

Breadth-First Search and Graph Applications

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 4.1 & 4.2

Duration: 4 hours (with breaks)

Learning Objectives

1. Understand the breadth-first search (BFS) algorithm, its use of a FIFO queue, and the role of the `marked[]`, `edgeTo[]`, and `distTo[]` arrays in systematic graph exploration.
 2. Prove that BFS computes shortest paths in unweighted graphs and implement path reconstruction using the `edgeTo[]` parent-link array.
 3. Compare and contrast BFS and DFS in terms of data structures, traversal order, path properties, and appropriate application domains.
 4. Extend graph processing to directed graphs (digraphs), including representations, directed search, reachability, and the concepts of in-degree, out-degree, and strong connectivity.
 5. Master topological sorting of directed acyclic graphs (DAGs) using both Kahn's algorithm (BFS-based with in-degree counting) and DFS-based reverse postorder, and prove correctness of each approach.
 6. Detect bipartite graphs using a two-coloring BFS algorithm, understand the odd-cycle characterization, and apply bipartiteness testing to real-world matching and scheduling problems.
-

Part I: Breadth-First Search

Estimated time: 30 minutes

Motivation and Intuition

In the previous lecture, we studied depth-first search (DFS), an algorithm that explores a graph by plunging as deep as possible along each branch before backtracking. DFS is elegant, naturally recursive, and useful for many structural queries about graphs. However, DFS does not, in general, find shortest paths. If we ask "what is the fewest number of edges on a path from vertex **s** to vertex **t**?", DFS may return a long, winding path even when a short one exists.

Breadth-first search (BFS) addresses this limitation. Rather than diving deep, BFS explores the graph in layers: it first visits all vertices at distance 1 from the source, then all vertices at distance 2, then distance 3, and so on. This layer-by-layer exploration is achieved by replacing DFS's stack (implicit in recursion or explicit) with a FIFO queue. The result is an algorithm that computes shortest paths in unweighted graphs as a natural byproduct of its traversal order.

BFS was first described by Edward F. Moore in 1959 in the context of finding shortest paths through mazes, and independently by C. Y. Lee in 1961 for wire routing in printed circuit boards. It has since become one of the fundamental algorithms in computer science, with applications spanning network analysis, web crawling, social network analysis, garbage collection, and artificial intelligence.

The BFS Algorithm

The core idea of BFS is straightforward:

1. Begin at a source vertex **s**. Mark it as visited and place it in a FIFO queue.
2. While the queue is not empty, remove the front vertex **v**. For each neighbor **w** of **v** that has not yet been visited, mark **w**, record that we reached **w** from **v**, record the distance to **w**, and add **w** to the back of the queue.
3. When the queue empties, every vertex reachable from **s** has been visited.

BFS maintains three arrays:

- **marked[v]** : A boolean array indicating whether vertex **v** has been discovered. Once a vertex is marked, it is never unmarked. This prevents revisiting vertices and ensures termination.
- **edgeTo[v]** : Records the vertex from which **v** was first discovered. This allows us to reconstruct the shortest path from the source **s** to any reachable vertex by following parent links backward.
- **distTo[v]** : Records the shortest-path distance (number of edges) from **s** to **v**. For the source itself, **distTo[s] = 0**. For any vertex **v** discovered from vertex **u**, **distTo[v] = distTo[u] + 1**.

BFS Pseudocode

```
procedure BFS(G, s):
    // Initialize all vertices as unvisited
    for each vertex v in G:
        marked[v] = false
        distTo[v] = INFINITY
        edgeTo[v] = -1

    // Initialize source
    marked[s] = true
    distTo[s] = 0

    // Create a FIFO queue and enqueue the source
    Q = new Queue()
    Q.enqueue(s)

    while Q is not empty:
        v = Q.dequeue()
        for each neighbor w of v in G.adj(v):
            if not marked[w]:
                marked[w] = true
                edgeTo[w] = v
                distTo[w] = distTo[v] + 1
                Q.enqueue(w)
```

A critical detail: we mark vertices when they are **enqueued**, not when they are dequeued. This prevents the same vertex from being added to the queue multiple times. If we delayed marking until dequeue time, a

vertex could be enqueued once by each of several neighbors that are processed before it, leading to redundant work and incorrect distance computations.

Detailed Trace of BFS

Consider the following undirected graph with 7 vertices (0 through 6) and 8 edges:

```
0 --- 1 --- 2
|     |     |
3     4     5
 \       /
  --- 6 ---
```

Adjacency lists (sorted for determinism):

```
0: [1, 3]
1: [0, 2, 4]
2: [1, 5]
3: [0, 6]
4: [1]
5: [2, 6]
6: [3, 5]
```

We run BFS from source vertex `s = 0`. The trace below shows the queue state, the vertex being processed, and the updates to `marked[]`, `edgeTo[]`, and `distTo[]` at each step.

Initialization:

```
marked[] = [F, F, F, F, F, F, F]
distTo[] = [inf, inf, inf, inf, inf, inf, inf]
edgeTo[] = [-1, -1, -1, -1, -1, -1, -1]
```

Mark vertex 0, set $\text{distTo}[0] = 0$

Enqueue 0

Queue: [0]

```
marked[] = [T, F, F, F, F, F, F]
distTo[] = [0, inf, inf, inf, inf, inf, inf]
```

Step 1: Dequeue vertex 0

Queue before dequeue: [0]

Dequeue: $v = 0$

Process neighbors of 0: {1, 3}

Neighbor 1: not marked

Mark 1, $\text{edgeTo}[1] = 0$, $\text{distTo}[1] = 0 + 1 = 1$

Enqueue 1

Neighbor 3: not marked

Mark 3, $\text{edgeTo}[3] = 0$, $\text{distTo}[3] = 0 + 1 = 1$

Enqueue 3

Queue after step: [1, 3]

```
marked[] = [T, T, F, T, F, F, F]
distTo[] = [0, 1, inf, 1, inf, inf, inf]
edgeTo[] = [-1, 0, -1, 0, -1, -1, -1]
```

Step 2: Dequeue vertex 1

```
Queue before dequeue: [1, 3]
Dequeue: v = 1
Process neighbors of 1: {0, 2, 4}

Neighbor 0: already marked -- skip

Neighbor 2: not marked
  Mark 2, edgeTo[2] = 1, distTo[2] = 1 + 1 = 2
  Enqueue 2

Neighbor 4: not marked
  Mark 4, edgeTo[4] = 1, distTo[4] = 1 + 1 = 2
  Enqueue 4

Queue after step: [3, 2, 4]
marked[] = [T, T, T, T, T, F, F]
distTo[] = [0, 1, 2, 1, 2, inf, inf]
edgeTo[] = [-1, 0, 1, 0, 1, -1, -1]
```

Step 3: Dequeue vertex 3

```
Queue before dequeue: [3, 2, 4]
Dequeue: v = 3
Process neighbors of 3: {0, 6}

Neighbor 0: already marked -- skip

Neighbor 6: not marked
  Mark 6, edgeTo[6] = 3, distTo[6] = 1 + 1 = 2
  Enqueue 6

Queue after step: [2, 4, 6]
marked[] = [T, T, T, T, T, F, T]
distTo[] = [0, 1, 2, 1, 2, inf, 2]
edgeTo[] = [-1, 0, 1, 0, 1, -1, 3]
```

Step 4: Dequeue vertex 2

```
Queue before dequeue: [2, 4, 6]
Dequeue: v = 2
Process neighbors of 2: {1, 5}

  Neighbor 1: already marked -- skip

  Neighbor 5: not marked
    Mark 5, edgeTo[5] = 2, distTo[5] = 2 + 1 = 3
    Enqueue 5

Queue after step: [4, 6, 5]
marked[] = [T, T, T, T, T, T, T]
distTo[] = [0, 1, 2, 1, 2, 3, 2]
edgeTo[] = [-1, 0, 1, 0, 1, 2, 3]
```

Step 5: Dequeue vertex 4

```
Queue before dequeue: [4, 6, 5]
Dequeue: v = 4
Process neighbors of 4: {1}

  Neighbor 1: already marked -- skip

Queue after step: [6, 5]
```

Step 6: Dequeue vertex 6

```
Queue before dequeue: [6, 5]
Dequeue: v = 6
Process neighbors of 6: {3, 5}

  Neighbor 3: already marked -- skip
  Neighbor 5: already marked -- skip

Queue after step: [5]
```

Step 7: Dequeue vertex 5

```
Queue before dequeue: [5]
Dequeue: v = 5
Process neighbors of 5: {2, 6}

Neighbor 2: already marked -- skip
Neighbor 6: already marked -- skip

Queue after step: []
```

Queue is empty. BFS complete.

Final state:

Vertex:	0	1	2	3	4	5	6
distTo:	0	1	2	1	2	3	2
edgeTo:	-1	0	1	0	1	2	3

Observe the layer structure: vertex 0 is at distance 0 (layer 0), vertices 1 and 3 are at distance 1 (layer 1), vertices 2, 4, and 6 are at distance 2 (layer 2), and vertex 5 is at distance 3 (layer 3). BFS processed vertices in exactly this order of increasing distance.

BFS Spanning Tree

The `edgeTo[]` array defines a **BFS spanning tree** (or BFS spanning forest, if the graph is disconnected). For each vertex `v` other than the source, the edge `(edgeTo[v], v)` is a tree edge. In our example, the tree edges are:

```
0 -- 1  (edgeTo[1] = 0)
0 -- 3  (edgeTo[3] = 0)
1 -- 2  (edgeTo[2] = 1)
1 -- 4  (edgeTo[4] = 1)
3 -- 6  (edgeTo[6] = 3)
2 -- 5  (edgeTo[5] = 2)
```

This tree has the special property that the path from the source to any vertex in the tree is a shortest path in the original graph. This is the fundamental property that distinguishes BFS trees from DFS trees.

Running Time Analysis

PROPOSITION A. *BFS runs in $O(V + E)$ time on a graph with V vertices and E edges.*

Proof. Each vertex is enqueued at most once (since we mark it before enqueueing, and we only enqueue unmarked vertices). Therefore, each vertex is dequeued at most once. When a vertex v is dequeued, we examine each of its neighbors, which takes time proportional to the degree of v . Summing over all vertices, the total work examining neighbors is proportional to the sum of all degrees, which equals $2E$ for an undirected graph (or E for a directed graph). The initialization loop takes $O(V)$ time. Therefore, the total running time is $O(V + E)$.

This is the same asymptotic complexity as DFS. Both algorithms are optimal in the sense that any algorithm that explores an entire graph must examine every vertex and every edge at least once.

Part II: Shortest Paths in Unweighted Graphs

Estimated time: 25 minutes

BFS and Shortest Paths

One of the most important properties of BFS is that it naturally computes shortest paths in unweighted graphs (or equivalently, in graphs where all edges have unit weight). This is not an accident; it is a direct consequence of the FIFO discipline of the queue.

DEFINITION. The **shortest-path distance** from vertex s to vertex v in an unweighted graph, denoted $d(s, v)$, is the minimum number of edges on any path from s to v . If no path exists, $d(s, v) = \text{infinity}$.

PROPOSITION B (Correctness of BFS Shortest Paths). *After running BFS from source s , for every vertex v reachable from s , $\text{distTo}[v] = d(s, v)$, and the path from s to v obtained by following $\text{edgeTo}[]$ links is a shortest path.*

Proof. We prove by strong induction on the distance $k = d(s, v)$.

Base case ($k = 0$): The only vertex at distance 0 from s is s itself. BFS sets $\text{distTo}[s] = 0$, which is correct.

Inductive step: Assume that for all vertices u with $d(s, u) < k$, BFS correctly computes $\text{distTo}[u] = d(s, u)$. We must show that for any vertex v with $d(s, v) = k$, BFS computes $\text{distTo}[v] = k$.

Since $d(s, v) = k$, there exists a shortest path $s = w_0, w_1, \dots, w_{k-1}, w_k = v$. By the definition of shortest-path distance, $d(s, w_{k-1}) = k - 1$. By the inductive hypothesis, $\text{distTo}[w_{k-1}] = k - 1$, and w_{k-1} was enqueued during the BFS.

When w_{k-1} is dequeued, BFS examines all of its neighbors, including v . At this point, either:

- v has already been marked, meaning $\text{distTo}[v]$ has already been set. We claim $\text{distTo}[v] \leq k$. Since $\text{distTo}[v]$ was set when v was first discovered via some vertex u with $\text{distTo}[u] = \text{distTo}[v] - 1$, and $\text{distTo}[u] \geq d(s, u)$ (distances can only be overestimates at the time of assignment, but by the inductive hypothesis they are exact for vertices closer than v), we have $\text{distTo}[v] \leq k$. Also, $\text{distTo}[v] \geq d(s, v) = k$ because BFS never assigns a distance smaller than the true shortest-path distance (it only assigns $\text{distTo}[v] = \text{distTo}[u] + 1$ for some vertex u , and $\text{distTo}[u] \geq d(s, u)$ by induction). Therefore, $\text{distTo}[v] = k$.
- v has not been marked. Then BFS sets $\text{distTo}[v] = \text{distTo}[w_{k-1}] + 1 = (k - 1) + 1 = k$. Correct.

In both cases, $\text{distTo}[v] = k = d(s, v)$. This completes the induction.

Furthermore, the `edgeTo[]` link for v points to some vertex u with $\text{distTo}[u] = \text{distTo}[v] - 1 = k - 1$. By induction, $\text{distTo}[u] = d(s, u)$, so u is at distance $k - 1$ from s . The edge (u, v) then extends a shortest path to u by one edge, yielding a shortest path to v of length k .

Path Reconstruction

Given the `edgeTo[]` array, we can reconstruct the shortest path from s to any vertex t by following parent links from t back to s :

```

procedure ShortestPath(s, t, edgeTo[]):
    if not marked[t]:
        return "No path exists"

    path = new Stack()
    v = t
    while v != s:
        path.push(v)
        v = edgeTo[v]
    path.push(s)

    return path    // pop elements for s -> ... -> t order

```

The path is reconstructed in reverse (from **t** to **s**), so we use a stack to reverse it. The time to reconstruct a path is $O(\text{length of the path})$, which is at most $O(V)$.

Multi-Source BFS

A useful variant of BFS starts from multiple source vertices simultaneously. Instead of enqueueing a single source, we enqueue all sources at the start:

```

procedure MultisourceBFS(G, sources):
    for each vertex v in G:
        marked[v] = false
        distTo[v] = INFINITY

    Q = new Queue()
    for each vertex s in sources:
        marked[s] = true
        distTo[s] = 0
        Q.enqueue(s)

    while Q is not empty:
        v = Q.dequeue()
        for each neighbor w of v in G.adj(v):
            if not marked[w]:
                marked[w] = true
                distTo[w] = distTo[v] + 1
                Q.enqueue(w)

```

After multi-source BFS, `distTo[v]` gives the shortest distance from `v` to the **nearest** source vertex. This is equivalent to adding a virtual super-source vertex connected to all source vertices and running standard BFS from the super-source.

Applications of multi-source BFS include:

- Finding the nearest facility (hospital, fire station) to every location in a road network.
- Computing Voronoi diagrams on grid graphs.
- Solving certain game-state exploration problems where multiple initial states are possible.

The Kevin Bacon Game (Six Degrees of Separation)

A celebrated application of BFS is the "Six Degrees of Kevin Bacon" game. The idea is to construct a graph where:

- Each actor is a vertex.
- Two actors are connected by an edge if they appeared in the same movie.

The "Bacon number" of an actor is the shortest-path distance from that actor to Kevin Bacon in this graph. For example, if actor A appeared in a movie with actor B, and actor B appeared in a movie with Kevin Bacon, then actor A has a Bacon number of 2.

BFS from the Kevin Bacon vertex computes the Bacon number of every actor in the database in $O(V + E)$ time. The Internet Movie Database contains hundreds of thousands of actors, and the vast majority have a Bacon number of 6 or less, illustrating the "small-world" phenomenon in social networks.

More precisely, the graph is modeled as a **bipartite** graph with actor vertices and movie vertices. An actor vertex is connected to a movie vertex if that actor appeared in that movie. The Bacon number is then half the BFS distance in this bipartite graph (since each hop alternates between an actor and a movie).

This application demonstrates the power of BFS: a simple, linear-time algorithm answers a question about shortest paths in a graph with millions of edges.

Part III: BFS vs DFS Comparison

Estimated time: 15 minutes

Structural Comparison

BFS and DFS are the two fundamental graph traversal algorithms. They share the same high-level structure -- systematically visit all vertices reachable from a source by examining neighbors of visited vertices -- but differ in the order in which vertices are processed.

Property	BFS	DFS
Data structure	FIFO Queue	LIFO Stack (or recursion)
Traversal order	Layer by layer (by distance)	Plunge deep, then backtrack
Tree shape	Short, wide (bushy)	Long, narrow (stringy)
Shortest paths	Yes (unweighted)	No (in general)
Memory usage	$O(\text{width of graph})$	$O(\text{depth of graph})$
Time complexity	$O(V + E)$	$O(V + E)$
Edge classification	Tree edges, cross edges	Tree edges, back edges (undirected)
Cycle detection	Yes (finds shortest cycle through source)	Yes (finds some cycle)
Connected components	Yes	Yes
Bipartite detection	Yes (natural two-coloring)	Yes (with coloring)
Topological sort	Yes (Kahn's algorithm)	Yes (reverse postorder)

When to Use BFS

BFS is the algorithm of choice when:

- **Shortest paths** in unweighted graphs are needed. This is the defining advantage of BFS.
- **Level-order traversal** is required, for example, to process all vertices at a given distance before those farther away.
- **Minimum-depth solutions** in state-space search, such as the 15-puzzle or Rubik's cube, where we want the fewest moves to reach a goal.
- **Web crawling** with bounded depth: visit all pages within k links of a starting page.
- **Network broadcasting**: a message propagates from a source to all nodes, and BFS models the propagation order.

When to Use DFS

DFS is preferred when:

- **Path existence** rather than shortest paths is the question. DFS uses less memory on many graph structures.
- **Topological sorting** via reverse postorder. While BFS can also perform topological sort (Kahn's algorithm), DFS-based topological sort is often simpler to implement.
- **Cycle detection** in directed graphs (via back edges).
- **Strongly connected components** (Tarjan's algorithm, Kosaraju's algorithm), which rely on DFS.
- **Articulation points and bridges**, which are detected using DFS tree properties.
- **Maze generation**: DFS produces long, winding passages; BFS produces short, branching corridors.

Memory Considerations

In the worst case, both BFS and DFS use $O(V)$ memory. However, the typical memory usage differs:

- **BFS** stores all vertices in the current "frontier" (the boundary between visited and unvisited vertices). In a graph with high branching factor b and depth d , the frontier can be as large as $O(b^d)$. For example, in a binary tree of depth 20, the last level has over a million nodes, all of which would be in the BFS queue simultaneously.
- **DFS** stores only the vertices along the current path from the source. In the same binary tree, DFS would only store 20 vertices on the stack. However, in a graph that is a long path (depth = V), DFS uses $O(V)$ stack space, which matches BFS.

For very large graphs (e.g., web-scale graphs with billions of vertices), the memory footprint of BFS can be prohibitive. In such cases, techniques like iterative deepening DFS (IDDFS) combine the shortest-path guarantee of BFS with the memory efficiency of DFS: IDDFS runs DFS with depth limits 0, 1, 2, ..., until the goal is found. It uses $O(d)$ memory (like DFS) and finds shortest paths (like BFS), at the cost of re-exploring vertices at each depth limit.

Edge Classification

BFS and DFS classify the edges of a graph differently:

- **DFS** on an undirected graph produces **tree edges** (edges in the DFS tree) and **back edges** (edges from a descendant to an ancestor in the DFS tree). There are no cross edges in undirected DFS.

- **BFS** on an undirected graph produces **tree edges** (edges in the BFS tree) and **cross edges** (edges between vertices in the same layer or adjacent layers that are not tree edges). There are no back edges spanning more than one layer in BFS, because if a back edge connected layers k and $k + j$ with $j \geq 2$, the vertex at layer $k + j$ would have been discovered at layer $k + 1$ instead.

This edge classification has practical implications. For example, in DFS, a back edge indicates a cycle. In BFS, cross edges within the same layer indicate odd cycles (relevant to bipartiteness testing).

Part IV: Directed Graphs (Digraphs)

Estimated time: 30 minutes

Definitions and Terminology

DEFINITION. A **directed graph** (or **digraph**) is a set of vertices V and a set of ordered pairs of vertices E , called **directed edges** or **arcs**. A directed edge (u, v) goes **from** u **to** v ; we say u is the **tail** and v is the **head** of the edge.

In contrast to undirected graphs, where the edge $\{u, v\}$ is the same as $\{v, u\}$, in a digraph the edge (u, v) is distinct from (v, u) . The edge (u, v) means there is a connection from u to v but not necessarily from v to u .

DEFINITION. The **out-degree** of a vertex v is the number of edges leaving v : $\text{out-deg}(v) = |\{(v, w) : (v, w) \in E\}|$. The **in-degree** of v is the number of edges entering v : $\text{in-deg}(v) = |\{(u, v) : (u, v) \in E\}|$.

Note that in a digraph, the sum of all out-degrees equals the sum of all in-degrees, and both equal $|E|$:

$$\text{Sum of out-deg}(v) \text{ for all } v = |E| = \text{Sum of in-deg}(v) \text{ for all } v$$

DEFINITION. A **directed path** from u to v is a sequence of vertices $u = v_0, v_1, \dots, v_k = v$ such that (v_i, v_{i+1}) is a directed edge for each i . The **length** of the path is k .

DEFINITION. A **directed cycle** is a directed path from a vertex back to itself: v_0, v_1, \dots, v_k where $v_0 = v_k$ and $k \geq 1$.

DEFINITION. A **directed acyclic graph** (DAG) is a digraph with no directed cycles.

DEFINITION. Vertex v is **reachable** from vertex u if there exists a directed path from u to v .

DEFINITION. A digraph is **strongly connected** if for every pair of vertices u and v , there is a directed path from u to v and a directed path from v to u .

Digraph Representations

Digraphs use the same representations as undirected graphs, with the distinction that edges are stored in only one direction.

Adjacency List Representation. For a digraph, the adjacency list $\text{adj}[v]$ contains only the vertices w such that (v, w) is a directed edge. In an undirected graph, the edge $\{u, v\}$ appears in both $\text{adj}[u]$ and $\text{adj}[v]$, but in a digraph, the edge (u, v) appears only in $\text{adj}[u]$.

```
Digraph with 5 vertices and 7 edges:
```

```
0 -> 1      1 -> 2      2 -> 3
0 -> 4      3 -> 0      4 -> 2
4 -> 3
```

```
Adjacency lists:
```

```
0: [1, 4]
1: [2]
2: [3]
3: [0]
4: [2, 3]
```

```
In-degrees:  0:1  1:1  2:2  3:2  4:1
```

```
Out-degrees: 0:2  1:1  2:1  3:1  4:2
```

Adjacency Matrix Representation. For a digraph, the adjacency matrix A has $A[u][v] = 1$ if (u, v) is a directed edge, and $A[u][v] = 0$ otherwise. Unlike undirected graphs, the adjacency matrix of a digraph is not necessarily symmetric.

Space and time complexities are the same as for undirected graphs: adjacency lists use $O(V + E)$ space, and adjacency matrices use $O(V^2)$ space. Adjacency lists are preferred for sparse graphs ($E \ll V^2$), which is the common case in practice.

Digraph Pseudocode

```
class Digraph:
    V          // number of vertices
    adj[]      // adjacency lists (one per vertex)

    procedure Digraph(V):
        this.V = V
        for i = 0 to V - 1:
            adj[i] = new List()

    procedure addEdge(v, w):
        adj[v].add(w)      // directed: only v -> w

    procedure adjacentTo(v):
        return adj[v]

    procedure reverse():
        // Return a new digraph with all edges reversed
        R = new Digraph(V)
        for v = 0 to V - 1:
            for each w in adj[v]:
                R.addEdge(w, v)
        return R

    procedure inDegree(v):
        count = 0
        for u = 0 to V - 1:
            for each w in adj[u]:
                if w == v:
                    count = count + 1
        return count
```

The `reverse()` operation is important for several digraph algorithms, including Kosaraju's algorithm for strongly connected components. It creates a new digraph where every edge `(v, w)` in the original becomes `(w, v)` in the reverse.

For efficient in-degree computation, we can maintain a separate `inDeg[]` array that is updated each time an edge is added:

```
procedure addEdge(v, w):  
    adj[v].add(w)  
    inDeg[w] = inDeg[w] + 1
```

Directed DFS and Directed BFS

Both DFS and BFS extend naturally to directed graphs. The only difference is that when processing a vertex `v`, we examine only the vertices in `adj[v]` (vertices reachable by a directed edge from `v`), rather than all neighbors.

Directed BFS from a source `s` discovers all vertices reachable from `s` via directed paths. The `distTo[v]` values give shortest directed-path distances.

```

procedure DirectedBFS(G, s):
    for each vertex v in G:
        marked[v] = false
        distTo[v] = INFINITY

    marked[s] = true
    distTo[s] = 0
    Q = new Queue()
    Q.enqueue(s)

    while Q is not empty:
        v = Q.dequeue()
        for each w in G.adj(v):      // only outgoing edges
            if not marked[w]:
                marked[w] = true
                distTo[w] = distTo[v] + 1
                edgeTo[w] = v
                Q.enqueue(w)

```

Directed DFS from a source **s** discovers all vertices reachable from **s**. In a digraph, the DFS classifies edges into four categories:

- **Tree edges**: edges in the DFS tree.
- **Back edges**: edges from a descendant to an ancestor (indicate directed cycles).
- **Forward edges**: edges from an ancestor to a descendant (but not tree edges).
- **Cross edges**: all other edges (between vertices with no ancestor-descendant relationship).

This richer edge classification is a key tool for analyzing digraphs.

Reachability

DEFINITION. The **reachability problem** asks: given a digraph **G** and two vertices **s** and **t**, is **t** reachable from **s**?

Single-source reachability is solved by running DFS or BFS from **s** and checking whether **t** was visited. This takes $O(V + E)$ time.

Single-source reachability (all targets): Run DFS or BFS from s . The set of marked vertices is exactly the set of vertices reachable from s .

Multi-source reachability: Given a set of source vertices S , find all vertices reachable from any vertex in S . This is solved by multi-source BFS (or DFS started from each source in S that has not yet been visited), taking $O(V + E)$ time.

Multi-source reachability is fundamental to garbage collection: the "roots" (local variables, global references) are the source set, and the reachable objects are the live objects that should not be collected.

Strong Connectivity

Two vertices u and v are **strongly connected** if there exists a directed path from u to v and a directed path from v to u . Strong connectivity is an equivalence relation, partitioning the vertices into **strongly connected components** (SCCs).

Computing SCCs is a classic problem solved by Kosaraju's algorithm or Tarjan's algorithm, both running in $O(V + E)$ time. While the full treatment of SCCs is beyond the scope of this lecture, we note that the first step of Kosaraju's algorithm is to compute a reverse postorder of the reverse digraph using DFS -- a concept closely related to topological sorting, which we discuss next.

Part V: Topological Sort

Estimated time: 30 minutes

Definition and Motivation

DEFINITION. A **topological order** (or **topological sort**) of a directed acyclic graph (DAG) is a linear ordering of its vertices such that for every directed edge (u, v) , vertex u appears before vertex v in the ordering.

Equivalently, a topological order is a permutation v_1, v_2, \dots, v_n of the vertices such that if there is a directed path from v_i to v_j , then $i < j$.

PROPOSITION C. A digraph has a topological order if and only if it is a DAG.

Proof. (\Rightarrow) If a digraph has a directed cycle $v_0, v_1, \dots, v_k = v_0$, then in any linear ordering, v_0 must come before v_1 , v_1 before v_2 , ..., v_{k-1} before $v_k = v_0$. But this means v_0 must come before v_0 , a contradiction. Therefore, a digraph with a topological order cannot have a directed cycle; it must be a DAG.

(\Leftarrow) We prove this constructively by showing that every DAG has at least one vertex with in-degree 0 (a "source" vertex). Remove this vertex and all its outgoing edges to obtain a smaller DAG. Repeat until all vertices are removed. The order of removal is a topological order.

To see that every DAG has a vertex with in-degree 0: suppose not. Then every vertex has at least one incoming edge. Starting from any vertex v_0 , follow an incoming edge to some vertex v_1 , then from v_1 follow an incoming edge to v_2 , and so on. Since the graph is finite, this sequence must eventually revisit a vertex, creating a directed cycle. But this contradicts the assumption that the graph is a DAG.

Topological sorting has numerous practical applications:

- **Course scheduling:** If course A is a prerequisite for course B, the edge (A, B) indicates that A must be taken before B. A topological order gives a valid semester-by-semester plan.
- **Build systems** (Make, Gradle, Bazel): If file A depends on file B, B must be compiled before A. Topological order determines the compilation sequence.
- **Task scheduling:** In a project with dependent tasks, topological order gives an execution sequence that respects all dependencies.
- **Spreadsheet evaluation:** If cell A references cell B, B must be evaluated before A. Topological order on the dependency graph gives the evaluation order.
- **Package managers** (apt, npm): Package dependencies form a DAG, and packages must be installed in topological order.

Kahn's Algorithm (BFS-Based Topological Sort)

Kahn's algorithm, published by Arthur B. Kahn in 1962, uses the observation that a DAG always has at least one vertex with in-degree 0. The algorithm repeatedly removes such vertices:

```

procedure TopologicalSortKahn(G):
    // Compute in-degrees
    for each vertex v in G:
        inDegree[v] = 0
    for each vertex v in G:
        for each w in G.adj(v):
            inDegree[w] = inDegree[w] + 1

    // Enqueue all vertices with in-degree 0
    Q = new Queue()
    for each vertex v in G:
        if inDegree[v] == 0:
            Q.enqueue(v)

    order = new List()
    count = 0

    while Q is not empty:
        v = Q.dequeue()
        order.append(v)
        count = count + 1

        for each w in G.adj(v):
            inDegree[w] = inDegree[w] - 1
            if inDegree[w] == 0:
                Q.enqueue(w)

    if count != G.V:
        return "Graph has a cycle -- no topological order"

    return order

```

The algorithm maintains a queue of "available" vertices (those with no remaining dependencies). When a vertex is removed from the queue and added to the output, its outgoing edges are effectively deleted by decrementing the in-degrees of its neighbors. Any neighbor whose in-degree drops to 0 becomes available and is enqueued.

If the algorithm terminates with `count < V`, some vertices were never enqueued because their in-degrees never reached 0. This means the graph contains a directed cycle.

Detailed Trace of Kahn's Algorithm

Consider the following DAG with 7 vertices and 8 edges, representing course prerequisites:

```
0 -> 1    0 -> 2    1 -> 3
2 -> 3    3 -> 4    2 -> 5
5 -> 6    4 -> 6
```

Adjacency lists:

```
0: [1, 2]
1: [3]
2: [3, 5]
3: [4]
4: [6]
5: [6]
6: []
```

Step 0: Compute in-degrees

```
Vertex:    0  1  2  3  4  5  6
inDegree:  0  1  1  2  1  1  2
```

Vertices with in-degree 0: {0}

```
Queue: [0]
Order: []
```

Step 1: Dequeue vertex 0

```
Dequeue: v = 0
Add 0 to order.
Process outgoing edges of 0:
  Edge 0 -> 1: inDegree[1] = 1 - 1 = 0 => Enqueue 1
  Edge 0 -> 2: inDegree[2] = 1 - 1 = 0 => Enqueue 2

Queue: [1, 2]
Order: [0]
inDegree: - 0 0 2 1 1 2
```

Step 2: Dequeue vertex 1

```
Dequeue: v = 1
Add 1 to order.
Process outgoing edges of 1:
  Edge 1 -> 3: inDegree[3] = 2 - 1 = 1 (not 0, do not enqueue)

Queue: [2]
Order: [0, 1]
inDegree: - - 0 1 1 1 2
```

Step 3: Dequeue vertex 2

```
Dequeue: v = 2
Add 2 to order.
Process outgoing edges of 2:
  Edge 2 -> 3: inDegree[3] = 1 - 1 = 0 => Enqueue 3
  Edge 2 -> 5: inDegree[5] = 1 - 1 = 0 => Enqueue 5

Queue: [3, 5]
Order: [0, 1, 2]
inDegree: - - - 0 1 0 2
```

Step 4: Dequeue vertex 3

```
Dequeue: v = 3
Add 3 to order.
Process outgoing edges of 3:
  Edge 3 -> 4: inDegree[4] = 1 - 1 = 0  => Enqueue 4

Queue: [5, 4]
Order: [0, 1, 2, 3]
inDegree: - - - - 0 0 2
```

Step 5: Dequeue vertex 5

```
Dequeue: v = 5
Add 5 to order.
Process outgoing edges of 5:
  Edge 5 -> 6: inDegree[6] = 2 - 1 = 1  (not 0, do not enqueue)

Queue: [4]
Order: [0, 1, 2, 3, 5]
inDegree: - - - - 0 - 1
```

Step 6: Dequeue vertex 4

```
Dequeue: v = 4
Add 4 to order.
Process outgoing edges of 4:
  Edge 4 -> 6: inDegree[6] = 1 - 1 = 0  => Enqueue 6

Queue: [6]
Order: [0, 1, 2, 3, 5, 4]
inDegree: - - - - - 0
```

Step 7: Dequeue vertex 6

```
Dequeue: v = 6
Add 6 to order.
No outgoing edges.

Queue: []
Order: [0, 1, 2, 3, 5, 4, 6]
```

Queue is empty. count = 7 = V. Success.

Topological order: **0, 1, 2, 3, 5, 4, 6**

We can verify: every edge goes from an earlier vertex to a later vertex in the ordering:

- 0 -> 1: position 0 < position 1 (valid)
- 0 -> 2: position 0 < position 2 (valid)
- 1 -> 3: position 1 < position 3 (valid)
- 2 -> 3: position 2 < position 3 (valid)
- 2 -> 5: position 2 < position 4 (valid)
- 3 -> 4: position 3 < position 5 (valid)
- 4 -> 6: position 5 < position 6 (valid)
- 5 -> 6: position 4 < position 6 (valid)

Note that the topological order is not unique. If we had used a different tie-breaking rule (e.g., processing 2 before 1 when both have in-degree 0), we would have obtained a different valid topological order.

DFS-Based Topological Sort (Reverse Postorder)

An alternative approach to topological sorting uses DFS. The key insight is that in a DAG, if we run DFS and record the order in which vertices finish (are completely processed), then the reverse of this order is a valid topological order.

DEFINITION. The **postorder** of a DFS is the order in which vertices complete their recursive calls (i.e., all descendants have been fully explored). The **reverse postorder** is the reverse of the postorder.

```

procedure TopologicalSortDFS(G):
    for each vertex v in G:
        marked[v] = false
    reversePost = new Stack()

    for each vertex v in G:
        if not marked[v]:
            DFS(G, v, reversePost)

    return reversePost    // pop elements for topological order

procedure DFS(G, v, reversePost):
    marked[v] = true
    for each w in G.adj(v):
        if not marked[w]:
            DFS(G, w, reversePost)
    reversePost.push(v)    // push after all descendants are done

```

PROPOSITION D (Correctness of DFS-Based Topological Sort). *If G is a DAG, then the reverse postorder of any DFS on G is a topological order.*

Proof. Consider any directed edge (u, v) in G . We must show that u appears before v in the reverse postorder, which is equivalent to showing that u finishes after v in the DFS postorder.

When DFS explores vertex u and encounters the edge (u, v) , there are three cases:

1. **v has not been visited.** Then DFS recurses on v , and v finishes before u (since v 's recursive call returns before u 's). So u finishes after v .
2. **v has been visited and has finished.** Then v is already in the postorder, and u has not yet finished. So u finishes after v .
3. **v has been visited but has not finished.** This means v is an ancestor of u in the DFS tree (since v started but hasn't finished, and u is currently being explored within v 's subtree). But the edge (u, v) then goes from a descendant to an ancestor -- this is a back edge, which implies a directed cycle $v \rightarrow \dots \rightarrow u \rightarrow v$. This contradicts the assumption that G is a DAG. Therefore, this case cannot occur.

In both valid cases, u finishes after v , so u appears before v in the reverse postorder. Since this holds for every directed edge, the reverse postorder is a topological order.

Comparison of Topological Sort Algorithms

Property	Kahn's Algorithm (BFS)	DFS Reverse Postorder
Approach	Iteratively remove sources	Record finish times, reverse
Data structure	Queue + in-degree array	Recursion stack + postorder stack
Cycle detection	Yes (count < V)	Requires separate back-edge check
Time complexity	$O(V + E)$	$O(V + E)$
Space complexity	$O(V)$	$O(V)$
Lexicographically smallest	Use min-heap instead of queue	Not straightforward
Parallelism hints	Vertices at same "level" can run in parallel	Less obvious

Kahn's algorithm has a practical advantage when detecting cycles: if the count of processed vertices is less than V , the graph has a cycle. With DFS, cycle detection requires checking for back edges during the traversal.

Kahn's algorithm also naturally identifies independent tasks: all vertices that are simultaneously in the queue have no dependencies among them and can be executed in parallel. This makes Kahn's algorithm useful for task scheduling with parallelism.

Applications of Topological Sort

Course scheduling. A university offers courses with prerequisites. Model each course as a vertex and each prerequisite as a directed edge. A topological order gives a valid sequence in which a student can take all courses.

Build systems. In a build system like Make, source files have dependencies. A topological sort of the dependency graph determines the order in which files should be compiled so that each file is compiled only after all its dependencies.

Spreadsheet evaluation. Cells in a spreadsheet may reference other cells. A topological sort of the dependency graph determines the evaluation order. If a cycle is detected, the spreadsheet reports a "circular reference" error.

Critical path analysis. Given a DAG where each vertex has a weight (task duration), the **critical path** is the longest path from a source to a sink, representing the minimum project completion time. Topological sort enables computation of earliest start times and latest start times in $O(V + E)$ time.

Serialization. When serializing a set of objects with dependencies (e.g., for network transmission or file storage), topological order ensures that each object is serialized after all objects it depends on.

Part VI: Bipartite Graphs

Estimated time: 20 minutes

Definition and Characterization

DEFINITION. An undirected graph $G = (V, E)$ is **bipartite** if the vertex set V can be partitioned into two disjoint sets A and B such that every edge connects a vertex in A to a vertex in B . Equivalently, G is bipartite if it is **2-colorable**: each vertex can be assigned one of two colors such that no two adjacent vertices have the same color.

Bipartite graphs arise naturally in many contexts:

- **Job assignments:** Workers are one set, jobs are the other; edges indicate which workers can do which jobs.
- **Student-course registration:** Students are one set, courses are the other; edges indicate enrollment.
- **Network routing:** In some network topologies, nodes alternate between two types.
- **Stable matching:** In the Gale-Shapley algorithm, men and women form the two sides of a bipartite graph.

PROPOSITION E (Characterization of Bipartite Graphs). *An undirected graph is bipartite if and only if it contains no odd-length cycle.*

Proof. (\Rightarrow) Suppose G is bipartite with parts A and B . Consider any cycle $v_0, v_1, \dots, v_k = v_0$. As we traverse the cycle, each edge alternates between A and B : if v_0 is in A , then v_1 is in B , v_2 is in A , and so on. After k steps, we must return to the same part as v_0 . If k is odd, the last vertex $v_k = v_0$ would be in the opposite part from v_0 , a contradiction. Therefore, all cycles in a bipartite graph have even length.

(\Leftarrow) Suppose G has no odd cycle. We show G is 2-colorable. For each connected component, pick any vertex s , run BFS from s , and assign colors based on distance: vertices at even distance from s get color 0, vertices at odd distance get color 1. We claim this is a valid 2-coloring.

Consider any edge $\{u, v\}$. If $\text{distTo}[u]$ and $\text{distTo}[v]$ have the same parity, then the path from s to u (length $\text{distTo}[u]$), the edge $\{u, v\}$ (length 1), and the reverse path from v to s (length $\text{distTo}[v]$) form a closed walk of length $\text{distTo}[u] + 1 + \text{distTo}[v]$, which has odd length (since $\text{distTo}[u]$ and $\text{distTo}[v]$ have the same parity). A closed walk of odd length contains an odd cycle (this is a standard result in graph theory). But we assumed G has no odd cycle, a contradiction. Therefore, $\text{distTo}[u]$ and $\text{distTo}[v]$ must have different parity, meaning u and v get different colors.

Bipartite Detection via BFS

The proof above gives us an algorithm: run BFS and check whether any edge connects two vertices at the same distance parity.

```

procedure IsBipartite(G):
    for each vertex v in G:
        color[v] = -1      // uncolored

    for each vertex v in G:
        if color[v] == -1: // not yet colored (new component)
            // BFS from v
            color[v] = 0
            Q = new Queue()
            Q.enqueue(v)

            while Q is not empty:
                u = Q.dequeue()
                for each w in G.adj(u):
                    if color[w] == -1:
                        color[w] = 1 - color[u]    // opposite color
                        Q.enqueue(w)
                    else if color[w] == color[u]:
                        return false                // odd cycle detected

    return true    // successfully 2-colored

```

The algorithm attempts to 2-color the graph using BFS. When visiting an unmarked neighbor, it assigns the opposite color. If it ever finds an edge between two vertices of the same color, the graph is not bipartite.

PROPOSITION F. *The BFS bipartite detection algorithm correctly determines whether a graph is bipartite in $O(V + E)$ time.*

Proof. If the algorithm returns `true`, it has produced a valid 2-coloring, so the graph is bipartite.

If the algorithm returns `false`, it found an edge $\{u, w\}$ where $\text{color}[u] = \text{color}[w]$. Consider the BFS tree paths from the source to `u` and from the source to `w`. These paths, together with the edge $\{u, w\}$, form a closed walk. Since `u` and `w` have the same color (same distance parity from the source), the walk has odd length, and therefore contains an odd cycle. By Proposition E, the graph is not bipartite.

The running time is $O(V + E)$ because the algorithm is simply BFS with a constant-time check per edge.

Bipartite Detection Trace

Consider the following graph:

```
0 --- 1 --- 2
|       |
3 --- 4 --- 5
```

Edges: {0,1}, {1,2}, {0,3}, {3,4}, {4,5}, {2,5}

BFS from vertex 0:

```

Step 0: color[0] = 0. Enqueue 0.
        Queue: [0]

Step 1: Dequeue 0. Neighbors: 1, 3
        color[1] = 1 (opposite of 0). Enqueue 1.
        color[3] = 1 (opposite of 0). Enqueue 3.
        Queue: [1, 3]

Step 2: Dequeue 1. Neighbors: 0, 2
        color[0] = 0, color[1] = 1: different -- OK
        color[2] = 0 (opposite of 1). Enqueue 2.
        Queue: [3, 2]

Step 3: Dequeue 3. Neighbors: 0, 4
        color[0] = 0, color[3] = 1: different -- OK
        color[4] = 0 (opposite of 1). Enqueue 4.
        Queue: [2, 4]

Step 4: Dequeue 2. Neighbors: 1, 5
        color[1] = 1, color[2] = 0: different -- OK
        color[5] = 1 (opposite of 0). Enqueue 5.
        Queue: [4, 5]

Step 5: Dequeue 4. Neighbors: 3, 5
        color[3] = 1, color[4] = 0: different -- OK
        color[5] = 1, color[4] = 0: different -- OK
        Queue: [5]

Step 6: Dequeue 5. Neighbors: 4, 2
        color[4] = 0, color[5] = 1: different -- OK
        color[2] = 0, color[5] = 1: different -- OK
        Queue: []

```

Result: Bipartite. The two-coloring is:

- Color 0 (Set A): {0, 2, 4}
- Color 1 (Set B): {1, 3, 5}

Now consider a non-bipartite example. Take the graph with an odd cycle:

```
0 --- 1
|   / |
|   / |
3 --- 2
```

Edges: {0,1}, {1,2}, {2,3}, {0,3}, {1,3}

BFS from vertex 0:

```
Step 0: color[0] = 0. Enqueue 0.

Step 1: Dequeue 0. Neighbors: 1, 3
        color[1] = 1. Enqueue 1.
        color[3] = 1. Enqueue 3.

Step 2: Dequeue 1. Neighbors: 0, 2, 3
        color[0] = 0, color[1] = 1: different -- OK
        color[2] = 0 (opposite of 1). Enqueue 2.
        color[3] = 1, color[1] = 1: SAME COLOR -- NOT BIPARTITE!
```

Result: Not bipartite. The edge {1, 3} connects two vertices of the same color. The odd cycle is 0 - 1 - 3 - 0 (length 3).

Applications of Bipartite Detection

Matching problems. In a bipartite graph, a **matching** is a set of edges with no shared vertices. The maximum matching problem -- finding the largest matching -- is solvable in polynomial time for bipartite graphs (via the Hopcroft-Karp algorithm) and is NP-hard for general graphs. Bipartite detection is the first step in many matching algorithms.

Two-machine scheduling. Given a set of jobs, some of which conflict (cannot run simultaneously), can the jobs be divided into two groups such that no two conflicting jobs are in the same group? This is equivalent to testing bipartiteness of the conflict graph.

Graph coloring. The chromatic number of a bipartite graph is at most 2 (or 1 if there are no edges). Testing bipartiteness is equivalent to testing 2-colorability. Testing k-colorability for $k \geq 3$ is NP-complete.

Stable matching. The Gale-Shapley algorithm for stable matching operates on a bipartite graph. Verifying that the input is indeed bipartite is a useful preprocessing step.

Part VII: Connected Graph Algorithms -- A Unified View

Estimated time: 10 minutes

The Graph Search Template

BFS and DFS both fit a common template, sometimes called "generic graph search":

```
procedure GenericGraphSearch(G, s):
    mark s
    add s to the "bag" (data structure)

    while bag is not empty:
        remove some vertex v from the bag
        for each unmarked neighbor w of v:
            mark w
            add w to the bag
```

The choice of "bag" determines the algorithm:

- **Stack** (LIFO): produces DFS.
- **Queue** (FIFO): produces BFS.
- **Priority queue** (min by some key): produces Dijkstra's algorithm (weighted shortest paths), Prim's algorithm (minimum spanning tree), or A* search (heuristic search), depending on the key.

This unified view highlights that many graph algorithms differ only in their frontier management strategy. The insight is powerful: by changing a single data structure, we fundamentally change what the algorithm computes.

Summary of Graph Algorithms So Far

Algorithm	Structure	Finds	Time	Chapters
DFS	Stack / recursion	Connected components, cycles, paths	$O(V + E)$	4.1
BFS	Queue	Shortest paths (unweighted), components	$O(V + E)$	4.1
Topological sort (Kahn)	Queue + in-degree	Topological order of DAG	$O(V + E)$	4.2
Topological sort (DFS)	Stack (reverse postorder)	Topological order of DAG	$O(V + E)$	4.2
Bipartite detection	BFS + 2-coloring	Whether graph is 2-colorable	$O(V + E)$	4.1
Directed BFS	Queue	Reachability, shortest directed paths	$O(V + E)$	4.2
Directed DFS	Stack / recursion	Reachability, cycle detection	$O(V + E)$	4.2

All of these algorithms run in $O(V + E)$ time, which is linear in the size of the graph. This is optimal in the sense that you cannot do better than examining every vertex and every edge at least once.

In the next lectures, we will study weighted graphs, where edges have associated costs, and algorithms like Dijkstra's and Kruskal's/Prim's that address shortest paths and minimum spanning trees in that richer setting.

Practice Problems

Problem 1: BFS Trace

Problem. Run BFS from vertex **A** on the following graph and give the contents of the queue after processing each vertex. Also give the final `distTo[]` and `edgeTo[]` arrays.

```
A --- B --- E
|       / |
C --- D   F
```

Edges: {A,B}, {A,C}, {B,E}, {C,D}, {D,E}, {E,F}

Solution.

Adjacency lists (alphabetical order):

```
A: [B, C]
B: [A, E]
C: [A, D]
D: [C, E]
E: [B, D, F]
F: [E]
```

Initialization: Mark A, $\text{distTo}[A] = 0$, enqueue A.

```
Queue: [A]
```

Step 1: Dequeue A. Neighbors: B (unmarked), C (unmarked).

Mark B, $\text{edgeTo}[B] = A$, $\text{distTo}[B] = 1$, enqueue B.

Mark C, $\text{edgeTo}[C] = A$, $\text{distTo}[C] = 1$, enqueue C.

```
Queue: [B, C]
```

Step 2: Dequeue B. Neighbors: A (marked), E (unmarked).

Mark E, $\text{edgeTo}[E] = B$, $\text{distTo}[E] = 2$, enqueue E.

```
Queue: [C, E]
```

Step 3: Dequeue C. Neighbors: A (marked), D (unmarked).

Mark D, $\text{edgeTo}[D] = C$, $\text{distTo}[D] = 2$, enqueue D.

```
Queue: [E, D]
```

Step 4: Dequeue E. Neighbors: B (marked), D (marked), F (unmarked).

Mark F, $\text{edgeTo}[F] = E$, $\text{distTo}[F] = 3$, enqueue F.

```
Queue: [D, F]
```

Step 5: Dequeue D. Neighbors: C (marked), E (marked). No new vertices.

```
Queue: [F]
```

Step 6: Dequeue F. Neighbors: E (marked). No new vertices.

```
Queue: []
```

Final arrays:

Vertex:	A	B	C	D	E	F
distTo:	0	1	1	2	2	3
edgeTo:	--	A	A	C	B	E

Shortest path from A to F: $F \leftarrow E \leftarrow B \leftarrow A$, i.e., A - B - E - F (length 3).

Problem 2: Shortest Path Computation

Problem. In the following unweighted graph, find the shortest path from vertex 0 to vertex 7. Give the path and its length.

```
0 -- 1 -- 2 -- 3
|       |   |
4 -- 5   6 -- 7
```

Edges: {0,1}, {1,2}, {2,3}, {0,4}, {4,5}, {2,6}, {3,7}, {6,7}

Solution.

Run BFS from vertex 0. Adjacency lists:

```
0: [1, 4]
1: [0, 2]
2: [1, 3, 6]
3: [2, 7]
4: [0, 5]
5: [4]
6: [2, 7]
7: [3, 6]
```

BFS from 0:

```
Enqueue 0 (dist 0).
Dequeue 0: mark 1 (dist 1), mark 4 (dist 1). Queue: [1, 4]
Dequeue 1: mark 2 (dist 2). Queue: [4, 2]
Dequeue 4: mark 5 (dist 2). Queue: [2, 5]
Dequeue 2: mark 3 (dist 3), mark 6 (dist 3). Queue: [5, 3, 6]
Dequeue 5: no new. Queue: [3, 6]
Dequeue 3: mark 7 (dist 4), edgeTo[7] = 3. Queue: [6, 7]
Dequeue 6: 7 already marked. Queue: [7]
Dequeue 7: no new. Queue: []
```

distTo[7] = 4. Path: $7 \leftarrow 3 \leftarrow 2 \leftarrow 1 \leftarrow 0$, i.e., **0 - 1 - 2 - 3 - 7** (length 4).

Note that the path $0 - 1 - 2 - 6 - 7$ also has length 4. Both are valid shortest paths. BFS found the one through vertex 3 because 3 was dequeued before 6.

Problem 3: Topological Sort Trace

Problem. Compute a topological sort of the following DAG using Kahn's algorithm.

```
Vertices: {A, B, C, D, E, F}
Edges: A->B, A->C, B->D, C->D, C->E, D->F, E->F
```

Solution.

In-degree computation:

Vertex:	A	B	C	D	E	F
inDegree:	0	1	1	2	1	2

Initial queue (vertices with in-degree 0): {A}

Step 1: Dequeue A. Output: [A]. Decrement neighbors:

- B: inDegree 1 -> 0, enqueue B
- C: inDegree 1 -> 0, enqueue C

Queue: [B, C]

Step 2: Dequeue B. Output: [A, B]. Decrement neighbors:

- D: inDegree 2 -> 1 (not 0)

Queue: [C]

Step 3: Dequeue C. Output: [A, B, C]. Decrement neighbors:

- D: inDegree 1 -> 0, enqueue D
- E: inDegree 1 -> 0, enqueue E

Queue: [D, E]

Step 4: Dequeue D. Output: [A, B, C, D]. Decrement neighbors:

- F: inDegree 2 -> 1 (not 0)

Queue: [E]

Step 5: Dequeue E. Output: [A, B, C, D, E]. Decrement neighbors:

- F: inDegree 1 -> 0, enqueue F

Queue: [F]

Step 6: Dequeue F. Output: [A, B, C, D, E, F].

Queue: []. count = 6 = V. Done.

Topological order: A, B, C, D, E, F.

Verification: Every edge goes from earlier to later in the ordering.

- A->B: A before B (valid)
- A->C: A before C (valid)
- B->D: B before D (valid)
- C->D: C before D (valid)
- C->E: C before E (valid)
- D->F: D before F (valid)
- E->F: E before F (valid)

Problem 4: Bipartite Check

Problem. Determine whether the following graph is bipartite. If yes, give the two-coloring. If no, identify an odd cycle.

```
1 --- 2 --- 3
|           |
4 --- 5 --- 6
      |
      7
```

Edges: {1,2}, {2,3}, {3,6}, {6,5}, {5,4}, {4,1}, {5,7}

Solution.

Run BFS from vertex 1 with 2-coloring:

```
color[1] = 0. Enqueue 1.
Dequeue 1: neighbors 2, 4
  color[2] = 1. Enqueue 2.
  color[4] = 1. Enqueue 4.

Dequeue 2: neighbors 1, 3
  color[1] = 0, color[2] = 1: different -- OK
  color[3] = 0. Enqueue 3.

Dequeue 4: neighbors 1, 5
  color[1] = 0, color[4] = 1: different -- OK
  color[5] = 0. Enqueue 5.

Dequeue 3: neighbors 2, 6
  color[2] = 1, color[3] = 0: different -- OK
  color[6] = 1. Enqueue 6.

Dequeue 5: neighbors 4, 6, 7
  color[4] = 1, color[5] = 0: different -- OK
  color[6] = 1, color[5] = 0: different -- OK
  color[7] = 1. Enqueue 7.

Dequeue 6: neighbors 3, 5
  color[3] = 0, color[6] = 1: different -- OK
  color[5] = 0, color[6] = 1: different -- OK

Dequeue 7: neighbors 5
  color[5] = 0, color[7] = 1: different -- OK
```

Result: Bipartite.

Two-coloring:

- Color 0 (Set A): {1, 3, 5}
- Color 1 (Set B): {2, 4, 6, 7}

Verification: Every edge connects a vertex in Set A with a vertex in Set B.

Problem 5: Digraph Reachability

Problem. Given the following digraph, determine which vertices are reachable from vertex 0.

```
0 -> 1 -> 3
|         ^
v         |
2 -> 4     5
|
v
5 -> 6
```

Edges: 0->1, 0->2, 1->3, 2->4, 4->5, 5->3, 5->6

Solution.

Run directed BFS (or DFS) from vertex 0:

```
Start: mark 0, enqueue 0.
```

```
Dequeue 0: outgoing edges to 1, 2.
```

```
Mark 1, enqueue 1.
```

```
Mark 2, enqueue 2.
```

```
Dequeue 1: outgoing edges to 3.
```

```
Mark 3, enqueue 3.
```

```
Dequeue 2: outgoing edges to 4.
```

```
Mark 4, enqueue 4.
```

```
Dequeue 3: no outgoing edges.
```

```
Dequeue 4: outgoing edges to 5.
```

```
Mark 5, enqueue 5.
```

```
Dequeue 5: outgoing edges to 3 (marked), 6 (unmarked).
```

```
Mark 6, enqueue 6.
```

```
Dequeue 6: no outgoing edges.
```

Vertices reachable from 0: {0, 1, 2, 3, 4, 5, 6} -- all vertices are reachable.

Distances from 0:

```
Vertex:  0  1  2  3  4  5  6
distTo:  0  1  1  2  2  3  4
```

Note that vertex 3 is reachable via two paths: 0->1->3 (length 2) and 0->2->4->5->3 (length 4). BFS correctly finds the shortest: distance 2 via 0->1->3.

Problem 6: Cycle Detection in a Digraph via Topological Sort

Problem. Use Kahn's algorithm to determine whether the following digraph is a DAG. If it has a cycle, identify it.

Edges: A->B, B->C, C->D, D->B, A->E

Solution.

In-degree computation:

Vertex:	A	B	C	D	E
inDegree:	0	2	1	1	1

Initial queue (in-degree 0): {A}

Step 1: Dequeue A. Output: [A]. count = 1.

- B: inDegree 2 -> 1

- E: inDegree 1 -> 0, enqueue E

Queue: [E]

Step 2: Dequeue E. Output: [A, E]. count = 2.

- No outgoing edges from E.

Queue: []

Queue is empty, but count = 2 < 5 = V. The graph has a cycle!

Vertices not processed: {B, C, D}. These vertices form the cycle (or are involved in one). Indeed, B -> C -> D -> B is a directed cycle of length 3.

Kahn's algorithm correctly detects the cycle: the vertices B, C, and D all have positive in-degree throughout the algorithm because each depends on another vertex in the cycle. Their in-degrees never reach 0, so they are never enqueued.

Problem 7: Multi-Source BFS

Problem. In the following graph, compute the shortest distance from each vertex to the nearest source in the set $S = \{0, 5\}$.

```
0 --- 1 --- 2 --- 3 --- 4 --- 5
```

Edges: {0,1}, {1,2}, {2,3}, {3,4}, {4,5}

Solution.

Run multi-source BFS with sources {0, 5}:

```
Initialize: mark 0, mark 5, distTo[0] = 0, distTo[5] = 0.
```

```
Enqueue both: Queue = [0, 5]
```

```
Dequeue 0: neighbor 1 unmarked.
```

```
    distTo[1] = distTo[0] + 1 = 1. Mark 1. Enqueue 1.
```

```
Dequeue 5: neighbor 4 unmarked.
```

```
    distTo[4] = distTo[5] + 1 = 1. Mark 4. Enqueue 4.
```

```
Queue: [1, 4]
```

```
Dequeue 1: neighbors 0 (marked), 2 (unmarked).
```

```
    distTo[2] = distTo[1] + 1 = 2. Mark 2. Enqueue 2.
```

```
Dequeue 4: neighbors 5 (marked), 3 (unmarked).
```

```
    distTo[3] = distTo[4] + 1 = 2. Mark 3. Enqueue 3.
```

```
Queue: [2, 3]
```

```
Dequeue 2: neighbors 1 (marked), 3 (marked). No new.
```

```
Dequeue 3: neighbors 2 (marked), 4 (marked). No new.
```

```
Queue: []
```

Result:

```
Vertex:  0  1  2  3  4  5
distTo:  0  1  2  2  1  0
```

Each vertex's distance is to the nearest source: vertices 0-2 are closest to source 0, vertices 3-5 are closest to source 5. Vertices 2 and 3 are equidistant from both sources (distance 2).

Problem 8: BFS on a Disconnected Graph

Problem. Run BFS on the following graph starting from vertex 0. Which vertices are visited? How would you modify BFS to visit all vertices even in a disconnected graph?

```
Component 1: 0 --- 1 --- 2
Component 2: 3 --- 4
Component 3: 5
```

Edges: {0,1}, {1,2}, {3,4}

Solution.

BFS from vertex 0:

```
Mark 0, enqueue 0.
Dequeue 0: mark 1, enqueue 1.
Dequeue 1: mark 2, enqueue 2.
Dequeue 2: no unmarked neighbors.
Queue empty.
```

Vertices visited: {0, 1, 2}. Vertices 3, 4, and 5 are not visited because they are in different connected components.

To visit all vertices, wrap BFS in a loop:

```
procedure BFS_All(G):
  for each vertex v in G:
    marked[v] = false

  for each vertex v in G:
    if not marked[v]:
      BFS(G, v)  // start a new BFS from this unvisited vertex
```

This produces a **BFS forest**: one BFS tree per connected component. The total running time is still $O(V + E)$, since each vertex and edge is processed exactly once across all BFS calls.

For this example:

- BFS from 0 visits {0, 1, 2}
- BFS from 3 visits {3, 4}
- BFS from 5 visits {5}

Three connected components detected.

Summary

This lecture covered breadth-first search and several major graph applications that build on BFS and directed graph theory.

Key Concepts

1. **BFS** explores a graph layer by layer using a FIFO queue, maintaining `marked[]`, `edgeTo[]`, and `distTo[]` arrays. It runs in $O(V + E)$ time.
2. **Shortest paths in unweighted graphs** are computed naturally by BFS. The `distTo[]` array gives exact shortest-path distances, provably correct by induction on distance. Path reconstruction follows `edgeTo[]` links backward.
3. **BFS vs DFS**: Both run in $O(V + E)$ time, but BFS uses a queue and finds shortest paths, while DFS uses a stack and is better suited for structural analysis (cycles, SCCs, articulation points). The generic graph search template unifies both.
4. **Directed graphs** extend undirected graphs with edge direction. Key concepts include in-degree, out-degree, reachability, DAGs, and strong connectivity. Both BFS and DFS extend naturally to digraphs.
5. **Topological sort** orders the vertices of a DAG so that all edges point forward. Kahn's algorithm uses BFS with in-degree counting; DFS-based topological sort uses reverse postorder. Both run in $O(V + E)$ time and can detect cycles.

6. **Bipartite detection** uses BFS two-coloring: assign opposite colors to neighbors and check for conflicts. A graph is bipartite if and only if it has no odd cycle. The algorithm runs in $O(V + E)$ time.

Algorithm Comparison Table

Algorithm	Input	Output	Data Structure	Time
BFS	Graph + source	Shortest paths, distances	Queue	$O(V+E)$
DFS	Graph + source	Reachability, tree structure	Stack/recursion	$O(V+E)$
Kahn's topological sort	DAG	Topological order	Queue + inDeg[]	$O(V+E)$
DFS topological sort	DAG	Topological order	Stack (postorder)	$O(V+E)$
Bipartite detection	Graph	2-coloring or odd cycle	Queue + color[]	$O(V+E)$
Multi-source BFS	Graph + sources	Nearest-source distances	Queue	$O(V+E)$

Connection to Companion C++ Code

The companion C++ implementation provides concrete implementations of:

- **Graph** : Undirected graph using adjacency lists. Supports `addEdge()`, `adj()`, `V()`, `E()`.
- **Digraph** : Directed graph using adjacency lists. Supports `addEdge()`, `adj()`, `reverse()`, in-degree computation.
- **bfsTraversal()** : Standard BFS with `marked[]`, `edgeTo[]`, `distTo[]`. Demonstrates queue-based layer-by-layer exploration.
- **bfsShortestPath()** : BFS with parent reconstruction. Uses `edgeTo[]` to build the actual shortest path via a stack-based reversal.
- **isBipartite()** : BFS two-coloring algorithm. Returns `true` if the graph is bipartite, `false` if an odd cycle is detected.
- **topologicalSort()** : Kahn's algorithm implementation with in-degree array, queue, and cycle detection (checks `count == V`).

These implementations make the pseudocode presented in this lecture concrete and executable, providing a foundation for experimentation and further study.

References

1. Sedgewick, R. and Wayne, K. *Algorithms*, 4th Edition. Addison-Wesley, 2011. Chapters 4.1 (Undirected Graphs) and 4.2 (Directed Graphs).
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd Edition. MIT Press, 2009. Chapters 22.1-22.4 (BFS, DFS, Topological Sort, Strongly Connected Components).
3. Moore, E. F. "The Shortest Path Through a Maze." *Proceedings of the International Symposium on the Theory of Switching*, Harvard University Press, 1959, pp. 285-292.
4. Kahn, A. B. "Topological Sorting of Large Networks." *Communications of the ACM*, Vol. 5, No. 11, November 1962, pp. 558-562.
5. Tarjan, R. E. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing*, Vol. 1, No. 2, 1972, pp. 146-160.
6. Kleinberg, J. and Tardos, E. *Algorithm Design*. Pearson, 2005. Chapter 3 (Graphs: BFS, DFS, Bipartiteness, Connectivity).
7. Skiena, S. S. *The Algorithm Design Manual*, 2nd Edition. Springer, 2008. Chapter 5 (Graph Traversal).
8. Even, S. *Graph Algorithms*, 2nd Edition. Cambridge University Press, 2011. Chapters on BFS, DFS, and network flow.
9. West, D. B. *Introduction to Graph Theory*, 2nd Edition. Prentice Hall, 2001. Chapters on graph coloring and bipartite graphs.
10. Sedgewick, R. and Wayne, K. *Algorithms* companion website: <https://algs4.cs.princeton.edu/>. Sections 4.1 and 4.2 with Java implementations and visualizations.

Lecture 10 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition