[SWE2015-41] Introduction to Data Structures (자료구조개론)

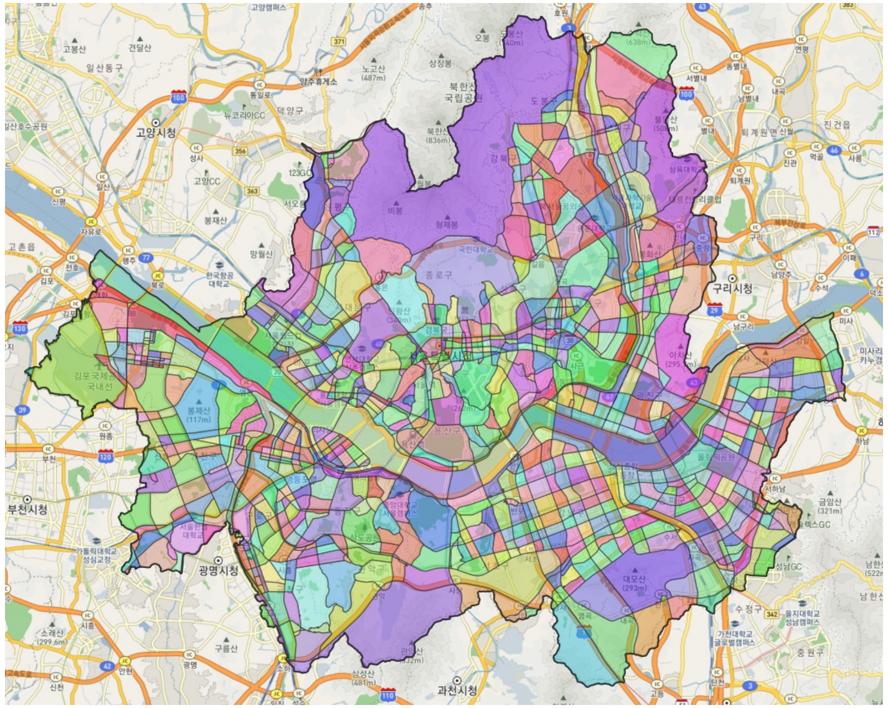
Graphs

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps



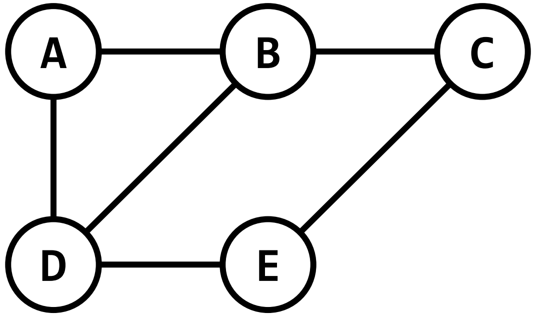
Social Networks
- Each vertex represents a person
- Each edge represents a relationship between two people

Maps
- Each vertex represents a building
- Each edge represents a street between two buildings

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`



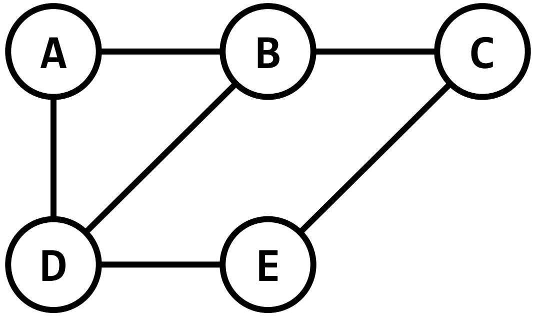

```
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C),
      (B,D), (C,E), (D,E), }
```

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Undirected graph**: vertices can be traversed from `u` to `v` as well as from `v` to `u`
    - The order between `u` and `v` is not important



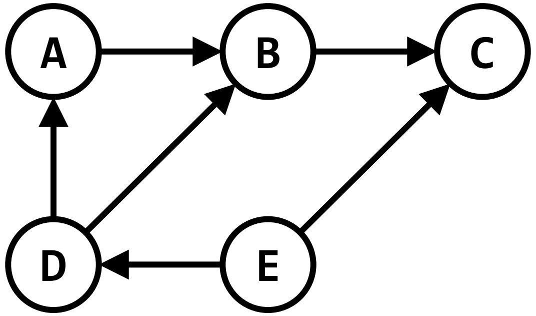

```
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C),
      (B,D), (C,E), (D,E), }
```

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Directed graph**: vertices can be traversed from `u` to `v`, not from `v` to `u`
    - The order between `u` and `v` is important



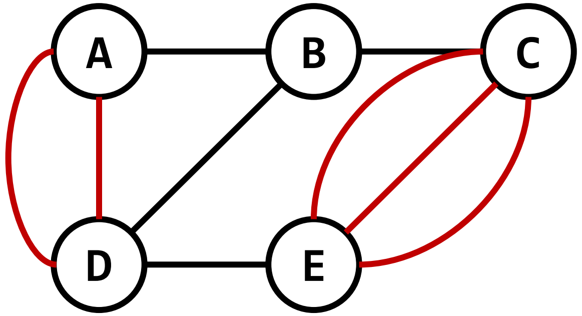

```
V = { A, B, C, D, E }

E = { (A,B), (B,C), (D,A),
      (D,B), (E,C), (E,D), }
```

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph G is defined by V and E, i.e., $G=(V,E)$ where
  - V is a set of vertices in G
  - E is a set of edges in G
  - An edge $(u,v) \in E$ is a pair of two vertices $u,v \in V$
  - **Parallel edges**: multiple edges between the same pair u and v
    - In this case, the set E is not a set anymore



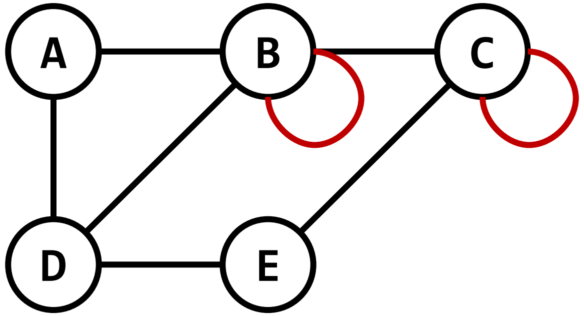
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C), (A,D), (C,E), (B,D), (C,E), (D,E), (C,E), }

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Self-loop edges**: an edge from a vertex `u` to itself `u`



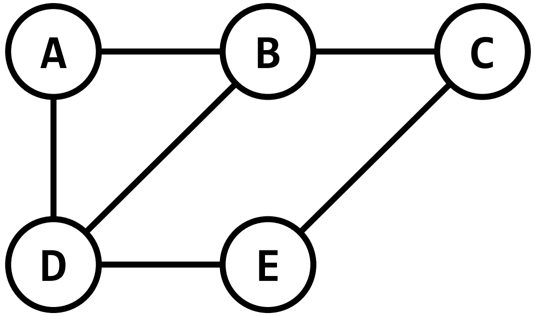

```
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C), (B,B),
      (B,D), (C,E), (D,E), (C,C), }
```

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Simple graphs** have no parallel edge and no self-loop edge
    - In this lecture, assume graphs are simple unless otherwise stated



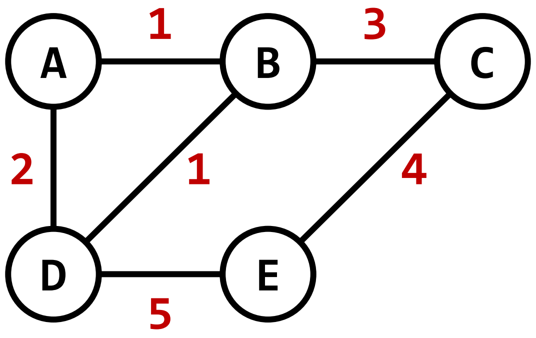

```
V = { A, B, C, D, E }

E = { (A,B), (A,D), (B,C),
      (B,D), (C,E), (D,E), }
```

What is Graph?

- A graph is a collection of **vertices** and **edges** that connect these vertices
  - Example: social networks, maps

- **Definition:** A graph `G` is defined by `V` and `E`, i.e., `G=(V,E)` where
  - `V` is a set of vertices in `G`
  - `E` is a set of edges in `G`
  - An edge `(u,v)` ∈ `E` is a pair of two vertices `u,v` ∈ `V`
  - **Weighted graphs**: each edge is associated with a **weight** value
    - The edge representation `(u,v)` is extended to `(u,v,w)` where `w` is the weight of `(u,v)`




```
V = { A, B, C, D, E }

E = { (A,B,1), (A,D,2), (B,C,3),
      (B,D,1), (C,E,4), (D,E,5), }
```

Graph Terminology

- **Adjacent** nodes or **Neighbors**
  - For every edge `(u,v)` ∈ E,
  - u is said to be **adjacent** to v (and vice versa)
  - u is said to be a **neighbor** of v (and vice versa)

- The **degree** of a vertex u (in an undirected graph)
  - the number of edges containing u (i.e., `(*,u)` & `(u,*)` edges)

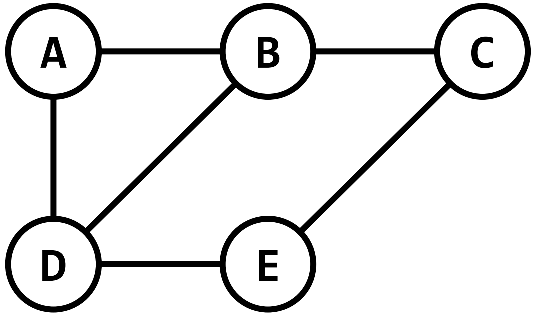
- The **in/out-degree** of a vertex u (in a directed graph)
  - in-degree is the number of edges coming to u (i.e., `(*,u)` edges)
  - out-degree is the number of edges starting from u (i.e., `(u,*)` edges)

Graph Terminology

In both undirected and directed graphs,

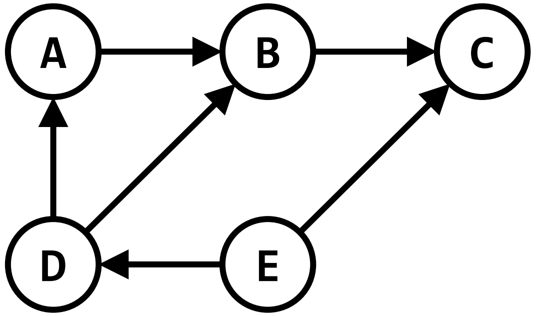
- A is **adjacent** to B and D
- B and D are **neighbors** of A

Undirected Graph



- `degree(A) = 2`
- `degree(D) = 3`

Directed Graph



- `in/out-degree(A) = 1 / 1`
- `in/out-degree(D) = 1 / 2`

Graph Terminology

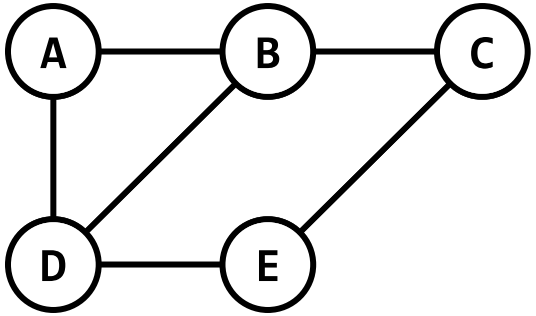
- A **path** P in a graph $G=(V,E)$
  - A sequence of vertices, $P = (v_0, v_1, v_2, ..., v_n)$ where for all i,
    - $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ when G is undirected
    - $(v_i, v_{i+1}) \in E$ when G is directed

  For the above path P,
  - Its **length** is equal to n, the number of edges on P
  - It is said to be **closed** (or a **cycle**) when $v_0 = v_n$
  - It is said to be **simple** when all the vertices in the path are distinct with an exception that v_0 may be equal to v_n

Graph Terminology

- A **path** P in a graph $G=(V,E)$
  - A sequence of vertices, $P = (v_0, v_1, v_2, ..., v_n)$ where for all i,
    - $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ when G is undirected
    - $(v_i, v_{i+1}) \in E$ when G is directed

  For the above path P,
  - Its **length** is equal to n, the number of edges on P
  - It is said to be **closed** (or a **cycle**) when $v_0 = v_n$
  - It is said to be **simple** when all the vertices in the path are distinct with an exception that v_0 may be equal to v_n

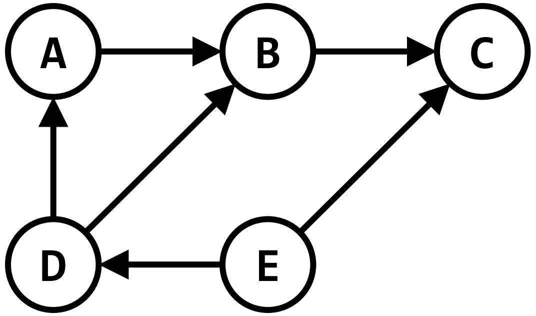


- (A,B,D,A) is a simple cycle
- (A,B,C,E,D,B) is not simple
- There are three simple cycles

Graph Terminology

- A **path** P in a graph $G=(V,E)$
  - A sequence of vertices, $P = (v_0, v_1, v_2, ..., v_n)$ where for all i,
    - $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ when G is undirected
    - $(v_i, v_{i+1}) \in E$ when G is directed

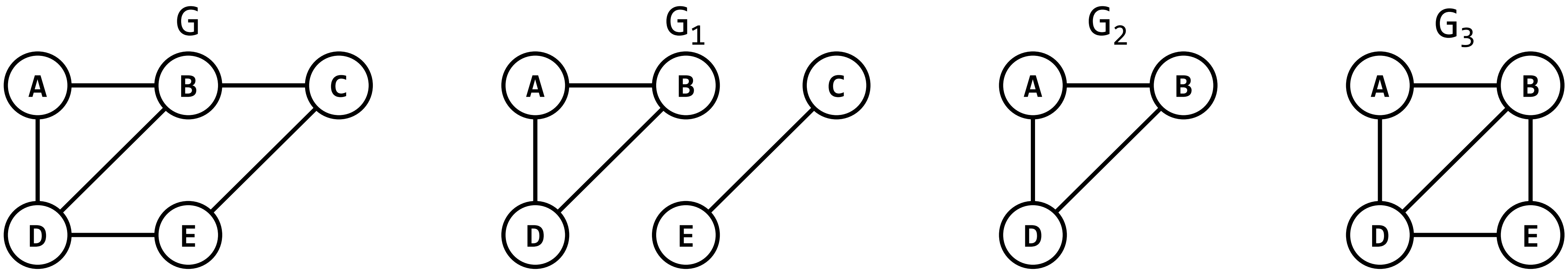
  For the above path P,
  - Its **length** is equal to n, the number of edges on P
  - It is said to be **closed** (or a **cycle**) when $v_0 = v_n$
  - It is said to be **simple** when all the vertices in the path are distinct with an exception that v_0 may be equal to v_n



- There is no cycle
- The length of (E,D,B,C) is 3
- (A,D,B,C) is not a path

Graph Terminology

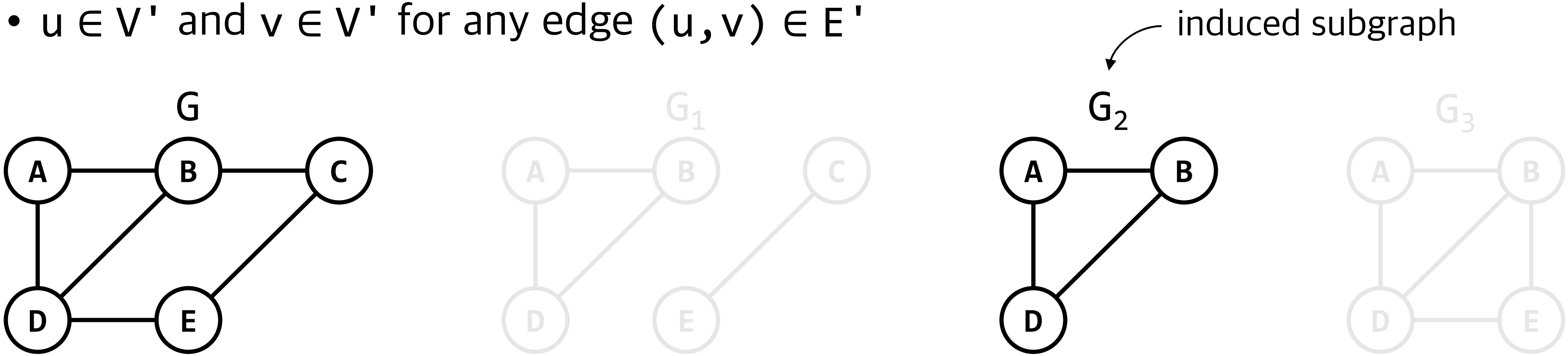
- A **subgraph** `G'=(V',E')` of a graph `G=(V,E)` is a graph satisfying …
  - `V'` ⊆ V and `E'` ⊆ E is
  - u ∈ `V'` and v ∈ `V'` for any edge `(u,v)` ∈ `E'`



- G_1 and G_2 subgraphs of G
- G_3 is not a subgraph of G
- G_2 is a subgraph of G_1

Graph Terminology

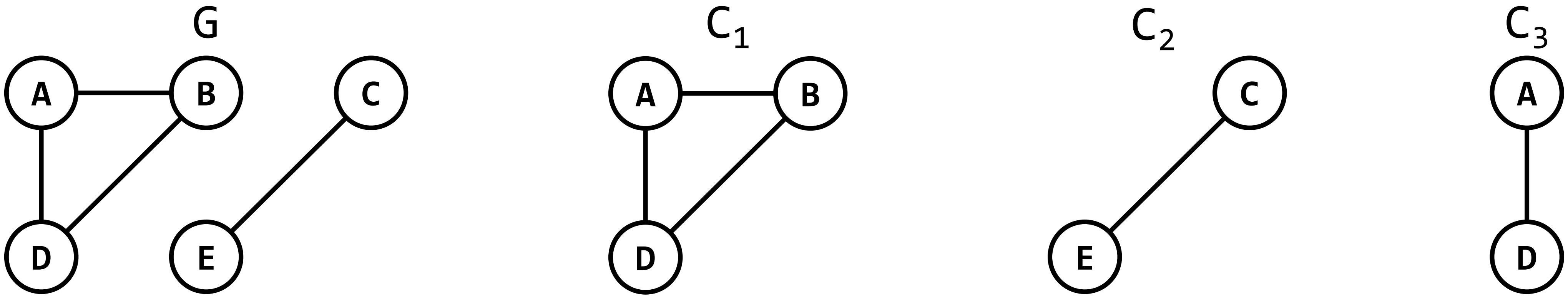
- A **subgraph** $G'=(V',E')$ of a graph $G=(V,E)$ is a graph satisfying …
  - $V' \subseteq V$ and $E' \subseteq E$ is
  - $u \in V'$ and $v \in V'$ for any edge $(u,v) \in E'$



- A subgraph **induced** by $S \subseteq V$ is the subgraph whose vertex set is S and whose edge set includes edges as many as possible
  - Such a subgraph is called an **induced subgraph** and denoted by $G[S]$
  - Formally, $G[S]=(S,E')$ where $E' = \{(u,v) \in E : u \in S \text{ and } v \in S\}$

Graph Terminology (in Undirected Graphs)

- A **connected component** is a **maximal connected** subgraph
  - A graph is said to be **connected** if there is a path between any pair of vertices
  - A connected subgraph is said to be **maximal**
    if it is not a subgraph of any another connected subgraph

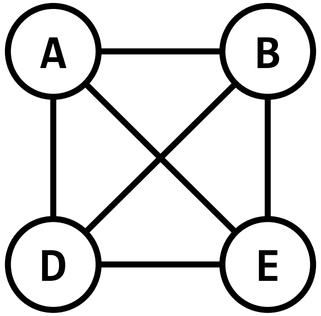


- G has two connected components C_1 and C_2
- C_3 is connected, but not maximal because it is a subgraph of C_1

Graph Terminology

- A graph is **completed** if there is an edge between any pair of vertices
  - An undirected complete graph has `N(N-1)/2` edges
  - A directed complete graph has `N(N-1)` edges

Example of an undirected complete graph of `N` vertices



- `E = { (A,B), (A,C), (A,D),`
  `(B,C), (B,D), (C,D), }` when `V = {A,B,C,D}`
- $3_{(A,*)} + 2_{(B,*)} + 1_{(C,*)} + 0_{(D,*)} = 6$
- For `N` vertices, the number is `(N-1)+...+2+1 = N(N-1)/2`

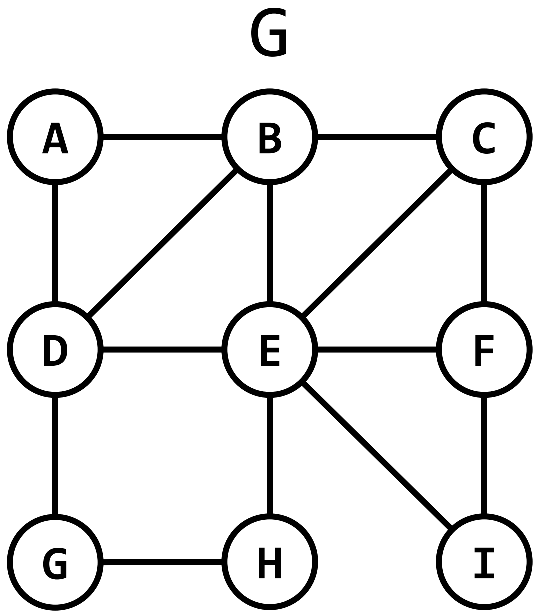
- A graph is said to be …
  - **dense** when $|E| \approx |V|^2$
  - **sparse** when $|E| \ll |V|^2$, e.g., $|E| \approx |V|$

Questions

For the graph `G`,

- **(Q1)** What are the neighbors of `E`?
- **(Q2)** What is the degree of `D`?
- **(Q3)** Draw the subgraph induced by `S={B,C,E,F,I}`

- **(Q4)** Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- **(Q5)** Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- **(Q6)** Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- **(Q7)** Find the shortest path from `C` to `G`
  - This can be found by shortest path algorithms, which will be covered next week

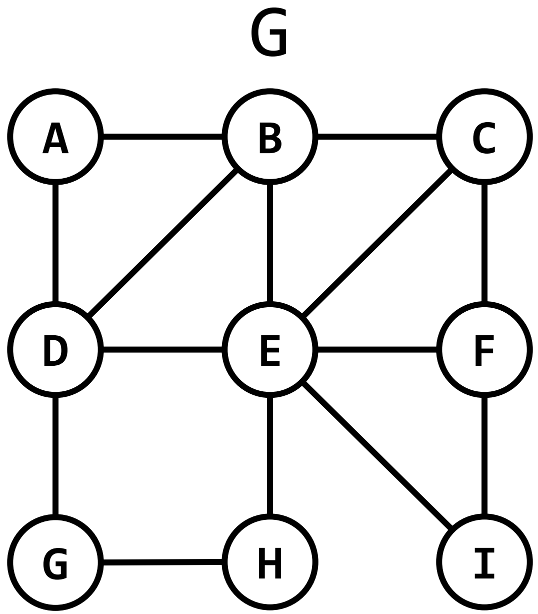


Questions

For the graph G,

- **(Q1)** What are the neighbors of E? B C D F H I
- **(Q2)** What is the degree of D? 4
- **(Q3)** Draw the subgraph induced by S={B,C,E,F,I}

- **(Q4)** Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- **(Q5)** Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- **(Q6)** Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- **(Q7)** Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week

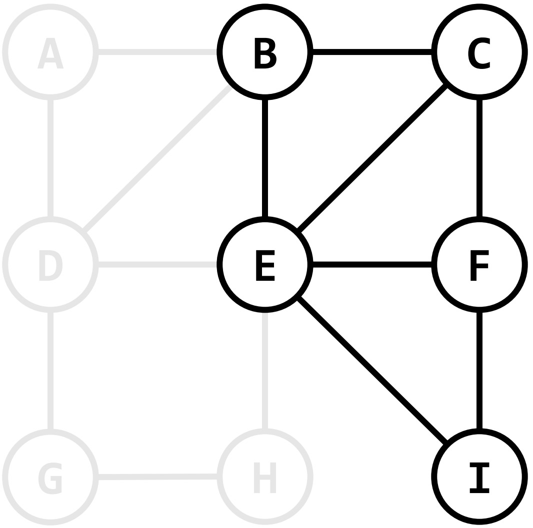


Questions

For the graph G,

- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week

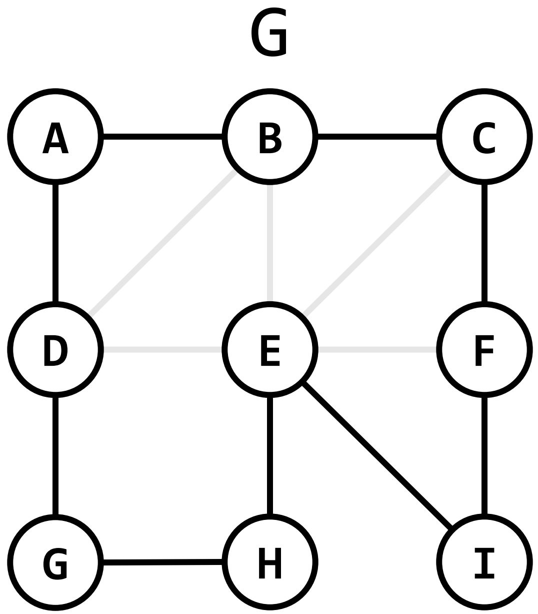
G



Questions

For the graph G,

- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}

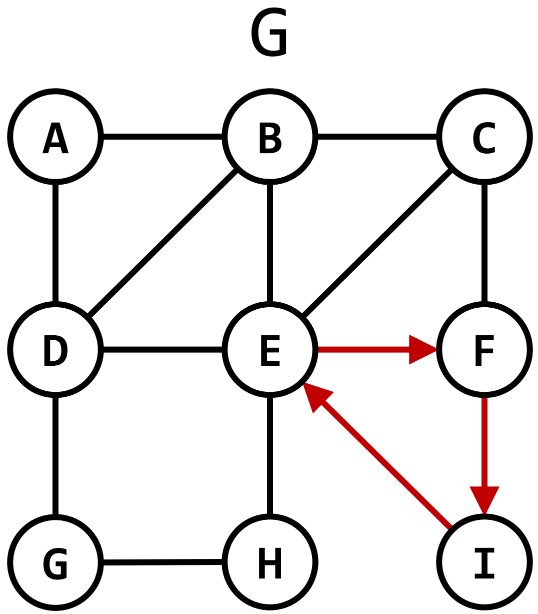


- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week

Questions

For the graph G,

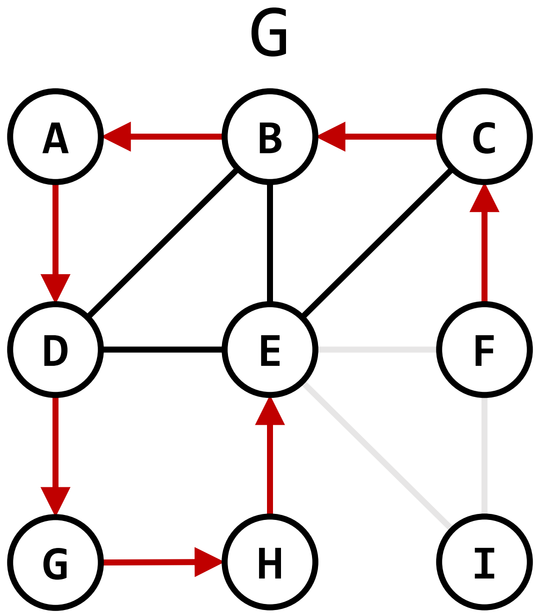
- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week



Questions

For the graph G,

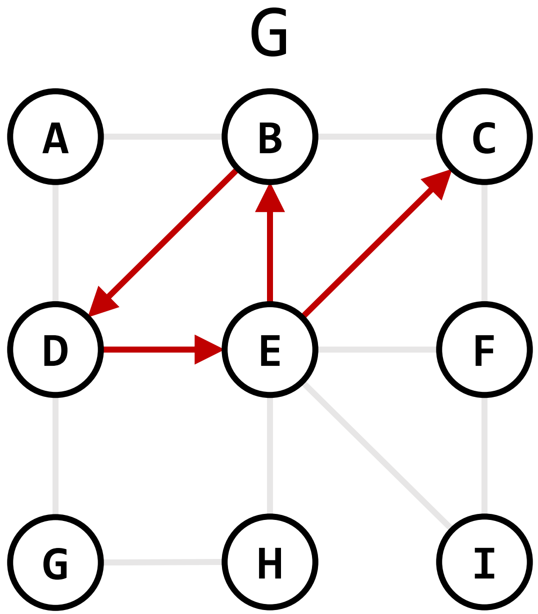
- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week



Questions

For the graph G,

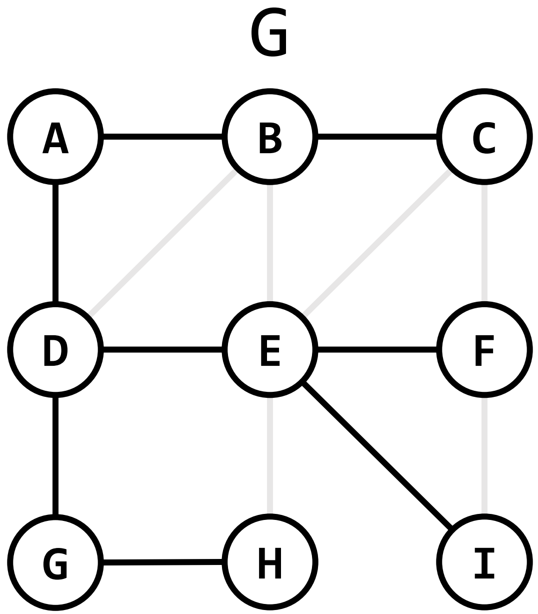
- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week



Questions

For the graph G,

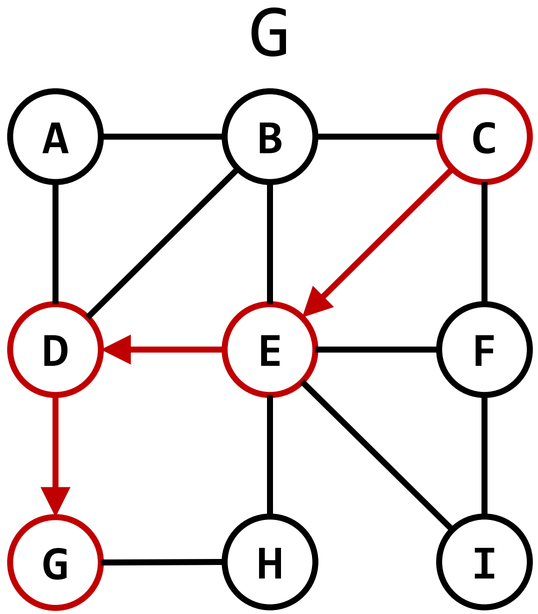
- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- **(Q6)** Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- **(Q7)** Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week



Questions

For the graph G,

- (Q1) What are the neighbors of E?
- (Q2) What is the degree of D?
- (Q3) Draw the subgraph induced by S={B,C,E,F,I}
- (Q4) Find a cycle that visits each vertex exactly once
  - This is known as **Hamiltonian cycle**
- (Q5) Find a path that visits every edge exactly once
  - This is known as **Eulerian path**
- (Q6) Find a connected subgraph that contains (a) all the vertices and (b) no cycle
  - This is known as a **spanning tree**, which will be covered next week
- (Q7) Find the shortest path from C to G
  - This can be found by shortest path algorithms, which will be covered next week



Questions

Let `G` be an undirected complete graph with `N` vertices

(Q1) What is # of subgraphs of `M` vertices?

(Q2) What is # of induced subgraphs?

(Q3) What is # of cycles of length `L`?

Questions

Let G be an undirected complete graph with N vertices

(Q1) What is # of subgraphs of M vertices?

- # of subsets of the vertex set V = $_{N}C_{M}$ = n!/m!(n-m)!
- There are two cases for each possible edge: $e \in E$ or $e \notin E$
- # of subgraphs = $_{N}C_{M} \times 2^{M(M-1)/2}$

(Q2) What is # of induced subgraphs?

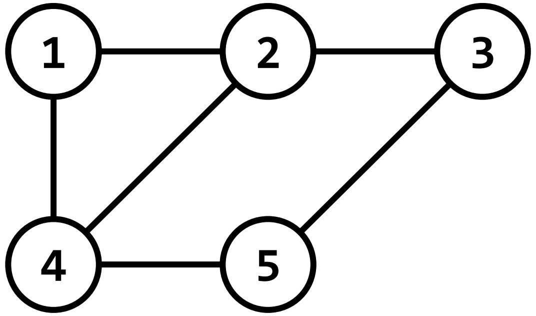
- There are two cases for each vertex: $v \in V$ or $v \notin V$
- # of induced subgraphs = 2^{N}

(Q3) What is # of cycles of length L?

- # of subsets of the vertex set V = $_{N}C_{L}$
- # of cycles in each subset = (L-1)!
- # of cycles = $_{N}C_{L} \times (L-1)!$

Implementation - Adjacent Matrix

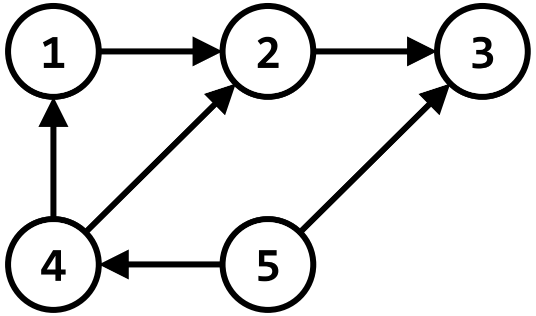
- **Adjacent Matrix** represents edges in a `N` × `N` matrix `A[u,v]`
  - Let `V` = `{ 1, ..., N }` be the set of vertices
  - `A[u,v]` = `1` if there is an edge from `u` to `v`, otherwise `A[u,v]` = `0`



G=(V,E)

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

A



G=(V,E)

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

A

Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a $N \times N$ matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

Interesting property

- `A[u,v]` can be considered as the number of paths from `u` to `v` of length 1
- What is the meaning of $A^2 = A \times A$ (matrix multiplication) ?

Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a `N` × `N` matrix `A[u,v]`
  - Let `V` = `{ 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

Interesting property

- `A[u,v]` can be considered as the number of paths from `u` to `v` of length 1

- What is the meaning of `A²` = `A` × `A` (matrix multiplication) ?

$$A^2[u,v] = \sum_{w \in V} A[u,w] \times A[w,v]$$

Implementation - Adjacent Matrix

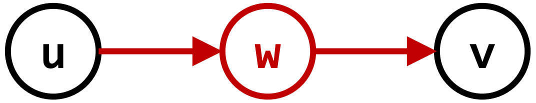
- **Adjacent Matrix** represents edges in a `N × N` matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

  Interesting property
  - `A[u,v]` can be considered as the number of paths from `u` to `v` of length 1

  - What is the meaning of `A² = A × A` (matrix multiplication) ?

$$A^2[u,v] = \sum_{w \in V} \underline{A[u,w] \times A[w,v]}$$

# of paths through `w`



Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a $N \times N$ matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from u to v, otherwise `A[u,v] = 0`

  Interesting property
  - `A[u,v]` can be considered as the number of paths from u to v of length 1

  - What is the meaning of $A^2 = A \times A$ (matrix multiplication) ?
  - A^2`[u,v]` is the number of paths from u to v of length 2

  - Similarly, A^k`[u,v]` is the number of paths from u to v of length k

Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a `N × N` matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

(Case 1) When you know the maximum number of vertices

```
#define MAX_SIZE 1000
int AdjacentMatrix[MAX_SIZE][MAX_SIZE];
```

Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a $N \times N$ matrix `A[u,v]`
  - Let V = `{ 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from u to v, otherwise `A[u,v] = 0`

`(Case 2)` If you want to allocate memory dynamically and use **1D array**,

```
int* createEmptyMatrix(int N) {
    int *matrix = (int*)malloc(sizeof(int)*N*N);
    for (int i = 0; i < N; i ++) {
        for (int j = 0; j < N; j ++) {
            matrix[i*N+j] = 0;
        }
    }
}
```

Implementation - Adjacent Matrix

- **Adjacent Matrix** represents edges in a `N × N` matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

`(Case 3)` If you want to allocate memory dynamically and use **2D array**,

```
int* createEmptyMatrix(int N) {
    int **matrix = (int**)malloc(sizeof(int*) * N);
    for (int i = 0; i < N; i ++) matrix[i] = (int*)malloc(sizeof(int) * N);
    for (int i = 0; i < N; i ++) {
        for (int j = 0; j < N; j ++) {
            matrix[i][j] = 0;
        }
   }
}
```

Implementation - Adjacent Matrix

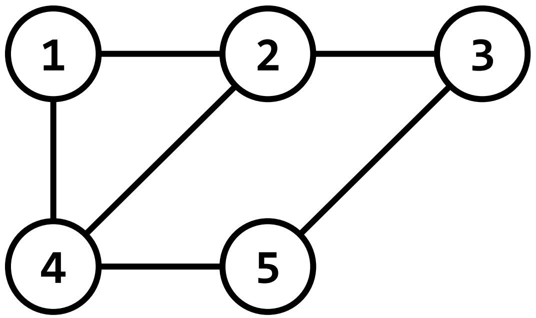
- **Adjacent Matrix** represents edges in a `N × N` matrix `A[u,v]`
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u,v] = 1` if there is an edge from `u` to `v`, otherwise `A[u,v] = 0`

Pros
- It is easy to implement
- It is easy to check whether an edge between `u` and `v` exists
- It is efficient to add or delete an edge

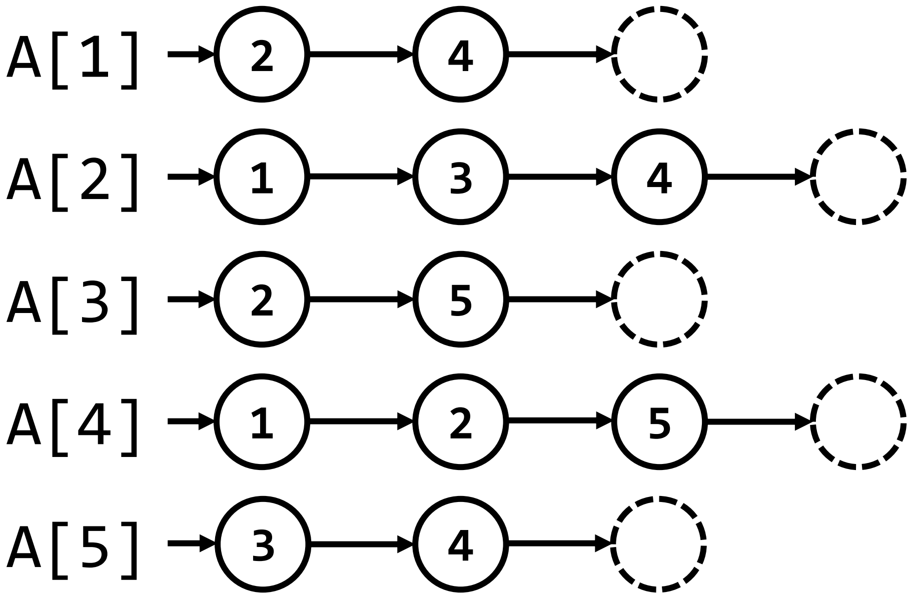
Cons
- It requires $O(N^2)$ space complexity even if the graph is spare → memory is wasted
- It is inefficient when adding or deleting a vertex

Implementation - Adjacent List

- **Adjacent List** represents neighbors of a vertex `u` as a linked list
  - Let `V = { 1, ..., N }` be the set of vertices
  - `A[u]` is the head pointer for the list of vertex `u`



G=(V,E)



Implementation - Adjacent List

- **Adjacent List** represents neighbors of a vertex `u` as a linked list
  - Let `V` = `{ 1, ..., N }` be the set of vertices
  - `A[u]` is the head pointer for the list of vertex `u`

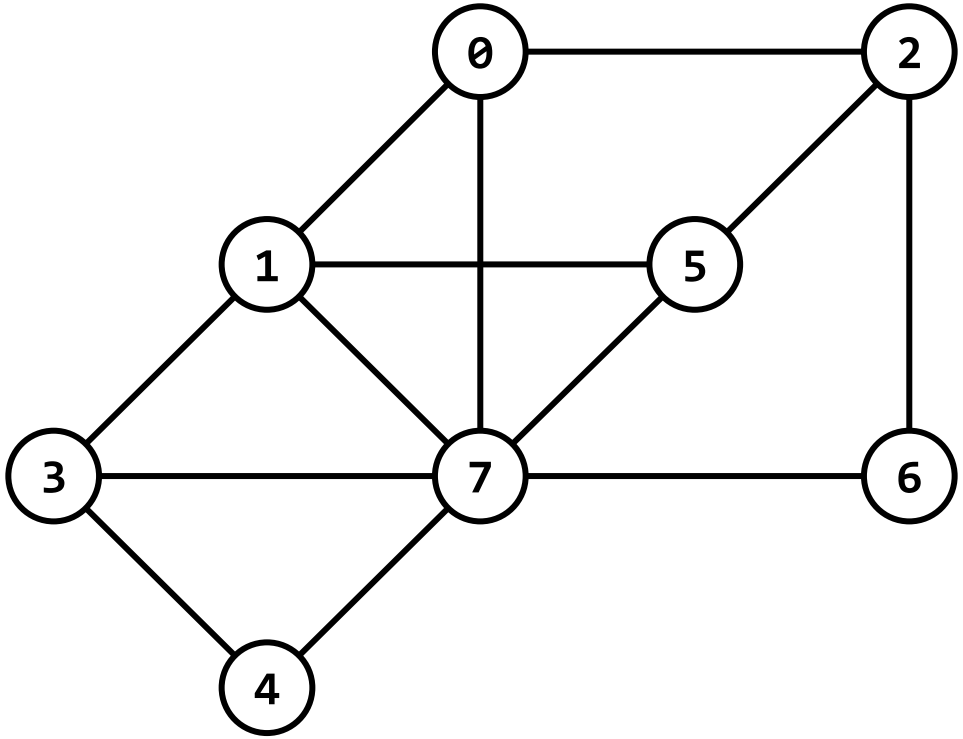
```
typedef struct _Vertex {
    int id;                 // vertex id
    struct _Vertex *next; // next vertex pointer for list of neighbors
} Vertex;

typedef struct _Graph {
    int size;        // # of vertices
    Vertex **heads; // array of head pointers for list of neighbors
} Graph;

Graph* createGraph(int size);
void removeGraph(Graph *G);
void addEdge(Graph *G, int u, int v);
void printGraph(Graph *G);
```

Implementation - Adjacent List

- Check the below example with your implementation



Graph Traversal

- **Graph Traversal** is the process of visiting all vertices once in a graph
  - How to visit them?
  - What is different from (Binary) **Tree Traversal**?

Graph Traversal

- **Graph Traversal** is the process of visiting all vertices once in a graph
  - How to visit them?
  - What is different from (Binary) **Tree Traversal**?

- (Recap) **Binary Tree Traversal**:

  Depth-First Search (DFS)
  - In-order traversal : `Left Subtree → Root → Right Subtree`
  - Pre-order traversal : `Root → Left Subtree → Right Subtree`
  - Post-order traversal : `Left Subtree → Right Subtree → Root`

  Breadth-First Search (BFS)
  - Level-order traversal : from top (`level=0`) to bottom (`level=height-1`)

Graph Traversal

- **Graph Traversal** is the process of visiting all vertices once in a graph
  - How to visit them?
  - What is different from (Binary) **Tree Traversal**?

- For Tree Traversal,
  - There is no cycle
  - There is an order based on parent-children relationship (e.g., bottom, top, level, …)
  - It is not required to care about that some node might be visited twice

Graph Traversal

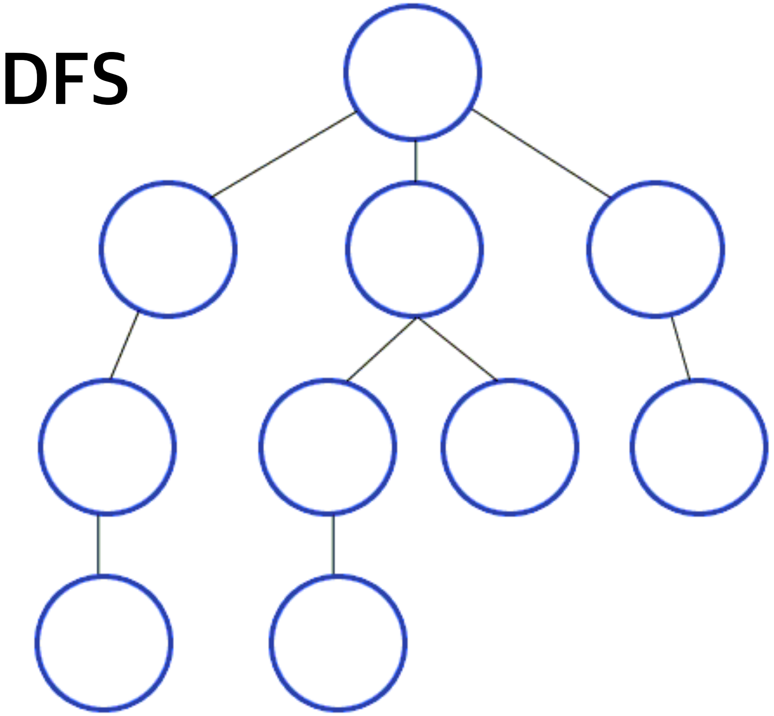
- **Graph Traversal** is the process of visiting all vertices once in a graph
    - How to visit them?
    - What is different from (Binary) **Tree Traversal**?

- For Graph Traversal,
    - There is a cycle
    - There is no order between vertices
    - Some node might be visited twice due to the existence of cycles
    - You must check whether a node was visited or not during traversal

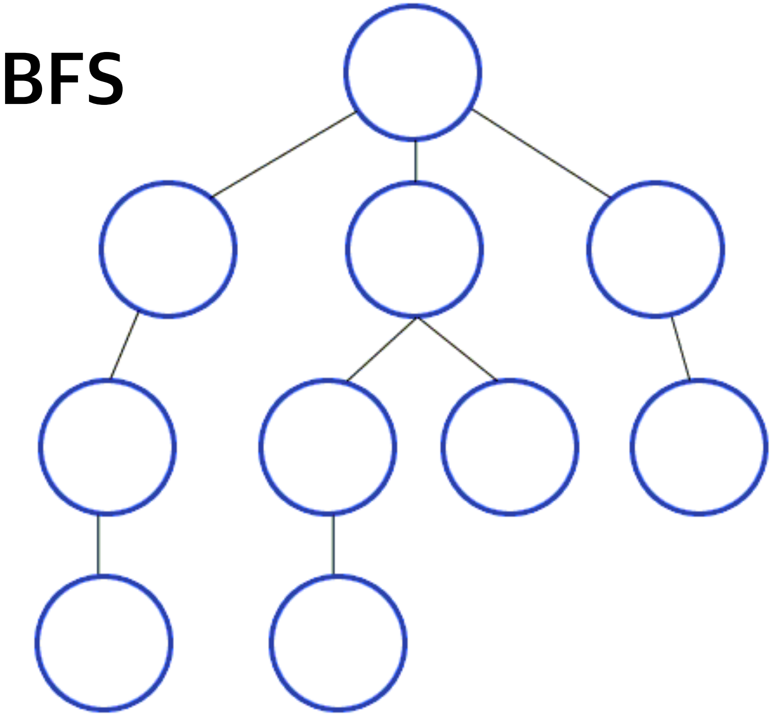
Graph Traversal

- Two common approaches for the traversal:
  - **Depth-First search (DFS)**
  - **Breadth-First search (BFS)**

DFS

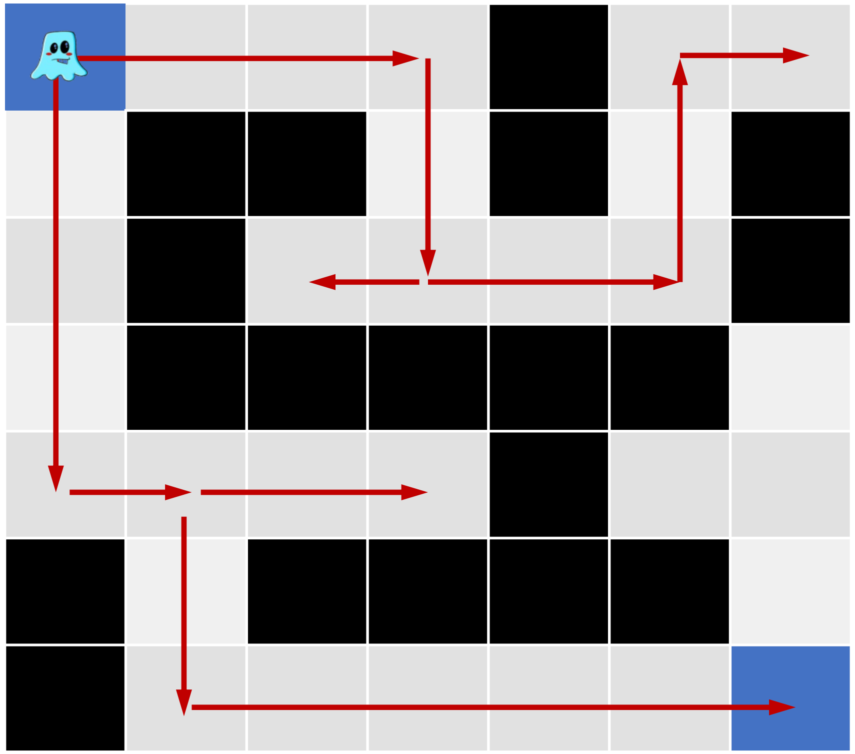


BFS



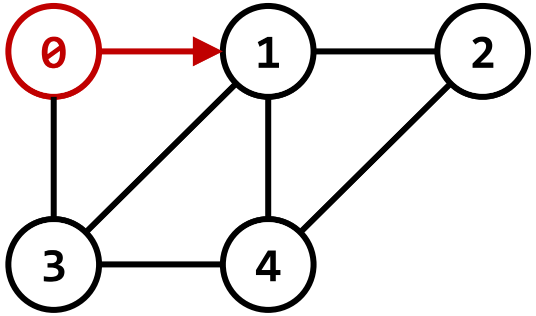
Graph Traversal - Depth-First Search

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



Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

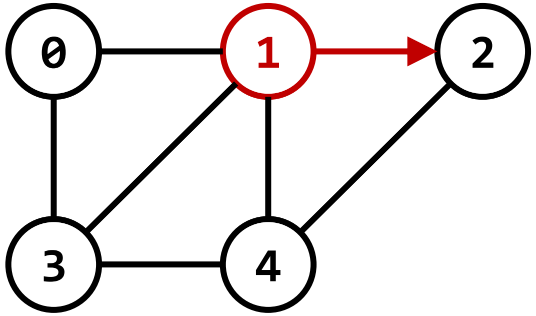


	0	1	2	3	4
visited	v				

	T				
stack	0				

Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

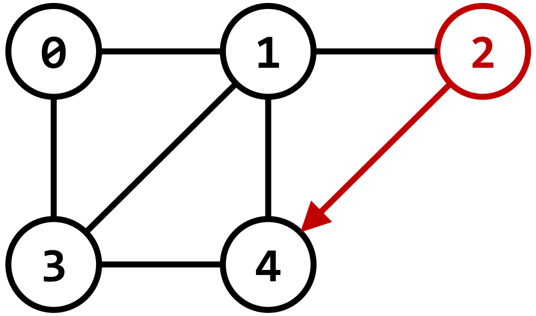


	0	1	2	3	4
visited	v	v			

		T			
stack	0	1			

Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

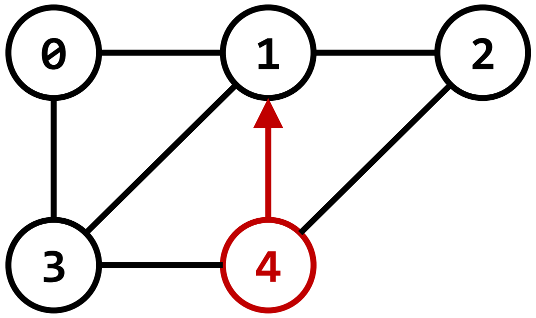


	0	1	2	3	4
visited	v	v	v		

			T		
stack	0	1	2		

Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

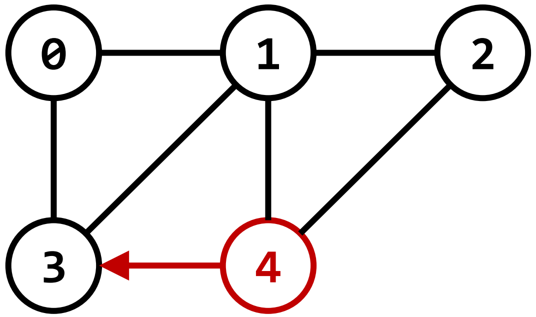


	0	1	2	3	4
visited	v	v	v		v

				T	
stack	0	1	2	4	

Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

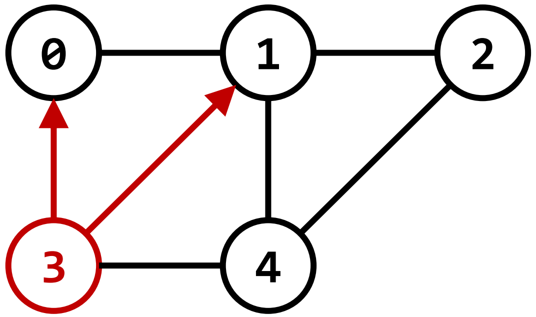


	0	1	2	3	4
visited	v	v	v		v

				T	
stack	0	1	2	4	

Graph Traversal - Depth-First Search

- **DFS** Algorithm with **Stack**
  - When inserting a vertex into the stack, check the vertex as **visited**
  - Push the starting vertex into the stack at the beginning
  - Treat the top element as the **currently visiting vertex**
  - Push an unvisited neighbor of **the current vertex**
  - Otherwise, pop **the current vertex** from the stack

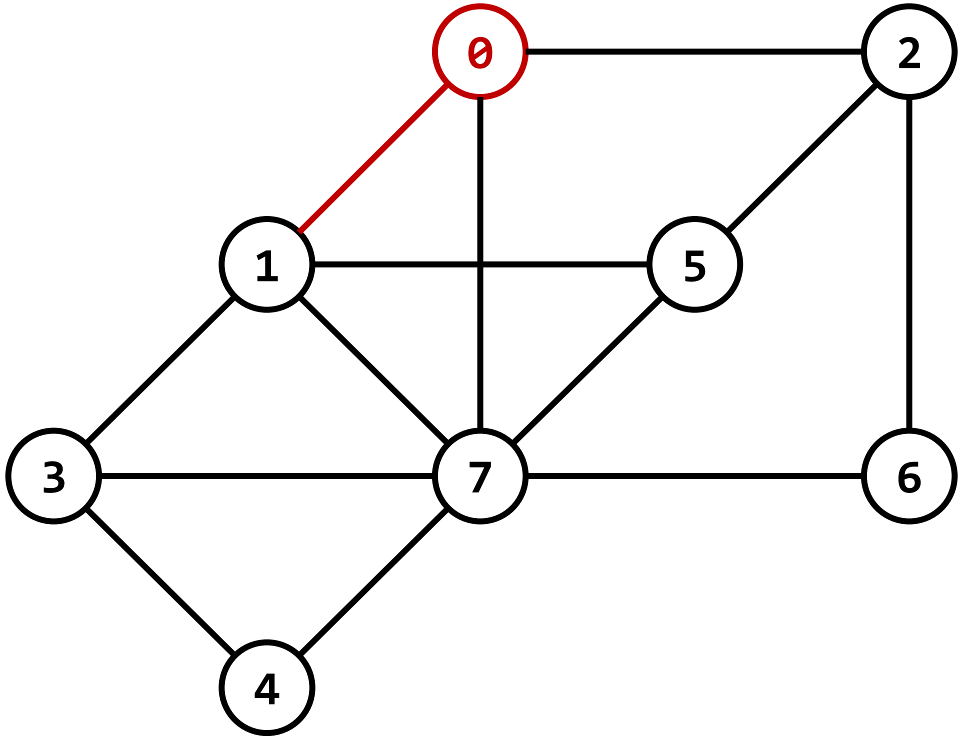


	0	1	2	3	4
visited	v	v	v	v	v

					T
stack	0	1	2	4	3

Example - DFS

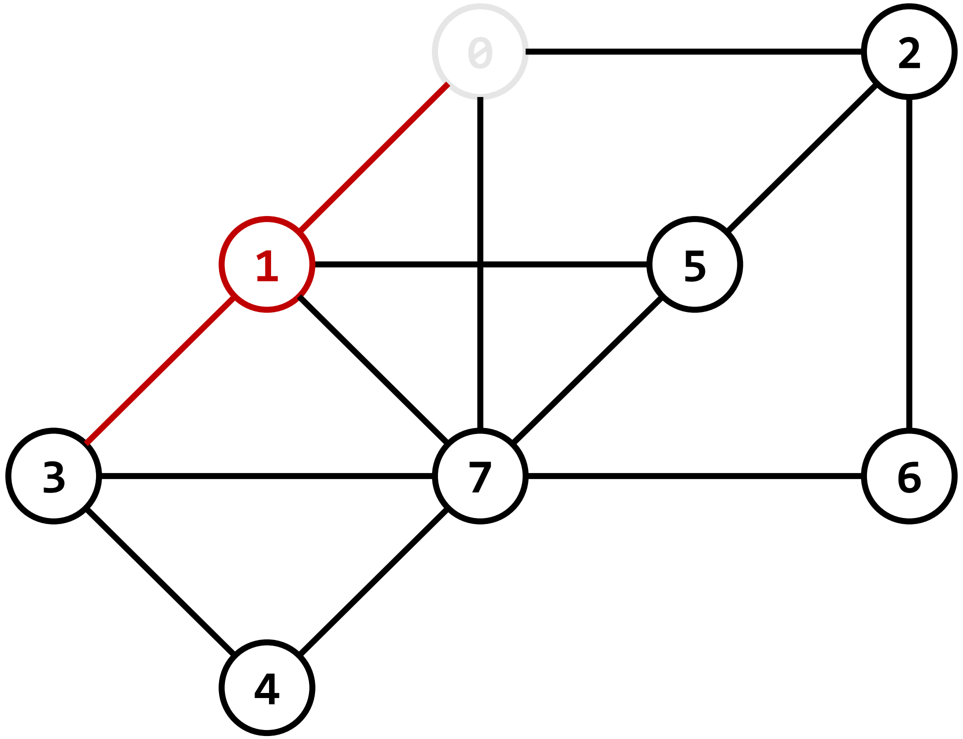
- Check the below example with your implementation



DFS: 0

Example - DFS

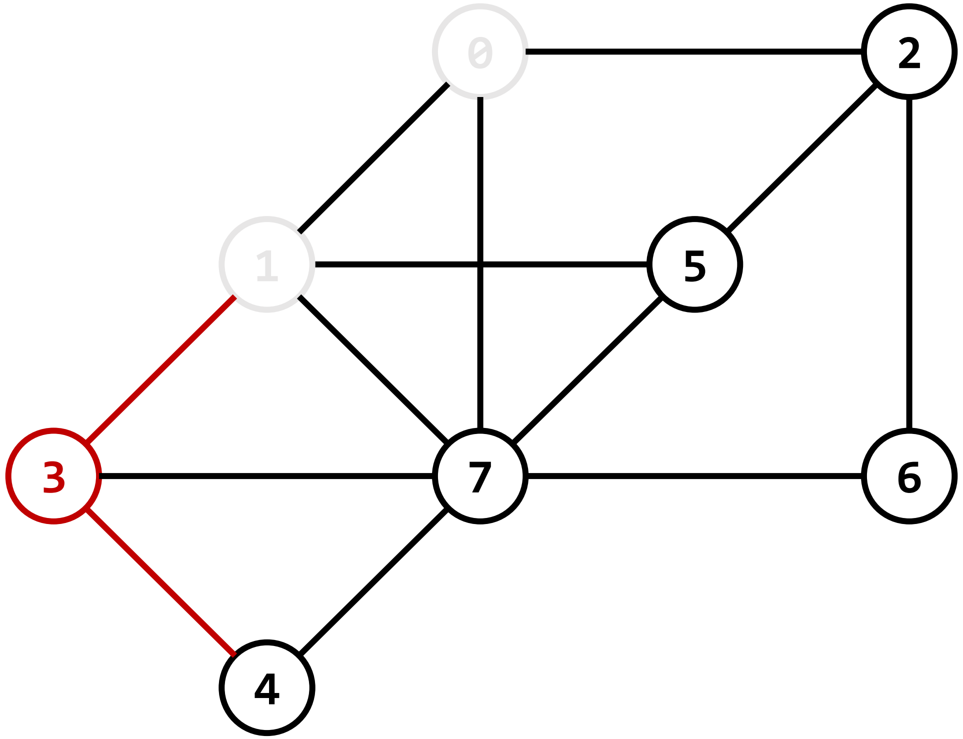
- Check the below example with your implementation



DFS: 0 1

Example - DFS

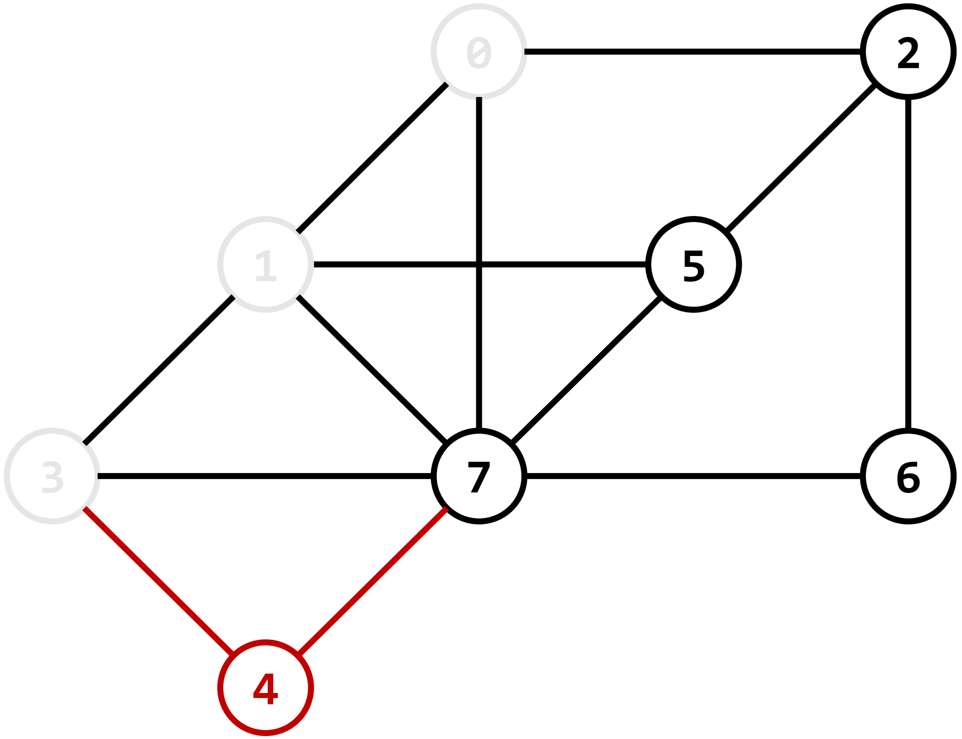
- Check the below example with your implementation



DFS: 0 1 3

Example - DFS

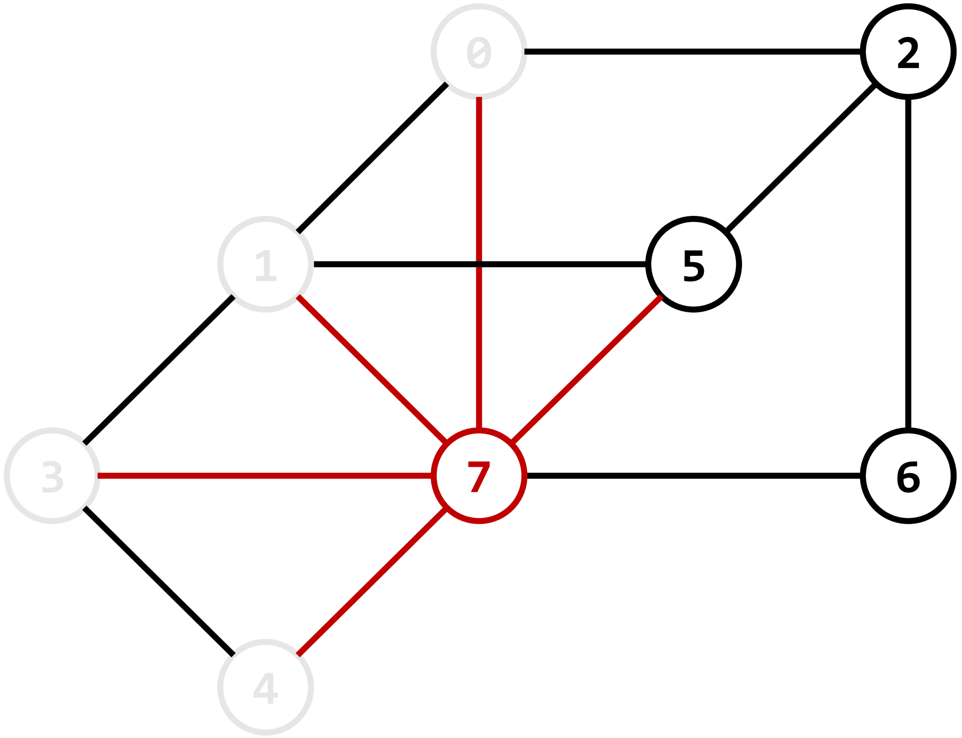
- Check the below example with your implementation



DFS: 0 1 3 4

Example - DFS

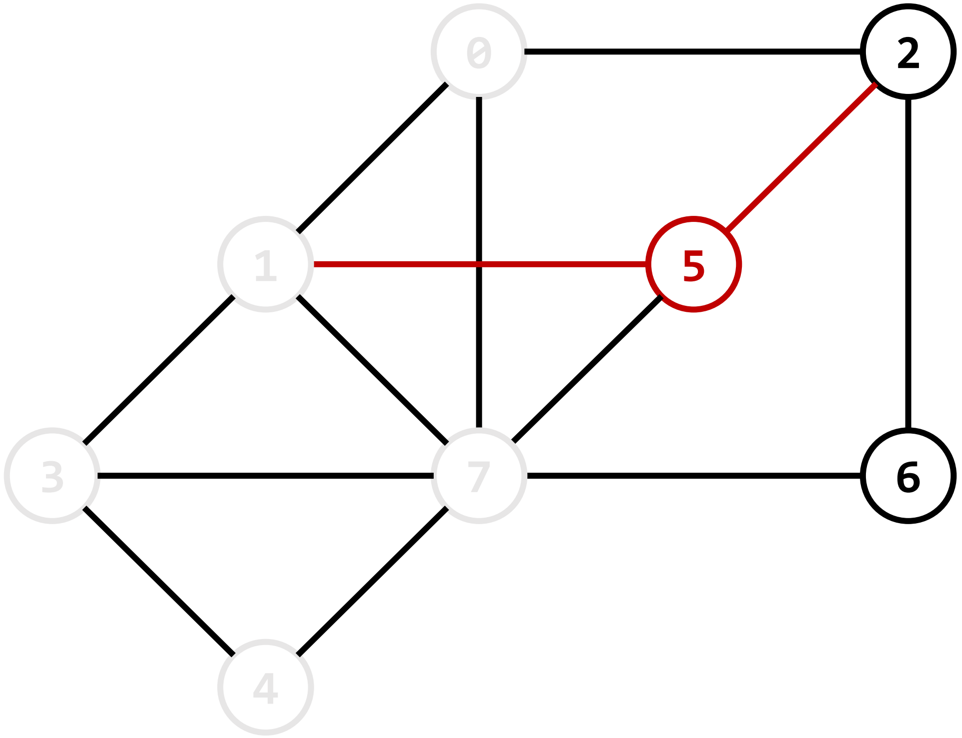
- Check the below example with your implementation



DFS: 0 1 3 4 7

Example - DFS

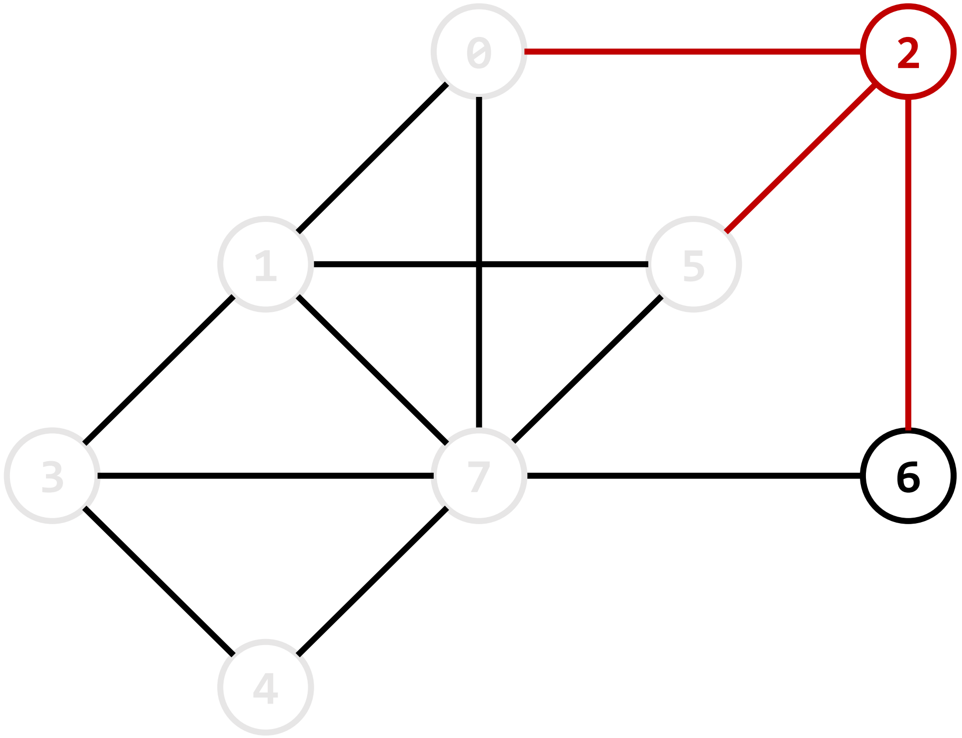
- Check the below example with your implementation



DFS: 0 1 3 4 7 5

Example - DFS

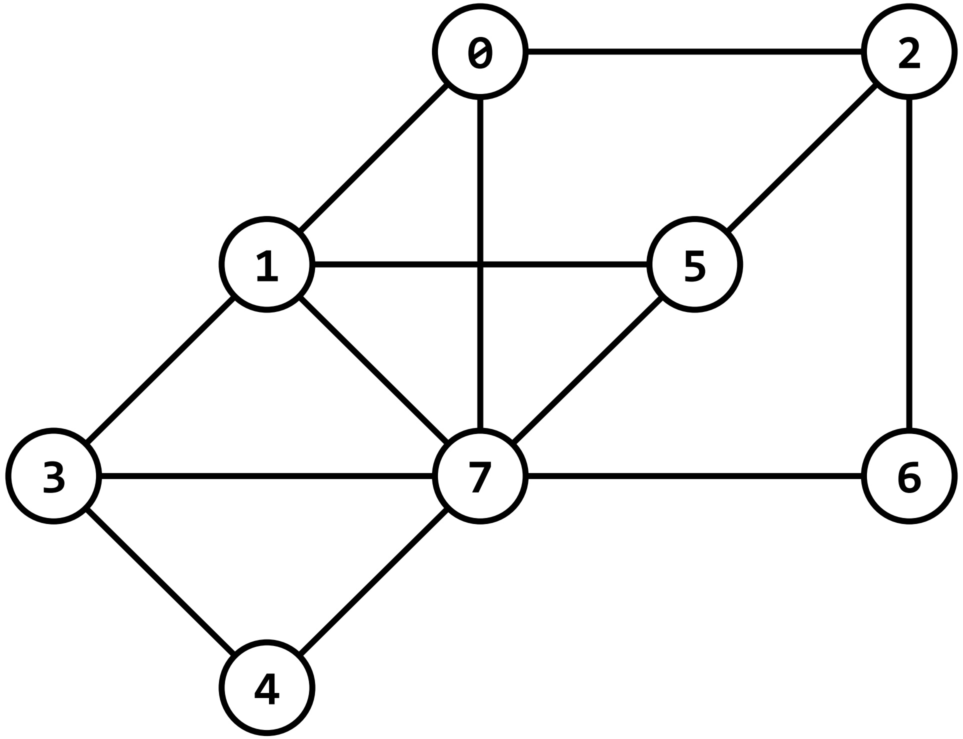
- Check the below example with your implementation



DFS: 0 1 3 4 7 5 2

Example - DFS

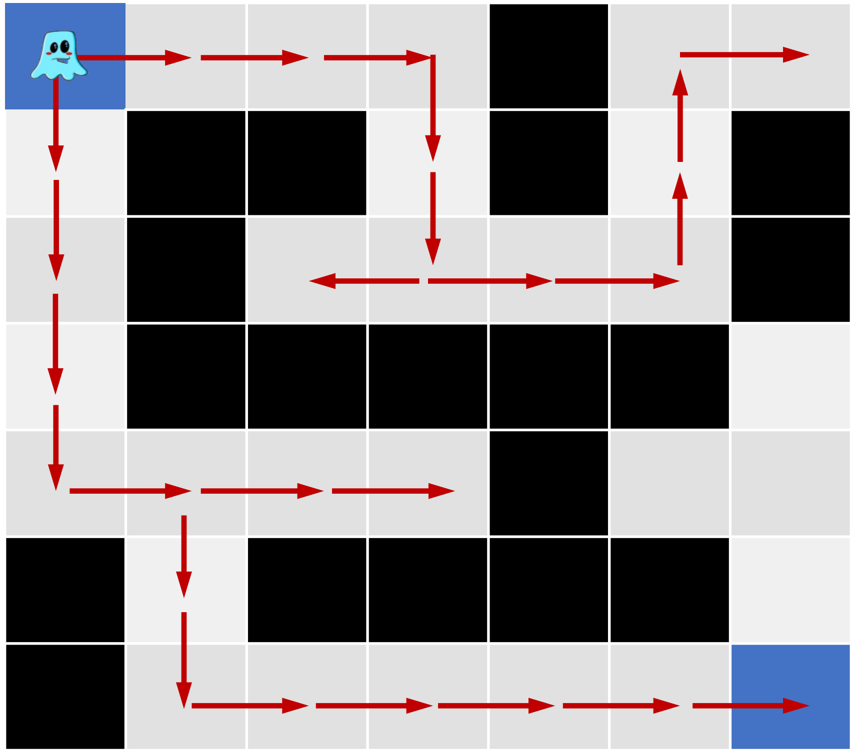
- Check the below example with your implementation



DFS: 0 1 3 4 7 5 2 6

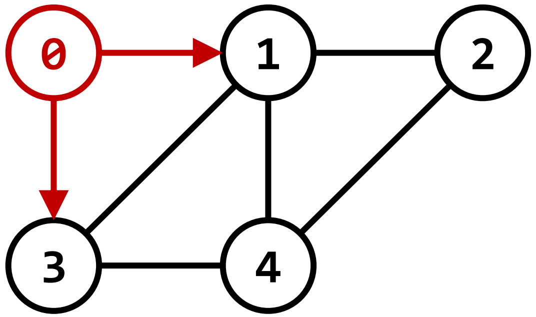
Graph Traversal - Breadth-First Search

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



Graph Traversal - Breadth-First Search

- **BFS** Algorithm with **Queue**
  - When inserting a vertex into the queue, check the vertex as **visited**
  - Enqueue the starting vertex into the queue at the beginning
  - Treat the front element as the **currently visiting vertex**
  - Enqueue all unvisited neighbors of **the current vertex**
  - Then, dequeue **the current vertex** from the queue

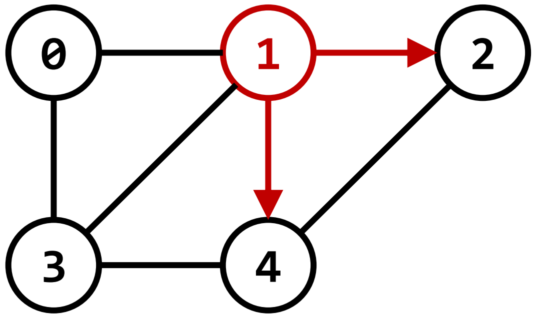


	0	1	2	3	4
visited	v				

	F	R			
queue	0				

Graph Traversal - Breadth-First Search

- **BFS** Algorithm with **Queue**
  - When inserting a vertex into the queue, check the vertex as **visited**
  - Enqueue the starting vertex into the queue at the beginning
  - Treat the front element as the **currently visiting vertex**
  - Enqueue all unvisited neighbors of **the current vertex**
  - Then, dequeue **the current vertex** from the queue

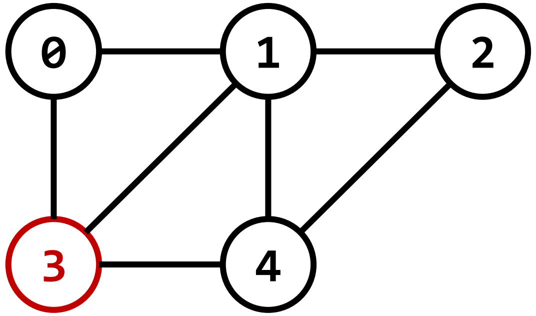


	0	1	2	3	4
visited	v	v		v	

		F		R	
queue	0	1	3		

Graph Traversal - Breadth-First Search

- **BFS** Algorithm with **Queue**
  - When inserting a vertex into the queue, check the vertex as **visited**
  - Enqueue the starting vertex into the queue at the beginning
  - Treat the front element as the **currently visiting vertex**
  - Enqueue all unvisited neighbors of **the current vertex**
  - Then, dequeue **the current vertex** from the queue

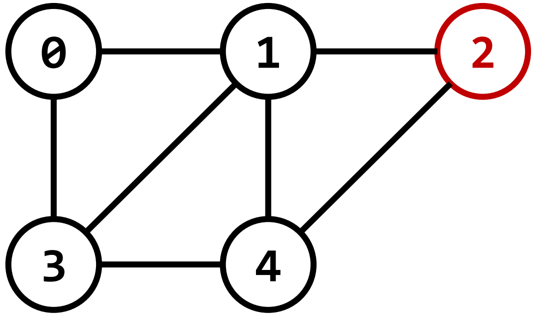


	0	1	2	3	4
visited	v	v	v	v	v

			F		
queue	0	1	3	2	4

Graph Traversal - Breadth-First Search

- **BFS** Algorithm with **Queue**
  - When inserting a vertex into the queue, check the vertex as **visited**
  - Enqueue the starting vertex into the queue at the beginning
  - Treat the front element as the **currently visiting vertex**
  - Enqueue all unvisited neighbors of **the current vertex**
  - Then, dequeue **the current vertex** from the queue

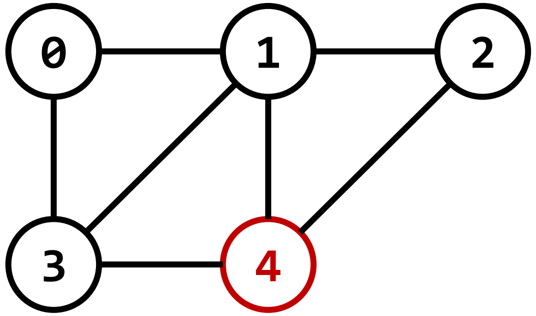


	0	1	2	3	4
visited	v	v	v	v	v

				F	
queue	0	1	3	2	4

Graph Traversal - Breadth-First Search

- **BFS** Algorithm with **Queue**
  - When inserting a vertex into the queue, check the vertex as **visited**
  - Enqueue the starting vertex into the queue at the beginning
  - Treat the front element as the **currently visiting vertex**
  - Enqueue all unvisited neighbors of **the current vertex**
  - Then, dequeue **the current vertex** from the queue

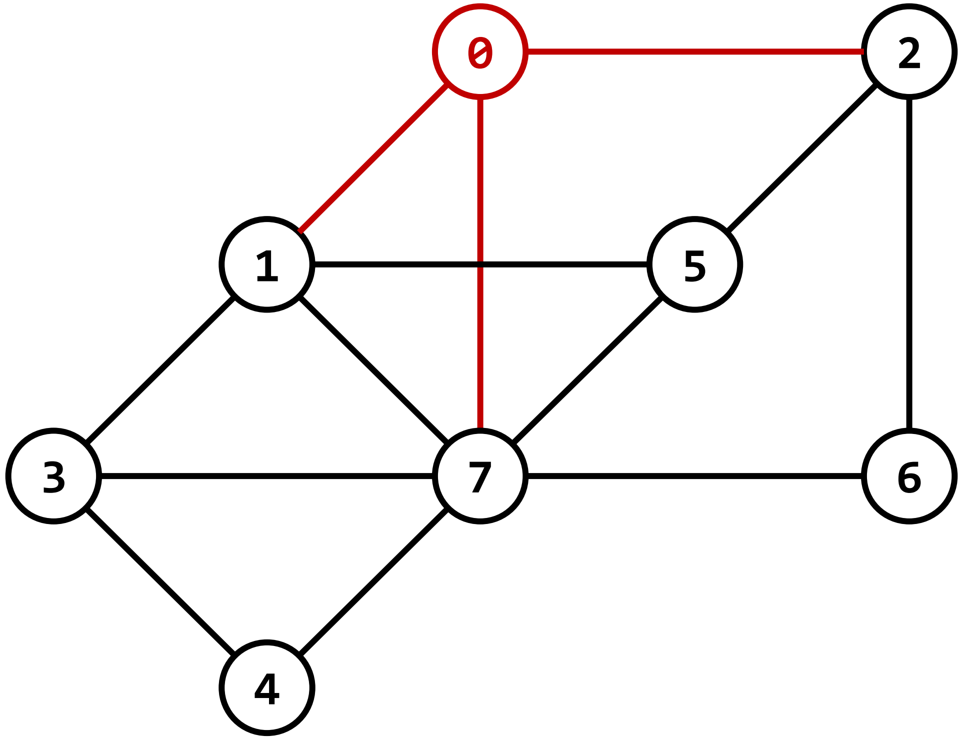


	0	1	2	3	4
visited	v	v	v	v	v

					F
queue	0	1	3	2	4

Example - BFS

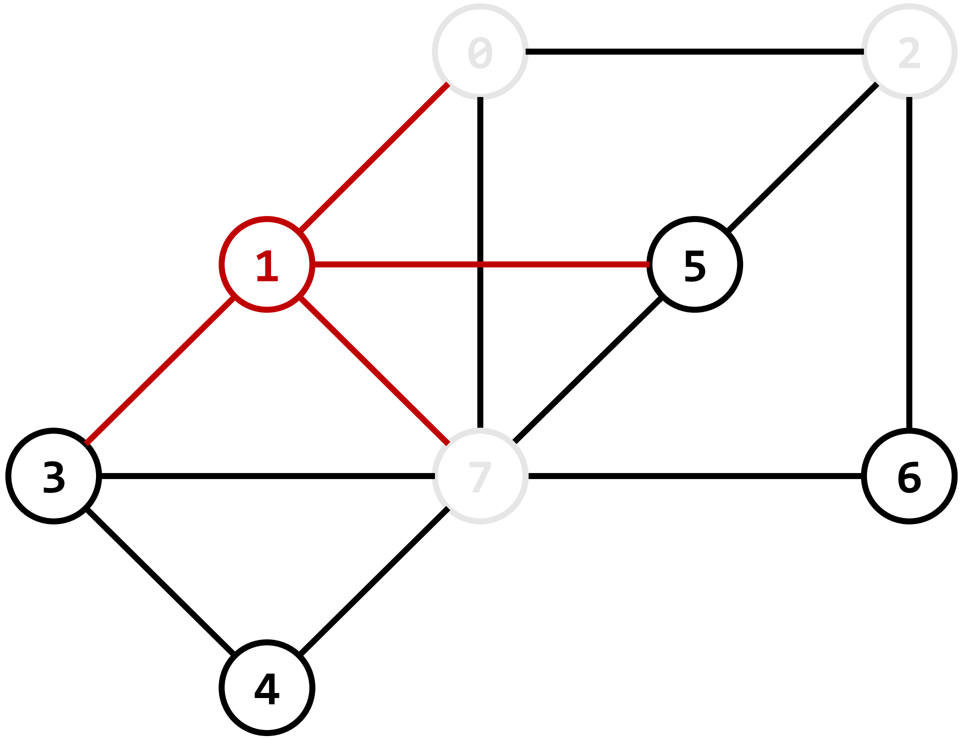
- Check the below example with your implementation



BFS: 0

Example - BFS

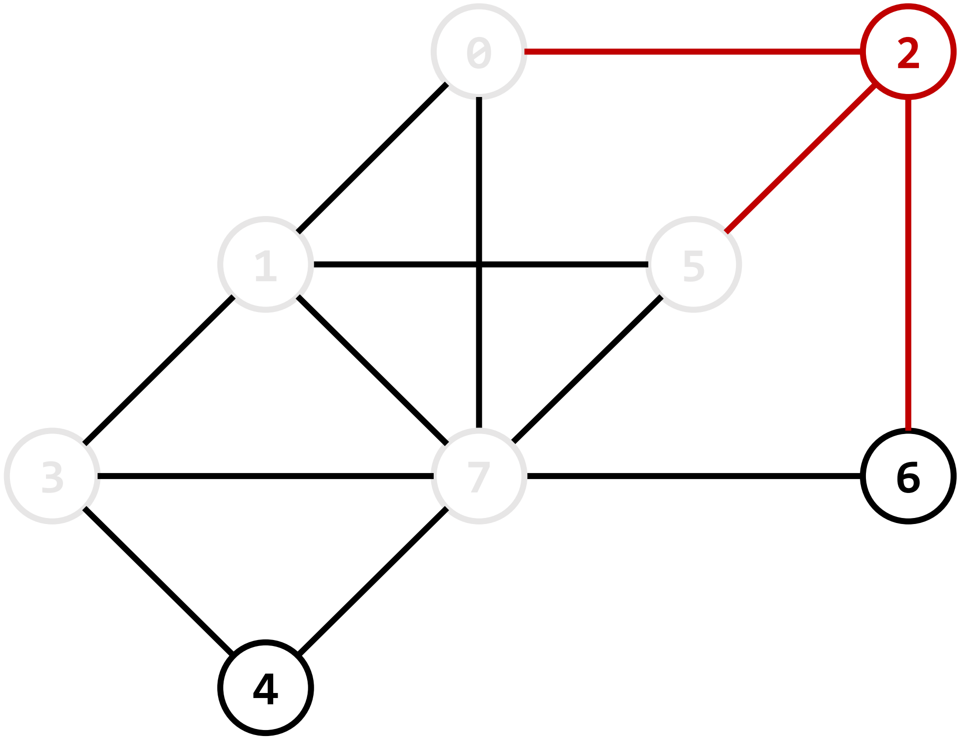
- Check the below example with your implementation



BFS: 0 1 2 7

Example - BFS

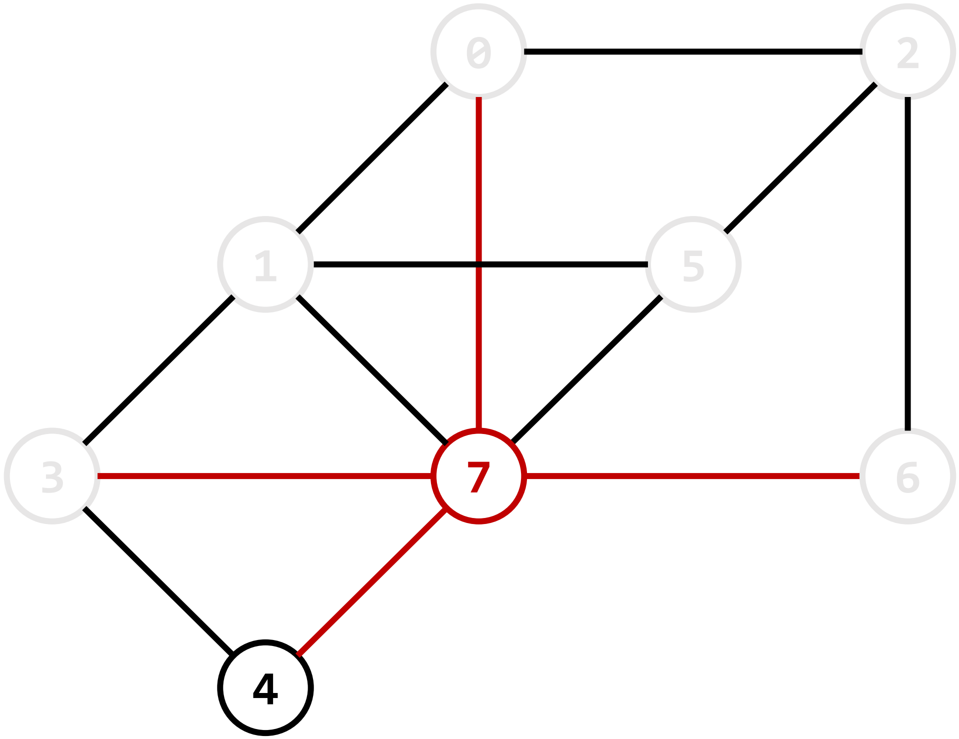
- Check the below example with your implementation



BFS: 0 1 2 7 3 5

Example - BFS

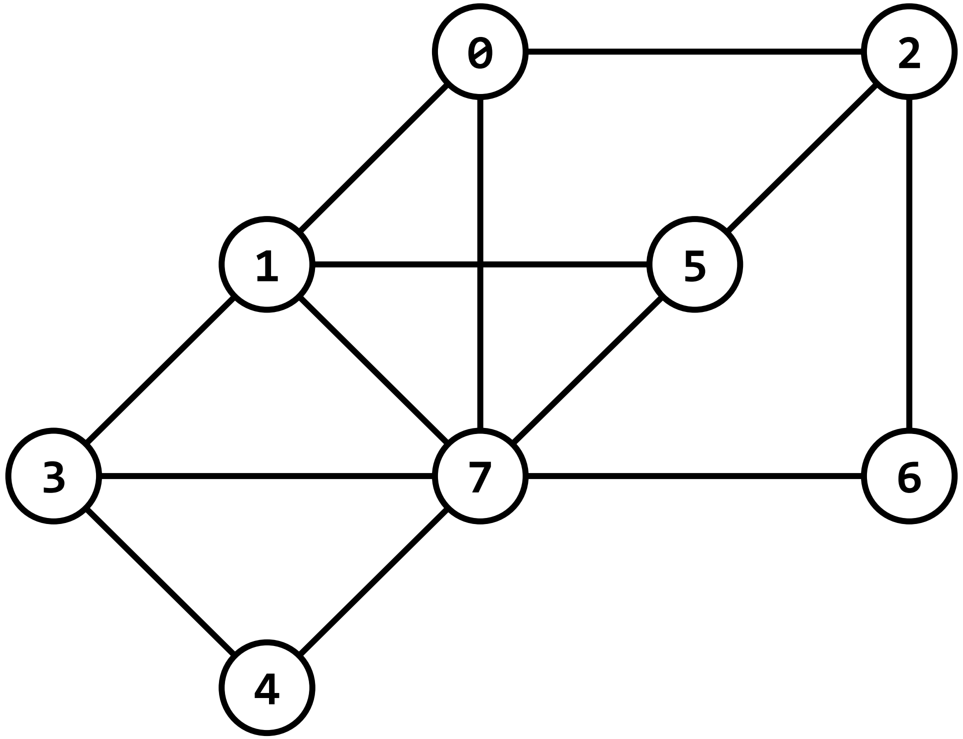
- Check the below example with your implementation



BFS: 0 1 2 7 3 5 6

Example - BFS

- Check the below example with your implementation



BFS: 0 1 2 7 3 5 6 4

Summary of DFS and BFS

- Implementations
  - DFS can be implemented with Stack
  - BFS can be implemented with Queue

- Time Complexities
  - $O(|V|^2)$ with the adjacent matrix
  - $O(|V| + |E|)$ with the adjacent list

- Using traversal algorithms, one can find connected components
  - All visited vertices after traversal form a **connected component**

Any Questions?