Building Compacted de Bruijn Graph from 100 Human Genomes with [Tool Name]

Ilia Minkin[1] and Paul Medvedev[1]

Department of Computer Science and Engineering, The Pennsylvania State University, USA

1 Introduction

A paragraph stub. Discuss the importance of de Bruijn graphs [1] in assembly [cite assembly applications] and comparative genomics [cite comparative genomics applications].

A paragraph stub. Tell about compressed graph and its advantages [cite paper where it first appeared]. State that it is desirable to avoid construction of an ordinary graph first.

A paragraph stub. Notice that all methods applicable to pan-genome are slow and/or require a lot of memory.

A paragraph stub. "We invented a new algorithm that is parallelizable and requires much smaller memory..." Say a few a words about the idea: based on Bloom Filters, constructs a partially compacted graph first, then filters out false positives.

2 The Basic Algorithm

A paragraph stub. Define ordinary de Bruijn graph [figure needed] for pan-genome. Define the compacted graph [figure needed]; define what a bifurcation is.

A paragraph stub. Say a few words high-level about Bloom filters: structure, supported operations, etc.

A paragraph stub. Basic observation #1: if a genomic substring S is flanked by a pair of bifurcations; S is an edge in the compacted graph. Note that it is true only for pan-genome case [figures needed].

A paragraph stub. Basic observation #2: suppose that we have a data structure that can list output/input edges of vertex. Given such a structure, it is easy to decide whether a vertex is a bifurcation.

A paragraph stub. If we use Bloom filter as such a structure, we can discover vertices of a partially compacted graph – candidates for true bifurcations[figure?]

A paragraph stub. We can quickly remove false bifurcations by explicitly exploring edges of candidate bifurcations [figure?].

A paragraph stub. Present a figure with the basic algorithm with three stages:

1. Filling the filter
2. Checking extensions, or partial compaction
3. Filtering out false positives, or full compaction

A paragraph stub. Discuss double-strandness: for each copy of a $k+1$-mer store its "canonical" version in a Bloom Filter.

3 Parallelization Scheme

A paragraph stub. An advantage of our Basic Algorithm is that it can be effectively parallelized

A paragraph stub. For the parallelization of first two stages, we use the following schema [figure needed]:

1. One reader thread that splits input into parts and sends them into worker queues
2. N worker threads that grab genome parts from queues and process them

In the first stage, worker threads work with shared Bloom filter – synchronization is done via atomic operations on it. In the second stage, workers have no shared data (except output file) – things are easy.

A paragraph stub. Third stage is simple: parallel sorting and exploration of $k+1$-mers [figure needed?]

4 Effects of Bloom Filter Size and Parameter Selection

A paragraph stub. Performance critically depends on the memory available. If Bloom Filter is too small – huge amount of false positives will degenerate the approach into mere sorting of $k+1$-mers. But we can fix this.

A paragraph stub. Observation: we can split all k-mers into a family of classes, and identify bifurcations within each class separately. Way to separate: take a range of hash function and split it into segments. Class = k-mers that hash into a single segment. Naive splitting can be uneven. How to split well?

A paragraph stub. Notice that FP-rate depends on the number of elements in the BF. So we are interested in classes that have number of edges as equal as possible. Algorithm [figure needed]:

1. Divide range of a HF into a number of buckets
2. For each bucket b estimate number of edges that have k-mers that hash into b. How?
3. Modify Bloom Filling procedure: before putting an edge into BF, check if it was seen before. If not, increment counter for the bucket corresponding to the value of hash function

   A paragraph stub. Present a figure with full algorithm: parameters split and run in parallel.

   A paragraph stub. Note that round-splitting leads to a perfectly scalable cluster implementation.

5 Analysis of The Algorithm

A paragraph stub. Explore efficiency of the round splitting: derive expected amount of false positives given the size of true bifurcation set.

A paragraph stub. Based on the analysis above, show the expected running time and memory usage by the algorithm.

6 Results

A paragraph stub. Overview the experiment design:

1. Comparison with other tools
2. Parallel scalability
3. Round-splitting efficiency: compare our procedure with naive splitting

A paragraph stub. Highlight the results of comparison with other tools Notice that Schatz's paper mentioned Sibelia in a totally, absolutely, completely, fully, entirely, perfectly, thoroughly incorrect way.

A paragraph stub. Discuss the results of scalability experiments.

A paragraph stub. Speculate about round-splitting results: the procedure works better than the naive approach.

7 Discussion

A paragraph stub. State that the algorithm works well and have the following advantages:

1. Faster than competitors
2. Smaller memory than competitors
3. Parallelization scalability on a SMP system
4. Smooth memory/time tradeoff
5. Parallelization scalability on a cluster
6. Simple: no need for a suffix array/tree

Note that experimental results directly support claims 1-4.

A paragraph stub. Discuss possible applicability of partially compacted graphs.

A paragraph stub. Show limitations & drawbacks:

1. Can't be applied to assembly setting
2. Bloom filters are cache inefficient
3. ?

A paragraph stub. Main take-home message: de Bruijn graph for pan-genome are easy to construct, and can form the backbone of sequence genome comparison: reference/variant representation, alignment and synteny blocks construction.

Acknowledgements. Say thanks to Daniel Lemire, author of [2] for his enormous support.

References

1. Bruijn, d.N.: A combinatorial problem. Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen. Series A 49(7), 758 (1946)
2. Lemire, D., Kaser, O.: Recursive n-gram hashing is pairwise independent, at best. Computer Speech & Language 24(4), 698–710 (2010)