VE281

Data Structures and Algorithms

Graphs

Review

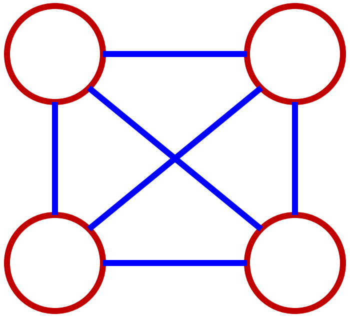
- 2-3 Tree: Removal
  - Swap the key with its in-order successor and then delete the key.
  - If the key is in a 2-node, removing the key violates the 2-3 tree property.
  - We restore the 2-3 tree property by either rotating keys or merging nodes.
- Graphs
  - Nodes, edges, simple graphs.

Outline

- Graph Basics
- Graph Representation
- Graph Search

Complete Graphs

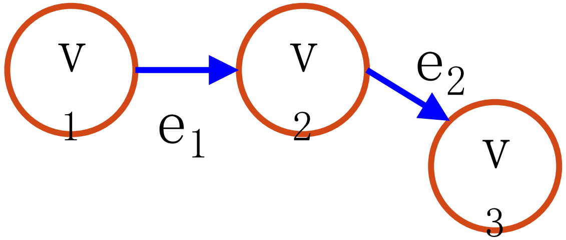
- A **complete graph** is a graph where every pair of nodes is directly connected.



- How many edges are there in a complete graph of N nodes?

Directed Graphs

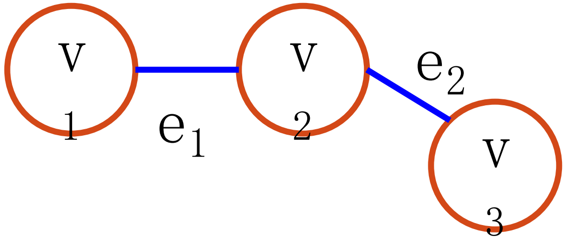
- **Directed graph** (digraph): edges are directional.



- Nodes incident to an edge form an **ordered** pair.
  - e = (v_1, v_2) means there is an edge **from** v_1 **to** v_2. However, there is no edge **from** v_2 **to** v_1.
- Examples: rivers and streams, one-way streets, customer-provider relationships.

Undirected Graphs

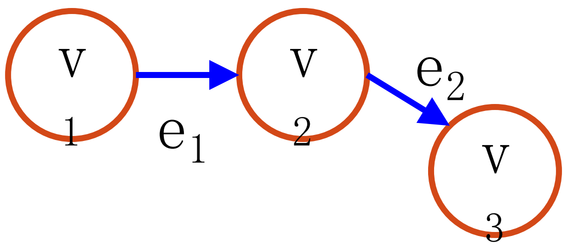
- **Undirected graph**: all edges have no orientation.



- There is no ordering of nodes on edges.
  - e = (v_1, v_2) means there is an edge **between** v_1 **and** v_2.
- Examples: friendship and two-way roads.

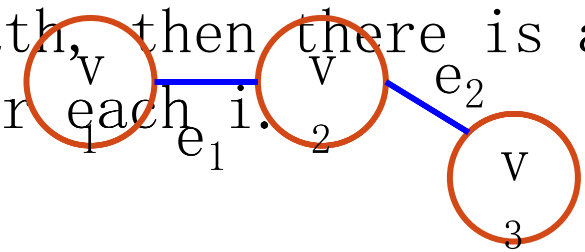
Paths

- A **path** is a series of nodes v_1, …, v_n that are connected by edges.
  - For a directed graph, if v_1, …, v_n is a path, then there is an edge **from** v_i **to** v_{i+1} for each i.



v_1, v_2, v_3 is a path.
v_3, v_2, v_1 is **not** a path.

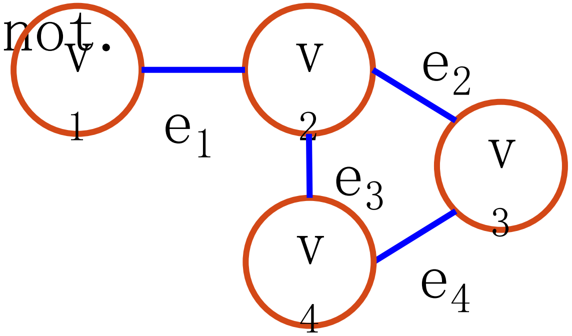
  - For an undirected graph, if v_1, …, v_n is a path, then there is an edge **between** v_i **and** v_{i+1} for each i.



v_1, v_2, v_3 is a path.
v_3, v_2, v_1 is **also** a path.

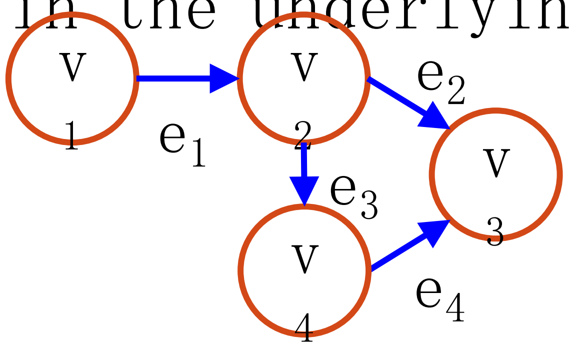
Simple Paths

- A **simple path** is a path with no node appearing twice
  - e.g., v_1, v_2, v_3 is a simple path; v_1, v_2, v_3, v_4, v_2 is not.



Connected Graphs

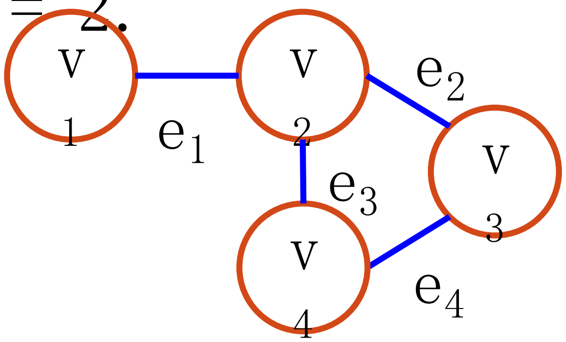
- A **connected graph** is a graph where a simple path exists between all pairs of nodes.
- A directed graph is **strongly connected** if there is a simple **directed path** between any pair of nodes.
- A directed graph is **weakly connected** if there is a simple path between any pair of nodes in the underlying undirected graph.



The directed graph is wea
connected, but not strong
connected.

Node Degree

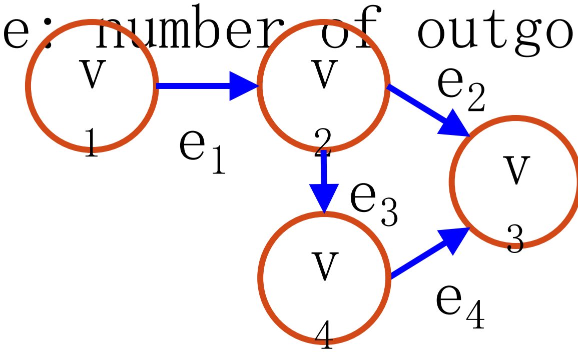
- The **degree** of a node is the number of edges incident to the node, e.g., degree(v_2) = 3, degree(v_3) = 2.



- What is the relationship between the sum of degrees of all nodes and the number of edges?
  - Sum(degrees) = 2 * Number(edges)

Node Degree for Directed Graphs

- For directed graphs, we differentiate between **incoming** edges and **outgoing** edges of a node. Thus we differentiate between a node's **in-degree** and its **out-degree**.
  - in-degree: number of incoming edges of a node
  - out-degree: number of outgoing edges of a node

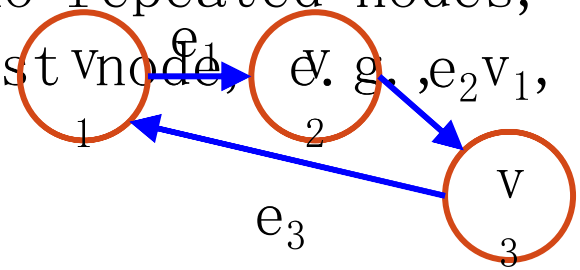


in-degree(v_2) = 1
out-degree(v_2) = 2

- Nodes with zero in-degree are **source** nodes, e.g., v_1.
- Nodes with zero out-degree are **sink** nodes, e.g., v_3.

Cycles and Directed Acyclic Graphs

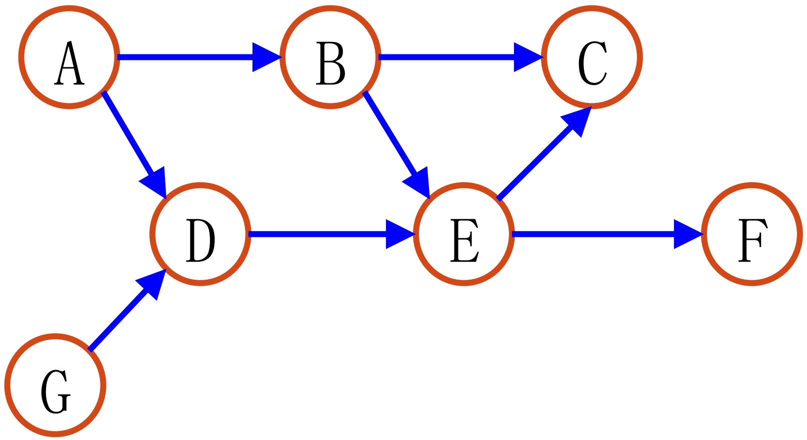
- A **cycle** is a path starting and finishing at the same node.
  - A self-loop is a cycle of length 1.
  - A **simple cycle** has no repeated nodes, except the first and the last node, e.g., v_1, v_2, v_3, v_1.

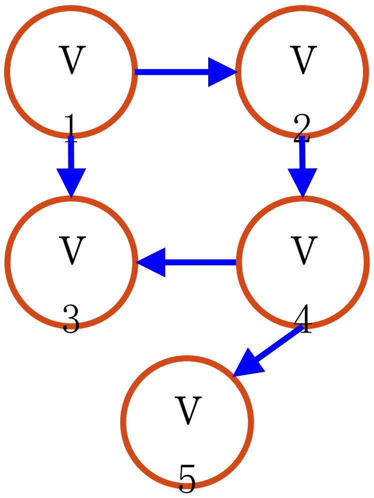


- A graph with no cycle is called an **acyclic graph**.
- A directed graph with no cycles is called a **directed acyclic graph**, or **DAG** for short.

Directed Acyclic Graphs (DAG)

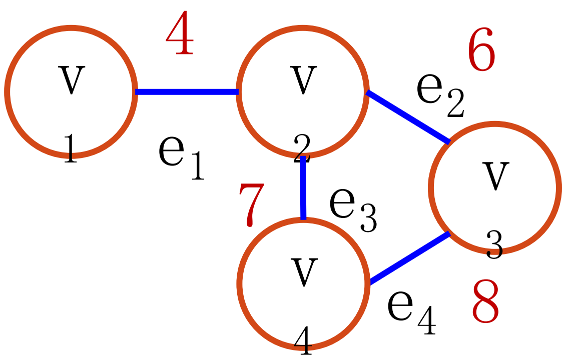
- Are the following graphs DAGs?





Weighted Graphs

- Weighted graph: edges of a graph may have different costs or weights.
  - For example, the weights on edges represent the distance between two nodes.



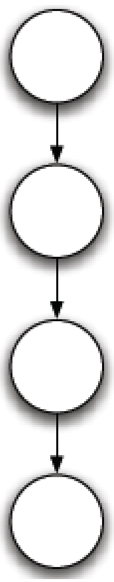
Graph Size and Complexity

- Whereas a BST increases height by extending a single path, a 2-3 tree increases height **globally** by **raising** the root.

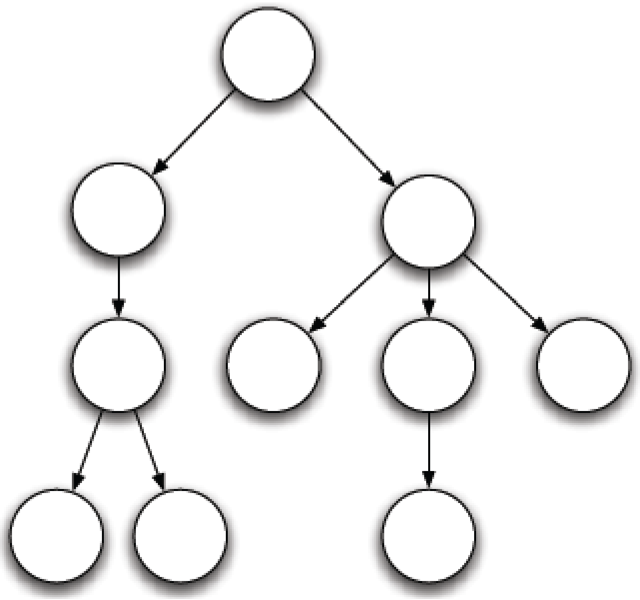
- Therefore, all of the leaves of a 2-3 tree are at the same level.
  - The 2-3 tree is always balanced.

- What is the worst case time complexity?
  - $O(\log N)$

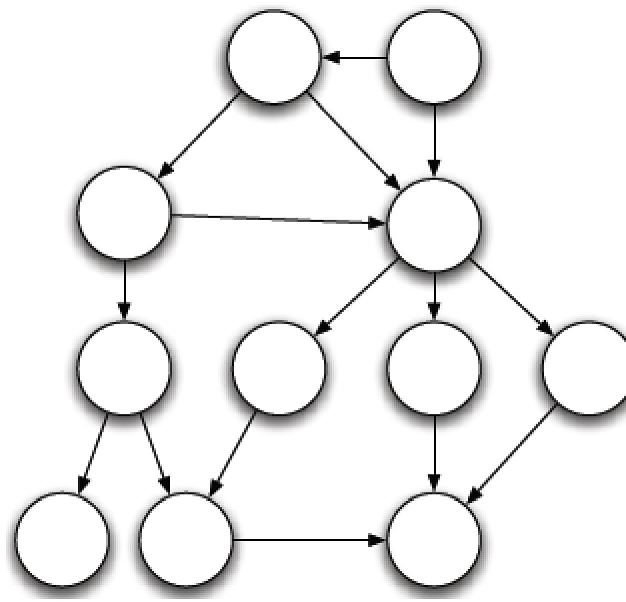
Linked Lists, Trees, and Graphs



Linked List

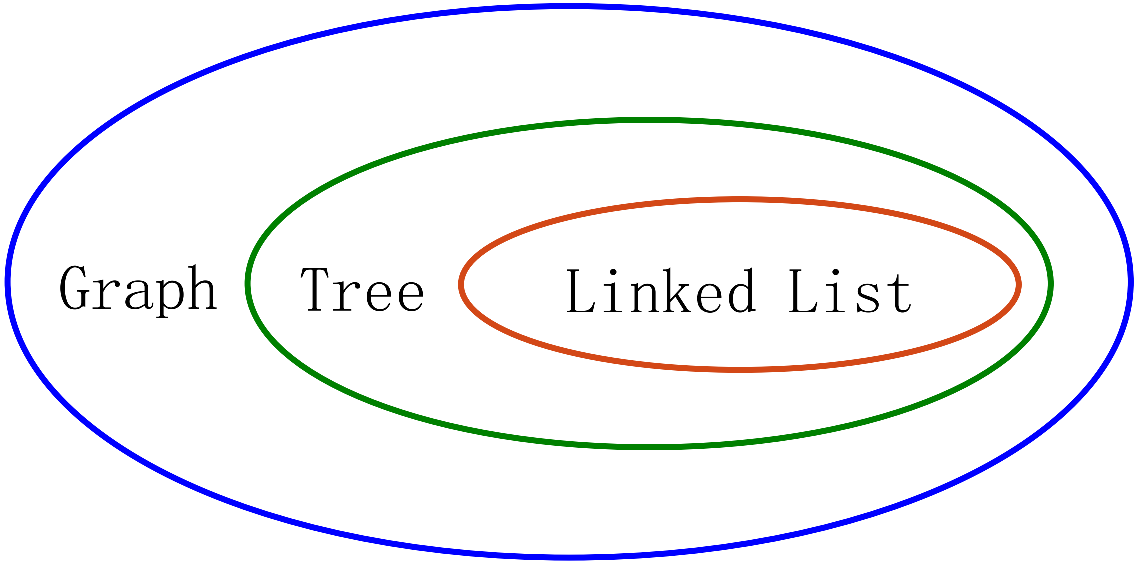


Tree



Graph

Linked Lists, Trees, and Graphs



Sample Graph Problems

- Path finding problems
  - Find if there exists a path between two given nodes.
  - Find the shortest path between two given nodes.

- Connectedness problems
  - Find if the graph is a connected graph.

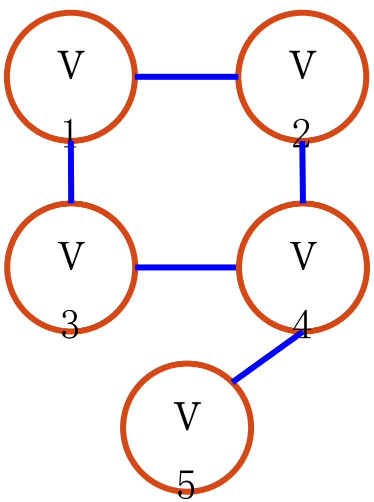
Outline

- Graph Basics
- Graph Representation
- Graph Search

Graph Representation

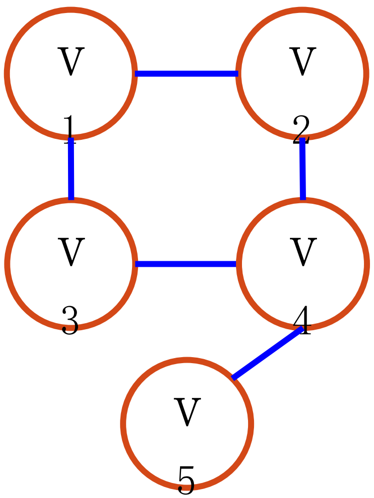
Adjacency Matrix

- Adjacency matrix: a $|V| \times |V|$ matrix representation of a graph.
- $A(i,j) = 1$, if (v_i, v_j) is an edge; otherwise $A(i,j) = 0$.



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

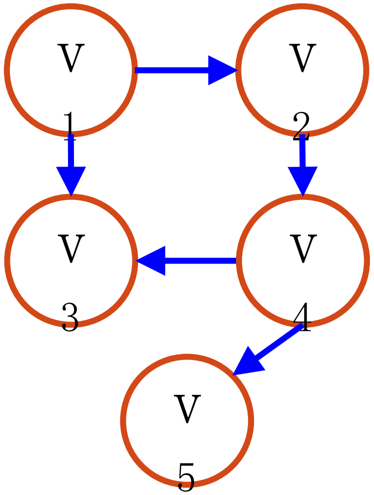
Adjacency Matrix for Undirected Graph

- 

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

- Diagonal entries are zero.
- The matrix is **symmetric**, i.e., $A(i,j) = A(j,i)$ for all i and j.
- Number of ones in the matrix is twice the number of edges.

Adjacency Matrix for Directed Graph

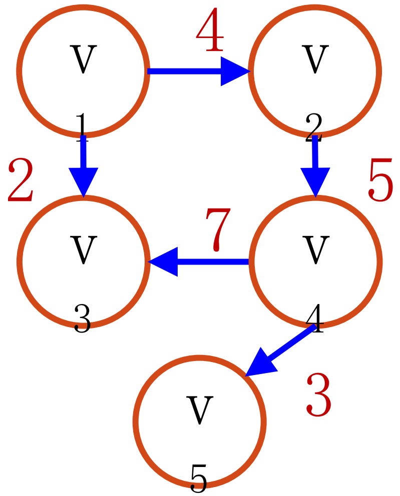


	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	0	0	0	0
4	0	0	1	0	1
5	0	0	0	0	0

- Diagonal entries are zero.
- The matrix need not be symmetric.
- Number of ones in the matrix equals the number of edges.

Adjacency Matrix for Weighted Graph

- If (v_i, v_j) is an edge and its weight is w_{ij}, then $A(i,j) = w_{ij}$; otherwise $A(i,j) = \infty$.



	1	2	3	4	5
1	∞	4	2	∞	∞
2	∞	∞	∞	5	∞
3	∞	∞	∞	∞	∞
4	∞	∞	7	∞	3
5	∞	∞	∞	∞	∞

Adjacency Matrix

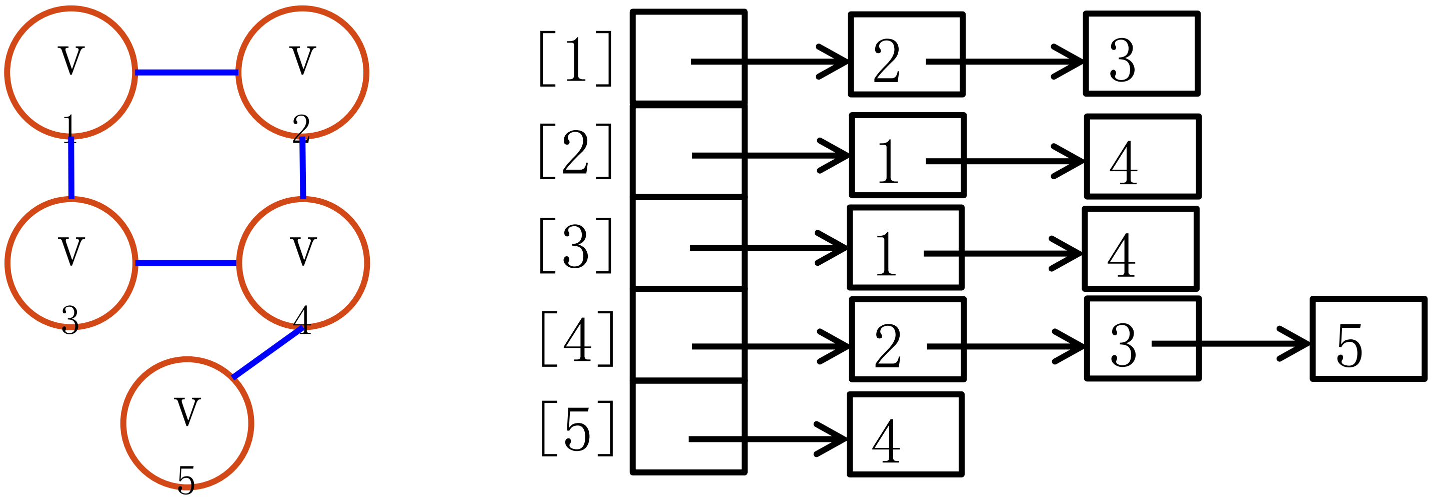
Properties

- Space complexity: $|V|^2$ units
  - For an unweighted graph, $|V|^2$ **bits**.
  - For an undirected graph, may store only the lower or upper triangle. Thus, $(|V| - 1)|V|/2$ units.

- What is the time complexity for finding if node v_i is adjacent to node v_j?
  - $O(1)$
- What is the time complexity for finding all nodes adjacent to a given node v_i?
  - $O(|V|)$

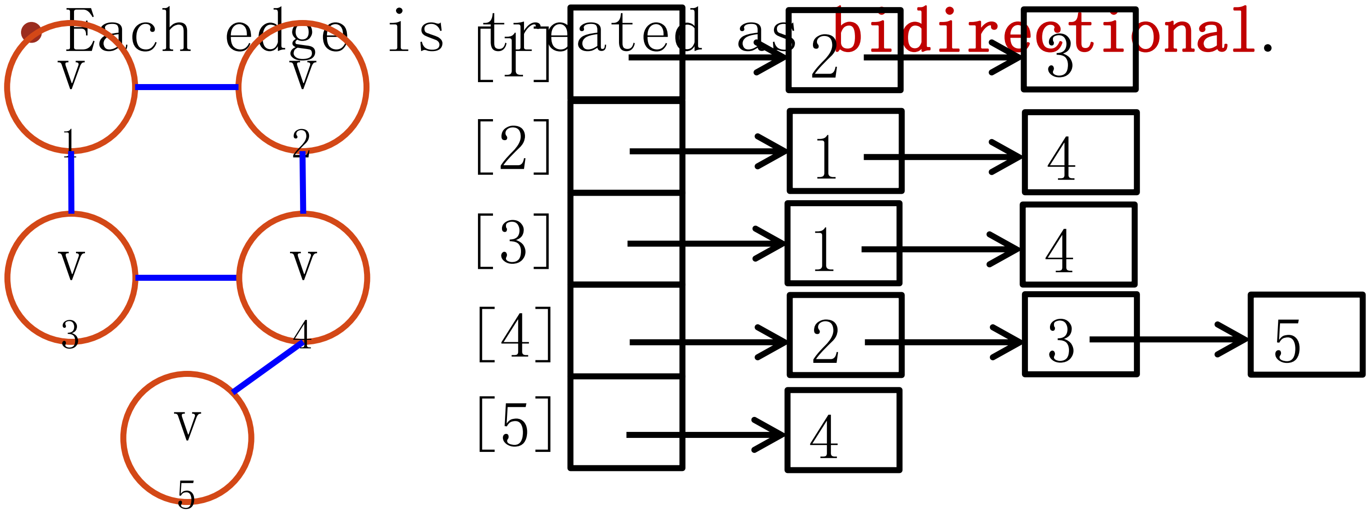
Graph Representation
Adjacency List

- Adjacency list: an array of $|V|$ linked lists.
  - Each array element represents a node and its linked list represents the node's neighbors.



Graph Representation
Adjacency List

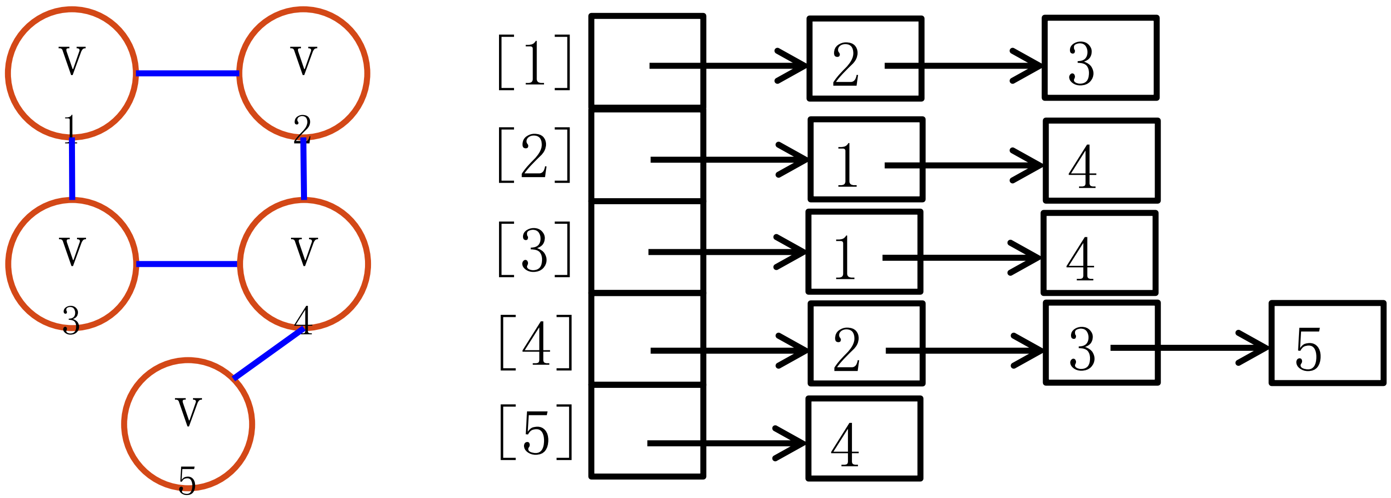
- Each edge in an undirected graph is represented twice.
  - Each edge is treated as bidirectional.



- Each edge in a directed graph is represented once.
- Weighted graph stores edge weight in linked-list node.

Adjacency List

Properties

- 

- What is the space complexity? $O(|E| + |V|)$
- What is the worst case time complexity for checking if node v_i is adjacent to node v_j? $O(|V|)$
- What is the worst case time complexity for finding all nodes adjacent to a given node v_i? $O(|V|)$

Comparison of Graph Representation

- Worst case time complexity for two common operations:

1. Determine whether v_i is adjacent to v_j
   - Adjacency matrix: $O(1)$; Adjacency list: $O(|V|)$
2. Determine all the nodes adjacent to v_i
   - Both adjacency matrix and adjacency list: $O(|V|)$

- Adjacency list often requires less space than adjacency matrix.
- Dense graphs are more efficiently represented as adjacency matrices and sparse graphs as adjacency lists.

Outline

- Graph Basics
- Graph Representation
- Graph Search

Graph Search

- A node u is **reachable** from a node v if and only if there is a path from v to u.
- A graph search method starts at a given node v and visits **every** node that is **reachable** from v.
- Many graph problems are solved using a search method.
  - Find a path from one node to another.
  - Find if the graph is connected.
- Commonly used search methods:
  - Depth-first search.
  - Breadth-first search.

Depth-First Search (DFS)

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

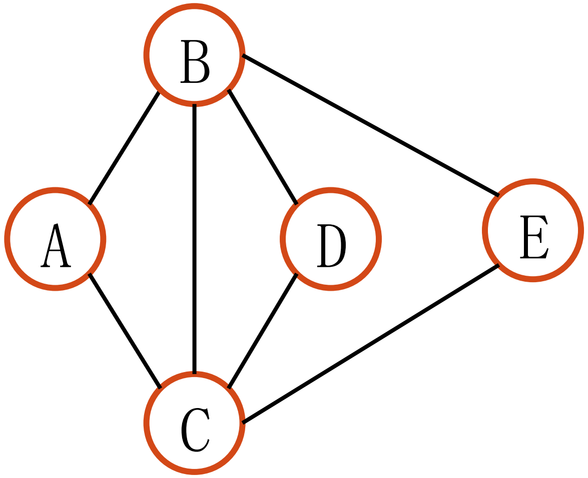
- How to mark a node "visited"?
  - Keep a "visited" field in the node, or
  - Keep a global "visited" array, one entry per node:
    - Initially mark all entries false.
    - When a node is visited, set its entry to true.
    - Check this array to avoid visiting previously visited node.

Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

Start from A.
DFS(A)

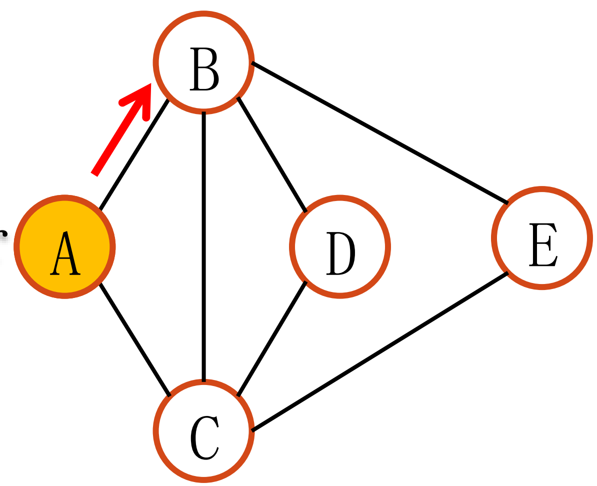


Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(A):
Mark A as visited;
Choose A' s neighbor
DFS(B)

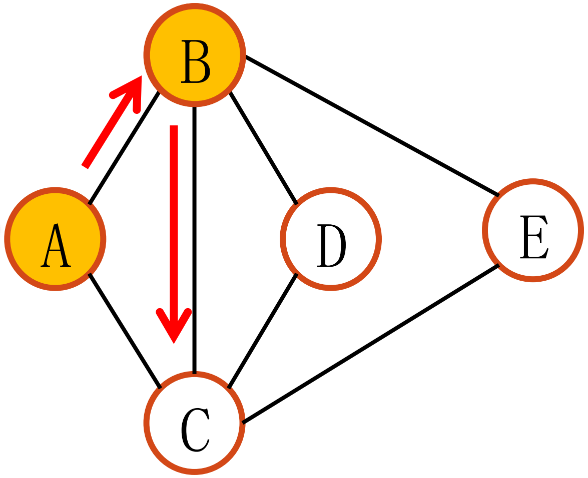


Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(B):
Visit and mark B;
Choose B’s neighbor
DFS(C)

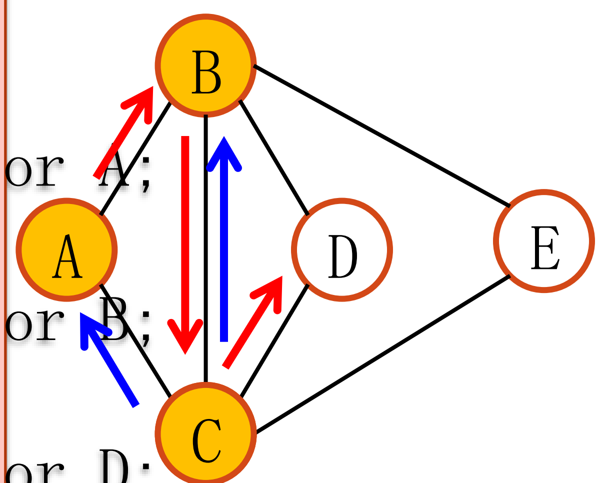


Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(C):
Visit and mark C;
Choose C’s neighbor A;
A is visited;
Choose C’s neighbor B;
B is visited;
Choose C’s neighbor D;
DFS(D)

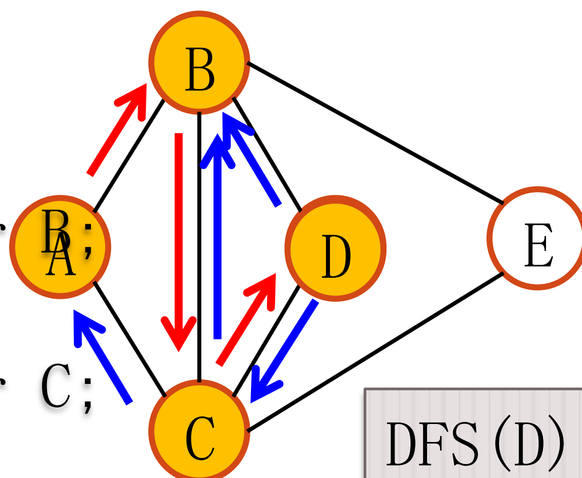


Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(D):
Visit and mark D;
Choose D’ s neighbor B;
B is visited;
Choose D’ s neighbor C;
C is visited;



DFS(D) finished.
Back to its caller DFS

Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

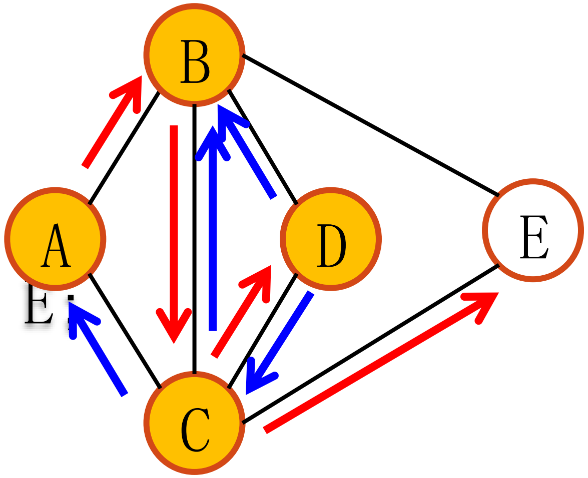
DFS(C):

…

DFS(D);

Choose C’ s neighbor E;

DFS(E)

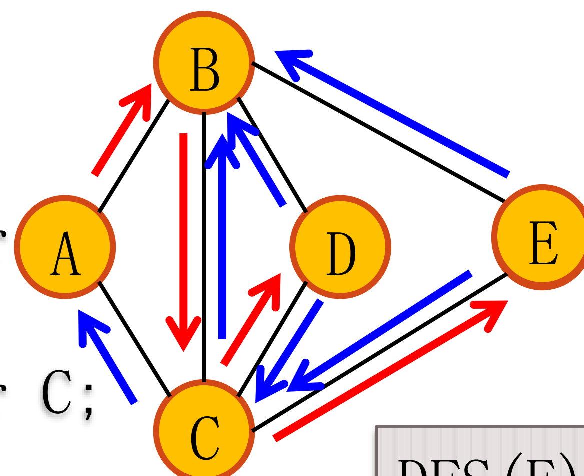


Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(E):
Visit and mark E;
Choose E’ s neighbor B;
B is visited;
Choose E’ s neighbor C;
C is visited;



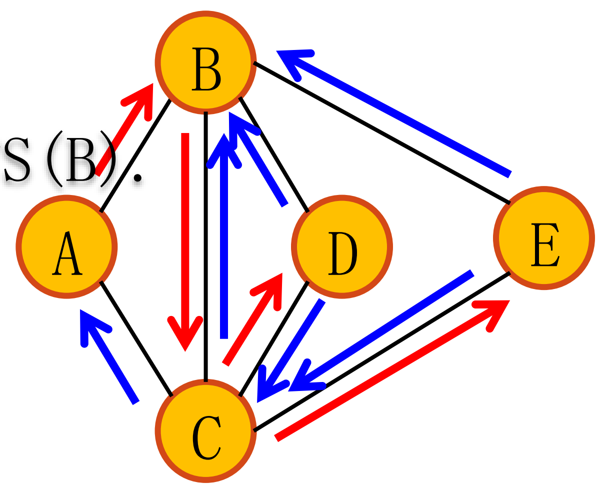
DFS(E) finished.
Back to its caller DF

Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

DFS(C) finished.
Back to its caller DFS(B).



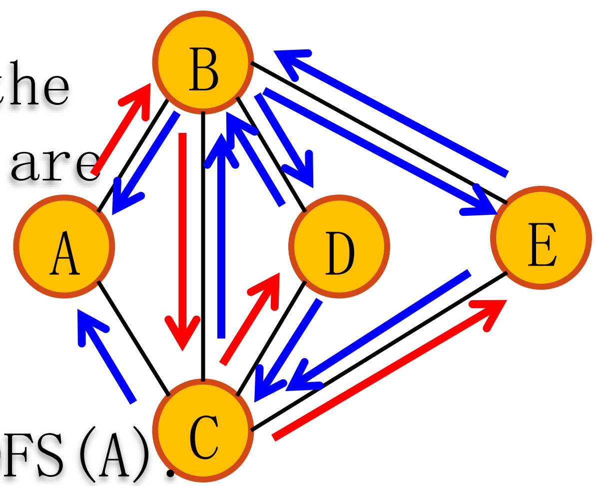
Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

Now in DFS(B), all the remaining neighbors are visited.

DFS(B) finished.
Back to its caller DFS(A).



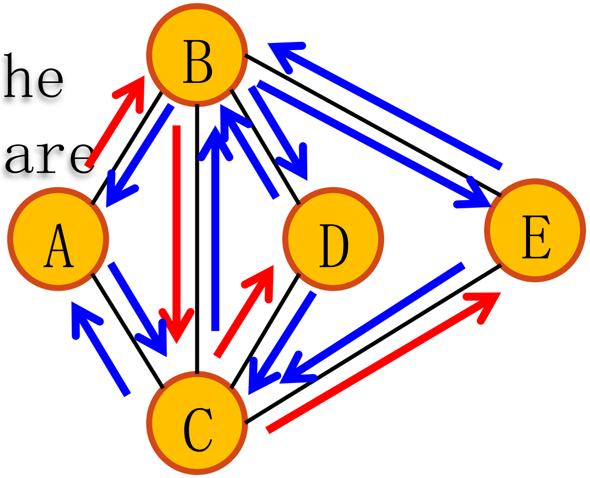
Depth-First Search (DFS)

Example

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

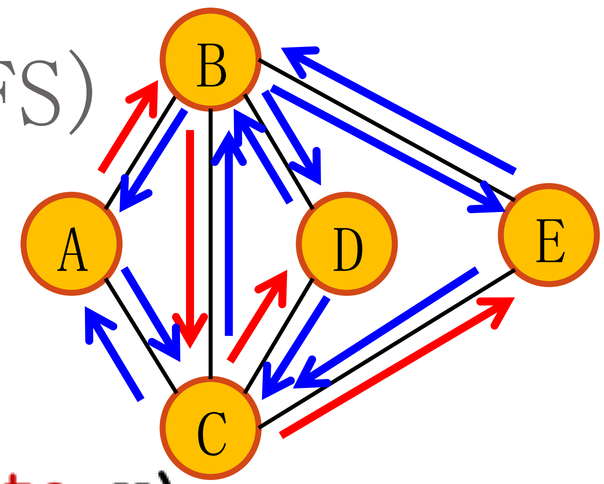
Now in DFS(A), all the remaining neighbors are visited.

DFS(A) finished.



Depth-First Search (DFS)

Time Complexity

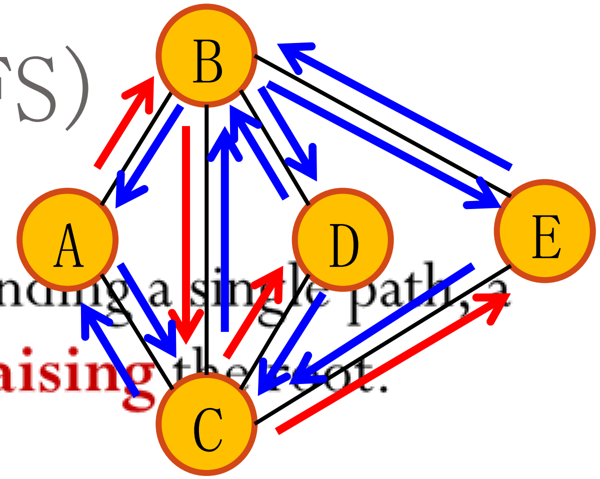



```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

- If graph is implemented as **adjacency matrix**:
  - Visit each node exactly once: $O(V)$.
  - The row of each node in the adjacency matrix is scanned once: $O(|V|)$ for each node.
  - Total running time: $O(|V|^2)$.

Depth-First Search (DFS)

Time Complexity



- Whereas a BST increases height by extending a single path, a 2-3 tree increases height **globally** by **raising** the root.

- Therefore, all of the leaves of a 2-3 tree are at the same level.
  - The 2-3 tree is always balanced.

- What is the worst case time complexity?
  - $O(\log N)$

Depth-First Search (DFS)

Summary

```
DFS(v) {
  visit v;
  mark v as visited;
  for(each node u adjacent to v)
    if(u is not visited) DFS(u);
}
```

- Explore the graph **as far as possible** along edges, before backtracking.
- When backtracking, return to the **most recent** node that hasn’t been fully explored.
- DFS can also be implemented non-recursively using a stack.