Graph Library: Algorithms

Document #:	**P3128r2**
Date:	2024-09-12
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency SG6 Numerics
Revises:	P3128r1

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 (Intel) Jens Maurer Richard Dosselmann (University of Regina) Matthew Galati (Amazon) Guy Davidson (Creative Assembly) Oliver Rosten

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**, describes the big picture of what we are proposing.
P3127	Active	**Background and Terminology** provides the motivation, theoretical background, and terminology used across the other documents.
P3128	Active	**Algorithms** covers 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** describes 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.
P3337	Soon	**Comparison to other graph libraries** on performance and usage syntax.

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* (P3126) paper to understand the focus and scope of our proposals. You'll also want to check out it stacks up against other graph libraries in performance and usage syntax in the *Comparison* (P3337) paper.

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

— If you want to **use the algorithms**, you should read the *Algorithms* (P3128) and *Graph Containers* (P3131) papers. You may also find the *Views* (P3129) and *Graph Container Interface* (P3130) papers helpful.

— If you want to **write new algorithms**, you should read the *Views* (P3129), *Graph Container Interface* (P3130), and *Graph Containers* (P3131) papers. 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 data structures**, you should read the *Graph Container Interface* (P3130) and *Graph Containers* (P3131) papers.

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.

— Identify all `concept` definitions as "For exposition only" until we have consensus of whether they belong in the standard or not.

P3128r1

— Create new *Traversal* section and move Breadth-First Search and Topological Sort algorithms to it. Also added new Depth-First Search algorithm to it.

— Revise the Dijkstra and Bellman-Ford shortest-path and shortest-distance algorithms

  — Add a visitor parameter to allow the caller to customize the behavior of the algorithm without having to modify the algorithm itself. The visitor functions are member functions on a user-defined class that must match the concept for the event. The visitor events mimic those used in the Boost Graph Library for each algorithm.

  — Remove overloads that excluded Compare and Combine functions because they don't add much value, and to keep the proposal small.

  — Add overloads for multiple sources. This is particularly important for Bellman-Ford to avoid repeated calls to the algorithm that would make an already slow algorithm even slower.

  — Change "invalid distance" to "infinite distance" to reflect how the value is used in the algorithm.

— Add the ability to detect and find the negative weight cycle in the Bellman-Ford algorithm.

P3128r2

— Add Oliver Rosten as contributor.

3 Algorithm Introduction

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

Complexity $\mathcal{O}(\lvert E\rvert + \lvert V\rvert)$	**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
EL		el	Edge list
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 .
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> .

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

Traversal

— Breadth-First search

— Depth-First search

— Topological sort

Shortest Paths

— Dijkstra's algorithm

— Bellman-Ford algorithm

Clustering

— Triangle counting

Communities

— Label propagation

Components

— Articulation points

— Connected components

— Biconnected components

— Kosaraju's Strongly CC

— Tarjan's Strongly CC

Maximal Independent Set

— Maximal independent set

Link Analysis

— Jaccard coefficient

Minimal Spanning Tree

— Kruskal's MST

— Prim's MST

Traversal and Shortest Paths algorithms include single-source and multi-source versions 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.

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

6 Common Algorithm Definitions

Common concepts used by algorithms are in this section, extending those in the Graph Container Interface.

6.1 Edge Weight Concepts

Edge weights are intrinsic numeric type for the current proposal, but could be any type in the future.

```
// For exposition only
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>>>>;

// For exposition only
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>>;
```

6.2 Visitor Concepts and Classes

Visitors are optional member functions on a user-defined class that are called during the execution of an algorithm. Each algorithm has its own set of visitor events that it supports, and each event function must match the visitor concepts shown in this section.

The visitor events mimic those used in the Boost Graph Library.

6.2.1 Vertex Visitor Concepts

```
template <class G, class Visitor>
concept has_on_initialize_vertex = //
      requires(Visitor& v, vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, void> vdesc) {
        { v.on_initialize_vertex(vdesc) };
      };
template <class G, class Visitor>
concept has_on_discover_vertex = //
      requires(Visitor& v, vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, void> vdesc) {
        { v.on_discover_vertex(vdesc) };
      };
template <class G, class Visitor>
concept has_on_examine_vertex = //
      requires(Visitor& v, vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, void> vdesc) {
        { v.on_examine_vertex(vdesc) };
      };
template <class G, class Visitor>
concept has_on_finish_vertex = //
      requires(Visitor& v, vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, void> vdesc) {
        { v.on_finish_vertex(vdesc) };
      };
```

The vertex events are called under the following conditions.

— `on_initialize_vertex(vdesc)` is called once for each vertex before the algorithm is run.

— `on_discover_vertex(vdesc)` is called once for each source vertex passed to the algorithm.

— `on_examine_vertex(vdesc)` is called for a vertex before any of its outgoing edges are examined. It is possible that it will be called multiple times for the same vertex if paths are found to it from other vertices with a shorter distance.

— `on_finish_vertex(vdesc)` is called for vertex that is being examined, after all its outgoing edges have been examined.

6.2.2 Edge Visitor Concepts

```
template <class G, class Visitor>
concept has_on_examine_edge = //
      requires(Visitor& v, edge_descriptor<vertex_id_t<G>, true, edge_reference_t<G>, void> edesc) {
        { v.on_examine_edge(edesc) };
      };
template <class G, class Visitor>
concept has_on_edge_relaxed = //
      requires(Visitor& v, edge_descriptor<vertex_id_t<G>, true, edge_reference_t<G>, void> edesc) {
        { v.on_edge_relaxed(edesc) };
      };
template <class G, class Visitor>
concept has_on_edge_not_relaxed = //
      requires(Visitor& v, edge_descriptor<vertex_id_t<G>, true, edge_reference_t<G>, void> edesc) {
        { v.on_edge_not_relaxed(edesc) };
      };
template <class G, class Visitor>
concept has_on_edge_minimized = //
      requires(Visitor& v, edge_descriptor<vertex_id_t<G>, true, edge_reference_t<G>, void> edesc) {
        { v.on_edge_minimized(edesc) };
      };
template <class G, class Visitor>
concept has_on_edge_not_minimized =
      requires(Visitor& v, edge_descriptor<vertex_id_t<G>, true, edge_reference_t<G>, void> edesc) {
        { v.on_edge_not_minimized(edesc) };
      };
```

The edge events are called under the following conditions.

— `on_examine_edge(edesc)` is called for edge of the source vertex that is being examined.

— `on_edge_relaxed(edesc)` is called when the distance to the target vertex of the edge is relaxed, or decreased.

— `on_edge_not_relaxed(edesc)` is called when the distance to the target vertex of the edge is not relaxed, or not decreased.

— `on_edge_minimized(edesc)` is called when the distance to the target vertex of the edge is minimized, or decreased to the minimum value.

— `on_edge_not_minimized(edesc)` is called when the distance to the target vertex of the edge is not minimized, or not decreased to the minimum value.

6.2.3 Visitor Classes

`empty_visitor` is used when no visitor is needed. It is a no-op struct that does nothing.

```
struct empty_visitor {};
```

7 Traversal

7.1 Breadth-First Search

7.1.1 Initialization

```
template <class DistanceValue>
constexpr auto breadth_first_search_infinite_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
  fill(distances, breadth_first_search_infinite_distance<range_value_t<Distances>>());
}

template <class Predecessors>
constexpr void init_breadth_first_search(Predecessors& predecessors) {
  ranges::iota(predecessors, 0); // exposition only
}
```

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

```
template <index_adjacency_list G,
          random_access_range Distances,
          random_access_range Predecessors
          >
requires is_arithmetic_v<range_value_t<Distances>> &&
         convertible_to<vertex_id_t<G>, 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,
          random_access_range Distances
          >
requires is_arithmetic_v<range_value_t<Distances>>
void breadth_first_search(
      G&& g, // graph
      vertex_id_t<G> source, // starting vertex_id
      Distances& distances) // out: Distances[uid] of uid from seed in number of edges
```

Preconditions:

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

— `distances[i] = breadth_first_search_infinite_distance()` for `0 <= i < num_vertices(g)`.

— `predecessors[i] = i` for `0 <= i < num_vertices(g)`.

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_infinite_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`.

2 *Throws:*

(2.1) — `out_of_range` is thrown when `source` is not in the range `0 <= source < num_vertices(g)` .

3 *Complexity:*

(3.1) — $\mathcal{O}((|E| + |V|) \log |V|)$

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

7.2 Depth-First Search

Coming soon.

7.3 Topological Sort

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.

7.3.1 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`.

7.3.2 Topological Sort, Single Source

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

```
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`.

8 Shortest Paths

8.1 Initialization

```
template <class DistanceValue>
constexpr auto shortest_path_infinite_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_infinite_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);
  ranges::iota(predecessors, 0);
}
```

1 *Effects:*

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

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

2 *Returns:*

(2.1) — `shortest_path_infinite_distance()` returns the largest distance value, 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.

8.2 Dijkstra 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?** Yes

Note that complexity may be $\mathcal{O}(\lvert E\rvert + \lvert V\rvert\log\lvert V\rvert)$ for certain implementations that use a Fibonacci heap instead of a binary heap implemented with `std::priority_queue`.

8.2.1 Dijkstra Shortest Paths

8.2.1.1 Single-Source Shortest Paths

```
template <index_adjacency_list G,
          random_access_range Distances,
          random_access_range Predecessors,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires is_arithmetic_v<range_value_t<Distances>> &&
        sized_range<Distances> &&
        sized_range<Predecessors> &&
        convertible_to<vertex_id_t<G>, range_value_t<Predecessors>> &&
        basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr void dijkstra_shortest_paths(
     G&& g,
     const vertex_id_t<G> source,
     Distances& distances,
     Predecessors& predecessor,
     WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
     Visitor&& visitor = empty_visitor(),
     Compare&& compare = less<range_value_t<Distances>>(),
     Combine&& combine = plus<range_value_t<Distances>>());
```

8.2.1.2 Multi-Source Shortest Paths

```
template <index_adjacency_list G,
          input_range Sources,
          random_access_range Distances,
          random_access_range Predecessors,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
        is_arithmetic_v<range_value_t<Distances>> &&
        sized_range<Distances> &&
        sized_range<Predecessors> &&
        convertible_to<vertex_id_t<G>, range_value_t<Predecessors>> &&
        basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr void dijkstra_shortest_paths(
     G&& g,
     const Sources& sources,
     Distances& distances,
     Predecessors& predecessor,
     WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
     Visitor&& visitor = empty_visitor(),
     Compare&& compare = less<range_value_t<Distances>>(),
     Combine&& combine = plus<range_value_t<Distances>>());
```

1 *Mandates:*

(1.1) — `0 <= source < num_vertices(graph)` for the single-source version.

(1.2) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

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

2 *Preconditions:*

(2.1) — `distances[i] = shortest_path_infinite_distance()` for `0 <= i < num_vertices(g)`.

(2.2) — `predecessors[i] = i` for `0 <= i < num_vertices(g)`.

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_infinite_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`.

(3.3) — Member functions on the `visitor` parameter are called during the algorithm's execution. The functions are optional and, when included, must follow the visitor concepts for the events. No overhead is incurred if the functions are not included. The events supported are `on_initialize_vertex`, `on_discover_vertex`, `on_examine_vertex`, `on_finish_vertex`, `on_examine_edge`, `on_edge_relaxed`, and `on_edge_not_relaxed`.

4 *Throws:*

(4.1) — An `out_of_range` exception is thrown in the following cases:

(4.1.1) — `size(distances)< size(vertices(g))`

(4.1.2) — `size(predecessor)< size(vertices(g))`

(4.1.3) — `source` is not in the range `0 <= source < num_vertices(graph)`.

(4.1.4) — The weight function returns a negative value. This check is not made if the weight value type is an unsigned integral type.

5 *Complexity:*

(5.1) — $\mathcal{O}((|E| + |V|) \log |V|)$ based on using the binary heap in `std::priority_queue` .

(5.2) — An implementation may choose to use a Fibonacci heap for a complexity of $\mathcal{O}(|E| + |V| \log |V|)$.

6 *Remarks:*

(6.1) — Duplicate sources do not affect the algorithm's complexity or correctness.

(6.2) — Bellman-Ford Shortest Paths allows negative weights with the consequence of greater complexity.

8.2.2 Dijkstra Shortest Distances

This is the same as *Shortest Paths* except that it excludes the predecessors, giving a small performance improvement with lower memory overhead.

8.2.2.1 Single-Source Shortest Distances

```
template <index_adjacency_list G,
          random_access_range Distances,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires is_arithmetic_v<range_value_t<Distances>> &&
         sized_range<Distances> &&
         basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr void dijkstra_shortest_distances(
      G&& g,
      const vertex_id_t<G> source,
      Distances& distances,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

8.2.2.2 Multi-Source Shortest Distances

```
template <index_adjacency_list G,
          input_range Sources,
          random_access_range Distances,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         sized_range<Distances> &&
         is_arithmetic_v<range_value_t<Distances>> &&
         basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr void dijkstra_shortest_distances(
      G&& g,
      const Sources& sources,
      Distances& distances,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

1 *Mandates:*

(1.1) — `0 <= source < num_vertices(graph)` for the single-source version.

(1.2) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

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

2 *Preconditions:*

(2.1) — `distances[i] = shortest_path_infinite_distance()` for `0 <= i < num_vertices(g)`.

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_infinite_distance()`.

(3.2) — Member functions on the `visitor` parameter are called during the algorithm's execution. The functions are optional and, when included, must follow the visitor concepts for the events. No overhead is incurred if the functions are not included. The events supported are `on_initialize_vertex`, `on_discover_vertex`, `on_examine_vertex`, `on_finish_vertex`, `on_examine_edge`, `on_edge_relaxed`, and `on_edge_not_relaxed`.

4 *Throws:*

(4.1) — An `out_of_range` exception is thrown in the following cases:

(4.1.1) — `size(distances)< size(vertices(g))`

(4.1.2) — `source` is not in the range `0 <= source < num_vertices(graph)`.

(4.1.3) — The weight function returns a negative value. This check is not made if the weight value type is an unsigned integral type.

5 *Complexity:*

(5.1) — $\mathcal{O}((|E| + |V|) \log |V|)$ based on using the binary heap in `std::priority_queue` .

(5.2) — An implementation may choose to use a Fibonacci heap for a complexity of $\mathcal{O}(|E| + |V| \log |V|)$.

6 *Remarks:*

(6.1) — Duplicate sources do not affect the algorithm's complexity or correctness.

(6.2) — Bellman-Ford Shortest Distances allows negative weights with the consequence of greater complexity.

8.3 Bellman-Ford Shortest Paths and Shortest Distances

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

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

The Bellman-Ford algorithm supports the use of negative edge weights, at cost in performance. Because of its complexity, it can only be used for small graphs. If a user can guarantee that a graph has positive edge weights then Dijkstra's algorithm provides far better performance.

There is a special case where edges form a negative cycle. "If a graph contains a 'negative cycle' (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle." **Wikipedia** ([**?**])

`find_negative_cycle` can be called after calling `bellman_ford_shortest_paths` to get the vertex ids of the negative weight cycle.

8.3.1 Bellman-Ford Shortest Paths

8.3.1.1 Single-Source Shortest Paths

```
template <index_adjacency_list G,
          random_access_range Distances,
          random_access_range Predecessors,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires is_arithmetic_v<range_value_t<Distances>> &&
         convertible_to<vertex_id_t<G>, range_value_t<Predecessors>> &&
         sized_range<Distances> &&
         sized_range<Predecessors> &&
         basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr optional<vertex_id_t<G>> bellman_ford_shortest_paths(
      G&& g,
      const vertex_id_t<G> source,
      Distances& distances,
      Predecessors& predecessor,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

8.3.1.2 Multi-Source Shortest Paths

```
template <index_adjacency_list G,
          input_range Sources,
          random_access_range Distances,
          random_access_range Predecessors,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         is_arithmetic_v<range_value_t<Distances>> &&
         convertible_to<vertex_id_t<G>, range_value_t<Predecessors>> &&
         sized_range<Distances> &&
```

```
        sized_range<Predecessors> &&
        basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr optional<vertex_id_t<G>> bellman_ford_shortest_paths(
      G&& g,
      const Sources& sources,
      Distances& distances,
      Predecessors& predecessor,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

1 *Mandates:*

(1.1) — `0 <= source < num_vertices(graph)` for the single-source version.

(1.2) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

2 *Preconditions:*

(2.1) — `distances[i] = shortest_path_infinite_distance()` for `0 <= i < num_vertices(g)`.

(2.2) — `predecessors[i] = i` for `0 <= i < num_vertices(g)`.

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_infinite_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`.

(3.3) — Member functions on the `visitor` parameter are called during the algorithm's execution. The functions are optional and, when included, must follow the visitor concepts for the events. No overhead is incurred if the functions are not included. The events supported are `on_examine_edge`, `on_edge_relaxed`, `on_edge_not_relaxed`, `on_edge_minimized`, and `on_edge_not_minimized`.

4 *Returns:*

(4.1) — `optional<vertex_id_t<G>>` If no negative weight cycle is found, there is no associated vertex id. If a negative weight cycle is found, a vertex id in the cycle is returned. `find_negative_cycle` can be called to get the vertex ids of the cycle.

5 *Throws:*

(5.1) — An `out_of_range` exception is thrown in the following cases:

(5.1.1) — `size(distances)< size(vertices(g))`

(5.1.2) — `source` is not in the range `0 <= source < num_vertices(graph)`.

6 *Complexity:* $\mathcal{O}(|E| \cdot |V|)$. Complexity may also be affected when visitor events are called.

7 *Remarks:*

(7.1) — Duplicate sources do not affect the algorithm's complexity or correctness.

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

8.3.2 Bellman-Ford Shortest Distances

This is the same as *Shortest Paths* except that it excludes the predecessors, giving a small performance improvement with lower memory overhead.

8.3.2.1 Single-Source Shortest Distances

```
template <index_adjacency_list G,
          random_access_range Distances,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires is_arithmetic_v<range_value_t<Distances>> &&
         sized_range<Distances> &&
         basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
constexpr optional<vertex_id_t<G>> bellman_ford_shortest_distances(
      G&& g,
      const vertex_id_t<G> source,
      Distances& distances,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

8.3.2.2 Multi-Source Shortest Distances

```
template <index_adjacency_list G,
          input_range Sources,
          random_access_range Distances,
          class WF = function<range_value_t<Distances>(edge_reference_t<G>)>,
          class Visitor = empty_visitor,
          class Compare = less<range_value_t<Distances>>,
          class Combine = plus<range_value_t<Distances>>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         is_arithmetic_v<range_value_t<Distances>> &&
         sized_range<Distances> &&
         basic_edge_weight_function<G, WF, range_value_t<Distances>, Compare, Combine>
[[nodiscard]] constexpr optional<vertex_id_t<G>> bellman_ford_shortest_distances(
      G&& g,
      const Sources& sources,
      Distances& distances,
      WF&& weight = [](edge_reference_t<G> uv) { return range_value_t<Distances>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<range_value_t<Distances>>(),
      Combine&& combine = plus<range_value_t<Distances>>());
```

1 *Mandates:*

(1.1) — `0 <= source < num_vertices(graph)` for the single-source version.

(1.2) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

2 *Preconditions:*

(2.1) — `distances[i] = shortest_path_infinite_distance()` for `0 <= i < num_vertices(g)`.

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_infinite_distance()`.

(3.2) — Member functions on the `visitor` parameter are called during the algorithm's execution. The functions are optional and, when included, must follow the visitor concepts for the events. No overhead is incurred if the functions are not included. The events supported are `on_examine_edge`, `on_edge_relaxed`, `on_edge_not_relaxed`, `on_edge_minimized`, and `on_edge_not_minimized`.

4 *Returns:*

(4.1) — `optional<vertex_id_t<G>` If no negative weight cycle is found, there is no associated vertex id. If a negative weight cycle is found, a vertex id in the cycle is returned. `bellman_ford_shortest_paths` must be used to get the predecessors if it is importantant to get the vertex ids of the cycle using `find_negative_cycle`.

5 *Throws:*

(5.1) — An `out_of_range` exception is thrown in the following cases:

(5.1.1) — `size(distances)< size(vertices(g))`

(5.1.2) — `source` is not in the range `0 <= source < num_vertices(graph)`.

6 *Complexity:* $\mathcal{O}(|E| \cdot |V|)$. Complexity may also be affected when visitor events are called.

7 *Remarks:*

(7.1) — Duplicate sources do not affect the algorithm's complexity or correctness.

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

8.3.3 Finding the Negative Cycle

If a cycle with negative weights is found, it's possible to get the vertex ids of the cycle using `find_negative_cycle` after calling `bellman_ford_shortest_paths` . It is not possible to get the cycle from `bellman_ford_shortest_distances` because it does not evaluate predecessors.

```
template <index_adjacency_list G, forward_range Predecessors, class OutputIterator>
requires output_iterator<OutputIterator, vertex_id_t<G>>
void find_negative_cycle(const G& g,
                         const Predecessors& predecessor,
                         const optional<vertex_id_t<G>>& cycle_vertex_id,
                         OutputIterator out_cycle);
```

1 *Preconditions:*

(1.1) — `predecessors` must be evaluated by `bellman_ford_shortest_paths`.

(1.2) — `cycle_vertex_id` is the return value of `bellman_ford_shortest_paths`.

2 *Effects:*

(2.1) — All vertex ids in the negative weight cycle are written to the `out_cycle` output iterator.

3 *Complexity:* $\mathcal{O}(|E| + |V|)$

9 Clustering

9.1 Triangle Counting

Compute the number of triangles in a graph.

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

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

1 *Preconditions:* The outgoing edges of a vertex are ordered by target_id.
2 *Returns:* Number of triangles
3 *Throws:* A `graph_error` is thrown when the target_id for an outgoing edge is less than the target_id of the previous edge.
4 *Complexity:* $\mathcal{O}(N^3)$
5 *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` .

10 Communities

10.1 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	**Directed?** Yes	**Cycles?** Yes	**Throws?** No
$\mathcal{O}(M)$	**Multi-edge?** Yes	**Self-loops** 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.
3 *Complexity:* $\mathcal{O}(M)$
4 *Remarks:* User is responsible for initial vertex labels.

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

5 *Preconditions:*

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

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

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

(5.4) — `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.

6 *Effects:* `label[uid]` is the label assignments of vertex id `uid` discovered by label propagation.

7 *Complexity:* $\mathcal{O}(M)$

8 *Remarks:* User is responsible for initial vertex labels.

11 Components

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

```
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`.

3 *Complexity:* $\mathcal{O}(\lvert E\rvert + \lvert V\rvert)$

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

```
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.

3 *Complexity:* $\mathcal{O}(|E| + |V|)$

11.3 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
$\mathcal{O}(\|E\| + \|V\|)$	**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`.

3 *Complexity:* $\mathcal{O}(|E| + |V|)$

11.4 Strongly Connected Components

11.4.1 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
$\mathcal{O}(\|E\| + \|V\|)$	**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`.

3 *Complexity:* $\mathcal{O}(|E| + |V|)$

11.4.2 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`.

3 *Complexity:* $\mathcal{O}(|E| + |V|)$

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

3 *Complexity:* $\mathcal{O}(|E|)$

13 Link Analysis

13.1 Jaccard Coefficient

Calculate the Jaccard coefficient of a graph

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

3 *Complexity:* $\mathcal{O}(|N|^3)$

14 Minimum Spanning Tree

14.1 Kruskal Minimum Spanning Tree

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

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

```
template <index_edgelist_range IELR, index_edgelist_range OELR>
void kruskal(IELR&& e, OELR&& t);

template <index_edgelist_range IELR, index_edgelist_range OELR T, CompareOp>
void kruskal(IELR&& e, OELR&& 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 range_value_t<EL>::value_type 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.

3 *Complexity:* $\mathcal{O}(|E|)$

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

14.2 Prim Minimum Spanning Tree

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

```
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`.

3 *Complexity:* $\mathcal{O}(|E|log|V|)$

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.