# MTH 4300/4299

**Introduction to Graphs**

Jaime Abbariao

# What is a Graph?

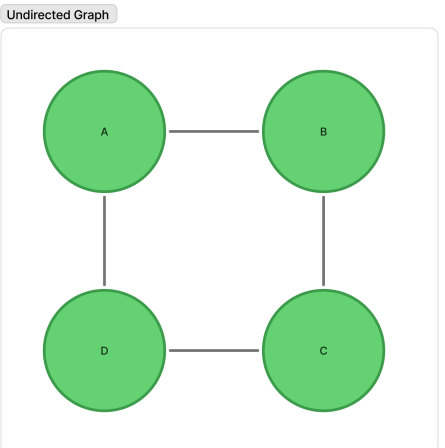


Figure 1: A graph is a collection of vertices (nodes) connected by edges

# What is a Graph? (ii)

**Key Components:**

- **Vertices (V)**: The nodes `(A, B, C, D)`
- **Edges (E)**: The connections between nodes
- **Graph G = (V, E)**: The complete structure
- **Degree of a vertex**: the number of edges that are incident to a vertex

# Types of Graphs

**Undirected Graph:**

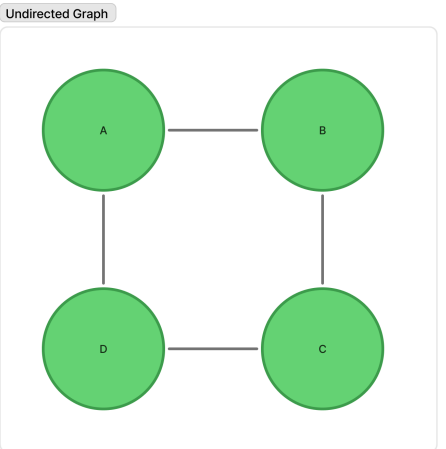


Figure 2: Edges have no directions

# Types of Graphs (ii)

An undirected graph means that we're allowed to travel on the edges between nodes in any direction.

Referencing our image above, this means that we can go from node `A` to node `B` and from node `B` to node `A`.

# Types of Graphs (iii)

**Directed Graph (Digraph):**

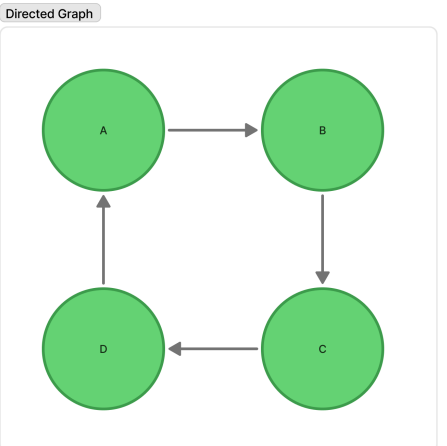


Figure 3: Edges have directions (arrows)

# Types of Graphs (iv)

Unlike the undirected graph, the directed graph enforces that nodes can only travel to other nodes in a specific direction.

In our example above, we can see that there is a directed edge from node `A` to node `B`. This means that we can only go from `A` to `B` and not the other way around.

# More Graph Types

**Weighted Graph:**

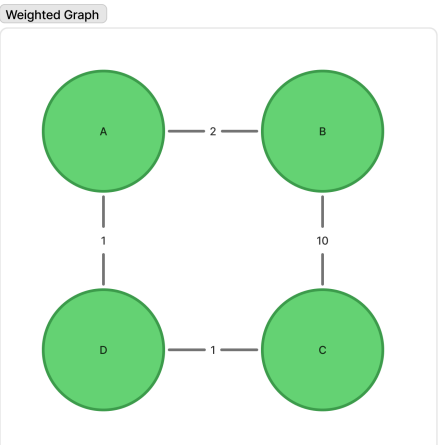


Figure 4: Edges have weights/costs

# More Graph Types (ii)

Unweighted graphs treat all connections as equal, but when using graphs to model real relationships, these connections all have some cost associated with them.

Figure 5: **Social networks** model the strength of relationships between users

# More Graph Types (iii)

Some other real world examples:

- **GPS Navigation**: prefers highways to local roads (speed here being one of the potential weights/cost we can use)
- **Flight connections**: minimizing travel time, ticket costs, etc and not just the number of stops

# More Graph Types (iv)

**Cyclic vs Acyclic:**

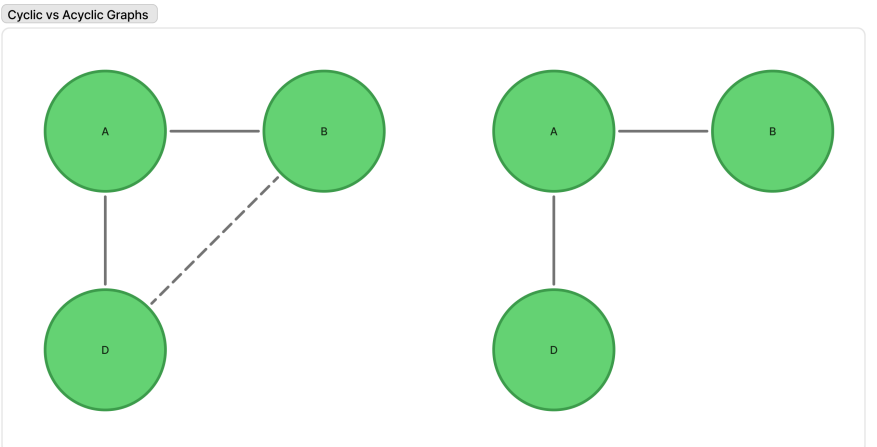


Figure 6: Cyclic vs Acyclic Graphs

# More Graph Types (v)

A **cyclic** graph:

- Contains at least one cycle (a path that starts and ends at the same vertex)
- You can follow the edges and return to your starting point

An **acyclic** graph:

- Contains no cycles
- No path exists that leads back to the starting vertex
- Trees are the most common example of acyclic graphs

# Graph Representations in C++

Two main approaches:

1. **Adjacency Matrix**
2. **Adjacency List**

Each has different trade-offs for memory and performance.

# Adjacency Matrix

2D array where `matrix[i][j] = 1` if edge exists from vertex `i` to vertex `j`.

```
// For graph: A-B, A-C, B-D, C-D
//     A B C D
// A [ 0 1 1 0 ]
// B [ 1 0 0 1 ]
// C [ 1 0 0 1 ]
// D [ 0 1 1 0 ]

std::vector<std::vector<int>> adj_matrix(4, std::vector<int>(4,
0));
adj_matrix[0][1] = 1; // A to B
adj_matrix[1][0] = 1; // B to A (undirected)
```

- **Pros:** $O(1)$ edge lookup, simple
- **Cons:** $O(V^2)$ space, even for sparse graphs

# Adjacency Matrix (ii)

However, the above example only works if you have simple vertices. What can we do if our vertices are a bit more complex?

There are two strategies we can consider here:

1. Mapping the more complex structure to a simple index
2. Use an adjacency list instead

# Mapping strategy

For example, let's suppose that the entries to our graph are object of the following `struct`:

```
struct User {
  std::string UUID; // a universally unique identifier
  std::string name;
  std::string address;
  // Omitting the rest of the implementation
};
```

If we wanted to use an adjacency matrix, we don't have a simple way to construct it! This means we need to create a method that allows us to easily go from an object to an index.

# Mapping strategy (ii)

```
class UserGraph {
  std::vector<std::vector<int>> adj_matrix;
  std::unordered_map<std::string, int> user_uuid_to_index;
  int vertex_count;
public:
  UserGraph(): vertex_count(0) {}
};
```

It's important to note here that we'll map some `string` to the simple index using an `unordered_map`.

In the case of our `User` object, we're going to store the `UUID -> simple index` relationship.

# Mapping strategy (iii)

```
// in the same class as above
  int add_vertex(User *a) {
    if (user_uuid_to_index.contains(a->UUID)) { return
user_uuid_to_index[a->UUID]; }

    user_uuid_to_index[a->UUID] = vertex_count;

    for (auto &row : adj_matrix) {
      row.push_back(0);
    }

    adj_matrix.push_back(std::vector<int>(vertex_count + 1,
0));

    return vertex_count++;
  }
```

# Mapping strategy (iv)

```
// in the same class as above
  void add_edge(User *a, User *b) {
    int a_index = add_vertex(a);
    int b_index = add_vertex(b);

    adj_matrix[a_index][b_index] = 1;
    adj_matrix[b_index][a_index] = 1;
  }
```

Using the `add_vertex` method we just implemented, we can then grab the appropriate simple index from the `User` object!

Once we have this, we can simply set the values in our adjacency matrix.

# Mapping strategy (v)

```
// in the same class as above
  bool has_edge(User *a, User *b) {
    if (
      !user_uuid_to_index.contains(a->UUID) ||
      !user_uuid_to_index.contains(b->UUID)
    ) {
      return false;
    }

    auto a_index = user_uuid_to_index[a->UUID];
    auto b_index = user_uuid_to_index[b->UUID];

    return (
      adj_matrix[a_index][b_index] != 0 ||
      adj_matrix[b_index][a_index] != 0
    )
  }
```

# Adjacency List

Each vertex stores a list of its neighbors.

```
// For same graph: A-B, A-C, B-D, C-D
std::vector<std::vector<int>> adj_list = {
    {1, 2},     // A connected to B(1), C(2)
    {0, 3},     // B connected to A(0), D(3)
    {0, 3},     // C connected to A(0), D(3)
    {1, 2}      // D connected to B(1), C(2)
};
```

If you wanted to support an adjacency list this way, you would have to create again another mapping method for your graph to keep track of the appropriate index for your vertex.

However, when typically working with adjacency lists, the more common approach is to use a Map!

# Adjacency List (ii)

By using a map, we're able to pick out an identifier as our key and have a vector of other identifiers as the value.

```
std::unordered_map<std::string, std::vector<std::string>>
graph;
graph["A"] = {"B", "C"};
graph["B"] = {"A", "D"};
```

- **Pros:** $O(V + E)$ space, efficient for sparse graphs
- **Cons:** $O(\text{degree})$ edge lookup

# Graph Class Implementation

```
class Graph {
private:
    std::unordered_map<int, std::vector<int>> adj_list;

public:
    Graph(int v) : adj_list(v) {}

    void print_graph() {
        for (const auto &[key, value] : adj_list) {
            std::cout << "Vertex " << key << ": ";
            for (int neighbor : value) {
                std::cout << neighbor << " ";
            }
            std::cout << std::endl;
        }
    }
};
```

# Graph Class Implementation (ii)

```
// in the same class as above
    void add_edge(int u, int v) {
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }
```

The implementation above isn't the most thorough, but it gets the point across. If we want to add an edge into the graph, then you should get the `vector` by `adj_list[u]` and then just call `push_back`

# Graph Traversal: Depth-First Search (DFS)

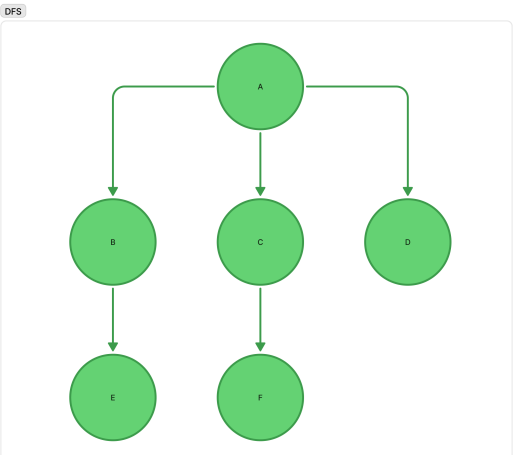


Figure 7: Visit as far as possible along each branch before backtracking

# Graph Traversal: Depth-First Search (DFS) (ii)

**Go as deep as possible before backtracking**

When you start thinking about DFS, think of it like exploring a maze - you follow one path until you hit a dead end, then backtrack to try the next unexplored path.

One thing to note here is that when you're implementing this algorithm, you want to make sure that you mark paths you've already explored!

This makes sure that we're only exploring new paths.

# DFS Implementation

```
// assume this code exists within the other Graph class
    void dfs_helper(int vertex, std::vector<bool>& visited) {
        visited[vertex] = true;
        std::cout << vertex << " ";
        for (int neighbor : adj_list[vertex]) {
            if (!visited[neighbor]) {
                dfs_helper(neighbor, visited);
            }
        }
    }
    void dfs(int start) {
        std::vector<bool> visited(vertices, false);
        std::cout << "DFS traversal starting from " << start <<
": ";
        dfs_helper(start, visited);
        std::cout << std::endl;
    }
```

# Graph Traversal: Breadth-First Search (BFS)

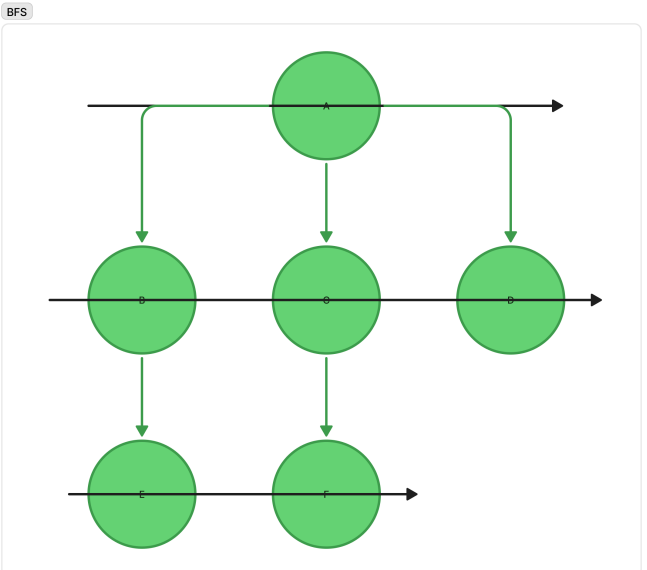


Figure 8: Visit all vertices at current level before moving to next level

# BFS Implementation

```
// this is from the Graph class
    void bfs(int start) {
        std::vector<bool> visited(vertices, false);
        std::queue<int> queue;

        visited[start] = true;
        queue.push(start);

        std::cout << "BFS traversal starting from " << start <<
": ";

        while (!queue.empty()) {
            int current = queue.front();
            queue.pop();
            std::cout << current << " ";

            for (int neighbor : adj_list[current]) {
```

# BFS Implementation (ii)

```
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.push(neighbor);
                }
            }
        }
        std::cout << std::endl;
    }
```

# Performance Comparison

| Operation | Adjacency Matrix | Adjacency List |
|---|---|---|
| Add Edge | $O(1)$ | $O(1)$ |
| Remove Edge | $O(1)$ | $O(\text{degree})$ |
| Check Edge | $O(1)$ | $O(\text{degree})$ |
| Space | $O(V^2)$ | $O(V + E)$ |
| Get All Neighbors | $O(V)$ | $O(\text{degree})$ |

**Choose based on your use case:**

- Dense graphs $\longrightarrow$ Matrix
- Sparse graphs $\longrightarrow$ List
- Many edge queries $\longrightarrow$ Matrix
- Memory constrained $\longrightarrow$ List

# Lab

1. Given an undirected graph and two vertices, determine if a valid path exists between them.

```
// Example: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0,
destination = 2
// Output: true
bool validPath(int n, std::vector<std::vector<int>> &edges, int
source, int destination) {
  // Build adjacency list, use DFS/BFS from source
}
```

# Lab (ii)

2. A star graph has one center node connected to all other nodes. Find the center.

```
// Example: edges = [[1,2],[2,3],[4,2]]
// Output: 2 (node 2 is connected to all others)
int findCenter(std::vector<std::vector<int>> &edges) {
  // Use degree counting or simple edge analysis
}
```