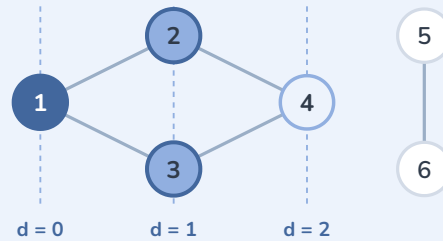


Graph search: BFS and DFS

Chapter 3.1 · Overall level: Intermediate



Stations joined by tracks, web pages joined by links, cells of a maze joined by open doors: all of them are graphs. *Which places can be reached from here, and in how few steps?*

Two systematic searches answer those questions: breadth-first search (BFS), which spreads out in rings, and depth-first search (DFS), which follows one route as far as it can. Both visit every vertex and every edge at most a constant number of times.

3.1.1 Vertices, edges and adjacency lists

A **graph** has **vertices** (here, stations numbered 1 to n) and **edges** that join pairs of them. An edge is **undirected** if it can be used both ways, like a two-way track, and **directed** if it goes one way only, like a link from one web page to another. Our running example has six stations and the undirected edges 1–2, 1–3, 2–4, 3–4 and 5–6.

A program stores a graph as **adjacency lists**: for every vertex, the list of vertices one edge away. Reading the edges once fills all lists, and a search can then ask “where can I go from u ?” by scanning one list.

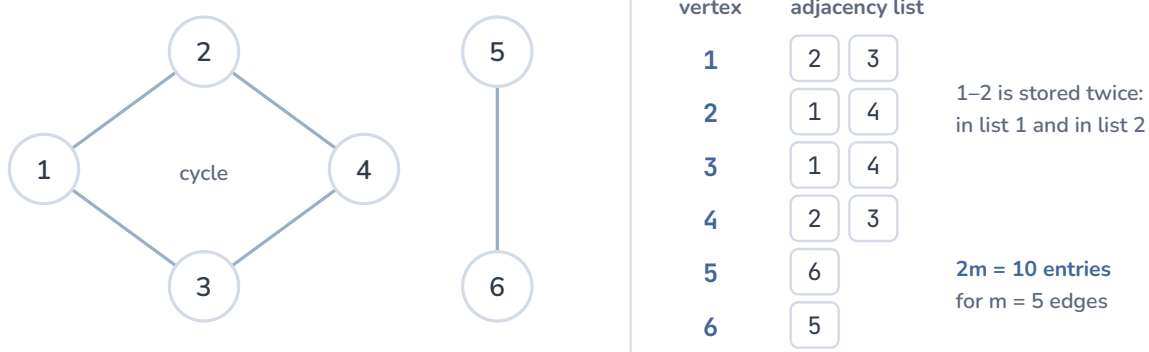


Figure 3.1.1 — The running example and its adjacency lists. An undirected edge is inserted into both lists, so the lists hold $2m$ entries in total.

Directed versus undirected insertion

For each input edge “ $u\ v$ ” the only difference is one line:

```
undirected:  adj[u].append(v);  adj[v].append(u)
directed:    adj[u].append(v)
```

If the five edges above were directed ($1 \rightarrow 2$, $1 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 4$, $5 \rightarrow 6$), the lists would be 1: [2, 3], 2: [4], 3: [4], 4: [], 5: [6], 6: []. From 4 nothing could then be reached. Forgetting the second line for an undirected graph is one of the most common graph bugs: the search silently refuses to travel backwards along an edge.

3.1.2 Cycles and the visited array

The edges $1-2$, $2-4$, $4-3$, $3-1$ form a **cycle**: a route that comes back to its start. A naive search that simply “goes to every neighbour” would walk $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow \dots$ forever, or explode into a huge number of repeated visits. The cure is a **visited** array with one flag per vertex. A vertex is handled the first time it is reached, and every later arrival is ignored.

With the flag, each vertex is handled at most once, and each adjacency list is scanned at most once. That is where the $O(n + m)$ running time of every search in this chapter comes from: n vertices plus $2m$ list entries.

3.1.3 Breadth-first search: distances from a queue

BFS explores in rings. Start with the source in a **queue** (first in, first out). Repeatedly take the vertex at the front, *process* it by scanning its list, and put every neighbour not seen before at the back, marking it visited and recording its distance as one more than the current vertex. Figure 3.1.2 traces BFS from vertex 1.

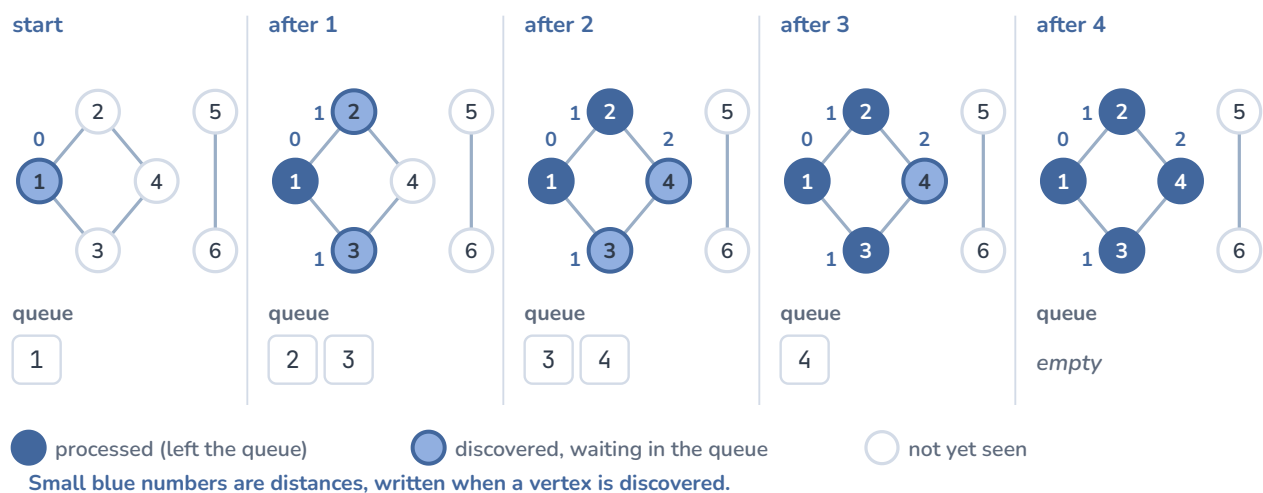


Figure 3.1.2 — Every vertex is enqueued once, when its shortest unweighted distance is first discovered. Processed vertices are dark; discovered vertices still in the queue are blue. In “after 3” vertex 4 is not added again: it is already marked.

Mark when you enqueue

Vertex 4 is a neighbour of both 2 and 3. Because it is marked the moment 2 discovers it, vertex 3 sees the mark and does not add it a second time. If vertices were marked only when they leave the queue, 4 would be queued twice, and in a dense graph the queue could hold up to $2m$ entries instead of n . Marking on enqueue keeps every vertex in the queue at most once.

Vertex	1	2	3	4	5	6
Distance from 1	0	1	1	2	-1	-1

Vertices 5 and 6 are never discovered, so they keep the starting value -1 , which the practice problems use for “unreachable”.

Why BFS distances are shortest

Call the set of vertices at true distance d from the source **layer d** . Layer 0 is the source alone. Suppose the queue holds exactly layer d , in some order, and all closer vertices are already processed. While layer d is processed, every vertex of layer $d + 1$ is discovered, because the last step of its shortest route comes from layer d . No vertex of a farther layer is discovered, because it has no neighbour in layer d ; and no closer vertex is discovered again, because it is already marked. So when layer d is finished, the queue holds exactly layer $d + 1$, each vertex labelled $d + 1$. By induction every label is the true distance.

In short, the queue always contains at most two neighbouring layers, nearer ones in front. A first discovery can never be improved later, because anything discovered later is at least as far away.

Pseudocode

BFS from s : $O(n + m)$ time, $O(n + m)$ memory

```

dist[1..n] ← -1;  dist[s] ← 0
queue ← [s]
while the queue is not empty:
    u ← take the front of the queue
    for each v in adj[u]:
        if dist[v] = -1:                                (dist doubles as the visited mark)
            dist[v] ← dist[u] + 1
            put v at the back of the queue

```

3.1.4 Depth-first search with an explicit stack

DFS goes deep first: from the current vertex it moves to an unvisited neighbour, from there to another, and backs up only when it is stuck. The natural tool is a **stack** (last in, first out). Pop a vertex; if it is already visited, skip it; otherwise visit it and push its unvisited neighbours. Pushing a list in reverse order makes the smallest neighbour come out first, which keeps traces tidy.



Figure 3.1.3 — Stack snapshots of iterative DFS from vertex 1 (only the component of 1 is shown). The search reaches 3 last, along $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$, although 3 is a direct neighbour of 1.

Two details are visible in the trace. First, vertex 3 sits in the stack twice: it was pushed by 1 and again by 4 before being visited. That is why the check happens when a vertex is popped; the stack can hold up to $2m$ entries, which is still $O(n + m)$. Second, DFS reached 3 by a route of three edges, while its distance is 1. **DFS does not find shortest routes**; use BFS for distances. For questions that only ask *whether* something is reachable, or which vertices belong together, either search works.

Pseudocode

Iterative DFS from s : $O(n + m)$ time, stack of at most $2m + 1$ entries

```

stack ← [s]
while the stack is not empty:
    u ← pop the top of the stack
    if visited[u]: skip to the next round
    visited[u] ← true; visit u
    for each v in adj[u], in reverse order:
        if not visited[v]: push v
  
```

Why not recursion?

DFS is often written recursively: visit u , then call the function on each unvisited neighbour. That is elegant, but on a long path of 200,000 vertices the recursion is 200,000 calls deep. Python stops at about 1,000 levels by default, and Java and C++ programs can crash with a stack overflow, because the call stack has a fixed, fairly small size. An explicit stack or queue lives in ordinary memory and has no such limit, so the programs in this library always search iteratively.

3.1.5 Code in four languages

Each program solves the first practice problem, **Reachable stations**. The first line holds n and m , then m lines “ $u\ v$ ” describe undirected edges. The program prints how many stations can be reached from station 1, including station 1 itself.

Sample input	Sample output
6 5 1 2 1 3 2 4 3 4 5 6	4

The programs run BFS from station 1 with the visited mark set on enqueue. Every reached station enters the queue exactly once, so the final queue length is the answer. Replacing the flag by a distance array, exactly as in the pseudocode, turns the same program into a solution for distances; the queue is an array with a head index, which avoids the slow removal of the first element of a list.

Language	Queue for BFS	Stack for DFS
Python	<code>list</code> + head index, or <code>collections.deque</code>	<code>list</code> : <code>append</code> , <code>pop()</code>
C++	<code>vector</code> + head index, or <code>std::queue</code>	<code>vector</code> : <code>push_back</code> , <code>pop_back</code>
Java	<code>int[]</code> + head/tail, or <code>ArrayDeque</code>	<code>int[]</code> + top, or <code>ArrayDeque</code>
JavaScript	array + head index (never <code>shift()</code>)	array: <code>push</code> , <code>pop</code>

The Python program writes `for u in queue` while appending to `queue`: a Python loop over a list also reaches items appended during the loop, so this is exactly a queue with a moving head.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    adj = [[] for _ in range(n + 1)]
    for k in range(m):
        u, v = int(data[2 + 2 * k]), int(data[3 + 2 * k])
        adj[u].append(v)          # undirected: store both directions
        adj[v].append(u)
    visited = [False] * (n + 1)
    visited[1] = True
    queue = [1]
    for u in queue:               # also visits vertices appended below
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True  # mark when it enters the queue
                queue.append(v)
    print(len(queue))            # every reached station entered once

main()
```

C++17

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n + 1);
    for (int k = 0; k < m; k++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);      // undirected: store both directions
        adj[v].push_back(u);
    }
    vector<bool> visited(n + 1, false);
    vector<int> queue = {1};
    visited[1] = true;
    for (size_t head = 0; head < queue.size(); head++) {
        int u = queue[head];
        for (int v : adj[u]) {
            if (visited[v]) continue;
            visited[v] = true;    // mark when it enters the queue
            queue.push_back(v);
        }
    }
    cout << queue.size() << '\n'; // every reached station entered once
}
```

Java 17

```

import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken(); int n = (int) in.nval;
        in.nextToken(); int m = (int) in.nval;
        List<List<Integer>> adj = new ArrayList<>();
        for (int v = 0; v <= n; v++) adj.add(new ArrayList<>());
        for (int k = 0; k < m; k++) {
            in.nextToken(); int u = (int) in.nval;
            in.nextToken(); int v = (int) in.nval;
            adj.get(u).add(v);           // undirected: both directions
            adj.get(v).add(u);
        }
        boolean[] visited = new boolean[n + 1];
        int[] queue = new int[n];        // each station enters at most once
        queue[0] = 1; visited[1] = true; // start at station 1
        int head = 0, tail = 1;         // queue[head..tail-1] is waiting
        while (head < tail) {
            for (int v : adj.get(queue[head++])) {
                if (visited[v]) continue;
                visited[v] = true;       // mark when it enters the queue
                queue[tail++] = v;
            }
        }
        System.out.println(tail);
    }
}

```

JavaScript (Node)

```

const data = require("fs").readFileSync(0, "utf8").trim().split(/\s+/).map(Number);
const n = data[0], m = data[1];
const adj = Array.from({ length: n + 1 }, () => []);
for (let k = 0; k < m; k++) {
    const u = data[2 + 2 * k], v = data[3 + 2 * k];
    adj[u].push(v);           // undirected: store both directions
    adj[v].push(u);
}
const visited = new Uint8Array(n + 1);
visited[1] = 1;
const queue = [1];
for (let head = 0; head < queue.length; head++) {
    for (const v of adj[queue[head]]) {
        if (visited[v]) continue;
        visited[v] = 1;       // mark when it enters the queue
        queue.push(v);
    }
}
console.log(queue.length);    // every reached station entered once

```

3.1.6 Connected components

A **connected component** is a maximal group of vertices that can all reach one another. The running example has two: {1, 2, 3, 4} and {5, 6}. One search finds exactly the component of its start: it follows every edge out of every vertex it reaches, so it cannot miss a vertex joined to the start by a route, and it never crosses to a vertex with no route.

To find all components, scan the vertices in order and start a new search from each one that is still unvisited, counting the starts:

```
components ← 0
for v from 1 to n:
    if not visited[v]:
        components ← components + 1
        search from v, marking everything it reaches
```

This also matters whenever a property must hold for the whole graph: an input may be disconnected, and a search started only at vertex 1 sees just one component. The total work is still $O(n + m)$, since every vertex is visited once over all searches.

3.1.7 Grids as graphs

A grid of free (.) and blocked (#) cells is a graph in disguise: each free cell is a vertex, and two free cells sharing a side are joined by an edge. There is no need to build adjacency lists. The neighbours of cell (r, c) are generated on the fly from two small offset arrays, $dr = [-1, 1, 0, 0]$ and $dc = [0, 0, -1, 1]$, skipping any position outside the grid *before* looking at the cell.

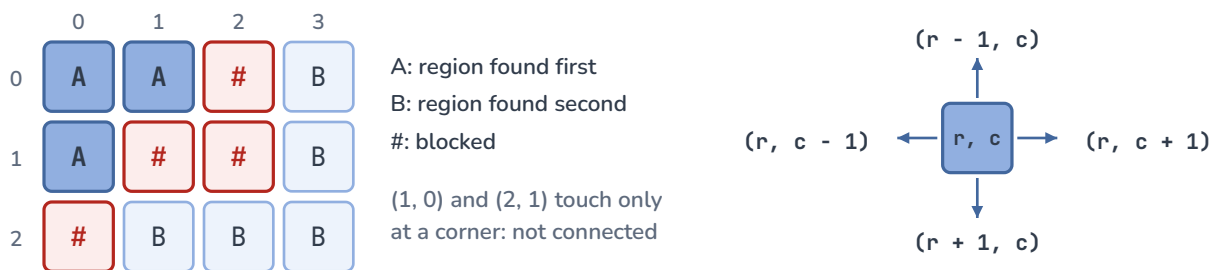


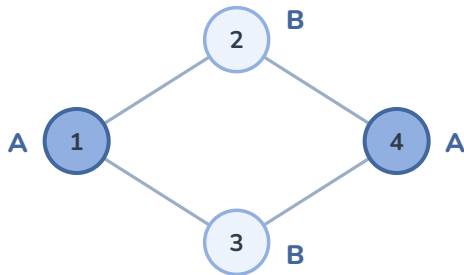
Figure 3.1.4 — Flood fill on a 3×4 grid finds two regions. Diagonal contact does not connect cells, so only the four side neighbours are used.

Counting regions is the component count again: scan the cells row by row, and from each free, unvisited cell start a BFS or DFS that marks its whole region (a **flood fill**). An $n \times m$ grid has nm vertices and fewer than $4nm$ neighbour checks, so the cost is $O(nm)$. Store a cell as one number $r \cdot m + c$ if a language makes pairs expensive.

3.1.8 Two-colouring

Can every vertex receive a label A or B so that each edge joins different labels? Such a graph is called **bipartite**: think of two groups where every edge goes between the groups, never inside one. BFS answers the question. Give the start label A; whenever a vertex is discovered, give it the opposite label to the vertex that found it; and whenever an edge joins two vertices that already have the same label, report a conflict. Run this from every unlabelled vertex, as in Section 3.1.6, because each component is labelled separately.

Even cycle: success



Odd cycle: conflict

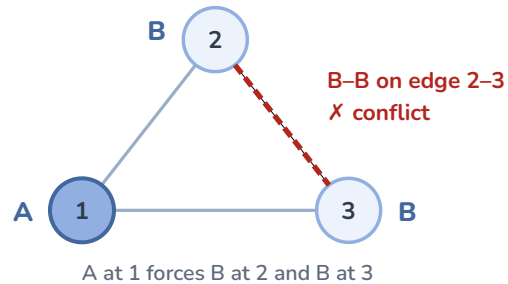


Figure 3.1.5 — An odd cycle forces a conflict in any proposed two-colouring. Labels A and B are written next to each vertex as well as shaded.

Why a conflict proves “impossible”. Inside one component the labels are forced: once the start is A, each neighbour must be B, each neighbour of those must be A, and so on, so a vertex’s label is A exactly when its BFS distance is even. (Swapping all labels of a component changes nothing important.) If an edge joins u and v with equal labels, their distances have the same parity, so the route from the start to u , the edge, and the route back from v form a closed walk of odd length, and an odd closed walk always contains an odd cycle. Around an odd cycle the labels must alternate A, B, A, ... and come back to the start with the wrong label, so no labelling exists. If no edge causes a conflict, the labels found are a valid answer.

In Figure 3.1.5 the square 1–2–4–3 is a cycle of length 4, and A, B, B, A works. The triangle has length 3, and the edge 2–3 always fails.

3.1.9 Cost and size

Task	Time	Extra memory
Read edges into adjacency lists	$O(n + m)$	2m list entries
BFS or iterative DFS from one vertex	$O(n + m)$	queue $\leq n$, or stack $\leq 2m$
All components or two-colouring	$O(n + m)$	one array of n labels
Flood fill on an $n \times m$ grid	$O(nm)$	one mark per cell

Distances are at most $n - 1$, so 32-bit integers are plenty. With $n = m = 200,000$ the lists hold 400,000 numbers, which every language handles easily.

3.1.10 Common mistakes

- **Using BFS for weighted shortest paths.** BFS counts edges, not their lengths. If 1–2 costs 5 while 1–3 and 3–2 cost 1 each, BFS reports one hop to 2, but the cheapest route costs 2. Weighted graphs need Dijkstra’s algorithm (Chapter 3.3).
- **Forgetting reverse edges.** For an undirected edge, append to both lists. A search that cannot walk back along an edge misses vertices.
- **Recursive DFS on a long path.** Deep recursion overflows the call stack. Use an explicit stack or BFS.
- **Marking visited only on dequeue in BFS.** A vertex can then be queued many times. Mark it, and set its distance, when it is enqueued.
- **Searching only from vertex 1.** Component counts and two-colouring must restart from every unvisited vertex; an odd cycle may hide in another component.
- **Checking a grid cell before the bounds.** Test $0 \leq r < n$ and $0 \leq c < m$ first, then read the cell.

3.1.11 Where plain search stops

BFS gives shortest routes only when every edge counts the same. With weights, use Dijkstra’s algorithm (Chapter 3.3). If edges keep arriving and the question is only “are these two connected now?”, re-running a search after every edge is too slow; disjoint sets (Chapter 3.2) answer it almost instantly. In directed graphs a search from u finds what u can reach, but groups that can reach each other both ways need strongly connected components (Chapter 4.4). Ordering tasks so that every arrow points forward (topological order) and counting shortest routes are further variations on the same searches.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Why does BFS mark a vertex as visited when it is enqueued, rather than only when it is dequeued?
2. ● Does DFS generally find a shortest unweighted route first? Use Figure 3.1.3 to explain.
3. ● Write the adjacency lists for the directed edges 1→2, 2→3, 3→1, 1→3. Then write them for the same edges treated as undirected.
4. ●● Run BFS from 1 on the undirected edges 1–2, 1–5, 2–3, 5–4, 3–4, 4–6, scanning each list in input order. Give each queue state and the final distances.
5. ●● How many regions of free cells does the grid `.# / ## / .#` have? Why?
6. ●● Can the cycle 1–2–3–4–5–1 be two-coloured? What about a cycle with six vertices?

Answers and explanations

Read these after attempting the exercises.

1. To avoid duplicate queue entries. Once a vertex is discovered its distance is final, so any later discovery must be ignored; marking at enqueue time is what lets other vertices see that it is already taken.
2. No. In Figure 3.1.3, DFS reaches 3 through $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$ (three edges), although the edge $1-3$ gives a route of one edge. Only BFS guarantees shortest unweighted routes.
3. Directed: 1: [2, 3], 2: [3], 3: [1]. Undirected: 1: [2, 3, 3], 2: [1, 3], 3: [2, 1, 1]. The edges $3 \rightarrow 1$ and $1 \rightarrow 3$ become two parallel undirected edges between 1 and 3, so each list holds the other vertex twice.
4. Lists: 1: [2, 5], 2: [1, 3], 3: [2, 4], 4: [5, 3, 6], 5: [1, 4], 6: [4]. Queues: [1], [2, 5], [5, 3], [3, 4], [4], [6], []. Distances for vertices 1 to 6: **0, 1, 2, 2, 1, 3**.
5. **Five**. Every free cell (the four corners and the centre) touches the others only diagonally, and only side neighbours connect.
6. A 5-cycle cannot: labels must alternate A, B, A, B, A around it, and the last edge $5-1$ joins two A vertices. A 6-cycle can: A, B, A, B, A, B.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Reachable stations

Apply

An undirected graph has n stations. Count the stations reachable from station 1, including station 1 itself.

Minimum hops

Connect

Find the minimum number of edges from station 1 to every vertex of an undirected graph. Print -1 for unreachable vertices.

Open regions

Reason

Count connected regions of free cells in a grid. Cells connect only through a shared side, not diagonally.

Two compatible groups

Stretch

Decide whether the vertices of an undirected graph can be labeled A or B so that every edge joins different labels.