Directed and Undirected Graphs

- A **graph** is a mathematical structure consisting of a set of **vertices** and a set of **edges** connecting the vertices.

  Formally: $G = (V, E)$, where V is a set and $E \subseteq V \times V$.

- $G = (V, E)$ **undirected** if for all $v, w \in V$:

$$(v, w) \in E \iff (w, v) \in E.$$

  Otherwise **directed**.

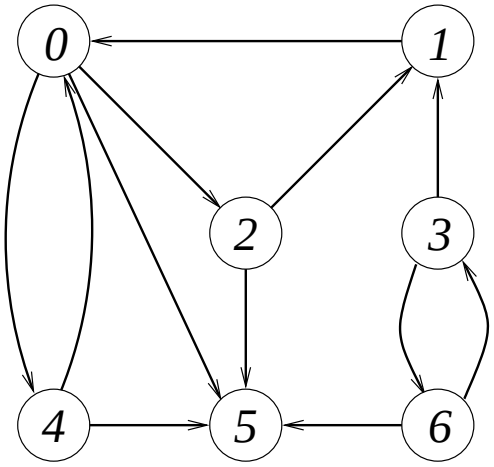
A directed graph

$G = (V, E)$ with vertex set

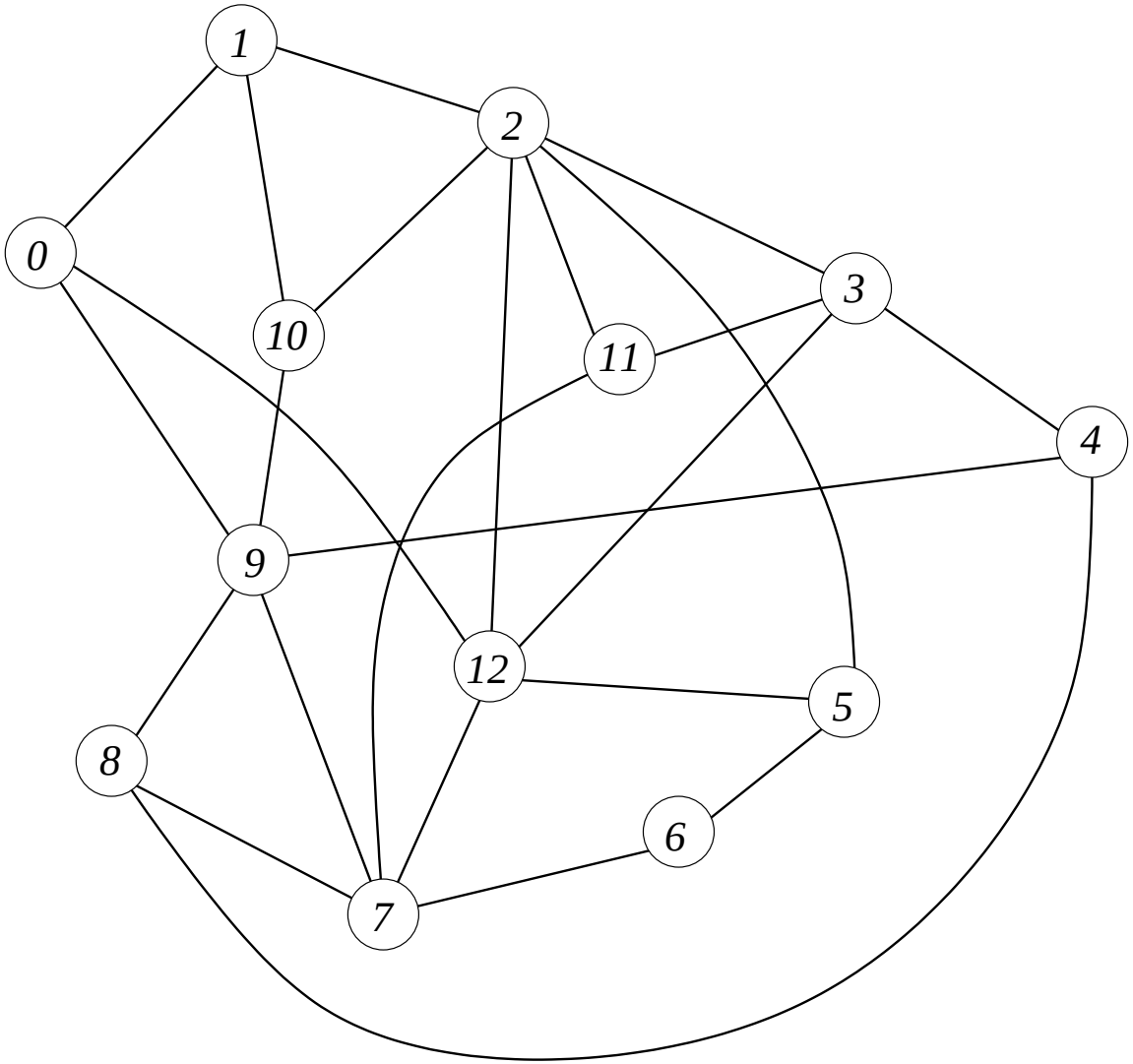
$$V = \{0, 1, 2, 3, 4, 5, 6\}$$

and edge set

$$E = \{(0,2), (0,4), (0,5), (1,0), (2,1), (2,5), (3,1), (3,6), (4,0), (4,5), (6,3), (6,5)\}.$$



An undirected graph



Examples of graphs in "real life"

Road Maps.

Edges represent streets and vertices represent crossings.

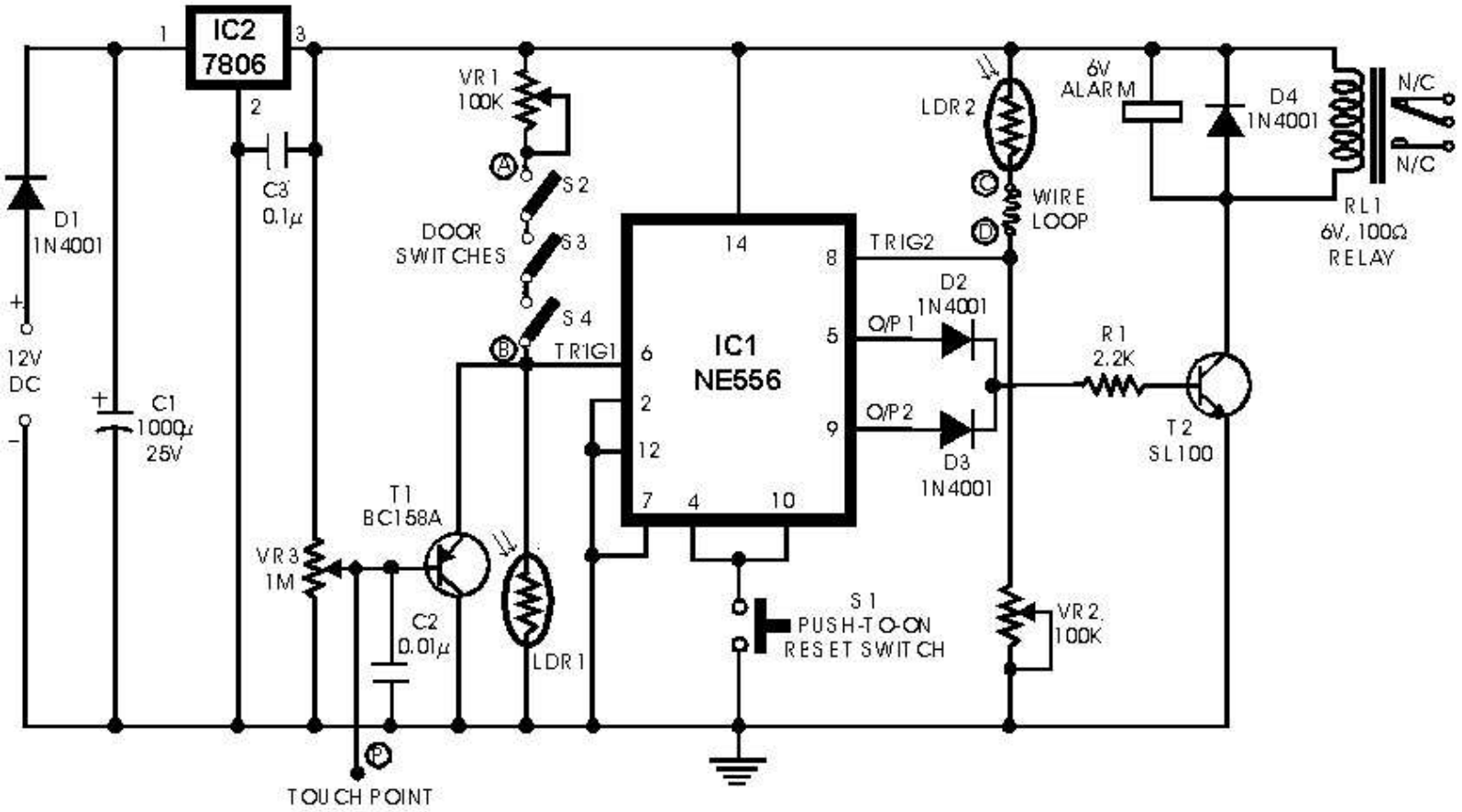
Examples (cont'd)

Airline route maps.
Vertices represent airports, and there is an edge from vertex A to vertex B if there is a direct flight from the airport represented by A to the airport represented by B.

Examples (cont'd)

Electrical Circuits.

Vertices represent diodes, transistors, capacitors, switches, etc., and edges represent wires connecting them.



Examples (cont'd)

Computer Networks.
Vertices represent computers and edges represent network connections (cables) between them.

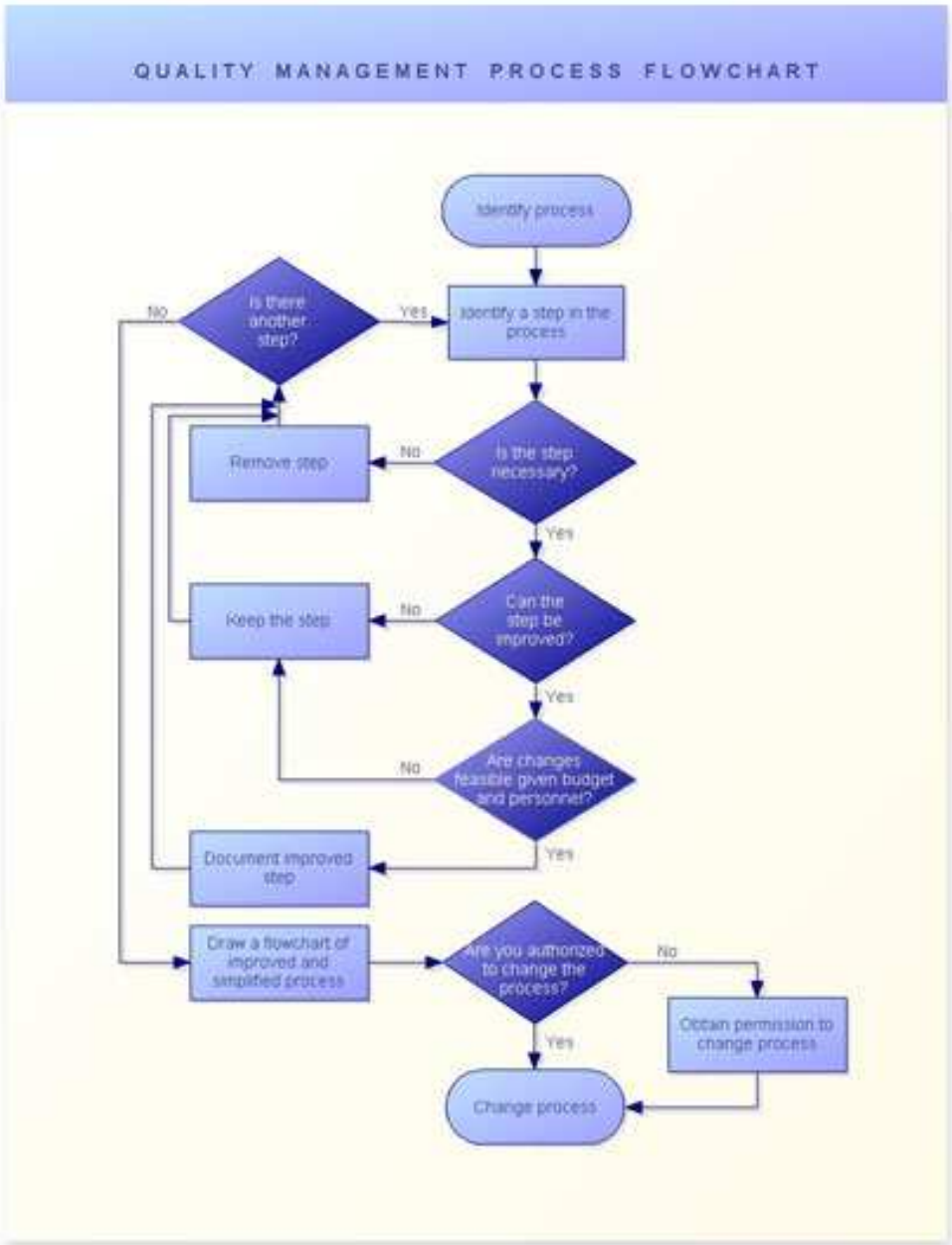
The World Wide Web.
Vertices represent webpages, and edges represent hyperlinks.

Examples (cont'd)

Flowcharts.
Vertices represent boxes and edges represent arrows.

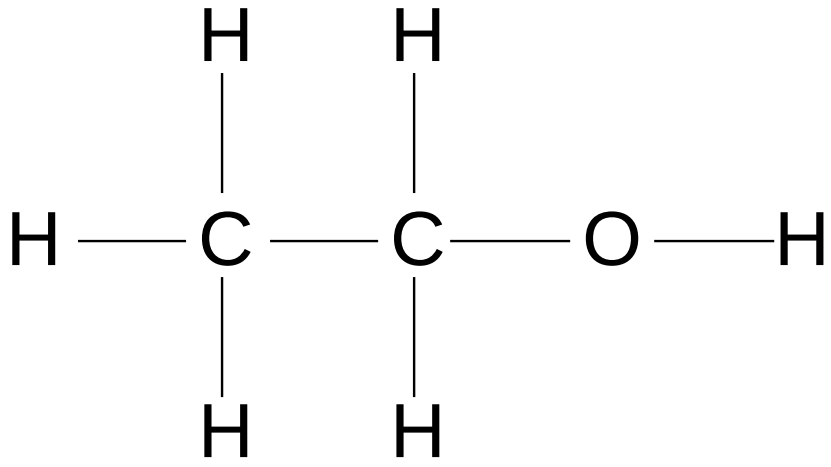
Conflict graphs in scheduling problems.
Example: Assignment of time slots to exams. No overlapping time slots for exams taken by the same students.

Modeled by graph whose vertices represent the exams, with an edge between two vertices if there is a student who has to take both exams.


QUALITY MANAGEMENT PROCESS FLOWCHART
Identify process
Is there another step?
No
Yes
Identify a step in the process
Is the step necessary?
No
Remove step
Yes
Can the step be improved?
No
Keep the step
Yes
Are changes feasible given budget and personnel?
No
Yes
Document improved step
Draw a flowchart of improved and simplified process
Are you authorized to change the process?
No
Obtain permission to change process
Yes
Change process

Examples (cont'd)

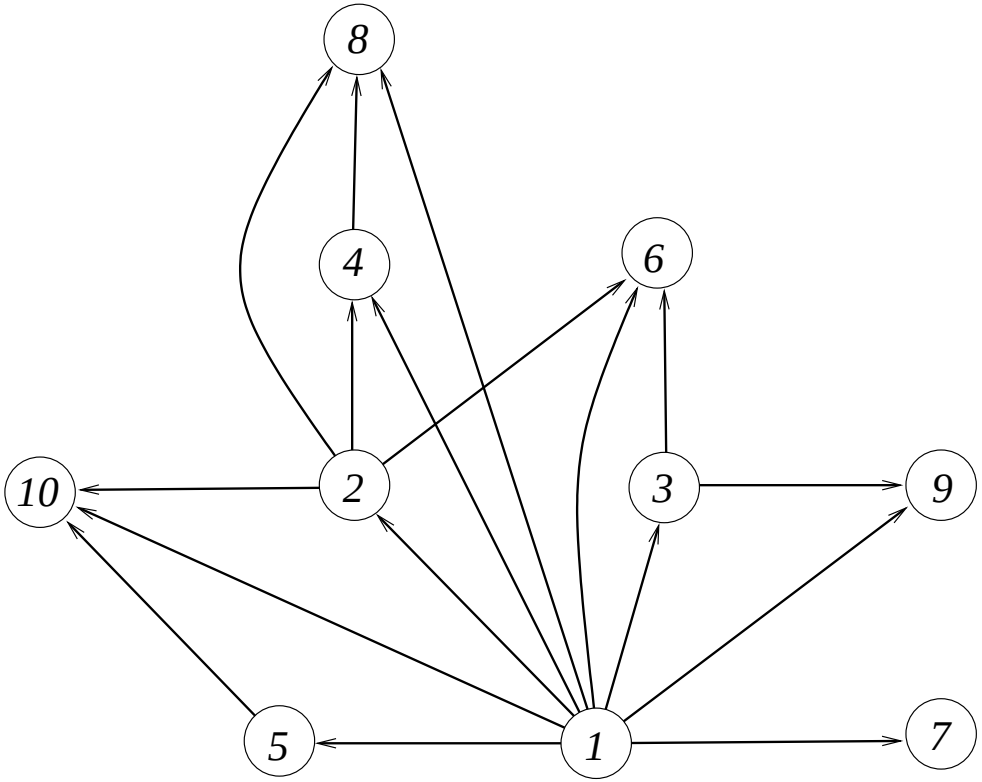
Molecules.
Vertices are atoms, edges are bonds between them.



Examples (cont'd)

Binary Relations in Mathematics.

For example, the 'is a proper divisor' relation on the integers.



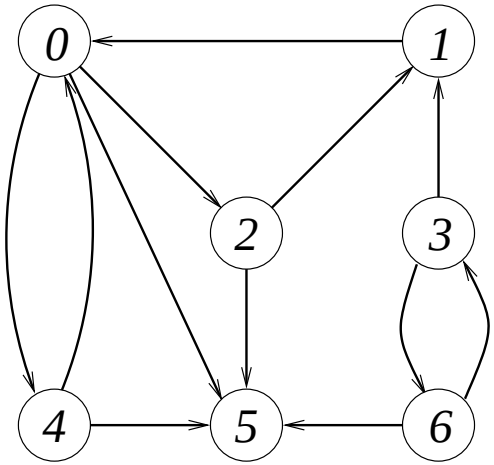
The adjacency matrix data structure

Let $G = (V, E)$ be a graph with n vertices. Vertices of G numbered $0, \ldots, n-1$.

The **adjacency matrix** of G is the $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ with

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j \\ 0 & \text{otherwise.} \end{cases}$$

Example

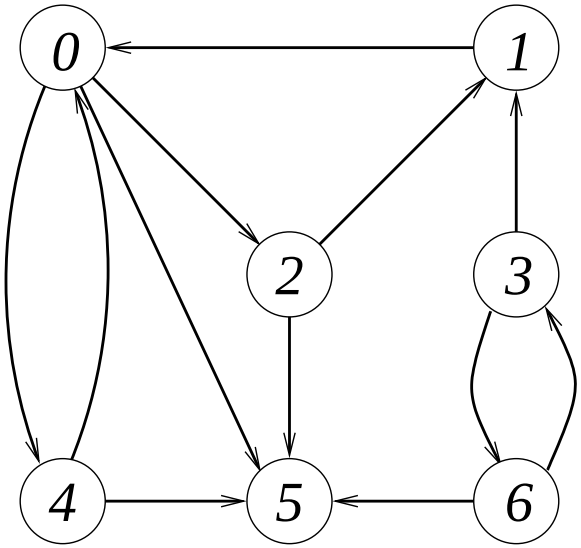


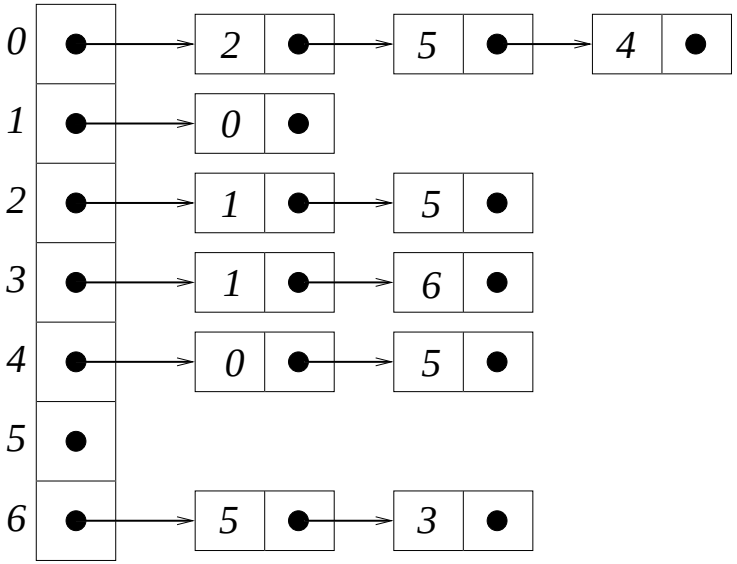
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

The adjacency list data structure

Array with one entry for each vertex v, which is a list of all vertices adjacent to v.

Example





Adjacency matrices vs adjacency lists

n = number of vertices, m = number of edges

	adjacency matrix	adjacency list
Space	$\Theta(n^2)$	$\Theta(n+m)$
Time to check if w adjacent to v	$\Theta(1)$	$\Theta(\text{out-degree}(v))$
Time to visit all w adjacent to v	$\Theta(n)$	$\Theta(\text{out-degree}(v))$
Time to visit all edges	$\Theta(n^2)$	$\Theta(n+m)$

Sparse and dense graphs

$G = (V, E)$ graph with n vertices and m edges

Observation: $m \leq n^2$

- G **dense** if m close to n^2
- G **sparse** if m much smaller than n^2

Graph traversals

A **traversal** is a strategy for visiting all vertices of a graph.

BFS $= breadth\text{-}first\ search$

DFS $= depth\text{-}first\ search$

General strategy:

1. Let v be an arbitrary vertex
2. Visit all vertices reachable from v
3. If there are vertices that have not been visited, let v be such a vertex and go back to (2)

BFS

Visit all vertices reachable from v in the following order:

- v
- all neighbours of v
- all neighbours of neighbours of v that have not been visited yet
- all neighbours of neighbours of neighbours of v that have not been visited yet
- etc.

BFS (cont'd)

Algorithm bfs(G)

1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. Initialise *Queue* Q
3. **for all** $v \in V$ **do**
4. **if** $visited[v] =$ FALSE **then**
5. **bfsFromVertex**(G, v)

BFS (cont'd)

Algorithm bfsFromVertex(G, v)

1. $visited[v] = \text{TRUE}$
2. Q.enqueue(v)
3. **while not** Q.isEmpty$()$ **do**
4. $\quad v \leftarrow Q$.dequeue$()$
5. $\quad$ **for all** w adjacent to v **do**
6. $\quad\quad$ **if** $visited[w] = \text{FALSE}$ **then**
7. $\quad\quad\quad visited[w] = \text{TRUE}$
8. $\quad\quad\quad Q$.enqueue(w)

DFS

Visit all vertices reachable from v in the following order:

- v
- some neighbor w of v that has not been visited yet
- some neighbor x of w that has not been visited yet
- etc., until the current vertex has no neighbour that has not been visited yet
- Backtrack to the first vertex that has a yet unvisited neighbour v'.
- Continue with v', a neighbour, a neighbour of the neighbour, etc., backtrack, etc.

DFS (cont'd)

Algorithm dfs(G)

1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. Initialise *Stack* S
3. **for all** $v \in V$ **do**
4. **if** $visited[v] =$ FALSE **then**
5. **dfsFromVertex**(G, v)

DFS (cont'd)

Algorithm dfsFromVertex(G, v)

1. S.push(v)
2. **while not** S.isEmpty() **do**
3. $\quad v \leftarrow S$.pop()
4. $\quad$ **if** $visited[v] =$ FALSE **then**
5. $\quad\quad visited[v] =$ TRUE
6. $\quad\quad$ **for all** w adjacent to v **do**
7. $\quad\quad\quad S$.push(w)