CortexJDK: a Java API for accessing and navigating linked de Bruijn graph data structures

Kiran V Garimella[1,2*], Warren Kretzschmar[4*], Zamin Iqbal[1,3] and Gil McVean[1,2]

[1]Wellcome Trust Centre for Human Genetics, Oxford, OX3 7BN, UK
[2]Big Data Institute, Li Ka Shing Centre for Health Information and Discovery, Oxford, OX3 7LF, UK
[3]European Bioinformatics Institute (EMBL-EBI), Wellcome Genome Campus, Hinxton, CB10 1SD, UK
[4]School of Engineering Sciences in Chemistry, Biotechnology and Health, Department of Gene Technology, SciLifeLab, KTH Royal Institute of Technology, Stockholm, Sweden

Abstract

Motivation: The linked de Bruijn graph data structure restores long-range connectivity information lost during initial de Bruijn graph construction, enabling multi-sample reference-free comparison and improved discovery of large and complex variants.
Results: Here, we present CortexJDK, a Java API that provides low-memory random access the contents of the underlying graph and sophisticated link-informed traversals over the data. By providing the bioinformatics community with a library for accessing and navigating these files, we substantially lower the barrier to entry for developers wishing to create innovative applications using the linked de Bruijn graph framework.
Availability: CortexJDK can be downloaded from https://github.com/mcveanlab/CortexJDK and is released under the MIT license.
Contact: kiran@well.ox.ac.uk

1 Introduction

The initial construction stage of a de Bruijn graph (dBG) requires input sequences to be split into overlapping, fixed-length substrings of length k, or "k-mers" (1; 2; 3). This facilitates rapid discovery of overlaps between sequences (as these sequences can be stored in a hash table for later $O(1)$ lookup). The fixed-length property of k-mers also makes such graphs "stackable", facilitating easy comparison of multiple samples (4). However, relationships between non-adjacent k-mers in an input sequence are discarded (5). This limits the ability of dBG-based assemblers and variant callers to leverage long-range connectivity information inherent in existing reference assemblies, paired-end reads from second-generation sequencing platforms, and long reads or draft assemblies from third-generation sequencing.

This lost long-range connectivity information can be restored by aligning reads and/or pre-existing haplotype data to the graph and storing edge choices supported by the sequences at ambiguous junctions. We term these connectivity annotations 'links", and we have previously reported on a joint data structure to encode this information, which we termed a "linked de Bruijn graph" (LdBG) (6). The LdBG achieves the best of both worlds: it retains the stackability of dBGs while preserving the connectivity information required to discover longer and more complex variants. Links are stored in an auxillary file alongside the graph, ensuring that new connectivity data can be added at any time without requiring the graphs themselves to be reassembled.

A substantial implementation barrier still exists for developers wishing to create LdBG-based applications. The graph and links formats are not trivial to parse. Large file sizes in typical projects can make naïve approaches that attempt to load all data into memory intractable. Link-informed navigation requires careful tracking and expiry of annotations relevant to the current traversal, and complex heuristics are required to decide which of several links should be trusted to disambiguate an edge choice.

Table 1: CortexJDK core functionality, Java interfaces, and implementing classes

Functionality	Interface	Implementing class
Iteration and random access over a graph	DeBruijnGraph	CortexGraph
Dynamically merge multiple graphs	DeBruijnGraph	CortexCollection
Random access over connectivity data	Connectivity	CortexLinks
Perform (optionally link-enabled) simple walks and depth-first searches	-	TraversalEngine
Configure a traversal engine object in a variety of ways	-	TraversalEngineFactory
Extend to provide custom rules for ending a depth-first search	StoppingRule	AbstractStoppingRule

To remove these hurdles, we have developed CortexJDK, a pure Java library that provides low-memory random access to multi-color graph data and link annotations. We provide classes representing the common Cortex file formats and internal record structures. We also provide a versatile traversal engine class capable of performing simple walks through singly-connected vertices, and complex depth-first searches with arbitrary stopping conditions. Links are optionally loaded and followed by the traversal engine automatically, making link-informed software development trivial.

2 Features and methods

Core functionality and implementing classes are listed in Table 1 and summarized below.

2.1 Basic graph access (CortexGraph and CortexLinks)

A Cortex graph can be regarded as containing two components: a header (containing metadata on the graph and each color contained therein) and a records section (listing all the vertices and edges for each color in the graph). For a graph with c colors, each record consists of

$1 + 2c$ fields: the k-mer (in either forward or reverse-complemented orientation, whichever is lexicographically lowest), coverage per color, and the incoming/outgoing edges per color. Record k-mers are 2-bit encoded and packed into an array of 64-bit long int elements. Per color coverage is represented as integers, while per-color edges are represented as a 1-byte bitmask. The CortexGraph object encapsulates access to all of these components, storing the header in memory for quick queries, and providing an iterable interface to the records section. A full example program (excluding package imports) is provided in Algorithm 1.

Algorithm 1 An example program iterating over CortexGraph records

```
CortexGraph cg = new CortexGraph("example.ctx");
for (CortexRecord cr : cg) {
    System.out.println(cr);
}
```

McCortex (the successor to Cortex) is capable of emitting graphs in sorted order, which permits the graph to serve as a self-index. The CortexGraph object thus additionally provides a random access method, find(), to fetch any record via binary search given a k-mer. The McCortex links format, unfortunately, does not lend itself to this form of indexing. CortexJDK provides a tool to convert links to a MapDB3 database, enabling random access to this dataset as well. The CortexLinks object accepts both formats, presenting their contents with a single interface.

Special attention has been given to preventing overhead from redundant operations in typical use cases. Because it is common to request the same graph record multiple times (e.g. attempting to evaluate a section of the graph over many colors), several thousand records are stored in a least-recently-used (LRU) cache. Additionally, we lazily decode edge information upon request and cache results for later use. These steps reduce redundant disk access and provide improved IO performance.

2.2 Dynamic graph merging (CortexCollection)

When graphs are joined, per-color coverage and edge information is merged into a single record, making comparisons between colors trivial. However, in the course of a bioinformatic inquiry, one often discovers a quick analysis that would benefit from joining one graph with another. Performing the join for a one-off analysis might be time-consuming and wasteful. To this end, we provide the CortexCollection object, which dynamically merges multiple sorted graphs and presents them to the programmer as if they were a single file. This can have substantially more overhead than operating on a pre-joined graph (particularly if random access queries are performed, as now they need to be executed separately on each of the N files), but facilitates rapid inquiries on datasets. Used in concert with the CortexGraphWriter object, it also provides a means to join many other graphs and write the result to disk without first loading all graphs into memory. Algorithm 2 provides an example program that merges two sorted graphs into a single sorted output file.

Algorithm 2 An example program to join multiple sorted graphs

```
CortexGraph cg1 = new CortexGraph("graph1.ctx");
CortexGraph cg2 = new CortexGraph("graph2.ctx");
CortexCollection cc = new CortexCollection(cg1, cg2);

CortexGraphWriter cgw = new CortexGraphWriter("out.ctx");
cgw.setHeader(cc.getHeader());
for (CortexRecord cr : cc) {
    cgw.addRecord(cr);
}
cgw.close();
```

2.3 Simple and complex graph traversal (TraversalEngine)

Our low-memory, random-access implementation of graph and links classes permits straightforward graph exploration. We provide a configurable engine for conducting these explorations. The TraversalEngine object enables a user to specify a graph (and optionally, links) to operate on, a traversal color, and a variety of other optional arguments that influence the traversal. The simplest operation, fetching the successor to a vertex, simply requires calls to `seek(...)`, `hasNext()`, and next() methods (or `hasPrevious()` and `previous()` for fetching the predecessor). The `hasNext()` (`hasPrevious()`) method will return true if only one adjacency is available for `next()` (`previous()`) to return. These methods automatically disambiguate junction choices using links. The `walk(...)` method links these together, returning a unitig and taking care to avoid cycles.

Some applications require a more sophisticated traversal than a simple walk. For example, aligning a haplotype to a graph allowing for some errors requires a lookup of *k*-mers shared between the haplotype and the graph, as well as a traversal between gaps that ignores irrelevant branches. Or, when exploring the boundaries of a variant in a graph with a high error rate, one may wish to begin and end navigation at *k*-mers that deviate and rejoin another sample in the graph, regardless of intervening errors along the way. TraversalEngine provides a depth-first search method (`dfs(...)`) with programmatic stopping conditions to facilitate these use cases. A developer can specify a callback object which implements two methods: `hasTraversalSucceeded()` and `hasTraversalFailed()`. Both methods are executed each time a new vertex is added to a search graph, evaluating the current state of the traversal and determining whether success or failure criteria has been met (e.g. has the exploration of the current branch reached a desired vertex, has the current branch joined with another color, is the branch too long to reasonably believe it will reach a valid destination). If the former returns `true`, the current branch exploration is halted and added to the traversal graph. Should the latter return `true`, the current branch is halted and discarded. Should neither be true, exploration continues. Algorithm 3 shows an example use of the `dfs()` to build a unitig starting from the first *k*-mer in a graph, recapitulating the results of `walk(...)`.

Algorithm 3 An example program to perform a depth first search

```
/* UnitigStopper.java */
public class UnitigStopper extends
        AbstractTraversalStoppingRule¡CortexVertex, CortexEdge¿ {
    public boolean hasTraversalSucceeded(TraversalState s) {
        return s.numAdjacentEdges() != 1;
    }
    public boolean hasTraversalFailed(TraversalState s) {
        return false;
    }
}

/* DFSExample.java */
CortexGraph cg = new CortexGraph("test.ctx");
CortexLinks cl = new CortexLinks("test.ctp.gz");
String seed = cg.iterator().next().getKmerAsString();

TraversalEngine e = new TraversalEngineFactory()
    .traversalColor(0)
    .graph(cg)
    .links(cl)
    .stoppingRule(UnitigStopper.class)
    .make();

Graph¡CortexVertex, CortexEdge¿ g = e.dfs(seed);
```

2.4 Extensibility to other file formats

Other approaches to preserving sequence connectivity in genome graphs are available (7; 8). While the ideas expressed are similar, the implementations and file formats are very different. We are mindful of the fact that very few standardized file formats exist in the *de novo* assembly space, inhibiting code sharing and forcing developers to do redundant work. For this reason, we have endeavored to make CortexJDK as extensible as possible. While our software is named for the assembler upon which it was initially based, CortexJDK is in principle agnostic to the particulars of the file format. A developer wishing to operate on different file formats need only create a new classes that implements our DeBruijnGraph and Connectivity interfaces. All other aspects of the traversal engine automatically generalize to the alternative formats, permitting the same sophisticated traversal engine to be applied to other datasets.

3 Examples and discussion

Our code repository contains a number of example programs to help a new developer get started with their own applications. One interesting and under-served avenue is in graph visualization. While reference-based genome analyses have benefited from various visualization resources (9; 10; 11), graph-based offerings are more sporadic and typically incompatible (12; 13; 14). Graph file formats have not been standardized, and formats typically require one load the entire dataset into memory, making visualization challenging. As a demonstration of CortexJDK's utility, we provide an example implementation, VisualCortex. VisualCortex is a simple webserver that takes graphs and links as console input, binds to a specified local socket address, and provides a textbox wherein a kmer can be specified or requested at random. The local graph is explored in all colors and presented as an interactive force-directed graph using the d3.js Javascript visualization library. Figure 1 shows a screenshot from this application, visualizing an indel between two different samples and nearby irrelevant branches induced by sequencing error.

CortexJDK is a powerful yet simple API for performing random access on genome graph and link data, enabling rapid development of innovative tools in the graph genome space. Both the source code and a developer's guide is available at https://github.com/mcveanlab/CortexJDK. The software is released under the MIT license. Pull requests are welcome.

Acknowledgements

We thank Isaac Turner for making several changes to McCortex to facilitate CortexJDK operations, feedback from Jerome Kelleher on implementation details, and the rest of the McVean group for useful discussions during the preparation of this manuscript.

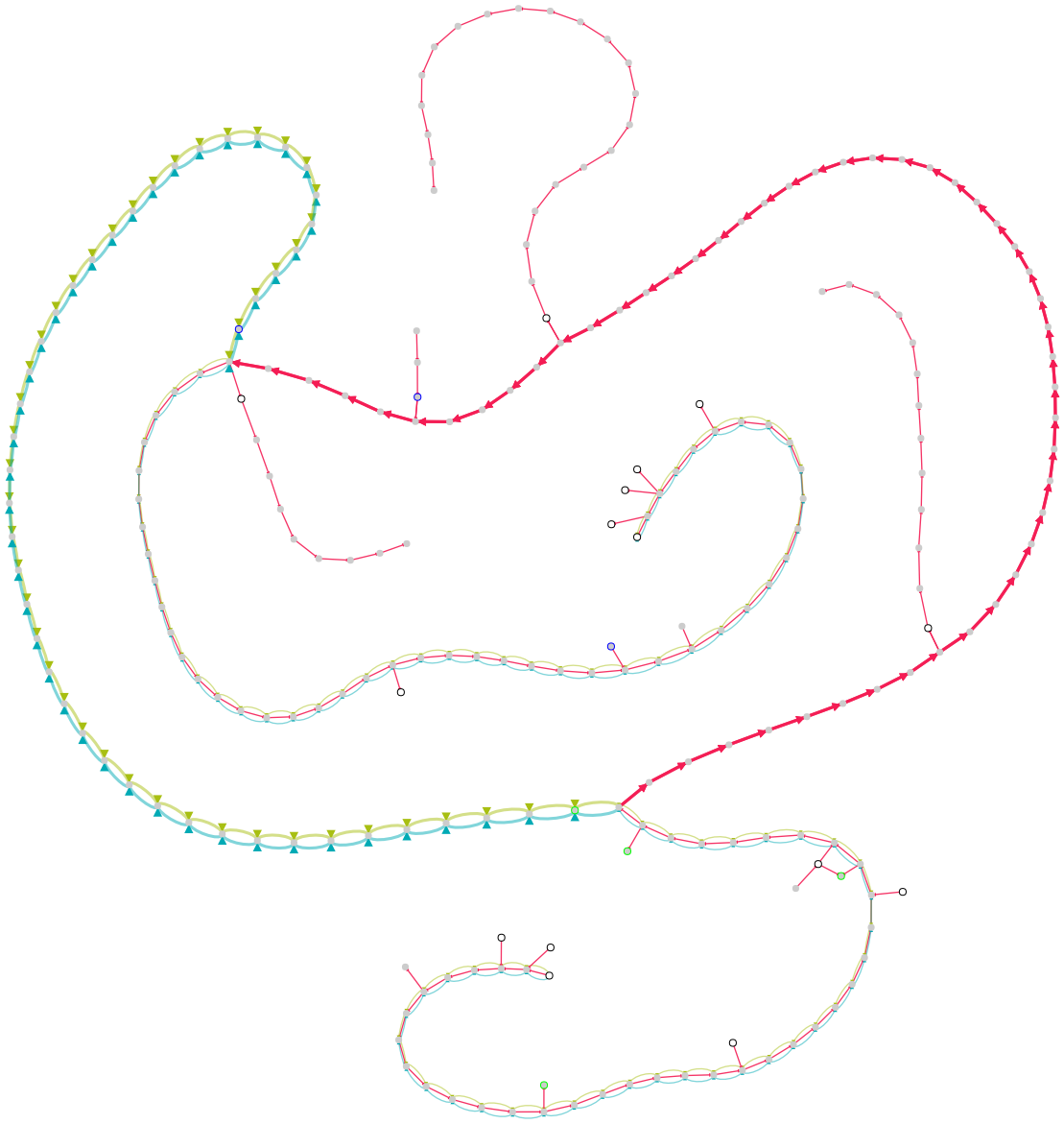

Figure 1: Visualization of a subgraph centered around an indel mutation between the red sample against the green/blue sample backgrounds. Vertices and edges involved in the mutation are drawn with thicker lines. Extraneous branches are shown to provide indicators of sequencing error.

Funding This work was supported by the Wellcome Trust (grant numbers 090532/Z/09/Z and 100956/Z/13/Z). K.V.G. was supported by Wellcome Trust Research Studentship award (097310/Z/11/Z). Z.I. was funded by a Wellcome Trust/Royal Society Sir Henry Dale Fellowship (grant 102541/Z/13/Z).

Conflicts of interest None.

References

[1] Bruijn, N. G. *A combinatorial problem* .

[2] Pevzner, P. A. l-Tuple DNA sequencing: computer analysis. *Journal of biomolecular structure {& dynamics* **7**, 63–73 .

[3] Idury, R. M. & Waterman, M. S. A new algorithm for DNA sequence assembly. *Journal of computational biology : a journal of computational molecular cell biology* **2**, 291–306 .

[4] Iqbal, Z., Caccamo, M., Turner, I., Flicek, P. & McVean, G. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics* **44**, 226–232 .

[5] Myers, E. W. The fragment assembly string graph. *Bioinformatics (Oxford, England)* **21**, ii79–ii85 .

[6] Turner, I., Garimella, K. V., Iqbal, Z. & McVean, G. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics (Oxford, England)* .

[7] Rozov, R., Goldshlager, G., Halperin, E. & Shamir, R. Faucet: streaming de novo assembly graph construction. *Bioinformatics (Oxford, England)* **34**, 147–154 .

[8] Bolger, A. M., Denton, A. K., Bolger, M. E. & Usadel, B. LOGAN: A framework for LOssless Graph-based ANalysis of high throughput sequence data. *bioRxiv* 175976 .

[9] Thorvaldsdóttir, H., Robinson, J. T. & Mesirov, J. P. Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in bioinformatics* **14**, 178–192 .

[10] Karolchik, D., Hinrichs, A. S. & Kent, W. J. *The UCSC Genome Browser*, vol. 37 .

[11] Nielsen, C. B., Cantor, M., Dubchak, I., Gordon, D. & Wang, T. Visualizing genomes: techniques and challenges. *Nature methods* **7**, S5–S15 .

[12] Wick, R. R., Schultz, M. B., Zobel, J. & Holt, K. E. Bandage: interactive visualization of de novo genome assemblies. **31**, 3350–3352 .

[13] Paananen, J. & Wong, G. FORG3D: Force-directed 3D graph editor for visualization of integrated genome scale data. *BMC Systems Biology* **3**, 26 .

[14] Pavlopoulos, G. A. *et al.* Visualizing genome and systems biology: technologies, tools, implementation techniques and trends, past, present and future. *GigaScience* **4**, 38 .