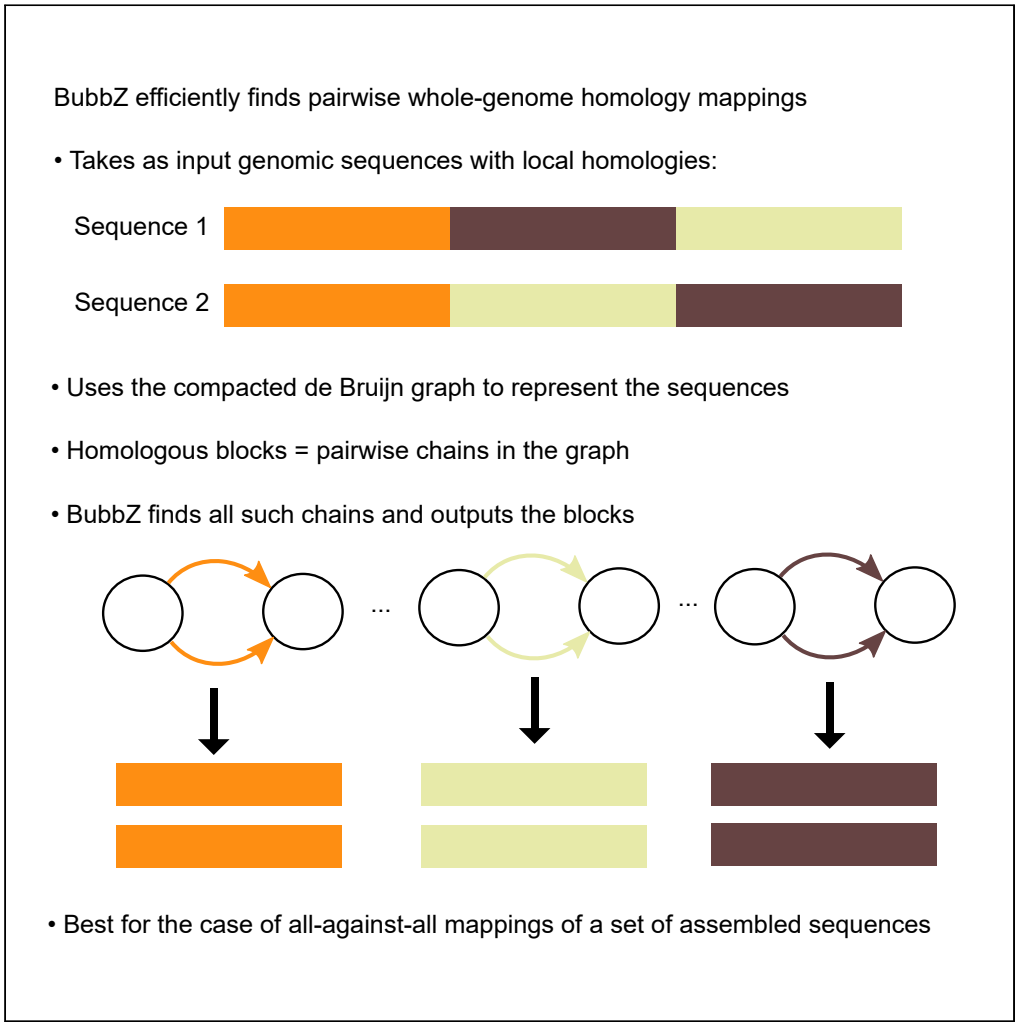
iScience

CellPress
OPEN ACCESS

Article

Scalable Pairwise Whole-Genome Homology Mapping of Long Genomes with BubbZ



Ilia Minkin, Paul Medvedev

ivminkin@gmail.com (I.M.)
pashadag@cse.psu.edu (P.M.)

HIGHLIGHTS

BubbZ is a fast whole-genome homology mapper

Works by finding pairwise chains in the compacted de Bruijn graph

Optimized for all-against-all mappings between multiple assembled sequences

Tested on 16 mice and 1,600 Salmonella, offers up to an order of magnitude speed-up

Minkin & Medvedev, iScience *23*, 101224
June 26, 2020
https://doi.org/10.1016/j.isci.2020.101224

iScience

CellPress
OPEN ACCESS

Article

Scalable Pairwise Whole-Genome Homology Mapping of Long Genomes with BubbZ

Ilia Minkin[1,4,*] and Paul Medvedev[1,2,3,*]

SUMMARY

Pairwise whole-genome homology mapping is the problem of finding all pairs of homologous intervals between a pair of genomes. As the number of available whole genomes has been rising dramatically in the last few years, there has been a need for more scalable homology mappers. In this paper, we develop an algorithm (BubbZ) for computing whole-genome pairwise homology mappings, especially in the context of all-to-all comparison for multiple genomes. BubbZ is based on an algorithm for computing chains in compacted de Bruijn graphs. We evaluate BubbZ on simulated datasets, a dataset composed of 16 long mouse genomes, and a large dataset of 1,600 Salmonella genomes. We show up to approximately an order of magnitude speed improvement, compared with MashMap2 and Minimap2, while retaining similar accuracy.

INTRODUCTION

Pairwise whole-genome homology mapping is the problem of finding all pairs of homologous intervals between a pair of genomes (see Dewey and Pachter (2006) for a discussion about the precise meaning of homology). Unlike local pairwise alignment, which provides base-to-base homology resolution, mapping only computes the boundaries of homologous blocks. This is, however, sufficient for many applications. For example, whole-genome homology mapping is a starting point for the analysis of genome rearrangements, which themselves are used in studies of breakpoint reuse (Pevzner and Tesler, 2003) and phylogenetics (Luo et al., 2012). It can be used as a precursor to whole-genome alignment (Dewey and Pachter, 2006; Armstrong et al., 2019) or for exploratory comparative analysis. It is also used as a tool for quality control of genome assembly (Vollger et al., 2019) and for identifying genomic duplications for the purposes of improving RNA mapping (Srivastava et al., 2019).

A straightforward solution to compute such a homology map is to do pairwise local alignment; however, alignment is a harder computational problem, requiring more resources than other more direct approaches. A related problem is locally collinear block reconstruction, where mapping blocks can have multiple instances and the overlap between them is fully resolved (Darling et al., 2004, 2010; Dewey, 2007; Paten et al., 2008; Pham and Pevzner, 2010; Minkin et al., 2013). Although solutions to this problem can be used to compute pairwise mappings, known methods perform poorly in regions with complex repeat structures (Minkin and Medvedev, 2019). The first direct approach to address the whole-genome homology map problem was chaining of smaller fragments using a line-sweeping approach inspired by computational geometry (Abouelhoda et al., 2008; Abouelhoda and Ohlebusch, 2005; Myers and Miller, 1995; Ohlebusch and Abouelhoda, 2006). Another approach treats sequences such as audio signals and uses cross-correlations to find homology (Grabherr et al., 2010).

However, the number of available whole genomes has been rising dramatically in the last few years, creating the need for more scalable homology mappers. Two recent tools are particularly notable in tackling this challenge; although both are known for read alignment, they also compute homology maps. Minimap2 (Li, 2018) is based on a seed-and-extend approach but using minimizers (Roberts et al., 2004) to quickly identify and reduce the number seeds. MashMap2 (Jain et al., 2018) uses a minimizer winnowing scheme to quickly identify candidates. These methods are able to compute the mapping of two mammalian-sized genomes in less than an hour. However, in a scenario where the input is a set of multiple genomes and each genome has to be mapped to every other one, even these methods can be too slow. With efforts such as the Vertebrate Genomes Project and Insect 5K promising to release thousands more genomes in the future, more scalable approaches will be needed.

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA

[2]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, PA 16802, USA

[3]Center for Computational Biology and Bioinformatics, The Pennsylvania State University, University Park, PA 16802, USA

[4]Lead Contact

*Correspondence: ivminkin@gmail.com (I.M.), pashadag@cse.psu.edu (P.M.)

https://doi.org/10.1016/j.isci.2020.101224

The approach we take in this paper is based on the compacted de Bruijn graph. This graph provides an efficient representation of the shared *k*-mers between closely related genomes, whereby potentially long shared sequences are represented by small structures within the graph. Approaches based on such graphs had already proven useful to construct synteny blocks (Pham and Pevzner, 2010; Minkin et al., 2013), but recent breakthroughs in the efficiency of graph construction algorithms (Marcus et al., 2014; Chikhi et al., 2016; Baier et al., 2016; Minkin et al., 2017) make them a promising approach for homology mapping. The latest methods can construct the graph for tens of mammalian genomes in minutes rather than weeks and could construct the graph for 100 human genomes in less than a day (Minkin et al., 2017).

In this paper, we propose BubbZ, an algorithm for computing whole-genome pairwise homology mappings, especially in the context of all-to-all comparison for multiple genomes. Our algorithm is based on ideas similar to the line-sweep algorithms (Abouelhoda et al., 2008; Abouelhoda and Ohlebusch, 2005; Ohlebusch and Abouelhoda, 2006) but allows for more efficient data structures that reduce the running time. We evaluated our method on both simulated datasets and on a large dataset composed of 16 mouse genomes, as well as on a dataset consisting of 1,600 Salmonella genomes. BubbZ shows up to approximately an order of magnitude speed improvement on the datasets of hundreds of bacteria and up to three times on the mice dataset, compared with MashMap2 and Minimap2, while retaining similar accuracy.

RESULTS

Algorithm Overview

Given a set of collection of sequences, BubbZ outputs homology information between all possible pairs of sequences in the collection. This includes homology information between a sequence and itself. Given two sequences *s* and *t*, BubbZ considers a homology, informally, to be a region of *s* and a region of *t* whose *k*-mer sequences are identical except for gaps of at most *b* *k*-mers (*b* is a parameter). BubbZ outputs the coordinates of all maximal homologies between *s* and *t*, except that if the regions of two homologies have the same right endpoints, only one with the most shared *k*-mers is output. In case of ties, BubbZ favors outputting the one with smaller gaps in *t*, roughly speaking. For a more precise and formal description of BubbZ, please see the Methods section.

Datasets

We evaluated BubbZ speed and accuracy on three datasets, the first based on long real mouse genomes, the second based on a large amount of short bacterial genomes, and the third containing short simulated genomes. For the long real genomes, we downloaded 16 mouse genomes from GenBank (Benson et al., 2017). These consisted of 15 different strains, assembled as part of a recent study (Lilue et al., 2018), and the mouse reference genome. The mouse reference has 377 scaffolds, whereas the other mouse strains have 2,977–7,154 scaffolds; the genomes' size fluctuates between 2.6 and 2.8 Gbp. To test the scalability of our pipeline in the number of genomes, we created four datasets from these 16 genomes. The four datasets contain genomes 1-2, 1-4, 1-8, and 1-16, respectively, with genome one being the reference genome. More details about the datasets, including accession numbers, are available as Table S1 in (Minkin and Medvedev, 2019).

The real bacterial dataset consisted of 1,600 Salmonella genomes that are a part of GenomeTrakr project (NCBI BioProject ID, 183844), a public effort to identify and track pathogens causing food-borne illness. Each genome consisted of approximately 4.6 million basepairs. As in the mouse experiment, we created four datasets containing genomes 1-200, 1-400, 1-800, and 1-1,600 genomes, respectively. Link to the ordered list of the genomes containing their RefSeq accession numbers is contained in the "Data and Code Availability" section. The goal of this dataset was to test scalability with respect to the number, rather than the length, of the genomes.

The other type of data we used were nine simulated datasets, generated as part of our earlier study (Minkin and Medvedev, 2019), and are available for download at https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt. Each dataset is an evolution simulation from a single ancestor genome, composed of 1,500 genes and of size approximately 1.5 Mbp; the result is ten genomes in each dataset. The datasets are distinguished by their divergence, with the evolutionary distance from the root genome to the leaves varying between 0.03 and 0.25 substitutions per site.

Evaluated Tools

We compared BubbZ against the two recent tools that are able to scale to the size of modern datasets, Minimap2 (Li, 2018) and MashMap2 (Jain et al., 2018). We ran all tools in order to produce an all-against-all mapping, including any duplications (i.e. mappings within a single genome or chromosome).

A common parameter for homology-finding tools is the minimum size of the block in the output. We tried to make the evaluated tools to generate blocks of comparable sizes but it was not possible due to the difference in the algorithmic approaches and implementations. We made BubbZ output blocks of at least 200bp. For MashMap2 we produced blocks of length at least 500bp (lowest possible setting) for bacteria and 5000bp (the default value) for mice. We used different values for MashMap2 because on the simulated bacteria dataset the default parameters produced insufficient recall. This setting is not applicable to Minimap2 because it uses alignment scores for cutoffs rather than block lengths, so we used parameters suggested by the author for all datasets.

All parameters and command lines are available at https://github.com/medvedevgroup/BubbZ/blob/master/supplementary.txt, but we highlight the important ones here. To run BubbZ, we first ran TwoPaco (Minkin et al., 2017) to construct the graph, using $k = 21$for real datasets and $k = 15$ for the smaller simulated ones. We then ran BubbZ using $b = 300$, $m = 200$, and for all datasets. The role of these parameters was explored in the context of multiple whole-genome alignment of the same datasets (Minkin and Medvedev, 2019), and we used values that were found to work best in that paper. Please refer to Minkin and Medvedev (2019) for guidance on how these parameters can be chosen and what the tradeoffs involved are.

Neither Minimap2 nor MashMap2 provide a ready-made option to compute all-against-all pairwise mappings for a collection of genomes: a user has to run the tools for each pair of genomes separately. For Minimap2 and MashMap2, we wrote a wrapper that created a separate run for each of the genome pairs; to find duplications, we also ran it on each genome separately. This wrapper can be parallelized in two ways: (1) the runs can be executed in parallel, and (2) each run can be internally parallelized by the respective tool. Assigning different amount of threads to "external" and "internal" parallelization leads to different trade-offs between running time and memory. We tried to minimize the overall running time of the mappings while keeping the peak memory usage reasonable. As we had 24 threads available, for Minimap2 and MashMap2 we decided to run six pairs of mappings simultaneously and allowed each tool to use four threads internally on the mice dataset. On the bacterial datasets, we used all 24 threads for external parallelization. For BubbZ we used all 24 threads internally because it natively supports all-against-all mappings and for TwoPaCo we used 16 threads as suggested by the documentation.

For mapping different genomes with Minimap2, we used default presets for sequences of 5% divergence. For mapping genomes against themselves, we used the parameters suggested by the author. For MashMap2 we used the default parameters, except (1) the minimum block size for bacterial genomes as described earlier, (2) that for mapping different genomes we used the orthologous filtering, whereas for computing duplications we disabled the filtering, as suggested by the authors.

Evaluation Metrics

For the smaller simulated dataset we computed both recall and precision using the mafTools package (Earl et al., 2014). This package requires an alignment for comparison, rather than just a map; we therefore took each one of the homology blocks in our output and computed an alignment of it using LAGAN (Brudno et al., 2003). To define precision and recall, mafTools views an alignment as an equivalence relation, which is the set of all equivalent position pairs participating in the true alignment. Let A denote the relation produced by an alignment algorithm, and let H denote the ground truth alignment relation (in our case, H is given by the simulator). The accuracy of A is then given as $recall(A) = 1 - |H \setminus A|/|H|$ and $precision(A) = 1 - |A \setminus H|/|A|$.

For the larger mouse dataset, there are unfortunately no ground-truth whole-genome homology maps or alignments available, making it difficult to evaluate precision. To evaluate the recall, we used an alignment of homologous protein-coding genes annotated in Ensembl. These ground-truth alignments, generated as part of our earlier study (Minkin and Medvedev, 2019) using LAGAN, are available for download at https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt. The alignment contains both orthologous and paralogous gene pairs, although most of the paralogous pairs come from the well-annotated mouse

	TwoPaCo + BubbZ			Minimap2	MashMap2
Dataset	TwoPaCo	BubbZ	Total		
1–2	15 (9.3)	6 (35.2)	21 (35.2)	73 (46.5)	233 (22.3)
1–4	22 (9.4)	14 (66.5)	36 (66.5)	75 (105.4)	240 (39.7)
1–8	40 (9.3)	26 (94.9)	66 (94.9)	104 (119.2)	464 (44.7)
1–16	83 (17.8)	42 (164.2)	125 (164.2)	411 (119.6)	1,530 (45.6)

Table 1. Running Time (Minutes) and Memory Usage (Gigabytes, in Parenthesis) on the Mouse Data

reference genome. For the purposes of analysis, we binned the pairs of homologous genes according to the nucleotide identity in their alignment. These alignments cover around 33% of the input genomes, i.e. 33% of base pairs in the input genome are included in the alignment. We could not compute recall using mafTools due to the computational cost of having to compute all alignments for all the homologous intervals in the output. Instead, consider all the aligned position pairs in a ground truth alignment and a homology mapping $\mathcal{C}$. We define recall as the fraction of aligned position pairs for which there exists a block in $\mathcal{C}$ covering both positions. We did not evaluate accuracy on the large bacterial dataset.

Results on the Mouse Data

The running time and memory consumption of all the tools are shown in Table 1. The pipeline consisting of TwoPaCo and BubbZ was 1.5–3 times faster than Minimap2 and 6–12 times faster than MashMap2. Starting at four genomes, we observe roughly linear scaling for BubbZ. For Minimap2 and MashMap the scaling seems superlinear, although it is difficult to make any firm conclusions given the limited number of datapoints. The linear scaling of BubbZ is consistent with the fact that only a linear number of runs is required (see Transparent Methods); however, the time of each run also grows with the size of the input. Nevertheless, we empirically found the scaling to be roughly linear.

For datasets consisting of 2, 4, and 8 mice Minimap2 uses 1.3–1.6 times more memory than BubbZ, whereas for 16 mice BubbZ uses roughly 1.4 times more memory. At the same time, MashMap2 has the lowest memory usage on all datasets: it uses 1.5–3.6 times less memory than BubbZ and 2.1–2.7 times less than Minimap2. We note that for MashMap2 and Minimap2 the peak memory usage is the cumulative peak memory usage of all instances of the tool being run simultaneously by our wrapper.

To compute the recall, we only used the dataset consisting of two genomes, because computing recall is otherwise computationally prohibitive. Figure 1 shows the recall, broken down by nucleotide identity of the gene pairs and by orthology/paralogy. Both versions of BubbZ demonstrate nearly the identical recall scores, with the exact version being marginally better. For the orthologous genes, all mappers have similar recall, although BubbZ has higher recall in genes of lower nucleotide identity. For the paralogous pairs, Minimap2 had slightly higher recall then BubbZ.

Results on the Bacterial Data

The running time and memory consumption of the mapping tools on the Salmonella dataset is shown in Table 3. The total running time of TwoPaCo and BubbZ is 6–12 times smaller than of Minimap2 and 6–9 times than MashMap2. In contrast with the experiment involving the mice dataset, MashMap2 is approximately 1.3 times faster than Minimap2. Both Minimap2 and MashMap2 have comparable memory consumption ($\lesssim$7.0 GB); in contrast, BubbZ consumes a lot more memory due to keeping the graph for the whole dataset in memory.

Results on the Simulated Data

Using simulated data, we can measure the accuracy more thoroughly than we could on real data. Figure 2 shows precision and recall of the three methods, as a function of divergence between genomes. For all tools, both recall and precision decline with increase of the divergence. Both versions of BubbZ demonstrate nearly the identical recall and precision. Recall is similar for all methods, with BubbZ having slightly better values for more divergent datasets. BubbZ and Minimap2 have nearly identical precision curves, and they are substantially higher than MashMap2.

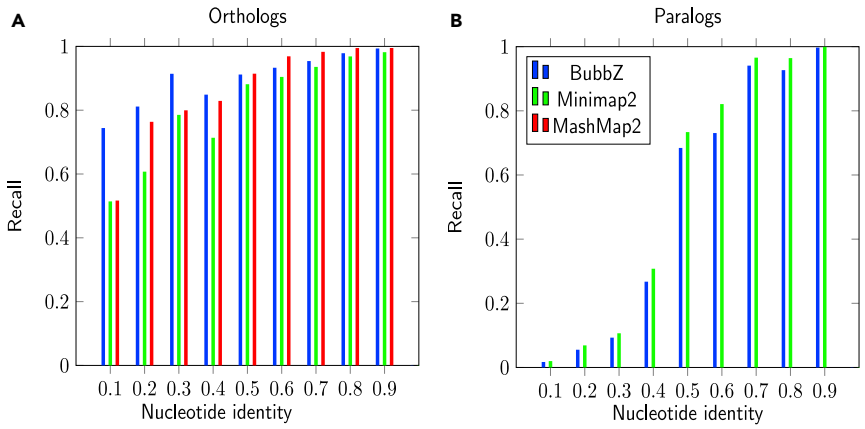


Figure 1. Results on the Mouse Data

Recall of the position pairs belonging to pairs of protein-coding genes by BubbZ(blue), Minimap2(green), and MashMap2(red). (A) corresponds to orthologs and(B) to paralogs. MashMap2 recall on paralogs could not be computed.

Table 2 shows the running time and memory usage, although because of the small size of the datasets, it is hard to draw any conclusions about scalability in the size or number of genomes. However, this did allow us to measure how the divergence affected each of the methods. For BubbZ and MashMap2 genomic divergence did not have a significant effect on the running time, whereas Minimap2 ran slower on more divergent genomes.

DISCUSSION

In this paper, we present BubbZ, a novel method for computing pairwise mapping between complete genomes. Empirical results indicate that for a large collection of bacterial genomes, our method can be up to 10 times faster than competing approaches. On closely related mammalian genomes, our method is also several times faster than competitors, while maintaining similar accuracy. Our approach for finding chains is based on the problem formulation of the sweep-line algorithms from (Ohlebusch and Abouelhoda, 2006). Those algorithms similarly defined chains and a dynamic programming formulation to find the longest chains. The main difference is that the previous work was focused on finding a single optimal chain, while we formally define the problem of finding all such non-redundant chains. In practice users are often interested in computing a set of chains that comprehensively represent homology between input genomes and are not redundant. Most practical solutions address this need by implementing heuristics that choose which chains to output. In contrast, we formally defined and solved a problem of finding all non-redundant optimal chains.

MiniMap2 also uses a chaining strategy but in a slightly different way than BubbZ. One can think of BubbZ as limiting the space of possible predecessors by restricting the gap size by a parameter *b*. MiniMap2, on the other hand, does not limit the gap size in its chaining algorithm. Instead, it explores at most *h* predecessors, where *h* is parameter. In some cases, it might mean that BubbZ explores more predecessors than MiniMap2, whereas in others it could be the other way around.

Dataset	TwoPaCo + BubbZ			Minimap2	MashMap2
	TwoPaCo	BubbZ	Total		
1–200	4 (17.5)	2 (7.8)	6 (17.5)	35 (3.5)	26 (1.6)
1–400	6 (17.5)	6 (16.7)	12 (17.5)	132 (3.5)	101 (1.8)
1–800	10 (17.6)	33 (44.3)	43 (44.3)	510 (4.3)	390 (2.3)
1–1,600	19 (17.8)	257 (149.2)	276 (149.2)	2,250 (7.0)	1876 (2.3)

Table 3. Running Time (Minutes) and Memory Usage (Gigabytes, in Parenthesis) on the Bacterial Data

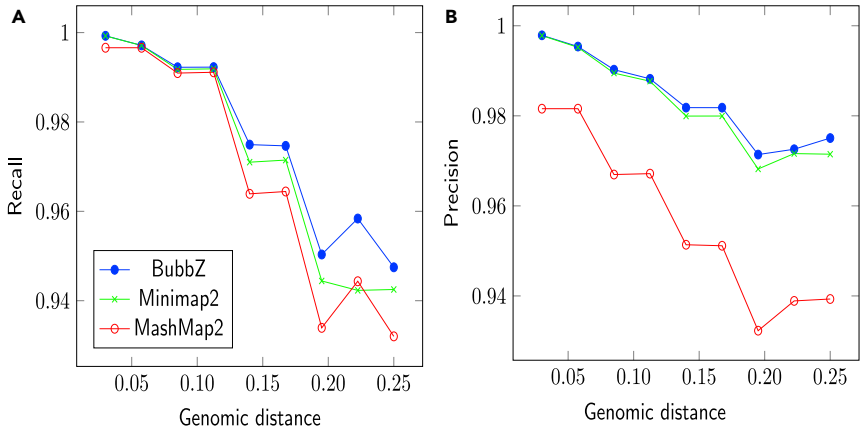


Figure 2. Results on the Simulated Data: Accuracy as a Function of the Genomic Distance
(A) shows recall, and (B) displays precision.

In addition, the approaches mentioned earlier require an efficient dynamic range-maximum-query structure (Abouelhoda et al., 2008; Abouelhoda and Ohlebusch, 2005; Ohlebusch and Abouelhoda, 2006). Although such data structures have theoretically asymptotic logarithmic query times, in practice they have a high constant due to their implementations relying on search trees with extra information. Instead of using a tree-based dynamic index, we rely on the sparseness of the compacted de Bruijn graph and use a simple BitVector-based algorithm and data structure to quickly compute optimal predecessors in the dynamic programming matrix.

Limitations of the Study

One particular limitation of our tool is its high memory usage, due to keeping in memory the graph constructed from all input genomes simultaneously. We believe that it should be possible to reduce the memory usage by developing a more efficient memory representation. One possible approach is a succinct data structure for the compacted de Bruijn graph, similar to recently published work (Almodaresi et al., 2017, 2018; Bowe et al., 2012; Muggli et al., 2017). However, such a representation should contain extra information to permit quick mapping operations required by our algorithm. We also note that we compared BubbZ against tools that are a based on different algorithmic approaches that have different parameters. As a result, it is hard to come up with a set of parameters that result in fair comparison for all the tools. It is particular evident in our attempts to externally parallelize runs of MashMap2 and Minimap2.

Dataset	TwoPaCo + BubbZ			Minimap2	MashMap2
	TwoPaCo	BubbZ	Total		
0.03	7 (1,240)	1 (36)	8 (1,240)	6 (904)	3 (147)
0.06	6 (1,291)	1 (51)	7 (1,291)	8 (820)	3 (154)
0.09	6 (1,246)	1 (74)	7 (1,246)	10 (824)	3 (168)
0.11	6 (1,292)	1 (77)	7 (1,292)	10 (634)	3 (172)
0.14	6 (1,250)	2 (80)	8 (1,250)	15 (1,341)	3 (171)
0.17	6 (1,277)	2 (80)	8 (1,277)	15 (1,340)	3 (157)
0.20	6 (1,238)	2 (82)	8 (1,238)	16 (1,113)	3 (165)
0.22	5 (1,237)	2 (71)	7 (1,237)	16 (614)	4 (164)
0.25	5 (1,207)	2 (82)	7 (1,207)	16 (1,204)	3 (168)

Table 2. Running Time (Seconds) and Memory Usage (Megabytes, in Parenthesis) on the Simulated Data
Each dataset is labeled by its corresponding divergence.

Resource Availability

Lead Contact

Further information and requests for resources should be directed to and will be fulfilled by the Lead Contact, Ilia Minkin (ivminkin@gmail.com).

Materials Availability

This study did not generate new unique reagents.

Data and Code Availability

Our tool is open source and freely available at https://github.com/medvedevgroup/bubbz. All parameters and command lines are available at https://github.com/medvedevgroup/BubbZ/blob/master/supplementary.txt. The nine simulated datasets we used for evaluation as well as the ground-truth alignments for the mouse data are available for download at https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt. The ordered list of accession numbers of 1,600 Salmonella genomes is available at https://github.com/medvedevgroup/BubbZ/blob/master/salmonella_refseq.txt.

METHODS

All methods can be found in the accompanying Transparent Methods supplemental file.

SUPPLEMENTAL INFORMATION

Supplemental Information can be found online at https://doi.org/10.1016/j.isci.2020.101224.

ACKNOWLEDGMENTS

This work has been supported in part by NSF awards DBI-1356529, CCF-1439057, IIS-1453527 to PM.

AUTHOR CONTRIBUTIONS

Conceptualization, IM; Methodology, IM; Software, IM; Validation, IM and PM, Writing—Original Draft, IM; Writing—Review & Editing, IM and PM, Funding Acquisition, PM.

DECLARATION OF INTERESTS

The authors declare no competing interests.

Received: January 20, 2020
Revised: May 25, 2020
Accepted: May 28, 2020
Published: June 26, 2020

REFERENCES

Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. (2008). Coconut: an efficient system for the comparison and analysis of genomes. BMC Bioinformatics *9*, 476.

Abouelhoda, M.I., and Ohlebusch, E. (2005). Chaining algorithms for multiple genome comparison. J. Discrete Algorithms *3*, 321–341.

Almodaresi, F., Pandey, P., and Patro, R. (2017). Rainbowfish: a succinct colored de bruijn graph representation. In 17th International Workshop on Algorithms in Bioinformatics (WABI 2017), Schwartz., Russell., Reinert., and Knut., eds. (Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik), pp. 18:1–18:15.

Almodaresi, F., Sarkar, H., Srivastava, A., and Patro, R. (2018). A space and time-efficient index for the compacted colored de bruijn graph. Bioinformatics *34*, i169–i177.

Armstrong, J., Fiddes, I.T., Diekhans, M., and Paten, B. (2019). Whole-genome alignment and comparative annotation. Annu. Rev. Anim. Biosci. *7*, 41–64.

Baier, U., Beller, T., and Ohlebusch, E. (2016). Graphical pan-genome analysis with compressed suffix trees and the burrows-wheeler transform. Bioinformatics *32*, 497–504.

Benson, D.A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Ostell, J., Pruitt, K.D., and Sayers, E.W. (2017). Genbank. Nucleic Acids Res. D41–D47.

Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de bruijn graphs. In International Workshop on Algorithms in Bioinformatics, Raphael., Ben., Tang., and Jijun., eds. (Springer), pp. 225–235.

Brudno, M., Do, C.B., Cooper, G.M., Kim, M.F., Davydov, E., Green, E.D., Sidow, A., and Batzoglou, S.; NISC Comparative Sequencing Program (2003). Lagan and multi-lagan: efficient tools for large-scale multiple alignment of genomic DNA. Genome Res. *13*, 721–731.

Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de bruijn graphs from sequencing data quickly and in low memory. Bioinformatics *32*, i201–i208.

Darling, A.C., Mau, B., Blattner, F.R., and Perna, N.T. (2004). Mauve: multiple alignment of conserved genomic sequence with rearrangements. Genome Res. *14*, 1394–1403.

Darling, A.E., Mau, B., and Perna, N.T. (2010). progressivemauve: multiple genome alignment with gene gain, loss and rearrangement. PLoS One *5*, e11147.

Dewey, C.N. (2007). Aligning multiple whole genomes with mercator and mavid. In Comparative Genomics (Springer), pp. 221–235.

Dewey, C.N., and Pachter, L. (2006). Evolution at the nucleotide level: the problem of multiple whole-genome alignment. Hum. Mol. Genet. *15* (suppl_1), R51–R56.

Earl, D., Nguyen, N., Hickey, G., Harris, R.S., Fitzgerald, S., Beal, K., Seledtsov, I., Molodtsov, V., Raney, B.J., Clawson, H., et al. (2014). Alignathon: a competitive assessment of whole-genome alignment methods. Genome Res. *24*, 2077–2089.

Grabherr, M.G., Russell, P., Meyer, M., Mauceli, E., Alföldi, J., Di Palma, F., and Lindblad-Toh, K. (2010). Genome-wide synteny through highly sensitive sequence alignment: Satsuma. Bioinformatics *26*, 1145–1151.

Jain, C., Koren, S., Dilthey, A., Phillippy, A.M., and Aluru, S. (2018). A fast adaptive algorithm for computing whole-genome homology maps. Bioinformatics *34*, i748–i756.

Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. Bioinformatics *34*, 3094–3100.

Lilue, J., Doran, A.G., Fiddes, I.T., Abrudan, M., Armstrong, J., Bennett, R., Chow, W., Collins, J., Czechanski, A., Danecek, P., et al. (2018). Multiple laboratory mouse reference genomes define strain specific haplotypes and novel functional loci. bioRxiv. https://doi.org/10.1101/235838.

Luo, H., Arndt, W., Zhang, Y., Shi, G., Alekseyev, M.A., Tang, J., Hughes, A.L., and Friedman, R. (2012). Phylogenetic analysis of genome rearrangements among five mammalian orders. Mol. Phylogenet. Evol. *65*, 871–882.

Marcus, S., Lee, H., and Schatz, M.C. (2014). Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics *30*, 3476–3483.

Minkin, I., and Medvedev, P. (2019). Scalable multiple whole-genome alignment and locally collinear block construction with sibeliaz. BioRxiv. https://doi.org/10.1101/548123.

Minkin, I., Patel, A., Kolmogorov, M., Vyahhi, N., and Pham, S. (2013). Sibelia: A Scalable and Comprehensive Synteny Block Generation Tool for Closely Related Microbial Genomes (Springer Berlin Heidelberg), pp. 215–229.

Minkin, I., Pham, S., and Medvedev, P. (2017). Twopaco: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. Bioinformatics *33*, 4024–4032.

Muggli, M.D., Bowe, A., Noyes, N.R., Morley, P.S., Belk, K.E., Raymond, R., Gagie, T., Puglisi, S.J., and Boucher, C. (2017). Succinct colored de bruijn graphs. Bioinformatics *33*, 3181–3187.

Myers, G., and Miller, W. (1995). Chaining Multiple-Alignment Fragments in Sub-quadratic Time. SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms (Society for Industrial and Applied Mathematics), pp. 38–47.

Ohlebusch, E., and Abouelhoda, M.I. (2006). Chaining algorithms and applications in comparative genomics. Handbook of Computational Molecular Biology, 15–1-12–26.

Paten, B., Herrero, J., Beal, K., Fitzgerald, S., and Birney, E. (2008). Enredo and pecan: genome-wide mammalian consistency-based multiple alignment with paralogs. Genome Res. *18*, 1814–1828.

Pevzner, P., and Tesler, G. (2003). Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution. Proc. Natl. Acad. Sci. U S A *100*, 7672–7677.

Pham, S., and Pevzner, P. (2010). Drimm-synteny: decomposing genomes into evolutionary conserved segments. Bioinformatics *26*, 2509–2516.

Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., and Yorke, J.A. (2004). Reducing storage requirements for biological sequence comparison. Bioinformatics *20*, 3363–3369.

Srivastava, A., Malik, L., Zakeri, M., Sarkar, H., Soneson, C., Love, M.I., Kingsford, C., and Patro, R. (2019). Alignment and mapping methodology influence transcript abundance estimation. BioRxiv. https://doi.org/10.1101/657874v2.

Vollger, M.R., Dishuck, P.C., Sorensen, M., Welch, A.E., Dang, V., Dougherty, M.L., Graves-Lindsay, T.A., Wilson, R.K., Chaisson, M.J., and Eichler, E.E. (2019). Long-read sequence and assembly of segmental duplications. Nat. Methods *16*, 88.

iScience, Volume *23*

Supplemental Information

Scalable Pairwise Whole-Genome Homology Mapping of Long Genomes with BubbZ

Ilia Minkin and Paul Medvedev

Scalable pairwise whole-genome homology mapping of long genomes with BubbZ
Supplemental Information

Ilia Minkin*†[1] and Paul Medvedev‡[1, 2, 3]

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, 16802, USA
[2]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, PA, 16802, USA
[3]Center for Computational Biology and Bioinformatics, The Pennsylvania State University, University Park, PA, 16802, USA

1 Transparent Methods

1.1 Preliminaries

Strings and de Bruijn graphs

Let s be a string, indexed starting from 1. By s_i we denote the k-mer starting at position i of s. Given s and a positive integer k, we define a multigraph $G(s,k)$ as the *de Bruijn graph* of s. The vertex set consists of all substrings of s of length k, called k-mers. For each $(k+1)$-mer substring x in s, we add a directed edge from u to v, where u is the prefix of x of length k and v the suffix of x of length k. Each occurrence of a $(k+1)$-mer yields a unique multiedge, and every multiedge corresponds to a unique location in s. See Figure S1a for an example. Note that unlike some other definitions of a de Bruijn graph, a $(k+1)$-mer that occurs multiple times in s will have multiple corresponding edges. The de Bruijn graph for a set of sequences S is $G(S,k) = \bigcup_{s\in S} G(s,k)$. That is, the vertex set of $G(S,k)$ is the union of all the vertex sets (where vertices with the same label are considered identical) and the multiedge set of $G(S,k)$ is the union of all the edge sets (but where each multiedge is preserved, even if it shares a label with another multiedge).

The set of a multiedges in a graph G is denoted by $E(G)$. We write (u,v) to denote a multiedge from vertex u to v. A *walk* w is a sequence of multiedges $((v_1,v_2),(v_2,v_3),\ldots,(v_{|w|},v_{|w|+1}))$ where each multiedge (v_i,v_{i+1}) belongs to $E(G)$. The length of the walk w, denoted by $|w|$, is the number of multiedges it contains. A walk is *genomic* if it was generated by a substring in the input, that is, if the multiedges correspond to consecutive $(k+1)$-mers in the input.

*ivminkin@gmail.com
†To whom correspondence should be addressed.
‡pashadag@cse.psu.edu

Chains

Consider two chromosome sequences s and t and their de Bruijn graph $G = G(\{s\} \cup \{t\}, k)$. There are several ways to mathematically define a homologous pair of substrings from s and t. The definition that lends itself to de Bruijn graph-based algorithms is that of a *chain* (Zhang *et al.*, 1994; Myers, 1995). In our context, a chain is, informally, a sequence of common k-mers that forms a sub-sequence (i.e. substrings allowing gaps) in both strings interleaved by potential point mutations or indels of bounded length. Formally, a *chain* c of *weight* n is two non-decreasing sequences of indices $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$ such that $s_{i_x} = t_{j_x}$ and $i_x - i_{x-1} \leq b$ and $j_x - j_{x-1} \leq b$ and if $i_x = i_{x-1}$ then $j_x \neq j_{x-1}$, for all x. Each chain is associated with two genomic walks in G; specifically, the genomic walk corresponding to the substring of s starting from position i_1 and ending in position i_n, and, similarly, the genomic walk corresponding to the substring of t from position t_1 to t_n. See Figure S1a for an example of a chain.

Let $c = ((i_1, \ldots, i_n), (j_1, \ldots, j_n))$ and $c' = ((i'_1, \ldots, i'_m), (j'_1, \ldots, j'_m))$ be two chains. The *concatenation* of c and c' is the pair of sequences

$$c \cdot c' = ((i_1, \ldots, i_n, i'_1, \ldots i'_m), (j_1, \ldots, j_n, j'_1, \ldots, j'_m))$$

Note that $c \cdot c'$ is a chain iff $i'_1 \geq i_n$, $j'_1 \geq j_n$ and $i'_1 - i_n, j'_1 - j_n \leq b$ and either $i'_1 \neq i_n$ or $j'_1 \neq j_n$. In practice, we will be interested in the concatenation operation only if the result is a chain. We say that a chain c is *right-maximal* if there is no other chain c' such that $c \cdot c'$ is a chain, *left-maximal* if there is no other chain c' such that $c' \cdot c$ is a chain, and *maximal* if it is both left- and right-maximal.

1.2 Problem formulation and recurrence solution

To formulate the pairwise whole-genome homology mapping problem, let us take as input two chromosome sequences s and t, and a positive integer parameter b. As we discussed, we define a pairwise homology as a chain. One could then formulate the problem as that of outputting all maximal chains. However, such an output would contain a lot of redundancy, because two chains can span similar regions in s and t but contain different shared k-mers. To remove some of the redundancy, and with an eye towards an efficient algorithm, we use the notion of *(i,j)-maximum* chains. A chain is *(i,j)-maximum* if it ends in positions i and j in s and t, respectively, and has the highest weight among all such chains. Our problem formulation is then:

Problem Definition. *Given two sequences s and t and a positive integer b, output, for every $1 \leq i \leq |s|$ and $1 \leq j \leq |t|$, a maximal (i,j)-maximum chain, if it exists.*

This problem formulation lends itself naturally to dynamic programming, because (i,j)-maximum chains have an optimal substructure property. Formally,

Property 1. *Let c be an (i,j)-maximum chain of length greater than one. Let us decompose it as $c = d \cdot ((i'), (j')) \cdot ((i), (j))$, where d may be empty. Let c' be any (i', j')-maximum chain. Then $c' \cdot ((i), (j))$ is an (i,j)-maximum chain.*

Proof. Let w be the weight of $d \cdot ((i'), (j'))$. The weight of c is $w + 1$. Since $d \cdot ((i'), (j'))$ ends in $((i'), (j'))$, any $((i'), (j'))$-maximum chain must have weight at least w. Hence the weight of c' is at least w, and the weight of $c' \cdot ((i), (j))$ is at least $w + 1$. Since this is the weight of c and c was (i,j)-maximum, $c' \cdot ((i), (j))$ is (i,j)-maximum as well. □

To determine the i' and j' of this Property, we will define the *predecessor* function. Consider a pair of positions i and j such that $s_i = t_j$. We define its possible *left-extensions* as the set of pairs

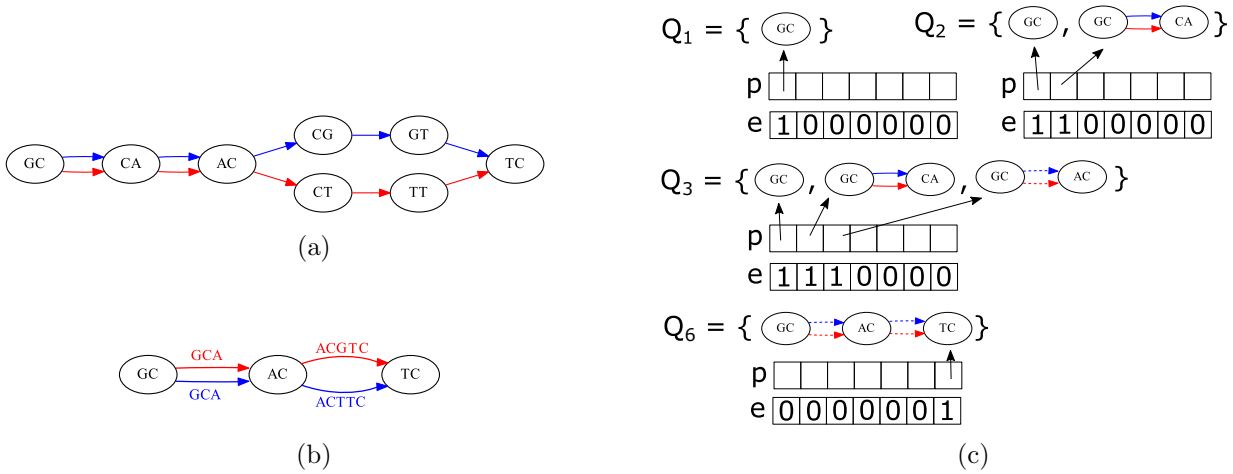


Figure S1: (a) De Bruijn graph built from strings $s =$"GCACGTC" and $t =$"GCACTTC", with $k = 2$. The two strings are reflected by the blue and red walks, respectively. The whole graph is chain $((1, 2, 3, 6), (1, 2, 3, 6))$. (b) The compacted version of the graph from panel (a); substrings generating the corresponding edges are shown adjacent to them. Note that the pair of edges between vertices "GC" and "AC" correspond to a case of parallel edges generated by identical substrings, while the edges between "AC" and "TC" form a bubble caused by a point mutation. (c) State of the list Q as well as vectors e and p, after considering each position i of the string s at which the algorithm adds a chain. The pointer in $p[j]$ leads to an element $C(i, j)$ of Q; $e[j] = 1$ if $p[j]$ is not a null pointer; $e[j] = 0$ otherwise. Q_1 shows the contents of Q after processing vertex "GC" (there is only one chain consisting of the initial k-mer); Q_2 contains the extended chain; and Q_6 has the whole graph minus the first chain that was removed due to being too far from the current position. Related to Table 1.

(i', j') such that $((i'), (j')) \cdot ((i), (j))$ is a valid chain. We define the *predecessor function* $\pi(i, j)$ to be the left-extension (i', j') such that the concatenation of an $((i'), (j'))$-maximum chain with $((i), (j))$ results in the chain of the highest weight. Formally,

$$\pi(i, j) = \arg\max\{\text{weight of } (i', j')\text{-maximum chain} \mid (i', j') \text{ is a left-extension of } (i, j)\}$$

Ties are broken by choosing the chain with a smaller value of j', and then with a smaller i' if the tie still exists. If there are no left extensions, we set $\pi(i, j) = \emptyset$. The predecessor function gives rises to our dynamic programming matrix C, where each entry $C(i, j)$ stores an (i, j)-maximum chain, as follows:

$$C(i,j) := \begin{cases} \emptyset & \text{if } s_i \neq t_j, \\ ((i),(j)) & \text{else if } \pi(i,j) = \emptyset, \\ C(\pi(i,j)) \cdot ((i),(j)) & \text{else} \end{cases} \tag{1}$$

The predecessor function can be computed by checking the value of C for all possible left-extensions. A solution to our problem is then to compute C and output every $C(i, j)$ that is also maximal.

Observe that an (i, j)-maximum chain is by definition left-maximal, so it suffices to check if $C(i, j)$ is right-maximal. This can be done easily, as follows. Observe that $C(i, j)$ is not right-maximal if and only if there are some offsets $1 \leq \alpha \leq b$ and $1 \leq \beta \leq b$ such that $s_{i+\alpha} = t_{j+\beta}$. In practice, b is quite small, when appropriate data structures are maintained (details omitted), we can check if a chain is right-maximal quickly. In what follow, we will therefore focus on just computing C.

1.3 High-level algorithm

Equation (1) immediately lends itself to a naive dynamic programming algorithm that uses a table where each cell corresponds to a row i and a column j. Such an algorithm can compute all $C(i, j)$ but will use $\Omega(|s||t|)$ memory, which is prohibitive. Instead, we present an algorithm that exploits the sparseness and structure of C as well as the fact that the maximum gap is limited by parameter b.

Let $C(i', j') = C(\pi(i, j))$ denote the predecessor chain of $C(i, j)$. First, we observe that if we compute the values of $C(i, j)$ in increasing order of i, we are guaranteed that the predecessor chain has already been computed, i.e. $i' < i$. Second, by definition of a valid chain, the predecessor chain must lie within the b previous columns, i.e. $i' \geq i - b$. Hence, it is not necessary to retain the whole table in memory, but rather, just the previous b columns. Third, the matrix is mostly sparse, since it only contains values when $s_i = t_j$. Therefore, storing it as a matrix is impractical. Instead, we will store the elements of the previous b columns in a queue Q that supports the lookup operation, $\mathit{Lookup}(Q, (i, j)) = C(i, j)$, if $C(i, j)$ is in Q. We will describe the implementation of the lookup function in Section 1.4.

The pseudocode of our method is in Algorithm 1. The outer for loop iterates over all the values of i. The inner for loop iterates over all values of j where $C(i, j) \neq \emptyset$. Lines 4 through 8 implement the logic of Equation (1). When column i is finished, lines 10 through 14 update Q by removing all chains from the now outdated column $i - b$ and, for those that are right-maximal, outputting them. Figure S1c shows an example of the contents of Q after several iterations.

Let us use $C(i, *)$ as shorthand for $C(i, j)$ for all j. The correctness of the algorithm follows from the previous discussion and the following theorem. For clarity, the pseudocode and the theorem omit some corner cases (e.g. when $i' = i$ or when we hit the end of the strings).

Algorithm 1 *Find-chains*

Input: strings s and t, graph $G(\{s\} \cup \{t\}, k)$, integers b and m
Output: the set of all chains in C that are right-maximal.

```
1: Q ← an empty doubly-linked list                          ▷ The set of current chains C(i, j)
2: for i ← 1 to |s| do
3:     for all j such that t_j = s_i do                     ▷ Consider all position of k-mer s_i in t
4:         if π(i, j) ≠ ∅ then
5:             r ← Lookup(Q, π(i, j))                       ▷ Equation (1)
6:             PushBack(Q, r · ((i), (j)))
7:         else
8:             PushBack(Q, ((i), (j)))
9:     let c ← Front(Q) and denote the end of c as (i′, j′)
10:    while i′ < i − b do                                  ▷ Cleaning-up and outputting Q
11:        if c is right-maximal then
12:            output c
13:        PopFront(Q)
14:        let c ← Front(Q) and denote the end of c as (i′, j′)
```

Theorem 1 (Correctness of Algorithm 1)**.** *At the end of the i-th iteration,*

1. *Q is the set of $C(i', *)$ for all $i - b \leq i' \leq i$, in front-to-back order of non-decreasing i'.*
2. *The algorithm's output has been the chains $C(i', *)$ which are right-maximal and for which $i' < i - b$.*

Proof. For the base case ($i = 0$), the statement holds since Q is empty. For the general case, the induction hypothesis tells us that at the start of the i-th iteration, Q contains $C(i', *)$ for all $i - b - 1 \leq i' \leq i - 1$, in order. To show (1), we will show that during the iteration, $C(i - b - 1, *)$ are popped from the front and $C(i, *)$ are pushed to the back. $C(i - b - 1, *)$ are popped from the front of Q during the while loop, using the fact that Q is in order. $C(i, *)$ are computed in the inner while loop using the logic of Equation (1), so all we need to show is that if $\pi(i, j) \neq \emptyset$, then $C(\pi(i, j)) \in Q$. Let $(i', j') = \pi(i, j)$. Because the gap between indices in a chain cannot exceed b, we have $i' \geq i - b$. By part (1) of the induction hypothesis, $C(i', *)$ is in Q, and hence $C(i, j)$ is pushed to the back of Q during the inner for loop.

Next we show (2). By induction, before the i-th iteration the output was $C(i', *)$ for all $i' < i - b - 1$. We need to then show that during the i-th iteration, the output is $C(i - b - 1, *)$. By part (1) of the induction hypothesis, the front of Q contains $C(i - b - 1, *)$. These elements will be popped during the while loop and output if they are right-maximal. □

1.4 Important details

There are additional aspects that the pseudocode does not address. We described the algorithm considering only the single strand of DNA. To handle both strands, we run a slightly modified version of our algorithm on the graph $G_{\text{comp}}(s, k) = G(s, k) \cup G(\bar{s}, k)$, where $\bar{s}$ is reverse complement of s (Minkin *et al.*, 2017). We also preprocess the graph by removing all k-mers occurring more than a times, where a is a parameter. High-frequency k-mers can clog up our data structures and slow down the algorithm. We allow the user to set a, thereby controlling the trade-off between speed and potential decrease in accuracy. Finally, to save space, we do not store the actual chains

in Q, but only their starting and ending coordinates, since this is what the final mapping will return anyway.

To support the computation of π and the lookup operation for Q (in Line 5), we need a specialized index. To quickly iterate over all left extensions of $((i),(j))$ we keep a bit vector e such that $e[j'] = 1$ if and only if Q contains a chain $C(i', j')$, for some i'. We also keep a vector p where $p[j']$ contains a pointer to an element $C(i', j')$ if one exists. If there are several such elements, we choose the one with the biggest i'.

Using a special machine instruction returning the number of trailing 0-bits, we can find all j' in the range of $j - b \leq j' \leq j$ such that $e[j'] = 1$ using using $\max(m, b/64)$ operations, where m is the number of ones in the range. In the GCC compiler the instruction is designated as __builtin_ctzll. Once we identify such values of j', we the use pointers in $p[j']$ to access the actual chains and select the one that yields the best predecessor for $C(i, j)$.[1]. Due to the nature of the de Bruijn graph, we expect the vector e to be sparse which results in efficient lookups. Figure S1c contains an example of state of vectors e and p during several iterations of running Algorithm 1

1.5 Adaptation of the algorithms to the compacted graph

For simplicity of exposition, we described our algorithm in terms of the regular de Bruijn graph. Our implementation, however, operates on the compacted de Bruijn graph which we build using TwoPaCo (Minkin *et al.*, 2017). The vertex set of the compacted graph is a subset of vertices of the regular graph, called *junctions*. Intuitively, a vertex is a junction if it is either a branching vertex or is the start or end of an input string (for an exact definition, please see Minkin *et al.* (2017)). The reason we can consider only junctions is that one can show that there is a one-to-one correspondence between the maximal chains in the ordinary graph and the ones in the compacted one. Particularly, any maximal chain starts and ends with a junction. Since the number of junctions is usually much smaller than the total number of k-mers, using only junctions greatly speeds up the algorithm and saves space, while not affecting the output of the algorithm.

A pair of vertices can be connected in the compacted graph by a pair of edge-disjoint genomic walks in two ways. These walks are either a pair of parallel edges representing a stretch of identical k-mers, or two walks forming a so called "bubble" which correspond to a sequence of point mutation or a short indels. Figure S1b shows an example of the compacted graph containing a pair of parallel edges and a bubble. To adapt our algorithm to the compacted graph, we modify the definition of chain such that it now consists of junction k-mers that are connected either by a pair of parallel edges or a bubble of size at most b.

Formally, two pairs of indices (i_1, j_1) and (i_2, j_2) are compatible if $s_{i_1} = t_{j_1}, s_{i_2} = t_{j_2}$ and either or both of the following holds: (1) $i_2 - i_1 \leq b$ and $j_2 - j_1 \leq b$; (2) $i_2 - i_1 = j_2 - j_1$ and $s_{i+p} = t_{j+p}$ for $i_1 \leq p \leq i_2$. The first condition models a bubble of size at most b, while the second one represents a stretch of identical k-mers corresponding to a pair of parallel edges in the compacted graph. Note that we have to handle the case of parallel edges separately because they might correspond to more than b k-mers and we should allow to "skip" over such pair of edges regardless of its length. A chain then is a pair of non-decreasing sequences of indices $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$ such that $s_{i_x} = t_{j_x}$, each s_{i_x} (t_{j_x}) is a junction, (i_x, j_x) and (i_{x+1}, j_{x+1}) are compatible for $1 \leq x < n$ and if $i_x = i_{x-1}$ then $j_x \neq j_{x-1}$, for all x.

We adjust the code of as follows. In the loop in Lines 2 to 14 of Algorithm 1 we iterate over junctions of $\ell(s)$ instead of ordinary k-mers. We also modify our lookup procedure to take the new

[1]Our actual implementation does not store the vector p explicitly; instead we use a mapping from the k-mer set to the data structure Q.

definition of chain into account, as well the clean-up procedure in Lines 10 to 14. Particularly we handle two separate cases for extension of a chain: by a pair of parallel edges, or by a "bubble."

1.6 Computational complexity

Here we will analyze the complexity of Algorithm 1. Let the maximum degree of a vertex considered by our algorithm be a. This could just be the maximum degree in the graph or it could be the parameter a set by the user. The number of iterations of the inner loop in Lines 4 to 8 is bounded by $a|s|$. Computing π and doing the lookup operation in Line 5 takes $O(b)$ operations in the worst case, as described in Section 1.4. The processing of Q in Lines 10 to 14 takes a total of $O(a|s|)$ time over the course of the algorithm, since this is the number of elements pushed into Q. As the result, the total time complexity is $O(ab|s|)$. The space complexity is dominated by the data structures to store the mappings for the shared k-mers. The amount of memory is strongly dependent on the structure of the input, and we therefore did not perform a worst case analysis.

1.7 Modes of operation

Algorithm 1 can also be used to find chains within a single genome, corresponding to duplications. To do this, the user should give as input a pair of identical sequences. To handle this case, we modify our algorithm to forbid chains that overlap with themselves. To implement this, we perform additional checks before concatenating chains in Line 5 (details omitted).

Algorithm 1 can also be used to compute an all-against-all mapping for a set of chromosomes $S = \{s_1, \ldots, s_{|S|}\}$. Rather than performing $\Theta(|S|^2)$ runs of the algorithm, we can modify the algorithm to run only $\Theta(|S|)$ times, at a potential cost of more memory, as follows. We first compute the de Bruijn graph from all of S, i.e. $G(S, k)$. Then we run Algorithm 1 $|S|$ times; in the i^{th} run, s_i plays the role of s and the chromosomes $\{s_i, \ldots, s_{|S|}\}$ play the role of t. In our pseudocode, t is a single string; but, we can easily modify it to allow t to be a set of strings by considering positions in all sequences of $\{s_i, \ldots, s_{|S|}\}$ in Line 2. It may also be that the underlying graph $G(S, k)$ could have vertices from some chromosome s_j that is not part of the comparison (i.e. $j < i$); however, since the algorithm only looks at k-mers that appear in s or t, those extra k-mers would not effect the execution of the algorithm. This approach to all-against-all mapping will give the same results as the naive $O(|S|^2)$ runs approach. However, it does have an associated memory cost, since we must maintain in memory a de Bruijn graph of $|S|$ sequences, rather than just the graph of 2 sequences. This strategy also lends itself to parallelization, by executing these $|S|$ runs in parallel using multithreading. Finally, note that the same strategy applies to all-against-all mapping of multiple multi-chromosomal genomes, since our algorithm does not distinguish between chromosomes on the same vs. different genomes.

References

Minkin, I., Pham, S., and Medvedev, P. (2017). Twopaco: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, **33**(24), 4024–4032.

Myers, G. (1995). Chaining multiple-alignment fragments in sub-quadratic time.

Zhang, Z., Raghavachari, B., Hardison, R. C., and Miller, W. (1994). Chaining multiple-alignment blocks. *Journal of Computational Biology*, **1**(3), 217–226.