P1709R3: Graph Library

Date:	2020-05-04
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21
Authors:	Phillip Ratzloff (SAS Institute) Richard Dosselmann (U of Regina) Michael Wong (Codeplay) Matthew Galati (SAS Institute) Andrew Lumsdaine (PNNL / University of Washington) Jens Maurer Domagoj Saric Jesun Firoz (PNNL) Kevin Deweese (University of Washington)
Contributors:	
Emails:	phil.ratzloff@sas.com dosselmr@cs.uregina.ca michael@codeplay.com Matthew.Galati@sas.com al75@uw.edu
Reply to:	phil.ratzloff@sas.com

Revision History

Revision	Description
P1709R3b	1. General a. Move library from std::graph into std namespace. b. Tighten up definition of directed & undirected graphs based on ordered or unordered vertices on an edge. c. Changed from Ariel font to Cambria for text and Bree Serif for headers. 2. Concepts a. Replace most previous graph concept definitions with new definitions: **incidence_graph, adjacency_graph, edge_list_graph, adjacency_matrix.** b. Add new mutability concepts: **incremental_vertex_graph, decremental_vertex_graph, dynamic_vertex_graph, static_vertex_graph, incremental_edge_graph, decremental_edge_grap, dynamic_edge_graph, static_edge_graph, dynamic_graph, static_graph** c. Add new path concepts: **vertex_path, edge_path, vertex_cycle, edge_cycle.** d. Add new directedness concepts: **directed**, **undirected**, **directed_or_undirected**. e. Other: **edge_iterator**, **const_edge_iterator, adjacency_iterator, const_adjacency_iterator** to help consolidate edge property functions. 3. Algorithms a. Require integral key type and random_access_range b. Use new concepts 4. Uniform API a. Reduce types in graph_traits to be value types and ranges. b. Replace vertex & edge references with iterators in graph API functions to make it easier for vertices in non-contiguous containers. c. Add **vertex_vertex_range, vertex_outward_vertex_range, vertex_indward_vertex_range** for adjacency graphs. d. Replace value(gve) with **graph_value**(g), **vertex_value**(g,u), **edge_value**(g,uv). e. Add **contains_vertex** and **contains_edge** functions. f. Remove range-specific functions in the API in favor of using standard range functions: **begin, end, cbegin, cend, empty, size, ssize**. g. Return optional<T> instead of pair<T,bool> for create functions.

	h. Add **create_edges**(g,erng,vrng,ekey_fnc,evalue_fnc,vvalue_fnc), **create_edges**(g,erng,ekey_fnc,evalue_fnc), **create_vertices**(g,vrng,vvalue_fnc) i. Add **ordered_pair** & **unordered_pair** structs for directed & undirected concepts. j. Remove functions specific to ranges in favor of using the ranges-based functions; e.g. size(edges(g,u)) instead of edges_size(g,u). k. Consolidate edge property functions with new **edge_iterator** and **adjacency_iterator** concepts. l. Rename ssize_t (and related) to **difference_t** to match names used in std. m. Add **degree**(g,u), **outward_degree**(g,u), **inward_degree**(g,u) functions by request. n. Rename edge property functions: inward_vertex -> **source_vertex**, outward_vertex -> **target_vertex**, etc. o. Add vertex_vertex and edge_list range adaptors (incomplete). p. Add **graph_allocator**(g). q. Remove **resize_vertices**(g) and **reserve_edges**(g) because of questionable value. 5. Directed API a. Added concepts: **outward_incidence_graph, outward_adjacency_graph, inward_incidence_graph, inward_adjacency_graph** b. Added mapping adaptors to map ranges and functions to the Uniform API (incomplete). 6. Data Structures a. Rename IndexT → KeyT template parameter b. Add **VContainer** and **EContainer** template parameters to directed_adjacency_vector, and VContainer template parameter to undirected_adjacency_list.
P1709R3a	1. All algorithms have variants that accept an iterator or a range. 2. Add **output_iterator** concept to algorithms with an output iterator. 3. Add **vertex_begin(g) / vertex_end(g)**, **edge_begin(g) / edge_end(g)** and **edge_begin(g,u) / edge_end(g,u)** functions to the uniform API. 4. Add **topological_sort_vertex_range** and **topological_sort_edge_range** definitions. 5. Rewrite **Graph Data Structures** section using directed_adjacency_vector & undirected_adjacency_list from the prototype. Dropped wording that an adjacency matrix implementation will be included in the library. 6. Add section for **Adapting to Existing Graphs**.

	7. Add **graph_traits<G>** for adaptability to external graphs. Remove type definitions from graph class (no longer necessary because of graph_traits). Also removed vertex & edge class definitions because they aren't used directly and cause confusion. 8. Add free functions for graphs which reflect the vertices in the graph: size, ssize, begin, end, cbegin, cend. 9. Remove degree functions (alias of existing size functions) to keep API simple. 10. Rename directed_adjacency_array to directed_adjacency_vector. 11. Add -ward suffix to in and out to create inward and outward terms. 12. Add **edge_key_t**<G>. 13. Add **vertices(g,u)**, **vertices_size(g,u)**, **vertices_ssize(g,u)**, **outward_vertices(g,u)**, **inward_vertices(g,u)**.
P1709R2	1. Define the **uniform API** for undirected & directed algorithms (an extended API also exists for directed graphs). 2. Added **concepts** for undirected, directed & bidirected graphs. 3. Refined **DFS & BFS range** definitions from prototype experience. 4. Refined **Shortest Paths** & **Transitive Closure** algorithms from input & prototype experience.
P1709R1	Rewrite with a focus on a **pure functional design**, emphasizing the algorithms and graph API. Also added concepts and ranges into the design. Addressed concerns from Cologne review to change to functional design.
P1709R0	Focus on **object-oriented API** for data structures and example code for a few algorithms.

Introduction

This document proposes the addition of (general) **graph** algorithms and data structures to the C++ **containers** library. Graphs are a fundamental abstraction in Computer Science that generalize basic containers by expressing relationships between entities. Numerous applications of graph algorithms exist in areas as diverse as electronic design automation, operations research, proteomics, machine learning, etc. Given the importance and ubiquity of graphs in Computer Science, standardized algorithms and data structures will have significant impact. This paper proposes an **interface** for a library of graph algorithms and data structures. This should be considered a proof of concept.

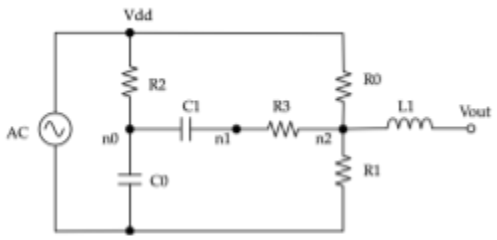
Impact on the Standard

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

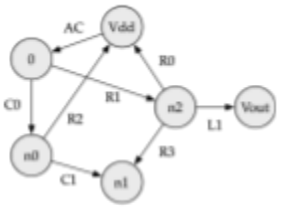
Background and Terminology

We present some standard terminology, which closely follows Cormen, Leiserson, Rivest, and Stein, 3rd Edition (hereafter referred to as CLRS).

A *graph* $G$is a pair of two sets V and E, where V is a (finite) set of elements called, and $E$is a set of pairs, each pair consisting of elements from V. We write $G = (V, E)$; an element in the set $V$is called a *vertex* (plural vertices) and an element in set E is called an *edge.* The elements of Emay be ordered pairs or unordered pairs, which we define in the following way. Given two vertices u and v, with $u \neq v$, $(u, v) \neq (v, u)$ for an ordered pair, whereas $(u, v) = (v, u)$ for an unordered pair. A graph whose edges are ordered is called a *directed* graph (sometimes, *digraph*); a graph whose edges are unordered is called an *undirected* graph. We denote the number of vertices in $V$by $|V|$ and the number of edges in $E$by $|E|$. Finally, vertices and edges often have various kinds of *properties* associated with them, stemming from what the graph is actually attempting to represent, and which may be used by algorithms during their execution.



$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{0, \text{Vdd}, \text{n0}, \text{n1}, \text{n2}, \text{Vout}\} \\
E &= \{(\text{n0}, \text{n1}), (\text{Vdd}, 0), \\
&\quad (\text{n0}, \text{Vdd}), (\text{n2}, \text{Vdd}), \\
&\quad (0, \text{n2}), (\text{n2}, \text{Vout}), \\
&\quad (0, \text{n0}), (\text{n2}, \text{n1})\}
\end{aligned}
$$

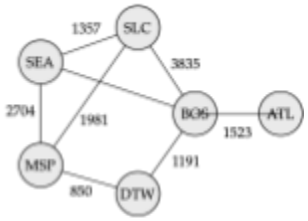


An electrical circuit modeled as a directed graph, showing both the set notation (center) and graphical representation (right). The circuit node names are vertex properties and the type of the circuit element between nodes are edge properties.

Airport
SEA
MSP
SLC
DTW
ATL
BOS

MSP	DTW	850
SLC	SEA	1357
MSP	SLC	1981
BOS	SLC	3835
SEA	BOS	4016
BOS	ATL	1523
SEA	MSP	2704
BOS	DTW	1191

$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{\text{SEA}, \text{MSP}, \text{SLC}, \text{DTW}, \text{ATL}, \text{BOS}\} \\
E &= \{\{\text{MSP}, \text{DTW}\}, \{\text{SLC}, \text{SEA}\}, \\
&\quad \{\text{MSP}, \text{SLC}\}, \{\text{BOS}, \text{SLC}\}, \\
&\quad \{\text{SEA}, \text{BOS}\}, \{\text{BOS}, \text{ATL}\}, \\
&\quad \{\text{SEA}, \text{MSP}\}, \{\text{BOS}, \text{DTW}\}\}
\end{aligned}
$$



An airline route table modeled as an undirected graph, with set notation (center) and graphical representation (right). The three-letter city codes are vertex properties and the distance in km for the indicated routes are edge properties.

We may also have the case of a graph representing relationships between two different kinds of entities. In that case we write $G = (U, V, E)$, where U and V are distinct sets of different kinds of entities (and in general U and V will not have the same cardinality). The set E of edges consists of ordered pairs (u, v), with u coming from U and v coming from V. This kind of graph is known as a *bipartite* graph; to distinguish it from a bipartite graph, a graph with just a single vertex set may also be referred to as a *unipartite* graph.

Movie Title	Starring
A Few Good Men	Kevin Bacon, Jack Nicholson
Balto	Kevin Bacon
Top Gun	Tom Cruise
Black Swan	Natalie Portman
V for Vendetta	Natalie Portman, Hugo Weaving
The Matrix	Carrie Ann Moss, Hugo Weaving
Footloose	Kevin Bacon

$$
\begin{aligned}
G &= \{U, V, E\} \\
V &= \{\text{A Few Good Men, Balto, Top Gun, Black Swan, V for Vendetta, The Matrix, Footloose}\} \\
V &= \{\text{Tom Cruise, Kevin Bacon, Hugo Weaving, Cary Anne Moss, Natalie Portman, Jack Nicholson}\} \\
E &= \{\{\text{Footloose, Kevin Bacon}\}, \{\text{Balto, Kevin Bacon}\}, \\
&\quad \{\text{A Few Good Men, Kevin Bacon}\}, \{\text{A Few Good Men, Jack Nicholson}\}, \\
&\quad \{\text{A Few Good Men, Tom Cruise}\}, \{\text{Top Gun, Tom Cruise}\}, \\
&\quad \{\text{Black Swan, Natalie Portman}\}, \{\text{V for Vendetta, Natalie Portman}\}, \\
&\quad \{\text{V for Vendetta, Hugo Weaving}\}, \{\text{The Matrix, Hugo Weaving}\}, \\
&\quad \{\text{V for Vendetta, Cary Anne Moss}\}\}
\end{aligned}
$$

Movie titles and actor database modeled as a bipartite graph.

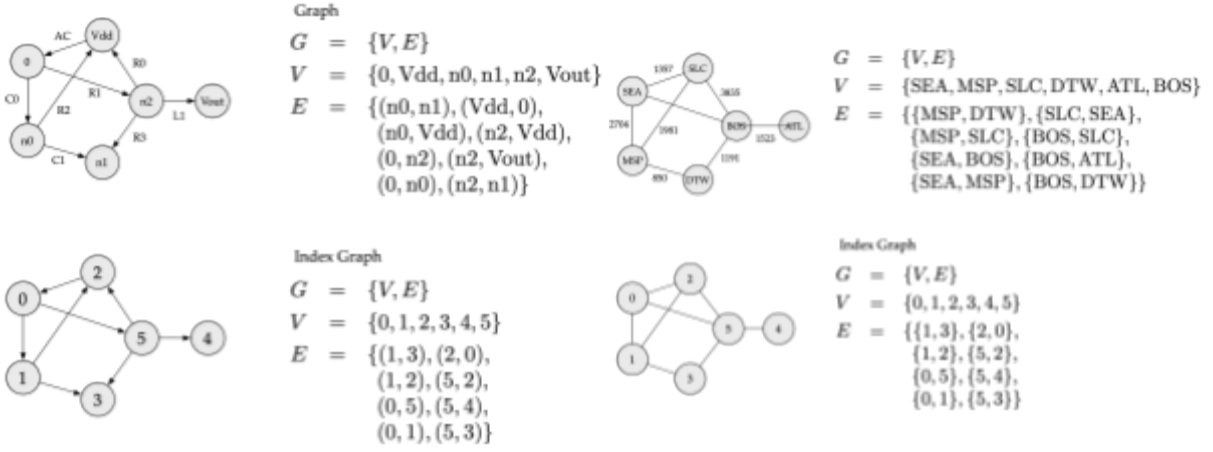
To define algorithms on graphs and to be able to reason about those algorithms, we need to define some representations for graphs—one can't really do very much with abstract sets of vertices and edges. So first we need to define some terminology regarding representations. Various characteristics of these representations are what we use to express algorithms (still abstractly) but when those algorithms are implemented as generic library functions, those characteristics will in turn become the basis for the library's concepts.

One of the fundamental operations in graph algorithms is *traversal*. That is, given a vertex $u$we would like to find the neighbors of u, i.e., all vertices $v$such that the edge (u, v) is in the graph. Then, for each of those edges, we would like to find their neighbors, and so on. The representation that we can define to make this efficient is an *adjacency list.* This is standard terminology for the abstract representation, we aren't going to require that an actual list be used (or any other actual type). Specific type requirements will be defined by the library concepts below.

There is an important transition in going from a graph (as a collection of vertex objects and pairs of vertex objects) to an adjacency list. Implied in using an adjacency list for traversal is that we would like to be able "find the neighbors" efficiently, i.e., in constant time, meaning we need to be able to take a vertex and do a constant time lookup to get all of the neighboring vertices. Then, with what we get back as the neighbors, we

also need to use to look up more neighbors. In short, regardless of what we consider to be the vertices or edges in our graph G, an adjacency list is something that stores indices which can be used to index into itself.

Given a graph $G = (V, E)$, we can define an adjacency-list representation in the following way. Assign to each element of Va unique index from the range $[0, |V|)$ and denote the vertex identified with index i as $V[i]$. We can now define a new graph with the same structure as G, but in terms of the indices in $[0, |V|)$. Let the *index graph* of G be the graph $\overline{G} = (\overline{V}, \overline{E})$, where $\overline{V} = [0, |V|)$and $\overline{E}$ consists of $|E|$pairs of indices from $\overline{V}$, such that a pair (i, j) is in $\overline{E}$ if and only if $(V[i], V[j])$ is in E. Which is all to say, the index graph of $G$is the graph we get by replacing all elements of G with their corresponding indices.

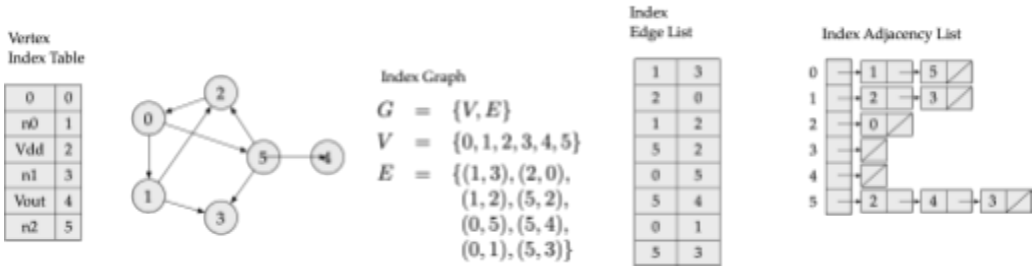


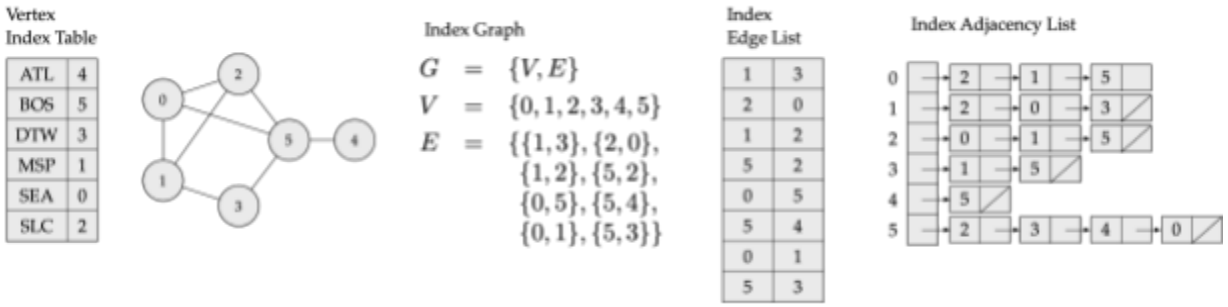
Graph and index graph for circuit example.

Graph and index graph for airline route example.

Of course, we don't need an underlying graph to define what an index graph itself is. We can say that an index graph G= (V, E) is any graph with the property that the vertex set is a set of contiguous indices, with $V = [0, |V| - 1)$. Since an index graph is just a graph, in cases where the context is clear, we may refer to an index graph simply as a graph. We note that an adjacency list can only be defined over an index graph.

An adjacency list of an index graph $G = (V, E)$ is an array Adj of size $|V|$(the array is indexed from 0 to $|V| - 1$). Each entry $Adj[u]$ in the array is a list of all the vertices $v$for which $(u, v)$is contained in E. This structure (an adjacency list of an index graph, or an index adjacency list) is the fundamental structure used by almost all graph algorithms.





From vertex and edges to index adjacency list.

NB (1): Although the standard term for this kind of abstraction is "adjacency list", and although it is often drawn with linked lists as elements, it is not necessary that this abstraction be implemented this way. What is important is that the items that are stored (vertex indices) can be used to index into the adjacency list to obtain other lists of neighbors.

NB (2): The index adjacency list does not store edges per se and therefore the index adjacency list is neither inherently directed nor undirected. Again, for a given vertex u,the container $Adj[u]$ contains the vertex $v$if the edge $(u,\ v)$ is contained in E. This means that for a directed graph with edge (u, v) in , $EAdj[u]$ will contain v. For an undirected graph with edge $(u,\ v)$ is contained in E, $Adj[u]$ will contain v***and*** $Adj[v]$ will contain u. Directedness of the original graph is thus made manifest in the *values* stored in the index adjacency list. But there is nothing about the structure or semantic properties of the adjacency list itself that reflects the directedness (or undirectedness) of the original graph.

Summarizing some points to note about computing with graphs, based on the preceding.

- The vertex set in an index graph is a contiguous range of identifiers $[0,\ |V|)$
- There are no vertices per se in an index graph (we don't materialize V).
- Edges in an index graph are pairs of vertex identifiers.
- An adjacency list does not store vertices. An adjacency list is a structured organization of the relationship information contained in the edges.
- An adjacency list does not store edges.
- Vertex ids obtained from iterating neighbor lists can be used to index back into the adjacency list.
- An adjacency is neither ordered nor unordered (but is over a graph that may be ordered or unordered).

And, finally, a meta-point. Although graphs and graph algorithms are the central focus of a graph library, graphs in some sense are ephemeral. The thing that will be computed on by a graph algorithm is an organized collection of vertex identifiers that were generated to refer back to some real data. Graph texts will often use a phrase like "edges and vertices have properties" but a more correct statement would be "properties have edges and vertices."

Design Proposals

Goals & Background

Although the abstract / theoretical terminology for graphs is that a graph $G = (V, E)$and so on, in fact, graph algorithms are based on efficient concrete representations of the relationships induced on V by E. In particular, graph algorithms require efficient access to this structure. Graph algorithms are interesting because of the structure and so almost all graph algorithms require efficient traversal, and hence require an index adjacency list to operate over. (A few graph algorithms only require access to the edges without the need to traverse the structure and hence only need the index edge list rather than the adjacency list.) Out of 30 algorithms examined from Boost.Graph, only one did not require an index adjacency list. Unfortunately, it is often the case that $G$and Adj will both be referred to as "the graph," and, in fact, it is most often the case that when someone refers to "the graph," they actually mean Adj. It is nonetheless important to maintain the distinction between the graph $G$and Adj, the index adjacency list over it.

As with the STL, our goal with this graph library proposal is to present a systematic organization of the domain of graph algorithms, consisting of a set of minimal type requirements for graph algorithms (aka "concepts" in the general sense), an API for realizing those type requirements in C++, a set of algorithms, and some representative data structures that might not be trivially realizable using composed containers from the standard library.

The library follows the principles of generic programming. That is, it is algorithm-centric and algorithms are realized as function templates with well-defined requirements. These type requirements are organized into a small set of related abstractions such that the algorithms defined with them can operate correctly and efficiently on the widest possible set of input types. In particular, it is essential that the library fully support composition with and among third-party libraries.

Consequently, the library is not built around a single graph type or even a single graph abstraction. We will group requirements together and use graph terminology to name those sets of requirements.

Requirements

Here we motivate the basic requirements for the components of the proposed graph library. C++ specifications of those requirements are given in subsequent sections.

Basic Concepts

Based on our observations of graphs and their representations, an index adjacency structure of a graph (i.e., $ADj(G)$) is a range of ranges, specifically, a random access range of forward ranges. As such, the outer range conforms completely to the requirements of the random_access_range concept and the inner range conforms completely to the requirements of the forward_range concept, including all valid expressions and associated types (such as begin, end, etc.).

The size of a graph is the cardinality of its vertex set. This can be accessed via the size() function on the outer range or via the num_vertices() function on the outer range.

The vertices and edges of a graph are abstract notions that are only implicitly reified in a program via vertex and edge properties (see examples above). However, these properties do need handles in order to be accessed, and the index adjacency structure of the graph needs indices in order to be traversed. Accordingly, we have **vertex keys** and **edge keys**.

The outer range of a graph is indexed with a vertex_key. Indexing into the outer range with a vertex key u returns an inner range, corresponding to the set of edges or vertices reachable from u,i.e., it contains information about all (u, v)in the edge set of the graph. The objects stored by the inner range associated with u can be accessed to obtain the source vertex key u,the target vertex key v, and (optionally) an edge key corresponding to the edge (u, v), or the properties associated with the edge (u, v). The former case is more flexible and allows the actual edge properties to be accessed, but requires an indirection. The latter case may involve copying property values, but avoids the extra indirection. We refer to this type of graph as an **incidence graph**. The neighborhood range associated with a given vertex is accessed using the function **out_edges**.

The following code assumes the inner range stores pairs of vertices, corresponding to the edges leaving each vertex), and would iterate through all of the edges in the graph:

```
using Graph = ....              // graph definition
Graph graph(...);               // adjacency construction
for (auto&& n : graph) {        // for reach inner range
  for (auto&& [u, v] : n) {     //   for each pair in the inner range
    std::cout << "( " << u << ", " << v << " )" << std::endl;
  }
}
```

Note that a graph composed of standard library containers, e.g., std:vector<std::forward_list<std::tuple<int, int>>> could be used in the example above.

For a select few algorithms, we may also need the information about every (v, u), given u. (This is essentially the transpose of (u, v)). A graph that stores information for both (v, u) and (u, v) is called a **bidirectional graph** (which is necessarily also an incidence graph). The "transpose" neighborhood range associated with a given vertex is accessed using the function **in_edges**.

Some graph algorithms require knowing the degree of each vertex (the size of each neighborhood). In that case, the inner range should also a sized range. The degree of each vertex can be accessed with the range function size on the inner range or with the degree() function (which takes a vertex key as argument). The degree must be accessible in constant time.

In rare cases a graph algorithm will require knowing the total number of edges in a graph. If all of the edges are stored in a single sized_range, this can be obtained from the size function on that range. In other cases, the number of edges will need to be maintained as part of the graph. The total number of edges of a graph can be obtained with the num_edges() function (which takes a graph as argument).

Graph Traversal

Traversing a graph is accomplished by iterating over the ranges comprising the graph. The iterator types are those associated with each range type. We refer to the outer range of the graph as a **vertex range** and its corresponding iterator as a **vertex iterator**. An **out edge iterator** is used to iterate over the out edges associated with a vertex in an incidence graph and an **in edge iterator** is used to iterate over the in edges of a bidirectional graph.

An example traversing an incidence graph is shown below.

```
using Graph = ....              // graph definition
Graph graph(...);               // adjacency construction
for (auto i = begin(graph); i != end(graph); ++i) {
  for (auto uv = begin(out_edges(g, i)); uv != end(out_edges(g,i)); ++uv) {
    //  Do something with edge uv
  }
}
```

Vertex and Edge Properties

Construction

Modification

Library Components

Uniform API

The uniform API is the set of types, functions and ranges that can be used for algorithms that support both directed and undirected graphs. It expands the concept of "container" in the original STL by providing a standard set of functions with a way to adapt the data types of the external data structures to those used by this proposal.

An external graph data structure can be adapted by specializing the graph_traits for their data structure and overriding the functions for their graph, allowing the user to use algorithms that use the Uniform API on their data structures.

Graph Traits

graph_traits defines the core data model required for the functions and algorithms in this proposal. It defines a set of value types and ranges for a graph data structure used by the functions and algorithms in this

proposal. While usable as-is, they are typically used by the template type aliases, such as vertex_range_t<G>, edge_range_t<G> and vertex_value_t<G> in algorithms.

graph_traits needs to be specialized for each graph data structure.

```
template <typename G>
struct graph_traits
{
  using graph_type                 = G;
  using graph_value_type           = ...;
  using allocator_type             = ...; // used by containers in the graph

  using vertex_type                = ...;
  using vertex_key_type            = ...;
  using vertex_value_type          = ...;

  using edge_type                  = ...;
  using edge_key_type              = ...; // ordered_pair | unordered_pair
  using edge_value_type            = ...;

  // The following 4 range sets are used by algorithms
  using vertex_range               = ...;
  using const_vertex_range         = ...;

  using vertex_edge_range          = ...;
  using const_vertex_edge_range    = ...;

  using vertex_vertex_range        = ...; // recommended
  using const_vertex_vertex_range  = ...; // recommended

  using edge_range                 = ...; // recommended
  using const_edge_range           = ...; // recommended

  // The following ranges are only for directed graphs
  using vertex_outward_edge_range         = ...; // optional
  using const_vertex_outward_edge_range   = ...; // optional

  using vertex_outward_vertex_range       = ...; // optional
  using const_vertex_outward_vertex_range = ...; // optional

  using vertex_inward_edge_range          = ...; // optional
  using const_vertex_inward_edge_range    = ...; // optional

  using vertex_inward_vertex_range        = ...; // optional
  using const_vertex_inward_vertex_range  = ...; // optional
};
```

Algorithms will use ranges through `const_vertex_vertex_range`, so it's recommended that they support those ranges for broadest usage. The outward and inward ranges are for directed graphs, particularly for

bi-directed graphs; outward and inward ranges are expected to be mapped to their matching Uniform ranges (e.g. vertex_outward_edge_range would be mapped to vertex_edge_range).

If there is no value associated with an object (e.g. for a vertex) then the empty_value struct should be used.

Concepts Definitions

The Concepts definitions in this section are useful for graph algorithms to express the requirements of the algorithm's parameters.

Incidence, Adjacency and Edge-List Graphs

These concepts are used by algorithms to describe the edge range(s) they need. Each includes vertices, a specific organization of edges and related functions. The concepts also include the graph functions that accompany the range and the value types and their functions (e.g. for graph, vertex, edge).

```
template <typename G>
concept vertex_list_graph = requires(G& g) {
    vertex_range<G, vertex_range_t<G>>;
    graph_value_types<G>;
    vertex_value_types<G>;
    { vertices(g) } -> convertible_to<vertex_range_t<G>>;
  };
```

The incidence_graph includes vertex_range, vertex_edge_range and standard functions and properties for the graph, vertex and edge object types.

```
template <typename G>
concept incidence_graph =
  requires(G&                                    g,
           ranges::iterator_t<vertex_range_t<G>> u,
           ranges::iterator_t<vertex_range_t<G>> v,
           vertex_key_t<G>                       ukey,
           vertex_key_t<G>                       vkey) {
    vertex_list_graph<G>;
    incidence_edge_range<G, vertex_edge_range_t<G>>;
    { edges(g,u) }                -> convertible_to<vertex_edge_range_t<G>>;
    { find_vertex_edge(g, u, v) } ->
convertible_to<ranges::iterator_t<vertex_edge_range_t<G>>>;
    { find_vertex_edge(g, ukey, vkey) }
                                  ->
convertible_to<ranges::iterator_t<vertex_edge_range_t<G>>>;
  };
```

The adjacency_graph includes vertex_range, vertex_vertex_range (neighboring vertices) and standard functions and properties for the graph, vertex and edge object types.

```
template <typename G>
concept adjacency_graph =
  requires(G&                                    g,
           ranges::iterator_t<vertex_range_t<G>> u,
           ranges::iterator_t<vertex_range_t<G>> v,
           vertex_key_t<G>                       ukey,
           vertex_key_t<G>                       vkey) {
    vertex_list_graph<G>;
    adjacency_edge_range<G, vertex_vertex_range_t<G>>;
```

```
    { vertices(g,u) }            -> convertible_to<vertex_vertex_range_t<G>>;
    { find_vertex_vertex(g,u,v) }->
convertible_to<ranges::iterator_t<vertex_vertex_range_t<G>>>;
    { find_vertex_vertex(g,ukey,vkey) }
                                 ->
convertible_to<ranges::iterator_t<vertex_vertex_range_t<G>>>;
  };
```

The edge_list_graph includes vertex_range, edge_range (all edges in the graph, independent of the vertices) and standard functions and properties for the graph, vertex and edge object types.

```
template <typename G>
concept edge_list_graph =
  requires(G&                                    g,
           ranges::iterator_t<vertex_range_t<G>> u,
           ranges::iterator_t<vertex_range_t<G>> v,
           vertex_key_t<G>                       ukey,
           vertex_key_t<G>                       vkey) {
    vertex_list_graph<G>;
    basic_edge_range<G, edge_range_t<G>>;
    edge_value_types<G>;
    convertible_to<ranges::range_value_t<edge_range_t<G>>, edge_t<G>>;
    { edge_key(g,uv) }           -> convertible_to<edge_key_t<G>>;
    { edge_value(g,uv) }         -> convertible_to<edge_value_t<G>&>;
    { edges(g) }                 -> convertible_to<edge_range_t<G>>;
    { find_edge(g, u, v) }       -> convertible_to<ranges::iterator_t<edge_range_t<G>>>;
    { find_edge(g, ukey, vkey) } -> convertible_to<ranges::iterator_t<edge_range_t<G>>>;
  };
```

Supporting Concepts

```
template<typename G, typename VR>
concept vertex_range = requires(G&                    g,
                                VR&                   vr,
                                ranges::iterator_t<VR> u,
                                vertex_key_t<G>        ukey) {
    ranges::forward_range<VR>;
    ranges::sized_range<VR>;
    convertible_to<ranges::range_value_t<VR>, vertex_t<G>>;
    { vertex_key(g,u) } -> convertible_to<vertex_key_t<G>>;
    { vertex_value(g,u) } -> convertible_to<vertex_value_t<G>&>;
    { degree(g,u) } -> convertible_to<ranges::range_size_t<VR>>;
    { find_vertex(g,ukey) } -> convertible_to<vertex_iterator_t<G>>;
    { contains_vertex(g, ukey) } -> convertible_to<bool>;
    { empty(vr) } -> convertible_to<bool>;
  };

template<typename G, typename R>
concept incidence_edge_range = requires(G& g, R& r, ranges::iterator_t<R> uv) {
    basic_edge_range<G,R>;
    ranges::sized_range<R>;
    edge_value_types<G>;
    convertible_to<ranges::range_value_t<R>, edge_t<G>>;
    { edge_key(g,uv) }   -> same_as<edge_key_t<G>>;
    { edge_value(g,uv) } -> same_as<edge_value_t<G>&>;
  };
```

```
template<typename G, typename R>
concept adjacency_edge_range = requires(R& r) {
    basic_edge_range<G,R>;
    ranges::sized_range<R>;
    convertible_to<ranges::range_value_t<R>, vertex_t<G>>;
  };

template<typename G, typename R>
concept basic_edge_range =
  requires(G&                                    g,
           R&                                    r,
           ranges::iterator_t<R>                 uv,
           ranges::iterator_t<vertex_range_t<G>> u,
           ranges::iterator_t<vertex_range_t<G>> src,
           vertex_key_t<G>                       src_key) {
    ranges::forward_range<R>;
    { vertex(g, uv, src) } -> convertible_to<ranges::iterator_t<vertex_range_t<G>>>;
    { vertex_key(g, uv, src_key) } -> convertible_to<ranges::iterator_t<vertex_range_t<G>>>;
    { target_vertex(g, uv) } -> convertible_to<ranges::iterator_t<vertex_range_t<G>>>;
    { target_vertex_key(g, uv) } -> convertible_to<vertex_key_t<G>>;
    { source_vertex(g, uv) } -> convertible_to<ranges::iterator_t<vertex_range_t<G>>>;
    { source_vertex_key(g, uv) } -> convertible_to<vertex_key_t<G>>;
    { empty(r) } -> convertible_to<bool>;
  };

template<typename G>
concept graph_value_types = requires(G& g1, G& g2) {
    graph_traits<G>::graph_type;
    graph_traits<G>::graph_value_type;
    graph_traits<G>::allocator_type;
    convertible_to<G, typename graph_traits<G>::graph_type>;
    semiregular<graph_value_t<G>>;
    { graph_value(g1) } -> convertible_to<graph_value_t<G>&>;
    { swap(g1,g2) };
  };

template<typename G>
concept vertex_value_types = requires(G& g) {
    graph_traits<G>::vertex_type;
    graph_traits<G>::vertex_key_type;
    graph_traits<G>::vertex_value_type;
    semiregular<vertex_value_t<G>>;
  };

template<typename G>
concept edge_value_types = requires(G& g) {
    graph_traits<G>::edge_type;
    graph_traits<G>::edge_key_type;
    graph_traits<G>::edge_value_type;
    graph_traits<G>::vertex_key_type;
    semiregular<edge_value_t<G>>;

    // edge_key_type is ordered_pair<K,K> | unordered_pair<K,K>
    derived_from<edge_key_t<G>, ordered_pair<vertex_key_t<G>,vertex_key_t<G>>> ||
    derived_from<edge_key_t<G>, unordered_pair<vertex_key_t<G>,vertex_key_t<G>>>;
  };
```

Adjacency Matrix

The adjacency_matrix defines additional edge look-up functions that are assumed to be O(1). It is intended to augment the previous concepts for incidence_graph, adjacency_graph or edge_list_graph.

```
template <typename G>
concept adjacency_matrix =
  requires(ranges::iterator_t<vertex_range_t<G>> u,
           ranges::iterator_t<vertex_range_t<G>> v,
           vertex_key_t<G>                       ukey,
           vertex_key_t<G>                       vkey) {
    { contains_edge(u,v) } -> convertible_to<bool>;
    { contains_edge(ukey,vkey) } -> convertible_to<bool>;
  };
```

Directed and Undirected

The distinction between directed and undirected graphs is whether the vertices on an edge are ordered or unordered - ordered vertices define a directed graph and unordered vertices define an undirected graph. The edge key type is used to define orderedness by the use of the new ordered_pair and unorderd_pair structs, described in the Supporting Data Structures section. This keeps the design simple and the rest of the design is not affected.

```
template <typename G>
concept directed =
      is_base_of_v<ordered_pair<typename graph_traits<G>::vertex_key_type,
                                typename graph_traits<G>::vertex_key_type>,
                   typename graph_traits<G>::edge_key_type>;

template <typename G>
concept undirected =
      is_base_of_v<unordered_pair<typename graph_traits<G>::vertex_key_type,
                                  typename graph_traits<G>::vertex_key_type>,
                   typename graph_traits<G>::edge_key_type>;

template <typename G>
concept directed_or_undirected = directed<G> || undirected<G>;
```

Mutable Graphs

```
template <typename G>
concept incremental_vertex_graph =
  requires(G&                    g,
           vertex_value_t<G>&& val) {
    { create_vertex(g) } -> convertible_to<optional<ranges::iterator_t<vertex_range_t<G>>>>;
    { create_vertex(g,val) } ->
convertible_to<optional<ranges::iterator_t<vertex_range_t<G>>>>;
    { create_vertex(g,move(val)) }
      -> convertible_to<optional<ranges::iterator_t<vertex_range_t<G>>>>;
    //{create_vertices(g,vrng,vvalue_fnc)};
  };

template <typename G>
concept decremental_vertex_graph =
  requires(G&                                    g,
```

```
          ranges::iterator_t<vertex_range_t<G>> u,
          vertex_key_t<G>                      ukey,
          vertex_range_t<G>&                   r) {
    { erase_vertex(g, u) };
    { erase_vertex(g, ukey) };
    { erase_vertices(g, r) };
  };

template <typename G>
concept dynamic_vertex_graph = incremental_vertex_graph<G> && decremental_vertex_graph<G>;
template <typename G>
concept static_vertex_graph = !incremental_vertex_graph<G> && !decremental_vertex_graph<G>;

template <typename G>
concept incremental_edge_graph =
  requires(G&                                   g,
          ranges::iterator_t<vertex_range_t<G>> u,
          ranges::iterator_t<vertex_range_t<G>> v,
          vertex_key_t<G>                       ukey,
          vertex_key_t<G>                       vkey,
          edge_value_t<G>&&                     val) {
    {create_edge(g,u,v)} -> convertible_to<optional<vertex_edge_iterator_t<G>>>;
    {create_edge(g,u,v,val)} -> convertible_to<optional<vertex_edge_iterator_t<G>>>;
    {create_edge(g,u,v,move(val))} -> convertible_to<optional<vertex_edge_iterator_t<G>>>;

    {create_edge(g,ukey,vkey)} -> convertible_to<optional<vertex_edge_iterator_t<G>>>;
    {create_edge(g,ukey,vkey,val)} -> convertible_to<optional<vertex_edge_iterator_t<G>>>;
    {create_edge(g,ukey,vkey,move(val))} ->
convertible_to<optional<vertex_edge_iterator_t<G>>>;

    //{create_edges(g,erng,ekey_fnc,evalue_fnc)};
    //{create_edges(g,erng,vrng,ekey_fnc,evalue_fnc,vvalue_fnc)};
  };

template <typename G, typename ER>
concept decremental_edge_range =
  requires(G&                                   g,
          ER                                    er,
          ranges::iterator_t<ER>                uv) {
    is_same_v<ranges::range_value_t<ER>, edge_value_t<G>>;
    { erase_edge(g,uv) } -> std::same_as<optional<ranges::iterator_t<ER>>>;
    { erase_edges(g,er) };
  };

template <typename G>
concept decremental_edge_graph =
  (decremental_edge_range<G,vertex_edge_range_t<G>> ||
    decremental_edge_range<G,vertex_vertex_range_t<G>> ||
    decremental_edge_range<G,vertex_edge_range_t<G>>) &&
  requires(G&                                   g,
          ranges::iterator_t<vertex_range_t<G>> u,
          ranges::iterator_t<vertex_range_t<G>> v,
          vertex_key_t<G>                       ukey,
          vertex_key_t<G>                       vkey) {
    { erase_edge(g,u,v) };
    { erase_edge(g,ukey,vkey) };
  };
```

```
template <typename G>
concept dynamic_edge_graph = incremental_edge_graph<G> && decremental_edge_graph<G>;
template <typename G>
concept static_edge_graph = !incremental_edge_graph<G> && !decremental_edge_graph<G>;

template <typename G>
concept dynamic_graph = dynamic_vertex_graph<G> && dynamic_edge_graph<G>;
template <typename G>
concept static_graph = static_vertex_graph<G> && static_edge_graph<G>;
```

Paths and Cycles

```
template <typename G, typename Path>
concept vertex_path =
          ranges::forward_range<Path> &&
          convertible_to<ranges::range_value_t<Path>, vertex_iterator_t<G>>;

template <typename G, typename Path>
concept edge_path =
          ranges::forward_range<Path> &&
          (const_edge_iterator<G,ranges::range_value_t<Path>> ||
          edge_iterator<G,ranges::range_value_t<Path>>);

template <typename G, typename Path>
concept vertex_cycle =
          ranges::forward_range<Path> &&
          convertible_to<ranges::range_value_t<Path>, vertex_iterator_t<G>>;

template <typename G, typename Path>
concept edge_cycle =
          ranges::forward_range<Path> &&
          (const_edge_iterator<G,ranges::range_value_t<Path>> ||
          edge_iterator<G,ranges::range_value_t<Path>>);
```

Other Concepts

The edge_iterator & adjacency_iterator concepts are used to consolidate “edge” property functions into a single definition when all that is different is the edge iterator type.

```
template <typename G, typename EI>
concept edge_iterator =
  forward_iterator<EI> &&
  convertible_to<iter_value_t<EI>,
                 add_lvalue_reference<
                    typename graph_traits<G>::edge_type>>;
template <typename G, typename EI>
concept const_edge_iterator =
  forward_iterator<EI> &&
  convertible_to<iter_value_t<EI>,
                 add_const<add_lvalue_reference<
                    typename graph_traits<G>::edge_type>>>;
```

```
template <typename G, typename EI>
concept adjacency_iterator =
  forward_iterator<EI> &&
  convertible_to<iter_value_t<EI>,
                 add_lvalue_reference<
                    typename graph_traits<G>::vertex_type>>;
template <typename G, typename EI>
concept const_adjacency_iterator =
  forward_iterator<EI> &&
  convertible_to<iter_value_t<EI>,
                 add_const<add_lvalue_reference<
                    typename graph_traits<G>::vertex_type>>>;
```

The value extractors are used to validate functions that extract a key or value from a range value in graph constructors, and in the create_edges and create_vertices functions.

```
template <typename VRng, typename VValueFnc>
concept  vertex_value_extractor = ranges::input_range<VRng> &&
                                  invocable<VValueFnc, typename VRng::value_type>;

template <typename ERng, typename EKeyFnc, typename EValueFnc>
concept edge_value_extractor = ranges::forward_range<ERng> &&
                               invocable<EKeyFnc, typename ERng::value_type> &&
                               invocable<EValueFnc, typename ERng::value_type>;
```

Types

Graph Value Types

```
template <directed_or_undirected G>
using graph_t = typename graph_traits<G>::graph_type;

template <directed_or_undirected G>
using graph_value_t = typename graph_traits<G>::graph_value_type;

template <directed_or_undirected G>
using graph_allocator_t = typename graph_traits<G>::allocator_type;
```

Vertex Value Types

```
template <directed_or_undirected G>
using vertex_t = typename graph_traits<G>::vertex_type;

template <directed_or_undirected G>
using vertex_key_t = typename graph_traits<G>::vertex_key_type;

template <directed_or_undirected G>
using vertex_value_t = typename graph_traits<G>::vertex_value_type;
```

Edge Value Types

```
template <directed_or_undirected G>
```

```
using edge_t = typename graph_traits<G>::edge_type;

template <directed_or_undirected G>
using edge_key_t = typename graph_traits<G>::edge_key_type;

template <directed_or_undirected G>
using edge_value_t = typename graph_traits<G>::edge_value_type;
```

Graph-Vertices Range Types

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

template <typename G>
using vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_range>;
template <directed_or_undirected G>
using const_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_range>;
template <typename G>
using vertex_sentinel_t = ranges::sentinel_t<typename graph_traits<G>::vertex_range>;

template <directed_or_undirected G>
using vertex_size_t = ranges::range_size_t<typename graph_traits<G>::vertex_range>;
template <directed_or_undirected G>
using vertex_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::vertex_range>;
```

Graph-Edges Range Types

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

template <directed_or_undirected G>
using edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::edge_range>;
template <directed_or_undirected G>
using const_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_edge_range>;
template <directed_or_undirected G>
using edge_sentinel_t = ranges::sentinel_t<typename graph_traits<G>::edge_range>;

template <directed_or_undirected G>
using edge_size_t = typename graph_traits<G>::edge_size_type;
template <directed_or_undirected G>
using edge_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::edge_range>;
```

Vertex-Edges Range Type

```
template <directed_or_undirected G>
using vertex_edge_range_t = typename graph_traits<G>::vertex_edge_range;
template <directed_or_undirected G>
using const_vertex_edge_range_t = typename graph_traits<G>::const_vertex_edge_range;

template <directed_or_undirected G>
using vertex_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_edge_range>;
template <directed_or_undirected G>
using const_vertex_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_edge_range>;
template <directed_or_undirected G>
using vertex_edge_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_edge_range>;

template <directed_or_undirected G>
using vertex_edge_size_t = typename graph_traits<G>::vertex_edge_size_type;
template <directed_or_undirected G>
using vertex_edge_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::vertex_edge_range>;
```

Vertex-Vertices Range Type

```
template <directed_or_undirected G>
using vertex_vertex_range_t = typename graph_traits<G>::vertex_vertex_range;
template <directed_or_undirected G>
using const_vertex_vertex_range_t =
      typename graph_traits<G>::const_vertex_vertex_range;

template <directed_or_undirected G>
using vertex_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_vertex_range>;
template <directed_or_undirected G>
using const_vertex_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_vertex_range>;
template <directed_or_undirected G>
using vertex_vertex_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_vertex_range>;

template <directed_or_undirected G>
using vertex_vertex_size_t = typename graph_traits<G>::vertex_vertex_size_type;
template <directed_or_undirected G>
using vertex_vertex_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::vertex_vertex_range>;
```

Functions

Common Container Free Functions

The common range functions are supported in all ranges that are supported by the graph and include **begin**(r), **end**(r), **cbegin**(r), **cend**(r), **size**(r), **ssize**(r), **empty**(r). For instance, begin(vertices(g)), and size(vertices(g)).

Graph Properties

```
template <directed_or_undirected G>
constexpr auto graph_value(G& g) -> graph_value_t<G>&;
template <directed_or_undirected G>
constexpr auto graph_value(G const& g) -> const graph_value_t<G>&;

template <directed_or_undirected G>
constexpr auto graph_allocator(const G& g) noexcept -> graph_allocator_t<G>;
```

The “contains” functions are only required when they can return results in O(1). The contains_edge functions are a requirement for the adjacency_matrix concept.

```
template <directed_or_undirected G>
constexpr bool contains_vertex(const G& g, vertex_key_t<G> ukey) noexcept;

template <directed_or_undirected G>
constexpr bool contains_edge(const G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey);
template <directed_or_undirected G>
constexpr bool contains_edge(const G& g, const_vertex_iterator_t<G> u,
const_vertex_iterator_t<G> v);
```

Vertex Properties

```
template <directed_or_undirected G>
constexpr auto vertex_key(const G&, const_vertex_iterator_t<G> u) -> vertex_key_t<G>;

template <directed_or_undirected G>
constexpr auto vertex_value(G& g, vertex_iterator_t<G> u)
    -> vertex_value_t<G>&;
template <directed_or_undirected G>
constexpr auto vertex_value(const G& g, const_vertex_iterator_t<G> u)
    -> const vertex_value_t<G>&;

template <directed_or_undirected G>
constexpr auto degree(const G& g, const_vertex_iterator_t<G> u) noexcept
    -> vertex_edge_size_t<G>;
```

Edge Properties

```
template <directed_or_undirected G>
constexpr auto edge_key(const G& g, const_vertex_iterator_t<G> u,
                                    const_vertex_iterator_t<G> v) -> edge_key_t<G>;
```

```
template <directed_or_undirected G>
constexpr auto edge_key(const G& g, vertex_key_t<G> ukey,
                                    vertex_key_t<G> vkey) -> edge_key_t<G>;

template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto edge_key(const G& g, EI uv) -> edge_key_t<G>;

template <directed_or_undirected G, typename EI>
  requires edge_iterator<G,EI>
constexpr auto edge_value(G& g, EI uv) -> edge_value_t<G>&;

template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto edge_value(const G& g, EI uv) -> const edge_value_t<G>&;


template <directed_or_undirected G, typename I>
  requires (edge_iterator<G,I> || adjacency_iterator<G,I>)
constexpr auto vertex(G& g, I uv, const_vertex_iterator_t<G> source)
  -> vertex_iterator_t<G>;

template <directed_or_undirected G, typename EI>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex(const G&, EI uv, const_vertex_iterator_t<G> source)
  -> const_vertex_iterator_t<G>;

template <directed G, typename EI>
  requires (edge_iterator<G,I> || adjacency_iterator<G,I>)
constexpr auto vertex(G& g, EI uv) -> vertex_iterator_t<G>;

template <directed G, typename EI>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex(const G&, EI uv) -> const_vertex_iterator_t<G>;


template <directed_or_undirected G, typename I>
  requires (edge_iterator<G,I> || adjacency_iterator<G,I>)
constexpr auto vertex(G& g, I uv, vertex_key_t<G> source_key)
  -> vertex_iterator_t<G>;

template <directed_or_undirected G, typename I>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex(const G&, I uv, vertex_key_t<G> source_key)
  -> const_vertex_iterator_t<G>;

template <directed G, typename I>
  requires (edge_iterator<G,I> || adjacency_iterator<G,I>)
constexpr auto vertex(G& g, I uv) -> vertex_iterator_t<G>;
```

```
template <directed G, typename I>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex(const G&, I uv) -> const_vertex_iterator_t<G>;


template <directed_or_undirected G, typename I>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex_key(const G&, I uv, const_vertex_iterator_t<G> source)
  -> vertex_key_t<G>;

template <directed_or_undirected G, typename I>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex_key(const G&, I uv, vertex_key_t<G> source_key)
  -> vertex_key_t<G>;

template <directed G, typename I>
  requires (const_edge_iterator<G,I> || const_adjacency_iterator<G,I>)
constexpr auto vertex_key(const G&, I uv) -> vertex_key_t<G>;


template <directed_or_undirected G, typename EI>
  requires edge_iterator<G,EI>
constexpr auto target_vertex(G& g, EI uv) -> vertex_iterator_t<G>;
template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto target_vertex(const G&, EI uv) -> const_vertex_iterator_t<G>;

template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto target_vertex_key(const G&, EI uv) -> vertex_key_t<G>;

template <directed_or_undirected G, typename EI>
  requires edge_iterator<G,EI>
constexpr auto source_vertex(G& g, EI uv) -> vertex_iterator_t<G>;
template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto source_vertex(const G&, EI uv) -> const_vertex_iterator_t<G>;

template <directed_or_undirected G, typename EI>
  requires const_edge_iterator<G,EI>
constexpr auto source_vertex_key(const G&, EI uv) -> vertex_key_t<G>;
```

Graph-Vertex Range Functions

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

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

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

Graph-Edge Range Functions

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

template <directed_or_undirected G>
constexpr auto find_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> edge_iterator_t<G>;
template <directed_or_undirected G>
constexpr auto find_edge(const G&,
                         const_vertex_iterator_t<G> u,
                         const_vertex_iterator_t<G> v) -> const_edge_iterator_t<G>;

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

Vertex-Edge Range Functions

```
template <directed_or_undirected G>
constexpr auto edges(G& g, vertex_iterator_t<G>& u) -> vertex_edge_range_t<G>;
template <directed_or_undirected G>
constexpr auto edges(const G&, const_vertex_iterator_t<G>& u)
      -> const_vertex_edge_range_t<G>;

template <directed_or_undirected G>
constexpr auto find_vertex_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_edge_iterator_t<G>;
template <directed_or_undirected G>
constexpr auto
find_vertex_edge(const G&, const_vertex_iterator_t<G> u, const_vertex_iterator_t<G> v)
      -> const_vertex_edge_iterator_t<G>;

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

Vertex-Vertex Range Functions

```
template <directed_or_undirected G>
constexpr auto vertices(G& g, vertex_iterator_t<G> u) -> vertex_vertex_range_t<G>;
```

```
template <directed_or_undirected G>
constexpr auto vertices(const G&, const_vertex_iterator_t<G> u)
      -> const_vertex_vertex_range_t<G>;

template <directed_or_undirected G>
constexpr auto
find_vertex_vertex(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_vertex_iterator_t<G>;
template <directed_or_undirected G>
constexpr auto find_vertex_vertex(const G&,
                                  const_vertex_iterator_t<G> u,
                                  const_vertex_iterator_t<G> v)
      -> const_vertex_vertex_iterator_t<G>;

template <directed_or_undirected G>
constexpr auto find_vertex_vertex(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> vertex_vertex_iterator_t<G>;
template <directed_or_undirected G>
constexpr auto
find_vertex_vertex(const G&, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> const_vertex_vertex_iterator_t<G>;
```

Modifying Graph Functions

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

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

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

Modifying Vertex Functions

```
template <directed_or_undirected G>
constexpr auto create_vertex(G& g) -> optional<vertex_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto create_vertex(G& g, const vertex_value_t<G>&)
      -> optional<vertex_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto create_vertex(G& g, vertex_value_t<G>&&)
      -> optional<vertex_iterator_t<G>>;

template <directed_or_undirected G, typename VRng, typename VValueFnc>
  requires vertex_value_extractor<VRng, VValueFnc>
void create_vertices(G& g, const VRng& vrng, const VValueFnc& vvalue_fnc);


template <directed_or_undirected G>
constexpr void erase_vertex(G& g, vertex_iterator_t<G>);
```

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

Modifying Edge Functions

```
template <directed_or_undirected G>
constexpr auto create_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> optional<vertex_edge_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto
create_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v, edge_value_t<G>&)
      -> optional<vertex_edge_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto
create_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v, edge_value_t<G>&&)
      -> optional<vertex_edge_iterator_t<G>>;

template <directed_or_undirected G>
constexpr auto create_edge(G& g, vertex_key_t<G>, vertex_key_t<G>)
      -> optional<vertex_edge_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto create_edge(G& g, vertex_key_t<G>, vertex_key_t<G>, edge_value_t<G>&)
      -> optional<vertex_edge_iterator_t<G>>;
template <directed_or_undirected G>
constexpr auto create_edge(G& g, vertex_key_t<G>, vertex_key_t<G>, edge_value_t<G>&&)
      -> optional<vertex_edge_iterator_t<G>>;

// clang-format off
template <directed_or_undirected G,
          typename               ERng,
          typename               EKeyFnc,
          typename               EValueFnc>
  requires edge_value_extractor<ERng, EKeyFnc, EValueFnc>
void create_edges(G&                 g,
                  const ERng&      rng,
                  const EKeyFnc&   ekey_fnc,
                  const EValueFnc& evalue_fnc);

template <directed_or_undirected G,
          typename               ERng,
          typename               EKeyFnc,
          typename               EValueFnc,
          typename               VRng,
          typename               VValueFnc>
  requires edge_value_extractor<ERng, EKeyFnc, EValueFnc> &&
           vertex_value_extractor<VRng, VValueFnc>
void create_edges(G&                 g,
                  const ERng&      erng,
                  const VRng&      vrng,
```

```
                  const EKeyFnc&   ekey_fnc,
                  const EValueFnc& evalue_fnc,
                  const VValueFnc& vvalue_fnc);

template <directed_or_undirected G>
constexpr void erase_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v);
template <directed_or_undirected G>
constexpr void erase_edge(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey);

template <directed_or_undirected G, typename EI>
  requires edge_iterator<G,EI>
constexpr auto erase_edge(G& g, EI uv) -> EI;

template <directed_or_undirected G, typename ER>
  requires edge_iterator<G, ranges::iterator_t<ER>>
constexpr void erase_edges(G& g, ER uv_rng);

template <directed_or_undirected G>
constexpr void clear_edges(G& g, vertex_iterator_t<G>);
```

Supporting Data Structures

ordered_pair and unordered_pair are derived from pair and are used to distinguish between a directed and undirected graph, respectively. They are in the std namespace.

ToDo: The current definitions are missing the following features to make it fully compatible with pair<T1,T2> and need to be added:

- `tuple_size<ordered_pair<T1,T2>> & tuple_size<unordered_pair<T1,T2>>`
- `tuple_element<orderd_pair<T1,T2>> & tuple_element<unorderd_pair<T1,T2>>`
- `get(ordered_pair<T1,T2>) & get(unordered_pair<T1,T2>)`
- `Conditionally explicit constructors`
- `piecewise_construct_t constructor`

ordered_pair

```
template <typename T1, typename T2>
struct ordered_pair : public pair<T1, T2> {
  using base_t      = pair<T1, T2>;
  using first_type  = typename base_t::first_type;
  using second_type = typename base_t::second_type;

  constexpr ordered_pair()                         = default;
  constexpr ordered_pair(const ordered_pair&)      = default;
  constexpr ordered_pair(ordered_pair&&) noexcept = default;
  //constexpr ~ordered_pair() = default; // dtor not defined in pair<>

  template <typename U1, typename U2>
  constexpr ordered_pair(const U1& x, const U2& y);

  template <typename U1, typename U2>
```

```
  constexpr ordered_pair(U1&& x, U2&& y) noexcept;

  template <typename U1, typename U2>
  constexpr explicit ordered_pair(const pair<U1, U2>& other);

  template <typename U1, typename U2>
  constexpr explicit ordered_pair(pair<U1, U2>&& other);

  constexpr ordered_pair& operator=(const ordered_pair& other);
  constexpr ordered_pair& operator=(ordered_pair&& other) noexcept;

  constexpr void
  swap(ordered_pair& other) noexcept(is_nothrow_swappable_v<first_type>&&
                                     is_nothrow_swappable_v<second_type>);
};

template <typename T1, typename T2>
auto operator<=>(const ordered_pair<T1, T2>& v1,
                 const ordered_pair<T1, T2>& v2);
template <typename T1, typename T2>
auto operator<=>(const ordered_pair<T1, T2>& v1, const pair<T1, T2>& v2);
template <typename T1, typename T2>
auto operator<=>(const pair<T1, T2>& v1, const ordered_pair<T1, T2>& v2);

template <typename T1, typename T2>
constexpr void
swap(ordered_pair<T1, T2>& v1, ordered_pair<T1, T2>& v2) noexcept(
      is_nothrow_swappable_v<typename ordered_pair<T1, T2>::first_type>&&
            is_nothrow_swappable_v<typename ordered_pair<T1, T2>::second_type>);

template <class T1, class T2>
constexpr auto make_ordered_pair(T1&& v1, T2&& v2);
```

unordered_pair

```
template <typename T1, typename T2>
struct unordered_pair : public pair<T1, T2> {
  using base_t      = pair<T1, T2>;
  using first_type  = typename base_t::first_type;
  using second_type = typename base_t::second_type;

  constexpr unordered_pair()                           = default;
  constexpr unordered_pair(const unordered_pair&)      = default;
  constexpr unordered_pair(unordered_pair&&) noexcept = default;
  //constexpr ~unordered_pair() = default; // destructor not defined in pair<>

  template <typename U1, typename U2>
  constexpr unordered_pair(const U1& x, const U2& y);

  template <typename U1, typename U2>
  constexpr unordered_pair(U1&& x, U2&& y) noexcept;
```

```
  template <typename U1, typename U2>
  constexpr explicit unordered_pair(const pair<U1, U2>& other);

  template <typename U1, typename U2>
  constexpr explicit unordered_pair(pair<U1, U2>&& other);

  constexpr unordered_pair& operator=(const unordered_pair& other);

  constexpr unordered_pair& operator=(unordered_pair&& other) noexcept;

  constexpr void swap(unordered_pair& other) noexcept(
        is_nothrow_swappable_v<first_type>&&
              is_nothrow_swappable_v<second_type>);
};

template <typename T1, typename T2>
auto operator<=>(const unordered_pair<T1, T2>& v1,
                 const unordered_pair<T1, T2>& v2);
template <typename T1, typename T2>
auto operator<=>(const unordered_pair<T1, T2>& v1, const pair<T1, T2>& v2);
template <typename T1, typename T2>
auto operator<=>(const pair<T1, T2>& v1, const unordered_pair<T1, T2>& v2);

template <typename T1, typename T2>
constexpr void
swap(unordered_pair<T1, T2>& v1, unordered_pair<T1, T2>& v2) noexcept(
      is_nothrow_swappable_v<typename unordered_pair<T1, T2>::first_type>&&
            is_nothrow_swappable_v<
                  typename unordered_pair<T1, T2>::second_type>);

template <class T1, class T2>
constexpr auto make_unordered_pair(T1&& v1, T2&& v2);
```

Adapting to External Graphs

The functions and iterators in this paper can be used by external data structures by specializing a graph_traits structure and matching Uniform API functions for the external graph. They should be created in the std namespace, following the guidelines in Extending the namespace std.

Directed API

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

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

Concepts

The outward and inward graph concepts are for directed graphs.

```
template<typename G>
concept outward_incidence_graph =
  vertex_list_graph<G> &&
  incidence_edge_range<G, vertex_outward_edge_range_t<G>> &&
  edge_value_types<G> &&
  requires(G&                                                g,
           ranges::iterator_t<vertex_outward_edge_range_t<G>> u,
           ranges::iterator_t<vertex_outward_edge_range_t<G>> v,
           vertex_key_t<G>                                   ukey,
           vertex_key_t<G>                                   vkey) {
    { vertex_outward_edges(g,u) }
      -> convertible_to<vertex_outward_edge_range_t<G>>;
    { find_vertex_outward_edge(g, u, v) }
      -> convertible_to<ranges::iterator_t<vertex_outward_edge_range_t<G>>>;
    { find_vertex_outward_edge(g, ukey, vkey) }
      -> convertible_to<ranges::iterator_t<vertex_outward_edge_range_t<G>>>;
  };

template <typename G>
concept outward_adjacency_graph =
  vertex_list_graph<G> &&
  adjacency_edge_range<G, vertex_outward_vertex_range_t<G>> &&
  edge_value_types<G> &&
  requires(G&                                                  g,
           ranges::iterator_t<vertex_outward_vertex_range_t<G>> u,
           ranges::iterator_t<vertex_outward_vertex_range_t<G>> v,
           vertex_key_t<G>                                     ukey,
           vertex_key_t<G>                                     vkey) {
    { vertex_outward_edges(g,u) }
      -> convertible_to<vertex_outward_vertex_range_t<G>>;
    { find_vertex_outward_edge(g, u, v) }
      -> convertible_to<ranges::iterator_t<vertex_outward_vertex_range_t<G>>>;
    { find_vertex_outward_edge(g, ukey, vkey) }
      -> convertible_to<ranges::iterator_t<vertex_outward_vertex_range_t<G>>>;
  };

template<typename G>
concept inward_incidence_graph =
  vertex_list_graph<G> &&
  incidence_edge_range<G, vertex_inward_edge_range_t<G>> &&
  edge_value_types<G> &&
  requires(G&                                               g,
           ranges::iterator_t<vertex_inward_edge_range_t<G>> u,
           ranges::iterator_t<vertex_inward_edge_range_t<G>> v,
           vertex_key_t<G>                                  ukey,
           vertex_key_t<G>                                  vkey) {
    { vertex_inward_edges(g,u) }
      -> convertible_to<vertex_inward_edge_range_t<G>>;
    { find_vertex_inward_edge(g, u, v) }
      -> convertible_to<ranges::iterator_t<vertex_inward_edge_range_t<G>>>;
    { find_vertex_inward_edge(g, ukey, vkey) }
```

```
      -> convertible_to<ranges::iterator_t<vertex_inward_edge_range_t<G>>>;
  };

template <typename G>
concept inward_adjacency_graph =
  vertex_list_graph<G> &&
  adjacency_edge_range<G, vertex_inward_vertex_range_t<G>> &&
  edge_value_types<G> &&
  requires(G&                                                g,
           ranges::iterator_t<vertex_inward_vertex_range_t<G>> u,
           ranges::iterator_t<vertex_inward_vertex_range_t<G>> v,
           vertex_key_t<G>                                   ukey,
           vertex_key_t<G>                                   vkey) {
    { vertex_inward_edges(g,u) }
      -> convertible_to<vertex_inward_vertex_range_t<G>>;
    { find_vertex_inward_edge(g, u, v) }
      -> convertible_to<ranges::iterator_t<vertex_inward_vertex_range_t<G>>>;
    { find_vertex_inward_edge(g, ukey, vkey) }
      -> convertible_to<ranges::iterator_t<vertex_inward_vertex_range_t<G>>>;
  }
```

Outward Range Types

```
template <directed G>
using vertex_outward_edge_range_t =
      typename graph_traits<G>::vertex_outward_edge_range;
template <directed G>
using const_vertex_outward_edge_range_t =
      typename graph_traits<G>::const_vertex_outward_edge_range;

template <directed G>
using vertex_outward_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_outward_edge_range>;
template <directed G>
using const_vertex_outward_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_outward_edge_range>;
template <directed G>
using vertex_outward_edge_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_outward_edge_range>;

template <directed G>
using vertex_outward_edge_size_t =
      ranges::range_size_t<typename graph_traits<G>::vertex_outward_edge_range>;
template <directed G>
using vertex_outward_edge_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::vertex_outward_edge_range>;


template <directed G>
using vertex_outward_vertex_range_t =
      typename graph_traits<G>::vertex_outward_vertex_range;
template <directed G>
using const_vertex_outward_vertex_range_t =
```

```
      typename graph_traits<G>::const_vertex_outward_vertex_range;

template <directed G>
using vertex_outward_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_outward_vertex_range>;
template <directed G>
using const_vertex_outward_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_outward_vertex_range>;
template <directed G>
using vertex_outward_vertex_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_outward_vertex_range>;

template <directed G>
using vertex_outward_vertex_size_t =
      ranges::range_size_t<typename graph_traits<G>::vertex_outward_vertex_range>;
template <directed G>
using vertex_outward_vertex_difference_t = ranges::range_difference_t<
      typename graph_traits<G>::vertex_outward_vertex_range>;
```

Outward Functions

Common Functions

```
template <directed G>
constexpr auto outward_degree(const G& g, const_vertex_iterator_t<G> u) noexcept
      -> vertex_outward_edge_size_t<G>;
```

Vertex-Outward-Edge Functions

```
template <directed G>
constexpr auto outward_edges(G& g, vertex_iterator_t<G> u)
      -> vertex_outward_edge_range_t<G>;
template <directed G>
constexpr auto outward_edges(const G&, const_vertex_iterator_t<G> u)
      -> const_vertex_outward_edge_range_t<G>;

template <directed G>
constexpr auto find_outward_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_outward_edge_iterator_t<G>;
template <directed G>
constexpr auto find_outward_edge(const G&,
                                 const_vertex_iterator_t<G> u,
                                 const_vertex_iterator_t<G> v)
      -> const_vertex_outward_edge_iterator_t<G>;

template <directed G>
constexpr auto find_outward_edge(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> vertex_outward_edge_iterator_t<G>;
template <directed G>
constexpr auto find_outward_edge(const G&, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
```

```
      -> const_vertex_outward_edge_iterator_t<G>;
```

Vertex-Outward-Vertex Functions

```
template <directed G>
constexpr auto outward_vertices(G& g, vertex_iterator_t<G> u)
      -> vertex_outward_vertex_range_t<G>;
template <directed G>
constexpr auto outward_vertices(const G&, const_vertex_iterator_t<G> u)
      -> const_vertex_outward_vertex_range_t<G>;

template <directed G>
constexpr auto
find_outward_vertex(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_outward_vertex_iterator_t<G>;
template <directed G>
constexpr auto find_outward_vertex(const G&,
                                   const_vertex_iterator_t<G> u,
                                   const_vertex_iterator_t<G> v)
      -> const_vertex_outward_vertex_iterator_t<G>;

template <directed G>
constexpr auto find_outward_vertex(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> vertex_outward_vertex_iterator_t<G>;
template <directed G>
constexpr auto
find_outward_vertex(const G&, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> const_vertex_outward_vertex_iterator_t<G>;
```

Modifying Functions

```
template <directed G>
constexpr void clear_outward_edges(G& g, vertex_iterator_t<G> u);
```

Outward to Uniform Range Adaptors

[TBD: map vertex_outward_edge_range → vertex_edge_range]
[TBD: map vertex_outward_vertex_range → vertex_vertex_range]

Inward Range Types

```
template <directed G>
using vertex_inward_edge_range_t = typename graph_traits<G>::vertex_inward_edge_range;
template <directed G>
using const_vertex_inward_edge_range_t =
      typename graph_traits<G>::const_vertex_inward_edge_range;

template <directed G>
using vertex_inward_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_inward_edge_range>;
template <directed G>
```

```
using const_vertex_inward_edge_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_inward_edge_range>;
template <directed G>
using vertex_inward_edge_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_inward_edge_range>;

template <directed G>
using vertex_inward_edge_size_t =
      ranges::range_size_t<typename graph_traits<G>::vertex_inward_edge_range>;
template <directed G>
using vertex_inward_edge_difference_t =
      ranges::range_difference_t<typename graph_traits<G>::vertex_inward_edge_range>;


template <directed G>
using vertex_inward_vertex_range_t =
      typename graph_traits<G>::vertex_inward_vertex_range;
template <directed G>
using const_vertex_inward_vertex_range_t =
      typename graph_traits<G>::const_vertex_inward_vertex_range;

template <directed G>
using vertex_inward_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::vertex_inward_vertex_range>;
template <directed G>
using const_vertex_inward_vertex_iterator_t =
      ranges::iterator_t<typename graph_traits<G>::const_vertex_inward_vertex_range>;
template <directed G>
using vertex_inward_vertex_sentinel_t =
      ranges::sentinel_t<typename graph_traits<G>::vertex_inward_vertex_range>;

template <directed G>
using vertex_inward_vertex_size_t =
      ranges::range_size_t<typename graph_traits<G>::vertex_inward_vertex_range>;
template <directed G>
using vertex_inward_vertex_difference_t = ranges::range_difference_t<
      typename graph_traits<G>::vertex_inward_vertex_range>;
```

Inward Functions

Common Functions

```
template <directed G>
constexpr auto inward_degree(const G& g, const_vertex_iterator_t<G> u) noexcept
      -> vertex_inward_edge_size_t<G>;
```

Vertex-Inward-Edge Range Functions

```
template <directed G>
constexpr auto inward_edges(G& g, vertex_iterator_t<G> u)
```

```
      -> vertex_inward_edge_range_t<G>;
template <directed G>
constexpr auto inward_edges(const G&, const_vertex_iterator_t<G> u)
      -> const_vertex_inward_edge_range_t<G>;

template <directed G>
constexpr auto find_inward_edge(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_inward_edge_iterator_t<G>;
template <directed G>
constexpr auto
find_inward_edge(const G&, const_vertex_iterator_t<G> u, const_vertex_iterator_t<G> v)
      -> const_vertex_inward_edge_iterator_t<G>;

template <directed G>
constexpr auto find_inward_edge(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> vertex_inward_edge_iterator_t<G>;
template <directed G>
constexpr auto find_inward_edge(const G&, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> const_vertex_inward_edge_iterator_t<G>;
```

Vertex-Inward_Vertex Range Functions

```
template <directed G>
constexpr auto inward_vertices(G& g, vertex_iterator_t<G> u)
      -> vertex_inward_vertex_range_t<G>;
template <directed G>
constexpr auto inward_vertices(const G&, const_vertex_iterator_t<G> u)
      -> const_vertex_inward_vertex_range_t<G>;

template <directed G>
constexpr auto
find_inward_vertex(G& g, vertex_iterator_t<G> u, vertex_iterator_t<G> v)
      -> vertex_inward_vertex_iterator_t<G>;
template <directed G>
constexpr auto find_inward_vertex(const G&,
                                  const_vertex_iterator_t<G> u,
                                  const_vertex_iterator_t<G> v)
      -> const_vertex_inward_vertex_iterator_t<G>;

template <directed G>
constexpr auto find_inward_vertex(G& g, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> vertex_inward_vertex_iterator_t<G>;
template <directed G>
constexpr auto
find_inward_vertex(const G&, vertex_key_t<G> ukey, vertex_key_t<G> vkey)
      -> const_vertex_inward_vertex_iterator_t<G>;
```

Modifying Functions

```
template <directed G>
constexpr void clear_inward_edges(G& g, vertex_iterator_t<G> u);
```

Adjacency and Edge-List Adaptors

vertex-vertex Range Adaptor

[TBD: create vertex_vertex_range from an incidence_graph]

edge_list_range Adaptor

[TBD: create edge_list_range from an incidence_graph]

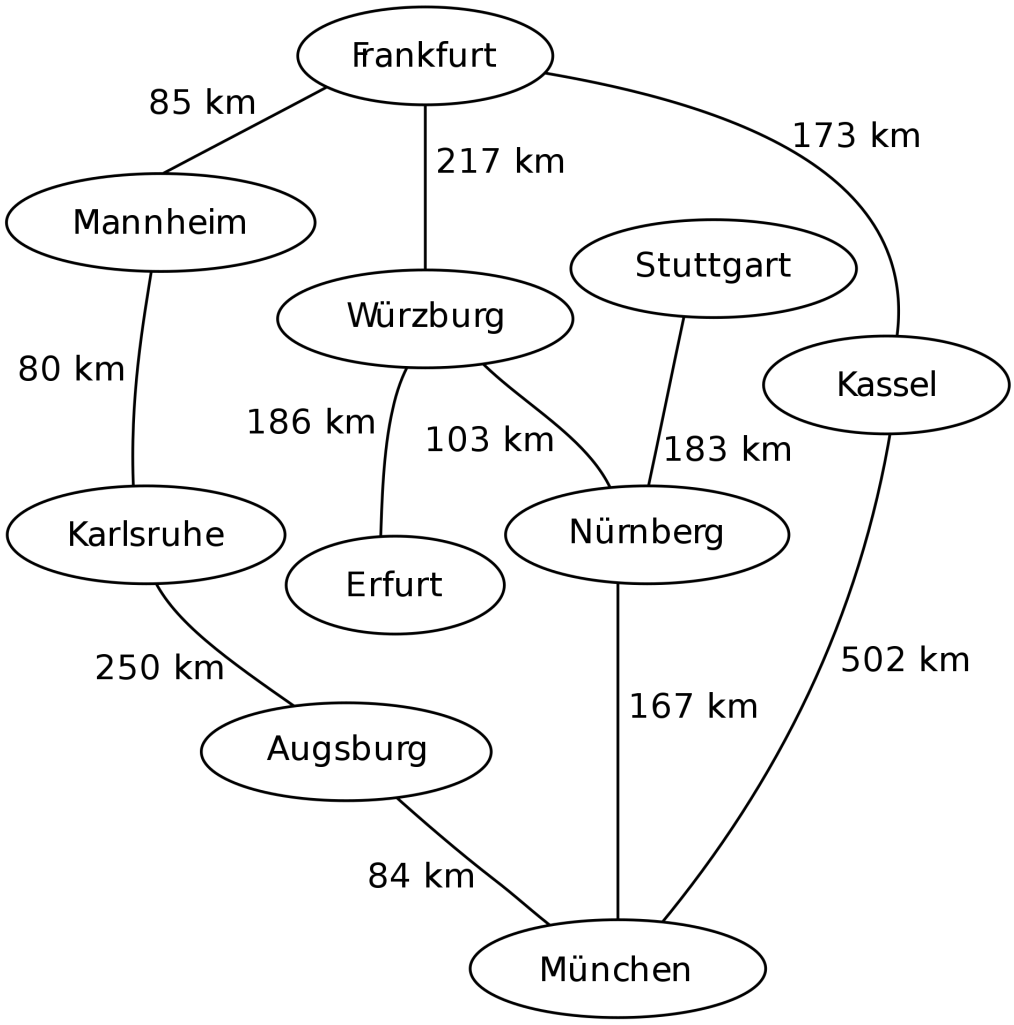
Search Ranges

Example Graph

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

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

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



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

When defined as a directed graph, assume a general top-to-bottom flow, with one exception being Nürnberg → Stuttgart.

Depth First Search (DFS)

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

The ranges have the following assumptions and behavior

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

depth_first_search_vertex_range

```
template <searchable_graph G, typename A = allocator<char>>
```

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class depth_first_search_vertex_range {
public:

  class const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = const_vertex_iterator_t<G>;
    using reference         = const value_type&;
    using difference_type = typename
           iterator_traits<const_vertex_iterator_t<G>>::difference_type;

    const_iterator();
    const_iterator(const_iterator&&);
    const_iterator(const_iterator const&);
    const_iterator(depth_first_search_vertex_range& dfs, bool end_iter = false);

    const_iterator& operator=(const_iterator&&);
    const_iterator& operator=(const_iterator const&);

    reference operator*() const;
    pointer   operator->() const;

    const_iterator& operator++();
    const_iterator operator++(int);

    bool operator==(const_iterator const& rhs) const;
    bool operator!=(const_iterator const& rhs) const;

    size_t  depth() const { return depth_first_search_->stack_.size(); }
    pointer parent() const { return parent_[vertex_key(graph_, *elem_.u)]; }
  };

  class iterator : public const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = vertex_iterator_t<G>;
    using reference         = value_type&;
    using difference_type =
          typename iterator_traits<vertex_iterator_t<G>>::difference_type;

    iterator();
    iterator(const_iterator&& iter);
    iterator(const_iterator const& iter);
    iterator(depth_first_search_vertex_range& dfs);

    iterator& operator=(iterator&& rhs);
```

```
    iterator& operator=(iterator const& rhs);

    reference operator*();
    pointer   operator->() const;

    iterator& operator++();
    iterator  operator++(int);

    pointer parent();
  };

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

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

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

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

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

Example

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

depth_first_search_edge_range

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class depth_first_search_edge_range {
```

```
public:
 class const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = edge_t<G>;
    using pointer           = const_vertex_edge_iterator_t<G>;
    using reference         = value_type const&;
    using difference_type = typename
       iterator_traits<const_vertex_edge_iterator_t<G>>::difference_type;

    const_iterator();
    const_iterator(const_iterator&&)default;
    const_iterator(const_iterator const&)default;
    const_iterator(depth_first_search_edge_range& dfs, bool end_iter = false);

    const_iterator& operator=(const_iterator&&);
    const_iterator& operator=(const_iterator const&);

    reference operator*() const;
    pointer   operator->() const;

    const_iterator& operator++();
    const_iterator operator++(int);

    bool operator==(const_iterator const& rhs) const;
    bool operator!=(const_iterator const& rhs) const;

    vertex_iterator_type inward_vertex() const;
    vertex_iterator_type outward_vertex() const;
    vertex_iterator_type back_vertex() const;

    size_t depth() const;

    bool is_path_end() const;
    bool is_back_edge() const;
  };


  class iterator : public const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = edge_t<G>;
    using reference         = value_type&;
    using pointer           = vertex_edge_iterator_t<G>;
    using difference_type =
        typename iterator_traits<vertex_edge_iterator_t<G>>::difference_type;

    iterator();
    iterator(const_iterator&& iter);
    iterator(const_iterator const& iter);
```

```
    iterator(depth_first_search_edge_range& dfs, bool end_iter = false);

    iterator& operator=(iterator&& rhs);
    iterator& operator=(iterator const& rhs);

    reference operator*();
    pointer   operator->() const;

    iterator& operator++();
    iterator operator++(int);

    vertex_iterator_type inward_vertex() const;
    vertex_iterator_type outward_vertex() const;
    vertex_iterator_type back_vertex() const;
  };

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

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

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

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

Example

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

  /* Output
```

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

Breadth First Search (BFS)

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

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

breadth_first_search_vertex_range

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class breadth_first_search_vertex_range {
  class const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = const_vertex_iterator_t<G>;
    using reference         = value_type const&;
    using difference_type =
       typename iterator_traits<const_vertex_iterator_t<G>>::difference_type;

    const_iterator();
    const_iterator(const_iterator&&);
    const_iterator(const_iterator const&);
    const_iterator(breadth_first_search_vertex_range& bfs, bool end_iter = false);

    const_iterator& operator=(const_iterator&&);
```

```
    const_iterator& operator=(const_iterator const&);

    reference operator*() const { return *elem_.u; }
    pointer   operator->() const { return elem_.u; }

    const_iterator& operator++();
    const_iterator operator++(int);

    bool operator==(const_iterator const& rhs);
    bool operator!=(const_iterator const& rhs) const;

    size_t depth() const;
  };

 class iterator : public const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = vertex_iterator_t<G>;
    using reference         = value_type&;
    using difference_type =
          typename iterator_traits<vertex_iterator_t<G>>::difference_type;

    iterator() = default;
    iterator(const_iterator&& iter);
    iterator(const_iterator const& iter);
    iterator(breadth_first_search_vertex_range& bfs, bool end_iter = false);

    iterator& operator=(iterator&& rhs);
    iterator& operator=(iterator const& rhs);

    reference operator*();
    pointer   operator->() const;

    iterator& operator++();
    iterator  operator++(int);
  };

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

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

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

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

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

Example

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

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

breadth_first_search_edge_range

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class breadth_first_search_edge_range {
 class const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = edge_t<G>;
    using pointer           = const_vertex_edge_iterator_t<G>;
    using reference         = value_type const&;
    using difference_type = typename
          iterator_traits<const_vertex_edge_iterator_t<G>>::difference_type;

    const_iterator();
    const_iterator(const_iterator&&);
    const_iterator(const_iterator const&);
    const_iterator(depth_first_search_edge_range& dfs, bool end_iter = false);

    const_iterator& operator=(const_iterator&&);
    const_iterator& operator=(const_iterator const&);

    reference operator*() const;
```

```
    pointer   operator->() const;

    const_iterator& operator++();
    const_iterator operator++(int);

    bool operator==(const_iterator const& rhs) const;
    bool operator!=(const_iterator const& rhs) const;

    vertex_iterator_type inward_vertex() const;
    vertex_iterator_type outward_vertex() const;
    vertex_iterator_type back_vertex() const;

    size_t depth() const;

    bool is_path_end() const;
    bool is_back_edge() const;
  };

class iterator : public const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = edge_t<G>;
    using reference         = value_type&;
    using pointer           = vertex_edge_iterator_t<G>;
    using difference_type =
        typename iterator_traits<vertex_edge_iterator_t<G>>::difference_type;

    iterator();
    iterator(const_iterator&& iter);
    iterator(const_iterator const& iter);
    iterator(depth_first_search_edge_range& dfs, bool end_iter = false);

    iterator& operator=(iterator&& rhs);
    iterator& operator=(iterator const& rhs);

    reference operator*();
    pointer   operator->() const;

    iterator& operator++();
    iterator  operator++(int);
  };

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

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

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

```
  const_iterator cend() const;
  };

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

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

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

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

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

Example

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

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

```
          view München
            travel Augsburg --> München 84km
            view Stuttgart
  */
```

Topological Sort

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

The ranges have the following assumptions and behavior

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

topological_sort_vertex_range

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class topological_sort_vertex_range {
 class const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = const_vertex_iterator_t<G>;
    using reference         = const value_type&;
    using difference_type   = typename
              iterator_traits<const_vertex_iterator_t<G>>::difference_type;

    const_iterator();
    const_iterator(const_iterator&&);
    const_iterator(const_iterator const&);
    const_iterator(depth_first_search_vertex_range& dfs, bool end_iter = false);

    const_iterator& operator=(const_iterator&&);
    const_iterator& operator=(const_iterator const&);

    reference operator*() const;
    pointer   operator->() const;

    const_iterator& operator++();
    const_iterator operator++(int);

    bool operator==(const_iterator const& rhs) const;
    bool operator!=(const_iterator const& rhs) const;

    size_t  depth() const { return depth_first_search_->stack_.size(); }
  };
```

```
  class iterator : public const_iterator {
  public:
    using iterator_category = input_iterator_tag;
    using value_type        = vertex_t<G>;
    using pointer           = vertex_iterator_t<G>;
    using reference         = value_type&;
    using difference_type =
          typename iterator_traits<vertex_iterator_t<G>>::difference_type;

    iterator();
    iterator(const_iterator&& iter);
    iterator(const_iterator const& iter);
    iterator(depth_first_search_vertex_range& dfs);

    iterator& operator=(iterator&& rhs);
    iterator& operator=(iterator const& rhs);

    reference operator*();
    pointer   operator->() const;

    iterator& operator++();
    iterator operator++(int);
  };

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

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

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

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

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

Example

topological_sort_edge_range

```
template <incidence_graph G, typename A = allocator<char>>
 requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_key_t<G>>
class topological_sort_edge_range {
```

```
  typename const_iterator; // input_iterator
  typename iterator;       // input_iterator

  topological_sort_edge_range(G&                    graph,
                              vertex_range_t<G> rng,
                              A                     alloc=A());

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

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

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

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

Example

Algorithms

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

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

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

Shortest Paths Algorithms

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

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

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

template <typename G, typename A = allocator<edge_iterator_t<G>>>
using vertex_path_range = decltype(
```

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

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

template <typename G, typename A>
using edge_path_range = decltype(
      make_subrange(*reinterpret_cast<edge_path_t<G, A>*>(nullptr)));

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

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

```
template <forward_iterator VertexIteratorT, typename DistanceT>
  requires is_arithmetic_v<DistanceT>
struct shortest_distance
{
  VertexIteratorT first;                  // source vertex
  VertexIteratorT last;                   // last vertex in path
  DistanceT       distance = DistanceT(); // sum of the path's edge distances
                                          // in the path
};
```

Shortest Paths - Example 1

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

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

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

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

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

for (auto& sp : short_paths) { // shortest_path<G,DistanceT,A>
  for (size_t i = 0; i < sp.vertex_path.size(); ++i) {
    if (i > 0)
      cout << " --> ";
```

```
    cout << sp.vertex_path[i]->name;
  }
  cout << "  " << sp.distance << "km\n";
}
/* Output: source = Frankfürt
    Frankfürt --> Mannheim --> Karlsruhe --> Augsburg  415km
    Frankfürt --> Würzburg --> Erfurt  403km
    Frankfürt  0km
    Frankfürt --> Mannheim --> Karlsruhe  165km
    Frankfürt --> Kassel  173km
    Frankfürt --> Mannheim  85km
    Frankfürt --> Würzburg --> Nürnberg --> München  487km
    Frankfürt --> Würzburg --> Nürnberg  320km
    Frankfürt --> Würzburg --> Nürnberg --> Stuttgart  503km
    Frankfürt --> Würzburg  217km
*/
```

Shortest Paths - Example 2

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

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

Dijkstra Algorithms

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

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

The allocator parameter used for allocation of internal data structures.

```
template <incidence_graph G,
          typename        OutIter,
          typename        DistFnc,
          typename        A = allocator<char>>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           is_arithmetic_v<invoke_result_t<DistFnc, edge_value_t<G>&>>
void dijkstra_shortest_paths(
      G&                    g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only  = true,
      DistFnc              distance_fnc = [](edge_value_t<G>&) -> size_t
                                          { return 1; },
      A                    alloc        = A());

template <incidence_graph G,
          typename        OutIter,
          typename        DistFnc,
          typename        A = allocator<char>>
  requires integral<vertex_key_t<G>> &&
           is_arithmetic_v<invoke_result_t<DistFnc, edge_value_t<G>&>> &&
           output_iterator<OutIter,
                       shortest_distance<ranges::iterator_t<vertex_range_t<G>>,
                                         invoke_result_t<DistFnc, edge_value_t<G>&>>>
template <incidence_graph G,
          typename        OutIter,
          typename        DistFnc,
          typename        A = allocator<char>>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           is_arithmetic_v<invoke_result_t<DistFnc, edge_value_t<G>&>> &&
           output_iterator<OutIter,
                       shortest_distance<ranges::iterator_t<vertex_range_t<G>>,
                                         invoke_result_t<DistFnc, edge_value_t<G>&>>>
void dijkstra_shortest_distances(
      G&                    g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only  = true,
       DistFnc               distance_fnc = [](edge_value_t<G>&) -> size_t { return 1;
},
      A                    alloc        = A())
```

Bellman-Ford Algorithms

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

```
template <incidence_graph G,
          typename        OutIter,
          typename        DistFnc,
          typename        A = allocator<char>>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           is_arithmetic_v<invoke_result_t<DistFnc, edge_value_t<G>&>>
bool bellman_ford_shortest_paths(
      G&                   g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only            = true,
      bool const           detect_neg_edge_cycles = true,
      DistFnc              distance_fnc           = [](edge_value_t<G>&) -> size_t
                                                           { return 1; },
      A                    alloc                  = A())

template <incidence_graph G,
          typename        OutIter,
          typename        DistFnc,
          typename        A = allocator<char>>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           is_arithmetic_v<invoke_result_t<DistFnc, edge_value_t<G>&>> &&
           output_iterator<OutIter,
                  shortest_distance<ranges::iterator_t<vertex_range_t<G>>,
                       invoke_result_t<DistFnc, edge_value_t<G>&>>>
bool bellman_ford_shortest_distances(
      G&                   g,
      vertex_iterator_t<G> source,
      OutIter              result_iter,
      bool const           leaves_only            = true,
      bool const           detect_neg_edge_cycles = true,
      DistFnc              distance_fnc = [](edge_value_t<G>&)
                                       -> size_t { return 1; },
      A                    alloc        = A());
```

Connected Components

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

```
template<typename G, integral CompT=uint32_t>
struct component {
  CompT                 component_number;
  vertex_iterator_t<G> vertex;

  component(CompT comp, vertex_iterator_t<G> vi);
  component()                 = default;
  component(component const&) = default;
  component(component&&)      = default;
  component& operator=(component const&) = default;
  component& operator=(component&&) = default;
  ~component()                       = default;
};

template <incidence_graph G,
          typename        OutIter,
          integral        CompT = uint32_t,
          typename        A     = allocator<char>>
  requires undirected<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, component<G, CompT>>
void connected_components(G&                    g,
                          vertex_iterator_t<G> start,
                          OutIter              result_iter,
                          A                    alloc = A());

template <incidence_graph G,
          typename        OutIter,
          integral        CompT = uint32_t,
          typename        A     = allocator<char>>
 requires directed<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, component<G, CompT>>
void connected_components(G&                 g,
                          vertex_range_t<G> rng,
                          OutIter           result_iter,
                          A                 alloc = A());
```

Strongly Connected Components

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

```
template <incidence_graph G,
          typename        OutIter,
          integral        CompT = uint32_t,
          typename        A     = allocator<char>>
  requires directed<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, component<G, CompT>>
void strongly_connected_components(G&                    g,
                                   vertex_iterator_t<G> start,
                                   OutIter              result_iter,
                                   A                    alloc = A());

template <incidence_graph G,
          typename        OutIter,
          integral        CompT = uint32_t,
          typename        A     = allocator<char>>
 requires directed<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, component<G, CompT>>
void strongly_connected_components(G&                 g,
                                   vertex_range_t<G> rng,
                                   OutIter           result_iter,
                                   A alloc           = A())
```

Biconnected Components

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

```
template<vertex_list_graph G, integral CompT=uint32_t>
struct bicomponent{
  CompT                component_number;
  vertex_iterator_t<G> vertex;

  bicomponent(CompT comp, vertex_iterator_t<G> vi);
  bicomponent()                 = default;
  bicomponent(bicomponent const&) = default;
  bicomponent(bicomponent&&)      = default;
  bicomponent& operator=(bicomponent const&) = default;
  bicomponent& operator=(bicomponent&&) = default;
  ~bicomponent()                      = default;
};

template<
  incidence_graph G,
  typename        OutIter,
  integral        CompT = uint32_t,
  typename        A     = allocator<char>
>
```

```
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, bicomponent<G, CompT>>
void biconnected_components(
       G&                   g,
       vertex_iterator_t<G> start,
       OutIter              result_iter,
       A                    alloc = A());

template<
  incidence_graph G,
  typename        OutIter,
  integral        CompT = uint32_t,
  typename        A     = allocator<char>
>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, bicomponent<G, CompT>>
void biconnected_components(
       G&                 g,
       vertex_range_t<G>  rng,
       OutIter            result_iter,
       A                  alloc = A());
```

Articulation Points

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

```
template<
  incidence_graph  G,
  typename         OutIter,
  integral         CompT = uint32_t,
  typename         A     = allocator<char>
>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, vertex_iterator_t<G>>
void articulation_points(
       G&                   g,
       vertex_iterator_t<G> start,
       OutIter              result_iter,
       A                    alloc = A());

template<
  incidence_graph  G,
  typename         OutIter,
  integral         CompT = uint32_t,
  typename         A     = allocator<char>
>
```

```
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, vertex_iterator_t<G>>
void articulation_points(
       G&                g,
       vertex_range_t<G> rng,
       OutIter           result_iter,
       A                 alloc = A());
```

Transitive Closure

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

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

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

```
template <incidence_graph G, typename OutIter, typename A = allocator<char>>
  requires ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, reaches<G>>
constexpr void dfs_transitive_closure(G&      g,
                                      OutIter result_iter,
                                      A       alloc = A())
```

The dense graph version uses Warshall's algorithm with $O(V^3)$ complexity. There are two versions, with and without an execution policy

```
template <incidence_graph G, typename OutIter, typename A = allocator<bool>>
  requires directed<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, reaches<G>>
constexpr void warshall_transitive_closure(G& g, OutIter result_iter, A alloc = A())

template <typename        ExecutionPolicy,
          incidence_graph G,
          typename        OutIter,
          typename        A = allocator<bool>>
  requires directed<G> &&
           ranges::random_access_range<vertex_range_t<G>> &&
           integral<vertex_key_t<G>> &&
           output_iterator<OutIter, reaches<G>>
constexpr void warshall_transitive_closure(ExecutionPolicy&& policy,
                                           G&                g,
```

```
                                        OutIter             result_iter,
                                        A                   alloc = A());
```

erase & erase_if (Uniform Container Erasure)

Uniform container erasure is not supported because the graph is needed for context in addition to vertices or edge(s) for all erase functions. For instance, erase(g,vrange) is the API needed, but this breaks the requirement for the standard function signature.

Additional Algorithms

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

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

Graph Data Structures

Graph data structures are used to store the graph in memory. The two data structures in the library store directed and undirected graphs for common use cases. Only constructors are shown in the paper. An implementation will likely have additional member functions to support the free functions, but their definition isn't important in this context.

Graph Template Parameters

The graph types are defined as templated classes with the following parameters.

Parameter	Valid Values	Description

GV	(user-defined)	The graph value type defined by the user. It can be most valid C++ value types 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 Defined Values section for more information.
VV	(user-defined)	The vertex value type. (See GV.)
EV	(user-defined)	The edge value type. (See GV.)
KeyT	Signed or unsigned integer (default is uint32_t)	For graphs that store vertices in vectors, this type is used for a vertex's key, typically stored in an edge.
VContainer<T,A>	Random-access container	The container used to store vertices. vector and deque are valid containers, or a user-defined random access container with value-type T and allocator A.
EContainer<T,A>	Random-access container	The container used to store edges. vector and deque are valid containers, or user-defined random access containers with value-type T and allocator A.
A	allocator<char>	A standard C++ allocator. Rebind is used to redefine for both vertex and edge types.

User Defined Value Types

User-defined value 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_adjacency_vector Graph

The directed_adjacency_vector is designed to minimize storage requirements and offer good performance characteristics, where vertices and edges are created when it is constructed. Vertices are stored in a random-access container and edges are stored in a separate container, ordered by their inward vertex key.

When the graph is constructed, reserve(n) will be called on the vertices container, where n is the number of vertices that will be created, only if the container defines a reserve(n) member function. The same is done for the edges container.

Feature	**Description**
Graph concepts?	incidence_graph, adjacency_graph, edge_list_graph, outward_incidence_graph, directed
vertex_range	random_access
vertex_edge_range	random_access
User-defined value type?	Graph, vertex, edge
Multi-edge?	Allowed
Mutable?	After construction, vertices and edges cannot be added but properties can be changed.
Other constraints	Edges in the source data must be ordered by their source vertex key during construction.

Class

```
template <typename VV                                          = empty_value,
          typename EV                                          = empty_value,
          typename GV                                          = empty_value,
          integral KeyT                                        = uint32_t,
          template <typename V, typename A> class VContainer = vector,
          template <typename E, typename A> class EContainer = vector,
          typename Alloc                                       = allocator<char>>
class directed_adjacency_vector {
public:
  directed_adjacency_vector()                                                 = default;
  directed_adjacency_vector(directed_adjacency_vector&& rhs) noexcept = default;
  directed_adjacency_vector(const directed_adjacency_vector&)         = default;

  directed_adjacency_vector(const allocator_type& alloc);
  directed_adjacency_vector(const graph_value_type&,
                            const allocator_type& alloc = allocator_type());
  directed_adjacency_vector(graph_value_type&&,
                            const allocator_type& alloc = allocator_type());
```

```
  // Construct a graph using functors to extract edge key & value and
  // vertex value for edge and vertex ranges.
  template <typename ERng,
            typename EKeyFnc,
            typename EValueFnc,
            typename VRng,
            typename VValueFnc>
    requires edge_value_extractor<ERng, EKeyFnc, EValueFnc>
          && vertex_value_extractor<VRng, VValueFnc>
  directed_adjacency_vector(const ERng&     erng,
                            const VRng&     vrng,
                            const EKeyFnc&  ekey_fnc,
                            const EValueFnc& evalue_fnc,
                            const VValueFnc& vvalue_fnc,
                            const GV&       gv    = GV(),
                            const Alloc&    alloc = Alloc());

  // Construct a graph using functors to extract edge key & value for edge range.
 template <typename ERng, typename EKeyFnc, typename EValueFnc>
    requires edge_value_extractor<ERng, EKeyFnc, EValueFnc>
  directed_adjacency_vector(const ERng&     rng,
                            const EKeyFnc&  ekey_fnc,
                            const EValueFnc& evalue_fnc,
                            const GV&       gv = GV(),
                            const Alloc&    alloc = Alloc());


  // Construct a graph from edges & their values
 directed_adjacency_vector(
    const initializer_list<
          tuple<vertex_key_type, vertex_key_type, edge_value_type>>& ilist,
    const Alloc&                                                     alloc = Alloc());

  // Construct a graph from edges
 directed_adjacency_vector(
    const initializer_list<tuple<vertex_key_type, vertex_key_type>>& ilist,
    const Alloc&                                                     alloc = Alloc());

  ~directed_adjacency_vector() = default;

  directed_adjacency_vector& operator=(const directed_adjacency_vector&) = default;
  directed_adjacency_vector& operator=(directed_adjacency_vector&&) = default;
};
```

Graph Traits

The following graph_traits specialization is defined for the directed_adjacency_vector. The types defined in this struct are required but their actual value may vary in each implementation and are only shown for exposition. For instance, there is no requirement that the types be defined in graph_type.

```
template <typename                                    VV,
          typename                                    EV,
          typename                                    GV,
          integral                                    KeyT,
          template <typename V, typename A> class VContainer,
          template <typename E, typename A> class EContainer,
          typename                                    Alloc>
struct graph_traits<directed_adjacency_vector<VV, EV, GV, KeyT,
                    VContainer, EContainer, Alloc>> {
 using graph_type =
        directed_adjacency_vector<VV, EV, GV, KeyT, VContainer, EContainer, Alloc>;
  using graph_value_type = typename graph_type::graph_value_type;
  using allocator_type   = typename graph_type::allocator_type;

  using vertex_type       = typename graph_type::vertex_type;
  using vertex_key_type   = typename graph_type::vertex_key_type;
  using vertex_value_type = typename graph_type::vertex_value_type;

  using edge_type       = typename graph_type::edge_type;
  using edge_key_type   = ordered_pair<vertex_key_type, vertex_key_type>;
  using edge_value_type = typename graph_type::edge_value_type;

  using vertex_range       = typename graph_type::vertex_range;
  using const_vertex_range = typename graph_type::const_vertex_range;

  using edge_range       = typename graph_type::edge_range;
  using const_edge_range = typename graph_type::const_edge_range;

  using vertex_outward_edge_range = typename graph_type::vertex_outward_edge_range;
  using const_vertex_outward_edge_range =
        typename graph_type::const_vertex_outward_edge_range;

  using vertex_outward_vertex_range =
        typename graph_type::vertex_outward_vertex_range;
  using const_vertex_outward_vertex_range =
        typename graph_type::const_vertex_outward_vertex_range;

  using vertex_edge_range       = vertex_outward_edge_range;
  using const_vertex_edge_range = const_vertex_outward_edge_range;

  using vertex_vertex_range       = vertex_outward_vertex_range;
  using const_vertex_vertex_range = const_vertex_outward_vertex_range;
};
```

Example 1

The following example shows how the graph can be constructed using extractor functors for values of vertices, and keys & values of edges. If `empty_value` is used for the type, they can simply return `empty_value()`.

```
struct route {
  string from;
  string to;
  int    miles;
};
vector<string> cities = {"Apex", "Cary", "Raleigh"};
vector<route>  routes = {{"Apex", "Cary", 5}, {"Apex", "Raleigh", 10}};
sort(cities);

using G          = directed_adjacency_vector<name_value, weight_value, name_value>;
using edge_key_t = typename G::edge_key_type;

auto find_city = [&cities](string const& city)
                 { return find(cities, city) - begin(cities); };

auto vertex_value = [](string const& name) { return name; };

auto edge_key    = [&cities, &find_city](route const& r)
                 { return edge_key_t(find_city(r.from), find_city(r.to)); };
auto edge_value  = [&cities](route const& r)
                 { return r.miles; };

G g(routes, cities, edge_key, edge_value, vertex_value,
    name_value("NC Routes"));
```

Notes

1. Vertex keys are in the range [0..n), and the edge_key refers to those indices to refer to the vertices.
2. Edges must be ordered in inwardvertex_key(g,uv) order.
3. If edges refer to more vertices than exist in VRng, additional vertices will be added to match the number required.
4. A similar constructor exists that excludes the vertex range & vertex value functor. The edges are scanned first to determine the maximum vertex index needed.

Example 2

A simpler way to construct the graph is to use the initializer lists, though only edges can be used. The following example is similar to the previous one except that there are no vertex values.

```
using G1 = directed_adjacency_vector<empty_value, weight_value>;
G1 g1{{0, 1, 5}, {0, 2, 10}};
```

Notes

1. Vertex keys are in the range [0..n), and the edge_key refers to those indices to refer to the vertices.
2. Edges must be ordered in inward_vertex_key(g,uv) order.
3. If edges refer to more vertices than exist in VRng, additional vertices will be added to match the number required.
4. A similar constructor exists without edge values, where only the inward_vertex_key and outward_vertex_key are provided.

undirected_adjacency_list Graph

The undirected_adjacency_list is designed to represent an undirected graph without minimal storage requirements and offer good performance characteristics, where vertices are only added during graph construction. Edges can be added when the graph is constructed and can be added or erased after the graph is constructed. Vertices are stored in a random-access container and edges are stored in doubly-linked lists.

When the graph is constructed, reserve(n) will be called on the vertices container, where n is the number of vertices that will be created, only if the container defines a reserve(n) member function.

Feature	**Description**
Graph concepts?	incidence_graph, adjacency_graph, edge_list_graph, incremental_edge_graph, decremental_edge_graph, undirected
vertex_range	random_access
vertex_edge_range	bidirectional
User-defined value type?	Graph, vertex, edge
Multi-edge?	Allowed
Mutable?	After construction, vertices cannot be added or deleted, but edges can be added or erased and properties can be changed for all types.
Other constraints	Edges in the source data must be ordered by their source vertex key during construction.

Class

```
template <typename VV                                          = empty_value,
          typename EV                                          = empty_value,
          typename GV                                          = empty_value,
          integral KeyT                                        = uint32_t,
          template <typename V, typename A> class VContainer = vector,
          typename Alloc                                       = allocator<char>>
class undirected_adjacency_list {
public:
  undirected_adjacency_list()                                              = default;
  undirected_adjacency_list(undirected_adjacency_list&& rhs) noexcept = default;
  undirected_adjacency_list(const undirected_adjacency_list&)         = default;

  undirected_adjacency_list(const allocator_type& alloc);
  undirected_adjacency_list(const graph_value_type&,
                            const allocator_type& alloc = allocator_type());
  undirected_adjacency_list(graph_value_type&&,
                            const allocator_type& alloc = allocator_type());

  // Construct a graph using functors to extract edge key & value and
```

```
  // vertex value for edge and vertex ranges.
 template <typename ERng,
            typename EKeyFnc,
            typename EValueFnc,
            typename VRng,
            typename VValueFnc>
    requires edge_value_extractor<ERng, EKeyFnc, EValueFnc>
          && vertex_value_extractor<VRng, VValueFnc>
  undirected_adjacency_list(const ERng&      erng,
                            const VRng&      vrng,
                            const EKeyFnc&   ekey_fnc,
                            const EValueFnc& evalue_fnc,
                            const VValueFnc& vvalue_fnc,
                            const GV&        gv    = GV(),
                            const Alloc&     alloc = Alloc());

  // Construct a graph using functors to extract edge key & value for edge range.
 template <typename ERng, typename EKeyFnc, typename EValueFnc>
    requires edge_value_extractor<ERng, EKeyFnc, EValueFnc>
  undirected_adjacency_list(const ERng&      erng,
                            const EKeyFnc&   ekey_fnc,
                            const EValueFnc& evalue_fnc,
                            const GV&        gv    = GV(),
                            const Alloc&     alloc = Alloc());

  // Construct a graph from edges & their values
 undirected_adjacency_list(
    const initializer_list<
          tuple<vertex_key_type, vertex_key_type, edge_value_type>>& ilist,
    const Alloc&                                                     alloc = Alloc());

  // Construct a graph from edges
 undirected_adjacency_list(
    const initializer_list<tuple<vertex_key_type, vertex_key_type>>& ilist,
    const Alloc&                                                     alloc = Alloc());

 ~undirected_adjacency_list();

  undirected_adjacency_list& operator=(const undirected_adjacency_list&) = default;
  undirected_adjacency_list& operator=(undirected_adjacency_list&&) = default;
};
```

Graph Traits

The following graph_traits specialization is defined for the undirected_adjacency_list. The types defined in this struct are required but their actual value may vary in each implementation and are only shown for exposition. For instance, there is no requirement that the types be defined in graph_type.

```
template <typename                                VV,
          typename                                EV,
          typename                                GV,
```

```
          integral                                KeyT,
          template <typename V, typename A> class VContainer,
          typename                                Alloc>
struct graph_traits<undirected_adjacency_list<VV, EV, GV, KeyT, A>> {
  using graph_type = undirected_adjacency_list<VV, EV, GV, KeyT, VContainer, Alloc>;
  using graph_value_type = typename graph_type::graph_value_type;
  using allocator_type   = typename graph_type::allocator_type;

  using vertex_type       = typename graph_type::vertex_type;
  using vertex_key_type   = typename graph_type::vertex_key_type;
  using vertex_value_type = typename graph_type::vertex_value_type;

  using edge_type       = typename graph_type::edge_type;
  using edge_key_type   = unordered_pair<vertex_key_type, vertex_key_type>;
  using edge_value_type = typename graph_type::edge_value_type;

  using vertex_range       = typename graph_type::vertex_range;
  using const_vertex_range = typename graph_type::const_vertex_range;

  using edge_range       = typename graph_type::edge_range;
  using const_edge_range = typename graph_type::const_edge_range;

  using vertex_edge_range       = typename vertex_type::vertex_edge_range;
  using const_vertex_edge_range = typename vertex_type::const_vertex_edge_range;

  using vertex_vertex_range       = typename vertex_type::vertex_vertex_range;
  using const_vertex_vertex_range = typename vertex_type::const_vertex_vertex_range;
};
```

Examples

The same examples and notes from directed_adjacency_vector can be used for undirected_adjacency_list. The graphs aren't the same though. In the directed graph, only the Apex → Raleigh route exists between the two cities (e.g. a one-way road), but the undirected graph also has a two-way road Apex <--> Raleigh.

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 outward 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.

In theory it should be possible to accept any kind of ordered key as a vertex key, allowing for the use of std::map and std::unordered_map to store vertices. However, this imposes requirements on algorithms with burdensome implementation and performance penalties because they need to create internal data structures that support similar behavior as those for the vertices. For instance, storing vertices in std::unordered_map

requires algorithms to also use std::unordered_map because of the potential sparsity of the keys. While this may be acceptable in some situations, it sacrifices important performance requirements and will limit its general acceptance. Additionally, the equivalent of a std:vector is used for storing vertices in real-world uses. For these reasons this paper is focused on random-access and contiguous containers (e.g. std::vector, std::array, std:deque), while acknowledging that other forms are possible.

A subtlety exists when using **undirected graphs** and **mutable user-defined edge properties**. Undirected graphs are commonly implemented by duplicating an edge for both it's vertices. For instance, given 2 vertices u & v, two edges are created, one for u --> v and another for v --> u. While this works for the algorithms, if there is a user-defined property for the edge then there are two copies of the property. Updating property on u --> v also requires updating it on v --> u which raises the risk to assure it is done and can have performance issues. Ideally a single edge instance should exist, but this creates a conflict because the edge is no longer assigned to an owning vertex – it is shared between its two vertices. The edge is in two edge lists, likely impacting performance. A common implementation is to use a doubly-linked list for edge lists, reducing locality of reference and increasing CPU cache misses when traversing the edges. There is clearly a trade-off between mutability, memory usage and performance that need to be considered when designing an unordered graph. This also applies to **bi-directional graphs**.

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. Andrew Lumsdaine and Kevin Deweese are supported in part by NSF Award 1716828.

References

1. Implementations
    a. The Boost Graph Library (BGL)
    b. IgraphT
    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. 3rd edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest