Graph Library: Views

Document #:	**P3129r2**
Date:	2025-04-13
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency
Revises:	P3129r1

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

P3129r0

- Split from P1709r5. Added *Getting Started* section.
- Removed allocator parameters on views, for consistency with existing views in the standard.

P3129r1

- Add the edgelist as an abstract data structure as a peer to the adjacency list. The range returned by `edgelist_view` adheres to the `basic_sourced_index_edgelist` concept, and to the `has_edge_value` concept if a `evf(uv)` function is passed. The same applies to all *sourced* versions of the BFS, DFS and Topological Sort views.

— Restore the allocator parameters on the DFS, BFS and Toplogical Sort views, based on feedback and by SG14/SG19 joint meeting.

— Add a note that we will be unable to support a freestanding graph library in this proposal because of the need for `stack`, `queue` and potential `bad_alloc` exception in many of the views.

— Rename `descriptor` structs to `info` structs in preparation for new BGL-like descriptors.

P3129r2

— Replace the use of *id* and *reference* with *descriptor*, leading to a simpler interace. It also creates a more flexible interface that can support associative containers in the future. The following changes were made:

  — The number of View functions has been halved because we no longer need separate functions that only have `id`, and another set that has has both `id` and `reference`. Only functions with `descriptor` are needed.

  — The `vertex_id` member has been removed from the `vertex_info` struct, and the `vertex` member can hold either an id or a descriptor, depending on the context it's used. The same changes have also been applied to the `edge_info` and `neighbor_info` structs.

  — The copyable info type aliases and concepts have been removed. `vertex_info`, `edge_info` and `neighbor_info` are always copyable because they no longer contain references.

  — See P3130 Graph Container Interface for more details about descriptors.

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 Introduction

The views in this paper provide common ways that algorithms use to traverse graphs. They are a simple as iterating through the set of vertices, or more complex ways such as depth-first search and breadth-first search. The also provide a consistent and reliable way to access related elements using the View Return Types, and guaranteeing expected values, such as that the target is really the target on unordered edges.

We are unable to support freestanding implementations in this proposal. Many of the views require a `stack` or `queue`, which are not available in a freestanding environment. Additionally, `stack` and `queue` require memory allocation which could throw a `bad_alloc` exception.

5 Info Structs (Return Types)

Views return one of the types in this section, providing a consistent set of value types for all graph data structures. They are templated so that the view can adjust the types of the members to be appropriate for its use. The three types, `vertex_info`, `edge_info` and `neighbor_info`, define the common data model used by algorithms.

The following examples show the general design and how it's used. The example focuses on vertexlist when iterating over vertices, and the same pattern applies with using the other view functions.

```
// the type of uu is vertex_info<vertex_t<G>, void>
for(auto&& uu : vertexlist(g)) {
  vertex_reference_t<G> u = uu.vertex;
  // ...  do something interesting
}
```

A function object can also be passed to return a value from the vertex. In this case, `vertexlist(g, vvf)` returns a struct with two members, `vertex` and `value`.

```
auto vvf = [&g](vertex_reference_t<G> u) { return vertex_value(g,u); };
// the type of uu is vertex_info<vertex_t<G>, decltype(vvf(u))>
for(auto&& uu : vertexlist(g, vvf)) {
    vertex_reference_t<G> u = uu.vertex;
    vertex_value_t<G>& value = uu.value;
    // ...  do something interesting
}
```

Structured bindings make it simpler.

```
for(auto&& [u] : vertexlist(g)) {
  // ...  do something interesting
}
```

Finally, using structured binding with the vertex value function.

```
// the type returned by vertexlist is vertex_info<vertex_t<G>, decltype(vvf(vertex_t<G>))>
auto vvf = [&g](vertex_reference_t<G> u) { return vertex_value(g,u); };
for(auto&& [u, value] : vertexlist(g, vvf)) {
  // ...  do something interesting
}
```

5.1 `struct vertex_info<VId, VV>`

`vertex_info` is used to return vertex information. It is used by `vertexlist(g)`, `vertices_breadth_first_search(g,u)`, `vertices_dfs(g,u)` and others. The `vertex` member is typically a vertex descriptor, but can also be a vertex id, and always exists.

```
template <class VorVId, class VV>
struct vertex_info {
  using vertex_type = VorVId; // e.g.  vertex_reference_t<G> or void
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  vertex_type vertex;
  value_type value;
};
```

Specializations are defined with `V=void` or `VV=void` to suppress the existance of their associated member variables, giving the following valid combinations in Table 3 . For instance, the second entry, `vertex_info<VId, void>` has one member `{vertex_type vertex;}` and `value_type` is `void` .

Template Arguments	Members	
`vertex_info<VorVId, VV>`	`vertex`	`value`
`vertex_info<VorVId, void>`	`vertex`	

Table 3: `vertex_info` Members

5.2 `struct edge_info<VId, Sourced, E, EV>`

`edge_info` is used to return edge information. It is used by `incidence(g,u)`, `edgelist(g)`, `edges_breadth_first_search(g,u)`, `edges_dfs(g,u)` and others. `source` and `target` are typically vertex descriptors, but can also be vertex ids. If no specific mention of vertex ids are used, assume they are vertex descriptors. In this section, `source` and `target` can be either vertex descriptors or vertex ids.

When `Sourced=true` , the `source` member is included with type `V` or `VId` . The `target` member always exists.

```
template <class VorVId, bool Sourced, class E, class EV>
struct edge_info {
  using source_type = VorVId; // e.g.  vertex_t<G> or vertex_id_t<G> when Sourced==true, or void
  using target_type = VorVId; // e.g.  vertex_t<G> or vertex_id_t<G>
  using edge_type = E; // e.g.  edge_reference_t<G> or void
  using value_type = EV; // e.g.  edge_value_t<G> or void

  source_type source;
  target_type target;
  edge_type edge;
  value_type value;
};
```

Specializations are defined with `Sourced=true|false` , `E=void` or `EV=void` to suppress the existance of the associated member variables, giving the following valid combinations in Table 4 . For instance, the second entry, `edge_info<VId,true,E>` has three members `{source_id_type source_id; target_id_type target_id; edge_type edge;}` and `value_type` is `void` .

5.3 `struct neighbor_info<VId, Sourced, V, VV>`

`neighbor_info` is used to return information for a neighbor vertex, through an edge. It is used by `neighbors(g,u)` . When `Sourced=true` , the `source` member is included with type `source_type` . The `target` member always exists.

Template Arguments	Members			
`edge_info<VorVId, true, E, EV>`	`source`	`target`	`edge`	`value`
`edge_info<VorVId, true, E, void>`	`source`	`target`	`edge`	
`edge_info<VorVId, true, void, EV>`	`source`	`target`		`value`
`edge_info<VorVId, true, void, void>`	`source`	`target`		
`edge_info<VorVId, false, E, EV>`		`target`	`edge`	`value`
`edge_info<VorVId, false, E, void>`		`target`	`edge`	
`edge_info<VorVId, false, void, EV>`		`target`		`value`
`edge_info<VorVId, false, void, void>`		`target`		

Table 4: `edge_info` Members

```
template <class VorVId, bool Sourced, class VV>
struct neighbor_info {
  using source_type = VorVId; // e.g.  vertex_t<G> or vertex_id_t<G> when Sourced==true, or void
  using target_type = VorVId; // e.g.  vertex_t<G> or vertex_id_t<G>
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  source_type source;
  target_type target;
  value_type value;
};
```

Specializations are defined with `Sourced=true|false` or `EV` =void to suppress the existance of the associated member variables, giving the following valid combinations in Table 5 . For instance, the second entry, `neighbor_info<V,true>` has two members `{source_type source; target_type target;}` and `value_type` is `void` .

Template Arguments	Members		
`neighbor_info<VorVId, true, EV>`	`source`	`target`	`value`
`neighbor_info<VorVId, true, void>`	`source`	`target`	
`neighbor_info<VorVId, false, EV>`		`target`	`value`
`neighbor_info<VorVId, false, void>`		`target`	

Table 5: `neighbor_info` Members

6 Graph Views

6.1 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_info` on each iteration. Table 6 shows the vertexlist functions overloads and their return values. `u` is a vertex descriptor. `first` and `last` are vertex iterators.

The `vertexlist` view without the value function is of limited value, since `vertices(g)` does the same thing, without using a structured binding. However, it is included for consistency with the overload that uses a value function.

6.2 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_info` on each iteration. Table 7 shows the `incidence` function overloads and their return values.

Since the source vertex `u` is available when calling an `incidence` function, there's no need to include sourced versions of the function to include the `source` vertex in the output.

Example	Return
`for(auto&& [u] : vertexlist(g))`	`vertex_info<void,V,void>`
`for(auto&& [u,val] : vertexlist(g,vvf))`	`vertex_info<void,V,VV>`
`for(auto&& [u] : vertexlist(g,first,last))`	`vertex_info<void,V,void>`
`for(auto&& [u,val] : vertexlist(g,first,last,vvf))`	`vertex_info<void,V,VV>`
`for(auto&& [u] : vertexlist(g,vr))`	`vertex_info<void,V,void>`
`for(auto&& [u,val] : vertexlist(g,vr,vvf))`	`vertex_info<void,V,VV>`

Table 6: `vertexlist` View Functions

The `incidence` view without the value function is of limited value, since `edges(g,u)` does the same thing, without using a structured binding. However, it is included for consistency with the overload that uses a value function.

Example	Return
`for(auto&& [uv] : incidence(g,u))`	`edge_info<void,false,E,void>`
`for(auto&& [uv,val] : incidence(g,u,evf))`	`edge_info<void,false,E,EV>`

Table 7: `incidence` View Functions

6.3 neighbors Views

`neighbors` views iterate over a range of edges for a vertex, returning a `vertex_info` of each neighboring target vertex on each iteration. Table 8 shows the `neighbors` function overloads and their return values.

Since the source vertex `u` is available when calling a `neighbors` function, there's no need to include sourced versions of the function to include `source` vertex in the output.

Example	Return
`for(auto&& [v] : neighbors(g,uid))`	`neighbor_info<void,false,V,void>`
`for(auto&& [v,val] : neighbors(g,uid,vvf))`	`neighbor_info<void,false,V,VV>`

Table 8: `neighbors` View Functions

6.4 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_info` on each iteration. Table 9 shows the `edgelist` function overloads and their return values.

The range returned by `edgelist` adheres to the `basic_sourced_index_edgelist` concept (future proposals may only adhere to `basic_sourced_edgelist`). If a `evf(uv)` function is passed, it adheres to the `has_edge_value` concept.

Example	Return
`for(auto&& [u,v,uv] : edgelist(g))`	`edge_info<V,true,E,void>`
`for(auto&& [u,v,uv,val] : edgelist(g,evf))`	`edge_info<V,true,E,EV>`

Table 9: `edgelist` View Functions

7 "Search" Views

7.1 Common Types and Functions for "Search"

The Depth First, Breadth First, and Topological Sort searches share a number of common types and functions.

Here are the types and functions for cancelling a search, getting the current depth of the search, and active elements in the search (e.g. number of vertices in a stack or queue).

```
// enum used to define how to cancel a search
enum struct cancel_search : int8_t {
  continue_search, // no change (ignored)
  cancel_branch, // stops searching from current vertex
  cancel_all // stops searching and dfs will be at end()
};

// stop searching from current vertex
template<class S)
void cancel(S search, cancel_search);

// Returns distance from the seed vertex to the current vertex,
// or to the target vertex for edge views
template<class S>
auto depth(S search) -> integral;

// Returns number of pending vertices to process
template<class S>
auto size(S search) -> integral;
```

Of particular note, `size(dfs)` is typically the same as `depth(dfs)` and is simple to calculate. breadth_first_search requires extra bookkeeping to evaluate `depth(bfs)` and returns a different value than `size(bfs)` .

The following example shows how the functions could be used, using `dfs` for one of the depth_first_search views. The same functions can be used for all all search views.

```
auto&& g = ...; // graph
auto&& dfs = vertices_dfs(g,0); // start with vertex_id=0
for(auto&& [vid,v] : dfs) {
  // No need to search deeper?
  if(depth(dfs) > 3) {
    cancel(dfs,cancel_search::cancel_branch);
    continue;
  }

  if(size(dfs) > 1000) {
    std::cout << "Big depth of " << size(dfs) << '\n';
  }

  // do useful things
}
```

The range returned by *sourced* views (includes source_id) adheres to the `basic_sourced_index_edgelist` concept. If a `evf(uv)` function is passed, it also adheres to the `has_edge_value` concept.

7.2 Depth First Search Views

Depth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_info` or `edge_info` on each iteration when it is first encountered, depending on the function used. Table 10 shows the functions and their return values.

7.3 Breadth First Search Views

Breadth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_info` or `edge_info` on each iteration when it is first encountered, depending on the function used. Table 11 shows the functions and their return values.

Example	Return
`for(auto&& [v] : vertices_dfs(g,seed))`	`vertex_info<void,V,void>`
`for(auto&& [v,val] : vertices_dfs(g,seed,vvf))`	`vertex_info<void,V,VV>`
`for(auto&& [v,uv] : edges_dfs(g,seed))`	`edge_info<V,false,E,void>`
`for(auto&& [v,uv,val] : edges_dfs(g,seed,evf))`	`edge_info<V,false,E,EV>`
`for(auto&& [u,v,uv] : sourced_edges_dfs(g,seed))`	`edge_info<V,true,E,void>`
`for(auto&& [u,v,uv,val] : sourced_edges_dfs(g,seed,evf))`	`edge_info<V,true,E,EV>`

Table 10: depth_first_search View Functions

Example	Return
`for(auto&& [v] : vertices_bfs(g,seed))`	`vertex_info<void,V,void>`
`for(auto&& [v,val] : vertices_bfs(g,seed,vvf))`	`vertex_info<void,V,VV>`
`for(auto&& [v,uv] : edges_bfs(g,seed))`	`edge_info<V,false,E,void>`
`for(auto&& [v,uv,val] : edges_bfs(g,seed,evf))`	`edge_info<V,false,E,EV>`
`for(auto&& [u,v,uv] : sourced_edges_bfs(g,seed))`	`edge_info<V,true,E,void>`
`for(auto&& [u,v,uv,val] : sourced_edges_bfs(g,seed,evf))`	`edge_info<V,true,E,EV>`

Table 11: breadth_first_search View Functions

7.4 Topological Sort Views

Topological Sort views iterate over the vertices and edges from a given seed vertex, returning a `vertex_info` or `edge_info` on each iteration when it is first encountered, depending on the function used. Table 12 shows the functions and their return values.

Example	Return
`for(auto&& [v] : vertices_topological_sort(g,seed))`	`vertex_info<void,V,void>`
`for(auto&& [v,val] : vertices_topological_sort(g,seed,vvf))`	`vertex_info<void,V,VV>`
`for(auto&& [v,uv] : edges_topological_sort(g,seed))`	`edge_info<V,false,E,void>`
`for(auto&& [v,uv,val] : edges_topological_sort(g,seed,evf))`	`edge_info<V,false,E,EV>`
`for(auto&& [u,v,uv] : sourced_edges_topological_sort(g,seed))`	`edge_info<V,true,E,void>`
`for(auto&& [u,v,uv,val] : sourced_edges_topological_sort(g,seed,evf))`	`edge_info<V,true,E,EV>`

Table 12: topological_sort View Functions

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.