P1709R1: Graph Library

Date:	2019-06-17
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 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
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 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.

Example

Concepts

The concepts shown in this section are a work in progress. They exhibit useful concepts when using a graph but not all definitions are defined yet. Other concepts may be added as more algorithms and patterns of requirements are discovered.

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

```
template<typename G>
concept graph_c = requires(G&& g) {
```

```
  vertices(g);
  edges(g);
  //value(g);  // value is optional
}

template<typename G> concept directed_graph_c;
template<typename G> concept bidirected_graph_c;
template<typename G> concept undirected_graph_c;

template<typename G> concept sparse_graph_c;
template<typename G> concept dense_graph_c;  // e.g. adjacency_matrix

template<typename V> concept vertex_c;
template<typename V> concept edge_c;

template<typename V> concept vertex_iterator_c;
template<typename V> concept edge_iterator_c;

template<typename V> concept eraseable_c; // items can be erased
                                          // (e.g. vertices or edges)

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

// for DFS, BFS & TopoSort iterators
template<typename SI> concept search_iterator_c
  requires { forward_iterator<T> && depth(SI); };
```

Type Traits

```
template<graph_c G>
struct is_adjacency_list;

template<graph_c G>
inline constexpr bool is_adjacency_list_v = is_adjacency_list<G>::value;

template<graph_c G>
struct is_adjacency_array;

template<graph_c G>
inline constexpr bool is_adjacency_array_v = is_adjacency_array<G>::value;

template<graph_c G>
struct is_adjacency_matrix;

template<graph_c G>
inline constexpr bool is_adjacency_matrix_v = is_adjacency_matrix<G>::value;
```

Types

In edges are only valid for graphs that have them.

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


template <graph_c G>
using vertex_key_t = typename G::vertex_key_type;

template <graph_c G>
using vertex_value_t = typename G::vertex_user_value;

template <graph_c G>
using vertex_range_t = typename G::vertex_range;

template <graph_c G>
using vertex_iterator_t = typename G::vertex_iterator;

template<graph_c G>
using vertex_sentinal_t = typename G::vertex_sentinal;

template <graph_c G>
using vertex_size_t = typename G::vertex_size_t;

template <graph_c G>
using edge_size_t = typename G::edge_size_t;


template <graph_c G>
using edge_value_t = typename G::edge_user_value;

template <graph_c G>
using edge_range_t = typename G::edge_range;

template <graph_c G>
using edge_iterator_t = typename G::edge_iterator;

template<graph_c G>
using edge_sentinal_t = typename G::edge_sentinal;

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

```
template <graph_c G>
using out_edge_range_t = typename G::out_edge_range;

template <graph_c G>
using out_edge_iterator_t = typename G::out_edge_iterator;

template <graph_c G>
using out_edge_sentinal_t = typename G::out_edge_sentinal;

template <graph_c G>
using out_edge_size_t = typename G::out_edge_size_type;


template <graph_c G>
using in_edge_range_t = typename G::in_edge_range;

template <graph_c G>
using in_edge_iterator_t = typename G::in_edge_iterator;

template <graph_c G>
using in_edge_sentinal_t = typename G::in_edge_sentinal;

template <graph_c G>
using in_edge_size_t = typename G::in_edge_size_type;
```

Graph Functions

Common Functions

```
template <typename T>
auto value(T& gve) -> decltype(get_user_value(gve));
```

Vertex Functions

```
template <graph_c G>
auto vertex_key(vertex_t<G>& u) -> vertex_key_t<G>&;


template<graph_c G>
auto out_edges(G& g, vertex_t<G>& u) -> out_edge_range_t<G>;

template<graph_c G>
auto out_size(G& g, vertex_t<G>& u) -> out_edge_size_t<G>;

template<graph_c G>
auto out_degree(G& g, vertex_t<G>& u) -> out_edge_size_t<G>;
```

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


template<graph_c G>
auto in_edges(G& g, vertex_t<G>& u) -> in_edge_range_t<G>;

template<graph_c G>
auto in_size(G& g, vertex_t<G>& u) -> in_edge_size_t<G>;

template<graph_c G>
auto in_degree(G& g, vertex_t<G>& u) -> in_edge_size_t<G>;

template<graph_c G>
void clear_in_edges(G& g, vertex_t<G>& u);


template<graph_c G>
auto create_vertex(G& g) -> pair<vertex_iterator<G>,bool>;

template<graph_c G>
auto create_vertex(G& g, vertex_value_t<T>&)
     -> pair<vertex_iterator<G>,bool>;

template<graph_c G>
auto create_vertex(G& g, vertex_value_t<T>&&)
     -> pair<vertex_iterator<G>,bool>;


template<graph_c G>
void erase_vertices(G& g, vertex_range_t<T>&);

template<graph_c G>
void erase_vertex(G& g, vertex_iterator_t<T>&);

template<graph_c G>
void erase_vertex(G& g, vertex_key_t<T>&);

template<graph_c G>
void clear_vertex(G& g, vertex_iterator_t<T>&);

template<graph_c G>
auto find_vertex(G& g, vertex_key_t<T>&) -> vertex_iterator_t<G>;
```

Edge Functions

```
template<graph_c G>
```

```
auto out_vertex(G& g, edge_iterator_t<G>) -> vertex_iterator<G>;

template<graph_c G>
auto out_vertex(G& g, out_edge_iterator_t<G>) -> vertex_iterator<G>;

template<graph_c G>
auto out_vertex(G& g, in_edge_iterator_t<G>) -> vertex_iterator<G>;


template<graph_c G>
auto in_vertex(G& g, edge_iterator_t<G>) -> vertex_iterator<G>;

template<graph_c G>
auto in_vertex(G& g, out_edge_iterator_t<G>) -> vertex_iterator<G>;

template<graph_c G>
auto in_vertex(G& g, in_edge_iterator_t<G>) -> vertex_iterator<G>;


template<graph_c G>
auto create_edge(G& g, vertex_iterator_t<G>, vertex_iterator_t<G>);

template<graph_c G>
auto create_edge(G& g,
                 vertex_iterator_t<G>,
                 vertex_iterator_t<G>,
                 edge_value_t<G>&);

template<graph_c G>
auto create_edge(G& g,
                 vertex_iterator_t<G>,
                 vertex_iterator_t<G>,
                 edge_value_t<G>&&);


template<graph_c G>
auto create_edge(G& g, vertex_key_t<G>&, vertex_key_t<G>&);

template<graph_c G>
auto create_edge(G& g,
                 vertex_key_t<G>&,
                 vertex_key_t<G>&,
                 edge_value_t<G>&);

template<graph_c G>
auto create_edge(G& g,
                 vertex_key_t<G>&,
```

```
                vertex_key_t<G>&,
                edge_value_t<G>&&);


template<graph_c G>
void erase_edges(G& g, edge_range_t);

template<graph_c G>
void erase_edges(G& g, out_edge_range_t);

template<graph_c G>
void erase_edges(G& g, in_edge_range_t);


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

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

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

template<graph_c G>
void erase_edge(G& g, out_edge_iterator_t<G> uv);

template<graph_c G>
void erase_edge(G& g, in_edge_iterator_t<G> uv);


template<graph_c G>
auto find_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v) ->
edge_iterator<G>;

template<graph_c G>
auto find_edge(G& g, vertex_key_t<G>& ukey, vertex_key_t<G>& vkey) ->
edge_iterator<G>;


template<graph_c G>
auto find_out_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v) ->
out_edge_iterator<G>;

template<graph_c G>
auto find_out_edge(G& g, vertex_key_t<G>& ukey, vertex_key_t<G>& vkey) ->
out_edge_iterator<G>;
```

```
template<graph_c G>
auto find_in_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v) ->
in_edge_iterator<G>;

template<graph_c G>
auto find_in_edge(G& g, vertex_key_t<G>& ukey, vertex_key_t<G>& vkey) ->
in_edge_iterator<G>;
```

Graph Functions

```
template<graph_c G>
auto vertices(G& g) -> vertex_range_t;

template<graph_c G>
auto vertices_size(G& g) -> vertex_size_t<G>;

template<graph_c G>
auto edges(G& g) -> edge_range_t;

template<graph_c G>
auto edges_size(G& g) -> edge_size_t<G>;

template<graph_c G>
void clear(G& g);

template<graph_c G>
void swap(G& a, G& b);

template<graph_c G>
auto create_adjacency_list(G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
auto create_adjacency_list(const graph_value_t<G>&,
                           G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
auto create_adjacency_list(graph_value_t<G>&&,
                           G::allocator_t alloc=G::allocator_t()) -> G*;


template<graph_c G>
auto create_adjacency_array(G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
auto create_adjacency_array(const graph_value_t<G>&,
                            G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
```

```
auto create_adjacency_array(graph_value_t<G>&&,
                            G::allocator_t alloc=G::allocator_t()) -> G*;


template<graph_c G>
auto create_adjacency_matrix(G::size_type vtx_cnt,
                             G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
auto create_adjacency_matrix(G::size_type vtx_cnt,
                             const graph_value_t<G>& gval,
                             G::allocator_t alloc=G::allocator_t()) -> G*;

template<graph_c G>
auto create_adjacency_matrix(G::size_type vtx_cnt,
                             graph_value_t<G>&& gval,
                             G::allocator_t alloc=G::allocator_t()) -> G*;
```

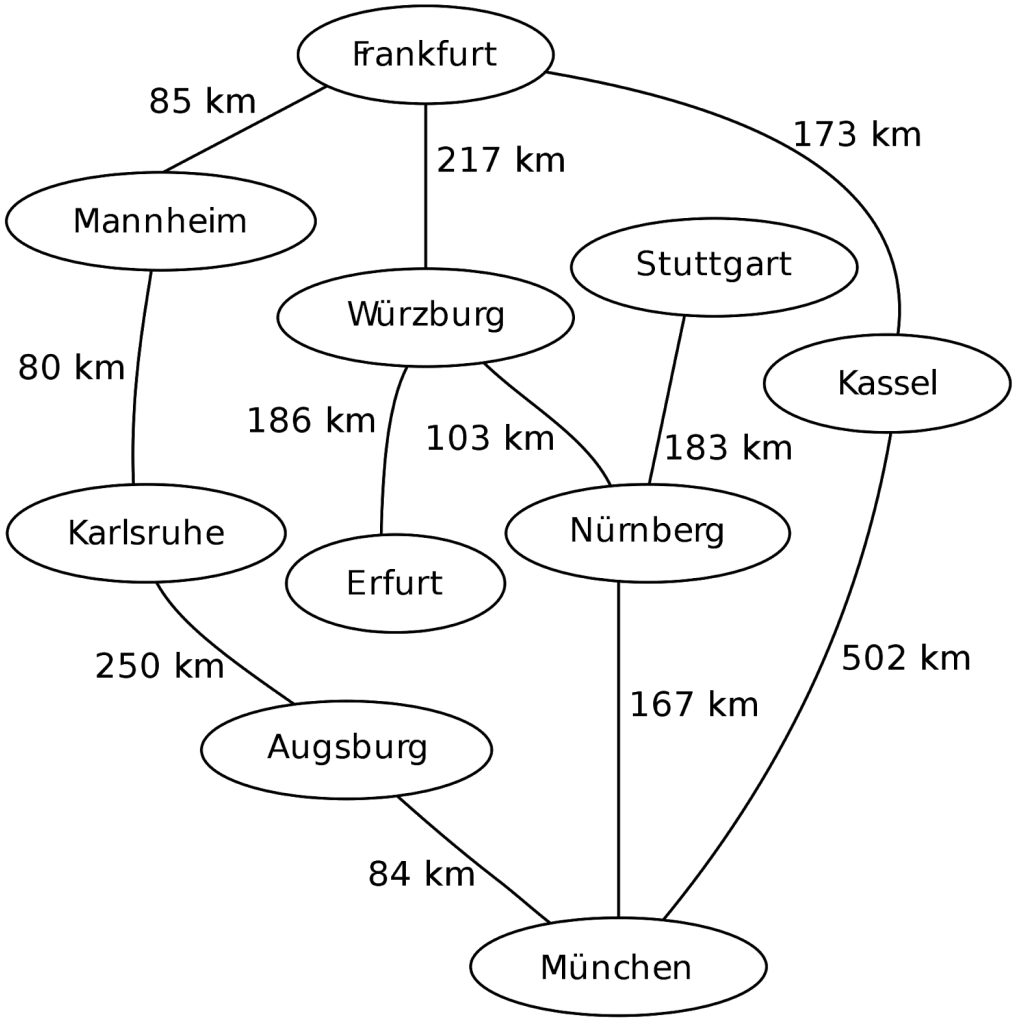
Iterators

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 Wikipedia article on Breadth-first search)

Depth First Search (DFS)

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

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

template<graph_c G, vertex_iterator_c I>
auto depth_first_search(I) -> dfs_range<G>;
```

Example

```
 for (auto city : depth_first_search(find_vertex("Frankfürt")))
    cout << string(depth(city) * 2) << vertex_key(*city) << '\n';
  /* Output
```

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

Breadth First Search (BFS)

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

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

template<graph_c G, vertex_iterator_c I>
auto bread_first_search(I) -> bfs_range<G>;
```

Example

```
 for (auto city : breadth_first_search(find_vertex("Frankfürt")))
    cout << string(depth(city) * 2) << vertex_key(*city) << '\n';
  /* Output
      Frankfürt
        Mannheim
        Würzburg
        Kassel
          Karlsruhe
          Erfurt
            Nürnberg
              Augsburg
              Stuttgart
              München
  */
```

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>;
```

Iterator Functions

These functions take BFS, DFS and TopoSort iterators.

```
template<vertex_iterator_c Search, typename I>
auto depth(Search) -> I;     // distance from starting (root) iterator

template<vertex_iterator_c Search>
bool is_first_visit(Search); // first time a vertex is visited?

template<vertex_iterator_c Search>
bool is_last_visit(Search); // last time a vertex will be visited?
```

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.

Concurrency and parallelism are not included because they are difficult or impossible to do in a general way. In particular, shared vertices between edges are difficult to provide significant performance benefits.

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 the type of weight it supports.

Two variants are supplied for each algorithm, one for a single source vertex and another as a range of vertex sources.

A shortest path between two vertices is described by a tuple as follows.

```
template<Iterator vtx_iter_t, arithmetic_c distance_t>
using shortest_path = tuple<vtx_iter_t, vtx_iter_t, distance_t>;
                    // tuple<from, to, distance>
```

Example

```
  using G = adjacency_list<name_value, double, empty_value,
                           edge_type_directed_fwddir>;
  auto g  = create_adjacency_list<G>();
  // (fill graph)

  vector <tuple<vertex_iterator_t<G>, vertex_iterator_t<G>, int> short_paths;
  bellman_ford_shortest_paths(
        g,
        g.vertices().begin(),
        [](edge_value_t<G>& uv) -> ptrdiff_t
             { return value(uv).weight; },
        back_inserter(short_paths));
  // short_paths hold a collection of shortest paths
```

Bellman-Ford Shortest Paths

weight_fnc is a function object that returns either negative or positive weight for an edge.

```
template<
  graph_c      G,
  typename     WFnc,
  arithmetic_c Dist = decltype(WFnc),
  OutputIterator<shortest_path<vertex_iterator_t<G>, Dist>> OutIter
>
void bellman_ford_shortest_paths(
       G&                   g,
       vertex_iterator_t<G> start_vertex,
       WFnc                 weight_fnc
                            = [](edge_value_t<G>&) -> ptrdiff_t {return 1;},
       OutIter              result_iter);


template<
  graph_c      G,
  typename     WFnc,
  arithmetic_c Dist = decltype(WFnc),
  OutputIterator<shortest_path<vertex_iterator_t<G>, Dist>> OutIter
>
void bellman_ford_shortest_paths(
       G&              g,
       vertex_range<G> start_vertices,
       WFnc,           weight_fnc
```

```
                    = [](value_t<G>&) -> ptrdiff_t { return 1; },
       OutIter      result_iter);
```

Dikstra's Shortest Paths

weight_fnc is a function object that returns a non-negative weight for an edge. Signed integers and floating point types are allowed and it is the callers responsibility to assure they are non-negative.

```
template<
  graph_c      G,
  typename     WFnc,
  arithmetic_c Dist=decltype(WFnc),
  OutputIterator<shortest_path<vertex_iterator_t<G>, Dist>> OutIter
>
void dijkstra_shortest_paths(
       G&                   g,
       vertex_iterator_t<G> start_vertex,
       WFnc                 weight_fnc
                            = [](edge_value_t<G>&) -> size_t {return 1;},
       OutIter              result_iter);

template<
  graph_c      G,
  typename     WFnc,
  arithmetic_c Dist = decltype(WFnc),
  OutputIterator<shortest_path<vertex_iterator_t<G>, Dist>> OutIter
>
void dijkstra_shortest_paths(
       G&              g,
       vertex_range<G> start_vertices,
       WFnc,           weight_fnc
                       = [](edge_value_t<G>&) -> size_t { return 1; },
       OutIter         result_iter);
```

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, integralComp = size_t>
using component = tuple<Comp, vtx_iter_t>;

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

```
template<Iterator vtx_iter_t, integralComp = size_t>
using bicomponent = tuple<Comp, vtx_iter_t>;
```

```
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

```
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

```
template<
  undirected_graph_c  G,
  OutputIterator<tuple<vertex_iterator_t<G>,vertex_iterator_t<G>>> OutIter
>
```

```
void transitive_closure(
       G&                    g,
       OutIter               result_iter);
```

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

Graph Data Structures

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.

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 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