D1709R4 Graph Library

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

Document Number: D1709R4
Date: December, 2022 (goal) (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

P1709R4

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.

P1709R3

A simple status revision to say a major change is coming soon.

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 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.

1.3 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.

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.

Parallel versions of the algorithms are not included for several reasons. The executors proposal in P2300r5 [1] 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.

Incoming edges on a vertex are not included, though it is hoped that a future proposal will be made for them.

The algorithms and views in this proposal expect that vertex_ids are densly assigned in a random access range, but it does not exclude the possibility of sparsely-defined vertex_ids stored in containers like `std::map` or `std::unordered_map` in future proposals.

The algorithms and views in this proposal expect that vertex_ids are integal, but it does not exclude non-integral or user-defined types in future proposals.

Hypergraphs are not supported.

1.4 Impact on the Standard

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

1.5 Interaction wtih Other Papers

There is no interaction with other proposals to the standard.

1.6 Implementation Experience

The github stdgraph repository contains an implementation for this proposal.

1.7 Usage Experience

1.8 Deployment Experience

1.9 Performance Considerations

The algorithms are being ported from NWGraph to the stdgraph implementation used for this proposal. Performance analysis from those algorithms can be found in the peer-reviewed paper for NWGraph [2].

1.10 Prior Art

boost::graph has been an important C++ graph implementation since 2001. It was developed with the goal to provide a modern library that addressed all the needs someone would want of a graph library. It is still a viable library used today, attesting to the value it brings.

However, boost::graph was written using C++98 and added many unneccesary abstractions that makes it difficult to use by a casual developer.

Andrew is a co-author of boost::graph.

NWGraph ([3] and [2]) was published in 2022 by Lumsdaine et al, bringing additional experience gained since creating boost::graph, to create a modern graph library using C++20 that was more accessible to the average developer with the latest algorithms.

While NWGraph made important strides, there are some areas that don't conform to established expectations in the Standard Library. Examples include using parameter packs to define multiple value types for an edge, rather than a single type; a focus on only defing value types on edges, but not vertices or the graph; there is no defined way to integrate existing graph containers to be used by the algorithms.

This proposal takes the best of NWGraph, with previous work done for P1709 to define a Graph Container Interface, to provide a library that embraces performance, ease-of-use and the ability to use the algorithms and views on externally defined graph containers.

1.11 Alternatives

There are no known alternative graph library we're aware of that meets the same requirements and uses concepts and ranges from C++20.

1.12 Feature Test Macro

The `__cpp_lib_graph` feature test macro is recommended to represent all features in this proposal including algorithms, views, concepts, traits, types, functions and graph container(s).

Future features could extend the name for new features. For instance, `__cpp_lib_graph_partite_container` could be used if an n-partite graph container were added in later C++ versions.

1.13 Freestanding

We believe this library can be used in a freestanding C++ implementation.

1.14 Namespaces

Graph containers and their views and algorithms are not interchangeable with existing containers and algorithms. It is believed that they are unique enough to warrant their own namespaces and is used in this proposal.

```
std::graph

std::graph::views
```

Alternative locations for the above respective namespaces could also be as follows:

`std::ranges`

`std::ranges::views`

2 Design - Introduction

Table 1 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`	Vertex Iterator. `ui` is the source (or only) vertex.
		`first`, `last`	`vi` is the target vertex.
`VVF`		`vvf`	Vertex Value Function: vvf(u) → 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) → value

Table 1: Naming Conventions for Types and Variables

2.1 Graph Definition

A *graph* [4] $G = (V, E)$ is a set of *vertices* [4] V, **points** in a space, and *edges* [4] E, **links** between these vertices. Edges may or may not be **oriented**, that is, *directed* [4] or *undirected* [4], respectively. Moreover, edges may be *weighted* [4], 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.

2.1.1 Adjacency List

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 Algorithm Concepts

Additional concepts used by algorithms.

```
template <class G, class F>
concept edge_weight_function = // e.g. weight(uv)
      copy_constructible<F> && is_arithmetic_v<invoke_result_t<F, edge_reference_t<G>>>;
```

```
// queueable<Q> can represent std::queue and std::priority_queue
template <class Q>
concept queueable = requires(Q&& q, Q::value_type value) {
  Q::value_type;
  Q::size_type;
  Q::reference;

  {q.top()};
  {q.push(value)};
  {q.pop()};
  {q.empty()};
  {q.size()};
};
```

3.2 Algorithms

All algorithms are customization points.

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

3.2.1 Dijkstra's Shortest Paths and Shortest Distances

Dijkstra's algorithm [**?**] is a single-source, shortest paths algorithm. It finds the shortest paths and their weighted distances to all vertices connected to a single seed vertex.

```
template <adjacency_list              G,
          ranges::random_access_range Distance,
          ranges::random_access_range Predecessor,
          class EVF    = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>,
          queueable Q =
              priority_queue<
                  weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>,
                  vector<weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>>,
                  greater<weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>>>>
requires ranges::random_access_range<vertex_range_t<G>> &&                    //
      integral<vertex_id_t<G>> &&                                             //
      is_arithmetic_v<ranges::range_value_t<Distance>> &&                     //
      convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> && //
      edge_weight_function<G, EVF>
constexpr void dijkstra_shortest_paths(
      G&&             g,           // graph
      vertex_id_t<G> seed,         // starting vertex_id
      Distance&       distance,    // out: distance[uid] of vertex_id uid from seed
      Predecessor&    predecessor, // out: predecessor[uid] of vertex_id uid in path
      EVF             weight_fn = [](edge_reference_t<G> uv)  // weight_fn(uv) -> 1
                                      { return ranges::range_value_t<Distance>(1); },
      Q               q = Q()
      );
```

Complexity	O(E+2V), where E is the number of edges connected to `seed`, directly or indirectly, and 2V is the time to initialize the `distance` and `predecessor` ranges.
Preconditions	`seed >= 0 && seed < size(vertices(g))` `size(distance)>= size(vertices(g))`; caller must pre-extend `size(predecessor)>= size(vertices(g))`; caller must pre-extend
Constraints	Values returned by `weight_fn` must be non-negative.
Effects	`distance[uid]` will be the shortest, weighted distance of vertex_id `uid` from `seed`. If `uid` is not connected to `seed` by any edges then it will have a value of `numeric_limits<range_value_t<Distance>>::max()`. `predecessor[uid]` will have the predecing vertex_id of `uid` in the weighted shortest path to `seed`. If `uid` is not connected to `seed` by any edges then it will have a value of `numeric_limits<range_value_t<Predecessor>>::max()`.

[PHIL: There's a hidden complexity of O(E+4V), not just O(E+2V), because we're requiring the caller to pre-extend `distance` and `predecessor` which also initializes the ranges. If there were customization points for `resize(c,n[,init])`, `clear(c)` and `reserve(c,n)` we could reduce the complexity to O(E+2V) overall.]

[PHIL: Describe the pros and cons of different kinds of queues]

The default weight function `weight_fn` returns a value of 1. When that is used, `distances[uid]` will have the shortest number of edges between vertex `uid` and vertex `seed`. The distance from `seed` to itself is zero.

If the caller wishes to use a different queue other than `priority_queue`, the queue will need to have elements of `weighted_vertex` which is used internally by the algorithm.

```
template <class G, class W>
struct weighted_vertex {
  vertex_id_t<G> vertex_id = vertex_id_t<G>();
  W              weight    = W();
  constexpr auto operator<=>(const weighted_vertex&) const noexcept; // compare vertex_id
};
```

The `dijkstra_shortest_distances` function is the same as `dijkstra_shortest_paths` except that it doesn't include the `predecessor` parameter and has a complexity of O(E+V).

```
template <adjacency_list              G,
          ranges::random_access_range Distance,
          class EVF   = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>,
          queueable Q =
              priority_queue<
                  weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>,
                  vector<weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>>,
                  greater<weighted_vertex<G,invoke_result_t<EVF,edge_reference_t<G>>>>>>
requires ranges::random_access_range<vertex_range_t<G>> &&                    //
      integral<vertex_id_t<G>> &&                                             //
      is_arithmetic_v<ranges::range_value_t<Distance>> &&                     //
      edge_weight_function<G, EVF>
constexpr void dijkstra_shortest_distances(
      G&&            g,           // graph
      vertex_id_t<G> seed,        // starting vertex_id
      Distance&      distance,    // out: distance[uid] of vertex_id uid from seed
      EVF            weight_fn = [](edge_reference_t<G> uv)  // weight_fn(uv) -> 1
                                      { return ranges::range_value_t<Distance>(1); },
      Q              q = Q()
      );
```

3.2.2 Bellman-Ford Shortest Paths

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

3.2.3 Connected Components

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

3.2.4 Strongly Connected Components

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

3.2.5 Biconnected Components

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

3.2.6 Articulation Points

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

3.2.7 Minimum Spanning Tree

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

3.2.8 [TBD] Page Rank

3.2.9 [TBD] Betweenness Centrality

3.2.10 [TBD] Triangle Count

3.2.11 [TBD] Subgraph Isomorphism

3.2.12 [TBD] Kruskell Minimum Spanning Tree

3.2.13 [TBD] Prim Minimum Spanning Tree

3.2.14 [TBD] Louvain (Community Detection)

3.2.15 [TBD] Label propagation (Community Detection)

3.3 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.

3.3.1 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 2 . 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

A useful type alias for copying vertex values (excluding the vertex reference) is also available.

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

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 3 . For instance, the second entry, edge_view<VId,true,E> has three members {VId source_id; VId target_id; E edge;}.

A useful type alias for copying edge values (excluding the edge reference) is also available.

```
template <class VId, class EV>
using copyable_edge_t = edge_view<VId, true, void, EV>; // {source_id, target_id [, value]}
```

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 4 . 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

3.3.2 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
}
```

3.3.3 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_view` on each iteration. Table 5 shows the vertexlist functions overloads and their return values. `first` and `last` are vertex iterators.

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

3.3.4 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_view` on each iteration. Table 6 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.

3.3.5 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 7 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,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

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

3.3.6 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_view` on each iteration. Table 8 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

3.3.7 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 9 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_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

3.3.8 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 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.

3.3.9 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 11 shows the functions and their return values.

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

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

3.4 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.

3.4.1 Concepts

Table 12 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.

3.4.2 Traits

Table 13 summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph's characteristics.

3.4.3 Types

Table 14 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.

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
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>>

Table 12: Graph Container Interface Concepts

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

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 15 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.

3.4.4 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 15 summarizes the functions in the Graph Container Interface. These are the primitive functions used to access an adacency 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

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

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.

3.5 Graph Container Implementation

3.5.1 csr_graph Graph Container

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

The following listing shows the public interface for the `csr_graph`.

When a value type template argument (EV, VV, GV) is void then no extra overhead is incurred for it. The selection of the VId template argument impacts the inter storage requirements. If you have a small graph where the number of vertices is less than 256, and the number of edges is less than 256, then a `uint8_t` would be sufficient.

Only the constructors, destructor and assignment operators for `csr_graph` are public. No other member functions or types are exposed. All other types are only accessible through the types and functions in the Graph Container Interface.

The `ERng` template parameter is assumed to be a range of `copyable_edge_t<VId,EV>`. If it isn't, an `EProj` projection function must be passed to to convert the ERng value type to a `copyable_edge_t<VId,EV>`.

Likewise, the `VRng` template parameter is assumed to be a range of `copyable_vertex_t<VId,VV>`. If it isn't, a `VProj` projection function must be passed to convert the VRng value type to a `copyable_vertex_t<VId,VV>`.

```
template <class    EV     = void,     // Edge Value type
          class    VV     = void,     // Vertex Value type
          class    GV     = void,     // Graph Value type
          integral VId    = uint32_t, // vertex id type
          integral EIndex = uint32_t, // edge index type
          class    Alloc  = allocator<uint32_t>> // for internal containers
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

A fuller description of csr_graph, including its constructors, can be found in the <csr_graph header section.

3.5.2 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

Customization Points can be implemented in one of three ways:

1. A static function object that can be overridden by using ADL. This is the current technique used in existing standard libraries.
2. P2547 Language Support for Customisable Functions, an active proposal for C++26.
3. P1895 tag_invoke: A general pattern for supporting customisable functions. This has been dismissed as a viable solution for use in the standard for a variety of reasons identified in P2547.

We are monitoring the progress of P2547 and will use that if is accepted into the Standard. In the interim, the current library prototype implementation is using `tag_invoke` for convenience, but it will be replaced with the function object or P2547 functionality at a future date.

5 Technical Specifications

This section is woefully incomplete and likely has errors when considering what needs to be included in the Standard. The intention is to give additional detailed information that isn't covered in earlier sections, for those who wish to know more.

We anticipate getting direction from the Library Working Group to fill this out more completely and acurately.

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

5.2 Functions

All functions are customization points.

```
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>);

} // namespace std::graph
```

5.3 Type Aliases

```
namespace std::graph {

// ...

} // namespace std::graph
```

5.4 Traits

```
namespace std::graph {

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>;
} // namespace std::graph
```

5.5 Concepts

```
namespace std::graph {

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, edge_t<G>> &&
      requires(G&& g, vertex_reference_t<G> u, vertex_id_t<G> uid) {
  { 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, edge_t<G>> &&
```

```
      requires(G&& g, edge_reference_t<G> uv) {
  edge_id(g, uv);
};
} // namespace std::graph
```

5.6 Header <graph_view> synopsis [graph.syn]

```
namespace std::graph {

... // view classes go here

} // namespace std::graph

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
// sourced_edges_breadth_first_search

// vertices_topological_sort
// edges_toplological_sort
// sourced_edges_topological_sort
} // namespace std::graph::views
```

5.7 Header <graph_algorithm> synopsis [graph.syn]

```
namespace std::graph {

} // namespace std::graph
```

5.8 Header <csr_graph> synopsis [graph.syn]

```
namespace std::graph {

template <class EV        = void,                // edge value type
          class VV        = void,                // vertex value type
          class GV        = void,                // graph value type
          integral VId    = uint32_t,            // vertex id type
          integral EIndex = uint32_t,            // edge index type
          class Alloc     = allocator<uint32_t>> // for internal containers
class csr_graph {
public: // Types
```

```
  using graph_type = csr_graph<EV, VV, GV, VId, EIndex, Alloc>;

  using edge_value_type   = EV;
  using vertex_value_type = VV;
  using graph_value_type  = GV;
  using value_type        = GV;

  using vertex_id_type = VId;

public: // Construction/Destruction
  constexpr csr_graph()                 = default;
  constexpr csr_graph(const csr_graph&) = default;
  constexpr csr_graph(csr_graph&&)      = default;
  constexpr ~csr_graph()                = default;

  constexpr csr_graph& operator=(const csr_graph&) = default;
  constexpr csr_graph& operator=(csr_graph&&)      = default;

  template <ranges::forward_range ERng, class EProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV>
  constexpr csr_graph(const ERng&  erng,
                      EProj        eprojection,
                      const Alloc& alloc = Alloc());

  template <ranges::forward_range ERng, ranges::forward_range VRng,
            class EProj = identity,     class VProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV> &&
           copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>, VId, VV>
  constexpr csr_graph(const ERng&  erng,
                      const VRng&  vrng,
                      EProj        eprojection = {},
                      VProj        vprojection = {},
                      const Alloc& alloc       = Alloc());

  constexpr csr_graph(const initializer_list<copyable_edge_t<VId, EV>>& ilist,
                      const Alloc& alloc = Alloc());

};

} // namespace std::graph
```

When `GV` is not `void` the following constructors are also available.

```
  template <ranges::forward_range ERng, class EProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV>
  constexpr csr_graph(const graph_value_type& value,
                      const ERng&             erng,
                      EProj                   eprojection,
                      const Alloc&            alloc = Alloc());

  template <ranges::forward_range ERng, class EProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV>
  constexpr csr_graph(graph_value_type&& value,
                      const ERng&        erng,
                      EProj              eprojection,
                      const Alloc&       alloc = Alloc());


  template <ranges::forward_range ERng, ranges::forward_range VRng,
            class EProj = identity,     class VProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV> &&
           copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>, VId, VV>
  constexpr csr_graph(const graph_value_type& value,
```

```
                    const ERng&             erng,
                    const VRng&             vrng,
                    EProj                   eprojection = {},
                    VProj                   vprojection = {},
                    const Alloc&            alloc       = Alloc());

template <ranges::forward_range ERng, ranges::forward_range VRng,
          class EProj = identity,     class VProj = identity>
requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV> &&
         copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>, VId, VV>
constexpr csr_graph(graph_value_type&& value,
                    const ERng&        erng,
                    const VRng&        vrng,
                    EProj              eprojection = {},
                    VProj              vprojection = {},
                    const Alloc&       alloc       = Alloc());
```

6 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.

References

[1] Dominiak Evtushenko Baker Teodorescu Howes Shoop Garland Niebler and Lelbach. P2300r5 std::execution. "https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r5.html".

[2] Lumsdaine D'Alessandro Deweese Firoz Liu McMillan Ratzloff Zalewski. Nwgraph: A library of generic graph algorithms and data structures in c++20. "https://drops.dagstuhl.de/opus/volltexte/2022/16259/".

[3] Lumsdaine D'Alessandro Deweese Firoz Liu McMillan Ratzloff Zalewski. Nwgraph library code. "https://github.com/pnnl/NWGraph".

[4] Jonathan L. Gross and Jay Yellen, editors. *Handbook of Graph Theory*. CRC Press, 2004.