Graph Library: Views

Document #:	**P3129r2**
Date:	2026-07-06
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) Scott McMillan (Carnegie Mellon University) 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	Active	**Comparison to other graph libraries** on performance and usage syntax.

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 Topological 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.

P3129r1b

— Replace the use of *id* and *reference* in view functions with the `vertex_t<G>` and `edge_t<G>` descriptors, leading to a simpler interace. 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 `vertex_t<G>` and `edge_t<G>` descriptors are needed.

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

P3129r2

— Renamed `vertex_info`, `edge_info`, and `neighbor_info` to `vertex_data`, `edge_data`, and `neighbor_data` respectively, to align their names to represent the core data models used.

— Updated all value functions from 1-arg `vvf(u)`/`evf(uv)` form to 2-arg `vvf(g,u)`/`evf(g,uv)` form, taking the graph as the first parameter.

— Added incoming-edge view variants (`in_incidence`, `in_neighbors`) for bidirectional graphs.

— Added `transpose` view for direction-swapping graph adaption.

— Added pipe syntax support via range adaptors for all view functions.

— Changed topological sort from seeded (single source) to all-vertex traversal.

— Added `_safe` topological sort factories with cycle detection via `std::expected`.

— Removed `sourced_edges_dfs`, `sourced_edges_bfs`, and `sourced_edges_topological_sort` — sourced search variants are no longer in the implementation.

— Changed search view control from free functions (`cancel()`, `depth()`, `size()`) to member functions (`.cancel()`, `.depth()`, `.num_visited()`).

— Added `search_view` concept.

— Replaced all uses of `vertex_reference_t<G>` with `vertex_t<G>` and `edge_reference_t<G>` with `edge_t<G>`, consistent with the descriptor-based architecture defined in P3130 Graph Container Interface.

— Moved `vertex_data`, `edge_data`, and `neighbor_data` to P3130 Graph Container Interface.

— Replaced all uses of `vertex_reference_t<G>` with `vertex_t<G>` and `edge_reference_t<G>` with `edge_t<G>`, consistent with the descriptor-based architecture defined in P3130 Graph Container Interface.

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>`	`u`,`v`	Vertex descriptor. `u` is the source (or only) vertex. `v` is the target vertex.
`VId`	`vertex_id_t<G>`	`uid`,`vid`,`source`	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(g, u)` → vertex value, or `vvf(g, uid)` → vertex value, depending on requirements of the consuming algorithm or view.
`VProj`		`vproj`	Vertex data projection function: `vproj(u)` → `vertex_data<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>`	`uv`,`vw`	Edge descriptor. `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(g, uv)` → edge value.
`EProj`		`eproj`	Edge data projection function: `eproj(uv)` → `edge_data<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 as simple as iterating through the set of vertices, or more complex such as iterating through the vertices in a depth-first and breadth-first order. They also provide a consistent and reliable way to access related elements using the data structs (§5), 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.

4.0.0.1 Value Function Concepts. Many views accept a *vertex value function* (`vvf`) or *edge value function* (`evf`) that projects a user-defined value from each vertex or edge. The `vertex_value_function` and `edge_value_function` concepts (defined in P3130, Graph Container Interface) constrain these parameters and are used throughout this paper.

5 Data 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_data`, `edge_data` and `neighbor_data`, 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_data<vertex_id_t<G>, vertex_t<G>, void>
for(auto&& uu : vertexlist(g)) {
  vertex_id_t<G> uid = uu.id;
  vertex_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 three members, `id`, `vertex` and `value`.

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

Structured bindings make it simpler.

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

Finally, using structured binding with the vertex value function.

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

See the Utility Types and Functions in P3130 Graph Container Interface for more details about the definition and use of the vertex, edge and neighbor data types.

6 Graph Views

6.1 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_data` on each iteration. Table 3 shows the vertexlist functions overloads and their return values. `uid` is a vertex id and `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.

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

Table 3: `vertexlist` View Functions

6.2 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_data` on each iteration. Table 4 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_id` in the output.

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&& [vid, uv] : incidence(g,u))`	`edge_data<VId,false,E,void>`
`for(auto&& [vid, uv, val] : incidence(g,u,evf))`	`edge_data<VId,false,E,EV>`

Table 4: `incidence` View Functions

6.3 in_incidence Views

`in_incidence` views iterate over the incoming edges of a vertex, returning an `edge_data` on each iteration. They mirror the `incidence` views but traverse edges whose target is the given vertex rather than edges whose source is the given vertex. These views require the graph to model `bidirectional_adjacency_list`. Table 5 shows the `in_incidence` function overloads and their return values.

Example	Return
`for(auto&& [vid, uv] : in_incidence(g,u))`	`edge_data<VId,false,E,void>`
`for(auto&& [vid, uv, val] : in_incidence(g,u,evf))`	`edge_data<VId,false,E,EV>`

Table 5: `in_incidence` View Functions

For incoming edges the returned `id` member is the source vertex id of the incoming edge (i.e. the "other" end). The data structs are identical to those used by `incidence`; only the direction of traversal changes.

6.4 neighbors Views

`neighbors` views iterate over a range of edges for a vertex, returning a `neighbor_data` of each neighboring target vertex on each iteration. Table 6 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.

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

Table 6: `neighbors` View Functions

6.5 in_neighbors Views

`in_neighbors` views iterate over the incoming neighbors of a vertex, returning a `neighbor_data` for each source vertex of an incoming edge. They mirror the `neighbors` views but traverse incoming rather than outgoing edges. These views require the graph to model `bidirectional_adjacency_list`. Table 7 shows the `in_neighbors` function overloads and their return values.

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

Table 7: `in_neighbors` View Functions

As with `in_incidence`, the returned `id` member is the source vertex id of the incoming edge.

6.5.0.1 Edge Accessor Mechanism (Exposition Only). Internally, the outgoing and incoming view variants share the same view class templates. An *edge accessor* policy selects the traversal direction: `out_edge_accessor` (the default) uses `edges(g,u)` and `target_id(g,uv)`, while `in_edge_accessor` uses `in_edges(g,u)` and `source_id(g,uv)`. This mechanism is exposition-only and not part of the public interface.

6.6 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_data` on each iteration. Table 8 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(g,uv)` function is passed, it adheres to the `has_edge_value` concept.

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

Table 8: `edgelist` View Functions

6.7 transpose View

The `transpose` view provides a zero-cost wrapper around a bidirectional graph that swaps the direction of edge traversal. Outgoing edges become incoming and vice versa. It is not a range itself but a graph adaptor: the result models the same graph concepts as the underlying graph with directions reversed.

`transpose_view<G>` requires the graph to model `bidirectional_adjacency_list<G>`. When the underlying graph also models `index_bidirectional_adjacency_list`, the transposed view does too.

```
auto gt = transpose(g); // gt is a transpose_view<G>
for(auto&& [vid, uv] : incidence(gt, u)) {
  // iterates what were incoming edges of u in g
}
```

Because `transpose` is a graph wrapper rather than an edge-level view, it composes with all other views: `incidence(transpose(g), u)` traverses incoming edges, `vertexlist(transpose(g))` is unchanged, and algorithm templates work transparently on the transposed graph.

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 member functions.

Search views provide member functions for cancelling a search, querying the current depth, and the number of vertices or edges visited so far. The `cancel_search` enumeration controls search cancellation:

```
enum class cancel_search {
  continue_search, // continue normal traversal
  cancel_branch, // skip current subtree/branch, continue with siblings
  cancel_all // stop entire search immediately
};
```

Each search view exposes the following member functions:

```
cancel_search cancel() const noexcept; // get current cancel state
void cancel(cancel_search c) noexcept; // set cancel state
std::size_t depth() const noexcept; // depth of currently yielded element (seed is 0)
std::size_t num_visited() const noexcept; // elements yielded so far, including the current
    element
```

The `search_view` concept captures these requirements:

```
template <class V>
concept search_view = requires(V& v, const V& cv) {
  { v.cancel() } -> std::convertible_to<cancel_search>;
  { cv.depth() } -> std::convertible_to<std::size_t>;
  { cv.num_visited() } -> std::convertible_to<std::size_t>;
};
```

Topological sort views support `cancel()` and `num_visited()` but not `depth()` (flat ordering has no tree structure), so they do not satisfy `search_view`.

For views that satisfy `search_view` (DFS and BFS), `depth()` denotes the depth of the currently yielded element measured from the seed. The seed has depth 0, and each traversed edge increases depth by 1.

`num_visited()` returns the number of elements yielded so far, *including the element currently held by the loop variable*. On the first iteration it equals 1 (the seed for vertex views, the first tree edge for edge views). This makes threshold checks inside the loop body intuitive: when the loop body executes, `num_visited()` reflects the current position rather than the previous one.

The following example shows how the member functions could be used, using `dfs` for one of the depth-first search views. The same members are available on all search views that satisfy `search_view`.

```
auto&& g = ...; // graph
auto&& dfs = vertices_dfs(g,0); // start with seed vertex_id=0
// On first iteration:  dfs.depth()==0, dfs.num_visited()==1
for(auto&& [uid,u] : dfs) {
  // No need to search deeper?
  if(dfs.depth() > 3) {
    dfs.cancel(cancel_search::cancel_branch);
    continue;
  }

  if(dfs.num_visited() > 1000) {
    std::cout << "Many visited: " << dfs.num_visited() << '\n';
  }

  // do useful things
}
```

7.2 Depth First Search Views

Depth First Search views iterate over vertices and edges reachable from a seed vertex, yielding a `vertex_data` or `edge_data` for each element when it is first encountered. Table 9 shows the factory functions and their return types.

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

Table 9: depth_first_search View Functions

7.3 Breadth First Search Views

Breadth First Search views iterate over vertices and edges reachable from a seed vertex, yielding a `vertex_data` or `edge_data` for each element when it is first encountered. Table 10 shows the factory functions and their return types.

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

Table 10: breadth_first_search View Functions

7.3.0.1 Design Note. Search views yield vertex and edge *descriptors* rather than vertex IDs (`VId` is `void` in the returned data structs). A vertex ID can always be obtained from a descriptor via `vertex_id(g, u)`. Descriptors provide richer access to vertex and edge properties through CPOs such as `vertex_value(g, u)` and `edge_value(g, uv)`.

7.3.0.2 Allocator Support. All DFS and BFS factory functions accept an optional trailing allocator parameter for their internal data structures. When a value function is also provided, the allocator is disambiguated via a `requires(!vertex_value_function<Alloc,...>)` constraint. For example:

```
vertices_dfs(g, seed, alloc) // allocator, no value function
vertices_dfs(g, seed, vvf, alloc) // value function + allocator
```

7.4 Topological Sort Views

Topological Sort views iterate over all vertices and edges in the graph in topological order, yielding a `vertex_data` or `edge_data` for each element when it is first encountered. Table 11 shows the factory functions and their return types.

Topological sort views support `cancel()` and `num_visited()` but not `depth()`, because the flat topological ordering has no tree structure. As a consequence, they do not satisfy the `search_view` concept. Note that `cancel_branch` behaves the same as `cancel_all` for topological sort, since there is no branch to skip in a flat ordering.

Example	Return
`for(auto&& [u] : vertices_topological_sort(g))`	`vertex_data<void,V,void>`
`for(auto&& [u,val] : vertices_topological_sort(g,vvf))`	`vertex_data<void,V,VV>`
`for(auto&& [uv] : edges_topological_sort(g))`	`edge_data<void,false,E,void>`
`for(auto&& [uv,val] : edges_topological_sort(g,evf))`	`edge_data<void,false,E,EV>`

Table 11: topological_sort View Functions

7.4.0.1 Cycle-detecting factories. The `_safe` factory variants detect cycles during construction and return a `std::expected`. On success the expected value is the topological sort view; on failure the unexpected value is the `vertex_t<G>` where the cycle was detected. Table 12 shows these factories.

Factory	Return
`vertices_topological_sort_safe(g)`	`std::expected<`*view*`, vertex_t<G>>`
`vertices_topological_sort_safe(g,vvf)`	`std::expected<`*view*`, vertex_t<G>>`
`edges_topological_sort_safe(g)`	`std::expected<`*view*`, vertex_t<G>>`
`edges_topological_sort_safe(g,evf)`	`std::expected<`*view*`, vertex_t<G>>`

Table 12: topological_sort Safe Factory Functions

The paper specifies `std::expected` (C++23). The reference implementation provides backward compatibility to C++20 via an external expected library (e.g., `tl::expected`), switching to `std::expected` when C++23 or later is available.

7.4.0.2 Allocator Support. Topological sort factory functions also accept an optional trailing allocator parameter, following the same pattern as DFS and BFS.

8 Range Adaptors (Pipe Syntax)

All range-returning, non-`_safe` view factories listed in Tables 13 and 14 are also available as range adaptor closure objects, enabling pipe syntax. The adaptor objects live in `namespace graph::views` and forward to the corresponding factory function. The `_safe` topological-sort factories return `std::expected` and are not adaptor closures. `transpose` is a graph adaptor (not a range adaptor) and remains documented separately.

Pipe Expression	Equivalent Factory Call
`g | vertexlist()`	`vertexlist(g)`
`g | vertexlist(vvf)`	`vertexlist(g,vvf)`
`g | incidence(uid)`	`incidence(g,uid)`
`g | incidence(uid, evf)`	`incidence(g,uid,evf)`
`g | in_incidence(uid)`	`in_incidence(g,uid)`
`g | in_incidence(uid, evf)`	`in_incidence(g,uid,evf)`
`g | neighbors(uid)`	`neighbors(g,uid)`
`g | neighbors(uid, vvf)`	`neighbors(g,uid,vvf)`
`g | in_neighbors(uid)`	`in_neighbors(g,uid)`
`g | in_neighbors(uid, vvf)`	`in_neighbors(g,uid,vvf)`
`g | edgelist()`	`edgelist(g)`
`g | edgelist(evf)`	`edgelist(g,evf)`

Table 13: Graph View Range Adaptors

Pipe Expression	Equivalent Factory Call
`g | vertices_dfs(seed)`	`vertices_dfs(g,seed)`
`g | vertices_dfs(seed, vvf)`	`vertices_dfs(g,seed,vvf)`
`g | edges_dfs(seed)`	`edges_dfs(g,seed)`
`g | edges_dfs(seed, evf)`	`edges_dfs(g,seed,evf)`
`g | vertices_bfs(seed)`	`vertices_bfs(g,seed)`
`g | vertices_bfs(seed, vvf)`	`vertices_bfs(g,seed,vvf)`
`g | edges_bfs(seed)`	`edges_bfs(g,seed)`
`g | edges_bfs(seed, evf)`	`edges_bfs(g,seed,evf)`
`g | vertices_topological_sort()`	`vertices_topological_sort(g)`
`g | vertices_topological_sort(vvf)`	`vertices_topological_sort(g,vvf)`
`g | edges_topological_sort()`	`edges_topological_sort(g)`
`g | edges_topological_sort(evf)`	`edges_topological_sort(g,evf)`

Table 14: Search View Range Adaptors

The `in_incidence` and `in_neighbors` require the graph to model `index_bidirectional_adjacency_list` when used as pipe adaptors (vertex-id overloads).

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.