

Graph Traversal

Riccardo Maso

September 24, 2026



RAVEN – Research on Algorithms Venice



Graph Traversal

Introduction

DFS

BFS

Connected Components

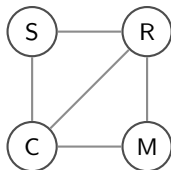
Flood Fill

Summary

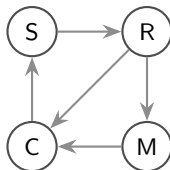
Your Task

What is a Graph?

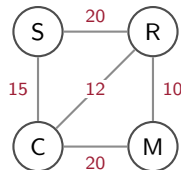
- $G = (V, E)$: **vertices** V connected by **edges** E
- example: a map of Venice, places = vertices, streets = edges



undirected
two-way streets



directed
one-way streets



weighted
minutes on foot

S = Stazione, R = Rialto, M = San Marco, C = Ca' Foscari

Graph Traversal — Overview

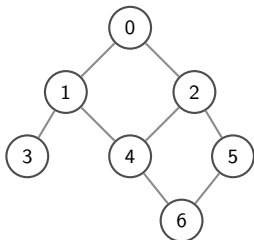
What is it?

- start from a **source** vertex s
- visit every vertex **reachable** from s , exactly once
- two ways: **DFS** and **BFS**

Why?

- reachability, connected components, shortest paths, ...
- building block of most graph algorithms (§4.2)

Storing a Graph (§2.4.1)

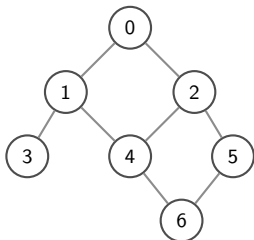


Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	1	0	0
2	1	0	0	0	1	1	0
3	0	1	0	0	0	0	0
4	0	1	1	0	0	0	1
5	0	0	1	0	0	0	1
6	0	0	0	0	1	1	0

- matrix: memory $O(V^2)$

Storing a Graph (§2.4.1)

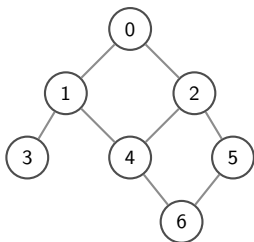


Edge List

(0,1)	(1,4)
(0,2)	(2,4)
(1,3)	(2,5)
(4,6)	(5,6)

- matrix: memory $O(V^2)$
- edge list: memory $O(E)$, but finding neighbours is slow

Storing a Graph (§2.4.1)



Adjacency List

u	AL[u]
0	1, 2
1	0, 3, 4
2	0, 4, 5
3	1
4	1, 2, 6
5	2, 6
6	4, 5

- matrix: memory $O(V^2)$
- edge list: memory $O(E)$, but finding neighbours is slow
- adjacency list: memory $O(V + E) \Rightarrow$ **what we use for traversal**

Introduction
○○○

DFS
●○○

BFS
○○○

Connected Components
○○○○○○○

Flood Fill
○○○○○○○○○

Summary
○○

Your Task
○○○

Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

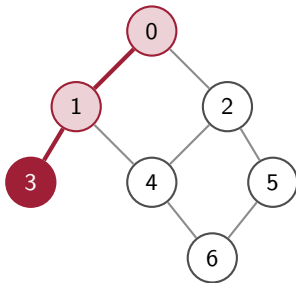
Your Task

Depth First Search (§4.2.2)

- go **deep**; when stuck, **backtrack**
- **recursion** remembers the way back: when $\text{DFS}(v)$ ends, we return to $\text{DFS}(u)$ and try its next neighbour

```
1 DFS(u):  
2   visited[u] ← true  
3   for each v in AL[u]:  
4     if not visited[v]:  
5       DFS(v)
```

DFS — Example: DFS(0)



go deep: $0 \rightarrow 1 \rightarrow 3$

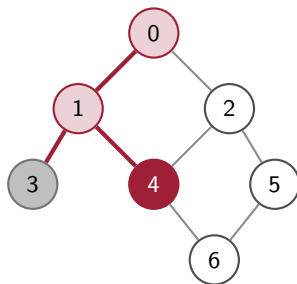
recursive calls

DFS(0)
└─ DFS(1)
 └─ DFS(3)

visit order

0 1 3

DFS — Example: DFS(0)



recursive calls

```

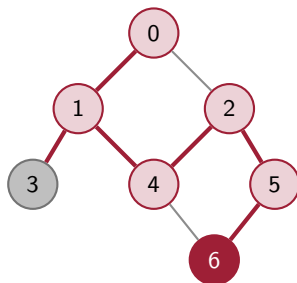
DFS(0)
├── DFS(1)
│   ├── DFS(3)
│   └── DFS(4)
  
```

visit order

0 1 3 4

DFS(3): no unvisited neighbour $\Rightarrow$ returns; DFS(1) goes on with DFS(4)

DFS — Example: DFS(0)



4 → 2 → 5 → 6

recursive calls

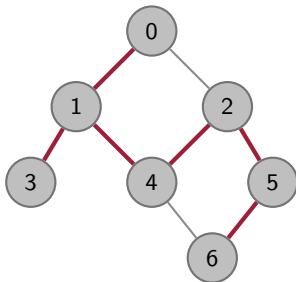
```

DFS(0)
├── DFS(1)
│   ├── DFS(3)
│   └── DFS(4)
│       ├── DFS(2)
│       └── DFS(5)
│           └── DFS(6)
    
```

visit order

0 1 3 4 2 5 6

DFS — Example: DFS(0)



recursive calls

```

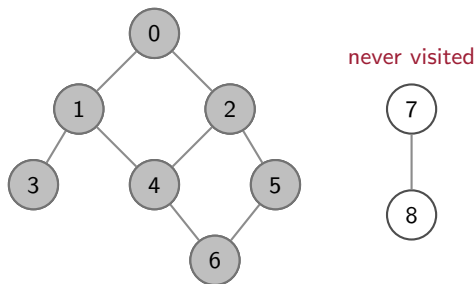
DFS(0)
├── DFS(1)
│   ├── DFS(3)
│   └── DFS(4)
│       ├── DFS(2)
│       └── DFS(5)
│           └── DFS(6)
  
```

visit order

0 1 3 4 2 5 6

all calls returned (grey): every vertex reachable from 0 is visited

DFS Only Reaches What Is Connected



- there is no path from 0 to 7 or 8 $\Rightarrow$ DFS(0) never visits them
- `visited[v]` after DFS(s) $\iff$ v is **connected** to s

Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

Your Task

Breadth First Search (§4.2.3)

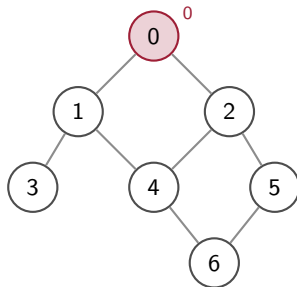
- visit **layer by layer**
- uses a **queue**; $\text{dist}[v] = \text{layer of } v$
- like DFS: vertices not connected to s keep $\text{dist} = \infty$

```

1 BFS(s):
2    $\text{dist}[*] \leftarrow \infty$ ;  $\text{dist}[s] \leftarrow 0$ ;  $Q \leftarrow [s]$ 
3   while Q not empty:
4      $u \leftarrow Q.\text{pop\_front}()$ 
5     for each v in AL[u]:
6       if  $\text{dist}[v] = \infty$ :
7          $\text{dist}[v] \leftarrow \text{dist}[u] + 1$ 
8         Q.push_back(v)

```

BFS — Example: BFS(0)



start: queue = [0], dist[0] = 0

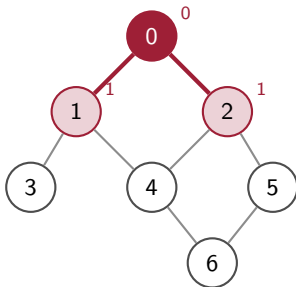
queue

0

visit order

-

BFS — Example: BFS(0)



pop 0 $\Rightarrow$ push 1, 2 (dist 1)

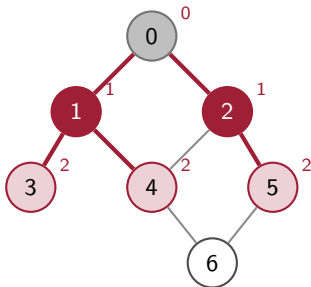
queue

1 2

visit order

0

BFS — Example: BFS(0)



queue

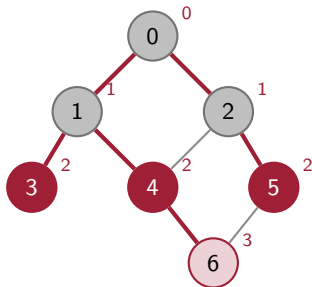
3 4 5

visit order

0 1 2

pop 1, 2 $\Rightarrow$ push 3, 4, 5 (dist 2)

BFS — Example: BFS(0)



pop 3, 4, 5 $\Rightarrow$ push 6 (dist 3)

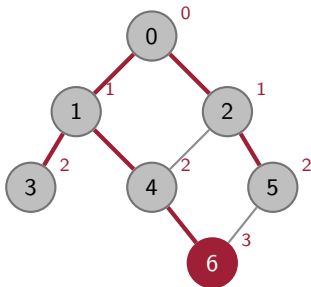
queue

6

visit order

0 1 2 3 4 5

BFS — Example: BFS(0)



pop 6 $\Rightarrow$ queue empty, done

queue

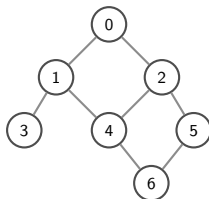
empty

visit order

0 1 2 3 4 5 6

DFS vs. BFS

	DFS	BFS
remembers via	recursion	queue
order from 0	0 1 3 4 2 5 6	0 1 2 3 4 5 6
time	$O(V + E)$	$O(V + E)$
bonus	short code	shortest paths (unweighted)



Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

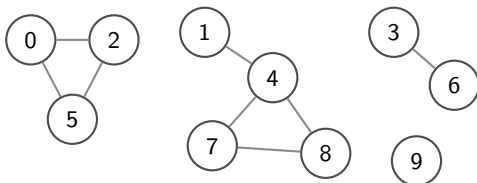
Your Task

Connected Components — The Problem (§4.2.4)

Problem

Given an **undirected** graph, find all its **connected components**.

- connected component = a maximal group of vertices where every two are joined by a path



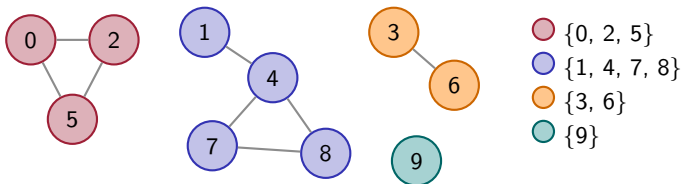
How many components? Which vertices are together?

Connected Components — The Problem (§4.2.4)

Problem

Given an **undirected** graph, find all its **connected components**.

- connected component = a maximal group of vertices where every two are joined by a path



4 components

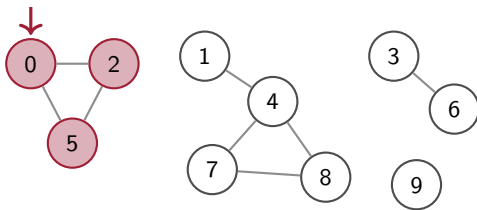
Connected Components — Algorithm

- DFS(u) visits **exactly** the component of u
- vertex still unvisited $\Rightarrow$ it starts a **new** component

```
1 numCC  $\leftarrow$  0
2 for u  $\leftarrow$  0 to V-1:
3     if not visited[u]:
4         numCC  $\leftarrow$  numCC + 1
5         DFS(u)           // marks the whole component
```

- every vertex/edge visited once $\Rightarrow O(V + E)$

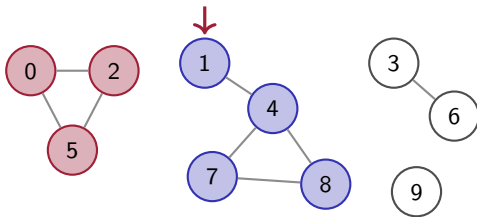
Connected Components — Execution



numCC = 1

$u = 0$ unvisited $\Rightarrow$ new component, DFS(0)

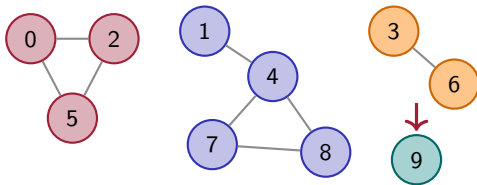
Connected Components — Execution



numCC = 2

$u = 1$ unvisited $\Rightarrow$ new component, DFS(1)

Connected Components — Execution



numCC = 4

then DFS(3) and DFS(9) $\Rightarrow$ **4** components

Together: Money Matters

open.kattis.com/problems/moneymatters

Problem

- n friends after a trip: friend i has balance o_i (> 0 : must receive money, < 0 : must pay); all balances sum to 0
- after some fights, only m friendships are left
- money can only be given between two people who are **still friends**
- can everybody end up with balance 0?

Input / Output

- n m ($n \leq 10^4$, $m \leq 5 \cdot 10^4$), then n lines o_i , then m lines x y
- print POSSIBLE or IMPOSSIBLE

Money Matters — Example

Sample 1

- $n = 5$, balances $o = [100, -75, -25, -42, 42]$
 - friendships = $\{(0, 1), (1, 2), (3, 4)\}$
- output:

Sample 2

- $n = 4$, balances $o = [15, 20, -10, -25]$
 - friendships = $\{(0, 2), (1, 3)\}$
- output:

Money Matters — Example

Sample 1

- $n = 5$, balances $o = [100, -75, -25, -42, 42]$
 - friendships = $\{(0, 1), (1, 2), (3, 4)\}$
- output: POSSIBLE

Sample 2

- $n = 4$, balances $o = [15, 20, -10, -25]$
 - friendships = $\{(0, 2), (1, 3)\}$
- output: IMPOSSIBLE

Money Matters — Hints

Hint 1

Money can travel along a chain of friends. Between which people can money move?

Money Matters — Hints

Hint 1

Money can travel along a chain of friends. Between which people can money move?

Hint 2

Inside one connected component, when can everybody get to 0?

Money Matters — Hints

Hint 1

Money can travel along a chain of friends. Between which people can money move?

Hint 2

Inside one connected component, when can everybody get to 0?

Hint 3

Modify $\text{DFS}(u)$ so that it also returns something useful for the component.

Money Matters — Solution

- money moves freely *inside* a component, never *between* components
- **POSSIBLE** $\iff$ the balances of **every component** sum to 0

```

1 SUM(u):                                     // DFS + component sum
2     visited[u] ← true;  s ← o[u]
3     for each v in AL[u]:
4         if not visited[v]:  s ← s + SUM(v)
5     return s
6
7 for u ← 0 to n-1:
8     if not visited[u] and SUM(u) ≠ 0:
9         print IMPOSSIBLE; stop
10 print POSSIBLE

```

- $n = 10^4$: a path can make the recursion deep $\Rightarrow$ BFS is safer

Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

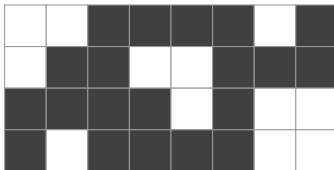
Your Task

Flood Fill — The Problem (§4.2.5)

- a **grid** is an $R \times C$ table of cells; cell (r, c) is in row r , column c and has a colour (here white or black)
- a **region** is a maximal set of cells of the same colour where you can go from any cell to any other moving only **horizontally or vertically** through that colour

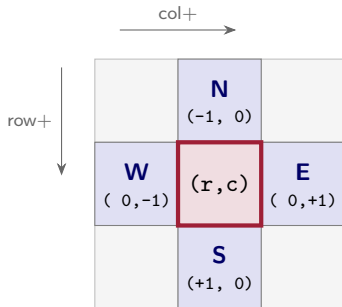
Problem

Given a grid of white and black cells, find all **regions** of white cells.



How many white regions?

The Grid Is a Graph: Neighbours



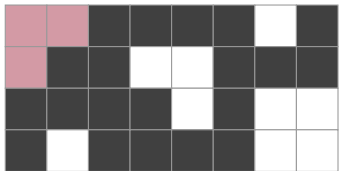
- cell = vertex; edges to the 4 cells around it
- each neighbour = a **move**: (rows, columns)
- store the moves in two arrays:
 $dr = [1, 0, -1, 0]$
 $dc = [0, 1, 0, -1]$
 (S, E, N, W)
- neighbour d:
 $(r + dr[d], c + dc[d]) \Rightarrow$ one loop instead of 4 ifs

Flood Fill — Algorithm

- region = connected component of white cells $\Rightarrow$ same loop as for CC

```
1 FLOODFILL(r, c):
2   if (r,c) outside or g[r][c]  $\neq$  white: return
3   g[r][c]  $\leftarrow$  black // mark visited
4   for d  $\leftarrow$  0 to 3:
5     FLOODFILL(r + dr[d], c + dc[d])
6
7 regions  $\leftarrow$  0
8 for each cell (r, c):
9   if g[r][c] = white:
10    regions  $\leftarrow$  regions + 1; FLOODFILL(r, c)
```

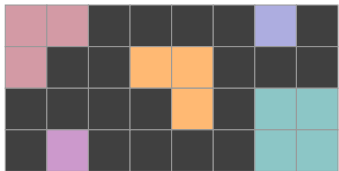
Flood Fill — Execution



regions = 1

cell (0,0) is white and unvisited $\Rightarrow$ new region, FLOODFILL(0,0)

Flood Fill — Execution



regions = 5

repeat for every unvisited white cell $\Rightarrow$ **5** regions

Together: Coast Length

`open.kattis.com/problems/coast`

Problem

- map of $N \times M$ squares of 1 km: 0 = water, 1 = land
- the map is surrounded by sea; **sea** = water connected to the edge of the map through water
- water enclosed by land is a **lake**, not sea
- compute the total length of **sea coast**: borders between land and sea

Input / Output

- N M (≤ 1000), then N strings of M digits
- print the coast length in km

Coast Length — Example

Input

5 6

011110

010110

111000

000010

000000

Output

0	1	1	1	1	0
0	1	0	1	1	0
1	1	1	0	0	0
0	0	0	0	1	0
0	0	0	0	0	0

Which water is sea and which is a lake? Where is the coast?

Coast Length — Example

Input

5 6

011110

010110

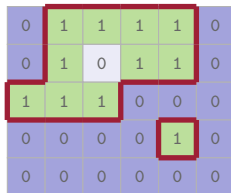
111000

000010

000000

Output

20



red = coast (20 sides). The 0 at row 1, column 2 is a lake: its borders do **not** count.

Coast Length — Hints

Hint 1

First find the sea. Which cell could you start a flood fill from? What if the sea is split into several pieces along the border?

Coast Length — Hints

Hint 1

First find the sea. Which cell could you start a flood fill from? What if the sea is split into several pieces along the border?

Hint 2

Add a frame of 0s around the map: now all the sea is **one** region.

Coast Length — Hints

Hint 1

First find the sea. Which cell could you start a flood fill from? What if the sea is split into several pieces along the border?

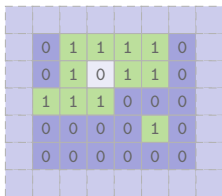
Hint 2

Add a frame of 0s around the map: now all the sea is **one** region.

Hint 3

Once the sea is marked, how do you count the coast using dx/dc ?

Coast Length — Solution



dashed = added frame; dark blue = sea

- 1 frame the map with 0s
- 2 flood fill the sea from (0,0) (up to 10^6 cells: use BFS)
- 3 for every land cell, count its 4 neighbours that are sea

```

1  add a frame of '0' around g          // (N+2) x (M+2)
2  FLOODFILL(0, 0, '0', 'S')           // recolour sea: 'S'
3  coast ← 0
4  for each (r, c) with g[r][c] = '1':
5      for d ← 0 to 3:
6          if g[r+dr[d]][c+dc[d]] = 'S':
7              coast ← coast + 1
8  print coast

```

Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

Your Task

Summary

- traversal = visit all vertices reachable from s , $O(V + E)$
- **DFS**: recursion, deep first
- **BFS**: queue, layer by layer, gives distances
- **CC**: restart DFS/BFS from every unvisited vertex
- **flood fill**: the same on a grid

Graph Traversal

Introduction

DFS

BFS

Connected Components

Flood Fill

Summary

Your Task

Your Task

- Birthday Party (CC): open.kattis.com/problems/birthday
- Terraces (flood fill): open.kattis.com/problems/terraces

References

- Sections 4.2.1–4.2.5 in Halim, Halim, and Effendy. *Competitive Programming 4*. Book 1.
- <https://visualgo.net/en/dfsbf>