bdsg Documentation

vgteam

Nov 02, 2020

Contents

CHAPTER 1

Setup

1.1 Build libbdsg

It is straightforward to build libbdsg on a Unix-based machine, such as Linux or macOS.

1.1.1 Clone the Repository

First, obtain a copy of the repository. Assuming you have Git installed already:

```
git clone --recursive https://github.com/vgteam/libbdsg.git
cd libbdsg
```

1.1.2 Install Dependencies

Some dependencies of libbdsg need to be installed before building the library. A working compiler with C++14 and OpenMP support, CMake 3.10 or newer, Doxygen, and development headers for `python3` are required. How to install these varies depending on your operating system. Please follow the section for your OS below:

macOS

On Mac, you will need to make sure you have OpenMP installed, as it is not part of the Mac system by default. To install it and other dependencies with Homebrew, you can do:

```
brew install libomp doxygen
```

Preinstalled Mac versions of Python already come with their development headers.

Ubuntu, Mint, and Other Debian Derivatives

```
sudo apt update
sudo apt install build-essential git cmake python3 python3-dev doxygen
```

Arch Linux

```
sudo pacman -Sy base-devel git cmake doxygen python3
```

Gentoo Linux

Gentoo already ships Python 3 as part of the base system, but the libbdsg build process goes most smoothly when the latest installed version of Python is selected as the default. Run the following as root:

```
emerge --sync
emerge dev-vcs/git dev-util/cmake app-doc/doxygen dev-lang/python
eselect python update --python3
```

1.1.3 Configure and Build

After installing dependencies, for all platforms, build through CMake. THe libbdsg build system expects an out-of-tree build, hence the creation of the `build` directory.

```
mkdir build
cd build
cmake ..
make
```

If you would like to run multiple build tasks in parallel, try `make -j4` or `make -j8` instead, for 4 or 8 prarllel tasks.

If you encounter error messages like `No download info given for 'sdsl-lite' and its source directory`, then you neglected to clone with `--recursive` and don't have the submodule dependencies downloaded. To fix this, you can run:

```
git submodule update --init --recursive
```

You might also encounter a message like:

This happens when you have installed a newer version of Python, but it is not set as the default `python3`. The easiest thing to do is to tell libbdsg to build against your current default `python3` instead of the newest installed one:

```
cmake -DPYTHON_EXECUTABLE=/usr/bin/python3 ..
make
```

1.1.4 Run Tests

To make sure that your built version of `libbdsg` works, yoiu can run the included tests.

If you were in `build`, make sure to run `cd ..` to go back to the root of the repository. Then run:

```
./bin/test_libbdsg
```

1.1.5 Build Documentation

To make a local copy of this documentation, first make sure you are in the root of the repository (not in `build`), and then run:

```
# Install Sphinx
virtualenv --python python3 venv
. venv/bin/activate
pip3 install -r docs/requirements.txt

# Build the documentation
make docs
```

Then open `docs/_build/html/index.html` in your web browser.

Note that for documentation updates in the source code to propagate to the HTML output, you first need to regenerate the Python bindings (to update the docstrings in the Python module source) and rerun the CMake-based build (to build the module, and to generate the C++ Docygen XML).

1.2 Use libbdsg From Python

To import the `bdsg` module in python, make sure that the compiled `lib/bdsg.cpython*.so` file is on your Python import path. There are three ways to do this:

1. Add `lib` to your `PYTHONPATH` environment variable.
2. Added `lib` your `sys.path` from within Python.
3. Just be in the `lib` directory.

Once the module is on your Python import path, you can run `import bdsg`.

For example, assuming that your current working directory is the root of the libbdsg project:

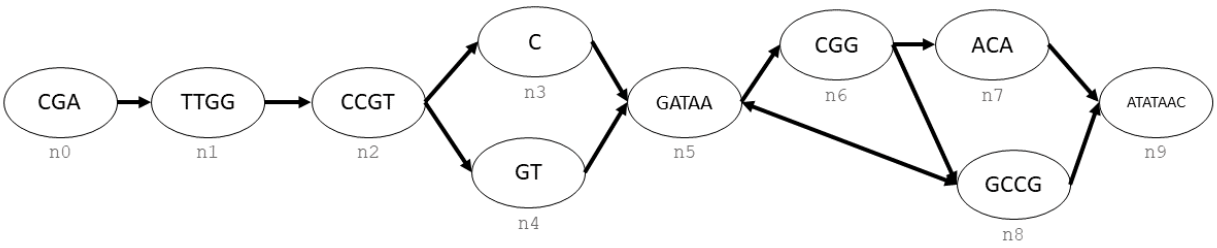
```
import sys
sys.path.append("./lib")
import bdsg
```

CHAPTER 2

Tutorial

2.1 Creating Graphs

Let's say that you wanted to create the following graph:



This graph is a combination of nodes (labelled as *n0*, *n1*, . . . , *n9*) and directed edges (arrows).

2.1.1 Graph Objects

Edges and *nodes* are accessed through an implementation of the `bdsg.handlegraph.HandleGraph` interface. Individual nodes in the graph are pointed at by `bdsg.handlegraph.handle_t` objects.

Paths exist in graphs that implement the `bdsg.handlegraph.PathHandleGraph`. Paths are accessed through `bdsg.handlegraph.path_handle_t`, which is a series of `bdsg.handlegraph.step_handle_t` linked together. Each `bdsg.handlegraph.step_handle_t` points to the node in that step, and also contains directional information regarding the nodes preceeding and following it.

Handles are pointers to specific pieces of the graph, and it is not possible to operate on them directly, aside from comparing whether the objects are equal. To get information regarding the object that each handle is pointing to, it is necessary to use the corresponding *get* accessor method on the graph that issued the handle.

Reference materials for these methods can be found at the *Python API*, as well as the *Sorted Glossary of Methods*, which contains lists sorted by object type for *Accessor Methods*, *Mutator Methods*, and *Iteratator Methods*.

2.1.2 Making a Graph

First, we must create the graph, then make each node and keep track of their handles. We're going to be using the **Optimized Dynamic Graph Implementation**, *bdsg.bdsg.ODGI*, which is a good all-around graph that implements *bdsg.handlegraph.MutablePathDeletableHandleGraph*.

```
from bdsg.bdsg import ODGI
gr = ODGI()
seq = ["CGA", "TTGG", "CCGT", "C", "GT", "GATAA", "CGG", "ACA", "GCCG", "ATATAAC"]
n = []
for s in seq:
    n.append(gr.create_handle(s))
```

Now we link together these nodes using their handles. Note that each of these handles is directional, and we create each edge from the first handle to the second. In order to create both of the edges between *n5* and *n8* (since each can follow the other) we use `create_edge` twice.

```
gr.create_edge(n[0], n[1])
gr.create_edge(n[1], n[2])
gr.create_edge(n[2], n[3])
gr.create_edge(n[2], n[4])
gr.create_edge(n[3], n[5])
gr.create_edge(n[5], n[6])
# Connect the end of n5 to the start of n8
gr.create_edge(n[5], n[8])
gr.create_edge(n[6], n[7])
gr.create_edge(n[6], n[8])
gr.create_edge(n[7], n[9])
gr.create_edge(n[8], n[9])
# Connect the end of n8 back around to the start of n5
gr.create_edge(n[8], n[5])
```

2.1.3 Traversing Edges

If we wanted to traverse these edges, we could do it using the iterator method *bdsg.handlegraph.HandleGraph.follow_edges()*.

```
def next_node_list(handle):
    lis = []
    gr.follow_edges(handle, False, lambda y: lis.append(y))
    return lis

print(f'n0: {gr.get_sequence(n[0])}')
next_node = next_node_list(n[0])[0]
print(f'n1: {gr.get_sequence(next_node)}')
next_node = next_node_list(next_node)[0]
print(f'n2: {gr.get_sequence(next_node)}')
```

Which will output the following:

```
n0: CGA
n1: TTGG
n2: CCGT
```

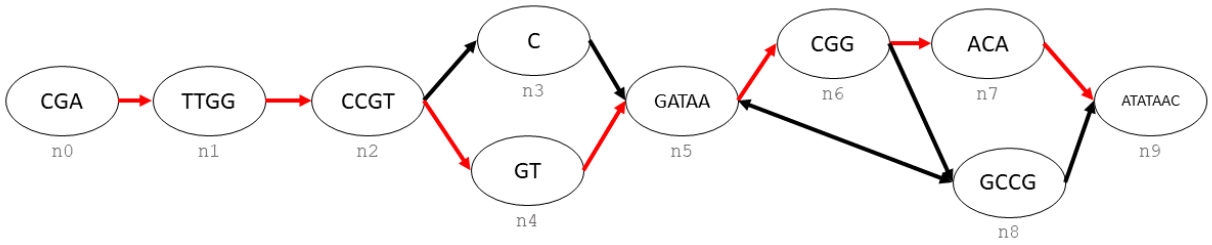
Since we are using *bdsg.bdsg.ODGI*, a text representation of the data can be generated using *bdsg.bdsg.ODGI.to_gfa()*. Use "-" as the destination filename to send the result to standard output, or provide no filename

to get a string returned.

```
print(gr.to_gfa())
```

2.1.4 Creating a Path

Generating a linear sequence from this graph could be done in infinitely many ways, due to the interal loop between *n5*, *n6*, and *n8*. If we wanted to define a single consensus sequence, we would do this by defining a path.



To create the hilighted path, we would need to create a *bdsg.handlegraph.path_handle_t* in the graph, and then append each *bdsg.handlegraph.handle_t* to the end of the path.

```
path = gr.create_path_handle("path")
gr.append_step(path, n[0])
gr.append_step(path, n[1])
gr.append_step(path, n[2])
gr.append_step(path, n[4])
gr.append_step(path, n[5])
gr.append_step(path, n[6])
gr.append_step(path, n[7])
gr.append_step(path, n[9])
```

Warning: *bdsg.handlegraph.MutablePathHandleGraph.append_step()* will not stop you from appending nodes that are not connected to the preceeding node.

```
# the following code runs without error
badpath = gr.create_path_handle("badpath")
gr.append_step(badpath, n[0])
gr.append_step(badpath, n[3])
```

2.1.5 Traversing a path

To traverse a path, we need to fetch a series of *bdsg.handlegraph.step_handle_t* from the graph. Note that although we are effectively asking the path for these items in it, all accessor methods are a part of the *bdsg.handlegraph.PathHandleGraph* object.

```
step = gr.path_begin(path)
while(gr.has_next_step(step)):
    # get the node handle from the step handle
    current_node_handle = gr.get_handle_of_step(step)
    # ask the node handle for the sequence
    print(gr.get_sequence(current_node_handle))
```

(continues on next page)

(continued from previous page)

```
    # progress to the next step
    step = gr.get_next_step(step)
current_node_handle = gr.get_handle_of_step(step)
print(gr.get_sequence(current_node_handle))
```

Which will output the following:

```
CGA
TTGG
CCGT
GT
GATAA
CGG
ACA
ATATAAC
```

2.2 Saving and Loading Graphs

Graphs that implement *bdsg.handlegraph.SerializableHandleGraph*, such as *bdsg.bdsg.ODGI*, can be saved and loaded through the *bdsg.handlegraph.SerializableHandleGraph.serialize()* and *bdsg.handlegraph.SerializableHandleGraph.deserialize()* methods.

2.2.1 Graph File Example

If you wish to save the graph from the above session, that can be done with:

```
gr.serialize("example_graph.odgi")
```

This can be loaded into a new python session by using:

```
from bdsg.bdsg import ODGI
gr = ODGI()
gr.deserialize("example_graph.odgi")
```

2.2.2 Loading in Pre-Existing Data

Each graph implementation knows how to read files in its respective file format.

For example, provided that data has been serialized in PackedGraph format, it is possible to read it directly from a file with *bdsg.bdsg.PackedGraph*. Download `this graph` and load it into python with:

```
from bdsg.bdsg import PackedGraph
brca2 = PackedGraph()
brca2.deserialize("cactus-brca2.pg")
```

We can poke around this data and get the sequence of the path with:

```
path_handle = []
handles = []
brca2.for_each_path_handle(lambda y: path_handle.append(y) or True)
brca2.for_each_step_in_path(path_handle[0],
```

(continues on next page)

(continued from previous page)

```
    lambda y: handles.append(brca2.get_handle_of_step(y)) or True)
sequence = ""
for handle in handles:
    sequence += brca2.get_sequence(handle)
print(sequence[0:10])
print(len(sequence))
```

Note how we are using `or True` in the iteratee callback lambda functions to make sure they return `True`. If a callback returns `False` or `None` (which is what is returned when you don't return anything), iteration will stop early and the `for_each` call will return `False`.

2.2.3 Reading in a Graph from vg

Graph assembies can be created with vg. Many `.vg` files that vg 1.28.0 or newer produces will be in HashGraph format, directly loadable by `bdsg.bdsg.HashGraph.deserialize()`. To check a file, you can use `vg stats --format`, like so:

```
vg stats --format graph.vg
```

If you see one of `format: HashGraph`, `format: PackedGraph`, or `format: ODGI`, you can probably (but not always; see the note on *encapsulation*) load the graph with *`bdsg.bdsg.HashGraph`*, *`bdsg.bdsg.PackedGraph`*, or *`bdsg.bdsg.ODGI`*, respectively.

However, in some circumstances, you will need to convert the graph to one of those formats. Some graphs (most notably, graphs from `vg construct`) will report `format: VG-Protobuf`, and `.xg` files will report `format: XG`. These graphs will need to be converted to HashGraph, PackedGraph, or ODGI format, and then loaded with the appropriate class. For example, you can do this:

```
vg convert --packed-out graph.vg > graph.pg
```

The resulting PackedGraph file can be loaded with `bdsg.bdsg.PackedGraph.deserialize()`.

```
from bdsg.bdsg import PackedGraph
graph = PackedGraph()
graph.deserialize("graph.pg")
```

To use *`bdsg.bdsg.HashGraph`* instead, substitute `--hash-out` for `--packed-out`. For *`bdsg.bdsg.ODGI`*, use `--odgi-out`.

2.2.4 Older vg Graphs with Encapsulation

Versions of vg before 1.28.0 would encapsulate HashGraph, PackedGraph, and ODGI graphs in a file format that vg can read but libbdsg cannot. Consequently, some older graph files will be reported as `format: HashGraph`, `format: PackedGraph`, or `format: ODGI` by `vg stats --format`, but will still not be readable using libbdsg.

If you encounter one of these files, you can use `vg view --extract-tag` to remove the encapsulation and pull out the internal file which libbdsg can understand. For example, for a file that reports `format: PackedGraph` but is not loadable by libbdsg, you can do:

```
vg view graph.vg --extract-tag PackedGraph > graph.pg
```

This also works for `HashGraph` and `ODGI` files, by replacing `PackedGraph` with `HashGraph` or `ODGI`.

Running the file through `vg convert` with vg 1.28.0 or newer will also solve the problem, but could take longer.

CHAPTER 3

Sorted Glossary of Methods

This page divides the Handle Graph API methods by category. It is written in terms of the Python methods, but applies equally well to the C++ interface.

3.1 Mutator Methods

The following lists breaks out methods from the various handle graph interfaces by what types of objects they modify.

3.1.1 Node Mutators

bdsg.handlegraph.MutableHandleGraph

bdsg.handlegraph.MutableHandleGraph.**create_handle**(**args*, ***kwargs*)

Overloaded function.

1. create_handle(self: bdsg.handlegraph.MutableHandleGraph, sequence: str) -> bdsg.handlegraph.handle_t

Create a new node with the given sequence and return the handle. The sequence may not be empty.

C++: handlegraph::MutableHandleGraph::create_handle(const std::string &) –> struct handlegraph::handle_t

2. create_handle(self: bdsg.handlegraph.MutableHandleGraph, sequence: str, id: int) -> bdsg.handlegraph.handle_t

Create a new node with the given id and sequence, then return the handle. The sequence may not be empty. The ID must be strictly greater than 0.

C++: handlegraph::MutableHandleGraph::create_handle(const std::string &, const long long &) –> struct handlegraph::handle_t

bdsg.handlegraph.MutableHandleGraph.**divide_handle**(**args*, ***kwargs*)

Overloaded function.

1. divide_handle(self: bdsg.handlegraph.MutableHandleGraph, handle: bdsg.handlegraph.handle_t, offsets: std::vector<unsigned long, std::allocator<unsigned long> >) -> std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >

Split a handle's underlying node at the given offsets in the handle's orientation. Returns all of the handles to the parts. Other handles to the node being split may be invalidated. The split pieces stay in the same local forward orientation as the original node, but the returned handles come in the order and orientation appropriate for the handle passed in. Updates stored paths.

C++: handlegraph::MutableHandleGraph::divide_handle(const struct handlegraph::handle_t &, const class std::vector<unsigned long> &) –> class std::vector<handlegraph::handle_t>

2. divide_handle(self: bdsg.handlegraph.MutableHandleGraph, handle: bdsg.handlegraph.handle_t, offset: int) -> Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

Specialization of divide_handle for a single division point

C++: handlegraph::MutableHandleGraph::divide_handle(const struct handlegraph::handle_t &, unsigned long) –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

bdsg.handlegraph.MutableHandleGraph.**apply_orientation**(*self: bdsg.handlegraph.MutableHandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Alter the node that the given handle corresponds to so the orientation indicated by the handle becomes the node's local forward orientation. Rewrites all edges pointing to the node and the node's sequence to reflect this. Invalidates all handles to the node (including the one passed). Returns a new, valid handle to the node in its new forward orientation. Note that it is possible for the node's ID to change. Does not update any stored paths. May change the ordering of the underlying graph.

C++: handlegraph::MutableHandleGraph::apply_orientation(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t

bdsg.handlegraph.DeletableHandleGraph

bdsg.handlegraph.DeletableHandleGraph.**destroy_handle**(*self: bdsg.handlegraph.DeletableHandleGraph*, *handle: bdsg.handlegraph.handle_t*) → None

Remove the node belonging to the given handle and all of its edges. Either destroys any paths in which the node participates, or leaves a "hidden", un-iterateable handle in the path to represent the sequence of the removed node. Invalidates the destroyed handle. May be called during serial for_each_handle iteration **ONLY** on the node being iterated. May **NOT** be called during parallel for_each_handle iteration. May **NOT** be called on the node from which edges are being followed during follow_edges. May **NOT** be called during iteration over paths, if it could destroy a path. May **NOT** be called during iteration along a path, if it could destroy that path.

C++: handlegraph::DeletableHandleGraph::destroy_handle(const struct handlegraph::handle_t &) –> void

3.1.2 Edge Mutators

bdsg.handlegraph.MutableHandleGraph

bdsg.handlegraph.MutableHandleGraph.**create_edge**(**args*, ***kwargs*)

Overloaded function.

1. create_edge(self: bdsg.handlegraph.MutableHandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

C++: handlegraph::MutableHandleGraph::create_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. create_edge(self: bdsg.handlegraph.MutableHandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for create_edge.

C++: handlegraph::MutableHandleGraph::create_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

bdsg.handlegraph.DeletableHandleGraph

bdsg.handlegraph.DeletableHandleGraph.**destroy_edge**(**args*, ***kwargs*)

Overloaded function.

1. destroy_edge(self: bdsg.handlegraph.DeletableHandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

C++: handlegraph::DeletableHandleGraph::destroy_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. destroy_edge(self: bdsg.handlegraph.DeletableHandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for destroy_edge.

C++: handlegraph::DeletableHandleGraph::destroy_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

3.1.3 Path Mutators

bdsg.handlegraph.MutablePathHandleGraph

bdsg.handlegraph.MutablePathHandleGraph.**create_path_handle**(**args*, ***kwargs*)

Overloaded function.

1. create_path_handle(self: bdsg.handlegraph.MutablePathHandleGraph, name: str) -> bdsg.handlegraph.path_handle_t

2. create_path_handle(self: bdsg.handlegraph.MutablePathHandleGraph, name: str, is_circular: bool) -> bdsg.handlegraph.path_handle_t

Create a path with the given name. The caller must ensure that no path with the given name exists already, or the behavior is undefined. Returns a handle to the created empty path. Handles to other paths must remain valid.

C++: handlegraph::MutablePathHandleGraph::create_path_handle(const std::string &, bool) –> struct handlegraph::path_handle_t

bdsg.handlegraph.MutablePathHandleGraph.**set_circularity**(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *circular: bool*) → None

Make a path circular or non-circular. If the path is becoming circular, the last step is joined to the first step. If the path is becoming linear, the step considered "last" is unjoined from the step considered "first" according to the method path_begin.

C++: handlegraph::MutablePathHandleGraph::set_circularity(const struct handlegraph::path_handle_t &, bool) –> void

bdsg.handlegraph.MutablePathHandleGraph.**destroy_path**(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*) → None

Destroy the given path. Invalidates handles to the path and its steps.

C++: handlegraph::MutablePathHandleGraph::destroy_path(const struct handlegraph::path_handle_t &) –> void

3.1.4 Path Step Mutators

bdsg.handlegraph.MutablePathHandleGraph

bdsg.handlegraph.MutablePathHandleGraph.**append_step**(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to prior steps on the path, and to other paths, must remain valid.

C++: handlegraph::MutablePathHandleGraph::append_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

bdsg.handlegraph.MutablePathHandleGraph.**prepend_step**(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_prepend: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Prepend a visit to a node to the given path. Returns a handle to the new first step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to later steps on the path, and to other paths, must remain valid.

C++: handlegraph::MutablePathHandleGraph::prepend_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

bdsg.handlegraph.MutablePathHandleGraph.**rewrite_segment**(*self: bdsg.handlegraph.MutablePathHandleGraph*, *segment_begin: bdsg.handlegraph.step_handle_t*, *segment_end: bdsg.handlegraph.step_handle_t*, *new_segment: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t>>*) → Tuple[bdsg.handlegraph.step_handle_t, bdsg.handlegraph.step_handle_t]

Delete a segment of a path and rewrite it as some other sequence of steps. Returns a pair of step_handle_t's that indicate the range of the new segment in the path. The segment to delete should be designated by the first (begin) and past-last (end) step handles. If the step that is returned by path_begin is deleted, path_begin will now return the first step from the new segment or, in the case that the new segment is empty, the step used as segment_end. Empty ranges consist of two copies of the same step handle. Empty ranges in empty paths consist of two copies of the end sentinel handle for the path. Rewriting an empty range inserts before the provided end handle.

C++: handlegraph::MutablePathHandleGraph::rewrite_segment(const struct handlegraph::step_handle_t &, const struct handlegraph::step_handle_t &, const class std::vector<handlegraph::handle_t> &) –> struct std::pair<struct handlegraph::step_handle_t, struct handlegraph::step_handle_t>

3.1.5 Graph Mutators

bdsg.handlegraph.MutableHandleGraph

bdsg.handlegraph.MutableHandleGraph.**optimize**(**args*, ***kwargs*)

Overloaded function.

1. optimize(self: bdsg.handlegraph.MutableHandleGraph) -> None
2. optimize(self: bdsg.handlegraph.MutableHandleGraph, allow_id_reassignment: bool) -> None

Adjust the representation of the graph in memory to improve performance. Optionally, allow the node IDs to be reassigned to further improve performance. Note: Ideally, this method is called one time once there is expected to be few graph modifications in the future.

C++: handlegraph::MutableHandleGraph::optimize(bool) –> void

bdsg.handlegraph.MutableHandleGraph.**apply_ordering**(**args*, ***kwargs*)

Overloaded function.

1. apply_ordering(self: bdsg.handlegraph.MutableHandleGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >) -> None
2. apply_ordering(self: bdsg.handlegraph.MutableHandleGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >, compact_ids: bool) -> None

Reorder the graph's internal structure to match that given. This sets the order that is used for iteration in functions like for_each_handle. Optionally may compact the id space of the graph to match the ordering, from 1->|ordering|. This may be a no-op in the case of graph implementations that do not have any mechanism to maintain an ordering.

C++: handlegraph::MutableHandleGraph::apply_ordering(const class std::vector<handlegraph::handle_t> &, bool) –> void

bdsg.handlegraph.MutableHandleGraph.**set_id_increment**(*self: bdsg.handlegraph.MutableHandleGraph*, *min_id: int*) → None

Set a minimum id to increment the id space by, used as a hint during construction. May have no effect on a backing implementation.

C++: handlegraph::MutableHandleGraph::set_id_increment(const long long &) –> void

bdsg.handlegraph.MutableHandleGraph.**increment_node_ids**(*self: bdsg.handlegraph.MutableHandleGraph*, *increment: int*) → None

Add the given value to all node IDs. Has a default implementation in terms of reassign_node_ids, but can be implemented more efficiently in some graphs.

C++: handlegraph::MutableHandleGraph::increment_node_ids(long long) –> void

bdsg.handlegraph.MutableHandleGraph.**reassign_node_ids**(*self: bdsg.handlegraph.MutableHandleGraph, get_new_id: Callable[[int], int]*) → None

Renumber all node IDs using the given function, which, given an old ID, returns the new ID. Modifies the graph in place. Invalidates all outstanding handles. If the graph supports paths, they also must be updated. The mapping function may return 0. In this case, the input ID will remain unchanged. The mapping function should not return any ID for which it would return 0.

C++: handlegraph::MutableHandleGraph::reassign_node_ids(const class std::function<long long (const long long &)> &) –> void

bdsg.handlegraph.DeletableHandleGraph

bdsg.handlegraph.DeletableHandleGraph.**clear**(*self: bdsg.handlegraph.DeletableHandleGraph*) → None

Remove all nodes and edges. May also remove all paths, if applicable.

C++: handlegraph::DeletableHandleGraph::clear() –> void

bdsg.handlegraph.SerializableHandleGraph

bdsg.handlegraph.SerializableHandleGraph.**deserialize**(*self: bdsg.handlegraph.SerializableHandleGraph*, *filename: str*) → None

Sets the contents of this graph to the contents of a serialized graph from a file. The serialized graph must be from the same implementation of the HandleGraph interface as is calling deserialize(). Can only be called on an empty graph.

C++: handlegraph::SerializableHandleGraph::deserialize(const std::string &) –> void

3.2 Accessor Methods

The following lists breaks out methods from the various handle graph interfaces by what types of objects they return information about.

3.2.1 Node Accessors

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**get_handle**(**args*, ***kwargs*)

Overloaded function.

1. get_handle(self: bdsg.handlegraph.HandleGraph, node_id: int) -> bdsg.handlegraph.handle_t
2. get_handle(self: bdsg.handlegraph.HandleGraph, node_id: int, is_reverse: bool) -> bdsg.handlegraph.handle_t

Look up the handle for the node with the given ID in the given orientation

C++: handlegraph::HandleGraph::get_handle(const long long &, bool) const –> struct handlegraph::handle_t

bdsg.handlegraph.HandleGraph.**get_id**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → int

Get the ID from a handle

C++: handlegraph::HandleGraph::get_id(const struct handlegraph::handle_t &) const –> long long

bdsg.handlegraph.HandleGraph.**get_is_reverse**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bool

Get the orientation of a handle

C++: handlegraph::HandleGraph::get_is_reverse(const struct handlegraph::handle_t &) const –> bool

bdsg.handlegraph.HandleGraph.**get_length**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → int

Get the length of a node

C++: handlegraph::HandleGraph::get_length(const struct handlegraph::handle_t &) const –> unsigned long

bdsg.handlegraph.HandleGraph.**get_sequence**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → str

Get the sequence of a node, presented in the handle's local forward orientation.

C++: handlegraph::HandleGraph::get_sequence(const struct handlegraph::handle_t &) const –> std::string

bdsg.handlegraph.HandleGraph.**get_subsequence**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*, *size: int*) → str

Returns a substring of a handle's sequence, in the orientation of the handle. If the indicated substring would extend beyond the end of the handle's sequence, the return value is truncated to the sequence's end. By default O(n) in the size of the handle's sequence, but can be overriden.

C++: handlegraph::HandleGraph::get_subsequence(const struct handlegraph::handle_t &, unsigned long, unsigned long) const –> std::string

bdsg.handlegraph.HandleGraph.**get_base**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*) → str

Returns one base of a handle's sequence, in the orientation of the handle.

C++: handlegraph::HandleGraph::get_base(const struct handlegraph::handle_t &, unsigned long) const –> char

bdsg.handlegraph.HandleGraph.**get_degree**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *go_left: bool*) → int

Get the number of edges on the right (go_left = false) or left (go_left = true) side of the given handle. The default implementation is O(n) in the number of edges returned, but graph implementations that track this information more efficiently can override this method.

C++: handlegraph::HandleGraph::get_degree(const struct handlegraph::handle_t &, bool) const –> unsigned long

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**get_step_count**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the number of node steps in the path

C++: handlegraph::PathHandleGraph::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

bdsg.handlegraph.PathHandleGraph.**steps_of_handle**(**args*, ***kwargs*)

Overloaded function.

1. steps_of_handle(self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t) -> std::vector<handlegraph::step_handle_t, std::allocator<handlegraph::step_handle_t> >

2. steps_of_handle(self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t, match_orientation: bool) -> std::vector<handlegraph::step_handle_t, std::allocator<handlegraph::step_handle_t> >

Returns a vector of all steps of a node on paths. Optionally restricts to steps that match the handle in orientation.

C++: handlegraph::PathHandleGraph::steps_of_handle(const struct handlegraph::handle_t &, bool) const –> class std::vector<handlegraph::step_handle_t>

bdsg.handlegraph.VectorizableHandleGraph

bdsg.handlegraph.VectorizableHandleGraph.**node_vector_offset**(*self: bdsg.handlegraph.VectorizableHandleGraph*, *node_id: int*) → int

Return the start position of the node in a (possibly implict) sorted array constructed from the concatenation of the node sequences

C++: handlegraph::VectorizableHandleGraph::node_vector_offset(const long long &) const –> unsigned long

bdsg.handlegraph.VectorizableHandleGraph.**node_at_vector_offset**(*self: bdsg.handlegraph.VectorizableHandleGr*
offset: int) → int

Return the node overlapping the given offset in the implicit node vector

C++: handlegraph::VectorizableHandleGraph::node_at_vector_offset(const unsigned long &) const –> long long

3.2.2 Node Handle Accessors

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**flip**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Invert the orientation of a handle (potentially without getting its ID)

C++: handlegraph::HandleGraph::flip(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

bdsg.handlegraph.HandleGraph.**forward**(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Get the locally forward version of a handle

C++: handlegraph::HandleGraph::forward(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

bdsg.handlegraph.ExpandingOverlayGraph

bdsg.handlegraph.ExpandingOverlayGraph.**get_underlying_handle**(*self: bdsg.handlegraph.ExpandingOverlayGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Returns the handle in the underlying graph that corresponds to a handle in the overlay

C++: handlegraph::ExpandingOverlayGraph::get_underlying_handle(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

3.2.3 Edge Accessors

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**edge_handle**(*self: bdsg.handlegraph.HandleGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

A pair of handles can be used as an edge. When so used, the handles have a canonical order and orientation.

C++: handlegraph::HandleGraph::edge_handle(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

bdsg.handlegraph.HandleGraph.**traverse_edge_handle**(*self: bdsg.handlegraph.HandleGraph*, *edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]*, *left: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Such a pair can be viewed from either inward end handle and produce the outward handle you would arrive at.

C++: handlegraph::HandleGraph::traverse_edge_handle(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &, const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

bdsg.handlegraph.VectorizableHandleGraph

bdsg.handlegraph.VectorizableHandleGraph.**edge_index**(*self: bdsg.handlegraph.VectorizableHandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]*) → int

Return a unique index among edges in the graph

C++: handlegraph::VectorizableHandleGraph::edge_index(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) const –> unsigned long

3.2.4 Path Accessors

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**get_path_handle**(*self: bdsg.handlegraph.PathHandleGraph, path_name: str*) → bdsg.handlegraph.path_handle_t

Look up the path handle for the given path name. The path with that name must exist.

C++: handlegraph::PathHandleGraph::get_path_handle(const std::string &) const –> struct handlegraph::path_handle_t

bdsg.handlegraph.PathHandleGraph.**get_path_name**(*self: bdsg.handlegraph.PathHandleGraph, path_handle: bdsg.handlegraph.path_handle_t*) → str

Look up the name of a path from a handle to it

C++: handlegraph::PathHandleGraph::get_path_name(const struct handlegraph::path_handle_t &) const –> std::string

bdsg.handlegraph.PathHandleGraph.**get_is_circular**(*self: bdsg.handlegraph.PathHandleGraph, path_handle: bdsg.handlegraph.path_handle_t*) → bool

Look up whether a path is circular

C++: handlegraph::PathHandleGraph::get_is_circular(const struct handlegraph::path_handle_t &) const –> bool

bdsg.handlegraph.PathHandleGraph.**get_step_count**(*self: bdsg.handlegraph.PathHandleGraph, path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the number of node steps in the path

C++: handlegraph::PathHandleGraph::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

bdsg.handlegraph.PathHandleGraph.**is_empty**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool

Returns true if the given path is empty, and false otherwise

C++: handlegraph::PathHandleGraph::is_empty(const struct handlegraph::path_handle_t &) const –> bool

bdsg.handlegraph.PathHandleGraph.**path_begin**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the first step, which will be an arbitrary step in a circular path that we consider "first" based on our construction of the path. If the path is empty, then the implementation must return the same value as path_end().

C++: handlegraph::PathHandleGraph::path_begin(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathHandleGraph.**path_end**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position past the end of a path. This position is returned by get_next_step for the final step in a path in a non-circular path. Note: get_next_step will *NEVER* return this value for a circular path.

C++: handlegraph::PathHandleGraph::path_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathHandleGraph.**path_back**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the last step, which will be an arbitrary step in a circular path that we consider "last" based on our construction of the path. If the path is empty then the implementation must return the same value as path_front_end().

C++: handlegraph::PathHandleGraph::path_back(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathHandleGraph.**path_front_end**(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position before the beginning of a path. This position is return by get_previous_step for the first step in a path in a non-circular path. Note: get_previous_step will *NEVER* return this value for a circular path.

C++: handlegraph::PathHandleGraph::path_front_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathPositionHandleGraph

bdsg.handlegraph.PathPositionHandleGraph.**get_path_length**(*self: bdsg.handlegraph.PathPositionHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the length of a path measured in bases of sequence.

C++: handlegraph::PathPositionHandleGraph::get_path_length(const struct handlegraph::path_handle_t &) const –> unsigned long

3.2.5 Path Step Accessors

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**get_path_handle_of_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.path_handle_t

Returns a handle to the path that an step is on

C++: handlegraph::PathHandleGraph::get_path_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::path_handle_t

bdsg.handlegraph.PathHandleGraph.**get_handle_of_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.handle_t

Get a node handle (node ID and orientation) from a handle to an step on a path

C++: handlegraph::PathHandleGraph::get_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::handle_t

bdsg.handlegraph.PathHandleGraph.**has_next_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool

Returns true if the step is not the last step in a non-circular path.

C++: handlegraph::PathHandleGraph::has_next_step(const struct handlegraph::step_handle_t &) const –> bool

bdsg.handlegraph.PathHandleGraph.**get_next_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the next step on the path. If the given step is the final step of a non-circular path, this method has undefined behavior. In a circular path, the "last" step will loop around to the "first" step.

C++: handlegraph::PathHandleGraph::get_next_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathHandleGraph.**has_previous_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool

Returns true if the step is not the first step in a non-circular path.

C++: handlegraph::PathHandleGraph::has_previous_step(const struct handlegraph::step_handle_t &) const –> bool

bdsg.handlegraph.PathHandleGraph.**get_previous_step**(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the previous step on the path. If the given step is the first step of a non-circular path, this method has undefined behavior. In a circular path, it will loop around from the "first" step (i.e. the one returned by path_begin) to the "last" step.

C++: handlegraph::PathHandleGraph::get_previous_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

bdsg.handlegraph.PathPositionHandleGraph

bdsg.handlegraph.PathPositionHandleGraph.**get_position_of_step**(*self: bdsg.handlegraph.PathPositionHandleGra step: bdsg.handlegraph.step_handle_t*) → int

Returns the position along the path of the beginning of this step measured in bases of sequence. In a circular path, positions start at the step returned by path_begin().

C++: handlegraph::PathPositionHandleGraph::get_position_of_step(const struct handlegraph::step_handle_t &) const –> unsigned long

bdsg.handlegraph.PathPositionHandleGraph.**get_step_at_position**(*self: bdsg.handlegraph.PathPositionHandleGra path: bdsg.handlegraph.path_handle_t, position: int*) → bdsg.handlegraph.step_handle_t

Returns the step at this position, measured in bases of sequence starting at the step returned by path_begin(). If the position is past the end of the path, returns path_end().

C++: handlegraph::PathPositionHandleGraph::get_step_at_position(const struct handlegraph::path_handle_t &, const unsigned long &) const –> struct handlegraph::step_handle_t

3.2.6 Graph Accessors

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**has_node**(*self: bdsg.handlegraph.HandleGraph*, *node_id: int*) → bool

Method to check if a node exists by ID

C++: handlegraph::HandleGraph::has_node(long long) const –> bool

bdsg.handlegraph.HandleGraph.**has_edge**(**args*, ***kwargs*)

Overloaded function.

1. has_edge(self: bdsg.handlegraph.HandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> bool

Returns true if there is an edge that allows traversal from the left handle to the right handle. By default O(n) in the number of edges on left, but can be overridden with more efficient implementations.

C++: handlegraph::HandleGraph::has_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> bool

2. has_edge(self: bdsg.handlegraph.HandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> bool

Convenient wrapper of has_edge for edge_t argument.

C++: handlegraph::HandleGraph::has_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) const –> bool

bdsg.handlegraph.HandleGraph.**get_node_count**(*self: bdsg.handlegraph.HandleGraph*) → int

Return the number of nodes in the graph

C++: handlegraph::HandleGraph::get_node_count() const –> unsigned long

bdsg.handlegraph.HandleGraph.**get_edge_count**(*self: bdsg.handlegraph.HandleGraph*) → int

Return the total number of edges in the graph. If not overridden, counts them all in linear time.

C++: handlegraph::HandleGraph::get_edge_count() const –> unsigned long

bdsg.handlegraph.HandleGraph.**get_total_length**(*self: bdsg.handlegraph.HandleGraph*) → int

Return the total length of all nodes in the graph, in bp. If not overridden, loops over all nodes in linear time.

C++: handlegraph::HandleGraph::get_total_length() const –> unsigned long

bdsg.handlegraph.HandleGraph.**min_node_id**(*self: bdsg.handlegraph.HandleGraph*) → int

Return the smallest ID in the graph, or some smaller number if the smallest ID is unavailable. Return value is unspecified if the graph is empty.

C++: handlegraph::HandleGraph::min_node_id() const –> long long

bdsg.handlegraph.HandleGraph.**max_node_id**(*self: bdsg.handlegraph.HandleGraph*) → int

Return the largest ID in the graph, or some larger number if the largest ID is unavailable. Return value is unspecified if the graph is empty.

C++: handlegraph::HandleGraph::max_node_id() const –> long long

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**get_path_count**(*self: bdsg.handlegraph.PathHandleGraph*) → int

Returns the number of paths stored in the graph

C++: handlegraph::PathHandleGraph::get_path_count() const –> unsigned long

bdsg.handlegraph.SerializableHandleGraph

bdsg.handlegraph.SerializableHandleGraph.**serialize**(*self: bdsg.handlegraph.SerializableHandleGraph*, *filename: str*) → None

Write the contents of this graph to a named file. Makes sure to include a leading magic number.

C++: handlegraph::SerializableHandleGraph::serialize(const std::string &) const –> void

bdsg.handlegraph.SerializableHandleGraph.**get_magic_number**(*self: bdsg.handlegraph.SerializableHandleGraph*) → int

Returns a number that is specific to the serialized implementation for type checking. Does not depend on the contents of any particular instantiation (i.e. behaves as if static, but cannot be static and virtual).

C++: handlegraph::SerializableHandleGraph::get_magic_number() const –> unsigned int

3.3 Iteratator Methods

The following lists breaks out methods from the various handle graph interfaces by what types of objects they iterate over. Note that iteration is **callback-based** and not via traditional Python iterator semantics. Iteratee functions should return `False` to stop iteration, and must return `True` to continue. Not returning anything (i.e. returning `None`) will stop iteration early.

3.3.1 Node Iterators

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**for_each_handle**(**args*, ***kwargs*)

Overloaded function.

1. for_each_handle(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]) -> bool

2. for_each_handle(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool], parallel: bool) -> bool

C++: handlegraph::HandleGraph::for_each_handle(const class std::function<bool (const struct handlegraph::handle_t &)> &, bool) const –> bool

3.3.2 Edge Iterators

bdsg.handlegraph.HandleGraph

bdsg.handlegraph.HandleGraph.**follow_edges**(*self: bdsg.handlegraph.HandleGraph, handle: bdsg.handlegraph.handle_t, go_left: bool, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]*) → bool

C++: handlegraph::HandleGraph::follow_edges(const struct handlegraph::handle_t &, bool, const class std::function<bool (const struct handlegraph::handle_t &)> &) const –> bool

bdsg.handlegraph.HandleGraph.**for_each_edge**(**args*, ***kwargs*)

Overloaded function.

1. for_each_edge(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]], bool]) -> bool

2. for_each_edge(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]], bool], parallel: bool) -> bool

C++: handlegraph::HandleGraph::for_each_edge(const class std::function<bool (const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &)> &, bool) const –> bool

3.3.3 Path Iterators

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**for_each_path_handle**(*self: bdsg.handlegraph.PathHandleGraph, iteratee: Callable[[bdsg.handlegraph.path_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_path_handle(const class std::function<bool (const struct handlegraph::path_handle_t &)> &) const –> bool

3.3.4 Path Step Iterators

bdsg.handlegraph.PathHandleGraph

bdsg.handlegraph.PathHandleGraph.**for_each_step_in_path**(*self: bdsg.handlegraph.PathHandleGraph, path: bdsg.handlegraph.path_handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_step_in_path(const struct handlegraph::path_handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

bdsg.handlegraph.PathHandleGraph.**for_each_step_on_handle**(*self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_step_on_handle(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

bdsg.handlegraph.PathPositionHandleGraph

bdsg.handlegraph.PathPositionHandleGraph.**for_each_step_position_on_handle**(*self: bdsg.handlegraph.PathP han- dle: bdsg.handlegraph.handl it- er- a- tee: Callable[[bdsg.handlegr bool, int], bool]*) → bool

Execute an iteratee on each step on a path, along with its orientation relative to the path (true if it is reverse the orientation of the handle on the path), and its position measured in bases of sequence along the path. Positions are always measured on the forward strand.

Iteration will stop early if the iteratee returns false. This method returns false if iteration was stopped early, else true

C++: handlegraph::PathPositionHandleGraph::for_each_step_position_on_handle(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &, const bool &, const unsigned long &)> &) const –> bool

CHAPTER 4

Python API

4.1 Handle Graph API

The *bdsg.handlegraph* module defines the handle graph interface.

Bindings for ::handlegraph namespace

4.1.1 Handles

The module contains definitions for different types of handles. THese are references to graph elements. A basic bdsg.handelgraph.handle_t is a reference to a strand or orientation of a node in the graph.

class bdsg.handlegraph.**handle_t**
Bases: pybind11_builtins.pybind11_object

Represents the internal id of a node traversal

assign (*self: bdsg.handlegraph.handle_t*, *: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t
C++: handlegraph::handle_t::operator=(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t &

class bdsg.handlegraph.**path_handle_t**
Bases: pybind11_builtins.pybind11_object

Represents the internal id of a path entity

assign (*self: bdsg.handlegraph.path_handle_t*, *: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.path_handle_t
C++: handlegraph::path_handle_t::operator=(const struct handlegraph::path_handle_t &) –> struct handlegraph::path_handle_t &

class bdsg.handlegraph.**step_handle_t**
Bases: pybind11_builtins.pybind11_object

A step handle is an opaque reference to a single step of an oriented node on a path in a graph

assign(*self: bdsg.handlegraph.step_handle_t*, *: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

C++: handlegraph::step_handle_t::operator=(const struct handlegraph::step_handle_t &) –> struct handlegraph::step_handle_t &

4.1.2 Graph Interfaces

The *bdsg.handlegraph* module also defines a hierarchy of interfaces for graph implementations that provide different levels of features.

HandleGraph

The most basic is the *bdsg.handlegraph.HandleGraph*, a completely immutable, unannotated graph.

class bdsg.handlegraph.**HandleGraph**

Bases: pybind11_builtins.pybind11_object

This is the interface that a graph that uses handles needs to support. It is also the interface that users should code against.

assign(*self: bdsg.handlegraph.HandleGraph*, *: bdsg.handlegraph.HandleGraph*) → bdsg.handlegraph.HandleGraph

C++: handlegraph::HandleGraph::operator=(const class handlegraph::HandleGraph &) –> class handlegraph::HandleGraph &

edge_handle(*self: bdsg.handlegraph.HandleGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

A pair of handles can be used as an edge. When so used, the handles have a canonical order and orientation.

C++: handlegraph::HandleGraph::edge_handle(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

flip(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Invert the orientation of a handle (potentially without getting its ID)

C++: handlegraph::HandleGraph::flip(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

follow_edges(*self: bdsg.handlegraph.HandleGraph, handle: bdsg.handlegraph.handle_t, go_left: bool, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]*) → bool

C++: handlegraph::HandleGraph::follow_edges(const struct handlegraph::handle_t &, bool, const class std::function<bool (const struct handlegraph::handle_t &)> &) const –> bool

for_each_edge(**args*, ***kwargs*)

Overloaded function.

1. for_each_edge(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]], bool]) -> bool

2. for_each_edge(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]], bool], parallel: bool) -> bool

C++: handlegraph::HandleGraph::for_each_edge(const class std::function<bool (const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &)> &, bool) const –> bool

for_each_handle(**args*, ***kwargs*)
Overloaded function.

1. for_each_handle(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]) -> bool

2. for_each_handle(self: bdsg.handlegraph.HandleGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool], parallel: bool) -> bool

C++: handlegraph::HandleGraph::for_each_handle(const class std::function<bool (const struct handlegraph::handle_t &)> &, bool) const –> bool

forward(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t
Get the locally forward version of a handle

C++: handlegraph::HandleGraph::forward(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

get_base(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*) → str

Returns one base of a handle's sequence, in the orientation of the handle.

C++: handlegraph::HandleGraph::get_base(const struct handlegraph::handle_t &, unsigned long) const –> char

get_degree(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *go_left: bool*) → int

Get the number of edges on the right (go_left = false) or left (go_left = true) side of the given handle. The default implementation is O(n) in the number of edges returned, but graph implementations that track this information more efficiently can override this method.

C++: handlegraph::HandleGraph::get_degree(const struct handlegraph::handle_t &, bool) const –> unsigned long

get_edge_count(*self: bdsg.handlegraph.HandleGraph*) → int

Return the total number of edges in the graph. If not overridden, counts them all in linear time.

C++: handlegraph::HandleGraph::get_edge_count() const –> unsigned long

get_handle(**args*, ***kwargs*)
Overloaded function.

1. get_handle(self: bdsg.handlegraph.HandleGraph, node_id: int) -> bdsg.handlegraph.handle_t

2. get_handle(self: bdsg.handlegraph.HandleGraph, node_id: int, is_reverse: bool) -> bdsg.handlegraph.handle_t

Look up the handle for the node with the given ID in the given orientation

C++: handlegraph::HandleGraph::get_handle(const long long &, bool) const –> struct handlegraph::handle_t

get_id(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → int
Get the ID from a handle

C++: handlegraph::HandleGraph::get_id(const struct handlegraph::handle_t &) const –> long long

get_is_reverse(*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bool
Get the orientation of a handle

C++: handlegraph::HandleGraph::get_is_reverse(const struct handlegraph::handle_t &) const –> bool

get_length (*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → int
Get the length of a node

C++: handlegraph::HandleGraph::get_length(const struct handlegraph::handle_t &) const –> unsigned long

get_node_count (*self: bdsg.handlegraph.HandleGraph*) → int
Return the number of nodes in the graph

C++: handlegraph::HandleGraph::get_node_count() const –> unsigned long

get_sequence (*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*) → str

Get the sequence of a node, presented in the handle's local forward orientation.

C++: handlegraph::HandleGraph::get_sequence(const struct handlegraph::handle_t &) const –> std::string

get_subsequence (*self: bdsg.handlegraph.HandleGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*, *size: int*) → str

Returns a substring of a handle's sequence, in the orientation of the handle. If the indicated substring would extend beyond the end of the handle's sequence, the return value is truncated to the sequence's end. By default O(n) in the size of the handle's sequence, but can be overriden.

C++: handlegraph::HandleGraph::get_subsequence(const struct handlegraph::handle_t &, unsigned long, unsigned long) const –> std::string

get_total_length (*self: bdsg.handlegraph.HandleGraph*) → int

Return the total length of all nodes in the graph, in bp. If not overridden, loops over all nodes in linear time.

C++: handlegraph::HandleGraph::get_total_length() const –> unsigned long

has_edge (**args*, ***kwargs*)
Overloaded function.

1. has_edge(self: bdsg.handlegraph.HandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> bool

Returns true if there is an edge that allows traversal from the left handle to the right handle. By default O(n) in the number of edges on left, but can be overridden with more efficient implementations.

C++: handlegraph::HandleGraph::has_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> bool

2. has_edge(self: bdsg.handlegraph.HandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> bool

Convenient wrapper of has_edge for edge_t argument.

C++: handlegraph::HandleGraph::has_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) const –> bool

has_node (*self: bdsg.handlegraph.HandleGraph*, *node_id: int*) → bool
Method to check if a node exists by ID

C++: handlegraph::HandleGraph::has_node(long long) const –> bool

max_node_id (*self: bdsg.handlegraph.HandleGraph*) → int

Return the largest ID in the graph, or some larger number if the largest ID is unavailable. Return value is unspecified if the graph is empty.

C++: handlegraph::HandleGraph::max_node_id() const –> long long

min_node_id(*self: bdsg.handlegraph.HandleGraph*) → int

Return the smallest ID in the graph, or some smaller number if the smallest ID is unavailable. Return value is unspecified if the graph is empty.

C++: handlegraph::HandleGraph::min_node_id() const –> long long

traverse_edge_handle(*self: bdsg.handlegraph.HandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t], left: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Such a pair can be viewed from either inward end handle and produce the outward handle you would arrive at.

C++: handlegraph::HandleGraph::traverse_edge_handle(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &, const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

PathHandleGraph

On top of this, there is the *bdsg.handlegraph.PathHandleGraph*, which allows for embedded, named paths in the graph.

class bdsg.handlegraph.**PathHandleGraph**

Bases: *bdsg.handlegraph.HandleGraph*

This is the interface for a handle graph that stores embedded paths.

assign(*self: bdsg.handlegraph.PathHandleGraph, : bdsg.handlegraph.PathHandleGraph*) → bdsg.handlegraph.PathHandleGraph

C++: handlegraph::PathHandleGraph::operator=(const class handlegraph::PathHandleGraph &) –> class handlegraph::PathHandleGraph &

for_each_path_handle(*self: bdsg.handlegraph.PathHandleGraph, iteratee: Callable[[bdsg.handlegraph.path_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_path_handle(const class std::function<bool (const struct handlegraph::path_handle_t &)> &) const –> bool

for_each_step_in_path(*self: bdsg.handlegraph.PathHandleGraph, path: bdsg.handlegraph.path_handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_step_in_path(const struct handlegraph::path_handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

for_each_step_on_handle(*self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool

C++: handlegraph::PathHandleGraph::for_each_step_on_handle(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

get_handle_of_step(*self: bdsg.handlegraph.PathHandleGraph, step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.handle_t

Get a node handle (node ID and orientation) from a handle to an step on a path

C++: handlegraph::PathHandleGraph::get_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::handle_t

get_is_circular(*self: bdsg.handlegraph.PathHandleGraph, path_handle: bdsg.handlegraph.path_handle_t*) → bool

Look up whether a path is circular

C++: handlegraph::PathHandleGraph::get_is_circular(const struct handlegraph::path_handle_t &) const –> bool

get_next_step(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the next step on the path. If the given step is the final step of a non-circular path, this method has undefined behavior. In a circular path, the "last" step will loop around to the "first" step.

C++: handlegraph::PathHandleGraph::get_next_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_path_count(*self: bdsg.handlegraph.PathHandleGraph*) → int

Returns the number of paths stored in the graph

C++: handlegraph::PathHandleGraph::get_path_count() const –> unsigned long

get_path_handle(*self: bdsg.handlegraph.PathHandleGraph*, *path_name: str*) → bdsg.handlegraph.path_handle_t

Look up the path handle for the given path name. The path with that name must exist.

C++: handlegraph::PathHandleGraph::get_path_handle(const std::string &) const –> struct handlegraph::path_handle_t

get_path_handle_of_step(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.path_handle_t

Returns a handle to the path that an step is on

C++: handlegraph::PathHandleGraph::get_path_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::path_handle_t

get_path_name(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → str

Look up the name of a path from a handle to it

C++: handlegraph::PathHandleGraph::get_path_name(const struct handlegraph::path_handle_t &) const –> std::string

get_previous_step(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the previous step on the path. If the given step is the first step of a non-circular path, this method has undefined behavior. In a circular path, it will loop around from the "first" step (i.e. the one returned by path_begin) to the "last" step.

C++: handlegraph::PathHandleGraph::get_previous_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_step_count(*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the number of node steps in the path

C++: handlegraph::PathHandleGraph::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

has_next_step(*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool

Returns true if the step is not the last step in a non-circular path.

C++: handlegraph::PathHandleGraph::has_next_step(const struct handlegraph::step_handle_t &) const –> bool

has_path (*self: bdsg.handlegraph.PathHandleGraph*, *path_name: str*) → bool
Determine if a path name exists and is legal to get a path handle for.

C++: handlegraph::PathHandleGraph::has_path(const std::string &) const –> bool

has_previous_step (*self: bdsg.handlegraph.PathHandleGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step is not the first step in a non-circular path.

C++: handlegraph::PathHandleGraph::has_previous_step(const struct handlegraph::step_handle_t &) const –> bool

is_empty (*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool
Returns true if the given path is empty, and false otherwise

C++: handlegraph::PathHandleGraph::is_empty(const struct handlegraph::path_handle_t &) const –> bool

path_back (*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the last step, which will be an arbitrary step in a circular path that we consider "last" based on our construction of the path. If the path is empty then the implementation must return the same value as path_front_end().

C++: handlegraph::PathHandleGraph::path_back(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_begin (*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the first step, which will be an arbitrary step in a circular path that we consider "first" based on our construction of the path. If the path is empty, then the implementation must return the same value as path_end().

C++: handlegraph::PathHandleGraph::path_begin(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_end (*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position past the end of a path. This position is returned by get_next_step for the final step in a path in a non-circular path. Note: get_next_step will *NEVER* return this value for a circular path.

C++: handlegraph::PathHandleGraph::path_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_front_end (*self: bdsg.handlegraph.PathHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position before the beginning of a path. This position is return by get_previous_step for the first step in a path in a non-circular path. Note: get_previous_step will *NEVER* return this value for a circular path.

C++: handlegraph::PathHandleGraph::path_front_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

scan_path (*self: bdsg.handlegraph.PathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*) → handlegraph::PathForEachSocket

Returns a class with an STL-style iterator interface that can be used directly in a for each loop like: for (handle_t handle : graph->scan_path(path)) { }

C++: handlegraph::PathHandleGraph::scan_path(const struct handlegraph::path_handle_t &) const –> class handlegraph::PathForEachSocket

steps_of_handle(**args*, ***kwargs*)
Overloaded function.

1. steps_of_handle(self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t) -> std::vector<handlegraph::step_handle_t, std::allocator<handlegraph::step_handle_t> >

2. steps_of_handle(self: bdsg.handlegraph.PathHandleGraph, handle: bdsg.handlegraph.handle_t, match_orientation: bool) -> std::vector<handlegraph::step_handle_t, std::allocator<handlegraph::step_handle_t> >

Returns a vector of all steps of a node on paths. Optionally restricts to steps that match the handle in orientation.

C++: handlegraph::PathHandleGraph::steps_of_handle(const struct handlegraph::handle_t &, bool) const –> class std::vector<handlegraph::step_handle_t>

Mutable and Deletable Interfaces

Then for each there are versions where the underlying graph is "mutable" (meaning that material can be added to it and nodes can be split) and "deletable" (meaning that nodes and edges can actually be removed from the graph), and for *bdsg.handlegraph.PathHandleGraph* there are versions where the paths can be altered.

class bdsg.handlegraph.**MutableHandleGraph**
Bases: *bdsg.handlegraph.HandleGraph*

This is the interface for a handle graph that supports addition of new graph material.

apply_ordering(**args*, ***kwargs*)
Overloaded function.

1. apply_ordering(self: bdsg.handlegraph.MutableHandleGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >) -> None

2. apply_ordering(self: bdsg.handlegraph.MutableHandleGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >, compact_ids: bool) -> None

Reorder the graph's internal structure to match that given. This sets the order that is used for iteration in functions like for_each_handle. Optionally may compact the id space of the graph to match the ordering, from 1->|ordering|. This may be a no-op in the case of graph implementations that do not have any mechanism to maintain an ordering.

C++: handlegraph::MutableHandleGraph::apply_ordering(const class std::vector<handlegraph::handle_t> &, bool) –> void

apply_orientation(*self: bdsg.handlegraph.MutableHandleGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Alter the node that the given handle corresponds to so the orientation indicated by the handle becomes the node's local forward orientation. Rewrites all edges pointing to the node and the node's sequence to reflect this. Invalidates all handles to the node (including the one passed). Returns a new, valid handle to the node in its new forward orientation. Note that it is possible for the node's ID to change. Does not update any stored paths. May change the ordering of the underlying graph.

C++: handlegraph::MutableHandleGraph::apply_orientation(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t

assign(*self: bdsg.handlegraph.MutableHandleGraph*, *: bdsg.handlegraph.MutableHandleGraph*) → bdsg.handlegraph.MutableHandleGraph

C++: handlegraph::MutableHandleGraph::operator=(const class handlegraph::MutableHandleGraph &) –> class handlegraph::MutableHandleGraph &

create_edge(**args*, ***kwargs*)

Overloaded function.

1. create_edge(self: bdsg.handlegraph.MutableHandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

C++: handlegraph::MutableHandleGraph::create_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. create_edge(self: bdsg.handlegraph.MutableHandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for create_edge.

C++: handlegraph::MutableHandleGraph::create_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

create_handle(**args*, ***kwargs*)

Overloaded function.

1. create_handle(self: bdsg.handlegraph.MutableHandleGraph, sequence: str) -> bdsg.handlegraph.handle_t

Create a new node with the given sequence and return the handle. The sequence may not be empty.

C++: handlegraph::MutableHandleGraph::create_handle(const std::string &) –> struct handlegraph::handle_t

2. create_handle(self: bdsg.handlegraph.MutableHandleGraph, sequence: str, id: int) -> bdsg.handlegraph.handle_t

Create a new node with the given id and sequence, then return the handle. The sequence may not be empty. The ID must be strictly greater than 0.

C++: handlegraph::MutableHandleGraph::create_handle(const std::string &, const long long &) –> struct handlegraph::handle_t

divide_handle(**args*, ***kwargs*)

Overloaded function.

1. divide_handle(self: bdsg.handlegraph.MutableHandleGraph, handle: bdsg.handlegraph.handle_t, offsets: std::vector<unsigned long, std::allocator<unsigned long> >) -> std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >

Split a handle's underlying node at the given offsets in the handle's orientation. Returns all of the handles to the parts. Other handles to the node being split may be invalidated. The split pieces stay in the same local forward orientation as the original node, but the returned handles come in the order and orientation appropriate for the handle passed in. Updates stored paths.

C++: handlegraph::MutableHandleGraph::divide_handle(const struct handlegraph::handle_t &, const class std::vector<unsigned long> &) –> class std::vector<handlegraph::handle_t>

2. divide_handle(self: bdsg.handlegraph.MutableHandleGraph, handle: bdsg.handlegraph.handle_t, offset: int) -> Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

Specialization of divide_handle for a single division point

C++: handlegraph::MutableHandleGraph::divide_handle(const struct handlegraph::handle_t &, unsigned long) –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

increment_node_ids(*self: bdsg.handlegraph.MutableHandleGraph*, *increment: int*) → None

Add the given value to all node IDs. Has a default implementation in terms of reassign_node_ids, but can be implemented more efficiently in some graphs.

C++: handlegraph::MutableHandleGraph::increment_node_ids(long long) –> void

optimize(**args*, ***kwargs*)

Overloaded function.

1. optimize(self: bdsg.handlegraph.MutableHandleGraph) -> None

2. optimize(self: bdsg.handlegraph.MutableHandleGraph, allow_id_reassignment: bool) -> None

Adjust the representation of the graph in memory to improve performance. Optionally, allow the node IDs to be reassigned to further improve performance. Note: Ideally, this method is called one time once there is expected to be few graph modifications in the future.

C++: handlegraph::MutableHandleGraph::optimize(bool) –> void

reassign_node_ids(*self: bdsg.handlegraph.MutableHandleGraph*, *get_new_id: Callable[[int], int]*) → None

Renumber all node IDs using the given function, which, given an old ID, returns the new ID. Modifies the graph in place. Invalidates all outstanding handles. If the graph supports paths, they also must be updated. The mapping function may return 0. In this case, the input ID will remain unchanged. The mapping function should not return any ID for which it would return 0.

C++: handlegraph::MutableHandleGraph::reassign_node_ids(const class std::function<long long (const long long &)> &) –> void

set_id_increment(*self: bdsg.handlegraph.MutableHandleGraph*, *min_id: int*) → None

Set a minimum id to increment the id space by, used as a hint during construction. May have no effect on a backing implementation.

C++: handlegraph::MutableHandleGraph::set_id_increment(const long long &) –> void

class bdsg.handlegraph.**DeletableHandleGraph**

Bases: *bdsg.handlegraph.MutableHandleGraph*

This is the interface for a handle graph that supports both addition of new nodes and edges as well as deletion of nodes and edges.

assign(*self: bdsg.handlegraph.DeletableHandleGraph*, *: bdsg.handlegraph.DeletableHandleGraph*) → bdsg.handlegraph.DeletableHandleGraph

C++: handlegraph::DeletableHandleGraph::operator=(const class handlegraph::DeletableHandleGraph &) –> class handlegraph::DeletableHandleGraph &

clear(*self: bdsg.handlegraph.DeletableHandleGraph*) → None

Remove all nodes and edges. May also remove all paths, if applicable.

C++: handlegraph::DeletableHandleGraph::clear() –> void

destroy_edge(**args*, ***kwargs*)

Overloaded function.

1. destroy_edge(self: bdsg.handlegraph.DeletableHandleGraph, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

C++: handlegraph::DeletableHandleGraph::destroy_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. destroy_edge(self: bdsg.handlegraph.DeletableHandleGraph, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for destroy_edge.

C++: handlegraph::DeletableHandleGraph::destroy_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

destroy_handle(*self: bdsg.handlegraph.DeletableHandleGraph*, *handle: bdsg.handlegraph.handle_t*) → None

Remove the node belonging to the given handle and all of its edges. Either destroys any paths in which the node participates, or leaves a "hidden", un-iterateable handle in the path to represent the sequence of the removed node. Invalidates the destroyed handle. May be called during serial for_each_handle iteration **ONLY** on the node being iterated. May **NOT** be called during parallel for_each_handle iteration. May **NOT** be called on the node from which edges are being followed during follow_edges. May **NOT** be called during iteration over paths, if it could destroy a path. May **NOT** be called during iteration along a path, if it could destroy that path.

C++: handlegraph::DeletableHandleGraph::destroy_handle(const struct handlegraph::handle_t &) –> void

class bdsg.handlegraph.**MutablePathHandleGraph**

Bases: *bdsg.handlegraph.PathHandleGraph*

This is the interface for a handle graph with embedded paths where the paths can be modified. Note that if the *graph* can also be modified, the implementation will also need to inherit from MutableHandleGraph, via the combination MutablePathMutableHandleGraph interface. TODO: This is a very limited interface at the moment. It will probably need to be extended.

append_step(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to prior steps on the path, and to other paths, must remain valid.

C++: handlegraph::MutablePathHandleGraph::append_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

assign(*self: bdsg.handlegraph.MutablePathHandleGraph*, *: bdsg.handlegraph.MutablePathHandleGraph*) → bdsg.handlegraph.MutablePathHandleGraph

C++: handlegraph::MutablePathHandleGraph::operator=(const class handlegraph::MutablePathHandleGraph &) –> class handlegraph::MutablePathHandleGraph &

create_path_handle(**args*, ***kwargs*)

Overloaded function.

1. create_path_handle(self: bdsg.handlegraph.MutablePathHandleGraph, name: str) -> bdsg.handlegraph.path_handle_t

2. create_path_handle(self: bdsg.handlegraph.MutablePathHandleGraph, name: str, is_circular: bool) -> bdsg.handlegraph.path_handle_t

Create a path with the given name. The caller must ensure that no path with the given name exists already, or the behavior is undefined. Returns a handle to the created empty path. Handles to other paths must remain valid.

C++: handlegraph::MutablePathHandleGraph::create_path_handle(const std::string &, bool) –> struct handlegraph::path_handle_t

destroy_path(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*) → None

Destroy the given path. Invalidates handles to the path and its steps.

C++: handlegraph::MutablePathHandleGraph::destroy_path(const struct handlegraph::path_handle_t &) –> void

prepend_step(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_prepend: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Prepend a visit to a node to the given path. Returns a handle to the new first step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to later steps on the path, and to other paths, must remain valid.

C++: handlegraph::MutablePathHandleGraph::prepend_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

rewrite_segment(*self: bdsg.handlegraph.MutablePathHandleGraph*, *segment_begin: bdsg.handlegraph.step_handle_t*, *segment_end: bdsg.handlegraph.step_handle_t*, *new_segment: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t>>*) → Tuple[bdsg.handlegraph.step_handle_t, bdsg.handlegraph.step_handle_t]

Delete a segment of a path and rewrite it as some other sequence of steps. Returns a pair of step_handle_t's that indicate the range of the new segment in the path. The segment to delete should be designated by the first (begin) and past-last (end) step handles. If the step that is returned by path_begin is deleted, path_begin will now return the first step from the new segment or, in the case that the new segment is empty, the step used as segment_end. Empty ranges consist of two copies of the same step handle. Empty ranges in empty paths consist of two copies of the end sentinel handle for the path. Rewriting an empty range inserts before the provided end handle.

C++: handlegraph::MutablePathHandleGraph::rewrite_segment(const struct handlegraph::step_handle_t &, const struct handlegraph::step_handle_t &, const class std::vector<handlegraph::handle_t> &) –> struct std::pair<struct handlegraph::step_handle_t, struct handlegraph::step_handle_t>

set_circularity(*self: bdsg.handlegraph.MutablePathHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *circular: bool*) → None

Make a path circular or non-circular. If the path is becoming circular, the last step is joined to the first step. If the path is becoming linear, the step considered "last" is unjoined from the step considered "first" according to the method path_begin.

C++: handlegraph::MutablePathHandleGraph::set_circularity(const struct handlegraph::path_handle_t &, bool) –> void

class bdsg.handlegraph.**MutablePathMutableHandleGraph**

Bases: *bdsg.handlegraph.MutablePathHandleGraph*, *bdsg.handlegraph.MutableHandleGraph*

This is the interface for a graph which is mutable and which has paths which are also mutable.

assign(*self: bdsg.handlegraph.MutablePathMutableHandleGraph*, *: bdsg.handlegraph.MutablePathMutableHandleGraph*) → bdsg.handlegraph.MutablePathMutableHandleGraph

C++: handlegraph::MutablePathMutableHandleGraph::operator=(const class handlegraph::MutablePathMutableHandleGraph &) –> class handlegraph::MutablePathMutableHandleGraph &

class bdsg.handlegraph.**MutablePathDeletableHandleGraph**

Bases: *bdsg.handlegraph.MutablePathMutableHandleGraph*, *bdsg.handlegraph.DeletableHandleGraph*

This is the interface for a graph which is deletable and which has paths which are also mutable.

assign(*self: bdsg.handlegraph.MutablePathDeletableHandleGraph*, *: bdsg.handlegraph.MutablePathDeletableHandleGraph*) → bdsg.handlegraph.MutablePathDeletableHandleGraph

C++: handlegraph::MutablePathDeletableHandleGraph::operator=(const class handlegraph::MutablePathDeletableHandleGraph &) –> class handlegraph::MutablePathDeletableHandleGraph &

Note that there is no `bdsg.handlegraph.PathMutableHandleGraph` or `bdsg.handlegraph.PathDeletableHandleGraph`; it does not make sense for the paths to be static while the graph can be modified.

Position and Ordering Interfaces

For paths, there is also the *bdsg.handlegraph.PathPositionHandleGraph* which provides efficient random access by or lookup of base offset along each embedded path. Additionally, there is *bdsg.handlegraph.VectorizableHandleGraph* which provides the same operations for a linearization of all of the graph's bases. There is also a *bdsg.handlegraph.RankedHandleGraph* interface, which provides an ordering, though not necessarily a base-level linearization, of nodes and edges.

class bdsg.handlegraph.**PathPositionHandleGraph**

Bases: *bdsg.handlegraph.PathHandleGraph*

This is the interface for a path handle graph with path positions

assign(*self: bdsg.handlegraph.PathPositionHandleGraph*, *: bdsg.handlegraph.PathPositionHandleGraph*) → bdsg.handlegraph.PathPositionHandleGraph

C++: handlegraph::PathPositionHandleGraph::operator=(const class handlegraph::PathPositionHandleGraph &) –> class handlegraph::PathPositionHandleGraph &

for_each_step_position_on_handle(*self: bdsg.handlegraph.PathPositionHandleGraph*, *handle: bdsg.handlegraph.handle_t*, *iteratee: Callable[[bdsg.handlegraph.step_handle_t, bool, int], bool]*) → bool

Execute an iteratee on each step on a path, along with its orientation relative to the path (true if it is reverse the orientation of the handle on the path), and its position measured in bases of sequence along the path. Positions are always measured on the forward strand.

Iteration will stop early if the iteratee returns false. This method returns false if iteration was stopped early, else true

C++: handlegraph::PathPositionHandleGraph::for_each_step_position_on_handle(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &, const bool &, const unsigned long &)> &) const –> bool

get_path_length(*self: bdsg.handlegraph.PathPositionHandleGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the length of a path measured in bases of sequence.

C++: handlegraph::PathPositionHandleGraph::get_path_length(const struct handlegraph::path_handle_t &) const –> unsigned long

get_position_of_step (*self: bdsg.handlegraph.PathPositionHandleGraph*, *step: bdsg.handlegraph.step_handle_t*) → int

Returns the position along the path of the beginning of this step measured in bases of sequence. In a circular path, positions start at the step returned by path_begin().

C++: handlegraph::PathPositionHandleGraph::get_position_of_step(const struct handlegraph::step_handle_t &) const –> unsigned long

get_step_at_position (*self: bdsg.handlegraph.PathPositionHandleGraph*, *path: bdsg.handlegraph.path_handle_t*, *position: int*) → bdsg.handlegraph.step_handle_t

Returns the step at this position, measured in bases of sequence starting at the step returned by path_begin(). If the position is past the end of the path, returns path_end().

C++: handlegraph::PathPositionHandleGraph::get_step_at_position(const struct handlegraph::path_handle_t &, const unsigned long &) const –> struct handlegraph::step_handle_t

class bdsg.handlegraph.**VectorizableHandleGraph**

Bases: *bdsg.handlegraph.RankedHandleGraph*

Defines an interface providing a vectorization of the graph nodes and edges, which can be co-inherited alongside HandleGraph.

assign (*self: bdsg.handlegraph.VectorizableHandleGraph*, *: bdsg.handlegraph.VectorizableHandleGraph*) → bdsg.handlegraph.VectorizableHandleGraph

C++: handlegraph::VectorizableHandleGraph::operator=(const class handlegraph::VectorizableHandleGraph &) –> class handlegraph::VectorizableHandleGraph &

edge_index (*self: bdsg.handlegraph.VectorizableHandleGraph*, *edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]*) → int

Return a unique index among edges in the graph

C++: handlegraph::VectorizableHandleGraph::edge_index(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) const –> unsigned long

node_at_vector_offset (*self: bdsg.handlegraph.VectorizableHandleGraph*, *offset: int*) → int

Return the node overlapping the given offset in the implicit node vector

C++: handlegraph::VectorizableHandleGraph::node_at_vector_offset(const unsigned long &) const –> long long

node_vector_offset (*self: bdsg.handlegraph.VectorizableHandleGraph*, *node_id: int*) → int

Return the start position of the node in a (possibly implict) sorted array constructed from the concatenation of the node sequences

C++: handlegraph::VectorizableHandleGraph::node_vector_offset(const long long &) const –> unsigned long

class bdsg.handlegraph.**RankedHandleGraph**

Bases: *bdsg.handlegraph.HandleGraph*

Defines an interface for a handle graph that can rank its nodes and handles.

Doesn't actually require an efficient node lookup by sequence position as in VectorizableHandleGraph.

assign (*self: bdsg.handlegraph.RankedHandleGraph*, *: bdsg.handlegraph.RankedHandleGraph*) → bdsg.handlegraph.RankedHandleGraph

C++: handlegraph::RankedHandleGraph::operator=(const class handlegraph::RankedHandleGraph &) –> class handlegraph::RankedHandleGraph &

handle_to_rank(*self: bdsg.handlegraph.RankedHandleGraph*, *handle: bdsg.handlegraph.handle_t*) → int

Return the rank of a handle (ranks start at 1 and are dense, and each orientation has its own rank). Handle ranks may not have anything to do with node ranks.

C++: handlegraph::RankedHandleGraph::handle_to_rank(const struct handlegraph::handle_t &) const –> unsigned long

id_to_rank(*self: bdsg.handlegraph.RankedHandleGraph*, *node_id: int*) → int
Return the rank of a node (ranks start at 1 and are dense).

C++: handlegraph::RankedHandleGraph::id_to_rank(const long long &) const –> unsigned long

rank_to_handle(*self: bdsg.handlegraph.RankedHandleGraph*, *rank: int*) → bdsg.handlegraph.handle_t
Return the handle with a given rank.

C++: handlegraph::RankedHandleGraph::rank_to_handle(const unsigned long &) const –> struct handlegraph::handle_t

rank_to_id(*self: bdsg.handlegraph.RankedHandleGraph*, *rank: int*) → int
Return the node with a given rank.

C++: handlegraph::RankedHandleGraph::rank_to_id(const unsigned long &) const –> long long

Algorithm implementers are encouraged to take the least capable graph type necessary for their algorithm to function.

SerializableHandleGraph

Orthogonal to the mutability and paths hierarchy, there is a *bdsg.handlegraph.SerializableHandleGraph* interface that is implemented by graphs that can be saved to and loaded from disk. The C++ API supports saving to and loading from C++ streams, but the Python API provides only the ability to save to or load from filenames.

class bdsg.handlegraph.**SerializableHandleGraph**
Bases: pybind11_builtins.pybind11_object

assign(*self: bdsg.handlegraph.SerializableHandleGraph*, *: bdsg.handlegraph.SerializableHandleGraph*) → bdsg.handlegraph.SerializableHandleGraph
C++: handlegraph::SerializableHandleGraph::operator=(const class handlegraph::SerializableHandleGraph &) –> class handlegraph::SerializableHandleGraph &

deserialize(*self: bdsg.handlegraph.SerializableHandleGraph*, *filename: str*) → None

Sets the contents of this graph to the contents of a serialized graph from a file. The serialized graph must be from the same implementation of the HandleGraph interface as is calling deserialize(). Can only be called on an empty graph.

C++: handlegraph::SerializableHandleGraph::deserialize(const std::string &) –> void

get_magic_number(*self: bdsg.handlegraph.SerializableHandleGraph*) → int

Returns a number that is specific to the serialized implementation for type checking. Does not depend on the contents of any particular instantiation (i.e. behaves as if static, but cannot be static and virtual).

C++: handlegraph::SerializableHandleGraph::get_magic_number() const –> unsigned int

serialize(*self: bdsg.handlegraph.SerializableHandleGraph*, *filename: str*) → None

Write the contents of this graph to a named file. Makes sure to include a leading magic number.

C++: handlegraph::SerializableHandleGraph::serialize(const std::string &) const –> void

4.2 libbdsg Handle Graph Implementations

The *bdsg.bdsg* module provides useful implementations of the Handle Graph API.

Bindings for ::bdsg namespace

4.2.1 Full Graph Implementations

There are three full graph implementations in the module: *bdsg.bdsg.PackedGraph*, *bdsg.bdsg.HashGraph*, and *bdsg.bdsg.ODGI*.

PackedGraph

class bdsg.bdsg.**PackedGraph**

Bases: *bdsg.handlegraph.MutablePathDeletableHandleGraph*, *bdsg.handlegraph.SerializableHandleGraph*

PackedGraph is a HandleGraph implementation designed to use very little memory. It stores its data in bit-packed integer vectors, which are dynamically widened as needed in O(1) amortized time. Within these vectors, graphs are stored using adjacency linked lists.

Since removals of elements can cause slots in the internal vectors to become unused, the graph will occasionally defragment itself after some modification operations, which involves copying its internal data structures.

This implementation is a good choice when working with very large graphs, where the final memory usage of the constructed graph must be minimized. It is not a good choice when large fractions of the graph will need to be deleted and replaced; ODGI or HashGraph may be better for such workloads.

append_step(*self: bdsg.bdsg.PackedGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. Handles to prior steps on the path, and to other paths, must remain valid.

C++: bdsg::PackedGraph::append_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

apply_ordering(**args*, ***kwargs*)

Overloaded function.

1. apply_ordering(self: bdsg.bdsg.PackedGraph, order: bdsg.std.vector_handlegraph_handle_t) -> None
2. apply_ordering(self: bdsg.bdsg.PackedGraph, order: bdsg.std.vector_handlegraph_handle_t, compact_ids: bool) -> None

Reorder the graph's internal structure to match that given. This sets the order that is used for iteration in functions like for_each_handle. Optionally compact the id space of the graph to match the ordering, from 1->|ordering|. This may be a no-op in the case of graph implementations that do not have any mechanism to maintain an ordering.

C++: bdsg::PackedGraph::apply_ordering(const class std::vector<handlegraph::handle_t> &, bool) –> void

apply_orientation(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Alter the node that the given handle corresponds to so the orientation indicated by the handle becomes the node's local forward orientation. Rewrites all edges pointing to the node and the node's sequence to reflect this. Invalidates all handles to the node (including the one passed). Returns a new, valid handle to the node in its new forward orientation. Note that it is possible for the node's ID to change. Does not update any stored paths. May change the ordering of the underlying graph.

C++: bdsg::PackedGraph::apply_orientation(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t

clear(*self: bdsg.bdsg.PackedGraph*) → None

Remove all nodes and edges. Does not update any stored paths.

C++: bdsg::PackedGraph::clear() –> void

create_edge(*self: bdsg.bdsg.PackedGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → None

Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

C++: bdsg::PackedGraph::create_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

create_handle(**args*, ***kwargs*)

Overloaded function.

1. create_handle(self: bdsg.bdsg.PackedGraph, sequence: str) -> bdsg.handlegraph.handle_t

Create a new node with the given sequence and return the handle. The sequence may not be empty.

C++: bdsg::PackedGraph::create_handle(const std::string &) –> struct handlegraph::handle_t

2. create_handle(self: bdsg.bdsg.PackedGraph, sequence: str, id: int) -> bdsg.handlegraph.handle_t

Create a new node with the given id and sequence, then return the handle. The sequence may not be empty. The ID must be strictly greater than 0.

C++: bdsg::PackedGraph::create_handle(const std::string &, const long long &) –> struct handlegraph::handle_t

create_path_handle(**args*, ***kwargs*)

Overloaded function.

1. create_path_handle(self: bdsg.bdsg.PackedGraph, name: str) -> bdsg.handlegraph.path_handle_t
2. create_path_handle(self: bdsg.bdsg.PackedGraph, name: str, is_circular: bool) -> bdsg.handlegraph.path_handle_t

Create a path with the given name. The caller must ensure that no path with the given name exists already, or the behavior is undefined. Returns a handle to the created empty path. Handles to other paths must remain valid.

C++: bdsg::PackedGraph::create_path_handle(const std::string &, bool) –> struct handlegraph::path_handle_t

destroy_edge(*self: bdsg.bdsg.PackedGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → None

Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

C++: bdsg::PackedGraph::destroy_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

destroy_handle(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → None

Remove the node belonging to the given handle and all of its edges. Destroys any paths in which the node participates. Invalidates the destroyed handle. May be called during serial for_each_handle iteration **ONLY** on the node being iterated. May **NOT** be called during parallel for_each_handle iteration. May **NOT** be called on the node from which edges are being followed during follow_edges. May **NOT** be called during iteration over paths, if it would destroy a path. May **NOT** be called during iteration along a path, if it would destroy that path.

C++: bdsg::PackedGraph::destroy_handle(const struct handlegraph::handle_t &) –> void

destroy_path(*self: bdsg.bdsg.PackedGraph*, *path: bdsg.handlegraph.path_handle_t*) → None

Destroy the given path. Invalidates handles to the path and its node steps.

C++: bdsg::PackedGraph::destroy_path(const struct handlegraph::path_handle_t &) –> void

divide_handle(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*, *offsets: bdsg.std.vector_unsigned_long*) → bdsg.std.vector_handlegraph_handle_t

Split a handle's underlying node at the given offsets in the handle's orientation. Returns all of the handles to the parts. Other handles to the node being split may be invalidated. The split pieces stay in the same local forward orientation as the original node, but the returned handles come in the order and orientation appropriate for the handle passed in. Updates stored paths.

C++: bdsg::PackedGraph::divide_handle(const struct handlegraph::handle_t &, const class std::vector<unsigned long> &) –> class std::vector<handlegraph::handle_t>

flip(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Invert the orientation of a handle (potentially without getting its ID)

C++: bdsg::PackedGraph::flip(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

follow_edges_impl(*self: bdsg.bdsg.PackedGraph, handle: bdsg.handlegraph.handle_t, go_left: bool, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]*) → bool

Loop over all the handles to next/previous (right/left) nodes. Passes them to a callback which returns false to stop iterating and true to continue. Returns true if we finished and false if we stopped early.

C++: bdsg::PackedGraph::follow_edges_impl(const struct handlegraph::handle_t &, bool, const class std::function<bool (const struct handlegraph::handle_t &)> &) const –> bool

for_each_handle_impl(**args*, ***kwargs*)

Overloaded function.

1. for_each_handle_impl(self: bdsg.bdsg.PackedGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]) -> bool

2. for_each_handle_impl(self: bdsg.bdsg.PackedGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool], parallel: bool) -> bool

Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false. Can be told to run in parallel, in which case stopping after a false return value is on a best-effort basis and iteration order is not defined.

C++: bdsg::PackedGraph::for_each_handle_impl(const class std::function<bool (const struct handlegraph::handle_t &)> &, bool) const –> bool

for_each_path_handle_impl(*self: bdsg.bdsg.PackedGraph, iteratee: Callable[[bdsg.handlegraph.path_handle_t], bool]*) → bool

Execute a function on each path in the graph

C++: bdsg::PackedGraph::for_each_path_handle_impl(const class std::function<bool (const struct handlegraph::path_handle_t &)> &) const –> bool

for_each_step_on_handle_impl(*self: bdsg.bdsg.PackedGraph, handle: bdsg.handlegraph.handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool

Calls the given function for each step of the given handle on a path.

C++: bdsg::PackedGraph::for_each_step_on_handle_impl(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

get_base(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*) → str

Returns one base of a handle's sequence, in the orientation of the handle.

C++: bdsg::PackedGraph::get_base(const struct handlegraph::handle_t &, unsigned long) const –> char

get_edge_count(*self: bdsg.bdsg.PackedGraph*) → int

Return the total number of edges in the graph. If not overridden, counts them all in linear time.

C++: bdsg::PackedGraph::get_edge_count() const –> unsigned long

get_handle(**args*, ***kwargs*)

Overloaded function.

1. get_handle(self: bdsg.bdsg.PackedGraph, node_id: int) -> bdsg.handlegraph.handle_t
2. get_handle(self: bdsg.bdsg.PackedGraph, node_id: int, is_reverse: bool) -> bdsg.handlegraph.handle_t

Look up the handle for the node with the given ID in the given orientation

C++: bdsg::PackedGraph::get_handle(const long long &, bool) const –> struct handlegraph::handle_t

get_handle_of_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.handle_t

Get a node handle (node ID and orientation) from a handle to an step on a path

C++: bdsg::PackedGraph::get_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::handle_t

get_id(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → int

Get the ID from a handle

C++: bdsg::PackedGraph::get_id(const struct handlegraph::handle_t &) const –> long long

get_is_circular(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool

Look up whether a path is circular

C++: bdsg::PackedGraph::get_is_circular(const struct handlegraph::path_handle_t &) const –> bool

get_is_reverse(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → bool

Get the orientation of a handle

C++: bdsg::PackedGraph::get_is_reverse(const struct handlegraph::handle_t &) const –> bool

get_length(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → int

Get the length of a node

C++: bdsg::PackedGraph::get_length(const struct handlegraph::handle_t &) const –> unsigned long

get_magic_number(*self: bdsg.bdsg.PackedGraph*) → int
Returns a static high-entropy number to indicate the class

C++: bdsg::PackedGraph::get_magic_number() const –> unsigned int

get_next_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the next step on the path. If the given step is the final step of a non-circular path, returns the past-the-last step that is also returned by path_end. In a circular path, the "last" step will loop around to the "first" (i.e. the one returned by path_begin). Note: to iterate over each step one time, even in a circular path, consider for_each_step_in_path.

C++: bdsg::PackedGraph::get_next_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_node_count(*self: bdsg.bdsg.PackedGraph*) → int
Return the number of nodes in the graph

C++: bdsg::PackedGraph::get_node_count() const –> unsigned long

get_path_count(*self: bdsg.bdsg.PackedGraph*) → int
Returns the number of paths stored in the graph

C++: bdsg::PackedGraph::get_path_count() const –> unsigned long

get_path_handle(*self: bdsg.bdsg.PackedGraph*, *path_name: str*) → bdsg.handlegraph.path_handle_t

Look up the path handle for the given path name. The path with that name must exist.

C++: bdsg::PackedGraph::get_path_handle(const std::string &) const –> struct handlegraph::path_handle_t

get_path_handle_of_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.path_handle_t
Returns a handle to the path that an step is on

C++: bdsg::PackedGraph::get_path_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::path_handle_t

get_path_name(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → str
Look up the name of a path from a handle to it

C++: bdsg::PackedGraph::get_path_name(const struct handlegraph::path_handle_t &) const –> std::string

get_previous_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the previous step on the path. If the given step is the first step of a non-circular path, this method has undefined behavior. In a circular path, it will loop around from the "first" step (i.e. the one returned by path_begin) to the "last" step. Note: to iterate over each step one time, even in a circular path, consider for_each_step_in_path.

C++: bdsg::PackedGraph::get_previous_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_sequence(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*) → str
Get the sequence of a node, presented in the handle's local forward orientation.

C++: bdsg::PackedGraph::get_sequence(const struct handlegraph::handle_t &) const –> std::string

get_step_count(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int

Returns the number of node steps in the path

C++: bdsg::PackedGraph::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

get_subsequence(*self: bdsg.bdsg.PackedGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*, *size: int*) → str

Returns a substring of a handle's sequence, in the orientation of the handle. If the indicated substring would extend beyond the end of the handle's sequence, the return value is truncated to the sequence's end.

C++: bdsg::PackedGraph::get_subsequence(const struct handlegraph::handle_t &, unsigned long, unsigned long) const –> std::string

get_total_length(*self: bdsg.bdsg.PackedGraph*) → int

Return the total length of all nodes in the graph, in bp. If not overridden, loops over all nodes in linear time.

C++: bdsg::PackedGraph::get_total_length() const –> unsigned long

has_next_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool

Returns true if the step is not the last step in a non-circular path.

C++: bdsg::PackedGraph::has_next_step(const struct handlegraph::step_handle_t &) const –> bool

has_node(*self: bdsg.bdsg.PackedGraph*, *node_id: int*) → bool

Method to check if a node exists by ID

C++: bdsg::PackedGraph::has_node(long long) const –> bool

has_path(*self: bdsg.bdsg.PackedGraph*, *path_name: str*) → bool

Determine if a path name exists and is legal to get a path handle for.

C++: bdsg::PackedGraph::has_path(const std::string &) const –> bool

has_previous_step(*self: bdsg.bdsg.PackedGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool

Returns true if the step is not the first step in a non-circular path.

C++: bdsg::PackedGraph::has_previous_step(const struct handlegraph::step_handle_t &) const –> bool

increment_node_ids(*self: bdsg.bdsg.PackedGraph*, *increment: int*) → None

Add the given value to all node IDs

C++: bdsg::PackedGraph::increment_node_ids(long long) –> void

max_node_id(*self: bdsg.bdsg.PackedGraph*) → int

Return the largest ID in the graph, or some larger number if the largest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::PackedGraph::max_node_id() const –> long long

min_node_id(*self: bdsg.bdsg.PackedGraph*) → int

Return the smallest ID in the graph, or some smaller number if the smallest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::PackedGraph::min_node_id() const –> long long

optimize(**args*, ***kwargs*)

Overloaded function.

1. optimize(self: bdsg.bdsg.PackedGraph) -> None
2. optimize(self: bdsg.bdsg.PackedGraph, allow_id_reassignment: bool) -> None

Adjust the representation of the graph in memory to improve performance. Optionally, allow the node IDs to be reassigned to further improve performance. Note: Ideally, this method is called one time once there is expected to be few graph modifications in the future.

C++: bdsg::PackedGraph::optimize(bool) –> void

path_back(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the last step, which will be an arbitrary step in a circular path that we consider "last" based on our construction of the path. If the path is empty then the implementation must return the same value as path_front_end().

C++: bdsg::PackedGraph::path_back(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_begin(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the first step, or in a circular path to an arbitrary step considered "first". If the path is empty, returns the past-the-last step returned by path_end.

C++: bdsg::PackedGraph::path_begin(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_end(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position past the end of a path. This position is return by get_next_step for the final step in a path in a non-circular path. Note that get_next_step will *NEVER* return this value for a circular path.

C++: bdsg::PackedGraph::path_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_front_end(*self: bdsg.bdsg.PackedGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position before the beginning of a path. This position is return by get_previous_step for the first step in a path in a non-circular path. Note: get_previous_step will *NEVER* return this value for a circular path.

C++: bdsg::PackedGraph::path_front_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

prepend_step(*self: bdsg.bdsg.PackedGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_prepend: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Prepend a visit to a node to the given path. Returns a handle to the new first step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to later steps on the path, and to other paths, must remain valid.

C++: bdsg::PackedGraph::prepend_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

reassign_node_ids(*self: bdsg.bdsg.PackedGraph, get_new_id: Callable[[int], int]*) → None

Reassign all node IDs as specified by the old->new mapping function.

C++: bdsg::PackedGraph::reassign_node_ids(const class std::function<long long (const long long &)> &) –> void

rewrite_segment(*self: bdsg.bdsg.PackedGraph*, *segment_begin: bdsg.handlegraph.step_handle_t*, *segment_end: bdsg.handlegraph.step_handle_t*, *new_segment: bdsg.std.vector_handlegraph_handle_t*) → Tuple[bdsg.handlegraph.step_handle_t, bdsg.handlegraph.step_handle_t]

Delete a segment of a path and rewrite it as some other sequence of steps. Returns a pair of step_handle_t's that indicate the range of the new segment in the path. The segment to delete should be designated by the first (begin) and past-last (end) step handles. If the step that is returned by path_begin is deleted, path_begin will now return the first step from the new segment or, in the case that the new segment is empty, the step used as segment_end. Empty ranges consist of two copies of the same step handle. Empty ranges in empty paths consist of two copies of the end sentinel handle for the path. Rewriting an empty range inserts before the provided end handle.

C++: bdsg::PackedGraph::rewrite_segment(const struct handlegraph::step_handle_t &, const struct handlegraph::step_handle_t &, const class std::vector<handlegraph::handle_t> &) –> struct std::pair<struct handlegraph::step_handle_t, struct handlegraph::step_handle_t>

set_circularity(*self: bdsg.bdsg.PackedGraph*, *path: bdsg.handlegraph.path_handle_t*, *circular: bool*) → None

Make a path circular or non-circular. If the path is becoming circular, the last step is joined to the first step. If the path is becoming linear, the step considered "last" is unjoined from the step considered "first" according to the method path_begin.

C++: bdsg::PackedGraph::set_circularity(const struct handlegraph::path_handle_t &, bool) –> void

set_id_increment(*self: bdsg.bdsg.PackedGraph*, *min_id: int*) → None

Set a minimum id to increment the id space by, used as a hint during construction. May have no effect on a backing implementation.

C++: bdsg::PackedGraph::set_id_increment(const long long &) –> void

HashGraph

class bdsg.bdsg.**HashGraph**

Bases: *bdsg.handlegraph.MutablePathDeletableHandleGraph*, *bdsg.handlegraph.SerializableHandleGraph*

HashGraph is a HandleGraph implementation designed for simplicity. Nodes are plain C++ objects stored in a hash map, which contain C++ vectors representing their adjacencies.

HashGraph is a good choice when fast access to or modification of a graph is required, but can use more memory than other graph implementations.

append_step(*self: bdsg.bdsg.HashGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to prior steps on the path, and to other paths, must remain valid.

C++: bdsg::HashGraph::append_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

apply_ordering(**args*, ***kwargs*)

Overloaded function.

1. apply_ordering(self: bdsg.bdsg.HashGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >) -> None

2. apply_ordering(self: bdsg.bdsg.HashGraph, order: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >, compact_ids: bool) -> None

Reorder the graph's internal structure to match that given. This sets the order that is used for iteration in functions like for_each_handle. Optionally compact the id space of the graph to match the ordering, from 1->|ordering|. This may be a no-op in the case of graph implementations that do not have any mechanism to maintain an ordering.

C++: bdsg::HashGraph::apply_ordering(const class std::vector<handlegraph::handle_t> &, bool) –> void

apply_orientation(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Alter the node that the given handle corresponds to so the orientation indicated by the handle becomes the node's local forward orientation. Rewrites all edges pointing to the node and the node's sequence to reflect this. Invalidates all handles to the node (including the one passed). Returns a new, valid handle to the node in its new forward orientation. Note that it is possible for the node's ID to change. Does not update any stored paths. May change the ordering of the underlying graph.

C++: bdsg::HashGraph::apply_orientation(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t

assign(*self: bdsg.bdsg.HashGraph*, *: bdsg.bdsg.HashGraph*) → bdsg.bdsg.HashGraph
C++: bdsg::HashGraph::operator=(const class bdsg::HashGraph &) –> class bdsg::HashGraph &

clear(*self: bdsg.bdsg.HashGraph*) → None
Remove all nodes and edges. Does not update any stored paths.

C++: bdsg::HashGraph::clear() –> void

create_edge(*self: bdsg.bdsg.HashGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → None

Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

C++: bdsg::HashGraph::create_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

create_handle(**args*, ***kwargs*)
Overloaded function.

1. create_handle(self: bdsg.bdsg.HashGraph, sequence: str) -> bdsg.handlegraph.handle_t

Create a new node with the given sequence and return the handle. The sequence may not be empty.

C++: bdsg::HashGraph::create_handle(const std::string &) –> struct handlegraph::handle_t

2. create_handle(self: bdsg.bdsg.HashGraph, sequence: str, id: int) -> bdsg.handlegraph.handle_t

Create a new node with the given id and sequence, then return the handle. The sequence may not be empty. The ID must be strictly greater than 0.

C++: bdsg::HashGraph::create_handle(const std::string &, const long long &) –> struct handlegraph::handle_t

create_path_handle(**args*, ***kwargs*)
Overloaded function.

1. create_path_handle(self: bdsg.bdsg.HashGraph, name: str) -> bdsg.handlegraph.path_handle_t
2. create_path_handle(self: bdsg.bdsg.HashGraph, name: str, is_circular: bool) -> bdsg.handlegraph.path_handle_t

Create a path with the given name. The caller must ensure that no path with the given name exists already, or the behavior is undefined. Returns a handle to the created empty path. Handles to other paths must remain valid.

C++: bdsg::HashGraph::create_path_handle(const std::string &, bool) –> struct handlegraph::path_handle_t

destroy_edge(*self: bdsg.bdsg.HashGraph*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → None

Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

C++: bdsg::HashGraph::destroy_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

destroy_handle(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → None

Remove the node belonging to the given handle and all of its edges. Destroys any paths in which the node participates. Invalidates the destroyed handle. May be called during serial for_each_handle iteration **ONLY** on the node being iterated. May **NOT** be called during parallel for_each_handle iteration. May **NOT** be called on the node from which edges are being followed during follow_edges. May **NOT** be called during iteration over paths, if it would destroy a path. May **NOT** be called during iteration along a path, if it would destroy that path.

C++: bdsg::HashGraph::destroy_handle(const struct handlegraph::handle_t &) –> void

destroy_path(*self: bdsg.bdsg.HashGraph*, *path: bdsg.handlegraph.path_handle_t*) → None
Destroy the given path. Invalidates handles to the path and its node steps.

C++: bdsg::HashGraph::destroy_path(const struct handlegraph::path_handle_t &) –> void

divide_handle(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*, *offsets: std::vector<unsigned long, std::allocator<unsigned long> >*) → std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >

Split a handle's underlying node at the given offsets in the handle's orientation. Returns all of the handles to the parts. Other handles to the node being split may be invalidated. The split pieces stay in the same local forward orientation as the original node, but the returned handles come in the order and orientation appropriate for the handle passed in. Updates stored paths.

C++: bdsg::HashGraph::divide_handle(const struct handlegraph::handle_t &, const class std::vector<unsigned long> &) –> class std::vector<handlegraph::handle_t>

flip(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t
Invert the orientation of a handle (potentially without getting its ID)

C++: bdsg::HashGraph::flip(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

follow_edges_impl(*self: bdsg.bdsg.HashGraph, handle: bdsg.handlegraph.handle_t, go_left: bool, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]*) → bool

Loop over all the handles to next/previous (right/left) nodes. Passes them to a callback which returns false to stop iterating and true to continue. Returns true if we finished and false if we stopped early.

C++: bdsg::HashGraph::follow_edges_impl(const struct handlegraph::handle_t &, bool, const class std::function<bool (const struct handlegraph::handle_t &)> &) const –> bool

for_each_handle_impl(**args*, ***kwargs*)
Overloaded function.

1. for_each_handle_impl(self: bdsg.bdsg.HashGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool]) -> bool

2. for_each_handle_impl(self: bdsg.bdsg.HashGraph, iteratee: Callable[[bdsg.handlegraph.handle_t], bool], parallel: bool) -> bool

Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false. Can be told to run in parallel, in which case stopping after a false return value is on a best-effort basis and iteration order is not defined.

C++: bdsg::HashGraph::for_each_handle_impl(const class std::function<bool (const struct handlegraph::handle_t &)> &, bool) const –> bool

for_each_path_handle_impl(*self: bdsg.bdsg.HashGraph, iteratee: Callable[[bdsg.handlegraph.path_handle_t], bool]*) → bool
Execute a function on each path in the graph

C++: bdsg::HashGraph::for_each_path_handle_impl(const class std::function<bool (const struct handlegraph::path_handle_t &)> &) const –> bool

for_each_step_on_handle_impl(*self: bdsg.bdsg.HashGraph, handle: bdsg.handlegraph.handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], bool]*) → bool
Calls a function with all steps of a node on paths.

C++: bdsg::HashGraph::for_each_step_on_handle_impl(const struct handlegraph::handle_t &, const class std::function<bool (const struct handlegraph::step_handle_t &)> &) const –> bool

get_base(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*) → str

Returns one base of a handle's sequence, in the orientation of the handle.

C++: bdsg::HashGraph::get_base(const struct handlegraph::handle_t &, unsigned long) const –> char

get_degree(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*, *go_left: bool*) → int
Efficiently get the number of edges attached to one side of a handle.

C++: bdsg::HashGraph::get_degree(const struct handlegraph::handle_t &, bool) const –> unsigned long

get_handle(**args*, ***kwargs*)
Overloaded function.

1. get_handle(self: bdsg.bdsg.HashGraph, node_id: int) -> bdsg.handlegraph.handle_t

2. get_handle(self: bdsg.bdsg.HashGraph, node_id: int, is_reverse: bool) -> bdsg.handlegraph.handle_t

Look up the handle for the node with the given ID in the given orientation

C++: bdsg::HashGraph::get_handle(const long long &, bool) const –> struct handlegraph::handle_t

get_handle_of_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.handle_t
Get a node handle (node ID and orientation) from a handle to a step on a path

C++: bdsg::HashGraph::get_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::handle_t

get_id(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → int
Get the ID from a handle

C++: bdsg::HashGraph::get_id(const struct handlegraph::handle_t &) const –> long long

get_is_circular(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool
Look up whether a path is circular

C++: bdsg::HashGraph::get_is_circular(const struct handlegraph::path_handle_t &) const –> bool

get_is_reverse(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → bool
Get the orientation of a handle

C++: bdsg::HashGraph::get_is_reverse(const struct handlegraph::handle_t &) const –> bool

get_length(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → int
Get the length of a node

C++: bdsg::HashGraph::get_length(const struct handlegraph::handle_t &) const –> unsigned long

get_magic_number(*self: bdsg.bdsg.HashGraph*) → int
Returns a static high-entropy number to indicate the class

C++: bdsg::HashGraph::get_magic_number() const –> unsigned int

get_next_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the next step on the path. If the given step is the final step of a non-circular path, returns the past-the-last step that is also returned by path_end. In a circular path, the "last" step will loop around to the "first" (i.e. the one returned by path_begin). Note: to iterate over each step one time, even in a circular path, consider for_each_step_in_path.

C++: bdsg::HashGraph::get_next_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_node_count(*self: bdsg.bdsg.HashGraph*) → int

Return the number of nodes in the graph TODO: can't be node_count because XG has a field named node_count.

C++: bdsg::HashGraph::get_node_count() const –> unsigned long

get_path_count(*self: bdsg.bdsg.HashGraph*) → int
Returns the number of paths stored in the graph

C++: bdsg::HashGraph::get_path_count() const –> unsigned long

get_path_handle(*self: bdsg.bdsg.HashGraph*, *path_name: str*) → bdsg.handlegraph.path_handle_t

Look up the path handle for the given path name. The path with that name must exist.

C++: bdsg::HashGraph::get_path_handle(const std::string &) const –> struct handlegraph::path_handle_t

get_path_handle_of_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.path_handle_t
Returns a handle to the path that a step is on

C++: bdsg::HashGraph::get_path_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::path_handle_t

get_path_name(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → str
Look up the name of a path from a handle to it

C++: bdsg::HashGraph::get_path_name(const struct handlegraph::path_handle_t &) const –> std::string

get_previous_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the previous step on the path. If the given step is the first step of a non-circular path, this method has undefined behavior. In a circular path, it will loop around from the "first" step (i.e. the one returned by path_begin) to the "last" step. Note: to iterate over each step one time, even in a circular path, consider for_each_step_in_path.

C++: bdsg::HashGraph::get_previous_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_sequence(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*) → str
Get the sequence of a node, presented in the handle's local forward orientation.

C++: bdsg::HashGraph::get_sequence(const struct handlegraph::handle_t &) const –> std::string

get_step_count(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → int
Returns the number of node steps in the path

C++: bdsg::HashGraph::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

get_subsequence(*self: bdsg.bdsg.HashGraph*, *handle: bdsg.handlegraph.handle_t*, *index: int*, *size: int*) → str

Returns a substring of a handle's sequence, in the orientation of the handle. If the indicated substring would extend beyond the end of the handle's sequence, the return value is truncated to the sequence's end.

C++: bdsg::HashGraph::get_subsequence(const struct handlegraph::handle_t &, unsigned long, unsigned long) const –> std::string

has_next_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step is not the last step in a non-circular path.

C++: bdsg::HashGraph::has_next_step(const struct handlegraph::step_handle_t &) const –> bool

has_node(*self: bdsg.bdsg.HashGraph*, *node_id: int*) → bool
Method to check if a node exists by ID

C++: bdsg::HashGraph::has_node(long long) const –> bool

has_path(*self: bdsg.bdsg.HashGraph*, *path_name: str*) → bool
Determine if a path name exists and is legal to get a path handle for.

C++: bdsg::HashGraph::has_path(const std::string &) const –> bool

has_previous_step(*self: bdsg.bdsg.HashGraph*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step is not the first step in a non-circular path.

C++: bdsg::HashGraph::has_previous_step(const struct handlegraph::step_handle_t &) const –> bool

increment_node_ids(*self: bdsg.bdsg.HashGraph*, *increment: int*) → None
Add the given value to all node IDs

C++: bdsg::HashGraph::increment_node_ids(long long) –> void

max_node_id(*self: bdsg.bdsg.HashGraph*) → int

Return the largest ID in the graph, or some larger number if the largest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::HashGraph::max_node_id() const –> long long

min_node_id(*self: bdsg.bdsg.HashGraph*) → int

Return the smallest ID in the graph, or some smaller number if the smallest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::HashGraph::min_node_id() const –> long long

optimize(**args*, ***kwargs*)
Overloaded function.

1. optimize(self: bdsg.bdsg.HashGraph) -> None
2. optimize(self: bdsg.bdsg.HashGraph, allow_id_reassignment: bool) -> None

Adjust the representation of the graph in memory to improve performance. Optionally, allow the node IDs to be reassigned to further improve performance. Note: Ideally, this method is called one time once there is expected to be few graph modifications in the future.

C++: bdsg::HashGraph::optimize(bool) –> void

path_back(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the last step, which will be an arbitrary step in a circular path that we consider "last" based on our construction of the path. If the path is empty then the implementation must return the same value as path_front_end().

C++: bdsg::HashGraph::path_back(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_begin(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the first step, or in a circular path to an arbitrary step considered "first". If the path is empty, returns the past-the-last step returned by path_end.

C++: bdsg::HashGraph::path_begin(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_end(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position past the end of a path. This position is return by get_next_step for the final step in a path in a non-circular path. Note that get_next_step will *NEVER* return this value for a circular path.

C++: bdsg::HashGraph::path_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_front_end(*self: bdsg.bdsg.HashGraph*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious position before the beginning of a path. This position is return by get_previous_step for the first step in a path in a non-circular path. Note: get_previous_step will *NEVER* return this value for a circular path.

C++: bdsg::HashGraph::path_front_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

prepend_step(*self: bdsg.bdsg.HashGraph*, *path: bdsg.handlegraph.path_handle_t*, *to_prepend: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Prepend a visit to a node to the given path. Returns a handle to the new first step on the path which is appended. If the path is cirular, the new step is placed between the steps considered "last" and "first" by the method path_begin. Handles to later steps on the path, and to other paths, must remain valid.

C++: bdsg::HashGraph::prepend_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

reassign_node_ids(*self: bdsg.bdsg.HashGraph, get_new_id: Callable[[int], int]*) → None

Reassign all node IDs as specified by the old->new mapping function.

C++: bdsg::HashGraph::reassign_node_ids(const class std::function<long long (const long long &)> &) –> void

rewrite_segment(*self: bdsg.bdsg.HashGraph*, *segment_begin: bdsg.handlegraph.step_handle_t*, *segment_end: bdsg.handlegraph.step_handle_t*, *new_segment: std::vector<handlegraph::handle_t, std::allocator<handlegraph::handle_t> >*) → Tuple[bdsg.handlegraph.step_handle_t, bdsg.handlegraph.step_handle_t]

Delete a segment of a path and rewrite it as some other sequence of steps. Returns a pair of step_handle_t's that indicate the range of the new segment in the path. The segment to delete should be designated by the first (begin) and past-last (end) step handles. If the step that is returned by path_begin is deleted, path_begin will now return the first step from the new segment or, in the case that the new segment is empty, the step used as segment_end. Empty ranges consist of two copies of the same step handle. Empty ranges in empty paths consist of two copies of the end sentinel handle for the path. Rewriting an empty range inserts before the provided end handle.

C++: bdsg::HashGraph::rewrite_segment(const struct handlegraph::step_handle_t &, const struct handlegraph::step_handle_t &, const class std::vector<handlegraph::handle_t> &) –> struct std::pair<struct handlegraph::step_handle_t, struct handlegraph::step_handle_t>

set_circularity(*self: bdsg.bdsg.HashGraph*, *path: bdsg.handlegraph.path_handle_t*, *circular: bool*) → None

Make a path circular or non-circular. If the path is becoming circular, the last step is joined to the first step. If the path is becoming linear, the step considered "last" is unjoined from the step considered "first" according to the method path_begin.

C++: bdsg::HashGraph::set_circularity(const struct handlegraph::path_handle_t &, bool) –> void

set_id_increment(*self: bdsg.bdsg.HashGraph*, *min_id: int*) → None

Set a minimum id to increment the id space by, used as a hint during construction. May have no effect on a backing implementation.

C++: bdsg::HashGraph::set_id_increment(const long long &) –> void

ODGI

class bdsg.bdsg.**ODGI**

Bases: *bdsg.handlegraph.MutablePathDeletableHandleGraph*, *bdsg.handlegraph.SerializableHandleGraph*

ODGI (Optimized Dynamic Graph Index) is a good all-around HandleGraph implementation. It represents nodes as C++ objects stored in a vector, but aggressively bit-packs and delta-compresses edge and path step information within each node. Using separate node objects can reduce the need for defragmentation as in PackedGraph, but some deletion operations are still lazy.

ODGI is a good implementation to choose if you don't need your graph to be as small as PackedGraph or as fast to access as HashGraph.

append_step(*self: bdsg.bdsg.ODGI*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. Handles to prior steps on the path, and to other paths, must remain valid.

C++: bdsg::ODGI::append_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

apply_ordering(**args*, ***kwargs*)
Overloaded function.

1. apply_ordering(self: bdsg.bdsg.ODGI, order: bdsg.std.vector_handlegraph_handle_t) -> None

2. apply_ordering(self: bdsg.bdsg.ODGI, order: bdsg.std.vector_handlegraph_handle_t, compact_ids: bool) -> None

Reorder the graph's internal structure to match that given. Optionally compact the id space of the graph to match the ordering, from 1->|ordering|.

C++: bdsg::ODGI::apply_ordering(const class std::vector<handlegraph::handle_t> &, bool) –> void

apply_orientation(*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Alter the node that the given handle corresponds to so the orientation indicated by the handle becomes the node's local forward orientation. Rewrites all edges pointing to the node and the node's sequence to reflect this. Invalidates all handles to the node (including the one passed). Returns a new, valid handle to the node in its new forward orientation. Note that it is possible for the node's ID to change. Updates all stored paths. May change the ordering of the underlying graph.

C++: bdsg::ODGI::apply_orientation(const struct handlegraph::handle_t &) –> struct handlegraph::handle_t

apply_path_ordering(*self: bdsg.bdsg.ODGI*, *order: bdsg.std.vector_handlegraph_path_handle_t*) → None
Reorder the graph's paths as given.

C++: bdsg::ODGI::apply_path_ordering(const class std::vector<handlegraph::path_handle_t> &) –> void

assign(*self: bdsg.bdsg.ODGI*, *: bdsg.bdsg.ODGI*) → bdsg.bdsg.ODGI
C++: bdsg::ODGI::operator=(const class bdsg::ODGI &) –> class bdsg::ODGI &

clear(*self: bdsg.bdsg.ODGI*) → None
Remove all nodes, edges, and paths.

C++: bdsg::ODGI::clear() –> void

clear_paths(*self: bdsg.bdsg.ODGI*) → None
Remove all stored paths

C++: bdsg::ODGI::clear_paths() –> void

combine_handles(*self: bdsg.bdsg.ODGI*, *handles: bdsg.std.vector_handlegraph_handle_t*) → bdsg.handlegraph.handle_t
C++: bdsg::ODGI::combine_handles(const class std::vector<handlegraph::handle_t> &) –> struct handlegraph::handle_t

create_edge(**args*, ***kwargs*)
Overloaded function.

1. create_edge(self: bdsg.bdsg.ODGI, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

C++: bdsg::ODGI::create_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. create_edge(self: bdsg.bdsg.ODGI, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for create_edge.

C++: bdsg::ODGI::create_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

create_handle(**args*, ***kwargs*)

Overloaded function.

1. create_handle(self: bdsg.bdsg.ODGI, sequence: str) -> bdsg.handlegraph.handle_t

Create a new node with the given sequence and return the handle. The sequence may not be empty.

C++: bdsg::ODGI::create_handle(const std::string &) –> struct handlegraph::handle_t

2. create_handle(self: bdsg.bdsg.ODGI, sequence: str, id: int) -> bdsg.handlegraph.handle_t

Create a new node with the given id and sequence, then return the handle. The sequence may not be empty. The ID must be strictly greater than 0.

C++: bdsg::ODGI::create_handle(const std::string &, const long long &) –> struct handlegraph::handle_t

create_hidden_handle(*self: bdsg.bdsg.ODGI*, *sequence: str*) → bdsg.handlegraph.handle_t

Create a node that is immediately "hidden" (i.e. to be used for parts of paths that traversed deleted portions of the graph). has_node for the ID of a hidden handle will return false. Also, no edges may be added to it.

C++: bdsg::ODGI::create_hidden_handle(const std::string &) –> struct handlegraph::handle_t

create_path_handle(**args*, ***kwargs*)

Overloaded function.

1. create_path_handle(self: bdsg.bdsg.ODGI, name: str) -> bdsg.handlegraph.path_handle_t

2. create_path_handle(self: bdsg.bdsg.ODGI, name: str, is_circular: bool) -> bdsg.handlegraph.path_handle_t

Create a path with the given name. The caller must ensure that no path with the given name exists already, or the behavior is undefined. Returns a handle to the created empty path. Handles to other paths must remain valid.

C++: bdsg::ODGI::create_path_handle(const std::string &, bool) –> struct handlegraph::path_handle_t

destroy_edge(**args*, ***kwargs*)

Overloaded function.

1. destroy_edge(self: bdsg.bdsg.ODGI, left: bdsg.handlegraph.handle_t, right: bdsg.handlegraph.handle_t) -> None

Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

C++: bdsg::ODGI::destroy_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

2. destroy_edge(self: bdsg.bdsg.ODGI, edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]) -> None

Convenient wrapper for destroy_edge.

C++: bdsg::ODGI::destroy_edge(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &) –> void

destroy_handle(*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → None

Remove the node belonging to the given handle and all of its edges. If any paths visit it, it becomes a "hidden" node accessible only via the paths. Invalidates the destroyed handle. May be called during serial for_each_handle iteration **ONLY** on the node being iterated. May **NOT** be called during parallel for_each_handle iteration. May **NOT** be called on the node from which edges are being followed during follow_edges.

C++: bdsg::ODGI::destroy_handle(const struct handlegraph::handle_t &) –> void

destroy_path(*self: bdsg.bdsg.ODGI*, *path: bdsg.handlegraph.path_handle_t*) → None
Destroy the given path. Invalidates handles to the path and its node steps.

C++: bdsg::ODGI::destroy_path(const struct handlegraph::path_handle_t &) –> void

display(*self: bdsg.bdsg.ODGI*) → None
A helper function to visualize the state of the graph

C++: bdsg::ODGI::display() const –> void

divide_handle(**args*, ***kwargs*)
Overloaded function.

1. divide_handle(self: bdsg.bdsg.ODGI, handle: bdsg.handlegraph.handle_t, offsets: bdsg.std.vector_unsigned_long) -> bdsg.std.vector_handlegraph_handle_t

Split a handle's underlying node at the given offsets in the handle's orientation. Returns all of the handles to the parts. Other handles to the node being split may be invalidated. The split pieces stay in the same local forward orientation as the original node, but the returned handles come in the order and orientation appropriate for the handle passed in. Updates stored paths.

C++: bdsg::ODGI::divide_handle(const struct handlegraph::handle_t &, const class std::vector<unsigned long> &) –> class std::vector<handlegraph::handle_t>

2. divide_handle(self: bdsg.bdsg.ODGI, handle: bdsg.handlegraph.handle_t, offset: int) -> Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

Specialization of divide_handle for a single division point

C++: bdsg::ODGI::divide_handle(const struct handlegraph::handle_t &, unsigned long) –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

edge_handle(*self: bdsg.bdsg.ODGI*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]

A pair of handles can be used as an edge. When so used, the handles have a canonical order and orientation.

C++: bdsg::ODGI::edge_handle(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t>

flip(*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t
Invert the orientation of a handle (potentially without getting its ID)

C++: bdsg::ODGI::flip(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

for_each_step_in_path (*self: bdsg.bdsg.ODGI, path: bdsg.handlegraph.path_handle_t, iteratee: Callable[[bdsg.handlegraph.step_handle_t], None]*) → None

Loop over all the steps along a path, from first through last

C++: bdsg::ODGI::for_each_step_in_path(const struct handlegraph::path_handle_t &, const class std::function<void (const struct handlegraph::step_handle_t &)> &) const –> void

forward (*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Get the locally forward version of a handle

C++: bdsg::ODGI::forward(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

get_degree (*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*, *go_left: bool*) → int

Get the number of edges on the right (go_left = false) or left (go_left = true) side of the given handle. The default implementation is O(n) in the number of edges returned, but graph implementations that track this information more efficiently can override this method.

C++: bdsg::ODGI::get_degree(const struct handlegraph::handle_t &, bool) const –> unsigned long

get_handle (**args*, ***kwargs*)

Overloaded function.

1. get_handle(self: bdsg.bdsg.ODGI, node_id: int) -> bdsg.handlegraph.handle_t

2. get_handle(self: bdsg.bdsg.ODGI, node_id: int, is_reverse: bool) -> bdsg.handlegraph.handle_t

Look up the handle for the node with the given ID in the given orientation

C++: bdsg::ODGI::get_handle(const long long &, bool) const –> struct handlegraph::handle_t

get_handle_of_step (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.handle_t

Get a node handle (node ID and orientation) from a handle to an step on a path

C++: bdsg::ODGI::get_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::handle_t

get_id (*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → int

Get the ID from a handle

C++: bdsg::ODGI::get_id(const struct handlegraph::handle_t &) const –> long long

get_is_circular (*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool

Returns true if the path is circular

C++: bdsg::ODGI::get_is_circular(const struct handlegraph::path_handle_t &) const –> bool

get_is_reverse (*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → bool

Get the orientation of a handle

C++: bdsg::ODGI::get_is_reverse(const struct handlegraph::handle_t &) const –> bool

get_length (*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → int

Get the length of a node

C++: bdsg::ODGI::get_length(const struct handlegraph::handle_t &) const –> unsigned long

get_magic_number (*self: bdsg.bdsg.ODGI*) → int

Return a high-entropy number to indicate which handle graph implementation this is

C++: bdsg::ODGI::get_magic_number() const –> unsigned int

get_next_step (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t

Returns a handle to the next step on the path

C++: bdsg::ODGI::get_next_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_node_count(*self: bdsg.bdsg.ODGI*) → int

Return the number of nodes in the graph TODO: can't be node_count because XG has a field named node_count.

C++: bdsg::ODGI::get_node_count() const –> unsigned long

get_path_count(*self: bdsg.bdsg.ODGI*) → int
Returns the number of paths stored in the graph

C++: bdsg::ODGI::get_path_count() const –> unsigned long

get_path_handle(*self: bdsg.bdsg.ODGI*, *path_name: str*) → bdsg.handlegraph.path_handle_t

Look up the path handle for the given path name. The path with that name must exist.

C++: bdsg::ODGI::get_path_handle(const std::string &) const –> struct handlegraph::path_handle_t

get_path_handle_of_step(*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.path_handle_t
Returns a handle to the path that a step is on

C++: bdsg::ODGI::get_path_handle_of_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::path_handle_t

get_path_name(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → str
Look up the name of a path from a handle to it

C++: bdsg::ODGI::get_path_name(const struct handlegraph::path_handle_t &) const –> std::string

get_previous_step(*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bdsg.handlegraph.step_handle_t
Returns a handle to the previous step on the path

C++: bdsg::ODGI::get_previous_step(const struct handlegraph::step_handle_t &) const –> struct handlegraph::step_handle_t

get_sequence(*self: bdsg.bdsg.ODGI*, *handle: bdsg.handlegraph.handle_t*) → str
Get the sequence of a node, presented in the handle's local forward orientation.

C++: bdsg::ODGI::get_sequence(const struct handlegraph::handle_t &) const –> std::string

get_step_count(**args*, ***kwargs*)
Overloaded function.

1. get_step_count(self: bdsg.bdsg.ODGI, path_handle: bdsg.handlegraph.path_handle_t) -> int

Returns the number of node steps in the path

C++: bdsg::ODGI::get_step_count(const struct handlegraph::path_handle_t &) const –> unsigned long

2. get_step_count(self: bdsg.bdsg.ODGI, handle: bdsg.handlegraph.handle_t) -> int

Returns the number of node steps on the handle

C++: bdsg::ODGI::get_step_count(const struct handlegraph::handle_t &) const –> unsigned long

has_edge(*self: bdsg.bdsg.ODGI*, *left: bdsg.handlegraph.handle_t*, *right: bdsg.handlegraph.handle_t*) → bool
Check if an edge exists

C++: bdsg::ODGI::has_edge(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) const –> bool

has_next_step (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step has a next step on the path, else false

C++: bdsg::ODGI::has_next_step(const struct handlegraph::step_handle_t &) const –> bool

has_node (*self: bdsg.bdsg.ODGI*, *node_id: int*) → bool
Method to check if a node exists by ID

C++: bdsg::ODGI::has_node(long long) const –> bool

has_path (*self: bdsg.bdsg.ODGI*, *path_name: str*) → bool
Determine if a path name exists and is legal to get a path handle for.

C++: bdsg::ODGI::has_path(const std::string &) const –> bool

has_previous_step (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step has a previous step on the path, else false

C++: bdsg::ODGI::has_previous_step(const struct handlegraph::step_handle_t &) const –> bool

increment_node_ids (*self: bdsg.bdsg.ODGI*, *increment: int*) → None
Add the given value to all node IDs

C++: bdsg::ODGI::increment_node_ids(long long) –> void

is_empty (*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bool
Returns true if the given path is empty, and false otherwise

C++: bdsg::ODGI::is_empty(const struct handlegraph::path_handle_t &) const –> bool

is_path_end (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step handle is an end magic handle

C++: bdsg::ODGI::is_path_end(const struct handlegraph::step_handle_t &) const –> bool

is_path_front_end (*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*) → bool
Returns true if the step handle is a front end magic handle

C++: bdsg::ODGI::is_path_front_end(const struct handlegraph::step_handle_t &) const –> bool

max_node_id (*self: bdsg.bdsg.ODGI*) → int

Return the largest ID in the graph, or some larger number if the largest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::ODGI::max_node_id() const –> long long

min_node_id (*self: bdsg.bdsg.ODGI*) → int

Return the smallest ID in the graph, or some smaller number if the smallest ID is unavailable. Return value is unspecified if the graph is empty.

C++: bdsg::ODGI::min_node_id() const –> long long

optimize (**args*, ***kwargs*)
Overloaded function.

1. optimize(self: bdsg.bdsg.ODGI) -> None

2. optimize(self: bdsg.bdsg.ODGI, allow_id_reassignment: bool) -> None

Organize the graph for better performance and memory use

C++: bdsg::ODGI::optimize(bool) –> void

path_back(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the last step, which is arbitrary in the case of a circular path

C++: bdsg::ODGI::path_back(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_begin(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to the first step in a path. The path MUST be nonempty.

C++: bdsg::ODGI::path_begin(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_end(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious handle one past the end of the path

C++: bdsg::ODGI::path_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

path_front_end(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*) → bdsg.handlegraph.step_handle_t

Get a handle to a fictitious handle one past the start of the path

C++: bdsg::ODGI::path_front_end(const struct handlegraph::path_handle_t &) const –> struct handlegraph::step_handle_t

prepend_step(*self: bdsg.bdsg.ODGI*, *path: bdsg.handlegraph.path_handle_t*, *to_append: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Append a visit to a node to the given path. Returns a handle to the new final step on the path which is appended. Handles to prior steps on the path, and to other paths, must remain valid.

C++: bdsg::ODGI::prepend_step(const struct handlegraph::path_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

reassign_node_ids(*self: bdsg.bdsg.ODGI, get_new_id: Callable[[int], int]*) → None

Reassign all node IDs as specified by the old->new mapping function.

C++: bdsg::ODGI::reassign_node_ids(const class std::function<long long (const long long &)> &) –> void

rewrite_segment(*self: bdsg.bdsg.ODGI*, *segment_begin: bdsg.handlegraph.step_handle_t*, *segment_end: bdsg.handlegraph.step_handle_t*, *new_segment: bdsg.std.vector_handlegraph_handle_t*) → Tuple[bdsg.handlegraph.step_handle_t, bdsg.handlegraph.step_handle_t]

Replace the path range with the new segment

C++: bdsg::ODGI::rewrite_segment(const struct handlegraph::step_handle_t &, const struct handlegraph::step_handle_t &, const class std::vector<handlegraph::handle_t> &) –> struct std::pair<struct handlegraph::step_handle_t, struct handlegraph::step_handle_t>

set_circularity(*self: bdsg.bdsg.ODGI*, *path_handle: bdsg.handlegraph.path_handle_t*, *circular: bool*) → None

Set if the path is circular or not

C++: bdsg::ODGI::set_circularity(const struct handlegraph::path_handle_t &, bool) –> void

set_id_increment(*self: bdsg.bdsg.ODGI*, *min_id: int*) → None

Set a minimum id to increment the id space by, used as a hint during construction. May have no effect on a backing implementation.

C++: bdsg::ODGI::set_id_increment(const long long &) –> void

set_step(*self: bdsg.bdsg.ODGI*, *step_handle: bdsg.handlegraph.step_handle_t*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.step_handle_t

Set the step to the given handle, possibly re-linking and cleaning up if needed

C++: bdsg::ODGI::set_step(const struct handlegraph::step_handle_t &, const struct handlegraph::handle_t &) –> struct handlegraph::step_handle_t

steps_of_handle(**args*, ***kwargs*)

Overloaded function.

1. steps_of_handle(self: bdsg.bdsg.ODGI, handle: bdsg.handlegraph.handle_t) -> bdsg.std.vector_handlegraph_step_handle_t

2. steps_of_handle(self: bdsg.bdsg.ODGI, handle: bdsg.handlegraph.handle_t, match_orientation: bool) -> bdsg.std.vector_handlegraph_step_handle_t

Returns a vector of all steps of a node on paths. Optionally restricts to steps that match the handle in orientation.

C++: bdsg::ODGI::steps_of_handle(const struct handlegraph::handle_t &, bool) const –> class std::vector<handlegraph::step_handle_t>

swap_handles(*self: bdsg.bdsg.ODGI*, *a: bdsg.handlegraph.handle_t*, *b: bdsg.handlegraph.handle_t*) → None

Swap the nodes corresponding to the given handles, in the ordering used by for_each_handle when looping over the graph. Other handles to the nodes being swapped must not be invalidated. If a swap is made while for_each_handle is running, it affects the order of the handles traversed during the current traversal (so swapping an already seen handle to a later handle's position will make the seen handle be visited again and the later handle not be visited at all).

C++: bdsg::ODGI::swap_handles(const struct handlegraph::handle_t &, const struct handlegraph::handle_t &) –> void

to_gfa(**args*, ***kwargs*)

Overloaded function.

1. to_gfa(self: bdsg.bdsg.ODGI, filename: str) -> None

Convert to GFA and send to the given filename, or "-" for standard output (the default).

C++: bdsg::ODGI::to_gfa(const std::string &) const –> void

2. to_gfa(self: bdsg.bdsg.ODGI) -> str

Convert to GFA and return as a string.

C++: bdsg::ODGI::to_gfa() const –> std::string

traverse_edge_handle(*self: bdsg.bdsg.ODGI*, *edge: Tuple[bdsg.handlegraph.handle_t, bdsg.handlegraph.handle_t]*, *left: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

View the given edge handle from either inward end handle and produce the outward handle you would arrive at.

C++: bdsg::ODGI::traverse_edge_handle(const struct std::pair<struct handlegraph::handle_t, struct handlegraph::handle_t> &, const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

4.2.2 Graph Overlays

In addition to these basic implementations, there are several "overlays". These overlays are graphs that wrap other graphs, providing features not avialable in the backing graph, or otherwise transforming it.

class bdsg.bdsg.**PositionOverlay**
Bases: *bdsg.handlegraph.PathPositionHandleGraph*, *bdsg.handlegraph.ExpandingOverlayGraph*

class bdsg.bdsg.**PackedPositionOverlay**
Bases: *bdsg.handlegraph.PathPositionHandleGraph*, *bdsg.handlegraph.ExpandingOverlayGraph*

class bdsg.bdsg.**MutablePositionOverlay**
Bases: *bdsg.bdsg.PositionOverlay*, *bdsg.handlegraph.MutablePathDeletableHandleGraph*

class bdsg.bdsg.**VectorizableOverlay**
Bases: *bdsg.handlegraph.VectorizableHandleGraph*, *bdsg.handlegraph.ExpandingOverlayGraph*

class bdsg.bdsg.**PathVectorizableOverlay**
Bases: *bdsg.bdsg.VectorizableOverlay*, *bdsg.handlegraph.PathHandleGraph*

class bdsg.bdsg.**PathPositionVectorizableOverlay**
Bases: *bdsg.bdsg.PathVectorizableOverlay*, *bdsg.handlegraph.PathPositionHandleGraph*

Many of these are based on the *bdsg.handlegraph.ExpandingOverlayGraph* interface, which guarantees that the overlay does not remove any graph material, and allows handles form the backing graph and the overlay graph to be interconverted.

class bdsg.handlegraph.**ExpandingOverlayGraph**
Bases: *bdsg.handlegraph.HandleGraph*

This is the interface for a graph that represents a transformation of some underlying HandleGraph where every node in the overlay corresponds to a node in the underlying graph, but where more than one node in the overlay can map to the same underlying node.

assign(*self: bdsg.handlegraph.ExpandingOverlayGraph*, *: bdsg.handlegraph.ExpandingOverlayGraph*) → bdsg.handlegraph.ExpandingOverlayGraph
C++: handlegraph::ExpandingOverlayGraph::operator=(const class handlegraph::ExpandingOverlayGraph &) –> class handlegraph::ExpandingOverlayGraph &

get_underlying_handle(*self: bdsg.handlegraph.ExpandingOverlayGraph*, *handle: bdsg.handlegraph.handle_t*) → bdsg.handlegraph.handle_t

Returns the handle in the underlying graph that corresponds to a handle in the overlay

C++: handlegraph::ExpandingOverlayGraph::get_underlying_handle(const struct handlegraph::handle_t &) const –> struct handlegraph::handle_t

4.3 Typed Collections

Some methods, such as *bdsg.handlegraph.MutableHandleGraph.divide_handle()*, produce or consume collections of typed objects: C++ STL vectors with Python bindings. The typed collection classes are available in *bdsg.std*. They are convertible from and to Python lists via their constructors and the list constructor, respectively.

Here is an example of how to use these typed collections:

```
import bdsg
g = bdsg.bdsg.HashGraph()
h = g.create_handle("GATTACA")
v = bdsg.std.vector_unsigned_long([1, 3])
parts = g.divide_handle(h, v)
# parts is a bdsg.std.vector_handlegraph_handle_t
print(list(parts))
```

Bindings for ::std namespace

class bdsg.std.**vector_handlegraph_handle_t**

append(*self: bdsg.std.vector_handlegraph_handle_t*, *x: bdsg.handlegraph.handle_t*) → None
Add an item to the end of the list

capacity(*self: bdsg.std.vector_handlegraph_handle_t*) → int
returns the number of elements that can be held in currently allocated storage

clear(**args*, ***kwargs*)
Overloaded function.

1. clear(self: bdsg.std.vector_handlegraph_handle_t) -> None

Clear the contents

2. clear(self: bdsg.std.vector_handlegraph_handle_t) -> None

clears the contents

count(*self: bdsg.std.vector_handlegraph_handle_t*, *x: bdsg.handlegraph.handle_t*) → int
Return the number of times `x` appears in the list

empty(*self: bdsg.std.vector_handlegraph_handle_t*) → bool
checks whether the container is empty

extend(**args*, ***kwargs*)
Overloaded function.

1. extend(self: bdsg.std.vector_handlegraph_handle_t, L: bdsg.std.vector_handlegraph_handle_t) -> None

Extend the list by appending all the items in the given list

2. extend(self: bdsg.std.vector_handlegraph_handle_t, L: iterable) -> None

Extend the list by appending all the items in the given list

insert(*self: bdsg.std.vector_handlegraph_handle_t*, *i: int*, *x: bdsg.handlegraph.handle_t*) → None
Insert an item at a given position.

max_size(*self: bdsg.std.vector_handlegraph_handle_t*) → int
returns the maximum possible number of elements

pop(**args*, ***kwargs*)
Overloaded function.

1. pop(self: bdsg.std.vector_handlegraph_handle_t) -> bdsg.handlegraph.handle_t

Remove and return the last item

2. pop(self: bdsg.std.vector_handlegraph_handle_t, i: int) -> bdsg.handlegraph.handle_t

Remove and return the item at index `i`

remove (*self: bdsg.std.vector_handlegraph_handle_t*, *x: bdsg.handlegraph.handle_t*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

reserve (*self: bdsg.std.vector_handlegraph_handle_t*, *arg0: int*) → None
reserves storage

shrink_to_fit (*self: bdsg.std.vector_handlegraph_handle_t*) → None
reduces memory usage by freeing unused memory

swap (*self: bdsg.std.vector_handlegraph_handle_t*, *arg0: bdsg.std.vector_handlegraph_handle_t*) → None
swaps the contents

class bdsg.std.**vector_handlegraph_path_handle_t**

append (*self: bdsg.std.vector_handlegraph_path_handle_t*, *x: bdsg.handlegraph.path_handle_t*) → None
Add an item to the end of the list

capacity (*self: bdsg.std.vector_handlegraph_path_handle_t*) → int
returns the number of elements that can be held in currently allocated storage

clear (**args*, ***kwargs*)
Overloaded function.

1. clear(self: bdsg.std.vector_handlegraph_path_handle_t) -> None

Clear the contents

2. clear(self: bdsg.std.vector_handlegraph_path_handle_t) -> None

clears the contents

count (*self: bdsg.std.vector_handlegraph_path_handle_t*, *x: bdsg.handlegraph.path_handle_t*) → int
Return the number of times x appears in the list

empty (*self: bdsg.std.vector_handlegraph_path_handle_t*) → bool
checks whether the container is empty

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: bdsg.std.vector_handlegraph_path_handle_t, L: bdsg.std.vector_handlegraph_path_handle_t) -> None

Extend the list by appending all the items in the given list

2. extend(self: bdsg.std.vector_handlegraph_path_handle_t, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: bdsg.std.vector_handlegraph_path_handle_t*, *i: int*, *x: bdsg.handlegraph.path_handle_t*) → None
Insert an item at a given position.

max_size (*self: bdsg.std.vector_handlegraph_path_handle_t*) → int
returns the maximum possible number of elements

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: bdsg.std.vector_handlegraph_path_handle_t) -> bdsg.handlegraph.path_handle_t

Remove and return the last item

2. pop(self: bdsg.std.vector_handlegraph_path_handle_t, i: int) -> bdsg.handlegraph.path_handle_t

Remove and return the item at index `i`

remove (*self: bdsg.std.vector_handlegraph_path_handle_t, x: bdsg.handlegraph.path_handle_t*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

reserve (*self: bdsg.std.vector_handlegraph_path_handle_t, arg0: int*) → None
reserves storage

shrink_to_fit (*self: bdsg.std.vector_handlegraph_path_handle_t*) → None
reduces memory usage by freeing unused memory

swap (*self: bdsg.std.vector_handlegraph_path_handle_t, arg0: bdsg.std.vector_handlegraph_path_handle_t*) → None
swaps the contents

class bdsg.std.**vector_handlegraph_step_handle_t**

append (*self: bdsg.std.vector_handlegraph_step_handle_t, x: bdsg.handlegraph.step_handle_t*) → None
Add an item to the end of the list

capacity (*self: bdsg.std.vector_handlegraph_step_handle_t*) → int
returns the number of elements that can be held in currently allocated storage

clear (**args, **kwargs*)
Overloaded function.

1. clear(self: bdsg.std.vector_handlegraph_step_handle_t) -> None

Clear the contents

2. clear(self: bdsg.std.vector_handlegraph_step_handle_t) -> None

clears the contents

count (*self: bdsg.std.vector_handlegraph_step_handle_t, x: bdsg.handlegraph.step_handle_t*) → int
Return the number of times `x` appears in the list

empty (*self: bdsg.std.vector_handlegraph_step_handle_t*) → bool
checks whether the container is empty

extend (**args, **kwargs*)
Overloaded function.

1. extend(self: bdsg.std.vector_handlegraph_step_handle_t, L: bdsg.std.vector_handlegraph_step_handle_t) -> None

Extend the list by appending all the items in the given list

2. extend(self: bdsg.std.vector_handlegraph_step_handle_t, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: bdsg.std.vector_handlegraph_step_handle_t, i: int, x: bdsg.handlegraph.step_handle_t*) → None
Insert an item at a given position.

max_size (*self: bdsg.std.vector_handlegraph_step_handle_t*) → int
returns the maximum possible number of elements

pop (**args, **kwargs*)
Overloaded function.

1. pop(self: bdsg.std.vector_handlegraph_step_handle_t) -> bdsg.handlegraph.step_handle_t

Remove and return the last item

2. pop(self: bdsg.std.vector_handlegraph_step_handle_t, i: int) -> bdsg.handlegraph.step_handle_t

Remove and return the item at index `i`

remove (*self: bdsg.std.vector_handlegraph_step_handle_t*, *x: bdsg.handlegraph.step_handle_t*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

reserve (*self: bdsg.std.vector_handlegraph_step_handle_t*, *arg0: int*) → None
reserves storage

shrink_to_fit (*self: bdsg.std.vector_handlegraph_step_handle_t*) → None
reduces memory usage by freeing unused memory

swap (*self: bdsg.std.vector_handlegraph_step_handle_t*, *arg0: bdsg.std.vector_handlegraph_step_handle_t*) → None
swaps the contents

class bdsg.std.**vector_unsigned_long**

append (*self: bdsg.std.vector_unsigned_long*, *x: int*) → None
Add an item to the end of the list

capacity (*self: bdsg.std.vector_unsigned_long*) → int
returns the number of elements that can be held in currently allocated storage

clear (**args*, ***kwargs*)
Overloaded function.

1. clear(self: bdsg.std.vector_unsigned_long) -> None

Clear the contents

2. clear(self: bdsg.std.vector_unsigned_long) -> None

clears the contents

count (*self: bdsg.std.vector_unsigned_long*, *x: int*) → int
Return the number of times `x` appears in the list

empty (*self: bdsg.std.vector_unsigned_long*) → bool
checks whether the container is empty

extend (**args*, ***kwargs*)
Overloaded function.

1. extend(self: bdsg.std.vector_unsigned_long, L: bdsg.std.vector_unsigned_long) -> None

Extend the list by appending all the items in the given list

2. extend(self: bdsg.std.vector_unsigned_long, L: iterable) -> None

Extend the list by appending all the items in the given list

insert (*self: bdsg.std.vector_unsigned_long*, *i: int*, *x: int*) → None
Insert an item at a given position.

max_size (*self: bdsg.std.vector_unsigned_long*) → int
returns the maximum possible number of elements

pop (**args*, ***kwargs*)
Overloaded function.

1. pop(self: bdsg.std.vector_unsigned_long) -> int

Remove and return the last item

2. pop(self: bdsg.std.vector_unsigned_long, i: int) -> int

Remove and return the item at index `i`

remove (*self: bdsg.std.vector_unsigned_long*, *x: int*) → None
Remove the first item from the list whose value is x. It is an error if there is no such item.

reserve (*self: bdsg.std.vector_unsigned_long*, *arg0: int*) → None
reserves storage

shrink_to_fit (*self: bdsg.std.vector_unsigned_long*) → None
reduces memory usage by freeing unused memory

swap (*self: bdsg.std.vector_unsigned_long*, *arg0: bdsg.std.vector_unsigned_long*) → None
swaps the contents

CHAPTER 5

C++ API

Being written in C++, `libbdsg` and `libhandlegraph` offer a C++ API.

5.1 Handle Graph API

The `handlegraph` namespace defines the handle graph interface.

5.1.1 Handles

The namespace contains definitions for different types of handles. THese are references to graph elements. A basic `handlegraph::handle_t` is a reference to a strand or orientation of a node in the graph.

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

5.1.2 Graph Interfaces

The `handlegraph` namespace also defines a hierarchy of interfaces for graph implementations that provide different levels of features.

HandleGraph

The most basic is the `handlegraph::HandleGraph`, a completely immutable, unannotated graph.

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

PathHandleGraph

On top of this, there is the `handlegraph::PathHandleGraph`, which allows for embedded, named paths in the graph.

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Mutable and Deletable Interfaces

Then for each there are versions where the underlying graph is "mutable" (meaning that material can be added to it and nodes can be split) and "deletable" (meaning that nodes and edges can actually be removed from the graph), and for `handlegraph::PathHandleGraph` there are versions where the paths can be altered.

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

> **Warning:** doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Note that there is no `handlegraph::PathMutableHandleGraph` or `handlegraph::PathDeletableHandleGraph`; it does not make sense for the paths to be static while the graph can be modified.

Position and Ordering Interfaces

For paths, there is also the `handlegraph::PathPositionHandleGraph` which provides efficient random access by or lookup of base offset along each embedded path. Additionally, there is `handlegraph::VectorizableHandleGraph` which provides the same operations for a linearization of all of the graph's bases. There is also a `handlegraph::RankedHandleGraph` interface, which provides an ordering, though not necessarily a base-level linearization, of nodes and edges.

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Algorithm implementers are encouraged to take the least capable graph type necessary for their algorithm to function.

SerializableHandleGraph

Orthogonal to the mutability and paths hierarchy, there is a `handlegraph::SerializableHandleGraph` interface that is implemented by graphs that can be saved to and loaded from disk. The C++ API supports saving to and loading from C++ streams, but the Python API provides only the ability to save to or load from filenames.

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

5.2 libbdsg Handle Graph Implementations

The `bdsg` namespace provides useful implementations of the Handle Graph API.

5.2.1 Full Graph Implementations

There are three full graph implementations in the module: `bdsg::PackedGraph`, `bdsg::HashGraph`, and `bdsg::ODGI`.

PackedGraph

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

HashGraph

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

ODGI

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

5.2.2 Graph Overlays

In addition to these basic implementations, there are several "overlays". These overlays are graphs that wrap other graphs, providing features not avialable in the backing graph, or otherwise transforming it.

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Many of these are based on the `handlegraph::ExpandingOverlayGraph` interface, which guarantees that the overlay does not remove any graph material, and allows handles form the backing graph and the overlay graph to be interconverted.

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Graph Overlay Helpers

From C++, some types are available to allow code to take an input of a more general type (say, a `bdsg::HandleGraph`) and get a view of it as a more specific type (such as a `bdsg::VectorizableHandleGraph`), using an overlay to bridge the gap if the backing graph implementation does not itself support the requested feature. For each pf these "overlay helpers", you instantiate the object (which allocates storage), use the `apply()` method to pass it a pointer to the backing graph and get a pointer to a graph of the requested type, and then use the `get()` method later if you need to get the requested-type graph pointer again.

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

All these overlay helpers are really instantiations of a couple of templates:

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/bdsg/checkouts/improve-docs/bdsg/build/doxygen/xml/index.xml

CHAPTER 6

Index

- genindex
- search

Python Module Index

b

Index

D

E

F

G

H

I

M

N

O

P

R

S

T

V