LECTURE 12: SEARCHING ON GRAPHS

Harold Soh
harold@comp.nus.edu.sg

ADMINISTRATIVE ISSUES

Please make sure you **enroll for the course on Kattis** and **join the problem set session**

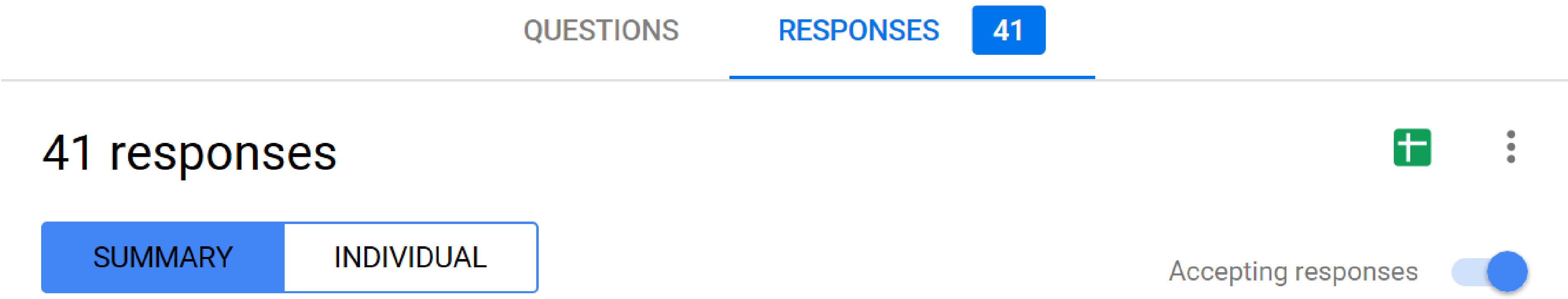
- If you don't enroll/jin, we cannot see your submission.
- No submission → Zero points!

If you aren't sure how, please approach your TA during DG.

ADMINISTRATIVE ISSUES

Thanks for the feedback!

Closing polls this Friday

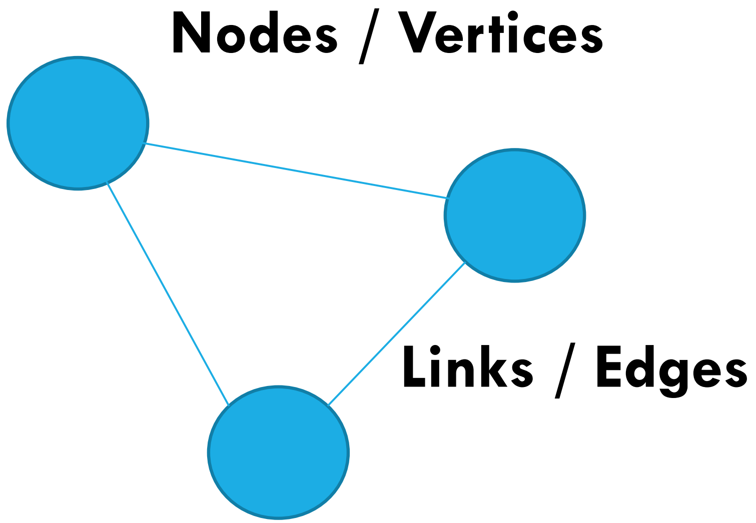


QUESTIONS?

UNDIRECTED GRAPHS: A FORMAL DEFINITION

Graph $G = \langle V, E \rangle$ ("a tuple of two sets")

- V is a set of nodes
- E is a set of edges
  - $E \subseteq \{ (v, w) : v, w \in V \}$



Simple Graph:

- $e = (v, w)$ for $v \neq w$ ("no self loops")
- $\forall e_1, e_2 \in E : e_1 \neq e_2$ ("only one edge per pair of nodes")

TERMINOLOGY SUMMARY

Graph: $G = \langle V, E \rangle$

Degree of a node: number of edges connected to it

Diameter: longest shortest path between two different nodes

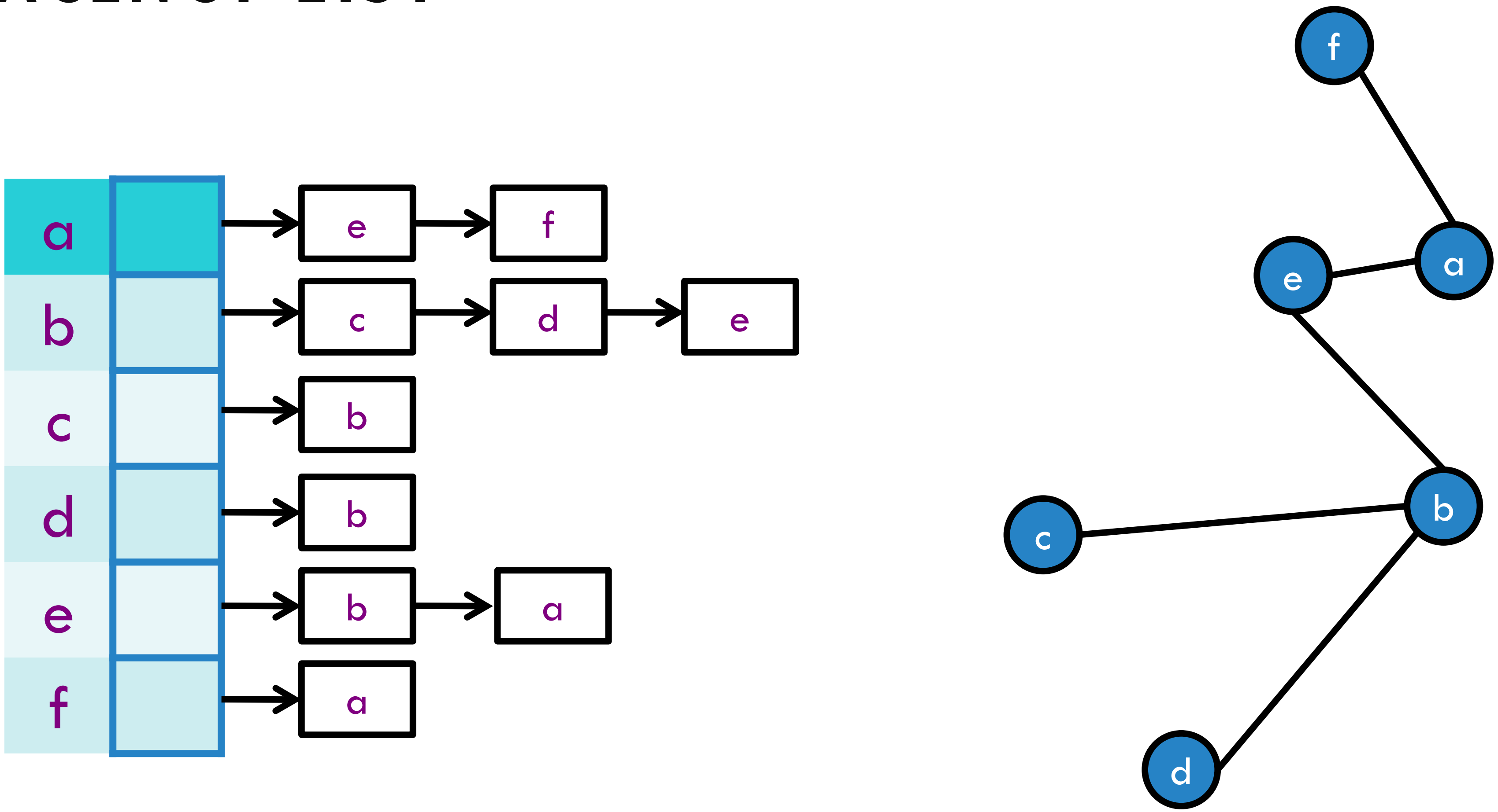
Connected Graph: path between any two nodes

Clique: fully connected graph

Line Graph: a line (duh!)

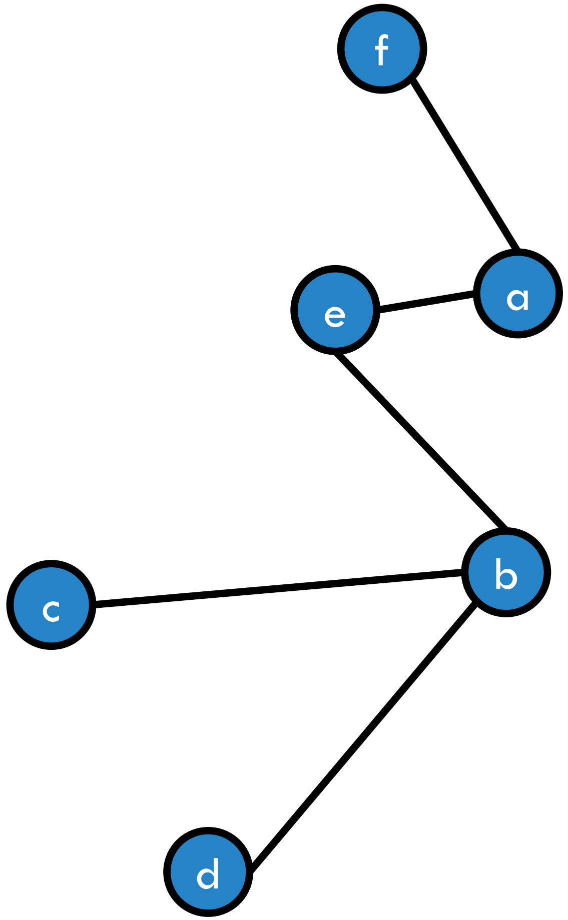
Star: central node connected to all other nodes.

ADJACENCY LIST



ADJACENCY MATRIX

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

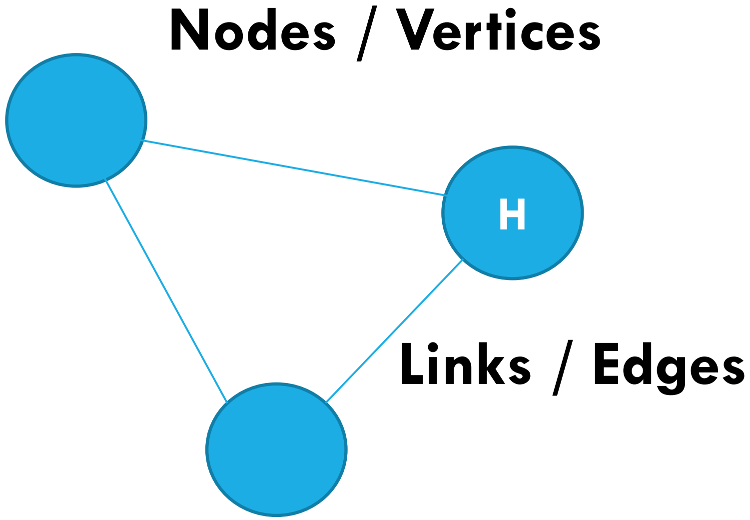


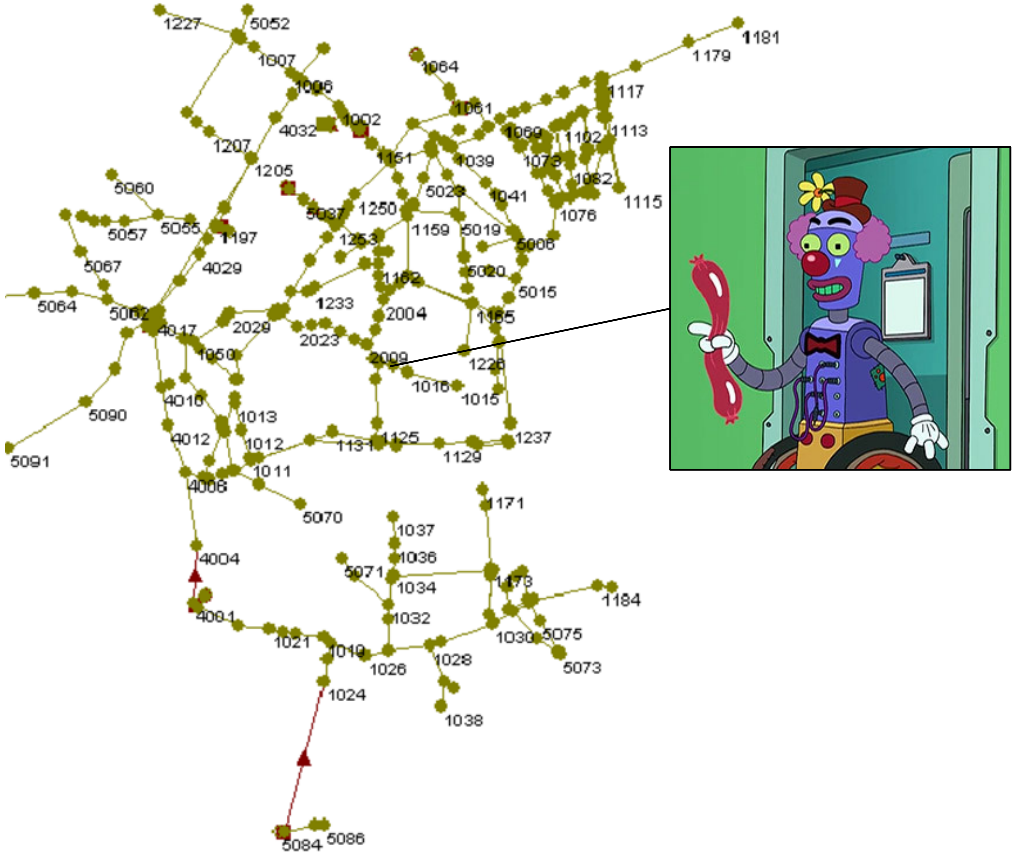
LEARNING OUTCOMES

By the end of this session, students should be able to:

- Explain the **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** Algorithms.
- State the **similarities** and **differences** between the two algorithms
- Analyze the **performance of BFS and DFS**
- Describe the **topological sort** algorithm

MODEL THE SEWER AS A GRAPH





SEARCHING A GRAPH

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

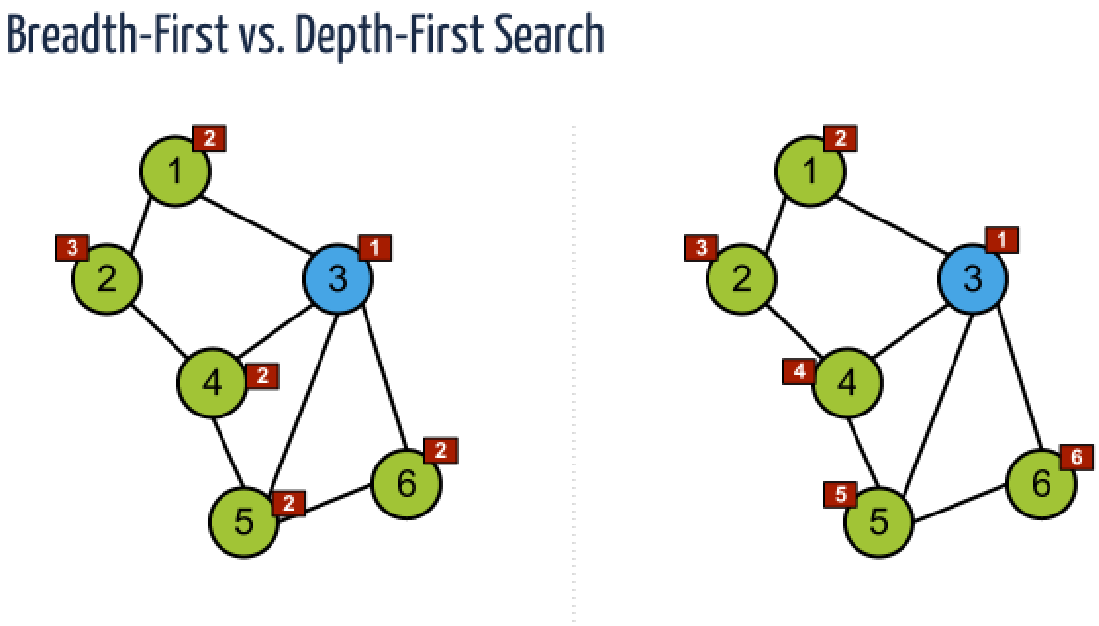
  Or: visit **all** the nodes in the graph

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list



BREADTH-FIRST SEARCH

Explore level by level

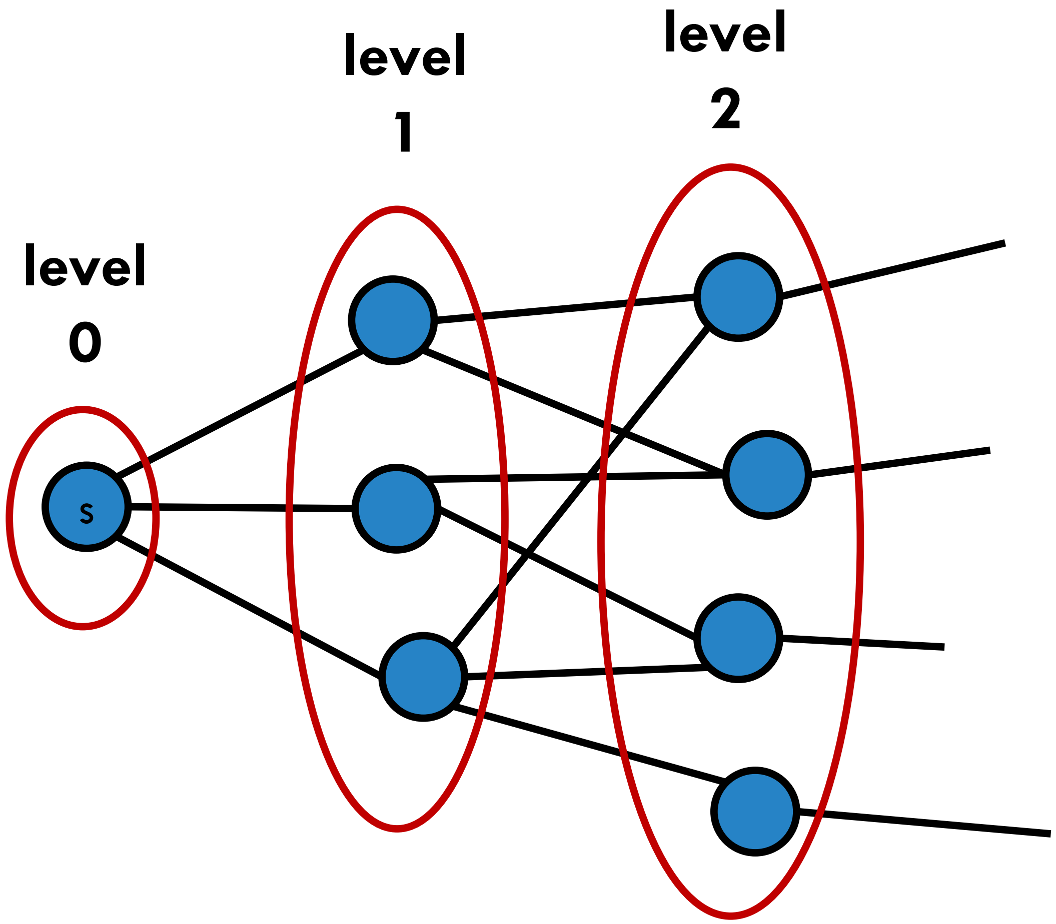
Frontier: current level

Initial frontier: {s}

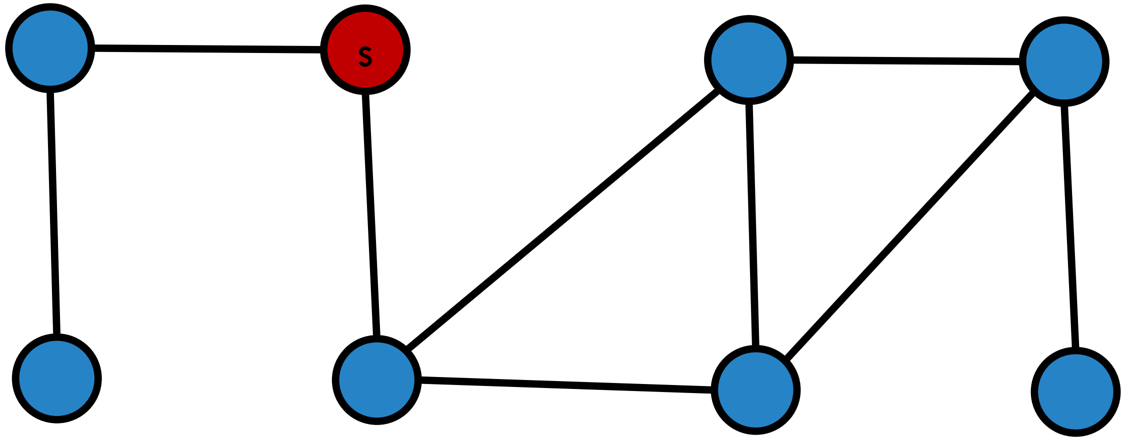
Advance frontier.

Don't go backward!

Finds <u>shortest</u> paths.



BREADTH-FIRST SEARCH EXAMPLE



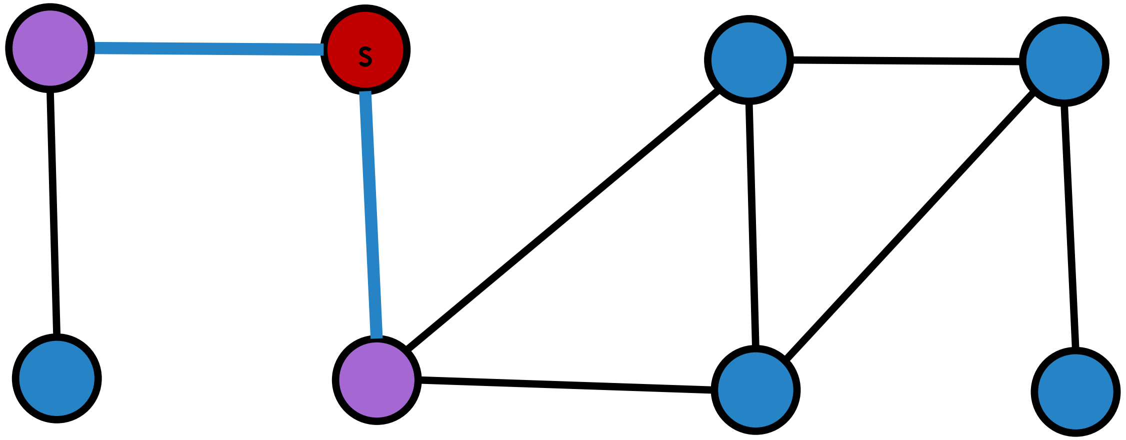
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



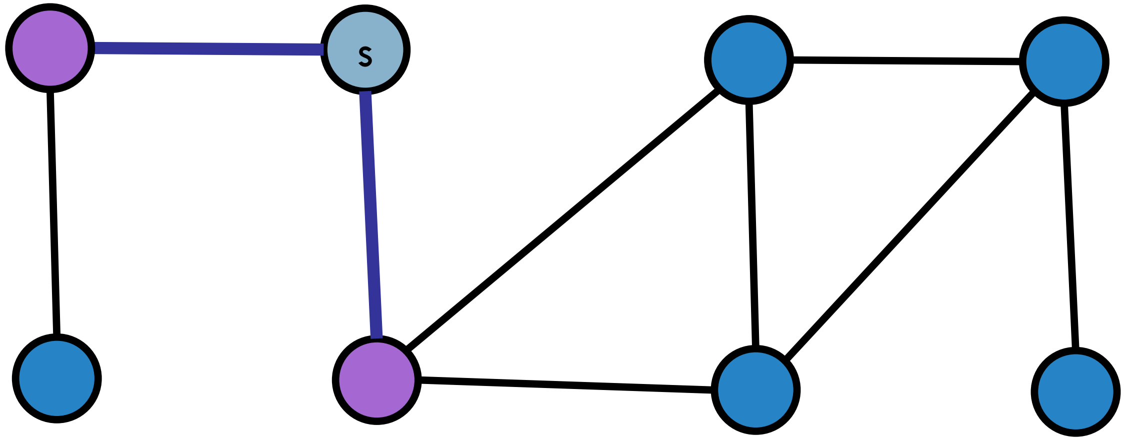
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



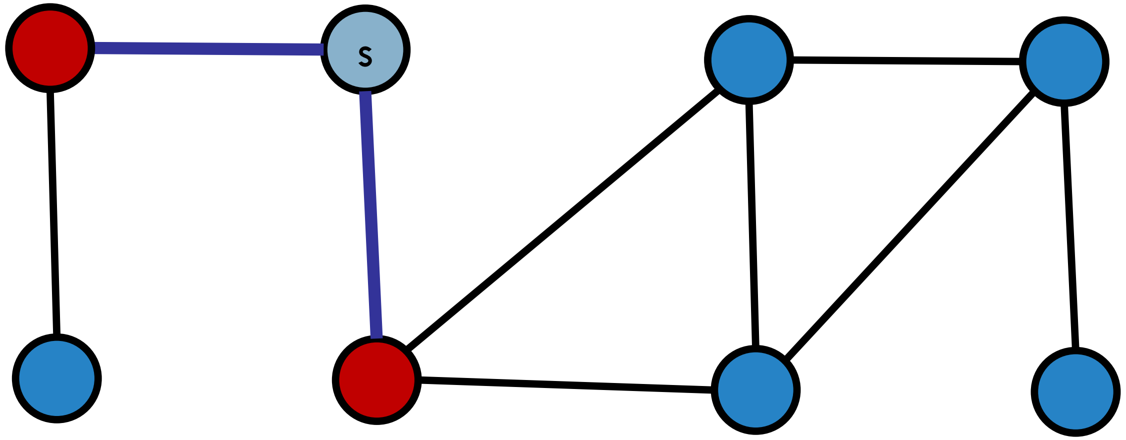
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



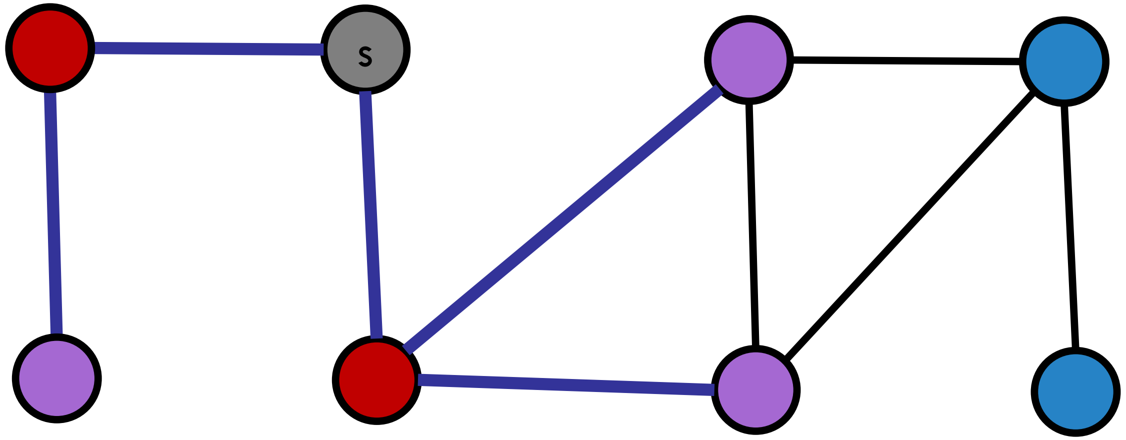
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



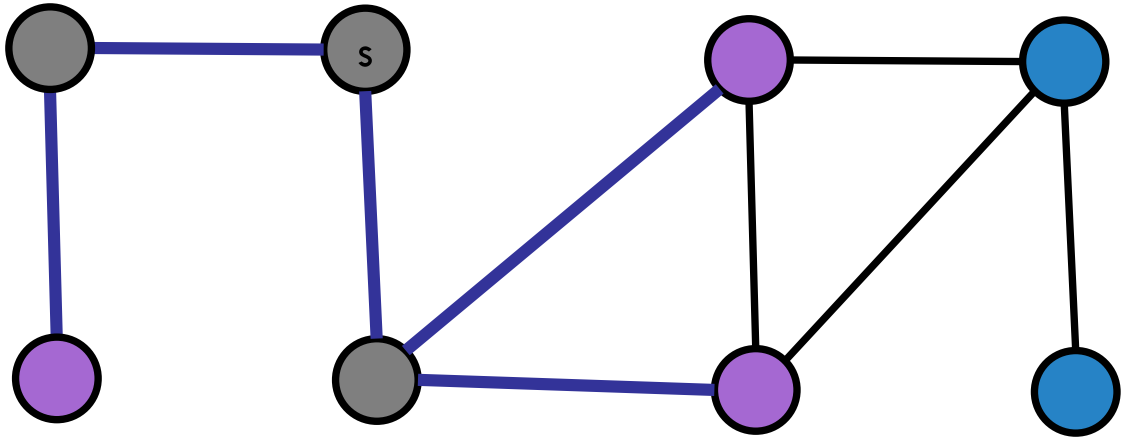
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



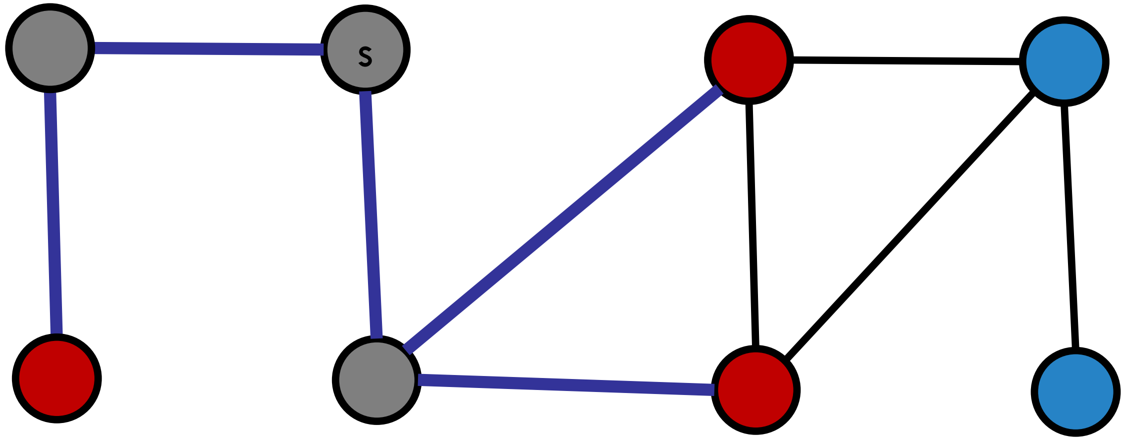
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



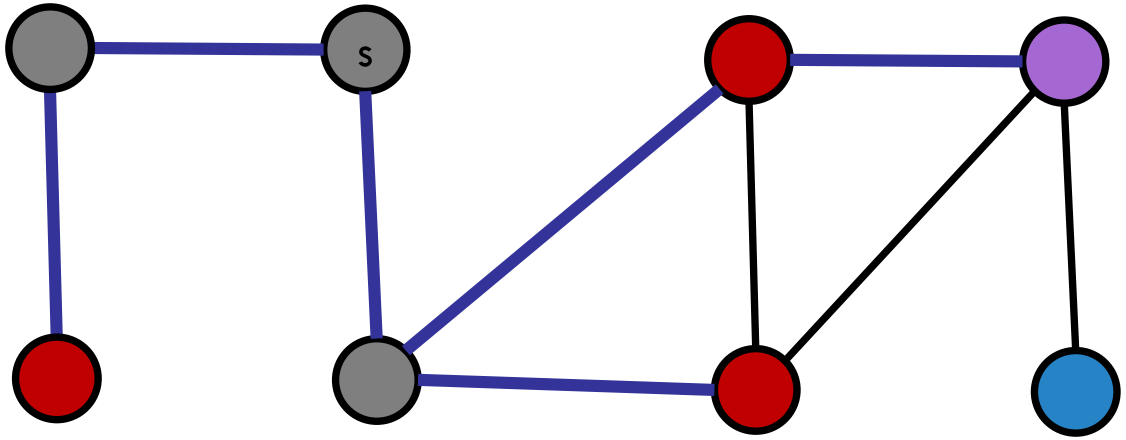
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



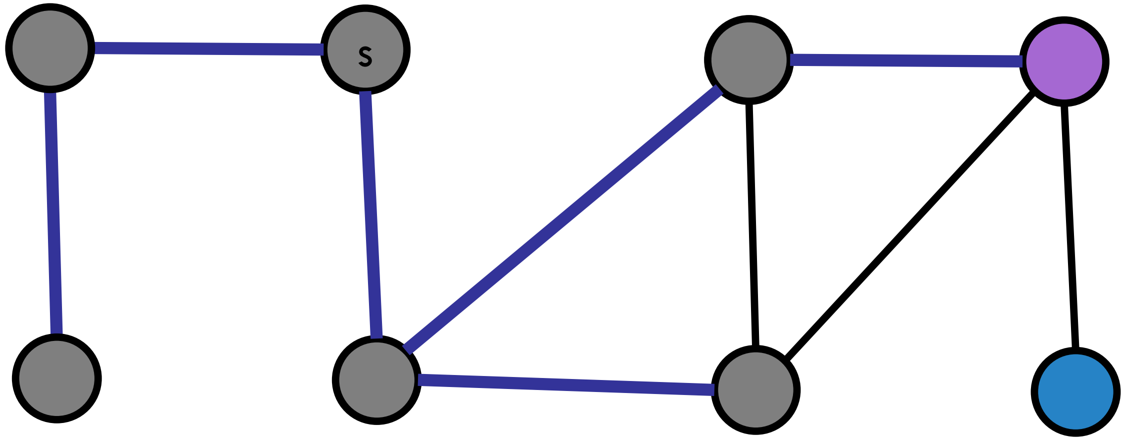
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



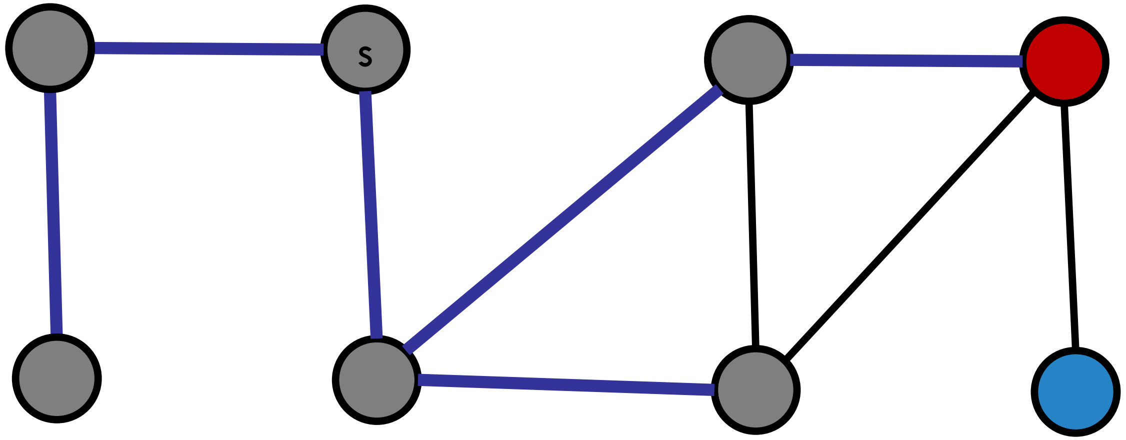
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



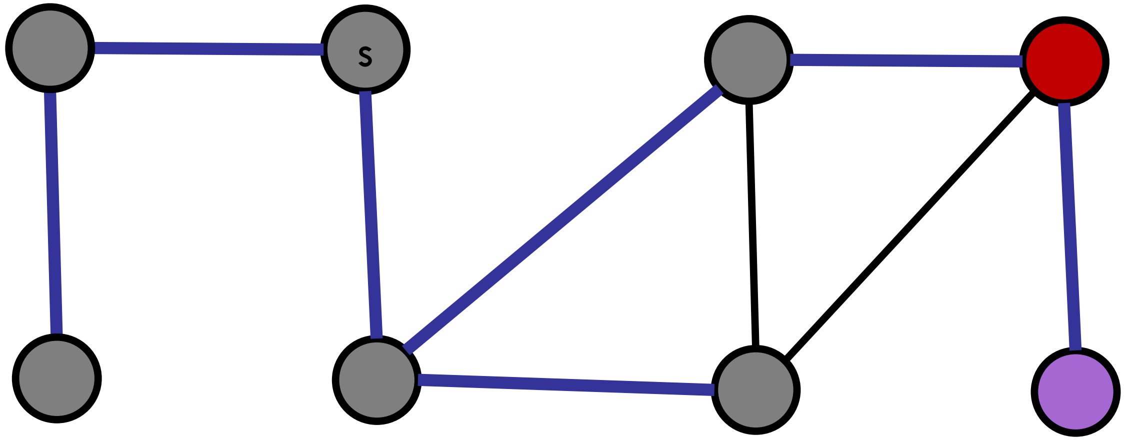
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



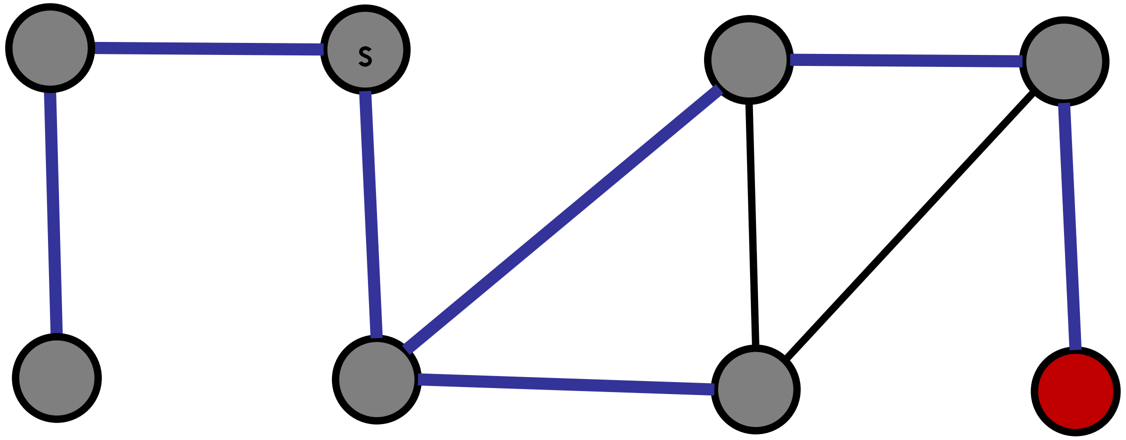
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



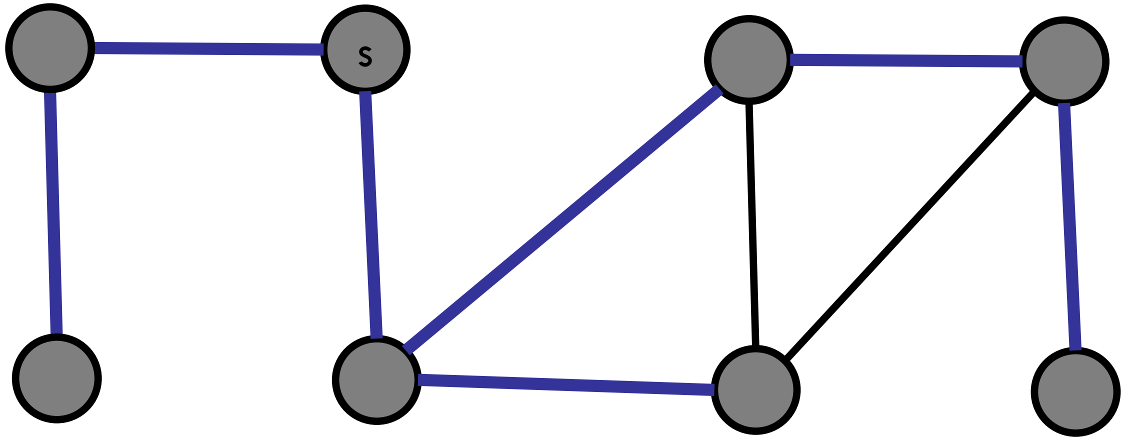
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



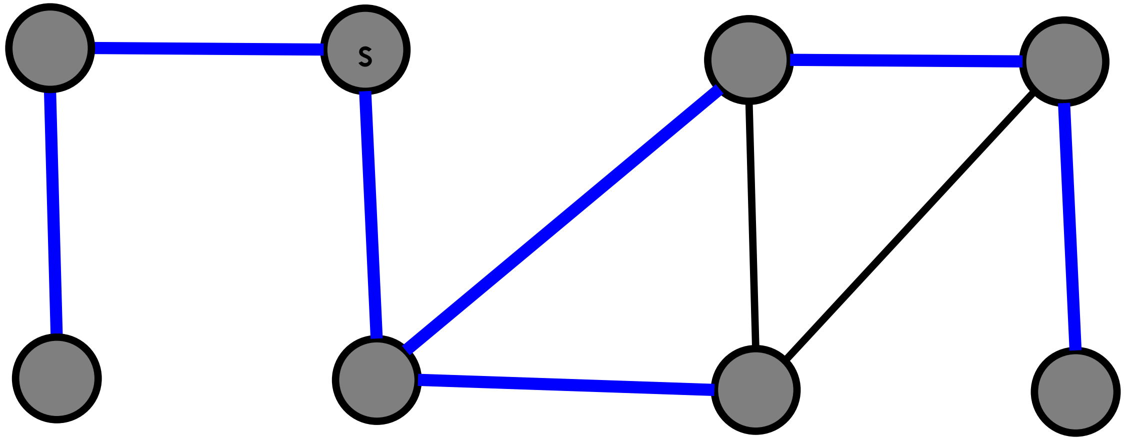
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

BREADTH-FIRST SEARCH EXAMPLE



Red = active frontier

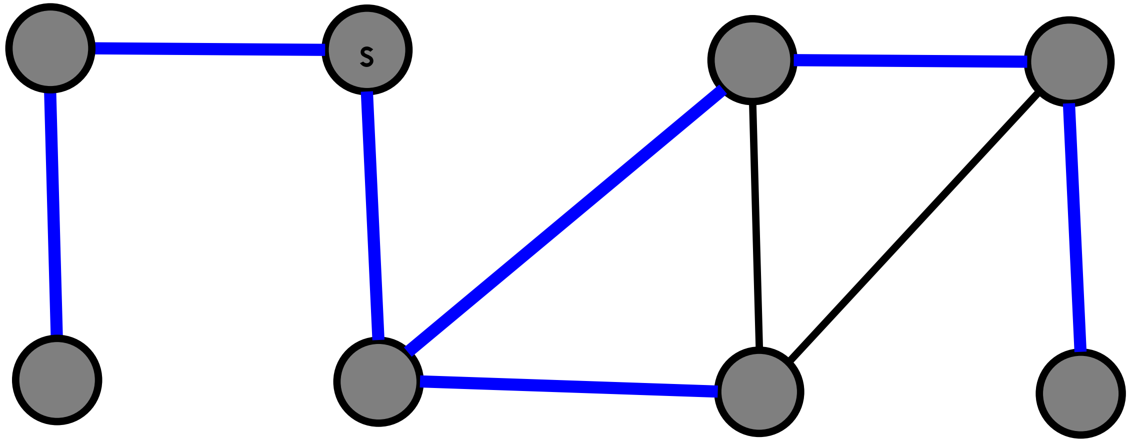
Purple = next

Gray = visited

Blue = unvisited

BFS PSEUDOCODE

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```



Red = active frontier

Purple = next

Gray = visited

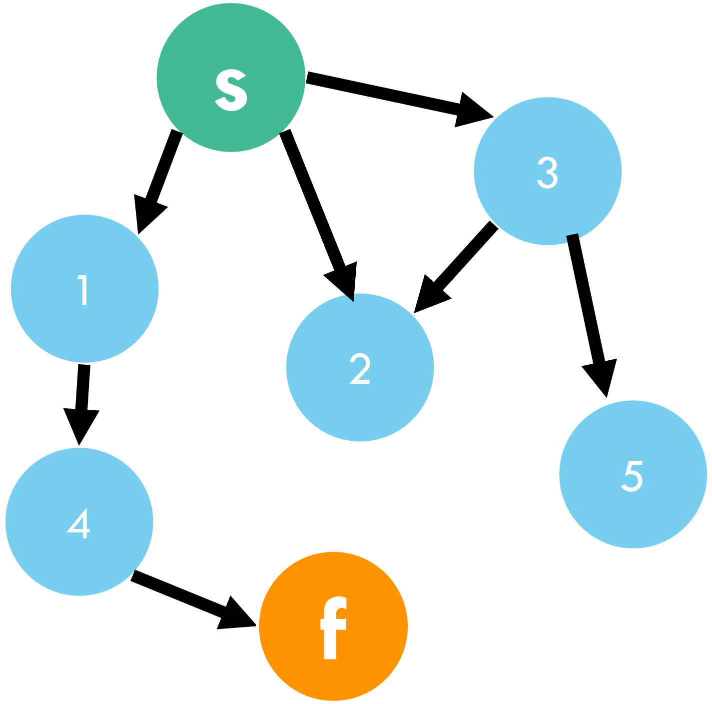
Blue = unvisited

Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

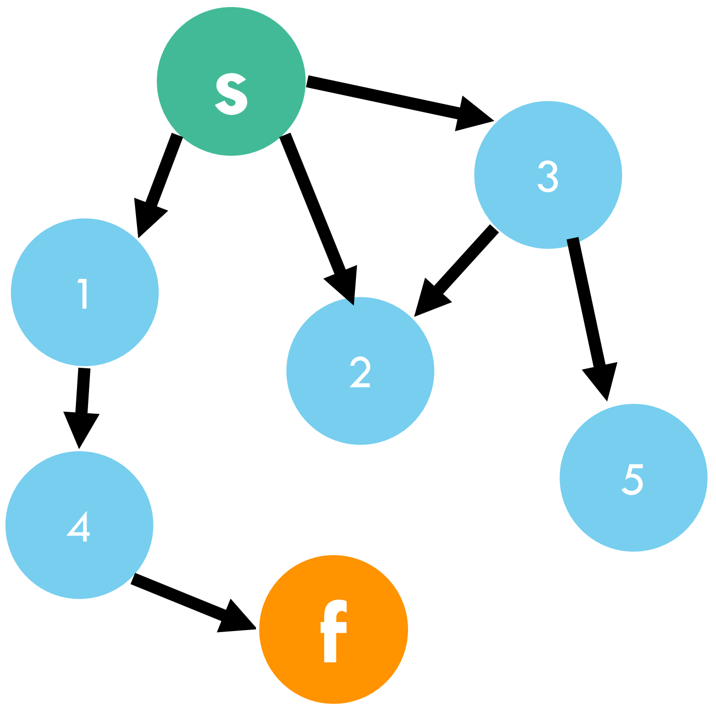


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: s

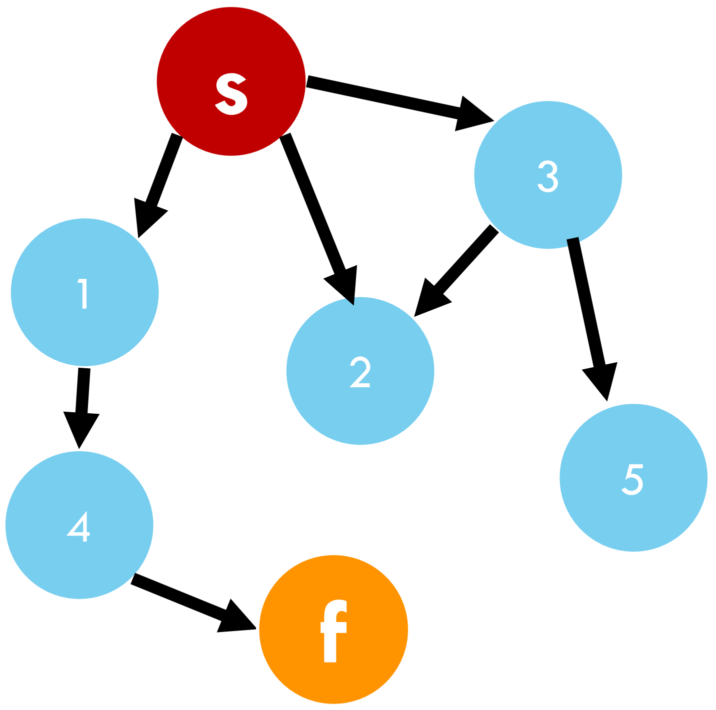


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

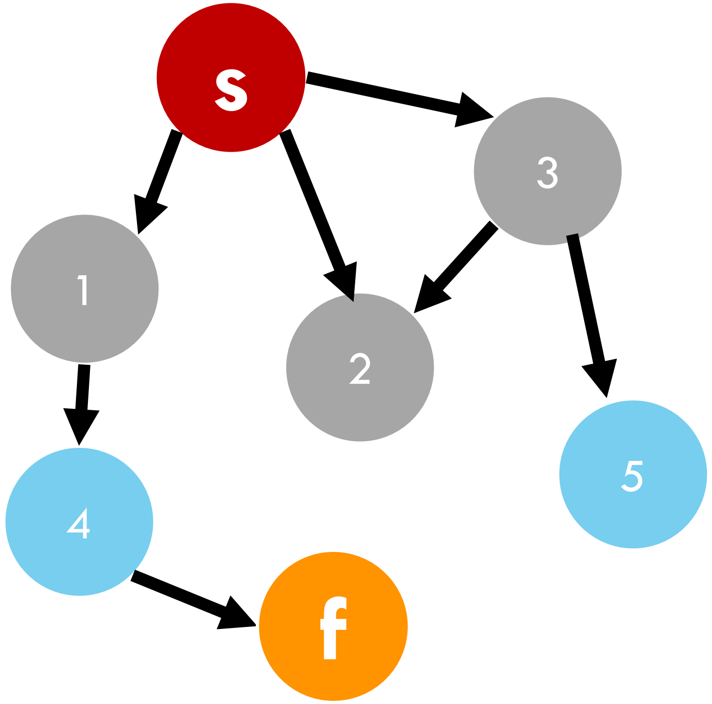


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

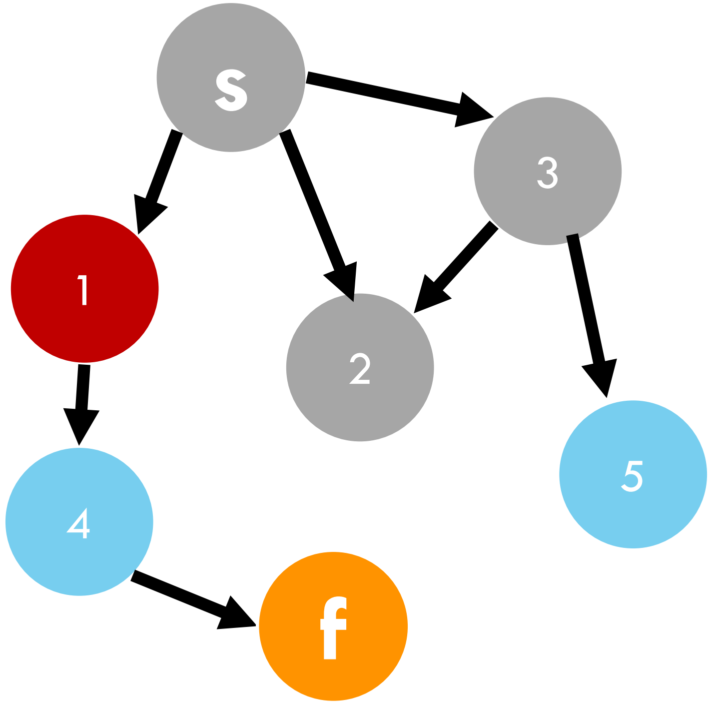
Queue: 1 2 3



BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

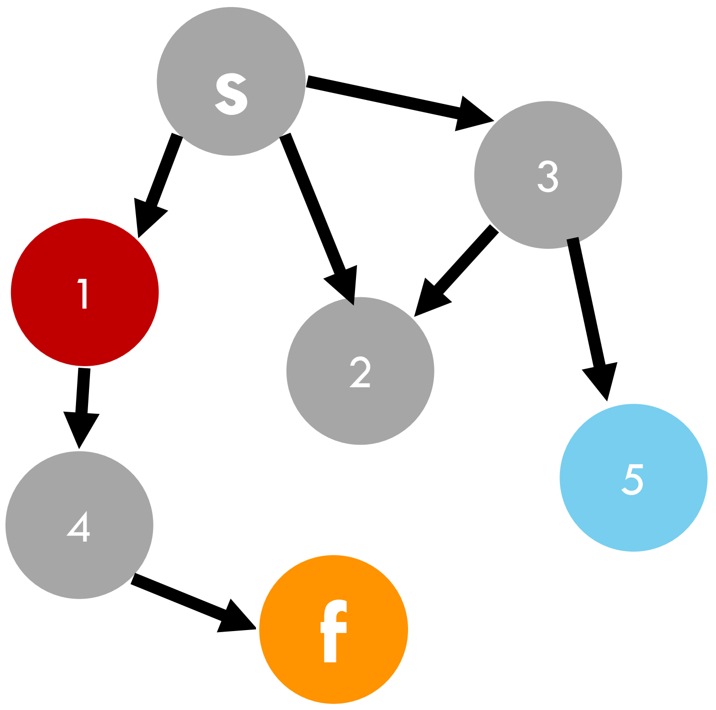
Queue: 2 3



BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 2 3 4

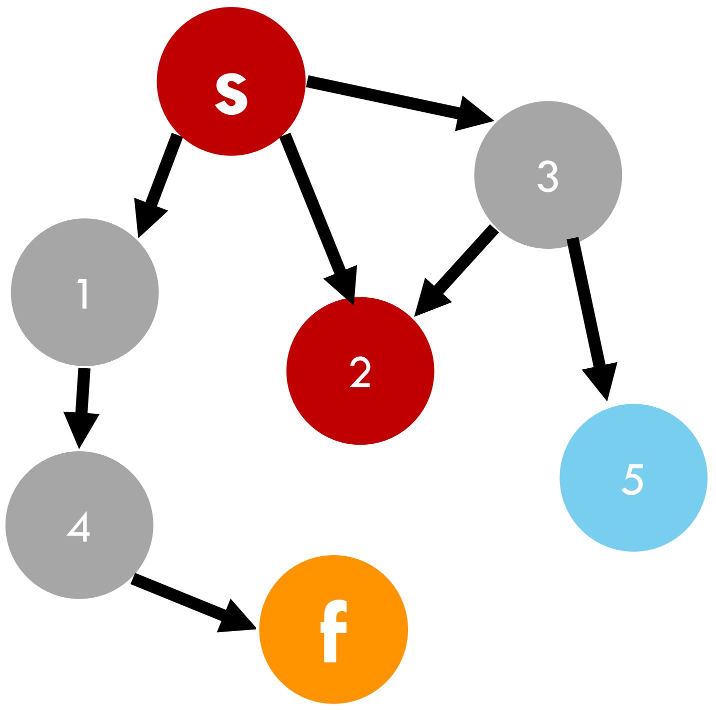


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 3 4

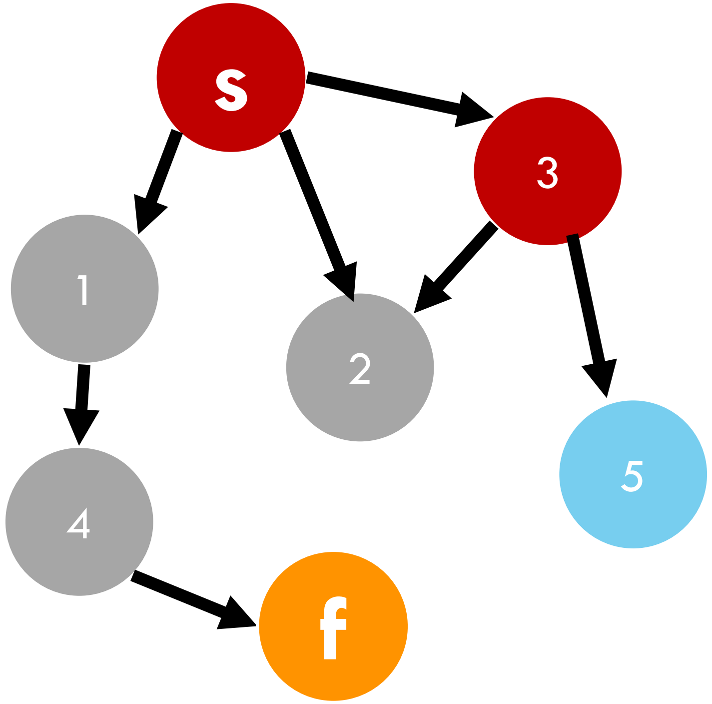


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 4

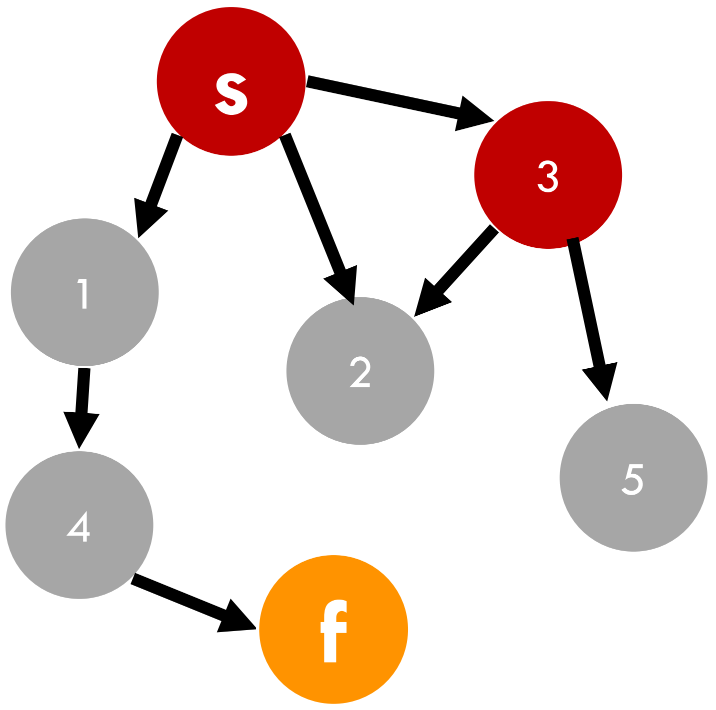


BFS: STEP-BY-STEP

Red = active
Gray = visited
Blue = unvisited

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 4 5

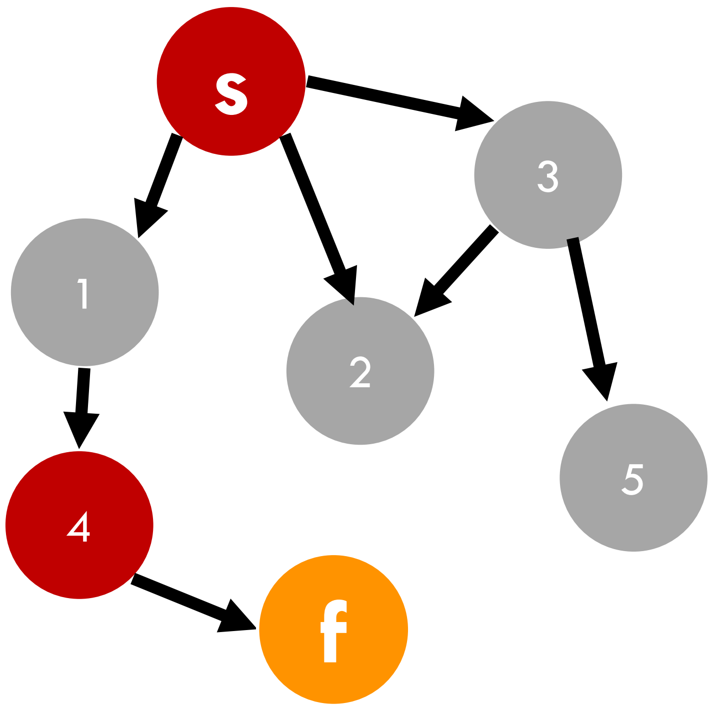


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 5

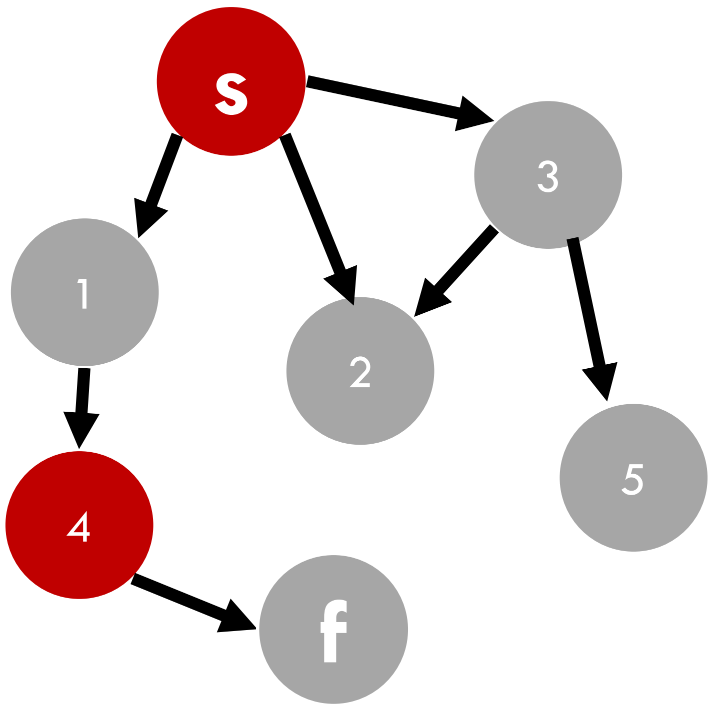


BFS: STEP-BY-STEP

Red = active
Gray = visited
Blue = unvisited

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue: 5 f

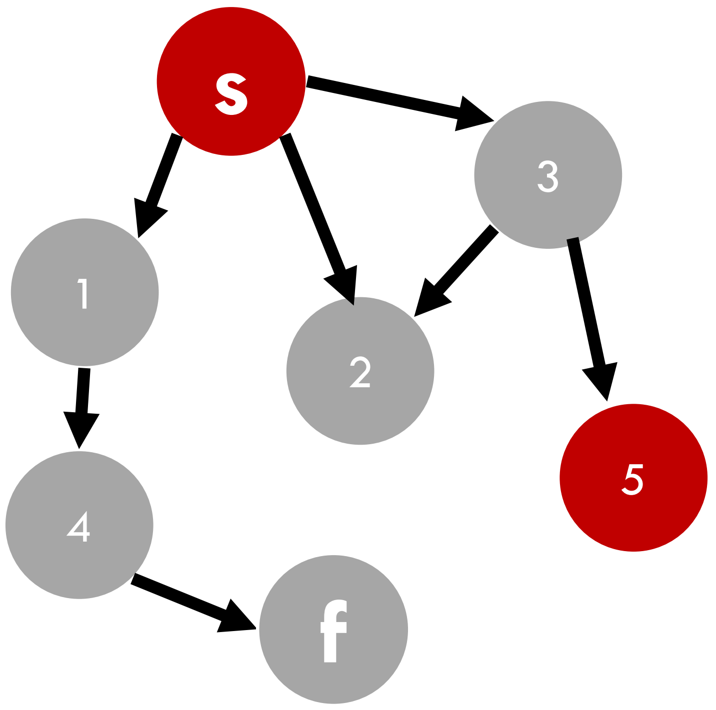


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

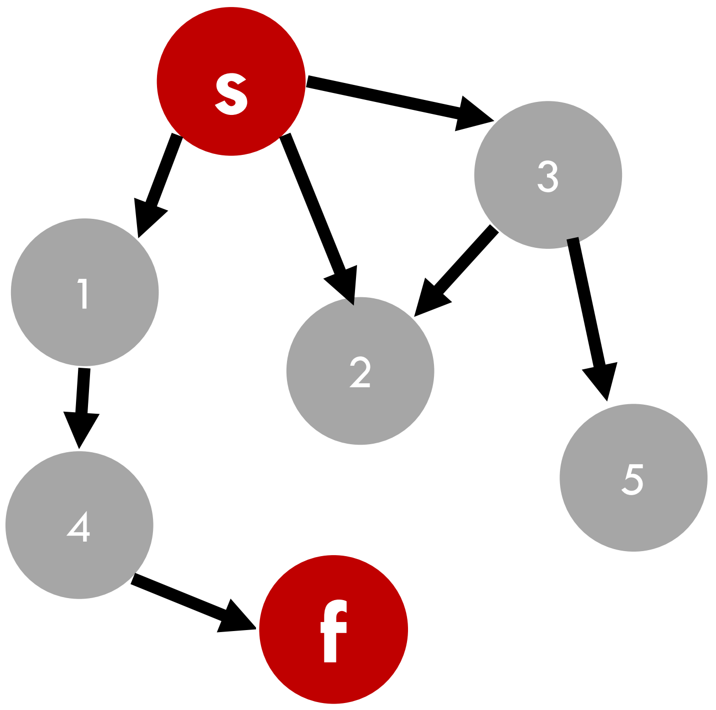
Queue: f



BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

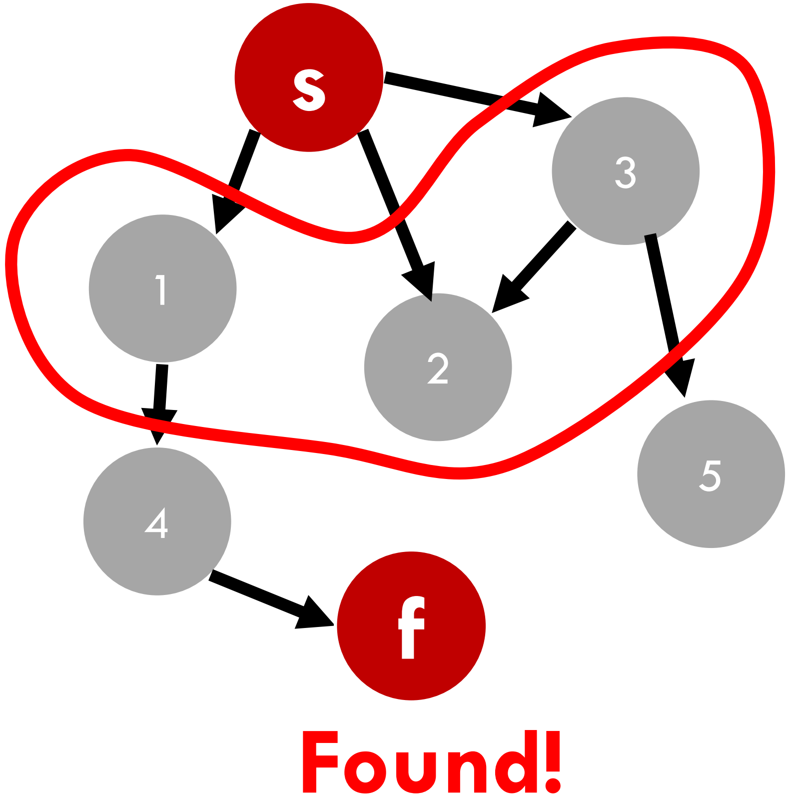


Found!

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

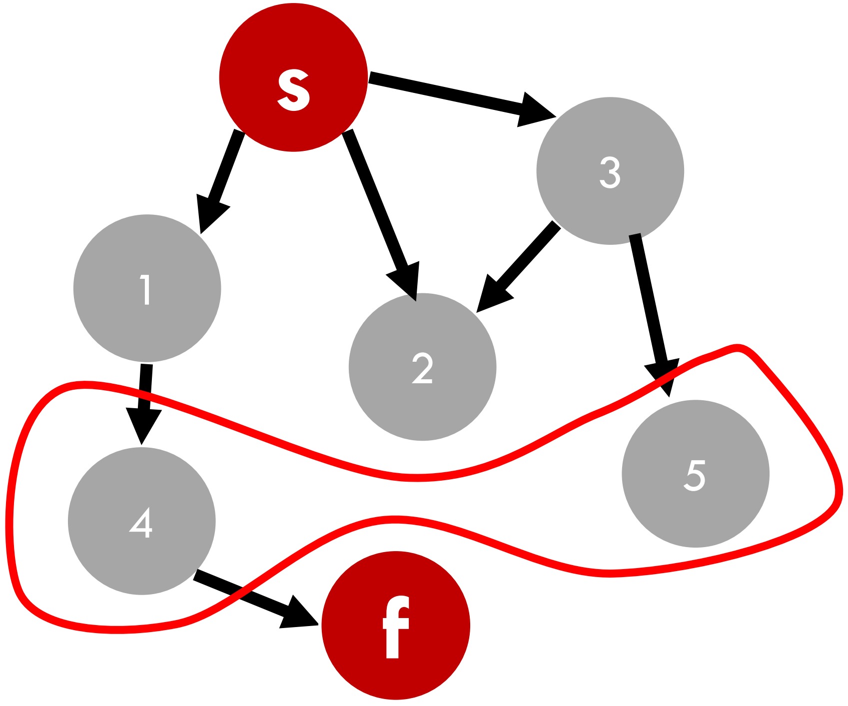


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

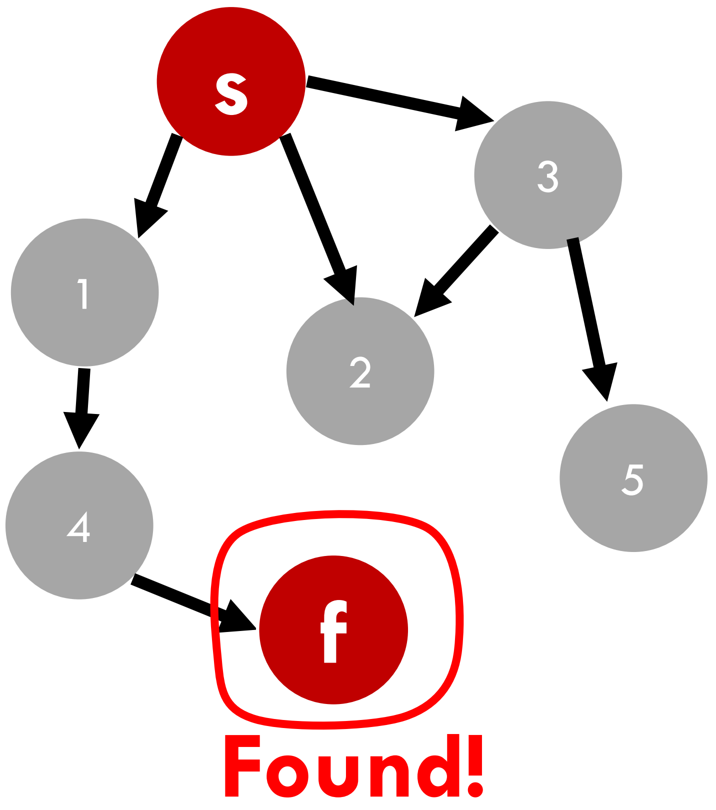


Found!

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:



Poll Everywhere

https://bit.ly/2LvG9bq

CAN BFS FAIL?

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

In what kind of graph can BFS fail?

A. In a clique
B. In a cycle
C. In a graph with > 1 component
D. In a sparse graph
E. In a dense graph
F. It always works!

Poll Everywhere

https://bit.ly/2LvG9bq

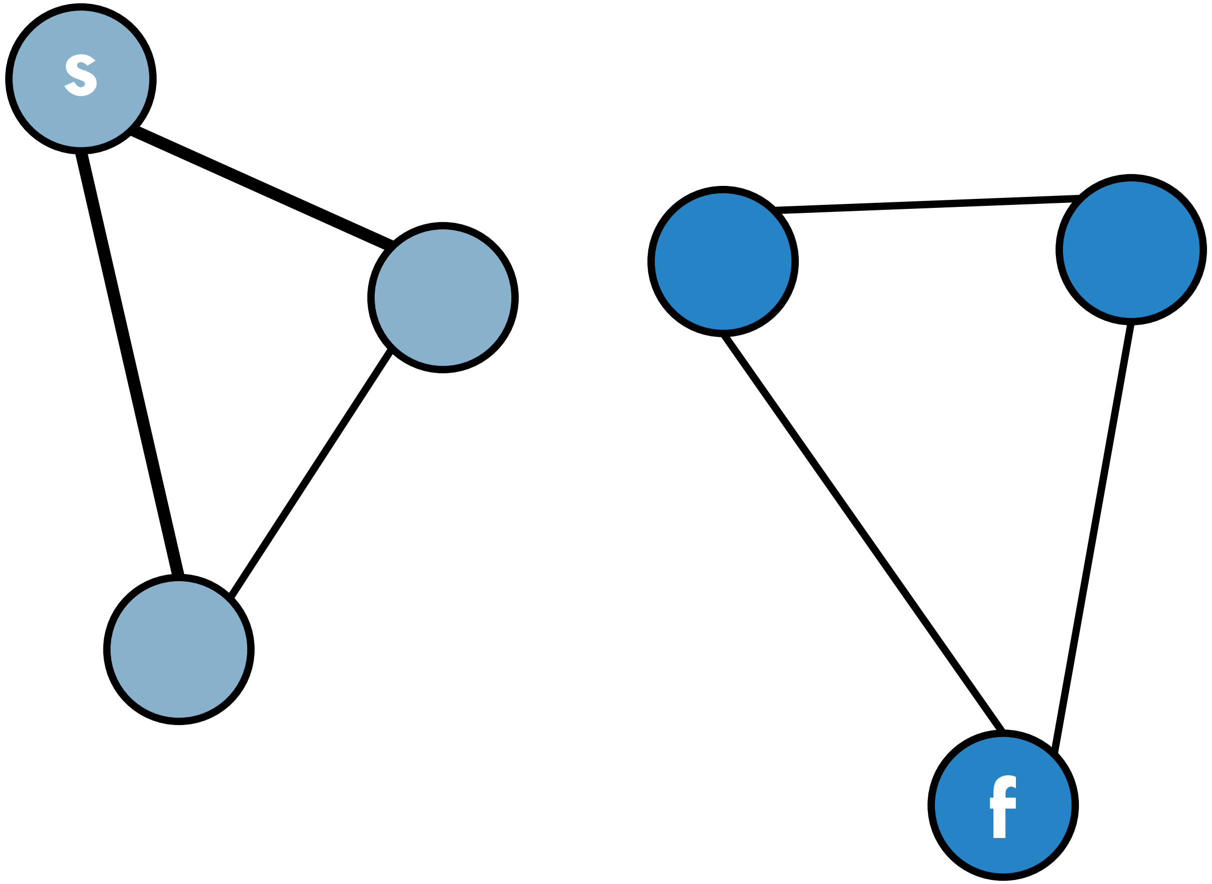
CAN BFS FAIL?

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

In what kind of graph can BFS fail?

A. In a clique
B. In a cycle
C. **In a graph with > 1 component**
D. In a sparse graph
E. In a dense graph
F. It always works!

BFS ON A DISCONNECTED GRAPH



WORST-CASE TIME COMPLEXITY

Poll Everywhere

https://bit.ly/2LvG9bq

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

What is the running time of BFS? (assume adj list)

A. $O(V)$
B. $O(E)$
C. $O(V+E)$
D. $O(VE)$
E. $O(V^2)$
F. I have no idea.

Poll Everywhere
https://bit.ly/2LvG9bq

WORST-CASE TIME COMPLEXITY

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

What is the running time of BFS? (assume adj list)

A. $O(V)$
B. $O(E)$
C. $\boldsymbol{O(V+E)}$
D. $O(VE)$
E. $O(V^2)$
F. I have no idea.

WORST-CASE TIME COMPLEXITY

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Analysis:

- Vertex v = “start” once.
- Vertex v added to queue once.
  - After visited, never re-added.

$O(V)$

- Each list of neighbors is enumerated once.
  - When v is removed from frontier.

$O(E)$

SEARCHING A GRAPH

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

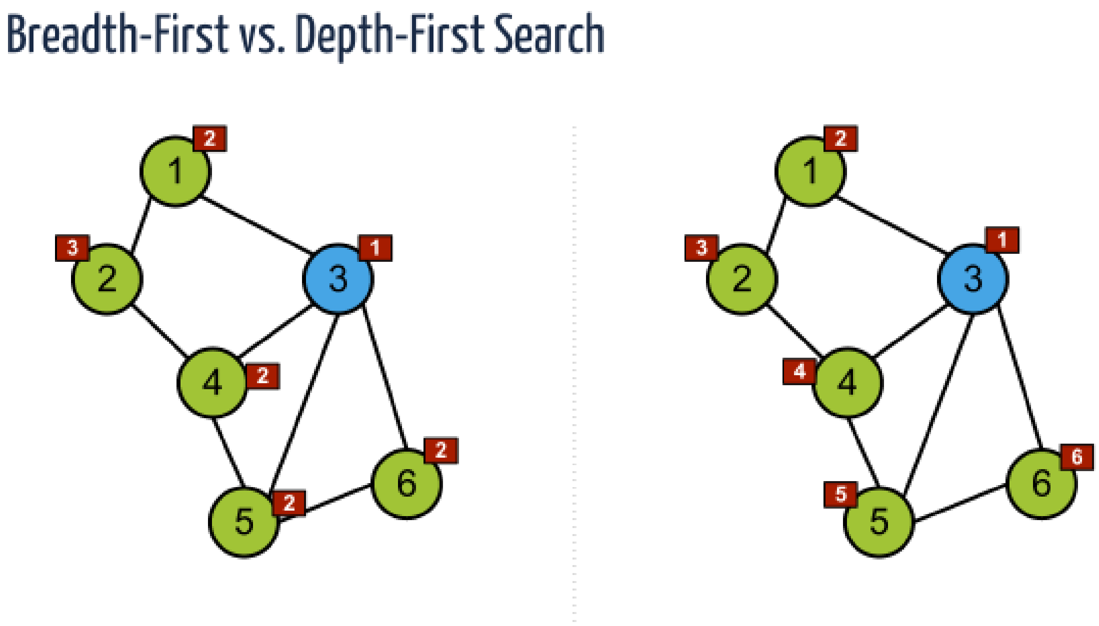
  Or: visit **all** the nodes in the graph

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list



DEPTH-FIRST SEARCH (DFS)

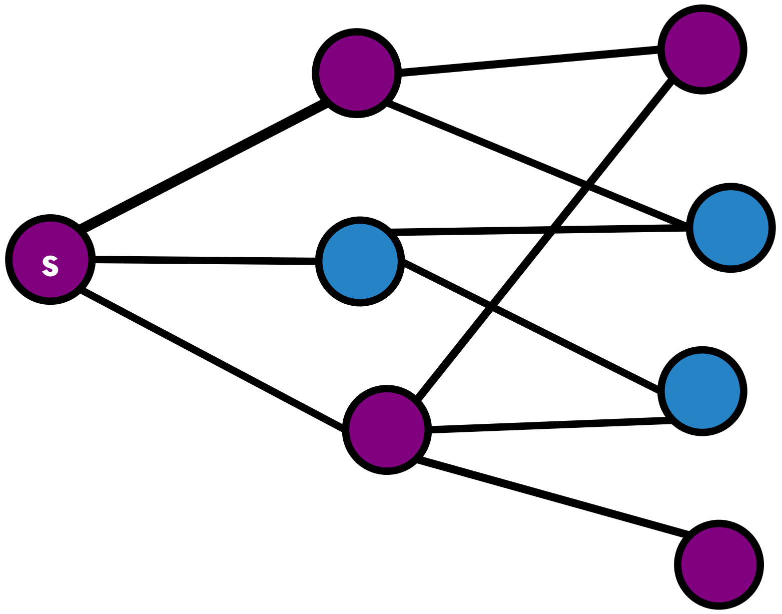
Exploring a maze:

- Follow path until stuck.
- Backtrack along breadcrumbs until reach unexplored neighbor.
- Recursively explore.

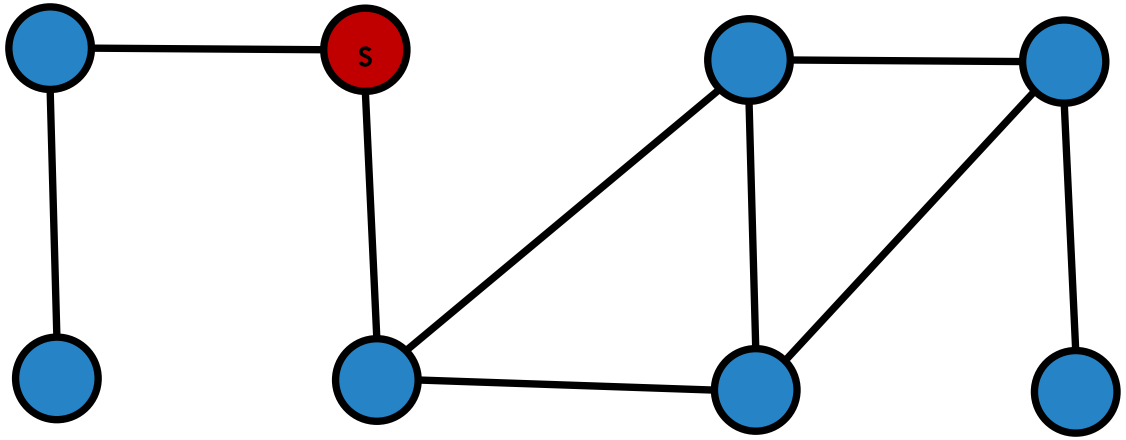
DEPTH-FIRST SEARCH (DFS)

Strategy

- Follow path until you get stuck
- Backtrack until you find a new edge
- Recursively explore it
- Don't repeat a vertex.



DEPTH-FIRST SEARCH EXAMPLE



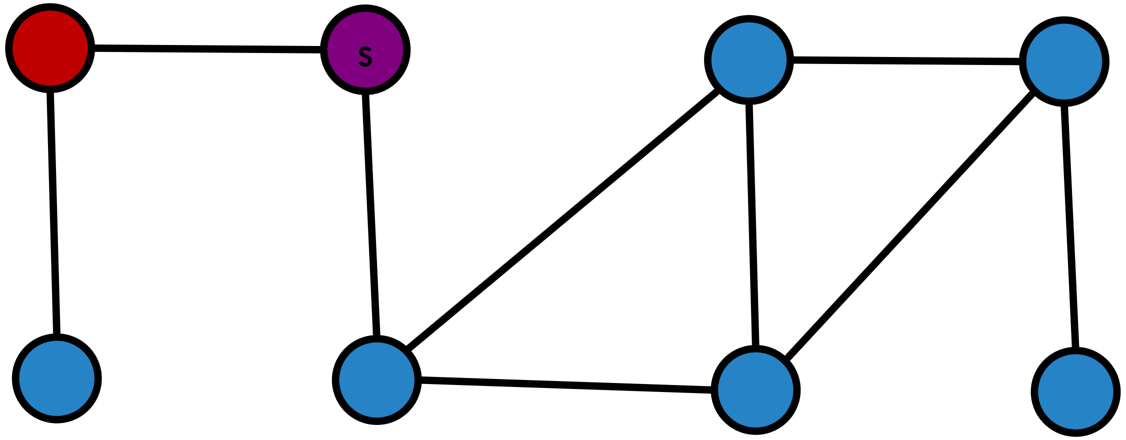
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



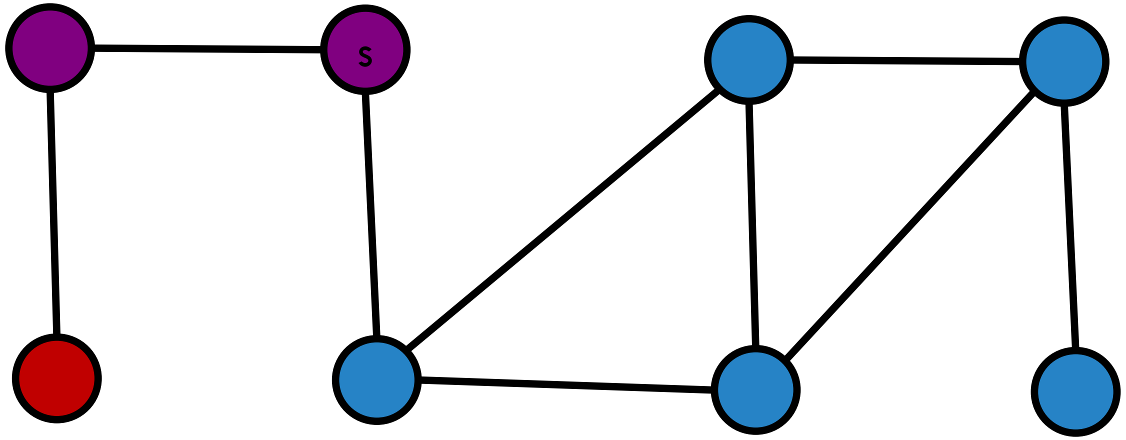
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



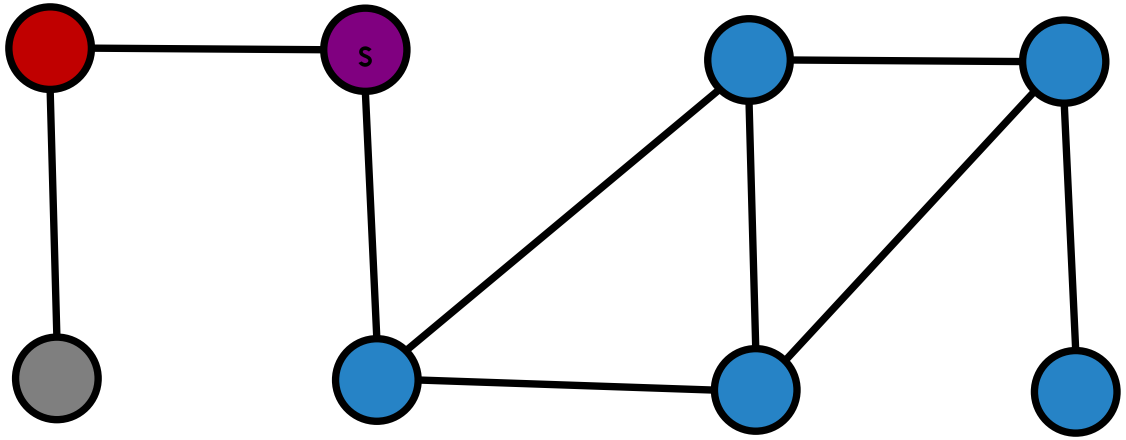
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



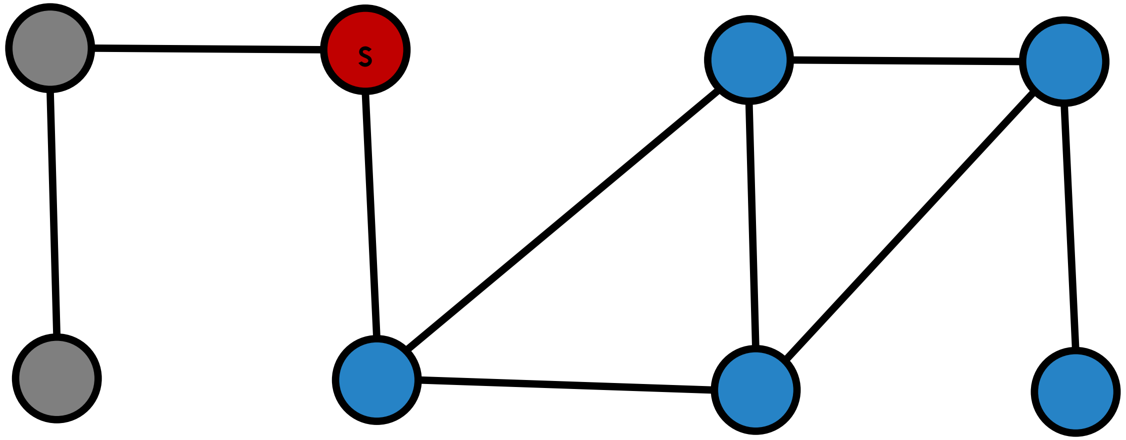
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



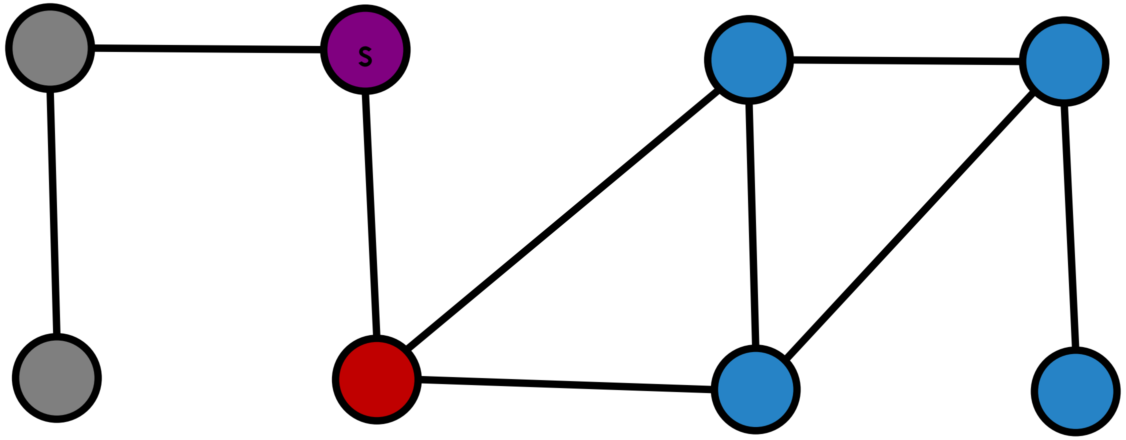
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



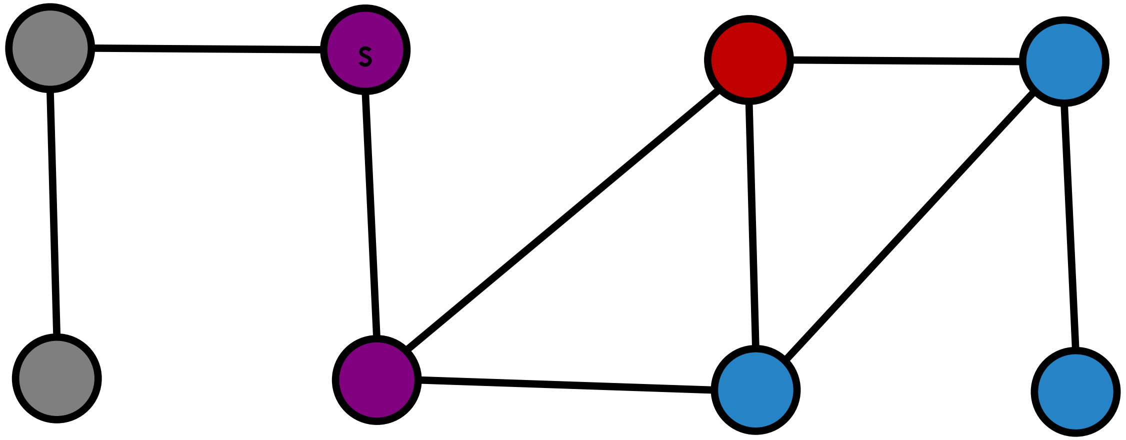
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



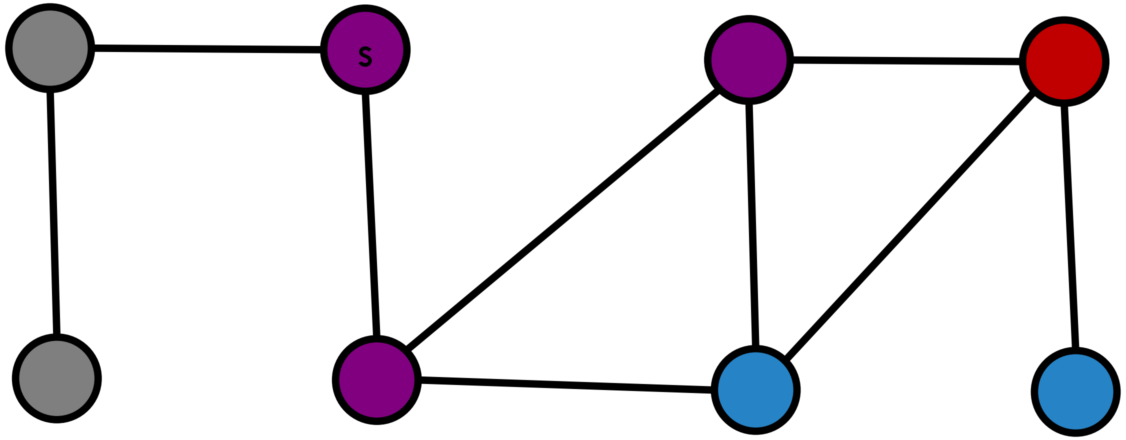
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



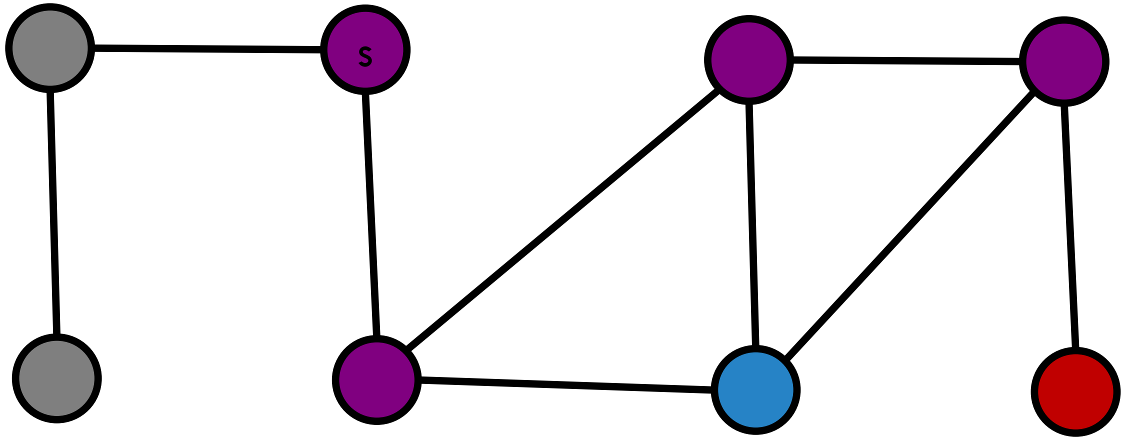
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



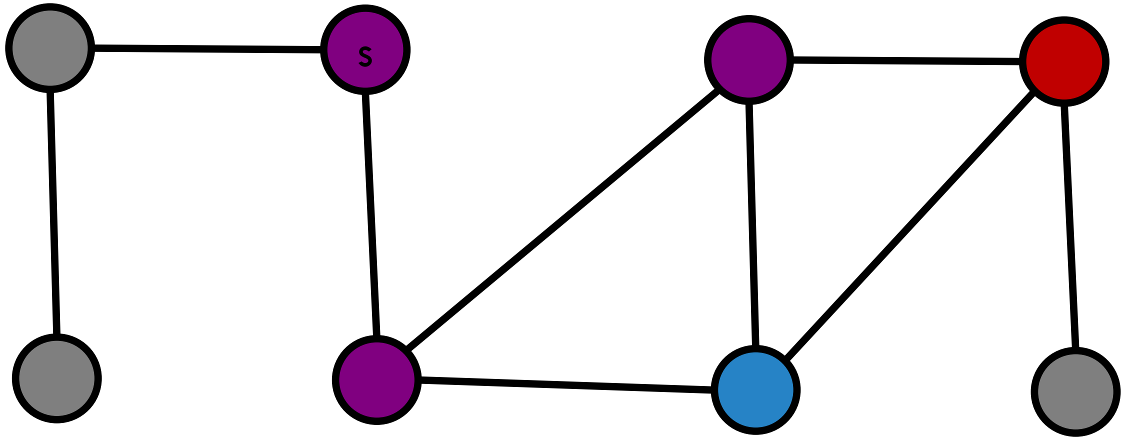
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



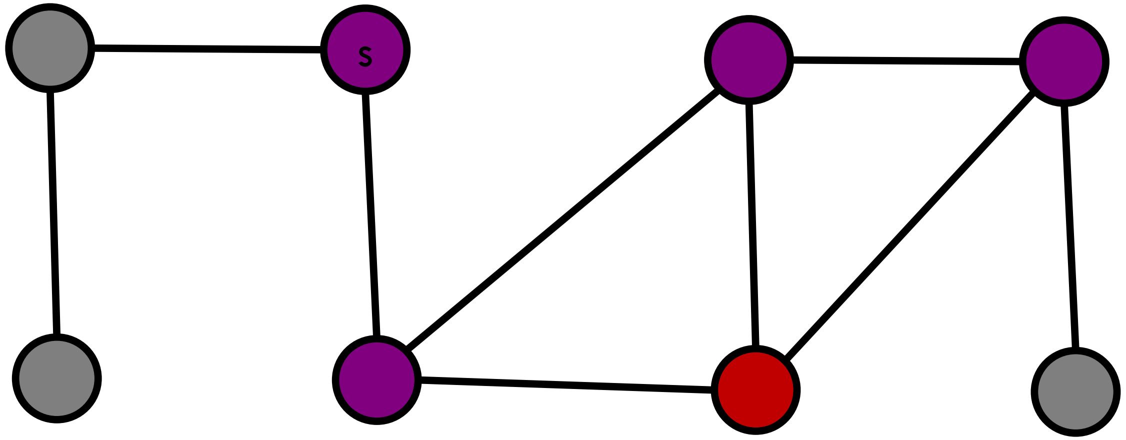
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



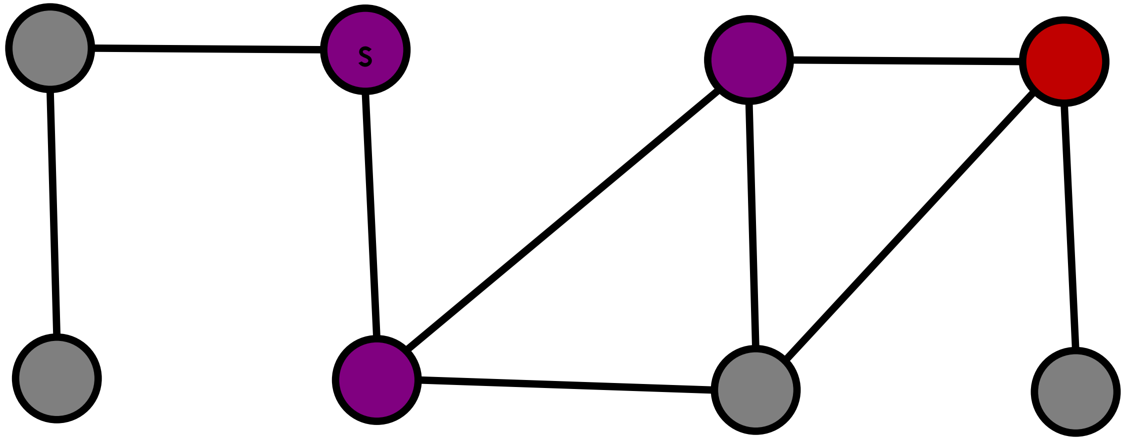
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



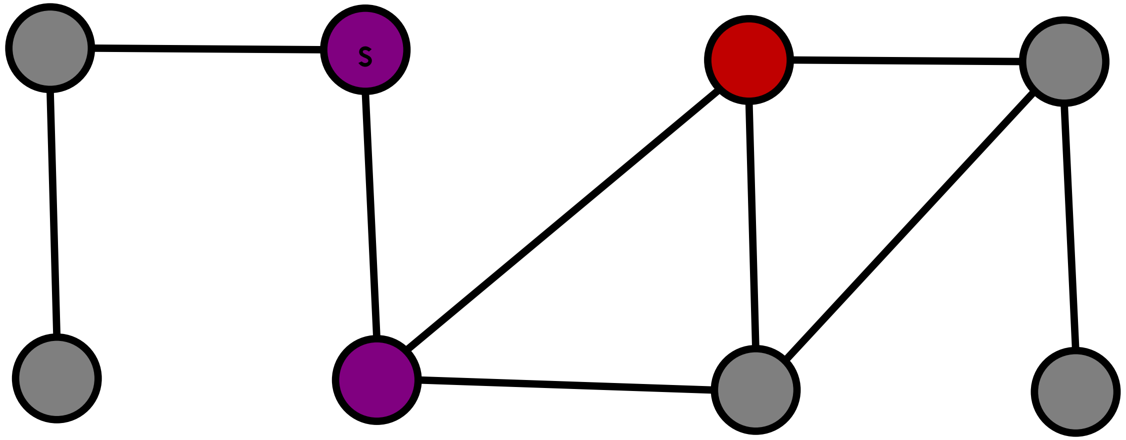
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



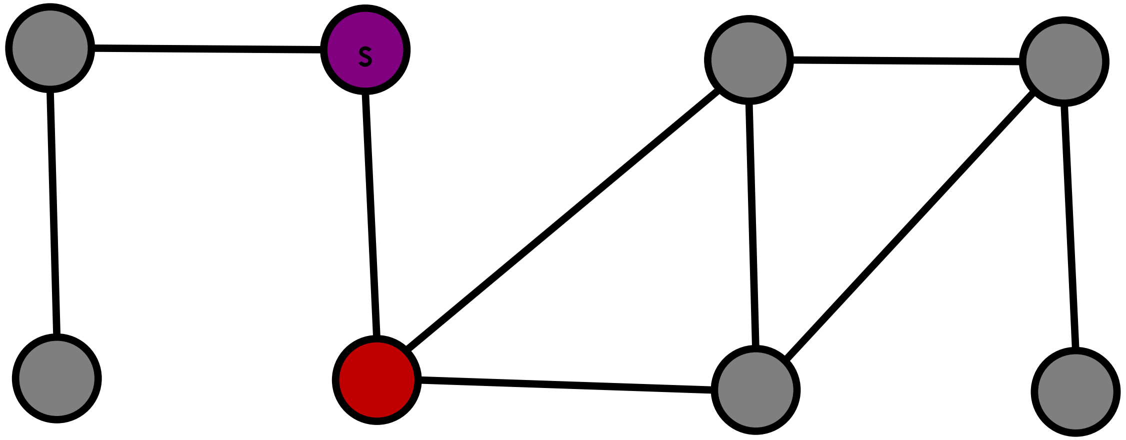
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



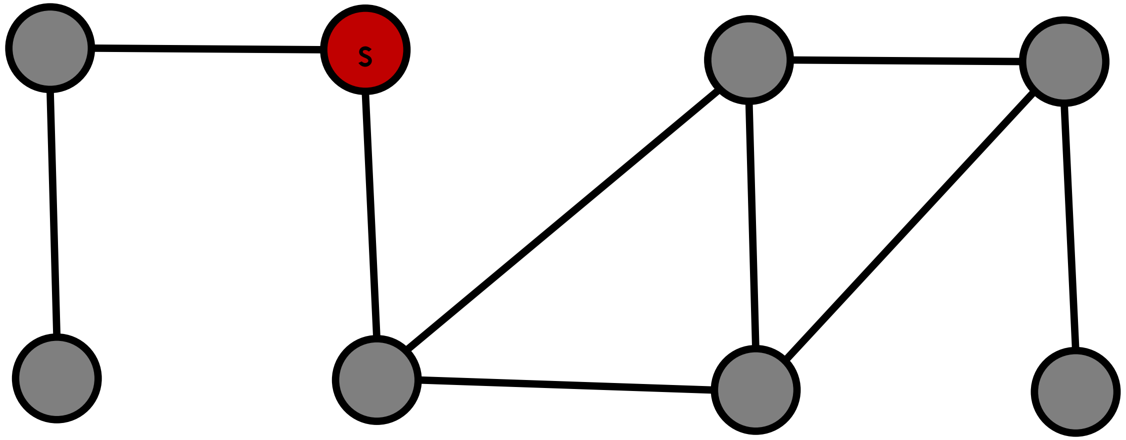
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



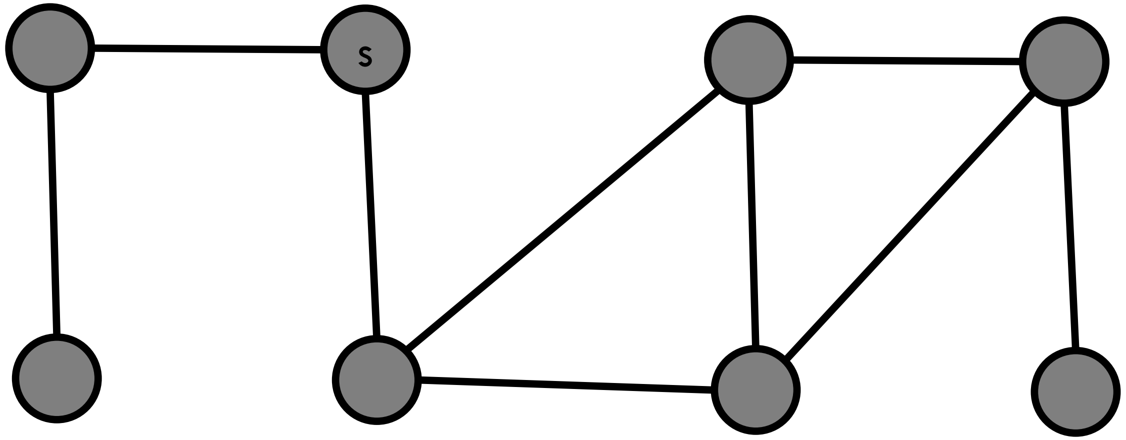
Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

DEPTH-FIRST SEARCH EXAMPLE



Red = active frontier

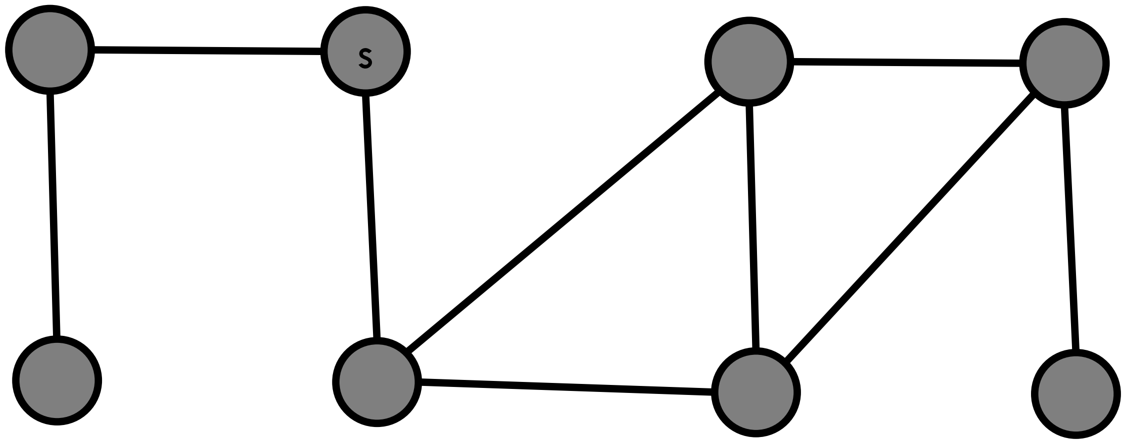
Purple = next

Gray = visited

Blue = unvisited

BFS PSEUDOCODE

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```



Red = active frontier

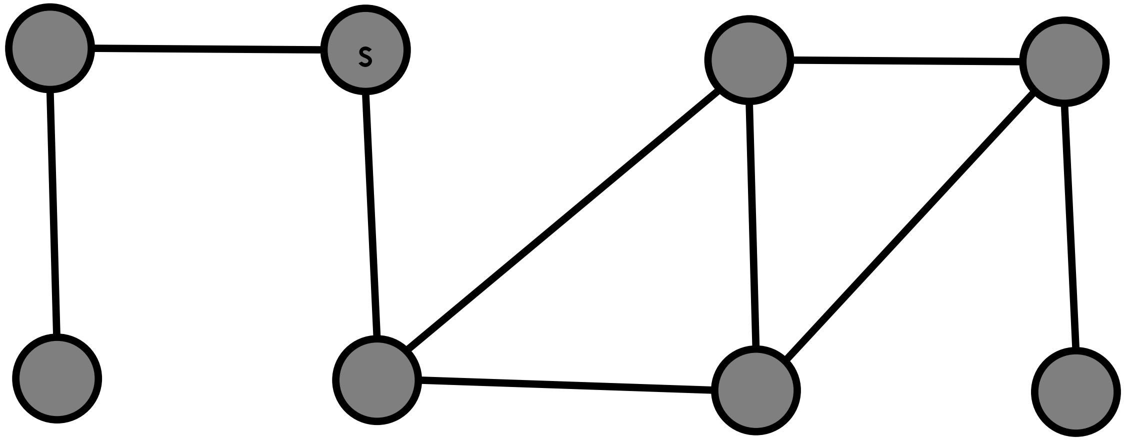
Purple = next

Gray = visited

Blue = unvisited

DFS PSEUDOCODE

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```



Red = active frontier

Purple = next

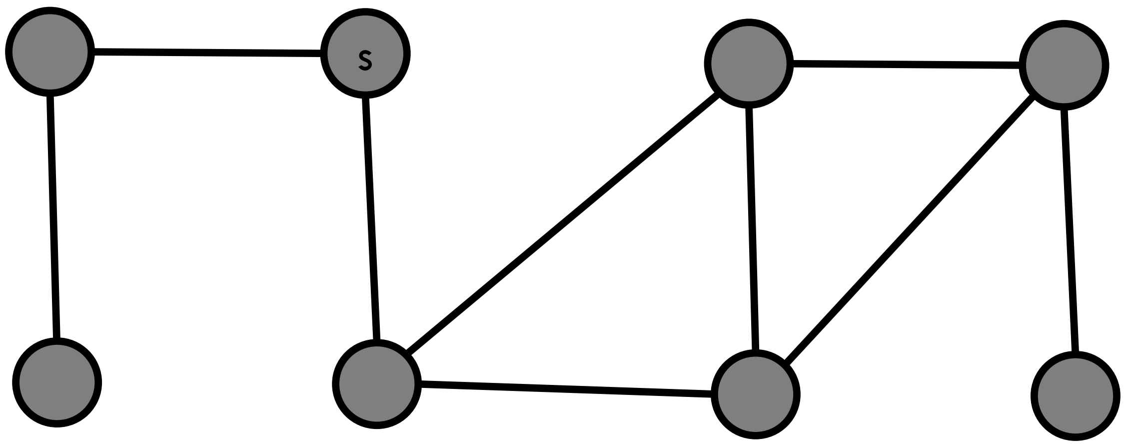
Gray = visited

Blue = unvisited

DFS PSEUDOCODE

BFS and DFS are the "same" algorithm!
One uses a queue and one uses a stack!

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```



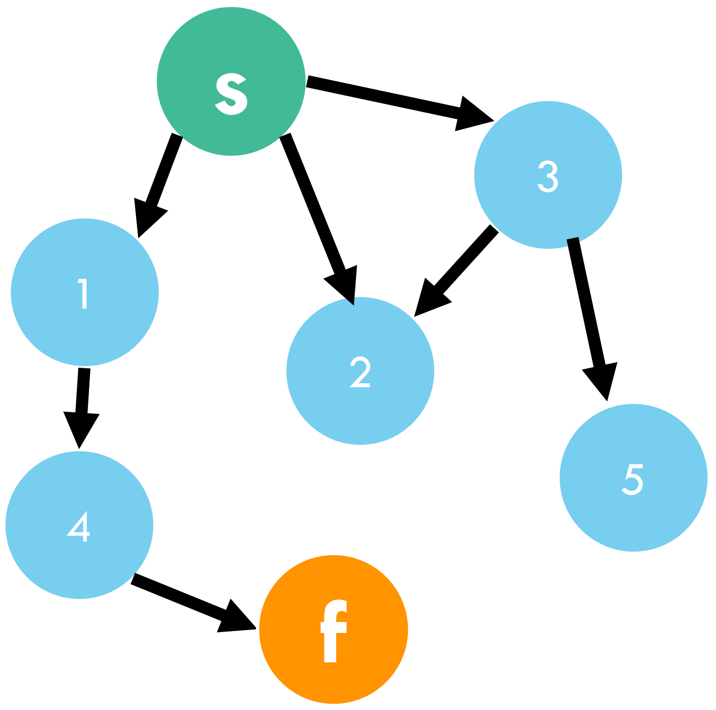
Red = active frontier
Purple = next
Gray = visited
Blue = unvisited

Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:

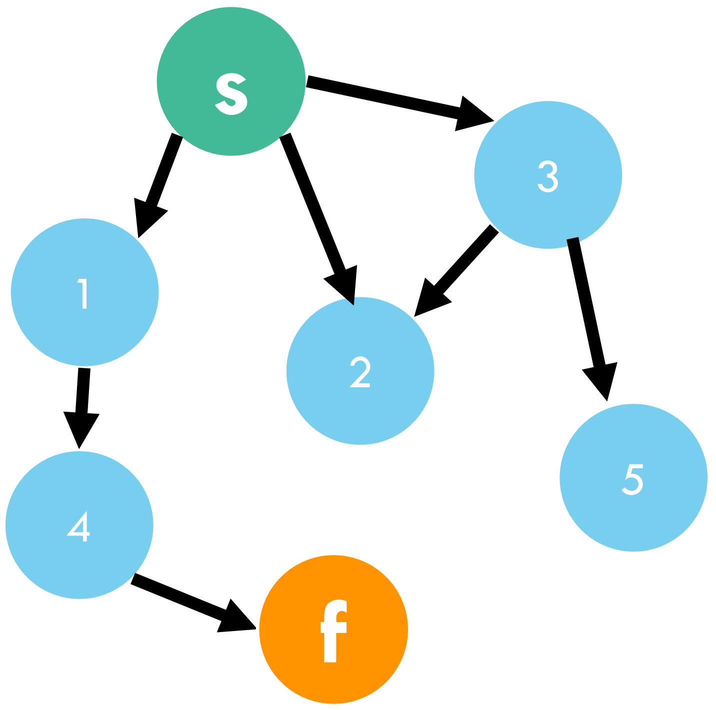


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack: s

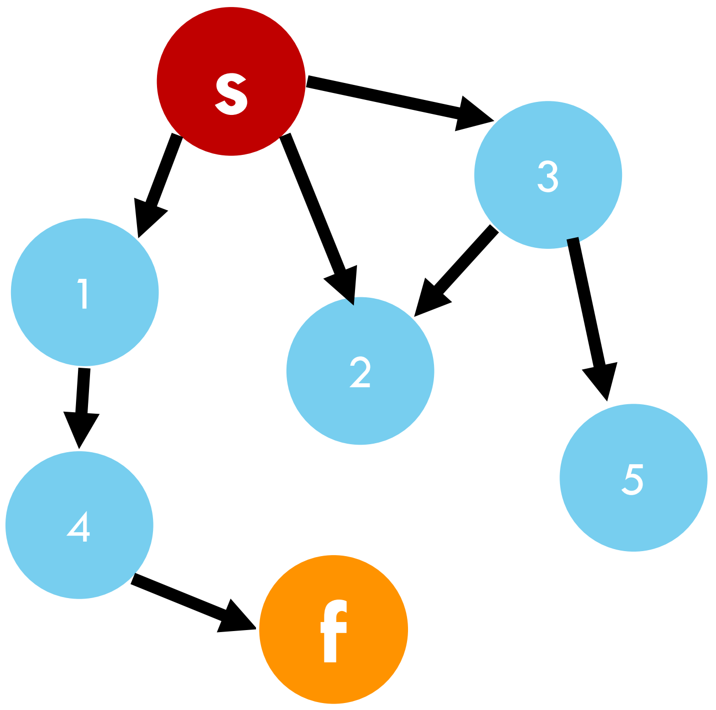


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

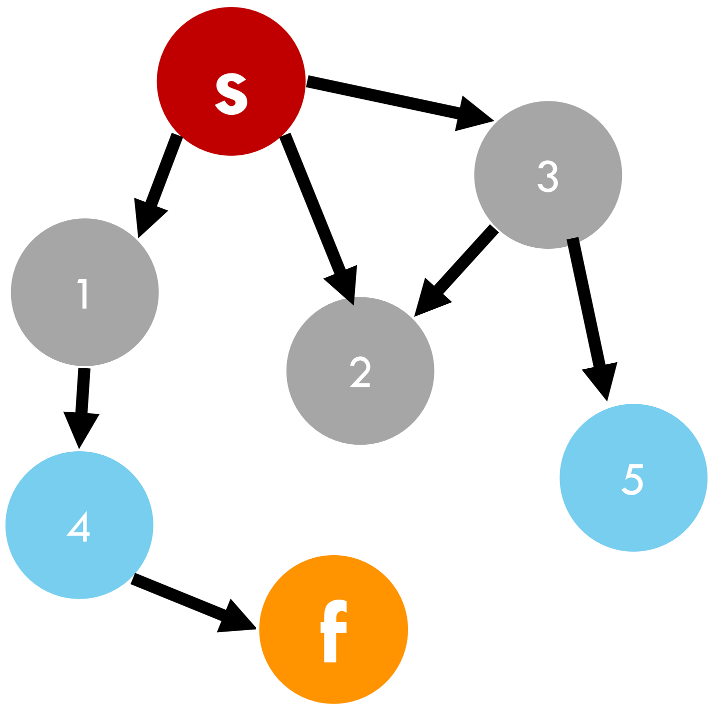
Stack:



DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack: 1 2 3

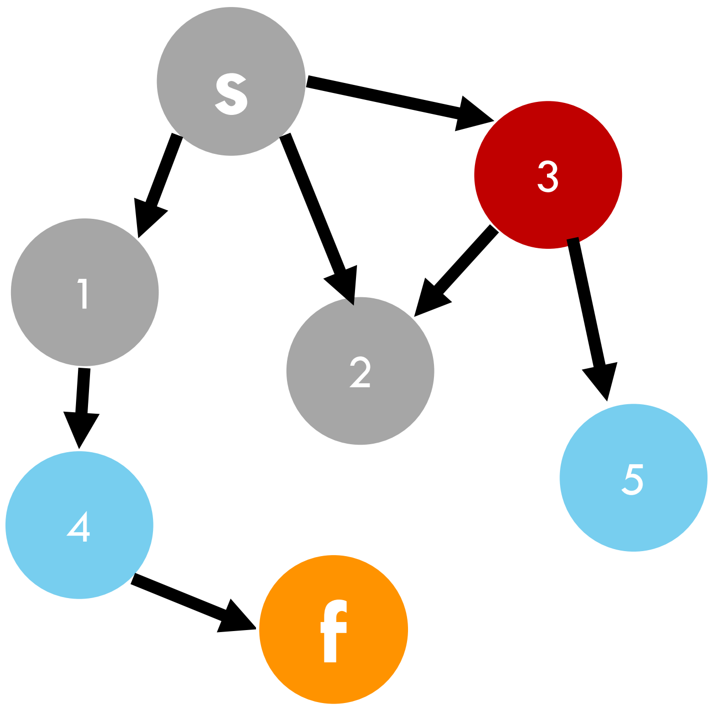


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack: 1 2

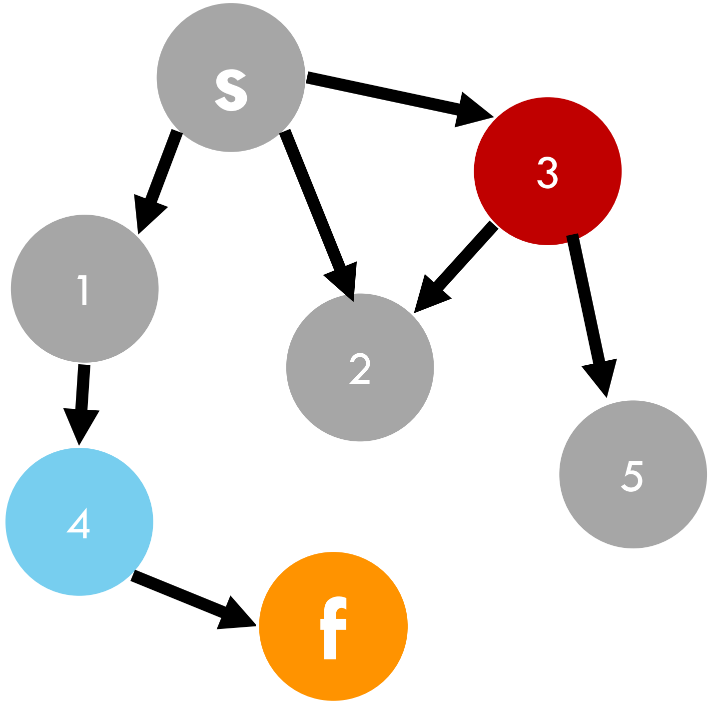


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

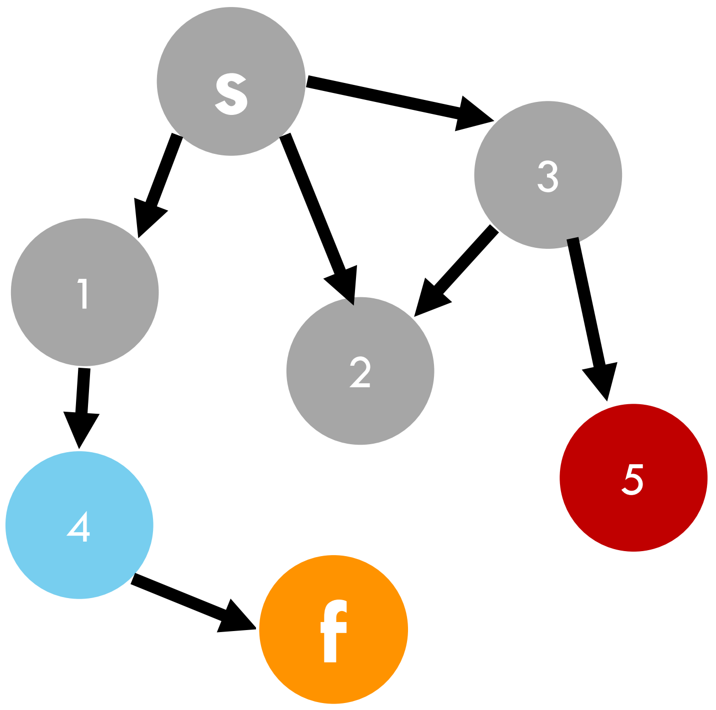
Stack: 1 2 5



DFS: STEP-BY-STEP

Stack: 1 2

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

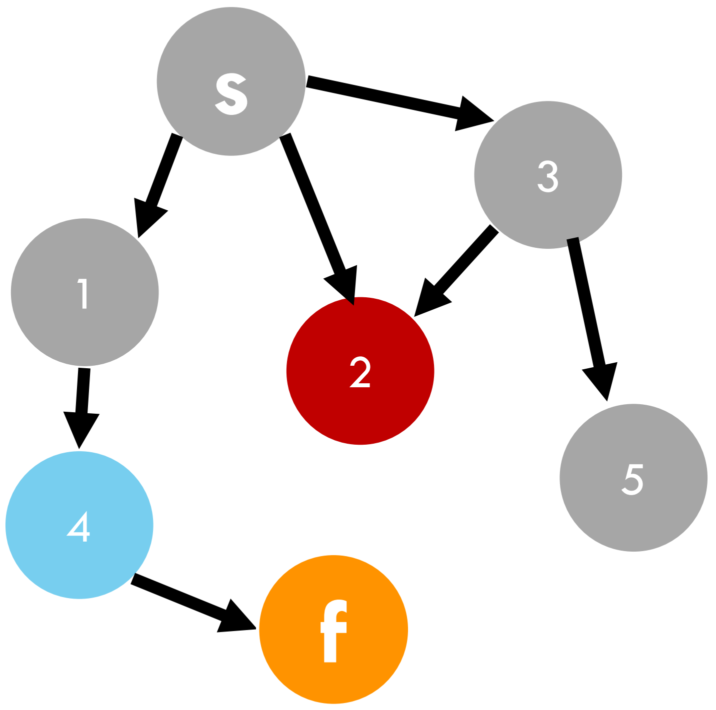


DFS: STEP-BY-STEP

Red = active
Gray = visited
Blue = unvisited

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack: 1

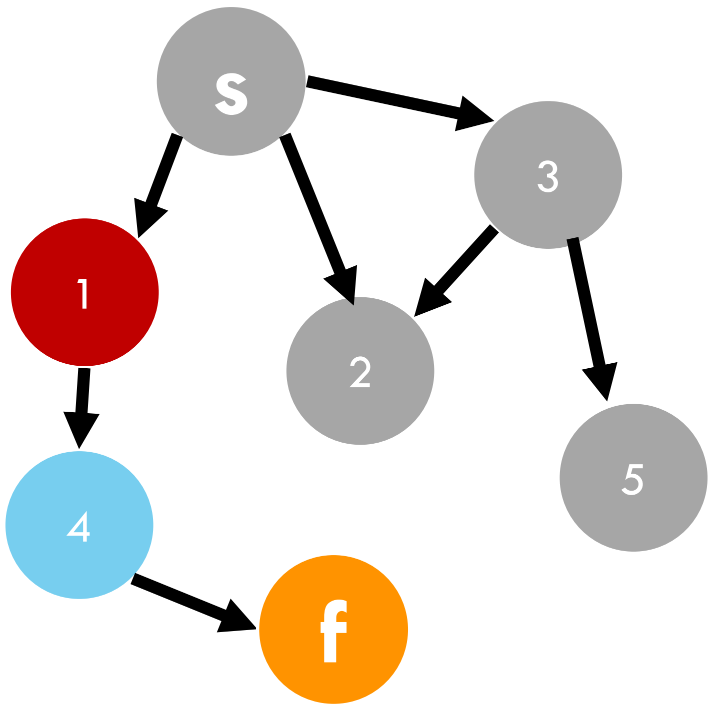


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

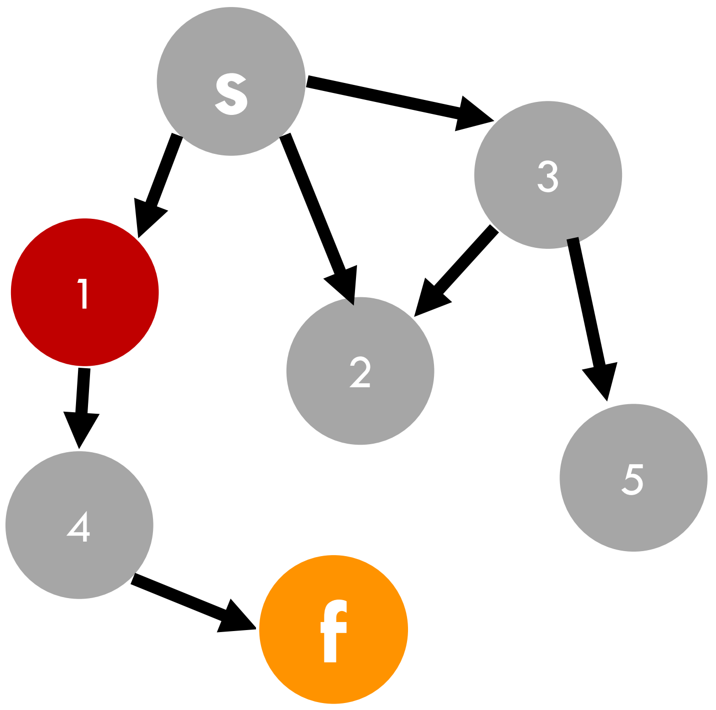
Stack:



DFS: STEP-BY-STEP

Stack: 4

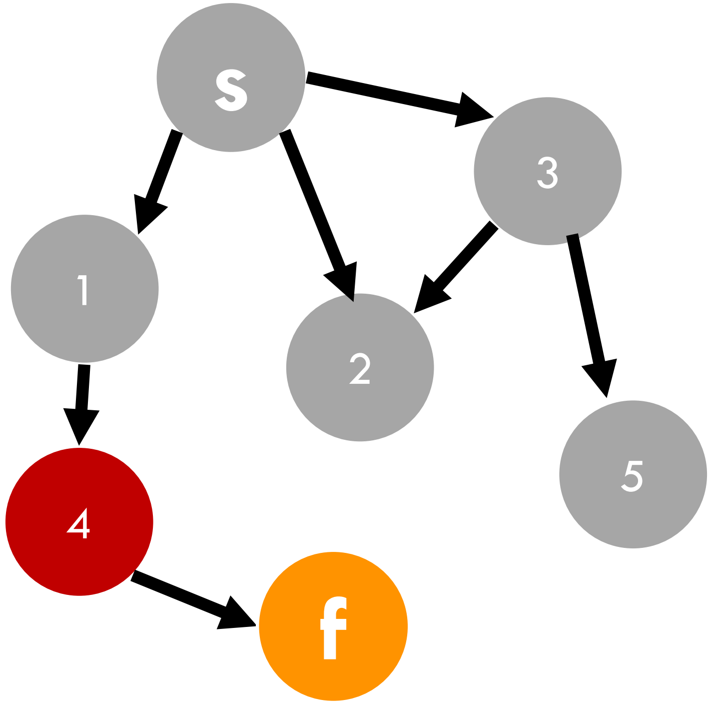
```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```



DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:

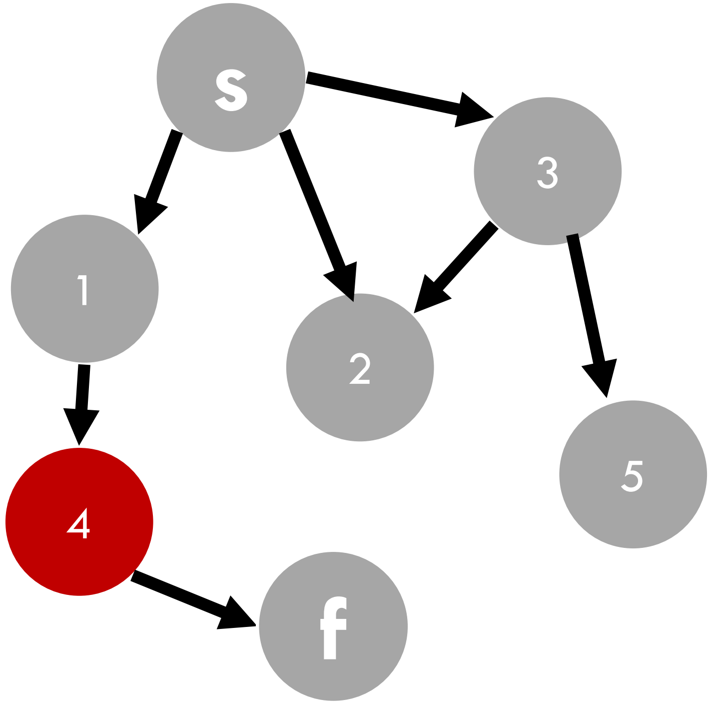


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack: f

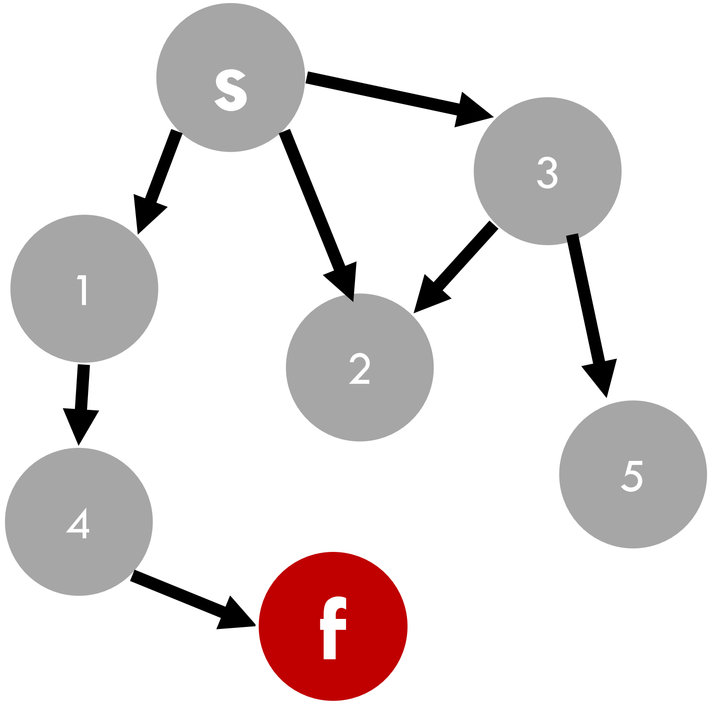


Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:

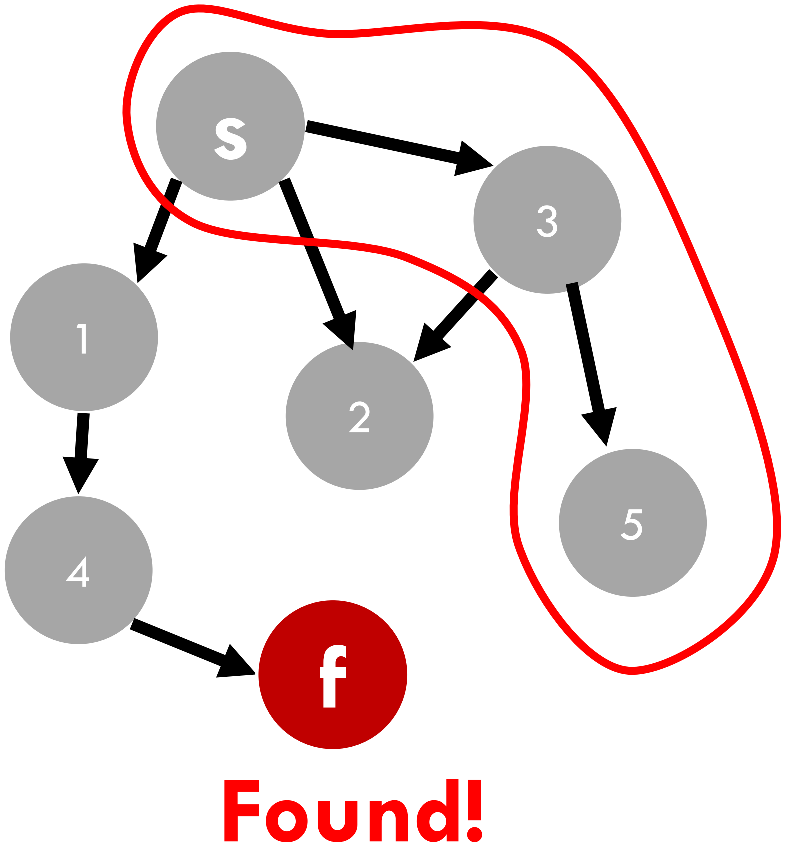


Found!

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:

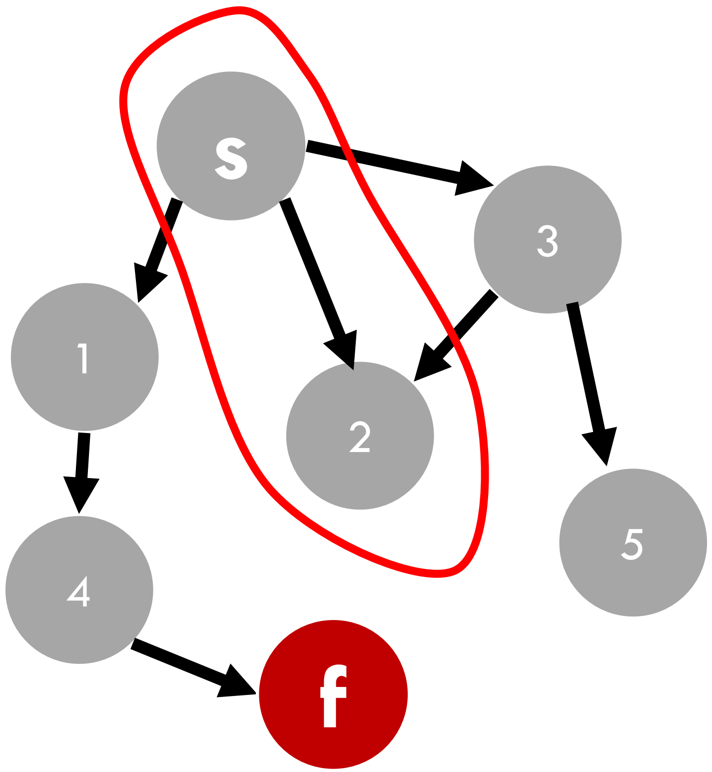


DFS: STEP-BY-STEP

Red = active
Gray = visited
Blue = unvisited

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



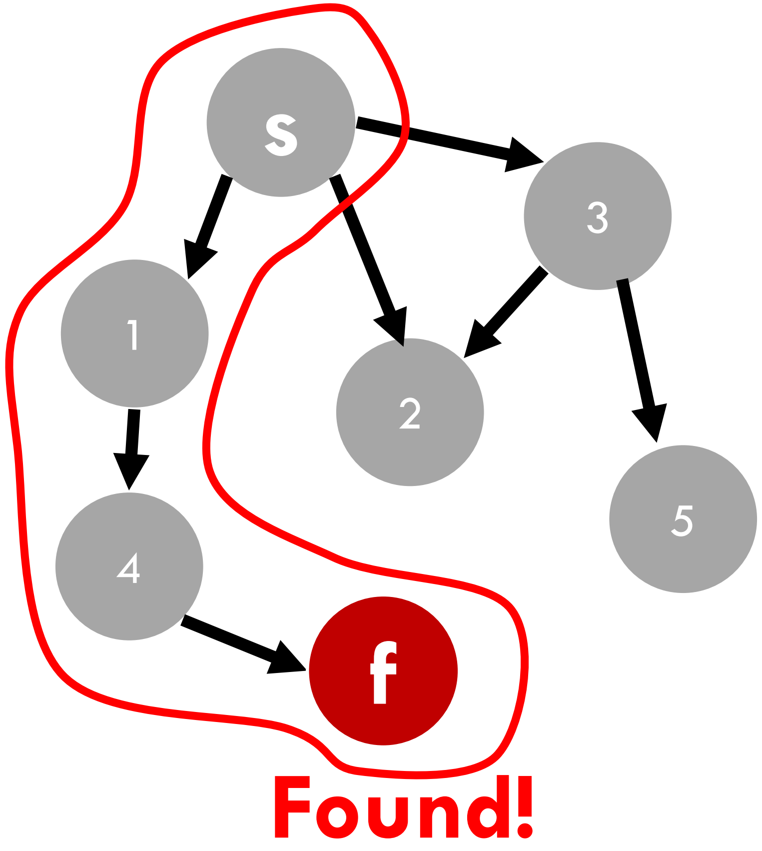
Found!

Red = active
Gray = visited
Blue = unvisited

DFS: STEP-BY-STEP

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Stack:



Poll Everywhere
https://bit.ly/2LvG9bq

WORST-CASE TIME COMPLEXITY

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

What is the running time of DFS?

A. $O(V)$
B. $O(E)$
C. $O(V+E)$
D. $O(VE)$
E. $O(V^2)$
F. I have no idea.

Poll Everywhere
https://bit.ly/2LvG9bq

WORST-CASE TIME COMPLEXITY

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

What is the running time of DFS?

A. $O(V)$
B. $O(E)$
C. $\boldsymbol{O(V+E)}$
D. $O(VE)$
E. $O(V^2)$
F. I have no idea.

WORST-CASE TIME COMPLEXITY

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

Analysis:

- Vertex v = "start" once.
- Vertex v added to stack once.
  - After visited, never re-added.

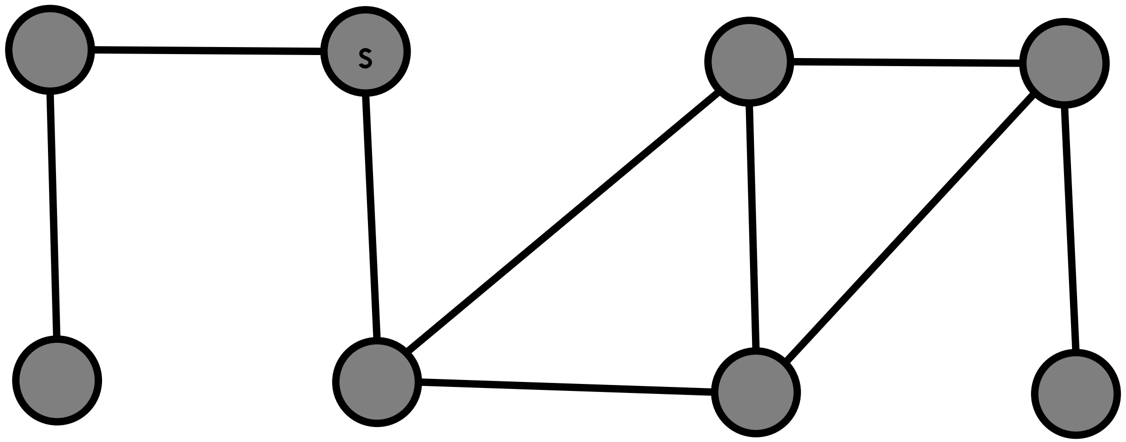
$O(V)$

- Each list of neighbors is enumerated once.

$O(E)$

DFS: A RECURSIVE VERSION

```
DFS(G, v, f)
    visit(v)
    if v == f
        // we found Herbert!
        return v
    for each neighbor u of v
        if u is not visited
            w = DFS(G, u, f)
            if w is not null
                return w
    return null
```



Red = active frontier

Purple = next

Gray = visited

Blue = unvisited

SEARCHING A GRAPH

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

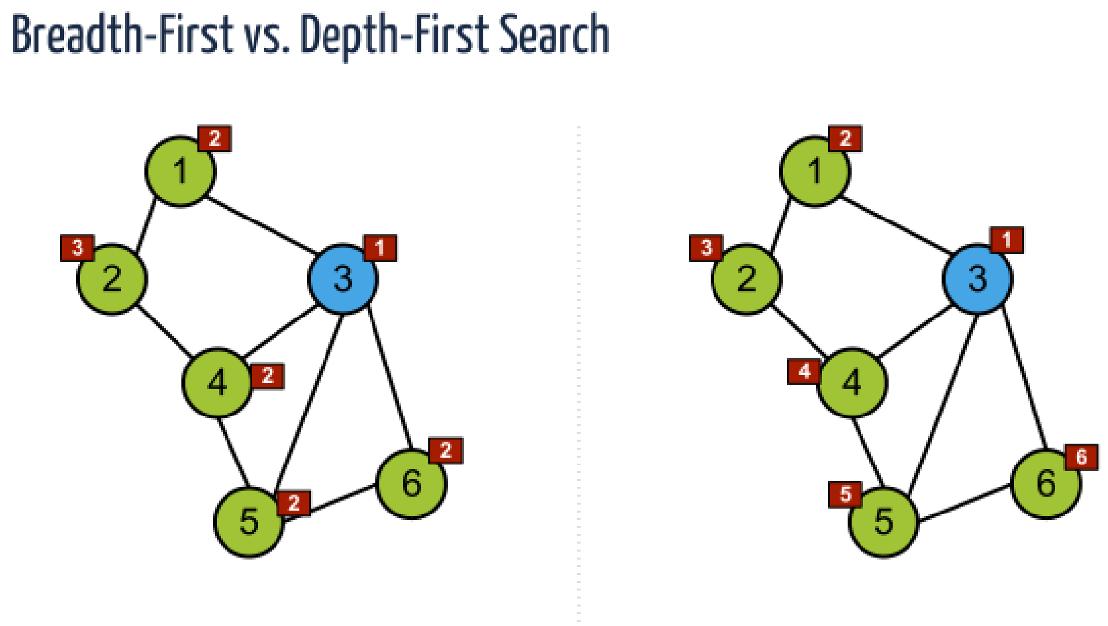
  Or: visit **all** the nodes in the graph

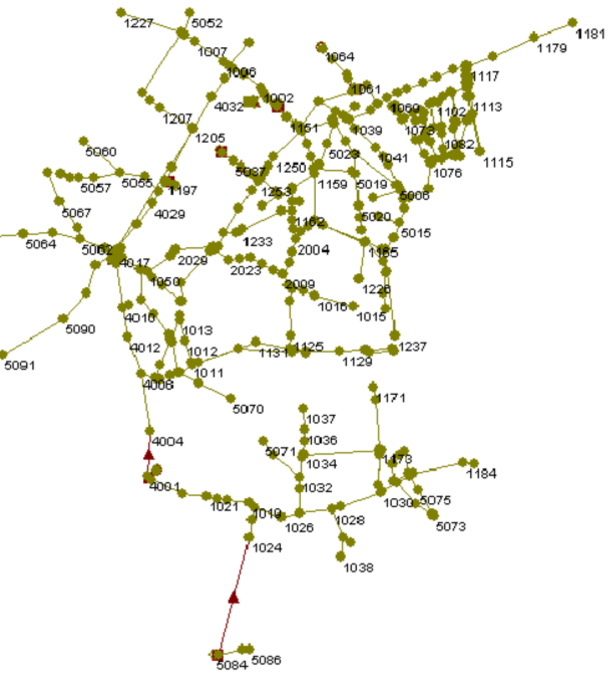
Two basic techniques:

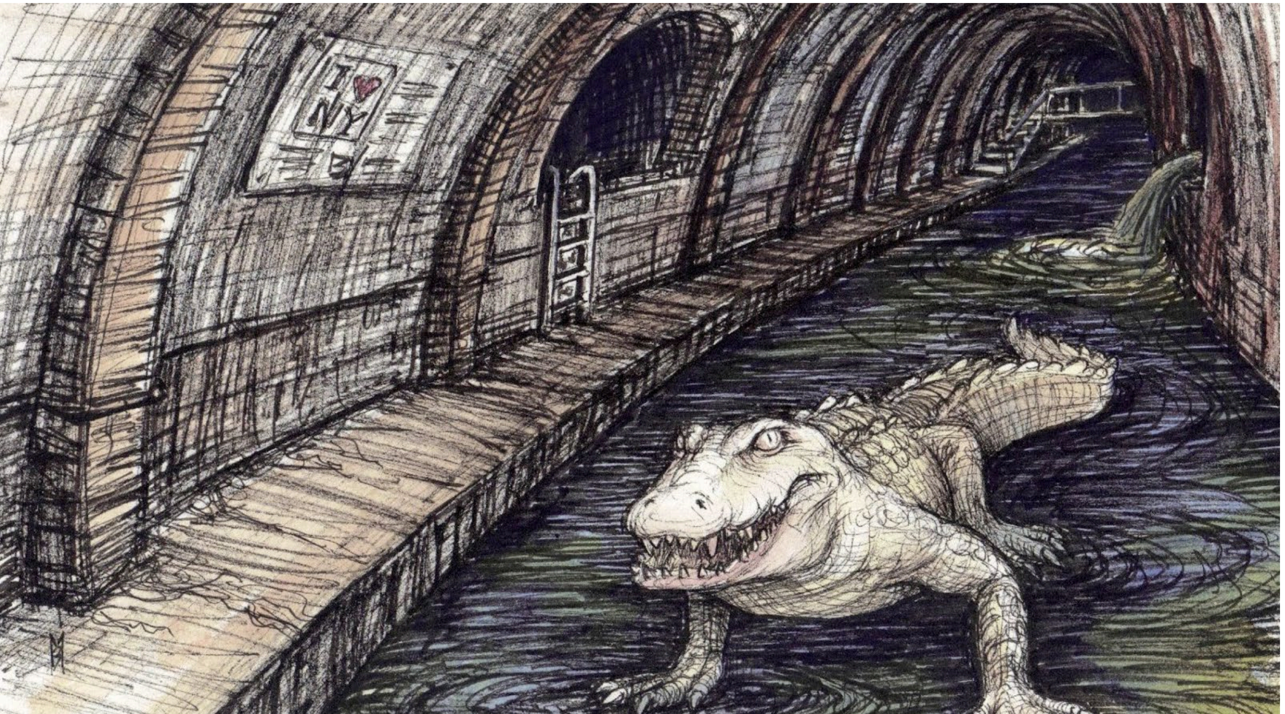
- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list







PROBLEM: FINDING HERBERT

Herbert has gone missing!

Last sighting: in the sewer system.

How can we *systematically* search for Herbert… before he gets destroyed by an alligator?

Use BFS or DFS!

STORING THE PATH

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

How do we augment the algorithms to store the path from s to f?

STORING THE PATH

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                edgeTo[u] = curr
                visit(u)
                Stack.push(u)
    return null
```

How do we augment the algorithms to store the path from s to f?

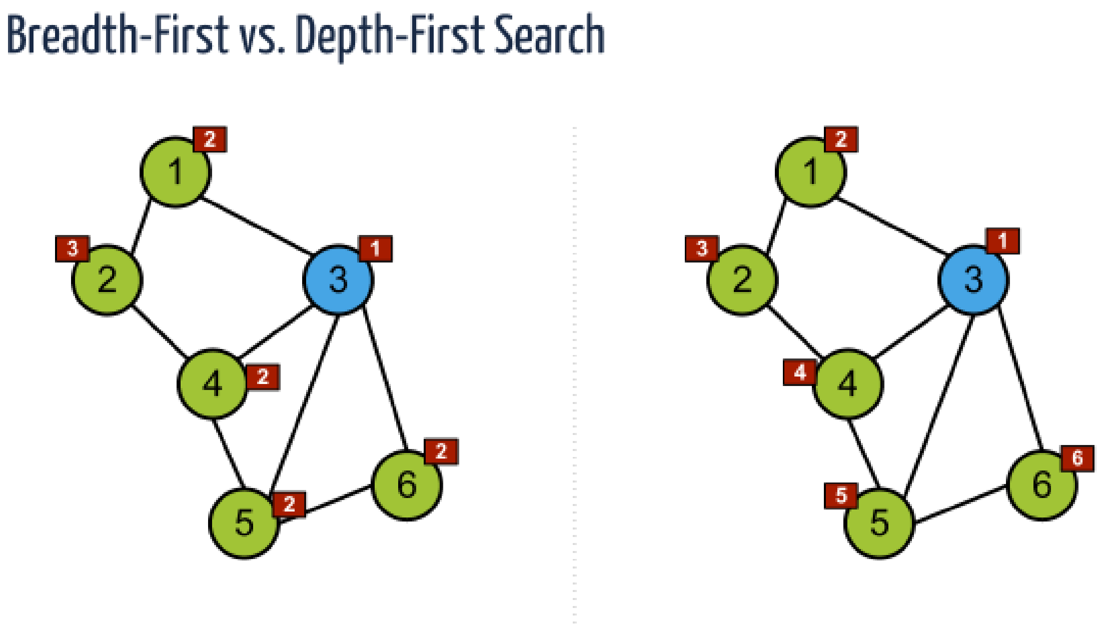
Store a reference to the parent

```
// check that path exists first
// store the path in a list
x = f
while (x != s)
    path.pushFront(x)
    x = edgeTo[x]
path.pushFront(s)
```

Poll Everywhere

https://bit.ly/2LvG9bq

SHORTEST PATHS?



Which finds the shortest path in a simple connected graph?

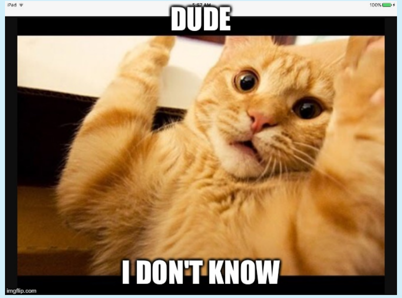
A. BFS

B. DFS

C. Both

D. Neither!

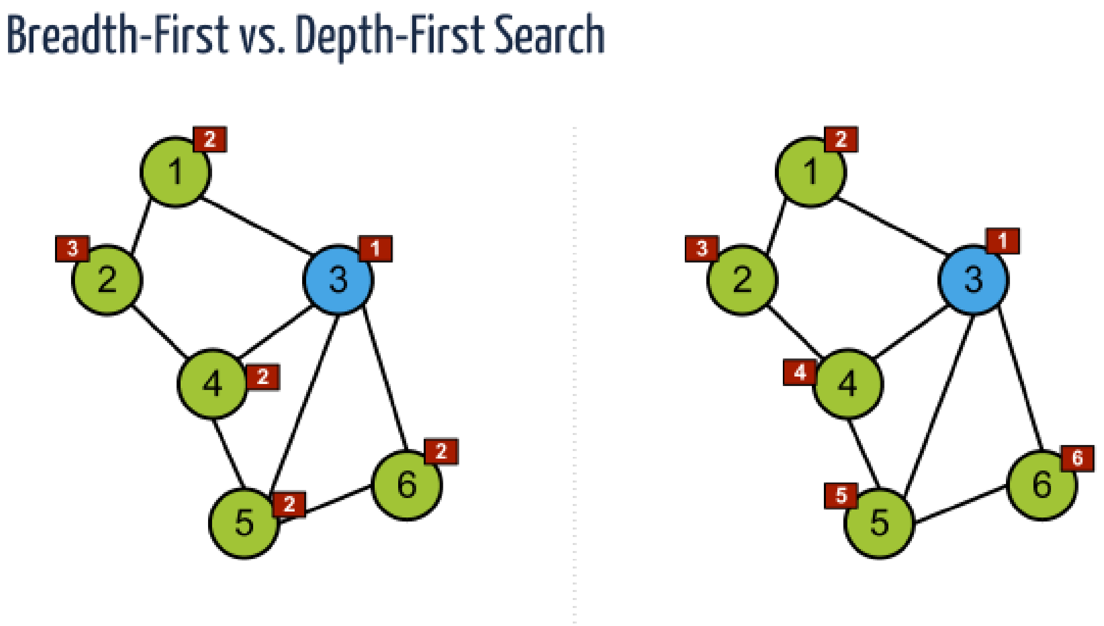
E.



Poll Everywhere

https://bit.ly/2LvG9bq

SHORTEST PATHS?



Which finds the shortest path in a simple connected graph?

A. **BFS**

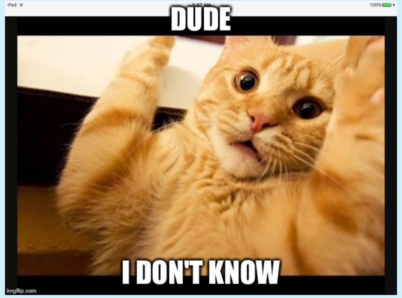
B. DFS

Why?

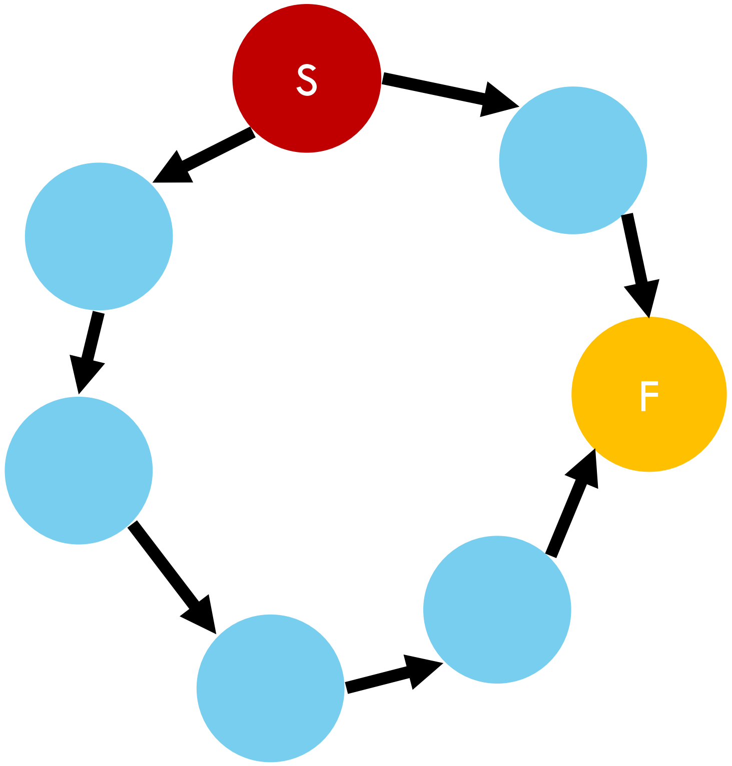
C. Both

D. Neither!

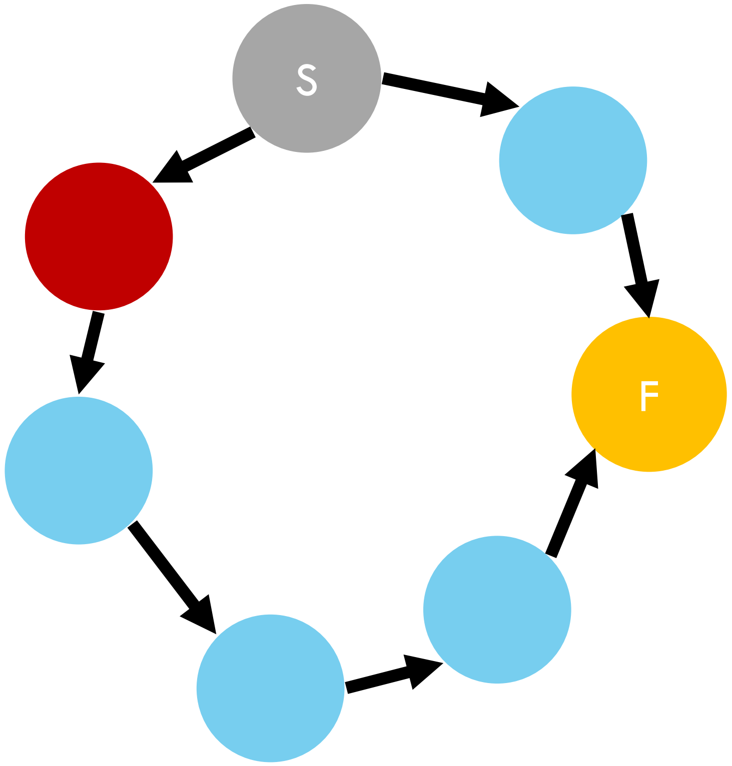
E.



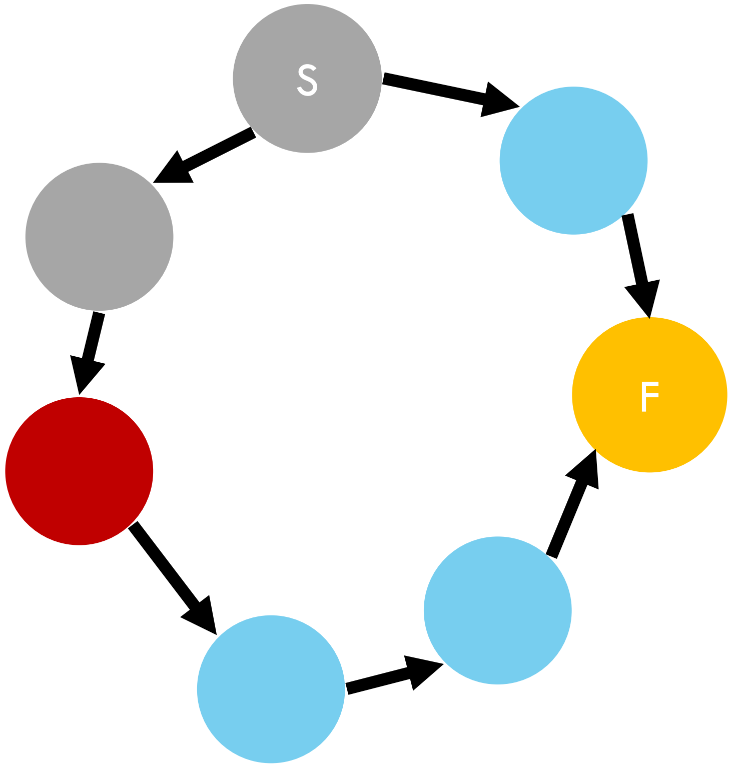
AN EXAMPLE: DFS



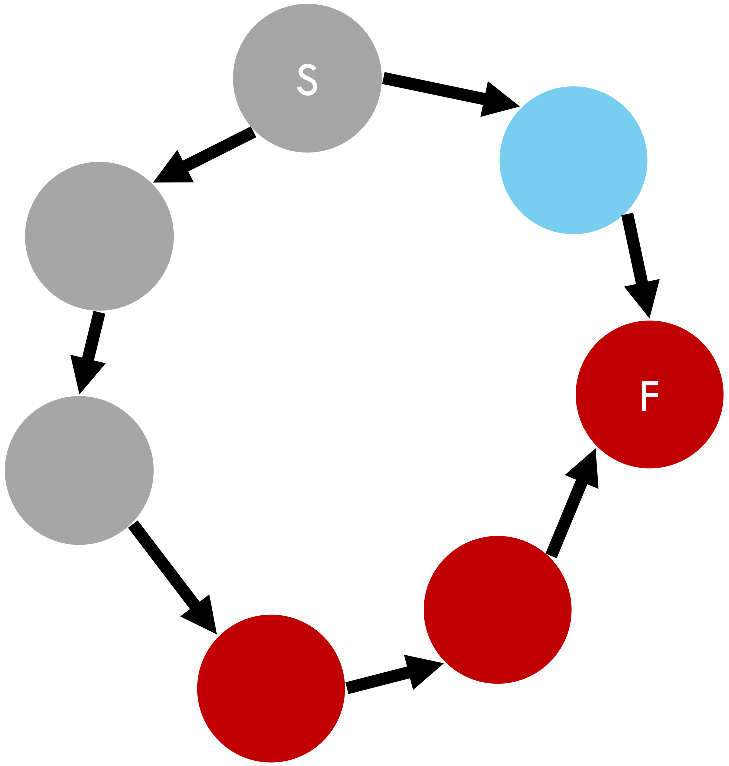
AN EXAMPLE: DFS



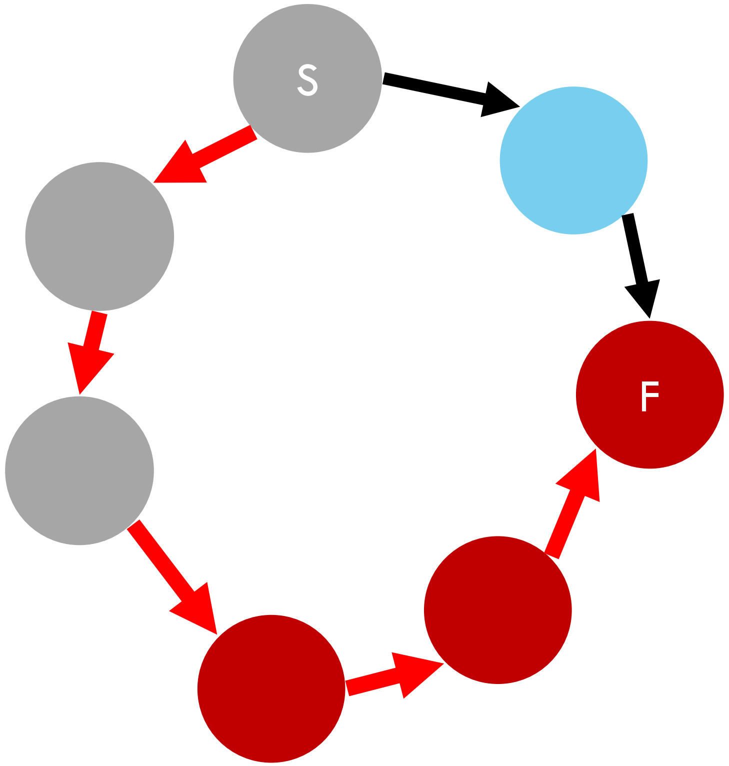
AN EXAMPLE: DFS



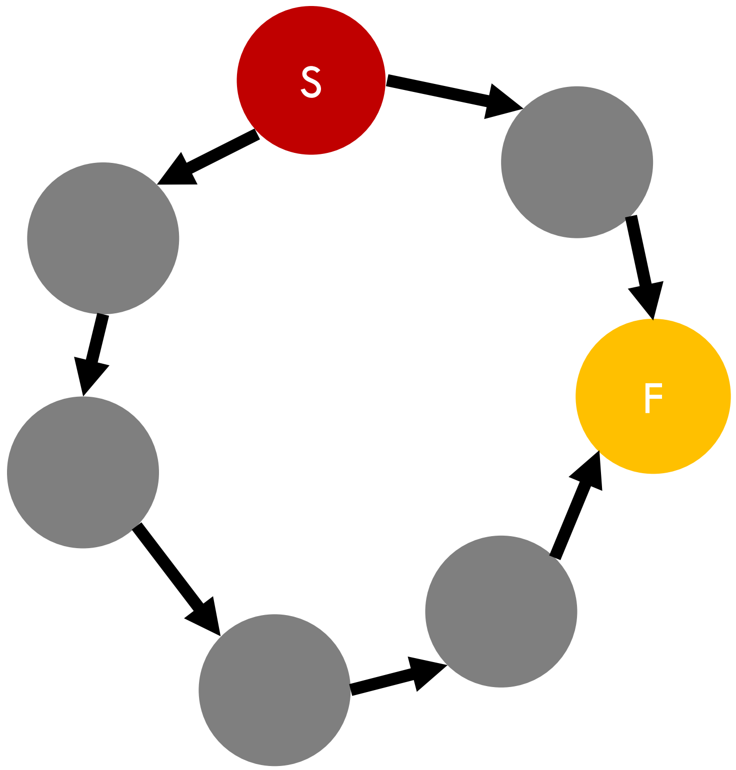
AN EXAMPLE: DFS



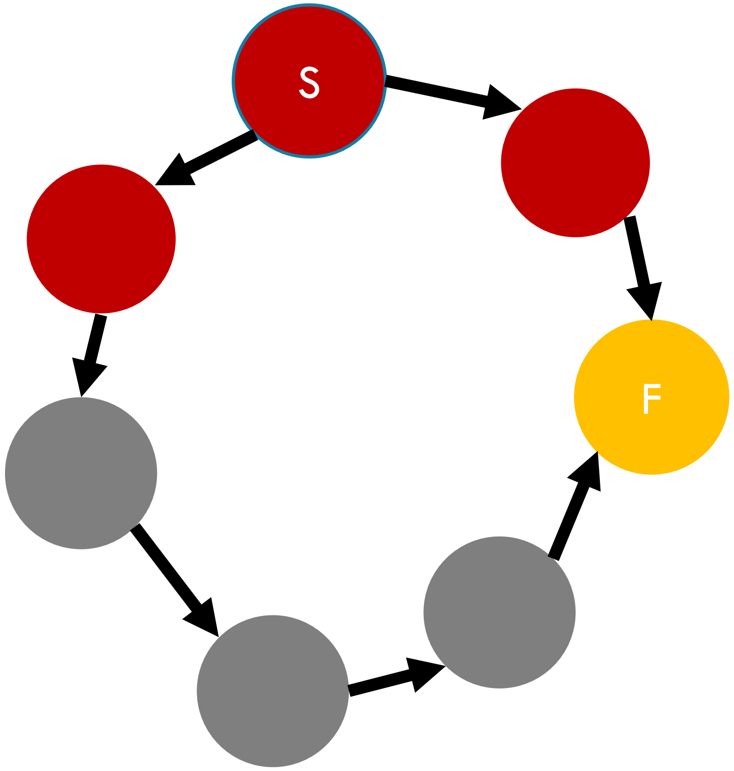
AN EXAMPLE: DFS



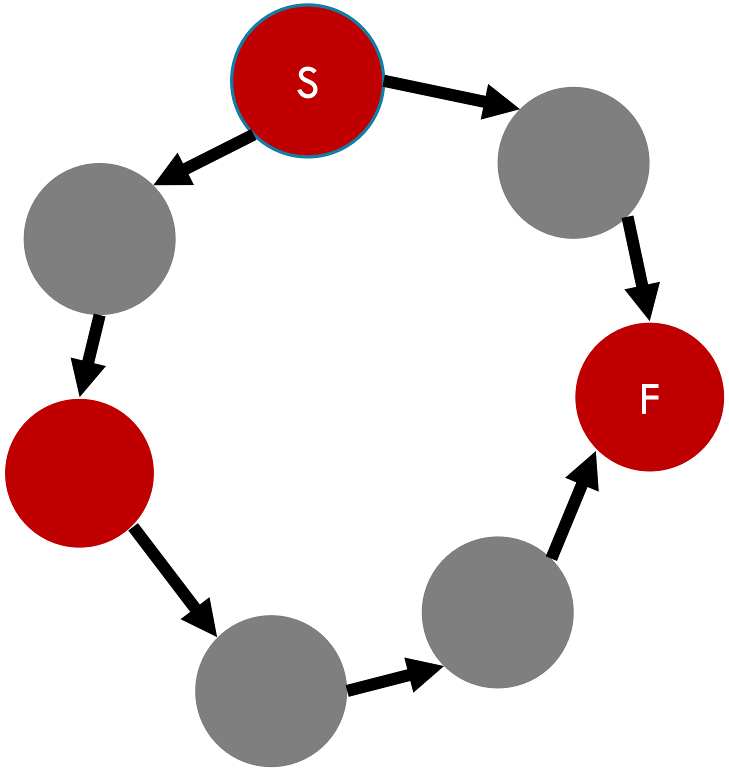
AN EXAMPLE: BFS



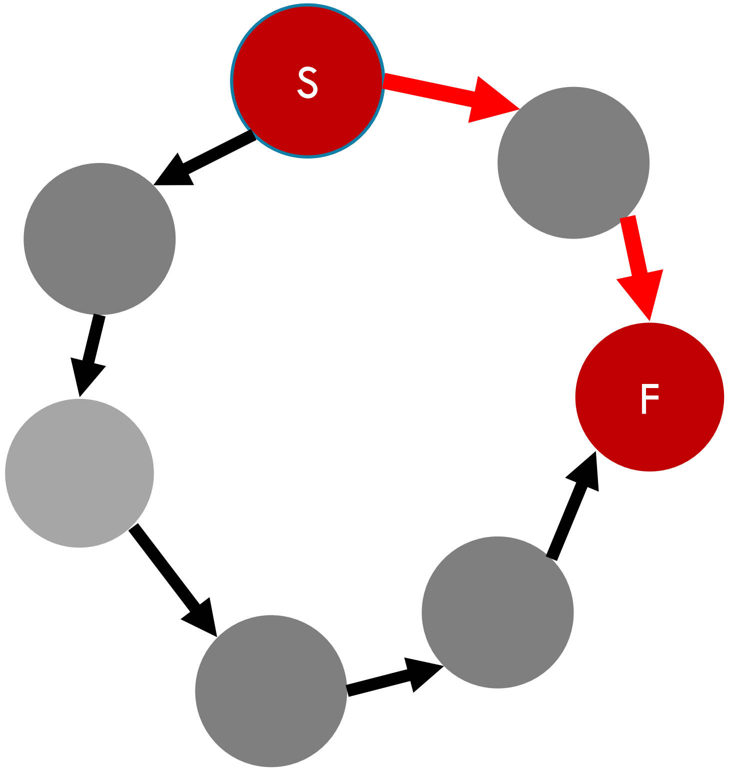
AN EXAMPLE: BFS



AN EXAMPLE: BFS



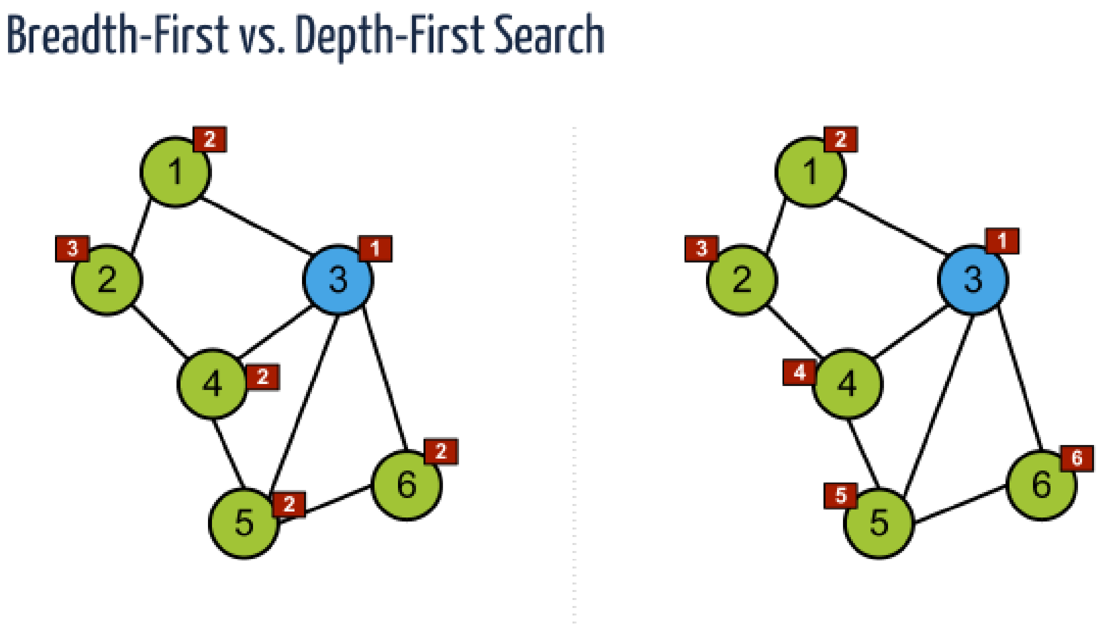
AN EXAMPLE: BFS



Poll Everywhere

https://bit.ly/2LvG9bq

VISITATIONS?



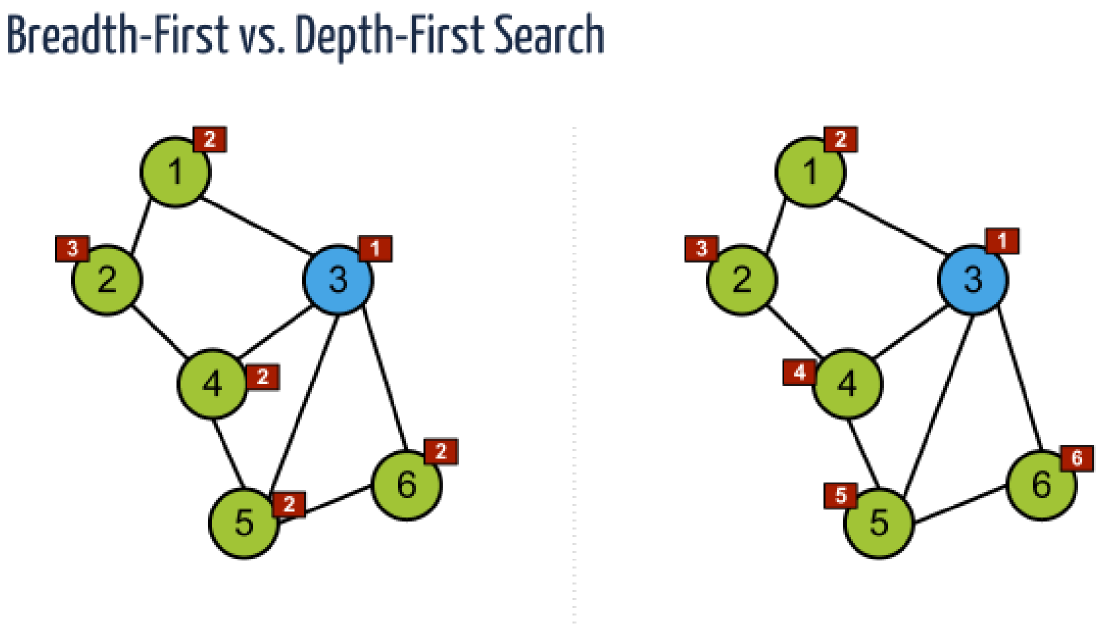
What do BFS and DFS visit?

A. every node
B. every edge
C. every path
D. A & B
E. 

Poll Everywhere

https://bit.ly/2LvG9bq

VISITATIONS?

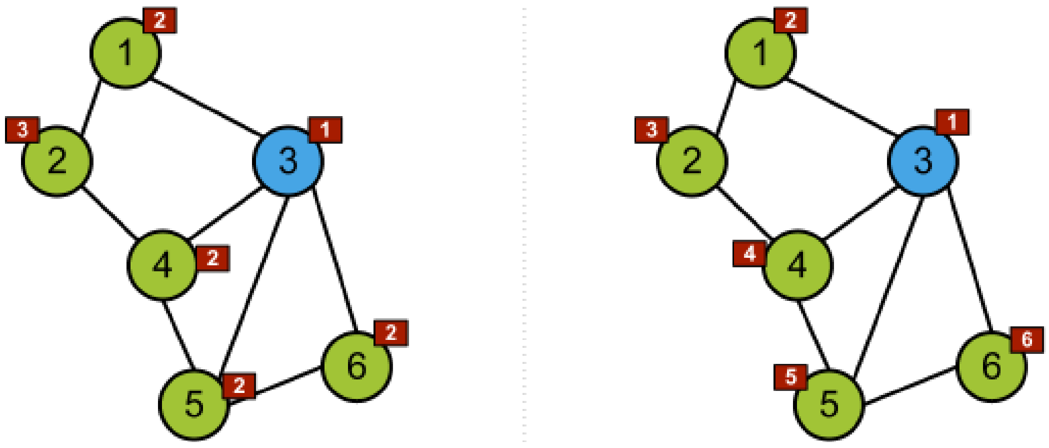


What do BFS and DFS visit?

A. **every node**
B. **every edge**
C. every path
D. **A & B**
E. 

VISITATIONS?

Breadth-First vs. Depth-First Search



What do BFS and DFS visit?

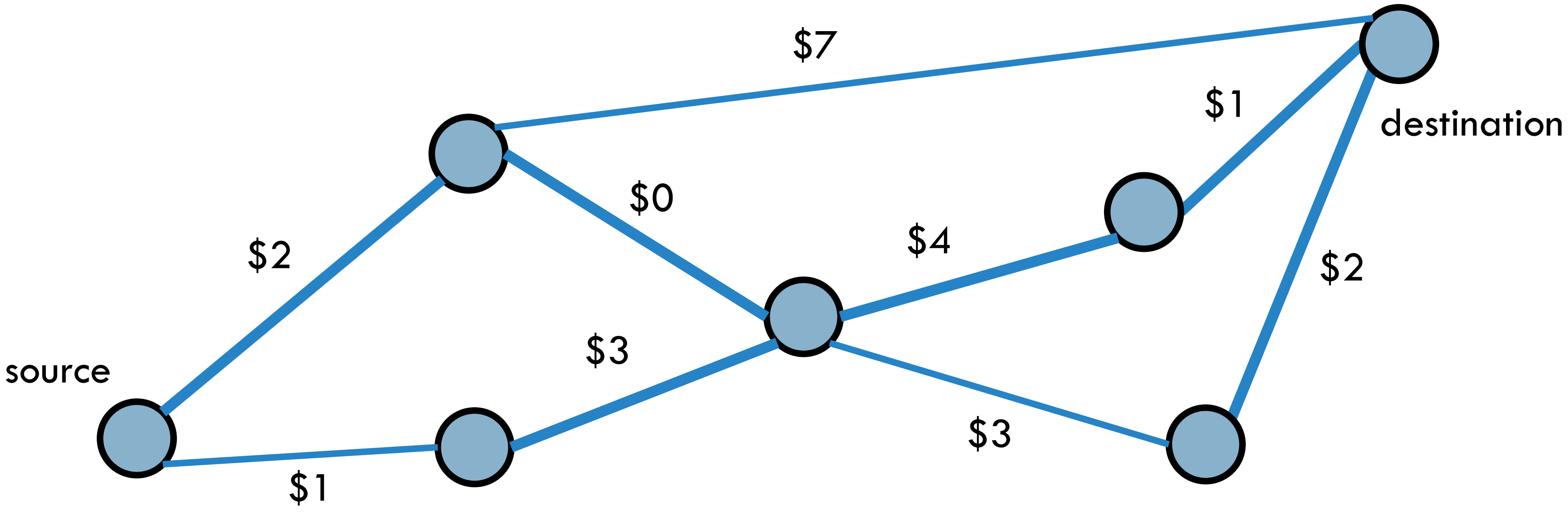
A. every node
B. every edge
C. every path
D. A & B
E. 

common mistake

COMMON MISTAKE WITH BFS/DFS

Problem: Make Money

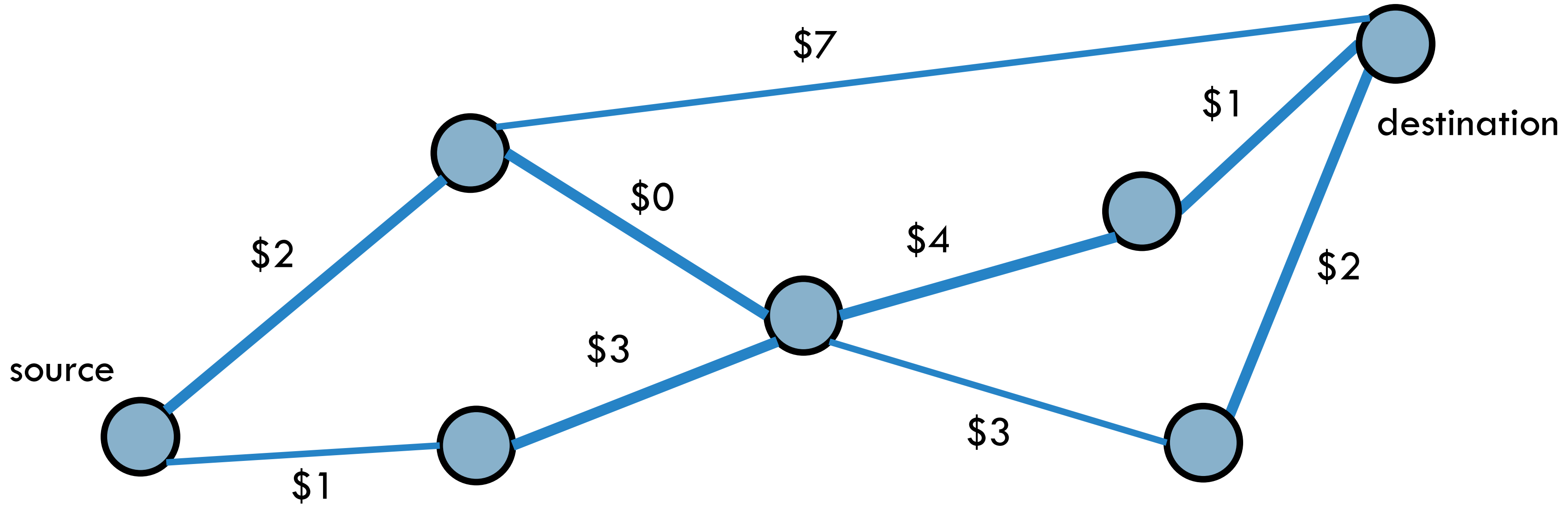
- Start at source s.
- Go to destination d.
- Each edge e earns money m(e).
- Find the path that makes the most money.



COMMON MISTAKE WITH BFS/DFS

NOT a solution:

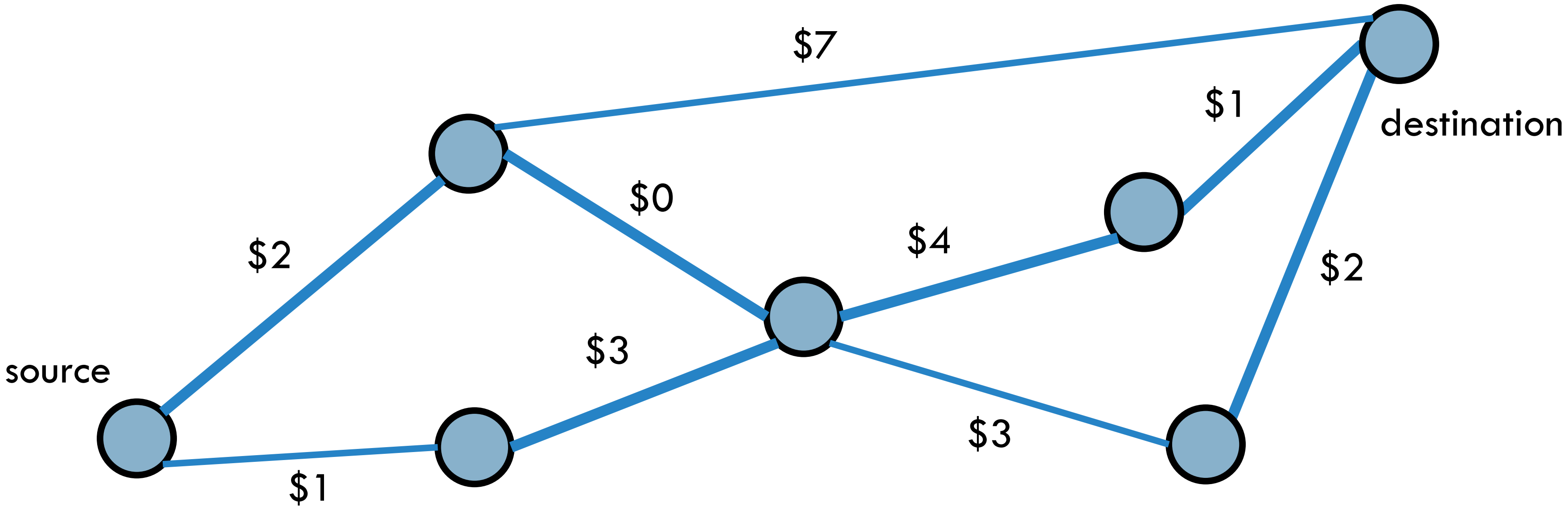
- Start at source s.
- Run BFS or DFS to explore every path.
- Keep track of the best path.



COMMON MISTAKE WITH BFS/DFS

Problem 1: Does not work.

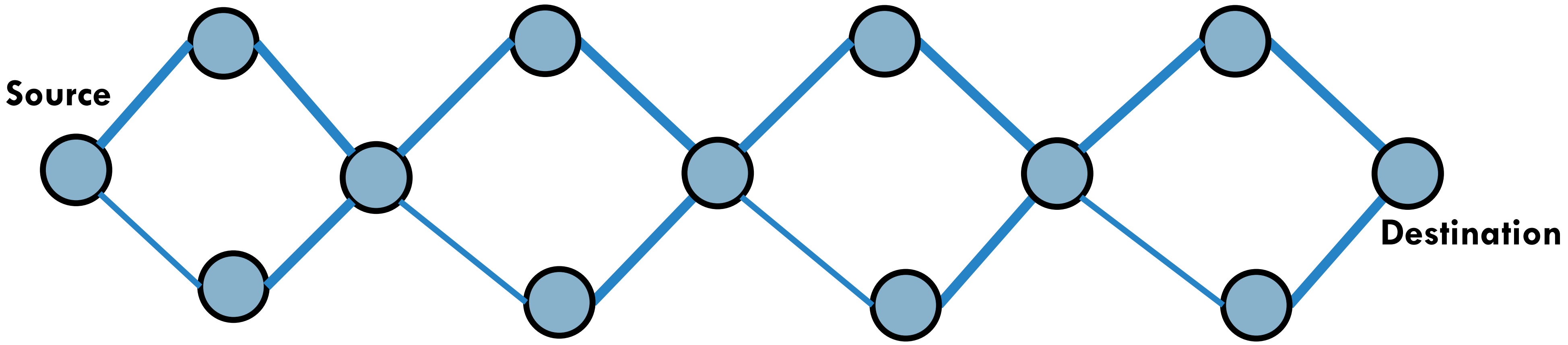
- DFS or BFS do **NOT** explore every path.
- Once a node is visited, it is never explored again.



COMMON MISTAKE WITH BFS/DFS

Problem 2: Too expensive.

- Some graphs have an exponential number of paths.
- It takes exponential time to explore all paths.



Example: $2^4 > 2^{n/4}$ different $s \to d$ paths.

SCHEDULING

Set of tasks for baking cookies:

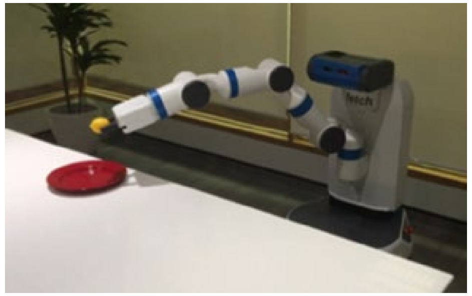
- Shop for groceries
- Put the cookies in the oven
- Clean the kitchen
- Beat the eggs in a bowl
- Measure the flour and sugar in a bowl
- Mix the eggs with the flour and sugar
- Turn on the oven
- Set the timer
- Take out the cookies

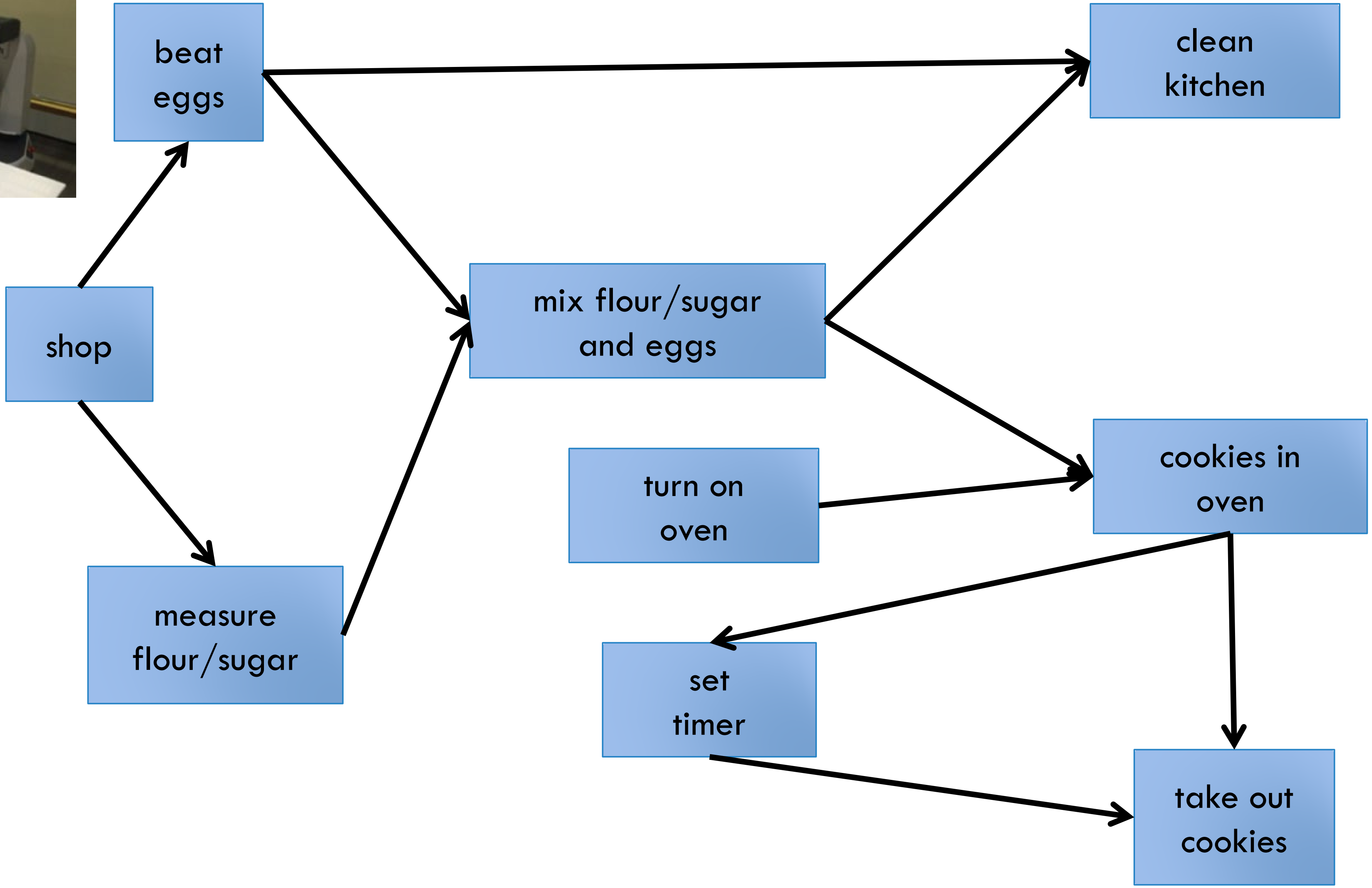
SCHEDULING

Ordering:

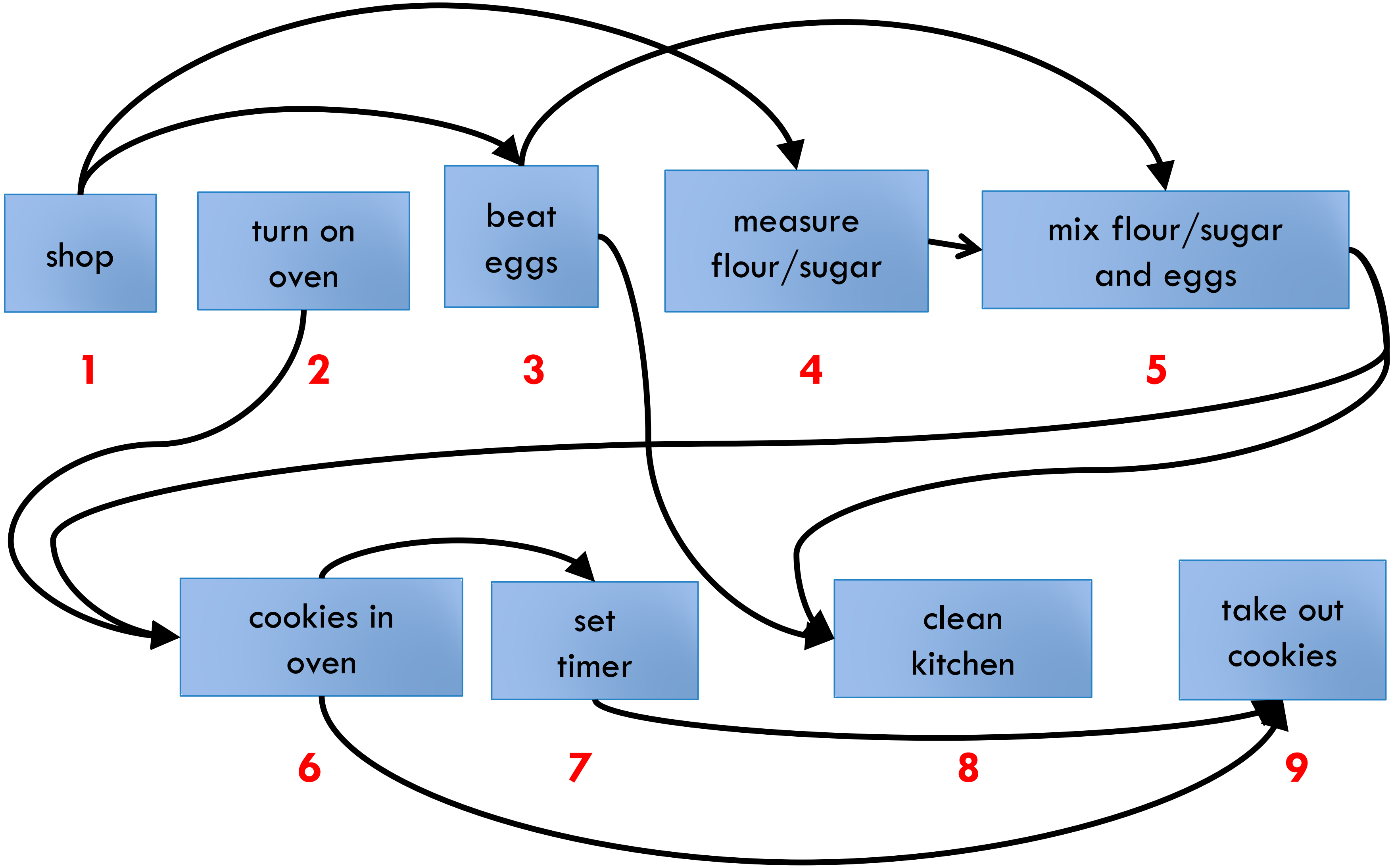
- Shop for groceries before beat the eggs
- Shop for groceries before measure the flour
- Turn on the oven before put the cookies in the oven
- Beat the eggs before mix the eggs with the flour
- Measure the flour before mix the eggs with the flour
- Put the cookies in the oven before set the timer
- Measure the flour before clean the kitchen
- Beat the eggs before clean the kitchen
- Mix the flour and the eggs before clean the kitchen

How do I find the sequence of tasks I should perform?





TOPOLOGICAL ORDERING



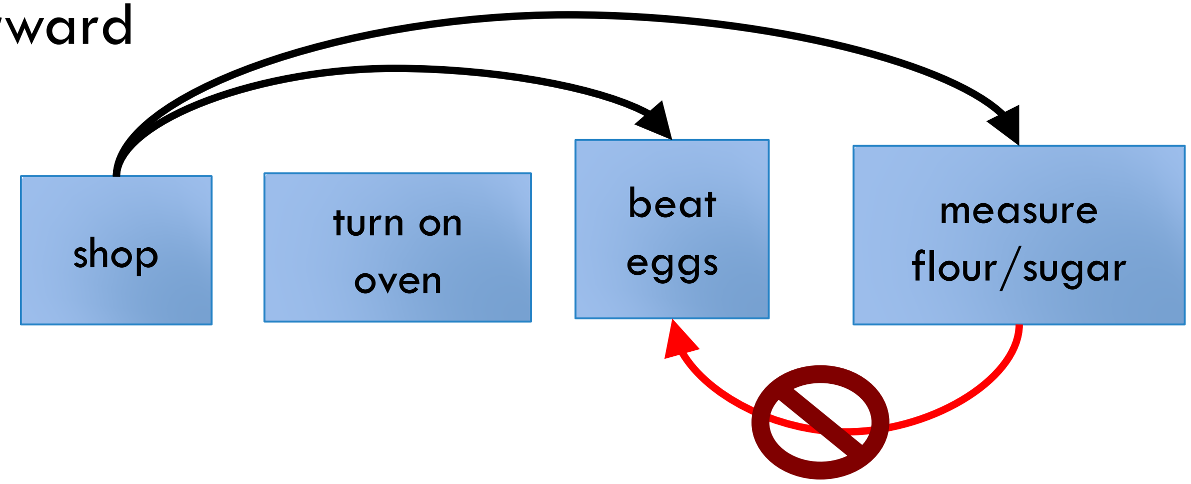
TOPOLOGICAL ORDER

Properties:

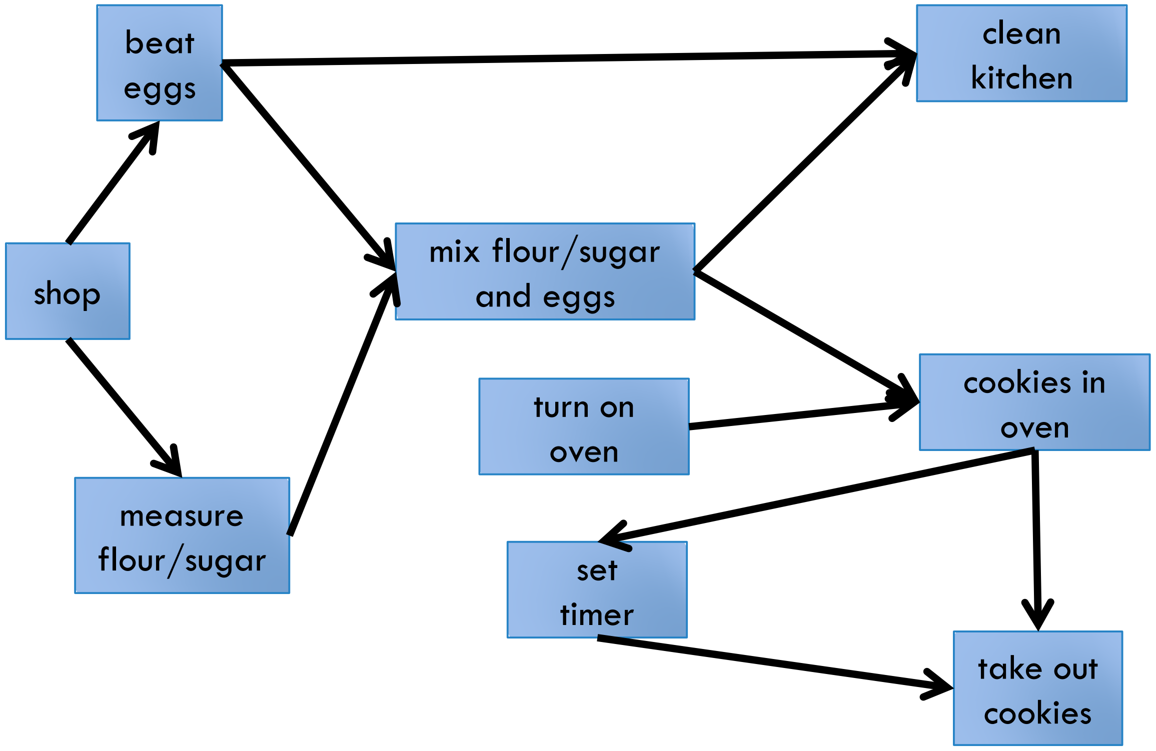
1. Sequential total ordering of all nodes

1. shop	2. turn on oven	3. measure flour/sugar	4. eggs

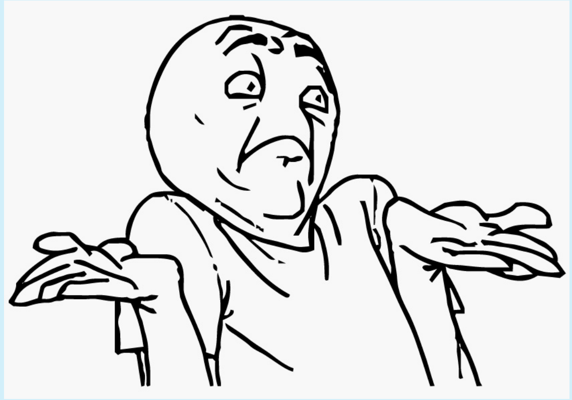
2. Edges only point forward



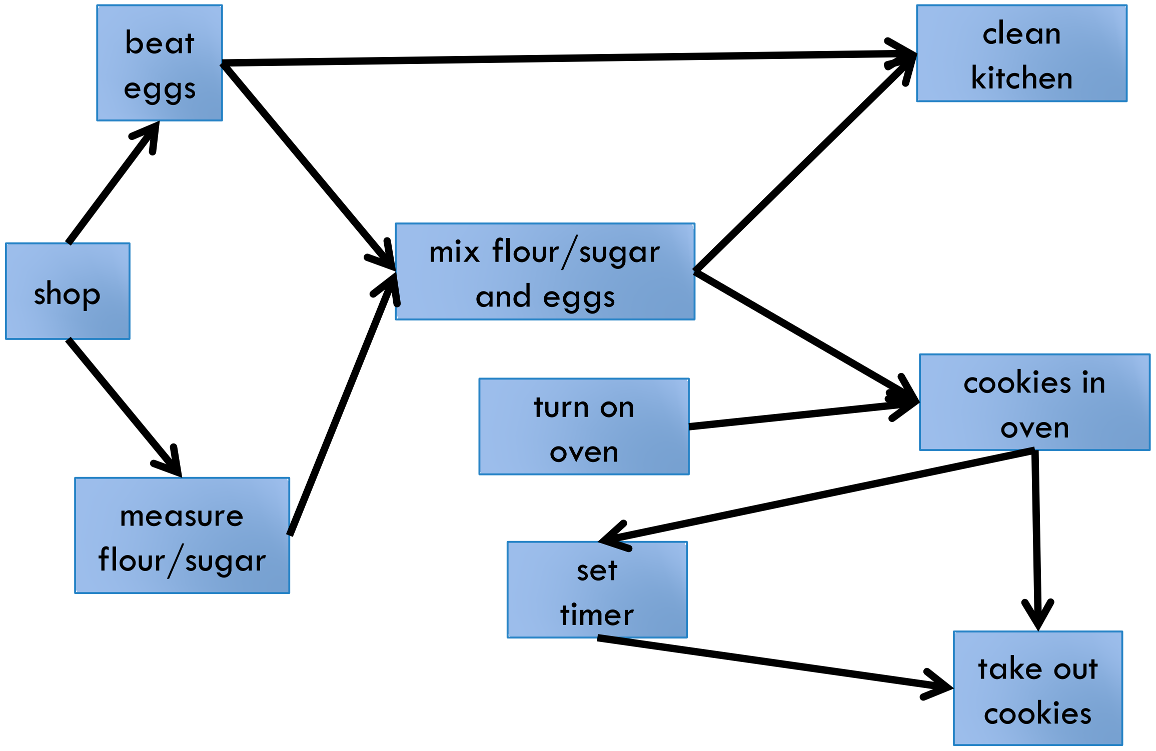
TOPOLOGICAL ORDERING



Does every directed graph have a topological ordering?

A. Yes!
B. No!
C. 

TOPOLOGICAL ORDERING

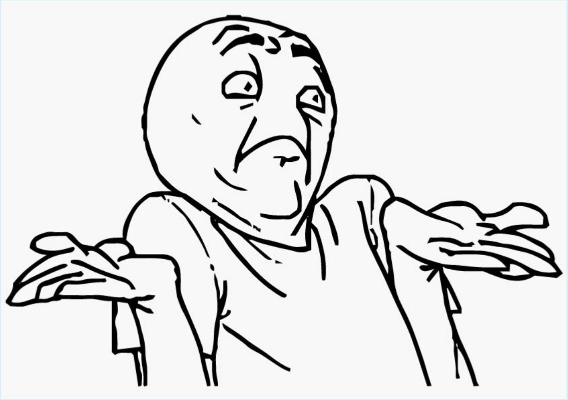


Does every directed graph have a topological ordering?

A. Yes!

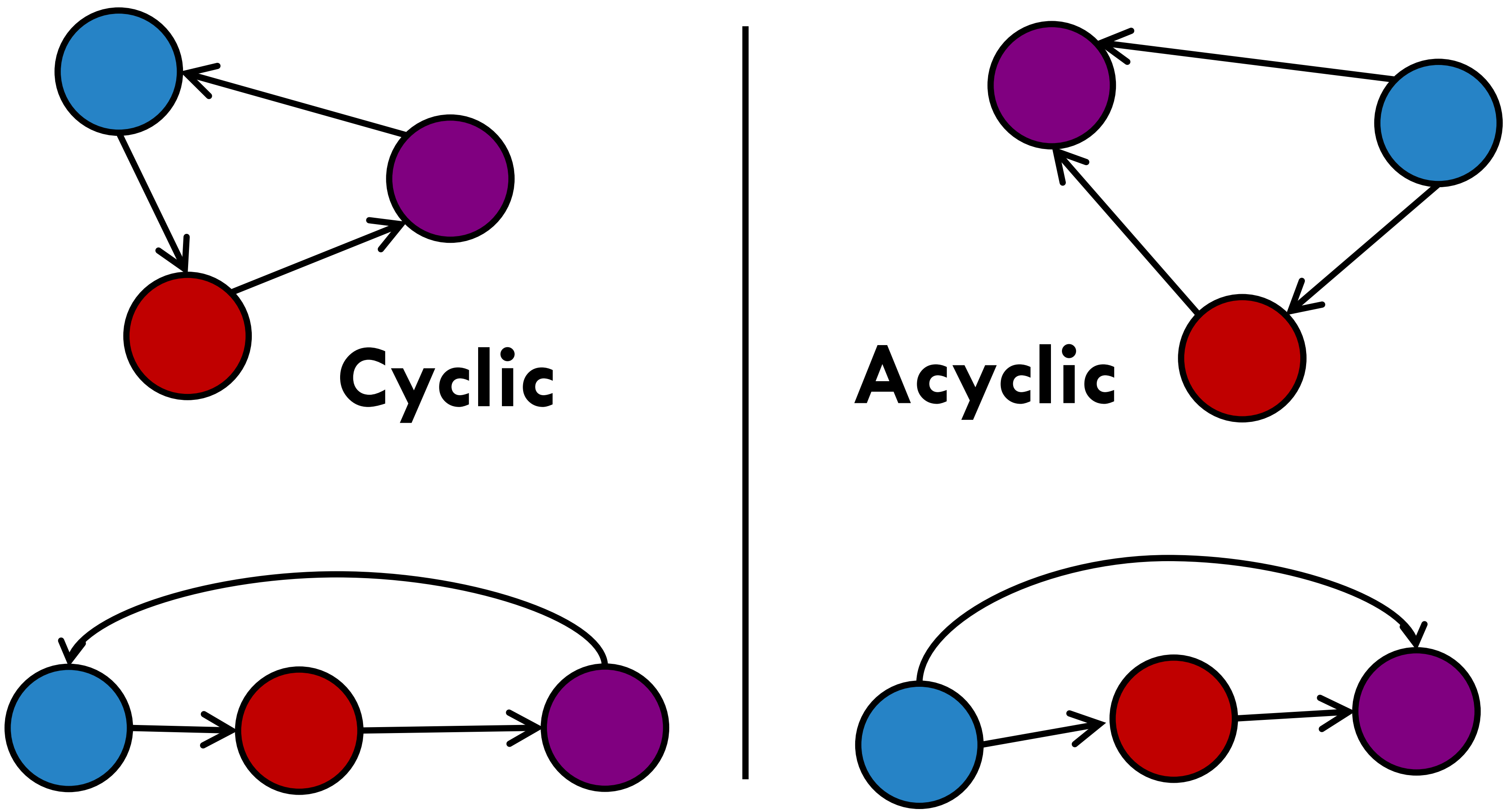
B. No!

C.



Why?

DIRECTED ACYCLIC GRAPHS (DAG)

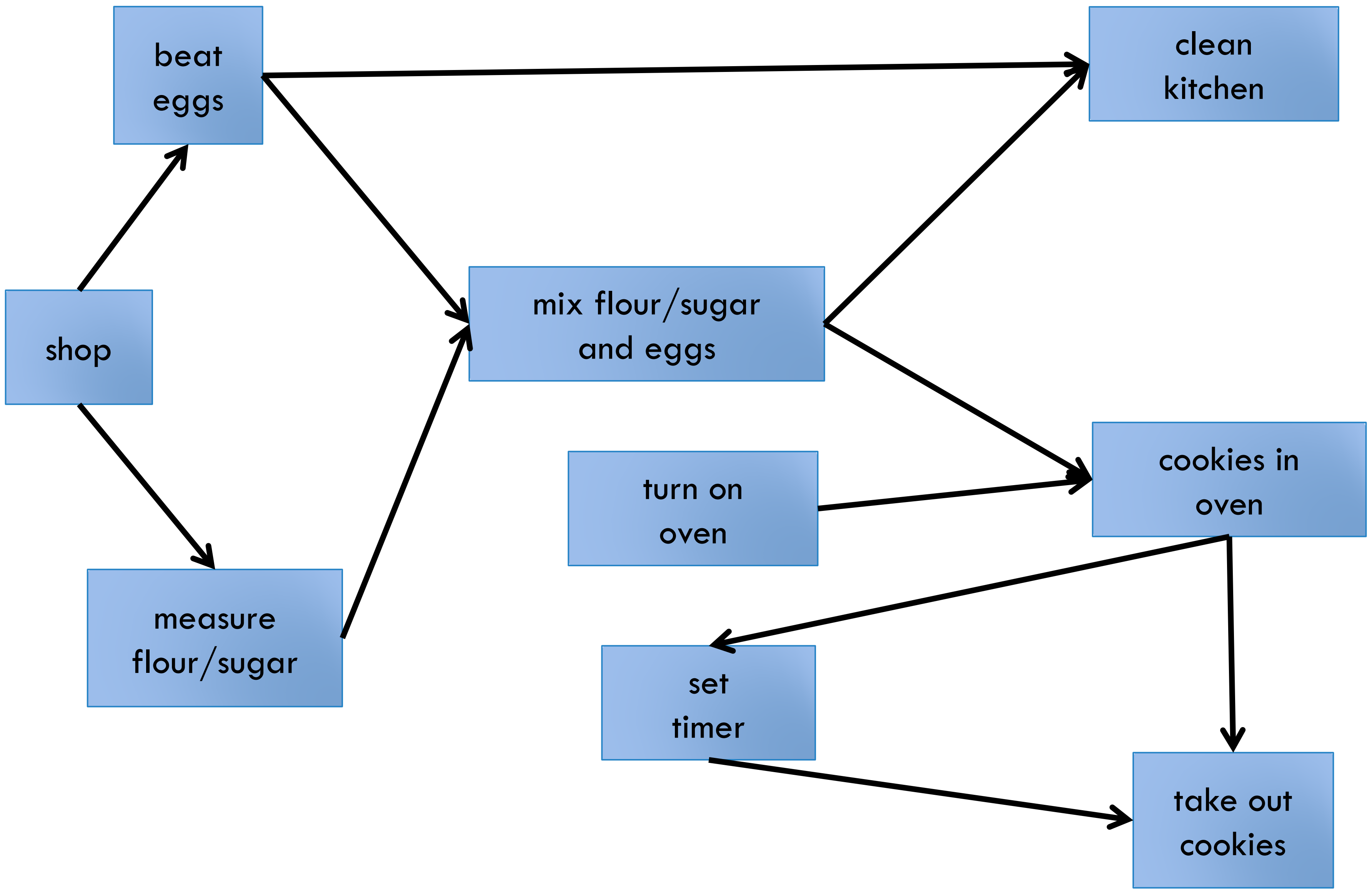


DAG

A topological ordering is only possible iff the graph is a DAG!

DAG = Directed Acyclic Graph

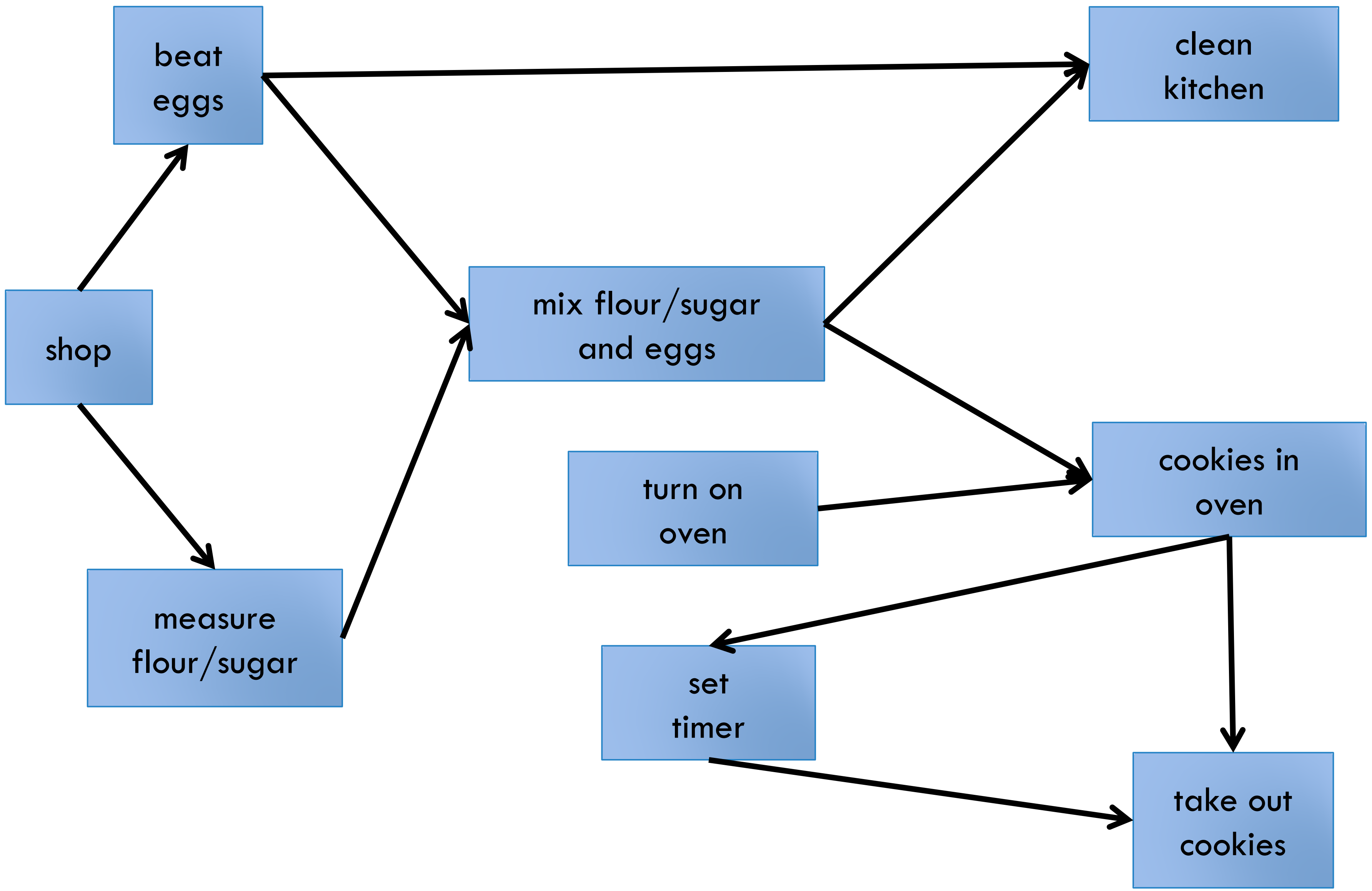
Problem: How do I find the sequence of tasks I should perform?



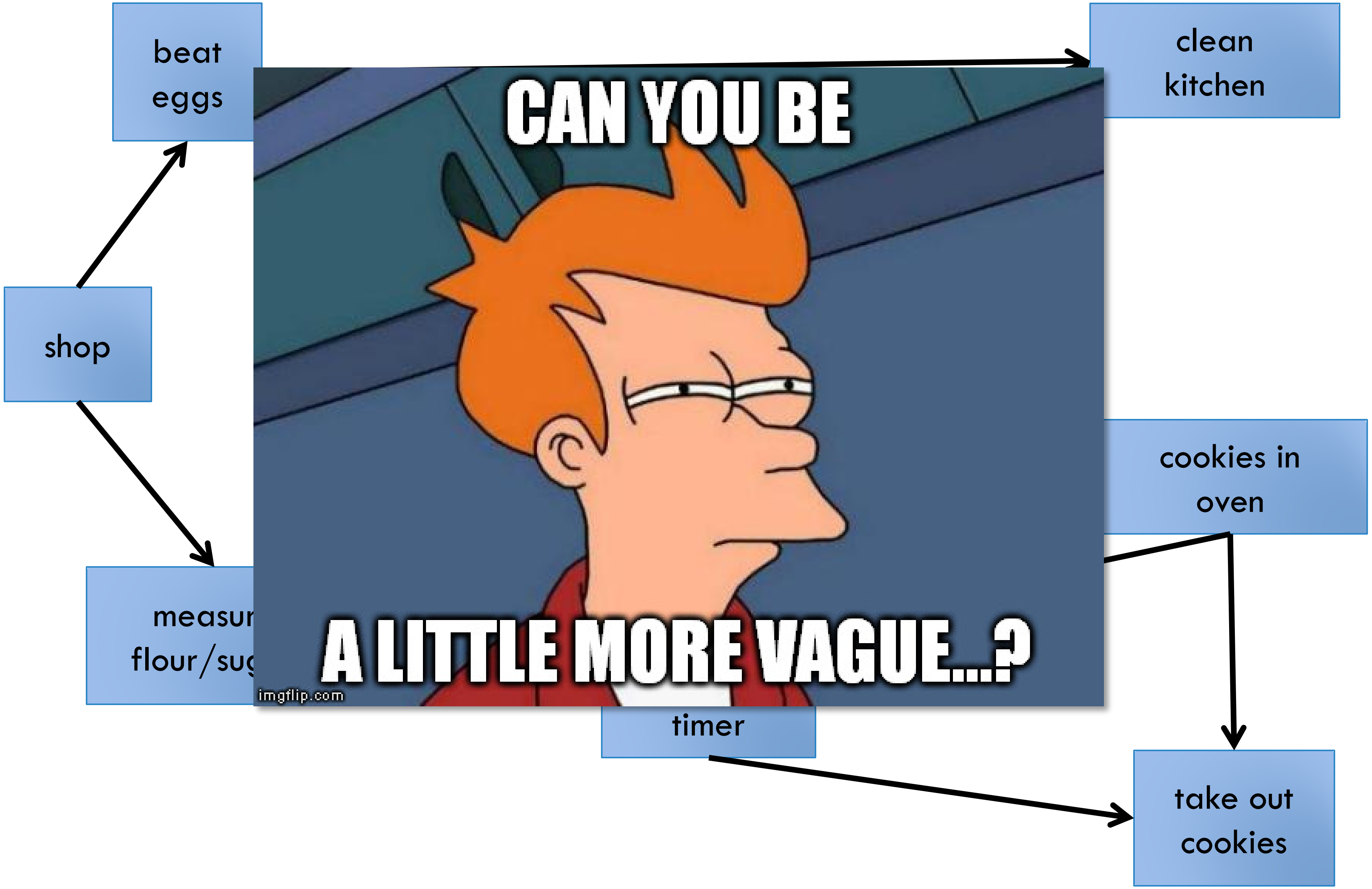
SAMPLE PROBLEM: TOPOLOGICAL ORDERING

Harold Soh
harold@comp.nus.edu.sg

Problem: How do I find the sequence of tasks I should perform?



Problem: How do I find the sequence of tasks I should perform?



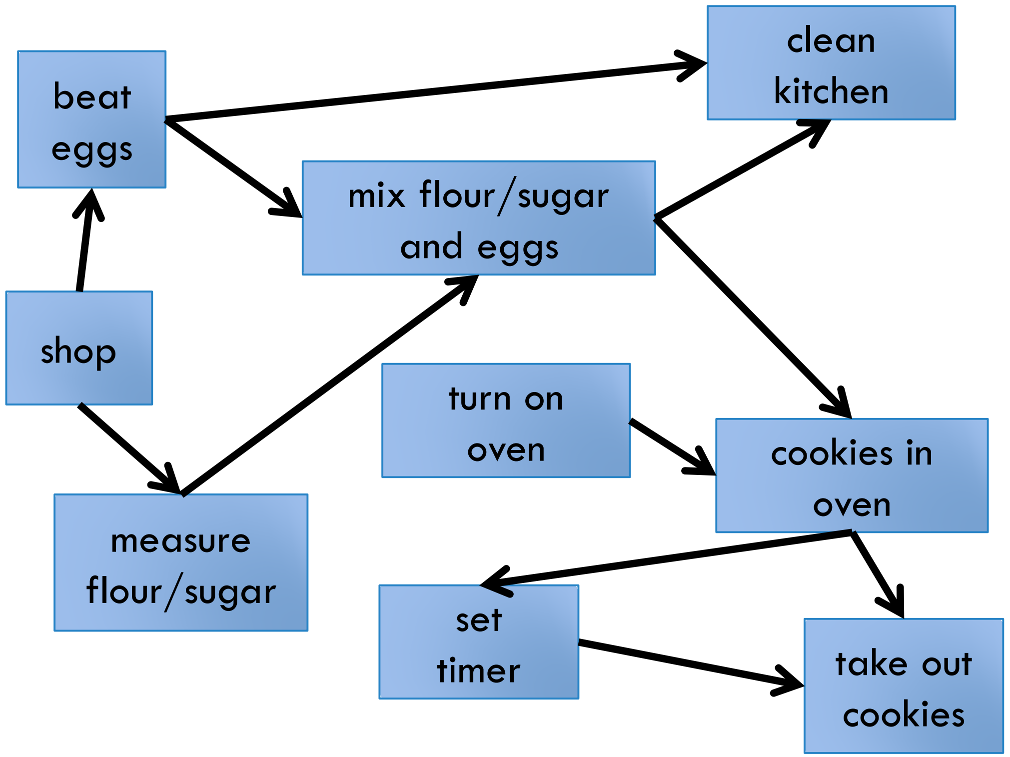
PROBLEM SPECIFICATION: INPUTS & OUTPUTS

Input(s):

- Input is a graph. Any graph?
- A DAG!
- Represented as a?
- Adjacency list

Output(s):

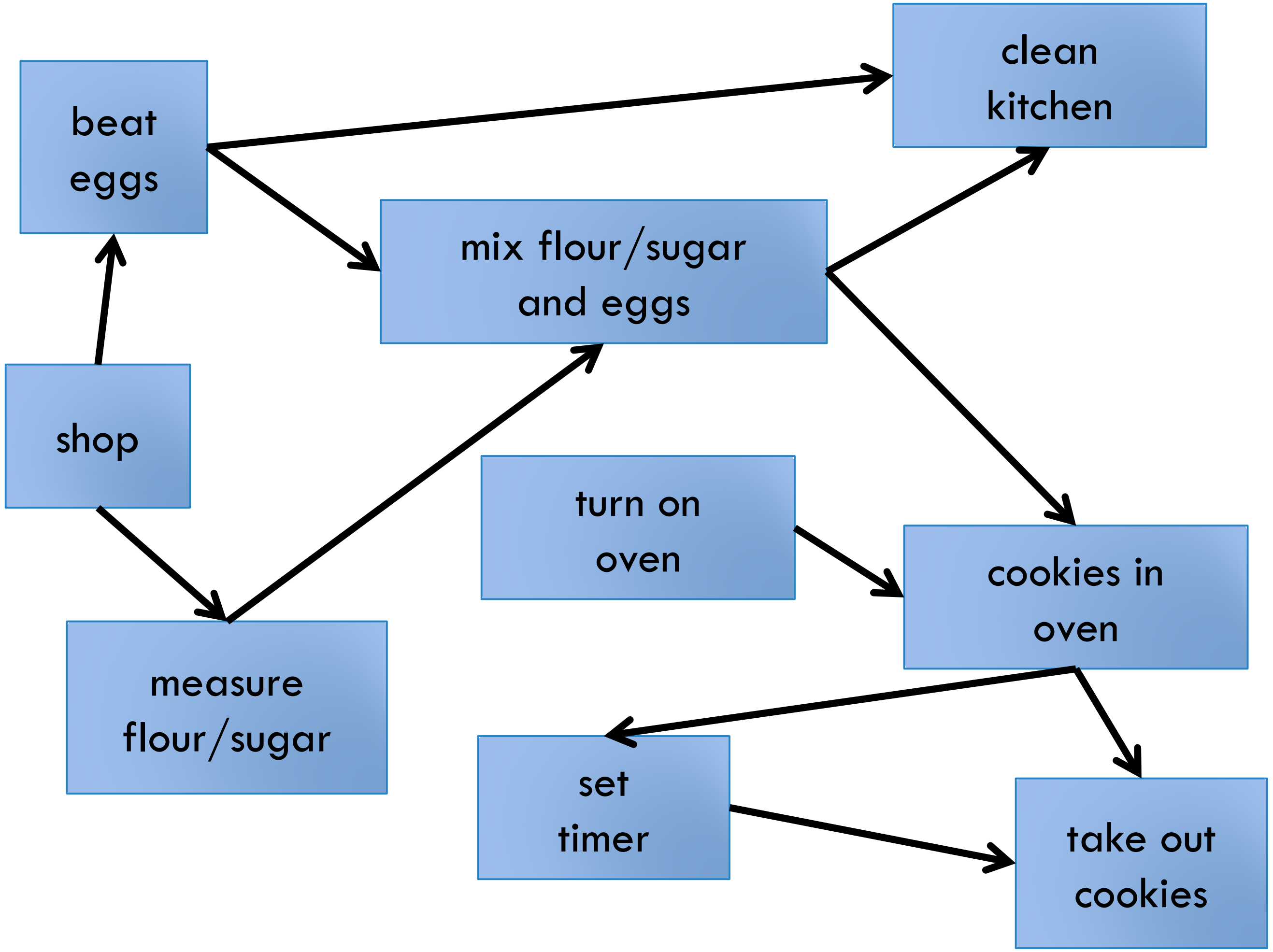
- A list of nodes in topological order.
  - No node in the list can have an incoming edge from a node that appears later (in the list).



TOPOLOGICAL ORDERING

Idea:

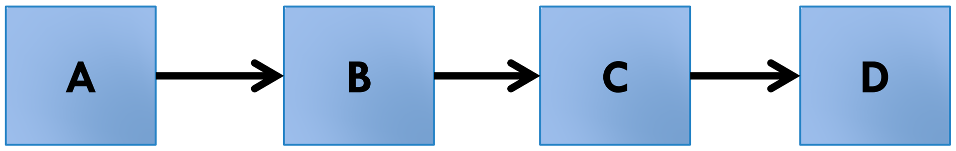
- What is the most straightforward approach?
- Where is a good place to start?
- Graph is complicated. **Let's simplify.**



TOPOLOGICAL ORDERING

Idea:

- What is the most straightforward approach?
- Where is a good place to start?
- Graph is complicated. **Let's simplify.**



IDEA

Where should I start?

Start at a node v with no incoming edges.

where to go next?

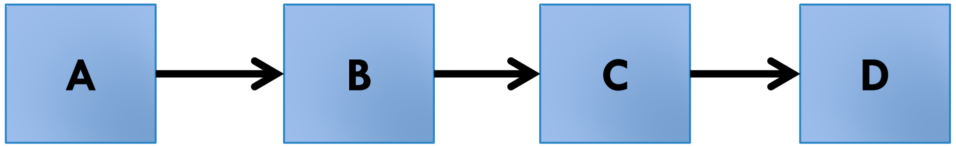
a node x that does not depend on any other node except v

How can we find this node?

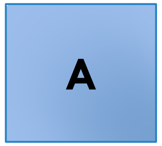
Remove all out-going edges from v and find node with no incoming edges!

What should I do next?

Repeat!



Ordered List:



IDEA

Where should I start?

Start at a node v with no incoming edges.

where to go next?

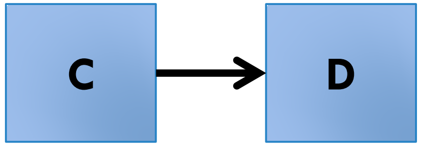
a node x that does not depend on any other node except v

How can we find this node?

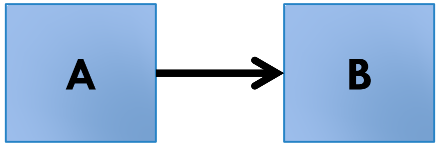
Remove all out-going edges from v and find node with no incoming edges!

What should I do next?

Repeat!



Ordered List:



IDEA

Where should I start?

Start at a node v with no incoming edges.

where to go next?

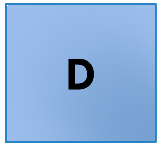
a node x that does not depend on any other node except v

How can we find this node?

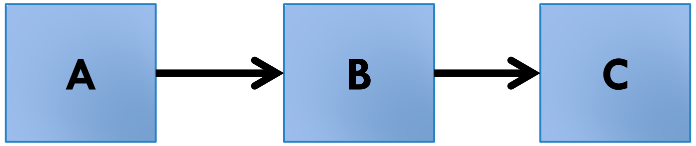
Remove all out-going edges from v and find node with no incoming edges!

What should I do next?

Repeat!



Ordered List:



IDEA

Where should I start?

Start at a node v with no incoming edges.

where to go next?

a node x that does not depend on any other node except v

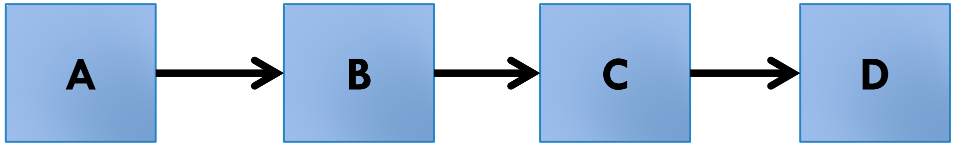
How can we find this node?

Remove all out-going edges from v and find node with no incoming edges!

What should I do next?

Repeat!

Ordered List:



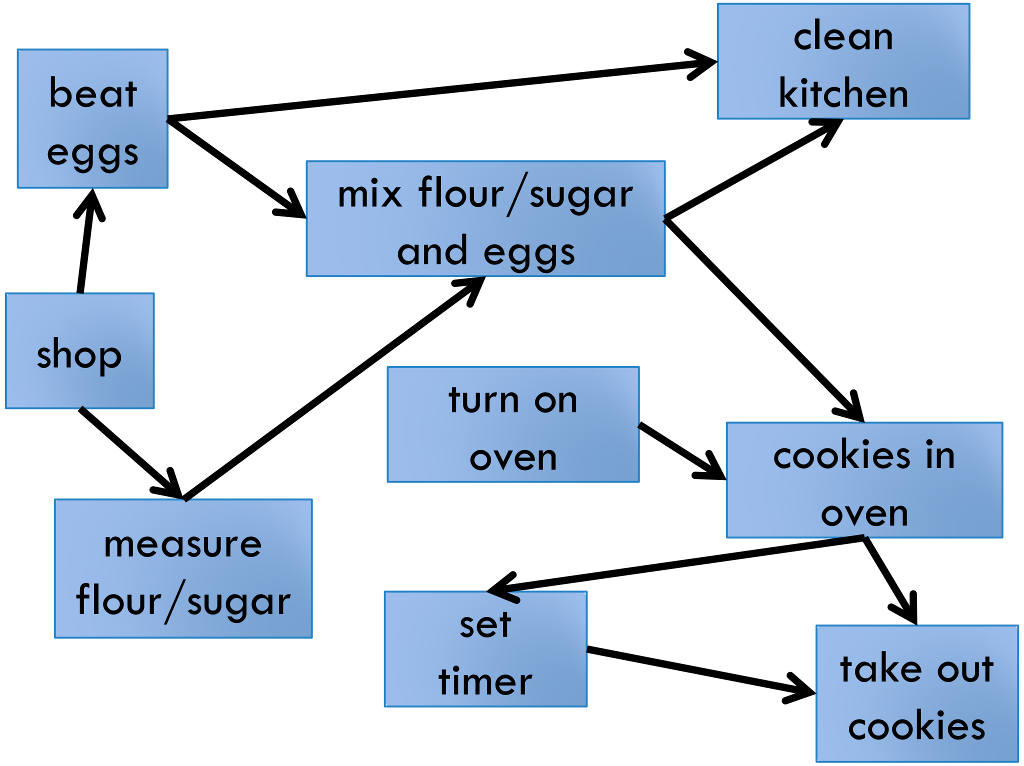
IDEA

Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat



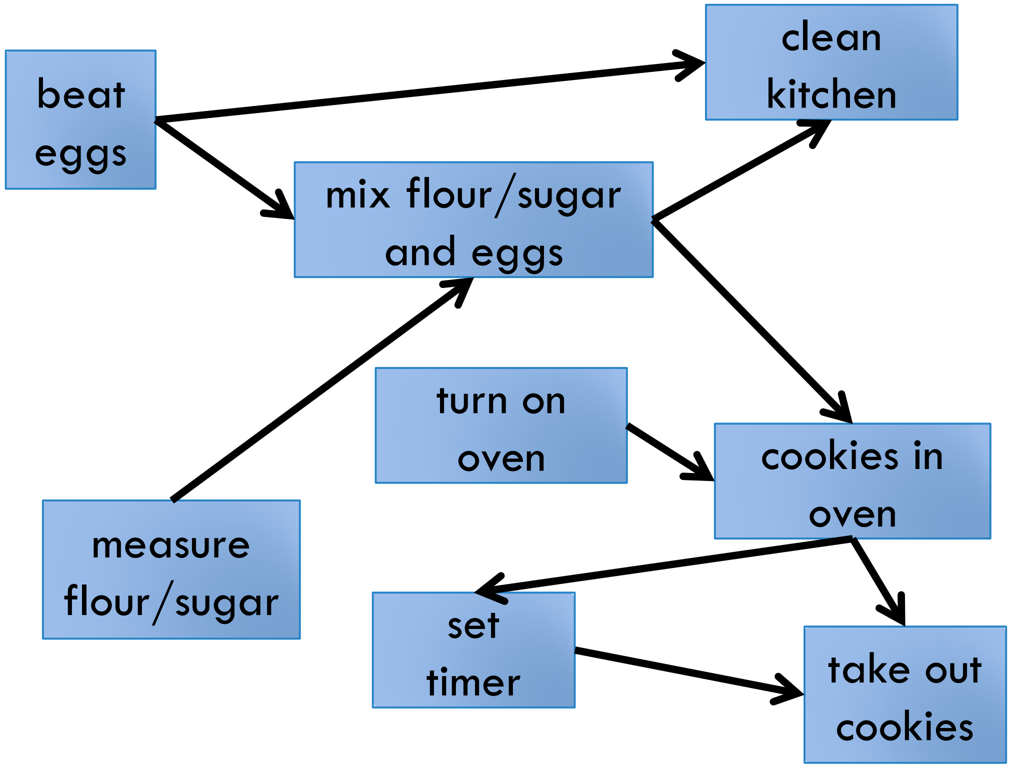
IDEA

Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat



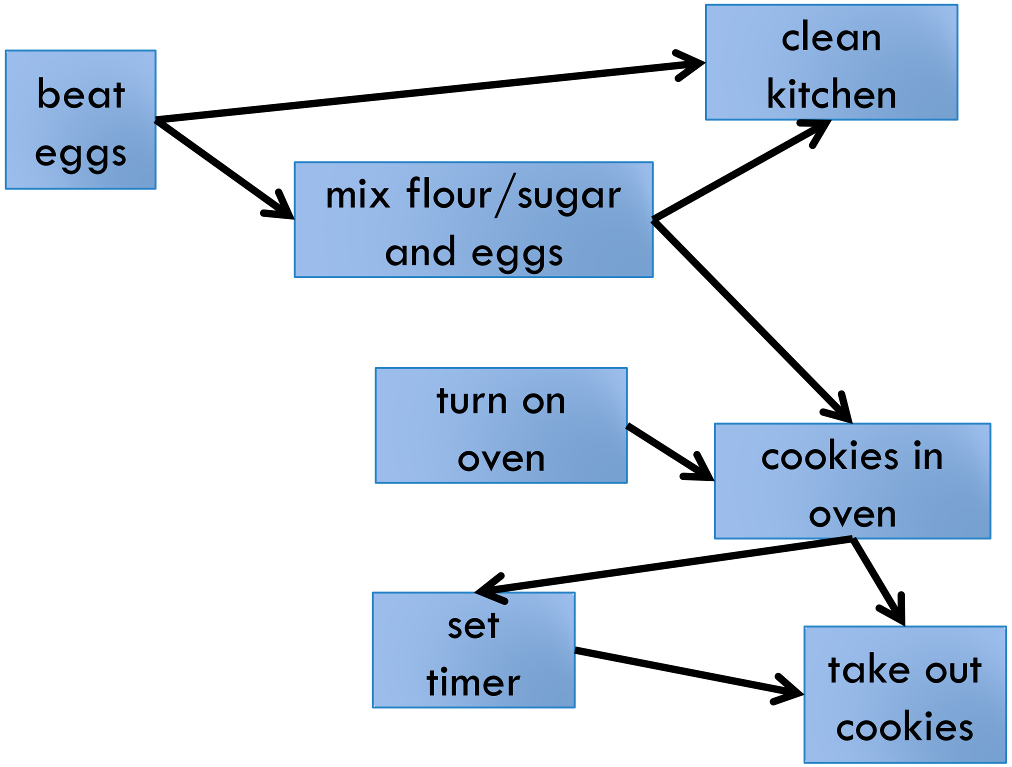
IDEA

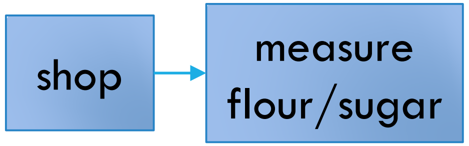
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





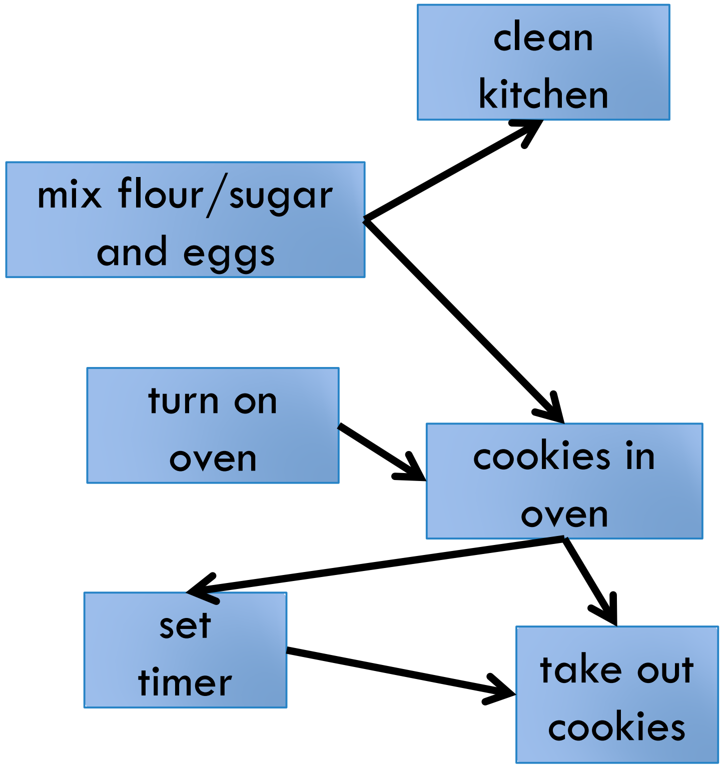
IDEA

Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat



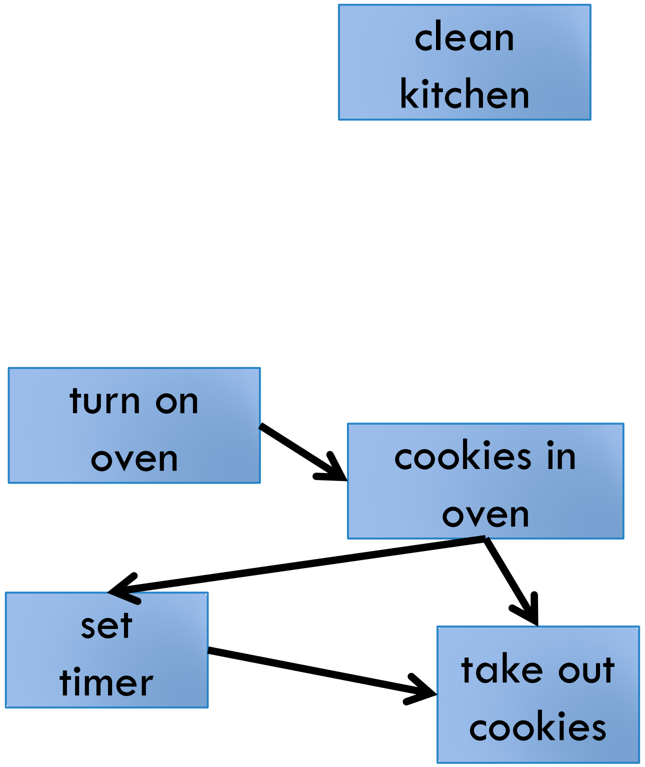
IDEA

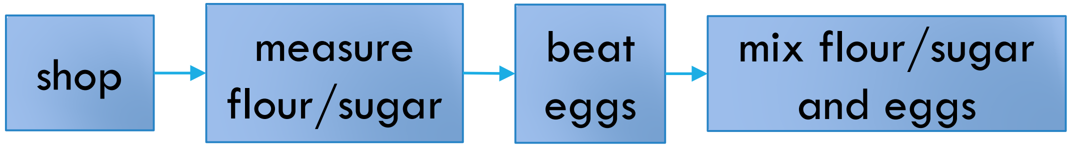
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





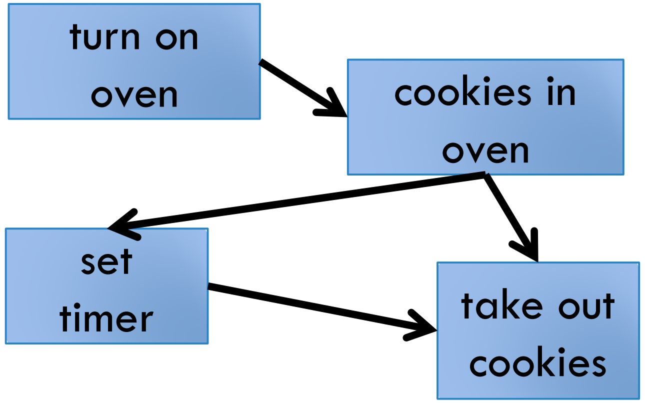
IDEA

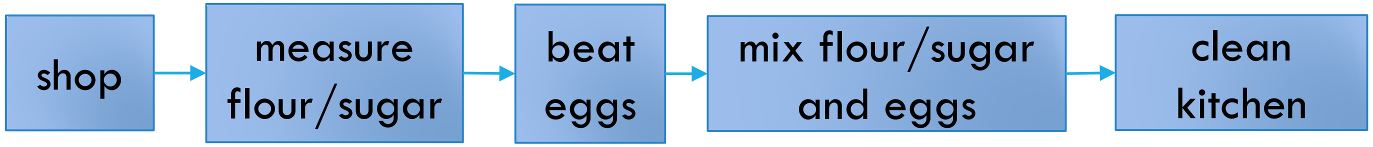
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





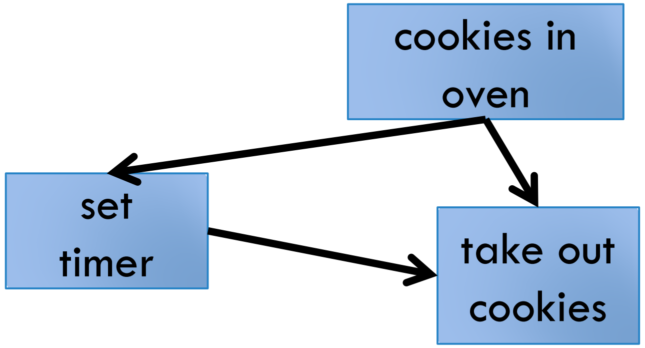
IDEA

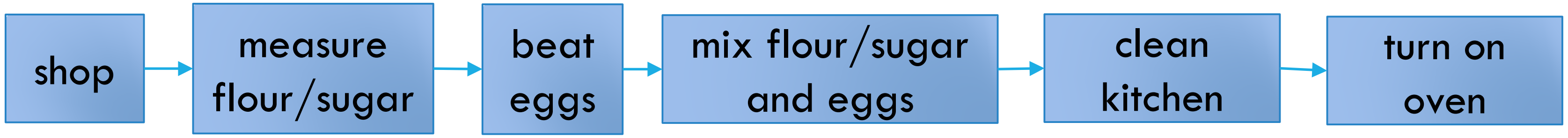
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





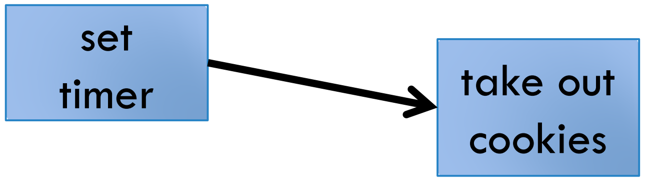
IDEA

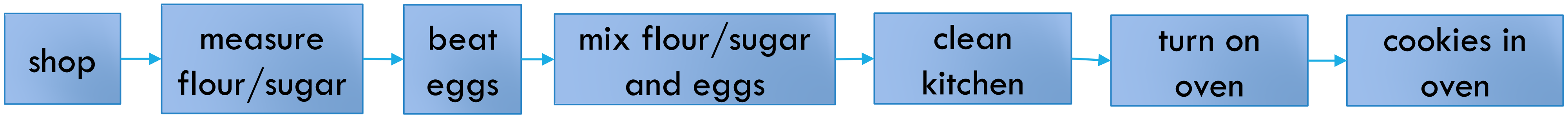
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





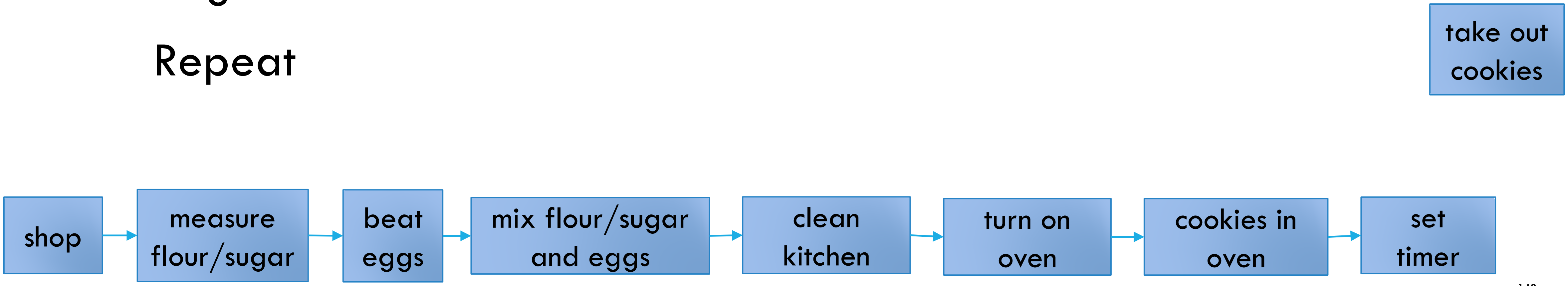
IDEA

Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat



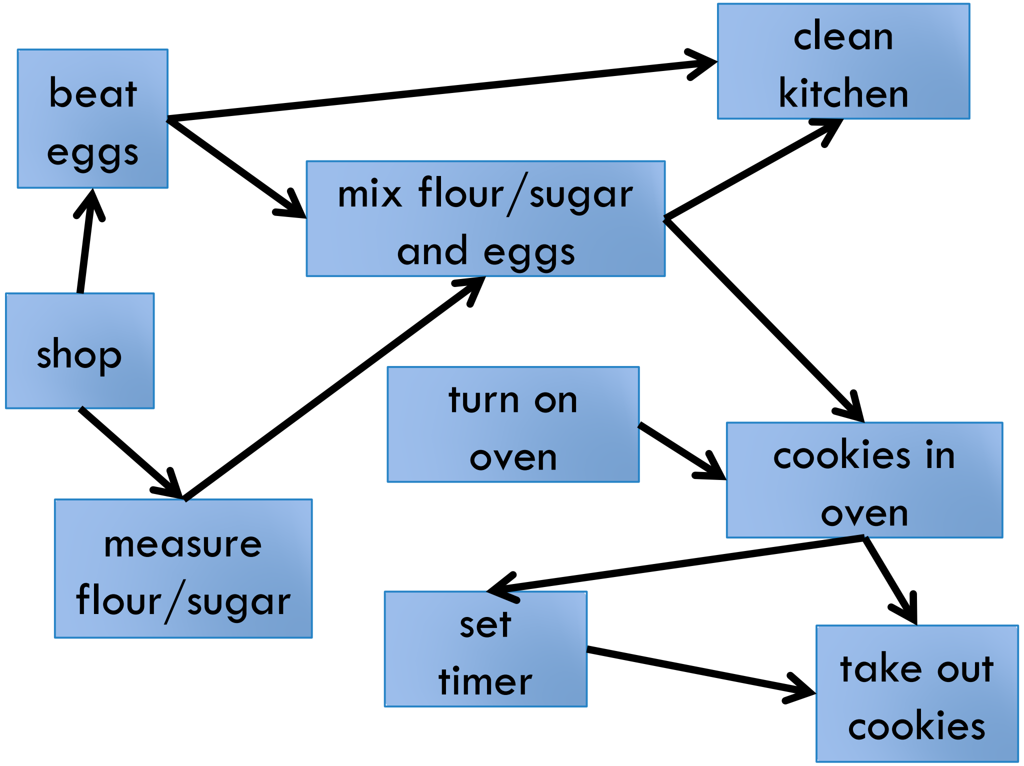
IDEA

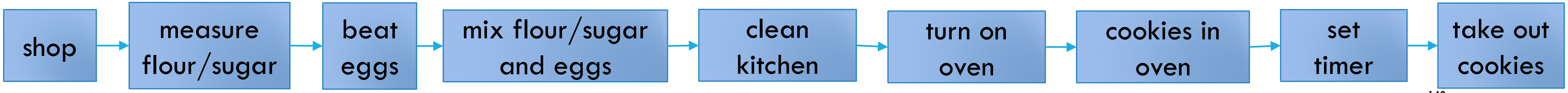
Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat





KAHN'S ALGORITHM

Start at any node v with no incoming edges.

Add v to our list

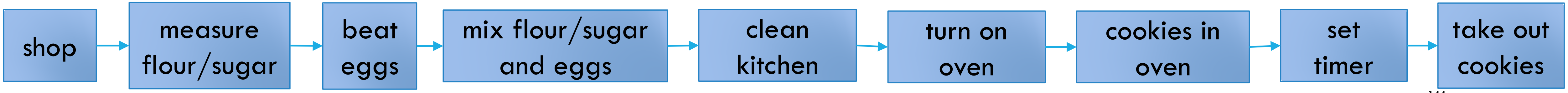
Remove v and all its outgoing edges.

Repeat

Pseudocode:

```
L = list()
S = list()
add all nodes with no incoming edge to S
while S is not empty:
    remove node v from S
    add v to tail of L
    for each of v's neighbors u
        remove edge e where source is v
        if u has no other incoming edges
            add u to S
```

What is the time complexity? (assume Adj. List)



Assume adjacency list

KAHN'S ALGORITHM

Start at any node v with no incoming edges.

Add v to our list

Remove v and all its outgoing edges.

Repeat

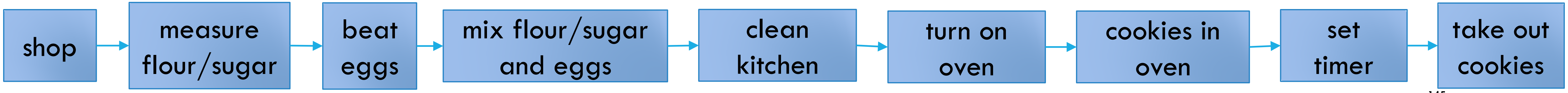
Pseudocode:

$O(V)$

```
L = list()
S = list()
add all nodes with no incoming edge to S
while S is not empty:
    remove node v from S
    add v to tail of L
    for each of v's neighbors u
        remove edge e where source is v
        if u has no other incoming edges
            add u to S
```

$O(E)$

What is the time complexity? $O(V + E)$



When a problem seems vague & complicated…

→

Make the problem **specific.**

What are the given inputs and desired outputs?

Assumptions?

→

Simplify the problem instance.

What is the minimal instance that satisfies the problem?

→

Figure out the algorithm step-by-step.

Think about the operations and data structures needed

→

Generalize the algorithm to the more complicated case.

(if necessary)

GENERAL STRATEGY

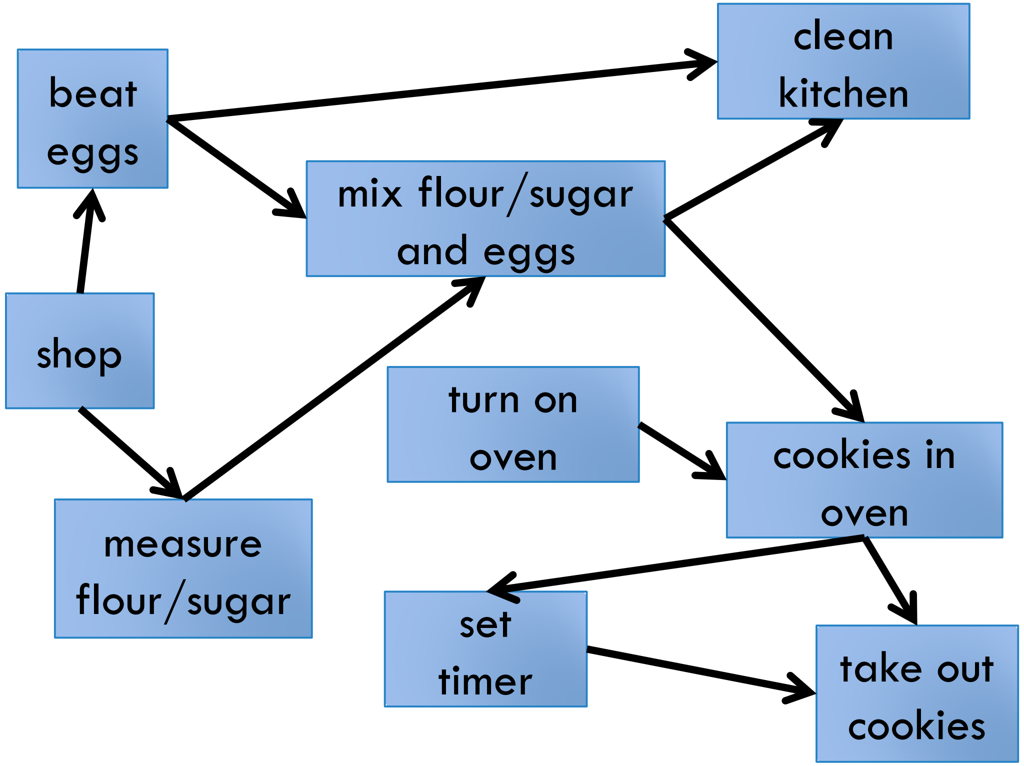
ALTERNATIVE SOLUTION

Kahn's Alg: Find first step and move forwards.

Should remind you of an algorithm.

Breadth-first search! We move the frontier forward (S).

Alternative: Can we use DFS?



TOPOLOGICAL SORT USING DFS (ASSUME DAG)

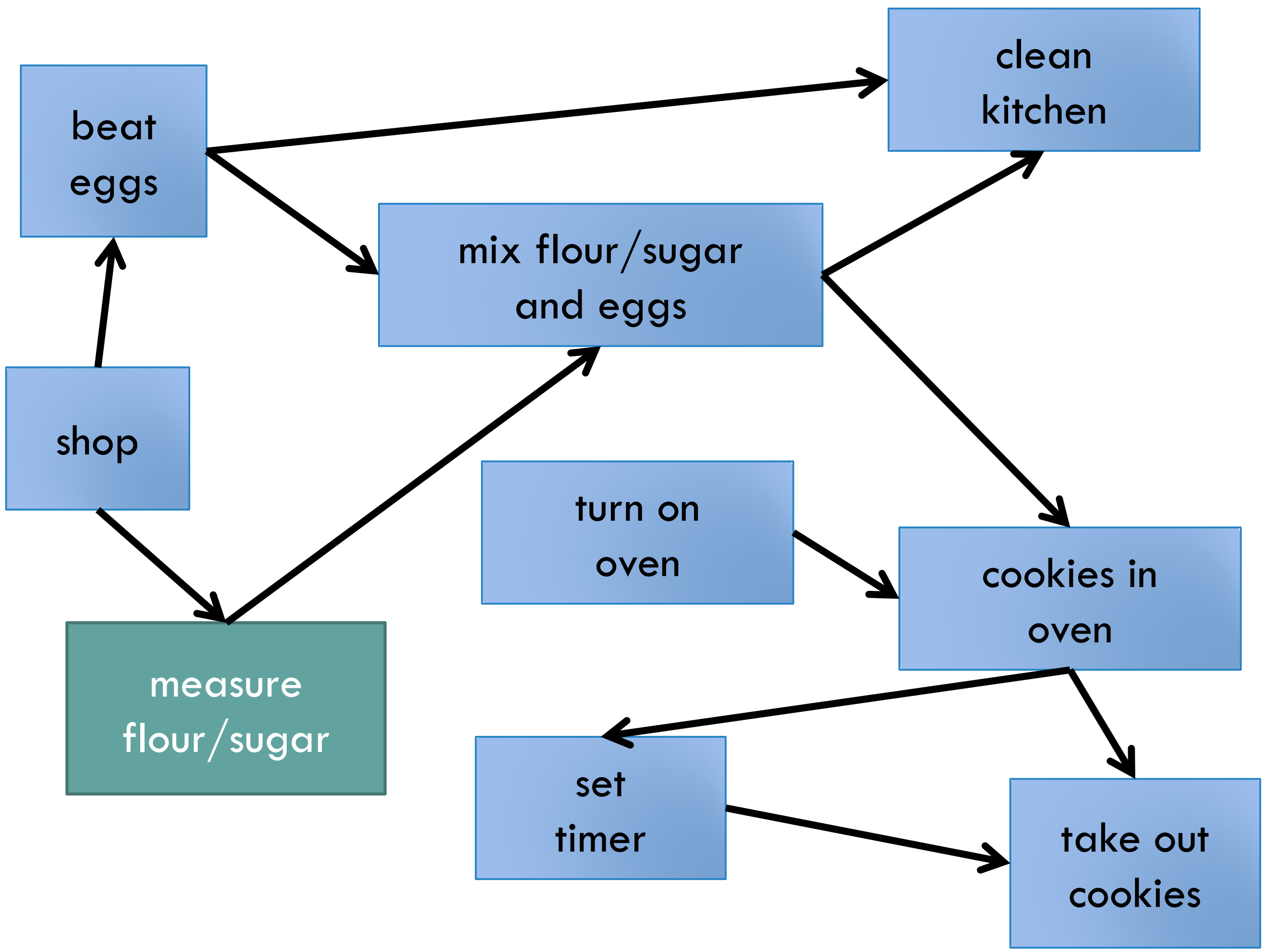
Idea: Process node when it is "last" visited.

```
L = list()
while there are unvisited nodes
    v = select unvisited node
    DFS(G, v, L)
```

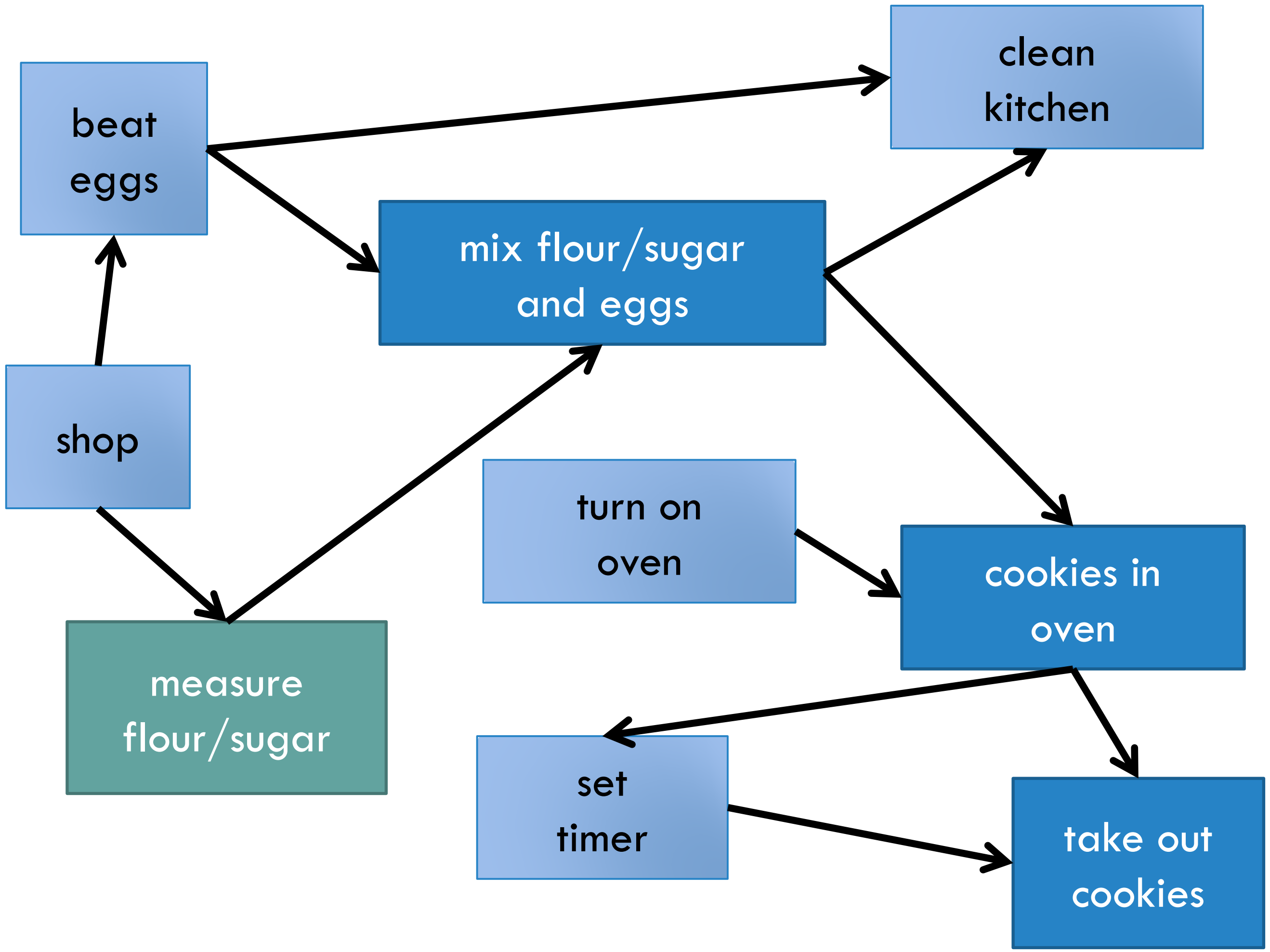
```
DFS(G,v,L)
    if v is visited
        return
    else
        for each of v's neighbor u
            DFS(G, u, L)
    visit(v)
    L.pushFront(v)
```

How can we quickly check if there are unvisited nodes or select unvisited nodes?

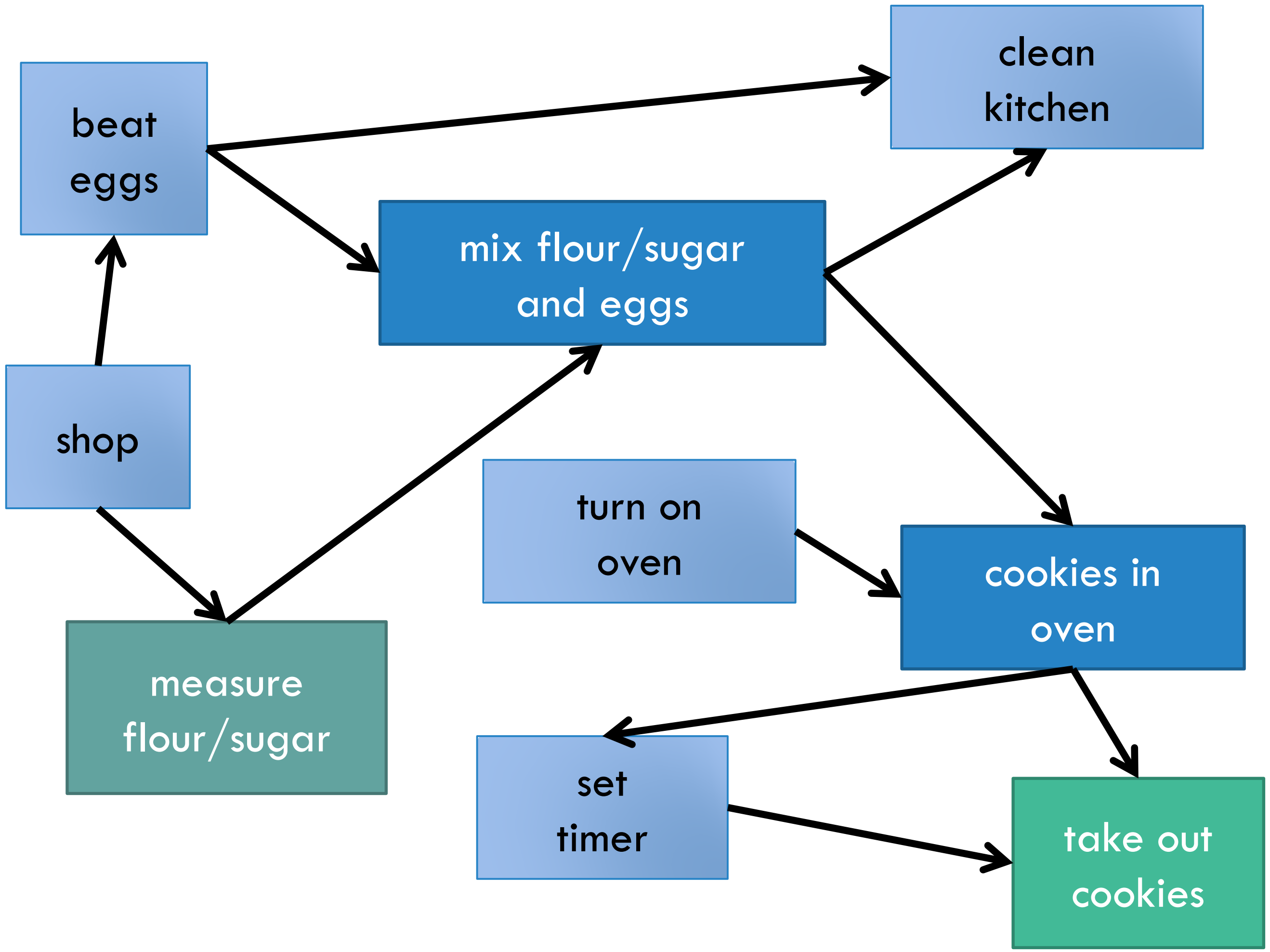
List/ Hash Table / Set



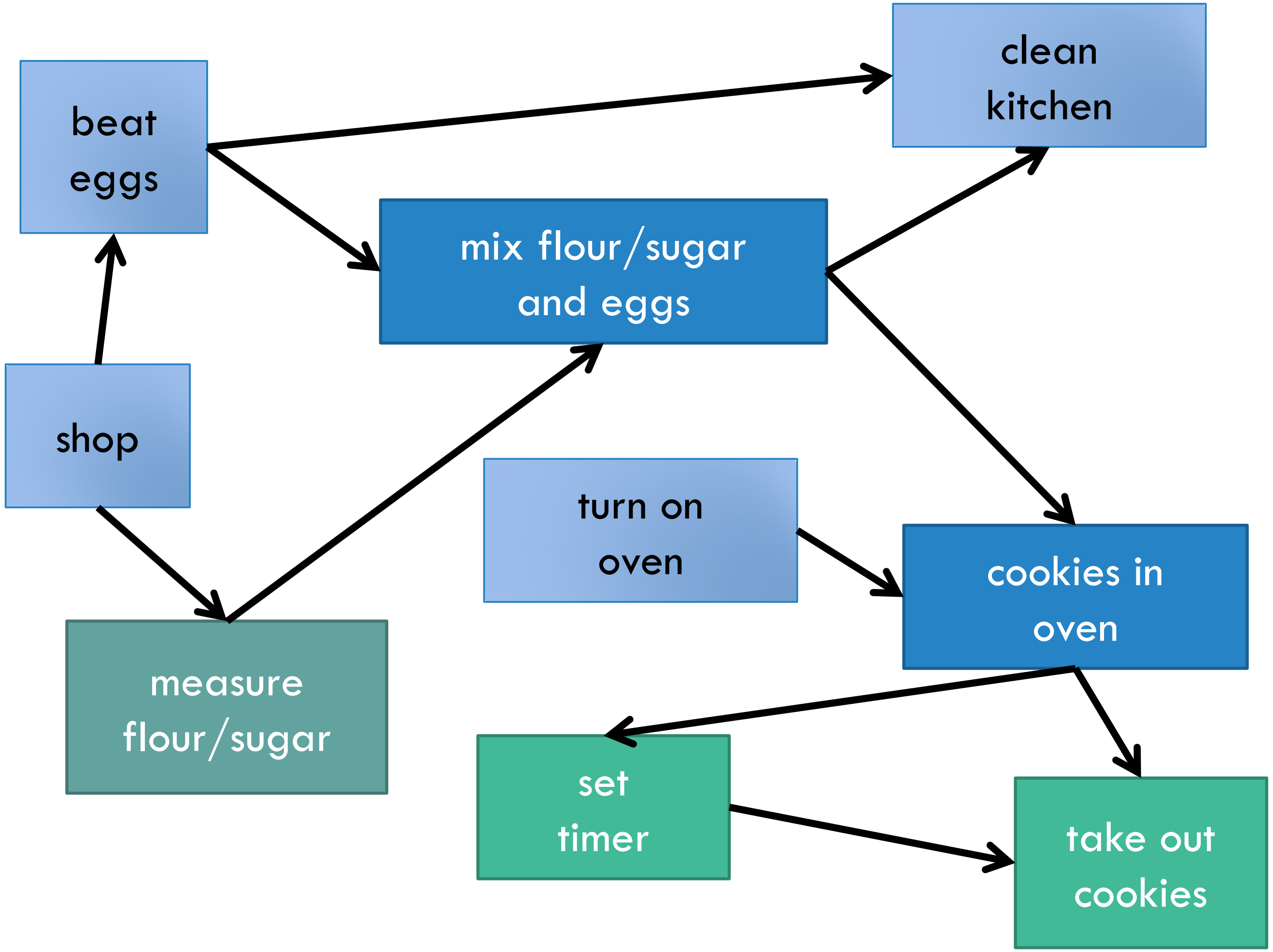
beat eggs
clean kitchen
mix flour/sugar and eggs
shop
turn on oven
cookies in oven
measure flour/sugar
set timer
take out cookies


beat eggs
clean kitchen
mix flour/sugar and eggs
shop
turn on oven
cookies in oven
measure flour/sugar
set timer
take out cookies

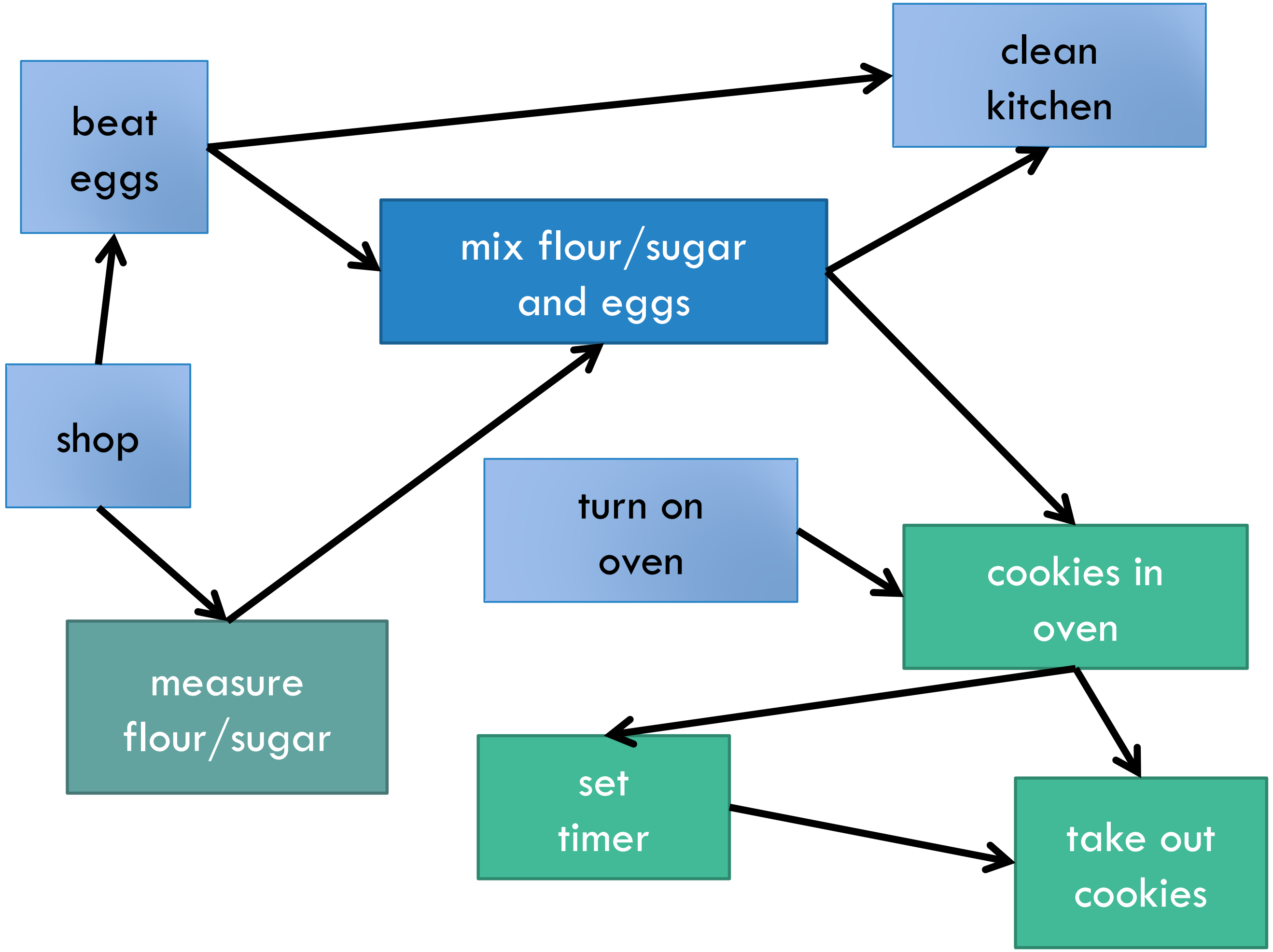
1.
2.
3.
4.
5.
6.
7.
8.
9. take out



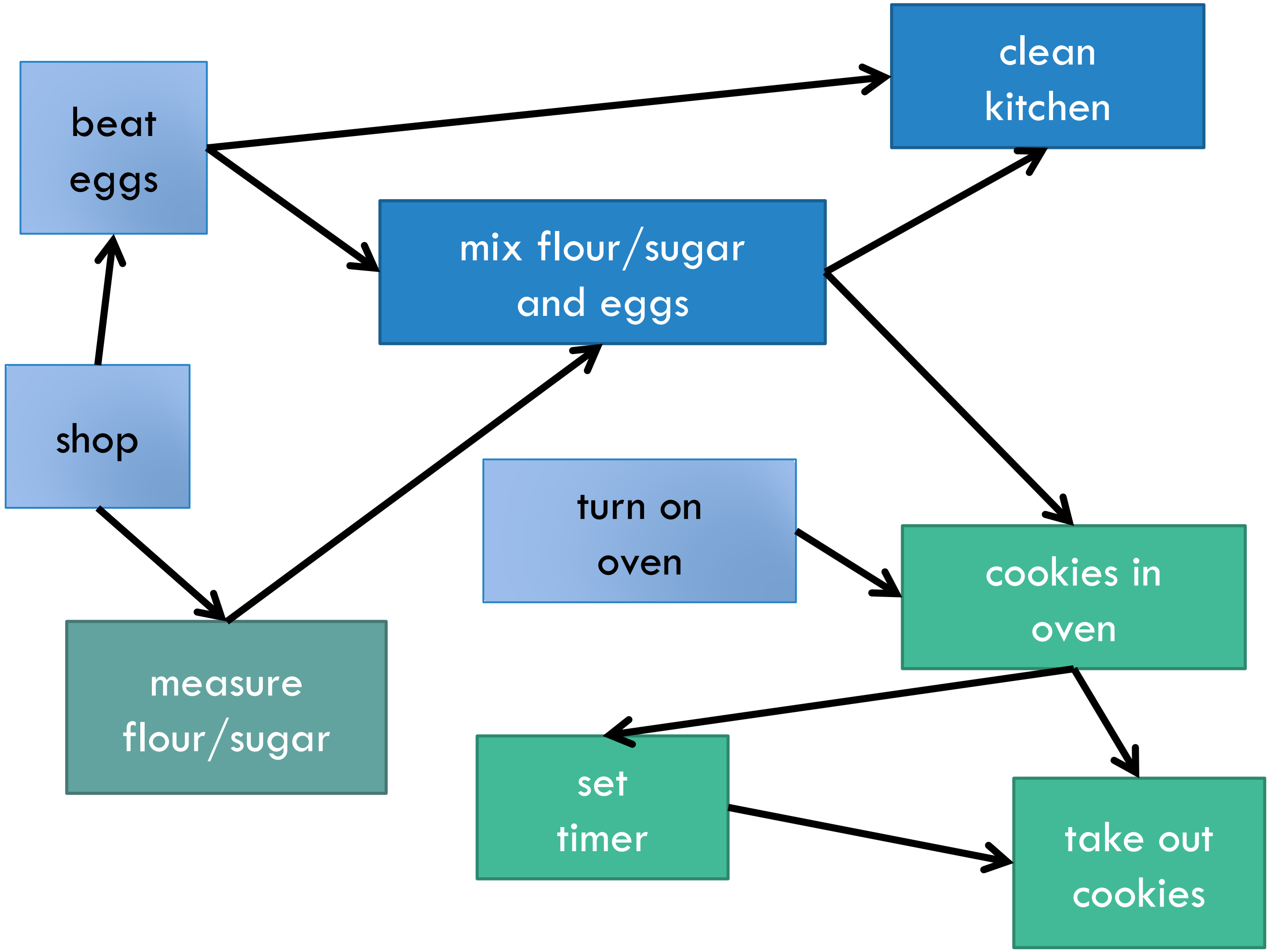
1.
2.
3.
4.
5.
6.
7.
8. set timer
9. take out



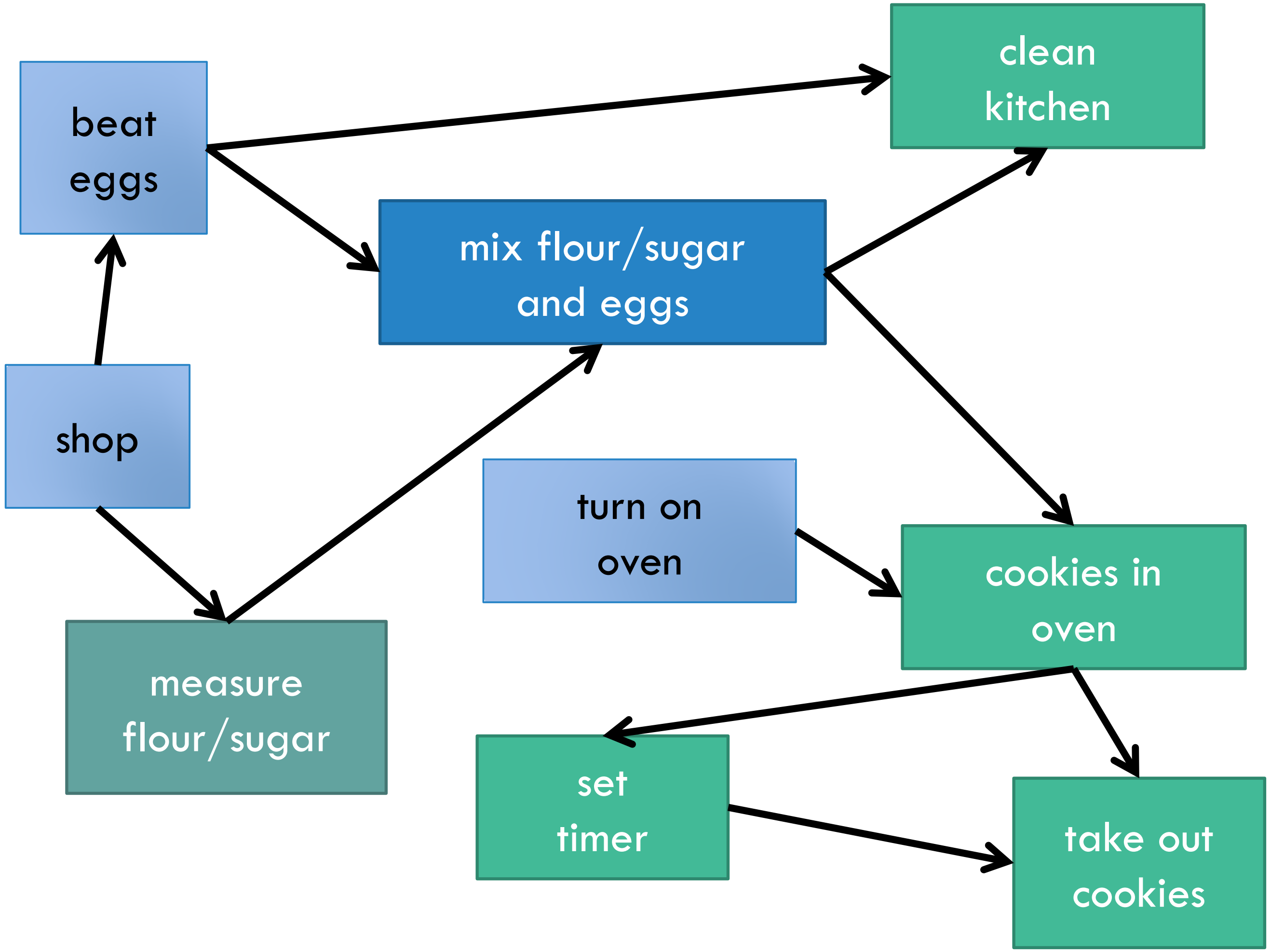
1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



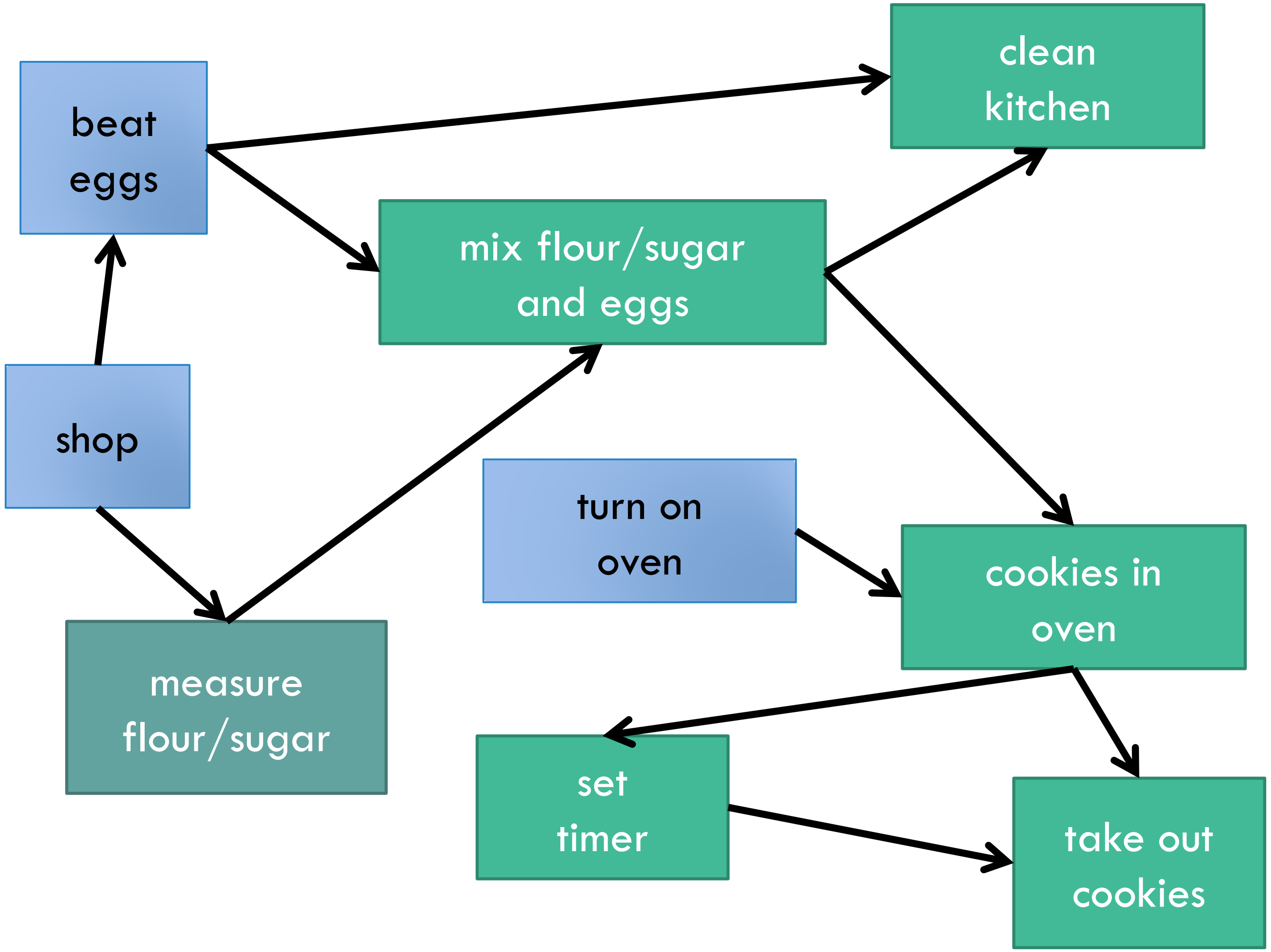
1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



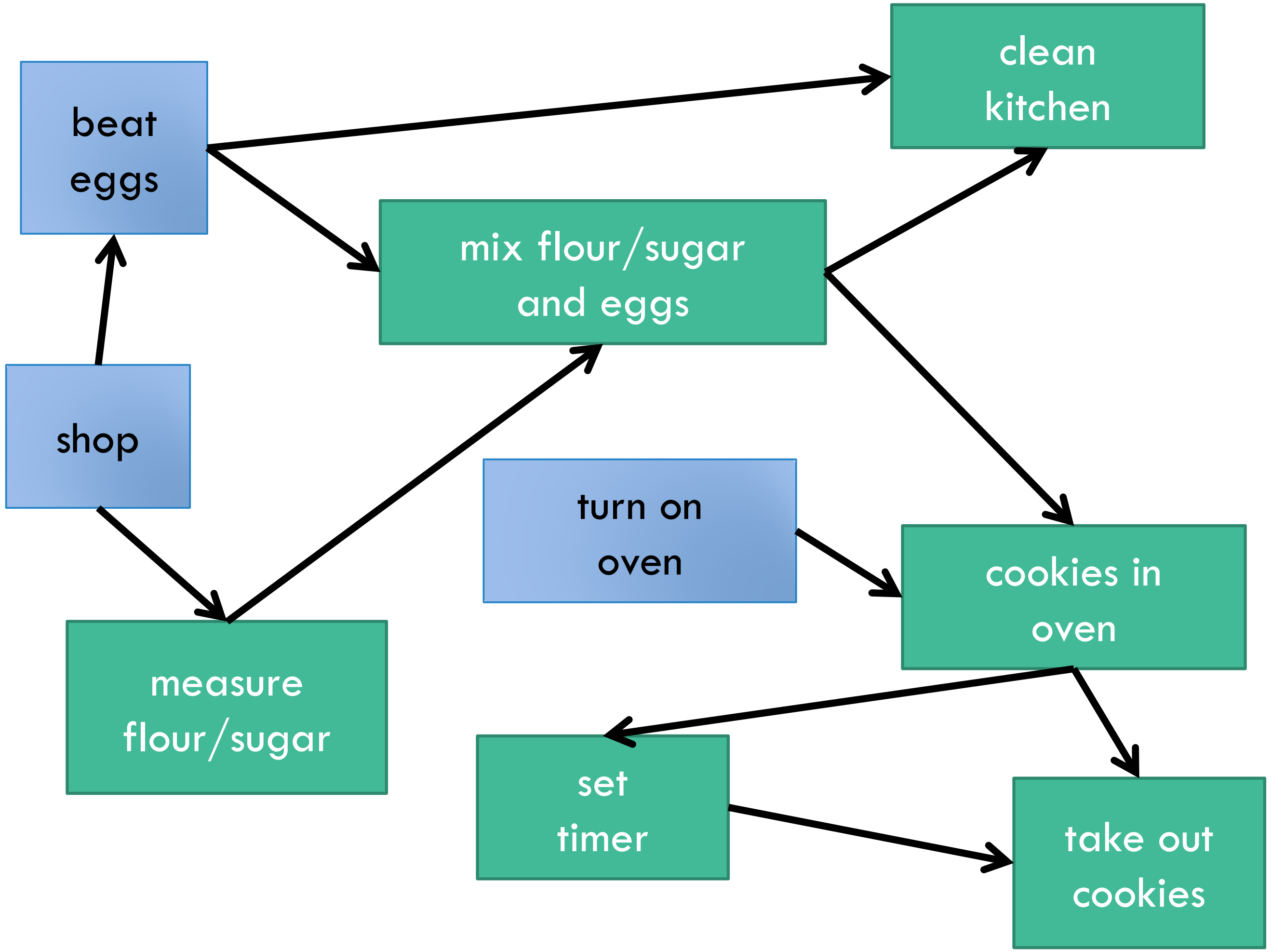
1.
2.
3.
4.
5.
6. clean
7. in oven
8. set timer
9. take out



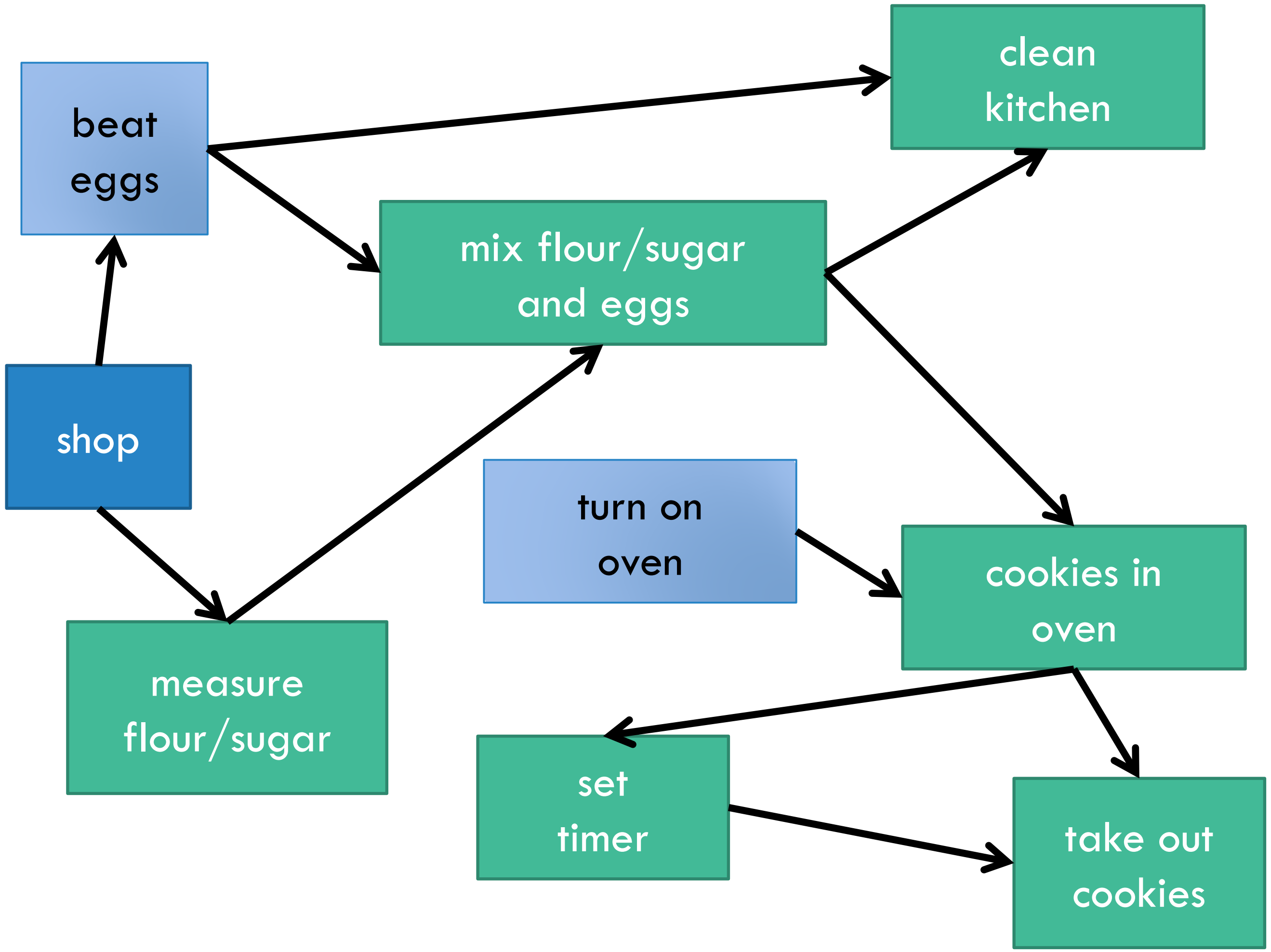
1.
2.
3.
4.
5. mix
6. clean
7. in oven
8. set timer
9. take out



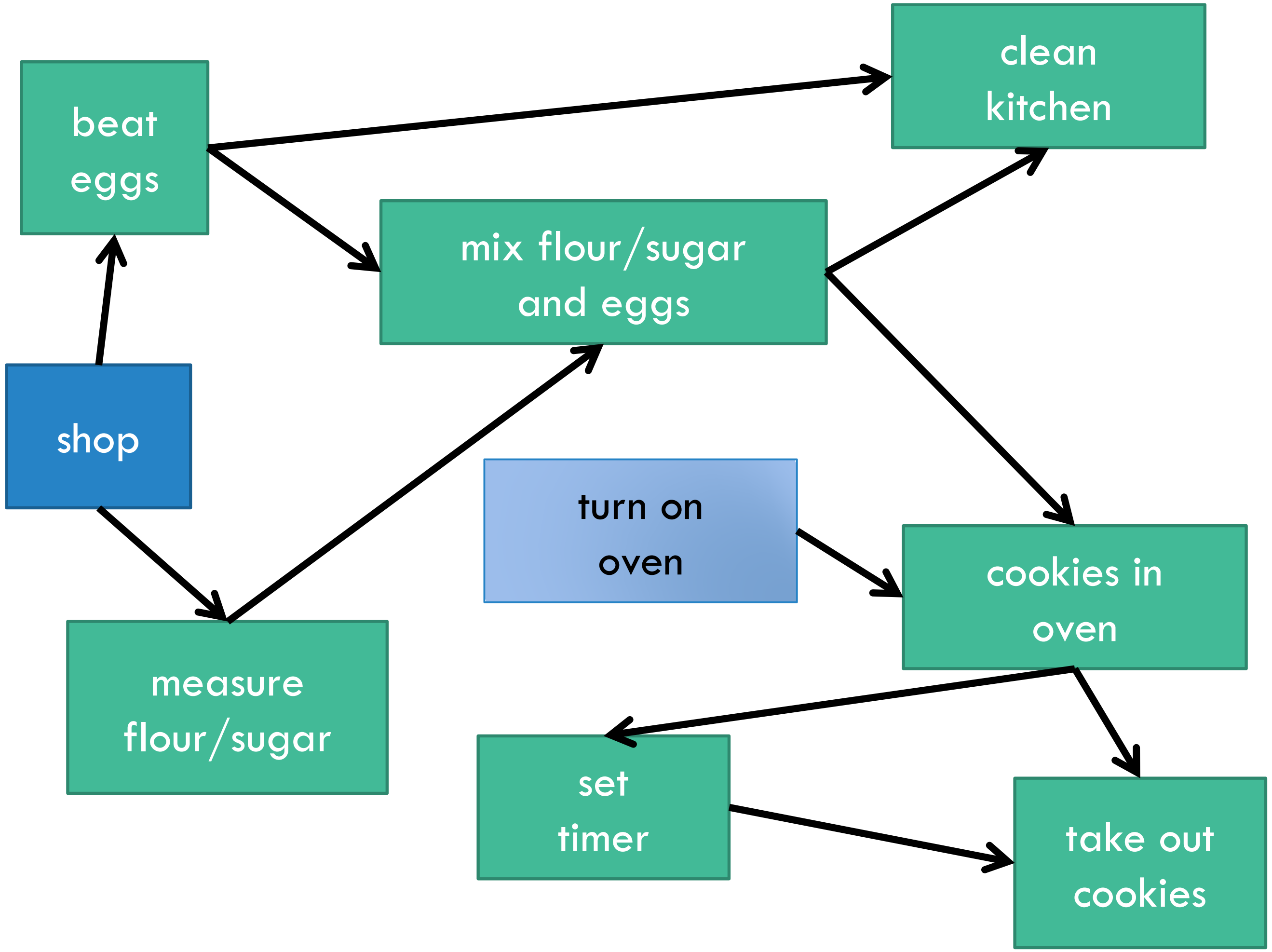
1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



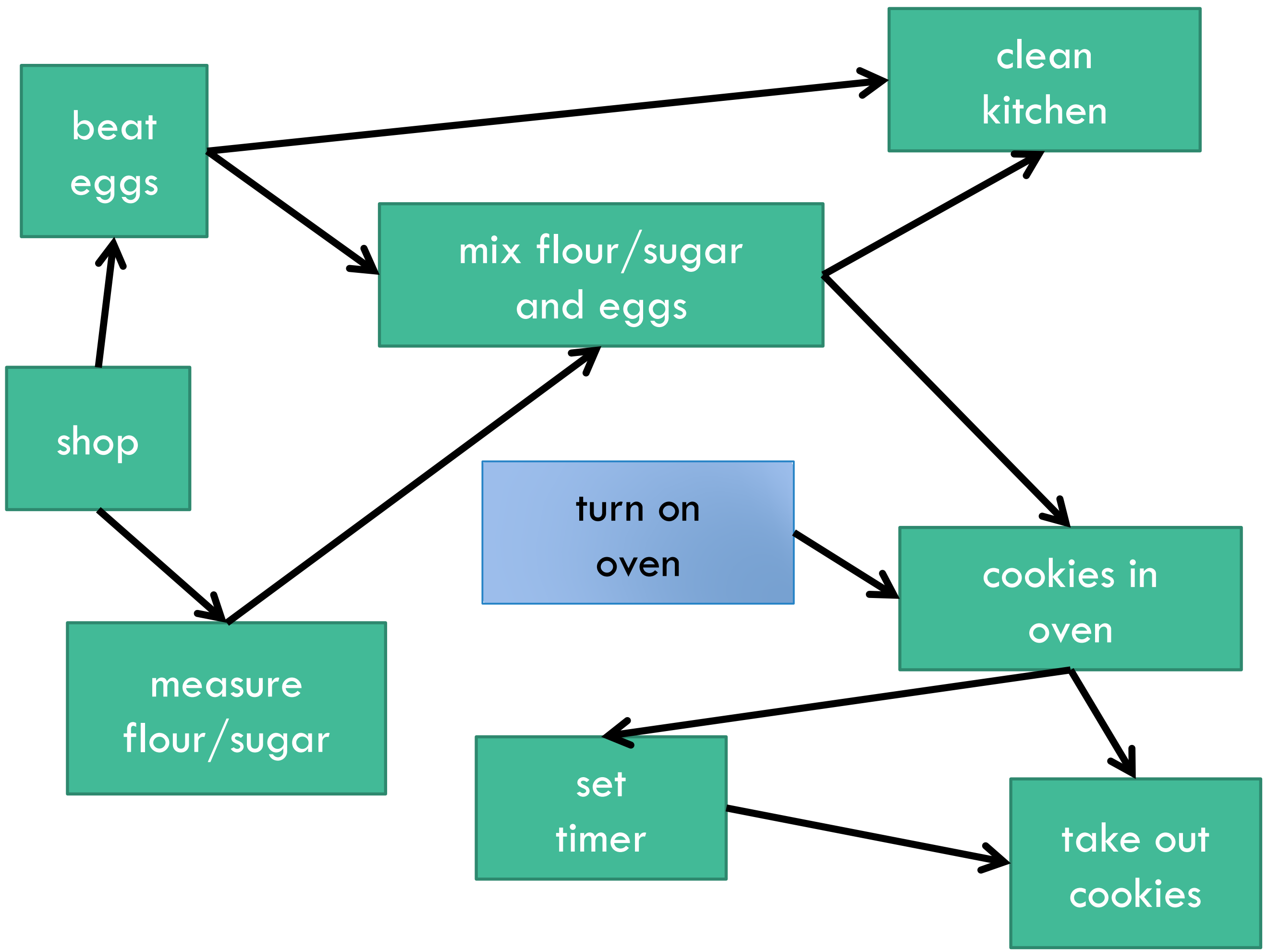
1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



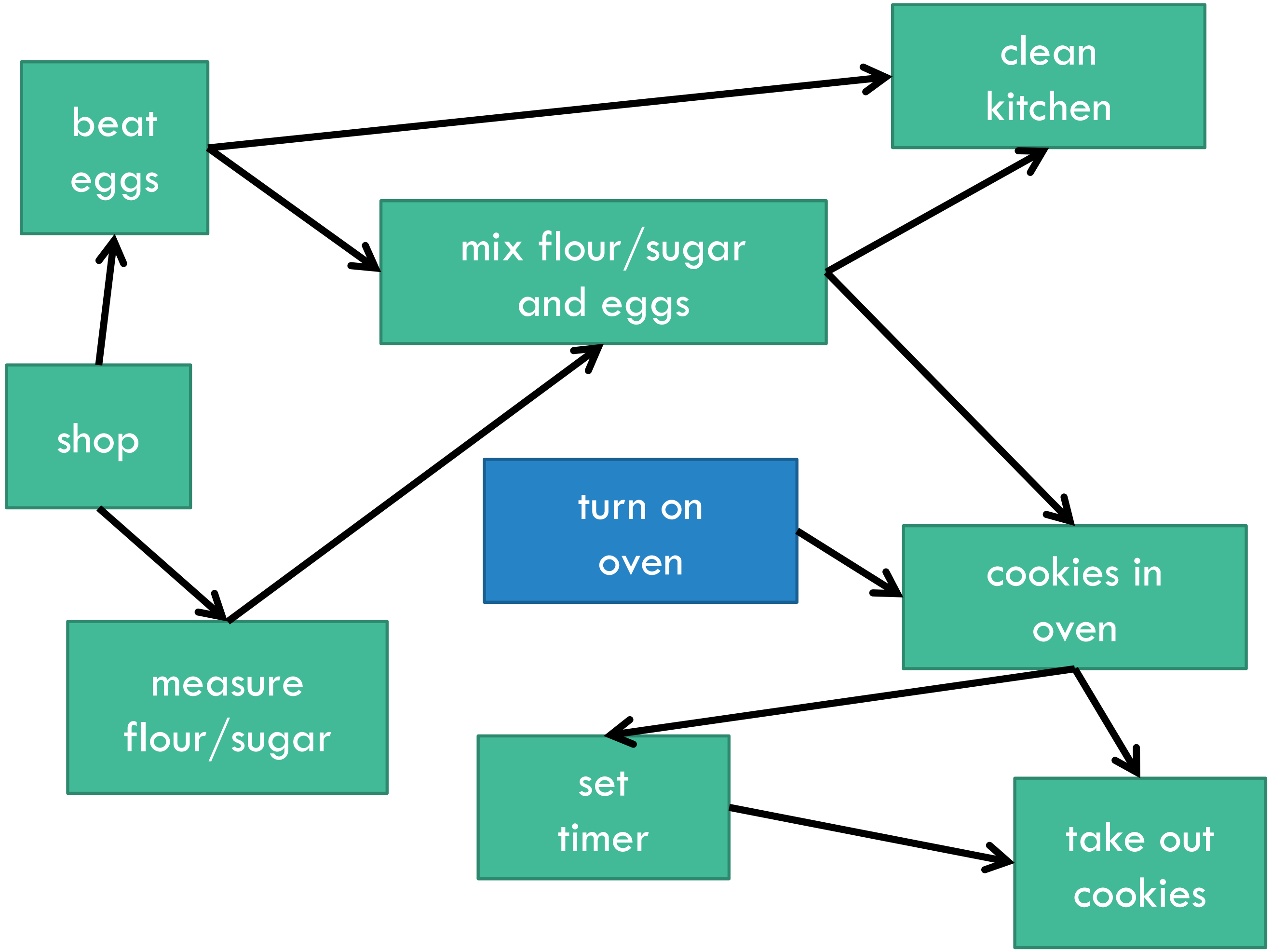
1.
2.
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



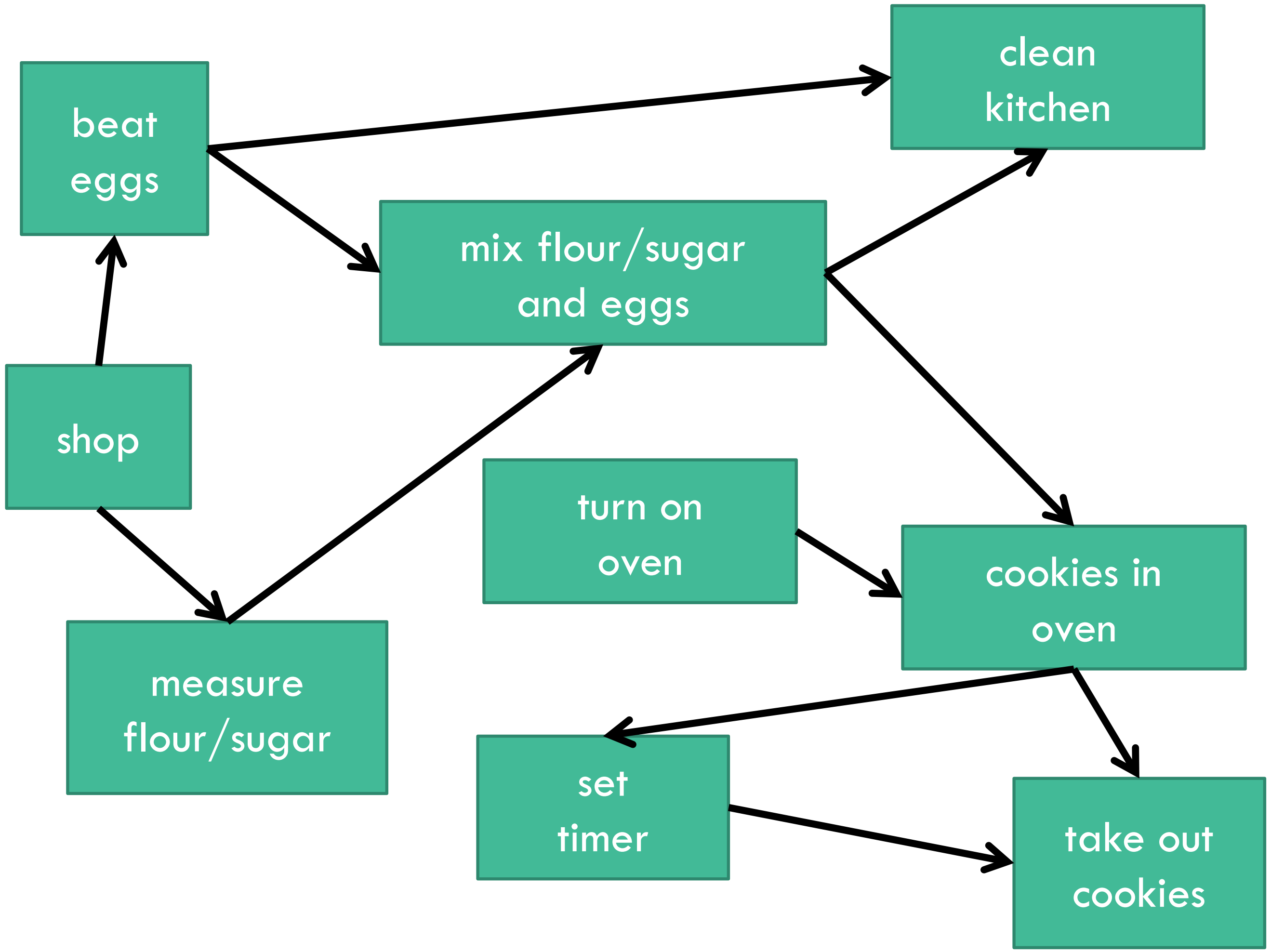
1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



1. on oven
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



TOPOLOGICAL SORT USING DFS (ASSUME DAG)

Idea: Process node when it is "last" visited.

```
L = list()
while there are unvisited nodes
    v = select unvisited node
    DFS(G, v, L)
```

```
DFS(G,v,L)
    if v is visited
        return
    else
        for each of v's neighbor u
            DFS(G, u, L)
    visit(v)
    L.pushFront(v)
```

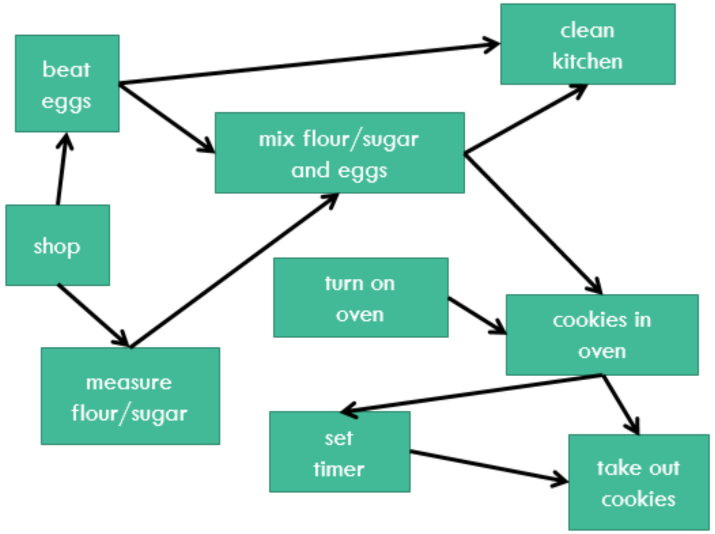
What is the time complexity?

$$O(V+E)$$

Same as DFS

IS A TOPOLOGICAL ORDERING *UNIQUE*?

1. on oven
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



Is every topological ordering unique

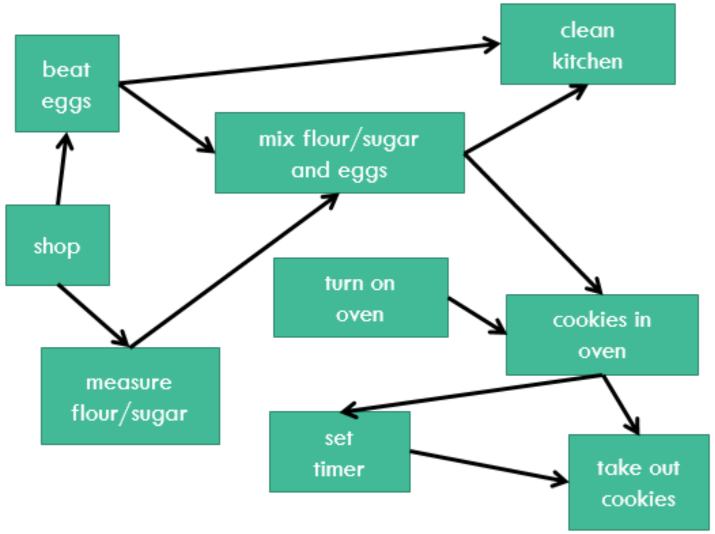
A. Yes!

B. No!

C. 

IS A TOPOLOGICAL ORDERING *UNIQUE*?

1. on oven
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



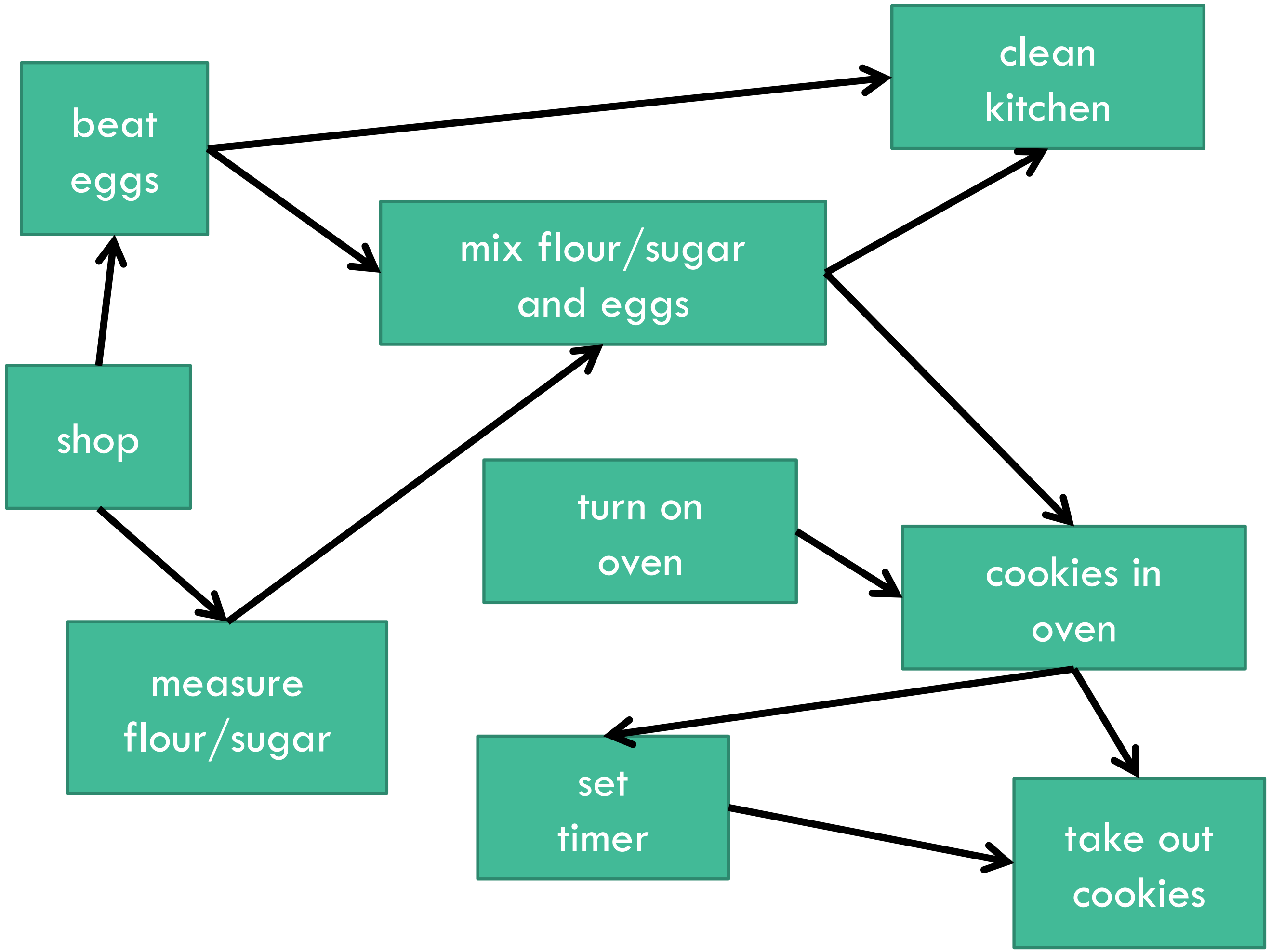
Is every topological ordering unique

A. Yes!

B. No!

C. 

1. **on oven**
2. **shop**
3. beat
4. measure
5. mix
6. **clean**
7. in oven
8. **set timer**
9. take out



SUMMARY: LEARNING OUTCOMES

By the end of this session, students should be able to:

- Explain the **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** Algorithms.
- State the **similarities** and **differences** between the two algorithms
- Analyze the **performance of BFS and DFS**
- Describe the **topological sort** algorithm

SEARCHING A GRAPH

Goal:

- Start at some vertex **s** = start.
- Find some other vertex **f** = finish.

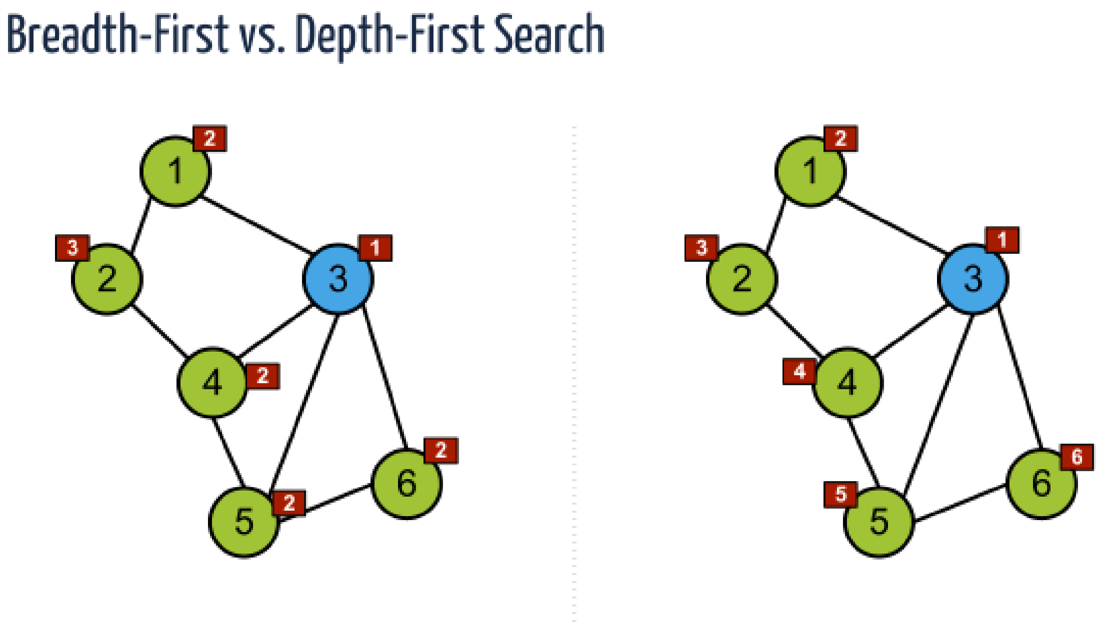
  Or: visit **all** the nodes in the graph

Two basic techniques:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

Graph representation:

- Adjacency list

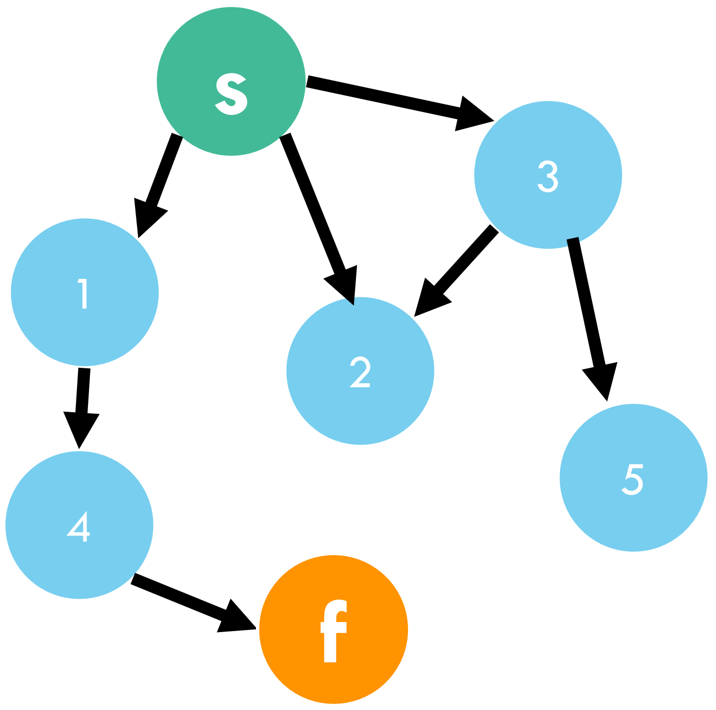


Red = active
Gray = visited
Blue = unvisited

BFS: STEP-BY-STEP

```
BFS(G, s, f)
    visit(s)
    Queue.add(s)
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Queue.enqueue(u)
    return null
```

Queue:

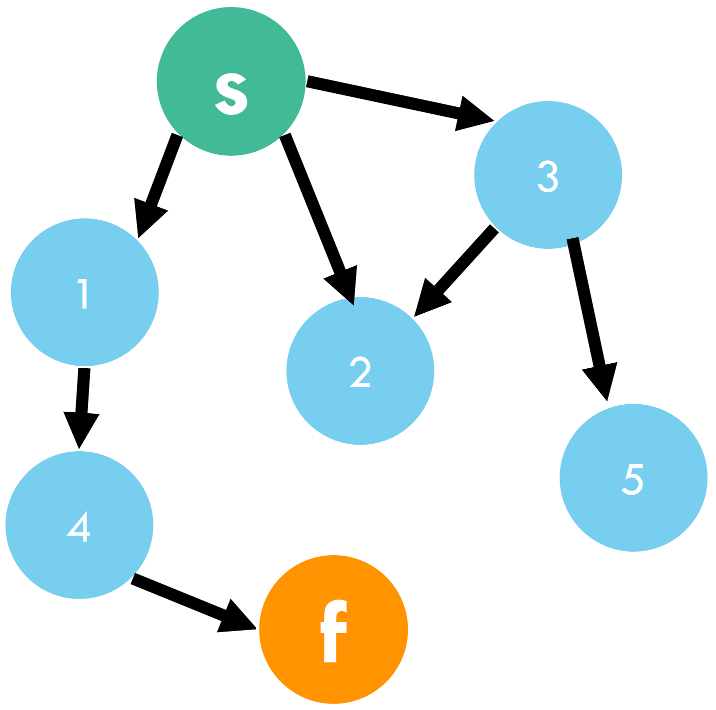


DFS: STEP-BY-STEP

Red = active
Gray = visited
Blue = unvisited

```
DFS(G, s, f)
    visit(s)
    Stack.push(s)
    while not Stack.empty()
        curr = Stack.pop()
        if curr == f
            return curr
        for each neighbor u of curr
            if u is not visited
                visit(u)
                Stack.push(u)
    return null
```

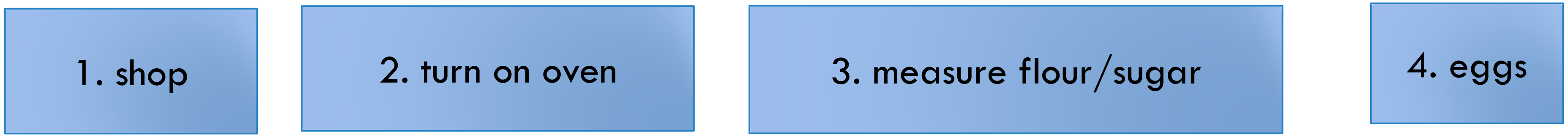
Stack:



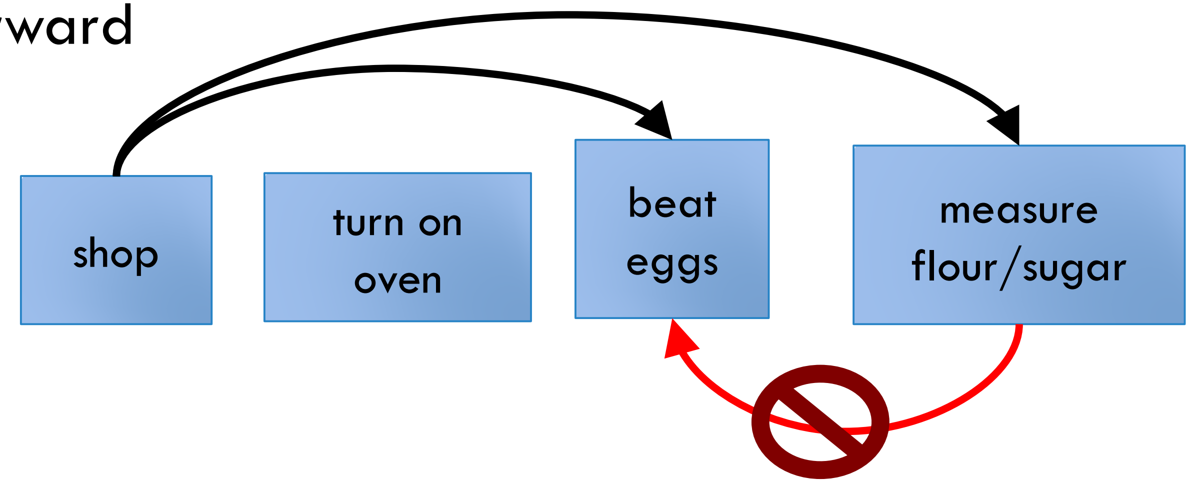
TOPOLOGICAL ORDER

Properties:

1. Sequential total ordering of all nodes



2. Edges only point forward



LEARNING OUTCOMES

By the end of this session, students should be able to:

- Explain the **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** Algorithms.
- State the **similarities** and **differences** between the two algorithms
- Analyze the **performance of BFS and DFS**
- Describe the **topological sort** algorithm

QUESTIONS?

BEFORE LECTURE TOMORROW

Please revise Single-Source Shortest Paths on Visualgo

- Sections 1-9 (until DFS)