

# Graph Library: Background and Terminology

Document #: **D3127r1**  
Date: 2024-02-05  
Project: Programming Language C++  
Audience: Library Evolution  
SG19 Machine Learning  
SG14 Game, Embedded, Low Latency  
SG6 Numerics  
Revises: P3127r0  
Reply-to: Phil Ratzloff (SAS Institute)  
[phil.ratzloff@sas.com](mailto:phil.ratzloff@sas.com)  
Andrew Lumsdaine  
[lumsdaine@gmail.com](mailto:lumsdaine@gmail.com)  
Contributors: Kevin Deweese  
Muhammad Osama (AMD, Inc)  
Jesun Firoz  
Michael Wong (Intel)  
Jens Maurer  
Richard Dosselmann (University of Regina)  
Matthew Galati (Amazon)  
Guy Davidson (Creative Assembly)

# 1 Getting Started

This paper is one of several interrelated papers for a proposed Graph Library for the Standard C++ Library. The Table 1 describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now separated into the following papers.
<a href="#">P3126</a>	Active	<b>Overview</b> , describes the big picture of what we are proposing.
<a href="#">P3127</a>	Active	<b>Background and Terminology</b> provides the motivation, theoretical background, and terminology used across the other documents.
<a href="#">P3128</a>	Active	<b>Algorithms</b> covers the initial algorithms as well as the ones we'd like to see in the future.
<a href="#">P3129</a>	Active	<b>Views</b> has helpful views for traversing a graph.
<a href="#">P3130</a>	Active	<b>Graph Container Interface</b> is the core interface used for uniformly accessing graph data structures by views and algorithms. It is also designed to easily adapt to existing graph data structures.
<a href="#">P3131</a>	Active	<b>Graph Containers</b> describes a proposed high-performance <a href="#">compressed_graph</a> container. It also discusses how to use containers in the standard library to define a graph, and how to adapt existing graph data structures.
<a href="#">P3337</a>	Soon	<b>Compares</b> the performance and usage syntax with other graph libraries.

Table 1: Graph Library Papers

Reading them in order will give the best overall picture. If you're limited on time, you can use the following guide to focus on the papers that are most relevant to your needs.

## Reading Guide

- If you're **new to the Graph Library**, we recommend starting with the *Overview* paper ([P3126](#)) to understand the focus and scope of our proposals.
- If you want to **understand the theoretical background** that underpins what we're doing, you should read the *Background and Terminology* paper ([P3127](#)).
- If you want to **use the algorithms**, you should read the *Algorithms* paper ([P3128](#)) and *Graph Containers* paper ([P3131](#)).
- If you want to **write new algorithms**, you should read the *Views* paper ([P3129](#)), *Graph Container Interface* paper ([P3130](#)), and *Graph Containers* paper ([P3131](#)). You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.
- If you want to **use your own graph container**, you should read the *Graph Container Interface* paper ([P3130](#)) and *Graph Containers* paper ([P3131](#)).
- If you want to see how the library in this proposal **stacks up against other graph libraries** in performance and usage syntax, you should read the *Comparison* paper ([P3337](#)).

# 2 Revision History

## D3127r0

- Split from the P1709r5 *Overview and Introduction* section and expanded with more details and examples. Also added *Getting Started* section.

### 3 Naming Conventions

Table 2 shows the naming conventions used throughout the Graph Library documents.

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

Table 2: Naming Conventions for Types and Variables

## 4 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 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 their 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, and 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 (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.

Along with these graph abstractions are the graph algorithms that are widely used for solving problems from these domains. 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 representations 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.

## 5 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 available types of data are actor-movie and movie-actor relationships. See Section 8 below.

## 6 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”) [1]. In defining terminology that is rich enough, yet precise enough, to be used as the basis of a C++ graph library, we emphasize the difference between a graph (an abstraction of entities

in a domain, along with their relationships) and the *representation* of a graph (a structure suitable for use by algorithms and/or for code)<sup>1</sup>.

## 7 Summary of Key Takeaways

A very brief summary of our terminology is the following:

- A graph comprises a set of vertices  $\{V\}$  and a set of edges  $\{E\}$ , written  $G = \{V, E\}$ .
- Expressing algorithms (mathematically as well as in code) requires a *representation* of a graph, the most basic of which is an *adjacency matrix*. An adjacency matrix is constructed using an *enumeration* of the vertices, not the vertices themselves.
- In addition to the (dense) adjacency matrix representation, we consider three sparse representations: coordinate, compressed, and packed coordinate. The sparse forms store *indices* used in the enumeration.
- The coordinate and compressed forms of the adjacency matrix, as taken from linear algebra, respectively correspond to the graph theoretical representations of *edge list* and *adjacency list*.

### 7.1 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 write  $G = \{V, E\}$  to express that the two sets  $V$  and  $E$  define a graph  $G$ .

Figures 1a and 1b show two examples of graph models, a network of airline routes between cities and a social network of names and followers. The figures indicate the domain-specific data to be modeled and the sets  $V$  and  $E$  for each graph. Each figure also includes a node and link diagram, a commonly-used graphical<sup>2</sup> notation.

### 7.2 Graph Representation: Enumerating the Vertices

To reason about graphs, and to write algorithms for them, we require a concrete *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<sup>3</sup>.

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 all graph representations: we *enumerate the vertices*. That is, we assign an index to each element of  $V$  and write  $V = \{v_0, v_1, \dots, v_{n-1}\}$ . Based on that enumeration, elements of  $E$  are expressed in the form  $\{v_i, v_j\}$ . Similarly, we can enumerate the edges, and write  $E = \{e_0, e_1, \dots, e_{m-1}\}$ , 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  $e_k$ .

<sup>1</sup>An example of the kind of ambiguity about graphs arising in typical usage is shown in Appendix A

<sup>2</sup>An unfortunate collision of terminology.

<sup>3</sup>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.

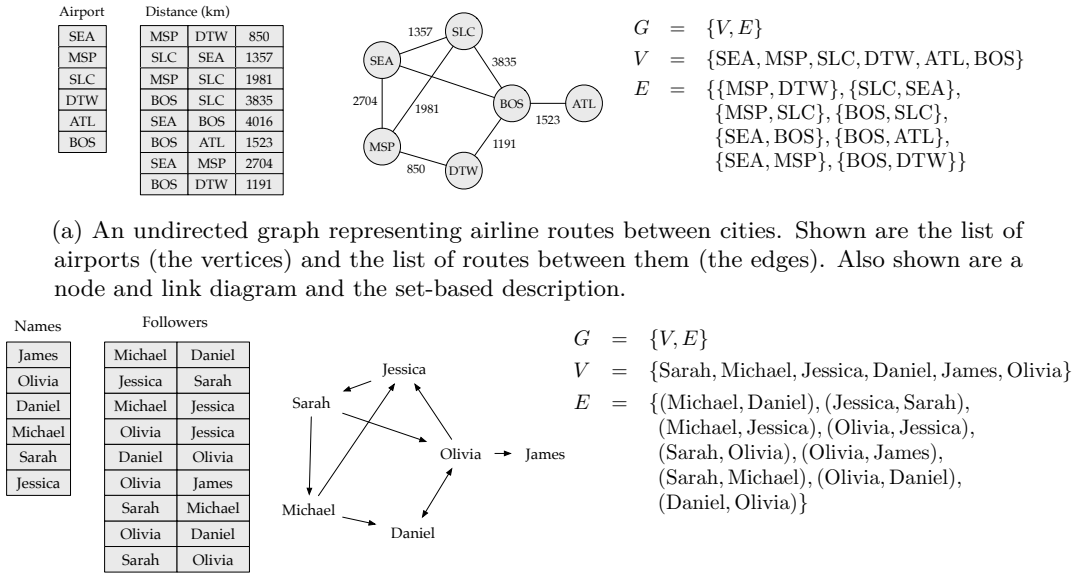


Figure 1: Graph models of an airline route system and of a social media follower network.

- 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 the neighbors of  $v_i$  is the *neighborhood* of  $v_i$ .
- A *path* is a sequence of vertices  $v_0, v_1, \dots, v_{k-1}$  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  $i = 0, 1, \dots, k - 2$ .

### 7.3 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

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

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”). We note that the difference between the adjacency matrices for a directed vs an undirected graph is that and the adjacency matrix for an undirected graph has  $a_{ji} = 1$  whenever  $a_{ij}$  is equal to one. That is, it is symmetric.

Here we can see also 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. The enumeration corresponding to vertices is implicit: the neighbor information for vertex  $v_i$  is stored on row  $i$  of the matrix. Similarly, we don’t store an edge  $(v_i, v_j)$  explicitly, but rather an indicator as to whether  $(v_i, v_j)$  exists in  $E$  or not.

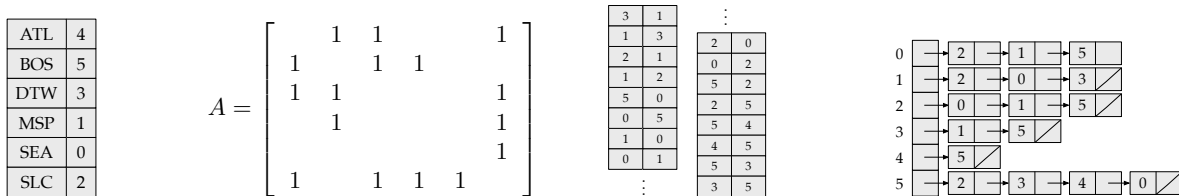
As a data structure to use for algorithms, the adjacency matrix is not very efficient, neither in terms of storage (which, at  $|V| \times |V|$  is prohibitive), nor for computation, because for many graphs, the adjacency matrix contains

almost all zero elements. 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  $A$ .

**NB:** 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<sup>4</sup> — 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 that represent adjacencies between vertices. In the case that  $C$  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 (is adjacent to)  $v_i$  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  $a_{ij} = 1$ )<sup>5</sup>. 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*<sup>6</sup>. 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  $G$ , based on an enumeration of the vertices in  $\{V\}$ .

Illustrations of the adjacency-matrix representations of the airline route graph and the electronic instagram graph are shown in Figures 2 and 3, respectively.



(a) An enumeration of the airport graph given in 1a.

(b) The adjacency matrix representation of the graph given in Figure 1a, using the enumeration given in Figure 2a.

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

(d) The compressed sparse adjacency matrix representation.

Figure 2: Adjacency matrix representations of the airport graph model.

## 8 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

<sup>4</sup>That is, the adjacency matrix is symmetric.

<sup>5</sup>The compressed sparse adjacency matrix is identical to the compressed sparse row format from linear algebra

<sup>6</sup>We concede that “adjacency list” rolls off the tongue much more easily than “compressed sparse adjacency matrix representation of a graph.”



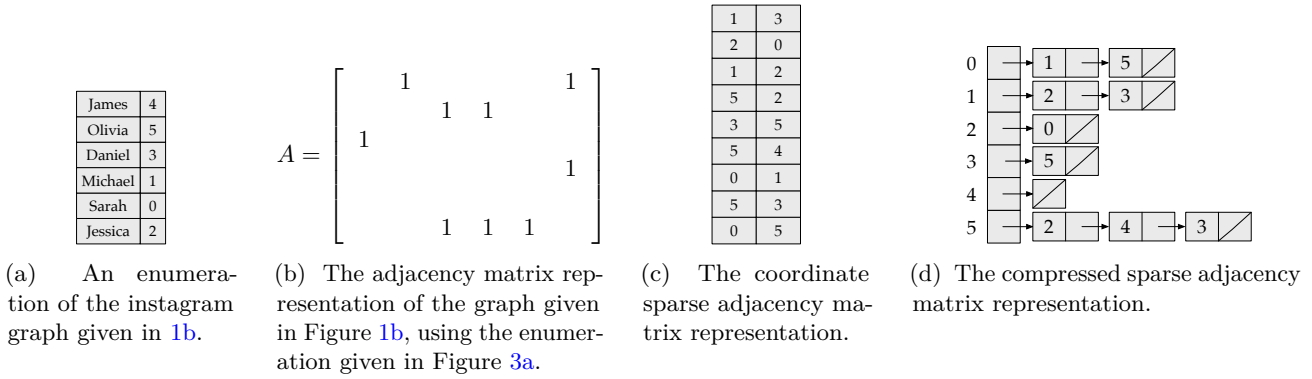


Figure 3: Adjacency matrix representations of the instagram graph model.

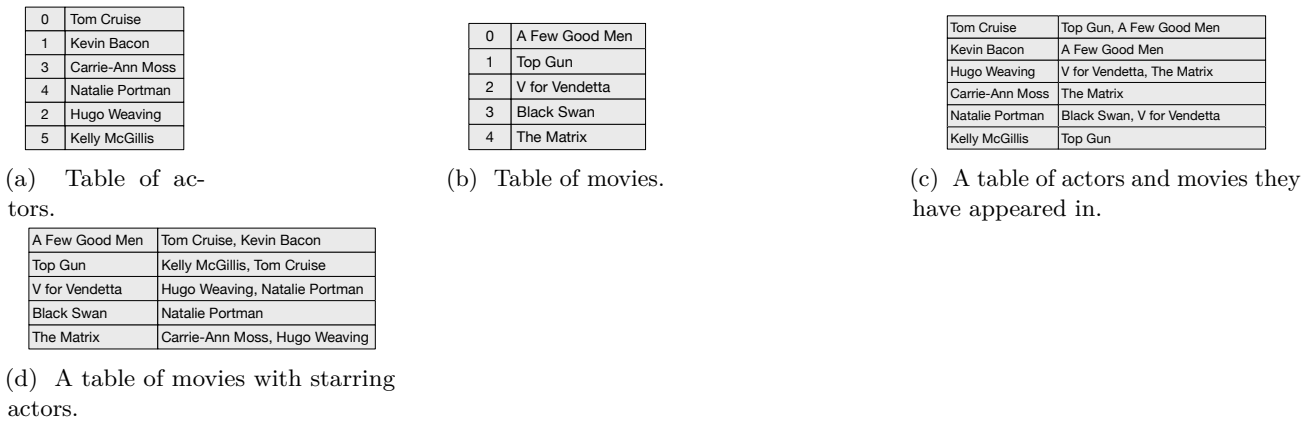


Figure 4: Illustrative simplification of IMDB actor and movie data.

(“movie-actor” and “actor-movie” tables, respectively). Such tables are shown in Figure 4.<sup>7</sup> 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  $U = \{u_0, u_1, \dots, u_{n0}\}$  and  $V = \{v_0, v_1, \dots, v_{n1}\}$ , and the edge set  $E$  consists of pairs  $(u_i, v_j)$  where  $u_i$  is in  $U$  and  $v_j$  is in  $V$ .

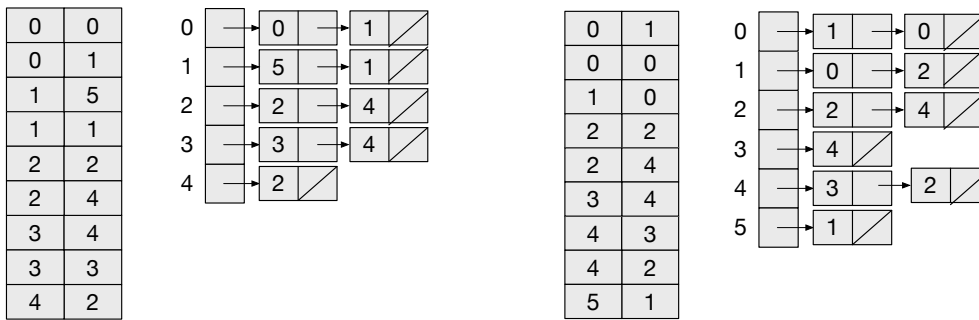
The *adjacency matrix representation* of a *structurally bipartite graph* is a  $|U| \times |V|$  matrix  $A = (a_{ij})$  such that,

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

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

<sup>7</sup>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>.

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.



(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 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.

## 9 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.,  $V = \{V_0, V_1, \dots\}$  which we enumerate as  $V_0 = \{v_0, v_1, \dots, v_{n_0-1}\}$ ,  $V_1 = \{v_{n_0}, \dots, v_{n_1-1}\}$  and so on. Each  $V_i$  is a *partition* of  $V$ . The total enumeration of  $V$  is  $V = \{v_0, v_1, \dots, v_{n-1}\}$ . 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 across partitions.

## 10 Direct Representations

Another approach to representing a graph is to model an adjacency list (e.g., Figure 2d or 3d) directly. That is, we can represent a vertex as a class and an edge as a class, and use pointers to represent adjacency.

For example, the following structures could be used to directly represent a graph (these are simplified, and slightly modernized, versions of the structs used in Donald Knuth's original Stanford Graph Base<sup>8</sup>):

<sup>8</sup>Stanford Graph Base was written in pre-ANSI C

```

class Vertex {
    Arc* arcs;
    int u, v, w;
};

class Arc {
    Vertex* tip;
    Arc* next;
    int a, b;
};

class Graph {
    Vertex* vertices;
    int uu, vv, ww;
};

```

In this example, a 'linlineGraph' is an array of `Vertex`. Each `Vertex` has a pointer to a list of `Arc` (an edge) and each `Arc` has a pointer to its target `Vertex`. Each of the classes contains not only the adjacency information, but properties as well (represented here by `int` fields in each of the classes);

Using modern C++ practice, we might instead express this as

```

template <class... Props>
class Vertex {
    std::forward_list<Arc> arcs;
    std::tuple<Props...> props;
};

template <class... Props>
class Arc {
    Vertex* tip;
    std::tuple<Props...> props;
};

template <class... Props>
class Graph {
    std::vector<Vertex> vertices;
    std::tuple<Props...> props;
};

```

Much of terminology for graphs still applies in a direct representation, except, of course, we have structures representing the different components of a graph, rather than their indices. There are a number of variations one could consider to this representation, such as using `std::vector` rather than `std::forward_list` to store outgoing `Arc` in a `Vertex`.

One advantage of a direct representation is that it can be embedded. That is, the `Vertex` / `Arc` representation may be implicit (or easily incorporated into) existing structures within an application. For example, one might have data structures to represent an electronic circuit:

```

struct two_terminal {
    node* from;
    node* to;
    double current;
};

struct resistor : public two_terminal {
    double conductance;
};

```

```

struct capacitor : public two_terminal {
    double capacitance;
};

struct node {
    std::forward_list<two_terminal> elements;
    double voltage;
};

struct circuit {
    std::vector<node> nodes;
};

```

Note that, although structures and fields are differently named, a circuit is inherently a direct representation of a graph.

## A On Ambiguous Terminology

Here we show how graph terminology in practical use is often ambiguous (and why we were so painstaking in our definitions). The definitions of graph and adjacency list from *The Handbook of Graph Theory, Second Edition* is a typical example:<sup>9</sup>

**D1:** A directed graph or digraph  $G = (V, E)$  consists of a finite, nonempty set of vertices  $V$  and a set of edges  $E$ . Each edge is an ordered pair  $(v, w)$  of vertices.

**E1:** A line drawing of a graph  $G = (V, E)$  is shown in Figure 1.1.1 [omitted, ed.]. It has vertex-set  $V = u, v, w, x$  and edge-set  $E = a, b, c, d, e, f$ .

**D9:** An adjacency list representation for a graph or digraph  $G = (V, E)$  is an array  $L$  of  $|V|$  lists, one for each vertex in  $V$ . For each vertex  $i$ , there is a pointer  $L_i$  to a linked list containing all vertices  $j$  adjacent to  $i$ .

The ambiguity occurs between **D1/E1** and **D9**.

In **D1**, a vertex  $v \in V$  is an element of unspecified type and in **E1** a particular case is given in  $E1$  where each vertex is a letter. Indeed, in the literature, it is common to identify vertices in  $V$  with domain entities that the graph is modeling (“a vertex is a person”). These are (literally) textbook definitions, and no one would bat an eye in reading them.

But if we look critically at **D9**, what is meant by “vertex” is completely ambiguous. The adjacency list in **D9** is representing graph  $G = \{V, E\}$ , that is, by **D1**, a vertex is a member of  $V$  and its type is unspecified. However, **D9** also uses the term “vertex” to refer to indices – “vertex”  $i$  indexes  $L_i$ , the linked list  $L_i$  stores “vertices” – yet, it is index values, not vertices that are being stored. Beyond “vertex,” many other terms are ambiguously used when referring to graphs. Perhaps the most egregious ambiguity is the term “graph” itself – which is used to refer to the graph  $G = \{V, E\}$ , to the representation of a graph, or even to the entities being modeled by the graph.

A more careful distinction needs to be made between a graph and its representation – and between the abstractions associated with a graph and the abstractions associated with the representation of a graph. We can’t use the same term (e.g., “vertex”) to mean two different things.

## B From Data to Graph

### B.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

<sup>9</sup>Although we take this example of ambiguity from the *Handbook*, in general the *Handbook* is a rigorous and extensive collection of important theoretical results in graph theory. We borrow from its formatting conventions here.

whitespace)<sup>10</sup>. 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);
}
```

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.

## B.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
```

<sup>10</sup>We take a broad view of what a comma is.

```
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., ISO CPP Foundation, Khronos and the Standards Council of Canada.

*Muhammad Osama's* time was made possible by Advanced Micro Devices, Inc.

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

## References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 4 ed., 2022.
- [2] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, Dec. 2001.