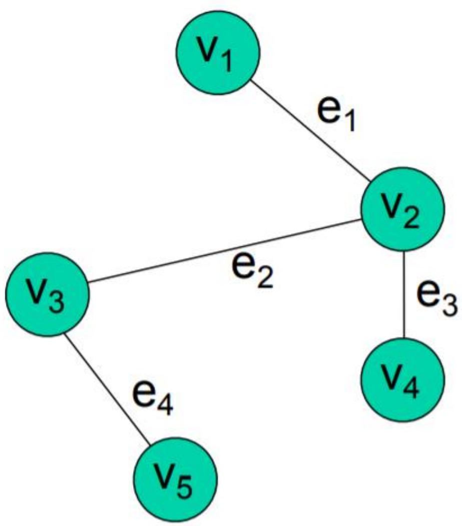
Graphs

Graphs

- A Graph G = (V, E) consists of a set of vertices (nodes) V and a set of edges E



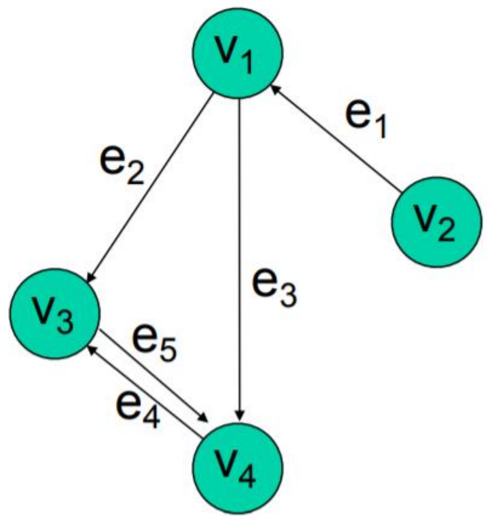
$G=(V,E)$
$V=\{v_1,v_2,v_3,v_4,v_5\}$
$E=\{e_1,e_2,e_3,e_4\}$
$e_1=(v_1,v_2)$
$e_2=(v_2,v_3)$
$e_3=(v_2,v_4)$
$e_4=(v_3,v_5)$

Uses of graphs

- Vertices are cities, edges are direct flights between cities
  - Want to find the best route between cities
- Vertices are classes, edges connect classes whose schedules overlap
  - Want to find feasible schedules for a student
- Vertices are objects in memory, edges connect objects that refer to each other
  - Want to know when an object can be freed

Directed graphs (“digraphs”)

- Edges have directions associated with them

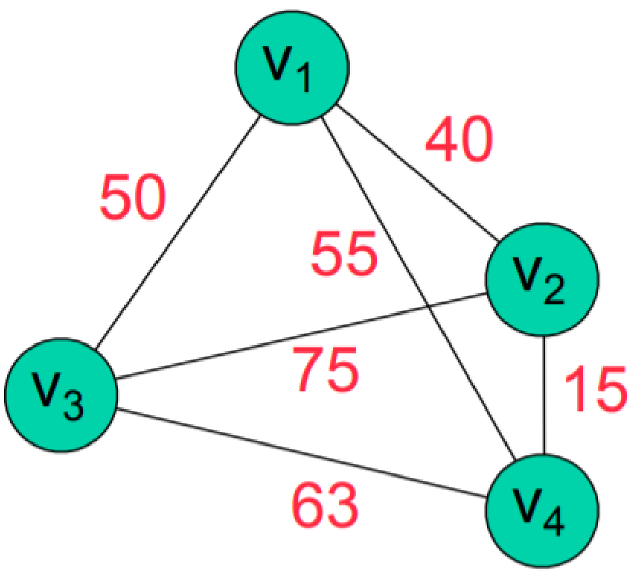


$G=(V,E)$
$V=\{v_1,v_2,v_3,v_4\}$
$E=\{e_1,e_2,e_3,e_4\}$
$e_1=(v_2,v_1)$
$e_2=(v_1,v_3)$
$e_3=(v_1,v_4)$
$e_4=(v_4,v_3)$
$e_5=(v_3,v_4)$

ordered pair
(predecessor, successor)

Weighted graphs

- There is a *weight* associated with each edge



$G=(V,E)$

$V=\{v_1, v_2, v_3, v_4\}$

$E=\{e_1, e_2, e_3, e_4, e_5\}$

....

....

Terminology (1)

- Vertex v_1 is *adjacent* to vertex v_2 if an edge connects v_1 and v_2
  - There exists an edge $e = (v_1, v_2) \in E$
- A *path* is a sequence of vertices in which each vertex is adjacent to the next one
  - *p* = $(v_1, \dots, v_n)$s.t. $(v_i, v_{i+1}) \in E$
  - The length of the path is the number of edges in it
- A cycle in a path is a sequence $(v_1, \dots, v_n)$ s.t. $(v_i, v_{i+1}) \in E$ and $(v_n, v_1) \in E$
- A graph with no cycles is an *acyclic graph*
- A *DAG* is a directed acyclic graph

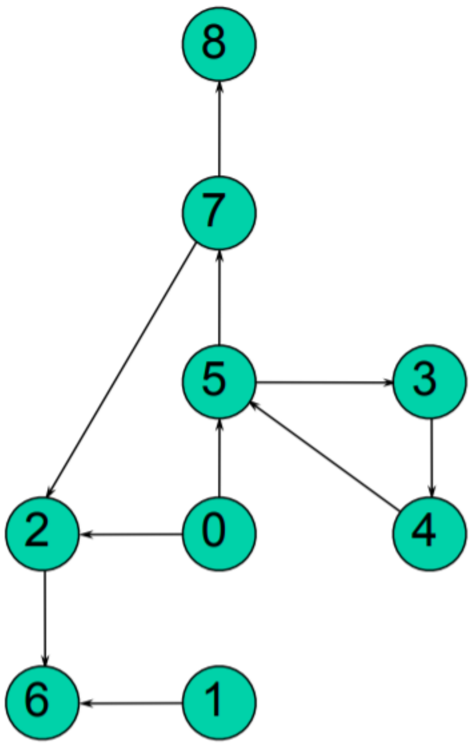
Terminology (2)

- A *simple path* is a path with no repetition of vertices
- A *simple cycle* is a cycle with no repetition of vertices
- Two vertices are *connected* is there is a path between them
- A subset of vertices is a connected component of G if each pair of vertices in the subset are connected.
- The *degree* of vertex v is the number of edges associated with v

Implementing the graph ADT

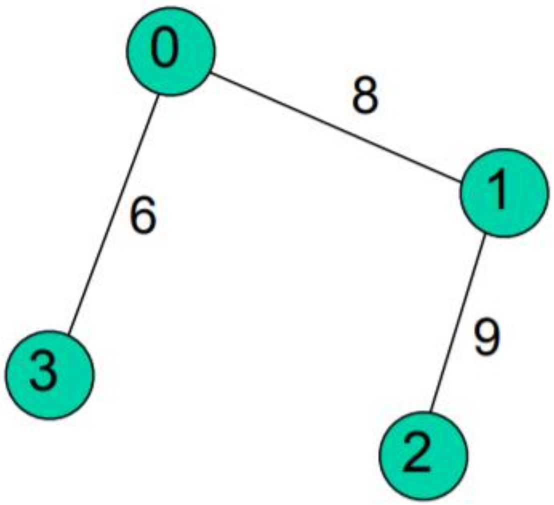
- Adjacency Matrix
  - An n x n matrix where M[i][j] =1 if there is an edge between v_i and v_j, and 0 otherwise
- Adjacency List
  - For $n = |V|$ vertices, n linked lists. The i-th linked list is a list of vertices adjacent to v_i.

Adjacency Matrix



	0	1	2	3	4	5	6	7	8
0	0	0	1	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	1	0	0
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	1	0	0	0
5	0	0	0	1	0	0	0	1	0
6	0	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0

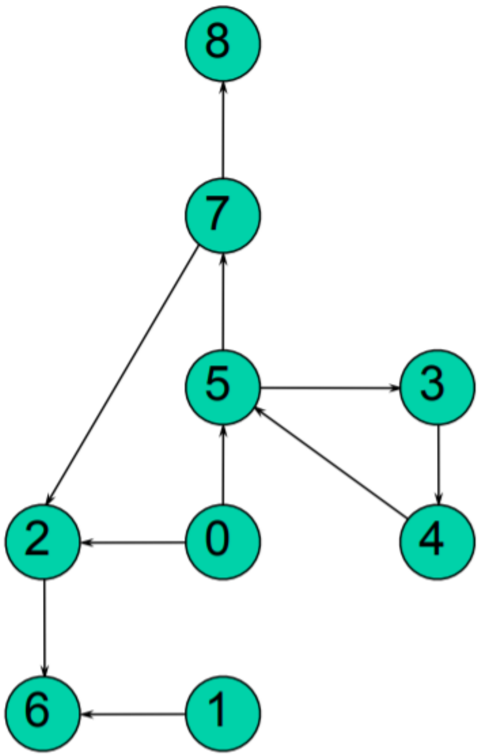
Adjacency Matrix

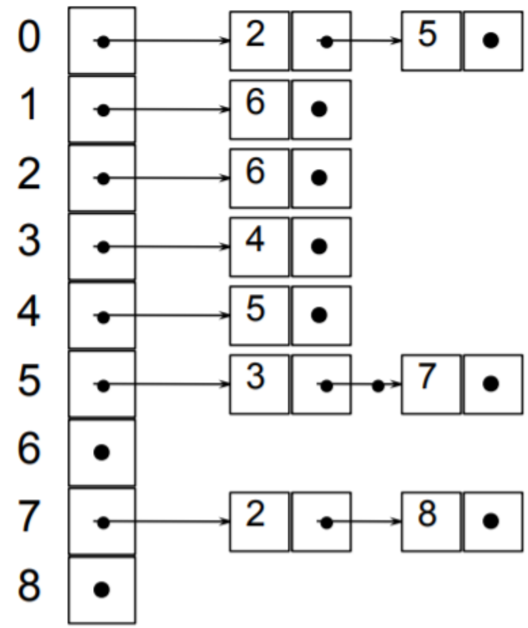


	0	1	2	3
0	∞	8	∞	6
1	8	∞	9	∞
2	∞	9	∞	∞
3	6	∞	∞	∞

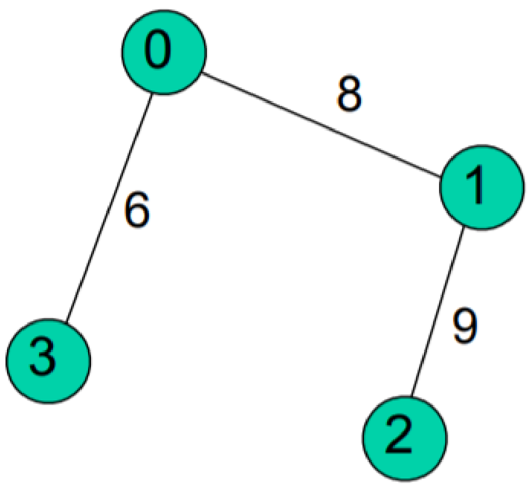
The matrix is symmetric for undirected graphs

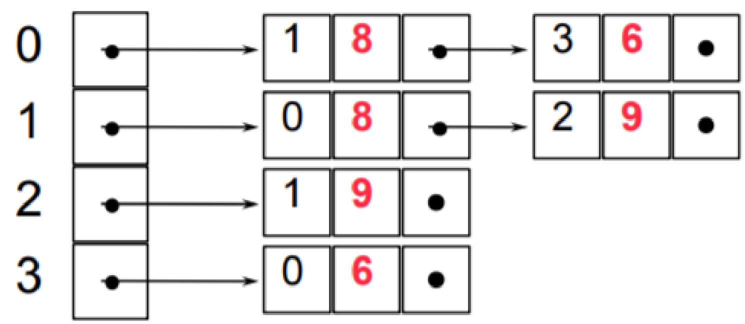
Adjacency List





Adjacency List





Complexity of operations

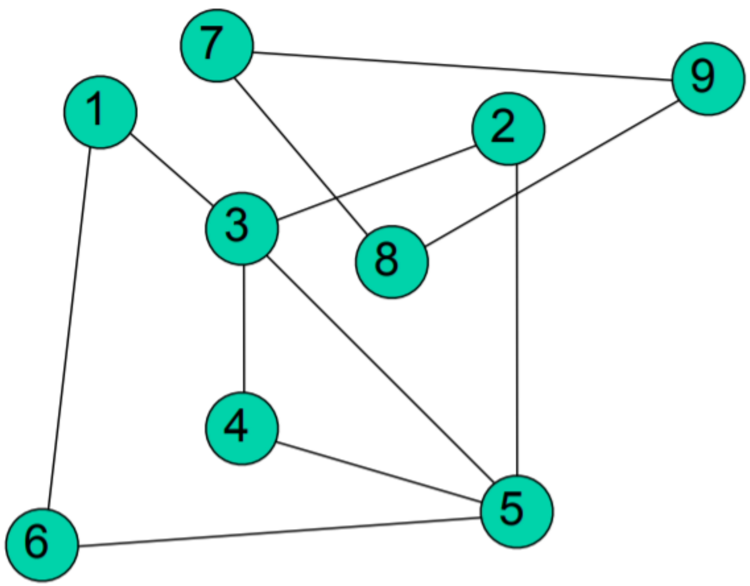
- Operations:
  - Is there an edge between v_i and v_j?
    - Adjacency Matrix: O(1)
    - Adjacency List: O(d)
      - d: the maximum degree in the graph
  - Find all vertices adjacent to v_i
    - Adjacency Matrix: O(|V|)
      - |V|: the number of vertices in the graph
    - Adjacency List: O(d)

Space requiements

- Adjacency Matrix: $O(|V|^2)$
  - Need to store $|V|^2$ matrix entries
- Adjacency list: $O(|V| + |E|)$
  - Need to store $|V|$ linked lists. Collectively, the linked list contain $|E|$ entries, so the space requirement is $a_1|V| + a_2|E|$, which is $O(|V| + |E|)$

Graph traversal

- Want to visit (e.g. in order to print) each vertex exactly once



	1	2	3	4	5	6	7	8	9
1			1			1			
2			1		1				
3	1	1		1	1				
4			1		1				
5		1	1	1		1			
6	1				1				
7								1	1
8							1		1
9							1	1	

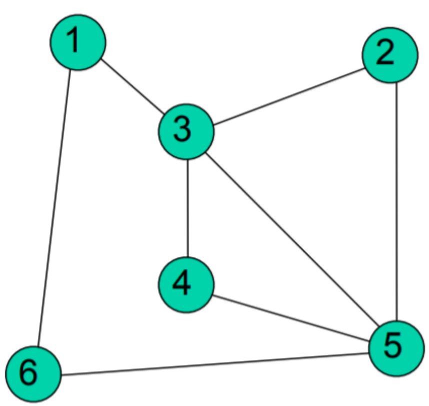
Graph traversal algorithm

```
while (there are non-visited nodes)
     Initialize data structure DS
     Add a non-visited vertex v_i to DS
     Mark v_i as visited
     while (DS is not empty)
       Remove v_j from DS
       Mark v_j as visited
       Add non-visited vertices adjacent to v_j to DS
```

Breadth-first traversal

- DS is a queue

while (there are non-visited nodes)
    Initialize data structure DS
    Add a non-visited vertex v_i to DS
    Mark v_i as visited
    while (DS is not empty)
      Remove v_j from DS
      Mark v_j as visited
      Add non-visited vertices adjacent to v_j to DS

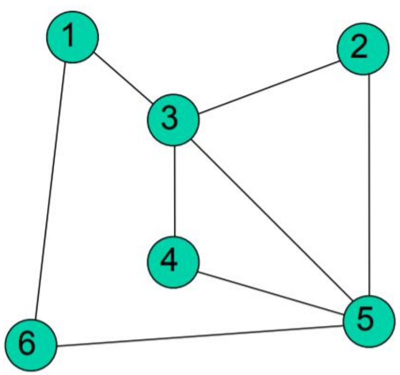


Queue contents:	Traversal:
1	
3 6	1
6 2 4 5	1 3
2 4 5	1 3 6
4 5	1 3 6 2
5	1 3 6 2 4
	$\boxed{1\ 3\ 6\ 2\ 4\ 5}$

Depth first traversal

- DS is a stack

```
while (there are non-visited nodes)
     Initialize data structure DS
     Add a non-visited vertex v_i to DS
     Mark v_i as visited
     while (DS is not empty)
       Remove v_j from DS
       Mark v_j as visited
       Add non-visited vertices adjacent to v_j to DS
```



Stack contents:

			4		
	6	5	2	2	
1	3	3	3	3	3

Traversal:

1	1	1	1	1	1
	6	6	6	6	6
		5	5	5	5
			4	4	4
				2	2
					3

Recursive Depth-First Traversal

```
DFS(v_i)
    Mark v_i as visited
    For each non-visited vertex v_j adjacent to v_i
        DFS(v_j)
```