Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 D9901r0 — Graph Library Overview and Introduction
Date: 2024-02-11
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: SG19, SG14, SG6, LEWG, LWG
Source: github.com/stdgraph/graph-v2
Prev. Version: www.wg21.link/P1709r5

Contents

Chapter1 Getting Started

This paper is one of several interrelated proposals related to a Graph Library proposal that have been broken out for easier consumption. The following table describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now broken into the following papers.
P9901	Active	**Graph Library Overview and Introduction**, describing the big picture of what we are proposing, and theortical background of graphs in general.
P9902	Active	**Graph Library Algorithms**, covering the initial algorithms as well as the ones we'd like to see in the future.
P9903	Active	**Graph Library Operators** includes useful utility functions when working with graphs.
P9904	Active	**Graph Library Views** including helpful views for traversing a graph.
P9905	Active	**Graph Library Container Inferface** is the core interface used for accessing the underlying graph data structure.
P9906	Active	**Graph Library Container** describing the high-performance `compressed_graph` container, based on a Compressed Sparse Row sparse matrix layout.
P9907	Active	**Graph Library Adaptors** containing useful utilities to convert graphs to different forms.

Table 1.1 — Graph Library Papers

1.1 Revision History

D9901r0

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

Chapter2 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.

2.1 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 in the Graph Container Interface and Views, 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.

2.2 Examples

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

[PHIL: Duplicated in Introduction. OK?]

[PHIL: Validate that code works]

```
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>>;
G costar_adjacency_list{
   {1, 5, 6}, {7, 10, 0, 5, 12}, {4, 3, 11}, {2, 11}, {8, 9, 2, 12}, {0, 1}, {7, 0},
   {6, 1, 10}, {4, 9}, {4, 8}, {7, 1}, {2, 3}, {1, 4} };

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

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

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

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

2.3 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.

2.4 Impact on the Standard

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

2.5 Interaction wtih Other Papers

There is no interaction with other proposals to the standard.

2.6 Implementation Experience

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

2.7 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.

2.8 Deployment Experience

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

2.9 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].

2.10 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.

2.11 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.

2.12 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).

2.13 Freestanding

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

2.14 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`

Chapter3 Introduction

3.1 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.

3.2 Example: Six Degrees of Kevin Bacon

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

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

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

using G = std::vector<std::vector<int>>;
G costar_adjacency_list{
   {1, 5, 6}, {7, 10, 0, 5, 12}, {4, 3, 11}, {2, 11}, {8, 9, 2, 12}, {0, 1}, {7, 0},
   {6, 1, 10}, {4, 9}, {4, 8}, {7, 1}, {2, 3}, {1, 4} };

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

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

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

Output:

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

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

3.3 Graph Background

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

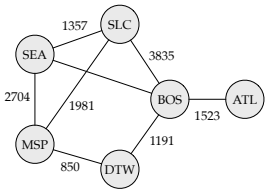
3.3.1 Basic Terminology

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

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

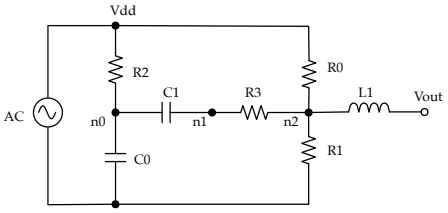
Airport
SEA
MSP
SLC
DTW
ATL
BOS

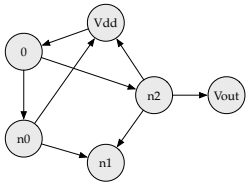
Distance (km)		
MSP	DTW	850
SLC	SEA	1357
MSP	SLC	1981
BOS	SLC	3835
SEA	BOS	4016
BOS	ATL	1523
SEA	MSP	2704
BOS	DTW	1191



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

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





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

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

Figure 3.1 — Graph models of an airline route system and of an electronic circuit.

3.3.2 Graph Representation: Enumerating the Vertices

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

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

We summarize some remaining terminology about vertices and edges.

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

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

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

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

[1] An unfortunate collision of terminology.

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

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

3.3.3 Adjacency-Based Representations

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

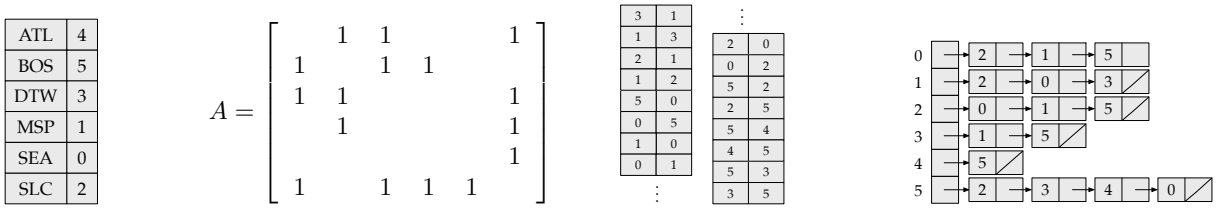
$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \qquad a_{ij} = a_{ji} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

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

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

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

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



(a) An enumeration of the airport graph given in 3.1a.

(b) The adjacency matrix representation of the graph given in Figure 3.1a, using the enumeration given in Figure 3.2a.

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

(d) The compressed sparse adjacency matrix representation.

Figure 3.2 — Adjacency matrix representations of the airport graph model.

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

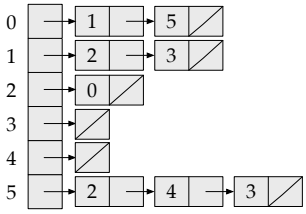
(a) An enumeration of the circuit graph given in 3.1b.

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

(b) The adjacency matrix representation of the graph given in Figure 3.1b, using the enumeration given in Figure 3.3a.

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

(c) The coordinate sparse adjacency matrix representation.



(d) The compressed sparse adjacency matrix representation.

Figure 3.3 — Adjacency matrix representations of the circuit graph model.

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

(a) Table of actors.

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

(b) Table of movies.

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

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

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

(d) A table of movies with starring actors.

Figure 3.4 — Illustrative simplification of IMDB actor and movie data.

3.4 Bipartite Graphs

So far, we have been considering graphs where edges in E are pairs of vertices, which are taken from a single set V. We refer to such a graph as a *unipartite* graph. But consider again the Kevin Bacon example. The source for the information comprising the Kevin Bacon data is the Internet Movie Database (IMDB). However, the IMDB does not contain any explicit information about the relationships between actors. Rather it contains files of tabular data, one of which contains an entry for each movie with the list of actors that have appeared in that movie, and another of which contains an entry for each actor with the list of movies that actor has appeared in ("movie-actor" and "actor-movie" tables, respectively). Such tables are shown in Figure 3.4.[6] Thus, a graph, as we have defined it, cannot model the IMDB.

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

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

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

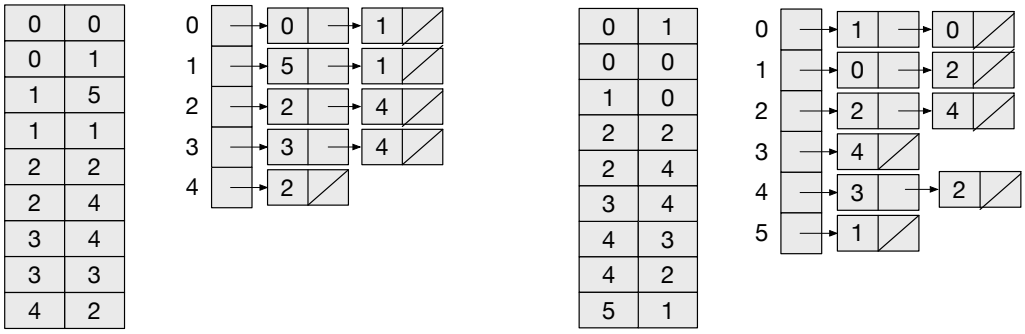
From this adjacency matrix representation we can readily construct coordinate and compressed sparse representations. The only structural difference between the representations of a structurally bipartite graph and that of a unipartite graph is that of vertex cardinality. That is, in a unipartite graph,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

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

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

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

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



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

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

Figure 3.5 — Sparse adjacency representations (edge lists and adjacency lists) for IMDB actor and movie data.

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

3.5 Partitioned Graphs

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

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

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

3.6 From Data to Graph

3.6.1 Columnar Data

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

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

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

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

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

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

3.6.2 Converting an Edge List to an Adjacency List

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

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

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

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

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

Acknowledgements

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

Portions of *Andrew Lumsdaine's* time was supported by NSF Award OAC-1716828 and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

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

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

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.