

LECTURE 9 OF 12

Graph Fundamentals and Depth-First Search

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

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

Chapter 4.1

Duration: 4 hours (with breaks)

Learning Objectives

1. Define fundamental graph terminology -- vertices, edges, paths, cycles, connectivity, and degree -- and identify these structures in real-world modeling problems such as social networks, transportation maps, and dependency scheduling.
2. Implement and compare the two canonical graph representations -- adjacency matrix and adjacency list -- analyzing their respective time and space complexities and understanding the engineering trade-offs that govern which representation to select.
3. Trace the execution of depth-first search (DFS) in both its recursive and iterative (explicit stack) formulations, maintaining a precise mental model of the marked array, the call stack or explicit stack, and the edgeTo parent-link array at every step of the algorithm.
4. Apply DFS to solve three classical graph problems -- path finding, connected component identification, and cycle detection -- understanding how a single linear-time traversal can extract rich structural information from an undirected graph.
5. Classify edges in a DFS traversal (tree edges, back edges, and, for directed graphs, forward and cross edges), explain their significance, and use edge classification to reason about graph properties such as acyclicity and bipartiteness.

6. Articulate and apply the graph-processing design pattern advocated by Sedgewick -- decoupling graph representation from graph-processing algorithms via constructor-based computation and query methods -- and recognize this pattern in the Paths, CC, and Cycle API classes.
-

Part I: Graph Terminology and Motivation

Estimated time: 20 minutes

1.1 Why Graphs?

Graphs are arguably the most versatile data structure in all of computer science. Whereas arrays, linked lists, stacks, queues, and trees each model a particular kind of relationship -- sequential access, hierarchical containment, LIFO or FIFO ordering -- a graph imposes no such structural constraint. A graph simply says: here are some objects, and here are pairwise relationships between them. That extraordinary generality is what makes graph algorithms simultaneously so powerful and so challenging.

Consider the range of phenomena that graphs model naturally. A social network is a graph in which the vertices are people and the edges represent friendship or following relationships. The World Wide Web is a graph whose vertices are pages and whose edges are hyperlinks. A road map is a graph with intersections as vertices and road segments as edges. A scheduling problem -- which tasks must precede which other tasks -- is a graph of dependencies. Electrical circuits, protein interaction networks, airline routes, the Internet backbone, citation networks in academic literature, compiler data-flow analysis: all of these are graphs.

The algorithms we study in this lecture and the next three lectures constitute the core toolkit for extracting useful information from graphs: Can we get from vertex A to vertex B? What is the shortest path? Are there cycles? How many connected pieces does the graph have? These questions arise in every domain where graphs appear, and the algorithms that answer them are among the most elegant and practically important in all of computing.

1.2 Formal Definition

DEFINITION (Graph). A *graph* $G = (V, E)$ consists of a finite set V of *vertices* (also called *nodes*) and a finite set E of *edges* (also called *links* or *arcs*). Each edge is a pair of vertices. In an *undirected graph*, an edge is an unordered pair $\{u, v\}$, meaning the connection between u and v is symmetric. In a *directed graph*

(or *digraph*), an edge is an ordered pair (u, v) , meaning the connection goes from u to v but not necessarily from v to u .

We denote the number of vertices by $V = |V|$ and the number of edges by $E = |E|$. When there is no ambiguity, we use V and E to refer both to the sets and to their cardinalities.

DEFINITION (Adjacency). Two vertices u and v in an undirected graph are *adjacent* (or *neighbors*) if $\{u, v\}$ is in E . In a directed graph, we say v is *adjacent to* u if (u, v) is in E .

DEFINITION (Degree). The *degree* of a vertex v in an undirected graph is the number of edges incident to v :

$$\deg(v) = |\{ e \text{ in } E : v \text{ in } e \}|$$

In a directed graph, the *in-degree* of v is the number of edges pointing to v , and the *out-degree* is the number of edges pointing away from v .

1.3 Paths, Cycles, and Connectivity

DEFINITION (Path). A *path* in a graph is a sequence of vertices v_0, v_1, \dots, v_k such that each consecutive pair (v_i, v_{i+1}) is an edge. The *length* of the path is k (the number of edges). A *simple path* is a path in which no vertex is repeated.

DEFINITION (Cycle). A *cycle* is a path v_0, v_1, \dots, v_k where $v_0 = v_k$ and $k \geq 3$ (for undirected graphs) or $k \geq 1$ (for directed graphs). A *simple cycle* is a cycle in which no vertex other than the start/end vertex is repeated.

DEFINITION (Connectivity). An undirected graph is *connected* if there is a path between every pair of vertices. A *connected component* is a maximal connected subgraph. Every graph can be uniquely decomposed into its connected components.

DEFINITION (Acyclic Graph). A graph with no cycles is called *acyclic*. An undirected acyclic connected graph is called a *tree*. An undirected acyclic graph (possibly disconnected) is called a *forest*.

1.4 Special Graph Types

DEFINITION (Simple Graph). A *simple graph* is a graph with no self-loops (edges from a vertex to itself) and no parallel edges (multiple edges between the same pair of vertices). Unless stated otherwise, "graph" in this course means "simple undirected graph."

DEFINITION (Multigraph). A *multigraph* allows parallel edges between the same pair of vertices. A *pseudograph* additionally allows self-loops.

DEFINITION (Bipartite Graph). A graph $G = (V, E)$ is *bipartite* if the vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge connects a vertex in V_1 to a vertex in V_2 . Equivalently, a graph is bipartite if and only if it contains no odd-length cycle.

DEFINITION (Weighted Graph). A *weighted graph* assigns a numerical value (the *weight*) to each edge. Weights can represent distances, costs, capacities, or any other quantitative measure of the relationship.

DEFINITION (Subgraph). A graph $H = (V', E')$ is a *subgraph* of $G = (V, E)$ if V' is a subset of V and E' is a subset of E , and every edge in E' connects two vertices in V' .

1.5 Real-World Examples

Social Networks. Facebook's friendship graph has billions of vertices and hundreds of billions of edges. Each vertex is a user; each undirected edge represents a mutual friendship. Algorithms on this graph power friend recommendations (short paths), community detection (connected components, clustering), and influence analysis (centrality measures).

The World Wide Web. Each web page is a vertex; each hyperlink is a directed edge. Google's PageRank algorithm -- the foundation of web search -- is a graph algorithm that assigns importance scores based on the link structure. The web graph is directed because a link from page A to page B does not imply a link from B to A.

Road Networks. Intersections are vertices; road segments are weighted edges (weighted by distance or travel time). Navigation systems like Google Maps use shortest-path algorithms (Dijkstra's algorithm, A* search) on these graphs millions of times per second.

Task Scheduling. In project management, tasks are vertices and precedence constraints are directed edges: if task A must be completed before task B begins, we draw an edge from A to B. Topological sort (a DFS application we study in Lecture 10) produces a valid schedule.

Compiler Dependency Analysis. Modules in a software system form a directed graph based on import/dependency relationships. Cycle detection tells us if there are circular dependencies; topological sort gives us a valid compilation order.

1.6 Fundamental Graph Properties

PROPOSITION A. In any undirected graph, the sum of all vertex degrees equals twice the number of edges.

Proof. Each edge $\{u, v\}$ contributes exactly 1 to $\deg(u)$ and exactly 1 to $\deg(v)$. Therefore every edge contributes exactly 2 to the total degree sum. Summing over all edges gives the result: the sum of $\deg(v)$ over all vertices v equals $2E$. *QED.*

This is sometimes called the *handshaking lemma* because if every person at a party counts the number of hands they shake, the total count is twice the number of handshakes.

PROPOSITION B. A tree on V vertices has exactly $V - 1$ edges.

Proof. By induction on V . Base case: a single vertex has 0 edges, and $V - 1 = 0$. Inductive step: assume every tree on k vertices has $k - 1$ edges. Consider a tree T on $k + 1$ vertices. Since T is connected and acyclic, it has at least one leaf (a vertex of degree 1). Removing a leaf and its incident edge produces a tree on k vertices, which by hypothesis has $k - 1$ edges. Therefore T has $(k - 1) + 1 = k$ edges. *QED.*

PROPOSITION C. A connected graph on V vertices has at least $V - 1$ edges.

Proof. A spanning tree of the graph uses exactly $V - 1$ edges, and every connected graph contains a spanning tree. Therefore $E \geq V - 1$. *QED.*

Part II: Graph Representations

Estimated time: 30 minutes

2.1 Overview

To work with graphs algorithmically, we must represent them in memory. The two standard representations are the *adjacency matrix* and the *adjacency list*. Each has strengths and weaknesses, and the choice between them depends on the density of the graph and the operations the algorithm requires.

Throughout this section, we assume vertices are labeled 0 through $V - 1$. This convention, which Sedgewick adopts consistently, allows us to use vertex labels as array indices, avoiding the overhead of hash maps or search trees.

2.2 Adjacency Matrix

DEFINITION (Adjacency Matrix). For a graph $G = (V, E)$ with V vertices, the *adjacency matrix* is a $V \times V$ boolean matrix A where:

```
A[u][v] = true   if {u, v} is an edge
A[u][v] = false  otherwise
```

For an undirected graph, the adjacency matrix is symmetric: $A[u][v] = A[v][u]$ for all u, v . For a weighted graph, we store the edge weight in $A[u][v]$ instead of a boolean, using a sentinel value (such as infinity or zero) for non-edges.

```

Structure GraphAdjMatrix:
    V      : integer          // number of vertices
    E      : integer          // number of edges
    adj[V][V] : boolean matrix // adjacency matrix

Procedure InitGraphMatrix(numVertices):
    V <- numVertices
    E <- 0
    for u from 0 to V - 1:
        for v from 0 to V - 1:
            adj[u][v] <- false

Procedure AddEdgeMatrix(u, v):
    if not adj[u][v]:
        adj[u][v] <- true
        adj[v][u] <- true // omit for directed graph
    E <- E + 1

Function HasEdgeMatrix(u, v):
    return adj[u][v]

Function NeighborsMatrix(u):
    result <- empty list
    for v from 0 to V - 1:
        if adj[u][v]:
            append v to result
    return result

```

Space complexity: $\Theta(V^2)$, regardless of the number of edges. This makes the adjacency matrix impractical for sparse graphs (graphs where E is much less than V^2) but acceptable for dense graphs.

Time complexity:

- Edge query (`HasEdge(u, v)`): $O(1)$ -- simply index into the matrix.
- Enumerate neighbors of u (`Neighbors(u)`): $O(V)$ -- must scan an entire row.
- Add an edge: $O(1)$.

The adjacency matrix excels when the algorithm frequently asks "is there an edge between u and v ?" and the graph is dense. Its chief disadvantage is the $O(V^2)$ space requirement, which becomes prohibitive for large sparse graphs. A graph with 10 million vertices but only 20 million edges (a common scenario in

social networks and web graphs) would require 10^{14} bits -- roughly 12 terabytes -- in adjacency matrix form, but only about 200 megabytes in adjacency list form.

2.3 Adjacency List

DEFINITION (Adjacency List). For a graph $G = (V, E)$, the *adjacency list* representation maintains, for each vertex v , a list of all vertices adjacent to v . The entire representation is an array of V lists (or similar dynamic collections).

```
Structure GraphAdjList:
    V      : integer           // number of vertices
    E      : integer           // number of edges
    adj[V] : array of linked lists // adjacency lists

Procedure InitGraphList(numVertices):
    V <- numVertices
    E <- 0
    for v from 0 to V - 1:
        adj[v] <- empty list

Procedure AddEdgeList(u, v):
    prepend v to adj[u]
    prepend u to adj[v]      // omit for directed graph
    E <- E + 1

Function HasEdgeList(u, v):
    for each w in adj[u]:
        if w = v:
            return true
    return false

Function NeighborsList(u):
    return adj[u]
```

Space complexity: $\Theta(V + E)$. Each vertex has one list header, and each edge contributes exactly two entries (one in each endpoint's list) for an undirected graph, or one entry for a directed graph.

Time complexity:

- Edge query (`HasEdge(u, v)`): $O(\deg(u))$ -- must scan the adjacency list of u .
- Enumerate neighbors of u (`Neighbors(u)`): $O(\deg(u))$ -- simply iterate the list.
- Add an edge: $O(1)$ -- prepend to the list.

The adjacency list is the representation of choice for most graph algorithms. Nearly all important graph algorithms -- DFS, BFS, Dijkstra's algorithm, topological sort, strongly connected components -- iterate over the neighbors of each vertex, and the adjacency list provides this operation in optimal time. The space requirement is proportional to the actual size of the graph, not to the square of the number of vertices.

2.4 Comparison Table

Operation	Adjacency Matrix	Adjacency List
Space	$\Theta(V^2)$	$\Theta(V + E)$
Add edge	$O(1)$	$O(1)$
Has edge (u,v) ?	$O(1)$	$O(\deg(u))$
Iterate neighbors	$O(V)$	$O(\deg(u))$
Remove edge	$O(1)$	$O(\deg(u))$
Best for	Dense graphs	Sparse graphs

2.5 When to Use Which

Use the adjacency matrix when:

- The graph is dense (E is close to V^2).
- The algorithm requires frequent edge-existence queries.
- V is small enough that V^2 fits comfortably in memory.
- Examples: Floyd-Warshall all-pairs shortest paths, small constraint satisfaction problems.

Use the adjacency list when:

- The graph is sparse (E is much less than V^2).
- The algorithm primarily iterates over neighbors of vertices.
- Memory is a concern.

- Examples: DFS, BFS, Dijkstra, Kruskal, Prim, topological sort -- virtually all algorithms we study in this course.

In practice, most real-world graphs are sparse. The World Wide Web has billions of pages but each page links to only a modest number of others. A road network has millions of intersections but each intersection connects to only a handful of roads. Social networks have millions of users but each user has at most a few thousand friends. For all of these, the adjacency list is the clear winner.

2.6 Sedgewick's Graph API

Sedgewick defines a clean, minimal API for undirected graphs. The key insight is that the `Graph` class should provide only the representation -- the ability to add edges and iterate over neighbors -- and should NOT include any processing algorithms. Processing algorithms are implemented in separate classes that take a `Graph` object as input. We will return to this design principle in Part VI.

API for an Undirected Graph:

```
Graph(V)           : create a graph with V vertices and 0 edges
Graph(inputStream) : create a graph from an input stream
V()                : return the number of vertices
E()                : return the number of edges
addEdge(u, v)      : add edge {u, v}
adj(v)              : return an iterable of vertices adjacent to v
toString()          : return a string representation
```

This API says nothing about whether the implementation uses an adjacency list, an adjacency matrix, or some other structure. The algorithms we write against this API work correctly regardless of the underlying representation. In practice, Sedgewick's implementation uses adjacency lists (implemented with a `Bag` data structure, which is essentially an unordered linked list), because adjacency lists offer the best performance for the algorithms in the book.

2.7 Building a Graph from Input

A common input format lists the number of vertices, the number of edges, and then the edges themselves:

Example input:

```
13          // V = 13 vertices (labeled 0 through 12)
13          // E = 13 edges
0 5          // edge between 0 and 5
4 3          // edge between 4 and 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Procedure BuildGraphFromInput(input):

```
V <- readInteger(input)
E <- readInteger(input)
G <- new GraphAdjList(V)
for i from 0 to E - 1:
    u <- readInteger(input)
    v <- readInteger(input)
    G.AddEdge(u, v)
return G
```

The graph constructed from the input above has the following adjacency lists (order may vary due to prepending):

Vertex : Adjacent vertices

```
0 : 6, 2, 1, 5
1 : 0
2 : 0
3 : 5, 4
4 : 5, 6, 3
5 : 3, 4, 0
6 : 0, 4
7 : 8
8 : 7
9 : 11, 10, 12
10 : 9
11 : 9, 12
12 : 11, 9
```

Notice that this graph has three connected components: $\{0, 1, 2, 3, 4, 5, 6\}$, $\{7, 8\}$, and $\{9, 10, 11, 12\}$. We will use this example graph throughout the lecture.

Part III: Depth-First Search

Estimated time: 40 minutes

3.1 The Idea

Depth-first search (DFS) is a fundamental graph traversal algorithm. The idea is simple yet powerful: starting from a source vertex, explore as far as possible along each branch before backtracking. DFS follows a path until it reaches a dead end (a vertex with no unvisited neighbors), then backtracks to the most recently visited vertex that still has unvisited neighbors, and continues from there.

The name "depth-first" reflects this behavior: the algorithm plunges as deep as possible into the graph before exploring breadth. This stands in contrast to breadth-first search (BFS), which we study in Lecture 10, which explores all vertices at distance k before exploring any vertex at distance $k + 1$.

DFS has a distinguished history in computer science. It can be traced back to the 19th-century work of Charles Pierre Tremaux, who devised a systematic strategy for solving mazes. The algorithm was

formalized and analyzed in the context of graph theory by Tarjan (1972), who demonstrated its remarkable power by using DFS to solve a wide variety of graph problems in linear time, including finding connected components, biconnected components, strongly connected components, and topological orderings.

3.2 The Marked Array

The central data structure in DFS (and indeed in any graph traversal) is the *marked array* (sometimes called the *visited array*). This is a boolean array indexed by vertex:

```
marked[v] = true    if vertex v has been visited
marked[v] = false   if vertex v has not yet been visited
```

The marked array serves two critical purposes. First, it ensures termination: without it, the algorithm would cycle forever around any cycle in the graph. Second, it ensures efficiency: each vertex is processed at most once, giving us a linear-time algorithm.

Initially, all entries of `marked[]` are false. When DFS visits a vertex v , it sets `marked[v]` to true. Before visiting any neighbor w of v , DFS checks whether `marked[w]` is true; if so, it skips w . This simple mechanism guarantees that each vertex is visited exactly once.

3.3 Recursive DFS

The most natural formulation of DFS is recursive. The recursion directly mirrors the "go as deep as possible, then backtrack" strategy: visiting a vertex triggers recursive visits to all of its unvisited neighbors, and the call stack handles the backtracking automatically.

```

Procedure DFS_Recursive(G, s):
    // G: graph, s: source vertex
    // marked[]: boolean array, initially all false
    // edgeTo[]: integer array, for path reconstruction

    marked[s] <- true
    for each w in G.adj(s):
        if not marked[w]:
            edgeTo[w] <- s
            DFS_Recursive(G, w)

```

To search the entire graph (not just the component containing s):

```

Procedure DFS_Full(G):
    for v from 0 to G.V() - 1:
        marked[v] <- false
    for v from 0 to G.V() - 1:
        if not marked[v]:
            DFS_Recursive(G, v)

```

The edgeTo[] array records the *parent* of each vertex in the DFS tree: edgeTo[w] = s means "we discovered w by following an edge from s." This parent-link representation allows us to reconstruct paths from any visited vertex back to the source, as we will see in Section 4.1.

3.4 Iterative DFS Using an Explicit Stack

While the recursive formulation is elegant, it has a practical drawback: for very large graphs, the recursion depth can exceed the system's call-stack limit, causing a stack overflow. The iterative formulation replaces the implicit call stack with an explicit stack data structure, avoiding this problem.

```

Procedure DFS_Iterative(G, s):
    // G: graph, s: source vertex
    // marked[]: boolean array, initially all false
    // edgeTo[]: integer array, for path reconstruction

    stack <- empty stack
    push s onto stack

    while stack is not empty:
        v <- pop from stack
        if not marked[v]:
            marked[v] <- true
            for each w in G.adj(v):           // push in reverse order
                if not marked[w]:           // for consistency with
                    push w onto stack        // recursive version
                    edgeTo[w] <- v

```

Important subtlety: The iterative version may push the same vertex onto the stack multiple times (once for each neighbor that discovers it). This is handled by the `if not marked[v]` check after popping. When a vertex is popped a second time, it is already marked and is simply skipped. An alternative approach checks `marked[w]` before pushing, but this can produce different traversal orders than the recursive version and requires updating `edgeTo[]` carefully.

Note on push order: If the adjacency list of v stores neighbors in a particular order, the iterative DFS processes them in the reverse of that order (because a stack is LIFO). To match the recursive version's behavior exactly, push neighbors in reverse order. In practice, the specific traversal order rarely matters for correctness -- what matters is that all reachable vertices are visited.

3.5 Detailed DFS Trace

Let us trace DFS on a smaller example graph to build intuition. Consider this undirected graph with 6 vertices:

```
Graph (6 vertices, 8 edges):
```

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

```
Adjacency lists (each list in order shown):
```

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

We run recursive DFS starting from vertex 0. At each step, we show the current vertex, the action taken, and the state of the `marked[]` and `edgeTo[]` arrays.

RECURSIVE DFS TRACE (source = 0)

=====

Call: DFS(0)

Mark 0. marked = [T, F, F, F, F, F]

Check neighbor 1: not marked -> set edgeTo[1]=0, recurse.

Call: DFS(1)

Mark 1. marked = [T, T, F, F, F, F]

Check neighbor 0: marked -> skip.

Check neighbor 2: not marked -> set edgeTo[2]=1, recurse.

Call: DFS(2)

Mark 2. marked = [T, T, T, F, F, F]

Check neighbor 0: marked -> skip.

Check neighbor 1: marked -> skip.

Check neighbor 3: not marked -> set edgeTo[3]=2, recurse.

Call: DFS(3)

Mark 3. marked = [T, T, T, T, F, F]

Check neighbor 2: marked -> skip.

Check neighbor 4: not marked -> set edgeTo[4]=3, recurse.

Call: DFS(4)

Mark 4. marked = [T, T, T, T, T, F]

Check neighbor 2: marked -> skip.

Check neighbor 3: marked -> skip.

Return from DFS(4).

Check neighbor 5: not marked -> set edgeTo[5]=3, recurse.

Call: DFS(5)

Mark 5. marked = [T, T, T, T, T, T]

Check neighbor 0: marked -> skip.

Check neighbor 3: marked -> skip.

Return from DFS(5).

Return from DFS(3).

```
    Check neighbor 4: marked -> skip.  
    Return from DFS(2).
```

```
Return from DFS(1).
```

```
    Check neighbor 2: marked -> skip.  
    Check neighbor 5: marked -> skip.  
    Return from DFS(0).
```

```
FINAL STATE:
```

```
    marked[] = [T, T, T, T, T, T]  
    edgeTo[] = [-, 0, 1, 2, 3, 3]
```

```
Visit order: 0, 1, 2, 3, 4, 5
```

```
DFS TREE:
```

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

Now let us trace the same graph using the iterative DFS with an explicit stack:

ITERATIVE DFS TRACE (source = 0)

=====

Initialize: push 0 onto stack.

Stack (top on right): [0]

marked = [F, F, F, F, F, F]

Step 1: Pop 0. Not marked.

Mark 0. marked = [T, F, F, F, F, F]

Push neighbors of 0 in reverse order: push 5, push 2, push 1.

Stack: [5, 2, 1]

Step 2: Pop 1. Not marked.

Mark 1. marked = [T, T, F, F, F, F]

edgeTo[1] = 0.

Push neighbors of 1 in reverse: push 2, push 0.

0 is marked -> skip. Push 2.

Stack: [5, 2, 2]

Step 3: Pop 2. Not marked.

Mark 2. marked = [T, T, T, F, F, F]

edgeTo[2] = 1.

Push neighbors of 2 in reverse: push 4, push 3, push 1, push 0.

0 is marked -> skip. 1 is marked -> skip.

Push 3, push 4... wait, reverse order: adj is [0,1,3,4], reverse is [4,3,1,0].

Skip 0 (marked), skip 1 (marked), push 3, push 4.

Stack: [5, 2, 3, 4]

Step 4: Pop 4. Not marked.

Mark 4. marked = [T, T, T, F, T, F]

edgeTo[4] = 2.

Neighbors of 4: [2, 3]. Reverse: [3, 2].

2 is marked -> skip. 3 is not marked -> push 3.

Stack: [5, 2, 3, 3]

Step 5: Pop 3. Not marked.

Mark 3. marked = [T, T, T, T, T, F]

edgeTo[3] = 4.

Neighbors of 3: [2, 4, 5]. Reverse: [5, 4, 2].

```
2 is marked -> skip. 4 is marked -> skip. 5 not marked -> push 5.  
Stack: [5, 2, 3, 5]
```

Step 6: Pop 5. Not marked.

```
Mark 5. marked = [T, T, T, T, T, T]
```

```
edgeTo[5] = 3.
```

```
Neighbors of 5: [0, 3]. Both marked -> nothing pushed.
```

```
Stack: [5, 2, 3]
```

Step 7: Pop 3. Already marked -> skip.

```
Stack: [5, 2]
```

Step 8: Pop 2. Already marked -> skip.

```
Stack: [5]
```

Step 9: Pop 5. Already marked -> skip.

```
Stack: [] (empty)
```

Done.

FINAL STATE (iterative):

```
marked[] = [T, T, T, T, T, T]
```

```
edgeTo[] = [-, 0, 1, 4, 2, 3]
```

Note that the iterative version produced a different `edgeTo[]` array and thus a different DFS tree. This is expected: the iterative version processes neighbors in a different effective order because of the stack's LIFO behavior and the potential for duplicate pushes. Both trees are valid DFS trees, and both visit every vertex. The set of visited vertices is the same; only the traversal order and the specific tree edges differ.

Iterative DFS Tree:

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

3.6 Time Complexity of DFS

PROPOSITION D. Depth-first search, starting from a single source vertex, runs in $O(V + E)$ time and uses $O(V)$ extra space (beyond the graph representation itself).

Proof. We analyze the recursive version; the iterative version has the same asymptotic complexity.

Time. Each vertex is marked at most once (the `if not marked[w]` guard ensures this). Therefore, the DFS procedure is called at most V times. For each call $\text{DFS}(v)$, the algorithm iterates over all entries in $\text{adj}(v)$. The total work across all calls is:

Sum over all v of $\text{deg}(v) = 2E$ (by the handshaking lemma)

Adding the $O(V)$ work for initializing the marked array gives a total of $O(V + E)$.

Space. The `marked[]` array uses $O(V)$ space. The `edgeTo[]` array uses $O(V)$ space. The recursion depth is at most V (in the worst case of a path graph), so the call stack uses $O(V)$ space. Total extra space: $O(V)$. *QED.*

PROPOSITION E. DFS visits every vertex reachable from the source vertex s , and only those vertices.

Proof. (Reachability.) We prove that if v is reachable from s , then DFS marks v . Suppose for contradiction that some vertex v is reachable from s but DFS does not mark it. Let $s = v_0, v_1, \dots, v_k = v$ be a path from s to v . Let v_i be the last vertex on this path that DFS marks (such a vertex exists because DFS marks s). Then

v_{i+1} is not marked. But DFS examines all neighbors of v_i , and v_{i+1} is a neighbor of v_i , so DFS would have recursed into v_{i+1} -- contradiction.

(Only reachable vertices.) DFS only visits a vertex w by following an edge from a previously visited vertex. By induction, every visited vertex is connected to s by a path of visited vertices, hence is reachable from s . *QED*.

Part IV: DFS Applications

Estimated time: 35 minutes

4.1 Finding Paths

The `edgeTo[]` array computed by DFS encodes a *DFS tree* rooted at the source vertex s . By following parent links from any vertex v back to s , we can reconstruct the path from s to v . This path is not necessarily the shortest path (that requires BFS), but it is a valid path.

```

Class DepthFirstPaths:
    marked[] : boolean array
    edgeTo[] : integer array
    s        : integer (source vertex)

    Constructor(G, s):
        this.s <- s
        marked <- new boolean[G.V()], all false
        edgeTo <- new integer[G.V()]
        DFS(G, s)

    Procedure DFS(G, v):
        marked[v] <- true
        for each w in G.adj(v):
            if not marked[w]:
                edgeTo[w] <- v
                DFS(G, w)

    Function hasPathTo(v):
        return marked[v]

    Function pathTo(v):
        if not hasPathTo(v):
            return null
        path <- empty stack
        x <- v
        while x != s:
            push x onto path
            x <- edgeTo[x]
        push s onto path
        return path

```

Example. Using the DFS tree from Section 3.5 (source = 0, recursive version):

```

edgeTo[] = [-, 0, 1, 2, 3, 3]

pathTo(5):
  x = 5, push 5.  edgeTo[5] = 3.
  x = 3, push 3.  edgeTo[3] = 2.
  x = 2, push 2.  edgeTo[2] = 1.
  x = 1, push 1.  edgeTo[1] = 0.
  x = 0 = s, push 0.  Stop.
  Path: 0 -> 1 -> 2 -> 3 -> 5

pathTo(4):
  x = 4, push 4.  edgeTo[4] = 3.
  x = 3, push 3.  edgeTo[3] = 2.
  x = 2, push 2.  edgeTo[2] = 1.
  x = 1, push 1.  edgeTo[1] = 0.
  x = 0 = s, push 0.  Stop.
  Path: 0 -> 1 -> 2 -> 3 -> 4

```

Notice that the DFS path from 0 to 5 (length 4) is not the shortest path; the edge 0--5 exists, giving a direct path of length 1. DFS finds *a* path, but not necessarily the shortest. For shortest paths in unweighted graphs, we need BFS.

4.2 Connected Components

DEFINITION (Connected Components Problem). Given an undirected graph G , assign each vertex to a connected component such that two vertices are in the same component if and only if there is a path between them.

DFS provides an elegant linear-time solution. The idea: run DFS from vertex 0 and label all vertices it visits as component 0. Then find the next unvisited vertex and run DFS from it, labeling all newly visited vertices as component 1. Continue until all vertices are visited.

```

Class ConnectedComponents:
    marked[] : boolean array
    id[]      : integer array    // component id for each vertex
    count     : integer         // number of components

    Constructor(G):
        marked <- new boolean[G.V()], all false
        id <- new integer[G.V()]
        count <- 0
        for v from 0 to G.V() - 1:
            if not marked[v]:
                DFS(G, v)
                count <- count + 1

    Procedure DFS(G, v):
        marked[v] <- true
        id[v] <- count
        for each w in G.adj(v):
            if not marked[w]:
                DFS(G, w)

    Function connected(v, w):
        return id[v] = id[w]

    Function componentId(v):
        return id[v]

    Function componentCount():
        return count

```

Example. On the 13-vertex graph from Section 2.7:

```

Component 0: vertices {0, 1, 2, 3, 4, 5, 6}
Component 1: vertices {7, 8}
Component 2: vertices {9, 10, 11, 12}

id[] = [0, 0, 0, 0, 0, 0, 0, 1, 1, 2, 2, 2, 2]
count = 3

```

After preprocessing (which takes $O(V + E)$ time), answering `connected(v, w)` takes $O(1)$ time -- a simple comparison of two array entries. This is a powerful example of the preprocessing paradigm: invest linear time upfront to answer constant-time queries later.

PROPOSITION F. The ConnectedComponents algorithm correctly identifies all connected components of an undirected graph G in $O(V + E)$ time and $O(V)$ space.

Proof. Correctness. By Proposition E, each DFS call visits exactly the vertices reachable from the starting vertex, which is precisely the connected component containing that vertex. The outer loop ensures every vertex is visited exactly once (because we only start a new DFS from an unvisited vertex), and each vertex receives the component id of the DFS call that visits it. Therefore, two vertices receive the same id if and only if they are in the same connected component.

Time. Each vertex is visited exactly once. Each edge is examined exactly twice (once from each endpoint). The outer loop contributes $O(V)$ overhead. Total: $O(V + E)$.

Space. The `marked[]` and `id[]` arrays each use $O(V)$ space. The recursion stack depth is at most V . Total extra space: $O(V)$. *QED.*

4.3 Cycle Detection in Undirected Graphs

DEFINITION (Cycle). An undirected graph has a *cycle* if there exists a path $v_0, v_1, \dots, v_k, v_0$ with $k \geq 2$ and all v_i distinct.

DFS detects cycles by recognizing *back edges*: an edge from the current vertex v to an already-visited vertex w that is not v 's parent in the DFS tree. If such an edge exists, it closes a cycle.

```

Class CycleDetection:
    marked[] : boolean array
    hasCycle : boolean

    Constructor(G):
        marked <- new boolean[G.V()], all false
        hasCycle <- false
        for v from 0 to G.V() - 1:
            if not marked[v]:
                DFS(G, v, -1)    // -1 indicates no parent

    Procedure DFS(G, v, parent):
        marked[v] <- true
        for each w in G.adj(v):
            if not marked[w]:
                DFS(G, w, v)
            else if w != parent:
                hasCycle <- true

```

Why the parent check? In an undirected graph, every edge $\{u, v\}$ appears in both $\text{adj}(u)$ and $\text{adj}(v)$. When DFS visits u and then recurses into v (via edge $\{u, v\}$), it will later encounter u in $\text{adj}(v)$. Without the parent check, DFS would mistake this for a back edge, incorrectly detecting a "cycle" of length 2. The parent check says: "if w is already visited and w is not the vertex we just came from, then we have found a genuine cycle."

Example. Consider DFS on this graph starting from vertex 0:

```

0 -- 1
|    |
3 -- 2

```

```

DFS(0, parent=-1):
  Mark 0. Visit neighbor 1.
  DFS(1, parent=0):
    Mark 1. Visit neighbor 0: marked, but 0 = parent -> skip.
    Visit neighbor 2.
    DFS(2, parent=1):
      Mark 2. Visit neighbor 1: marked, but 1 = parent -> skip.
      Visit neighbor 3.
      DFS(3, parent=2):
        Mark 3. Visit neighbor 0: marked, and 0 != parent (2).
        -> CYCLE DETECTED! (Back edge 3-0)
        Visit neighbor 2: marked -> skip (parent check or already seen).

```

The cycle is 0 -> 1 -> 2 -> 3 -> 0.

PROPOSITION G. An undirected graph is acyclic if and only if DFS discovers no back edge (i.e., no edge to a visited non-parent vertex).

Proof. (If a cycle exists, DFS discovers a back edge.) Consider a cycle C in the graph. Let u be the first vertex of C that DFS visits. DFS will traverse edges of C , eventually reaching a vertex w that has an edge back to u (or to another already-visited vertex on C). This edge is a back edge.

(If DFS discovers a back edge, a cycle exists.) Suppose DFS at vertex v discovers a back edge to vertex w (where $w \neq \text{parent of } v$ and w is already marked). Then w is an ancestor of v in the DFS tree. The tree path from w to v , together with the back edge from v to w , forms a cycle. *QED.*

4.4 Bipartiteness Testing

DEFINITION (Two-Colorability). A graph is *bipartite* (equivalently, *two-colorable*) if we can assign each vertex one of two colors such that no edge connects two vertices of the same color.

We can test bipartiteness with a modified DFS. Assign the source vertex color 0. For each unvisited neighbor, assign the opposite color. If we ever encounter a visited neighbor with the same color as the current vertex, the graph is not bipartite.

```

Class BipartiteTest:
    marked[] : boolean array
    color[] : boolean array    // false = color 0, true = color 1
    isBipartite : boolean

    Constructor(G):
        marked <- new boolean[G.V()], all false
        color <- new boolean[G.V()], all false
        isBipartite <- true
        for v from 0 to G.V() - 1:
            if not marked[v]:
                DFS(G, v)

    Procedure DFS(G, v):
        marked[v] <- true
        for each w in G.adj(v):
            if not marked[w]:
                color[w] <- not color[v]
                DFS(G, w)
            else if color[w] = color[v]:
                isBipartite <- false

```

PROPOSITION H. A graph is bipartite if and only if it contains no odd-length cycle. The BipartiteTest algorithm correctly determines bipartiteness in $O(V + E)$ time.

Proof. (Odd cycle implies not bipartite.) Consider an odd cycle $v_0, v_1, \dots, v_{2k}, v_0$. If the graph were bipartite, the colors would alternate along the cycle. Starting with color 0 for v_0 , we get color 1 for v_1 , color 0 for v_2 , and so on. After $2k$ steps, v_{2k} gets color 0, the same as v_0 . But edge $\{v_{2k}, v_0\}$ connects two vertices of the same color -- contradiction.

(No odd cycle implies bipartite.) DFS assigns colors by alternation along tree edges. If DFS encounters a cross edge to a vertex of the same color, the tree path plus the cross edge form an odd cycle. Contrapositive: if no odd cycle exists, DFS never encounters a same-color neighbor, so the coloring is valid and the graph is bipartite.

The time analysis is identical to Proposition D: $O(V + E)$. *QED.*

Example. Consider this bipartite graph:

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

DFS from 0:

```
color[0] = 0.  Visit 1: color[1] = 1.
Visit 2: color[2] = 1.  Visit 3: color[3] = 0.
Check edge 3-1: color[3]=0 != color[1]=1 -> OK.
-> Bipartite. Partition: {0, 3} and {1, 2}.
```

Now consider this non-bipartite graph (triangle):

```
0 -- 1
 \  /
  2
```

DFS from 0:

```
color[0] = 0.  Visit 1: color[1] = 1.
Visit 2: color[2] = 0.
Check edge 2-0: color[2]=0 = color[0]=0 -> NOT bipartite.
```

Part V: Edge Classification

Estimated time: 15 minutes

5.1 The DFS Forest

When DFS runs on a graph, it implicitly constructs a *DFS forest* -- a collection of DFS trees, one per connected component (in an undirected graph) or one per DFS invocation (in a directed graph). Every edge in the graph falls into one of several categories relative to this forest.

5.2 Edge Types in Undirected Graphs

In an undirected graph, DFS classifies every edge into exactly one of two categories:

DEFINITION (Tree Edge). An edge $\{u, v\}$ is a *tree edge* if DFS discovers v for the first time by traversing this edge from u . Tree edges form the DFS tree (or DFS forest). In the `edgeTo[]` array, a tree edge $\{u, v\}$ is recorded as `edgeTo[v] = u`.

DEFINITION (Back Edge). An edge $\{u, v\}$ is a *back edge* if, when DFS at vertex u examines this edge, v is already marked and v is not the parent of u . In an undirected graph, a back edge connects a vertex to one of its ancestors in the DFS tree.

In undirected graphs, these are the only two types. Every edge is either a tree edge or a back edge. There are no forward edges or cross edges because, if an edge $\{u, v\}$ exists and v is already marked when DFS processes u , then v must be an ancestor of u (otherwise DFS would have discovered the edge from v 's side and made it a tree edge or explored u before reaching v through another path that makes v an ancestor).

PROPOSITION I. In a DFS traversal of an undirected graph, every edge is classified as either a tree edge or a back edge. An undirected graph has a cycle if and only if DFS discovers at least one back edge.

Proof. We have already established the cycle-detection result in Proposition G. For the classification claim: consider any edge $\{u, v\}$. Without loss of generality, assume DFS visits u before v . When `DFS(u)` examines the edge to v :

- If v is not yet marked, DFS recurses into v , and $\{u, v\}$ is a tree edge.
- If v is already marked, then v was visited before u . Since u is currently on the recursion stack (we are in `DFS(u)`), v must be an ancestor of u in the DFS tree. Therefore $\{u, v\}$ is a back edge. *QED.*

5.3 Edge Types in Directed Graphs (Preview)

In a directed graph (digraph), which we study in detail in Lecture 10, DFS classifies edges into four categories:

1. **Tree edges:** Edges in the DFS tree (same as above).
2. **Back edges:** Edges from a vertex to one of its ancestors in the DFS tree. Back edges indicate cycles; a digraph is acyclic (a DAG) if and only if DFS discovers no back edges.
3. **Forward edges:** Edges from a vertex to one of its descendants in the DFS tree (but not a tree edge). These edges are "shortcuts" that skip over intermediate tree edges.
4. **Cross edges:** Edges from a vertex to a vertex in a different subtree of the DFS forest, or to a vertex in a subtree that has already been completely explored. Cross edges connect vertices that have no ancestor-descendant relationship.

To distinguish these four types, DFS uses *discovery times* and *finish times* (or equivalently, vertex colors: white for undiscovered, gray for discovered but not finished, black for finished). We will formalize this in Lecture 10.

5.4 DFS Forest Example

Consider the DFS forest from our trace in Section 3.5:

Graph edges:

0-1, 0-2, 0-5, 1-2, 2-3, 2-4, 3-4, 3-5

DFS Tree (rooted at 0):

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

Tree edges: 0-1, 1-2, 2-3, 3-4, 3-5 (5 edges = $V-1 = 6-1$)

Back edges: {0,2}, {2,4}, {0,5} (3 edges = $E-(V-1) = 8-5$)

Analysis of back edges:

- * Edge {0,2}: when DFS(2) checks neighbor 0, 0 is marked and is an ancestor of 2. Back edge.
- * Edge {2,4}: when DFS(4) checks neighbor 2, 2 is marked and is an ancestor of 4. Back edge.
- * Edge {0,5}: when DFS(5) checks neighbor 0, 0 is marked and is an ancestor of 5. Back edge.

These 3 back edges indicate that the graph has cycles:

Cycle from {0,2}: 0 -> 1 -> 2 -> 0

Cycle from {2,4}: 2 -> 3 -> 4 -> 2

Cycle from {0,5}: 0 -> 1 -> 2 -> 3 -> 5 -> 0

5.5 Counting Edges

PROPOSITION J. In a DFS traversal of a connected undirected graph with V vertices and E edges, there are exactly $V - 1$ tree edges and $E - (V - 1)$ back edges.

Proof. The DFS tree spans all V vertices and is a tree, so it has exactly $V - 1$ edges (Proposition B). Every edge that is not a tree edge is a back edge. Therefore the number of back edges is $E - (V - 1)$. *QED.*

Corollary. A connected graph is a tree if and only if it has no back edges, if and only if $E = V - 1$.

Part VI: Design Pattern -- Graph Processing

Estimated time: 15 minutes

6.1 Decoupling Representation from Processing

A key software engineering insight in Sedgwick's approach to graph algorithms is the strict separation between the graph data structure and the algorithms that process it. The Graph class provides only the representation: vertices, edges, and adjacency queries. All algorithmic processing is done in separate classes that take a Graph object as a constructor argument.

This design has several advantages:

1. **Single Responsibility.** The Graph class has one job: represent the graph. Each processing class has one job: solve a specific problem.
2. **Open/Closed Principle.** We can add new graph algorithms without modifying the Graph class. The Graph class is "closed for modification, open for extension."
3. **Reusability.** The same Graph object can be passed to multiple processing classes. Build the graph once, then run DFS paths, connected components, cycle detection, and bipartiteness testing on it.
4. **Testability.** Each processing class can be tested independently with small, hand-crafted graphs.

6.2 The Constructor-Query Pattern

Sedgewick's graph-processing classes follow a consistent pattern:

1. **Constructor takes graph (and possibly a source vertex).** The constructor runs the algorithm -- DFS, BFS, or whatever -- and stores the results in instance variables.
2. **Query methods answer questions.** After construction, clients call query methods that return answers in constant time (or time proportional to the output size).

Pattern:

```
Class GraphProcessor:
    // instance variables: store results of computation

    Constructor(G, ...):
        // run the algorithm
        // store results

    Function query1(...):
        // answer a question using stored results

    Function query2(...):
        // answer another question
```

6.3 Sedgewick's Graph-Processing Classes

Here are the principal graph-processing classes from Chapter 4.1, all following the constructor-query pattern:

DepthFirstPaths:

```
Constructor: DepthFirstPaths(G, s)
- Runs DFS from source s
- Builds edgeTo[] and marked[] arrays

Queries:
  hasPathTo(v) : boolean    // is v reachable from s?
  pathTo(v)    : iterable   // path from s to v (or null)
```

CC (Connected Components):

```
Constructor: CC(G)
- Runs DFS from each unvisited vertex
- Assigns component id to each vertex

Queries:
  connected(v, w) : boolean  // are v and w in same component?
  id(v)           : integer  // component identifier for v
  count()        : integer  // number of components
```

Cycle:

```
Constructor: Cycle(G)
- Runs DFS with parent tracking
- Determines if graph has a cycle

Queries:
  hasCycle() : boolean      // does the graph have a cycle?
  cycle()    : iterable     // vertices of a cycle (if any)
```

Bipartite:

```
Constructor: Bipartite(G)
  - Runs DFS with two-coloring
  - Determines if graph is bipartite

Queries:
  isBipartite() : boolean    // is the graph bipartite?
  color(v)      : boolean    // which side of bipartition?
```

6.4 Client Code Example

The beauty of this design is visible in client code:

```
Procedure AnalyzeGraph(G):
  // Find connected components
  cc <- new CC(G)
  print "Number of components: " + cc.count()

  // Find paths from vertex 0
  paths <- new DepthFirstPaths(G, 0)
  for v from 0 to G.V() - 1:
    if paths.hasPathTo(v):
      print "0 to " + v + ": " + paths.pathTo(v)

  // Check for cycles
  cyc <- new Cycle(G)
  if cyc.hasCycle():
    print "Graph has a cycle: " + cyc.cycle()

  // Check bipartiteness
  bip <- new Bipartite(G)
  if bip.isBipartite():
    print "Graph is bipartite"
  else:
    print "Graph is NOT bipartite"
```

Each processing class preprocesses the graph in $O(V + E)$ time. After that, queries run in $O(1)$ time (or $O(\text{path length})$ for `pathTo`). The total cost is $O(V + E)$ for each processing class, which is optimal since we must at least read the entire graph.

6.5 Extensibility

This design pattern makes it straightforward to add new graph-processing algorithms. To add articulation point detection, for example, we simply create a new class:

```
Class ArticulationPoints:
    Constructor(G):
        // run modified DFS
        // identify articulation points

    Function isArticulationPoint(v) : boolean
    Function articulationPoints() : iterable
```

No changes to the Graph class are needed. The new class simply takes a Graph as input and answers queries about articulation points. This is the power of decoupling representation from processing.

Part VII: Advanced DFS Concepts

Estimated time: 15 minutes

7.1 DFS Timestamps

A useful augmentation of DFS assigns two timestamps to each vertex: a *discovery time* (when the vertex is first visited) and a *finish time* (when DFS backtracks from the vertex, having explored all of its descendants).

```

Procedure DFS_WithTimestamps(G, v):
    time <- time + 1
    discovery[v] <- time
    marked[v] <- true
    for each w in G.adj(v):
        if not marked[w]:
            edgeTo[w] <- v
            DFS_WithTimestamps(G, w)
    time <- time + 1
    finish[v] <- time

```

PROPOSITION K (Parenthesis Theorem). In a DFS forest, vertex u is an ancestor of vertex v if and only if the interval $[discovery[u], finish[u]]$ contains the interval $[discovery[v], finish[v]]$.

Proof. If u is an ancestor of v , then DFS discovers u before v (so $discovery[u] < discovery[v]$) and finishes v before u (so $finish[v] < finish[u]$). Therefore $[discovery[v], finish[v]]$ is contained in $[discovery[u], finish[u]]$.

Conversely, if the interval containment holds, then u is discovered before v and finished after v , meaning v is discovered and finished during the DFS call to u . This means v is a descendant of u in the DFS tree. *QED.*

The name "parenthesis theorem" comes from the analogy with nested parentheses: if we write "(" at discovery time and ")" at finish time, the parentheses are always properly nested.

Example with a path graph 0-1-2-3:

DFS from 0:

Vertex:	0	1	2	3
Discovery:	1	2	3	4
Finish:	8	7	6	5

Parenthesis view:

0:	()
1:	()
2:	()	
3:	()		
	1	2	3	4	5 6 7 8

7.2 Vertex States During DFS

At any point during a DFS traversal, each vertex is in one of three states:

- **Undiscovered (WHITE):** The vertex has not yet been encountered. It has no discovery time.
- **Discovered (GRAY):** The vertex has been discovered but DFS has not finished exploring all of its descendants. It is on the recursion stack. It has a discovery time but no finish time yet.
- **Finished (BLACK):** DFS has finished exploring the vertex and all of its descendants. It has both a discovery time and a finish time.

These states are particularly useful for edge classification in directed graphs:

- If DFS at vertex u examines an edge to a WHITE vertex w : tree edge.
- If DFS at vertex u examines an edge to a GRAY vertex w : back edge (w is an ancestor of u on the current DFS path).
- If DFS at vertex u examines an edge to a BLACK vertex w : forward or cross edge (digraphs only).

7.3 DFS and Stack Frames

Understanding DFS requires understanding the relationship between the algorithm and the call stack (or explicit stack). Each recursive call to $\text{DFS}(v)$ creates a stack frame that records:

1. The current vertex v .
2. The current position in v 's adjacency list (which neighbor are we about to examine?).
3. Local variables.

The DFS recursion stack at any point during execution represents the current path from the source to the vertex being explored. This is a key property: the set of GRAY vertices at any moment forms a path from the root of the current DFS tree to the vertex currently being processed.

This property is what makes cycle detection work: if we encounter a GRAY vertex while exploring, we have found a path from that vertex to the current vertex (via the recursion stack), and the edge we just examined closes the path into a cycle.

7.4 Comparison with Breadth-First Search

While a full treatment of BFS belongs in Lecture 10, it is instructive to contrast DFS and BFS at a high level now, since both are graph traversals that visit every reachable vertex in $O(V + E)$ time.

Property	DFS	BFS
Data structure	Stack (implicit or explicit)	Queue
Exploration order	Deep before wide	Wide before deep
Path quality	Some path (not shortest)	Shortest path (unweighted)
Space (worst case)	$O(V)$ (stack depth)	$O(V)$ (queue width)
Applications	Cycles, components, topo	Shortest paths, layers
Edge classification	2 types (undirected)	2 types (undirected)

Both algorithms are linear-time, both visit every reachable vertex exactly once, and both build a spanning tree of the connected component. The fundamental difference is which vertex to explore next: DFS picks the most recently discovered vertex (LIFO), while BFS picks the least recently discovered vertex (FIFO). This simple difference leads to dramatically different tree structures and different algorithmic applications.

Practice Problems

Problem 1: DFS Trace

Problem. Trace DFS on the following graph starting from vertex A. Show the marked array and edgeTo array after each vertex is visited. List the vertices in the order they are visited.

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

Adjacency lists (alphabetical order within each list):

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

Solution.

We use vertex indices: A=0, B=1, C=2, D=3, E=4, F=5.

DFS(A):

Mark A. marked = [T,F,F,F,F,F] edgeTo = [-,-,-,-,-,-]

Neighbor B: not marked. edgeTo[B]=A. Recurse.

DFS(B):

Mark B. marked = [T,T,F,F,F,F] edgeTo = [-,A,-,-,-,-]

Neighbor A: marked -> skip.

Neighbor D: not marked. edgeTo[D]=B. Recurse.

DFS(D):

Mark D. marked = [T,T,F,T,F,F] edgeTo = [-,A,-,B,-,-]

Neighbor B: marked -> skip.

Neighbor C: not marked. edgeTo[C]=D. Recurse.

DFS(C):

Mark C. marked = [T,T,T,T,F,F] edgeTo = [-,A,D,B,-,-]

Neighbor A: marked -> skip.

Neighbor D: marked -> skip.

Return from DFS(C).

Neighbor F: not marked. edgeTo[F]=D. Recurse.

DFS(F):

Mark F. marked = [T,T,T,T,F,T] edgeTo = [-,A,D,B,-,D]

Neighbor D: marked -> skip.

Return from DFS(F).

Return from DFS(D).

Neighbor E: not marked. edgeTo[E]=B. Recurse.

DFS(E):

Mark E. marked = [T,T,T,T,T,T] edgeTo = [-,A,D,B,B,D]

Neighbor B: marked -> skip.

Return from DFS(E).

Return from DFS(B).

Neighbor C: marked -> skip.

```
Return from DFS(A).
```

```
Visit order: A, B, D, C, F, E
```

```
DFS Tree:
```

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

Verification: The tree has 5 edges ($V - 1 = 6 - 1 = 5$). The original graph has 6 edges. Therefore there is $6 - 5 = 1$ back edge. That back edge is $\{A, C\}$ (or equivalently $\{C, A\}$): when $\text{DFS}(C)$ checks neighbor A, A is already marked and A is not the parent of C (D is). This confirms the graph has exactly one cycle: $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$.

Problem 2: Connected Components

Problem. Find all connected components of the following graph with 10 vertices.

```
Edges: {0,1}, {0,2}, {1,2}, {3,4}, {5,6}, {5,7}, {6,7}, {8,9}
```

Solution.

Run DFS from vertex 0: visits 0, 1, 2. Label these as component 0.

Next unvisited vertex: 3. Run DFS from 3: visits 3, 4. Label as component 1.

Next unvisited vertex: 5. Run DFS from 5: visits 5, 6, 7. Label as component 2.

Next unvisited vertex: 8. Run DFS from 8: visits 8, 9. Label as component 3.

```
Component 0: {0, 1, 2}    (triangle)
Component 1: {3, 4}      (single edge)
Component 2: {5, 6, 7}   (triangle)
Component 3: {8, 9}      (single edge)

id[] = [0, 0, 0, 1, 1, 2, 2, 2, 3, 3]
count = 4

Queries:
  connected(0, 2) = true   (both id 0)
  connected(3, 7) = false  (id 1 vs id 2)
  connected(6, 5) = true   (both id 2)
```

Problem 3: Cycle Detection

Problem. Determine whether the following graph has a cycle. If so, identify it.

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

Adjacency lists:
0: [1]
1: [0, 2]
2: [1, 3]
3: [2, 4]
4: [3, 5]
5: [4]
```

Solution.

This graph is a simple path: 0-1-2-3-4-5. It has no cycles.

DFS trace (with parent tracking):

```

DFS(0, parent=-1): Mark 0.
  Neighbor 1: not marked. DFS(1, parent=0).
    DFS(1, parent=0): Mark 1.
      Neighbor 0: marked, but 0 = parent -> skip (not a back edge).
      Neighbor 2: not marked. DFS(2, parent=1).
        DFS(2, parent=1): Mark 2.
          Neighbor 1: marked, but 1 = parent -> skip.
          Neighbor 3: not marked. DFS(3, parent=2).
            DFS(3, parent=2): Mark 3.
              Neighbor 2: marked, but 2 = parent -> skip.
              Neighbor 4: not marked. DFS(4, parent=3).
                DFS(4, parent=3): Mark 4.
                  Neighbor 3: marked, but 3 = parent -> skip.
                  Neighbor 5: not marked. DFS(5, parent=4).
                    DFS(5, parent=4): Mark 5.
                      Neighbor 4: marked, but 4 = parent -> skip.
                      Return.
                  Return.
                Return.
              Return.
            Return.
          Return.
        Return.
      Return.
    Return.
  Return.

No back edge encountered -> NO CYCLE.

```

Now consider adding edge {0, 5}:

```

DFS would proceed the same until DFS(5, parent=4):
  Neighbor 4: marked, parent -> skip.
  Neighbor 0: marked, and 0 != parent (4) -> BACK EDGE! CYCLE DETECTED.
  Cycle: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 0

```

Problem 4: Proof -- DFS Visits All Reachable Vertices

Problem. Prove that DFS starting from vertex s visits every vertex reachable from s .

Solution.

Claim. If there exists a path from s to v in graph G , then $\text{DFS}(G, s)$ sets $\text{marked}[v] = \text{true}$.

Proof by strong induction on the length of the shortest path from s to v .

Base case (distance 0): The only vertex at distance 0 from s is s itself. $\text{DFS}(G, s)$ sets $\text{marked}[s] = \text{true}$ as its first action. The claim holds.

Inductive hypothesis: Assume that for all vertices u at distance at most k from s , $\text{DFS}(G, s)$ sets $\text{marked}[u] = \text{true}$.

Inductive step: Consider a vertex v at distance $k + 1$ from s . Let $s = v_0, v_1, \dots, v_k, v_{k+1} = v$ be a shortest path from s to v . Then v_k is at distance k from s , so by the inductive hypothesis, DFS marks v_k .

When $\text{DFS}(G, v_k)$ executes, it iterates over all neighbors of v_k . Vertex $v = v_{k+1}$ is a neighbor of v_k . Either v is already marked (and we are done) or v is not marked, in which case DFS recurses into v and marks it. In either case, $\text{marked}[v] = \text{true}$. *QED.*

Corollary. The set of vertices marked by $\text{DFS}(G, s)$ is exactly the connected component of G containing s .

Proof. DFS marks every vertex reachable from s (by the claim above). DFS marks only reachable vertices (because DFS only visits a vertex by following an edge from an already-visited vertex, and by induction every visited vertex is reachable from s). The set of vertices reachable from s is precisely the connected component containing s . *QED.*

Problem 5: Draw the DFS Tree

Problem. Draw the DFS tree for the following graph when DFS starts from vertex 0. List all tree edges and back edges.

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

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

Adjacency lists (sorted):

0: [1, 3, 4]

1: [0, 2]

2: [1, 3]

3: [0, 2, 4]

4: [0, 3]

Solution.

```

DFS(0): Mark 0.
Visit 1: not marked. edgeTo[1]=0. Tree edge {0,1}.
DFS(1): Mark 1.
Visit 0: marked (parent) -> skip.
Visit 2: not marked. edgeTo[2]=1. Tree edge {1,2}.
DFS(2): Mark 2.
Visit 1: marked (parent) -> skip.
Visit 3: not marked. edgeTo[3]=2. Tree edge {2,3}.
DFS(3): Mark 3.
Visit 0: marked, not parent (parent=2). BACK EDGE {3,0}.
Visit 2: marked (parent) -> skip.
Visit 4: not marked. edgeTo[4]=3. Tree edge {3,4}.
DFS(4): Mark 4.
Visit 0: marked, not parent (parent=3). BACK EDGE {4,0}.
Visit 3: marked (parent) -> skip.
Return.
Return.
Return.
Return.
Visit 3: marked -> skip.
Visit 4: marked -> skip.
Return.

```

DFS Tree:

```

0
|
1
|
2
|
3
|
4

```

Tree edges: {0,1}, {1,2}, {2,3}, {3,4} (4 edges = $V-1 = 5-1$)

Back edges: {3,0}, {4,0} (2 edges = $E - (V-1) = 6-4$)

Cycles revealed by back edges:

```
{3,0}: 0 -> 1 -> 2 -> 3 -> 0
{4,0}: 0 -> 1 -> 2 -> 3 -> 4 -> 0
```

Problem 6: Bipartiteness Testing

Problem. Determine whether the following graph is bipartite. If so, give a valid two-coloring. If not, identify an odd cycle.

Graph 1:

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

Graph 2:

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

Solution.

Graph 1:

Adjacency lists:

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

DFS(0): color[0] = 0.

Visit 1: color[1] = 1.

DFS(1):

Visit 0: marked. color[0]=0 != color[1]=1 -> OK.

Visit 2: color[2] = 0.

DFS(2):

Visit 1: marked. color[1]=1 != color[2]=0 -> OK.

Visit 3: color[3] = 1.

DFS(3):

Visit 0: marked. color[0]=0 != color[3]=1 -> OK.

Visit 2: marked. color[2]=0 != color[3]=1 -> OK.

Graph 1 IS bipartite.

Partition: {0, 2} (color 0) and {1, 3} (color 1).

This makes sense: Graph 1 is a 4-cycle, which is an even cycle.

Graph 2:

Adjacency lists:

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

3: [2, 4, 5] 4: [0, 3] 5: [3]

DFS(0): color[0] = 0.

Visit 1: color[1] = 1. DFS(1).

Visit 0: marked, different color -> OK.

Visit 2: color[2] = 0. DFS(2).

Visit 1: marked, different color -> OK.

Visit 3: color[3] = 1. DFS(3).

Visit 2: marked, different color -> OK.

Visit 4: color[4] = 0. DFS(4).

Visit 0: marked. color[0]=0 = color[4]=0 -> SAME COLOR!

NOT BIPARTITE!

Odd cycle: 0 -> 1 -> 2 -> 3 -> 4 -> 0 (length 5, which is odd).

Problem 7: Algorithm Design

Problem. Given an undirected graph G and two vertices s and t , design an $O(V + E)$ algorithm to determine if there is exactly one simple path from s to t .

Solution.

Key insight: In an undirected connected graph, there is exactly one simple path between every pair of vertices if and only if the graph is a tree (connected and acyclic). More generally, within a connected component, there is exactly one simple path between two vertices if and only if the component contains no cycle.

Algorithm:

1. Run DFS from s to find all vertices reachable from s (the connected component of s). Time: $O(V + E)$.
2. Check if t is in this component. If not, there is no path (and hence not exactly one). Time: $O(1)$.
3. Check if the component is acyclic (a tree). Use the cycle detection algorithm from Section 4.3, restricted to this component. Time: $O(V + E)$.
4. If the component is acyclic and t is reachable, there is exactly one simple path from s to t .

Alternative approach (even simpler): Count the number of vertices (V_c) and edges (E_c) in the component. The component is a tree if and only if $E_c = V_c - 1$. A single DFS from s counts both.

```

Procedure HasExactlyOnePath(G, s, t):
    Run DFS from s.
    If not marked[t]:
        return false    // no path at all

    // Count vertices and edges in s's component
    Vc <- number of marked vertices
    Ec <- 0
    for each marked vertex v:
        for each w in G.adj(v):
            if marked[w]:
                Ec <- Ec + 1
    Ec <- Ec / 2    // each edge counted twice

    return (Ec = Vc - 1)

```

Correctness argument: If the component containing s and t has no cycle ($E_c = V_c - 1$), then it is a tree, and every pair of vertices in a tree has exactly one simple path between them. If the component has a cycle, then there exist vertices with multiple simple paths between them, and in particular there are multiple simple paths between s and t (or at least between some pair -- but we can show that if s and t are in the same component with a cycle, there are at least two simple paths between s and t if the component is 2-connected, though this is not always the case). Actually, the presence of a cycle in the component does not guarantee multiple paths between s and t specifically. However, the problem asks for a sufficient condition, and the tree condition is both necessary and sufficient for *every* pair in the component to have exactly one path.

Refined answer: Exactly one simple path exists between s and t if and only if (1) s and t are in the same connected component, and (2) removing any single edge on the unique tree path between s and t disconnects them (i.e., the component restricted to the s - t path has no cycle). The simplest correct $O(V+E)$ approach: run DFS from s , check that t is reachable, and check that the component is acyclic.

Problem 8: Iterative DFS Correctness

Problem. The following iterative DFS implementation has a subtle bug. Identify it and explain how to fix it.

```

Procedure BuggyDFS(G, s):
    stack <- empty stack
    push s onto stack
    marked[s] <- true

    while stack is not empty:
        v <- pop from stack
        for each w in G.adj(v):
            if not marked[w]:
                marked[w] <- true
                edgeTo[w] <- v
                push w onto stack

```

Solution.

This implementation is actually *correct* for the purpose of visiting all reachable vertices and building a valid spanning tree. It marks vertices when they are pushed (not when they are popped), which prevents any vertex from being pushed more than once. Each vertex is popped exactly once, and its neighbors are examined exactly once. The algorithm runs in $O(V + E)$ time.

However, this implementation does NOT produce the same traversal order as recursive DFS. Consider a vertex v with neighbors w_1, w_2, w_3 (in adjacency list order). Recursive DFS would fully explore the subtree rooted at w_1 before even looking at w_2 . But this iterative version pushes w_1, w_2 , and w_3 onto the stack simultaneously. Then it pops w_3 (LIFO), explores w_3 's subtree, then pops w_2 , and so on. The result is that neighbors are explored in reverse adjacency-list order, and all neighbors are "committed to" at the same level before descending.

The truly buggy version would be one that marks on pop instead of push, without guarding against duplicates on the stack:

```

Procedure TrulyBuggyDFS(G, s):
    stack <- empty stack
    push s onto stack

    while stack is not empty:
        v <- pop from stack
        if not marked[v]:
            marked[v] <- true
            for each w in G.adj(v):
                if not marked[w]:
                    edgeTo[w] <- v    // BUG: may overwrite!
                    push w onto stack

```

In this version, `edgeTo[w]` can be overwritten multiple times because `w` can be pushed by multiple vertices before it is popped and marked. The last vertex to push `w` overwrites `edgeTo[w]`, potentially producing an invalid or unexpected tree.

Fix: Either (a) mark vertices when pushed (as in the first version), which prevents duplicate pushes entirely, or (b) keep the mark-on-pop approach but accept that `edgeTo[]` may reflect a different (but still valid) DFS tree, or (c) add a separate `onStack[]` boolean array:

```

Procedure CorrectIterativeDFS(G, s):
    stack <- empty stack
    push s onto stack
    onStack[s] <- true
    edgeTo[s] <- -1

    while stack is not empty:
        v <- pop from stack
        marked[v] <- true
        for each w in G.adj(v):
            if not marked[w] and not onStack[w]:
                edgeTo[w] <- v
                onStack[w] <- true
                push w onto stack

```

This ensures each vertex is pushed at most once, marked when popped, and `edgeTo[]` is set exactly once per vertex.

Summary

Key Takeaways

1. **Graphs are universal.** Graphs model pairwise relationships and appear in virtually every domain of computing: social networks, the web, maps, scheduling, compilers, biology, and beyond. Mastering graph algorithms is essential for any practicing computer scientist or software engineer.
2. **Two representations dominate.** The adjacency matrix ($O(V^2)$ space, $O(1)$ edge query) is suitable for dense graphs; the adjacency list ($O(V + E)$ space, $O(\text{degree})$ edge query) is suitable for sparse graphs. Since most real-world graphs are sparse, the adjacency list is the standard choice and is used throughout Sedgewick's textbook.
3. **DFS is foundational.** Depth-first search is a simple yet extraordinarily powerful graph traversal. From a single linear-time pass, DFS can extract paths, connected components, cycles, bipartite structure, and more. Understanding DFS -- both its recursive and iterative formulations, its marked/edgeTo arrays, and its edge classification -- is prerequisite to nearly every advanced graph algorithm.
4. **DFS runs in $O(V + E)$ time.** This is optimal for any algorithm that must examine the entire graph. The proof relies on the handshaking lemma and the fact that each vertex is processed exactly once.
5. **Edge classification reveals structure.** In undirected graphs, DFS produces tree edges and back edges. Back edges indicate cycles. In directed graphs (studied in Lecture 10), two additional types appear: forward edges and cross edges. The classification of edges by DFS is the key to many advanced algorithms, including Tarjan's strongly connected components algorithm.
6. **Decouple representation from processing.** Sedgewick's design pattern -- separate Graph class with processing delegated to classes like DepthFirstPaths, CC, and Cycle -- is a model of clean software design. The Graph class is reusable, the processing classes are independently testable, and new algorithms can be added without modifying the Graph class.
7. **DFS trees encode structural information.** The edgeTo[] (parent-link) array encodes a spanning tree of the connected component, enabling path reconstruction. The discovery and finish timestamps enable ancestor/descendant queries and are essential for edge classification in directed graphs.

8. **Cycle detection uses back edges.** An undirected graph has a cycle if and only if DFS encounters a back edge (an edge to a visited vertex that is not the current vertex's parent). This simple observation, combined with the linear-time guarantee of DFS, gives an optimal cycle-detection algorithm.

Looking Ahead

In **Lecture 10**, we extend our study to directed graphs (digraphs) and breadth-first search (BFS). BFS finds shortest paths in unweighted graphs -- a property DFS lacks. We will also study topological sort (ordering vertices of a DAG so that all edges point forward) and strongly connected components (maximal sets of mutually reachable vertices in a digraph), both of which are DFS applications.

In **Lecture 11**, we study weighted graphs and the minimum spanning tree problem, solved by Kruskal's and Prim's algorithms. In **Lecture 12**, we address shortest paths in weighted graphs with Dijkstra's and Bellman-Ford algorithms.

Companion C++ Implementation

The companion C++ file for this lecture implements the following classes and functions, mirroring the pseudocode presented here:

- `GraphAdjList` -- Adjacency list representation with `addEdge()`, `adj()`, `V()`, `E()`.
- `GraphAdjMatrix` -- Adjacency matrix representation with the same interface.
- `dfsRecursive(G, s, marked, edgeTo)` -- Recursive DFS with path tracking.
- `dfsIterative(G, s, marked, edgeTo)` -- Iterative DFS using an explicit stack with reverse neighbor push.
- `hasCycleDFS(G)` -- Cycle detection via DFS with parent tracking.
- `connectedComponents(G)` -- Connected component identification using DFS.

References

1. Sedgewick, R. and Wayne, K. *Algorithms*, 4th Edition. Addison-Wesley, 2011. Chapter 4.1: Undirected 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--22.3: Representations of Graphs, Breadth-First Search, Depth-First Search.
3. Tarjan, R.E. "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing*, 1(2):146--160, 1972.
4. Hopcroft, J.E. and Tarjan, R.E. "Efficient Algorithms for Graph Manipulation." *Communications of the ACM*, 16(6):372--378, 1973.
5. Kleinberg, J. and Tardos, E. *Algorithm Design*. Addison-Wesley, 2005. Chapter 3: Graphs.
6. Even, S. *Graph Algorithms*, 2nd Edition. Cambridge University Press, 2012.
7. Skiena, S.S. *The Algorithm Design Manual*, 2nd Edition. Springer, 2008. Chapter 5: Graph Traversal.
8. Diestel, R. *Graph Theory*, 5th Edition. Springer, 2017. Chapters 1--2.
9. Sedgewick, R. and Wayne, K. "Algorithms, Part II." Coursera, Princeton University. Available at: <https://www.coursera.org/learn/algorithms-part2>
10. Tremaux, C.P. Unpublished manuscript on maze-solving, c. 1882. Discussed in Even (2012) and Sedgewick (2011).

Lecture 9 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition