Graph Library: Graph Containers

Document #:	**P3131r3**
Date:	2025-04-13
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency
Revises:	P3131r2

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

P3131r0

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

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

— GCI overloads are no longer required for adjacency lists constructed with standard containers. Data structures that follow the pattern `random_access_range<forward_range<integral>>` and `random_access_range<forward_range<tuple<integral,...>>>` are automatically recognized as an adjacency list, including containers from non-standard libraries. The `integral` value is used as the target_id.

P3131r1

— Added feature summary of `compressed_graph` beyond the typical CSR implementation.

— Added complexity for `num_edges(g)` and `has_edge(g)` functions in `compressed_graph`.

— Add constructors to `compressed_graph` to complement the removal of the load functions from P3130r1 Graph Container Interface. An optional `partition_start_ids` parameter is also included.

P3131r2

— Add the edgelist as an abstract data structure as a peer to the adjacency list. A section on edgelists has been added to Using Existing Data Structures.

— Add `is_directed` item in the feature summary box to `compressed_graph`.

P3131r3

— Changes related to boost::graph-like descriptors as described in P3130r3 Graph Container Interface. Related changes that affect this paper include:

  — Rename `descriptor` structs to `info` structs for new boost::graph-like descriptors.

  — Remove `copyable_vertex` and `copyable_edge` from the `compressed_graph` `requires` clause because decriptors are always 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 compressed_graph Graph Container

`compressed_graph` is a graph container being proposed for the standard library. It is a high-performance data structure that uses Compressed Sparse Row (CSR) format to store its vertices, edges and associated values. Once constructed, vertices and edges cannot be added or deleted but values on vertices and edges can be modified.

There are a number of features added beyond the typical CSR implmentation:

— **User-defined values** The typical CSR implementation stores values on edges (columns) by defining the `EV` template paraemter. `compressed_graph` extends that to also allow values on vertices (rows) and the graph itself by defining the `VV` and `GV` template arguments respectively. If a type is void, no memory overhead is incurred.

— **Index type sizes** The size of the integral indexes into the internal vertex (row) and edge (column) structures can be controlled by the `VId` and `EIndex` template arguments respectively to give a balance between capacity, memory usage and performance.

— **Multi-partite graphs** The vertices can optionally be partitioned into multiple partitions by passing the starting vertex id of each partition in the `partition_start_ids` argument in the constructors. If no partitions are specified, the graph is single-partite.

The listings in the following sections show the prototypes for the `compressed_graph` when the graph value type `GV` is non-`void` (section 4.1) and a class template specialization when it is `void` (section 4.2).

Only the constuctors and destructor shown for `compressed_graph` are public. All other types and functions related to the graph are only accessible through the types and functions in the Graph Container Interface.

vertex_id assignment: Contiguous **Vertices range:** Contiguous **Edge range:** Contiguous	`has_edge(g)` $O(1)$ `num_edges(g)` $O(1)$ `partition_id(g,uid)` $O(log(P+1))$	**Append vertices?** No **Append edges?** No **Partions?** Yes **is_directed?** No

P is the number of partitions and is exepected to be small, e.g. $P = 2$ for bipartite and $P \leq 10$ for typical multi-partite graphs.

The `is_directed` trait is not supported. If `compressed_graph` is intended to be used for an undirected graph, then the edge pairs must be included for both directions, (uid,vid) and (vid,uid), when constructing the graph.

4.1 compressed_graph when `GV` is not `void`

```
template <class EV,
         class VV,
         class GV,
         integral VId=uint32_t,
         integral EIndex=uint32_t,
         class Alloc=allocator<VId>>
class compressed_graph {
public: // Construction/Destruction/Assignment
  constexpr compressed_graph() = default;
  constexpr compressed_graph(const compressed_graph&) = default;
  constexpr compressed_graph(compressed_graph&&) = default;
  constexpr ~compressed_graph() = default;

  constexpr compressed_graph& operator=(const compressed_graph&) = default;
  constexpr compressed_graph& operator=(compressed_graph&&) = default;

  // compressed_graph( alloc)
  // compressed_graph(gv&, alloc)
  // compressed_graph(gv&&, alloc)
```

```
constexpr compressed_graph(const Alloc& alloc);
constexpr compressed_graph(const graph_value_type& value, const Alloc& alloc = Alloc());
constexpr compressed_graph(graph_value_type&& value, const Alloc& alloc = Alloc());

// compressed_graph(erng, eprojection, alloc)
// compressed_graph(gv&, erng, eprojection, alloc)
// compressed_graph(gv&&, erng, eprojection, alloc)

template <ranges::forward_range ERng, ranges::forward_range PartRng, class EProj = identity>
requires convertible_to<ranges::range_value_t<PartRng>, VId>
constexpr compressed_graph(const ERng& erng,
                           EProj eprojection,
                           const PartRng& partition_start_ids = vector<VId>(),
                           const Alloc& alloc = Alloc());

template <ranges::forward_range ERng, ranges::forward_range PartRng, class EProj = identity>
requires convertible_to<ranges::range_value_t<PartRng>, VId>
constexpr compressed_graph(const graph_value_type& value,
                           const ERng& erng,
                           EProj eprojection,
                           const PartRng& partition_start_ids = vector<VId>(),
                           const Alloc& alloc = Alloc());

template <ranges::forward_range ERng, ranges::forward_range PartRng, class EProj = identity>
requires convertible_to<ranges::range_value_t<PartRng>, VId>
constexpr compressed_graph(graph_value_type&& value,
                           const ERng& erng,
                           EProj eprojection,
                           const PartRng& partition_start_ids = vector<VId>(),
                           const Alloc& alloc = Alloc());

// compressed_graph(erng, vrng, eprojection, vprojection, alloc)
// compressed_graph(gv&, erng, vrng, eprojection, vprojection, alloc)
// compressed_graph(gv&&, erng, vrng, eprojection, vprojection, alloc)

template <ranges::forward_range ERng,
          ranges::forward_range VRng,
          ranges::forward_range PartRng,
          class EProj = identity,
          class VProj = identity>
requires convertible_to<ranges::range_value_t<PartRng>, VId>
constexpr compressed_graph(const ERng& erng,
                           const VRng& vrng,
                           EProj eprojection = {},
                           VProj vprojection = {},
                           const PartRng& partition_start_ids = vector<VId>(),
                           const Alloc& alloc = Alloc());

template <ranges::forward_range ERng,
          ranges::forward_range VRng,
          ranges::forward_range PartRng,
          class EProj = identity,
          class VProj = identity>
requires convertible_to<ranges::range_value_t<PartRng>, VId>
constexpr compressed_graph(const graph_value_type& value,
                           const ERng& erng,
                           const VRng& vrng,
                           EProj eprojection = {},
                           VProj vprojection = {},
```

```
                          const PartRng& partition_start_ids = vector<VId>(),
                          const Alloc& alloc = Alloc());

  template <ranges::forward_range ERng,
            ranges::forward_range VRng,
            ranges::forward_range PartRng,
            class EProj = identity,
            class VProj = identity>
  requires convertible_to<ranges::range_value_t<PartRng>, VId>
  constexpr compressed_graph(graph_value_type&& value,
                          const ERng& erng,
                          const VRng& vrng,
                          EProj eprojection = {},
                          VProj vprojection = {},
                          const PartRng& partition_start_ids = vector<VId>(),
                          const Alloc& alloc = Alloc());

  constexpr compressed_graph(const initializer_list<copyable_edge_t<VId, EV>>& ilist,
                          const Alloc& alloc = Alloc());
};
```

4.2 compressed_graph specialization when `GV` is `void`

When `GV` is void the number of constructors decreases significantly as shown in the following listing.

```
template <class EV,
          class VV,
          integral VId=uint32_t,
          integral EIndex=uint32_t,
          class Alloc=allocator<VId>>
template <class EV, class VV, integral VId, integral EIndex, class Alloc>
class compressed_graph<EV, VV, void, VId, EIndex, Alloc>
public: // Construction/Destruction
  constexpr compressed_graph() = default;
  constexpr compressed_graph(const compressed_graph&) = default;
  constexpr compressed_graph(compressed_graph&&) = default;
  constexpr ~compressed_graph() = default;

  constexpr compressed_graph& operator=(const compressed_graph&) = default;
  constexpr compressed_graph& operator=(compressed_graph&&) = default;

  // edge-only construction
  template <ranges::forward_range ERng,
            class EProj = identity,
            ranges::forward_range PartRng = vector<VId>>
  requires copyable_edge<invoke_result_t<EProj, ranges::range_value_t<ERng>>, VId, EV>
  constexpr compressed_graph(const ERng& erng,
                          EProj eprojection = identity(),
                          const PartRng& partition_start_ids = vector<VId>(),
                          const Alloc& alloc = Alloc())

  // edge and vertex value construction
  template <ranges::forward_range ERng,
            ranges::forward_range VRng,
            ranges::forward_range PartRng,
            class EProj = identity,
            class VProj = identity>
  constexpr compressed_graph(const ERng& erng,
                          const VRng& vrng,
```

```
                        EProj eprojection = {},
                        VProj vprojection = {},
                        const PartRng& partition_start_ids = vector<VId>(),
                        const Alloc& alloc = Alloc());

  // initializer list using edge_info<VId,true,void,EV>
  constexpr compressed_graph(const initializer_list<copyable_edge_t<VId, EV>>& ilist,
                        const Alloc& alloc = Alloc());
};
```

4.3 compressed_graph description

1 *Mandates:*

(1.1) — The `EV` template argument for an edge value must be a copyable type or `void`.

(1.2) — The `VV` template argument for a vertex value must be a copyable type or `void`.

(1.3) — When the `GV` template argument for a graph value is not `void` it can be movable or copyable. It must have a default constructor if it is not passed in a `compressed_graph` constructor.

(1.4) — The `EProj` template argument must be a projection that returns a value of `copyable_edge<VId, true, EV>` type given a value of `erng` . If the value type of `ERng` is already a `copyable_edge<VId, true, EV>` type, then `EProj` can be `identity` .

(1.5) — The `VProj` template argument must be a projection that returns a value of `copyable_vertex<VId, VV>` type, given a value of `vrng` . If the value type of `Vrng` is already a `copyable_vertex<VId, VV>` type, then `VProj` can be `identity` .

2 *Preconditions:*

(2.1) — The `VId` template argument must be able to store a value of |V|+1, where |V| is the number of vertices in the graph. The size of this type impacts the size of the *edges*.

(2.2) — The `EIndex` template argument must be able to store a value of |E|+1, where |E| is the number of edges in the graph. The size of this type impact the size of the *vertices*.

(2.3) — The `EProj` and `VProj` template arguments must be valid projections.

(2.4) — The `partition_start_ids` range includes the starting vertex id for each partition. If it is empty, then the graph is single-partite and the number of partitions is 1. If it is not empty, then the number of partitions is the size of the range, where the first element must be 0 and all elements are in ascending order. A vertex id in the range must not exceed the number of vertices in the graph. Any violation of these conditions results in undefined behavior.

3 *Effects:*

(3.1) — When `EV` , `VV` , or `GV` are `void` , no extra memory overhead is incurred for that type.

4 *Remarks:*

(4.1) — The `VId` and `EIndex` template arguments impact the capacity, internal storage requirements and performance. The default of `uint32_t` is sufficient for most graphs and provides a good balance between storage and performance.

The memory requirements are roughly,

$$|V| \times (sizeof(EIndex) + sizeof(VV)) + |E| \times (sizeof(VId) + sizeof(EV)) + sizeof(GV)$$

where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. *sizeof* `void` is 0 when considering *sizeof* for `VV` , `EV` , and `GV` . Alignment and overhead for internal vectors are not included in this calculation.

(4.2) — The allocator passed to constructors is rebound for different types used by different internal containers.

5 Using Existing Data Structures

Reasonable defaults have been defined for the adjacency list and edgelist using data structures and types in the standard library, with some adaptation for externally defined containers, out of the box that require no function overrides.

5.1 Adjacency List Data Structures

5.1.1 Using Standard Containers for an Adjacency List

When the graph is defined using standard containers, the GCI functions can be used without any function overrides.

For example this we'll use `G = vector<forward_list<tuple<int,double>>>` to define the graph, where `g` is an instance of `G`. `tuple<int,double>` defines the target_id and weight property respectively. We can write loops to go through the vertices, and edges within each vertex, as follows.

```
using G = vector<forward_list<tuple<int,double>>>;
auto weight = [&g](edge_t& uv) { return get<1>(uv); }

G g;
load_graph(g, ...); // load some data

// Using GCI functions
for(auto&& [u] : vertices(g)) {
  for(auto&& [vid, uv]: edges(g,u)) {
    auto w = weight(uv);
    // do something...
  }
}
```

`edge_t<G>` is defined as `tuple<int,double>` in the example; the only requirement is that the first element in the `tuple` is integral and is used as the target_id. The `edge_value` function is not defined, as it is assumed that algorithms will take a lambda to extract the value from the edge, if needed.

If all you need is the target_id without any values, you can use `G = vector<forward_list<int>>`. The `int` is used as the target_id. Again, the only requirement is that it be `integral`.

Note that no function override was required. More formally, the two patterns recognized are `random_access_range<forward_range<integral>>` and `random_access_range<forward_range<tuple<integral,...>>>`. This extends to any range type. For instance, boost::containers can be used just as easily as std containers.

Table 3 shows how the types are defined for the example above.

Function or Value	Concrete Type
`vertices(g)`	`vector<forward_list<tuple<int,double>>>` (when `random_access_range<G>`)
`u`	`forward_list<tuple<int,double>>`
`edges(g,u)`	`forward_list<tuple<int,double>>` (when `random_access_range<vertex_range_t<G>>`)
`uv`	`tuple<int,double>`
`edge_value(g,uv)`	`tuple<int,double>` (when `random_access_range<vertex_range_t<G>>`)
`target_id(g,uv)`	`integral`, when `uv` is either `integral` or `tuple<integral,...>`

Table 3: Types When Using Standard Containers

5.1.2 Using Other Graph Data Structures

For other graph data structures function overrides are required. Table 4 shows the common function overrides anticipated for most cases, keeping in mind that all functions can be overridden if the default implementation is not suitable. When they are defined they must be in the same namespace as the data structures.

Function	Comment
`vertices(g)`	
`edges(g,u)`	
`target_id(g,uv)`	
`edge_value(g,uv)`	If edges have value(s) in the graph
`vertex_value(g,u)`	If vertices have value(s) in the graph
`graph_value(g)`	If the graph has value(s)
When edges have the optional source_id on an edge	
`source_id(g,uv)`	
When the graph supports multiple partitions	
`num_partitions(g)`	
`partition_id(g,u)`	
`vertices(g,u,pid)`	

Table 4: Common CPO Function Overrides

5.2 Edgelist Data Structures

5.2.1 Using Standard Containers for an Edgelist

Like the adjacency list, an edgelist can be defined using standard containers and types without requiring any function overrides.

```
using EL = vector<tuple<int, int, double>>;
using E = std::ranges::range_value_t<EL>;
EL el{{1, 2, 11.1}, {1, 4, 22.2}, {2, 3, 3.33}, {2, 4, 4.44}};
for (auto&& e : el) {
  int sid = source_id(e);
  int tid = target_id(e);
  double val = edge_value(e);
}
```

An alternative is to use the `edge_info` used in this proposal. Notice that the only difference is the definition of the edgelist type. All other code is identical to the previous example.

```
using EL = vector<edge_info<int, true, void, double>>;
using E = std::ranges::range_value_t<EL>;
EL el{{1, 2, 11.1}, {1, 4, 22.2}, {2, 3, 3.33}, {2, 4, 4.44}};
for (auto&& e : el) {
  int sid = source_id(e);
  int tid = target_id(e);
  double val = edge_value(e);
}
```

While these examples show the optional edge_value that is a `double`, it can be omitted if the edges do not have values.

Type alias are in the namespace `std::graph::edgelist` to avoid conflicts with adjacency_list types.

5.2.2 Using Other Graph Data Structures

If you have different edge type not covered by the standard types, you can override the `source_id(e)` , `target_id(e)` and `edge_value(E)` functions for that type. The functions must be in the same namespace as the edge data structure you want to use.

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.