D1709r3 Graph Library

Phillip Ratzloff (SAS Institute)
Andrew Lumsdaine (TileDB/University of Washington)

Document Number:	D1709r3
Date:	TBD, 2022 (mailing)
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21, LEWG
Source:	Github
Issue Tracking:	Github
Contributors:	Richard Dosselmann (University of Regina)
	Michael Wong (Codeplay)
	Matthew Galati (Amazon)
	Jens Maurer
	Domagoj Saric
	Jesun Firoz
	Kevin Deweese
Emails:	`Phil.Ratzloff@sas.com`
	`Andrew.Lumsdaine@tiledb.com`
	`dosselmr@cs.uregina.ca`
	`michael@codeplay.com`
	`magalati@amazon.com`
Reply to:	**`Phil.Ratzloff@sas.com`**

Contents

Revision History

P1709R3

This was a major redesign that incorporated all the experience and input from the past 3 years.

- Reduce the scope to focus on an edge list and adjacency graph with outgoing edges only, and remove mutable interface functions.
- Replace directed and undirected concepts with overridable types of unordered_edge for a graph implementation type.
- Simplify the Graph Container types and functions. In particular, const and non-const variations were consolidated to a single definition to handle both cases when appropriate.
- All Graph Container Interface functions are customization points.
- Introduce Views, inspired by NWGraph design, resulting in simpler and cleaner interfaces to traverse a graph, and simplifying the container interface design.
- Revisit the algorithms to be considered. transitive_closure has been dropped. The final list hasn't been finalized yet.
- Replace the two container implementations with csr_graph.

P1709R2

Define the **uniform API** for undirected and directed algorithms (an extended API also exists for directed graphs). Added **concepts** for undirected, directed and bidirected graphs. Refined **DFS** and **BFS** range definitions from prototype experience. Refined **shortest paths** and **transitive closure** algorithms from input and prototype experience.

P1709R1

Rewrite with a focus on a **purely functional design**, emphasizing the algorithms and graph API. Also added **concepts** and **ranges** into the design. Addressed concerns from Cologne review to change to functional design.

P1709R0

Focus on **object-oriented API** for data structures and example code for a few algorithms.

1 Introduction

This document proposes the addition of **graph algorithms**, **graph views**, **graph container interface** and a **graph container implementation** to the C++ library to support **machine learning** (ML), as well as other applications. ML is a large and growing field, both in the **research community** and **industry**, that has received a great deal of attention in recent years. This paper presents an **interface** of the proposed algorithms, views, graph functions and containers.

1.1 Motivation

Graphs, used in ML and other **scientific** domains, as well as **industrial** and **general** programming, do **not** presently exist in the C++ standard. In ML, a graph forms the underlying structure of an **artificial neural network** (ANN). In a **game**, a graph can be used to represent the **map** of a game world. In **business** environments, graphs arise as **entity relationship diagrams** (ERD) or **data flow diagrams** (DFD). In the realm of **social media**, a graph represents a **social network**.

1.2 Impact on the Standard

This proposal is a pure **library** extension.

1.3 Interaction wtih Other Papers

There is no interaction with other proposals to the standard.

1.4 Goals and Priorities

- Follow the separation of algorithms, ranges, views and containers established by the standard library.
- All free functions should be customization point objects, unless there's a good reason not to. Reasonable default implementations should be provided whenever possible.
- Graph algorithms have the following characteristics
  - Support syntax that is simple, expressive and easy to understand. This should not compromise the ability to write high-performance algorithms.
  - Vertices are required to be in random access containers with an integral vertex_id in this proposal.
- Graph views provide common traversals of a graph's vertices and edges that is more concise and consistant than using the graph container interface directly. They include simple traversals like vertexlist (all vertices in the graph) and incidence edges (edges on a vertex), as well as more complex traversals like depth-first and breath-first searches.
- The Graph Container Interface provides a consistent interface that can be used by algorithms and views. It has the following characteristics:
  - The interface models an adjacency graph container, which is an outer range of vertices with an inner range of outgoing (a.k.a. incidence) edges on each vertex.
  - Definition of concepts, types, type traits, type aliases, and functions used by algorithms and views.
    - Type traits will be defined that can be overridden for each graph container to give additional hints that can be used by algorithms to refine their behavior, such as adjacency_matrix and unordered_edge.
  - Support of optional user-defined value types on an edge, vertex and/or the graph itself.
  - Allow for useful extensions of the graph data model in future proposals or in external graph implementations.
- Provide an initial suite of useful functionality that includes algorithms, views, container interface, and at least one container implementation.

1.5 What this proposal is not

This paper limits itself to adjacency graphs only, including an outer range of vertices with an inner range of outgoing edges on each vertex. It also includes an edgelist of all edges in the graph, either as a edgelist view or a simple range with a source_id and target_id. It does not include incoming edges on a vertex, though that could be a future extension.

Bipartite graphs are being investigated. A general design has been established and it needs to be implemented to validate that it will work and see what areas of the design are impacted.

Hypergraphs are not supported.

Parallel versions of the algorithms are not included for several reasons. The executors proposal in P2300r5 [**?**] is expected to introduce new and better ways to do parallel algorithms beyond that used in the parallel STL algorithms and we would like to wait for finalization of that proposal before committing to parallel implementations. Secondly, many graph algorithms don't benefit from parallel implementations so there is less need to offer an implementation. Lastly, it will help limit the size of this proposal which is already looking to be large without it. It is expected that future proposals will be submitted for parallel graph algorithms.

1.6 Prior Art

boost::graph
Other?

1.6.1 Inspiration

NWGraph

1.7 Alternatives

1.8 Future Directions

2 Design - Introduction

Table **??** shows the naming conventions used throughout this document.

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` `first,last`	Vertex Iterator. `ui` is the source (or only) vertex. `vi` is the target vertex.
`VVF`		`vvf`	Vertex Value Function: vvf(u) $\rightarrow$ value
`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`.
`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) $\rightarrow$ value

Table 1: Naming Conventions for Types and Variables

2.1 Graph Data Models

2.1.1 Adjacency Graph

A *graph* [**?**] $G = (V, E)$ is a set of *vertices* [**?**] V, **points** in a space, and *edges* [**?**] E, **links** between these vertices. Edges may or may not be **oriented**, that is, *directed* [**?**] or *undirected* [**?**], respectively. Moreover, edges may be *weighted* [**?**], that is, assigned a value. Both **static** and **dynamic** implementations of a graph exist, specifically a (static) **matrix**, each having the typical advantages and disadvantages associated with static and dynamic data structures.

2.1.2 Edge List

2.2 Examples

2.2.1 Example: User

2.2.2 Example: Graph Author

3 Design - User Side

3.1 Algorithms

[PHIL: Algorithms marked [TBD] are provisional and may be moved to a separate proposal to keep the size of this proposal manageable]

3.1.1 Dijkstra's Shortest Paths

Dijkstra's algorithm [**?**] ...

3.1.2 Bellman-Ford Shortest Paths

The Bellman-Ford algorithm [**?**] ...

3.1.3 Connected Components

Connected components [**?**] ...

3.1.4 Strongly Connected Components

Strongly connected components [**?**] ...

3.1.5 Biconnected Components

Biconnected components [**?**] ...

3.1.6 Articulation Points

Articulation points [**?**] ...

3.1.7 Minimum Spanning Tree

Minimum Spanning Tree [**?**] ...

3.1.8 [TBD] Page Rank

3.1.9 [TBD] Betweenness Centrality

3.1.10 [TBD] Triangle Count

3.1.11 [TBD] Subgraph Isomorphism

3.1.12 [TBD] Kruskell Minimum Spanning Tree

3.1.13 [TBD] Prim Minimum Spanning Tree

3.1.14 [TBD] Louvain (Community Detection)

3.1.15 [TBD] Label propagation (Community Detection)

3.2 Views

The views in this section provide comman 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.

3.2.1 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 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 view functions.

```
// the type of uu is vertex_view<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_view <vertex_id_t<G>, vertex_reference_t<G>, decltype(vvf(u))>`.

```
// the type returned by vertexlist is
// vertex_view<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
}
```

`struct vertex_view<Vid, V, VV>`
`vertex_view` is used to return vertex information. It is used by `vertexlist(g)`, `vertices_breadth_first_search (g,u)`, `vertices_depth_first_search(g,u)` and others. The `id` member always exists.

```
template <class VId, class V, class VV>
struct vertex_view {
  VId id;     // vertex_id_t<G>, always exists
  V   vertex; // vertex_reference_t<t>
  VV  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 ?? . For instance, the second entry, `vertex_view<VId, V>` has two members `{VId id; V vertex;}`.

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

Table 2: `vertex_view` Members

`struct edge_view<VId, Sourced, E, EV>`
`edge_view` is used to return edge information. It is used by `incidence(g,u)`, `edgelist(g)`, `edges_breadth_first_search (g,u)`, `edges_depth_first_search(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_view {
  VId source_id; // vertex_id_t<G>, exists when SourceId==true
  VId target_id; // vertex_id_t<G>, always exists
  E   edge;      // edge_reference_t<G>
  EV  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 **??** . For instance, the second entry, edge_view<VId,true,E> has three members {VId source_id; VId target_id; E edge;}.

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

Table 3: edge_view Members

struct neighbor_view<VId, Sourced, V, VV>
neighbor_view 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 VId. The target_id member always exists.

```
template <class VId, bool Sourced, class V, class VV>
struct neighbor_view {
  VId source_id;  // vertex_id_t<G>
  VId target_id;  // vertex_id_t<G>, always exists
  V   target;     // vertex_reference_t<G>
  VV  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 **??** . For instance, the second entry, neighbor_view<VId,true,E> has three members {VId source_id; VId target_id; V target;}.

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

Table 4: neighbor_view Members

3.2.2 Common Types and Functions for "Search"

The depth_first_search, breadth_first_search, and toplogical_sort searches there are a number of common types and functions that apply to them.

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_depth_first_search(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
}
```

3.2.3 vertexlist Views

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

3.2.4 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_view` on each iteration. Table **??** 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.

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

Table 5: `vertexlist` View Functions

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

Table 6: `incidence` View Functions

3.2.5 neighbors Views

`neighbors` views iterate over a range of edges for a vertex, returning a `vertex_view` of each neighboring target vertex on each iteration. Table **??** 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,u))`	`neighbor_view<VId,false,V,void>`
`for(auto&& [vid,v,val] : neighbors(g,u,vvf))`	`neighbor_view<VId,false,V,VV>`

Table 7: `neighbors` View Functions

3.2.6 edgelist Views

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

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

Table 8: `edgelist` View Functions

3.2.7 depth_first_search Views

depth_first_search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_view` or `edge_view` on each iteration when it is first encountered, depending on the function used. Table **??** 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.

3.2.8 breadth_first_search Views

breadth_first_search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_view` or `edge_view` on each iteration when it is first encountered, depending on the function used. Table **??** shows the functions and their return values.

Example	Return
`for(auto&& [vid,v] : vertices_depth_first_search(g,seed))`	`vertex_view<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_depth_first_search(g,seed,vvf))`	`vertex_view<VId,V,VV>`
`for(auto&& [vid,uv] : edges_depth_first_search(g,seed))`	`edge_view<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_depth_first_search(g,seed,evf))`	`edge_view<VId,false,E,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_depth_first_search(g,seed))`	`edge_view<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_depth_first_search(g,seed,evf))`	`edge_view<VId,true,E,EV>`

Table 9: depth_first_search View Functions

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.

Example	Return
`for(auto&& [vid,v] : vertices_breadth_first_search(g,seed))`	`vertex_view<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_breadth_first_search(g,seed,vvf))`	`vertex_view<VId,V,VV>`
`for(auto&& [vid,uv] : edges_breadth_first_search(g,seed))`	`edge_view<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_breadth_first_search(g,seed,evf))`	`edge_view<VId,false,E,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_breadth_first_search(g,seed))`	`edge_view<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_breadth_first_search(g,seed,evf))`	`edge_view<VId,true,E,EV>`

Table 10: breadth_first_search View Functions

3.2.9 topological_sort Views

topological_sort views iterate over the vertices and edges from a given seed vertex, returning a `vertex_view` or `edge_view` on each iteration when it is first encountered, depending on the function used. Table **??** 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.

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

Table 11: topological_sort View Functions

3.3 Graph Container Interface

The Graph Container Interface defines the primitive concepts, traits, types and functions used to define and access an adacency graph, no matter its internal design and organization. Thus, it is designed to reflect all forms of adjacency graphs including a vector of lists, CSR graph and adjacency matrix, whether they are in the standard or external to the standard.

All algorithms in this proposal require that vertices are stored in random access containers and that `vertex_id_t<G>` is integral, and it is assumed that all future algorithm proposals will also have the same requirements.

The Graph Container Interface is designed to support a wider scope of graph containers than required by the views and algorithms in this proposal. This enables for future growth of the graph data model (e.g. incoming edges on a vertex), or as a framework for graph implementations outside of the standard. For instance, existing implementations may have requirements that cause them to define features with looser constraints, such as sparse vertex_ids, non-integral vertex_ids, or storing vertices in associative bi-directional containers (e.g. std::map or std::unordered_map). Such features require specialized implementations for views and algorithms. The performance for such algorithms will be sub-optimal, but is preferrable to run them on the existing container rather than loading the graph into a high-performance graph container and then running the algorithm on it, where the loading time can far outweigh the time to run the sub-optimal algorithm. To achieve this, care has been taken to make sure that the use of concepts chosen is appropriate for algorithm, view and container.

3.3.1 Concepts

Table **??** summarizes the concepts in the Graph Container Interface, allowing views and algorithms to verify a graph implementation has the expected requirements for an `adjacency_list` or `sourced_adjacency_list`.

Sourced edges have a source_id on them in addition to a target_id. A `sourced_adjacency_list` has sourced edges.

Concept	Definition
`vertex_range<G>`	`vertices(g)` returns a sized, forward_range; `vertex_id(g,ui)` exists
`targeted_edge<G>`	`target_id(g,uv)` and `target(g,uv)` exist
`sourced_edge<G>`	`source_id(g,uv)` and `source(g,uv)` exist
`adjacency_list<G>`	`vertex_range<G>` and `targeted_edge<G,edge<G>>` and `edges(g,_)` functions return a forward_range
`sourced_adjacency_list<G>`	`adjacency_list<G>` and `sourced_edge<G, edge_t<G>>` and `edge_id(g,uv)` exists

Table 12: Graph Container Interface Concepts

3.3.2 Traits

Table **??** summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph's characteristics.

Trait	Type	Comment
`has_degree<G>`	concept	Is the `degree(g,u)` function available?
`has_find_vertex<G>`	concept	Are the `find_vertex(g,_)` functions available?
`has_find_vertex_edge<G>`	concept	Are the `find_vertex_edge(g,_)` functions available?
`has_contains_edge<G>`	concept	Is the `contains_edge(g,uid,vid)` function available?
`define_unordered_edge<G,E> : false_type`	struct	Specialize for edge implementation to derive from `true_type` for unordered edges
`is_unordered_edge<G,E>`	struct	`conjunction<define_unordered_edge<E>, is_sourced_edge<G, E>>`
`is_unordered_edge_v<G,E>`	type alias	
`unordered_edge<G,E>`	concept	
`is_ordered_edge<G,E>`	struct	`negation<is_unordered_edge<G,E>>`
`is_ordered_edge_v<G,E>`	type alias	
`ordered_edge<G,E>`	concept	
`define_adjacency_matrix<G> : false_type`	struct	Specialize for graph implementation to derive from `true_type` for edges stored as a square 2-dimensional array
`is_adjacency_matrix<G>`	struct	
`is_adjacency_matrix_v<G>`	type alias	
`adjacency_matrix<G>`	concept	

Table 13: Graph Container Interface Type Traits

3.3.3 Types

Table **??** summarizes the type aliases in the Graph Container Interface. These are the types used to define the objects in a graph container, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, CSR graph and adjacency matrix.

The type aliases are defined by either a function specialization for the underlying graph container, or a refinement of one of those types (e.g. an iterator of a range). Table **??** describes the functions in more detail.

`graph_value(g)`, `vertex_value(g,u)` and `edge_value(g,uv)` can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types.

Type Alias	Definition	Comment
`graph_reference_t<G>`	`add_lvalue_reference<G>`	
`graph_value_t<G>`	`decltype(graph_value(g))`	optional
`vertex_range_t<G>`	`decltype(vertices(g))`	
`vertex_iterator_t<G>`	`iterator_t<vertex_range_t<G>>`	
`vertex_t<G>`	`range_value_t<vertex_range_t<G>>`	
`vertex_reference_t<G>`	`range_reference_t<vertex_range_t<G>>`	
`vertex_id_t<G>`	`decltype(vertex_id(g))`	
`vertex_value_t<G>`	`decltype(vertex_value(g))`	optional
`vertex_edge_range_t<G>`	`decltype(edges(g,u))`	
`vertex_edge_iterator_t<G>`	`iterator_t<vertex_edge_range_t<G>>`	
`edge_t<G>`	`range_value_t<vertex_edge_range_t<G>>`	
`edge_reference_t<G>`	`range_reference_t<vertex_edge_range_t<G>>`	
`edge_value_t<G>`	`decltype(edge_value(g))`	optional
The following is only available when the optional `source_id(g,uv)` is defined for the edge		
`edge_id_t<G>`	`decltype(pair(source_id(g,uv),target_id(g,uv)))`	

Table 14: Graph Container Interface Type Aliases

3.3.4 Functions

[PHIL: The functions in the Graph Container Interface are semi-stable. New functions are not expected, but overloads may be added or removed for different combinations of vertex_id and references as we refine our use cases.]

Table **??** summarizes the functions in the Graph Container Interface. These are the primitive functions used to access an adjacency graph, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, CSR graph and adjacency matrix.

Functions that have n/a for their Default Implementation must be defined by the author of a Graph Container implementation. `graph_value(g)`, `vertex_value(g,u)` and `edge_value(g,uv)` can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types.

`vertex_id_t<G>` is defined by the type returned by `vertex_id(g)` and it defaults to the difference_type of the underlying container used for vertices (e.g int64_t for 64-bit systems). This is sufficient for all situations. However, there are often space and performance advantages if a smaller type is used, such as int32_t or even int16_t. It is recommended to consider overriding this function for optimal results, assuring that it is also large enough for the number of possible vertices and edges in the application. It will also need to be overridden if the implementation doesn't expose the vertices as a range.

`find_vertex(g,uid)` is constant complexity because all algorithms in this proposal require that `vertex_range_t <G>` is a random access range.

If the concept requirements for the default implementation aren't met by the graph container the function will need to be overridden.

Edgelists are assumed to be either be an edgelist view of an adjacency graph, or a standard range with source_id and target_id values. There is no need for additional functions when a range is used.

3.4 Graph Container Implementation

3.4.1 csr_graph Graph Container

The csr_graph is a high-performance, static graph container that uses Compressed Sparse Row format to store it's vertices, edges and associated values. Once constructed, it cannot be modified.

```
template <class    EV    = void,
          class    VV    = void,
          class    GV    = void,
          integral VId   = uint32_t,
          class    Alloc = allocator<uint32_t>>
class csr_graph;
```

Function	Return Type	Complexity	Default Implementation
`graph_value(g)`	`graph_value_t<G>`	constant	n/a, optional
`vertices(g)`	`vetex_range_t<G>`	constant	n/a
`vertex_id(g,ui)`	`vetex_id_t<G>`	constant	`ui - begin(vertices(g))` Override to define a different `vertex_id_t<G>` type (e.g. int32_t).
`vertex_value(g,u)`	`vertex_value_t<G>`	constant	n/a, optional
`degree(g,u)`	`integral`	constant	`size(edges(g,u))` if `sized_range<vertex_edge_range_t<G>>`
`find_vertex(g,uid)`	`vertex_iterator_t<G>`	constant	`begin(vertices(g))+ uid` if `random_access_range<vertex_range_t<G>>`
`edges(g,u)`	`vertex_edge_range_t<G>`	constant	n/a
`edges(g,uid)`	`vertex_edge_range_t<G>`	constant	`edges(g,*find_vertex(g,uid))`
`target_id(g,uv)`	`vertex_id_t<G>`	constant	n/a
`target(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ target_id(g, uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_value(g,uv)`	`edge_value_t<G>`	constant	n/a, optional
`find_vertex_edge(g,u,vid)`	`vertex_edge_t<G>`	linear	`find(edges(g,u), [](uv)target_id(g,uv)==vid;})`
`find_vertex_edge(g,uid,vid)`	`vertex_edge_t<G>`	linear	`find_vertex_edge(g,*find_vertex(g,uid),vid)`
`contains_edge(g,uid,vid)`	`bool`	constant	`uid < size(vertices(g))&& vid < size(vertices(g))` if `is_adjacency_matrix_v<G>`.
		linear	`find_vertex_edge(g,uid)!= end(edges(g,uid))` otherwise.
The following are only available when the optional `source_id(g,uv)` is defined for the edge			
`source_id(g,uv)`	`vertex_id_t<G>`	constant	n/a, optional
`source(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ source_id(g,uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_id(g,uv)`	`edge_id_t<G>`	constant	`pair(source_id(g,uv),target_id(g,uv))`

Table 15: Graph Container Interface Functions

3.4.2 csr_partite_graph Graph Container (In Design)

[PHIL: This is experimental]

The `csr_partite_graph` extends `csr_graph` to have multiple partitions, where each partition defines a different value type for the vertex and edge. The same template arguments are used, but it also expects that the VV and EV arguments are `std::variant`, and the number of types in each is the same. The number of types in the variants define the number of partitions. The edge types apply to the outgoing edges of the vertices in the same partition. `std::monostate` can be used if no value is needed for a vertex or edge in a partition.

Example usage

```
using VV = std::variant<int,double,bool>;
using EV = std::variant<int,int,std::monostate>; // no outgoing edges in the final
    partition
using G  = csr_partite_graph<EV, VV>;
G g = ...; // construct g with data
for(size_t p = 0; p < partition_size(g); ++p) {
  for(auto&& [uid,u] : partition(g,p)) {
    for(auto&& [vid,uv] : incidence(g,u)) {
      // do interesting things with uv
```

```
        }
    }
}
```

4 Design - Graph Container Author

4.1 Customization Points

[PHIL: Describe CPO's in general that can apply to both niebloids and tag_invoke. tag_invoke may have some controversy because it is so new.]

We follow the lead of P2300r5 [?] section 5.9 for our implementation of customization points. The text in this section has been taken from that paper and modified to be specific to graphs.

The contemporary technique for customization in the Standard Library is customization point objects. A customization point object, will it look for member functions and then for nonmember functions with the same name as the customization point, and calls those if they match. This is the technique used by the C++20 ranges library. However, it has several unfortunate consequences:

1. It does not allow for easy propagation of customization points unknown to the adaptor to a wrapped object, which makes writing universal adapter types much harder - and this proposal uses quite a lot of those.

2. It effectively reserves names globally. Because neither member names nor ADL-found functions can be qualified with a namespace, every customization point object that uses the ranges scheme reserves the name for all types in all namespaces.

This paper proposes to instead use the mechanism described in tag_invoke: A general pattern for supporting customisable functions: `tag_invoke`; the wording for `tag_invoke` has been incorporated into the proposed specification in this paper.

In short, instead of using globally reserved names, `tag_invoke` uses the type of the customization point object itself as the mechanism to find customizations. It globally reserves only a single name - `tag_invoke` - which itself is used the same way that ranges-style customization points are used. All other customization points are defined in terms of `tag_invoke`. For example, the customization for `std::graph::vertices(g)` will call `tag_invoke(std::graph::vertices, g )`, instead of attempting to invoke `g.vertices()`, and then `vertices(g)` if the member call is not valid.

Using `tag_invoke` has the following benefits:

1. It reserves only a single global name, instead of reserving a global name for every customization point object we define.

2. It is possible to propagate customizations to a subobject, because the information of which customization point is being resolved is in the type of an argument, and not in the name of the function:

   ```
   // forward most customizations to a subobject
   template<typename Tag, typename ...Args>
   friend auto tag_invoke(Tag && tag, wrapper & self, Args &&... args) {
       return std::forward<Tag>(tag)(self.subobject, std::forward<Args>(args)...);
   }

   // but override one of them with a specific value
   friend auto tag_invoke(specific_customization_point_t, wrapper & self) {
       return self.some_value;
   }
   ```

3. It is possible to pass those as template arguments to types, because the information of which customization point is being resolved is in the type.

4.2 Function Specialization

5 Specification

6 Library introduction [library]

7 Graph introduction [graph]

8 algorithms [graph.algorithms]

9 views [graph.views]

10 customization points [graph.functions]

11 Old Introduction (Content after this will be moved above or removed)

11.1 Namespaces and Headers

Graph algorithms, views and containers are unique and not easily interchanged with other elements of the standard library. For instance, graph algorithms wouldn't be used on a non-graph range. For this reason it is recommended to have them in their own namespaces. Suggestions for namespaces are `std::graph` and `std::ranges::graph` for the root namespace. The graph namespaces proposed include

`std::tag_invoke`

`std::graph`

`std::graph::views`

Proposed headers include

`<graph>`

`<graph_view>`

`<graph_algorithm>`

`<csr_graph>`

12 Technical Specifications

TBA

12.1 Header `<graph>` synopsis [graph.syn]

```
namespace std::graph {

// ...

template <typename G>
auto vertices(G&& g);

template <typename G>
auto vertex_id(G&& g, vertex_iterator_t<G>);

template <typename G>
auto vertex_value(G&& g, vertex_reference_t<G>);

// ...

template <class G>
```

```
inline constexpr bool is_adjacency_matrix_v = false;

template <class E>
inline constexpr bool is_undirected_edge_v = false;

template <class G>
concept vertex_range = ranges::forward_range<vertex_range_t<G>> &&
                                   ranges::sized_range<vertex_range_t<G>> &&
requires(G&& g, vertex_iterator_t<G> ui) {
  { vertices(g) } -> ranges::forward_range;
  vertex_id(g, ui);
};

template <class G, class ER>
concept targeted_edge = ranges::forward_range<ER> &&
requires(G&& g, ranges::range_reference_t<ER> uv) {
  target_id(g, uv);
  target(g, uv);
};

template <class G, class ER>
concept sourced_edge =
requires(G&& g, ranges::range_reference_t<ER> uv) {
  source_id(g, uv);
  source(g, uv);
};

template <class G>
concept adjacency_list = vertex_range<G> &&
                          targeted_edge<G, vertex_edge_range_t<G>> &&
requires(
      G&& g, vertex_reference_t<G> u, vertex_id_t<G> uid, ranges::range_reference_t<
          vertex_edge_range_t<G>> uv) {
  { edges(g, u) } -> ranges::forward_range;
  { edges(g, uid) } -> ranges::forward_range;
};

template <class G>
concept sourced_adjacency_list = adjacency_list<G> && sourced_edge<G,
    vertex_edge_range_t<G>> &&
      requires(G&& g, edge_reference_t<G> uv) {
  edge_id(g, uv);
};

template <class G>
concept undirected_incidence_graph = sourced_adjacency_list<G> &&
    is_undirected_edge_v<edge_t<G>>;

template <class G>
concept directed_incidence_graph = !undirected_incidence_graph<G>;

template <class G>
concept adjacency_matrix = is_adjacency_matrix_v<G>;

template <class G>
concept has_degree = requires(G&& g, vertex_reference_t<G> u) {
  {degree(g, u)};
};

template <class G>
concept has_find_vertex = requires(G&& g, vertex_id_t<G> uid) {
  { find_vertex(g, uid) } -> forward_iterator;
};

template <class G>
concept has_find_vertex_edge = requires(G&& g, vertex_id_t<G> uid, vertex_id_t<G>
    vid, vertex_reference_t<G> u) {
  { find_vertex_edge(g, u, vid) } -> forward_iterator;
  { find_vertex_edge(g, uid, vid) } -> forward_iterator;
};

template <class G>
concept has_contains_edge = requires(G&& g, vertex_id_t<G> uid, vertex_id_t<G> vid)
    {
  { contains_edge(g, uid, vid) } -> convertible_to<bool>;
};

}
```

The following is a synopsis of the functions and classes above.

```
template <typename G>
auto vertices(G&& g);
```

- Preconditions: TBA.
- Effects: TBA.
- Complexity: TBA.
- Returns: TBA.
- Remarks: TBA.

```
template <typename G>
auto vertex_id (G&& g, vertex_iterator_t<G>);
```

- Preconditions: TBA.
- Effects: TBA.
- Complexity: TBA.
- Returns: TBA.
- Remarks: TBA.

```
template <typename G>
auto vertex_value (G&& g, vertex_reference_t<G>);
```

- Preconditions: TBA.
- Effects: TBA.
- Complexity: TBA.
- Returns: TBA.
- Remarks: TBA.

12.2 Header `<graph_view>` synopsis [graph.syn]

```
namespace std::graph::views {

// ...

template <typename G>
auto vertexlist(G&& g);

template <typename G>
auto incidence(G&& g, vertex_reference_t<G>);

template <typename G>
auto neighbors(G&& g, vertex_reference_t<G>);

template <typename G>
auto edgelist(G&& g);

// ...
// vertices_depth_first_search
// edges_depth_first_search
// sourced_edges_depth_first_search
// vertices_breadth_first_search
// edges_breadth_first_search
}
```

12.3 Header `<graph_algorithm>` synopsis [graph.syn]

```
namespace std::graph {

// ...tag_invoke


// ...
}
```

12.4 Header `<csr_graph>` synopsis [graph.syn]

```
namespace std::graph {

// ...

template<class EV, class VV, class GV, class VId, class Alloc>
class csr_graph;

// ...
}
```

13 Acknowledgements

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada. The authors wish to further thank the members of SG19 for their contributions.