P1709R0: Graph Data Structures

Date:	2019-06-17 (Pre-Cologne mailing): 10 AM ET
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21

Authors:	Phillip Ratzloff (SAS Institute) Richard Dosselmann (U of Regina) Michael Wong (Codeplay)
Contributors:	N/A
Emails:	phil.ratzloff@sas.com dosselmr@cs.uregina.ca michael@codeplay.com
Reply to:	phil.ratzloff@sas.com

Introduction

This document proposes the inclusion of a (general) **graph** data structure in the C++ **containers** library to support **machine learning** (ML), as well as all other applications. ML is a large and growing field, both in the **research community** and **industry**. **Artificial intelligence** (AI), a subset of ML, 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, be *directed* or *undirected*, respectively. Both **static** and **dynamic** implementations of a graph exist, using a (adjacency) **matrix,** (adjacency) **array** and (adjacency) **list**, each having the typical advantages and disadvantages associated with static and dynamic data structures. Given that a static implementation of a graph employs a matrix, a structure that is currently under development, this document proposes a **dynamic** implementation of a graph.

This paper starts the discussion on Graph Data structure with a proposed example interface. Though we might modify that interface significantly as we continue discussion.

Revision History

N/A

Motivation

A graph data structure, used in ML and other **scientific** domains, as well as **industry** 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 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

Background and Goals

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 Connected Components (undirected graphs) or Strongly Connected Components (directed graphs) algorithms.

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 possible.

This proposal recognizes all 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 is also 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. A noticeable difference in type and function names exists between directed and undirected graphs by design to reflect the different nature of the graphs.

Common algorithms will be provided, and it is expected more will be added over time. Features for common functionality will be provided, such as general breadth-first search (BFS), depth-first search (DFS), [and general visitor]. All algorithms should work with different forms of graphs, as described above, with little or no modification. Notable exceptions include attempting to use an algorithm that requires a directed graph on an undirected graph, or an algorithm that requires a

bi-directional algorithm on an undirected or forward-only graph. [These characteristics point toward Concepts that may be useful for graphs.]

Examples

This section introduces the capabilities of the graphs using a simple example to copy one graph to another, reversing the edges.

An adjacency list is defined where vertices have a name property and edges have a weight. A property can also be defined for the adjacency list. The default is forward-only and uses singly-linked lists to store the outgoing edges on each vertex, and uses a vector to store the vertices. These can all be overridden.

```
using name_t   = name_value; // struct name_value { string name; }
using weight_t = int;
using GL       = adjaceny_list<name_t, weight_t>;

std::vector<string> vertex_values = {"a", "b", "c", "d", "e", "f"};
std::vector<tuple<vertex_id, vertex_id, weight_t>> edge_values = {
    // {in vertex_id, out vertex_id, weight}
    {0, 1, 2}, {1, 2, 3}, {2, 3, 5}, {4, 3, 1}, {3, 5, 10}, {3, 0, 3},
    {0, 5, 1}, {5, 4, 1}, {1, 4, 2}, {4, 2, 3}, {5, 1, 4}};
```

Creating an adjacency list graph

```
GL g;
for (auto& label : vertex_values)
  g.create_vertex(label);
for (auto& uv : edge_values)
  g.create_edge(get<0>(uv), get<1>(uv), get<2>(uv));
```

Creating a new adjacency list graph with edge directions reversed. user_value() is name_t for vertices and weight_t for edges.

```
GL rev;
for (auto& [id, u] : g.vertices())
  rev.create_vertex(u.user_value());
for (auto& uv : g.edges())
  rev.create_edge(uv.out_vertex_id(), uv.in_vertex_id(), uv.user_value());
```

Reversing the edges for an adjacency array is a little different because outgoing edges for a vertex need to be added together. To accomplish that we need to be able to traverse the incoming edges to get the source vertices, requiring a new definition of the adjacency list.

```
using GL2 = adjaceny_list<name_t,       // vertex value type
                          weight_t,     // edge value type
                          empty_value,  // graph value
                          edge_type_directed_bidir, // bidirectional
                          vector_vertex_set_proxy>  // vertex in vectors
```

```
                                ;
```

We can now reverse the edges when copying to an adjacency array.

```
GL2 g;
//(same code to create the G2 graph)

using GA = adjaceny_array<name_t, weight_t>;
GA rev;
for (auto& [id, v] : g.vertices()) {
  rev.create_vertex(v.user_value());
  for (auto& uv : v.in_edges())
    rev.create_edge(uv.out_vertex_id(),uv.in_vertex_id(),
                    uv.user_value());
}
```

Reversing the edges on an adjacency matrix is a different concept because all edges exist between vertices. Creation of the adjacency matrix requires knowing the number of vertices when it is created, and all vertices and edges are created when the graph is created. Copying the graph is simply copying the vertex & edge values.

```
GM rev(g.vertices().size());
for (auto& [id, v] : g.vertices()) {
  auto& [rev_id, rev_v] = *rev.find(id);
  rev_v.user_value() = v.user_value();
}
for (auto& uv : g.edges()) {
  auto& rev_uv = *rev.find(uv.out_vertex_id(), uv.in_vertex_id());
  rev_uv.user_value() = uv.user_value();
}
```

All edges in rev that don't have a matching edge in g will have a default weight of 0.

Experimental Function Interface and Examples

This section is identical to the previous section except that it uses a free function interface instead of a object-oriented interface. This may be useful to support function composition that is coming with the Range v3 functionality in C++20. Only one of the interfaces should exist, not both.

This section introduces the capabilities of the graphs using a simple example to copy one graph to another, reversing the edges.

An adjacency list is defined where vertices have a name property and edges have a weight. A property can also be defined for the adjacency list. The default is forward-only and uses

singly-linked lists to store the outgoing edges on each vertex, and uses a vector to store the vertices. These can all be overridden.

```
using name_t   = std::string;
using weight_t = int;
using GL       = adjaceny_list<name_t, weight_t>;

std::vector<string> vertex_values = {"a", "b", "c", "d", "e", "f"};
std::vector<tuple<vertex_id, vertex_id, weight_t>> edge_values = {
    // {in vertex_id, out vertex_id, weight}
    {0, 1, 2}, {1, 2, 3}, {2, 3, 5}, {4, 3, 1}, {3, 5, 10}, {3, 0, 3},
    {0, 5, 1}, {5, 4, 1}, {1, 4, 2}, {4, 2, 3}, {5, 1, 4}};
```

Creating an adjacency list graph

```
GL g;
for (auto& label : vertex_values)
  create_vertex(g, label);
for (auto& uv : edge_values)
  create_edge(g, get<0>(uv), get<1>(uv), get<2>(uv));
```

Creating a new adjacency list graph with edge directions reversed. user_value() is name_t for vertices and weight_t for edges.

```
GL rev;
for (auto& [id, u] : vertices(g))
  create_vertex(rev, user_value(u));
for (auto& uv : edges(g))
  create_edge(rev, out_vertex_id(uv), in_vertex_id(uv), user_value(uv));
```

Reversing the edges for an adjacency array is a little different because outgoing edges for a vertex need to be added together. To accomplish that we need to be able to traverse the incoming edges to get the source vertices, requiring a new definition of the adjacency list.

```
using GL2 = adjaceny_list<name_t,       // vertex value type
                          weight_t,     // edge value type
                          empty_value,  // graph value
                          edge_type_directed_bidir, // bidirectional
                          vector_vertex_set_proxy>  // vertex in vectors
                          ;
```

We can now reverse the edges when copying to an adjacency array.

```
GL2 g;
//(same code to create the G2 graph)

using GA = adjaceny_array<name_t, weight_t>;
GA rev;
for (auto& [id, v] : vertices(g)) {
  create_vertex(rev, user_value(v));
  for (auto& uv : in_edges(v,g))
```

```
    create_edge(rev, out_vertex_id(uv),in_vertex_id(uv),
                     user_value(uv));
}
```

Reversing the edges on an adjacency matrix is a different concept because all edges exist between vertices. Creation of the adjacency matrix requires knowing the number of vertices when it is created, and all vertices and edges are created when the graph is created. Copying the graph is simply copying the vertex & edge values.

```
GM rev(vertices(g).size());
For (auto& [id, v] : vertices(g)) {
  auto& [rev_id, rev_v] = *find_vertex(rev, id);
  user_value(rev_v) = user_value(v);
}
for (auto& uv : edges(g)) {
  auto& rev_uv = *find_out_edge(rev, out_vertex_id(uv), in_vertex_id(uv));
  user_value(rev_uv) = user_value(uv);
}
```

All edges in rev that don't have a matching edge in g will have a default weight of 0.

Adjacency Types

The different types of adjacency graphs are determined by the physical organization of edges in memory. There is a tradeoff between adding and removing edges/vertices and performance when visiting the graph.

	Adjacency List	Adjacency Array	Adjacency Matrix
Out-edges storage	Singly- or doubly-linked list	Array for all edges (contiguous memory)	Array (contiguous memory)
In-edges storage (bi-directional only)	Single- or doubly-linked list	Single- or doubly-linked list	Array (noncontiguous memory)
Undirected edges storage	Single- or doubly-linked list	n/a	n/a
Add vertices	O(1)	O(1)	n/a
Remove vertices	O(1 + |e|) O(log V + |e|)	n/a	n/a

Add edges	O(1)	O(1) All out edges for a vertex must be added at once	O(1)
Remove edges	O(n) for singly linked O(1) for doubly linked	n/a	n/a

Incoming edges are always in non-contiguous memory. They are kept in a linked list for adjacency list and adjacency array.

Graph Parameters

All adjacency types are defined as a templatized graph class used to define and customize the graph. Their usage is clearer in the context of the adjacency types in the following sections. The graph prototype is defined as:

```
template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
          class A>
class graph;
```

Parameter	Valid Values	Description
ADJ	adj_list_type adj_array_type adj_matrix_type	The adjacency type.
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.
A	allocator<char>	A standard C++ allocator. Rebind is used to redefine for both vertex and edge types.

Additional template parameters are used internally elsewhere in other classes. Their definition is included here for completeness.

Parameter	Valid Values	Description
VMEM	mem_fixed mem_movable	Identifies whether vertices are fixed or movable in memory. Determines whether a pointer or identifier is used to reference a vertex.
EMEM	mem_fixed mem_movable	Identifies whether edges are fixed or movable in memory. Determines whether a pointer or identifier is used to reference an edge.

Adjacency List

An adjacency list is defined by edges stored in linked list.

```
  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 = graph<adj_list_type, GV, VV, VSP, EV, EDIR, ELNK, A>;
```

Adjacency Array

An adjacency array is defined by edges stored in a growable array (e.g. or std::vector).

```
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 = graph<adj_array_type, GV, VV, VSP, EV, EDIR, ELNK, A>;
```

Adjacency Matrix

An adjacency array 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. See the graph class section for adjacency matrix for more information.

```
  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 = graph<adj_matrix_type,
                                GV,
                                 VV,
                                 VSP,
                                 EV,
                                edge_type_directed_bidir,
                                 edge_link_none,
                                 A>;
```

User Values (a.k.a. Properties)

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.

Directed and Undirected

Forward-only, bidirectional and undirected characteristics of a graph have the biggest impact on the interface of graph, vertices and edges.

Forward-only graphs support outgoing edges, iterators and related functions. Bidirectional graphs extend forward-only graphs by adding functionality for incoming edges. Here is an example using a vertex.

```
template<…, edge_type_directed_fwddir, …>
class vertex {
  using out_iterator = …;
  using const_out_iterator = …;
  out_edge_list& out_edges();
  const_out_edge_list& out_edges();
};
```

```
template<…, edge_type_directed_bidir, …>
class vertex {
  using out_iterator = …;
  using const_out_iterator = …;
  out_edge_list& out_edges();
  const_out_edge_list& out_edges();

  using in_iterator = …;
  using const_in_iterator = …;
  in_edge_list& in_edges();
  const_in_edge_list& in_edges();
};
```

Undirected graphs have edges but without any directionality.

```
template<…, edge_type_undirected, …>
class vertex {
  using iterator = …;
  using const_iterator = …;
  edge_list& undir_edges();
  const_edge_list& undir_edges();
};
```

In an effort to keep complexity and confusion to a minimum, class definitions in this document will have sections devoted to common functionality (e.g. applies to all forms of directionality), as well as a section for each form of directionality (forward-only, bidirectional and undirected).

Namespace

The graph classes are kept in the std::graph namespace.

Common types and functions defined in the namespace are as follows.

```
namespace std::graph {
  using index_t   = ptrdiff_t;
  using vertex_id = index_t;
  using edge_id   = index_t;

  struct empty_value {}; // empty graph|vertex|edge value

  struct adj_list_type {};
  struct adj_array_type {};
  struct adj_matrix_type {};

  struct edge_type_directed_fwddir {};
  struct edge_type_directed_bidir {};
  struct edge_type_undirected {};
```

```
  struct edge_link_none {};
  struct edge_link_single {};
  struct edge_link_double {};

  struct mem_fixed {};
  struct mem_movable {};

  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) {}
  };

  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)) {}
  };

  // functions
  static constexpr vertex_id null_vertex_id();
  static constexpr edge_id   null_edge_id();
} // namespace std::graph
```

The template aliases for adjacency types are also included, as shown in the Adjacency Types section.

Graph Classes

The graph classes include graph, vertex, edge, vertex set (collection of vertices), and edge set. An edge set may be a collection of all edges (e.g. adjacency array and adjacency matrix) or a definition for the set of edges on a vertex (e.g. adjacency list).

The class definitions in this section show the functionality available for each class, with sections identifying types and functions available depending on the adjacency type (common, forward-only, directed, undirected).

All functions shown are those available for use. Because the implementation may use inheritance to compose the functionality, the actual implementation may be in another class that is derived from.

Vertices

Vertices have the ability to store **any** user-defined type. A vertex can have **outgoing** edges for forward-only graph, **outgoing** and **incoming** edges for bidirectional graphs, and **undirected** edges for undirected graphs.

Erasing a vertex will automatically erase all edges associated with it.

The proposed form of a vertex is:

```
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>
             , public out_edge_list<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>
             , public in_edge_list<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>
             , public undir_edge_list<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>
{
public:
  using user_value_type      = VV;
  using vertex_type          = vertex<ADJ, VV, VMEM, EV, EDIR, EMEM, ELNK>;
  using vertex_value_type    = pair<const vertex_id, vertex_type>;
  using base_user_value_type = conditional_t<
                                     graph_value_needs_wrap<VV>::value,
                                     graph_value<VV>,VV>;

  using edge_user_type  = EV;
  using edge_type       = edge<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using edge_value_type = edge_type;

  vertex() = default;
  vertex(user_value_type const& value) : base_user_value_type(value) {}
  vertex(user_value_type&& value) : base_user_value_type(move(value)) {}

  user_value_type&       user_value();
  user_value_type const& user_value() const;

//>>> Forward-only and Bidirectional
public:
  using out_edge_list_type = out_edge_list<ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>;
  out_edge_list&       out_edges();
  out_edge_list const& out_edges() const;
```

```
//>>> + Bidirectional
public:
  using in_edge_list_type  = in_edge_list <ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>;
  in_edge_list&       in_edges();
  in_edge_list const& in_edges() const;

//>>> Undirected
public:
  using undir_edge_list_type
        = undir_edge_list <ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>;
  undir_edge_list&       edges();
  undir_edge_list const& edges() const;
}
```

Edge containers all support the same behavior. Valid iterator operations is dependent on the underlying container (linked list or vector) used to store the edges. The following example shows the out_edge_list, and the same functionality is also available for in_edge_list and undir_edge_list.

```
template <class ADJ,
          class VV, class VMEM,
          class EV, class EDIR, class ELNK, class EMEM>
class out_edge_list {
public:
  class iterator;
  class const_iterator;

  size_t size() const;

  template<class G> iterator       begin();
  template<class G> const_iterator begin() const;
  template<class G> const_iterator cbegin() const;

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

Edges

Edges can contain any user-defined type as explained in the User Values section.

All edges are either **directed** or **undirected** as described in the section for Directed and Undirected.

```
template<class ADJ,
         class VV, class VMEM,
         class EV, class EDIR, class ELNK, class EMEM>
class edge : conditional_t<graph_value_needs_wrap<EV>::value,
                           graph_value<EV>, EV>
           , public out_vertex_link<ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>
           , public in_vertex_link<ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>
           , public out_edge_list_link<ADJ,VV,VMEM,EV,EDIR,ELNK,EMEM>
           , public in_edge_list_link<ADJ,VV,VMEM,EV,EDIR,ELNK, EMEM>
{
public:
  using vertex_type       = vertex<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using vertex_value_type = pair<const vertex_id, vertex_type>;

  using user_value_type      = EV;
  using edge_type            = edge<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using edge_value_type      = edge_type;
  using base_user_value_type = conditional_t<graph_value_needs_wrap<EV>::value,
graph_value<EV>, EV>;

  edge() = default;
  edge(vertex_value_type& u, vertex_value_type& v);
  edge(vertex_value_type& u, vertex_value_type& v,
       user_value_type const& value);
  edge(vertex_value_type& u, vertex_value_type& v,
       user_value_type&& value);

  user_value_type&       user_value();
  user_value_type const& user_value() const;

//>>> Forward-only and Bidirectional
public:
  vertex_type&       out_vertex();
  vertex_type const& out_vertex() const;

//>>> + Bidirectional
public:
  vertex_type&       in_vertex();
  vertex_type const& in_vertex() const;

//>>> Undirected
public:
  vertex_type&       vertex1();
  vertex_type const& vertex1() const;

  vertex_type&       vertex2();
```

```
  vertex_type const& vertex2() const;
};
```

Vertex Set

A vertex set contains the vertices of a graph and provides a common interface to standard containers to add, delete find and iterate across vertices. Performance of those operations is dependent on the underlying container. Iterator capabilities inherit the capabilities of the underlying container. The user can create a vertex set using their own container (e.g. a fast hashed container) as long as they implement the full interface.

Vertex types are defined as

```
  using user_value_type      = VV;
  using vertex_type          = vertex<ADJ, VV, VMEM, EV, EDIR, EMEM, ELNK>;
  using vertex_value_type    = pair<const vertex_id, vertex_type>;
```

where

- `user_value_type` is the user-defined value type stored in each vertex. It defaults to empty_value.
- `vertex_type` is the fully qualified type of a vertex class.
- `vertex_value_type` is the vertex value stored in the vertex set. It includes a uniform description that is used in vector-like and mapped containers.

`vertex_value_type` provides a consistent structure for algorithms, makes it easier to switch to a different `vertex_set` for evaluation, and guarantees the `vertex_id` is always known for a vertex. A `vertex_id` value is automatically assigned to a vertex, though the user can override the value when adding the vertex.

vertex_id is an integer that uniquely identifies a vertex and is guaranteed to be unique. Built-in support for `vertex_id` will be provided for the purposes of **searching** and **removing** vertices, as well as **assigning** data to new vertices (either by a user or automatically). The same interface is used for all vertex sets, regardless of the underlying container types.

The proposed form of the vertex set (with an underlying **vector** implementation) is:

```
template<class ADJ,
         class VV,
         class EV,
         class EDIR,
         class ELNK,
         class EMEM,
         class A>
class vector_vertex_set {
  using vertex_ref            = vertex_id;
```

```
  using memory_movable_type   = mem_movable;

  using vertex_user_type = VV;
  using vertex_type      = vertex<ADJ,VV,memory_movable_type,
                                  EV,EDIR,ELNK,EMEM>;
  using value_type       = pair<const vertex_id, vertex_type>;
  using allocator_type   = typename allocator_traits<typename A>::
                               template rebind_alloc<value_type>;

  using container_type   = vector<value_type, allocator_type>;
  using size_type        = typename container_type::size_type;
  using iterator         = typename container_type::iterator;
  using const_iterator   = typename container_type::const_iterator;

  vector_vertex_set() = default;
  vector_vertex_set(allocator_type alloc);

  size_type size() const;
  bool      empty() const;

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

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

  iterator       find(vertex_id id);
  const_iterator find(vertex_id id) const;

//>>> Adjacency List and Adjacency Array
  pair<iterator, bool>
  create(vertex_id id = graph_traits_type::null_vertex_id());

  pair<iterator, bool>
  create(vertex_user_type const& value, vertex_id id = null_vertex_id());

  pair<iterator, bool>
  create(vertex_user_type&& value, vertex_id id = null_vertex_id());

  void erase(iterator it);
  void erase(vertex_id id);
//<<< Adjacency List and Adjacency Array

  void resize(size_t n);
  void reserve(size_t n);
```

```
private:
  container_type vertices_;
};
```

vertex_set Proxy

A vertex set proxy is used to reduce compile-time dependencies and support user-defined containers and is used by the graph class. A full example of the proxy for the vector_vertex_set is

```
 struct vector_vertex_set_proxy {
    using memory_movable_type = mem_movable;

    template <class ADJ,
              class VV,
              class EV,
              class EDIR,
              class ELNK,
              class EMEM,
              class A>
    using vertex_set = vector_vertex_set<ADJ, VV, EV, EDIR, ELNK, EMEM, A>;
  };
```

`Vector_vertex_set_proxy` is the type passed in the VSP parameter of the graph.

vertex_set Types

`vertex_set`s are defined by the container type it uses to store vertices, with different tradeoffs:

Feature	**`vector`**	**`deque`**	**`ordered_map`**	**`unordered_map`**
Storage	Array (dense)	Block array (dense)	Node (sparse)	Node (sparse)
Traversal	O(1)	O(1)	O(1)	O(1)
In/out vector type on edge	vertex_id	pointer	pointer	pointer
Find vertex_id	O(1)	O(1)	O(log *n*)	O(>1)
Find edge	O(n)	O(n)	O(n)	O(n)
Add vertex	O(1)	O(1)	O(log n)	O(> 1)
Add edge	O(1)	O(1)	O(1)	O(1)
Erase vertex	O(1) + O(n)	O(1) + O(n)	O(log n) + O(n)	O(>1) + O(n)

(+ edges)				
Erase edge	O(1)	O(1)	O(1)	O(1)

The invariant `vertices[n].vertex_id == n` is used by `vector_vertex_set` and `deque_vertex_set`. This implies several things to achieve this:

1. If a new vertex is added with a specific vertex_id greater than the last vertex_id, then new null vertices will be allocated to fill the gap. For instance if 10 vertices have been created (vertex_id range 0..9), and a new vertex is added with vertex_id=20, then space is allocated for vertices with vertex_id range 10..19 before the new vertex is added. null_vertex_id() is assigned to the extra vertices to indicate they haven't been assigned yet.
2. Deletion of a vertex causes its destructor to be called and it's vertex_id to be set to null_vertex_id(). The size of the container doesn't change because erasing it from the container would invalidate the invariant.
3. All functions for find, create and erase need to behave consistently across all vertex_sets. For instance, find needs to fail when a vertex exists with vertex_id==null_vertex_id(), even though it physically exists. This is different from map, where vertex_id will never be null_vertex_id() and physical existence defines failure or success for find.

O(1) Traversal

Achieving O(1) traversal has different requirements for the underlying container used to store vertices in the vertex set. When using map, the location of the vertex remains stable for the life of the vertex so a pointer to the vertex can be stored for the in and out vertices on an edge. When a vector is used, a vertex's location is not stable. It will be moved in memory when the internal array needs to be resized as new vertices are added. In that case, the vertex_id is the only stable option to use for an edge's in and vertices and is used to find the vertex. vertex_id can also be used for map but incurs a O(log n) penalty when finding the vertex during traversal.

It's possible to accommodate both vector and map requirements for O(1) traversal by storing either a vertex_id or pointer to a vertex for the in and out vertices on an edge. The following definitions on a vertex set allows an edge to adapt to either a vertex_type* (shown) or vertex_id. The helper functions provide conversions to/from edge_vertex_ref and vertex references and vertex_id's.

```
using edge_vertex_ref = vertex_type*; // or, = vertex_id

edge_vertex_ref to_edge_vertex_ref(vertex_iterator);
edge_vertex_ref to_edge_vertex_ref(vertex_type&);

vertex_type&       get_edge_vertex(edge_vertex_ref);
vertex_type const& get_edge_vertex(edge_vertex_ref) const;
```

```
  vertex_id          get_vertex_id(edge_vertex_ref) const;
```

Supporting unordered_map is identical to map. deque can support both vertex_id or vertex_type* because the vertex won't move in memory (using iterators will not work in place of a pointer because they are invalidated in MSVC when anything is added or removed from the deque).

With a traversal time guarantee of O(1), the use of different vertex sets only impacts time for adding, removing and finding elements. It also has an impact on memory consumption. vector is the most compact storage, followed by deque, assuming few gaps in vertex_id usage. The advantage of using a deque over a vector is that it doesn't need to copy all vertices when expanding the container; it just adds a new block.

The use of map and unordered_map for vertex sets would be most useful when an integer id already exists for a vertex and is sparsely assigned. In all other cases a vector or deque vertex set should be sufficient.

Edge Set

An edge set is used to define the type of the collection of edges on a vertex (outgoing, incoming or undirected), and optionally provided storage of the edges in the case of an adjacency array and adjacency matrix. Where the edges are actually stored is internal to the implementation and should not affect the interface.

The edge set is ultimately responsible to take care of adding and deleting an edge, either directly or coordinating with the edge's vertices. It is also responsible for providing a way to iterate through all edges in the graph.

It's expected that the user will use the mirrored functions for adding and deleting edges in the graph class for convenience and a simpler interface, rather than using them directly hear. Iterating through edges would be done by using the begin/end functions because it is natural to do so, as an edge set appears to be a container.

```
template <class VV, class VSP, class EV, class ELNK, class A>
class edge_set<adj_array_type,VV,VSP,EV,edge_type_directed_fwddir,ELNK,A> {
  // types
public:
  using ADJ  = adj_array_type;
  using EDIR = edge_type_directed_fwddir;
  using EMEM = mem_movable;

  using allocator_type      = A;
  using memory_movable_type = EMEM;
```

```
  using vertex_user_type  = VV;
  using vertex_type       = vertex<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using vertex_value_type = pair<const vertex_id, vertex_type>;

  using edge_user_type  = EV;
  using edge_type       = edge<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using edge_value_type = edge_type;

  class iterator;
  class const_iterator;

  // Construction/Destruction
public:
  edge_set() = default;
  edge_set(allocator_type alloc);

  template<class G> iterator       begin();
  template<class G> const_iterator begin() const;
  template<class G> const_iterator cbegin() const;

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

  template<class G> void erase_edge(G& g, vertex_id uid, vertex_id vid);

//>>> forward-only and bidirectional
  class out_iterator;
  class const_out_iterator;

  template<class G>
  out_iterator create_edge(G& g, vertex_value_type& u, vertex_value_type& v);

  template<class G>
  out_iterator create_edge(G& g, vertex_value_type& u, vertex_value_type& v,
                           edge_user_type const& value);
  template<class G>
  out_iterator create_edge(G& g, vertex_value_type& u, vertex_value_type& v,
                           edge_user_type&& value);

//>>> undirected
  class undir_iterator;
  class const_undir_iterator;
  Undir_iterator create_edge(vertex_id uid, vertex_id vid);
  Undir_iterator create_edge(vertex_id uid, vertex_id vid,
                           edge_user_type const& value);
  Undir_iterator create_edge(vertex_id uid, vertex_id vid,
                           edge_user_type&& value);
```

```
};
```

The edge lists on the vertex are described in the description of vertex in this document.

Graph

The proposed form of the graph container for the adjacency list and adjacency array is:

```
template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
          class A>
class graph : public conditional_t<graph_value_needs_wrap<GV>::value,
                                   graph_value<GV>, GV>
{
  using base_t = conditional_t<graph_value_needs_wrap<GV>::value,
                                   graph_value<GV>,
                                   GV>;
  using graph_user_value = GV;
  using allocator_type   = A;

  using vertex_user_type  = VV;
  using vertex_type       = vertex<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using vertex_value_type = pair<const vertex_id, vertex_type>;

  using edge_user_type  = EV;
  using edge_type       = edge<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using edge_value_type = edge_type;

  using vertex_set =
        typename VSP::template vertex_set<ADJ, VV, EV, EDIR, ELNK, EMEM, A>;
  using vertex_iterator               = typename vertex_set::iterator;
  using const_vertex_iterator         = typename vertex_set::const_iterator;

  using edge_set    = typename edge_set<ADJ, VV, VSP, EV, EDIR, ELNK, A>;

  graph();
  graph(allocator_type alloc);
  graph(graph_user_type const& value, allocator_type alloc=allocator_type());
  graph(graph_user_type&& value, allocator_type alloc=allocator_type());
  graph(graph const&);

  allocator_type allocator() const;

  //
  // vertices
  //
```

```
vertex_set&       vertices();
vertex_set const& vertices() const;

void resize(size_t n);
void reserve(size_t n);

vertex_iterator       vertex_begin();
vertex_iterator       vertex_end();
const_vertex_iterator vertex_begin() const;
const_vertex_iterator vertex_end() const;
const_vertex_iterator vertex_cbegin() const;
const_vertex_iterator vertex_cend() const;

pair<vertex_iterator, bool> create_vertex(vertex_id=null_vertex_id());
pair<vertex_iterator, bool> create_vertex(vertex_user_type const&,
  vertex_id id = null_vertex_id());
pair<vertex_iterator, bool> create_vertex(vertex_user_type&&,
  vertex_id id = null_vertex_id());

bool erase_vertex(vertex_id);
bool erase_vertex(vertex_iterator);

vertex_iterator       find_vertex(vertex_id);
const_vertex_iterator find_vertex(vertex_id) const;

//
// All Edges
//
using edge_iterator       = typename edge_set::iterator;
class const_edge_iterator = typename edge_set::const_iterator;

edge_iterator       edge_begin();       // sentinel
const_edge_iterator edge_begin() const;
const_edge_iterator edge_cbegin() const;

edge_iterator       edge_end();         // sentinel
const_edge_iterator edge_end() const;
const_edge_iterator edge_cend() const;

edge_iterator  find_edge(vertex_iterator, vertex_iterator);
edge_iterator  find_edge(vertex_id, vertex_id);

edge_iterator create_edge(vertex_type& u, vertex_type& v);
edge_iterator create_edge(vertex_type& u, vertex_type& v,
                          edge_user_type const&);
edge_iterator create_edge(vertex_type& u, vertex_type& v,
                          edge_user_type&&);
```

```
  edge_iterator create_edge(vertex_iterator, vertex_iterator);
  edge_iterator create_edge(vertex_iterator, vertex_iterator,
                            edge_user_type const&);
  edge_iterator create_edge(vertex_iterator, vertex_iterator,
                            edge_user_type&&);

  edge_iterator create_edge(vertex_id, vertex_id);
  edge_iterator create_edge(vertex_id, vertex_id, edge_user_type const&);
  edge_iterator create_edge(vertex_id, vertex_id, edge_user_type&&);

  void erase_edge(edge_iterator);
  void erase_edge(vertex_iterator, vertex_iterator);
  void erase_edge(vertex_id, vertex_id);

//>>> Forward-only and Bidirectional
  //
  // Out Edges
  //
  using out_edge_iterator       = typename edge_set::out_iterator;
  using const_out_edge_iterator = typename edge_set::const_out_iterator;

  out_edge_iterator       out_edge_end();        // sentinel
  const_out_edge_iterator out_edge_end() const;

  out_edge_iterator create_out_edge(vertex_type& u, vertex_type& v);
  out_edge_iterator create_out_edge(vertex_type& u, vertex_type& v,
                                    edge_user_type const&);
  out_edge_iterator create_out_edge(vertex_type& u, vertex_type& v,
                                    edge_user_type&&);

  out_edge_iterator create_out_edge(vertex_iterator, vertex_iterator);
  out_edge_iterator create_out_edge(vertex_iterator, vertex_iterator,
                                    edge_user_type const&);
  out_edge_iterator create_out_edge(vertex_iterator, vertex_iterator,
                                    edge_user_type&&);

  out_edge_iterator create_out_edge(vertex_id, vertex_id);
  out_edge_iterator create_out_edge(vertex_id, vertex_id,
                                       edge_user_type const&);
  out_edge_iterator create_out_edge(vertex_id, vertex_id,
                                       edge_user_type&&);

  void erase_edge(out_edge_iterator);

  out_edge_iterator find_out_edge(vertex_iterator, vertex_iterator);
  out_edge_iterator find_out_edge(vertex_id, vertex_id);

//>>> + Bidirectional
```

```
  //
  // In Edges
  //
  using in_edge_iterator       = typename edge_set::in_iterator;
  using const_in_edge_iterator = typename edge_set::const_in_iterator;

  in_edge_iterator       in_edge_end();          // sentinel
  const_in_edge_iterator in_edge_end() const;

  in_edge_iterator create_in_edge(vertex_type& u, vertex_type& v);
  in_edge_iterator create_in_edge(vertex_type& u, vertex_type& v,
                                  edge_user_type const&);
  in_edge_iterator create_in_edge(vertex_type& u, vertex_type& v,
                                  edge_user_type&&);

  in_edge_iterator create_in_edge(vertex_iterator, vertex_iterator);
  in_edge_iterator create_in_edge(vertex_iterator, vertex_iterator,
                                  edge_user_type const&);
  in_edge_iterator create_in_edge(vertex_iterator, vertex_iterator,
                                  edge_user_type&&);

  in_edge_iterator create_in_edge(vertex_id, vertex_id);
  in_edge_iterator create_in_edge(vertex_id, vertex_id,
                                  edge_user_type const&);
  in_edge_iterator create_in_edge(vertex_id, vertex_id,
                                  edge_user_type&&);


  void erase_edge(in_edge_iterator);

  in_edge_iterator  find_in_edge(vertex_iterator, vertex_iterator);
  in_edge_iterator  find_in_edge(vertex_id, vertex_id);

  out_edge_iterator       to_out_edge_iterator(in_edge_iterator);
  const_out_edge_iterator to_out_edge_iterator(const_in_edge_iterator);
  in_edge_iterator        to_in_edge_iterator(out_edge_iterator);
  const_in_edge_iterator  to_in_edge_iterator(const_out_edge_iterator);
//>>> Undirected
  //
  // Undirected Edges
  //
  using undir_edge_iterator       = typename edge_set::undir_iterator;
  using const_undir_edge_iterator = typename edge_set::const_undir_iterator;

  undir_edge_iterator       undir_edge_end();          // sentinel
  const_undir_edge_iterator undir_edge_end() const;

  undir_edge_iterator create_undir_edge(vertex_type& u, vertex_type& v);
```

```
  undir_edge_iterator create_undir_edge(vertex_type& u, vertex_type& v,
                                edge_user_type const&);
  undir_edge_iterator create_undir_edge(vertex_type& u, vertex_type& v,
                                edge_user_type&&);

  undir_edge_iterator create_undir_edge(vertex_iterator, vertex_iterator);
  undir_edge_iterator create_undir_edge(vertex_iterator, vertex_iterator,
                                edge_user_type const&);
  undir_edge_iterator create_undir_edge(vertex_iterator, vertex_iterator,
                                edge_user_type&&);

  undir_edge_iterator create_undir_edge(vertex_id, vertex_id);
  undir_edge_iterator create_undir_edge(vertex_id, vertex_id,
                                    edge_user_type const&);
  undir_edge_iterator create_undir_edge(vertex_id, vertex_id,
                                    edge_user_type&&);

  void erase_edge(undir_edge_iterator);

  undir_edge_iterator find_undir_edge(vertex_iterator, vertex_iterator);
  undir_edge_iterator find_undir_edge(vertex_id, vertex_id);

//<<< End Directedness

private:
  vertex_set     vertex_set_;
  edge_set       edge_set_;
  allocator_type alloc_;
};
```

The proposed form of the graph container for the adjacency matrix follows. It differs from the adjacency list and adjacency array in that it requires knowing the number of vertices when the graph is constructed. Vertices and edges are also created when the graph is constructed. Vertices and edges cannot be created or erased.

```
template <class GV,
          class VV,
          class VSP,
          class EV,
          class A>
class graph<adj_matrix_type,
            GV,
            VV,
            VSP,
            EV,
            edge_type_directed_bidir,
```

```
            edge_link_none>
      : public conditional_t<graph_value_needs_wrap<GV>::value,
                             graph_value<GV>, GV>
{
  using base_t = typename conditional_t<graph_value_needs_wrap<GV>::value,
                                        graph_value<GV>,
                                        GV>;
  using graph_user_value = GV;
  using allocator_type   = A;
  using ADJ              = adj_matrix_type;
  using EDIR             = edge_type_directed_fwddir;
  using ELNK             = edge_link_none;

  using size_type        = allocator_type::size_type;

  using vertex_user_type  = VV;
  using vertex_type       = vertex<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using vertex_value_type = pair<const vertex_id, vertex_type>;

  using edge_user_type  = EV;
  using edge_type       = edge<ADJ, VV, VMEM, EV, EDIR, ELNK, EMEM>;
  using edge_value_type = edge_type;

  using vertex_set =
        typename VSP::template vertex_set<ADJ, VV, EV, EDIR, ELNK, EMEM, A>;
  using vertex_iterator               = typename vertex_set::iterator;
  using const_vertex_iterator         = typename vertex_set::const_iterator;

  using edge_set    = typename edge_set<ADJ, VV, VMEM, EV, EDIR, ELNK, A>;

  graph();
  graph(size_type vertex_count, allocator_type alloc);
  graph(size_type vertex_count, graph_user_type const& value,
        allocator_type alloc=allocator_type());
  graph(size_type vertex_count, graph_user_type&& value,
        allocator_type alloc=allocator_type());
  graph(graph const&);

  allocator_type allocator() const;

  // Vertices
  vertex_set&       vertices();
  vertex_set const& vertices() const;

  vertex_iterator       vertex_begin();
  vertex_iterator       vertex_end();
  const_vertex_iterator vertex_begin() const;
```

```
  const_vertex_iterator vertex_end() const;
  const_vertex_iterator vertex_cbegin() const;
  const_vertex_iterator vertex_cend() const;

  vertex_iterator       find_vertex(vertex_id);
  const_vertex_iterator find_vertex(vertex_id) const;

  // All Edges
  using edge_iterator       = typename edge_set::iterator;
  class const_edge_iterator = typename edge_set::const_iterator;

  edge_iterator       edge_begin();       // sentinel
  const_edge_iterator edge_begin() const;
  const_edge_iterator edge_cbegin() const;

  edge_iterator       edge_end();         // sentinel
  const_edge_iterator edge_end() const;
  const_edge_iterator edge_cend() const;

  edge_iterator  find_edge(vertex_iterator, vertex_iterator);
  edge_iterator  find_edge(vertex_id, vertex_id);

  // Out Edges
  using out_edge_iterator       = typename edge_set::out_iterator;
  using const_out_edge_iterator = typename edge_set::const_out_iterator;

  out_edge_iterator       out_edge_end();        // sentinel
  const_out_edge_iterator out_edge_end() const;

  out_edge_iterator find_out_edge(vertex_iterator, vertex_iterator);
  out_edge_iterator find_out_edge(vertex_id, vertex_id);

  // In Edges
  using in_edge_iterator       = typename edge_set::in_iterator;
  using const_in_edge_iterator = typename edge_set::const_in_iterator;

  in_edge_iterator       in_edge_end();          // sentinel
  const_in_edge_iterator in_edge_end() const;

  in_edge_iterator  find_in_edge(vertex_iterator, vertex_iterator);
  in_edge_iterator  find_in_edge(vertex_id, vertex_id);

  out_edge_iterator       to_out_edge_iterator(in_edge_iterator);
  const_out_edge_iterator to_out_edge_iterator(const_in_edge_iterator);
  in_edge_iterator        to_in_edge_iterator(out_edge_iterator);
  const_in_edge_iterator  to_in_edge_iterator(const_out_edge_iterator);

private:
```

```
  vertex_set     vertex_set_;
  edge_set       edge_set_;
  allocator_type alloc_;
};
```

Algorithms

We propose the addition of **non-member functions** to allow a user to perform **depth-** and **breadth-first** searches of a graph, find a **shortest path**, …, operations common in **data mining** research.

All work in this section needs revisiting in light of the range-v3 functionality that is coming in C++20. While informative, it should be considered a work-in-progress and will be changing.

Algorithm Class Design

When considering graph algorithms, the class design needs to use reasonable defaults and must allow a user to customize those algorithms to meet their needs. Below is the **Bellman-Ford shortest path** algorithm to demonstrate how this is accomplished. It provides a default design using an edge weight of 1 and weight type of `int`. Both edge function and type can be overridden. The function should accept any function object (lambda, functor, or free function) that meets the signature requirements of the algorithm, as defined by the WeightFnc template parameter.

```
template<class G,
         class Weight = int,
         class WeightFnc = function<Weight(GT::edge_type&)>>
class bellman_ford_shortest_paths_example {
public:
  using graph_type        = typename G;
  using vertex_value_type = typename G::vertex_value_type;
  using edge_value_type   = typename G::edge_value_type;

  using weight_type     = Weight;
  using edge_weight_fnc = WeightFnc;
  using path            = ...;

public:
  bellman_ford_shortest_paths_example(
           graph_type&      g,
```

```
          edge_weight_fnc& f_edge_weight =
                                []](edge_type&) -> weight_type { return 1; })
    : graph_(&g), edge_weight_fnc_(f_edge_weight) {}

  // return shortest path between u & v
  path operator()(vertex_type const& u, vertex_type const& v);

private:
  graph_type*     graph_;
  edge_weight_fnc edge_weight_fnc_;
    // accessor: edge_weight_fnc_(uv) -> weight_type
};
```

The default usage of this class is given as follows.

```
G g;
(add vertices & edges)
bellman_ford_shortest_paths<G> bfsp(g);
auto& uit = g.find_vertex(...);
auto& vit = g.find_vertex(...);
auto path = bfsp(uit,vit);
```

Depth-First & Breadth-First Searches

DFS (Depth-First Search) functions

```
  template <class FwdGraph, class Visitor>
  void depth_first_visit(FwdGraph&                              g,
                         typename FwdGraph::vertex_iterator first,
                         typename FwdGraph::vertex_iterator last,
                         Visitor vis = dfs_visit_base<FwdGraph>);

  // interface function for Breadth First Search
  template <class FwdGraph, class Visitor>
  void depth_first_visit(FwdGraph&                               g,
                         typename FwdGraph::vertex_value_type& u,
                         Visitor vis = dfs_visit_base<FwdGraph>);
```

Base class to derive a DFS visitor from

```
  template <class G>
  struct dfs_visit_base {
    using graph_type  = G;
    using vertex_value_type = typename G::vertex_value_type;
    using edge_value_type   = typename G::edge_value_type;
```

```
  void discover_vertex(vertex_value_type& u) {}
  void examine_vertex(vertex_value_type& u) {}
  void finish_vertex(vertex_value_type& u) {}

  void examine_out_edge(edge_value_type& uv) {}
};
```

BFS (Breadth-First Search) functions

```
template <class FwdGraph, class Visitor>
void breadth_first_visit(FwdGraph&                          g,
                         typename FwdGraph::vertex_iterator first,
                         typename FwdGraph::vertex_iterator last,
                         Visitor vis = bfs_visit_base<FwdGraph>);

// interface function for Breadth First Search
template <class FwdGraph, class Visitor>
void breadth_first_visit(FwdGraph&                            g,
                         typename FwdGraph::vertex_value_type& u,
                         Visitor vis = bfs_visit_base<FwdGraph>);
```

Base class to derive a visitor from

```
template <class G>
struct bfs_visit_base {
  using graph_type  = G;
  using vertex_value_type = typename G::vertex_value_type;
  using edge_value_type   = typename G::edge_value_type;

  void discover_vertex(vertex_value_type& u) {}
  void examine_vertex(vertex_value_type& u) {}
  void finish_vertex(vertex_value_type& u) {}

  void examine_out_edge(edge_value_type& uv) {}
};
```

Shortest Paths

Future: Bellman-Ford algorithm
Future: Dijsktra algorithm

erase() and erase_if() non-member functions

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

```
template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
```

```
          class A>
void erase(typename graph<ADJ,GV,VV,VSP,EV,EDIR,ELNK,A>::vertex_set& vs,
           VV const& value);

template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
          class A, class Pred>
void erase_if(typename graph<ADJ,GV,VV,VSP,EV,EDIR,ELNK,A>::vertex_set& vs,
              Pred pred);

template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
          class A>
void erase(typename graph<ADJ,GV,VV,VSP,EV,EDIR,ELNK,A>::edge_set& vs,
           EV const& value);

template <class ADJ, class GV,
          class VV, class VSP,
          class EV, class EDIR, class ELNK,
          class A, class Pred>
void erase_if(typename graph<ADJ,GV,VV,VSP,EV,EDIR,ELNK,A>::edge_set& vs,
              Pred pred);
```

Design Notes

A graph contains a number of **types** and **containers**, including **vertices**, **edges** and **edge lists**, making it more complex than a standard container that holds a single element type.

The interrelated nature of the elements of a graph brings additional challenges. Standard containers take full ownership of their elements, but edges belong to both incoming and outgoing edge lists for the in and out vertices, requiring it to use an intrusive idiom where each list's node is a member variable on the edge. The edge takes on more responsibility during its construction and destruction.

Because all the types are closely related, it becomes challenging to break cyclic dependencies during compilation. Defining a common set of template parameters that are used consistently helps with this. It adds more parameters than is common, but allows each class to define a related class without having to include its definition in a header. All it needs is a forward declaration of the class.

User-defined values are a single type, specified via template parameters for graph, vertex and edge. They can be any value type and are used as the base class of the graph, vertex or edge

that uses it. Types that aren't normally allowed to be base classes are wrapped in a special `graph_value struct` with a single "value" member with the defined type. This gives a natural access to the values as public member variables. When a value isn't needed, a special `empty_value struct` with no members is used.

Template arguments are customization points for graph, vertex and edge values, and for selecting the type of `vertex_set` based on `vector`, `map` or `unordered_map`.

Open Questions & Issues

1. How should existing Concepts be used by graph types?
2. What additional Concepts might be useful for graph? (e.g. directed/undirected, cyclic/acyclic, …)
3. How does range v3 impact algorithm and iterator design?
4. Should coroutines be considered in the algorithm design? How?
5. Should this be in its own module?
6. Where would constexpr add the most value?
7. Are reverse iterators desired/useful?

Acknowledgements

This paper is thanks to discussion in 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