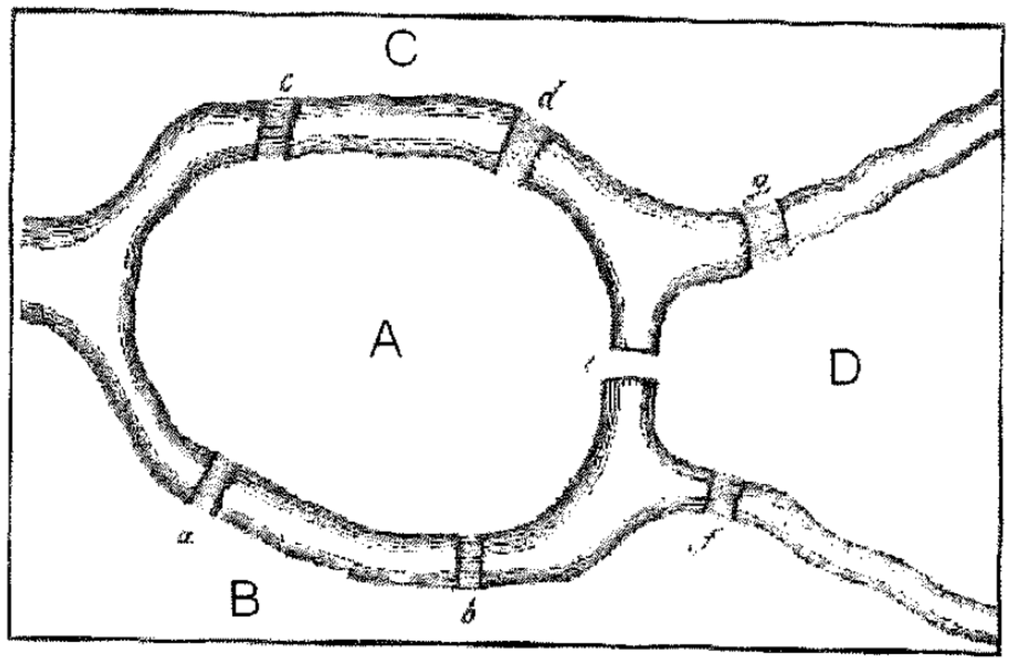
Graph

Prologue: Eulerian Path

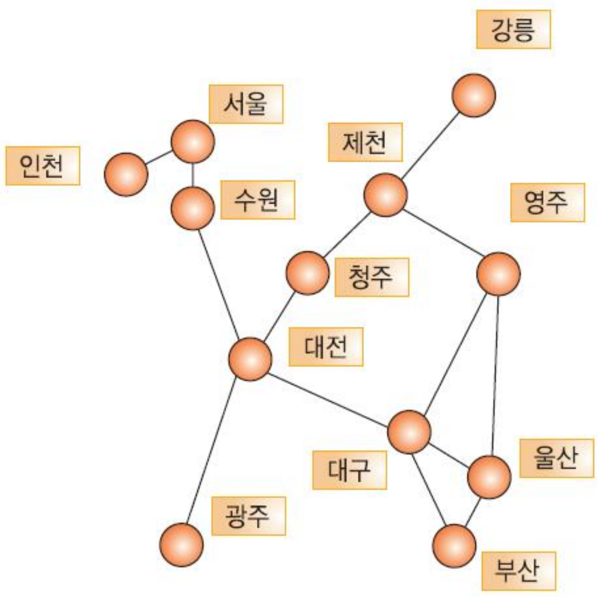
- Seven bridges of Königsberg problem
  - Q: Is it possible to take a walk starting by any of the four parts of land, **crossing each one of the bridges just once**?



  - A: This problem was resolved by **Leonhard Euler** in 1736.

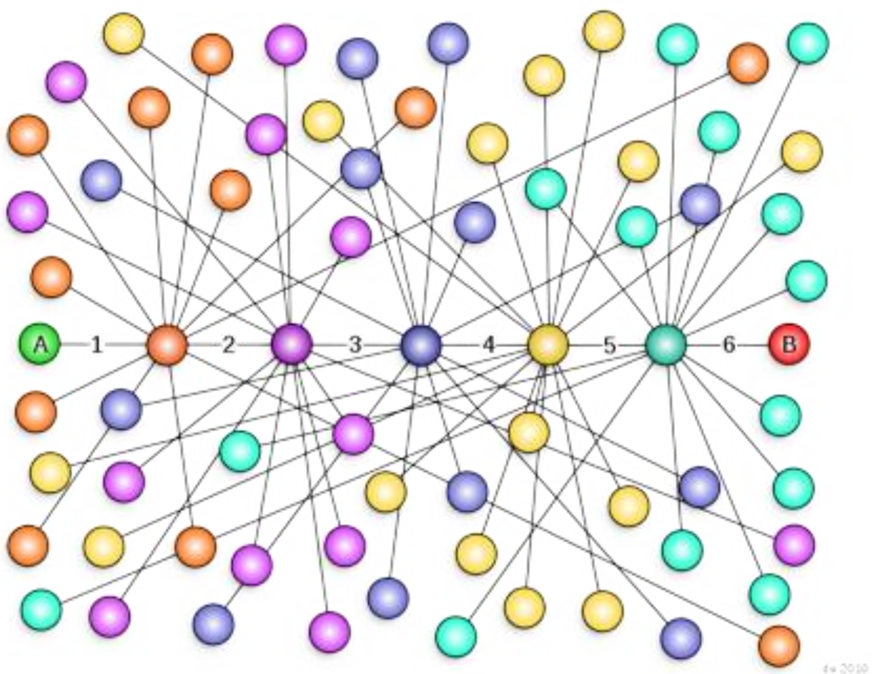
What is Graph?

- Definition
  - A collection of **nodes** and **arcs** to represent a structure
    - **Objects**: A finite set of **nodes** (or **vertices**, **points**)
    - **Relationship**: A finite set of **arcs** (or **edges**, **links**)

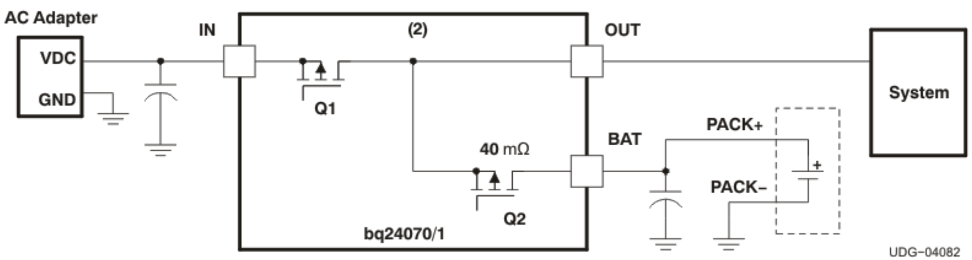


Example: Six Degrees of Separation

- Everyone in the world is six or fewer steps away from each other.
  - A chain of "**a friend of a friend**" can be made to connect any two people in a **maximum of six steps**.

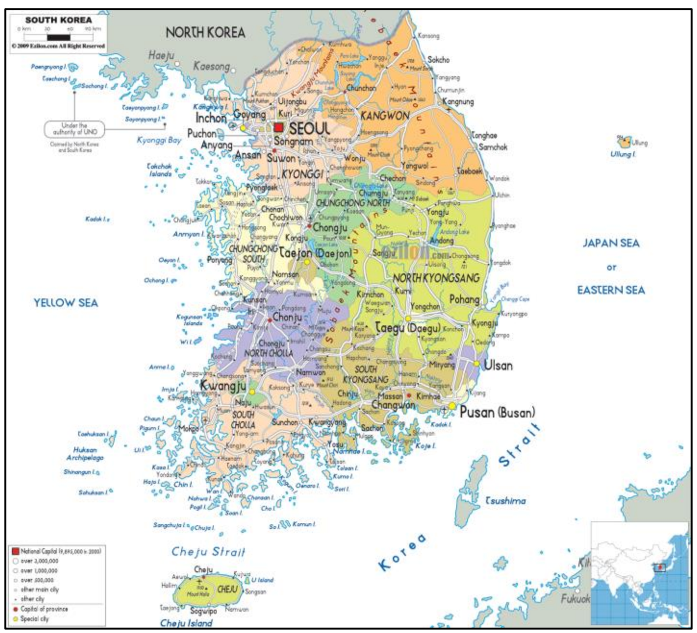


Applications of Graphs

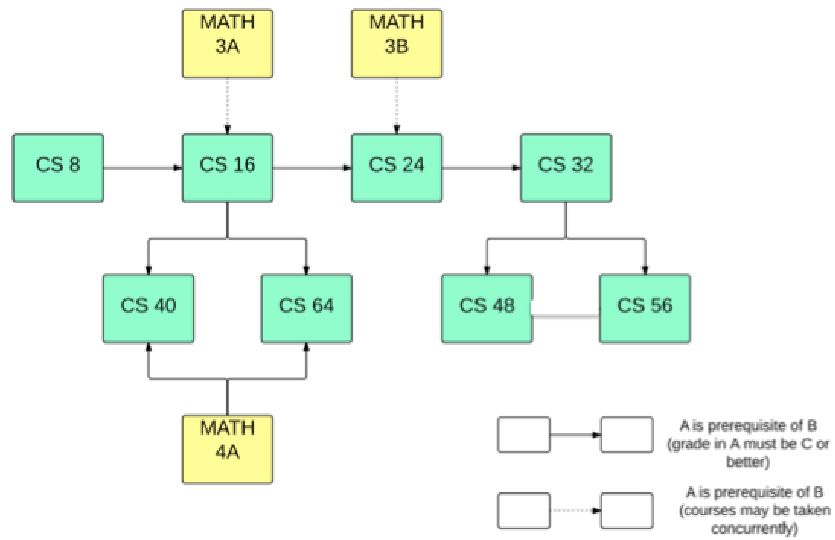


IC circuit path

Computer network



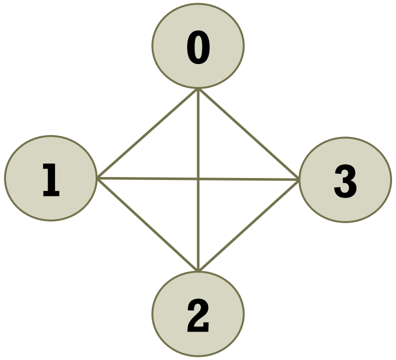
Map & road network



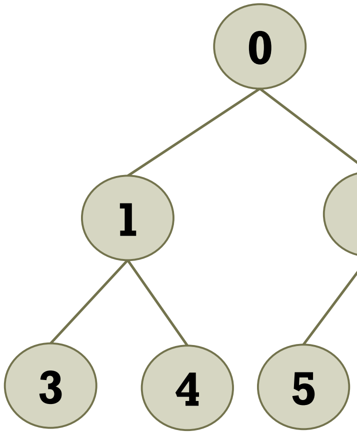
CS prerequisite table

Graph: Formal Definition

- Formally, a graph is denoted by $\boldsymbol{G = (V, E)}$.
  - $\boldsymbol{V}$ **: a set of vertices**
  - $\boldsymbol{E}$ **: a set of edges**, a set of 2-elements of $\boldsymbol{V}$
    - $E \subseteq \{(u, v): u, v \in V\}$



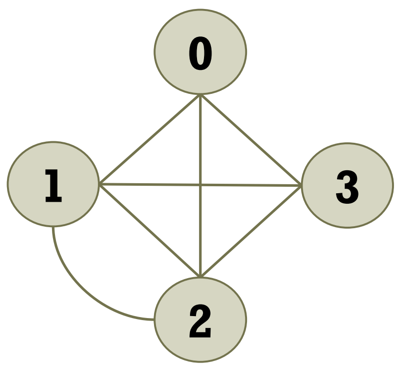
- Example
  - $G_1 = (V, E)$, where
    - $V = \{0, 1, 2, 3\}$
    - $E = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

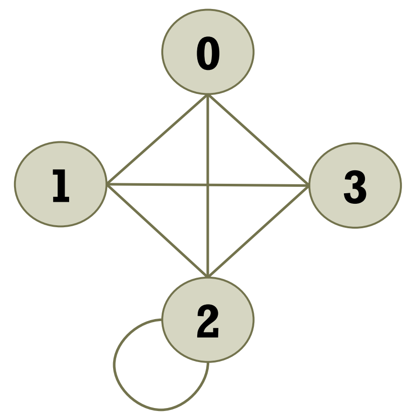


  - $G_2 = (V, E)$, where
    - $V = \{0, 1, 2, 3, 4, 5\}$
    - $E = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5)\}$

Simple Graph

- In general, a graph may have parallel edges and self loops.

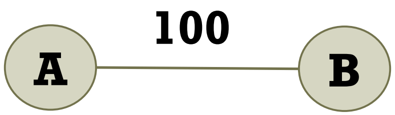




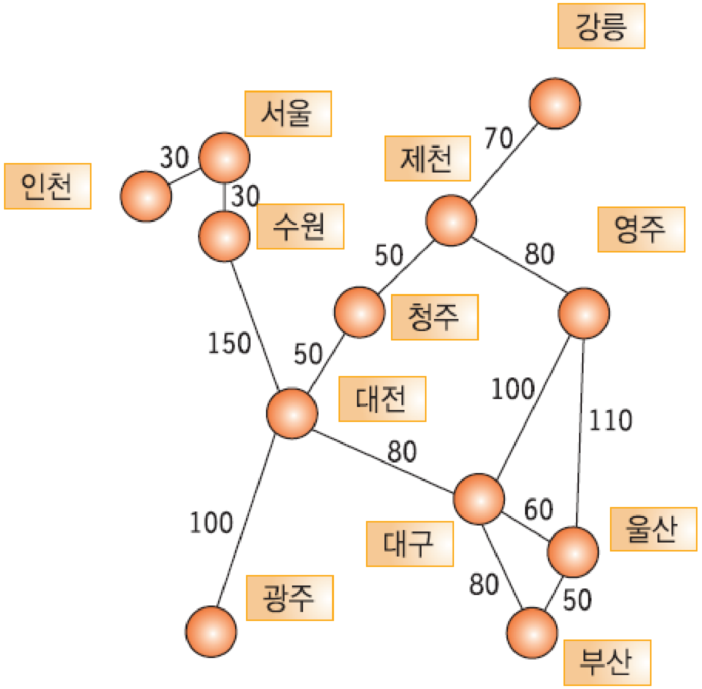
- Note: Simple graphs do not have parallel edges and self-loops.
  - Assume that the graph is simple unless otherwise specified.

Weighted Graph

- Edges may be associated with “weighted” values

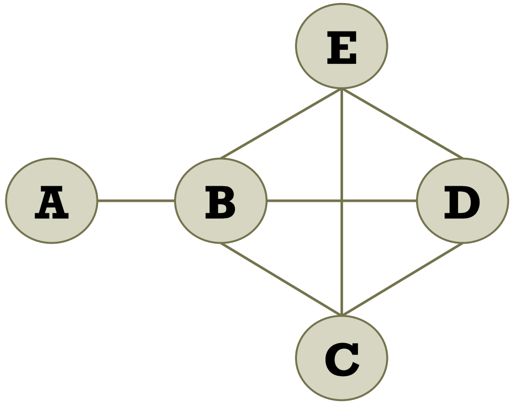


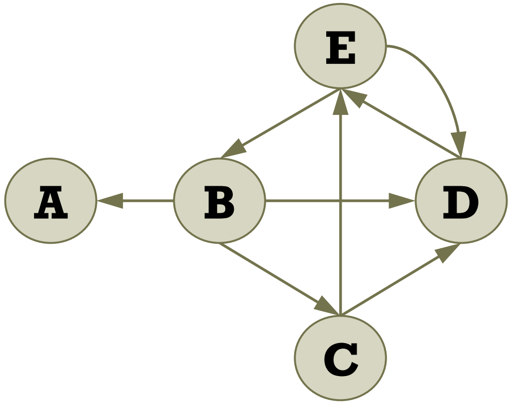
- Example: Road network
  - Vertices: City
  - Edges: Connection between two cities
    - Weight: The length of the road for edges



Undirected vs. Directed Graphs

- Undirected graph
  - Vertices on edges form **unordered** pairs.
    - The order of vertices in edges is not important.
    - (u, v) means there is an edge **between** u and v.
- Directed graph (aka. digraph)
  - Edges have a direction.
  - Vertices on edges form **ordered** pairs.
    - The order of vertices in edge is **important**.
    - (u, v) means there is an edge **from** u **to** v.

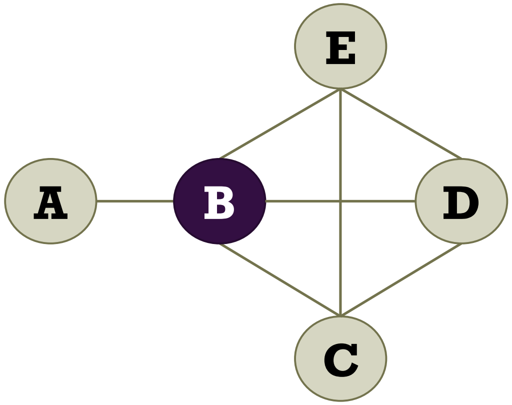




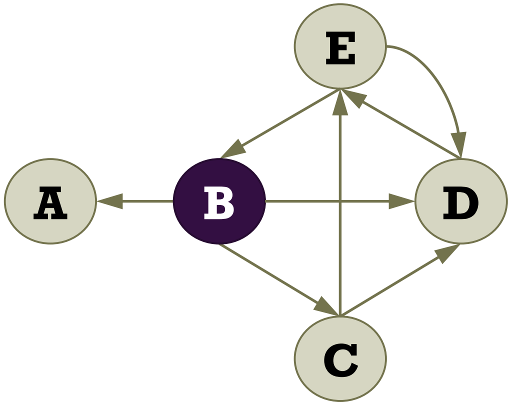
Degree of Vertex

- Degree of a vertex in undirected graphs
  - The number of edges incident to the vertex
  - The number of edges = (sum of degree of all vertices) / 2

- For digraphs, there are two types: indegree and outdegree.



The degree of B is 4.



The indegree of B is 1.

The outdegree of B is 3.

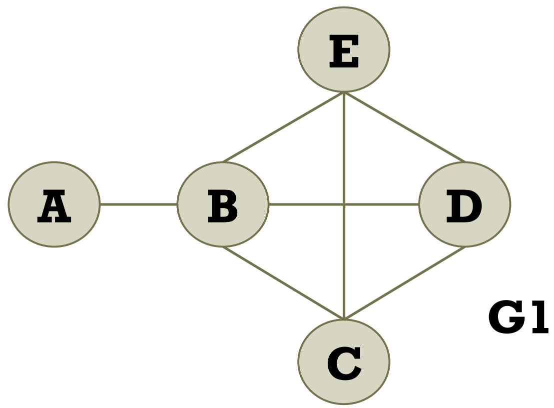
Paths in Graph

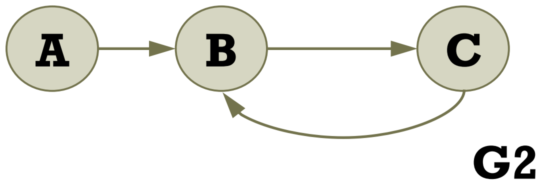
- Adjacent vertex
  - A vertex $\boldsymbol{u}$ is said to be **adjacent** to another vertex $\boldsymbol{v}$ in $\boldsymbol{G}$ if the graph contains an edge $(\boldsymbol{u}, \boldsymbol{v})$.

- Path from a vertex $\boldsymbol{u}$ to another vertex $\boldsymbol{v}$ in $\boldsymbol{G}$
  - **A sequence of vertices**, $u,\ v_1,\ v_2,\ \dots,\ v_k, v$ where $(u, v_1), (v_1, v_2), \dots, (v_k, v)$ are edges in G
  - **Length of a path:** The number of edges between $\boldsymbol{u}$ and $\boldsymbol{v}$
  - **Simple path**: A path that does not have **redundant** edges
  - **Cycle**: A path where the starting and ending vertices are **same**.

Paths in Graph

- Example
  - Adjacent vertices of B in G1 are A, C, D, and E.
  - The length of path A-B in G1 is 1.
  - The length of path A-B-D in G1 is 2.





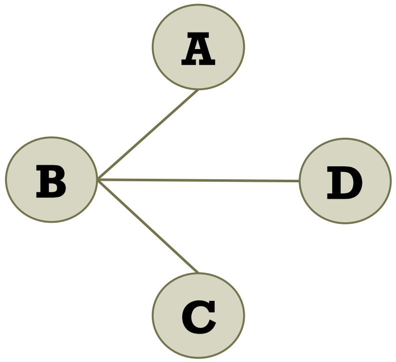
  - The path A-B-E-C-B-E in G1 is not a simple path.
  - The path B-E-D-B in G1 is a cycle.
  - The path B-C-B in G2 is a cycle.

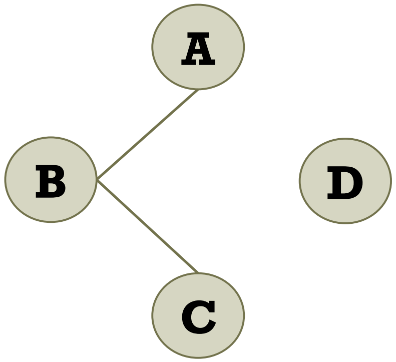
Connected Graph

- In undirected graph $\boldsymbol{G}$, $\boldsymbol{u}$ and $\boldsymbol{v}$ are connected if there is a path from $\boldsymbol{u}$ and $\boldsymbol{v}$.
  - If any vertex in $\boldsymbol{G}$ is connected to any other vertices, it is said that **graph $\boldsymbol{G}$ is connected**.

- Connected component
  - A **maximal connected subgraph**

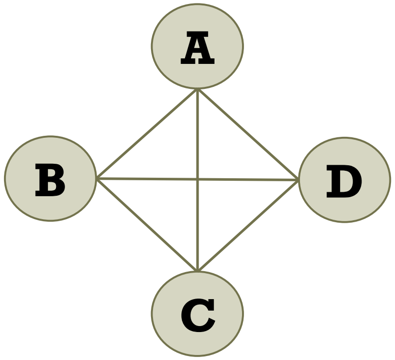
- What is a connected component for each graph?





Complete Graph

- A graph of which all vertices are adjacent to any other vertices
  - Undirected graph: $\boldsymbol{n(n-1)/2}$ **edges**
  - Directed graph: $\boldsymbol{n(n-1)}$ **edges**

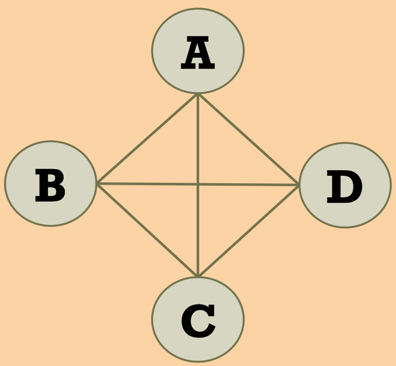


- Dense graph
  - Has many edges $|E| \approx |V|^2$.
  - Represented as an **adjacency matrix**.

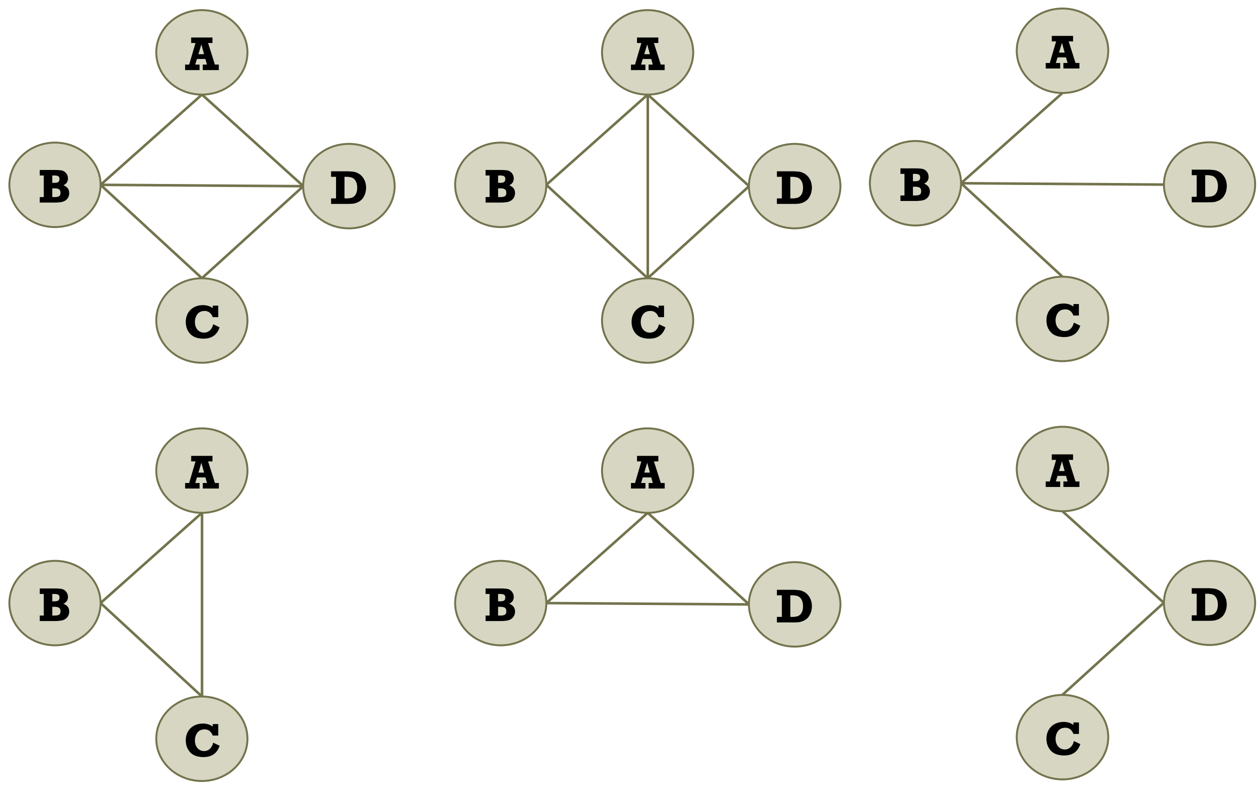
- Sparse graph
  - Has few edges $|E| \ll |V|^2$ or $|E| \approx |V|$.
  - Represented as an **adjacency list**.

Subgraph

- Subgraph $\boldsymbol{G'} = (\boldsymbol{V'}, \boldsymbol{E'})$ of graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$
  - A graph such that such that $\boldsymbol{V'} \subseteq \boldsymbol{V}$ and $\boldsymbol{E'} \subseteq \boldsymbol{E}$



graph G



Possible subgraphs of G

Complexity of Graph Algorithms

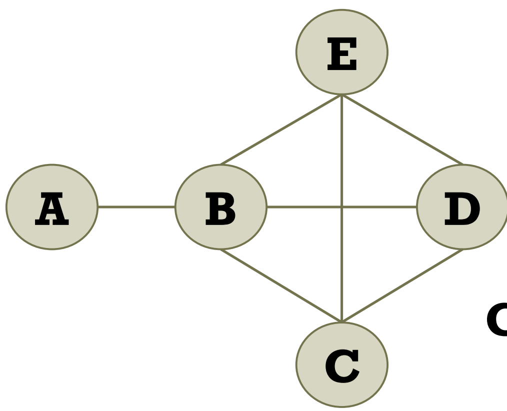
- The complexity of graph algorithms is typically defined in terms of
  - Number of vertices $|V|$, Number of edges $|E|$, or both

- Graph representation
  - Adjacency matrix
  - Adjacency list

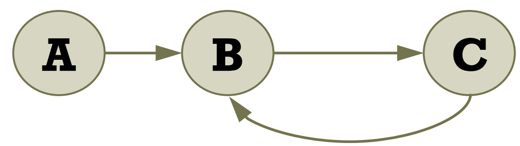
- Graph traversal
  - Depth first search (DFS)
  - Breadth first search (BFS)

Adjacency Matrix

- Allocate $|V| \times |V|$ matrix M.
  - $M[i][j] = 1$ if there is an edge between v_i and v_j
  - $M[i][j] = 0$ if there is not an edge between v_i and v_j



G1



G2

$$
\begin{array}{c} \\ \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \\ \mathbf{D} \\ \mathbf{E} \end{array}
\begin{array}{c} \begin{array}{ccccc} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \end{array} \\
\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}
$$

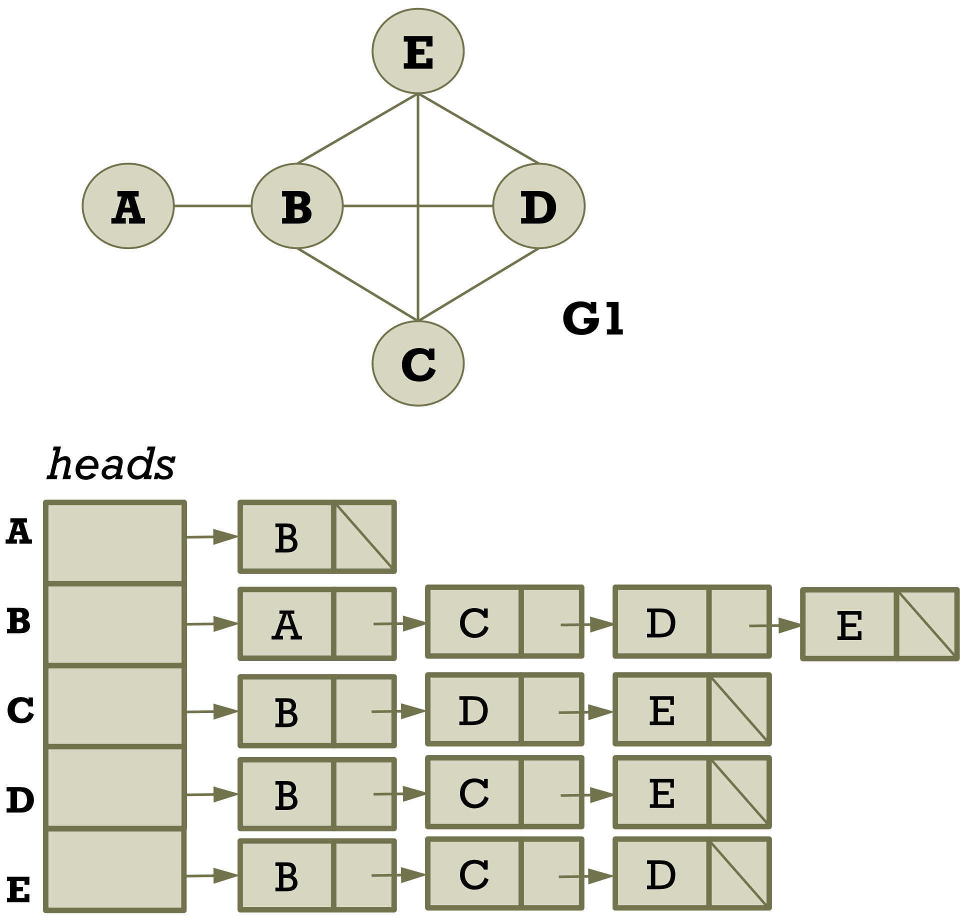
$$
\begin{array}{c} \\ \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \end{array}
\begin{array}{c} \begin{array}{ccc} \mathbf{A} & \mathbf{B} & \mathbf{C} \end{array} \\
\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{array}
$$

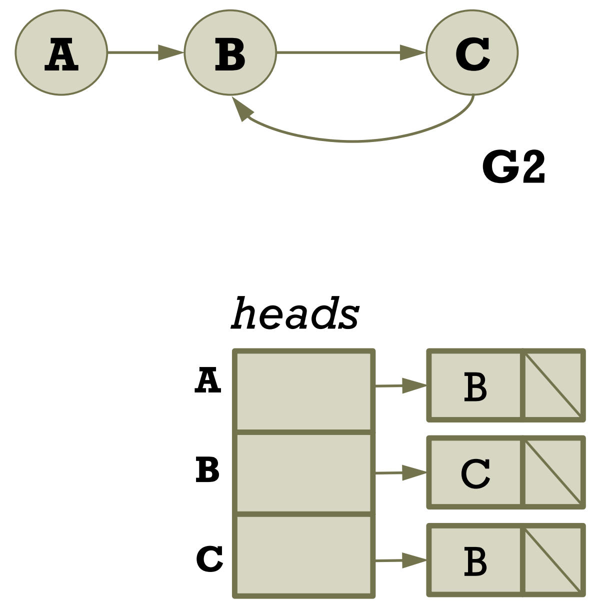
Adjacency Matrix

- Time Complexity
  - Is there an edge between v_i and v_j? $O(1)$
  - How many edges are in G? $O(|V|^2)$
  - What is the out-degree of v_i? $O(|V|)$
  - What is the in-degree of v_i? $O(|V|)$
- Space Complexity: $O(|V|^2)$
- If there is the small number of edges in G,
  - Adjacency matrix is sparse.
  - Space is wasted.

Adjacency List

- Allocate an array called *heads*
  - *heads*[i] points to a linked list of nodes connected to v_i





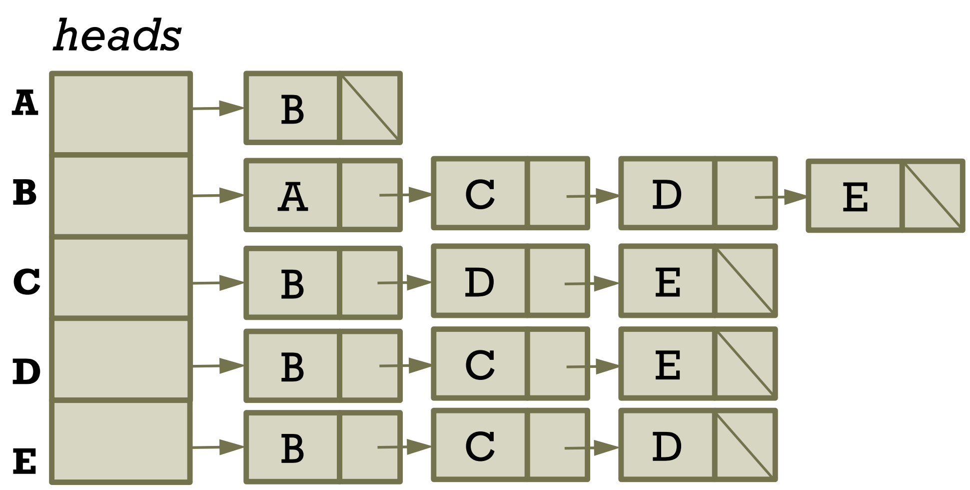
Adjacency List

- Time Complexity
  - Is there an edge between v_i and v_j? $O(|V|)$
  - How many edges are in G? $O(|E|)$
  - What is the out-degree of v_i? $O(|V|)$
  - What is the in-degree of v_i? $O(|E|)$
- Space Complexity: $O(|V| + |E|)$
- The adjacency list is effective for a sparse graph.

Adjacency List Implementation

- Node representation in a graph
  - ***GNode***: it is connected from each head pointer.
  - ***Graph***: heads are *GNode* pointers.

```
typedef struct _GNode
{
    int id;
    struct _GNode* next;
} GNode;

typedef struct
{
    int num;
    GNode** heads;
} Graph;
```



Adjacency List Implementation

- Operations

```
// Create a graph.
void CreateGraph(Graph* pgraph, int num);
// Destroy a graph.
void DestroyGraph(Graph* pgraph);
// Add an undirected edge from src to dest.
void AddEdge(Graph* pgraph, int src, int dest);
// Print a graph for each vertex.
void PrintGraph(Graph* pgraph);

// Depth first search
void DFS(Graph* pgraph);
// Breadth first search
void BFS(Graph* pgraph);
```

Create and Destroy Operations

```
void CreateGraph(Graph* pgraph, int num)
{
     pgraph->num = num;
     pgraph->heads = (GNode **)malloc(sizeof(GNode*)* num);
     for (int i = 0; i < num; i++) {
          // Make a dummy node.
          pgraph->heads[i] = (GNode *)malloc(sizeof(GNode));
          pgraph->heads[i]->next = NULL;
     }
}

void DestroyGraph(Graph* pgraph)
{
     for (int i = 0; i < pgraph->num; i++) {
          GNode* cur = pgraph->heads[i];
          while (cur != NULL) {
               GNode* temp = cur;
               cur = cur->next;
               free(temp);
          }
     }
     free(pgraph->heads);
}
```

AddEdge Operation

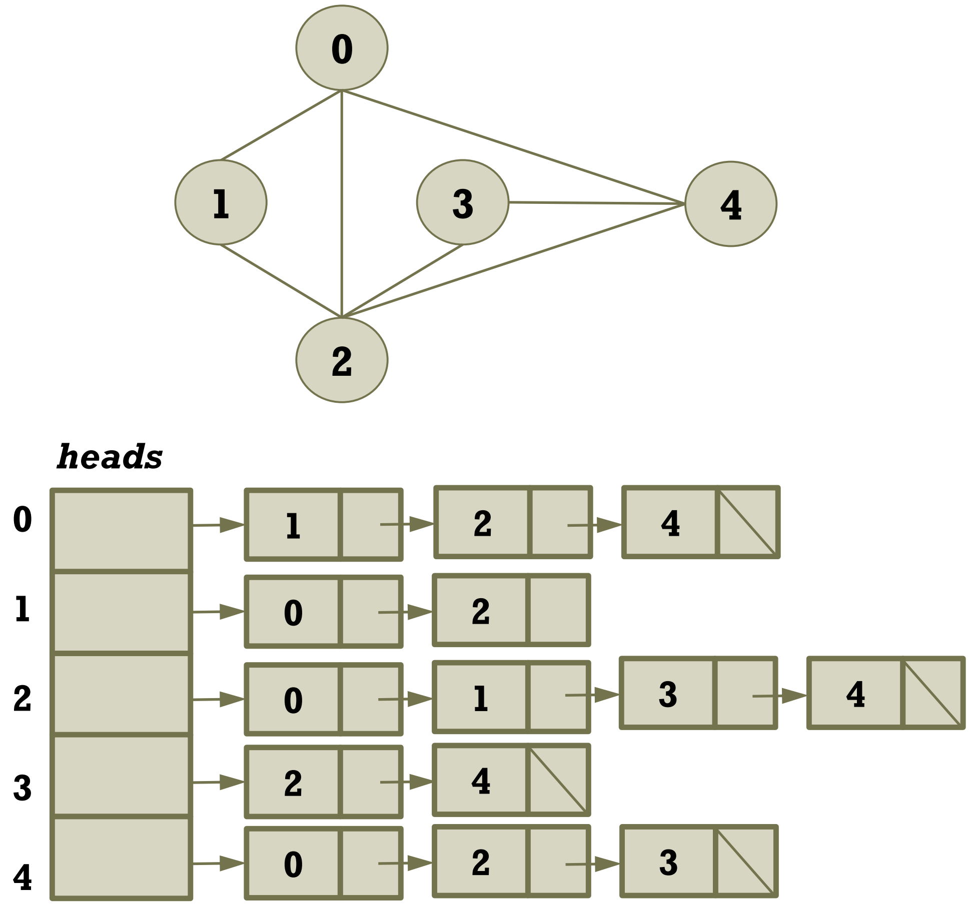
- Adding an undirected edge between src and desc

```
void AddEdge(Graph* pgraph, int src, int dest)
{
    GNode* newNode1, *newNode2, *cur;

    newNode1 = (GNode *)malloc(sizeof(GNode));
    newNode1->id = dest;
    newNode1->next = NULL;

    cur = pgraph->heads[src];  // Create a node for dest in src.
    while (cur->next != NULL)   // unsorted
        cur = cur->next;        // parallel edges
    cur->next = newNode1;

    newNode2 = (GNode *)malloc(sizeof(GNode));
    newNode2->id = src;
    newNode2->next = NULL;

    cur = pgraph->heads[dest]; // Create a node for src in dest.
    while (cur->next != NULL)
        cur = cur->next;
    cur->next = newNode2;
}
```

Building a Graph

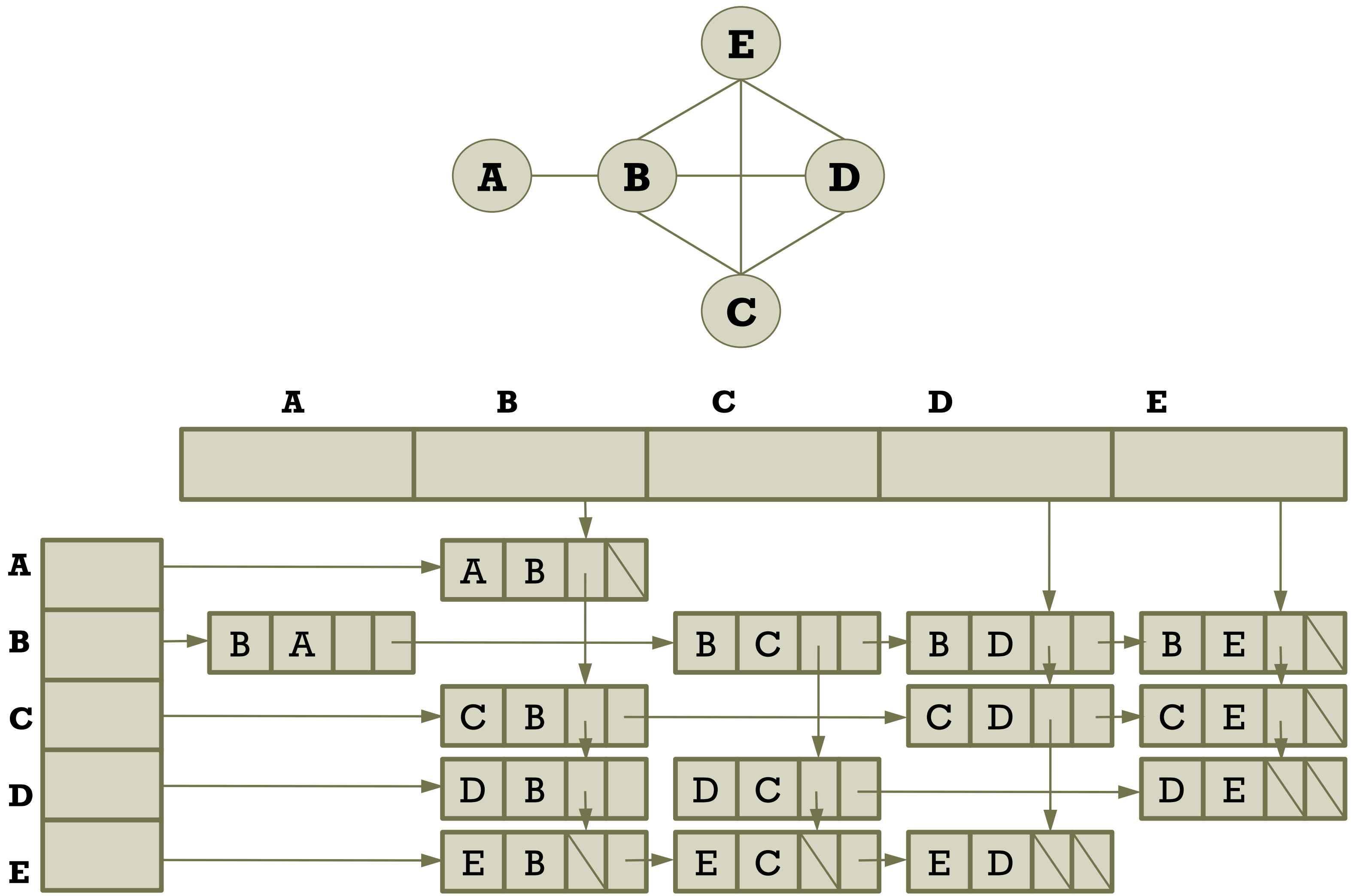
- Example

```
int main()
{
    Graph g;
    CreateGraph(&g, 5);
    AddEdge(&g, 0, 1);
    AddEdge(&g, 0, 2);
    AddEdge(&g, 0, 4);
    AddEdge(&g, 1, 2);
    AddEdge(&g, 2, 3);
    AddEdge(&g, 2, 4);
    AddEdge(&g, 3, 4);

    PrintGraph(&g);
    DestroyGraph(&g);

    return 0;
}
```



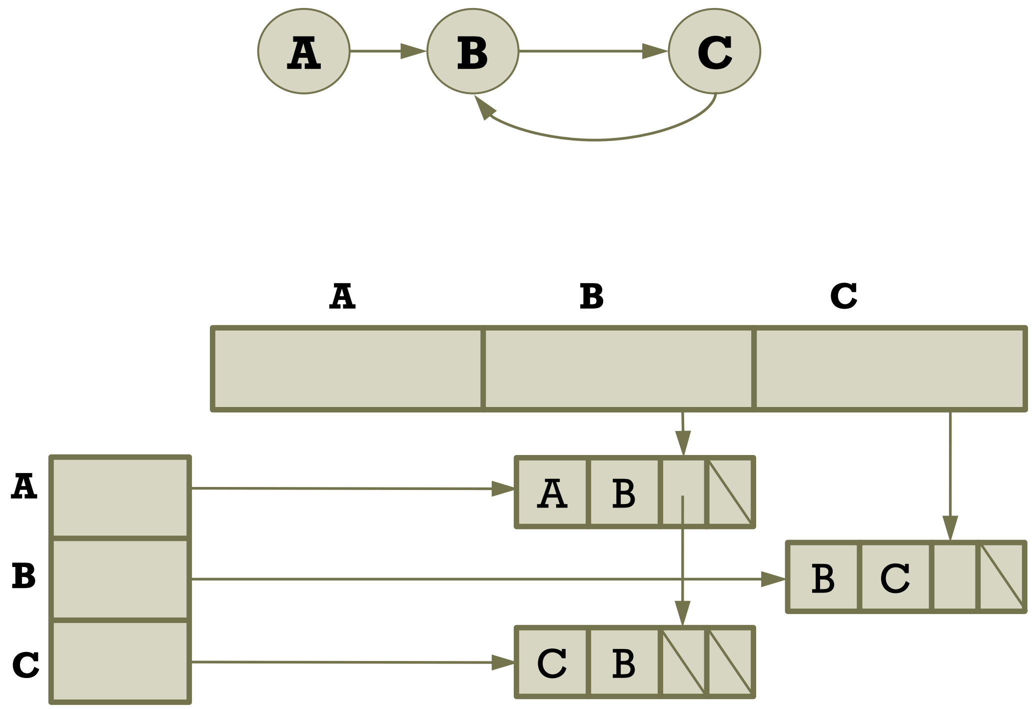
Another Adjacency List

- Use another heads for columns. (why?)



Another Adjacency List

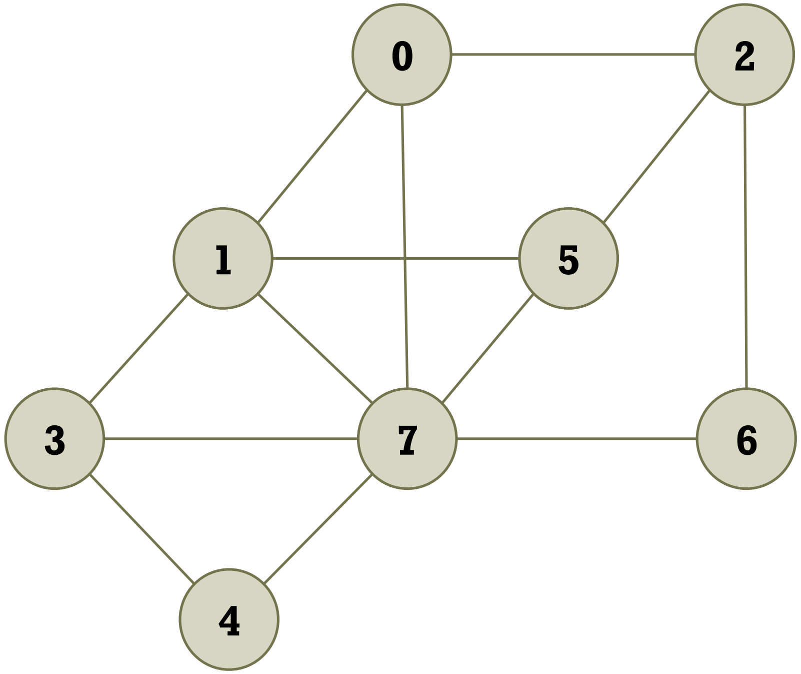
- Use another heads for columns.



Graph Traversal

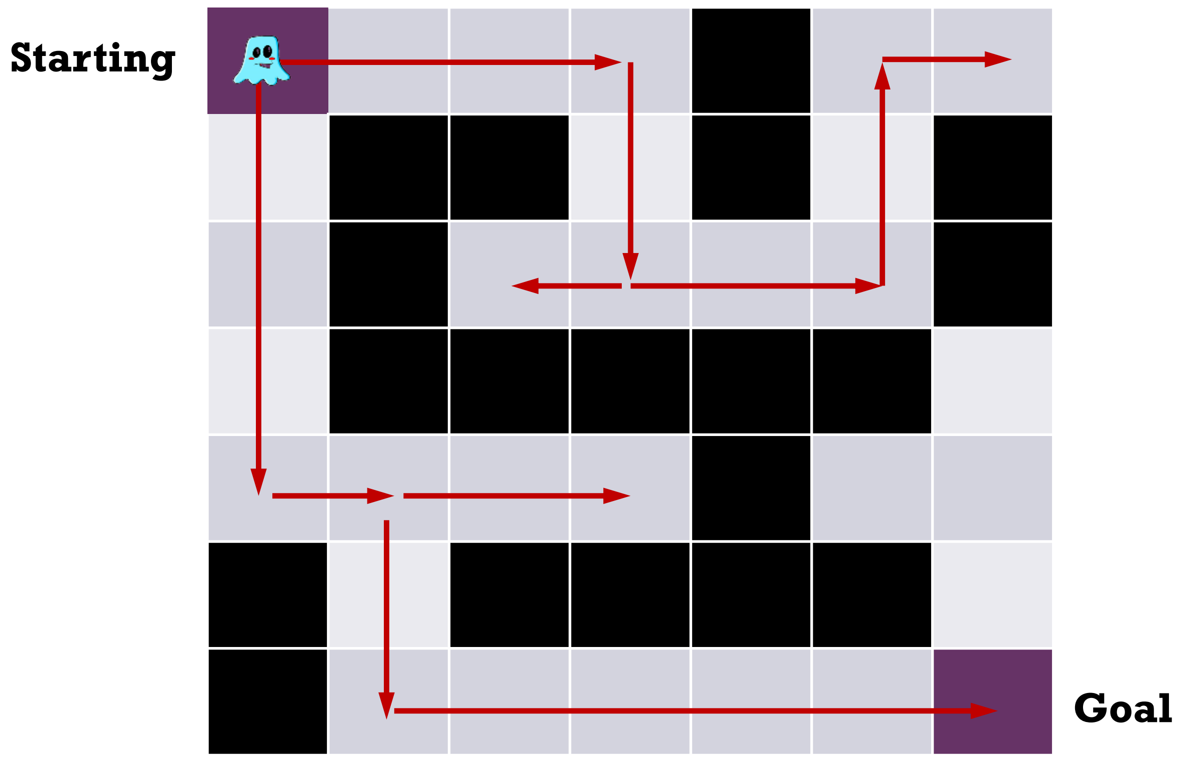
- Traversal
  - The process of visiting each node in a graph

- How to visit all nodes once?
  - Depth-first search (DFS)
  - Breadth-first search (BFS)



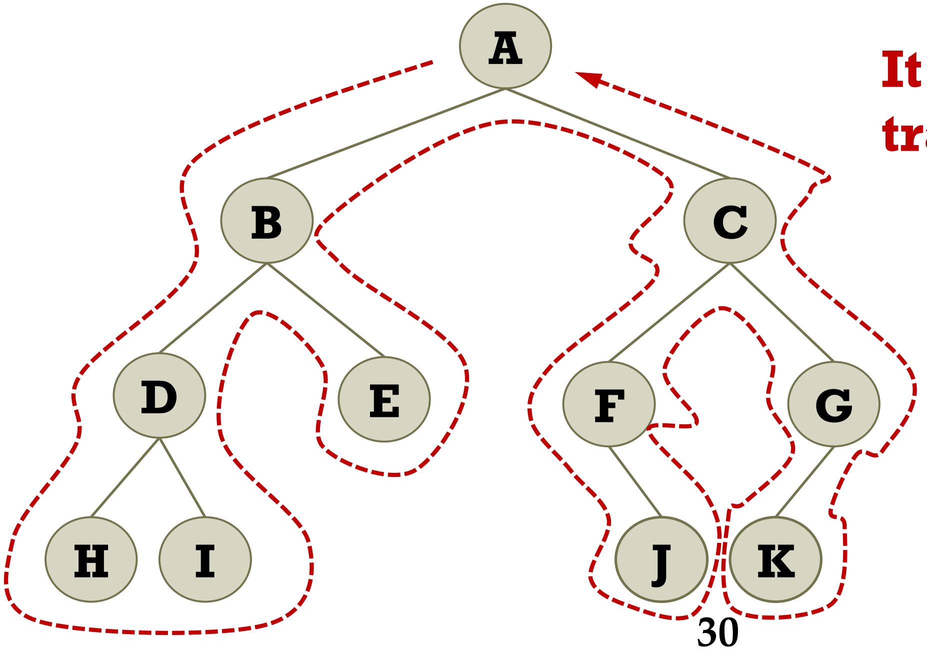
What is Depth-First Search (DFS)?

- Basic strategy of DFS
  - Keep moving until there is no more possible block.
  - Go back to the previous step and move other unvisited blocks.



Depth-First Search (DFS)

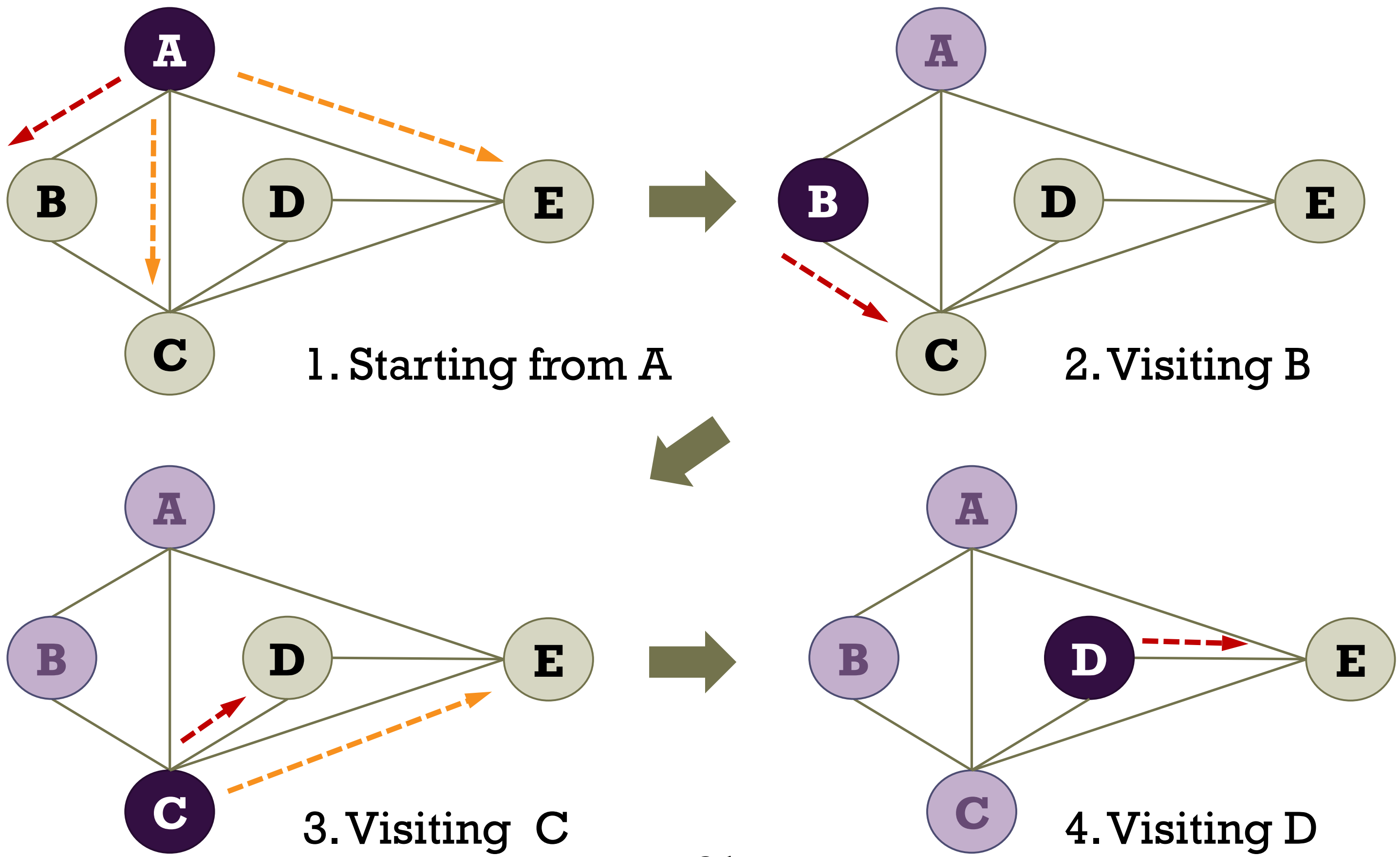
- Algorithm of DFS
  1. Visit the start vertex.
  2. Visit **one of unvisited** vertices neighboring to the start vertex.
  3. Repeat step 2 until there is no more unvisited vertex.
  4. If there is no more unvisited vertex, go back one step and visit another unvisited vertex.



It is similar to preorder traversal in a tree!

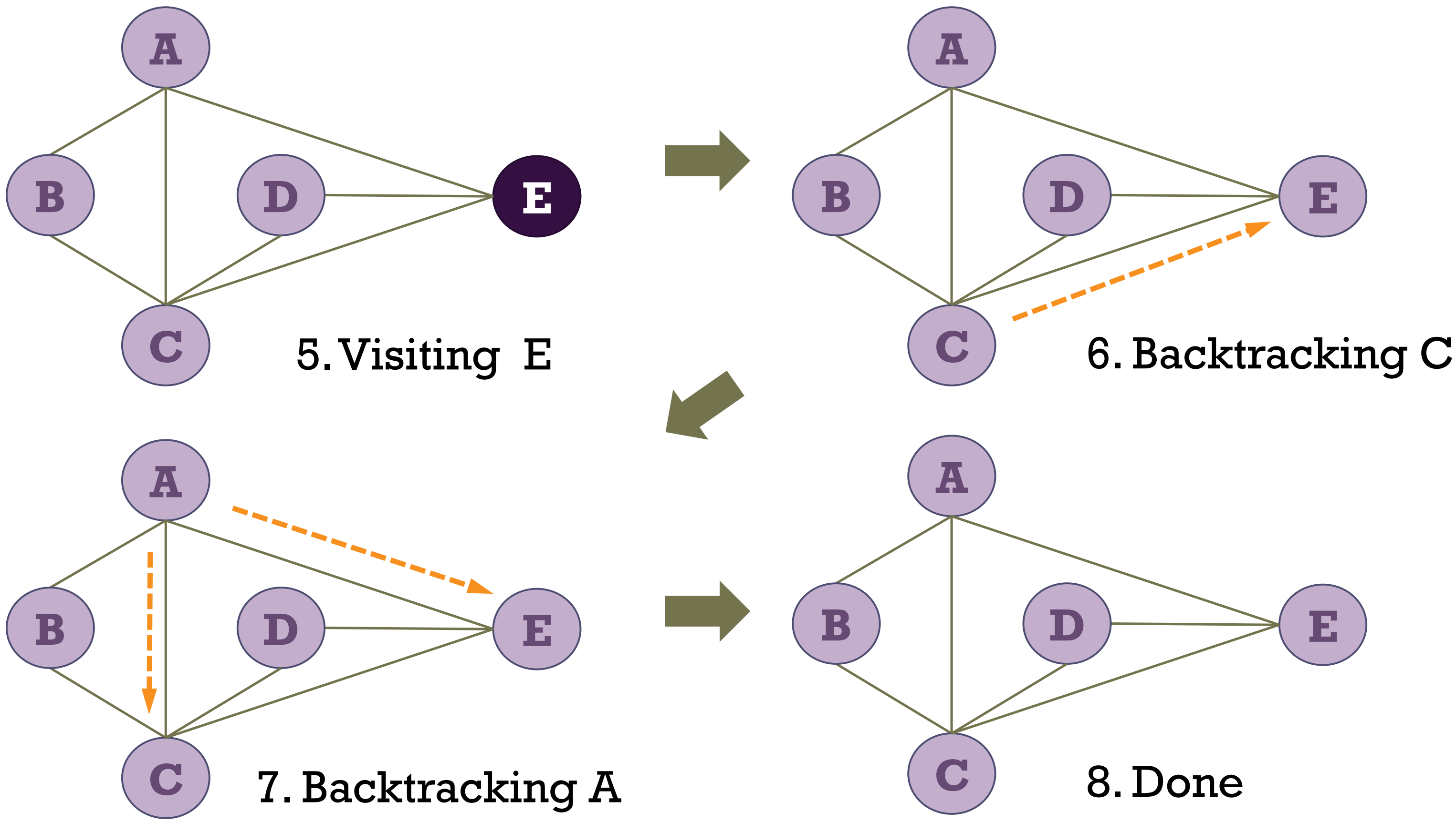
Depth-First Search (DFS)

- Example
  - Assume that the vertex is visited in alphabetical order if unvisited vertices are two or more.



Depth-First Search (DFS)

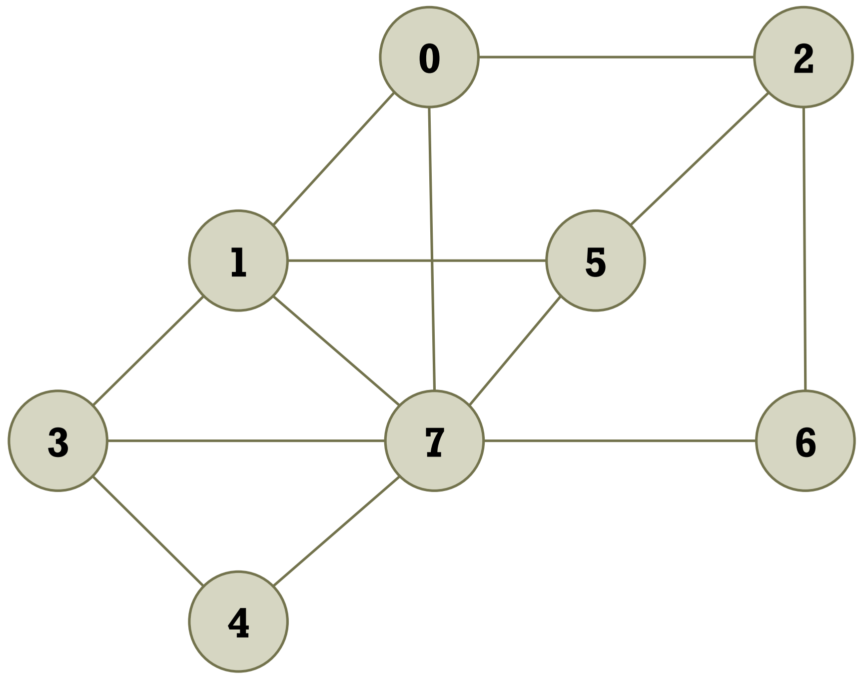
- Example
  - Assume that the vertex is visited in alphabetical order if unvisited vertices are two or more.



```c
void DFS(Graph* pgraph)
{
    Stack stack;
    bool* visited = (bool *)malloc(sizeof(bool)* pgraph->num);
    for (int i = 0; i < pgraph->num; i++)
        visited[i] = false;    // Make all vertices unvisited.

    InitStack(&stack);
    Push(&stack, 0);   // Push the initial vertex.
    while (!IsSEmpty(&stack)) {
        GNode* cur;
        int vtx = SPeek(&stack);
        Pop(&stack);

        // Skip if the vertex has been visited.
        if (visited[vtx]) continue;
        else {
            visited[vtx] = true;
            printf("%d ", vtx);
        }

        // Push the vertex if it has not been visited.
        cur = pgraph->heads[vtx]->next;
        while (cur != NULL)  {
            if (!visited[cur->id])
                Push(&stack, cur->id);
            cur = cur->next;
        }
    }
}
```

Depth-First Search (DFS)

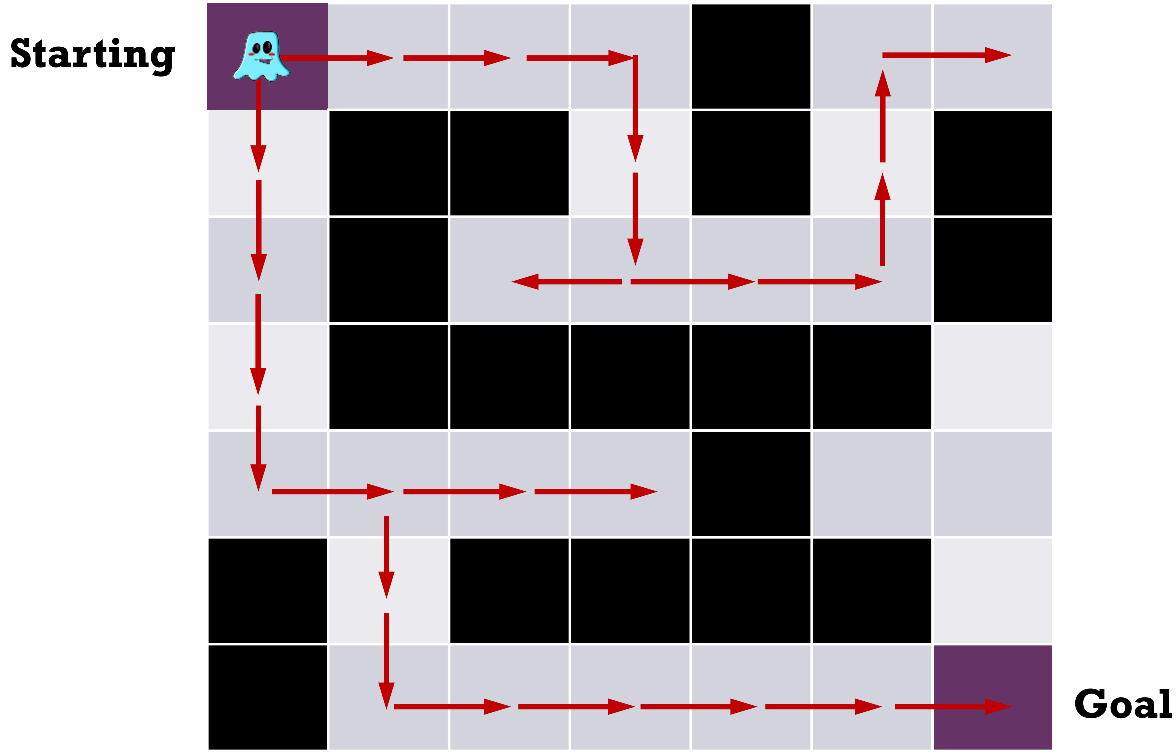
- How does DFS work?



0 1 3 4 7 5 2 6

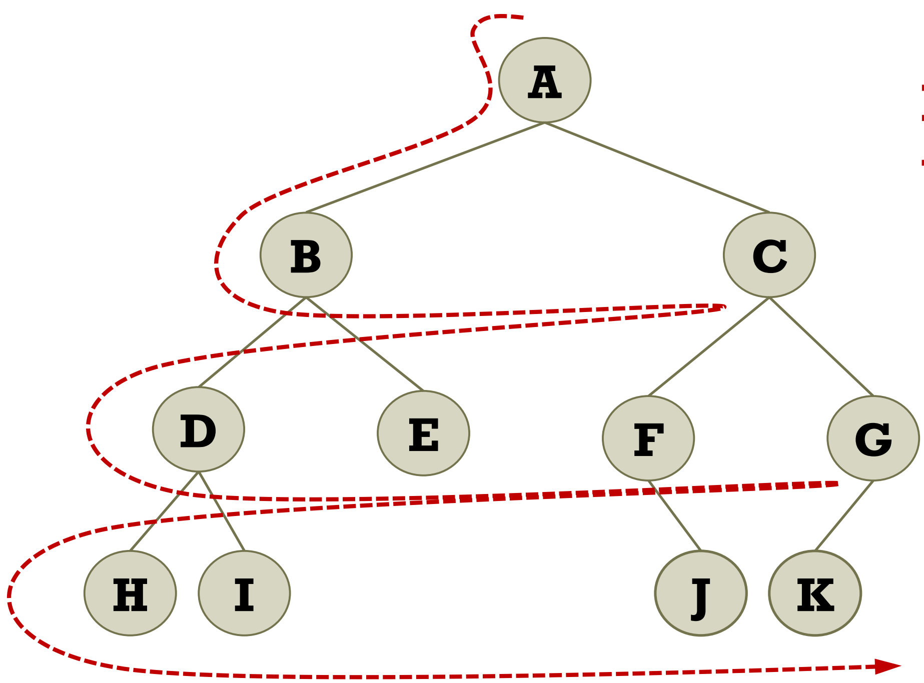
What is Breadth-First Search (BFS)?

- Basic strategy of BFS
  - Keep moving step-by-step for all possible blocks.



Breadth-First Search (BFS)

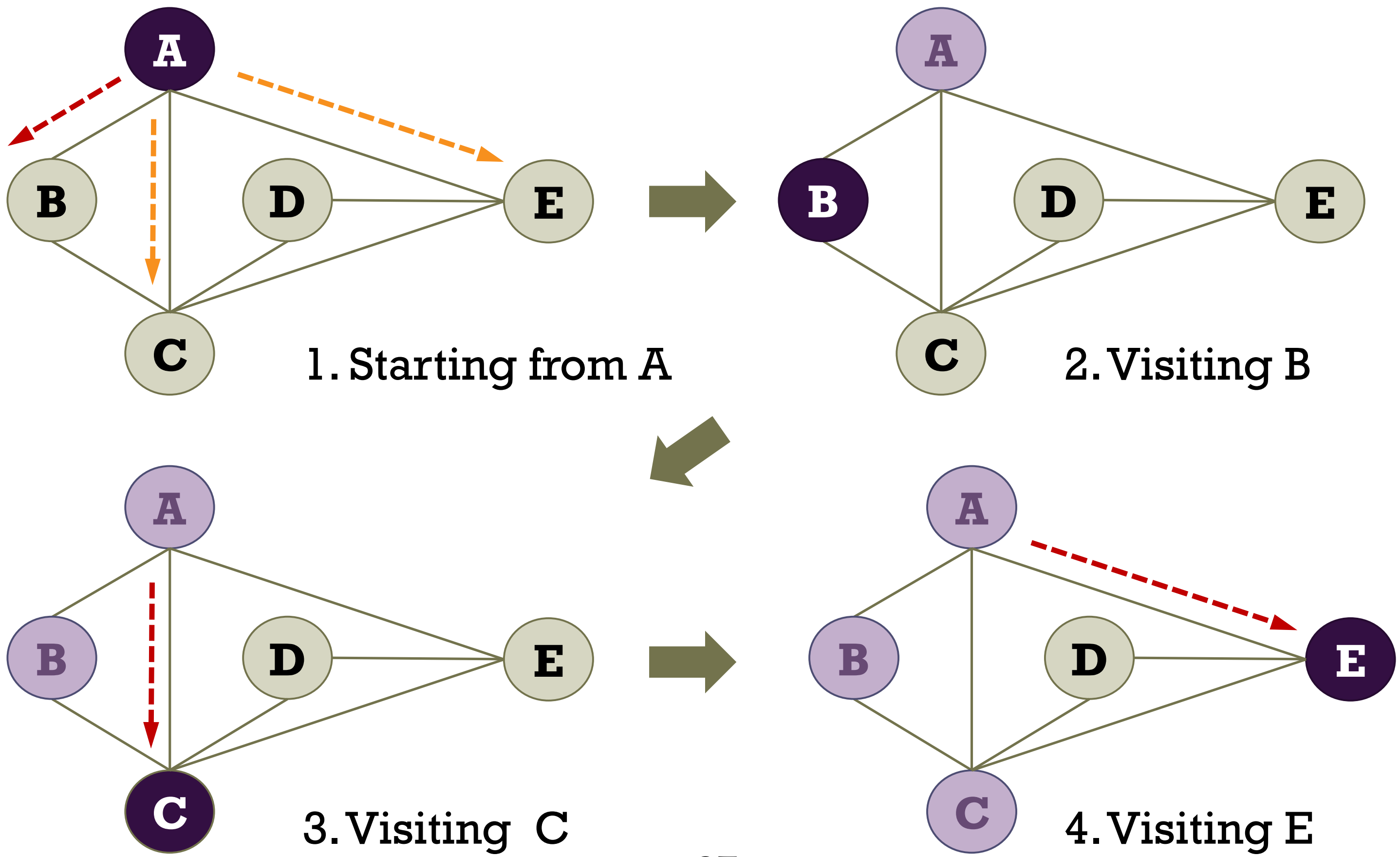
- Algorithm of BFS
  1. Visit the start vertex.
  2. Visit **all unvisited** vertices neighboring to the start vertex.
  3. Repeat step 2 until there is no more unvisited vertex.



It is similar to level order traversal in a tree!

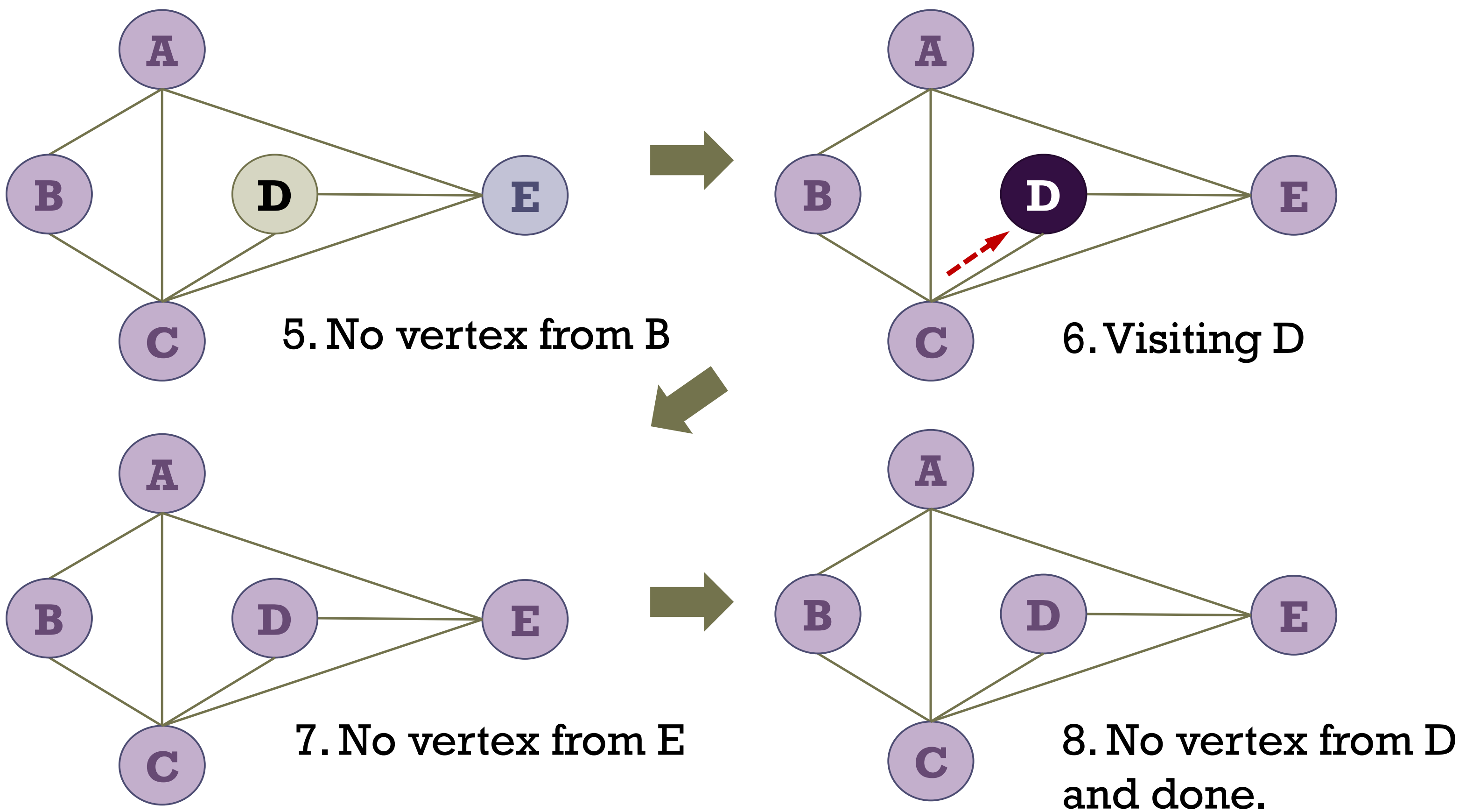
Breadth-First Search (BFS)

- Example
  - Assume that the vertex is visited in alphabetical order if unvisited vertices are two or more.



Breadth-First Search (BFS)

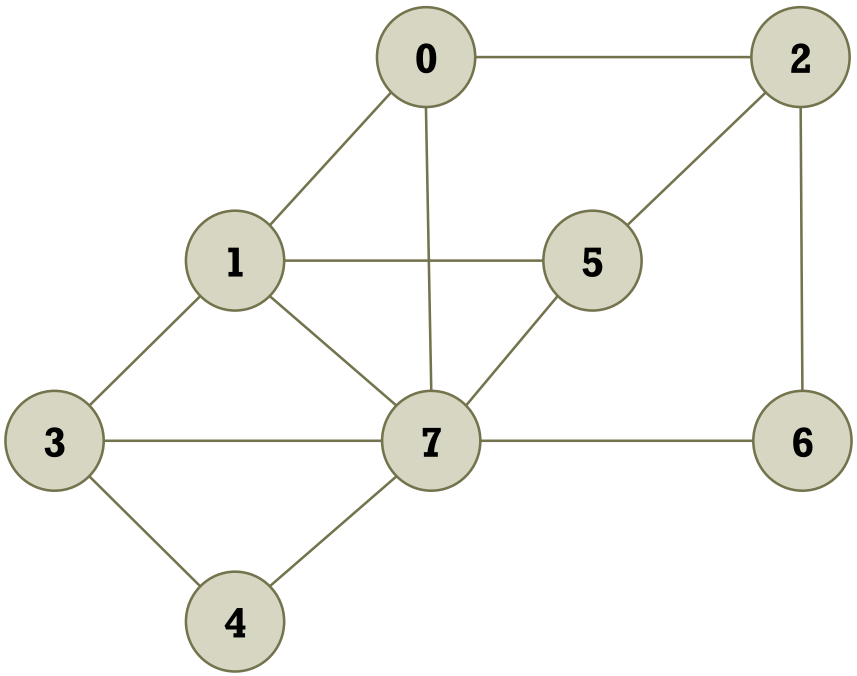
- Example
  - Assume that the vertex is visited in alphabetical order if unvisited vertices are two or more.



```c
void BFS(Graph* pgraph)
{
    Queue queue;
    bool* visited = (bool *)malloc(sizeof(bool)* pgraph->num);
    for (int i = 0; i < pgraph->num; i++)
        visited[i] = false;    // Make all vertices unvisited.

    InitQueue(&queue);
    EnQueue(&queue, 0);// Enqueue the initial vertex.
    while (!IsQEmpty(&queue)) {
        GNode* cur;
        int vtx = QPeek(&queue);
        DeQueue(&queue);

        // Skip if the vertex has been visited.
        if (visited[vtx]) continue;
        else {
            visited[vtx] = true;
            printf("%d ", vtx);
        }

        // Enqueue the vertex if it has been unvisited.
        cur = pgraph->heads[vtx]->next;
        while (cur != NULL)  {
            if (!visited[cur->id])
                EnQueue(&queue, cur->id);
            cur = cur->next;
        }
    }
}
```

Breadth-First Search (BFS)

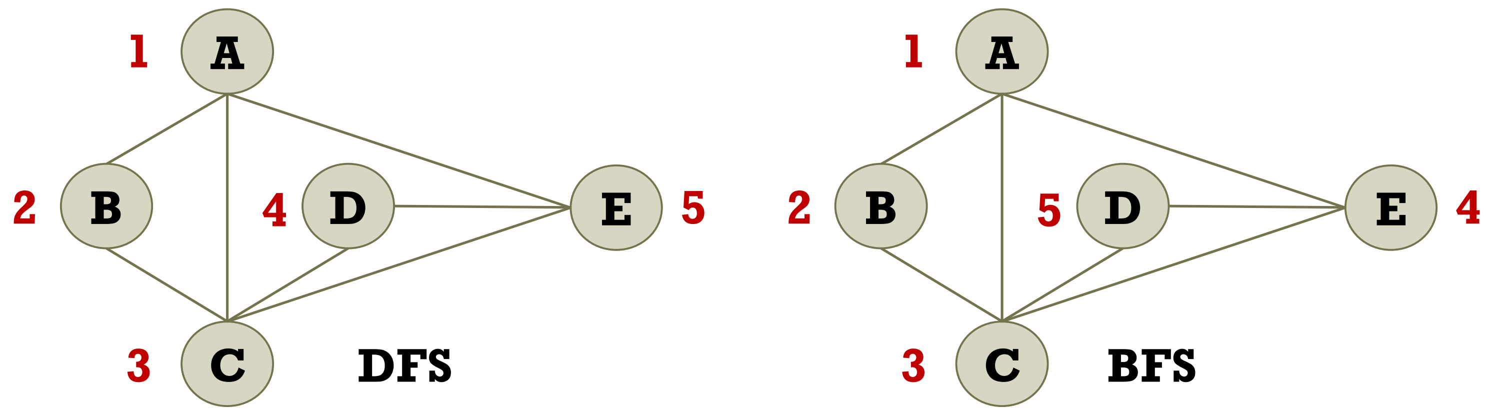
- How does BFS work? (sorted linked list)



0 1 2 7 3 5 6 4

Summary of DFS and BFS

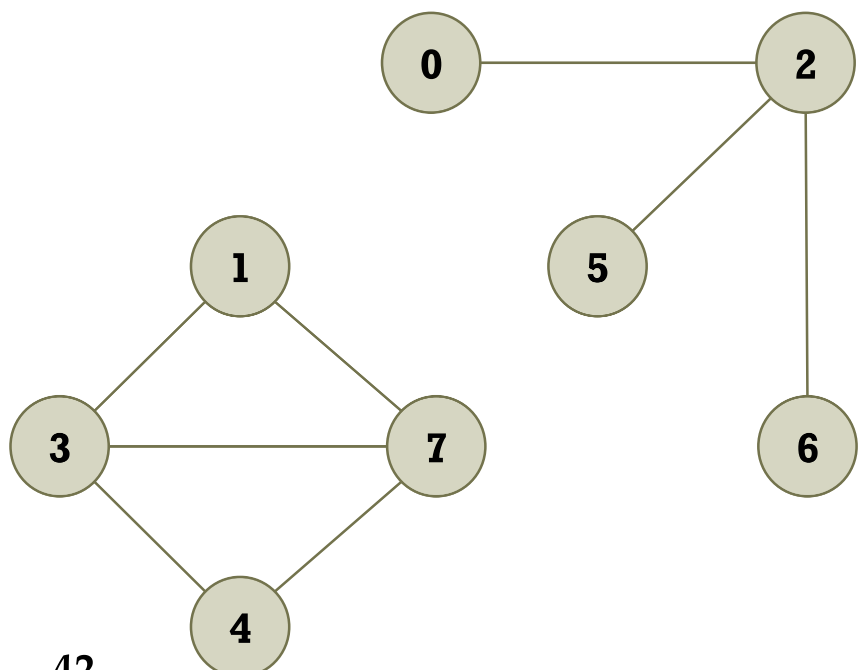
- Implementation
  - DFS is implemented with the **stack**.
  - BFS is implemented with the **queue**.



- Time complexity
  - Adjacency matrix: $O(|V|^2)$
  - Adjacency list: $O(|V| + |E|)$

Connected Component

- How to count the number of vertices in the connected component?
  - Do depth first search or breadth first search.
  - Check all vertex has been visited.
  - If not, do depth or breadth first search from one of the unvisited vertices until all vertices are visited.



- Time complexity
  - Adjacency matrix: $O(|V|^2)$
  - Adjacency list: $O(|V| + |E|)$