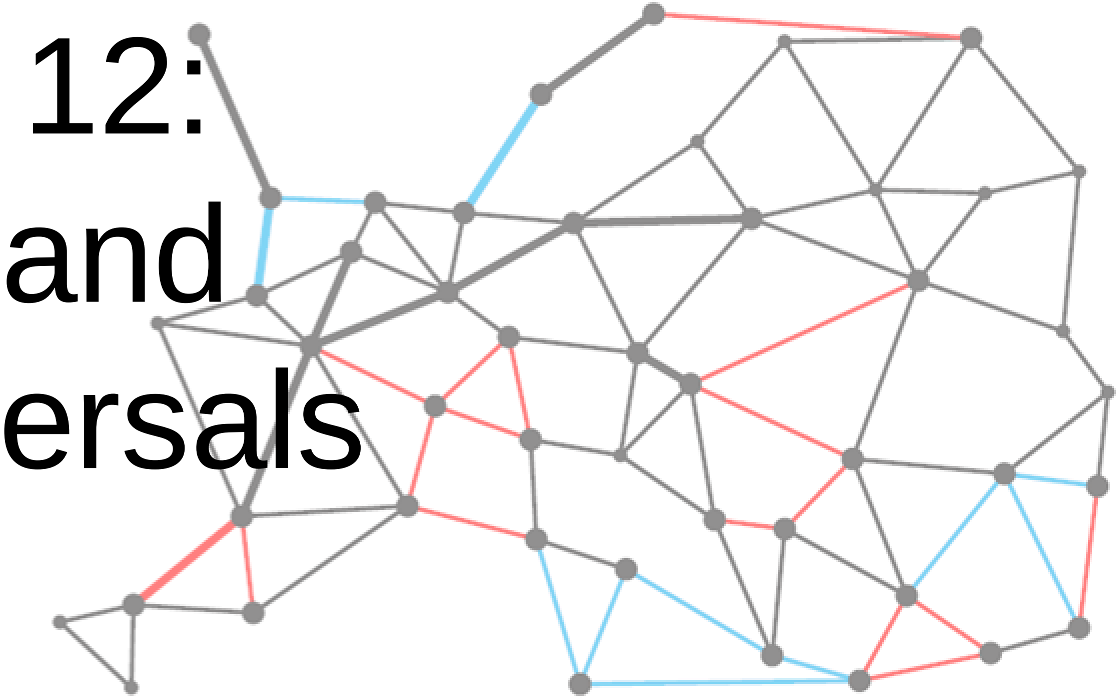
Lecture 12: Graphs and Their Traversals



These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

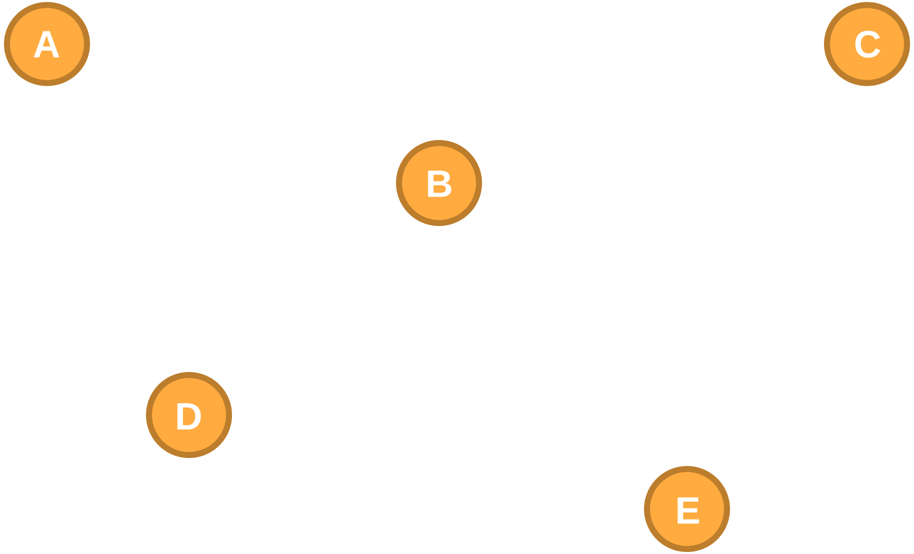
- Lab 11 pre-lab due tonight; post-lab and code due 11/27
  - exists() method bugfix: see Piazza post from Prof. Cole
- Exam 2 graded: regrade requests open until Sunday night
- Lab 6 regrade requests re-opened until tomorrow night
  - If your grade wasn't posted before last Sunday at 12 am

- Exam 3 Wednesday, May 1st 10 am

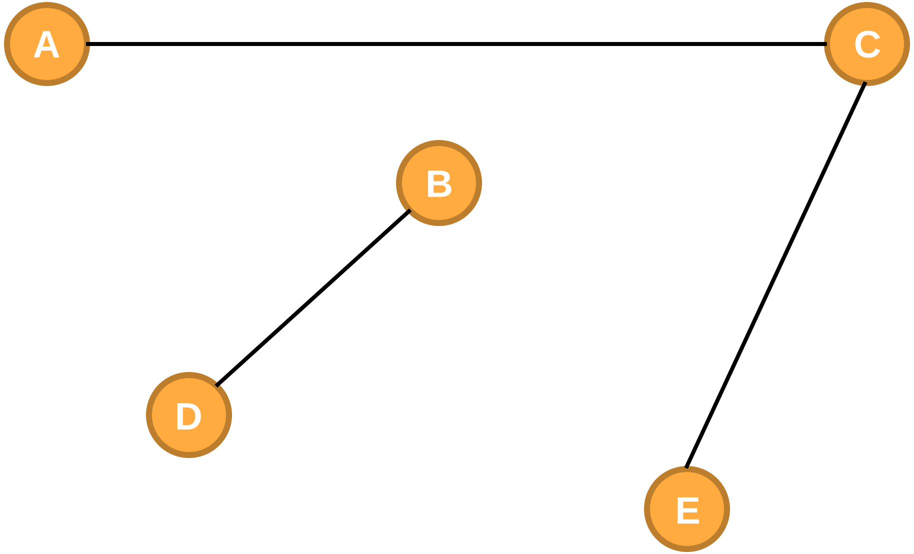
Review: What is a Graph?

- Collections describe groups of objects / entities
- But sometimes, we also want to describe relationships among objects
- A **graph** is a way of describing **pairwise** relationships among a set of objects.

Objects

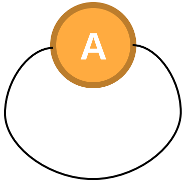


Relationships Among Pairs of Objects



Graphs: Some Definitions

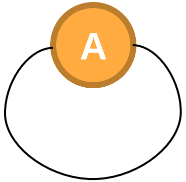
- A **graph** G = (V,E) is a set V of nodes or vertices, together with a set E of edges (described as pairs of vertices)
- Each pair of vertices u and v *may* be connected by an edge (u,v), or not.
- *Optional*: are self-edges (u,u) allowed?



Graphs: Some Definitions

- A **graph** G = (V vertices, together with a set E of of vertices)
- Each pair of ve nected by an edge (u,v), or r
- *Optional*: are self-edges (u,u) allowed?

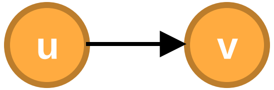
By default, we will assume self-edges are not allowed in our graphs. Such graphs are sometimes called “simple”.



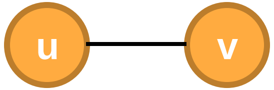
Directions in Graphs

- *Is (u,v) the same edge as (v,u)?*

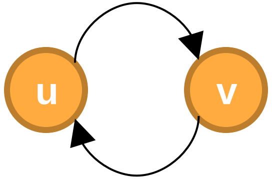
- **No**: graph is directed



- **Yes**: graph is undirected



- A directed graph may have either or both edges (u,v) and (v,u)



Which Kind of Graph Might We Use?

- Railroad lines connecting cities (A connected to B)
- Currency transactions (A sells a stock to B)
- Compatible pairings for tennis doubles match (A can play together with B)
- Web page references (A links to B)
- Road map (Can drive from A to B)

Which Kind of Graph Might We Use?

- Railroad lines connecting cities (A connected to B) [undirected]
- Currency transactions (A sells a stock to B) [directed]
- Compatible pairings for tennis doubles match (A can play together with B) [undirected]
- Web page references (A links to B) [directed]
- Road map (Can drive from A to B) [??? – one way streets?]

Which Kind of Graph Might We Use?

- Railroad lines connecting cities (A connected to B) [undirected]
- Currency tran cted]
- Compatible p can play together with B) [undir
- Web page ref

If the relationship is asymmetric (A → B does not imply B → A), then a directed graph makes sense. If it is symmetric, an undirected graph makes sense.

- Road map (Can drive from A to B) [??? – one way streets?]

How Many Edges Can a Graph Have?

- If a (simple) graph has n vertices...
- If directed, max # of edges is **???**

How Many Edges Can a Graph Have?

- If a (simple) graph has n vertices...
- If directed, max # of edges is **n(n-1)**
- If undirected, max # of edges is **???**

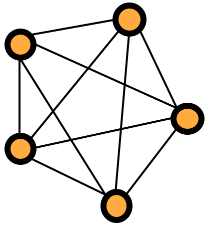
How Many Edges Can a Graph Have?

- If a (simple) graph has n vertices...
- If directed, max # of edges is **n(n-1)**
- If undirected, max # of edges is **n(n-1)/2**
- In either case, n vertices implies **O(n^2)** edges

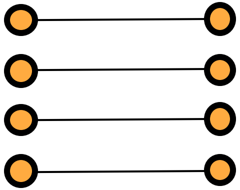
Definitions Related to Edge Count

- If a graph has n vertices...
- If the graph has $\Theta(n^2)$ edges, it is **dense**
- If the graph has $O(n)$ edges, it is **sparse**
- (Some graphs are in between)

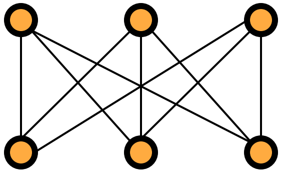
Examples of Dense and Sparse Graph Families



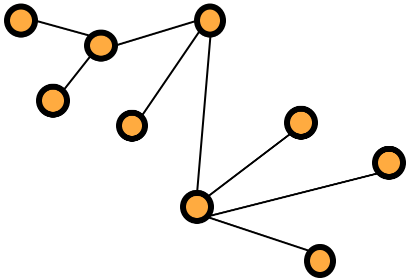
Complete graph



Ladder



Complete bipartite graph

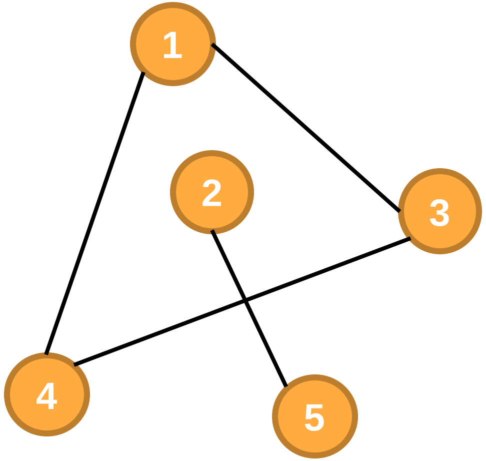


Tree

How Do We Represent Graphs in a Computer?

- Two strategies: adjacency list and adjacency matrix

- **Matrix**: M_{nxn} – M(i,j) is 1 if edge (i,j) exists

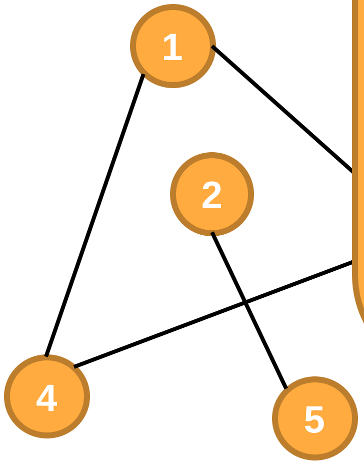


$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

How Do We Represent Graphs in a Computer?

- Two strategies: adjacency list and adjacency matrix

- **Matrix**: M_{nx} s



An adjacency matrix for an undirected graph is always **symmetric**. *Not true for directed graphs.*

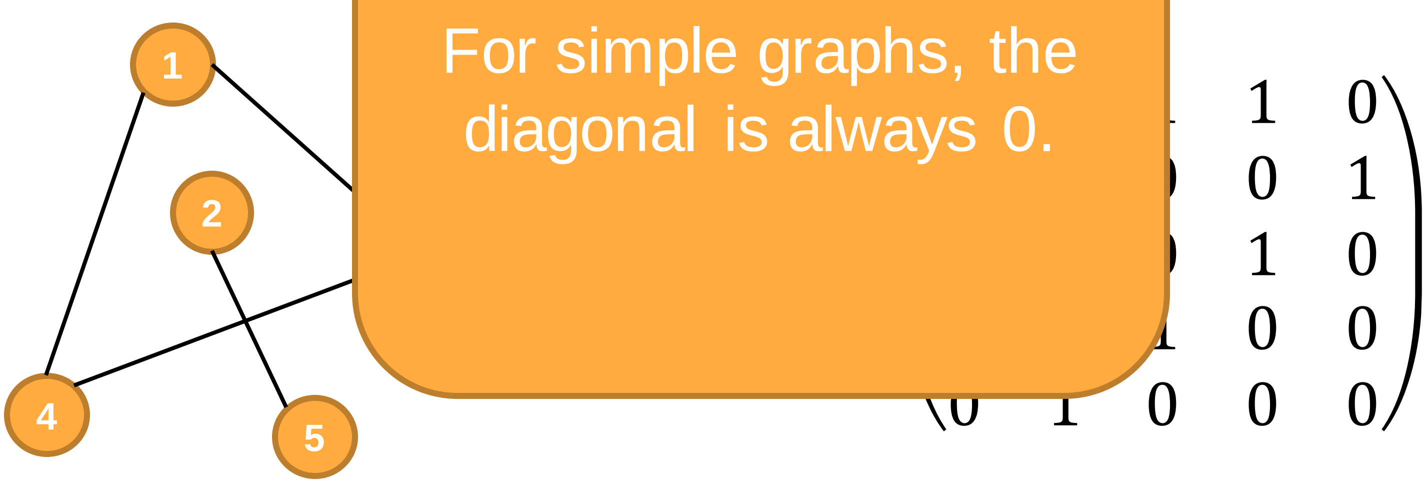
1 0
0 1
1 0
0 0
0 1 0 0 0

How Do We Represent Graphs in a Computer?

- Two strategies: adjacency list and adjacency matrix

- **Matrix**: M_{nx} s

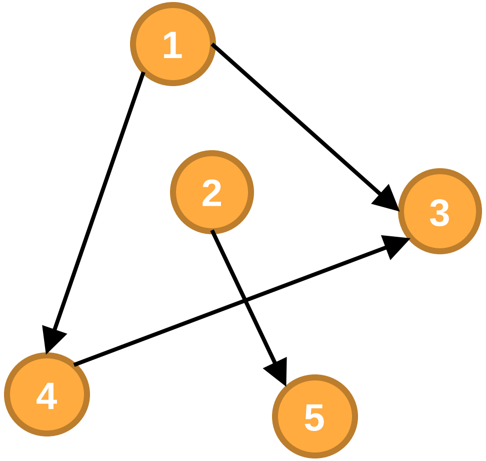
For simple graphs, the diagonal is always 0.



How Do We Represent Graphs in a Computer?

- Two strategies: adjacency list and adjacency matrix

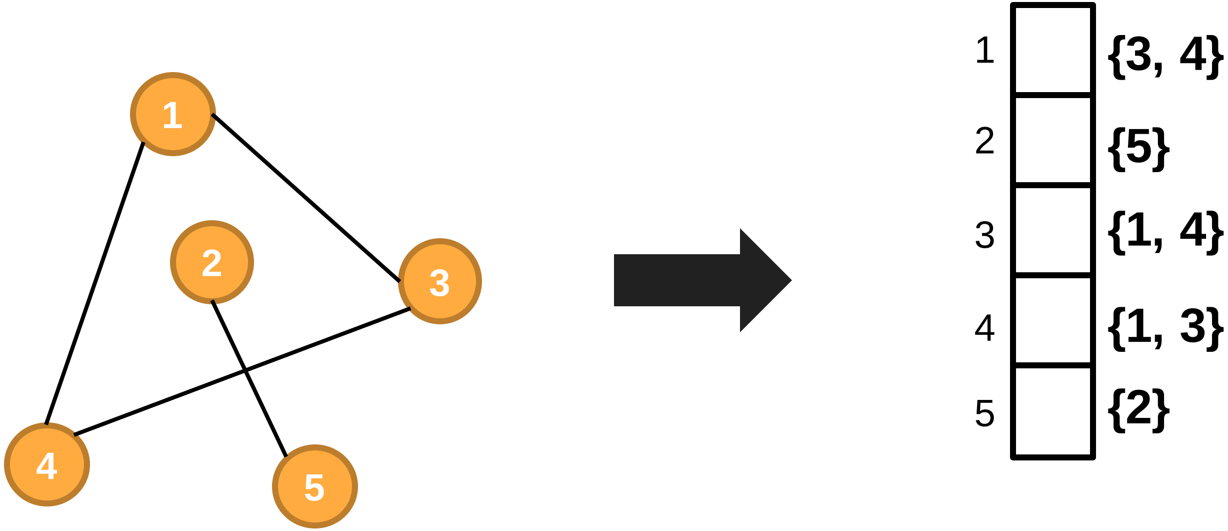
- **Matrix**: M_{nxn} – M(i,j) is 1 if edge (i,j) exists



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

How Do We Represent Graphs in a Computer?

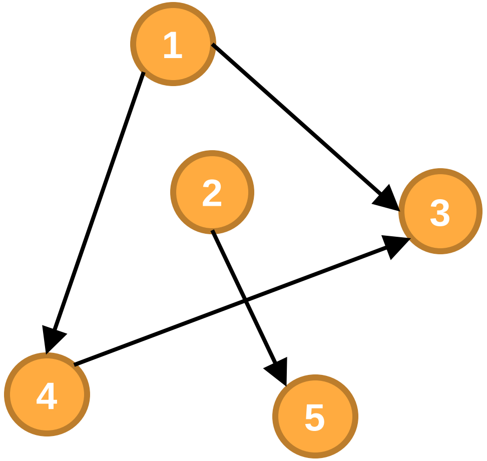
- **List**: Array A[1..n] – A[i] contains list of edges (i,j)



1	**{3, 4}**
2	**{5}**
3	**{1, 4}**
4	**{1, 3}**
5	**{2}**

How Do We Represent Graphs in a Computer?

- **List**: Array A[1..n] – A[i] contains list of edges (i,j)



1	{3, 4}
2	{5}
3	{ }
4	{3}
5	{ }

Properties of Adjacency List vs Matrix

● For graph **G = (V,E)**	**List**	**Matrix**
● **Space to represent G**	**???**	**???**
● **Time to check if edge (u,v) exists**		
● **Time to enumerate all edges in G**		

Properties of Adjacency List vs Matrix

● For graph **G = (V,E)**	**List**	**Matrix**
● **Space to represent G**	**$\Theta(\|V\|+\|E\|)$**	**$\Theta(\|V\|^2)$**
● **Time to check if edge (u,v) exists**	**???**	**???**
● **Time to enumerate all edges in G**		

Properties of Adjacency List vs Matrix

For graph **G = (V,E)**	**List**	**Matrix**
Space to represent G	**$\Theta(\lvert V\rvert+\lvert E\rvert)$**	**$\Theta(\lvert V\rvert^2)$**
Time to check if edge (u,v) exists	**$\Theta(\lvert E\rvert)$\***	**O(1)**
Time to enumerate all edges in G	**???**	**???**

** More precisely, proportional to # of edges adjacent to u.*

Properties of Adjacency List vs Matrix

● For graph **G = (V,E)**	**List**	**Matrix**
● **Space to represent G**	$\Theta(\lvert V\rvert+\lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$
● **Time to check if edge (u,v) exists**	$\Theta(\lvert E\rvert)$*	$O(1)$
● **Time to enumerate all edges in G**	$\Theta(\lvert V\rvert+\lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$

* *More precisely, proportional to # of edges adjacent to u.*

Properties of Adjacency List vs Matrix

● For graph **G = (V,E**	**Matrix**
● **Space to repres**	$\Theta(\lvert V\rvert^2)$
● **Time to check if**	$O(1)$
● **Time to enumer**	$\Theta(\lvert V\rvert^2)$

Most graph algorithms we'll consider here use the adjacency list.

** More precisely, proportional to # of edges adjacent to u.*

So, What Can We Do With Graphs?

Exploration – Graph Traversals

- *Given a starting vertex v, try to discover every vertex in the graph*
- We can move between vertices *only* by following edges
- When we see a vertex for first time, we mark it to avoid repeated work
- Two basic strategies for traversal
  - **Breadth-first search (BFS)**
  - **Depth-first search (DFS)**
- **These traversals reveal different properties of graph**

BFS: First Come, First Searched

- BFS utilizes a FIFO queue **Q** that tracks vertices to be searched.

- Initially, Q contains *only* starting vertex v, which is marked

```
While Q is not empty
    u ← Q.dequeue()
    for each edge (u,w)
        if w is not marked
            mark w
            Q.enqueue(w)
```

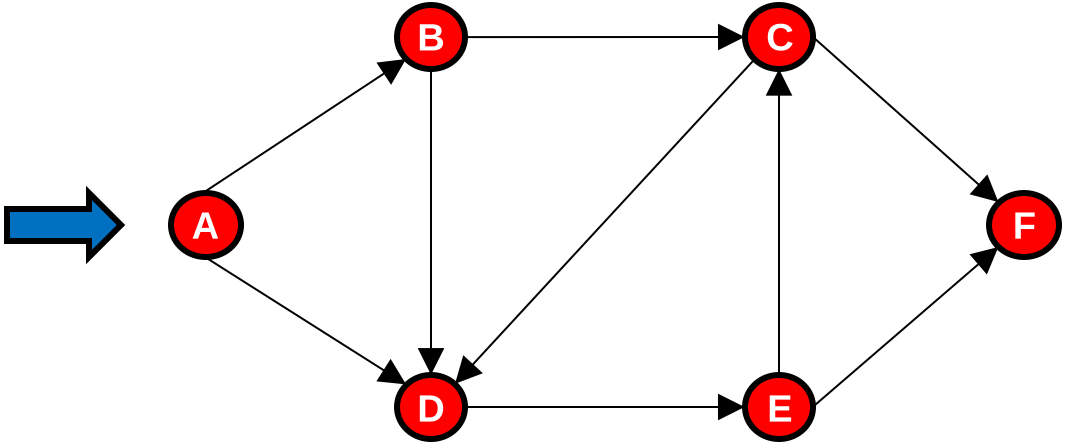
BFS Example

Q

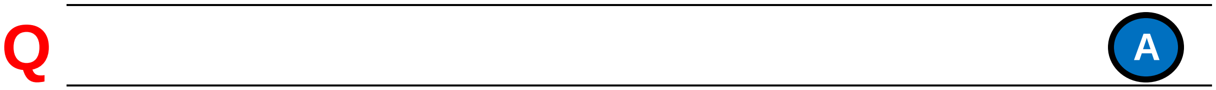
🔴 unmarked

🔵 marked

Start at A

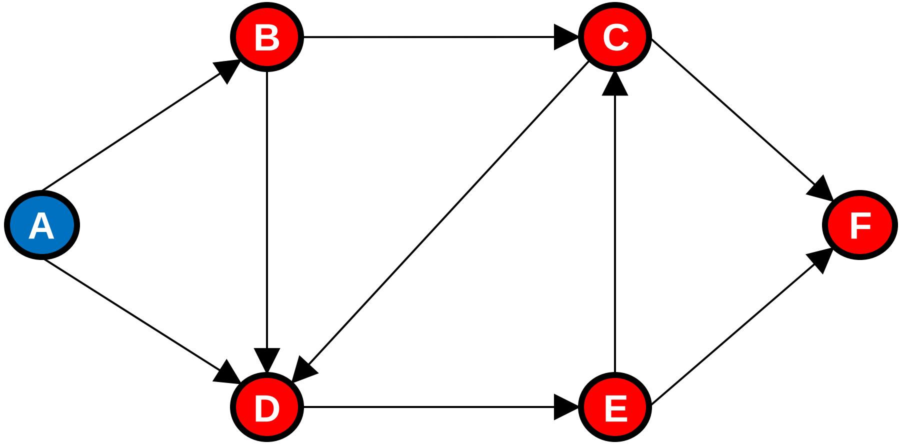


BFS Example



🔴 unmarked

🔵 marked



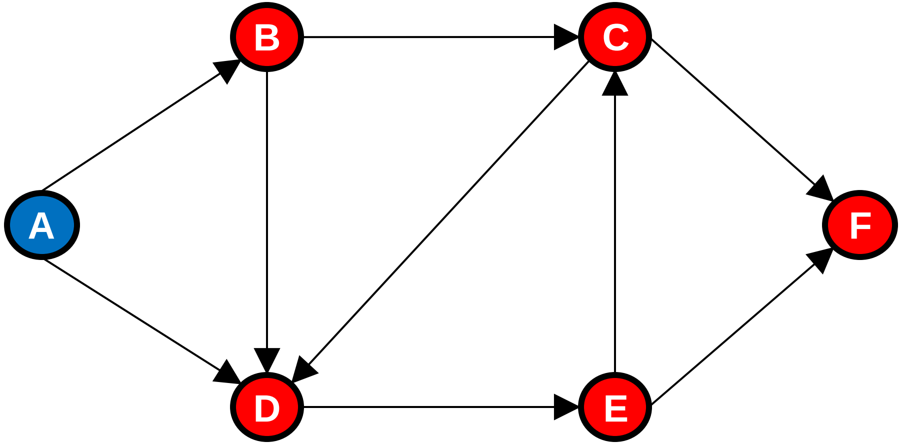
BFS Example

Q

EXPLORE A

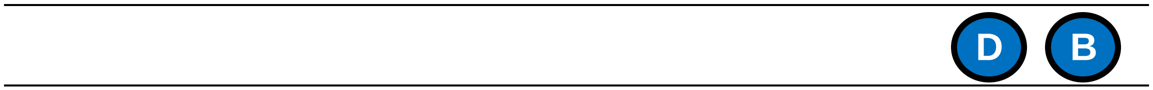
🔴 unmarked

🔵 marked



BFS Example

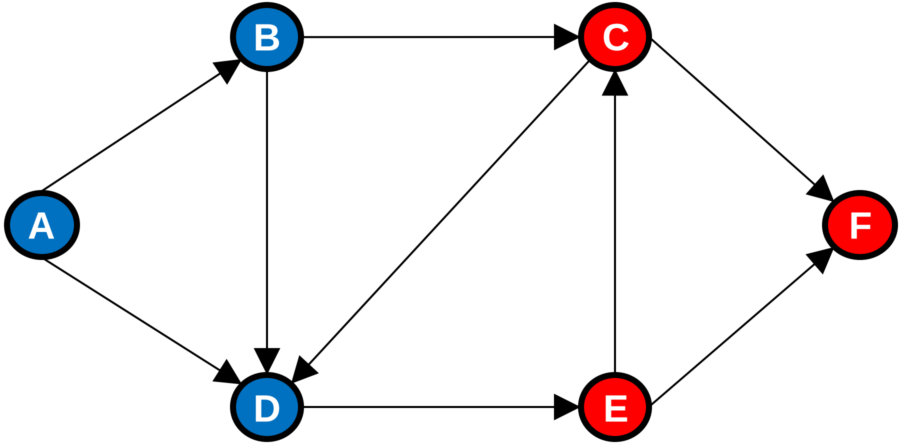
Q



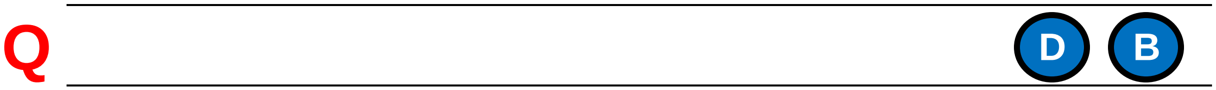
EXPLORE A

🔴 unmarked

🔵 marked



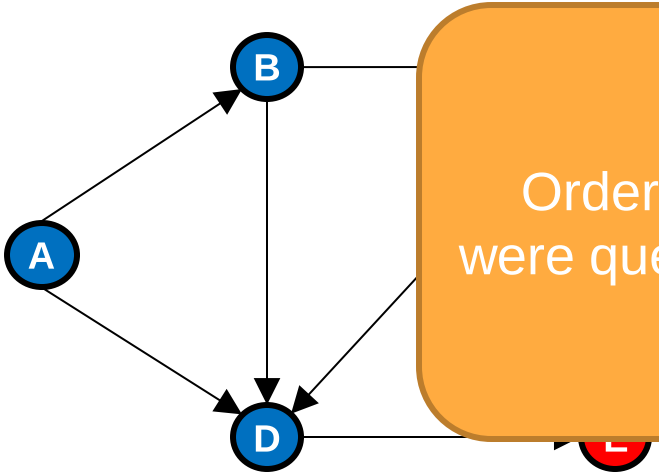
BFS Example



EXPLORE A

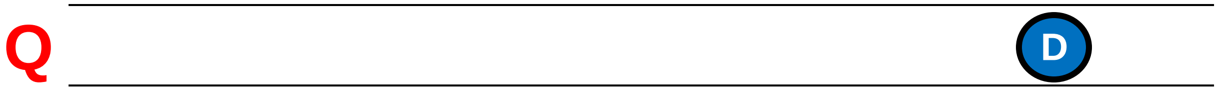
🔴 unmarked

🔵 marked



Order in which B, D were queued is arbitrary.

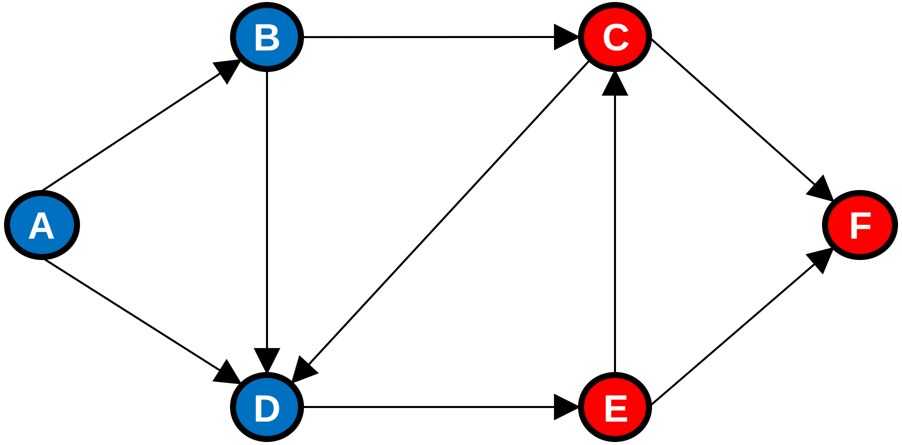
BFS Example



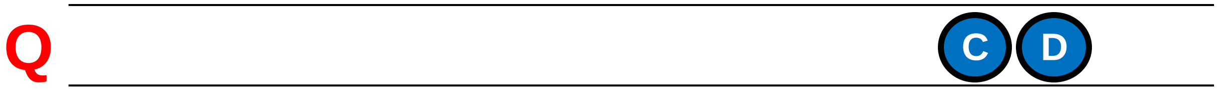
EXPLORE B

🔴 unmarked

🔵 marked



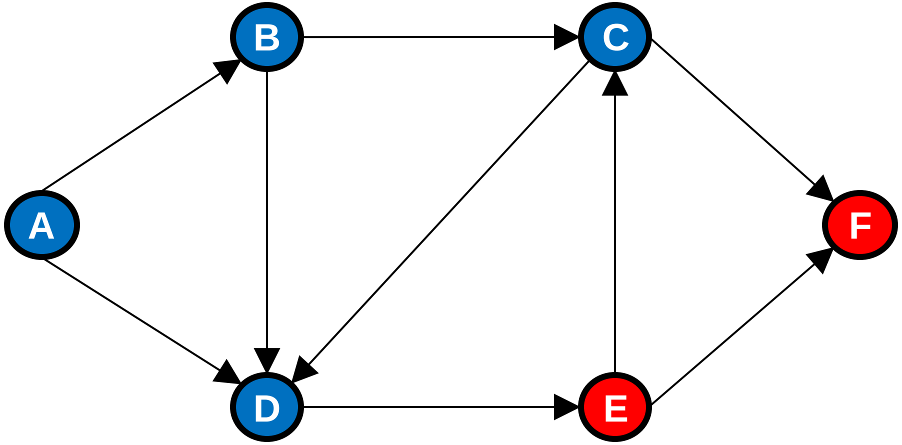
BFS Example



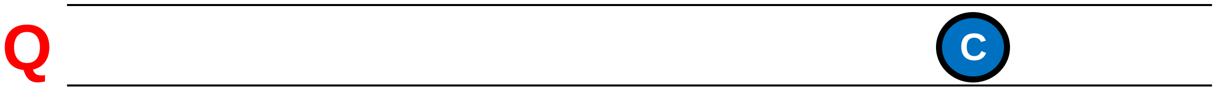
EXPLORE B

🔴 unmarked

🔵 marked



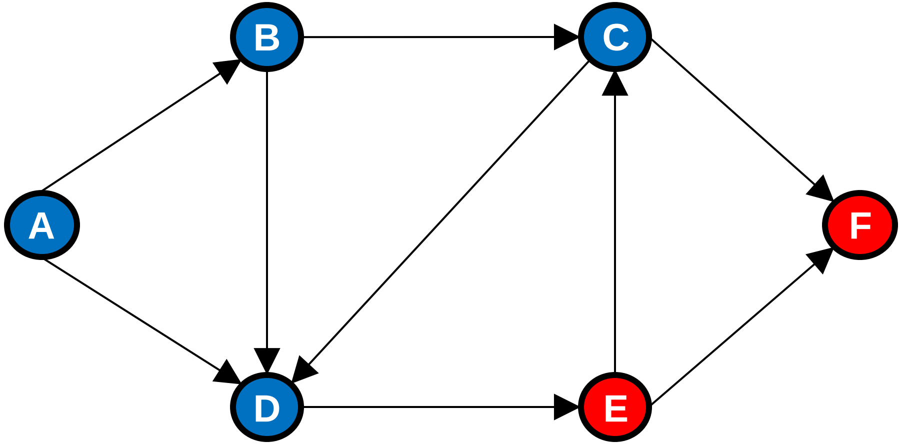
BFS Example



EXPLORE D

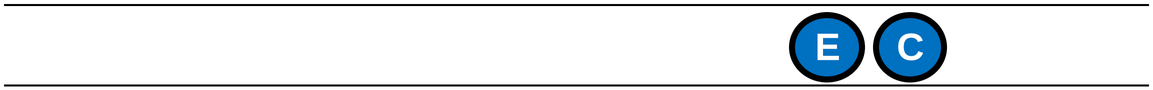
🔴 unmarked

🔵 marked



BFS Example

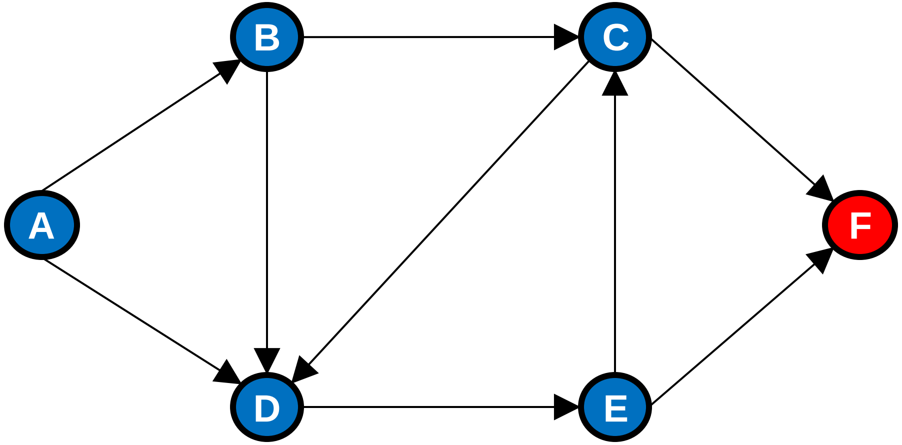
Q



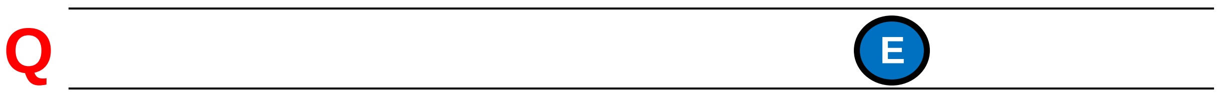
EXPLORE D

🔴 unmarked

🔵 marked



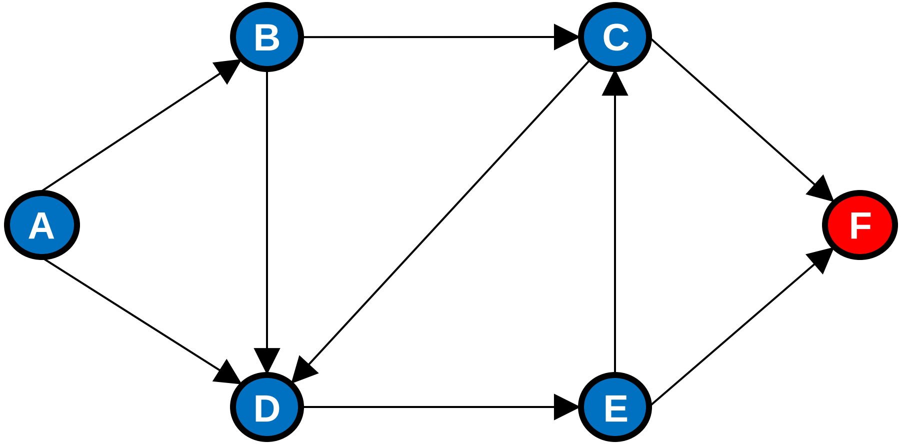
BFS Example



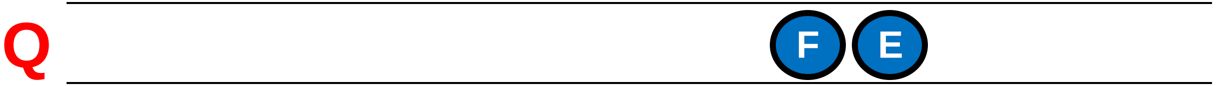
EXPLORE C

🔴 unmarked

🔵 marked



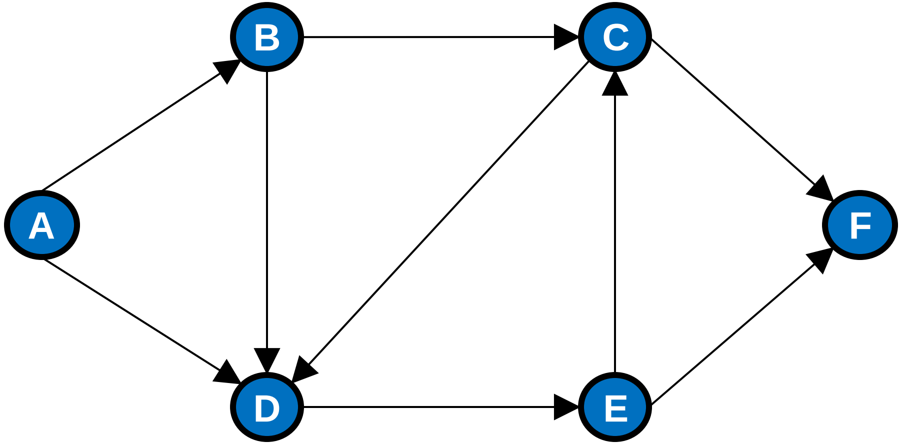
BFS Example



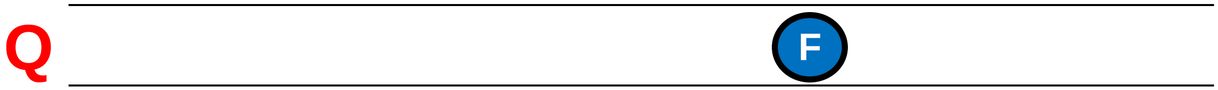
EXPLORE C

🔴 unmarked

🔵 marked



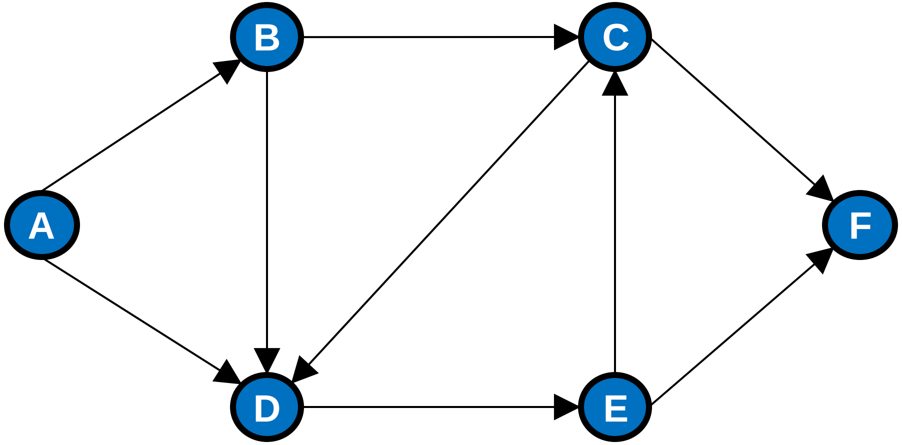
BFS Example



EXPLORE E

🔴 unmarked

🔵 marked



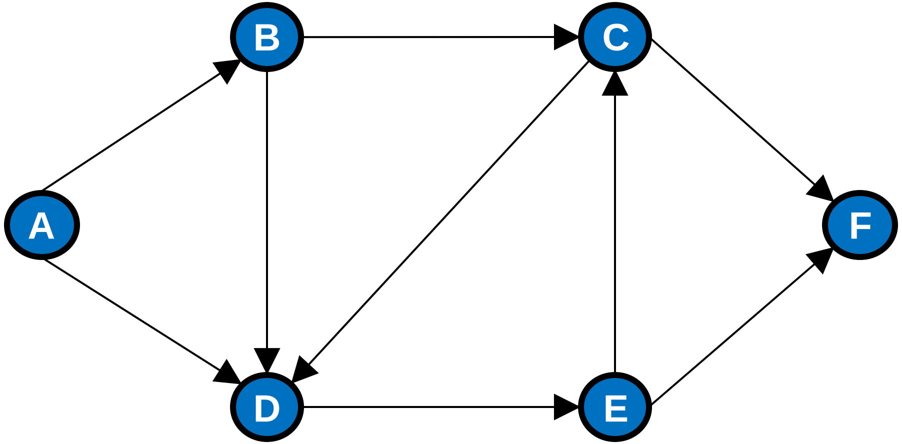
BFS Example

Q

DONE

🔴 unmarked

🔵 marked



What Can We Learn from BFS?

- For any vertices v and u,
  distance D(v,u) = smallest # of edges on any path from v to u.

- By definition, D(v,v) = 0.

- For any fixed v, we can use BFS to compute D(v,u) for all u.

- We can also compute a path from v to each u with D(v,u) edges.

BFS Augmented for Distances, Starting Vertex v

```
mark v; v.distance ← 0; v.parent ← null
Q.enqueue(v)

While Q is not empty
    u ← Q.dequeue()
    for each edge (u,w)
        if w is not marked
            mark w; w.distance ← u.distance + 1; w.parent ← u
            Q.enqueue(w)
```

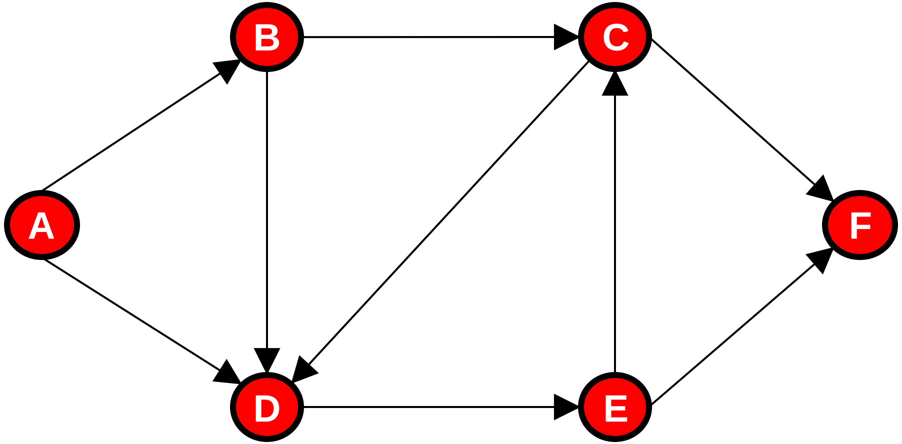
BFS Example

Q

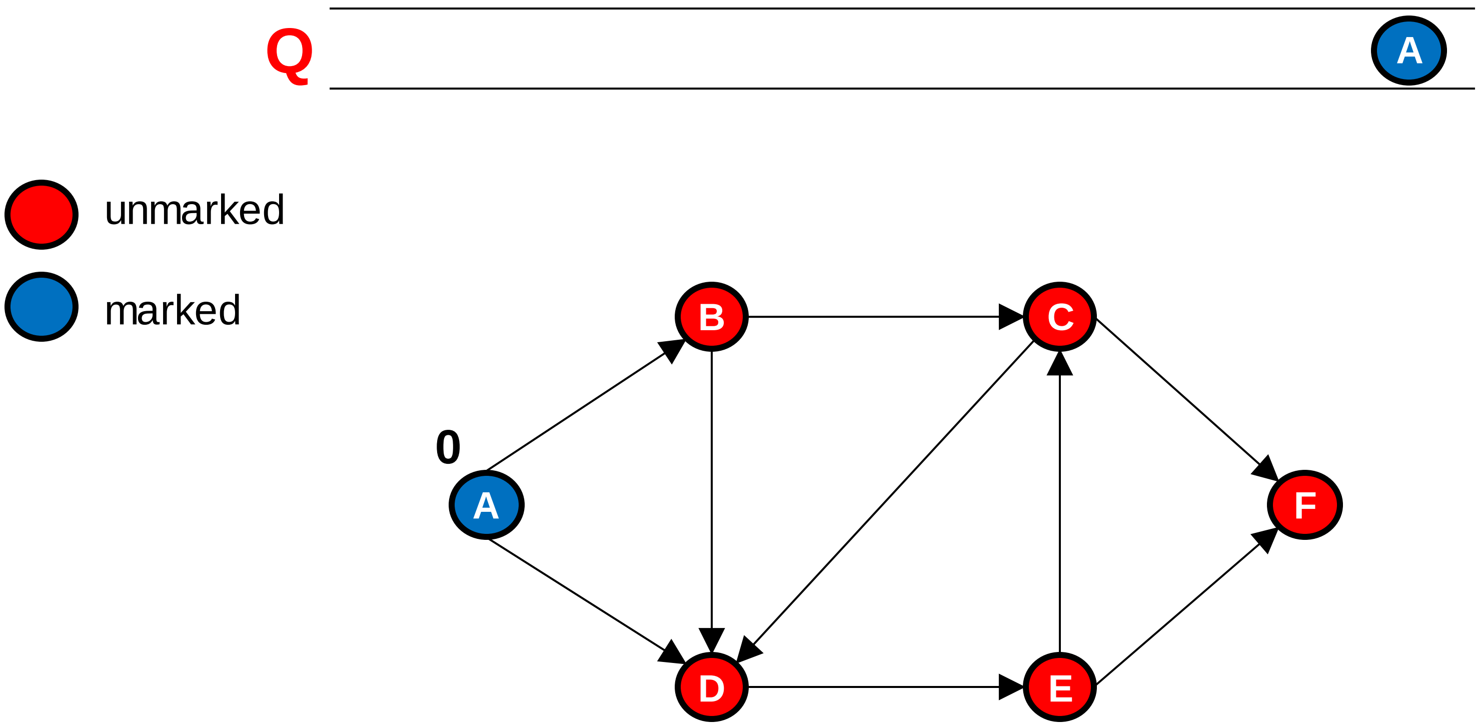
🔴 unmarked

🔵 marked

Start at A ➡



BFS Example



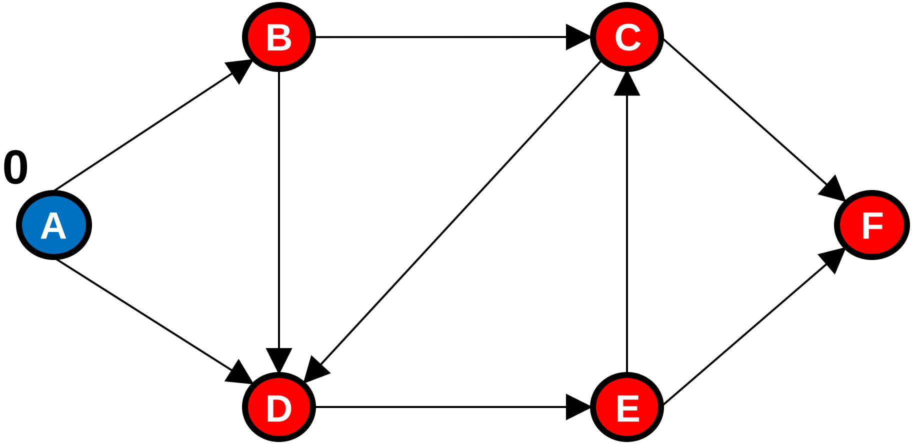
BFS Example

Q

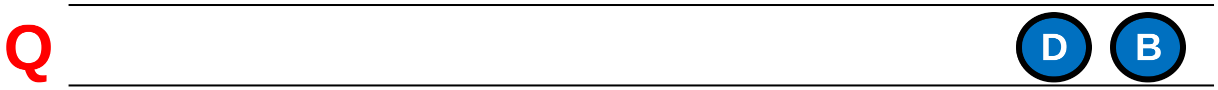
EXPLORE A

🔴 unmarked

🔵 marked



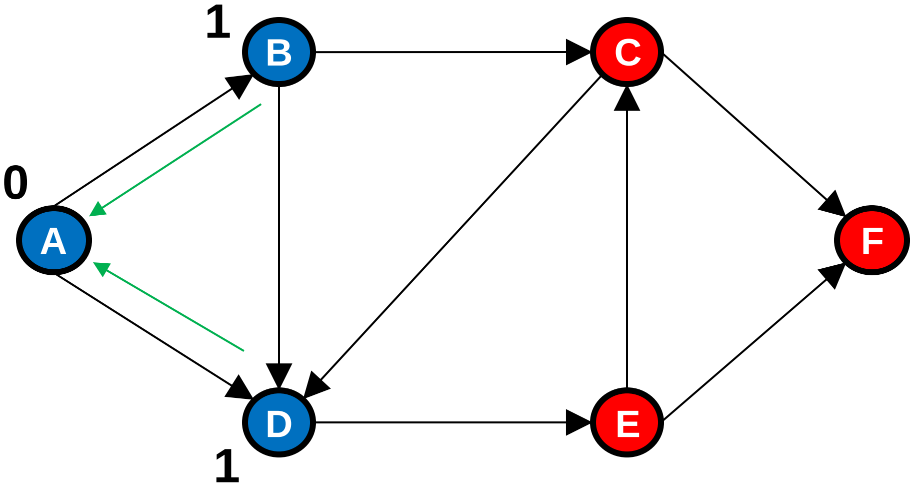
BFS Example



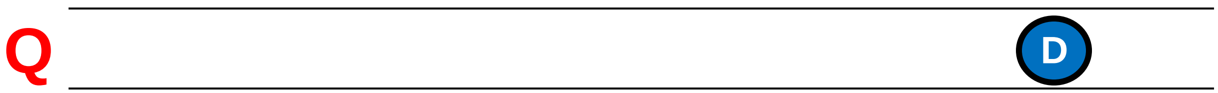
EXPLORE A

🔴 unmarked

🔵 marked



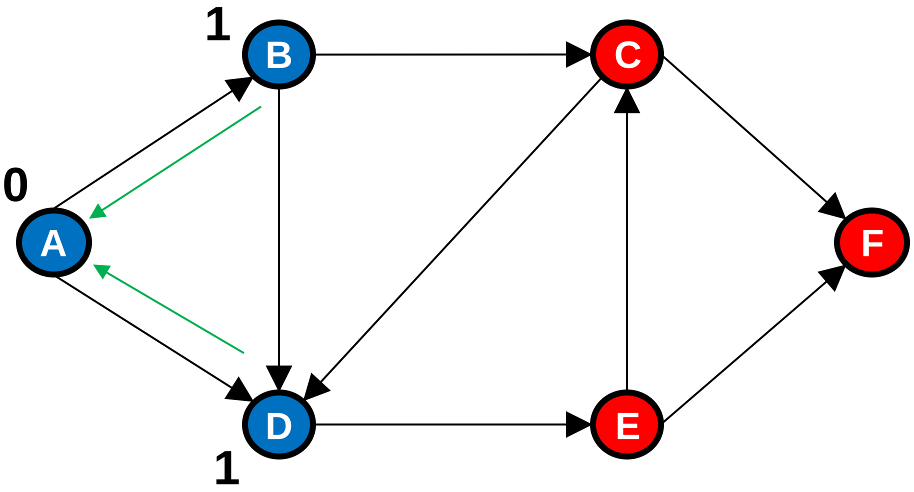
BFS Example



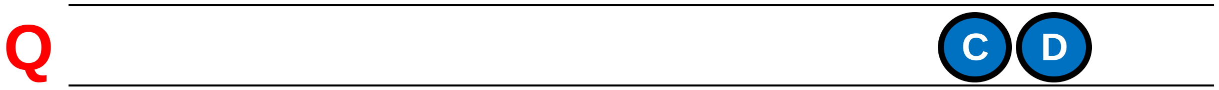
EXPLORE B

🔴 unmarked

🔵 marked



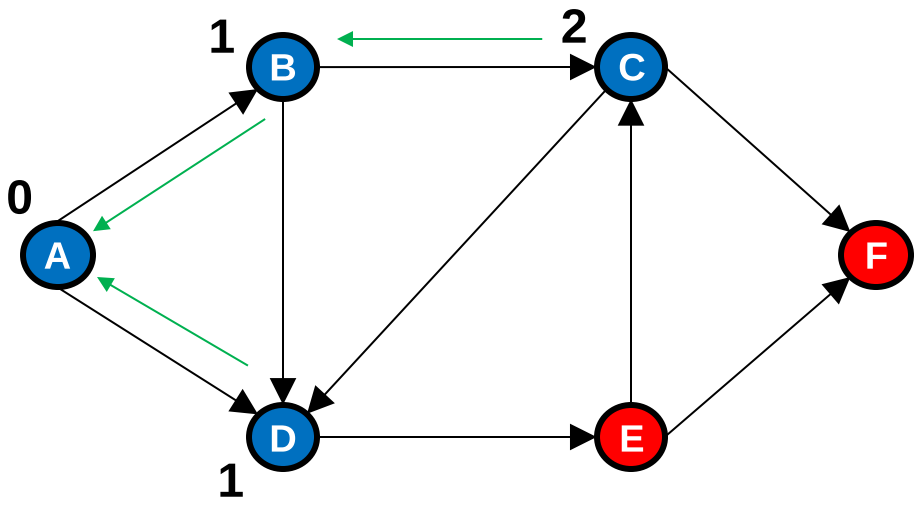
BFS Example



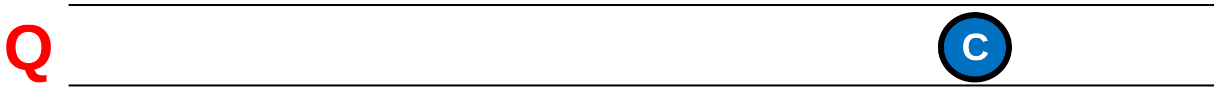
EXPLORE B

🟥 unmarked

🟦 marked



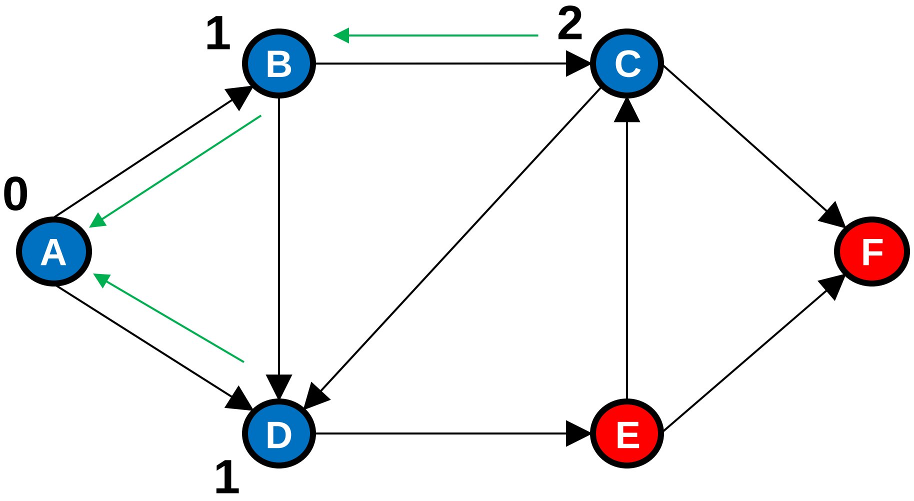
BFS Example



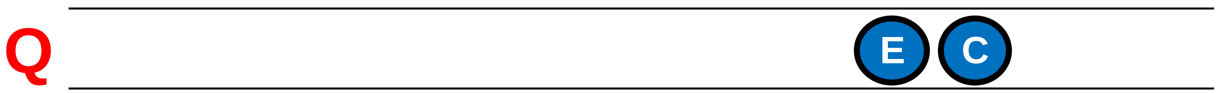
EXPLORE D

🔴 unmarked

🔵 marked



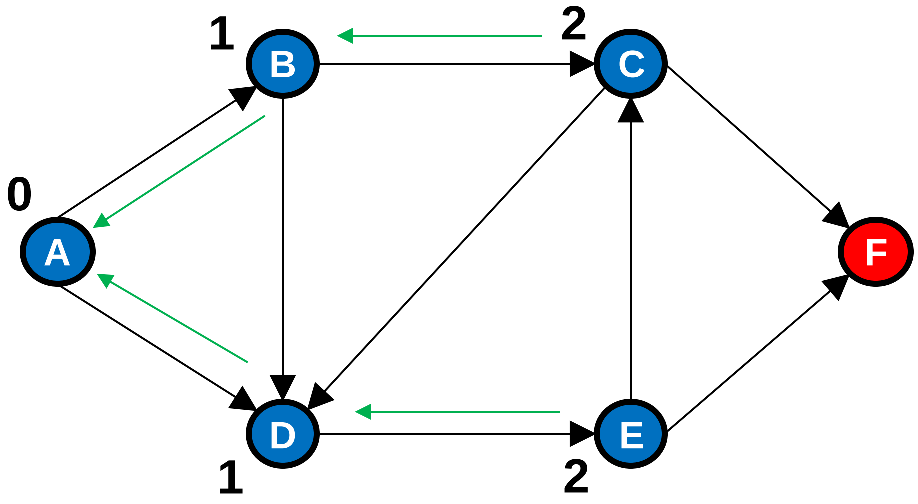
BFS Example



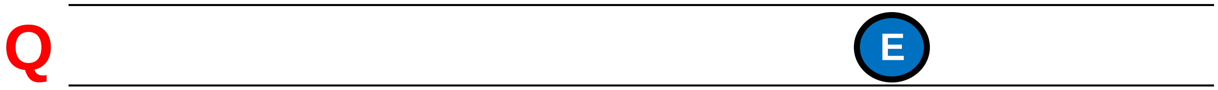
EXPLORE D

🔴 unmarked

🔵 marked



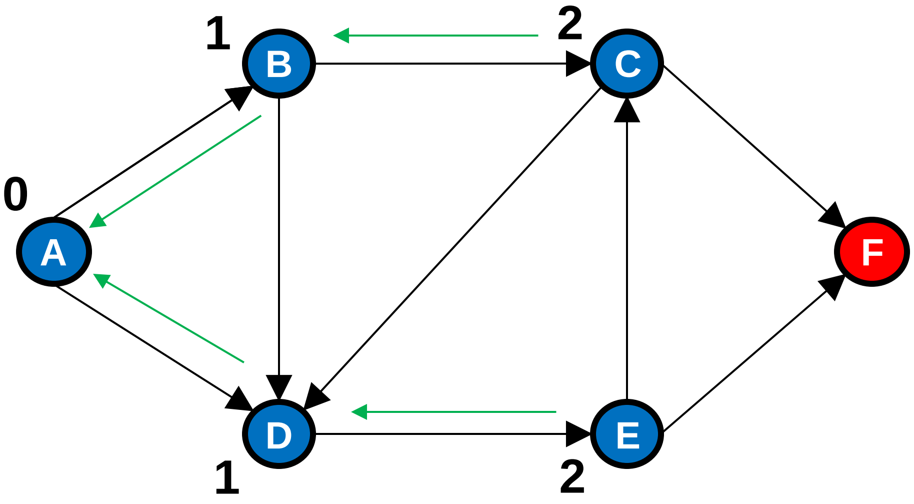
BFS Example



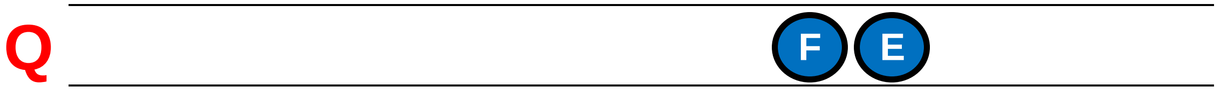
EXPLORE C

🔴 unmarked

🔵 marked



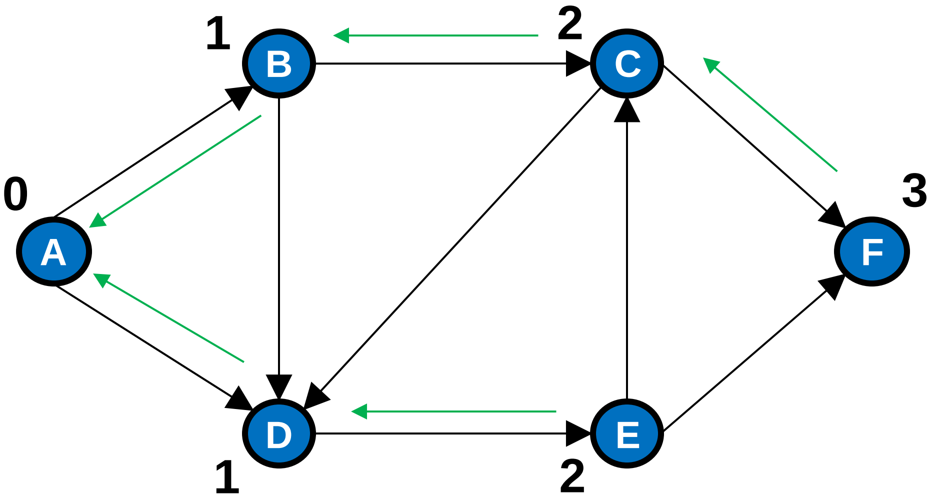
BFS Example



EXPLORE C

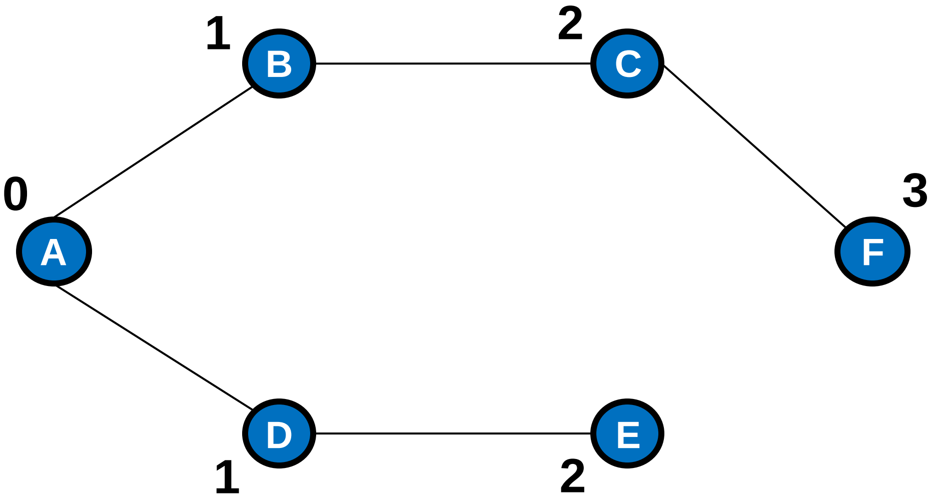
🔴 unmarked

🔵 marked



BFS Example

Parent pointers form a **tree of shortest paths** connecting each vertex to starting point.



BFS Computes Shortest Paths (1/4)

- **Claim**: BFS enqueues **every** vertex w with D(v,w) = d before **any** vertex x with D(v,x) > d.
- **Pf**: by induction on d
- **Bas (d = 0):** v itself is enqueued first and has D(v,v) = 0

BFS Computes Shortest Paths (2/4)

- **Ind:** consider vertex w with D(v,w) = d.
- There is some u s.t. D(v,u) = d-1, and edge (u,w) exists.
- By IH, u is enqueued before any vertex with distance ≥ d.
- Hence, by FIFO property of Q, u is dequeued before any vertex with dist ≥ d.

BFS Computes Shortest Paths (3/4)

- When u is dequeued, *w is enqueued* (if not yet seen)
- Any vertex with distance > d must be discovered via edge from a vertex at distance ≥ d, which is dequeued *after* u.
- *Conclude that no vertex at distance > d will be enqueued prior to w.* **QED**

BFS Computes Shortest Paths (4/4)

- Above argument proves that BFS enqueues vertices in order of distance from v.

- **Corollary**: BFS assigns every vertex its correct shortest-path distance from v.

- **NB**: if graph not **connected**, some vertices may be unreachable from v → *their distances should be* ∞

Cost of BFS

- For every vertex reachable from start, we
  - Mark it; enqueue it; dequeue it (all **O(1)**)
  - Enumerate its adjacent edges (**???**)

Cost of BFS

- For every vertex reachable from start, we
    - Mark it; enqueue it; dequeue it (all **O(1) per vertex, Θ(|V|) total**)
    - Enumerate its adjacent edges (**Θ(|E|) summed over all vertices**)
    - [assuming we use an adjacency list]

- → Total cost is **Θ(|V| + |E|)**

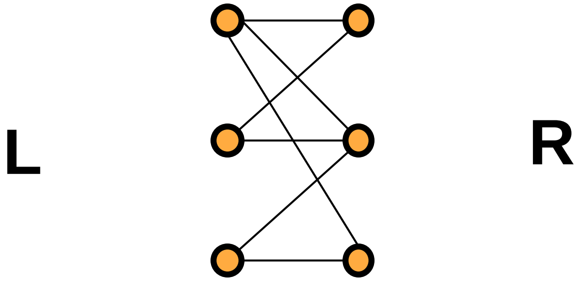
Cost of BFS

- For every vertex reachable from start, we
  - Mark it; en ertex, Θ(|V|) total)
  - Enumerat d over all vertices)
  - [assuming
- → Total cos

Exercise: if we used an adjacency matrix, how would the algorithm's cost change?

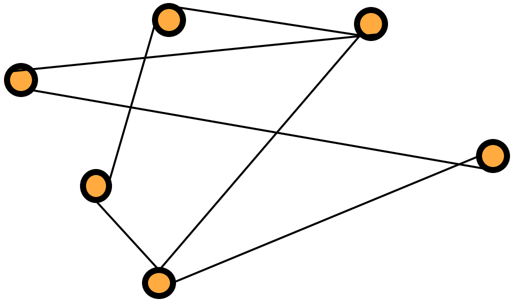
Example Application: Bipartite Testing

- A bipartite graph consists of two sets L, R of vertices, s.t. all edges go between L and R.



Example Application: Bipartite Testing

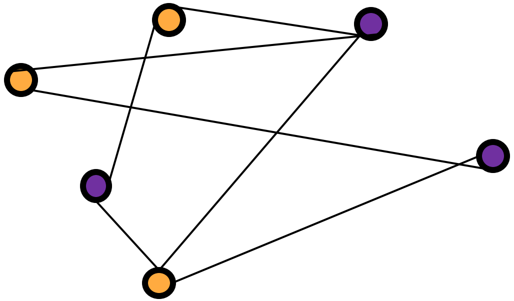
- A bipartite graph consists of two sets L, R of vertices, s.t. all edges go between L and R.



How can we tell if an arbitrary graph is bipartite?

Example Application: Bipartite Testing

- A bipartite graph consists of two sets L, R of vertices, s.t. all edges go between L and R.

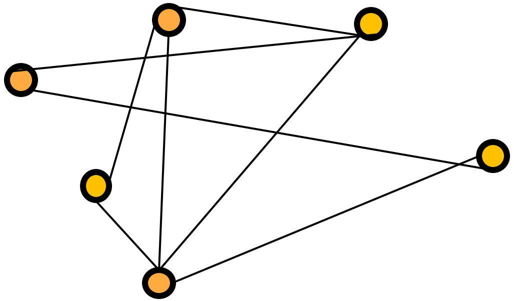


Yes!

How can we tell if an arbitrary graph is bipartite?

Example Application: Bipartite Testing

- A bipartite graph consists of two sets L, R of vertices, s.t. all edges go between L and R.



No!

How can we tell if an arbitrary graph is bipartite?

Idea: Use BFS to Label Two Sides of Graph

- Pick arbitrary starting vertex v; label v to be on side L.

- Run BFS. If we discover vertex w via edge (u,w), label w to be on opposite side from u.

- Claim: graph is bipartite iff BFS never labels both endpoints of an edge (u,w) with same side.

Proof Idea of Claim

- Claim: graph is bipartite iff BFS never labels both endpoints of an edge (u,w) with same side.

- Can show that a graph is bipartite iff it contains no odd-length cycle (e.g. a triangle).

- If not bipartite, impossible to label vertices of odd cycle L or R w/o labeling both endpoints of some edge the same.

- If bipartite, vertices on side L are at even distance from start, while those on side R are at odd distance, so labels will be consistent.

And Now for Something Completely Different…

DFS: First Started, Last Finished

- DFS finds all vertices reachable from a given v before completing v.

- Instead of simply marking vertices, we assign them two integer *times:*
  - Time at which we first discover vertex (v.start)
  - Time at which we complete vertex (v.finish)

- (Time “ticks” after each assignment to a vertex.)

DFS Pseudocode (Recursive)

- Once again, pick a starting vertex v.
- Set global **time** variable = 0

```
DFSVisit(v)
    v.start ← time++
    for each edge (v,u)
        if (u.start is not yet set)
            DFSVisit(u)
    v.finish ← time++
```

DFS Pseudocode (Recursive)

- Once again, pick a starting vertex v.
- Set global **time** variable = 0

```
DFSVisit(v)
    v.start ← time++
    for each edge (v,u)
        if (u.start is not yet set)
            DFSVisit(u)
    v.finish ← time++
```

Recursive code implicitly uses a stack; could implement with explicit stack (vs queue for BFS)

DFS Example

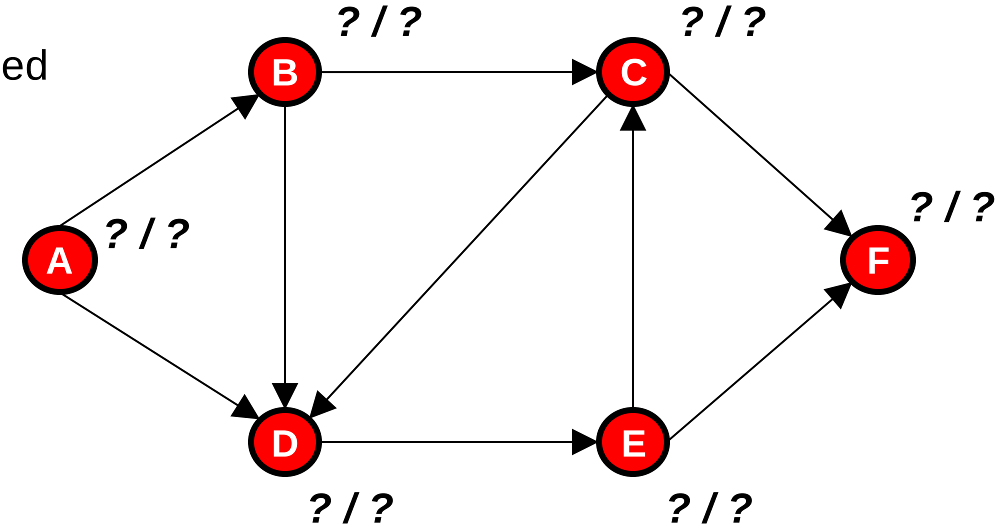
Time = 0

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



start / finish

⚪

DFS Example

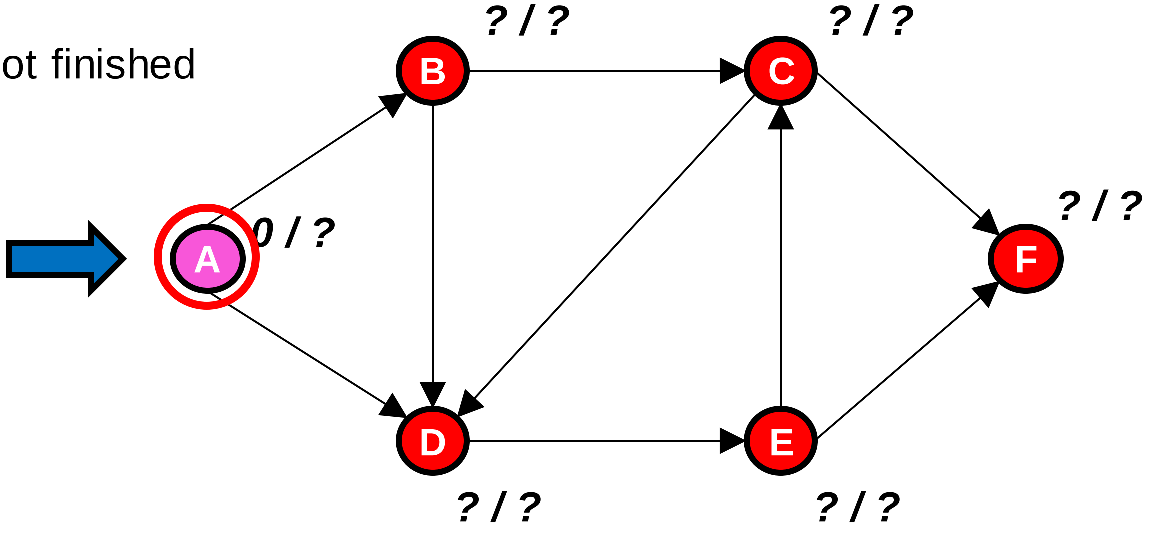
Time = 1

🔴 not started

🟣 started, not finished

🔵 finished

Start at A



⚪ *start / finish*

DFS Example

Time = 2

🔴 not started

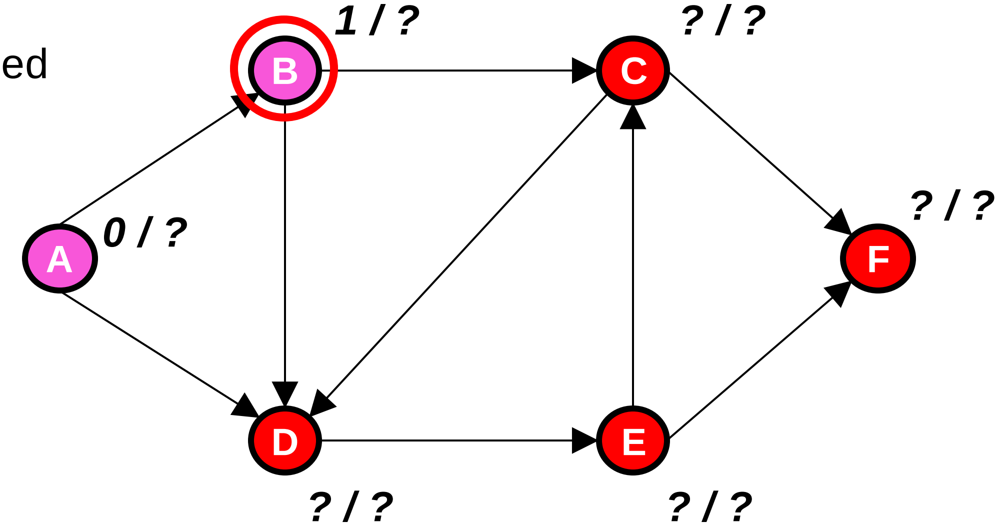
🟣 started, not finished

🔵 finished

Start at A ➡

start / finish

⚪



DFS Example

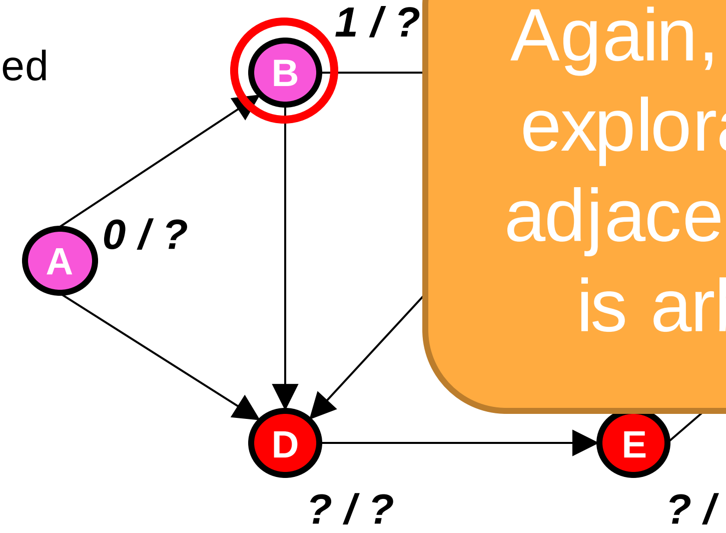
Time = 2

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



Again, order of exploration for adjacent edges is arbitrary.

start / finish

⚪

DFS Example

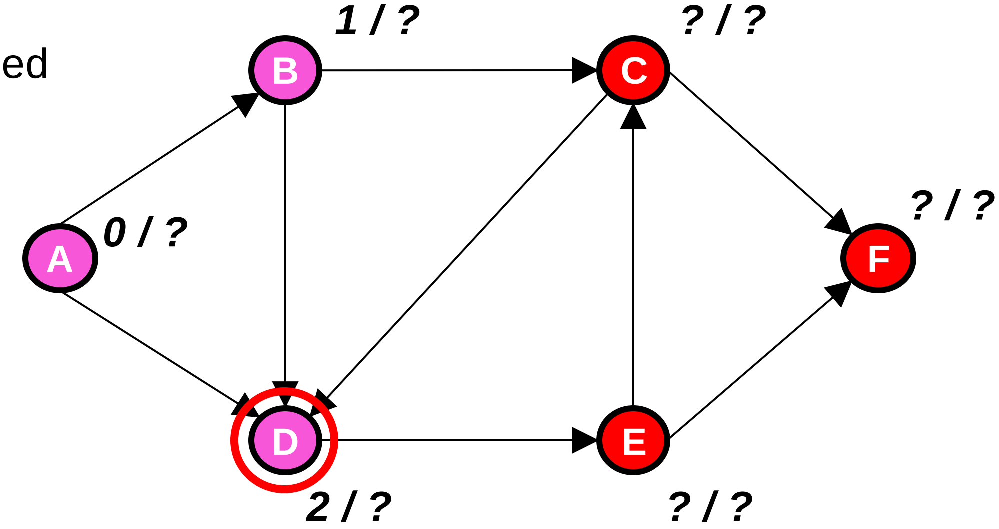
Time = 3

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



start / finish

⚪

DFS Example

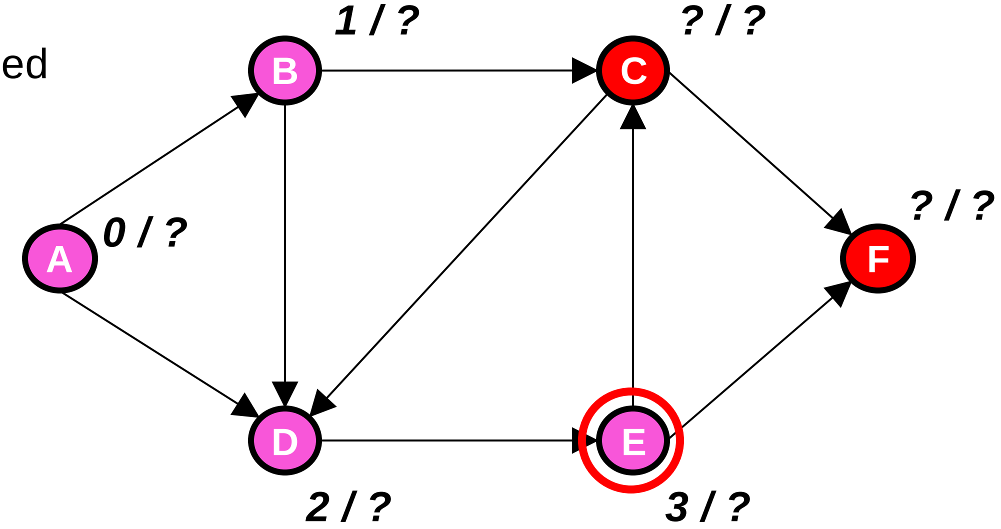
Time = 4

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



○ *start / finish*

DFS Example

Time = 5

🔴 not started

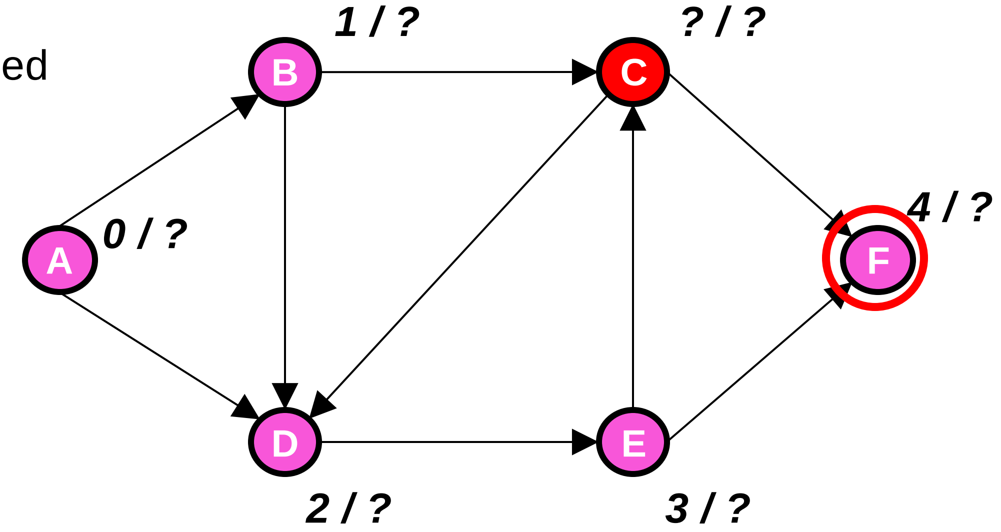
🟣 started, not finished

🔵 finished

Start at A ➡

start / finish

⚪



DFS Example

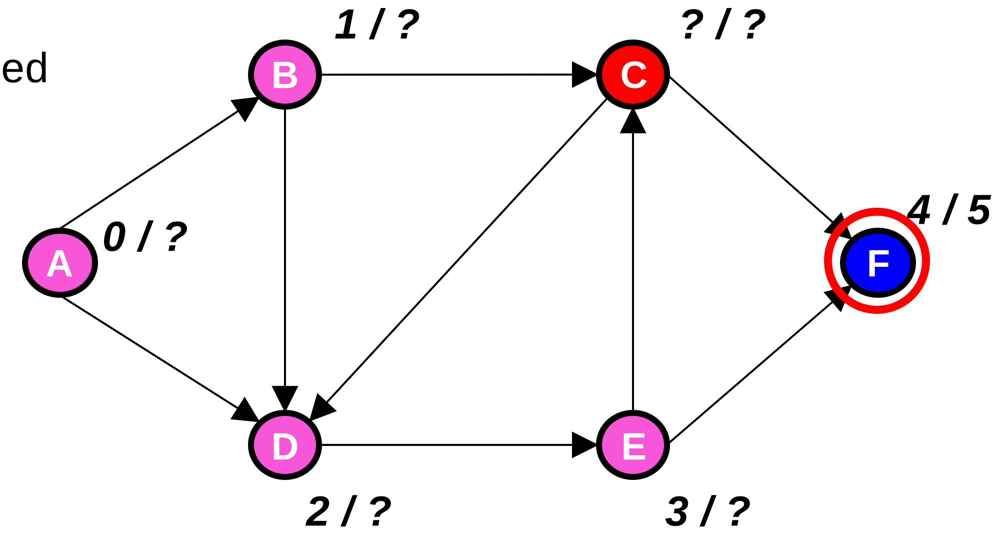
Time = 6

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



start / finish

⚪

DFS Example

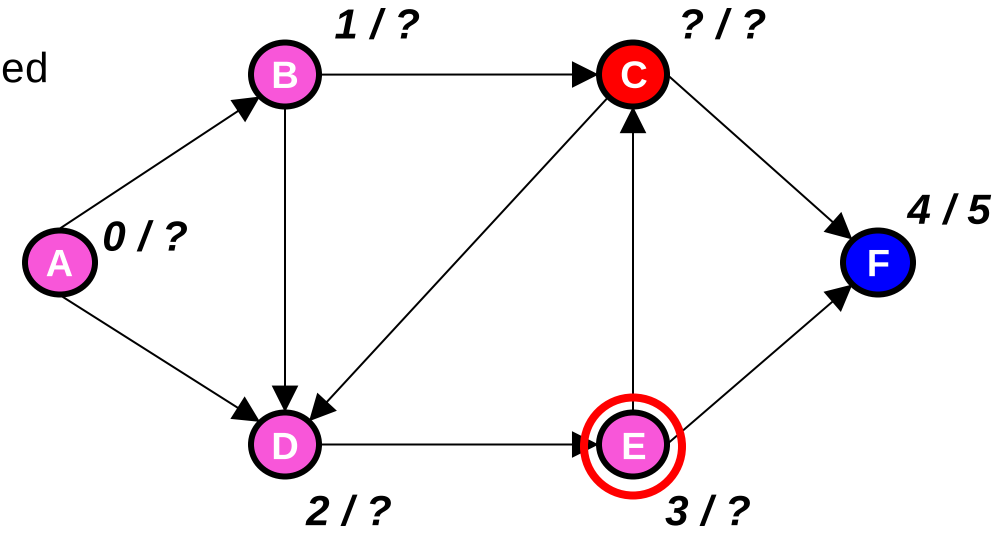
Time = 6

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



start / finish

○

DFS Example

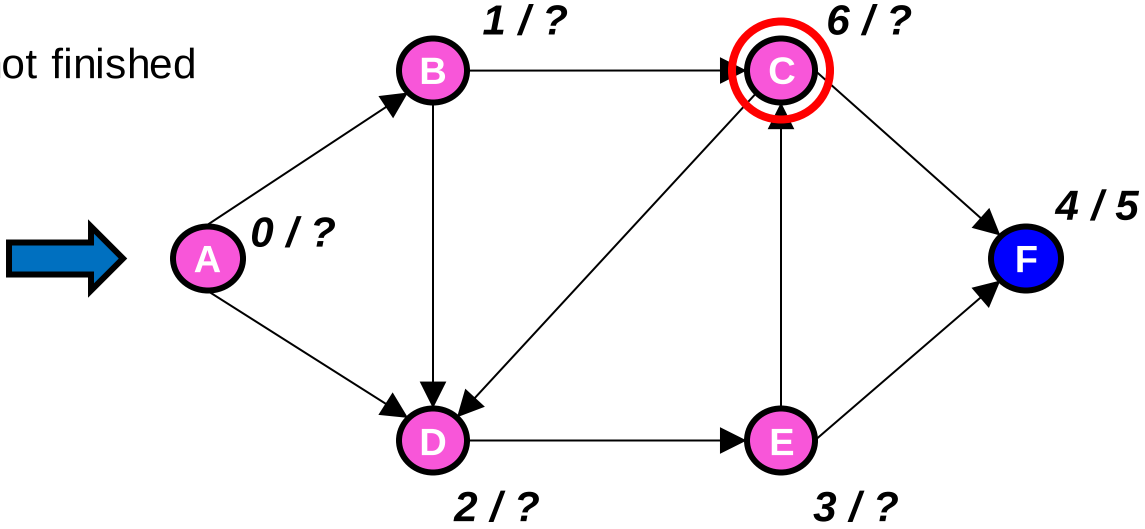
Time = 7

🔴 not started

🟣 started, not finished

🔵 finished

Start at A



⚪ *start / finish*

DFS Example

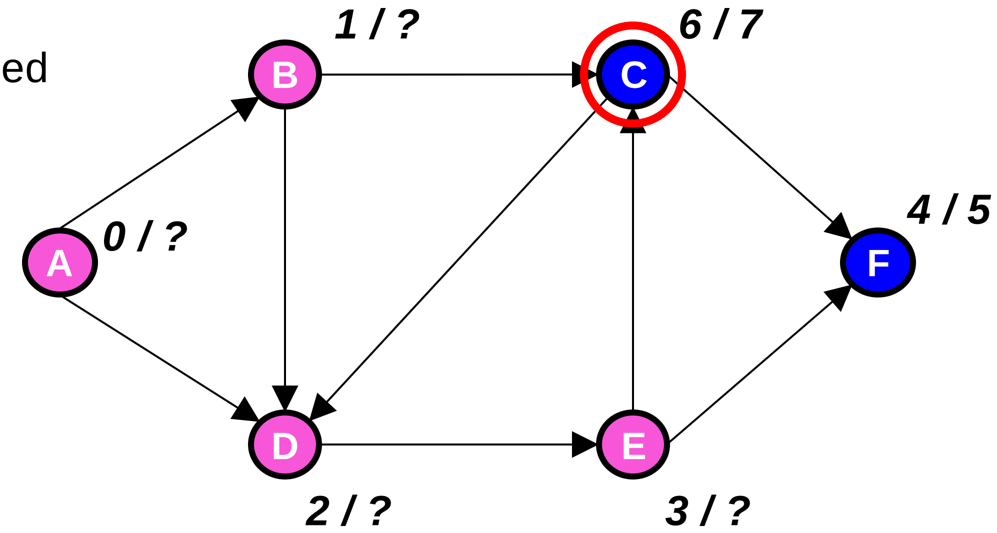
Time = 8

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



start / finish ⚪

DFS Example

Time = 9

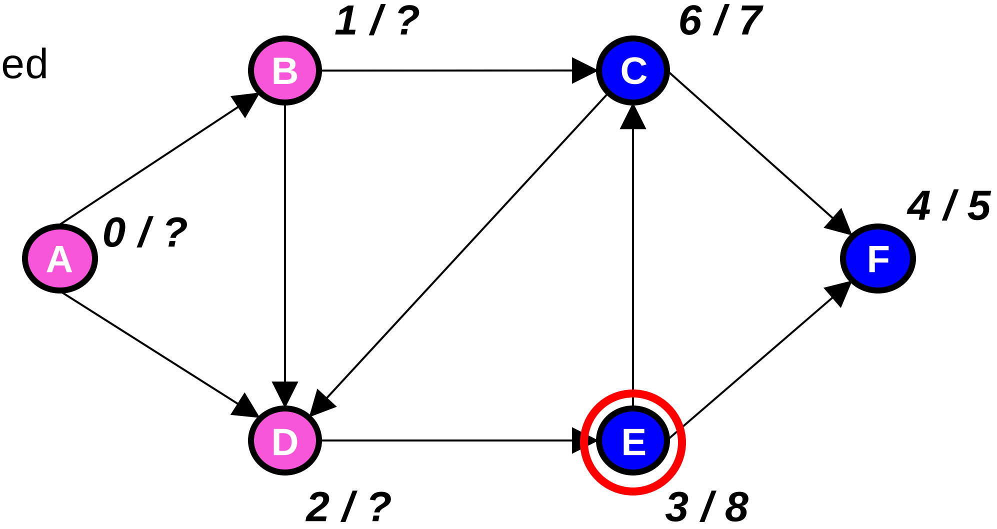
🔴 not started

● started, not finished

🔵 finished

Start at A ➡

⚪ *start / finish*



DFS Example

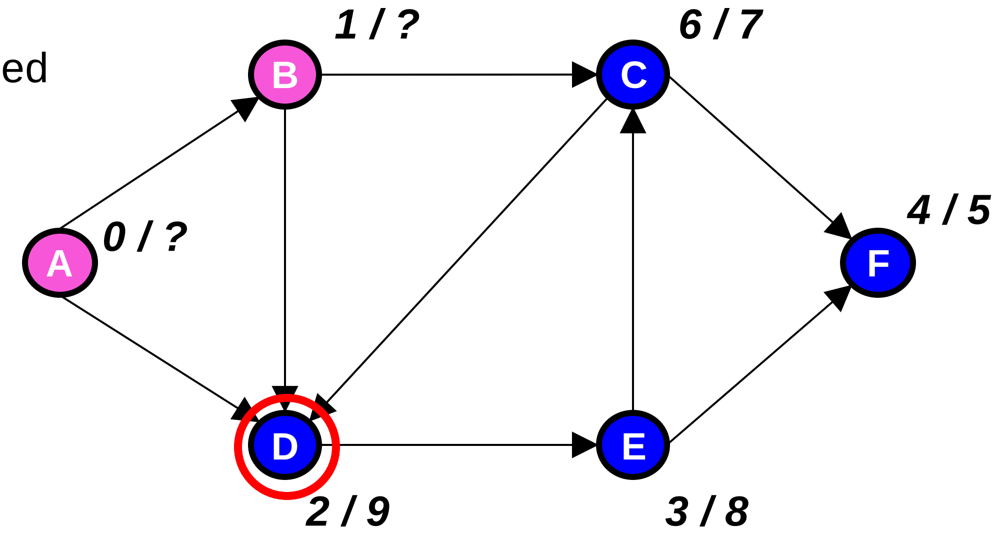
Time = 10

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



⚪ *start / finish*

DFS Example

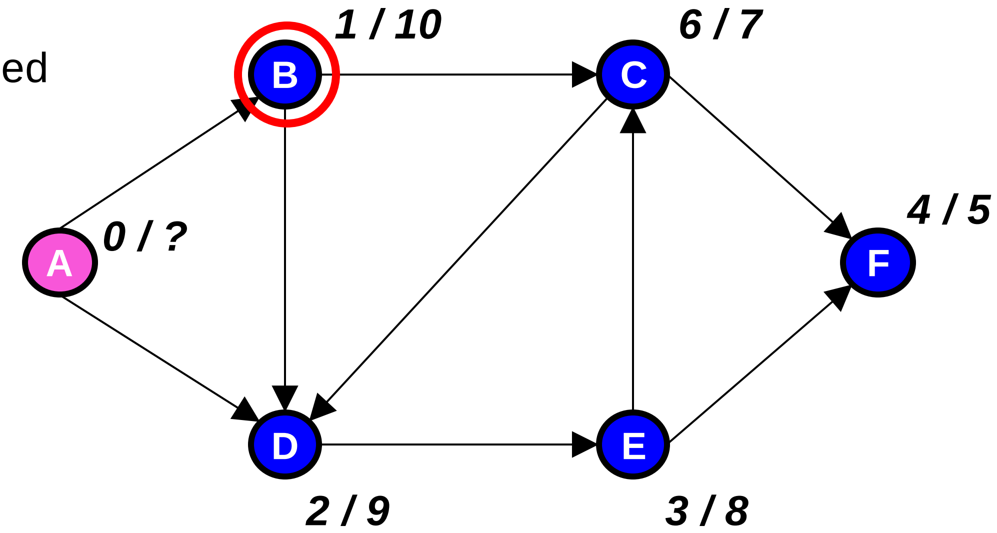
Time = 11

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



⚪ *start / finish*

DFS Example

Time = 12

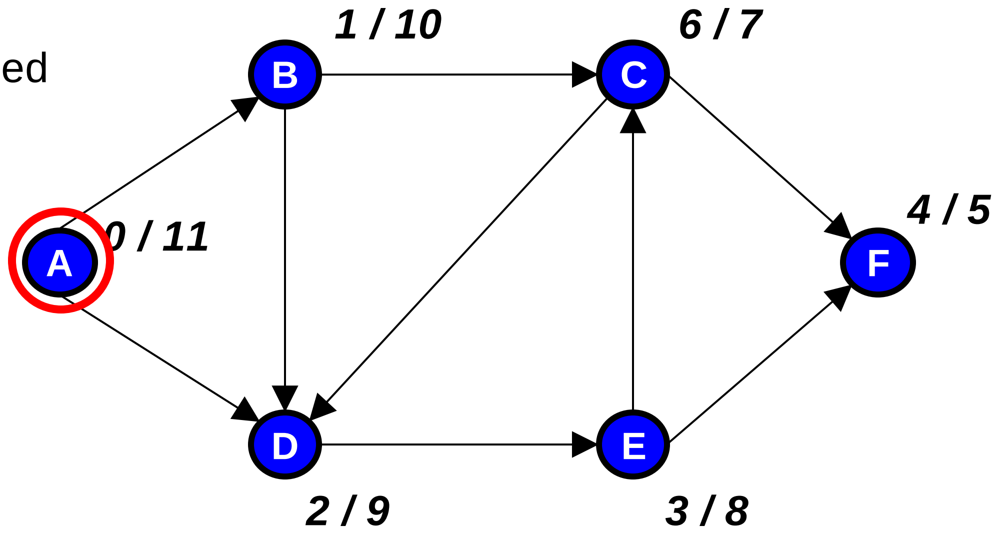
● not started

● started, not finished

● finished

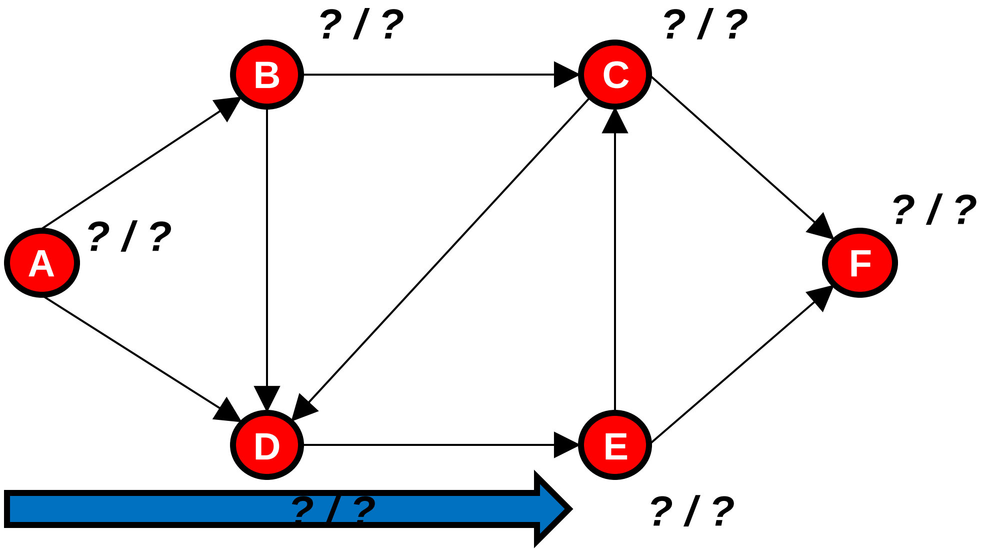
start / finish
●

Start at A ➡



Question: What If We Didn’t Start At Vertex A?

- If we finish DFSVisit of starting vertex without labeling entire graph…
- Continue by calling DFSVisit again on any unlabeled vertex.

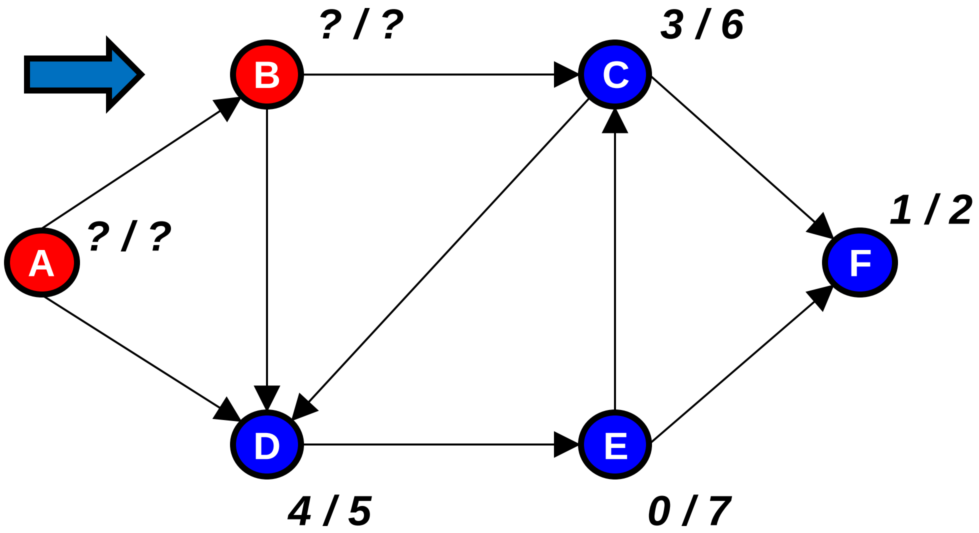


Start at E

Question: What If We Didn't Start At Vertex A?

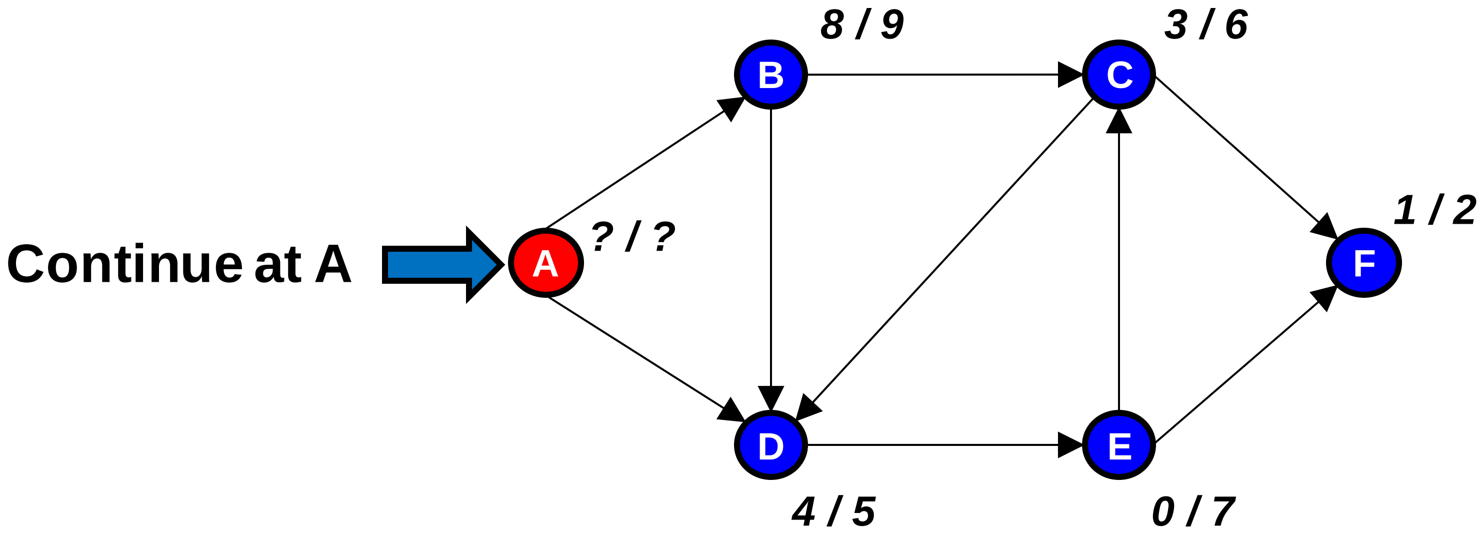
- If we finish DFSVisit of starting vertex without labeling entire graph…
- Continue by calling DFSVisit again on any unlabeled vertex.

Continue at B



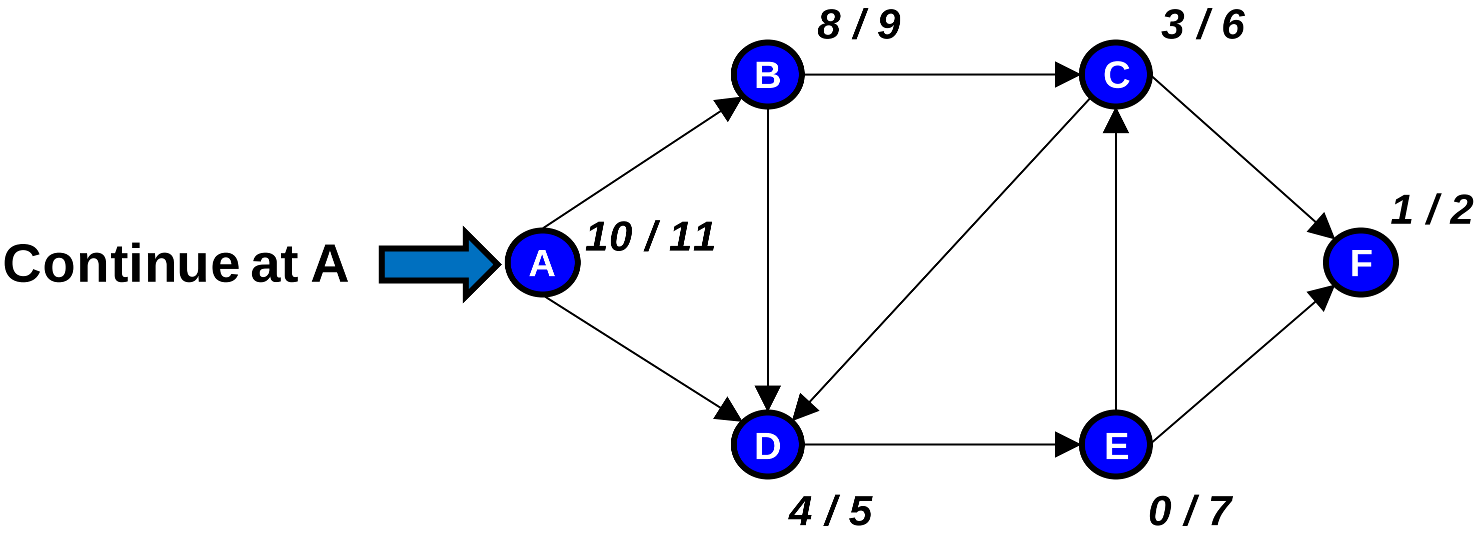
Question: What If We Didn’t Start At Vertex A?

- If we finish DFSVisit of starting vertex without labeling entire graph…
- Continue by calling DFSVisit again on any unlabeled vertex.



Question: What If We Didn't Start At Vertex A?

- If we finish DFSVisit of starting vertex without labeling entire graph…
- Continue by calling DFSVisit again on any unlabeled vertex.



Cost of DFS

- Similar analysis to BFS
- Every vertex must be discovered and marked in time O(1) apiece
- For each vertex, we check all edges that touch it.

- Hence, total cost is still **Θ(|V| + |E|)**

- *(assuming adjacency list)*

What Good is DFS?

- Search ordering can be used to infer properties of graph.

- **Example**: a graph G contains a cycle iff DFSVisit(v) ever finds an edge (v,u) for which u has been started but not finished.

Cycle Finding Example

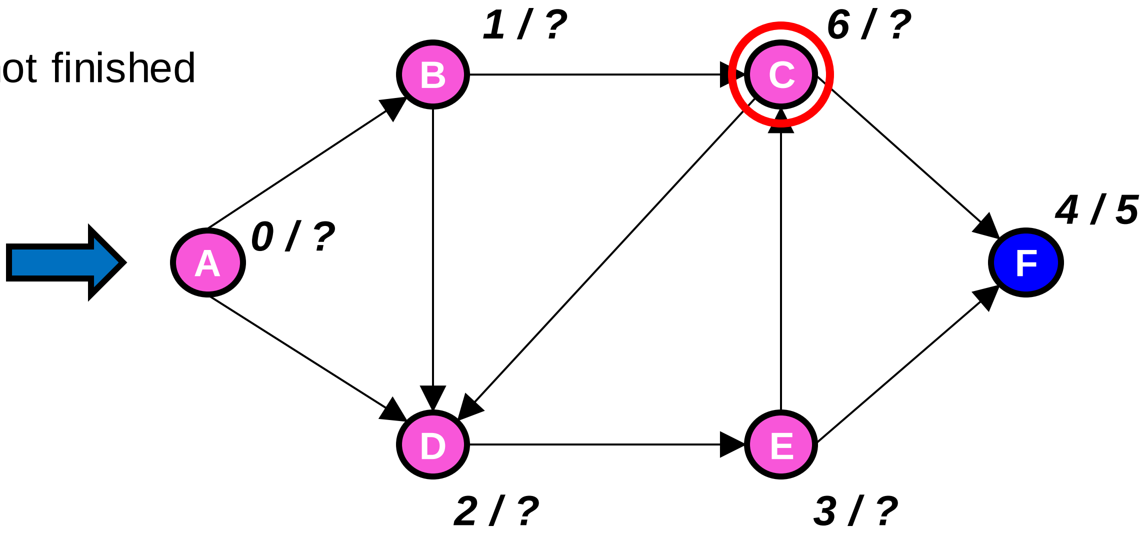
Time = 7

● not started

● started, not finished

● finished

Start at A



start / finish

●

Cycle Finding Example

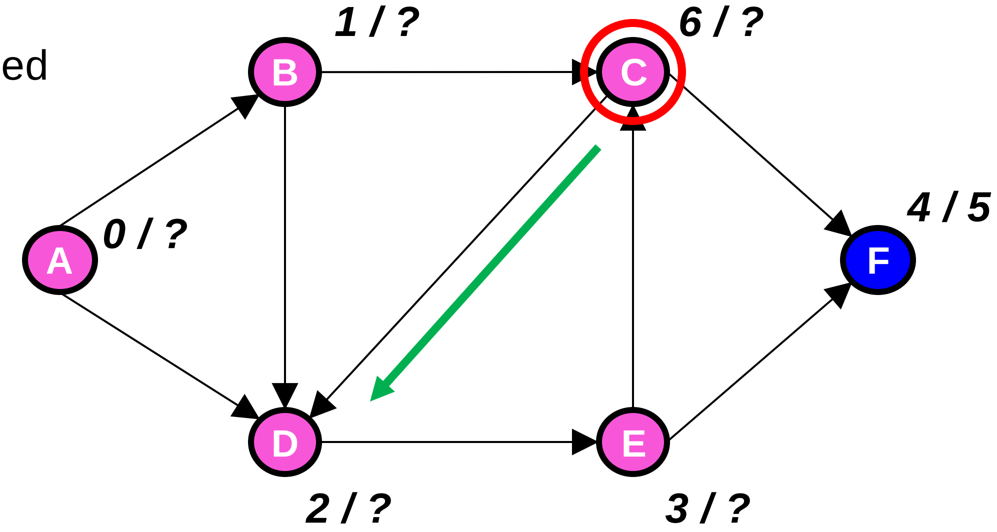
Time = 7

● not started

● started, not finished

● finished

Start at A ➡



DFSVisit(C) explores edge (C,D)

start / finish

○

Cycle Finding Example

Time = 7

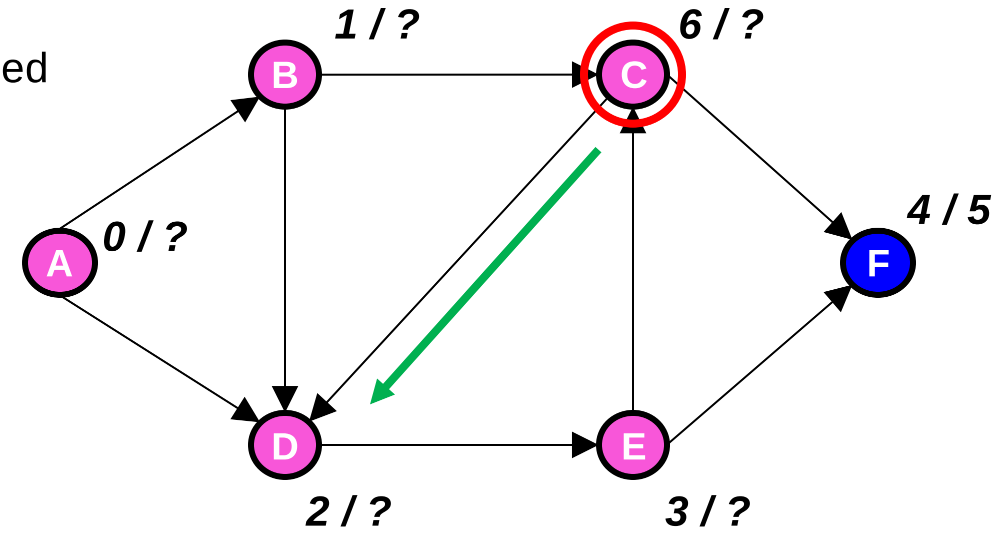
🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡

⚪ *start / finish*



D is started, not finished

Cycle Finding Example

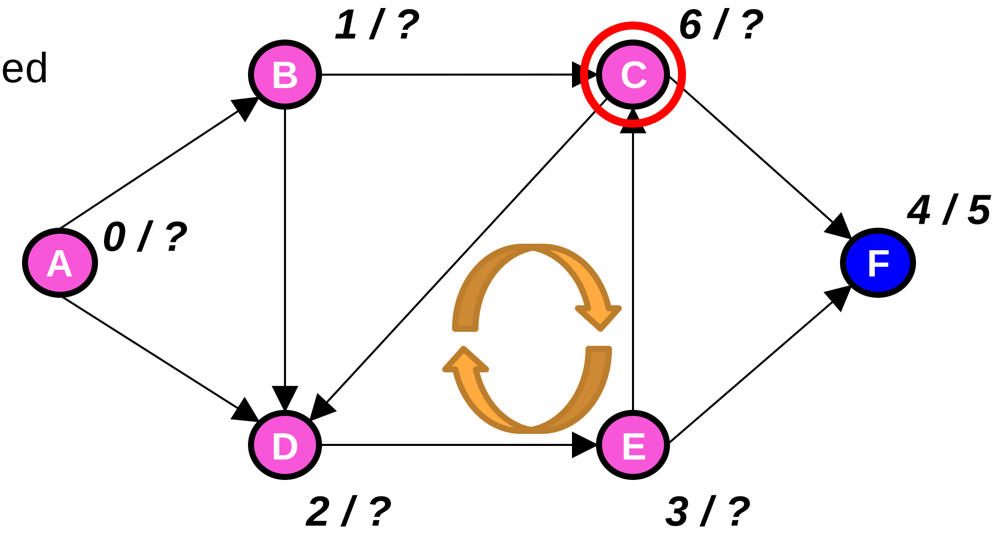
Time = 7

🔴 not started

🟣 started, not finished

🔵 finished

Start at A ➡



Cycle DEC

start / finish ○

Why Cycle Finding Works (1/2)

- Claim: G contains a cycle iff DFS finds a vertex that is started, not finished.

- If DFSVisit(u) finds adjacent vertex w that is started, not finished…

- DFSVisit(w) was called earlier and is not yet done.
- Hence, DFS found a path from w to u.
- But edge (u,w) also exists, hence a cycle.

Why Cycle Finding Works (2/2)

- Claim: G contains a cycle iff DFS finds a vertex that is started, not finished.

- If G contains a cycle, let w be ***first*** vertex of cycle found by DFS, and suppose cycle includes edge (u,w).

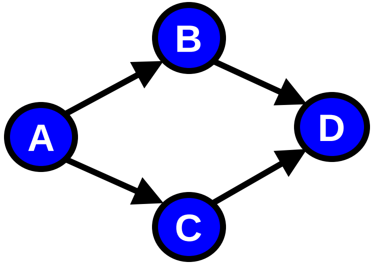
- DFSVisit(w) does not return until it finds **every** vertex reachable from w.
- That includes u, so DFSVisit(u) finds unfinished vertex w.

What Else Can We Do With DFS?

- If a ***directed*** graph does not contain a cycle, we can assign an order to its vertices.

- Defn: if u ≠ v, u < v if there exists a path in G from u to v.

- This rule yields a *partial* order on G.



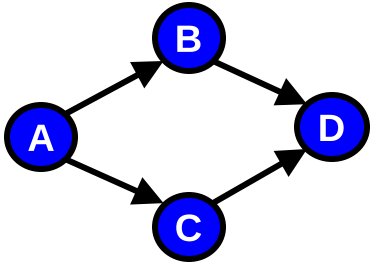
A < B
A < C
B < D
C < D

What Else Can We Do With DFS?

- If a ***directed*** graph does not contain a cycle, we can assign an order to its vertices.

- Defn: if u ≠ v, u < v if there exists a path in G from u to v.

- This rule yields a *partial* order on G.



A < B

A < C

B < D

C < D

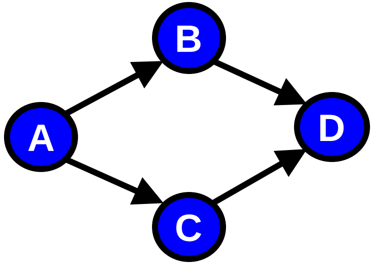
A < D

What Else Can We Do With DFS?

- If a ***directed*** graph does not contain a cycle, we can assign an order to its vertices.

- Defn: if u ≠ v, u < v if there exists a path in G from u to v.

- This rule yields a *partial* order on G.



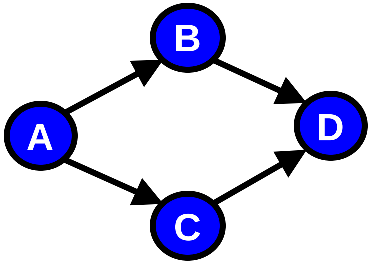
A < B
A < C
B < D
C < D

A < D

B, C *incomparable*

Topological Order

- A topological order on a directed, acyclic graph (DAG) is any total ordering of the vertices consistent with the partial order defined by the edges.



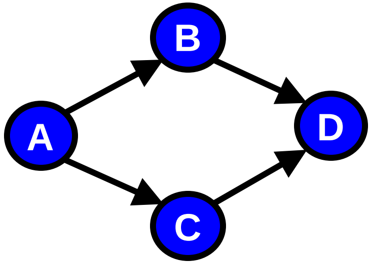
A < B
A < C
B < D
C < D

A < D

{ A, B, C, D }

Topological Order

- A topological order on a directed, acyclic graph (DAG) is any total ordering of the vertices consistent with the partial order defined by the edges.



A < B
A < C
B < D
C < D

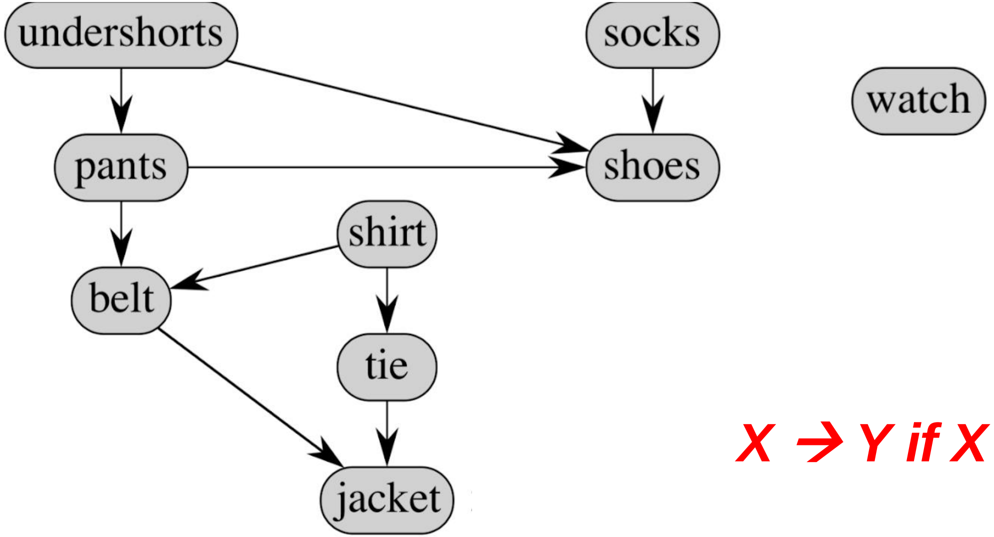
A < D

{ A, B, C, D }

{ A, C, B, D }

A DAG may have more than one topological order.

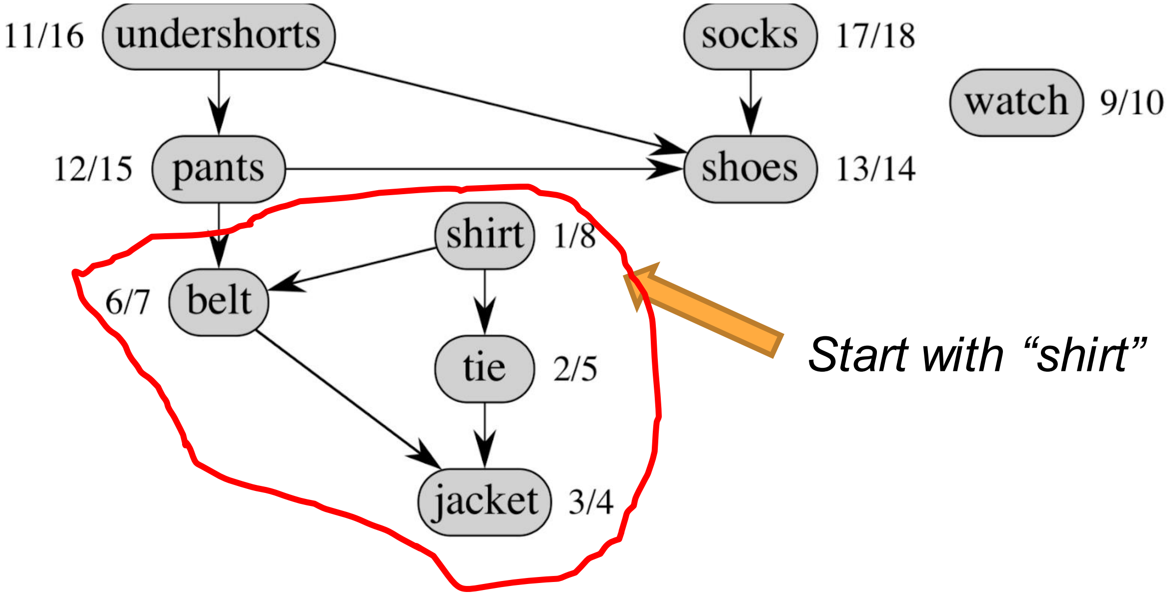
Example (from Book) – Getting Dressed



X → Y if X must be done before Y

Example (from Book) – Getting Dressed

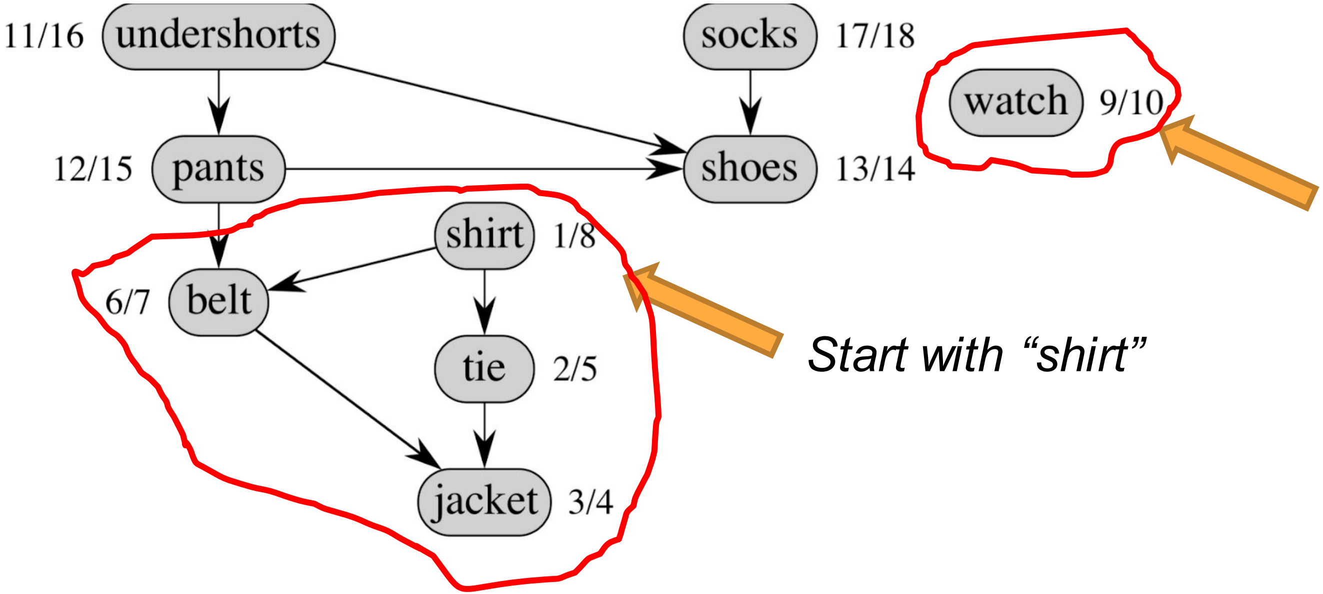
(time in book starts at 1, not 0)



Start with "shirt"

Example (from Book) – Getting Dressed

(time in book starts at 1, not 0)

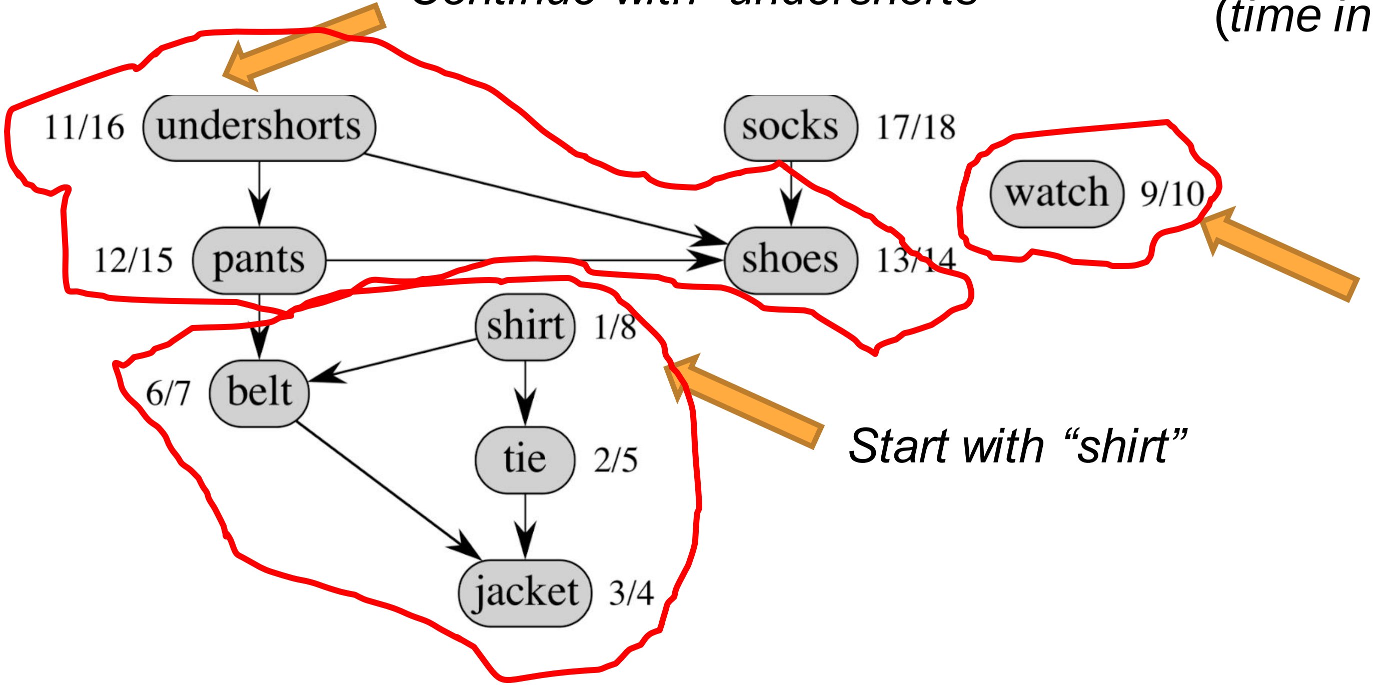


Continue with "watch"

Start with "shirt"

Example (from Book) – Getting Dressed

(time in book starts at 1, not 0)



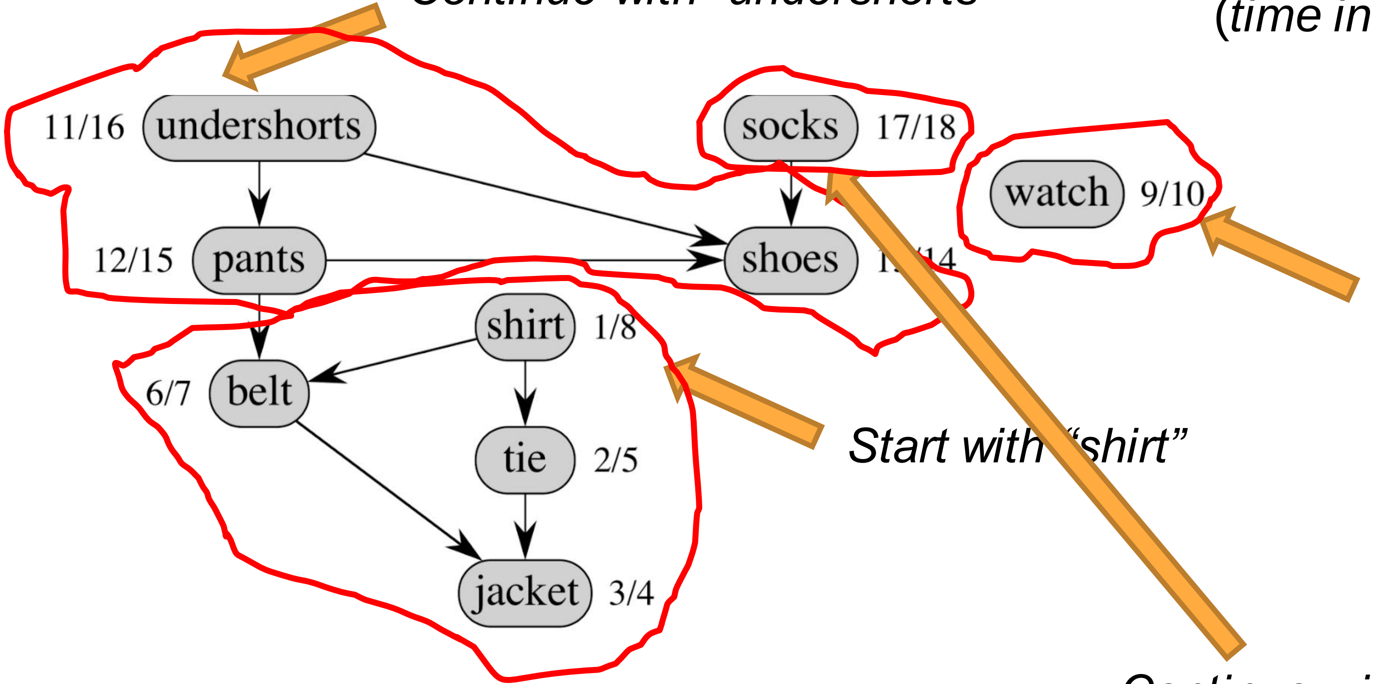
Continue with "undershorts"

Start with "shirt"

Continue with "watch"

Example (from Book) – Getting Dressed

(time in book starts at 1, not 0)



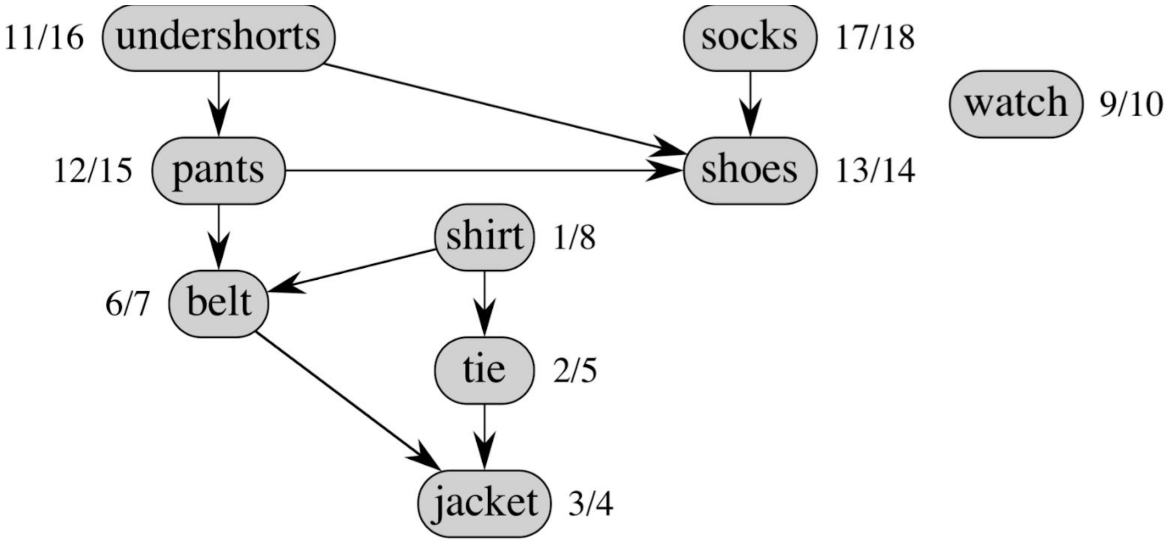
Continue with “undershorts”

Continue with “watch”

Start with “shirt”

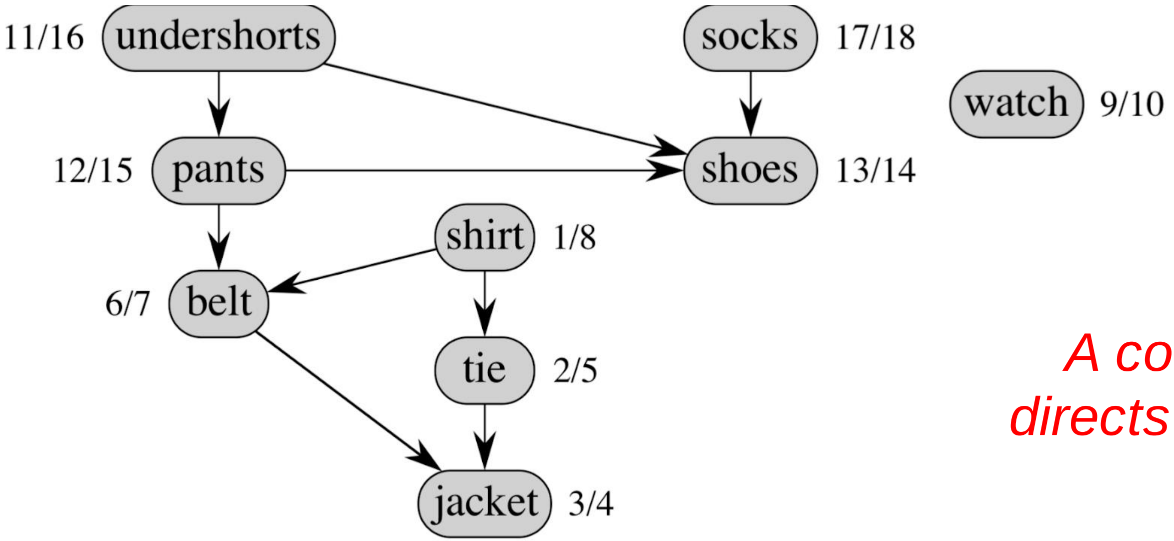
Continue with “socks”

Example (from Book) – Getting Dressed

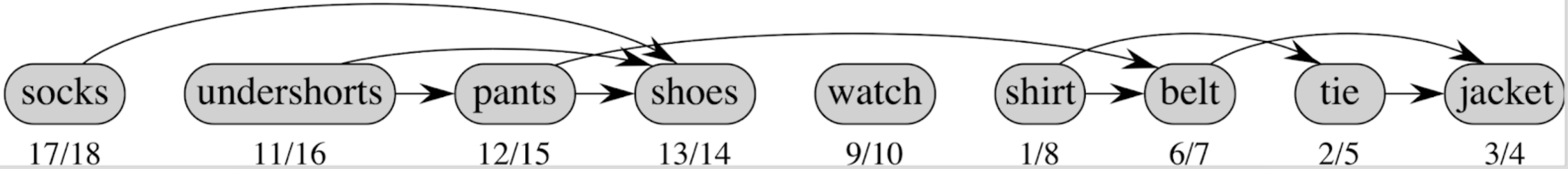


Give one possible topological ordering of this graph.

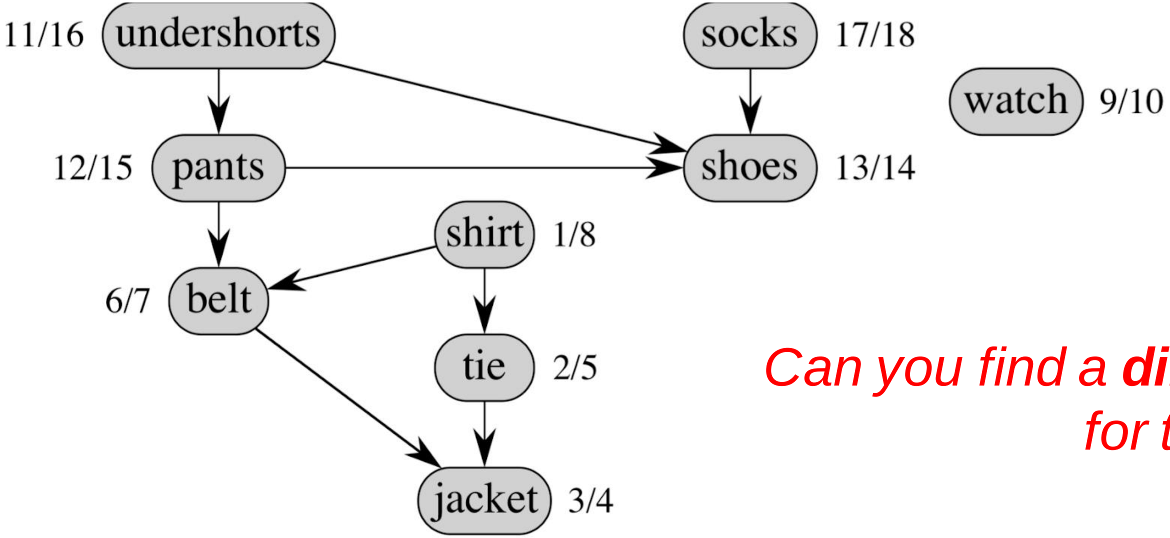
Example (from Book) – Getting Dressed



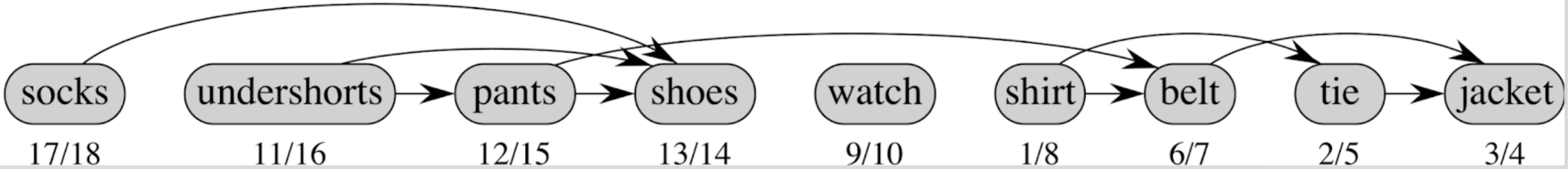
A consistent total order directs all edges left → right



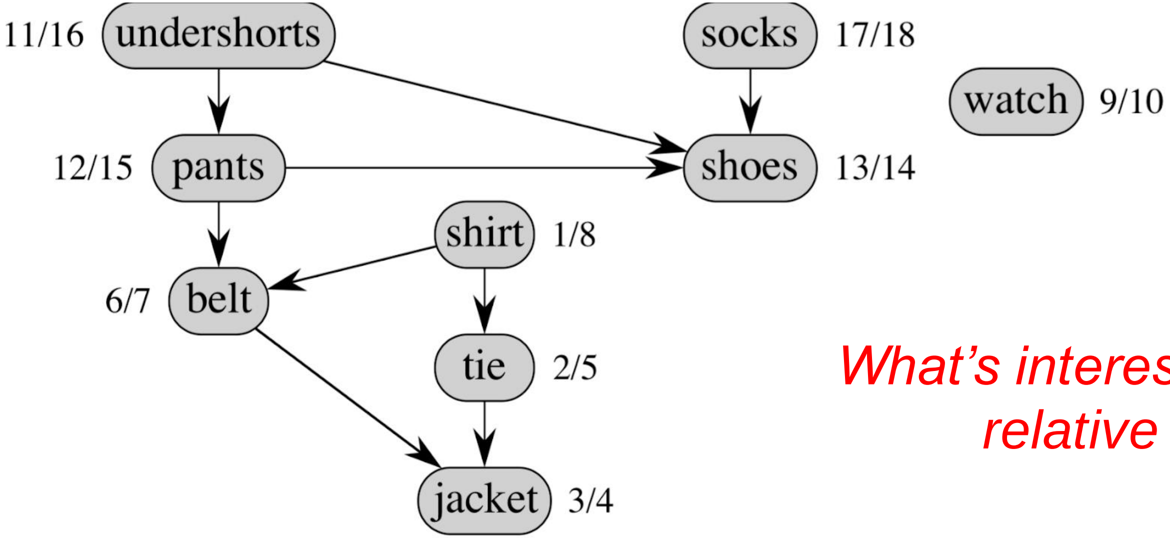
Example (from Book) – Getting Dressed



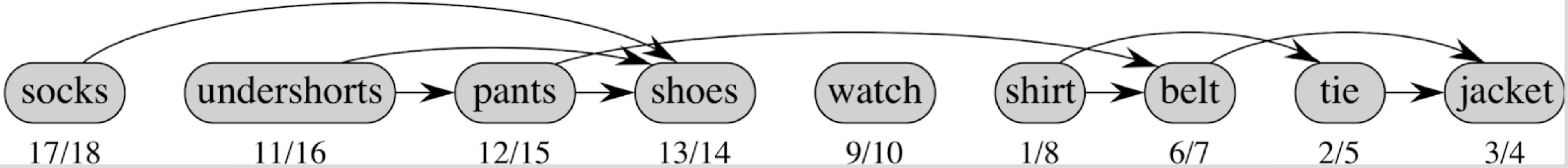
Can you find a ***different*** *topological order for this graph?*



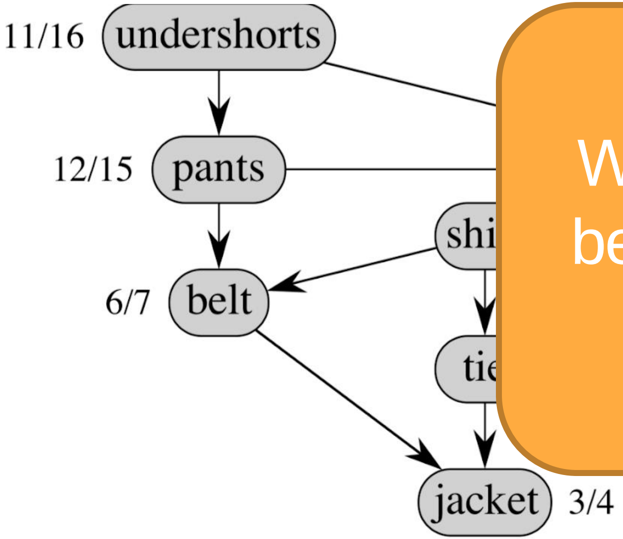
Example (from Book) – Getting Dressed



*What's interesting about **this** order relative to DFS times?*



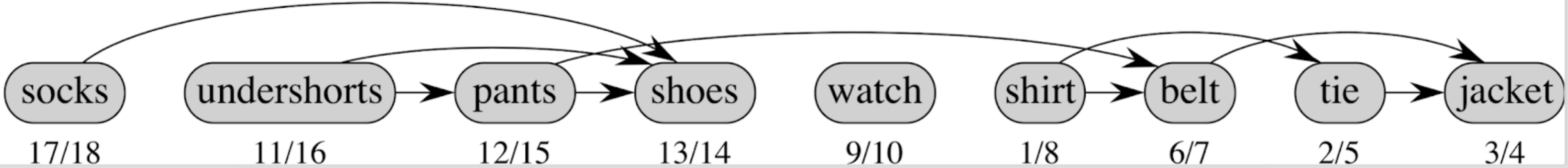
Example (from Book) – Getting Dressed



We'll explore the relationship between DFS and topological order in Studio 13.

is order

s?



Some Uses of BFS and DFS

BFS

- Shortest distances
- Bipartite detection
- Bipartite matching
- State-space search in AI

DFS

- Cycle detection
- Dependency resolution
- Reachability (e.g. strongly connected components)
- Compiler analyses