Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 P1709r5
Date: 2022-12-06
Reply to: Phil Ratzloff (phil.ratzloff@sas.com),
Andrew Lumsdaine (lumsdaine@gmail.com)

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

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

Contents

Revision History

P1709R5

Extensions and refinements to r4

- Added basic_* versions for depth first search, breadth first search and topoloical sort views. Also shortended the view names to use bfs and dfs to avoid long names.
- Replace `adjacency_list` with `index_adjacency_list` concept in algorithms to simplify the definitions.
- Updated Shortest Paths algorithms with final definitions.
- Added Topological Sort algorithm description.
- Added summary table for `compressed_graph` .

P1709R4

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

- Revisit the algorithms to be considered.
- Reduce the scope to focus on an adjacency list with outgoing edges, edge list, and remove mutable interface functions.
- Replace directed and undirected concepts with overridable types of unordered_edge for a graph type.
- Simplify the Graph Container types and functions. In particular, const and non-const variations were consolidated to a single definition to handle both cases when appropriate.
- All Graph Container Interface functions are customization points.
- Introduce Views, inspired by NWGraph design, resulting in simpler and cleaner interfaces to traverse a graph, and simplifying the container interface design.
- Add support for bipartite and multipartite graphs.
- Replace the two container implementations with compressed_graph, based on the Compressed Sparse Row matrix, a commonly used data structure for high-performance graphs.

P1709R3

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

P1709R2

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

P1709R1

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

P1709R0

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

Chapter1 Overview

Graphs, used in ML and other **scientific** domains, as well as **industrial** and **general** programming, do **not** presently exist in the C++ standard. In ML, a graph forms the underlying structure of an **artificial neural network** (ANN). In a **game**, a graph can be used to represent the **map** of a game world. In **business** environments, graphs arise as **entity relationship diagrams** (ERD) or **data flow diagrams** (DFD). In the realm of **social media**, a graph represents a **social network**.

This document proposes the addition of **graph algorithms**, **graph views**, **graph container interface** and a **graph container implementation** to the C++ library to support **machine learning** (ML), as well as other applications. ML is a large and growing field, both in the **research community** and **industry**, that has received a great deal of attention in recent years. This paper presents an **interface** of the proposed algorithms, views, graph functions and containers.

1.1 Goals and Priorities

- Follow the separation of algorithms, ranges, views and containers established by the standard library.
- Graph algorithms have the following characteristics
  - Support syntax that is simple, expressive and easy to understand. This should not compromise the ability to write high-performance algorithms.
  - Vertices are required to be in random access containers with an integral vertex_id in this proposal.
- Graph views provide common traversals of a graph's vertices and edges that is more concise and consistant than using the graph container interface directly. They include simple traversals like vertexlist (all vertices in the graph) and incidence edges (edges on a vertex), as well as more complex traversals like depth-first and breath-first searches.
- All free functions are customization point objects, unless noted otherwise. Reasonable default implementations are provided whenever possible.
- The Graph Container Interface provides a consistent interface that can be used by algorithms and views. It has the following characteristics:
  - The interface models an adjacency graph container, which is an outer range of vertices with an inner range of outgoing (a.k.a. incidence) edges on each vertex.
  - Definition of concepts, types, type traits, type aliases, and functions used by algorithms and views.
    - Type traits will be defined that can be overridden for each graph container to give additional hints that can be used by algorithms to refine their behavior, such as adjacency_matrix and unordered_edge.
  - Support of optional user-defined value types on an edge, vertex and/or the graph itself.
  - Support bipartite and multipartite graphs, as long as the underlying graph supports it. If the underlying graph doesn't support either, it is considered unipartite with a single partition.
  - Allow for useful extensions of the graph data model in future proposals or in external graph implementations.
- Define an Edge List interface, required by some algorithms, that can be used by user-defined ranges for algorithms that require them.
- Provide an initial suite of useful functionality that includes algorithms, views, container interface, and at least one model container implementation.

1.2 Examples

The following code demonstrates how a simple graph can be created as a range of ranges, using the standard containers.

```
std::vector<std::string> actors { "Tom Cruise", "Kevin Bacon", "Hugo Weaving",
                          "Carrie-Anne Moss", "Natalie Portman", "Jack Nicholson",
                          "Kelly McGillis", "Harrison Ford", "Sebastian Stan",
                          "Mila Kunis", "Michelle Pfeiffer", "Keanu Reeves",
                          "Julia Roberts" };

using G = std::vector<std::vector<int>>;
```

```
auto target_id(const G& g, edge_reference_t<G> uv) {return get<0>(uv);}
G costar_adjacency_list{
   {1, 5, 6}, {7, 10, 0, 5, 12}, {4, 3, 11}, {2, 11}, {8, 9, 2, 12}, {0, 1}, {7, 0},
   {6, 1, 10}, {4, 9}, {4, 8}, {7, 1}, {2, 3}, {1, 4} };

int main() {
  std::vector<int> bacon_number(size(actors));

  // 1 -> Kevin Bacon
  for (auto&& [uid,vid] : basic_sourced_edges_bfs(costar_adjacency_list, 1)) {
    bacon_number[vid] = bacon_number[uid] + 1;
  }

  for (int i = 0; i < size(actors); ++i) {
    std::cout << actors[i] << " has Bacon number " << bacon_number[i] << std::endl;
  }
}
```

`target_id(g,uv)` defines the required function to get a target_id for an edge in the graph `G`. Other functions can also be overridden to allow a developer to adapt their own graph data structures to the library.

1.3 What this proposal is not

This paper limits itself to adjacency graphs and edgelists only. An adjacency graph is an outer range of vertices with an inner range of outgoing edges on each vertex. An edgelist is a view of edges, which is either all the edges in the adjacency graph or a projection of a user-defined range.

Parallel versions of the algorithms are not included for several reasons. The executors proposal in P2300r5 [1] is expected to introduce new and better ways to do parallel algorithms beyond that used in the parallel STL algorithms and we would like to wait for finalization of that proposal before committing to parallel implementations. Secondly, many graph algorithms don't benefit from parallel implementations so there is less need to offer an implementation. Lastly, it will help limit the size of this proposal which is already looking to be large without it. It is expected that future proposals will be submitted for parallel graph algorithms.

Incoming edges on a vertex are not included, though it is hoped that a future proposal will be made for them.

The algorithms and views in this proposal expect that vertex_ids are densly assigned in a random access range, but it does not exclude the possibility of sparsely-defined vertex_ids stored in containers like `std::map` or `std::unordered_map` in future proposals.

The algorithms and views in this proposal expect that vertex_ids are integal, but it does not exclude non-integral or user-defined types in future proposals.

Hypergraphs are not supported.

1.4 Impact on the Standard

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

1.5 Interaction wtih Other Papers

There is no interaction with other proposals to the standard.

1.6 Implementation Experience

The github github.com/stdgraph repository contains an implementation for this proposal.

1.7 Usage Experience

There is no current use of the library. There are plans to begin using it in the next year in a commercial setting.

1.8 Deployment Experience

There is no current deployment experience of the library. There are plans for this to follow the usage experience.

1.9 Performance Considerations

The algorithms are being ported from NWGraph to the github.com/stdgraph implementation used for this proposal. Performance analysis from those algorithms can be found in the peer-reviewed papers for NWGraph [2, 3].

1.10 Prior Art

boost::graph has been an important C++ graph implementation since 2001. It was developed with the goal of providing a modern (at the time) generic library that addressed all the needs someone would want of a graph library. It is still a viable library used today, attesting to the value it brings.

However, boost::graph was written using C++98 in an "expert-friendly" style, adding many abstractions and using sophisticated tempate metaprogramming, making it difficult to use by a casual developer.

(Andrew is a co-author of boost::graph.)

NWGraph ([4] and [2]) was published in 2022 by Lumsdaine et al, bringing additional experience gained since creating boost::graph, to create a modern graph library using C++20 for its implementation that was more accessible to the average developer.

While NWGraph made important strides to introduce the idea of the graph as a range-of-ranges and implemented many important algorithms, there are some areas it didn't address that come a practical use in the field. For instance, it didn't have a well-defined API for graph data structures that could be applied to existing graphs, and there wasn't a uniform approach to properties.

This proposal takes the best of NWGraph, with previous work done for P1709 to define a Graph Container Interface, to provide a library that embraces performance, ease-of-use and the ability to use the algorithms and views on externally defined graph containers.

1.11 Alternatives

There are no known alternative graph library we're aware of that meets the same requirements and uses concepts and ranges from C++20.

1.12 Feature Test Macro

The `__cpp_lib_graph` feature test macro is recommended to represent all features in this proposal including algorithms, views, concepts, traits, types, functions and graph container(s).

1.13 Freestanding

We believe this library can be used in a freestanding C++ implementation.

1.14 Namespaces

Graph containers and their views and algorithms are not interchangeable with existing containers and algorithms. Additionally, there are some domain-specific terms that may clash with existing or future names, such as `degree` and `partition_id`. For these reasons, we recommend their own namespaces as follows. This assumption is used in this proposal.

`std::graph`

`std::graph::views`

Alternative locations for the above respective namespaces could also be as follows:

`std::ranges`

`std::ranges::views`

Chapter2 Introduction

2.1 Motivation

The original STL revolutionized the way that C++ programmers could apply algorithms to different kinds of containers, by defining *generic* algorithms, realized via function templates. A hierarchy of *iterators* were the mechanism by which algorithms could be made generic with respect to different kinds of containers, Named requirements specified the valid expressions and associated types that algorithms required of their arguments. As of C++20, we now have both ranges and concepts, which now provide language-based mechanisms for specifying requirements for generic algorithms.

As powerful as the algorithms in the standard library are, the underlying basis for them is a range (or iterator pair), which inherently can only specify a one-dimensional container. Iterator pairs (equiv. ranges) specify a `begin()` and an `end()` and can move between those two limits in various ways, depending on the type of iterator. As a result, important classes of problems that programmers are regularly faced with use structures that are not one-dimensional containers, and so the standard library algorithms can't be directly used. Multi-dimensional arrays are an example of one such kind of data structure. Matrices do have the nice property that they (typically) have the ability to be "raveled", i.e., the data underlying the matrix can still be treated as a one-dimensional container. Multi-dimensional arrays also have the property that, even though they can be thought of as hierarchical containers, the hierarchy is uniform—an N-dimensional array is a container of N-1 dimensional arrays.

Another important problem domain that does not fit into the category of one-dimensional ranges is that of *graph algorithms and data structures*. Graphs are a powerful abstraction for modeling relationships between entities in a given problem domain, irrespective of what the actual entities are, and irrespective of what the actual relationships are. In that sense, graphs are, by there very nature, generic. Graphs are a fundamental abstraction in computer science, and are ubiquitous in real-world applications.

Any problem concerned with connectivity can be modeled as a graph. Just a small set of examples include Internet routing, circuit partitioning and layout, finding the best route to take to a destination on map. There are also relationships between entities that are inferred from large sets of data, for example the graph of consumers who have purchased the same product, or who have viewed the same movie. Yet more interesting structures arise (hypergraphs or k-partite graphs) can arise when we want to model relationships between diverse types of data, such as the graph of consumers, the products they have purchased, and the vendors of the products. And, of course, graphs play a critical role in multiple aspects of machine learning.

On the flip side of graph structures are the graph algorithms that are widely used for problems such as the above. Well-known graph algorithms include breadth-first search, Dijkstra's algorithm, connected components, and so on. Because graphs can come from so many different problem domains, they will also be represented with many different kinds of data structures. To make graph algorithms as usable as possible across arbitrary representation requires application of the same principles that were used in the original STL: a collection of related algorithms from a problem domain (in our case, graphs), minimizing the requirements imposed by the algorithms on their arguments, systematically organizing the requirements, and realizing this framework of requirements in the form of concepts.

There are also many uses of graphs that would not be met by a standard set of algorithms. A standardized interface for graphs is eminently useful in such situations as well. In the most basic case, it would provide a well-defined framework for development. But in keeping with the foundational goal of generic programming to enable reuse, it would also empower users to develop and deploy their own reusable graph components. In the best case, such algorithms would be available to the broader C++ programmer community.

Because graphs are so ubiquitous and so important to modern software systems, a standardized library of graph algorithms and data structures would have enormous benefit to the C++ development community. This proposal contains the specification of such a library, developed using the principles above.

2.2 Example: Six Degrees of Kevin Bacon

A classic example of the use of a graph algorithm is the game "The Six Degrees of Kevin Bacon." The game is played by connecting actors to each other through movies they have appeared in together. The goal is to find the smallest number of movies that connect a given actor to Kevin Bacon. That number is called the "Bacon number" of the actor. Kevin Bacon himself has a Bacon number of 0. Since Kevin Bacon appeared with Tom Cruise in "A Few Good Men", Tom Cruise has a Bacon number of 1.

The following program computes the Bacon number for a small selection of actors.

```
std::vector<std::string> actors { "Tom Cruise", "Kevin Bacon", "Hugo Weaving",
                          "Carrie-Anne Moss", "Natalie Portman", "Jack Nicholson",
                          "Kelly McGillis", "Harrison Ford", "Sebastian Stan",
                          "Mila Kunis", "Michelle Pfeiffer", "Keanu Reeves",
                          "Julia Roberts" };

using G = std::vector<std::vector<int>>;
auto target_id(const G& g, edge_reference_t<G> uv) {return get<0>(uv);}
G costar_adjacency_list{
   {1, 5, 6}, {7, 10, 0, 5, 12}, {4, 3, 11}, {2, 11}, {8, 9, 2, 12}, {0, 1}, {7, 0},
   {6, 1, 10}, {4, 9}, {4, 8}, {7, 1}, {2, 3}, {1, 4} };

int main() {
  std::vector<int> bacon_number(size(actors));

  // 1 -> Kevin Bacon
  for (auto&& [uid,vid] : basic_sourced_edges_bfs(costar_adjacency_list, 1)) {
    bacon_number[vid] = bacon_number[uid] + 1;
  }

  for (int i = 0; i < size(actors); ++i) {
    std::cout << actors[i] << " has Bacon number " << bacon_number[i] << std::endl;
  }
}
```

Output:

```
Tom Cruise has Bacon number 1
Kevin Bacon has Bacon number 0
Hugo Weaving has Bacon number 3
Carrie-Anne Moss has Bacon number 4
Natalie Portman has Bacon number 2
Jack Nicholson has Bacon number 1
Kelly McGillis has Bacon number 2
Harrison Ford has Bacon number 1
Sebastian Stan has Bacon number 3
Mila Kunis has Bacon number 3
Michelle Pfeiffer has Bacon number 1
Keanu Reeves has Bacon number 4
Julia Roberts has Bacon number 1
```

In graph parlance, we are creating a graph where the vertices are actors and the edges are movies. The number of movies that connect an actor to Kevin Bacon is the shortest path in the graph from Kevin Bacon to that actor. In the example above, we compute shortest paths from Kevin Bacon to all other actors and print the results. Note, however, that actor-actor relationships are not how data about actors is available in the wild (from IMDB, for example). Rather, two types of relationships available are actor-movie and movie-actor. See Section **??** below.

2.3 Graph Background

For clarity, we briefly review some of the basic terminology of graphs. We use commonly accepted terminology for graph data structures and algorithms and adopt the particular terminology used in the textbook by Cormen, Leiserson, Rivest, and Stein ("CLRS") [5].

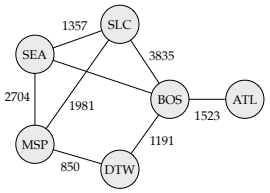
2.3.1 Basic Terminology

To model the relationships between entities, a *graph* G comprises two sets: a *vertex set* V, whose elements correspond to the entities, and an *edge set* E, whose elements are pairs corresponding to elements in V that have some relationship with each other. That is, if u and v are members of V that have some relationship that we wish to capture, then there is a pair $\{u, v\}$ in E. We can express that together V and E define a graph as $G = \{V, E\}$.

Two examples of graph models are shown in Figures **??** and **??**, which respectively model a network of routes between and an electronic circuit. The figures show the domain-specific data to be modeled and the sets V and E for each graph. Also shown for each graph is a node and link diagram, a commonly-used graphical[1] notation.

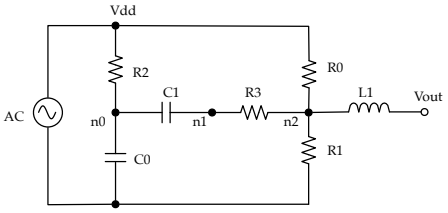
Airport
SEA
MSP
SLC
DTW
ATL
BOS

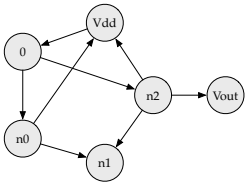
Distance (km)		
MSP	DTW	850
SLC	SEA	1357
MSP	SLC	1981
BOS	SLC	3835
SEA	BOS	4016
BOS	ATL	1523
SEA	MSP	2704
BOS	DTW	1191



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

(a) An undirected graph representing airline routes between cities. Shown are the list of airports (the vertices) and the list of routes between them (the edges). Also shown are a node and link diagram and the set-based description.





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

(b) A directed graph representing an electronic circuit. Shown in the circuit diagram are the labeled circuit nodes (the vertices) and the circuit elements connecting the nodes (the edges). Since circuit elements are oriented, we use a directed graph to model the circuit. Also shown are a node and link diagram and the set-based description.

Figure 2.1 — Graph models of an airline route system and of an electronic circuit.

2.3.2 Graph Representation: Enumerating the Vertices

To reason about graphs, and to write algorithms for them, we require a *representation* of the graph. We note that *a graph and its representation are not the same thing*. It is therefore essential that we be precise about this distinction as we develop a software library of graph algorithms and data structures[2].

The representations that we will be using are familiar ones: adjacency matrix, edge list, and adjacency list. We begin with a process that is so standard that we typically don't even notice it, but it forms the foundation of graph representations: we *enumerate the vertices.* That is, we assign an index to each element of V and write $V = \{v_0, v_1, \ldots v_{n-1}\}$. Based on that enumeration, elements of E are expressed in the form $\{v_i, v_j\}$. Similarly, we can enumerate the edges, and write $E = \{e_0, e_1, \ldots e_{m-1}\}$, though the enumeration of E does not play a role in standard representations of graphs. The number of elements in V is denoted by $|V|$ and the number of elements in E is denoted by $|E|$.

We summarize some remaining terminology about vertices and edges.

— An edge e_k may be *directed*, denoted as the ordered pair $e_k = (v_i, v_j)$, or it may be *undirected*, denoted as the (unordered) set $e_k = \{v_i, v_j\}$. The edges in E are either all directed or all undirected, corresponding respectively to a *directed graph* or to an *undirected* graph.

— If the edge set E of a directed graph contains an edge $e_k = (v_i, v_j)$, then vertex v_j is said to be *adjacent* to vertex v_i. The edge e_k is an *out-edge* of vertex v_i and an in-edge of vertex v_j. Vertex v_i is the *source* of edge e_k, while v_j is the *target* of edge e_k.

— If the edge set E of an undirected graph contains an edge $e_k = \{v_i, v_j\}$, then e_k is said to be *incident* on the vertices v_i and v_j. Moreover, vertex v_j is adjacent to vertex v_i *and* vertex v_i is adjacent to vertex v_j. The edge e_k is an out-edge of both v_i and v_j and it is an in-edge of both v_i and v_j.

— The *neighbors* of a vertex v_i are all the vertices v_j that are adjacent to v_i. The set of all of the neighbors is the *neighborhood* of v_i.

[1] An unfortunate collision of terminology.

[2] In fact, if we are to be completely precise, the library we are proposing is one of algorithms and data structures for graph representations. We will make concessions to commonly accepted terminology, while precisely defining that terminology.

— A *path* as a sequence of vertices $v_0, v_1, \ldots, v_{k-1}$ such that there is an edge from v_0 to v_1, an edge from v_1 to v_2, and so on. That is, a path is a set of edges $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k-2$.

2.3.3 Adjacency-Based Representations

We begin our development of graph representations with the almost universally-accepted definition of the adjacency matrix representation of a graph. The *adjacency matrix representation* of a graph G is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that, respectively for a directed or undirected graph

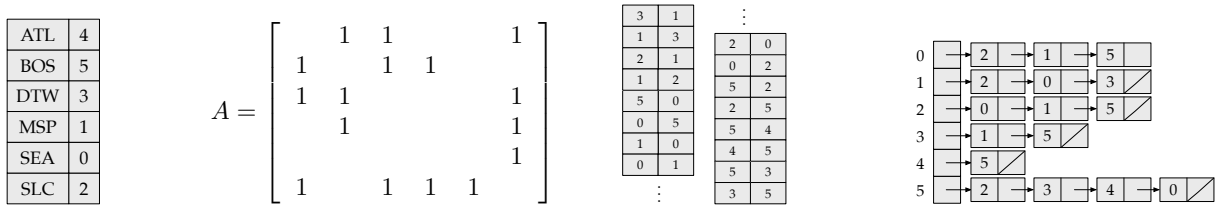
$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \qquad a_{ij} = a_{ji} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

That is, $a_{ij} = 1$ if and only if v_j is adjacent to v_i in the original graph G (hence the name "adjacency matrix"). Here we can see why we said that the initial enumeration of V is foundational to representations: *The adjacency matrix is based solely on the indices used in that enumeration*. It does not contain the vertices or edges themselves.

As a data structure to use for algorithms, the adjacency matrix is not very efficient, neither in terms of storage (which, at $|V| \times |V|$ is prohibitive), nor for computation. Instead of storing the entire adjacency matrix, we can simply store the index values of its non-zero elements. A *sparse coordinate adjacency matrix* is a container C of pairs (i, j) for every a_{ij} in A. At first glance, it may seem that we have simply created a data structure C that has a pair (i, j) if E in the original graph has an edge from v_i to v_j. This is true in the directed case. However, in the undirected case, if there is an edge between v_i and v_j, then v_i is adjacent to v_j and v_j is adjacent to v_i. In other words, if there is an edge between v_i and v_j in an undirected graph, then both the entries a_{ij} and a_{ji} are equal to 1[3] — and therefore for a single edge between v_i and v_j, C contains two index pairs: (i, j) and (j, i). The sparse coordinate representation is commonly known as *edge list*. However, we caution the reader that C does not store edges, but rather indices and that, in the case that it represents an undirected graph, there is not a 1-1 correspondence between the edges in E and the contents of C.

Although the sparse coordinate adjacency matrix is much more efficient in terms of storage than the original adjacency matrix, it isn't as efficient as it could be. Much more importantly, it is not useful for the types of operations used by most graph algorithms, which need to be able to get the set of neighbors of a given vertex in constant time. To support this type of operation, we use a *compressed sparse adjacency matrix*, which is an array J with $|V|$ entries, where each $J[i]$ is a linear container of indices $\{j\}$ such that v_j is a neighbor of v_i in G. That is j is contained in $J[i]$ if and only if there is an edge (v_i, v_j) in E (or, equivalently, if there is a pair (i, j) in C or, equivalently, if $a_{ij} = 1$)[4]. We note that if (v_i, v_j) is an edge in an undirected graph, $J[i]$ will contain j and $J[j]$ will contain i. The common name for this data structure is *adjacency list*. Although this name is problematic (for instance, it is not actually a list), it is so widely used that we also use it here—but *we mean specifically that an "adjacency list" is the compressed sparse adjacency matrix representation of a graph*[5]. Again we emphasize the distinction between a graph and its representation: An adjacency list J is not the same as the graph G—it is a representation of G.

Illustrations of the adjacency-matrix representations of the airline route graph and the electronic circuit graph are shown in Figures **??** and **??**, respectively.



(a) An enumeration of the airport graph given in 2.1a.

(b) The adjacency matrix representation of the graph given in Figure 2.1a, using the enumeration given in Figure 2.2a.

(c) The coordinate sparse adjacency matrix representation (shown split into two columns).

(d) The compressed sparse adjacency matrix representation.

Figure 2.2 — Adjacency matrix representations of the airport graph model.

0	0
n0	1
Vdd	2
n1	3
Vout	4
n2	5

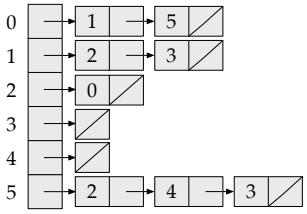
(a) An enumeration of the circuit graph given in 2.1b.

$$A = \begin{bmatrix} & 1 & & & & 1 \\ & & 1 & 1 & & \\ 1 & & & & & \\ & & & & & \\ & & & & & \\ & & 1 & 1 & 1 & \end{bmatrix}$$

(b) The adjacency matrix representation of the graph given in Figure 2.1b, using the enumeration given in Figure 2.3a.

1	3
2	0
1	2
5	2
0	5
5	4
0	1
5	3

(c) The coordinate sparse adjacency matrix representation.



(d) The compressed sparse adjacency matrix representation.

Figure 2.3 — Adjacency matrix representations of the circuit graph model.

0	Tom Cruise
1	Kevin Bacon
3	Carrie-Ann Moss
4	Natalie Portman
2	Hugo Weaving
5	Kelly McGillis

(a) Table of actors.

0	A Few Good Men
1	Top Gun
2	V for Vendetta
3	Black Swan
4	The Matrix

(b) Table of movies.

Tom Cruise	Top Gun, A Few Good Men
Kevin Bacon	A Few Good Men
Hugo Weaving	V for Vendetta, The Matrix
Carrie-Ann Moss	The Matrix
Natalie Portman	Black Swan, V for Vendetta
Kelly McGillis	Top Gun

(c) A table of actors and movies they have appeared in.

A Few Good Men	Tom Cruise, Kevin Bacon
Top Gun	Kelly McGillis, Tom Cruise
V for Vendetta	Hugo Weaving, Natalie Portman
Black Swan	Natalie Portman
The Matrix	Carrie-Ann Moss, Hugo Weaving

(d) A table of movies with starring actors.

Figure 2.4 — Illustrative simplification of IMDB actor and movie data.

2.4 Bipartite Graphs

So far, we have been considering graphs where edges in E are pairs of vertices, which are taken from a single set V. We refer to such a graph as a *unipartite* graph. But consider again the Kevin Bacon example. The source for the information comprising the Kevin Bacon data is the Internet Movie Database (IMDB). However, the IMDB does not contain any explicit information about the relationships between actors. Rather it contains files of tabular data, one of which contains an entry for each movie with the list of actors that have appeared in that movie, and another of which contains an entry for each actor with the list of movies that actor has appeared in ("movie-actor" and "actor-movie" tables, respectively). Such tables are shown in Figure 2.4.[6] Thus, a graph, as we have defined it, cannot model the IMDB.

There is a small generalization we can make to the definition of graph that will result in a suitable abstraction for modeling the IMDB. In particular, we need one set of vertices corresponding to actors, another set of vertices corresponding to movies, and then a set of edges corresponding to the relationships between actors and movies. There are two kinds of relationships to consider actors in movies or movies starring actors. To be well-defined, the edge set may only contain one kind of relationship. To capture this kind of model, we define a *structurally bipartite graph* $H = \{U, V, E\}$, where vertex sets U and V are enumerated $U = \{u_0, u_1, \ldots, u_{n0}\}$ and $V = \{v_0, v_1, \ldots v_{n1}\}$, and the edge set E consists of pairs (u_i, v_j) where u_i is in U and v_j is in V.

The *adjacency matrix representation of a structurally bipartite graph* is a $|U| \times |V|$ matrix $A = (a_{ij})$ such that,

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

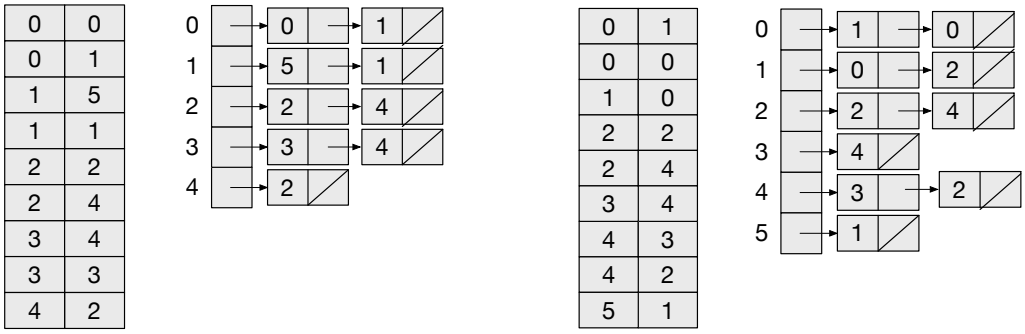
From this adjacency matrix representation we can readily construct coordinate and compressed sparse representations. The only structural difference between the representations of a structurally bipartite graph and that of a unipartite graph is that of vertex cardinality. That is, in a unipartite graph, edges map from V to V, and hence the values in the left hand column and in the right hand column of a coordinate representation would be in the same range: $[0, |V|)$. However, for a structurally bipartite graph, this is no longer the case. Although the coordinate representation still consists of pairs of vertex indices, the range of values in the left hand column is $[0, |U|)$, while in the right hand column it is $[0, |V|)$. Similarly, the compressed representation will have $|U|$ entries, but the values stored in each entry may range from $[0, |V|)$. We note that these are constraints on values, not on structure.

[3] That is, the adjacency matrix is symmetric.

[4] The compressed sparse adjacency matrix is identical to the compressed sparse row format from linear algebra

[5] We concede that "adjacency list" rolls off the tongue much more easily than "compressed sparse adjacency matrix representation of a graph."

[6] This is a greatly simplified version of the CSV files that actually comprise the IMDB. The full set of files is available for non-commercial use at https://datasets.imdbws.com.



(a) Coordinate and compressed sparse adjacency representations for movies with their starring actors.

(b) Coordinate and compressed sparse adjacency representations for actors and the movies they have appeared in.

Figure 2.5 — Sparse adjacency representations (edge lists and adjacency lists) for IMDB actor and movie data.

We distinguish a structurally bipartite graph from simply a bipartite graph because the former applies separate enumerations to U and V. In customary graph terminology, a *bipartite* graph is one in which the vertices can be partitioned into two disjoint sets, such that all of the edges in the graph only connect vertices from one set to vertices of the other set. However, although the vertices are partitioned, they are still taken from the same original vertex set V and have a single enumeration. Whether a graph can be partitioned in this way is a run-time property inherent to the graph itself (which can be discovered with an appropriate algorithm). This is not a natural way to model separate categories of entities, such as movies and actors, where entities are categorized completely independently of each other and it is therefore most appropriate to have independent enumerations for them. A structurally bipartite graph explicitly captures distinct vertex categories.

2.5 Partitioned Graphs

In contrast to structurally bipartite graphs, there are certainly cases where one would want to maintain two categories of entities, or otherwise distinguish the vertices, from the same vertex set. In that case, we would use a *partitioned graph*, which we define as $G = \{V, E\}$, where the vertex set V consists of non-overlapping subsets, i.e., $V = \{V_0, V_1, \ldots\}$ which we enumerate as $V_0 = \{v_0, v_1, \ldots, v_{n0-1}\}$, $V_1 = \{v_{n0}, \ldots, v_{n1-1}\}$ and so on. Each V_i is a *partition* of V. The total enumeration of V is $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Just as each V_i is a partition of V, the enumeration of each V_i is a partitioning of the enumeration of V.

The edge set E still consists of edges (v_i, v_j) (or $\{v_i, v_j\}$ where, in general, v_i and v_j may come from any partition.

We note that partitioned graphs are not restricted to two partitions—a partitioned graph can represent an arbitrary number of partitions, i.e., a *multipartite* graph (a graph with multiple subsets of vertices such that edges only go between subsets). While partitioned graphs can be used to model multipartite graphs, partitioned graphs are not necessarily multipartite; edges can comprise vertices within a partition as well as well as across partitions.

2.6 From Data to Graph

2.6.1 Columnar Data

Here we show how one might create an unlabeled edge list from a table of data stored in a CSV file. The following loads a list of directed edges from a CSV file (the values in each row are assumed to be separated by whitespace)[7]. The elements of the first column are considered to be the source vertices and the elements of the second column are the destination vertices. If the edges also had properties, the third column would contain the property values. In this example, the edges are loaded into a vector of tuples, which meets the requirements of a (presumed) `sparse_coordinate` concept.

```
auto sparse_coordinate edges = std::vector<std::tuple<vertex_id_t, vertex_id_t>;
auto input = std::ifstream ("input.csv");
vertex_id_t src, dst;
while (input >> src >> dst) {
   edges.emplace_back (src, dst);
}
```

[7] We take a broad view of what a comma is.

Similarly, we could load a list of undirected edges from a CSV file into a `sparse_coordinate` structure. Note that, as discussed above, the coordinate sparse adjacency matrix representation (aka an edge list), contains an entry (i, j) as well as an entry (j, i) for each undirected edge $\{v_i, v_j\}$. Hence, we add both `(src, dst)` and `(dst, src)` to `edges`.

```
auto sparse_coordinate edges = std::vector<std::tuple<vertex_id_t, vertex_id_t, double>
    edges;
auto input = std::ifstream ("input.csv");
vertex_id_t src, dst;
double val;
while (input >> src >> dst >> val) {
   edges.emplace_back (src, dst, val);
   edges.emplace_back (dst, src, val);
}
```

These examples are meant to be illustrative and not necessarily comprehensive (nor efficient). There are, of course, many ways to define containers that meet the requirements of the edge list concept and many ways to create an edge list from columnar data.

2.6.2 Converting an Edge List to an Adjacency List

The following creates a compressed sparse representation (an adjacency list) from a coordinate sparse representation. The adjacency list is represented as a `std::vector<std::vector<vertex_id_t>>;`

```
auto sparse_coordinate edges = std::vector<std::tuple<vertex_id_t, vertex_id_t>;
// Read the edges
auto sparse_compressed adj_list = std::vector<std::vector<vertex_id_t>>;
for (auto [src, dst] : edges) {
  if (src >= adj_list.size()) {
    adj_list.resize(src + 1);
  }
  adj_list[src].push_back (dst);
}
```

We note that the `sparse_coordinate` representation is agnostic as to whether it was originally created based on directed edges or undirected edges. An optimization to the sparse coordinate representation would be to use a *packed coordinate* representation, which would only maintain a single entry for each undirected edge. In that case, we would need to have two complementary insertions into the adjacency list for each entry in the packed coordinate representation.

The following example illustrates the use of a packed coordinate format to construct an adjacency list with an edge property.

```
auto packed_sparse_coordinate edges = std::vector<std::tuple<vertex_id_t, vertex_id_t,
    double>>;
// Read the edges
auto compressed_sparse adj_list = std::vector<std::vector<std::tuple<vertex_id_t, double
    >>>(edges.num_vertices();
for (auto [src, dst, val] : edges) {
  adj_list[src].push_back (dst, val);
  adj_list[dst].push_back (src, val);
}
```

Chapter3 Algorithms

Our proposed set of algorithms are grouped into Tier 1, Tier 2, and Tier 3. All Tier 1 algorithms are included in this proposal and summarized in the lists below. Other tiers are outlined in the section 3.12 Other Algorithms.

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

Clustering
- — Triangle counting

Communities
- — Label propagation

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

Directed Acyclic Graphs
- — Topological sort

Maximal Independent Set
- — Maximal independent set

Link Analysis
- — Jaccard coefficient

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

3.1 Introduction

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

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

The parts of the table have the following meaning:

- — **Complexity** The complexity of the algorithm based on the number of vertices (V) and edges (E).
- — **Throws?** Will the algorithm throw at all? If so, look at the *Throws* section after the function prototypes for details.
- — **Multi-edge?** Does the algorithm act as expected if more than one edge with the same direction exists between the same two vertices?
- — **Cycles?** Does the algorithm act act as expected if a vertex (or edge) is part of a cycle?
- — **Directed?** Is the algorithm only for directed graphs, or can it also be used for undirected graphs that have complimentary edges, with different directions, between two vertices.

3.2 Algorithm Concepts

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

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

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

template <class G, class WF, class DistanceValue>
concept edge_weight_function = // e.g.  weight(uv)
```

```
    is_arithmetic_v<invoke_result_t<WF, edge_reference_t<G>>> &&
    basic_edge_weight_function<G,
                          WF,
                          DistanceValue,
                          less<DistanceValue>,
                          plus<DistanceValue>>;
```

3.3 Shortest Paths

3.3.1 Unweighted Shortest Paths: Breadth-First Search

3.3.1.1 Breadth-First Search, Single Source, Initialization

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

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

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

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

Effects:

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

3.3.1.2 Breadth-First Search, Single Source

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

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

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

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class Allocator = allocator<vertex_id_t<G>>
        >
requires is_arithmetic_v<ranges::range_value_t<Distances>> &&
       convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessors>>
void breadth_first_search(
    G&& g, // graph
    vertex_id_t<G> source, // starting vertex_id
    Distances& distances, // out:  Distances[uid] of uid from source in number of edges
    Predecessors& predecessors, // out:  predecessor[uid] of uid in path
```

```
    Allocator alloc = Allocator());

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

1 *Preconditions:*

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

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

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

2 *Effects:*

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

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

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

3.3.2 Weighted Shortest Paths

3.3.2.1 Shortest Paths Initialization

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

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

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

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

1 *Effects:* :

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

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

2 *Returns:*

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

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

3.3.2.2 Dijkstra Single Source Shortest Paths and Shortest Distances

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

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

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

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

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

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

```
template <index_adjacency_list G,
        ranges::random_access_range Distances,
        ranges::random_access_range Predecessors,
        class Compare,
        class Combine,
        class WF = function<ranges::range_value_t<Distances>(edge_reference_t<G>)>,
```

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

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

1 *Mandates:*

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

2 *Preconditions:*

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

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

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

3 *Effects:*

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

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

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

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

3.3.2.3 Bellman-Ford Single Source Shortest Paths and Shortest Distances

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

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

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

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

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

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

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

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

1 *Preconditions:*

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

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

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

2 *Effects:*

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

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

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

4 *Remarks:*

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

3.4 Clustering

3.4.1 Triangle Counting

Compute the number of triangles in a graph.

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

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

1 *Returns:* Number of triangles

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

3.5 Communities

3.5.1 Label Propagation

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

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

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

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

1 *Preconditions:*

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

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

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

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

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

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

4 *Preconditions:*

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

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

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

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

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

3.6 Components

3.6.1 Articulation Points

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

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

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

1 *Preconditions:*

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

2 *Effects:*

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

3.6.2 BiConnected Components

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

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

```
template <index_adjacency_list G,
          ranges::forward_range OuterContainer>
requires ranges::forward_range<std::ranges::range_value_t<OuterContainer>> &&
         integral<ranges::forward_range_t<ranges::forward_range_t<OuterContainer>>>
void biconnected_components(G&& g,
                            OuterContainer& components);
```

1 *Preconditions:*

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

2 *Effects:*

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

3.6.3 Connected Components

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

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

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

1 *Preconditions:*

(1.1) — Size of `component` is greater than or equal to `num_vertices(g)`.

2 *Effects:*

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

3.6.4 Strongly Connected Components

3.6.4.1 Kosaraju's SCC

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

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

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

1 *Preconditions:*

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

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

2 *Effects:*

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

3.6.4.2 Tarjan's SCC

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

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

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

1 *Preconditions:*

(1.1) — Size of `component` is greater than or equal to `num_vertices(g)`.

2 *Effects:*

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

3.7 Directed Acyclic Graphs

3.7.1 Topological Sort, Single Source

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

3.7.1.1 Initialization

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

Effects:

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

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

3.7.1.2 Topological Sort, Single Source

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

1 *Preconditions:*

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

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

2 *Effects:*

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

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

3.8 Maximal Independent Set

3.8.1 Maximal Independent Set

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

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

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

1 *Preconditions:*

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

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

2 *Effects:*

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

3.9 Link Analysis

3.9.1 Jaccard Coefficient

Calculate the Jaccard coefficient of a graph

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

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

1 *Preconditions:*

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

2 *Effects:*

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

3.10 Minimum Spanning Tree

3.10.1 Kruskal Minimum Spanning Tree

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

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

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

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

1 *Preconditions:*

(1.1) — `e` is an `edgelist`.

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

2 *Effects:*

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

3.10.2 Prim Minimum Spanning Tree

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

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

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

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

1 *Preconditions:*

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

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

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

2 *Effects:*

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

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

3.11 Operators

3.11.1 Degree

3.11.2 Join

3.11.3 Relabel

3.11.4 Sort

3.11.5 Transpose

3.12 Other Algorithms

Additional algorithms that were considered but not included in this proposal are identified in Table 3.1. It is assumed that future proposals will include them, with a recommendation of each Tier being in its own proposal. Tier X algorithms are variations of shortest paths algorithms that complement the Single Source, Multiple Target algorithms in this proposal.

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

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

Table 3.1 — Other Algorithms

Chapter4 Views

The views in this section provide common ways that algorithms use to traverse graphs. They are a simple as iterating through the set of vertices, or more complex ways such as depth-first search and breadth-first search. The also provide a consistent and reliable way to access related elements using the View Return Types, and guaranteeing expected values, such as that the target is really the target on unordered edges.

4.1 Return Types (Descriptors)

Views return one of the types in this section, providing a consistent set of values. They are templated so that the view can adjust the actual values returned to be appropriate for its use. The three types, `vertex_descriptor`, `edge_descriptor` and `neighbor_descriptor` , define the data model used by the algorithms.

The following examples show the general design and how it's used. While it focuses on vertexlist to iterate over all vertices, it applies to all descriptors and view functions.

```
// the type of uu is vertex_descriptor<vertex_id_t<G>, vertex_reference_t<G>, void>
for(auto&& uu : vertexlist(g)) {
  vertex_id<G> id = uu.id;
  vertex_reference_t<G> u = uu.vertex;
  // ...  do something interesting
}
```

Structured bindings make it simpler.

```
for(auto&& [id, u] : vertexlist(g)) {
  // ...  do something interesting
}
```

A function object can also be passed to return a value from the vertex. In this case, `vertexlist(g)` returns `vertex_descriptor <vertex_id_t<G>, vertex_reference_t<G>, decltype(vvf(u))>` .

```
// the type returned by vertexlist is
// vertex_descriptor<vertex_id_t<G>,
// vertex_reference_t<G>,
// decltype(vvf(vertex_reference_t<G>))>
auto vvf = [&g](vertex_reference_t<G> u) { return vertex_value(g,u); };
for(auto&& [id, u, value] : vertexlist(g, vvf)) {
  // ...  do something interesting
}
```

A simpler version also exists if all you need is a vertex id. The vertex value function takes a vertex id instead of a vertex reference.

```
for(auto&& [uid] : basic_vertexlist(g)) {
  // ...  do something interesting
}

auto vvf = [&g](vertex_id_t<G> uid) { return vertex_value(g,uid); };
for(auto&& [uid, value] : basic_vertexlist(g,vvf)) {
  // ...  do something interesting
}
```

4.1.0.1 `struct vertex_descriptor<VId, V, VV>`

`vertex_descriptor` is used to return vertex information. It is used by `vertexlist(g)` , `vertices_breadth_first_search (g,u)` , `vertices_dfs(g,u)` and others. The `id` member always exists.

```
template <class VId, class V, class VV>
struct vertex_descriptor {
  using id_type = VId; // e.g.  vertex_id_t<G>
  using vertex_type = V; // e.g.  vertex_reference_t<G> or void
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  id_type id;
  vertex_type vertex;
  value_type value;
};
```

Specializations are defined with `V=void` or `VV=void` to suppress the existance of their associated member variables, giving the following valid combinations in Table 4.1 . For instance, the second entry, `vertex_descriptor<VId, V>` has two members `{id_type id; vertex_type vertex;}` and `value_type` is `void` .

Template Arguments	Members		
`vertex_descriptor<VId, V, VV>`	`id`	`vertex`	`value`
`vertex_descriptor<VId, V, void>`	`id`	`vertex`	
`vertex_descriptor<VId, void, VV>`	`id`		`value`
`vertex_descriptor<VId, void, void>`	`id`		

Table 4.1 — `vertex_descriptor` Members

A useful type alias for copying vertex values (excluding the vertex reference) is also available.

```
template <class VId, class VV>
using copyable_vertex_t = vertex_descriptor<VId, void, VV>; // id, value
```

4.1.0.2 `struct edge_descriptor<VId, Sourced, E, EV>`

`edge_descriptor` is used to return edge information. It is used by `incidence(g,u)`, `edgelist(g)`, `edges_breadth_first_searc(g,u)`, `edges_dfs(g,u)` and others. When `Sourced=true` , the `source_id` member is included with type `VId` . The `target_id` member always exists.

```
template <class VId, bool Sourced, class E, class EV>
struct edge_descriptor {
  using source_id_type = VId; // e.g.  vertex_id_t<G> when SourceId==true, or void
  using target_id_type = VId; // e.g.  vertex_id_t<G>
  using edge_type = E; // e.g.  edge_reference_t<G> or void
  using value_type = EV; // e.g.  edge_value_t<G> or void

  source_id_type source_id;
  target_id_type target_id;
  edge_type edge;
  value_type value;
};
```

Specializations are defined with `Sourced=true|false` , `E=void` or `EV=void` to suppress the existance of the associated member variables, giving the following valid combinations in Table 4.2 . For instance, the second entry, `edge_descriptor<VId,true,E>` has three members `{source_id_type source_id; target_id_type target_id; edge_type edge;}` and `value_type` is `void` .

A useful type alias for copying edge values (excluding the edge reference) is also available.

```
template <class VId, class EV>
using copyable_edge_t = edge_descriptor<VId, true, void, EV>; // source_id,target_id[,value]
```

Template Arguments	Members			
edge_descriptor<VId, true, E, EV>	source_id	target_id	edge	value
edge_descriptor<VId, true, E, void>	source_id	target_id	edge	
edge_descriptor<VId, true, void, EV>	source_id	target_id		value
edge_descriptor<VId, true, void, void>	source_id	target_id		
edge_descriptor<VId, false, E, EV>		target_id	edge	value
edge_descriptor<VId, false, E, void>		target_id	edge	
edge_descriptor<VId, false, void, EV>		target_id		value
edge_descriptor<VId, false, void, void>		target_id		

Table 4.2 — edge_descriptor Members

4.1.0.3 struct neighbor_descriptor<VId, Sourced, V, VV>

neighbor_descriptor is used to return information for a neighbor vertex, through an edge. It is used by neighbors(g,u). When Sourced=true, the source_id member is included with type source_id_type. The target_id member always exists.

```
template <class VId, bool Sourced, class V, class VV>
struct neighbor_descriptor {
  using source_id_type = VId; // e.g.  vertex_id_t<G> when Sourced==true, or void
  using target_id_type = VId; // e.g.  vertex_id_t<G>
  using vertex_type = V; // e.g.  vertex_reference_t<G> or void
  using value_type = VV; // e.g.  vertex_value_t<G> or void

  source_id_type source_id;
  target_id_type target_id;
  vertex_type target;
  value_type value;
};
```

Specializations are defined with Sourced=true|false, E=void or EV=void to suppress the existance of the associated member variables, giving the following valid combinations in Table 4.3. For instance, the second entry, neighbor_descriptor<VId,true,E> has three members {source_id_type source_id; target_id_type target_id; vertex_type target;} and value_type is void.

Template Arguments	Members			
neighbor_descriptor<VId, true, E, EV>	source_id	target_id	target	value
neighbor_descriptor<VId, true, E, void>	source_id	target_id	target	
neighbor_descriptor<VId, true, void, EV>	source_id	target_id		value
neighbor_descriptor<VId, true, void, void>	source_id	target_id		
neighbor_descriptor<VId, false, E, EV>		target_id	target	value
neighbor_descriptor<VId, false, E, void>		target_id	target	
neighbor_descriptor<VId, false, void, EV>		target_id		value
neighbor_descriptor<VId, false, void, void>		target_id		

Table 4.3 — neighbor_descriptor Members

4.2 Copyable Descriptors

4.2.1 Copyable Descriptor Types

Copyable descriptors are specializations of the descriptors that can be copied. More specifically, they don't include a vertex or edge reference. copyable_vertex_t<G> shows the simple definition.

```
template <class VId, class VV>
```

```
using copyable_vertex_t = vertex_descriptor<VId, void, VV>; // id, value
```

Type	Definition
copyable_vertex_t<T,VId,VV>	vertex_descriptor<VId, void, VV>
copyable_edge_t<T,Vid,EV>	edge_descriptor<VId, true, void, EV>>
copyable_neighbor_t<Vid,VV>	neighbor_descriptor<VId, true, void, VV>

Table 4.4 — Descriptor Concepts

4.2.2 Copyable Descriptor Concepts

Given the copyable types, it's useful to have concepts to determine if a type is a desired copyable type.

Concept	Definition
copyable_vertex<T,VId,VV>	convertible_to<T, copyable_vertex_t<VId, VV>>
copyable_edge<T,Vid,EV>	convertible_to<T, copyable_edge_t<VId, EV>>
copyable_neighbor<T,Vid,VV>	convertible_to<T, copyable_neighbor_t<VId, VV>>

Table 4.5 — Descriptor Concepts

4.3 Common Types and Functions for "Search"

The Depth First, Breadth First, and Topological Sort searches share a number of common types and functions.

Here are the types and functions for cancelling a search, getting the current depth of the search, and active elements in the search (e.g. number of vertices in a stack or queue).

```
// enum used to define how to cancel a search
enum struct cancel_search : int8_t {
  continue_search, // no change (ignored)
  cancel_branch, // stops searching from current vertex
  cancel_all // stops searching and dfs will be at end()
};

// stop searching from current vertex
template<class S)
void cancel(S search, cancel_search);

// Returns distance from the seed vertex to the current vertex,
// or to the target vertex for edge views
template<class S>
auto depth(S search) -> integral;

// Returns number of pending vertices to process
template<class S>
auto size(S search) -> integral;
```

Of particular note, `size(dfs)` is typically the same as `depth(dfs)` and is simple to calculate. breadth_first_search requires extra bookkeeping to evaluate `depth(bfs)` and returns a different value than `size(bfs)` .

The following example shows how the functions could be used, using `dfs` for one of the depth_first_search views. The same functions can be used for all all search views.

```
auto&& g = ...; // graph
auto&& dfs = vertices_dfs(g,0); // start with vertex_id=0
for(auto&& [vid,v] : dfs) {
  // No need to search deeper?
  if(depth(dfs) > 3) {
    cancel(dfs,cancel_search::cancel_branch);
```

```
    continue;
  }

  if(size(dfs) > 1000) {
    std::cout << "Big depth of " << size(dfs) << '\n';
  }

  // do useful things
}
```

4.4 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_descriptor` on each iteration. Table 4.6 shows the vertexlist functions overloads and their return values. `first` and `last` are vertex iterators.

`vertexlist` views require a `vvf(u)` function, and the `basic_vertexlist` views require a `vvf(uid)` function.

Example	Return
`for(auto&& [uid,u] : vertexlist(g))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [uid,u,val] : vertexlist(g,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [uid,u] : vertexlist(g,first,last))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [uid,u,val] : vertexlist(g,first,last,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [uid,u] : vertexlist(g,vr))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [uid,u,val] : vertexlist(g,vr,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [uid] : basic_vertexlist(g))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [uid,val] : basic_vertexlist(g,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [uid] : basic_vertexlist(g,first,last))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [uid,val] : basic_vertexlist(g,first,last,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [uid] : basic_vertexlist(g,vr))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [uid,val] : basic_vertexlist(g,vr,vvf))`	`vertex_descriptor<VId,void,VV>`

Table 4.6 — `vertexlist` View Functions

4.5 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_descriptor` on each iteration. Table 4.7 shows the `incidence` function overloads and their return values.

Since the source vertex `u` is available when calling an `incidence` function, there's no need to include sourced versions of the function to include `source_id` in the output.

`incidence` views require a `evf(uv)` function, and `basic_incidence` views require a `evf(eid)` function.

Example	Return
`for(auto&& [vid,uv] : incidence(g,uid))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : incidence(g,uid,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [vid] : basic_incidence(g,uid))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_incidence(g,uid,evf))`	`edge_descriptor<VId,false,void,EV>`

Table 4.7 — `incidence` View Functions

4.6 neighbors Views

`neighbors` views iterate over a range of edges for a vertex, returning a `vertex_descriptor` of each neighboring target vertex on each iteration. Table 4.8 shows the `neighbors` function overloads and their return values.

Since the source vertex `u` is available when calling a `neighbors` function, there's no need to include sourced versions of the function to include `source_id` in the output.

`neighbors` views require a `vvf(u)` function, and the `basic_neighbors` views require a `vvf(uid)` function.

Example	Return
`for(auto&& [vid,v] : neighbors(g,uid))`	`neighbor_descriptor<VId,false,V,void>`
`for(auto&& [vid,v,val] : neighbors(g,uid,vvf))`	`neighbor_descriptor<VId,false,V,VV>`
`for(auto&& [vid] : basic_neighbors(g,uid))`	`neighbor_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_neighbors(g,uid,vvf))`	`neighbor_descriptor<VId,false,void,VV>`

Table 4.8 — `neighbors` View Functions

4.7 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_descriptor` on each iteration. Table 4.9 shows the `edgelist` function overloads and their return values.

`edgelist` views require a `evf(uv)` function, and `basic_edgelist` views require a `evf(eid)` function.

Example	Return
`for(auto&& [uid,vid,uv] : edgelist(g))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : edgelist(g,evf))`	`edge_descriptor<VId,true,E,EV>`
`for(auto&& [uid,uv] : basic_edgelist(g))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,uv,val] : basic_edgelist(g,evf))`	`edge_descriptor<VId,true,void,EV>`

Table 4.9 — `edgelist` View Functions

4.8 Depth First Search Views

Depth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 4.10 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`vertices_dfs` views require a `vvf(u)` function, and the `basic_vertices_dfs` views require a `vvf(uid)` function. `edges_dfs` views require a `evf(uv)` function. `basic_sourced_edges_dfs` views require a `evf(eid)` function. A `basic_edges_dfs` view with a evf is not available because `evf(eid)` requires that the `source_id` is available.

Example	Return
`for(auto&& [vid] : basic_vertices_dfs(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_dfs(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_dfs(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_dfs(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_dfs(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_dfs(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_dfs(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_dfs(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_dfs(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_dfs(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 4.10 — depth_first_search View Functions

4.9 Breadth First Search Views

Breadth First Search views iterate over the vertices and edges from a given seed vertex, returning a `vertex_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 4.11 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`vertices_bfs` views require a `vvf(u)` function, and the `basic_vertices_bfs` views require a `vvf(uid)` function. `edges_bfs` views require a `evf(uv)` function.

`basic_sourced_edges_bfs` views require a `evf(eid)` function. A `basic_edges_bfs` view with a evf is not available because `evf(eid)` requires that the `source_id` is available.

Example	Return
`for(auto&& [vid] : basic_vertices_bfs(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_bfs(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_bfs(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_bfs(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_bfs(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_bfs(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_bfs(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_bfs(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_bfs(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_bfs(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 4.11 — breadth_first_search View Functions

4.10 Topological Sort Views

Topological Sort views iterate over the vertices and edges from a given seed vertex, returning a `vertex_descriptor` or `edge_descriptor` on each iteration when it is first encountered, depending on the function used. Table 4.12 shows the functions and their return values.

While not shown in the examples, all functions have a final, optional allocator parameter that defaults to `std::allocator<bool>`. It is used for containers that are internal to the view. The `<bool>` argument has no particular meaning.

`vertices_topological_sort` views require a `vvf(u)` function, and the `basic_vertices_topological_sort` views require a `vvf(uid)` function. `edges_topological_sort` views require a `evf(uv)` function.

Example	Return
`for(auto&& [vid] : basic_vertices_topological_sort(g,seed))`	`vertex_descriptor<VId,void,void>`
`for(auto&& [vid,val] : basic_vertices_topological_sort(g,seed,vvf))`	`vertex_descriptor<VId,void,VV>`
`for(auto&& [vid,v] : vertices_topological_sort(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_topological_sort(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid] : basic_edges_topological_sort(g,seed))`	`edge_descriptor<VId,false,void,void>`
`for(auto&& [vid,val] : basic_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,false,void,EV>`
`for(auto&& [vid,uv] : edges_topological_sort(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid] : basic_sourced_edges_topological_sort(g,seed))`	`edge_descriptor<VId,true,void,void>`
`for(auto&& [uid,vid,val] : basic_sourced_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,true,void,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_topological_sort(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_topological_sort(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 4.12 — topological_sort View Functions

Chapter5 Graph Container Interface

The Graph Container Interface defines the primitive concepts, traits, types and functions used to define and access an adacency graph, no matter its internal design and organization. Thus, it is designed to reflect all forms of adjacency graphs including a vector of lists, CSR-based graph and adjacency matrix, whether they are in the standard or external to the standard.

All algorithms in this proposal require that vertices are stored in random access containers and that `vertex_id_t<G>` is integral, and it is assumed that all future algorithm proposals will also have the same requirements.

The Graph Container Interface is designed to support a wider scope of graph containers than required by the views and algorithms in this proposal. This enables for future growth of the graph data model (e.g. incoming edges on a vertex), or as a framework for graph implementations outside of the standard. For instance, existing implementations may have requirements that cause them to define features with looser constraints, such as sparse vertex_ids, non-integral vertex_ids, or storing vertices in associative bi-directional containers (e.g. std::map or std::unordered_map). Such features require specialized implementations for views and algorithms. The performance for such algorithms will be sub-optimal, but is preferrable to run them on the existing container rather than loading the graph into a high-performance graph container and then running the algorithm on it, where the loading time can far outweigh the time to run the sub-optimal algorithm. To achieve this, care has been taken to make sure that the use of concepts chosen is appropriate for algorithm, view and container.

5.1 Naming Conventions

Table 5.1 shows the naming conventions used throughout this document.

5.2 Concepts

Table 5.2 summarizes the concepts in the Graph Container Interface, allowing views and algorithms to verify a graph implementation has the expected requirements for an `adjacency_list` or `sourced_adjacency_list`.

Sourced edges have a source_id on them in addition to a target_id. A `sourced_adjacency_list` has sourced edges.

Indexed adjacency lists reflect a common use case where vertices are kept in a random access container and have an integral id.

5.3 Traits

Table 5.3 summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph's characteristics.

5.4 Types

Table 5.4 summarizes the type aliases in the Graph Container Interface. These are the types used to define the objects in a graph container, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, compressed_graph and adjacency matrix.

The type aliases are defined by either a function specialization for the underlying graph container, or a refinement of one of those types (e.g. an iterator of a range). Table 5.5 describes the functions in more detail.

`graph_value(g)`, `vertex_value(g,u)` and `edge_value(g,uv)` can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types.

There is no contiguous requirement for `vertex_id` from one partition to the next, though in practice they will often be assigned contiguously. Gaps in `vertex_id`s between partitions should be allowed.

5.5 Classes and Structs

The `graph_error` exception class is available, inherited from `runtime_error`. While any function may use it, it is only anticipated to be used by the `load` functions at this time. No additional functionality is added beyond that provided by `runtime_error`.

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

Table 5.1 — Naming Conventions for Types and Variables

5.6 Functions

Table 5.5 summarizes the functions in the Graph Container Interface. These are the primitive functions used to access an adacency graph, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, CSR-based graph and adjacency matrix.

Functions that have n/a for their Default Implementation must be defined by the author of a Graph Container implementation.

Value functions (graph_value(g) , vertex_value(g,u) and edge_value(g,uv)) can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types. They return a single value and can

Concept	Definition
vertex_range<G>	vertices(g) returns a sized, forward_range; vertex_id(g,ui) exists
targeted_edge<G>	target_id(g,uv) and target(g,uv) exist
adjacency_list<G>	Extends vertex_range<G> by adding edges(g,u) and edges(g,uid) that returns a forward_range
sourced_edge<G>	source_id(g,uv) and source(g,uv) exist
sourced_adjacency_list<G>	adjacency_list<G> and sourced_edge<G, edge_t<G>> and edge_id(g,uv) exists
index_vertex_range<G>	Extends vertex_range<G> by requiring vertices(g) return a random_access_range and vertex_id(g) return an integer
index_adjacency_list<G>	Extends adjacency_list<G> by requiring vertices(g) return a random_access_range and vertex_id(g) return an integer

Table 5.2 — Graph Container Interface Concepts

Trait	Type	Comment
has_degree<G>	concept	Is the degree(g,u) function available?
has_find_vertex<G>	concept	Are the find_vertex(g,_) functions available?
has_find_vertex_edge<G>	concept	Are the find_vertex_edge(g,_) functions available?
has_contains_edge<G>	concept	Is the contains_edge(g,uid,vid) function available?
define_unordered_edge<G,E> : false_type	struct	Specialize for edge implementation to derive from true_type for unordered edges
is_unordered_edge<G,E>	struct	conjunction<define_unordered_edge<E>, is_sourced_edge<G, E>>
is_unordered_edge_v<G,E>	type alias	
unordered_edge<G,E>	concept	
is_ordered_edge<G,E>	struct	negation<is_unordered_edge<G,E>>
is_ordered_edge_v<G,E>	type alias	
ordered_edge<G,E>	concept	
define_adjacency_matrix<G> : false_type	struct	Specialize for graph implementation to derive from true_type for edges stored as a square 2-dimensional array
is_adjacency_matrix<G>	struct	
is_adjacency_matrix_v<G>	type alias	
adjacency_matrix<G>	concept	

Table 5.3 — Graph Container Interface Type Traits

be scaler, struct, class, union, or tuple. These are abstract types used by the GVF, VVF and EVF function objects to retrieve values used by algorithms. As such it's valid to return the "enclosing" owning class (graph, vertex or edge), or some other embedded value in those objects.

vertex_id_t<G> is defined by the type returned by vertex_id(g) and it defaults to the difference_type of the underlying container used for vertices (e.g int64_t for 64-bit systems). This is sufficient for all situations. However, there are often space and performance advantages if a smaller type is used, such as int32_t or even int16_t. It is recommended to consider overriding this function for optimal results, assuring that it is also large enough for the number of possible vertices and edges in the application. It will also need to be overridden if the implementation doesn't expose the vertices as a range.

find_vertex(g,uid) is constant complexity because all algorithms in this proposal require that vertex_range_t<G> is a random access range.

If the concept requirements for the default implementation aren't met by the graph container the function will need to be overridden.

Edgelists are assumed to be either be an edgelist view of an adjacency graph, or a standard range with source_id and target_id

Type Alias	Definition	Comment
graph_reference_t<G>	add_lvalue_reference<G>	
graph_value_t<G>	decltype(graph_value(g))	optional
vertex_range_t<G>	decltype(vertices(g))	
vertex_iterator_t<G>	iterator_t<vertex_range_t<G>>	
vertex_t<G>	range_value_t<vertex_range_t<G>>	
vertex_reference_t<G>	range_reference_t<vertex_range_t<G>>	
vertex_id_t<G>	decltype(vertex_id(g))	
vertex_value_t<G>	decltype(vertex_value(g))	optional
vertex_edge_range_t<G>	decltype(edges(g,u))	
vertex_edge_iterator_t<G>	iterator_t<vertex_edge_range_t<G>>	
edge_t<G>	range_value_t<vertex_edge_range_t<G>>	
edge_reference_t<G>	range_reference_t<vertex_edge_range_t<G>>	
edge_value_t<G>	decltype(edge_value(g))	optional
The following is only available when the optional source_id(g,uv) is defined for the edge		
edge_id_t<G>	edge_descriptor<vertex_id_t<G>,true,void,void>	
partition_id_t<G>	decltype(partition_id(g,u))	optional
partition_vertex_range_t<G>	vertices(g,pid)	optional
partition_edge_range_t<G>	edges(g,u,pid)	optional

Table 5.4 — Graph Container Interface Type Aliases

values. There is no need for additional functions when a range is used.

5.7 Unipartite, Bipartite and Multipartite Graph Representation

partition_count(g) returns the number of partitions, or partiteness, of the graph. It has a range of 1 to n, where 1 identifies a unipartite graph, 2 is a bipartite graph, and a value of 2 or more can be considered a multipartite graph.

If a graph data structure doesn't support partitions then it is unipartite with one partition and partite functions will reflect that. For instance, partition_count(g) returns a value of 1, and vertices(g,0) (vertices in the first partition) will return a range that includes all vertices in the graph.

A partition identifies a type of a vertex, where the vertex value types are assumed to be uniform in each partition. This creates a dilemma because the existing vertex_value(g,u) returns a single type based template parameter for the vertex value type. Supporting multiple types can be addressed in different ways using C++ features. The key to remember is that the actual value used by algorithms is done by calling a function object that retrieves the value to be used. That function is specific to the graph data structure, using the partition to determine how to get the appropriate value.

— std::variant : The lambda returns the appropriate variant value based on the partition.

— Base class pointer: The lambda can call a member function to return the value based on the partition.

— void* : The lambda can cast the pointer to a concrete type based on the partition, and then return the appropriate value.

edges(g,uid,pid) and edges(g,ui,pid) filter the edges where the target is in the partition pid passed. This isn't needed for bipartite graphs.

5.8 Loading Graph Data

The load functions are used to load vertex and edge data into a graph. They may throw a graph_error exception.

All graph data structures need to implement load_graph , load_vertices and load_edges . Whether load_vertices or load_edges can be called multiple times, or after load_graph is called, is dependent on the underlying graph data structure. load_partition only needs to be implemented if a graph supports partitions.

Projections are used to convert values in the input range to the expected copyable type. In the following load_vertices prototype, vproj(ranges::range_value_t<VRng>&) → vertex_descriptor<vertex_id_t<G>, vertex_value_t<G>> . If there is no vertex value stored in the graph then vertex_value_t<G> will be void and the resulting vertex_descriptor

Function	Return Type	Complexity	Default Implementation
`graph_value(g)`	`graph_value_t<G>`	constant	n/a, optional
`partition_count(g)`	`vertex_id_t<G>`	constant	1
`vertices(g)`	`vertex_range_t<G>`	constant	`g` if `random_access_range<G>` , n/a otherwise
`num_vertices(g)`	`integral`	constant	`size(vertices(g))`
`find_vertex(g,uid)`	`vertex_iterator_t<G>`	constant	`begin(vertices(g))+ uid` if `random_access_range<vertex_range_t<G>>`
`vertex_id(g,ui)`	`vetex_id_t<G>`	constant	`ui - begin(vertices(g))` Override to define a different `vertex_id_t<G>` type (e.g. int32_t).
`vertex_value(g,u)`	`vertex_value_t<G>`	constant	n/a, optional
`vertex_value(g,uid)`	`vertex_value_t<G>`	constant	`vertex_value(g,*find_vertex(g,uid))` , optional
`degree(g,u)`	`integral`	constant	`size(edges(g,u))` if `sized_range<vertex_edge_range_t<G>>`
`degree(g,uid)`	`integral`	constant	`size(edges(g,uid))` if `sized_range<vertex_edge_range_t<G>>`
`partition_id(g,u)`	`partition_id_t<G>`	constant	0
`partition_id(g,uid)`	`partition_id_t<G>`	constant	`partition_id(g,*find_vertex(g,uid))`
`vertices(g,pid)`	`partition_vertex_range_t<G>`	constant	`vertices(g)`
`num_vertices(g,pid)`	`integral`	constant	`size(vertices(g))`
`edges(g,u)`	`vertex_edge_range_t<G>`	constant	`u` if `forward range<vertex_t<G>>` , n/a otherwise
`edges(g,uid)`	`vertex_edge_range_t<G>`	constant	`edges(g,*find_vertex(g,uid))`
`target_id(g,uv)`	`vertex_id_t<G>`	constant	n/a
`target(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ target_id(g,uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_value(g,uv)`	`edge_value_t<G>`	constant	`uv` if `forward_range<vertex_t<G>>` , n/a otherwise, optional
`find_vertex_edge(g,u,vid)`	`vertex_edge_t<G>`	linear	`find(edges(g,u), [](uv)target_id(g,uv)==vid;})`
`find_vertex_edge(g,uid,vid)`	`vertex_edge_t<G>`	linear	`find_vertex_edge(g,*find_vertex(g,uid),vid)`
`contains_edge(g,uid,vid)`	`bool`	constant	`uid < size(vertices(g))&& vid < size(vertices(g))` if `is_adjacency_matrix_v<G>` .
		linear	`find_vertex_edge(g,uid)!= end(edges(g,uid))` otherwise.
`edges(g,u,pid)`	`partition_edge_range_t<G>`	linear	`edges(g,u)`
`edges(g,uid,pid)`	`partition_edge_range_t<G>`	linear	`edges(g,uid)`
The following are only available when the optional `source_id(g,uv)` is defined for the edge			
`source_id(g,uv)`	`vertex_id_t<G>`	constant	n/a, optional
`source(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ source_id(g,uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_id(g,uv)`	`edge_id_t<G>`	constant	`edge_descriptor<vertex_id_t<G>,true, void,void>{source_id(g,uv),target_id(g,uv)}`

Table 5.5 — Graph Container Interface Functions

will have a single id member. If `vproj(ranges::range_value_t<VRng>&)` is the same as `vertex_descriptor<vertex_id_t<G>, vertex_value_t<G>>` then `VProj = identity` can be used.

```
template <adjacency_list G, ranges::forward_range VRng, class VProj = identity>
requires copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>,
                  vertex_id_t<G>, vertex_value_t<G>>
constexpr void load_vertices(G&, const VRng& vrng, VProj vproj);
```

The same pattern is applied using `ERng` and `EProj` for edges.

For graphs with vertex values, `load_vertices` should be called before `load_edges`.

Whether `load_vertices` or `load_edges` can be called multiple times is graph-dependent.

For graphs with partititions, `load_partition` must be called to load vertices for each partition `pid`. `pid` values must be contiguous and their vertices should be loaded contiguously. `empty(vrng)` may be empty if there are no vertices in the partition.

Function	Return Type	Complexity	Default Implementation
`load_graph(g, erng, vrng, eproj=identity(), vproj=identity())`	void	V + E	n/a
`load_vertices(g, vrng, vproj=identity())`	void	V	n/a
`load_partition(g, pid, vrng, vproj=identity())`	void	V(p)	`load_vertices` is called if partitions are not supported; there will be a single partition.
`load_edges(g, erng, eproj=identity(), vertex_count=0)`	void	E	n/a

Table 5.6 — Graph Load Functions

5.9 Using Existing Graph Data Structures

Reasonable defaults have been defined for the GCI functions to minimize the amount of work needed to adapt an existing graph data structure to be used by the views and algorithms.

There are two cases supported. The first is for the use of standard containers to define the graph and the other is for a broader set of more complicated implementations.

5.9.1 Using Standard Containers for the Graph Data Structure

For example this we'll use `G = vector<forward_list<tuple<int,double>>>` to define the graph, where `g` is an instance of `G`. `tuple<int,double>` defines the target_id and weight property respectively. We can write loops to go through the vertices, and edges within each vertex, as follows.

```
using G = vector<forward_list<tuple<int,double>>>;
auto target_id(const G& g, edge_t<const G>& uv) { return get<0>(uv); }
auto weight = [&g](edge_t& uv) { return get<1>(uv); }

G g;
load_graph(g, ...); // load some data

// Using GCI functions
for(auto&& [uid, u] : vertices(g)) {
  for(auto&& [vid, uv]: edges(g,u)) {
    auto w = weight(uv);
    // do something...
  }
}
```

Note that `target_id(g,uv)` was the only CPO function overriden; all other functions were automatically defined based on the rules shown in Table 5.7.

Function or Value	Concrete Type
`vertices(g)`	`vector<forward_list<tuple<int,double>>>` (when `random_access_range<G>`)
`u`	`forward_list<tuple<int,double>>`
`edges(g,u)`	`forward_list<tuple<int,double>>` (when `random_access_range<vertex_range_t<G>>`)
`uv`	`tuple<int,double>`
`edge_value(g,uv)`	`tuple<int,double>` (when `random_access_range<vertex_range_t<G>>`)

Table 5.7 — Types When Using Standard Containers

5.9.2 Using Other Graph Data Structures

For other graph data structures more function overrides are required. The following table identifies the common function overrides anticipated for most cases, keeping in mind that all functions in Table 5.5 can be overridden.

Function	Comment
`vertices(g)`	
`edges(g,u)`	
`target_id(g,uv)`	
`edge_value(g,uv)`	If edges have value(s) in the graph
`vertex_value(g,u)`	If vertices have value(s) in the graph
`graph_value(g)`	If the graph has value(s)
When edges have the optional source_id on an edge	
`source_id(g,uv)`	
When the graph supports multiple partitions	
`partition_count(g)`	
`partition_id(g,u)`	
`vertices(g,u,pid)`	

Table 5.8 — Common CPO Function Overrides

Chapter6 Graph Container Implementation

6.1 compressed_graph

The compressed_graph is a high-performance graph container that uses Compressed Sparse Row format to store its vertices, edges and associated values. Once constructed, vertices and edges cannot be added or deleted but values on vertices and edges can be modified.

The following listing shows the prototype for the `compressed_graph` . Only the members shown for `compressed_graph` are public. No other member functions or types are exposed as part of the standard. All other types are only accessible through the types and functions in the Graph Container Interface. Multiple partitions (multi-partite) can be defined by passing the number of partitions in a constructor.

When a value type template argument (EV, VV, GV) is void then no extra overhead is incurred for it. The selection of the VId template argument impacts the inter storage requirements. If you have a small graph where the number of vertices is less than 256, and the number of edges is less than 256, then a `uint8_t` would be sufficient.

Implements `load_graph` ? Yes **Implements `load_vertices` ?** Yes **Implements `load_edges` ?** Yes **Implements `load_partition` ?** Yes	**Can append vertices?** No **Can append vertices?** No	**vertex_id assignment:** Contiguous **Vertices range:** Contiguous **Edge range:** Contiguous

Vertices and edges cannot be appended to an existing partition, but they can be added to a new partition.

```
template <class EV = void, // Edge Value type
        class VV = void, // Vertex Value type
        class GV = void, // Graph Value type
        integral VId = uint32_t, // vertex id type
        integral EIndex = uint32_t, // edge index type
        class Alloc = allocator<uint32_t>> // for internal containers
class compressed_graph {
public:
   compressed_graph();
   compressed_graph(size_t num_partitions); // multi-partite
   compressed_graph(const compressed_graph&);
   compressed_graph(compressed_graph&&);
   {tilde}compressed_graph();

   compressed_graph& operator=(const compressed_graph&);
   compressed_graph& operator=(compressed_graph&&);
}
```

Chapter7 Graph Adaptors

7.1 Edge List Adaptor

Acknowledgements

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

Bibliography

[1] Dominiak, Evtushenko, Baker, Teodorescu, Howes, K. Shoop, M. Garland, E. Niebler, and B. Lelbach, "P2300r5 std::execution." "https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r5.html".

[2] A. Lumsdaine, L. D'Alessandro, K. Deweese, J. Firoz, T. Liu, S. McMillan, P. Ratzloff, and M. Zalewski, "Nwgraph: A library of generic graph algorithms and data structures in c++20." "https://drops.dagstuhl.de/opus/volltexte/2022/16259/".

[3] A. Azad, M. M. Aznaveh, S. Beamer, M. P. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang, and Y. Zhang, "Evaluation of graph analytics frameworks using the gap benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 216–227, 2020.

[4] A. Lumsdaine, L. D'Alessandro, K. Deweese, J. Firoz, T. Liu, S. McMillan, P. Ratzloff, and M. Zalewski, "Nwgraph library code." "https://github.com/pnnl/NWGraph".

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 4 ed., 2022.

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