Graph Library: Views

Document #:	**P3129r0**
Date:	2024-02-05
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency SG6 Numerics
Revises:	P1709r5

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 (Codeplay)
Jens Maurer
Richard Dosselmann (University of Regina)
Matthew Galati (Amazon)

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**, describing the big picture of what we are proposing.
P3127	Active	**Background and Terminology** providing the motivation, theoretical background and terminology used across the other documents.
P3128	Active	**Algorithms** covering 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** describing 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.

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* paper (P3126) to understand focus and scope of our proposals.

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

— If you want to **use the algorithms**, you should read the *Algorithms* paper (P3128) and *Graph Containers* paper (P3131).

— If you want to **write new algorithms**, you should read the *Views* paper (P3129), *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131). 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 container**, you should read the *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131).

2 Revision History

P3129r0

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

— Removed allocator parameters on views, for consistency with existing views.

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
V	vertex_t<G>		Vertex
	vertex_reference_t<G>	u,v,x,y	Vertex 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.
		first,last	vi is the target vertex.
VVF		vvf	Vertex Value Function: vvf(u) → vertex value, or vvf(uid) → vertex value, depending on requirements of the consume algorithm or view.
VProj		vproj	Vertex descriptor projection function: vproj(x) → vertex_descriptor<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
	edge_reference_t<G>	uv,vw	Edge reference. uv is an edge from vertices u to v . vw is an edge from vertices v to w .
EId	edge_id_t<G>	eid,uvid	Edge id, a pair of vertex_ids.
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, or evf(eid) → edge value, depending on the requirements of the consuming algorithm or view.
EProj		eproj	Edge descriptor projection function: eproj(x) → edge_descriptor<VId,Sourced,EV> .
PER	partition_edge_range_t<G>		Partition Edge Range for edges of a partition vertex.

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.

5 Descriptors (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 actual values returned to be appropriate for its use. The three types, `vertex_descriptor`, `edge_descriptor` and `neighbor_descriptor`, define the common data model used by algorithms.

The following examples show the general design and how it's used. While it focuses on vertexlist to iterate over all vertices, it applies to all descriptors and view functions.

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

Structured bindings make it simpler.

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

A function object can also be passed to return a value from the vertex. In this case, `vertexlist(g)` returns `vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, decltype(vvf(u))>`.

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

A simpler version also exists if all you need is a vertex id. The vertex value function takes a vertex id instead of a vertex reference.

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

auto vvf = [&g](vertex_id_t<G> uid) { return vertex_value(g,uid); };
for(auto&& [uid, value] : basic_vertexlist(g,vvf)) {
  // ...  do something interesting
}
```

5.1 `struct vertex_descriptor<VId, V, VV>`

`vertex_descriptor` 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 `id` member always exists.

```
template <class VId, class V, class VV>
struct vertex_descriptor {
  using id_type = VId; // e.g.  vertex_id_t<G>
  using vertex_type = V; // e.g.  vertex_reference_t<G> or void
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  id_type id;
  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_descriptor<VId, V>` has two members `{id_type id; vertex_type vertex;}` and `value_type` is `void` .

Template Arguments	Members		
`vertex_descriptor<VId, V, VV>`	`id`	`vertex`	`value`
`vertex_descriptor<VId, V, void>`	`id`	`vertex`	
`vertex_descriptor<VId, void, VV>`	`id`		`value`
`vertex_descriptor<VId, void, void>`	`id`		

Table 3: `vertex_descriptor` Members

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

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

```
template <class VId, bool Sourced, class E, class EV>
struct edge_descriptor {
  using source_id_type = VId; // e.g.  vertex_id_t<G> when SourceId==true, or void
  using target_id_type = VId; // e.g.  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_id_type source_id;
  target_id_type target_id;
  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_descriptor<VId,true,E>` has three members `{source_id_type source_id; target_id_type target_id; edge_type edge;}` and `value_type` is `void` .

Template Arguments	Members			
`edge_descriptor<VId, true, E, EV>`	`source_id`	`target_id`	`edge`	`value`
`edge_descriptor<VId, true, E, void>`	`source_id`	`target_id`	`edge`	
`edge_descriptor<VId, true, void, EV>`	`source_id`	`target_id`		`value`
`edge_descriptor<VId, true, void, void>`	`source_id`	`target_id`		
`edge_descriptor<VId, false, E, EV>`		`target_id`	`edge`	`value`
`edge_descriptor<VId, false, E, void>`		`target_id`	`edge`	
`edge_descriptor<VId, false, void, EV>`		`target_id`		`value`
`edge_descriptor<VId, false, void, void>`		`target_id`		

Table 4: `edge_descriptor` Members

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

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

```
template <class VId, bool Sourced, class V, class VV>
struct neighbor_descriptor {
  using source_id_type = VId; // e.g.  vertex_id_t<G> when Sourced==true, or void
  using target_id_type = VId; // e.g.  vertex_id_t<G>
  using vertex_type = V; // e.g.  vertex_reference_t<G> or void
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  source_id_type source_id;
  target_id_type target_id;
  vertex_type target;
  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 5. For instance, the second entry, `neighbor_descriptor<VId,true,E>` has three members {`source_id_type source_id`; `target_id_type target_id`; `vertex_type target`;} and `value_type` is `void`.

Template Arguments	Members			
`neighbor_descriptor<VId, true, E, EV>`	`source_id`	`target_id`	`target`	`value`
`neighbor_descriptor<VId, true, E, void>`	`source_id`	`target_id`	`target`	
`neighbor_descriptor<VId, true, void, EV>`	`source_id`	`target_id`		`value`
`neighbor_descriptor<VId, true, void, void>`	`source_id`	`target_id`		
`neighbor_descriptor<VId, false, E, EV>`		`target_id`	`target`	`value`
`neighbor_descriptor<VId, false, E, void>`		`target_id`	`target`	
`neighbor_descriptor<VId, false, void, EV>`		`target_id`		`value`
`neighbor_descriptor<VId, false, void, void>`		`target_id`		

Table 5: `neighbor_descriptor` Members

5.4 Copyable Descriptors

5.4.1 Copyable Descriptor Types

Copyable descriptors are specializations of the descriptors that can be copied. More specifically, they don't include a vertex or edge reference. `copyable_vertex_t<G>` shows the simple definition.

```
template <class VId, class VV>
using copyable_vertex_t = vertex_descriptor<VId, void, VV>; // id, value
```

Type	Definition
`copyable_vertex_t<T,VId,VV>`	`vertex_descriptor<VId, void, VV>`
`copyable_edge_t<T,Vid,EV>`	`edge_descriptor<VId, true, void, EV>>`
`copyable_neighbor_t<Vid,VV>`	`neighbor_descriptor<VId, true, void, VV>`

Table 6: Descriptor Concepts

5.4.2 Copyable Descriptor Concepts

Given the copyable types, it's useful to have concepts to determine if a type is a desired copyable type.

Concept	Definition
`copyable_vertex<T,VId,VV>`	`convertible_to<T, copyable_vertex_t<VId, VV>>`
`copyable_edge<T,Vid,EV>`	`convertible_to<T, copyable_edge_t<VId, EV>>`
`copyable_neighbor<T,Vid,VV>`	`convertible_to<T, copyable_neighbor_t<VId, VV>>`

Table 7: Descriptor Concepts

6 Graph Views

6.1 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_descriptor` on each iteration. Table 8 shows the vertexlist functions overloads and their return values. `first` and `last` are vertex iterators.

`vertexlist` views require a `vvf(u)` function, and the `basic_vertexlist` views require a `vvf(uid)` function.

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

Table 8: `vertexlist` View Functions

6.2 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_descriptor` on each iteration. Table 9 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 `source_id` in the output.

incidence views require a evf(uv) function, and basic_incidence views require a evf(eid) function.

Example	Return
for(auto&& [vid,uv] : incidence(g,uid))	edge_descriptor<VId,false,E,void>
for(auto&& [vid,uv,val] : incidence(g,uid,evf))	edge_descriptor<VId,false,E,EV>
for(auto&& [vid] : basic_incidence(g,uid))	edge_descriptor<VId,false,void,void>
for(auto&& [vid,val] : basic_incidence(g,uid,evf))	edge_descriptor<VId,false,void,EV>

Table 9: incidence View Functions

6.3 neighbors Views

neighbors views iterate over a range of edges for a vertex, returning a vertex_descriptor of each neighboring target vertex on each iteration. Table 10 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_id in the output.

neighbors views require a vvf(u) function, and the basic_neighbors views require a vvf(uid) function.

Example	Return
for(auto&& [vid,v] : neighbors(g,uid))	neighbor_descriptor<VId,false,V,void>
for(auto&& [vid,v,val] : neighbors(g,uid,vvf))	neighbor_descriptor<VId,false,V,VV>
for(auto&& [vid] : basic_neighbors(g,uid))	neighbor_descriptor<VId,false,void,void>
for(auto&& [vid,val] : basic_neighbors(g,uid,vvf))	neighbor_descriptor<VId,false,void,VV>

Table 10: neighbors View Functions

6.4 edgelist Views

edgelist views iterate over all edges for all vertices, returning a edge_descriptor on each iteration. Table 11 shows the edgelist function overloads and their return values.

edgelist views require a evf(uv) function, and basic_edgelist views require a evf(eid) function.

Example	Return
for(auto&& [uid,vid,uv] : edgelist(g))	edge_descriptor<VId,true,E,void>
for(auto&& [uid,vid,uv,val] : edgelist(g,evf))	edge_descriptor<VId,true,E,EV>
for(auto&& [uid,uv] : basic_edgelist(g))	edge_descriptor<VId,true,void,void>
for(auto&& [uid,uv,val] : basic_edgelist(g,evf))	edge_descriptor<VId,true,void,EV>

Table 11: 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
}
```

7.2 Depth First Search Views

Depth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 12 shows the functions and their return values.

`vertices_dfs` views require a `vvf(u)` function, and the `basic_vertices_dfs` views require a `vvf(uid)` function. `edges_dfs` views require a `evf(uv)` function. `basic_sourced_edges_dfs` views require a `evf(eid)` function. A `basic_edges_dfs` view with a evf is not available because `evf(eid)` requires that the `source_id` is available.

7.3 Breadth First Search Views

Breadth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 13 shows the functions and their return values.

`vertices_bfs` views require a `vvf(u)` function, and the `basic_vertices_bfs` views require a `vvf(uid)` function. `edges_bfs` views require a `evf(uv)` function.

`basic_sourced_edges_bfs` views require a `evf(eid)` function. A `basic_edges_bfs` view with a evf is not available because `evf(eid)` requires that the `source_id` is available.

Example	Return
`for(auto&& [vid] : basic_vertices_dfs(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_dfs(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_dfs(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_dfs(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_dfs(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_dfs(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_dfs(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_dfs(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_dfs(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 12: depth_first_search View Functions

Example	Return
`for(auto&& [vid] : basic_vertices_bfs(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_bfs(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_bfs(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_bfs(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_bfs(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_bfs(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_bfs(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_bfs(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_bfs(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 13: 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_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 14 shows the functions and their return values.

`vertices_topological_sort` views require a `vvf(u)` function, and the `basic_vertices_topological_sort` views require a `vvf(uid)` function. `edges_topological_sort` views require a `evf(uv)` function.

Example	Return
`for(auto&& [vid] : basic_vertices_topological_sort(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_topological_sort(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_topological_sort(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_topological_sort(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_topological_sort(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_topological_sort(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_topological_sort(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_topological_sort(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 14: 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.