Graph Library: Algorithms

Document #:	**P3128r4**
Date:	2026-07-06
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency
Revises:	P3128r3

Reply-to:	Phil Ratzloff (SAS Institute) phil.ratzloff@sas.com Andrew Lumsdaine lumsdaine@gmail.com
Contributors:	Kevin Deweese Muhammad Osama (AMD, Inc) Scott McMillan (Carnegie Mellon University) 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	Active	**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 how 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 freestanding 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.

P3128r3

— Add a note that we will be unable to support a freestanding graph library in this proposal because of the need for `stack`, `queue` and potential `bad_alloc` exception in many of the algorithms.

— Restore use of exceptions in the algorithms since a freestanding implementation cannot supported.

— Add depth-first search algorithm to the Traversal section.

— Add visitor events for the depth-first search algorithm.

— Update the breadth-first search and topological sort algorithms to be in line with changes made to the Dijkstra and Bellman-Ford algorithms, such as adding a visitor parameter and overloads for multiple sources.

P3128r4

— Changed C++ version requirements from C++20 to C++23 throughout all documents.

— Adopt value-based descriptor design throughout: `vertex_t<G>` and `edge_t<G>` value types replace the former `vertex_reference_t<G>` and `edge_reference_t<G>`. Also added views `vertex_descriptor_view_t<G>` and `edge_descriptor_view_t<G>` to provide consistent range wrapper for underlying vertex and edge ranges.

— Add `ordered_vertex_edges<G>` semantic concept for algorithms requiring sorted adjacency.

— Add hardened preconditions section to each algorithm for C++26. Reword existing clauses to conform to standard expectations; move statements formerly in *Mandates* and *Preconditions* to *Hardened preconditions*; add *Throws* clauses where appropriate.

— Include space complexity in addition to time complexity to every algorithm section.

— New algorithm: Minimum Spanning Tree: Inplace Kruskal.

— Replace raw container (range) predecessor and distance parameters with function objects (`PredecessorFn`, `DistanceFn`) constrained by new `predecessor_fn_for` and `distance_fn_for` concepts, enabling flexible storage strategies for per-vertex properties. Add `container_value_fn` adaptor and `_null_predecessor` sentinel for convenience.

— Add `vertex_property_map` facility: a type alias, factory functions, access helpers, a value-type trait, and a concept for uniform per-vertex storage across index and mapped graphs.

— Add an allocator parameter to algorithms that require internal containers such as a `stack` or `queue`.

— Remove items in the algorithm summary box: *Directed?* applies to the underlying characteristics of the graph, not the algorithm itself, and *Multi-edge?* handling is described in the algorithm remarks section.

3 Algorithm Introduction

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

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

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

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

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

We are unable to support freestanding implementations in this proposal. Many of the algorithms require a `stack` or `queue`, which are not available in a freestanding environment. Additionally, `stack` and `queue` require memory allocation which could throw a `bad_alloc` exception.

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>	u,v	Vertex descriptor. u is the source (or only) vertex. v is the target vertex.
VId	vertex_id_t<G>	uid,vid,source	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 iterator. vi is the target vertex iterator.
		first,last	first and last are the begin and end iterators of a vertex range.
VVF		vvf	Vertex Value Function: vvf(g, u) → vertex value, or vvf(g, uid) → vertex value, depending on requirements of the consuming algorithm or view.
VProj		vproj	Vertex data projection function: vproj(u) → vertex_data<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>	uv,vw	Edge descriptor. 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(g, uv) → edge value.
EProj		eproj	Edge data projection function: eproj(uv) → edge_data<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, though the exact mix for each proposal will depend on feedback received and our experience with the current proposal.

[Phil: We may want to revisit the Driver idea in the future after we have more algorithms.]

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

[Andrew: All Pairs: Tier 2? People bring this up a lot – 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.]

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

6 Common Algorithm Definitions

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

6.1 Value Function and Edge Weight Concepts

Value function concepts constrain callables that extract per-vertex or per-edge values. Edge weights are intrinsic numeric type (`edge_weight_function`), but could be any type for more general use (`basic_edge_weight_function`).

```
// For exposition only:  base concept for any 2-arg vertex value function.
template <class VVF, class Graph, class VertexDescriptor>
concept vertex_value_function =
    invocable<VVF, const Graph&, VertexDescriptor> &&
    (!is_void_v<invoke_result_t<VVF, const Graph&, VertexDescriptor>>);

// For exposition only:  base concept for any 2-arg edge value function.
template <class WF, class G, class E>
concept edge_value_function = // For exposition only
     std::invocable<WF, const G&, E> && !std::is_void_v<std::invoke_result_t<WF, const G&, E>>;

// For exposition only
template <class G, class WF, class DistanceValue, class Compare, class Combine>
concept basic_edge_weight_function = // e.g.  weight(g, uv)
     edge_value_function<WF, std::remove_reference_t<G>, edge_t<G>> &&
     strict_weak_order<Compare, DistanceValue, DistanceValue> &&
     assignable_from<add_lvalue_reference_t<DistanceValue>,
          invoke_result_t<Combine, DistanceValue,
               invoke_result_t<WF, const std::remove_reference_t<G>&, edge_t<G>>>>;

// For exposition only
template <class G, class WF, class DistanceValue>
concept edge_weight_function = // e.g.  weight(g, uv)
     is_arithmetic_v<DistanceValue> &&
     is_arithmetic_v<invoke_result_t<WF, const std::remove_reference_t<G>&, edge_t<G>>> &&
     basic_edge_weight_function<G, WF, DistanceValue, less<DistanceValue>, plus<DistanceValue>>;
```

6.2 Ordered Vertex Edges Concept

The `ordered_vertex_edges` concept requires that the outgoing edges of each vertex are sorted in ascending order of `target_id`. This is needed by algorithms such as `triangle_count` that rely on sorted adjacency lists for efficient intersection. This is a semantic requirement that is not enforced by the graph container interface, but it is documented as a requirement for algorithms that need it.

```
// For exposition only
template <class G>
concept ordered_vertex_edges = adjacency_list<G> &&
  requires(G& g, vertex_t<G> u) {
    requires forward_iterator<decltype(ranges::begin(out_edges(g, u)))>;
  };
```

6.3 Vertex Property Function Concepts

Most algorithms maintain per-vertex state (distances, predecessors, component labels, etc.) through callable *function objects* of the form `(const G&, vertex_id_t<G>)-> value&`. This function-first design is more flexible than direct container access: property values may reside in vertex properties, in external containers, or in any custom storage.

6.3.1 `vertex_property_fn_for` and `vertex_fn_value_t`

`vertex_property_fn_for<VF,G>` is the general concept for per-vertex property functions, requiring a callable that returns a mutable lvalue reference to the stored value. It is used directly by `connected_components`,

kosaraju, tarjan_scc, and label_propagation. vertex_fn_value_t<VF,G> extracts the corresponding value type.

```
// For exposition only
template <class VF, class G>
concept vertex_property_fn_for =
     invocable<VF&, const remove_reference_t<G>&, const vertex_id_t<G>&> &&
     is_lvalue_reference_v<invoke_result_t<VF&, const remove_reference_t<G>&, const vertex_id_t<G
         >&>>;

// For exposition only
template <class VF, class G>
using vertex_fn_value_t = remove_cvref_t<
    invoke_result_t<VF&, const remove_reference_t<G>&, const vertex_id_t<G>&>>;
```

6.3.2 distance_fn_for and predecessor_fn_for

distance_fn_for and predecessor_fn_for are concept aliases for vertex_property_fn_for, giving semantic names to the distance and predecessor function parameters of shortest-path and MST algorithms. distance_fn_value_t and predecessor_fn_value_t are corresponding type aliases for vertex_fn_value_t.

```
// For exposition only
template <class DF, class G>
concept distance_fn_for = vertex_property_fn_for<DF, G>;

// For exposition only
template <class DF, class G>
using distance_fn_value_t = vertex_fn_value_t<DF, G>;

// For exposition only
template <class PF, class G>
concept predecessor_fn_for = vertex_property_fn_for<PF, G>;

// For exposition only
template <class PF, class G>
using predecessor_fn_value_t = vertex_fn_value_t<PF, G>;
```

6.3.3 container_value_fn Adaptor

container_value_fn<Container> adapts any subscriptable container into a property function, making it straightforward to reuse a std::vector or std::unordered_map:

```
std::vector<double> dist_vec(num_vertices(g), infinite_distance<double>());
auto dist_fn = container_value_fn{dist_vec}; // dist_fn(g, uid) -> dist_vec[uid] (by ref)
dijkstra_shortest_paths(g, source, dist_fn, _null_predecessor, weight_fn);
```

6.3.4 Legacy: vertex_property_map_for Concept

The container-subscript concept vertex_property_map_for<M,G> is retained for the init_shortest_paths helper functions, which take subscriptable containers directly. It is satisfied by std::vector<T> for index graphs and std::unordered_map<vertex_id_t<G>,T> for mapped graphs.

6.4 Vertex Property Map

Algorithms frequently need per-vertex storage for quantities such as distances, predecessor IDs, component labels, and visited flags. Because the graph library supports both *index-based* vertex containers (std::vector, std::deque) and *key-based* vertex containers (std::map, std::unordered_map), the appropriate external container differs:

— For index graphs, a std::vector<T> addressed by vertex_id_t<G> provides $\mathcal{O}(1)$ access.

— For mapped graphs, a `std::unordered_map<vertex_id_t<G>, T>` is required because vertex IDs are non-contiguous keys.

The `vertex_property_map` facility provides a uniform abstraction that selects the correct container automatically and offers factory functions and access helpers so that algorithms and users need not distinguish between the two cases.

6.4.1 Synopsis

```
// vertex_property_map:  per-vertex associative container
// vector for index graphs, unordered_map for mapped graphs.
template <class G, class T>
using vertex_property_map = conditional_t<
    index_vertex_range<G>,
    vector<T>,
    unordered_map<vertex_id_t<G>, T>>;

// Eager initialization:  every vertex pre-populated with init_value.  O(V).
template <class G, class T>
constexpr auto make_vertex_property_map(const G& g, const T& init_value);

// Lazy initialization:  sized for index graphs, empty and reserved for mapped graphs.
template <class G, class T>
constexpr auto make_vertex_property_map(const G& g);

// Test whether a vertex ID has an entry in the map.
template <class Map, class Key>
constexpr bool vertex_property_map_contains(const Map& m, const Key& uid);

// Read with default fallback; no insertion for unordered_map.
template <class Map, class Key, class T>
constexpr auto vertex_property_map_get(const Map& m, const Key& uid,
                                       const T& default_val);

// Extract the per-vertex value type from a vertex_property_map container.
// For vector:  value_type.  For unordered_map:  mapped_type.
template <class Container>
using vertex_property_map_value_t =
    typename detail::vertex_property_map_value_impl<Container>::type;

// Concept:  container usable as a per-vertex property map for graph G.
// Requires subscript access with the graph's vertex ID type, returning a value
// convertible to the map's value type.
template <class M, class G>
concept vertex_property_map_for = requires(M& m, const vertex_id_t<G>& uid) {
  { m[uid] } -> convertible_to<vertex_property_map_value_t<M>>;
};
```

6.4.2 `vertex_property_map<G, T>` Type Alias

A conditional type alias that resolves to `std::vector<T>` when the graph `G` satisfies `index_vertex_range<G>`, and to `std::unordered_map<vertex_id_t<G>, T>` otherwise.

6.4.3 `make_vertex_property_map` Factory Functions

Two overloads create a `vertex_property_map`:

— **Eager initialization** — `make_vertex_property_map(g, init_value)` pre-populates every vertex with `init_value`. Useful when an algorithm reads all entries before writing (e.g. component labels). $\mathcal{O}(|V|)$.

— **Lazy initialization** — `make_vertex_property_map<G, T>(g)` creates a default-constructed (index graphs) or empty-but-reserved (mapped graphs) container. Useful with `vertex_property_map_get` when absence has a semantic meaning (e.g. infinity for distances, `false` for visited flags).

6.4.4 Access Helpers

— `vertex_property_map_contains(m, uid)` — Tests whether vertex `uid` has an entry. For `vector`: `uid < size(m)`. For `unordered_map`: `m.contains(uid)`.

— `vertex_property_map_get(m, uid, default_val)` — Returns the stored value if present, otherwise returns `default_val` *without insertion*. This avoids accidentally growing an `unordered_map` through `operator[]`.

6.4.5 `vertex_property_map_value_t<Container>`

A trait that extracts the per-vertex value type from a `vertex_property_map` container: `T` for `vector<T>`, `V` for `unordered_map<K,V>`.

6.4.6 `vertex_property_map_for<M, G>` Concept

Requires that `m[uid]` is valid for `uid` of type `vertex_id_t<G>` and the result is convertible to `vertex_property_map_value_t<M>`. This concept constrains algorithm parameters that accept subscriptable containers directly, such as the `init_shortest_paths` helpers.

6.5 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.

6.5.1 Vertex Visitor Concepts

```
template <class G, class Visitor>
concept has_on_initialize_vertex = // For exposition only
     requires(Visitor& v, const G& g, const vertex_t<G>& u) {
       { v.on_initialize_vertex(g, u) };
     };
template <class G, class Visitor>
concept has_on_discover_vertex = // For exposition only
     requires(Visitor& v, const G& g, const vertex_t<G>& u) {
       { v.on_discover_vertex(g, u) };
     };
template <class G, class Visitor>
concept has_on_start_vertex = // For exposition only
     requires(Visitor& v, const G& g, const vertex_t<G>& u) {
       { v.on_start_vertex(g, u) };
     };
template <class G, class Visitor>
concept has_on_examine_vertex = // For exposition only
     requires(Visitor& v, const G& g, const vertex_t<G>& u) {
       { v.on_examine_vertex(g, u) };
     };
template <class G, class Visitor>
concept has_on_finish_vertex = // For exposition only
     requires(Visitor& v, const G& g, const vertex_t<G>& u) {
       { v.on_finish_vertex(g, u) };
     };
```

Each vertex event concept requires an event function whose second argument is `const vertex_t<G>&`, with the first argument being `const G&`.

The vertex events are called under the following conditions.

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

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

— `on_examine_vertex(g, u)` 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_start_vertex(g, u)` is called for a vertex that is being examined.

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

6.5.2 Edge Visitor Concepts

```
template <class G, class Visitor>
concept has_on_examine_edge = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_examine_edge(g, e) };
     };
template <class G, class Visitor>
concept has_on_edge_relaxed = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_edge_relaxed(g, e) };
     };
template <class G, class Visitor>
concept has_on_edge_not_relaxed = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_edge_not_relaxed(g, e) };
     };
template <class G, class Visitor>
concept has_on_edge_minimized = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_edge_minimized(g, e) };
     };
template <class G, class Visitor>
concept has_on_edge_not_minimized = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_edge_not_minimized(g, e) };
     };
template <class G, class Visitor>
concept has_on_tree_edge = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_tree_edge(g, e) };
     };
template <class G, class Visitor>
concept has_on_back_edge = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_back_edge(g, e) };
     };
template <class G, class Visitor>
concept has_on_forward_or_cross_edge = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_forward_or_cross_edge(g, e) };
     };
template <class G, class Visitor>
concept has_on_finish_edge = // For exposition only
     requires(Visitor& v, const G& g, const edge_t<G>& e) {
       { v.on_finish_edge(g, e) };
     };
```

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.

— `on_tree_edge(edesc)` is called when the edge is added to the tree.

— `on_back_edge(edesc)` is called when the edge is a back edge.

— `on_forward_or_cross_edge(edesc)` is called when the edge is a forward or cross edge.

— `on_finish_edge(edesc)` is called when the edge is finished being examined.

6.5.3 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

Compute the breadth-first path from one or more `source` vertices to all reachable vertices in `graph`.

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

7.1.1 Single Source Breadth-First Search

```
template <adjacency_list G, class Visitor = empty_visitor,
          class Alloc = allocator<byte>>
void breadth_first_search(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Visitor&& visitor = empty_visitor(),
     const Alloc& alloc = Alloc())
```

7.1.2 Multi-Source Breadth-First Search

```
template <adjacency_list G, input_range Sources, class Visitor = empty_visitor,
          class Alloc = allocator<byte>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>>
void breadth_first_search(G&& g, // graph
                          const Sources& sources,
                          Visitor&& visitor = empty_visitor(),
                          const Alloc& alloc = Alloc())
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — For the multi-source version, `Sources` satisfies `input_range` with `range_value_t<Sources>` convertible to `vertex_id_t<G>`.

2 *Preconditions:*

(2.1) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

3 *Hardened preconditions:*

(3.1) — `0 <= source < num_vertices(graph)` for the single-source version.

4 *Effects:*

(4.1) — Does not modify the graph `g`.

(4.2) — 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`, and `on_examine_edge`.

5 *Postconditions:*

(5.1) — All vertices reachable from any source are visited exactly once.

(5.2) — Visitor callbacks are invoked in BFS (level-order) traversal order.

(5.3) — The graph `g` is unchanged.

6 *Throws:*

(6.1) — `std::bad_alloc` if the visited array or queue cannot be allocated.

(6.2) — Any exception thrown by `visitor` callbacks is propagated.

(6.3) — **Exception guarantee:** Basic — `g` remains unchanged; partial traversal may have occurred.

7 *Complexity:*

(7.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — BFS uses a FIFO queue; each vertex and edge is visited at most once.

(7.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — visited array and queue.

8 *Remarks:*

(8.1) — `dijkstra_shortest_paths` provides extended functionality if `breadth_first_search` doesn't have enough capability.

(8.2) — The optional `alloc` parameter supplies an allocator used for the internal FIFO queue. Defaults to `std::allocator<std::byte>`. Provide a custom allocator (e.g. a pool allocator) to control the memory used by the traversal queue.

7.2 Depth-First Search

Traverse the depth-first path from one or more `source` vertices to all reachable vertices in `graph`.

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

7.2.1 Single Source Depth-First Search

```
template <adjacency_list G, class Visitor = empty_visitor,
          class Alloc = allocator<byte>>
void depth_first_search(
     G&& g, // graph
     vertex_id_t<G> source, // starting vertex_id
     Visitor&& visitor = empty_visitor(),
     const Alloc& alloc = Alloc())
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

2 *Preconditions:*

(2.1) — `source` is a valid vertex ID in `g`.

(2.2) — The graph `g` is not modified during traversal.

3 *Hardened preconditions:*

(3.1) — `0 <= source < num_vertices(graph)`.

4 *Effects:*

(4.1) — Does not modify the graph `g`.

(4.2) — Classifies every edge reachable from `source` as tree, back, or forward/cross.

(4.3) — 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_start_vertex`, `on_discover_vertex`, `on_finish_vertex`, `on_examine_edge`, `on_tree_edge`, `on_back_edge`, `on_forward_or_cross_edge`, and `on_finish_edge`.

5 *Postconditions:*

(5.1) — All vertices reachable from `source` are visited exactly once.

(5.2) — `on_finish_vertex` is called in reverse topological order for DAGs.

(5.3) — The graph `g` is unchanged.

6 *Throws:*

(6.1) — `std::bad_alloc` if the color array or stack cannot be allocated.

(6.2) — Any exception thrown by `visitor` callbacks is propagated.

(6.3) — **Exception guarantee:** Basic — `g` remains unchanged; partial traversal may have occurred.

7 *Complexity:*

(7.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — each vertex and edge is visited at most once.

(7.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — color array and DFS stack.

8 *Remarks:*

(8.1) — Uses iterative DFS with explicit stack to avoid recursion-depth limits.

(8.2) — Edge iterators are stored on the stack for $\mathcal{O}(1)$ resume after backtracking.

(8.3) — The optional `alloc` parameter supplies an allocator used for the internal DFS stack. Defaults to `std::allocator<std::byte>`. Provide a custom allocator (e.g. a pool allocator) to control the memory used by the traversal stack.

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.

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

7.3.1 Full-Graph Topological Sort

```
// The initialization helper has been removed; initialization is now
// internal to the algorithm.
```

7.3.2 Single Source Topological Sort

```
// Single-source topological sort
template <adjacency_list G, class OutputIterator, class Alloc = allocator<byte>>
requires output_iterator<OutputIterator, vertex_id_t<G>>
[[nodiscard]] bool
topological_sort(const G& g, const vertex_id_t<G>& source, OutputIterator result,
```

```
                const Alloc& alloc = Alloc());

// Full-graph topological sort
template <adjacency_list G, class OutputIterator, class Alloc = allocator<byte>>
requires output_iterator<OutputIterator, vertex_id_t<G>>
[[nodiscard]] bool topological_sort(const G& g, OutputIterator result,
                                    const Alloc& alloc = Alloc());
```

7.3.3 Multi-Source Topological Sort

```
template <adjacency_list G, input_range Sources, class OutputIterator,
          class Alloc = allocator<byte>>
requires convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         output_iterator<OutputIterator, vertex_id_t<G>>
[[nodiscard]] bool topological_sort(const G& g, const Sources& sources, OutputIterator result,
                                    const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `OutputIterator` satisfies `output_iterator<OutputIterator, vertex_id_t<G>>`.

2 *Preconditions:*

(2.1) — `0 <= source < num_vertices(graph)` for the single-source version.

(2.2) — `0 <= source < num_vertices(graph)`, for each `source` in `sources`, for the multi-source version.

3 *Effects:*

(3.1) — Vertices reachable from the source(s) are written to the output iterator in reverse topological (finish-time) order.

(3.2) — The full-graph overload visits all vertices regardless of reachability from a single source.

4 *Returns:*

(4.1) — `true` if the topological ordering succeeded.

(4.2) — `false` if a cycle was detected in the (sub)graph reachable from the source(s).

(4.3) — Attribute: `[[nodiscard]]`.

5 *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — DFS-based; each vertex and edge is visited at most once.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — color array, finish-order vector, and DFS stack.

6 *Remarks:*

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

(6.2) — The full-graph overload visits all vertices regardless of reachability from a single source.

(6.3) — The optional `alloc` parameter supplies an allocator used for the internal finish-order vector and DFS stack. Defaults to `std::allocator<std::byte>`. All three overloads (single-source, multi-source, full-graph) accept this parameter.

8 Shortest Paths

8.1 Initialization

```
// General concept:  a callable returning a mutable lvalue reference to any per-vertex property
    value
// For exposition only
template <class VF, class G>
concept vertex_property_fn_for =
```

```
      invocable<VF&, const remove_reference_t<G>&, const vertex_id_t<G>&> &&
      is_lvalue_reference_v<invoke_result_t<VF&, const remove_reference_t<G>&, const vertex_id_t<G
          >&>>;

// Type alias:  extracts the value type from a vertex property function's return type
// For exposition only
template <class VF, class G>
using vertex_fn_value_t = remove_cvref_t<
    invoke_result_t<VF&, const remove_reference_t<G>&, const vertex_id_t<G>&>>;

// Concept:  a callable returning a mutable reference to a per-vertex distance value
// For exposition only
template <class DF, class G>
concept distance_fn_for = vertex_property_fn_for<DF, G>;

// Type alias:  extracts the distance value type from a distance function's return type
// For exposition only
template <class DF, class G>
using distance_fn_value_t = vertex_fn_value_t<DF, G>;

// Concept:  a callable returning a mutable reference to a per-vertex predecessor value
// For exposition only
template <class PF, class G>
concept predecessor_fn_for = vertex_property_fn_for<PF, G>;

// Type alias:  extracts the predecessor value type from a predecessor function's return type
// For exposition only
template <class PF, class G>
using predecessor_fn_value_t = vertex_fn_value_t<PF, G>;

// Null predecessor function - used when predecessor tracking is not needed
// For exposition only
struct _null_predecessor_fn {
  template <class G, class VId>
  size_t& operator()(const G&, const VId&);
};
inline _null_predecessor_fn _null_predecessor; // global instance

// Adapts a subscriptable container into a property function:  fn(g, uid) returns container[uid]
    by ref
// For exposition only
template <class Container>
struct container_value_fn {
  Container& c;
  template <class G, class VId>
  constexpr auto& operator()(const G&, const VId& uid) const { return c[uid]; }
};
template <class Container>
container_value_fn(Container&) -> container_value_fn<Container>;

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

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

template <class G, class Distances>
requires adjacency_list<G> && vertex_property_map_for<Distances, G>
constexpr void init_shortest_paths(const G& g, Distances& distances) {
  // exposition only:  sets all entries to infinite distance
```

```
}

template <class G, class Distances, class Predecessors>
requires adjacency_list<G> && vertex_property_map_for<Distances, G> &&
         vertex_property_map_for<Predecessors, G>
constexpr void init_shortest_paths(const G& g, Distances& distances, Predecessors& predecessors)
    {
  // exposition only:  sets distances to infinite distance, predecessors[v] = v
}
```

1 *Effects:*

(1.1) — `init_shortest_paths(g, distances)` sets `distances[v] = infinite_distance()` for each vertex `v` in `g`.

(1.2) — `init_shortest_paths(g, distances, predecessors)` does the same and additionally sets `predecessors[v] = v` for each vertex `v` in `g`.

2 *Returns:*

(2.1) — `infinite_distance<T>()` returns the largest value of type `T`, typically `numeric_limits<T>::max()` for numeric types.

(2.2) — `zero_distance<T>()` returns a zero-valued instance of type `T`, typically `T()` 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	**Cycles?** No	**Throws?** Yes
$\mathcal{O}((\lvert E\rvert + \lvert V\rvert)\log\lvert V\rvert)$	**Multi-edge?** No	

Two heap-selector tag types control the internal priority queue strategy and are passed as the `Heap` template argument:

```
// Heap-selector tag: use std::priority_queue with lazy deletion (default).
// Heap entries may grow to O(E); stale entries are skipped at pop time.
struct use_default_heap {};

// Heap-selector tag: use an indexed d-ary heap with true decrease-key.
// Heap size is bounded by O(V); no stale pops.
// @tparam Arity Children per node (default 4, matching Boost's d_ary_heap_indirect).
template <size_t Arity = 4>
struct use_indexed_dary_heap {
  static constexpr size_t arity = Arity;
};
```

The `Heap` template parameter selects the internal priority queue strategy at compile time. Two heap-selector tags are provided:

— `use_default_heap` (default) — uses `std::priority_queue` with lazy deletion. Stale entries are skipped at pop time; the heap may grow to $\mathcal{O}(|E|)$ in the worst case. Recommended for sparse graphs ($|E|/|V| \lesssim 4$), grid-like topologies, and path graphs.

— `use_indexed_dary_heap<Arity>` — uses an indexed d-ary heap with true decrease-key. Heap size is bounded by $\mathcal{O}(|V|)$; no stale pops. Recommended for dense or hub-heavy graphs ($|E|/|V| \gtrsim 8$). `Arity=8` is the recommended setting on x86_64 for high-$|E|/|V|$ workloads; `Arity=4` matches Boost's `d_ary_heap_indirect`.

8.2.1 Dijkstra Shortest Paths

[PHIL: Feedback: use std::invocable instead of std::function to avoid performance penalty. (Is this really an issue, when the function parameter has a default lambda?)]

8.2.1.1 Single-Source Shortest Paths

```
template <adjacency_list G,
          class DistanceFn,
          class PredecessorFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>,
          class Heap = use_default_heap,
          class Alloc = allocator<byte>>
requires distance_fn_for<DistanceFn, G> &&
         predecessor_fn_for<PredecessorFn, G> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
constexpr void dijkstra_shortest_paths(
      G&& g,
      const vertex_id_t<G>& source,
      DistanceFn&& distance,
      PredecessorFn&& predecessor,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>(),
      Heap heap_tag = Heap{},
      const Alloc& alloc = Alloc());
```

8.2.1.2 Multi-Source Shortest Paths

```
template <adjacency_list G,
          input_range Sources,
          class DistanceFn,
          class PredecessorFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>,
          class Heap = use_default_heap,
          class Alloc = allocator<byte>>
requires distance_fn_for<DistanceFn, G> &&
         predecessor_fn_for<PredecessorFn, G> &&
         convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
constexpr void dijkstra_shortest_paths(
      G&& g,
      const Sources& sources,
      DistanceFn&& distance,
      PredecessorFn&& predecessor,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>(),
      Heap heap_tag = Heap{},
      const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `DistanceFn` satisfies `distance_fn_for<DistanceFn, G>` with `is_arithmetic_v<distance_fn_value_t<DistanceFn, G>>`.

(1.3) — `PredecessorFn` satisfies `predecessor_fn_for<PredecessorFn, G>`.

(1.4) — `WF` satisfies `basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>`.

2 *Constant When:*

(2.1) — All operations on the distance, predecessor, and weight types are constant subexpressions, and no visitor callback throws.

3 *Preconditions:*

(3.1) — Each `source` in `sources` is a valid vertex ID in `g`, for the multi-source version.

(3.2) — `distance(g, vid)= infinite_distance()` for every vertex `vid` in `g`.

(3.3) — `predecessor(g, vid)= vid` for every vertex `vid` in `g`.

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

4 *Hardened preconditions:*

(4.1) — `source` is a valid vertex ID in `g` for the single-source version.

5 *Effects:*

(5.1) — If vertex with index `i` is reachable from vertex `source`, then `distance(g, i)` will contain the distance from `source` to vertex `i`. Otherwise `distance(g, i)` will contain `infinite_distance()`.

(5.2) — If vertex with index `i` is reachable from vertex `source`, then `predecessor(g, i)` will contain the predecessor vertex of vertex `i`. Otherwise `predecessor(g, i)` will contain `i`.

(5.3) — Does not modify the graph `g`.

(5.4) — 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`.

6 *Postconditions:*

(6.1) — `distance(g, s)== 0` for all sources `s`.

(6.2) — For every reachable vertex `v`, `distance(g, v)` contains the shortest distance from the nearest source.

(6.3) — For every reachable vertex `v`, `predecessor(g, v)` contains the predecessor on a shortest path tree.

(6.4) — For every unreachable vertex `v`, `distance(g, v)== infinite_distance<distance_fn_value_t<DistanceFn,G>>()`.

7 *Throws:*

(7.1) — `std::out_of_range` if any source vertex ID is not in `vertices(g)`.

(7.2) — `std::out_of_range` if a negative edge weight is encountered (for signed weight types).

(7.3) — Any exception thrown by `visitor` callbacks is propagated.

(7.4) — **Exception guarantee:** Basic — `g` remains unchanged; `distances` and `predecessor` may be partially modified.

8 *Complexity:*

(8.1) — **Time:** $\mathcal{O}((|E|+|V|)\log|V|)$ with `use_default_heap` (binary heap via `std::priority_queue`). With `use_indexed_dary_heap<d>` the heap size is bounded by $\mathcal{O}(|V|)$ and each operation costs $\mathcal{O}(\log_d|V|)$.

(8.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary for `use_indexed_dary_heap`; up to $\mathcal{O}(|E|)$ for `use_default_heap` due to lazy deletion.

9 *Remarks:*

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

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

(9.3) — `DistanceFn` and `PredecessorFn` must satisfy `distance_fn_for` and `predecessor_fn_for` respectively. For index graphs, wrap a `std::vector` using `container_value_fn`; for mapped graphs, wrap a `std::unordered_map` the same way (see § *Vertex Property Function Concepts*).

(9.4) — Pass `_null_predecessor` as `predecessor` when predecessor tracking is not needed (avoids storing results).

(9.5) — The `Heap` template parameter selects the internal heap implementation. Pass `use_default_heap{}` (default) for sparse or grid-like graphs; pass `use_indexed_dary_heap<Arity>{}` for dense or hub-heavy graphs where many edges trigger distance relaxation. `use_indexed_dary_heap<8>` is recommended on x86_64 for high-$|E|/|V|$ workloads.

(9.6) — The optional `alloc` parameter supplies an allocator used for the internal priority queue. Defaults to `std::allocator<std::byte>`. Provide a custom allocator (e.g. a pool allocator) to control the memory used by the priority queue during traversal.

8.2.1.3 Example: Dijkstra on a Mapped Graph

The following example demonstrates `dijkstra_shortest_paths` on a graph whose vertex IDs are `std::string` values. Lambda closures over `std::unordered_map` instances serve as the `DistanceFn` and `PredecessorFn` arguments. All other algorithm code is identical to the index-graph case.

```
// Example:  Dijkstra on a mapped graph (string vertex IDs, double edge weights)

#include <graph/container/dynamic_graph.hpp>
#include <graph/container/traits/uov_graph_traits.hpp>
#include <graph/adj_list/vertex_property_map.hpp>
#include <graph/algorithm/dijkstra_shortest_paths.hpp>
using namespace graph;
using namespace graph::adj_list;

using Traits = container::uov_graph_traits<double, void, void, std::string>;
using G = container::dynamic_graph<double, void, void, std::string, false, Traits>;

G g({{"a", "b", 4.0}, {"a", "c", 2.0},
     {"b", "d", 5.0},
     {"c", "b", 1.0}, {"c", "d", 8.0}});

constexpr double inf = std::numeric_limits<double>::max();

std::unordered_map<std::string, double> dist_map;
std::unordered_map<std::string, std::string> pred_map;
for (auto&& [uid, u] : views::vertexlist(g)) {
    dist_map[uid] = inf;
    pred_map[uid] = uid;
}

dijkstra_shortest_paths(g, std::string{"a"},
    [&dist_map](const auto&, const auto& uid) -> double& { return dist_map[uid]; },
    [&pred_map](const auto&, const auto& uid) -> std::string& { return pred_map[uid]; },
    [](const auto& g, const edge_t<G>& e) { return edge_value(g, e); });

// Shortest a to d:  a to c (2) to b (1) to d (5) = 8
// dist_map["d"] == 8
```

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 <adjacency_list G,
          class DistanceFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>,
          class Heap = use_default_heap,
          class Alloc = allocator<byte>>
requires distance_fn_for<DistanceFn, G> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
constexpr void dijkstra_shortest_distances(
      G&& g,
      const vertex_id_t<G>& source,
      DistanceFn&& distance,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>(),
      Heap heap_tag = Heap{},
      const Alloc& alloc = Alloc());
```

8.2.2.2 Multi-Source Shortest Distances

```
template <adjacency_list G,
          input_range Sources,
          class DistanceFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>,
          class Heap = use_default_heap,
          class Alloc = allocator<byte>>
requires distance_fn_for<DistanceFn, G> &&
         convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
constexpr void dijkstra_shortest_distances(
      G&& g,
      const Sources& sources,
      DistanceFn&& distance,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>(),
      Heap heap_tag = Heap{},
      const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `DistanceFn` satisfies `distance_fn_for<DistanceFn, G>` with `is_arithmetic_v<distance_fn_value_t<DistanceFn, G>>`.

(1.3) — `WF` satisfies `basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>`.

2 *Constant When:*

(2.1) — All operations on the distance and weight types are constant subexpressions, and no visitor callback throws.

3 *Preconditions:*

(3.1) — Each `source` in `sources` is a valid vertex ID in `g`, for the multi-source version.

(3.2) — `distance(g, vid)= infinite_distance()` for every vertex `vid` in `g`.

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

4 *Hardened preconditions:*

(4.1) — `source` is a valid vertex ID in `g` for the single-source version.

5 *Effects:*

(5.1) — If vertex with index `i` is reachable from vertex `source`, then `distance(g, i)` will contain the distance from `source` to vertex `i`. Otherwise `distance(g, i)` will contain `infinite_distance()`.

(5.2) — Does not modify the graph `g`.

(5.3) — 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`.

6 *Postconditions:*

(6.1) — `distance(g, s)== 0` for all sources `s`.

(6.2) — For every reachable vertex `v`, `distance(g, v)` contains the shortest distance from the nearest source.

(6.3) — For every unreachable vertex `v`, `distance(g, v)== infinite_distance<distance_fn_value_t<DistanceFn,G>>()`.

7 *Throws:*

(7.1) — `std::out_of_range` if any source vertex ID is not in `vertices(g)`.

(7.2) — `std::out_of_range` if a negative edge weight is encountered (for signed weight types).

(7.3) — Any exception thrown by `visitor` callbacks is propagated.

(7.4) — **Exception guarantee:** Basic — `g` remains unchanged; `distances` may be partially modified.

8 *Complexity:*

(8.1) — **Time:** $\mathcal{O}((|E|+|V|)\log|V|)$ with `use_default_heap` (binary heap via `std::priority_queue`). With `use_indexed_dary_heap<d>` the heap size is bounded by $\mathcal{O}(|V|)$ and each operation costs $\mathcal{O}(\log_d|V|)$.

(8.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary for `use_indexed_dary_heap`; up to $\mathcal{O}(|E|)$ for `use_default_heap` due to lazy deletion.

9 *Remarks:*

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

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

(9.3) — `DistanceFn` must satisfy `distance_fn_for<DistanceFn,G>`. Wrap a `std::vector` or `std::unordered_map` with `container_value_fn` (see §*Vertex Property Function Concepts*).

(9.4) — The `Heap` template parameter selects the internal heap implementation. Pass `use_default_heap{}` (default) for sparse or grid-like graphs; pass `use_indexed_dary_heap<Arity>{}` for dense or hub-heavy graphs where many edges trigger distance relaxation. `use_indexed_dary_heap<8>` is recommended on x86_64 for high-$|E|/|V|$ workloads.

(9.5) — The optional `alloc` parameter supplies an allocator used for the internal priority queue. Defaults to `std::allocator<std::byte>`. Provide a custom allocator (e.g. a pool allocator) to control the memory used by the priority queue during traversal.

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 $\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$	**Cycles?** No **Multi-edge?** No	**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, Bellman–Ford algorithm**]

`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 <adjacency_list G,
          class DistanceFn,
          class PredecessorFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>>
requires distance_fn_for<DistanceFn, G> &&
         predecessor_fn_for<PredecessorFn, G> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
[[nodiscard]] constexpr optional<vertex_id_t<G>> bellman_ford_shortest_paths(
      G&& g,
      const vertex_id_t<G>& source,
      DistanceFn&& distance,
      PredecessorFn&& predecessor,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>());
```

8.3.1.2 Multi-Source Shortest Paths

```
template <adjacency_list G,
          input_range Sources,
          class DistanceFn,
          class PredecessorFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>>
requires distance_fn_for<DistanceFn, G> &&
         predecessor_fn_for<PredecessorFn, G> &&
```

```
        convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
        basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
[[nodiscard]] constexpr optional<vertex_id_t<G>> bellman_ford_shortest_paths(
      G&& g,
      const Sources& sources,
      DistanceFn&& distance,
      PredecessorFn&& predecessor,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `DistanceFn` satisfies `distance_fn_for<DistanceFn, G>` with `is_arithmetic_v<distance_fn_value_t<DistanceFn, G>>`.

(1.3) — `PredecessorFn` satisfies `predecessor_fn_for<PredecessorFn, G>`.

(1.4) — `WF` satisfies `basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>`.

2 *Constant When:*

(2.1) — All operations on the distance, predecessor, and weight types are constant subexpressions, and no visitor callback throws.

3 *Preconditions:*

(3.1) — Each `source` in `sources` is a valid vertex ID in `g`, for the multi-source version.

(3.2) — `distance(g, vid)= infinite_distance()` for every vertex `vid` in `g`.

(3.3) — `predecessor(g, vid)= vid` for every vertex `vid` in `g`.

4 *Hardened preconditions:*

(4.1) — `source` is a valid vertex ID in `g` for the single-source version.

5 *Effects:*

(5.1) — If vertex with index `i` is reachable from vertex `source`, then `distance(g, i)` will contain the distance from `source` to vertex `i`. Otherwise `distance(g, i)` will contain `infinite_distance()`.

(5.2) — If vertex with index `i` is reachable from vertex `source`, then `predecessor(g, i)` will contain the predecessor vertex of vertex `i`. Otherwise `predecessor(g, i)` will contain `i`.

(5.3) — Does not modify the graph `g`.

(5.4) — 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`.

6 *Postconditions:*

(6.1) — `distance(g, s)== 0` for all sources `s`.

(6.2) — If no negative cycle is detected: for every reachable vertex `v`, `distance(g, v)` contains the shortest distance from the nearest source.

(6.3) — If a negative cycle is detected, `distance` and `predecessor` may contain intermediate values.

(6.4) — For every unreachable vertex `v`, `distance(g, v)== infinite_distance<distance_fn_value_t<DistanceFn,G>>()`.

7 *Returns:*

(7.1) — `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.

(7.2) — Attribute: `[[nodiscard]]`.

8 *Throws:*

(8.1) — `std::out_of_range` if any source vertex ID is not in `vertices(g)`.

(8.2) — **Exception guarantee:** Basic — `g` remains unchanged; `distances` and `predecessors` may be partially modified.

9 *Complexity:*

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

(9.2) — **Space:** $\mathcal{O}(1)$ auxiliary beyond caller-provided arrays.

10 *Remarks:*

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

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

(10.3) — `DistanceFn` and `PredecessorFn` must satisfy `distance_fn_for` and `predecessor_fn_for` respectively. Wrap containers with `container_value_fn` (see §*Vertex Property Function Concepts*).

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 <adjacency_list G,
          class DistanceFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>>
requires distance_fn_for<DistanceFn, G> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
[[nodiscard]] constexpr optional<vertex_id_t<G>> bellman_ford_shortest_distances(
      G&& g,
      const vertex_id_t<G>& source,
      DistanceFn&& distance,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>());
```

8.3.2.2 Multi-Source Shortest Distances

```
template <adjacency_list G,
          input_range Sources,
          class DistanceFn,
          class WF = function<distance_fn_value_t<DistanceFn, G>(const remove_reference_t<G>&,
                                                                 const edge_t<G>&)>,
          class Visitor = empty_visitor,
          class Compare = less<distance_fn_value_t<DistanceFn, G>>,
          class Combine = plus<distance_fn_value_t<DistanceFn, G>>>
```

```
requires distance_fn_for<DistanceFn, G> &&
         convertible_to<range_value_t<Sources>, vertex_id_t<G>> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>
[[nodiscard]] constexpr optional<vertex_id_t<G>> bellman_ford_shortest_distances(
      G&& g,
      const Sources& sources,
      DistanceFn&& distance,
      WF&& weight = [](const auto&,
                       const edge_t<G>& uv) { return distance_fn_value_t<DistanceFn, G>(1); },
      Visitor&& visitor = empty_visitor(),
      Compare&& compare = less<distance_fn_value_t<DistanceFn, G>>(),
      Combine&& combine = plus<distance_fn_value_t<DistanceFn, G>>());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `DistanceFn` satisfies `distance_fn_for<DistanceFn, G>` with `is_arithmetic_v<distance_fn_value_t<DistanceFn, G>>`.

(1.3) — `WF` satisfies `basic_edge_weight_function<G, WF, distance_fn_value_t<DistanceFn, G>, Compare, Combine>`.

2 *Constant When:*

(2.1) — All operations on the distance and weight types are constant subexpressions, and no visitor callback throws.

3 *Preconditions:*

(3.1) — Each `source` in `sources` is a valid vertex ID in `g`, for the multi-source version.

(3.2) — `distance(g, vid)== infinite_distance()` for every vertex `vid` in `g`.

4 *Hardened preconditions:*

(4.1) — `source` is a valid vertex ID in `g` for the single-source version.

5 *Effects:*

(5.1) — If vertex with index `i` is reachable from vertex `source`, then `distance(g, i)` will contain the distance from `source` to vertex `i`. Otherwise `distance(g, i)` will contain `infinite_distance()`.

(5.2) — Does not modify the graph `g`.

(5.3) — 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`.

6 *Postconditions:*

(6.1) — `distance(g, s)== 0` for all sources `s`.

(6.2) — If no negative cycle is detected: for every reachable vertex `v`, `distance(g, v)` contains the shortest distance from the nearest source.

(6.3) — For every unreachable vertex `v`, `distance(g, v)== infinite_distance<distance_fn_value_t<DistanceFn,G>>()`.

7 *Returns:*

(7.1) — `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 important to get the vertex ids of the cycle using `find_negative_cycle`.

(7.2) — Attribute: `[[nodiscard]]`.

8 *Throws:*

(8.1) — `std::out_of_range` if any source vertex ID is not in `vertices(g)`.

(8.2) — **Exception guarantee:** Basic — `g` remains unchanged; `distances` may be partially modified.

9 *Complexity:*

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

(9.2) — **Space:** $\mathcal{O}(1)$ auxiliary beyond caller-provided arrays.

10 *Remarks:*

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

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

(10.3) — `DistanceFn` must satisfy `distance_fn_for<DistanceFn,G>`. Wrap a container with `container_value_fn` (see §*Vertex Property Function Concepts*).

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 <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 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `Predecessors` satisfies `forward_range`.

(1.3) — `OutputIterator` satisfies `output_iterator<OutputIterator, vertex_id_t<G>>`.

2 *Preconditions:*

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

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

3 *Effects:*

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

4 *Complexity:*

(4.1) — **Time:** $\mathcal{O}(|E| + |V|)$.

(4.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — cycle-tracking array.

9 Clustering

9.1 Triangle Counting

Compute the number of triangles in a graph. A triangle is a set of three vertices $\{u, v, w\}$ where each pair is connected by an edge. Two functions are provided: `triangle_count` counts triangles (3-cliques) in undirected graphs, and `directed_triangle_count` counts directed 3-cycles in directed graphs.

Both algorithms use a merge-based set intersection approach on sorted adjacency lists, which is more efficient than nested-loop or hash-based methods for sparse graphs. The algorithms require the graph to satisfy both `adjacency_list<G>` and `ordered_vertex_edges<G>`.

9.1.1 Triangle Count (Undirected)

Count the number of triangles (3-cliques) in an undirected graph with sorted adjacency lists.

Complexity $\mathcal{O}(m^{3/2})$	**Cycles?** Yes **Multi-edge?** No	**Throws?** No

```
template <adjacency_list G>
requires ordered_vertex_edges<G>
[[nodiscard]] size_t triangle_count(G&& g) noexcept;
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `G` satisfies `ordered_vertex_edges<G>`: the outgoing edges of each vertex must be sorted in ascending order of `target_id`.

2 *Preconditions:*

(2.1) — The graph represents an undirected graph, with edges stored bidirectionally (both (u, v) and (v, u)).

(2.2) — Adjacency lists are sorted by `target_id` in ascending order.

3 *Effects:*

(3.1) — For each edge (u, v) where `vertex_id(g, u)` $<$ `vertex_id(g, v)`:

1. Get the sorted adjacency lists for u and v.
2. Perform a merge-based intersection of the two lists (analogous to `std::set_intersection`).
3. For each common neighbor w where `vertex_id(g, w)` $>$ `vertex_id(g, v)`, increment the triangle count.

(3.2) — The ordering constraint `vertex_id(g, u)` $<$ `vertex_id(g, v)` $<$ `vertex_id(g, w)` guarantees each triangle is counted exactly once.

4 *Returns:*

(4.1) — `[[nodiscard]]`. The total number of triangles in the graph.

(4.2) — Returns 0 for empty graphs or graphs with fewer than 3 vertices.

5 *Throws:* This function is `noexcept`. *Complexity:*

(6.1) — **Time:** $\mathcal{O}(m^{3/2})$ average, where $m = |E|$. Best case $\mathcal{O}(|V| + |E|)$ for triangle-free graphs. Worst case $\mathcal{O}(|V| \cdot d_{\max}^2)$ where $d_{\max}$ is the maximum vertex degree.

(6.2) — **Space:** $\mathcal{O}(1)$ auxiliary (excluding graph storage).

7 *Remarks:*

(7.1) — Edge weights, if present, are ignored.

(7.2) — Self-loops are ignored (a vertex cannot form a triangle with itself).

(7.3) — Multi-edges: each parallel edge contributes to the triangle count separately.

(7.4) — Works correctly on both connected and disconnected graphs.

(7.5) — The `ordered_vertex_edges` requirement enables $\mathcal{O}(d)$ set intersection per edge rather than $\mathcal{O}(d^2)$ nested loops, which is critical for performance on high-degree vertices. Graph types using `std::set` or `std::map` edge containers (e.g. `vos`, `uos`, `dos`, `mos`) naturally satisfy this requirement.

Complexity $\mathcal{O}(m^{3/2})$	**Cycles?** Yes **Multi-edge?** No	**Throws?** No

9.1.2 Directed Triangle Count

Count the number of directed 3-cycles in a directed graph with sorted adjacency lists. A directed 3-cycle is a set of three vertices $\{u, v, w\}$ where directed edges $u \to v$, $v \to w$, and $u \to w$ all exist.

```
template <adjacency_list G>
requires ordered_vertex_edges<G>
[[nodiscard]] size_t directed_triangle_count(G&& g) noexcept;
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `G` satisfies `ordered_vertex_edges<G>`: the outgoing edges of each vertex must be sorted in ascending order of `target_id`.

2 *Preconditions:*

(2.1) — Adjacency lists (out-neighbor lists) are sorted by `target_id` in ascending order.

3 *Effects:*

(3.1) — For each vertex u, for each out-neighbor v ($v \neq u$):

1. Get the sorted out-neighbor lists for u and v.
2. Perform a merge-based intersection of the two lists.
3. For each common out-neighbor w ($w \neq u$ and $w \neq v$), increment the count.

(3.2) — Each directed 3-cycle ($u \to v, v \to w, u \to w$) is counted exactly once.

(3.3) — Self-loops are skipped.

4 *Returns:*

(4.1) — `[[nodiscard]]`. The total number of directed 3-cycles in the graph.

5 *Throws:* This function is `noexcept`. *Complexity:*

(6.1) — **Time:** $\mathcal{O}(m^{3/2})$ average, where $m = |E|$. Worst case $\mathcal{O}(|V| \cdot d_{\max}^2)$ where $d_{\max}$ is the maximum out-degree.

(6.2) — **Space:** $\mathcal{O}(1)$ auxiliary (excluding graph storage).

7 *Remarks:*

(7.1) — For an undirected graph stored with bidirectional edges, this function counts each undirected triangle 6 times (once per permutation of the 3 vertices). Use `triangle_count` instead for undirected graphs.

(7.2) — Unlike `triangle_count`, this function does not impose vertex ID ordering constraints on the counted cycles.

(7.3) — Edge weights, if present, are ignored.

10 Communities

10.1 Label Propagation

Propagate vertex labels for community detection by majority voting among neighbours. Each iteration shuffles the vertex processing order, then sets every vertex's label to the most popular label among its neighbours. Ties are broken randomly using the supplied random-number generator. The algorithm iterates until no label changes (convergence) or until `max_iters` iterations have been performed. $\mathcal{O}(M)$ complexity, where $M = |E|$,

is based on one iteration (each edge is examined once); the number of iterations required for convergence is typically small relative to graph size.

A second overload accepts an `empty_label` sentinel for semi-supervised propagation: vertices whose label equals `empty_label` are treated as unlabelled — they do not vote and are not counted in neighbour tallies, but can acquire a label from a labelled neighbour. Vertices in components with no labelled vertex retain `empty_label` after convergence.

Complexity $\mathcal{O}(M)$	**Cycles?** Yes **Multi-edge?** Yes	**Throws?** Yes

10.1.0.1 Basic Label Propagation

```
template <adjacency_list G,
          class LabelFn,
          class Gen = default_random_engine,
          class T = size_t>
requires vertex_property_fn_for<LabelFn, G> &&
         equality_comparable<vertex_fn_value_t<LabelFn, G>> &&
         uniform_random_bit_generator<remove_cvref_t<Gen>>
void label_propagation(G&& g,
                       LabelFn&& label,
                       Gen&& rng = default_random_engine{},
                       T max_iters = numeric_limits<T>::max());
```

10.1.0.2 Label Propagation with Empty-Label Sentinel

```
template <adjacency_list G,
          class LabelFn,
          class Gen = default_random_engine,
          class T = size_t>
requires vertex_property_fn_for<LabelFn, G> &&
         equality_comparable<vertex_fn_value_t<LabelFn, G>> &&
         uniform_random_bit_generator<remove_cvref_t<Gen>>
void label_propagation(G&& g,
                       LabelFn&& label,
                       vertex_fn_value_t<LabelFn, G> empty_label,
                       Gen&& rng = default_random_engine{},
                       T max_iters = numeric_limits<T>::max());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `LabelFn` satisfies `vertex_property_fn_for<LabelFn, G>`.

(1.3) — `vertex_fn_value_t<LabelFn, G>` satisfies `std::equality_comparable`.

(1.4) — `Gen` satisfies `std::uniform_random_bit_generator`.

2 *Preconditions:*

(2.1) — `label(g, v)` returns a meaningful initial label for every vertex `v` in `g`.

(2.2) — For the sentinel overload, vertices to be treated as unlabelled must have their label initialised to `empty_label`.

3 *Effects:*

(3.1) — Each iteration: shuffles the vertex processing order using `rng`, then for each vertex u in that order:

1. Tallies the labels of all neighbours of u (skipping neighbours with `empty_label` in the sentinel overload).
2. Assigns `label(g, u)` to the most frequent neighbour label, breaking ties randomly via `rng`.
3. Isolated vertices (no neighbours, or all neighbours unlabelled) retain their current label.

(3.2) — Iteration stops when no label changes in a full pass (convergence) or `max_iters` is reached.

(3.3) — The graph `g` is not modified.

4 *Throws:*

(4.1) — `std::bad_alloc` from internal frequency map and candidate vector allocations.

(4.2) — **Exception guarantee:** Basic — `g` remains unchanged; `label` may be partially modified.

5 *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|E|)$ per iteration; convergence typically requires few iterations relative to graph size.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — shuffled vertex-ID vector and per-vertex frequency map.

6 *Remarks:*

(6.1) — User is responsible for providing meaningful initial vertex labels.

(6.2) — Some label propagation implementations use vertex IDs as initial labels; this is not done here because the label type can be more general than the vertex ID type.

(6.3) — Directed edges are followed as-is; for undirected semantics the graph should store both (u, v) and (v, u).

(6.4) — Multi-edges and self-loops are counted in the neighbour tally and may bias the label assignment.

(6.5) — When `empty_label` is not present in `label`, the sentinel overload behaves identically to the basic overload.

11 Components

11.1 Articulation Points

Find articulation points (cut vertices) of a graph. An articulation point is a vertex whose removal, along with its incident edges, disconnects the graph into two or more connected components. The algorithm uses an iterative Hopcroft-Tarjan DFS, tracking discovery times (`disc[v]`) and low-link values (`low[v]`), defined as the minimum discovery time reachable from the subtree rooted at v via back-edges. A vertex u is an articulation point under one of two rules:

— **Root rule:** u is the root of a DFS tree and has two or more DFS children.

— **Non-root rule:** u is not a root and has a child v with `low[v]` $\geq$ `disc[u]`.

Complexity $\mathcal{O}(\|E\| + \|V\|)$	**Cycles?** Yes **Multi-edge?** Yes	**Throws?** Yes

```
template <adjacency_list G, class Iter, class Alloc = allocator<byte>>
requires output_iterator<Iter, vertex_id_t<G>>
void articulation_points(G&& g, Iter cut_vertices, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `Iter` satisfies `std::output_iterator<Iter, vertex_id_t<G>>`.

2 *Preconditions:*

(2.1) — For undirected graph semantics, each edge $\{u, v\}$ must be stored as both (u, v) and (v, u) in `g`.

3 *Effects:*

(3.1) — Performs an iterative DFS over all vertices (including disconnected components).

(3.2) — Writes each articulation point vertex ID to `cut_vertices` exactly once.

(3.3) — `g` is not modified.

(3.4) — Self-loops are ignored and do not affect articulation point detection.

(3.5) — Multi-edges: only the first reverse edge to the DFS parent is treated as the tree edge; additional parallel edges are treated as back-edges that update low-link values.

4 *Throws:*

(4.1) — `std::bad_alloc` if internal vector allocation fails.

(4.2) — **Exception guarantee:** Basic — `g` remains unchanged; the output iterator may be partially written.

5 *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — each vertex and edge is visited exactly once.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — discovery time, low-link, parent, child-count, and emitted arrays, plus DFS stack.

6 *Remarks:*

(6.1) — There is no ordering guarantee on the articulation points written to `cut_vertices`.

(6.2) — Disconnected graphs are handled correctly; the outer loop restarts DFS from each unvisited vertex.

(6.3) — The optional `alloc` parameter supplies an allocator used for the internal DFS stack and discovery/low-link arrays. Defaults to `std::allocator<std::byte>`.

11.2 BiConnected Components

Find the biconnected components of a graph. A biconnected component (also called a 2-connected component) is a maximal biconnected subgraph — one that is connected and has no articulation points. Equivalently, any two vertices in a biconnected component lie on a common simple cycle.

The algorithm extends the Hopcroft-Tarjan DFS with an explicit edge stack. During the DFS, each tree edge and back edge is pushed onto the edge stack. When a biconnected-component boundary is detected on backtrack (i.e., `low[v]` $\geq$ `disc[u]` for child v and parent u), the edge stack is popped down to and including edge (u, v) and the unique vertex IDs from those edges form one biconnected component. Isolated vertices are emitted as trivial single-vertex components. Articulation-point vertices appear in more than one component.

Complexity $\mathcal{O}(\|E\| + \|V\|)$	**Cycles?** Yes **Multi-edge?** Yes	**Throws?** Yes

```
template <adjacency_list G, class OuterContainer, class Alloc = allocator<byte>>
void biconnected_components(G&& g, OuterContainer& components, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `OuterContainer` supports `push_back` with an inner container type constructible from a pair of iterators over `vertex_id_t<G>`.

2 *Preconditions:*

(2.1) — For undirected semantics, each edge $\{u, v\}$ must be stored as both (u, v) and (v, u) in `g`.

3 *Effects:*

(3.1) — Performs an iterative Hopcroft-Tarjan DFS over all vertices.

(3.2) — For each biconnected component detected, one inner container of vertex IDs is `push_back`'d into `components`.

(3.3) — Isolated vertices (degree 0) are emitted as trivial single-vertex components.

(3.4) — Every vertex appears in at least one component; articulation-point vertices appear in more than one component.

(3.5) — Self-loops are ignored and do not affect component detection.

(3.6) — Multi-edges: only the first reverse edge to the DFS parent is treated as the tree edge; additional parallel edges are treated as back-edges.

(3.7) — `g` is not modified.

4 *Throws:*

(4.1) — `std::bad_alloc` from internal vector, set, or stack allocations.

(4.2) — **Exception guarantee:** Basic — `g` remains unchanged; `components` may be partially written.

5 *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — each vertex and edge is visited exactly once.

(5.2) — **Space:** $\mathcal{O}(|V| + |E|)$ auxiliary — discovery/low-link arrays and parent map ($\mathcal{O}(|V|)$), DFS frame stack ($\mathcal{O}(|V|)$), and edge stack ($\mathcal{O}(|E|)$).

6 *Remarks:*

(6.1) — There is no ordering guarantee on the components or on vertex IDs within a component.

(6.2) — Disconnected graphs are handled correctly; the outer loop restarts DFS from each unvisited vertex.

(6.3) — The implementation uses iterative DFS with stored edge iterators to avoid recursion-depth limits and $\mathcal{O}$(degree) re-scans on resume.

(6.4) — The optional `alloc` parameter supplies an allocator used for the internal DFS stack, edge stack, and discovery/low-link arrays. Defaults to `std::allocator<std::byte>`.

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

```
template <adjacency_list G, class ComponentFn, class Alloc = allocator<byte>>
requires vertex_property_fn_for<ComponentFn, G>
size_t connected_components(G&& g, ComponentFn&& component, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `ComponentFn` satisfies `vertex_property_fn_for<ComponentFn, G>`.

2 *Preconditions:*

(2.1) — `component(g, v)` is callable for each vertex `v` in `g`.

3 *Effects:*

(3.1) — `component(g, v)` is set to the connected component id of vertex `v`.

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

4 *Returns:* The number of connected components[5] found. *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|E| + |V|)$.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — component array and DFS stack.

6 *Remarks:*

(6.1) — If the maximum component id is 0, then all vertices are in a single connected component.

(6.2) — If the maximum component id is `num_vertices(g)-1`, each vertex is the sole vertex in its own connected component.

(6.3) — The optional `alloc` parameter supplies an allocator used for the internal DFS stack. Defaults to `std::allocator<std::byte>`.

11.4 Strongly Connected Components

11.4.1 Kosaraju

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

```
template <adjacency_list G, adjacency_list GT, class ComponentFn,
          class Alloc = allocator<byte>>
requires vertex_property_fn_for<ComponentFn, G>
void kosaraju(G&& g, GT&& g_t, ComponentFn&& component, const Alloc& alloc = Alloc());

template <bidirectional_adjacency_list G, class ComponentFn,
          class Alloc = allocator<byte>>
requires vertex_property_fn_for<ComponentFn, G>
void kosaraju(G&& g, ComponentFn&& component, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `ComponentFn` satisfies `vertex_property_fn_for<ComponentFn, G>`.

2 *Preconditions:*

(2.1) — For the two-graph overload, `g_t` is the transpose of `g`, where edge `uv` in `g` implies edge `vu` in `g_t`.

3 *Hardened preconditions:*

(3.1) — `num_vertices(g)== num_vertices(g_t)` for the two-graph overload.

4 *Effects:*

(4.1) — `component(g, v)` is set to the strongly connected component id of vertex `v`.

(4.2) — The component id is in the range `0 <= component(g, v)< num_vertices(g)`.

5 *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|E| + |V|)$.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — visited array and finish order.

6 *Remarks:*

(6.1) — If the maximum component id is 0, then all vertices are in a single strongly connected component.

(6.2) — If the maximum component id is `num_vertices(g)-1`, each vertex is the sole vertex in its own strongly connected component.

(6.3) — The optional `alloc` parameter supplies an allocator used for the internal finish-order vector and DFS stacks. Defaults to `std::allocator<std::byte>`.

11.4.2 Tarjan's SCC

Find strongly connected components of a directed graph using Tarjan's algorithm. A strongly connected component (SCC) is a maximal set of vertices such that there is a directed path from every vertex in the set to every other vertex in the set.

Complexity	**Cycles?** Yes	**Throws?** Yes
$\mathcal{O}(\|E\| + \|V\|)$	**Multi-edge?** Yes	

```
/*
 * Tarjan's Strongly Connected Components
 */
template <adjacency_list G, class ComponentFn, class Alloc = allocator<byte>>
requires vertex_property_fn_for<ComponentFn, G>
size_t tarjan_scc(G&& g, ComponentFn&& component, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `ComponentFn` satisfies `vertex_property_fn_for<ComponentFn, G>`.

2 *Preconditions:*

(2.1) — `component` must be callable for each vertex of `g`.

3 *Hardened preconditions:*

(3.1) — `g` must be a directed graph.

4 *Effects:*

(4.1) — `component(g, uid)` is set to the SCC id of vertex `uid` for every vertex in `g`.

(4.2) — Component ids are assigned in the range `0 <= component(g, uid)< num_vertices(g)`, numbered in reverse topological order of the condensed SCC DAG.

(4.3) — Does not modify the graph `g`.

5 *Returns:* The number of strongly connected components found. *Throws:*

(6.1) — `std::bad_alloc` if internal allocations (discovery time, low-link, on-stack flag, DFS stack, or SCC stack) fail.

(6.2) — **Exception guarantee:** Basic — `g` is unchanged; partial component assignments may have occurred.

7 *Complexity:*

(7.1) — **Time:** $\mathcal{O}(|V| + |E|)$ — single DFS pass, each vertex and edge visited at most once.

(7.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — discovery time, low-link, on-stack flag arrays, DFS stack, and SCC stack.

8 *Remarks:*

(8.1) — Compared to `kosaraju`, Tarjan's algorithm requires only a single DFS pass and does not require (or accept) a transpose graph. It therefore works on any `adjacency_list` graph, whereas the single-graph overload of `kosaraju` requires `bidirectional_adjacency_list`.

(8.2) — Uses an iterative DFS with an explicit stack to avoid recursion-depth limits.

(8.3) — A vertex `u` is the root of an SCC when `disc[u] == low[u]` after all of its outgoing edges have been processed.

(8.4) — Low-link values track the earliest reachable discovery time in the current DFS subtree. Back and cross edges to vertices already in a *completed* SCC do not update low-link values.

(8.5) — If the return value is `1`, all vertices belong to a single strongly connected component (the graph is strongly connected).

(8.6) — If the return value equals `num_vertices(g)`, every vertex is its own SCC (a DAG).

(8.7) — The optional `alloc` parameter supplies an allocator used for the internal DFS stack, SCC stack, and discovery/low-link arrays. Defaults to `std::allocator<std::byte>`.

12 Maximal Independent Set

12.1 Maximal Independent Set

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

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

```
template <adjacency_list G, class Iter>
requires output_iterator<Iter, vertex_id_t<G>>
size_t maximal_independent_set(G&& g, Iter mis, const vertex_id_t<G>& seed = 0);
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `Iter` satisfies `output_iterator<Iter, vertex_id_t<G>>`.

2 *Hardened preconditions:*

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

3 *Effects:*

(3.1) — Output iterator `mis` contains the maximal independent set of vertices that includes `seed`, which is a subset of `vertices(graph)`.

4 *Returns:* The number of vertices in the maximal independent set. *Complexity:* $\mathcal{O}(|E|)$

13 Link Analysis

13.1 Jaccard Coefficient

Calculate the Jaccard coefficient of a graph

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

```
template <adjacency_list G, typename OutOp, typename T = double>
requires invocable<OutOp, vertex_id_t<G>, vertex_id_t<G>, edge_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<pair<vertex_id_t<G>,vertex_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 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `OutOp` satisfies `invocable<OutOp, vertex_id_t<G>, vertex_id_t<G>, edge_t<G>&, T>`.

2 *Preconditions:*

(2.1) — `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)`.

3 *Effects:*

(3.1) — 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.

4 *Complexity:*

(4.1) — **Time:** $\mathcal{O}(|V| + |E| \cdot d_{\min})$, where $d_{\min}$ is the minimum degree of either endpoint.

(4.2) — **Space:** $\mathcal{O}(|V| + |E|)$ auxiliary — precomputed neighbor sets.

14 Minimum Spanning Tree

14.1 Kruskal Minimum Spanning Tree

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

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

```
template <x_index_edgelist_range IELR, x_index_edgelist_range OELR,
          class Alloc = allocator<byte>>
auto kruskal(IELR&& e, OELR&& t, const Alloc& alloc = Alloc());

template <x_index_edgelist_range IELR, x_index_edgelist_range OELR, class CompareOp,
          class Alloc = allocator<byte>>
auto kruskal(IELR&& e, OELR&& t, CompareOp compare, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `IELR` satisfies `x_index_edgelist_range`.

(1.2) — `OELR` satisfies `x_index_edgelist_range`.

2 *Preconditions:*

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

3 *Effects:*

(3.1) — 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.

4 *Returns:* A `pair<EV, size_t>` containing the total weight of the spanning tree or forest and the number of connected components. *Complexity:*

(5.1) — **Time:** $\mathcal{O}(|E| \log |E|)$.

(5.2) — **Space:** $\mathcal{O}(|E| + |V|)$ auxiliary — edge copy and union-find.

6 *Remarks:* The optional `alloc` parameter supplies an allocator used for the internal disjoint-set (union-find) storage. Defaults to `std::allocator<std::byte>`.

14.2 Inplace Kruskal Minimum Spanning Tree

Memory-efficient variant of Kruskal's algorithm that sorts the input edge list in place rather than making a copy. The input range must satisfy `permutable<iterator_t<IELR>>`.

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

```
template <x_index_edgelist_range IELR, x_index_edgelist_range OELR,
          class Alloc = allocator<byte>>
requires permutable<iterator_t<IELR>>
auto inplace_kruskal(IELR&& e, OELR&& t, const Alloc& alloc = Alloc());

template <x_index_edgelist_range IELR, x_index_edgelist_range OELR, class CompareOp,
          class Alloc = allocator<byte>>
requires permutable<iterator_t<IELR>>
auto inplace_kruskal(IELR&& e, OELR&& t, CompareOp compare, const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `IELR` satisfies `x_index_edgelist_range`.

(1.2) — `OELR` satisfies `x_index_edgelist_range`.

(1.3) — `iterator_t<IELR>` satisfies `permutable`.

2 *Preconditions:*

(2.1) — `compare` operator, when provided, is a valid comparison operation on two edge values of type `range_value_t<EL>::value_type` which returns a bool.

3 *Effects:*

(3.1) — The input edge list `e` is sorted by edge weight (or by `compare` when provided).

(3.2) — 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.

4 *Returns:* A `pair<EV, size_t>` containing the total weight of the spanning tree or forest and the number of connected components. *Complexity:*

5

(5.1) — **Time:** $\mathcal{O}(|E| \log |E|)$.

(5.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary — union-find only; no edge copy.

6 *Remarks:* The input edge list is modified (sorted). If the caller needs to preserve the original order, use `kruskal` instead. The optional `alloc` parameter supplies an allocator used for the internal disjoint-set (union-find) storage. Defaults to `std::allocator<std::byte>`.

14.3 Prim Minimum Spanning Tree

Find the minimum weight spanning tree of a graph using Prim's algorithm. Delegates to `dijkstra_shortest_paths` with a projecting combine function that uses the edge weight directly rather than accumulating path distance.

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

```
template <adjacency_list G,
          class WeightFn,
          class PredecessorFn,
          class WF = function<distance_fn_value_t<WeightFn, G>(const remove_reference_t<G>&,
                                                               const edge_t<G>&)>,
          class CompareOp = less<distance_fn_value_t<WeightFn, G>>,
          class Heap = use_default_heap,
          class Alloc = allocator<byte>>
requires distance_fn_for<WeightFn, G> &&
         is_arithmetic_v<distance_fn_value_t<WeightFn, G>> &&
         predecessor_fn_for<PredecessorFn, G> &&
         basic_edge_weight_function<G, WF, distance_fn_value_t<WeightFn, G>, CompareOp,
                                    plus<distance_fn_value_t<WeightFn, G>>>
auto prim(G&& g,
          const vertex_id_t<G>& seed,
          WeightFn&& weight,
```

```
        PredecessorFn&& predecessor,
        WF&& weight_fn = [](const auto& gr, const edge_t<G>& uv) { return edge_value(gr, uv); },
        CompareOp compare = less<distance_fn_value_t<WeightFn, G>>(),
        Heap heap_tag = Heap{},
        const Alloc& alloc = Alloc());
```

1 *Mandates:*

(1.1) — `G` satisfies `adjacency_list<G>`.

(1.2) — `WeightFn` satisfies `distance_fn_for<WeightFn, G>` with `is_arithmetic_v<distance_fn_value_t<WeightFn, G>>`.

(1.3) — `PredecessorFn` satisfies `predecessor_fn_for<PredecessorFn, G>`.

(1.4) — `WF` satisfies `basic_edge_weight_function<G, WF, distance_fn_value_t<WeightFn, G>, CompareOp, plus<distance_fn_value_t<WeightFn, G>>>`.

(1.5) — `compare` is a valid strict weak ordering on values of type `distance_fn_value_t<WeightFn, G>`.

2 *Preconditions:*

(2.1) — When `weight_fn` is omitted, the default is `edge_value(g, uv)`.

3 *Hardened preconditions:*

(3.1) — `seed` is a valid vertex ID in `g`.

(3.2) — `weight(g, v)` and `predecessor(g, v)` have been initialized for each vertex `v` in `g` before calling (e.g. via `init_shortest_paths(g, weight_cont, pred_cont)` on the underlying containers, then wrapping with `container_value_fn`).

4 *Effects:*

(4.1) — `predecessor(g, v)` is the parent vertex of `v` in a tree rooted at `seed` and `weight(g, v)` is the value of the edge between `v` and `predecessor(g, v)` in the tree. `predecessor(g, seed)== seed`. When `compare` is `<`, `predecessor` represents a minimum weight spanning tree.

(4.2) — For vertices not reachable from `seed`, `predecessor(g, v)` and `weight(g, v)` retain their initialized values.

5 *Returns:* The total weight of the minimum spanning tree rooted at `seed`. *Throws:*

(6.1) — `std::out_of_range` if `seed` is not a valid vertex ID in `g`.

(6.2) — `std::out_of_range` if `predecessor` or `weight` are undersized for the graph.

(6.3) — `std::out_of_range` if a negative edge weight is encountered (for signed weight types).

(6.4) — `std::logic_error` if an internal invariant violation is detected.

7 *Complexity:*

(7.1) — **Time:** $\mathcal{O}(|E| \log |V|)$ with `use_default_heap`. With `use_indexed_dary_heap<d>` the heap size is bounded by $\mathcal{O}(|V|)$ and each operation costs $\mathcal{O}(\log_d |V|)$.

(7.2) — **Space:** $\mathcal{O}(|V|)$ auxiliary for `use_indexed_dary_heap`; up to $\mathcal{O}(|E|)$ for `use_default_heap` due to lazy deletion.

8 *Remarks:*

(8.1) — Only produces a spanning tree for the connected component containing `seed`. For disconnected graphs, call `prim` once per component with a seed vertex from each component.

(8.2) — The `Heap` template parameter selects the internal heap implementation (forwarded to `dijkstra_shortest_paths`). Pass `use_default_heap{}` (default) for sparse or grid-like graphs; pass `use_indexed_dary_heap<Arity>{}` for dense or hub-heavy graphs. See the Dijkstra Shortest Paths section for guidance on arity selection.

(8.3) — The optional `alloc` parameter supplies an allocator used for the internal priority queue (forwarded to `dijkstra_shortest_paths`). Defaults to `std::allocator<std::byte>`.

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.