Graph Library: Graph Container Interface

Document #:	**P3130r4**
Date:	2026-07-06
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency
Revises:	P3130r3

Reply-to:	Phil Ratzloff (SAS Institute) phil.ratzloff@sas.com Andrew Lumsdaine lumsdaine@gmail.com
Contributors:	Kevin Deweese Muhammad Osama (AMD, Inc) Scott McMillan (Carnegie Mellon University) Jesun Firoz Michael Wong (Intel) Jens Maurer Richard Dosselmann (University of Regina) Matthew Galati (Amazon) Guy Davidson (Creative Assembly) Oliver Rosten

1 Getting Started

This paper is one of several interrelated papers for a proposed Graph Library for the Standard C++ Library. The Table 1 describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now separated into the following papers.
P3126	Active	**Overview**, describes the big picture of what we are proposing.
P3127	Active	**Background and Terminology** provides the motivation, theoretical background, and terminology used across the other documents.
P3128	Active	**Algorithms** covers the initial algorithms as well as the ones we'd like to see in the future.
P3129	Active	**Views** has helpful views for traversing a graph.
P3130	Active	**Graph Container Interface** is the core interface used for uniformly accessing graph data structures by views and algorithms. It is also designed to easily adapt to existing graph data structures.
P3131	Active	**Graph Containers** describes a proposed high-performance `compressed_graph` container. It also discusses how to use containers in the standard library to define a graph, and how to adapt existing graph data structures.
P3337	Active	**Comparison to other graph libraries** on performance and usage syntax.

Table 1: Graph Library Papers

Reading them in order will give the best overall picture. If you're limited on time, you can use the following guide to focus on the papers that are most relevant to your needs.

Reading Guide

— If you're **new to the Graph Library**, we recommend starting with the *Overview* (P3126) paper to understand the focus and scope of our proposals. You'll also want to check out how it stacks up against other graph libraries in performance and usage syntax in the *Comparison* (P3337) paper.

— If you want to **understand the terminology and theoretical background** that underpins what we're doing, you should read the *Background and Terminology* (P3127) paper.

— If you want to **use the algorithms**, you should read the *Algorithms* (P3128) and *Graph Containers* (P3131) papers. You may also find the *Views* (P3129) and *Graph Container Interface* (P3130) papers helpful.

— If you want to **write new algorithms**, you should read the *Views* (P3129), *Graph Container Interface* (P3130), and *Graph Containers* (P3131) papers. You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.

— If you want to **use your own graph data structures**, you should read the *Graph Container Interface* (P3130) and *Graph Containers* (P3131) papers.

2 Revision History

P3130r0

— Split from P1709r5. Added *Getting Started* section.

— Add default implementation for `target_id(g,uv)` when the graph type matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range<forward_range<tuple<integral,...>>>`; `vertex_id_t<G>` also defaults to the `integral` type given.

— Revised concept definitions, adding `sourced_targeted_edge` and `target_edge_range`, and replaced summary table with code for clarity. Also assured that all combinations of adjacency list concepts for *basic*, *sourced* and *index* exist.

— Move text for graph data structures created from std containers from Graph Container Interface to Container Implementation paper.

— Identify all `concept` definitions as "For exposition only" until we have consensus of whether they belong in the standard or not.

P3130r1

— Add `num_edges(g)` and `has_edge(g)` functions. Split function table into 3 tables for graph, vertex and edge functions because it was getting too big.

— Removed the Load Graph Data section with it's load functions from P3130 Graph Container Interface because it unnecessarily complicates the interface with constructors for graph data structures. To complement this, constructors have been added for `compressed_graph` in P3131 Graph Containers.

— Revised partition functions after implementation in `compressed_graph` to reflect usage, including: renaming `partition_count(g)` to `num_partitions(g)` to match other names used, changed `partition_id(g,u)` to `partition_id(g,uid)` because vertices may not exist when the function is called, and removing `edges(g,u,pid)` because it can easily be implemented as a filter using ranges functionality when target vertices can be in different partitions.

P3130r2

— Add the edgelist as an abstract data structure as a peer to the adjacency list. This causes a reorganization of this paper and the addition of a new section for the edgelist.

— Remove unnecessary `E` edge template parameter in concepts.

— Remove type traits `is_unordered_edge` and `is_ordered_edge` because their matching concepts, `unordered_edge` and `ordered_edge`, don't need them.

— Remove `edge_id(g,uv)` and `edge_id_t<G>` because they don't add value to the interface and can easily be implemented if needed.

— Added description of why the return type isn't validated for `target_id(g,uv)` in the `basic_targeted_edge` concept.

P3130r3

— Introduction of new boost::graph-like descriptors with the following changes:

  — Concrete vertex and edge types defined by a graph container are replaced with absract vertex and edge descriptors. This allows the graph container to be decoupled from the underlying container type. This enables future expansion by allowing a graph container that uses associative containers with minimal impact to the interface.

    While this is a major revision to the implementation, it isn't a big conceptual change to the user. The visible changes include replacing `vertex_id(g,ui)` with `vertex_id(g,u)`, the addition of `partition_id(g,ui)`, and `vertices(g)` and `edges(g,u)` now return a `descriptor_view` instead of a range of the underlying container.

  — A descriptor replaces the combination of id and reference parameters from the previous version of the interface. This reduces the number of concepts by the removal of the "basic" name qualifiers for vertex_id-only concepts. It also enables consolidation of the views in P3129, where the "basic" views are no longer needed, reducing the number of view functions in half.

  — The previous `descriptor` structs in P3129 Views have been renamed to `info` structs to avoid name clashes. Additionally, the "copyable" type aliases for the `info` structs are no longer needed because reference was replaced with `descriptor`, which is now copyable.

P3130r4

— **Mapped vertices and edges**: vertex container may now be associative (e.g. `std::map`, `std::unordered_map`). Likewise, an edge container/range may also be associative or include a key-only container such as `std::set`. Related changes:

  — Support for non-integral vertex ids.

  - New concept `mapped_vertex_range` alongside `index_vertex_range`.
  - New *raw-vertex-id-type* exposition-only type alias.
  - New `vertex_property_map<G,T>` utility type alias; `make_vertex_property_map` (eager and lazy), `vertex_property_map_contains`, `vertex_property_map_get`.
- **Bidirectional graph support**: a graph may now expose in-edge ranges alongside out-edge ranges, enabling efficient reverse traversal. Related changes:
  - New concept `in_edge_range<R,G>` alongside `out_edge_range<R,G>` (together replacing `targeted_edge_range`).
  - Existing edge CPOs renamed with `out_` prefix for symmetry: `edges(g,u)` → `out_edges(g,u)` (`edges` kept as alias); `degree(g,u)` → `out_degree(g,u)`; `find_vertex_edge` → `find_out_edge`; `contains_edge` → `contains_out_edge`.
  - New in-edge CPOs: `in_edges`, `in_degree`, `find_in_edge`, `contains_in_edge`.
  - New type aliases `in_edge_range_t<G>` and `in_edge_t<G>`; `vertex_edge_range_t<G>` / `edge_t<G>` retained as aliases for `out_edge_range_t<G>` / `out_edge_t<G>`.
- Ground the descriptor types introduced in r3 with concrete specifications backed by a working prototype: `vertex_descriptor<VertexIter>` and `edge_descriptor<EdgeIter,VertexIter,EdgeDirection>` now have fully specified, exposition-only internals. Descriptor traits (`is_vertex_descriptor_v`, `is_edge_descriptor_v`, etc.) are defined in a dedicated header. `vertex_descriptor_view` and `edge_descriptor_view` replace the former `descriptor_view`.
- Simplify the concept hierarchy: the single `edge<G,E>` concept replaces the previous `targeted_edge`/`sourced_edge` split; six adjacency list concepts cover the index/mapped/bidirectional axes; `basic_*` and `sourced_*` variants are removed.
- Removed type aliases: `graph_reference_t`, `vertex_reference_t`, `edge_reference_t`.
- New concepts: `vertex_value_function` and `edge_value_function` (value-function concepts for views and algorithms).
- Edgelist namespace renamed from `std::graph::edgelist` to `std::graph::edge_list` to distinguish it from the edgelist view. Adjacency list symbols now formally reside in `std::graph::adj_list` and are re-exported to `std::graph`; the two sub-namespaces are peers.
- `vertex_data`, `edge_data` and `neighbor_data` aggregates (renamed from `vertex_info`, `edge_info`, `neighbor_info`) with partial-specialisation families supply the value types held in associative containers.
- Edgelist CPOs moved to 2-argument form `f(el,uv)`; `edge_info` renamed to `edge_data`; `edge_reference_t<EL>` removed; `raw-vertex-id-type` added.
- Completed the migration of std-container material to the Graph Containers proposal: this paper now states only the normative recognition patterns (random-access / associative vertex patterns and the integral/`pair`/`tuple` edge element patterns), with the concrete container catalog, trade-off table and worked examples moved to that paper.
- Moved `vertex_data`, `edge_data`, and `neighbor_data` descriptions from P3129 Views to this paper.
- Simplified the GCI interface by removing most GCI function overloads that take a vertex id as an argument. `find_out_edge(g,u,v)` replaces all overloads that take a vertex id as an argument.

3 Naming Conventions

Table 2 shows the naming conventions used throughout the Graph Library documents.

Template Parameter	Type Alias	Variable Names	Description
G			Graph
	graph_reference_t<G>	g	Graph reference
GV		val	Graph Value, value or reference
EL		el	Edge list
V	vertex_t<G>	u,v	Vertex descriptor. u is the source (or only) vertex. v is the target vertex.
VId	vertex_id_t<G>	uid,vid,source	Vertex id. uid is the source (or only) vertex id. vid is the target vertex id.
VV	vertex_value_t<G>	val	Vertex Value, value or reference. This can be either the user-defined value on a vertex, or a value returned by a function object (e.g. VVF) that is related to the vertex.
VR	vertex_range_t<G>	ur,vr	Vertex Range
VI	vertex_iterator_t<G>	ui,vi	Vertex Iterator. ui is the source (or only) vertex iterator. vi is the target vertex iterator.
		first,last	first and last are the begin and end iterators of a vertex range.
VVF		vvf	Vertex Value Function: vvf(g, u) → vertex value, or vvf(g, uid) → vertex value, depending on requirements of the consuming algorithm or view.
VProj		vproj	Vertex data projection function: vproj(u) → vertex_data<VId,VV>.
	partition_id_t<G>	pid	Partition id.
		P	Number of partitions.
PVR	partition_vertex_range_t<G>	pur,pvr	Partition vertex range.
E	edge_t<G>	uv,vw	Edge descriptor. uv is an edge from vertices u to v. vw is an edge from vertices v to w.
EV	edge_value_t<G>	val	Edge Value, value or reference. This can be either the user-defined value on an edge, or a value returned by a function object (e.g. EVF) that is related to the edge.
ER	vertex_edge_range_t<G>		Edge Range for edges of a vertex
EI	vertex_edge_iterator_t<G>	uvi,vwi	Edge Iterator for an edge of a vertex. uvi is an iterator for an edge from vertices u to v. vwi is an iterator for an edge from vertices v to w.
EVF		evf	Edge Value Function: evf(g, uv) → edge value.
EProj		eproj	Edge data projection function: eproj(uv) → edge_data<VId,Sourced,EV>.

Table 2: Naming Conventions for Types and Variables

4 Graph Container Interface

The Graph Container Interface (GCI) defines the primitive concepts, traits, types and functions used to define and access adjacency lists and edge lists, no matter their internal design and organization. For instance, an adjacency list can be a vector of lists from standard containers, CSR-based graph, or an adjacency matrix. Likewise, an edge list can be the edges from an adjacency list, or a range of edges from a standard container or externally defined edge types, provided they have a source_id, target_id and optional edge_value.

The GCI covers two peer abstract data types, each with its own namespace:

— `std::graph::adj_list` — concepts, type aliases, CPOs and descriptors for adjacency lists.

— `std::graph::edge_list` — concepts, type aliases and CPOs for edge lists.

All adjacency list symbols are additionally re-exported into `std::graph` for convenience. Edge list symbols are *not* re-exported because several names (notably `vertex_id_t`) have different definitions in each namespace and cannot coexist at the root level.

[PHIL: Consider not exporting the adjacency list symbols into `std::graph` and instead require users to explicitly use the `std::graph::adj_list` namespace.]

The GCI functions are customization point objects (CPOs) that provide access to adjacency lists and edge lists, providing a uniform interface for retrieving vertices, edges, and related properties regardless of the underlying container. They may be overridden for external adjacency list data structures to enable the use of algorithms on those data structures. This achieves the same goals as the STL, where algorithms can be used on any container that meets the requirements of the algorithm.

5 Adjacency List Interface

5.1 Namespace

All adjacency list concepts, types, CPOs, and functions are defined in the `std::graph::adj_list` namespace. Every symbol is also re-exported into `std::graph` via `using` declarations so that callers who include `<graph>` can write `graph::vertices(g)` rather than `graph::adj_list::vertices(g)`. Code that wishes to be explicit about the choice of abstract data type, or that uses both an adjacency list and an edge list simultaneously, may qualify names with the `graph::adj_list::` prefix.

5.2 Concepts

This section describes the concepts used for adjacency lists in the Graph Library. Three qualifiers are used in concept names.

— **index** — the vertex range is random-access and the vertex id is an integral type.

— **mapped** — vertices are stored in an associative container (e.g.`map` or `unordered_map`); vertex lookup uses `find_vertex(g, uid)`.

— **bidirectional** — the graph exposes in-edges via `in_edges(g, u)` in addition to out-edges.

While we believe the use of concepts is appropriate for graphs as a range-of-ranges, we are marking them as "For exposition only" until we have consensus of whether they belong in the standard or not.

5.2.1 Edge Concepts

Two layered edge concepts are defined. `basic_edge<G,E>` is the shared floor: an edge must provide `source_id(g,e)` and `target_id(g,e)`. It is defined in `std::graph` and depends only on the shared id CPOs, so it ties an edge to both the adjacency list and the edge list (it is the concept required by `basic_sourced_edgelist`, below).

[PHIL: The `basic_sourced_edgelist` uses a carry-over name from using "sourced" when source was optional, which no longer occurs. Consider renaming.]

The `edge<G,E>` concept refines `basic_edge` by additionally requiring the `source(g,e)` and `target(g,e)` that return vertex descriptors. Within an `adjacency_list`, `out_edges(g,u)` for vertex `u` is always available and yields a range of edge descriptors, and `in_edges(g,u)` additionally yields a range of edge descriptors when a

`bidirectional_adjacency_list` is used. `adjacency_list` implies a `vertex_range` (hence `find_vertex`), so the default `source`/`target` resolve to `*find_vertex(g, source_id/target_id(g,e))`. The descriptor requirement is therefore free for conforming graphs while keeping edge-list elements — which have no vertex container — out of the adjacency-list `edge` concept; those satisfy only `basic_edge`.

```
// For exposition only

// Shared edge floor (namespace std::graph):  ids only.
template <class G, class E>
concept basic_edge = requires(G& g, const E& uv) {
  source_id(g, uv); // returns vertex_id_t<G>
  target_id(g, uv); // returns vertex_id_t<G>
};

// Adjacency-list refinement (namespace std::graph::adj_list):
// adds the source/target vertex descriptors.
template <class G, class E>
concept edge = basic_edge<G, E> && is_edge_descriptor_v<E> && requires(G& g, const E& uv) {
  source(g, uv); // returns vertex descriptor vertex_t<G>
  target(g, uv); // returns vertex descriptor vertex_t<G>
};
```

Return types are not validated for better error reporting. Constraining the return type to `vertex_id_t<G>` often results in obscure error messages on failure. Leaving the return type unconstrained allows the compiler to report the actual type mismatch at the point of use, which is easier to diagnose. There is precedent for this choice in the `sized_range` concept.

5.2.1.1 Edge Range Concepts

`out_edge_range<R,G>` constrains the range returned by `out_edges(g,u)`, and `in_edge_range<R,G>` constrains the range returned by `in_edges(g,u)`. Both require a `forward_range` whose value type satisfies `edge<G, range_value_t<R>>`.

```
// For exposition only

template <class R, class G>
concept out_edge_range = forward_range<R> &&
                      edge<G, range_value_t<R>>;

template <class R, class G>
concept in_edge_range = forward_range<R> &&
                     edge<G, range_value_t<R>>;
```

5.2.2 Vertex Concepts

The `vertex<G,V>` concept requires that `V` is a vertex descriptor (`is_vertex_descriptor_v<V>`) that supports `vertex_id(g,u)` and `find_vertex(g,uid)`. The descriptor guard distinguishes a vertex descriptor from a raw vertex value, so the concept does not accidentally match unrelated types. Building on it, `vertex_range<R,G>` requires a `forward_range` and `sized_range` whose value type satisfies `vertex<G,...>`.

The `hashable_vertex_id<G>` concept holds when `vertex_id_t<G>` is usable as a key in a `std::unordered_map` (i.e. it is hashable). It is used by `mapped_vertex_range` and by the `vertex_property_map` utilities, which back per-vertex state with an `unordered_map` for mapped graphs.

Two specializations of vertex range are provided:

- `index_vertex_range<G>` — vertex id is integral and the vertex range is random-access with an integral `storage_type`. Used for high-performance graphs such as `index_adjacency_list`.
- `mapped_vertex_range<G>` — not an `index_vertex_range`, the vertex id is `hashable_vertex_id`, and vertices are found via `find_vertex(g, uid)`. Used for graphs whose vertices are stored in associative containers.

[PHIL: `integral<typenamevertex_range_t<G>::storage_type>` is an obscure way to express that the storage type of the `index_vertex_range` is an integral vs. iterator/mapped type.]

```
// For exposition only

template <class G, class V>
concept vertex = is_vertex_descriptor_v<remove_cvref_t<V>> &&
                 requires(G& g, const V& u, const vertex_id_t<G>& uid) {
                   vertex_id(g, u);
                   { find_vertex(g, uid) } -> std::forward_iterator;
                 };

template <class R, class G>
concept vertex_range = forward_range<R> &&
                       sized_range<R> &&
                       vertex<G, remove_cvref_t<range_value_t<R>>>;

template <class G>
concept index_vertex_range =
    integral<vertex_id_t<G>> &&
    integral<typename vertex_range_t<G>::storage_type> &&
    requires(G& g) {
      { vertices(g) } -> vertex_range<G>;
    };

// vertex id usable as an unordered_map key (for mapped property maps)
template <class G>
concept hashable_vertex_id = requires(const vertex_id_t<G>& uid) {
  { hash<vertex_id_t<G>>{}(uid) } -> convertible_to<size_t>;
};

template <class G>
concept mapped_vertex_range =
    !index_vertex_range<G> &&
    hashable_vertex_id<G> &&
    requires(G& g, const vertex_id_t<G>& uid) {
      { vertices(g) } -> forward_range;
      find_vertex(g, uid);
    };
```

5.2.3 Adjacency List Concepts

The adjacency list concepts combine the vertex and edge concepts into six main concepts covering three axes (index vs. mapped storage, unidirectional vs. bidirectional), plus two orthogonal refinements. All algorithms currently proposed for the Graph Library require `index_adjacency_list`. Figure 1 shows the full refinement hierarchy.

`ordered_vertex_edges<G>` is a primarily *semantic* concept: it can only confirm structurally that `out_edges(g,u)` is a `forward_range`, while the ascending-`target_id` ordering guarantee is asserted by the graph author. Graphs that store edges in ordered containers (e.g. `set` or `map` keyed by target id) satisfy it; unordered storage (e.g. `vector` appended in arbitrary order, or `unordered_set`) generally does not. It is used by algorithms that rely on a linear set-intersection merge, such as triangle counting (see the Algorithms proposal).

```
// For exposition only

template <class G>
concept adjacency_list = requires(G& g, vertex_t<G> u) {
  { vertices(g) } -> vertex_range<G>;
  { out_edges(g, u) } -> out_edge_range<G>;
};

template <class G>
concept index_adjacency_list = adjacency_list<G> &&
                               index_vertex_range<G>;
```

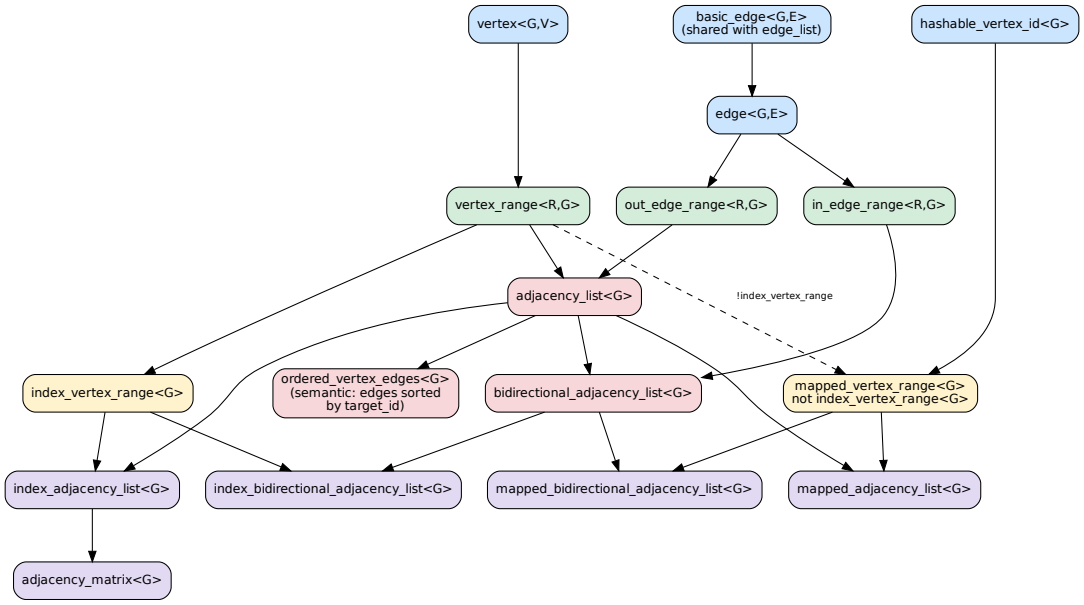

Figure 1: Adjacency list concept refinement hierarchy. Arrows denote refinement (child concept requires parent). Dashed arrow indicates negation constraint. All algorithms in this proposal require `index_adjacency_list` .

```
template <class G>
concept bidirectional_adjacency_list =
    adjacency_list<G> &&
    requires(G& g, vertex_t<G> u, in_edge_t<G> ie) {
      { in_edges(g, u) } -> in_edge_range<G>;
      { source_id(g, ie) } -> convertible_to<vertex_id_t<G>>;
    };

template <class G>
concept index_bidirectional_adjacency_list =
    bidirectional_adjacency_list<G> &&
    index_vertex_range<G>;

template <class G>
concept mapped_adjacency_list = adjacency_list<G> &&
                            mapped_vertex_range<G>;

template <class G>
concept mapped_bidirectional_adjacency_list =
    bidirectional_adjacency_list<G> &&
    mapped_vertex_range<G>;

template <class G>
concept adjacency_matrix = index_adjacency_list<G> &&
                        requires(G& g, vertex_t<G> u, vertex_t<G> v) {
                          find_out_edge(g, u, v) -> std::forward_iterator;
                          { contains_out_edge(g, u, v) } -> std::convertible_to<bool>;
                        };

// Semantic refinement:  each vertex's out-edges are sorted by ascending
// target_id.  The structural check below only confirms a forward edge range;
```

```
// the ascending-order property is a semantic requirement the author asserts.
template <class G>
concept ordered_vertex_edges =
    adjacency_list<G> &&
    requires(G& g, vertex_t<G> u) {
      requires forward_iterator<decltype(begin(out_edges(g, u)))>;
    };
```

5.3 Traits

Table 3 summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph's characteristics.

Trait	Type	Comment
`has_degree<G>`	concept	Is the `out_degree(g,u)` function available?
`has_find_vertex<G>`	concept	Are the `find_vertex(g,_)` functions available?
`has_find_out_edge<G>`	concept	Are the `find_out_edge(g,_)` functions available?
`has_contains_edge<G,V>`	concept	Is the `contains_out_edge(g,uid,vid)` function available?
`has_in_degree<G>`	concept	Is the `in_degree(g,u)` function available?
`has_find_in_edge<G>`	concept	Are the `find_in_edge(g,_)` functions available?
`has_contains_in_edge<G,V>`	concept	Is the `contains_in_edge(g,uid,vid)` function available?
`has_basic_queries<G>`	concept	`has_degree<G> && has_find_vertex<G> && has_find_vertex_edge<G>`
`has_full_queries<G>`	concept	`has_basic_queries<G> && has_contains_edge<G, vertex_t<G>>`

Table 3: Graph Container Interface Type Traits

5.4 Types

Table 4 summarizes the type aliases in the Graph Container Interface. These are the types used to define the objects in a graph container, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, compressed_graph and adjacency matrix.

The type aliases are defined by either a function specialization for the underlying graph container, or a refinement of one of those types (e.g. an iterator of a range). Table 5 describes the functions in more detail.

`graph_value(g)`, `vertex_value(g,u)` and `edge_value(g,uv)` can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types.

There is no contiguous requirement for `vertex_id` from one partition to the next, though in practice they will often be assigned contiguously. Gaps in `vertex_id`s between partitions should be allowed.

5.5 Classes and Structs

The `graph_error` exception class is available, inherited from `runtime_error`. While any function may use it, it is only anticipated to be used by the `load` functions at this time. No additional functionality is added beyond that provided by `runtime_error`.

While we believe the use of concepts is appropriate for graphs as a range-of-ranges, we are marking them as "For exposition only" until we have consensus of whether they belong in the standard or not.

5.6 Functions

Tables 5, 6 and 7 summarize the primitive functions in the Graph Container Interface. used to access an adjacency graph, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs such as a vector of lists, CSR-based graph and adjacency matrix.

Type Alias	Definition	Comment
`graph_value_t<G>`	`decltype(graph_value(g))`	optional
`vertex_range_t<G>`	`decltype(vertices(g))`	
`vertex_iterator_t<G>`	`iterator_t<vertex_range_t<G>>`	
`vertex_t<G>`	`range_value_t<vertex_range_t<G>>`	descriptor
`vertex_id_t<G>`	`remove_cvref_t<decltype(vertex_id(g,vertex_t<G>))>`	
raw-vertex-id-type	`decltype(vertex_id(g,vertex_t<G>))`	exposition only
`vertex_value_t<G>`	`decltype(vertex_value(g,u))`	optional
`out_edge_range_t<G>`	`decltype(out_edges(g,vertex_t<G>))`	
`out_edge_iterator_t<G>`	`iterator_t<out_edge_range_t<G>>`	
`out_edge_t<G>`	`range_value_t<out_edge_range_t<G>>`	descriptor
`vertex_edge_range_t<G>`	`out_edge_range_t<G>`	alias
`vertex_edge_iterator_t<G>`	`out_edge_iterator_t<G>`	alias
`edge_t<G>`	`out_edge_t<G>`	alias
`edge_value_t<G>`	`decltype(edge_value(g,uv))`	optional
`in_edge_range_t<G>`	`decltype(in_edges(g,vertex_t<G>))`	
`in_edge_iterator_t<G>`	`iterator_t<in_edge_range_t<G>>`	
`in_edge_t<G>`	`range_value_t<in_edge_range_t<G>>`	descriptor
`partition_id_t<G>`	`decltype(partition_id(g,u))`	optional
`partition_vertex_range_t<G>`	`vertices(g,pid)`	optional

Table 4: Graph Container Interface Type Aliases

Function	Return Type	Complexity	Default Implementation
`graph_value(g)`	`graph_value_t<G>`	constant	n/a, optional
`vertices(g)`	`vertex_range_t<G>`	constant	`g` if `random_access_range<G>`, n/a otherwise
`num_vertices(g)`	`integral`	constant	`size(vertices(g))`
`num_edges(g)`	`integral`	|E|	`n=0; for(u: vertices(g))n+=distance(out_edges(g,u)); return n;`
`has_edges(g)`	`bool`	|V|	`for(u: vertices(g))if !empty(out_edges(g,u))return true; return false;`
`num_partitions(g)`	`integral`	constant	1
`vertices(g,pid)`	`partition_vertex_range_t<G>`	constant	`vertices(g)` if `pid==0`, else empty range
`num_vertices(g,pid)`	`integral`	constant	`size(vertices(g))` if `pid==0`, else 0

Table 5: Graph Functions

The complexity shown above for `num_edges(g)` and `has_edges(g)` is for the default implementation. Specific graph implementations may have better characteristics.

The only vertex function that requires a vertex id (`uid`) is `find_vertex(g,uid)`. The other functions that use it are convenience functions that imply a call to `find_vertex(g,uid)` to get the vertex descriptor before a call to the overloaded function that takes a descriptor is made.

The complexity shown above for `vertices(g,pid)` and `num_vertices(g,pid)` is for the default implementation. Specific graph implementations may have different characteristics.

[PHIL: The same edge type is required for both incoming and outgoing edge ranges. Consider enabling different edge types.]

The default implementation for the `out_degree` and `in_degree` functions assumes that `out_edge_range_t<G>` or `in_edge_range_t<G>` is a sized range to have constant complexity. If the underlying container has a non-linear `size(R)` function, the degree functions will also be non-linear. This is expected to be an uncommon case.

The descriptor form `partition_id(g,u)` is always available, defaulting to `0` (all vertices in partition 0). The id form `partition_id(g,uid)` — the convenience overload that would resolve to `partition_id(g,*find_vertex(g,uid))` — is not yet in the reference implementation.

When the graph matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range`

Function	Return Type	Cmplx	Default Implementation
`find_vertex(g,uid)`	`vertex_iterator_t<G>`	constant	`begin(vertices(g))+ uid` if `random_access_range<vertex_range_t<G>>`
`vertex_id(g,u)`	`vertex_id_t<G>`	constant	(see Determining the vertex_id type below) Override to define a different `vertex_id_t<G>` type (e.g. int32_t).
`vertex_value(g,u)`	`vertex_value_t<G>`	constant	n/a, optional
`out_edges(g,u)`	`out_edge_range_t<G>`	constant	`u` if `forward_range<vertex_t<G>>`, n/a otherwise
`out_degree(g,u)`	`integral`	constant	`size(out_edges(g,u))` if `sized_range<out_edge_range_t<G>>`
`in_edges(g,u)`	`in_edge_range_t<G>`	constant	n/a (requires bidirectional graph)
`in_degree(g,u)`	`integral`	constant	`size(in_edges(g,u))` if `sized_range<in_edge_range_t<G>>`
`partition_id(g,u)`	`partition_id_t<G>`	constant	`0` (partition 0)
`partition_id(g,uid)`	`partition_id_t<G>`	constant	`partition_id(g,*find_vertex(g,uid))`

Table 6: Vertex Functions

Function	Return Type	Cmplx	Default Implementation
`target_id(g,uv)`	`vertex_id_t<G>`	constant	(see below)
`target(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ target_id(g, uv))` if `random_access_range<vertex_range_t<G>>` `&& integral<decltype(target_id(g,uv))>`
`edge_value(g,uv)`	`edge_value_t<G>`	constant	`uv` if `forward_range<vertex_t<G>>`, n/a otherwise, optional
`find_out_edge(g,u,v)`	`out_edge_t<G>`	linear	`find(out_edges(g,u), [](uv){ return target_id(g,uv)==vertex_id (v); })`
`contains_out_edge(g,uid,vid)`	`bool`	constant	`uid < size(vertices(g)) && vid < size(vertices(g))` if `is_adjacency_matrix_v<G>`.
		linear	`find_out_edge(g,uid,vid) != end(out_edges(g,uid))` otherwise.
`source_id(g,uv)`	`vertex_id_t<G>`	constant	mandatory for descriptor-based edges
`source(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ source_id(g,uv))` if `random_access_range<vertex_range_t<G>>` `&& integral<decltype(source_id(g,uv))>`
`find_in_edge(g,u,v)`	`in_edge_t<G>`	linear	n/a (requires bidirectional graph)
`contains_in_edge(g,uid,vid)`	`bool`	linear	`find_in_edge(g,uid,vid) != end(in_edges(g,uid))`

Table 7: Edge Functions

`<forward_range<tuple<integral,...>>>`, the default implementation for `target_id(g,uv)` will return the `integral`. Additionally, if the caller does not override `vertex_id(g,u)`, the `integral` value will define the `vertex_id_t<G>` type.

Functions that have n/a for their Default Implementation must be defined by the author of a Graph Container implementation.

The constant-time `contains_out_edge(g,uid,vid)` branch guarded by `is_adjacency_matrix_v<G>` depends on the adjacency_matrix trait family, which is not yet in the reference implementation; until then the linear `find_out_edge`-based default is always used.

Value functions (`graph_value(g)`, `vertex_value(g,u)` and `edge_value(g,uv)`) can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types. They return a single value and can be scalar, struct, class, union, or tuple. These are abstract types used by the GVF, VVF and EVF function objects to retrieve values used by algorithms. As such it's valid to return the "enclosing" owning class (graph, vertex or edge), or some other embedded value in those objects.

`find_vertex(g,uid)` is constant complexity because all algorithms in this proposal require that `vertex_range_t<G>` is a random access range.

If the concept requirements for the default implementation aren't met by the graph container the function will need to be overridden.

5.7 Determining the vertex_id and its type

To determine the type for `vertex_id_t<G>` the following steps are taken, in order, to determine its type.

1. Use the type returned by `vertex_id(g,u)` when overridden for a graph.
2. When the graph matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range<forward_range<tuple<integral,...>>>`, use the `integral` type specified, which is assumed to be the target_id on an edge.
3. Use `size_t` in all other cases.

`vertex_id_t<G>` is defined by the type returned by `vertex_id(g)` and it defaults to the difference_type of the underlying container used for vertices (e.g. int64_t for 64-bit systems). This is sufficient for all situations. However, there are often space and performance advantages if a smaller type is used, such as int32_t or even int16_t. It is recommended to consider overriding this function for optimal results, assuring that it is also large enough for the number of possible vertices and edges in the application. It will also need to be overridden if the implementation doesn't expose the vertices as a range.

`vertex_id(g,u)` is evaluated in the context of a descriptor using the following rules:

1. Use the value returned by `vertex_id(g,u)` when overridden for a graph.
2. Use the index value on the descriptor.

5.8 Vertex and Edge Descriptor Views

Graph containers may store vertices and edges in very different data structures: a `vector`-based adjacency list uses integer indices as vertex identifiers while a `map`-based graph uses the map key. Exposing raw iterators directly would require concepts, algorithms, and user code to handle each storage strategy separately. Descriptors solve this by providing a lightweight, opaque handle that works uniformly regardless of the underlying container type. Table 8 compares raw iterators with descriptors.

Concern	Raw iterators	Descriptors
Abstraction	Expose container layout	Hide storage strategy — one concept covers indexed and keyed graphs
Overload reduction	Separate overloads for index vs. iterator	A single overload accepts the descriptor
Invalidation	Invalidated by container mutation	Can be re-resolved via `find_vertex`
Copyability	Varies (e.g. `forward_list::iterator`)	Always trivially copyable or cheap
Size	May be pointer-sized or larger	Index-based vertex: one integer; edge: an iterator plus the source vertex descriptor

Table 8: Raw iterators vs. descriptors

Index-based vertex descriptors provide an additional level of abstraction: they can refer to a vertex even when no physical vertex object exists in the graph. A `compressed_graph`, for example, stores only edges; there is no C++ object corresponding to each vertex. Because the vertex is identified purely by its index, the GCI's algorithms and views work identically whether or not the container materialises explicit vertex storage. An iterator-based approach cannot offer this property — an iterator must point to an actual object.

For such index-only graphs the GCI provides a canonical descriptor specialization: `index_iterator` is the iterator type of an `iota_view<size_t,size_t>` (a random-access iterator that yields indices by value, with no backing element), and `index_vertex_descriptor` is `vertex_descriptor<index_iterator>`. Because `index_iterator` is random-access, the descriptor stores a `size_t` index and behaves like any other index-based

`vertex_descriptor`, except that `underlying_value`/`inner_value` are not applicable (there is no physical element to dereference).

CPOs automatically detect the storage pattern of an adjacency list and select the appropriate descriptor strategy: **random-access** containers (`vector`, `deque`) produce index-based vertex descriptors; **associative** containers (`map`, `unordered_map`) produce key-based (iterator) descriptors. Edge patterns — `int`, `pair<int,W>`, `tuple<int,...>`, or custom structs — are recognised automatically so that `target_id`, `source_id`, and `edge_value` CPOs are provided without user specialisation.

The vertex and edge descriptors are defined as the `vertex_t<G>` and `edge_t<G>` types, respectively. `vertex_descriptor<VertexIter>` stores either a `size_t` index (for random-access containers) or an iterator (for associative containers) as an exposition-only private member. `edge_descriptor<EdgeIter,VertexIter,EdgeDirection>` always stores the edge as its iterator (`EdgeIter`) — unlike a vertex, an edge is always backed by a physical container element, so there is no index-only edge path. It additionally stores the source `vertex_descriptor` and an `EdgeDirection` tag (`out_edge_tag` or `in_edge_tag`) that controls source/target semantics, allowing bidirectional graphs to use the same edge type for both outgoing and incoming edges.

Both descriptor types support equality comparison (`==`, `!=`), ordering (`<`, `<=`, `>`, `>=` where supported), copy, assignment, and default construction. They also provide `underlying_value(c)` and `inner_value(c)` member function templates, where `c` is the owning container, to retrieve a reference to the underlying element. These are needed only when overriding CPOs to adapt an external graph container to the GCI.

Because descriptors are small and trivially copyable, CPO signatures accept them by value. A `vertex_descriptor` is at most pointer-sized (one index or iterator). An `edge_descriptor` is larger because it carries an edge iterator plus the source `vertex_descriptor`, but is still cheap to copy. The `const&` form appears only in `requires` clauses of concepts where value copies are not permitted.

```
// For exposition only

struct out_edge_tag {};
struct in_edge_tag {};

template <class VertexIter>
class vertex_descriptor {
public:
  using iterator_type = VertexIter;
  using value_type = iter_value_t<VertexIter>;

  // index for random-access, iterator for bidirectional (for exposition only)
  using storage_type =
      conditional_t<random_access_iterator<VertexIter>, size_t, VertexIter>;

  constexpr vertex_descriptor() = default;
  constexpr explicit vertex_descriptor(storage_type val) noexcept;

  // Properties
  constexpr storage_type value() const noexcept;
  constexpr decltype(auto) vertex_id() const noexcept;

  template <class Container>
  constexpr decltype(auto) underlying_value(Container& c) const noexcept;

  template <class Container>
  constexpr decltype(auto) inner_value(Container& c) const noexcept;

  // Iterator-like interface
  constexpr vertex_descriptor& operator++() noexcept;
  constexpr vertex_descriptor operator++(int) noexcept;

  // Comparison
  constexpr auto operator<=>(const vertex_descriptor&) const noexcept = default;
  constexpr bool operator==(const vertex_descriptor&) const noexcept = default;

private:
```

```
  storage_type storage_ = storage_type(); // for exposition only
};


template <class EdgeIter,
          class VertexIter,
          class EdgeDirection = out_edge_tag>
class edge_descriptor {
public:
  using edge_iterator_type = EdgeIter;
  using vertex_iterator_type = VertexIter;
  using vertex_desc = vertex_descriptor<VertexIter>;
  using edge_direction = EdgeDirection;

  static constexpr bool is_in_edge = is_same_v<EdgeDirection, in_edge_tag>;
  static constexpr bool is_out_edge = is_same_v<EdgeDirection, out_edge_tag>;

  // Edges are always backed by a physical container, so the edge is
  // stored as its iterator directly (no index-only path).  (for exposition only)
  using edge_storage_type = EdgeIter;

  constexpr edge_descriptor() = default;
  constexpr edge_descriptor(edge_storage_type edge_val,
                            vertex_desc source) noexcept;

  // Properties
  constexpr edge_storage_type value() const noexcept;
  constexpr vertex_desc source() const noexcept;
  constexpr decltype(auto) source_id() const noexcept;

  template <class VertexData>
  constexpr auto target_id(const VertexData& vd) const noexcept;

  template <class VertexData>
  constexpr decltype(auto) underlying_value(VertexData& vd) const noexcept;

  // Comparison
  constexpr auto operator<=>(const edge_descriptor&) const noexcept = default;
  constexpr bool operator==(const edge_descriptor&) const noexcept = default;

private:
  edge_storage_type edge_storage_ = edge_storage_type(); // for exposition only
  vertex_desc source_ = vertex_desc(); // for exposition only
};
```

The following type traits identify whether a type is a vertex or edge descriptor.

```
// For exposition only

template <class T> struct is_vertex_descriptor : false_type {};
template <class T> struct is_edge_descriptor : false_type {};
template <class T> struct is_descriptor : false_type {};

// Specializations for vertex_descriptor
template <class VertexIter>
struct is_vertex_descriptor<vertex_descriptor<VertexIter>> : true_type {};
template <class VertexIter>
struct is_descriptor<vertex_descriptor<VertexIter>> : true_type {};

// Specializations for edge_descriptor
template <class EdgeIter, class VertexIter, class EdgeDirection>
struct is_edge_descriptor<edge_descriptor<EdgeIter, VertexIter, EdgeDirection>>
    : true_type {};
template <class EdgeIter, class VertexIter, class EdgeDirection>
```

```
struct is_descriptor<edge_descriptor<EdgeIter, VertexIter, EdgeDirection>>
    : true_type {};

// Helper variable templates
template <class T>
inline constexpr bool is_vertex_descriptor_v = is_vertex_descriptor<remove_cv_t<T>>::value;

template <class T>
inline constexpr bool is_edge_descriptor_v = is_edge_descriptor<remove_cv_t<T>>::value;

template <class T>
inline constexpr bool is_descriptor_v = is_descriptor<remove_cv_t<T>>::value;
```

The only vertex function that requires a vertex id (`uid`) is `find_vertex(g,uid)`. All other functions that accept vertex id are convenience functions that imply a call to `find_vertex(g,uid)` to get the vertex descriptor before a call.

The following are the descriptor views used by `vertices(g)` and `out_edges(g,u)`.

```
// For exposition only

template <class VertexIter>
class vertex_descriptor_view : public view_interface<vertex_descriptor_view<VertexIter>> {
public:
  using vertex_desc = vertex_descriptor<VertexIter>;
  using storage_type = typename vertex_desc::storage_type;
  using underlying_iterator = VertexIter;

  class iterator; // forward iterator yielding vertex_desc by value
  using const_iterator = iterator;

  constexpr vertex_descriptor_view() = default;

  template <class Container>
  constexpr explicit vertex_descriptor_view(Container& c);

  constexpr iterator begin() const noexcept;
  constexpr iterator end() const noexcept;
  constexpr size_t size() const noexcept;
};


template <class EdgeIter,
          class VertexIter,
          class EdgeDirection = out_edge_tag>
class edge_descriptor_view
    : public view_interface<edge_descriptor_view<EdgeIter, VertexIter, EdgeDirection>> {
public:
  using edge_desc = edge_descriptor<EdgeIter, VertexIter, EdgeDirection>;
  using vertex_desc = vertex_descriptor<VertexIter>;

  class iterator; // forward iterator yielding edge_desc by value
  using const_iterator = iterator;

  constexpr edge_descriptor_view() = default;

  template <class Container>
  constexpr edge_descriptor_view(Container& c, vertex_desc source);

  constexpr iterator begin() const noexcept;
  constexpr iterator end() const noexcept;
  constexpr size_t size() const noexcept;
};
```

5.9 Unipartite, Bipartite and Multipartite Graph Representation

`num_partitions(g)` returns the number of partitions, or partiteness, of the graph. It has a range of 1 to n, where 1 identifies a unipartite graph, 2 is a bipartite graph, and a value of 2 or more can be considered a multipartite graph.

If a graph data structure doesn't support partitions then it is unipartite with one partition and partite functions will reflect that. For instance, `num_partitions(g)` returns a value of 1, and `vertices(g,0)` (vertices in the first partition) will return a range that includes all vertices in the graph.

A partition identifies a type of a vertex, where the vertex value types are assumed to be uniform in each partition. This creates a dilemma because the existing `vertex_value(g,u)` returns a single type based template parameter for the vertex value type. Supporting multiple types can be addressed in different ways using C++ features. The key to remember is that the actual value used by algorithms is done by calling a function object that retrieves the value to be used. That function is specific to the graph data structure, using the partition to determine how to get the appropriate value.

— `std::variant`: The lambda returns the appropriate variant value based on the partition.

— Base class pointer: The lambda can call a member function to return the value based on the partition.

— `void*`: The lambda can cast the pointer to a concrete type based on the partition, and then return the appropriate value.

`out_edges(g,uid,pid)` and `out_edges(g,u,pid)` filter the edges where the target is in the partition `pid` passed. This isn't needed for bipartite graphs. These partition-filtered `out_edges` overloads are not yet in the reference implementation.

5.10 Utility Types and Functions

5.10.1 Vertex, Edge and Neighbor Data

`vertex_data`, `edge_data` and `neighbor_data` are used to provide structured definitions of the core data model associated with vertices, edges and neighbors. They're useful aggregates that show up in different contexts including for structured-binding iteration, and binding external data for use in graph construction.

5.10.1.1 `struct vertex_data<VId, V, VV>`

`vertex_data` is used to define or return vertex information. The `id` member is the vertex id, the `vertex` member is the vertex descriptor, and `value` is the result of the value function, if provided.

```
// Primary template
template <class VId, class V, class VV>
struct vertex_data {
  using id_type = VId; // e.g.  vertex_id_t<G> or void
  using vertex_type = V; // e.g.  vertex_t<G> or void
  using value_type = VV; // e.g.  invoke_result of vvf, or void

  id_type id;
  vertex_type vertex;
  value_type value;
};

// Specialization:  no value function
template <class VId, class V>
struct vertex_data<VId, V, void> {
  using id_type = VId;
  using vertex_type = V;

  id_type id;
  vertex_type vertex;
};

// Specialization:  no vertex descriptor
template <class VId, class VV>
struct vertex_data<VId, void, VV> {
```

```
  using id_type = VId;
  using value_type = VV;

  id_type id;
  value_type value;
};

// Specialization:  id only
template <class VId>
struct vertex_data<VId, void, void> {
  using id_type = VId;

  id_type id;
};
```

Specializations are defined with `V=void` or `VV=void` to suppress the existence of their associated member variables, giving the following valid combinations in Table **??** . For instance, the second entry, `vertex_data<VId, V, void>` has two members `{id_type id; vertex_type vertex;}` and `value_type` is `void`.

`vertex_data<VId, V, VV>` has the following members depending on the template parameters (a `void` parameter omits the corresponding member):

Members present	Specialization	Typical use
`id, vertex, value`	`<VId, V, VV>`	Full vertexlist with descriptor and value
`id, vertex`	`<VId, V, void>`	Vertexlist with descriptor, no value
`id, value`	`<VId, void, VV>`	Vertexlist with value, no descriptor
`id`	`<VId, void, void>`	ID-only vertexlist
`vertex, value`	`<void, V, VV>`	Descriptor-based, no id
`vertex`	`<void, V, void>`	Descriptor only
`value`	`<void, void, VV>`	Value only

Table 9: `vertex_data` Specializations

5.10.1.2 `struct edge_data<VId, Sourced, E, EV>`

When `Sourced=true`, the `source_id` member is included. The `target_id` member always exists.

```
// Primary template
template <class VId, bool Sourced, class E, class EV>
struct edge_data {
  using source_id_type = VId; // vertex_id_t<G> when Sourced==true, or void
  using target_id_type = VId; // vertex_id_t<G>
  using edge_type = E; // edge_t<G> or void
  using value_type = EV; // invoke_result of evf, or void

  source_id_type source_id;
  target_id_type target_id;
  edge_type edge;
  value_type value;
};

// Specialization:  incidence (unsourced, no value)
template <class VId, class E>
struct edge_data<VId, false, E, void> {
  using target_id_type = VId;
  using edge_type = E;

  target_id_type target_id;
  edge_type edge;
};
```

```
// Specialization:  edgelist (sourced, no value)
template <class VId, class E>
struct edge_data<VId, true, E, void> {
  using source_id_type = VId;
  using target_id_type = VId;
  using edge_type = E;

  source_id_type source_id;
  target_id_type target_id;
  edge_type edge;
};

// Specialization:  basic incidence (unsourced, no descriptor, no value)
template <class VId>
struct edge_data<VId, false, void, void> {
  using target_id_type = VId;

  target_id_type target_id;
};

// Specialization:  search edge views (no id, unsourced)
template <class E>
struct edge_data<void, false, E, void> {
  using edge_type = E;

  edge_type edge;
};
```

`edge_data<VId, Sourced, E, EV>` has the following members. The `Sourced` boolean controls whether `source_id` is included. A `void` VId or EV omits the corresponding member; a `void` E omits the edge descriptor member.

Members present	Specialization	Typical use
`source_id, target_id, edge, value`	`<VId,true,E,EV>`	Sourced incidence with descriptor and value
`source_id, target_id, edge`	`<VId,true,E,void>`	Sourced incidence with descriptor
`source_id, target_id, value`	`<VId,true,void,EV>`	Sourced incidence with value
`source_id, target_id`	`<VId,true,void,void>`	Sourced incidence, IDs only
`target_id, edge, value`	`<VId,false,E,EV>`	Unsourced incidence with descriptor and value
`target_id, edge`	`<VId,false,E,void>`	Unsourced incidence with descriptor
`target_id, value`	`<VId,false,void,EV>`	Unsourced incidence with value
`target_id`	`<VId,false,void,void>`	Target ID only

Table 10: `edge_data` Specializations (VId present)

5.10.1.3 `struct neighbor_data<VId, Sourced, V, VV>`

`neighbor_data` is used to return information for a neighbor vertex, through an edge. The `target_id` member always exists. The `target` member holds the neighbor's vertex descriptor when `V` is not `void`.

```
// Primary template
template <class VId, bool Sourced, class V, class VV>
struct neighbor_data {
  using source_id_type = VId; // vertex_id_t<G> when Sourced==true, or void
  using target_id_type = VId; // vertex_id_t<G>
  using vertex_type = V; // vertex_t<G> or void
  using value_type = VV; // invoke_result of vvf, or void

  source_id_type source_id;
  target_id_type target_id;
```

```
  vertex_type target;
  value_type value;
};

// Specialization:  neighbors (unsourced, no value)
template <class VId, class V>
struct neighbor_data<VId, false, V, void> {
  using target_id_type = VId;
  using vertex_type = V;

  target_id_type target_id;
  vertex_type target;
};

// Specialization:  basic neighbors (unsourced, no descriptor, no value)
template <class VId>
struct neighbor_data<VId, false, void, void> {
  using target_id_type = VId;

  target_id_type target_id;
};

// Specialization:  neighbors with value (unsourced)
template <class VId, class V, class VV>
struct neighbor_data<VId, false, V, VV> {
  using target_id_type = VId;
  using vertex_type = V;
  using value_type = VV;

  target_id_type target_id;
  vertex_type target;
  value_type value;
};
```

`neighbor_data<VId, Sourced, V, VV>` is analogous to `edge_data` but replaces the edge descriptor with a target vertex descriptor. It is used by adjacency (neighbor) views.

Members present	Specialization	Typical use
`source_id, target_id, target, value`	`<VId,true,V,VV>`	Sourced neighbor descriptor and value
`source_id, target_id, target`	`<VId,true,V,void>`	Sourced neighbor descriptor
`source_id, target_id, value`	`<VId,true,void,VV>`	Sourced neighbor with value
`source_id, target_id`	`<VId,true,void,void>`	Sourced neighbor, IDs only
`target_id, target, value`	`<VId,false,V,VV>`	Unsourced neighbor descriptor and value
`target_id, target`	`<VId,false,V,void>`	Unsourced neighbor descriptor
`target_id, value`	`<VId,false,void,VV>`	Unsourced neighbor with value
`target_id`	`<VId,false,void,void>`	Unsourced neighbor, IDs only

Table 11: `neighbor_data` Specializations (VId present)

5.10.1.4 Copyable vertex, edge, and neighbor types

5.10.2 Value Function Concepts

`vertex_value_function` and `edge_value_function` define the required interface for callables passed to views and algorithms to extract per-vertex or per-edge scalar values (e.g. a weight or a label). Both concepts use the two-argument `f(const G&, descriptor)` convention so that stateless lambdas may be captured into `std::views` pipelines without holding a reference to the graph.

```
template <class VVF, class Graph, class VertexDescriptor>
concept vertex_value_function =
    invocable<VVF, const Graph&, VertexDescriptor> &&
```

```
    (!is_void_v<invoke_result_t<VVF, const Graph&, VertexDescriptor>>);

template <class EVF, class Graph, class EdgeDescriptor>
concept edge_value_function =
    invocable<EVF, const Graph&, EdgeDescriptor> &&
    (!is_void_v<invoke_result_t<EVF, const Graph&, EdgeDescriptor>>);
```

`edge_weight_function` (defined in the algorithms proposal) refines `edge_value_function` with the additional requirement that the return type is arithmetic.

5.10.3 Vertex Property Map

Algorithms that maintain per-vertex state (visited flags, distances, component labels, etc.) need a container indexed by vertex ID. For index-based graphs a `vector` is natural; for mapped (key-based) graphs an `unordered_map` is required. `vertex_property_map<G, T>` abstracts over this difference.

```
template <class G, class T>
using vertex_property_map =
    conditional_t<index_vertex_range<G>,
                  vector<T>,
                  unordered_map<vertex_id_t<G>, T>>;
```

Four helper functions complete the interface:

Function	Description
`make_vertex_property_map(g, init)`	Eager factory: creates a map with every vertex pre-populated to `init`. O(V).
`make_vertex_property_map<G,T>(g)`	Lazy factory: creates an empty map with capacity reserved. O(V) for index, O(1) for mapped.
`vertex_property_map_contains(m, uid)`	Returns `true` if `uid` has an entry (`uid < size(m)` for `vector`; `m.contains(uid)` for `unordered_map`).
`vertex_property_map_get(m, uid, dflt)`	Returns the stored value or `dflt` if absent. Does not insert.

Table 12: Vertex Property Map Functions

Two further utilities let algorithms accept a caller-supplied property map without caring which underlying container backs it:

```
// Per-vertex value type of a property-map container:
// vector<T> yields T (value_type); unordered_map<K,V> yields V (mapped_type)
template <class Container>
using vertex_property_map_value_t = /* see prose */;

// A container usable as a per-vertex property map for graph G:
// it can be subscripted by the graph's vertex id.
template <class M, class G>
concept vertex_property_map_for =
    requires(M& m, const vertex_id_t<G>& uid) {
      { m[uid] } -> convertible_to<vertex_property_map_value_t<M>>;
    };
```

`vertex_property_map_value_t<Container>` extracts the stored value type from either container shape (the `mapped_type` of an `unordered_map`, or the `value_type` of a `vector`). `vertex_property_map_for<M,G>` is the precise constraint for algorithm parameters such as *Distances* or *Predecessors*: it requires only that `m[uid]` is valid for the graph's `vertex_id_t<G>`, since algorithms subscript such maps by vertex id rather than iterating them as a range. Both `vector<T>` (index graphs) and `unordered_map<vertex_id_t<G>,T>` (mapped graphs) satisfy it.

6 Edgelist Interface

An edgelist is a range of values where we can get the source_id and target_id, and an optional edge_value. It is similar to edges in an adjacency list or edges in the incidence view, but is a distinct range of values that

are separate from the others.

Like the adjacency list, the edgelist has default implementations that use the standard library for simple implementations out of the box. It's also able to easily adapt to externally defined edge types by overriding the `source_id(el,uv)`, `target_id(el,uv)` and `edge_value(el,uv)` functions.

6.1 Namespace

Edge list concepts, type aliases and CPOs are defined in the `std::graph::edge_list` namespace. Unlike the adjacency list, these symbols are *not* re-exported into `std::graph` because certain names have distinct meanings in each context — most notably `vertex_id_t<G>` is derived from `vertex_id(g,u)` for an adjacency list but from `source_id(el,uv)` for an edge list. Bringing both into the root namespace would create an irresolvable ambiguity.

`graph::adj_list` and `graph::edge_list` are *peer* namespaces: neither is a subset or refinement of the other. An adjacency list provides per-vertex edge ranges and supports fast neighbour traversal; an edge list is a flat range of `(source, target [, value])` tuples suited to edge-centric algorithms such as Kruskal's MST or bulk construction. CPOs that operate directly on edge data — `source_id`, `target_id` and `edge_value` — are shared and live in `std::graph`, as is the `basic_edge` concept built on them, which both the adjacency-list `edge` concept and the edge-list `basic_sourced_edgelist` concept refine.

6.2 Concepts

The edgelist concepts use a 2-argument form `f(el, uv)` where `el` is the edgelist and `uv` is the edge value, matching the adjacency list CPO convention. All edgelist CPOs require both the container and the element to allow for container-specific dispatch.

`basic_sourced_edgelist<EL>` is the base concept: an input range whose element type is not itself a range, whose element satisfies the shared `basic_edge` concept (`source_id(el,uv)` and `target_id(el,uv)`). This is the same `basic_edge` floor required by the adjacency-list `edge` concept, tying the two abstract data types together at the edge level. `basic_sourced_index_edgelist<EL>` additionally requires both ids to be integral. `has_edge_value<EL>` refines `basic_sourced_edgelist` to also provide `edge_value(el,uv)`.

```
// For exposition only
namespace std::graph::edge_list {

template <class EL>
concept basic_sourced_edgelist =
    ranges::input_range<EL> &&
    !ranges::range<ranges::range_value_t<EL>> &&
    basic_edge<EL, ranges::range_value_t<EL>>; // shared edge floor:  source_id + target_id

template <class EL>
concept basic_sourced_index_edgelist =
    basic_sourced_edgelist<EL> &&
    requires(EL& el, ranges::range_value_t<EL> uv) {
      { source_id(el, uv) } -> integral;
      { target_id(el, uv) } -> integral;
    };

template <class EL>
concept has_edge_value =
    basic_sourced_edgelist<EL> &&
    requires(EL& el, ranges::range_value_t<EL> uv) {
      { edge_value(el, uv) };
    };

} // namespace std::graph::edge_list
```

6.3 Traits

Table 13 summarizes the type traits in the Edgelist Interface, allowing views and algorithms to query the graph's characteristics.

Trait	Type	Comment

Table 13: Graph Container Interface Type Traits

6.4 Types

Table 14 summarizes the type aliases in the Edgelist Interface.

The type aliases are defined by either a function specialization for the edgelist implementation, or a refinement of one of those types (e.g. an iterator of a range). Table 15 describes the functions in more detail.

`edge_value(el,uv)` can be optionally implemented, depending on whether or not the edgelist has values on the edge types.

Type Alias	Definition	Comment
`edge_range_t<EL>`	`EL`	
`edge_iterator_t<EL>`	`iterator_t<edge_range_t<EL>>`	
`edge_t<EL>`	`range_value_t<edge_range_t<EL>>`	
`edge_value_t<EL>`	`decltype(edge_value(el,uv))`	optional
`vertex_id_t<EL>`	`remove_cvref_t<decltype(source_id(el,uv))>`	
raw-vertex-id-type	`decltype(source_id(el,uv))`	exposition only

Table 14: Edgelist Interface Type Aliases

6.5 Functions

Table 15 shows the functions available in the Edgelist Interface. Unlike the adjacency list, `source_id(el,uv)` is always available.

Function	Return Type	Cmplx	Default Implementation
`target_id(el,uv)`	`vertex_id_t<EL>`	constant	(see below)
`source_id(el,uv)`	`vertex_id_t<EL>`	constant	(see below)
`edge_value(el,uv)`	`edge_value_t<EL>`	constant	optional, see below
`contains_edge(el,uid,vid)`	`bool`	linear	`find_if(el, [](auto&& uv){ return source_id(el,uv)==uid && target_id(el,uv)==vid; } )`
`num_edges(el)`	`integral`	constant	`size(el)`
`has_edges(el)`	`bool`	constant	`num_edges(el)>0`

Table 15: Edgelist Interface Functions

The `contains_edge(el,uid,vid)`, `num_edges(el)` and `has_edges(el)` functions are not yet in the reference implementation; the `source_id`, `target_id` and `edge_value` CPOs are available now.

6.6 Determining the source_id, target_id and edge_value types

Special patterns are recognized for edges based on the `pair`, `tuple` and `edge_data` types. When they are used the `source_id(el,uv)`, `target_id(el,uv)` and `edge_value(el,uv)` functions will be defined automatically.

The `pair` pattern is

— `pair<integral,integral>` for `source_id(el,uv)` and `target_id(el,uv)` respectively.

The `tuple` patterns are

— `tuple<integral,integral>` for `source_id(el,uv)` and `target_id(el,uv)` respectively.

— `tuple<integral,integral,scalar>` for `source_id(el,uv)`, `target_id(el,uv)` and `edge_value(el,uv)` respectively.

The `edge_data` patterns are

— `edge_data<VId,true,void,void>` with `source_id(el,uv)` and `target_id(el,uv)`.

— `edge_data<VId,true,void,EV>` with `source_id(el,uv)`, `target_id(el,uv)` and `edge_value(el,uv)`.

— `edge_data<VId,true,E&,void>` with `source_id(el,uv)`, `target_id(el,uv)` and an edge descriptor reference.

— `edge_data<VId,true,E&,EV>` with `source_id(el,uv)`, `target_id(el,uv)`, an edge descriptor reference and `edge_value(el,uv)`.

In all other cases the functions will need to be overridden for the edge type.

7 Using Existing Data Structures

Reasonable defaults have been defined for the adjacency list and edgelist functions to minimize the amount of work needed to adapt existing data structures to be used by the views and algorithms.

This section specifies the structural patterns the GCI CPOs recognize automatically — the *normative trigger* that determines when no function overrides are required. The companion Graph Containers proposal catalogs the concrete standard containers that match these patterns, their performance trade-offs, and worked examples; see that paper for the catalog and usage.

7.1 Recognized Vertex Patterns

When a graph type `G` is itself a `forward_range` whose elements are also ranges, the GCI CPOs detect and adapt to the storage strategy automatically. Two broad categories of vertex container are recognized, each producing a different descriptor and vertex-id strategy.

7.1.1 Random-Access Vertex Pattern

When `G` satisfies `random_access_range` and its elements satisfy `forward_range`, the vertices are assumed to reside in a contiguous or random-access container such as `std::vector` or `std::deque`.

— `vertex_id_t<G>` defaults to an integral type (index into the range).

— `find_vertex(g, uid)` uses indexed access: `begin(vertices(g))+ uid`, giving $\mathcal{O}(1)$ lookup.

— `out_edges(g, u)` returns the inner range at that index.

7.1.2 Associative Vertex Pattern

When `G` is an associative container (`std::map` or `std::unordered_map`) whose mapped values are forward ranges, vertices are keyed by an arbitrary id type rather than a dense index.

— `vertex_id_t<G>` defaults to the `key_type` of the container.

— `find_vertex(g, uid)` uses `g.find(uid)`, giving $\mathcal{O}(\log |V|)$ for `map` or amortized $\mathcal{O}(1)$ for `unordered_map`.

— `out_edges(g, u)` extracts the `.second` member (the mapped edge range) from the key-value pair.

7.2 Recognized Edge Patterns

The edge range returned by `out_edges(g, u)` is the inner range stored at (or referenced by) each vertex. Any range satisfying `forward_range` whose element type matches a recognized edge element pattern (below) is accepted automatically. The library does *not* maintain a fixed list of containers; it relies on concept-based detection, so non-standard containers (e.g. `boost` containers) work identically.

For associative edge containers whose `value_type` is a `pair<const Key, Mapped>` (i.e. `map` and `unordered_map`), the key is treated as the target vertex id and the mapped type as the edge value: `target_id(g,uv)` returns `.first` and `edge_value(g,uv)` returns `.second`. For set-like containers (`set`, `unordered_set`) each element is the edge itself, and the element-level rules below apply.

7.2.1 Edge Element Patterns

Within each vertex's edge range the GCI recognizes the following element forms and automatically provides `target_id(g,uv)` and, where applicable, `edge_value(g,uv)`:

Edge Element Type	Automatic CPOs
`integral`	`target_id(g,uv)` returns the value directly. No `edge_value`.
`pair<integral, EV>`	`target_id(g,uv)` returns `.first`; `edge_value(g,uv)` returns `.second`.
`tuple<integral, ...>`	`target_id(g,uv)` returns `get<0>(uv)`; remaining elements accessible via `edge_value`.
custom struct	The user overrides `target_id(g,uv)` and optionally `edge_value(g,uv)` for the type.

Table 16: Recognized Edge Element Patterns

These vertex and edge patterns combine freely: any recognized vertex container can hold any recognized edge container, and any recognized edge container can hold any recognized edge element type. When the element type does not match a recognized pattern, the user must override the appropriate CPOs for the graph type. The Graph Containers proposal enumerates the standard containers in each category and their trade-offs.

Acknowledgements

Phil Ratzloff's time was made possible by SAS Institute.

Portions of *Andrew Lumsdaine's* time was supported by NSF Award OAC-1716828 and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

Muhammad Osama's time was made possible by Advanced Micro Devices, Inc.

The authors thank the members of SG19 and SG14 study groups for their invaluable input.