Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 D1709R4
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

P1709R4

This was a major redesign that incorporated all the experience and input from the past 3 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.

[PHIL: Need "basic" bfs so we don't have to include uv]
[PHIL: Duplicated in Introduction]

```
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,uv] : sourced_edges_breadth_first_search(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.

[PHIL: Duplicated in Overview's Examples]

```
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,uv] : sourced_edges_breadth_first_search(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.

2.3 Graph Background and Terminology

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].

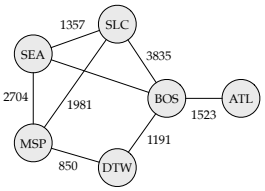
2.3.1 Theoretical Terminology

A *graph* G is a set comprised of two other sets, the vertex set V and the edge set E, typically expressed as $G = \{V, E\}$. We can number the members of the vertex set and write $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Similarly, we can write $E = \{e_0, e_1, \ldots, e_{m-1}\}$. The number of elements in V is denoted by $|V|$ and the number of elements in E is denoted by $|E|$. Edges in E are pairs of vertices; for any $e_k \in E$, we write $e_k = (v_i, v_j)$, where $v_i \in V$ and $v_j \in V$. The edges in E may be *unordered*, in which case $(v_i, v_j) = (v_j, v_i)$ or *ordered*, in which case $(v_i, v_j) \neq (v_j, v_i)$ (unless $i = j$). A graph consisting of unordered edges is said to be *undirected*; a graph consisting of ordered edges is said to be *directed*. Since there is a single set of vertices in this definition of G, that is, edges connect vertices in V to vertices in V, we say that G is a *unipartite* graph. If the graph further has no loops (edges connecting a vertex to itself) and no multiple edges (edges connecting the same pair of vertices), then G is called a *simple* graph. A *complete graph* is a simple graph G where every vertex is connected to every other vertex; a complete graph has n vertices and $n(n-1)/2$ edges. A *path* is 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 all $i = 0, 1, \ldots, k-2$.

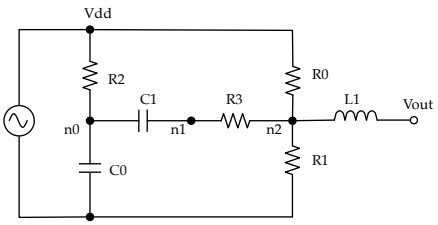
Graphs are often deipicted as a collection of nodes and links, as shown in Figure 2.1. In this model, vertices are represented graphically as nodes, which are shown as circles (or some other shape, including points). Edges are represented as links, shown as lines connecting the nodes.

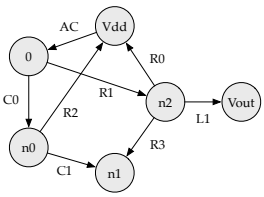
Airport
SEA
MSP
SLC
DTW
ATL
BOS

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



(a) An undirected graph representing airline routes between cities.





(b) A directed graph representing an electronic circuit.

Figure 2.1 — Node and link graph models, an intuitive representation of relationships between entities.

Associated with every $v_i \in V$ is the integer i. We call i the *index* (or the *vertex identifier*) of v_i. The index of a vertex is a unique identifier for that vertex. The index of a vertex is not necessarily the same as the position of the vertex in the vertex set V. For example, in the graph $G = \{\{0, 1, 2, 3\}, \{(0, 1), (1, 2), (2, 3)\}\}$, the vertex v_0 has index 0, v_1 has index 1, v_2 has index 2, and v_3 has index 3. In the literature, a vertex and its index are often used interchangeably.

[ANDREW: A separate proposal for this? However, the Kevin Bacon example requires bipartite graphs. I also have questions about bipartiteness (or labeling in general) at every talk I give about C++ graph libraries.]

[PHIL: I don't think we need a separate proposal. I've added the types and functions I think we need for multipartite graphs and the compressed_graph can be extended to support it.]

A *bipartite graph* is a graph G with two disjoint sets of vertices U and V such that every edge in E connects a vertex in U to a vertex in V. A bipartite graph is useful for modeling connections between two different types of objects. For example, a bipartite graph can be used to model connections between actors and movies that they have appeared in. To obtain an actor-actor co-star graph from an actor-movie graph, we perform a *join* operation on the graph and its *transpose*, which is the graph connecting movies to actors, that is the graph with the same vertices as the original graph but with edge directions reversed, i.e., a graph connecting movies to actors (see Figure 2.2).

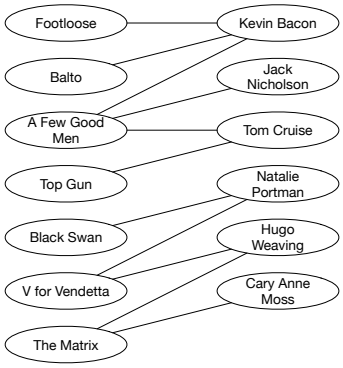


Figure 2.2 — A movie-actor graph. The links indicate that an actor has appeared in a movie.

2.3.2 Adjacency List

An *adjacency list* of G is a data structure that represents a graph G as an array indexed by the vertex indices of the vertices of G. The adjacency list of G is denoted as $Adj(G)$. For each vertex $v_i \in G$, the array entry $Adj(G)[i]$ is a list of vertex indices of the vertices adjacent to v_i. The adjacency list is the most common representation of a graph in the graph literature and is used in the majority of graph algorithms as it has the important property for traversal that (in the case of a unipartite graph) an index j stored neighbor list $Adj(G)[i]$ can be used to index back into $Adj(G)$. In this formulation, the adjacency list does not store vertices per se (rather, it stores vertex indices), it captures the structure of the graph. This structure is the essence of the graph abstraction and is the essential information needed for graph algorithms. Even so, it is still possible to store vertex data in the adjacency list by storing the vertex data in an auxiliary array indexed by vertex indices.

2.3.3 Edge List

An *edge list* of G is a data structure that contains the edges of G. More precisely, as with the adjacency list structure, the edge list stores pairs of vertex indices. That is, for every $(v_i, v_j) \in E$, the edge list contains the pair (i, j). The edge list is a less common representation of a graph than the adjacency list, but it is useful for some graph algorithms. Moreover, it is a natural intermediate representation for converting between data that has not yet been interpreted as a graph, e.g., a table, and an adjacency list. Moreover, certain tasks, such as sorting or relabeling vertices, are more natural to express in terms of the edge list.

2.4 From Data to Graph

2.4.1 Columnar Data

Figure 2.3 shows how one might create an unlabeled edge list from a table of data stored in a CSV file. The values in each row are assumed to be separated by whitespace. The elements of the first column are considered to be the source vertices and the elements of the second column are the destination vertices. For edges with properties, the third column contains the property values.

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

(a) Creating a directed edge list with no edge properties. The edge list container is a vector of tuples.

```
auto undirected_edge_list_graph<
    vertex_id_t, vertex_id_t, double>
  graph<std::vector<vertex_id_t>, std::
      vector<vertex_id_t>, std::vector<
      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);
}
```

(b) Creating an undirected edge list with edge properties. The edge list container is comprised of three vectors, wrapped by the 'lstinlinegraph' adapter.

Figure 2.3 — Creating an edge list from columnar data.

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.

2.4.2 Converting an Edge List to an Adjacency List

[ANDREW: We should have note that anything wrapped in a graph adapter has a 'num_vertices' method (and other members). We should also provide a constructor for the adapted adjacency list that takes and edge list as argument.]

2.5 BiPartite Graphs

[PHIL: Need to reword to accomodate multipartite graphs] So far, we have been considering graphs where the vertices, and both elements of an edge, are members of a single set $|V|$. A graph with a single vertex set is called a *unipartite* graph. If the vertices in a graph can be partitioned into two disjoint sets such that all of the edges in the graph only connect vertices from one set of the vertices of the other set, the graph is called a *bipartite* graph. [ANDREW: Set membership == vertex label. How many labels can a vertex have? How many sets can a vertex belong to?]

Even if a graph is designated to be unipartite, it may be the case that its vertices can be partitioned into two disjont sets. In such a case, the graph is bipartite, but as a run-time property, not a compile-time property. That is, determining whether a given graph is bipartite requires a run-time analysis of the graph, with an appropriate algorithm.

However, there are numerous cases of interest where the vertices of a graph fall naturally into two disjoint sets. For example, in the in a social network, the vertices represent people and the edges represent friendships. In such a case, it is natural Of

```
auto edge_list<vertex_id_t, vertex_id_t>
     std::vector<std::tuple<vertex_id_t,
     vertex_id_t> edges;
auto adjacency_list<vertex_id_t> std::
    vector<std::vector<vertex_d_t>>
    adj_list;
for (auto [src, dst] : edges) {
  if (src >= adj_list.size()) {
    adj_list.resize(src + 1);
  }
  adj_list[src].push_back (dst);
}
```

(a) Creating an adjacency list from a directed edge list. The adjacency list container is a vector of vectors.

```
auto edge_list_graph<vertex_id_t,
    vertex_id_t>
  graph<std::vector<std::tuple<
      vertex_id_t, vertex_id_t>> edges;
auto adjaceny_list_graph<vertex_id_t,
    vertex_id_t, double>
  graph<std::vector<std::vector<
      vertex_d_t>, std::vector<double>>
      adj_list{edges.num_vertices()};
for (auto [src, dst, val] : edges) {
  adj_list[src].push_back (dst, val);
  adj_list[dst].push_back (src, val);
}
```

(b) Creating an adjacency list from an undirected edge list with properties. Since the edges in the edge list are undirected, for a given edge (u, v, w), u is a neighbor of v and v is a neighbor of u. Thus, we insert both (u, v, w) and (v, u, w) into the adjacency list. The adjacency list container is comprised of three vectors, wrapped by the 'lstinlinegraph' adapter.

Figure 2.4 — Creating an adjacency list from an edge list with properties.

particular interest for realizing a bipartite graph is when the graph is structurally bipartite, that is, when we are explicitly given two different sets of vertices and the corresponding set of edges that connect vertices of the two sets. This common—and important—use case arises when modeling relationships between different types of entities. For example, we might use a structurally bipartite graph in which one vertex set represents customers and another vertex set represents products. An edge between a customer and a product would be used to indicate that a customer has purchased a particular product. Another such example is from the Kevin Bacon game above, where one set represents actors and the other set represents movies; edges represent whether and actor appeared in a movie. Thus edges only connect vertices from the set of actors to the set of movies.

We can refer to graphs of this form as *structurally bipartite* graphs and denote them as $G = (U, V, E)$, where the vertex sets U and V are disjoint.

2.5.1 BiPartite Graph Models

2.5.2 BiPartite Graph Implementations

2.6 Naming Conventions

Table 2.1 shows the naming conventions used throughout this document.

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

Table 2.1 — Naming Conventions for Types and Variables

Chapter3 Algorithms

Our proposed set of algorithms are grouped into Tier 1, Tier 2, and Tier 3 (similar to the approach used to prioritize algorithms in `std::ranges`).

Tier 1:

— Breadth-First search
— Dijkstra's algorithm
— Bellman-Ford
— Triangle counting
— Label propagation
— Articulation points
— Connected components
— Biconnected components
— Strongly connected components
— Topological sort
— Maximal independent set
— Jaccard coefficient
— Kruskal Minimal Spanning Tree
— Prim Minimal Spanning Tree

3.1 Introduction

The following table shows an example of the summary characteristics of an algorithm.

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.

[PHIL: The Directed? section needs work.]

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

template <class G, class WF, class DistVal>
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,
                           Distance,
                           less<DistVal>,
                           plus<DistVal>>;
```

3.3 Shortest Paths

[ANDREW: Note that NetworkX also specifies single source single target and multiple source versions of the shortest paths algorithms. BGL does not have these (nor NWGraph). We should discuss whether or not to consider those and whether or not to make them Tier 1, 2, 3, or infinity.]

3.3.1 Unweighted Shortest Paths: Breadth-First Search

3.3.1.1 Breadth-First Search, Single Source, Initialization

```
template <property P, property D>
void init_breadth_first_search(P&& predecessors, D&& distances);

template <property P, property D>
void init_breadth_first_search(P&& predecessors,
                       D&& distances,
                       std::ranges::range_value_t<D> init);

template <property D>
void init_breadth_first_distances(D&& distances);

template <property D>
void init_breadth_first_distances(D&& distances, std::ranges::range_value_t<D> init);
```

Effects:

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

— Each `distance[i]` is initialized to `std::numeric_limits::max()` or to `init`.

3.3.1.2 Breadth-First Search, Single Source

[PHIL: What value does this bring over the `breadth_first_search` view?]

```
template <adjacency_list_graph G, property D, property P>
void breadth_first_search(const G& graph,
                  vertex_id_t<G> source,
                  P&& predecessors,
                  D&& distances);

template <adjacency_list_graph G, property D, property P, queueable Q>
void breadth_first_search(
    const G& graph, vertex_id_t<G> source, P&& predecessors, D&& distances, Q&& q);
```

```
template <adjacency_list_graph G, property D>
void breadth_first_distances(const G& graph, vertex_id_t<G> source, D&& distances);

template <adjacency_list_graph G, property D, queueable Q>
void breadth_first_distances(const G& graph, vertex_id_t<G> source, D&& distances, Q&& q);
```

1 *Mandates:*

(1.1) — `graph` is an `adjacency_list`, which may be directed or undirected.

2 *Preconditions:*

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

(2.2) — `distance[i] == std::numeric_limits<range_value_t<D>>::max()` for `0 <= i < num_vertices(graph)`. [ANDREW: invalid distance?]

(2.3) — The `predecessors` range must be initialized so that `precessors[i] == i` for all `i` such that `0 <= i < num_vertices(graph)`.

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

4 *Result:*

(4.1) — 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 `std::numeric_limits<range_value_t<D>>::max()`.

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

5 *Returns:* `void`

6 *Throws:* none. [ANDREW: Throw if `source` out of range?]

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

8 *Remarks:*

3.3.2 Weighted Shortest Paths

3.3.2.1 Dijkstra Initialization

[ANDREW: Actually, I don't think dijkstra_invalid_distance() or dijkstra_zero() useful – those need to be specified for the actual init functions. As free functions they don't really do anything. Or are they meant to be CPOs?]

[ANDREW: Is there a run-time overhead introduced by using functions and/or CPOs for things like dijkstra invalid or dijkstra zero?]

[ANDREW: I would like to see a use case for the dijkstra invalid and dijkstra zero functions.]

```
template <class DistanceValue>
auto dijkstra_invalid_distance() {
  return std::numeric_limits<DistanceValue>::max();
}

template <class DistanceValue>
auto dijkstra_zero() {
  return {};
}

template <property P, property D>
void init_dijkstra_shortest_paths(P&& predecessors, D&& distances);

template <property P, property D>
void init_dijkstra_shortest_paths(P&& predecessors,
                          D&& distances,
                          std::ranges::range_value_t<D> init);

template <property D>
```

```
void init_dijkstra_shortest_distances(D&& distances);

template <property D>
void init_dijkstra_shortest_distances(D&& distances, std::ranges::range_value_t<D> init);
```

1 *Effects:* :

(1.1) — Each `predessors[i]` is initialized to `i`.

(1.2) — Each `distance[i]` is initialized to `std::numeric_limits::max()` or to `init`.

3.3.2.2 Dijkstra Single Source Shortest Paths

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.

```
// Concepts and types from std::ranges don't include the namespace prefix for brevity
// and clarity of purpose

// A fake range with no values and does nothing
inline static null_range_type null_predecessors;

template <index_adjacency_list G,
        random_access_range Distance,
        random_access_range Predecessor
        class WF = function<range_value_t<Distance>(edge_reference_t<G>)>>
requires convertible_to<vertex_id_t<G>, range_value_t<Predecessor>> &&
       edge_weight_function<G, WF, range_value_t<Distance>>
void dijkstra_shortest_paths(
     const G& graph,
     vertex_id_t<G> source,
     Distance& distances,
     Predecessor& predecessors = null_predecessors,
     WF&& w = [](edge_reference_t<G> uv) { return range_value_t<Distance>(1); });

template <index_adjacency_list G,
        random_access_range Distance,
        random_access_range Predecessor,
        class Compare,
        class Combine,
        class WF = function<range_value_t<Distance>(edge_reference_t<G>)>>
requires convertible_to<vertex_id_t<G>, range_value_t<Predecessor>> &&
       basic_edge_weight_function<G, WF, range_value_t<Distance>, Compare, Combine>
void dijkstra_shortest_paths(
     const G& graph,
     vertex_id_t<G> source,
     Compare&& compare,
     Combine&& combare
     Distance& distances,
     Predecessor& predecessors = null_predecessors,
     WF&& w = [](edge_reference_t<G> uv) { return range_value_t<Distance>(1); });
```

1 *Mandates:*

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

2 *Preconditions:*

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

(2.2) — `distance[i] = numeric_limits<range_value_t<Distance>>::max()` for `0 <= i < num_vertices(graph)`.
[ANDREW: invalid distance?]

(2.3) — `precessors[i] = i` for `0 <= i < num_vertices(graph)`.

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 `numeric_limits<range_value_t<Distance>>::max()`.

(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:*

(5.1) — Passing `null_predecessors` in the `predecessors` parameter will result in a compile-time decision to not evaluate predecessors.

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

3.3.2.3 Bellman-Ford Single Source Shortest Paths

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

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

```
// Concepts and types from std::ranges don't include the namespace prefix for brevity
// and clarity of purpose

template <index_adjacency_list G,
          random_access_range Distance,
          random_access_range Predecessor
          class WF = function<range_value_t<Distance>(edge_reference_t<G>)>>
requires convertible_to<vertex_id_t<G>, range_value_t<Predecessor>> &&
         edge_weight_function<G, WF, range_value_t<Distance>>
void bellman_ford_shortest_paths(
     const G& graph,
     vertex_id_t<G> source,
     Distance& distances,
     Predecessor& predecessors = null_predecessors,
     WF&& w = [](edge_reference_t<G> uv) { return range_value_t<Distance>(1); });

template <index_adjacency_list G,
          random_access_range Distance,
          random_access_range Predecessor,
          class Compare,
          class Combine,
          class WF = function<range_value_t<Distance>(edge_reference_t<G>)>>
requires convertible_to<vertex_id_t<G>, range_value_t<Predecessor>> &&
         basic_edge_weight_function<G, WF, range_value_t<Distance>, Compare, Combine>
void bellman_ford_shortest_paths(
     const G& graph,
     vertex_id_t<G> source,
     Compare&& compare,
     Combine&& combare,
     Distance& distances,
     Predecessor& predecessors = null_predecessors,
     WF&& w = [](edge_reference_t<G> uv) { return range_value_t<Distance>(1); });
```

1 *Preconditions:*

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

(1.2) — `distance[i] = numeric_limits<range_value_t<Distance>>::max()` for `0 <= i < num_vertices(g)`.
[ANDREW: invalid distance?]

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

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 `numeric_limits<range_value_t<Distance>>::max()`.

(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) — Passing `null_predecessors` in the `predecessors` parameter will result in a compile-time decision to not evaluate predecessors.

(4.2) — 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

[PHIL: Description needed]

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

```
template <adjacency_list G>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
size_t triangle_count(G&& g) {}
```

1 *Returns:* Number of triangles

3.5 Communities

3.5.1 Label Propagation

[PHIL: Description needed]

Complexity	**Throws?** No	**Cycles?** No
$\mathcal{O}(N)$ [KEVIN: ???]	**Multi-edge?** No	**Directed?** Yes

```
template <adjacency_list G,
          ranges::random_access_range Label,
          class Gen = std::default_random_engine,
          class T = size_t>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
void label_propagation(G&& g,
                       Label& label,
                       Gen&& rng = std::default_random_engine {},
                       T max_iters = std::numeric_limits<T>::max()) {}
```

1 *Preconditions:*

(1.1) — `label` contains vertex labels.

(1.2) — `rng` is a random number generator.

(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.

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

```
template <adjacency_list G,
          ranges::random_access_range Label,
          class Gen = std::default_random_engine
          class T = size_t>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
void label_propagation(G&& g,
                       Label& label,
                       std::ranges::range_value_t<Label>& empty_label
                       Gen&& rng = std::default_random_engine {},
                       T max_iters = std::numeric_limits<T>::max()) {}
```

3 *Preconditions:*

(3.1) — `label` contains vertex labels.

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

(3.3) — `rng` is a random number generator.

(3.4) — `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.

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

3.6 Components

3.6.1 Articulation Points

[PHIL: Description needed]

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

```
template <adjacency_list G,
          class Iter>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>> &&
         std::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

[PHIL: Description needed]

```
template <adjacency_list G,
          std::ranges::forward_range OuterContainer>
```

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

```
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>> &&
       std::ranges::forward_range<std::ranges::range_value_t<OuterContainer>> &&
       integral<std::ranges::forward_range_t<std::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

[PHIL: Description needed]

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

```
template <adjacency_list G,
          ranges::random_access_range Component>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
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

[PHIL: Description needed]

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

```
template <adjacency_list G,
          adjacency_list GT,
          ranges::random_access_range Component>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>> &&
       ranges::random_access_range<vertex_range_t<GT>> && integral<vertex_id_t<GT>>
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

[PHIL: Description needed]

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

[PHIL: Description needed]
[PHIL: Do we need this as an algorithm, in addition to the view? What value does it bring?]

3.8 Maximal Independent Set

3.8.1 Maximal Independent Set

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

```
template <adjacency_list G, class Iter>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>> &&
     std::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

[PHIL: Description needed]

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

```
template <adjacency_list G, typename OutOp, typename T = double>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>> &&
    is_invocable_v<OutOp, vertex_id_t<G>&, vertex_id_t<G>&, edge_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

[PHIL: Description needed]

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

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

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

```
template <adjacency_list G,
          ranges::random_access_range Predecessor,
          ranges::random_access_range Weight>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
void prim(G&& g, Predecessor& predecessor, Weight& weight, vertex_id_t<G> seed = 0) {}

template <adjacency_list G,
          ranges::random_access_range Predecessor,
          ranges::random_access_range Weight,
          class CompareOp>
requires ranges::random_access_range<vertex_range_t<G>> && integral<vertex_id_t<G>>
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

[ANDREW: Need to be able to sort edge lists along source or target column as well as to sort lists of neighbors in an adjacency list. The former is necessary for efficiently building CSR from an edge list. The second is necessary for preconditions on various algorithms.]

3.11.5 Transpose

[ANDREW: I've used NetworkX as inspiration for organization. Oddly, NetworkX only has DFS as an adaptor (view).]

Chapter4 Other Algorithms

Additional algorithms that were considered but not included in this proposal are identified in Table 4.1. 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.

Tier 2	**Tier 3**	**Tier X**
All Pairs Shortest Paths	Jones Plassman	Single Source, Single Target: Shortest Paths
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
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
		Multiple Source, Single Target: BFS
		Multiple Source, Single Target: Dijkstra
		Multiple Source, Single Target: Bellman-Ford
		Multiple Source, Single Target: Delta Stepping

Table 4.1 — Other Algorithms

[ANDREW: All Pairs: Tier 2? People bring this up alot – but it is very expensive in terms of computation and memory.]

Chapter5 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.

5.1 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_view<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_view<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] : basic_vertexlist(g,vvf)) {
  // ...  do something interesting
}
```

5.1.0.1 `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_depth_first_search(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 5.1 . 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 5.1 — `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
```

5.1.0.2 `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_search(g,u)`, `edges_depth_first_search(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 5.2 . 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 5.2 — edge_descriptor Members

5.1.0.3 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 5.3. 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 5.3 — neighbor_descriptor Members

5.2 Copyable Descriptors

5.2.1 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 5.4 — Descriptor Concepts

5.2.2 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 5.5 — Descriptor Concepts

5.3 Common Types and Functions for "Search"

The depth_first_search, breadth_first_search, and toplogical_sort searches there are a number of common types and functions that apply to them.

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_depth_first_search(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
}
```

5.4 vertexlist Views

`vertexlist` views iterate over a range of vertices, returning a `vertex_descriptor` on each iteration. Table 5.6 shows the vertexlist functions overloads and their return values. `first` and `last` are vertex iterators.

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 5.6 — `vertexlist` View Functions

5.5 incidence Views

`incidence` views iterate over a range of adjacent edges of a vertex, returning a `edge_descriptor` on each iteration. Table 5.7 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.

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 5.7 — `incidence` View Functions

5.6 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 5.8 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.

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 5.8 — `neighbors` View Functions

5.7 edgelist Views

`edgelist` views iterate over all edges for all vertices, returning a `edge_descriptor` on each iteration. Table 5.9 shows the `edgelist` function overloads and their return values.

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 5.9 — `edgelist` View Functions

5.8 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 5.10 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.

[PHIL: Need ”basic” variants that don’t include edge reference]

Example	Return
`for(auto&& [vid,v] : vertices_depth_first_search(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_depth_first_search(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid,uv] : edges_depth_first_search(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_depth_first_search(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_depth_first_search(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_depth_first_search(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 5.10 — depth_first_search View Functions

5.9 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 5.11 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.

[PHIL: Need ”basic” variants that don’t include edge reference]

5.10 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 5.12 shows the functions and their return values.

Example	Return
`for(auto&& [vid,v] : vertices_breadth_first_search(g,seed))`	`vertex_descriptor<VId,V,void>`
`for(auto&& [vid,v,val] : vertices_breadth_first_search(g,seed,vvf))`	`vertex_descriptor<VId,V,VV>`
`for(auto&& [vid,uv] : edges_breadth_first_search(g,seed))`	`edge_descriptor<VId,false,E,void>`
`for(auto&& [vid,uv,val] : edges_breadth_first_search(g,seed,evf))`	`edge_descriptor<VId,false,E,EV>`
`for(auto&& [uid,vid,uv] : sourced_edges_breadth_first_search(g,seed))`	`edge_descriptor<VId,true,E,void>`
`for(auto&& [uid,vid,uv,val] : sourced_edges_breadth_first_search(g,seed,evf))`	`edge_descriptor<VId,true,E,EV>`

Table 5.11 — breadth_first_search View Functions

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.

[PHIL: Need "basic" variants that don't include edge reference]

Example	Return
`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,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,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 5.12 — topological_sort View Functions

[PHIL: Is Topological Sort a view, an algorithm or both?]

Chapter6 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.

6.1 Concepts

Table **??** 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.

[PHIL: Need to resolve how to deal with ”basic” concepts which don’t require vertex or edge references. Can a GGI exist without the references? Is it just a View artifact?]

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 6.1 — Descriptor Concepts

6.2 Traits

Table 6.2 summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph’s characteristics.

6.3 Types

Table 6.3 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.

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 6.2 — Graph Container Interface Type Traits

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 6.4 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.

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 6.3 — Graph Container Interface Type Aliases

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.

6.4 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` .

6.5 Functions

Table 6.4 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 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 values. There is no need for additional functions when a range is used.

6.6 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.

6.7 Loading Graph Data

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

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 6.4 — Graph Container Interface Functions

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 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 6.5 — Graph Load Functions

6.8 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.

6.8.1 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&& u : vertices(g)) {
  for(auto&& 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 6.6.

6.8.2 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 6.4 can be overridden.

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 6.6 — Types When Using Standard Containers

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 6.7 — Common CPO Function Overrides

Chapter7 Graph Container Implementation

7.1 compressed_graph

The compressed_graph is a high-performance graph container that uses Compressed Sparse Row format to store it's 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.

compressed_graph supports multipartite graphs.

All the `load` functions are supported, including `load_graph`, `load_vertices`, `load_partition` and `load_edges` .

```
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&&);
}
```

[PHIL: Additional members to define?]

[PHIL: Add table capabilities/limitations]

[PHIL: tilde in destructor is giving problems in latex]

Chapter8 Graph Adaptors

8.1 Edge List Adaptor

[ANDREW: Need an adaptor to convert an edgelist to an adjacency_list.]

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.