Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: 2 April 2015
Applications Note

Genome analysis

Learned Data Models over the de Bruijn Graph for Alignment

Andrew Mao [1,*], Sean Lin [1]

[1] Department of Computer Science, University of Maryland, College Park, 20742, United States

* To whom correspondence should be addressed.

Abstract

Motivation: Fast and low memory sequence alignment is a fundamental problem in high throughput genomics. Learned index models such as Sapling are a recent approach that accelerate queries on the suffix array, a full text index. Approaches that index the compacted de Bruijn Graph (cdBG) have also been successful in reducing index size and query time. We explore the performance impact of combining the two approaches, and implement a learned data model for the suffix array on the cdBG.
Results: We find that our model has faster build times, query times, and less memory than Sapling once the genome size exceeds a critical threshold. The performance gap increases as the reference size increases, due to the cdBG's ability to better compress the reference.
Availability: The code for our project is available at https://github.com/shuai626/dbgrepling.
Contact: amao1@terpmail.umd.edu

1 Introduction

Fast and low memory sequence alignment is a fundamental problem in high throughput genomics. The time for performing high-throughput DNA sequencing and population-level genomics depends heavily on searching for short sequences in a database of reference sequences. A typical component of a solution involves building an index of the database to accelerate the search for exact substrings. There are two major lines of work on making efficient indices for exact substring match: full-text indices, such as the suffix array (Manber and Myers, 1993) and FM-index (Ferragina and Manzini, 2000); and hash tables for k-mers (Almodaresi *et al.*, 2018; Pibiri, 2022).

Another way to categorize approaches is by the underlying model of the reference genome. One set of approaches uses the compacted de Bruijn graph (cdBG) as the underlying data structure (Chikhi *et al.*, 2014; Almodaresi *et al.*, 2018; Pibiri, 2022), which is attractive as it naturally collapses repetitive sequences of the reference, which can speed up queries.

Recent work uses "learned index structures" (Kraska *et al.*, 2018), which model patterns in the data to accelerate queries on data structures such as the suffix array (Kirsche *et al.*, 2021) and the FM-index (Ho *et al.*, 2021).

One question is whether learned indices may benefit from running on the cdBG, instead of the full reference genome. We propose *dbGrepling*, a learned data model for the suffix array on the cdBG. dbGrepling is based on Sapling (Kirsche *et al.*, 2021), a learned model on the suffix array on the full text. From our experiments, we find that for small k and large N, dbGrepling is both faster than Sapling and uses less space. We attribute the performance improvement to the cdBG's ability to compress the reference, and the performance gap increases as the compression gap increases. Our contributions are as follows:

1. We propose dbGrepling, a learned data model for the suffix array on the cdBG. dbGrepling is based on Sapling, but adds an additional step to find the stitched unitig that contains the match.
2. We benchmark dbGrepling, Sapling, binary search, and SSHash, measuring query time and index space. We find that for small k and large N, dbGrepling is both faster than Sapling and uses less space.
3. We analyze the cdBG size versus the reference size for a number of real and synthetic genomes, to understand when a model on the cdBG may be advantageous.

2 Background

Sequence alignment is the task of aligning sequencing reads to a reference genome or collection of genomes. A common method for finding exact and approximate alignments is the "seed-and-extend" algorithm, which relies on being able to quickly search for exact matches of short seed sequences in the reference (exact substring search problem). This is done by indexing the reference genome to support fast exact substring search. Many data structures can be used as the basis for the index, such as suffix arrays (Manber and Myers, 1993), suffix trees (Weiner, 1973), hash tables (Karp and Rabin, 1987) and FM-indexes (Ferragina and Manzini, 2000).

The suffix array is a key data structure for seed-and-extend algorithms used by Star (Dobin et al., 2013), MUMMER4 (Marçais et al., 2018) and others. The suffix array consists of the lexicographically ordered list of suffixes present in a string, and a modified binary search can be used to quickly locate exact matches (Manber and Myers, 1993).

Learned index structures (Kraska *et al.*, 2018) are a technique for accelerating queries on a variety of data structures by using patterns in the underlying dataset.

Sapling (Kirsche *et al.*, 2021) is an algorithm for sequence alignment, which uses a learned data model to augment the suffix array and enable faster queries. The authors investigate different types of data models such as neural networks, but find that a simple piecewise linear model provides the best balance of speed and memory.

LISA (Ho *et al.*, 2021) is also a learned index structure for alignment, which is built on the FM-index. While LISA outperforms Sapling, we chose to work with Sapling due to it being easier to set up and having better documentation.

Our model, dbgrepling, combines Sapling and indices on the de Bruijn graph. The de Bruijn graph is an attractive data structure to index, due to its natural ability to collapse highly-repetitive sequence regions. Existing structures for representing the de Bruijn graph typically fall into two categories; those that are hashing-based and provide very fast access to the underlying k-mer information (Pufferfish, SSHash), and those that build a full text index on top, which is space-frugal but provides slower search (dBG-FM). We go the direction of building a full text index on top of the stitched unitigs on the cdBG, but we use a technique from Pufferfish, one of the hashing approaches.

Pufferfish (Almodaresi *et al.*, 2018) uses a minimum perfect hash function (MPHF) to map each k-mer to its location in the unitigs, and a vector of unitig positions endowed with rank operations to locate the unitig. The authors also proposed a sparse version of the index where the vector of positions is sampled to improve space usage at the expense of query time. The unitig search step of dbGrepling is the same as the step in Pufferfish that searches in the useq array.

SSHash (Pibiri, 2022) is an exact, associative, and compressed dictionary data structure for k-mers, based on MPH and compact encodings, that exploits the sparse and skewly distributed properties of minimizers. SSHash improves on Pufferfish, and we ran some preliminary experiments comparing dBGrepling to SSHash.

3 Approach

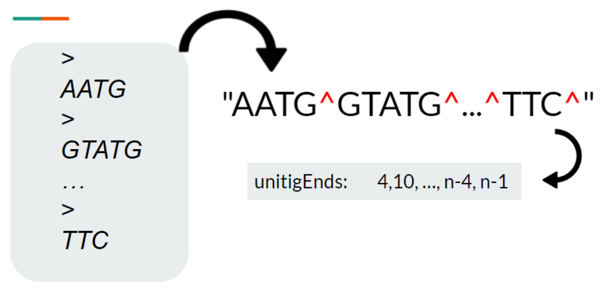

Fig. 1. Diagram of the stitched unitig representation of the dBG that dbGrepling operates on.

In this work, we modify the learned model of Sapling to work on the compacted dBG, represented as a set of concatenated maximal simple paths, or stitched unitigs.

Our hypothesis is that a learned model may speed up search over the unitig sequence versus the full reference, when the reference contains many repetitive sequences.

At construction, we modify Sapling to work on stitched unitigs. As we read in each unitig, we concatenate them together using an 'ˆ' as the separator character. These separator characters ensure that query strings do not match with substrings that span across several unitigs.

The positions of each separator characters are recorded in a list. The resulting in-memory string is treated as if it is the reference text, and passed to underlying Sapling logic to create a suffix array and learned index.

When querying, dbGrepling first uses Sapling to return the location in the concatenated unitig string where it found a match. We apply a learned model to return this global index.

Then, we seek to locate the id of the unitig that contains this index. We implement two solutions for this. First, we implement a simple binary search over the positions of the separator characters. Second, we convert the concatenated unitig string into a bit-victor where positions containing a separator characeter are marked 1 and 0 otherwise. We use the rank algorithm for a bit-vector to return the unitig associated at any index. The rank query has a constant time complexity, and should be faster than binary search.

3.1 Optimization

With the current setup, dbGrepling is slower than Sapling on the full reference and binary search. We identify the slowdown to be caused by the separator characters in the stitched unitigs. Sapling treats all unknown characters as "A" when creating k-mer codes; we hypothesize that this mismatch results in large errors when building the model which results in wide guesses, causing the slowdown (performance similar to binary search on the whole interval).

To address this, we remove the separator characters from the concatenated string and add a final step in the query to filter out false positives: if the suffix array query returns a match, we check if the match spans over a unitig boundary.

If it does, then the match is a false positive. We then perform a linear scan in both directions on the suffix array starting at the initial matched index to check if there are any additional matches. All additional matches are examined to see if they span over a unitig boundary.

If all additional matches are false positives, then the query string does not exist in the cdBG.

4 Methods

We evaluate dbGrepling against two other models for indexing. We perform 1 million queries of 11-mers across varying substrings lengths of C. elegans: from 1,000 to 100,000,000. We then evaluate each method's performance over real full-text references: E.coli, GRCh38, C.elegans, S.scrofa, and G.morhau. We guarantee all queries exist in the reference and record the overall query time and memory usage of each index.

To obtain the cdBG of a full-text reference, we preprocess the reference into stitched unitigs using BCALM2 and UST. BCALM2 constructs a compacted deBruijn Graph. UST then further compacts the representation using a spectrum-preserving string set.

We use the following models:

- dbGrepling on the stitched unitigs. We set the overhead of 10%, which is the fraction of the memory of the reference.

Genome	raw size (bytes)	stitched unitigs size, k=11 (bytes)	stitched unitigs size, k=31 (bytes)
E. Coli	4,719,087	6,758,070	4,579,745
GRCh38: Chromosome 13	116,270,459	9,270,693	99,709,776
C. elegans	101,958,257	9,740,837	98,641,615
GRCh38: Chromosome 6	173,652,802	9,526,653	174,610,067
S. scrofa: Chromosome 1	278,902,769	9,657,167	284,676,385
Gadus morhua	681,137,419	9,743,856	–
GRCh38	3,144,230,986	9,755,907	–

Table 1. cdBG (stitched unitigs) sizes for a few different genomes. We see that the cdBG becomes smaller than the reference for k=11 but not k=31

- Sapling on the full reference. We use the piece-wise linear model, and set the overhead of 10%, which is the fraction of the memory of the reference.
- The "simple accelerant" binary search for the suffix array, on the stitched unitigs.

For dbGrepling, we measure its performance on queries that are all positive hits (exist in the reference), random queries (may span unitig boundaries), and the lower bound (no checks on the validity of the query). For plotting, we take the average of 5 trials.

4.1 Technical Details

Experiments were performed on a machine using a 3.1 GHz Intel Core i7 Processor and 16 GB of RAM.

To obtain our stitched unitigs, we run a reference sequence through the tools BCALM2 and UST.

We modified the Sapling codebase, which was written in C++. Data analysis was done in Google Sheets and Python.

4.2 When is using the de Bruijn Graph advantageous?

To motivate the benefits of the cdBG, we measured the size of the stitched unitigs against the full-text reference size for a few genomes (Table 1). We find that the cdBG is an order of magnitude smaller than the reference when k=11 (k is the size of the k-mer used in creating nodes of the cdBG). This relationship disappears when k=31. Note that we were unable to build a stiched unitig file for k=31 for G. morhau and the full GRCh38 due to limitations of our machine in the given time frame.

We also plot the cdBG size for different size substrings of C. Elegans (Figure 2). We find that the cdBG size isn't smaller than the reference until its length N is greater than 10^7 and when k=11.

We conclude that whether a cdBG is smaller than its reference depends on both N and k. How to choose k is still unclear, and we leave this as future work.

4.3 Results

For the range of substrings of C. elegans and the parameters we tested, we found that dbGrepling built and ran faster than Sapling on the reference for N greater than 10^7 (Figure 3).

dbGrepling with positive queries is faster than with random queries, which is explained by the additional time to check for valid matches. dbGrepling with positive queries performs close to its optimum lower bound.

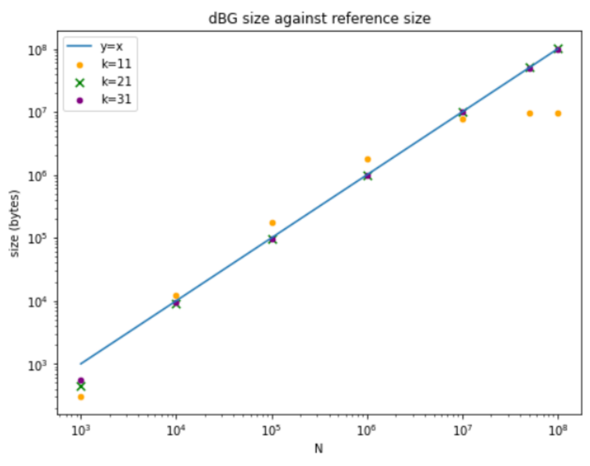

Fig. 2. Plot of the dBG size against the reference size, for different substrings of c. elegans. The dBG is smaller than the reference above N=10^7 and k=11.

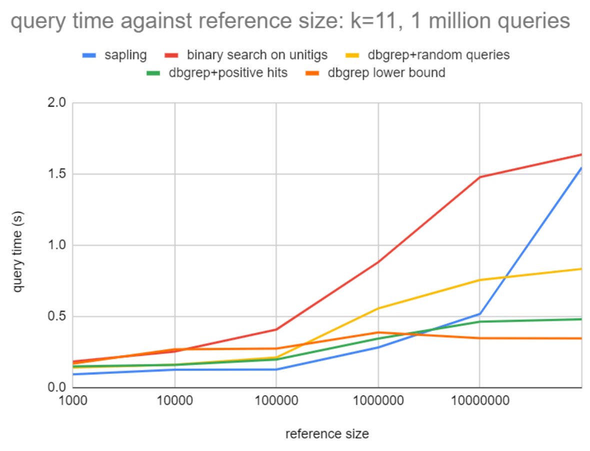

Fig. 3. Plot of the time to query 1 million 11-mers for different size references. dbGrepling outperforms Sapling for $N >= 10^7$. A "relaxed" version of dbGrepling is even faster, and provides an experimental lower bound.

Genome	Sapling: Query Time (seconds)	dbGrepling: Query Time (seconds)
E. Coli	0.370643	0.755654
GRCh38: Chromosome 13	1.44057	0.798625
C. elegans	1.94166	0.893499
GRCh38: Chromosome 6	1.78704	0.987004
S. scrofa: Chromosome 1	1.92539	0.752208
Gadus morhua	–	0.882214
GRCh38	–	0.934265

Table 2. Query times between Sapling v dbGrepling over genomes

For full-text references, dbGrepling outperformed Sapling in both build and query time whenever the cdBG representation was smaller than the original reference (Table 2). One exciting note is that over the two largest references: G.morhau and the full GRCh38 genome, dbGrepling built and queried the benchmark well before Sapling finished building the suffix array. In fact, Sapling was unable to build G.morhau nor the full GRCh38 genome in the allotted time frame.

Our preliminary results for measuring space is that the space usage of dbGrepling scales closely with the size of the cdBG, and the size of unitig boundaries is negligible. Hence on references that the cdBG compresses well, dbGrepling is capable of constructing the suffix array and underlying Sapling data structure over the cdBG faster than native Sapling over the full reference. We conclude that dbGrepling outperforms Sapling in space.

5 Discussion

The results from these experiments are promising steps towards a faster model for sequence alignment, however much work still needs to be done. dbGrepling is missing the step that matches the unitig and offset with matches in the reference text. dbGrepling doesn't support searching reverse complements or partial matches. We didn't do a careful analysis of the space/time tradeoff of dbGrepling, or its performance on larger genomes. We noticed that the data has non-negligible variance in repeated trials, and want to quantify this noise. Further work needs to be done to build a complete aligner around dbGrepling, and comparing it to existing models like LISA.

6 Conclusion

In this paper, we proposed dbGrepling, a learned data model for the suffix array on the stitched unitigs of the compacted de Bruijn graph. We found that for small k and large N, dbGrepling is both faster than Sapling and uses less space. We attribute the performance improvement to the dBG's ability to compress the reference, and the performance gap increases as the compression gap increases. We believe learned indices and indices on the dBG are a promising direction for accelerating alignment and reducing space.

7 Reflection

We're pretty satisfied with our project, and we think we took a principled approach. We chose a codebase that was decently documented, we focused on getting results quickly and tweaked our project goals to allow that, so we had plenty of time to iterate and improve on our initial results, and ended up getting interesting results! Our general advice to other groups would be to focus on getting results quickly, and make it easy to get them; for us this meant using existing code, writing bash scripts to run many experiments, and starting with small data. It provides stress relief, motivation, and the ability to iterate on the results. For our specific project, we pivoted to working with a learned index data structure on the suffix array (Sapling) instead of the FM index (LISA), as LISA was poorly documented and we couldn't run the code.

The rest of the project was pretty well scoped; it was easy to make the stitched unitigs, run an existing model on top, and a little harder working with sdsl for rank, which had some strange behavior. The hardest and longest part was improving the model performance, and identifying where the slowdown was happening. For future work, we think a logical direction would be to add or implement a reference lookup step at the end to map unitigs to their positions in the full-text reference (similar to Pufferfish). This would make dbGrepling a full aligner.

Acknowledgements

We want to thank Professor Patro for the idea, guidance, and many pieces of advice on this project.

References

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**(13), i169–i177.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., and Medvedev, P. (2014). On the representation of de bruijn graphs. In *International conference on Research in computational molecular biology*, pages 35–55. Springer.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE.

Ho, D., Kalikar, S., Misra, S., Ding, J., Md, V., Tatbul, N., Li, H., and Kraska, T. (2021). Lisa: Learned indexes for sequence analysis. *bioRxiv*, pages 2020–12.

Kirsche, M., Das, A., and Schatz, M. C. (2021). Sapling: accelerating suffix array queries with learned data models. *Bioinformatics*, **37**(6), 744–749.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. (2018). The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504.

Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, **22**(5), 935–948.

Pibiri, G. E. (2022). Sparse and skew hashing of k-mers. *bioRxiv*.