

# 1 Graph (Section 11.1)

A graph  $G = (V, E)$  consists of a set of vertices  $V$  and a set of edges  $E$ . In an undirected graph, each edge is a set of 2 vertices  $\{u, v\}$ . In a directed graph, each edge  $e \in E$  is an ordered pair  $(u, v)$  for  $u, v \in V$ .

**Example 1.**  $G = (V, E)$  in Figure 1 is undirected, where  $V = \{1, 2, 3, 4\}$ , and  $E = \{\{1, 2\}; \{1, 3\}; \{2, 4\}\}$ : Note that  $\{1, 2\} = \{2, 1\}$ .

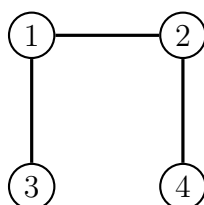


Figure 1: An Example Undirected Graph

**Example 2.** The graph  $G = (V, E)$  in Figure 2 is directed, where  $V = \{1, 2, 3, 4\}$ , and  $E = \{(1, 2); (2, 1); (3, 3); (4, 2)\}$ : Note that  $(1, 2) \neq (2, 1)$ .

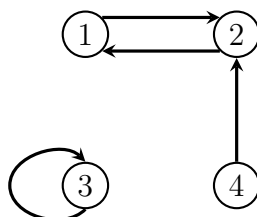


Figure 2: An Example Directed Graph

In an undirected graph, two vertices  $u, v \in V$  are adjacent if  $\{u, v\} \in E$ . The degree of a vertex  $v \in V$  is the number of edges containing  $v$ .

In a directed graph,  $v$  is adjacent to  $u$  if  $(u, v) \in E$ . We distinguish between in-degree: the number of edges leading to the vertex, and out-degree: the number of edges coming from the vertex.

A path in a graph  $G = (V, E)$  is a sequence of edges from the graph

$$e_1, e_2, \dots, e_{k-1}, e_k,$$

where  $e_i = \{v_i, v_{i+1}\}$ , and  $e_{i+1} = \{v_{i+1}, v_{i+2}\}$ .

**Example 3.** Given the following graph (Figure 3), two example paths in the graph are:

$$\{1, 6\}; \{6, 3\}; \{3, 4\}; \{4, 5\}$$

$$\{6, 5\}; \{5, 1\}; \{1, 5\}; \{5, 4\}$$

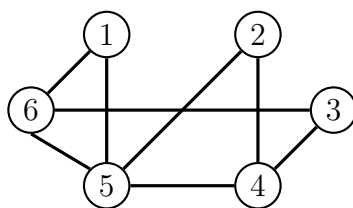


Figure 3: Another Example Undirected Graph

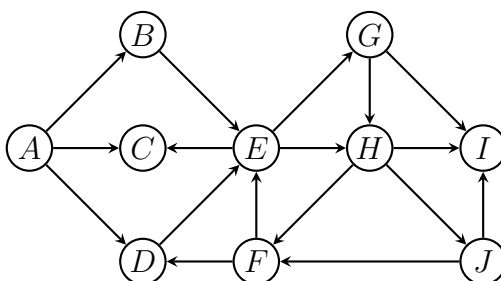
A path is simple if it contains no repeated vertex. A circuit is a closed path (with the same first and last vertex). A cycle is a circuit with no repeated edges or vertices (except the first and the last ones). A graph is acyclic if it contains no cycles. A graph is weighted if every edge  $e \in E$  is assigned a “weight/cost”, written as  $w(e) \in \mathcal{R}$ .

Major graph-related operations include:

- adding and removing vertices and edges;
- edge query: given  $u, v \in V$ , find out if edge  $\{u, v\}$ , or  $(u, v)$ , belongs to  $E$ ;
- neighbourhood( $v$ ): for  $v \in V$ , returns  $\{u \in V \mid \{u, v\} \in E\}$ ;
- degree( $v$ ): return  $S$ , size of neighbourhood. For directed graphs, we use in-degree( $v$ ), and out-degree( $v$ ) instead.

## 2 Graph Representation (Section 11.2)

Consider the following graph  $G$ :



Note that,

- $F$ ,  $I$ , and  $J$  are adjacent to  $H$ .  $H$  is adjacent to  $E$  and  $G$ .
- $E$  is reachable from  $A$  as there exists at least a path from  $A$  to  $E$ :  $[(A, B), (B, E)]$ , or  $[(A, D), (D, E)]$ .
- $A$  is not reachable for any other vertex in the graph.

We could store the graph simply as two lists (one for vertices, one for edges), but this is very inefficient and impractical. We use two standard data structures: adjacency matrix, and adjacency lists.

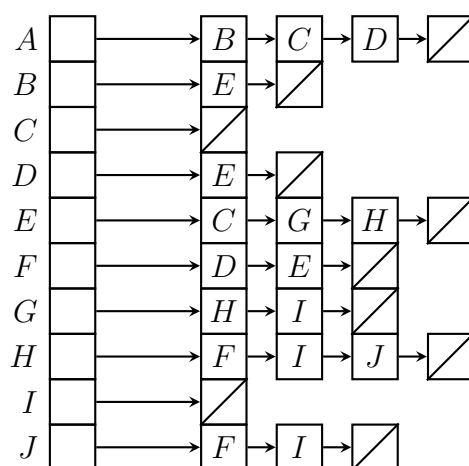
An adjacency matrix for a graph  $G$  is a 2-D array with the same number of rows and columns as  $G$  contains vertices. The entry at row  $i$  column  $j$  simply indicates whether or not there

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>
<i>A</i>		✓	✓	✓						
<i>B</i>					✓					
<i>C</i>										
<i>D</i>					✓					
<i>E</i>			✓				✓	✓		
<i>F</i>				✓	✓					
<i>G</i>								✓	✓	
<i>H</i>						✓			✓	✓
<i>I</i>										
<i>J</i>						✓			✓	

Table 1: An Adjacency Matrix for the Graph *G*

is an edge in *G* from *i* to *j*. For directed graphs (see Table 1 for an example), there are as many “checkmarks” as edges; For undirected graphs, every edge appears twice in the matrix. The Adjacency Matrix for the graph *G*:

An adjacency list stores a graph as a list of linked-lists. For each vertex *i* in the graph, we store a list of the vertices adjacent to *i*. Each list can be stored using either an array or a linked list (as pictured in Figure 4). For undirected graph, each edge  $\{u, v\}$  is stored twice (*v* is stored at position *u*, and *u* is stored at position *v*). The Adjacency List for the graph *G*:

Figure 4: An Adjacency List for the Graph *G*

### 3 Basic Search Algorithms (Section 11.3)

Given a graph  $G = (V, E)$  and a specific vertex  $v \in V$ , we want to traverse the graph, i.e., to “visit/reach” as many vertices as possible starting from *v* and moving along edges in *G* (e.g., for the graph above, let us say that vertex *A* is given as the start vertex).

We need to keep track of the vertices that have been reached already. We need an array of boolean values to indicate, for each vertex, whether or not its been reached. Deciding which

discovered vertex to reach next is important. We can apply one of the following two basic strategies:

- Breadth-First search (BFS) tries to reach vertices *as soon as possible*.
- Depth-First search (DFS) tries to go *as far as possible* before looking at alternatives.

To keep track of the order in which to reach vertices, we use a first-in-first-out Queue to maintain the set of discovered vertices in BFS; a first-in-last-out Stack to maintain the set of discovered vertices in DFS.

### 3.1 Breadth-first Search

The Breadth-First Search: Algorithm is given below:

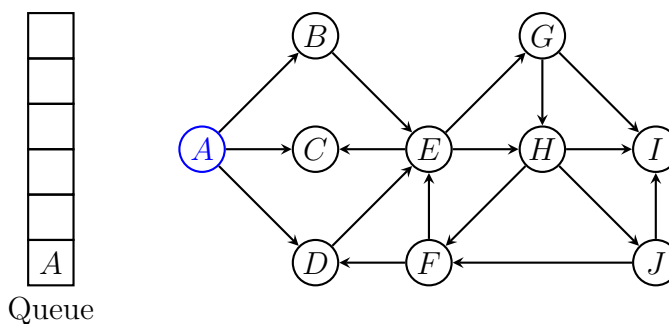
```

BFS(G, v) // search starts at vertex v.
  Queue Q = {} // start with an empty queue
  for each vertex u, set reached[u] = false;
  enqueue(Q v);
  while (Q is not empty)
    u = dequeue(Q);
    if (not reached[u])
      reached[u] = true;
      for each w adjacent to u and not reached
        enqueue(Q,w);
      end for
    end if
  end while
END

```

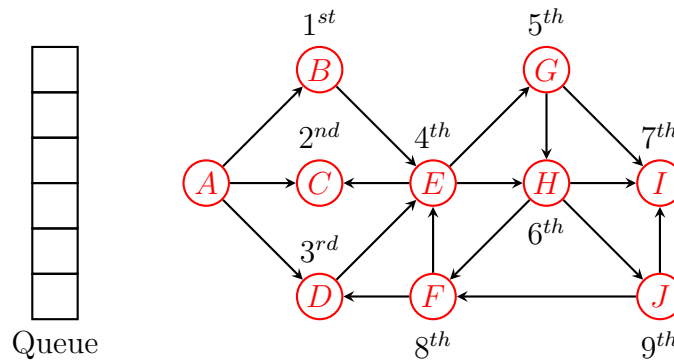
Initialization:

Vertex	A	B	C	D	E	F	G	H	I	J
Reached	X	X	X	X	X	X	X	X	X	X



After BFS Finished:

Vertex	A	B	C	D	E	F	G	H	I	J
Reached	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓



### 3.2 Depth-first Search

The Depth-First Search: Algorithm is given below:

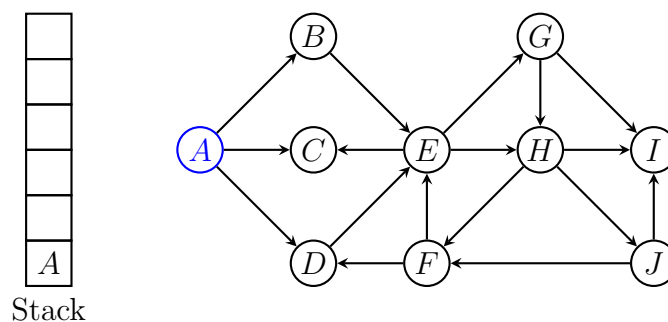
```

DFS(G, v) // search starts at vertex v.
  Stack Q = {} // start with an empty stack
  for each vertex u, set reached[u] = false;
  push(S v);
  while (S is not empty)
    u = pop(S);
    if (not reached[u])
      reached[u] = true;
      for each w adjacent to u and not reached
        push(S,w);
      end for
    end if
  end while
END

```

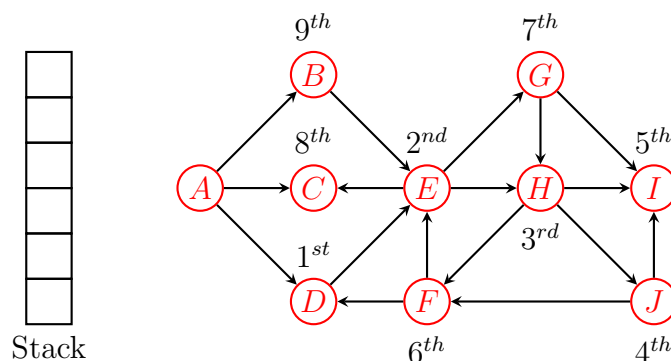
Initialization:

Vertex	A	B	C	D	E	F	G	H	I	J
Reached	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗



After DFS Finished:

Vertex	A	B	C	D	E	F	G	H	I	J
Reached	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓



### 3.3 Remarks

- BFS/DFS both run in linear time in the size of  $G$ .
- The edges that were used to visit a new vertex can be tracked in BFS/DFS:
  - For each vertex  $w$ , we use  $predecessor[w]$  that stores the vertex  $w$  was reached from.
  - If  $w$  is to be enqueued when reaching  $u$ , we set  $predecessor[w]$  to be equal to  $u$ .
- we can impose additional array of boolean values to distinguish discovered and undiscovered vertices. During the search, a vertex  $v$  can be in any one of the three states
  - *undiscovered*,  $v$  has not been touched at all.
  - *discovered*,  $v$  has been put in the queue.
  - *reached*,  $v$  has been taken out of the queue.