Graph Library: Background and Terminology

Document #:	**D3127r1**
Date:	2024-09-12
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 Andrew Lumsdaine 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) Oliver Rosten

1 Getting Started

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

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

Table 1: Graph Library Papers

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

Reading Guide

— If you're **new to the Graph Library**, we recommend starting with the *Overview* (P3126) paper to understand the focus and scope of our proposals. You'll also want to check out how it stacks up against other graph libraries in performance and usage syntax in the *Comparison* (P3337) paper.

— If you want to **understand the terminology and theoretical background** that underpins what we're doing, you should read the *Background and Terminology* (P3127) paper.

— If you want to **use the algorithms**, you should read the *Algorithms* (P3128) and *Graph Containers* (P3131) papers. You may also find the *Views* (P3129) and *Graph Container Interface* (P3130) papers helpful.

— If you want to **write new algorithms**, you should read the *Views* (P3129), *Graph Container Interface* (P3130), and *Graph Containers* (P3131) papers. You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.

— If you want to **use your own graph data structures**, you should read the *Graph Container Interface* (P3130) and *Graph Containers* (P3131) papers.

2 Revision History

D3127r0

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

D3127r1

— Move text from the Motivation section to the Overview section in P3126.

— Remove the Six Degrees of Kevin Bacon example, a duplication of the same example in P3126.

— Update the Direct Representation with C++ code examples, and add content for special cases that occur in graphs such as *self-loops*, *multigraph*, *cycle*, *tree*, etc.

— Add a sections on Incident Matrices and Regarding Algorithms.

3 Naming Conventions

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

Template Parameter	Type Alias	Variable Names	Description
`G`			Graph
	`graph_reference_t<G>`	`g`	Graph reference
`GV`		`val`	Graph Value, value or reference
`EL`		`el`	Edge list
`V`	`vertex_t<G>`		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, or vvf(uid) → vertex value, depending on requirements of the consume algorithm or view.
`VProj`		`vproj`	Vertex info projection function: `vproj(x)` → `vertex_info<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, or evf(eid) → edge value, depending on the requirements of the consuming algorithm or view.
`EProj`		`eproj`	Edge info projection function: `eproj(x)` → `edge_info<VId,Sourced,EV>` .

Table 2: Naming Conventions for Types and Variables

4 Graph Background

[ANDREW: We should use a standard latex macro for C++ – it doesn't really look right in plain text.]

For clarity in the material contained in other documents, here we briefly review some of the basic terminology of graphs. We use commonly accepted terminology for graph data structures and algorithms and specifically adopt the 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)[1]. We note that because of the precision with which we define representations, there are results that may be unexpected for some.

5 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* $\{E\}$, and is 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* defined by the enumeration.

— The *coordinate* and *compressed* forms of the adjacency matrix[2] respectively correspond to representations of the graph theoretical *edge list* and *adjacency list*.

6 Basic Terminology

To model the relationships between entities in some domain, a *graph* G comprises two sets: a *vertex set* V, whose elements correspond to the domain 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 we can express that by the existence of a pair $\{u, v\}$ in E. We write $G = \{V, E\}$ to express that the two sets V and E define a graph G. We can also describe set membership of a vertex in V or and edge in E with set notation as $v \in V$ or $e \in E$, but we will generally try to avoid using too much purely mathematical notation.

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[3] notation.

6.1 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[4].

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 $V = \{v_0, v_1, \ldots v_{n-1}\}$. Based on that enumeration, elements of E are expressed in the form $\{v_i, v_j\}$. Similarly,

[1] An example of the kind of ambiguity about graphs arising in typical usage is shown in Appendix A

[2] the terems coordinate and compressed are taken from linear algebra.

[3] An unfortunate collision of terminology.

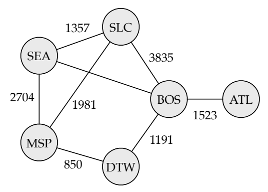
[4] In fact, if we are to be completely precise, the library we are proposing is one of algorithms and data structures for graph representations. We will make concessions to commonly accepted terminology, while precisely defining that terminology.

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



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

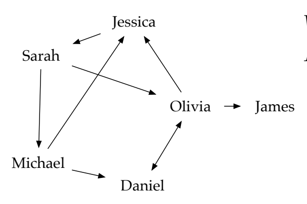
(a) An undirected graph representing airline routes between cities. Shown are the list of airports (the vertices) and the list of routes between them (the edges). Also shown are a node and link diagram and the set-based description.

Names

James
Olivia
Daniel
Michael
Sarah
Jessica

Followers

Michael	Daniel
Jessica	Sarah
Michael	Jessica
Olivia	Jessica
Daniel	Olivia
Olivia	James
Sarah	Michael
Olivia	Daniel
Sarah	Olivia



$$\begin{aligned} G &= \{V, E\} \\ V &= \{\text{Sarah}, \text{Michael}, \text{Jessica}, \text{Daniel}, \text{James}, \text{Olivia}\} \\ E &= \{(\text{Michael}, \text{Daniel}), (\text{Jessica}, \text{Sarah}), \\ &\quad (\text{Michael}, \text{Jessica}), (\text{Olivia}, \text{Jessica}), \\ &\quad (\text{Sarah}, \text{Olivia}), (\text{Olivia}, \text{James}), \\ &\quad (\text{Sarah}, \text{Michael}), (\text{Olivia}, \text{Daniel}), \\ &\quad (\text{Daniel}, \text{Olivia})\} \end{aligned}$$

(b) A directed graph representing followers in a social network. Shown in the diagram are the names of members (the vertices) and the members they follow (the edges). Since following is a one-way relationship, we use a directed graph to model it. Also shown are a node and link diagram and the set-based description.

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

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 the neighbors of v_i is the *neighborhood* of v_i.

— 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 $i = 0, 1, \ldots, k-2$.

There are some special cases that deserve mention, as their presence or absence may determine algorithmic properties.

— A *self-loop* is an edge from a vertex v_i to itself, that is, there is an edge v_i, v_i in E.

— An *isolated vertex* v_i is one that has no edge incident on it, that is, a vertex v_i for which there is no edge $\{v_i, v_j\}$ nor $\{v_j, v_i\}$.

— A *multigraph* is a graph G for which there exist multiple edges between the same vertices, i.e., there are multiple edges $\{v_i, v_j\}$ for the same i and j in E.

— A *hypergraph* is a graph $G = \{V, E\}$ for which the elements of E are arbitrary subsets of V. That is, elements of E may be $\{v_i, v_j, v_k, \ldots\}$. Consideration of hypergraphs is outside the scope of this proposal.

— A *hypersparse* graph is a graph for which the enumeration is not contiguous. That is for $V = \{v_i, v_j, v_k, \ldots\}$, with $i < j < k < \ldots$ the set $\{i, j, k, \ldots\}$ may not be contiguous and may not start at 0.

— A *path* is sequence of edges $\{v_i, v_j\}, \{v_j, v_k\}, \{v_k, v_l\}, \ldots$ such that every v_i is distinct. That is, any v_i appears once and only once in an edge $\{v_j, v_i\}$ and once and only once in an edge $\{v_i, v_k\}$.

— A *cycle* is a path such that every vertex appears twice, that is, for every v_i there is an edge $\{v_j, v_i\}$ and an edge $\{v_i, v_k\}$. In terms of the sequence above, a path is a cycle if the second vertex of the last edge is the first vertex of the first edge.

— A *directed acyclic graph (DAG)* is a directed graph with no cycles.

— A *tree* is a connected graph with no cycles. Trees are a special case of graphs but are important enough that they have their own rich theory (and corresponding software). As such, we omit trees from this proposal and look forward to separate library proposals for trees.

— A *subgraph* of $G = \{V, E\}$ is a graph $H = \{V, F\}$ such that F is a subset of E.

— A *spanning tree* is a subgraph of G that is also a tree.

If any of these properties is important to the correct functioning of an algorithm, either positively or negatively, it will be part of the corresponding requirements of the algorithm. In general we assume that graphs are not multigraphs, not hyperspase, and that they do not have self-loops.

[ANDREW: We should probably have pictures for all of these – and others – saves 1k (or 10k) words.]

6.2 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"). 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 $a_i j$ 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[5] — 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

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

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 $a_{ij} = 1$)[6]. 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*[7]. 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.

ATL	4
BOS	5
DTW	3
MSP	1
SEA	0
SLC	2

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

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

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

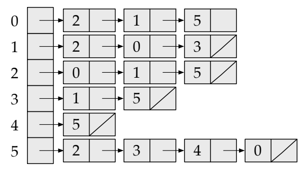
3	1
1	3
2	1
1	2
5	0
0	5
1	0
0	1

⋮

⋮

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

(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.

James	4
Olivia	5
Daniel	3
Michael	1
Sarah	0
Jessica	2

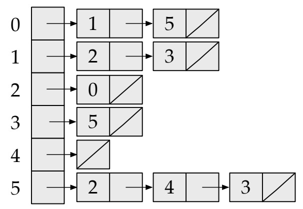
(a) An enumeration of the instagram graph given in 1b.

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

(b) The adjacency matrix representation of the graph given in Figure 1b, using the enumeration given in Figure 3a.

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

(c) The coordinate sparse adjacency matrix representation.



(d) The compressed sparse adjacency matrix representation.

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

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

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

6.3 Incident Matrices

An *Incidence matrix* of a directed graph G is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

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

We note that the product $BB^\top$ of an incident matrix B is the adjacency matrix of the graph G, i.e., $G = BB^\top$.

7 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

```
struct Edge;
struct Vertex;

struct Vertex {
  std::forward_list<Edge> edges;
  std::string name;
};

struct Arc {
  Vertex* tip;
  double distance;
};

struct Graph {
  std::vector<Vertex> vertices;
};
```

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`.

Direct representations of graphs will often be implicitly part of other structures ("embedded") in a given 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, with nodes as vertices and two_terminal elements as edges.

8 Bipartite Graphs

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 4: Illustrative simplification of IMDB actor and movie data.

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 4.[8] 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.

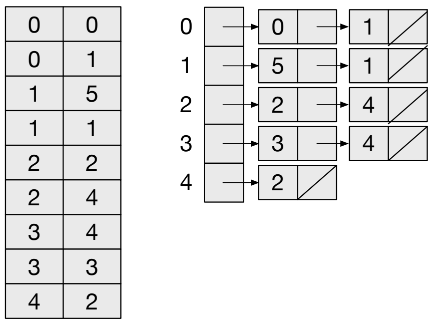
The *adjacency matrix representation of a structurally bipartite graph* is a $|U| \times |V|$ matrix $A = (a_{ij})$ such that,

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

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

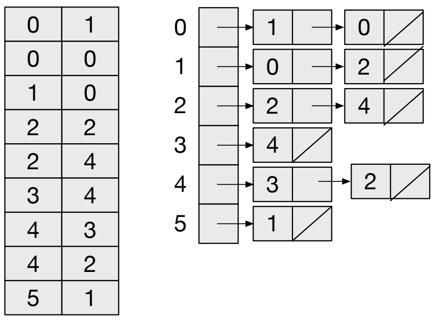
[8]This is a greatly simplified version of the CSV files that actually comprise the IMDB. The full set of files is available for non-commercial use at https://datasets.imdbws.com.

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.



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

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



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

(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.

We note that a structurally bipartite graph may have an edge (u_i, v_i), that is, an edge between two vertices with the same index. Even though u_i and v_i are not the same vertex, we opt to consider such an edge to still be called a self-loop.

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., $V = \{V_0, V_1, \ldots\}$ 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.

10 Regarding Algorithms

[Andrew: We need to note that we can see some abstract properties of graph representations that are universal and can used for defining concepts. I suggest that that be done where we define our concepts.]

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:[9]

> **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 whitespace)[10]. 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);
}
```

[9]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.
[10]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.

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
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);
}
```

C Graphs and Sparse Matrices

The relationship between graphs and sparse matrices is natural and important enough that a few words are in order.

In numerical linear algebra, a sparse matrix is one that only stores elements of interest [11]. Elements that are not stored are assumed to be zero. The information necessary to use a matrix includes the row index, the column index, and the entry value itself—abstractly a triple (i, j, v) The elements can be stored in coordinate form, where each triple is stored (either as separate arrays or as tuples in a single array), or in compressed sparse form, which compresses one of the index dimensions and stores the other index and the value.

A sparse matrix can be considered as a structurally bipartite graph representation. Suppose that we have a sparse matrix A represented as $\{(i, j, a_{ij})\}$. We can create a structurally bipartite $J = \{U, V, E\}$ such that (i, j) is in E if and only if $\{(i, j, a_{ij})\}$ is in A. The coordinate and compressed representations for a sparse matrix are the same as for the graph (and, in fact, the terminology "coordinate" and "compressed sparse" originate in sparse numerical linear algebra). For a matrix, the sets U and V have a particular meaning. Either the indices of U consist of row numbers and V of column numbers, or vice versa. In the former case, a compressed representation is known as "compressed sparse row." In the latter case, it is known as "compressed sparse column."

[Andrew: This code is from nwgraph, need to bring it up to std::graph] The following code snippet illustrates a sparse matrix vector product when a compressed adjacency representation is interpreted as a compressed sparse row matrix.

```
for (auto&& [row, u_neighbors] : make_neighbor_range(graph)) {
  for (auto&& [col, val] : u_neighbors) {
    y[row] += x[col] * val;
  }
}
```

The following code snippet illustrates a sparse matrix vector product but for a compressed sparse row matrix.

```
for (auto&& [col, u_neighbors] : make_neighbor_range(graph)) {
  for (auto&& [row, v] : u_neighbors) {
    y[row] += x[col] * v;
  }
}
```

Acknowledgements

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

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

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

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

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

[11]These are typically called "non-zeroes", though the stored value could be zero.

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.