P1709R2: Graph Library

Date:	2020-01-13
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21

Authors:	Phillip Ratzloff (SAS Institute) Richard Dosselmann (U of Regina) Michael Wong (Codeplay) Matthew Galati (SAS Institute) Andrew Lumsdaine (PNNL / University of Washington) Jens Maurer Domagoj Saric Jesun Firoz Kevin Deweese
Contributors:	
Emails:	phil.ratzloff@sas.com dosselmr@cs.uregina.ca michael@codeplay.com Matthew.Galati@sas.com
Reply to:	phil.ratzloff@sas.com

Introduction

This document proposes the addition of a (general) **graph** algorithms and data structure to the C++ **containers** 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**. **Artificial intelligence** (AI), a subset of ML, in particular has received a great deal of attention in recent years.

A *graph* $G = (V, E)$ is a set of *vertices*, **points** in a space, and *edges*, **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,** (static) **array** and (dynamic) **list**, each having the typical advantages and disadvantages associated with static and dynamic data structures.

This paper presents an **interface** of the proposed graph algorithms and data structures. This should be considered a proof of concept.

Revision History

Revision	Description
P1709R2	Define the **uniform API** for undirected & directed algorithms (an extended API also exists for directed graphs). Added **concepts** for undirected, directed & bidirected graphs. Refined **DFS & BFS range** definitions from prototype experience. Refined **Shortest Paths** & **Transitive Closure** algorithms from input & prototype experience.
P1709R1	Rewrite with a focus on a **pure 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.

Motivation

A graph data structure, used in ML and other **scientific** domains, as well as **industrial** and **general** programming, does **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**, an *Entity Relationship Diagram* (ERD) or *Data Flow Diagram* (DFD) is a graph. In the realm of **social media**, a graph represents a social network.

Impact on the Standard

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

Design Proposals

Goals & Background

Graphs are used in a wide variety of situations. To meet the varied demands there are a number of different characteristics with different behavior and performance to meet requirements. This section identifies the different types of graphs and introduces the goals of this proposal. The remaining sections provide the details.

The characteristics that are often used to describe graphs include the following:

1. **Property Graphs**: The user can define properties, or values, on edges, vertices and the graph itself.

   This proposal supports optional user-defined types for edge, vertex and graph types. Any C++ type is allowed, including class, struct, union, tuple, enum and scalers.
2. **Directed (forward-only and bi-directional) and Undirected Graphs**: Edges can represent a direction, in-vertex ànd out-vertex, or can be undirected. Directed graphs also have a designation of forward-only or bi-directional.

   This proposal supports directed forward-only, directed bi-directional, and undirected graphs.
3. **Adjacency List | Adjacency Array | Adjacency Matrix**: How edges are represented/implemented has an impact on performance when modifying the graph or executing algorithms, often conflicting with each other. These are design decisions made by developers for their situation.

   An Adjacency List uses linked lists to store edges and adapts to change well, an Adjacency Array stores all edges in a single “array” (e.g. std::vector) with a balance between change and good performance, and Adjacency Matrix stores all combinations of edges in a dense 2-dimensional array for performance and space advantages for dense graphs.

   All forms are supported in this proposal.
4. **Single-edge and Multi-edge Graphs (multigraphs)**: Each pair of vertices can have one or more edges between them.

   This proposal supports both single- and multi-edge graphs. No special attention is given to prevent multiple edges between two vertices. The Adjacency Matrix prevents multiple edges by its nature.
5. **Acyclic and Cyclic Graphs**: Cyclic graphs include paths that trace one or more edges from one vertex back to itself, while acyclic graphs have no such paths.

   This proposal supports both acyclic and cyclic graphs. No special attention is given to prevent cyclic graphs. Detection of cycles requires the depth-first search algorithm.

6. **Sparse and Dense Graphs**: A sparse graph is loosely defined as $|E| < |V|^2 / 2$, and a dense graph is the opposite. Some algorithms may work better with one or the other and will be noted in the algorithms's description.

The goal of any graph library is to be able to be as flexible as possible, making necessary compromises as needed. A challenge is to manage the list of various combinations, while recognizing that some are not reasonable or possible.

An important concept in the design is the idea that a graph is a range of ranges, more specifically a range of vertices, each with a range of edges. For edge-focused algorithms, a graph can also be viewed as a range of edges, with two vertices as properties of the edge. Toward that end, a user of the library might write code similar to this:

```
using G = … <graph definition>
G g(...); // graph construction
for(auto& u : g)              // thru vertices of graph g
  for(auto& uv : edges(g,u))  // thru edges of vertex u
    ...;                      // do something interesting with uv & u

for(auto& uv : edges(g)) {    // thru all edges in graph
   auto& u = in_vertex(g,uv);
   auto& v = out_vertex(g,uv);
   ...;                       // do something interesting with uv, u, v
}
```

Another important concept in the design is an emphasis on a uniform API that allows algorithms to work with both directed and undirected graphs. In general, a directed graph uses the same functions as an undirected graph with the simple convention that functions alias their equivalent "out" version. For instance, in the example above edges(g,u) would be an alias for out_edges(g,u) in a directed graph. For an undirected graph, edges(g,u) would be the actual function and out_edges(g,u) would not exist and generate a compile error if used.

For algorithms that only allow directed or undirected graphs, or where there are different implementations for each, concept definitions allow algorithm authors to define their requirements for the type of graph they support. These are covered in the Concept Definitions section.

The API is presented as a set of templatized free functions, allowing adaptation to existing graph data structures. There is also a technical reason that makes it problematic to use a class-based design, which is covered in the Design Notes at the end of this document.

A vertex key is defined as a way to find a vertex. Its type is dependent on the underlying container used to store the vertex. When a vector or deque is used, the key is an index into the container. When a map-like container is used, the container's key type is used.

The ranges and algorithms in this paper assume vertex keys are always integral and that vertices are kept in contiguous containers (e.g. vector or array) by specifying the following:

```
requires integral<vertex_key_t<G>>
&& contiguous_range<vertex_range_t<G>>
```

This reflects the state of the current prototype. It's expected to be relaxed in future papers to allow keys of any type and for vertices to be stored in any random-access container with an implied key (e.g. an index like used in a vertex, array or deque), or user-defined key (e.g. map or unordered_map).

Uniform API

The uniform API is the set of types and functions that can be used for algorithms that support both directed and undirected graphs. Functions that are direction neutral in undirected graphs, but are directional for directed graphs, are aliased to the "out" functions in directed graphs.

The exception to this rule is in_vertex/out_vertex and in_vertex_key/out_vertex_key functions. An edge is always associated with two vertices. In and out are used as a general convention but should not be considered that they imply direction when used in an undirected graph.

Types

Graph Types

```
template <typename G>
using graph_value_t = typename G::graph_user_value;
template <typename G>
using const_graph_value_t = typename const G::graph_user_value;

template <typename G>
using edge_range_t = typename G::edge_range;
template <typename G>
using const_edge_range_t = typename G::const_edge_range;

template <typename G>
using edge_iterator_t = typename G::edge_iterator;
template <typename G>
using const_edge_iterator_t = typename G::const_edge_iterator;

template <typename G>
using edge_sentinel_t = typename G::edge_sentinel;

template <typename G>
using edge_size_t = typename G::edge_size_type;
```

Vertex Types

```
template <typename G>
using vertex_t = typename G::vertex_type;
template <typename G>
using const_vertex_t = typename G::const_vertex_type;

template <typename G>
using vertex_key_t = typename G::vertex_key_type;
template <typename G>
using const_vertex_key_t = typename const G::vertex_key_type;

template <typename G>
using vertex_value_t = typename G::vertex_user_value;
template <typename G>
using const_vertex_value_t = typename const G::vertex_user_value;

template <typename G>
using vertex_range_t = typename G::vertex_range;
template <typename G>
using const_vertex_range_t = typename G::const_vertex_range;

template <typename G>
using vertex_iterator_t = typename G::vertex_iterator;
template <typename G>
using const_vertex_iterator_t = typename G::const_vertex_iterator;

template <typename G>
using vertex_sentinel_t = typename G::vertex_sentinel;

template <typename G>
using vertex_size_t = typename G::vertex_size_type;
```

Edge Types

```
template <typename G>
using edge_t = typename G::edge_type;
template <typename G>
using const_edge_t = typename G::const_edge_type;

template <typename G>
using edge_value_t = typename G::edge_user_value_type;
template <typename G>
using const_edge_value_t = typename const G::edge_user_value_type;

template <typename G>
using vertex_edge_range_t = typename G::vertex_edge_range;
```

```
template <typename G>
using const_vertex_edge_range_t = typename G::const_vertex_edge_range;

template <typename G>
using vertex_edge_iterator_t = typename G::vertex_edge_iterator;
template <typename G>
using const_vertex_edge_iterator_t = typename G::const_vertex_edge_iterator;

template <typename G>
using vertex_edge_sentinel_t = typename G::vertex_edge_sentinel;

template <typename G>
using vertex_edge_size_t = typename G::vertex_edge_size_type;
```

Functions

```
template <typename T>
constexpr auto value(T& gve) noexcept -> decltype(user_value(gve));
```

Graph Functions

```
template <typename G>
constexpr auto vertices(G& g) noexcept -> vertex_range_t<G>;
template <typename G>
constexpr auto vertices(G const& g) noexcept -> const_vertex_range_t<G>;

template <typename G>
constexpr auto vertices_size(G const& g) noexcept -> vertex_size_t<G>;

template <typename G>
constexpr auto begin(G& g) noexcept -> vertex_iterator_t<G>;
template <typename G>
constexpr auto begin(G const& g) noexcept -> const_vertex_iterator_t<G>;
template <typename G>
constexpr auto cbegin(G const& g) noexcept -> const_vertex_iterator_t<G>;

template <typename G>
constexpr auto end(G& g) noexcept -> vertex_iterator_t<G>;
template <typename G>
constexpr auto end(G const& g) noexcept -> const_vertex_iterator_t<G>;
template <typename G>
constexpr auto cend(G const& g) noexcept -> const_vertex_iterator_t<G>;


template <typename G>
constexpr auto find_vertex(G& g, vertex_key_t<G> const&) noexcept
      -> vertex_iterator_t<G>;
template <typename G>
```

```
constexpr auto find_vertex(G const& g, vertex_key_t<G> const&) noexcept
      -> const_vertex_iterator_t<G>;

template <typename G>
void reserve_vertices(G& g, vertex_size_t<G>) {}

template <typename G>
void resize_vertices(G& g, vertex_size_t<G>) {}


template <typename G>
constexpr auto edges(G& g) noexcept -> edge_range_t<G>;
template <typename G>
constexpr auto edges(G const& g) noexcept -> const_edge_range_t<G>;

template <typename G>
constexpr auto edges_size(G const& g) noexcept -> edge_size_t<G>;

template <typename G>
constexpr auto find_edge(G& g, vertex_t<G>& u, vertex_t<G>& v) noexcept
      -> edge_iterator_t<G>;
template <typename G>
constexpr auto
find_edge(G const& g, vertex_t<G> const& u, vertex_t<G> const& v) noexcept
      -> const_edge_iterator_t<G>;

template <typename G>
constexpr auto find_edge(G&               g,
                         vertex_key_t<G>& ukey,
                         vertex_key_t<G>& vkey) noexcept -> edge_iterator_t<G>;
template <typename G>
constexpr auto find_edge(G const&               g,
                         vertex_key_t<G> const& ukey,
                         vertex_key_t<G> const& vkey) noexcept
      -> const_edge_iterator_t<G>;

template <typename G>
constexpr void erase_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G>
v);

template <typename G>
constexpr void erase_edge(G& g, vertex_key_t<G>& ukey, vertex_key_t<G>& vkey);

template <typename G>
constexpr void erase_edge(G& g, edge_iterator_t<G> uv);

template <typename G>
constexpr void erase_edges(G& g, edge_range_t<G>);
```

```
template <typename G>
void reserve_edges(G& g, edge_size_t<G>) {}

template <typename G>
void clear(G& g);
template <typename G>
void clear(G& g, graph_value_t<G>&&);
template <typename G>
void clear(G& g, graph_value_t<G> const&);

template <typename G>
constexpr void swap(G& a, G& b);
```

Vertex Functions

```
template <typename G>
constexpr auto vertex_key(G const&, vertex_t<G> const& u) noexcept
      -> vertex_key_t<G>;

template <typename G>
constexpr auto edges(G& g, vertex_t<G>& u) noexcept -> vertex_edge_range_t<G>;
template <typename G>
constexpr auto edges(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_edge_range_t<G>;

template <typename G>
constexpr auto begin(G& g, vertex_t<G>& u) noexcept ->
vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto begin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto cbegin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_edge_iterator_t<G>;

template <typename G>
constexpr auto end(G& g, vertex_t<G>& u) noexcept -> vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto end(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto cend(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_edge_iterator_t<G>;

template <typename G>
constexpr auto create_vertex(G& g) -> pair<vertex_iterator_t<G>, bool>;
template <typename G>
```

```
constexpr auto create_vertex(G& g, vertex_value_t<G> const&)
      -> pair<vertex_iterator_t<G>, bool>;
template <typename G>
constexpr auto create_vertex(G& g, vertex_value_t<G> &&)
      -> pair<vertex_iterator_t<G>, bool>;

template <typename G>
constexpr void erase_vertices(G& g, vertex_range_t<G>&);
template <typename G>
constexpr void erase_vertex(G& g, vertex_iterator_t<G>&);
template <typename G>
constexpr void erase_vertex(G& g, vertex_key_t<G>&);

template <typename G>
constexpr auto edges_size(G const& g, vertex_t<G> const& u) noexcept
      -> vertex_edge_size_t<G>;

template <typename G>
constexpr auto edges_degree(G const& g, vertex_t<G> const& u) noexcept
      -> vertex_edge_size_t<G>;

template <typename G>
constexpr auto find_vertex_edge(G& g, vertex_t<G>& u, vertex_t<G>& v) noexcept
      -> vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto
find_vertex_edge(G const& g, vertex_t<G> const& u, vertex_t<G> const& v)
noexcept
      -> const_vertex_edge_iterator_t<G>;

template <typename G>
constexpr auto
find_vertex_edge(G& g, vertex_key_t<G>& ukey, vertex_key_t<G>& vkey) noexcept
      -> vertex_edge_iterator_t<G>;
template <typename G>
constexpr auto find_vertex_edge(G const&               g,
                                vertex_key_t<G> const& ukey,
                                vertex_key_t<G> const& vkey) noexcept
      -> const_vertex_edge_iterator_t<G>;

template <typename G>
constexpr auto erase_edge(G& g, vertex_edge_iterator_t<G> uv)
      -> vertex_edge_iterator_t<G>;

template <typename G>
constexpr auto erase_edges(G& g, vertex_edge_range_t<G> uv)
      -> vertex_edge_iterator_t<G>;
```

```
template <typename G>
constexpr void clear_edges(G& g, vertex_t<G>&);
```

Edge Functions

```
template <typename G>
constexpr auto vertex(G& g, edge_t<G>& uv) noexcept -> vertex_iterator_t<G>&;
template <typename G>
constexpr auto vertex(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_t<G> const&;
template <typename G>
constexpr auto vertex_key(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_key_t<G>;

template <typename G>
constexpr auto vertex(G& g, edge_t<G>& uv, vertex_t<G> const& source) noexcept
      -> vertex_iterator_t<G>&;
template <typename G>
constexpr auto vertex(G const&           g,
                      edge_t<G> const&   uv,
                      vertex_t<G> const& source) noexcept -> vertex_t<G>
const&;
template <typename G>
constexpr auto vertex_key(G const&              g,
                          edge_t<G> const&      uv,
                          vertex_key_t<G> const source_key) noexcept
      -> vertex_key_t<G>;

template <typename G>
constexpr auto out_vertex(G& g, edge_t<G>& uv) noexcept ->
vertex_iterator_t<G>&;
template <typename G>
constexpr auto out_vertex(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_iterator_t<G> const&;
template <typename G>
constexpr auto out_vertex_key(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_key_t<G>;

template <typename G>
constexpr auto in_vertex(G& g, edge_t<G>& uv) noexcept ->
vertex_iterator_t<G>&;
template <typename G>
constexpr auto in_vertex(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_iterator_t<G> const&;
template <typename G>
constexpr auto in_vertex_key(G const& g, edge_t<G> const& uv) noexcept
      -> vertex_key_t<G>;
```

```
template <typename G>
constexpr auto create_edge(G& g, vertex_t<G>& u, vertex_t<G>& v)
      -> pair<vertex_edge_iterator_t<G>, bool>;
template <typename G>
constexpr auto create_edge(G& g, vertex_t<G>& u, vertex_t<G>& v,
edge_value_t<G>&)
      -> pair<vertex_edge_iterator_t<G>, bool>;
template <typename G>
constexpr auto
create_edge(G& g, vertex_t<G>& u, vertex_t<G>& v, edge_value_t<G> &&)
      -> pair<vertex_edge_iterator_t<G>, bool>;

template <typename G>
constexpr auto create_edge(G& g, vertex_key_t<G>&, vertex_key_t<G>&)
      -> pair<vertex_edge_iterator_t<G>, bool>;
template <typename G>
constexpr auto
create_edge(G& g, vertex_key_t<G>&, vertex_key_t<G>&, edge_value_t<G>&)
      -> pair<vertex_edge_iterator_t<G>, bool>;
template <typename G>
constexpr auto
create_edge(G& g, vertex_key_t<G>&, vertex_key_t<G>&, edge_value_t<G> &&)
      -> pair<vertex_edge_iterator_t<G>, bool>;
```

Directed API

The Directed types and functions extend the Uniform API for directed graphs. They will never exist for undirected graphs. A directed graph will have outgoing edges and matching types and functions, and a bidirectional graph will extend that to include the incoming types and functions.

It is also certainly valid for a graph to have only incoming edges but isn't addressed outside of this statement. All functions are oriented toward undirected, directed, or bidirectional graphs and this paper addresses those common use cases.

Outgoing Types

```
template <typename G>
using vertex_out_edge_range_t = typename G::vertex_out_edge_range;
template <typename G>
using const_vertex_out_edge_range_t = typename G::const_vertex_out_edge_range;

template <typename G>
using vertex_out_edge_iterator_t = typename G::vertex_out_edge_iterator;
template <typename G>
using const_vertex_out_edge_iterator_t =
      typename G::const_vertex_out_edge_iterator;
```

```
template <typename G>
using vertex_out_edge_sentinel_t = typename G::vertex_out_edge_sentinel;

template <typename G>
using vertex_out_edge_size_t = typename G::vertex_out_edge_size_type;
```

Outgoing Functions

Vertex Functions

```
// Directed API (outgoing): Vertex functions
template <typename G>
constexpr auto out_edges(G& g, vertex_t<G>& u) noexcept
      -> vertex_out_edge_range_t<G>;
template <typename G>
constexpr auto out_edges(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_out_edge_range_t<G>;

template <typename G>
constexpr auto out_begin(G& g, vertex_t<G>& u) noexcept
      -> vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto out_begin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto out_cbegin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_out_edge_iterator_t<G>;

template <typename G>
constexpr auto out_end(G& g, vertex_t<G>& u) noexcept
      -> vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto out_end(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto out_cend(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_out_edge_iterator_t<G>;

template <typename G>
constexpr auto out_size(G const& g, vertex_t<G> const& u) noexcept
      -> vertex_out_edge_size_t<G>;
template <typename G>
constexpr auto out_degree(G const& g, vertex_t<G> const& u) noexcept
      -> vertex_out_edge_size_t<G>;

template <typename G>
```

```
constexpr auto find_out_edge(G& g, vertex_t<G>& u, vertex_t<G>& v) noexcept
      -> vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto
find_out_edge(G const& g, vertex_t<G> const& u, vertex_t<G> const& v) noexcept
      -> const_vertex_out_edge_iterator_t<G>;

template <typename G>
constexpr auto find_out_edge(G&                      g,
                             vertex_key_t<G> const& ukey,
                             vertex_key_t<G> const& vkey) noexcept
      -> vertex_out_edge_iterator_t<G>;
template <typename G>
constexpr auto find_out_edge(G const&                g,
                             vertex_key_t<G> const& ukey,
                             vertex_key_t<G> const& vkey) noexcept
      -> const_vertex_out_edge_iterator_t<G>;

template <typename G>
constexpr void erase_edges(G& g, vertex_out_edge_range_t<G>);

template <typename G>
constexpr void clear_out_edges(G& g, vertex_t<G>& u);
```

Edge Functions

```
template <typename G>
constexpr auto erase_edge(G& g, vertex_out_edge_iterator_t<G> uv)
      -> vertex_out_edge_iterator_t<G>;
```

Incoming Types

```
template <typename G>
using vertex_in_edge_range_t = typename G::vertex_in_edge_range;
template <typename G>
using const_vertex_in_edge_range_t = typename G::const_vertex_in_edge_range;

template <typename G>
using vertex_in_edge_iterator_t = typename G::vertex_in_edge_iterator;
template <typename G>
using const_vertex_in_edge_iterator_t = typename
G::const_vertex_in_edge_iterator;

template <typename G>
using vertex_in_edge_sentinel_t = typename G::vertex_in_edge_sentinel;

template <typename G>
using vertex_in_edge_size_t = typename G::vertex_in_edge_size_type;
```

Incoming Functions

Vertex Functions

```
template <typename G>
constexpr auto in_edges(G& g, vertex_t<G>& u) noexcept
      -> vertex_in_edge_range_t<G>;
template <typename G>
constexpr auto in_edges(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_in_edge_range_t<G>;

template <typename G>
constexpr auto in_begin(G& g, vertex_t<G>& u) noexcept
      -> vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto in_begin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto in_cbegin(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_in_edge_iterator_t<G>;

template <typename G>
constexpr auto in_end(G& g, vertex_t<G>& u) noexcept
      -> vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto in_end(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto in_cend(G const& g, vertex_t<G> const& u) noexcept
      -> const_vertex_in_edge_iterator_t<G>;

template <typename G>
constexpr auto in_size(G& g, vertex_t<G>& u) noexcept ->
vertex_in_edge_size_t<G>;
template <typename G>
constexpr auto in_degree(G& g, vertex_t<G>& u) noexcept
      -> vertex_in_edge_size_t<G>;

template <typename G>
constexpr auto find_in_edge(G& g, vertex_t<G>& u, vertex_t<G>& v) noexcept
      -> vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto
find_in_edge(G const& g, vertex_t<G> const& u, vertex_t<G> const& v) noexcept
      -> const_vertex_in_edge_iterator_t<G>;
```

```
template <typename G>
constexpr auto find_in_edge(G&                      g,
                            vertex_key_t<G> const& ukey,
                            vertex_key_t<G> const& vkey) noexcept
      -> vertex_in_edge_iterator_t<G>;
template <typename G>
constexpr auto find_in_edge(G const&                g,
                            vertex_key_t<G> const& ukey,
                            vertex_key_t<G> const& vkey) noexcept
      -> const_vertex_in_edge_iterator_t<G>;

template <typename G>
constexpr void erase_edges(G& g, vertex_in_edge_range_t<G>);

template <typename G>
constexpr void clear_in_edges(G& g, vertex_t<G>& u);
```

Edge Functions

```
template <typename G>
constexpr auto erase_edge(G& g, vertex_in_edge_iterator_t<G> uv)
      -> vertex_in_edge_iterator_t<G>;
```

Concept Definitions

The Concept definitions in this section are useful for graph algorithms to express the expectations of the algorithm's parameters. This is a work in progress.

All concepts defined in this paper are distinguished with a "_c" suffix.

```
template <typename G>
concept uniform_graph_c = requires(G&& g, vertex_t<G>& u) {
  { edges(g, u) } -> vertex_edge_range_t<G>;
  { edges(g) } -> edge_range_t<G>;
};

template <typename G>
concept out_directed_graph_c = requires(G&& g, vertex_t<G>& u) {
  { out_edges<G>(g, u) } -> vertex_out_edge_range_t<G>;
};

template <typename G>
concept in_directed_graph_c = requires(G&& g, vertex_t<G>& u) {
  { in_edges(g, u) } -> vertex_in_edge_range_t<G>;
};

template <typename G>
```

```
concept directed_graph_c = uniform_graph_c<G> && out_directed_graph<G>;

template <typename G>
concept bidirected_graph_c =
      uniform_graph_c<G> && out_directed_graph<G> && in_directed_graph<G>;

template <typename G>
concept undirected_graph_c =
      uniform_graph_c && !out_directed_graph_c<G> && !in_directed_graph_c<G>;

template<typename T> concept arithmetic_c requires is_arithmetic_v<T>;

template <typename G>
concept searchable_graph_c = uniform_graph_c<G> &&
  requires(G&& g, vertex_iterator_t<G> u, vertex_edge_iterator_t<G> uv) {
    ranges::forward_range<G>;
    ranges::forward_iterator<vertex_iterator_t<G>>;
    ranges::forward_iterator<vertex_edge_iterator_t<G>>;
    //ranges::forward_range<vertex_t<G>>;
    // vertex begin/end require graph parameter so it doesn't apply
    integral<vertex_key_t<G>>;
    { vertices(g) } ->vertex_range_t<G>;
    { begin(g, *u) } ->vertex_edge_iterator_t<G>;
    { end(g, *u) } ->vertex_edge_iterator_t<G>;
    { vertex(g, *uv) } ->vertex_iterator_t<G>;
    { vertex_key(g,*u) } ->vertex_key_t<G>;
  };
```

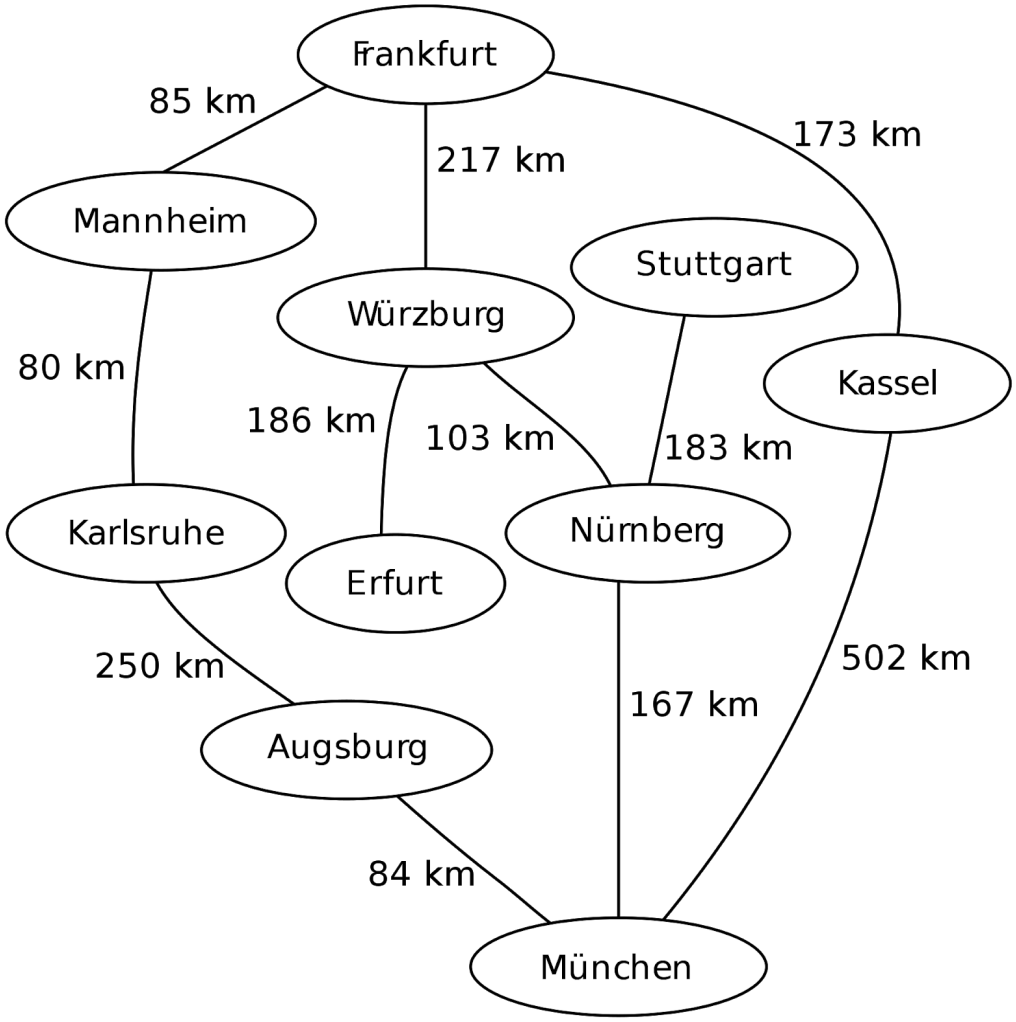
Ranges

Example Graph

```
 struct route {
    string from;
    string to;
    int    km = 0;

    route(string const& from_city, string const& to_city, int kilometers)
          : from(from_city), to(to_city), km(kilometers) {}
  };
  vector<route> routes{
        {"Frankfürt", "Mannheim", 85},  {"Frankfürt", "Würzburg", 217},
        {"Frankfürt", "Kassel", 173},   {"Mannheim", "Karlsruhe", 80},
        {"Karlsruhe", "Augsburg", 250}, {"Augsburg", "München", 84},
        {"Würzburg", "Erfurt", 186},    {"Würzburg", "Nürnberg", 103},
        {"Nürnberg", "Stuttgart", 183}, {"Nürnberg", "München", 167},
        {"Kassel", "München", 502}};
```

```
using G1 = adjacency_list<name_value, weight_value, empty_value,
                          edge_type_undirected, edge_link_double,
                          map_vertex_set_proxy>;
auto g1  = create_adjacency_list<G1>();
for (auto& r : routes)
  create_edge(g, r.from, r.to, r.km);
```



(Diagram and example from the Wikipedia article on Breadth-first search)

Depth First Search (DFS)

The algorithms are designed to work with both directed & undirected graphs by using general functions such as vertex() and edges() instead of out_vertex() and out_edges().

The ranges have the following assumptions and behavior

1. The graph structure is assumed to remain stable during the duration of iteration.
2. The depth is a 1-based integer. A value of 0 indicates there is nothing visited.

3. The state of the traversal is in the range object. Calling begin() returns the current state, not the beginning of the range.

dfs_vertex_range

```
template <searchable_graph_c G, typename A = allocator<char>>
requires uniform_graph_c<G>
      && integral<vertex_key_t<G>>
      && contiguous_range<vertex_range_t<G>>
class dfs_vertex_range {
public:
  typename const_iterator; // forward_iterator
  typename iterator;       // forward_iterator

  dfs_vertex_range(G& graph, vertex_iterator_t<G> seed, A alloc=A());

  iterator       begin();
  const_iterator begin() const;
  const_iterator cbegin() const;

  iterator       end();
  const_iterator end() const;
  const_iterator cend() const;
};
```

Note: the iterator classes have the following member function(s):

- depth() -> int that returns the distance from the seed vertex

Example

```
 dfs_vertex_range rng(g, g.find_vertex("Frankfürt"));
 for (vertex_iterator_t<G> u = rng.begin(); u != rng.end(); ++u)
    cout << string(u.depth() * 2, ' ') << u->name << endl;
  /* Output
      Frankfürt
        Mannheim
          Karlsruhe
            Augsburg
              München
        Würzburg
          Erfurt
          Nürnberg
            Stuttgart
        Kassel
```

```
  */
```

dfs_edge_range

```
template <searchable_graph_c G, typename A = allocator<char>>
requires uniform_graph_c<G>
      && integral<vertex_key_t<G>>
      && contiguous_range<vertex_range_t<G>>
class dfs_edge_range {
public:
  typename const_iterator; // forward_iterator
  typename iterator;       // forward_iterator

  dfs_edge_range(G& graph, vertex_iterator_t<G> seed, A alloc=A());

  iterator       begin();
  const_iterator begin() const;
  const_iterator cbegin() const;

  iterator       end();
  const_iterator end() const;
  const_iterator cend() const;
};
```

Note: the iterator classes have the following member function(s):

- depth() -> int that returns the distance from the seed vertex
- is_back_edge() -> bool that returns true if there is no out edge for a vertex.

Example

```
  dfs_edge_range rng(g, begin(g) + 2); // Frankfürt
  for (vertex_edge_iterator_t<G> uv = rng.begin(); uv != rng.end(); ++uv) {
    vertex_iterator_t<G> u     = out_vertex(g, *uv);
    vertex_key_t<G>      u_key = vertex_key(g, *u);
    if (uv.is_back_edge()) {
     cout << string(uv.depth() * 2, ' ')
          << "view " << uv.back_vertex()->name << endl;
    } else {
      vtx_iter_t v = out_vertex(g, *uv); // or vertex(g, *uv)
      cout << string(uv.depth() * 2, ' ') << "travel " << u->name
           << " --> " << v->name << " " << uv->weight << "km" << endl;
    }
  }
```

```
/* Output
    travel Frankfürt --> Mannheim 85km
      travel Mannheim --> Karlsruhe 80km
      travel Karlsruhe --> Augsburg 250km
      travel Augsburg --> München 84km
      view München
    travel Frankfürt --> Würzburg 217km
      travel Würzburg --> Erfurt 186km
        view Erfurt
      travel Würzburg --> Nürnberg 103km
      travel Nürnberg --> Stuttgart 183km
        view Stuttgart
      travel Nürnberg --> München 167km
    travel Frankfürt --> Kassel 173km
    travel Kassel --> München 502km
*/
```

Breadth First Search (BFS)

The algorithms are designed to work with both directed & undirected graphs by using general functions such as vertex() and edges() instead of out_vertex() and out_edges().

The ranges have the following assumptions and behavior

4. The graph structure is assumed to remain stable during the duration of iteration.
5. The depth is a 1-based integer. A value of 0 indicates there is nothing visited.
6. The state of the traversal is in the range object. Calling begin() returns the current state, not the beginning of the range.

bfs_vertex_range

```
template <searchable_graph_c G, typename A = allocator<char>>
requires integral<vertex_key_t<G>>
      && contiguous_range<vertex_range_t<G>>
class bfs_vertex_range {
  typename const_iterator; // forward_iterator
  typename iterator;       // forward_iterator

  bfs_vertex_range(G& graph, vertex_iterator_t<G> seed, A alloc=A());

  iterator       begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
```

```
  iterator       end();
  const_iterator end() const;
  const_iterator cend() const;
};
```

Note: the iterator classes have the following member function(s):

- depth() -> int that returns the distance from the seed vertex
- is_back_edge() -> bool that returns true if there is no out edge for a vertex.

Example

```
  bfs_vertex_range bfs_vtx_rng(g, begin(g) + 2); // Frankfürt
  for (auto u = bfs_vtx_rng.begin(); u != bfs_vtx_rng.end(); ++u)
    cout << string(u.depth() * 2, ' ') << u->name << endl;

  /* Output
      Frankfürt
        Mannheim
        Würzburg
        Kassel
          Karlsruhe
          Erfurt
          Nürnberg
          München
            Augsburg
            Stuttgart
  */
```

bfs_edge_range

```
template <searchable_graph_c G, typename A = allocator<char>>
requires integral<vertex_key_t<G>>
      && contiguous_range<vertex_range_t<G>>
class bfs_edge_range {
  typename const_iterator; // forward_iterator
  typename iterator;       // forward_iterator

  bfs_edge_range(G& graph, vertex_iterator_t<G> seed, A alloc=A());

  iterator       begin();
  const_iterator begin() const;
  const_iterator cbegin() const;
```

```
  iterator       end();
  const_iterator end() const;
  const_iterator cend() const;
};
```

Note: the iterator classes have the following member function(s):

- depth() -> int that returns the distance from the seed vertex

Example

```
  bfs_edge_range rng(g, begin(g) + 2); // Frankfürt
  for (auto uv = rng.begin(); uv != rng.end(); ++uv) {
    vertex_iterator_t<Graph> u     = in_vertex(g, *uv);
    vertex_key_t<Graph>      u_key = vertex_key(g, *u);
    if (uv.is_back_edge()) {
      cout << string(uv.depth() * 2, ' ')
           << "view " << uv.back_vertex()->name << endl;
    } else {
      vtx_iter_t v = vertex(g, *uv);
      cout << string(uv.depth() * 2, ' ') << "travel " << u->name
           << " --> " << v->name << " " << uv->weight << "km" << endl;
    }
  }

  /* Output
      travel Frankfürt --> Mannheim 85km
      travel Frankfürt --> Würzburg 217km
      travel Frankfürt --> Kassel 173km
        travel Mannheim --> Karlsruhe 80km
        travel Würzburg --> Erfurt 186km
        travel Würzburg --> Nürnberg 103km
        travel Kassel --> München 502km
          travel Karlsruhe --> Augsburg 250km
          view Erfurt
          travel Nürnberg --> Stuttgart 183km
          travel Nürnberg --> München 167km
          view München
            travel Augsburg --> München 84km
            view Stuttgart
  */
```

Topological Sort (TopoSort)

```
template<graph_c G>
class topological_sort_iterator;  // forward iterator

template<graph_c G>
class topological_sort_range;  // forward range

template<graph_c G, vertex_iterator_c I>
auto topological_sort(vertex_iterator_c) -> topological_sort_range<G>;
```

Algorithms

Algorithms deliver the value of graphs and are the primary focus. They follow the established STL design of working with iterators independent of the graph container.

The algorithms have been selected for a balance of their usefulness without being overly complex for their implementation for the initial library. Other algorithms have also been identified for the future in the Additional Algorithms section.

Not all algorithms have known parallel implementations. When a parallel implementation is known it will be identified and enabled for an algorithm.

Shortest Paths Algorithms

Shortest paths algorithms find the distance of the shortest path between vertices and return the results to an output iterator. If no out edges exist on a vertex then no paths exist. Each algorithm is distinguished by whether it supports negative weights or not.

A shortest path is defined by the distance (sum of each edge distance, as evaluated by the passed distance function) and the vertices and/or edges that make up the shortest path.

```
template <typename G, typename A = allocator<vertex_iterator_t<G>>>
using vertex_path_t = vector<vertex_iterator_t<G>, A>;

template <typename G, typename A = allocator<edge_iterator_t<G>>>
using vertex_path_range = decltype(
      make_subrange(*reinterpret_cast<vertex_path_t<G, A>*>(nullptr)));

template <typename G, typename A = allocator<edge_iterator_t<G>>>
using edge_path_t = vector<edge_iterator_t<G>, A>;

template <typename G, typename A>
using edge_path_range = decltype(
```

```
      make_subrange(*reinterpret_cast<edge_path_t<G, A>*>(nullptr)));

template <typename G,
          arithmetic_c DistanceT,
          typename A = allocator<vertex_iterator_t<G>>>
struct shortest_path {
  DistanceT               distance;
  vertex_path_range<G, A> vertex_path;
  edge_path_range<G, A>   edge_path;
};
```

The allocator type used for shortest_path should be the same type used in the shortest_path functions.

Shortest Paths - Example 1

This example shows paths to all cities reachable from Frankfürt by using parameter leaves_only=false.

This shows the Bellman-Ford algorithm, but Dikjstra can also be used in this case because the distances between cities are all non-negative, a requirement for Dijkstra. Parameter detect_neg_edge_cycles=true to cause detection of negative cycles, and the result is returned by the function.

```
Graph                    g = create_germany_routes_graph();
vertex_iterator_t<Graph> u = find_if(
    g, [](vertex_t<Graph>& u) { return u.name == "Frankfürt"; });

auto weight_fnc = [](edge_value_t<Graph>& uv) -> int { return uv.weight; };

bool neg_edge_cycle_exists = bellman_ford_shortest_paths<int>(
      g, u, back_inserter(short_paths), false, true, weight_fnc);

for (auto& sp : short_paths) { // shortest_path<G,DistanceT,A>
  for (size_t i = 0; i < sp.vertex_path.size(); ++i) {
    if (i > 0)
      cout << " --> ";
    cout << sp.vertex_path[i]->name;
  }
  cout << "  " << sp.distance << "km\n";
}
/* Output: source = Frankfürt
    Frankfürt --> Mannheim --> Karlsruhe --> Augsburg  415km
    Frankfürt --> Würzburg --> Erfurt  403km
    Frankfürt  0km
    Frankfürt --> Mannheim --> Karlsruhe  165km
    Frankfürt --> Kassel  173km
```

```
    Frankfürt --> Mannheim  85km
    Frankfürt --> Würzburg --> Nürnberg --> München  487km
    Frankfürt --> Würzburg --> Nürnberg  320km
    Frankfürt --> Würzburg --> Nürnberg --> Stuttgart  503km
    Frankfürt --> Würzburg  217km
*/
```

Shortest Paths - Example 2

As in Distances - Example 2, parameter leaves_only=true only outputs the paths and their distances to final vertices that end a path.

```
short_paths.clear();
bool neg_edge_cycle_exists = bellman_ford_shortest_paths<int>(
    g, u, back_inserter(short_paths), true, true, weight_fnc);
for (auto& sp : short_paths) { // shortest_path<G,DistanceT,A>
  for (size_t i = 0; i < sp.path.size(); ++i) {
      if (i > 0)
        cout << " --> ";
      cout << sp.vertex_path[i]->name;
  }
}
cout << "  " << sp.distance << "km\n";
}
/* Output: source = Frankfürt
    Frankfürt --> Würzburg --> Erfurt  403km
    Frankfürt --> Würzburg --> Nürnberg --> München  487km
    Frankfürt --> Würzburg --> Nürnberg --> Stuttgart  503km
*/
```

Dijkstra Algorithms

Dijkstra's algorithm is the fastest of the shortest path/distance algorithms with O(|E| + |V|log|V|), but has a requirement that the edge distances (aka weights) are non-negative. There is no additional time penalty for parameter leaves_only=true like there is for Bellman-Ford. Directed and undirected graphs are both supported.

Signed distance types are accepted for DistanceT to allow the algorithms to accommodate real-world situations, like the use of floating point values. It's the caller's responsibility to assure their distances are non-negative.

The allocator parameter used for allocation of internal data structures.

```
template <arithmetic_c DistanceT,
          typename G,
          typename OutIter,
          typename DistFnc,
```

```
          typename A = allocator<char>>
requires integral<vertex_key_t<G>>
    && contiguous_range<vertex_range_t<G>>
    && output_iterator<OutIter, shortest_path<G,DistanceT,A>>
void dijkstra_shortest_paths(
      G&                   g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only  = true,
      DistFnc              distance_fnc = [](edge_value_t<G>&) -> size_t
                                          { return 1; },
      A                    alloc        = A());
```

Bellman-Ford Algorithms

The Bellman-Ford algorithm accepts negative edge distances and performs in O(|V| * |E|). Additional time of O(|V| + |E|) is taken when parameter leaves_only=true, and O(|E|) when parameter detect_neg_edge_cycles=true. Directed and undirected graphs are both supported.

```
template <arithmetic_c DistanceT,
          typename G,
          typename OutIter,
          typename DistFnc,
          typename A = allocator<DistanceT>>
requires integral<vertex_key_t<G>>
      && contiguous_range<vertex_range_t<G>>
      && output_iterator<OutIter, shortest_path<G, DistanceT, A>>
bool bellman_ford_shortest_paths(
      G&                   g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only            = true,
      bool const           detect_neg_edge_cycles = true,
      DistFnc distance_fnc = [](edge_value_t<G>&) -> size_t { return 1; },
      A       alloc        = A());
```

Connected Components

A connected component is a collection of all vertices that are joined by edges in an undirected graph.

```
template<Iterator vtx_iter_t, integral Comp = size_t>
struct component {
  Comp       component_number;
  vtx_iter_t vertex;
};
```

```
template<
  undirected_graph_c  G,
  OutputIterator<component<Comp, vertex_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void connected_components(G&                    g,
                          vertex_iterator_t<G> start,
                          OutIter              result_iter);

template<
  undirected_graph_c  G,
  OutputIterator<component<Comp, vertex_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void connected_components(
       G&                g,
       vertex_range_t<G> start,
       OutIter           result_iter);
```

Strongly Connected Components

A strongly connected component is a collection of all vertices that are joined by directed edges in a directed graph.

```
template<
  directed_graph_c  G,
  OutputIterator<component<Comp, vertex_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void strongly_connected_components(
       G&                   g,
       vertex_iterator_t<G> start,
       OutIter              result_iter);

template<
  directed_graph_c  G,  // directed
  OutputIterator<component<Comp, vertex_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void strongly_connected_components(
       G&                g,
       vertex_range_t<G> start,
       OutIter           result_iter);
```

Biconnected Components

A biconnected component is the set of connected vertices that, when one is removed, the component remains connected. In other words, the vertices in the component are not articulation points.

```
template<Iterator vtx_iter_t, integral Comp = size_t>
struct bicomponent {
  Comp       component;
  vtx_iter_t vertex;
};

template<
  undirected_graph_c  G,
  OutputIterator<bicomponent<Comp, edge_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void biconnected_components(
       G&                   g,
       vertex_iterator_t<G> start,
       OutIter              result_iter);

template<
  undirected_graph_c  G,
  OutputIterator<bicomponent<Comp, edge_iterator_t<G>>> OutIter,
  integral Comp = size_t
>
void biconnected_components(
       G&                 g,
       vertex_range_t<G> start,
       OutIter            result_iter);
```

Articulation Points

An articulation point is a vertex that, when removed, would split a graph into two or more disconnected components.

```
template<
  undirected_graph_c                   G,
  OutputIterator<vertex_iterator_t<G>> OutIter,
  integral                             Comp = size_t
>
void articulation_points(
       G&                   g,
       vertex_iterator_t<G> start,
       OutIter              result_iter);
```

```
template<
  undirected_graph_c                    G,
  OutputIterator<vertex_iterator_t<G>> OutIter,
  integral                              Comp = size_t
>
void articulation_points(
       G&                g,
       vertex_range_t<G> start,
       OutIter           result_iter);
```

Transitive Closure

The transitive closure discovers the reachability of each vertex to other vertices in a graph. Specialized algorithms are provided for sparse & dense graphs. Both assign reaches<vertex_iterator_t<G>> values to the output iterator passed.

```
template<directed_graph_c G>
struct reaches {
  vertex_iterator_t<G> from;
  vertex_iterator_t<G> to;
};
```

The sparse graph version evaluates a DFS for all vertices in the graph. It is sensitive to the number of edges, with complexity of V(VE).

```
template <directed_graph_c G, typename OutIter, typename A = allocator<char>>
  requires integral<vertex_key_t<G>>
        && contiguous_range<vertex_range_t<G>>
constexpr void dfs_transitive_closure(G& g,
                                      OutIter result_iter,
                                      A alloc = A())
```

The dense graph version uses Warshall's algorithm with $O(V^3)$ complexity.

```
template <directed_graph_c G, typename OutIter, typename A = allocator<bool>>
  requires integral<vertex_key_t<G>>
        && contiguous_range<vertex_range_t<G>>
constexpr void warshall_transitive_closure(G& g,
                                           OutIter result_iter,
                                           A alloc = A());
```

Example

To Do: create a new graph, transform a graph

erase & erase_if

We also propose the addition of **non-member functions** `erase` and `erase_if` to **remove** specified **vertices** and **edges**, that is, **uniform container erasure**.

Uniform container erasure is not supported because the graph is needed for all erase functions.

Additional Algorithms

The following algorithms have been identified for consideration in an additional paper(s):

1. Minimum spanning tree
2. Maximum flow
3. Matching
4. Bipartite matching
5. Min-cost network flow
6. Isomorphism
7. Subgraph isomorphism
8. Centrality
9. Minimum cut
10. Cycle detection
11. Path enumeration
12. Community detection
13. Clique enumeration
14. Find triangles
15. Lowest common ancestor
16. Dominator algorithms

Graph Data Structures

[This section is rough and is likely to change significantly in future papers. The goal is to provide useful implementations of directed and undirected graphs that can be used for general purpose problems, and as examples of how to define the API defined in this paper to a graph implementation.]

A graph is defined by its data structures and the public API used to access and manipulate it, as defined by the Uniform API and (optional) Directed API defined earlier in this document. The use of classes and other functions to implement the graph outside of those types and functions is considered internal, the one exception being construction of the graph. The graph constructors will be defined by each implementation according to its capabilities and constraints that best suit its needs.

This proposal recognizes common capabilities and representations of graphs and provide the user the ability to select all reasonable combinations that do not conflict. It also enables the user to extend the vertex, edge and graph implementations beyond those provided. For instance, the user can store vertices in a different container than those supplied by the standard by defining their own vertex set.

Attention should be given to keeping object sizes to the minimum needed to provide the required functionality. For instance, edges in an adjacency matrix should only be as big as the user-defined edge value, and for an adjacency array should be the size of user-defined edge value and in and out vertex references (vertex_id or pointer).

A common interface between different graphs is also a priority whenever possible, allowing for easier learning and transitioning between different characteristics of graphs.

Graph Template Parameters

All adjacency types are defined as a templatized graph class used to define and customize the graph.

The parameters shown here and in the adjacency template definitions should be considered proof-of-concept. They may vary slightly as refinements are made in future papers.

Parameter	Valid Values	Description
GV	(user-defined)	The graph value type defined by the user. It can be most valid C++ value type including class, struct, tuple, union, enum, array, reference or scalar value. If no value is needed then the empty_value struct can be used. See the User Values section for more information.
VV	(user-defined)	The vertex value type. (See GV.)
VSP	vector_vertex_set_proxy deque_vertex_set_proxy ordered_map_vertex_set_proxy unordered_map_vertex_set_proxy (user-defined)	The vertex set proxy used to define the container used to store vertices. The user can define their own vertex set as long as they support the common interface.
EV	(user-defined)	The edge value type. (See GV.)

EDIR	edge_type_directed_fwddir edge_type_directed_bidir edge_type_undirected	Edge directionality. fwddir supports directed outgoing edges, bidir supports incoming and outgoing edges, and undirected supports undirected edges. Bidir is a superset of fwddir. This has the biggest impact on the interface available.
ELNK	edge_link_double edge_link_single edge_link_none	Use doubly- or singly-linked lists for edges. This only applies when linked lists are used.
IndexT	Signed or unsigned integer (default is uint32_t)	For graphs that store vertices in vectors, this type is used for a vertex's index.
A	allocator<char>	A standard C++ allocator. Rebind is used to redefine for both vertex and edge types.

Graph Types

adjacency_list

An adjacency_list is the most compact data structure for sparse graphs. Edge instances are stored in the outgoing edges of a vertex. When in-edges are included, they are intrusive containers embedded in the outgoing edges and are limited to a linked-list node-based container.

```
  template <class VV   = empty_value,
            class EV   = empty_value,
            class GV   = empty_value,
            class EDIR = edge_type_directed_fwddir,
            class ELNK = edge_link_single,
            class VSP  = vector_vertex_set_proxy,
            class A      = allocator<char>>
  using adjaceny_list;
```

adjacency_array

An adjacency_array is defined by edges stored in a single container. Use of contiguous or semi-contiguous containers such as vector and deque will favor edge-oriented algorithms. Out and In edges of vertices will be additional containers that refer to the edges.

```
template <class VV   = empty_value,
          class EV   = empty_value,
```

```
          class GV   = empty_value,
          class EDIR = edge_type_directed_fwddir,
          class ELNK = edge_link_single,
          class VSP  = vector_vertex_set_proxy,
          class A    = allocator<char>>
using adjaceny_array;
```

adjacency_matrix

An adjacency_matrix is defined by edges stored in a 2-dimensional square array, where the size of the dimensions are the number vertices.

The number of vertices is passed during construction of the adjacency matrix when all vertices and edges are also constructed. Vertices and edges cannot be created or erased after the graph is constructed.

```
template <class VV  = empty_value,
          class EV  = empty_value,
          class GV  = empty_value,
          class VSP = vector_vertex_set_proxy,
          class A   = allocator<char>>
using adjaceny_matrix;
```

User Values

User-defined types can be used to define values for a vertex, edge and graph. Given the following definition:

```
struct name_value {
  string name;

  name_value()               = default;
  name_value(name_value const&) = default;
  name_value& operator=(name_value const&) = default;
  name_value(string const& s) : name(s) {}
  name_value(string&& s) : name(move(s)) {}
};

struct weight_value {
  int weight = 0;

  weight_value()                    = default;
  weight_value(weight_value const&) = default;
  weight_value& operator=(weight_value const&) = default;
  weight_value(int const& w) : weight(w) {}
```

```
};

using G = adjacency_list<name_value, weight_value>;
G g;
auto& [iter, successful] = g.create_vertex(name("a"));
auto& [uid, u] = *iter;
auto& [vid, v] = *g.create_vertex(name("b")).first;
auto& uv_iter  = g.create_edge(uid, vid, weight_value(42));
auto& uv       = *iter;
string nm = u.name;    // nm == "a"
int    w  = uv.weight; // w == 42
```

A class is also usable. There's no limit on the number of values in the struct used. The requirements are that it be default constructible, copy constructible and assignable. Move constructible is also supported.

Non-struct & non-class types can also be used, including scaler, array, union and enum. In those cases they are assigned the member name of "value" on the vertex.

```
using weight_t = int;
using G = adjacency_list<name_value, weight_t>;
(create vertices u & v as before)
auto& uv_iter  = g.create_edge(uid, vid, 42);
auto& uv       = *uv_iter;
int w          = u.value;        // w == 42
int w2         = u.user_value(); // w2 == 42
```

The reason for using "value" is that vertex inherits from it's property value and some types, like "int", are not a valid base class, nor are union, array, union or enum which all use "value". The benefit is that empty-base optimization is used when no value is needed.

The empty_value struct is used when no value is needed.

```
struct empty_value {};
```

Here is a simplified version of the vertex class to demonstrate how the value is defined as well as the graph_value_needs_wrap<> definition.

```
template <class ADJ, class VV, class VMEM, class EV, class EDIR, class ELNK,
class EMEM>
class vertex
      : public conditional_t<graph_value_needs_wrap<VV>::value,
graph_value<VV>, VV>
{ … }

template <class T>
struct graph_value_needs_wrap
      : integral_constant<bool,
                          is_scalar<T>::value || is_array<T>::value ||
```

```
                    is_union<T>::value || is_reference<T>::value> {};
```

The benefit of using inheritance is that no memory is used when empty_value is used because of the empty base optimization.

Design Notes

A class-based design was considered but was rejected. Assume all edges for the graph are stored in a single vector. A vertex would need to keep indexes into its outgoing edges (using an edge iterator would be unstable when edges are added). To get an iterator to an edge, the vertex would either need to store a pointer to the edge container, or the out edges container would have to include a parameter for the graph in the begin() and end() methods. Neither option is good. Using a pure function interface provides an abstraction that avoids this issue. The internal implementation can still use classes but the public interface will be free functions.

User-defined graph structures can be used by defining specializations of the graph functions for the user-defined graph, vertex and edge types.

Acknowledgements

This paper is the result of the discussions of SG19 Machine Learning.

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

References

1. Implementations
    a. The Boost Graph Library (BGL)
    b. JgraphT
    c. The Stanford cslib package
    d. dlib.net graph
2. Data sets
    a. Graph500
    b. GAP Benchmark Suite
3. Algorithm References
    a. Algorithms in C++ 3rd Edition, Part 5 Graph Algorithms by Robert Sedgewick

b. The Boost Graph Library User Guide and Reference, by Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine
c. wikipedia.org
d. Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest