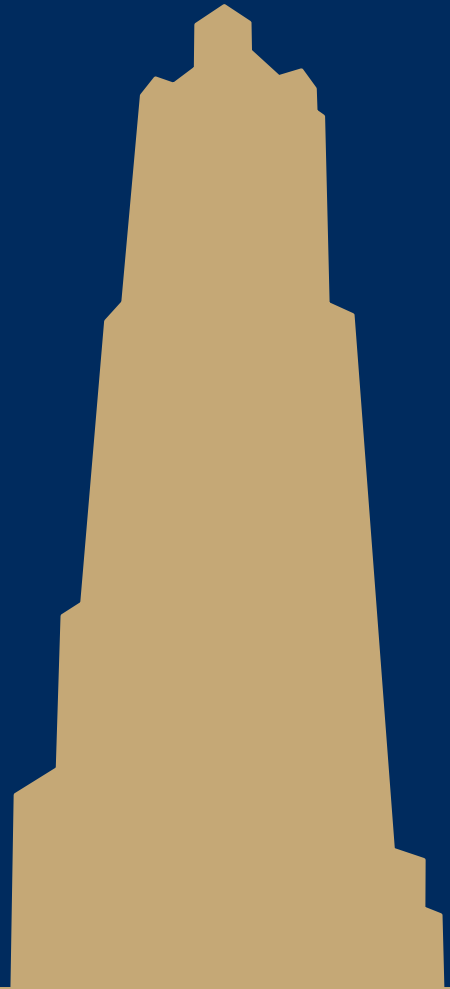
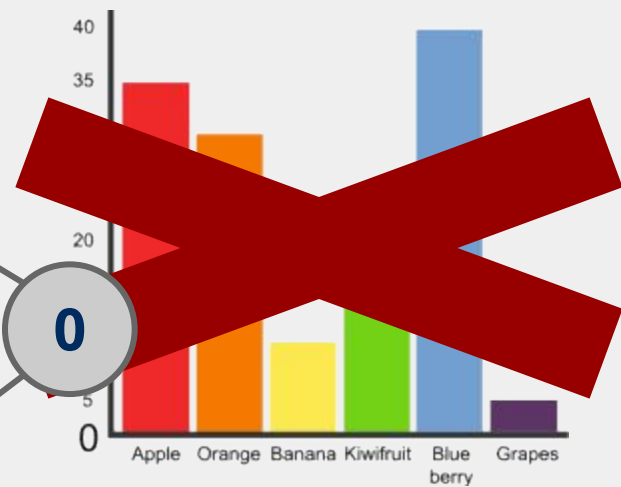
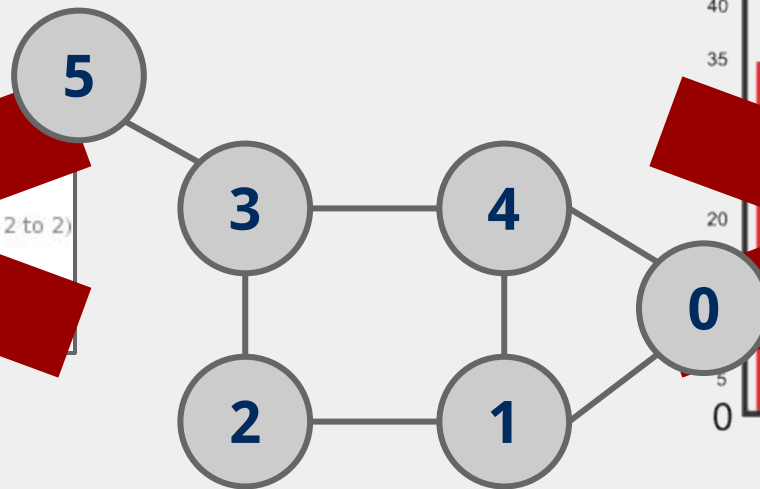
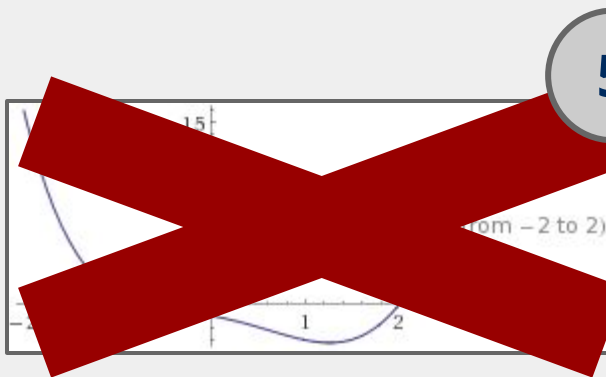


CS/COE 1501

www.cs.pitt.edu/~nlf4/cs1501/

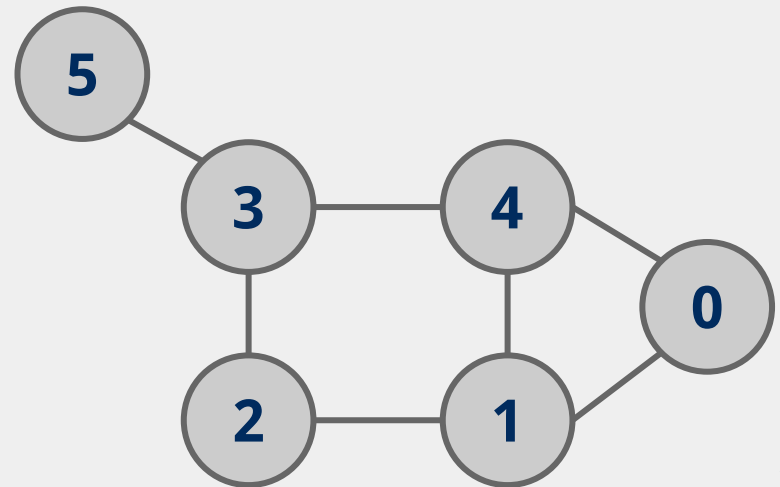
Graphs





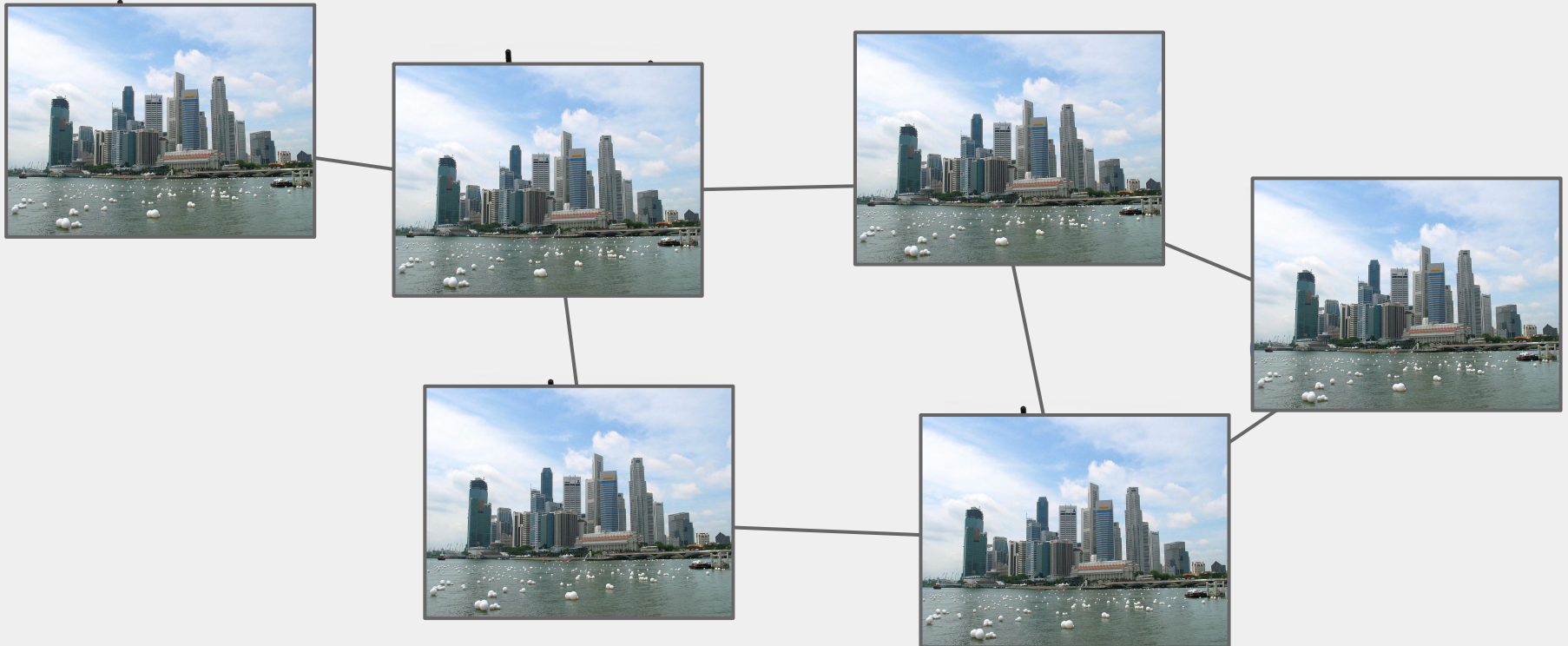
Graphs

- A graph $G = (V, E)$
 - Where V is a set of vertices
 - E is a set of edges connecting vertex pairs
- Example:
 - $V = \{0, 1, 2, 3, 4, 5\}$
 - $E = \{(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)\}$



Why?


- Can be used to model many different scenarios



Some definitions

- Undirected graph
 - Edges are unordered pairs: $(A, B) == (B, A)$
- Directed graph
 - Edges are ordered pairs: $(A, B) != (B, A)$
- Adjacent vertices, or neighbors
 - Vertices connected by an edge

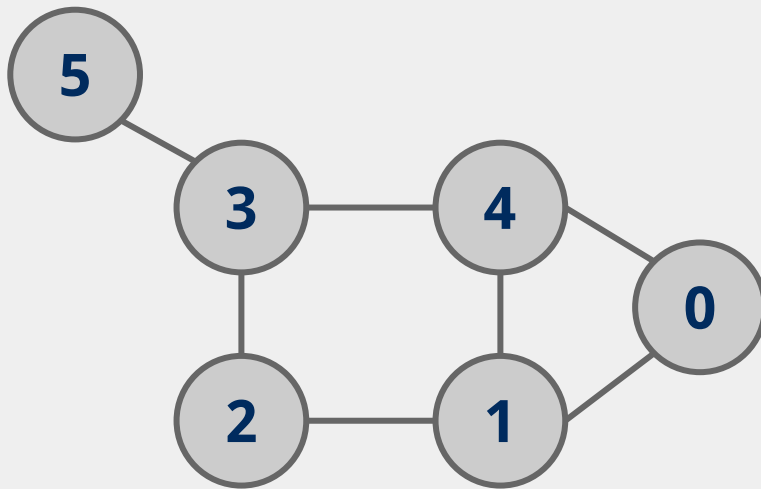
Graph sizes

- Let $v = |V|$, and $e = |E|$
- Given v , what are the minimum/maximum sizes of e ?
 - Minimum value of e ?
 - Definition doesn't necessitate that there are any edges...
 - So, 0
 - Maximum of e ?
 - Depends...
 - Are self edges allowed? 
 - Directed graph or undirected graph?
 - In this class, we'll assume directed graphs have self edges while undirected graphs do not

More definitions

- A graph is considered *sparse* if:
 - $e \leq v \lg v$
- A graph is considered *dense* as it approaches the maximum number of edges
 - I.e., $e \approx \text{MAX} - \epsilon$
- A *complete* graph has the maximum number of edges

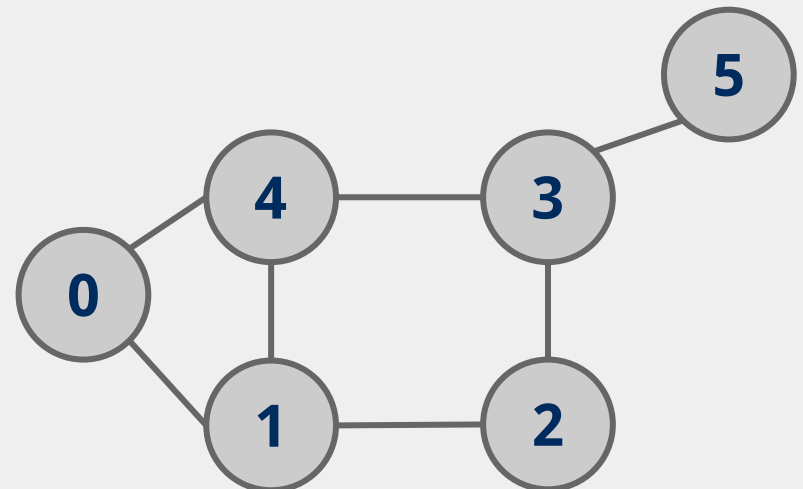
Question:



==

or

!=



• ?

Representing graphs


- Trivially, graphs can be represented as:
 - List of vertices
 - List of edges
- Performance?
 - Assume we're going to be analyzing static graphs
 - I.e., no insert and remove
 - So what operations should we consider?

Using an adjacency matrix

- Rows/columns are vertex labels
 - $M[i][j] = 1$ if $(i, j) \in E$
 - $M[i][j] = 0$ if $(i, j) \notin E$

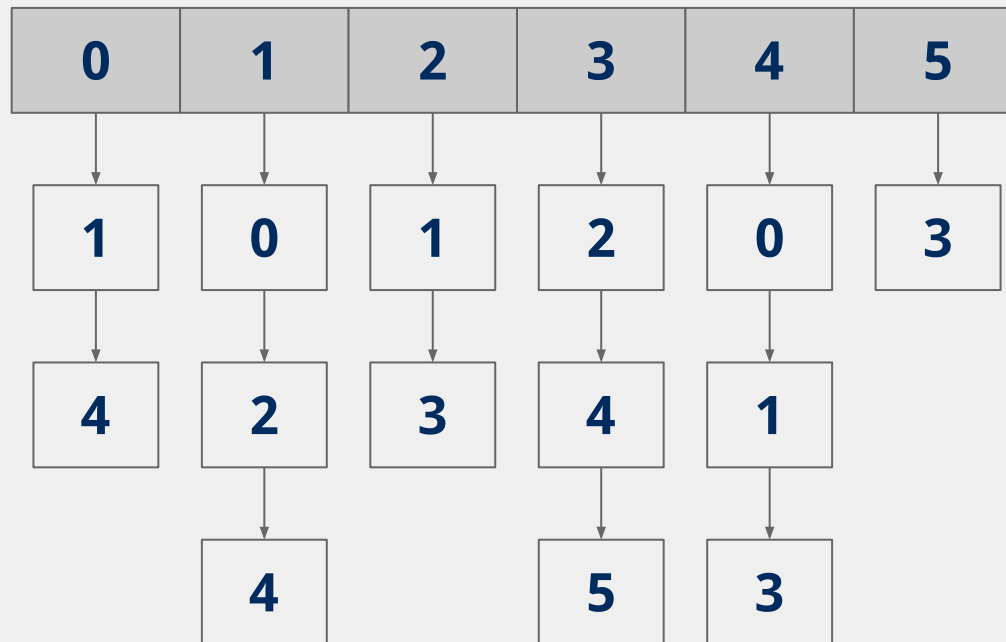
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

Adjacency matrix analysis

- Runtime? 
- Space?

Adjacency lists

- Array of neighbor lists
 - $A[i]$ contains a list of the neighbors of vertex i



Adjacency list analysis

- Runtime? 

- Space? 

Comparison

- Where would we want to use adjacency lists vs adjacency matrices?
 - What about the list of nodes/list of edges approach?

Even more definitions

- Path
 - A sequence of adjacent vertices
- Simple Path
 - A path in which no vertices are repeated
- Simple Cycle
 - A simple path with the same first and last vertex
- Connected Graph
 - A graph in which a path exists between all vertex pairs
- Connected Component
 - Connected subgraph of a graph
- Acyclic Graph
 - A graph with no cycles
- Tree
 - ?
 - A connected, acyclic graph
 - Has exactly $v-1$ edges

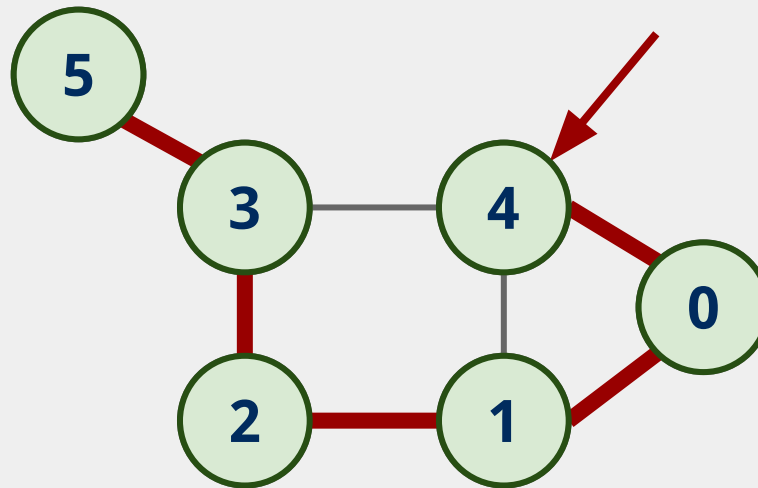
Graph traversal

- What is the best order to traverse a graph?
- Two primary approaches:
 - Depth-first search (DFS)
 - “Dive” as deep as possible into the graph first
 - Branch when necessary
 - Breadth-first search (BFS)
 - Search all directions evenly
 - I.e., from i , visit all of i 's neighbors, then all of their neighbors, etc.

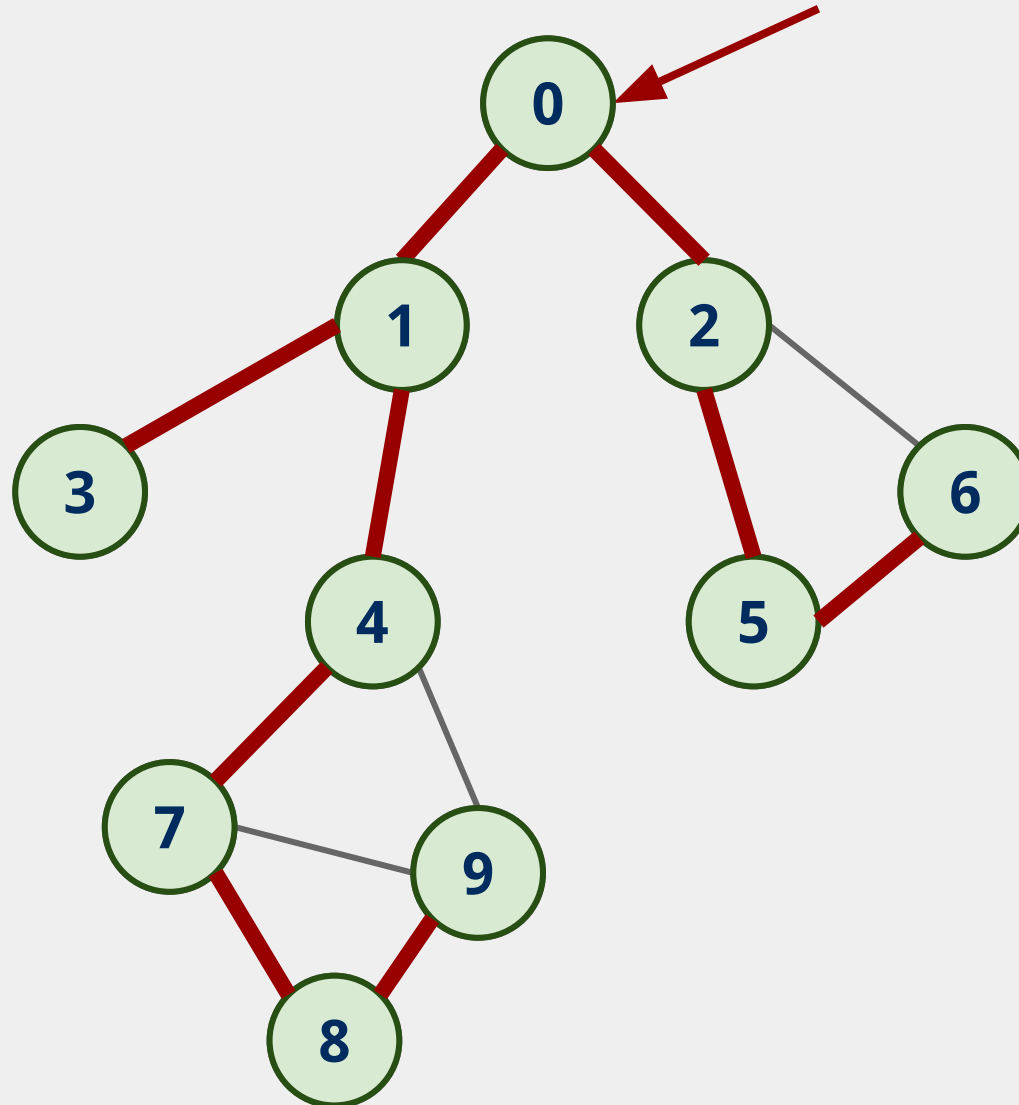
DFS

- Already seen and used this throughout the term
 - For tries...
 - For Huffman encoding...
- Can be easily implemented recursively
 - For each node, visit first unseen neighbor
 - Backtrack at dead ends (i.e., nodes with no unseen neighbors)
 - Try next unseen neighbor after backtracking

DFS example



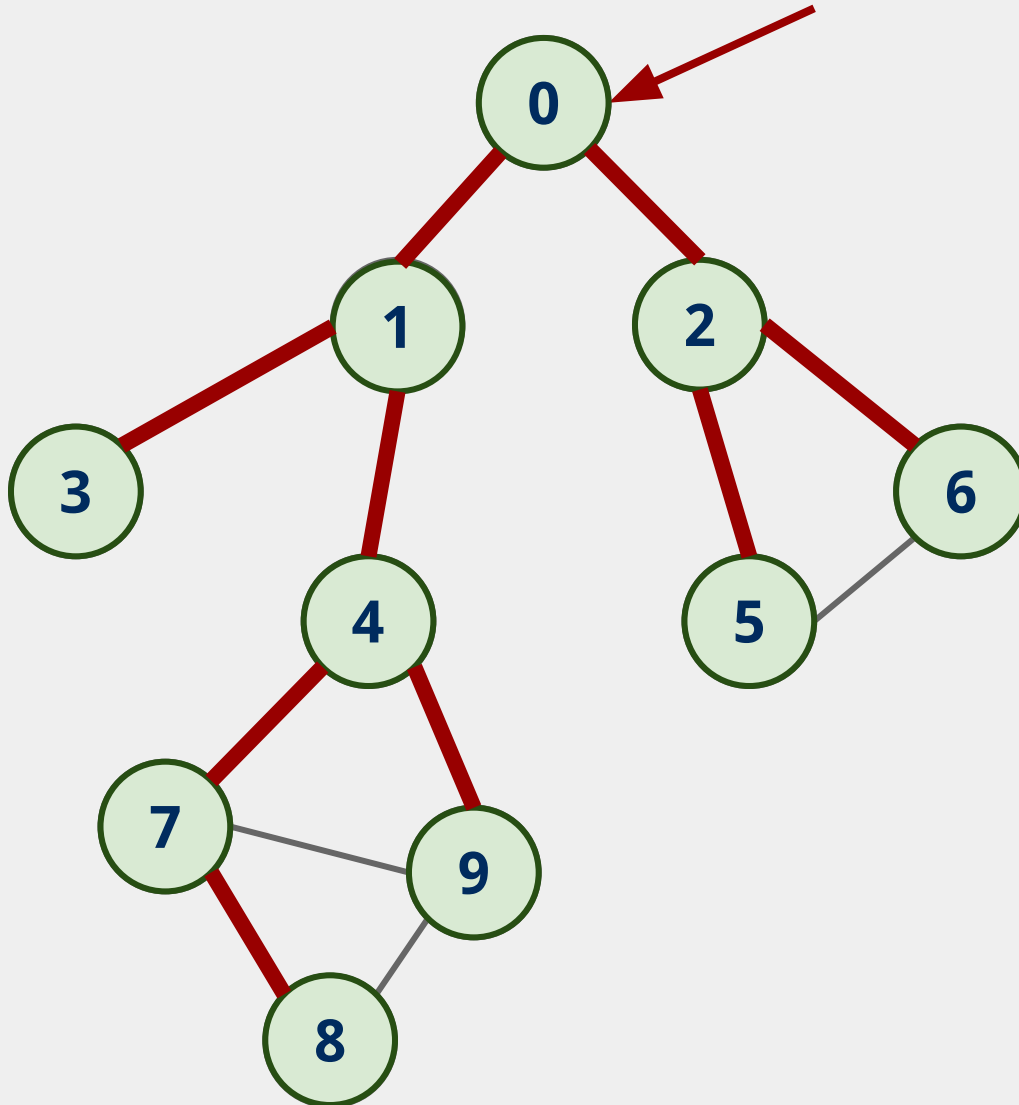
DFS example 2



BFS

- Can be easily implemented using a queue
 - For each node visited, add all of its neighbors to the queue
 - Vertices that have been seen but not yet visited are said to be the *fringe*
 - Pop head of the queue to be the next visited vertex
- See example


BFS example



Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

Analysis of graph traversals

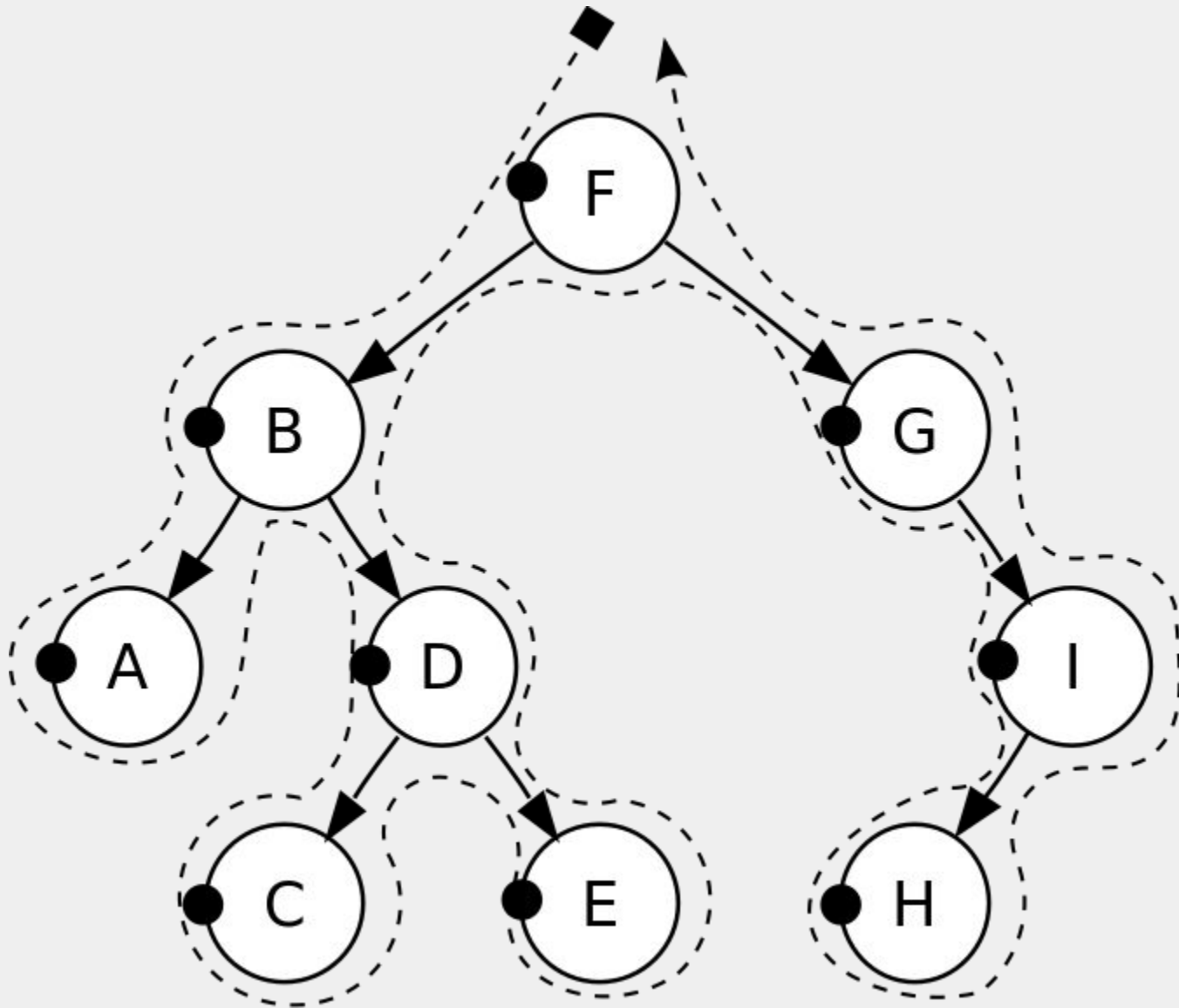
- At a high level, DFS and BFS have the same runtime 
 - Each node must be seen and then visited, but the order will differ between these two approaches
- How will the representation of the graph affect the runtimes of these traversal algorithms?



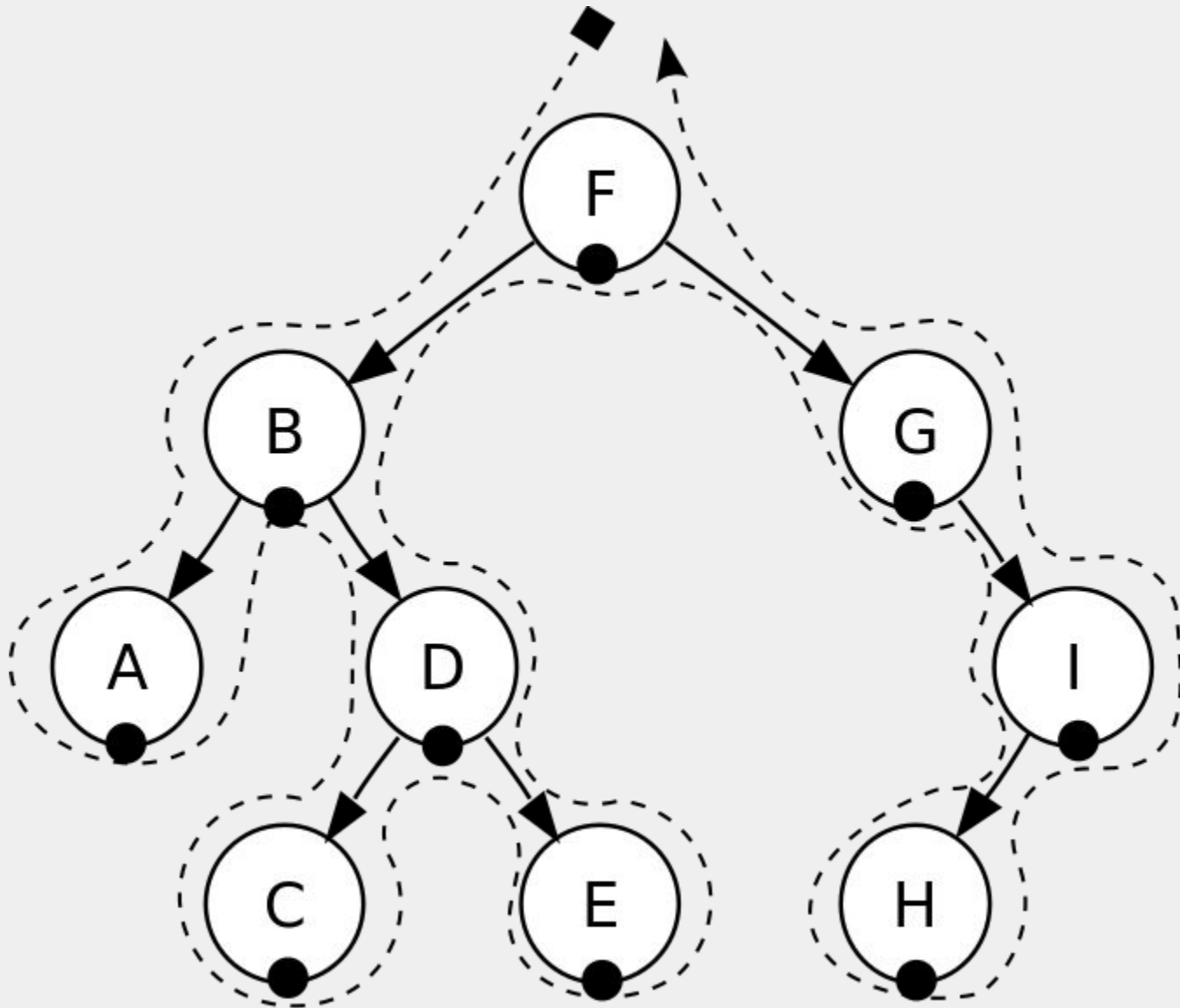
DFS and BFS would be called from a wrapper function

- If the graph is connected:
 - dfs()/bfs() is called only once and returns a *spanning tree*
- Else:
 - A loop in the wrapper function will have to continually call dfs()/bfs() while there are still unseen vertices
 - Each call will yield a spanning tree for a connected component of the graph

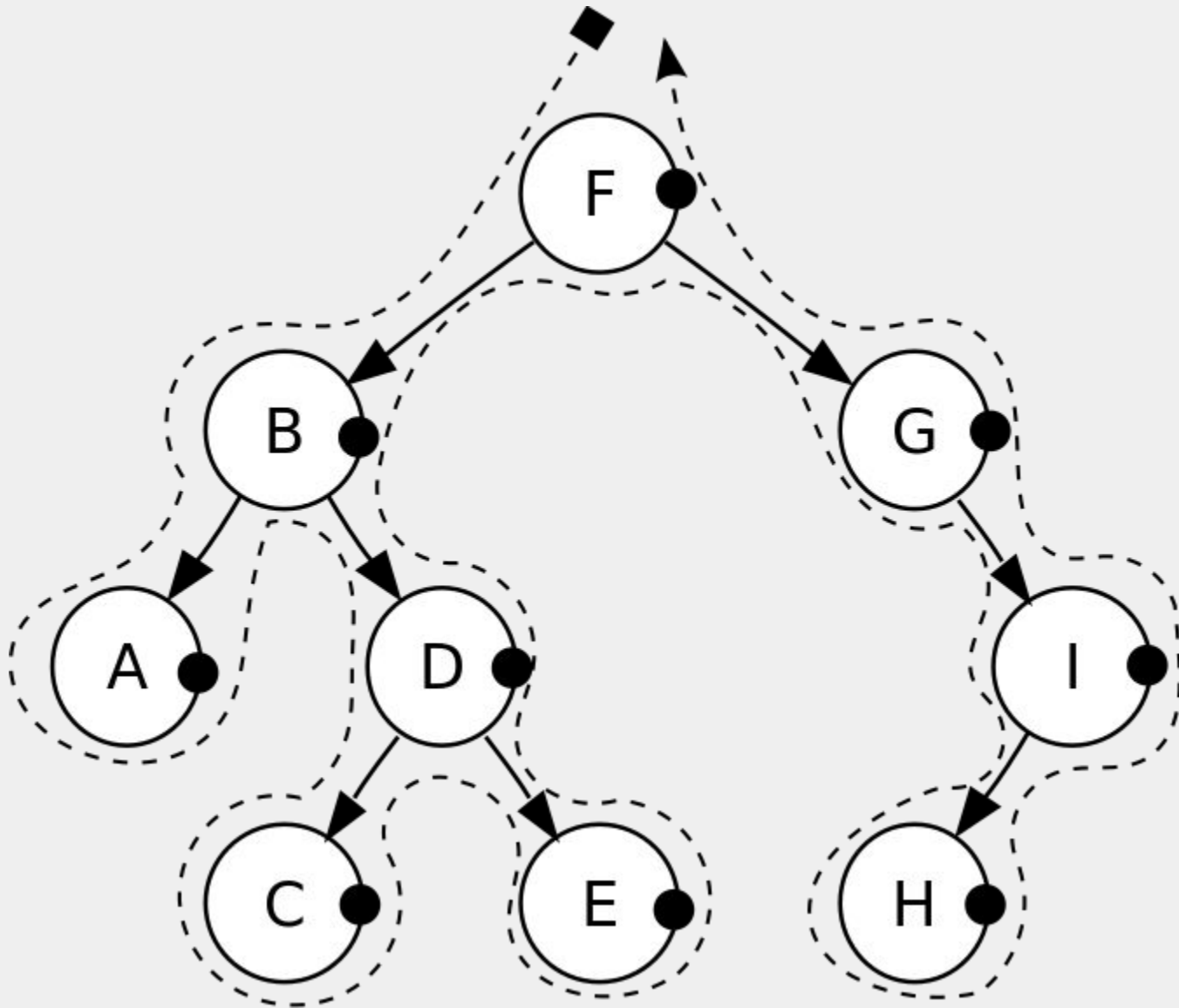
DFS pre-order traversal



DFS in-order traversal



DFS post-order traversal



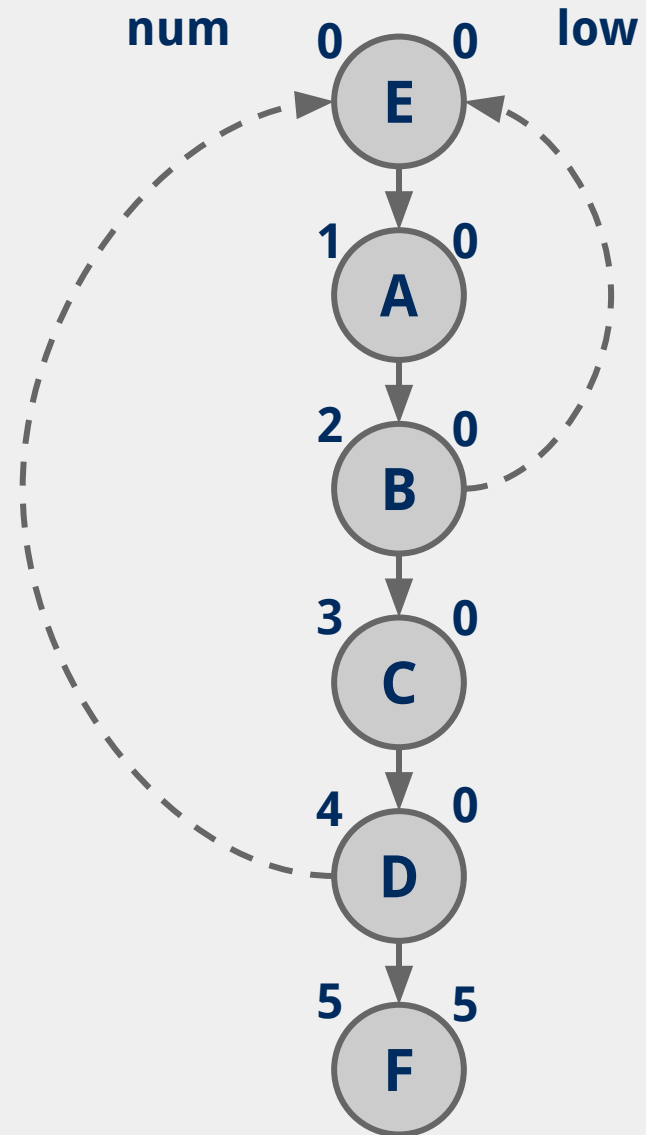
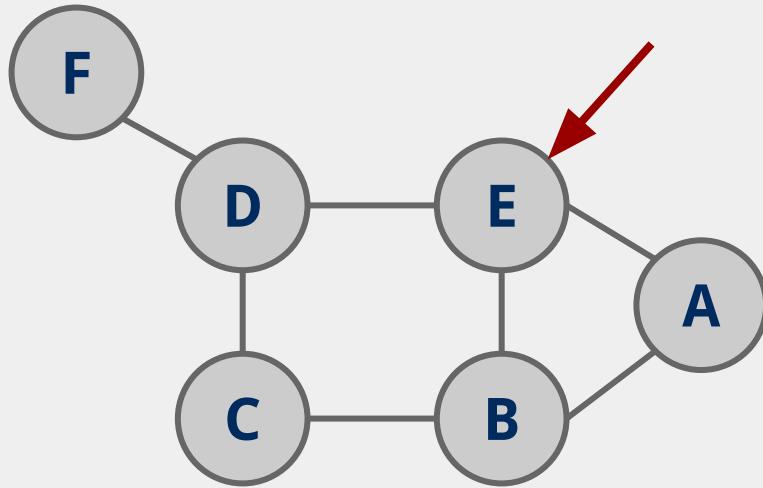
Biconnected graphs

- A *biconnected graph* has at least 2 distinct paths (no common edges or vertices) between all vertex pairs
- Any graph that is not biconnected has one or more *articulation points*
 - Vertices, that, if removed, will separate the graph
- Any graph that has no articulation points is biconnected
 - Thus we can determine that a graph is biconnected if we look for, but do not find any articulation points

Finding articulation points

- Variation on DFS
- Consider building up the spanning tree
 - Have it be directed
 - Create “back edges” when considering a node that has already been visited in constructing the spanning tree
 - Label each vertex v with with two numbers:
 - $\text{num}(v)$ = pre-order traversal order
 - $\text{low}(v)$ = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then at most one back edge
 - Min of:
 - $\text{num}(v)$
 - Lowest $\text{num}(w)$ of all back edges (v, w)
 - Lowest $\text{low}(w)$ of all spanning tree edges (v, w)

Finding articulation points example



So where are the articulation points?

- If any (non-root) vertex v has some child w such that $\text{low}(w) \geq \text{num}(v)$, v is an articulation point
- What about if we start at an articulation point?
 - If the root of the spanning tree has more than one child, it is an articulation point