Graph: Definitions and Representations

A graph is usually denoted as $G = (V,E)$, where V represents vertices, and E represents edges. There are two types of graphs, directed ones and undirected ones (see Figures below).

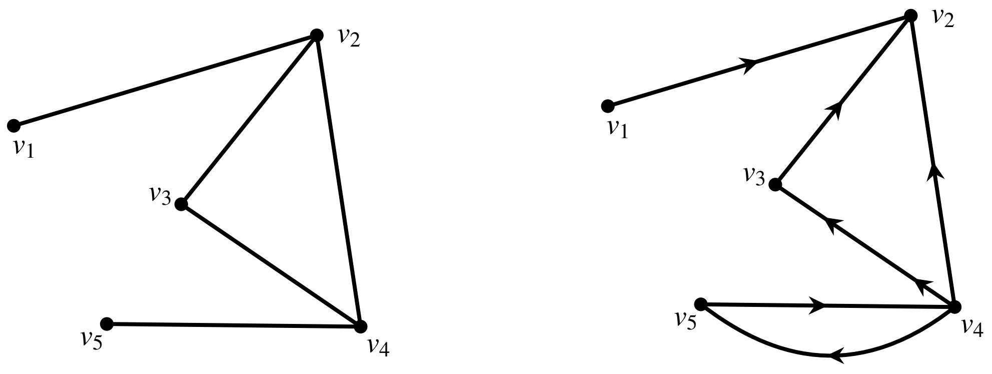
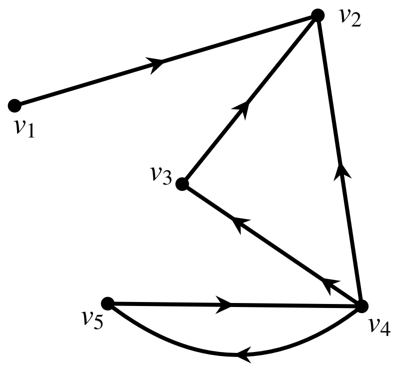


Figure 1: Left: a undirected graph $G = (V,E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1,v_2),(v_3,v_2),(v_4,v_3),(v_4,v_2),(v_4,v_5)\}$. Right: a directed graph $G = (V,E)$, where $V = \{v_1,v_2,v_3,v_4,v_5\}$, and $E = \{(v_1,v_2),(v_3,v_2),(v_4,v_3),(v_4,v_2),(v_4,v_5),(v_5,v_4)\}$.

Notice that for an edge (v_i, v_j) in an undirected graph, the order of v_i and v_j are interchangable, i.e., $(v_i, v_j) = (v_j, v_i)$. In directed graph this is not the case.

Adjacency matrix and adjacency list are two commonly-used data structures to represent a graph. Adjacency matrix uses a binary matrix M of size $|V| \times |V|$ to store a graph $G = (V,E)$: $M[i, j] = 1$ if and only if $(v_i, v_j) \in E$. This definition applies to both directed graphs and undirected graphs.

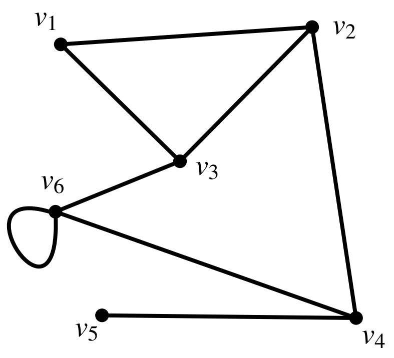
For undirected graphs, clearly the adjacency matrix M is symmetric. If we assume that there is no "self-loop" edges in the form of (v_i, v_i), then the number of "1"s in M is exactly $2|E|$ for undirected graph. The number of "1"s in M is exactly $|E|$ for directed graph.



$$\begin{array}{c|c|c|c|c|c|}
 & v_1 & v_2 & v_3 & v_4 & v_5 \\ \hline
v_1 & 0 & 1 & 0 & 0 & 0 \\ \hline
v_2 & 0 & 0 & 0 & 0 & 0 \\ \hline
v_3 & 0 & 1 & 0 & 0 & 0 \\ \hline
v_4 & 0 & 1 & 1 & 0 & 1 \\ \hline
v_5 & 0 & 0 & 0 & 1 & 0 \\ \hline
\end{array}$$

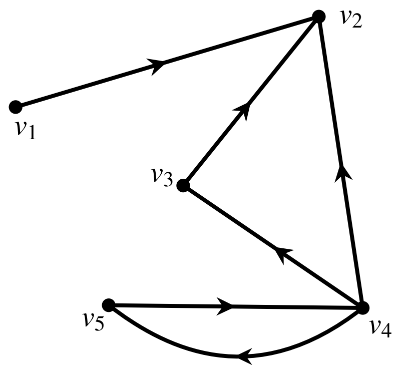
Figure 2: Adjacency matrix representation (directed graph).

Adjacency list maintains a list/array A_i for each vertex $v_i \in V$, where A_i stores $\{v_j \in V \mid (v_i, v_j) \in E\}$, i.e., the adjacent edges/vertices of v_i. A pointer is usually maintained for each vertex v_i that points to the array A_i. And we can use an array of size $|V|$ to store these pointers. Clearly, for undirected graph, $\sum_{v_i \in V} |A_i| = 2|E|$, assuming that there is no "self-loop" edges. For directed graph, $\sum_{v_i \in V} |A_i| = |E|$.



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	0	0	0
v_2	1	0	1	1	0	0
v_3	1	1	0	0	0	1
v_4	0	1	0	0	1	1
v_5	0	0	0	1	0	0
v_6	0	0	1	0	0	1

Figure 3: Adjacency matrix representation (undirected graph).



$v_1 \to (v_2)$

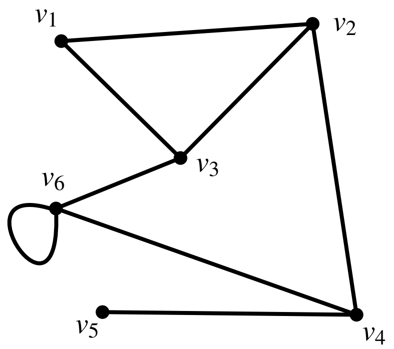
$v_2 \to ()$

$v_3 \to (v_2)$

$v_4 \to (v_2,\ v_3,\ v_5)$

$v_5 \to (v_4)$

Figure 4: Adjacency list representation (directed graph).



$v_1 \to (v_2,\ v_3)$

$v_2 \to (v_1,\ v_3,\ v_4)$

$v_3 \to (v_1,\ v_2,\ v_6)$

$v_4 \to (v_2,\ v_5,\ v_6)$

$v_5 \to (v_4)$

$v_6 \to (v_3,\ v_6)$

Figure 5: Adjacency list representation (undirected graph).

Which one is better, adjacency matrix or adjacency list? Let's consider some measures.

- The space complexity. Clearly, the adjacency matrix needs $\Theta(|V|^2)$ space to store. The adjacency list can be stored in $\Theta(|V|+|E|)$ space, where $\Theta(|V|)$ is used to store all $|V|$ pointers, and $\Theta(|E|)$ is used to store all arrays $\{A_i\}$ as we've seen $\sum_{v_i \in V} |A_i| = |E|$. Therefore, for the space complexity, adjacency list is better, as $|E| = O(|V|^2)$; in particular in *sparse* graphs, $|E|$ will be way smaller than $|V|^2$.

- Querying if $(v_i, v_j) \in E$. Given v_i and v_j, whether $(v_i, v_j) \in E$ can be done in $\Theta(1)$ time if the graph G is represented with an adjacency matrix, as this can be answered by a direct access to $M[i, j]$. If G

is represented with an adjacency list, to check if $(v_i, v_j) \in E$, one needs to tranverse A_i and see if v_j is in A_i or not, and clearly this takes $\Theta(|A_i|)$ time. Note that, if we assume that the vertices in A_i are sorted, then searching for the appearance of v_j in A_i can be done through *binary search* which takes $\Theta(\log |A_i|)$ time. In any case, adjacency matrix is better for fast querying.

- Listing adjacent (out) vertices of a vertex. Given v_i, one needs to tranverse the entire row (the i-th row) of the adjacency matrix to find all adjacent vertices of v_i which takes $\Theta(|V|)$ time. In case of adjacency list, one just needs to return the pointer to A_i which takes $\Theta(1)$ time. Hence, adjacency list is better in listing adjacent vertices.

In practice, adjacency list is usually the first choice, for an obvious reason that, for huge graphs, it is not possible to store an $|V| \times |V|$ matrix in memory.