MTH 4300/4299: Programming and Computer Science II

Lecture 23: Graphs and Traversals

Jaime Abbariao

Announcements

Midterm 2 Statistics

Grades were good!

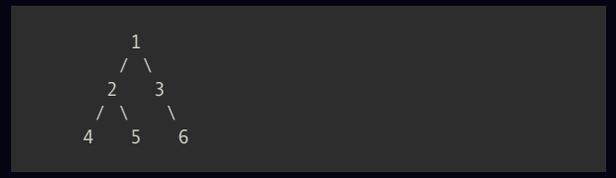
Median was 30. Average was 25.

Final Exam

- May 18th, 2026 from 6-8pm at B-6-118.
- I will not be there in person, but the covering instructor will be briefed.
- Topics from the whole semester are fair game, with a focus on:
  - Trees (Binary Trees, BSTs, Tries)
  - Graphs (BFS, DFS, Topological Sort) — today's lecture

Warm-up

Consider the following binary tree:



Write down the level-order traversal of this tree.

Hint: you used a `std::queue<TreeNode*>` to do this in Lecture 21.

▓▓▓ Warm-up Solution

Level-order traversal: `1 2 3 4 5 6`

Algorithm:

1. Push the root into a queue.
2. While the queue is not empty: pop the front, visit it, push its children.

Setup question for today:

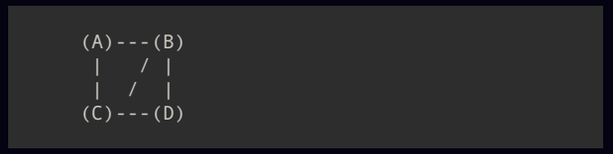
What if the tree had a cycle — say, node 6 also pointed back to node 2? What would `level-order` do?

It would loop forever. We'd push 2, visit it, push 4 and 5, eventually push 6, then push 2 again from 6, then push 4 and 5 again ...

This is exactly the problem graphs force us to deal with.

What is a Graph?

A graph is a set of vertices (nodes) connected by edges.



Real-world graphs you already use:

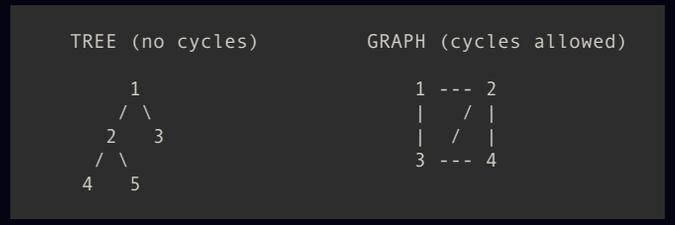
- Twitter follows: vertices = users, edges = "X follows Y" (directed)
- Facebook friends: vertices = users, edges = "X is friends with Y" (undirected)
- Google Maps: vertices = intersections, edges = roads with miles (weighted)
- Course prerequisites: vertices = courses, edges = "must take X before Y" (directed)
- The web: vertices = pages, edges = hyperlinks (directed)

▓▓ Graph Vocabulary

Term	Meaning	Example
Directed	edges have a direction (one-way)	Twitter follows
Undirected	edges go both ways	Facebook friends
Weighted	edges carry a number (cost, distance)	Google Maps roads
Unweighted	edges are just "connected or not"	Friends list
Cycle	path that returns to its start	A -> B -> C -> A
Connected component	maximal set of mutually reachable vertices	Two disjoint friend groups
Degree	number of edges touching a vertex	A has degree 2

For today, all examples are undirected and unweighted unless we say otherwise. The directed case (topological sort) comes later.

▓▓ Trees vs Graphs (and why we need a `visited` set)



A tree is a special kind of graph: connected, no cycles, exactly `n-1` edges for `n` vertices.

In a tree, recursion / level-order just works — you can never revisit a node.

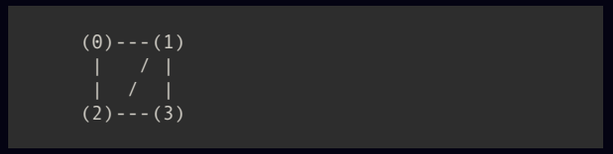
In a graph, you can reach the same node by multiple paths (or in a cycle). Without bookkeeping, traversal loops forever.

The fix: keep a `visited` set. Mark a vertex when you first see it. Never enqueue or recurse into it again.

> The `visited` set is the heart of every graph algorithm today. Remember it.

▓▓ Three Ways to Represent a Graph

Same graph, drawn three different ways:



Edge list — a list of pairs:

```
{(0,1), (0,2), (1,2), (1,3), (2,3)}
```

Adjacency matrix — `n x n` table; `M[i][j] = 1` if `i` and `j` are connected:

```
    0  1  2  3
0 [ 0  1  1  0 ]
1 [ 1  0  1  1 ]
2 [ 1  1  0  1 ]
3 [ 0  1  1  0 ]
```

Adjacency list — for each vertex, the list of its neighbors:

```
0: [1, 2]
1: [0, 2, 3]
2: [0, 1, 3]
3: [1, 2]
```

Tradeoffs: matrix is `O(V^2)` space (great if dense), adjacency list is `O(V + E)` (great if sparse). We will use adjacency lists.

▓▓▓ Adjacency List in C++

We'll label vertices `0, 1, 2, ..., n-1` and use:

```
std::vector<std::vector<int>> adj;
```

- The outer index `i` is a vertex id.
- The inner vector `adj[i]` holds the neighbors of vertex `i`.

For our 4-node graph:

```
#include <iostream>
#include <vector>

int main() {
  int n = 4;
  std::vector<std::vector<int>> adj(n);

  // This is called a lambda
  auto add_edge = [&](int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);  // undirected: add both directions
  };

  add_edge(0, 1);
  add_edge(0, 2);
  add_edge(1, 2);
  add_edge(1, 3);
  add_edge(2, 3);

  for (int i = 0; i < n; ++i) {
    std::cout << i << ": ";
    for (int v : adj[i]) std::cout << v << " ";
    std::cout << std::endl;
  }
  return 0;
}
```

> If your vertices are names (like `"Math"`, `"CS"`), map them to integers with a `std::unordered_map<std::string,int>` and reuse this code.

▓▓ Exercise: Build a Graph

Given these edges of an undirected graph with 5 vertices `(0..4)`:

```
(0,1), (0,4), (1,2), (1,3), (3,4)
```

Build the adjacency list using `std::vector<std::vector<int>>` and print each vertex's neighbors.

Exercise Solution

```cpp
#include <iostream>
#include <vector>
#include <utility>

int main() {
  int n = 5;
  std::vector<std::pair<int,int>> edges = {
    {0,1}, {0,4}, {1,2}, {1,3}, {3,4}
  };

  std::vector<std::vector<int>> adj(n);
  for (auto& e : edges) {
    adj[e.first].push_back(e.second);
    adj[e.second].push_back(e.first);
  }

  for (int i = 0; i < n; ++i) {
    std::cout << i << ": ";
    for (int v : adj[i]) std::cout << v << " ";
    std::cout << std::endl;
  }
  return 0;
}
```

Output:

```
0: 1 4
1: 0 2 3
2: 1
3: 1 4
4: 0 3
```

▓▓ BFS on a Graph

Remember level-order traversal? Same queue pattern — plus a `visited` set so we don't loop forever.

Invariant: at any moment, every vertex in the queue is at distance `d` or `d+1` from the source. That's why BFS gives you shortest path lengths in unweighted graphs for free.

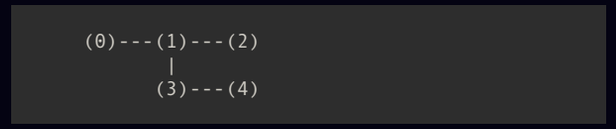
```cpp
#include <queue>
#include <vector>

void bfs(int start, const std::vector<std::vector<int>>& adj) {
  int n = adj.size();
  std::vector<bool> visited(n, false);
  std::queue<int> q;

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

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

    for (int v : adj[u]) {
      if (!visited[v]) {
        visited[v] = true;
        q.push(v);
      }
    }
  }
}
```

> What if we removed `visited`? On a graph with a cycle (like 0-1-2-0), the queue would push the same vertices forever — infinite loop. The bug is the lesson: graphs need `visited`.

▓▓ Exercise: Shortest Path Length with BFS

Given an unweighted, undirected graph and a source vertex `s`, return a vector `dist` where `dist[v]` is the number of edges on the shortest path from `s` to `v`. Use `-1` if `v` is unreachable.

Test on this graph with `s = 0`:



Expected: `dist = [0, 1, 2, 2, 3]`.

Hint: same BFS, but instead of marking `visited[v] = true`, store `dist[v] = dist[u] + 1`.

Try it for 5 minutes.

Exercise Solution

```cpp
#include <iostream>
#include <queue>
#include <vector>

std::vector<int> bfs_distances(int s,
                               const std::vector<std::vector<int>>& adj) {
  int n = adj.size();
  std::vector<int> dist(n, -1);
  std::queue<int> q;

  dist[s] = 0;
  q.push(s);

  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : adj[u]) {
      if (dist[v] == -1) {
        dist[v] = dist[u] + 1;
        q.push(v);
      }
    }
  }
  return dist;
}
```

`dist[v] == -1` doubles as our "not visited" check. The first time we reach `v`, the BFS invariant guarantees it's the shortest path.

▓▓ DFS on a Graph

In trees we wrote `dfs(node)` and recursed on `node->left`, `node->right`. On a graph, we recurse on each unvisited neighbor — and we still need a `visited` set because of cycles.

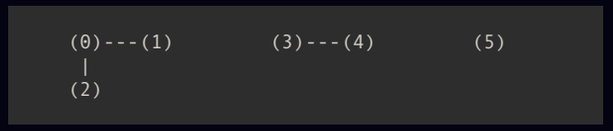
```cpp
#include <vector>

void dfs(int u,
         const std::vector<std::vector<int>>& adj,
         std::vector<bool>& visited) {
  visited[u] = true;
  std::cout << u << " ";

  for (int v : adj[u]) {
    if (!visited[v]) {
      dfs(v, adj, visited);
    }
  }
}
```

The recursion call stack is our stack — no explicit `std::stack` needed (but you could write the iterative version with one).

> Cycle detection (FYI): in an undirected graph, while doing DFS, if you reach an already-visited neighbor that isn't the parent you came from, there's a cycle. One sentence; it's worth remembering.

▓▓▓ Exercise: Count Connected Components

Given an undirected graph, count how many separate "islands" of connected vertices it has.



Expected output: 3.

Starter scaffold:

```cpp
int count_components(const std::vector<std::vector<int>>& adj) {
  int n = adj.size();
  std::vector<bool> visited(n, false);
  int count = 0;

  for (int i = 0; i < n; ++i) {
    if (!visited[i]) {
      // TODO: dfs from i, then increment count
    }
  }
  return count;
}
```

The outer loop is the new pattern: in trees we always started from the root, but a graph may have multiple components.

Try it for 5 minutes.

Exercise Solution

```cpp
#include <vector>

void dfs(int u,
         const std::vector<std::vector<int>>& adj,
         std::vector<bool>& visited) {
  visited[u] = true;
  for (int v : adj[u]) {
    if (!visited[v]) dfs(v, adj, visited);
  }
}

int count_components(const std::vector<std::vector<int>>& adj) {
  int n = adj.size();
  std::vector<bool> visited(n, false);
  int count = 0;

  for (int i = 0; i < n; ++i) {
    if (!visited[i]) {
      dfs(i, adj, visited);
      ++count;
    }
  }
  return count;
}
```

Each call to `dfs` from an unvisited vertex marks one whole component. The `for` loop ensures we don't miss isolated pieces.

▓▓▓ Topological Sort: What Problem Does It Solve?

You're a Baruch student. To take MTH 4300, you must first take MTH 2610 and CSC 1110. To take MTH 2610 you must first take MTH 2003. And so on.

Question: In what order can you take the courses so every prerequisite is satisfied?

That ordering is a topological sort of the prerequisite graph.

Requirements:

- The graph must be directed (prereq → course).
- The graph must have no cycles (otherwise the courses are mutually impossible). A directed graph without cycles is called a DAG (Directed Acyclic Graph).

Other places topo sort shows up: build systems (compile order), spreadsheet recalculation, package managers (`apt`, `npm`).

Topological Sort: Kahn's Algorithm Intuition

Indegree of a vertex = number of incoming edges (= number of unsatisfied prereqs).

Idea: a course you can take right now is one with indegree 0 (no prereqs left).

Algorithm:

1. Compute the indegree of every vertex.
2. Push every indegree-0 vertex into a queue.
3. Pop a vertex `u`, output it. For each neighbor `v` of `u`, decrement `indegree[v]`. If it hits 0, push `v`.
4. Repeat until the queue is empty.

> Notice: this is just BFS with an indegree counter instead of a `visited` set. You already know how to do this.

Trace on this DAG (course prereqs, edges = "must take before"):

Vertices: 2003, 2610, 1110, 4300.

Step	Queue	Indegrees	Output
start	[2003, 1110]	2003:0, 2610:1, 1110:0, 4300:2	
pop 2003	[1110]	2610:0, 4300:2	2003
push 2610	[1110, 2610]		
pop 1110	[2610]	4300:1	2003, 1110
pop 2610	[]	4300:0	2003, 1110, 2610
push 4300	[4300]		
pop 4300	[]		2003, 1110, 2610, 4300

Valid order: take 2003, then 1110, then 2610, then 4300. (2003 and 1110 are interchangeable.)

Topological Sort: Code

```cpp
#include <queue>
#include <vector>

std::vector<int> topo_sort(const std::vector<std::vector<int>>& adj) {
  int n = adj.size();
  std::vector<int> indegree(n, 0);
  for (int u = 0; u < n; ++u) {
    for (int v : adj[u]) ++indegree[v];
  }

  std::queue<int> q;
  for (int i = 0; i < n; ++i) {
    if (indegree[i] == 0) q.push(i);
  }

  std::vector<int> order;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    order.push_back(u);
    for (int v : adj[u]) {
      if (--indegree[v] == 0) q.push(v);
    }
  }

  // If order.size() < n, the graph had a cycle -> no valid topo order.
  return order;
}
```

If the returned order has fewer than `n` vertices, the graph had a cycle and no valid ordering exists.

Lab Preview

Problem. You'll be given a list of courses and prerequisite pairs:

```
4
2003 2610
1110 2610
2610 4300
1110 4300
```

(First line: number of prereq pairs. Each pair `a b` means a is a prereq of b.)

Your program should:

1. Read the input.
2. Build a directed adjacency list (you'll need to map course names to integer ids — `std::unordered_map<std::string,int>` is your friend).
3. Run Kahn's algorithm.
4. Print a valid order, or `"CYCLE DETECTED"` if no valid order exists.