Graphs

Representations and Traversals

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

What is a Graph?

Graph: Set of vertices (nodes) connected by edges

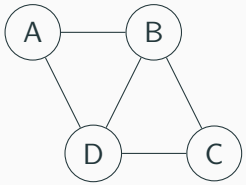
Notation: $G = (V, E)$

- V = set of vertices
- E = set of edges

Example: Social network

- Vertices = people
- Edges = friendships

Simple graph example:



$V = \{A, B, C, D\}$

$E =$

$\{(A, B), (A, D), (B, C), (B, D), (C, D)\}$

Types of Graphs

Undirected vs Directed:

Undirected: Edge has no direction

Friendship: A $\leftrightarrow$ B

Directed (Digraph): Edge has direction

Following: A $\rightarrow$ B

Unweighted vs Weighted:

Unweighted: All edges equal

Weighted: Edges have values

Distance, cost, capacity

Graph Terminology

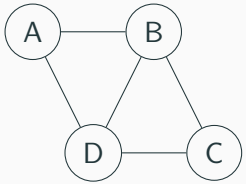
Degree: Number of edges connected to a vertex

Path: Sequence of vertices connected by edges

Cycle: Path that starts and ends at same vertex

Connected: Path exists between any two vertices

Acyclic: No cycles (e.g., tree is acyclic graph)



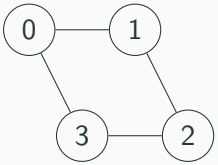
Degree of B: 3

Path A→C: A-B-C or A-D-C

Cycle: B-D-C-B

Representation 1: Adjacency Matrix

2D array: matrix[i][j] = 1 if edge exists



	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

Properties:

- Space: $O(V^2)$
- Check edge: $O(1)$
- Get neighbors: $O(V)$
- Add edge: $O(1)$

Good for:

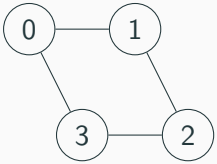
- Dense graphs (many edges)
- Fast edge lookup
- Matrix operations

Bad for:

- Sparse graphs (wastes space)
- Finding all neighbors

Representation 2: Adjacency List

Array of lists: Each vertex has list of neighbors




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

Properties:

- Space: $O(V + E)$
- Check edge: $O(degree)$
- Get neighbors: $O(degree)$
- Add edge: $O(1)$

Good for:

- Sparse graphs
- Iterating neighbors
- Space efficiency

Bad for:

- Fast edge lookup

Most common choice!

Adjacency Matrix vs List

Operation	Matrix	List
Space	$O(V^2)$	$O(V+E)$
Check edge (u,v)	$O(1)$	$O(degree(u))$
Get all neighbors of u	$O(V)$	$O(degree(u))$
Add edge	$O(1)$	$O(1)$
Add vertex	$O(V^2)$	$O(1)$

Rule of thumb:

- **Dense graph** ($|E| \approx |V|^2$): Use matrix
- **Sparse graph** ($|E| \ll |V|^2$): Use adjacency list
- **Most real-world graphs are sparse** $\rightarrow$ adjacency list

Graph Traversal

Goal: Visit every vertex in the graph systematically

Two main algorithms:

Breadth-First Search (BFS)

- Visit level by level
- Uses a **queue**
- Finds shortest path (unweighted)
- Like ripples in water

Applications:

- Shortest path
- Web crawling
- Social network distance

Depth-First Search (DFS)

- Go deep before wide
- Uses a **stack** (or recursion)
- Explores one path fully
- Like maze solving

Applications:

- Cycle detection
- Topological sort
- Connected components

Both: $O(V + E)$ time complexity

Breadth-First Search (BFS): The Idea

Strategy: Explore all neighbors at distance d before exploring distance $d + 1$

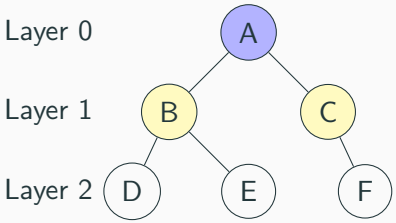
Algorithm:

1. Start at source vertex
2. Mark it visited
3. Add to queue
4. While queue not empty:
   - Dequeue vertex
   - Visit unvisited neighbors
   - Mark them, add to queue

Key: Use **queue** (FIFO)

Visual intuition:

Start at A, explore by layers:



Order: A $\rightarrow$ B,C $\rightarrow$ D,E,F

BFS Algorithm

```
void BFS(Graph& g, int start) {
    vector<bool> visited(g.V, false);
    queue<int> q;

    // Start from source
    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int v = q.front();
        q.pop();

        cout << v << " "; // Process vertex

        // Visit all neighbors
        for (int neighbor : g.adj[v]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

Time Complexity:

- Each vertex visited once: $O(V)$
- Each edge examined once: $O(E)$
- **Total: O(V + E)**

Space:

- Visited array: $O(V)$
- Queue: $O(V)$ worst case
- **Total: O(V)**

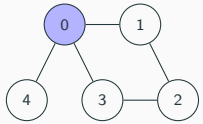
Key insight:

Queue ensures we visit vertices in order of distance from start

BFS Example: Step by Step

Start at vertex 0:

Step 1: Visit 0



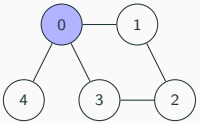
Queue: [] → [0] → []

Visited: 0

BFS Example: Step by Step

Start at vertex 0:

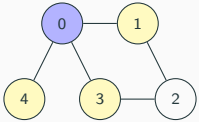
Step 1: Visit 0



Queue: [] → [0] → []

Visited: 0

Step 2: Visit neighbors of 0



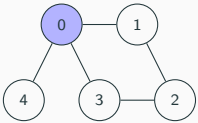
Queue: [1,3,4]

Visited: 0,1,3,4

BFS Example: Step by Step

Start at vertex 0:

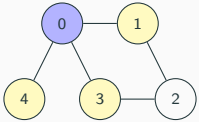
Step 1: Visit 0



Queue: [] → [0] → []

Visited: 0

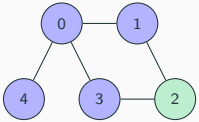
Step 2: Visit neighbors of 0



Queue: [1,3,4]

Visited: 0,1,3,4

Step 3: Visit 2 from 1 or 3



Queue: []

Order: 0,1,3,4,2

BFS Application: Shortest Path (Unweighted)

BFS finds shortest path in unweighted graphs!

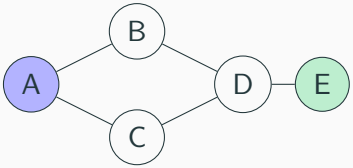
Modified BFS:

```
vector<int> shortestPath(Graph& g, int start) {
    vector<int> dist(g.V, -1);
    vector<int> parent(g.V, -1);
    queue<int> q;

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

    while (!q.empty()) {
        int v = q.front();
        q.pop();

        for (int u : g.adj[v]) {
            if (dist[u] == -1) {
                dist[u] = dist[v] + 1;
                parent[u] = v;
                q.push(u);
            }
        }
    }

    return dist; // or reconstruct path
}
```

Why BFS works:

- Visits vertices in order of distance
- First time we reach a vertex = shortest path
- Each edge has weight 1

Example: Shortest path from A to E



Path: A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E (distance 3)

Depth-First Search (DFS): The Idea

Strategy: Go as deep as possible, then backtrack

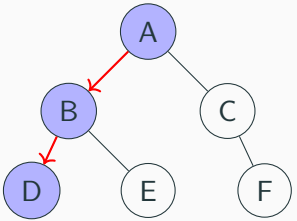
Algorithm:

1. Start at source vertex
2. Mark it visited
3. For each unvisited neighbor:
   - Recursively visit it (go deep!)
4. Backtrack when no unvisited neighbors

Key: Use **stack** (recursion = implicit stack)

Visual intuition:

Start at A, go deep first:



Order: A → B → D (deep first!)
Then backtrack to explore E, C, F

DFS Algorithm (Recursive)

```
void DFS(Graph& g, int v, vector<bool>& visited) {
    // Mark current vertex as visited
    visited[v] = true;
    cout << v << " "; // Process vertex

    // Recur for all unvisited neighbors
    for (int u : g.adj[v]) {
        if (!visited[u]) {
            DFS(g, u, visited);
        }
    }
}

// Wrapper function
void DFSTraversal(Graph& g, int start) {
    vector<bool> visited(g.V, false);
    DFS(g, start, visited);
}
```

Recursion provides implicit stack!

Time Complexity:

- Each vertex visited once: $O(V)$
- Each edge examined once: $O(E)$
- **Total: O(V + E)**

Space:

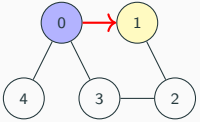
- Visited array: $O(V)$
- Recursion stack: $O(V)$ worst
- **Total: O(V)**

Iterative version:
Use explicit stack instead of recursion (same complexity)

DFS Example: Step by Step

Start at vertex 0:

Step 1: Visit 0, recur on 1



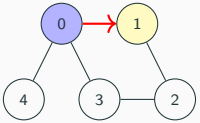
Visited: 0

Going to: 1

DFS Example: Step by Step

Start at vertex 0:

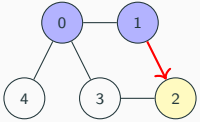
Step 1: Visit 0, recur on 1



Visited: 0

Going to: 1

Step 2: From 1, recur on 2



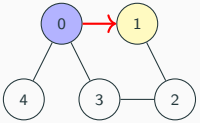
Visited: 0,1

Going to: 2

DFS Example: Step by Step

Start at vertex 0:

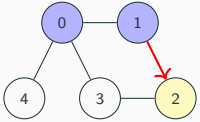
Step 1: Visit 0, recur on 1



Visited: 0

Going to: 1

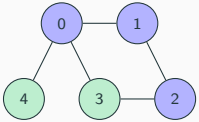
Step 2: From 1, recur on 2



Visited: 0,1

Going to: 2

Step 3: Backtrack, visit 3, 4



Order: 0,1,2,3,4

(DFS goes deep!)

BFS vs DFS Comparison

	BFS	DFS
Data structure	Queue	Stack (recursion)
Strategy	Level by level	Deep first
Time	$O(V + E)$	$O(V + E)$
Space	$O(V)$	$O(V)$
Shortest path	Yes (unweighted)	No
Memory use	More (stores level)	Less (just path)
Implementation	Iterative	Recursive or iterative

When to use BFS:

- Shortest path (unweighted)
- Level-order traversal
- Testing connectivity

When to use DFS:

- Detecting cycles
- Topological sorting
- Connected components
- Maze solving

Motivation: Why Search?

Many problems can be framed as search:

Classic Examples:

- Route finding (GPS)
- Puzzle solving (8-puzzle, Rubik's cube)
- Game playing (chess, checkers)
- Robot path planning

The Pattern:

- Start state
- Goal state
- Set of actions
- Find sequence of actions

Key Insight: Search is about exploring possibilities systematically

Problem Formulation

Components of a search problem

A search problem consists of:

1. **Initial state**: Where we start
2. **Actions**: What we can do in each state
3. **Transition model**: What each action does
4. **Goal test**: Are we done?
5. **Path cost**: Cost of a sequence of actions

Solution: A sequence of actions from initial state to goal

Optimal solution: Solution with lowest path cost

Interactive Visualization: Search Algorithms

Watch the algorithms explore

https://qiao.github.io/PathFinding.js/visual/

What you'll see:

- Maze pathfinding with multiple algorithms
- Visual exploration: blue = exploring, yellow = visited
- Compare BFS, A*, Dijkstra, Best-First, and more

Try this:

1. Click and drag to draw walls (create a maze)
2. Select "Breadth-First Search" from the dropdown
3. Click "Start Search" and watch it explore evenly
4. Try "A* Search" - notice how it's more focused toward the goal

Great for comparing BFS with informed search algorithms!

Example: Maze Solving - Concrete Representation

Problem Components:

Initial State:

- Position (row, col): (0, 0)

Actions:

- {UP, DOWN, LEFT, RIGHT}

Transition Model:

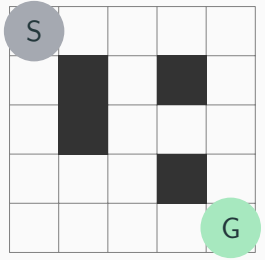
- UP: (r, c) $\rightarrow$ (r-1, c)
- Must be valid: in bounds, not wall

Goal Test:

- Current position == goal position

Path Cost:

- 1 per move



How to represent in code:

- **Grid:** 2D array (0=open, 1=wall)
- **State:** (row, col) tuple
- **Check valid:** grid[r][c] == 0

Interactive Visualization: DFS vs BFS

Watch the algorithms explore

`https://visualgo.net/en/dfsbfs`

What you'll see:

- Graph nodes colored as they're explored
- Step-by-step execution of each algorithm
- Clear difference between DFS (goes deep) and BFS (explores level by level)

Try this:

1. Select "BFS" from the menu and click "Go"
2. Watch how it explores all neighbors before going deeper
3. Now select "DFS" and click "Go"
4. Notice how it follows one path all the way down before backtracking

Example: 8-Puzzle

Initial State:

7	2	4
5		6
8	3	1

Goal State:

	1	2
3	4	5
6	7	8

Formulation:

- **States:** 3×3 grid configurations
- **Actions:** Move blank up/down/left/right
- **Transition:** Swap blank with adjacent tile
- **Goal test:** Match goal configuration
- **Cost:** Number of moves (1 per move)

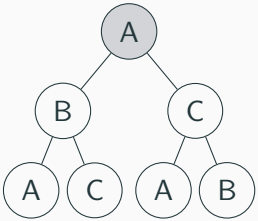
State space: $9!/2 = 181,440$ reachable states

How do we find the solution efficiently?

Search Trees vs Search Graphs

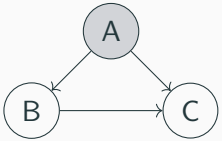
Search Tree:

- Nodes can repeat
- Shows exploration path
- May be infinite
- Simple to implement



Search Graph:

- Track visited states
- Avoid revisiting
- More efficient
- Requires more memory



In practice, always use graph search to avoid redundant work!

Uninformed Search Strategies

Uninformed Search

No domain knowledge, just brute force

Also called "blind search"

Characteristics:

- No knowledge about goal location
- Only know: goal vs non-goal
- Differ in order of exploration

Key data structure:

- **Frontier (fringe):** nodes to explore
- Order determines strategy

We'll cover:

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)
3. Uniform-Cost Search (UCS)

Evaluate on:

- Completeness
- Optimality
- Time complexity
- Space complexity

Search Algorithm Template

[1] frontier $\leftarrow$ {initial state} explored $\leftarrow$ {}
frontier not empty node $\leftarrow$ frontier.pop()
node is goal solution explored.add(node)
child in expand(node) child not in explored
frontier.add(child) failure

Key variations:

- **BFS:** frontier is queue (FIFO)
- **DFS:** frontier is stack (LIFO)
- **UCS:** frontier is priority queue

The only difference is how we choose which node to expand next!

Breadth-First Search

Breadth-First Search (BFS)

Explore level by level

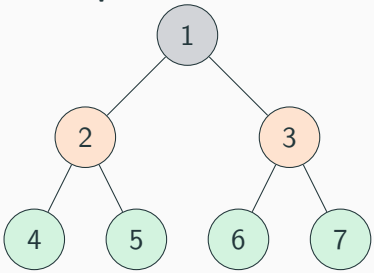
Strategy:

- Expand shallowest unexpanded node
- Use FIFO queue
- Explores by depth level

Properties:

- Complete
- Optimal (if uniform cost)
- High memory

Exploration Order:



Level 0: 1

Level 1: 2, 3

Level 2: 4, 5, 6, 7

BFS Implementation

```
from collections import deque
def bfs(problem):
    frontier = deque([problem.initial_state])
    explored = set()
    while frontier:
        node = frontier.popleft()  # FIFO - leftmost
        if problem.is_goal(node):
            return solution(node)
        explored.add(node)
        for child in problem.expand(node):
            if child not in explored and child not in frontier:
                frontier.append(child)
    return None  # No solution found
```

Key: Use deque for efficient FIFO operations

BFS Analysis

Let:

- b = branching factor (children per node)
- d = depth of shallowest solution

Time Complexity:

- Level 0: 1 node
- Level 1: b nodes
- Level 2: b^2 nodes
- Level d: b^d nodes

Total: $O(b^d)$

Space Complexity:

- Must store all nodes at current level
- Frontier size at level d: b^d

Total: $O(b^d)$

Memory is the main limitation!

Maze Example: Designing Breadth-First Search

To Implement BFS, What Do We Need?

Designing the algorithm before writing code

Before we write *any code*, we must answer a few key design questions:

- What is a **state**?
- What states should we explore next?
- When do we mark a state as **visited**?
- What does it mean to **expand** a state?
- How do we check for **success**?
- How do we **reconstruct the solution**?

Breadth-First Search is defined entirely by how we answer these questions.

What Is a State?

Defining the state space

Question: What information fully describes a situation?

What Is a State?

Defining the state space

Question: What information fully describes a situation?

Maze example:

- A state is the agent's current position
- Represented as a tuple: (row, col)

State space:

- All valid (non-wall) grid cells
- Each cell is a node in a graph

Search problems are graphs — states are nodes, actions are edges.

What States Should We Explore Next?

The frontier

Question: How do we choose which state to explore next?

What States Should We Explore Next?

The frontier

Question: How do we choose which state to explore next?

BFS answer:

- Always explore the **shallowest** unexpanded states
- Use a **FIFO queue**

Frontier:

- Contains states that have been discovered
- But not yet expanded

The frontier data structure defines the search strategy.

When Do We Mark a State as Visited?

Avoiding repeated work

Question: When should a state be marked as visited?

When Do We Mark a State as Visited?

Avoiding repeated work

Question: When should a state be marked as visited?

BFS rule:

- Mark a state as visited **when it is first discovered**
- That is, when it is added to the frontier

Why?

- Prevents adding the same state multiple times
- Guarantees the first path found is the shortest

Visited $\neq$ explored. Visited means "seen before."

What Does It Mean to Expand a State?

Generating successors

Question: What happens when we expand a state?

What Does It Mean to Expand a State?

Generating successors

Question: What happens when we expand a state?

To **expand** a state:

1. Remove it from the frontier
2. Generate all valid successor states
3. Add unvisited successors to the frontier

In a maze:

- Up, down, left, right
- Must be in bounds and not a wall

Expansion defines how the search graph grows.

How Do We Check for Success?

Goal testing

Question: When do we check if we are done?

How Do We Check for Success?

Goal testing

Question: When do we check if we are done?

BFS approach:

- Check the goal when a state is expanded
- The first time we reach the goal:
  - We have found the shortest path

Why this works:

- BFS explores states in increasing depth order

This is why BFS is optimal for uniform-cost problems.

How Do We Reconstruct the Solution?

Remembering how we got there

Question: How do we recover the path after reaching the goal?

How Do We Reconstruct the Solution?

Remembering how we got there

Question: How do we recover the path after reaching the goal?

Idea: parent pointers

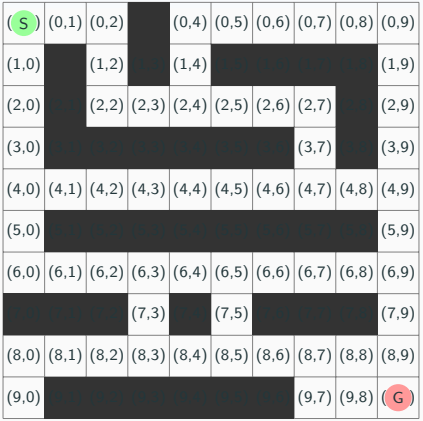
- When adding a state, record where it came from
- Store: `parent[child] = parent`

At the goal:

- Follow parent pointers backward
- Reverse the list to get start $\rightarrow$ goal

Search finds the goal; parent pointers give the solution.

Maze for BFS Example (with Coordinates)



Start = (0,0), Goal = (9,9)

Breadth-First Search: Setup

Maze coordinates: (row, col) **Neighbor order:** Up, Down, Left, Right

Start: $(0, 0)$
Goal: $(9, 9)$

Data structures:

- Queue (FIFO)
- `came_from` dictionary
- `explored` set

BFS Step 0

Initial state

Queue:

[_____]

Expand: $(0, 0)$

Which neighbors are added (in order)?

_____, _____

BFS Step 0 — Answer

Queue:

$$[(0,0)]$$

Neighbors of $(0,0)$:

$$(1,0),\ (0,1)$$

Queue after expansion:

$$[(1,0),\ (0,1)]$$

BFS Step 1

Expand: $(1, 0)$

Which new node is added?

_____

Queue after expansion:

[_____]

BFS Step 1 — Answer

New node added:

$$(2, 0)$$

Queue:

$$[(0, 1),\ (2, 0)]$$

BFS Step 2

Expand: $(0, 1)$

Which neighbor is valid and new?

_____

Queue becomes:

[_____]

BFS Step 2 — Answer

New node added:

$$(0, 2)$$

Queue:

$$[(2, 0),\ (0, 2)]$$

BFS Step 3

Expand: $(2, 0)$

Which node is added?

_____

Queue after expansion:

[_____]

BFS Step 3 — Answer

New node added:

$$(3, 0)$$

Queue:

$$[(0, 2),\ (3, 0)]$$

BFS Concept Check

Q1. Why is $(0, 0)$ never added again?

Answer: It is already recorded in `came_from`.

Q2. Why is $(3, 0)$ expanded after $(0, 2)$?

Answer: FIFO queue — $(0, 2)$ was discovered earlier.

Q3. Why does BFS guarantee the shortest path?

Answer: Nodes are expanded in order of increasing path length.

Q4. What changes if we reverse neighbor order?

Answer: The path shape may change, but its length remains minimal.

Depth-First Search: Setup

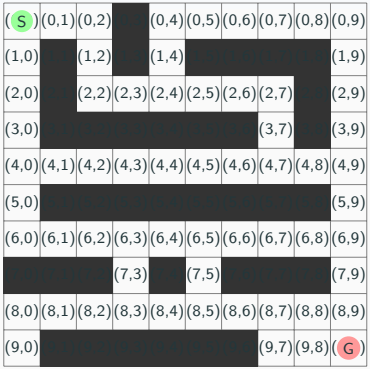
Maze coordinates: (row, col) **Neighbor order:** Up, Down, Left, Right

Start: $(0, 0)$
Goal: $(9, 9)$

Data structures:

- Stack (LIFO)
- `came_from` dictionary
- `explored` set

Depth-First Search (DFS): First Steps



Start $(0, 0)$ Goal $(9, 9)$

Algorithm: Depth-First Search **Structure:** Stack (LIFO) **Neighbor order:** Up, Down, Left, Right

Step 0

Stack:

$$[(0, 0)]$$

Expand $(0, 0)$

Valid neighbors:

$$(1, 0),\ (0, 1)$$

Push in order:

$$[(1, 0),\ (0, 1)]$$

Step 1

Pop top:

$$(0, 1)$$

New neighbor:

$$(0, 2)$$

Stack:

$$[(1, 0),\ (0, 2)]$$

Key idea: DFS follows one path as deep as possible before backtracking.

DFS Step 0

Initial state

Stack:

[_____]

Expand: $(0, 0)$

Which neighbors are pushed (in order)?

_____, _____

DFS Step 0 — Answer

Stack:

$$[(0,0)]$$

Neighbors of $(0,0)$ (Up blocked):

$$(1,0),\ (0,1)$$

Stack after push:

$$[(1,0),\ (0,1)]$$

Next expanded: $(0,1)$ (top of stack)

DFS Step 1

Expand: $(0, 1)$

Which neighbor is valid and new?

_____

Stack after expansion:

[_____]

DFS Step 1 — Answer

New node pushed:

$$(0, 2)$$

Stack:

$$[(1, 0),\ (0, 2)]$$

Next expanded: $(0, 2)$

DFS Step 2

Expand: $(0, 2)$

Which neighbor is pushed next?

_____

Stack becomes:

[_____]

DFS Step 2 — Answer

New node pushed:

$$(1, 2)$$

Stack:

$$[(1, 0),\ (1, 2)]$$

DFS Step 3

Expand: $(1, 2)$

Which direction does DFS go next?

DFS Step 3 — Answer

DFS continues **downward**:

$$(2, 2)$$

Key observation:

DFS commits to a direction until it hits a dead end.

DFS Concept Check

Q1. Why does DFS go deep instead of wide?

Answer: Because it uses a LIFO stack.

Q2. Can DFS find the goal faster than BFS?

Answer: Sometimes — if it goes in the right direction early.

Q3. Does DFS guarantee the shortest path?

Answer: No. It may find a much longer path first.

Q4. What happens if DFS reaches a dead end?

Answer: It backtracks by popping from the stack.

BFS vs DFS: Key Difference

	BFS	DFS
Data structure	Queue (FIFO)	Stack (LIFO)
Exploration	Level by level	Deep first
Shortest path	Yes	No
Memory use	Higher	Lower

Implementation: What the BFS Code Is Doing

BFS: From Design to Code

Mapping ideas to implementation

Every part of BFS design corresponds to a concrete data structure:

Concept	Implementation
State	(row, col) tuple
Frontier	`deque` (FIFO queue)
Visited	Dictionary / set
Expansion	Neighbor generation
Goal test	Equality check
Path	Parent pointers

Frontier and Visited in Code

Initialization

```
frontier = deque([start])
came_from = {start: None}
```

- `frontier`: states waiting to be expanded
- `came_from`: marks states as visited
- Also stores parent pointers

A state is visited as soon as it enters the frontier.

Expansion Loop in Code

The core of BFS

```
current = frontier.popleft()
for neighbor in get_neighbors(grid, current):
    if neighbor not in came_from:
        came_from[neighbor] = current
        frontier.append(neighbor)
```

- FIFO queue enforces breadth-first order
- Neighbors are added exactly once
- Parent pointers recorded immediately

This loop is BFS. Everything else is bookkeeping.

Goal Test and Path Reconstruction

Finishing the search

```
if current == goal:
    path = []
    while current is not None:
        path.append(current)
        current = came_from[current]
    path.reverse()
```

- Goal test during expansion
- Parent pointers recover the solution
- Path length equals depth of goal

Search explores the graph; reconstruction tells the story.

Key Takeaway: BFS as Design Choices

Breadth-First Search

BFS is not a single algorithm — it is a set of design decisions:

- States define the graph
- Frontier ordering defines behavior
- Visited policy ensures correctness
- Parent pointers recover solutions

Change one design choice, and you get DFS, UCS, or A*.

Depth-First Search

Depth-First Search (DFS)

Explore as deep as possible

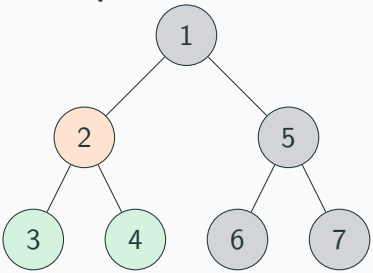
Strategy:

- Expand deepest unexpanded node
- Use LIFO stack
- Go deep, then backtrack

Properties:

- Not complete (can get stuck)
- Not optimal
- Low memory

Exploration Order:



Order: 1 → 2 → 3, 4 → 5, 6, 7

DFS Implementation

```
def dfs(problem):
    frontier = [problem.initial_state]  # Stack (list)
    explored = set()
    while frontier:
        node = frontier.pop()  # LIFO - rightmost
        if problem.is_goal(node):
            return solution(node)
        explored.add(node)
        for child in problem.expand(node):
            if child not in explored and child not in frontier:
                frontier.append(child)
    return None
```

Key: Use list with pop() for LIFO (stack) operations

DFS Analysis

Let:

- b = branching factor
- m = maximum depth of tree

Time Complexity:

- Worst case: explore entire tree
- Could explore all nodes up to depth m

Total: $O(b^m)$

Problem: If $m >> d$, very wasteful!

Space Complexity:

- Only store path from root to current node
- Plus siblings of nodes on path

Total: $O(bm)$

Advantage: Much less memory than BFS!

Example: $b = 10$, $m = 100 \Rightarrow$ only 1,000 nodes in memory

DFS Variants

Depth-Limited Search (DLS):

- DFS with depth limit ℓ
- Prevents infinite loops
- Not complete if $\ell < d$

Iterative Deepening DFS (IDDFS):

- Run DLS with increasing limits: $\ell = 0, 1, 2, \ldots$
- Combines benefits of BFS and DFS
- Complete like BFS
- Low memory like DFS
- Optimal (if uniform cost)

IDDFS is often the best uninformed search strategy!

Uniform-Cost Search

Uniform-Cost Search (UCS)

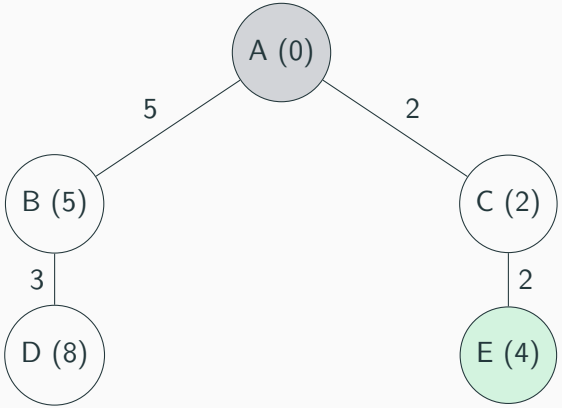
Expand lowest-cost node

Strategy:

- Expand node with lowest path cost
- Use priority queue
- Priority $= g(n)$ (cost so far)

Properties:

- Complete
- Optimal
- Can be slow



Expansion order: A(0) $\rightarrow$ C(2) $\rightarrow$ E(4) $\rightarrow$ B(5)

UCS Implementation

```
import heapq
def ucs(problem):
    frontier = [(0, problem.initial_state)]  # (cost, node)
    explored = set()
    while frontier:
        cost, node = heapq.heappop(frontier)  # Get min cost
        if problem.is_goal(node):
            return solution(node)
        explored.add(node)
        for child, action_cost in problem.expand(node):
            new_cost = cost + action_cost
            if child not in explored:
                heapq.heappush(frontier, (new_cost, child))
    return None
```

UCS Analysis

Let:

- C^* = cost of optimal solution
- ϵ = minimum step cost

Time Complexity:

- Expands all nodes with cost $< C^*$
- Number of such nodes: $O(b^{C^*/\epsilon})$

Can be much worse than BFS!

Space Complexity:

- Must store all generated nodes
- Same as time: $O(b^{C^*/\epsilon})$

Also high memory usage

When to use: When actions have different costs and you need optimal solution

Uniform Cost Search: Setup

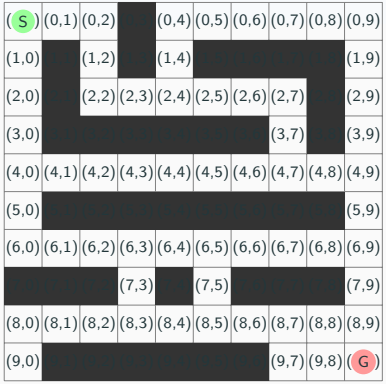
Maze coordinates: (row, col) **Neighbor order:** Up, Down, Left, Right **Step cost:** 1 per move

Start: $(0, 0)$
Goal: $(9, 9)$

Data structures:

- Priority queue ordered by $g(n)$
- `g_cost` dictionary
- `came_from` dictionary
- `explored` set

Uniform Cost Search (UCS): First Steps



Start $(0, 0)$ Goal $(9, 9)$

Data structure: Priority Queue (min g) **Edge cost:** 1 per move

Step 0

Priority Queue:

$$[(0, (0, 0))]$$

Expand $(0, 0)$

Valid neighbors:

$$(1, 0),\ (0, 1)$$

Add with cost:

$$(1, (1, 0)),\ (1, (0, 1))$$

Queue:

$$[(1, (1, 0)),\ (1, (0, 1))]$$

Step 1

Remove lowest-cost node:

$$(1, (1, 0))$$

New neighbor:

$$(2, 0) \text{ with } g = 2$$

Queue:

$$[(1, (0, 1)),\ (2, (2, 0))]$$

Key idea: UCS always expands the lowest-cost path so far.

Uniform Cost Search: Step 0

Initial state

Priority queue:

[(____, _____)]

Expand: $(0, 0)$

Which neighbors are inserted (with cost)?

_____, _____

Uniform Cost Search: Step 0 — Answer

Priority queue:

$$[(0,\ (0,0))]$$

Neighbors of $(0,0)$:

$$(1,0),\ (0,1)$$

Each has cost:

$$g = 1$$

Priority queue:

$$[(1,\ (1,0)),\ (1,\ (0,1))]$$

Uniform Cost Search: Step 1

Which node is removed next?

_____

Why?

_____

Uniform Cost Search: Step 1 — Answer

Node removed:

$$(1, 0)$$

Reason:

It has the smallest $g(n)$

New neighbor added:

$$(2, 0) \text{ with } g = 2$$

Priority queue:

$$[(1,\ (0, 1)),\ (2,\ (2, 0))]$$

Uniform Cost Search: Step 2

Next node removed:

_____

New node added:

_____

Uniform Cost Search: Step 2 — Answer

Node removed:

$$(0, 1)$$

New node added:

$$(0, 2) \text{ with } g = 2$$

Priority queue:

$$[(2,\ (2, 0)),\ (2,\ (0, 2))]$$

Important Observation

- UCS always expands the node with **lowest path cost**
- When all step costs are equal:
  - UCS behaves exactly like BFS
- When step costs differ:
  - UCS may expand deeper nodes first

UCS guarantees:

$$\text{First time goal is removed} \Rightarrow \text{optimal path}$$

Uniform Cost Search: Concept Check

Q1. Why doesn't UCS use a FIFO queue?

Answer: Because expansion order depends on cost, not discovery time.

Q2. Why is UCS optimal?

Answer: It never expands a higher-cost path before a lower-cost one.

Q3. When does UCS reduce to BFS?

Answer: When all step costs are equal.

Q4. What does A* add on top of UCS?

Answer: A heuristic estimate of remaining cost.

Comparison and Summary

Algorithm Comparison

Algorithm	Complete	Optimal	Time	Space
BFS		*	$O(b^d)$	$O(b^d)$
DFS			$O(b^m)$	$O(bm)$
IDDFS		*	$O(b^d)$	$O(bd)$
UCS			$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$

*Optimal only if all step costs are equal

Use BFS when:

- Uniform costs
- Shallow solution
- Have memory

Use DFS/IDDFS when:

- Limited memory
- Deep trees
- Many solutions

DFS Application: Cycle Detection

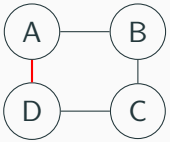
Undirected graphs:

```
bool hasCycleDFS(Graph& g, int v,
                 vector<bool>& visited, int parent) {
    visited[v] = true;
    for (int u : g.adj[v]) {
        if (!visited[u]) {
            if (hasCycleDFS(g, u, visited, v))
                return true;
        }
        // Visited and not parent = back edge = cycle!
        else if (u != parent) {
            return true;
        }
    }
    return false;
}

bool hasCycle(Graph& g) {
    vector<bool> visited(g.V, false);
    for (int i = 0; i < g.V; i++) {
        if (!visited[i] && hasCycleDFS(g, i, visited, -1)
            )
            return true;
    }
    return false;
}
```

Key idea:

If we visit a vertex that's already visited AND it's not our parent, we found a cycle!

Example with cycle:



DFS from A: A→B→C→D

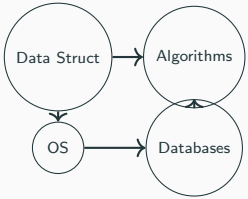
At D, we see A is visited and not parent → cycle!

DFS Application: Topological Sort

Topological sort: Linear ordering of vertices in a DAG (Directed Acyclic Graph)

For every edge u $\rightarrow$ v, u comes before v in the ordering

Example: Course prerequisites



Valid order: DS $\rightarrow$ OS $\rightarrow$ Algo $\rightarrow$ DB
or: DS $\rightarrow$ Algo $\rightarrow$ OS $\rightarrow$ DB

Algorithm (DFS-based):

1. Do DFS
2. When finishing a vertex (all neighbors visited), add to stack
3. Pop stack to get topological order

Time: O(V + E)

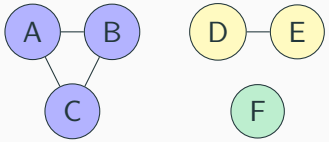
Note:

- Only works on DAGs
- If cycle exists, no valid ordering
- Multiple valid orderings may exist

Connected Components

Connected component: Maximal set of vertices where path exists between any two

Example:



3 components: {A,B,C}, {D,E}, {F}

Algorithm (using DFS):

```
int countComponents(Graph& g) {
    vector<bool> visited(g.V, false);
    int count = 0;

    for (int v = 0; v < g.V; v++) {
        if (!visited[v]) {
            DFS(g, v, visited);
            count++;
        }
    }

    return count;
}
```

Time: $O(V + E)$

Each DFS call explores one component

Summary

Graphs: Vertices connected by edges

Representations:

- Adjacency matrix: O(V^2) space, O(1) edge check, dense graphs
- Adjacency list: O(V + E) space, O(degree) edge check, sparse graphs (most common)

Traversals: Both O(V + E) time

- BFS: Queue, level-by-level, shortest path (unweighted)
- DFS: Stack/recursion, go deep, cycle detection, topological sort

Applications:

- Shortest path (BFS)
- Cycle detection (DFS)
- Topological sort (DFS)
- Connected components (DFS or BFS)

What's Next

Today: Graphs - representations, BFS, DFS

Next: Greedy Algorithms

- Strategy: Make locally optimal choices
- Activity selection problem
- Minimum spanning trees (Kruskal's algorithm)
- When greedy works and when it doesn't

Later:

- Dynamic Programming
- Complexity Theory (P vs NP)