Graphs

Modelling connections: BFS, DFS, shortest paths and classic problems

Dr. Quang-Vinh Dang
Senior Lecturer (Assoc. Prof)

Algorithms and Programming Concepts | Year 1 | Python 3.14

British University Vietnam

Graphs model connections

A **graph** is a set of things (**nodes**) and the links between them (**edges**). It is the most flexible structure in computing:

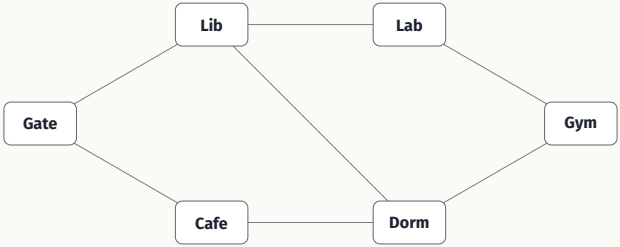
- **maps**: places and roads (Google Maps);
- **social networks**: people and friendships;
- **the web**: pages and hyperlinks;
- **dependencies**: which course needs which;
- **networks**: routers and cables.

Lists and trees are special cases of graphs.

After this lecture you can

- represent a graph with a dictionary (adjacency list) or a matrix;
- explore it with **BFS** and **DFS**, step by step;
- find the **fewest-hops** route and the **cheapest** route (Dijkstra);
- find **components**, **cycles** and a **topological order**;
- solve classic problems: islands, mazes, word ladders, bipartite check, spanning trees.

Our running example: a small campus



vertex / node a place: Gate, Lib, …

edge a footpath between two places.

neighbours nodes joined by an edge: Lib's neighbours are Gate, Lab, Dorm.

degree number of neighbours: Dorm has 3.

path a walk along edges.

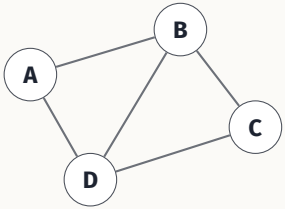
cycle a path back to its start.

connected every node can reach every other.

This graph is **undirected** (paths work both ways) and, later, **weighted** (each path takes some minutes).

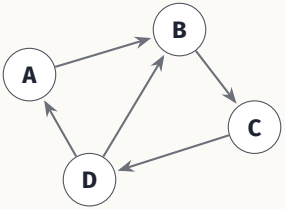
Kinds of graph

undirected



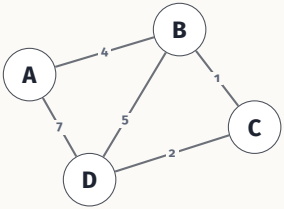
Undirected: an edge works both ways. Friendship, footpaths.

directed



Directed: an edge has a direction. Follows on a social network, one-way streets, “course A is needed for B”.

weighted



Weighted: each edge has a number (minutes, kilometres, cost). Shortest path then means smallest total weight.

Two ways to store a graph

```
graph = {
    "Gate": ["Lib", "Cafe"],
    "Lib":  ["Gate", "Lab", "Dorm"],
    "Cafe": ["Gate", "Dorm"],
    "Lab":  ["Lib", "Gym"],
    "Dorm": ["Lib", "Cafe", "Gym"],
    "Gym":  ["Lab", "Dorm"],
}
```

```
Gate  -> ['Lib', 'Cafe']
Lib   -> ['Gate', 'Lab', 'Dorm']
Cafe  -> ['Gate', 'Dorm']
Lab   -> ['Lib', 'Gym']
Dorm  -> ['Lib', 'Cafe', 'Gym']
Gym   -> ['Lab', 'Dorm']
```

Adjacency list: a dict from each node to the list of its neighbours.

	Gate	Lib	Cafe	Lab	Dorm	Gym
Gate	0	1	1	0	0	0
Lib	1	0	0	1	1	0
Cafe	1	0	0	0	1	0
Lab	0	1	0	0	0	1
Dorm	0	1	1	0	0	1
Gym	0	0	0	1	1	0

Adjacency matrix: a table with 1 where an edge exists (built by the code below).

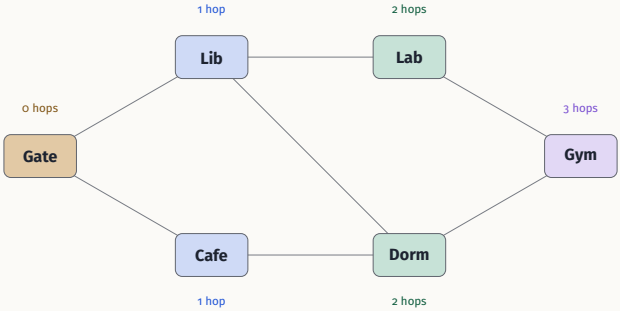
```
names = list(graph)
index = {n: i for i, n in enumerate(names)}
matrix = [[0] * len(names) for _ in names]
for u, nbrs in graph.items():
    for v in nbrs:
        matrix[index[u]][index[v]] = 1
```

Which representation?

	Adjacency list	Adjacency matrix
Memory	$O(V + E)$	$O(V^2)$
Is there an edge $u-v$?	scan u's list: $O(\text{degree})$	one lookup: $O(1)$
List all neighbours of u	$O(\text{degree})$	scan a row: $O(V)$
Best for	**sparse** graphs (most real ones)	**dense** graphs, quick edge tests

A social network with a billion people where each knows a few hundred others has $E \approx 10^{11}$ but $V^2 = 10^{18}$ cells. The matrix would not fit in any computer, so real systems use adjacency lists. We use them from now on.

Breadth-first search (BFS): explore in layers



Start at the Gate. Visit everything **1 hop** away, then everything **2 hops** away, and so on: the search spreads like a ripple.

To do this we keep a **queue** (first in, first out): neighbours are added at the back, and we always take the node at the front.

Bonus: the first time BFS reaches a node, it has used the **fewest possible edges**. So BFS finds shortest paths in an *unweighted* graph.

BFS in Python

```python
from collections import deque


def bfs(graph, start):
    parent = {start: None}
    queue = deque([start])
    order = []
    while queue:
        u = queue.popleft()
        order.append(u)
        for v in graph[u]:
            if v not in parent:
                parent[v] = u
                queue.append(v)
    return order, parent
```

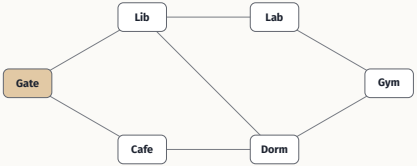
`parent` does two jobs: it remembers which nodes we have seen (so none is queued twice), and records *how we got there*.

```python
def path_to(parent, target):
    path = []
    while target is not None:
        path.append(target)
        target = parent[target]
    return path[::-1]
```

```
visit order: Gate Lib Cafe Lab Dorm Gym
path to Gym: Gate-Lib-Lab-Gym
0 hops: Gate
1 hops: Lib Cafe
2 hops: Lab Dorm
3 hops: Gym
```

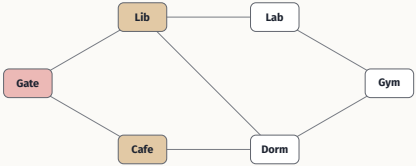
Following `parent` back from Gym gives the route. Each node is queued once and each edge examined once: $O(V + E)$.

BFS step by step (1/2)



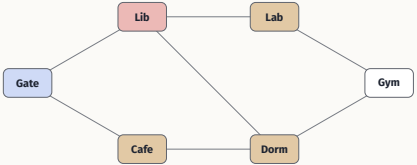
start: queue the source

queue: Gate



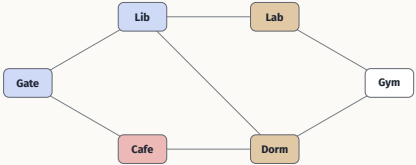
1. pop Gate

queue: Lib Cafe



2. pop Lib

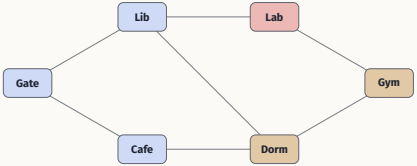
queue: Cafe Lab Dorm



3. pop Cafe

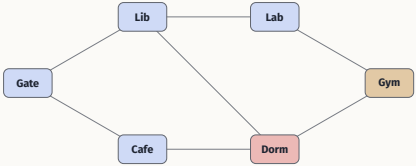
queue: Lab Dorm

BFS step by step (2/2)



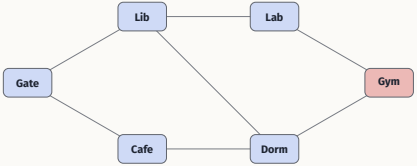
4. pop Lab

queue: Dorm Gym



5. pop Dorm

queue: Gym



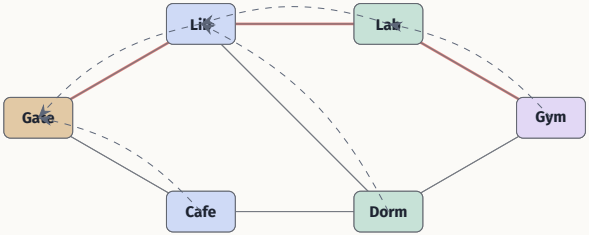
6. pop Gym

queue: (empty)

Colour key

not seen yet | waiting | being processed now | finished

BFS layers and the shortest path in an unweighted graph



path found: Gate, Lib, Lab, Gym

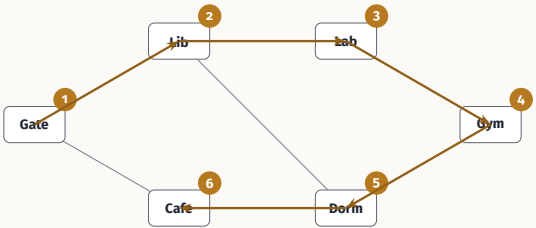
dashed arrows: each node points to its `parent`.

BFS finishes layer k before it starts layer $k+1$, so the layer number is the **hop distance** from the source.

To get the route, start at the target and follow `parent` back to the source, then reverse the list (`path_to`).

Not unique: Gate–Cafe–Dorm–Gym also has 3 hops. BFS reports the one it met first. When edges have costs, hops are the wrong measure (see Dijkstra below).

Depth-first search (DFS): go deep, then backtrack



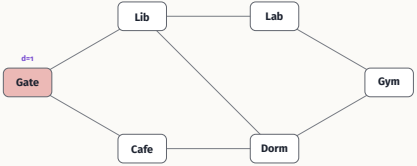
Follow one path as far as it goes; when you hit a dead end (or a place already visited), **backtrack** to the last choice and try the next neighbour.

```python
def dfs(graph, u, seen=None):
    if seen is None:
        seen = []
    seen.append(u)
    for v in graph[u]:
        if v not in seen:
            dfs(graph, v, seen)
    return seen
```

```
DFS order  : Gate Lib Lab Gym Dorm Cafe
iterative  : Gate Lib Lab Gym Dorm Cafe
```

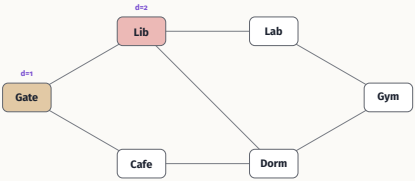
Recursion *is* the stack: each call remembers where to continue. Also $O(V + E)$. DFS does not find shortest paths, but it is ideal for “can I get there?”, cycles, mazes and ordering tasks.

DFS step by step (1/2): discovery times



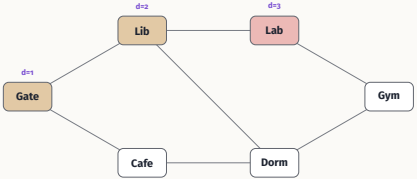
1. discover Gate

stack: Gate



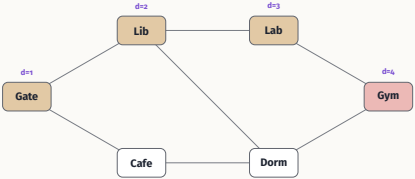
2. discover Lib

stack: Gate Lib



3. discover Lab

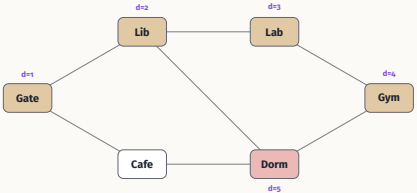
stack: Gate Lib Lab



4. discover Gym

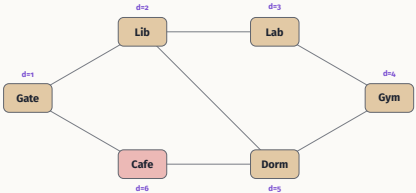
stack: Gate Lib Lab Gym

DFS step by step (2/2): backtracking and finish times



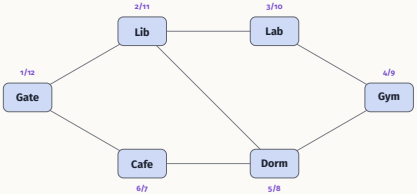
5. discover Dorm

stack: Gate Lib Lab Gym Dorm



6. discover Cafe

stack: Gate Lib Lab Gym Dorm Cafe



backtrack: found/done times

done order: Cafe Dorm Gym Lab Lib Gate

Colour key

not seen yet | waiting | being processed now | finished

DFS discovery and finish order

```
def dfs_times(graph, start):
    clock = 0
    found, done = {}, {}

    def visit(u):
        nonlocal clock
        clock += 1
        found[u] = clock
        for v in graph[u]:
            if v not in found:
                visit(v)
        clock += 1
        done[u] = clock

    visit(start)
    return found, done
```

```
node  found  done
Gate      1    12
Lib       2    11
Lab       3    10
Gym       4     9
Dorm      5     8
Cafe      6     7
done order: Cafe Dorm Gym Lab Lib Gate
```

A clock ticks at every **discovery** and every **finish**. The two numbers of a node form an interval, and the intervals are **nested**: Gate $[1, 12]$ contains Lib $[2, 11]$, which contains Lab $[3, 10]$, and so on.

If v is discovered while u is still open, then v finishes first: that is the recursion stack.

Why care? The **reverse finish order** of a DFS on a DAG is a topological order.

Queue versus stack: one loop, two behaviours

BFS: the to-do list is a **queue** (front first):

```
pop Gate  queue: Lib Cafe
pop Lib   queue: Cafe Lab Dorm
pop Cafe  queue: Lab Dorm
pop Lab   queue: Dorm Gym
pop Dorm  queue: Gym
pop Gym   queue: (empty)
```

Nodes are served in the order they were discovered: the whole layer of distance 1 before any node at distance 2.

DFS: the to-do list is a **stack** (the open calls):

```
enter Gate  stack: Gate
enter Lib   stack: Gate Lib
enter Lab   stack: Gate Lib Lab
enter Gym   stack: Gate Lib Lab Gym
enter Dorm  stack: Gate Lib Lab Gym Dorm
enter Cafe  stack: Gate Lib Lab Gym Dorm Cafe
leave Cafe  stack: Gate Lib Lab Gym Dorm
leave Dorm  stack: Gate Lib Lab Gym
leave Gym   stack: Gate Lib Lab
leave Lab   stack: Gate Lib
leave Lib   stack: Gate
leave Gate  stack: (empty)
```

The newest node is always continued first; older nodes wait at the bottom until everything above them has finished.

DFS without recursion: an explicit stack

```python
def dfs_iter(graph, start):
    seen, order, stack = set(), [], [start]
    while stack:
        u = stack.pop()
        if u in seen:
            continue
        seen.add(u)
        order.append(u)
        stack.extend(reversed(graph[u]))
    return order
```

Replace the call stack by a `list`. Pushing the neighbours **reversed** makes the first neighbour the first to come out, so we get the *same* order as the recursive version (see the output of the DFS slide: `iterative` line).

`if u in seen: continue` is needed because a node can be pushed several times before it is popped.

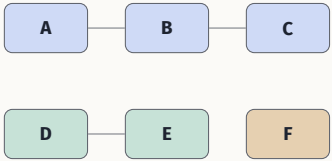
Why bother? Python limits recursion to about 1 000 calls. A DFS on a graph with a long path (a maze with a million cells) would crash recursively, but works with a list.

Swap the `list` for a `deque` and `pop()` for `popleft()` and the same loop becomes BFS.

Connected components: separate islands

```
def components(graph):
    seen, result = set(), []
    for start in graph:
        if start not in seen:
            comp = dfs(graph, start)
            seen.update(comp)
            result.append(comp)
    return result
```

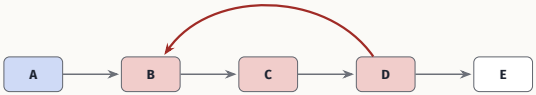
```
A B C
D E
F
```



A **component** is a group of nodes that can all reach each other. Start a DFS from any unseen node: everything it reaches is one component; repeat until all nodes are seen.

Uses: “are these two accounts connected?”, finding islands in an image, checking that a network is not split in two.

Cycle detection in a directed graph: three colours



back edge $D \to B$: B is still *open*

During a DFS every node is in one of three states:

- **white**: not seen yet;
- **grey**: on the current path (the call is still open);
- **black**: finished.

An edge to a **grey** node closes a loop: you came from that node, and now you can reach it again. That is a **cycle**.

An edge to a **black** node is harmless: that part of the graph is finished and never led back.

Why it matters: if tasks depend on each other in a cycle (B needs C, C needs D, D needs B), nothing can ever start. Build tools, spreadsheets and package managers all check for cycles.

Same $O(V + E)$ cost as DFS.

Cycle detection in Python

```python
def find_cycle(graph):
    state, path = {}, []   # 1 = on path, 2 = finished

    def visit(u):
        state[u] = 1
        path.append(u)
        for v in graph[u]:
            if state.get(v) == 1:
                return path[path.index(v):] + [v]
            if v not in state and (c := visit(v)):
                return c
        state[u] = 2
        path.pop()

    for u in graph:
        if u not in state and (c := visit(u)):
            return c
    return None
```

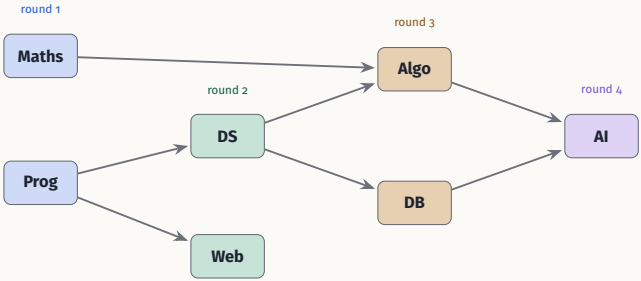
```
with D->B : ['B', 'C', 'D', 'B']
after fix : None
```

`path` holds the grey nodes in order. When an edge leads back to a node on the path, the slice `path[path.index(v):]` is the cycle itself.

`:=` (the walrus operator) assigns and tests in one line.

Remove the edge `D`→`B` and the function returns `None`.

Topological sort: an order that respects dependencies



An arrow $X \to Y$ means "take X before Y". Colours: the round in which Kahn's algorithm frees each course.

Kahn's algorithm

1. Count the **in-degree** of each node: how many prerequisites it still waits for.
2. Put every node with in-degree 0 in a queue.
3. Take one out, add it to the order, and lower the in-degree of its successors. Any that reach 0 join the queue.
4. If the order is shorter than the graph, there was a **cycle**.

Cost: $O(V + E)$.

Topological sort in Python

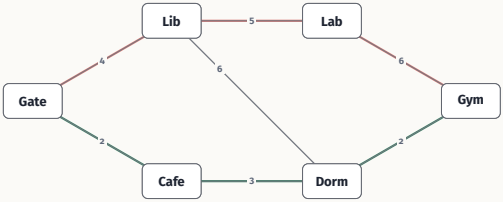
```python
def topo_sort(graph):
    indeg = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            indeg[v] += 1
    ready = deque(u for u, d in indeg.items()
                  if d == 0)
    order = []
    while ready:
        u = ready.popleft()
        order.append(u)
        for v in graph[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                ready.append(v)
    if len(order) < len(graph):
        return None
    return order
```

```
in-degrees: Maths=0 Prog=0 DS=1 Algo=2
            DB=1 Web=1 AI=2
take Maths freed: -
take Prog  freed: DS Web
take DS    freed: Algo DB
take Web   freed: -
take Algo  freed: -
take DB    freed: AI
take AI    freed: -
order: Maths Prog DS Web Algo DB AI
with AI -> Maths: None
```

`Prog` frees `DS` and `Web`; `DS` frees `Algo` (`Maths` is already done) and `DB`.

Many valid orders exist; this is the first the queue produces. Making `AI` a prerequisite of `Maths` creates a cycle: the function returns `None`.

Fewest hops is not the same as quickest



Both routes from Gate to Gym have **3 edges**, so BFS cannot tell them apart. It returned Gate–Lib–Lab–Gym: **red** $4 + 5 + 6 = \mathbf{15}$ min, but **green** takes $2 + 3 + 2 = \mathbf{7}$ min.

The weighted graph in Python: a dict of dicts (neighbour → minutes to walk).

```
roads = {
    "Gate": {"Lib": 4, "Cafe": 2},
    "Lib":  {"Gate": 4, "Lab": 5, "Dorm": 6},
    "Cafe": {"Gate": 2, "Dorm": 3},
    "Lab":  {"Lib": 5, "Gym": 6},
    "Dorm": {"Lib": 6, "Cafe": 3, "Gym": 2},
    "Gym":  {"Lab": 6, "Dorm": 2},
}
```

When edges have different costs we need an algorithm that adds up the *weights*: **Dijkstra's algorithm**.

Dijkstra's algorithm: settle the closest node first

1. Give every node a distance: 0 for the start, ∞ for the rest.
2. Take the **unsettled node with the smallest distance**. Its distance is now final.
3. **Relax** its edges: if going through it makes a neighbour closer, update that neighbour.
4. Repeat until everything is settled.

A **heap** (priority queue) gives "the closest node" in $O(\log n)$. Why is it correct? All weights are ≥ 0, so a longer walk can never become shorter.

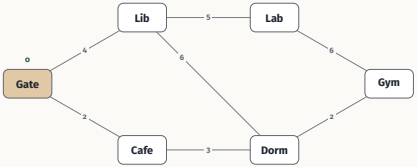
Negative weights break this argument: use Bellman–Ford.

The order in which the real code settles the nodes:

```
settle Gate  at  0
settle Cafe  at  2
settle Lib   at  4
settle Dorm  at  5
settle Gym   at  7
settle Lab   at  9
```

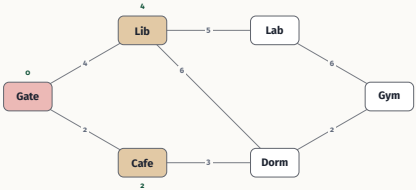
Distances never decrease: 0, 2, 4, 5, 7, 9. The next two slides draw each step.

Dijkstra step by step (1/2)



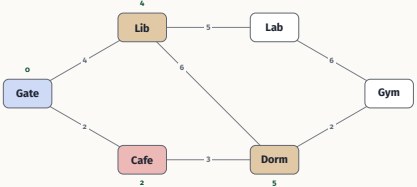
start: dist[Gate] = 0

heap: (0,Gate)



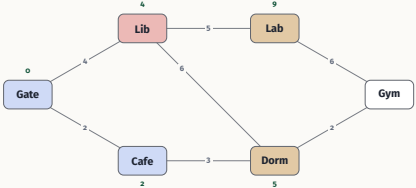
1. settle Gate (0)

heap: (2,Cafe) (4,Lib)



2. settle Cafe (2)

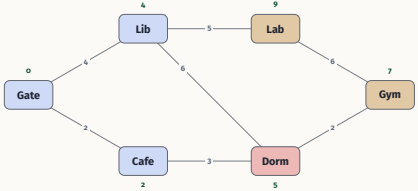
heap: (4,Lib) (5,Dorm)



3. settle Lib (4)

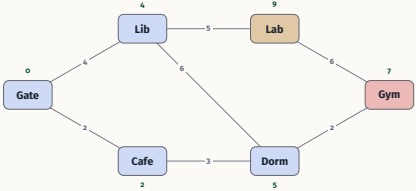
heap: (5,Dorm) (9,Lab)

Dijkstra step by step (2/2)



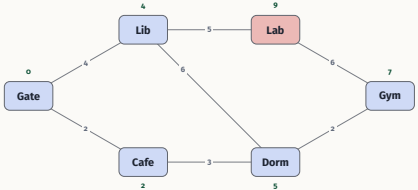
4. settle Dorm (5)

heap: (7,Gym) (9,Lab)



5. settle Gym (7)

heap: (9,Lab)



6. settle Lab (9)

heap: (empty)

Colour key

not seen yet | waiting | being processed now | finished

Dijkstra: the distance table

step	Gate	Lib	Cafe	Lab	Dorm	Gym	settled
0	0	∞	∞	∞	∞	∞	–
1	0	4	2	∞	∞	∞	Gate
2	**0**	4	2	∞	5	∞	Cafe
3	**0**	4	**2**	9	5	∞	Lib
4	**0**	**4**	**2**	9	5	7	Dorm
5	**0**	**4**	**2**	9	**5**	7	Gym
6	**0**	**4**	**2**	9	**5**	**7**	Lab

Bold numbers are final (settled). Read down a column to see a distance improve: `Dorm` is first estimated at 5 (via Cafe) and never changes; `Lib`'s route to `Dorm` (4+6=10) is worse, so the update is rejected. `Lab` gets 9 via Lib; the later offer 7+6=13 from Gym is rejected too.

Dijkstra in Python

```python
from heapq import heappop, heappush

INF = float("inf")

def dijkstra(roads, start):
    dist = {start: 0}
    parent = {start: None}
    heap = [(0, start)]
    while heap:
        d, u = heappop(heap)
        if d > dist[u]:
            continue
        for v, w in roads[u].items():
            nd = d + w
            if nd < dist.get(v, INF):
                dist[v] = nd
                parent[v] = u
                heappush(heap, (nd, v))
    return dist, parent
```

```
minutes from Gate:
Gate=0 Cafe=2 Lib=4 Dorm=5 Gym=7 Lab=9
fewest hops: Gate-Lib-Lab-Gym (15 min)
cheapest   : Gate-Cafe-Dorm-Gym (7 min)
```

`dist[v]` is the best time found so far to reach `v`; `parent` rebuilds the route, as in BFS.

`if d > dist[u]: continue` skips *stale* heap entries: a node may be pushed several times, and only the smallest distance matters.

Cost: each edge may push once, each push or pop is $O(\log n)$: $O((V + E) \log V)$.

Problem 1: number of islands




```
1100010
1100010
0001000
0011001
1000001
1000111
islands: 5
sizes  : #1:4 #2:2 #3:3 #4:5 #5:2
```

Problem. A grid of `1` (land) and `0` (water). Count the islands: groups of land cells joined **up, down, left or right**.

Idea. The grid is a graph: each cell is a node, and neighbouring land cells are joined by an edge. An island is a **connected component**. Scan the grid; at each unlabelled land cell, start a DFS that gives every cell it reaches the same label.

Diagonal cells do not count: in the picture the single cells of islands 4 and 5 touch only diagonally.

Cost: every cell is labelled once, so $O(\text{rows} \times \text{cols})$.

Problem 1: number of islands, solution i

```
def grid_neighbours(grid, r, c):
    for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < len(grid) and 0 <= nc < len(grid[0]):
            yield nr, nc
```

`grid_neighbours` is a **generator**: it yields the (up to four) cells next to `(r, c)` that lie inside the grid. The graph is never stored: neighbours are computed on demand.

In `islands`, `label` maps each land cell to its island number and doubles as the “seen” set. Counting the distinct labels gives the answer: 5 islands, with sizes 4, 2, 3, 5 and 2 (output on the previous slide).

Problem 1: number of islands, solution ii

```
def islands(grid):
    label = {}
    for r0, row in enumerate(grid):
        for c0, ch in enumerate(row):
            if ch != "1" or (r0, c0) in label:
                continue
            tag = len(set(label.values())) + 1
            label[r0, c0] = tag
            stack = [(r0, c0)]
            while stack:
                r, c = stack.pop()
                for nr, nc in grid_neighbours(grid, r, c):
                    if grid[nr][nc] == "1" and (nr, nc) not in label:
                        label[nr, nc] = tag
                        stack.append((nr, nc))
    return label
```

Problem 2: shortest path in a maze




```
S.#.....
.##.###.
....#...
.####.#.
......#E
steps: 15
moves: DDDDRRRRRUURRDD
```

Problem. `S` is the start, `E` the exit, `#` a wall. Find the **fewest steps** from `S` to `E`, moving up, down, left or right.

Idea. Every free cell is a node; all steps cost 1. That is exactly the **unweighted shortest path**: use **BFS**.

The numbers in the picture are the BFS distances from `S`: the search spreads as a ripple, and the first time it reaches `E` the path (yellow) is shortest. BFS stops there, so cells beyond it are not numbered.

Why not DFS? It would find *a* path, but usually not the shortest one.

Problem 2: maze solution i

```
def maze_bfs(maze, start, goal):
    parent = {start: None}
    queue = deque([start])
    while queue:
        r, c = queue.popleft()
        if (r, c) == goal:
            break
        for nr, nc in grid_neighbours(maze, r, c):
            if maze[nr][nc] != "#" and (nr, nc) not in parent:
                parent[nr, nc] = (r, c)
                queue.append((nr, nc))
    return parent
```

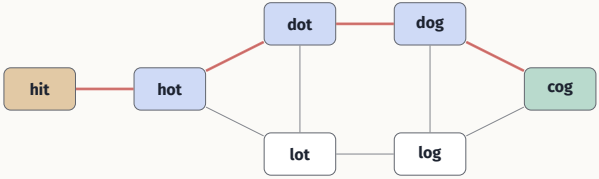
Same BFS as before, with cells instead of names. `parent` records where each cell was reached from; the loop stops as soon as the goal is dequeued.

Problem 2: maze solution ii

```
def solve_maze(maze):
    cells = {(r, c): ch for r, row in enumerate(maze)
             for c, ch in enumerate(row)}
    start = next(p for p, ch in cells.items() if ch == "S")
    goal = next(p for p, ch in cells.items() if ch == "E")
    parent = maze_bfs(maze, start, goal)
    if goal not in parent:
        return None, parent
    path = []
    while goal is not None:
        path.append(goal)
        goal = parent[goal]
    return path[::-1], parent
```

`solve_maze` finds `S` and `E`, runs the BFS and then walks the `parent` chain back from the goal, exactly like `path_to`. If the goal was never reached the maze has no solution and the function returns `None`.

Problem 3: word ladder



words differ in exactly one letter ⇒ edge

```
hit -> cog : hit hot dot dog cog
hit -> cat : None
```

Problem. Change `hit` into `cog`, one letter at a time, and every intermediate word must be in the dictionary. What is the shortest sequence?

Idea. Words are nodes; two words are joined if they differ in one letter. The graph is too big to build, but BFS only needs "the neighbours of w".

Trick: bucket the words by *pattern*. `hot`, `dot` and `lot` all fit `*ot`, so they are neighbours. Then each word has few patterns (one per letter) instead of comparing with every other word.

`hit`→`cat` is impossible: `cat` is not in the dictionary.

Problem 3: word ladder, solution i

```
def patterns(word):
    return [word[:i] + "*" + word[i+1:]
            for i in range(len(word))]
```

`patterns("hot")` gives `["*ot", "h*t", "ho*"]`: the word with one letter replaced by a star. Two words are neighbours exactly when they share a pattern.

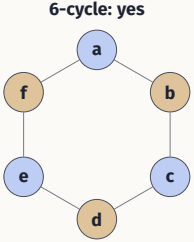
In `ladder`, **step 1** builds the `buckets` dict: pattern $\rightarrow$ words. **Step 2** is plain BFS; the neighbours of `w` are the words in the buckets of its patterns. When the goal is reached, `path_to` (from the BFS slides) rebuilds the ladder.

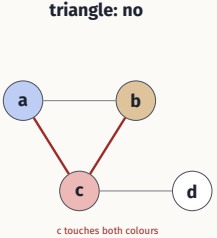
Cost: with n words of length L there are nL patterns, each built in $O(L)$: about $O(nL^2)$ in all.

Problem 3: word ladder, solution ii

```
def ladder(start, goal, words):
    buckets = {}
    for w in [start, *words]:
        for p in patterns(w):
            buckets.setdefault(p, []).append(w)
    parent = {start: None}
    queue = deque([start])
    while queue:
        w = queue.popleft()
        if w == goal:
            return path_to(parent, goal)
        for p in patterns(w):
            for nxt in buckets[p]:
                if nxt not in parent:
                    parent[nxt] = w
                    queue.append(nxt)
    return None
```

Problem 4: is the graph bipartite? (two-colouring)






```
6-cycle : a=0 b=1 c=0 d=1 e=0 f=1
triangle: None
```

Problem. Can the nodes be split into two groups so that every edge joins *different* groups? (Teams with no friends inside a team, jobs that clash with each other…)

Idea. Colour the start node 0. BFS: give each new neighbour the **opposite** colour. If we ever meet an edge whose two ends have the **same** colour, the answer is no.

Fact: a graph is bipartite exactly when it has **no odd cycle**. The triangle has 3 edges, so it fails.

Problem 4: two-colouring, solution

```
def two_colour(graph):
    colour = {}
    for s in graph:
        if s in colour:
            continue
        colour[s] = 0
        queue = deque([s])
        while queue:
            u = queue.popleft()
            for v in graph[u]:
                if v not in colour:
                    colour[v] = 1 - colour[u]
                    queue.append(v)
                elif colour[v] == colour[u]:
                    return None
    return colour
```

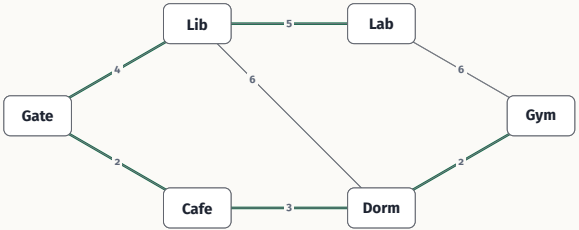
The outer `for` restarts from every unvisited node, so **disconnected** graphs are handled too.

`colour` doubles as the visited set. `1 - colour[u]` flips 0 and 1.

`elif colour[v] == colour[u]`: an edge inside one group. We return `None` at once.

Cost: BFS, so $O(V + E)$. Output: see the previous slide.

Problem 5: minimum spanning tree (Kruskal)




```
Cafe-Gate  2: take
Dorm-Gym   2: take
Cafe-Dorm  3: take
Gate-Lib   4: take
Lab-Lib    5: take
Dorm-Lib   6: skip (cycle)
Gym-Lab    6: skip (cycle)
total weight: 16 (all paths: 28)
```

Problem. Connect all places with the **cheapest set of paths** (a *spanning tree*: connected, no cycle). Used for cables, pipes, road repair.

Kruskal's idea: sort the edges by weight; take an edge **unless it would close a cycle**.

Cycle test: keep a **union-find** structure. Every node points to a "leader" of its group; an edge joins two groups only if the leaders differ.

Lib–Dorm (6) and Lab–Gym (6) are rejected: their ends are already connected.

Problem 5: Kruskal and union-find, solution i

```
def find(parent, x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]
        x = parent[x]
    return x
```

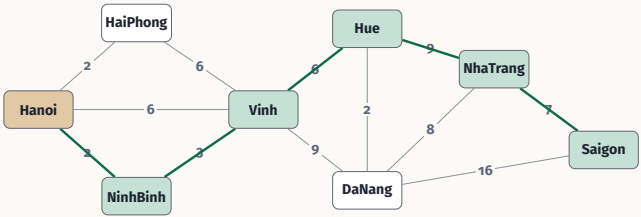
`find` follows parent links to the group leader. **Path halving** (`parent[x] = parent[parent[x]]`) shortens the chain each time, so later finds are almost $O(1)$.

Problem 5: Kruskal and union-find, solution ii

```
def kruskal(nodes, edges):
    parent = {u: u for u in nodes}
    tree, total = [], 0
    for w, u, v in sorted(edges):
        ru, rv = find(parent, u), find(parent, v)
        if ru != rv:
            parent[ru] = rv
            tree.append((u, v, w))
            total += w
    return tree, total
```

Edges are `(weight, u, v)` tuples, so `sorted` orders them by weight. Sorting dominates: $O(E \log E)$. Output: previous slide.

Problem 6: Dijkstra on a road network




```
Hanoi      0 h via Hanoi
HaiPhong   2 h via Hanoi-HaiPhong
NinhBinh   2 h via Hanoi-NinhBinh
Vinh       5 h via Hanoi-NinhBinh-Vinh
Hue       11 h via Hanoi-NinhBinh-Vinh-Hue
DaNang    13 h via Hanoi-NinhBinh-Vinh-Hue-DaNang
NhaTrang  20 h via Hanoi-NinhBinh-Vinh-Hue-NhaTrang
Saigon    27 h via Hanoi-NinhBinh-Vinh-Hue-NhaTrang-Saigon
```

Problem. Travel times (hours) between 8 cities. How long from Hanoi to every other city, and by which route?

Solution. The `dijkstra` function from earlier works unchanged. Only the data is new: `undirected` turns a list of roads into a graph with both directions.

Green: the best route to Saigon, 27 hours. Direct roads can lose: Hanoi–Vinh takes 6 h, but Hanoi–Ninh Binh–Vinh takes $2 + 3 = 5$ h.

Problem 6: building the network i

```
def undirected(edges):
    g = {}
    for u, v, w in edges:
        g.setdefault(u, {})[v] = w
        g.setdefault(v, {})[u] = w
    return g
```

`undirected` stores each road twice: `g[u][v]` and `g[v][u]`. `setdefault(u, {})` creates the inner dict the first time a city is seen. Then `dijkstra(highways, "Hanoi")` returns `dist` and `parent`, and `path_to(parent, city)` rebuilds each route (the output on the previous slide).

Problem 6: building the network ii

```
highways = undirected([
    ("Hanoi", "HaiPhong", 2), ("Hanoi", "NinhBinh", 2),
    ("Hanoi", "Vinh", 6), ("NinhBinh", "Vinh", 3),
    ("HaiPhong", "Vinh", 6), ("Vinh", "Hue", 6),
    ("Vinh", "DaNang", 9), ("Hue", "DaNang", 2),
    ("Hue", "NhaTrang", 9), ("DaNang", "NhaTrang", 8),
    ("DaNang", "Saigon", 16), ("NhaTrang", "Saigon", 7),
])
```

Each tuple is `(city, city, hours)`. Real map services do the same on graphs with hundreds of millions of roads, adding tricks such as A* and precomputed shortcuts.

Which graph algorithm?

Algorithm	Use it to …	Cost	Data structure
BFS	fewest-edge path, layers, grids and mazes, bipartite check	$O(V + E)$	queue (`deque`)
DFS	reachability, components, cycles, islands	$O(V + E)$	recursion / stack
Topological sort	order tasks with dependencies, detect cycles	$O(V + E)$	queue + in-degrees
Dijkstra	cheapest path, weights ≥ 0	$O((V + E) \log V)$	heap (`heapq`)
Bellman–Ford	cheapest path with negative weights	$O(V \cdot E)$	arrays
Kruskal	minimum spanning tree	$O(E \log E)$	sort + union-find
A*	cheapest path to *one* target, using a guess of the remaining distance	often much faster than Dijkstra	heap

Recognising the problem is half the work: “fewest steps” → BFS, “can I reach it / how many groups” → DFS, “cheapest with costs” → Dijkstra, “what must come first” → topological sort.

Recap and practice

Take-aways

1. A graph is nodes plus edges; store it as an **adjacency list** (a dict of lists).
2. **BFS** (queue) explores in layers and finds the fewest-hop path; **DFS** (stack or recursion) dives deep.
3. DFS gives discovery/finish times, components, cycles and, in reverse, a topological order.
4. Grids and word puzzles are graphs in disguise: **model first**, then pick BFS or DFS.
5. With edge costs use **Dijkstra** (heap); for the cheapest connection of everything use **Kruskal**.
6. BFS and DFS cost $O(V + E)$; Dijkstra $O((V + E) \log V)$.

Practice

1. Add a node `Pool` connected to `Gym`. What changes in the BFS output?
2. Write `has_path(graph, a, b)` using BFS or DFS.
3. Make Dijkstra return the distance to *one* target and stop early.
4. Does Dijkstra still work if you change one road to 0 minutes? To -1? Why?
5. Draw the BFS queue by hand for a 3×3 grid started at a corner.
6. Add the edge `D`$\rightarrow$`A` to the cycle example. What cycle is reported?

Challenge problems

1. **Max island area**: return the size of the largest island. Then count islands when *diagonal* neighbours also connect.
2. **Rotting oranges**: in a grid, every minute each rotten orange (2) rots its fresh (1) neighbours. How many minutes until none is fresh, or -1 if impossible? (Multi-source BFS.)
3. **Course schedule II**: return the order of courses, or an empty list if the prerequisites contain a cycle. How many *semesters* do you need if you can take any number of courses per semester?
4. **Clone a graph** given as an adjacency dict, without sharing any list with the original.
5. **Shortest path with at most k stops**: adapt BFS or Bellman–Ford so that a route may use at most k edges.
6. **Is it a tree?** A graph with n nodes is a tree if it is connected and has exactly $n - 1$ edges. Implement it, and explain why both conditions are needed.
7. **Dijkstra vs BFS**: if every edge weight is 1, does Dijkstra return the same distances as BFS? Test with random graphs.