Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 D9902r0 — Graph Library Algorithms
Date: 2024-02-11
Reply to: Phil Ratzloff (phil.ratzloff@sas.com),
Andrew Lumsdaine (lumsdaine@gmail.com)

Contributors: Richard Dosselmann (University of Regina)
Michael Wong (Codeplay)
Matthew Galati (Amazon)
Jens Maurer
Jesun Firoz
Kevin Deweese
Muhammad Osama (AMD, Inc)

Audience: SG19, SG14, SG6, LEWG, LWG
Source: github.com/stdgraph/graph-v2
Prev. Version: www.wg21.link/P1709r5

Contents

Contents 1

1 Getting Started 2
1.1 Revision History . . . 2

2 Views 3
2.1 Descriptors (Return Types) . . . 3
2.2 Copyable Descriptors . . . 5
2.3 Common Types and Functions for “Search” . . . 6
2.4 vertexlist Views . . . 7
2.5 incidence Views . . . 7
2.6 neighbors Views . . . 7
2.7 edgelist Views . . . 7
2.8 Depth First Search Views . . . 8
2.9 Breadth First Search Views . . . 8
2.10 Topological Sort Views . . . 9

Acknowledgements 10

Chapter1 Getting Started

This paper is one of several interrelated proposals related to a Graph Library proposal that have been broken out for easier consumption. The following table describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now broken into the following papers.
P9901	Active	**Graph Library Overview and Introduction**, describing the big picture of what we are proposing, and theortical background of graphs in general.
P9902	Active	**Graph Library Algorithms**, covering the initial algorithms as well as the ones we'd like to see in the future.
P9903	Active	**Graph Library Operators** includes useful utility functions when working with graphs.
P9904	Active	**Graph Library Views** including helpful views for traversing a graph.
P9905	Active	**Graph Library Container Inferface** is the core interface used for accessing the underlying graph data structure.
P9906	Active	**Graph Library Container** describing the high-performance `compressed_graph` container, based on a Compressed Sparse Row sparse matrix layout.
P9907	Active	**Graph Library Adaptors** containing useful utilities to convert graphs to different forms.

Table 1.1 — Graph Library Papers

1.1 Revision History

D9902r0

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

Chapter2 Views

The views in this section 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.

2.1 Descriptors (Return Types)

Views return one of the types in this section, providing a consistent set of values. 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 data model used by the 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
}
```

2.1.0.1 `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 2.1 . 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 2.1 — `vertex_descriptor` Members

2.1.0.2 `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_searc(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 2.2 . 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` .

2.1.0.3 `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>
```

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 2.2 — edge_descriptor Members

```
  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 2.3 . 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 2.3 — neighbor_descriptor Members

2.2 Copyable Descriptors

2.2.1 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 2.4 — Descriptor Concepts

2.2.2 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 2.5 — Descriptor Concepts

2.3 Common Types and Functions for "Search"

[PHIL: Do these apply to all "search" functions?]

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
}
```

2.4 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_descriptor` on each iteration. Table 2.6 shows the vertexlist functions overloads and their return values. `first` and `last` are vertex iterators.

[PHIL: Change naming to `vertexlist` and `extended_vertexlist` instead of `basic_vertexlist` and `vertexlist` ?]

`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 2.6 — `vertexlist` View Functions

2.5 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_descriptor` on each iteration. Table 2.7 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 `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 2.7 — `incidence` View Functions

2.6 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 2.8 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_id` in the output.

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

2.7 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_descriptor` on each iteration. Table 2.9 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&& [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 2.8 — `neighbors` View Functions

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 2.9 — `edgelist` View Functions

2.8 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 2.10 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`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.

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 2.10 — depth_first_search View Functions

2.9 Breadth First Search Views

[PHIL: NetworkX provides an optional depth_limit parameter for bfs. Add?]

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 2.11 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`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_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 2.11 — breadth_first_search View Functions

2.10 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 2.12 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`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 2.12 — topological_sort View Functions

[PHIL: Is Topological Sort a view, an algorithm or both?]

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.

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