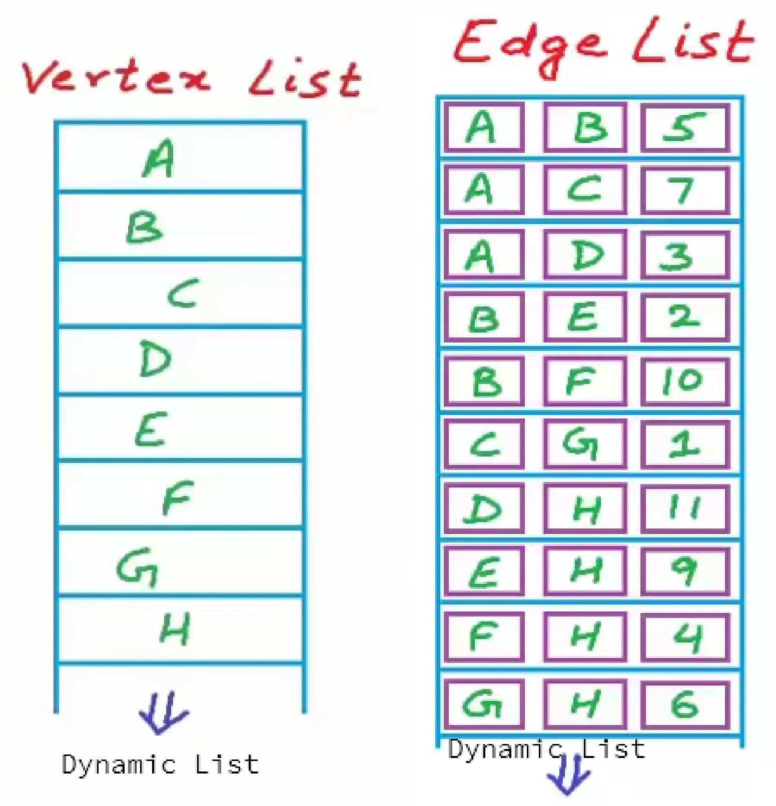
Representations of graph

Edge List

we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v.



Implementation performance for ADT methods

We begin by discussing the ***space usage***, which is *O*(n+m) for representing a graph with n vertices and m edges.

In terms of ***running time***,for reporting the number of vertices or edges, or in producing an iteration of those vertices or edges. By querying the respective list V or E, the numVertices and numEdges methods run in *O*(1) time, and by iterating through the appropriate list, the methods vertices and edges run respectively in *O*(n) and *O*(m) time.

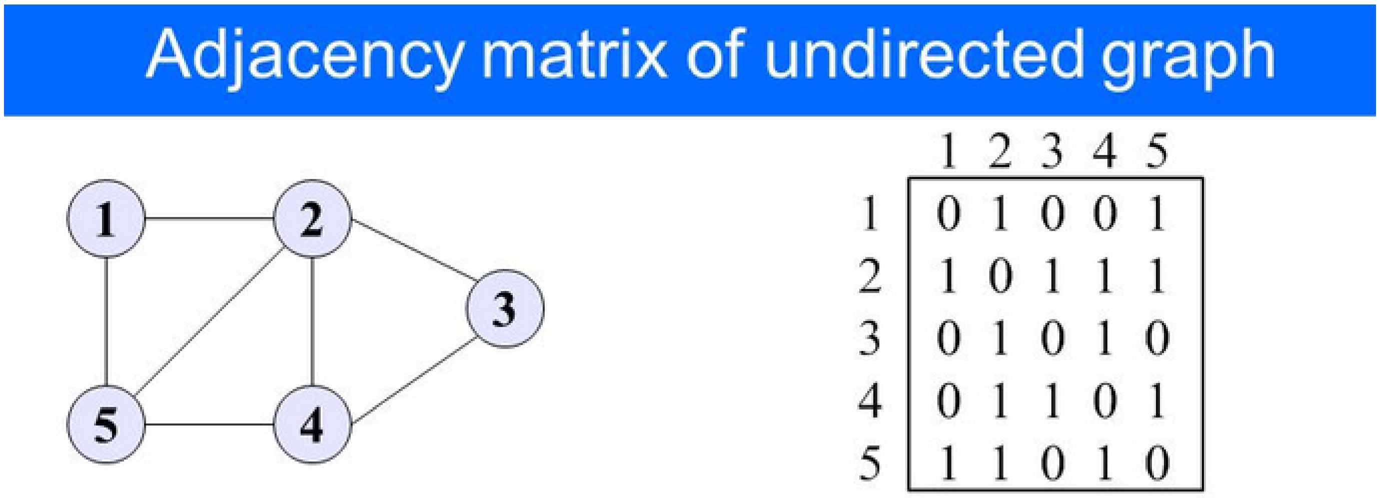
The **most significant limitations** of an edge list structure, are the *O*(m) running times of methods getEdge(u, v), outDegree(v), and outgoingEdges(v) (and corresponding methods inDegree and incomingEdges). The problem is that with all edges of the graph in an unordered list E, the only way to answer those queries is through an exhaustive inspection of all edges.

Finally, we consider the methods that **update the graph**. It is easy to add a new vertex or a new edge to the graph in *O*(1) time. For example, a new edge can be added to the graph by creating an Edge instance, adding that instance to the list E and recording its resulting position within E as an attribute of the edge. That stored position can later be used to locate and remove this edge from E in *O*(1) time, and thus implement the method *removeEdge(e)*. It is worth discussing why the *removeVertex(v)* method has a running time of *O*(m). As stated in the graph ADT, when a vertex v is removed from the graph, all edges incident to v must also be removed.

Another optimization can be done if vertex list is an array with index 0 ... n, than first edge can have value of [0][1][5] denoting edge between A & B which are at index 0 and 1 respectively in vertex array. This way the edge list maintains a list of indexes and weight and not the actual data.

Adjacency matrix

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be `A[][]`, a slot `A[i][j] = 1` indicates that there is an edge from vertex i to vertex j else 0 otherwise. Adjacency matrix for *undirected graph is always symmetric*. Adjacency Matrix is also used to represent weighted graphs. If `A[i][j] = w`, then there is an edge from vertex i to vertex j with weight w.



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Pros: edge insertion and deletion is rapid and takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse, it consumes the same space. Adding a vertex is O(V^2) time. . There is some potential to save space by packing multiple bits per word(using compression) or simulating a triangular matrix on undirected graphs. But using these methods we lose the simplicity that makes adjacency matrices so appealing. Moreover even with these techniques the traversal remains quadratic

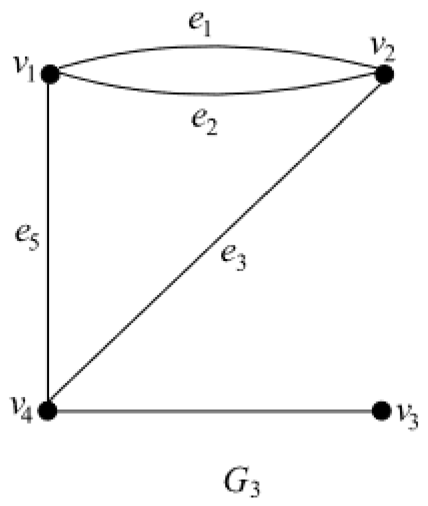
Incidence Matrix

Let G be a graph with n vertices,m edges and without self-loops. The incidence matrix A of G is an n×m matrix $A = [a_{i,j}]$ whose n rows correspond to the n vertices and the m columns correspond to m edges such that

$$a_{ij} = \begin{cases} 0, & \text{if } j^{th} \ edge \ m_j \ is \ incident \ on \ the \ i^{th} \ vertex \\ 1, & \text{if } otherwise \end{cases}$$

It is also called vertex-edge incidence matrix and is denoted by A(G).

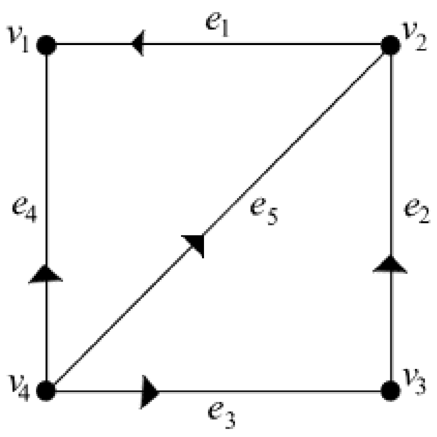
$$A(G_3) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \left[\begin{array}{ccccc} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{array}\right] \end{array}$$



<u>For Directed Graphs:</u>

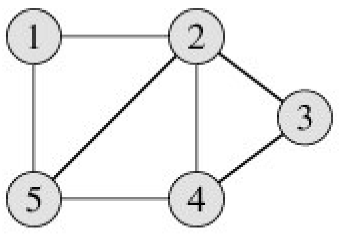
$$a_{ij} = \begin{cases} 1, & \text{if } v_i \ is \ the \ positive \ end \ of \ e_j, \\ -1, & \text{if } v_i \ is \ the \ negative \ end \ of \ e_j, \\ 0, & \text{if } v_i \ is \ not \ incident \ with \ e_j \end{cases}$$

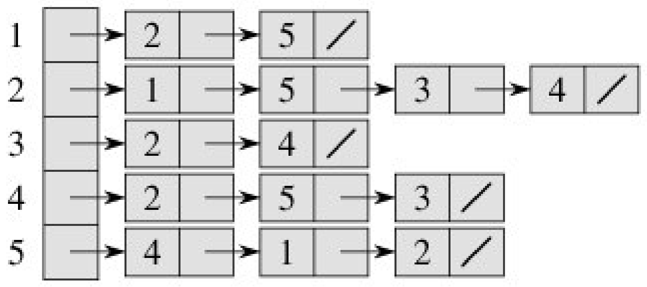
$$F = \begin{array}{c|ccccc} & e_1 & e_2 & e_3 & e_4 & e_5 \\ \hline v_1 & 1 & 0 & 0 & 1 & 0 \\ v_2 & -1 & 1 & 0 & 0 & 1 \\ v_3 & 0 & -1 & 1 & 0 & 0 \\ v_4 & 0 & 0 & -1 & -1 & -1 \end{array}$$



Adjacency List

In adjacency list an array of lists is used. Size of the array is equal to the number of vertices. Let the array be `vertices[]` , an entry `vertices[i]` represents the list of vertices adjacent to the i^{th} vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs `[vertex | weight]` . This organization allows us to more efficiently find all edges incident to a given vertex.





Asymptotically, the space requirements for an adjacency list are the same as an edge list structure, using *O*(n+m) space for a graph with n vertices and m edges.

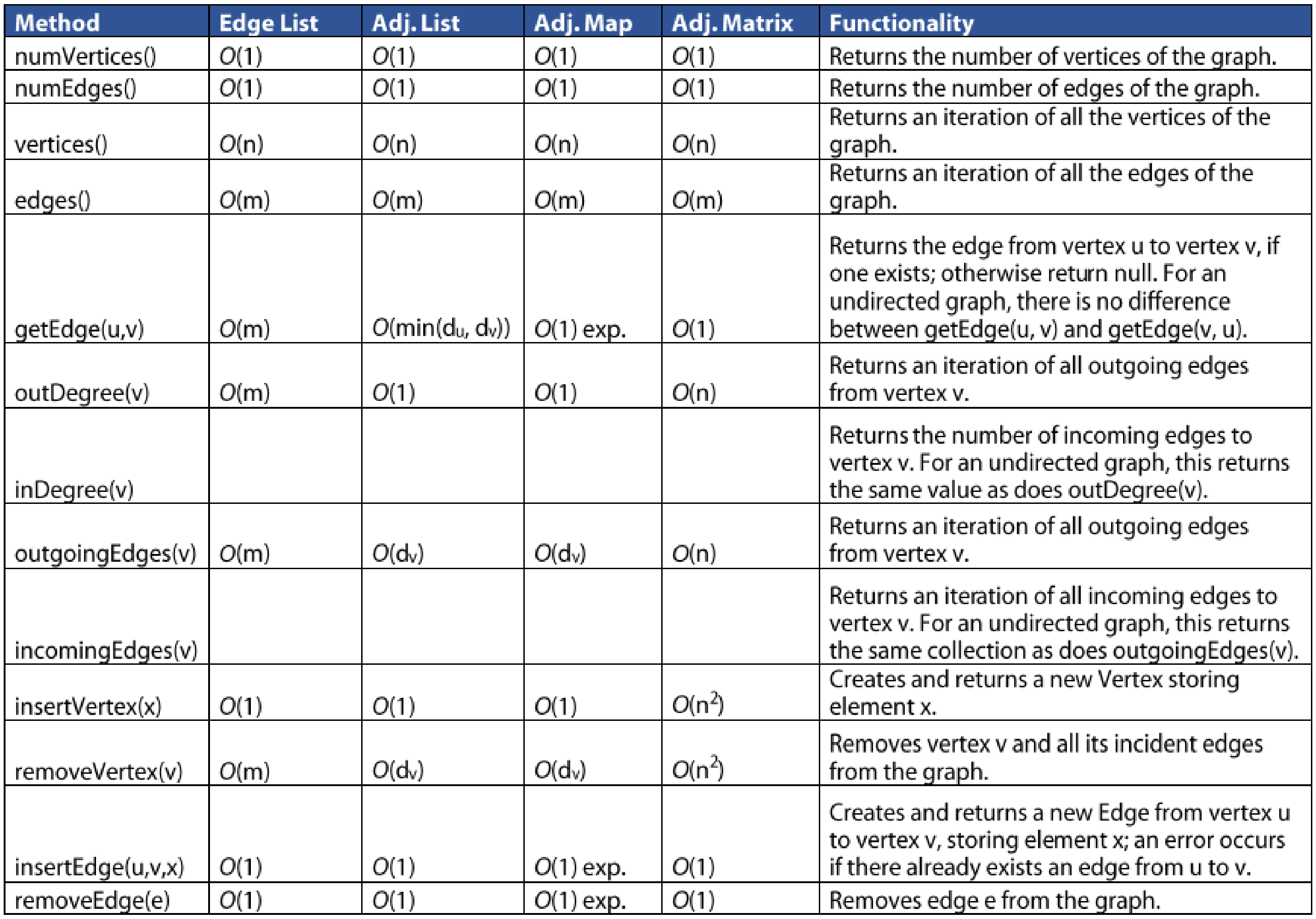
Adjacency Map

It is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (u,v), for example, in O(1) expected time with hashing.

```
// creating adjacency list using arraylist of linkedlist of integers
ArrayList<LinkedList<Integer>> adj_list = new ArrayList<LinkedList<Integer>>();
adj_list.add(new LinkedList<Integer>());
adj_list.get(0).add(3); //directed edge from 0 to 3 vertex

// creating adjacency list using fixed size array of linkedlist of integers
LinkedList<Integer> adjListArray[] = new LinkedList[V];
graph.adjListArray[src].add(dest);
graph.adjListArray[dest].add(src);
```

Performance summary of ADT methods of different graph representation



Method	Edge List	Adj. List	Adj. Map	Adj. Matrix	Functionality
numVertices()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Returns the number of vertices of the graph.
numEdges()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Returns the number of edges of the graph.
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$	Returns an iteration of all the vertices of the graph.
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$	Returns an iteration of all the edges of the graph.
getEdge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
outDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$	Returns an iteration of all outgoing edges from vertex v.
inDegree(v)					Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v)					Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$	Creates and returns a new Vertex storing element x.
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$	Removes vertex v and all its incident edges from the graph.
insertEdge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$	Removes edge e from the graph.

We let n denote the number of vertices, m the number of edges, and d_v the degree of vertex v. Note that the adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n + m)$ space.