Easy and Efficient Variation Graphs

UNIVERSITY OF CALIFORNIA SANTA CRUZ Genomics Institute

*Jordan M. Eizenga, *Adam M. Novak, Emily Kobayashi, Flavia Villani, Cecilia Cisar, Simon Heumos, Glenn Hickey, Vincenza Colonna, Benedict Paten, and Erik Garrison

Photo Credit: Karolina Karlic (Arts + Genomics Initiative)

Why Variation Graphs?

With our growing understanding of the genetic variation, especially large-scale structural variation, between individuals, it has become clear that even a single perfect haploid genome assembly is insufficient for doing genomics.

Some genomicists have opted to create ethnicity-specific reference genomes, to try and improve on the single reference assembly when working in their ethnicities of interest. Because the field of human genetics has a long history of racism, and deep connections to eugenics, this is a dangerous approach. Moreover, as variation between individuals exceeds that between ethnic groups, an approach that can capture individual variation is needed.

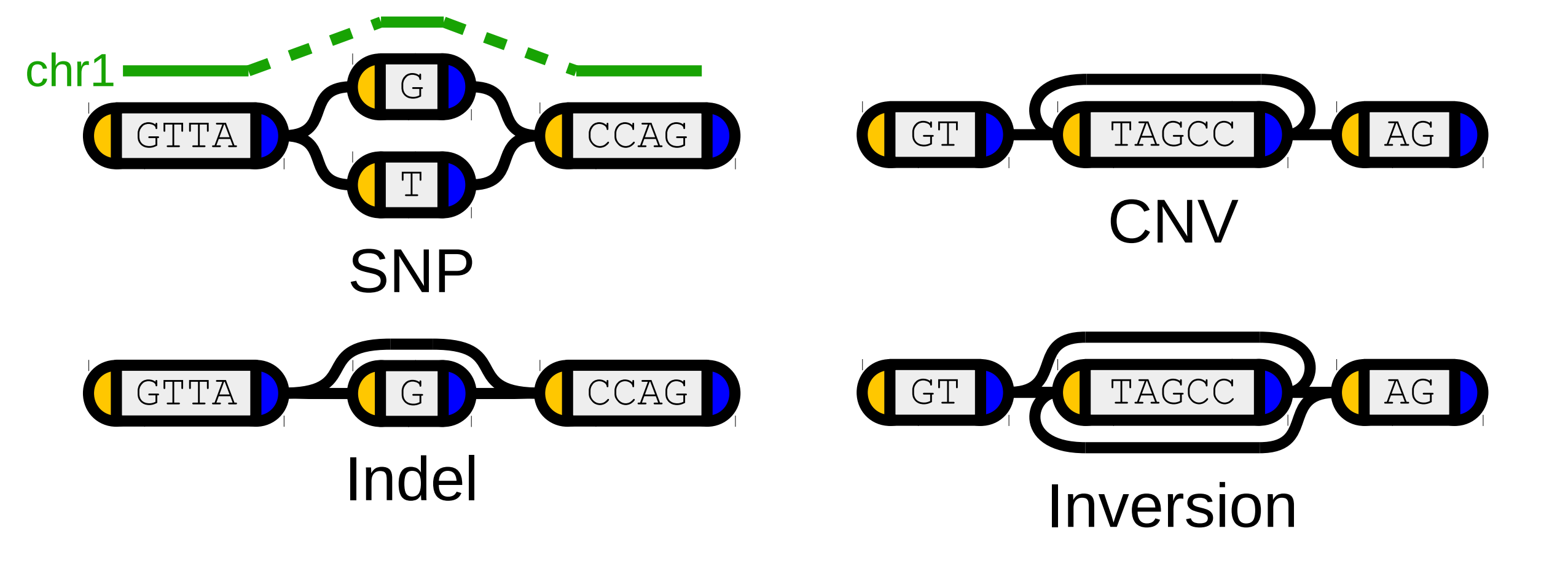
Variation graphs and other pangenomics tools offer an alternative to ethnicity-specific reference genomes.

“There is a slippage... between the fact of individual variation and the presumed importance of ‘national’ or ‘ethnic’ reference genomes.”

“[G]enome graphs enable diversity to be represented without automatically reinforcing notions of “national and ethnic” biological differences.”

Kowal, E., & Llamas, B. (2019). Race in a genome: long read sequencing, ethnicity-specific reference genomes and the shifting horizon of race. *Journal of Anthropological Sciences*, 97, 1-16.

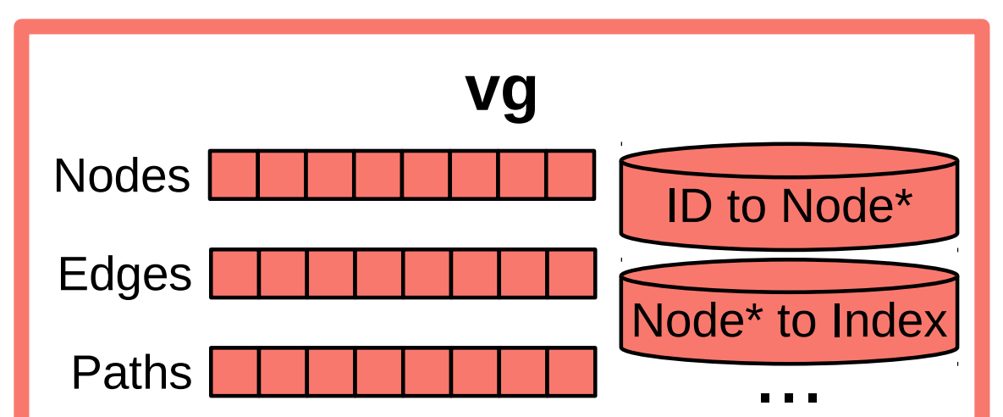
Variation Graphs



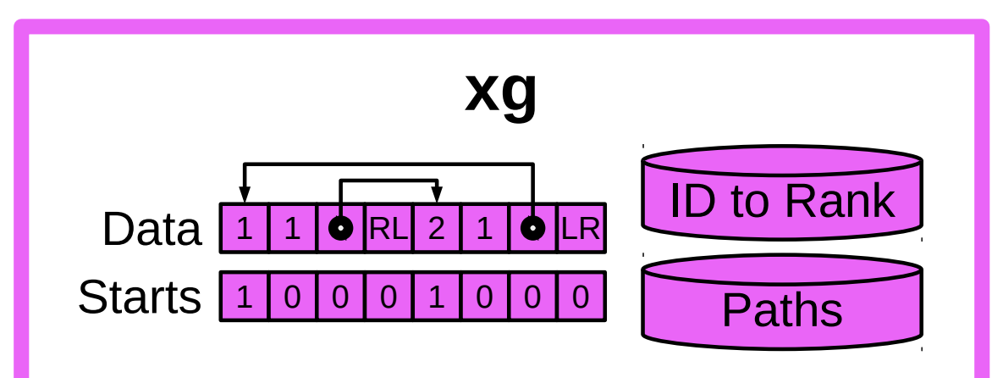
Bidirected sequence graphs (left and right sides)
Embedded paths (chr1)

Project Motivation

The vg graph genomics toolkit had two first-generation graph implementations, with no common interface.



Node, Edge, and Path objects in separate arrays, with duplicative hash tables.



Static vector encodes nodes and offsets to connected nodes. Lookup is fast by start offset but slow by ID via bit vector select().

Methods

vg originally thought of graphs as collections of nodes and bidirected edge tuples. However, edge tuples were ordered, by default connecting the end of a “from” node to the start of a “to” node. This made talking about the edges of a node awkward, because there were always two collections: the edges where the node was “from” and where it was “to”. Self loops could be in both sets, and getting all the edges of one side of a node, which turned out to be the most basic operation, was even more complex.

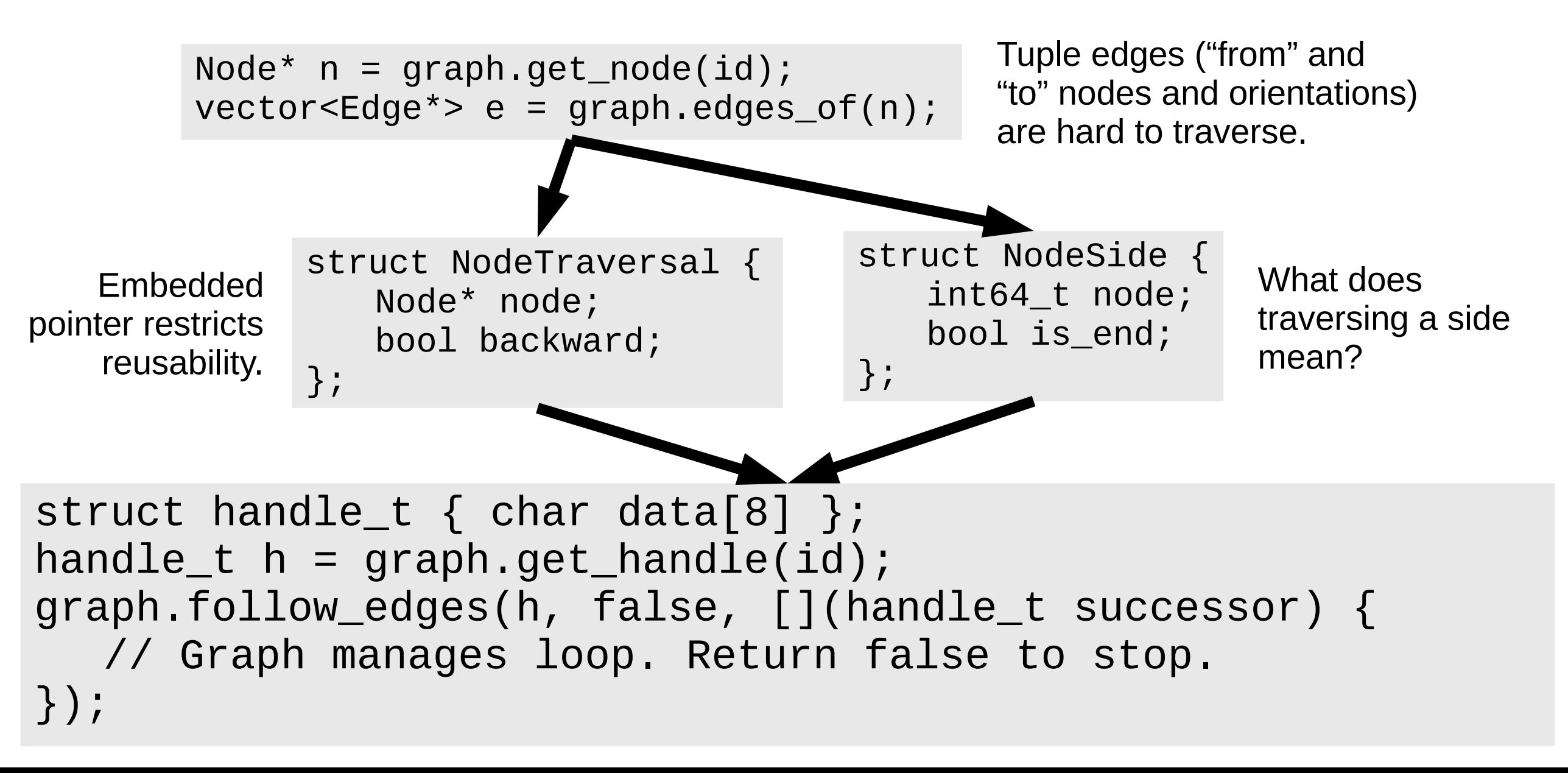
To make edge bookkeeping easier, the NodeSide type was developed; an edge could then be thought of as connecting an unordered pair of NodeSides. However, the explicit semantics of representing a side of a node made it awkward to use when traversing the graph.

To idiomatically represent traversing a node, the NodeTraversal type was developed. However, using a pointer to a node restricted it to being used with only a single graph implementation.

Moreover, the vg tool had two graph implementations, and, even if an algorithm used the implementation-independent NodeSide, the method calls needed to traverse edges were completely different. To resolve these issues, the concept of a “handle” was developed. A handle represents an oriented reference to a node, in a format appropriate for the graph it belongs to. Graphs that use handles implement a common “HandleGraph” interface, exposing operations to get handles to nodes (in either orientation), to exchange handles for information about the referenced node (such as its ID and sequence), and to inspect the edges of the graph (by starting at a handle, moving either left or right in its local orientation, and receiving all the possible destination handles via callback).

Additional derived interfaces provide support for embedded paths (via “path handles” and “step handles”), for graph modification and node removal, and for more specialized features (such as 0-based densely-assigned ranks) available only in some graph implementations.

Evolution of Graph APIs in vg



```
Node* n = graph.get_node(id);
vector<Edge*> e = graph.edges_of(n);
```

Tuple edges (“from” and “to” nodes and orientations) are hard to traverse.

Embedded pointer restricts reusability.

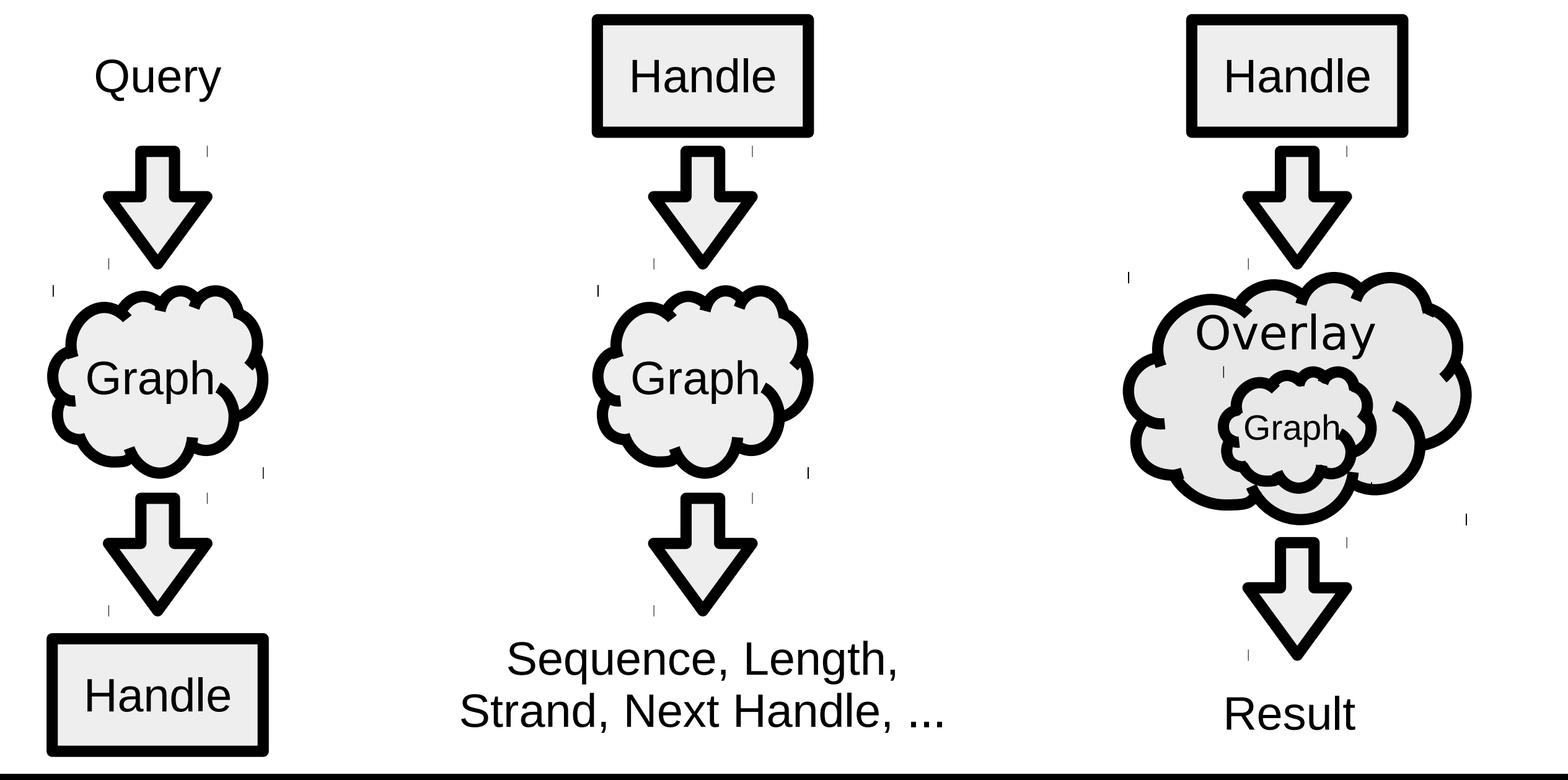
```
struct NodeTraversal {
    Node* node;
    bool backward;
};
```

```
struct NodeSide {
    int64_t node;
    bool is_end;
};
```

What does traversing a side mean?

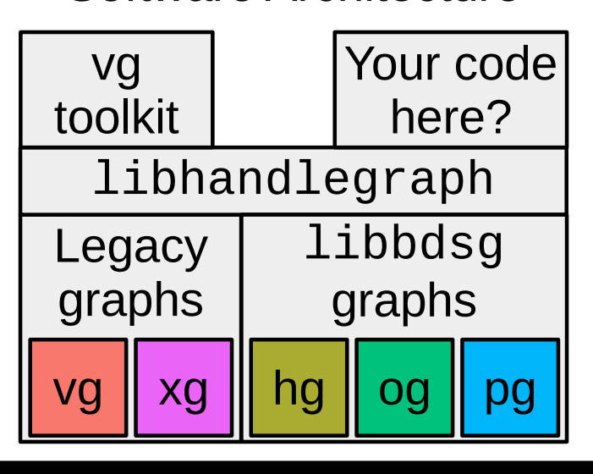
```
struct handle_t { char data[8] };
handle_t h = graph.get_handle(id);
graph.follow_edges(h, false, [](handle_t successor) {
   // Graph manages loop. Return false to stop.
});
```

How HandleGraphs Work

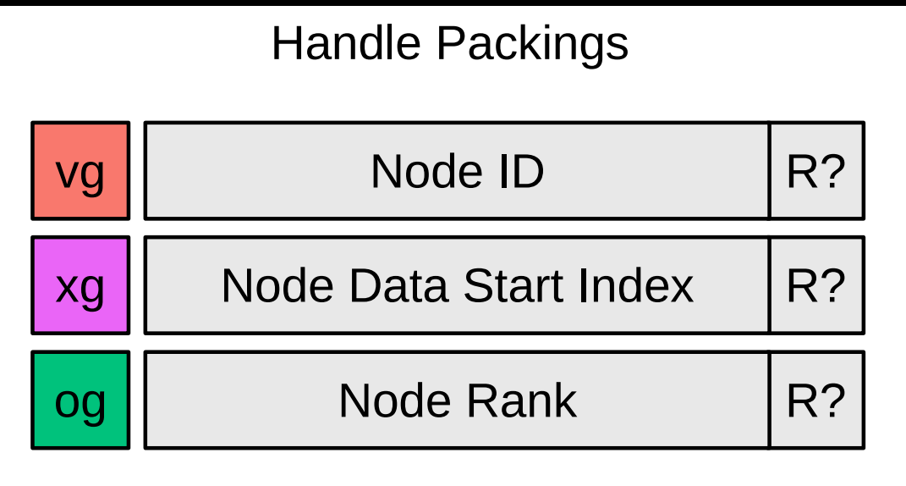


HandleGraphs never expose nodes themselves, only handles. The graph is responsible for fulfilling requests for information about nodes referenced by handles, such as their sequences and their edge connectivity.

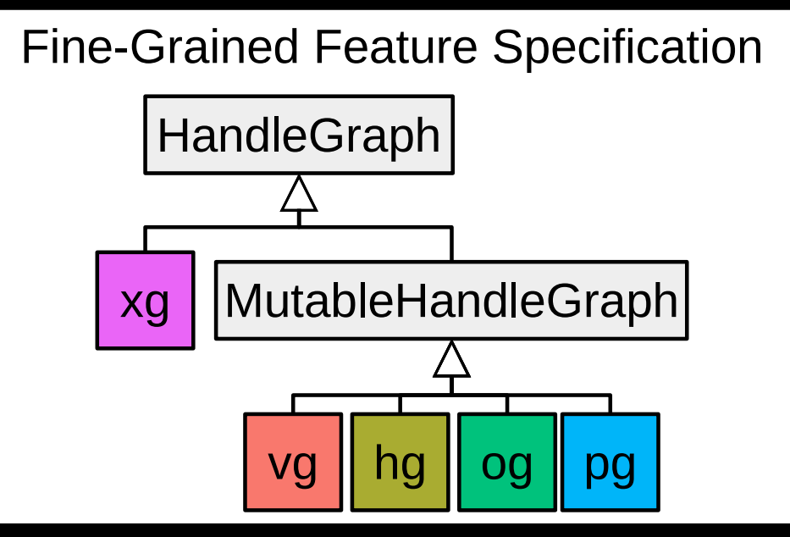
This can be exploited to construct “overlay” graphs, which can add, remove, duplicate, or transform the nodes of an underlying graph lazily, as queries are made. No graph data needs to be copied or modified, and overlay nodes do not need to be instantiated or stored.



libhandlegraph bridges algorithms and swappable graph implementations.

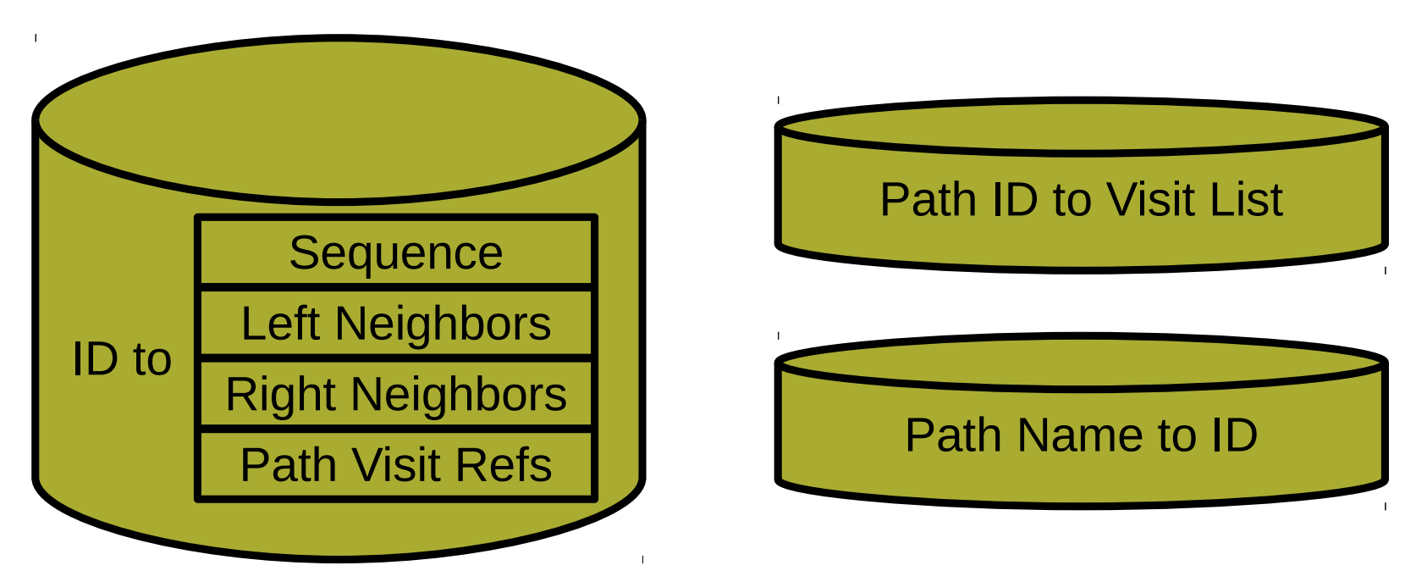


Different graph implementations stash different data in the fixed-size handles.



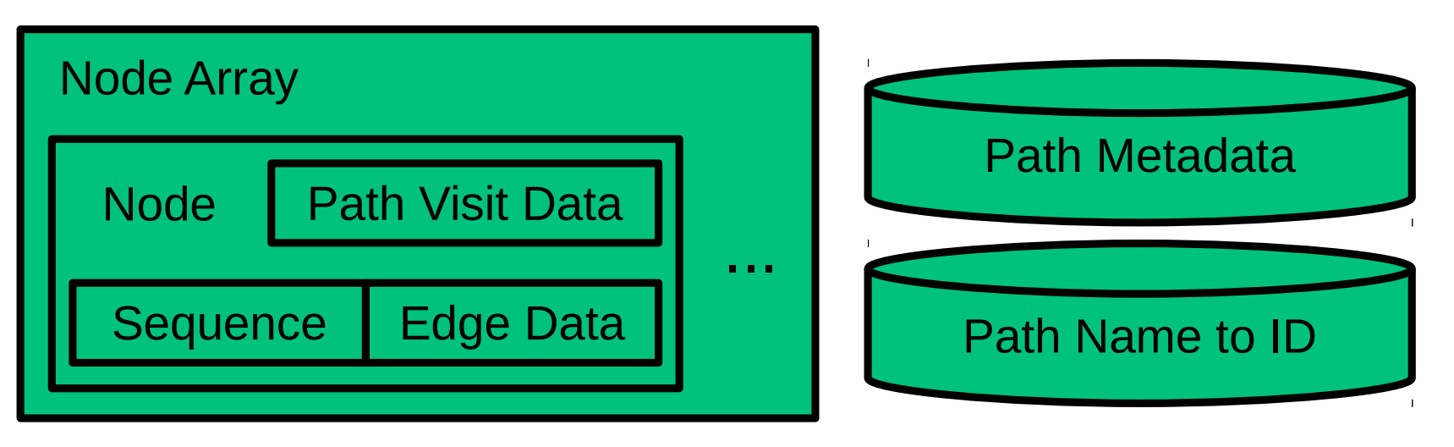
Graph Implementations in libbdsg

HashGraph



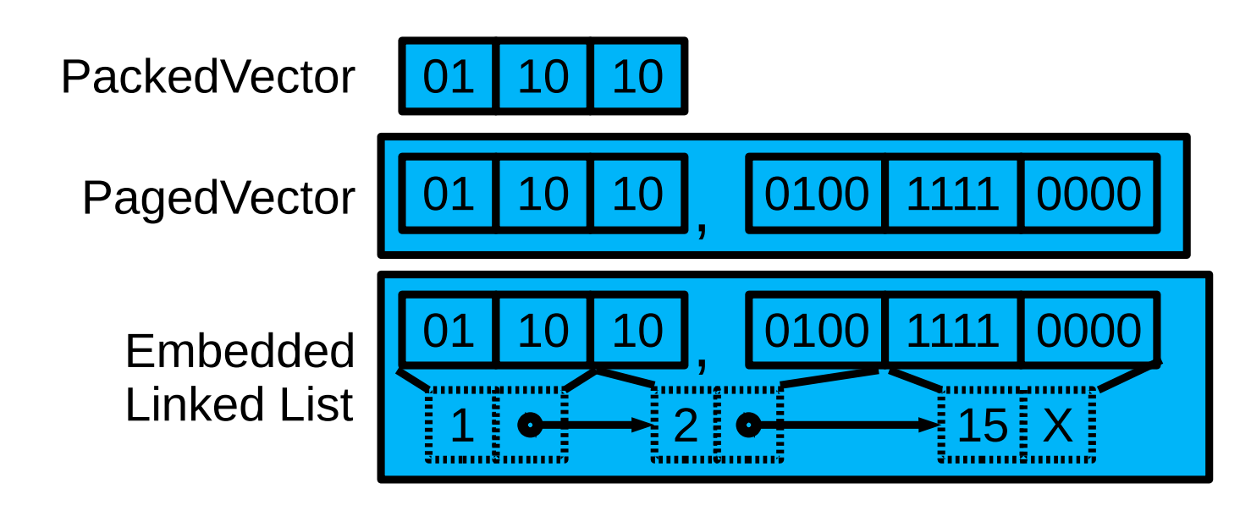
Node records and path linked lists are stored in fast hash tables. No ancillary indexes needed.

Optimized Dynamic Graph Index (ODGI)



Node objects contain sequence and adjacency data in a single byte array, and path visits in a dynamic succinct integer vector.

PackedGraph



Node records, adjacency linked lists, path linked lists, name strings, and sequence strings are all allocated within bit-packed PackedVectors and PagedVectors.

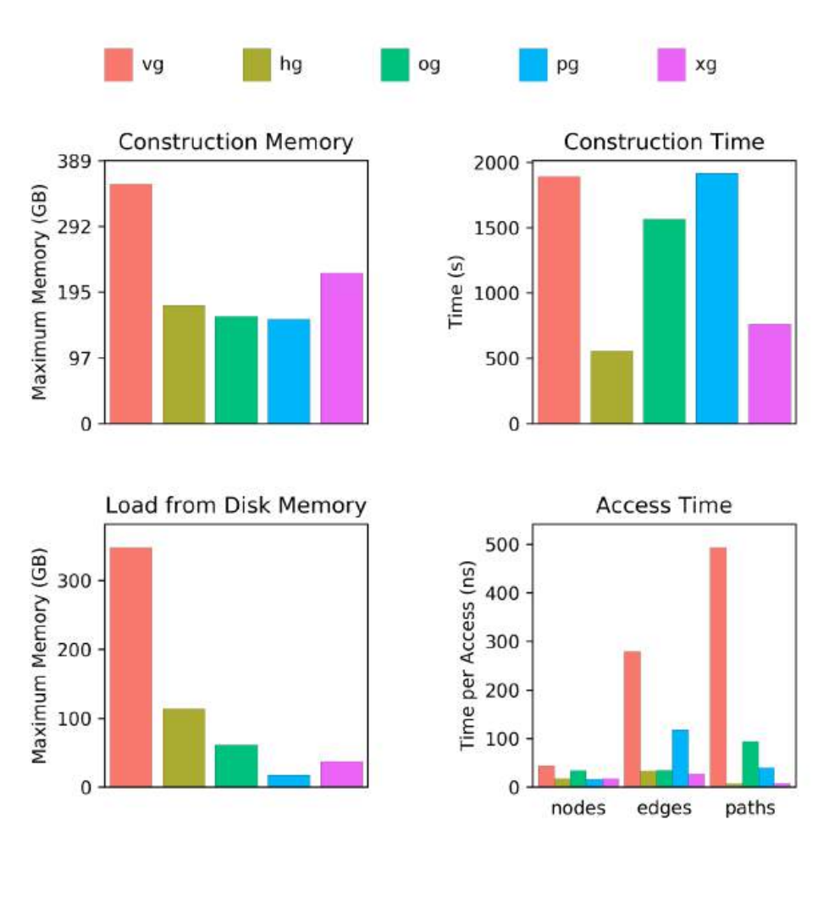
Model	HashGraph	ODGI	PackedGraph
Design goal	Simplicity, speed	Balanced speed/memory	Memory efficiency
Topology data structure	Hash table	Single integer vector	Several integer vectors
Topology compression	None	Delta encoding	Windowed bit compression
Sequence compression	None	None	Bit compression
Path list link encoding	Memory pointer	Delta-encoded rank	Vector index

Once the handle graph abstraction was created, vg’s algorithms could be written to work independently of the backing graph implementation. In order to improve vg’s performance, we developed three next-generation graph backends as part of the libbdsg project: HashGraph (a simple but well-designed hashtable-based implementation), ODGI (a node-centric representation), and PackedGraph (a bit-packed implementation designed to conserve memory). We implemented these reusable **B**i**D**irected **S**equence **G**raphs in the libbdsg library.

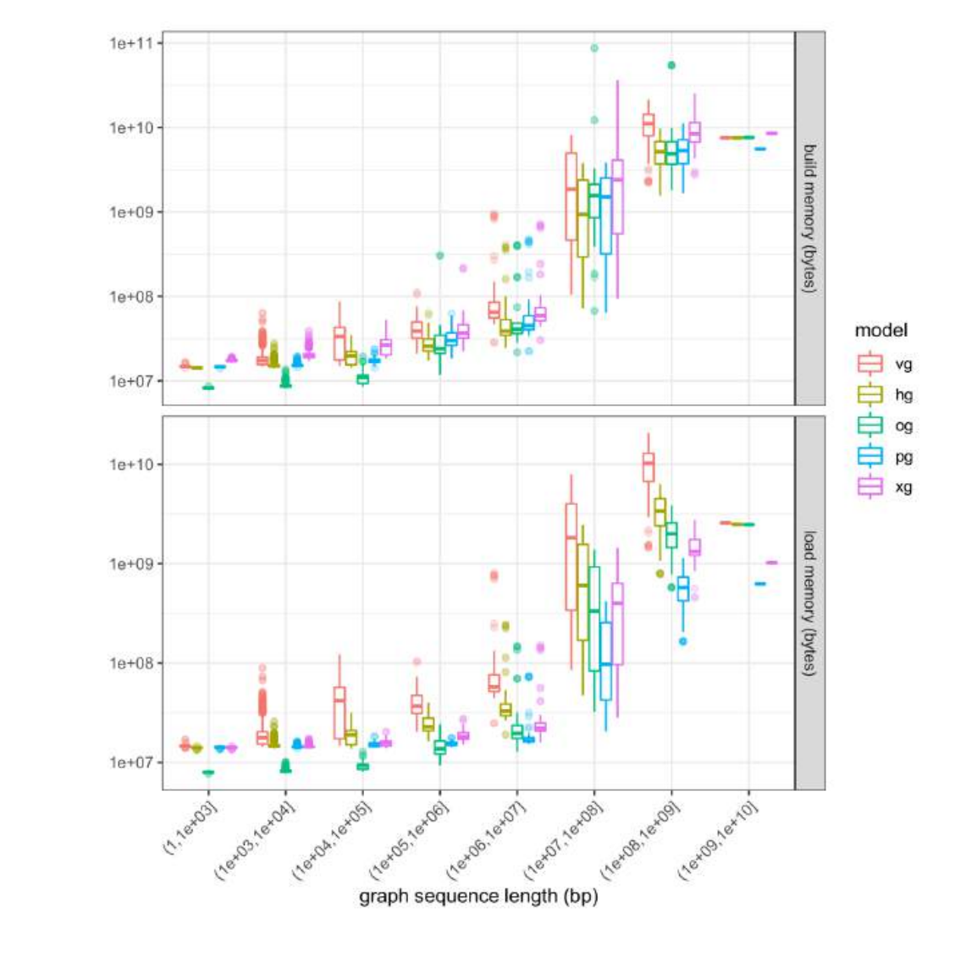
We compared the speed and memory usage when using these implementations to perform common graph operations on a collection of 2299 graphs that we encountered during actual pangenomics research. We used vg’s existing vg and xg implementations as comparison baselines.

Results

Improved Performance



HGSVC Graph Performance



Memory Usage on 2299 Graphs

The libbdsg implementations were found to be able to outperform the vg implementation by 10x or more on the Human Genome Structural Variant Consortium graph. Performance of the xg implementation was comparable to the libbdsg implementations, but xg is immutable while the libbdsg implementations are dynamic. Across the 2299-graph collection, memory usage was roughly linear in graph size, with constant overhead.

Documentation and Tutorials

https://bdsg.readthedocs.io

```
from bdsg.bdsg import ODGI
gr = ODGI()
seq = ["CGA", "TTGG", "CCGT"]
n = []
for s in seq:
    n.append(gr.create_handle(s))
```

```
#include <bdsg/odgi.hpp>
void main() {
    bdsg::ODGI gr;
    std::vector<std::string> seq {"CGA", "TTGG", "CCGT"};
    std::vector<handlegraph::handle_t> n;
    for (auto& s : seq) {
        n.push_back(gr.create_handle(s));
    }
}
```

The libbdsg library has a Python binding, with a tutorial designed to allow programmers to easily use their own data with libbdsg’s efficient graph implementations. Full API documentation for the Python and C++ interfaces is also available.

Future Work

Development of libbdsg and libhandlegraph are ongoing. Future work includes improved interoperability with the vg toolkit (including the ability to read vg-generated files without having to remove vg’s framing), a CMake-based project template for libbdsg C++ client applications, and a plain C API for interacting with handle graphs.

Acknowledgments

Glenn Hickey, Jouni Sirén, Jean Monlong, Jonas Sibbesen, Xian Chang, Charles Markello, Robin Rounthwaite, Yohei Rosen, Yatish Turakhia, Mike Lin, Eric Dawson, Michael Crusoe, David Haussler

NIH U01HG010961, U41HG010972, R01HG010485, 2U41HG007234, 5U54HG007990, 5T32HG008345-04, U01HL137183

W. M. KECK FOUNDATION DT06172015

Bundesministerium für Wirtschaft und Energie — Central Innovation Programme (ZIM) for SMEs of the Federal Ministry for Economic Affairs and Energy of Germany