Design and Analysis of Algorithms

Graphs: Traversal, Components, Topological Order

A network of six cities.

Visit every city exactly once — **wave by wave**, or **dive deep**?

Dr. Golzari

University of Hormozgan

Fall 1405

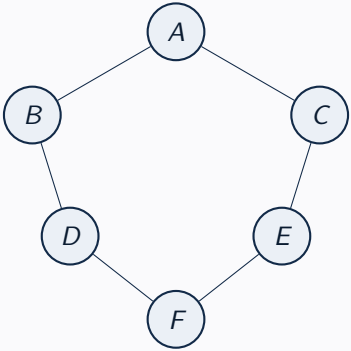
Where We're Going

1. Graphs and adjacency lists
2. BFS: explore wave by wave
3. DFS: dive and backtrack
4. Connected components: count the islands
5. Topological order: tasks in the right sequence
6. Bonus: catching cycles

Graphs and Adjacency Lists

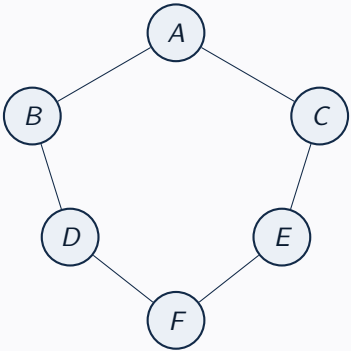
What's a Graph?

Vertices (cities) joined by **edges** (roads).



What's a Graph?

Vertices (cities) joined by **edges** (roads).



Edges today: *A*–*B*, *A*–*C*, *B*–*D*, *C*–*E*, *D*–*F*, *E*–*F*.

- Undirected (roads go both ways). Later: **directed** = one-way arrows.

How Do We Store It? The Adjacency List

For each vertex: the list of its neighbors, **sorted alphabetically** (so our walks are reproducible):

A:	B, C
B:	A, D
C:	A, E
D:	B, F
E:	C, F
F:	D, E

How Do We Store It? The Adjacency List

For each vertex: the list of its neighbors, **sorted alphabetically** (so our walks are reproducible):

A:	B, C
B:	A, D
C:	A, E
D:	B, F
E:	C, F
F:	D, E

- Space $\Theta(V + E)$ — one entry per vertex, two per edge.
- (An adjacency matrix costs $\Theta(V^2)$ — fine for tiny graphs, wasteful for real networks.)

The Task: Visit Every Vertex Exactly Once

- Starting from A, walk the edges and **mark** each vertex the first time you reach it.

The Task: Visit Every Vertex Exactly Once

- Starting from A, walk the edges and **mark** each vertex the first time you reach it.
- Two classic strategies:
  - **BFS** — sweep in rings, nearest first (a queue).
  - **DFS** — follow one road to the end, then backtrack (recursion).

The Task: Visit Every Vertex Exactly Once

- Starting from A, walk the edges and **mark** each vertex the first time you reach it.
- Two classic strategies:
  - **BFS** — sweep in rings, nearest first (a queue).
  - **DFS** — follow one road to the end, then backtrack (recursion).

Both cost

$\Theta(V + E)$ — every vertex once, every edge once or twice.

BFS: The Wave

BFS: Ripples from the Start

Explore in **rings**: first everything at distance 1, then distance 2, ...

BFS: Ripples from the Start

Explore in **rings**: first everything at distance 1, then distance 2, ...

Level 0	A
Level 1	B, C
Level 2	D, E
Level 3	F

BFS: Ripples from the Start

Explore in **rings**: first everything at distance 1, then distance 2, ...

Level 0	A
Level 1	B, C
Level 2	D, E
Level 3	F

- Tool: a **queue** — first discovered, first expanded.

BFS: The Algorithm

1. Put the start s in a queue; mark s.

BFS: The Algorithm

1. Put the start s in a queue; mark s.
2. Pop the front vertex v.

BFS: The Algorithm

1. Put the start s in a queue; mark s.
2. Pop the front vertex v.
3. For each neighbor u of v: if u is unvisited, **mark** it, record $dist[u] = dist[v] + 1$, and **push** it.

BFS: The Algorithm

1. Put the start s in a queue; mark s.
2. Pop the front vertex v.
3. For each neighbor u of v: if u is unvisited, **mark** it, record $dist[u] = dist[v] + 1$, and **push** it.
4. Repeat until the queue is empty.

BFS: The Algorithm

1. Put the start s in a queue; mark s.
2. Pop the front vertex v.
3. For each neighbor u of v: if u is unvisited, **mark** it, record $dist[u] = dist[v] + 1$, and **push** it.
4. Repeat until the queue is empty.

- The queue guarantees: all distance-k vertices are handled before any distance-$(k{+}1)$ vertex.

Walkthrough: BFS from A

- pop $A \rightarrow$ visit $B,\ C$ queue: $[B,\ C]$

Walkthrough: BFS from A

- pop $A \rightarrow$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \rightarrow$ visit D queue: $[C,\ D]$

Walkthrough: BFS from A

- pop $A \rightarrow$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \rightarrow$ visit D queue: $[C,\ D]$
- pop $C \rightarrow$ visit E queue: $[D,\ E]$

Walkthrough: BFS from A

- pop $A \rightarrow$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \rightarrow$ visit D queue: $[C,\ D]$
- pop $C \rightarrow$ visit E queue: $[D,\ E]$
- pop $D \rightarrow$ visit F queue: $[E,\ F]$

Walkthrough: BFS from A

- pop $A \to$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \to$ visit D queue: $[C,\ D]$
- pop $C \to$ visit E queue: $[D,\ E]$
- pop $D \to$ visit F queue: $[E,\ F]$
- pop $E \to$ (F already marked) queue: $[F]$

Walkthrough: BFS from A

- pop $A \to$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \to$ visit D queue: $[C,\ D]$
- pop $C \to$ visit E queue: $[D,\ E]$
- pop $D \to$ visit F queue: $[E,\ F]$
- pop $E \to$ (F already marked) queue: $[F]$
- pop $F \to$ queue empty. Done.

Walkthrough: BFS from A

- pop $A \to$ visit $B,\ C$ queue: $[B,\ C]$
- pop $B \to$ visit D queue: $[C,\ D]$
- pop $C \to$ visit E queue: $[D,\ E]$
- pop $D \to$ visit F queue: $[E,\ F]$
- pop $E \to$ (F already marked) queue: $[F]$
- pop $F \to$ queue empty. Done.

Visit order

$A,\ B,\ C,\ D,\ E,\ F$

BFS Gives Shortest Paths for Free

vertex	A	B	C	D	E	F
dist	0	1	1	2	2	3

BFS Gives Shortest Paths for Free

vertex	A	B	C	D	E	F
dist	0	1	1	2	2	3

- *dist* = fewest **edges** from A — e.g. $A \to C \to E \to F$: 3 edges.

BFS Gives Shortest Paths for Free

vertex	A	B	C	D	E	F
dist	0	1	1	2	2	3

- *dist* = fewest **edges** from A — e.g. $A \to C \to E \to F$: 3 edges.
- Guaranteed: we discover vertices in distance order.

BFS Gives Shortest Paths for Free

vertex	A	B	C	D	E	F
dist	0	1	1	2	2	3

- *dist* = fewest **edges** from A — e.g. $A \to C \to E \to F$: 3 edges.
- Guaranteed: we discover vertices in distance order.
- Total cost: $\Theta(V + E)$.

DFS: The Dive

DFS: Follow One Road to the End

- From the current vertex, walk to an unvisited neighbor **immediately**.

DFS: Follow One Road to the End

- From the current vertex, walk to an unvisited neighbor **immediately**.
- Dead end? **Backtrack** to the last junction and try the next road.

DFS: Follow One Road to the End

- From the current vertex, walk to an unvisited neighbor **immediately**.
- Dead end? **Backtrack** to the last junction and try the next road.
- Tool: **recursion** (the call stack remembers the way back).

DFS: Follow One Road to the End

- From the current vertex, walk to an unvisited neighbor **immediately**.
- Dead end? **Backtrack** to the last junction and try the next road.
- Tool: **recursion** (the call stack remembers the way back).

Same graph as BFS — different walk!

Watch the order change.

DFS: The Algorithm

1. Mark v as visited.

DFS: The Algorithm

1. Mark v as visited.
2. For each neighbor u of v (alphabetical order):

DFS: The Algorithm

1. Mark v as visited.
2. For each neighbor u of v (alphabetical order):
3. if u is unvisited, **recurse**: **DFS**(u).

DFS: The Algorithm

1. Mark v as visited.
2. For each neighbor u of v (alphabetical order):
3. if u is unvisited, **recurse**: **DFS**(u).
4. Return when all neighbors are done — the stack unwinds by itself.

DFS: The Algorithm

1. Mark v as visited.
2. For each neighbor u of v (alphabetical order):
3. if u is unvisited, **recurse**: **DFS**(u).
4. Return when all neighbors are done — the stack unwinds by itself.

- Recursion = an implicit **stack**: the current path sits on top.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.
- B: A visited $\to$ dive to D.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.
- B: A visited $\to$ dive to D.
- D: B visited $\to$ dive to F.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.
- B: A visited $\to$ dive to D.
- D: B visited $\to$ dive to F.
- F: D visited $\to$ dive to E.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.
- B: A visited $\to$ dive to D.
- D: B visited $\to$ dive to F.
- F: D visited $\to$ dive to E.
- E: neighbor C unvisited $\to$ dive to C.

Walkthrough: DFS from A

Dive (alphabetical neighbors), then backtrack:

$$A \to B \to D \to F \to E \to C$$

- A: first neighbor B unvisited $\to$ dive.
- B: A visited $\to$ dive to D.
- D: B visited $\to$ dive to F.
- F: D visited $\to$ dive to E.
- E: neighbor C unvisited $\to$ dive to C.
- C: both neighbors visited $\to$ dead end — backtrack all the way.

Visit order

$A,\ B,\ D,\ F,\ E,\ C$ — BFS said A, B, C, D, E, F!

BFS vs DFS — Same Graph, Two Walks

	BFS	**DFS**
tool	queue (FIFO)	stack / recursion
walk on our graph	A, B, C, D, E, F	A, B, D, F, E, C
shape	rings, level by level	one path at a time
cost	$\Theta(V + E)$	$\Theta(V + E)$
famous for	shortest unweighted paths	cycles, topological order

BFS vs DFS — Same Graph, Two Walks

	BFS	**DFS**
tool	queue (FIFO)	stack / recursion
walk on our graph	A, B, C, D, E, F	A, B, D, F, E, C
shape	rings, level by level	one path at a time
cost	$\Theta(V + E)$	$\Theta(V + E)$
famous for	shortest unweighted paths	cycles, topological order

- Same rules of the game, **different data structure** $\Rightarrow$ different walk.

Connected Components

The Problem: Count the Islands

Some graphs are **disconnected** — separate pieces ("islands", "friend groups").

The Problem: Count the Islands

Some graphs are **disconnected** — separate pieces ("islands", "friend groups").



The Problem: Count the Islands

Some graphs are **disconnected** — separate pieces ("islands", "friend groups").



Connected component

A **maximal** group: every pair inside is connected by some path — and no path leaves the group.

The Algorithm — Solved Completely

1. Scan vertices left to right.

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.
3. Keep scanning; each new launch = one more component.

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.
3. Keep scanning; each new launch = one more component.

Run it on the islands graph:

- launch from A: reaches B, C (triangle) $\Rightarrow$ component $\{A, B, C\}$

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.
3. Keep scanning; each new launch = one more component.

Run it on the islands graph:

- launch from A: reaches B, C (triangle) $\Rightarrow$ component $\{A, B, C\}$
- scan: D unvisited $\rightarrow$ reaches $E \Rightarrow$ component $\{D, E\}$

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.
3. Keep scanning; each new launch = one more component.

Run it on the islands graph:

- launch from A: reaches B, C (triangle) $\Rightarrow$ component $\{A, B, C\}$
- scan: D unvisited $\rightarrow$ reaches $E \Rightarrow$ component $\{D, E\}$
- scan: F unvisited $\rightarrow$ reaches nothing new $\Rightarrow$ component $\{F\}$

The Algorithm — Solved Completely

1. Scan vertices left to right.
2. Unvisited? Launch a BFS/DFS from it — everything it reaches is **one component**.
3. Keep scanning; each new launch = one more component.

Run it on the islands graph:

- launch from A: reaches B, C (triangle) $\Rightarrow$ component $\{A, B, C\}$
- scan: D unvisited $\rightarrow$ reaches $E \Rightarrow$ component $\{D, E\}$
- scan: F unvisited $\rightarrow$ reaches nothing new $\Rightarrow$ component $\{F\}$

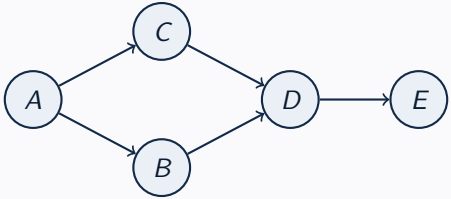
Answer

3 components: $\{A, B, C\}$, $\{D, E\}$, $\{F\}$ — total cost still $\Theta(V + E)$.

Topological Order

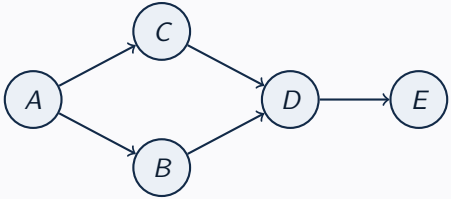
New Twist: One-Way Edges

Courses with prerequisites — arrows mean "**must come before**":



New Twist: One-Way Edges

Courses with prerequisites — arrows mean "**must come before**":

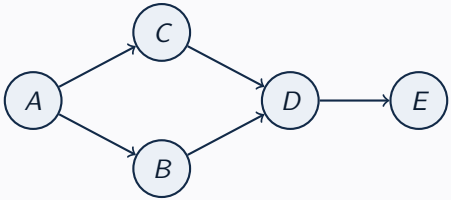


Topological order

List all vertices so that **every arrow points forward** — prerequisite before dependent.

New Twist: One-Way Edges

Courses with prerequisites — arrows mean "**must come before**":



Topological order

List all vertices so that **every arrow points forward** — prerequisite before dependent.

- Works only on **DAG**s (Directed Acyclic Graph — no directed cycles). A cycle = impossible demands!

Pop Quiz

Which of these are valid topological orders of the DAG above?

(1) $A,\ B,\ C,\ D,\ E$ **(2)** $A,\ C,\ B,\ D,\ E$ **(3)** $B,\ A,\ C,\ D,\ E$

Pop Quiz

Which of these are valid topological orders of the DAG above?

(1) $A,\ B,\ C,\ D,\ E$ **(2)** $A,\ C,\ B,\ D,\ E$ **(3)** $B,\ A,\ C,\ D,\ E$

Answer: (1) and (2) — both keep every arrow forward.

Pop Quiz

Which of these are valid topological orders of the DAG above?

(1) $A,\ B,\ C,\ D,\ E$ **(2)** $A,\ C,\ B,\ D,\ E$ **(3)** $B,\ A,\ C,\ D,\ E$

Answer: (1) and (2) — both keep every arrow forward.

Key insight

A topological order is **not unique** — any arrow-respecting order is acceptable.

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

- $A \to B \to D \to E$: E has no outgoing $\to$ **finish** E

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

- $A \to B \to D \to E$: E has no outgoing $\to$ **finish** E
- D done $\to$ **finish** D; B done $\to$ **finish** B

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

- $A \to B \to D \to E$: E has no outgoing $\to$ **finish** E
- D done $\to$ **finish** D; B done $\to$ **finish** B
- back at $A \to C$: both targets finished $\to$ **finish** C, then A

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

- $A \to B \to D \to E$: E has no outgoing $\to$ **finish** E
- D done $\to$ **finish** D; B done $\to$ **finish** B
- back at $A \to C$: both targets finished $\to$ **finish** C, then A

Finish order: $E,\ D,\ B,\ C,\ A$

Method 1: DFS + Reverse Finish Order

Run DFS from A (alphabetical neighbors); when a vertex **finishes** (all outgoing done), push it on a list.

- $A \to B \to D \to E$: E has no outgoing $\to$ **finish** E
- D done $\to$ **finish** D; B done $\to$ **finish** B
- back at $A \to C$: both targets finished $\to$ **finish** C, then A

Finish order: $E,\ D,\ B,\ C,\ A$

Reverse it

$A,\ C,\ B,\ D,\ E$ — a valid topological order! ✓

Method 2: Peel the Sources (Kahn)

A **source** = vertex with no incoming arrow — safe to schedule next.

Method 2: Peel the Sources (Kahn)

A **source** = vertex with no incoming arrow — safe to schedule next.

- sources: $\{A\} \rightarrow$ output A, delete it $\rightarrow$ sources: $\{B, C\}$

Method 2: Peel the Sources (Kahn)

A **source** = vertex with no incoming arrow — safe to schedule next.

- sources: $\{A\} \rightarrow$ output A, delete it $\rightarrow$ sources: $\{B, C\}$
- output $B \rightarrow D$ becomes a source

Method 2: Peel the Sources (Kahn)

A **source** = vertex with no incoming arrow — safe to schedule next.

- sources: $\{A\} \rightarrow$ output A, delete it $\rightarrow$ sources: $\{B, C\}$
- output $B \rightarrow D$ becomes a source
- output C, then D, then E

Method 2: Peel the Sources (Kahn)

A **source** = vertex with no incoming arrow — safe to schedule next.

- sources: $\{A\}$ → output A, delete it → sources: $\{B, C\}$
- output $B \to D$ becomes a source
- output C, then D, then E

Result

$A,\ B,\ C,\ D,\ E$ — also valid! (Two methods, two legal orders.)

Bonus: Catching a Cycle

Add one arrow: $E \to B$.

Bonus: Catching a Cycle

Add one arrow: $E \to B$.
Kahn's method: remove A ... then:

- B: incoming from E — stuck
- D: incoming from B — stuck
- E: incoming from D — stuck

Bonus: Catching a Cycle

Add one arrow: $E \to B$.
Kahn's method: remove A … then:

- B: incoming from E — stuck
- D: incoming from B — stuck
- E: incoming from D — stuck

No source remains, but vertices are left over:

$$B \to D \to E \to B \qquad \textbf{cycle!}$$

Bonus: Catching a Cycle

Add one arrow: $E \to B$.
Kahn's method: remove A … then:

- B: incoming from E — stuck
- D: incoming from B — stuck
- E: incoming from D — stuck

No source remains, but vertices are left over:

$$B \to D \to E \to B \qquad \textbf{cycle!}$$

For free

Kahn's method doubles as a **cycle detector**: leftovers form the cycle.

Wrap-Up

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).
- **BFS**: queue, rings, gives **shortest unweighted** paths.

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).
- **BFS**: queue, rings, gives **shortest unweighted** paths.
- **DFS**: recursion, one path at a time, backtracks.

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).
- **BFS**: queue, rings, gives **shortest unweighted** paths.
- **DFS**: recursion, one path at a time, backtracks.
- **Components**: launch a traversal from every unvisited vertex; each launch = one island.

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).
- **BFS**: queue, rings, gives **shortest unweighted** paths.
- **DFS**: recursion, one path at a time, backtracks.
- **Components**: launch a traversal from every unvisited vertex; each launch = one island.
- **Topological order**: DFS-reverse-finish or Kahn's sources; leftovers = **cycle**.

Summary

- Graph = vertices + edges; store as an **adjacency list** ($\Theta(V+E)$).
- **BFS**: queue, rings, gives **shortest unweighted** paths.
- **DFS**: recursion, one path at a time, backtracks.
- **Components**: launch a traversal from every unvisited vertex; each launch = one island.
- **Topological order**: DFS-reverse-finish or Kahn's sources; leftovers = **cycle**.
- Every algorithm today: $\Theta(V+E)$.

Next session

Weighted shortest paths: Dijkstra's algorithm.

Exercises

1. Graph edges: A–B, B–C, C–D, A–E, E–D. Run BFS and DFS from A (alphabetical neighbors); give both orders and all distances.
2. Find the connected components of: A–B, C–D, D–E, F–G.
3. List **all** valid topological orders of: $A \to B$, $A \to C$, $B \to D$, $C \to D$.
4. Run Kahn's method on: $A \to B$, $B \to C$, $C \to A$, and a separate vertex D. What do you find?
5. **(Challenge)** Explain why BFS's distance labels are always the true fewest-edge distances.