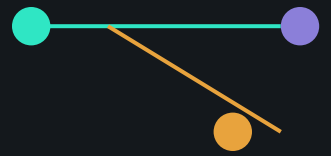


LECTURE 26

Graph Traversal: BFS and DFS

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

LECTURE 26

Agenda

1. Why graph traversal needs a “visited” set that tree traversal never did
2. Breadth-First Search (BFS), using a queue
3. Depth-First Search (DFS), using recursion (the call stack)
4. Tracing both algorithms step by step
5. Traversal on a graph with a cycle, and when BFS wins vs. DFS wins

THE TRAVERSAL CONCEPT

Why Graphs Need a “Visited” Set

Trees never revisit a node — no cycles. A graph can cycle, so traversal needs one extra piece of bookkeeping: a **visited** set, so the algorithm can't loop forever.

Both traversals answer: “visit every vertex reachable from a starting point” — but explore in a different order.

Q

BFS

Level by level, using an explicit queue.

S

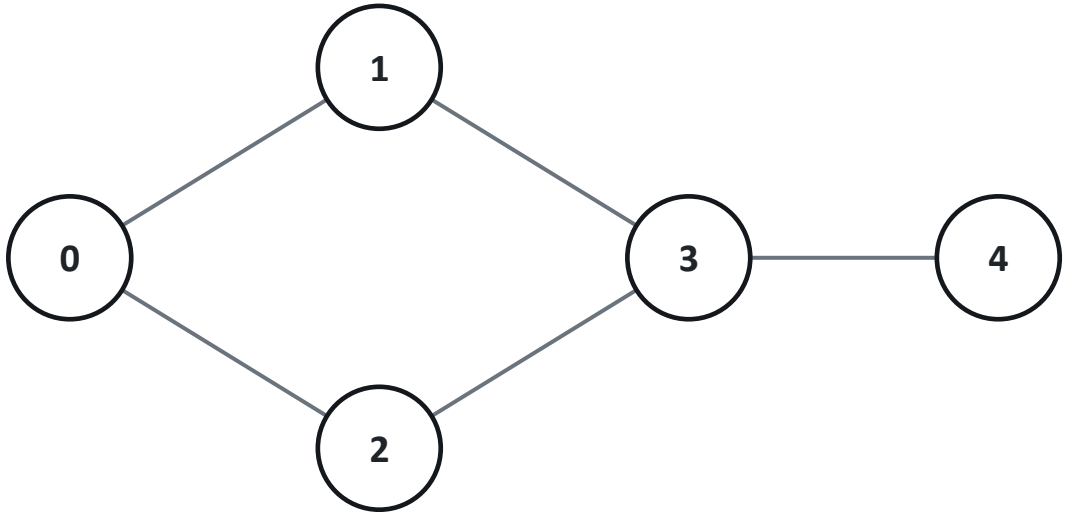
DFS

As deep as possible, then backtrack, using recursion.

!

visited[]

Both add this one array so a cycle can't loop forever.



TRAVERSAL STRATEGY #1

Breadth-First Search (BFS)

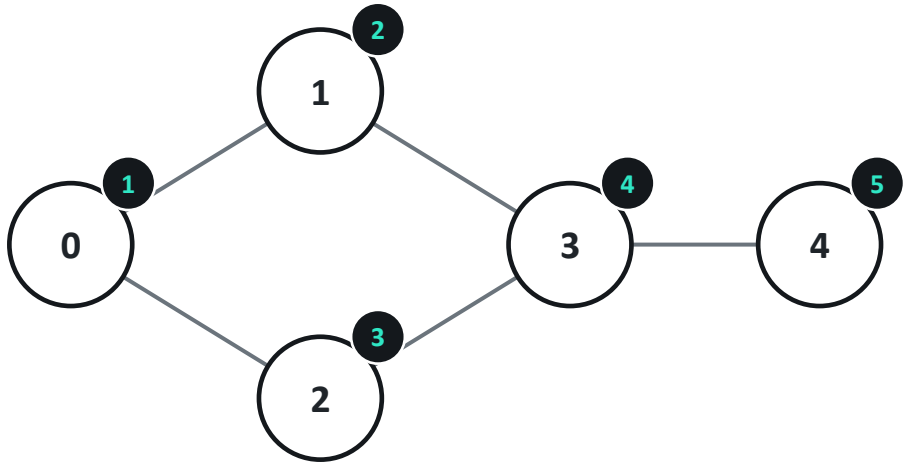
Visit the start, then all direct neighbors, then all of *their* unvisited neighbors — level by level, using a **queue**.

graph_bfs.cpp — bfs()

```
void bfs(int start) const {
  vector<bool> visited(numVertices, false);
  queue<int> toVisit;
  visited[start] = true;
  toVisit.push(start);

  while (!toVisit.empty()) {
    int current = toVisit.front(); toVisit.pop();
    for (int nb : adjList[current]) {
      if (!visited[nb]) {
        visited[nb] = true;
  mark on ENQUEUE
        toVisit.push(nb);
      }
    }
  }
}
```

Output: 0 1 2 3 4



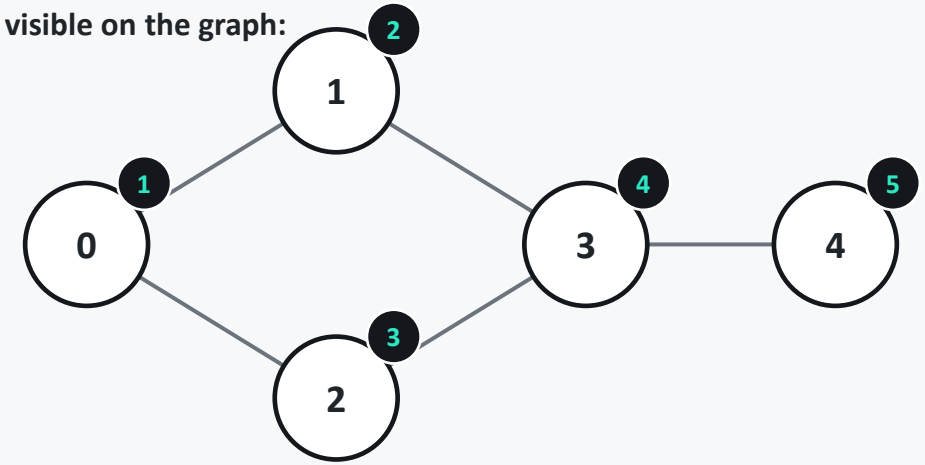
Warning: mark visited[] when ENQUEUING, not dequeuing — else a vertex can be pushed twice before it's processed.

BFS IN MOTION

Tracing BFS Step by Step

Step	Dequeued	Queue before	Newly enqueued	Queue after
1	0	[0]	1, 2	[1, 2]
2	1	[1, 2]	3	[2, 3]
3	2	[2, 3]	(none)	[3]
4	3	[3]	4	[4]
5	4	[4]	(none)	[]

Visit order, made visible on the graph:



Reading top-to-bottom, left-to-right reproduces BFS's order exactly: 0, 1, 2, 3, 4 — everything at distance 1 (1,2) finishes before distance 2 (3), before distance 3 (4).

TRAVERSAL STRATEGY #2

Depth-First Search (DFS)

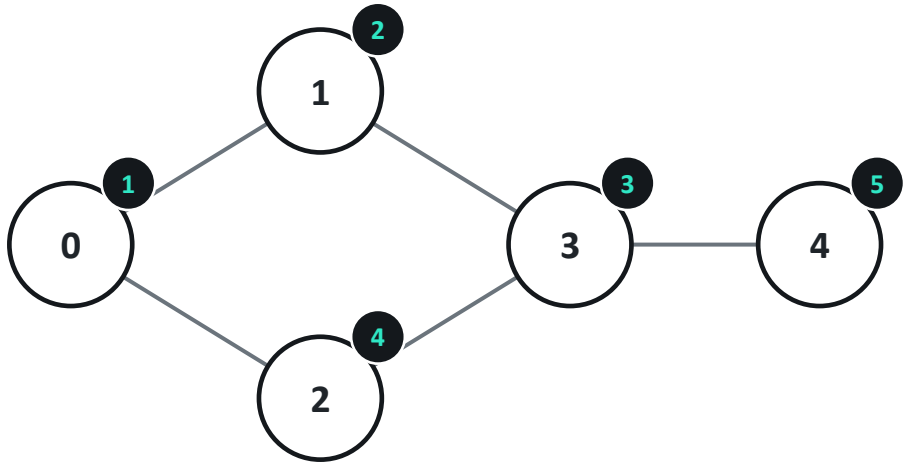
Go as far as possible down one path, then backtrack — using **recursion** (the call stack).

graph_dfs.cpp — dfsHelper()

```
void dfsHelper(int current, vector<bool>& visited) {
  visited[current] = true;
  cout << current << " ";

  for (int nb : adjList[current]) {
    if (!visited[nb]) {
      dfsHelper(nb, visited);
  recurse deeper
    }
  }
}
```

Output: 0 1 3 2 4



Commits to 1, then 3, then 2 (2's other neighbors already visited) — backtracks to 3, then visits 4.

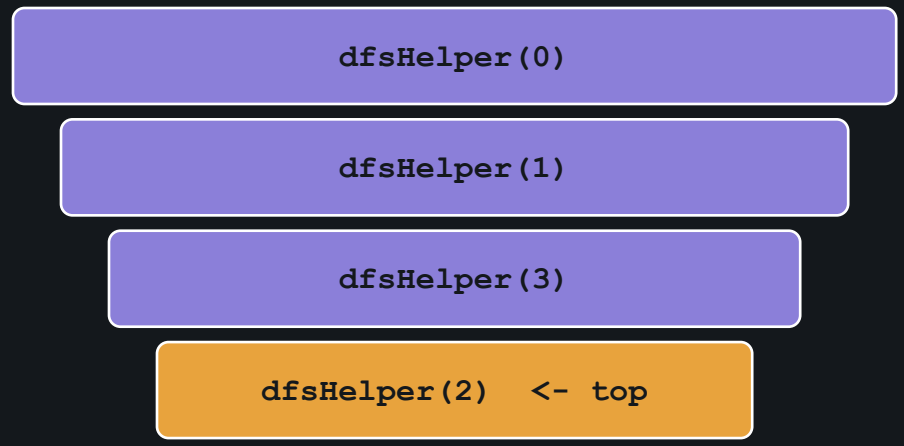
DFS IN MOTION

Tracing DFS: The Call Stack Made Visible

graph_dfs_trace.cpp output

```
visit 0 (depth 0)
  visit 1 (depth 1)
    (skip 0, already visited)
    visit 3 (depth 2)
      (skip 1, already visited)
      visit 2 (depth 3)
        (skip 0, already visited)
        (skip 3, already visited)
      visit 4 (depth 3)
        (skip 3, already visited)
  (skip 2, already visited)
```

The indentation IS the call stack



visit 2 at depth 3 is nested 3 calls deep inside visit 3 inside visit 1 inside visit 0 — exactly the chain of open dfsHelper calls waiting on the stack.

DIRECT COMPARISON

BFS versus DFS

	BFS	DFS
Data structure	Queue (explicit)	Call stack (recursion)
Exploration pattern	Level by level, outward	As deep as possible, then backtrack
Finds shortest path (unweighted)	Yes — fewest edges guaranteed	No guarantee
Memory (worst case)	O(V) — an entire level	O(V) — recursion depth

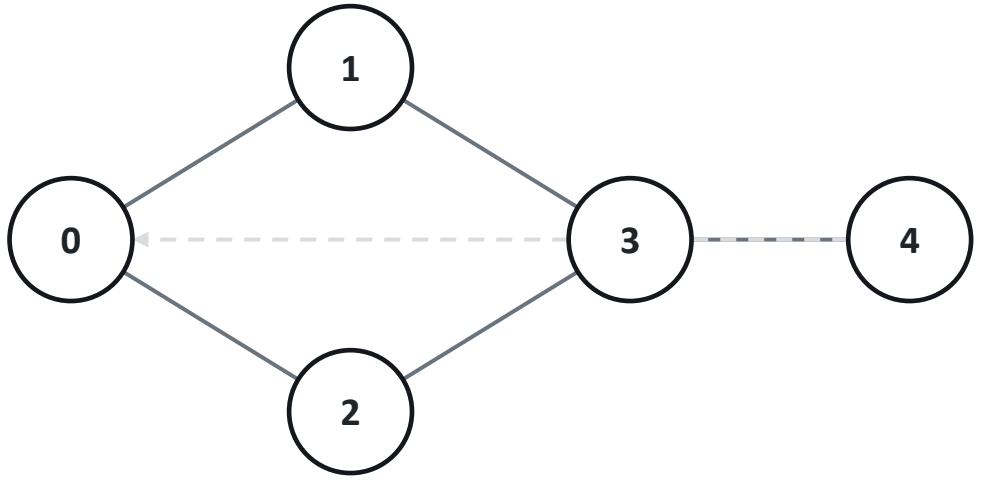
O(V + E)

both BFS and DFS — every vertex visited once, every edge examined once per endpoint

CYCLE SAFETY

Traversal on a Graph With a Cycle

Adding edge 4-0 closes a second loop, 0-2-3-4-0. Without a visited set, this **back edge** would restart the traversal forever.



Dashed edge = back edge. Reaching 4 then following it back to 0 would restart the traversal — the visited[] guard checks and discards it in O(1).

With the cycle edge added:

```
BFS: 0 1 2 4 3
DFS: 0 1 3 2 4
```

Both orders changed (0 now discovers 4 directly) — but every vertex still appears exactly once. No special cycle-handling code was needed: the ordinary if (!visited[neighbor]) guard is entirely sufficient.

APPLICATIONS

When BFS Wins vs. When DFS Wins

Shortest path by hop count	**BFS**
“Is there a cycle?”	**DFS**
Within k hops (social network)	**BFS**
Solving a maze (any path out)	**DFS**
Deep, narrow dependency chain	**DFS**
Web crawler, breadth of coverage	**BFS**

Both run in the same O(V + E) time — the choice is about which question you're answering, and whether your graph is “wide” (favors DFS's low memory) or “deep” (favors care with recursion depth).

KEY TAKEAWAYS

- Graph traversal needs a visited set that tree traversal never required, because graphs can contain cycles.
- BFS explores level by level using an explicit queue — the right choice for shortest-path-by-hop-count queries.
- DFS explores as deep as possible before backtracking, using recursion (the call stack) — the right choice for cycle detection and exhaustive path exploration.
- On a graph with a genuine cycle, neither algorithm needs special-case code — the ordinary if (!visited[neighbor]) guard is exactly what makes traversal correct as well as efficient.
- Both run in O(V + E), which is exactly why the adjacency list representation from Lecture 25 matters.