figure 4.3 (left). We find that coresets of the same size have similar accuracy regardless of $|X|$, which confirms our theory that the size of the coreset is independent of $|X|$ in structured graphs. We also verify in Figure 4.3 (right) that a coreset constructed for a target value $k = 25$ performs well also as a coreset for fewer centers (various $k' < k$). While this should not be surprising and follows from the coreset definition, it is very useful in practice when k is not known in advance, and a coreset (constructed for large enough k) can be used to experiment and investigate different $k' < k$.

4.6.4 Speedup of Local Search

An important application of coresets is to speed up existing approximation algorithms. To this end, we demonstrate the speedup of the local search algorithm of [42] achieving 5-approximation for graph k-MEDIAN by using our coreset. In particular, we run the local search on top of our coreset D (denoted as $D \times V$), and then compare the accuracy and the overall running

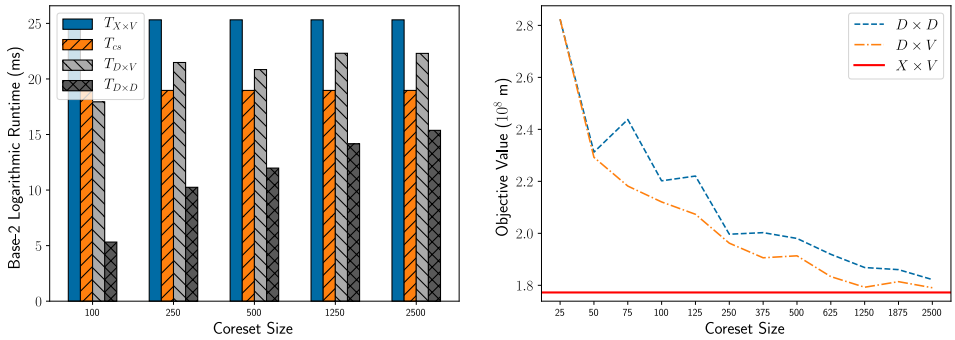


Figure 4.4: Performance of local search on $X \times V$, $D \times V$, and $D \times D$. The running time is shown on the left, where the coreset construction time T_{cs} is separated out (so $T_{D\times V}$ and $T_{D\times D}$ do not include T_{cs}). The objective values reached are shown on the right. Here $k = 25$ and $X = V$ is the whole NYC of size $|X| \approx 10^6$.

time with those of running the local search on the original data X (denoted as $X \times V$). Notice that by definition of k-MEDIAN, the centers always come from V, which defines the search space, and a smaller data set can only affect the time required to evaluate the objective. This limits the potential speedup of local search, and therefore we additionally evaluate the running time and accuracy of local search on D when also the centers come from D (denoted as $D \times D$).

We report separately the running time of the coreset construction, denoted T_{cs}, and that of the local search on the coreset. Indeed, as mentioned in Section 4.6.3, a coreset D constructed for large k can be used also when clustering for $k' < k$, and since one can experiment with any clustering algorithm on D (e.g. Jain-Vazirani, local search, etc.), the coreset construction is one-time effort that may be averaged out when successive clustering tasks are performed on D.

The results are illustrated in Figure 4.4, where we find that the coreset

construction is very efficient, about 100 times faster than local search on X, not to mention that the coreset may be used for successive clustering tasks. We see that the speedup of local search $D \times V$ is only moderate (which matches the explanation above), but the alternative local search on $D \times D$ performs extremely well — for example using $|D| \approx 1000$, it is about 1000 times faster than the naive local search on X, and it achieves similar objective value (i.e. $5\% - 10\%$ error). This indicates that local search on $D \times D$ may be a good candidate for practical use.

References

[1] Daniel Baker, Vladimir Braverman, Lingxiao Huang, Shaofeng H.-C. Jiang, Robert Krauthgamer, and Xuan Wu. "Coresets for Clustering in Graphs of Bounded Treewidth". In: *CoRR* abs/1907.04733 (2019).

[2] Silviu Maniu, Pierre Senellart, and Suraj Jog. "An Experimental Study of the Treewidth of Real-World Graph Data (Extended Version)". In: *CoRR* abs/1901.06862 (2019). arXiv: 1901.06862. URL: http://arxiv.org/abs/1901.06862.

[3] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2020.

[4] Vincent Cohen-Addad, Philip N. Klein, and Claire Mathieu. "Local Search Yields Approximation Schemes for k-Means and k-Median in Euclidean and Minor-Free Metrics". In: *SIAM J. Comput.* 48.2 (2019), pp. 644–667. DOI: 10.1137/17M112717X.

[5] Mikkel Thorup. "Quick k-Median, k-Center, and Facility Location for Sparse Graphs". In: *SIAM J. Comput.* 34.2 (2005), pp. 405–432. ISSN: 0097-5397. DOI: 10.1137/S0097539701388884.

[6] Sariel Har-Peled and Soham Mazumdar. "On coresets for k-means and k-median clustering". In: *36th Annual ACM Symposium on Theory of Computing*. 2004, pp. 291–300. DOI: 10.1145/1007352.1007400.

[7] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. "Approximating Extent Measures of Points". In: *J. ACM* 51.4 (2004), pp. 606–635. ISSN: 0004-5411. DOI: 10.1145/1008731.1008736.

[8] Hendrik Fichtenberger, Marc Gillé, Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. "BICO: BIRCH Meets Coresets for k-Means Clustering". In: *ESA*. Vol. 8125. Lecture Notes in Computer Science. Springer, 2013, pp. 481–492. DOI: 10.1007/978-3-642-40450-4_41.

[9] Maria-Florina F Balcan, Steven Ehrlich, and Yingyu Liang. "Distributed k-means and k-median Clustering on General Topologies". In: *NIPS*. 2013, pp. 1995–2003.

[10] Lingxiao Huang, Shaofeng Jiang, Jian Li, and Xuan Wu. "Epsilon-Coresets for Clustering (with Outliers) in Doubling Metrics". In: *59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2018, pp. 814–825.

[11] Zachary Friggstad, Mohsen Rezapour, and Mohammad R. Salavatipour. "Local Search Yields a PTAS for k-Means in Doubling Metrics". In: *SIAM J. Comput.* 48.2 (2019), pp. 452–480.

[12] Jeff M. Phillips. "Coresets and Sketches". In: *Handbook of discrete and computational geometry*. Ed. by Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. 3rd. Chapman and Hall/CRC, 2017. Chap. 48. DOI: 10.1201/9781315119601-48.

[13] Alexander Munteanu and Chris Schwiegelshohn. "Coresets-Methods and History: A Theoreticians Design Pattern for Approximation and Streaming Algorithms". In: *KI* 32.1 (2018), pp. 37–53.

[14] Dan Feldman. "Core-sets: An updated survey". In: *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 10.1 (2020).

[15] Sariel Har-Peled and Akash Kushal. "Smaller Coresets for k-Median and k-Means Clustering". In: *Discrete & Computational Geometry* 37.1 (2007), pp. 3–19. DOI: 10.1007/s00454-006-1271-x.

[16] Dan Feldman and Michael Langberg. "A unified framework for approximating and clustering data". In: *43rd Annual ACM Symposium on Theory of computing*. ACM. 2011, pp. 569–578.

[17] Christian Sohler and David P. Woodruff. "Strong Coresets for k-Median and Subspace Approximation: Goodbye Dimension". In: *FOCS*. IEEE Computer Society, 2018, pp. 802–813.

[18] Santo Fortunato. "Community detection in graphs". In: *Physics reports* 486.3-5 (2010), pp. 75–174.

[19] Ivan Herman, Guy Melançon, and M. Scott Marshall. "Graph Visualization and Navigation in Information Visualization: A Survey". In: *IEEE Trans. Vis. Comput. Graph.* 6.1 (2000), pp. 24–43.

[20] Shashi Shekhar and Duen-Ren Liu. "CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations". In: *IEEE Trans. Knowl. Data Eng.* 9.1 (1997), pp. 102–119.

[21] Man Lung Yiu and Nikos Mamoulis. “Clustering Objects on a Spatial Network”. In: *SIGMOD Conference*. ACM, 2004, pp. 443–454.

[22] Matthew J Rattigan, Marc Maier, and David Jensen. “Graph clustering with network structure indices”. In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 783–790.

[23] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong, and Xiaoming Li. “Geometry-based edge clustering for graph visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (2008), pp. 1277–1284.

[24] Barbaros C Tansel, Richard L Francis, and Timothy J Lowe. “State of the art—location on networks: a survey, Part I and II.” In: *Management Science* 29.4 (1983), pp. 482–497.

[25] Ke Chen. “On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications”. In: *SIAM Journal on Computing* 39.3 (2009), pp. 923–947.

[26] Neil Robertson and Paul D. Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *Journal of algorithms* 7.3 (1986), pp. 309–322.

[27] Ton Kloks. *Treewidth: computations and approximations*. Vol. 842. Springer Science & Business Media, 1994.

[28] Silviu Maniu, Pierre Senellart, and Suraj Jog. “An Experimental Study of the Treewidth of Real-World Graph Data”. In: *ICDT*. Vol. 127. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 12:1–12:18.

[29] Dan Feldman, Melanie Schmidt, and Christian Sohler. “Turning Big Data Into Tiny Data: Constant-Size Coresets for k-Means, PCA, and Projective Clustering”. In: *SIAM J. Comput.* 49.3 (2020), pp. 601–657.

[30] Kamal Jain, Mohammad Mahdian, and Amin Saberi. “A new greedy approach for facility location problems”. In: *STOC*. ACM, 2002, pp. 731–740.

[31] Jaroslaw Byrka, Thomas Pensyl, Bartosz Rybicki, Aravind Srinivasan, and Khoa Trinh. “An Improved Approximation for *k*-Median and Positive Correlation in Budgeted Optimization”. In: *ACM Trans. Algorithms* 13.2 (2017), 23:1–23:31.

[32] Vincent Cohen-Addad, Marcin Pilipczuk, and Michal Pilipczuk. “Efficient Approximation Schemes for Uniform-Cost Clustering Problems in Planar Graphs”. In: *ESA*. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 33:1–33:14.

[33] Alaa Maalouf, Ibrahim Jubran, and Dan Feldman. “Fast and Accurate Least-Mean-Squares Solvers”. In: *NeurIPS*. 2019, pp. 8305–8316.

[34] Pankaj K Agarwal and Cecilia Magdalena Procopiuc. “Exact and approximation algorithms for clustering”. In: *Algorithmica* 33.2 (2002), pp. 201–226.

[35] Sariel Har-Peled. “Clustering motion”. In: *Discrete & Computational Geometry* 31.4 (2004), pp. 545–565.

[36] Vladimir Braverman, Shaofeng H.-C. Jiang, Robert Krauthgamer, and Xuan Wu. “Coresets for Ordered Weighted Clustering”. In: *ICML*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 744–753.

[37] Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. “Fair Coresets and Streaming Algorithms for Fair k-Means Clustering”. In: *CoRR* abs/1812.10854 (2018).

[38] Lingxiao Huang, Shaofeng H.-C. Jiang, and Nisheeth K. Vishnoi. “Coresets for Clustering with Fairness Constraints”. In: *NeurIPS*. 2019, pp. 7587–7598.

[39] Yair Marom and Dan Feldman. “k-Means Clustering of Lines for Big Data”. In: *NeurIPS*. 2019, pp. 12797–12806.

[40] Daniel N. Baker, Vladimir Braverman, Lingxiao Huang, Shaofeng H.-C. Jiang, Robert Krauthgamer, and Xuan Wu. “Coresets for Clustering in Graphs of Bounded Treewidth”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 569–579. URL: http://proceedings.mlr.press/v119/baker20a.html.

[41] Kamal Jain and Vijay V. Vazirani. “Approximation algorithms for metric facility location and k-Median problems using the primal-dual schema and Lagrangian relaxation”. In: *J. ACM* 48.2 (2001), pp. 274–296.

[42] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. “Local search heuristic for k-median and facility location problems”. In: *STOC*. ACM, 2001, pp. 21–29.

[43] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2020.

Chapter 5

Fast and memory-efficient scRNA-seq k-means clustering with various distances

5.1 Context

This thesis chapter is comprised of the Minicore paper [1], published as the proceedings of the ACM-BCB (Association for Computing Machinery - Bioinformatics, Computational Biology and Health Informatics) conference. The generating process of droplet single-cell RNA-Sequencing (scRNA-seq) is not Gaussian but something close to Poisson or Negative-Binomial models, summarization methods based on Euclidean space fail to account for the nature of the data. These models belong to the regular exponential family distributions, for which Bregman divergences provide a notion of distance or dissimilarity between instances.

Recent theory for clustering and coresets for regular exponential family models provides us generalized extensions of the kmeans++ and kmeans clustering algorithms [2, 3, 4, 5]. We then implemented fast vectorized code

for computing these distances for both sparse dense data, vectorized weighted sampling implementation which we make available in libsimdsampling [6], and provide both complete and mini-batch k-means implementations. To avoid undefined values, we apply a nonzero pseudocount adjustment to all fields, which corresponds to a $\Gamma(\beta, \beta)$ prior on the Poisson model.

We demonstrate our method's correctness by comparing solutions and runtime to sci-kit learn, yielding speed-up factors in the hundreds. By working on sparse representations directly, we cluster a 4-million cell atlas with over 60,000 features in minutes and using 11GiB of RAM.

5.2 Introduction

Single-cell RNA-sequencing (scRNA-seq) is capable of measuring transcriptome-wide gene expression in millions of cells per experiment. With the arrival of multi-million-cell datasets [7, 8], and larger efforts like the Human Cell Atlas [9] on the horizon, the need for methods that rapidly analyze and cluster (empirically group) cells is growing. This necessitates computational advances in methods for unsupervised clustering and summarizing large collections of cells.

k-meansis one popular clustering framework. It is classically formulated as an expectation maximization problem that starts from an initial set of k data points that act as "centers" [10], iterating to obtain final centers. These centers induce a clustering of the observations into k classes. k-means++[2] improves how the initial centers are found, yielding clear mathematical guarantees for

the overall clustering. Besides their direct application as clustering methods, *k*-means and *k*-means++ are useful as individual components of other methods, including for data quantization [10], spectral clustering [11], outlier detection [12], machine learning [13] and construction of sketches and coresets [5, 14]. For example, in scRNA-seq analysis, sketching – the selection of a possibly weighted subset of cells to use – can be used to identify rare cell types. The Geometric sketching [15], Hopper [16], and submodular sketch [17] methods all employ some form of center-finding as a subroutine.

We describe a new open source, highly efficient software library called minicore, which implements an array of algorithms to find the "center" of a group of cells – essentially a rough clustering – and for performing *k*-meansclustering seeded by those centers. The advantages of minicoreare threefold. First, minicoreuses a new vectorized weighted reservoir sampling algorithm for its initial center-finding step, making it far more efficient than competing *k*-means++implementations, such as scikit-learn [18] or pyclustering [19]. Second, Minicoreimplements a variety of distance measures, including the widely-used squared Euclidean distance, but also including others like Jensen-Shannon Divergence, Kullback-Leibler Divergence, and Bhattacharyya distance, which can be directly applied to count data or probability distributions. Third, minicoreis able to process both dense, dimensionality-reduced data – the typical input for scRNA-seq clustering methods – as well as full, sparse, non-reduced matrices of counts. Minicoreis unique in its ability to handle scRNA-seq data in both sparse and dense forms, and its support for distance measures that account for the original count-based nature of the data.

On real scRNA-seq datasets with up to millions of cells and using squared Euclidean distance, minicoreis substantially faster than scikit-learn and achieves lower objective-function cost. Further, minicorecan produce centers using a wide variety of distance measures with only minor differences in the overall running time, facilitating use of distance measures that are better attuned to the count nature of the data and do not require prior transformations [20]. Finally, we show that a complete pipeline consisting of minicore's implementations of k-means++, localsearch++ and mini-batch k-meanscan cluster a 4-million cell dataset in minutes using 20 threads and a maximum resident set size (RAM) of less than 10 GiB.

5.3 Related work

Due to its wide applicability, several methods for accelerating k-means algorithms have been introduced. These include reducing point-center comparisons, reducing the cost of centroid calculation, and approximation, whether by sampling or dimensionality reduction.

For distance metrics and ρ-metrics, the triangle inequality can provide substantial runtime improvements without altering the result [21]. Other approaches use nearest neighbor oracles [22] or LSH tables [23] to select points for centroids [22]. These can provide asymptotic improvements with high probability, though they are dependent on the success of neighbor retrieval.

For ℓ_2, data dimensionality can be reduced before clustering, as in Makarychev, Makarychev and Razenshteyn [24] (2018), and applied to scRNA-seq in SHARP [25] which works with high probability. This uses all data points

at each iteration, but in reduced dimensions. Alternatively, the points can be sampled during centroid calculation, as in [26, 27], which minicoresupports. In fact, for applications in Euclidean space, these techniques could be used together.

However, many of the dissimilarity measures that motivate our work, including KLD, and ISD, and JSD, do not satisfy the triangle inequality or support fast and effective LSH querying. Also, dimension reduction transformations typically operate in Euclidean space, not capturing the count nature of scRNA-seq data. For this reason, we prioritized accelerating distance calculations in the sparse high-dimensional setting, though advances in neighbor retrieval may allow us to make further runtime improvements.

5.4 Results

We collected scRNA-seq datasets of varying size: (a) the PBMC dataset consisting of 68,579 peripheral blood mononuclear cells (PBMC) from human [28], (b) the Cao et al. mouse organogenesis dataset (Cao2m) consisting of 2,058,652 cells [7], and (c) the Cao et al. human fetal gene expression dataset (Cao4m) consisting of 4,062,980 cells [29]. In all cases, the original form of the data is a sparse matrix of gene-by-cell nonnegative integer counts. For datasets not originally represented in compressed-sparse-row (CSR) format, we convert them to that format prior to our experiments. Each of the three datasets has an associated set of cell-type labels, obtained by the original authors through an analysis that combined an initial clustering with foreknowledge of specific marker genes [28, 7, 29]. While these label assignments are not "ground truth,"

they capture some biological foreknowledge and so we use them to evaluate our final clusterings below.

While minicorecan cluster sparse counts directly, we also generated a dense version of each of the three datasets after applying a dimensionality reduction method. Specifically, we used the truncated Singular Value Decomposition (SVD) from scikit-learn. Rows of the final matrix consist of the original data's projection into the first 500 principal components. We note that a standard PCA has a "centering" step where the mean is subtracted from each feature. We used a non-centered SVD since centering causes the matrix to lose its zero entries and become dense, in turn requiring terabytes of memory for an SVD computation over millions of cells. While non-centered SVD avoids this problem by keeping the matrix sparse, a drawback is that the resulting principal components are selected based not just on the amount of variability but also on the magnitudes of the values. This is addressed further in Discussion.

5.4.1 Fast and accurate center finding

We used minicorev0.3 and compared it to scikit-learn's v0.24 function for k-means++center finding (`sklearn.cluster.kmeans_plusplus`). We considered various values for the number of centers, k. We note that scikit-learn supports only the squared Euclidean distance measure and does not support the use of multiple threads in parallel. For the most direct comparison, we used a single thread and the squared Euclidean distance only. In all cases, we measured the running time and squared-Euclidean objective cost of the resulting set of centers. In the case of minicore, we benchmarked both the k-means++method

(MC), as well as the *k*-means++method augmented by localsearch++ (MCLS). The scikit-learn results are labeled SKL.

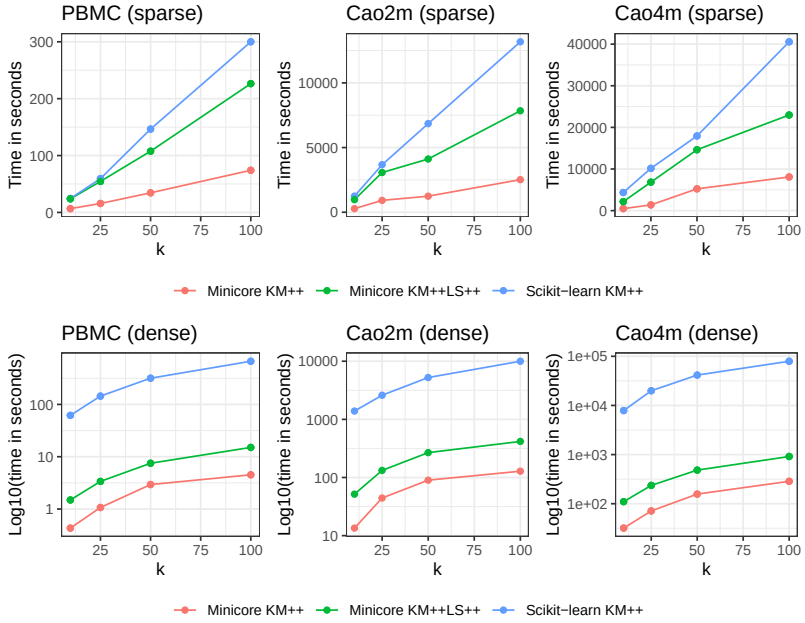


Figure 5.1: minicore*k*-means++is faster than scikit-learn *k*-means++. Performance evaluation (y-axis) of elapsed time (seconds) for sparse data (top) and $\log_{10}$ transformed time for dense data (bottom) for increasing sizes of *k* (x-axis) for the PBMC dataset with 68k cells (left), Cao et al. dataset with 2 million cells (middle), and Cao et al. dataset with 4 million cells (right). Results for minicore*k*-means++are in red (standard) and green (with localsearch++); scikit-learn *k*-means++is blue.

Using the three datasets, we found that our minicore*k*-means++(MC) implementation is significantly faster when compared to scikit-learn *k*-means++(SKL) using both sparse and dense data (Figure 5.1, Table 5.4.2). For dense input data, the MC mode of minicorehad a dramatic speed advantage, achieving 100–150 times greater speed for the PBMC dataset compared to scikit-learn, about 50–100 times greater for Cao2m, and about 240–280 times greater for Cao4m. For sparse data, the MC mode of minicorewas 3–9 times faster than

scikit-learn depending on the experiment. Our implementation of minicore*k*-means++augmented by the localsearch++ procedure (MCLS) was also always faster than SKL, and was only about 2.5–5 times slower than the MC mode, depending on the experiment.

Further, we found that both of our minicore*k*-means++implementations (MC and MCLS) obtained comparable or lower costs of the objective function compared to SKL (Table 5.4.2). MCLS obtained the lowest objective in nearly all cases across the three datasets (both dense and sparse).

Overall, the results showed that minicoreproduces high-quality centers and readily scales to multi-million cell datasets, even in their original sparse form. For example, the MC mode used about 2h:15m (single-threaded) to find $k = 100$ centers for the 4-million cell Cao4m dataset.

Similarly, minicoremakes economical use of memory even when working directly on sparse representations. The 4-million cell dataset can be clustered using less than 10 GiB RAM, allowing it to run on commodity hardware.

5.4.2 Support for both count data and continuous data

To evaluate minicore's speed for distance measures beyond the commonly used squared Euclidean distance (SQE) , we ran minicoreusing other measures, including the Bhattacharyya Metric (BAT), Kullback-Leibler Divergence (KLD), Jensen-Shannon Divergence (JSD), and cosine distance (COS). While these measures involve computationally demanding operations like logarithms and square roots, minicoreoptimizes these inner loops using the SLEEF library and vectorization [30]. An additional challenge is the need to handle 0

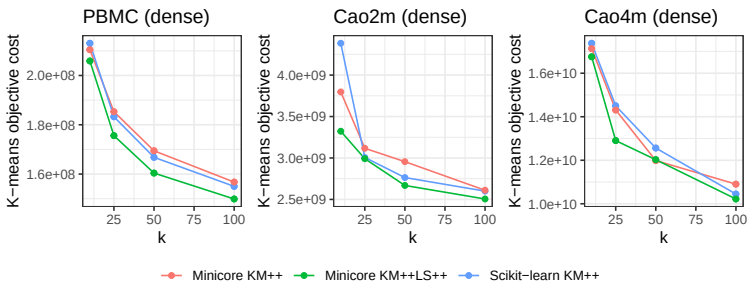


Figure 5.2: The choice of distance has minor impact on the speed of minicore*k*-means++. Performance evaluation (y-axis) of elapsed time (seconds) for sparse data for increasing sizes of k (x-axis) for Cao et al. dataset with 2 million cells (left), and Cao et al. dataset with 4 million cells (right). For a given dataset and k, the slowest measure never requires more than 61% more time than is required by the fastest measure. All experiments used 16 simultaneous threads and the localsearch++ improvement was not run.

counts, which can result in infinite divergence for measures like the KLD. To address this, we use a lazily applied prior that avoids having to instantiate a dense version of the matrix at any point. See Methods for more details on both these points.

Dataset	k	Method	Dense Time	Dense Cost	Sparse Time	Sparse Cost
PBMC	10	MC	**0.43**	2.10e+08	**6.67**	2.57e+08
		MCLS	1.49	**2.06e+08**	23.92	**2.52e+08**
		SKL	61.77	2.13e+08	24.50	2.57e+08
	25	MC	**1.07**	1.85e+08	**15.74**	2.33e+08
		MCLS	3.38	**1.76e+08**	54.53	**2.24e+08**
		SKL	144.16	1.83e+08	59.47	2.39e+08
	50	MC	**2.94**	1.69e+08	**34.46**	2.20e+08
		MCLS	7.50	**1.60e+08**	107.64	**2.17e+08**
		SKL	317.10	1.67e+08	146.53	2.20e+08
	100	MC	**4.51**	1.57e+08	**74.12**	2.12e+08
		MCLS	15.06	**1.50e+08**	226.45	**2.03e+08**
		SKL	668.40	1.55e+08	299.97	2.08e+08
Cao2m	10	MC	**13.47**	3.80e+09	**277.72**	**5.34e+09**
		MCLS	51.70	**3.32e+09**	970.49	5.39e+09

		SKL	1,389.78	4.38e+09	1,244.76	5.81e+09
	25	MC	**44.35**	3.12e+09	**916.31**	5.05e+09
		MCLS	132.44	**2.99e+09**	3,069.49	**5.04e+09**
		SKL	2,599.96	3.01e+09	3,673.07	5.28e+09
	50	MC	**89.98**	2.96e+09	**1,240.03**	5.00e+09
		MCLS	267.69	**2.67e+09**	4,115.19	**4.95e+09**
		SKL	5,241.62	2.76e+09	6,849.92	4.96e+09
	100	MC	**128.20**	2.61e+09	**2,519.19**	4.69e+09
		MCLS	416.85	**2.51e+09**	7,843.00	**4.60e+09**
		SKL	9,985.61	2.60e+09	13,176.25	4.78e+09
Cao4m	10	MC	**31.90**	1.71e+10	**478.84**	2.60e+10
		MCLS	110.01	**1.68e+10**	2,158.43	**2.35e+10**
		SKL	7,863.25	1.74e+10	4,350.81	2.61e+10
	25	MC	**71.26**	1.43e+10	**1,378.24**	2.03e+10
		MCLS	235.75	**1.29e+10**	6,839.51	**1.95e+10**
		SKL	19,889.98	1.45e+10	10,160.67	2.02e+10
	50	MC	**156.88**	**1.20e+10**	**5,229.03**	1.91e+10
		MCLS	482.96	**1.20e+10**	14,629.39	**1.77e+10**
		SKL	41,450.85	1.26e+10	17,937.88	1.86e+10
	100	MC	**285.76**	1.09e+10	**8,085.92**	1.67e+10
		MCLS	913.19	**1.02e+10**	22,974.58	**1.64e+10**
		SKL	79,269.90	1.04e+10	40,560.60	1.70e+10

Using the 2 million and 4 million Cao et al. datasets, we found that the choice of distance metric used for minicore's *k*-means++algorithm does impact speed, but not dramatically (Figure 5.2). Specifically, we found that the Bhattacharyya Metric (BAT) required less time than squared Euclidean in all cases, whereas KLD required roughly the same amount of time as SQE, and JSD generally required the most time. Importantly, the slowest measure (often the JSD) never requires more than 61% more computation time than the fastest measure.

5.4.3 k-means and mini-batch k-means clustering algorithms

The minicorelibrary also supports both full k-meansclustering using Lloyd's algorithm [31], and the faster mini-batch k-meansalgorithm [27, 4]. We sought to measure the efficiency and accuracy of a full k-meansclustering pipeline built from the k-means++, localsearch++, and mini-batch k-meanscomponents of the minicorelibrary. We chose mini-batch k-meansrather than Lloyd's algorithm because the mini-batch approach has recently been shown to be significantly faster for large datasets and provides similar results [4]. In all cases, we used $k = 25$, a mini-batch k-meansbatch size of 10,000, 25 rounds of localsearch++, and a prior of 0.01

We again analyzed the PBMC, Cao2m and Cao4m datasets. We evaluated the clusterings using the cell-type labels provided by the authors of the datasets [28, 7, 29]. Specifically, we used our k-meansclusters as empirical cell labels, using the Adjusted Rand Index (ARI) to compare these to the provided labels. Notably, the provided labels are not "ground truth," but were derived using a combination of K-nearest-neighbor graph clustering and marker gene analysis. While the ARIs we measured were generally low (sometimes negative), we caution against over-interpreting these values since the given labeling is only somewhat biologically meaningful.

While we began with the full sparse matrix, we subsampled the rows to consist of the 500 most variable genes [32], as this often achieved greater Adjusted Rand Index compared to analyzing the entire matrix. We tried several distance measures: Bhattacharyya Metric (BATMET), Jensen-Shannon Divergence (JSD), the Kullback-Leibler Divergence (KLD), and Squared Euclidean

Distance (SQE). We ran minicoreusing 20 simultaneous threads.

We found that minicorewas able to cluster the cells in all three datasets in minutes, with the slowest experiment taking about 12 minutes (Figure 5.3). For the Cao2m and Cao4m datasets, timings were in the range of 325–365 seconds and 200–700 seconds respectively.

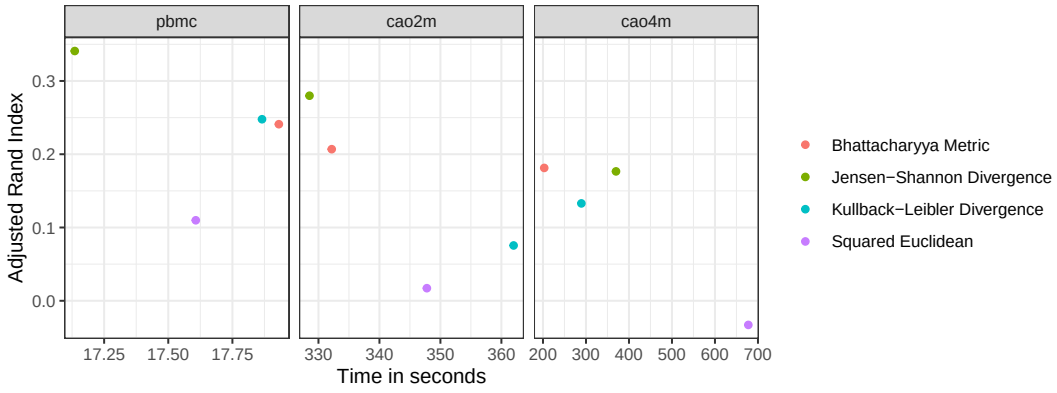


Figure 5.3: Clustering accuracy (ARI, vertical) versus running time (seconds, horizontal) for various datasets and distance measures. All experiments used the 500 most variable genes, $k = 25$, a mini-batch k-meansbatch size of 10,000, 25 rounds of localsearch++, and a prior of 0.01.

For both Cao2m and Cao4m, the Bhattacharyya Metric (BATMET) was superior to the Kullback-Leibler Divergence (KLD) and Squared Euclidean distance (SQE), achieving both greater speed and a higher Adjusted Rand Index for its final clustering. In the case of Cao2m, the JSD was superior to BATMET on both speed and ARI, but this relationship is reversed for the Cao4m dataset.

We measured minicore's peak memory footprint (resident set size) when processing the Cao4m dataset and found that it was less than 10GiB RAM. In short, we found that minicorewas capable of analyzing a 4-million cell

dataset in a few minutes using computational resources consistent with a typical commodity laptop.

5.5 Discussion

We introduced a new library called minicorefor *k*-meansclustering of scRNA-seq datasets. An efficient, vectorized sampling kernel fuels both its *k*-means++center finding algorithm and its localsearch++ algorithm for refining centers. Combined with an efficient mini-batch *k*-meansimplementation, these components form a complete and efficient pipeline for *k*-meansclustering of scRNA-seq data, requiring about 3.5 minutes to cluster a $>$4 million cell dataset when using 20 threads and less than 10GiB RAM. This low memory requirement brings even atlas-scale clustering within reach of laptops and other commodity hardware. While we applied minicoreto scRNA-seq here, its algorithms are readily adaptable to other applications, for instance in data quantization, outlier detection and spectral clustering [10, 11, 12].

5.5.1 Applications

Minicore's fast implementations of various distance measures, gives users the flexibility to tailor the distance measure to the data. Different measures might be appropriate depending on whether cells are best viewed as vectors of real numbers, vectors of counts, or probability distributions. We showed that distance measures other than squared Euclidean can perform substantially better when evaluated using given cell type labels. In future work, we plan to explore how minicorecan be applied beyond to work, for example, with

graph-induced metrics [33].

Another likely application of the algorithms in minicoreis to build "sketches" of large single-cell data compendia. A sketch is a weighted subset of cells that effectively span the gene-expression space and – like centers – facilitate the identification of accurate predicted cluster labels downstream. Sketching approaches have been applied to the problem of obtaining cluster labels that accurately capture empirical groupings of rare cell types [16, 15, 34].

Finally, we further seek to explore whether our optimized weighted sampling kernel may also be applicable in the mini-batch k-meansalgorithm, specifically for the importance sampling required to drive the gradient-descent version of mini-batch k-means[26, 35].

In some experiments described here, we used a non-centered version of the truncated Singular Value Decomposition (SVD) to project datasets into their first 500 principal components. We avoided the mean centering in order to keep the data sparse in preparation for the SVD. This has the drawback that the truncated SVD was selecting components based not only on variability, but also on the magnitudes of the points. In the future, we would like to address this by implementing or otherwise integrating a sparse version of a centered SVD computation into minicore. This could become an optional first step allowing users to create smaller, dense representations.

5.6 Methods

5.6.1 *k*-means++ algorithms

k-meansgives an efficient way to choose an initial set of centers in preparation for the more work-intensive *k*-meansoptimization procedure. Unlike the simple strategy of choosing centers uniformly at random, *k*-means++guarantees that the objective achieved by the downstream *k*-meansprocedure will be within a multiplicative $O(\log k)$ factor of the optimal cost objective.

The *k*-means++algorithm involves choosing one center per step across *k* steps. In the first step, a center is chosen from among the data points uniformly at random. In subsequent steps, a new center is chosen in a weighted random fashion, with the probability of selecting a given point being proportional to its cost, specifically the distance to the nearest already-selected center. The algorithm therefore is a weighted sampling procedure. We now describe in detail, as similar sampling procedures form the core of multiple components of minicore.

5.6.1.1 Sampling kernel

In a given step of *k*-means++, a simple sampling strategy would be to calculate the cost of each as-yet-unchosen data point (potential "center" gene) then draw a random variate from a multinomial distribution weighted by those costs. Computationally, this can be accomplished in four steps: first, calculate a cost for each point, next calculate a prefix sum over the array of all costs, next generate a uniform random variate in $[0, C]$ where C is the total cost, then perform binary search over the prefix-sum array to identify the

point corresponding to the random variate. While binary search is fast, the costs, and therefore the prefix sum, must be at least partially re-computed in each of the k steps. Further, the prefix sum computation has an inherent dependence structure that inhibits parallelization, though $O(n \log n)$-time parallel solutions exist.

Minicoreinstead uses a parallelized reservoir-sampling approach that extends an algorithm by Hübschle-Schneider & Sanders [36]. That algorithm uses the fact that weighted sampling without replacement is equivalent to generating an exponential random variate for each data point, then selecting the point(s) with minimal variates. Importantly, variates can be drawn in parallel batches using single instruction multiple data (SIMD) instructions, providing instruction-level parallelism. Specifically, we use the SIMD-accelerated Polynomial Congruential Generator (PCG) SIMDPCG [37, 38]. Because variates are drawn independently for each point, minicorecan additionally use multiple simultaneous threads to generate variates in parallel across processors.

While drawing the random variates involves a computationally expensive logarithm, we used the SLEEF library to compute batches of logarithms accurately and in parallel using SIMD instructions. As described in [36], exponential random variates can be sampled equivalently either by generating a random value $v \sim U(0, 1)$ and exponentiating by the inverse of the weight $v^{\frac{1}{w}}$, or, equivalently logging and dividing by the weight $\frac{-\ln v}{w}$, which is more numerically stable. We found this numerically stable alternative to be about 3 times as fast as exponentiating.

It is common for k-means++implementations to select more than one

potential new center in a single step, ultimately choosing the center that yields the lowest overall cost. Our parallel implementation accomplishes this using a per-thread heap data structure. SIMD instructions are used to determine which from among the random variates in a chunk are small enough to be added to the heap. If any are small enough, a serial loop extracts the variates and adds them. As a thread proceeds along the array of variates, heap updates become rarer, allowing the vast majority of the computation to remain SIMD parallelized. Finally, the samples in the per-thread heaps are combined to obtain an overall sample.

We can eliminate a significant number of branches in building the heap using Population Counts (popcount) and Count Trailing Zeros (CTZ) instructions. For each vector of new candidate variates, we compare it to the broadcasted ceiling, convert to a bitmask, and popcount, and switch on the value of the popcount, performing the heap update once per nonzero in the bitmask. We access the "current" bit by counting trailing zeros and indexing the relevant variate.

This sampling kernel is a core feature of our library, accessible with C and C++ APIs in the free and MIT-licensed [6] library. While we described the sampling approach in the context of *k*-means++, it also forms the core of the localsearch++ algorithm described below.

5.6.1.2 localsearch++.

Lattanzi and Sohler suggested an augmentation of *k*-means++that adds sampling with local search heuristics [39]. At each iteration in localsearch++, the

Name	Abbreviation	Formula
Squared Euclidean	SQE	$\sum_i (X_i - Y_i)^2$
Kullbeck-Liebler Divergence	KLD	$\sum_i \hat{X}_i \times \log \frac{\hat{X}_i}{\hat{Y}_i}$
Jensen-Shannon Divergence	JSD	$\frac{1}{2} \times (KLD(X, \frac{X+Y}{2}) + KLD(Y, \frac{X+Y}{2}))$
Bhattacharyya Metric	BATMET	$\sqrt{1 - \sqrt{X} \cdot \sqrt{Y}}$

Table 5.2: Formulas for distance measures implemented in minicore. Let X_i, Y_i denote the i^{th} observation (gene) for cells X and Y. Let $\hat{X}_i$ and $\hat{Y}_i$ denote the scaled (normalized by total cell-wide count) version of this entry. These all belong to the class of Bregman divergences, except the JSD which is a convex combination of Bregman divergences.

center whose removal increases the objective the least is removed, and a new point is sampled in proportion to its cost. We re-use the previous sampling kernel to implement the weighted sampling required by this approach. To our knowledge, this is the first application of localsearch++ to distance measures beyond squared Euclidean distance.

5.6.2 Distances, and sparsity in minicore

While k-means++is most commonly implemented using squared Euclidean distance, it has also been shown that the k-means++procedure yields a $\mathcal{O}(\log k)$-approximate solution in expectation when using other distance measures and divergences [3]. Specifically, this applies to the class known as Bregman divergences. This class includes relevant measures such as Kullback-Leibler Divergence (KLD), Squared Euclidean distance (SQE), Itakura-Saito divergence. Other relevant distances are convex combinations of these (Jensen-Shannon Divergence) and . Given this fact, we decided to implement the four distance measures detailed in Table 5.2.

An important concern when implementing these other measures is how they handle 0 values in the data matrix. KL Divergences can be infinite for zero-valued entries, and other measures can have issues with numerical stability in these cases. This can be addressed by the use of a "prior" [40] of a Gamma(β, β) distribution, with a value $\beta > 0$. The *a posteriori* estimates are then $N_i + \beta$, ensuring no 0-valued entries. These are effectively "pseudo-counts," a common way to adjust scRNA-seq data. Selecting $\beta = 1$ corresponds to a Dirichlet prior, while smaller values will penalize missing or low-count observations, and larger values will move points closer together for probability distribution-based distances.

This in turn creates another concern: a matrix adjusted by the prior will have no zero-valued entries, essentially becoming a dense matrix. This greatly increases the space and time required, making these distances impractical for large datasets. We instead compute distances with a lazy prior adjustment for all features, accounting for the zero-count features in aggregate. This is particularly advantageous for sparse matrices with a small number of nonzero values (nnz). In particular, we can perform distance computations in $\mathcal{O}(nnz)$ space and time rather than $\mathcal{O}(d)$, where d is the number of features. The general pseudocode for our distance computations is in Algorithm 1 3. For perspective, the 4-million cell dataset with 63,561 columns would require 960GiB of memory, nearly 100 times the 9.8GiB of the Compressed-Sparse Row ("CSR") representation when using 16-bit indices and data fields. In this way, minicorecan cluster atlas-scale datasets in reasonable working memory, operating directly on the sparse data.

5.6.3 Other optimizations

While minicorecan cluster datasets in a fraction of the space the dense instantiation would require, it can scale even further while managing memory requirements through the use of memory-mapping. This can be done in Python by loading the input data from disk via *numpy.memmap* instead of *numpy.fromfile*, applied either to the original matrix (in the case of dense data) or on the "data", "indices", and "indptr" arrays (in the case of CSR arrays).

Because these arrays are often traversed in predictable fashion, typically sequential, we can off-load to disk, running transparently on datasets which significantly exceed machine RAM even in compressed form.

We also use memory-mapping by default in localsearch++, as an array of size (k, n_{points}) may exceed available memory, and its sequential access patterns are convenient for memory-mapped data.

5.7 Acknowledgments

We thank Daniel Lemire and Wenzel Jakob for their fast SIMD Polynomial Congruential Generator Pseudorandom Number Generators.

Part of this research project was conducted using computational resources at the Maryland Advanced Research Computing Center (MARCC).

Algorithm 3: Generic Algorithm for Sparsity-Preserving distance computations given a prior adjustment β

Result: Distance under prior β
Given: X = (V_x, I_x, N_x)
Y = (V_y, I_y, N_y),
d = dimensionality of data,
and $\beta > 0$;
X and Y are in triple notation, where (V_j, I_j, N_j) represents a compressed-sparse vector's "data", "indices", and "length" fields.
N_{nsnz}: number of shared non-zero fields in the merged pair of vectors
x_i and y_i are indexing variables into left and right sparse vectors
$x_i \leftarrow 0, y_i \leftarrow 0$
B_x and B_y = empty buffers;
$N_{nsnz} = 0$ Number of nonzeros in the merged vector
while $x_i < N_x$ *or* $y_i < N_y$ **do**
    if $x_i < y_i$ **then**
        append(B_x, $(V_x^{x_i} + \beta)$);
        append(B_y, (β));
        $N_{nsnz} \leftarrow N_{nsnz} + 1$
        $x_i \leftarrow x_i + 1$;
    end
    else if $y_i < x_i$ **then**
        append(B_x, (β));
        append(B_y, $(V_y^{y_i} + \beta)$);
        $N_{nsnz} \leftarrow N_{nsnz} + 1$
        $y_i \leftarrow y_i + 1$;
    end
    else
        append(B_x, $(V_x^{x_i} + \beta)$);
        append(B_y, $(y_i^{x_i} + \beta)$);
        $N_{nsnz} \leftarrow N_{nsnz} + 1$
        $x_i \leftarrow x_i + 1; y_i \leftarrow y_i + 1$;
    end
end
return $(d - N_{nsnz}) \times$ distance(0, 0) $+ \sum_{x,y \in (B_x, B_y)} distance(x, y)$

References

[1] D. N. Baker, N. Dyjack, V. Braverman, S. C. Hicks, and B. Langmead. "k-means clustering with various distances". In: *ACM BCB* 2021 (2021).

[2] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: *SODA*. SODA '07 (2007), 1027–1035.

[3] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. "Clustering with Bregman Divergences". In: *Journal of Machine Learning Research* 6.58 (2005), pp. 1705–1749. URL: http://jmlr.org/papers/v6/banerjee05b.html.

[4] Stephanie C. Hicks, Ruoxi Liu, Yuwei Ni, Elizabeth Purdom, and Davide Risso. "mbkmeans: Fast clustering for single cell data using mini-batch k-means". In: *PLOS Computational Biology* 17.1 (2021), pp. 1–18. DOI: 10.1371/journal.pcbi.1008625. URL: https://doi.org/10.1371/journal.pcbi.1008625.

[5] Mario Lucic, Olivier Bachem, and Andreas Krause. "Strong Coresets for Hard and Soft Bregman Clustering with Applications to Exponential Family Mixtures". In: *CoRR* (2016). arXiv: 1508.05243 [stat.ML].

[6] Daniel Baker. *libsimdsampling*. http://github.com/dnbaker/libsimdsampling. 2008.

[7] J. Cao, M. Spielmann, X. Qiu, X. Huang, D. M. Ibrahim, A. J. Hill, F. Zhang, S. Mundlos, L. Christiansen, F. J. Steemers, C. Trapnell, and J. Shendure. "The single-cell transcriptional landscape of mammalian organogenesis". In: *Nature* 566.7745 (2019), pp. 496–502.

[8] Paul Datlinger, André F Rendeiro, Thorina Boenke, Thomas Krausgruber, Daniele Barreca, and Christoph Bock. "Ultra-high throughput single-cell RNA sequencing by combinatorial fluidic indexing". In: *bioRxiv* (2019). DOI: 10.1101/2019.12.17.879304. eprint: https://www.biorxiv.org/content/early/2019/12/18/2019.12.17.879304.

full.pdf. URL: https://www.biorxiv.org/content/early/2019/12/18/2019.12.17.879304.

[9] O. Rozenblatt-Rosen, M. J. T. Stubbington, A. Regev, and S. A. Teichmann. "The Human Cell Atlas: from vision to reality". In: *Nature* 550.7677 (2017), pp. 451–453.

[10] Stuart P. Lloyd. "Least squares quantization in pcm". In: *IEEE Transactions on Information Theory* 28 (1982), pp. 129–137.

[11] Xinlei Chen and Deng Cai. "Large Scale Spectral Clustering with Landmark-Based Representation". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI'11. San Francisco, California: AAAI Press, 2011, 313–318.

[12] Yuanyuan Wei, Julian Jang-Jaccard, Fariza Sabrina, and Timothy R. McIntosh. "MSD-Kmeans: A Novel Algorithm for Efficient Detection of Global and Local Outliers". In: *CoRR* abs/1910.06588 (2019). arXiv: 1910.06588. URL: http://arxiv.org/abs/1910.06588.

[13] Euijoon Ahn, Ashnil Kumar, Dagan Feng, Michael J. Fulham, and Jinman Kim. "Unsupervised Feature Learning with K-means and An Ensemble of Deep Convolutional Neural Networks for Medical Image Classification". In: *CoRR, arXiv:1906.03359* (2019). arXiv: 1906.03359.

[14] Dan Feldman and Michael Langberg. "A Unified Framework for Approximating and Clustering Data". In: *CoRR* abs/1106.1379 (2011). arXiv: 1106.1379. URL: http://arxiv.org/abs/1106.1379.

[15] B. Hie, H. Cho, B. DeMeo, B. Bryson, and B. Berger. "Geometric Sketching Compactly Summarizes the Single-Cell Transcriptomic Landscape". In: *Cell Syst* 8.6 (2019), pp. 483–493.

[16] B. DeMeo and B. Berger. "Hopper: a mathematically optimal algorithm for sketching biological data". In: *Bioinformatics* 36 (2020), pp. i236–i241.

[17] Wei Yang, Jeffrey Bilmes, and William Stafford Noble. "Submodular Sketches of Single-Cell RNA-Seq Measurements". In: *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. BCB '20. Virtual Event, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3388440.3412409. URL: https://doi.org/10.1145/3388440.3412409.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[19] Andrei Novikov. "PyClustering: Data Mining Library". In: *Journal of Open Source Software* 4.36 (2019), p. 1230. DOI: 10.21105/joss.01230. URL: https://doi.org/10.21105/joss.01230.

[20] F. W. Townes, S. C. Hicks, M. J. Aryee, and R. A. Irizarry. "Feature selection and dimension reduction for single-cell RNA-Seq based on a multinomial model". In: *Genome Biol* 20.1 (2019), p. 295.

[21] Charles Elkan. "Using the Triangle Inequality to Accelerate K-Means". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. Washington, DC, USA: AAAI Press, 2003, 147–153. ISBN: 1577351894.

[22] Andrei Broder, Lluis Garcia-Pueyo, Vanja Josifovski, Sergei Vassilvitskii, and Srihari Venkatesan. "Scalable K-Means by Ranked Retrieval". In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. WSDM '14. New York, New York, USA: Association for Computing Machinery, 2014, 233–242. ISBN: 9781450323512. DOI: 10.1145/2556195.2556260. URL: https://doi.org/10.1145/2556195.2556260.

[23] Qiuhong Li, Peng Wang, Wei Wang, Hao Hu, Zhongsheng Li, and Junxian Li. "An Efficient K-means Clustering Algorithm on MapReduce". In: *Database Systems for Advanced Applications*. Ed. by Sourav S. Bhowmick, Curtis E. Dyreson, Christian S. Jensen, Mong Li Lee, Agus Muliantara, and Bernhard Thalheim. Cham: Springer International Publishing, 2014, pp. 357–371. ISBN: 978-3-319-05810-8.

[24] Konstantin Makarychev, Yury Makarychev, and Ilya P. Razenshteyn. "Performance of Johnson-Lindenstrauss Transform for k-Means and k-Medians Clustering". In: *CoRR* abs/1811.03195 (2018). arXiv: 1811.03195. URL: http://arxiv.org/abs/1811.03195.

[25] Shibiao Wan, Junil Kim, and Kyoung Jae Won. "SHARP: hyper-fast and accurate processing of single-cell RNA-seq data via ensemble random projection". In: *Genome Research* (2020). DOI: 10.1101/gr.254557.119. eprint: http://genome.cshlp.org/content/early/2020/01/28/

gr.254557.119.full.pdf+html. URL: http://genome.cshlp.org/content/early/2020/01/28/gr.254557.119.abstract.

[26] Leon Bottou and Yoshua Bengio. "Convergence properties of the k-means algorithms". In: *Advances in neural information processing systems*. 1995, pp. 585–592.

[27] D. Sculley. "Web-Scale k-Means Clustering". In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, 1177–1178. ISBN: 9781605587998. DOI: 10.1145/1772690.1772862. URL: https://doi.org/10.1145/1772690.1772862.

[28] G. X. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, T. D. Wheeler, G. P. McDermott, J. Zhu, M. T. Gregory, J. Shuga, L. Montesclaros, J. G. Underwood, D. A. Masquelier, S. Y. Nishimura, M. Schnall-Levin, P. W. Wyatt, C. M. Hindson, R. Bharadwaj, A. Wong, K. D. Ness, L. W. Beppu, H. J. Deeg, C. McFarland, K. R. Loeb, W. J. Valente, N. G. Ericson, E. A. Stevens, J. P. Radich, T. S. Mikkelsen, B. J. Hindson, and J. H. Bielas. "Massively parallel digital transcriptional profiling of single cells". In: *Nat Commun* 8 (2017), p. 14049.

[29] J. Cao, D. R. O'Day, H. A. Pliner, P. D. Kingsley, M. Deng, R. M. Daza, M. A. Zager, K. A. Aldinger, R. Blecher-Gonen, F. Zhang, M. Spielmann, J. Palis, D. Doherty, F. J. Steemers, I. A. Glass, C. Trapnell, and J. Shendure. "A human cell atlas of fetal gene expression". In: *Science* 370.6518 (2020).

[30] Naoki Shibata and Francesco Petrogalli. "SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions". In: *IEEE Transactions on Parallel and Distributed Systems* 31.6 (2020), 1316–1327. ISSN: 2161-9883. DOI: 10.1109/tpds.2019.2960333. URL: http://dx.doi.org/10.1109/TPDS.2019.2960333.

[31] Stuart P. Lloyd. "Least squares quantization in PCM". In: *IEEE Trans. Information Theory* 28 (1982), pp. 129–136.

[32] P. Brennecke, S. Anders, J. K. Kim, A. A. Kołodziejczyk, X. Zhang, V. Proserpio, B. Baying, V. Benes, S. A. Teichmann, J. C. Marioni, and M. G. Heisler. "Accounting for technical noise in single-cell RNA-seq experiments". In: *Nat Methods* 10.11 (2013), pp. 1093–1095.

[33] Maria-Florina F Balcan, Steven Ehrlich, and Yingyu Liang. "Distributed *k*-means and *k*-median Clustering on General Topologies". In: *Advances in Neural Information Processing Systems* 26 (2013), pp. 1995–2003.

[34] Wei Yang, Jacob Schreiber, Jeffrey Bilmes, and William Stafford Noble. "Submodular sketches of single-cell RNA-seq measurements". In: *bioRxiv* (2020). DOI: 10.1101/2020.05.01.066738. eprint: https://www.biorxiv.org/content/early/2020/05/07/2020.05.01.066738.full.pdf. URL: https://www.biorxiv.org/content/early/2020/05/07/2020.05.01.066738.

[35] Deanna Needell, Nathan Srebro, and Rachel Ward. *Stochastic Gradient Descent, Weighted Sampling, and the Randomized Kaczmarz algorithm*. 2015. arXiv: 1310.5715 [math.NA].

[36] Lorenz Hübschle-Schneider and Peter Sanders. "Communication-Efficient (Weighted) Reservoir Sampling from Fully Distributed Data Streams". In: *CoRR* (2020). arXiv: 1910.11069 [cs.DS].

[37] Daniel Lemire. *SIMDPCG*. https://lemire.me/blog/2018/06/07/vectorizing-random-number-generators-for-greater-speed-pcg-and-xorshift128-avx-512-edition/. 2016-2018.

[38] Wenzel Jakob Daniel Lemire. *SIMDPCG*. https://github.com/lemire/simdpcg. 2013.

[39] Silvio Lattanzi and Christian Sohler. "A Better k-means++ Algorithm via Local Search". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3662–3671. URL: http://proceedings.mlr.press/v97/lattanzi19a.html.

[40] Daniela M. Witten. "Classification and clustering of sequencing data using a Poisson model". In: *The Annals of Applied Statistics* 5.4 (2011), 2493–2518. ISSN: 1932-6157. DOI: 10.1214/11-aoas493. URL: http://dx.doi.org/10.1214/11-AOAS493.

Chapter 6

Discussion and Conclusion

In this thesis, we have leveraged randomness, exploited sparsity, inverted indexes, and hardware improvements to provide practical, efficient tools capable of scaling to both enormous datasets and large collections thereof.

We used Sketching and MinHash to practically analyze large genomes, and we applied coresets to large collections, from sparse single-cell expression data to real-world road networks and facility location problems.

In the future, we see both many possible applications and extensions of these methods.

6.1 Applications

6.1.1 Farther downstream

In our method, we primarily use the MinHash sketches as the final structure being compared. Many applications, however, use sketching to pre-filter candidates for more exhaustive analysis. For example, Dashing2 uses an LSH table to generate candidates with either OrderMinHash or k-mer set minhash,

and can be instructed to emit near neighbors as evaluted by edit distance. This could be easily extended to dynamic programming, multiple sequence alignment, or genome mapping after reference selection.

6.1.2 Sequence MinHash Applications

We see sequence similarity search via edit distance LSH as a primary future application. For instance, when grouping protein sequences or small ribosomal subunit sequences, an the edit distance LSH is more specific than a k-mer MinHash LSH. We could use dynamic programming alignment as the final step instead of edit distance. If we had more specific LSH functions for dynamic programming alignment (compared to edit distance), this could perform even better.

These techniques may be applicable to improving seeding for sequence aligners. These could be inserted into graph minimizer indexes (in place of graph *k*-mers), or involve weighting of minimizers for linear genomes such as WinnowMap.

Lastly, these could also serve as part of bait design and sequence subset selection techniques. For computational biology applications, often one must design bait sequences for molecular assays, to which only sequences close to the complement of the bait will bind. Using either edit distance LSH (for selecting subsets of sequences) or k-mer set LSH, we could very quickly select a set of non-redundant sequences to include. An orthogonal approach would be to generate set of sampled minimizers from a collection of sequences. For either of these approaches, minimizer weighting schemes related to WinnowMap

might yield practical improvements. For instance, entropy-weighted minimizers might generate more easily-detected sequence minimizers in long-read technology.

6.2 Method Improvements

6.2.1 MinHash Improvements

The next steps to push forward Locality-Sensitive Hashing applications lies in moving the sampling further into the problem at hand. Winnowmap [1] provides a useful example. They weight *k*-mers by their occurrence count, so as the more often select rarer *k*-mers for seeding. In Dashing2, we provide an entropy-weighted option for selection, which tends to select higher-information items.

The edit distance LSH [2] is particularly important. For many downstream applications, *k*-mers which match have different affinities. For dynamic programming alignment, certain pairs of residues give higher or lower scores. A family of LSH functions for dynamic programming alignment would be a crucial next step. We suspect that weighting OrderMinHash keys according to the match score given a specific scoring matrix could yield such a method.

6.2.2 LSH Table Improvements

While our LSH tables built on hash tables are very efficient and work correctly, they do require a substantial amount of memory, and the randomized nature of hash table probing causes many cache misses. We suspect that building on the storage and query formats that [3] uses would improve speed, especially

for bulk lookups. This also would give us the flexibility to not choose a-priori the exact number of registers we wish to match at a time.

Additionally, we investigated LSH tables for the generalized Jensen-Shannon divergence [4], but found that naive tables did not significantly improve lookups. We suspect that this may be related to the simplicity of our method. If more flexible trie-based matching is implemented and more candidates are considered, we suspect that we could build effective tables for nearest neighbors over this distance.

6.2.3 Count Vector Sketching

For our work with sketching streaming read sets, we used feature hashing to limit the memory requirements of the streaming inputs.

We could incorporate improvements in some randomized structures; we could take generalizations or improvements of the Count-Min sketch, such as the weighted median sketch [5] and counting quotient filter [6], respectively. Approximate-counting variants [7] can be used to reduce storage, and bulk updates can amortize memory page requests.

Alternatively, we could modify the counting structure so as to prioritize counts of certain relative magnitudes. We could use the Count-Sketch, which would prioritize preserving the quantities of heavy-hitting algorithms, at the expense of lower-count items. The Heavy Keeper sketch [8] prioritizes tracking the highest-magnitude options. While its worst-case behavior has no guarantees, it might be good for applications with large streams with few but large heavy hitters.

6.2.4 Bitwise Interleaving

Further performance improvements could be gained by interleaving bits between signatures. This is the technique used in BinDash which allowed it to sometimes surpass Dashing2's comparison speed.

This interleaving process allows comparison computation to stop once all bits in all signatures have diverged. This algorithm, for comparing b-bit signatures for equality, could be extended to greater than and less than calculations. We will leave this step to future work, as Dashing2 is still very efficient.

References

[1] C. Jain, A. Rhie, H. Zhang, C. Chu, B. P. Walenz, S. Koren, and A. M. Phillippy. "Weighted minimizer sampling improves long read mapping". In: *Bioinformatics* 36.Suppl_1 (2020), pp. i111–i118.

[2] G. Marçais, D. DeBlasio, P. Pandey, and C. Kingsford. "Locality-sensitive hashing for the edit distance". In: *Bioinformatics* 35.14 (2019), pp. i127–i135.

[3] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. "PUFFINN: Parameterless and Universally Fast FInding of Nearest Neighbors". In: *CoRR* abs/1906.12211 (2019). arXiv: 1906.12211. URL: http://arxiv.org/abs/1906.12211.

[4] Lin Chen, Hossein Esfandiari, Thomas Fu, and Vahab S. Mirrokni. *Locality-Sensitive Hashing for f-Divergences: Mutual Information Loss and Beyond*. 2019. arXiv: 1910.12414 [cs.LG].

[5] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. "Finding Heavily-Weighted Features in Data Streams." In: *CoRR* abs/1711.02305 (2017). URL: http://arxiv.org/abs/1711.02305.

[6] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. "A General-Purpose Counting Filter: Making Every Bit Count". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, 775–787. ISBN: 9781450341974. DOI: 10.1145/3035918.3035963. URL: https://doi.org/10.1145/3035918.3035963.

[7] Jelani Nelson and Huacheng Yu. "Optimal bounds for approximate counting". In: *CoRR* abs/2010.02116 (2020). arXiv: 2010.02116. URL: https://arxiv.org/abs/2010.02116.

[8] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 909–921. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/gong.