



UNIVERSITY
OF TRIESTE

Algorithmic Design

University of Trieste – a.a. 2026/2027

Prof. Antonio Emanuele Cinà

`antonio.cina@units.it`

Department of Mathematics, Informatics and Geosciences

University of Trieste

a.a. 2026/2027

Module B – Data Mining through Algorithmic Methods

Chapter 8:

Graphs as computational models

Modelling relationships

Chapter 7 asked which representation makes a data-mining question explicit. Graphs model relationships: roads connect locations; hyperlinks connect web pages; friendships connect people; prerequisites connect courses.

Choose the model before its representation

State what the vertices and edges mean. A graph is the abstract object; an adjacency list or matrix is a way to store it.

Breadth-first search explores distance layers. **Depth-first search** explores a branch and then backtracks.

Both provide foundations for weighted graphs and graph-based clustering later in this module.

Directed and undirected graphs

A graph is a pair $G = (V, E)$: vertices V and edges E . Write $n = |V|$ and $m = |E|$.

Simple directed graph

$E \subseteq \{(u, v) \in V \times V : u \neq v\}$ consists of *ordered* pairs. An edge (u, v) goes from u to v ; it does not imply (v, u) .

Simple undirected graph

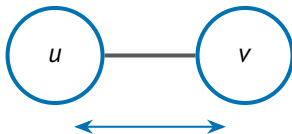
$E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ consists of *unordered* pairs of distinct vertices. Thus $\{u, v\} = \{v, u\}$; loops and parallel edges are excluded.

Both simple models exclude self-loops and parallel edges. A set of pairs cannot represent parallel copies of the same edge.

Multigraphs, labels and weights need an explicitly declared model.

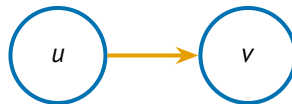
Direction is part of the model

Undirected edge $\{u, v\}$



the relation is symmetric

Directed edge (u, v)



only the direction $u \rightarrow v$ is present

In a road model, a one-way street gives a directed edge. A two-way street may give an undirected edge. (u, v) says nothing about whether (v, u) is present.

Counting edge incidences

The degree $\deg(v)$ counts incident edges in a simple undirected graph.

Handshake lemma

For every finite undirected graph $G = (V, E)$,

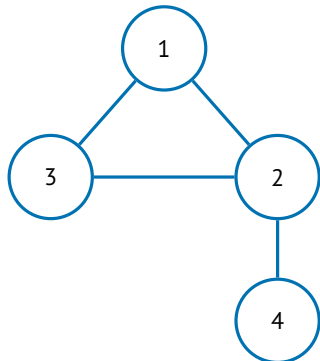
$$\sum_{v \in V} \deg(v) = 2m.$$

Consequently, the number of odd-degree vertices is even.

If loops are admitted, each loop contributes *two* to the degree of its endpoint.

Handshake lemma in a graph

The numbers inside the circles are vertex labels. The graph has $n = 4$ vertices and $m = 4$ edges.

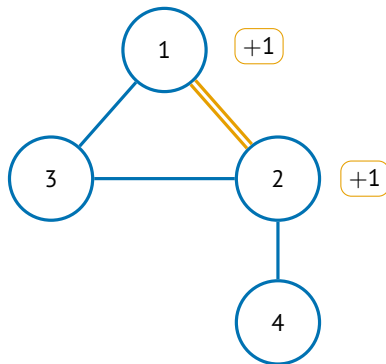


Vertex	Neighbours	Degree
1	2, 3	2
2	1, 3, 4	3
3	1, 2	2
4	2	1

Each degree counts the edges touching that vertex.

$$\sum_{v \in V} \deg(v) = 2 + 3 + 2 + 1 = 8 = 2 \cdot 4 = 2m.$$

Every edge contributes twice



Follow the double-lined edge $\{1, 2\}$

It contributes 1 to $\deg(1)$ and 1 to $\deg(2)$.

It is *one edge*, but it contributes *two* to the degree sum.

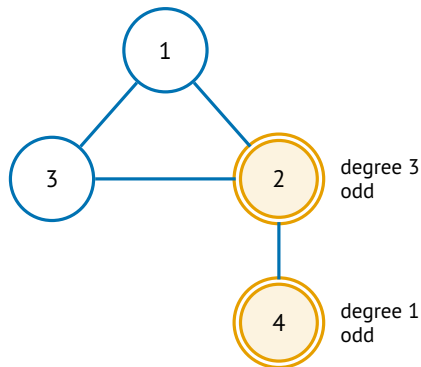
The same reasoning applies to each of the four edges, not just the highlighted one.

General argument

For any finite undirected graph, count edge ends by vertices to obtain $\sum_v \deg(v)$, or by edges to obtain $2m$. Both count the same incidences.

A loop, if admitted, has both ends at the same vertex and contributes 2 there.

Why odd degrees occur in pairs



Double circles mark odd-degree vertices.

In this graph

Even-degree vertices 1, 3: $2 + 2 = 4$.

Odd-degree vertices 2, 4: $3 + 1 = 4$.

There are **two** odd-degree vertices.

In any undirected graph

The handshake lemma makes the total degree even.
Removing the even degrees leaves an even sum:

$$\sum_{\deg(v) \text{ odd}} \deg(v) = 2m - \sum_{\deg(v) \text{ even}} \deg(v).$$

A sum of odd integers is even only when it has an even number of terms.

Degree and graph density

For directed graphs, $\deg^-(v)$ counts incoming edges and $\deg^+(v)$ outgoing edges:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = m.$$

Each edge contributes one head and one tail.

How many edges are present?

A simple undirected graph has at most $\binom{n}{2}$ edges.

- A sparse graph has relatively few edges, often with m of order n .
- A dense graph has a substantial fraction of the possible edges, with m of order n^2 .

Density helps us choose a storage representation.

Selecting vertices and edges

Definition (Subgraph and induced subgraph)

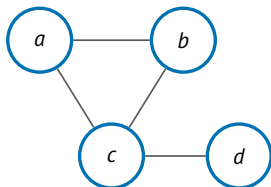
A graph $H = (V_H, E_H)$ is a subgraph of G when $V_H \subseteq V$ and its edges are edges of G with endpoints in V_H .

The subgraph induced by $S \subseteq V$ contains every edge of G whose endpoints both belong to S .

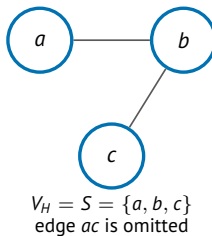
A general subgraph may omit an available edge. Once S is chosen, the induced subgraph $G[S]$ may not omit any internal edge.

Keeping every internal edge

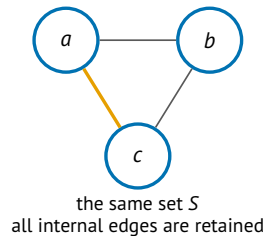
Original graph G



A subgraph H



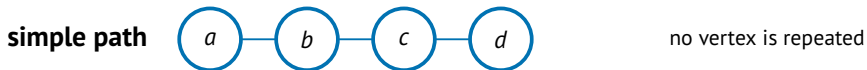
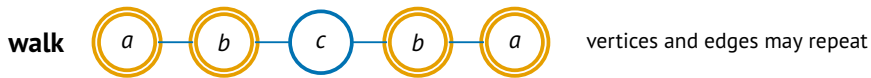
The induced subgraph $G[S]$



Both selections use $S = \{a, b, c\}$. The induced subgraph must keep ac as well as ab and bc .

What may be repeated?

A **walk** follows an edge at every step. A **trail** repeats no edge. A **simple path** repeats no vertex.

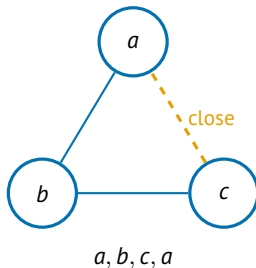


Every simple path is a trail; every trail is a walk. The converses are false.

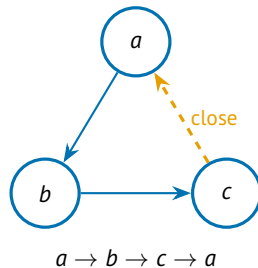
Closing a path into a cycle

A cycle starts and ends at the same vertex, repeats no other vertex, and repeats no edge.

Undirected cycle



Directed cycle



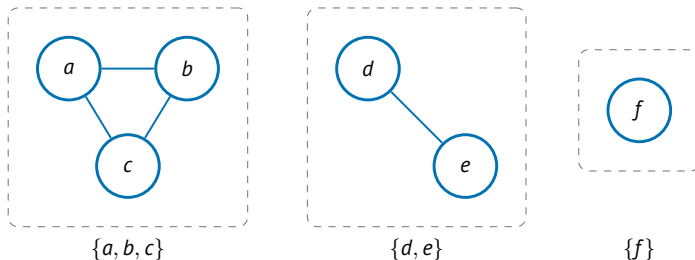
The dashed closing edge completes the path a, b, c . In the directed graph, every arrow must follow the traversal.

In a simple undirected graph, a cycle has at least three edges and reuses no edge: u, v, u is not a cycle. An admitted directed self-loop is a cycle of length one.

Matrix powers count *walks*, whereas many graph problems ask for *paths*.

Connected components

An undirected graph is connected if every pair of vertices is joined by a simple path. A connected component is a maximal vertex set $C \subseteq V$ for which the induced subgraph $G[C]$ is connected.



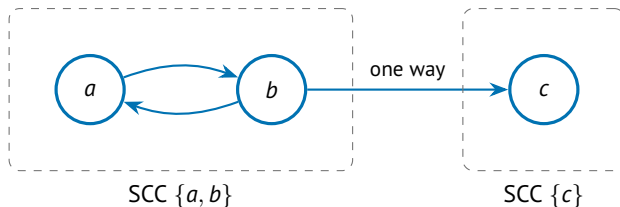
The dashed boundaries show three components. There is no path between different components; an isolated vertex forms a component by itself.

Why maximal?

Maximal means it cannot be enlarged, not largest. The connected set $\{a, b\}$ is not a component: it can include c .

Directed reachability

Reachability follows the arrows. Being able to go from u to v does not imply that we can return.



From a we can reach b and c . From c we cannot reach a or b .

Strongly connected components (SCCs)

An SCC is a maximal set of vertices that can all reach one another. Here the SCCs are $\{a, b\}$ and $\{c\}$. A vertex reaches itself by a path of length zero.

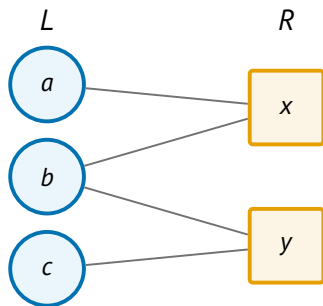
Strongly connected components partition V . Their computation is deferred to later study.

Separating vertices into two parts

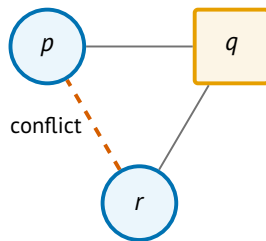
A graph is **bipartite** if $V = L \dot{\cup} R$ and every edge joins one vertex in L to one in R . No edge stays inside one part.

Examples: students and their courses; jobs and machines that can execute them.

Bipartite



Not bipartite



the triangle is an odd cycle

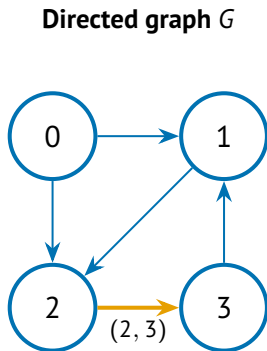
One graph, several representations

An algorithm needs to test edges and enumerate neighbours. A drawing alone does not specify how those operations are implemented.

- An **adjacency matrix** stores a position for every ordered pair.
- **Adjacency lists** store the outgoing neighbours of each vertex.
- An **edge list** stores the edge records directly.

The graph stays the same; the operation costs change.

Testing edges in a matrix



Adjacency matrix

	target v			
	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

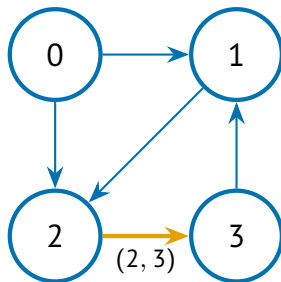
$M[2, 3] = 1$

The thick orange arrow $(2, 3)$ corresponds to the outlined entry $M[2, 3] = 1$.

The matrix uses $\Theta(n^2)$ space and tests an edge in $\Theta(1)$ time.

Enumerating neighbours from adjacency lists

Directed graph G



Adjacency lists

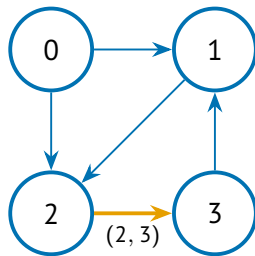
0 :	1	2
1 :	2	
2 :	3	
3 :	1	

$$3 \in \text{Adj}[2]$$

The arrow $(2, 3)$ appears as $3 \in \text{Adj}[2]$. Reading one list enumerates the outgoing neighbours of its vertex. Adjacency lists use $\Theta(n + m)$ space. In an undirected graph, each edge normally appears in two lists.

Storing edges as a list of pairs

Directed graph G



Edge list

(0, 1)
(0, 2)
(1, 2)
(2, 3)
(3, 1)

The arrow $(2, 3)$ appears as one record in the edge list. To enumerate neighbours of 2, scan the records and select those whose source is 2.

The list stores m edge records. Including vertex records gives $\Theta(n + m)$ space.

Space and edge queries

Let $\deg^+(u)$ be the number of outgoing neighbours of u .

Operation	Matrix	Adjacency lists	Edge list
Space	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(n + m)$
Test (u, v)	$\Theta(1)$	$O(\deg^+(u))$	$O(m)$

The list bounds count inspected records. Including constant setup, write $O(1 + \deg^+(u))$ and $O(1 + m)$ for empty lists as well.

A matrix provides direct access; an unsorted list may need a scan.

The cost of scanning a graph

Operation	Matrix	Adjacency lists	Edge list
Enumerate neighbours of u	$\Theta(n)$	$\Theta(\deg^+(u))$	$\Theta(m)$
Scan all edges	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(m)$

These scan bounds count inspected entries; include an additive 1 for an empty scan.

Representation affects traversal time

Scanning every vertex and edge takes $\Theta(n + m)$ with adjacency lists. Scanning every matrix row takes $\Theta(n^2)$, even for a sparse graph.

Breadth-first search

Breadth-first search (BFS) explores an unweighted graph outwards from a source, one distance layer at a time.

Input

A graph $G = (V, E)$ and a source vertex $s \in V$.

Goal

Discover every vertex reachable from s and find the minimum number of edges needed to reach each one.

Output

A distance for each vertex and parent links that reconstruct a shortest path. Unreachable vertices retain distance ∞ .

We first define distance, then the FIFO queue that keeps exploration in layer order.

Shortest paths measured in edges

Definition (Unweighted graph distance)

The distance $\delta(s, v)$ is the minimum number of edges on any path from s to v , and is ∞ when no such path exists.

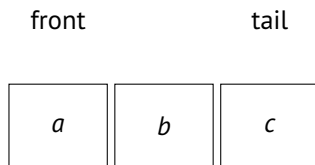
BFS answers both reachability and shortest-path questions in this unweighted sense.

$s \longrightarrow \text{distance 1} \longrightarrow \text{distance 2} \longrightarrow \dots$

What distance means here

Distance counts edges; it does not sum edge weights.

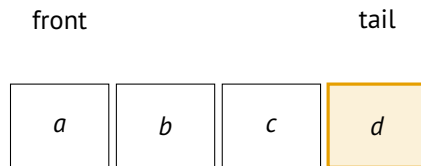
A queue serves the oldest item first



The items arrived in the order a, b, c . The oldest item, a , is at the front.

Enqueue adds at the tail. **Dequeue** removes and returns the front. This is **first in, first out (FIFO)**.

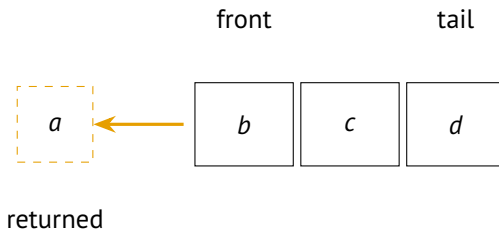
Enqueue adds d at the tail



Apply $\text{Enqueue}(Q, d)$ to $Q = [a, b, c]$.

The queue becomes $[a, b, c, d]$. Item d joins behind every item already waiting; the front is still a .

Deque remove and returns a



Now apply $\text{Dequeue}(Q)$ to $Q = [a, b, c, d]$.

It returns a and leaves $Q = [b, c, d]$. The new front is b ; the relative order of the remaining items is unchanged.

Exploring one distance layer at a time

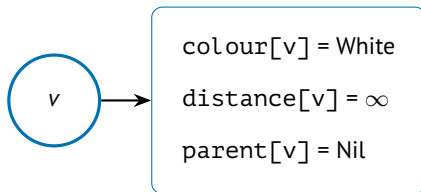
BFS keeps discovered but not yet processed vertices in a FIFO queue.

1. Remove the vertex u at the front.
2. Scan all neighbours of u .
3. For each newly discovered neighbour v , set its distance to $\text{distance}[u] + 1$ and insert it at the tail.

Why a queue?

FIFO processes distance 0, then distance 1, then distance 2, and so on. A later layer cannot overtake an earlier one; this is why first discovery can give the minimum edge count.

What BFS stores for each vertex



Initial values for a vertex
before the source is initialized

$\text{colour}[v]$ records whether v is undiscovered, discovered or fully processed.

$\text{distance}[v]$ is the edge-count label from the source; ∞ means no path has been found yet.

$\text{parent}[v]$ is the vertex that first discovered v . Nil means no parent is assigned.

Fields associated with a vertex

The pseudocode uses arrays indexed by v ; these fields need not be physically stored inside the vertex. $\text{Adj}[v]$ is its neighbour list in the input graph.

Initializing a breadth-first search

```
BFS(G, s)
  for each vertex v in V
    colour[v] <- White
    distance[v] <- infinity
    parent[v] <- Nil
  colour[s] <- Grey
  distance[s] <- 0
  Q <- empty queue
  Enqueue(Q, s)
```

All vertices start undiscovered, with no parent and infinite distance. The source s has distance zero and enters the queue first.

Discovering each vertex once

```

while Q is not empty
  u <- Dequeue(Q)                # remove the front
  for each v in Adj[u]          # scan neighbours
    if colour[v] = White        # first discovery
      colour[v] <- Grey
      distance[v] <- distance[u] + 1
      parent[v] <- u
      Enqueue(Q, v)
  colour[u] <- Black            # full scan complete
return distance, parent

```

Continuation of BFS(G, s). Assign the distance and parent only at first discovery.

Reading one BFS iteration

White Undiscovered: distance ∞ and parent Nil.

Grey Discovered: queued or currently being processed.

Black Complete: every adjacency-list entry has been examined.

Frontier invariant

At each while-loop test, the queue contains exactly the grey vertices. During the scan, the current grey vertex u is outside the queue.

Marking and enqueueing are the same discovery event: a vertex cannot be inserted twice.

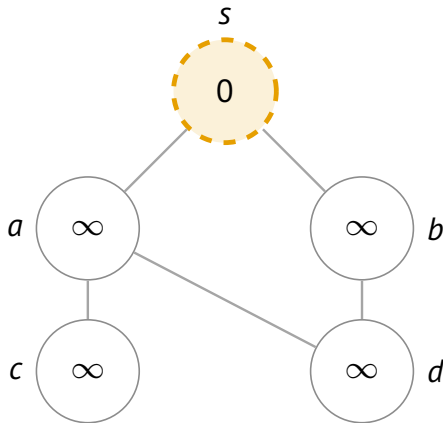
Code states and drawing colours

Grey in the algorithm $\longrightarrow$ **orange dashed circles.**

Black in the algorithm $\longrightarrow$ **blue solid circles.**

White remains white. The drawing colours improve readability; they do not change the algorithmic states.

BFS starts with the source



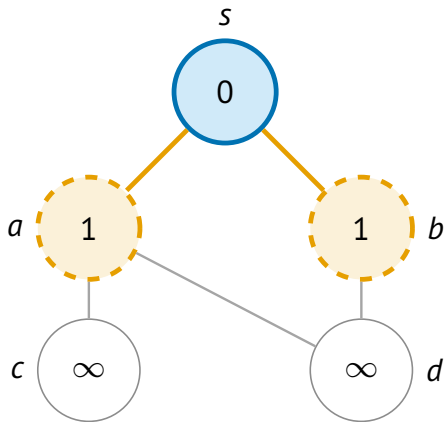
Initial queue
 $Q = [s]$

Write $d[v] = \text{distance}[v]$, the label shown inside each circle.

Initialize $d[s] = 0$ and $\text{parent}[s] = \text{Nil}$. Mark s grey and enqueue it.

All other vertices are white, with distance ∞ and parent Nil .

BFS processes s



Queue before and after the scan

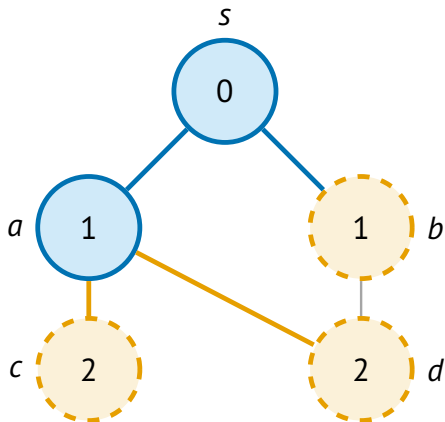
$[s] \longrightarrow [a, b]$

Remove s from the front.

Discover a , then b : assign distance 1 and parent s , mark each grey and enqueue it.

After the complete scan, mark s black. The two orange edges are the new parent edges.

BFS processes a



Queue before and after the scan

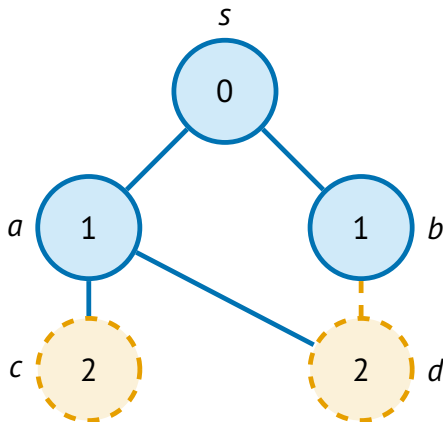
$[a, b] \longrightarrow [b, c, d]$

Remove a ; b is still waiting. The neighbour s is already discovered.

Discover c and d at distance 2, with parent a . Append them behind b .

Mark a black. Orange edges show the two discoveries; blue edges are older parent edges.

BFS processes b



Queue before and after the scan

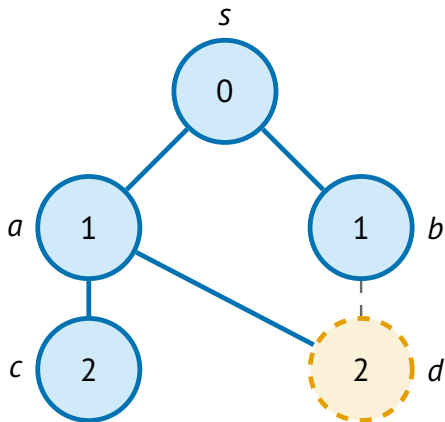
$[b, c, d] \longrightarrow [c, d]$

Remove b and inspect its neighbours. Vertex s is black and d is already grey.

The dashed orange edge highlights the check of d :
no insertion and no parent change.

Mark b black. The queue still contains c, d , in that order.

BFS processes c



Queue before and after the scan

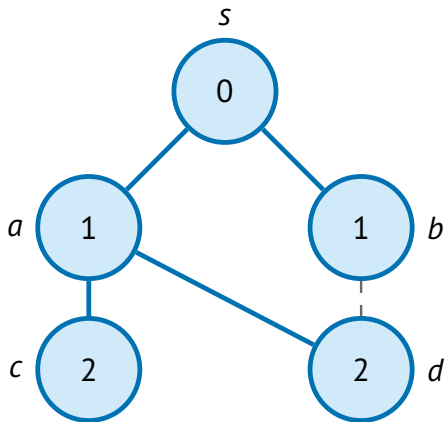
$[c, d] \longrightarrow [d]$

Remove c . Its only neighbour, a , is already black.

No vertex is discovered. Mark c black.

Vertex d is the only remaining queued vertex.

BFS processes d



Queue before and after the scan

$[d] \rightarrow []$

Remove d . Both neighbours, a and b , are already black.

Mark d black. The queue is empty and the search stops.

The parent path s, a, d has two edges, matching $\text{distance}[d] = 2$.

BFS running time

Let R be the vertices reachable from s . Assume adjacency lists and constant-time queue operations.

```

for each vertex  $v$  in  $V$                                 # Theta( $n$ ), once
    initialize colour, distance, parent
Enqueue( $Q$ ,  $s$ )                                         # Theta(1)
while  $Q$  is not empty                                   # one iteration per  $v$  in  $R$ 
     $u \leftarrow$  Dequeue( $Q$ )                             # Theta(1)
    for each  $v$  in Adj[ $u$ ]                               # length(Adj[ $u$ ]) iterations
        if colour[ $v$ ] = White                          # Theta(1) per list entry
            assign fields; Enqueue( $Q$ ,  $v$ )               # Theta(1), at most once per  $v$ 
    
```

Marking at insertion means each reachable vertex enters and leaves the queue once.

Sum the lists actually scanned

Let $G = (V, E)$, with vertex set V , edge set E , $n = |V|$ and $m = |E|$.

Let $R \subseteq V$ be the vertices reachable from the source s . Define m_R as the number of adjacency-list entries scanned from vertices in R :

$$\sum_{u \in R} |\text{Adj}[u]| = m_R.$$

The inner loop scans each of these entries once across the *whole execution*; it does not scan m entries for every vertex.

$$T_{\text{BFS}} = \underbrace{\Theta(n)}_{\text{initialization}} + \underbrace{\Theta(|R|)}_{\text{queue and vertex work}} + \underbrace{\Theta(m_R)}_{\text{adjacency scans}} = \Theta(n + m_R).$$

A directed edge contributes one entry; an undirected edge normally contributes two. When all vertices are reachable, the bound is $\Theta(n + m)$.

BFS space and representation

Auxiliary space

Colour, distance and parent arrays use $\Theta(n)$ space. At most n vertices can be in the queue, so total auxiliary space is $\Theta(n)$.

With an adjacency matrix

Processing one vertex scans a complete row of n entries. For reachable set R , initialization and row scans cost $\Theta(n + n|R|)$, hence $\Theta(n^2)$ when every vertex is processed, even on a sparse graph.

The representation is part of the algorithmic design. Auxiliary space excludes the stored input graph.

Why the BFS labels are shortest distances

Could BFS discover a vertex by a longer route before finding a shorter one?

$$d[v] = \text{distance}[v] \quad \delta(s, v) = \text{minimum number of edges from } s \text{ to } v.$$

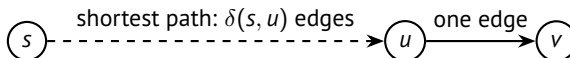
1. **One-edge distance bound:** relate optimal distances across an edge.
2. **Real parent paths:** every finite label is the length of a candidate path, so $\delta(s, v) \leq d[v]$.
3. **Queue-layer invariant:** FIFO processes labels in order and keeps at most two consecutive layers in the queue.

Together, these facts will rule out an incorrect label. We establish the lemmas first, then prove the shortest-path theorem.

Lemma 1: One-edge distance bound

For a directed edge (u, v) , or either orientation of an undirected edge,

$$\delta(s, v) \leq \delta(s, u) + 1.$$



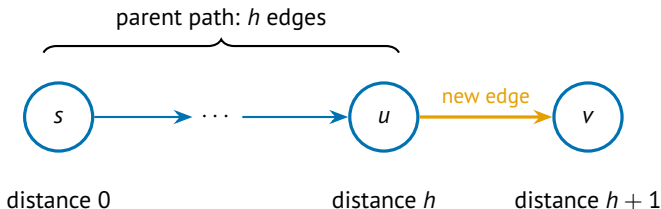
If u is reachable, appending (u, v) gives a walk of length $\delta(s, u) + 1$. Removing any repeated-vertex portion gives a path no longer than this walk.

If u is unreachable, the right-hand side is ∞ , so the inequality is immediate.

Lemma 2: BFS labels describe real paths

Whenever $d[v]$ is finite, its parent links describe a path from s to v with exactly $d[v]$ edges.

Proof. At first discovery from u , BFS sets $\text{parent}[v] = u$ and $\text{distance}[v] = \text{distance}[u] + 1$.

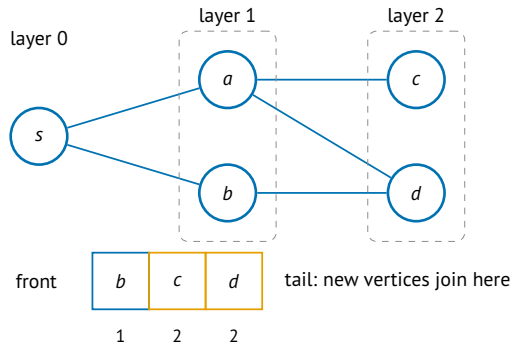


The source has distance 0. Inductively, if the parent path to u has h edges, appending (u, v) gives a path of $h + 1$ edges to v .

$$\delta(s, v) \leq \text{distance}[v].$$

The shortest path cannot be longer than this particular parent path.

Lemma 3: Queue-layer invariant



For a nonempty queue $Q = [q_1, \dots, q_r]$,

$$d[q_1] \leq \dots \leq d[q_r],$$

$$d[q_r] \leq d[q_1] + 1.$$

After processing a , the queue $[b, c, d]$ has labels $[1, 2, 2]$.

The labels are nondecreasing from front to tail. The tail is at most one layer ahead of the front.

We prove these two claims together by induction over BFS iterations.

Why the queue invariant is preserved

Initially $Q = [s]$, so both claims hold. Assume they hold at the next iteration. Remove its front vertex u with label h .

1. The labels stay in order

The remaining labels are h or $h + 1$, in that order. Every newly discovered neighbour receives $h + 1$ and joins at the tail, after the waiting vertices.

2. The queue spans at most two layers

If the new front still has label h , the tail has label at most $h + 1$. If the new front has label $h + 1$, all remaining labels after this scan equal $h + 1$.

Thus both claims hold for the resulting nonempty queue. An empty queue needs no front-to-tail comparison. Induction proves the lemma.

Watching the queue move between layers

In this illustrative queue sequence, subscripts are distance labels:

Before processing a_1 $[a_1, b_1]$

After processing a_1 $[b_1, c_2, d_2]$

After processing b_1 $[c_2, d_2, e_2]$

After processing c_2 $[d_2, e_2, f_3]$

What FIFO prevents

A layer-2 vertex cannot overtake a layer-1 vertex. A layer-3 vertex is created only after processing begins in layer 2.

This is a separate schematic sequence, not another execution of the five-vertex running example. Within one layer, order may depend on the adjacency lists.

BFS finds shortest paths

Theorem (BFS shortest-path property)

In an unweighted graph, for every vertex v reachable from s , BFS assigns

$$\text{distance}[v] = \delta(s, v),$$

where distance counts edges, and its parent links form a shortest path.

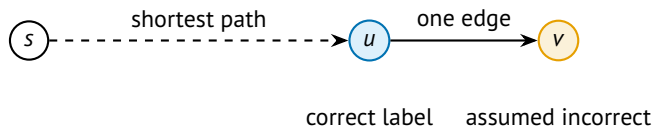
Unreachable vertices retain distance ∞ and parent Nil.

Write $d[v] = \text{distance}[v]$ for the computed label and $\delta(s, v)$ for the optimum. We need *both* inequalities:

$$\delta(s, v) \leq d[v] \quad \text{and} \quad d[v] \leq \delta(s, v).$$

Choose a closest incorrect vertex

Assume a reachable vertex has an incorrect final label. Choose such a v with minimum $\delta(s, v)$. Since $d[s] = 0$, we have $v \neq s$.



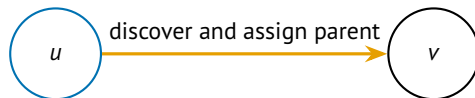
Let u immediately precede v on a shortest path. Then

$$\delta(s, v) = \delta(s, u) + 1.$$

Vertex u is closer to s , so its label is correct: $d[u] = \delta(s, u)$.

Examine the moment when BFS processes u and scans (u, v) .

Case 1: v is still white



$$d[u] = \delta(s, u)$$

white before the scan

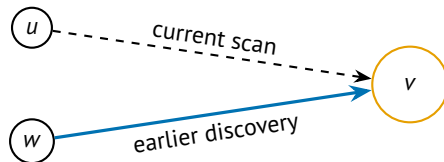
BFS discovers v from u and assigns

$$d[v] = d[u] + 1 = \delta(s, u) + 1 = \delta(s, v).$$

This label is correct and will not change: BFS assigns labels only at discovery.

This contradicts the assumption that v finishes with an incorrect label.

Case 2: v was discovered earlier



Vertex v is grey or black. It was discovered from a vertex w processed no later than u . FIFO order gives $d[w] \leq d[u]$.

$$d[v] = d[w] + 1 \leq d[u] + 1 = \delta(s, v).$$

The parent-path lemma gives $\delta(s, v) \leq d[v]$. Therefore $d[v] = \delta(s, v)$, contradicting the choice of v .

Every reported distance is correct

Both possible states of v contradict the assumption of an incorrect label. Hence

$$d[v] = \delta(s, v) \quad \text{for every reachable } v.$$

Every parent step decreases the label by one. Following parents reaches s after exactly $d[v]$ edges, so the parent chain is a shortest path.

What can change with neighbour order?

When two shortest routes exist, their parent choices may differ. The minimum number of edges, and therefore each BFS distance, stays the same.

Unreachable vertices retain distance ∞ and parent Nil. This is a shortest-path guarantee by edge count, not by total edge weight.

Searching every component

For an undirected graph, restart the search from each still-white vertex.

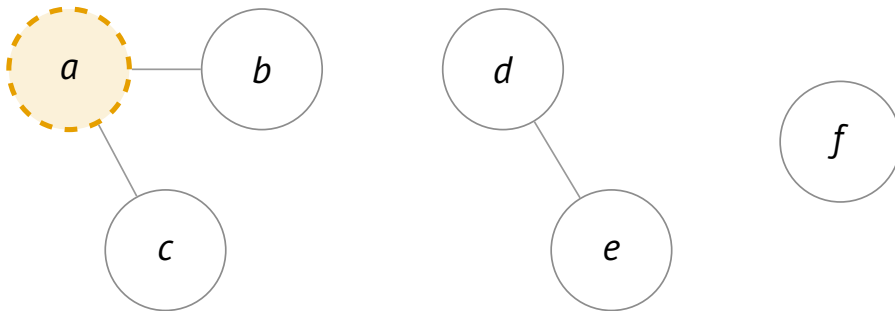
1. Initialize all vertices once.
2. Choose a white vertex as a new root with distance 0.
3. Run the queue loop until that component is complete.
4. Continue until no white vertex remains.

Keep existing colours when starting another root: do not reset earlier components.

Parent edges form one rooted tree per component, collectively a **forest**. Distances are measured from the root of each component.

The full traversal costs $\Theta(n + m)$ with adjacency lists.

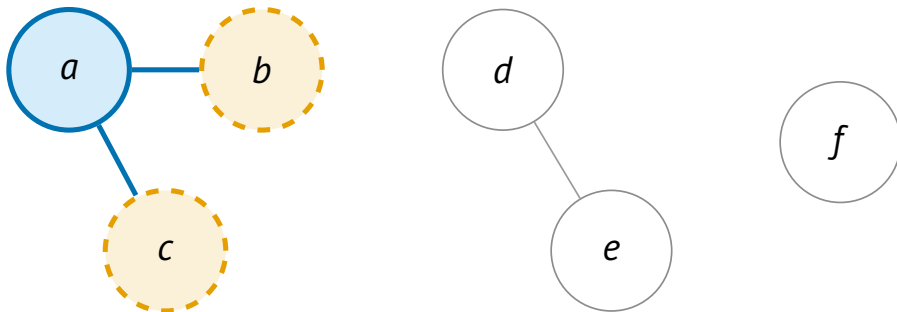
Start the first component



$$Q = [a]$$

This execution uses edges ab , ac , de and isolated f . Scan vertices in order a, b, c, d, e, f and initialize fields once. Start at white vertex a : set its distance to 0, mark it grey and enqueue it.

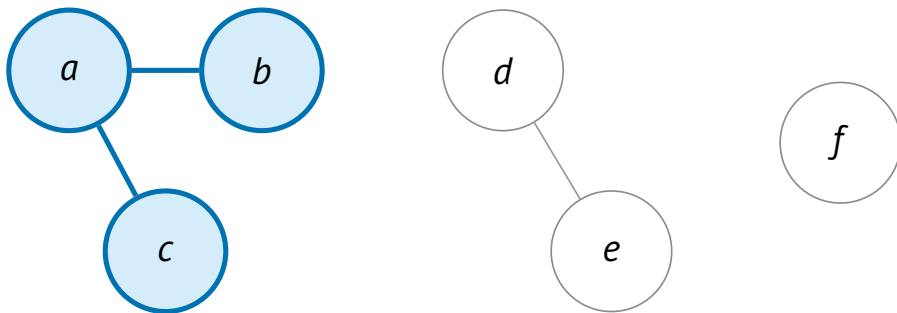
Discover the neighbours of a



$$Q = [b, c]$$

Process a : discover b and c at distance 1, both with parent a . The queue becomes $[b, c]$.

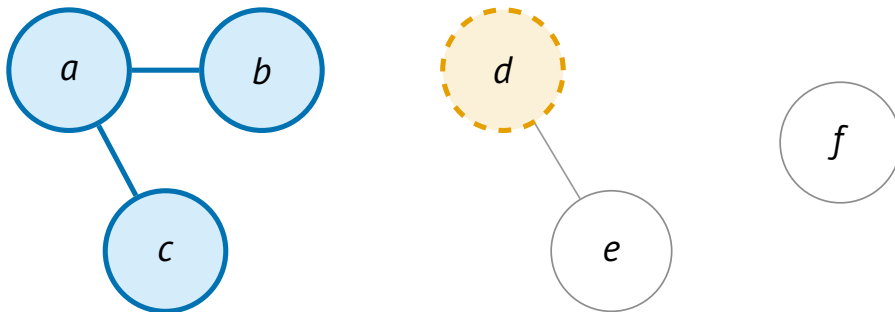
Finish the first component



$$Q = []$$

Process b , then c . Their neighbours are already discovered, so no duplicate is inserted. The first component is complete. Vertices d, e, f remain white.

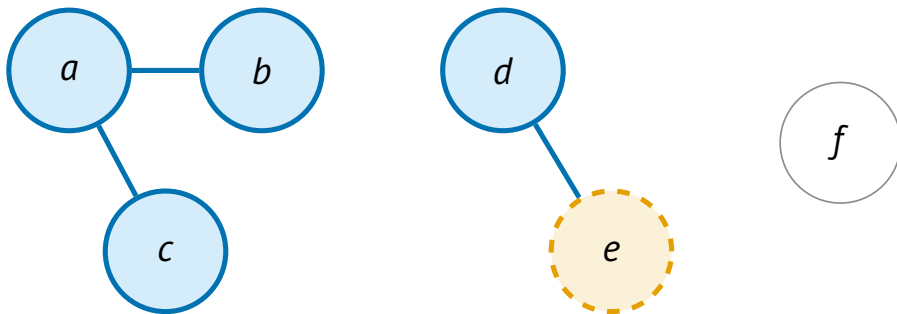
Start a new tree at d



$$Q = [d]$$

Resume the outer scan. Skip b, c , which are black. Vertex d is white: start a new BFS tree with distance 0 and parent Nil. Keep the previous colours and parents.

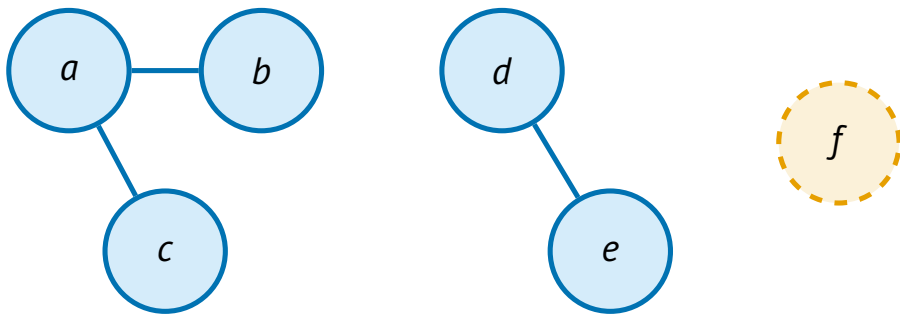
Explore the second component



$$Q = [e]$$

Process d : discover e at distance 1 and set its parent to d . Next, processing e empties this queue and completes the second component.

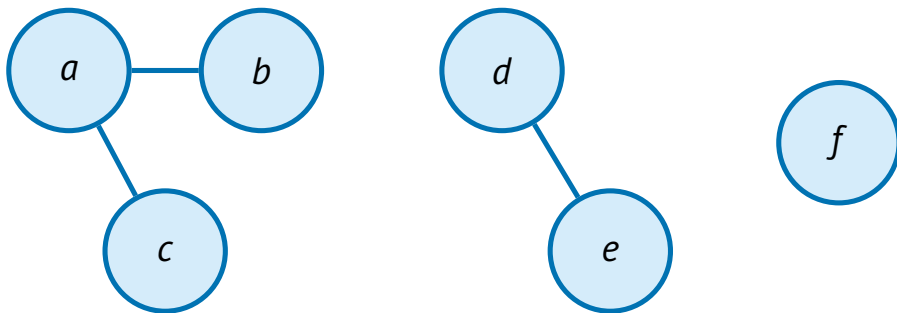
Start the isolated component



$$Q = [f]$$

Skip the completed vertices and start at the isolated white vertex f . Its distance is 0 and its parent is Nil. An isolated vertex is a component of its own.

The traversal produces a forest



$$Q = []$$

Process f : its neighbour list is empty. All vertices are now black. The three roots a, d, f define three BFS trees, including the one-vertex tree at f .

Assigning sides by distance parity

Run BFS in every undirected connected component and assign

$$L = \{v : \text{distance}[v] \text{ is even}\}, \quad R = \{v : \text{distance}[v] \text{ is odd}\}.$$

Each component uses its own BFS root.

- If every edge joins opposite sides, L, R form a bipartition.
- If an edge joins equal sides, it witnesses an odd cycle with the parent paths.

Following a branch and backtracking

DFS follows an unexplored branch as far as possible. It then returns to the most recent unfinished vertex and tries another edge.

The recursive calls act as a stack

The most recently entered unfinished call resumes first: **last in, first out** (LIFO), rather than the FIFO discipline of BFS.

A call for u waits while its child explores a branch. On return, its scan resumes at the next neighbour. Only after all adjacency entries have been examined does u finish.

Active-path invariant: grey vertices are exactly the active calls, forming the root-to-current path. Finishing removes the last call from that path.

An outer loop is needed when one starting vertex cannot reach the entire graph.

Starting a new DFS tree

```
DFS(G)
  for each vertex u in V
    colour[u] <- White
    parent[u] <- Nil
  time <- 0
  for each vertex u in V
    if colour[u] = White
      DFS-visit(G, u)
  return parent, discover, finish
```

Each still-white vertex starts a new tree. The clock time is shared by all calls, including calls in different trees.

Discovery, recursion and finishing

```
DFS-visit(G, u)
  time <- time + 1
  discover[u] <- time
  colour[u] <- Grey
  for each v in Adj[u]
    if colour[v] = White
      parent[v] <- u
      DFS-visit(G, v)
  colour[u] <- Black
  time <- time + 1
  finish[u] <- time
```

The same shared clock records both events. Assign a parent only when discovering a white vertex.

Fix the order of exploration

Use the same undirected graph as the BFS example, with these ordered adjacency lists:

$$\text{Adj}[s] = [a, b], \quad \text{Adj}[a] = [s, c, d],$$

$$\text{Adj}[c] = [a], \quad \text{Adj}[d] = [a, b],$$

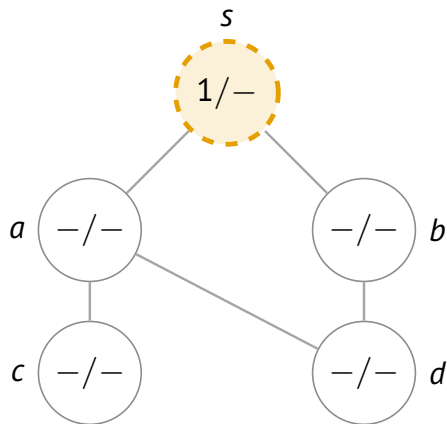
$$\text{Adj}[b] = [s, d].$$

Reading the next drawings

Inside each circle: discovery/finish time. A dash means the event has not occurred. Grey algorithm states are drawn as dashed orange circles (active calls). Black states are drawn as solid blue circles (finished). White circles are undiscovered.

The stack is written from oldest to newest call; the rightmost call is current. Every call on the stack is active. Adjacency order determines the trace, but not the DFS structural properties.

DFS discovers s

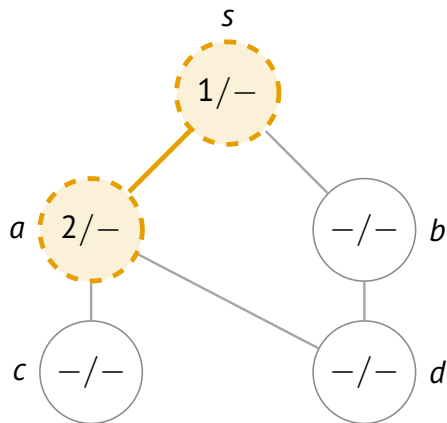


Time 1 $\text{stack} = [s]$

Enter the call for s . Increment the clock to 1, record discovery and mark s grey.

Its first neighbour is a , which is white. The next event will be a recursive call for a .

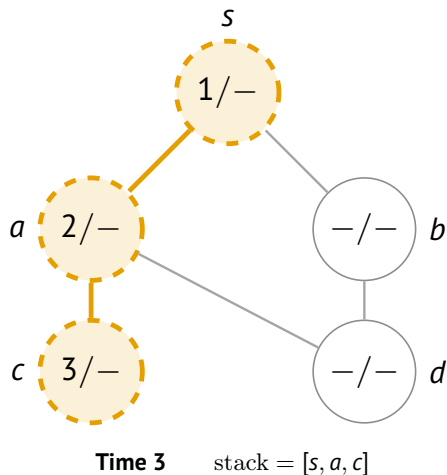
DFS descends to a



Set $\text{parent}[a] = s$ and immediately call $\text{DFS-visit}(G, a)$. Record discovery time 2.

The call for s pauses before examining b . In the list $[s, c, d]$ of a , skip the already-grey s and continue to c .

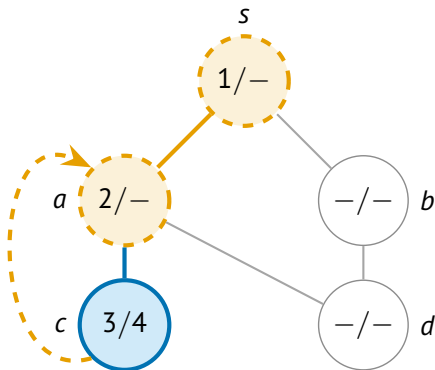
DFS descends to c



Set $\text{parent}[c] = a$ and record discovery time 3.

Both s and a are paused. Vertex c has only neighbour a , which is already grey: no new recursive call is made.

DFS finishes c and returns to a



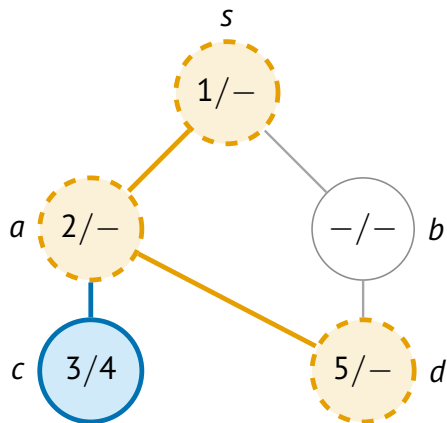
Time 4 $\text{stack} = [s, a]$

The list of c is exhausted. Mark c black and record finishing time 4.

Return to the paused call for a . Resume its list **after** c , so the next neighbour is d .

The dashed arrow shows a return of control, not an extra graph edge.

DFS continues with d

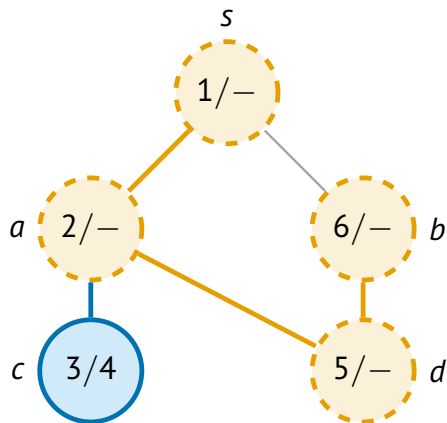


Time 5 $\text{stack} = [s, a, d]$

Back in the call for a , discover the white neighbour d : set parent a and discovery time 5.

The list of d is $[a, b]$. Skip the already-grey a ; the next neighbour b is still white.

DFS descends to b



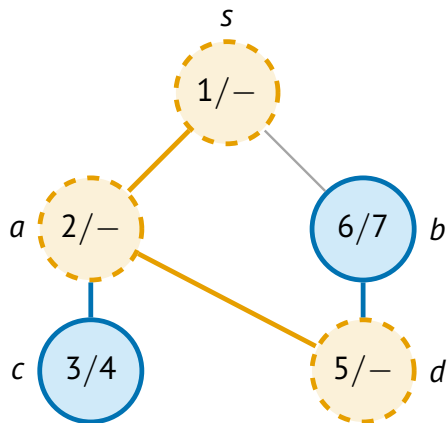
Time 6 $\text{stack} = [s, a, d, b]$

Set $\text{parent}[b] = d$ and record discovery time 6.

Both neighbours s, d are already grey. No recursive call is made.

Although s is adjacent to b , its scan is still paused. The parent of b is d .

DFS finishes b

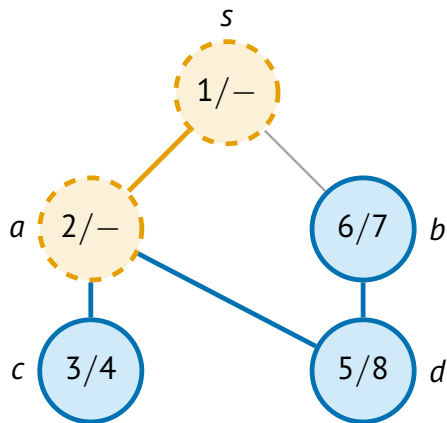


Time 7 $\text{stack} = [s, a, d]$

The list $[s, d]$ is exhausted. Mark b black and record finishing time 7.

Return to d , whose scan resumes after b . There are no remaining neighbours.

DFS finishes d

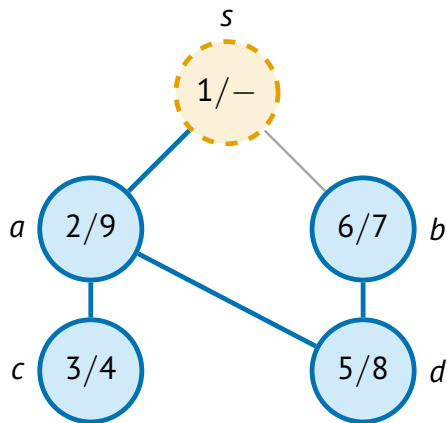


Time 8 $\text{stack} = [s, a]$

Mark d black and record finishing time 8.

Return to a . Its list $[s, c, d]$ resumes after d and is now exhausted.

DFS finishes a

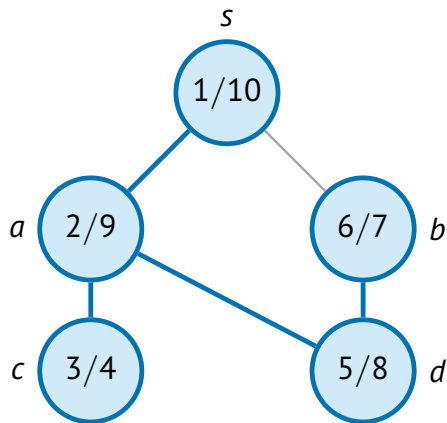


Time 9 $\text{stack} = [s]$

Mark a black and record finishing time 9.

Return to s and resume $[a, b]$ after a . The remaining neighbour b is already black, so skip it.

DFS finishes s



Time 10 `stack = []`

The list of s is exhausted. Mark s black and record finishing time 10.

The stack is empty. All five vertices were reached, so the outer loop starts no further tree.

The blue edges form the DFS tree; $s-b$ was not selected.

DFS running time

```

for each vertex u in V                # Theta(n), once
    initialize colour and parent
for each vertex u in V                # Theta(n), once
    if colour[u] = White
        DFS-visit(G, u)

DFS-visit(G, u)                       # called once per vertex
    record discovery; colour[u] <- Grey # Theta(1)
    for each v in Adj[u]               # length(Adj[u]) iterations
        if colour[v] = White          # Theta(1) per list entry
            parent[v] <- u
            DFS-visit(G, v)
    colour[u] <- Black; record finish  # Theta(1)
    
```

Marking grey before scanning neighbours prevents a second call for the same vertex.

Every adjacency list is scanned once

Each vertex triggers one call, with constant work outside its adjacency loop. Across all calls,

$$\sum_{u \in V} |\text{Adj}[u]| = \begin{cases} m & \text{directed graph,} \\ 2m & \text{undirected graph.} \end{cases}$$

$$T_{\text{DFS}}(n, m) = \Theta(n) + \Theta(m) = \Theta(n + m).$$

The arrays use $\Theta(n)$ space. The stack holds the current root-to-vertex path: its size is the current depth, at most $O(n)$. Total auxiliary space is $\Theta(n)$.

With a matrix, every visit scans n entries. The full traversal takes $\Theta(n^2)$ time.

Nested and disjoint activity intervals

The interval

$$[\text{discover}[u], \text{finish}[u]]$$

is the time during which the call for u is active.

Reading the timestamps

For any two vertices u and v , their DFS activity intervals are disjoint or one contains the other. The interval of u strictly contains that of v exactly when v is a proper descendant of u in the DFS forest.

When v is a proper descendant of u , the timestamps satisfy

$$\text{discover}[u] < \text{discover}[v] < \text{finish}[v] < \text{finish}[u].$$

Classifying a directed edge

When DFS scans (u, v) , inspect the state of v .

White: tree edge. It discovers a new vertex and sets its parent to u .

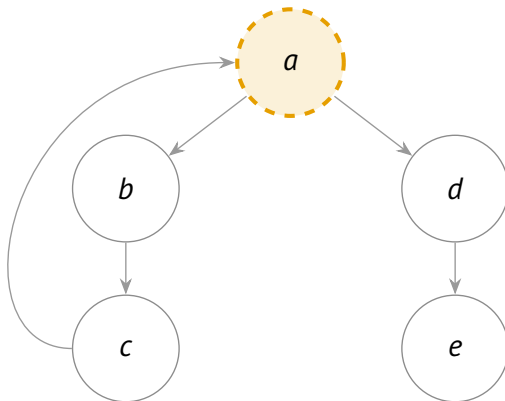
Grey: back edge. It reaches an active ancestor (or u itself for a self-loop).

Black: forward or cross edge. It reaches a completed vertex.

A forward edge reaches a proper descendant discovered through other tree edges. A cross edge joins disjoint activity intervals, possibly in different DFS trees.

Use the timestamps to distinguish these last two cases. An edge to a black vertex is not a back edge.

DFS starts at a

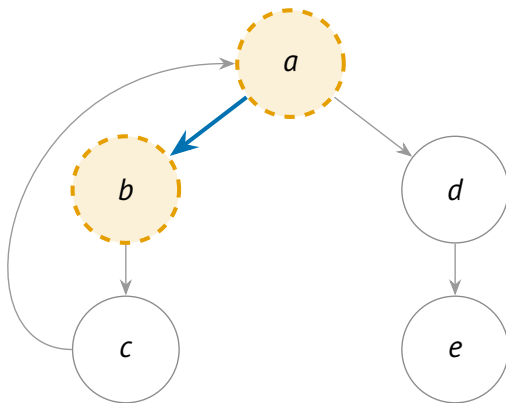


$\text{stack} = [a]$

Use the directed graph shown. Read
 $\text{Adj}[a] = [b, d]$, $\text{Adj}[b] = [c]$, $\text{Adj}[c] = [a]$,
 $\text{Adj}[d] = [e]$.

Discover a . Orange dashed circles represent
 grey vertices: active calls. The remaining
 vertices are white.

DFS discovers b from a

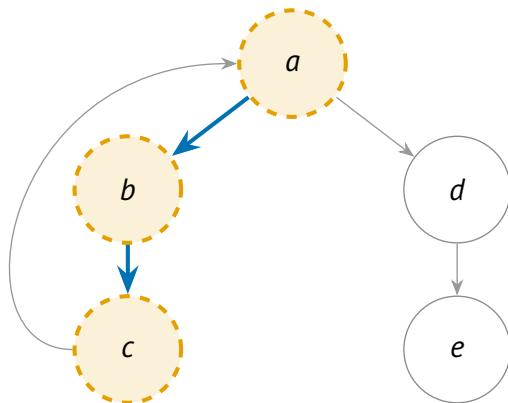


$\text{stack} = [a, b]$

Vertex b is white. Set $\text{parent}[b] = a$ and enter its call.

The call for a pauses before exploring d . The blue arrow $a \rightarrow b$ is a tree edge.

DFS discovers c from b

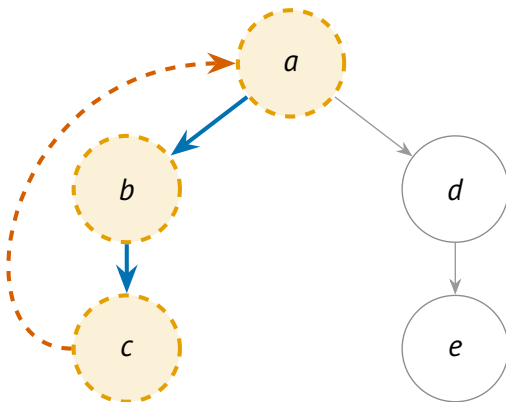


stack = $[a, b, c]$

Vertex c is white. Set $\text{parent}[c] = b$ and enter its call.

The active tree path is $a \rightarrow b \rightarrow c$. Now c scans its outgoing edge to a .

A back edge reveals the cycle



stack = $[a, b, c]$

Vertex a is grey: its call is still on the stack.
Thus $c \rightarrow a$ is a back edge.

The tree path and back edge give

$$a \rightarrow b \rightarrow c \rightarrow a.$$

No recursive call to a is made. This directed cycle is already witnessed; the unvisited branch through d, e is irrelevant to its existence.

Detecting directed cycles

Proposition (Back edges and directed cycles)

A directed graph contains a cycle exactly when DFS finds an edge to a grey ancestor.

Here the current vertex counts as an ancestor of itself if self-loops are admitted.

One direction. A back edge (u, v) closes a cycle with the DFS-tree path from v to u . If $u = v$, the edge itself is a cycle.

The test refers to *directed* graphs. In an undirected graph, the reverse occurrence of the parent edge needs separate treatment.

Why a cycle produces a back edge

Converse proof. Suppose a directed cycle exists. Let v be its first vertex discovered by DFS.

1. Every other cycle vertex is still white when v is discovered.
2. Before v can finish, DFS explores all still-white vertices reachable along the cycle, so they become descendants of v .
3. In particular, the predecessor of v on the cycle scans its edge to v while v is still active.

That edge reaches a grey ancestor and is a back edge. A self-loop gives the same conclusion immediately.

Guided design problem: distribution stages

A company sends products from central warehouse A , through warehouses B, C , to points of sale D, E . Routes can be used only in their declared direction.

$$A \rightarrow B, \quad A \rightarrow C, \quad C \rightarrow B, \quad B \rightarrow D, \quad C \rightarrow D.$$

The request

For every location, return the minimum number of transport stages from A . Use ∞ when no route exists.

All routes count as one stage. We are not measuring monetary cost or travel time.

Discuss: How do we formalise the input, output and objective? What should we return for A and an unreachable location?

A contract for minimum stages

Input

A finite set of locations, directed routes and source A . Write n for locations and m for routes.

Output

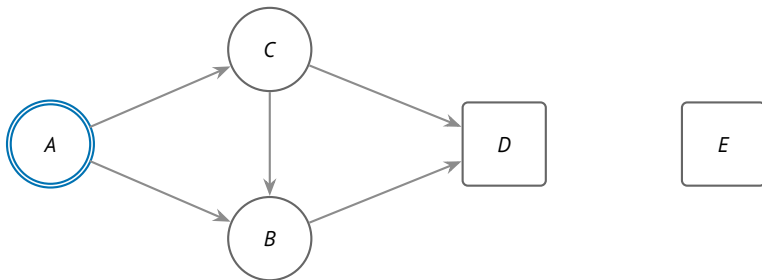
One pair $(v, \delta(A, v))$ for every location, including A and unreachable locations.

Objective

Minimise the number of edges in an A -to- v path. The empty path gives $\delta(A, A) = 0$; no path gives ∞ .

Discuss: How would we draw this network? What do vertices and arrows represent, and where does E belong?

One location per vertex, one route per arrow



Circles: warehouses. Rounded rectangles: points of sale. The double circle is the source A .

Keep E in the vertex set even though it has no incident route. Here $n = 5$ and $m = 5$.

What does an empty row tell us?

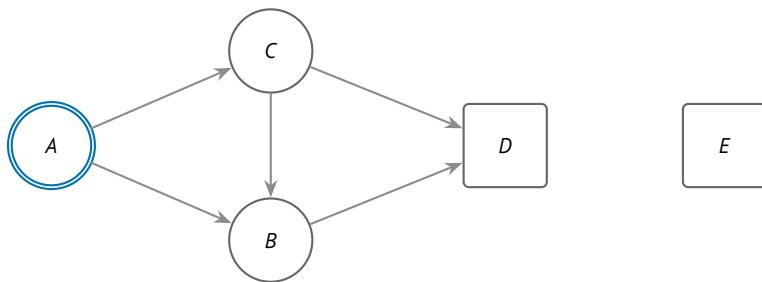
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
<i>A</i>	0	1	1	0	0
<i>B</i>	0	0	0	1	0
<i>C</i>	0	1	0	1	0
<i>D</i>	0	0	0	0	0
<i>E</i>	0	0	0	0	0

Discuss before continuing

Rows are origins; columns are destinations. A 1 denotes a direct route.

Rows *D* and *E* contain only zeros. Are both locations unreachable from *A*?

Outgoing routes and reachability are different



Row D is empty, but $A \rightarrow B \rightarrow D$ reaches D . Row E is empty and no path from A reaches E .

Read the matrix precisely

A zero row means no outgoing route. Reachability depends on complete paths from the source.

Discuss: Which algorithm finds the fewest stages? What data structure preserves layer order, and what should we store?

Use BFS and retain distance and parent

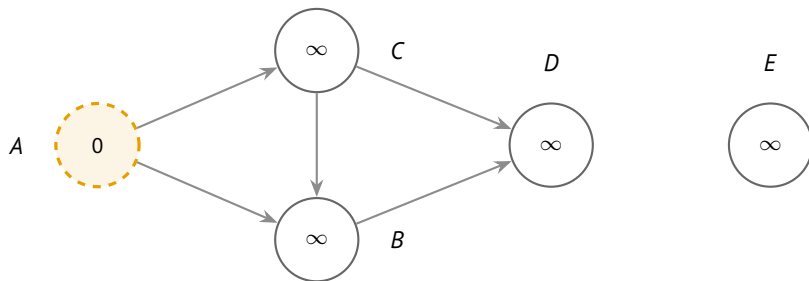
This is a directed, unweighted, single-source problem. Each route adds exactly one stage.

Why BFS?

A FIFO queue processes layers in order. The first discovery of a vertex already uses the fewest possible edges.

Retain distance for the requested values and parent to reconstruct routes. In this version, distance ∞ also marks a vertex not yet discovered.

Start with the central warehouse



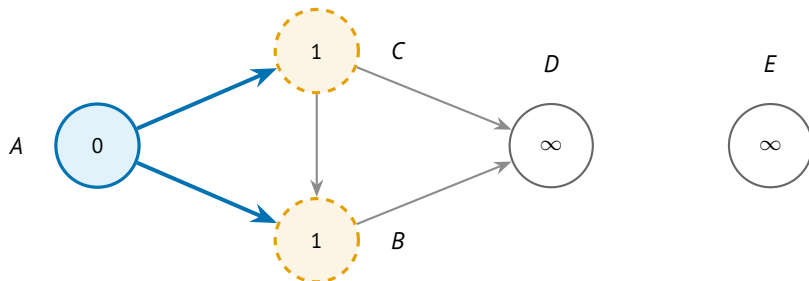
Initially $Q = [A]$. Write distance inside each vertex; all parents are Nil.

Orange dashed: queued. Blue: processed. White: undiscovered. Thick blue arrows: parent links.

Scan matrix columns in order A, B, C, D, E .

Discuss: After removing A and scanning its row, what are the new queue, distances and parents?

Process A and discover the first layer

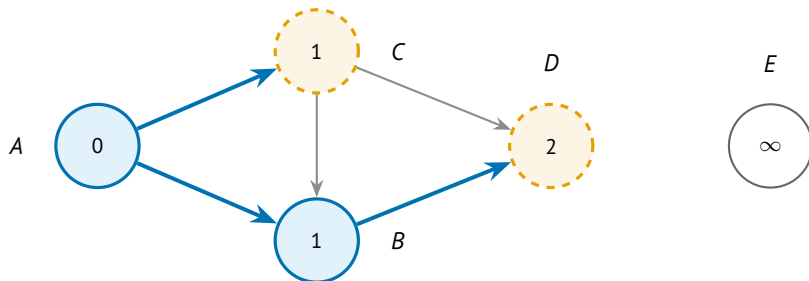


Queue: $[A] \longrightarrow [B, C]$. $d[B] = d[C] = 1$.

Both parents are A. The distance is assigned before enqueueing; later scans cannot insert the same vertex again.

Predict: Which vertex leaves the queue next? What distance will its newly discovered neighbour receive?

Process B and discover D

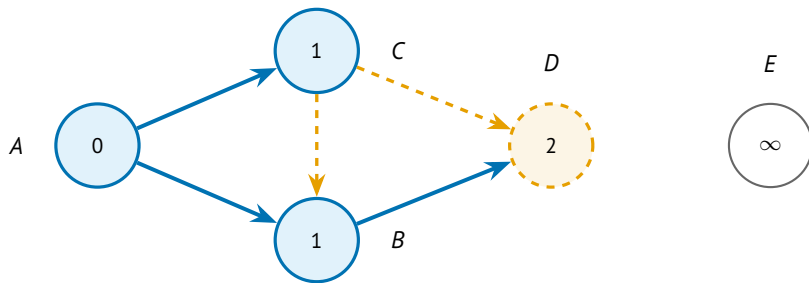


Queue: $[B, C] \longrightarrow [C, D]$. $d[D] = d[B] + 1 = 2$.

Set $\text{parent}[D] = B$. Vertex C was already waiting and stays ahead of D .

Discuss: Next, C reaches B and D . Should either be enqueued again, or should the parent of D change?

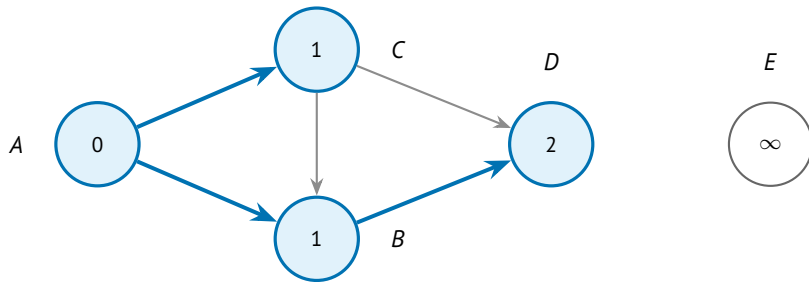
Process C without inserting duplicates



Queue: $[C, D] \rightarrow [D]$. Both B and D already have finite distances.

$A \rightarrow C \rightarrow D$ is another two-stage route. Keep $d[D] = 2$ and parent B . No minimum comparison or distance decrease is needed.

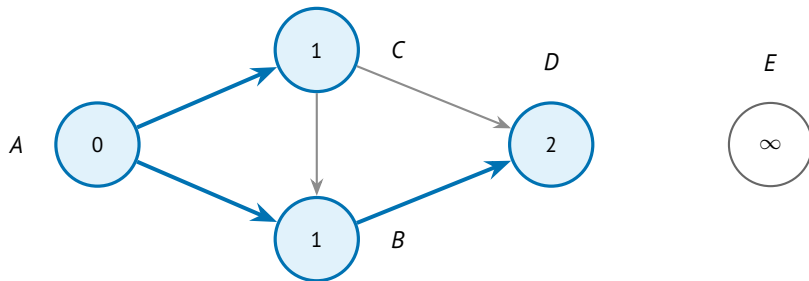
Process D and stop when the queue is empty



Queue: $[D] \rightarrow []$. Row D has no outgoing edge.

E never entered the queue. It remains in the output with distance ∞ .

Read the layers and return every location



$$L_0 = \{A\}, \quad L_1 = \{B, C\}, \quad L_2 = \{D\}.$$

$$[(A, 0), (B, 1), (C, 1), (D, 2), (E, \infty)]$$

Discuss: How would we code the row scan? Which test detects a first discovery, and when should its distance change?

Initialise the matrix-based BFS

```
Minimum-stages(M, source)
  n <- number of rows of M
  for each vertex v
    distance[v] <- infinity
    parent[v] <- Nil
  distance[source] <- 0
  Q <- empty FIFO queue
  Enqueue(Q, source)
```

Locations A, B, C, D, E correspond to matrix indices 0, 1, 2, 3, 4.

Scan the outgoing row of the next vertex

```
while Q is not empty
  u <- Dequeue(Q)
  for v <- 0 to n-1
    if M[u,v] = 1 and distance[v] = infinity
      distance[v] <- distance[u] + 1
      parent[v] <- u
      Enqueue(Q, v)
return distance, parent
```

Continuation of Minimum-stages. Pair each location with its returned distance to produce the requested list.

Discuss: How do the parents reconstruct a route to D ? What if C is processed before B ?

Reverse the parent chain

$$D \leftarrow B \leftarrow A \implies A \rightarrow B \rightarrow D.$$

The other reachable routes are $[A]$, $A \rightarrow B$ and $A \rightarrow C$. No route exists to E .

A different scan order may choose a different parent

Processing C before B can give $\text{parent}[D] = C$, hence $A \rightarrow C \rightarrow D$. Both routes have two edges; the distance stays 2.

Discuss: How many queue removals and matrix checks are needed? Compare adjacency lists and separate auxiliary space from input storage.

Count the rows and edges actually scanned

Let R be the locations reachable from A ; let m_R count their outgoing edges.

Representation	Traversal time	Worst case
Matrix	$\Theta(n + n R)$	$\Theta(n^2)$
Adjacency lists	$\Theta(n + m_R)$	$\Theta(n + m)$

For this input

Four dequeued vertices scan $4 \cdot 5 = 20$ matrix entries. Adjacency lists scan just five edges. Both initialise all five locations.

Auxiliary space is $\Theta(n)$ for arrays and the queue. Input storage is $\Theta(n^2)$ for the matrix or $\Theta(n + m)$ for lists.

From fewest edges to minimum cost

BFS answers reachability and shortest-path questions when distance means the number of edges. DFS exposes nested exploration and directed cycles.

The next modelling step

Keep the vertices and edges, but attach numerical weights to the edges. A route with fewer edges need not have lower total cost.

Chapter 9 develops relaxation, priority queues and algorithms for minimum-cost paths.

Thank you!



UNIVERSITY
OF TRIESTE