Graph Library: Algorithms

Document #:	**P3128r0**
Date:	2024-02-05
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency SG6 Numerics
Revises:	P1709r5

Reply-to: Phil Ratzloff (SAS Institute)
phil.ratzloff@sas.com
Andrew Lumsdaine
lumsdaine@gmail.com

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

1 Getting Started

This paper is one of several interrelated papers for a proposed Graph Library for the Standard C++ Library. The Table 1 describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now separated into the following papers.
P3126	Active	**Overview**, describing the big picture of what we are proposing.
P3127	Active	**Background and Terminology** providing the motivation, theoretical background and terminology used across the other documents.
P3128	Active	**Algorithms** covering the initial algorithms as well as the ones we'd like to see in the future.
P3129	Active	**Views** has helpful views for traversing a graph.
P3130	Active	**Graph Container Interface** is the core interface used for uniformly accessing graph data structures by views and algorithms. It is also designed to easily adapt to existing graph data structures.
P3131	Active	**Graph Containers** describing a proposed high-performance `compressed_graph` container. It also discusses how to use containers in the standard library to define a graph, and how to adapt existing graph data structures.

Table 1: Graph Library Papers

Reading them in order will give the best overall picture. If you're limited on time, you can use the following guide to focus on the papers that are most relevant to your needs.

Reading Guide

— If you're **new to the Graph Library**, we recommend starting with the *Overview* paper (P3126) to understand focus and scope of our proposals.

— If you want to **understand the theoretical background** that underpins what we're doing, you should read the *Background and Terminology* paper (P3127).

— If you want to **use the algorithms**, you should read the *Algorithms* paper (P3128) and *Graph Containers* paper (P3131).

— If you want to **write new algorithms**, you should read the *Views* paper (P3129), *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131). You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.

— If you want to **use your own graph container**, you should read the *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131).

2 Revision History

P3128r0

— Split from P1709r5. Added *Getting Started* section.

— Added A*, Best-first search and Adamic-Adar Index to Tier 2 algorithms based on input.

— Removed allocator parameters for consistency with existing algorithms. It was observed that `stable_sort` allocates memory, but does not take an allocator parameter.

— Removed exception throwing from algorithms to support free-standing C++. The caller will need to follow the preconditions to avoid undefined behavior. The other option considered was to return an error code.

3 Algorithm Introduction

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

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

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).

— **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.

— **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?

— **Self-loops?** Does the algorithm act act as expected if an edge exists with the same source and target?

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

4 Naming Conventions

Table 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	Vertex Iterator. ui is the source (or only) vertex.
		first,last	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 2: Naming Conventions for Types and Variables

5 Algorithm Selection

When determining the algorithms to propose we split them into different tiers. Tier 1 algorithms are included in this proposal. The algorithms selected are a result of balancing a few things:

— Include a rich enough set of algorithms for the library to be useful.

— Include algorithms with well-defined functionality and agreed-upon algorithmic description.

— Don't include so many that the proposal will get bogged down for years and years.

5.1 Tier 1 Algorithms

Shortest Paths

— Breadth-First search

— Dijkstra's algorithm

— Bellman-Ford

Clustering

— Triangle counting

Communities

— Label propagation

Components

— Articulation points

— Connected components

— Biconnected components

— Kosaraju's Strongly CC

— Tarjan's Strongly CC

Directed Acyclic Graphs

— Topological sort

Maximal Independent Set

— Maximal independent set

Link Analysis

— Jaccard coefficient

Minimal Spanning Tree

— Kruskal's MST

— Prim's MST

Shortest Paths and Topological Sort are all single source with multiple targets.

5.2 Other Algorithms

Additional algorithms that were considered but not included in this proposal are shown in Table 3. Tier X algorithms are variations of shortest paths algorithms that complement the Single Source, Multiple Target algorithms in this proposal. It is assumed that future proposals will include them, thought the exact mix for each proposal will depend on feedback received and our experience with the current proposal.

Parallel versions of algorithms will also be considered, keeping in mind that not all algorithms benefit from parallelism.

6 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>>>>;
```

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
Link Analysis: Adamic-Adar Index		
Pathfinding: A*		Multiple Source, Single Target: Shortest Paths Driver
Best-first search		Multiple Source, Single Target: BFS
		Multiple Source, Single Target: Dijkstra
		Multiple Source, Single Target: Bellman-Ford
		Multiple Source, Single Target: Delta Stepping

Table 3: Other Algorithms

```
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>>;
```

7 Shortest Paths

7.1 Unweighted Shortest Paths

7.1.1 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`.

7.1.2 Breadth-First Search, Single Source

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

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

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
          >
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

template <index_adjacency_list G,
          ranges::random_access_range Distances
          >
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
```

Preconditions:

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

— `distances` will be initialized with `init_breadth_first_search`.

— `predecessors` will be initialized with `init_breadth_first_search`.

1 *Effects:*

(1.1) — 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()`.

(1.2) — 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`.

7.2 Weighted Shortest Paths

7.2.1 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.

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

Note that complexity may be $\mathcal{O}(\lvert E\rvert+\lvert V\rvert\log\lvert V\rvert)$ 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>)>
          >
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); });

template <index_adjacency_list G,
          ranges::random_access_range Distances,
          class WF = function<ranges::range_value_t<Distances>(edge_reference_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); });
```

```
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>)>
          >
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); });

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>)>
          >
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); } );
```

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

7.2.3 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` .

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

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>)>
          >
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); })

template <index_adjacency_list G,
          ranges::random_access_range Distances,
          class WF = function<ranges::range_value_t<Distances>(edge_reference_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); });
```

```
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>)>
          >
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); });

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>)>
          >
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); });
```

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 *Remarks:*

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

8 Clustering

8.1 Triangle Counting

Compute the number of triangles in a graph.

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

```
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` .

9 Communities

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

```
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[3]. *Remarks:* User is responsible for initial vertex labels.

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

```
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.

10 Components

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

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

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`.

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

```
template <index_adjacency_list G,
        ranges::forward_range OuterContainer>
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);
```

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.

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

```
template <index_adjacency_list G,
          ranges::random_access_range Component>
void connected_components(G&& g,
                          Component& component);
```

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.

10.4 Strongly Connected Components

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

```
template <index_adjacency_list G,
          index_adjacency_list GT,
          ranges::random_access_range Component>
void strongly_connected_components(G&& g,
                                   GT&& g_t,
                                   Component& component);
```

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`.

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

```
template <adjacency_list G,
          ranges::random_access_range Component>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
void strongly_connected_components(G&& g,
                                   Component& component);
```

1 *Preconditions:*

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

2 *Effects:*

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

11 Directed Acyclic Graphs

11.1 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.

11.1.1 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`.

11.1.2 Topological Sort, Single Source

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

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

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`.

12 Maximal Independent Set

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

13 Link Analysis

13.1 Jaccard Coefficient

Calculate the Jaccard coefficient of a graph

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

```
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);
```

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.

14 Minimum Spanning Tree

14.1 Kruskal Minimum Spanning Tree

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

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

```
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.

14.2 Prim Minimum Spanning Tree

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

Complexity $\mathcal{O}(\|E\|log\|V\|)$	**Directed?** No **Multi-edge?** No	**Cycles?** No **Self-loops** No	**Throws?** 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`.

Acknowledgements

Phil Ratzloff's time was made possible by SAS Institute.

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.

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

Muhammad Osama's time was made possible by Advanced Micro Devices, Inc.

The authors thank the members of SG19 and SG14 study groups for their invaluable input.