Graph Library: Graph Container Interface

Document #: **P3130r3**
Date: 2025-04-13
Project: Programming Language C++
Audience: Library Evolution
SG19 Machine Learning
SG14 Game, Embedded, Low Latency
Revises: P3130r2

Reply-to: Phil Ratzloff (SAS Institute)
phil.ratzloff@sas.com
Andrew Lumsdaine
lumsdaine@gmail.com

Contributors: Kevin Deweese
Muhammad Osama (AMD, Inc)
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	In process	**Comparison to other graph libraries** on performance and usage syntax. Not published yet.

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.

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>`		Vertex descriptor
	`vertex_reference_t<G>`	`u,v`	Vertex descriptor reference. `u` is the source (or only) vertex. `v` is the target vertex.
`VId`	`vertex_id_t<G>`	`uid,vid,seed`	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(u)` → vertex value, or `vvf(uid)` → vertex value, depending on requirements of the consuming algorithm or view.
`VProj`		`vproj`	Vertex info projection function: `vproj(u)` → `vertex_info<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>`		Edge descriptor
	`edge_reference_t<G>`	`uv,vw`	Edge descriptor reference. `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(uv)` → edge value.
`EProj`		`eproj`	Edge info projection function: `eproj(uv)` → `edge_info<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 an adacency lists (aka graph) and edgelists, no matter their internal design and organization. For instance, an adjacency list can be a vector of lists from standard containers, CSR-based graph and adjacency matrix. Likewise, an edgelist can be 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.

If there is a desire to use the algorithms against externally defined data structures, the GCI exposes is functions as customization points to be overridden as needed. Likewise, externally defined algorithms can be used to operate on other data structures that meet the GCI requirements. This achieves the same goals as the STL, where algorithms can be used on any container that meets the requirements of the algorithm.

The GCI is designed to support a wider scope of graph containers than required by the views and algorithms in this proposal. This enables for future growth of the graph data model (e.g. incoming edges on a vertex), or as a framework for graph implementations outside of the standard. For instance, existing implementations may have requirements that cause them to define features with looser constraints, such as sparse vertex_ids, non-integral vertex_ids, or storing vertices in associative bi-directional containers (e.g. std::map or std::unordered_map).

Such features require specialized implementations for views and algorithms. The performance for such algorithms will be sub-optimal, but may be preferrable to run them on the existing container rather than loading the graph into a high-performance graph container and then running the algorithm on it, where the loading time can far outweigh the time to run the sub-optimal algorithm. To achieve this, care has been taken to make sure that the use of concepts chosen is appropriate for algorithm, view and container.

All algorithms in this and related proposals require that adjacency list vertices are stored in random access containers and that `vertex_id_t<G>` is integral. Future designs may relax these requirements, but for now they are required.

5 Adjacency List Interface

5.1 Concepts

This section describes the concepts to describe the adjacency lists used for graphs in the Graph Library. There are a couple of qualifiers that are used in concept names.

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

— **sourced** where an edge has a source id.

While we belive 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.

5.1.1 Edge Concepts

The types of edges that can occur in a graph are described with the edges concepts.

```
// For exposition only

template <class G>
concept targeted_edge = requires(G&& g, edge_reference_t<G> uv) {
                           target(g, uv);
                           target_id(g, uv);
                        };

template <class G>
concept sourced_edge = requires(G&& g, edge_reference_t<G> uv) {
                        source(g, uv);
                        source_id(g, uv);
```

```
                  };

template <class G>
concept sourced_targeted_edge = targeted_edge<G> && sourced_edge<G>;
```

Return types are not validated in order to provide flexibility, and because it offers little value. Let's look at the options using `target_id(g,uv)` as an example.

```
    { target_id(g,uv) } -> integral;
```

This may seem obvious on first glance to some, but doing so limits us to integral ids. Graphs can use non-integral vertex id types for vertices stored in a `map` or `unordered_map`. It can be more efficient to simply run an algorithm on the existing graph rather than to copy it into a "high performance" graph data structure just to run the algorithm because the copying operation can far outweigh the cost of running the algorithm on the native data structures, even when those data structures offer $\mathcal{O}(log(n))$ lookup on vertices. While we're not proposing algorithms that can do this today, the library needs to keep the door open to such algorithms in the future as well as supporting such algorithms outside the standard library.

```
    { target_id(g,uv) } -> vertex_id_t<G>;
```

This is better than the previous example. It doesn't require an integral vertex id and maintains the integrity of the expected type. A problem with this is that if it fails, the error reported will be something like "doesn't meet concept requirements" which is obscure and takes time by the user to understand and resolve.

```
    target_id(g,uv);
```

The final option allows the compiler to report a regular error or warning if the returned value isn't what's expected in the context it's used because the types are included in the error message, making it easier to understand what the problem is. Additionally, functions aren't distiguished by their return type, so there's little value in attempting to check it in this case.

There is precedent for this design choice of not validating the return type, as can be seen in the `sized_range` concept.

5.1.1.1 Edge Range Concepts

There is a single edge range concept.

```
// For exposition only

template <class G>
concept targeted_edge_range =
        basic_targeted_edge_range<G> &&
        requires(G&& g, vertex_reference_t<G> u, vertex_id_t<G> uid) {
          { edges(g, u) } -> ranges::forward_range;
          { edges(g, uid) } -> ranges::forward_range; // implies call to find_vertex(g,uid)
        };
```

5.1.2 Vertex Concepts

The `vertex_range` concept is the general definition used for adjacency lists while `index_vertex_range` is used for high performance graphs where vertices typically stored in a `vector`.

```
// For exposition only

template <class G>
concept _common_vertex_range = ranges::sized_range<vertex_range_t<G>> &&
                               requires(G&& g, vertex_iterator_t<G> ui) { vertex_id(g, ui); };
```

```
template <class G>
concept vertex_range = _common_vertex_range<vertex_range_t<G>> &&
                         ranges::forward_range<vertex_range_t<G>>;

template <class G>
concept index_vertex_range = _common_vertex_range<vertex_range_t<G>> &&
                               ranges::random_access_range<vertex_range_t<G>> &&
                               integral<vertex_id_t<G>>;
```

5.1.3 Adjacency List Concepts

The adjacency list concepts bring together the vertex and edge concepts used for core graph concepts. All algorithms initially proposed for the Graph Library use the `index_adjacency_list` . Future proposals may use the `adjacency_list` concept which allows for vertices in associative containers.

```
// For exposition only

template <class G>
concept adjacency_list = vertex_range<G> && //
                         targeted_edge_range<G> && //
                         targeted_edge<G>;

template <class G>
concept index_adjacency_list = index_vertex_range<G> && //
                               targeted_edge_range<G> && //
                               targeted_edge<G>;

template <class G>
concept sourced_adjacency_list = vertex_range<G> && //
                                 targeted_edge_range<G> && //
                                 sourced_targeted_edge<G>;

template <class G>
concept sourced_index_adjacency_list = index_vertex_range<G> && //
                                       targeted_edge_range<G> && //
                                       sourced_targeted_edge<G>;
```

5.2 Traits

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

5.3 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.

Trait	Type	Comment
has_degree<G>	concept	Is the degree(g,u) function available?
has_find_vertex<G>	concept	Are the find_vertex(g,_) functions available?
has_find_vertex_edge<G>	concept	Are the find_vertex_edge(g,_) functions available?
has_contains_edge<G>	concept	Is the contains_edge(g,uid,vid) function available?
define_unordered_edge<G> : false_type	struct	Specialize to derive from true_type for a graph with unordered edges
unordered_edge<G>	concept	
ordered_edge<G>	concept	
define_adjacency_matrix<G> : false_type	struct	Specialize for graph implementation to derive from true_type for edges stored as a square 2-dimensional array
is_adjacency_matrix<G>	struct	
is_adjacency_matrix_v<G>	type alias	
adjacency_matrix<G>	concept	

Table 3: Graph Container Interface Type Traits

Type Alias	Definition	Comment
graph_reference_t<G>	add_lvalue_reference<G>	
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>>	
vertex_reference_t<G>	range_reference_t<vertex_range_t<G>>	
vertex_id_t<G>	decltype(vertex_id(g,u))	
vertex_value_t<G>	decltype(vertex_value(g,u))	optional
vertex_edge_range_t<G>	decltype(edges(g,u))	
vertex_edge_iterator_t<G>	iterator_t<vertex_edge_range_t<G>>	
edge_t<G>	range_value_t<vertex_edge_range_t<G>>	
edge_reference_t<G>	range_reference_t<vertex_edge_range_t<G>>	
edge_value_t<G>	decltype(edge_value(g,uv))	optional
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

5.4 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 belive 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.

5.5 Functions

Tables 5, 6 and 7 summarize the primitive functions in the Graph Container Interface. used to access an adacency graph, 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, CSR-based graph and adjacency matrix.

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(edges(g,u)); return n;
has_edge(g)	bool	|V|	for(u: vertices(g))if !empty(edges(g, u))return true; return false;
num_partitions(g)	integral	constant	1
vertices(g,pid)	partition_vertex_range_t<G>	constant	vertices(g)
num_vertices(g,pid)	integral	constant	size(vertices(g))

Table 5: Graph Functions

The complexity shown above for num_edges(g) and has_edge(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.

Function	Return Type	Complexity	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)	vetex_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
vertex_value(g,uid)	vertex_value_t<G>	constant	vertex_value(g,*find_vertex(g,uid)) , optional
degree(g,u)	integral	constant	size(edges(g,u)) if sized_range<vertex_edge_range_t<G>>
degree(g,uid)	integral	constant	size(edges(g,uid)) if sized_range<vertex_edge_range_t<G>>
edges(g,u)	vertex_edge_range_t<G>	constant	u if forward_range<vertex_t<G>> , n/a otherwise
edges(g,uid)	vertex_edge_range_t<G>	constant	edges(g,*find_vertex(g,uid))
partition_id(g,u)	partition_id_t<G>	constant	
partition_id(g,uid)	partition_id_t<G>	constant	

Table 6: Vertex Functions

The default implementation for the degree functions assumes that vertex_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.

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

Function	Return Type	Complexity	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<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_vertex_edge(g,u,vid)`	`vertex_edge_t<G>`	linear	`find(edges(g,u), [](uv)target_id(g,uv)== vid;})`
`find_vertex_edge(g,uid,vid)`	`vertex_edge_t<G>`	linear	`find_vertex_edge(g,*find_vertex(g,uid),vid )`
`contains_edge(g,uid,vid)`	`bool`	constant	`uid < size(vertices(g))&& vid < size( vertices(g))` if `is_adjacency_matrix_v<G>` .
		linear	`find_vertex_edge(g,uid)!= end(edges(g,uid) )` otherwise.
The following are only available when the optional `source_id(g,uv)` is defined for the edge			
`source_id(g,uv)`	`vertex_id_t<G>`	constant	n/a, optional
`source(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ source_id(g,uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`

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.

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 scaler, 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.

5.6 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.

5.7 Vertex and Edge Descriptor Views

The ranges returned by `vertices(g)` and `edges(g,u)` are views of their respective underlying container in the adjacency list. The value_type of the view is a descriptor, which refers to an object in the underlying range.

Descriptors are opaque, abstract objects that represent vertices and edges in an adjacency list. They are particularly useful because they abstract away the implementation details of the adjacency list, allowing a user to work with different graph types in a consistent manner.

A benefit of using descriptors is that it reduces the number of functions and concepts needed for the GCI compared to previous designs. Without them, an iterface would require additional functions and concepts, and algorithms would need to be specialized for vertices stored in random-access containers compared to associative containers.

Practically, a descriptor is either an integral index or an iterator, depending on the underlying container of vertices or edges. For example, a descriptor for a vertex in a `vector` is an index, while the descriptor for an edge in a `list` is an iterator. Looking forward to the future, beyond this proposal, a descriptor for a vertex in a `map` or `unordered_map` would use an iterator; the door is opened for future expansion with minimal impact to the GCI.

The vertex and edge descriptors are defined as the `vertex_t<G>` and `edge_t<G>` types, respectively. The following are characteristics of descriptors:

— Equality comparison: Descriptors can be compared for equality using the `==` and `!=` operators.

— Ordering: Descriptors can be ordered using the `<` , `<=` , `>` , and `>=` operators, if supported by the iterators in underlying container being used.

— Copy and assignment: Descriptors can be copied and assigned, ensuring they can be used in standard algorithms and containers.

— Default construction: Descriptors can be default-constructed, though the resulting value are not guaranteed to represent a valid vertex or edge.

In addition, a descriptor has an `inner_value()` member function that returns a reference to the underlying value in the underlying container. This is only needed for overriding the customization points when you're adapting your own graph container to the GCI.

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 `edges(g,u)` . `descriptor_subrange_view` is used when edges for multiple vertices are stored in a single container, such as a CSR or adjacency matrix (e.g. `compressed_graph`).

```
template <forward_range R>
constexpr auto descriptor_view(R&& r);

template <forward_range R>
using descriptor_view_t = decltype(descriptor_view(declval<R>()));

template <forward_range R>
constexpr auto descriptor_subrange_view(R&& rng, R&& subrng);

template <forward_range R>
```

```
using descriptor_subrange_view_t = decltype(descriptor_subrange_view(declval<R>(), declval<R>()));
```

5.8 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.

`edges(g,uid,pid)` and `edges(g,u,pid)` filter the edges where the target is in the partition `pid` passed. This isn't needed for bipartite graphs.

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 implmentations out of the box. It's also able to easily adapt to externally defined edge types by overriding the `source_id(e)` , `target_id(e)` and `edge_value(e)` functions.

6.1 Namespace

The concepts and types for the edgelist are defined in the `std::graph::edgelist` namespace to avoid conflicts with the adjacency list.

6.2 Concepts

The concepts for edgelists follow the same naming conventions as the adjacency lists.

```
template <class EL> // For exposition only
concept sourced_edgelist = ranges::input_range<EL> &&
                           !ranges::range<ranges::range_value_t<EL>> &&
                           requires(ranges::range_value_t<EL> e) {
                             { source_id(e) };
                             { target_id(e) } -> same_as<decltype(source_id(e))>;
                           };

template <class EL> // For exposition only
concept sourced_index_edgelist = sourced_edgelist<EL> &&
```

```
                                requires(ranges::range_value_t<EL> e) {
                                  { source_id(e) } -> integral;
                                  { target_id(e) } -> integral; // redundant, for clarity
                                };

template <class EL> // For exposition only
concept has_edge_value = sourced_edgelist<EL> && //
                         requires(ranges::range_value_t<EL> e) {
                           { edge_value(e) };
                         };
```

6.3 Traits

Table 8 summarizes the type traits in the Edgelist Interface, allowing views and algorithms to query the graph's characteristics.

Trait	Type	Comment
`is_directed<EL> : false_type` `is_directed_v<EL>`	struct	When specialized for an edgelist to derive from `true_type` , it may be used during graph construction to add a second edge with source_id and target_id reversed.

Table 8: Graph Container Interface Type Traits

6.4 Types

Table 9 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 10 describes the functions in more detail.

`edge_value(g,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_reference_t<EL>`	`range_reference_t<edge_range_t<EL>>`	
`edge_value_t<EL>`	`decltype(edge_value(e))`	optional
`vertex_id_t<EL>`	`decltype(target_id(e))`	

Table 9: Edgelist Interface Type Aliases

6.5 Functions

Table 10 shows the functions available in the Edgelist Interface. Unlike the adjacency list, `source_id(e)` is always available.

6.6 Determining the source_id, target_id and edge_value types

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

The `tuple` patterns are

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

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

Table 10: Edgelist Interface Functions

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

The `edge_info` patterns are

— `edge_info<integral,true,void,void>` with `source_id(e)` and `target_id(e)` .

— `edge_info<integral,true,void,scaler>` with `source_id(e)` , `target_id(e)` and `edge_value(e)` .

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.

Useful defaults have been created using types and containers in the standard library, with the ability to override them for external data structures. This is described in more detail in the paper for Graph Library Containers.

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.