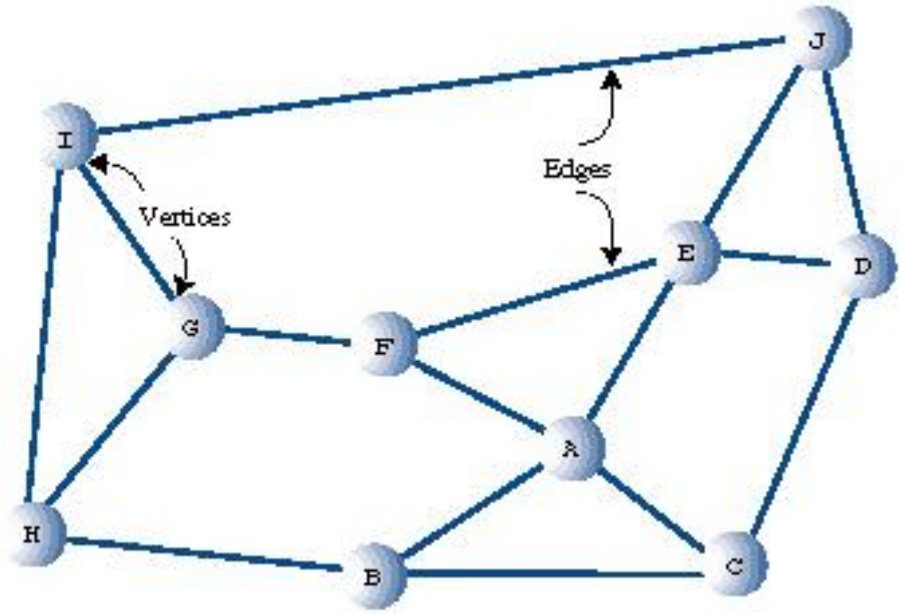
Graph Introduction

Wahyono, Ph.D.
Department of Computer Science and Electronics
Faculty of Mathematics and Natural Sciences
Universitas Gadjah Mada, Yogyakarta, Indonesia
Email: wahyo@ugm.ac.id

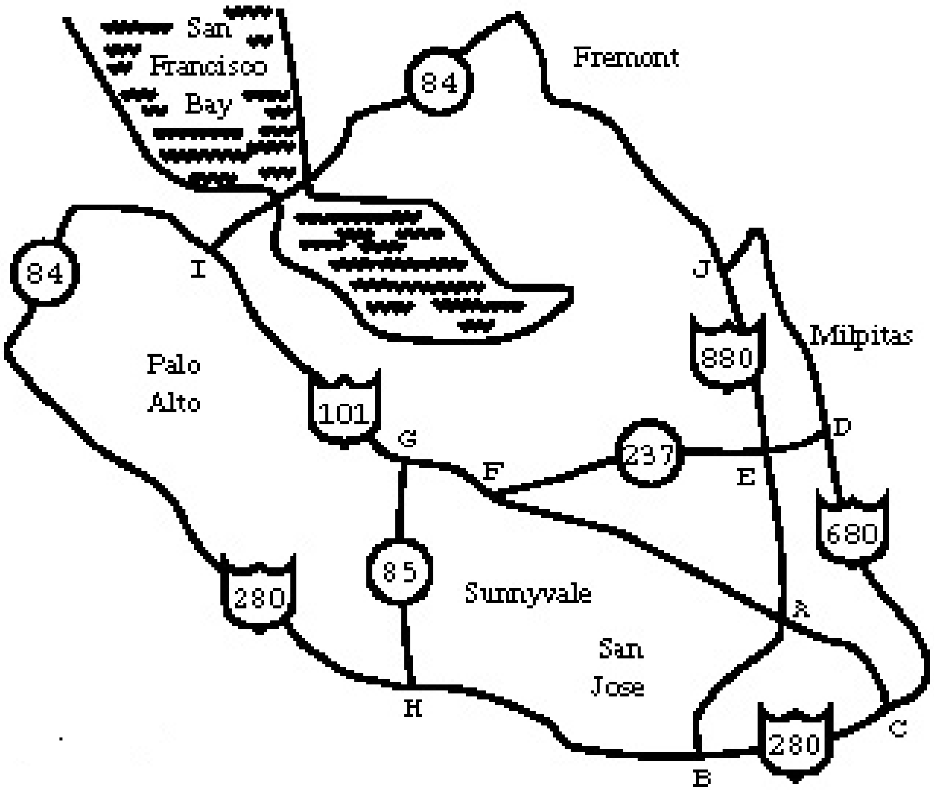
Graphs

- Graphs are a data structure which represent relationships between entities
  - *Vertices* represent entities
  - *Edges* represent some kind of relationship



Example

- The graph on the previous page could be used to model San Jose freeway connections:

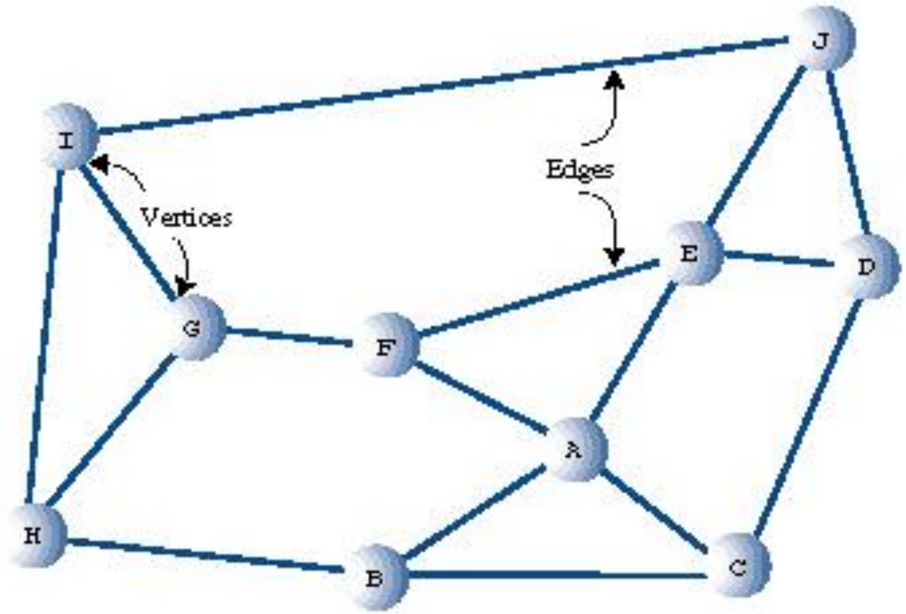


Adjacency

- Two vertices are ***adjacent*** to one another if they are connected by a single edge

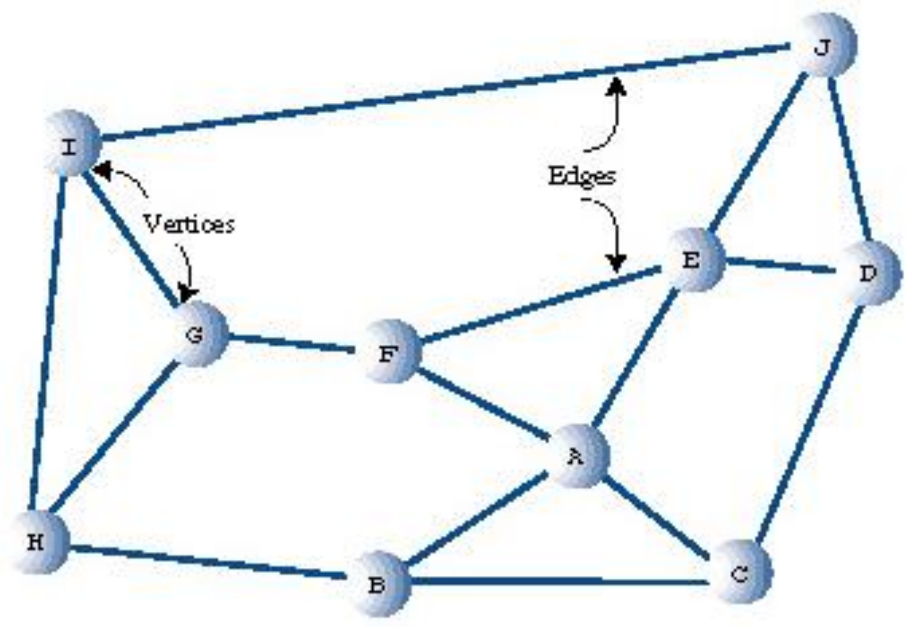
- For example:
  - I and G are adjacent
  - A and C are adjacent
  - I and F are not adjacent

- Two adjacent nodes are considered ***neighbors***



Path

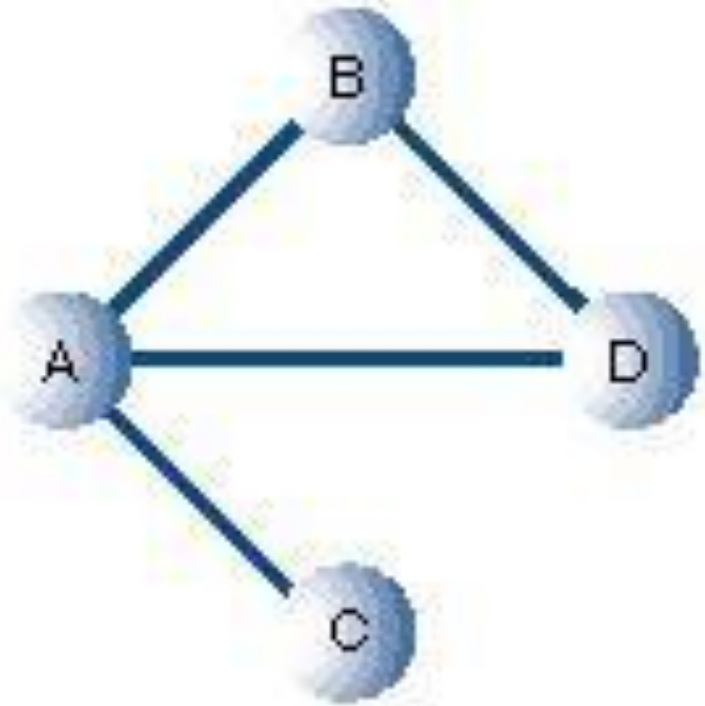
- A ***path*** is a sequence of edges



- Paths in this graph include:
  - BAEJ, CAFG, HGFEJDCAB, HIJDCBAFE, etc.

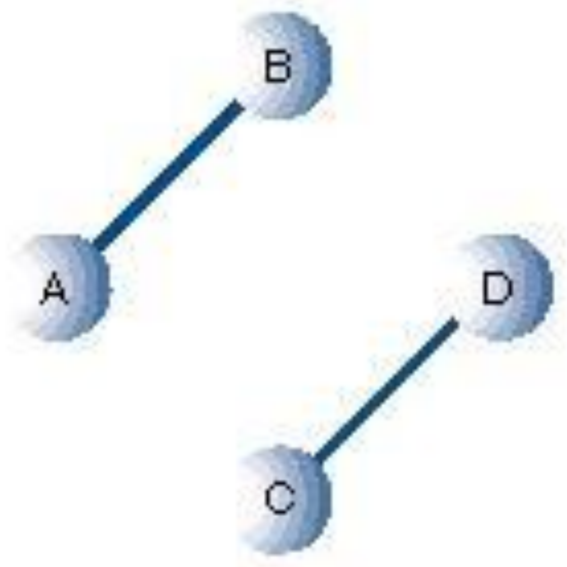
Connected Graphs

- A graph is connected if there is at least one path from every vertex to every other vertex



Unconnected Graph

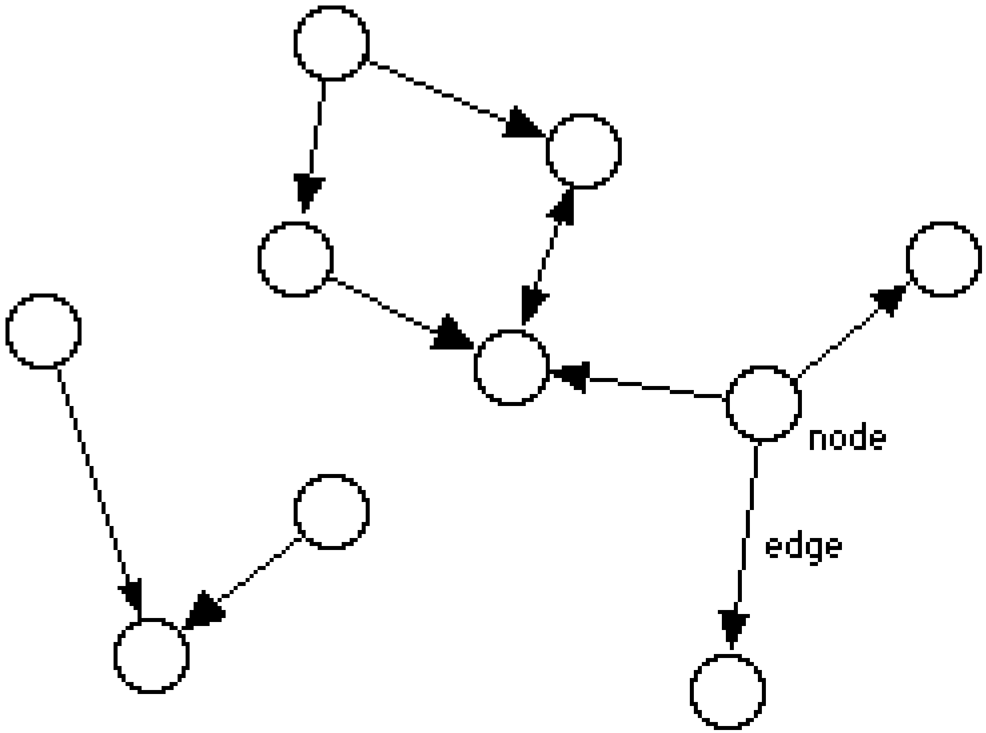
- An ***unconnected graph*** consists of several ***connected components***:



- Connected components of this graph are:
  - AB, and CD
- We'll be working with connected graphs

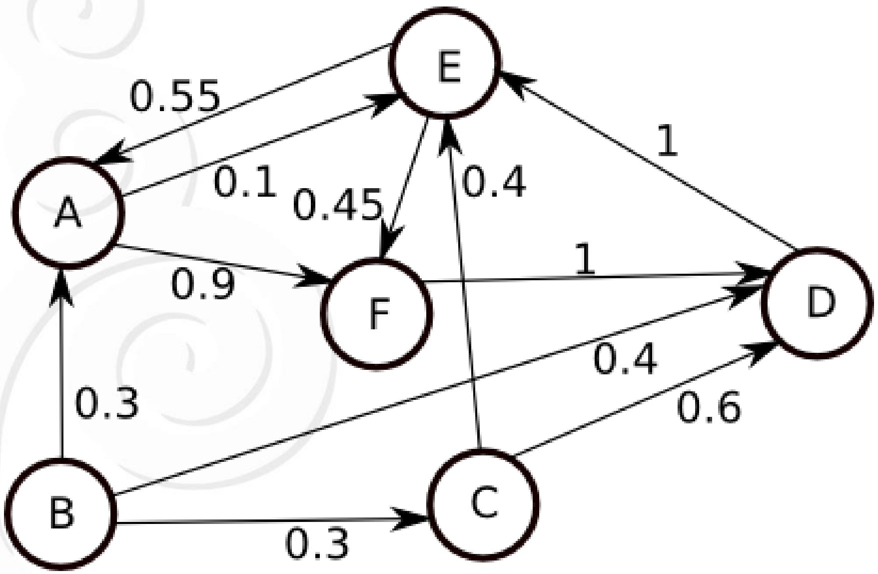
Directed Graphs

- A graph where edges have directions
  - Usually designated by an arrow



Weighted Graphs

- A graph where edges have weights, which quantifies the relationship
  - For example, you may assign path distances between cities
  - Or airline costs
- These graphs can be directed or undirected



Vertices: Java Implementation

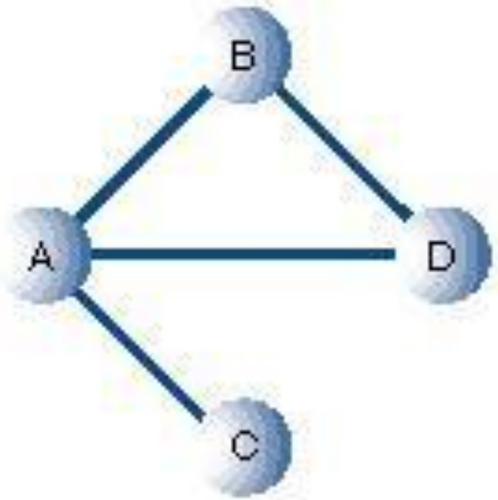
- We can represent a vertex as a Java class with:
  - Character data
  - A boolean data member to check if it has been visited

- Now we need to specify edges.
  - But in a graph, we don't know how many there will be!

- We can do this using either an adjacency ***matrix*** or an adjacency ***list***. We'll look at both.

Adjacency Matrix

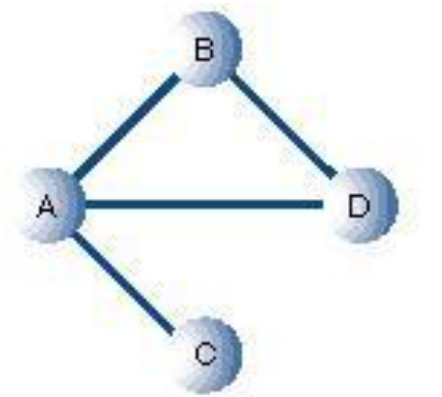
- An adjacency matrix for a graph with n nodes, is size n x n
  - Position (i, j) contains a 1 if there is an edge connecting node i with node j
  - Zero otherwise
- For example, here is a graph and its adjacency matrix:



	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Redundant?

- This may seem a bit redundant:

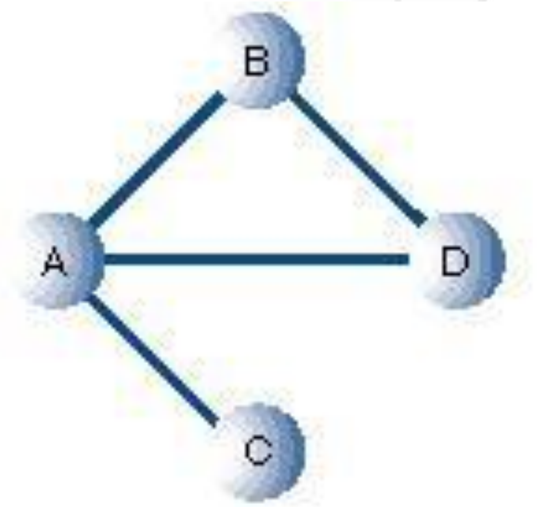


	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

- Why store two pieces of information for the same edge?
  - i.e., (A, B) and (B, A)
- Unfortunately, there's no easy way around it
  - Because edges have no direction
  - No concept of 'parents' and 'children'

Adjacency List

- An array of linked lists
  - Index by vertex, and obtain a linked list of neighbors
- Here is the same graph, with its adjacency list:



Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

Application: Searches

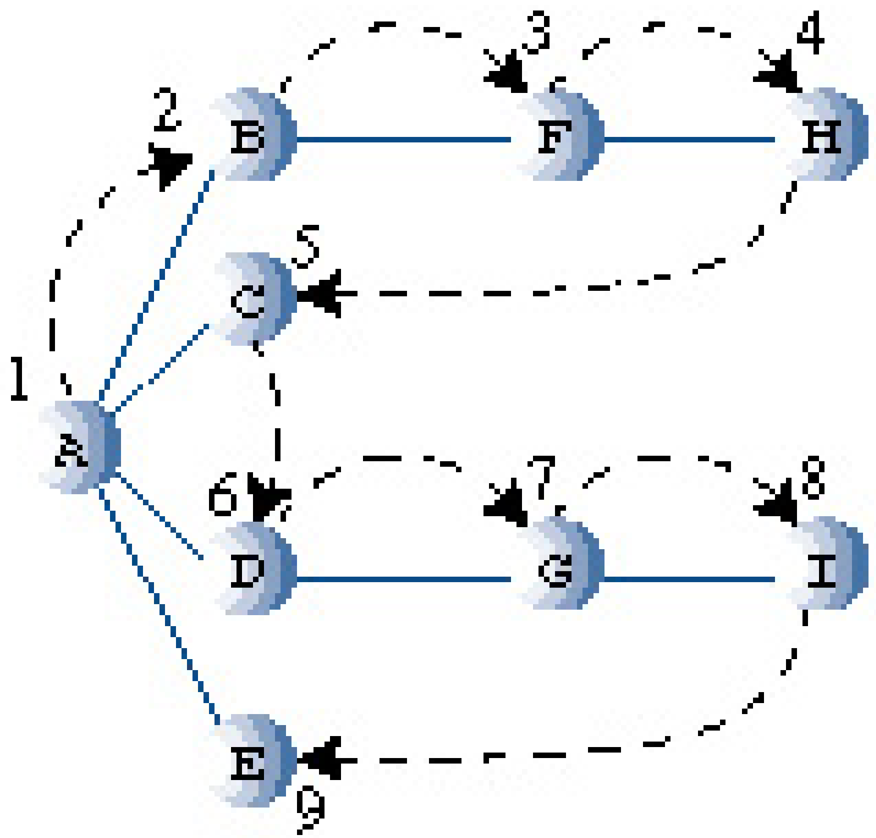
- A fundamental operation for a graph is:
  - Starting from a particular vertex
  - Find all other vertices which can be reached by following paths

- Example application
  - How many towns in the US can be reached by train from Tampa?

- Two approaches
  - Depth first search (DFS)
  - Breadth first search (BFS)

Depth First Search (DFS)

- Idea
  - Pick a starting point
  - Follow a path to unvisited vertices, as long as you can until you hit a dead end
  - When you hit a dead end, go back to a previous spot and hit unvisited vertices
  - Stop when every path is a dead end



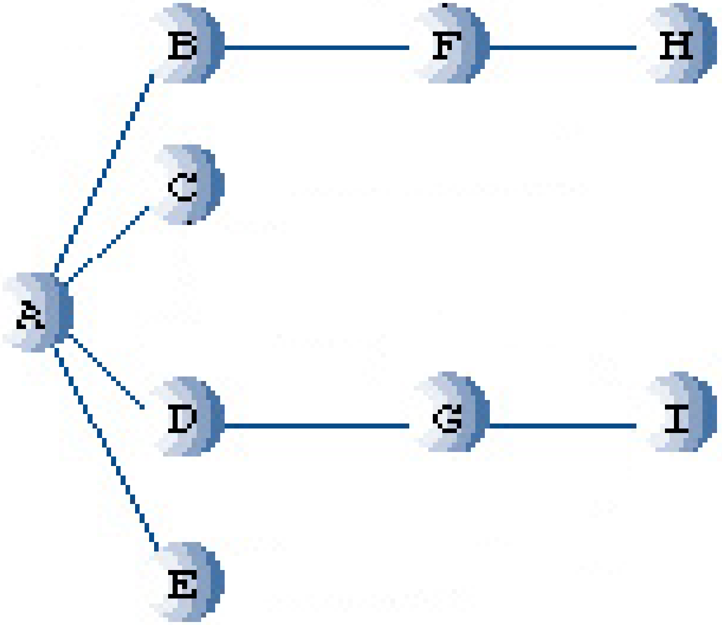
Depth First Search (DFS)

- Algorithm
  - Pick a vertex (call it A) as your starting point
  - Visit this vertex, and:
    - Push it onto a stack of visited vertices
    - Mark it as visited (so we don't visit it again)
  - Visit any neighbor of A that hasn't yet been visited
    - Repeat the process
  - When there are no more unvisited neighbors
    - Pop the vertex off the stack
  - Finished when the stack is empty

- Note: We get as far away from the starting point until we reach a dead end, then pop (can be applied to mazes)

Example

- Start from A, and execute depth first search on this graph, showing the contents of the stack at each step
  - Every step, we'll either have a visit or pop



Depth First Search: Complexity

- Let |V| be the number of vertices in a graph
- And let |E| be the number of edges

- In the worst case, we visit every vertex and every edge:
  - O(|V| + |E|) time

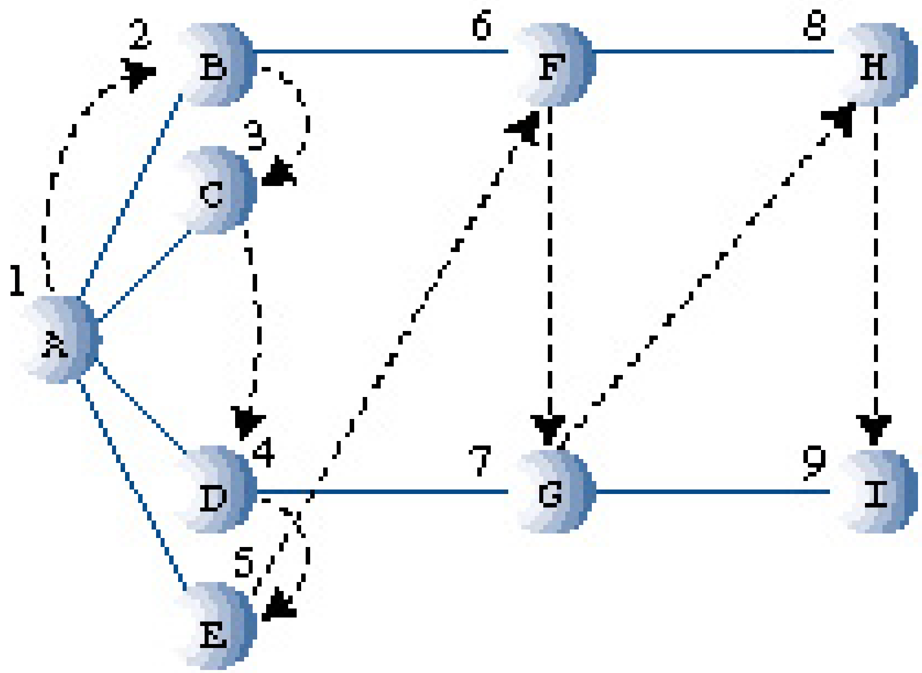
At first glance, this doesn't look too bad

- But remember a graph can have lots of edges!
- Worst case, every vertex is connected to every other:
  - $(n-1) + (n-2) + \ldots + 1 = O(n^2)$
- So it can be expensive if the graph has many edges

Breadth First Search (BFS)

- Same application as DFS; we want to find all vertices which we can get to from a starting point, call it A

- However this time, instead of going as far as possible until we find a dead end, like DFS
  - We visit all the closest vertices first
  - Then once all the closest vertices are visited, branch further out

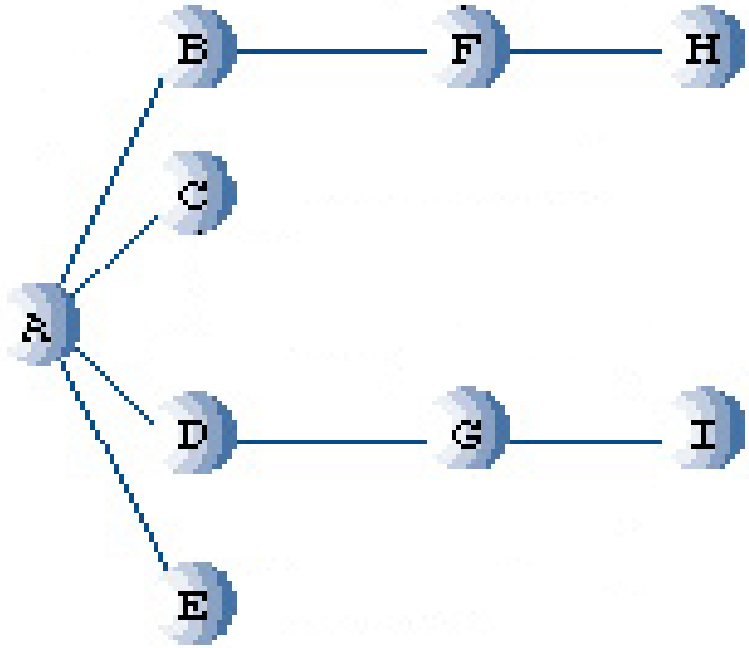


Breadth First Search (BFS)

- We're going to use a queue instead of a stack!
- Algorithm
  - Start at a vertex, call it current
  - If there is an unvisited neighbor, mark it, and insert it into the queue
  - If there is not:
    - If the queue is empty, we are done, otherwise:
    - Remove a vertex from the queue and set current to that vertex, and repeat the process

Example

- Start from A, and execute breadth first search on this graph, showing the contents of the queue at each step
  - Every step, we'll either have a visit or a removal



Breadth First Search: Complexity

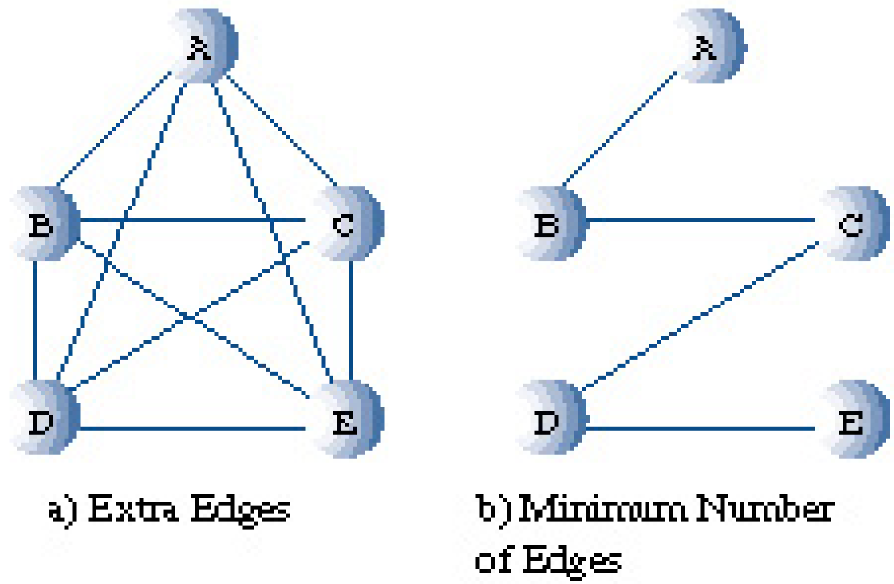
- Let |V| be the number of vertices in a graph
- And let |E| be the number of edges

- In the worst case, we visit every vertex and every edge:
  - O(|V| + |E|) time
  - Same as DFS

- Again, if the graph has lots of edges, you approach quadratic run time which is the worst case

Minimum Spanning Trees (MSTs)

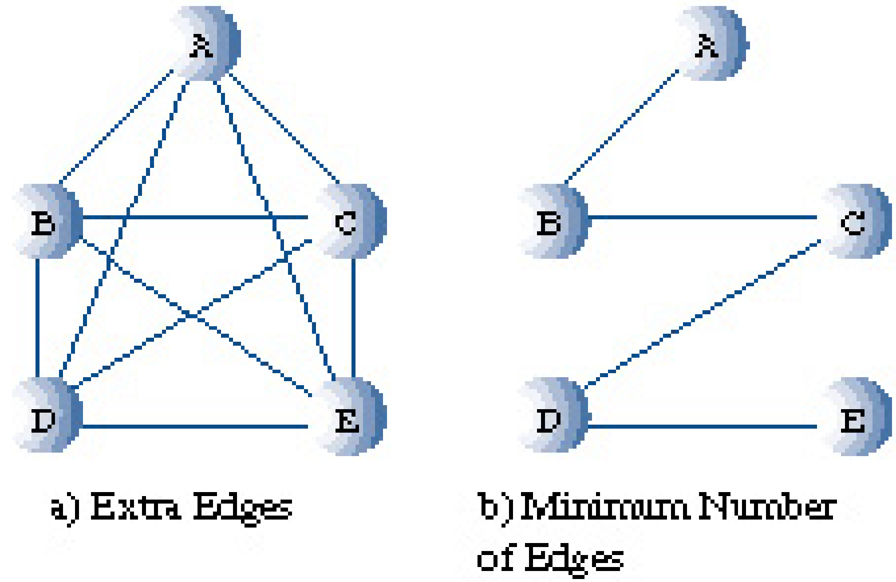
- On that note of large numbers of edges slowing down our precious search algorithms:
  - Let's look at MSTs, which can help ameliorate this problem
- It would be nice to take a graph and reduce the number of edges to the minimum number required to span all vertices:



What's the number of edges now?

We've done it already…

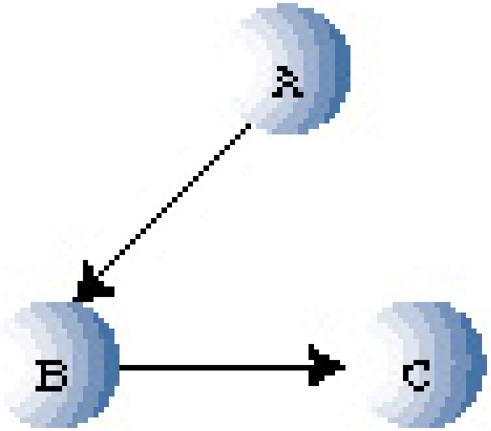
- Actually, if you execute DFS you've already computed the MST!



- Think about it: you follow a path for as long as you can, then backtrack (visit every vertex at most once)
  - You just have to save edges as you go

Directed Graphs

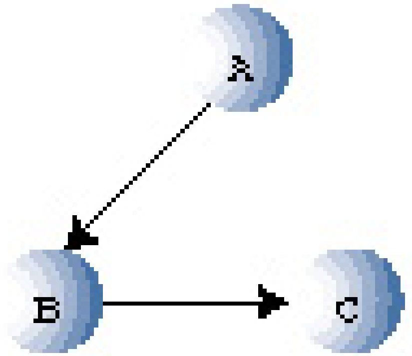
- A directed graph is a graph where the edges have direction, signified by arrows:



- This will simplify the adjacency matrix a bit…

Adjacency Matrix

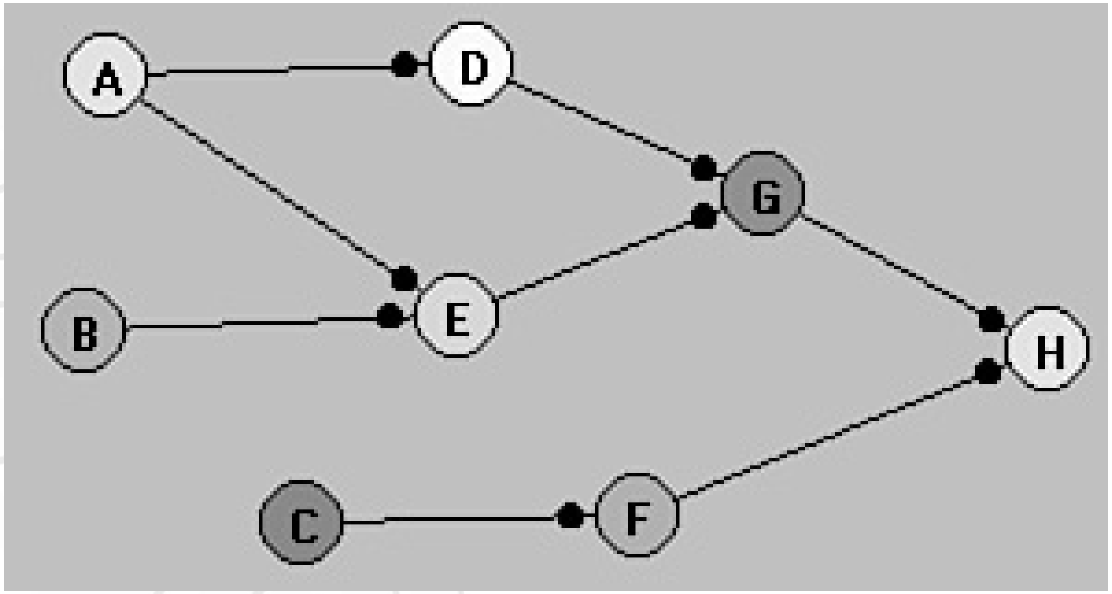
- The adjacency matrix for this graph does not contain redundant entries
  - Because now each edge has a source and a sink
  - So entry (i, j) is only set to 1 if there is an edge going from i to j
    - 0 otherwise



	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

Topological Sort

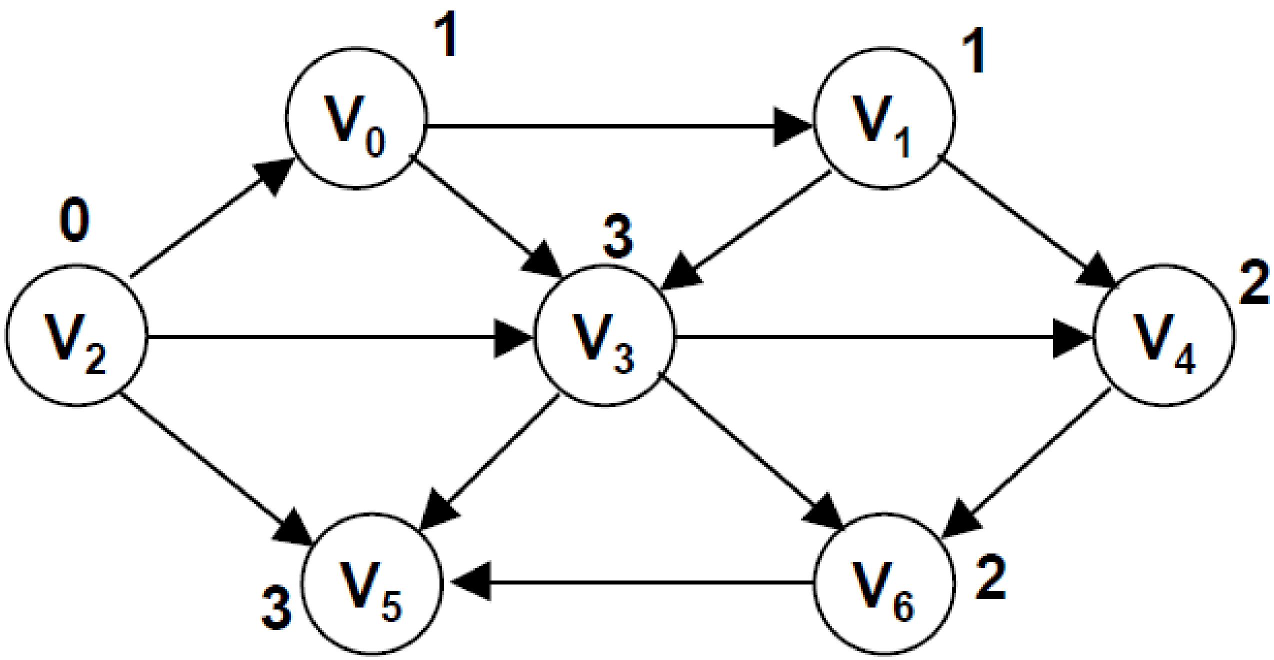
- Only works with DAGs (Directed Acyclic Graphs)
  - That is if the graph has a cycle, this will not work



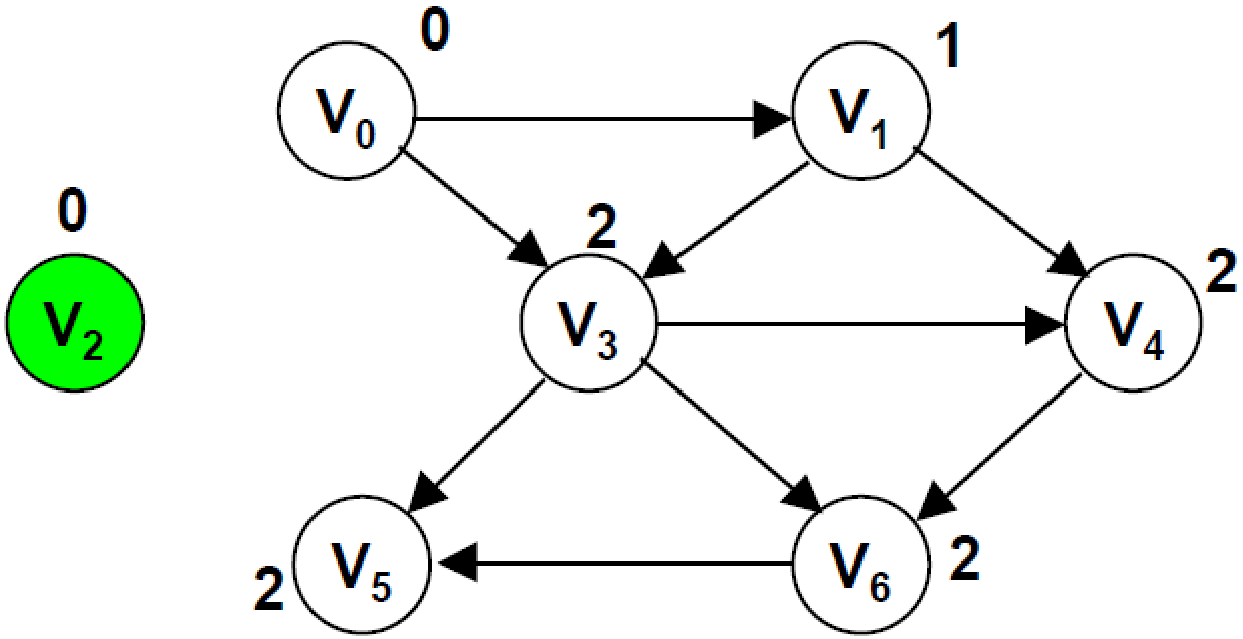
- Idea: Sort all vertices such that if a vertex's successor always appears after it in a list (application: course prerequisites)

Topological Sorting

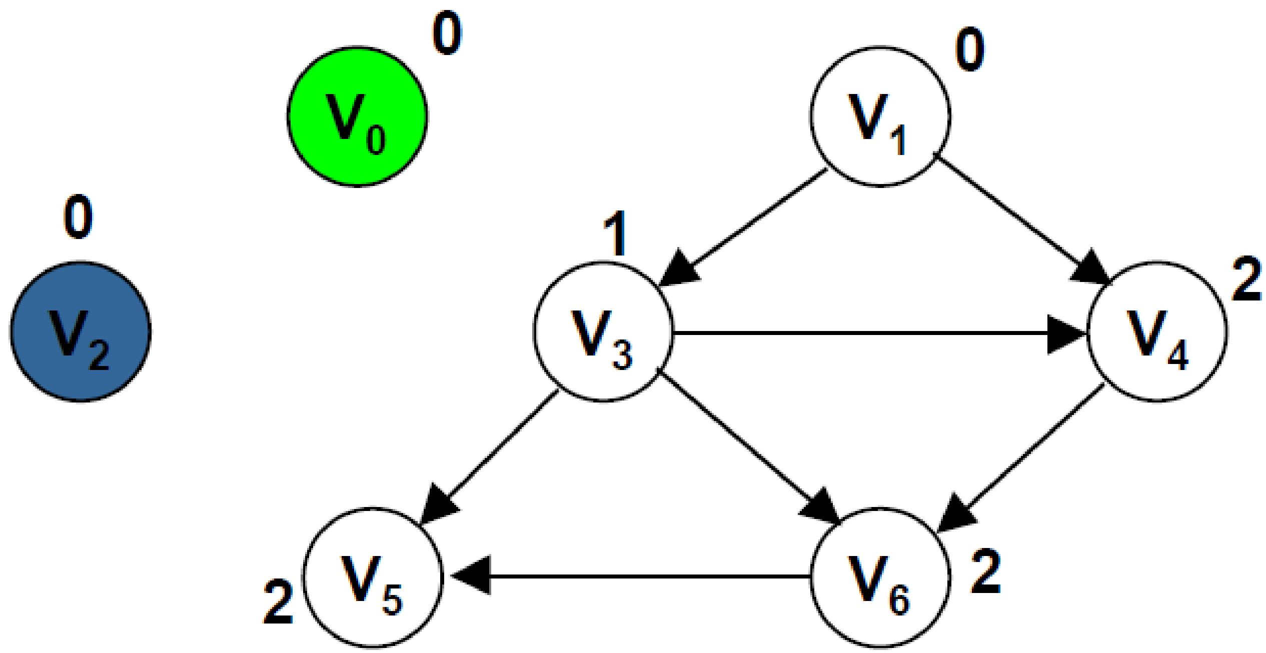
- **Start from vertex that have in-degree 0**



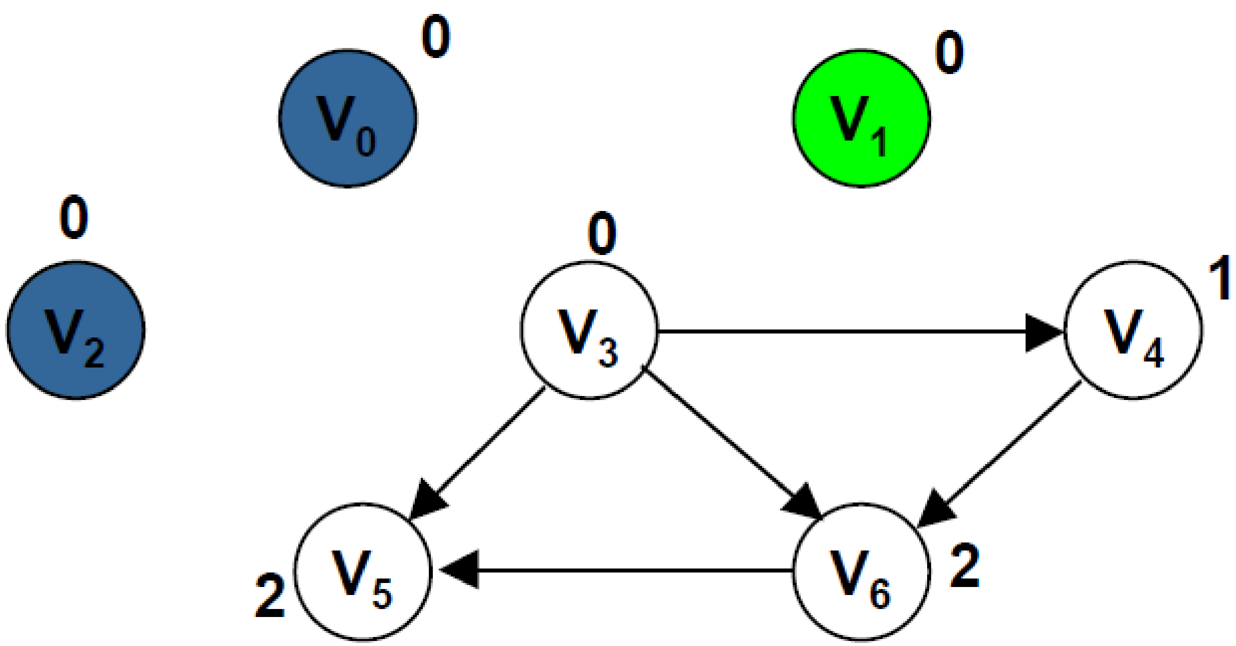
Topological Sorting



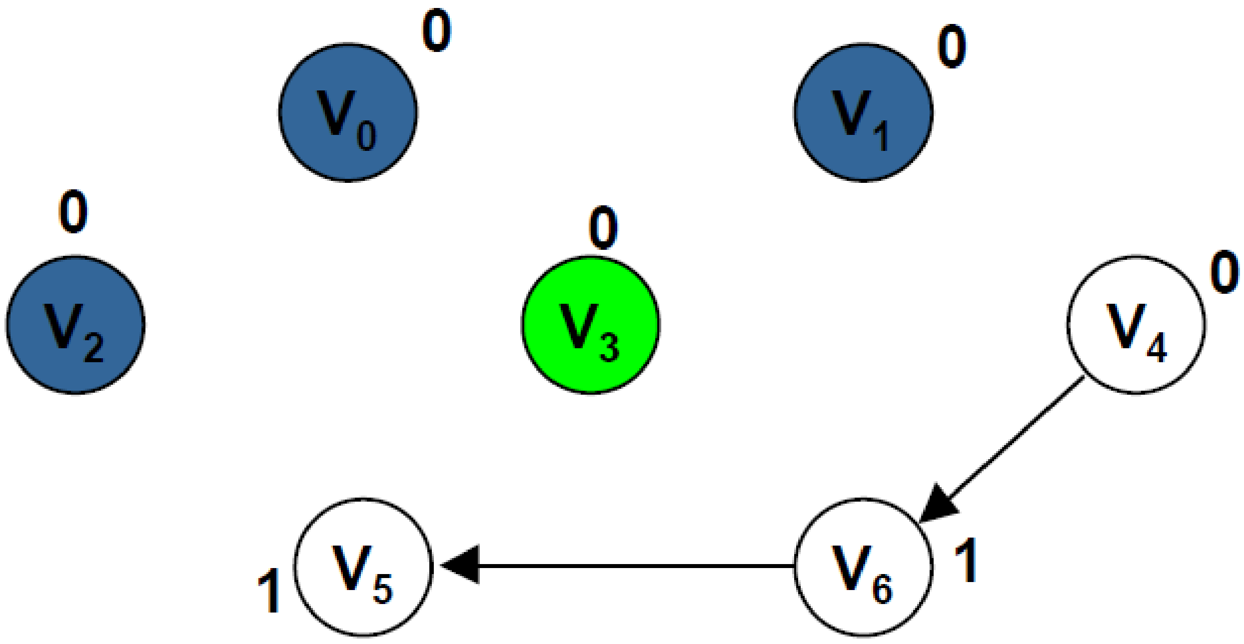
Topological Sorting



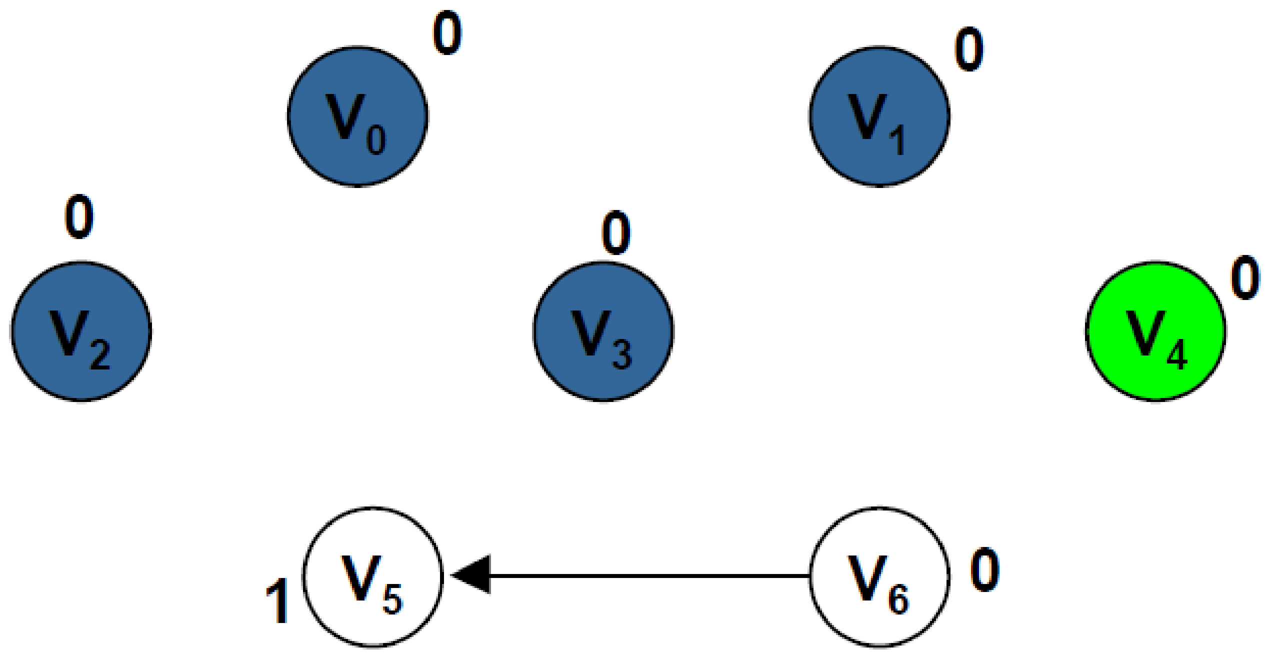
Topological Sorting



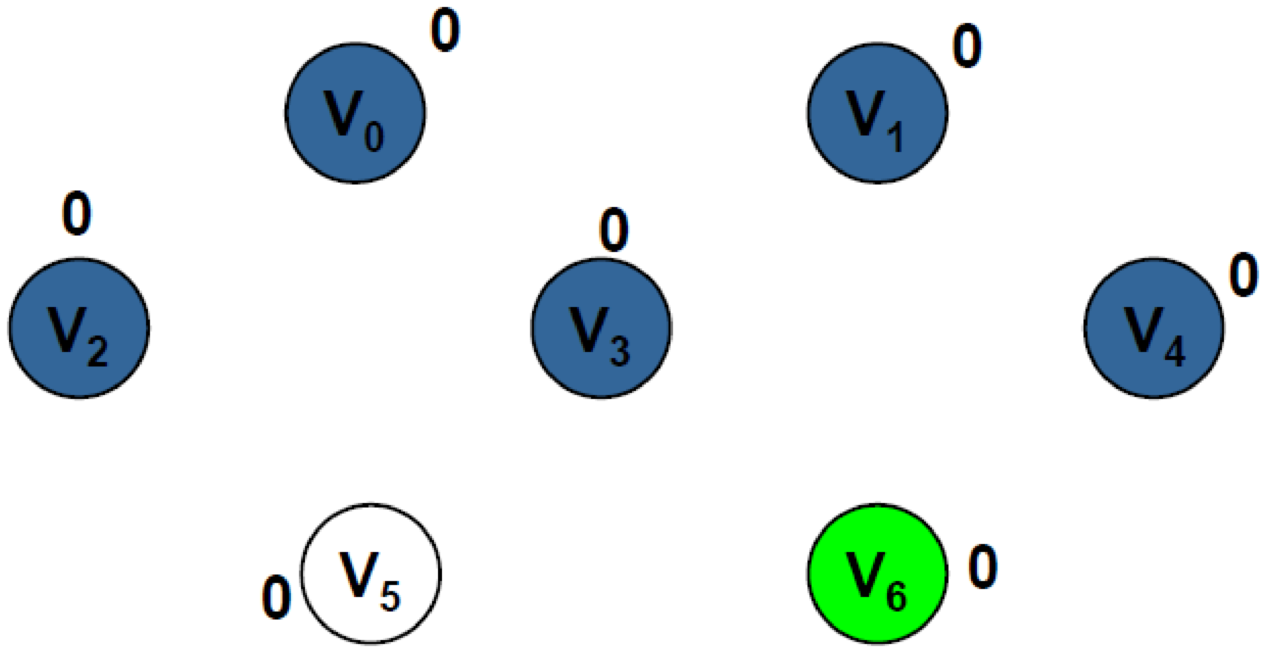
Topological Sorting



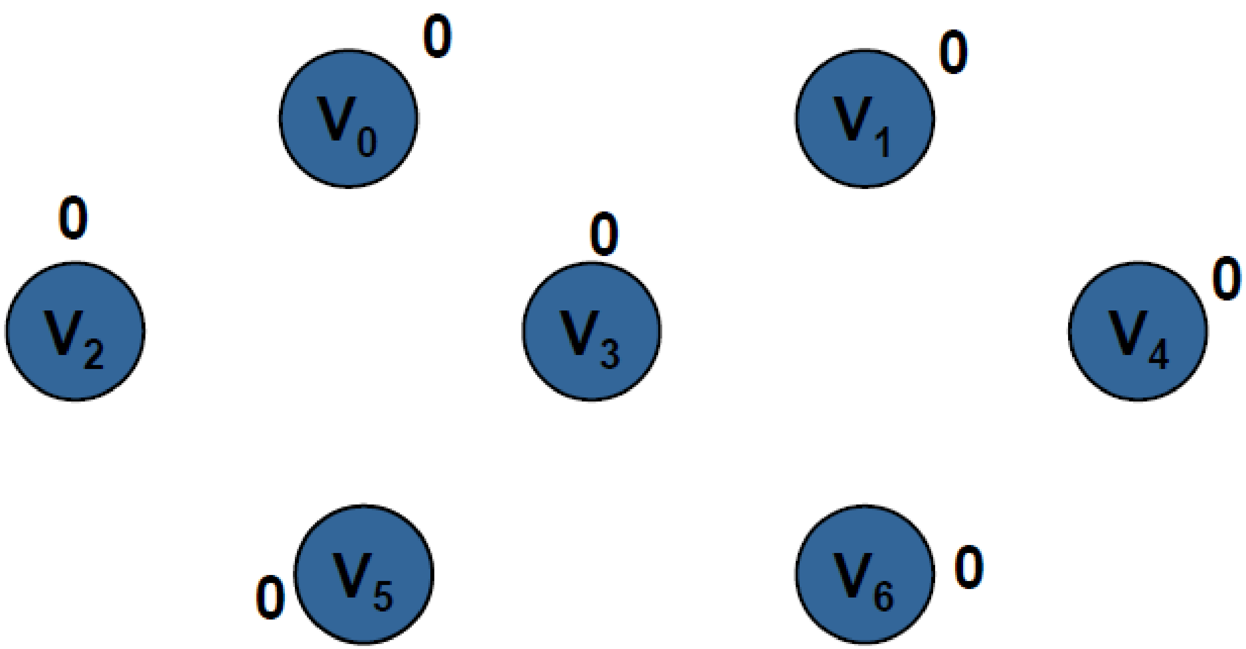
Topological Sorting



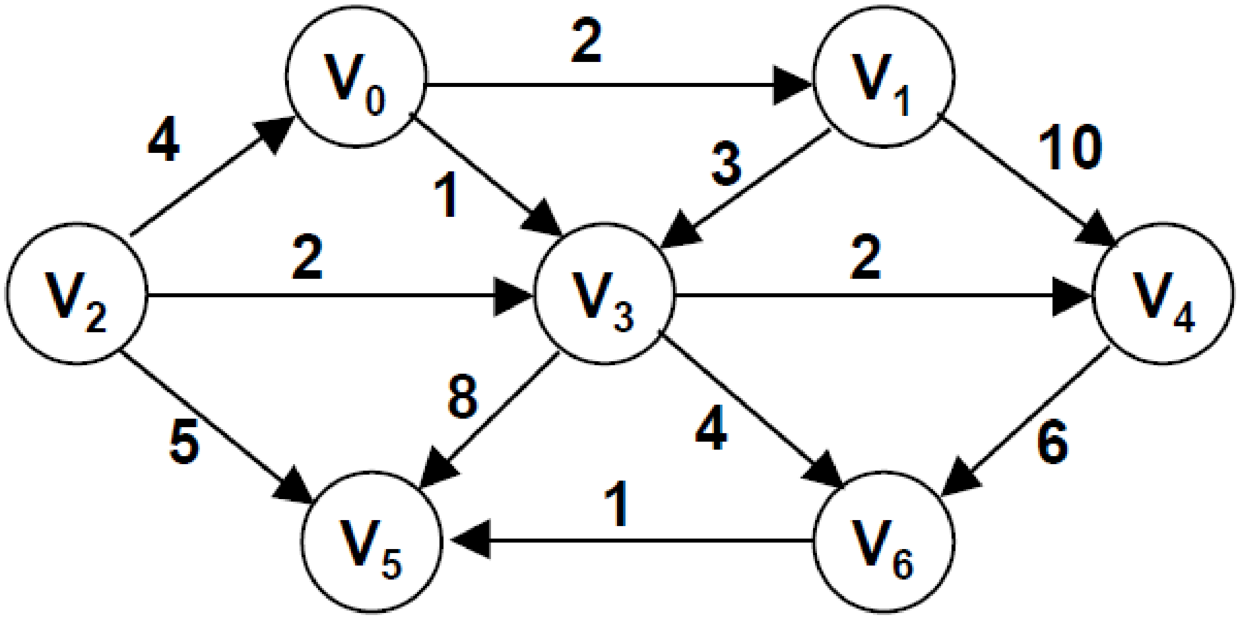
Topological Sorting



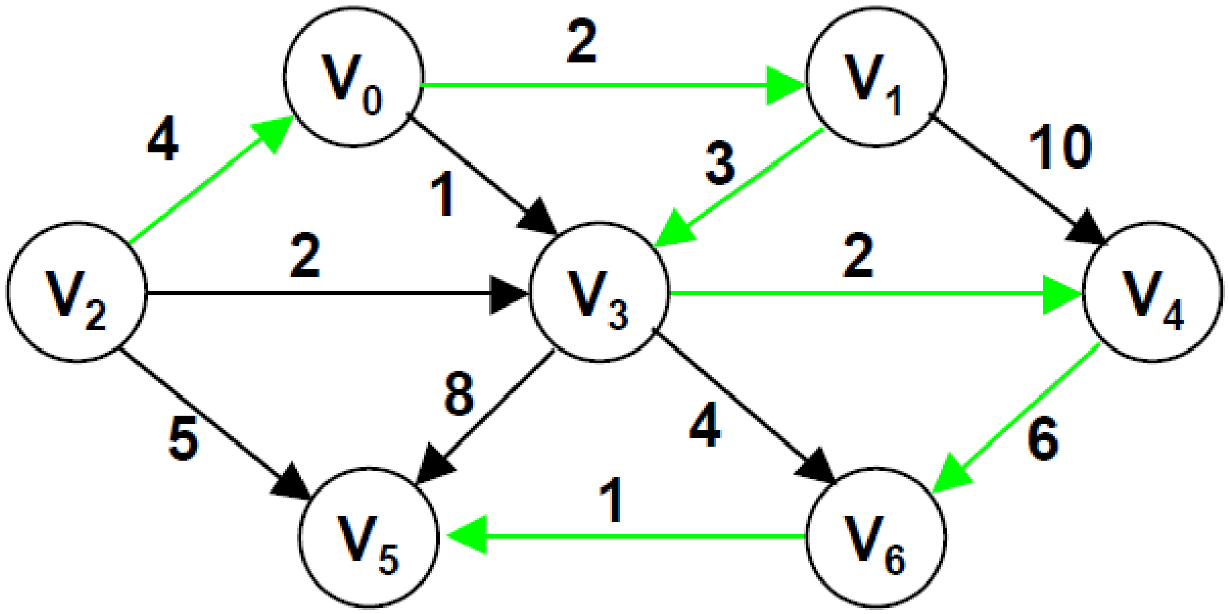
Topological Sorting



Topological Sorting



Topological Sorting



THANK YOU

Mengakar Kuat
Menjulang
Tinggi