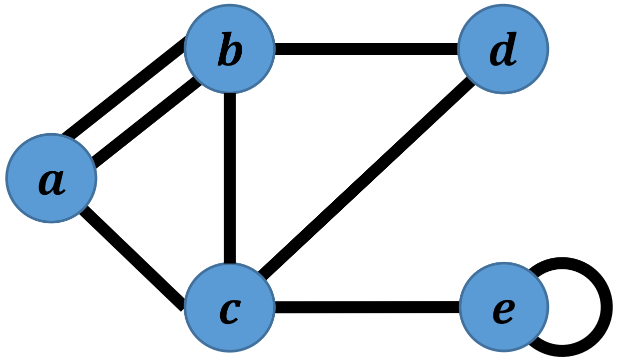


Lecture 2: Graph Review

CSC 226: Algorithms and Data Structures II

JAN 13

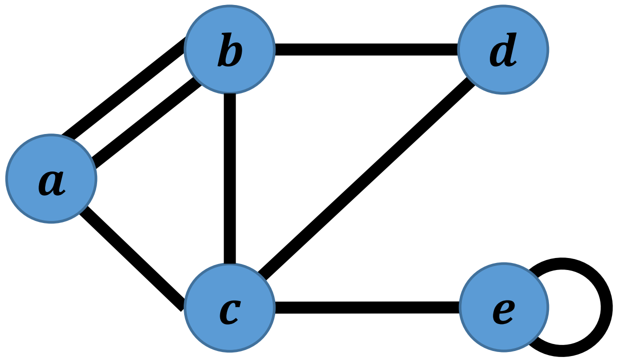
University of Victoria

What is a graph?

A **graph** $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ consists of

- a set $\boldsymbol{V}$ of **vertices** (nodes)
- a collection $\boldsymbol{E}$ of pairs of vertices from $\boldsymbol{V}$ called **edges** (arcs)

Example of a graph:



$$\boldsymbol{V} = \{a, b, c, d, e\}$$

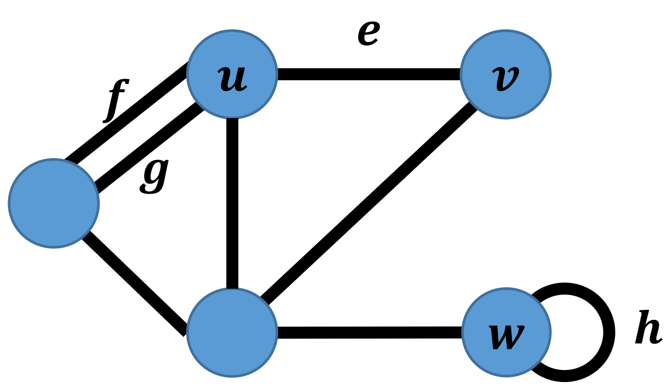
$$\boldsymbol{E} = \left\{ \begin{array}{c} \{a,b\}, \{a,b\}, \{a,c\}, \\ \{b,c\}, \{b,d\}, \\ \{c,d\}, \{c,e\}, \\ \{e,e\} \end{array} \right\}$$

Undirected Edges

An **undirected edge** $\boldsymbol{e}$ represents a **symmetric** relationship between vertices $\boldsymbol{u}$ and $\boldsymbol{v}$

- We write $\boldsymbol{e} = \{\boldsymbol{u}, \boldsymbol{v}\}$, where $\{\boldsymbol{u}, \boldsymbol{v}\}$ is an unordered pair
- $\boldsymbol{u}$ and $\boldsymbol{v}$ are the **endpoints** of the edge
- $\boldsymbol{u}$ is **adjacent** to $\boldsymbol{v}$ & v is adjacent to u
- $\boldsymbol{e}$ is **incident** to $\boldsymbol{u}$ and $\boldsymbol{v}$
- The **degree** of a vertex is the number of incident edges

  e.g. $\mathbf{deg}(\boldsymbol{u}) = 4$, $\mathbf{deg}(\boldsymbol{v}) = 2$



- **Parallel edges** (multi-edges): more than one edge between pairs of vertices

  e.g. $\boldsymbol{f}$ and $\boldsymbol{g}$
- **Self-loop**: edge that connects a vertex to itself

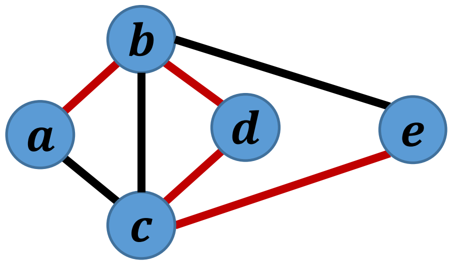
  e.g. $\boldsymbol{h}$ is a self loop. Note that $\mathbf{deg}(\boldsymbol{w}) = 3$. Self-loop adds two degrees to a node.
- Usually, we denote the number of vertices by $\boldsymbol{n}$ and the number of edges by $\boldsymbol{m}$

  nodes

Undirected Paths

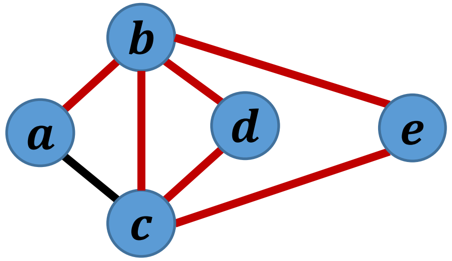
A **walk** in a graph is a sequence of vertices $\boldsymbol{v_1, v_2, \ldots, v_k}$ such that there exist edges $\boldsymbol{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}}$ aka a connection between v_1 & v_k.

- The **length** of a walk is the number of edges
- If $v_1 = v_k$, the walk is **closed**. Otherwise, it is **open**.



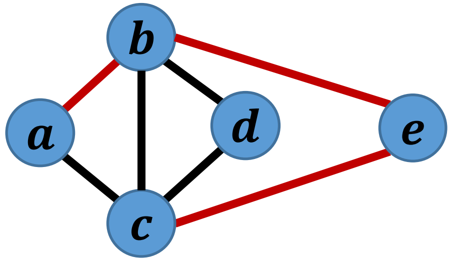
$\boldsymbol{a, b, d, c, e}$

- If no edge is repeated, it is a **trail**.
- A closed trail is a **circuit**.



$\boldsymbol{a, b, c, d, b, e, c}$

- If no vertex is repeated, it's a **path**.
- A **cycle** is a path with the same start and end vertices



$\boldsymbol{a, b, e, c}$

Walk: A connection between two nodes V_1 and V_k.

Closed Walk: Starting and ending node is the same.

Open Walk: Starting and ending nodes are different.

Trail: No repeated edge.

Circuit: A closed trail. No repeated edge and starting node = ending node.

Path: No repeated vertex.

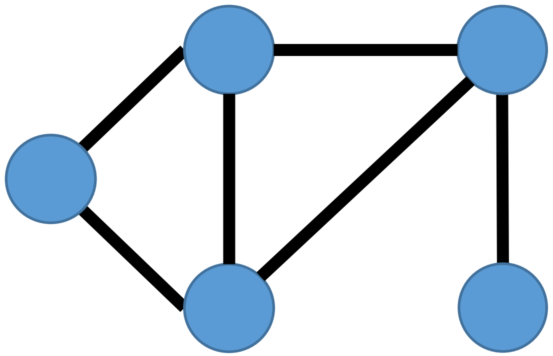
Cycle: A closed path. No repeated vertex except starting node = ending node.

Connected Graphs

A graph is **connected** if every pair of vertices is connected by a path

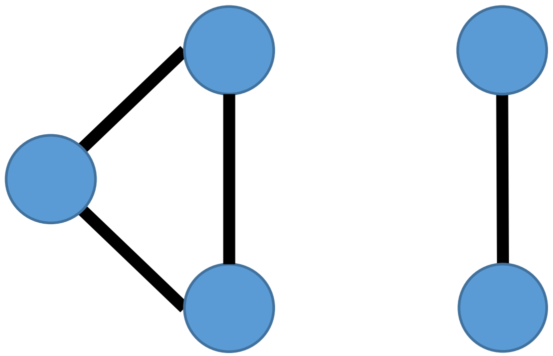
Example:

- **Connected** graph



Example:

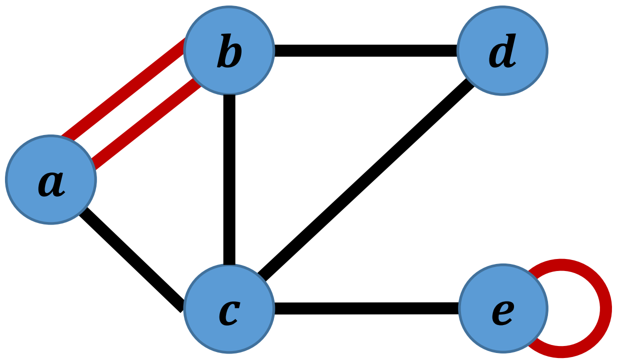
- **Disconnected** graph
- Two **connected components**



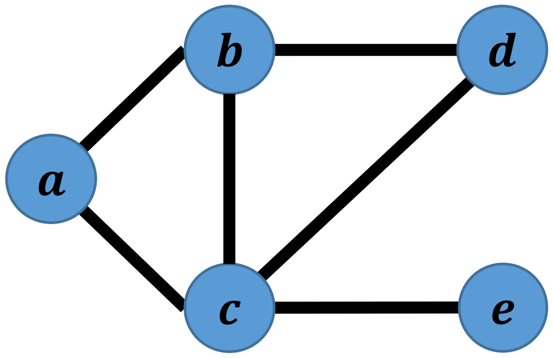
Simple Graphs

A **simple graph** is a graph with **no self-loops and no parallel / multi-edges**

Not a simple graph :



Simple graph:



For the most part, we will only be working with simple graphs in this course.

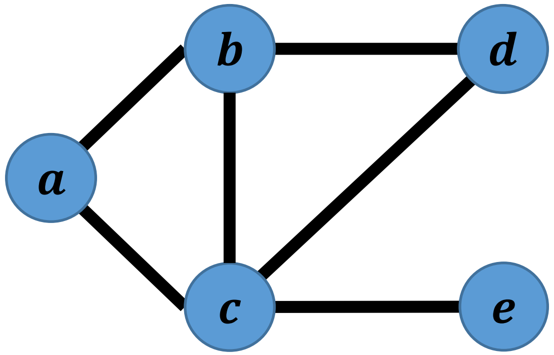
A lot of algorithms only work on simple graphs.

Simple Graphs

A **simple graph** is a graph with **no self-loops and no parallel / multi-edges**

Theorem: If $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a graph with $\boldsymbol{m}$ edges, then

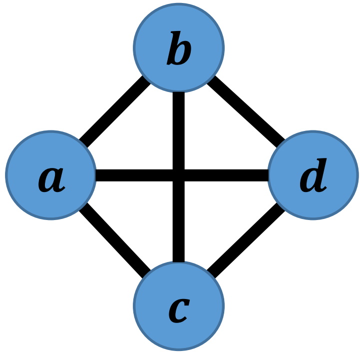
$$\sum_{v \in V} \deg(v) = 2m$$



Theorem: Let $\boldsymbol{G}$ be a simple graph with $\boldsymbol{n}$ vertices and $\boldsymbol{m}$ edges. Then,

Max No. of edges = $\frac{n(n-1)}{2}$ since each node can be connected to all other nodes and this goes on. If $n=5$, $5+4+3+2+1$.

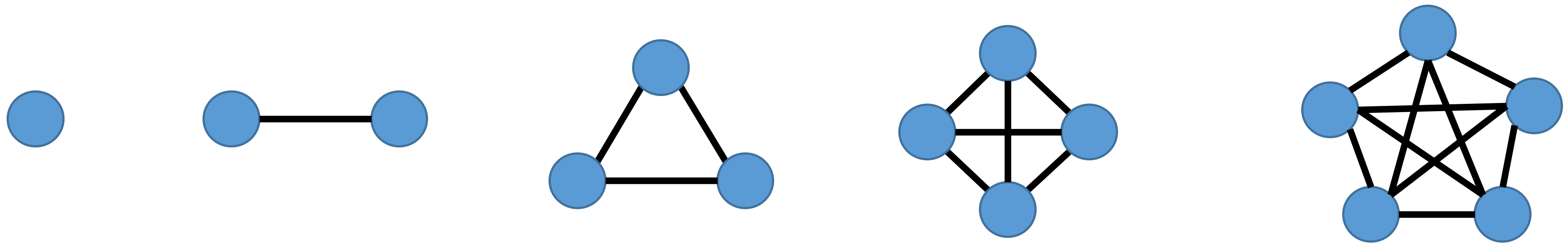
$$m \leq \frac{n\,(n-1)}{2}$$



Corollary: A simple graph with $\boldsymbol{n}$ vertices has $\boldsymbol{O}(\boldsymbol{n}^2)$ edges.

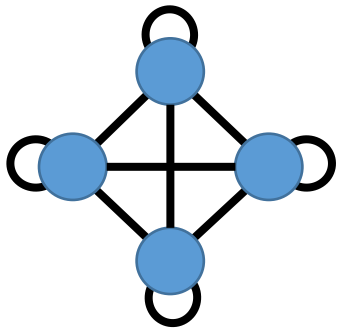
Complete Graphs

A **complete graph** is a simple graph where every pair of vertices is connected by an edge



- The complete graph on $\boldsymbol{n}$ vertices has exactly $\frac{\boldsymbol{n(n-1)}}{\boldsymbol{2}}$ edges

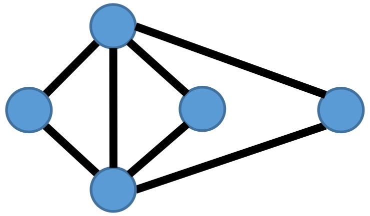
- The complete graph on $\boldsymbol{n}$ vertices with one self-loop per vertex has exactly $\frac{\boldsymbol{n(n+1)}}{\boldsymbol{2}}$ edges



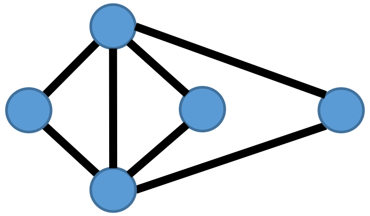
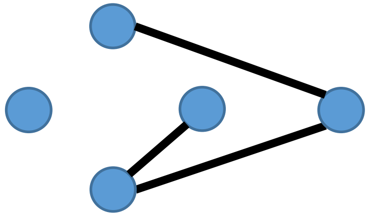
Subgraphs

A **subgraph** of $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a graph $\boldsymbol{G'} = (\boldsymbol{V'}, \boldsymbol{E'})$ where

- $\boldsymbol{V'}$ is a subset of $\boldsymbol{V}$
- $\boldsymbol{E'}$ consists of edges $\{\boldsymbol{u}, \boldsymbol{v}\}$ in $\boldsymbol{E}$ such that both $\boldsymbol{u}$ and $\boldsymbol{v}$ are in $\boldsymbol{V'}$

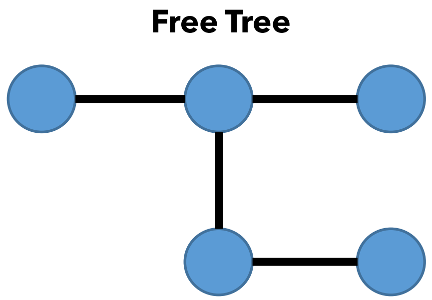
$\boldsymbol{G} =$  $\boldsymbol{G'} =$ 

A **spanning subgraph** $\boldsymbol{G'}$ of $\boldsymbol{G}$ contains all vertices of $\boldsymbol{G}$

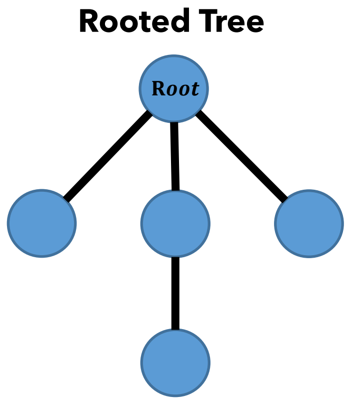
$\boldsymbol{G} =$  $\boldsymbol{G'} =$ 

Trees and Forests

A **(free) tree** is an undirected graph T such that
- T is connected
- T has no cycles

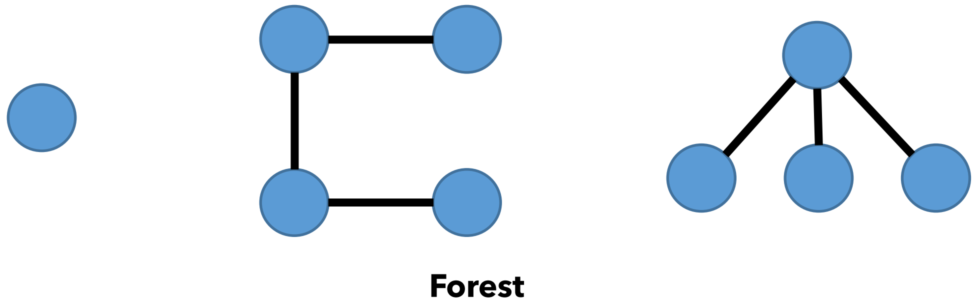


A **rooted tree** is a tree where one vertex is designated as the root



Cycle : A closed path. No repeated vertex except starting node = ending node.

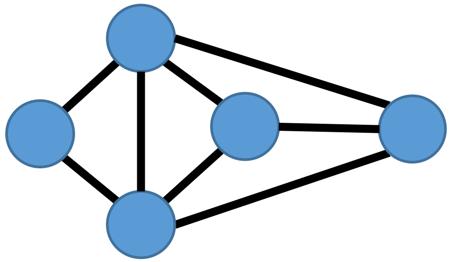
A **forest** is an undirected graph without cycles (collection of disconnected trees)
- The connected components of a forest are trees



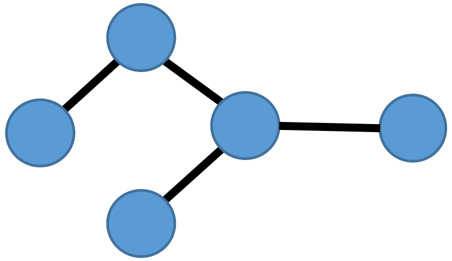
Spanning Trees and Forests

A **spanning tree** of a connected graph is a spanning subgraph that is a tree

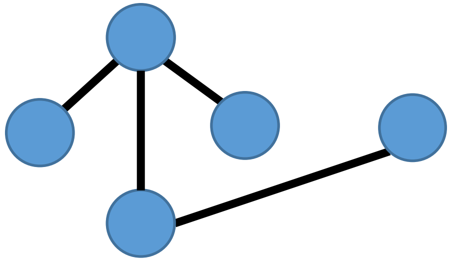
- A spanning tree is not unique unless the graph is a tree



Graph

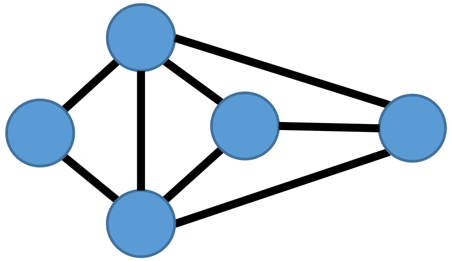


Spanning Tree

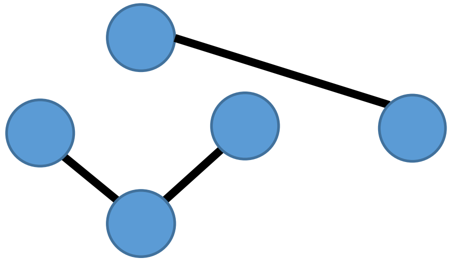


Spanning Tree

A **spanning forest** of a graph is a spanning subgraph that is a forest



Graph



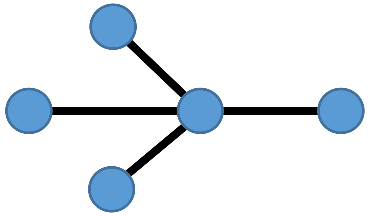
Spanning Forest

Properties of Graphs, Trees, and Forests

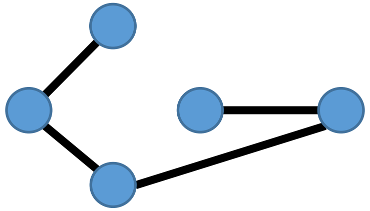
Theorem: Let G be an undirected simple graph with n vertices and m edges. Then have the following:

- If G is **connected**, then $m \geq n - 1$. But if $m = n-1$, does not guarantee it is connected.

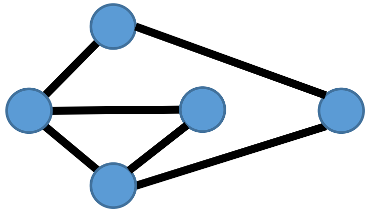
$n = 5$
$m = 4$



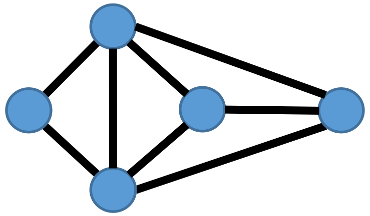
$n = 5$
$m = 4$



$n = 5$
$m = 6$

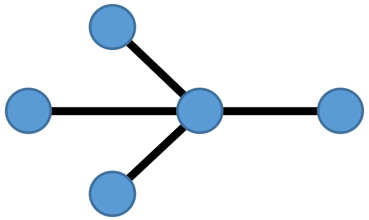


$n = 5$
$m = 8$

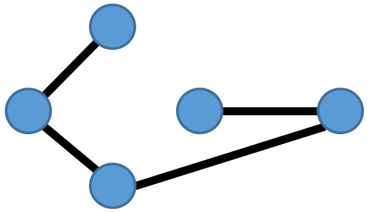


- If G is a **tree**, then $m = n - 1$

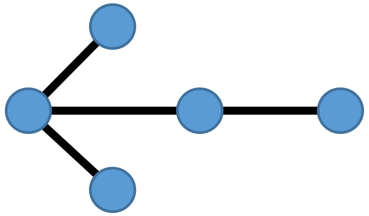
$n = 5$
$m = 4$



$n = 5$
$m = 4$

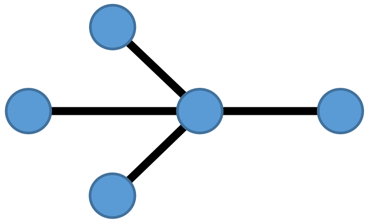


$n = 5$
$m = 4$

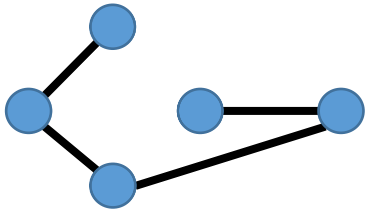


- If G is a **forest**, then $m \leq n - 1$

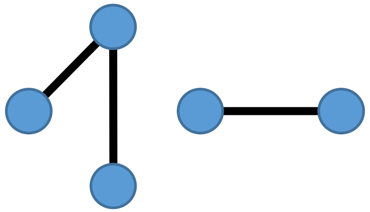
$n = 5$
$m = 4$



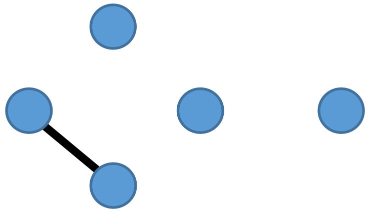
$n = 5$
$m = 4$



$n = 5$
$m = 3$



$n = 5$
$m = 1$



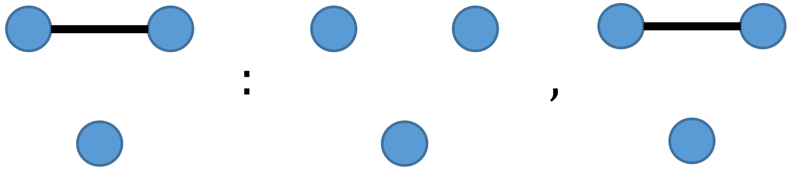
Number of Possible Spanning Subgraphs

How many possible **spanning subgraphs** are there for a given graph? $= 2^n$

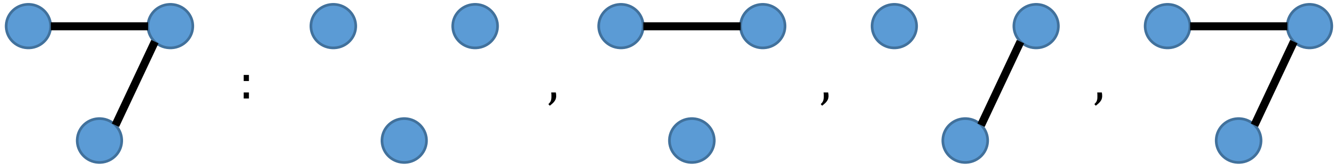
- Since all vertices must be included in the subgraph, we only have a "choice" about which edges to include

$G =$ 

$m = 1$
$2^1 = 2$ possibilities

$G =$ 

$m = 1$
$2^1 = 2$ possibilities

$G =$ 

$m = 2$
$2^2 = 4$ possibilities

Number of Possible Subgraphs

How many possible **subgraphs** are there for a given graph?

- We choose which vertices to include **and** which edges to include between those vertices

$G =$ 

:

$\binom{2}{0} = \mathbf{1}$ way to pick $\mathbf{0}$ vertices out of $\boldsymbol{n} = \mathbf{2}$ vertices

- No edges included

$\binom{2}{1} = \mathbf{2}$ ways to pick $\mathbf{1}$ vertex out of $\boldsymbol{n} = \mathbf{2}$ vertices 

- No edges included

$\binom{2}{2} = \mathbf{1}$ ways to pick $\mathbf{2}$ vertices out of $\boldsymbol{n} = \mathbf{2}$ vertices

- For each way to pick 2 vertices (only one way), need to also consider which edges to include between those vertices

- Each edge (which are between the picked vertices) can be **in** or **not**

$\mathbf{1}$ subgraph

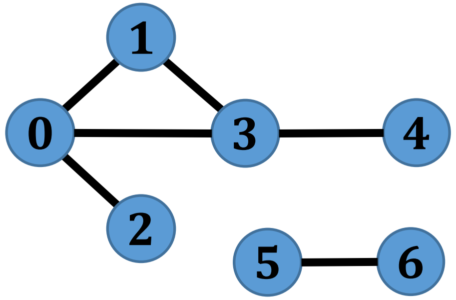
$\mathbf{2}$ subgraphs

$\mathbf{2}$ subgraphs

$\mathbf{5}$ possible subgraphs

Graph Representation: Set of Edges

Maintain a list of edges (array or linked list)



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

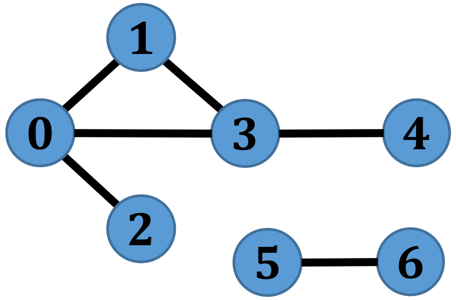
Each edge pair is only stored once.

- Also have to store a separate list of vertices since some vertices have no edges
- Simple representation, but can be inefficient (e.g. traversal algorithms)
- To find all the vertices adjacent to a vertex, need to look through the entire list of edges ($\boldsymbol{O}(\boldsymbol{m})$)
  - E.g. DFS will take $\mathbf{O}(\boldsymbol{n} \cdot \boldsymbol{m})$ time with this representation

Graph Representation: Adjacency Matrix

Maintain a **2-dimensional** $\boldsymbol{n} \times \boldsymbol{n}$ boolean array

- For each edge $\{\boldsymbol{u}, \boldsymbol{v}\}$, $\mathbf{adj}[\boldsymbol{u}][\boldsymbol{v}] = \mathbf{adj}[\boldsymbol{v}][\boldsymbol{u}] = \mathbf{true}\ (\mathbf{1})$

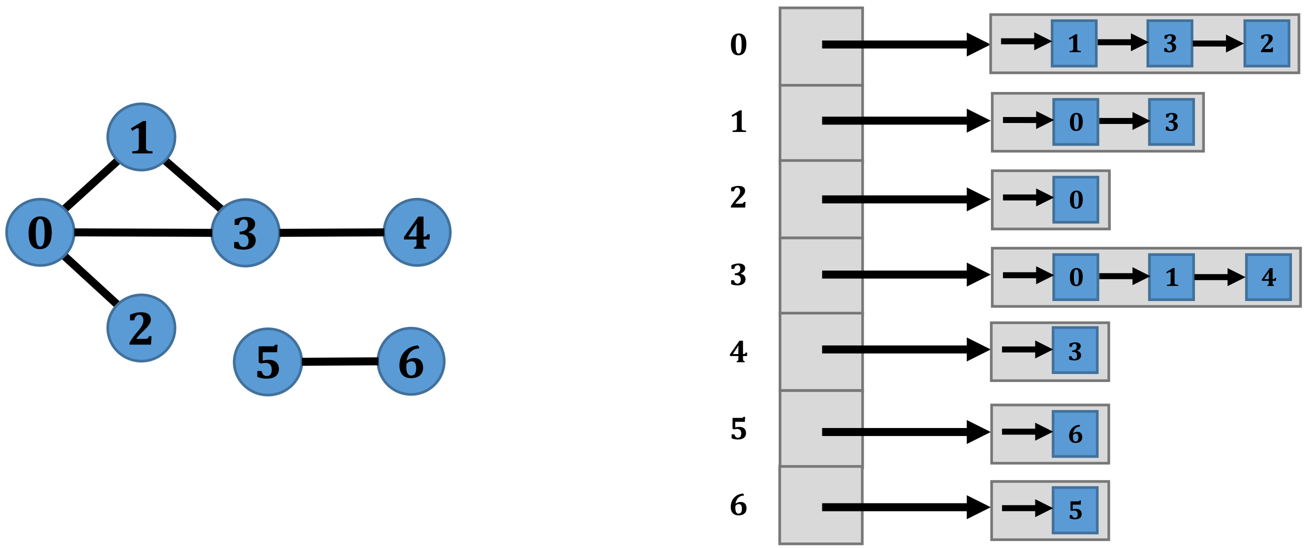


	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	0	0	0	0
3	1	1	0	0	1	0	0
4	0	0	0	1	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	1	0

- To find the vertices adjacent to a vertex, we need to look at a row in the matrix ($\boldsymbol{O}(\boldsymbol{n})$)
  - Slightly more efficient than the edge list representation since there could be many more edges than there are vertices
  - e.g. DFS will take $\boldsymbol{O}(\boldsymbol{n}^2)$ time with this representation
- Best representation for querying if two vertices are adjacent ($\boldsymbol{O}(\mathbf{1})$)
- Requires a lot of storage space ($\boldsymbol{O}(\boldsymbol{n}^2)$), but good representation for dense graphs (many edges)

Graph Representation: Adjacency List

Maintain an array indexed by vertices which points to a list of adjacent vertices



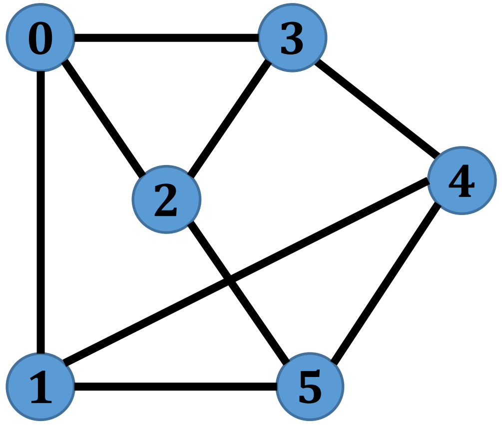
Coz we might have just nodes, so O(n+m) & not O(m)

- Commonly used representation since it is the most efficient for most graphs
- If the graph is sparse, the space required ($\boldsymbol{O}(\boldsymbol{n}+\boldsymbol{m})$) is strictly less than the adjacency matrix representation
- To find the vertices adjacent to a vertex $\boldsymbol{v}$, only takes $\boldsymbol{O}(\deg(\boldsymbol{v}))$ time
  - Faster than both previous representations
  - Adjacency lists are best for algorithms which involve finding all adjacent vertices
- DFS will take $\boldsymbol{O}(\boldsymbol{n}+\boldsymbol{m})$ time with this representation

Depth First Search (DFS)

$\boldsymbol{DFS}\ (\boldsymbol{G}, \boldsymbol{u})$**:**
Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$
Output: Labeling of edges in the connected component as discovery edges and back edges

Label $\boldsymbol{u}$ as explored
for each edge $\boldsymbol{e}$ incident to $\boldsymbol{u}$ **do**
    if $\boldsymbol{e}$ is unexplored **then**
        $\boldsymbol{v} \leftarrow$ other endpoint of $\boldsymbol{e}$
        if $\boldsymbol{v}$ is unexplored **then**
            Label $\boldsymbol{e}$ as an explored *discovery* edge
            $\boldsymbol{DFS}(\boldsymbol{G}, \boldsymbol{v})$
        else
            Label $\boldsymbol{e}$ as an explored *back* edge



Depth First Search (DFS)

$\boldsymbol{DFS}\ (\boldsymbol{G}, \boldsymbol{u})$**:**

Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery edges and back edges

Label $\boldsymbol{u}$ as explored

for each edge $\boldsymbol{e}$ incident to $\boldsymbol{u}$ **do**

  if $\boldsymbol{e}$ is unexplored **then**

    $\boldsymbol{v} \leftarrow$ other endpoint of $\boldsymbol{e}$

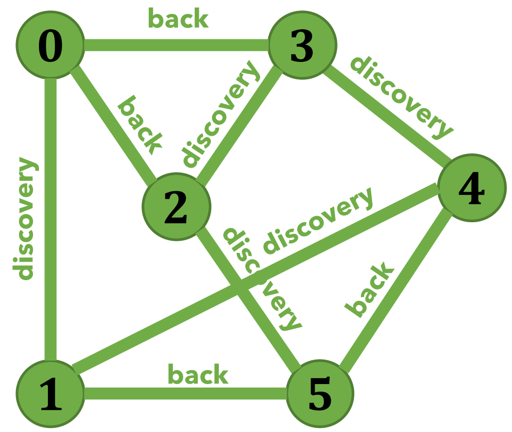
    if $\boldsymbol{v}$ is unexplored **then**

      Label $\boldsymbol{e}$ as an explored *discovery* edge

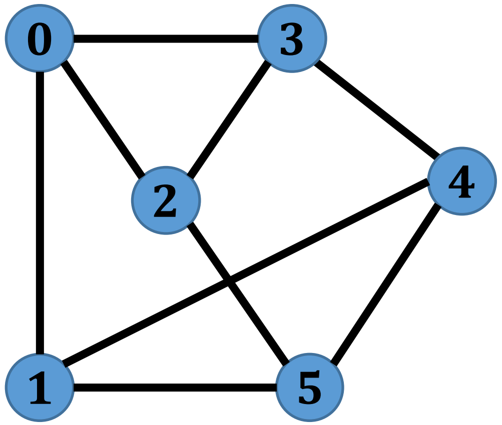
      $\boldsymbol{DFS}(\boldsymbol{G}, \boldsymbol{v})$

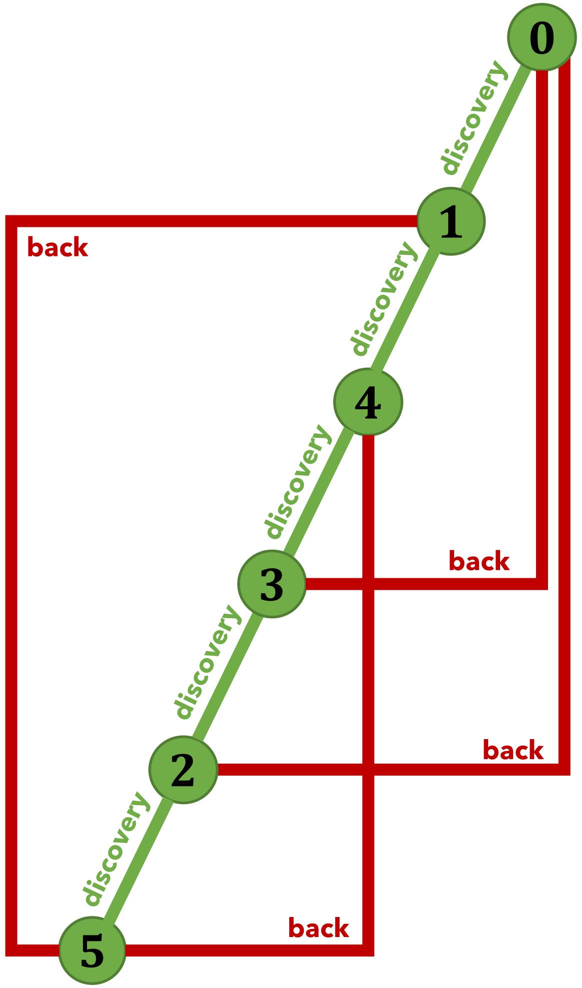
    else

      Label $\boldsymbol{e}$ as an explored *back* edge



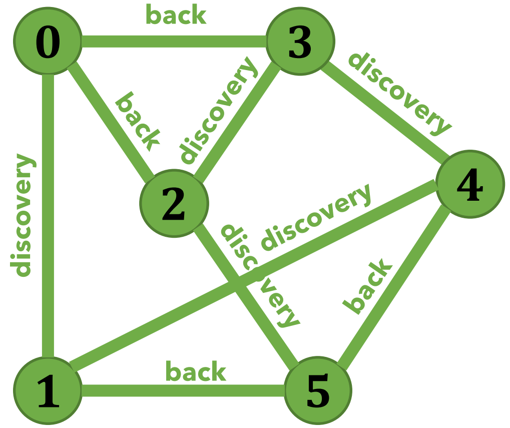
DFS Tree





Discovery edges lead to unvisited nodes in the traversal and form a spanning tree

Back edges go from nodes to one of its ancestors in the traversal discovery spanning tree



Breadth First Search (BFS)

$\boldsymbol{BFS}\ (\boldsymbol{G}, \boldsymbol{u})$**:**

Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery edges and cross edges

$\boldsymbol{Q} \leftarrow$ new empty queue

Label $\boldsymbol{u}$ as explored

$\boldsymbol{Q}$.enqueue($\boldsymbol{u}$)

while $\boldsymbol{Q}$ is not empty **do**

  $\boldsymbol{u} \leftarrow \boldsymbol{Q}$.dequeue()

  for each edge $\boldsymbol{e} = \{\boldsymbol{u}, \boldsymbol{v}\}$ incident to $\boldsymbol{u}$ **do**

    if $\boldsymbol{e}$ is unexplored **then**

      if $\boldsymbol{v}$ is unexplored **then**

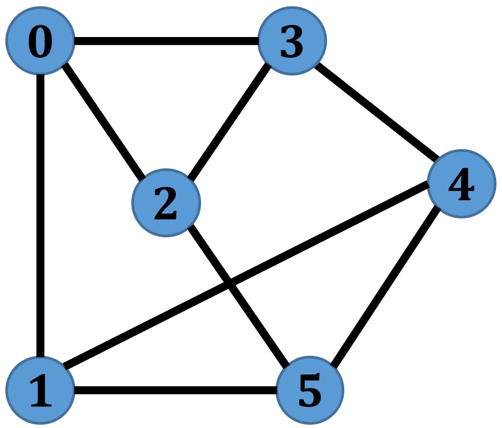
        Label $\boldsymbol{e}$ as an explored *discovery* edge

        Mark $\boldsymbol{v}$ as explored

        $\boldsymbol{Q}$.enqueue($\boldsymbol{v}$)

      else

        Label $\boldsymbol{e}$ as an explored *cross* edge



Breadth First Search (BFS)

$\boldsymbol{BFS}\ (\boldsymbol{G}, \boldsymbol{u})$**:**

Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery edges and cross edges

$\boldsymbol{Q} \leftarrow$ new empty queue

Label $\boldsymbol{u}$ as explored

$\boldsymbol{Q}$.enqueue($\boldsymbol{u}$)

while $\boldsymbol{Q}$ is not empty **do**

    $\boldsymbol{u} \leftarrow \boldsymbol{Q}$.dequeue()

    for each edge $\boldsymbol{e} = \{\boldsymbol{u}, \boldsymbol{v}\}$ incident to $\boldsymbol{u}$ **do**

        if $\boldsymbol{e}$ is unexplored **then**

            if $\boldsymbol{v}$ is unexplored **then**

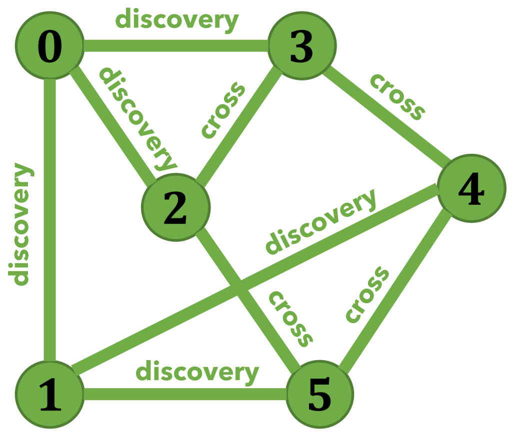
                Label $\boldsymbol{e}$ as an explored *discovery* edge

                Mark $\boldsymbol{v}$ as explored

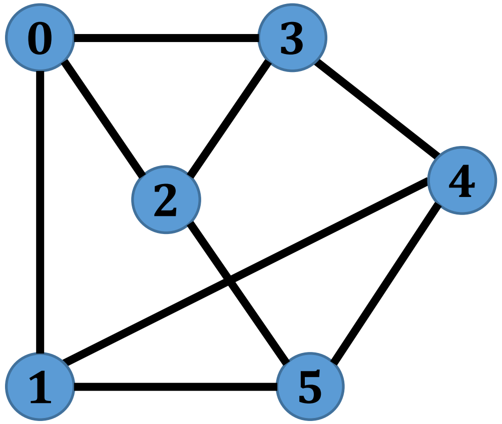
                $\boldsymbol{Q}$.enqueue($\boldsymbol{v}$)

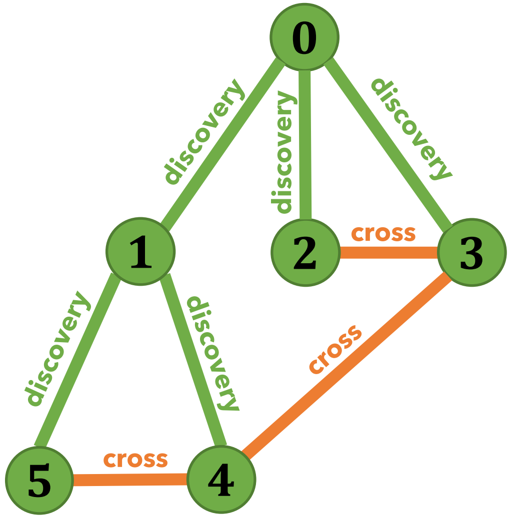
            else

                Label $\boldsymbol{e}$ as an explored *cross* edge



BFS Tree





Discovery edges lead to unvisited nodes in the traversal and form a spanning tree

Cross edges connect two nodes which do not have any ancestor and descendant relationship in the traversal discovery spanning tree

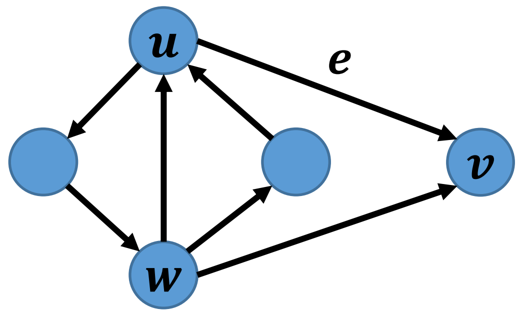
Directed Graphs (Digraphs)

A **directed graph (digraph)** is a graph whose edges are all directed

- Can implement undirected graphs using directed graphs
- Applications include one-way streets, flights, task scheduling

A **directed edge** $\boldsymbol{e}$ represents an **asymmetric** relationship between two vertices $\boldsymbol{u}$ and $\boldsymbol{v}$

- We write $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$ is an ordered pair
- $\boldsymbol{u}$ and $\boldsymbol{v}$ are the **endpoints** of the edge
- $\boldsymbol{u}$ is **adjacent** to $\boldsymbol{v}$ and vice versa
- $\boldsymbol{e}$ is **incident** to $\boldsymbol{u}$ and $\boldsymbol{v}$
- $\boldsymbol{u}$ is the **source vertex** and $\boldsymbol{v}$ is the **destination vertex**

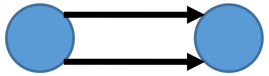


- The **indegree** of a vertex is the number of incoming edges ($\mathbf{indeg}(\boldsymbol{w}) = 1$)
- The **outdegree** of a vertex is the number of outgoing edges ($\mathbf{outdeg}(\boldsymbol{w}) = 3$)

Simple Digraphs

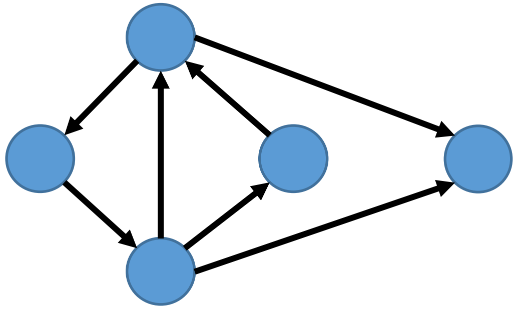
A **simple digraph** is a graph with **no self-loops and no parallel / multi-edges**

- Note that parallel edges in digraphs refer to edges pointing in the same direction



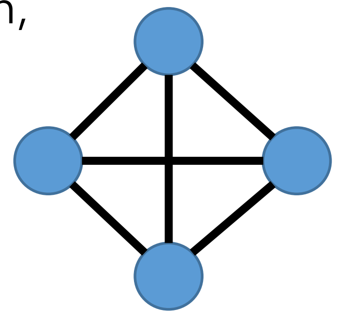
Theorem: If $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a digraph with $\boldsymbol{m}$ edges, then

$$\sum_{\boldsymbol{v} \in \boldsymbol{V}} \mathrm{indeg}(\boldsymbol{v}) = \sum_{\boldsymbol{v} \in \boldsymbol{V}} \mathrm{outdeg}(\boldsymbol{v}) = \boldsymbol{m}$$

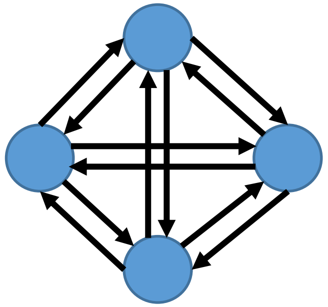


Theorem: Let $\boldsymbol{G}$ be a simple digraph with $\boldsymbol{n}$ vertices and $\boldsymbol{m}$ edges. Then,

$$\boldsymbol{m} \leq \boldsymbol{n}(\boldsymbol{n} - \mathbf{1})$$



$\frac{\boldsymbol{n}(\boldsymbol{n}-\mathbf{1})}{\mathbf{2}}$ edges

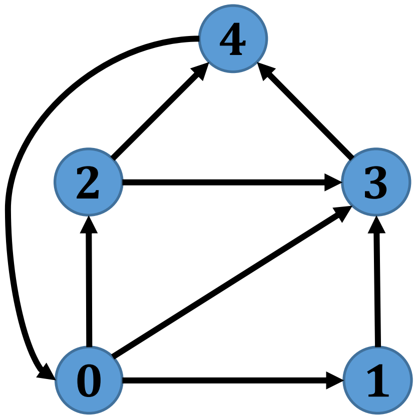


$\boldsymbol{n}(\boldsymbol{n} - \mathbf{1})$ edges

Corollary: A simple digraph with $\boldsymbol{n}$ vertices has $\boldsymbol{O}(\boldsymbol{n}^{\mathbf{2}})$ edges

Digraph Representation: Set of Edges

Maintain a list of directed edges (array or linked list)



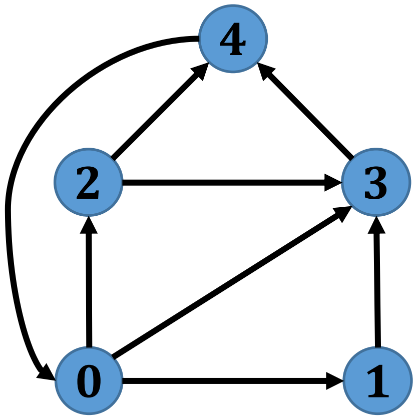
$(0, 1)$
$(0, 2)$
$(0, 3)$
$(1, 3)$
$(2, 3)$
$(2, 4)$
$(3, 4)$
$(4, 0)$

- Also have to store a separate list of vertices since some vertices have no edges

Digraph Representation: Adjacency Matrix

Maintain a **2-dimensional** $\boldsymbol{n} \times \boldsymbol{n}$ boolean array

- For each directed edge $(\boldsymbol{u}, \boldsymbol{v})$, i.e $\boldsymbol{u} \rightarrow \boldsymbol{v}$, $\mathbf{adj}[\boldsymbol{u}][\boldsymbol{v}] = \mathbf{true}\ (\mathbf{1})$

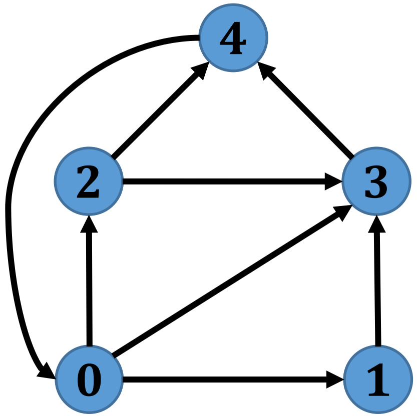


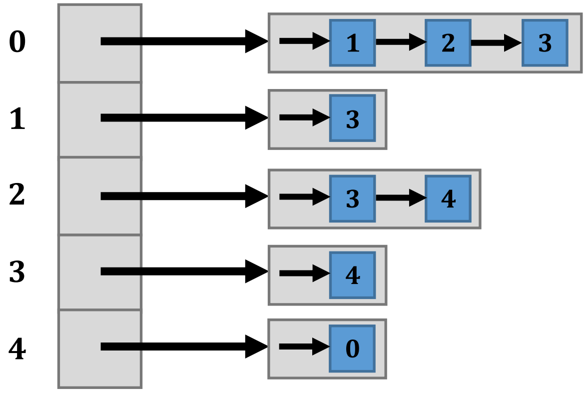
Source \ Destination	0	1	2	3	4
0	0	1	1	1	0
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	0	0	0	0

- In undirected graphs, every edge appears twice. In directed graphs, each edge appears once.

Digraph Representation: Adjacency List

Maintain an array indexed by vertices which points to a list of adjacent vertices





- In undirected graphs, every edge appears twice. In directed graphs, each edge appears once.