



Genome analysis

Efficient dynamic variation graphs

Jordan M. Eizenga ^{1,2,†}, Adam M. Novak ^{1,2,†}, Emily Kobayashi ^{1,3},
Flavia Villani ^{4,5}, Cecilia Cisar ^{1,2}, Simon Heumos ⁶, Glenn Hickey¹,
Vincenza Colonna⁴, Benedict Paten^{1,2} and Erik Garrison^{1,2,*}

¹Genomics Institute and ²Biomolecular Engineering and Bioinformatics, University of California Santa Cruz, Santa Cruz, CA 95064, USA, ³Bioinformatics and Systems Biology, University of California San Diego, La Jolla, CA 92093, USA, ⁴Institute of Genetics and Biophysics, Consiglio Nazionale di Ricerche, Naples 80131, Italy, ⁵Biotechnologie Mediche, Università degli Studi di Napoli Federico II, Naples 80138, Italy and ⁶Quantitative Biology Center (QBiC), University of Tübingen, Tübingen 72076, Germany

*To whom correspondence should be addressed.

†The authors wish it to be known that, in their opinion, the first two authors should be regarded as Joint First Authors.

Associate Editor: Peter Robinson

Received on April 21, 2020; revised on June 20, 2020; editorial decision on July 7, 2020; accepted on July 9, 2020

Abstract

Motivation: Pangenomics is a growing field within computational genomics. Many pangenomic analyses use bidirected sequence graphs as their core data model. However, implementing and correctly using this data model can be difficult, and the scale of pangenomic datasets can be challenging to work at. These challenges have impeded progress in this field.

Results: Here, we present a stack of two C++ libraries, `libbdsg` and `libhandlegraph`, which use a simple, field-proven interface, designed to expose elementary features of these graphs while preventing common graph manipulation mistakes. The libraries also provide a Python binding. Using a diverse collection of pangenome graphs, we demonstrate that these tools allow for efficient construction and manipulation of large genome graphs with dense variation. For instance, the speed and memory usage are up to an order of magnitude better than the prior graph implementation in the VG toolkit, which has now transitioned to using `libbdsg`'s implementations.

Availability and implementation: `libhandlegraph` and `libbdsg` are available under an MIT License from <https://github.com/vgteam/libhandlegraph> and <https://github.com/vgteam/libbdsg>.

Contact: erik.garrison@ucsc.edu

1 Introduction

As increasingly many individuals have been sequenced from certain species, the field of **computational pangenomics** has emerged to analyze whole populations of genomes rather than individual genomes (Computational Pan-Genomics Consortium, 2016). Much of the research in computational pangenomics has coalesced around graph-based approaches for representing populations of genomes (Paten *et al.*, 2017). Unlike conventional string-based representations, graph data structures can represent genomic variation like substitutions, insertions, deletions and other more complex genomic events.

Graph-based data structures present new computational challenges. In addition to sequence, genome graphs must represent topology. Given the size of many genomes, this can be quite demanding on computer memory. However, the total information content in a genome graph is only incrementally more than the sequences of the pangenome. This suggests that significant memory savings should be possible. There is also significant impetus to make the graph data structures computationally efficient, since they are frequently the core data structure in pangenomics applications.

Early versions of the variation graph toolkit (VG) (Garrison *et al.*, 2018) have provided a cautionary tale of a naïve implementation. VG used full-width machine words as identifiers for graph elements, and stored the elements and graph topology in a set of hash tables. Loading the 1000 Genomes Project's variant set into the VG toolkit used to consume more than 300 GB of memory, which is ~30 times as large as the serialized representation (Garrison, 2019).

Although VG provided a memory-efficient representation of the graph (XG) that could be used during read mapping and variant calling, this representation did not allow for dynamic updates to the graph. The dynamic implementation remained necessary for graph-modifying steps of VG pipelines, such as the original construction of the graph and augmenting the graph with novel variants. Some pipelines could be made feasible by breaking large graphs into connected components. However, this strategy reduces efficiency, and it is untenable for pangenome graphs that consist of a single component.

To overcome this limitation, we have developed three new graph genome data structures that are dynamic (allowing efficient updates and edits) and also memory-efficient for real world genome graphs. Here, we compare the performance of these data structures to the

original VG representation and to XG, using a diverse collection of genome graphs obtained during our work in graphical pangenomics.

In addition to demonstrating the possibility of working with large, complex graphs in small amounts of memory, these implementations expose a common API based on the HandleGraph model described below. This model provides an interface to genome graphs, based on their fundamental elements, which is intended to be implementable atop a broad diversity of graph storage designs. The VG toolkit has been refactored to use this API as its default means of loading, saving and manipulating graphs since version 1.22.0, allowing it to use any of the implementations presented here.

We have packaged these implementations behind equivalent C++ and Python APIs in `libbbds`. This software library will reduce the need for individual research groups to continually reimplement these core data structures and ease the development of algorithms that manipulate large, complex pangenome graphs. Moreover, the reduction in memory requirements makes it possible to move workloads that would otherwise need specialized high-memory machines onto cheaper ones that often also have more processing power (e.g. from Amazon's r4 instances to c5 instances). Combined with improvements in access speed over the previous VG dynamic graph implementation, substantial cost and time savings can be realized.

2 Implementation

2.1 Data model

Our libraries adopt node-labeled bidirected graphs as a formalism for sequence graphs. In a bidirected graph, nodes are considered to have left and right 'sides', and edges connect two sides rather than two nodes. In bidirected sequence graphs, a node's sides correspond to the 5' and 3' ends of its DNA sequence. Nodes can be traversed either from left to right, which is interpreted as the forward strand of the sequence, or from right to left, which is interpreted as the reverse complement. This provides a natural means to encode DNA strandedness.

Longer sequences can be formed by concatenating the sequences of multiple adjacent nodes together. These nodes form a path, which is defined as a list of oriented nodes (either forward or reverse), such that the graph contains an edge between the adjacent sides of each pair of subsequent oriented nodes in the list (Unlike the usage in many graph theoretic contexts, we do not intend the term path to indicate that these nodes must be distinct.). Some paths correspond to sequences of interest, such as reference genomes or annotations of the reference. Because paths like these are so frequently important in practice, our graph formalism also includes a set of paths along with the graph's node and edges.

2.2 The HandleGraph interface

The `libhandlegraph` library describes an interface that exposes basic operations on our sequence graph data model. The HandleGraph model focuses on five fundamental entities in bidirected sequence graphs (Fig. 1):

- *Nodes* identify pairs of complementary DNA strands and have unique numerical identifiers (IDs).
- *Strands* represent one strand of a node's DNA sequence.
- *Edges* link pairs of strands, in order.
- *Paths* represent sequences of interest as paths through the graph.
- *Steps* describe paths' visits to nodes' strands.

The defining feature of the model is that none of these entities are accessed directly. Instead, they are accessed via *handles*, which are references modeled after the concept of file handles. The handles are implemented as a data type with no methods and no prespecified meaning for its contents. Thus, we say that handles are 'opaque' in that user code cannot usefully look inside them or manipulate their contents. Instead, the `libhandlegraph` interface requires the sequence graph implementation to provide queries that consume and produce handles, to expose graph information to users.

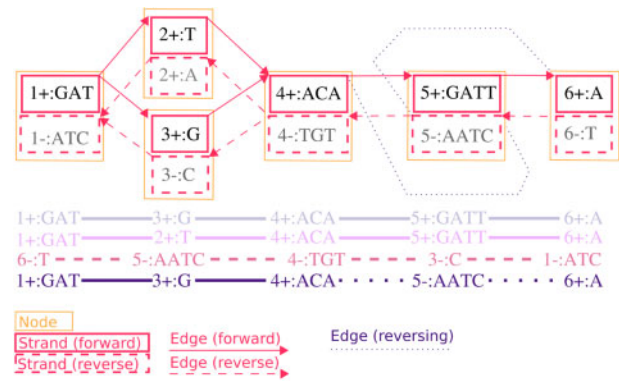


Fig. 1. Entities in the bidirected sequence graph. Top: a variation graph showing *nodes* (yellow rectangles), each of which contain a forward and reverse *strand* (red solid and dashed rectangles, respectively). Strands show the node identifier, the direction (+ or -) and the sequence of the strand. Note that reverse strands show the reverse complement sequence of the forward strand. All *edges* are shown as connections between nodes, with forward-to-forward edges denoted by solid lines, and reverse-to-reverse edges denoted by dashed lines. Two edges that invert from forward to reverse and reverse to forward are shown with dotted lines. Edges run from the strand at their beginning to that at their end, as indicated by the arrowhead. Bottom: an illustration of four *paths*. Each has a name, and can be referenced by a handle, which are omitted for brevity. Each path is shown in its natural direction as a series of connected *steps* that refer to strands in the graph. The first two paths differ by a SNP, with one passing through 2+:T, and the other through 3+:G. The third path is the reverse complement of the first. The fourth is the same as the first, but contains an inversion, passing through 5-:AATC rather than 5+:GATT. (Color version of this figure is available at [Bioinformatics](https://www.bioinformatics.org) online.)

For example, we could obtain a handle to a strand from a HandleGraph implementation by providing a node's ID and an orientation (forward or reverse). We could then provide this handle to another of the graph's methods to obtain handles to this strand's neighbors, and a further method would map the neighbors' handles to their node IDs. Alternatively, we could obtain a handle to a path from its name (e.g. 'chr22'), and then iterate over handles to the path's steps to follow its course through the graph.

One benefit of this design is that any algorithm designed for one HandleGraph implementation can be applied to all other implementations. Since the actual contents of a handle are unspecified, this benefit is achieved while simultaneously maintaining flexibility in the implementation. Another benefit is that, since the user works only through handles that they cannot forge or modify, their ability to make mistakes can be restricted. For example, the interface can enforce the constraints that define valid paths through bidirected graphs during edge traversal. Furthermore, implementations can be made memory-safe by eliminating raw pointers and other direct access to graph elements.

2.3 Graph implementations

We consider five implementations of the HandleGraph model. To ground our experimental results, here, we provide a high-level overview of each implementation. Two implementations, VG and XG, have been described previously (Garrison et al., 2018; Garrison, 2019). The others are combined in the `libbbds` library (<https://github.com/vgteam/libbbds>), which provides three concrete implementations: HashGraph, ODGI (Optimized Dynamic Graph Implementation) and PackedGraph. Each implementation represents a different tradeoff in terms of speed, memory use and capabilities. All of the implementations except XG are dynamic. They support efficient addition and deletion of nodes, edges, paths and steps, as well as some specialized methods such as splitting a node into multiple shorter nodes. Table 1 provides a high-level summary of the differences between the `libbbds` implementations.

2.3.1 VG

We have extended the graph representation in VG, previously described in the study by (Garrison et al., 2018), to match the

Table 1. Comparison of features between libbdsg graph implementations

Model	HashGraph	ODGI	PackedGraph
Design goal	Simplicity, speed	Memory efficiency	Balanced speed/memory
Topology data structure	Hash table	Single integer vector	Several integer vectors
Topology compression	None	Delta encoding	Windowed bit compression
Sequence compression	None	None	Bit compression
Pointer encoding	Memory addresses	Delta-encoded ranks	Vector indexes

Note: The three graph implementations all use adjacency lists to encode graph topology and linked lists to encode paths. The differences in encoding these structures reflects different design goals for each implementation.

HandleGraph API. The backing data structures used remain the same. The graph entities are stored as objects in a backing vector, and referred to internally by hash tables that map between node identifiers and pointers into this vector. Edges are indexed in a hash table mapping pairs of handles to edge objects. Paths are stored in a set of linked lists, with a hash table mapping between nodes and path steps. This arrangement was tenable for the early development of algorithms working on variation graphs. Its inefficiency, caused by unnecessary overheads and data duplication, has resulted in significant difficulties for groups working with VG. The other HandleGraph implementations respond to the limitations of this approach. In version 1.22.0, vg was updated to use HashGraph (below) as the default format, though it remains compatible with all implementations described in this article via the HandleGraph API.

2.3.2 XG

XG was initially developed in response to the memory and runtime costs of VG, which prevent its application to large graphs. It additionally provides positional indexes over paths that are required for read mapping and variant calling, and is the graph data model used in most established bioinformatic operations on variation graphs (Garrison *et al.*, 2018; Hickey *et al.*, 2020). Unlike other HandleGraph implementations, XG is a static graph index. This permits a more powerful set of efficient queries against the graph, especially for paths. The encoding is designed to balance speed and low-memory usage. The topology of the graph is encoded in a single vector of bit-compressed integers, which promotes cache efficiency. Rank and select operations on succinct bit vectors are used to provide random access over the variable-length records, which each encode a node's sequence, ID and edges. Embedded paths are encoded in variable-length integer vectors with Elias gamma encoding. Rank and select operations on succinct bit vectors also provide queries by base-pair position along paths. A detailed description of XG can be found in the study by Garrison (2019).

2.3.3 HashGraph

HashGraph is a relatively simple encoding, which is largely similar to the original VG graph. As such, it can be seen as a streamlined point of comparison for the other new dynamic graph implementations. However, the simplicity of this encoding has the benefit of allowing fast queries. Thus, even though HashGraph still has relatively high memory requirements, it can still be useful in high memory compute environments or for small sequence graphs (such as subgraphs of genome graphs).

Like VG, HashGraph encodes the topology of the graph in a hash table indexed by node IDs. However, what were separate hash tables in VG have been consolidated to avoid storing the keys multiple times. The hash table it uses is a drop-in replacement for the equivalent standard library (STL) data structure, and has been shown to outperform it in empirical evaluations (Brehm, 2019). Each hash table entry contains the sequence, an adjacency list of the edges in two STL vectors, and a vector indicating the path steps that the node can be found on. The graph's paths are represented using doubly linked lists to support efficient modification at any position.

In contrast to the more memory-efficient implementations, all of these data structures support computation in their native in-memory representation. Thus, the run time to access graph elements does not

also include decompressing the data. This is how HashGraph maintains its comparative speed advantage.

2.3.4 Optimized dynamic graph implementation

ODGI is based on a node-centric encoding that is designed to improve cache efficiency when traversing or modifying the graph. This encoding is split between graph topology and paths, which is important for achieving a balance of runtime performance and memory usage on graphs with large path sets. It uses delta encoding of edges and path steps to reduce the cost of representing graphs with local partial order and sparsity, both of which are common features of pangenome graphs. ODGI is the default data model of the ODGI toolkit (<https://github.com/vgteam/odgi>), which provides high-level algorithms for graph manipulation and interrogation that are designed to work at the scale of large pangenomes.

In ODGI, each node $\mathcal{N} = (\mathcal{B}, \mathcal{P})$ is represented by a structure that contains a byte array $\mathcal{B} = (\mathcal{Q}, \mathcal{E})$ encoding its sequence and associated edges, and a compressed integer vector $\mathcal{P} = \mathcal{S}_1 \dots \mathcal{S}_s$ describing the path steps that traverse it. The full graph model is simply an array of these node records $\mathcal{G} = \mathcal{N}_1 \dots \mathcal{N}_{|\mathcal{G}|}$ with some additional data structures to allow for random access of paths by name, and to maintain important statistics about the size of the graph, its node ID space, and its path set.

Each node's sequence \mathcal{Q} is stored using a full byte per character at the start of the byte array \mathcal{B} . This allows ODGI to represent protein as well as DNA sequence graphs, and allows for copy-free reference to the node sequences. The edges that begin or end at the node are recorded in the remainder of \mathcal{B} , encoded as deltas between the rank of the other end of the edge and the current node.

ODGI stores paths as bidirectional linked lists that allow efficient insertions, deletions and replacements of path steps. These paths are encoded in a manner that exploits common properties of pangenome graphs, and node-level data structures are organized to support efficient operation on graphs with very deep path coverage. The path steps $\mathcal{P} = \mathcal{S}_1 \dots \mathcal{S}_s$ on each node are recorded as a series of records in a dynamic integer vector which is compressed so that only the largest integer entry is stored at full bit-width (Prezza, 2017). Each step $\mathcal{S} = (p_{id}, \delta_p, \delta_n, r_p, r_n)$ contains a path identifier p_{id} , references to the previous δ_p and next δ_n node ID and strands on the path encoded as deltas relative to the current node, and the ranks of the previous r_p and next r_n steps among the path steps on their respective nodes. This path encoding scheme is similar to that used in the dynamic GBWT (Sirén *et al.*, 2019), but differs in that the paths are not prefix sorted.

2.3.5 PackedGraph

PackedGraph is designed to have a very low-memory footprint. The backing data structures are implemented using bit-compressed integer vectors. The bit-width of these vectors is chosen dynamically, starting with a bit-width of 1 and then reallocating the vector at a higher width whenever an edit operation introduces an integer that is too large to be represented with the current width. In the typical case that the value of i th entry in the vector is $O(i)$, these reallocations have an $O(1)$ amortized run time per edit.

Many of the integer vectors tend to also have entries that are highly correlated with their neighbors. PackedGraph exploits this characteristic to achieve greater compression by only storing one

entry per fixed-size window at full bit-width. The rest of the entries are stored in a separate integer vector and expressed as a difference from that entry. Since the differences within a window tend to be small, this encoding keeps the bit-width for each window small as well.

The data associated with each node is recorded in several compressed integer vectors (at the same index in each). Contrast XG and ODGI, which encode data in a single vector to improve cache efficiency. Recording only one homogenous data type in a vector increases the correlation between neighboring values, which in turn improves compression. The adjacency list for the graph, the steps that each node is found on, and the paths themselves are represented using linked lists. The linked lists reside within the same bit-compressed integer vectors, where pointers are created by treating some integer entries as indexes into the vector itself. This pointer encoding also guarantees the technical condition that accessing the i th entry is $O(i)$. The linked lists that occur on every node (the adjacency lists and node step lists) are included in a single vector across all nodes. This serves two purposes. First, the windowed compression scheme in the integer vectors is inefficient if lists are smaller than the window size, as is often the case. Second, due to the local partial order that is found in many pangenome graphs, neighboring nodes often connect to the same nodes and are found on the same paths as each other. Thus, the values they store are also highly correlated.

2.4 Python binding

We have implemented a Python binding to the graph implementations in `libbdsg` using `Pybind11` (Jakob et al., 2017). This allows the data structures to be used in Python applications, significantly lowering the barrier-to-entry for pangenomic application developers. This functionality is documented at <https://bdsg.readthedocs.io>, including a tutorial. This documentation also serves as useful introduction to the `HandleGraph` API.

2.5 Code availability

Both `libhandlegraph` and `libbdsg` are open sources under an MIT License. They are available on GitHub at <https://github.com/vgteam/libhandlegraph> and <https://github.com/vgteam/libbdsg>. Documentation for the two libraries, including the C++ handle graph API, `HashGraph`, `ODGI` and `PackedGraph`, is available at <https://bdsg.readthedocs.io> alongside the documentation for the Python binding.

3 Evaluation

3.1 Human genome with structural variants

We measured the core operation performance of the four graph implementations and the graph class from the popular VG software (as implemented prior to version 1.22.0). In particular, we measured (i) memory usage to construct a graph, (ii) time to construct a graph, (iii) memory usage to load an already-constructed graph and (iv) time to access nodes, edges and steps of a path. These access operations are one of the major drivers of run time in pangenomic applications, such as VG's read mapping algorithm. Accesses were performed with a single thread, and the reported access time is the average time taken when accessing each graph element sequentially. All evaluations were performed on a 3.1 GHz Intel Xeon Platinum 8000 series processor. The presented results are from a graph describing the structural variants of the Human Genome Structural Variation Consortium (Chaisson et al., 2019), which was recently used to genotype structural variants (Hickey et al., 2020). Specifically, the graph consists of the GRCh38 primary scaffolds and 72 485 indel variants ranging in size from 50 bp to 76 kbp. The results generally match our expectations based on the implementations' design goals (Fig. 2).

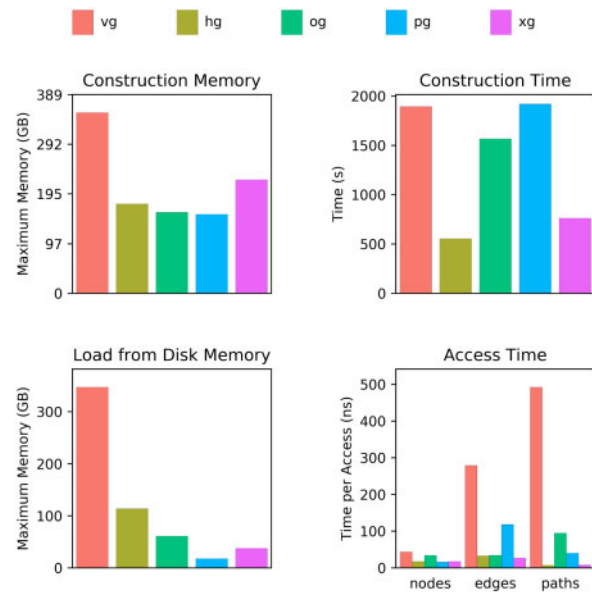


Fig. 2. Performance on a graph of structural variants from the HGSC. Abbreviations used here and in subsequent figures and tables: vg, VG; hg, HashGraph; og, ODGI; pg, PackedGraph; xg, XG. All four new graph implementations compare favorably to VG. PackedGraph tends to be the most memory efficient, HashGraph tends to be the fastest, and ODGI is balanced in between. XG provides good performance on both memory usage and speed, but it is static

3.2 Genome graph collection

To compare the methods' performances across a wide variety of different graphs, we applied each to a collection of 2299 graphs collected during our research on graphical pangenomics. For each graph and graph implementation, we measured the same metrics described in the previous section as well as various graph properties including size, edge count, cyclicity and path depth. We summarize these results in Figures 3 and 4.

For graph construction and loading, we observe similar trends as for the HGSC graph. VG's performance in terms of memory usage is very poor, both during construction and load. For construction and load, all models exhibit largely linear scaling characteristics, outside of very small graphs where static memory overheads dominate. PackedGraph yields the best memory performance for larger graphs (which are mostly the chromosomes of the 1000 Genomes Project graph), while for the medium-sized graphs in the collection (~ 1 Mbp), ODGI requires less memory.

For graph queries and iteration, the relative performance of the models is largely maintained across the entire range of graph sizes. However, we observe that the hash-based models (VG and HashGraph) have very good performance for smaller graphs (in handle and edge enumeration) but decrease in throughput as the graph size increases. Smaller, less dramatic decreases in performance can be seen for the other implementations. For path enumeration, the highest-performing methods are XG and HashGraph at approximately 10 times faster than ODGI, whose relative path storage is costly to traverse.

3.3 1000 Genome Project chromosome graphs

Variation graphs built from the 1000 Genomes Project (1000GP) variant catalog and the human reference genome have fairly homogenous and regular features. In addition, they have connected components of very different sizes, each corresponding to a chromosome. This provides a natural, fairly controlled means to explore the scaling behavior of our data structures. Moreover, graphs of this form are seeing increasing use in variant-aware resequencing analyses (Crysnanto and Pausch, 2019). Thus, the performance of data structures on these graphs is of general interest.

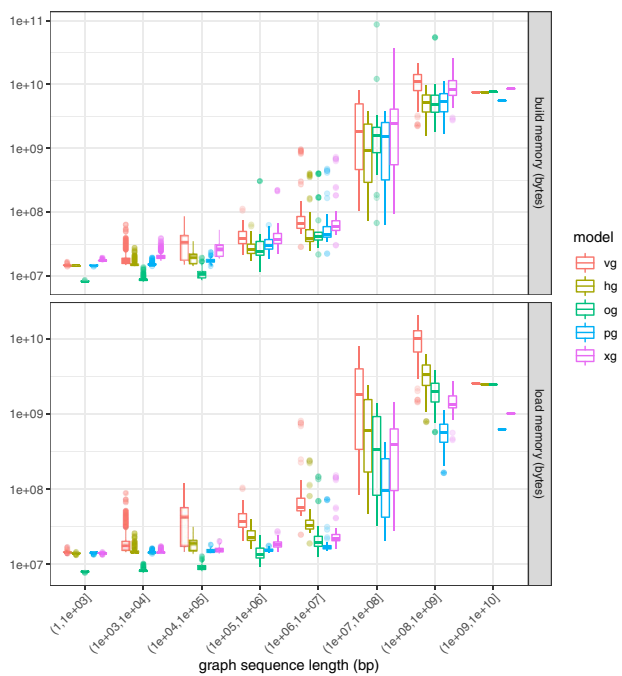


Fig. 3. Memory requirements for model construction and loading. Memory costs versus graph sequence size for the graph collection, colored by HandleGraph model. The memory requirements for graph construction tend to be higher than those for loading the graph model. All methods show fixed overheads of several megabytes, seen in the flat tail to the left of both plots. Outside of this region, all methods show roughly linear scaling in both build and load costs per input base pair. The relative differences in memory costs appear to be stable between different methods across many orders of magnitude in graph size. (Color version of this figure is available at *Bioinformatics* online.)

We first evaluated the scaling performance of the various HandleGraph implementations relative to node count for each of the nuclear chromosomes in the 1000GP (Fig. 5). We find that for all methods, load memory scales almost perfectly with node count, with an average $R^2 = 0.998$. Due to differences in variant density among the chromosomes, the average correlation relative to sequence length is lower ($R^2 = 0.986$).

In Table 2, we report the average memory performance of the methods relative to graph sequence length, and also the iteration performance in terms of elements per second. We find that the best-performing method in terms of memory usage is PackedGraph, which consumes around 1/20th the memory of VG per basepair of graph in the 1000GP set. Moreover, it provides much better iteration performance for nodes (handles), edges and path steps. HashGraph and XG have similar iteration performance, but XG, by virtue of its use of compressed, static data structures, requires less than half as much memory. ODGI optimized for efficient dynamic operations on graphs with higher path coverage, and in general is not as performant as other methods on this set.

4 Discussion

We have presented a set of simple formalisms, the HandleGraph abstraction, which provides a coherent interface to address and manipulate the components of a genome variation graph. To explore the utility of this model, we implemented data structures to encode variation graphs and matched them to this interface. This allowed us to directly compare these HandleGraph implementations on a diverse set of genome graphs obtained during our research. These experiments reveal that genome graphs need not pay the computational expense of the early versions of VG. The best-performing models require an order of magnitude less memory than VG while providing higher performance for basic graph access operation and element iteration. For these reasons, VG has transitioned to using these newer graph implementations.

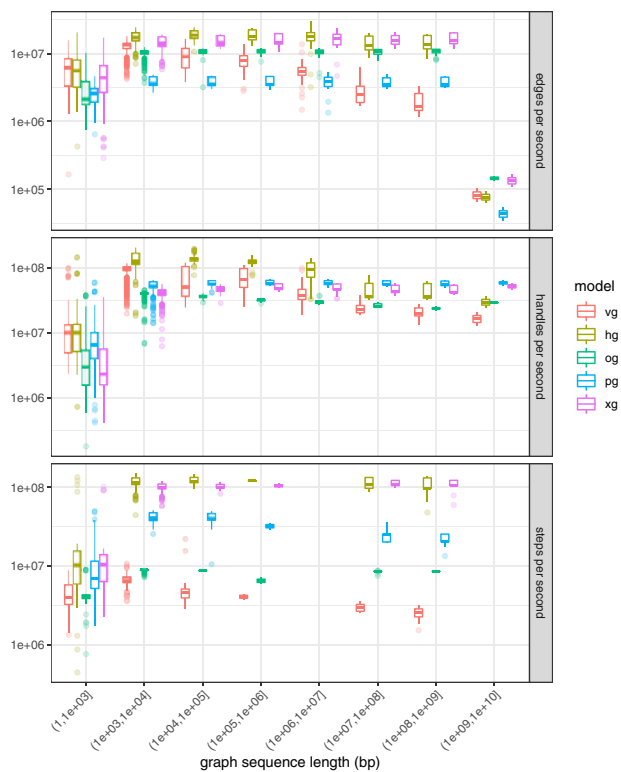


Fig. 4. Graph element enumeration performance. Iteration performance for edges, nodes and path steps for the full graph collection, shown in terms of elements per second. HashGraph provides the highest performance for all element iteration types on smaller graphs, but this performance falls off with larger graphs, presumably due to scaling properties of the backing hash tables. The same pattern can be seen for VG, although the overall performance is worse. Although it has the worst edge iteration performance, PackedGraph provides good performance on node and path step iteration. The relative path encoding in ODGI yields poor performance on path iteration, and node decoding overheads appear to reduce its node iteration performance, but it has good graph topology traversal performance, perhaps due to cache efficiency of the edge encoding. XG provides excellent iteration performance in all cases

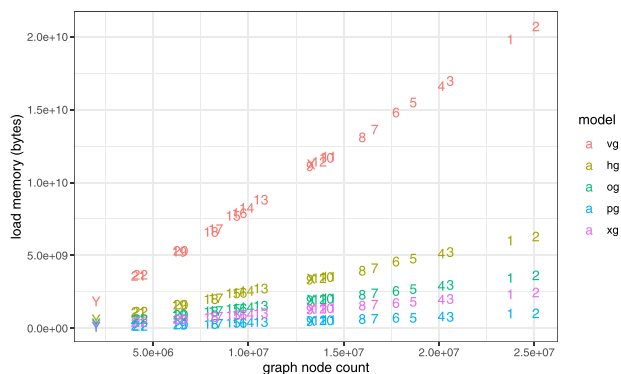


Fig. 5. Load memory versus node count for chromosome graphs built from 1000 Genomes Project variants and GRCh37. For each method, memory requirements are more strongly correlated with the number of nodes in the graph ($R^2 = 0.998$) than with the graph sequence length ($R^2 = 0.986$). Although the memory requirements are dominated by graph sequence size, node count will increase with variant density. Methods generally incur an overhead for each node that is larger than the sequence length. Linear scales clarify that the absolute difference in performance between VG and the other methods is substantial

The efficiency of these methods and their encapsulation within a coherent programming interface will support their reuse within a diverse set of application domains. Variation graphs have deep similarity with graphs used in assembly; these libraries could be used as

Table 2. Performance on 1000 Genomes Project chromosome graphs

Model	Build	Load	Iteration rate (millions)		
	B/bp	B/bp	Node/s	Edge/s	Step/s
vg	80.2	77.2	24.6	2.8	2.9
hg	36.7	23.9	59.5	18.9	127.2
og	30.3	13.7	24.1	11.5	8.2
pg	37.6	3.80	63.7	4.6	24.3
xg	54.3	9.31	54.2	20.5	117.0

Note: Average build memory, load memory and iteration times for graph elements for the chromosome-level graphs built from all the variants in the 1000 Genomes Project and the GRCh37 reference genome against which the variant set was originally reported. VG requires ~20 times as much memory to load the graphs as PackedGraph, while even the most costly libbdsf model (HashGraph) requires ~1/3 as much memory. In these graphs, ODGI provides the lowest performance for handle iteration. However, in all other metrics, VG performs much worse than the other models. Bold indicates lowest memory usage or fastest operation.

the basis for assembly methods. They could also be used for genotyping and haplotype inference methods based on graphs (Garg et al., 2018).

Ongoing work is establishing large numbers of highly contiguous whole genome assemblies for humans (<https://humanpangenome.org/>). Improvements in sequencing technology are likely to make such surveys routine. It is natural to consider a pangenome reference system, based on the whole genome alignments of such assemblies, as the output of these pangenome projects. Recent results demonstrate that many basic bioinformatic problems can be generalized to operate on such structures. Should these pangenome representations become common or standard, then variation graph data structures like those we have presented here will form the basis for a wide range of pangenomic methods.

Funding

This work was supported, in part, by the National Institutes of Health (award numbers U01HG010961, U41HG010972, R01HG010485, 2U41HG007234, 5U54HG007990, 5T32HG008345-04, U01HL137183 to B.P.) and the W. M. Keck Foundation (award number DT06172015 to B.P.). S.H. acknowledges funding from the Central Innovation Programme (ZIM) for SMEs of the Federal Ministry for Economic Affairs and Energy of Germany.

Conflict of Interest: none declared.

References

- Brehm, W. (2019) Hash tables with pseudorandom global order. *INFOCOMP J. Comput. Sci.*, **18**, 20–25.
- Chaisson, M.J. et al. (2019) Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nat. Commun.*, **10**, 1784.
- Computational Pan-Genomics Consortium. (2016) Computational pan-genomics: status, promises and challenges. *Brief. Bioinf.*, **19**, 118–135.
- Crysnanto, D. and Pausch, H. (2019) Sequence read mapping and variant discovery from bovine breed-specific augmented reference graphs. 10.1101/2019.12.20.882423.
- Garg, S. et al. (2018) A graph-based approach to diploid genome assembly. *Bioinformatics*, **34**, i105–i114.
- Garrison, E. (2019) *Graphical Pangenomics*. Ph.D. thesis, University of Cambridge.
- Garrison, E. et al. (2018) Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nat. Biotechnol.*, **36**, 875–879.
- Hickey, G. et al. (2020) Genotyping structural variants in pangenome graphs using the vg toolkit. *Genome Biol.*, **21**, 1–17.
- Jakob, W. et al. (2017) pybind11 – seamless operability between c++11 and python. <https://github.com/pybind/pybind11>.
- Paten, B. et al. (2017) Genome graphs and the evolution of genome inference. *Genome Res.*, **27**, 665–676.
- Prezza, N. (2017) A framework of dynamic data structures for string processing. In *International Symposium on Experimental Algorithms*. Leibniz International Proceedings in Informatics (LIPIcs). King's College, London, UK.
- Sirén, J. et al. (2019) Haplotype-aware graph indexes. *Bioinformatics*, **36**, 400–407.