vg

Release

Dec 01, 2017

Contents

CHAPTER 1

vg Internal API Reference

Below is an index of all classes, files, and namespaces in `vg`, in alphabetical order.

Useful starting points include *vg::Node*, *vg::Edge*, *vg::Path*, and *vg::Graph*, which define the Protobuf graph data model, and *vg::VG*, which is the main graph class with all the useful graph methods on it.

struct *#include <genotyper.hpp>*

Public Functions

`vg::Genotyper::Affinity::`**`Affinity`**`()`

`vg::Genotyper::Affinity::`**`Affinity`**`(`double *affinity*, bool *is_reverse*`)`

Public Members

bool `vg::Genotyper::Affinity::`**`consistent`**

double `vg::Genotyper::Affinity::`**`affinity`**

bool `vg::Genotyper::Affinity::`**`is_reverse`**

double `vg::Genotyper::Affinity::`**`score`**

double `vg::Genotyper::Affinity::`**`likelihood_ln`**

class *#include <gssw_aligner.hpp>* An ordinary aligner. Inherits from *vg::BaseAligner*

Public Functions

`Aligner::`**`Aligner`**`(`int8_t *_match* = default_match, int8_t *_mismatch* = default_mismatch, int8_t *_gap_open* = default_gap_open, int8_t *_gap_extension* = default_gap_extension, int8_t *_full_length_bonus* = default_full_length_bonus, double *_gc_content* = default_gc_content`)`

`vg::Aligner::`**`~Aligner`**`(`void`)`

void Aligner::**align**(Alignment &*alignment*, Graph &*g*, bool *traceback_aln*, bool *print_score_matrices*)
Store optimal local alignment against a graph in the *Alignment* object. Gives the full length bonus separately on each end of the alignment. Assumes that graph is topologically sorted by node index.

void Aligner::**align_pinned**(Alignment &*alignment*, Graph &*g*, bool *pin_left*)

void Aligner::**align_pinned_multi**(Alignment &*alignment*, vector<Alignment> &*alt_alignments*, Graph &*g*, bool *pin_left*, int32_t *max_alt_alns*)

void Aligner::**align_global_banded**(Alignment &*alignment*, Graph &*g*, int32_t *band_padding* = 0, bool *permissive_banding* = true)

void Aligner::**align_global_banded_multi**(Alignment &*alignment*, vector<Alignment> &*alt_alignments*, Graph &*g*, int32_t *max_alt_alns*, int32_t *band_padding* = 0, bool *permissive_banding* = true)

int32_t Aligner::**score_exact_match**(**const** Alignment &*aln*, size_t *read_offset*, size_t *length*)
Compute the score of an exact match in the given alignment, from the given offset, of the given length.

int32_t Aligner::**score_exact_match**(**const** string &*sequence*, **const** string &*base_quality*) **const**
Compute the score of an exact match of the given sequence with the given qualities. Qualities may be ignored by some implementations.

int32_t Aligner::**score_exact_match**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*, string::const_iterator *base_qual_begin*) **const**
Compute the score of an exact match of the given range of sequence with the given qualities. Qualities may be ignored by some implementations.

int32_t Aligner::**score_exact_match**(**const** string &*sequence*) **const**

int32_t Aligner::**score_exact_match**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*) **const**

Private Functions

void Aligner::**align_internal**(Alignment &*alignment*, vector<Alignment> **multi_alignments*, Graph &*g*, bool *pinned*, bool *pin_left*, int32_t *max_alt_alns*, bool *traceback_aln*, bool *print_score_matrices*)

struct Alignments link query strings, such as other genomes or reads, to *Paths*. **Public Members**

string vg::Alignment::**sequence**
The sequence that has been aligned.

Path vg::Alignment::**path**
The *Path* that the sequence follows in the graph it has been aligned to, containing the *Edit*s that modify the graph to produce the sequence.

string vg::Alignment::**name**
The name of the sequence that has been aligned. Similar to read name in BAM.

bytes vg::Alignment::**quality**
The quality scores for the sequence, as values on a 0-255 scale.

int32 vg::Alignment::**mapping_quality**
The mapping quality score for the alignment, in Phreds.

int32 vg::Alignment::**score**
The score for the alignment, in points.

int32 vg::Alignment::**query_position**
The offset in the query at which this *Alignment* occurs.

string vg::Alignment::**sample_name**
The name of the sample that produced the aligned read.

string vg::Alignment::**read_group**
The name of the read group to which the aligned read belongs.

Alignment vg::Alignment::**fragment_prev**
The previous *Alignment* in the fragment. Contains just enough information to locate the full *Alignment*; e.g. contains an *Alignment* with only a name, or only a graph mapping position.

Alignment vg::Alignment::**fragment_next**
Similarly, the next *Alignment* in the fragment.

bool vg::Alignment::**is_secondary**
Flag marking the *Alignment* as secondary. All but one maximal-scoring alignment of a given read in a GAM file must be secondary.

double vg::Alignment::**identity**
Portion of aligned bases that are perfect matches, or 0 if no bases are aligned.

repeated<*Path*> vg::Alignment::**fragment**
An estimate of the length of the fragment, if this *Alignment* is paired.

repeated<*Locus*> vg::Alignment::**locus**
The loci that this alignment supports. TODO: get rid of this, we have annotations in our data model again.

repeated<*Position*> vg::Alignment::**refpos**
Position of the alignment in reference paths embedded in graph.

bool vg::Alignment::**read_paired**
SAMTools-style flags.

bool vg::Alignment::**read_mapped**

bool vg::Alignment::**mate_unmapped**

bool vg::Alignment::**read_on_reverse_strand**

bool vg::Alignment::**mate_on_reverse_strand**

bool vg::Alignment::**soft_clipped**

bool vg::Alignment::**discordant_insert_size**

double vg::Alignment::**uniqueness**
The fraction of bases in the alignment that are covered by MEMs with <=1 total hits in the graph.

double vg::Alignment::**correct**
Correctness metric 1 = perfectly aligned to truth, 0 = not overlapping true alignment.

repeated<int32> vg::Alignment::**secondary_score**
The ordered list of scores of secondary mappings.

double vg::Alignment::**fragment_score**
Score under the given fragment model, assume higher is better.

bool vg::Alignment::**mate_mapped_to_disjoint_subgraph**

string vg::Alignment::**fragment_length_distribution**
The fragment length distribution under which a paired-end alignment was aligned.

class *#include <mapper.hpp>*

Public Functions

vg::AlignmentChainModel::**AlignmentChainModel**(vector<vector<*Alignment*>> &*bands*, *Mapper* *mapper, **const** function<double>() **const** *Alignment*&, **const** *Alignment*&, **const** map<string, vector<pair<size_t, bool>>>&, **const** map<string, vector<pair<size_t, bool>>>& > &*transition_weight*, int *vertex_band_width* = 10, int *position_depth* = 1, int *max_connections* = 30

void vg::AlignmentChainModel::**score**(**const** set<*AlignmentChainModelVertex* *> &*exclude*)

AlignmentChainModelVertex *vg::AlignmentChainModel::**max_vertex**(void)

vector<*Alignment*> vg::AlignmentChainModel::**traceback**(**const** *Alignment* &*read*, int *alt_alns*, bool *paired*, bool *debug*)

void vg::AlignmentChainModel::**display**(ostream &*out*)

void vg::AlignmentChainModel::**clear_scores**(void)

Public Members

vector<*AlignmentChainModelVertex*> vg::AlignmentChainModel::**model**

map<string, map<int64_t, vector<vector<*AlignmentChainModelVertex*>::iterator>>> vg::AlignmentChainModel::**posi**

set<vector<*AlignmentChainModelVertex*>::iterator> vg::AlignmentChainModel::**redundant_vertexes**

vector<*Alignment*> vg::AlignmentChainModel::**unaligned_bands**

class *#include <mapper.hpp>*

Public Functions

vg::AlignmentChainModelVertex::**AlignmentChainModelVertex**(void)

vg::AlignmentChainModelVertex::**AlignmentChainModelVertex**(**const** *AlignmentChainModelVertex*&)

vg::AlignmentChainModelVertex::**AlignmentChainModelVertex**(*AlignmentChainModelVertex*&&)

AlignmentChainModelVertex &vg::AlignmentChainModelVertex::**operator=**(**const** *AlignmentChainModelVertex*&)

AlignmentChainModelVertex &vg::AlignmentChainModelVertex::**operator=**(*AlignmentChainModelVertex*&&)

virtual vg::AlignmentChainModelVertex::**~AlignmentChainModelVertex**()

Public Members

Alignment *vg::AlignmentChainModelVertex::**aln**

vector<pair<*AlignmentChainModelVertex* *, double>> vg::AlignmentChainModelVertex::**next_cost**

vector<pair<*AlignmentChainModelVertex* *, double>> vg::AlignmentChainModelVertex::**prev_cost**

double vg::AlignmentChainModelVertex::**weight**

double vg::AlignmentChainModelVertex::**score**

map<string, vector<pair<size_t, bool>>> vg::AlignmentChainModelVertex::**positions**

int vg::AlignmentChainModelVertex::**band_begin**

int vg::AlignmentChainModelVertex::**band_idx**

AlignmentChainModelVertex *vg::AlignmentChainModelVertex::**prev**
class Public Functions

bool AlnSorter::**mycomparison**(**const** bool &*reverse* = false)

bool AlnSorter::**operator()**(**const** Alignment &*a_one*, **const** Alignment &*a_two*) **const**

Private Members

bool AlnSorter::**invert**
class *#include <banded_global_aligner.hpp>* Specialized linked list stack that finds and keeps track of the top-scoring non-optimal alignments. Sub-optimal alignments are represented by the locations where their traceback deviates from the optimal traceback, with other steps of the traceback implicitly following the optimal path. New tracebacks can be discovered while performing any of the tracebacks and added to the stack. Public Functions

BandedGlobalAligner::AltTracebackStack::**AltTracebackStack**(int64_t *max_multi_alns*, int32_t *empty_score*, unordered_set<BAMatrix *> &*source_node_matrices*, unordered_set<BAMatrix *> &*sink_node_matrices*, int8_t *gap_open*, int8_t *gap_extend*, IntType *min_inf*)

BandedGlobalAligner::AltTracebackStack::**~AltTracebackStack**()

void BandedGlobalAligner::AltTracebackStack::**get_alignment_start**(int64_t &*node_id*, matrix_t &*matrix*)
Get the start position of the current alignment and advance deflection pointer to the first deflection.

bool BandedGlobalAligner::AltTracebackStack::**has_next**()
Are there any more alternate alignments in the stack?

void BandedGlobalAligner::AltTracebackStack::**next_traceback_alignment**()
Advance to the next alternate trac.

bool BandedGlobalAligner::AltTracebackStack::**next_is_empty**()
Is the next highest scoring alignment an empty path with no DP matrices?

void BandedGlobalAligner::AltTracebackStack::**next_empty_alignment**(Alignment &*alignment*)
Get the next empty path alignment.

void BandedGlobalAligner::AltTracebackStack::**propose_deflection**(**const** IntType *score*, **const** int64_t *from_node_id*, **const** int64_t *row_idx*, **const** int64_t *col_idx*, **const** int64_t *to_node_id*, **const** matrix_t *to_matrix*)
Check if a deflection from the current traceback is better than any alignments currently in the stack and if so insert it in the correct position

IntType BandedGlobalAligner::AltTracebackStack::**current_traceback_score**()
Score of the current traceback.

const list<int64_t> &BandedGlobalAligner::AltTracebackStack::**current_empty_prefix**()
The prefix of the current traceback that consists of only empty nodes.

bool BandedGlobalAligner::AltTracebackStack::**at_next_deflection**(int64_t *node_id*, int64_t *row_idx*, int64_t *col_idx*)
Are these the coordinates of the next deflection?

BandedGlobalAligner<IntType>::matrix_t BandedGlobalAligner::AltTracebackStack::**deflect_to_matrix**
Get the matrix to deflect to and advance to the next deflection.

BandedGlobalAligner<IntType>::matrix_t BandedGlobalAligner::AltTracebackStack::**deflect_to_matrix**

Get the matrix and the node id to deflect to (for use at a node boundary)

Private Functions

void BandedGlobalAligner::AltTracebackStack::**insert_traceback**(**const** vector<Deflection> &*traceback_prefix*, **const** IntType *score*, **const** int64_t *from_node_id*, **const** int64_t *row_idx*, **const** int64_t *col_idx*, **const** int64_t *to_node_id*, **const** matrix_t *to_matrix*, **const** list<int64_t> &*empty_node_prefix*)

Internal method for propose_deflection.

Private Members

template<>
int64_t vg::BandedGlobalAligner<IntType>::AltTracebackStack::**max_multi_alns**

Maximum number of alternate alignments to keep track of (including the optimal alignment)

template<>
list<tuple<vector<Deflection>, IntType, list<int64_t>>> vg::BandedGlobalAligner<IntType>::AltTracebackSt

List of tuples that contain the scores of alternate alignments, the places where their traceback deviates from the optimum (Deflections), and all empty nodes that occur at the suffix of the traceback (if any). Maintains an invariant where stack is sorted in descending score order.

template<>
list<list<int64_t>> vg::BandedGlobalAligner<IntType>::AltTracebackStack::**empty_full_paths**

All of the paths through the graph that take only empty nodes.

template<>
int32_t vg::BandedGlobalAligner<IntType>::AltTracebackStack::**empty_score**

template<>
list<tuple<vector<Deflection>, IntType, list<int64_t>>>::iterator vg::BandedGlobalAligner<IntType>::AltTraceb

Pointer to the traceback directions for the alignment we are currently tracing back.

template<>
vector<Deflection>::iterator vg::BandedGlobalAligner<IntType>::AltTracebackStack::**curr_deflxn**

Pointer to the next deviation from the optimal traceback we will take.

struct *#include <genome_state.hpp>* Inherits from *vg::GenomeStateCommand* **Public Functions**

GenomeStateCommand *vg::AppendHaplotypeCommand::**execute**(*GenomeState* **const** *&state*)
Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual vg::AppendHaplotypeCommand::**~AppendHaplotypeCommand**()

Public Members

vector<*handle_t*> vg::AppendHaplotypeCommand::**haplotype**
We just feed in a full traversal from one end of a telomere pair to the other. Must start and end on the boundary nodes of telomere snarls. TODO: unary telomeres will work strangely. Internally the *GenomeState* has to work out how to divide this into snarls.

struct *#include <genotypekit.hpp>* General interface for an augmented graph. This is a graph that was constructed by adding some read information to an original ("base") graph. We preserve mappings back to the base graph via translations. Augmented graphs can be made using edit (such as in vg mod -i) or pileup. Todo : further abstract to handle graph interface Subclassed by *vg::SupportAugmentedGraph* **Public Functions**

pair<**const** *Edge* *, bool> vg::AugmentedGraph::**base_edge**(**const** *Edge* *augmented_edge*)

bool vg::AugmentedGraph::**is_novel_node**(**const** *Node* *augmented_node*)

bool vg::AugmentedGraph::**is_novel_edge**(**const** *Edge* *augmented_edge*)

void vg::AugmentedGraph::**clear**()
Clear the contents.

void vg::AugmentedGraph::**augment_from_alignment_edits**(vector<*Alignment*> *&alignments*, bool *unique_names* = true, bool *leave_edits* = false)
Construct an augmented graph using edit() on a set of alignments

void vg::AugmentedGraph::**load_translations**(istream *&in_file*)
Load the translations from a file

void vg::AugmentedGraph::**write_translations**(ostream *&out_file*)
Write the translations to a file

Public Members

VG vg::AugmentedGraph::**graph**

VG *vg::AugmentedGraph::**base_graph**

Translator vg::AugmentedGraph::**translator**

class *#include <banded_global_aligner.hpp>* Translates a traceback path into a *Path* object and stores it in an *Alignment* object **Public Functions**

BandedGlobalAligner::BABuilder::**BABuilder**(Alignment *&alignment*)

BandedGlobalAligner::BABuilder::**~BABuilder**()

void BandedGlobalAligner::BABuilder::**update_state**(matrix_t *matrix*, Node **node*, int64_t *read_idx*, int64_t *node_idx*, bool *empty_node_seq* = false)

Add next step in traceback.

void BandedGlobalAligner::BABuilder::**finalize_alignment**(**const** list<int64_t> &*empty_prefix*)

Call after concluding traceback to finish adding edits to alignment.

Private Functions

void BandedGlobalAligner::BABuilder::**finish_current_edit**()

void BandedGlobalAligner::BABuilder::**finish_current_node**()

Private Members

template<>
Alignment &vg::BandedGlobalAligner<IntType>::BABuilder::**alignment**

template<>
list<*Mapping*> vg::BandedGlobalAligner<IntType>::BABuilder::**node_mappings**

template<>
list<*Edit*> vg::BandedGlobalAligner<IntType>::BABuilder::**mapping_edits**

template<>
matrix_t vg::BandedGlobalAligner<IntType>::BABuilder::**matrix_state**

template<>
bool vg::BandedGlobalAligner<IntType>::BABuilder::**matching**

template<>
Node *vg::BandedGlobalAligner<IntType>::BABuilder::**current_node**

template<>
int64_t vg::BandedGlobalAligner<IntType>::BABuilder::**edit_length**

template<>
int64_t vg::BandedGlobalAligner<IntType>::BABuilder::**edit_read_end_idx**
class *#include <banded_global_aligner.hpp>* Represents the band from the DP matrix for one node in the graph **Public Functions**

BandedGlobalAligner::BAMatrix::**BAMatrix**(Alignment &*alignment*, Node **node*, int64_t *top_diag*, int64_t *bottom_diag*, *BAMatrix* ***seeds*, int64_t *num_seeds*, int64_t *cumulative_seq_len*)

BandedGlobalAligner::BAMatrix::**~BAMatrix**()

void BandedGlobalAligner::BAMatrix::**fill_matrix**(int8_t **score_mat*, int8_t **nt_table*, int8_t *gap_open*, int8_t *gap_extend*, bool *qual_adjusted*, IntType *min_inf*)

Use DP to fill the band with alignment scores.

void BandedGlobalAligner::BAMatrix::**traceback**(BABuilder &*builder*, AltTracebackStack &*traceback_stack*, matrix_t *start_mat*, int8_t **score_mat*, int8_t **nt_table*, int8_t *gap_open*, int8_t *gap_extend*, bool *qual_adjusted*, IntType *min_inf*)

Traceback through the band after using DP to fill it.

void BandedGlobalAligner::BAMatrix::**print_full_matrices**()

Debugging function.

void BandedGlobalAligner::BAMatrix::**print_rectangularized_bands**()

Debugging function.

Private Functions

void BandedGlobalAligner::BAMatrix::**traceback_internal**(BABuilder &*builder*, AltTracebackStack &*traceback_stack*, int64_t *start_row*, int64_t *start_col*, matrix_t *start_mat*, bool *in_lead_gap*, int8_t **score_mat*, int8_t **nt_table*, int8_t *gap_open*, int8_t *gap_extend*, bool *qual_adjusted*, IntType *min_inf*)

void BandedGlobalAligner::BAMatrix::**print_matrix**(matrix_t *which_mat*)

Debugging function.

void BandedGlobalAligner::BAMatrix::**print_band**(matrix_t *which_mat*)

Debugging function.

Private Members

template<>
int64_t vg::BandedGlobalAligner<IntType>::BAMatrix::**top_diag**

The diagonals in the DP matrix that the band passes through with the bottom index inclusive.

template<>
int64_t vg::BandedGlobalAligner<IntType>::BAMatrix::**bottom_diag**

template<>
Node *vg::BandedGlobalAligner<IntType>::BAMatrix::**node**

template<>
Alignment &vg::BandedGlobalAligner<IntType>::BAMatrix::**alignment**

template<>
int64_t vg::BandedGlobalAligner<IntType>::BAMatrix::**cumulative_seq_len**
Length of shortest sequence leading to matrix from a source node.

template<>
BAMatrix **vg::BandedGlobalAligner<IntType>::BAMatrix::**seeds**
Matrices for nodes with edges into this node.

template<>
int64_t vg::BandedGlobalAligner<IntType>::BAMatrix::**num_seeds**

template<>
IntType *vg::BandedGlobalAligner<IntType>::BAMatrix::**match**
DP matrix.

template<>
IntType *vg::BandedGlobalAligner<IntType>::BAMatrix::**insert_col**
DP matrix.

template<>
IntType *vg::BandedGlobalAligner<IntType>::BAMatrix::**insert_row**
DP matrix.

Friends

friend vg::BandedGlobalAligner::BAMatrix::BABuilder

friend vg::BandedGlobalAligner::BAMatrix::AltTracebackStack

template <**class** IntType>
class *#include <banded_global_aligner.hpp>* The outward-facing interface for banded global graph alignment. It computes optimal alignment of a DNA sequence to a DAG with POA. The alignment will start at any source node in the graph and end at any sink node. It is also restricted to falling within a certain diagonal band from the start node. Any signed integer type can be used for the dynamic programming matrices, but there are no checks for overflow.

Public Functions

BandedGlobalAligner::**BandedGlobalAligner**(Alignment &*alignment*, Graph &*g*, int64_t *band_padding*, bool *permissive_banding* = false, bool *adjust_for_base_quality* = false)
Initializes banded alignment

Args: alignment empty alignment with a sequence (and possibly base qualities) g graph to align to band_padding width to expand band by permissive_banding expand band, not necessarily symmetrically, to allow all node paths adjust_for_base_quality perform base quality adjusted alignment (see *QualAdjAligner*)

BandedGlobalAligner::**BandedGlobalAligner**(Alignment &*alignment*, Graph &*g*, vector<Alignment> &*alt_alignments*, int64_t *max_multi_alns*, int64_t *band_padding*, bool *permissive_banding* = false, bool *adjust_for_base_quality* = false)

Initializes banded multi-alignment, which computes the top scoring alternate alignments in addition to the optimal alignment

Args: alignment empty alignment with a sequence (and possibly base qualities) g graph to align to alt_alignments an empty vector to store alternate alignments in, the first element will be a copy of the primary alignment max_multi_alns the maximum number of alternate alignments (including the primary) band_padding width to expand band by permissive_banding expand band, not necessarily symmetrically, to allow all node paths adjust_for_base_quality perform base quality adjusted alignment (see *QualAdjAligner*)

`BandedGlobalAligner::~BandedGlobalAligner()`

void `BandedGlobalAligner::align`(int8_t **score_mat*, int8_t **nt_table*, int8_t *gap_open*, int8_t *gap_extend*)

Adds path and score to the alignment object given in the constructor. If a multi-alignment vector was also supplied, fills the vector with the top scoring alignments as well.

Note: the score arguments are not <IntType> so that they can interact easily with the *Aligner*

Args: score_mat matrix of match/mismatch scores from *Aligner* (if performing base quality adjusted alignment, use *QualAdjAligner*'s adjusted score matrix) nt_table table of indices by DNA char from *Aligner* gap_open gap open penalty from Algner (if performing base quality adjusted alignment, use *QualAdjAligner*'s scaled penalty) gap_extend gap extension penalty from Algner (if performing base quality adjusted alignment, use *QualAdjAligner*'s scaled penalty)

Private Types

enum type `vg::BandedGlobalAligner::matrix_t`

Matrices used in Smith-Waterman-Gotoh alignment algorithm.

Values:

Private Functions

`BandedGlobalAligner::BandedGlobalAligner`(Alignment &*alignment*, Graph &*g*, vector<Alignment> **alt_alignments*, int64_t *max_multi_alns*, int64_t *band_padding*, bool *permissive_banding* = false, bool *adjust_for_base_quality* = false)

Internal constructor that the public constructors funnel into.

void `BandedGlobalAligner::traceback`(int8_t **score_mat*, int8_t **nt_table*, int8_t *gap_open*, int8_t *gap_extend*, IntType *min_inf*)

Traceback through dynamic programming matrices to compute alignment.

void `BandedGlobalAligner::graph_edge_lists`(Graph &*g*, bool *outgoing_edges*, vector<vector<int64_t>> &*out_edge_list*)

Constructor helper function: converts *Graph* object into adjacency list representation.

void `BandedGlobalAligner::topological_sort`(Graph &*g*, vector<vector<int64_t>> &*node_edges_out*, vector<Node *> &*out_topological_order*)

Constructor helper function: compute topoligical ordering.

void BandedGlobalAligner::**path_lengths_to_sinks**(**const** string &*read*, vector<vector<int64_t>> &*node_edges_in*, vector<int64_t> &*shortest_path_to_sink*, vector<int64_t> &*longest_path_to_sink*)

Constructor helper function: compute the longest and shortest path to a sink for each node.

void BandedGlobalAligner::**find_banded_paths**(**const** string &*read*, bool *permissive_banding*, vector<vector<int64_t>> &*node_edges_in*, vector<vector<int64_t>> &*node_edges_out*, int64_t *band_padding*, vector<bool> &*node_masked*, vector<pair<int64_t, int64_t>> &*band_ends*)

Constructor helper function: compute which diagonals the bands cover on each node's matrix.

void BandedGlobalAligner::**shortest_seq_paths**(vector<vector<int64_t>> &*node_edges_out*, unordered_set<Node *> &*source_nodes*, vector<int64_t> &*seq_lens_out*)

Constructor helper function: compute the shortest path from a source to each node.

Private Members

Alignment &vg::BandedGlobalAligner::**alignment**

The primary alignment.

vector<*Alignment*> *vg::BandedGlobalAligner::**alt_alignments**

Vector for alternate alignments, or null if not making any.

int64_t vg::BandedGlobalAligner::**max_multi_alns**

Number of alignments to compute, including the optimal alignment.

bool vg::BandedGlobalAligner::**adjust_for_base_quality**

Use base quality adjusted scoring for alignments?

vector<*BAMatrix* *> vg::BandedGlobalAligner::**banded_matrices**

Dynamic programming matrices for each node.

unordered_map<int64_t, int64_t> vg::BandedGlobalAligner::**node_id_to_idx**

Map from node IDs to the index used in internal vectors.

vector<*Node* *> vg::BandedGlobalAligner::**topological_order**

A topological ordering of the nodes.

unordered_set<*Node* *> vg::BandedGlobalAligner::**source_nodes**

Source nodes in the graph.

unordered_set<*Node* *> vg::BandedGlobalAligner::**sink_nodes**

Sink nodes in the graph.

class *#include <gssw_aligner.hpp>* The interface that any *Aligner* should implement, with some default implementations. Subclassed by *vg::Aligner*, *vg::QualAdjAligner* **Public Functions**

double BaseAligner::**max_possible_mapping_quality**(int *length*)

double BaseAligner::**estimate_max_possible_mapping_quality**(int *length*, double *min_diffs*, double *next_min_diffs*)

virtual void vg::BaseAligner::**align**(*Alignment* &*alignment*, *Graph* &*g*, bool *traceback_aln*, bool *print_score_matrices*)
= 0Store optimal local alignment against a graph in the *Alignment* object. Gives the full length bonus separately on each end of the alignment. Assumes that graph is topologically sorted by node index.

virtual void vg::BaseAligner::**align_pinned**(*Alignment* &*alignment*, *Graph* &*g*, bool *pin_left*)
= 0

virtual void vg::BaseAligner::**align_pinned_multi**(*Alignment* &*alignment*, vector<*Alignment*> &*alt_alignments*, *Graph* &*g*, bool *pin_left*, int32_t *max_alt_alns*)
= 0

virtual void vg::BaseAligner::**align_global_banded**(*Alignment* &*alignment*, *Graph* &*g*, int32_t *band_padding* = 0, bool *permissive_banding* = true)
= 0

virtual void vg::BaseAligner::**align_global_banded_multi**(*Alignment* &*alignment*, vector<*Alignment*> &*alt_alignments*, *Graph* &*g*, int32_t *max_alt_alns*, int32_t *band_padding* = 0, bool *permissive_banding* = true)
= 0

virtual int32_t vg::BaseAligner::**score_exact_match**(**const** *Alignment* &*aln*, size_t *read_offset*, size_t *length*)
= 0Compute the score of an exact match in the given alignment, from the given offset, of the given length.

virtual int32_t vg::BaseAligner::**score_exact_match**(**const** string &*sequence*, **const** string &*base_quality*) **const**
= 0Compute the score of an exact match of the given sequence with the given qualities. Qualities may be ignored by some implementations.

virtual int32_t vg::BaseAligner::**score_exact_match**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*, string::const_iterator *base_qual_begin*) **const**
= 0Compute the score of an exact match of the given range of sequence with the given qualities. Qualities may be ignored by some implementations.

int32_t BaseAligner::**score_gap**(size_t *gap_length*)
Returns the score of an insert or deletion of the given length.

void BaseAligner::**compute_mapping_quality**(vector<Alignment> &*alignments*, int *max_mapping_quality*, bool *fast_approximation*, double *cluster_mq*, bool *use_cluster_mq*, int *overlap_count*, double *mq_estimate*, double *maybe_mq_threshold*, double *identity_weight*)
stores -10 * log_10(P_err) in alignment mapping_quality field where P_err is the probability that the

alignment is not the correct one (assuming that one of the alignments in the vector is correct). alignments must have been created with this *Aligner* for quality score to be valid

void BaseAligner::**compute_paired_mapping_quality**(pair<vector<Alignment>, vector<Alignment>> &*alignment_pairs*, **const** vector<double> &*frag_weights*, int *max_mapping_quality1*, int *max_mapping_quality2*, bool *fast_approximation*, double *cluster_mq*, bool *use_cluster_mq*, int *overlap_count1*, int *overlap_count2*, double *mq_estimate1*, double *mq_estimate2*, double *maybe_mq_threshold*, double *identity_weight*)

same function for paired reads, mapping qualities are stored in both alignments in the pair

int32_t BaseAligner::**compute_mapping_quality**(vector<double> &*scores*, bool *fast_approximation*)

Computes mapping quality for the optimal score in a vector of scores.

double BaseAligner::**mapping_quality_score_diff**(double *mapping_quality*) **const**

Returns the difference between an optimal and second-best alignment scores that would result in this mapping quality using the fast mapping quality approximation

double BaseAligner::**score_to_unnormalized_likelihood_ln**(double *score*)

Convert a score to an unnormalized log likelihood for the sequence. Requires log_base to have been set.

size_t BaseAligner::**longest_detectable_gap**(**const** Alignment &*alignment*, **const** string::const_iterator &*read_pos*) **const**

The longest gap detectable from a read position without soft-clipping.

size_t BaseAligner::**longest_detectable_gap**(**const** Alignment &*alignment*) **const**

The longest gap detectable from any read position without soft-clipping.

int32_t BaseAligner::**score_gappy_alignment**(**const** Alignment &*aln*, **const** function<size_t(pos_t, pos_t, size_t)> &*estimate_distance*, bool *strip_bonuses* = false)

Use the score values in the aligner to score the given alignment, scoring gaps caused by jumping between between nodes using a custom gap length estimation function (which takes the from position, the to position, and a search limit in bp that happens to be the read length).

May include full length bonus or not. TODO: bool flags are bad.

int32_t BaseAligner::**score_ungapped_alignment**(**const** Alignment &*aln*, bool *strip_bonuses* = false)

Use the score values in the aligner to score the given alignment assuming that there are no gaps between Mappings in the *Path*

int32_t BaseAligner::**remove_bonuses**(**const** Alignment &*aln*, bool *pinned* = false, bool *pin_left* = false)

Without necessarily rescoring the entire alignment, return the score of the given alignment with bonuses removed. Assumes that bonuses are actually included in the score. Needs to know if the alignment was pinned-end or not, and, if so, which end was pinned.

Public Members

int8_t *vg::BaseAligner::**nt_table**

int8_t *vg::BaseAligner::**score_matrix**

int8_t vg::BaseAligner::**match**

int8_t vg::BaseAligner::**mismatch**

int8_t vg::BaseAligner::**gap_open**

int8_t vg::BaseAligner::**gap_extension**

int8_t vg::BaseAligner::**full_length_bonus**

double vg::BaseAligner::**log_base**

Protected Functions

vg::BaseAligner::**BaseAligner**()

BaseAligner::**~BaseAligner**()

gssw_graph *BaseAligner::**create_gssw_graph**(Graph &*g*)

void vg::BaseAligner::**visit_node**(gssw_node **node*, list<gssw_node *> &*sorted_nodes*, set<gssw_node *> &*unmarked_nodes*, set<gssw_node *> &*temporary_marks*)

void BaseAligner::**reverse_graph**(Graph &*g*, Graph &*reversed_graph_out*)

void BaseAligner::**unreverse_graph**(Graph &*graph*)

void BaseAligner::**unreverse_graph_mapping**(gssw_graph_mapping **gm*)

void BaseAligner::**gssw_mapping_to_alignment**(gssw_graph **graph*, gssw_graph_mapping **gm*, Alignment &*alignment*, bool *pinned*, bool *pin_left*, bool *print_score_matrices* = false)

string BaseAligner::**graph_cigar**(gssw_graph_mapping **gm*)

double BaseAligner::**maximum_mapping_quality_exact**(vector<double> &*scaled_scores*, size_t **max_idx_out*)

double BaseAligner::**maximum_mapping_quality_approx**(vector<double> &*scaled_scores*, size_t **max_idx_out*)

double BaseAligner::**estimate_next_best_score**(int *length*, double *min_diffs*)

void BaseAligner::**init_mapping_quality**(double *gc_content*)

class *#include <mapper.hpp>* Inherits from *vg::Progressive* Subclassed by *vg::Mapper*, *vg::MultipathMapper*

Public Functions

vg::BaseMapper::**BaseMapper**(*xg*::*XG* **xidex*, *gcsa*::GCSA **g*, *gcsa*::LCPArray **a*)

vg::BaseMapper::**BaseMapper**(void)

vg::BaseMapper::**~BaseMapper**(void)

double vg::BaseMapper::**estimate_gc_content**(void)

int vg::BaseMapper::**random_match_length**(double *chance_random*)

void vg::BaseMapper::**set_alignment_scores**(int8_t *match*, int8_t *mismatch*, int8_t *gap_open*, int8_t *gap_extend*, int8_t *full_length_bonus*)

void vg::BaseMapper::**set_fragment_length_distr_params**(size_t *maximum_sample_size* = 1000, size_t *reestimation_frequency* = 1000, double *robust_estimation_fraction* = 0.95)

void vg::BaseMapper::**set_alignment_threads**(int *new_thread_count*)
Set the alignment thread count, updating internal data structures that are per thread. Note that this resets aligner scores to their default values!

void vg::BaseMapper::**set_cache_size**(int *new_cache_size*)

bool vg::BaseMapper::**has_fixed_fragment_length_distr**()
Returns true if fragment length distribution has been fixed.

void vg::BaseMapper::**force_fragment_length_distr**(double *mean*, double *stddev*)
Use the given fragment length distribution parameters instead of estimating them.

vector<*MaximalExactMatch*> vg::BaseMapper::**find_mems_deep**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*, double &*lcp_avg*, double &*fraction_filtered*, int *max_mem_length* = 0, int *min_mem_length* = 1, int *reseed_length* = 0, bool *use_lcp_reseed_heuristic* = false, bool *use_diff_based_fast_reseed* = false, bool *include_parent_in_sub_mem_count* = false, bool *record_max_lcp* = false, int *reseed_below_count* = 0)

vector<*MaximalExactMatch*> vg::BaseMapper::**find_mems_simple**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*, int *max_mem_length* = 0, int *min_mem_length* = 1, int *reseed_length* = 0)

Public Members

int vg::BaseMapper::**min_mem_length**

int vg::BaseMapper::**mem_reseed_length**

bool vg::BaseMapper::**fast_reseed**

double vg::BaseMapper::**fast_reseed_length_diff**

bool vg::BaseMapper::**adaptive_reseed_diff**

double vg::BaseMapper::**adaptive_diff_exponent**

int vg::BaseMapper::**hit_max**

bool vg::BaseMapper::**use_approx_sub_mem_count**

bool vg::BaseMapper::**strip_bonuses**

bool vg::BaseMapper::**assume_acyclic**

bool vg::BaseMapper::**adjust_alignments_for_base_quality**

MappingQualityMethod vg::BaseMapper::**mapping_quality_method**

int vg::BaseMapper::**max_mapping_quality**

Protected Functions

void vg::BaseMapper::**find_sub_mems**(**const** vector<*MaximalExactMatch*> &*mems*, int *mem_idx*, string::const_iterator *next_mem_end*, int *min_mem_length*, vector<pair<*MaximalExactMatch*, vector<size_t>>> &*sub_mems_out*)

Locate the sub-MEMs contained in the last MEM of the mems vector that have ending positions before the end the next SMEM, label each of the sub-MEMs with the indices of all of the SMEMs that contain it

void vg::BaseMapper::**find_sub_mems_fast**(**const** vector<*MaximalExactMatch*> &*mems*, int *mem_idx*, string::const_iterator *next_mem_end*, int *min_sub_mem_length*, vector<pair<*MaximalExactMatch*, vector<size_t>>> &*sub_mems_out*)

Provides same semantics as find_sub_mems but with a different algorithm. This algorithm uses the min_mem_length as a pruning tool instead of the LCP index. It can be expected to be faster when both the min_mem_length reasonably large relative to the reseed_length (e.g. 1/2 of SMEM size or similar).

void vg::BaseMapper::**fill_nonredundant_sub_mem_nodes**(vector<*MaximalExactMatch*> &*parent_mems*, vector<pair<*MaximalExactMatch*, vector<size_t>>>::iterator *sub_mem_records_begin*, vector<pair<*MaximalExactMatch*, vector<size_t>>>::iterator *sub_mem_records_end*)

finds the nodes of sub MEMs that do not occur inside parent MEMs, each sub MEM should be associated with a vector of the indices of the SMEMs that contain it in the parent MEMs vector

void vg::BaseMapper::**first_hit_positions_by_index**(*MaximalExactMatch* &*mem*, vector<set<*pos_t*>> &*positions_by_index_out*)
fills a vector where each element contains the set of positions in the graph that the MEM touches at that index for the first MEM hit in the GCSA array

void vg::BaseMapper::**mem_positions_by_index**(*MaximalExactMatch* &*mem*, *pos_t* *hit_pos*, vector<set<*pos_t*>> &*positions_by_index_out*)
fills a vector where each element contains the set of positions in the graph that the MEM touches at that index starting at a given hit

char vg::BaseMapper::**pos_char**(*pos_t* *pos*)

map<*pos_t*, char> vg::BaseMapper::**next_pos_chars**(*pos_t* *pos*)

set<*pos_t*> vg::BaseMapper::**positions_bp_from**(*pos_t* *pos*, int *distance*, bool *rev*)

set<*pos_t*> vg::BaseMapper::**sequence_positions**(**const** string &*seq*)

size_t vg::BaseMapper::**get_adaptive_min_reseed_length**(size_t *parent_mem_length*)

void vg::BaseMapper::**check_mems**(**const** vector<*MaximalExactMatch*> &*mems*)

void vg::BaseMapper::**init_aligner**(int8_t *match*, int8_t *mismatch*, int8_t *gap_open*, int8_t *gap_extend*, int8_t *full_length_bonus*)

void vg::BaseMapper::**clear_aligners**(void)

BaseAligner *vg::BaseMapper::**get_aligner**(bool *have_qualities* = true) **const**
Get the appropriate aligner to use, based on adjust_alignments_for_base_quality. By setting have_qualities to false, you can force the non-quality-adjusted aligner, for reads that lack quality scores.

QualAdjAligner *vg::BaseMapper::**get_qual_adj_aligner**() **const**

Aligner *vg::BaseMapper::**get_regular_aligner**() **const**

Protected Attributes

int vg::BaseMapper::**alignment_threads**

xg::*XG* *vg::BaseMapper::**xindex**

gcsa::GCSA *vg::BaseMapper::**gcsa**

gcsa::LCPArray *vg::BaseMapper::**lcp**

FragmentLengthDistribution vg::BaseMapper::**fragment_length_distr**

Protected Static Attributes

thread_local vector<size_t> vg::BaseMapper::**adaptive_reseed_length_memo**

Private Members

QualAdjAligner *vg::BaseMapper::**qual_adj_aligner**

Aligner *vg::BaseMapper::**regular_aligner**

template <typename *Value*, typename *Parser* = *OptionValueParser*<Value>>
class *#include <option.hpp>* The correct user entry point is one of the Option<> specializations, not this class.

Represents a configurable parameter for a class that we might want to expose on the command line.

We need to allow these to be defined with default values in the class's header, but be overridable by simple assignment of a value of the right type. But we also need to be able to interrogate an instance of the class to get all its options that need to be filled in and their help text, and ideally to choose short options to assign to them that don't conflict.

So our approach is to have the class keep track of all its option objects, and to have the options register themselves at construction time with a passed this pointer (since this is in scope in the class body in the header).

Option instances MUST exist as MEMBERS of classes that inherit from Configurable! They can't live in vectors or anything, or the magic code that tracks them as the enclosing object moves won't work!

We'll have an assignment operator from the wrapped type to make setting options manually easy.

We also have magical conversion to the wrapped type.

We wrap all the type-specific parsing into a parser template. You could specify your own if you want custom parsing logic for like a map or something.

Inherits from *vg::OptionInterface*

Subclassed by *vg::Option< Value, Parser >*

Public Functions

vg::BaseOption::**BaseOption**()
No default constructor.

virtual vg::BaseOption::**~BaseOption**()
Default destructor.

vg::BaseOption::**BaseOption**(*Configurable* *owner, **const** string &long_opt, **const** string &short_opts, **const** Value &default_value, **const** string &description)
Make a new *BaseOption* that lives in a class, with the given name, short option characters, and default value.

BaseOption<Value, Parser> &vg::BaseOption::**operator=**(**const** *BaseOption*<Value, Parser> &other)
Assignment from an *Option* of the correct type.

BaseOption<Value, Parser> &vg::BaseOption::**operator=**(**const** Value &other)
Assignment from an unwrapped value.

vg::BaseOption::**operator Value&**()
Conversion to the wrapped type.

virtual const string &vg::BaseOption::**get_long_option**() **const**
Get the long oiption text without , like "foos-to-bar".

virtual const string &vg::BaseOption::**get_short_options**() **const**
Gets a list of short option characters that the option wants, in priority order. If none of these are available, the option will be automatically assigned some other free character.

virtual const string &vg::BaseOption::**get_description**() **const**
Get the description, like "number of foos to bar per frobnitz".

virtual string vg::BaseOption::**get_default_value**() **const**
Get the default value as a string.

virtual bool vg::BaseOption::**has_argument**() **const**
Returns true if the option takes an argument, and false otherwise.

virtual void vg::BaseOption::**parse**()
Called for no-argument options when the parser encounters them.

virtual void vg::BaseOption::**parse**(**const** string &*arg*)
Called for argument-having options when the parser encounters them. The passed reference is only valid during the function call, so the option should make a copy.

Protected Attributes

string vg::BaseOption::**long_opt**
What is the options's long option name.

string vg::BaseOption::**short_opts**
What is the option's short option name.

string vg::BaseOption::**description**
How is this option described to the user?

Value vg::BaseOption::**value**
We keep a value of our chosen type. It just has to be copyable and assignable.

Value vg::BaseOption::**default_value**
We also keep the default value around, in case we somehow get assigned and then get asked to report our metadata. TODO: do this as a string instead?

struct Summarizes reads that map to single position in the graph. This structure is pretty much identical to a line in Samtools pileup format if qualities set, it must have size = num_bases **Public Members**

int32 vg::BasePileup::**ref_base**

int32 vg::BasePileup::**num_bases**

string vg::BasePileup::**bases**

bytes vg::BasePileup::**qualities**

struct *#include <benchmark.hpp>* Represents the results of a benchmark run. Tracks the mean and standard deviation of a number of runs of a function under test, interleaved with runs of a standard control function. **Public Functions**

double vg::BenchmarkResult::**score**() **const**
How many control-standardized "points" do we score?

double vg::BenchmarkResult::**score_error**() **const**
What is the uncertainty on the score?

Public Members

size_t vg::BenchmarkResult::**runs**
How many runs were done.

benchtime vg::BenchmarkResult::**test_mean**
What was the mean runtime of each test run.

benchtime vg::BenchmarkResult::**test_stddev**
What was the standard deviation of test run times.

benchtime vg::BenchmarkResult::**control_mean**
What was the mean runtime of each control run.

benchtime vg::BenchmarkResult::**control_stddev**
What was the standard deviation of control run times.

string vg::BenchmarkResult::**name**
What was the name of the test being run.

struct *#include <srpe.hpp>*

Public Functions

int vg::BREAKPOINT::**total_supports**()

bool vg::BREAKPOINT::**overlap**(*BREAKPOINT p*, int *dist*)

string vg::BREAKPOINT::**to_string**()

Public Members

string vg::BREAKPOINT::**name**

Position vg::BREAKPOINT::**position**

vector<*BREAKPOINT*> vg::BREAKPOINT::**mates**

string vg::BREAKPOINT::**contig**

int64_t vg::BREAKPOINT::**start**

int64_t vg::BREAKPOINT::**upper_bound**

int64_t vg::BREAKPOINT::**lower_bound**

bool vg::BREAKPOINT::**isForward**

int vg::BREAKPOINT::**SV_TYPE**

int vg::BREAKPOINT::**normal_supports**

int vg::BREAKPOINT::**tumor_supports**

int vg::BREAKPOINT::**fragl_supports**

int vg::BREAKPOINT::**split_supports**

int vg::BREAKPOINT::**other_supports**

struct *#include <cactus.hpp>*Public Members

int64_t `vg::CactusSide::`**`node`**

bool `vg::CactusSide::`**`is_end`**
class *#include <genotypekit.hpp>* Class for finding all snarls using the base-level Cactus snarl decomposition interface. Inherits from *vg::SnarlFinder* Public Functions

`vg::CactusSnarlFinder::`**`CactusSnarlFinder`**`(`*VG &graph*`)`
Make a new *CactusSnarlFinder* to find snarls in the given graph. We can't filter trivial bubbles because that would break our chains.

Optionally takes a hint path name.

`vg::CactusSnarlFinder::`**`CactusSnarlFinder`**`(`*VG &graph*, **const** string *&hint_path*`)`
Make a new *CactusSnarlFinder* with a single hinted path to base the decomposition on.

SnarlManager `vg::CactusSnarlFinder::`**`find_snarls`**`()`
Find all the snarls with Cactus, and put them into a *SnarlManager*.

Private Functions

const *Snarl* *`vg::CactusSnarlFinder::`**`recursively_emit_snarls`**`(`**const** *Visit &start*, **const** *Visit &end*, **const** *Visit &parent_start*, **const** *Visit &parent_end*, stList **chains_list*, stList **unary_snarls_list*, *SnarlManager &destination*`)`
Create a snarl in the given *SnarlManager* with the given start and end, containing the given child snarls in the list of chains of children and the given list of unary children. Recursively creates snarls in the *SnarlManager* for the children. Returns a pointer to the finished snarl in the *SnarlManager*. Start and end may be empty visits, in which case no snarl is created, all the child chains are added as root chains, and null is returned. If parent_start and parent_end are empty Visits, no parent() is added to the produced snarl.

Private Members

VG &`vg::CactusSnarlFinder::`**`graph`**
Holds the vg graph we are looking for sites in.

unordered_set<string> `vg::CactusSnarlFinder::`**`hint_paths`**
Holds the names of reference path hints.
struct *#include <snarls.hpp>* We want to be able to loop over a chain and get iterators to pairs of the snarl and its orientation in the chain. So we define some iterators. Public Functions

ChainIterator &`vg::ChainIterator::`**`operator++`**`()`
Advance the iterator.

pair<**const** *Snarl* *, bool> `vg::ChainIterator::`**`operator*`**`()` **const**
Get the snarl we're at and whether it is backward.

const pair<**const** *Snarl* *, bool> *vg::ChainIterator::**operator->**() **const**
Get a pointer to the thing we get when we dereference the iterator.

bool vg::ChainIterator::**operator==**(**const** *ChainIterator* &*other*) **const**
We need to define comparison because C++ doesn't give it to us for free.

bool vg::ChainIterator::**operator!=**(**const** *ChainIterator* &*other*) **const**

Public Members

bool vg::ChainIterator::**go_left**
Are we a reverse iterator or not?

bool vg::ChainIterator::**backward**
Is the snarl we are at backward or forward in its chain?

Chain::const_iterator vg::ChainIterator::**pos**
What position in the underlying vector are we in?

Chain::const_iterator vg::ChainIterator::**chain_start**
What are the bounds of that underlying vector?

Chain::const_iterator vg::ChainIterator::**chain_end**

bool vg::ChainIterator::**is_rend**
Since we're using backing random access itarators to provide reverse iterators, we need a flag to see if we are rend (i.e. before the beginning)

bool vg::ChainIterator::**complement**
When dereferencing, should we flip snarl orientations form the orientations they appear at in the chain when read left to right?

pair<**const** *Snarl* *, bool> vg::ChainIterator::**scratch**
In order to dereference to a pair with -> we need a place to put the pair so we can have a pointer to it. Gets lazily set to wherever the iterator is pointing when we do ->

class *#include <colors.hpp>*

Public Functions

vg::Colors::**Colors**(void)

vg::Colors::**Colors**(int *seed_val*)

vg::Colors::**~Colors**(void)

string vg::Colors::**hashed**(**const** string &*str*)

string vg::Colors::**random**(void)

Public Members

const vector<string> vg::Colors::**colors**

Private Members

mt19937 vg::Colors::**rng**
class *#include <option.hpp>* Represents a thing-doing class that has options. You construct the class, configure its options, and then set it going. Subclassed by *vg::SupportCaller* **Public Functions**

void vg::Configurable::**register_option**(*OptionInterface* *option)
Each option will call this on construction and register with its owner.

vector<*OptionInterface* *> vg::Configurable::**get_options**()
Get all the options for this class. These pointers are only valid unless or until the underlying *Configurable* object moves or is assigned to.

string vg::Configurable::**get_name**()
Get the name of the thing being configured, for titling the help section.

Private Members

vector<ptrdiff_t> vg::Configurable::**option_offsets**
We really should be using something like CRTP and pointer-to-members on the actual class that's configurable, but for now we'll just exploit int<->pointer conversion and store the offset from us in memory to the location of the option in memory. If the option is correctly a mamber of a class derived from this one, it will work correctly because all instances share the same memory layout for members.
class *#include <option.hpp>* Actual implementation class that connects a bunch of *Configurable* things to getopt. TODO: right now every option must have a long option and a char short option. In theory we can use any int as an ID for a long option. We should support long options with untypeable or non-char option IDs. **Public Functions**

vg::ConfigurableParser::**ConfigurableParser**(**const** char **base_short_options* = nullptr, **const** **struct** option **base_long_options* = nullptr, function<void(int)> *handle_base_option* = [](int c){throw runtime_error("Invalid option: "+string(1,(char) c));})
Constructor that sets up our magic option assigning code. Makes a parser that will parse the given base Getopt null-terminated long and short options with Getopt and call the given function with each.

The base option handler, if specified, should error on unrecognized options.

void vg::ConfigurableParser::**register_configurable**(*Configurable* *configurable)
Register a *Configurable* thing and allocate characters for all its options. The *Configurable* is not allowed to move after registration.

void vg::ConfigurableParser::**print_help**(ostream &*out*) **const**
Print option descriptions for registered Configurables to the given stream.

void vg::ConfigurableParser::**parse**(int *argc*, char *******argv*)
Parse the options with getopt. Uses the existing value of the global optind, and updates the global optind, just as getopt would.

Private Members

vector<**struct** option> vg::ConfigurableParser::**long_options**
Holds all the long option structs for options we have.

set<int> vg::ConfigurableParser::**available_short_options**
Holds all the available short option characters that have not been assigned. Stored as ints so we can check ints against it safely.

string vg::ConfigurableParser::**short_options**
Holds the short options string (with colons) for the short options we have.

map<int, *OptionInterface* *> vg::ConfigurableParser::**options_by_code**
Keep a map from assigned character to actual option.

map<*OptionInterface* *, int> vg::ConfigurableParser::**codes_by_option**
And a reverse map for when we look up assigned codes for printing. TODO: No forgery!

set<string> vg::ConfigurableParser::**long_options_used**
And a set of long options used so we can detect collisions.

vector<*Configurable* *> vg::ConfigurableParser::**configurables**
And a vector of Configurables in the order registered so we can interrogate them and group their options when printing help.

function<void(int)> vg::ConfigurableParser::**handle_base_option**
Handle an option not assigned to an *Option* object.

class *#include <genotypekit.hpp>* Represents a strategy for computing consistency between Alignments and SnarlTraversals. Determines whether a read is consistent with a *SnarlTraversal* or not (but has access to all the SnarlTraversals). Polymorphic base class/interface. Subclassed by *vg::SimpleConsistencyCalculator* **Public Functions**

virtual vg::ConsistencyCalculator::**~ConsistencyCalculator**()

virtual vector<bool> vg::ConsistencyCalculator::**calculate_consistency**(**const** *Snarl* &*site*, **const** vector<*SnarlTraversal*> &*traversals*, **const** *Alignment* &*read*) **const** = 0Return true or false for each tarversal of the site, depending on if the read is consistent with it or not.

struct *#include <constructor.hpp>* Represents a constructed region of the graph alogn a single linear sequence. Contains the protobuf *Graph* holding all the created components (which may be too large to serialize), a set of node IDs whose left sides need to be connected to when you connect to the start of the chunk, and a set of node IDs whose right sides need to be connected to when you connect to the end of the chunk. *Node* ordering is restricted: if there is a single source, it must be the very first node in the graph with ID 1, and if there is a single sink it must be the very last node in the graph with ID max_id. Additionally, single sources and single sinks must be visited by only a single path, the reference path. The overall reference path must also always be path 0. Also, all mappings in all paths must be full-length matches on the forward strand, and they must be sorted by rank. Ranks must be filled and start with rank 1 in each path. **Public Members**

Graph vg::ConstructedChunk::**graph**

id_t vg::ConstructedChunk::**max_id**

set<*id_t*> vg::ConstructedChunk::**left_ends**

set<*id_t*> vg::ConstructedChunk::**right_ends**

class *#include <constructor.hpp>* Inherits from *vg::Progressive*, *vg::NameMapper*

Public Functions

ConstructedChunk vg::Constructor::**construct_chunk**(string *reference_sequence*, string *reference_path_name*, vector<vcflib::Variant> *variants*, size_t *chunk_offset*) **const**

Construct a *ConstructedChunk* of graph from the given piece of sequence, with the given name, applying the given variants. The variants need to be sorted by start position, and have their start positions set to be ZERO- BASED. However, they also need to have their start positions relative to the global start of the contig, so that hash-based names come out right for them. They also need to not overlap with any variants not in the vector we have (i.e. we need access to all overlapping variants for this region). The variants must not extend beyond the given sequence, though they can abut its edges.

Variants in the vector may not use symbolic alleles.

chunk_offset gives the global 0-based position at which this chunk starts in the reference contig it is part of, which is used to correctly place variants.

void vg::Constructor::**construct_graph**(string *vcf_contig*, FastaReference &*reference*, *VcfBuffer* &*variant_source*, **const** vector<FastaReference *> &*insertion*, function<void) *Graph*&

> *callback*Construct a graph for the given VCF contig name, using the given reference and the variants from the given buffered VCF file. Emits a sequence of *Graph* chunks, which may be too big to serealize directly.

Doesn't handle any of the setup for VCF indexing. Just scans all the variants that can come out of the buffer, so make sure indexing is set on the file first before passing it in.

void vg::Constructor::**construct_graph**(**const** vector<FastaReference *> &*references*, **const** vector<vcflib::VariantCallFile *> &*variant_files*, **const** vector<FastaReference *> &*insertions*, function<void) *Graph*&

> *callback*Construct a graph using the given FASTA references and VCFlib VCF files. The VCF files are assumed to be grouped by contig and then sorted by position within the contig, such that each contig is present in only one file. If multiple FASTAs are used, each contig must be present in only one FASTA file. Reference and VCF vectors may not contain nulls.

Public Members

bool vg::Constructor::**flat**

bool vg::Constructor::**alt_paths**

bool vg::Constructor::**do_svs**

bool vg::Constructor::**alts_as_loci**

bool vg::Constructor::**greedy_pieces**

bool vg::Constructor::**chain_deletions**

bool vg::Constructor::**warn_on_lowercase**

size_t vg::Constructor::**max_node_size**

size_t vg::Constructor::**vars_per_chunk**

size_t vg::Constructor::**bases_per_chunk**

set<string> vg::Constructor::**allowed_vcf_names**

map<string, pair<size_t, size_t>> vg::Constructor::**allowed_vcf_regions**

Protected Attributes

set<string> vg::Constructor::**symbolic_allele_warnings**
Remembers which unusable symbolic alleles we've already emitted a warning about during construction.

id_t vg::Constructor::**max_id**
All chunks are generated with IDs starting at 1, but graphs emitted from construct_graph need to have the IDs rewritten so they don't overlap. Moreover, multiple calls to construct_graph need to not have conflicting IDs, because some construct_graph implementations call other ones. What we do for now is globally track the max ID already used, so all calls to construct_graph follow a single ID ordering.

Private Members

unordered_set<string> vg::Constructor::**warned_sequences**
What sequences have we warned about containing lowercase characters?

Private Static Functions

void vg::Constructor::**trim_to_variable**(vector<list<vcflib::VariantAllele>> &*parsed_alleles*)
Given a vector of lists of VariantAllele edits, trim in from the left and right, leaving a core of edits bounded by edits that actually change the reference in at least one allele.

Postcondition: either all lists of VariantAlleles are empty, or at least one begins with a non-match and at least one ends with a non-match.

void vg::Constructor::**condense_edits**(list<vcflib::VariantAllele> &*parsed_allele*)
Given a list of VariantAllele edits, condense adjacent perfect match edits to be maximally long.

pair<int64_t, int64_t> vg::Constructor::**get_bounds**(**const** vector<list<vcflib::VariantAllele>> &*trimmed_variant*)
Given a vector of lists of VariantAllele edits that have been trimmed with *trim_to_variable()* above, one per non-reference alt for a variant, return the position of the first varaible base, and the position of the last variable base. If there's no variable-region, the result is max int64_t and -1, and if there's a 0-length variable region, the result is the base after it and the base before it.

struct *#include <readfilter.hpp>*

Public Functions

vg::ReadFilter::Counts::**Counts**()

Counts &vg::ReadFilter::Counts::**operator+=**(**const** *Counts* &*other*)

Public Members

vector<size_t> vg::ReadFilter::Counts::**read**

vector<size_t> vg::ReadFilter::Counts::**filtered**

vector<size_t> vg::ReadFilter::Counts::**min_score**

vector<size_t> vg::ReadFilter::Counts::**max_overhang**

vector<size_t> vg::ReadFilter::Counts::**min_end_matches**

vector<size_t> vg::ReadFilter::Counts::**min_mapq**

vector<size_t> vg::ReadFilter::Counts::**split**

vector<size_t> vg::ReadFilter::Counts::**repeat**

vector<size_t> vg::ReadFilter::Counts::**defray**

struct Public Functions

bool custom_aln_sort_key::**operator()**(**const** Alignment *a_one*, **const** Alignment *a_two*)

struct Public Functions

bool custom_pos_sort_key::**operator()**(**const** Position *lhs*, **const** Position *rhs*)

class *#include <deconstructor.hpp>*Public Functions

vg::Deconstructor::**Deconstructor**()

vg::Deconstructor::**~Deconstructor**()

pair<bool, vector<string>> vg::Deconstructor::**get_alleles**(vector<*SnarlTraversal*> *travs*, string *refpath*, *vg*::*VG* **graph*)

Takes in a vector of snarltraversals returns their sequences as a vector<string> returns a boolean hasRef if a reference path is present, hasRef is set to true and the first string in the vector is the reference allele otherwise, hasRef is set to false and all strings are alt alleles.

void vg::Deconstructor::**deconstruct**(string *refpath*, *vg*::*VG* **graph*)

void vg::Deconstructor::**deconstruct**(vector<string> *refpaths*, *vg*::*VG* **graph*)

Convenience wrapper function for deconstruction of multiple paths.

Public Members

map<string, *PathIndex* *> vg::Deconstructor::**pindexes**

Private Members

bool vg::Deconstructor::**headered**

class *#include <banded_global_aligner.hpp>* Represents a deviation from the optimal traceback that a sub-optimal traceback takes. **Public Functions**

`BandedGlobalAligner::AltTracebackStack::Deflection::`**`Deflection`**`(`**const** int64_t *from_node_id*, **const** int64_t *row_idx*, **const** int64_t *col_idx*, **const** int64_t *to_node_id*, **const** matrix_t *to_matrix*`)`

`BandedGlobalAligner::AltTracebackStack::Deflection::`**`~Deflection`**`()`

Public Members

template<>
const int64_t `vg::BandedGlobalAligner<IntType>::AltTracebackStack::Deflection::`**`from_node_id`**
Node ID where deflection occurs.

template<>
const int64_t `vg::BandedGlobalAligner<IntType>::AltTracebackStack::Deflection::`**`row_idx`**
Coordinate from rectangularized band (not original matrix) where deflection occurs.

template<>
const int64_t `vg::BandedGlobalAligner<IntType>::AltTracebackStack::Deflection::`**`col_idx`**
Coordinate from rectangularized band (not original matrix) where deflection occurs.

template<>
const int64_t `vg::BandedGlobalAligner<IntType>::AltTracebackStack::Deflection::`**`to_node_id`**
Node to deflect to.

template<>
const matrix_t `vg::BandedGlobalAligner<IntType>::AltTracebackStack::Deflection::`**`to_matrix`**
Dynamic programming matrix deflect to.

struct *#include <genome_state.hpp>* Inherits from *vg::GenomeStateCommand* **Public Functions**

GenomeStateCommand *`vg::DeleteHaplotypeCommand::`**`execute`**`(`*GenomeState* **const** *&state*`)`
Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual `vg::DeleteHaplotypeCommand::`**`~DeleteHaplotypeCommand`**`()`

Public Members

unordered_map<**const** *Snarl* *, vector<size_t>> `vg::DeleteHaplotypeCommand::`**`deletions`**
For each snarl, delete the haplotype in each overall lane in the vector. You must specify out the deletions for all the snarls in a haplotype; we won't automatically go and find the children if you just list to delete from the parents. Deletions happen from begin to end through each vector. Lane numbers for a given snarl must be strictly decreasing.

class *#include <srpe.hpp>* Overview: Use the GAM/GAM index and a filter to locate Alignments which may indicate the presence of structural variants at a given site. Signatures include: Deletions/Insertions: Stacked soft clips (tips) Inversions: mismatched P/E reads(< && > rather than the expected (> <) Duplications: Read depth signals Translocations: Distant read pairs **Public Functions**

`vg::DepthMap::`**`DepthMap`**(int64_t *sz*)

`vg::DepthMap::`**`DepthMap`**()

`vg::DepthMap::`**`DepthMap`**(*vg*::*VG* **graph*)

int8_t `vg::DepthMap::`**`get_depth`**(int64_t *node_id*, int64_t *offset*)

void `vg::DepthMap::`**`set_depth`**(int64_t *node_id*, int64_t *offset*, int8_t *d*)

void `vg::DepthMap::`**`increment_depth`**(int64_t *node_id*, int64_t *offset*)

void `vg::DepthMap::`**`fill_depth`**(**const** *vg*::*Path* &*p*)

Public Members

int8_t **`vg::DepthMap::`**`depths`
Map <node_id : offset : depth> or Map <*SnarlTraversal* : support count>

uint64_t `vg::DepthMap::`**`size`**

map<int64_t, uint64_t> `vg::DepthMap::`**`node_pos`**

vg::*VG* **`vg::DepthMap::`**`g_graph`
struct *#include <cluster.hpp>***Public Functions**

`vg::OrientedDistanceClusterer::DPScoreComparator::`**`DPScoreComparator`**()

`vg::OrientedDistanceClusterer::DPScoreComparator::`**`DPScoreComparator`**(**const** vector<*ODNode*> &*nodes*)

`vg::OrientedDistanceClusterer::DPScoreComparator::`**`~DPScoreComparator`**()

bool `vg::OrientedDistanceClusterer::DPScoreComparator::`**`operator()`**(**const** size_t *i*, **const** size_t *j*)

Private Members

const vector<*ODNode*> &vg::OrientedDistanceClusterer::DPScoreComparator::**nodes**
struct *Edges* describe linkages between nodes. They are bidirected, connecting the end (default) or start of the "from" node to the start (default) or end of the "to" node. **Public Members**

int64 vg::Edge::**from**
ID of upstream node.

int64 vg::Edge::**to**
ID of downstream node.

bool vg::Edge::**from_start**
If the edge leaves from the 5' (start) of a node.

bool vg::Edge::**to_end**
If the edge goes to the 3' (end) of a node.

int32 vg::Edge::**overlap**
Length of overlap between the connected *Node*s.
struct Keep pileup-like record for reads that span edges. **Public Members**

Edge vg::EdgePileup::**edge**

int32 vg::EdgePileup::**num_reads**
total reads mapped

int32 vg::EdgePileup::**num_forward_reads**
number of reads mapped on forward strand

bytes vg::EdgePileup::**qualities**
struct Edits describe how to generate a new string from elements in the graph. To determine the new string, just walk the series of edits, stepping from_length distance in the basis node, and to_length in the novel element, replacing from_length in the basis node with the sequence. There are several types of *Edit*: *matches*: from_length == to_length; sequence is empty *snps*: from_length == to_length; sequence = alt *deletions*: to_length == 0 && from_length > to_length; sequence is empty *insertions*: from_length < to_length; sequence = alt

Public Members

int32 vg::Edit::**from_length**
Length in the target/ref sequence that is removed.

int32 vg::Edit::**to_length**
Length in read/alt of the sequence it is replaced with.

string vg::Edit::**sequence**
The replacement sequence, if different from the original sequence.
struct *#include <pileup_augmenter.hpp>* **Public Functions**

vg::NodeDivider::Entry::**Entry**(*Node* *r = 0, vector<*StrandSupport*> *sup_r* = vector< StrandSupport >(), *Node* *a1 = 0, vector<*StrandSupport*> *sup_a1* = vector< StrandSupport >(), *Node* *a2 = 0, vector<*StrandSupport*> *sup_a2* = vector< StrandSupport >())

Node *&vg::NodeDivider::Entry::**operator[]**(int *i*)

vector<*StrandSupport*> &vg::NodeDivider::Entry::**sup**(int *i*)

Public Members

Node *vg::NodeDivider::Entry::**ref**

Node *vg::NodeDivider::Entry::**alt1**

Node *vg::NodeDivider::Entry::**alt2**

vector<*StrandSupport*> vg::NodeDivider::Entry::**sup_ref**

vector<*StrandSupport*> vg::NodeDivider::Entry::**sup_alt1**

vector<*StrandSupport*> vg::NodeDivider::Entry::**sup_alt2**

class *#include <multipath_mapper.hpp>* **Public Members**

string::const_iterator vg::ExactMatchNode::**begin**

string::const_iterator vg::ExactMatchNode::**end**

Path vg::ExactMatchNode::**path**

vector<pair<size_t, size_t>> vg::ExactMatchNode::**edges**

class *#include <genotypekit.hpp>* Inherits from *vg::TraversalFinder* **Public Functions**

vg::ExhaustiveTraversalFinder::**ExhaustiveTraversalFinder**(*VG* &*graph*, *SnarlManager* &*snarl_manager*, bool *include_reversing_traversals* = false)

vg::ExhaustiveTraversalFinder::**~ExhaustiveTraversalFinder**()

vector<*SnarlTraversal*> vg::ExhaustiveTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)

Exhaustively enumerate all traversals through the site. Only valid for acyclic Snarls.

Private Functions

void vg::ExhaustiveTraversalFinder::**stack_up_valid_walks**(*NodeTraversal* *walk_head*, vector<*NodeTraversal*> &*stack*)

void vg::ExhaustiveTraversalFinder::**add_traversals**(vector<*SnarlTraversal*> &*traversals*, *NodeTraversal* *traversal_start*, set<*NodeTraversal*> &*stop_at*, set<*NodeTraversal*> &*yield_at*)

Private Members

VG &vg::ExhaustiveTraversalFinder::**graph**

SnarlManager &vg::ExhaustiveTraversalFinder::**snarl_manager**

bool vg::ExhaustiveTraversalFinder::**include_reversing_traversals**
struct *#include <feature_set.hpp>* Represents a *Feature* that occurs on a path between two inclusive coordinates. Carries along all the extra feature data that BED files store. **Public Members**

string vg::FeatureSet::Feature::**path_name**
What *Path* is the feature on?

size_t vg::FeatureSet::Feature::**first**
What is the first base's position on that path, inclusive?

size_t vg::FeatureSet::Feature::**last**
What is the last base's position on the path, inclusive?

string vg::FeatureSet::Feature::**feature_name**
What is the feature name?

vector<string> vg::FeatureSet::Feature::**extra_data**
What extra BED data should we bring along with this feature? TODO: BED blocks are not parsed and updated, but they should be.
class *#include <feature_set.hpp>* Stores a bunch of features defined on paths, as would be found in a BED file, and listens for messages describing edits to the paths that the features are on. *Edit* operations are of the form "On path X, range Y to Z has been replaced with a segment of length W". Updates the positions and boundaries of features appropriately. Originally designed to allow BED files to be carried through "vg simplify", but could be used for other annotation liftover tasks. **Public Functions**

void vg::FeatureSet::**load_bed**(istream &*in*)
Read features from the given BED stream. Adds them to the collection of loaded features.

void vg::FeatureSet::**save_bed**(ostream &*out*) **const**
Save features to the given BED stream.

void vg::FeatureSet::**on_path_edit**(**const** string &*path*, size_t *start*, size_t *old_length*, size_t *new_length*)
Notify the *FeatureSet* that, at the given path, from the given start position, the given number of bases bave been replaced with the other given number of bases. Can handle pure inserts, pure deletions, length-preserving substitutions, and general length-changing substitutions.

Updates the contained features that need to change.

Note that this is currently O(n) in features on the path. TODO: come up with a truly efficient algorithm to back this using some kind of skip list or rope or something.

const vector<*FeatureSet*::*Feature*> &vg::FeatureSet::**get_features**(**const** string &*path*) **const**
Get the features on a path. Generally used for testing.

Private Members

map<string, vector<*Feature*>> vg::FeatureSet::**features**
Stores all the loaded features by path name.

class *#include <filter.hpp>***Public Functions**

vg::Filter::**Filter**()

vg::Filter::**~Filter**()

bool vg::Filter::**perfect_filter**(*Alignment* &*aln*)

bool vg::Filter::**anchored_filter**(*Alignment* &*aln*)

bool vg::Filter::**mark_sv_alignments**(*Alignment* &*a*, *Alignment* &*b*)

bool vg::Filter::**mark_smallVariant_alignments**(*Alignment* &*a*, *Alignment* &*b*)

Alignment vg::Filter::**depth_filter**(*Alignment* &*aln*)

Alignment vg::Filter::**qual_filter**(*Alignment* &*aln*)
Looks for Alignments that have large overhangs at the end of them.

Default behavior: if an alignment has a right- or left- clip that is longer than the maximum allowed, return an empty alignment.

Inverse Behavior: if the alignment has a clip that is larger than the maximum allowed at either end, return the alignment. CLI: vg filter -d 10 -q 40 -r -R -r: track depth of both novel variants and those in the graph. -R: remove edits that fail the filter (otherwise toss the whole alignment)

Alignment vg::Filter::**coverage_filter**(*Alignment* &*aln*)

Alignment vg::Filter::**avg_qual_filter**(*Alignment* &*aln*)

Alignment vg::Filter::**percent_identity_filter**(*Alignment* &*aln*)
Filter reads that are less than <PCTID> reference. I.E. if a read matches the reference along 80% of its length, and your cutoff is 90% PCTID, throw it out.

bool vg::Filter::**soft_clip_filter**(*Alignment* &*aln*)

bool vg::Filter::**unmapped_filter**(*Alignment* &*aln*)

bool vg::Filter::**split_read_filter**(*Alignment* &*aln*)
Split reads map to two separate paths in the graph OR vastly separated non-consecutive nodes in a single path.

They're super important for detecting structural variants, so we may want to filter them out or collect only split reads.

Alignment vg::Filter::**path_divergence_filter**(*Alignment* &*aln*)
Looks for alignments that transition from one path to another over their length. This may occur for one of several reasons:

1. The read covers a translocation
2. The read looks a lot like two different (but highly-similar paths)
3. The read is shattered (e.g. as in chromothripsis)

Default behavior: if the *Alignment* is path divergent, return an empty *Alignment*, else return aln Inverse behavior: if the *Alignment* is path divergent, return aln, else return an empty *Alignment*

Alignment vg::Filter::**reversing_filter**(*Alignment* &*aln*)
Looks for alignments that change direction over their length. This may happen because of:

1. *Mapping* artifacts
2. Cycles
3. Highly repetitive regions
4. Inversions (if you're lucky enough)

Default behavior: if the *Alignment* reverses, return an empty *Alignment*. inverse behavior: if the *Alignment* reverses, return the *Alignment*.

vector<*Alignment*> `vg::Filter::`**`remap`**(*Alignment &aln*)

vector<*Alignment*> `vg::Filter::`**`remap`**(string *seq*)

bool `vg::Filter::`**`is_left_clipped`**(*Alignment &a*)

pair<*Alignment*, int> `vg::Filter::`**`refactor_split_alignment`**(*Alignment &a*)

Alignment `vg::Filter::`**`path_length_filter`**(*Alignment &aln*)

bool `vg::Filter::`**`one_end_anchored_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

bool `vg::Filter::`**`interchromosomal_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

bool `vg::Filter::`**`insert_size_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

bool `vg::Filter::`**`pair_orientation_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

pair<*Alignment*, *Alignment*> `vg::Filter::`**`deletion_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

pair<*Locus*, *Locus*> `vg::Filter::`**`insertion_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

Filters insertion-characterizing reads based upon discordant pairs (fragment length too short) one-end anchored / softclipped portions and read depth, one day

pair<*Locus*, *Locus*> `vg::Filter::`**`duplication_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

Find reads that support duplications

bool `vg::Filter::`**`inversion_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

Find reads that support inversions split reads discordant insert size bad orientation instead of -> <– we'll see <- <– or -> –>

pair<*Locus*, *Locus*> `vg::Filter::`**`breakend_filter`**(*Alignment &aln_first*, *Alignment &aln_second*)

split reads or discordant reads/insert size may indicate a breakend but not a clean SV type we'd like to report all possible breakends, even if that don't match an SV type very well.

void `vg::Filter::`**`set_min_depth`**(int *depth*)

void `vg::Filter::`**`set_min_qual`**(int *qual*)

void `vg::Filter::`**`set_min_percent_identity`**(double *pct_id*)

void `vg::Filter::`**`set_avg`**(bool *do_avg*)

void `vg::Filter::`**`set_filter_matches`**(bool *fm*)

void vg::Filter::**set_remove_failing_edits** (bool *fm*)

void vg::Filter::**set_soft_clip_limit** (int *max_clip*)

void vg::Filter::**set_split_read_limit** (int *split_limit*)

void vg::Filter::**set_window_length** (int *window_length*)

void vg::Filter::**set_my_vg** (*vg*::*VG* *vg)

void vg::Filter::**set_my_xg_idx** (*xg*::*XG* *xg_idx)

void vg::Filter::**set_inverse** (bool *do_inv*)

void vg::Filter::**init_mapper** ()

void vg::Filter::**fill_node_to_position** (string *pathname*)

int64_t vg::Filter::**distance_between_positions** (*Position first*, *Position second*)

string vg::Filter::**get_clipped_seq** (*Alignment* &*a*)

int64_t vg::Filter::**get_clipped_ref_position** (*Alignment* &*a*)

Position vg::Filter::**get_clipped_position** (*Alignment* &*a*)

Alignment vg::Filter::**remove_clipped_portion** (*Alignment* &*a*)

Public Members

vg::*VG* *vg::Filter::**my_vg**

xg::*XG* *vg::Filter::**my_xg_index**

gcsa::GCSA *vg::Filter::**gcsa_ind**

gcsa::LCPArray *vg::Filter::**lcp_ind**

Mapper *vg::Filter::**my_mapper**

map<int64_t, int64_t> vg::Filter::**node_to_position**

unordered_map<string, unordered_map<string, int>> vg::Filter::**pos_to_edit_to_depth**

unordered_map<int, int> vg::Filter::**pos_to_qual**

bool vg::Filter::**inverse**

bool vg::Filter::**do_remap**

bool vg::Filter::**remove_failing_edits**

bool vg::Filter::**filter_matches**

bool vg::Filter::**use_avg**

int vg::Filter::**min_depth**

int vg::Filter::**min_qual**

int vg::Filter::**min_cov**

int vg::Filter::**window_length**

int vg::Filter::**qual_offset**

int vg::Filter::**soft_clip_limit**

int vg::Filter::**split_read_limit**

double vg::Filter::**min_percent_identity**

double vg::Filter::**min_avg_qual**

int vg::Filter::**max_path_length**

int vg::Filter::**my_max_distance**

float vg::Filter::**insert_mean**

float vg::Filter::**insert_sd**
class *#include <genotypekit.hpp>* This genotype prior calculator has a fixed prior for homozygous genotypes and a fixed prior for hets. Inherits from *vg::GenotypePriorCalculator* **Public Functions**

virtual vg::FixedGenotypePriorCalculator::**~FixedGenotypePriorCalculator**()

double vg::FixedGenotypePriorCalculator::**calculate_log_prior**(**const** *Genotype &genotype*)

Return the log prior of the given genotype.

TODO: ploidy priors on nested sites???

Public Members

double vg::FixedGenotypePriorCalculator::**homozygous_prior_ln**

double vg::FixedGenotypePriorCalculator::**heterozygous_prior_ln**
class *#include <flow_sort.hpp>***Public Functions**

vg::FlowSort::**FlowSort**(*VG &vg*)

std::unique_ptr<list<*NodeTraversal*>> vg::FlowSort::**max_flow_sort**(**const** string *&ref_name*, bool *isGrooming* = true)

void vg::FlowSort::**fast_linear_sort**(**const** string *&ref_name*, bool *isGrooming* = true)

void vg::FlowSort::**flow_sort_nodes**(list<*NodeTraversal*> *&sorted_nodes*, **const** string *&ref_name*, bool *isGrooming*)

int vg::FlowSort::**get_node_degree**(*WeightedGraph &wg*, *id_t node_id*)

void vg::FlowSort::**update_in_out_edges**(*EdgeMapping &edges_in*, *EdgeMapping &edges_out*, *Edge *e*)

void vg::FlowSort::**erase_in_out_edges**(*EdgeMapping &edges_in*, *EdgeMapping &edges_out*, *Edge *e*)

void vg::FlowSort::**reverse_edge**(*Edge* *&*e*)

void vg::FlowSort::**reverse_from_start_to_end_edge**(*Edge* *&*e*)

id_t vg::FlowSort::**from_simple_reverse**(*Edge* *&*e*)

id_t vg::FlowSort::**from_simple_reverse_orientation**(*Edge* *&*e*)

id_t vg::FlowSort::**to_simple_reverse**(*Edge* *&*e*)

id_t vg::FlowSort::**to_simple_reverse_orientation**(*Edge* *&*e*)

vector<set<*id_t*>> vg::FlowSort::**get_cc_in_wg**(*EdgeMapping* &*edges_in*, *EdgeMapping* &*edges_out*, **const** set<*id_t*> &*all_nodes*, *id_t start_ref_node*)

void vg::FlowSort::**groom_components**(*EdgeMapping* &*edges_in*, *EdgeMapping* &*edges_out*, set<*id_t*> &*isolated_nodes*, set<*id_t*> &*main_nodes*, map<*id_t*, set<*Edge* *>> &*minus_start*, map<*id_t*, set<*Edge* *>> &*minus_end*)

id_t vg::FlowSort::**get_next_node_recalc_degrees**(*WeightedGraph* &*wg*, *std*::vector<*std*::set<*id_t*>> &*degrees*, *std*::set<*id_t*> &*sources*, *id_t node*)

id_t vg::FlowSort::**find_max_node**(*std*::vector<*std*::set<*id_t*>> *nodes_degree*)

bool vg::FlowSort::**bfs**(set<*id_t*> &*nodes*, map<*id_t*, map<*id_t*, int>> &*edge_weight*, *id_t s*, *id_t t*, map<*id_t*, *id_t*> &*parent*)

void vg::FlowSort::**dfs**(set<*id_t*> &*nodes*, *id_t s*, set<*id_t*> &*visited*, map<*id_t*, map<*id_t*, int>> &*edge_weight*)

void vg::FlowSort::**find_in_out_web**(list<*NodeTraversal*> &*sorted_nodes*, *Growth* &*in_out_growth*, *WeightedGraph* &*weighted_graph*, set<*id_t*> &*unsorted_nodes*, *id_t start_node*, bool *in_out*, int *count*)

void vg::FlowSort::**process_in_out_growth**(*EdgeMapping* &*edges_out_nodes*, *id_t current_id*, *Growth* &*in_out_growth*, *WeightedGraph* &*weighted_graph*, set<*id_t*> &*visited*, list<*NodeTraversal*> &*sorted_nodes*, bool *reverse*, set<*id_t*> &*unsorted_nodes*, bool *in_out*, int *count*)

void vg::FlowSort::**mark_dfs**(*EdgeMapping* &*graph_matrix*, *id_t s*, set<*id_t*> &*new_nodes*, set<*id_t*> &*visited*, bool *reverse*, set<*id_t*> &*nodes*, set<*id_t*> &*backbone*)

vector<pair<*id_t*, *id_t*>> vg::FlowSort::**min_cut**(map<*id_t*, map<*id_t*, int>> &*graph_weight*, set<*id_t*> &*nodes*, *id_t s*, *id_t t*, *EdgeMapping* &*edges_out_nodes*, set<*Edge* *> &*in_joins*)

void vg::FlowSort::**remove_edge**(*EdgeMapping* &*nodes_to_edges*, *id_t node*, *id_t to*, bool *reverse*)

Public Static Attributes

const size_t vg::FlowSort::**DEFAULT_PATH_WEIGHT**

Private Members

VG &vg::FlowSort::**vg**

class *#include <mapper.hpp>*

Public Functions

vg::FragmentLengthDistribution::**FragmentLengthDistribution**(size_t *maximum_sample_size*, size_t *reestimation_frequency*, double *robust_estimation_fraction*)

Initialize distribution

Args: maximum_sample_size sample size at which reestimation stops reestimation_frequency update running estimate after this many samples robust_estimation_fraction robustly estimate using this fraction of samples

vg::FragmentLengthDistribution::**FragmentLengthDistribution**(void)

vg::FragmentLengthDistribution::**~FragmentLengthDistribution**()

void vg::FragmentLengthDistribution::**force_parameters**(double *mean*, double *stddev*)

Instead of estimating anything, just use these parameters.

void vg::FragmentLengthDistribution::**register_fragment_length**(int64_t *length*)

Record an observed fragment length.

double vg::FragmentLengthDistribution::**mean**() **const**

Robust mean of the distribution observed so far.

double vg::FragmentLengthDistribution::**stdev**() **const**

Robust standard deviation of the distribution observed so far.

bool vg::FragmentLengthDistribution::**is_finalized**() **const**

Returns true if the maximum sample size has been reached, which finalizes the distribution estimate

size_t vg::FragmentLengthDistribution::**max_sample_size**() **const**

Returns the max sample size up to which the distribution will continue to reestimate parameters

size_t vg::FragmentLengthDistribution::**curr_sample_size**() **const**

Returns the number of samples that have been collected so far.

multiset<double>::const_iterator vg::FragmentLengthDistribution::**measurements_begin**() **const**

Begin iterator to the measurements that the distribution has used to estimate the parameters

multiset<double>::const_iterator vg::FragmentLengthDistribution::**measurements_end**() **const**

End iterator to the measurements that the distribution has used to estimate the parameters

Private Functions

void vg::FragmentLengthDistribution::**estimate_distribution**()

Private Members

multiset<double> vg::FragmentLengthDistribution::**lengths**

bool vg::FragmentLengthDistribution::**is_fixed**

double vg::FragmentLengthDistribution::**robust_estimation_fraction**

size_t vg::FragmentLengthDistribution::**maximum_sample_size**

size_t vg::FragmentLengthDistribution::**reestimation_frequency**

double vg::FragmentLengthDistribution::**mu**

double vg::FragmentLengthDistribution::**sigma**

class *#include <mapper.hpp>* Keeps track of statistics about fragment length within the *Mapper* class. Belongs to a single thread. **Public Functions**

void vg::FragmentLengthStatistics::**record_fragment_configuration**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*, *Mapper* **mapper*)

string vg::FragmentLengthStatistics::**fragment_model_str**(void)

void vg::FragmentLengthStatistics::**save_frag_lens_to_alns**(*Alignment* &*aln1*, *Alignment* &*aln2*, **const** map<string, int64_t> &*approx_frag_lengths*, bool *is_consistent*)

double vg::FragmentLengthStatistics::**fragment_length_stdev**(void)

double vg::FragmentLengthStatistics::**fragment_length_mean**(void)

double vg::FragmentLengthStatistics::**fragment_length_pdf**(double *length*)

double vg::FragmentLengthStatistics::**fragment_length_pval**(double *length*)

bool vg::FragmentLengthStatistics::**fragment_orientation**(void)

bool vg::FragmentLengthStatistics::**fragment_direction**(void)

Public Members

double vg::FragmentLengthStatistics::**cached_fragment_length_mean**

double vg::FragmentLengthStatistics::**cached_fragment_length_stdev**

bool vg::FragmentLengthStatistics::**cached_fragment_orientation_same**

bool vg::FragmentLengthStatistics::**cached_fragment_direction**

int64_t vg::FragmentLengthStatistics::**since_last_fragment_length_estimate**

int64_t vg::FragmentLengthStatistics::**fragment_model_update_interval**

deque<double> vg::FragmentLengthStatistics::**fragment_lengths**

deque<bool> vg::FragmentLengthStatistics::**fragment_orientations**

deque<bool> vg::FragmentLengthStatistics::**fragment_directions**

int64_t vg::FragmentLengthStatistics::**fragment_max**

int64_t vg::FragmentLengthStatistics::**fragment_size**

double vg::FragmentLengthStatistics::**fragment_sigma**

int64_t vg::FragmentLengthStatistics::**fragment_length_cache_size**

float vg::FragmentLengthStatistics::**perfect_pair_identity_threshold**

bool vg::FragmentLengthStatistics::**fixed_fragment_model**

class *#include <gamsorter.hpp>*

Public Functions

void GAMSorter::**sort**(vector<Alignment> &*alns*)

void GAMSorter::**paired_sort**(string *gamfile*)

void GAMSorter::**write_temp**(vector<Alignment> &*alns*)

void GAMSorter::**stream_sort**(string *gamfile*)

void GAMSorter::**dumb_sort**(string *gamfile*)

bool GAMSorter::**min_aln_first**(Alignment &*a*, Alignment &*b*)

Position GAMSorter::**get_min_position**(Alignment *a*)

Position GAMSorter::**get_min_position**(Path *p*)

void GAMSorter::**write_index**(string *gamfile*, string *outfile*, bool *isSorted* = false)

bool GAMSorter::**equal_to**(Position *a*, Position *b*)

bool GAMSorter::**less_than**(Position *a*, Position *b*)

bool GAMSorter::**greater_than**(Position *a*, Position *b*)

Private Members

map<int, int> vg::GAMSorter::**split_to_split_size**

vector<string> vg::GAMSorter::**tmp_filenames**

vector<ifstream *> vg::GAMSorter::**tmp_files**

int vg::GAMSorter::**max_buf_size**

unordered_map<string, *Alignment*> `vg::GAMSorter::`**`pairs`**

We want to keep pairs together, with the lowest-coordinate pair coming first. If one read is unmapped, it follows its partner in the sorted GAM file. Pairs with two unmapped reads are considered the highest-mapped in the graph, i.e. they come at the end of a sorted GAM file.

Since we can't hash alignments, we store both mates in the unordered map by name.

unordered_map<string, pair<*Alignment*, *Alignment*>> `vg::GAMSorter::`**`paired_pairs`**

class *#include <genome_state.hpp>* Define a way to represent a phased set of haplotypes on a graph that is under consideration as a variant calling solution. There should only be one of these for any genotyping run; operations all mutate the single copy and, if you don't like the result, can be rolled back by doing the reverse mutation. **Public Functions**

void `vg::GenomeState::`**`dump`**`()` **const**

Dump internal state to cerr.

`vg::GenomeState::`**`GenomeState`**`(`**const** *SnarlManager* &*manager*, **const** *HandleGraph* **graph*, **const** unordered_set<pair<**const** *Snarl* *, **const** *Snarl* *>> *telomeres*`)`

Make a new *GenomeState* on the given *SnarlManager*, manageing snarls in the given graph, with the given telomere pairs. Each telomere can appear in only one pair.

const *NetGraph* *`vg::GenomeState::`**`get_net_graph`**`(`**const** *Snarl* **snarl*`)`

DeleteHaplotypeCommand `vg::GenomeState::`**`append_haplotype`**`(`**const** *AppendHaplotypeCommand* &*c*`)`

Create a haplotype and return a command to delete it.

SwapHaplotypesCommand `vg::GenomeState::`**`swap_haplotypes`**`(`**const** *SwapHaplotypesCommand* &*c*`)`

Swap two haplotypes and return a command to swap them back.

ReplaceLocalHaplotypeCommand `vg::GenomeState::`**`replace_snarl_haplotype`**`(`**const** *ReplaceSnarlHaplotypeCommand* &*c*`)`

Replace part(s) of some haplotype(s) with other material.

DeleteHaplotypeCommand `vg::GenomeState::`**`insert_haplotype`**`(`**const** *InsertHaplotypeCommand* &*c*`)`

Insert a haplotype and return a command to delete it.

InsertHaplotypeCommand `vg::GenomeState::`**`delete_haplotype`**`(`**const** *DeleteHaplotypeCommand* &*c*`)`

Delete a haplotype and return a command to insert it.

ReplaceLocalHaplotypeCommand `vg::GenomeState::`**`replace_local_haplotype`**`(`**const** *ReplaceLocalHaplotypeCommand* &*c*`)`

Replace part(s) of some haplotype(s) with other material.

GenomeStateCommand *vg::GenomeState::**execute**(*GenomeStateCommand* *command)
Execute a command. Return a new heap-allocated command that undoes the command being executed. Frees the passed command. TODO: does that make sense?

size_t vg::GenomeState::**count_haplotypes**(**const** pair<**const** *Snarl* *, **const** *Snarl* *> &*telomere_pair*) **const**
Count the number of haplotypes connecting the given pair of telomeres. It must be a pair of telomeres actually passed during construction.

size_t vg::GenomeState::**count_haplotypes**(**const** *Snarl* *snarl) **const**
Count the number of haplotypes within a given snarl.

void vg::GenomeState::**trace_haplotype**(**const** pair<**const** *Snarl* *, **const** *Snarl* *> &*telomere_pair*, size_t *overall_lane*, **const** function<void) **const** *handle_t*& > &*iteratee* **const**Trace the haplotype starting at the given start telomere snarl with the given overall lane. Calls the callback with each backing *HandleGraph* handle.

Protected Functions

void vg::GenomeState::**insert_handles**(**const** vector<*handle_t*> &*to_add*, unordered_map<**const** *Snarl* *, vector<size_t>> &*lanes_added*, size_t *top_lane* = numeric_limits< size_t >::max())
We have a generic stack-based handle-vector-to-per-snarl-haplotypes insertion walker function. The handles to add have to span one or more entire snarls, and we specify the lane in the spanned snarls to put them in.

Protected Attributes

unordered_set<pair<**const** *Snarl* *, **const** *Snarl* *>> vg::GenomeState::**telomeres**
We keep track of pairs of telomere snarls. The haplotypes we work on connect a left telomere and its corresponding right telomere. We can traverse the snarl decomposition from one telomere to the other either following chains or pollowing paths inside of some snarl we are in. The start snarl of a pair must be in its local forward orientation.

unordered_map<**const** *Snarl* *, *NetGraph*> vg::GenomeState::**net_graphs**
We precompute all the net graphs and keep them around.

unordered_map<**const** *Snarl* *, *SnarlState*> vg::GenomeState::**state**
We have a state for every snarl, which depends on the corresponding net graph.

const *HandleGraph* *vg::GenomeState::**backing_graph**
We remember the backing graph, because sometimes we need to translate from backing graph handles to IDs and orientations to look up snarls.

const *SnarlManager* &vg::GenomeState::**manager**
We keep a reference to the *SnarlManager* that knows what children are where and which snarl we should look at next after leaving a previous snarl.

struct *#include <genome_state.hpp>* Represents a modification of a *GenomeState*. We use a command pattern to enable undo-ability. Applying a command always returns a command that will undo what you did. Subclassed by *vg::AppendHaplotypeCommand*, *vg::DeleteHaplotypeCommand*, *vg::InsertHaplotypeCommand*, *vg::ReplaceLocalHaplotypeCommand*, *vg::ReplaceSnarlHaplotypeCommand*, *vg::SwapHaplotypesCommand*

Public Functions

virtual `vg::GenomeStateCommand::~GenomeStateCommand()`

virtual *GenomeStateCommand* *`vg::GenomeStateCommand::execute`*(*GenomeState* **const** *&state*) = 0Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

struct Describes a genotype at a particular locus.

Public Members

repeated<int32> `vg::Genotype::allele`
These refer to the offsets of the alleles in the *Locus* object.

bool `vg::Genotype::is_phased`

double `vg::Genotype::likelihood`

double `vg::Genotype::log_likelihood`
Likelihood natural logged.

double `vg::Genotype::log_prior`
Prior natural logged.

double `vg::Genotype::log_posterior`
Posterior natural logged (unnormalized).

class *#include <genotypekit.hpp>* Represents a strategy for calculating genotype likelihood for a (nested) Site. Polymorphic base class/interface.

Public Functions

virtual `vg::GenotypeLikelihoodCalculator::~GenotypeLikelihoodCalculator()`

virtual double vg::GenotypeLikelihoodCalculator::**calculate_log_likelihood**(**const** *Snarl* &*site*, **const** vector<*SnarlTraversal*> &*traversals*, **const** *Genotype* &*genotype*, **const** vector<vector<bool>> &*consistencies*, **const** vector<*Support*> &*supports*, **const** vector<*Alignment* *> &*reads*)

= 0Return the log likelihood of the given genotype.

class *#include <genotypekit.hpp>* Represents a strategy for assigning genotype priors. Polymorphic base class/interface. Subclassed by *vg::FixedGenotypePriorCalculator* **Public Functions**

virtual vg::GenotypePriorCalculator::**~GenotypePriorCalculator**()

virtual double vg::GenotypePriorCalculator::**calculate_log_prior**(**const** *Genotype* &*genotype*)

= 0Return the log prior of the given genotype.

TODO: ploidy priors on nested sites???

class *#include <genotyper.hpp>* Class to hold on to genotyping parameters and genotyping functions. **Public Functions**

void vg::Genotyper::**run**(*AugmentedGraph* &*graph*, vector<*Alignment*> &*alignments*, ostream &*out*, string *ref_path_name*, string *contig_name* = "", string *sample_name* = "", string *augmented_file_name* = "", bool *subset_graph* = false, bool *show_progress* = false, bool *output_vcf* = false, bool *output_json* = false, int *length_override* = 0, int *variant_offset* = 0)

int vg::Genotyper::**alignment_qual_score**(*VG* &*graph*, **const** *Snarl* *snarl, **const** *Alignment* &*alignment*)

Given an *Alignment* and a *Snarl*, compute a phred score for the quality of the alignment's bases within the snarl overall (not counting the start and end nodes), which is supposed to be interpretable as the probability

that the call of the sequence is wrong (to the degree that it would no longer support the alleles it appears to support).

In practice we're just going to average the quality scores for all the bases interior to the snarl (i.e. not counting the start and end nodes).

If the alignment doesn't have base qualities, or no qualities are available for bases internal to the snarl, returns a default value.

list<*Mapping*> `vg::Genotyper::`**`get_traversal_of_snarl`**(*VG &graph*, **const** *Snarl *snarl*, **const** *SnarlManager &manager*, **const** *Path &path*)

Given a path (which may run either direction through a snarl, or not touch the ends at all), collect a list of NodeTraversals in order for the part of the path that is inside the snarl, in the same orientation as the path.

string `vg::Genotyper::`**`traversals_to_string`**(*VG &graph*, **const** list<*Mapping*> *&path*)

Make a list of NodeTraversals into the string they represent.

vector<list<*Mapping*>> `vg::Genotyper::`**`get_paths_through_snarl`**(*VG &graph*, **const** *Snarl *snarl*, **const** *SnarlManager &manager*, **const** map<string, *Alignment* *> *&reads_by_name*)

For the given snarl, emit all subpaths with unique sequences that run from start to end, out of the paths in the graph. Uses the map of reads by name to determine if a path is a read or a real named path. *Paths* through the snarl supported only by reads are subject to a min recurrence count, while those supported by actual embedded named paths are not.

string `vg::Genotyper::`**`get_qualities_in_snarl`**(*VG &graph*, **const** *Snarl *snarl*, **const** *Alignment &alignment*)

Get all the quality values in the alignment between the start and end nodes of a snarl. Handles alignments that enter the snarl from the end, and alignments that never make it through the snarl.

If we run out of qualities, or qualities aren't present, returns no qualities.

If an alignment goes through the snarl multipe times, we get all the qualities from when it is in the snarl.

Does not return qualities on the start and end nodes. May return an empty string.

map<*Alignment* *, vector<*Genotyper*::*Affinity*>> vg::Genotyper::**get_affinities**(*VG* &*graph*, **const** map<string, *Alignment* *> &*reads_by_name*, **const** *Snarl* **snarl*, **const** *SnarlManager* &*manager*, **const** vector<list<*Mapping*>> &*superbubble_paths*)

Get the affinity of all the reads relevant to the superbubble to all the paths through the superbubble.

Affinity is a double out of 1.0. Higher is better.

map<*Alignment* *, vector<*Genotyper*::*Affinity*>> vg::Genotyper::**get_affinities_fast**(*VG* &*graph*, **const** map<string, *Alignment* *> &*reads_by_name*, **const** *Snarl* **snarl*, **const** *SnarlManager* &*manager*, **const** vector<list<*Mapping*>> &*superbubble_paths*, bool *allow_internal_alignments* = false)

Get affinities as above but using only string comparison instead of alignment. Affinities are 0 for mismatch and 1 for a perfect match.

Locus vg::Genotyper::**genotype_snarl**(*VG &graph*, **const** *Snarl *snarl*, **const** vector<list<*Mapping*>> &*superbubble_paths*, **const** map<*Alignment* *, vector<*Affinity*>> &*affinities*)

Compute annotated genotype from affinities and superbubble paths. Needs access to the graph so it can chop up the alignments, which requires node sizes.

double vg::Genotyper::**get_genotype_log_likelihood**(*VG &graph*, **const** *Snarl *snarl*, **const** vector<int> &*genotype*, **const** vector<pair<*Alignment* *, vector<*Affinity*>>> &*alignment_consistency*)

Compute the probability of the observed alignments given the genotype.

Takes a genotype as a vector of allele numbers, and support data as a collection of pairs of Alignments and vectors of bools marking whether each alignment is consistent with each allele.

Alignments should have had their quality values trimmed down to just the part covering the superbubble.

Returns a natural log likelihood.

double vg::Genotyper::**get_genotype_log_prior**(**const** vector<int> &*genotype*)

Compute the prior probability of the given genotype.

Takes a genotype as a vector of allele numbers. It is not guaranteed that allele 0 corresponds to any notion of primary reference-ness.

Returns a natural log prior probability.

TODO: add in strand bias

vector<vcflib::Variant> vg::Genotyper::**locus_to_variant**(*VG &graph*, **const** *Snarl *snarl*, **const** *SnarlManager &manager*, **const** *PathIndex &index*, vcflib::VariantCallFile &*vcf*, **const** *Locus* &*locus*, **const** string &*sample_name* = "SAMPLE")

Make a VCFlib variant from a called *Locus*. Depends on an index of the reference path we want to call against.

Returns 0 or more variants we can articulate from the superbubble. Sometimes if we can't make a variant for the superbubble against the reference path, we'll emit 0 variants.

void vg::Genotyper::**write_vcf_header**(*std*::ostream &*stream*, **const** *std*::string &*sample_name*, **const** *std*::string &*contig_name*, size_t *contig_size*)

Make a VCF header

vcflib::VariantCallFile *vg::Genotyper::**start_vcf**(*std*::ostream &*stream*, **const** *PathIndex &index*, **const** string &*sample_name*, **const** string &*contig_name*, size_t *contig_size*)

Start VCF output to a stream. Returns a VCFlib VariantCallFile that needs to be deleted.

pair<pair<int64_t, int64_t>, bool> vg::Genotyper::**get_snarl_reference_bounds**(**const** *Snarl* *snarl, **const** *PathIndex* &index, **const** *HandleGraph* *graph)

Utility function for getting the reference bounds (start and past-end) of a snarl with relation to a given reference index in the given graph. Computes bounds of the variable region, not including the fixed start and end node lengths. Also returns whether the reference path goes through the snarl forwards (false) or backwards (true).

void vg::Genotyper::**report_snarl**(**const** *Snarl* *snarl, **const** *SnarlManager* &manager, **const** *PathIndex* *index, *VG* &graph)

Tell the statistics tracking code that a snarl exists. We can do things like count up the snarl length in the reference and so on. Called only once per snarl, but may be called on multiple threads simultaneously.

void vg::Genotyper::**report_snarl_traversal**(**const** *Snarl* *snarl, **const** *SnarlManager* &manager, **const** string &read_name, *VG* &graph)

Tell the statistics tracking code that a read traverses a snarl completely. May be called multiple times for a given read and snarl, and may be called in parallel.

void vg::Genotyper::**print_statistics**(ostream &out)

Print snarl statistics to the given stream.

void vg::Genotyper::**edge_allele_labels**(**const** *VG* &graph, **const** *Snarl* *snarl, **const** vector<list<*NodeTraversal*>> &superbubble_paths, unordered_map<pair<*NodeTraversal*, *NodeTraversal*>, unordered_set<size_t>, *hash_oriented_edge*> *out_edge_allele_sets)

void vg::Genotyper::**allele_ambiguity_log_probs**(**const** *VG* &graph, **const** *Snarl* *snarl, **const** vector<list<*NodeTraversal*>> &superbubble_paths, **const** unordered_map<pair<*NodeTraversal*, *NodeTraversal*>, unordered_set<size_t>, *hash_oriented_edge*> &edge_allele_sets, vector<unordered_map<vector<size_t>, double, *hash_ambiguous_allele_set*>> *out_allele_ambiguity_probs)

Public Members

size_t vg::Genotyper::**max_path_search_steps**

int vg::Genotyper::**unfold_max_length**

int vg::Genotyper::**dagify_steps**

double vg::Genotyper::**max_het_bias**

bool vg::Genotyper::**use_mapq**

bool vg::Genotyper::**realign_indels**

int vg::Genotyper::**default_sequence_quality**

int vg::Genotyper::**min_recurrence**

int vg::Genotyper::**min_consistent_per_strand**

double vg::Genotyper::**min_score_per_base**

double vg::Genotyper::**diploid_prior_logprob**

double vg::Genotyper::**het_prior_logprob**

double vg::Genotyper::**haploid_prior_logprob**

double vg::Genotyper::**deleted_prior_logprob**

double vg::Genotyper::**polyploid_prior_success_logprob**

Translator vg::Genotyper::**translator**

unordered_map<size_t, size_t> vg::Genotyper::**snarl_reference_length_histogram**

unordered_map<**const** *Snarl* *, set<string>> vg::Genotyper::**snarl_traversals**

unordered_set<**const** *Snarl* *> vg::Genotyper::**all_snarls**

Aligner vg::Genotyper::**normal_aligner**

QualAdjAligner vg::Genotyper::**quality_aligner**

Public Static Functions

bool vg::Genotyper::**mapping_enters_side**(**const** *Mapping* &*mapping*, **const** *handle_t* &*side*, **const** *HandleGraph* *graph)

Check if a mapping corresponds to the beginning or end of snarl by making sure it crosses the given side in the expected direction. The handle should be forward for the left side and reverse for the right side.

bool vg::Genotyper::**mapping_exits_side**(**const** *Mapping* &*mapping*, **const** *handle_t* &*side*, **const** *HandleGraph* *graph)

struct *Graphs* are collections of nodes and edges. They can represent subgraphs of larger graphs or be wholly-self-sufficient. Protobuf memory limits of 67108864 bytes mean we typically keep the size of them small generating graphs as collections of smaller subgraphs. **Public Members**

repeated<*Node*> vg::Graph::**node**

The *Node*s that make up the graph.

repeated<*Edge*> vg::Graph::**edge**

The *Edge*s that connect the *Node*s in the graph.

repeated<*Path*> vg::Graph::**path**
A set of named *Path*s that visit sequences of oriented *Node*s.

class *#include <graph_synchronizer.hpp>* Let threads get exclusive locks on subgraphs of a vg graph, for reading and editing. Whan a subgraph is locked, a copy is accessible through the lock object and the underlying graph can be edited (through the lock) without affecting any other locked subgraphs. A thread may only hold a lock on a single subgraph at a time. Trying to lock another subgraph while you already have a subgraph locked is likely to result in a deadlock. **Public Functions**

vg::GraphSynchronizer::**GraphSynchronizer**(*VG &graph*)
Create a *GraphSynchronizer* for synchronizing on parts of the given graph.

const string &vg::GraphSynchronizer::**get_path_sequence**(**const** string &*path_name*)
Since internally we keep PathIndexes for paths in the graph, we expose this method for getting the strings for paths.

void vg::GraphSynchronizer::**with_path_index**(**const** string &*path_name*, **const** function<void(**const** *PathIndex*&)> &*to_run*)We can actually let users run whatever function they want with an exclusive handle on a *PathIndex*, with the guarantee that the graph won't change while they're working.

Protected Functions

PathIndex &vg::GraphSynchronizer::**get_path_index**(**const** string &*path_name*)
Get the index for the given path name. *Lock* on the indexes and graph must be held already.

void vg::GraphSynchronizer::**update_path_indexes**(**const** vector<*Translation*> &*translations*)
Update all the path indexes according to the given translations. *Lock* on the indexes and graph must be held already.

Protected Attributes

VG &vg::GraphSynchronizer::**graph**
The graph we manage.

mutex vg::GraphSynchronizer::**whole_graph_lock**
We use this to lock the whole graph, for when we're exploring and trying to lock a context, or for when we're making an edit, or for when we're trying to lock a context, or for when we're working with the PathIndexes. It's only ever held during functions in this class or internal classes (monitor-style), so we don't need it to be a recursive mutex.

condition_variable vg::GraphSynchronizer::**wait_for_region**
We have one condition variable where we have blocked all the threads that are waiting to lock subgraphs but couldn't the first time because we ran into already locked nodes. When nodes get unlocked, we wake them all up, and they each grab the mutex and check, one at a time, to see if they can have all their nodes this time.

map<string, *PathIndex*> vg::GraphSynchronizer::**indexes**
We need indexes of all the paths that someone might want to use as a basis for locking. This holds a *PathIndex* for each path we touch by path name.

set<*id_t*> vg::GraphSynchronizer::**locked_nodes**
This holds all the node IDs that are currently locked by someone.

struct *#include <flow_sort.hpp>*Public Functions

vg::FlowSort::Growth::**Growth**()

Public Members

set<*id_t*> vg::FlowSort::Growth::**nodes**

set<*id_t*> vg::FlowSort::Growth::**backbone**

list<*id_t*> vg::FlowSort::Growth::**ref_path**
struct *#include <handle.hpp>* A handle is 8 (assuming id_t is still int64_t) opaque bytes. A handle refers to an oriented node. Two handles are equal iff they refer to the same orientation of the same node in the same graph. Handles have no ordering, but can be hashed. **Public Members**

char vg::handle_t::**data**[sizeof(id_t)]
class *#include <handle.hpp>* This is the interface that a graph that uses handles needs to support. It is also the interface that users should code against. Subclassed by *vg::MutableHandleGraph*, *vg::NetGraph*, *xg::XG*
Public Functions

virtual *handle_t* vg::HandleGraph::**get_handle**(**const** *id_t* &*node_id*, bool *is_reverse* = false) **const** = 0Look up the handle for the node with the given ID in the given orientation.

virtual *id_t* vg::HandleGraph::**get_id**(**const** *handle_t* &*handle*) **const** = 0Get the ID from a handle.

virtual bool vg::HandleGraph::**get_is_reverse**(**const** *handle_t* &*handle*) **const** = 0Get the orientation of a handle.

virtual *handle_t* vg::HandleGraph::**flip**(**const** *handle_t* &*handle*) **const** = 0Invert the orientation of a handle (potentially without getting its ID)

virtual size_t vg::HandleGraph::**get_length**(**const** *handle_t* &*handle*) **const** = 0Get the length of a node.

virtual string vg::HandleGraph::**get_sequence**(**const** *handle_t* &*handle*) **const** = 0Get the sequence of a node, presented in the handle's local forward orientation.

virtual bool vg::HandleGraph::**follow_edges**(**const** *handle_t* &*handle*, bool *go_left*, **const** function<bool(**const** *handle_t*&)> &*iteratee*) **const** = 0Loop 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.

virtual void vg::HandleGraph::**for_each_handle**(**const** function<bool(**const** *handle_t*&)> &*iteratee*) **const** = 0Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false.

virtual size_t vg::HandleGraph::**node_size**() **const** = 0Return the number of nodes in the graph TODO: can't be node_count because XG has a field named node_count.

template <typename T>

auto vg::HandleGraph::**follow_edges**(**const** *handle_t* &*handle*, bool *go_left*, T &&*iteratee*) **const**

Loop over all the handles to next/previous (right/left) nodes. Works with a callback that just takes all the handles and returns void. MUST be pulled into implementing classes with `using` in order to work!

template <typename T>

auto vg::HandleGraph::**for_each_handle**(T &&*iteratee*) **const**

Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Works with void-returning iteratees. MUST be pulled into implementing classes with `using` in order to work!

handle_t vg::HandleGraph::**get_handle**(**const** *Visit* &*visit*) **const**

Get a handle from a *Visit* Protobuf object. Must be using'd to avoid shadowing.

Visit vg::HandleGraph::**to_visit**(**const** *handle_t* &*handle*) **const**

Get a Protobuf *Visit* from a handle.

handle_t vg::HandleGraph::**forward**(**const** *handle_t* &*handle*) **const**

Get the locally forward version of a handle.

pair<*handle_t*, *handle_t*> vg::HandleGraph::**edge_handle**(**const** *handle_t* &*left*, **const** *handle_t* &*right*) **const**

class *#include <phased_genome.hpp>* Specialized linked list that tracks a walk through the variation graph and maintains an index of sites. **Public Functions**

vg::PhasedGenome::Haplotype::**Haplotype**(*NodeTraversal node_traversal*)

Construct a haplotype with a single node.

template <typename NodeTraversalIterator>

vg::PhasedGenome::Haplotype::**Haplotype**(NodeTraversalIterator *first*, NodeTraversalIterator *last*)

Construct a haplotype with an iterator that yields NodeTraversals.

vg::PhasedGenome::Haplotype::**~Haplotype**()

PhasedGenome::*HaplotypeNode* *vg::PhasedGenome::Haplotype::**append_left**(*NodeTraversal node_traversal*)

Add a haplotype node for this node traversal to left end of haplotye and return new node. Does not maintain indices; intended for use by the *PhasedGenome* for initial haplotype build.

PhasedGenome::*HaplotypeNode* *vg::PhasedGenome::Haplotype::**append_right**(*NodeTraversal node_traversal*)

Add a haplotype node for this node traversal to right end of haplotye and return new node. Does not maintain indices; intended for use by the *PhasedGenome* for initial haplotype build.

Private Members

PhasedGenome::*HaplotypeNode* *vg::PhasedGenome::Haplotype::**left_telomere_node**

Leftmost node in walk.

PhasedGenome::*HaplotypeNode* *vg::PhasedGenome::Haplotype::**right_telomere_node**

Rightmost node in walk.

unordered_map<**const** *Snarl* *, pair<*HaplotypeNode* *, *HaplotypeNode* *>> vg::PhasedGenome::Haplotype::**sites**

Index of the location in the haplotype of nested sites. Locations of sites are stored as the nodes on haplotype that correspond to the start and end node of the site. The pair of haplotype nodes is stored in left-to-right order along the haplotype (i.e. .first->prev and .second->next are outside the bubble).

Friends

friend vg::PhasedGenome::Haplotype::PhasedGenome

friend vg::PhasedGenome::Haplotype::HaplotypeNode
struct *#include <phased_genome.hpp>* A node in walk through the graph taken by a haplotype. **Public Functions**

vg::PhasedGenome::HaplotypeNode::**HaplotypeNode**(*NodeTraversal node_traversal*, *HaplotypeNode *next*, *HaplotypeNode *prev*)
Constructor.

vg::PhasedGenome::HaplotypeNode::**~HaplotypeNode**()
Destructor.

Public Members

NodeTraversal vg::PhasedGenome::HaplotypeNode::**node_traversal**
Node and strand.

HaplotypeNode *vg::PhasedGenome::HaplotypeNode::**next**
Next node in walk.

HaplotypeNode *vg::PhasedGenome::HaplotypeNode::**prev**
Previous node in walk.

template <>
struct
class std::**hash<const** *vg*::*Snarl***>**
#include <snarls.hpp> hash function for Snarls

Public Functions

size_t std::hash::**operator()**(**const** *vg*::*Snarl* &*snarl*) **const**

template <typename A, **typename** B>
struct
class std::**hash**<pair<A, B>>
#include <hash_map_set.hpp>

Public Functions

size_t std::hash::**operator()**(**const** pair<A, B> &*x*) **const**

size_t std::hash::**operator()**(**const** pair<A, B> &*x*) **const**

template <typename... *TT*>
struct
class std::**hash**<*std*::tuple<TT...>>
#include <hash_map.hpp>

Public Functions

size_t std::hash::**operator()**(*std*::tuple<TT...> **const** &*tt*) **const**

template <>
struct
class std::**hash**<*vg*::*handle_t*>
#include <*handle.hpp*> Define hashes for handles.

Public Functions

size_t std::hash::**operator()**(**const** *vg*::*handle_t* &*handle*) **const**

template <>
struct
class std::**hash**<*vg*::*NodeSide*>
#include <*nodeside.hpp*> Hash functor to hash NodeSides. We need to implement a hash function for these if we want to be able to use them in keys in hash maps.

Public Functions

size_t std::hash::**operator()**(**const** *vg*::*NodeSide* &*item*) **const**
Produce a hash of a NodeSide.

template <>
struct
class std::**hash**<*vg*::*NodeTraversal*>
#include <*nodetraversal.hpp*> hash function for NodeTraversals

Public Functions

size_t std::hash::**operator()**(**const** *vg*::*NodeTraversal* &*trav*) **const**
struct *#include* <*genotyper.hpp*>**Public Functions**

size_t vg::Genotyper::hash_ambiguous_allele_set::**operator()**(**const** vector<size_t> &*ambiguous_set*) **const**

template <typename *K*, typename *V*>
class *#include* <*hash_map.hpp*> Inherits from google::dense_hash_map< K, V >

Public Functions

vg::hash_map::**hash_map**()

template <**typename** K, **typename** V>
class *#include* <*hash_map_set.hpp*> Inherits from google::sparse_hash_map< K, V >

Public Functions

xg::hash_map::**hash_map**()

template <**typename** K, **typename** V>
class
class vg::**hash_map**<K *, V>
#include <*hash_map.hpp*> Inherits from google::dense_hash_map< K *, V >

Public Functions

vg::hash_map::**hash_map**()
template <**typename** K, **typename** V>
class
class xg::**hash_map**<K *, V>
#include <*hash_map_set.hpp*> Inherits from google::sparse_hash_map< K *, V >

Public Functions

xg::hash_map::**hash_map**()
struct *#include* <*genotyper.hpp*>**Public Functions**

size_t vg::Genotyper::hash_node_traversal::**operator()**(**const** *NodeTraversal* &*node_traversal*) **const**
struct *#include* <*genotyper.hpp*>**Public Functions**

size_t vg::Genotyper::hash_oriented_edge::**operator()**(**const** pair<**const** *NodeTraversal*, **const** *NodeTraversal*> &*edge*) **const**
template <**typename** K>
class *#include* <*hash_map_set.hpp*> Inherits from google::sparse_hash_set< K >

Public Functions

xg::hash_set::**hash_set**()
template <**typename** K>
class
class xg::**hash_set**<K *>
#include <*hash_map_set.hpp*> Inherits from google::sparse_hash_set< K * >

Public Functions

xg::hash_set::**hash_set**()
class *#include* <*homogenizer.hpp*>**Public Functions**

void Homogenizer::**homogenize**(*vg*::*VG* *graph, *xg*::*XG* *xindex, gcsa::GCSA *gcsa_index, gcsa::LCPArray *lcp_index, Paths *p*, int *kmer_size*)
Locates tips in the graph and tries to generate a single edge / node to represent them. This edge is then added, the offending sequences are remapped, and the process is repeated until the graph becomes stable.

void Homogenizer::**homogenize**(*vg*::*VG* *graph, *xg*::*XG* *xindex, gcsa::GCSA *gcsa_index, gcsa::LCPArray *lcp_index, *vg*::*Index* *reads_index*)

Private Functions

vector<*vg*::*id_t*> `Homogenizer::`**`find_tips`**(*vg*::*VG* **graph*)
Find tips (nodes with an indegree/outdegree of 0 in the graph

vector<*vg*::*id_t*> `Homogenizer::`**`find_non_ref_tips`**(*vg*::*VG* **graph*)
Find non-ref tips

int `vg::Homogenizer::`**`remap`**(vector<*Alignment*> *reads*, *vg*::*VG graph*)
remap a set of Alignments to the graph

void `Homogenizer::`**`cut_tips`**(*vg*::*VG* **graph*)
Remove all tips from the graph. WARNING: may cut head/tail nodes.

void `Homogenizer::`**`cut_tips`**(vector<id_t> *tip_ids*, *vg*::*VG* **graph*)
Remove specific nodes and their edges from the graph

void `Homogenizer::`**`cut_nonref_tips`**(*vg*::*VG* **graph*)
Remove non-reference tips from the graph.

Private Members

Translator `vg::Homogenizer::`**`translator`**
struct *#include <utility.hpp>*

Public Functions

`vg::IncrementIter::`**`IncrementIter`**(size_t *number*)

IncrementIter &`vg::IncrementIter::`**`operator=`**(**const** *IncrementIter* &*other*)

bool `vg::IncrementIter::`**`operator==`**(**const** *IncrementIter* &*other*) **const**

bool `vg::IncrementIter::`**`operator!=`**(**const** *IncrementIter* &*other*) **const**

IncrementIter `vg::IncrementIter::`**`operator++`**()

IncrementIter `vg::IncrementIter::`**`operator++`**(int)

size_t `vg::IncrementIter::`**`operator*`**()

Private Members

size_t `vg::IncrementIter::`**`current`**
class *#include <index.hpp>*

Public Functions

`vg::Index::`**`Index`**(void)

`vg::Index::`**`Index`**(string &*name*)

`vg::Index::`**`~Index`**(void)

rocksdb::Options `vg::Index::`**`GetOptions`**(bool *read_only*)

void `vg::Index::`**`open`**(**const** *std*::string &*dir*, bool *read_only* = false)

void vg::Index::**open_read_only**(string &*dir*)

void vg::Index::**open_for_write**(string &*dir*)

void vg::Index::**open_for_bulk_load**(string &*dir*)

void vg::Index::**reset_options**(void)

void vg::Index::**flush**(void)

void vg::Index::**compact**(void)

void vg::Index::**close**(void)

void vg::Index::**load_graph**(*VG* &*graph*)

void vg::Index::**dump**(*std*::ostream &*out*)

void vg::Index::**for_all**(*std*::function<void) string&, string&
> *lambda*

void vg::Index::**for_range**(string &*key_start*, string &*key_end*, *std*::function<void) string&, string&
> *lambda*

void vg::Index::**put_node**(**const** *Node* **node*)

void vg::Index::**put_edge**(**const** *Edge* **edge*)

void vg::Index::**batch_node**(**const** *Node* **node*, rocksdb::WriteBatch &*batch*)

void vg::Index::**batch_edge**(**const** *Edge* **edge*, rocksdb::WriteBatch &*batch*)

void vg::Index::**put_kmer**(**const** string &*kmer*, **const** int64_t *id*, **const** int32_t *pos*)

void vg::Index::**batch_kmer**(**const** string &*kmer*, **const** int64_t *id*, **const** int32_t *pos*, rocksdb::WriteBatch &*batch*)

void vg::Index::**put_metadata**(**const** string &*tag*, **const** string &*data*)

void vg::Index::**put_node_path**(int64_t *node_id*, int64_t *path_id*, int64_t *path_pos*, bool *backward*, **const** *Mapping* &*mapping*)

void vg::Index::**put_path_position**(int64_t *path_id*, int64_t *path_pos*, bool *backward*, int64_t *node_id*, **const** *Mapping* &*mapping*)

void vg::Index::**put_mapping**(**const** *Mapping* &*mapping*)

void vg::Index::**put_alignment**(**const** *Alignment* &*alignment*)

void vg::Index::**put_base**(int64_t *aln_id*, **const** *Alignment* &*alignment*)

void vg::Index::**put_traversal**(int64_t *aln_id*, **const** *Mapping* &*mapping*)

void vg::Index::**cross_alignment**(int64_t *aln_id*, **const** *Alignment* &*alignment*)

rocksdb::Status vg::Index::**get_node**(int64_t *id*, *Node* &*node*)

rocksdb::Status vg::Index::**get_edge**(int64_t *from*, bool *from_start*, int64_t *to*, bool *to_end*, *Edge* &*edge*)

rocksdb::Status vg::Index::**get_metadata**(**const** string &*key*, string &*data*)

int vg::Index::**get_node_path**(int64_t *node_id*, int64_t *path_id*, int64_t &*path_pos*, bool &*backward*, *Mapping* &*mapping*)

void vg::Index::**get_mappings**(int64_t *node_id*, vector<*Mapping*> &*mappings*)

void vg::Index::**get_alignments**(int64_t *node_id*, vector<*Alignment*> &*alignments*)

void vg::Index::**get_alignments**(int64_t *id1*, int64_t *id2*, vector<*Alignment*> &*alignments*)

void vg::Index::**for_alignment_in_range**(int64_t *id1*, int64_t *id2*, *std*::function<void) **const** *Alignment*&
> *lambda*

void vg::Index::**for_alignment_to_node**(int64_t *node_id*, *std*::function<void) **const** *Alignment*&
> *lambda*

void vg::Index::**for_alignment_to_nodes**(**const** vector<int64_t> &*ids*, *std*::function<void) **const** *Alignment*&
> *lambda*

void vg::Index::**for_base_alignments**(**const** set<int64_t> &*aln_ids*, *std*::function<void) **const** *Alignment*&
> *lambda*

const string vg::Index::**key_for_node**(int64_t *id*)

const string vg::Index::**key_for_edge_on_start**(int64_t *node_id*, int64_t *other*, bool *backward*)

const string vg::Index::**key_for_edge_on_end**(int64_t *node_id*, int64_t *other*, bool *backward*)

const string vg::Index::**key_prefix_for_edges_on_node_start**(int64_t *node*)

const string vg::Index::**key_prefix_for_edges_on_node_end**(int64_t *node*)

const string vg::Index::**key_for_kmer**(**const** string &*kmer*, int64_t *id*)

const string vg::Index::**key_prefix_for_kmer**(**const** string &*kmer*)

const string vg::Index::**key_for_metadata**(**const** string &*tag*)

const string vg::Index::**key_for_path_position**(int64_t *path_id*, int64_t *path_pos*, bool *backward*, int64_t *node_id*)

const string vg::Index::**key_for_node_path_position**(int64_t *node_id*, int64_t *path_id*, int64_t *path_pos*, bool *backward*)

const string vg::Index::**key_prefix_for_node_path**(int64_t *node_id*, int64_t *path_id*)

const string vg::Index::**key_for_mapping_prefix**(int64_t *node_id*)

const string vg::Index::**key_for_mapping**(**const** *Mapping* &*mapping*)

const string vg::Index::**key_for_alignment_prefix**(int64_t *node_id*)

const string vg::Index::**key_for_alignment**(**const** *Alignment* &*alignment*)

const string vg::Index::**key_for_base**(int64_t *aln_id*)

const string vg::Index::**key_prefix_for_traversal**(int64_t *node_id*)

const string vg::Index::**key_for_traversal**(int64_t *aln_id*, **const** *Mapping* &*mapping*)

void vg::Index::**parse_node**(**const** string &*key*, **const** string &*value*, int64_t &*id*, *Node* &*node*)

void vg::Index::**parse_edge**(**const** string &*key*, **const** string &*value*, char &*type*, int64_t &*id1*, int64_t &*id2*, *Edge* &*edge*)

void vg::Index::**parse_edge**(**const** string &*key*, char &*type*, int64_t &*node_id*, int64_t &*other_id*, bool &*backward*)

void vg::Index::**parse_kmer**(**const** string &*key*, **const** string &*value*, string &*kmer*, int64_t &*id*, int32_t &*pos*)

void vg::Index::**parse_node_path**(**const** string &*key*, **const** string &*value*, int64_t &*node_id*, int64_t &*path_id*, int64_t &*path_pos*, bool &*backward*, *Mapping* &*mapping*)

void vg::Index::**parse_path_position**(**const** string &*key*, **const** string &*value*, int64_t &*path_id*, int64_t &*path_pos*, bool &*backward*, int64_t &*node_id*, *Mapping* &*mapping*)

void vg::Index::**parse_mapping**(**const** string &*key*, **const** string &*value*, int64_t &*node_id*, uint64_t &*nonce*, *Mapping* &*mapping*)

void vg::Index::**parse_alignment**(**const** string &*key*, **const** string &*value*, int64_t &*node_id*, uint64_t &*nonce*, *Alignment* &*alignment*)

void vg::Index::**parse_base**(**const** string &*key*, **const** string &*value*, int64_t &*aln_id*, *Alignment* &*alignment*)

void vg::Index::**parse_traversal**(**const** string &*key*, **const** string &*value*, int64_t &*node_id*, int16_t &*rank*, bool &*backward*, int64_t &*aln_id*)

string vg::Index::**entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**graph_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**kmer_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**position_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**metadata_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**node_path_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**path_position_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**mapping_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**alignment_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**base_entry_to_string**(**const** string &*key*, **const** string &*value*)

string vg::Index::**traversal_entry_to_string**(**const** string &*key*, **const** string &*value*)

void vg::Index::**get_context**(int64_t *id*, *VG* &*graph*)

void vg::Index::**expand_context**(*VG* &*graph*, int *steps*)

void vg::Index::**get_range**(int64_t *from_id*, int64_t *to_id*, *VG* &*graph*)

void vg::Index::**for_graph_range**(int64_t *from_id*, int64_t *to_id*, function<void(string&, string&)> *lambda*

void vg::Index::**get_connected_nodes**(*VG* &*graph*)

void vg::Index::**get_edges_of**(int64_t *node*, vector<*Edge*> &*edges*)

void vg::Index::**get_edges_on_end**(int64_t *node*, vector<*Edge*> &*edges*)

void vg::Index::**get_edges_on_start**(int64_t *node*, vector<*Edge*> &*edges*)

void vg::Index::**get_nodes_next**(int64_t *node*, bool *backward*, vector<pair<int64_t, bool>> &*destinations*)

void vg::Index::**get_nodes_prev**(int64_t *node*, bool *backward*, vector<pair<int64_t, bool>> &*destinations*)

void vg::Index::**get_path**(*VG* &*graph*, **const** string &*name*, int64_t *start*, int64_t *end*)

void vg::Index::**node_path_position**(int64_t *id*, string &*path_name*, int64_t &*position*, bool &*backward*, int64_t &*offset*)

pair<list<pair<int64_t, bool>>, pair<int64_t, bool>> vg::Index::**get_nearest_node_prev_path_member**(int64_t *node_id*, bool *backward*, int64_t *path_id*, int64_t &*path_p*, bool &*relative_orie*, int *max_step* = 4)

pair<list<pair<int64_t, bool>>, pair<int64_t, bool>> vg::Index::**get_nearest_node_next_path_member**(int64_t *node_id*, bool *backward*, int64_t *path_id*, int64_t &*path_p*, bool &*relative_orie*, int *max_step* = 4)

bool vg::Index::**get_node_path_relative_position**(int64_t *node_id*, bool *backward*, int64_t *path_id*, list<pair<int64_t, bool>> &*path_prev*, int64_t &*prev_pos*, bool &*prev_orientation*, list<pair<int64_t, bool>> &*path_next*, int64_t &*next_pos*, bool &*next_orientation*)

Mapping vg::Index::**path_relative_mapping**(int64_t *node_id*, bool *backward*, int64_t *path_id*, list<pair<int64_t, bool>> &*path_prev*, int64_t &*prev_pos*, bool &*prev_orientation*, list<pair<int64_t, bool>> &*path_next*, int64_t &*next_pos*, bool &*next_orientation*)

void vg::Index::**path_layout**(map<string, pair<pair<int64_t, bool>, pair<int64_t, bool>>> &*layout*, map<string, int64_t> &*lengths*)

pair<int64_t, bool> vg::Index::**path_first_node**(int64_t *path_id*)

pair<int64_t, bool> vg::Index::**path_last_node**(int64_t *path_id*, int64_t &*path_length*)

void vg::Index::**get_kmer_subgraph**(**const** string &*kmer*, *VG* &*graph*)

uint64_t vg::Index::**approx_size_of_kmer_matches**(**const** string &*kmer*)

void vg::Index::**approx_sizes_of_kmer_matches**(**const** vector<string> &*kmers*, vector<uint64_t> &*sizes*)

void vg::Index::**for_kmer_range**(**const** string &*kmer*, function<void) string&, string& > *lambda*

void vg::Index::**get_kmer_positions**(**const** string &*kmer*, map<int64_t, vector<int32_t>> &*positions*)

void vg::Index::**get_kmer_positions**(**const** string &*kmer*, map<string, vector<pair<int64_t, int32_t>>> &*positions*)

void vg::Index::**prune_kmers**(int *max_kb_on_disk*)

void vg::Index::**remember_kmer_size**(int *size*)

set<int> vg::Index::**stored_kmer_sizes**(void)

void vg::Index::**store_batch**(map<string, string> &*items*)

void vg::Index::**kmer_matches**(*std*::string &*kmer*, *std*::set<int64_t> &*node_ids*, *std*::set<int64_t> &*edge_ids*)

string vg::Index::**first_kmer_key**(**const** string &*kmer*)

pair<int64_t, int64_t> vg::Index::**compare_kmers**(*Index* &*other*)

int64_t vg::Index::**get_max_path_id**(void)

void vg::Index::**put_max_path_id**(int64_t *id*)

int64_t vg::Index::**new_path_id**(**const** string &*name*)

string vg::Index::**path_name_prefix**(**const** string &*name*)

string vg::Index::**path_id_prefix**(int64_t *id*)

void vg::Index::**put_path_id_to_name**(int64_t *id*, **const** string &*name*)

void vg::Index::**put_path_name_to_id**(int64_t *id*, **const** string &*name*)

string vg::Index::**get_path_name**(int64_t *id*)

int64_t vg::Index::**get_path_id**(**const** string &*name*)

void vg::Index::**load_paths**(*VG* &*graph*)

void vg::Index::**store_paths**(*VG* &*graph*)

void vg::Index::**store_path**(*VG* &*graph*, **const** *Path* &*path*)

map<string, int64_t> vg::Index::**paths_by_id**(void)

void vg::Index::**for_each_mapping**(function<void) **const** *Mapping*&
> *lambda*

void vg::Index::**for_each_alignment**(function<void) **const** *Alignment*&
> *lambda*

char vg::Index::**graph_key_type**(**const** string &*key*)

Public Members

string vg::Index::**name**

char vg::Index::**start_sep**

char vg::Index::**end_sep**

int vg::Index::**threads**

rocksdb::DB *vg::Index::**db**

rocksdb::Options vg::Index::**db_options**

rocksdb::WriteOptions vg::Index::**write_options**

rocksdb::ColumnFamilyOptions vg::Index::**column_family_options**

bool vg::Index::**bulk_load**

std::atomic<uint64_t> vg::Index::**next_nonce**
class *#include <index.hpp>* Inherits from exception **Public Functions**

vg::indexOpenException::**indexOpenException**(string *message* = "")

Private Functions

virtual const char *vg::indexOpenException::**what**() **const**

Private Members

string vg::indexOpenException::**message**
struct *#include <genome_state.hpp>* Inherits from *vg::GenomeStateCommand* **Public Functions**

GenomeStateCommand *vg::InsertHaplotypeCommand::**execute**(*GenomeState* **const** *&state*)
Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual vg::InsertHaplotypeCommand::**~InsertHaplotypeCommand**()

Public Members

unordered_map<**const** *Snarl* *, vector<vector<pair<*handle_t*, size_t>>>> vg::InsertHaplotypeCommand::**insertion**
For each snarl, holds several haplotype traversals. The handles in each traversal are annotated with their local lane assignments. This annotation lets the command be basically an undelete, because we can exactly reverse a delete. Insertions are applied from begin to end within each vector. Lane numbers for a given forward handle must strictly increase between vectors and within vectors.
struct *#include <pileup_augmenter.hpp>* **Public Members**

Node *vg::PileupAugmenter::InsertionRecord::**node**

StrandSupport vg::PileupAugmenter::InsertionRecord::**sup**

int64_t vg::PileupAugmenter::InsertionRecord::**orig_id**

int vg::PileupAugmenter::InsertionRecord::**orig_offset**
struct We need to suppress overlapping variants, but interval trees are hard to write. This accomplishes the collision check with a massive bit vector. **Public Functions**

vg::IntervalBitfield::**IntervalBitfield**(size_t *length*)
Make a new *IntervalBitfield* covering a region of the specified length.

bool vg::IntervalBitfield::**collides**(size_t *start*, size_t *pastEnd*)
Scan for a collision (O(n) in interval length)

void vg::IntervalBitfield::**add**(size_t *start*, size_t *pastEnd*)
Take up an interval.

Public Members

vector<bool> vg::IntervalBitfield::**used**
class *#include <phased_genome.hpp>* Unidirectional iterator to obtain the NodeTraversals of a haplotype. Can be come invalid if the *PhasedGenome* is edited while iterating. **Public Functions**

vg::PhasedGenome::iterator::**iterator**()
Default constructor.

vg::PhasedGenome::iterator::**iterator**(**const** *iterator &other*)
Copy constructor.

vg::PhasedGenome::iterator::**~iterator**()
Destructor.

iterator &vg::PhasedGenome::iterator::**operator=**(**const** *iterator* &*other*)

bool vg::PhasedGenome::iterator::**operator==**(**const** *iterator* &*other*) **const**

bool vg::PhasedGenome::iterator::**operator!=**(**const** *iterator* &*other*) **const**

iterator vg::PhasedGenome::iterator::**operator++**()

iterator vg::PhasedGenome::iterator::**operator++**(int)

NodeTraversal vg::PhasedGenome::iterator::**operator***()

int vg::PhasedGenome::iterator::**which_haplotype**()

Private Functions

vg::PhasedGenome::iterator::**iterator**(size_t *rank*, int *haplotype_number*, *HaplotypeNode* **haplo_node*)

Private Members

size_t vg::PhasedGenome::iterator::**rank**
Ordinal position along the haplotype (to distinguish the same node with multiple copies)

int vg::PhasedGenome::iterator::**haplotype_number**
The ID of the haplotype.

HaplotypeNode *vg::PhasedGenome::iterator::**haplo_node**
The position along the haplotype.

Friends

friend vg::PhasedGenome::iterator::PhasedGenome

friend vg::PhasedGenome::iterator::Haplotype
class *#include <cluster.hpp>* Actual iterator class. **Public Functions**

ShuffledPairs::*iterator* &vg::ShuffledPairs::iterator::**operator++**()
Advance to the next pair.

pair<size_t, size_t> vg::ShuffledPairs::iterator::**operator***() **const**
Get the pair pointed to.

bool vg::ShuffledPairs::iterator::**operator==**(**const** *iterator* &*other*) **const**
see if two iterators are equal.

bool vg::ShuffledPairs::iterator::**operator!=**(**const** *iterator* &*other*) **const**
see if two iterators are not equal.

vg::ShuffledPairs::iterator::**iterator**(**const** *iterator* &*other*)

iterator &vg::ShuffledPairs::iterator::**operator=**(**const** *iterator* &*other*)

Private Functions

vg::ShuffledPairs::iterator::**iterator**(**const** *ShuffledPairs* &*iteratee*, size_t *start_at*)

Private Members

size_t vg::ShuffledPairs::iterator::**permutation_idx**

size_t vg::ShuffledPairs::iterator::**permuted**

const *ShuffledPairs* &vg::ShuffledPairs::iterator::**iteratee**

Friends

friend **vg::ShuffledPairs::iterator::ShuffledPairs**

class Inherits from exception

Public Functions

j2pb_error::**j2pb_error**(**const** *std*::string &*e*)

j2pb_error::**j2pb_error**(**const** FieldDescriptor **field*, **const** *std*::string &*e*)

virtual j2pb_error::**~j2pb_error**()

virtual **const** char *j2pb_error::**what**() **const**

Private Members

std::string j2pb_error::**_error**

struct

Public Functions

json_autoptr::**json_autoptr**(json_t **json*)

json_autoptr::**~json_autoptr**()

json_t *json_autoptr::**release**()

Public Members

json_t *json_autoptr::**ptr**

template <**class** T>

class *#include <json2pb.h>*

Public Functions

JSONStreamHelper::**JSONStreamHelper**(**const** *std*::string &*file_name*)

JSONStreamHelper::**~JSONStreamHelper**()

std::function<bool (T&)> JSONStreamHelper::**get_read_fn**

int64_t JSONStreamHelper::**write**(*std*::ostream &*out*, bool *json_out* = false, int64_t *buf_size* = 1000)

Private Members

FILE *JSONStreamHelper::**_fp**
class *#include <index.hpp>* Inherits from exception **Private Functions**

virtual const char *vg::keyNotFoundException::**what**() **const**
struct Used to serialize kmer matches. **Public Members**

string vg::KmerMatch::**sequence**

int64 vg::KmerMatch::**node_id**

sint32 vg::KmerMatch::**position**

bool vg::KmerMatch::**backward**
If true, this kmer is backwards relative to its node, and position counts from the end of the node.
struct *#include <vg.hpp>* We create a struct that represents each kmer record we want to send to gcsa2 **Public Members**

string vg::KmerPosition::**kmer**

string vg::KmerPosition::**pos**

set<char> vg::KmerPosition::**prev_chars**

set<char> vg::KmerPosition::**next_chars**

set<string> vg::KmerPosition::**next_positions**
struct *Support* pinned to a location, which can be either a node or an edge. **Public Members**

Support vg::LocationSupport::**support**
The support.

oneof vg::LocationSupport::**oneof_location**
The location.

int64 vg::LocationSupport::**node_id**
class *#include <graph_synchronizer.hpp>* This represents a request to lock a particular context on a particular *GraphSynchronizer*. It fulfils the BasicLockable concept requirements, so you can wait on it with std::unique_lock. **Public Functions**

vg::GraphSynchronizer::Lock::**Lock**(*GraphSynchronizer* &*synchronizer*, **const** string &*path_name*, size_t *path_offset*, size_t *context_bases*, bool *reflect*)
Create a request to lock a certain radius around a certain position along a certain path in the graph controlled by the given synchronizer.

vg::GraphSynchronizer::Lock::**Lock**(*GraphSynchronizer* &*synchronizer*, **const** string &*path_name*, size_t *start*, size_t *past_end*)
Create a request to lock a certain range of a certain path, from start to end. The start and end positions must line up with the boundaries of nodes in the graph. Also locks attached things that can be reached by paths of the same length or shorter. Note that the range must be nonempty.

void vg::GraphSynchronizer::Lock::**lock**()
Block until a lock is obtained.

void vg::GraphSynchronizer::Lock::**unlock**()
If a lock is held, unlock it.

VG &vg::GraphSynchronizer::Lock::**get_subgraph**()
May only be called when locked. Grab the subgraph that was extracted when the lock was obtained. Does not contain any path information.

pair<*NodeSide*, *NodeSide*> vg::GraphSynchronizer::Lock::**get_endpoints**() **const**
May only be called when locked. Returns the pair of NodeSides corresponding to the start and end positions used when the lock was created.

May only be called on locks that lock a start to end range.

set<*NodeSide*> vg::GraphSynchronizer::Lock::**get_peripheral_attachments**(*NodeSide* *graph_side*)
Get the NodeSides for nodes not in the extracted subgraph but in its periphery that are attached to the given *NodeSide* in the subgraph.

vector<*Translation*> vg::GraphSynchronizer::Lock::**apply_edit**(**const** *Path* &*path*, set<*NodeSide*> &*dangling*)
May only be called when locked. Apply an edit against the base graph and return the resulting translation. Note that this updates only the underlying *VG* graph, not the copy of the locked subgraph stored in the lock. Also note that the edit may only edit locked nodes.

Edit operations will create new nodes, and cannot delete nodes or apply changes (other than dividing and connecting) to existing nodes.

Any new nodes created are created already locked.

Any new nodes created on the left of the alignment (and any existing nodes visited) will be attached to the given "dangling" NodeSides. The set will be populated with the NodeSides for the ends of nodes created/visited at the end of the alignment.

vector<*Translation*> vg::GraphSynchronizer::Lock::**apply_edit**(**const** *Path* &*path*)
May only be called when locked. Apply a path as an edit to the base graph, leaving new nodes at the ends of the path unattached on their outer sides.

vector<*Translation*> vg::GraphSynchronizer::Lock::**apply_full_length_edit**(**const** *Path* &*path*)
May only be called when locked. Apply a path as an edit to the base graph, attaching the outer sides of any newly created nodes to the sides in the periphery attached to the extraction start and end sides, respectively. The lock must have been obtained on a range, rather than a radius.

The alignment must be in the local forward orientation of the graph for this to make sense.

Protected Attributes

GraphSynchronizer &vg::GraphSynchronizer::Lock::**synchronizer**
This points back to the synchronizer we synchronize with when we get locked.

string vg::GraphSynchronizer::Lock::**path_name**

size_t vg::GraphSynchronizer::Lock::**path_offset**

size_t vg::GraphSynchronizer::Lock::**context_bases**

bool vg::GraphSynchronizer::Lock::**reflect**

size_t vg::GraphSynchronizer::Lock::**start**

size_t vg::GraphSynchronizer::Lock::**past_end**

VG vg::GraphSynchronizer::Lock::**subgraph**
This is the subgraph that got extracted during the locking procedure.

pair<*NodeSide*, *NodeSide*> vg::GraphSynchronizer::Lock::**endpoints**
These are the endpoints that the subgraph was extracted between, if applicable.

set<*id_t*> vg::GraphSynchronizer::Lock::**periphery**
These are the nodes connected to the subgraph but not actually available for editing. We just need no one else to edit them.

map<*NodeSide*, set<*NodeSide*>> vg::GraphSynchronizer::Lock::**peripheral_attachments**
This connects internal NodeSides to NodeSides of nodes on the periphery.

set<*id_t*> vg::GraphSynchronizer::Lock::**locked_nodes**
This is the set of nodes that this lock has currently locked.

struct Describes a genetic locus with multiple possible alleles, a genotype, and observational support. **Public Members**

string vg::Locus::**name**
A locus may have an identifying name.

repeated<*Path*> vg::Locus::**allele**
These are all the alleles at the locus, not just the called ones. Note that a primary reference allele may or may not appear.

repeated<*Support*> vg::Locus::**support**
These supports are per-allele, matching the alleles above.

repeated<*Genotype*> vg::Locus::**genotype**
sorted by likelihood or posterior the first one is the "call"

Support vg::Locus::**overall_support**
We also have a *Support* for the locus overall, because reads may have supported multiple alleles and we want to know how many total there were.

repeated<double> vg::Locus::**allele_log_likelihood**
We track the likelihood of each allele individually, in addition to genotype likelihoods. Stores the likelihood natural logged.

class *#include <mapper.hpp>* Inherits from *vg::BaseMapper* **Public Functions**

`vg::Mapper::`**`Mapper`**(*xg*::*XG* *xidex, gcsa::GCSA *g, gcsa::LCPArray *a)

`vg::Mapper::`**`Mapper`**(void)

`vg::Mapper::`**`~Mapper`**(void)

map<string, vector<size_t>> `vg::Mapper::`**`node_positions_in_paths`**(gcsa::node_type *node*)

double `vg::Mapper::`**`graph_entropy`**(void)

map<string, vector<pair<size_t, bool>>> `vg::Mapper::`**`alignment_initial_path_positions`**(**const** *Alignment* &*aln*)

void `vg::Mapper::`**`annotate_with_initial_path_positions`**(*Alignment* &*aln*)

void `vg::Mapper::`**`annotate_with_initial_path_positions`**(vector<*Alignment*> &*alns*)

bool `vg::Mapper::`**`alignments_consistent`**(**const** map<string, double> &*pos1*, **const** map<string, double> &*pos2*, int *fragment_size_bound*)

bool `vg::Mapper::`**`pair_consistent`**(*Alignment* &*aln1*, *Alignment* &*aln2*, double *pval*)
use the fragment length annotations to assess if the pair is consistent or not

pair<bool, bool> `vg::Mapper::`**`pair_rescue`**(*Alignment* &*mate1*, *Alignment* &*mate2*, int *match_score*, int *full_length_bonus*, bool *traceback*)
use the fragment configuration statistics to rescue more precisely

Alignment `vg::Mapper::`**`realign_from_start_position`**(**const** *Alignment* &*aln*, int *extra*, int *iteration*)
assuming the read has only been score-aligned, realign from the end position backwards

set<*MaximalExactMatch* *> `vg::Mapper::`**`resolve_paired_mems`**(vector<*MaximalExactMatch*> &*mems1*, vector<*MaximalExactMatch*> &*mems2*)

vector<*Alignment*> `vg::Mapper::`**`mems_id_clusters_to_alignments`**(**const** *Alignment* &*alignment*, vector<*MaximalExactMatch*> &*mems*, int *additional_multimaps*)

set<**const** vector<*MaximalExactMatch*> *> `vg::Mapper::`**`clusters_to_drop`**(**const** vector<vector<*MaximalExactMatch*>> &*clusters*)

Alignment `vg::Mapper::`**`mems_to_alignment`**(**const** *Alignment* &*aln*, vector<*MaximalExactMatch*> &*mems*)

Alignment `vg::Mapper::`**`mem_to_alignment`**(*MaximalExactMatch* &*mem*)

int32_t vg::Mapper::**score_alignment**(**const** *Alignment* &*aln*, bool *use_approx_distance* = false)

Use the scoring provided by the internal aligner to re-score the alignment, scoring gaps between nodes using graph distance from the XG index. Can use either approximate or exact (with approximate fallback) XG-based distance estimation. Will strip out bonuses if the appropriate *Mapper* flag is set.

void vg::Mapper::**remove_full_length_bonuses**(*Alignment* &*aln*)

Given an alignment scored with full length bonuses on, subtract out the full length bonus if it was applied.

Alignment vg::Mapper::**patch_alignment**(**const** *Alignment* &*aln*, int *max_patch_length*)

VG vg::Mapper::**cluster_subgraph_strict**(**const** *Alignment* &*aln*, **const** vector<*MaximalExactMatch*> &*mems*)

Alignment vg::Mapper::**align_cluster**(**const** *Alignment* &*aln*, **const** vector<*MaximalExactMatch*> &*mems*, bool *traceback*)

double vg::Mapper::**compute_uniqueness**(**const** *Alignment* &*aln*, **const** vector<*MaximalExactMatch*> &*mems*)

Alignment vg::Mapper::**align_maybe_flip**(**const** *Alignment* &*base*, *Graph* &*graph*, bool *flip*, bool *traceback*, bool *banded_global* = false)

bool vg::Mapper::**adjacent_positions**(**const** *Position* &*pos1*, **const** *Position* &*pos2*)

int64_t vg::Mapper::**get_node_length**(int64_t *node_id*)

bool vg::Mapper::**check_alignment**(**const** *Alignment* &*aln*)

VG vg::Mapper::**alignment_subgraph**(**const** *Alignment* &*aln*, int *context_size* = 1)

Alignment vg::Mapper::**align**(**const** string &*seq*, int *kmer_size* = 0, int *stride* = 0, int *max_mem_length* = 0, int *band_width* = 1000)

Alignment vg::Mapper::**align**(**const** *Alignment* &*read*, int *kmer_size* = 0, int *stride* = 0, int *max_mem_length* = 0, int *band_width* = 1000)

vector<*Alignment*> vg::Mapper::**align_multi**(**const** *Alignment* &*aln*, int *kmer_size* = 0, int *stride* = 0, int *max_mem_length* = 0, int *band_width* = 1000)

pair<vector<*Alignment*>, vector<*Alignment*>> vg::Mapper::**align_paired_multi**(**const** *Alignment* &*read1*, **const** *Alignment* &*read2*, bool &*queued_resolve_later*, int *max_mem_length* = 0, bool *only_top_scoring_pair* = false, bool *retrying* = false)

Alignment vg::Mapper::**surject_alignment**(**const** *Alignment* &*source*, set<string> &*path_names*, string &*path_name*, int64_t &*path_pos*, bool &*path_reverse*, int *window*)

double vg::Mapper::**compute_cluster_mapping_quality**(**const** vector<vector<*MaximalExactMatch*>> &*clusters*, int *read_length*)

double vg::Mapper::**estimate_max_possible_mapping_quality**(int *length*, double *min_diffs*, double *next_min_diffs*)

double vg::Mapper::**max_possible_mapping_quality**(int *length*)

int64_t vg::Mapper::**graph_distance**(*pos_t pos1*, *pos_t pos2*, int64_t *maximum* = 1e3)

int64_t vg::Mapper::**graph_mixed_distance_estimate**(*pos_t pos1*, *pos_t pos2*, int64_t *maximum*)

int64_t vg::Mapper::**approx_distance**(*pos_t pos1*, *pos_t pos2*)

int64_t vg::Mapper::**approx_position**(*pos_t pos*)

int64_t vg::Mapper::**approx_alignment_position**(**const** *Alignment* &*aln*)
returns approximate position of alignnment start in xindex or -1.0 if alignment is unmapped

map<string, vector<pair<size_t, bool>>> vg::Mapper::**alignment_path_offsets**(**const** *Alignment* &*aln*, bool *just_min* = true, bool *nearby* = false)

map<string, vector<pair<size_t, bool>>> vg::Mapper::**alignment_refpos_to_path_offsets**(**const** *Alignment* &*aln*)

Position vg::Mapper::**alignment_end_position**(**const** *Alignment* &*aln*)

int64_t vg::Mapper::**approx_fragment_length**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*)
returns approximate distance between alignment starts or -1.0 if not possible to determine

pos_t vg::Mapper::**likely_mate_position**(**const** *Alignment* &*aln*, bool *is_first*)

vector<*pos_t*> vg::Mapper::**likely_mate_positions**(**const** *Alignment* &*aln*, bool *is_first*)

id_t vg::Mapper::**node_approximately_at**(int64_t *approx_pos*)

Alignment vg::Mapper::**walk_match**(**const** string &*seq*, *pos_t pos*)

vector<*Alignment*> vg::Mapper::**walk_match**(**const** *Alignment* &*base*, **const** string &*seq*, *pos_t pos*)

vector<*Alignment*> vg::Mapper::**mem_to_alignments**(*MaximalExactMatch* &*mem*)

map<string, int64_t> vg::Mapper::**min_pair_fragment_length**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*)

double vg::Mapper::**average_node_length**(void)

Public Members

vector<pair<*Alignment*, *Alignment*>> vg::Mapper::**imperfect_pairs_to_retry**

bool vg::Mapper::**debug**

int vg::Mapper::**min_cluster_length**

int vg::Mapper::**context_depth**

int vg::Mapper::**max_attempts**

int vg::Mapper::**thread_extension**

int vg::Mapper::**max_target_factor**

size_t vg::Mapper::**max_query_graph_ratio**

int vg::Mapper::**max_multimaps**

int vg::Mapper::**softclip_threshold**

int vg::Mapper::**max_softclip_iterations**

float vg::Mapper::**min_identity**

int vg::Mapper::**min_banded_mq**

int vg::Mapper::**extra_multimaps**

int vg::Mapper::**min_multimaps**

int vg::Mapper::**band_multimaps**

double vg::Mapper::**maybe_mq_threshold**

int vg::Mapper::**max_cluster_mapping_quality**

bool vg::Mapper::**use_cluster_mq**

double vg::Mapper::**identity_weight**

bool vg::Mapper::**always_rescue**

bool vg::Mapper::**include_full_length_bonuses**

bool vg::Mapper::**simultaneous_pair_alignment**

int vg::Mapper::**max_band_jump**

float vg::Mapper::**drop_chain**

float vg::Mapper::**mq_overlap**

int vg::Mapper::**mate_rescues**

double vg::Mapper::**pair_rescue_hang_threshold**

double vg::Mapper::**pair_rescue_retry_threshold**

FragmentLengthStatistics vg::Mapper::**frag_stats**

Private Functions

Alignment vg::Mapper::**align_to_graph**(**const** *Alignment* &*aln*, *Graph* &*graph*, size_t *max_query_graph_ratio*, bool *traceback*, bool *pinned_alignment* = false, bool *pin_left* = false, bool *global* = false, bool *keep_bonuses* = true)

vector<*Alignment*> vg::Mapper::**align_multi_internal**(bool *compute_unpaired_qualities*, **const** *Alignment* &*aln*, int *kmer_size*, int *stride*, int *max_mem_length*, int *band_width*, double &*cluster_mq*, int *keep_multimaps* = 0, int *additional_multimaps* = 0, vector<*MaximalExactMatch*> **restricted_mems* = nullptr)

void vg::Mapper::**compute_mapping_qualities**(vector<*Alignment*> &*alns*, double *cluster_mq*, double *mq_estimate*, double *mq_cap*)

void vg::Mapper::**compute_mapping_qualities**(pair<vector<*Alignment*>, vector<*Alignment*>> &*pair_alns*, double *cluster_mq*, double *mq_estmate1*, double *mq_estimate2*, double *mq_cap1*, double *mq_cap2*)

vector<*Alignment*> vg::Mapper::**score_sort_and_deduplicate_alignments**(vector<*Alignment*> &*all_alns*, **const** *Alignment* &*original_alignment*)

void vg::Mapper::**filter_and_process_multimaps**(vector<*Alignment*> &*all_alns*, int *total_multimaps*)

vector<*Alignment*> vg::Mapper::**make_bands**(**const** *Alignment* &*read*, int *band_width*, vector<pair<int, int>> &*to_strip*)

vector<*Alignment*> vg::Mapper::**align_banded**(**const** *Alignment* &*read*, int *kmer_size* = 0, int *stride* = 0, int *max_mem_length* = 0, int *band_width* = 1000)

vector<*Alignment*> vg::Mapper::**align_mem_multi**(**const** *Alignment* &*aln*, vector<*MaximalExactMatch*> &*mems*, double &*cluster_mq*, double *lcp_avg*, double *fraction_filtered*, int *max_mem_length*, int *keep_multimaps*, int *additional_multimaps*)

struct A *Mapping* defines the relationship between a node in system and another entity. An empty edit list implies complete match, however it is preferred to specify the full edit structure. as it is more complex to handle special cases. **Public Members**

Position vg::Mapping::**position**

The position at which the first *Edit*, if any, in the *Mapping* starts. Inclusive.

repeated<*Edit*> vg::Mapping::**edit**
The series of *Edit*s to transform to region in read/alt.

int64 vg::Mapping::**rank**
The 1-based rank of the mapping in its containing path.

template <class *From*, class *To*>
class

Public Functions

vg::NGSSimulator::MarkovDistribution::**MarkovDistribution**(size_t *seed*)

void vg::NGSSimulator::MarkovDistribution::**record_transition**(From *from*, To *to*)
record a transition from the input data

void vg::NGSSimulator::MarkovDistribution::**finalize**()
indicate that there is no more data and prepare for sampling

To vg::NGSSimulator::MarkovDistribution::**sample_transition**(From *from*)
sample according to the training data

Private Members

default_random_engine vg::NGSSimulator::MarkovDistribution::**prng**

unordered_map<From, uniform_int_distribution<size_t>> vg::NGSSimulator::MarkovDistribution::**samplers**

unordered_map<To, size_t> vg::NGSSimulator::MarkovDistribution::**column_of**

vector<To> vg::NGSSimulator::MarkovDistribution::**value_at**

unordered_map<From, vector<size_t>> vg::NGSSimulator::MarkovDistribution::**cond_distrs**

class *#include <mem.hpp>*

Public Functions

vg::MaximalExactMatch::**MaximalExactMatch**(string::const_iterator *b*, string::const_iterator *e*, gcsa::range_type *r*, size_t *m* = 0)

string vg::MaximalExactMatch::**sequence**(void) **const**

int vg::MaximalExactMatch::**length**(void) **const**

size_t vg::MaximalExactMatch::**count_Ns**(void) **const**

vg::MaximalExactMatch::**MaximalExactMatch**(void)

vg::MaximalExactMatch::**MaximalExactMatch**(**const** *MaximalExactMatch*&)

vg::MaximalExactMatch::**MaximalExactMatch**(*MaximalExactMatch*&&)

MaximalExactMatch &vg::MaximalExactMatch::**operator=**(**const** *MaximalExactMatch*&)

MaximalExactMatch &vg::MaximalExactMatch::**operator=**(*MaximalExactMatch*&&)

Public Members

string::const_iterator vg::MaximalExactMatch::**begin**

string::const_iterator vg::MaximalExactMatch::**end**

gcsa::range_type vg::MaximalExactMatch::**range**

size_t vg::MaximalExactMatch::**match_count**

int vg::MaximalExactMatch::**fragment**

bool vg::MaximalExactMatch::**primary**

std::vector<gcsa::node_type> vg::MaximalExactMatch::**nodes**

map<string, vector<pair<size_t, bool>>> vg::MaximalExactMatch::**positions**

Friends

bool **operator==**(**const** MaximalExactMatch &*m1*, **const** MaximalExactMatch &*m2*)

bool **operator<**(**const** MaximalExactMatch &*m1*, **const** MaximalExactMatch &*m2*)

ostream &**operator<<**(ostream &*out*, **const** MaximalExactMatch &*m*)

class *#include <cluster.hpp>*

Public Functions

vg::MEMChainModel::**MEMChainModel**(**const** vector<size_t> &*aln_lengths*, **const** vector<vector<*MaximalExactMatch*>> &*matches*, **const** function<int64_t(*pos_t*) > &*approx_position*, **const** function<map<string, vector<pair<size_t, bool>>>(*pos_t*)> &*path_position*, **const** function<double(**const** *MaximalExactMatch*&, **const** *MaximalExactMatch*&)> &*transition_weight*, int *band_width* = 10, int *position_depth* = 1, int *max_connections* = 20

void vg::MEMChainModel::**score**(**const** set<*MEMChainModelVertex* *> &*exclude*)

MEMChainModelVertex *vg::MEMChainModel::**max_vertex**(void)

vector<vector<*MaximalExactMatch*>> vg::MEMChainModel::**traceback**(int *alt_alns*, bool *paired*, bool *debug*)

void vg::MEMChainModel::**display**(ostream &*out*)

void vg::MEMChainModel::**clear_scores**(void)

Public Members

vector<*MEMChainModelVertex*> vg::MEMChainModel::**model**

map<string, map<int64_t, vector<vector<*MEMChainModelVertex*>::iterator>>> vg::MEMChainModel::**positions**

set<vector<*MEMChainModelVertex*>::iterator> vg::MEMChainModel::**redundant_vertexes**

class *#include <cluster.hpp>*

Public Functions

vg::MEMChainModelVertex::**MEMChainModelVertex**(void)

vg::MEMChainModelVertex::**MEMChainModelVertex**(**const** *MEMChainModelVertex*&)

vg::MEMChainModelVertex::**MEMChainModelVertex**(*MEMChainModelVertex*&&)

MEMChainModelVertex &vg::MEMChainModelVertex::**operator=**(**const** *MEMChainModelVertex*&)

MEMChainModelVertex &vg::MEMChainModelVertex::**operator=**(*MEMChainModelVertex*&&)

virtual vg::MEMChainModelVertex::**~MEMChainModelVertex**()

Public Members

MaximalExactMatch vg::MEMChainModelVertex::**mem**

vector<pair<*MEMChainModelVertex* *, double>> vg::MEMChainModelVertex::**next_cost**

vector<pair<*MEMChainModelVertex* *, double>> vg::MEMChainModelVertex::**prev_cost**

double vg::MEMChainModelVertex::**weight**

double vg::MEMChainModelVertex::**score**

MEMChainModelVertex *vg::MEMChainModelVertex::**prev**
struct A subgraph of the unrolled *Graph* in which each non-branching path is associated with an alignment of part of the read and part of the graph such that any path through the *MultipathAlignment* indicates a valid alignment of a read to the graph **Public Members**

string vg::MultipathAlignment::**sequence**

bytes vg::MultipathAlignment::**quality**

string vg::MultipathAlignment::**name**

string vg::MultipathAlignment::**sample_name**

string vg::MultipathAlignment::**read_group**

repeated<*Subpath*> vg::MultipathAlignment::**subpath**
non-branching paths of the multipath alignment, each containing an alignment of part of the sequence to a *Graph* IMPORTANT: downstream applications will assume these are stored in topological order

int32 vg::MultipathAlignment::**mapping_quality**
-10 * log_10(probability of mismapping)

repeated<uint32> vg::MultipathAlignment::**start**
optional: indices of Subpaths that align the beginning of the read (i.e. source nodes)

string vg::MultipathAlignment::**paired_read_name**

class *#include <multipath_mapper.hpp>* **Public Functions**

vg::MultipathAlignmentGraph::**MultipathAlignmentGraph**(*VG &vg*, **const** *MultipathMapper::memcluster_t &hits*, **const** unordered_map<*id_t*, pair<*id_t*, bool>> *&projection_trans*, *SnarlManager *cutting_snarls* = nullptr, int64_t *max_snarl_cut_size* = 5)

vg::MultipathAlignmentGraph::**~MultipathAlignmentGraph**()

void vg::MultipathAlignmentGraph::**topological_sort**(vector<size_t> *&order_out*)
Fills input vector with node indices of a topological sort.

void vg::MultipathAlignmentGraph::**remove_transitive_edges**(**const** vector<size_t> *&topological_order*)
Removes all transitive edges from graph (reduces to minimum equivalent graph) Note: reorders internal representation of adjacency lists

void vg::MultipathAlignmentGraph::**prune_to_high_scoring_paths**(**const** *Alignment &alignment*, **const** *BaseAligner *aligner*, double *MultipathAlignmentGraph*, **const** vector<size_t> *&topological_order*)
Removes nodes and edges that are not part of any path that has an estimated score within some amount of the highest scoring path

void vg::MultipathAlignmentGraph::**reorder_adjacency_lists**(**const** vector<size_t> *&order*)
Reorders adjacency list representation of edges so that they follow the indicated ordering of their target nodes

Public Members

vector<*ExactMatchNode*> vg::MultipathAlignmentGraph::**match_nodes**

class *#include <multipath_mapper.hpp>* Inherits from *vg::BaseMapper* **Public Types**

using We often pass around clusters of MEMs and their graph positions.

using This represents a graph for a cluster, and holds a pointer to the actual extracted graph, a list of assigned MEMs, and the number of bases of read coverage that that MEM cluster provides (which serves as a priority).

Public Functions

vg::MultipathMapper::**MultipathMapper**(*xg*::*XG* **xg_index*, gcsa::GCSA **gcsa_index*, gcsa::LCPArray **lcp_array*, *SnarlManager* **snarl_manager* = nullptr)

vg::MultipathMapper::**~MultipathMapper**()

void vg::MultipathMapper::**multipath_map**(**const** *Alignment* *&alignment*, vector<*MultipathAlignment*> *&multipath_alns_out*, size_t *max_alt_mappings*)

Map read in alignment to graph and make multipath alignments.

void vg::MultipathMapper::**multipath_map_paired**(**const** *Alignment* *&alignment1*, **const** *Alignment* *&alignment2*, vector<pair<*MultipathAlignment*, *MultipathAlignment*>> *&multipath_aln_pairs_out*, vector<pair<*Alignment*, *Alignment*>> *&ambiguous_pair_buffer*, size_t *max_alt_mappings*)

Map a paired read to the graph and make paired multipath alignments. Assumes reads are on the same strand of the DNA/RNA molecule. If the fragment length distribution is still being estimated and the pair cannot be mapped unambiguously, adds the reads to a buffer for ambiguous pairs and does not output any multipath alignments.

bool vg::MultipathMapper::**validate_multipath_alignment**(**const** *MultipathAlignment* *&multipath_aln*) **const**

Debugging function to check that multipath alignment meets the formalism's basic invariants. Returns true if multipath alignment is valid, else false. Does not validate alignment score.

void vg::MultipathMapper::**set_automatic_min_clustering_length**(double *random_mem_probability* = 0.5)

Sets the minimum clustering MEM length to the approximate length that a MEM would have to be to have at most the given probability of occurring in random sequence of the same size as the graph

Public Members

int64_t vg::MultipathMapper::**max_snarl_cut_size**

int32_t vg::MultipathMapper::**band_padding**

size_t vg::MultipathMapper::**max_expected_dist_approx_error**

int32_t vg::MultipathMapper::**num_alt_alns**

double vg::MultipathMapper::**mem_coverage_min_ratio**

double vg::MultipathMapper::**max_suboptimal_path_score_ratio**

size_t vg::MultipathMapper::**num_mapping_attempts**

double vg::MultipathMapper::**log_likelihood_approx_factor**

size_t vg::MultipathMapper::**min_clustering_mem_length**

size_t vg::MultipathMapper::**max_p_value_memo_size**

Protected Functions

void vg::MultipathMapper::**multipath_map_internal**(**const** *Alignment* &*alignment*, *MappingQualityMethod* *mapq_method*, vector<*MultipathAlignment*> &*multipath_alns_out*, size_t *max_alt_mappings*)

Wrapped internal function that allows some code paths to circumvent the current mapping quality method option.

void vg::MultipathMapper::**attempt_unpaired_multipath_map_of_pair**(**const** *Alignment* &*alignment1*, **const** *Alignment* &*alignment2*, vector<pair<*MultipathAlignment*, *MultipathAlignment*>> &*multipath_aln_pairs_out*, vector<pair<*Alignment*, *Alignment*>> &*ambiguous_pair_buffer*)

Before the fragment length distribution has been estimated, look for an unambiguous mapping of the reads using the single ended routine. If we find one record the fragment length and report the pair, if we don't find one, add the read pair to a buffer instead of the output vector.

bool vg::MultipathMapper::**attempt_rescue**(**const** *MultipathAlignment* &*multipath_aln*, **const** *Alignment* &*other_aln*, bool *rescue_forward*, *MultipathAlignment* &*rescue_multipath_aln*)

Extracts a section of graph at a distance from the *MultipathAlignment* based on the fragment length distribution and attempts to align the other paired read to it. If rescuing forward, assumes the provided *MultipathAlignment* is the first read and vice versa if rescuing backward. Rescue constructs a conventional local alignment with gssw and converts the *Alignment* to a *MultipathAlignment*. The *MultipathAlignment* will be stored in the object passed by reference as an argument.

void vg::MultipathMapper::**align_to_cluster_graphs**(**const** *Alignment* *&alignment*, *MappingQualityMethod* *mapq_method*, vector<*clustergraph_t*> *&cluster_graphs*, vector<*MultipathAlignment*> *&multipath_alns_out*, size_t *max_alt_mappings*)

After clustering MEMs, extracting graphs, and assigning hits to cluster graphs, perform multipath alignment

void vg::MultipathMapper::**align_to_cluster_graph_pairs**(**const** *Alignment* *&alignment1*, **const** *Alignment* *&alignment2*, vector<*clustergraph_t*> *&cluster_graphs1*, vector<*clustergraph_t*> *&cluster_graphs2*, vector<pair<pair<size_t, size_t>, int64_t>> *&cluster_pairs*, vector<pair<*MultipathAlignment*, *MultipathAlignment*>> *&multipath_aln_pairs_out*, size_t *max_alt_mappings*)

After clustering MEMs, extracting graphs, assigning hits to cluster graphs, and determining which cluster graph pairs meet the fragment length distance constraints, perform multipath alignment

bool vg::MultipathMapper::**align_to_cluster_graphs_with_rescue**(**const** *Alignment* *&alignment1*, **const** *Alignment* *&alignment2*, vector<*clustergraph_t*> *&cluster_graphs1*, vector<*clustergraph_t*> *&cluster_graphs2*, vector<pair<*MultipathAlignment*, *MultipathAlignment*>> *&multipath_aln_pairs_out*, size_t *max_alt_mappings*)

Align the read ends independently, but also try to form rescue alignments for each from the other. Return true if output obeys pair consistency and false otherwise.

auto vg::MultipathMapper::**query_cluster_graphs**(**const** *Alignment* &*alignment*, **const** vector<*MaximalExactMatch*> &*mems*, **const** vector<*memcluster_t*> &*clusters*)

Extracts a subgraph around each cluster of MEMs that encompasses any graph position reachable (according to the *Mapper*'s aligner) with local alignment anchored at the MEMs. If any subgraphs overlap, they are merged into one subgraph. Returns a vector of all the merged cluster subgraphs, their MEMs assigned from the mems vector according to the MEMs' hits, and their read coverages in bp. The caller must delete the *VG* objects produced!

void vg::MultipathMapper::**split_multicomponent_alignments**(vector<*MultipathAlignment*> &*multipath_alns_out*) **const**

If there are any MultipathAlignments with multiple connected components, split them up and add them to the return vector

void vg::MultipathMapper::**split_multicomponent_alignments**(vector<pair<*MultipathAlignment*, *MultipathAlignment*>> &*multipath_aln_pairs_out*, vector<pair<pair<size_t, size_t>, int64_t>> &*cluster_pairs*) **const**

If there are any MultipathAlignments with multiple connected components, split them up and add them to the return vector, also measure the distance between them and add a record to the cluster pairs vector

void vg::MultipathMapper::**multipath_align**(**const** *Alignment* &*alignment*, *VG* *vg, *memcluster_t* &*graph_mems*, *MultipathAlignment* &*multipath_aln_out*) **const**

Make a multipath alignment of the read against the indicated graph and add it to the list of multimappings.

void vg::MultipathMapper::**strip_full_length_bonuses**(*MultipathAlignment* &*multipath_aln*) **const**

Remove the full length bonus from all source or sink subpaths that received it.

void vg::MultipathMapper::**sort_and_compute_mapping_quality**(vector<*MultipathAlignment*> &*multipath_alns*, *MappingQualityMethod* *mapq_method*) **const**

Sorts mappings by score and store mapping quality of the optimal alignment in the *MultipathAlignment* object.

void vg::MultipathMapper::**sort_and_compute_mapping_quality**(vector<pair<*MultipathAlignment*, *MultipathAlignment*>> &*multipath_aln_pairs*, vector<pair<pair<size_t, size_t>, int64_t>> &*cluster_pairs*) **const**

Sorts mappings by score and store mapping quality of the optimal alignment in the *MultipathAlignment*

object If there are ties between scores, breaks them by the expected distance between pairs as computed by the OrientedDistanceClusterer::cluster_pairs function (modified cluster_pairs vector)

double vg::MultipathMapper::**fragment_length_log_likelihood**(int64_t *length*) **const**
Computes the log-likelihood of a given fragment length in the trained distribution.

bool vg::MultipathMapper::**likely_mismapping**(**const** *MultipathAlignment* &*multipath_aln*)
Would an alignment this good be expected against a graph this big by chance alone.

size_t vg::MultipathMapper::**score_pseudo_length**(int32_t *score*) **const**
A scaling of a score so that it approximately follows the distribution of the longest match in p-value test.

double vg::MultipathMapper::**random_match_p_value**(size_t *match_length*, size_t *read_length*)
The approximate p-value for a match length of the given size against the current graph.

int64_t vg::MultipathMapper::**distance_between**(**const** *MultipathAlignment* &*multipath_aln_1*, **const** *MultipathAlignment* &*multipath_aln_2*) **const**
Compute the approximate distance between two multipath alignments.

bool vg::MultipathMapper::**are_consistent**(**const** *MultipathAlignment* &*multipath_aln_1*, **const** *MultipathAlignment* &*multipath_aln_2*) **const**
Are two multipath alignments consistently placed based on the learned fragment length distribution?

bool vg::MultipathMapper::**is_consistent**(int64_t *distance*) **const**
Is this a consistent inter-pair distance based on the learned fragment length distribution?

double vg::MultipathMapper::**read_coverage_z_score**(int64_t *coverage*, **const** *Alignment* &*alignment*) **const**
Computes the Z-score of the number of matches against an equal length random DNA string.

bool vg::MultipathMapper::**share_start_position**(**const** *MultipathAlignment* &*multipath_aln_1*, **const** *MultipathAlignment* &*multipath_aln_2*) **const**
Return true if any of the initial positions of the source Subpaths are shared between the two multipath alignments

Protected Attributes

SnarlManager *vg::MultipathMapper::**snarl_manager**

Protected Static Functions

int64_t vg::MultipathMapper::**read_coverage**(**const** *memcluster_t* &*mem_hits*)
Computes the number of read bases a cluster of MEM hits covers.

Protected Static Attributes

thread_local unordered_map<pair<size_t, size_t>, double> vg::MultipathMapper::**p_value_memo**

class *#include <handle.hpp>* This is the interface for a handle graph that supports modification. Inherits from *vg::HandleGraph* Subclassed by *vg::VG* **Public Functions**

virtual *handle_t* vg::MutableHandleGraph::**create_handle**(**const** string &*sequence*)
= 0Create a new node with the given sequence and return the handle.

virtual void vg::MutableHandleGraph::**destroy_handle**(**const** *handle_t* &*handle*)
= 0Remove the node belonging to the given handle and all of its edges. Does not update any stored paths.

virtual void vg::MutableHandleGraph::**create_edge**(**const** *handle_t* &*left*, **const** *handle_t* &*right*)
= 0Create an edge connecting the given handles in the given order and orientations. Ignores existing edges.

virtual void vg::MutableHandleGraph::**destroy_edge**(**const** *handle_t* &*left*, **const** *handle_t* &*right*)
= 0Remove the edge connecting the given handles in the given order and orientations. Ignores nonexistent edges. Does not update any stored paths.

virtual void vg::MutableHandleGraph::**swap_handles**(**const** *handle_t* &*a*, **const** *handle_t* &*b*)
= 0Swap 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).

virtual *handle_t* vg::MutableHandleGraph::**apply_orientation**(**const** *handle_t* &*handle*)
= 0Alter 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.

virtual vector<*handle_t*> vg::MutableHandleGraph::**divide_handle**(**const** *handle_t* &*handle*, **const** vector<size_t> &*offsets*)
= 0Split 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.

pair<*handle_t*, *handle_t*> vg::MutableHandleGraph::**divide_handle**(**const** *handle_t* &*handle*, size_t *offset*)
Specialization of divide_handle for a single division point.

class *#include <name_mapper.hpp>* Class to do name mapping (or to mix in and provide name mapping functionality to other classes. Subclassed by *vg::Constructor*, *vg::VariantAdder* **Public Functions**

void vg::NameMapper::**add_name_mapping**(**const** string &*vcf_name*, **const** string &*fasta_name*)
Add a name mapping between a VCF contig name and a FASTA sequence name or graph path name. Both must be unique.

string vg::NameMapper::**vcf_to_fasta**(**const** string &*vcf_name*) **const**
Convert the given VCF contig name to a FASTA sequence or graph path name, through the rename mappings.

string vg::NameMapper::**fasta_to_vcf**(**const** string &*fasta_name*) **const**
Convert the given FASTA sequence name or graph path name to a VCF contig name, through the rename mappings.

Protected Attributes

map<string, string> vg::NameMapper::**vcf_to_fasta_renames**
This map maps from VCF sequence names to FASTA sequence names. If a VCF sequence name doesn't appear in here, it gets passed through unchanged.

map<string, string> vg::NameMapper::**fasta_to_vcf_renames**
This is the reverse map from FASTA sequence name to VCF sequence name.

class *#include <nested_traversal_finder.hpp>* This *TraversalFinder* emits at least one traversal representing every node, edge, or child *Snarl*. Only works on ultrabubbles, and so does not handle cycles. Inherits from *vg::TraversalFinder* **Public Functions**

vg::NestedTraversalFinder::**NestedTraversalFinder**(*SupportAugmentedGraph* &*augmented*, *SnarlManager* &*snarl_manager*)

virtual vg::NestedTraversalFinder::**~NestedTraversalFinder**()

vector<*SnarlTraversal*> vg::NestedTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)
Find traversals to cover the nodes, edges, and children of the snarl. Always emits the primary path traversal first, if applicable.

Public Members

bool vg::NestedTraversalFinder::**verbose**
Should we emit verbose debugging info?

Protected Functions

pair<*Support*, vector<*Visit*>> vg::NestedTraversalFinder::**find_bubble**(*Node* **const* *node*, *Edge* **const* *edge*, *Snarl* **const* *child*, **const** *Snarl* &*site*)
Given an edge or node or child snarl in the augmented graph, look out from the edge or node or child in both directions to find a shortest bubble connecting the start and end of the given site.

Exactly one of edge and node and child must be non-null.

Return the found traversal as a vector of Visits, including anchoring Visits to the site's start and end nodes. Also return the minimum support found on any edge or node in the bubble that is not contained within a child.

If there is no path with any support, returns a zero *Support* and a possibly empty *Path*.

Support vg::NestedTraversalFinder::**min_support_in_path**(**const** vector<*Visit*> &*path*)
Get the minimum support of all nodes and edges used in the given path that are not inside child snarls.

set<pair<size_t, list<*Visit*>>> vg::NestedTraversalFinder::**search_left**(**const** *Visit* &*root*, **const** *Snarl* &*site*)
Do a breadth-first search left from the given node traversal, and return lengths (in visits) and paths starting at the given node and ending on the given indexed path. Refuses to visit nodes with no support.

Lengths are included so that shorter paths sort first.

set<pair<size_t, list<*Visit*>>> vg::NestedTraversalFinder::**search_right**(**const** *Visit* &*root*, **const** *Snarl* &*site*)
Do a breadth-first search right from the given node traversal, and return lengths (in visits) and paths starting at the given node and ending on the given indexed path.

Lengths are included so that shorter paths sort first.

size_t vg::NestedTraversalFinder::**bp_length**(**const** list<*Visit*> &*path*)
Get the length of a path through nodes and child sites, in base pairs. Ignores any bases inside child sites.

Protected Attributes

SupportAugmentedGraph &vg::NestedTraversalFinder::**augmented**
The annotated, augmented graph we're finding traversals in.

SnarlManager &vg::NestedTraversalFinder::**snarl_manager**
The *SnarlManager* managiung the snarls we use.

class *#include <snarls.hpp>* Allow traversing a graph of nodes and child snarl chains within a snarl within another *HandleGraph*. Uses its own internal child index because it's used in the construction of snarls to feed to SnarlManagers. Assumes that the chains we get from Cactus are in a consistent order, so the start of the first snarl is the very first thing in the chain, and the end of the last snarl is the very last. We adapt the handle graph abstraction as follows: A chain becomes a single node with the ID and local forward orientation of its first snarl's start. A chain node connects on its left to everything connected to its first start and on its right to everything connected to its last end. A unary snarl becomes a single node, too. It is identified by its boundary node's ID. If you're not using internal connectivity, a chain node or a unary snarl node behaves just like an ordinary node. If you are using internal connectivity, edges are slightly faked: A chain node also sees out its right everything that is out its left if it has a left-left connected snarl before any disconnected snarl. And similarly for the mirror case. All the edges on either side of a unary snarl node are the same. In this part of the code we talk about "heads" (the inward-facing base graph handles used to represent child snarls/chains), and "tails" (the inward-facing ending handles of child chains). Inherits from *vg::HandleGraph* **Public Functions**

template <typename ChainContainer**>**
vg::NetGraph::**NetGraph**(**const** *Visit* &*start*, **const** *Visit* &*end*, **const** ChainContainer &*child_chains_mixed*, **const** *HandleGraph* *&*graph*, bool *use_internal_connectivity* = false)
Make a new *NetGraph* for the given snarl in the given backing graph, using the given chains as child chains. Unary snarls are stored as single-snarl chains just like other trivial chains.

template <typename ChainContainer, **typename** SnarlContainer**>**
vg::NetGraph::**NetGraph**(**const** *Visit* &*start*, **const** *Visit* &*end*, **const** ChainContainer &*child_chains*, **const** SnarlContainer &*child_unary_snarls*, **const** *HandleGraph* *&*graph*, bool *use_internal_connectivity* = false)
Make a net graph from the given chains and unary snarls (as pointers) in the given backing graph.

vg::NetGraph::**NetGraph**(**const** *Visit* &*start*, **const** *Visit* &*end*, **const** vector<vector<*Snarl*>> &*child_chains*, **const** vector<*Snarl*> &*child_unary_snarls*, **const** *HandleGraph* *&*graph*, bool *use_internal_connectivity* = false)
Make a net graph from the given chains and unary snarls (as raw values) in the given backing graph.

Mostly for testing.

handle_t vg::NetGraph::**get_handle**(**const** *id_t &node_id*, bool *is_reverse* = false) **const**
Look up the handle for the node with the given ID in the given orientation.

id_t vg::NetGraph::**get_id**(**const** *handle_t &handle*) **const**
Get the ID from a handle.

bool vg::NetGraph::**get_is_reverse**(**const** *handle_t &handle*) **const**
Get the orientation of a handle.

handle_t vg::NetGraph::**flip**(**const** *handle_t &handle*) **const**
Invert the orientation of a handle (potentially without getting its ID)

size_t vg::NetGraph::**get_length**(**const** *handle_t &handle*) **const**
Get the length of a node.

string vg::NetGraph::**get_sequence**(**const** *handle_t &handle*) **const**
Get the sequence of a node, presented in the handle's local forward orientation.

bool vg::NetGraph::**follow_edges**(**const** *handle_t &handle*, bool *go_left*, **const** function<bool(**const** *handle_t*&)> &*iteratee*) **const**
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.

void vg::NetGraph::**for_each_handle**(**const** function<bool(**const** *handle_t*&)> &*iteratee*, bool *parallel* = false) **const**
Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false.

size_t vg::NetGraph::**node_size**() **const**
Return the number of nodes in the graph.

const *handle_t* &vg::NetGraph::**get_start**() **const**
Get the inward-facing start handle for this net graph. Useful when working with traversals.

const *handle_t* &vg::NetGraph::**get_end**() **const**
Get the outward-facing end handle for this net graph. Useful when working with traversals.

bool vg::NetGraph::**is_child**(**const** *handle_t &handle*) **const**
Returns true if the given handle represents a meta-node for a child chain or unary snarl, and false if it is a normal node actually in the net graph snarl's contents.

handle_t vg::NetGraph::**get_inward_backing_handle**(**const** *handle_t &child_handle*) **const**
Get the handle in the backing graph reading into the child chain or unary snarl in the orientation represented by this handle to a node representing a child chain or unary snarl.

Protected Functions

vg::NetGraph::**NetGraph**(**const** *Visit &start*, **const** *Visit &end*, **const** *HandleGraph* **graph*, bool *use_internal_connectivity* = false)
Make a *NetGraph* without filling in any of the child indexes.

void vg::NetGraph::**add_unary_child**(**const** *Snarl* **unary*)
Add a unary child snarl to the indexes.

void vg::NetGraph::**add_chain_child**(**const** Chain &*chain*)
Add a chain of one or more non-unary snarls to the index.

Protected Attributes

const *HandleGraph* *vg::NetGraph::**graph**

handle_t vg::NetGraph::**start**

handle_t vg::NetGraph::**end**

bool vg::NetGraph::**use_internal_connectivity**

unordered_set<*handle_t*> vg::NetGraph::**unary_boundaries**

unordered_map<*handle_t*, *handle_t*> vg::NetGraph::**chain_end_rewrites**

unordered_map<*handle_t*, *handle_t*> vg::NetGraph::**chain_ends_by_start**

unordered_map<*id_t*, tuple<bool, bool, bool>> vg::NetGraph::**connectivity**
class *#include <sampler.hpp>* Class that simulates reads with alignments to a graph that mimic the error profile of NGS sequencing data. **Public Functions**

vg::NGSSimulator::**NGSSimulator**(*xg*::*XG* &*xg_index*, **const** string &*ngs_fastq_file*, bool *interleaved_fastq* = false, **const** vector<string> &*source_paths* = { }, double *substition_polymorphism_rate* = 0.001, double *indel_polymorphism_rate* = 0.0002, double *indel_error_proportion* = 0.01, double *insert_length_mean* = 1000.0, double *insert_length_stdev* = 75.0, double *error_multiplier* = 1.0, bool *retry_on_Ns* = true, size_t *seed* = 0)
Initialize simulator. FASTQ file will be used to train an error distribution. Most reads in the FASTQ should be the same length. Polymorphism rates apply uniformly along a read, whereas errors are distributed as indicated by the learned distribution. The simulation can also be restricted to named paths in the graph.

Alignment vg::NGSSimulator::**sample_read**()
Sample an individual read and alignment.

pair<*Alignment*, *Alignment*> vg::NGSSimulator::**sample_read_pair**()
Sample a pair of reads an alignments.

Private Functions

vg::NGSSimulator::**NGSSimulator**(void)

void vg::NGSSimulator::**record_read_quality**(**const** *Alignment* &*aln*, bool *read_2* = false)
Add a quality string to the training data.

void vg::NGSSimulator::**record_read_pair_quality**(**const** *Alignment* &*aln_1*, **const** *Alignment* &*aln_2*)
Add a pair of quality strings to the training data.

void vg::NGSSimulator::**finalize**()
Indicate that there is no more training data.

string vg::NGSSimulator::**sample_read_quality**()
Get a quality string that mimics the training data.

pair<string, string> vg::NGSSimulator::**sample_read_quality_pair**()
Get a pair of quality strings that mimic the training data.

string vg::NGSSimulator::**sample_read_quality_internal**(uint8_t *first*, vector<*MarkovDistribution*<uint8_t, uint8_t>> &*transition_distrs*)
Wrapped internal function for quality sampling.

void vg::NGSSimulator::**sample_read_internal**(*Alignment* &*aln*, size_t &*offset*, bool &*is_reverse*, *pos_t* &*curr_pos*, **const** string &*source_path*)
Internal method called by paired and unpaired samplers for both whole- graph and path sources. Offset and is_reverse are only used (and drive the iteration and update of curr_pos) in path node. Otherwise, in whole graph mode, they are ignored and curr_pos is used to traverse the graph directly.

void vg::NGSSimulator::**sample_start_pos**(size_t &*offset*, bool &*is_reverse*, *pos_t* &*pos*, string &*source_path*)
Sample an appropriate starting position according to the mode. Updates the arguments.

pos_t vg::NGSSimulator::**sample_start_graph_pos**()
Get a random position in the graph.

tuple<size_t, bool, *pos_t*, string> vg::NGSSimulator::**sample_start_path_pos**()
Get a random position along the source path.

string vg::NGSSimulator::**get_read_name**()
Get an unclashing read name.

bool vg::NGSSimulator::**advance**(size_t &*offset*, bool &*is_reverse*, *pos_t* &*pos*, char &*graph_char*, **const** string &*source_path*)
Move forward one position in either the source path or the graph, depending on mode. Update the arguments. Return true if we can't because we hit a tip or false otherwise

bool vg::NGSSimulator::**advance_by_distance**(size_t &*offset*, bool &*is_reverse*, *pos_t* &*pos*, size_t *distance*, **const** string &*source_path*)
Move forward a certain distance in either the source path or the graph, depending on mode. Update the arguments. Return true if we can't because we hit a tip or false otherwise

bool vg::NGSSimulator::**advance_on_path**(size_t &*offset*, bool &*is_reverse*, *pos_t* &*pos*, char &*graph_char*, **const** string &*source_path*)
Move forward one position in the source path, return true if we can't because we hit a tip or false otherwise

bool vg::NGSSimulator::**advance_on_path_by_distance**(size_t &*offset*, bool &*is_reverse*, *pos_t* &*pos*, size_t *distance*, **const** string &*source_path*)
Move forward a certain distance in the source path, return true if we can't because we hit a tip or false otherwise

bool vg::NGSSimulator::**advance_on_graph**(*pos_t* &*pos*, char &*graph_char*)
Move forward one position in the graph along a random path, return true if we can't because we hit a tip or false otherwise

bool vg::NGSSimulator::**advance_on_graph_by_distance**(*pos_t* &*pos*, size_t *distance*)
Move forward a certain distance in the graph along a random path, return true if we can't because we hit a tip or false otherwise

pos_t vg::NGSSimulator::**walk_backwards**(**const** *Path* &*path*, size_t *distance*)
Returns the position a given distance from the end of the path, walking backwards.

void vg::NGSSimulator::**apply_deletion**(*Alignment* &*aln*, **const** *pos_t* &*pos*)
Add a deletion to the alignment.

void vg::NGSSimulator::**apply_insertion**(*Alignment* &*aln*, **const** *pos_t* &*pos*)
Add an insertion to the alignment.

void vg::NGSSimulator::**apply_aligned_base**(*Alignment* &*aln*, **const** *pos_t* &*pos*, char *graph_char*, char *read_char*)
Add a match/mismatch to the alignment.

Private Members

unordered_map<char, string> vg::NGSSimulator::**mutation_alphabets**
Remainder of the alphabet after removing a given character.

vector<double> vg::NGSSimulator::**phred_prob**
Memo for Phred -> probability conversion.

vector<*MarkovDistribution*<uint8_t, uint8_t>> vg::NGSSimulator::**transition_distrs_1**
A Markov distribution for each read position.

vector<*MarkovDistribution*<uint8_t, uint8_t>> vg::NGSSimulator::**transition_distrs_2**
A second set of Markov distributions for the second read in a pair.

MarkovDistribution<uint8_t, pair<uint8_t, uint8_t>> vg::NGSSimulator::**joint_initial_distr**
A distribution for the joint initial qualities of a read pair.

xg::*XG* &vg::NGSSimulator::**xg_index**

LRUCache<*id_t*, *Node*> vg::NGSSimulator::**node_cache**

LRUCache<*id_t*, vector<*Edge*>> vg::NGSSimulator::**edge_cache**

default_random_engine vg::NGSSimulator::**prng**

discrete_distribution vg::NGSSimulator::**path_sampler**

vector<uniform_int_distribution<size_t>> vg::NGSSimulator::**start_pos_samplers**

uniform_int_distribution<uint8_t> vg::NGSSimulator::**strand_sampler**

uniform_int_distribution<size_t> vg::NGSSimulator::**background_sampler**

uniform_int_distribution<size_t> vg::NGSSimulator::**mut_sampler**

uniform_real_distribution<double> vg::NGSSimulator::**prob_sampler**

normal_distribution<double> vg::NGSSimulator::**insert_sampler**

const double vg::NGSSimulator::**sub_poly_rate**

const double vg::NGSSimulator::**indel_poly_rate**

const double vg::NGSSimulator::**indel_error_prop**

const double vg::NGSSimulator::**insert_mean**

const double vg::NGSSimulator::**insert_sd**

size_t vg::NGSSimulator::**sample_counter**

size_t vg::NGSSimulator::**seed**

const bool vg::NGSSimulator::**retry_on_Ns**

vector<string> vg::NGSSimulator::**source_paths**
Restrict reads to just these paths (path-only mode) if nonempty.

Private Static Attributes

const string vg::NGSSimulator::**alphabet**
DNA alphabet.
class *#include <banded_global_aligner.hpp>* This gets thrown when the aligner can't find any valid alignment in the band that was requested. Inherits from exception **Private Functions**

const char *NoAlignmentInBandException::**what**() **const**

Private Static Attributes

const string NoAlignmentInBandException::**message**
struct *Nodes* store sequence data. **Public Members**

string vg::Node::**sequence**
Sequence of DNA bases represented by the *Node*.

string vg::Node::**name**
A name provides an identifier.

int64 vg::Node::**id**
Each *Node* has a unique positive nonzero ID within its *Graph*.
struct *#include <pileup_augmenter.hpp>***Public Types**

enum type vg::NodeDivider::**EntryCat**
Values:
= 0

typedef

typedef

Public Functions

void vg::NodeDivider::**add_fragment**(**const** *Node* *orig_node, int *offset*, *Node* *subnode, *EntryCat* *cat*, vector<*StrandSupport*> *sup*)

NodeDivider::*Entry* vg::NodeDivider::**break_end**(**const** *Node* *orig_node, *VG* *graph, int *offset*, bool *left_side*)

list<*Mapping*> vg::NodeDivider::**map_node**(int64_t *node_id*, int64_t *start_offset*, int64_t *length*, bool *reverse*)

void vg::NodeDivider::**clear**()

Public Members

NodeHash vg::NodeDivider::**index**

int64_t *vg::NodeDivider::**_max_id**

struct **Public Types**

typedef

Public Functions

NodeLengthBuffer::**NodeLengthBuffer**(**const** *xg*::*XG* &*xg_index*)

size_t NodeLengthBuffer::**operator()**(*xg*::*id_t* *id*)

Public Members

const *xg*::*XG* &NodeLengthBuffer::**index**

std::vector<*entry_type*> NodeLengthBuffer::**buffer**

std::hash<*xg*::*id_t*> NodeLengthBuffer::**hash**

Public Static Attributes

const size_t NodeLengthBuffer::**BUFFER_SIZE**

struct Collect pileup records by node. Saves some space and hashing over storing individually, assuming not too sparse and avg. node length more than couple bases the ith *BasePileup* in the array corresponds to the position at offset i. **Public Members**

int64 vg::NodePileup::**node_id**

repeated<*BasePileup*> vg::NodePileup::**base_pileup**

class *#include <nodeside.hpp>* Represents one side of a *Node*, identified by ID, for the purposes of indexing edges. TODO: duplicates much of the functionality of *NodeTraversal*, and causes API duplication to accomodate both. There should only be one. **Public Functions**

`vg::NodeSide::`**`NodeSide`**`(`*id_t node*, bool *is_end* = false`)`
Create a *NodeSide* for the given side of the given *Node*. We need this to be a converting constructor so we can represent the empty and deleted item keys in a *pair_hash_map*.

`vg::NodeSide::`**`NodeSide`**`()`
Create a *NodeSide* for no *Node*.

bool `vg::NodeSide::`**`operator==`**`(`**const** *NodeSide* &*other*`)` **const**
Equality operator.

bool `vg::NodeSide::`**`operator!=`**`(`**const** *NodeSide* &*other*`)` **const**
Inequality operator.

bool `vg::NodeSide::`**`operator<`**`(`**const** *NodeSide* &*other*`)` **const**
Comparison operator for sets and maps.

NodeSide `vg::NodeSide::`**`flip`**`(void)` **const**
Reverse complement the node side, obtaining the other side of the same *Node*.

Visit `vg::NodeSide::`**`to_visit`**`()` **const**
Convert to a *Visit*.

Public Members

id_t `vg::NodeSide::`**`node`**
What *Node* are we a side of?

bool `vg::NodeSide::`**`is_end`**
Are we the end side? Or the start side?

Public Static Functions

static pair<*NodeSide*, *NodeSide*> `vg::NodeSide::`**`pair_from_edge`**`(`*Edge* **e*`)`
Make an edge into a canonically ordered pair of NodeSides.

static pair<*NodeSide*, *NodeSide*> `vg::NodeSide::`**`pair_from_edge`**`(`**const** *Edge* &*e*`)`
Make an edge into a canonically ordered pair of NodeSides.

static pair<*NodeSide*, *NodeSide*> `vg::NodeSide::`**`pair_from_start_edge`**`(`*id_t start_id*, **const** pair<*id_t*, bool> &*oriented_other*`)`
Make a canonically ordered pair of NodeSides from an edge off of the start of a node, to another node in the given relative orientation.

static pair<*NodeSide*, *NodeSide*> `vg::NodeSide::`**`pair_from_end_edge`**`(`*id_t end_id*, **const** pair<*id_t*, bool> &*oriented_other*`)`
Make a canonically ordered pair of NodeSides from an edge off of the end of a node, to another node in the given relative orientation.

class *#include <nodetraversal.hpp>* Represents a node traversed in a certain orientation. The default orientation is start to end, but if `backward` is set, represents the node being traversed end to start. A list of these can serve as an edit-free version of a path, especially if supplemented with a length and an initial node offset. A path node has a left and a right side, which are the start and end of the node if it is forward, or the end and start of the node if it is backward. **Public Functions**

`vg::NodeTraversal::`**`NodeTraversal`**`(`*Node* *node, bool *backward* = false`)`
Make a *NodeTraversal* that traverses the given *Node* in the given orientation. We don't want Node*s to turn into NodeTraversals when we aren't expecting it, so this is explicit.

`vg::NodeTraversal::`**`NodeTraversal`**`()`
Create a *NodeTraversal* of no node.

bool `vg::NodeTraversal::`**`operator==`**`(`**const** *NodeTraversal* &*other*`)` **const**
Equality operator.

bool `vg::NodeTraversal::`**`operator!=`**`(`**const** *NodeTraversal* &*other*`)` **const**
Inequality operator.

bool `vg::NodeTraversal::`**`operator<`**`(`**const** *NodeTraversal* &*other*`)` **const**
Comparison operator for sorting in sets and maps. Make sure to sort by node ID and not pointer value, because people will expect that.

NodeTraversal `vg::NodeTraversal::`**`reverse`**`(`void`)` **const**
Reverse complement the node traversal, returning a traversal of the same node in the opposite direction.

Public Members

Node *`vg::NodeTraversal::`**`node`**
What *Node* is being traversed?

bool `vg::NodeTraversal::`**`backward`**
In what orientation is it being traversed?

class *#include <cluster.hpp>* **Public Functions**

`vg::OrientedDistanceClusterer::ODEdge::`**`ODEdge`**`(`size_t *to_idx*, int32_t *weight*`)`

`vg::OrientedDistanceClusterer::ODEdge::`**`ODEdge`**`()`

`vg::OrientedDistanceClusterer::ODEdge::`**`~ODEdge`**`()`

Public Members

size_t `vg::OrientedDistanceClusterer::ODEdge::`**`to_idx`**
Index of the node that the edge points to.

int32_t `vg::OrientedDistanceClusterer::ODEdge::`**`weight`**
Weight for dynamic programming.

class *#include <cluster.hpp>* **Public Functions**

`vg::OrientedDistanceClusterer::ODNode::`**`ODNode`**`(`**const** *MaximalExactMatch* &*mem*, *pos_t* *start_pos*, int32_t *score*`)`

vg::OrientedDistanceClusterer::ODNode::**ODNode**()

vg::OrientedDistanceClusterer::ODNode::**~ODNode**()

Public Members

const *MaximalExactMatch* *vg::OrientedDistanceClusterer::ODNode::**mem**

pos_t vg::OrientedDistanceClusterer::ODNode::**start_pos**
Position of GCSA hit in the graph.

int32_t vg::OrientedDistanceClusterer::ODNode::**score**
Score of the exact match this node represents.

int32_t vg::OrientedDistanceClusterer::ODNode::**dp_score**
Score used in dynamic programming.

vector<*ODEdge*> vg::OrientedDistanceClusterer::ODNode::**edges_from**
Edges from this node that are colinear with the read.

vector<*ODEdge*> vg::OrientedDistanceClusterer::ODNode::**edges_to**
Edges to this node that are colinear with the read.

template <typename *Value*, typename *Parser* = *OptionValueParser*<Value>>
class *#include <option.hpp>* Represents an option for a type with no extra methods.

Inherits from *vg::BaseOption< Value, Parser >*

Public Functions

vg::Option::**Option**()

virtual vg::Option::**~Option**()

template <**typename** Item, **typename** Parser>
class
class vg::**Option**<vector<Item>, Parser>
#include <option.hpp> Specialize and add vector methods. TODO: magically autodetect if a container type has things and expose them. TODO: switch to operator* and operator->.

Inherits from *vg::BaseOption< vector< Item >, Parser >*

Public Types

using

Public Functions

using vg::Option<vector<Item>, Parser>::**Value** = vector<Item> vg::Option::**Opti**

virtual vg::Option::**~Option**()

size_t vg::Option::**size**() **const**

bool vg::Option::**empty**() **const**

Value::value_type &vg::Option::**at**(size_t *i*)

const Value::value_type &vg::Option::**at**(size_t *i*) **const**

Value::iterator vg::Option::**begin**()

Value::iterator vg::Option::**end**()

Value::const_iterator vg::Option::**begin**() **const**

Value::const_iterator vg::Option::**end**() **const**

class *#include <option.hpp>* All of the option templates inherit from this base class, which the command-line parser uses to feed them strings. Subclassed by *vg::BaseOption< bool, OptionValueParser< bool > >*, *vg::BaseOption< double, OptionValueParser< double > >*, *vg::BaseOption< int64_t, OptionValueParser< int64_t > >*, *vg::BaseOption< size_t, OptionValueParser< size_t > >*, *vg::BaseOption< string, OptionValueParser< string > >*, *vg::BaseOption< vector< Item >, Parser >*, *vg::BaseOption< vector< size_t >, OptionValueParser< vector< size_t > > >*, *vg::BaseOption< vector< string >, OptionValueParser< vector< string > > >*, *vg::BaseOption< Value, Parser >*

Public Functions

virtual const string &vg::OptionInterface::**get_long_option**() **const**
= 0Get the long option text without , like "foos-to-bar".

virtual const string &vg::OptionInterface::**get_short_options**() **const**
= 0Gets a list of short option characters that the option wants, in priority order. If none of these are available, the option will be automatically assigned some other free character.

virtual const string &vg::OptionInterface::**get_description**() **const**
= 0Get the description, like "number of foos to bar per frobnitz".

virtual string vg::OptionInterface::**get_default_value**() **const**
= 0Get the default value as a string. May be generated on the fly.

virtual bool vg::OptionInterface::**has_argument**() **const**
= 0Returns true if the option takes an argument, and false otherwise.

virtual void vg::OptionInterface::**parse**()
= 0Called for no-argument options when the parser encounters them.

virtual void vg::OptionInterface::**parse**(**const** string &*arg*)
= 0Called for argument-having options when the parser encounters them. The passed reference is only valid during the function call, so the option should make a copy.

virtual vg::OptionInterface::**~OptionInterface**()
Everyone needs a virtual destructor!

template <typename Value**>**
class *#include <option.hpp>* This class holds static methods explaining how to parse a type.

Public Functions

template <>
bool vg::OptionValueParser::**has_argument**()
Bool options don't need arguments.

template <>

void vg::OptionValueParser::**parse_default**(**const** bool &*default_value*, bool &*value*)
When someone gives a bool option they mean to invert its default value.

template <>
void vg::OptionValueParser::**parse**(**const** string &*arg*, bool &*value*)
If someone gives a value to a bool, explode.

template <>
string vg::OptionValueParser::**unparse**(**const** bool &*value*)
Represent default values for bools as true and false.

Public Static Functions

static bool vg::OptionValueParser::**has_argument**()
Return true if we need an argument and false otherwise.

static void vg::OptionValueParser::**parse_default**(**const** Value &*default_value*, Value &*value*)
Parse from no argument, but a default value.

static void vg::OptionValueParser::**parse**(**const** string &*arg*, Value &*value*)
Parse from an argument.

static string vg::OptionValueParser::**unparse**(**const** Value &*value*)
Stringify a default value.

template <**typename** Item>
class
class vg::**OptionValueParser**<vector<Item>>
#include <option.hpp> For vector options, we recurse.

Public Static Functions

static bool vg::OptionValueParser::**has_argument**()
Return true if we need an argument and false otherwise.

static void vg::OptionValueParser::**parse_default**(**const** vector<Item> &*default_value*, vector<Item> &*value*)
Parse from no argument, but a default value.

static void vg::OptionValueParser::**parse**(**const** string &*arg*, vector<Item> &*value*)
Parse from an argument.

static string vg::OptionValueParser::**unparse**(**const** vector<Item> &*value*)
Stringify a default value.

class *#include <cluster.hpp>*

Public Types

using Each hit contains a pointer to the original MEM and the position of that particular hit in the graph.

using Each cluster is a vector of hits.

using A memo for the results of *XG::oriented_paths_of_node*.

using A memo for the results of *XG::get_handle*.

Public Functions

vg::OrientedDistanceClusterer::**OrientedDistanceClusterer**(**const** *Alignment* &*alignment*, **const** vector<*MaximalExactMatch*> &*mems*, **const** *QualAdjAligner* &*aligner*, *xg*::*XG* *xgindex, size_t *max_expected_dist_approx_error* = 8, size_t *min_mem_length* = 1, *node_occurrence_on_paths_memo_t* *paths_of_node_memo = nullptr, *handle_memo_t* *handle_memo = nullptr)

Constructor using *QualAdjAligner*, optionally memoizing succinct data structure operations.

vg::OrientedDistanceClusterer::**OrientedDistanceClusterer**(**const** *Alignment* &*alignment*, **const** vector<*MaximalExactMatch*> &*mems*, **const** *Aligner* &*aligner*, *xg*::*XG* *xgindex, size_t *max_expected_dist_approx_error* = 8, size_t *min_mem_length* = 1, *node_occurrence_on_paths_memo_t* *paths_of_node_memo = nullptr, *handle_memo_t* *handle_memo = nullptr)

Constructor using *Aligner*, optionally memoizing succinct data structure operations.

vector<*OrientedDistanceClusterer*::*cluster_t*> vg::OrientedDistanceClusterer::**clusters**(int32_t *max_qual_score* = 60, int32_t *log_likelihood_approx* = 0)

Returns a vector of clusters. Each cluster is represented a vector of MEM hits. Each hit contains a pointer to the original MEM and the position of that particular hit in the graph.

Public Static Functions

vector<pair<pair<size_t, size_t>, int64_t>> vg::OrientedDistanceClusterer::**pair_clusters**(**const** *Alignment* &*alignment_1*, **const** *Alignment* &*alignment_2*, **const** vector<*cluster_t* *> &*left_clusters*, **const** vector<*cluster_t* *> &*right_clusters*, *xg*::*XG* **xgindex*, int64_t *min_inter_cluster*, int64_t *max_inter_cluste*, *node_occurrence_* **paths_of_node_* = nullptr, *handle_memo_t* **handle_memo* = nullptr)

Given two vectors of clusters, an xg index, an bounds on the distance between clusters, returns a vector of pairs of cluster numbers (one in each vector) matched with the estimated distance

Private Functions

vg::OrientedDistanceClusterer::**OrientedDistanceClusterer**(**const** *Alignment* &*alignment*, **const** vector<*MaximalExactMatch*> &*mems*, **const** *Aligner* ***aligner*, **const** *QualAdjAligner* ***qual_adj_aligner*, *xg*::*XG* ***xgindex*, size_t *max_expected_dist_approx_error*, size_t *min_mem_length*, *node_occurrence_on_paths_memo_t* ***paths_of_node_memo*, *handle_memo_t* ***handle_memo*)

Internal constructor that public constructors filter into.

vector<pair<size_t, size_t>> vg::OrientedDistanceClusterer::**compute_tail_mem_coverage**(**const** *Alignment* &*alignment*, **const** vector<*MaximalE* &*mems*)

Returns a vector containing the number of SMEM beginnings to the left and the number of SMEM endings to the right of each read position

void vg::OrientedDistanceClusterer::**identify_sources_and_sinks**(vector<size_t> &*sources_out*, vector<size_t> &*sinks_out*)

Fills input vectors with indices of source and sink nodes.

void vg::OrientedDistanceClusterer::**connected_components**(vector<vector<size_t>> &*components_out*)

Identify weakly connected components in the graph.

void vg::OrientedDistanceClusterer::**topological_order**(vector<size_t> &*order_out*)

Fills the input vector with the indices of a topological sort.

void vg::OrientedDistanceClusterer::**perform_dp**()

Perform dynamic programming and store scores in nodes.

Private Members

vector<*ODNode*> vg::OrientedDistanceClusterer::**nodes**

const *Aligner* *vg::OrientedDistanceClusterer::**aligner**

const *QualAdjAligner* *vg::OrientedDistanceClusterer::**qual_adj_aligner**

Private Static Functions

unordered_map<pair<size_t, size_t>, int64_t> vg::OrientedDistanceClusterer::**get_on_strand_distance_t**

> *&get_position*, **const** function<int64_tsize_t> *&get_offset*, *node_occurrence_on_paths_memo_t* *paths_of_node_memo* = nullptr, *handle_memo_t* *handle_memo* = nullptrGiven a certain number of items, and a callback to get each item's position, and a callback to a fixed offset from that position build a distance forest with trees for items that we can verify are on the same strand of the same molecule.

We use the distance approximation to cluster the MEM hits according to the strand they fall on using the oriented distance estimation function in xg.

Returns a map from item pair (lower number first) to distance (which may be negative) from the first to the second along the items' forward strand.

void vg::OrientedDistanceClusterer::**extend_dist_tree_by_permutations**(int64_t *max_failed_distance_probes*, int64_t *max_search_distance_to_pa*, size_t *decrement_frequency*, size_t &*num_possible_merges_rem*, *UnionFind* &*component_union_find*, unordered_map<pair<size_t, size_t>, int64_t> &*recorded_finite_dists*, map<pair<size_t, size_t>, size_t> &*num_infinite_dists*, size_t *num_items*, *xg*::*XG* **xgindex*, **const** function<*pos_t*) size_t > &*get_position*, **const** function<int64_tsize_t> &*get_offset*, *node_occurrence_on_paths_memo_t* **paths_of_node_memo* = nullptr, *handle_memo_t* **handle_memo* = nullptrAdds edges into the distance tree by estimating the distance between pairs generated by a high entropy deterministic permutation

void vg::OrientedDistanceClusterer::**extend_dist_tree_by_path_buckets**(int64_t *max_failed_distance_probes*, size_t &*num_possible_merges_rem*, *Union-Find* &*component_union_find*, unordered_map<pair<size_t, size_t>, int64_t> &*recorded_finite_dists*, map<pair<size_t, size_t>, size_t> &*num_infinite_dists*, size_t *num_items*, *xg*::*XG* **xgindex*, **const** function<*pos_t*) size_t > &*get_position*, **const** function<int64_tsize_t> &*get_offset*, *node_occurrence_on_paths_memo_t* **paths_of_node_memo* = nullptr, *handle_memo_t* **handle_memo* = nullptrAdds edges into the distance tree by estimating the distance only between pairs of items that can be directly inferred to share a path based on the memo of node occurrences on paths

vector<unordered_map<size_t, int64_t>> vg::OrientedDistanceClusterer::**flatten_distance_tree**(size_t *num_ite* **const** unordered size_t> int64_t &*recor*

Given a number of nodes, and a map from node pair to signed relative distance on a consistent strand (defining a forrest of trees, as generated by *get_on_strand_distance_tree()*), flatten all the trees.

Returns a vector of maps from node ID to relative position in linear space, one map per input tree.

Assumes all the distances are transitive, even though this isn't quite true in graph space.

class *#include <packer.hpp>*

Public Functions

vg::Packer::**Packer**(void)

vg::Packer::**Packer**(*xg*::*XG* **xidx*, size_t *bin_size*)

vg::Packer::**~Packer**(void)

void vg::Packer::**merge_from_files**(**const** vector<string> &*file_names*)

void vg::Packer::**merge_from_dynamic**(vector<*Packer* *> &*packers*)

void vg::Packer::**load_from_file**(**const** string &*file_name*)

void vg::Packer::**save_to_file**(**const** string &*file_name*)

void vg::Packer::**load**(istream &*in*)

size_t vg::Packer::**serialize**(*std*::ostream &*out*, *sdsl*::structure_tree_node *s = NULL, *std*::string *name* = "")

void vg::Packer::**make_compact**(void)

void vg::Packer::**make_dynamic**(void)

void vg::Packer::**add**(**const** *Alignment* &*aln*, bool *record_edits* = true)

size_t vg::Packer::**graph_length**(void) **const**

size_t vg::Packer::**position_in_basis**(**const** *Position* &*pos*) **const**

string vg::Packer::**pos_key**(size_t *i*) **const**

string vg::Packer::**edit_value**(**const** *Edit* &*edit*, bool *revcomp*) **const**

vector<*Edit*> vg::Packer::**edits_at_position**(size_t *i*) **const**

size_t vg::Packer::**coverage_at_position**(size_t *i*) **const**

void vg::Packer::**collect_coverage**(**const** *Packer* &*c*)

ostream &vg::Packer::**as_table**(ostream &*out*, bool *show_edits* = true)

ostream &vg::Packer::**show_structure**(ostream &*out*)

void vg::Packer::**write_edits**(vector<ofstream *> &*out*) **const**

void vg::Packer::**write_edits**(ostream &*out*, size_t *bin*) **const**

size_t vg::Packer::**get_bin_size**(void) **const**

size_t vg::Packer::**get_n_bins**(void) **const**

bool vg::Packer::**is_dynamic**(void)

Public Members

xg::*XG* *vg::Packer::**xgidx**

Private Functions

void vg::Packer::**ensure_edit_tmpfiles_open**(void)

void vg::Packer::**close_edit_tmpfiles**(void)

void vg::Packer::**remove_edit_tmpfiles**(void)

size_t vg::Packer::**bin_for_position**(size_t *i*) **const**

string vg::Packer::**escape_delim**(**const** string &*s*, char *d*) **const**

string vg::Packer::**escape_delims**(**const** string &*s*) **const**

string vg::Packer::**unescape_delim**(**const** string &*s*, char *d*) **const**

string vg::Packer::**unescape_delims**(**const** string &*s*) **const**

Private Members

bool vg::Packer::**is_compacted**

gcsa::CounterArray vg::Packer::**coverage_dynamic**

vector<string> vg::Packer::**edit_tmpfile_names**

vector<ofstream *> vg::Packer::**tmpfstreams**

size_t vg::Packer::**n_bins**

size_t vg::Packer::**bin_size**

size_t vg::Packer::**edit_length**

size_t vg::Packer::**edit_count**

dac_vector vg::Packer::**coverage_civ**

vector<csa_sada<enc_vector<>, 32, 32, sa_order_sa_sampling<>, isa_sampling<>, succinct_byte_alphabet<>>> vg::Packer

char vg::Packer::**delim1**

char vg::Packer::**delim2**

class *#include <packer.hpp>* Inherits from vector< Packer > **Private Functions**

void vg::Packers::**load**(**const** vector<string> &*file_names*)

ostream &vg::Packers::**as_table**(ostream &*out*)

template <typename *K*, typename *V*>

class *#include <hash_map.hpp>* Inherits from google::dense_hash_map< K, V, std::hash< K > >

Public Functions

vg::pair_hash_map::**pair_hash_map**()

template <**typename** K, **typename** V>

class *#include <hash_map_set.hpp>* Inherits from google::sparse_hash_map< K, V, std::hash< K > >

Public Functions

xg::pair_hash_map::**pair_hash_map**()

template <**typename** K>

class *#include <hash_map_set.hpp>* Inherits from google::sparse_hash_set< K, std::hash< K > >

Public Functions

xg::pair_hash_set::**pair_hash_set**()
struct *Paths* are walks through nodes defined by a series of *Edit*s. They can be used to represent: haplotypes mappings of reads, or alignments, by including edits relationships between nodes annotations from other data sources, such as: genes, exons, motifs, transcripts, peaks

Public Members

string vg::Path::**name**
The name of the path. *Path* names starting with underscore (_) are reserved for internal *VG* use.

repeated<*Mapping*> vg::Path::**mapping**
The *Mapping*s which describe the order and orientation in which the *Path* visits *Node*s.

bool vg::Path::**is_circular**
Set to true if the path is circular.

int64 vg::Path::**length**
Optional length annotation for the *Path*.
class *#include <genotypekit.hpp>* Inherits from *vg::TraversalFinder* **Public Functions**

vg::PathBasedTraversalFinder::**PathBasedTraversalFinder**(*vg*::*VG* &*graph*, *SnarlManager* &*sm*)

virtual vg::PathBasedTraversalFinder::**~PathBasedTraversalFinder**()

vector<*SnarlTraversal*> vg::PathBasedTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)

Private Members

vg::*VG* &vg::PathBasedTraversalFinder::**graph**

SnarlManager &vg::PathBasedTraversalFinder::**snarlmanager**
class *#include <chunker.hpp>* Chunk up a graph along a path, using a given number of context expansion steps to fill out the chunks. Most of the work done by exising xg functions. For gams, the rocksdb index is also required. **Public Functions**

vg::PathChunker::**PathChunker**(*xg*::XG **xg* = NULL)

vg::PathChunker::**~PathChunker**()

void vg::PathChunker::**extract_subgraph**(**const** *Region* &*region*, int *context*, int *length*, bool *forward_only*, *VG* &*subgraph*, *Region* &*out_region*)
Extract subgraph corresponding to given path region into its own vg graph, and send it to out_stream. The boundaries of the extracted graph (which can be different because we expand context and don't cut nodes) are written to out_region. If forward_only set, context is only expanded in the forward direction

NOTE: we follow convention of *Region* coordinates being 1-based inclusive.

void vg::PathChunker::**extract_id_range**(*vg*::*id_t* *start*, *vg*::*id_t* *end*, int *context*, int *length*, bool *forward_only*, *VG* &*subgraph*, *Region* &*out_region*)
Like above, but use (inclusive) id range instead of region on path.

int64_t vg::PathChunker::**extract_gam_for_subgraph**(*VG* &*subgraph*, *Index* &*index*, ostream **out_stream*, bool *only_fully_contained* = false, bool *search_all_positions* = false, bool *unsorted_index* = false)
Extract all alignments that touch a node in a subgraph and write them to an output stream using the rocksdb index (and this->gam_buffer_size)

int64_t vg::PathChunker::**extract_gam_for_ids**(vector<*vg*::*id_t*> &*graph_ids*, *Index* &*index*, ostream **out_stream*, bool *contiguous_id_range* = false, bool *only_fully_contained* = false, bool *search_all_positions* = false, bool *unsorted_index* = false)
More general interface used by above two functions

Public Members

xg::XG *vg::PathChunker::**xg**

size_t vg::PathChunker::**gam_buffer_size**
struct *#include <path_index.hpp>* Holds indexes of the reference in a graph: position to node, node to position and orientation, and the full reference string. Also knows about the lengths of nodes on the path, and lets you iterate back and forth over it. **Public Types**

using We keep iterators to node occurrences along the ref path.

Public Functions

vg::PathIndex::**PathIndex**(**const** *Path* &*path*)
Index just a path.

vg::PathIndex::**PathIndex**(**const** list<*Mapping*> &*mappings*)
Index just a list of mappings.

vg::PathIndex::**PathIndex**(**const** list<*Mapping*> &*mappings*, *VG* &*vg*)
Index a list of mappings embedded in the given vg's *Paths* object, and pull sequence from the given vg.

vg::PathIndex::**PathIndex**(**const** *Path* &*path*, **const** *xg*::*XG* &*vg*)
Index a path and pull sequence from an XG index.

vg::PathIndex::**PathIndex**(*VG* &*vg*, **const** string &*path_name*, bool *extract_sequence* = false)
Make a *PathIndex* from a path in a graph.

vg::PathIndex::**PathIndex**(**const** *xg*::*XG* &*index*, **const** string &*path_name*, bool *extract_sequence* = false)
Make a *PathIndex* from a path in an indexed graph.

void vg::PathIndex::**update_mapping_positions**(*VG* &*vg*, **const** string &*path_name*)
Rebuild the mapping positions map by tracing all the paths in the given graph. TODO: We ought to move this functionality to the *Paths* object and make it use a good datastructure instead of brute force.

NodeSide vg::PathIndex::**at_position**(size_t *position*) **const**
Find what node and orientation covers a position. The position must not be greater than the path length.

bool vg::PathIndex::**path_contains_node**(int64_t *node_id*)

PathIndex::*iterator* vg::PathIndex::**begin**() **const**
Get the iterator to the first node occurrence on the indexed path.

PathIndex::*iterator* vg::PathIndex::**end**() **const**
Get the iterator to the last node occurrence on the indexed path.

PathIndex::*iterator* vg::PathIndex::**find_position**(size_t *position*) **const**
Find the iterator at the given position along the ref path. The position must not be greater than the path length.

size_t vg::PathIndex::**node_length**(**const** *iterator* &*here*) **const**
Get the length of the node occurrence on the path represented by this iterator.

pair<size_t, size_t> vg::PathIndex::**round_outward**(size_t *start*, size_t *past_end*) **const**
Given an end-exclusive range on the path, round outward to the nearest node boundary positions.

void vg::PathIndex::**apply_translation**(**const** *Translation* &*translation*)
Update the index to reflect the changes described by a *Translation*. References to nodes along the "from" path are changed to references to nodes along the "to" path. The translation must contain two paths of equal length, containing only matches. The translation must only divide nodes; it may not join nodes together. The translation must fully account for each old node that it touches (it can't translate only part of a node). The translation may not re-use the ID from one original node for a piece of a different original node. All the Mappings in the *Translation* must have Edits.

void vg::PathIndex::**apply_translations**(**const** vector<*Translation*> &*translations*)
Update the index to reflect the changes described by the given collection of Translations. These translations are expected to be in the format produced by *VG::edit()* which is one to *Mapping* per translation. The vector may include both forward and reverse versions of each to node, and may also include translations mapping nodes that did not change to themselves.

Public Members

map<int64_t, pair<size_t, bool>> vg::PathIndex::**by_id**
Index from node ID to first position on the reference string and orientation it occurs there.

map<size_t, *NodeSide*> vg::PathIndex::**by_start**
Index from start position on the reference to the side of the node that begins there. If it is a right side, the node occurs on the path in a reverse orientation.

std::string vg::PathIndex::**sequence**
The actual sequence of the path, if desired.

map<**const** *Mapping* *, size_t> vg::PathIndex::**mapping_positions**
Index from *Mapping* pointers in a *VG Paths* object to their actual positions along their paths. Pointers may dangle if the vg graph changes the path.

Protected Functions

map<*id_t*, vector<*Mapping*>> vg::PathIndex::**parse_translation**(**const** *Translation* &*translation*)

Convert a *Translation* that partitions old nodes into a map from old node ID to the Mappings that replace it in its forward orientation.

void vg::PathIndex::**replace_occurrence**(*iterator to_replace*, **const** vector<*Mapping*> &*replacements*)

Given an iterator into by_start, replace the occurrence of the node there with occurrences of the nodes given in the vector of mappings, which partition the forward strand of the node being replaced.

Protected Attributes

size_t vg::PathIndex::**last_node_length**

This, combined with by_start, gets us the length of every node on the indexed path.

map<*id_t*, vector<*iterator*>> vg::PathIndex::**node_occurrences**

This holds all the places that a particular node occurs, in order. TODO: use this to replace by_id

class *#include <path.hpp>*

Public Functions

vg::Paths::**Paths**(void)

vg::Paths::**Paths**(**const** *Paths* &*other*)

vg::Paths::**Paths**(*Paths* &&*other*)

Paths &vg::Paths::**operator=**(**const** *Paths* &*other*)

Paths &vg::Paths::**operator=**(*Paths* &&*other*)

void vg::Paths::**sort_by_mapping_rank**(void)

void vg::Paths::**rebuild_mapping_aux**(void)

Reassign ranks and rebuild indexes, treating the mapping lists in _paths as the truth.

bool vg::Paths::**is_head_or_tail_node**(*id_t id*)

vector<string> vg::Paths::**all_path_names**(void)

void vg::Paths::**make_circular**(**const** string &*name*)

void vg::Paths::**make_linear**(**const** string &*name*)

void vg::Paths::**rebuild_node_mapping**(void)

list<*Mapping*>::iterator vg::Paths::**find_mapping**(*Mapping* **m*)

list<*Mapping*>::iterator vg::Paths::**remove_mapping**(*Mapping* **m*)

list<*Mapping*>::iterator vg::Paths::**insert_mapping**(list<*Mapping*>::iterator *w*, **const** string &*path_name*, **const** *Mapping* &*m*)

pair<*Mapping* *, *Mapping* *> vg::Paths::**divide_mapping**(*Mapping* **m*, **const** *Position* &*pos*)

pair<*Mapping* *, *Mapping* *> vg::Paths::**divide_mapping**(*Mapping* **m*, size_t *offset*)

pair<*Mapping* *, *Mapping* *> vg::Paths::**replace_mapping**(*Mapping* **m*, pair<*Mapping*, *Mapping*> *n*)

void vg::Paths::**remove_paths**(**const** set<string> &*names*)

void vg::Paths::**remove_path**(**const** string &*name*)

void vg::Paths::**keep_paths**(**const** set<string> &*name*)

void vg::Paths::**remove_node**(*id_t id*)

bool vg::Paths::**has_path**(**const** string &*name*)

void vg::Paths::**to_json**(ostream &*out*)

list<*Mapping*> &vg::Paths::**get_path**(**const** string &*name*)

list<*Mapping*> &vg::Paths::**get_create_path**(**const** string &*name*)

list<*Mapping*> &vg::Paths::**create_path**(**const** string &*name*)

bool vg::Paths::**has_mapping**(**const** string &*name*, size_t *rank*)

bool vg::Paths::**has_node_mapping**(*id_t id*)

bool vg::Paths::**has_node_mapping**(*Node* **n*)

map<string, set<*Mapping* *>> &vg::Paths::**get_node_mapping**(*Node* **n*)

map<string, set<*Mapping* *>> &vg::Paths::**get_node_mapping**(*id_t id*)

map<string, map<int, *Mapping* *>> vg::Paths::**get_node_mappings_by_rank**(*id_t id*)

map<string, map<int, *Mapping*>> vg::Paths::**get_node_mapping_copies_by_rank**(*id_t id*)

Mapping *vg::Paths::**traverse_left**(*Mapping* **mapping*)

Mapping *vg::Paths::**traverse_right**(*Mapping* **mapping*)

const string vg::Paths::**mapping_path_name**(*Mapping* **m*)

set<string> vg::Paths::**of_node**(*id_t id*)

map<string, int> vg::Paths::**node_path_traversal_counts**(*id_t id*, bool *rev* = false)

vector<string> vg::Paths::**node_path_traversals**(*id_t id*, bool *rev* = false)

bool vg::Paths::**are_consecutive_nodes_in_path**(*id_t id1*, *id_t id2*, **const** string &*path_name*)

vector<string> vg::Paths::**over_edge**(*id_t id1*, bool *rev1*, *id_t id2*, bool *rev2*, vector<string> *following*)

vector<string> vg::Paths::**over_directed_edge**(*id_t id1*, bool *rev1*, *id_t id2*, bool *rev2*, vector<string> *following*)

size_t vg::Paths::**size**(void) **const**

bool vg::Paths::**empty**(void) **const**

void vg::Paths::**clear**(void)

void vg::Paths::**clear_mapping_ranks**(void)

void vg::Paths::**compact_ranks**(void)

void vg::Paths::**load**(istream &*in*)

void vg::Paths::**write**(ostream &*out*)

void vg::Paths::**to_graph**(*Graph* &*g*)
Add all paths into the given Protobuf graph. Creates a new path for every path.

Path vg::Paths::**path**(**const** string &*name*)

void vg::Paths::**append_mapping**(**const** string &*name*, **const** *Mapping* &*m*)

void vg::Paths::**append_mapping**(**const** string &*name*, *id_t id*, size_t *rank* = 0, bool *is_reverse* = false)

void vg::Paths::**prepend_mapping**(**const** string &*name*, **const** *Mapping* &*m*)

void vg::Paths::**prepend_mapping**(**const** string &*name*, *id_t id*, size_t *rank* = 0, bool *is_reverse* = false)

size_t vg::Paths::**get_next_rank**(**const** string &*name*)

void vg::Paths::**append**(*Paths* &*p*)

void vg::Paths::**append**(*Graph* &*g*)

void vg::Paths::**extend**(*Paths* &*p*)

void vg::Paths::**extend**(**const** *Path* &*p*)

void vg::Paths::**for_each**(**const** function<void) **const** *Path*&
> &*lambda*

void vg::Paths::**for_each_name**(**const** function<void) **const** string&
> &*lambda*

void vg::Paths::**for_each_stream**(istream &*in*, **const** function<void) *Path*&
> &*lambda*

void vg::Paths::**increment_node_ids**(*id_t inc*)

void vg::Paths::**swap_node_ids**(*hash_map*<*id_t*, *id_t*> &*id_mapping*)

void vg::Paths::**reassign_node**(*id_t new_id*, *Mapping* *m)

void vg::Paths::**for_each_mapping**(**const** function<void) *Mapping* *
> &*lambda*

Public Members

map<string, list<*Mapping*>> vg::Paths::**_paths**

map<*Mapping* *, list<*Mapping*>::iterator> vg::Paths::**mapping_itr**

map<*Mapping* *, string> vg::Paths::**mapping_path**

map<string, map<size_t, *Mapping* *>> vg::Paths::**mappings_by_rank**

map<*id_t*, map<string, set<*Mapping* *>>> vg::Paths::**node_mapping**

set<*id_t*> vg::Paths::**head_tail_nodes**

set<string> vg::Paths::**circular**
class *#include <phased_genome.hpp>* A collection of haplotypes that represent all of the chromosomes of a genome (including phasing) as walks through a variation graph. Designed for fast editing at a site level, so it maintains indices of sites for that purpose. **Public Functions**

vg::PhasedGenome::**PhasedGenome**(*SnarlManager* &*snarl_manager*)
Constructor.

vg::PhasedGenome::**~PhasedGenome**()

template <typename NodeTraversalIterator>
int vg::PhasedGenome::**add_haplotype**(NodeTraversalIterator *first*, NodeTraversalIterator *last*)
Build a haplotype in place from an iterator that returns *NodeTraversal* objects from its dereference operator (allows construction without instantiating the haplotype elsewhere) returns the numerical id of the new haplotype

note: the haplotype must have at least one node

void vg::PhasedGenome::**build_indices**()
Construct the site ends, node locations, and haplotype site location indices. This method is intended to be called one time after building haplotypes. After this, the are maintained automatically during edit operations.

size_t vg::PhasedGenome::**num_haplotypes**()

PhasedGenome::*iterator* vg::PhasedGenome::**begin**(int *which_haplotype*)
Unidirectional iterator starting at the left telomere and moving to the right. Requires a haplotype id as input

PhasedGenome::*iterator* vg::PhasedGenome::**end**(int *which_haplotype*)
Iterator representing the past-the-last position of the given haplotype, with the last position being the right telomere node.

void vg::PhasedGenome::**swap_alleles**(**const** *Snarl* &*site*, int *haplotype_1*, int *haplotype_2*)
Swap the allele from between two haplotypes, maintaining all indices. If a nested site is being swapped, this method should be called only once for the top-most site. Child sites are swapped along with the top-most site automatically.

template <typename NodeTraversalIterator>
void vg::PhasedGenome::**set_allele**(**const** *Snarl* &*site*, NodeTraversalIterator *first*, NodeTraversalIterator *last*, int *which_haplotype*)
Set the allele at a site with an iterator that yields its node sequence. The allele should be provided in the order indicated by the *Snarl* (i.e. from start to end) and it should not include the boundary nodes of the *Snarl*.

Note: This method does not check that the allele path takes only edges that are actually included in the graph, so client must ensure this itself.

int32_t vg::PhasedGenome::**optimal_score_on_genome**(**const** *MultipathAlignment* &*multipath_aln*, *VG* &*graph*)

Private Functions

void vg::PhasedGenome::**build_site_indices_internal**(**const** *Snarl* *snarl)

void vg::PhasedGenome::**insert_left**(*NodeTraversal* node_traversal, *HaplotypeNode* *haplo_node)
Insert a node traversal to the left of this haplotype node and update indices.

void vg::PhasedGenome::**insert_right**(*NodeTraversal* node_traversal, *HaplotypeNode* *haplo_node)
Insert a node traversal to the right of this haplotype node and update indices.

void vg::PhasedGenome::**remove**(*HaplotypeNode* *haplo_node)
Remove this haplotype node from its haplotype and update indices.

void vg::PhasedGenome::**swap_label**(**const** *Snarl* &site, *Haplotype* &haplotype_1, *Haplotype* &haplotype_2)
Update a subsite's location in indices after swapping its parent allele.

Private Members

SnarlManager &vg::PhasedGenome::**snarl_manager**

vector<*Haplotype* *> vg::PhasedGenome::**haplotypes**
All haplotypes in the genome (generally 2 per chromosome)

unordered_map<int64_t, list<*HaplotypeNode* *>> vg::PhasedGenome::**node_locations**
Index of where nodes from the graph occur in the phased genome.

unordered_map<int64_t, **const** *Snarl* *> vg::PhasedGenome::**site_starts**
Index of which nodes are starts of Snarls.

unordered_map<int64_t, **const** *Snarl* *> vg::PhasedGenome::**site_ends**
Index of which nodes are ends of Snarls.

class *#include <phase_duplicator.hpp>* Transforms complex subregions of a graph into collections of disconnected distinct traversal haplotypes. Requires an xg index with a gPBWT in order to work. **Public Functions**

vg::PhaseDuplicator::**PhaseDuplicator**(**const** *xg*::*XG* &index)
Make a new *PhaseDuplicator* backed by the gievn index with gPBWT.

pair<*Graph*, vector<*Translation*>> vg::PhaseDuplicator::**duplicate**(**const** set<*id_t*> &subgraph, *id_t* &next_id) **const**
Duplicate out the subgraph induced by the given set of node IDs. Border nodes of the subgraph, which edges out to the rest of the graph will be identified, and all unique traversals from one border node to another will be generated as new nodes, with edges connecting them to material outside the graph and Translations embedding them in the original graph.

TODO: also generate one-end-anchored traversals and internal not- attached-to-a-border traversals, because there may be phase breaks in the region.

New IDs will be generated startign with next_id, and next_id will be updated to the next ID after the IDs of all generated material.

TODO: invent some kind of thread safe ID allocator so we can do multiple subgraphs in parallel without renumbering later.

vector<pair<*xg*::*XG*::*thread_t*, int>> `vg::PhaseDuplicator::`**`list_haplotypes`**(**const** set<*id_t*> &*subgraph*) **const**

List all the distinct haplotypes within a subgraph and their counts. Reports each haplotype in only one direction.

vector<pair<*xg*::*XG*::*thread_t*, int>> `vg::PhaseDuplicator::`**`list_haplotypes_through`**(*xg*::*XG*::*ThreadMapping* *start_node*, **const** set<*id_t*> &*subgraph*) **const**

List all the distinct haplotypes going through the given node in the given orientation, and staying within the given subgraph, along with their counts. Some may be suffixes of others.

vector<pair<*xg*::*XG*::*thread_t*, int>> `vg::PhaseDuplicator::`**`list_haplotypes_from`**(*xg*::*XG*::*ThreadMapping* *start_node*, **const** set<*id_t*> &*subgraph*) **const**

List all the distinct haplotypes actually beginning at the given node in the given orientation, and staying within the given subgraph, along with their counts. Haplotypes that begin and end at the same side may or may not be reported multiple times, in differing orientations.

vector<pair<*xg*::*XG*::*thread_t*, int>> `vg::PhaseDuplicator::`**`list_haplotypes`**(*xg*::*XG*::*ThreadMapping* *start_node*, *xg*::*XG*::*ThreadSearchState* *start_state*, **const** set<*id_t*> &*subgraph*) **const**

List all the distinct haplotypes beginning at the given node, using the given starting search state, that traverse through the subgraph.

set<*xg*::*XG*::*ThreadMapping*> `vg::PhaseDuplicator::`**`find_borders`**(**const** set<*id_t*> *subgraph*) **const**

Get the traversals that represent the borders of the subgraph, through which we can enter the subgraph.

vector<*Edge*> `vg::PhaseDuplicator::`**`find_border_edges`**(*xg*::*XG*::*ThreadMapping* *mapping*, bool *on_start*, **const** set<*id_t*> &*subgraph*) **const**

Find the edges on the given side of the given oriented ThreadMapping that cross the border of the given subgraph.

Public Static Functions

xg::*XG*::*ThreadMapping* `vg::PhaseDuplicator::`**`traverse_edge`**(**const** *Edge* &*e*, **const** *xg*::*XG*::*ThreadMapping* &*prev*)

Follow the given edge from the given node visited in the given orientation, and return the node and orientation we land in.

TODO: I think I may have written this logic already with one of the other two oriented node types (*NodeTraversal* and *NodeSide*).

xg::*XG*::*thread_t* `vg::PhaseDuplicator::`**`canonicalize`**(**const** *xg*::*XG*::*thread_t* &*thread*)
Return a copy of the given thread in a canonical orientation (either forward or reverse, whichever compares smaller).

Private Members

const *xg*::*XG* &`vg::PhaseDuplicator::`**`index`**
What XG index describes the graph we operate on?
class *#include <pictographs.hpp>*

Public Functions

`vg::Pictographs::`**`Pictographs`**(void)

`vg::Pictographs::`**`Pictographs`**(int *seed_val*)

`vg::Pictographs::`**`~Pictographs`**(void)

string `vg::Pictographs::`**`hashed`**(**const** string &*str*)

string `vg::Pictographs::`**`hashed_char`**(**const** string &*str*)

string `vg::Pictographs::`**`random`**(void)

Public Members

const string `vg::Pictographs::`**`symbols`**

const int `vg::Pictographs::`**`symbol_count`**

const string `vg::Pictographs::`**`chars`**

const int `vg::Pictographs::`**`char_count`**

Private Members

mt19937 `vg::Pictographs::`**`rng`**
struct Bundle up *Node* and *Edge* pileups. **Public Members**

repeated<*NodePileup*> `vg::Pileup::`**`node_pileups`**

repeated<*EdgePileup*> `vg::Pileup::`**`edge_pileups`**
class *#include <pileup_augmenter.hpp>* Super simple graph augmentor/caller. Idea: Idependently process *Pileup* records, using simple model to make calls that take into account read errors with diploid assumption. Edges and node positions are called independently for now. Outputs either a sample graph (only called nodes and edges) or augmented graph (include uncalled nodes and edges too). **Public Types**

typedef
typedef
typedef

typedef

typedef

Public Functions

vg::PileupAugmenter::**PileupAugmenter**(*VG* *graph, int default_quality = Default_default_quality, int min_aug_support = Default_min_aug_support)

vg::PileupAugmenter::**~PileupAugmenter**()

void vg::PileupAugmenter::**clear**()

void vg::PileupAugmenter::**write_augmented_graph**(ostream &out, bool json)

void vg::PileupAugmenter::**call_node_pileup**(**const** *NodePileup* &pileup)

void vg::PileupAugmenter::**call_edge_pileup**(**const** *EdgePileup* &pileup)

void vg::PileupAugmenter::**update_augmented_graph**()

void vg::PileupAugmenter::**map_path**(**const** *Path* &base_path, list<*Mapping*> &aug_path, bool expect_edits)

void vg::PileupAugmenter::**apply_mapping_edits**(**const** *Mapping* &base_mapping, list<*Mapping*> &aug_mappings)

void vg::PileupAugmenter::**map_paths**()

void vg::PileupAugmenter::**verify_path**(**const** *Path* &in_path, **const** list<*Mapping*> &call_path)

void vg::PileupAugmenter::**call_base_pileup**(**const** *NodePileup* &np, int64_t offset, bool insertions)

void vg::PileupAugmenter::**compute_top_frequencies**(**const** *BasePileup* &bp, **const** vector<pair<int64_t, int64_t>> &base_offsets, string &top_base, int &top_count, int &top_rev_count, string &second_base, int &second_count, int &second_rev_count, int &total_count, bool inserts)

double vg::PileupAugmenter::**total_base_quality**(**const** *BasePileup* &pb, **const** vector<pair<int64_t, int64_t>> &base_offsets, **const** string &val)

void vg::PileupAugmenter::**create_node_calls**(**const** *NodePileup* &np)

void vg::PileupAugmenter::**create_augmented_edge**(*Node* *node1, int from_offset, bool left_side1, bool aug1, *Node* *node2, int to_offset, bool left_side2, bool aug2, char cat, *StrandSupport* support)

void vg::PileupAugmenter::**annotate_augmented_node**(*Node* *node, char call, *StrandSupport* support, int64_t orig_id, int orig_offset)

void vg::PileupAugmenter::**annotate_augmented_edge**(*Edge* *edge, char *call*, *StrandSupport* *support*)

void vg::PileupAugmenter::**annotate_augmented_nodes**()

void vg::PileupAugmenter::**annotate_non_augmented_nodes**()

Public Members

VG *vg::PileupAugmenter::**_graph**

SupportAugmentedGraph vg::PileupAugmenter::**_augmented_graph**

vector<*Genotype*> vg::PileupAugmenter::**_node_calls**

vector<pair<*StrandSupport*, *StrandSupport*>> vg::PileupAugmenter::**_node_supports**

vector<*Genotype*> vg::PileupAugmenter::**_insert_calls**

vector<pair<*StrandSupport*, *StrandSupport*>> vg::PileupAugmenter::**_insert_supports**

const *Node* *vg::PileupAugmenter::**_node**

int64_t vg::PileupAugmenter::**_max_id**

NodeDivider vg::PileupAugmenter::**_node_divider**

unordered_set<int64_t> vg::PileupAugmenter::**_visited_nodes**

unordered_map<pair<*NodeSide*, *NodeSide*>, *StrandSupport*> vg::PileupAugmenter::**_called_edges**

EdgeHash vg::PileupAugmenter::**_augmented_edges**

InsertionHash vg::PileupAugmenter::**_inserted_nodes**

EdgeSupHash vg::PileupAugmenter::**_insertion_supports**

EdgeSupHash vg::PileupAugmenter::**_deletion_supports**

int vg::PileupAugmenter::**_buffer_size**

char vg::PileupAugmenter::**_default_quality**

int vg::PileupAugmenter::**_min_aug_support**

Public Static Functions

static double vg::PileupAugmenter::**safe_log**(double *v*)

static bool vg::PileupAugmenter::**missing_call**(**const** *Genotype* &*g*)

static bool vg::PileupAugmenter::**ref_call**(**const** *Genotype* &*g*)

static int vg::PileupAugmenter::**call_cat**(**const** *Genotype* &*g*)

Public Static Attributes

const double `vg::PileupAugmenter::`**`Log_zero`**

const char `vg::PileupAugmenter::`**`Default_default_quality`**

const int `vg::PileupAugmenter::`**`Default_min_aug_support`**
class *#include <pileup.hpp>* This is a collection of protobuf *NodePileup* records that are indexed on their position, as well as *EdgePileup* records. *Pileups* can be merged and streamed, and computed from Alignments. The pileup records themselves are essentially protobuf versions of lines in Samtools pileup format, with deletions represented using a graph-based notation. **Public Types**

typedef

typedef

Public Functions

`vg::Pileups::`**`Pileups`**(*VG* *graph, int *min_quality* = 0, int *max_mismatches* = 1, int *window_size* = 0, int *max_depth* = 1000, bool *use_mapq* = false)

`vg::Pileups::`**`Pileups`**(**const** *Pileups* &*other*)
copy constructor

`vg::Pileups::`**`Pileups`**(*Pileups* &&*other*)
move constructor

Pileups &`vg::Pileups::`**`operator=`**(**const** *Pileups* &*other*)
copy assignment operator

Pileups &`vg::Pileups::`**`operator=`**(*Pileups* &&*other*)
move assignment operator

`vg::Pileups::`**`~Pileups`**()
delete contents of table

void `vg::Pileups::`**`clear`**()

void `vg::Pileups::`**`to_json`**(ostream &*out*)
write to JSON

void `vg::Pileups::`**`load`**(istream &*in*)
read from protobuf

void `vg::Pileups::`**`write`**(ostream &*out*, uint64_t *buffer_size* = 5)
write to protobuf

void `vg::Pileups::`**`for_each_node_pileup`**(**const** function<void) *NodePileup*& > &*lambda*apply function to each pileup in table

NodePileup *`vg::Pileups::`**`get_node_pileup`**(int64_t *node_id*)
search hash table for node id

NodePileup *`vg::Pileups::`**`get_create_node_pileup`**(**const** *Node* *node)
get a pileup. if it's null, create a new one and insert it.

void vg::Pileups::**for_each_edge_pileup**(**const** function<void) *EdgePileup*& > *&lambda*

EdgePileup *vg::Pileups::**get_edge_pileup**(pair<*NodeSide*, *NodeSide*> *sides*)
search hash table for edge id

EdgePileup *vg::Pileups::**get_create_edge_pileup**(pair<*NodeSide*, *NodeSide*> *sides*)
get a pileup. if it's null, create a new one and insert it.

void vg::Pileups::**extend**(*Pileup* *&pileup*)

bool vg::Pileups::**insert_node_pileup**(*NodePileup* **pileup*)
insert a pileup into the table. it will be deleted by *~Pileups()*!!! return true if new pileup inserted, false if merged into existing one

bool vg::Pileups::**insert_edge_pileup**(*EdgePileup* **edge_pileup*)

void vg::Pileups::**compute_from_alignment**(*Alignment* *&alignment*)
create / update all pileups from a single alignment

void vg::Pileups::**compute_from_edit**(*NodePileup* *&pileup*, int64_t *&node_offset*, int64_t *&read_offset*, **const** *Node* *&node*, **const** *Alignment* *&alignment*, **const** *Mapping* *&mapping*, **const** *Edit* *&edit*, **const** *Edit* **next_edit*, **const** vector<int> *&mismatch_counts*, pair<**const** *Mapping* *, int64_t> *&last_match*, pair<**const** *Mapping* *, int64_t> *&last_del*, pair<**const** *Mapping* *, int64_t> *&open_del*)
create / update all pileups from an edit (called by above). query stores the current position (and nothing else).

bool vg::Pileups::**pass_filter**(**const** *Alignment* *&alignment*, int64_t *read_offset*, int64_t *length*, **const** vector<int> *&mismatches*) **const**
check base quality as well as miss match filter

Pileups &vg::Pileups::**merge**(*Pileups* *&other*)
move all entries in other object into this one. if two positions collide, they are merged. other will be left empty. this is returned

BasePileup &vg::Pileups::**merge_base_pileups**(*BasePileup* *&p1*, *BasePileup* *&p2*)
merge p2 into p1 and return 1. p2 is left an empty husk

NodePileup &vg::Pileups::**merge_node_pileups**(*NodePileup* *&p1*, *NodePileup* *&p2*)
merge p2 into p1 and return 1. p2 is lef an empty husk

EdgePileup &vg::Pileups::**merge_edge_pileups**(*EdgePileup* *&p1*, *EdgePileup* *&p2*)
merge p2 into p1 and return 1. p2 is lef an empty husk

char vg::Pileups::**combined_quality**(char *base_quality*, int *map_quality*) **const**
create combine map quality (optionally) with base quality

Public Members

VG *vg::Pileups::**_graph**

NodePileupHash vg::Pileups::**_node_pileups**
This maps from *Position* to *Pileup*.

EdgePileupHash vg::Pileups::**_edge_pileups**

int vg::Pileups::**_min_quality**
Ignore bases with quality less than this.

int vg::Pileups::**_max_mismatches**
max mismatches within window_size

int vg::Pileups::**_window_size**
number of bases to scan in each direction for mismatches

int vg::Pileups::**_max_depth**
prevent giant protobufs

bool vg::Pileups::**_use_mapq**
toggle whether we incorporate *Alignment.mapping_quality*

uint64_t vg::Pileups::**_min_quality_count**
Keep count of bases filtered by quality.

uint64_t vg::Pileups::**_max_mismatch_count**
keep count of bases filtered by mismatches

uint64_t vg::Pileups::**_bases_count**
overall count for perspective on above

Public Static Functions

void vg::Pileups::**count_mismatches**(*VG* &*graph*, **const** *Path* &*path*, vector<int> &*mismatches*, bool *skipIndels* = false)
do one pass to count all mismatches in read, so we can do mismatch filter efficiently in 2nd path. mismatches[i] stores number of mismatches in range (0, i)

static *BasePileup* *vg::Pileups::**get_base_pileup**(*NodePileup* &*np*, int64_t *offset*)
get ith *BasePileup* record

static const *BasePileup* *vg::Pileups::**get_base_pileup**(**const** *NodePileup* &*np*, int64_t *offset*)

static *BasePileup* *vg::Pileups::**get_create_base_pileup**(*NodePileup* &*np*, int64_t *offset*)
get ith *BasePileup* record, create if doesn't exist

void vg::Pileups::**parse_base_offsets**(**const** *BasePileup* &*bp*, vector<pair<int64_t, int64_t>> &*offsets*)
the bases string in *BasePileup* doesn't allow random access. This function will parse out all the offsets of snps, insertions, and deletions into one array, each offset is a pair of indexes in the bases and qualities arrays

void vg::Pileups::**casify**(string &*seq*, bool *is_reverse*)
transform case of every character in string

void vg::Pileups::**make_match**(string &*seq*, int64_t *from_length*, bool *is_reverse*)
make the sam pileup style token

void vg::Pileups::**make_insert**(string &*seq*, bool *is_reverse*)

void vg::Pileups::**make_delete**(string &*seq*, bool *is_reverse*, **const** pair<**const** *Mapping* *, int64_t> &*last_match*, **const** *Mapping* &*mapping*, int64_t *node_offset*)

void vg::Pileups::**make_delete**(string &*seq*, bool *is_reverse*, int64_t *from_id*, int64_t *from_offset*, bool *from_start*, int64_t *to_id*, int64_t *to_offset*, bool *to_end*)

void vg::Pileups::**parse_insert**(**const** string &*tok*, int64_t &*len*, string &*seq*, bool &*is_reverse*)

void vg::Pileups::**parse_delete**(**const** string &*tok*, bool &*is_reverse*, int64_t &*from_id*, int64_t &*from_offset*, bool &*from_start*, int64_t &*to_id*, int64_t &*to_offset*, bool &*to_end*)

bool vg::Pileups::**base_equal**(char *c1*, char *c2*, bool *is_reverse*)

char vg::Pileups::**extract_match**(**const** *BasePileup* &*bp*, int64_t *offset*)
get a pileup value on forward strand

string vg::Pileups::**extract**(**const** *BasePileup* &*bp*, int64_t *offset*)
get arbitrary value from offset on forward strand

struct *#include <vg.hpp>* Structure for managing parallel construction of a graph.

Public Functions

vg::VG::Plan::**Plan**(*VG* **graph*, map<long, vector<vcflib::VariantAllele>> &&*alleles*, map<pair<long, int>, vector<bool>> &&*phase_visits*, map<pair<long, int>, vector<pair<string, int>>> &&*variant_alts*, string *seq*, string *name*)

Public Members

VG *vg::VG::Plan::**graph**

map<long, vector<vcflib::VariantAllele>> vg::VG::Plan::**alleles**

map<pair<long, int>, vector<bool>> vg::VG::Plan::**phase_visits**

map<pair<long, int>, vector<pair<string, int>>> vg::VG::Plan::**variant_alts**

string vg::VG::Plan::**seq**

string vg::VG::Plan::**name**

struct A position in the graph is a node, direction, and offset. The node is stored by ID, and the offset is 0-based and counts from the start of the node in the specified orientation. The direction specifies which orientation of the node we are considering, the forward (as stored) or reverse complement. Example:

```
seq+        G A T T A C A
offset+  → 0 1 2 3 4 5 6 7

seq-        C T A A T G T
offset-  → 0 1 2 3 4 5 6 7
```

Or both at once:

```
offset-    7 6 5 4 3 2 1 0 ←
seq+        G A T T A C A
offset+  → 0 1 2 3 4 5 6 7
```

Public Members

int64 vg::Position::**node_id**
The *Node* on which the *Position* is.

int64 vg::Position::**offset**
The offset into that node's sequence at which the *Position* occurs.

bool vg::Position::**is_reverse**
True if we obtain the original sequence of the path by reverse complementing the mappings.

string vg::Position::**name**
If the position is used to represent a position against a reference path.
class *#include <support_caller.hpp>* We use this to represent a contig in the primary path, with its index and coverage info. **Public Functions**

vg::SupportCaller::PrimaryPath::**PrimaryPath**(*SupportAugmentedGraph* &*augmented*, **const** string &*ref_path_name*, size_t *ref_bin_size*)
Index the given path in the given augmented graph, and compute all the coverage bin information with the given bin size.

const *Support* &vg::SupportCaller::PrimaryPath::**get_support_at**(size_t *primary_path_offset*) **const**
Get the support at the bin appropriate for the given primary path offset.

size_t vg::SupportCaller::PrimaryPath::**get_bin_index**(size_t *primary_path_offset*) **const**
Get the index of the bin that the given path position falls in.

size_t vg::SupportCaller::PrimaryPath::**get_min_bin**() **const**
Get the bin with minimal coverage.

size_t vg::SupportCaller::PrimaryPath::**get_max_bin**() **const**
Get the bin with maximal coverage.

const *Support* &vg::SupportCaller::PrimaryPath::**get_bin**(size_t *bin*) **const**
Get the support in the given bin.

size_t vg::SupportCaller::PrimaryPath::**get_total_bins**() **const**
Get the total number of bins that the path is divided into.

Support vg::SupportCaller::PrimaryPath::**get_average_support**() **const**
Get the average support over the path.

Support vg::SupportCaller::PrimaryPath::**get_total_support**() **const**
Get the total support for the path.

PathIndex &vg::SupportCaller::PrimaryPath::**get_index**()
Get the *PathIndex* for this primary path.

const *PathIndex* &vg::SupportCaller::PrimaryPath::**get_index**() **const**
Get the *PathIndex* for this primary path.

const string &vg::SupportCaller::PrimaryPath::**get_name**() **const**
Gets the path name we are representing.

Public Static Functions

Support vg::SupportCaller::PrimaryPath::**get_average_support**(**const** map<string, *PrimaryPath*> &*paths*)

Get the average support over a collection of paths.

Protected Attributes

size_t vg::SupportCaller::PrimaryPath::**ref_bin_size**

How wide is each coverage bin along the path?

PathIndex vg::SupportCaller::PrimaryPath::**index**

This holds the index for this path.

string vg::SupportCaller::PrimaryPath::**name**

This holds the name of the path.

vector<*Support*> vg::SupportCaller::PrimaryPath::**binned_support**

What's the expected in each bin along the path? Coverage gets split evenly over both strands.

size_t vg::SupportCaller::PrimaryPath::**min_bin**

Which bin has min support?

size_t vg::SupportCaller::PrimaryPath::**max_bin**

Which bin has max support?

Support vg::SupportCaller::PrimaryPath::**total_support**

What's the total *Support* over every bin?

class *#include <progressive.hpp>* Inherit form this class to give your class *create_progress()*, *update_progress()*, and *destroy_progress()* methods, and a public show_progress field that can be toggled on and off. Must not be destroyed while a progress bar is active. Subclassed by *vg::BaseMapper*, *vg::Constructor*, *vg::Simplifier*, *vg::VariantAdder*, *vg::VG* **Public Functions**

void vg::Progressive::**preload_progress**(**const** string &*message*)

If no progress bar is currently displayed, set the message to use for the next progress bar to be created. Does nothing if show_progress is false or when a progress bar is displayed.

Public so that users of a class can provide descriptive messages for generic progress operations (like *VG*'s for_each_kmer_parallel).

void vg::Progressive::**create_progress**(**const** string &*message*, long *count*)

Create a progress bar showing the given message, with the given number of items to process. Does nothing if show_progress is false. Replaces any existing progress bar.

void vg::Progressive::**create_progress**(long *count*)

Create a progress bar with the given number of items to process, using either a default message, or the message passed to the last preload_progress call since a progress bar was destroyed. Does nothing if show_progress is false. Replaces any existing progress bar.

void vg::Progressive::**update_progress**(long *i*)

Update the progress bar, noting that the given number of items have been processed. Does nothing if no progress bar is displayed.

void vg::Progressive::**increment_progress**()
Update the progress bar, noting that one additional item has been processed. Does nothing if no progress bar is displayed.

void vg::Progressive::**destroy_progress**(void)
Destroy the current progress bar, if it exists.

Public Members

bool vg::Progressive::**show_progress**

Private Members

string vg::Progressive::**progress_message**

long vg::Progressive::**progress_count**

long vg::Progressive::**last_progress**

long vg::Progressive::**progress_seen**

ProgressBar *vg::Progressive::**progress**

template <**typename** T>
class *#include <stream.hpp>*

Public Functions

stream::ProtobufIterator::**ProtobufIterator**(*std*::istream &*in*)
Constructor.

bool stream::ProtobufIterator::**has_next**()

void stream::ProtobufIterator::**get_next**()

T stream::ProtobufIterator::**operator***()

Private Functions

void stream::ProtobufIterator::**handle**(bool *ok*)

Private Members

T stream::ProtobufIterator::**value**

uint64_t stream::ProtobufIterator::**where**

uint64_t stream::ProtobufIterator::**chunk_count**

uint64_t stream::ProtobufIterator::**chunk_idx**

google::*protobuf*::io::IstreamInputStream stream::ProtobufIterator::**raw_in**

google::*protobuf*::io::GzipInputStream `stream::ProtobufIterator::`**`gzip_in`**

google::*protobuf*::io::CodedInputStream `stream::ProtobufIterator::`**`coded_in`**
class *#include <gssw_aligner.hpp>* An aligner that uses read base qualities to adjust its scores and alignments. Inherits from *vg::BaseAligner* **Public Functions**

`QualAdjAligner::`**`QualAdjAligner`**(int8_t *_match* = default_match, int8_t *_mismatch* = default_mismatch, int8_t *_gap_open* = default_gap_open, int8_t *_gap_extension* = default_gap_extension, int8_t *_full_length_bonus* = default_full_length_bonus, int8_t *_max_scaled_score* = default_max_scaled_score, uint8_t *_max_qual_score* = default_max_qual_score, double *gc_content* = default_gc_content)

`vg::QualAdjAligner::`**`~QualAdjAligner`**(void)

void `QualAdjAligner::`**`align`**(Alignment &*alignment*, Graph &*g*, bool *traceback_aln*, bool *print_score_matrices*)
Store optimal local alignment against a graph in the *Alignment* object. Gives the full length bonus separately on each end of the alignment. Assumes that graph is topologically sorted by node index.

void `QualAdjAligner::`**`align_global_banded`**(Alignment &*alignment*, Graph &*g*, int32_t *band_padding* = 0, bool *permissive_banding* = true)

void `QualAdjAligner::`**`align_pinned`**(Alignment &*alignment*, Graph &*g*, bool *pin_left*)

void `QualAdjAligner::`**`align_global_banded_multi`**(Alignment &*alignment*, vector<Alignment> &*alt_alignments*, Graph &*g*, int32_t *max_alt_alns*, int32_t *band_padding* = 0, bool *permissive_banding* = true)

void `QualAdjAligner::`**`align_pinned_multi`**(Alignment &*alignment*, vector<Alignment> &*alt_alignments*, Graph &*g*, bool *pin_left*, int32_t *max_alt_alns*)

void `vg::QualAdjAligner::`**`init_mapping_quality`**(double *gc_content*)

int32_t `QualAdjAligner::`**`score_exact_match`**(**const** Alignment &*aln*, size_t *read_offset*, size_t *length*)
Compute the score of an exact match in the given alignment, from the given offset, of the given length.

int32_t `QualAdjAligner::`**`score_exact_match`**(**const** string &*sequence*, **const** string &*base_quality*) **const**
Compute the score of an exact match of the given sequence with the given qualities. Qualities may be ignored by some implementations.

int32_t `QualAdjAligner::`**`score_exact_match`**(string::const_iterator *seq_begin*, string::const_iterator *seq_end*, string::const_iterator *base_qual_begin*) **const**
Compute the score of an exact match of the given range of sequence with the given qualities. Qualities may be ignored by some implementations.

Public Members

uint8_t vg::QualAdjAligner::**max_qual_score**

int8_t vg::QualAdjAligner::**scale_factor**

Private Functions

void QualAdjAligner::**align_internal**(Alignment &*alignment*, vector<Alignment> **multi_alignments*, Graph &*g*, bool *pinned*, bool *pin_left*, int32_t *max_alt_alns*, bool *traceback_aln*, bool *print_score_matrices*)

class *#include <readfilter.hpp>*

Public Functions

int vg::ReadFilter::**filter**(istream **alignment_stream*, *xg*::*XG* **xindex* = nullptr)

Filter the alignments available from the given stream, placing them on standard output or in the appropriate file. Returns 0 on success, exit code to use on error.

If an XG index is required, use the specified one. If one is required and not provided, the function will complain and return nonzero.

TODO: Refactor to be less CLI-aware and more modular-y.

bool vg::ReadFilter::**trim_ambiguous_ends**(*xg*::*XG* **index*, *Alignment* &*alignment*, int *k*)

Look at either end of the given alignment, up to k bases in from the end. See if that tail of the alignment is mapped such that another embedding in the given graph can produce the same sequence as the sequence along the embedding that the read actually has, and if so trim back the read.

In the case of softclips, the aligned portion of the read is considered, and if trimmign is required, the softclips are hard-clipped off.

Returns true if the read had to be modified, and false otherwise.

MUST NOT be called with a null index.

Public Members

double vg::ReadFilter::**min_secondary**

double vg::ReadFilter::**min_primary**

bool vg::ReadFilter::**frac_score**

bool vg::ReadFilter::**sub_score**

int vg::ReadFilter::**max_overhang**

int vg::ReadFilter::**min_end_matches**

int vg::ReadFilter::**context_size**

bool vg::ReadFilter::**verbose**

double vg::ReadFilter::**min_mapq**

int vg::ReadFilter::**repeat_size**

int vg::ReadFilter::**defray_length**

int vg::ReadFilter::**defray_count**

bool vg::ReadFilter::**drop_split**

int vg::ReadFilter::**threads**

string vg::ReadFilter::**regions_file**

string vg::ReadFilter::**outbase**

bool vg::ReadFilter::**append_regions**

Private Functions

bool vg::ReadFilter::**has_repeat**(*Alignment* &*aln*, int *k*)
* quick and dirty filter to see if removing reads that can slip around and still map perfectly helps vg call. returns true if at either end of read sequence, at least k bases are repetitive, checking repeats of up to size 2k

bool vg::ReadFilter::**trim_ambiguous_end**(*xg*::*XG* **index*, *Alignment* &*alignment*, int *k*)
Trim only the end of the given alignment, leaving the start alone. Two calls of this implement trim_ambiguous_ends above.

bool vg::ReadFilter::**is_split**(*xg*::*XG* **index*, *Alignment* &*alignment*)
Return false if the read only follows edges in the xg index, and true if the read is split (or just incorrect) and takes edges not in the index.

Throws an error if no XG index is specified.

class *#include <genotypekit.hpp>* Inherits from *vg::TraversalFinder*

Public Functions

vg::ReadRestrictedTraversalFinder::**ReadRestrictedTraversalFinder**(*VG* &*graph*, *SnarlManager* &*snarl_manager*, **const** map<string, *Alignment* *> &*reads_by_name*, int *min_recurrence* = 2, int *max_path_search_steps* = 100)

vg::ReadRestrictedTraversalFinder::**~ReadRestrictedTraversalFinder**()

vector<*SnarlTraversal*> vg::ReadRestrictedTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)
For the given site, emit all traversals with unique sequences that run from start to end, out of the paths in the graph. Uses the map of reads by name to determine if a path is a read or a real named path. *Paths*

through the site supported only by reads are subject to a min recurrence count, while those supported by actual embedded named paths are not.

Private Members

VG &vg::ReadRestrictedTraversalFinder::**graph**

SnarlManager &vg::ReadRestrictedTraversalFinder::**snarl_manager**

const map<string, *Alignment* *> &vg::ReadRestrictedTraversalFinder::**reads_by_name**

int vg::ReadRestrictedTraversalFinder::**min_recurrence**

int vg::ReadRestrictedTraversalFinder::**max_path_search_steps**

struct *#include <region.hpp>*

Public Members

string vg::Region::**seq**

int64_t vg::Region::**start**

int64_t vg::Region::**end**

struct *#include <genome_state.hpp>* We can use this to replace a local haplotype within one or more snarls. We also could just express all the deletions and insertions in terms of this. Inherits from *vg::GenomeStateCommand*

Public Functions

GenomeStateCommand *vg::ReplaceLocalHaplotypeCommand::**execute**(*GenomeState* &*state*) **const**

Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual vg::ReplaceLocalHaplotypeCommand::**~ReplaceLocalHaplotypeCommand**()

Public Members

unordered_map<**const** *Snarl* *, vector<size_t>> vg::ReplaceLocalHaplotypeCommand::**deletions**

Holds, for each snarl, the overall lanes that have to be deleted. Each deleted overall lane for a snarl that is still traversed from its parent needs to be replaced in insertions. Deletions happen first.

unordered_map<**const** *Snarl* *, vector<vector<pair<*handle_t*, size_t>>>> vg::ReplaceLocalHaplotypeCommand::**ins**

Holds, for each *Snarl*, the visits and their lane assignments that have to be inserted.

struct *#include <genome_state.hpp>* Inherits from *vg::GenomeStateCommand*

Public Functions

GenomeStateCommand *vg::ReplaceSnarlHaplotypeCommand::**execute**(*GenomeState* &*state*) **const**

Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual vg::ReplaceSnarlHaplotypeCommand::**~ReplaceSnarlHaplotypeCommand**()

Public Members

const *Snarl* *vg::ReplaceSnarlHaplotypeCommand::**snarl**
Which snarl are we working on?

size_t vg::ReplaceSnarlHaplotypeCommand::**lane**
Which lane in the snarl are we changing?

vector<*handle_t*> vg::ReplaceSnarlHaplotypeCommand::**haplotype**
What fully specified haplotype should we replace it with? This gets around any problems to do with increases or decreases in the copy numbers of child snarls being underspecified.

class *#include <genotypekit.hpp>* This *TraversalFinder* is derived from the old vg call code, and emits at least one traversal representing every node, and one traversal representing every edge. Inherits from *vg::TraversalFinder*

Public Functions

vg::RepresentativeTraversalFinder::**RepresentativeTraversalFinder**(*SupportAugmentedGraph* &*augmented*, *SnarlManager* &*snarl_manager*, size_t *max_depth*, size_t *max_width*, size_t *max_bubble_paths*, function<*PathIndex* *(**const** *Snarl*&)> *get_index* = [](const Snarl &s){return nullptr;}Make a new *RepresentativeTraversalFinder* to find traversals. Uses the given augmented graph as the graph with coverage annotation, and reasons about child snarls with the given *SnarlManager*. Explores up to max_depth in the BFS search when trying to find its way across snarls, and considers up to max_width search states at a time. When combining search results on either side of a graph element to be represented, thinks about max_bubble_paths combinations.

Uses the given get_index function to try and find a *PathIndex* for a reference path traversing a child snarl.

virtual vg::RepresentativeTraversalFinder::**~RepresentativeTraversalFinder**()

vector<*SnarlTraversal*> vg::RepresentativeTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)
Find traversals to cover the nodes and edges of the snarl. Always emits the primary path traversal first, if applicable.

Public Members

bool vg::RepresentativeTraversalFinder::**verbose**
Should we emit verbose debugging info?

Protected Functions

Path vg::RepresentativeTraversalFinder::**find_backbone**(**const** *Snarl* &*site*)
Find a *Path* that runs from the start of the given snarl to the end, which we can use to backend our traversals into when a snarl is off the primary path.

pair<*Support*, vector<*Visit*>> vg::RepresentativeTraversalFinder::**find_bubble**(*Node* **node*, *Edge* **edge*, **const** *Snarl* **snarl*, *PathIndex* &*index*, **const** *Snarl* &*site*)
Given an edge or node in the augmented graph, look out from the edge or node or snarl in both directions to find a shortest bubble relative to the path, with a consistent orientation. The bubble may not visit the same node twice.

Exactly one of edge and node and snarl must be not null.

Takes a max depth for the searches producing the paths on each side.

Return the ordered and oriented nodes in the bubble, with the outer nodes being oriented forward along the path for which an index is provided, and with the first node coming before the last node in the reference. Also return the minimum support found on any edge or node in the bubble (including the reference node endpoints and their edges which aren't stored in the path).

Support vg::RepresentativeTraversalFinder::**min_support_in_path**(**const** list<*Visit*> &*path*)
Get the minimum support of all nodes and edges in path

set<pair<size_t, list<*Visit*>>> vg::RepresentativeTraversalFinder::**bfs_left**(*Visit* *visit*, *PathIndex* &*index*, bool *stopIfVisited* = false, **const** *Snarl* **in_snarl* = nullptr)
Do a breadth-first search left from the given node traversal, and return lengths and paths starting at the given node and ending on the given indexed path. Refuses to visit nodes with no support, if support data is available in the augmented graph.

set<pair<size_t, list<*Visit*>>> vg::RepresentativeTraversalFinder::**bfs_right**(*Visit visit*, *PathIndex &index*, bool *stopIfVisited* = false, **const** *Snarl *in_snarl* = nullptr)
Do a breadth-first search right from the given node traversal, and return lengths and paths starting at the given node and ending on the given indexed path. Refuses to visit nodes with no support, if support data is available in the augmented graph.

size_t vg::RepresentativeTraversalFinder::**bp_length**(**const** list<*Visit*> &*path*)
Get the length of a path through nodes, in base pairs.

Protected Attributes

SupportAugmentedGraph &vg::RepresentativeTraversalFinder::**augmented**
The annotated, augmented graph we're finding traversals in.

SnarlManager &vg::RepresentativeTraversalFinder::**snarl_manager**
The *SnarlManager* managiung the snarls we use.

function<*PathIndex* *(**const** *Snarl*&)> vg::RepresentativeTraversalFinder::**get_index**
We keep around a function that can be used to get an index for the appropriate path to use to scaffold a given site, or null if no appropriate index exists.

size_t vg::RepresentativeTraversalFinder::**max_depth**
What DFS depth should we search to?

size_t vg::RepresentativeTraversalFinder::**max_width**
How many DFS searches should we let there be on the stack at a time?

size_t vg::RepresentativeTraversalFinder::**max_bubble_paths**
How many search intermediates can we allow?

class *#include <sampler.hpp>* Generate Alignments (with or without mutations, and in pairs or alone) from an XG index. **Public Functions**

vg::Sampler::**Sampler**(*xg*::*XG* *x, int *seed* = 0, bool *forward_only* = false, bool *allow_Ns* = false, **const** vector<string> &*source_paths* = { })

void vg::Sampler::**set_source_paths**(**const** vector<string> &*source_paths*)
Make a path sampling distribution based on relative lengths.

pos_t vg::Sampler::**position**(void)

string vg::Sampler::**sequence**(size_t *length*)

Alignment vg::Sampler::**alignment**(size_t *length*)
Get an alignment against the whole graph, or against the source path if one is selected.

Alignment vg::Sampler::**alignment_to_graph**(size_t *length*)
Get an alignment against the whole graph.

Alignment vg::Sampler::**alignment_to_path**(**const** string &*source_path*, size_t *length*)
Get an alignment against the currently set source_path.

Alignment vg::Sampler::**alignment_with_error**(size_t *length*, double *base_error*, double *indel_error*)

vector<*Alignment*> vg::Sampler::**alignment_pair**(size_t *read_length*, size_t *fragment_length*, double *fragment_std_dev*, double *base_error*, double *indel_error*)

size_t vg::Sampler::**node_length**(*id_t id*)

char vg::Sampler::**pos_char**(*pos_t pos*)

map<*pos_t*, char> vg::Sampler::**next_pos_chars**(*pos_t pos*)

Alignment vg::Sampler::**mutate**(**const** *Alignment* &*aln*, double *base_error*, double *indel_error*)

vector<*Edit*> vg::Sampler::**mutate_edit**(**const** *Edit* &*edit*, **const** *pos_t* &*position*, double *base_error*, double *indel_error*, **const** string &*bases*, uniform_real_distribution<double> &*rprob*, uniform_int_distribution<int> &*rbase*)
Mutate the given edit, producing a vector of edits that should replace it. *Position* is the position of the start of the edit, and is updated to point to the next base after the mutated edit.

string vg::Sampler::**alignment_seq**(**const** *Alignment* &*aln*)

bool vg::Sampler::**is_valid**(**const** *Alignment* &*aln*)
Return true if the alignment is semantically valid against the XG index we wrap, and false otherwise. Checks from_lengths on mappings to make sure all node bases are accounted for. Won't accept alignments with internal jumps between graph locations or regions; all skipped bases need to be accounted for by deletions.

Public Members

xg::*XG* *vg::Sampler::**xgidx**

LRUCache<*id_t*, *Node*> vg::Sampler::**node_cache**

LRUCache<*id_t*, vector<*Edge*>> vg::Sampler::**edge_cache**

mt19937 vg::Sampler::**rng**

int64_t vg::Sampler::**nonce**

bool vg::Sampler::**forward_only**

bool vg::Sampler::**no_Ns**

vector<string> vg::Sampler::**source_paths**

discrete_distribution vg::Sampler::**path_sampler**

class *#include <cluster.hpp>* Iterate over pairsets of integers in a pseudorandom but deterministic order. We use the same permutation every time for a given number of items to pair up. **Public Types**

using

Public Functions

`vg::ShuffledPairs::`**`ShuffledPairs`**`(`size_t *num_items*`)`
Make a new iterable pairing up the given number of items.

ShuffledPairs::*iterator* `vg::ShuffledPairs::`**`begin`**`()` **const**
Get an iterator to the first pair.

ShuffledPairs::*iterator* `vg::ShuffledPairs::`**`end`**`()` **const**
Get an iterator to the past-the-end pair.

Private Members

size_t `vg::ShuffledPairs::`**`num_items`**

size_t `vg::ShuffledPairs::`**`num_pairs`**

size_t `vg::ShuffledPairs::`**`larger_prime`**

size_t `vg::ShuffledPairs::`**`primitive_root`**
class *#include <genotypekit.hpp>* Inherits from *vg::ConsistencyCalculator* **Public Functions**

`vg::SimpleConsistencyCalculator::`**`~SimpleConsistencyCalculator`**`()`

vector<bool> `vg::SimpleConsistencyCalculator::`**`calculate_consistency`**`(`**const** *Snarl &site*, **const** vector<*SnarlTraversal*> *&traversals*, **const** *Alignment &read*`)` **const**
Return true or false for each tarversal of the site, depending on if the read is consistent with it or not.
class *#include <genotypekit.hpp>* Inherits from *vg::TraversalSupportCalculator* **Public Functions**

`vg::SimpleTraversalSupportCalculator::`**`~SimpleTraversalSupportCalculator`**`()`

vector<*Support*> vg::SimpleTraversalSupportCalculator::**calculate_supports**(**const** *Snarl &site*, **const** vector<*SnarlTraversal*> *&traversals*, **const** vector<*Alignment* *> *&reads*, **const** vector<vector<bool>> *&consistencies*) **const**

Return Supports for all the SnarlTraversals, given the reads and their consistency flags.

class *#include <simplifier.hpp>* A class that can be used to simplify a graph, by repeatedly popping leaf bubbles under a certain size. Keeps graph paths and an optional set of BED- like features up to date. TODO: doesn't handle path start and end positions within nodes. Inherits from *vg::Progressive* **Public Functions**

vg::Simplifier::**Simplifier**(*VG &graph*)

Make a simplifier that simplifies the given graph in place.

pair<size_t, size_t> vg::Simplifier::**simplify_once**(size_t *iteration* = 0)

Simplify the graph by one step. Returns the number of nodes deleted and the number of edges deleted. Can be passed an iteration for its progress messages.

void vg::Simplifier::**simplify**()

Simplify the graph until material stops being deleted or the maximum iteration count is reached.

Public Members

size_t vg::Simplifier::**min_size**

What's the miniumum size of a bubble to keep, in involved bases? Everything smaller will get squished away.

size_t vg::Simplifier::**max_iterations**

How many iterations of simplification should we allow in a *simplify()* call?

bool vg::Simplifier::**drop_hairpin_paths**

Should we simplify bubbles where paths come in and leave through the enterance node (and delete those paths) (true)? Or should we leave those bubbles unsimplified?

FeatureSet vg::Simplifier::**features**

Stores the features in the graph, and gets updated as simplification proceeds. The user should load the features in and pull them out.

Protected Attributes

VG &vg::Simplifier::**graph**
Holds a reference to the graph we're simplifying.

SnarlManager vg::Simplifier::**site_manager**
This keeps track of the sites to simplify.

TrivialTraversalFinder vg::Simplifier::**traversal_finder**
This is used to find traversals of those sites.
struct Describes a subgraph that is connected to the rest of the graph by two nodes. **Public Members**

SnarlType vg::Snarl::**type**
What type of snarl is this?

Visit vg::Snarl::**start**
Visits that connect the *Snarl* to the rest of the graph.

points *INTO* the snarl

Visit vg::Snarl::**end**
points *OUT OF* the snarl

Snarl vg::Snarl::**parent**
If this *Snarl* is nested in another, this field should be filled in with a *Snarl* that has the start and end visits filled in (other information is optional/extraneous)

string vg::Snarl::**name**
Allows snarls to be named, e.g. by the hash of the VCF variant they come from.

bool vg::Snarl::**start_self_reachable**
Indicate whether there is a reversing path contained in the *Snarl* from either the start to itself or the end to itself

bool vg::Snarl::**end_self_reachable**

bool vg::Snarl::**start_end_reachable**
Indicate whether the start of the *Snarl* is connected through to the end.

bool vg::Snarl::**directed_acyclic_net_graph**
Indicate whether the snarl's net graph is free of directed cycles.
class *#include <genotypekit.hpp>* Represents a strategy for finding (nested) sites in a vg graph that can be described by snarls. Polymorphic base class/interface. Subclassed by *vg::CactusSnarlFinder* **Public Functions**

virtual vg::SnarlFinder::**~SnarlFinder**()

virtual *SnarlManager* vg::SnarlFinder::**find_snarls**()
= 0Run a function on all root-level NestedSites in parallel. Site trees are passed by value so they have a clear place to live during parallel operations.
class *#include <snarls.hpp>* A structure to keep track of the tree relationships between Snarls and perform utility algorithms on them **Public Functions**

template <typename SnarlIterator**>**
vg::SnarlManager::**SnarlManager**(SnarlIterator *begin*, SnarlIterator *end*)
Construct a *SnarlManager* for the snarls returned by an iterator Also covers iterators of chains of snarls.

vg::SnarlManager::**SnarlManager**(istream &*in*)
Construct a *SnarlManager* for the snarls contained in an input stream.

vg::SnarlManager::**SnarlManager**()
Default constructor.

vg::SnarlManager::**~SnarlManager**()
Destructor.

vg::SnarlManager::**SnarlManager**(**const** *SnarlManager* &*other*)
Cannot be copied because of all the internal pointer indexes.

SnarlManager &vg::SnarlManager::**operator=**(**const** *SnarlManager* &*other*)

vg::SnarlManager::**SnarlManager**(*SnarlManager* &&*other*)
Can be moved.

SnarlManager &vg::SnarlManager::**operator=**(*SnarlManager* &&*other*)

const vector<**const** *Snarl* *> &vg::SnarlManager::**children_of**(**const** *Snarl* **snarl*) **const**
Returns a vector of pointers to the children of a *Snarl*. If given null, returns the top-level root snarls.

const *Snarl* *vg::SnarlManager::**parent_of**(**const** *Snarl* **snarl*) **const**
Returns a pointer to the parent of a *Snarl* or nullptr if there is none.

const *Snarl* *vg::SnarlManager::**into_which_snarl**(int64_t *id*, bool *reverse*) **const**
Returns the *Snarl* that a traversal points into at either the start or end, or nullptr if the traversal does not point into any *Snarl*. Note that Snarls store the end *Visit* pointing out of rather than into the *Snarl*, so they must be reversed to query it.

const *Snarl* *vg::SnarlManager::**into_which_snarl**(**const** *Visit* &*visit*) **const**
Returns the *Snarl* that a *Visit* points into. If the *Visit* contains a *Snarl* rather than a node ID, returns a pointer the managed version of that snarl.

const Chain *vg::SnarlManager::**chain_of**(**const** *Snarl* **snarl*) **const**
Get the Chain that the given snarl participates in. Instead of asking this class to walk the chain for you, use ChainIterators on this chain.

bool vg::SnarlManager::**in_nontrivial_chain**(**const** *Snarl* **here*) **const**
Return true if a *Snarl* is part of a nontrivial chain of more than one snarl.

const deque<Chain> &vg::SnarlManager::**chains_of**(**const** *Snarl* **snarl*) **const**
Get all the snarls in all the chains under the given parent snarl. If the parent snarl is null, gives the top-level chains that connect and contain the top-level root snarls. Unary snarls and snarls in trivial chains will be presented as their own chains. Snarls are not necessarily oriented appropriately given their ordering in the chain. Useful for making a net graph.

NetGraph vg::SnarlManager::**net_graph_of**(**const** *Snarl* **snarl*, **const** *HandleGraph* **graph*, bool *use_internal_connectivity* = true) **const**
Get the net graph of the given *Snarl*'s contents, using the given backing *HandleGraph*. If use_internal_connectivity is false, each chain and unary child snarl is treated as an ordinary node which is assumed to be only traversable from one side to the other. Otherwise, traversing the graph works like it would if you actually went through the internal graphs fo child snarls.

bool vg::SnarlManager::**is_leaf**(**const** *Snarl* **snarl*) **const**
Returns true if snarl has no children and false otherwise.

bool vg::SnarlManager::**is_root**(**const** *Snarl* **snarl*) **const**
Returns true if snarl has no parent and false otherwise.

const vector<**const** *Snarl* *> &vg::SnarlManager::**top_level_snarls**() **const**
Returns a reference to a vector with the roots of the *Snarl* trees.

void vg::SnarlManager::**flip**(**const** *Snarl* *snarl)
Reverses the orientation of a snarl.

const *Snarl* *vg::SnarlManager::**add_snarl**(**const** *Snarl* &*new_snarl*)
Add the given snarl to the *SnarlManager* as neither a root nor a child of any other snarl. The snarl must eventually either be added to a parent snarl or as a root snarl through *add_chain()* or it will not be visible.

void vg::SnarlManager::**add_chain**(**const** Chain &*new_chain*, **const** *Snarl* *chain_parent)
Add the given chain of snarls that have already been added with *add_snarl()*. Parents the chain to the given parent snarl, also added with *add_snarl()*. If the parent is null, makes the chain a root chain and all of its snarls root snarls. Note that the chains are allowed to be reallocated until the last child chain of a snarl is added.

pair<unordered_set<*Node* *>, unordered_set<*Edge* *>> vg::SnarlManager::**shallow_contents**(**const** *Snarl* *snarl*, *VG* &*graph*, bool *include_boundary_no* **const**
Returns the Nodes and Edges contained in this *Snarl* but not in any child Snarls (always includes the Nodes that form the boundaries of child Snarls, optionally includes this *Snarl*'s own boundary Nodes)

pair<unordered_set<*Node* *>, unordered_set<*Edge* *>> vg::SnarlManager::**deep_contents**(**const** *Snarl* *snarl*, *VG* &*graph*, bool *include_boundary_nodes*) **const**
Returns the Nodes and Edges contained in this *Snarl*, including those in child Snarls (optionally includes *Snarl*'s own boundary Nodes)

vector<*Visit*> vg::SnarlManager::**visits_left**(**const** *Visit* &*visit*, *VG* &*graph*, **const** *Snarl* *in_snarl) **const**
Look left from the given visit in the given graph and gets all the attached Visits to nodes or snarls.

vector<*Visit*> vg::SnarlManager::**visits_right**(**const** *Visit* &*visit*, *VG* &*graph*, **const** *Snarl* *in_snarl) **const**
Look left from the given visit in the given graph and gets all the attached Visits to nodes or snarls.

unordered_map<pair<int64_t, bool>, **const** *Snarl* *> vg::SnarlManager::**snarl_boundary_index**() **const**
Returns a map from all *Snarl* boundaries to the *Snarl* they point into. Note that this means that end boundaries will be reversed.

unordered_map<pair<int64_t, bool>, **const** *Snarl* *> vg::SnarlManager::**snarl_start_index**() **const**
Returns a map from all *Snarl* start boundaries to the *Snarl* they point into.

unordered_map<pair<int64_t, bool>, **const** *Snarl* *> `vg::SnarlManager::`**`snarl_end_index`**`()` **const**
Returns a map from all *Snarl* end boundaries to the *Snarl* they point into. Note that this means that end boundaries will be reversed.

void `vg::SnarlManager::`**`for_each_top_level_snarl`**`(`**const** function<void)`(`**const** *Snarl* *> *&lambda* **const**Execute a function on all top level sites.

void `vg::SnarlManager::`**`for_each_snarl_preorder`**`(`**const** function<void)`(`**const** *Snarl* *> *&lambda* **const**Execute a function on all sites in a preorder traversal.

void `vg::SnarlManager::`**`for_each_top_level_snarl_parallel`**`(`**const** function<void)`(`**const** *Snarl* *> *&lambda* **const**Execute a function on all top level sites in parallel.

void `vg::SnarlManager::`**`for_each_snarl_parallel`**`(`**const** function<void)`(`**const** *Snarl* *> *&lambda* **const**Execute a function on all sites in parallel.

const *Snarl* *`vg::SnarlManager::`**`manage`**`(`**const** *Snarl* *&not_owned*`)` **const**
Given a *Snarl* that we don't own (like from a *Visit*), find the pointer to the managed copy of that *Snarl*.

Private Types

using Define the key type.

Private Functions

SnarlManager::*key_t* `vg::SnarlManager::`**`key_form`**`(`**const** *Snarl* *snarl`)` **const**
Converts *Snarl* to the form used as keys in internal data structures.

void `vg::SnarlManager::`**`build_indexes`**`()`
Builds tree indexes after Snarls have been added to the snarls vector.

deque<Chain> `vg::SnarlManager::`**`compute_chains`**`(`**const** vector<**const** *Snarl* *> *&input_snarls*`)`
Actually compute chains for a set of already indexed snarls, which is important when chains were not provided. Returns the chains.

Visit `vg::SnarlManager::`**`next_snarl`**`(`**const** *Visit* *&here*`)` **const**
Get a *Visit* to the snarl coming after the given *Visit* to a snarl, or a *Visit* with no *Snarl* no next snarl exists. Accounts for snarls' orientations.

Visit `vg::SnarlManager::`**`prev_snarl`**`(`**const** *Visit* *&here*`)` **const**
Get a *Visit* to the snarl coming before the given *Visit* to a snarl, or a *Visit* with no *Snarl* no previous snarl exists. Accounts for snarls' orientations.

const *Snarl* *`vg::SnarlManager::`**`snarl_sharing_start`**`(`**const** *Snarl* *here`)` **const**
Get the *Snarl*, if any, that shares this *Snarl*'s start node as either its start or its end. Does not count this snarl, even if this snarl is unary. Basic operation used to traverse a chain. Caller must account for snarls' orientations within a chain.

const *Snarl* *`vg::SnarlManager::`**`snarl_sharing_end`**`(`**const** *Snarl* *here`)` **const**
Get the *Snarl*, if any, that shares this *Snarl*'s end node as either its start or its end. Does not count this snarl, even if this snarl is unary. Basic operation used to traverse a chain. Caller must account for snarls' orientations within a chain.

Private Members

deque<*Snarl*> `vg::SnarlManager::`**`snarls`**
Master list of the snarls in the graph. Use a deque so pointers never get invalidated but we still have some locality.

vector<**const** *Snarl* *> `vg::SnarlManager::`**`roots`**
Roots of snarl trees.

deque<Chain> `vg::SnarlManager::`**`root_chains`**
Chains of root-level snarls. Uses a deque so Chain* pointers don't get invalidated.

unordered_map<*key_t*, vector<**const** *Snarl* *>> `vg::SnarlManager::`**`children`**
Map of snarls to the child snarls they contain.

unordered_map<*key_t*, deque<Chain>> `vg::SnarlManager::`**`child_chains`**
Map of snarls to the child chains they contain Uses a deque so Chain* pointers don't get invalidated.

unordered_map<*key_t*, **const** *Snarl* *> `vg::SnarlManager::`**`parent`**
Map of snarls to their parent snarls.

unordered_map<*key_t*, **const** Chain *> `vg::SnarlManager::`**`parent_chain`**
Map of snarls to the chain each appears in.

unordered_map<*key_t*, *Snarl* *> `vg::SnarlManager::`**`self`**
Map of snarl keys to the pointer to the managed copy in the snarls vector. Is non-const so we can do flip nicely.

unordered_map<pair<int64_t, bool>, **const** *Snarl* *> `vg::SnarlManager::`**`snarl_into`**
Map of node traversals to the snarls they point into.

class *#include <genome_state.hpp>* Represents the state of a snarl: zero or more haplotypes traversing its *NetGraph*. Only admits full-length traversals of a snarl from start to end. Every traversing haplotype is assigned a "lane" number at which it traverses the snarl. Lane is the same looking backward or forward through the snarl. Within each child snarl or child node, the traversal is also assigned a lane. The lane assignments at the start and end nodes are the same, and define the lane assignments for the overall snarl. Traversals can be inserted at any lane number in any internal node. **Public Functions**

`vg::SnarlState::`**`SnarlState`**(**const** *NetGraph* *graph)
Create a *SnarlState* that uses the given net graph.

void `vg::SnarlState::`**`dump`**() **const**
Dump internal state to cerr.

size_t `vg::SnarlState::`**`size`**() **const**
How many haplotypes traverse this snarl?

void `vg::SnarlState::`**`trace`**(size_t *overall_lane*, bool *backward*, **const** function<void(*handle_t*&, size_t > &*iteratee*) **const**
Trace the haplotype int eh given overall lane in the given orientation. Yields the oriented handles visited and the per-forward-handle lane assignments.

void `vg::SnarlState::`**`insert`**(**const** vector<pair<*handle_t*, size_t>> &*haplotype*)
Insert the given traversal of this snarl from start to end, with the given lane assignments for each oriented handle. If handles to the same node or child snarl appear more than once, their lane numbers must be strictly increasing.

const vector<pair<*handle_t*, size_t>> &vg::SnarlState::**append**(**const** vector<*handle_t*> &*haplotype*, bool *backward* = false)

Insert the given traversal of this snarl from start to end or end to start (as determined by the backward flag), assigning each visit to a handle to the next available lane. Returns the haplotype annotated with lane assignments. If handles to the same node or child snarl appear more than once, their lane numbers will be strictly increasing.

const vector<pair<*handle_t*, size_t>> &vg::SnarlState::**insert**(size_t *overall_lane*, **const** vector<*handle_t*> &*haplotype*, bool *backward* = false)

Insert the given traversal of this snarl from start to end or end to start (as determined by the backward flag), assigning it to the given overall lane. Returns the haplotype annotated with lane assignments for all the internal handles. If the internal handles represent child snarls, this can be used to recurse down and insert traversals of them at the right lanes. If handles to the same node or child snarl appear more than once, their assigned lane numbers will be strictly increasing. Returns the haplotype annotated with lane assignments.

vector<pair<*handle_t*, size_t>> vg::SnarlState::**erase**(size_t *overall_lane*)

Erase the traversal of this haplotype in the given overall lane. Shifts everything in a higher lane 1 rank down. Returns the erased haplotype and its old lane assignments.

void vg::SnarlState::**swap**(size_t *lane1*, size_t *lane2*)

Swap the traversals of this haplotype in the two given overall lanes. Internal lane assignments (as are used by child snarls) are not affected.

Protected Attributes

vector<vector<pair<*handle_t*, size_t>>> vg::SnarlState::**haplotypes**

unordered_map<*handle_t*, vector<decltype(haplotypes)::value_type::iterator>> vg::SnarlState::**net_node_lanes**

const *NetGraph* *vg::SnarlState::**graph**

We need to keep track of the net graph, because we may need to traverse haplotypes forward or reverse and we need to flip things.

struct Describes a walk through a *Snarl* where each step is given as either a node or a child *Snarl* (leaving the walk through the child *Snarl* to another *SnarlTraversal*) **Public Members**

repeated<*Visit*> vg::SnarlTraversal::**visits**

Steps of the walk through a *Snarl*, including the start and end nodes. If the traversal includes a *Visit* that represents a *Snarl*, both the node entering the *Snarl* and the node leaving the *Snarl* should be included in the traversal.

string vg::SnarlTraversal::**name**

The name of the traversal can be used for a variant allele id (e.g. <parentSnarlHash>_0, <parentSnarlHash>_1... or by some other arbitrary annotation , unique or non-unique, e.g. deleteterious, gain_of_function, etc., though these will be lost in any indices).

class *#include <srpe.hpp>***Public Functions**

void vg::SRPE::**call_svs_paired_end**(*vg*::*VG* *graph, ifstream &*gamstream*, vector<*BREAKPOINT*> &*bps*, string *refpath* = "")

void vg::SRPE::**call_svs_split_read**(*vg*::*VG* *graph, ifstream &*gamstream*, vector<*BREAKPOINT*> &*bps*, string *refpath* = "")

void vg::SRPE::**call_svs**(string *graphfile*, string *gamfile*, string *refpath*)

double vg::SRPE::**discordance_score**(vector<*Alignment*> *alns*, *VG* **subgraph*)

void vg::SRPE::**aln_to_bseq**(*Alignment* &*a*, bseq1_t **read*)

void vg::SRPE::**assemble**(vector<*Alignment*> *alns*, vector<fml_utg_t> &*unitigs*)
function assemble inputs: a vector of Alignments to be assembled (based on their sequences) outputs:

void vg::SRPE::**assemble**(string *refpath*, int64_t *start_pos*, int64_t *end_pos*, vector<fml_utg_t> &*unitigs*)

void vg::SRPE::**assemble**(int64_t *node_id*, int64_t *offset*, int *window_size*)

Public Members

vector<string> vg::SRPE::**ref_names**

map<string, *PathIndex*> vg::SRPE::**pindexes**

vector<pair<int, int>> vg::SRPE::**intervals**

bool vg::SRPE::**overlapping_refs**

DepthMap vg::SRPE::**depth**

vg::*Filter* vg::SRPE::**ff**

map<string, *Alignment*> vg::SRPE::**name_to_aln**

map<string, string> vg::SRPE::**aln_to_mate**

vg::*VG* *vg::SRPE::**graph**

int vg::SRPE::**max_reads**
class *#include <ssw_aligner.hpp>*

Public Functions

vg::SSWAligner::**SSWAligner**(uint8_t *_match* = 1, uint8_t *_mismatch* = 4, uint8_t *_gap_open* = 6, uint8_t *_gap_extension* = 1)

vg::SSWAligner::**~SSWAligner**(void)

Alignment vg::SSWAligner::**align**(**const** string &*query*, **const** string &*ref*)

Alignment vg::SSWAligner::**ssw_to_vg**(**const** StripedSmithWaterman::Alignment &*ssw_aln*, **const** string &*query*, **const** string &*ref*)

void vg::SSWAligner::**PrintAlignment**(**const** StripedSmithWaterman::Alignment &*alignment*)

Public Members

uint8_t vg::SSWAligner::**match**

uint8_t vg::SSWAligner::**mismatch**

uint8_t vg::SSWAligner::**gap_open**

uint8_t vg::SSWAligner::**gap_extension**

struct *#include <suffix_tree.hpp>* A node of a suffix tree corresponding a substring of the string **Public Functions**

`vg::SuffixTree::STNode::`**`STNode`**(int64_t *first*, int64_t *last*)
Constructor.

`vg::SuffixTree::STNode::`**`~STNode`**()

int64_t `vg::SuffixTree::STNode::`**`length`**(int64_t *phase*)
The length of the the node during a phase of construction.

int64_t `vg::SuffixTree::STNode::`**`final_index`**(int64_t *phase*)
The last index contained on the this node during a phase of construction.

Public Members

unordered_map<char, *STNode* *> `vg::SuffixTree::STNode::`**`children`**
Edges down the tree.

int64_t `vg::SuffixTree::STNode::`**`first`**
First index of string on this node.

int64_t `vg::SuffixTree::STNode::`**`last`**
Last index of string on this node, inclusive (-1 indicates end sentinel during consruction)

struct *#include <pileup_augmenter.hpp>***Public Functions**

`vg::StrandSupport::`**`StrandSupport`**(int *f* = 0, int *r* = 0, double *q* = 0)

bool `vg::StrandSupport::`**`operator<`**(**const** *StrandSupport* &*other*) **const**

bool `vg::StrandSupport::`**`operator>=`**(**const** *StrandSupport* &*other*) **const**

bool `vg::StrandSupport::`**`operator==`**(**const** *StrandSupport* &*other*) **const**

StrandSupport `vg::StrandSupport::`**`operator-`**(**const** *StrandSupport* &*other*) **const**

StrandSupport &`vg::StrandSupport::`**`operator+=`**(**const** *StrandSupport* &*other*)

int `vg::StrandSupport::`**`total`**()

Public Members

int `vg::StrandSupport::`**`fs`**

int `vg::StrandSupport::`**`rs`**

double `vg::StrandSupport::`**`qual`**

template <**typename** K, **typename** V>
class *#include <hash_map.hpp>* Inherits from google::dense_hash_map< K, V >

Public Functions

vg::string_hash_map::**string_hash_map**()

template <**typename** K, **typename** V>

class *#include <hash_map_set.hpp>* Inherits from google::sparse_hash_map< K, V >

Public Functions

xg::string_hash_map::**string_hash_map**()

template <**typename** K>

class *#include <hash_map_set.hpp>* Inherits from google::sparse_hash_set< K >

Public Functions

xg::string_hash_set::**string_hash_set**()

class *#include <subcommand.hpp>* Represents a subcommand with a name, a description, and some functions. Registers itself on construction in a static registry, and provides static functions for enumerating through that registry. **Public Functions**

vg::subcommand::Subcommand::**Subcommand**(*std*::string *name*, *std*::string *description*, *CommandCategory category*, int *priority*, *std*::function<int) int, char **

> *main_function*Make and register a subcommand with the given name and description, in the given category, with the given priority (lower is better), which calls the given main function when invoked.

vg::subcommand::Subcommand::**Subcommand**(*std*::string *name*, *std*::string *description*, *CommandCategory category*, *std*::function<int) int, char **

> *main_function*Make and register a subcommand with the given name and description, in the given category, with worst priority, which calls the given main function when invoked.

vg::subcommand::Subcommand::**Subcommand**(*std*::string *name*, *std*::string *description*, *std*::function<int) int, char **

> *main_function*Make and register a subcommand with the given name and description, in the WIDGET category, with worst priority, which calls the given main function when invoked.

const *std*::string &vg::subcommand::Subcommand::**get_name**() **const**

Get the name of a subcommand.

const *std*::string &vg::subcommand::Subcommand::**get_description**() **const**

Get the description of a subcommand.

const *CommandCategory* &vg::subcommand::Subcommand::**get_category**() **const**

Get the category of a subcommand, which determines who might want to use it and why.

const int &vg::subcommand::Subcommand::**get_priority**() **const**

Get the priority level of a subcommand (lower is more important).

const int vg::subcommand::Subcommand::**operator()**(int *argc*, char ***argv*) **const**

Run the main function of a subcommand. Return the return code.

Public Static Functions

const *Subcommand* *vg::subcommand::Subcommand::**get**(int *argc*, char **argv*)
Get the appropriate subcommand to handle the given arguments, or nullptr if no matching subcommand is found.

void vg::subcommand::Subcommand::**for_each**(**const** *std*::function<void) **const** *Subcommand*& > &*lambda*Call the given lambda with each known subcommand, in order.

void vg::subcommand::Subcommand::**for_each**(*CommandCategory* *category*, **const** *std*::function<void) **const** *Subcommand*& > &*lambda*Call the given lambda with each known subcommand in the given category, in order.

Private Functions

const *std*::function<int(int, char **)> &vg::subcommand::Subcommand::**get_main**
constGet the main function of a subcommand.

Private Members

std::string vg::subcommand::Subcommand::**name**

std::string vg::subcommand::Subcommand::**description**

CommandCategory vg::subcommand::Subcommand::**category**

int vg::subcommand::Subcommand::**priority**

std::function<int(int, char **)> vg::subcommand::Subcommand::**main_function**

Private Static Functions

std::map<*std*::string, *Subcommand* *> &vg::subcommand::Subcommand::**get_registry**()
Since we can't rely on a static member field being constructed before any static code that creates actual subcommands gets run, we rely on keeping the registry in a static variable inside a static method, so it gets constructed on first use. Note that at shutdown some of the poinbters in the registry may be to already-destructed static objects.

struct A non-branching path of a *MultipathAlignment*. **Public Members**

Path vg::Subpath::**path**
describes node sequence and edits to the graph sequences

repeated<uint32> vg::Subpath::**next**
the indices of subpaths in the multipath alignment that are to the right of this path where right is in the direction of the end of the read sequence

int32 vg::Subpath::**score**
score of this subpath's alignment

class *#include <suffix_tree.hpp>* An implementation of a suffix tree with linear time and space complexity for construction. **Public Functions**

`vg::SuffixTree::SuffixTree`(string::const_iterator *begin*, string::const_iterator *end*)
Linear time constructor.

Note: string cannot have null characters, but this is not checked.

`vg::SuffixTree::~SuffixTree`()

size_t `vg::SuffixTree::longest_overlap`(**const** string &*str*)
Returns the length of the longest prefix of str that exactly matches a suffix of the string used to construct the suffix tree.

Note: string cannot have null characters, but this is not checked.

size_t `vg::SuffixTree::longest_overlap`(string::const_iterator *begin*, string::const_iterator *end*)

vector<size_t> `vg::SuffixTree::substring_locations`(**const** string &*str*)
Retuns a vector of all of the indices where a string occurs as a substring of the string used to construct the suffix tree. Indices are ordered arbitrarily.

Note: string cannot have null characters, but this is not checked.

vector<size_t> `vg::SuffixTree::substring_locations`(string::const_iterator *begin*, string::const_iterator *end*)

Public Members

const string::const_iterator `vg::SuffixTree::begin`
Beginning of string used to make tree.

const string::const_iterator `vg::SuffixTree::end`
End of string used to make tree.

Private Functions

char `vg::SuffixTree::get_char`(size_t *i*)
Returns a char of the string or null char at past-the-last index.

string `vg::SuffixTree::to_string`()

string `vg::SuffixTree::partial_tree_to_string`(int64_t *phase*)

string `vg::SuffixTree::label_string`(unordered_map<char, *STNode* *> **branch_point*)

string `vg::SuffixTree::node_string`(*STNode* **node*, int64_t *phase*)

string `vg::SuffixTree::active_point_string`(unordered_map<char, *STNode* *> **active_branch_point*, *STNode* **active_node*, int64_t *active_length*, int64_t *phase*)

string `vg::SuffixTree::suffix_links_string`(unordered_map<unordered_map<char, *STNode* *> *, unordered_map<char, *STNode* *> *> &*suffix_links*)

Private Members

list<*STNode*> `vg::SuffixTree::`**`nodes`**
All nodes in the tree (in a list to avoid difficulties with pointers and reallocations)

unordered_map<char, *STNode* *> `vg::SuffixTree::`**`root`**
The edges from the root node.
struct Aggregates information about the reads supporting an allele. **Public Members**

double `vg::Support::`**`quality`**
The overall quality of all the support, as -10 * log10(P(all support is wrong))

double `vg::Support::`**`forward`**
The number of supporting reads on the forward strand (which may be fractional)

double `vg::Support::`**`reverse`**
The number of supporting reads on the reverse strand (which may be fractional)

double `vg::Support::`**`left`**
TODO: what is this?

double `vg::Support::`**`right`**
TODO: What is this?
struct *#include <genotypekit.hpp>* Augmented *Graph* that holds some *Support* annotation data specific to vg call. Inherits from *vg::AugmentedGraph* **Public Functions**

bool `vg::SupportAugmentedGraph::`**`has_supports`**`()`
Return true if we have support information, and false otherwise.

Support `vg::SupportAugmentedGraph::`**`get_support`**`(`*Node* *`node`*`)`
Get the *Support* for a given *Node*, or 0 if it has no recorded support.

Support `vg::SupportAugmentedGraph::`**`get_support`**`(`*Edge* *`edge`*`)`
Get the *Support* for a given *Edge*, or 0 if it has no recorded support.

void `vg::SupportAugmentedGraph::`**`clear`**`()`
Clear the contents.

void `vg::SupportAugmentedGraph::`**`load_supports`**`(`istream *&`*`in_file`*`)`
Read the supports from protobuf.

void `vg::SupportAugmentedGraph::`**`write_supports`**`(`ostream *&`*`out_file`*`)`
Write the supports to protobuf

Public Members

map<*Node* *, *Support*> `vg::SupportAugmentedGraph::`**`node_supports`**

map<*Edge* *, *Support*> `vg::SupportAugmentedGraph::`**`edge_supports`**
class *#include <support_caller.hpp>* *SupportCaller*: take an augmented graph from a Caller and produce actual calls in a VCF. Inherits from *vg::Configurable* **Public Functions**

`vg::SupportCaller::`**`SupportCaller`**`()`
Set up to call with default parameters.

void vg::SupportCaller::**call**(*SupportAugmentedGraph* &*augmented*, string *pileup_filename* = "")

Produce calls for the given annotated augmented graph. If a pileup_filename is provided, the pileup is loaded again and used to add comments describing variants

tuple<vector<*Support*>, vector<size_t>> vg::SupportCaller::**get_traversal_supports_and_sizes**(*SupportAugmented* &*augmented*, *SnarlManager* &*snarl_manager*, **const** *Snarl* &*site*, **const** vector<*SnarlTraversal* &*traversals*, **const** *SnarlTraversal* **minus_traversal* = NULL)

Get the support and size for each traversal in a list. Discount support of minus_traversal if it's specified. Use average_support_switch_threshold and use_average_support to decide whether to return min or avg supports

Get the support for each traversal in a list, using average_support_switch_threshold to decide if we use the minimum or average

vector<*SnarlTraversal*> vg::SupportCaller::**find_best_traversals**(*SupportAugmentedGraph* &*augmented*, *SnarlManager* &*snarl_manager*, *TraversalFinder* **finder*, **const** *Snarl* &*site*, **const** *Support* &*baseline_support*, size_t *copy_budget*, function<void()> **const** *Locus*&, **const** *Snarl* * > *emit_locus*For the given snarl, find the reference traversal, the best traversal, and the second-best traversal, recursively, if any exist. These traversals will be fully filled in with nodes.

Only snarls which are ultrabubbles can be called.

Expects the given baseline support for a diploid call.

Will not return more than 1 + copy_budget SnarlTraversals, and will return less if some copies are called as having the same traversal.

Does not deduplicate agains the ref traversal; it may be the same as the best or second-best.

Uses the given copy number allowance, and emits a *Locus* for this *Snarl* and any child Snarls.

If no path through the *Snarl* can be found, emits no *Locus* and returns no SnarlTraversals.

bool `vg::SupportCaller::`**`is_reference`**(**const** *SnarlTraversal* *&trav*, *AugmentedGraph* *&augmented*)
Decide if the given *SnarlTraversal* is included in the original base graph (true), or if it represents a novel variant (false).

Looks at the nodes in the traversal, and sees if their calls are CALL_REFERENCE or not.

Handles single-edge traversals.

bool `vg::SupportCaller::`**`is_reference`**(**const** *Path* *&path*, *AugmentedGraph* *&augmented*)
Decide if the given *Path* is included in the original base graph (true) or if it represents a novel variant (false).

Looks at the nodes, and sees if their calls are CALL_REFERENCE or not.

The path can't be empty; it has to be anchored to something (probably the start and end of the snarl it came from).

map<string, *SupportCaller*::*PrimaryPath*>::iterator `vg::SupportCaller::`**`find_path`**(**const** *Snarl* *&site*, map<string, *PrimaryPath*> *&primary_paths*)
Find the primary path, if any, that the given site is threaded onto.

TODO: can only work by brute-force search.

Public Members

function<double (**const** *Support*&) > `vg::SupportCaller::`**`support_val`**
Get the amount of support. Can use this function to toggle between unweighted (total from genotypekit) and quality-weighted (support_quality below) in one place.

Option<bool> `vg::SupportCaller::`**`convert_to_vcf`**
Should we output in VCF (true) or Protobuf *Locus* (false) format?

size_t `vg::SupportCaller::`**`locus_buffer_size`**
How big should our output buffer be?

Option<vector<string>> `vg::SupportCaller::`**`ref_path_names`**
What are the names of the reference paths, if any, in the graph?

Option<vector<string>> `vg::SupportCaller::`**`contig_name_overrides`**
What name should we give each contig in the VCF file? Autodetected from path names if empty or too short.

Option<vector<size_t>> `vg::SupportCaller::`**`length_overrides`**
What should the total sequence length reported in the VCF header be for each contig? Autodetected from path lengths if empty or too short.

Option<string> vg::SupportCaller::**sample_name**
What name should we use for the sample in the VCF file?

Option<int64_t> vg::SupportCaller::**variant_offset**
How far should we offset positions of variants?

Option<int64_t> vg::SupportCaller::**max_search_depth**
How many nodes should we be willing to look at on our path back to the primary path? Keep in mind we need to look at all valid paths (and all combinations thereof) until we find a valid pair.

Option<int64_t> vg::SupportCaller::**max_search_width**
How many search states should we allow on the DFS stack when searching for traversals?

Option<double> vg::SupportCaller::**min_fraction_for_call**
What fraction of average coverage should be the minimum to call a variant (or a single copy)? Default to 0 because vg call is still applying depth thresholding

Option<double> vg::SupportCaller::**max_het_bias**
What fraction of the reads supporting an alt are we willing to discount? At 2, if twice the reads support one allele as the other, we'll call homozygous instead of heterozygous. At infinity, every call will be heterozygous if even one read supports each allele.

Option<double> vg::SupportCaller::**max_ref_het_bias**
Like above, but applied to ref / alt ratio (instead of alt / ref)

Option<double> vg::SupportCaller::**max_indel_het_bias**
Like the max het bias, but applies to novel indels.

Option<double> vg::SupportCaller::**max_indel_ma_bias**
Like the max het bias, but applies to multiallelic indels.

Option<size_t> vg::SupportCaller::**min_total_support_for_call**
What's the minimum integer number of reads that must support a call? We don't necessarily want to call a SNP as het because we have a single

Option<size_t> vg::SupportCaller::**ref_bin_size**
Bin size used for counting coverage along the reference path. The bin coverage is used for computing the probability of an allele of a certain depth

Option<double> vg::SupportCaller::**expected_coverage**
On some graphs, we can't get the coverage because it's split over parallel paths. Allow overriding here

Option<bool> vg::SupportCaller::**use_average_support**
Should we use average support instead of minimum support for our calculations?

Option<size_t> vg::SupportCaller::**average_support_switch_threshold**
Max traversal length threshold at which we switch from minimum support to average support (so we don't use average support on pairs of adjacent errors and miscall them, but we do use it on long runs of reference inside a deletion where the min support might not be representative.

size_t vg::SupportCaller::**max_bubble_paths**
What's the maximum number of bubble path combinations we can explore while finding one with maximum support?

Option<size_t> vg::SupportCaller::**min_mad_for_filter**
what's the minimum ref or alt allele depth to give a PASS in the filter column? Also used as a min actual support for a second-best allele call

Option<size_t> vg::SupportCaller::**max_dp_for_filter**
what's the maximum total depth to give a PASS in the filter column

Option<double> vg::SupportCaller::**max_dp_multiple_for_filter**
what's the maximum total depth to give a PASS in the filter column, as a multiple of the global baseline coverage?

Option<double> vg::SupportCaller::**max_local_dp_multiple_for_filter**
what's the maximum total depth to give a PASS in the filter column, as a multiple of the local baseline coverage?

Option<double> vg::SupportCaller::**min_ad_log_likelihood_for_filter**
what's the min log likelihood for allele depth assignments to PASS?

Option<bool> vg::SupportCaller::**write_trivial_calls**

Option<bool> vg::SupportCaller::**call_other_by_coverage**
Should we call on nodes/edges outside of snarls by coverage (true), or just assert that primary path things exist and off-path things don't (false)?

Option<bool> vg::SupportCaller::**use_support_count**
Use total support count (true) instead of total support quality (false) when choosing top alleles and deciding gentypes based on the biases.

Option<string> vg::SupportCaller::**support_file_name**
Path of supports file generated from the *PileupAugmenter* (via vg augment)

bool vg::SupportCaller::**verbose**
print warnings etc. to stderr

Public Static Functions

static double vg::SupportCaller::**support_quality**(**const** *Support* &*support*)
struct *#include* <*genome_state.hpp*> Inherits from *vg::GenomeStateCommand* **Public Functions**

GenomeStateCommand *vg::SwapHaplotypesCommand::**execute**(*GenomeState* &*state*) **const**
Execute this command on the given state and return the reverse command. Generally ends up calling a command-type-specific method on the *GenomeState* that does the actual work.

virtual vg::SwapHaplotypesCommand::**~SwapHaplotypesCommand**()

Public Members

pair<**const** *Snarl* *, **const** *Snarl* *> vg::SwapHaplotypesCommand::**telomere_pair**
Work on the given pair of telomeres.

pair<size_t, size_t> vg::SwapHaplotypesCommand::**to_swap**
Swap the haplotypes at the given ranks.
struct *#include* <*xg.hpp*>**Public Functions**

bool xg::XG::ThreadMapping::**operator<**(**const** *ThreadMapping* &*other*) **const**
We need comparison for deduplication in sets and canonically orienting threads.

Public Members

int64_t xg::XG::ThreadMapping::**node_id**

bool xg::XG::ThreadMapping::**is_reverse**
struct *#include <xg.hpp>* Represents the search state for the graph PBWT, so that you can continue a search with more of a thread, or backtrack. By default, represents an un-started search (with no first visited side) that can be extended to the whole collection of visits to a side. **Public Functions**

int64_t xg::XG::ThreadSearchState::**count**()

bool xg::XG::ThreadSearchState::**is_empty**()

Public Members

int64_t xg::XG::ThreadSearchState::**current_side**

int64_t xg::XG::ThreadSearchState::**range_start**

int64_t xg::XG::ThreadSearchState::**range_end**
struct Translations map from one graph to another. A collection of these provides a covering mapping between a from and to graph. If each "from" path through the base graph corresponds to a "to" path in an updated graph, then we can use these translations to project positions, mappings, and paths in the new graph into the old one using the *Translator* interface. **Public Members**

Path vg::Translation::**from**

Path vg::Translation::**to**
class *#include <translator.hpp>* Class to map paths into a base graph found via a set of Translations **Public Functions**

vg::Translator::**Translator**(void)

vg::Translator::**Translator**(istream &*in*)

vg::Translator::**Translator**(**const** vector<*Translation*> &*trans*)

void vg::Translator::**load**(**const** vector<*Translation*> &*trans*)

void vg::Translator::**build_position_table**(void)

Translation vg::Translator::**get_translation**(**const** *Position* &*position*)

bool vg::Translator::**has_translation**(**const** *Position* &*position*, bool *ignore_strand* = true)

Position vg::Translator::**translate**(**const** *Position* &*position*)

Position vg::Translator::**translate**(**const** *Position* &*position*, **const** *Translation* &*translation*)

Edge vg::Translator::**translate**(**const** *Edge* &*edge*)

Mapping vg::Translator::**translate**(**const** *Mapping* &*mapping*)

Path vg::Translator::**translate**(**const** *Path* &*path*)

Alignment vg::Translator::**translate**(**const** *Alignment* &*aln*)

Locus vg::Translator::**translate**(**const** *Locus* &*locus*)

Translation vg::Translator::**overlay**(**const** *Translation* &*trans*)

Public Members

vector<*Translation*> vg::Translator::**translations**

map<*pos_t*, *Translation* *> vg::Translator::**pos_to_trans**
class *#include <genotypekit.hpp>* Represents a strategy for finding traversals of (nested) sites. Polymorphic base class/interface. Subclassed by *vg::ExhaustiveTraversalFinder*, *vg::NestedTraversalFinder*, *vg::PathBasedTraversalFinder*, *vg::ReadRestrictedTraversalFinder*, *vg::RepresentativeTraversalFinder*, *vg::TrivialTraversalFinder* **Public Functions**

virtual vg::TraversalFinder::**~TraversalFinder**()

virtual vector<*SnarlTraversal*> vg::TraversalFinder::**find_traversals**(**const** *Snarl* &*site*) = 0
class *#include <genotypekit.hpp>* Represents a strategy for calculating Supports for SnarlTraversals. Polymorphic base class/interface. Subclassed by *vg::SimpleTraversalSupportCalculator* **Public Functions**

virtual vg::TraversalSupportCalculator::**~TraversalSupportCalculator**()

virtual vector<*Support*> vg::TraversalSupportCalculator::**calculate_supports**(**const** *Snarl* &*site*, **const** vector<*SnarlTraversal*> &*traversals*, **const** vector<*Alignment* *> &*reads*, **const** vector<vector<bool>> &*consistencies*) **const** = 0Return Supports for all the SnarlTraversals, given the reads and their consistency flags.
template <**typename** T>
struct *#include <utility.hpp>*

Public Types

typedef

Public Functions

vg::Tree::**Tree**(*Node* *r = 0)

vg::Tree::**~Tree**()

void vg::Tree::**for_each_preorder**(function<void) *Node* *
> *lambda*

void vg::Tree::**for_each_postorder**(function<void) *Node* *
> *lambda*

Public Members

Node *vg::Tree::**root**

template <**typename** T>
struct *#include <utility.hpp>*

Public Functions

vg::TreeNode::**TreeNode**()

vg::TreeNode::**~TreeNode**()

void vg::TreeNode::**for_each_preorder**(function<void) *TreeNode*<T> *
> *lambda*

void vg::TreeNode::**for_each_postorder**(function<void) *TreeNode*<T> *
> *lambda*

Public Members

T vg::TreeNode::**v**

vector<*TreeNode*<T> *> vg::TreeNode::**children**

TreeNode<T> *vg::TreeNode::**parent**
class *#include <genotypekit.hpp>* This traversal finder finds one or more traversals through leaf sites with no children. It uses a depth-first search. It doesn't work on non-leaf sites, and is not guaranteed to find all traversals. Only works on ultrabubbles. Inherits from *vg::TraversalFinder* **Public Functions**

vg::TrivialTraversalFinder::**TrivialTraversalFinder**(*VG* &*graph*)

virtual vg::TrivialTraversalFinder::**~TrivialTraversalFinder**()

vector<*SnarlTraversal*> vg::TrivialTraversalFinder::**find_traversals**(**const** *Snarl* &*site*)
Find at least one traversal of the site by depth first search, if any exist. Only works on sites with no children.

Private Members

VG &vg::TrivialTraversalFinder::**graph**
struct *#include <utility.hpp>*

Public Functions

vg::UnionFind::UFNode::**UFNode**(size_t *index*)

vg::UnionFind::UFNode::**~UFNode**()

Public Members

size_t vg::UnionFind::UFNode::**rank**

size_t vg::UnionFind::UFNode::**size**

size_t vg::UnionFind::UFNode::**head**

unordered_set<size_t> vg::UnionFind::UFNode::**children**
class *#include <utility.hpp>* A custom Union-Find data structure that supports merging a set of indices in disjoint sets in amortized nearly linear time. This implementation also supports querying the size of the group containing an index in constant time and querying the members of the group containing an index in linear time in the size of the group.

Public Functions

vg::UnionFind::**UnionFind**(size_t *size*)
Construct *UnionFind* for this many indices.

vg::UnionFind::**~UnionFind**()
Destructor.

size_t vg::UnionFind::**size**()
Returns the number of indices in the *UnionFind*.

size_t vg::UnionFind::**find_group**(size_t *i*)
Returns the group ID that index i belongs to (can change after calling union)

void vg::UnionFind::**union_groups**(size_t *i*, size_t *j*)
Merges the group containing index i with the group containing index j.

size_t vg::UnionFind::**group_size**(size_t *i*)
Returns the size of the group containing index i.

vector<size_t> vg::UnionFind::**group**(size_t *i*)
Returns a vector of the indices in the same group as index i.

vector<vector<size_t>> vg::UnionFind::**all_groups**()
Returns all of the groups, each in a separate vector.

string vg::UnionFind::**current_state**()
A string representation of the current state for debugging.

Private Members

vector<*UFNode*> vg::UnionFind::**uf_nodes**
class *#include <variant_adder.hpp>* A tool class for adding variants to a *VG* graph. Integrated *NameMapper* provides name translation for the VCF contigs. Inherits from *vg::NameMapper*, *vg::Progressive* **Public Functions**

vg::VariantAdder::**VariantAdder**(*VG* &*graph*)
Make a new *VariantAdder* to add variants to the given graph. Modifies the graph in place.

void vg::VariantAdder::**add_variants**(vcflib::VariantCallFile **vcf*)
Add in the variants from the given non-null VCF file. The file must be freshly opened. The variants in the file must be sorted.

May be called from multiple threads. Synchronizes internally on the graph.

Alignment vg::VariantAdder::**smart_align**(*vg*::*VG* &*graph*, pair<*NodeSide*, *NodeSide*> *endpoints*, **const** string &*to_align*, size_t *max_span*)
Align the given string to the given graph, wetween the given endpoints, using the most appropriate alignment method, depending on the relative sizes involved and whether a good alignment exists. max_span gives the maximum length in the graph that we expect our string to possibly align over (for cases of large deletions, where we might want to follow a long path in the graph).

The endpoints have to be heads/tails of the graph.

Treats N/N substitutions as matches.

TODO: now that we have a smart aligner that can synthesize deletions without finding them with the banded global aligner, do we need max_span anymore?

Mostly exposed for testability.

void vg::VariantAdder::**align_ns**(*vg*::*VG* &*graph*, *Alignment* &*aln*)
Turn any N/N substitutions in the given alignment against the given graph into matches. Modifies the alignment in place.

Public Members

size_t vg::VariantAdder::**variant_range**
How wide of a range in bases should we look for nearby variants in?

size_t vg::VariantAdder::**flank_range**
How much additional context should we try and add outside the radius of our group of variants we actually find?

bool vg::VariantAdder::**ignore_missing_contigs**
Should we accept and ignore VCF contigs that we can't find in the graph?

size_t vg::VariantAdder::**max_context_radius**
What's the max radius on a variant we can have in order to use that variant as context for another main variant?

size_t vg::VariantAdder::**whole_alignment_cutoff**
What's the cut-off for the graph's size or the alt's size in bp under which we can just use permissive banding and large band padding? If either is larger than this, we use the pinned-alignment-based do-each-end-and-splice mode.

size_t vg::VariantAdder::**large_alignment_band_padding**
When we're above that cutoff, what amount of band padding can we use looking for an existing version of our sequence?

double vg::VariantAdder::**min_score_factor**
When we're doing a restricted band padding alignment, how good does it have to be, as a fraction of the perfect match score for the whole context, in order to use it?

size_t vg::VariantAdder::**pinned_tail_size**
If the restricted band alignment doesn't find anything, we resort to pinned alignments from the ends and cutting and pasting together. How big should each pinned tail be?

Aligner vg::VariantAdder::**aligner**
We use this *Aligner* to hold the scoring parameters. It may be accessed by multiple threads at once.

size_t vg::VariantAdder::**edge_max**
Sometimes, we have to make Mappers, for graphs too big to safely use our global banded aligner on. If we do that, what max edge crossing limit should we use for simplification?

size_t vg::VariantAdder::**kmer_size**
What base kmer size should we use?

size_t vg::VariantAdder::**doubling_steps**
What number of doubling steps should we use?

size_t vg::VariantAdder::**subgraph_prune**
If nonzero, prune short subgraphs smaller than this before GCSA2-indexing.

size_t vg::VariantAdder::**thin_alignment_cutoff**

size_t vg::VariantAdder::**mapper_alignment_cutoff**

bool vg::VariantAdder::**skip_structural_duplications**
We have code to skip large structural duplications, because aligners won't be able to distinguish the copies. TODO: we want to actually make them into cycles.

bool vg::VariantAdder::**print_updates**
Should we print out periodic updates about the variants we have processed?

Protected Functions

set<vector<int>> vg::VariantAdder::**get_unique_haplotypes**(**const** vector<vcflib::Variant *> &*variants*, *WindowedVcfBuffer* *cache = nullptr) **const**
Get all the unique combinations of variant alts represented by actual haplotypes. Arbitrarily phases unphased variants.

Can (and should) take a *WindowedVcfBuffer* that owns the variants, and from which cached pre-parsed genotypes can be extracted.

Returns a set of vectors or one number per variant, giving the alt number (starting with 0 for reference) that appears on the haplotype.

TODO: ought to just take a collection of pre-barsed genotypes, but in an efficient way (a vector of pointers to vectors of sample genotypes?)

string vg::VariantAdder::**haplotype_to_string**(**const** vector<int> &*haplotype*, **const** vector<vcflib::Variant *> &*variants*)
Convert a haplotype on a list of variants into a string. The string will run from the start of the first variant through the end of the last variant.

Can't be const because it relies on non-const operations on the synchronizer.

vector<vcflib::Variant *> vg::VariantAdder::**filter_local_variants**(**const** vector<vcflib::Variant *> &*before*, vcflib::Variant **variant*, **const** vector<vcflib::Variant *> &*after*) **const**
Glom all the given variants into one vector, throwing out variants from the before and after vectors that are too big to be in a context.

Protected Attributes

VG &vg::VariantAdder::**graph**
The graph we are modifying.

GraphSynchronizer vg::VariantAdder::**sync**
We keep a *GraphSynchronizer* so we can have multiple threads working on different parts of the same graph.

set<string> vg::VariantAdder::**path_names**
We cache the set of valid path names, so we can detect/skip missing ones without locking the graph.

Protected Static Functions

size_t vg::VariantAdder::**get_radius**(**const** vcflib::Variant &*variant*)
Get the radius of the variant around its center: the amount of sequence that needs to be pulled out to make sure you have the ref and all the alts, if they exist. This is just going to be twice the longest of the ref and the alts.

size_t vg::VariantAdder::**get_center**(**const** vcflib::Variant &*variant*)
Get the center position of the given variant.

pair<size_t, size_t> vg::VariantAdder::**get_center_and_radius**(**const** vector<vcflib::Variant *> &*variants*)
Get the center and radius around that center needed to extract everything that might be involved in a group of variants.

class *#include <vcf_buffer.hpp>* Provides a one-variant look-ahead buffer on a vcflib::VariantFile. Lets construction functions peek and see if they want the next variant, or lets them ignore it for the next construction function for a different contig to handle. Ought not to be copied. Handles conversion from 1-based vcflib coordinates to 0-based vg coordinates. **Public Functions**

vcflib::Variant *vg::VcfBuffer::**get**()
Return a pointer to the buffered variant, or null if no variant is buffered. Pointer is invalidated when the buffer is handled. The variant will have a 0-based start coordinate.

Although the variant is not const, it may not be moved out of or modified in ways that confuse vcflib.

void vg::VcfBuffer::**handle_buffer**()
To be called when the buffer is filled. Marks the buffered variant as handled, discarding it, and allowing another to be read.

void vg::VcfBuffer::**fill_buffer**()
Can be called when the buffer is filled or empty. If there is no variant in the buffer, tries to load a variant into the buffer, if one can be obtained from the file.

bool vg::VcfBuffer::**has_tabix**() **const**
This returns true if we have a tabix index, and false otherwise. If this is false, set_region may be called, but will do nothing and return false.

bool vg::VcfBuffer::**set_region**(**const** string &*contig*, int64_t *start* = -1, int64_t *end* = -1)
This tries to set the region on the underlying vcflib VariantCallFile to the given contig and region, if specified. Coordinates coming in should be 0-based, and will be converted to 1-based internally.

Returns true if the region was successfully set, and false otherwise (for example, if there is not tabix index, or if the given region is not part of this VCF. Note that if there is a tabix index, and set_region returns false, the position in the VCF file is undefined until the next successful set_region call.

If either of start and end are specified, then both of start and end must be specified.

Discards any variants previously in the buffer.

vg::VcfBuffer::**VcfBuffer**(vcflib::VariantCallFile **file* = nullptr)
Make a new *VcfBuffer* buffering the file at the given pointer (which must outlive the buffer, but which may be null).

Protected Attributes

bool vg::VcfBuffer::**has_buffer**

bool vg::VcfBuffer::**safe_to_get**

vcflib::Variant vg::VcfBuffer::**buffer**

vcflib::VariantCallFile ***const** vg::VcfBuffer::**file**

Private Functions

vg::VcfBuffer::**VcfBuffer**(**const** *VcfBuffer* &*other*)

VcfBuffer &vg::VcfBuffer::**operator=**(**const** *VcfBuffer* &*other*)
class *#include <genotypekit.hpp>* Represents a strategy for converting *Locus* objects to VCF records. Polymorphic base class/interface. **Public Functions**

virtual vg::VcfRecordConverter::**~VcfRecordConverter**()

virtual vcflib::Variant vg::VcfRecordConverter::**convert**(**const** *Locus* &*locus*)
= 0

class *#include <genotypekit.hpp>* Represents a filter that passes or rejects VCF records according to some criteria. Polymorphic base class/interface. **Public Functions**

virtual `vg::VcfRecordFilter::~VcfRecordFilter()`

virtual bool `vg::VcfRecordFilter::accept_record`(**const** vcflib::Variant &*variant*) = 0Returns true if we should keep the given VCF record, and false otherwise.

class *#include <vectorizer.hpp>***Public Functions**

`Vectorizer::Vectorizer`(*xg*::*XG* **x*)

`Vectorizer::~Vectorizer()`

void `Vectorizer::add_bv`(bit_vector *v*)

void `Vectorizer::add_name`(string *n*)

void `Vectorizer::emit`(ostream &*out*, bool *r_format*, bool *annotate*)

bit_vector `Vectorizer::alignment_to_onehot`(Alignment *a*)

vector<int> `Vectorizer::alignment_to_a_hot`(Alignment *a*)

vector<double> `Vectorizer::alignment_to_custom_score`(Alignment *a*, *std*::function<double) Alignment > *lambda*

vector<double> `Vectorizer::alignment_to_identity_hot`(Alignment *a*)

string `Vectorizer::output_wabbit_map`()

template <typename T>
string `Vectorizer::format`(T *v*)

template <typename T>
string `Vectorizer::wabbitize`(string *name*, T *v*)

Private Members

xg::*XG* **Vectorizer::my_xg**

vector<bit_vector> `Vectorizer::my_vectors`

vector<string> `Vectorizer::my_names`

bool `Vectorizer::output_tabbed`

bool `Vectorizer::output_names`

unordered_map<string, int> `Vectorizer::wabbit_map`

class *#include <vg.hpp>* Represents a variation graph. Graphs consist of nodes, connected by edges. Graphs are bidirected and may be cyclic. Nodes carry forward-oriented sequences. Edges are directed, with a "from" and to" node, and are generally used to connect the end of the "from" node to the start of the "to" node. However, edges can connect to either the start or end of either node. Inherits from *vg::Progressive*, *vg::MutableHandleGraph* **Public Functions**

handle_t `vg::VG::get_handle`(**const** *id_t* &*node_id*, bool *is_reverse*) **const**
Look up the handle for the node with the given ID in the given orientation.

id_t vg::VG::**get_id**(**const** *handle_t &handle*) **const**
Get the ID from a handle.

bool vg::VG::**get_is_reverse**(**const** *handle_t &handle*) **const**
Get the orientation of a handle.

handle_t vg::VG::**flip**(**const** *handle_t &handle*) **const**
Invert the orientation of a handle (potentially without getting its ID)

size_t vg::VG::**get_length**(**const** *handle_t &handle*) **const**
Get the length of a node.

string vg::VG::**get_sequence**(**const** *handle_t &handle*) **const**
Get the sequence of a node, presented in the handle's local forward orientation.

bool vg::VG::**follow_edges**(**const** *handle_t &handle*, bool *go_left*, **const** function<bool(*handle_t&*)> *&iteratee*) **const**
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.

void vg::VG::**for_each_handle**(**const** function<bool(*handle_t&*)> *&iteratee*) **const**
Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false.

size_t vg::VG::**node_size**() **const**
Return the number of nodes in the graph.

handle_t vg::VG::**create_handle**(**const** string *&sequence*)
Create a new node with the given sequence and return the handle.

void vg::VG::**destroy_handle**(**const** *handle_t &handle*)
Remove the node belonging to the given handle and all of its edges.

void vg::VG::**create_edge**(**const** *handle_t &left*, **const** *handle_t &right*)
Create an edge connecting the given handles in the given order and orientations.

void vg::VG::**destroy_edge**(**const** *handle_t &left*, **const** *handle_t &right*)
Remove the edge connecting the given handles in the given order and orientations.

void vg::VG::**swap_handles**(**const** *handle_t &a*, **const** *handle_t &b*)
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.

handle_t vg::VG::**apply_orientation**(**const** *handle_t &handle*)
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.

vector<*handle_t*> vg::VG::**divide_handle**(**const** *handle_t &handle*, **const** vector<size_t> *&offsets*)
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.

void vg::VG::**set_edge**(*Edge* *edge)
Set the edge indexes through this function. Picks up the sides being connected by the edge automatically, and silently drops the edge if they are already connected.

void vg::VG::**print_edges**(void)

vector<pair<*id_t*, bool>> &vg::VG::**edges_start**(*Node* *node)
Get nodes and backward flags following edges that attach to this node's start.

vector<pair<*id_t*, bool>> &vg::VG::**edges_start**(*id_t id*)
Get nodes and backward flags following edges that attach to this node's start.

vector<pair<*id_t*, bool>> &vg::VG::**edges_end**(*Node* *node)
Get nodes and backward flags following edges that attach to this node's end.

vector<pair<*id_t*, bool>> &vg::VG::**edges_end**(*id_t id*)
Get nodes and backward flags following edges that attach to this node's end.

size_t vg::VG::**size**(void)
Number of nodes.

size_t vg::VG::**length**(void)
Total sequence length.

vg::VG::**VG**(void)
Default constructor.

vg::VG::**VG**(istream &*in*, bool *showp* = false, bool *warn_on_duplicates* = true)
Construct from protobufs.

vg::VG::**VG**(function<bool() *Graph*&
> &*get_next_graph*, bool *showp* = false, bool *warn_on_duplicates* = trueConstruct from an arbitrary source of *Graph* protobuf messages (which populates the given *Graph* and returns a flag for whether it's valid).

vg::VG::**VG**(set<*Node* *> &*nodes*, set<*Edge* *> &*edges*)
Construct from sets of nodes and edges. For example, from a subgraph of another graph.

map<*id_t*, vcflib::Variant> vg::VG::**get_node_id_to_variant**(vcflib::VariantCallFile *vfile*)
Takes in a VCF file and returns a map [node] = vcflib::variant. Unfortunately this is specific to a given graph and VCF.

It will need to throw warnings if the node or variant is not in the graph.

This is useful for VCF masking:

```
if map.find(node) then mask variant
```

It's also useful for calling known variants

```
for m in alignment.mappings:
   node = m.Pos.nodeID
   if node in node_to_vcf:
       return (alignment supports variant)
```

It would be nice if this also supported edges (e.g. for inversions/transversions/breakpoints?).

void vg::VG::**dice_nodes**(int *max_node_size*)
Chop up the nodes.

void vg::VG::**unchop**(void)
Does the reverse combines nodes by removing edges where doing so has no effect on the graph labels.

set<list<*NodeTraversal*>> vg::VG::**simple_components**(int *min_size* = 1)
Get the set of components that could be merged into single nodes without changing the path space of the graph. Emits oriented traversals of nodes, in the order and orientation in which they are to be merged.

set<list<*NodeTraversal*>> vg::VG::**simple_multinode_components**(void)
Get the simple components of multiple nodes.

set<set<*id_t*>> vg::VG::**strongly_connected_components**(void)
Get the strongly connected components of the graph.

set<set<*id_t*>> vg::VG::**multinode_strongly_connected_components**(void)
Get only multi-node strongly connected components.

bool vg::VG::**is_acyclic**(void)
Returns true if the graph does not contain cycles.

bool vg::VG::**is_directed_acyclic**(void)
Returns true if the graph does not contain a directed cycle (but it may contain a reversing cycle)

bool vg::VG::**is_single_stranded**(void)
Return true if there are no reversing edges in the graph.

void vg::VG::**keep_multinode_strongly_connected_components**(void)
Remove all elements which are not in a strongly connected component.

bool vg::VG::**is_self_looping**(*Node* **node*)
Does the specified node have any self-loops?

set<list<*NodeTraversal*>> vg::VG::**elementary_cycles**(void)
Get simple cycles following Johnson's elementary cycles algorithm.

Node *vg::VG::**concat_nodes**(**const** list<*NodeTraversal*> &*nodes*)
Concatenates the nodes into a new node with the same external linkage as the provided component. After calling this, paths will be invalid until *Paths::compact_ranks()* is called.

Node *vg::VG::**merge_nodes**(**const** list<*Node* *> &*nodes*)
Merge the nodes into a single node, preserving external linkages. Use the orientation of the first node as the basis.

void vg::VG::**normalize**(int *max_iter* = 1, bool *debug* = false)
Use unchop and sibling merging to simplify the graph into a normalized form.

void vg::VG::**bluntify**(void)
Remove redundant overlaps.

VG vg::VG::**dagify**(uint32_t *expand_scc_steps*, unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*, size_t *target_min_walk_length* = 0, size_t *component_length_max* = 0)
Turn the graph into a dag by copying strongly connected components expand_scc_steps times and translating the edges in the component to flow through the copies in one direction.

VG vg::VG::**backtracking_unroll**(uint32_t *max_length*, uint32_t *max_depth*, unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*)
Generate a new graph that unrolls the current one using backtracking. Caution: exponential in branching.

VG vg::VG::**unfold**(uint32_t *max_length*, unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*)
Ensure that all traversals up to max_length are represented as a path on one strand or the other without taking an inverting edge. All inverting edges are converted to non-inverting edges to reverse complement nodes. If no inverting edges are present, the strandedness of all nodes is the same as the input graph. If inverting edges are present, node strandedness is arbitrary.

VG vg::VG::**split_strands**(unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*)
Create reverse complement nodes and edges for the entire graph. Doubles the size. Converts all inverting edges into non-inverting edges.

VG vg::VG::**reverse_complement_graph**(unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*)
Create the reverse complemented graph with topology preserved. Record translation in provided map.

void vg::VG::**identity_translation**(unordered_map<*id_t*, pair<*id_t*, bool>> &*node_translation*)
Record the translation of this graph into itself in the provided map.

unordered_map<*id_t*, pair<*id_t*, bool>> vg::VG::**overlay_node_translations**(**const** unordered_map<*id_t*, pair<*id_t*, bool>> &*over*, **const** unordered_map<*id_t*, pair<*id_t*, bool>> &*under*)
Assume two node translations, the over is based on the under; merge them.

vector<*Edge*> vg::VG::**break_cycles**(void)
Use our topological sort to quickly break cycles in the graph, return the edges which are removed. Very non-optimal, but fast.

void vg::VG::**remove_non_path**(void)
Remove pieces of the graph which are not part of any path.

void vg::VG::**remove_path**(void)
Remove pieces of the graph which are part of some path.

set<*Edge* *> vg::VG::**get_path_edges**(void)
Get all of the edges that are on any path.

void vg::VG::**flip_doubly_reversed_edges**(void)
Convert edges that are both from_start and to_end to "regular" ones from end to start.

void vg::VG::**from_gfa**(istream &*in*, bool *showp* = false)
Build a graph from a GFA stream.

void vg::VG::**from_turtle**(string *filename*, string *baseuri*, bool *showp* = false)
Build a graph from a Turtle stream.

vg::VG::**~VG**(void)
Destructor.

vg::VG::**VG**(**const** *VG* &*other*)
Copy constructor.

vg::VG::**VG**(*VG* &&*other*)
Move constructor.

VG &vg::VG::**operator=**(**const** *VG* &*other*)
Copy assignment operator.

VG &vg::VG::**operator=**(*VG* &&*other*)
Move assignment operator.

void vg::VG::**build_indexes**(void)

void vg::VG::**build_node_indexes**(void)

void vg::VG::**build_edge_indexes**(void)

void vg::VG::**build_indexes_no_init_size**(void)

void vg::VG::**build_node_indexes_no_init_size**(void)

void vg::VG::**build_edge_indexes_no_init_size**(void)

void vg::VG::**index_paths**(void)

void vg::VG::**clear_node_indexes**(void)

void vg::VG::**clear_node_indexes_no_resize**(void)

void vg::VG::**clear_edge_indexes**(void)

void vg::VG::**clear_edge_indexes_no_resize**(void)

void vg::VG::**clear_indexes**(void)

void vg::VG::**clear_indexes_no_resize**(void)

void vg::VG::**resize_indexes**(void)

void vg::VG::**rebuild_indexes**(void)

void vg::VG::**rebuild_edge_indexes**(void)

void vg::VG::**merge**(*Graph* &*g*)
Literally merge protobufs.

void vg::VG::**merge**(*VG* &*g*)
Literally merge protobufs.

void vg::VG::**clear_paths**(void)
Clear the paths object (which indexes the graph.paths) and the graph paths themselves.

void vg::VG::**sync_paths**(void)
Synchronize in-memory indexes and protobuf graph.

void vg::VG::**merge_union**(*VG* &*g*)
Merge protobufs after removing overlaps. Good when there aren't many overlaps.

void vg::VG::**remove_duplicated_in**(*VG* &*g*)
Helper to merge_union.

void vg::VG::**remove_duplicates**(void)
Remove duplicated nodes and edges.

void vg::VG::**prune_complex_paths**(int *length*, int *edge_max*, *Node* *head_node, *Node* *tail_node)
Limit the local complexity of the graph, connecting pruned components to a head and tail node depending on the direction which we come into the node when the edge_max is passed.

void vg::VG::**prune_short_subgraphs**(size_t *min_size*)

void vg::VG::**serialize_to_ostream**(ostream &*out*, *id_t* *chunk_size* = 1000)
Write to a stream in chunked graphs.

void vg::VG::**serialize_to_file**(**const** string &*file_name*, *id_t* *chunk_size* = 1000)

id_t vg::VG::**max_node_id**(void)
Get the maximum node ID in the graph.

id_t vg::VG::**min_node_id**(void)
Get the minimum node ID in the graph.

void vg::VG::**compact_ids**(void)
Squish the node IDs down into as small a space as possible. Fixes up paths itself.

void vg::VG::**increment_node_ids**(*id_t* *increment*)
Add the given value to all node IDs. Preserves the paths.

void vg::VG::**decrement_node_ids**(*id_t* *decrement*)
Subtract the given value from all the node IDs. Must not create a node with 0 or negative IDs. Invalidates the paths.

void vg::VG::**swap_node_id**(*id_t* *node_id*, *id_t* *new_id*)
Change the ID of the node with the first id to the second, new ID not used by any node. Invalidates any paths containing the node, since they are not updated.

void vg::VG::**swap_node_id**(*Node* *node, *id_t* *new_id*)
Change the ID of the given node to the second, new ID not used by any node. Invalidates the paths. Invalidates any paths containing the node, since they are not updated.

void vg::VG::**extend**(*VG* &*g*, bool *warn_on_duplicates* = false)
Iteratively add when nodes and edges are novel. Good when there are very many overlaps. TODO: If you are using this with warn on duplicates on, and you know there shouldn't be any duplicates, maybe you should use merge instead. This version sorts paths on rank after adding in the path mappings from the other graph.

void vg::VG::**extend**(*Graph* &*graph*, bool *warn_on_duplicates* = false)
This version does not sort path mappings by rank. In order to preserve paths, call *Paths::sort_by_mapping_rank()* and *Paths::rebuild_mapping_aux()* after you are done adding in graphs to this graph.

void vg::VG::**append**(*VG* &*g*)
Add another graph into this graph, attaching tails to heads. Modify ids of the second graph to ensure we don't have conflicts. Then attach tails of this graph to the heads of the other, and extend(g).

void vg::VG::**combine**(*VG* &*g*)
Add another graph into this graph. Don't append or join the nodes in the graphs; just ensure that ids are unique, then apply extend.

void vg::VG::**include**(**const** *Path* &*path*)
Edit the graph to include the path.

vector<*Translation*> vg::VG::**edit**(**const** vector<*Path*> &*paths*)

Edit the graph to include all the sequence and edges added by the given paths. Can handle paths that visit nodes in any orientation. Returns a vector of Translations, one per node existing after the edit, describing how each new or conserved node is embedded in the old graph. Note that this method sorts the graph and rebuilds the path index, so it should not be called in a loop.

vector<*Translation*> vg::VG::**edit_fast**(**const** *Path* &*path*, set<*NodeSide*> &*dangling*)

Edit the graph to include all the sequences and edges added by the given path. Returns a vector of Translations, one per original-node fragment. Completely novel nodes are not mentioned, and nodes with no Translations are assumed to be carried through unchanged. Invalidates the rank-based *Paths* index. Does not sort the graph. Suitable for calling in a loop. Can attach newly created nodes on the left of the path to the given set of dangling NodeSides, and populates the set at the end with the *NodeSide* corresponding to the end of the path.

void vg::VG::**find_breakpoints**(**const** *Path* &*path*, map<*id_t*, set<*pos_t*>> &*breakpoints*)

Find all the points at which a *Path* enters or leaves nodes in the graph. Adds them to the given map by node ID of sets of bases in the node that will need to become the starts of new nodes.

map<*pos_t*, *Node* *> vg::VG::**ensure_breakpoints**(**const** map<*id_t*, set<*pos_t*>> &*breakpoints*)

Take a map from node ID to a set of offsets at which new nodes should start (which may include 0 and 1-past-the-end, which should be ignored), break the specified nodes at those positions. Returns a map from old node start position to new node pointer in the graph. Note that the caller will have to crear and rebuild path rank data.

Returns a map from old node start position to new node. This map contains some entries pointing to null, for positions past the ends of original nodes. It also maps from positions on either strand of the old node to the same new node pointer; the new node's forward strand is always the same as the old node's forward strand.

map<*id_t*, set<*pos_t*>> vg::VG::**forwardize_breakpoints**(**const** map<*id_t*, set<*pos_t*>> &*breakpoints*)

Flips the breakpoints onto the forward strand.

void vg::VG::**add_nodes_and_edges**(**const** *Path* &*path*, **const** map<*pos_t*, *Node* *> &*node_translation*, map<pair<*pos_t*, string>, vector<*Node* *>> &*added_seqs*, map<*Node* *, *Path*> &*added_nodes*, **const** map<*id_t*, size_t> &*orig_node_sizes*, set<*NodeSide*> &*dangling*, size_t *max_node_size* = 1024)

Given a path on nodes that may or may not exist, and a map from start position in the old graph to a node in the current graph, add all the new sequence and edges required by the path. The given path must not contain adjacent perfect match edits in the same mapping, or any deletions on the start or end of mappings (the removal of which can be accomplished with the *Path::simplify()* function).

Outputs (and caches for subsequent calls) novel node runs in added_seqs, and *Paths* describing where novel nodes translate back to in the original graph in added_nodes. Also needs a map of the original sizes of nodes deleted from the original graph, for reverse complementing. If dangling is nonempty, left edges of nodes created for initial inserts will connect to the specified sides. At the end, dangling is populated with the side corresponding to the last edit in the path.

void vg::VG::**add_nodes_and_edges**(**const** *Path* &*path*, **const** map<*pos_t*, *Node* *> &*node_translation*, map<pair<*pos_t*, string>, vector<*Node* *>> &*added_seqs*, map<*Node* *, *Path*> &*added_nodes*, **const** map<*id_t*, size_t> &*orig_node_sizes*, size_t *max_node_size* = 1024)

This version doesn't require a set of dangling sides to populate.

vector<*Translation*> vg::VG::**make_translation**(**const** map<*pos_t*, *Node* *> &*node_translation*, **const** map<*Node* *, *Path*> &*added_nodes*, **const** map<*id_t*, size_t> &*orig_node_sizes*)
Produce a graph *Translation* object from information about the editing process.

void vg::VG::**add_node**(**const** *Node* &*node*)
Add in the given node, by value.

void vg::VG::**add_nodes**(**const** vector<*Node*> &*nodes*)
Add in the given nodes, by value.

void vg::VG::**add_edge**(**const** *Edge* &*edge*)
Add in the given edge, by value.

void vg::VG::**add_edges**(**const** vector<*Edge*> &*edges*)
Add in the given edges, by value.

void vg::VG::**add_edges**(**const** vector<*Edge* *> &*edges*)
Add in the given edges, by value.

void vg::VG::**add_nodes**(**const** set<*Node* *> &*nodes*)
Add in the given nodes, by value.

void vg::VG::**add_edges**(**const** set<*Edge* *> &*edges*)
Add in the given edges, by value.

size_t vg::VG::**node_count**(void) **const**
Return the number of nodes in the graph.

size_t vg::VG::**edge_count**(void) **const**
Count the number of edges in the graph.

id_t vg::VG::**total_length_of_nodes**(void)
Get the total sequence length of nodes in the graph. TODO: redundant with *length()*.

int vg::VG::**node_rank**(*Node* *node)
Get the rank of the node in the protobuf array that backs the graph.

int vg::VG::**node_rank**(*id_t* *id*)
Get the rank of the node in the protobuf array that backs the graph.

int vg::VG::**start_degree**(*Node* *node)
Get the number of edges attached to the start of a node.

int vg::VG::**end_degree**(*Node* *node)
Get the number of edges attached to the end of a node.

int vg::VG::**left_degree**(*NodeTraversal* *node*)
Get the number of edges attached to the left side of a *NodeTraversal*.

int vg::VG::**right_degree**(*NodeTraversal* *node*)
Get the number of edges attached to the right side of a *NodeTraversal*.

void vg::VG::**edges_of_node**(*Node* *node, vector<*Edge* *> &*edges*)
Get the edges of the specified node, and add them to the given vector. Guaranteed to add each edge only once per call.

vector<*Edge* *> vg::VG::**edges_of**(*Node* *node)
Get the edges of the specified node.

vector<*Edge* *> vg::VG::**edges_from**(*Node* *node)
Get the edges from the specified node.

vector<*Edge* *> vg::VG::**edges_to**(*Node* *node)
Get the edges to the specified node.

void vg::VG::**edges_of_nodes**(set<*Node* *> &*nodes*, set<*Edge* *> &*edges*)
Get the edges of the specified set of nodes, and add them to the given set of edge pointers.

set<*NodeSide*> vg::VG::**sides_to**(*NodeSide side*)
Get the sides on the other side of edges to this side of the node.

set<*NodeSide*> vg::VG::**sides_from**(*NodeSide side*)
Get the sides on the other side of edges from this side of the node.

set<*NodeSide*> vg::VG::**sides_from**(*id_t id*)
Get the sides from both sides of the node.

set<*NodeSide*> vg::VG::**sides_to**(*id_t id*)
Get the sides to both sides of the node.

set<*NodeSide*> vg::VG::**sides_of**(*NodeSide side*)
Union of sides_to and sides_from.

set<pair<*NodeSide*, bool>> vg::VG::**sides_context**(*id_t node_id*)
Get all sides connecting to this node.

bool vg::VG::**same_context**(*id_t id1*, *id_t id2*)
Use sides_from an sides_to to determine if both nodes have the same context.

bool vg::VG::**is_ancestor_prev**(*id_t node_id*, *id_t candidate_id*)
Determine if the node is a prev ancestor of this one.

bool vg::VG::**is_ancestor_prev**(*id_t node_id*, *id_t candidate_id*, set<*id_t*> &*seen*, size_t *steps* = 64)
Determine if the node is a prev ancestor of this one by trying to find it in a given number of steps.

bool vg::VG::**is_ancestor_next**(*id_t node_id*, *id_t candidate_id*)
Determine if the node is a next ancestor of this one.

bool vg::VG::**is_ancestor_next**(*id_t node_id*, *id_t candidate_id*, set<*id_t*> &*seen*, size_t *steps* = 64)
Determine if the node is a next ancestor of this one by trying to find it in a given number of steps.

id_t vg::VG::**common_ancestor_prev**(*id_t id1*, *id_t id2*, size_t *steps* = 64)
Try to find a common ancestor by walking back up to steps from the first node.

id_t vg::VG::**common_ancestor_next**(*id_t id1*, *id_t id2*, size_t *steps* = 64)
Try to find a common ancestor by walking forward up to steps from the first node.

set<*NodeTraversal*> vg::VG::**siblings_to**(**const** *NodeTraversal* &*traversal*)
To-siblings are nodes which also have edges to them from the same nodes as this one.

set<*NodeTraversal*> vg::VG::**siblings_from**(**const** *NodeTraversal* &*traversal*)
From-siblings are nodes which also have edges to them from the same nodes as this one.

set<*NodeTraversal*> vg::VG::**full_siblings_to**(**const** *NodeTraversal* &*trav*)
Full to-siblings are nodes traversals which share exactly the same upstream *NodeSide*s.

set<*NodeTraversal*> vg::VG::**full_siblings_from**(**const** *NodeTraversal* &*trav*)
Full from-siblings are nodes traversals which share exactly the same downstream *NodeSide*s.

set<*Node* *> vg::VG::**siblings_of**(*Node* **node*)
Get general siblings of a node.

void vg::VG::**simplify_siblings**(void)
Remove easily-resolvable redundancy in the graph.

void vg::VG::**simplify_to_siblings**(**const** set<set<*NodeTraversal*>> &*to_sibs*)
Remove easily-resolvable redundancy in the graph for all provided to-sibling sets.

void vg::VG::**simplify_from_siblings**(**const** set<set<*NodeTraversal*>> &*from_sibs*)
Remove easily-resolvable redundancy in the graph for all provided from-sibling sets.

set<set<*NodeTraversal*>> vg::VG::**transitive_sibling_sets**(**const** set<set<*NodeTraversal*>> &*sibs*)
Remove intransitive sibling sets, such as where (A, B, C) = S1 but C ∈ S2.

set<set<*NodeTraversal*>> vg::VG::**identically_oriented_sibling_sets**(**const** set<set<*NodeTraversal*>> &*sibs*)
Remove sibling sets which don't have identical orientation.

bool vg::VG::**adjacent**(**const** *Position* &*pos1*, **const** *Position* &*pos2*)
Determine if pos1 occurs directly before pos2.

Node *vg::VG::**create_node**(**const** string &*seq*, *id_t* *id* = 0)
Create a node. Use the *VG* class to generate ids.

Node *vg::VG::**get_node**(*id_t* *id*)
Find a particular node.

void vg::VG::**nonoverlapping_node_context_without_paths**(*Node* **node*, *VG* &*g*)
Get the subgraph of a node and all the edges it is responsible for (where it has the minimal ID) and add it into the given *VG*.

void vg::VG::**expand_context**(*VG* &*g*, size_t *distance*, bool *add_paths* = true, bool *use_steps* = true)
Expand the context of what's already in the given graph by the given distance, either in nodes or in bases. Pulls material from this graph.

void vg::VG::**expand_context_by_steps**(*VG* &*g*, size_t *steps*, bool *add_paths* = true)
Expand the context of the given graph by the given number of steps.

void vg::VG::**expand_context_by_length**(*VG* &*g*, size_t *length*, bool *add_paths* = true, bool *reflect* = false, **const** set<*NodeSide*> &*barriers* = set< NodeSide >())
Expand the context of the given graph by the given number of bases. If reflect is true, bounce off the ends of nodes to get siblings of nodes you came from. Can take a set of NodeSides not to look out from, that act as barriers to context expansion. These barriers will have no edges attached to them in the final graph.

void vg::VG::**destroy_node**(*Node* **node*)
Destroy the node at the given pointer. This pointer must point to a *Node* owned by the graph.

void vg::VG::**destroy_node**(*id_t* *id*)
Destroy the node with the given ID.

bool vg::VG::**has_node**(*id_t id*)
Determine if the graph has a node with the given ID.

bool vg::VG::**has_node**(*Node* **node*)
Determine if the graph contains the given node.

bool vg::VG::**has_node**(**const** *Node* &*node*)
Determine if the graph contains the given node.

Node *vg::VG::**find_node_by_name_or_add_new**(string *name*)
Find a node with the given name, or create a new one if none is found.

void vg::VG::**for_each_node**(function<void) *Node* *
> *lambda*Run the given function on every node.

void vg::VG::**for_each_node_parallel**(function<void) *Node* *
> *lambda*Run the given function on every node in parallel.

void vg::VG::**for_each_connected_node**(*Node* **node*, function<void) *Node* *
> *lambda*Go through all the nodes in the same connected component as the given node. Ignores relative orientation.

void vg::VG::**dfs**(**const** function<void) *NodeTraversal*
> &*node_begin_fn*, **const** function<void*NodeTraversal*> &*node_end_fn*, **const** function<boolvoid> &*break_fn*, **const** function<void*Edge* *> &*edge_fn*, **const** function<void*Edge* *> &*tree_fn*, **const** function<void*Edge* *> &*edge_curr_fn*, **const** function<void*Edge* *> &*edge_cross_fn*, **const** vector<*NodeTraversal*> **sources*, **const** unordered_set<*NodeTraversal*> **sinks*

Parameters

- `node_begin_fn`: Called when node orientattion is first encountered.
- `node_end_fn`: Called when node orientation goes out of scope.
- `break_fn`: Called to check if we should stop the DFS.
- `edge_fn`: Called when an edge is encountered.
- `tree_fn`: Called when an edge forms part of the DFS spanning tree.
- `edge_curr_fn`: Called when we meet an edge in the current tree component.
- `edge_cross_fn`: Called when we meet an edge in an already-traversed tree component.
- `sources`: Start only at these node traversals.
- `sinks`: When hitting a sink, don't keep walking.

Do a DFS search of the bidirected graph. A bidirected DFS starts at some root node, and traverses first all the nodes found reading out the right of that node in their appropriate relative orientations (including the root), and then all the nodes found reading left out of that node in their appropriate orientations (including the root). If any unvisited nodes are left in other connected components, the process will repeat from one such node, until all nodes have been visited in each orientation.

void vg::VG::**dfs**(**const** function<void) *NodeTraversal*
> &*node_begin_fn*, **const** function<void*NodeTraversal*> &*node_end_fn*, **const** vector<*NodeTraversal*> **sources* = NULL, **const** unordered_set<*NodeTraversal*> **sinks* = NULLSpecialization of dfs for only handling nodes.

void vg::VG::**dfs**(**const** function<void) *NodeTraversal*
> &*node_begin_fn*, **const** function<void*NodeTraversal*> &*node_end_fn*, **const** function<boolvoid> &*break_fn*Specialization of dfs for only handling nodes + break function.

bool vg::VG::**empty**(void)
Is the graph empty?

const string vg::VG::**hash**(void)
Generate a digest of the serialized graph.

void vg::VG::**remove_null_nodes**(void)
Remove nodes with no sequence. These are created in some cases during the process of graph construction.

void vg::VG::**remove_node_forwarding_edges**(*Node* **node*)
Remove a node but connect all of its predecessor and successor nodes with new edges.

void vg::VG::**remove_null_nodes_forwarding_edges**(void)
Remove null nodes but connect predecessors and successors, preserving structure.

void vg::VG::**remove_orphan_edges**(void)
Remove edges for which one of the nodes is not present.

void vg::VG::**remove_inverting_edges**(void)
Remove edges representing an inversion and edges on the reverse complement.

bool vg::VG::**has_inverting_edges**(void)
Determine if the graph has inversions.

void vg::VG::**keep_paths**(set<string> &*path_names*, set<string> &*kept_names*)
Keep paths in the given set of path names. Populates kept_names with the names of the paths it actually found to keep. The paths specified may not overlap. Removes all nodes and edges not used by one of the specified paths.

void vg::VG::**keep_path**(**const** string &*path_name*)

int vg::VG::**path_edge_count**(list<*NodeTraversal*> &*path*, int32_t *offset*, int *path_length*)
Path stats. Starting from offset in the first node, how many edges do we cross? path must be nonempty and longer than the given length. offset is interpreted as relative to the first node in its on-path orientation, and is inclusive.

int vg::VG::**path_end_node_offset**(list<*NodeTraversal*> &*path*, int32_t *offset*, int *path_length*)
Determine the offset in its last node at which the path starting at this offset in its first node ends. path must be nonempty and longer than the given length. offset is interpreted as relative to the first node in its on-path orientation, and is inclusive. Returned offset is remaining unused length in the last node touched.

const vector<*Alignment*> vg::VG::**paths_as_alignments**(void)
Convert the stored paths in this graph to alignments.

const string vg::VG::**path_sequence**(**const** *Path* &*path*)
Return sequence string of path.

double vg::VG::**path_identity**(**const** *Path* &*path1*, **const** *Path* &*path2*)
Return percent identity between two paths (# matches / (#matches + #mismatches)). Note: uses ssw aligner, so will only work on small paths.

string vg::VG::**trav_sequence**(**const** *NodeTraversal* &*trav*)
Get the sequence of a *NodeTraversal*.

id_t vg::VG::**get_node_at_nucleotide**(string *pathname*, int *nuc*)
Takes in a pathname and the nucleotide position (like from a vcf) and returns the node id which contains that position.

Edge *vg::VG::**create_edge**(*Node* *from*, *Node* *to*, bool *from_start* = false, bool *to_end* = false)
Create an edge. If the given edge cannot be created, returns null. If the given edge already exists, returns the existing edge.

Edge *vg::VG::**create_edge**(*id_t from*, *id_t to*, bool *from_start* = false, bool *to_end* = false)
Create an edge. If the given edge cannot be created, returns null. If the given edge already exists, returns the existing edge.

Edge *vg::VG::**create_edge**(*NodeTraversal left*, *NodeTraversal right*)
Make a left-to-right edge from the left *NodeTraversal* to the right one, respecting orientations. If the given edge cannot be created, returns null. If the given edge already exists, returns the existing edge.

Edge *vg::VG::**create_edge**(*NodeSide side1*, *NodeSide side2*)
Make an edge connecting the given sides of nodes. If the given edge cannot be created, returns null. If the given edge already exists, returns the existing edge.

Edge *vg::VG::**get_edge**(**const** *NodeSide* &*side1*, **const** *NodeSide* &*side2*)
Get a pointer to the specified edge. This can take sides in any order.

Edge *vg::VG::**get_edge**(**const** pair<*NodeSide*, *NodeSide*> &*sides*)
Get a pointer to the specified edge. This can take sides in any order.

Edge *vg::VG::**get_edge**(**const** *NodeTraversal* &*left*, **const** *NodeTraversal* &*right*)
Get the edge connecting the given oriented nodes in the given order.

void vg::VG::**destroy_edge**(*Edge* *edge)
Destroy the edge at the given pointer. This pointer must point to an edge owned by the graph.

void vg::VG::**destroy_edge**(**const** *NodeSide* &*side1*, **const** *NodeSide* &*side2*)
Destroy the edge between the given sides of nodes. These can be in either order.

void vg::VG::**destroy_edge**(**const** pair<*NodeSide*, *NodeSide*> &*sides*)
Destroy the edge between the given sides of nodes. This can take sides in any order.

void vg::VG::**unindex_edge_by_node_sides**(**const** *NodeSide* &*side1*, **const** *NodeSide* &*side2*)
Remove an edge from the node side indexes, so it doesn't show up when you ask for the edges connected to the side of a node. Makes the edge untraversable until the indexes are rebuilt.

void vg::VG::**unindex_edge_by_node_sides**(*Edge* *edge)
Remove an edge from the node side indexes, so it doesn't show up when you ask for the edges connected to the side of a node. Makes the edge untraversable until the indexes are rebuilt.

void vg::VG::**index_edge_by_node_sides**(*Edge* *edge)
Add an edge to the node side indexes. Doesn't touch the index of edges by node pairs or the graph; those must be updated seperately.

bool vg::VG::**has_edge**(**const** *NodeSide* &*side1*, **const** *NodeSide* &*side2*)
Get the edge between the given node sides, which can be in either order.

bool vg::VG::**has_edge**(**const** pair<*NodeSide*, *NodeSide*> &*sides*)
Determine if the graph has an edge. This can take sides in any order.

bool vg::VG::**has_edge**(*Edge* *edge)
Determine if the graph has an edge. This can take sides in any order.

bool vg::VG::**has_edge**(**const** *Edge* &*edge*)
Determine if the graph has an edge. This can take sides in any order.

bool vg::VG::**has_inverting_edge**(*Node* *n)
Determine if the graph has an inverting edge on the given node.

bool vg::VG::**has_inverting_edge_from**(*Node* *n)
Determine if the graph has an inverting edge from the given node.

bool vg::VG::**has_inverting_edge_to**(*Node* *n)
Determine if the graph has an inverting edge to the given node.

void vg::VG::**for_each_edge**(function<void) *Edge* *
> *lambda*Run the given function for each edge.

void vg::VG::**for_each_edge_parallel**(function<void) *Edge* *
> *lambda*Run the given function for each edge, in parallel.

void vg::VG::**circularize**(*id_t head*, *id_t tail*)
Circularize a subgraph / path using the head / tail nodes.

void vg::VG::**circularize**(vector<string> *pathnames*)

void vg::VG::**connect_node_to_nodes**(*NodeTraversal node*, vector<*NodeTraversal*> &*nodes*)
Connect node -> nodes. Connects from the right side of the first to the left side of the second.

void vg::VG::**connect_node_to_nodes**(*Node* *node, vector<*Node* *> &*nodes*, bool *from_start* = false)
Connect node -> nodes. You can optionally use the start of the first node instead of the end.

void vg::VG::**connect_nodes_to_node**(vector<*NodeTraversal*> &*nodes*, *NodeTraversal node*)
connect nodes -> node. Connects from the right side of the first to the left side of the second.

void vg::VG::**connect_nodes_to_node**(vector<*Node* *> &*nodes*, *Node* *node, bool *to_end* = false)
connect nodes -> node.

void vg::VG::**divide_node**(*Node* *node, int *pos*, *Node* *&*left*, *Node* *&*right*)
Divide a node at a given internal position. Inserts the new nodes in the correct paths, but can't update the ranks, so they need to be cleared and re-calculated by the caller.

void vg::VG::**divide_node**(*Node* *node, vector<int> &*positions*, vector<*Node* *> &*parts*)
Divide a node at a given internal position. This version works on a collection of internal positions, in linear time.

void vg::VG::**divide_path**(map<long, *id_t*> &*path*, long *pos*, *Node* *&*left*, *Node* *&*right*)
Divide a path at a position. Also invalidates stored rank information.

void vg::VG::**to_dot**(ostream &*out*, vector<*Alignment*> *alignments* = {}, vector<*Locus*> *loci* = {}, bool *show_paths* = false, bool *walk_paths* = false, bool *annotate_paths* = false, bool *show_mappings* = false, bool *simple_mode* = false, bool *invert_edge_ports* = false, bool *color_variants* = false, bool *ultrabubble_labeling* = false, bool *skip_missing_nodes* = false, bool *ascii_labels* = false, int *random_seed* = 0)
Convert the graph to Dot format.

void vg::VG::**to_dot**(ostream &*out*, vector<*Alignment*> *alignments* = {}, bool *show_paths* = false, bool *walk_paths* = false, bool *annotate_paths* = false, bool *show_mappings* = false, bool *invert_edge_ports* = false, int *random_seed* = 0, bool *color_variants* = false)
Convert the graph to Dot format.

void vg::VG::**to_gfa**(ostream &*out*)
Convert the graph to GFA format.

void vg::VG::**to_turtle**(ostream &*out*, **const** string &*rdf_base_uri*, bool *precompress*)
Convert the graph to Turtle format.

bool vg::VG::**is_valid**(bool *check_nodes* = true, bool *check_edges* = true, bool *check_paths* = true, bool *check_orphans* = true)
Determine if the graph is valid or not, according to the specified criteria.

void vg::VG::**lazy_sort**(void)
Topologically order the nodes in the Protobuf graph. Only valid if the graph is a DAG with all no reversing edges or doubly reversing edges. No guarantee of system independent behavior, but significantly faster than VG::sort().

void vg::VG::**swap_nodes**(*Node* *a, *Node* *b)
Swap the given nodes. TODO: what does that mean?

void vg::VG::**force_path_match**(void)
For each path, assign edits that describe a total match of the mapping to the node.

void vg::VG::**fill_empty_path_mappings**(void)
For each path, if a mapping has no edits then make it a perfect match against a node. This is the same as force_path_match, but only for empty mappings.

Alignment vg::VG::**align**(**const** string &*sequence*, *Aligner* *aligner, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)
Align without base quality adjusted scores. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

Alignment vg::VG::**align**(**const** *Alignment* &*alignment*, *Aligner* *aligner, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)
Align without base quality adjusted scores. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

Alignment vg::VG::**align**(**const** *Alignment* &*alignment*, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)
Align with default *Aligner*. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

Alignment vg::VG::**align**(**const** string &*sequence*, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)
Align with default *Aligner*. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

Alignment vg::VG::**align_qual_adjusted**(**const** *Alignment* &*alignment*, *QualAdjAligner* **qual_adj_aligner*, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)

Align with base quality adjusted scores. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

Alignment vg::VG::**align_qual_adjusted**(**const** string &*sequence*, *QualAdjAligner* **qual_adj_aligner*, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)

Align with base quality adjusted scores. Align to the graph. May modify the graph by re-ordering the nodes. May add nodes to the graph, but cleans them up afterward.

void vg::VG::**for_each_kpath**(int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(list<*NodeTraversal*>::iterator, list<*NodeTraversal*>&)> *lambda*)

Calls a function on all node-crossing paths with up to length across node boundaries. Considers each node in forward orientation to produce the kpaths around it.

void vg::VG::**for_each_kpath_parallel**(int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(list<*NodeTraversal*>::iterator, list<*NodeTraversal*>&)> *lambda*)

Calls a function on all kpaths of the given node.

void vg::VG::**for_each_kpath**(int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(size_t, *Path*&)> *lambda*)

Calls a function on all node-crossing paths with up to length across node boundaries. Considers each node in forward orientation to produce the kpaths around it.

void vg::VG::**for_each_kpath_parallel**(int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(size_t, *Path*&)> *lambda*)

Calls a function on all kpaths of the given node.

void vg::VG::**for_each_kpath_of_node**(*Node* **node*, int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(list<*NodeTraversal*>::iterator, list<*NodeTraversal*>&)> *lambda*)

Calls a function on all kpaths of the given node.

void vg::VG::**for_each_kpath_of_node**(*Node* **n*, int *k*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *handle_prev_maxed*, function<void(*NodeTraversal*)> *handle_next_maxed*, function<void(size_t, *Path*&)> *lambda*)

Calls a function on all kpaths of the given node.

void vg::VG::**kpaths**(set<list<*NodeTraversal*>> &*paths*, int *length*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*)> *prev_maxed*, function<void(*NodeTraversal*)> *next_maxed*)

Get kpaths. TODO: what is this for?

void vg::VG::**kpaths**(vector<*Path*> &*paths*, int *length*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*
> *prev_maxed*, function<void*NodeTraversal*> *next_maxed*Get kpaths. TODO: what is this for?

void vg::VG::**kpaths_of_node**(*Node* *node*, set<list<*NodeTraversal*>> &*paths*, int *length*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*
> *prev_maxed*, function<void*NodeTraversal*> *next_maxed*Get kpaths on a particular node. TODO: what is this for?

void vg::VG::**kpaths_of_node**(*Node* *node*, vector<*Path*> &*paths*, int *length*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*
> *prev_maxed*, function<void*NodeTraversal*> *next_maxed*Get kpaths on a particular node. TODO: what is this for?

void vg::VG::**kpaths_of_node**(*id_t node_id*, vector<*Path*> &*paths*, int *length*, bool *path_only*, int *edge_max*, function<void(*NodeTraversal*
> *prev_maxed*, function<void*NodeTraversal*> *next_maxed*Get kpaths on a particular node. TODO: what is this for?

void vg::VG::**prev_kpaths_from_node**(*NodeTraversal node*, int *length*, bool *path_only*, int *edge_max*, bool *edge_bounding*, list<*NodeTraversal*> *postfix*, set<list<*NodeTraversal*>> &*walked_paths*, **const** vector<string> &*followed_paths*, function<void(*NodeTraversal*
> &*maxed_nodes*Given an oriented start node, a length in bp, a maximum number of edges to cross, and a stack of nodes visited so far, fill in the set of paths with all the paths starting at the oriented start node and going left off its end no longer than the specified length, calling maxed_nodes on nodes which can't be visited due to the edge-crossing limit. Produces paths ending with the specified node. TODO: postfix should not be (potentially) copied on every call.

void vg::VG::**next_kpaths_from_node**(*NodeTraversal node*, int *length*, bool *path_only*, int *edge_max*, bool *edge_bounding*, list<*NodeTraversal*> *prefix*, set<list<*NodeTraversal*>> &*walked_paths*, **const** vector<string> &*followed_paths*, function<void(*NodeTraversal*
> &*maxed_nodes*Do the same as prec_kpaths_from_node, except going right, producing a path starting with the specified node.

void vg::VG::**paths_between**(*Node* *from*, *Node* *to*, vector<*Path*> &*paths*)

void vg::VG::**paths_between**(*id_t from*, *id_t to*, vector<*Path*> &*paths*)

void vg::VG::**likelihoods**(vector<*Alignment*> &*alignments*, vector<*Path*> &*paths*, vector<long double> &*likelihoods*)

void vg::VG::**nodes_prev**(*NodeTraversal n*, vector<*NodeTraversal*> &*nodes*)
Get the nodes attached to the left side of the given *NodeTraversal*, in their proper orientations.

vector<*NodeTraversal*> vg::VG::**nodes_prev**(*NodeTraversal n*)
Get the nodes attached to the left side of the given *NodeTraversal*, in their proper orientations.

set<*NodeTraversal*> vg::VG::**travs_to**(*NodeTraversal node*)
Get traversals before this node on the same strand. Same as nodes_prev but using set.

void vg::VG::**nodes_next**(*NodeTraversal n*, vector<*NodeTraversal*> &*nodes*)
Get the nodes attached to the right side of the given *NodeTraversal*, in their proper orientations.

vector<*NodeTraversal*> vg::VG::**nodes_next**(*NodeTraversal n*)
Get the nodes attached to the right side of the given *NodeTraversal*, in their proper orientations.

set<*NodeTraversal*> vg::VG::**travs_from**(*NodeTraversal node*)
Get traversals after this node on the same strand. Same as nodes_next but using set.

set<*NodeTraversal*> vg::VG::**travs_of**(*NodeTraversal node*)
Get traversals either before or after this node on the same strand.

int vg::VG::**node_count_prev**(*NodeTraversal n*)
Count the nodes attached to the left side of the given *NodeTraversal*.

int vg::VG::**node_count_next**(*NodeTraversal n*)
Count the nodes attached to the right side of the given *NodeTraversal*.

Path vg::VG::**create_path**(**const** list<*NodeTraversal*> &*nodes*)
Create a path.

Path vg::VG::**create_path**(**const** vector<*NodeTraversal*> &*nodes*)
Create a path.

string vg::VG::**path_string**(**const** list<*NodeTraversal*> &*nodes*)
Get the string sequence for all the NodeTraversals on the given path.

string vg::VG::**path_string**(**const** *Path* &*path*)
Get the string sequence for traversing the given path. Assumes the path covers the entirety of any nodes visited. Handles backward nodes.

void vg::VG::**expand_path**(**const** list<*NodeTraversal*> &*path*, vector<*NodeTraversal*> &*expanded*)
Expand a path. TODO: what does that mean?

void vg::VG::**node_starts_in_path**(**const** list<*NodeTraversal*> &*path*, map<*Node* *, int> &*node_start*)
Fill in the node_start map with the first index along the path at which each node appears. Caller is responsible for dealing with orientations.

bool vg::VG::**nodes_are_perfect_path_neighbors**(*NodeTraversal left*, *NodeTraversal right*)
Return true if nodes share all paths and the mappings they share in these paths are adjacent, in the specified relative order and orientation.

bool vg::VG::**mapping_is_total_match**(**const** *Mapping* &*m*)
Return true if the mapping completely covers the node it maps to and is a perfect match.

map<string, vector<*Mapping*>> vg::VG::**concat_mappings_for_node_pair**(*id_t id1*, *id_t id2*)
Concatenate the mappings for a pair of nodes; handles multiple mappings per path.

map<string, vector<*Mapping*>> vg::VG::**concat_mappings_for_nodes**(**const** list<*NodeTraversal*> &*nodes*)
Concatenate mappings for a list of nodes that we want to concatenate. Returns, for each path name, a vector of merged mappings, once per path traversal of the run of nodes. Those merged mappings are in the orientation of the merged node (so mappings to nodes that are traversed in reverse will have their flags toggled). We assume that all mappings on the given nodes are full-length perfect matches, and that all the nodes are perfect path neighbors.

void vg::VG::**expand_path**(list<*NodeTraversal*> &*path*, vector<list<*NodeTraversal*>::iterator> &*expanded*)
Expand a path. TODO: what does that mean? These versions handle paths in which nodes can be traversed multiple times. Unfortunately since we're throwing non-const iterators around, we can't take the input path as const.

void vg::VG::**node_starts_in_path**(list<*NodeTraversal*> &*path*, map<*NodeTraversal* *, int> &*node_start*)

Find node starts in a path. TODO: what does that mean? To get the starts out of the map this produces, you need to dereference the iterator and then get the address of the *NodeTraversal* (stored in the list) that you are talking about.

void vg::VG::**for_each_kmer_parallel**(int *kmer_size*, bool *path_only*, int *edge_max*, function<void(string&, list<*NodeTraversal*>::iterator, int, list<*NodeTraversal*>&, *VG*&)> *lambda*, int *stride* = 1, bool *allow_dups* = false, bool *allow_negatives* = false)

Call a function for each kmer in the graph, in parallel.

void vg::VG::**for_each_kmer**(int *kmer_size*, bool *path_only*, int *edge_max*, function<void(string&, list<*NodeTraversal*>::iterator, int, list<*NodeTraversal*>&, *VG*&)> *lambda*, int *stride* = 1, bool *allow_dups* = false, bool *allow_negatives* = false)

Call a function for each kmer in the graph.

void vg::VG::**for_each_kmer_of_node**(*Node* *node, int *kmer_size*, bool *path_only*, int *edge_max*, function<void(string&, list<*NodeTraversal*>::iterator, int, list<*NodeTraversal*>&, *VG*&)> *lambda*, int *stride* = 1, bool *allow_dups* = false, bool *allow_negatives* = false)

Call a function for each kmer on a node.

void vg::VG::**kmer_context**(string &*kmer*, int *kmer_size*, bool *path_only*, int *edge_max*, bool *forward_only*, list<*NodeTraversal*> &*path*, list<*NodeTraversal*>::iterator *start_node*, int32_t *start_offset*, list<*NodeTraversal*>::iterator &*end_node*, int32_t &*end_offset*, set<tuple<char, *id_t*, bool, int32_t>> &*prev_positions*, set<tuple<char, *id_t*, bool, int32_t>> &*next_positions*)

For the given kmer of the given length starting at the given offset into the given *Node* along the given path, fill in end_node and end_offset with where the end of the kmer falls (counting from the right side of the *NodeTraversal*), prev_chars with the characters that preceed it, next_chars with the characters that follow it, prev_ and next_positions with the ((node ID, orientation), offset) pairs of the places you can come from/go next (from the right end of the kmer). Refuses to follow more than edge_max edges. Offsets are in the path orientation. Meant for gcsa2.

void vg::VG::**gcsa_handle_node_in_graph**(*Node* *node, int *kmer_size*, bool *path_only*, int *edge_max*, int *stride*, bool *forward_only*, *Node* *head_node, *Node* *tail_node, function<void(*KmerPosition*&)> *lambda*)

Do the GCSA2 kmers for a node. head_node and tail_node must both be non- null, but only one of those nodes actually needs to be in the graph. They will be examined directly to get their representative characters. They also don't need to be actually owned by the graph; they can be copies.

void vg::VG::**for_each_gcsa_kmer_position_parallel**(int *kmer_size*, bool *path_only*, int *edge_max*, int *stride*, bool *forward_only*, *id_t* &*head_id*, *id_t* &*tail_id*, function<void(*KmerPosition*&)> *lambda*)

Call a function for each GCSA2 kemr position in parallel. GCSA kmers are the kmers in the graph with each node existing in both its forward and reverse-complement orientation. *Node* IDs in the GCSA graph are 2 * original node ID, +1 if the GCSA node represents the reverse complement, and +0 if it does not. Non-reversing edges link the forward copy of the from node to the forward copy of the to node, and similarly for the reverse complement copies, while reversing edges link the forward copy of the from node to the *reverse complement* copy of the to node, and visa versa. This allows us to index both the

forward and reverse strands of every node, and to deal with GCSA's lack of support for reversing edges, with the same trick. Note that start_tail_id, if zero, will be replaced with the ID actually used for the start/end node before lambda is ever called.

void vg::VG::**get_gcsa_kmers**(int *kmer_size*, bool *path_only*, int *edge_max*, int *stride*, bool *forward_only*, **const** function<void) vector<gcsa::KMer>&, bool > &*handle_kmers*, *id_t* &*head_id*, *id_t* &*tail_id*Get the GCSA2 kmers in the graph.

void vg::VG::**write_gcsa_kmers**(int *kmer_size*, bool *path_only*, int *edge_max*, int *stride*, bool *forward_only*, ostream &*out*, *id_t* &*head_id*, *id_t* &*tail_id*)
Writhe the GCSA2 kmer file for the graph to the goven stream.

string vg::VG::**write_gcsa_kmers_to_tmpfile**(int *kmer_size*, bool *paths_only*, bool *forward_only*, *id_t* &*head_id*, *id_t* &*tail_id*, size_t *doubling_steps* = 2, size_t *size_limit* = 200, **const** string &*base_file_name* = ".vg-kmers-tmp-")
Write the GCSA2 kmers to a temp file with the given base. Return the name of the file.

void vg::VG::**build_gcsa_lcp**(gcsa::GCSA *&*gcsa*, gcsa::LCPArray *&*lcp*, int *kmer_size*, bool *paths_only*, bool *forward_only*, size_t *doubling_steps* = 2, size_t *size_limit* = 200, **const** string &*base_file_name* = ".vg-kmers-tmp-")
Construct the GCSA2 index for this graph.

void vg::VG::**prune_complex**(int *path_length*, int *edge_max*, *Node* *head_node*, *Node* *tail_node*)
Take all nodes that would introduce paths of > edge_max edge crossings, remove them, and link their neighbors to head_node or tail_node depending on which direction the path extension was stopped. For pruning graph prior to indexing with gcsa2.

void vg::VG::**prune_complex_with_head_tail**(int *path_length*, int *edge_max*)
Wrap the graph with heads and tails before doing the prune. Utility function for preparing for indexing.

Alignment vg::VG::**random_read**(size_t *read_len*, mt19937 &*rng*, *id_t* *min_id*, *id_t* *max_id*, bool *either_strand*)
Generate random reads. Note that even if either_strand is false, having backward nodes in the graph will result in some reads from the global reverse strand.

void vg::VG::**disjoint_subgraphs**(list<*VG*> &*subgraphs*)
Find subgraphs.

void vg::VG::**head_nodes**(vector<*Node* *> &*nodes*)
Get the head nodes (nodes with edges only to their right sides). These are required to be oriented forward.

vector<*Node* *> vg::VG::**head_nodes**(void)
Get the head nodes (nodes with edges only to their right sides). These are required to be oriented forward.

bool vg::VG::**is_head_node**(*id_t* *id*)
Determine if a node is a head node.

bool vg::VG::**is_head_node**(*Node* *node*)
Determine if a node is a head node.

int32_t vg::VG::**distance_to_head**(*NodeTraversal* *node*, int32_t *limit* = 1000)
Get the distance in bases from start of node to closest head node of graph, or -1 if that distance exceeds the limit.

int32_t vg::VG::**distance_to_head**(*NodeTraversal node*, int32_t *limit*, int32_t *dist*, set<*NodeTraversal*> &*seen*)
Get the distance in bases from start of node to closest head node of graph, or -1 if that distance exceeds the limit. dist increases by the number of bases of each previous node until you reach the head node seen is a set that holds the nodes that you have already gotten the distance of, but starts off empty

vector<*Node* *> vg::VG::**tail_nodes**(void)
Get the tail nodes (nodes with edges only to their left sides). These are required to be oriented forward.

void vg::VG::**tail_nodes**(vector<*Node* *> &*nodes*)
Get the tail nodes (nodes with edges only to their left sides). These are required to be oriented forward.

bool vg::VG::**is_tail_node**(*id_t id*)
Determine if a node is a tail node.

bool vg::VG::**is_tail_node**(*Node* **node*)
Determine if a node is a tail node.

int32_t vg::VG::**distance_to_tail**(*NodeTraversal node*, int32_t *limit* = 1000)
Get the distance from tail of node to end of graph, or -1 if limit exceeded.

int32_t vg::VG::**distance_to_tail**(*NodeTraversal node*, int32_t *limit*, int32_t *dist*, set<*NodeTraversal*> &*seen*)
Get the distance in bases from end of node to closest tail of graph, or -1 if that distance exceeds the limit. dist increases by the number of bases of each next node until you reach the tail node seen is a set that holds the nodes that you have already gotten the distance of, but starts off empty

int32_t vg::VG::**distance_to_tail**(*id_t id*, int32_t *limit* = 1000)
Get the distance from tail of node to end of graph, or -1 if limit exceeded.

void vg::VG::**collect_subgraph**(*Node* **node*, set<*Node* *> &*subgraph*)
Collect the subgraph of a *Node*. TODO: what does that mean?

Node *vg::VG::**join_heads**(void)
Join head nodes of graph to common null node, creating a new single head.

void vg::VG::**join_heads**(*Node* **node*, bool *from_start* = false)
Join head nodes of graph to specified node. Optionally from the start/to the end of the new node.

void vg::VG::**join_tails**(*Node* **node*, bool *to_end* = false)
Join tail nodes of graph to specified node. Optionally from the start/to the end of the new node.

void vg::VG::**wrap_with_null_nodes**(void)
Add singular head and tail null nodes to graph.

void vg::VG::**add_start_end_markers**(int *length*, char *start_char*, char *end_char*, *Node* *&*start_node*, *Node* *&*end_node*, *id_t start_id* = 0, *id_t end_id* = 0)
Add a start node and an end node, where all existing heads in the graph are connected to the start node, and all existing tails in the graph are connected to the end node. Any connected components in the graph which do not have either are connected to the start at an arbitrary point, and the end node from nodes going to that arbitrary point. If start_node or end_node is null, a new node will be created. Otherwise, the passed node will be used. Note that this visits every node, to make sure it is attached to all connected components. Note that if a graph has, say, heads but no tails, the start node will be attached buut the end node will be free-floating.

Public Members

Graph vg::VG::**graph**
Protobuf-based representation.

Paths vg::VG::**paths**
Manages paths of the graph. Initialized by setting paths._paths = graph.paths.

string vg::VG::**name**
Name of the graph.

id_t vg::VG::**current_id**
Current id for *Node* to be added next.

hash_map<*id_t*, *Node* *> vg::VG::**node_by_id**
*Node*s by id.

pair_hash_map<pair<*NodeSide*, *NodeSide*>, *Edge* *> vg::VG::**edge_by_sides**
*Edge*s by sides of *Node*s they connect. Since duplicate edges are not permitted, two edges cannot connect the same pair of node sides. Each edge is indexed here with the smaller *NodeSide* first. The actual node order is recorded in the *Edge* object.

hash_map<*Node* *, int> vg::VG::**node_index**
nodes by position in nodes repeated field. this is critical to allow fast deletion of nodes

hash_map<*Edge* *, int> vg::VG::**edge_index**

hash_map<*id_t*, vector<pair<*id_t*, bool>>> vg::VG::**edges_on_start**
Stores the destinations and backward flags for edges attached to the starts of nodes (whether that node is "from" or "to").

hash_map<*id_t*, vector<pair<*id_t*, bool>>> vg::VG::**edges_on_end**
Stores the destinations and backward flags for edges attached to the ends of nodes (whether that node is "from" or "to").

map<string, *SnarlTraversal*> vg::VG::**variant_to_traversal**

Private Functions

void vg::VG::**_for_each_kmer**(int *kmer_size*, bool *path_only*, int *edge_max*, function<void(string&, list<*NodeTraversal*>::iterator, int, list<*NodeTraversal*>&, *VG*&)> *lambda*, bool *parallel*, int *stride*, bool *allow_dups*, bool *allow_negatives*, *Node* **node* = nullptr)
Call the given function on each kmer. If parallel is specified, goes through nodes one per thread. If node is not null, looks only at kmers of that specific node.

Alignment vg::VG::**align**(**const** *Alignment* &*alignment*, *Aligner* **aligner*, *QualAdjAligner* **qual_adj_aligner*, bool *traceback* = true, bool *acyclic* = false, size_t *max_query_graph_ratio* = 0, bool *pinned_alignment* = false, bool *pin_left* = false, bool *banded_global* = false, size_t *band_padding_override* = 0, size_t *max_span* = 0, bool *print_score_matrices* = false)
Private method to funnel other align functions into. max_span specifies the min distance to unfold the graph to, and is meant to be the longest path that the specified sequence could cover, accounting for deletions. If it's less than the sequence's length, the sequence's length is used. band_padding_override gives the band padding to use for banded global alignment. In banded global mode, if the band padding

override is nonzero, permissive banding is not used, and instead the given band padding is provided. If the band padding override is not provided, the max span is used as the band padding and permissive banding is enabled.

void vg::VG::**init**(void)
setup, ensures that gssw == NULL on startup

Private Members

vector<*id_t*> vg::VG::**empty_ids**
Placeholder for functions that sometimes need to be passed an empty vector.

vector<pair<*id_t*, bool>> vg::VG::**empty_edge_ends**
Placeholder for functions that sometimes need to be passed an empty vector.

Private Static Attributes

const size_t vg::VG::**HIGH_BIT**

const size_t vg::VG::**LOW_BITS**

class *#include <vg_set.hpp>*

Public Functions

vg::VGset::**VGset**()

vg::VGset::**VGset**(vector<string> &*files*)

void vg::VGset::**transform**(*std*::function<void) *VG* *
> *lambda*

void vg::VGset::**for_each**(*std*::function<void) *VG* *
> *lambda*

int64_t vg::VGset::**merge_id_space**(void)

void vg::VGset::**to_xg**(*xg*::*XG* &*index*, bool *store_threads* = false)
Transforms to a succinct, queryable representation.

void vg::VGset::**to_xg**(*xg*::*XG* &*index*, bool *store_threads*, **const** regex &*paths_to_take*, map<string, *Path*> &*removed_paths*)
As above, except paths with names matching the given regex are removed and returned separately by inserting them into the provided map.

void vg::VGset::**store_in_index**(*Index* &*index*)

void vg::VGset::**store_paths_in_index**(*Index* &*index*)

void vg::VGset::**index_kmers**(*Index* &*index*, int *kmer_size*, bool *path_only*, int *edge_max*, int *stride* = 1, bool *allow_negatives* = false)

void vg::VGset::**for_each_kmer_parallel**(**const** function<void) string&, list<*NodeTraversal*>::iterator, int, list<*NodeTraversal*>&, *VG*&
> &*lambda*, int *kmer_size*, bool *path_only*, int *edge_max*, int *stride*, bool *allow_dups*, bool *allow_negatives* = false

void vg::VGset::**write_gcsa_out**(ostream &*out*, int *kmer_size*, bool *path_only*, bool *forward_only*, int64_t *head_id* = 0, int64_t *tail_id* = 0)

void vg::VGset::**write_gcsa_kmers_binary**(ostream &*out*, int *kmer_size*, bool *path_only*, bool *forward_only*, int64_t *head_id* = 0, int64_t *tail_id* = 0)

void vg::VGset::**get_gcsa_kmers**(int *kmer_size*, bool *path_only*, bool *forward_only*, **const** function<void) vector<gcsa::KMer>&, bool > &*handle_kmers*, int64_t *head_id* = 0, int64_t *tail_id* = 0

vector<string> vg::VGset::**write_gcsa_kmers_binary**(int *kmer_size*, bool *path_only*, bool *forward_only*, int64_t *head_id* = 0, int64_t *tail_id* = 0)

Public Members

vector<string> vg::VGset::**filenames**

bool vg::VGset::**show_progress**

Private Functions

void vg::VGset::**for_each_gcsa_kmer_position_parallel**(int *kmer_size*, bool *path_only*, bool *forward_only*, int64_t &*head_id*, int64_t &*tail_id*, function<void) *KmerPosition*& > *lambda*

struct Describes a step of a walk through a *Snarl* either on a node or through a child *Snarl*. **Public Members**

int64 vg::Visit::**node_id**
The node ID or snarl of this step (only one should be given)

Snarl vg::Visit::**snarl**
only needs to contain the start and end Visits

bool vg::Visit::**backward**
Indicates: if node_id is specified reverse complement of node if feature_id is specified traversal of a child snarl entering backwards through end and leaving backwards through start

struct *#include <flow_sort.hpp>* **Public Functions**

void vg::FlowSort::WeightedGraph::**construct**(*FlowSort* &*fs*, **const** string &*ref_name*, bool *isGrooming* = true)

Public Members

EdgeMapping vg::FlowSort::WeightedGraph::**edges_out_nodes**

EdgeMapping vg::FlowSort::WeightedGraph::**edges_in_nodes**

map<*Edge* *, int> vg::FlowSort::WeightedGraph::**edge_weight**
class *#include <vcf_buffer.hpp>* Provides a look-around buffer for VCFs where you can look at each variant in the context of nearby variants. Also caches parsings of genotypes, so you can iterate over genotypes efficiently without parsing them out over and over again. **Public Functions**

vg::WindowedVcfBuffer::**WindowedVcfBuffer**(vcflib::VariantCallFile **file*, size_t *window_size*)
Make a new *WindowedVcfBuffer* buffering the file at the given pointer (which must outlive the buffer, but which may be null). The VCF in the file must be sorted, but may contain overlapping variants.

bool vg::WindowedVcfBuffer::**next**()
Advance to the next variant, making it the current variant. Returns true if a next variant exists, and false if no next variant can be found. Must be called (and return true) before the first call to *get()* after constructing the *WindowedVcfBuffer* or setting the region.

tuple<vector<vcflib::Variant *>, vcflib::Variant *, vector<vcflib::Variant *>> vg::WindowedVcfBuffer::**get**()
Get the current variant in its context. Throws an exception if no variant is current. Returns a vector of variants in the window before the current variant, the current variant, and a vector of variants in the window after the current variant.

Pointers will be invalidated upon the next call to *next()* or *set_region()*.

tuple<vector<vcflib::Variant *>, vcflib::Variant *, vector<vcflib::Variant *>> vg::WindowedVcfBuffer::**get_nonoverl**
Like *get()*, but elides variants in the context that overlap the current variant, or each other.

const vector<vector<int>> &vg::WindowedVcfBuffer::**get_parsed_genotypes**(vcflib::Variant **variant*)
Given a pointer to a cached variant owned by this *WindowedVcfBuffer* (such as might be obtained from *get()*), return the cached parsed-out genotypes for all the samples, in the order the samples appear in the VCF file.

Returns a reference which is valid until the variant passed in is scrolled out of the buffer.

bool vg::WindowedVcfBuffer::**has_tabix**() **const**
This returns true if we have a tabix index, and false otherwise. If this is false, set_region may be called, but will do nothing and return false.

bool vg::WindowedVcfBuffer::**set_region**(**const** string &*contig*, int64_t *start* = -1, int64_t *end* = -1)
This tries to set the region on the underlying vcflib VariantCallFile to the given contig and region, if specified. Coordinates coming in should be 0-based, and will be converted to 1-based internally.

Returns true if the region was successfully set, and false otherwise (for example, if there is not tabix index, or if the given region is not part of this VCF. Note that if there is a tabix index, and set_region returns false, the position in the VCF file is undefined until the next successful set_region call.

If either of start and end are specified, then both of start and end must be specified.

Discards any variants previously in the buffer.

Protected Attributes

VcfBuffer vg::WindowedVcfBuffer::**reader**

size_t vg::WindowedVcfBuffer::**window_size**

list<unique_ptr<vcflib::Variant>> vg::WindowedVcfBuffer::**variants_before**

list<unique_ptr<vcflib::Variant>> vg::WindowedVcfBuffer::**variants_after**

unique_ptr<vcflib::Variant> vg::WindowedVcfBuffer::**current**

map<vcflib::Variant *, vector<vector<int>>> vg::WindowedVcfBuffer::**cached_genotypes**

vector<size_t> vg::WindowedVcfBuffer::**map_order_to_original**

Protected Static Functions

vector<int> vg::WindowedVcfBuffer::**decompose_genotype_fast**(**const** string &*genotype*)

Quickly decompose a genotype without any string copies.

Private Functions

vg::WindowedVcfBuffer::**WindowedVcfBuffer**(**const** *WindowedVcfBuffer* &*other*)

WindowedVcfBuffer &vg::WindowedVcfBuffer::**operator=**(**const** *WindowedVcfBuffer* &*other*)

class *#include <xg.hpp>* Provides succinct storage for a graph, its positional paths, and a set of embedded threads. Inherits from *vg::HandleGraph* **Public Types**

using

using

Public Functions

xg::XG::**XG**(void)

xg::XG::**~XG**(void)

xg::XG::**XG**(istream &*in*)

xg::XG::**XG**(Graph &*graph*)

xg::XG::**XG**(function<void) function<void Graph&>> *get_chunks*

xg::XG::**XG**(**const** *XG* &*other*)

xg::XG::**XG**(*XG* &&*other*)

XG &xg::XG::**operator=**(**const** *XG* &*other*)

XG &xg::XG::**operator=**(*XG* &&*other*)

void xg::XG::**from_stream**(istream &*in*, bool *validate_graph* = false, bool *print_graph* = false, bool *store_threads* = false, bool *is_sorted_dag* = false)

void xg::XG::**from_graph**(Graph &*graph*, bool *validate_graph* = false, bool *print_graph* = false, bool *store_threads* = false, bool *is_sorted_dag* = false)

void xg::XG::**from_callback**(function<void) function<void Graph&>> *get_chunks*, bool *validate_graph* = false, bool *print_graph* = false, bool *store_threads* = false, bool *is_sorted_dag* = false

void xg::XG::**build**(vector<pair<*id_t*, string>> &*node_label*, unordered_map<*side_t*, vector<*side_t*>> &*from_to*, unordered_map<*side_t*, vector<*side_t*>> &*to_from*, map<string, vector<*trav_t*>> &*path_nodes*, bool *validate_graph*, bool *print_graph*, bool *store_threads*, bool *is_sorted_dag*)

void xg::XG::**load**(istream &*in*)

size_t xg::XG::**serialize**(*std*::ostream &*out*, *sdsl*::structure_tree_node *v = NULL, *std*::string *name* = "")

const uint64_t *xg::XG::**sequence_data**(void) **const**

const size_t xg::XG::**sequence_bit_size**(void) **const**

size_t xg::XG::**id_to_rank**(int64_t *id*) **const**

int64_t xg::XG::**rank_to_id**(size_t *rank*) **const**

size_t xg::XG::**max_node_rank**(void) **const**

bool xg::XG::**has_node**(int64_t *id*) **const**

int64_t xg::XG::**node_at_seq_pos**(size_t *pos*) **const**
Get the node ID at the given sequence position. Works in 1-based coordinates.

size_t xg::XG::**node_start**(int64_t *id*) **const**

Node xg::XG::**node**(int64_t *id*) **const**

string xg::XG::**node_sequence**(int64_t *id*) **const**

size_t xg::XG::**node_length**(int64_t *id*) **const**

char xg::XG::**pos_char**(int64_t *id*, bool *is_rev*, size_t *off*) **const**

string xg::XG::**pos_substr**(int64_t *id*, bool *is_rev*, size_t *off*, size_t *len* = 0) **const**

size_t xg::XG::**node_graph_idx**(int64_t *id*) **const**

size_t xg::XG::**edge_graph_idx**(**const** Edge &*edge*) **const**

bool xg::XG::**has_edge**(int64_t *id1*, bool *is_start*, int64_t *id2*, bool *is_end*) **const**
Returns true if the given edge is present in the given orientation, and false otherwise.

bool xg::XG::**has_edge**(**const** Edge &*edge*) **const**
Returns true if the given edge is present in either orientation, and false otherwise.

vector<Edge> xg::XG::**edges_of**(int64_t *id*) **const**

vector<Edge> xg::XG::**edges_to**(int64_t *id*) **const**

vector<Edge> xg::XG::**edges_from**(int64_t *id*) **const**

vector<Edge> xg::XG::**edges_on_start**(int64_t *id*) **const**

vector<Edge> xg::XG::**edges_on_end**(int64_t *id*) **const**

Edge xg::XG::**canonicalize**(**const** Edge &*edge*) **const**
Get the rank of the edge, or numeric_limits<size_t>.max() if no such edge exists.

Graph xg::XG::**node_subgraph_id**(int64_t *id*) **const**
use the unified graph storage to return a node subgraph

Graph xg::XG::**node_subgraph_g**(int64_t *g*) **const**
returns the graph with graph offsets rather than ids on edges and nodes

Graph xg::XG::**graph_context_id**(**const** pos_t &*pos*, int64_t *length*) **const**
provide the graph context up to a given length from the current position; step by nodes

Graph xg::XG::**graph_context_g**(**const** pos_t &*pos*, int64_t *length*) **const**

Edge xg::XG::**edge_from_encoding**(int64_t *from*, int64_t *to*, int *type*) **const**
return an edge from the three-part encoding used in the graph vector Edge type encoding: 1: end to start 2: end to end 3: start to start 4: start to end

void xg::XG::**idify_graph**(Graph &*graph*) **const**

int xg::XG::**edge_type**(bool *from_start*, bool *to_end*) **const**
a numerical code for the edge type (based on the two reversal flags)

int xg::XG::**edge_type**(**const** Edge &*edge*) **const**

handle_t xg::XG::**get_handle**(**const** *id_t* &*node_id*, bool *is_reverse*) **const**
Look up the handle for the node with the given ID in the given orientation.

id_t xg::XG::**get_id**(**const** handle_t &*handle*) **const**
Get the ID from a handle.

bool xg::XG::**get_is_reverse**(**const** handle_t &*handle*) **const**
Get the orientation of a handle.

handle_t xg::XG::**flip**(**const** handle_t &*handle*) **const**
Invert the orientation of a handle (potentially without getting its ID)

size_t xg::XG::**get_length**(**const** handle_t &*handle*) **const**
Get the length of a node.

string xg::XG::**get_sequence**(**const** handle_t &*handle*) **const**
Get the sequence of a node, presented in the handle's local forward orientation.

bool xg::XG::**follow_edges**(**const** handle_t &*handle*, bool *go_left*, **const** function<bool) **const**
handle_t&
> &*iteratee* **const**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.

void xg::XG::**for_each_handle**(**const** function<bool) **const** handle_t&
> &*iteratee* **const**Loop over all the nodes in the graph in their local forward orientations, in their internal stored order. Stop if the iteratee returns false.

size_t xg::XG::**node_size**() **const**
Return the number of nodes in the graph.

void xg::XG::**neighborhood**(int64_t *id*, size_t *dist*, Graph &*g*, bool *use_steps* = true) **const**

void xg::XG::**for_path_range**(**const** string &*name*, int64_t *start*, int64_t *stop*, function<void) int64_t node_id
> *lambda*, bool *is_rev* = false **const**

void xg::XG::**get_path_range**(**const** string &*name*, int64_t *start*, int64_t *stop*, Graph &*g*, bool *is_rev* = false) **const**

void xg::XG::**expand_context**(Graph &*g*, size_t *dist*, bool *add_paths* = true, bool *use_steps* = true, bool *expand_forward* = true, bool *expand_backward* = true, int64_t *until_node* = 0) **const**

void xg::XG::**expand_context_by_steps**(Graph &*g*, size_t *steps*, bool *add_paths* = true, bool *expand_forward* = true, bool *expand_backward* = true, int64_t *until_node* = 0) **const**

void xg::XG::**expand_context_by_length**(Graph &*g*, size_t *length*, bool *add_paths* = true, bool *expand_forward* = true, bool *expand_backward* = true, int64_t *until_node* = 0) **const**

void xg::XG::**get_connected_nodes**(Graph &*g*) **const**

void xg::XG::**get_id_range**(int64_t *id1*, int64_t *id2*, Graph &*g*) **const**

void xg::XG::**get_id_range_by_length**(int64_t *id1*, int64_t *length*, Graph &*g*, bool *forward*) **const**

Path xg::XG::**path**(**const** string &*name*) **const**

const *XGPath* &xg::XG::**get_path**(**const** string &*name*) **const**

size_t xg::XG::**path_rank**(**const** string &*name*) **const**

size_t xg::XG::**max_path_rank**(void) **const**

string xg::XG::**path_name**(size_t *rank*) **const**

vector<size_t> xg::XG::**paths_of_node**(int64_t *id*) **const**

map<string, vector<Mapping>> xg::XG::**node_mappings**(int64_t *id*) **const**

bool xg::XG::**path_contains_node**(**const** string &*name*, int64_t *id*) **const**

void xg::XG::**add_paths_to_graph**(map<int64_t, Node *> &*nodes*, Graph &*g*) **const**

size_t xg::XG::**node_occs_in_path**(int64_t *id*, **const** string &*name*) **const**

size_t xg::XG::**node_occs_in_path**(int64_t *id*, size_t *rank*) **const**

vector<size_t> xg::XG::**node_ranks_in_path**(int64_t *id*, **const** string &*name*) **const**

vector<size_t> xg::XG::**node_ranks_in_path**(int64_t *id*, size_t *rank*) **const**

vector<size_t> xg::XG::**position_in_path**(int64_t *id*, **const** string &*name*) **const**

vector<size_t> xg::XG::**position_in_path**(int64_t *id*, size_t *rank*) **const**

map<string, vector<size_t>> xg::XG::**position_in_paths**(int64_t *id*, bool *is_rev* = false, size_t *offset* = 0) **const**

map<string, vector<pair<size_t, bool>>> xg::XG::**offsets_in_paths**(pos_t *pos*) **const**

map<string, vector<pair<size_t, bool>>> xg::XG::**nearest_offsets_in_paths**(pos_t *pos*, int64_t *max_search*) **const**

map<string, vector<size_t>> xg::XG::**distance_in_paths**(int64_t *id1*, bool *is_rev1*, size_t *offset1*, int64_t *id2*, bool *is_rev2*, size_t *offset2*) **const**

int64_t xg::XG::**min_distance_in_paths**(int64_t *id1*, bool *is_rev1*, size_t *offset1*, int64_t *id2*, bool *is_rev2*, size_t *offset2*) **const**

int64_t xg::XG::**node_at_path_position**(**const** string &*name*, size_t *pos*) **const**
Get the ID of the node that covers the given 0-based position along the path.

Mapping xg::XG::**mapping_at_path_position**(**const** string &*name*, size_t *pos*) **const**
Get the Mapping that covers the given 0-based position along the path.

size_t xg::XG::**node_start_at_path_position**(**const** string &*name*, size_t *pos*) **const**
Get the 0-based start position in the path that covers the given 0-based position along the path.

pos_t xg::XG::**graph_pos_at_path_position**(**const** string &*name*, size_t *pos*) **const**
Get the graph position at the given 0-based path position.

Alignment xg::XG::**target_alignment**(**const** string &*name*, size_t *pos1*, size_t *pos2*, **const** string &*feature*) **const**

size_t xg::XG::**path_length**(**const** string &*name*) **const**

size_t xg::XG::**path_length**(size_t *rank*) **const**

pair<int64_t, vector<size_t>> xg::XG::**nearest_path_node**(int64_t *id*, int *max_steps* = 16) **const**

int64_t xg::XG::**min_approx_path_distance**(int64_t *id1*, int64_t *id2*) **const**

pair<pos_t, int64_t> xg::XG::**next_path_position**(pos_t *pos*, int64_t *max_search*) **const**
nearest position that is in a path and the distance between it and the current position

bool xg::XG::**paths_on_same_component**(size_t *path_rank_1*, size_t *path_rank_2*) **const**
returns true if the paths are on the same connected component of the graph (constant time)

vector<pair<size_t, vector<pair<size_t, bool>>>> xg::XG::**oriented_paths_of_node**(int64_t *id*) **const**
returns all of the paths that a node traversal occurs on, the rank of these occurrences on the path and the orientation of the occurrences. false indicates that the traversal occurs in the same orientation as in the path, true indicates.

void xg::XG::**memoized_oriented_paths_of_node**(int64_t *id*, vector<pair<size_t, vector<pair<size_t, bool>>>> &*local_paths_var*, vector<pair<size_t, vector<pair<size_t, bool>>>> *&*paths_of_node_ptr_out*, unordered_map<*id_t*, vector<pair<size_t, vector<pair<size_t, bool>>>>> **paths_of_node_memo* = nullptr) **const**
sets a pointer to a memoized result from oriented_paths_of_node. if no memo is provided, the result will be queried and stored in local_paths_var and the pointer will be set to point to this variable so that no additional code paths are necessary

handle_t xg::XG::**memoized_get_handle**(int64_t *id*, bool *rev*, unordered_map<pair<int64_t, bool>, handle_t> **handle_memo* = nullptr) **const**
returns a the memoized result from get_handle if a memo is provided that the result has been queried previously otherwise, returns the result of get_handle directly and stores it in the memo if one is provided

int64_t xg::XG::**closest_shared_path_oriented_distance**(int64_t *id1*, size_t *offset1*, bool *rev1*, int64_t *id2*, size_t *offset2*, bool *rev2*, size_t *max_search_dist* = 100, unordered_map<*id_t*, vector<pair<size_t, vector<pair<size_t, bool>>>>> **paths_of_node_memo* = nullptr, unordered_map<pair<int64_t, bool>, handle_t> **handle_memo* = nullptr) **const**
the oriented distance (positive if pos2 is further along the path than pos1, otherwise negative) estimated by the distance along the nearest shared path to the two positions plus the distance to traverse to that path. returns numeric_limits<int64_t>::max() if no pair of nodes that occur same on the strand of a path are reachable within the max search distance (measured in sequence length, not node length).

vector<tuple<int64_t, bool, size_t>> xg::XG::**jump_along_closest_path**(int64_t *id*, bool *is_rev*, size_t *offset*, int64_t *jump_dist*, size_t *max_search_dist*, unordered_map<*id_t*, vector<pair<size_t, vector<pair<size_t, bool>>>>> **paths_of_node_memo* = nullptr, unordered_map<pair<int64_t, bool>, handle_t> **handle_memo* = nullptr) **const**
returns a vector of (node id, is reverse, offset) tuples that are found by jumping a fixed oriented distance along path(s) from the given position. if the position is not on a path, searches from the position to a path and adds/subtracts the search distance to the jump depending on the search direction. returns an empty vector if there is no path within the max search distance or if the jump distance goes past the end of the path

int64_t xg::XG::**where_to**(int64_t *current_side*, int64_t *visit_offset*, int64_t *new_side*) **const**

int64_t xg::XG::**where_to**(int64_t *current_side*, int64_t *visit_offset*, int64_t *new_side*, vector<Edge> &*edges_into_new*, vector<Edge> &*edges_out_of_old*) **const**

int64_t xg::XG::**node_height**(*ThreadMapping node*) **const**

void xg::XG::**set_thread_names**(**const** vector<string> &*names*)
Replace the existing thread names with the new names. Each name must be unique.

void xg::XG::**insert_thread**(**const** *thread_t* &*t*, **const** string &*name*)
Insert a thread. Name must be unique or empty. *bs_bake()* and *tn_bake()* need to be called before queries.

void xg::XG::**insert_threads_into_dag**(**const** vector<*thread_t*> &*t*, **const** vector<string> &*names*)
Insert a whole group of threads. Names should be unique or empty (though they aren't used yet). The indexed graph must be a DAG, at least in the subset traversed by the threads. (Reversing edges are fine, but the threads in a node must all run in the same direction.) This uses a special efficient batch insert algorithm for DAGs that lets us just scan the graph and generate nodes' B_s arrays independently. This must be called only once, and no threads can have been inserted previously. Otherwise the gPBWT data structures will be left in an inconsistent state.

auto xg::XG::**extract_threads**(bool *extract_reverse*) **const**
Read all the threads embedded in the graph.

thread_t xg::XG::**extract_thread**(**const** string &*name*) **const**
Extract a particular thread by name. Name may not be empty.

auto xg::XG::**extract_threads_matching**(**const** string &*pattern*, bool *reverse*) **const**
Extract a set of threads matching a pattern.

XG::*thread_t* xg::XG::**extract_thread**(*xg*::*XG*::*ThreadMapping* *node*, int64_t *offset*, int64_t *max_length*)
Extract a particular thread, referring to it by its offset at node; step it out to a maximum of max_length

size_t xg::XG::**count_matches**(**const** *thread_t* &*t*) **const**
Count matches to a subthread among embedded threads.

size_t xg::XG::**count_matches**(**const** Path &*t*) **const**

void xg::XG::**extend_search**(*ThreadSearchState* &*state*, **const** *thread_t* &*t*) **const**

void xg::XG::**extend_search**(*ThreadSearchState* &*state*, **const** *ThreadMapping* &*t*) **const**
Extend a search with the given single *ThreadMapping*.

XG::*ThreadSearchState* xg::XG::**select_starting**(**const** *ThreadMapping* &*start*) **const**
Select only the threads (if any) starting with a particular *ThreadMapping*, and not those continuing through it.

int64_t xg::XG::**id_rev_to_side**(int64_t *id*, bool *is_rev*) **const**
Take a node id and side and return the side id.

pair<int64_t, bool> xg::XG::**side_to_id_rev**(int64_t *side*) **const**
Take a side and give a node id / rev pair.

int64_t xg::XG::**threads_starting_on_side**(int64_t *side*) **const**
The number of threads starting at this side.

int64_t xg::XG::**thread_starting_at**(int64_t *side*, int64_t *offset*) **const**
Given a side and offset, return the id of the thread starting there (or 0 if none)

pair<int64_t, int64_t> xg::XG::**thread_start**(int64_t *thread_id*) **const**
Given a thread id and the reverse state get the starting side and offset.

string xg::XG::**thread_name**(int64_t *thread_id*) **const**
Given a thread id, return its name.

pair<int64_t, int64_t> xg::XG::**thread_start**(int64_t *thread_id*, bool *is_rev*) **const**
Gives the thread start for the given thread.

vector<int64_t> xg::XG::**threads_named_starting**(**const** string &*pattern*) **const**
Gives the thread ids of those whose names start with this pattern.

XG::*ThreadSearchState* xg::XG::**select_continuing**(**const** *ThreadMapping* &*start*) **const**
Select only the threads (if any) continuing through a particular *ThreadMapping*, and not those starting there.

void xg::XG::**bs_dump**(ostream &*out*) **const**

Public Members

size_t xg::XG::**seq_length**

size_t xg::XG::**node_count**

size_t xg::XG::**edge_count**

size_t xg::XG::**path_count**

char xg::XG::**start_marker**

char xg::XG::**end_marker**

Public Static Attributes

const uint32_t xg::XG::**MAX_INPUT_VERSION**

const uint32_t xg::XG::**OUTPUT_VERSION**

Private Types

using

Private Functions

bool xg::XG::**edge_filter**(int *type*, bool *is_to*, bool *want_left*, bool *is_reverse*) **const**
This is a utility function for the edge exploration. It says whether we want to visit an edge depending on its type, whether we're the to or from node, whether we want to look left or right, and whether we're forward or reverse on the node.

bool xg::XG::**do_edges**(**const** size_t &*g*, **const** size_t &*start*, **const** size_t &*count*, bool *is_to*, bool *want_left*, bool *is_reverse*, **const** function<bool) **const** handle_t& > &*iteratee* **const**

void xg::XG::**index_component_path_sets**()

void xg::XG::**create_succinct_component_path_sets**(int_vector<> &*path_ranks_iv_out*, bit_vector &*path_ranks_bv_out*) **const**

void xg::XG::**unpack_succinct_component_path_sets**(**const** int_vector<> &*path_ranks_iv*, **const** bit_vector &*path_ranks_bv*)

XG::*destination_t* xg::XG::**bs_get**(int64_t *side*, int64_t *offset*) **const**

size_t xg::XG::**bs_rank**(int64_t *side*, int64_t *offset*, *destination_t value*) **const**

void xg::XG::**bs_set**(int64_t *side*, vector<*destination_t*> *new_array*)

void xg::XG::**bs_insert**(int64_t *side*, int64_t *offset*, *destination_t value*)

void xg::XG::**bs_bake**()

void xg::XG::**tn_bake**()

Private Members

int_vector xg::XG::**g_iv**
locally traversable graph storage

Encoding designed for efficient compression, cache locality, and relativistic traversal of the graph.

node := { header, edges_to, edges_from } header := { node_id, node_start, node_length, edges_to_count, edges_from_count } node_id := integer node_start := integer (offset in s_iv) node_length := integer edges_to_count := integer edges_from_count := integer edges_to := { edge_to, ... } edges_from := { edge_from, ... } edge_to := { offset_to_previous_node, edge_type } edge_to := { offset_to_next_node, edge_type }

bit_vector xg::XG::**g_bv**
delimit node records to allow lookup of nodes in g_civ by rank

rank_support_v<1> xg::XG::**g_bv_rank**

bit_vector::select_1_type xg::XG::**g_bv_select**

int_vector xg::XG::**s_iv**

bit_vector xg::XG::**s_bv**

rank_support_v<1> xg::XG::**s_bv_rank**

bit_vector::select_1_type xg::XG::**s_bv_select**

int_vector xg::XG::**i_iv**

int64_t xg::XG::**min_id**

int64_t xg::XG::**max_id**

int_vector xg::XG::**r_iv**

int_vector xg::XG::**pn_iv**

csa_wt xg::XG::**pn_csa**

bit_vector xg::XG::**pn_bv**

rank_support_v<1> xg::XG::**pn_bv_rank**

bit_vector::select_1_type xg::XG::**pn_bv_select**

int_vector xg::XG::**pi_iv**

vector<*XGPath* *> xg::XG::**paths**

int_vector xg::XG::**np_iv**

bit_vector xg::XG::**np_bv**

rank_support_v<1> xg::XG::**np_bv_rank**

bit_vector::select_1_type xg::XG::**np_bv_select**

int_vector xg::XG::**h_iv**

vlc_vector xg::XG::**h_civ**

int_vector xg::XG::**ts_iv**

vlc_vector xg::XG::**ts_civ**

vector<string> xg::XG::**bs_arrays**

rank_select_int_vector xg::XG::**bs_single_array**

csa_bitcompressed xg::XG::**tn_csa**

sd_vector xg::XG::**tn_cbv**

sd_vector::rank_1_type xg::XG::**tn_cbv_rank**

sd_vector::select_1_type xg::XG::**tn_cbv_select**

vlc_vector xg::XG::**tin_civ**

vlc_vector xg::XG::**tio_civ**

wt_int xg::XG::**side_thread_wt**

string xg::XG::**names_str**

vector<unordered_set<size_t>> xg::XG::**component_path_sets**

vector<size_t> xg::XG::**component_path_set_of_path**

Private Static Attributes

const int xg::XG::**G_NODE_ID_OFFSET**

const int xg::XG::**G_NODE_SEQ_START_OFFSET**

const int xg::XG::**G_NODE_LENGTH_OFFSET**

const int xg::XG::**G_NODE_TO_COUNT_OFFSET**

const int xg::XG::**G_NODE_FROM_COUNT_OFFSET**

const int xg::XG::**G_NODE_HEADER_LENGTH**

const int xg::XG::**G_EDGE_OFFSET_OFFSET**

const int xg::XG::**G_EDGE_TYPE_OFFSET**

const int xg::XG::**G_EDGE_LENGTH**

const size_t xg::XG::**HIGH_BIT**

const size_t xg::XG::**LOW_BITS**

const *XG*::*destination_t* xg::XG::**BS_SEPARATOR**

const *XG*::*destination_t* xg::XG::**BS_NULL**
class *#include <xg.hpp>* Thrown when attempting to interpret invalid data as an *XG* index. Inherits from runtime_error
class *#include <xg.hpp>***Public Functions**

xg::XGPath::**XGPath**(void)

xg::XGPath::**~XGPath**(void)

xg::XGPath::**XGPath**(**const** string &*path_name*, **const** vector<*trav_t*> &*path*, size_t *node_count*, *XG* &*graph*, size_t **unique_member_count_out* = nullptr)

xg::XGPath::**XGPath**(**const** *XGPath* &*other*)

xg::XGPath::**XGPath**(*XGPath* &&*other*)

XGPath &xg::XGPath::**operator=**(**const** *XGPath* &*other*)

XGPath &xg::XGPath::**operator=**(*XGPath* &&*other*)

void xg::XGPath::**load**(istream &*in*)

size_t xg::XGPath::**serialize**(*std*::ostream &*out*, *sdsl*::structure_tree_node **v* = NULL, *std*::string *name* = "") **const**

Mapping xg::XGPath::**mapping**(size_t *offset*) **const**

Public Members

rrr_vector xg::XGPath::**nodes**

rrr_vector::rank_1_type xg::XGPath::**nodes_rank**

rrr_vector::select_1_type xg::XGPath::**nodes_select**

wt_gmr xg::XGPath::**ids**

sd_vector xg::XGPath::**directions**

int_vector xg::XGPath::**positions**

int_vector xg::XGPath::**ranks**

bit_vector xg::XGPath::**offsets**

rank_support_v<1> xg::XGPath::**offsets_rank**

bit_vector::select_1_type xg::XGPath::**offsets_select**

namespace
namespace
namespace

namespace
namespace
namespace

namespace
namespace **Functions**

template <typename T>
bool `stream::write`(*std*::ostream &*out*, uint64_t *element_count*, uint64_t *chunk_elements*, **const** *std*::function<T) uint64_t, uint64_t
> &*lambda*Write objects using adaptive chunking. Takes a stream to write to, a total element count to write, a guess at how manye elements should be in a chunk, and a function that, given a start element and a length, returns a Protobuf object representing that range of elements.

Adaptively sets the chunk size, in elements, so that no too-large Protobuf records are serialized.

template <typename T>
bool `stream::write`(*std*::ostream &*out*, uint64_t *count*, **const** *std*::function<T) uint64_t
> &*lambda*

template <typename T>
bool `stream::write_buffered`(*std*::ostream &*out*, *std*::vector<T> &*buffer*, uint64_t *buffer_limit*)

template <typename T>
void `stream::for_each`(*std*::istream &*in*, **const** *std*::function<void) T&
> &*lambda*, **const** *std*::function<voiduint64_t> &*handle_count*

template <typename T>
void `stream::for_each`(*std*::istream &*in*, **const** *std*::function<void) T&
> &*lambda*

template <typename T>
void `stream::for_each_parallel_impl`(*std*::istream &*in*, **const** *std*::function<void) T&, T&
> &*lambda2*, **const** *std*::function<voidT&> &*lambda1*, **const** *std*::function<voiduint64_t> &*handle_count*, **const** *std*::function<boolvoid> &*single_threaded_until_true*

template <typename T>
void `stream::for_each_interleaved_pair_parallel`(*std*::istream &*in*, **const** *std*::function<void) T&, T&
> &*lambda2*

template <typename T>
void `stream::for_each_interleaved_pair_parallel_after_wait`(*std*::istream &*in*, **const** *std*::function<void) T&, T&
> &*lambda2*, **const** *std*::function<boolvoid> &*single_threaded_until_true*

template <typename T>
void `stream::for_each_parallel`(*std*::istream &*in*, **const** *std*::function<void) T&
> &*lambda1*, **const** *std*::function<voiduint64_t> &*handle_count*

template <typename T>
void `stream::for_each_parallel`(*std*::istream &*in*, **const** *std*::function<void) T&
> &*lambda*

template <typename T>
std::pair<*ProtobufIterator*<T>, *ProtobufIterator*<T>> `stream::protobuf_iterator_range`(*std*::istream &*in*)
Returns iterators that act like begin() and end() for a stream containing protobuf data.

Variables

const size_t stream::**MAX_PROTOBUF_SIZE**
Protobuf will refuse to read messages longer than this size.

const size_t stream::**TARGET_PROTOBUF_SIZE**
We aim to generate messages that are this size.

namespace Typedefs

using vg::benchtime = typedef chrono::nanoseconds
We define a duration type for expressing benchmark times in.

typedef

typedef

using vg::real_t = typedef long double

typedef

using vg::Chain = typedef vector<const Snarl*>
Snarls are defined at the Protobuf level, but here is how we define chains as real objects.

The *SnarlManager* is going to have one official copy of each chain stored, and it will give you a pointer to it on demand.

typedef Represents a *Node* ID. ID type is a 64-bit signed int.

typedef Represents an offset along the sequence of a *Node*. Offsets are size_t.

typedef Represents an oriented position on a *Node*. *Position* type: id, direction, offset.

Enums

enum type vg::**MappingQualityMethod**
Values:

enum type vg::**SnarlType**
Enumeration of the classifications of snarls.

Values:

= 0

= 1

= 2

Functions

int vg::**hts_for_each**(string &*filename*, function<void) *Alignment*&
> *lambda*

int vg::**hts_for_each_parallel**(string &*filename*, function<void) *Alignment*&
> *lambda*

bam_hdr_t *vg::**hts_file_header**(string &*filename*, string &*header*)

bam_hdr_t *vg::**hts_string_header**(string &*header*, map<string, int64_t> &*path_length*, map<string, string> &*rg_sample*)

bool vg::**get_next_alignment_from_fastq**(gzFile *fp*, char **buffer*, size_t *len*, *Alignment* &*alignment*)

bool vg::**get_next_interleaved_alignment_pair_from_fastq**(gzFile *fp*, char **buffer*, size_t *len*, *Alignment* &*mate1*, *Alignment* &*mate2*)

bool vg::**get_next_alignment_pair_from_fastqs**(gzFile *fp1*, gzFile *fp2*, char **buffer*, size_t *len*, *Alignment* &*mate1*, *Alignment* &*mate2*)

size_t vg::**fastq_unpaired_for_each_parallel**(**const** string &*filename*, function<void(*Alignment*&)> *lambda*)

size_t vg::**fastq_paired_interleaved_for_each_parallel**(**const** string &*filename*, function<void(*Alignment*&, *Alignment*&)> *lambda*)

size_t vg::**fastq_paired_interleaved_for_each_parallel_after_wait**(**const** string &*filename*, function<void(*Alignment*&, *Alignment*&)> *lambda*, function<bool()> *single_threaded_until_true*)

size_t vg::**fastq_paired_two_files_for_each_parallel**(**const** string &*file1*, **const** string &*file2*, function<void(*Alignment*&, *Alignment*&)> *lambda*)

size_t vg::**fastq_paired_two_files_for_each_parallel_after_wait**(**const** string &*file1*, **const** string &*file2*, function<void(*Alignment*&, *Alignment*&)> *lambda*, function<bool()> *single_threaded_until_true*)

size_t vg::**fastq_unpaired_for_each**(**const** string &*filename*, function<void(*Alignment*&)> *lambda*)

size_t vg::**fastq_paired_interleaved_for_each**(**const** string &*filename*, function<void(*Alignment*&, *Alignment*&)> *lambda*)

size_t vg::**fastq_paired_two_files_for_each**(**const** string &*file1*, **const** string &*file2*, function<void(*Alignment*&, *Alignment*&)> *lambda*)

void vg::**parse_rg_sample_map**(char **hts_header*, map<string, string> &*rg_sample*)

void vg::**write_alignments**(*std*::ostream &*out*, vector<*Alignment*> &*buf*)

short vg::**quality_char_to_short**(char *c*)

char vg::**quality_short_to_char**(short *i*)

void vg::**alignment_quality_short_to_char**(*Alignment* &*alignment*)

string vg::**string_quality_short_to_char**(**const** string &*quality*)

void vg::**alignment_quality_char_to_short**(*Alignment* &*alignment*)

string vg::**string_quality_char_to_short**(**const** string &*quality*)

bam1_t *vg::**alignment_to_bam**(**const** string &*sam_header*, **const** *Alignment* &*alignment*, **const** string &*refseq*, **const** int32_t *refpos*, **const** bool *refrev*, **const** string &*cigar*, **const** string &*mateseq*, **const** int32_t *matepos*, **const** int32_t *tlen*)

string vg::**alignment_to_sam**(**const** *Alignment* &*alignment*, **const** string &*refseq*, **const** int32_t *refpos*, **const** bool *refrev*, **const** string &*cigar*, **const** string &*mateseq*, **const** int32_t *matepos*, **const** int32_t *tlen*)

string vg::**cigar_string**(vector<pair<int, char>> &*cigar*)

string vg::**mapping_string**(**const** string &*source*, **const** *Mapping* &*mapping*)

void vg::**mapping_cigar**(**const** *Mapping* &*mapping*, vector<pair<int, char>> &*cigar*)

string vg::**cigar_against_path**(**const** *Alignment* &*alignment*, bool *on_reverse_strand*)

int32_t vg::**sam_flag**(**const** *Alignment* &*alignment*, bool *on_reverse_strand*)

Alignment vg::**bam_to_alignment**(**const** bam1_t *b, map<string, string> &*rg_sample*)

int vg::**alignment_to_length**(**const** *Alignment* &*a*)

int vg::**alignment_from_length**(**const** *Alignment* &*a*)

Alignment vg::**strip_from_start**(**const** *Alignment* &*aln*, size_t *drop*)

Alignment vg::**strip_from_end**(**const** *Alignment* &*aln*, size_t *drop*)

Alignment vg::**trim_alignment**(**const** *Alignment* &*aln*, **const** *Position* &*pos1*, **const** *Position* &*pos2*)

vector<*Alignment*> vg::**alignment_ends**(**const** *Alignment* &*aln*, size_t *len1*, size_t *len2*)

Alignment vg::**alignment_middle**(**const** *Alignment* &*aln*, int *len*)

vector<*Alignment*> vg::**reverse_complement_alignments**(**const** vector<*Alignment*> &*alns*, **const** function<int64_t) int64_t > &*node_length*

Alignment vg::**reverse_complement_alignment**(**const** *Alignment* &*aln*, **const** function<int64_t) *id_t* > &*node_length*

void vg::**reverse_complement_alignment_in_place**(*Alignment* *aln, **const** function<int64_t) *id_t* > &*node_length*

Alignment vg::**merge_alignments**(**const** vector<*Alignment*> &*alns*, bool *debug*)

Alignment &vg::**extend_alignment**(*Alignment* &*a1*, **const** *Alignment* &*a2*, bool *debug*)

Alignment vg::**merge_alignments**(**const** *Alignment* &*a1*, **const** *Alignment* &*a2*, bool *debug*)

void vg::**translate_nodes**(*Alignment* &*a*, **const** unordered_map<*id_t*, pair<*id_t*, bool>> &*ids*, **const** *std*::function<size_t) int64_t > &*node_length*

void vg::**flip_nodes**(*Alignment* &*a*, **const** set<int64_t> &*ids*, **const** *std*::function<size_t) int64_t > &*node_length*

int vg::**softclip_start**(**const** *Alignment* &*alignment*)

int vg::**softclip_end**(**const** *Alignment* &*alignment*)

int vg::**query_overlap**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*)

int vg::**edit_count**(**const** *Alignment* &*alignment*)

size_t vg::**to_length_after_pos**(**const** *Alignment* &*aln*, **const** *Position* &*pos*)

size_t vg::**from_length_after_pos**(**const** *Alignment* &*aln*, **const** *Position* &*pos*)

size_t vg::**to_length_before_pos**(**const** *Alignment* &*aln*, **const** *Position* &*pos*)

size_t vg::**from_length_before_pos**(**const** *Alignment* &*aln*, **const** *Position* &*pos*)

const string vg::**hash_alignment**(**const** *Alignment* &*aln*)

Alignment vg::**simplify**(**const** *Alignment* &*a*)
Simplifies the *Path* in the *Alignment*. Note that this removes deletions at the start and end of Mappings, so code that handles simplified Alignments needs to handle offsets on internal Mappings.

void vg::**write_alignment_to_file**(**const** *Alignment* &*aln*, **const** string &*filename*)

map<*id_t*, int> vg::**alignment_quality_per_node**(**const** *Alignment* &*aln*)

string vg::**middle_signature**(**const** *Alignment* &*aln*, int *len*)

pair<string, string> vg::**middle_signature**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*, int *len*)

string vg::**signature**(**const** *Alignment* &*aln*)

pair<string, string> vg::**signature**(**const** *Alignment* &*aln1*, **const** *Alignment* &*aln2*)

void vg::**parse_bed_regions**(istream &*bedstream*, *xg*::*XG* *xgindex*, vector<*Alignment*> *out_alignments*)

int vg::**fastq_for_each**(string &*filename*, function<void) *Alignment*& > *lambda*

ostream &vg::**operator<<**(ostream &*out*, **const** *BenchmarkResult* &*result*)
Benchmark results can be output to streams

void vg::**benchmark_control**()
The benchmark control function, designed to take some amount of time that might vary with CPU load.

BenchmarkResult vg::**run_benchmark**(**const** string &*name*, size_t *iterations*, **const** function<void) void > &*under_test*Run the given function the given number of times, interleaved with runs of the control function, and return a *BenchmarkResult* describing its performance.

BenchmarkResult vg::**run_benchmark**(**const** string &*name*, size_t *iterations*, **const** function<void) void
> &*setup*, **const** function<voidvoid> &*under_test*Run a benchmark with a setup function.

Node vg::**xg_cached_node**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*)

vector<*Edge*> vg::**xg_cached_edges_of**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

vector<*Edge*> vg::**xg_cached_edges_on_start**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

vector<*Edge*> vg::**xg_cached_edges_on_end**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

string vg::**xg_cached_node_sequence**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*)

size_t vg::**xg_cached_node_length**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*)
Get the length of a *Node* from an *xg::XG* index, with cacheing of deserialized nodes.

int64_t vg::**xg_cached_node_start**(*id_t id*, *xg*::*XG* *xgidx, LRUCache<*id_t*, int64_t> &*node_start_cache*)
Get the node start position in the sequence vector.

char vg::**xg_cached_pos_char**(*pos_t pos*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*)
Get the character at a position in an *xg::XG* index, with cacheing of deserialized nodes.

map<*pos_t*, char> vg::**xg_cached_next_pos_chars**(*pos_t pos*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)
Get the characters at positions after the given position from an *xg::XG* index, with cacheing of deserialized nodes.

set<*pos_t*> vg::**xg_cached_next_pos**(*pos_t pos*, bool *whole_node*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

int64_t vg::**xg_cached_distance**(*pos_t pos1*, *pos_t pos2*, int64_t *maximum*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

set<*pos_t*> vg::**xg_cached_positions_bp_from**(*pos_t pos*, int64_t *distance*, bool *rev*, *xg*::*XG* *xgidx, LRUCache<*id_t*, *Node*> &*node_cache*, LRUCache<*id_t*, vector<*Edge*>> &*edge_cache*)

static void vg::**compute_side_components**(*VG* &*graph*, vector<*SideSet*> &*components*, *Side2Component* &*side_to_component*)

void *vg::**mergeNodeObjects**(void *a, void *b)

void vg::**getReachableBridges2**(stCactusEdgeEnd *edgeEnd1, stHash *bridgeEndsToBridgeNodes, stList *bridgeEnds)
Get the bridge ends that form boundary pairs with edgeEnd1, using the given getBridgeEdgeEndsToBridgeNodes hash map. Duplicated from the pinchesAndCacti tests.

void vg::**getReachableBridges**(stCactusEdgeEnd *edgeEnd1, stList *bridgeEnds)
Get the bridge ends that form boundary pairs with edgeEnd1. Duplicated from the pinchesAndCacti tests.

void vg::**addArbitraryTelomerePair**(vector<stCactusEdgeEnd *> *ends*, stList **telomeres*)
Finds an arbitrary pair of telomeres in a Cactus graph, which are are either a pair of bridge edge ends or a pair of chain edge ends, oriented such that they form a pair of boundaries.

Mostly copied from the pinchesAndCacti unit tests.

pair<stCactusGraph *, stList *> vg::**vg_to_cactus**(*VG* *&graph*, **const** unordered_set<string> *&hint_paths*)

VG vg::**cactus_to_vg**(stCactusGraph **cactus_graph*)

VG vg::**cactusify**(*VG* *&graph*)

Graph vg::**cluster_subgraph**(**const** *xg*::*XG* *&xg*, **const** *Alignment* *&aln*, **const** vector<*vg*::*MaximalExactMatch*> *&mems*, double *expansion*)
return a subgraph form an xg for a cluster of MEMs from the given alignment

Graph vg::**cluster_subgraph**(**const** *xg*::*XG* *&xg*, **const** *Alignment* *&aln*, **const** vector<*MaximalExactMatch*> *&mems*, double *expansion* = 1.61803)
return a subgraph form an xg for a cluster of MEMs from the given alignment

template <typename T>
bool vg::**convert**(**const** *std*::string *&s*, T *&r*)

template <typename T>
std::string vg::**convert**(**const** T *&r*)

real_t vg::**gamma_ln**(real_t *x*)
Calculate the natural log of the gamma function of the given argument.

real_t vg::**factorial_ln**(int *n*)
Calculate the natural log of the factorial of the given integer. TODO: replace with a cache or giant lookup table from Freebayes.

real_t vg::**pow_ln**(real_t *m*, int *n*)
Raise a log probability to a power

real_t vg::**choose_ln**(int *n*, int *k*)
Compute the number of ways to select k items from a collection of n distinguishable items, ignoring order. Returns the natural log of the (integer) result.

real_t vg::**multinomial_choose_ln**(int *n*, vector<int> *k*)
Compute the number of ways to select k_1, k_2, ... k_i items into i buckets from a collection of n distinguishable items, ignoring order. All of the items have to go into the buckets, so all k_i must sum to n. To compute choose you have to call this function with a 2-element vector, to represent the chosen and not-chosen buckets. Returns the natural log of the (integer) result.

TODO: Turns out we don't actually need this for the ambiguous multinomial after all.

real_t vg::**poisson_prob_ln**(int *observed*, real_t *expected*)
Compute the log probability of a Poisson-distributed process: observed events in an interval where expected events happen on average.

template <typename ProbIn>
real_t vg::**multinomial_sampling_prob_ln**(**const** vector<ProbIn> *&probs*, **const** vector<int> *&obs*)
Get the probability for sampling the counts in obs from a set of categories weighted by the probabilities in probs. Works for both double and real_t probabilities. Also works for binomials.

template <typename ProbIn>

real_t vg::**binomial_cmf_ln**(ProbIn *success_logprob*, size_t *trials*, size_t *successes*)
Compute the probability of having the given number of successes or fewer in the given number of trials, with the given success probability. Returns the resulting log probability.

template <typename ProbIn**>**
real_t vg::**geometric_sampling_prob_ln**(ProbIn *success_logprob*, size_t *trials*)
Get the log probability for sampling the given value from a geometric distribution with the given success log probability. The geometric distribution is the distribution of the number of trials, with a given success probability, required to observe a single success.

template <typename Iter**>**
bool vg::**advance_split**(Iter *start*, Iter *end*)
Given a split of items across a certain number of categories, as ints between the two given bidirectional iterators, advance to the next split and return true. If there is no next split, leave the collection unchanged and return false.

template <typename ProbIn**>**
real_t vg::**multinomial_censored_sampling_prob_ln**(**const** vector<ProbIn> &*probs*, **const** unordered_map<vector<bool>, int> &*obs*)
Get the log probability for sampling any actual set of category counts that is consistent with the constraints specified by obs, using the per-category probabilities defined in probs.

Obs maps from a vector of per-category flags (called a "class") to a number of items that might be in any of the flagged categories.

For example, if there are two equally likely categories, and one item flagged as potentially from either category, the probability of sampling a set of category counts consistent with that constraint is 1. If instead there are three equally likely categories, and one item flagged as potentially from two of the three but not the third, the probability of sampling a set of category counts consistent with that constraint is 2/3.

bool vg::**edit_is_match**(**const** *Edit* &*e*)

bool vg::**edit_is_sub**(**const** *Edit* &*e*)

bool vg::**edit_is_insertion**(**const** *Edit* &*e*)

bool vg::**edit_is_deletion**(**const** *Edit* &*e*)

pair<*Edit*, *Edit*> vg::**cut_edit_at_to**(**const** *Edit* &*e*, size_t *to_off*)

pair<*Edit*, *Edit*> vg::**cut_edit_at_from**(**const** *Edit* &*e*, size_t *from_off*)

Edit vg::**reverse_complement_edit**(**const** *Edit* &*e*)

bool vg::**operator==**(**const** *Edit* &*e1*, **const** *Edit* &*e2*)

double vg::**entropy**(**const** string &*st*)

double vg::**entropy**(**const** char **st*, size_t *len*)

Support vg::**make_support**(double *forward*, double *reverse*, double *quality*)
TBD Create a *Support* for the given forward and reverse coverage and quality.

double vg::**total**(**const** *Support* &*support*)
Get the total read support in a *Support*.

Support vg::**support_min**(**const** *Support* &*a*, **const** *Support* &*b*)
Get the minimum support of a pair of Supports, by taking the min in each orientation.

Support vg::**support_max**(**const** *Support* &*a*, **const** *Support* &*b*)
Get the maximum support of a pair of Supports, by taking the max in each orientation.

Support vg::**operator+**(**const** *Support* &*one*, **const** *Support* &*other*)
Add two *Support* values together, accounting for strand.

Support &vg::**operator+=**(*Support* &*one*, **const** *Support* &*other*)
Add in a *Support* to another.

bool vg::**operator<**(**const** *Support* &*a*, **const** *Support* &*b*)
Support less-than, based on total coverage.

bool vg::**operator>**(**const** *Support* &*a*, **const** *Support* &*b*)
Support greater-than, based on total coverage.

ostream &vg::**operator<<**(ostream &*stream*, **const** *Support* &*support*)
Allow printing a *Support*.

string vg::**to_vcf_genotype**(**const** *Genotype* &*gt*)
Get a VCF-style 1/2, 1|2|3, etc. string from a *Genotype*.

template <typename Scalar**>**
Support vg::**operator***(**const** *Support* &*support*, **const** Scalar &*scale*)
Scale a *Support* by a factor.

template <typename Scalar**>**
Support &vg::**operator*=**(*Support* &*support*, **const** Scalar &*scale*)
Scale a *Support* by a factor, in place.

template <typename Scalar**>**
Support vg::**operator***(**const** Scalar &*scale*, **const** *Support* &*support*)
Scale a *Support* by a factor, the other way

template <typename Scalar**>**
Support vg::**operator/**(**const** *Support* &*support*, **const** Scalar &*scale*)
Divide a *Support* by a factor.

template <typename Scalar**>**
Support &vg::**operator/=**(*Support* &*support*, **const** Scalar &*scale*)
Divide a *Support* by a factor, in place.

string vg::**allele_to_string**(*VG* &*graph*, **const** *Path* &*allele*)
Turn the given path into an allele. Drops the first and last mappings and looks up the sequences for the nodes of the others.

template <typename T**>**
void vg::**set_intersection**(**const** unordered_set<T> &*set_1*, **const** unordered_set<T> &*set_2*, unordered_set<T> **out_intersection*)

void vg::**sort_by_id_dedup_and_clean**(*Graph* &*graph*)
remove duplicates and sort by id

void vg::**remove_duplicates**(*Graph* &*graph*)
remove duplicate nodes and edges

void vg::**remove_duplicate_edges**(*Graph* &*graph*)
remove duplicate edges

void vg::**remove_duplicate_nodes**(*Graph* &*graph*)
remove duplicate nodes

void vg::**remove_orphan_edges**(*Graph* &*graph*)
remove edges that link to a node that is not in the graph

void vg::**sort_by_id**(*Graph* &*graph*)
order the nodes and edges in the graph by id

void vg::**sort_nodes_by_id**(*Graph* &*graph*)
order the nodes in the graph by id

void vg::**sort_edges_by_id**(*Graph* &*graph*)
order the edges in the graph by id pairs

bool vg::**is_id_sortable**(**const** *Graph* &*graph*)
returns true if the graph is id-sortable (no reverse links)

bool vg::**has_inversion**(**const** *Graph* &*graph*)
returns true if we find an edge that may specify an inversion

void vg::**flip_doubly_reversed_edges**(*Graph* &*graph*)
clean up doubly-reversed edges

int64_t &vg::**as_integer**(*handle_t* &*handle*)
View a handle as an integer.

const int64_t &vg::**as_integer**(**const** *handle_t* &*handle*)
View a const handle as a const integer.

handle_t &vg::**as_handle**(int64_t &*value*)
View an integer as a handle.

const *handle_t* &vg::**as_handle**(**const** int64_t &*value*)
View a const integer as a const handle.

bool vg::**operator==**(**const** *handle_t* &*a*, **const** *handle_t* &*b*)
Define equality on handles.

bool vg::**operator!=**(**const** *handle_t* &*a*, **const** *handle_t* &*b*)
Define inequality on handles.

void vg::**node_path_position**(int64_t *id*, string &*path_name*, int64_t &*position*, bool *backward*, int64_t &*offset*)

void vg::**index_positions**(*VG* &*graph*, map<long, *Node* *> &*node_path*, map<long, *Edge* *> &*edge_path*)

int vg::**sub_overlaps_of_first_aln**(**const** vector<*Alignment*> &*alns*, float *overlap_fraction*)

set<*pos_t*> vg::**gcsa_nodes_to_positions**(**const** vector<gcsa::node_type> &*nodes*)

const int vg::**balanced_stride**(int *read_length*, int *kmer_size*, int *stride*)

const vector<string> vg::**balanced_kmers**(**const** string &*seq*, **const** int *kmer_size*, **const** int *stride*)

pair<int64_t, int64_t> vg::**mem_min_oriented_distances**(**const** *MaximalExactMatch* &*m1*, **const** *MaximalExactMatch* &*m2*)

bool vg::**operator==**(**const** *MaximalExactMatch* &*m1*, **const** *MaximalExactMatch* &*m2*)

bool vg::**operator<**(**const** *MaximalExactMatch* &*m1*, **const** *MaximalExactMatch* &*m2*)

ostream &vg::**operator<<** (ostream &*out*, **const** *MaximalExactMatch* &*mem*)

const string vg::**mems_to_json** (**const** vector<*MaximalExactMatch*> &*mems*)

vector<string::const_iterator> vg::**cluster_cover** (**const** vector<*MaximalExactMatch*> &*cluster*)

int vg::**cluster_coverage** (**const** vector<*MaximalExactMatch*> &*cluster*)

bool vg::**mems_overlap** (**const** *MaximalExactMatch* &*mem1*, **const** *MaximalExactMatch* &*mem2*)

int vg::**mems_overlap_length** (**const** *MaximalExactMatch* &*mem1*, **const** *MaximalExactMatch* &*mem2*)

bool vg::**clusters_overlap_in_read** (**const** vector<*MaximalExactMatch*> &*cluster1*, **const** vector<*MaximalExactMatch*> &*cluster2*)

int vg::**clusters_overlap_length** (**const** vector<*MaximalExactMatch*> &*cluster1*, **const** vector<*MaximalExactMatch*> &*cluster2*)

vector<*pos_t*> vg::**cluster_nodes** (**const** vector<*MaximalExactMatch*> &*cluster*)

bool vg::**clusters_overlap_in_graph** (**const** vector<*MaximalExactMatch*> &*cluster1*, **const** vector<*MaximalExactMatch*> &*cluster2*)

void vg::**topologically_order_subpaths** (*MultipathAlignment* &*multipath_aln*)
Put subpaths in topological order (assumed to be true for other algorithms)

void vg::**identify_start_subpaths** (*MultipathAlignment* &*multipath_aln*)
Finds the start subpaths (i.e. the source nodes of the multipath DAG) and stores them in the 'start' field of the *MultipathAlignment*

int32_t vg::**optimal_alignment_internal** (**const** *MultipathAlignment* &*multipath_aln*, *Alignment* **aln_out*)

void vg::**optimal_alignment** (**const** *MultipathAlignment* &*multipath_aln*, *Alignment* &*aln_out*)
Stores the highest scoring alignment contained in the *MultipathAlignment* in an *Alignment*

Note: Assumes that each subpath's *Path* object uses one *Mapping* per node and that start subpaths have been identified

Args: multipath_aln multipath alignment to find optimal path through aln_out empty alignment to store optimal alignment in (data will be overwritten if not empty)

int32_t vg::**optimal_alignment_score** (**const** *MultipathAlignment* &*multipath_aln*)
Returns the score of the highest scoring alignment contained in the *MultipathAlignment*

Note: Assumes that each subpath's *Path* object uses one *Mapping* per node and that start subpaths have been identified

Args: multipath_aln multipath alignment to find optimal score in

void vg::**rev_comp_subpath** (**const** *Subpath* &*subpath*, **const** function<int64_t) int64_t > &*node_length*, *Subpath* &*rev_comp_out*Stores the reverse complement of a *Subpath* in another *Subpath*

note: this is not included in the header because reversing a subpath without going through the multipath alignment can break invariants related to the edge lists

Args: subpath subpath to reverse complement node_length a function that returns the length of a node sequence from its node ID rev_comp_out empty subpath to store reverse complement in (data will be overwritten if not empty)

void vg::**rev_comp_multipath_alignment**(**const** *MultipathAlignment* &*multipath_aln*, **const** function<int64_t) int64_t > &*node_length*, *MultipathAlignment* &*rev_comp_out*Stores the reverse complement of a *MultipathAlignment* in another *MultipathAlignment*

Args: multipath_aln multipath alignment to reverse complement node_length a function that returns the length of a node sequence from its node ID rev_comp_out empty multipath alignment to store reverse complement in (some data may be overwritten if not empty)

void vg::**rev_comp_multipath_alignment_in_place**(*MultipathAlignment* *&*multipath_aln*, **const** function<int64_t) int64_t > &*node_length*Stores the reverse complement of a *MultipathAlignment* in another *MultipathAlignment*

Args: multipath_aln multipath alignment to reverse complement in place node_length a function that returns the length of a node sequence from its node ID

void vg::**to_multipath_alignment**(**const** *Alignment* &*aln*, *MultipathAlignment* &*multipath_aln_out*)
Converts a *Alignment* into a Multipath alignment with one *Subpath* and stores it in an object

Args: aln alignment to convert multipath_aln empty multipath alignment to store converted alignment in (data may be be overwritten if not empty)

void vg::**transfer_read_metadata**(**const** *MultipathAlignment* &*from*, *MultipathAlignment* &*to*)
Copies metadata from an *MultipathAlignment* object and transfers it to another *MultipathAlignment*

Args: from copy metadata from this to into this

void vg::**transfer_read_metadata**(**const** *Alignment* &*from*, *MultipathAlignment* &*to*)
Copies metadata from an *Alignment* object and transfers it to a *MultipathAlignment*

Args: from copy metadata from this to into this

void vg::**transfer_read_metadata**(**const** *MultipathAlignment* &*from*, *Alignment* &*to*)
Copies metadata from an *MultipathAlignment* object and transfers it to a *Alignment*

Args: from copy metadata from this to into this

vector<vector<int64_t>> vg::**connected_components**(**const** *MultipathAlignment* &*multipath_aln*)
Returns a vector whose elements are vectors with the indexes of the Subpaths in each connected component

void vg::**extract_sub_multipath_alignment**(**const** *MultipathAlignment* &*multipath_aln*, **const** vector<int64_t> &*subpath_indexes*, *MultipathAlignment* &*sub_multipath_aln*)
Extract the *MultipathAlignment* consisting of the Subpaths with the given indexes into a new *MultipathAlignment* object

NodeSide vg::**node_start**(*id_t id*)
Produce the start *NodeSide* of a *Node*.

NodeSide vg::**node_end**(*id_t id*)
Produce the end *NodeSide* of a *Node*.

ostream &vg::**operator<<**(ostream &*out*, **const** *NodeSide* &*nodeside*)
Print a *NodeSide* to a stream.

ostream &vg::**operator<<**(ostream &*out*, **const** *NodeTraversal* &*nodetraversal*)
Print the given *NodeTraversal*.

Path &vg::**append_path**(*Path* &*a*, **const** *Path* &*b*)

int vg::**path_to_length**(**const** *Path* &*path*)

int vg::**path_from_length**(**const** *Path* &*path*)

int vg::**mapping_to_length**(**const** *Mapping* &*m*)

int vg::**mapping_from_length**(**const** *Mapping* &*m*)

int vg::**softclip_start**(**const** *Mapping* &*mapping*)

int vg::**softclip_end**(**const** *Mapping* &*mapping*)

Position vg::**first_path_position**(**const** *Path* &*path*)

Position vg::**last_path_position**(**const** *Path* &*path*)

int vg::**to_length**(**const** *Mapping* &*m*)

int vg::**from_length**(**const** *Mapping* &*m*)

Path &vg::**extend_path**(*Path* &*path1*, **const** *Path* &*path2*)

Path vg::**concat_paths**(**const** *Path* &*path1*, **const** *Path* &*path2*)

Path vg::**simplify**(**const** *Path* &*p*)
Simplify the path for addition as new material in the graph. Remove any mappings that are merely single deletions, merge adjacent edits of the same type, strip leading and trailing deletion edits on mappings, and make sure no mappings have missing positions.

Mapping vg::**concat_mappings**(**const** *Mapping* &*m*, **const** *Mapping* &*n*)

Mapping vg::**simplify**(**const** *Mapping* &*m*)
Merge adjacent edits of the same type, strip leading and trailing deletion edits (while updating positions if necessary), and makes sure position is actually set.

Mapping vg::**merge_adjacent_edits**(**const** *Mapping* &*m*)
Merge adjacent edits of the same type.

Path vg::**trim_hanging_ends**(**const** *Path* &*p*)

bool vg::**mapping_ends_in_deletion**(**const** *Mapping* &*m*)

bool vg::**mapping_starts_in_deletion**(**const** *Mapping* &*m*)

bool vg::**mapping_is_total_deletion**(**const** *Mapping* &*m*)

bool vg::**mapping_is_simple_match**(**const** *Mapping* &*m*)

bool vg::**path_is_simple_match**(**const** *Path* &*p*)

const string vg::**mapping_sequence**(**const** *Mapping* &*mp*, **const** string &*node_seq*)

const string vg::**mapping_sequence**(**const** *Mapping* &*mp*, **const** *Node* &*n*)

Mapping vg::**reverse_complement_mapping**(**const** *Mapping* &*m*, **const** function<int64_t) *id_t* > &*node_length*

void vg::**reverse_complement_mapping_in_place**(*Mapping* *m, **const** function<int64_t) *id_t* > &*node_length*

Path vg::**reverse_complement_path**(**const** *Path* &*path*, **const** function<int64_t) *id_t* > &*node_length*

void vg::**reverse_complement_path_in_place**(*Path* *path, **const** function<int64_t) *id_t* > &*node_length*

pair<*Mapping*, *Mapping*> vg::**cut_mapping**(**const** *Mapping* &*m*, **const** *Position* &*pos*)

pair<*Mapping*, *Mapping*> vg::**cut_mapping**(**const** *Mapping* &*m*, size_t *offset*)

pair<*Path*, *Path*> vg::**cut_path**(**const** *Path* &*path*, **const** *Position* &*pos*)

pair<*Path*, *Path*> vg::**cut_path**(**const** *Path* &*path*, size_t *offset*)

bool vg::**maps_to_node**(**const** *Path* &*p*, *id_t id*)

Position vg::**path_start**(**const** *Path* &*path*)

string vg::**path_to_string**(*Path p*)

Position vg::**path_end**(**const** *Path* &*path*)

bool vg::**adjacent_mappings**(**const** *Mapping* &*m1*, **const** *Mapping* &*m2*)

bool vg::**mapping_is_match**(**const** *Mapping* &*m*)

double vg::**divergence**(**const** *Mapping* &*m*)

double vg::**identity**(**const** *Path* &*path*)

void vg::**decompose**(**const** *Path* &*path*, map<*pos_t*, int> &*ref_positions*, map<*pos_t*, *Edit*> &*edits*)

double vg::**overlap**(**const** *Path* &*p1*, **const** *Path* &*p2*)

void vg::**translate_node_ids**(*Path* &*path*, **const** unordered_map<*id_t*, *id_t*> &*translator*)

void vg::**translate_oriented_node_ids**(*Path* &*path*, **const** unordered_map<*id_t*, pair<*id_t*, bool>> &*translator*)

pos_t vg::**initial_position**(**const** *Path* &*path*)

pos_t vg::**final_position**(**const** *Path* &*path*)

Path vg::**path_from_node_traversals**(**const** list<*NodeTraversal*> &*traversals*)

Path &vg::**increment_node_mapping_ids**(*Path* &*p*, *id_t inc*)

const *Paths* vg::**paths_from_graph**(*Graph* &*g*)

Path vg::**merge_adjacent_edits**(**const** *Path* &*m*)
Merge adjacent edits of the same type.

ostream &vg::**operator<<**(ostream &*os*, **const** *NodeDivider*::*NodeMap* &*nm*)

ostream &vg::**operator<<**(ostream &*os*, *NodeDivider*::*Entry entry*)

ostream &vg::**operator<<**(ostream &*os*, **const** *PileupAugmenter*::*NodeOffSide* &*no*)

StrandSupport vg::**minSup**(vector<*StrandSupport*> &*s*)

StrandSupport vg::**maxSup**(vector<*StrandSupport*> &*s*)

StrandSupport vg::**avgSup**(vector<*StrandSupport*> &*s*)

StrandSupport vg::**totalSup**(vector<*StrandSupport*> &*s*)

ostream &vg::**operator<<**(ostream &*os*, **const** *StrandSupport* &*sup*)

pos_t vg::**make_pos_t**(**const** *Position* &*pos*)
Convert a *Position* to a (much smaller) pos_t.

pos_t vg::**make_pos_t**(*id_t id*, bool *is_rev*, *off_t off*)
Create a pos_t from a *Node* ID, an orientation flag, and an offset.

pos_t vg::**make_pos_t**(gcsa::node_type *node*)
Create a pos_t from a gcsa node.

Position vg::**make_position**(**const** *pos_t* &*pos*)
Convert a pos_t to a *Position*.

Position vg::**make_position**(*id_t id*, bool *is_rev*, *off_t off*)
Create a *Position* from a *Node* ID, an orientation flag, and an offset.

Position vg::**make_position**(gcsa::node_type *node*)
Make a *Position* from a gcsa node.

bool vg::**is_empty**(**const** *pos_t* &*pos*)
Return true if a pos_t is unset.

id_t vg::**id**(**const** *pos_t* &*pos*)
Extract the id of the node a pos_t is on.

bool vg::**is_rev**(**const** *pos_t* &*pos*)
Return true if a pos_t is on the reverse strand of its node.

off_t vg::**offset**(**const** *pos_t* &*pos*)
Get the offset from a pos_t.

id_t &vg::**get_id**(*pos_t* &*pos*)
Get a reference to the *Node* ID of a pos_t.

bool &vg::**get_is_rev**(*pos_t* &*pos*)
Get a reference to the reverse flag of a pos_t.

off_t &vg::**get_offset**(*pos_t* &*pos*)
Get a reference to the offset field of a pos_t.

pos_t vg::**reverse**(**const** *pos_t* &*pos*, size_t *node_length*)
Reverse a pos_t and get a pos_t at the same base, going the other direction.

Position vg::**reverse**(**const** *Position* &*pos*, size_t *node_length*)
Reverse a *Position* and get a *Position* at the same base, going the orther direction.

ostream &vg::**operator<<**(ostream &*out*, **const** *pos_t* &*pos*)
Print a pos_t to a stream.

pair<int64_t, int64_t> vg::**min_oriented_distances**(**const** map<string, vector<pair<size_t, bool>>> &*path_offsets1*, **const** map<string, vector<pair<size_t, bool>>> &*path_offsets2*)
Find the min distance in the path offsets where the path orientation is the same and different.

void vg::**parse_bed_regions**(**const** string &*bed_path*, vector<*Region*> &*out_regions*)

void vg::**parse_region**(string &*region*, *Region* &*out_region*)

pos_t vg::**position_at**(*xg*::*XG* **xgidx*, **const** string &*path_name*, **const** size_t &*path_offset*, bool *is_reverse*)
We have a helper function to convert path positions and orientations to pos_t values.

We have a utility function for turning positions along paths, with orientations, into pos_ts. Remember that pos_t counts offset from the start of the reoriented node, while here we count offset from the beginning of the forward version of the path.

bool vg::**start_backward**(**const** Chain &*chain*)
Return true if the first snarl in the given chain is backward relative to the chain.

bool vg::**end_backward**(**const** Chain &*chain*)
Return true if the last snarl in the given chain is backward relative to the chain.

Visit vg::**get_start_of**(**const** Chain &*chain*)
Get the inward-facing start *Visit* for a chain.

Visit vg::**get_end_of**(**const** Chain &*chain*)
Get the outward-facing end *Visit* for a chain.

ChainIterator vg::**chain_begin**(**const** Chain &*chain*)
We define free functions for getting iterators forward and backward through chains.

ChainIterator vg::**chain_end**(**const** Chain &*chain*)

ChainIterator vg::**chain_rbegin**(**const** Chain &*chain*)

ChainIterator vg::**chain_rend**(**const** Chain &*chain*)

ChainIterator vg::**chain_rcbegin**(**const** Chain &*chain*)
We also define some reverse complement iterators, which go from right to left through the chains, but give us the reverse view. For ecample, if all the snarls are oriented forward in the chain, we will iterate through the snarls in reverse order, with each individual snarl also reversed.

ChainIterator vg::**chain_rcend**(**const** Chain &*chain*)

ChainIterator vg::**chain_begin_from**(**const** Chain &*chain*, **const** *Snarl* **start_snarl*, bool *snarl_orientation*)
We also define a function for getting the *ChainIterator* (forward or reverse complement) for a chain starting with a given snarl in the given inward orientation

ChainIterator vg::**chain_end_from**(**const** Chain &*chain*, **const** *Snarl* **start_snarl*, bool *snarl_orientation*)
And the end iterator for the chain (forward or reverse complement) viewed from a given snarl in the given inward orientation

bool vg::**operator==**(**const** *Visit* &*a*, **const** *Visit* &*b*)
Two Visits are equal if they represent the same traversal of the same *Node* or *Snarl*.

bool vg::**operator!=**(**const** *Visit* &*a*, **const** *Visit* &*b*)
Two Visits are unequal if they are not equal.

bool vg::**operator<**(**const** *Visit* &*a*, **const** *Visit* &*b*)
A *Visit* is less than another *Visit* if it represents a traversal of a smaller node, or it represents a traversal of a smaller snarl, or it represents a traversal of the same node or snarl forward instead of backward.

ostream &vg::**operator<<**(ostream &*out*, **const** *Visit* &*visit*)
A *Visit* can be printed.

bool vg::**operator==**(**const** *SnarlTraversal* &*a*, **const** *SnarlTraversal* &*b*)
Two SnarlTraversals are equal if their snarls are equal and they have the same number of visits and all their visits are equal.

bool vg::**operator!=**(**const** *SnarlTraversal* &*a*, **const** *SnarlTraversal* &*b*)
Two SnarlTraversals are unequal if they are not equal.

bool vg::**operator<**(**const** *SnarlTraversal* &*a*, **const** *SnarlTraversal* &*b*)
A SnalTraversal is less than another if it is a traversal of a smaller *Snarl*, or if its list of Visits has a smaller *Visit* first, or if its list of Visits is shorter.

bool vg::**operator==**(**const** *Snarl* &*a*, **const** *Snarl* &*b*)
Two Snarls are equal if their types are equal and their bounding Visits are equal and their parents are equal.

bool vg::**operator!=**(**const** *Snarl* &*a*, **const** *Snarl* &*b*)
Two Snarls are unequal if they are not equal.

bool vg::**operator<**(**const** *Snarl* &*a*, **const** *Snarl* &*b*)
A *Snarl* is less than another *Snarl* if its type is smaller, or its start *Visit* is smaller, or its end *Visit* is smaller, or its parent is smaller.

ostream &vg::**operator<<**(ostream &*out*, **const** *Snarl* &*snarl*)
A *Snarl* can be printed.

NodeTraversal vg::**to_node_traversal**(**const** *Visit* &*visit*, **const** *VG* &*graph*)
Converts a *Visit* to a *NodeTraversal*. Throws an exception if the *Visit* is of a *Snarl* instead of a *Node*

NodeTraversal vg::**to_rev_node_traversal**(**const** *Visit* &*visit*, **const** *VG* &*graph*)
Converts a *Visit* to a *NodeTraversal* in the opposite orientation. Throws an exception if the *Visit* is of a *Snarl* instead of a *Node*

NodeSide vg::**to_left_side**(**const** *Visit* &*visit*)
Converts a *Visit* to a node or snarl into a *NodeSide* for its left side.

NodeSide vg::**to_right_side**(**const** *Visit* &*visit*)
Converts a *Visit* to a node or snarl into a *NodeSide* for its right side.

Visit vg::**to_visit**(**const** *NodeTraversal* &*node_traversal*)
Converts a *NodeTraversal* to a *Visit*.

Visit vg::**to_visit**(**const** *Mapping* &*mapping*)
Converts a *Mapping* to a *Visit*. The mapping must represent a full node match.

Visit vg::**to_visit**(*id_t node_id*, bool *is_reverse*)
Make a *Visit* from a node ID and an orientation.

Visit vg::**to_visit**(**const** *Snarl* &*snarl*)
Make a *Visit* from a snarl to traverse.

Visit vg::**reverse**(**const** *Visit* &*visit*)
Get the reversed version of a visit.

Visit vg::**to_rev_visit**(**const** *NodeTraversal* &*node_traversal*)
Converts a *NodeTraversal* to a *Visit* in the opposite orientation.

Mapping vg::**to_mapping**(**const** *Visit* &*visit*, *std*::function<size_t) *id_t*
> *node_length*Converts a *Visit* to a *Mapping*. Throws an exception if the *Visit* is of a *Snarl* instead of a *Node*. Uses a function to get node length.

Mapping vg::**to_mapping**(**const** *Visit* &*visit*, *VG* &*vg*)
Converts a *Visit* to a *Mapping*. Throws an exception if the *Visit* is of a *Snarl* instead of a *Node*. Uses a graph to get node length.

void vg::**transfer_boundary_info**(**const** *Snarl* &*from*, *Snarl* &*to*)
Copies the boundary Visits from one *Snarl* into another.

NodeTraversal vg::**to_node_traversal**(**const** *Visit* &*visit*, *VG* &*graph*)

NodeTraversal vg::**to_rev_node_traversal**(**const** *Visit* &*visit*, *VG* &*graph*)

template <typename T>
string vg::**to_string_ss**(T *val*)

double vg::**strand_bias**(**const** *Support* &*support*)
Get the strand bias of a *Support*.

string vg::**char_to_string**(**const** char &*letter*)
Make a letter into a full string because apparently that's too fancy for the standard library.

void vg::**write_vcf_header**(ostream &*stream*, **const** vector<string> &*sample_names*, **const** vector<string> &*contig_names*, **const** vector<size_t> &*contig_sizes*, int *min_mad_for_filter*, int *max_dp_for_filter*, double *max_dp_multiple_for_filter*, double *max_local_dp_multiple_for_filter*, double *min_ad_log_likelihood_for_filter*)
Write a minimal VCF header for a file with the given samples, and the given contigs with the given lengths.

bool vg::**can_write_alleles**(vcflib::Variant &*variant*)
Return true if a variant may be output, or false if this variant is valid but the GATK might choke on it.

Mostly used to throw out variants with very long alleles, because GATK has an allele length limit. How alleles that really *are* 1 megabase deletions are to be specified to GATK is left as an exercise to the reader.

bool vg::**mapping_is_perfect_match**(**const** *Mapping* &*mapping*)
Return true if a mapping is a perfect match, and false if it isn't.

string vg::**get_pileup_line**(**const** map<int64_t, *NodePileup*> &*node_pileups*, **const** set<pair<int64_t, size_t>> &*refCrossreferences*, **const** set<pair<int64_t, size_t>> &*altCrossreferences*)
Given a collection of pileups by original node ID, and a set of original node id:offset cross-references in both ref and alt categories, produce a VCF comment line giving the pileup for each of those positions on those nodes. Includes a trailing newline if nonempty.

TODO: VCF comments aren't really a thing.

void vg::**trace_traversal**(**const** *SnarlTraversal* &*traversal*, **const** *Snarl* &*site*, function<void) size_t, *id_t*
> *handle_node*, function<voidsize_t, *NodeSide*, *NodeSide*> *handle_edge*, function<voidsize_t, *Snarl*>

*handle_child*Trace out the given traversal, handling nodes, child snarls, and edges associated with particular visit numbers.

tuple<*Support*, *Support*, size_t> vg::**get_traversal_support**(*SupportAugmentedGraph* &*augmented*, *SnarlManager* &*snarl_manager*, **const** *Snarl* &*site*, **const** *SnarlTraversal* &*traversal*, **const** *SnarlTraversal* **already_used* = nullptr)

Get the min support, total support, bp size (to divide total by for average support), and min likelihood for a traversal, optionally special-casing the material used by another traversal. Material used by another traversal only makes half its coverage available to this traversal.

bool vg::**is_match**(**const** *Translation* &*translation*)

char vg::**reverse_complement**(**const** char &*c*)

string vg::**reverse_complement**(**const** string &*seq*)

void vg::**reverse_complement_in_place**(string &*seq*)

bool vg::**is_all_n**(**const** string &*seq*)

Return True if the given string is entirely Ns of either case, and false otherwise.

int vg::**get_thread_count**(void)

std::vector<*std*::string> &vg::**split_delims**(**const** *std*::string &*s*, **const** *std*::string &*delims*, *std*::vector<*std*::string> &*elems*)

std::vector<*std*::string> vg::**split_delims**(**const** *std*::string &*s*, **const** *std*::string &*delims*)

const *std*::string vg::**sha1sum**(**const** *std*::string &*data*)

const *std*::string vg::**sha1head**(**const** *std*::string &*data*, size_t *head*)

string vg::**wrap_text**(**const** string &*str*, size_t *width*)

bool vg::**is_number**(**const** *std*::string &*s*)

bool vg::**allATGC**(**const** string &*s*)

string vg::**nonATGCNtoN**(**const** string &*s*)

string vg::**toUppercase**(**const** string &*s*)

string vg::**tmpfilename**(**const** string &*base*)

Create a temporary file starting with the given base name.

string vg::**find_temp_dir**()

Find the system temp directory using defaults and environment variables.

string vg::**tmpfilename**()

Create a temporary file in the appropriate system temporary directory.

string vg::**get_or_make_variant_id**(**const** vcflib::Variant &*variant*)

string vg::**make_variant_id**(**const** vcflib::Variant &*variant*)

double vg::**median**(*std*::vector<int> &*v*)

vector<size_t> vg::**range_vector**(size_t *begin*, size_t *end*)

void vg::**get_input_file**(int &*optind*, int *argc*, char ****argv*, function<void) istream&
> *callback*

string vg::**get_input_file_name**(int &*optind*, int *argc*, char ****argv*)

string vg::**get_output_file_name**(int &*optind*, int *argc*, char ****argv*)

void vg::**get_input_file**(**const** string &*file_name*, function<void) istream&
> *callback*

double vg::**phi**(double *x1*, double *x2*)

double vg::**normal_inverse_cdf**(double *p*)
Inverse CDF of a standard normal distribution. Must have 0 < quantile < 1.

void vg::**create_ref_allele**(vcflib::Variant &*variant*, **const** *std*::string &*allele*)
Create the reference allele for an empty vcflib Variant, since apaprently there's no method for that already. Must be called before any alt alleles are added.

int vg::**add_alt_allele**(vcflib::Variant &*variant*, **const** *std*::string &*allele*)
Add a new alt allele to a vcflib Variant, since apaprently there's no method for that already.

If that allele already exists in the variant, does not add it again.

Retuerns the allele number (0, 1, 2, etc.) corresponding to the given allele string in the given variant.

double vg::**slope**(**const** *std*::vector<double> &*x*, **const** *std*::vector<double> &*y*)

double vg::**fit_zipf**(**const** vector<double> &*y*)

size_t vg::**integer_power**(uint64_t *base*, uint64_t *exponent*)
Computes base^exponent in log(exponent) time.

size_t vg::**modular_exponent**(uint64_t *base*, uint64_t *exponent*, uint64_t *modulus*)
Computes base^exponent mod modulus in log(exponent) time without requiring more than 64 bits to represent exponentiated number

bool vg::**is_number**(**const** string &*s*)

double vg::**stdev**(**const** *std*::vector<double> &*v*)

template <typename T>
double vg::**stdev**(**const** T &*v*)

double vg::**add_log**(double *log_x*, double *log_y*)

double vg::**ln_to_log10**(double *ln*)
Convert a number ln to the same number log 10.

double vg::**log10_to_ln**(double *l10*)
Convert a number log 10 to the same number ln.

double vg::**prob_to_logprob**(double *prob*)

double vg::**logprob_to_prob**(double *logprob*)

double vg::**logprob_add**(double *logprob1*, double *logprob2*)

double vg::**logprob_invert**(double *logprob*)

double vg::**phred_to_prob**(int *phred*)

double vg::**prob_to_phred**(double *prob*)

double vg::**phred_to_logprob**(int *phred*)

double vg::**logprob_to_phred**(double *logprob*)

double vg::**logprob_geometric_mean**(double *lnprob1*, double *lnprob2*)

double vg::**phred_geometric_mean**(double *phred1*, double *phred2*)

template <**typename** T>
T vg::**normal_pdf**(T *x*, T *m*, T *s*)

template <**typename** T, **typename** V>
set<T> vg::**map_keys_to_set**(**const** map<T, V> &*m*)

template <**typename** T>
vector<T> vg::**pmax**(**const** *std*::vector<T> &*a*, **const** *std*::vector<T> &*b*)

template <**typename** T>
vector<T> vg::**vpmax**(**const** *std*::vector<*std*::vector<T>> &*vv*)

template <**typename** Collection>
Collection::value_type vg::**sum**(**const** Collection &*collection*)

Compute the sum of the values in a collection. Values must be default- constructable (like numbers are).

template <**typename** Collection>
Collection::value_type vg::**logprob_sum**(**const** Collection &*collection*)

Compute the sum of the values in a collection, where the values are log probabilities and the result is the log of the total probability. Items must be convertible to/from doubles for math.

template <template< class T, class A=std::allocator< T >> class *Container*, **typename** Input, **typename** Output>
Container<Output> vg::**map_over**(**const** Container<Input> &*in*, **const** *std*::function<Output(**const** Input&)> &*lambda*

We have a transforming map function that we can chain.

template <template< class T, class A=std::allocator< T >> class *Container*, **typename** Item>
Container<**const** Item *> vg::**pointerfy**(**const** Container<Item> &*in*)

We have a wrapper of that to turn a container reference into a container of pointers.

size_t vg::**integer_power**(size_t *x*, size_t *power*)

void vg::**genotype_svs**(*VG* **graph*, string *gamfile*, string *refpath*)

Takes a graph and two GAMs, one tumor and one normal Locates existing variation supported by the tumor and annotate it with a path Then overlay the normal sample Use a depthmap of snarltraversal transforms, one for tumor, one for normal which we can use to count the normal and tumor alleles void somatic_genotyper(VG* graph, string tumorgam, string normalgam);

Do smart augment, maintaining a depth map for tumor/normal perfect matches and then editing in all of the SV reads (after normalization) with a T/N_ prefix Then, get our Snarls count reads supporting each and genotype void somatic_caller_genotyper(VG* graph, string tumorgam, string normalgam);

void vg::**variant_recall**(*VG* **graph*, vcflib::VariantCallFile **vars*, FastaReference **ref_genome*, vector<FastaReference *> *insertions*, string *gamfile*, bool *isIndex*)

run with : vg genotype -L -V v.vcf -I i.fa -R ref.fa

static void vg::**triple_to_vg**(void **user_data*, raptor_statement **triple*)

Node vg::**xg_node**(*id_t id*, *xg*::*XG* **xgidx*)

vector<*Edge*> vg::**xg_edges_on_start**(*id_t id*, *xg*::*XG* **xgidx*)

vector<*Edge*> vg::**xg_edges_on_end**(*id_t id*, *xg*::*XG* **xgidx*)

string vg::**xg_node_sequence**(*id_t id*, *xg*::*XG* **xgidx*)

size_t vg::**xg_node_length**(*id_t id*, *xg*::*XG* **xgidx*)
Get the length of a *Node* from an *xg::XG* index, with cacheing of deserialized nodes.

int64_t vg::**xg_node_start**(*id_t id*, *xg*::*XG* **xgidx*)
Get the node start position in the sequence vector.

char vg::**xg_pos_char**(*pos_t pos*, *xg*::*XG* **xgidx*)
Get the character at a position in an *xg::XG* index, with cacheing of deserialized nodes.

map<*pos_t*, char> vg::**xg_next_pos_chars**(*pos_t pos*, *xg*::*XG* **xgidx*)
Get the characters at positions after the given position from an *xg::XG* index, with cacheing of deserialized nodes.

set<*pos_t*> vg::**xg_next_pos**(*pos_t pos*, bool *whole_node*, *xg*::*XG* **xgidx*)

int64_t vg::**xg_distance**(*pos_t pos1*, *pos_t pos2*, int64_t *maximum*, *xg*::*XG* **xgidx*)

set<*pos_t*> vg::**xg_positions_bp_from**(*pos_t pos*, int64_t *distance*, bool *rev*, *xg*::*XG* **xgidx*)

Variables

const char ***const** vg::**BAM_DNA_LOOKUP**

constexpr size_t vg::**spaced_primes**[62]

constexpr size_t vg::**primitive_roots_of_unity**[62]

const map<char, char> vg::**COMPLEMENTARY_NUCLEOTIDES**

const int8_t vg::**default_match**

const int8_t vg::**default_mismatch**

const int8_t vg::**default_gap_open**

const int8_t vg::**default_gap_extension**

const int8_t vg::**default_full_length_bonus**

const int8_t vg::**default_max_scaled_score**

const uint8_t vg::**default_max_qual_score**

const double vg::**default_gc_content**

const int vg::**MAX_ALLELE_LENGTH**

const double vg::**LOG_ZERO**

const char vg::**complement**[256]

const char *vg::**VG_VERSION_STRING**

namespace Enums

enum type vg::subcommand::**CommandCategory**

Defines what kind of command each subcommand is.

Values:

Some commands are part of the main build-graph, align, call variants pipeline.

Some subcommands are important parts of the toolkit/swiss army knife for working with graphs and data.

Some commands are less important but potentially useful widgets that let you do a thing you might need.

Some commands are useful really only for developers.

Functions

std::ostream &vg::subcommand::**operator<<**(*std*::ostream &*out*, **const** *CommandCategory* &*category*)

Define a way to print the titles of the different categories.

namespace Functions

int vg::unittest::**run_unit_tests**(int *argc*, char *******argv*)

Take the original argc and argv from a `vg unittest` command-line call and run the unit tests. We keep this in its own CPP/HPP to keep our unit test library from being a dependency of main.o and other real application code.

Passes the args along to the unit test system.

Returns exit code 0 on success, other codes on failure.

namespace Typedefs

typedef

typedef

typedef

Functions

id_t xg::**side_id**(**const** *side_t* &*side*)

bool xg::**side_is_end**(**const** *side_t* &*side*)

side_t xg::**make_side**(*id_t* *id*, bool *is_end*)

id_t xg::**trav_id**(**const** *trav_t* &*trav*)

bool xg::**trav_is_rev**(**const** *trav_t* &*trav*)

int32_t xg::**trav_rank**(**const** *trav_t* &*trav*)

trav_t xg::**make_trav**(*id_t* *id*, bool *is_rev*, int32_t *rank*)

int xg::**dna3bit**(char *c*)

char xg::**revdna3bit**(int *i*)

Mapping xg::**new_mapping**(**const** string &*name*, int64_t *id*, size_t *rank*, bool *is_reverse*)

void xg::**parse_region**(**const** string &*target*, string &*name*, int64_t &*start*, int64_t &*end*)

void xg::**to_text**(ostream &*out*, Graph &*graph*)

size_t xg::**serialize**(*XG*::*rank_select_int_vector* &*to_serialize*, ostream &*out*, *sdsl*::structure_tree_node **parent*, **const** *std*::string *name*)

void xg::**deserialize**(*XG*::*rank_select_int_vector* &*target*, istream &*in*)

bool xg::**edges_equivalent**(**const** Edge &*e1*, **const** Edge &*e2*)

bool xg::**relative_orientation**(**const** Edge &*e1*, **const** Edge &*e2*)

bool xg::**arrive_by_reverse**(**const** Edge &*e*, int64_t *node_id*, bool *node_is_reverse*)

bool xg::**depart_by_reverse**(**const** Edge &*e*, int64_t *node_id*, bool *node_is_reverse*)

Edge xg::**make_edge**(int64_t *from*, bool *from_start*, int64_t *to*, bool *to_end*)

char xg::**reverse_complement**(**const** char &*c*)

string xg::**reverse_complement**(**const** string &*seq*)

void xg::**extract_pos**(**const** string &*pos_str*, int64_t &*id*, bool &*is_rev*, size_t &*off*)

void xg::**extract_pos_substr**(**const** string &*pos_str*, int64_t &*id*, bool &*is_rev*, size_t &*off*, size_t &*len*)

file **alignment.cpp**
#include "alignment.hpp"#include "stream.hpp"#include <regex>

file **alignment.hpp**
#include <iostream>#include <functional>#include <zlib.h>#include "utility.hpp"#include "path.hpp"#include "position.hpp"#include "vg.pb.h"#include "xg.hpp"#include "htslib/hfile.h"#include "htslib/hts.h"#include "htslib/sam.h"#include "htslib/vcf.h"

file **banded_global_aligner.cpp**
#include "banded_global_aligner.hpp"#include "json2pb.h"

file **banded_global_aligner.hpp**
#include <stdio.h>#include <ctype.h>#include <iostream>#include <vector>#include <unordered_set>#include <unordered_map>#include <list>#include <exception>#include "vg.pb.h"

file **benchmark.cpp**
#include "benchmark.hpp"#include <vector>#include <iostream>#include <cassert>#include <numeric>#include <cmath>#include <iomanip>

file **benchmark.hpp**
#include <chrono>#include <functional>#include <iostream>#include <string> : implementations of benchmarking functions

Contains utilities for running (micro)benchmarks of vg code.

file **bin2ascii.h**
#include <string>#include <stdexcept>

Defines

VG_BIN2ASCII_H_INCLUDED

Functions

std::string **hex2bin**(**const** *std*::string &*s*)

std::string **bin2hex**(**const** *std*::string &*s*)

std::string **b64_encode**(**const** *std*::string &*s*)

std::string **b64_decode**(**const** *std*::string &*s*)

file **cached_position.cpp**
#include "cached_position.hpp"

file **cached_position.hpp**
#include "vg.pb.h"#include "types.hpp"#include "xg.hpp"#include "lru_cache.h"#include "utility.hpp"#include "json2pb.h"#include <gcsa/gcsa.h>#include <iostream> Functions for working with cached Positions and `pos_t`s.

file **cactus.cpp**
#include <unordered_set>#include "cactus.hpp"#include "vg.hpp"#include "algorithms/topological_sort.hpp"#include <unordered_map>#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checko "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/cached_position.hpp"#include "../vg.pb.h"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/hash_map.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/handle.hpp"#include "algorithms/weakly_connected_components.hpp"#include <vector>#include "algorithms/find_shortest_paths.hpp"#include "sonLib.h"#include "stCactusGraphs.h"

file **cactus.hpp**
#include <vector>#include <map>#include "types.hpp"#include "utility.hpp"#include "nodeside.hpp"#include "vg.hpp"#include "sonLib.h"#include "stCactusGraphs.h" *cactus.hpp*: Wrapper utility functions for for pinchesAndCacti

file **chunker.cpp**
#include <iostream>#include <unordered_set>#include "stream.hpp"#include "chunker.hpp"

file **chunker.hpp**
#include <iostream>#include <map>#include <chrono>#include <ctime>#include "lru_cache.h"#include "vg.hpp"#include "xg.hpp"#include "json2pb.h"#include "region.hpp"#include "index.hpp"

file **cluster.cpp**
#include <string>#include <algorithm>#include <utility>#include <cstring>#include "cluster.hpp"

file **cluster.hpp**
#include <gcsa/gcsa.h>#include <gcsa/lcp.h>#include "position.hpp"#include "gssw_aligner.hpp"#include "utility.hpp"#include "mem.hpp"#include "xg.hpp"#include <functional>#include <string>#include <vector>#include <map> Chaining and clustering tools to work with maximal exact matches.

file **colors.hpp**
#include <vector>#include <random>

file **constructor.cpp**
#include "vg.hpp"#include "constructor.hpp"#include <cstdlib>#include <set>#include <tuple>#include <list>#include <algorithm>#include <memory> *constructor.cpp*: contains implementations for vg construction functions.

file **constructor.hpp**
#include <vector>#include <set>#include <map>#include <cstdlib>#include <functional>#include <regex>#include "types.hpp"#include "progressive.hpp"#include "vg.pb.h"#include "Variant.h"#include "Fasta.h"#include "vcf_buffer.hpp"#include "name_mapper.hpp" *constructor.hpp*: defines a tool class used for constructing VG graphs from VCF files.

file **convert.hpp**
#include <sstream>

file **deconstructor.cpp**
#include "deconstructor.hpp"

file **deconstructor.hpp**
#include <vector>#include <string>#include <sstream>#include <ostream>#include "genotypekit.hpp"#include "path_index.hpp"#include "Variant.h"#include "path.hpp"#include "vg.hpp"#include "vg.pb.h"#include "Fasta.h" Deconstruct is getting rewritten. New functionality: -Detect superbubbles and bubbles -Fix command line interface. -harmonize on XG / raw graph (i.e. deprecate index) -Use unroll/DAGify if needed to avoid cycles

Much of this is taken from Brankovic's "Linear-Time Superbubble Identification Algorithm for Genome Assembly"

file **distributions.hpp**
#include <map>#include <cmath>#include "utility.hpp"

file **edit.cpp**
#include "edit.hpp"#include "utility.hpp"

file **edit.hpp**
#include "vg.pb.h"#include <utility>#include <iostream>#include "json2pb.h"

file **entropy.cpp**
#include "entropy.hpp"

file **entropy.hpp**
#include <iostream>#include <set>#include <vector>#include <string>#include <cmath>#include <map>

file **feature_set.cpp**
#include "feature_set.hpp"#include <sstream>

file **feature_set.hpp**
#include <string>#include <vector>#include <map>#include <iostream> Defines a FeatureSet class that can read and write BED files, and that can keep BED features up to date as paths are edited.

file **filter.cpp**
#include "filter.hpp"

file **filter.hpp**
#include <vector>#include <cstdlib>#include <iostream>#include <unordered_map>#include <sstream>#include <string>#include <math.h>#include "vg.hpp"#include "mapper.hpp"#include "xg.hpp"#include "vg.pb.h" Provides a way to filter Edits contained within Alignments. This can be used to clean out sequencing errors and to find high-quality candidates for variant calling.

file **flow_sort.cpp**
#include "vg.hpp"#include "stream.hpp"#include "gssw_aligner.hpp"#include "vg.pb.h"#include "flow_sort.hpp"#include <raptor2/raptor2.h>

file **flow_sort.hpp**
#include "vg.pb.h"

file **gamsorter.cpp**
#include "gamsorter.hpp"

Variables

struct *custom_pos_sort_key* **possortkey**

struct *custom_aln_sort_key* **alnsortkey**

file **gamsorter.hpp**
#include “vg.pb.h”#include “stream.hpp”#include <string>#include <queue>#include <sstream>#include <functional>#include <algorithm>#include <iostream>#include <set>#include <vector>#include <unordered_map>#include <tuple>#include “google/protobuf/stubs/common.h”#include “google/protobuf/io/zero_copy_stream.h”#include “google/protobuf/io/zero_copy_stream_impl.h”#include “google/protobuf/io/gzip_stream.h”#include “google/protobuf/io/coded_stream.h”

file **genome_state.cpp**
#include “genome_state.hpp”

file **genome_state.hpp**
#include “handle.hpp”#include “snarls.hpp”#include <vector>#include <unordered_map>#include <utility>#include <tuple> MCMC-friendly genome state representation.

file **genotypekit.cpp**
#include “genotypekit.hpp”#include “algorithms/topological_sort.hpp”#include “algorithms/is_directed_acyclic.hpp”#include “/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/handle.hpp”#include <unordered_set>#include <vector>#include “cactus.hpp”

file **genotypekit.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <cmath>#include <regex>#include <limits>#include <unordered_set>#include <unordered_map>#include <list>#include “vg.pb.h”#include “vg.hpp”#include “translator.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “types.hpp”#include “distributions.hpp”#include “snarls.hpp”#include “path_index.hpp”#include “sonLib.h”

file **genotyper.cpp**
#include <cstdint>#include “genotyper.hpp”#include “algorithms/topological_sort.hpp”

file **genotyper.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <numeric>#include <cmath>#include <limits>#include <unordered_set>#include <unordered_map>#include <regex>#include <vector>#include <list>#include “vg.pb.h”#include “vg.hpp”#include “translator.hpp”#include “deconstructor.hpp”#include “srpe.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “types.hpp”#include “genotypekit.hpp”#include “path_index.hpp”#include “index.hpp”#include “distributions.hpp”

file **graph.cpp**
#include “graph.hpp”

file **graph.hpp**
#include “vg.pb.h”#include “types.hpp”#include <set>#include <algorithm>

file **graph_synchronizer.cpp**
#include “graph_synchronizer.hpp”#include “algorithms/extract_connecting_graph.hpp”#include <unordered_map>#include “/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/position.hpp”#include “/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/cached_position.hpp”#include “/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/handle.hpp”#include ”../vg.pb.h”#include “/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/hash_map.hpp”#include <iterator>

file **graph_synchronizer.hpp**
#include “vg.hpp”#include “path_index.hpp”#include <thread>#include <mutex>#include <condition_variable> : define a GraphSynchronizer that can manage concurrent access and updates to a VG graph.

file **gssw_aligner.cpp**
#include “gssw_aligner.hpp”#include “json2pb.h”

Variables

const double **quality_scale_factor**

const double **exp_overflow_limit**

file **gssw_aligner.hpp**

#include <algorithm>#include <utility>#include <vector>#include <set>#include <string>#include <unordered_map>#include "gssw.h"#include "vg.pb.h"#include "vg.hpp"#include "Variant.h"#include "Fasta.h"#include "path.hpp"#include "utility.hpp"#include "banded_global_aligner.hpp"

file **handle.cpp**

#include "handle.hpp"#include "snarls.hpp" Implement handle graph utility methods.

file **handle.hpp**

#include "types.hpp"#include "vg.pb.h"#include <functional>#include <cstdint>#include <vector> Defines a handle type that can refer to oriented nodes of, and be used to traverse, any backing graph implementation. Not just an ID or a pos_t because XG (and maybe other implementations) provide more efficient local traversal mechanisms if you can skip ID lookups.

file **haplotype_extracter.cpp**

#include <iostream>#include "vg.hpp"#include "haplotype_extracter.hpp"#include "json2pb.h"#include "xg.hpp"

Functions

void **trace_haplotypes_and_paths**(*xg*::*XG* &*index*, *vg*::*id_t* *start_node*, int *extend_distance*, Graph &*out_graph*, map<string, int> &*out_thread_frequencies*, bool *expand_graph*)

void **output_haplotype_counts**(ostream &*annotation_ostream*, vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*)

Graph **output_graph_with_embedded_paths**(vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*)

void **output_graph_with_embedded_paths**(ostream &*subgraph_ostream*, vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*, bool *json*)

void **thread_to_graph_spanned**(*thread_t* &*t*, Graph &*g*, *xg*::*XG* &*index*)

void **add_thread_nodes_to_set**(*thread_t* &*t*, set<int64_t> &*nodes*)

void **add_thread_edges_to_set**(*thread_t* &*t*, set<pair<int, int>> &*edges*)

void **construct_graph_from_nodes_and_edges**(Graph &*g*, *xg*::*XG* &*index*, set<int64_t> &*nodes*, set<pair<int, int>> &*edges*)

Path **path_from_thread_t**(*thread_t* &*t*)

vector<pair<*thread_t*, int>> **list_haplotypes**(*xg*::*XG* &*index*, *xg*::*XG*::*ThreadMapping* *start_node*, int *extend_distance*)

file **haplotype_extracter.hpp**

#include <cstdio>#include <cstdlib>#include <iostream>#include <vector>#include "vg.pb.h"#include "xg.hpp"

Typedefs

using

Functions

void **trace_haplotypes_and_paths**(*xg*::*XG* &*index*, *vg*::*id_t* *start_node*, int *extend_distance*, Graph &*out_graph*, map<string, int> &*out_thread_frequencies*, bool *expand_graph* = true)

Path **path_from_thread_t**(*thread_t* &*t*)

vector<pair<*thread_t*, int>> **list_haplotypes**(*xg*::*XG* &*index*, *xg*::*XG*::*ThreadMapping* *start_node*, int *extend_distance*)

void **output_graph_with_embedded_paths**(ostream &*subgraph_ostream*, vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*, bool *json* = true)

Graph **output_graph_with_embedded_paths**(vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*)

void **output_haplotype_counts**(ostream &*annotation_ostream*, vector<pair<*thread_t*, int>> &*haplotype_list*, *xg*::*XG* &*index*)

void **thread_to_graph_spanned**(*thread_t* &*t*, Graph &*graph*, *xg*::*XG* &*index*)

void **add_thread_nodes_to_set**(*thread_t* &*t*, set<int64_t> &*nodes*)

void **add_thread_edges_to_set**(*thread_t* &*t*, set<pair<int, int>> &*edges*)

void **construct_graph_from_nodes_and_edges**(Graph &*g*, *xg*::*XG* &*index*, set<int64_t> &*nodes*, set<pair<int, int>> &*edges*)

file **hash_map.hpp**

#include "sparsehash/sparse_hash_map"#include "sparsehash/dense_hash_map"#include <tuple>

Defines

OVERLOAD_PAIR_HASH

USE_DENSE_HASH

file **hash_map_set.hpp**

#include "sparsehash/sparse_hash_map"#include "sparsehash/sparse_hash_set"

Defines

OVERLOAD_PAIR_HASH

file **homogenizer.cpp**

#include "homogenizer.hpp"

file **homogenizer.hpp**

#include <iostream>#include <vector>#include "vg.hpp"#include "translator.hpp"#include "filter.hpp"#include "mapper.hpp"#include "vg.pb.h"#include "types.hpp"

Defines

VG_HOMOGENIZEER

file **index.cpp**

#include "index.hpp"

Defines

S (x)

file **index.hpp**

#include <iostream>#include <exception>#include <sstream>#include <climits>#include "rocksdb/db.h"#include "rocksdb/env.h"#include "rocksdb/options.h"#include "rocksdb/write_batch.h"#include "rocksdb/memtablerep.h"#include "rocksdb/statistics.h"#include "rocksdb/cache.h"#include "rocksdb/slice_transform.h"#include "rocksdb/table.h"#include "rocksdb/filter_policy.h"#include "json2pb.h"#include "vg.hpp"#include "hash_map.hpp"

file **json2pb.cpp**

#include <errno.h>#include <jansson.h>#include <google/protobuf/message.h>#include <google/protobuf/descriptor.h>#include <json2pb.h>#include <stdexcept>#include <cstdio>#include "bin2ascii.h"

Defines

json_boolean (val)

_CONVERT (type, ctype, fmt, sfunc, afunc)

_SET_OR_ADD (sfunc, afunc, value)

_CONVERT (type, ctype, fmt, sfunc, afunc)

Functions

static json_t ***_pb2json** (**const** Message &*msg*)

static json_t ***_field2json** (**const** Message &*msg*, **const** FieldDescriptor **field*, size_t *index*)

static void **_json2pb** (Message &*msg*, json_t **root*)

static void **_json2field** (Message &*msg*, **const** FieldDescriptor **field*, json_t **jf*)

void **json2pb** (Message &*msg*, **const** char **buf*, size_t *size*)

void **json2pb** (Message &*msg*, FILE **fp*)

int **json_dump_std_string** (**const** char **buf*, size_t *size*, void **data*)

std::string **pb2json** (**const** Message &*msg*)

file **json2pb.h**

#include <string>#include <cstdio>#include <functional>#include <vector>#include <stream.hpp>#include <iostream>

Functions

void **json2pb** (*google*::*protobuf*::Message &*msg*, **const** char **buf*, size_t *size*)

void **json2pb** (*google*::*protobuf*::Message &*msg*, FILE **fp*)

std::string **pb2json** (**const** *google*::*protobuf*::Message &*msg*)

file **main.cpp**
#include <iostream>#include <fstream>#include <ctime>#include <cstdio>#include <getopt.h>#include <sys/stat.h>#include "google/protobuf/stubs/common.h"#include "version.hpp"#include "subcommand/subcommand.hpp"

Functions

void **vg_help** (char **argv)

int **main** (int *argc*, char *argv[])

file **mapper.cpp**
#include <unordered_set>#include "mapper.hpp"#include "algorithms/extract_containing_graph.hpp"#include <vector>#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/position.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/xg.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/vg.hpp"#include "../vg.pb.h"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/handle.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/hash_map.hpp"

file **mapper.hpp**
#include <iostream>#include <map>#include <chrono>#include <ctime>#include "omp.h"#include "vg.hpp"#include "xg.hpp"#include "index.hpp"#include <gcsa/gcsa.h>#include <gcsa/lcp.h>#include "alignment.hpp"#include "path.hpp"#include "position.hpp"#include "xg_position.hpp"#include "lru_cache.h"#include "json2pb.h"#include "entropy.hpp"#include "gssw_aligner.hpp"#include "mem.hpp"#include "cluster.hpp"#include "graph.hpp"

file **mem.cpp**
#include <string>#include <algorithm>#include <cstring>#include <sstream>#include "mem.hpp"

file **mem.hpp**
#include <gcsa/gcsa.h>#include <gcsa/lcp.h>#include "position.hpp"#include "utility.hpp"#include <functional>#include <string>#include <vector>#include <map> Maximal Exact Matches (MEMs) header

file **multipath_alignment.cpp**
#include "multipath_alignment.hpp"

file **multipath_alignment.hpp**
#include <stdio.h>#include <vector>#include <list>#include <unordered_map>#include <algorithm>#include "vg.pb.h"#include "path.hpp"#include "alignment.hpp"#include "utility.hpp"

file **multipath_mapper.cpp**
#include "multipath_mapper.hpp"#include "algorithms/extract_containing_graph.hpp"#include "algorithms/extract_connecting_graph.hpp"#include "algorithms/extract_extending_graph.hpp"#include <unordered_map>#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/position.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/cached_position.hpp"#include "../vg.pb.h"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/hash_map.hpp"#include "/home/docs/checkouts/readthedocs.org/user_builds/vg/checkouts/latest/src/handle.hpp"#include "algorithms/topological_sort.hpp"#include "algorithms/weakly_connected_components.hpp"

file **multipath_mapper.hpp**
#include "hash_map.hpp"#include "suffix_tree.hpp"#include "mapper.hpp"#include "gssw_aligner.hpp"#include "types.hpp"#include "multipath_alignment.hpp"#include "xg.hpp"#include "vg.pb.h"#include "position.hpp"#include "path.hpp"#include "edit.hpp"#include "snarls.hpp"

file **name_mapper.cpp**
#include "name_mapper.hpp" : contains implementations for the NameMapper to convert between VCF and graph/FASTA contig names.

file **name_mapper.hpp**
#include <map>#include <string> name_mapper.hpp: defines a class that maps back and forth from VCF-space contig names to graph or FASTA-space contig names.

file **nested_traversal_finder.cpp**
#include "nested_traversal_finder.hpp" : Contains implementation for a TraversalFinder that works on non-leaf Snarls and produces covering paths.

file **nested_traversal_finder.hpp**
#include "genotypekit.hpp" : Defines a TraversalFinder that produces covering paths, with an awareness of nested Snarls.

file **nodeside.hpp**
#include <ostream>#include <utility>#include "vg.pb.h"#include "types.hpp"#include "hash_map.hpp"

file **nodetraversal.hpp**
#include "vg.pb.h"#include "hash_map.hpp"

file **option.cpp**
#include <typeinfo>#include <cxxabi.h>#include <cassert>#include "option.hpp" : Definitions for our thing-doer-class-attached option system

file **option.hpp**
#include <string>#include <vector>#include <set>#include <map>#include <functional>#include <iostream>#include <sstream>#include <getopt.h> : Command-line options that can be attached to configurable thing-doer calsses (variant callers, mappers, etc.)

To use: make your configurable object inherit from Configurable, and make all your command-line-option fields into Option<whatever type>.

file **packer.cpp**
#include "packer.hpp"

file **packer.hpp**
#include <iostream>#include <map>#include <chrono>#include <ctime>#include "omp.h"#include "xg.hpp"#include "alignment.hpp"#include "path.hpp"#include "position.hpp"#include "json2pb.h"#include "graph.hpp"#include "gcsa/internal.h"#include "xg_position.hpp"#include "utility.hpp"

file **path.cpp**
#include "path.hpp"#include "stream.hpp"#include "region.hpp"

file **path.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <set>#include <list>#include <sstream>#include "json2pb.h"#include "vg.pb.h"#include "edit.hpp"#include "hash_map.hpp"#include "utility.hpp"#include "types.hpp"#include "position.hpp"#include "nodetraversal.hpp"

file **path_index.cpp**
#include "path_index.hpp"

file **path_index.hpp**
#include <map>#include <utility>#include <string>#include "vg.hpp"#include "xg.hpp" Provides an index for indexing an individual path for fast random access. Stores all the mappings uncompressed in memory.

Used for the reference path during VCF creation and interpretation.

file **phase_duplicator.cpp**
#include "phase_duplicator.hpp"

file **phase_duplicator.hpp**
#include <utility>#include <vector>#include <xg.hpp>#include "vg.pb.h"#include "types.hpp" This file contains the PhaseDuplicator, which duplicates and un-crosslinks complex regions of a graph using information about phased traversals.

file **phased_genome.cpp**
#include “phased_genome.hpp”

file **phased_genome.hpp**
#include <stdio.h>#include <cassert>#include <list>#include “vg.pb.h”#include “vg.hpp”#include “node-traversal.hpp”#include “genotypekit.hpp”#include “hash_map.hpp”#include “snarls.hpp”

file **pictographs.hpp**
#include <vector>#include <random>#include <functional>

file **pileup.cpp**
#include <cstdlib>#include <stdexcept>#include <regex>#include “json2pb.h”#include “pileup.hpp”#include “stream.hpp”

file **pileup.hpp**
#include <iostream>#include <algorithm>#include <functional>#include “vg.pb.h”#include “vg.hpp”#include “hash_map.hpp”#include “utility.hpp”

file **pileup_augmenter.cpp**
#include <cstdlib>#include <stdexcept>#include “json2pb.h”#include “pileup_augmenter.hpp”#include “stream.hpp”

file **pileup_augmenter.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <cmath>#include <limits>#include <unordered_set>#include <tuple>#include “vg.pb.h”#include “vg.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “pileup.hpp”#include “path_index.hpp”#include “genotypekit.hpp”#include “option.hpp”

file **position.cpp**
#include “position.hpp”

file **position.hpp**
#include “vg.pb.h”#include “types.hpp”#include “utility.hpp”#include “json2pb.h”#include <gcsa/gcsa.h>#include <iostream> Functions for working with Positions and `pos_t`s.

file **progressive.cpp**
#include “progressive.hpp”#include <iostream>

file **progressive.hpp**
#include <string>#include “progress_bar.hpp”

file **readfilter.cpp**
#include “readfilter.hpp”#include “IntervalTree.h”#include <fstream>#include <sstream>

file **readfilter.hpp**
#include <vector>#include <cstdlib>#include <iostream>#include <string>#include “vg.hpp”#include “xg.hpp”#include “vg.pb.h” Provides a way to filter and transform reads, implementing the bulk of the `vg filter` command.

file **region.cpp**
#include <iostream>#include <fstream>#include <cassert>#include “region.hpp”

file **region.hpp**
#include <string>#include <vector>#include <sstream>#include “xg.hpp”

file **sampler.cpp**
#include “sampler.hpp”

file **sampler.hpp**
#include <iostream>#include <map>#include <unordered_map>#include <vector>#include <sstream>#include <algorithm>#include <chrono>#include <ctime>#include “vg.hpp”#include

"xg.hpp"#include "alignment.hpp"#include "path.hpp"#include "position.hpp"#include "cached_position.hpp"#include "lru_cache.h"#include "json2pb.h"

file **simplifier.cpp**
#include "simplifier.hpp"

file **simplifier.hpp**
#include "progressive.hpp"#include "vg.hpp"#include "vg.pb.h"#include "genotypekit.hpp"#include "utility.hpp"#include "path.hpp"#include "feature_set.hpp"#include "path_index.hpp" Provides a class for simplifying graphs by popping bubbles.

file **snarls.cpp**
#include "snarls.hpp"#include "json2pb.h"

file **snarls.hpp**
#include <cstdint>#include <stdio.h>#include <unordered_map>#include <unordered_set>#include <fstream>#include <deque>#include "stream.hpp"#include "vg.hpp"#include "handle.hpp"#include "vg.pb.h"#include "hash_map.hpp"

file **srpe.cpp**
#include "srpe.hpp"

file **srpe.hpp**
#include <string>#include <cstdint>#include <Variant.h>#include "filter.hpp"#include "index.hpp"#include "path_index.hpp"#include "IntervalTree.h"#include "vg.pb.h"#include "fml.h"#include "vg.hpp"#include <gcsa/gcsa.h>#include "alignment.hpp"#include "genotypekit.hpp"

file **ssw_aligner.cpp**
#include "ssw_aligner.hpp"

file **ssw_aligner.hpp**
#include <vector>#include <set>#include <string>#include "ssw_cpp.h"#include "vg.pb.h"#include "path.hpp"

file **stream.hpp**
#include <cassert>#include <iostream>#include <istream>#include <fstream>#include <functional>#include <vector>#include <list>#include "google/protobuf/stubs/common.h"#include "google/protobuf/io/zero_copy_stream.h"#include "google/protobuf/io/zero_copy_stream_impl.h"#include "google/protobuf/io/gzip_stream.h"#include "google/protobuf/io/coded_stream.h"

file **add_main.cpp**
#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../variant_adder.hpp" Defines the "vg add" subcommand, which adds in variation from a VCF to an existing graph.

Functions

void **help_add**(char **argv*)

int **main_add**(int *argc*, char **argv*)

Variables

Subcommand vg_add("add","add variants from a VCF to a graph", main_add)

file **align_main.cpp**
#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "sub-

command.hpp"#include "../vg.hpp" Defines the "vg align" subcommand, which aligns a read against an entire graph.

Functions

void **help_align** (char **argv*)

int **main_align** (int *argc*, char **argv*)

Variables

Subcommand vg_align("align","local alignment", main_align)

file **annotate_main.cpp**

#include "subcommand.hpp"#include "../vg.hpp"#include "../utility.hpp"#include "../mapper.hpp"#include "../stream.hpp"#include "../alignment.hpp"#include <unistd.h>#include <getopt.h>

Functions

void **help_annotate** (char **argv*)

int **main_annotate** (int *argc*, char **argv*)

Variables

Subcommand vg_annotate("annotate","annotate alignments with graphs and graphs with ali

file **augment_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../option.hpp"#include "../vg.hpp"#include "../pileup_augmenter.hpp" Defines the "vg augment" subcommand, which augments a graph using a GAM as a first step before computing sites and calling variants.

It presently only supports augmentation via pileup (as originally used by vg call).

vg mod -i is an alternative, but as currently implemented it is too expensive. Ideally, a more heuristic/fast implementation would get added as an option here.

The new vg count structure, provided edge/edit/strand/quality is supported, may be an interesting replacement for the current protobuf pileups which ought to go, one way or the other.

Functions

Pileups ***compute_pileups** (VG **graph*, **const** string &*gam_file_name*, int *thread_count*, int *min_quality*, int *max_mismatches*, int *window_size*, int *max_depth*, bool *use_mapq*, bool *show_progress*)

void **augment_with_pileups** (PileupAugmenter &*augmenter*, Pileups &*pileups*, bool *expect_subgraph*, bool *show_progress*)

void **add_trivial_edits** (VG &*graph*)

void **help_augment** (char **argv*, ConfigurableParser &*parser*)

int **main_augment** (int *argc*, char **argv*)

Variables

Subcommand vg_augment("augment","augment a graph from an alignment", PIPELINE, 5, main_

file **benchmark_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../benchmark.hpp"#include "../version.hpp"#include "../vg.hpp"#include "../xg.hpp"#include "../algorithms/extract_connecting_graph.hpp"#include "../algorithms/topological_sort.hpp"#include "../algorithms/weakly_connected_components.hpp" Defines the "vg benchmark" subcommand, which runs and reports on microbenchmarks.

Functions

void **help_benchmark** (char **argv*)

int **main_benchmark** (int *argc*, char **argv*)

Variables

Subcommand vg_benchmark("benchmark","run and report on performance benchmarks", DEVELO

file **bugs_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <stdlib.h>#include <iostream>#include "subcommand.hpp" Defines the "vg bugs" subcommand, which shows and opens Github issues.

Functions

void **help_bugs** (char **argv*)

int **main_bugs** (int *argc*, char **argv*)

Variables

Subcommand vg_bugs("bugs","show or create bugs", DEVELOPMENT, main_bugs)

file **call_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../option.hpp"#include "../vg.hpp"#include "../support_caller.hpp" Defines the "vg call" subcommand, which calls variation from an augmented graph

Functions

void **help_call** (char **argv*, ConfigurableParser &*parser*)

int **main_call** (int *argc*, char **argv*)

Variables

Subcommand vg_call("call","call variants on an augmented graph", PIPELINE, 5, main_cal

file **chunk_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <string>#include <vector>#include <regex>#include "subcommand.hpp"#include "../vg.hpp"#include "../stream.hpp"#include "../utility.hpp"#include "../chunker.hpp"#include "../region.hpp"#include "../haplotype_extracter.hpp"

Functions

void **help_chunk** (char **argv)

int **main_chunk** (int *argc*, char **argv)

Variables

Subcommand vg_chunk("chunk","split graph or alignment into chunks", main_chunk)

file **circularize_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp" Defines the "vg circularize" subcommand

Functions

void **help_circularize** (char **argv)

int **main_circularize** (int *argc*, char **argv)

Variables

Subcommand vg_circularize("circularize","circularize a path within a graph", main_circ

file **compare_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../index.hpp" Defines the "vg compare" subcommand

Functions

void **help_compare** (char **argv)

int **main_compare** (int *argc*, char **argv)

Variables

Subcommand vg_compare("compare","compare the kmer space of two graphs", main_compare)

file **concat_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp" Defines the "vg concat" subcommand

Functions

void **help_concat** (char **argv)

int **main_concat** (int *argc*, char **argv)

Variables

Subcommand vg_concat("concat","concatenate graphs tail-to-head", main_concat)

file **construct_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <memory>#include "subcommand.hpp"#include "../stream.hpp"#include "../constructor.hpp"#include "../region.hpp"

Functions

void **help_construct** (char **argv)

int **main_construct** (int *argc*, char **argv)

Variables

Subcommand vg_construct("construct","graph construction", PIPELINE, 1, main_construct)

file **deconstruct_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../deconstructor.hpp" Defines the "vg deconstruct" subcommand, which turns graphs back into VCFs.

Functions

void **help_deconstruct** (char **argv)

int **main_deconstruct** (int *argc*, char **argv)

Variables

Subcommand vg_deconstruct("deconstruct","convert a graph into VCF relative to a refere

file **explode_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include "subcommand.hpp"#include "../vg.hpp"#include "../stream.hpp"#include "../utility.hpp" : break a graph into connected components

Functions

void **help_explode** (char **argv)

int **main_explode** (int *argc*, char **argv)

Variables

Subcommand vg_explode("explode","split graph into connected components", main_explode)

file **filter_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../readfilter.hpp" Defines the "vg filter" subcommand, which filters alignments.

Functions

void **help_filter**(char **argv*)

int **main_filter**(int *argc*, char **argv*)

Variables

Subcommand vg_vectorize("filter","filter reads", main_filter)

file **find_main.cpp**

#include "subcommand.hpp"#include "../vg.hpp"#include "../utility.hpp"#include "../mapper.hpp"#include "../stream.hpp"#include <unistd.h>#include <getopt.h>

Functions

void **help_find**(char **argv*)

int **main_find**(int *argc*, char **argv*)

Variables

Subcommand vg_msga("find","use an index to find nodes, edges, kmers, paths, or positio

file **gamsort_main.cpp**

#include "gamsorter.hpp"#include "stream.hpp"#include <getopt.h>#include "subcommand.hpp"#include "index.hpp"

Functions

void **help_gamsort**(char **argv*)

int **main_gamsort**(int *argc*, char **argv*)

Variables

Subcommand vg_gamsort("gamsort","Perform naive sorts and indexing on a GAM file.", mai

file **genotype_main.cpp**

#include <getopt.h>#include "subcommand.hpp"#include "index.hpp"#include "stream.hpp"#include "genotyper.hpp"#include "genotypekit.hpp"#include "variant_recall.hpp"

Functions

void **help_genotype** (char **argv*)

int **main_genotype** (int *argc*, char **argv*)

Variables

Subcommand vg_genotype("genotype","Genotype (or type) graphs, GAMS, and VCFs.", main_g

file **help_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <stdlib.h>#include <iostream>#include "subcommand.hpp"#include "../version.hpp" Defines the "vg help" subcommand, which describes subcommands.

Functions

void **help_help** (char **argv*)

int **main_help** (int *argc*, char **argv*)

Variables

Subcommand vg_help("help","show all subcommands", PIPELINE, main_help)

file **ids_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../vg_set.hpp"#include "../algorithms/topological_sort.hpp" Defines the "vg ids" subcommand, which modifies node IDs.

Functions

void **help_ids** (char **argv*)

int **main_ids** (int *argc*, char **argv*)

Variables

Subcommand vg_ids("ids","manipulate node ids", TOOLKIT, main_ids)

file **index_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <string>#include <vector>#include <regex>#include "subcommand.hpp"#include "../vg.hpp"#include "../index.hpp"#include "../stream.hpp"#include "../vg_set.hpp"#include "../utility.hpp"#include "../path_index.hpp"#include <gcsa/gcsa.h>#include <gcsa/algorithms.h>#include <gbwt/dynamic_gbwt.h>

Functions

void **help_index** (char **argv*)

xg::*XG*::*ThreadMapping* **gbwt_to_thread_mapping** (gbwt::node_type *node*)

gbwt::node_type **mapping_to_gbwt** (**const** Mapping &*mapping*)

gbwt::node_type **node_side_to_gbwt** (**const** NodeSide &*side*)

int **main_index** (int *argc*, char **argv*)

Variables

Subcommand vg_construct("index","index graphs or alignments for random access or mappi

file **join_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp" Defines the "vg join" subcommand, which attaches graphs together.

Functions

void **help_join** (char **argv*)

int **main_join** (int *argc*, char **argv*)

Variables

Subcommand vg_join("join","combine graphs via a new head", main_join)

file **kmers_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../vg_set.hpp" Defines the "vg kmers" subcommand, which generates the kmers in a graph.

Functions

void **help_kmers** (char **argv*)

int **main_kmers** (int *argc*, char **argv*)

Variables

Subcommand vg_kmers("kmers","enumerate kmers of the graph", main_kmers)

file **locify_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../index.hpp"#include "../convert.hpp" Defines the "vg locify" subcommand

Functions

void **help_locify** (char **argv*)

int **main_locify** (int *argc*, char **argv*)

Variables

Subcommand vg_locify("locify","find loci", main_locify)

file **map_main.cpp**

#include "subcommand.hpp"#include "../vg.hpp"#include "../utility.hpp"#include "../mapper.hpp"#include "../stream.hpp"#include <unistd.h>#include <getopt.h>

Functions

void **help_map**(char **argv*)

int **main_map**(int *argc*, char **argv*)

Variables

Subcommand vg_map("map","MEM-based read alignment", PIPELINE, 3, main_map)

file **mod_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <string>#include <vector>#include <regex>#include "subcommand.hpp"#include "../vg.hpp"#include "../cactus.hpp"#include "../stream.hpp"#include "../utility.hpp"#include "../algorithms/topological_sort.hpp"

Functions

void **help_mod**(char **argv*)

int **main_mod**(int *argc*, char **argv*)

Variables

Subcommand vg_mod("mod","filter, transform, and edit the graph", TOOLKIT, main_mod)

file **mpmap_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include "subcommand.hpp"#include "../multipath_mapper.hpp"#include "../path.hpp" : multipath mapping of reads to a graph

Functions

void **help_mpmap**(char **argv*)

int **main_mpmap**(int *argc*, char **argv*)

Variables

Subcommand vg_mpmap("mpmap","multipath alignments of reads to a graph", main_mpmap)

file **msga_main.cpp**

#include "subcommand.hpp"#include "../vg.hpp"#include "../utility.hpp"#include "../mapper.hpp"#include "../stream.hpp"#include "../algorithms/topological_sort.hpp"#include <unistd.h>#include <getopt.h>

Functions

void **help_msga** (char **argv*)

int **main_msga** (int *argc*, char **argv*)

Variables

Subcommand vg_msga("msga","multiple sequence graph alignment", main_msga)

file **pack_main.cpp**

#include "subcommand.hpp"#include "../vg.hpp"#include "../utility.hpp"#include "../packer.hpp"#include "../stream.hpp"#include <unistd.h>#include <getopt.h>

Functions

void **help_pack** (char **argv*)

int **main_pack** (int *argc*, char **argv*)

Variables

Subcommand vg_pack("pack","convert alignments to a compact coverage, edit, and path in

file **paths_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp" Defines the "vg paths" subcommand, which reads paths in the graph.

Functions

void **help_paths** (char **argv*)

int **main_paths** (int *argc*, char **argv*)

Variables

Subcommand vg_paths("paths","traverse paths in the graph", main_paths)

file **sift_main.cpp**

#include <iostream>#include <fstream>#include <ctime>#include <cstdio>#include <getopt.h>#include <functional>#include <omp.h>#include "subcommand.hpp"#include <gcsa/gcsa.h>#include "stream.hpp"#include "json2pb.h"#include "vg.hpp"#include "vg.pb.h"#include "filter.hpp"#include "alignment.hpp"

Functions

void **help_sift** (char **argv*)

int **main_sift** (int *argc*, char **argv*)

Variables

Subcommand vg_sift("sift","Filter Alignments by various metrics related to variant cal

file **sim_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../mapper.hpp"#include "../sampler.hpp" Defines the "vg sim" subcommand, which generates potential reads from a graph.

Functions

void **help_sim**(char **argv*)

int **main_sim**(int *argc*, char **argv*)

Variables

Subcommand vg_sim("sim","simulate reads from a graph", TOOLKIT, main_sim)

file **simplify_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../simplifier.hpp"

Functions

void **help_simplify**(char **argv*)

int **main_simplify**(int *argc*, char **argv*)

Variables

Subcommand vg_simplify("simplify","graph simplification", main_simplify)

file **snarls_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include <regex>#include "subcommand.hpp"#include "../vg.hpp"#include "vg.pb.h"#include "../genotypekit.hpp"

Functions

void **help_snarl**(char **argv*)

int **main_snarl**(int *argc*, char **argv*)

Variables

Subcommand vg_snarl("snarls","compute snarls and their traversals", main_snarl)

file **sort_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../flow_sort.hpp" Defines the "vg sort" subcommand, which sorts graph nodes.

Functions

void **help_sort** (char **argv*)

int **main_sort** (int *argc*, char **argv*[])

Variables

Subcommand vg_sort("sort","sort variant graph using max flow algorithm or Eades fast h

file **srpe_main.cpp**

#include <iostream>#include <vector>#include <getopt.h>#include <functional>#include <regex>#include "subcommand.hpp"#include "srpe.hpp"#include "stream.hpp"#include "index.hpp"#include "position.hpp"#include "vg.pb.h"#include "path.hpp"#include "genotypekit.hpp"#include "genotyper.hpp"#include "path_index.hpp"#include "vg.hpp"#include <math.h>#include "filter.hpp"#include "utility.hpp"#include "Variant.h"#include "translator.hpp"#include "Fasta.h"#include "IntervalTree.h"

Functions

void **help_srpe** (char **argv*)

int **main_srpe** (int *argc*, char **argv*)

Variables

Subcommand vg_srpe("srpe","graph-external SV detection", main_srpe)

file **stats_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../distributions.hpp"#include "../genotypekit.hpp" Defines the "vg stats" subcommand, which evaluates graphs and alignments.

Functions

void **help_stats** (char **argv*)

int **main_stats** (int *argc*, char **argv*)

Variables

Subcommand vg_stats("stats","metrics describing graph properties", TOOLKIT, main_stats

file **subcommand.cpp**

#include "subcommand.hpp"#include <algorithm>#include <utility>#include <vector>#include <limits>

file **subcommand.hpp**

#include <map>#include <functional>#include <string>#include <iostream> *subcommand.hpp*: defines a system for registering subcommands of the vg command (vg construct, vg view, etc.) at compile time. Replaces the system of defining two functions and a giant run of if statements in *main.cpp*.

main.cpp does *not* need to include any subcommand headers!

Subcommands are created as static global objects in their own compilation units, which have to be explicitly linked into the binary (they won't be pulled out of a library if nothing references their symbols).

Subcommands are responsible for printing their own help; we can do "vg help" and print all the subcommands that exist (via a help subcommand), but we can't do "vg help subcommand" and have that be equivalent to "vg subcommand –help" (because the help subcommand doesn't know how to get help info on the others).

We have a subcommand importance/category system, so we can tell people about just the main pipeline and keep the subcommands they don't want out of their brains and off their screen.

Subcommands get passed all of argv, so they have to skip past their names when parsing arguments.

To make a subcommand, do something like this in a cpp file in this "subcommand" directory:

```
#include "subcommand.hpp"
using namespace vg::subcommand;

int main_frobnicate(int argc, char** argv) {
    return 0;
}

static Subcommand vg_frobnicate("frobnicate", "frobnicate nodes and edges",
    main_frobnicate);
```

file **surject_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <string>#include <vector>#include <set>#include "subcommand.hpp"#include "../vg.hpp"#include "../stream.hpp"#include "../utility.hpp"#include "../mapper.hpp"

Functions

void **help_surject** (char **argv)

int **main_surject** (int *argc*, char **argv)

Variables

Subcommand vg_surject("surject","map alignments onto specific paths", main_surject)

file **test_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../unittest/driver.hpp" Defines the "vg test" subcommand, which runs unit tests.

Functions

int **main_test** (int *argc*, char **argv)

Variables

Subcommand vg_test("test","run unit tests", DEVELOPMENT, main_test)

file **trace_main.cpp**

#include <getopt.h>#include <string>#include <vector>#include "subcommand.hpp"#include "../vg.hpp"#include "../haplotype_extracter.hpp"

Typedefs

using

Functions

void **help_trace**(char **argv*)

int **main_trace**(int *argc*, char **argv*)

Variables

Subcommand vg_trace("trace","trace haplotypes", main_trace)

file **translate_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp"#include "../translator.hpp" Defines the "vg translate" subcommand

Functions

void **help_translate**(char **argv*)

int **main_translate**(int *argc*, char **argv*)

Variables

Subcommand vg_version("translate","project alignments and paths through a graph transl

file **validate_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../vg.hpp" Defines the "vg validate" subcommand

Functions

void **help_validate**(char **argv*)

int **main_validate**(int *argc*, char **argv*)

Variables

Subcommand vg_validate("validate","validate the semantics of a graph", DEVELOPMENT, ma

file **vectorize_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../vg.hpp"#include "../vectorizer.hpp"#include "../mapper.hpp" Defines the "vg vectorize" subcommand, which turns graphs into ML vectors.

Functions

void **help_vectorize** (char **argv)

int **main_vectorize** (int *argc*, char **argv)

Variables

Subcommand vg_vectorize("vectorize","transform alignments to simple ML-compatible vect

file **version_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <iostream>#include "subcommand.hpp"#include "../version.hpp" Defines the "vg version" subcommand, which evaluates graphs and alignments.

Functions

void **help_version** (char **argv)

int **main_version** (int *argc*, char **argv)

Variables

Subcommand vg_version("version","version information", DEVELOPMENT, main_version)

file **view_main.cpp**

#include <omp.h>#include <unistd.h>#include <getopt.h>#include <list>#include <fstream>#include "subcommand.hpp"#include "../multipath_alignment.hpp"#include "../vg.hpp" Defines the "vg view" subcommand, which converts formats.

Functions

void **help_view** (char **argv)

int **main_view** (int *argc*, char **argv)

Variables

Subcommand vg_view("view","format conversions for graphs and alignments", TOOLKIT, mai

file **xg_main.cpp**

#include <omp.h>#include <unistd.h>#include <iostream>#include <fstream>#include <getopt.h>#include "subcommand.hpp"#include "sdsl/bit_vectors.hpp"#include "../version.hpp"#include "../stream.hpp"#include "../cpp/vg.pb.h"#include "../xg.hpp"

Functions

void **help_xg** (char **argv)

int **main_xg** (int *argc*, char **argv)

Variables

Subcommand vg_xg("xg","manipulate xg files", main_xg)

file **suffix_tree.cpp**
#include “suffix_tree.hpp”

file **suffix_tree.hpp**
#include <stdio.h>#include <unordered_map>#include <list>#include <vector>#include <sstream>#include <iostream>

file **support_caller.cpp**
#include <iostream>#include <fstream>#include <sstream>#include <regex>#include <set>#include <unordered_set>#include <utility>#include <algorithm>#include <getopt.h>#include “vg.hpp”#include “index.hpp”#include “Variant.h”#include “genotypekit.hpp”#include “snarls.hpp”#include “path_index.hpp”#include “support_caller.hpp”#include “stream.hpp”#include “nested_traversal_finder.hpp”

file **support_caller.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <cmath>#include <limits>#include <unordered_set>#include <tuple>#include “vg.pb.h”#include “vg.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “pileup.hpp”#include “path_index.hpp”#include “genotypekit.hpp”#include “option.hpp”

file **swap_remove.hpp**
#include <vector>#include <algorithm>

Functions

template <typename T>
bool **swap_remove** (*std*::vector<T> &*v*, **const** T &*e*)

file **translator.cpp**
#include “translator.hpp”#include “stream.hpp”

file **translator.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <set>#include <vector>#include <list>#include “vg.pb.h”#include “vg.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “types.hpp” Defines the Translator, which maps paths on an augmented graph into the base graph defined by a set of Translations from the augmented graph

file **types.hpp**
#include <tuple> Contains typedefs for basic types useful for talking about graphs.

file **driver.cpp**
#include “driver.hpp”#include “catch.hpp”#include <sstream>#include <algorithm>#include <string>#include <vector>#include <cstddef>#include <iomanip>#include <limits>#include <stdint.h>#include <iterator>#include <stdlib.h>#include <cmath>#include <set>#include <stdexcept>#include <iosfwd>#include <streambuf>#include <ostream>#include <fstream>#include <memory>#include <cctype>#include <map>#include <ctime>#include <assert.h>#include <signal.h>#include <cstdio>#include <iostream>#include <cerrno>#include <unistd.h>#include <sys/time.h>#include <cstring>#include <cfloat>

Defines

CATCH_CONFIG_RUNNER

file **driver.hpp**

file **utility.cpp**
#include “utility.hpp”

file **utility.hpp**
#include <string>#include <vector>#include <sstream>#include <omp.h>#include <cstring>#include <algorithm>#include <numeric>#include <cmath>#include <unordered_set>#include <unistd.h>#include “vg.pb.h”#include “sha1.hpp”#include “Variant.h”

file **variant_adder.cpp**
#include “variant_adder.hpp”#include “mapper.hpp”

file **variant_adder.hpp**
#include “vcf_buffer.hpp”#include “path_index.hpp”#include “vg.hpp”#include “progressive.hpp”#include “name_mapper.hpp”#include “graph_synchronizer.hpp” *variant_adder.hpp*: defines a tool class used for adding variants from VCF files into existing graphs.

file **variant_recall.cpp**
#include <cstdint>#include “variant_recall.hpp”#include “algorithms/topological_sort.hpp”

file **variant_recall.hpp**
#include <iostream>#include <algorithm>#include <functional>#include <numeric>#include <cmath>#include <limits>#include <unordered_set>#include <unordered_map>#include <regex>#include <vector>#include <list>#include “vg.pb.h”#include “vg.hpp”#include “translator.hpp”#include “deconstructor.hpp”#include “srpe.hpp”#include “hash_map.hpp”#include “utility.hpp”#include “types.hpp”#include “genotypekit.hpp”#include “path_index.hpp”#include “index.hpp”#include “distributions.hpp”

file **vcf_buffer.cpp**
#include “vcf_buffer.hpp” *constructor.cpp*: contains implementations for vg construction functions.

file **vcf_buffer.hpp**
#include <list>#include <tuple>#include “Variant.h” *vcf_buffer.hpp*: defines a buffer for reading through VCF files with look-ahead.

file **vectorizer.cpp**
#include “vectorizer.hpp”

file **vectorizer.hpp**
#include <iostream>#include <sstream>#include “sdsl/bit_vectors.hpp”#include <vector>#include <unordered_map>#include “vg.hpp”#include “xg.hpp”#include “vg.pb.h”

file **version.cpp**
#include “version.hpp”#include “vg_git_version.hpp”

Defines

VG_GIT_VERSION

file **version.hpp**

file **vg.cpp**
#include “vg.hpp”#include “stream.hpp”#include “gssw_aligner.hpp”#include “genotypekit.hpp”#include “algorithms/topological_sort.hpp”#include <raptor2/raptor2.h>#include <stPinchGraphs.h>

file **vg.hpp**
#include <vector>#include <set>#include <string>#include <deque>#include <list>#include <array>#include <omp.h>#include <unistd.h>#include <limits.h>#include <algorithm>#include <random>#include “gssw.h”#include <gcsa/gcsa.h>#include <gcsa/lcp.h>#include “gssw_aligner.hpp”#include “ssw_aligner.hpp”#include “region.hpp”#include “path.hpp”#include “utility.hpp”#include

“alignment.hpp”#include “vg.pb.h”#include “hash_map.hpp”#include “progressive.hpp”#include “lru_cache.h”#include “Variant.h”#include “Fasta.h”#include “swap_remove.hpp”#include “pictographs.hpp”#include “colors.hpp”#include “types.hpp”#include “gfakluge.hpp”#include “nodetraversal.hpp”#include “nodeside.hpp”#include “handle.hpp”

file **vg.proto**

Variables

syntax

file **vg_set.cpp**
#include “vg_set.hpp”#include “stream.hpp”

file **vg_set.hpp**
#include <set>#include <regex>#include <stdlib.h>#include <gcsa/gcsa.h>#include “vg.hpp”#include “index.hpp”#include “xg.hpp”

file **xg.cpp**
#include “xg.hpp”#include “stream.hpp”#include <bitset>#include <arpa/inet.h>

file **xg.hpp**
#include <iostream>#include <fstream>#include <map>#include <queue>#include <omp.h>#include <unordered_map>#include <unordered_set>#include “cpp/vg.pb.h”#include “sdsl/bit_vectors.hpp”#include “sdsl/enc_vector.hpp”#include “sdsl/dac_vector.hpp”#include “sdsl/vlc_vector.hpp”#include “sdsl/wavelet_trees.hpp”#include “sdsl/csa_wt.hpp”#include “sdsl/suffix_arrays.hpp”#include “hash_map_set.hpp”#include “position.hpp”#include “graph.hpp”#include “path.hpp”#include “handle.hpp”

Defines

MODE_DYNAMIC

MODE_SDSL

GPBWT_MODE

file **xg_position.cpp**
#include “xg_position.hpp”

file **xg_position.hpp**
#include “vg.pb.h”#include “types.hpp”#include “xg.hpp”#include “lru_cache.h”#include “utility.hpp”#include “json2pb.h”#include <gcsa/gcsa.h>#include <iostream> Functions for working with cached Positions and `pos_t`s.

dir **src**

dir **src/subcommand**

dir **src/unittest**

`vg` is a tool suite for working with **variation graphs**: graph-based data structures used to represent genomic information. These pages document the internals of the `vg` tool.

User documentation for `vg` is available at the vg wiki.

Index

Symbols

A

B

C

E

G

H

J

L

M

N

O

P

Q

S

T

U

V

X