Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 D1709r5
Date: 2022-12-06
Reply to: Phil Ratzloff (phil.ratzloff@sas.com),
Andrew Lumsdaine (lumsdaine@gmail.com)

Contributors: Richard Dosselmann (University of Regina)
Michael Wong (Codeplay)
Matthew Galati (Amazon)
Jens Maurer
Jesun Firoz
Kevin Deweese
Muhammad Osama (AMD, Inc)

Audience: SG1, LEWG, LWG
Source: github.com/stdgraph/graph-v2

Contents

Revision History

P1709R6

Extensions and refinements to r5

— Add default implementation for `target_id(g,uv)` when the graph matches the pattern `forward_range<forward_range<integral>>` or `forward_range<forward_range<tuple<integral,...>>>`.

P1709R5

Extensions and refinements to r4

— Added basic_* versions for depth first search, breadth first search and topoloical sort views. Also shortended the view names to use bfs and dfs to avoid long names.

— Replace `adjacency_list` with `index_adjacency_list` concept in algorithms to simplify the definitions.

— Updated Shortest Paths algorithms with final definitions.

— Added Topological Sort algorithm description.

— Added summary table for `compressed_graph`.

— Added and updated algorithm descriptions.

— Added Graph Operators chapter.

P1709R4

This was a major redesign that incorporated all the experience and input from the past four years.

— Revisit the algorithms to be considered.

— Reduce the scope to focus on an adjacency list with outgoing edges, edge list, and remove mutable interface functions.

— Replace directed and undirected concepts with overridable types of unordered_edge for a graph type.

— Simplify the Graph Container types and functions. In particular, const and non-const variations were consolidated to a single definition to handle both cases when appropriate.

— All Graph Container Interface functions are customization points.

— Introduce Views, inspired by NWGraph design, resulting in simpler and cleaner interfaces to traverse a graph, and simplifying the container interface design.

— Add support for bipartite and multipartite graphs.

— Replace the two container implementations with compressed_graph, based on the Compressed Sparse Row matrix, a commonly used data structure for high-performance graphs.

P1709R3

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

P1709R2

Define the **uniform API** for undirected and directed algorithms (an extended API also exists for directed graphs). Added **concepts** for undirected, directed and bidirected graphs. Refined **DFS** and **BFS** range definitions from prototype experience. Refined **shortest paths** and **transitive closure** algorithms from input and prototype experience.

P1709R1

Rewrite with a focus on a **purely functional design**, emphasizing the algorithms and graph API. Also added **concepts** and **ranges** into the design. Addressed concerns from Cologne review to change to functional design.

P1709R0

Focus on **object-oriented API** for data structures and example code for a few algorithms.

Chapter1 Getting Started

This paper is one of several interrelated proposals related to a Graph Library that have been broken out for easier consumption. The following table describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now broken into the following papers.
P9901	Active	Graph Library Overview and Introduction.
P9902	Active	Graph Library Algorithms.
P9903	Active	Graph Library Operators.
P9904	Active	Graph Library Views.
P9905	Active	Graph Library Container Inferface.
P9906	Active	Graph Library Containers.
P9907	Active	Graph Library Adaptors.

Table 1.1 — Graph Library Papers

1.1 Naming Conventions

Table 1.2 shows the naming conventions used throughout the graph library documents.

Template Parameter	Type Alias	Variable Names	Description
G			Graph
	graph_reference_t<G>	g	Graph reference
GV		val	Graph Value, value or reference
V	vertex_t<G>		Vertex
	vertex_reference_t<G>	u,v,x,y	Vertex reference. u is the source (or only) vertex. v is the target vertex.
VId	vertex_id_t<G>	uid,vid,seed	Vertex id. uid is the source (or only) vertex id. vid is the target vertex id.
VV	vertex_value_t<G>	val	Vertex Value, value or reference. This can be either the user-defined value on a vertex, or a value returned by a function object (e.g. VVF) that is related to the vertex.
VR	vertex_range_t<G>	ur,vr	Vertex Range
VI	vertex_iterator_t<G>	ui,vi first,last	Vertex Iterator. ui is the source (or only) vertex. vi is the target vertex.
VVF		vvf	Vertex Value Function: vvf(u) → vertex value, or vvf(uid) → vertex value, depending on requirements of the consume algorithm or view.
VProj		vproj	Vertex descriptor projection function: vproj(x) → vertex_descriptor<VId,VV> .
	partition_id_t<G>	pid	Partition id.
		P	Number of partitions.
PVR	partition_vertex_range_t<G>	pur,pvr	Partition vertex range.
E	edge_t<G>		Edge
	edge_reference_t<G>	uv,vw	Edge reference. uv is an edge from vertices u to v . vw is an edge from vertices v to w .
EId	edge_id_t<G>	eid,uvid	Edge id, a pair of vertex_ids.
EV	edge_value_t<G>	val	Edge Value, value or reference. This can be either the user-defined value on an edge, or a value returned by a function object (e.g. EVF) that is related to the edge.
ER	vertex_edge_range_t<G>		Edge Range for edges of a vertex
EI	vertex_edge_iterator_t<G>	uvi,vwi	Edge Iterator for an edge of a vertex. uvi is an iterator for an edge from vertices u to v . vwi is an iterator for an edge from vertices v to w .
EVF		evf	Edge Value Function: evf(uv) → edge value, or evf(eid) → edge value, depending on the requirements of the consuming algorithm or view.
EProj		eproj	Edge descriptor projection function: eproj(x) → edge_descriptor<VId,Sourced,EV> .
PER	partition_edge_range_t<G>		Partition Edge Range for edges of a partition vertex.

Table 1.2 — Naming Conventions for Types and Variables

Chapter2 Algorithms

Our proposed set of algorithms are grouped into Tier 1, Tier 2, and Tier 3. All Tier 1 algorithms are included in this proposal and summarized in the lists below. Other tiers are outlined in the section 2.11 Other Algorithms.

Shortest Paths
— Breadth-First search
— Dijkstra's algorithm
— Bellman-Ford

Clustering
— Triangle counting

Communities
— Label propagation

Components
— Articulation points
— Connected components
— Biconnected components
— Strongly connected components

Directed Acyclic Graphs
— Topological sort

Maximal Independent Set
— Maximal independent set

Link Analysis
— Jaccard coefficient

Minimal Spanning Tree
— Kruskal Minimal Spanning Tree
— Prim Minimal Spanning Tree

2.1 Introduction

[PHIL: Include this in the Conventions secton for Algorithms and Operators]

Basic characteristics of the algorithms shown below are summarized in tables of the following form:

Complexity $\mathcal{O}(\lvert E\rvert + \lvert V\rvert)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

The parts of the table have the following meaning:

— **Complexity** The complexity of the algorithm based on the number of vertices (V) and edges (E).

— **Throws?** Will the algorithm throw at all? If so, look at the *Throws* section after the function prototypes for details.

— **Multi-edge?** Does the algorithm act as expected if more than one edge with the same direction exists between the same two vertices?

— **Cycles?** Does the algorithm act act as expected if a vertex (or edge) is part of a cycle?

— **Directed?** Is the algorithm only for directed graphs, or can it also be used for undirected graphs that have complimentary edges, with different directions, between two vertices.

[PHIL: The Directed? section needs work.]

2.2 Algorithm Concepts

The abstraction that is used for describing and analyzing almost all graph algorithms is the adjacency list. Naturally then implementations of graph algorithms in C++ will operate on a data structure representing an adjacency list. And generic algorithms will be written in terms of concepts that capture the essential operations that a concrete data structure must provide in order to be used as an abstraction of an adjacency list.

Most fundamentally (as illustrated above), an adjacency list is a collection of vertices, each of which has a collection of outgoing edges. In terms of existing C++ concepts, we can consider an adjacency list to be a range of ranges (or, more specifically, a random access range of forward ranges). The outer range is the collection of vertices, and the inner ranges are the collections of outgoing edges.

```
template <class G, class WF, class DistanceValue, class Compare, class Combine>
concept basic_edge_weight_function = // e.g.  weight(uv)
    is_arithmetic_v<DistanceValue> &&
    strict_weak_order<Compare, DistanceValue, DistanceValue> &&
    assignable_from<add_lvalue_reference_t<DistanceValue>,
        invoke_result_t<Combine, DistanceValue, invoke_result_t<WF, edge_reference_t<G>>>>;
```

```
template <class G, class WF, class DistanceValue>
concept edge_weight_function = // e.g.  weight(uv)
     is_arithmetic_v<invoke_result_t<WF, edge_reference_t<G>>> &&
     basic_edge_weight_function<G,
                              WF,
                              DistanceValue,
                              less<DistanceValue>,
                              plus<DistanceValue>>;
```

2.3 Shortest Paths

2.3.1 Unweighted Shortest Paths: Breadth-First Search

2.3.1.1 Breadth-First Search, Single Source, Initialization

```
template <class DistanceValue>
constexpr auto breadth_first_search_invalid_distance() {
  return numeric_limits<DistanceValue>::max(); // exposition only
}

template <class DistanceValue>
constexpr auto breadth_first_search_zero() { return DistanceValue(); } // exposition only

template <class Distances>
constexpr void init_breadth_first_search(Distances& distances) {
  // exposition only
  ranges::fill(distances,
            breadth_first_search_invalid_distance<ranges::range_value_t<Distances>>());
}

template <class Predecessors>
constexpr void init_breadth_first_search(Predecessors& predecessors) {
  // exposition only
  size_t i = 0;
  for(auto& pred : predecessors)
    pred = i++;
}
```

Effects:

— Each `predecessors[i]` is initialized to `i`.

2.3.1.2 Breadth-First Search, Single Source

Compute the breadth-first path and associated distance from vertex `source` to all reachable vertices in `graph`.

Complexity	Throws? Yes	Cycles? No
$\mathcal{O}((\lvert E\rvert + \lvert V\rvert)\log\lvert V\rvert)$	**Multi-edge?** No	**Directed?** Yes

Note that complexity may be $\mathcal{O}(\lvert E\rvert + \lvert V\rvert\log\lvert V\rvert)$ for certain implementations.

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>>
void breadth_first_search(
     G&& g, // graph
```

```
    vertex_id_t<G> source, // starting vertex_id
    Distances& distances, // out:  Distances[uid] of uid from source in number of edges
    Predecessors& predecessors, // out:  predecessor[uid] of uid in path
    Allocator alloc = Allocator());

template <index_adjacency_list G,
        ranges::random_access_range Distances,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>>
void breadth_first_search(
    G&& g, // graph
    vertex_id_t<G> seed, // starting vertex_id
    Distances& distances, // out:  Distances[uid] of uid from seed in number of edges
    Allocator alloc = Allocator());
```

1 *Preconditions:*

(1.1) — `0 <= source < num_vertices(graph)`.

(1.2) — `distances` will be initialized with `init_breadth_first_search`.

(1.3) — `predecessors` will be initialized with `init_breadth_first_search`.

2 *Effects:*

(2.1) — If vertex with index `i` is reachable from vertex `source`, then `distances[i]` will contain the lowest number of edges from `source` to vertex `i`. Otherwise `distances[i]` will contain `breadth_first_search_invalid_distance ()`.

(2.2) — If vertex with index `i` is reachable from vertex `source`, then `predecessors[i]` will contain the predecessor vertex of vertex `i`. Otherwise `predecessors[i]` will contain `i`.

3 *Throws:* `out_of_range` is thrown when `source` is not in the range `0 <= source < num_vertices(graph)`.

2.3.2 Weighted Shortest Paths

2.3.2.1 Shortest Paths Initialization

```
template <class DistanceValue>
constexpr auto shortest_path_invalid_distance() {
  return numeric_limits<DistanceValue>::max(); // exposition only
}

template <class DistanceValue>
constexpr auto shortest_path_zero() { return DistanceValue(); } // exposition only

template <class Distances>
constexpr void init_shortest_paths(Distances& distances) {
  // exposition only
  ranges::fill(distances,
            shortest_path_invalid_distance<ranges::range_value_t<Distances>>());
}

template <class Distances, class Predecessors>
constexpr void init_shortest_paths(Distances& distances, Predecessors& predecessors) {
  // exposition only
  init_shortest_paths_distances(distances);
  size_t i = 0;
  for(auto& pred : predecessors)
    pred = i++;
}
```

1 *Effects:* :

(1.1) — `init_shortest_paths(distances)` sets all elements in `distance` to `shortest_path_invalid_distance()`

(1.2) — `init_shortest_paths(distances,predecessors)` does the same as `shortest_path_invalid_distance(distances)` and sets `predecessors[i] = i` for `i < size(predecessors)`.

2 *Returns:*

(2.1) — `shortest_path_invalid_distance()` returns a sentinel value for an invalid distance, typically `numeric_limits<DistanceValue>::max()` for numeric types.

(2.2) — `shortest_path_zero()` returns a value for for a zero-length path, typically `0` for numeric types.

2.3.2.2 Dijkstra Single Source Shortest Paths and Shortest Distances

Compute the shortest path and associated distance from vertex `source` to all reachable vertices in `graph` using non-negative weights.

Complexity $\mathcal{O}((\|E\| + \|V\|)\log\|V\|)$	**Throws?** Yes **Multi-edge?** No	**Cycles?** No **Directed?** Yes

Note that complexity may be $\mathcal{O}(|E| + |V|\log|V|)$ for certain implementations.

The following functions are split into the common and general cases, where the general cases allow the caller to specify `Compare` and `Combine` functions (e.g. less and add). Concepts and types from `std::ranges` don't include the namespace prefix for brevity and clarity of purpose.

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>> &&
       edge_weight_function<G, WF, ranges::range_value_t<Distances>>
void dijkstra_shortest_paths(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Distances& distances, // out: Distances[uid] of uid from source
     Predecessors& predecessors, // out: predecessor[uid] of uid in path
     WF&& weight =
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());

template <index_adjacency_list G,
        ranges::random_access_range Distances,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       edge_weight_function<G, WF, ranges::range_value_t<Distances>>
void dijkstra_shortest_distances(
     G&& g, // graph
     vertex_id_t<G> seed, // starting vertex_id
     Distances& distances, // out: Distances[uid] of uid from seed
     WF&& weight =
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());
```

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class Compare,
```

```
        class Combine,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>> &&
       basic_edge_weight_function<G, WF, ranges::range_value_t<Distances>, Compare, Combine>
void dijkstra_shortest_paths(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from source
     Predecessors& predecessors, // out:  predecessor[uid] of uid in path
     Compare&& compare,
     Combine&& combine,
     WF&& weight = // default weight(uv) -> 1
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());

template <index_adjacency_list G,
        ranges::random_access_range Distances,
        class Compare,
        class Combine,
        class WF = std::function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       basic_edge_weight_function<G, WF, ranges::range_value_t<Distances>, Compare, Combine>
void dijkstra_shortest_distances(
     G&& g, // graph
     vertex_id_t<G> seed, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from seed
     Compare&& compare,
     Combine&& combine,
     WF&& weight = // default weight(uv) -> 1
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());
```

1 *Mandates:*

(1.1) — The weight function `w` must return a non-negative value.

2 *Preconditions:*

(2.1) — `0 <= source < num_vertices(graph)`.

(2.2) — `distances` will be initialized with `init_shortest_paths`.

(2.3) — `predecessors` will be initialized with `init_shortest_paths`.

3 *Effects:*

(3.1) — If vertex with index `i` is reachable from vertex `source`, then `distances[i]` will contain the distance from `source` to vertex `i`. Otherwise `distances[i]` will contain `shortest_path_invalid_distance()`.

(3.2) — If vertex with index `i` is reachable from vertex `source`, then `predecessors[i]` will contain the predecessor vertex of vertex `i`. Otherwise `predecessors[i]` will contain `i`.

4 *Throws:* `out_of_range` is thrown when `source` is not in the range `0 <= source < num_vertices(graph)`.

5 *Remarks:* Bellman-Ford Shortest Paths allows negative weights with the consequence of greater complexity.

2.3.2.3 Bellman-Ford Single Source Shortest Paths and Shortest Distances

Compute the shortest path and associated distance from vertex `source` to all reachable vertices in `graph`.

The following functions are split into the common and general cases, where the general cases allow the caller to specify `Compare` and `Combine` functions (e.g. less and add). Concepts and types from `std::ranges` don't include the namespace

Complexity $\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$	**Throws?** Yes **Multi-edge?** No	**Cycles?** No **Directed?** Yes

prefix for brevity and clarity of purpose.

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>> &&
       edge_weight_function<G, WF, ranges::range_value_t<Distances>>
void bellman_ford_shortest_paths(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from source
     Predecessors& predecessors, // out:  predecessor[uid] of uid in path
     WF&& weight =
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator())

template <index_adjacency_list G,
        ranges::random_access_range Distances,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       edge_weight_function<G, WF, ranges::range_value_t<Distances>>
void bellman_ford_shortest_distances(
     G&& g, // graph
     vertex_id_t<G> seed, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from seed
     WF&& weight =
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());
```

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class Compare,
        class Combine,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>> &&
       basic_edge_weight_function<G, WF, ranges::range_value_t<Distances>, Compare, Combine>
void bellman_ford_shortest_paths(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from source
     Predecessors& predecessors, // out:  predecessor[uid] of uid in path
     Compare&& compare,
     Combine&& combine,
     WF&& weight = // default weight(uv) -> 1
         [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());

template <index_adjacency_list G,
```

```
        ranges::random_access_range Distances,
        class Compare,
        class Combine,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       basic_edge_weight_function<G, WF, ranges::range_value_t<Distances>, Compare, Combine>
void bellman_ford_shortest_distances(
     G&& g, // graph
     vertex_id_t<G> seed, // starting vertex_id
     Distances& distances, // out:  Distances[uid] of uid from seed
     Compare&& compare,
     Combine&& combine,
     WF&& weight = // default weight(uv) -> 1
          [](edge_reference_t<G> uv) { return ranges::range_value_t<Distances>(1); },
     Allocator alloc = Allocator());
```

[PHIL: Should negative weight cycles be a pre-condition, or should it be detected with an exception thrown when it exists?]

[PHIL: NetworkX has `negative_edge_cycle` and `find_negative_cycle`. These are needed if negative weight cycles are a pre-condition?]

1 *Preconditions:*

(1.1) — `0 <= source < num_vertices(graph)`.

(1.2) — `distance` will be initialized with `init_shortest_paths`.

(1.3) — `predecessors` will be initialized with `init_shortest_paths`.

2 *Effects:*

(2.1) — If vertex with index `i` is reachable from vertex `source`, then `distances[i]` will contain the distance from `source` to vertex `i`. Otherwise `distances[i]` will contain `shortest_path_invalid_distance()`.

(2.2) — If vertex with index `i` is reachable from vertex `source`, then `predecessors[i]` will contain the predecessor vertex of vertex `i`. Otherwise `predecessors[i]` will contain `i`.

3 *Throws:* `out_of_range` is thrown when `source` is not in the range `0 <= source < num_vertices(graph)`.

4 *Remarks:*

(4.1) — Unlike Dijkstra's algorithm, Bellman-Ford allows negative edge weights. Performance constraints limit this to smaller graphs.

2.4 Clustering

2.4.1 Triangle Counting

Compute the number of triangles in a graph.

Complexity		
Complexity $\mathcal{O}(N^3)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <index_adjacency_list G>
size_t triangle_count(G&& g);
```

1 *Returns:* Number of triangles

2 *Remarks:* To avoid duplicate counting, only directed triangles of a certain orientation will be detected. If `vertex_id(u)< vertex_id(v)< vertex_id(w)`, count triangle if graph contains edges `uv, vw, uw`.

2.5 Communities

2.5.1 Label Propagation

Propagate vertex labels by setting each vertex's label to the most popular label of its neighboring vertices. Every vertex voting on its new label represents one iteration of label propagation. Vertex voting order is randomized every iteration. The algorithm will iterate until label convergence, or optionally for a user specified number of iterations. Convergence occurs when no vertex label changes from the previous iteration. $\mathcal{O}(M)$ complexity is based on the complexity of one iteration, with number of iterations required for convergence considered small relative to graph size.

Some label propagation implementations use vertex ids as an initial labeling. This is not supported here because the label type can be more generic than the vertex id type. User is responsible for meaningful initial labeling.

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(M)$	**Multi-edge?** No	**Directed?** Yes

```
template <index_adjacency_list G,
        ranges::random_access_range Label,
        class Gen = default_random_engine,
        class T = size_t>
void label_propagation(G&& g,
                       Label& label,
                       Gen&& rng = default_random_engine {},
                       T max_iters = numeric_limits<T>::max());
```

1 *Preconditions:*

(1.1) — `label` contains initial vertex labels.

(1.2) — `rng` is a random number generator for vertex voting order.

(1.3) — `max_iters` is the maximum number of iterations of the label propagation, or equivalently the maximum distance a label will propagate from its starting vertex.

2 *Effects:* `label[uid]` is the label assignments of vertex id `uid` discovered by label propagation. *Remarks:* User is responsible for initial vertex labels.

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(M)$	**Multi-edge?** No	**Directed?** Yes

```
template <index_adjacency_list G,
        ranges::random_access_range Label,
        class Gen = default_random_engine
        class T = size_t>
void label_propagation(G&& g,
                       Label& label,
                       ranges::range_value_t<Label>& empty_label
                       Gen&& rng = default_random_engine {},
                       T max_iters = numeric_limits<T>::max());
```

4 *Preconditions:*

(4.1) — `label` contains initial vertex labels.

(4.2) — `empty_label` defines a label that is considered empty and will not be propagated.

(4.3) — `rng` is a random number generator for vertex voting order.

(4.4) — `max_iters` is the maximum number of iterations of the label propagation, or equivalently the maximum distance a label will propagate from its starting vertex.

5 *Effects:* `label[uid]` is the label assignments of vertex id `uid` discovered by label propagation. *Remarks:* User is responsible for initial vertex labels.

2.6 Components

2.6.1 Articulation Points

Find articulation points, or cut vertices, which when removed disconnect the graph into multiple components. Time complexity based on Hopcroft-Tarjan algorithm.

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(\|E\|+\|V\|)$	**Multi-edge?** No	**Directed?** Yes

```
template <index_adjacency_list G, class Iter, class Allocator = allocator<vertex_id<G>>>
requires output_iterator<Iter, vertex_id_t<G>>
void articulation_points(G&& g, Iter cut_vertices, Allocator alloc = Allocator());
```

[PHIL: Should target of output iterator be convertible to vertex_id, not same_as vertex_id?]

1 *Preconditions:*

(1.1) — Output iterator `cut_vertices` can be assigned vertices of type `vertex_id_t<G>` when dereferenced.

2 *Effects:*

(2.1) — Output iterator `cut_vertices` contains articulation point vertices, those which removed increase the number of components of `g`.

2.6.2 BiConnected Components

Find the biconnected components, or maximal biconnected subgraphs of a graph, which are components that will remain connected if a vertex is removed. Time complexity based on Hopcroft-Tarjan algorithm.

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(\|E\|+\|V\|)$	**Multi-edge?** No	**Directed?** Yes

```
template <index_adjacency_list G,
          ranges::forward_range OuterContainer,
          class Allocator = allocator<vertex_id<G>>
requires ranges::forward_range<ranges::range_value_t<OuterContainer>> &&
         integral<ranges::forward_range_t<ranges::forward_range_t<OuterContainer>>>
void biconnected_components(G&& g,
                            OuterContainer& components,
                            Allocator alloc = Allocator());
```

[PHIL: `push_back`, `push_front` and `insert` are all valid ways to add to containers that support forward_range, depending on the specific container type. Are all supported?]

[PHIL: I think `convertible_to<...,vertex_id<G>>` would be better than `integral<...>` because it will catch truncation when assigning vertex_id to smaller ints in the inner container.]

[PHIL: Is an allocator parameter needed?]

1 *Preconditions:*

(1.1) — `components` is a container of containers. The inner container stores vertex ids.

2 *Effects:*

(2.1) — `components` contains groups of biconnected components.

2.6.3 Connected Components

Find weakly connected components of a graph. Weakly connected components are subgraphs where a path exists between all pairs of vertices when ignoring edge direction.

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(\|E\|+\|V\|)$	**Multi-edge?** No	**Directed?** No

```
template <index_adjacency_list G,
        ranges::random_access_range Component,
        class Allocator = allocator<vertex_id<G>>
void connected_components(G&& g,
                          Component& component,
                          Allocator alloc = Allocator());
```

[PHIL: Return number of components `C` ? If `C==num_vertices(g)` then all components are of `size==1` (a.k.a. no components).]

[PHIL: Should there be an allocator parameter?]

1 *Preconditions:*

(1.1) — `size(component)>= num_vertices(g)`.

2 *Effects:*

(2.1) — `component[v]` is the connected component id of vertex `v`.

(2.2) — There is at least one Connected Component, with compondent id of `0`, for `num_vertices(g)> 0`.

2.6.4 Strongly Connected Components

2.6.4.1 Kosaraju's SCC

Find strongly connected components of a graph using Kosaraju's algorithm. Strongly connected components are subgraphs where a path exists between all pairs of vertices.

Complexity $\mathcal{O}(\|E\| + \|V\|)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <index_adjacency_list G,
        index_adjacency_list GT,
        ranges::random_access_range Component,
        class Allocator = allocator<vertex_id<G>>
void strongly_connected_components(G&& g,
                                   GT&& g_t,
                                   Component& component,
                                   Allocator alloc = Allocator());
```

[PHIL: Return number of components `C` ? If `C==num_vertices(g)` then all components are of `size==1` (a.k.a. no components).]

[PHIL: Should there be an allocator parameter?]

1 *Preconditions:*

(1.1) — `g_t` is the transpose of `g`. Edge `uv` in `g` implies edge `vu` in `g_t`. `num_vertices(g)` equals `num_vertices(g_t)`.

(1.2) — `size(component)>= num_vertices(g)`.

2 *Effects:*

(2.1) — `component[v]` is the strongly connected component id of vertex `v`.

2.6.4.2 Tarjan's SCC

Find strongly connected components of a graph using Tarjan's algorithm. Strongly connected components are subgraphs where a path exists between all pairs of vertices.

Complexity $\mathcal{O}(\|E\| + \|V\|)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <adjacency_list G,
```

```
        ranges::random_access_range Component,
        class Allocator = allocator<vertex_id<G>>>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
void strongly_connected_components(G&& g,
                                   Component& component
                                   Allocator alloc = Allocator());
```

[PHIL: Return number of components `C` ? If `C==num_vertices(g)` then all components are of `size==1` (a.k.a. no components).]

[PHIL: Should there be an allocator parameter?]

1 *Preconditions:*

(1.1) — `size(component)>= num_vertices(g)`.

2 *Effects:*

(2.1) — `component[v]` is the strongly connected component id of `v`.

2.7 Directed Acyclic Graphs

2.7.1 Topological Sort, Single Source

A linear ordering of vertices such that for every directed edge (u,v) from vertex u to vertex v, u comes before v in the ordering.

2.7.1.1 Initialization

```
template <class Predecessors>
constexpr void init_topological_sort(Predecessors& predecessors) {
  // exposition only
  size_t i = 0;
  for(auto& pred : predecessors)
    pred = i++;
}
```

Effects:

— Each `predecessors[i]` is initialized to `i`.

2.7.1.2 Topological Sort, Single Source

Complexity $\mathcal{O}((\|E\| + \|V\|))$	**Throws?** Yes **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <index_adjacency_list G,
        class Predecessors,
        class Allocator = allocator<vertex_id_t<G>>
void topological_sort(const G& graph,
                      vertex_id_t<G> source,
                      Predecessors& predecessors,
                      Allocator alloc = Allocator());
```

[PHIL: Add overload with `distances`]

1 *Preconditions:*

(1.1) — `0 <= source < num_vertices(graph)`.

(1.2) — `predecessors` will be initialized with `init_topological_sort`.

2 *Effects:*

(2.1) — If vertex with index `i` is reachable from vertex `source`, then `predecessors[i]` will contain the predecessor vertex of vertex `i`. Otherwise `predecessors[i]` will contain `i`.

3 *Throws:* `out_of_range` is thrown when `source` is not in the range `0 <= source < num_vertices(graph)`.

2.8 Maximal Independent Set

2.8.1 Maximal Independent Set

Find a maximally independent set of vertices in a graph starting from a seed vertex. An independent vertex set indicates no pair of vertices in the set are adjacent.

Complexity $\mathcal{O}(\|E\|)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** No

```
template <index_adjacency_list G, class Iter>
requires output_iterator<Iter, vertex_id_t<G>>
void maximal_independent_set(G&& g, Iter mis, vertex_id_t<G> seed);
```

1 *Preconditions:*

(1.1) — `0 <= seed < num_vertices(graph)`.

(1.2) — `mis` output iterator can be assigned vertices of type `vertex_id_t<G>` when dereferenced.

2 *Effects:*

(2.1) — Output iterator `mis` contains maximal independent set of vertices containing `seed`, which is a subset of `vertices(graph)`.

2.9 Link Analysis

2.9.1 Jaccard Coefficient

Calculate the Jaccard coefficient of a graph

Complexity $\mathcal{O}(\|N\|^3)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <index_adjacency_list G, typename OutOp, typename T = double>
requires is_invocable_v<OutOp, vertex_id_t<G>&, vertex_id_t<G>&, edge_reference_t<G>, T>
void jaccard_coefficient(G&& g, OutOp out);
```

[PHIL: Consider using `out(uid,vid,uv,val)` as the function descriptor in Preconditions to make it more readable.]

[PHIL: Would an output iterator be appropriate? `pair<edge_id_t<G>>,T>` might work for the output type. This starts to get into the same realm of the views as to whether the edge reference is useful by the consumer or not (e.g. basic_ vs. regular versions).]

1 *Preconditions:*

(1.1) — `out` is an operator for setting the resulting Jaccard coefficient. This function is expected to be of the form `out(vertex_id_t<G> uid, vertex_id_t<G> vid, edge_t<G> uv, T val)`.

2 *Effects:*

(2.1) — For every pair of neighboring vertices `(uid, vid)`, the function `out` is called, passing the vertex ids, the edge `uv` between them, and the calculated Jaccard coefficient.

2.10 Minimum Spanning Tree

2.10.1 Kruskal Minimum Spanning Tree

Find the minimum weight spanning tree of a graph using Kruskal's algorithm.

Complexity $\mathcal{O}(\lvert E \rvert)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

```
template <edgelist::edgelist E, edgelist::edgelist T>
void kruskal(E&& e, T&& t);

template <edgelist::edgelist E, edgelist::edgelist T, CompareOp>
void kruskal(E&& e, T&& t, CompareOp compare);
```

1 *Preconditions:*

(1.1) — `e` is an `edgelist`.

(1.2) — `compare` operator is a valid comparison operation on two edge values of type `edge_value_t<EL>` which returns a bool.

2 *Effects:*

(2.1) — Edgelist `t` contains edges representing a spanning tree or forest, which minimize the comparison operator. When `compare` is <, `t` represents a minimum weight spanning tree.

2.10.2 Prim Minimum Spanning Tree

Find the minimum weight spanning tree of a graph using Prim's algorithm.

[PHIL: Use general form of dijkstra's shortest path?]

Complexity $\mathcal{O}(\lvert E \rvert log \lvert V \rvert)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** No

```
template <index_adjacency_list G,
       ranges::random_access_range Predecessor,
       ranges::random_access_range Weight>
void prim(G&& g, Predecessor& predecessor, Weight& weight, vertex_id_t<G> seed = 0);

template <index_adjacency_list G,
       ranges::random_access_range Predecessor,
       ranges::random_access_range Weight,
       class CompareOp>
void prim(G&& g,
       Predecessor& predecessor,
       Weight& weight,
       CompareOp compare,
       ranges::range_value_t<Weight> init_dist,
       vertex_id_t<G> seed = 0);
```

1 *Preconditions:*

(1.1) — `0 <= seed < num_vertices(g)`.

(1.2) — Size of `weight` and `predecessor` is greater than or equal to `num_vertices(g)`.

(1.3) — `compare` operator is a valid comparison operation on two edge values of type `edge_value_t<G>` which returns a bool.

2 *Effects:*

(2.1) — `predecessor[v]` is the parent vertex of `v` in a tree rooted at `seed` and `weight[v]` is the value of the edge between `v` and `predecessor[v]` in the tree. When `compare` is < and `init_dist==+inf`, `predecessor` represents a minimum weight spanning tree.

(2.2) — If `predecessor` and `weight` are not initialized by user, and the graph is not fully connected, `predecessor[v]` and `weight[v]` will be undefined for vertices not in the same connected component as `seed`.

[ANDREW: I've tagged the algorithms below as Tier 2 or Tier 3 – denoting whether they should be done right now or done later or done much later.]

[ANDREW: I've used NetworkX as inspiration for organization. Oddly, NetworkX only has DFS as an adaptor (view).] [PHIL: If the use of Yield is any indicator, then NetworkX implements topological sort as adaptor also.]

2.11 Other Algorithms

Additional algorithms that were considered but not included in this proposal are identified in Table 2.1. It is assumed that future proposals will include them, with a recommendation of each Tier being in its own proposal. Tier X algorithms are variations of shortest paths algorithms that complement the Single Source, Multiple Target algorithms in this proposal.

The Shortest Paths Driver is an idea of having a unified interface that chooses the best Shortest Path algorithm based on characteristics like non-negative edge weight, multi-threading, etc.

Tier 2	**Tier 3**	**Tier X**
All Pairs Shortest Paths	Jones Plassman	Single Source, Single Target: Shortest Paths Driver
Floyd-Warshall	Cores: k-cores	Single Source, Single Target: BFS
Johnson	Cores: k-truss	Single Source, Single Target: Dijkstra
Centrality: Betweenness Centrality	Subgraph Isomorphism	Single Source, Single Target: Bellman-Ford
Coloring: Greedy		Single Source, Single Target: Delta Stepping
Communities: Louvain		
Connectivity: Minimum Cuts		Multiple Source: Shortest Paths Driver
Transitive Closure		Multiple Source: BFS
Flows: Edmunds Karp		Multiple Source: Dijkstra
Flows: Push Relabel		Multiple Source: Bellman-Ford
Flows: Boykov Kolmogorov		Multiple Source: Delta Stepping
		Multiple Source, Single Target: Shortest Paths Driver
		Multiple Source, Single Target: BFS
		Multiple Source, Single Target: Dijkstra
		Multiple Source, Single Target: Bellman-Ford
		Multiple Source, Single Target: Delta Stepping

Table 2.1 — Other Algorithms

[ANDREW: All Pairs: Tier 2? People bring this up alot – but it is very expensive in terms of computation and memory.] [PHIL: If it's useful to enough people it should be included. Users can make their own determination of whether they want to use it, based on the cost.]

[ANDREW: Note that NetworkX also specifies single source single target and multiple source versions of the shortest paths algorithms. BGL does not have these (nor NWGraph). We should discuss whether or not to consider those and whether or not to make them Tier 1, 2, 3, or infinity.] [PHIL: I think we're beyond considering these for the initial proposal. They can be added in the future, unless we're told otherwise.]

[PHIL: The same variations for Shortest Paths algorithms can also be useful for topological sort.]

[PHIL: Add Adamic-Adar Index to complement Jaccard? To Tier 2?]

Acknowledgements

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada. Portions of Andrew Lumsdaine's time was supported by NSF Award OAC-1716828 and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. The authors wish to further thank the members of SG19 for their contributions.

Bibliography

[1] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, Dec. 2001.