Graphs:

Set of vertices (or nodes) denoted by \( V \)

Set of edges between pairs of vertices denoted by \( E \)

Edges can be undirected so \( E \) is a set of unordered pairs,

\[ E = \{ \{1,2\}, \{1,5\}, \{3,4\}, \{4,2\} \} \]

it's symmetric; \( \forall i,j \in E \) iff \( \forall j,i \in E \)

unordered pair

OR Edges can be directed so \( E \) is a set of ordered pairs

\( E \) can denote by ordered set \( (i,j) \) or can denote as \( i \rightarrow j \)

The graph is denoted as \( G = (V,E) \)

and we use \( n = |V| = \# \) of vertices

\( m = |E| = \# \) of edges.
Representing graphs:

1) Adjacency matrix $A$:

An $n \times n$ matrix $A$ where $(i,j)$ entry is

$$A_{ij} = \begin{cases} 1 & \text{if edge } i \rightarrow j \\ 0 & \text{if no edge } i \rightarrow j \end{cases}$$

For undirected $G$, $A$ is symmetric:

$$A_{ij} = A_{ji}$$

Takes $O(n^2)$ space (even if $m$ is small)

Takes $O(1)$ time to check if there is an edge $i \rightarrow j$?

But takes $O(n)$ time to find all neighbors of a specific vertex.

2) Adjacency list:

Array of $n$ linked lists.

Linked list $i$ contains the neighbors of vertex $i$ (in any order).

Uses $O(n+m)$ space.

Takes $O(\text{degree}(v))$ time to find all neighbors of $v$ & $O(n+m)$ time to find all edges of $G$.

But $O(\text{degree}(v))$ time to check if edge $v \rightarrow w$. 
Exploring graphs:

DFS = depth first search
BFS = breadth first search.

DFS uses a stack = LIFO = last-in first-out

BFS uses a queue = FIFO = first-in first-out

Example:

DFS:
DFS: can implement a stack using recursion
Running DFS starting from a vertex $v$.

$\text{DFS}(G)$:
for all $w \in V$, set $\text{visited}(w) = \text{FALSE}$
$\text{Explore}(v)$

$\text{Explore}(w)$:
$\text{visited}(w) = \text{TRUE}$
for all $(w, z) \in E$:
if not $\text{visited}(z)$ then $\text{Explore}(z)$

$\text{Explore}(z)$: finds all vertices reachable from $z$
can use it to find connected components of undirected graphs.

DFS works on undirected & directed graphs,
assumes $G$ is given in adj. list. form,
takes $O(n+m)$ time.
For undirected $G$:

vertices $v$ & $w$ are connected if path between them.
Connected component = maximal set of connected vertices.

Example:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

3 components: \{A, B, E, I, J\}, \{F, G\}, \{C, D, G, H, K, L\}

Approach to find connected components:

1) Choose arbitrary start vertex $z$.
2) Run Explore($z$) to find component containing $z$.
3) Choose an unexplored $z$ & repeat.
**DFS(G):**

for all \( v \in V, \) \( \text{visited}(v) = \text{FALSE} \)

\( cc = 0 \)

for all \( v \in V, \)

if not \( \text{visited}(v) \) then:

\[
\text{for each } (v, w) \in E :
\]

if not \( \text{visited}(w) \) then \( \text{Explore}(w) \)

\( \text{Explore}(z) : \)

\( \text{visited}(z) = \text{TRUE} \)

\( ccnum(z) = cc \)

for each \( (z, w) \in E : \)

if not \( \text{visited}(w) \) then \( \text{Explore}(w) \)

**Running time is** \( O(n+m), \) \( n=|V| \)

\( m=|E| \)

for \( G \) in adj. list. representation.
For directed graphs to solve connectivity, need more information from DFS. Add a clock, keep track of preorder # and postorder #

\[ \text{Pre}(v) = \text{time start exploring } v \]
\[ \text{Post}(v) = \text{time finish exploring all neighbors of } v \]

\[ \text{DFS}(G) : \]
\hspace{1cm} for all \( v \in V \), \( \text{visited}(v) = \text{FALSE} \)
\hspace{1cm} \text{clock} = 1
\hspace{1cm} for all \( v \in V \),
\hspace{1.5cm} if not visited \( (v) \) then Explore \( (v) \)

\[ \text{Explore}(z) : \]
\hspace{1cm} \text{visited}(z) = \text{TRUE}
\hspace{1cm} \text{Pre}(z) = \text{clock}
\hspace{1cm} \text{clock}++
\hspace{1cm} for each \( (z, w) \in E \):
\hspace{2cm} if not visited \( (w) \) then Explore \( (w) \)
\hspace{1cm} \text{Post}(z) = \text{clock}
\hspace{1.5cm} \text{clock}++
Example:

Run DFS on above graph starting at B
explored edges = tree edges

Pre ↑
1→6 ← post ↓

2→11

3→10

4→7

8→9

5→6

13→15

13→14
Dashed edges = non-tree edges = unexplored edges

3 types of non-tree edges:

Forward: Example: D → G, vertex → descendant

\[ \text{Pre} (D) < \text{Pre} (G) < \text{post} (G) < \text{post} (D) \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td>(G)</td>
</tr>
</tbody>
</table>

Back: E → A, F → B, vertex → ancestor

\[ \text{Pre} (B) < \text{Pre} (F) < \text{post} (F) < \text{post} (B) \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td>(E)</td>
</tr>
</tbody>
</table>

Cross: F → H, H → G

\[ \text{Pre} (H) < \text{post} (H) < \text{Pre} (F) < \text{post} (F) \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(H)</td>
<td>(F)</td>
</tr>
</tbody>
</table>

Tree edge: B → A, vertex → child

\[ \text{Pre} (B) < \text{Pre} (A) < \text{post} (A) < \text{post} (B) \] (Same form as forward edge)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td>(A)</td>
</tr>
</tbody>
</table>
Property: Directed $G$ has a cycle iff its DFS tree contains a back edge.

Proof:

$(\Leftarrow)$ Consider a back edge, say $e = u \rightarrow v$.
So $u$ is a descendant of $v$ in the DFS tree.
In the DFS tree, there is a path $P$ from $v$ to $u$. Take this path $P$ plus $e$
this gives a cycle $C = P \cup e$.

$(\Rightarrow)$ Consider a cycle $C$ in $G$.
Some vertex of $C$ is visited first, say $v$.
Rest of $C$ is reachable from $v$ so is in $v$'s subtree.
One of these other vertices in $C - v$ has an edge to $v$, and this will be a back edge.

So if no cycles then no back edges.

For edge $z \rightarrow w$,
if back edge, $\text{post}(z) < \text{post}(w)$
if cross, forward, or tree edge,
$\text{post}(w) < \text{post}(z)$.

So can detect if $G$ has a cycle by checking Post #’s to see if we ever
find a back edge.
A DAG = directed acyclic graph
up
no cycles
hence no back edges

Topologically sorting a DAG
= order the vertices so that all
edges go left to right
(lower) (higher)

Easy to do:
Run DFS &
sort by decreasing Post #

highest Post # \[\rightarrow\] lowest Post #

Since no back edges, for every edge \(Z \rightarrow W\)
we have \(Post(W) < Post(Z)\)
Example:

```
Example:

A ---- C ---- E
|        |        |
|        |        |
B ---- D
```

Run DFS:

```
Run DFS:

|                   |
|                   |
E[3,4,15] ---- F[5,6]
```

Topological sorting:

```
Topological sorting:

B ---- D ---- A ---- C ---- E ---- E
```

(Note, there are other topological orderings)