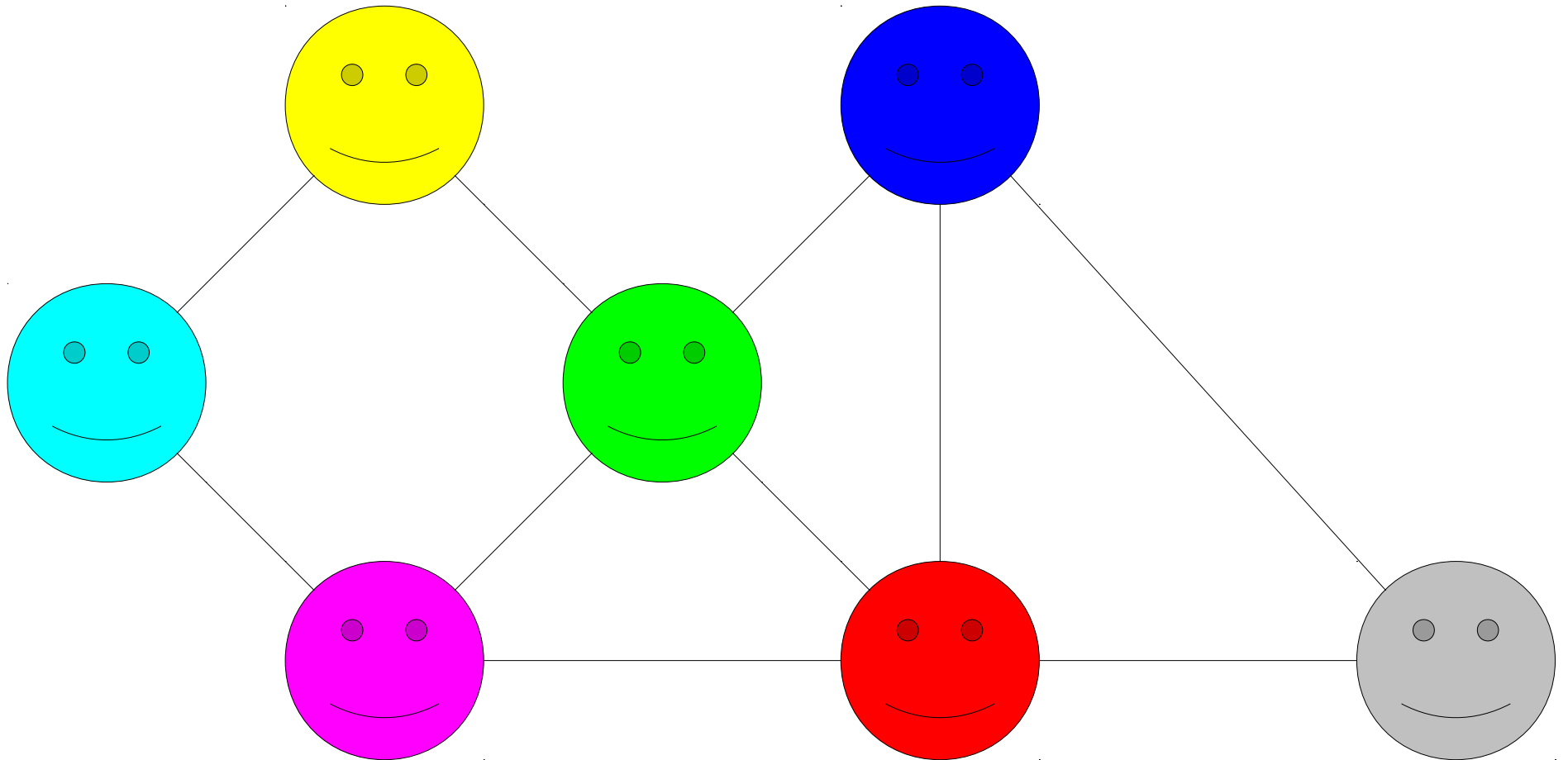
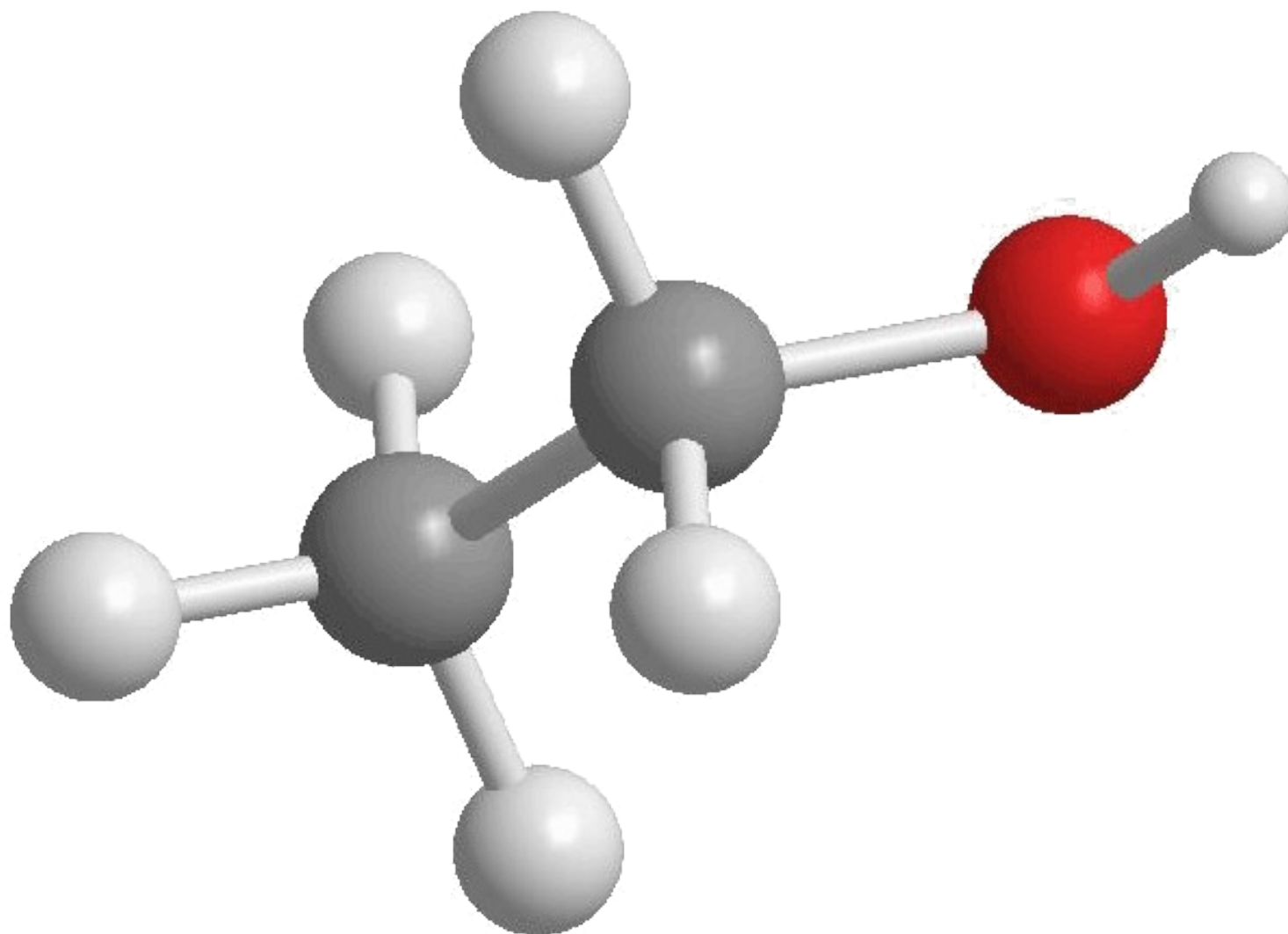


# Graphs

# A Social Network

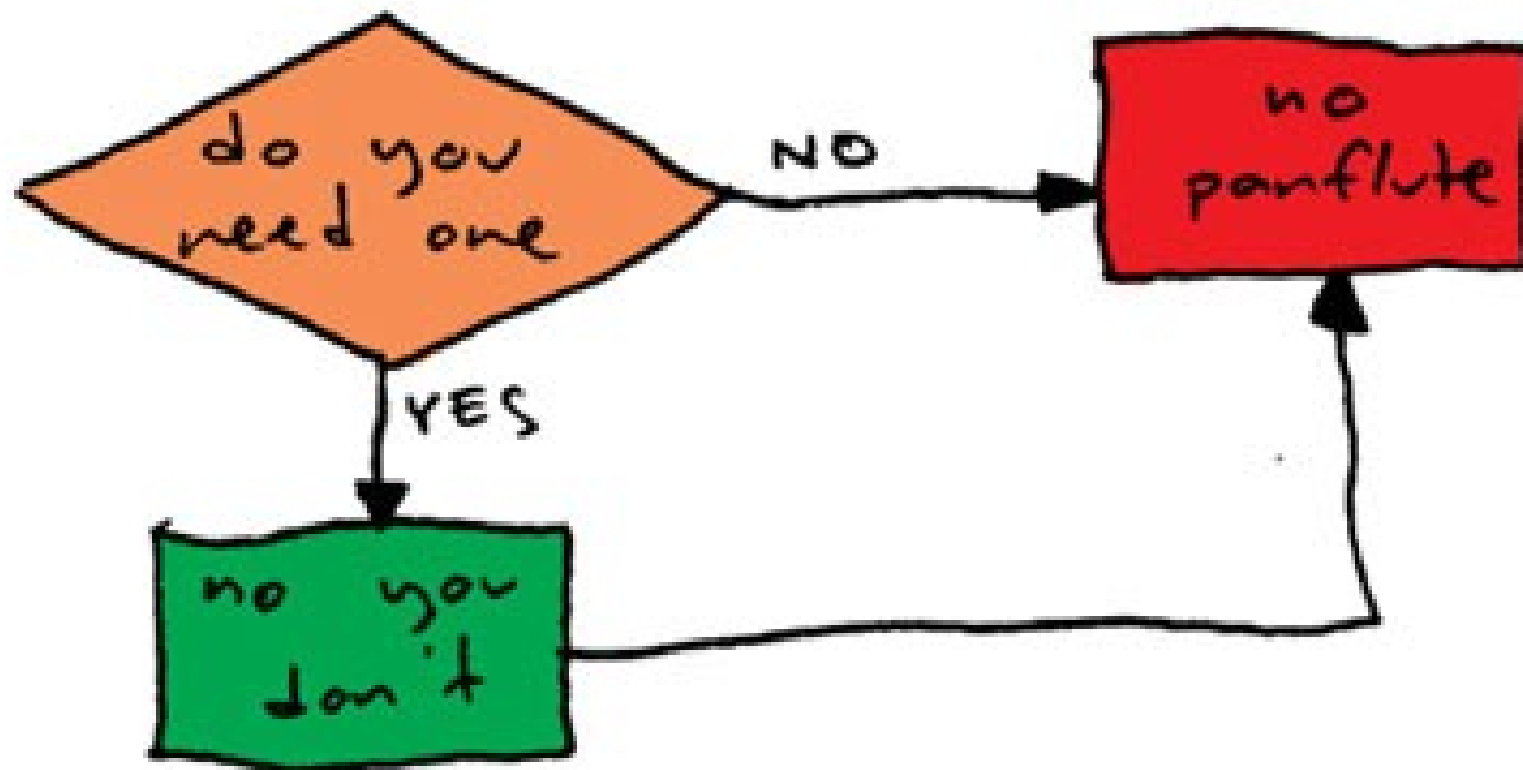


# Chemical Bonds





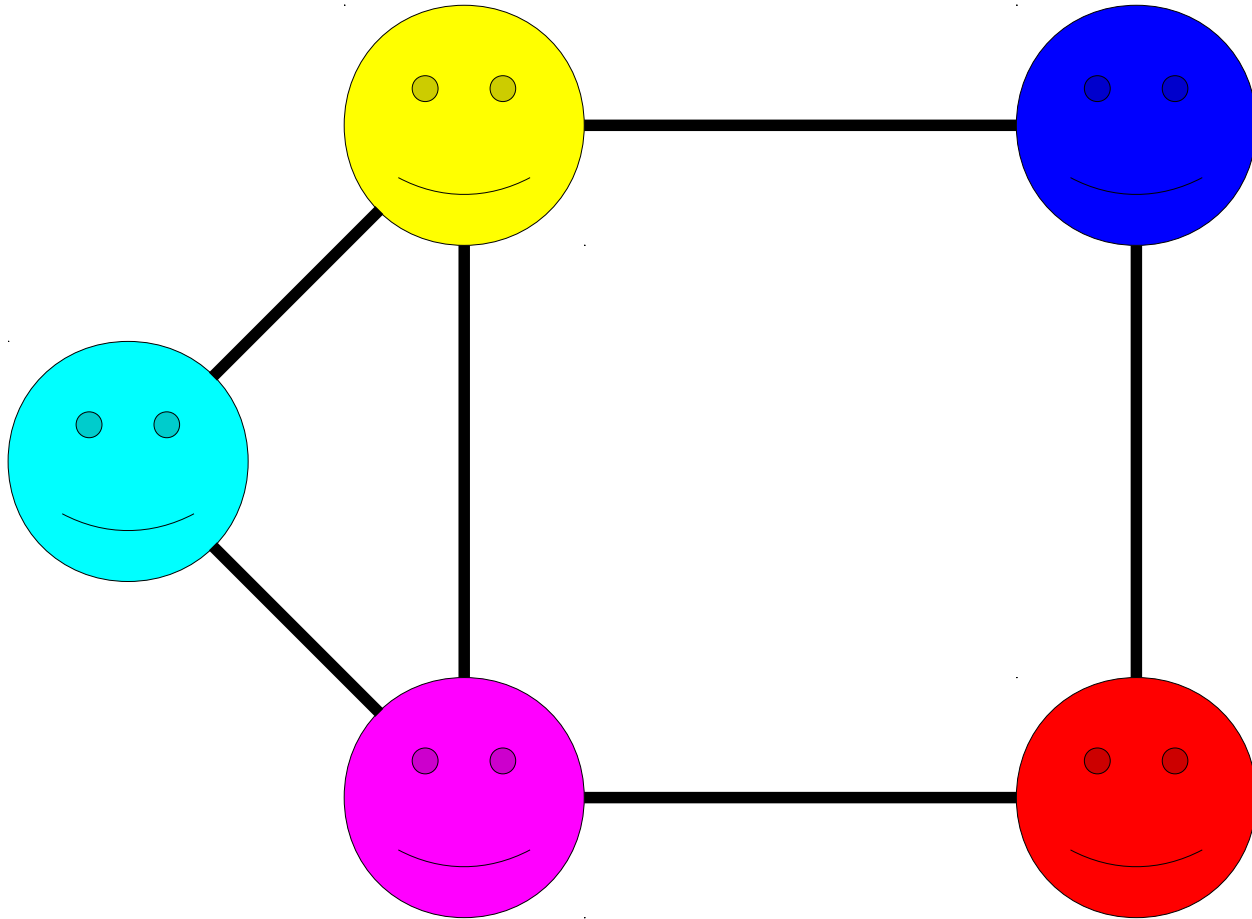
# PANFLUTE FLOWCHART



A ***graph*** is a mathematical structure for representing relationships.

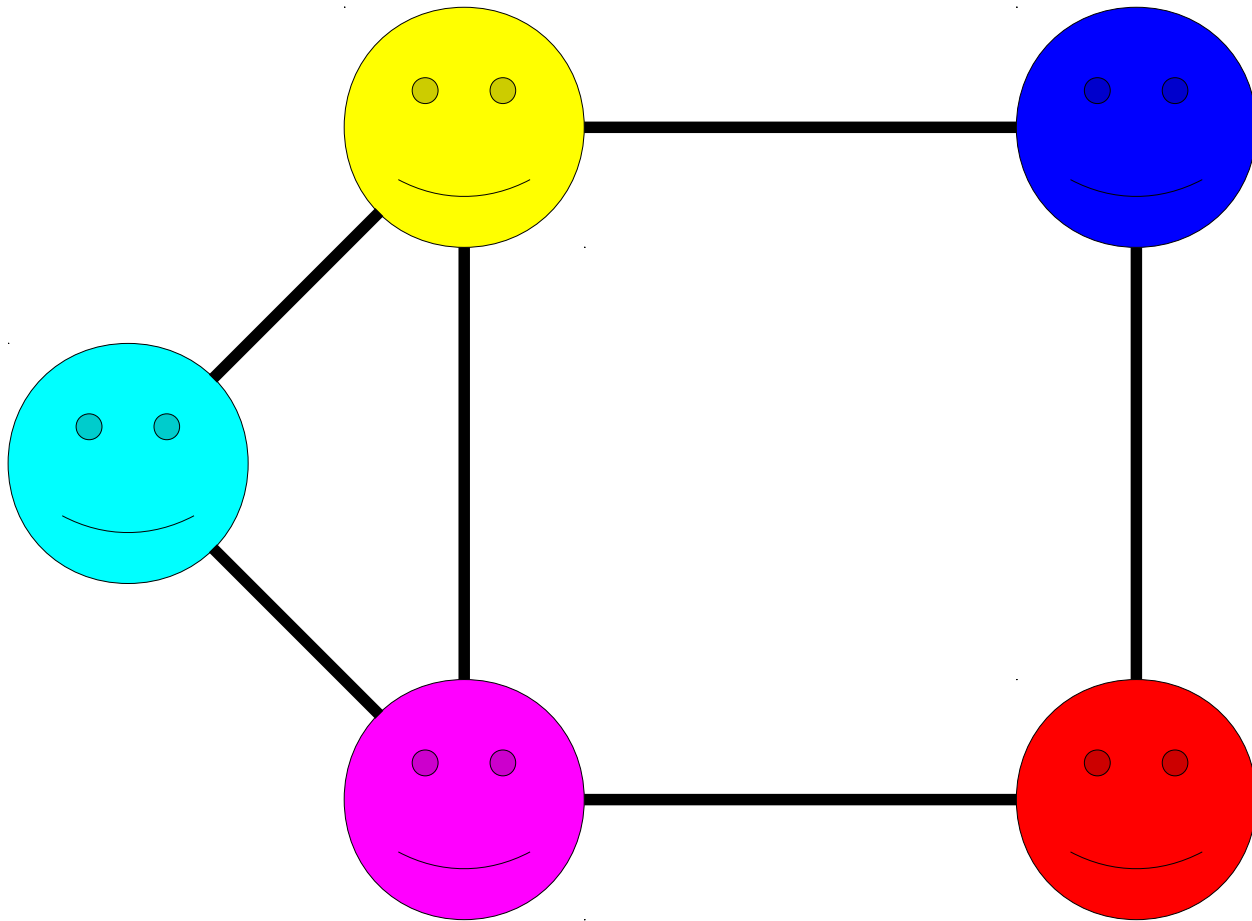
A ***graph*** is a mathematical structure for representing relationships.

A ***graph*** is a mathematical structure for representing relationships.



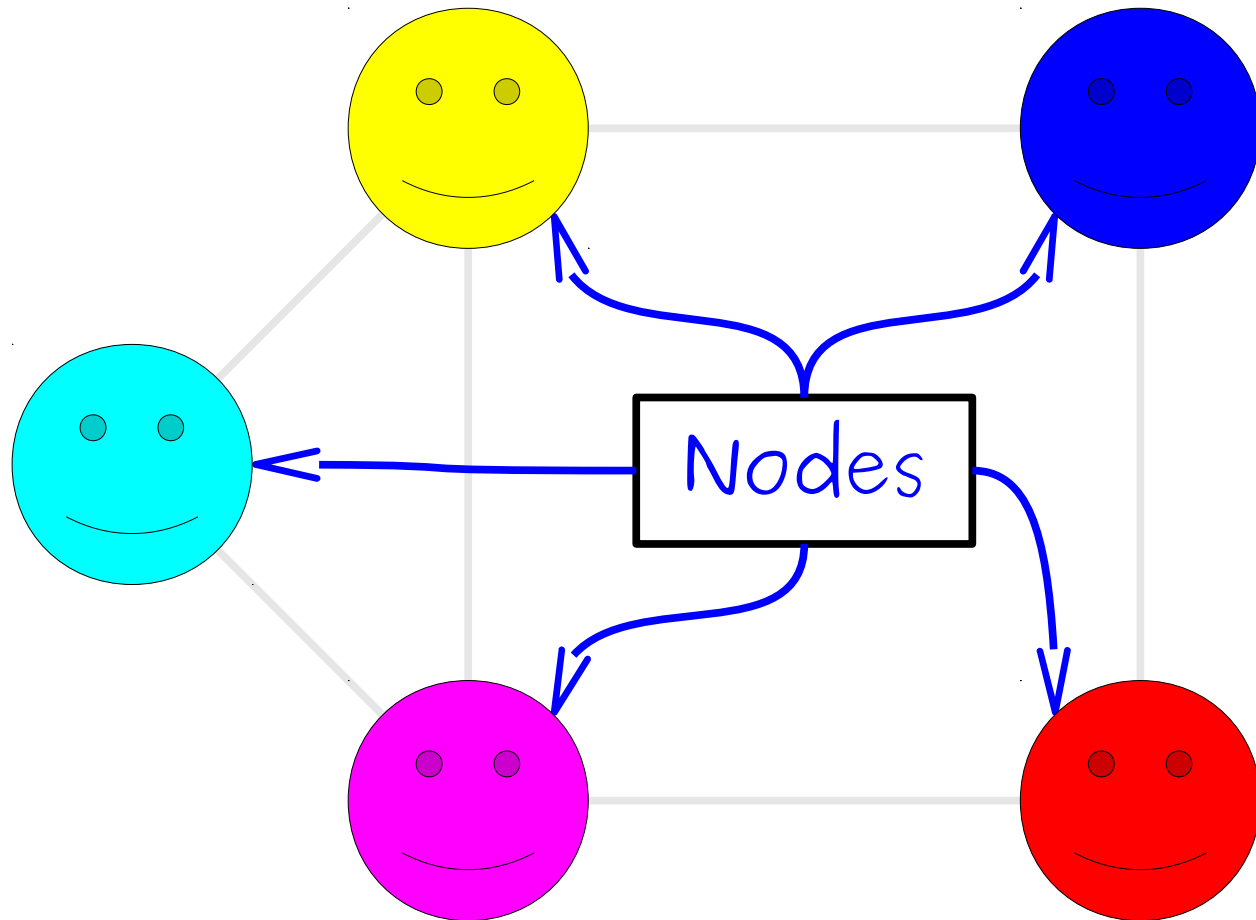


A **graph** is a mathematical structure for representing relationships.



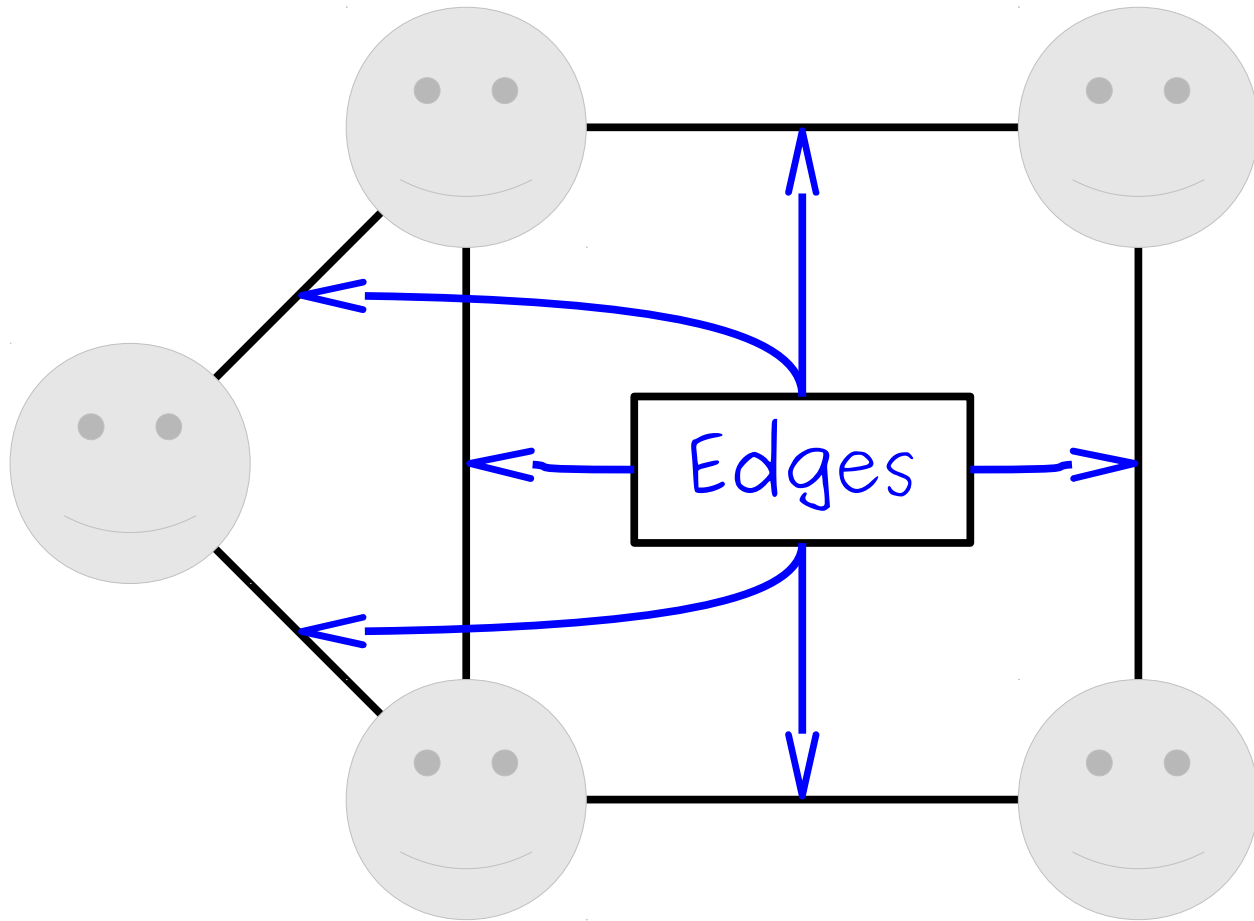
A graph consists of a set of **nodes** connected by **edges**.

A **graph** is a mathematical structure for representing relationships.



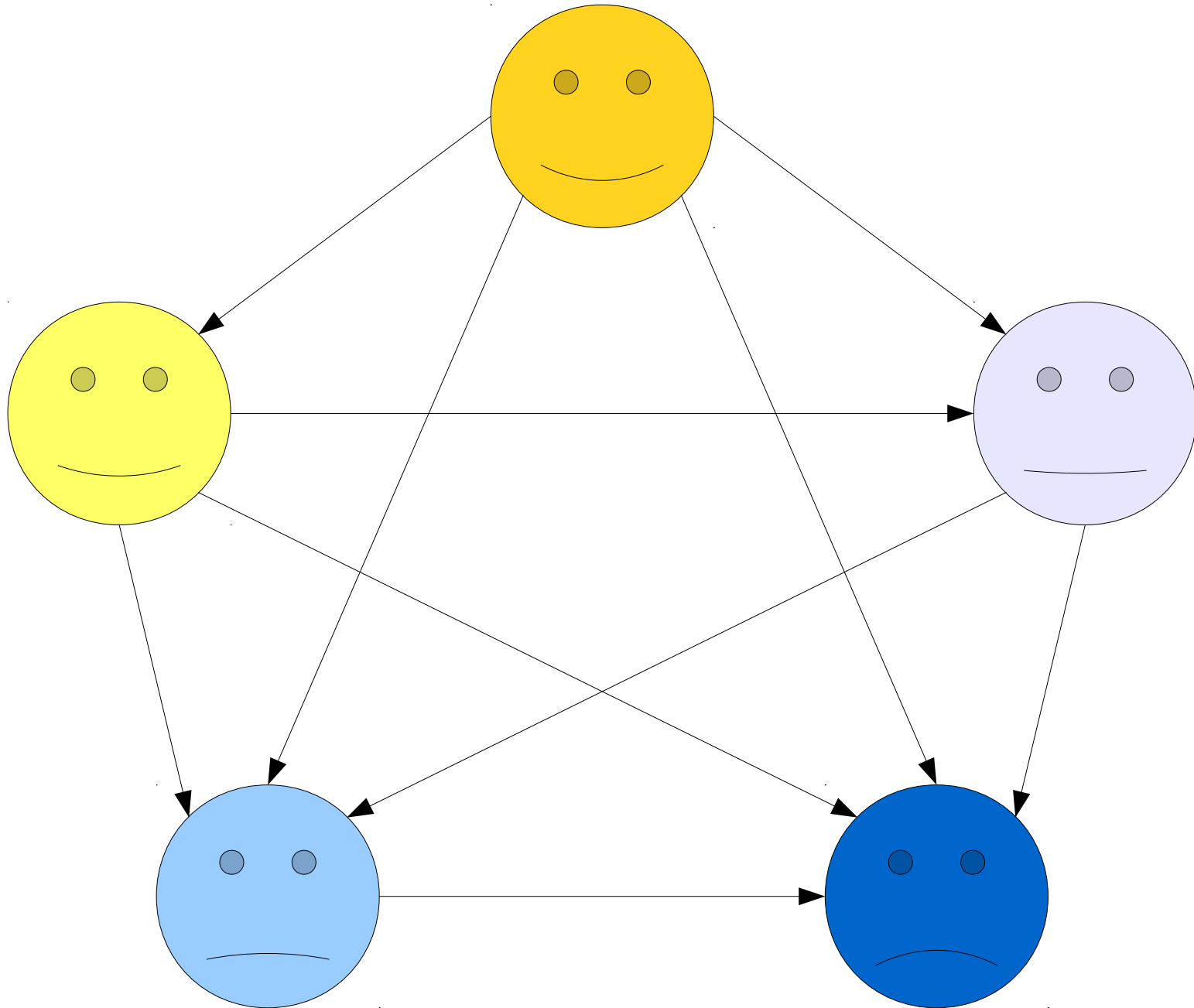
A graph consists of a set of **nodes** connected by **edges**.

A **graph** is a mathematical structure for representing relationships.

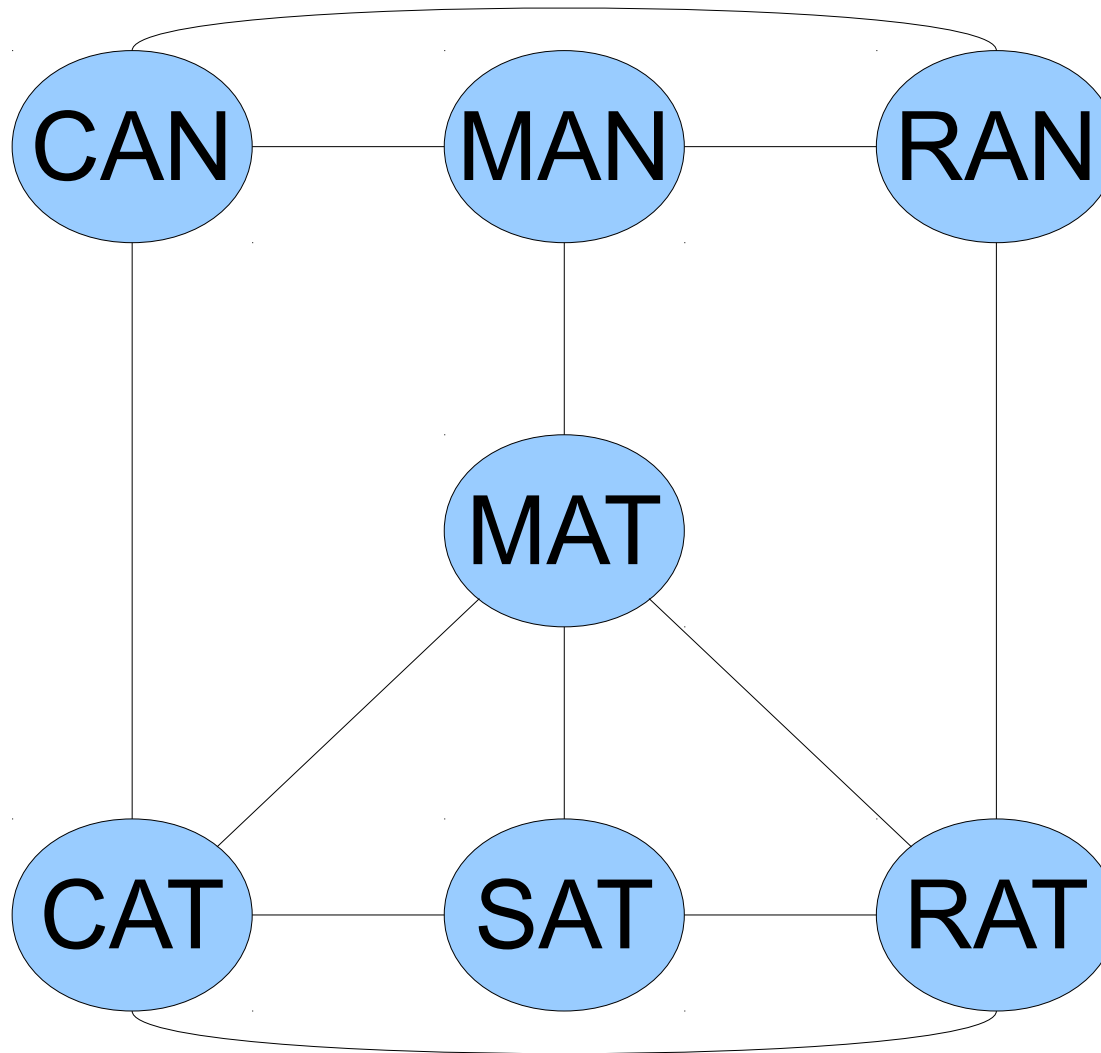


A graph consists of a set of **nodes** connected by **edges**.

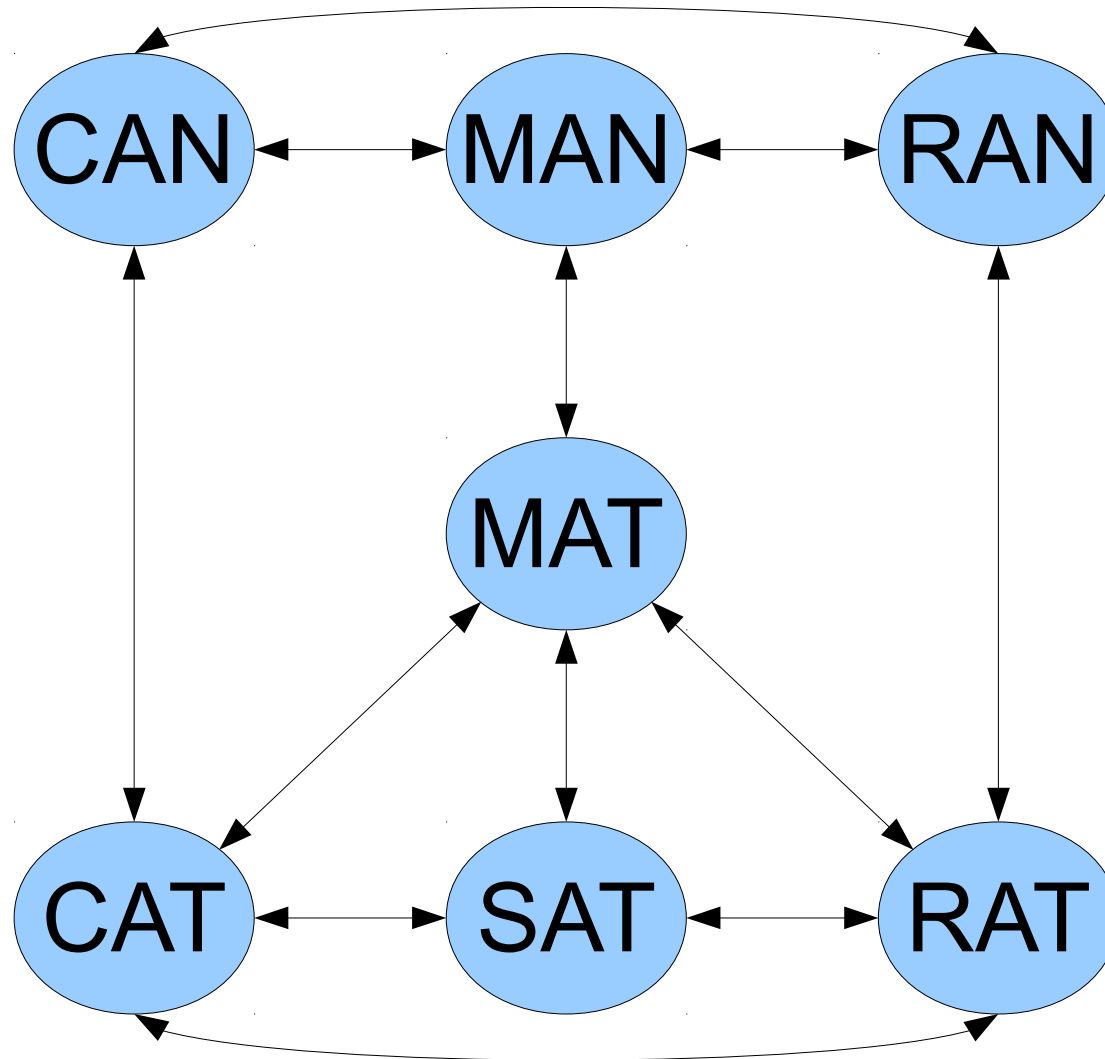
Some graphs are *directed*.



Some graphs are *undirected*.



Some graphs are *undirected*.

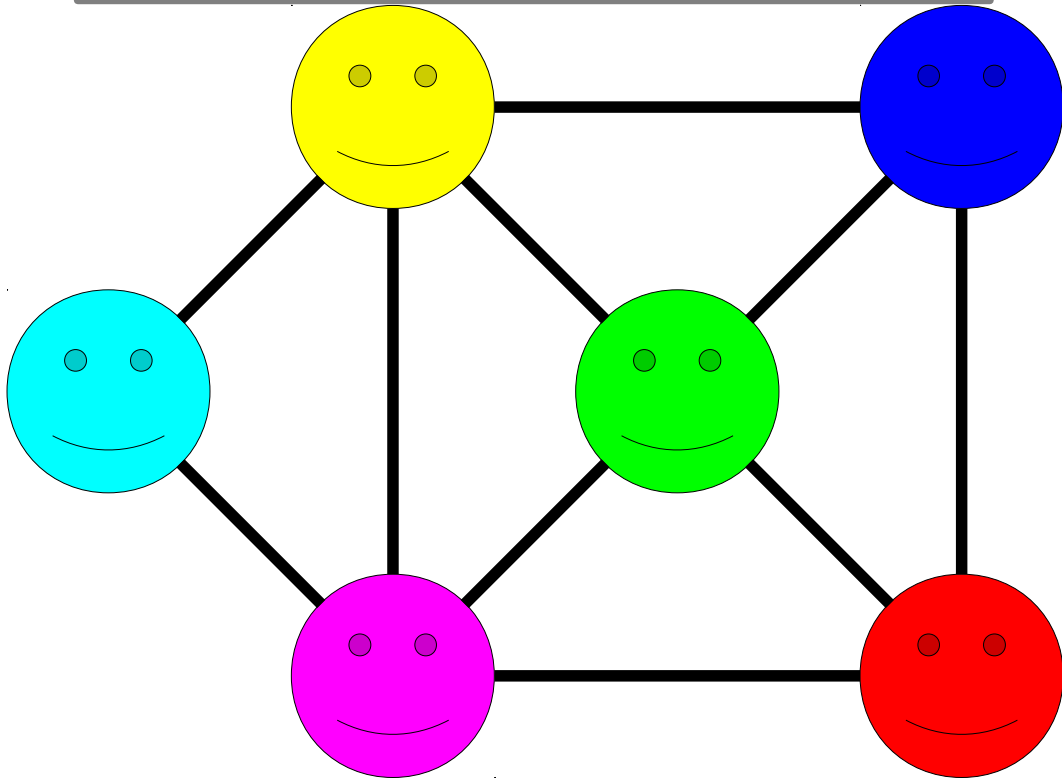


It sometimes helps to think of them as directed graphs with edges both ways.

How can we represent graphs in C++?

# Representing Graphs

We can represent a graph as a map from nodes to the list of nodes each node is connected to.



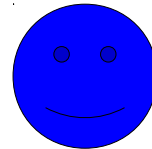
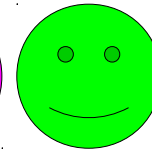
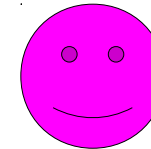
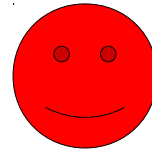
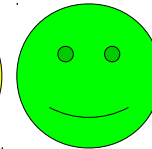
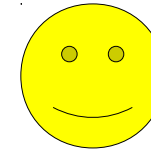
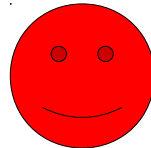
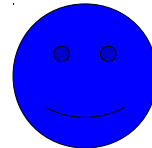
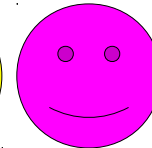
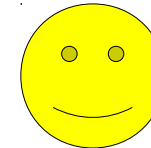
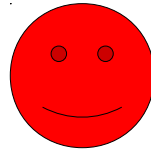
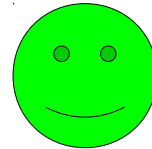
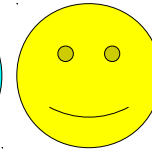
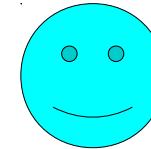
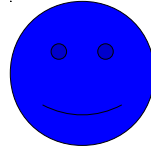
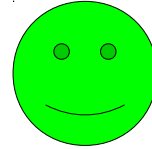
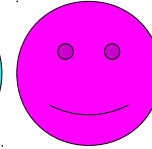
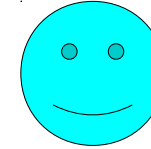
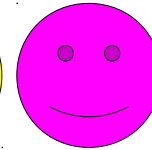
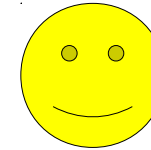
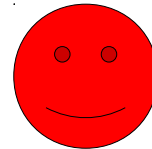
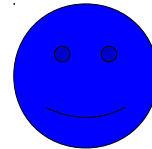
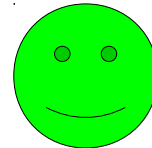
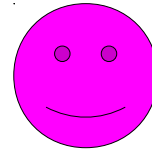
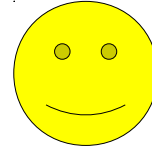
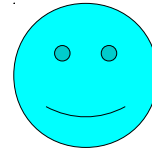
Map<**Node**, Vector<**Node**>>

**Node**

Vector<**Node**>

**Node**

**Adjacent To**





# Representing Graphs

- The approach we just saw is called an *adjacency list* in comes in a number of different forms:

Map<string, Vector<string>>

Map<string, Set<string>>

HashMap<string, HashSet<string>>

Vector<Vector<int>>

- The core idea is that we have some kind of mapping associating each node with its outgoing edges.

# Representing Graphs

The approach we just saw is called an *adjacency list* and comes in a number of different forms:

Map<string, Vector<string>>

Map<string, Set<string>>

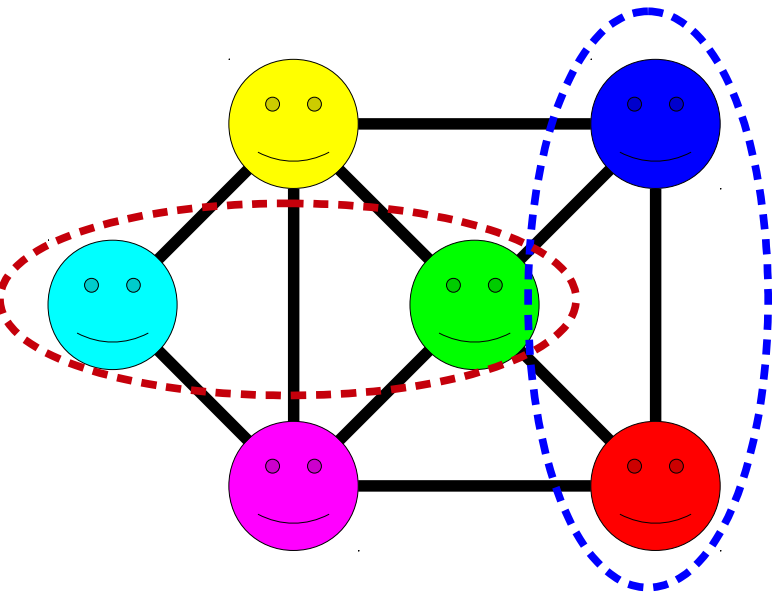
HashMap<string,

Vector<V

Question to ponder:  
where have you seen this  
before?

The core idea is that we have some kind of mapping associating each node with its outgoing edges.

# Other Graph Representations

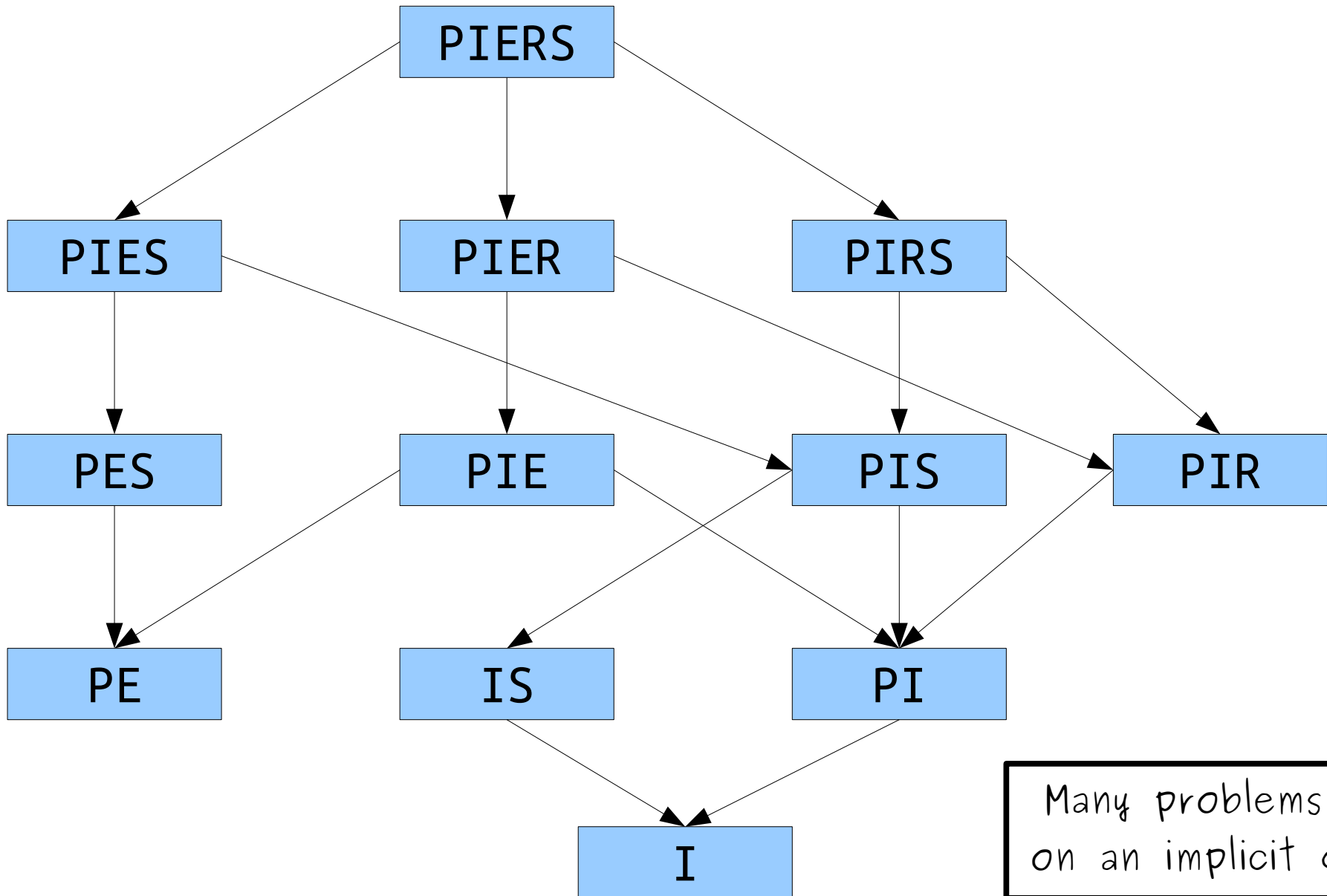


This representation is called an *adjacency matrix*.

For those of you in Math 51: if  $A$  is an adjacency matrix for a graph  $G$ , what is the significance of the matrix  $A^2$ ?

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



























# Other Representations



You'll find graphs just  
about everywhere you look.

They're an *extremely* versatile and  
powerful abstraction to be aware of.

Going forward, unless stated otherwise, assume we're using an *adjacency list*.

<i>Node</i>	<i>Adjacent To</i>
	 
	   
	   
	   
	  
	  

**Time-Out for Announcements!**

# Assignment 6

- Assignment 6 (***MiniBrowser***) is due this Friday.
  - If you're following our suggested timetable, you should be done with the browser history at this point and be working on Autocomplete or Line Manager.
- Have questions? Stop by the LaIR/CLaIR, go on Piazza, ask your section leader, or stop by Kate or Keith's office hours!



# Midterm Regrades

- As a reminder, midterm regrade requests are due this Wednesday.
  - Check your inbox for an email from Kate with details.
- We've posted a testing harness up on the course website that you can use to edit and run your solutions.

**Back to CS106B!**

# Traversing Graphs

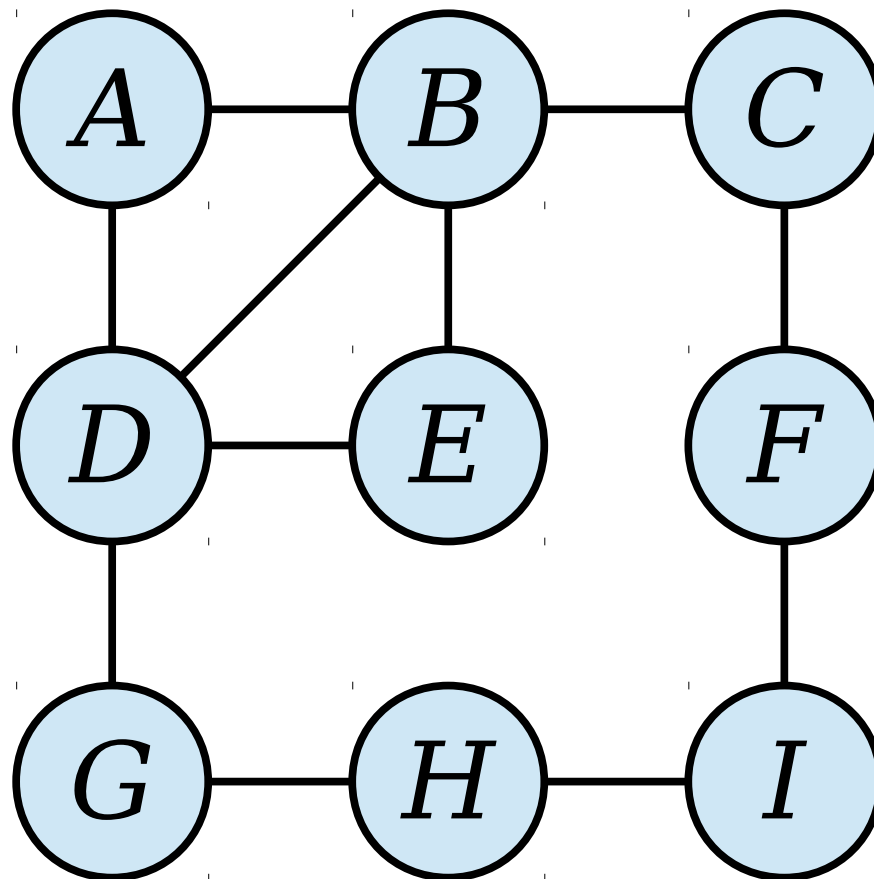
# Iterating over a Graph

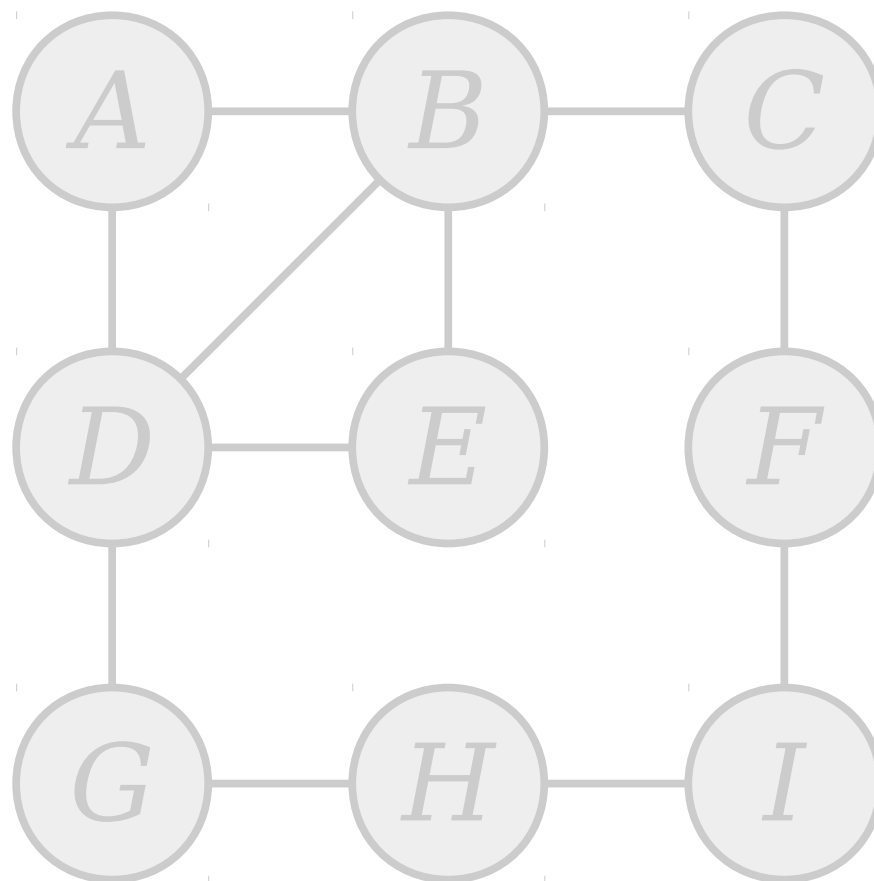
- In a singly-linked list, there's pretty much one way to iterate over the list: start at the front and go forward!
- In a binary search tree, there are many traversal strategies:
  - An ***inorder traversal*** that produces all the elements in sorted order.
  - A ***postorder traversal*** used to delete all the nodes in the BST.
- There are *many* ways to iterate over a graph, each of which have different properties.

# Where We're Going

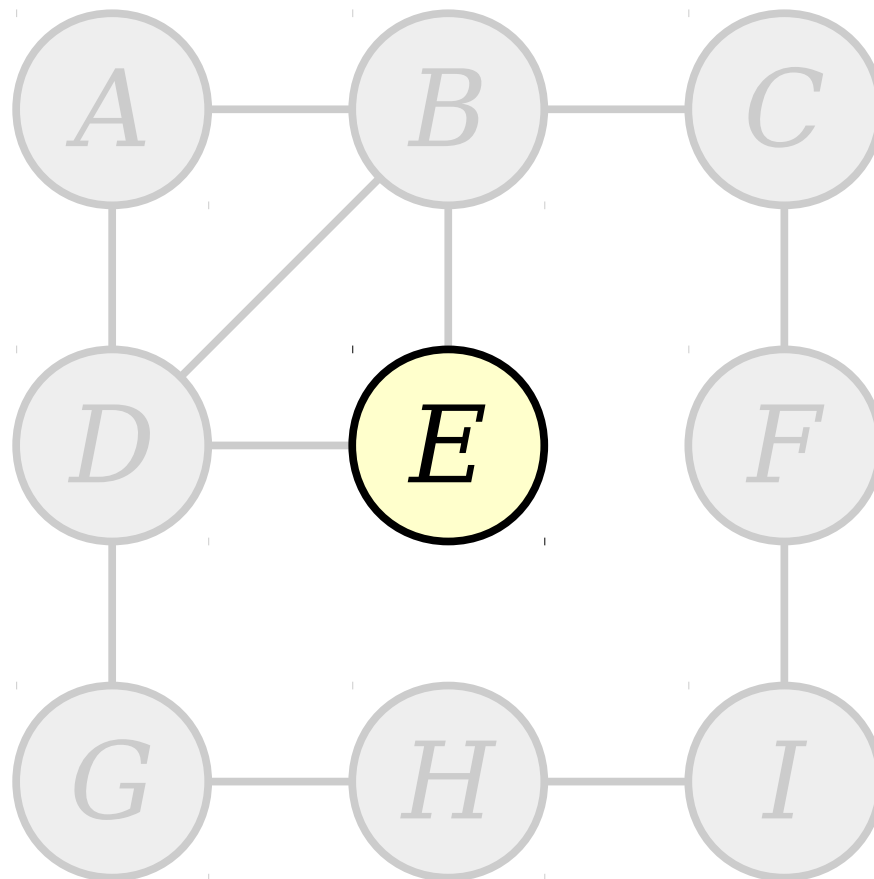
- Today, we'll cover ***breadth-first search***, which can be used to find shortest paths in a graph.
- On Wednesday, we'll see ***depth-first search***, which can be used to order prerequisites and recover interesting structures.
- There are many other approaches as well. Take CS161 or CS221 for details!

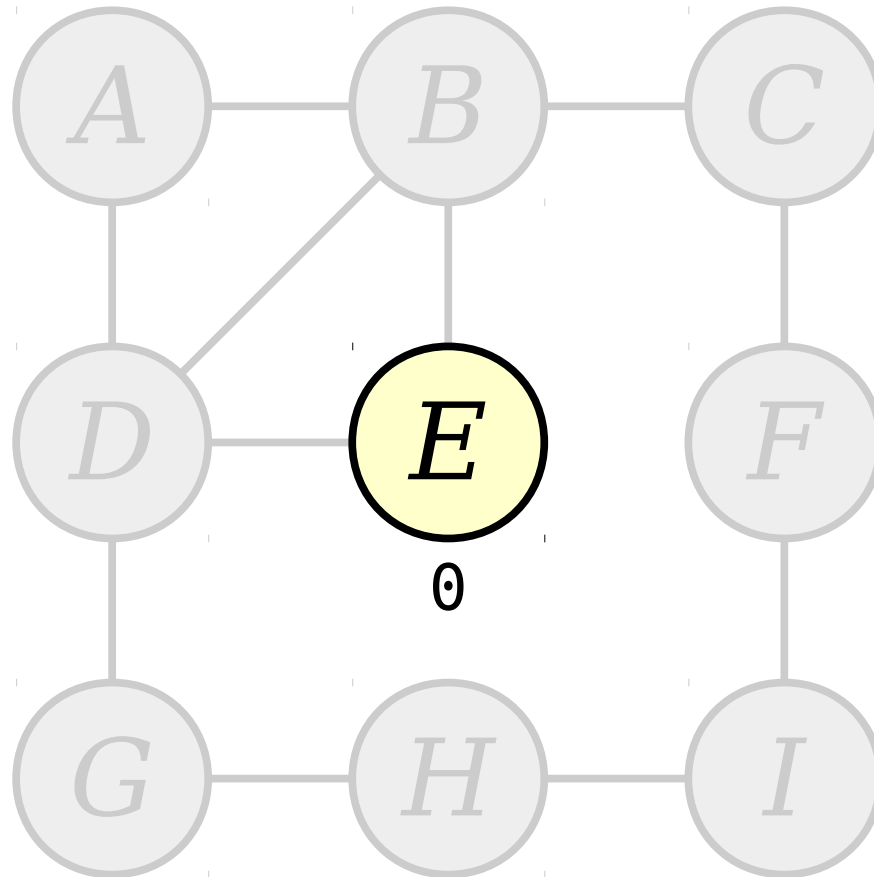
# An Initial Search Strategy



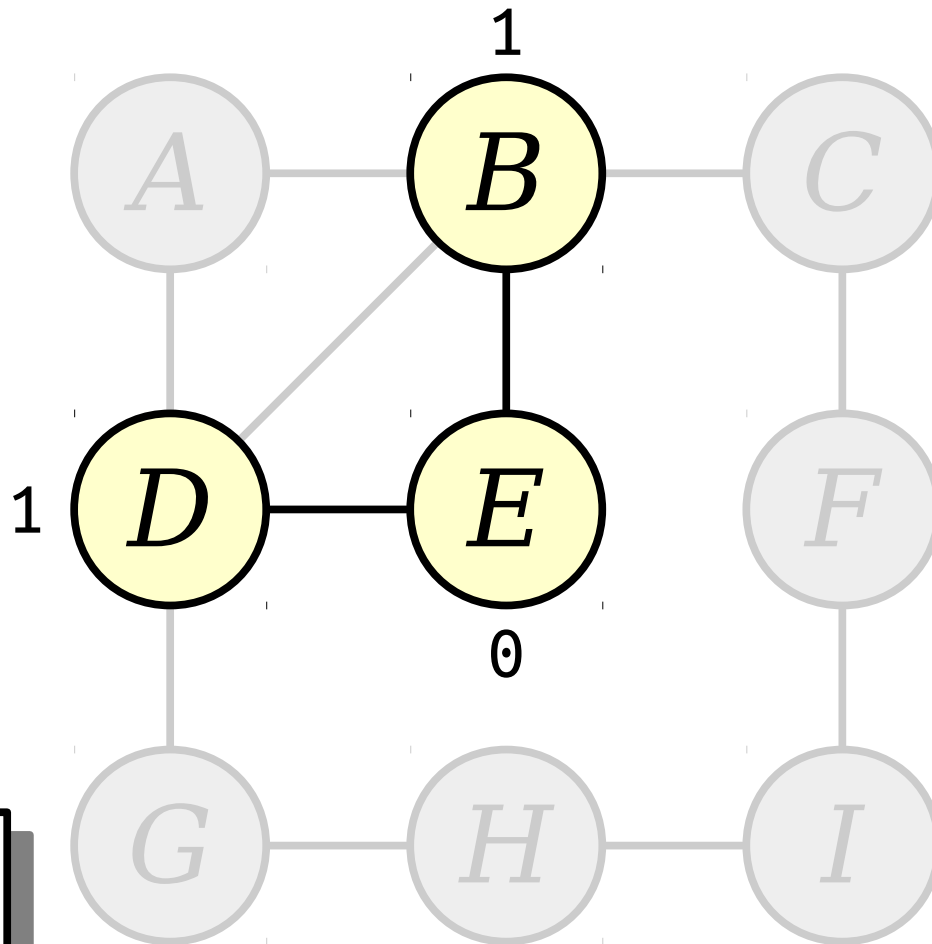




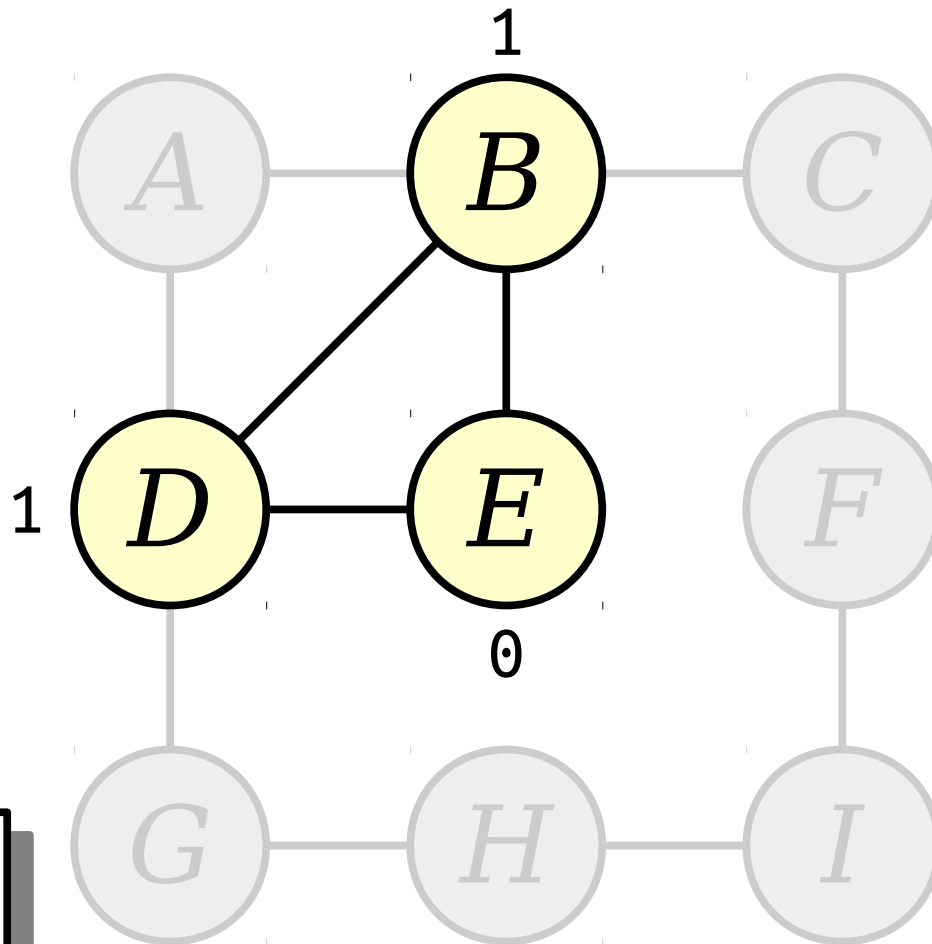




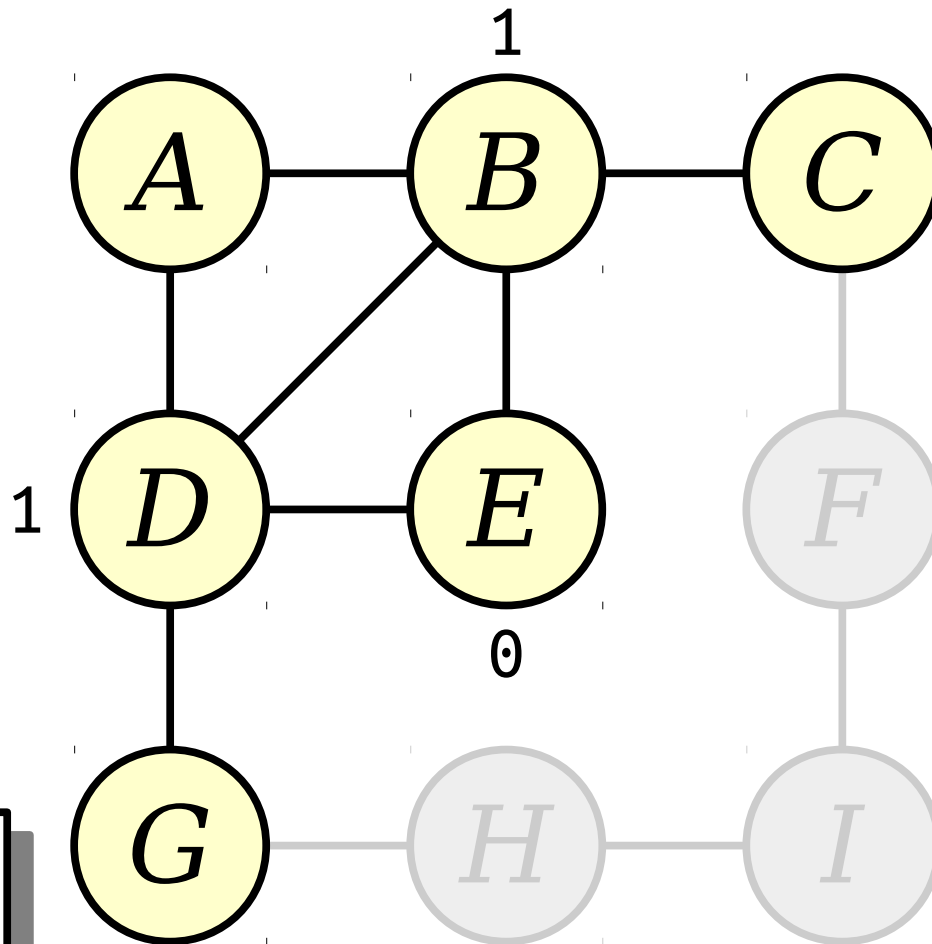
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



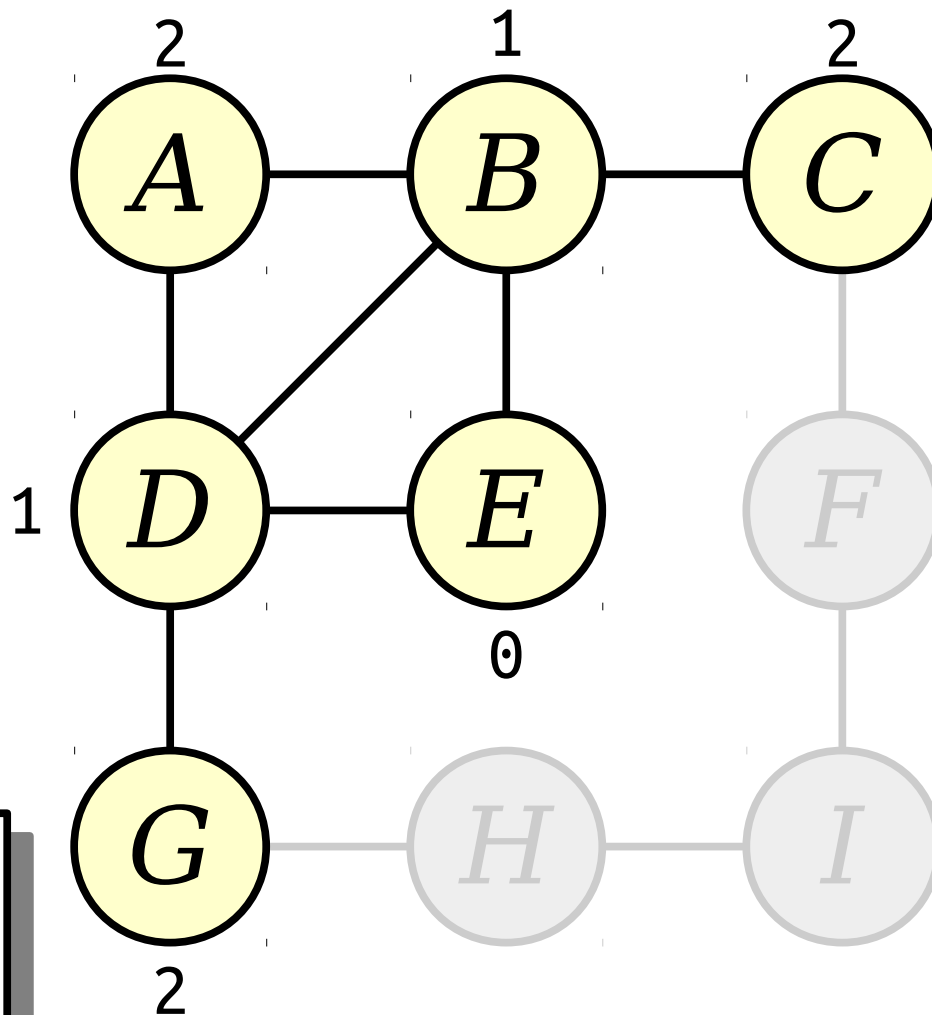
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



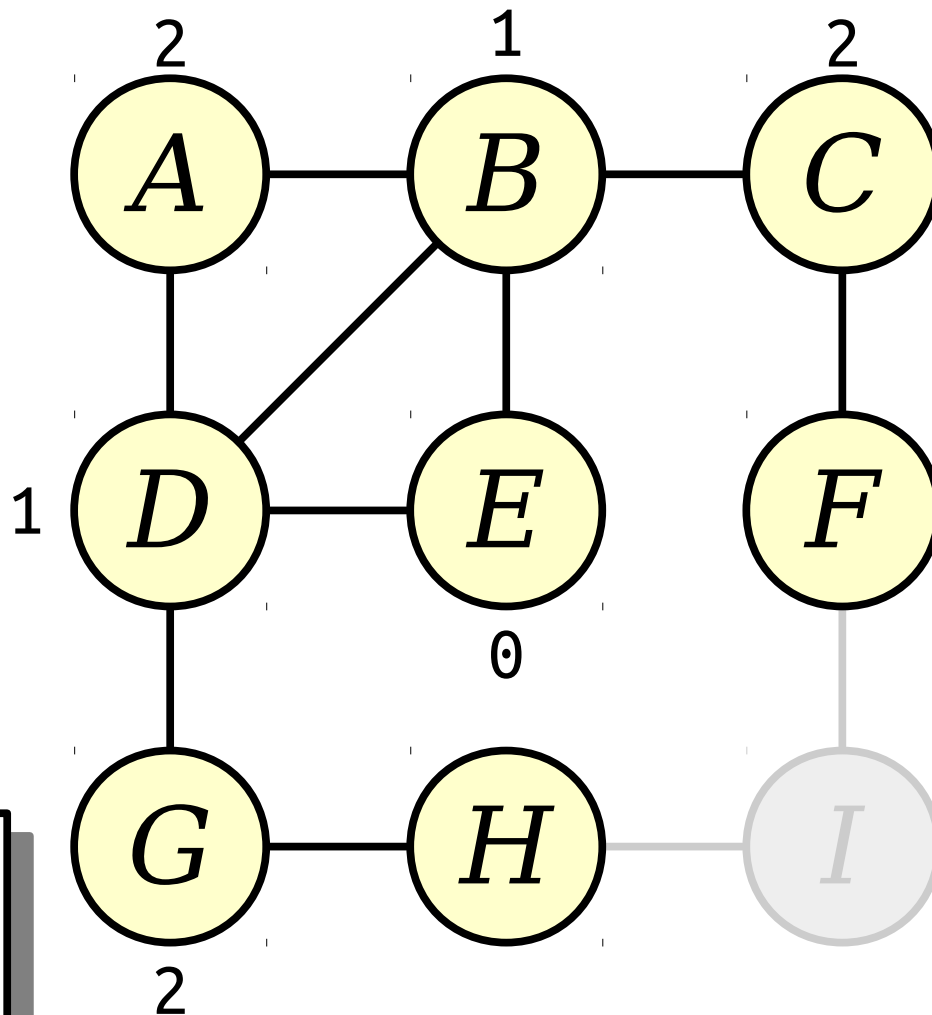
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



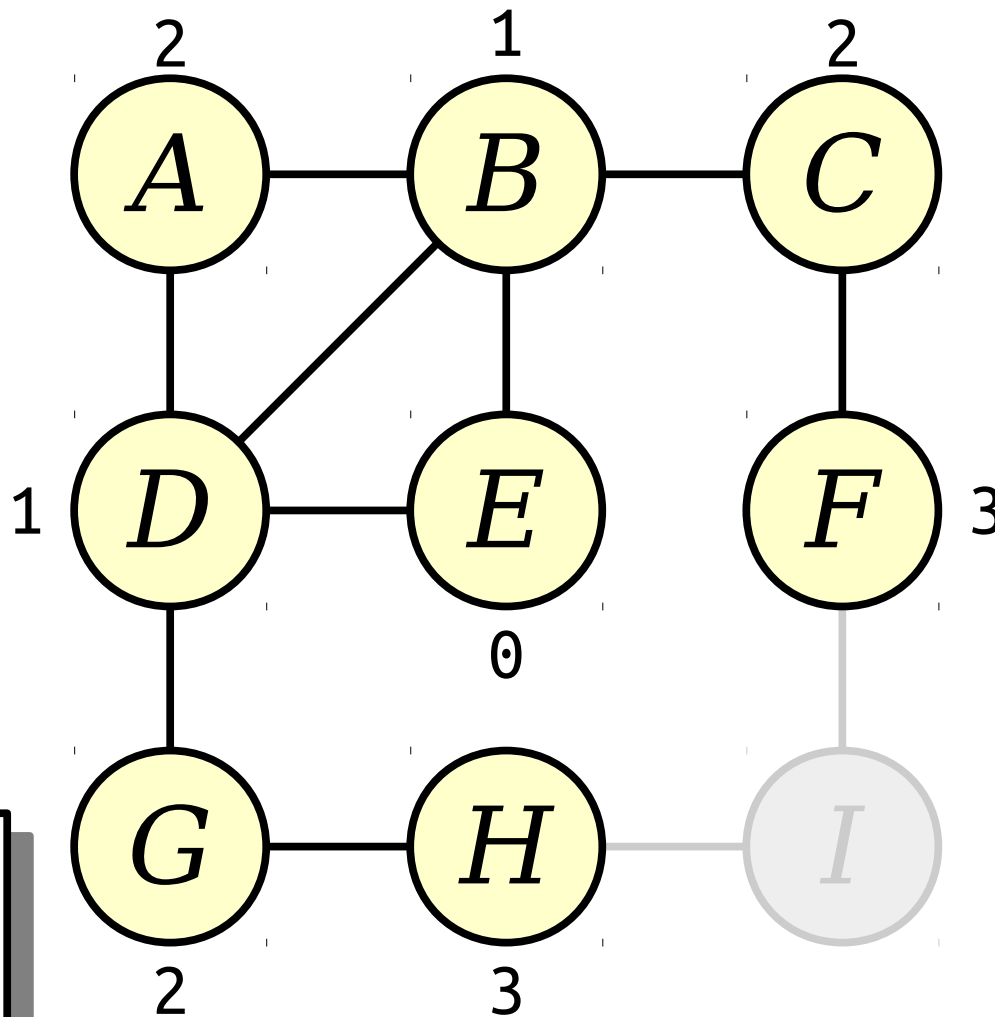
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.

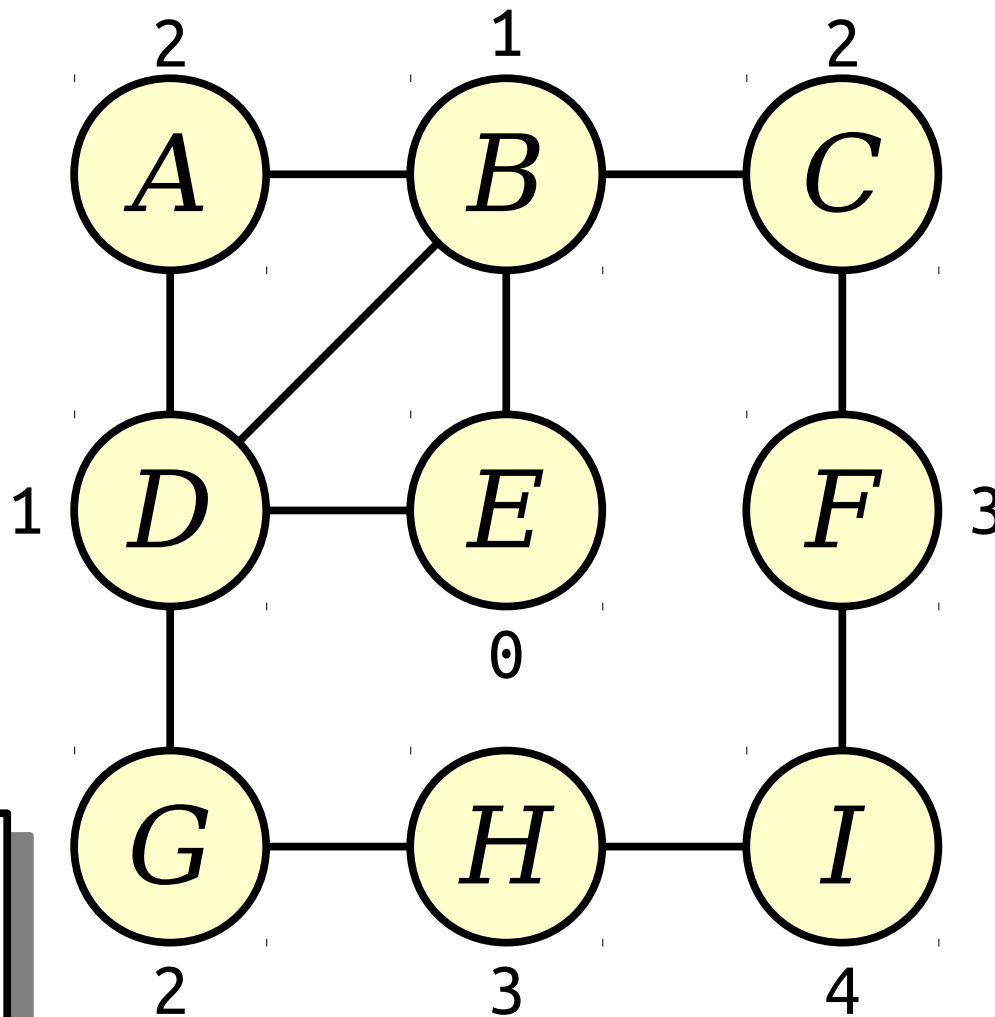


**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



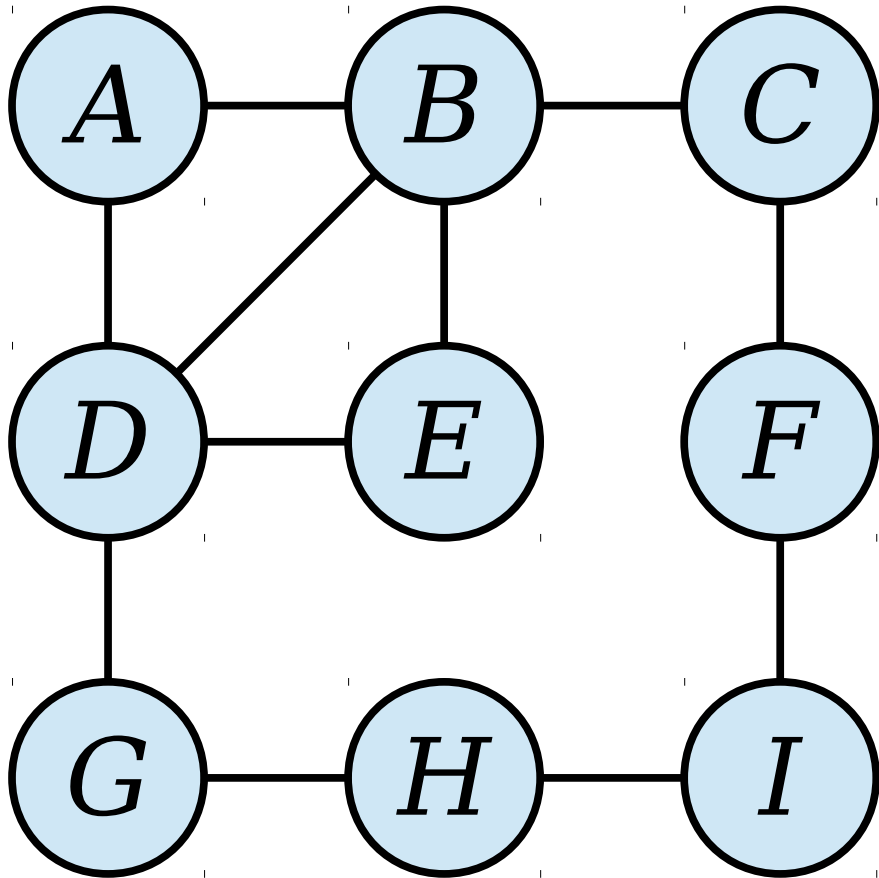
**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.



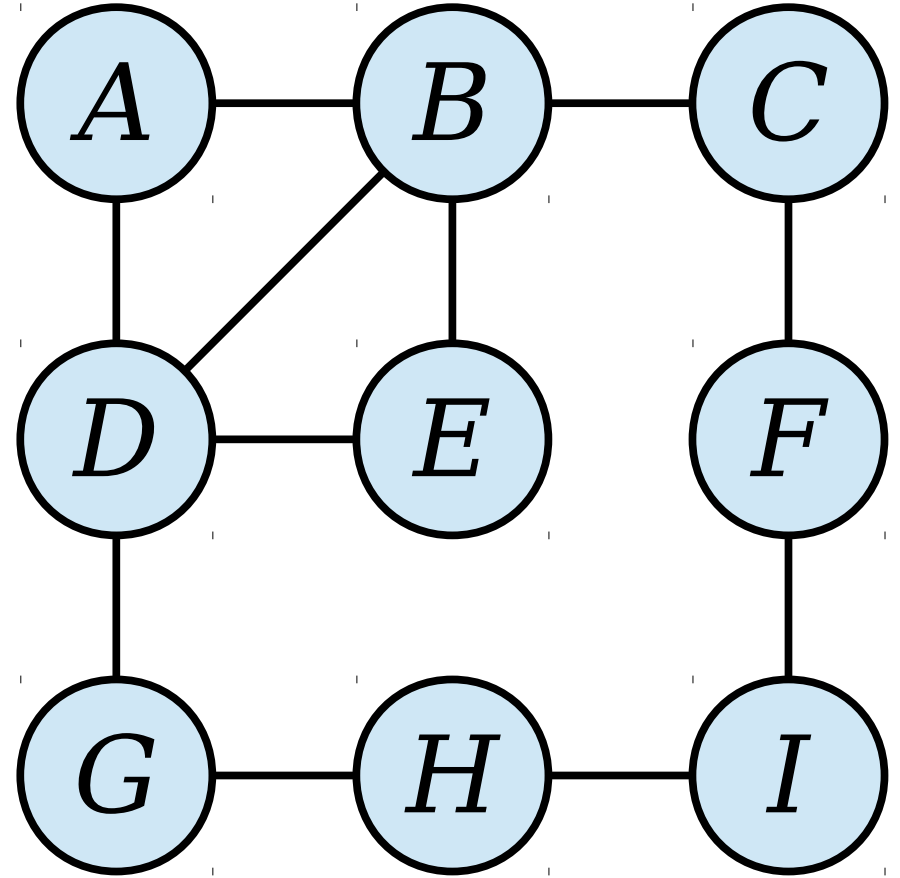


**Core idea:** Find everything one hop away from the start, then two hops away, then three hops away, etc.

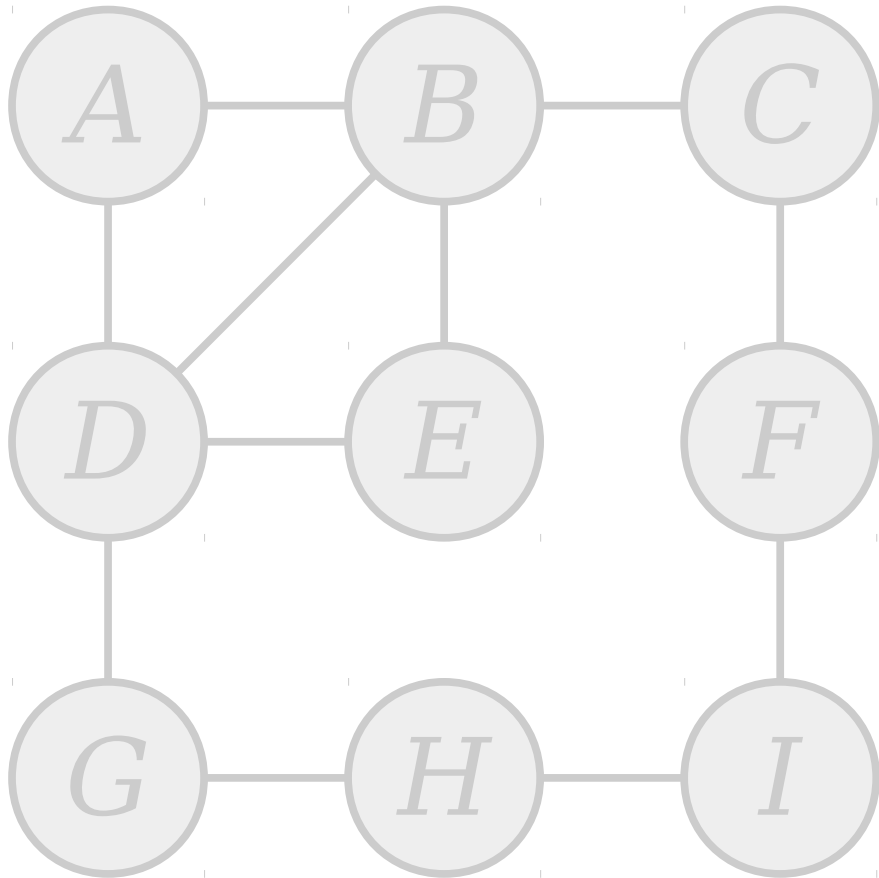
Implementing this Idea



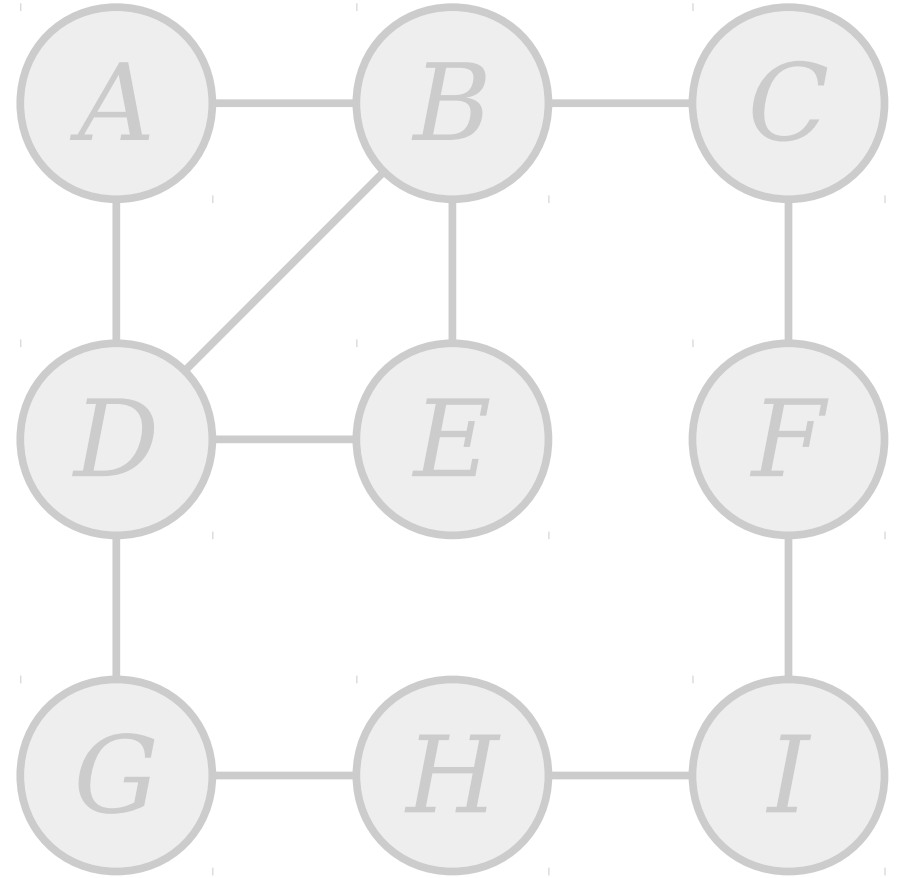
Visit nodes in ascending order of distance from the start node *E*.



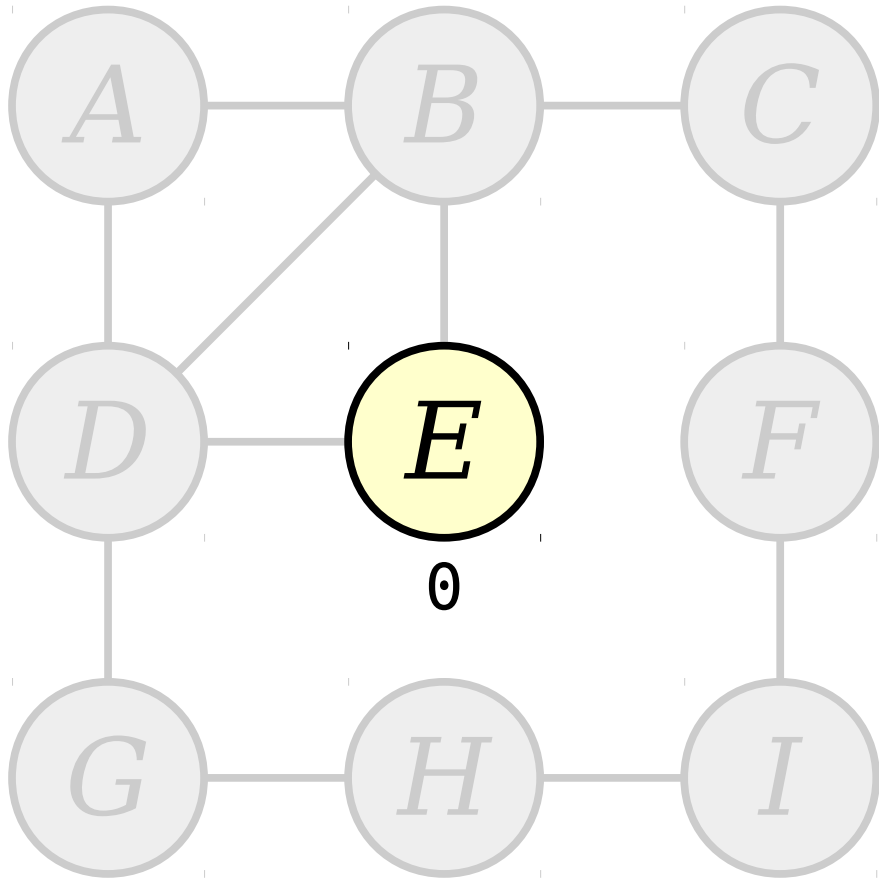
Load newly-discovered nodes into a queue.



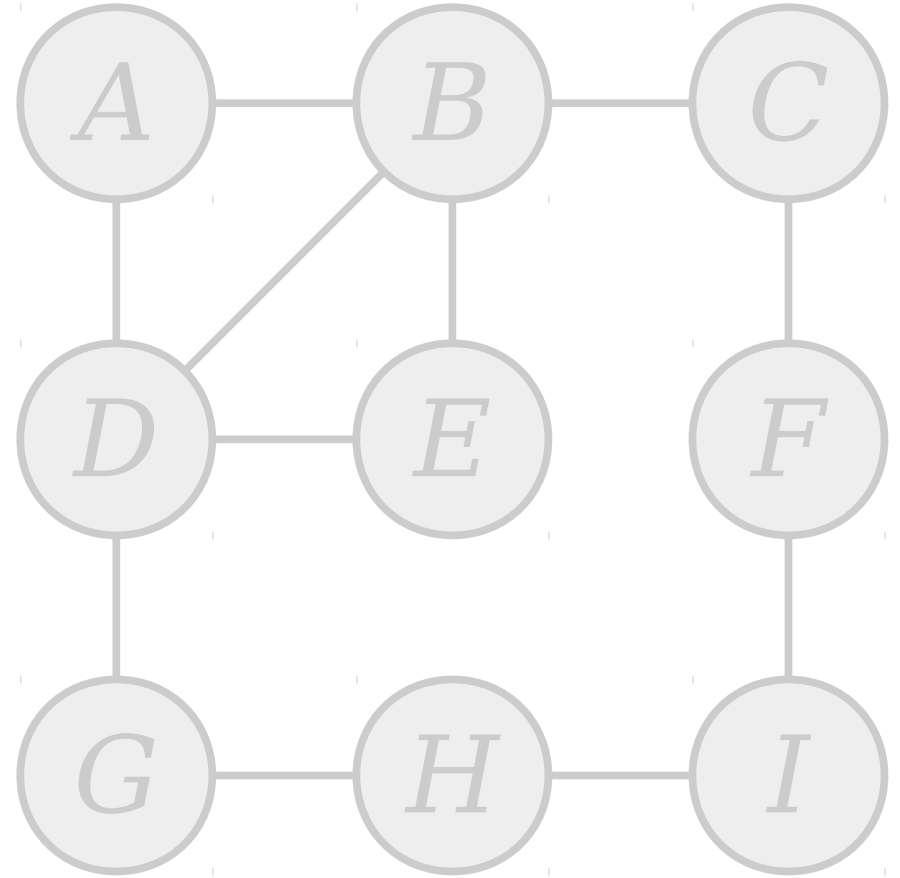
Visit nodes in ascending order of distance from the start node *E*.



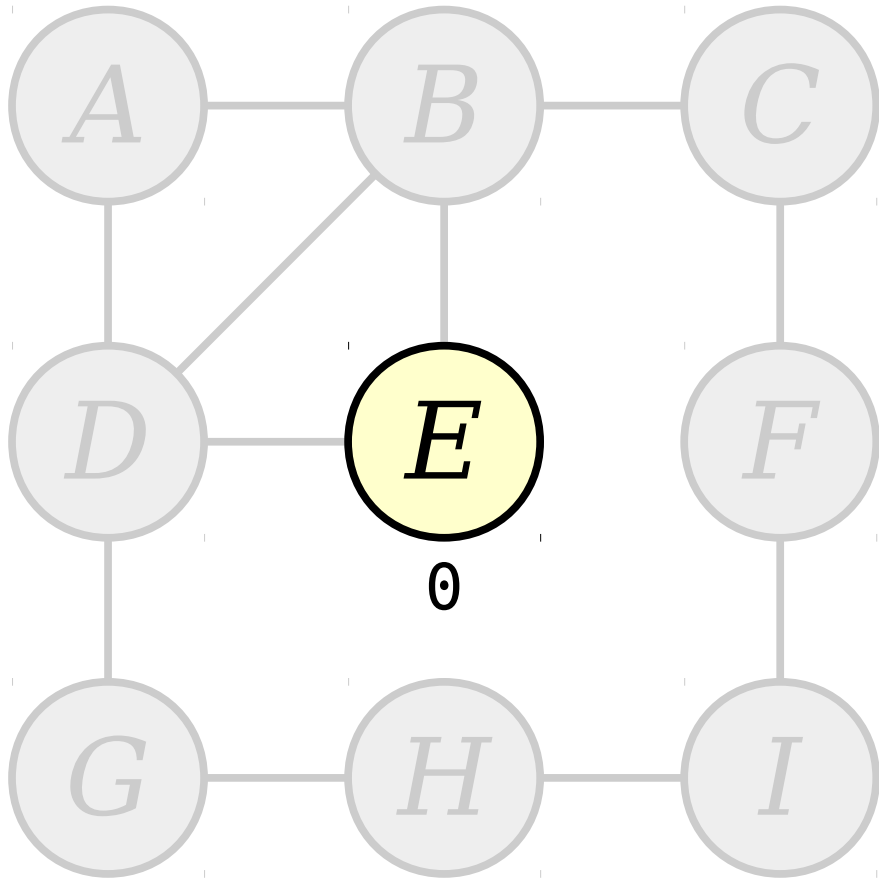
Load newly-discovered nodes into a queue.



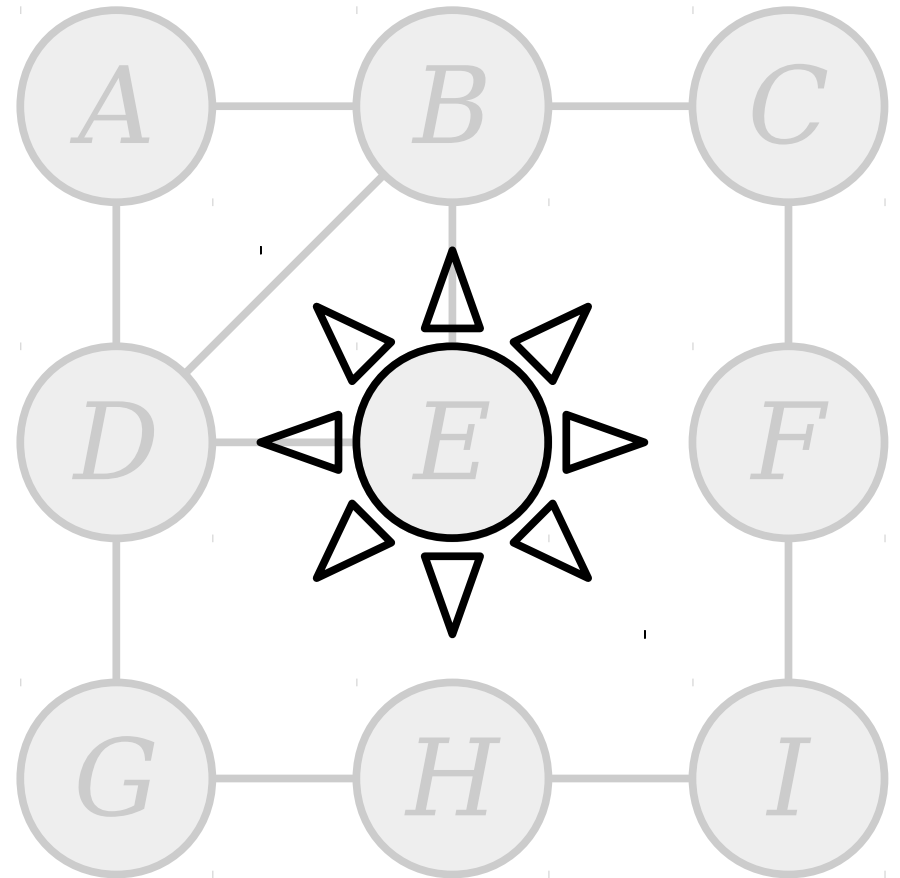
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

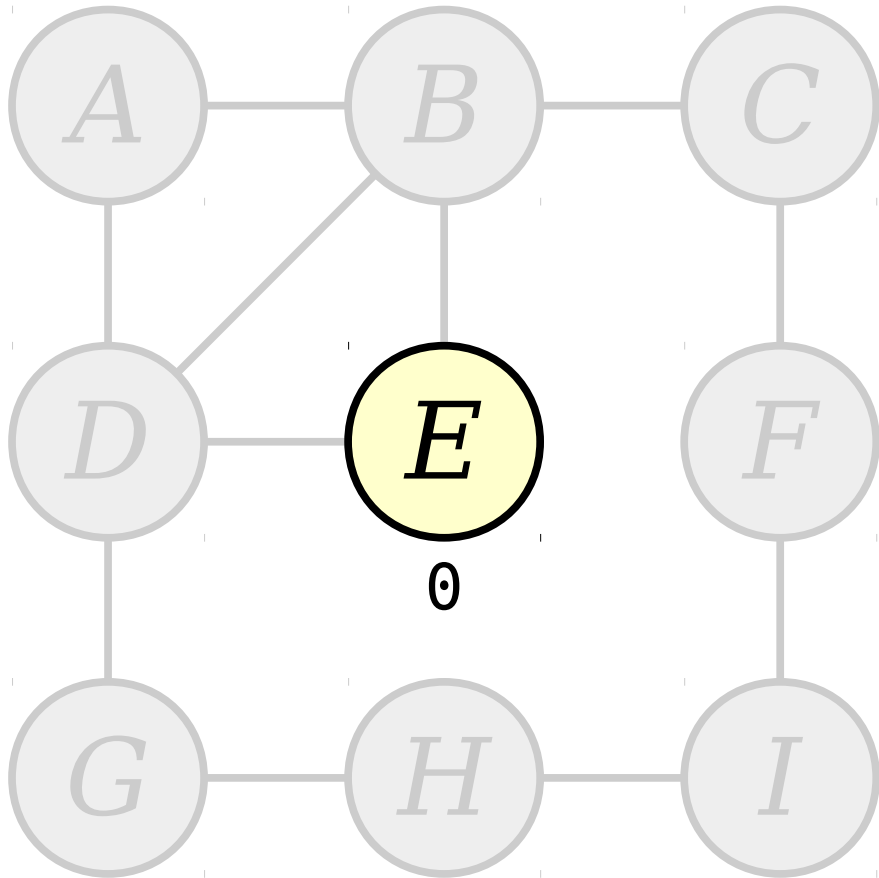


Visit nodes in ascending order of distance from the start node *E*.

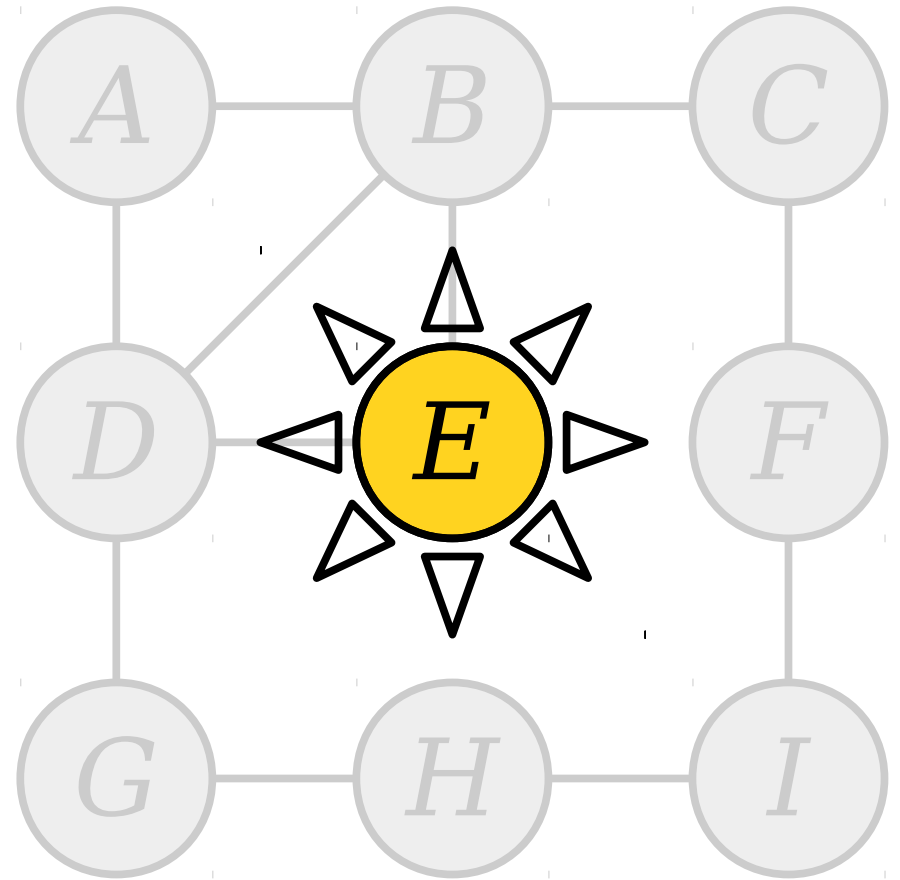


Queue: 

Load newly-discovered nodes into a queue.

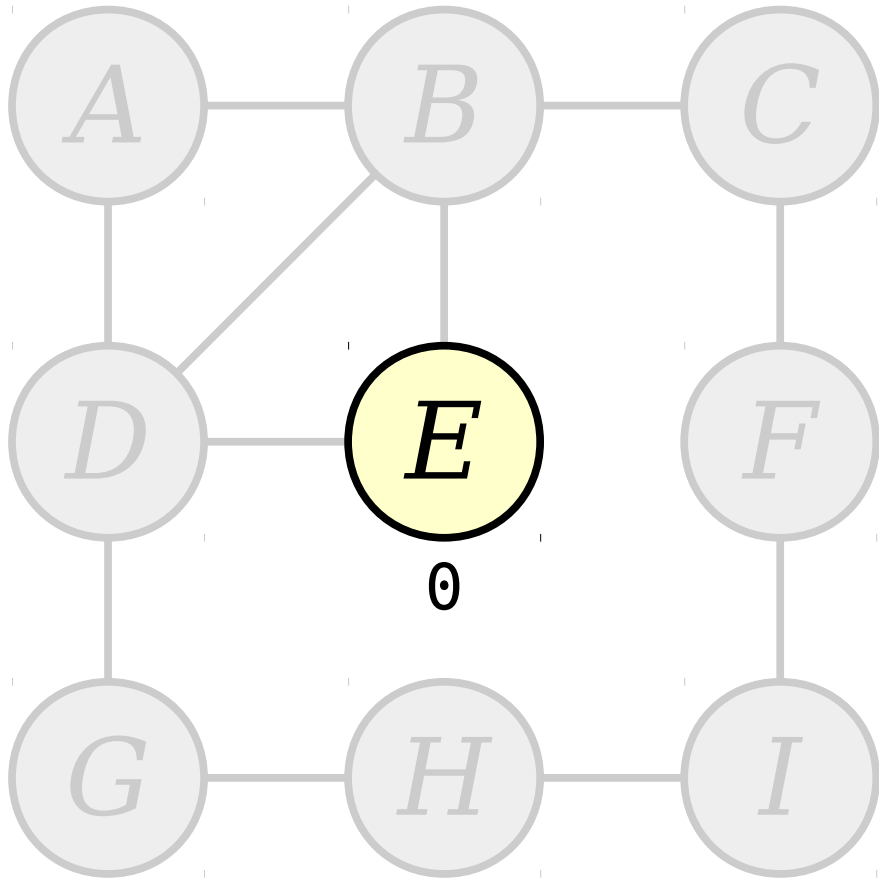


Visit nodes in ascending order of distance from the start node *E*.

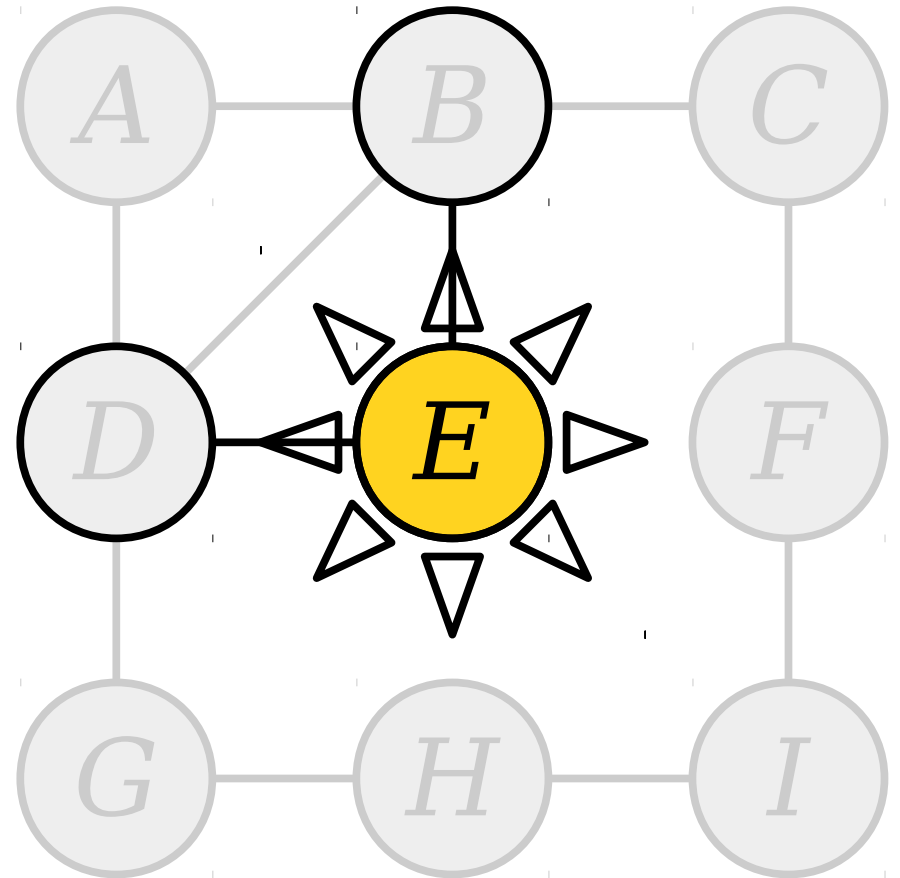


Queue:

Load newly-discovered nodes into a queue.



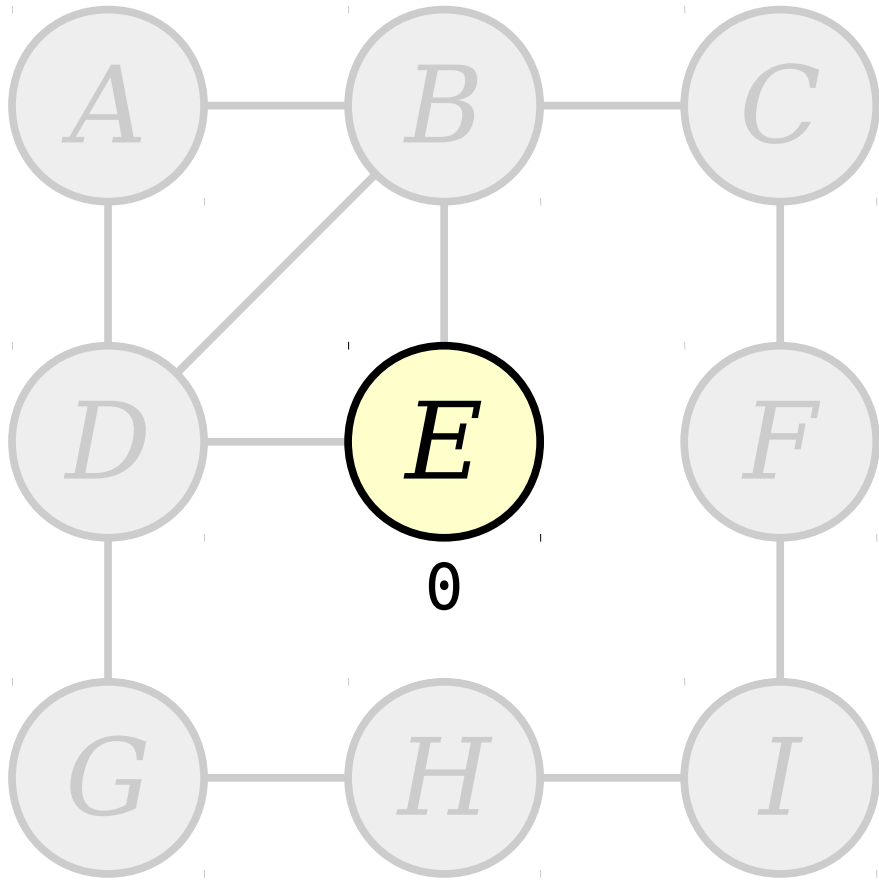
Visit nodes in ascending order of distance from the start node *E*.



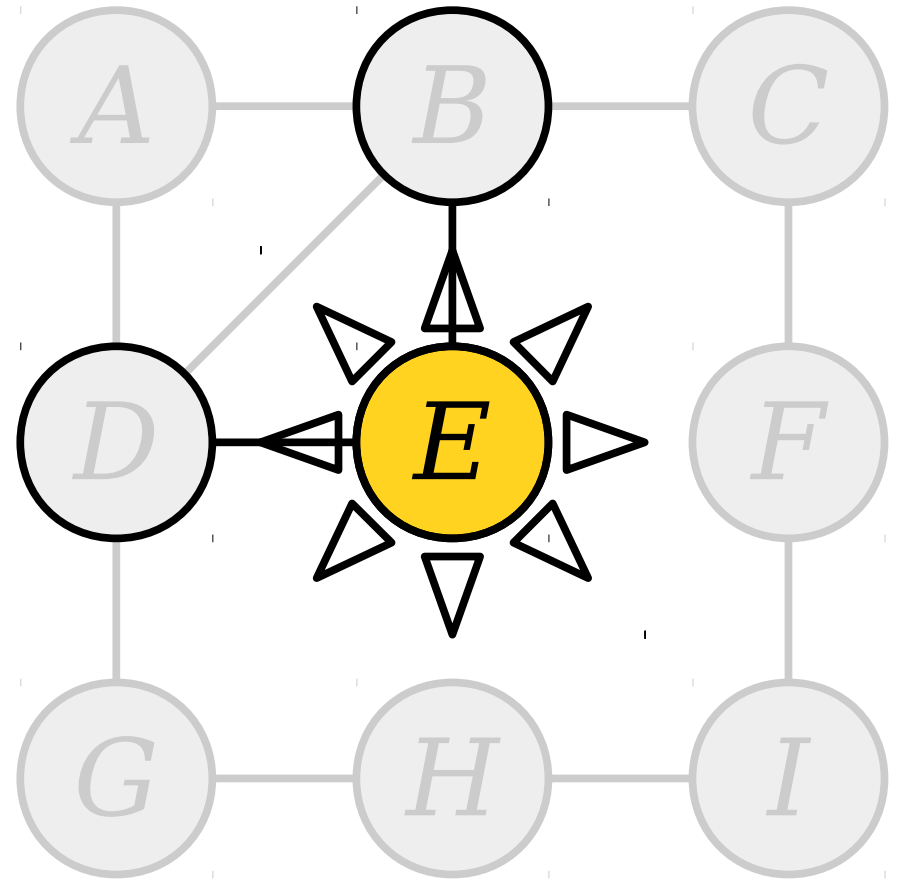
Queue:

Load newly-discovered nodes into a queue.



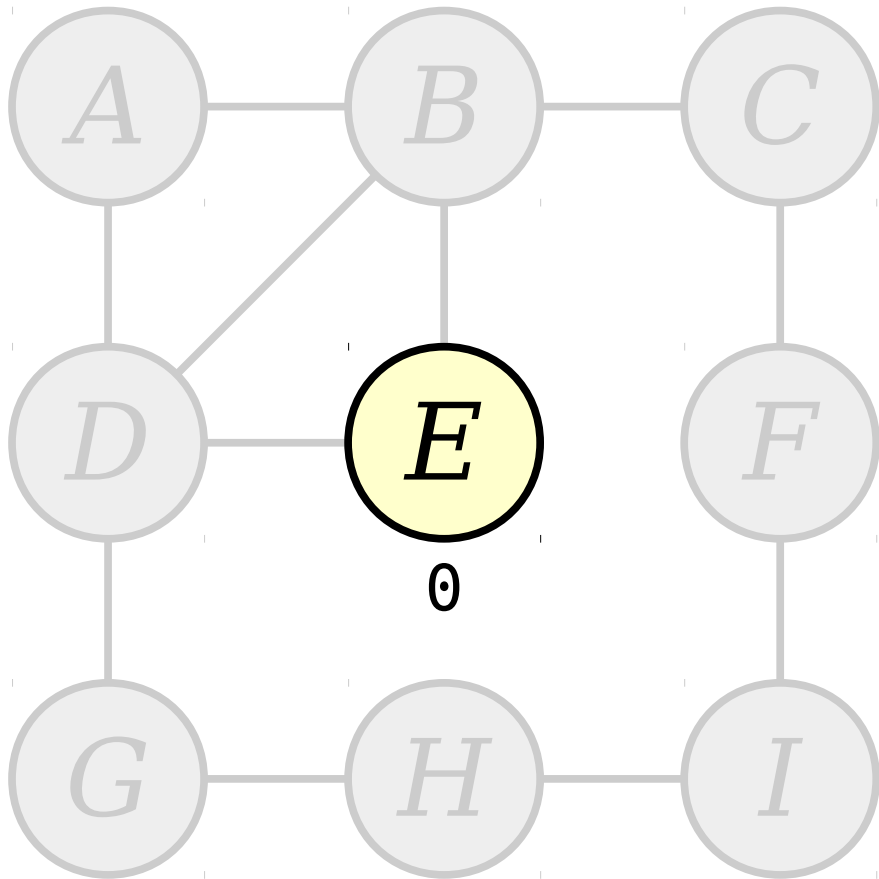


Visit nodes in ascending order of distance from the start node *E*.

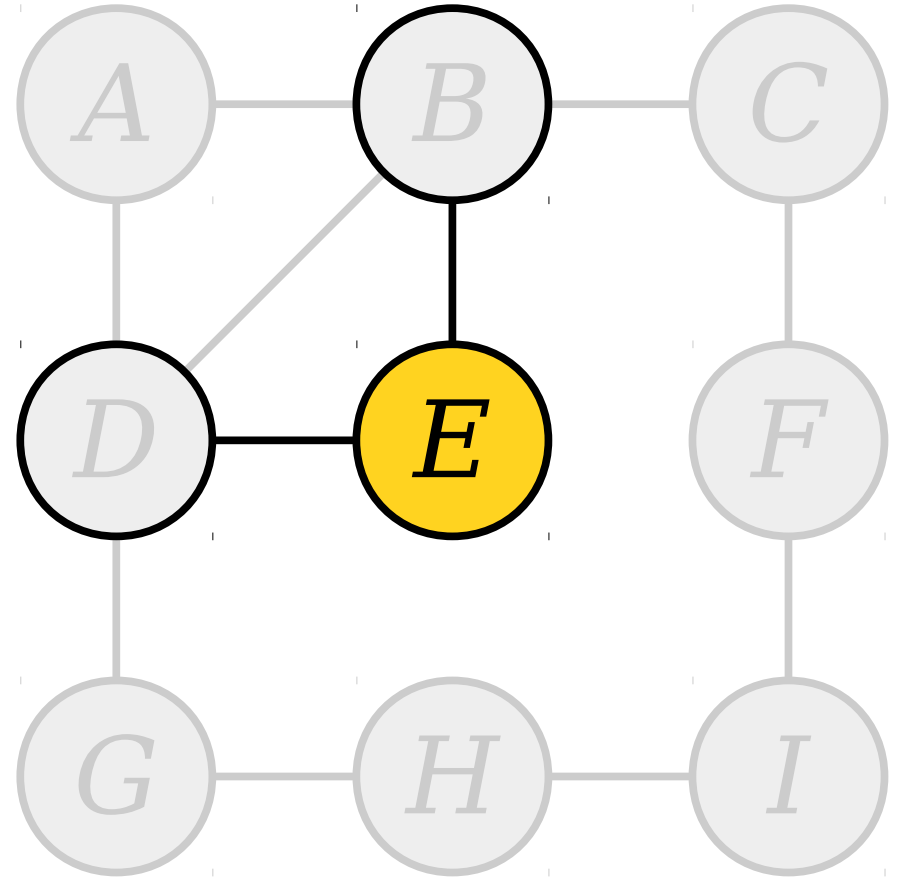


Queue: *D* *B*

Load newly-discovered nodes into a queue.

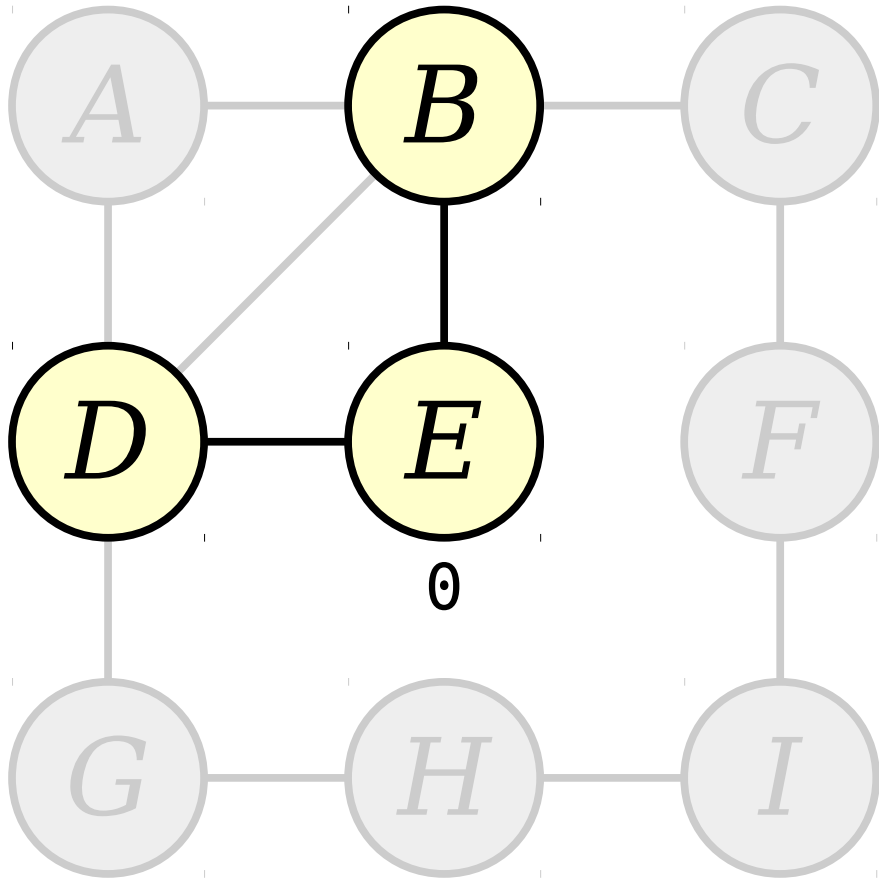


Visit nodes in ascending order of distance from the start node *E*.

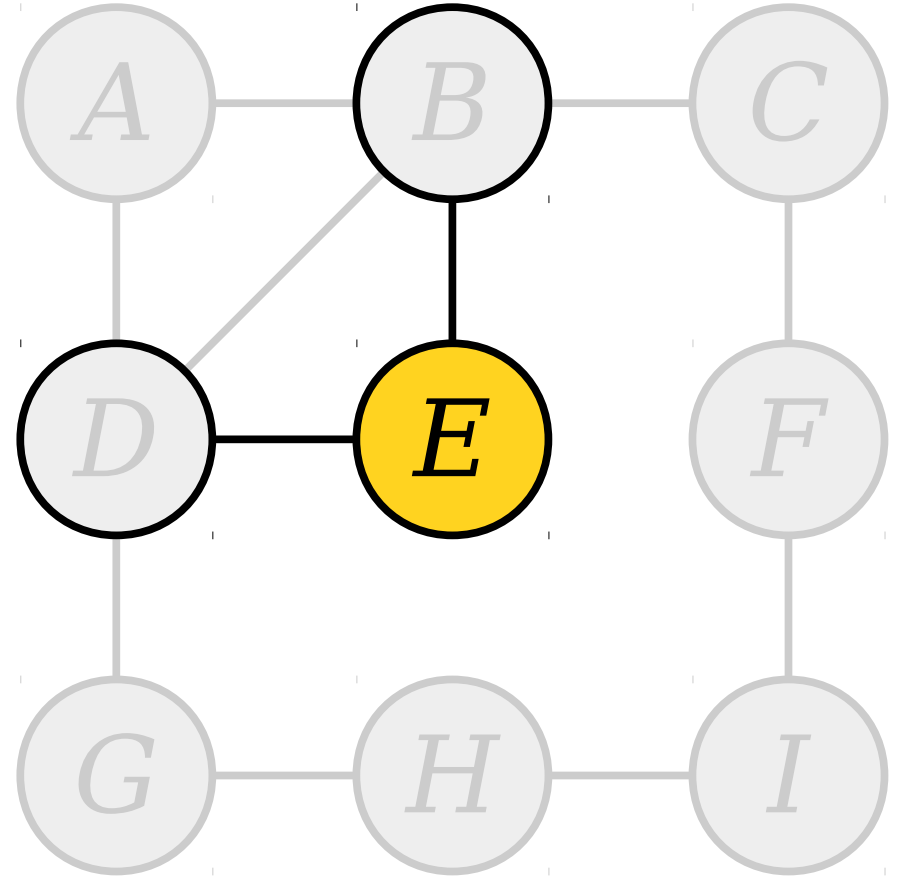


Queue: *D* *B*

Load newly-discovered nodes into a queue.

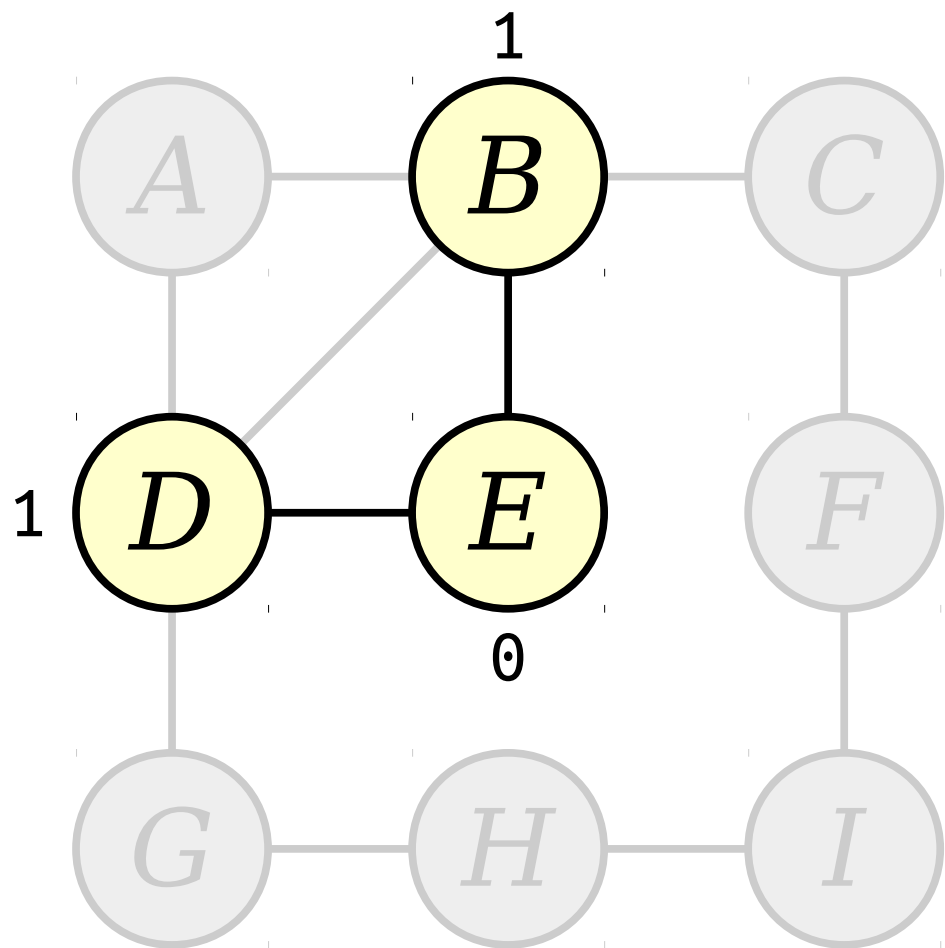


Visit nodes in ascending order of distance from the start node *E*.

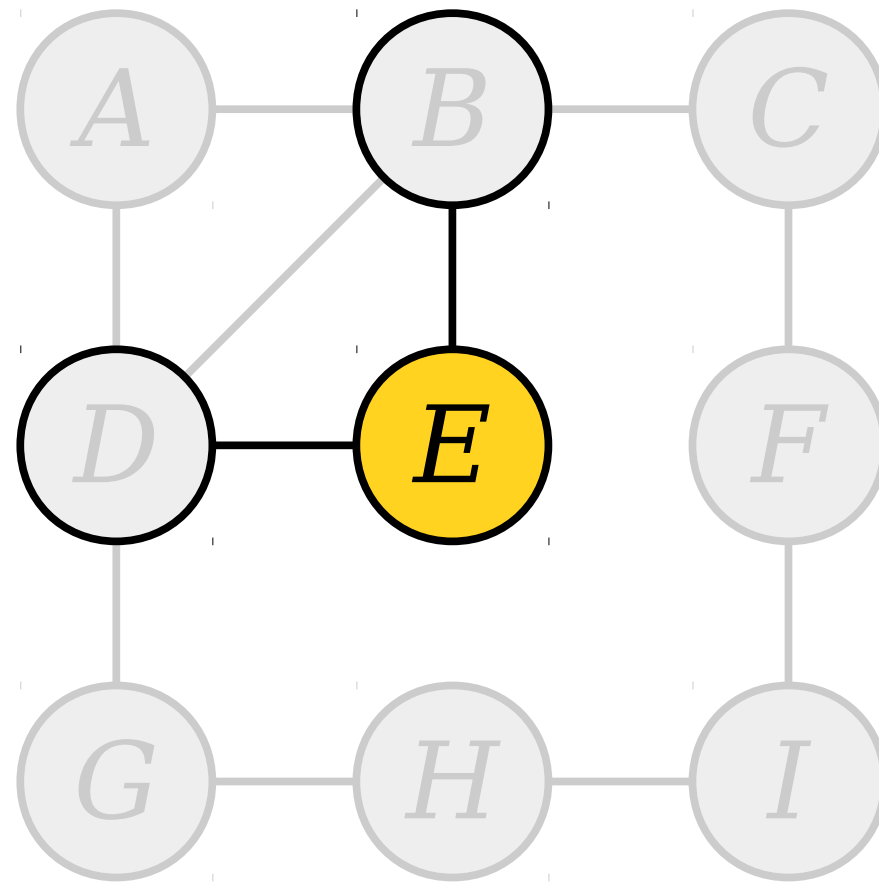


Queue: *D* *B*

Load newly-discovered nodes into a queue.

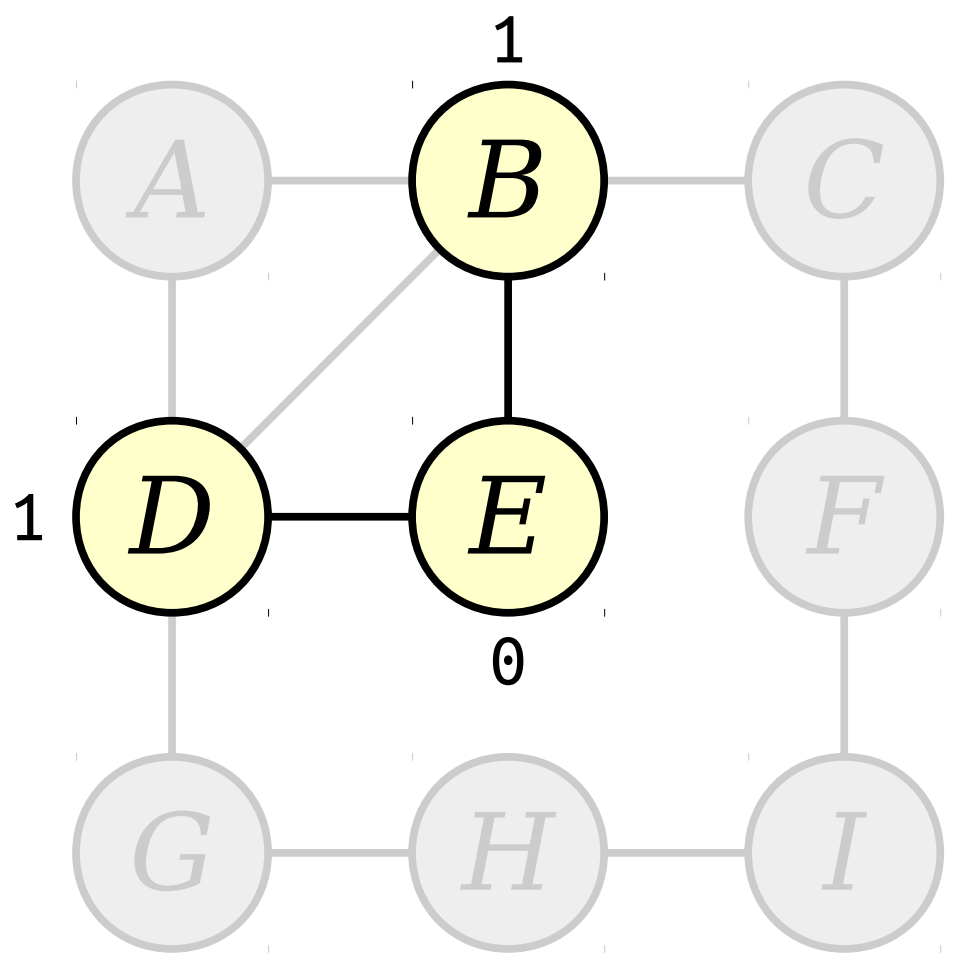


Visit nodes in ascending order of distance from the start node  $E$ .

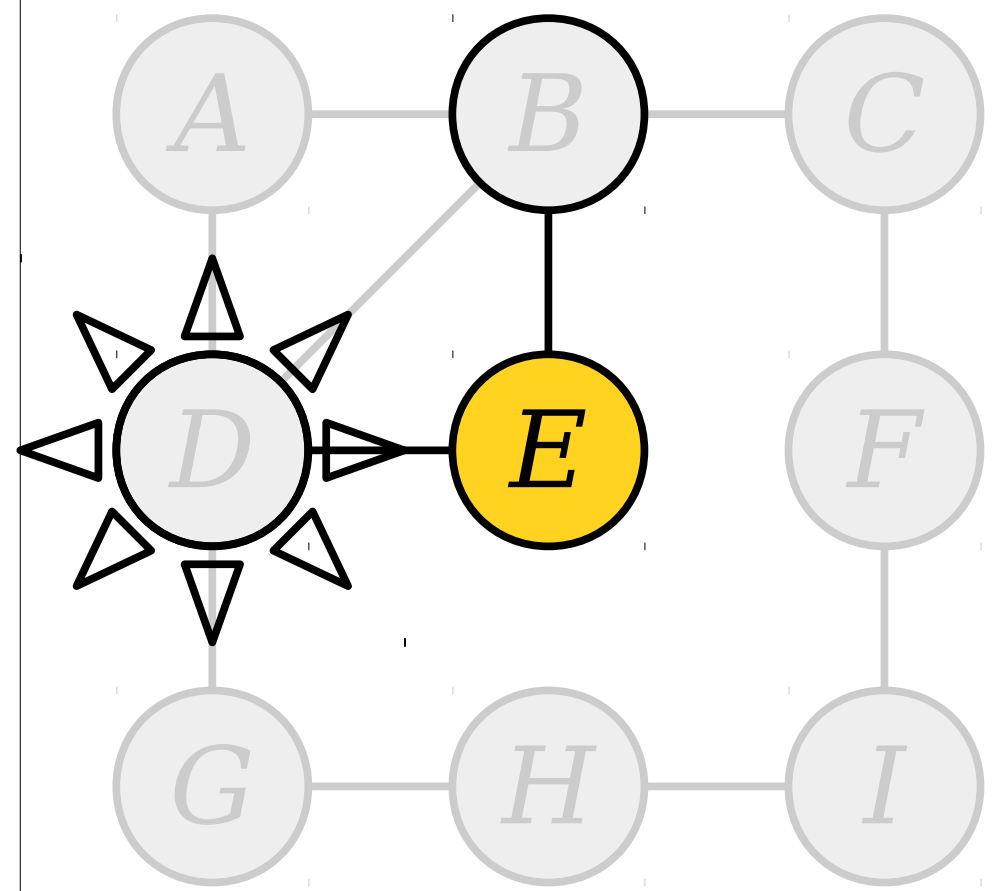


Queue:  $D$   $B$

Load newly-discovered nodes into a queue.

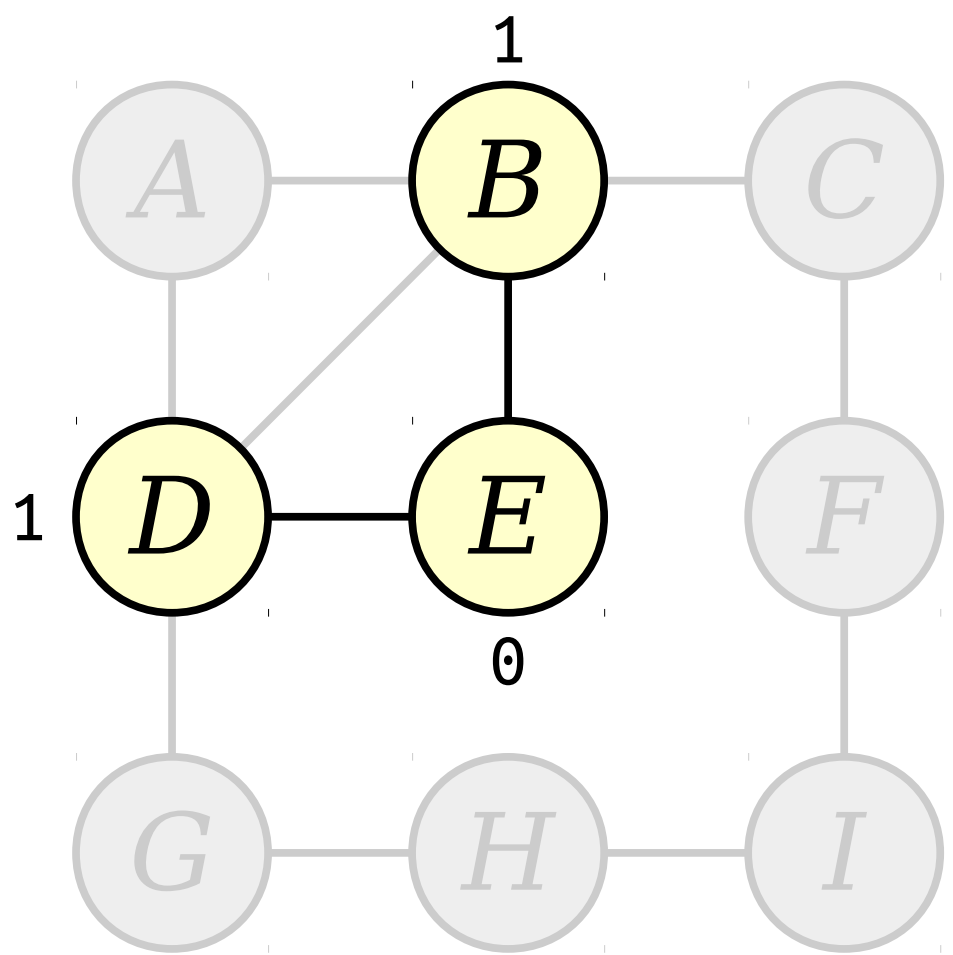


Visit nodes in ascending order of distance from the start node *E*.

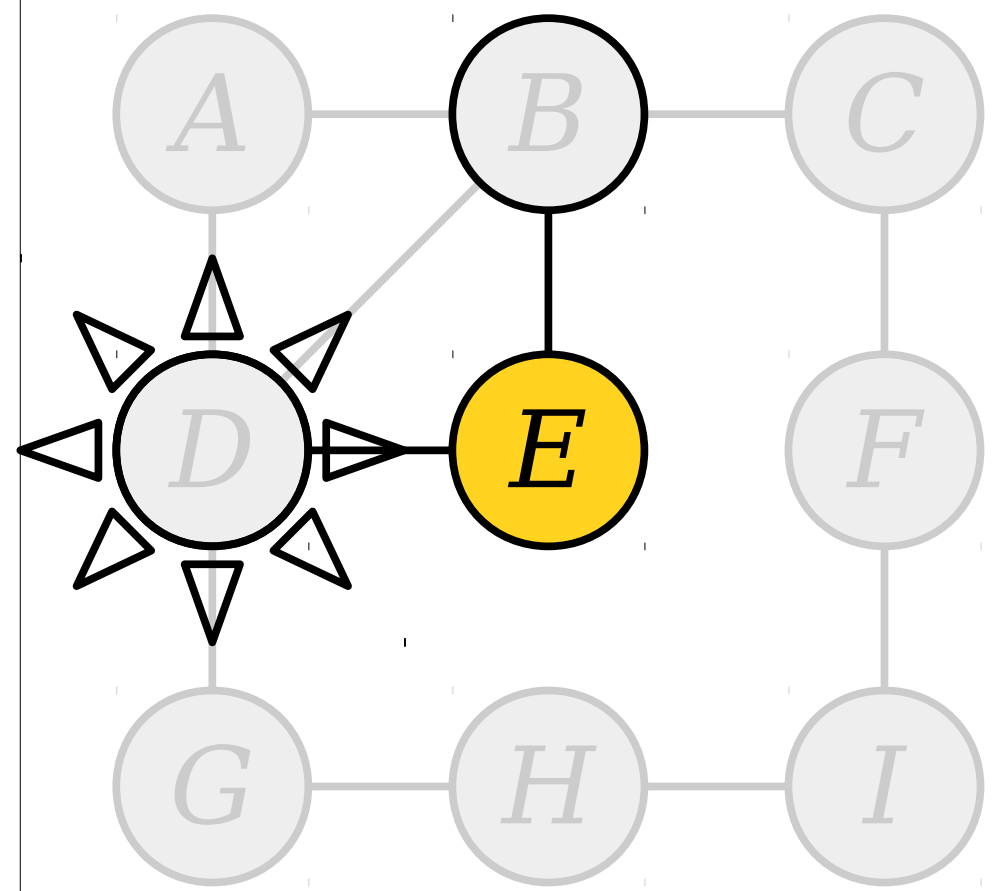


Queue: *D* *B*

Load newly-discovered nodes into a queue.

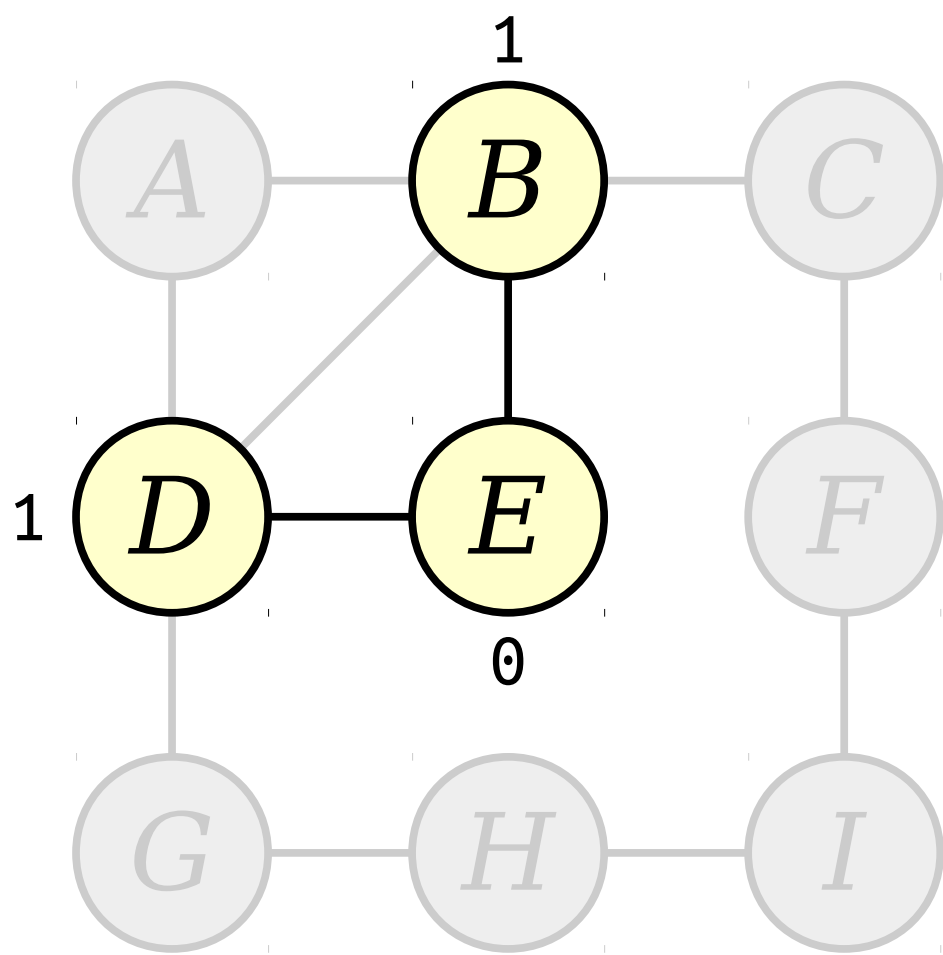


Visit nodes in ascending order of distance from the start node *E*.

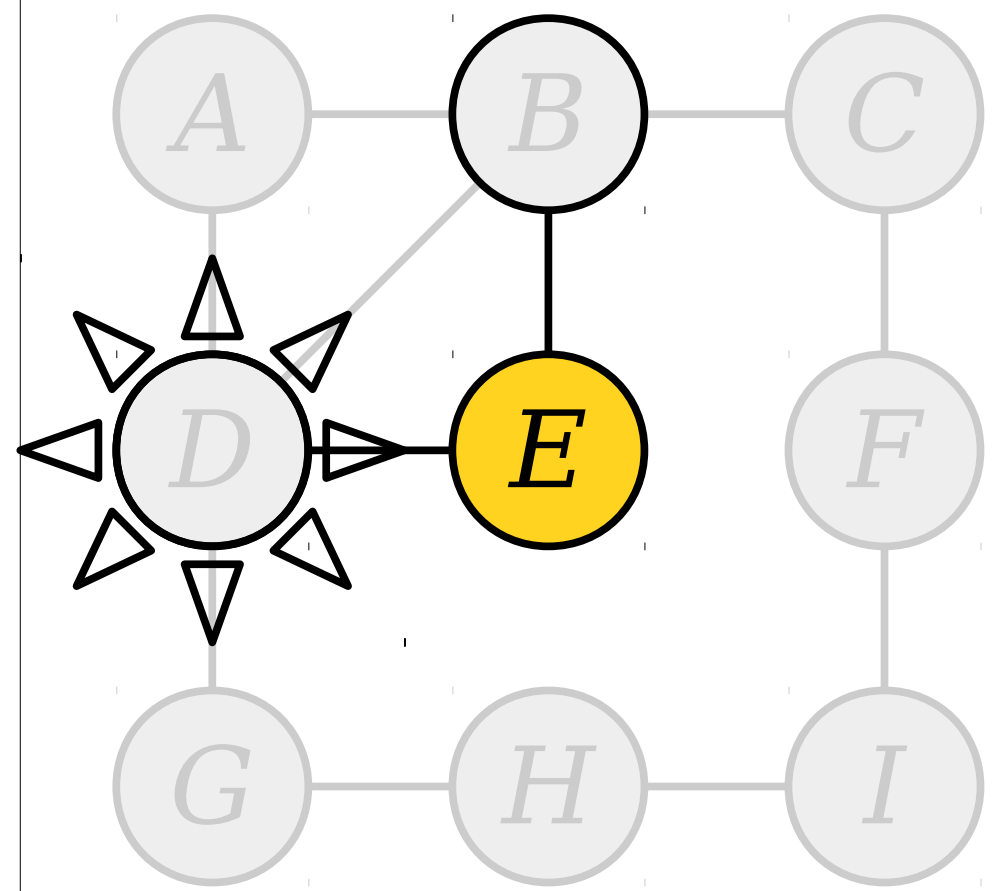


Queue: *B*

Load newly-discovered nodes into a queue.

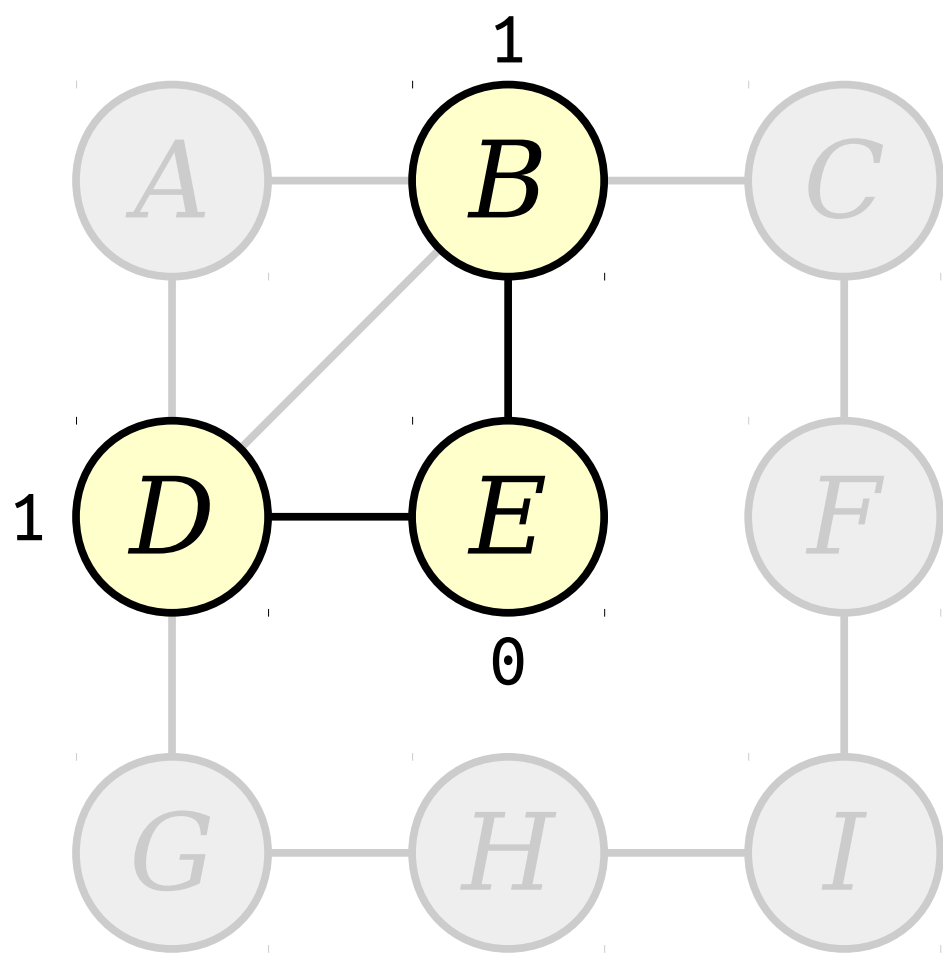


Visit nodes in ascending order of distance from the start node *E*.

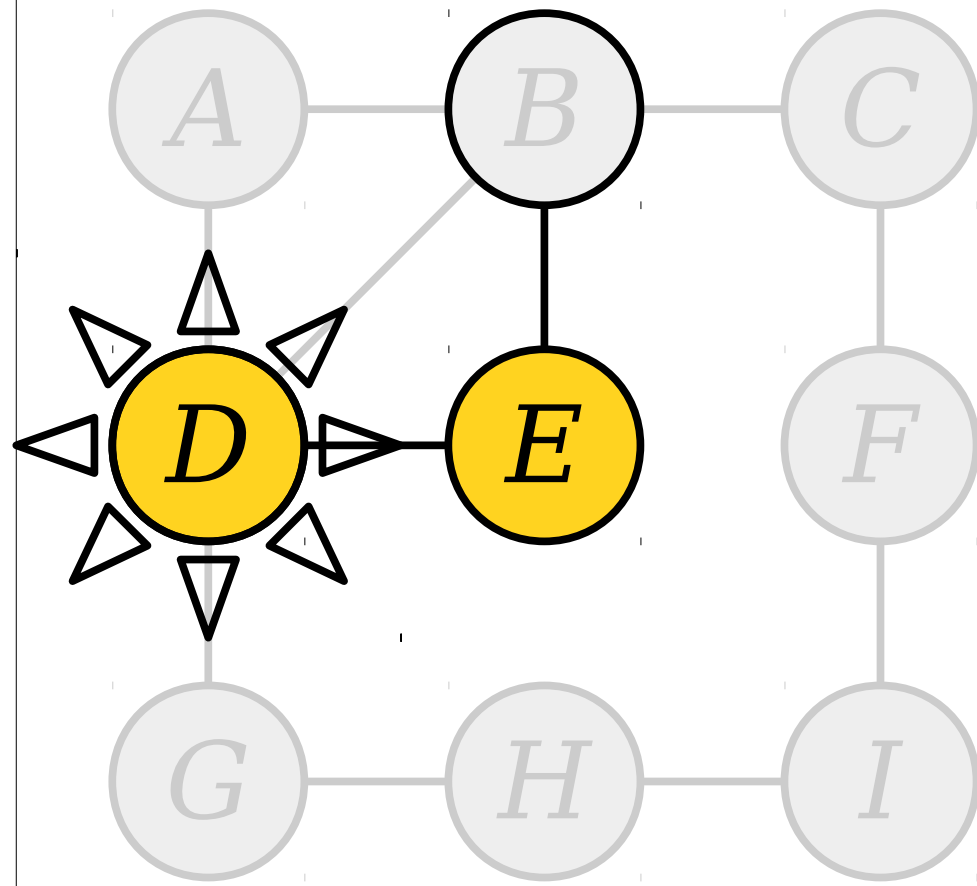


Queue: **B**

Load newly-discovered nodes into a queue.



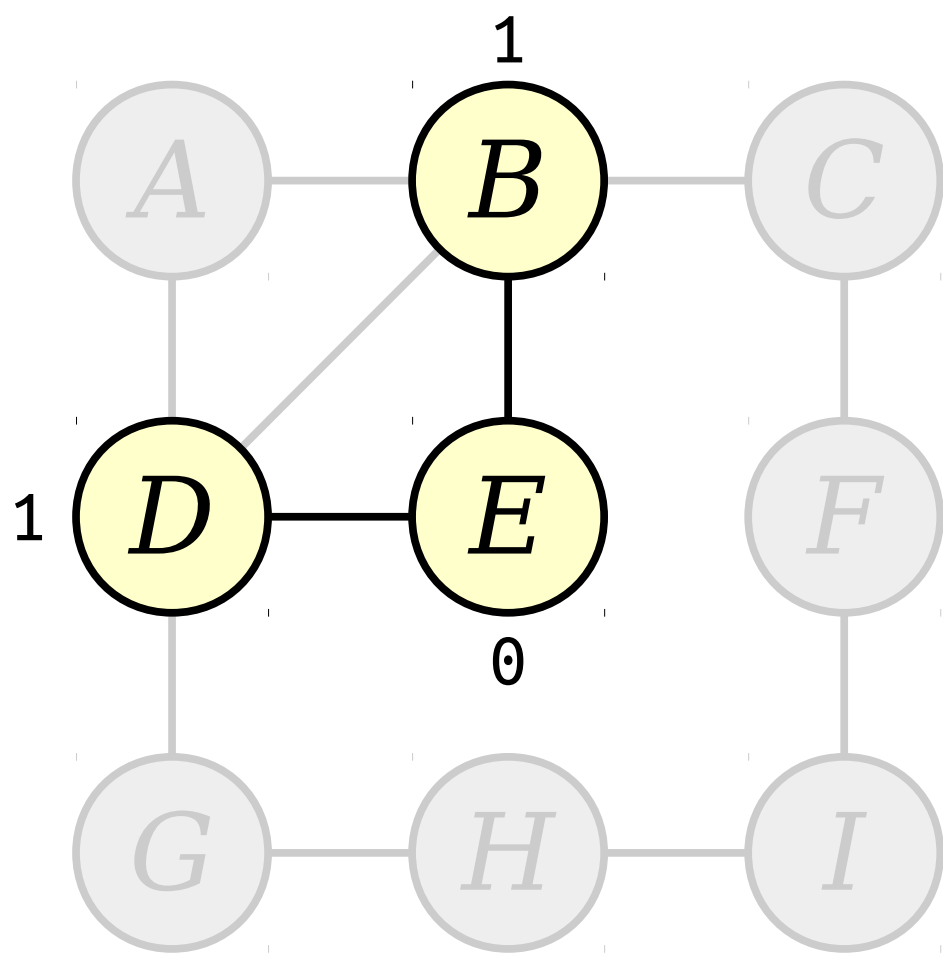
Visit nodes in ascending order of distance from the start node *E*.



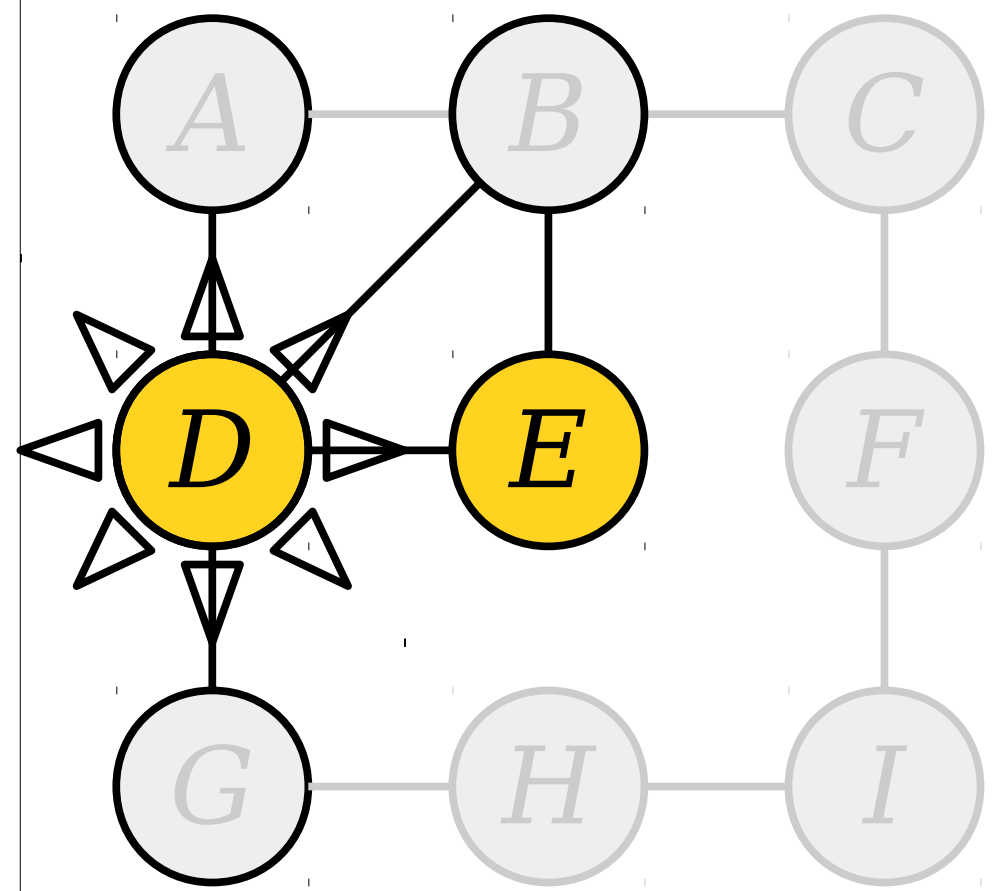
Queue: *B*

Load newly-discovered nodes into a queue.



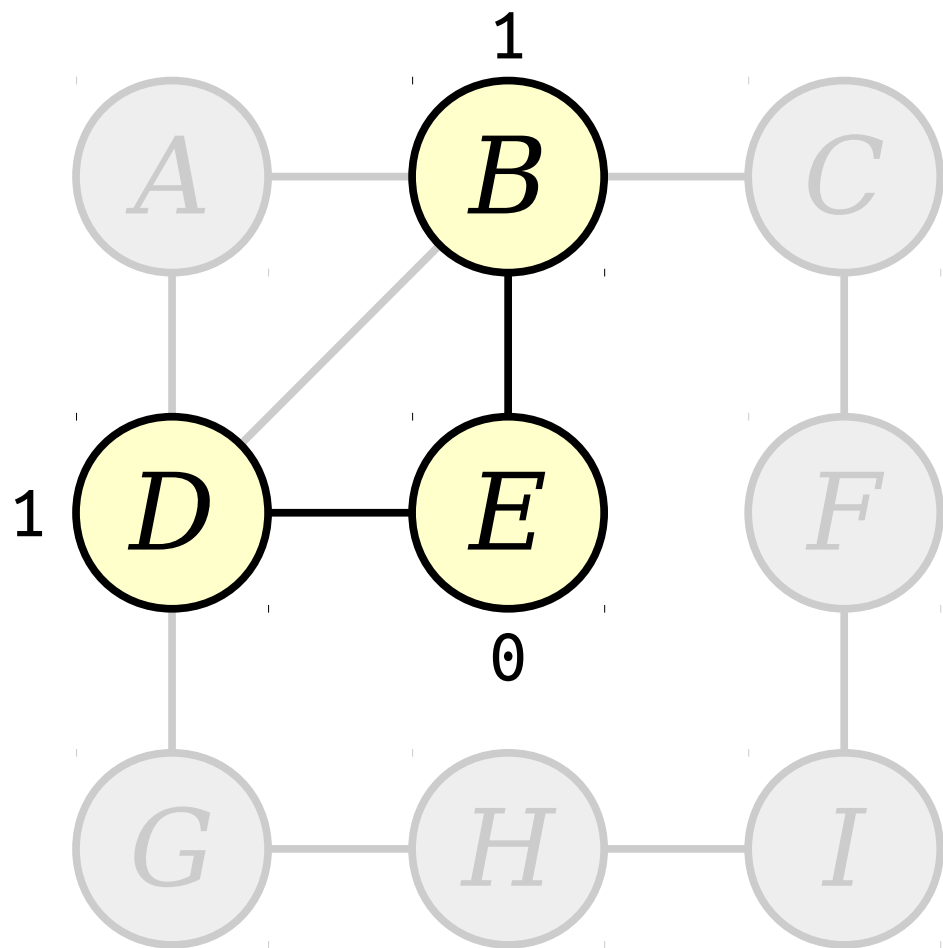


Visit nodes in ascending order of distance from the start node *E*.

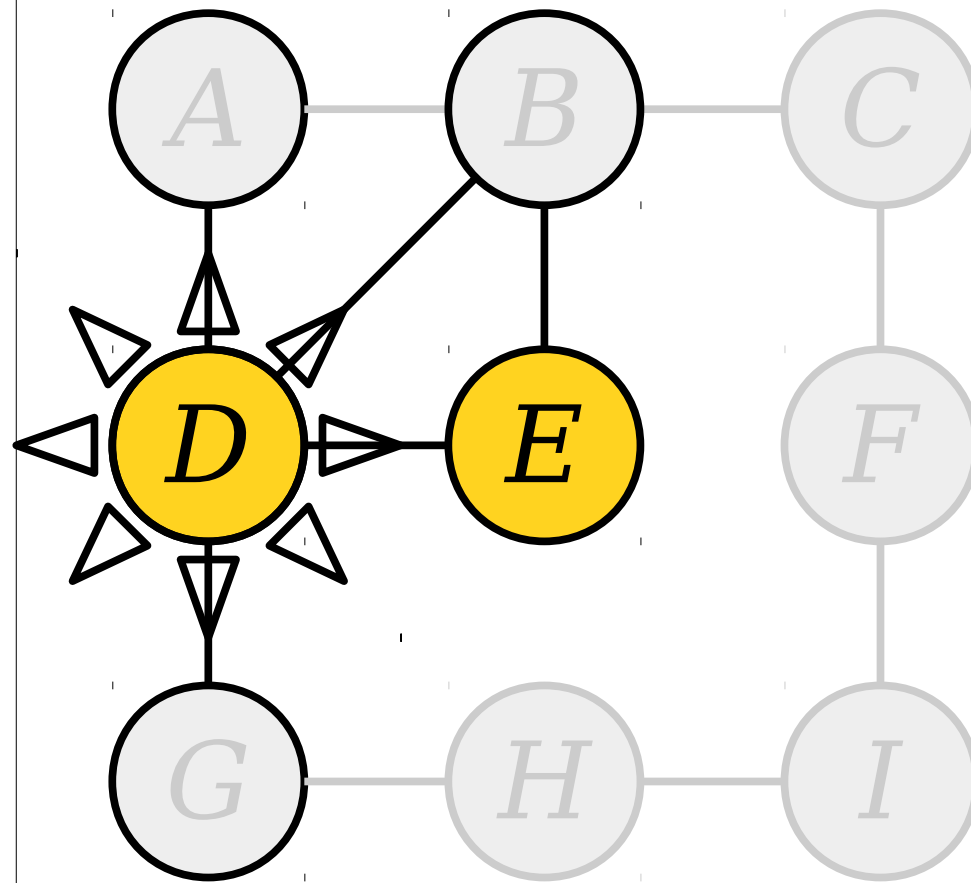


Queue: *B*

Load newly-discovered nodes into a queue.

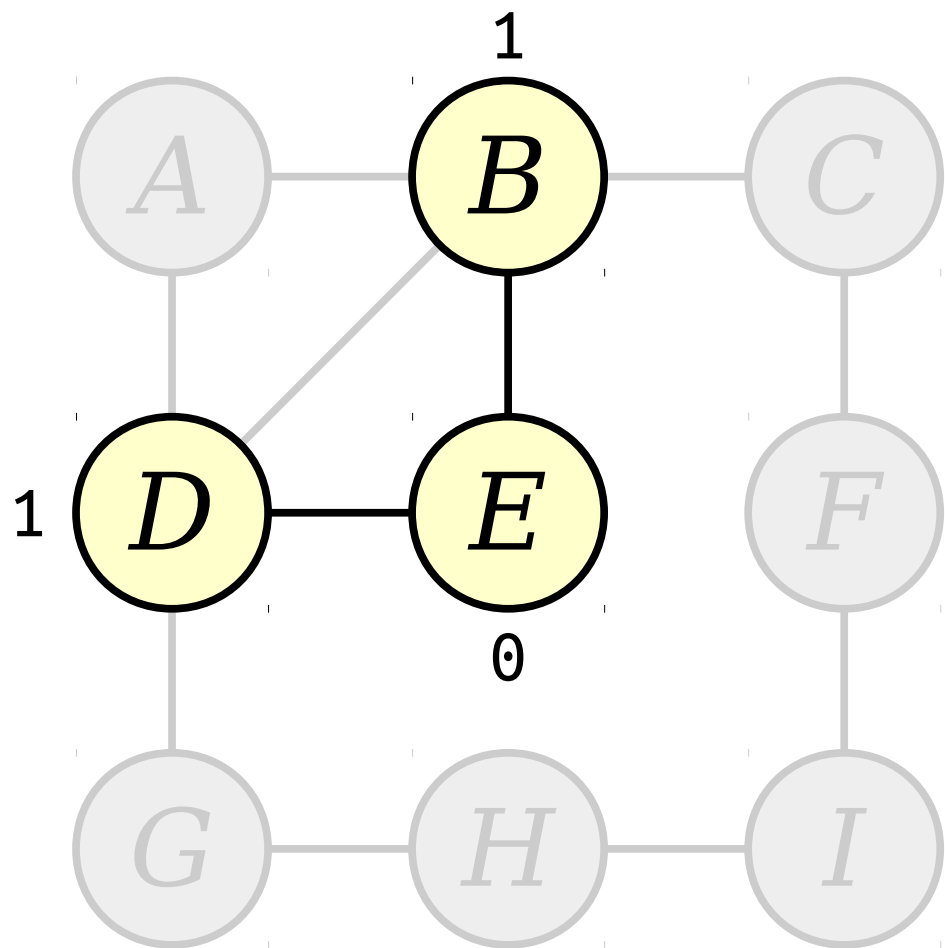


Visit nodes in ascending order of distance from the start node *E*.

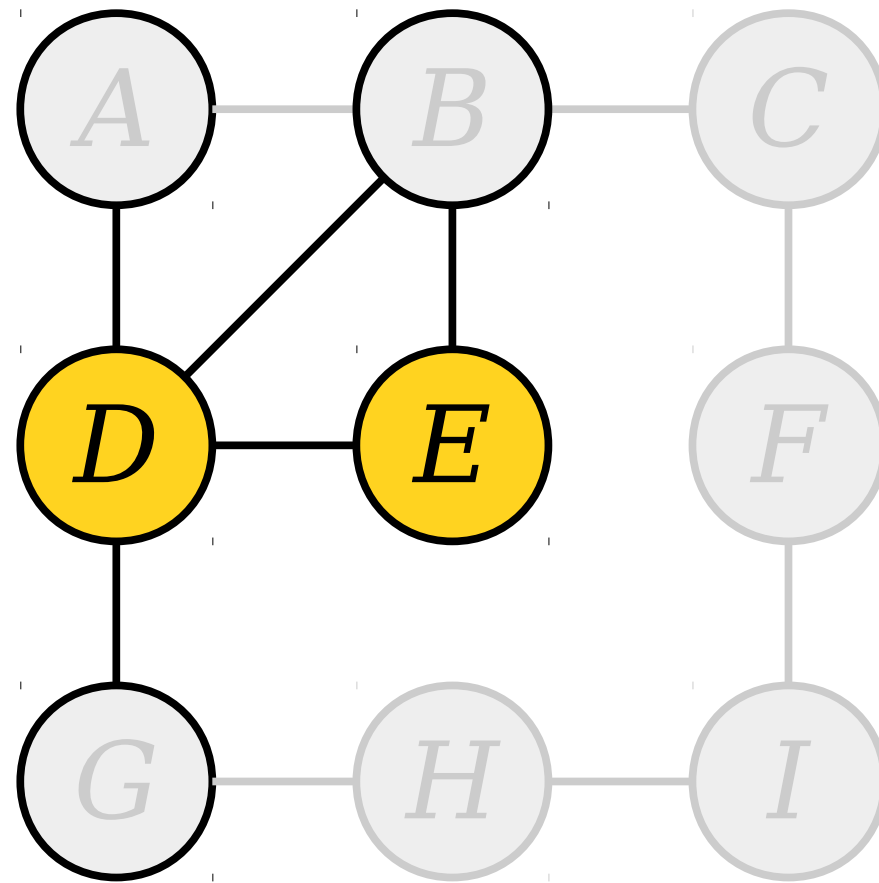


Queue: *B* *A* *G*

Load newly-discovered nodes into a queue.

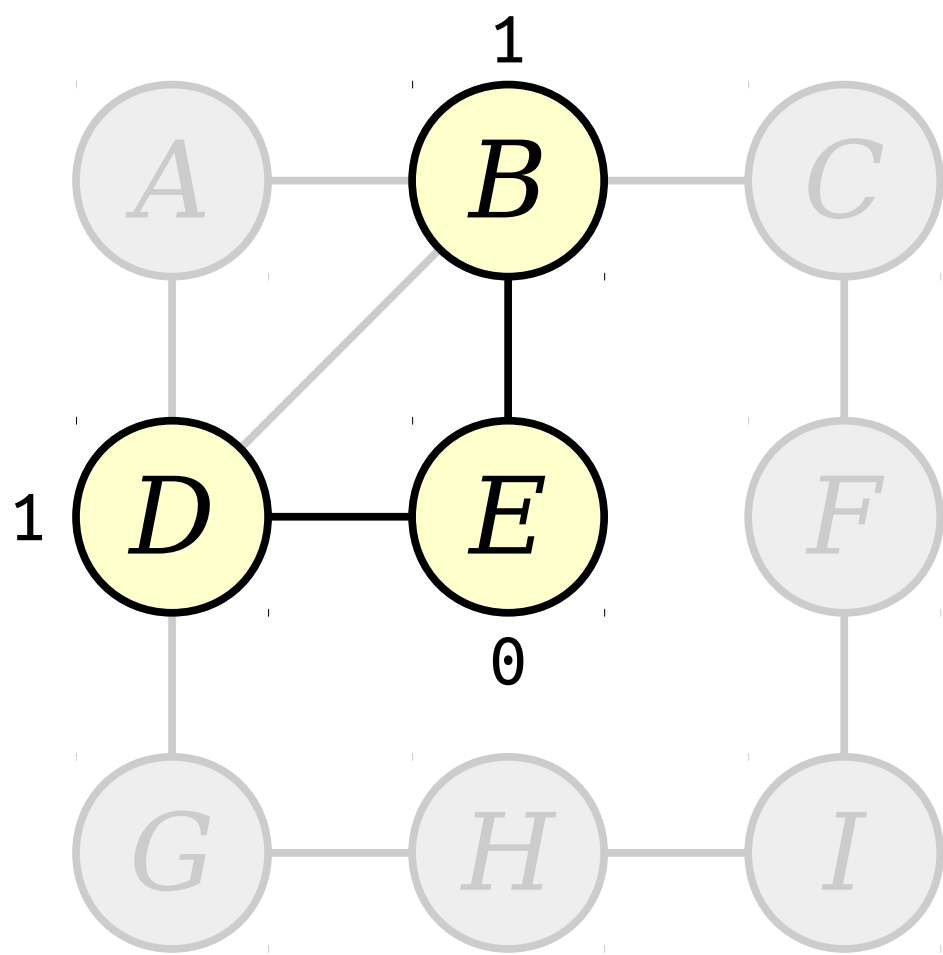


Visit nodes in ascending order of distance from the start node *E*.

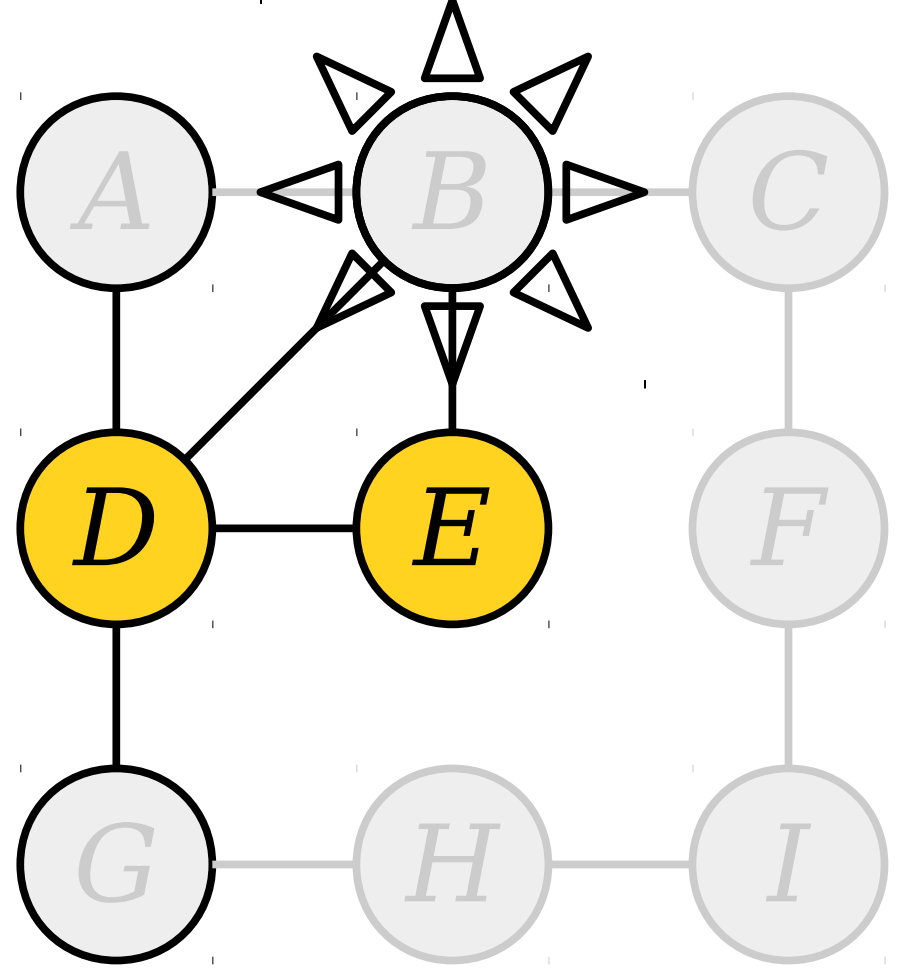


Queue: *B* *A* *G*

Load newly-discovered nodes into a queue.

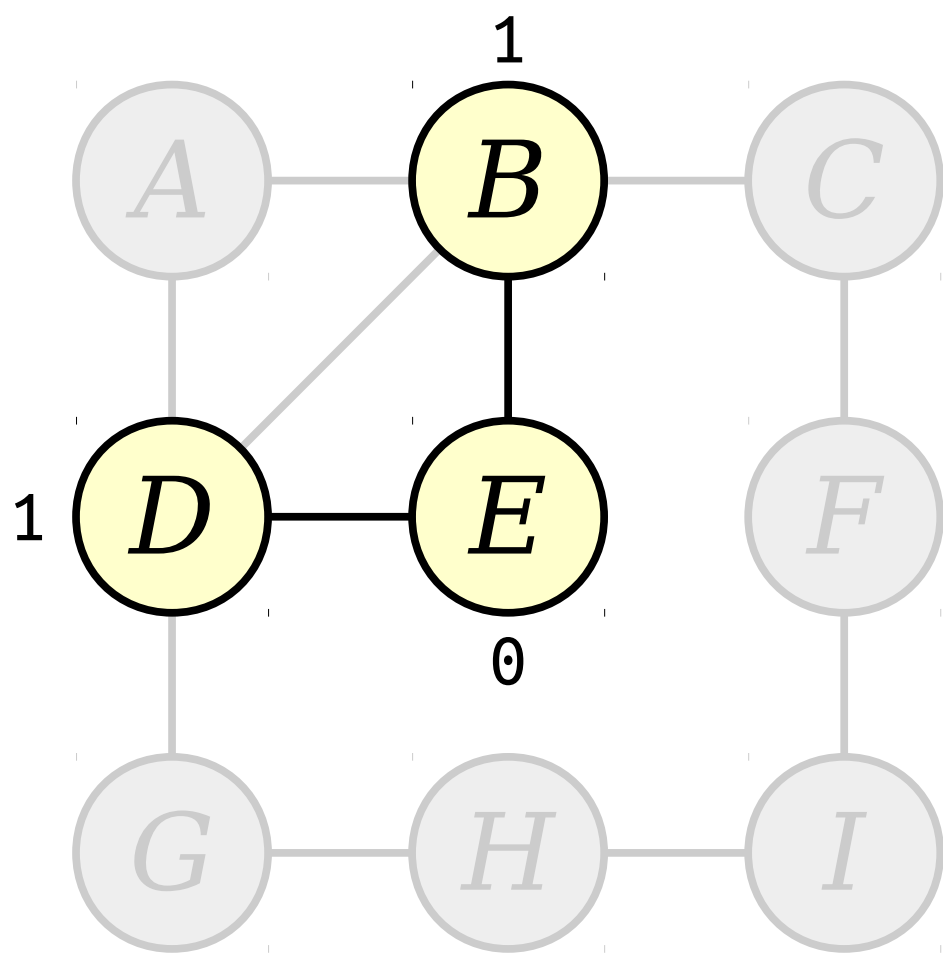


Visit nodes in ascending order of distance from the start node *E*.

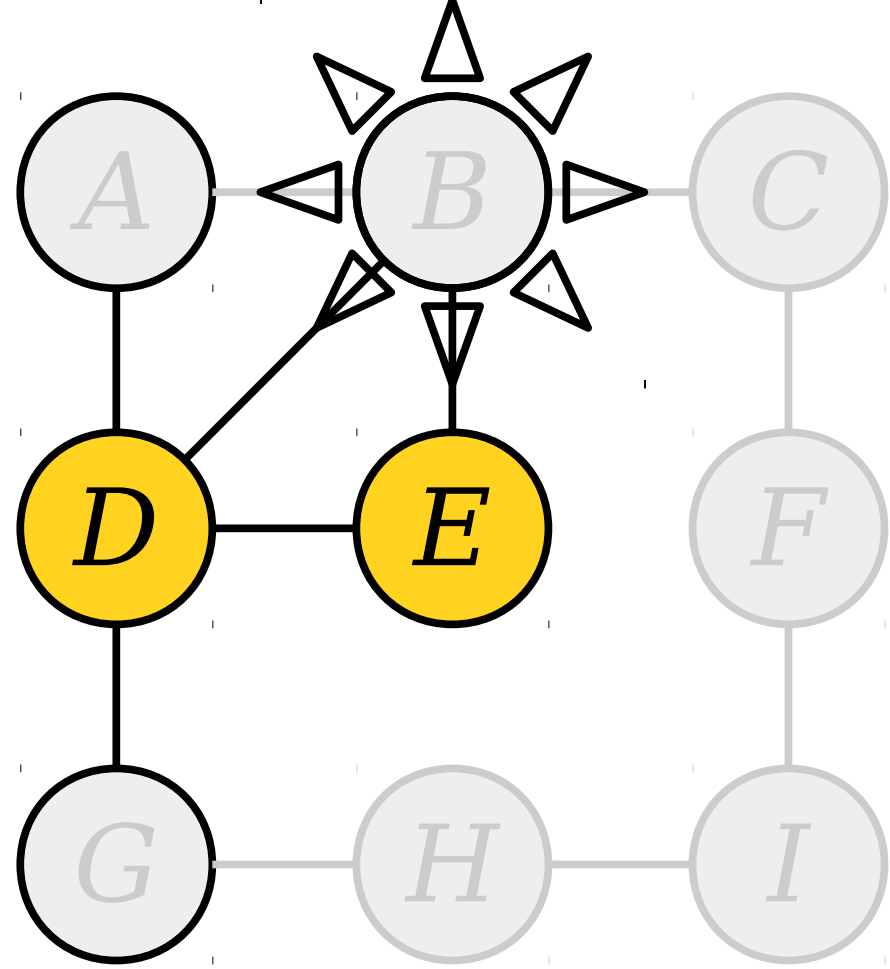


Queue: *B* *A* *G*

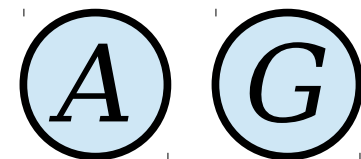
Load newly-discovered nodes into a queue.



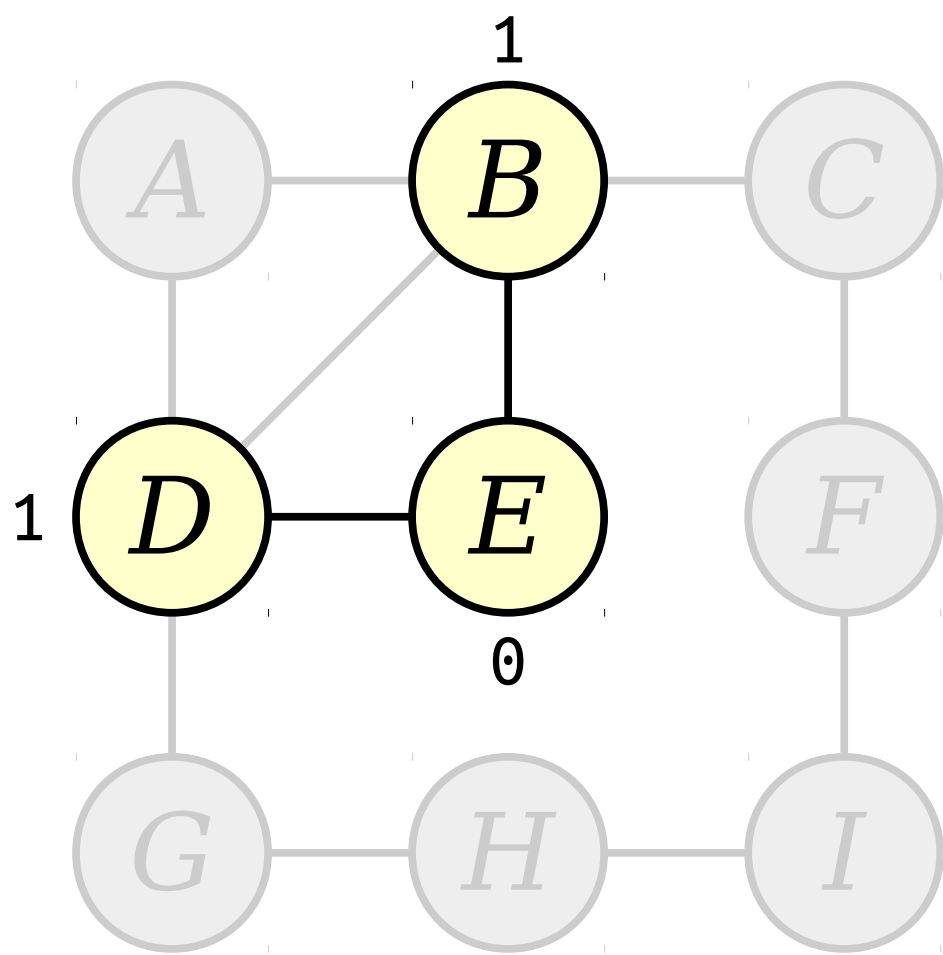
Visit nodes in ascending order of distance from the start node *E*.



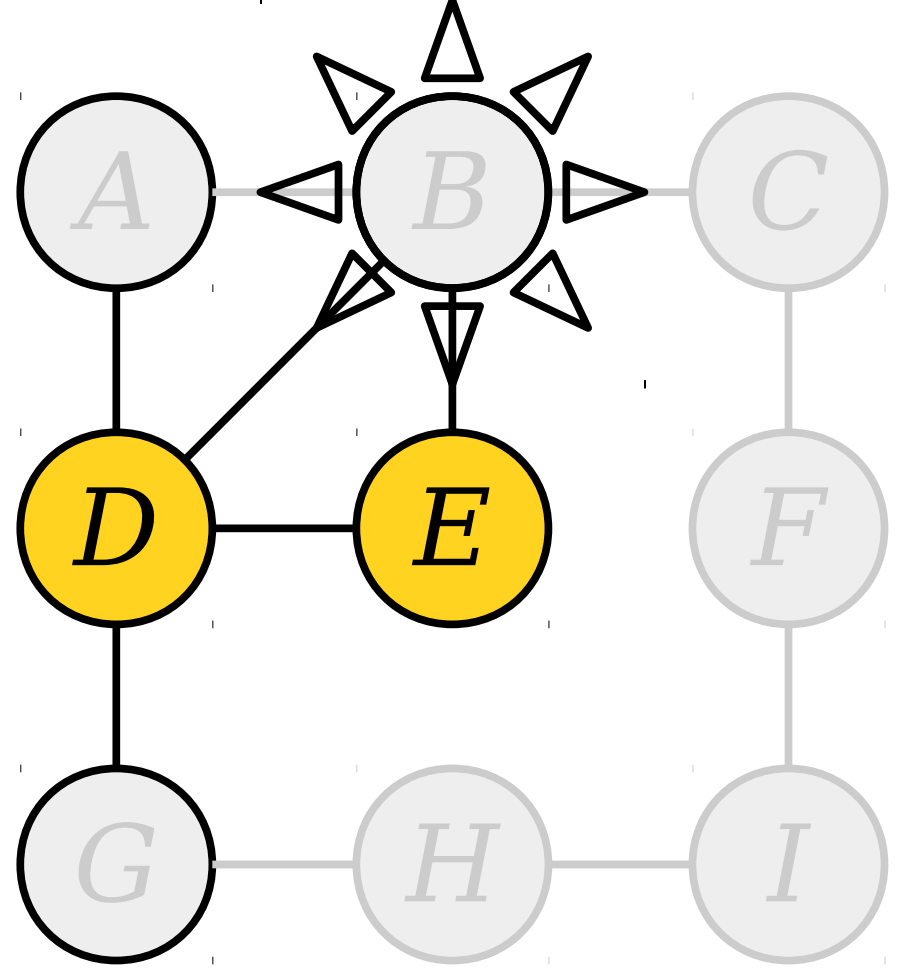
Queue:



Load newly-discovered nodes into a queue.

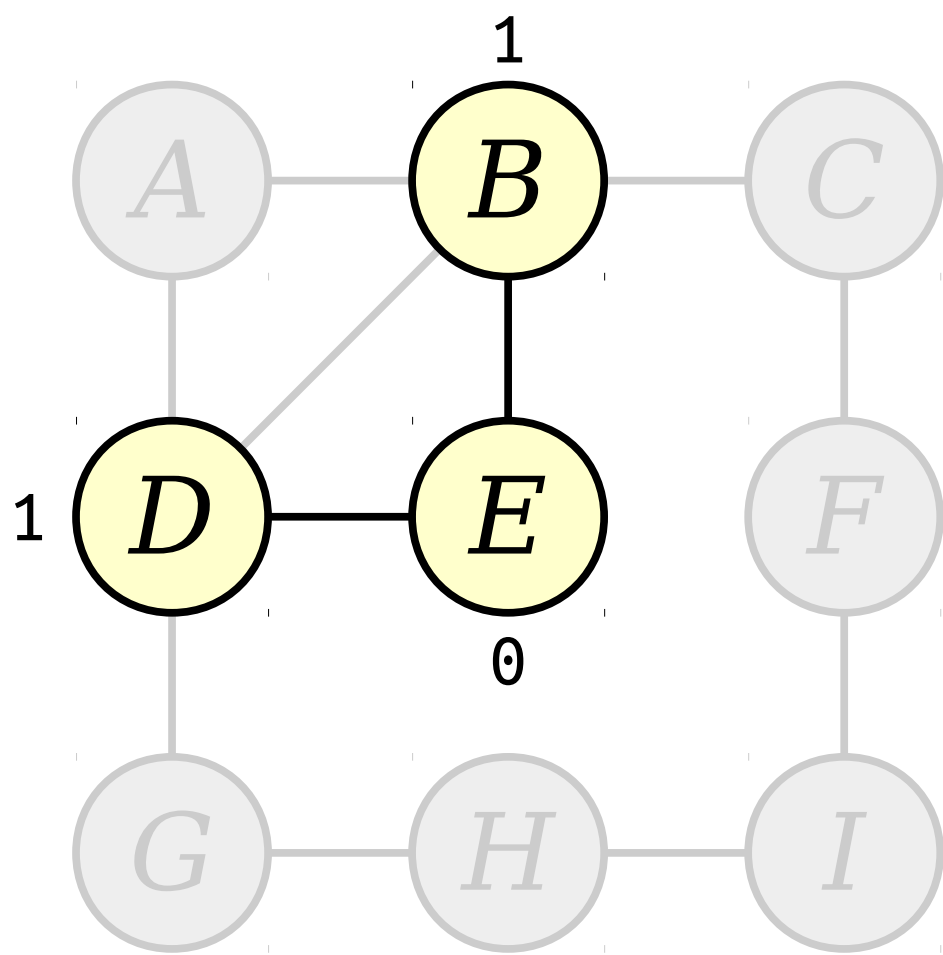


Visit nodes in ascending order of distance from the start node *E*.

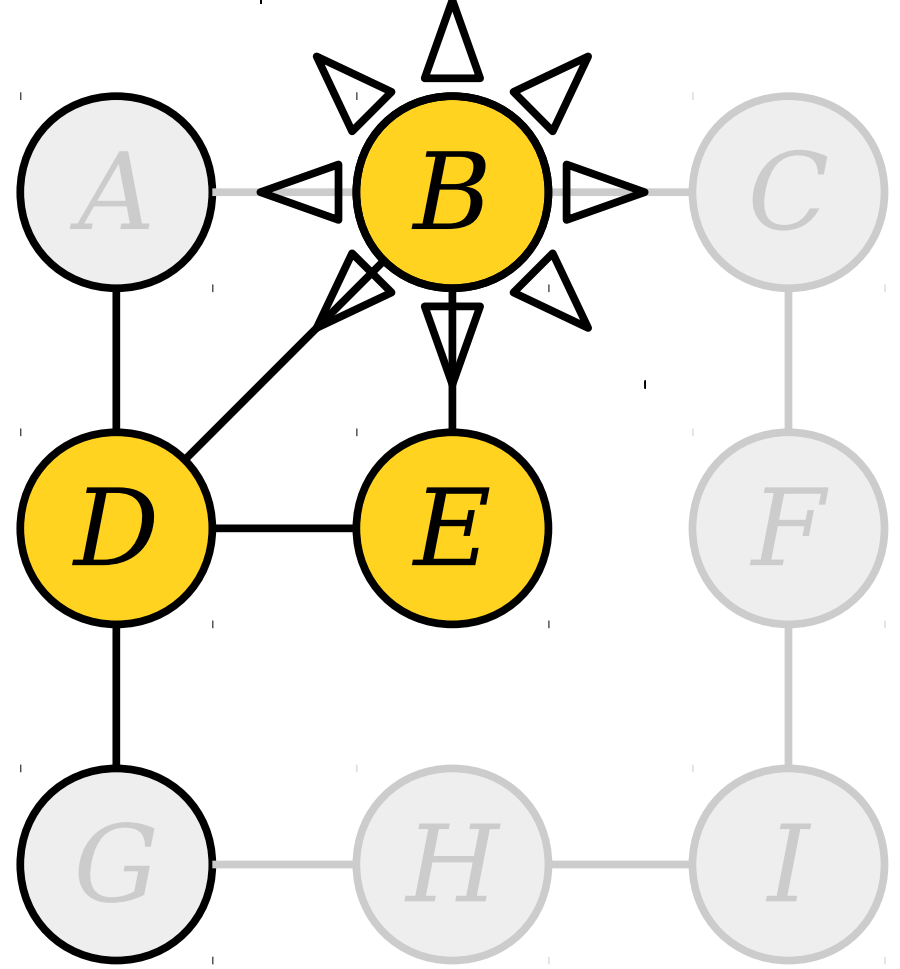


Queue: *A* *G*

Load newly-discovered nodes into a queue.

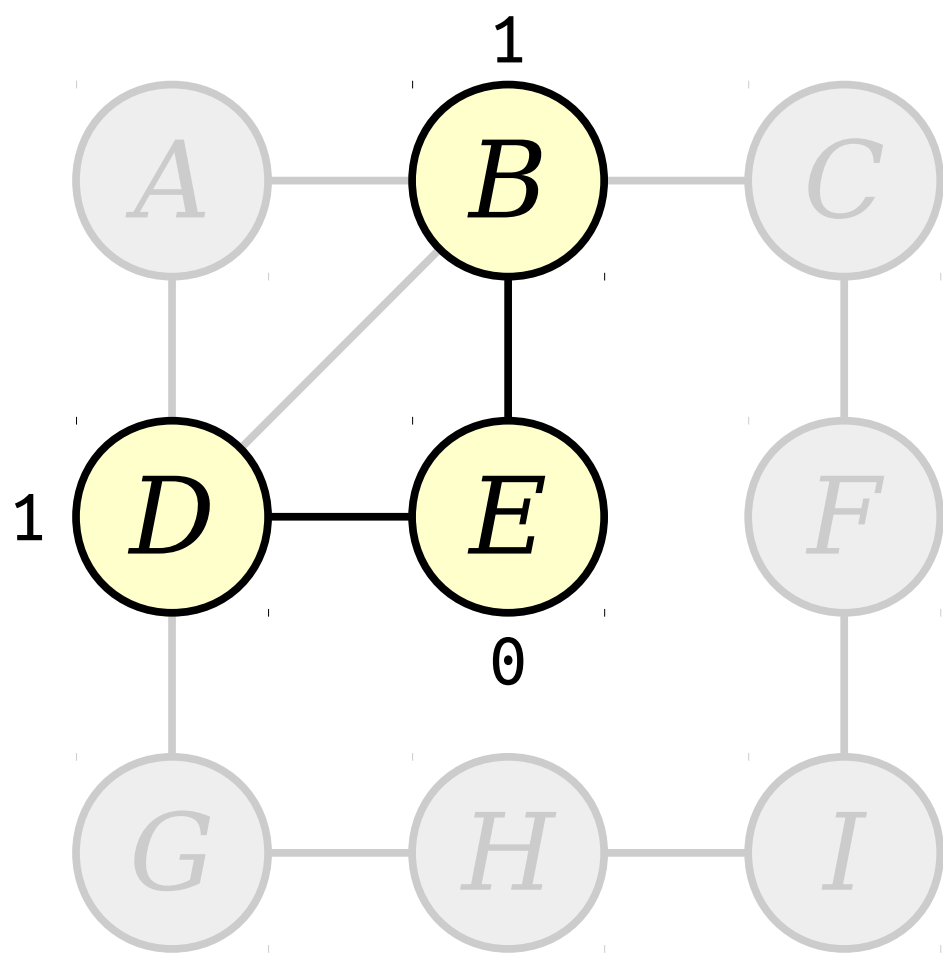


Visit nodes in ascending order of distance from the start node *E*.

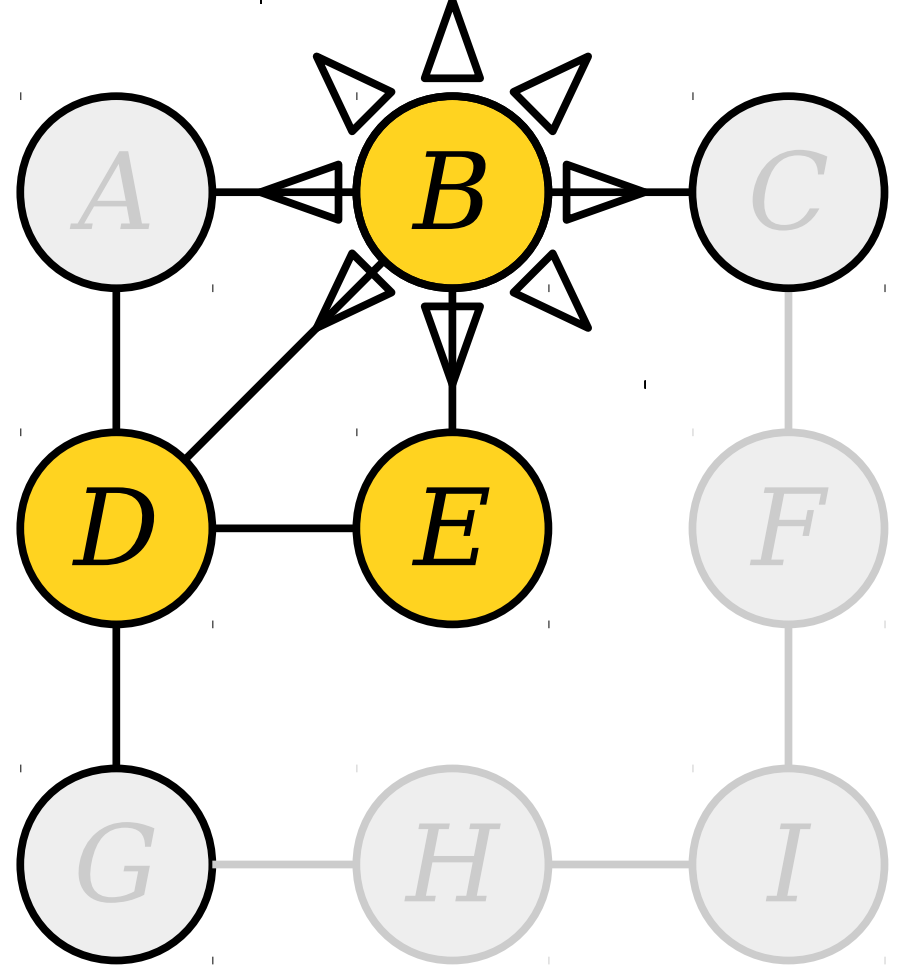


Queue: *A* *G*

Load newly-discovered nodes into a queue.



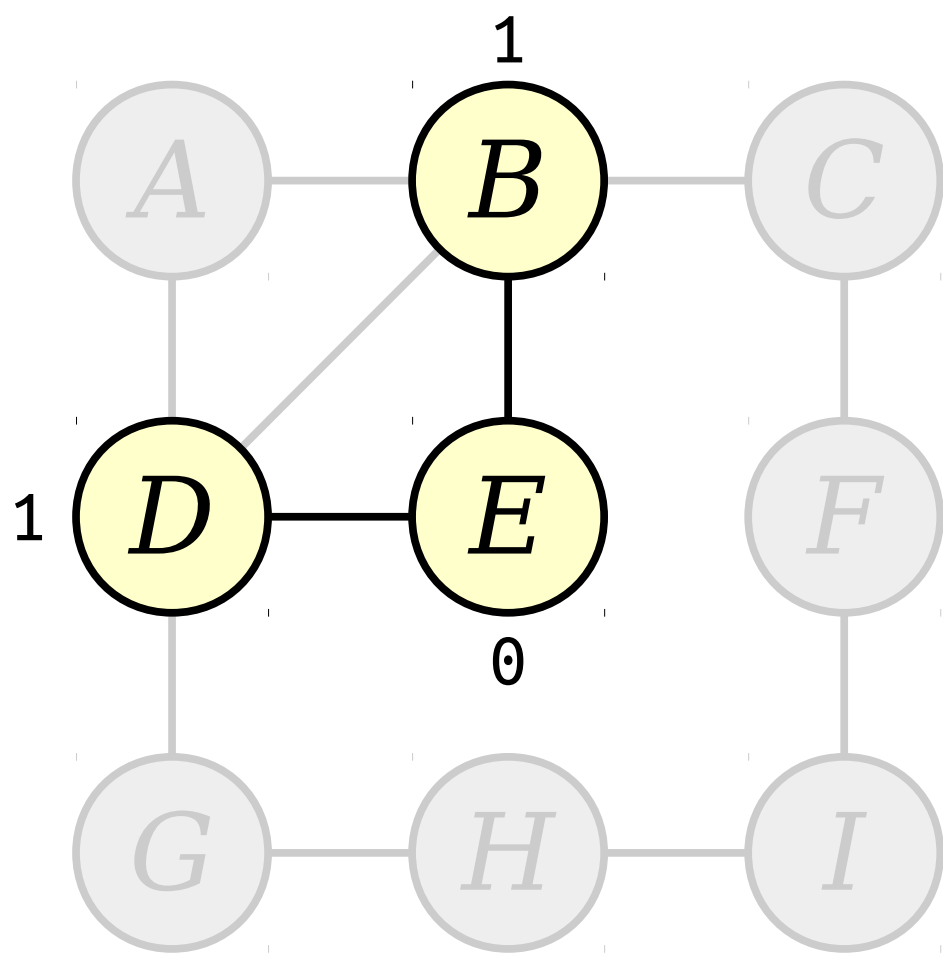
Visit nodes in ascending order of distance from the start node *E*.



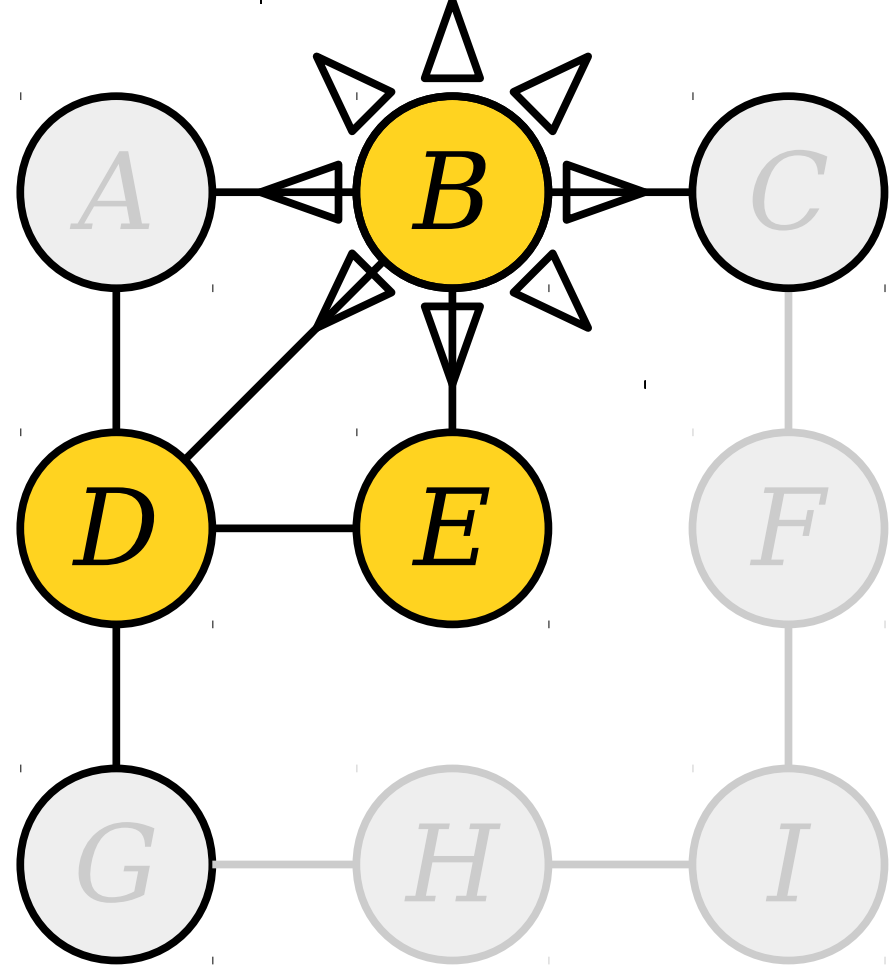
Queue: **A** **G**

Load newly-discovered nodes into a queue.



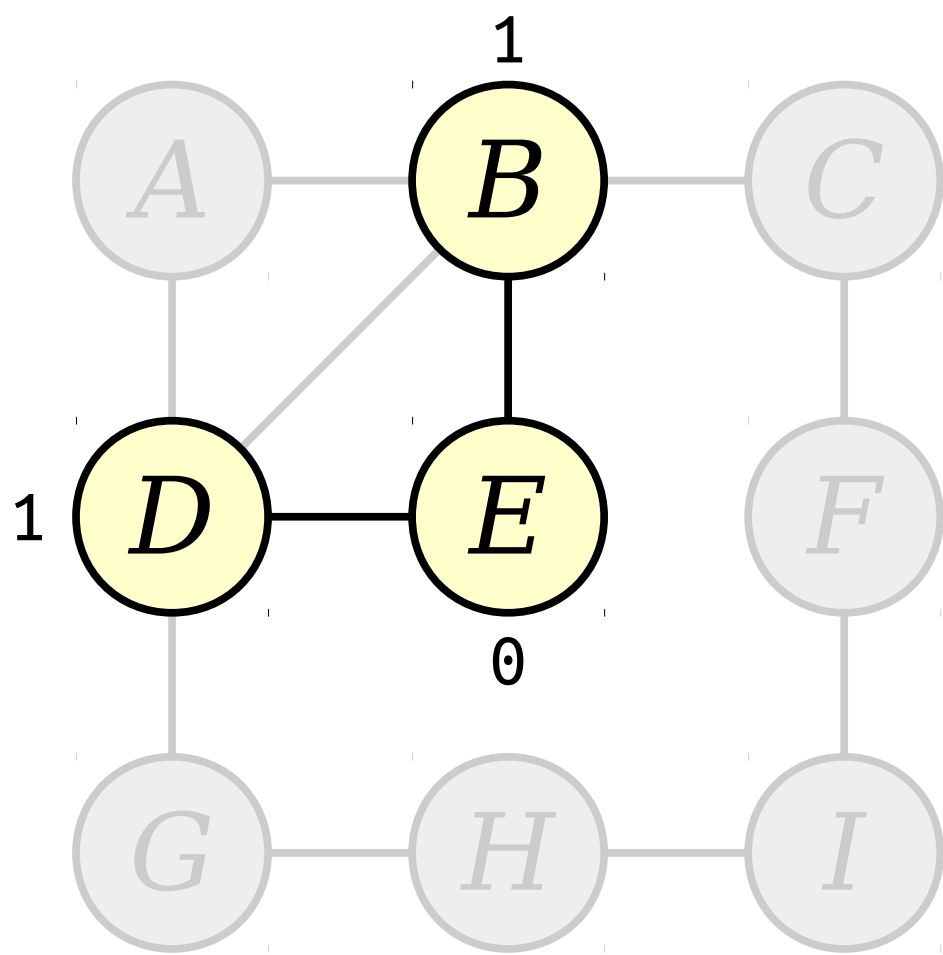


Visit nodes in ascending order of distance from the start node *E*.

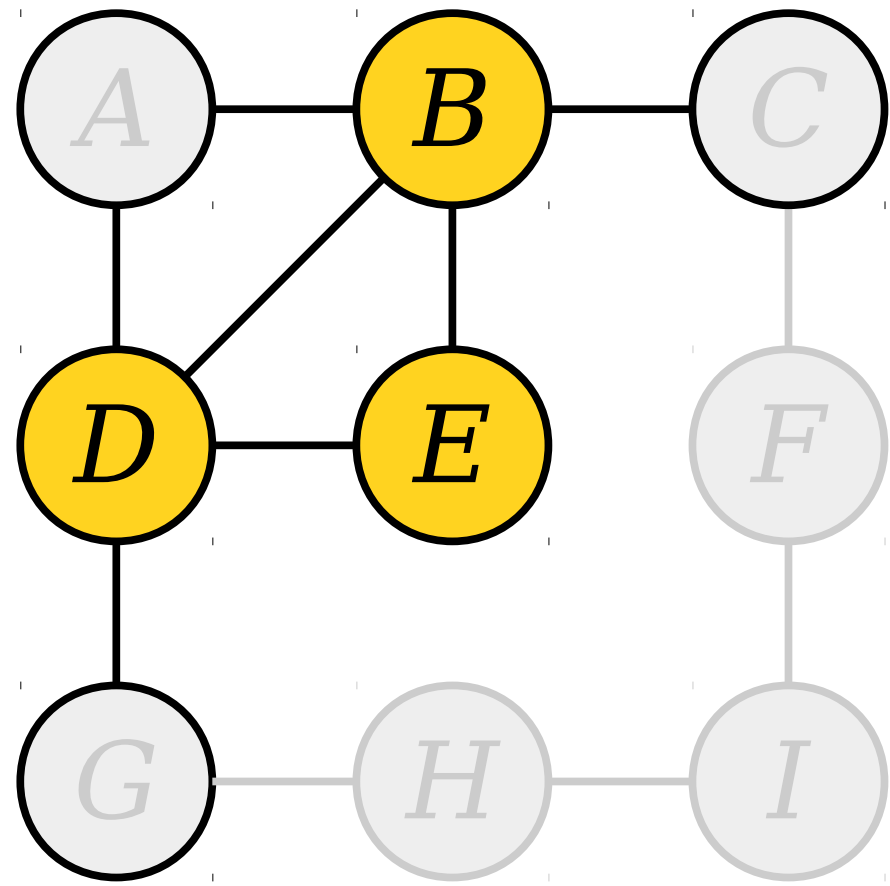


Queue: *A* *G* *C*

Load newly-discovered nodes into a queue.

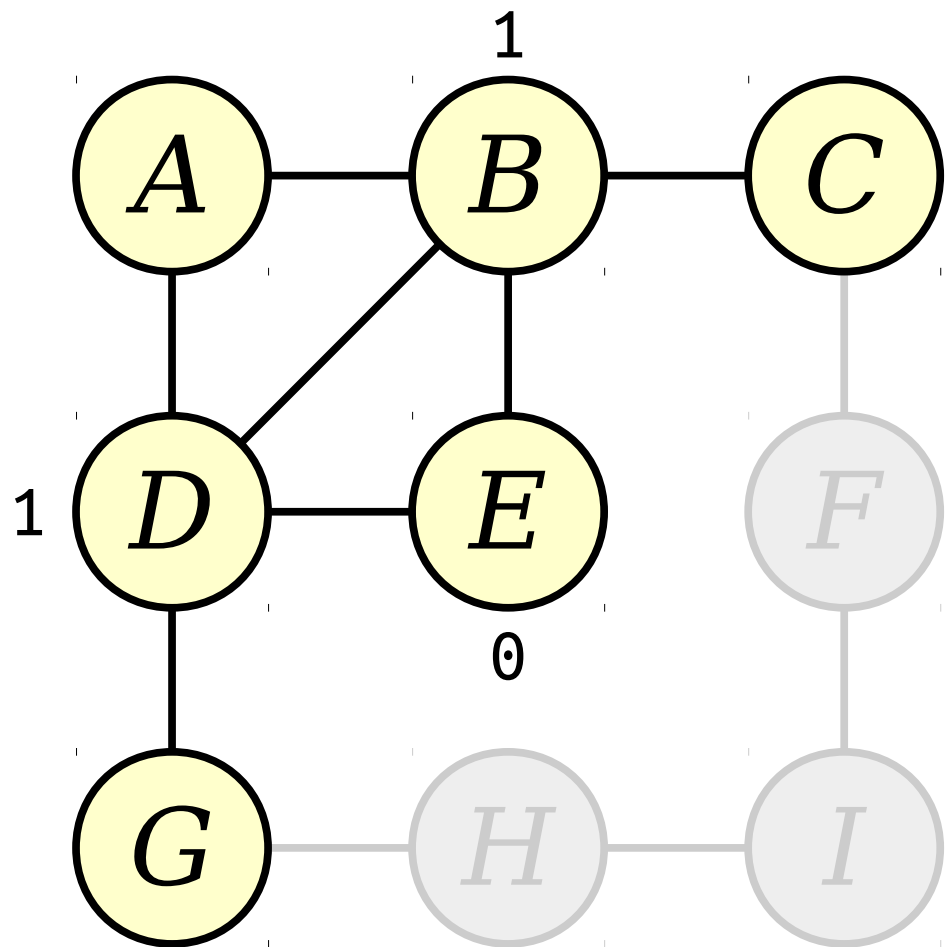


Visit nodes in ascending order of distance from the start node *E*.

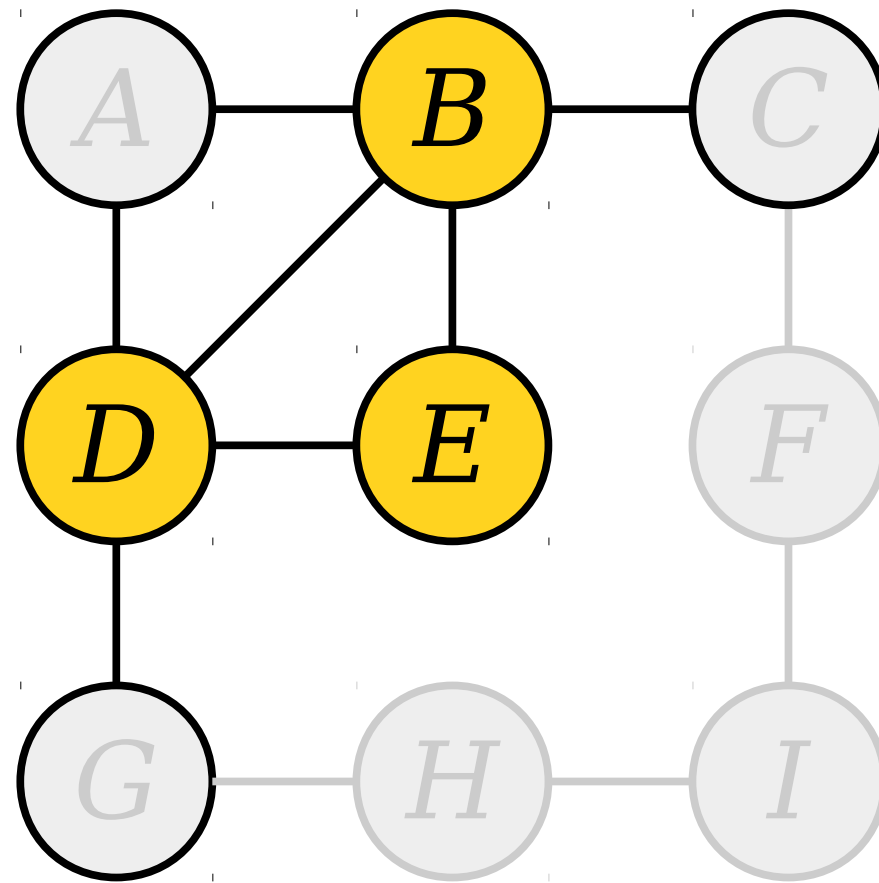


Queue: *A* *G* *C*

Load newly-discovered nodes into a queue.

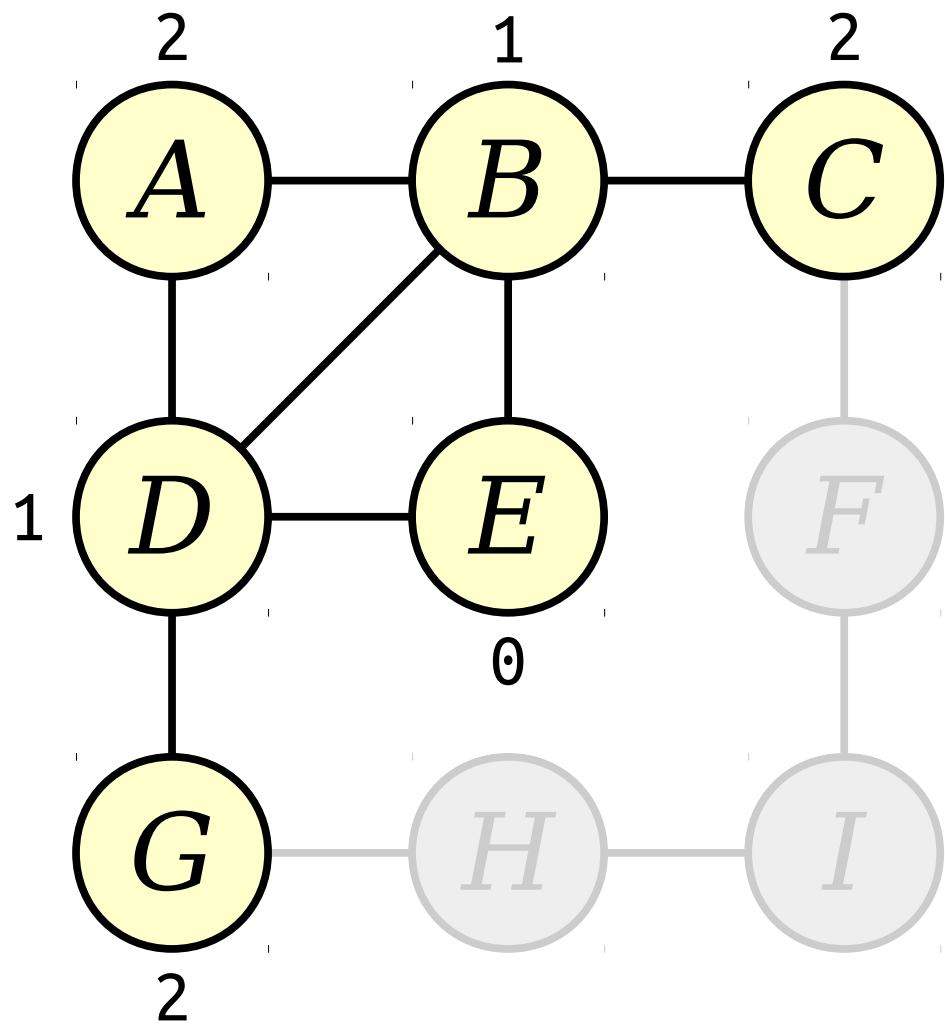


Visit nodes in ascending order of distance from the start node *E*.

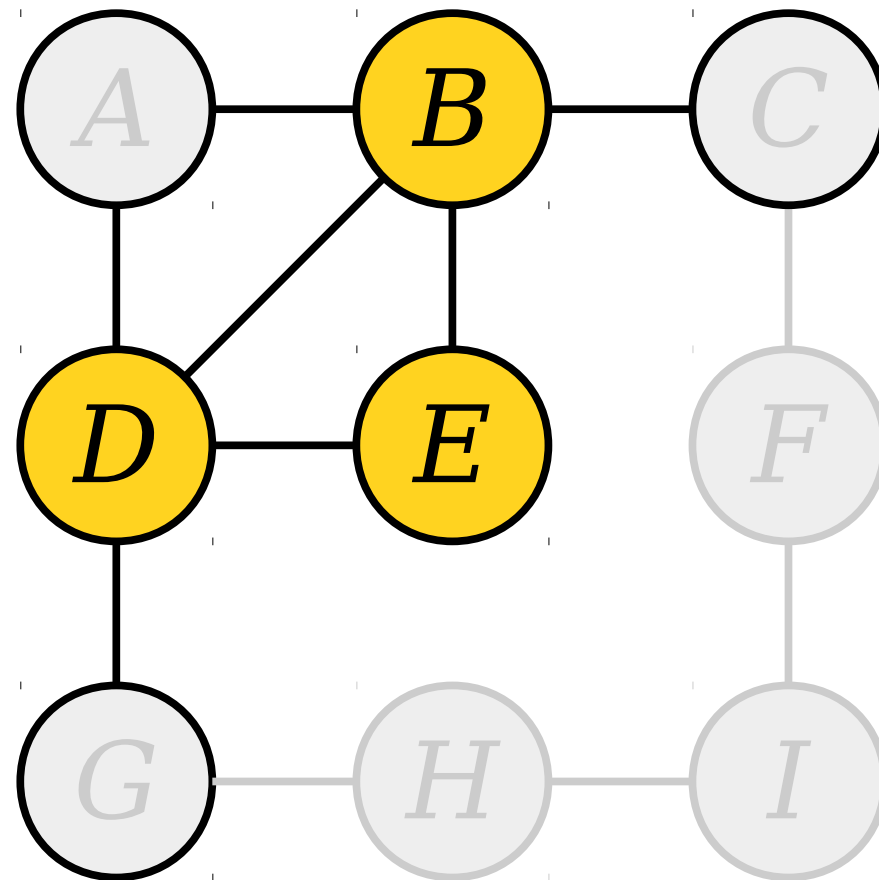


Queue: **A** **G** **C**

Load newly-discovered nodes into a queue.

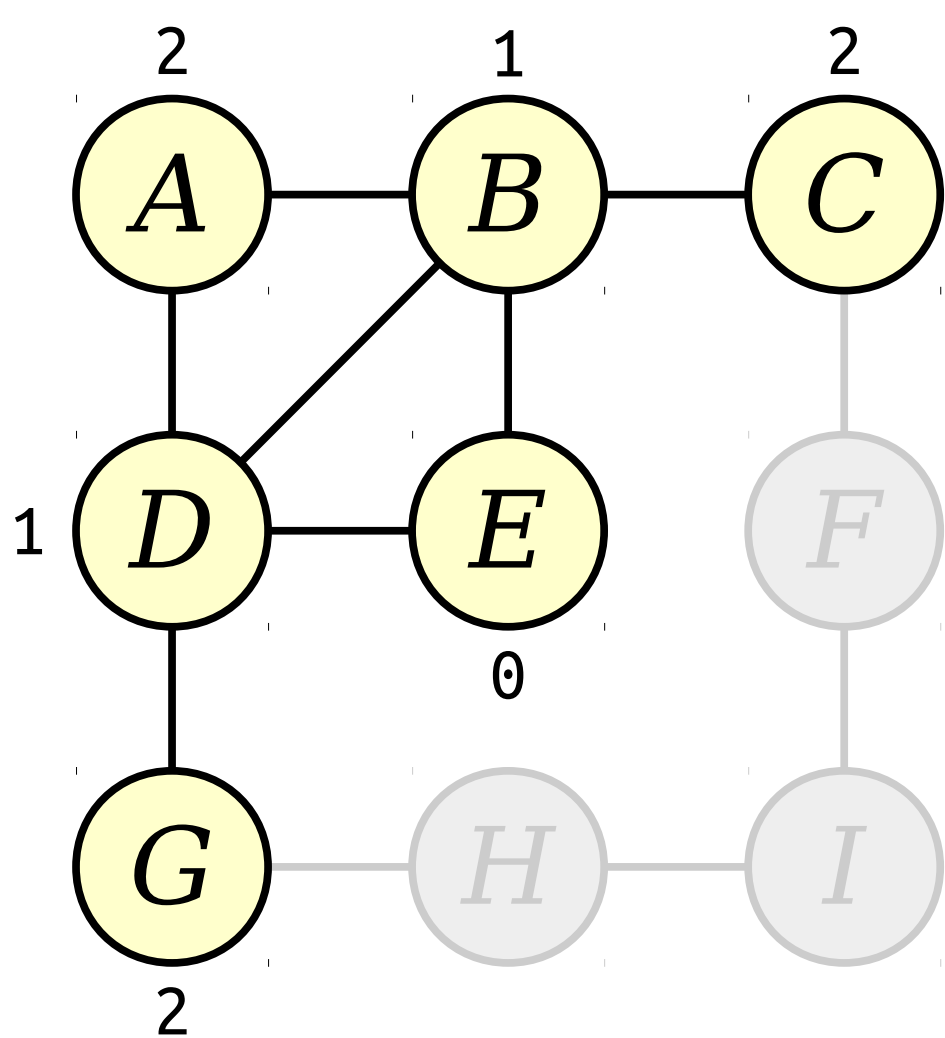


Visit nodes in ascending order of distance from the start node *E*.

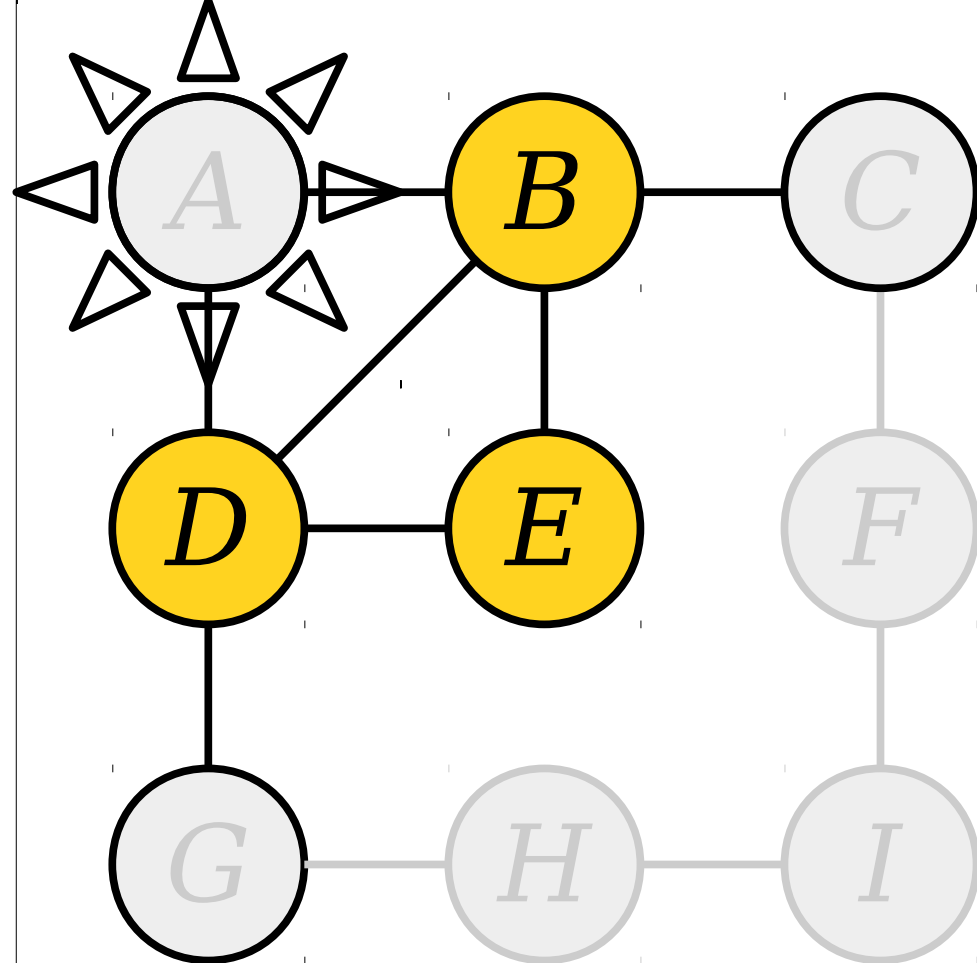


Queue: *A* *G* *C*

Load newly-discovered nodes into a queue.

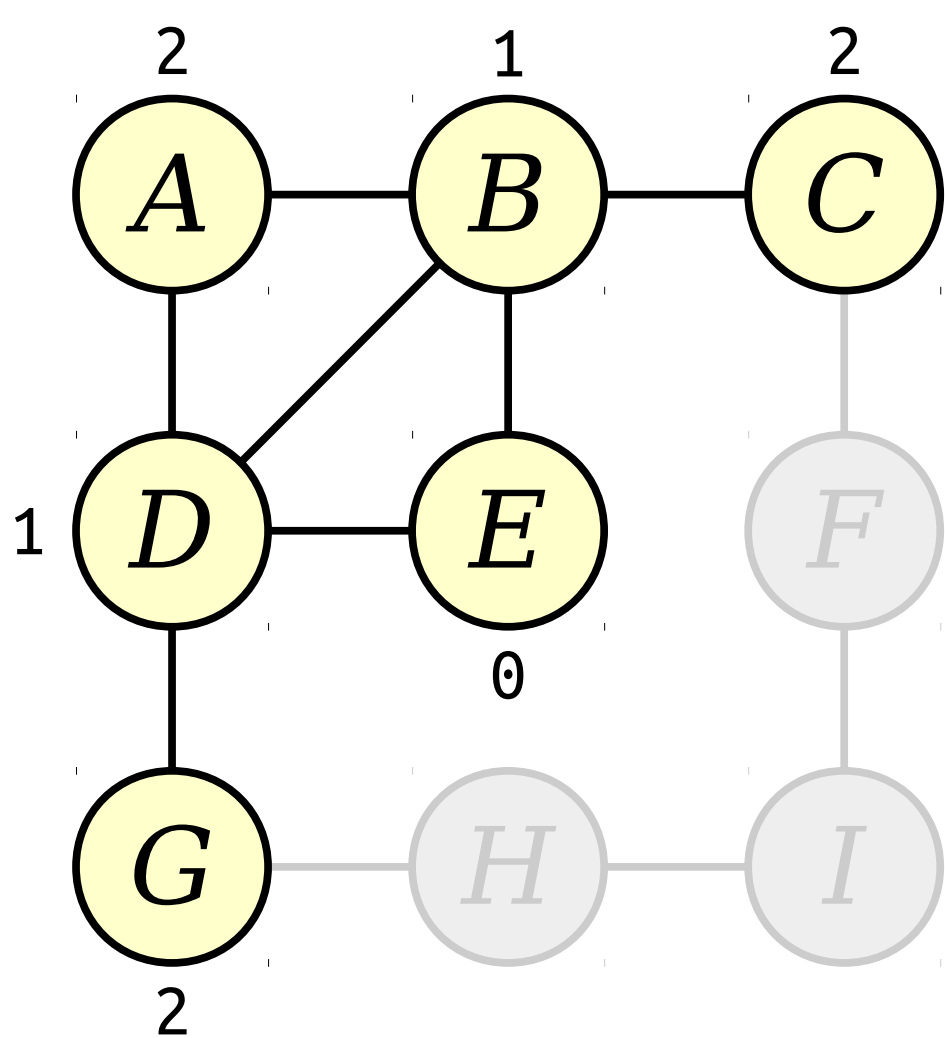


Visit nodes in ascending order of distance from the start node *E*.

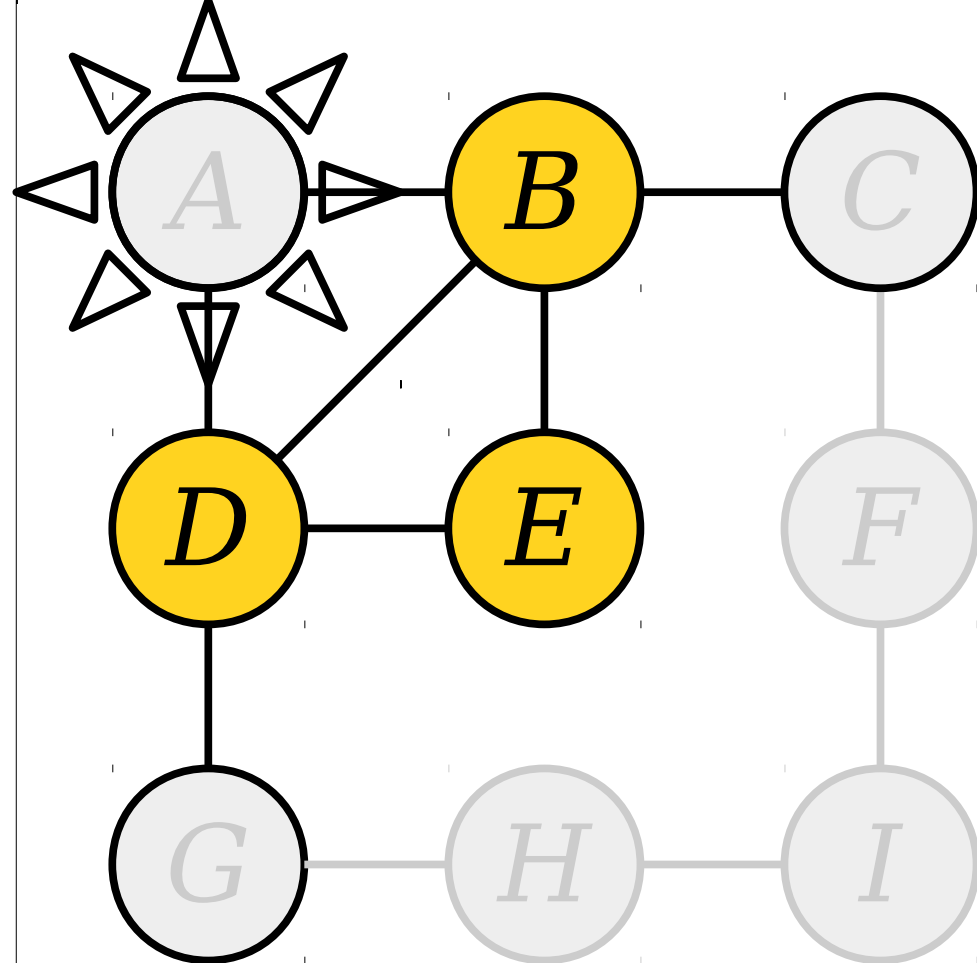


Queue: *A* *G* *C*

Load newly-discovered nodes into a queue.

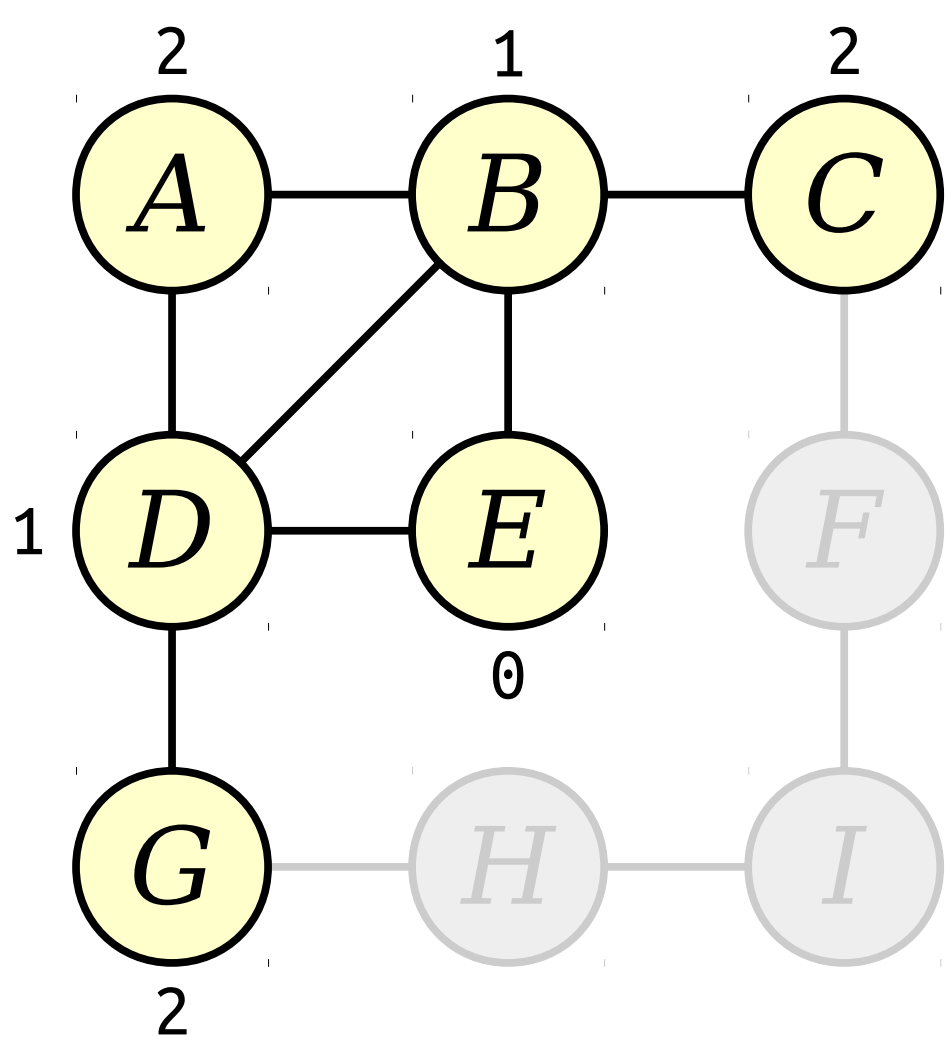


Visit nodes in ascending order of distance from the start node *E*.

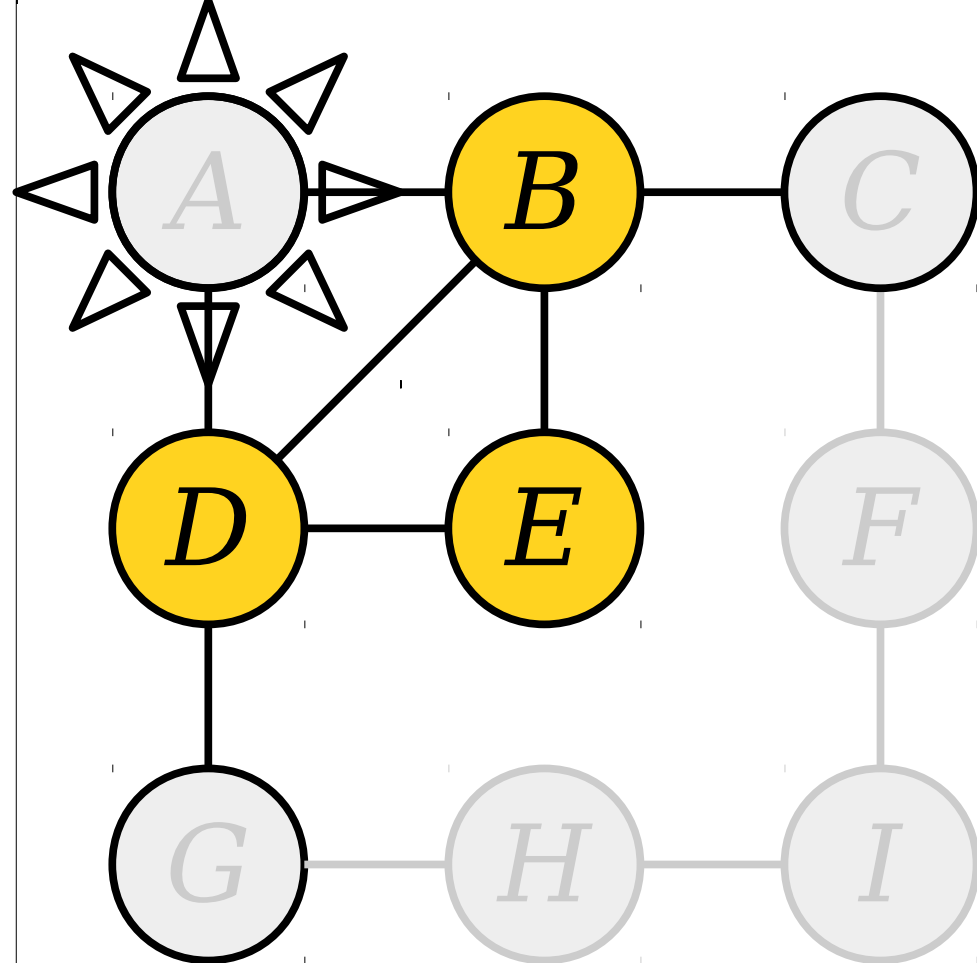


Queue: **G** **C**

Load newly-discovered nodes into a queue.

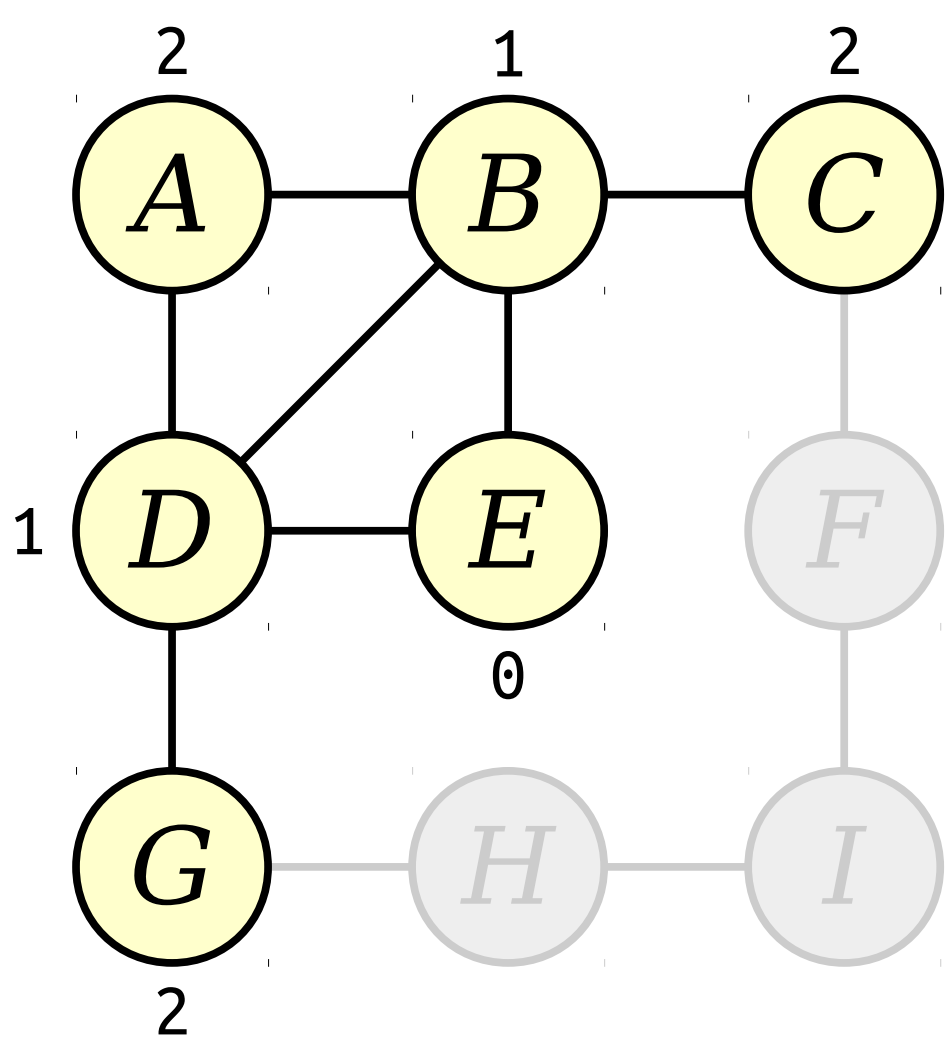


Visit nodes in ascending order of distance from the start node *E*.

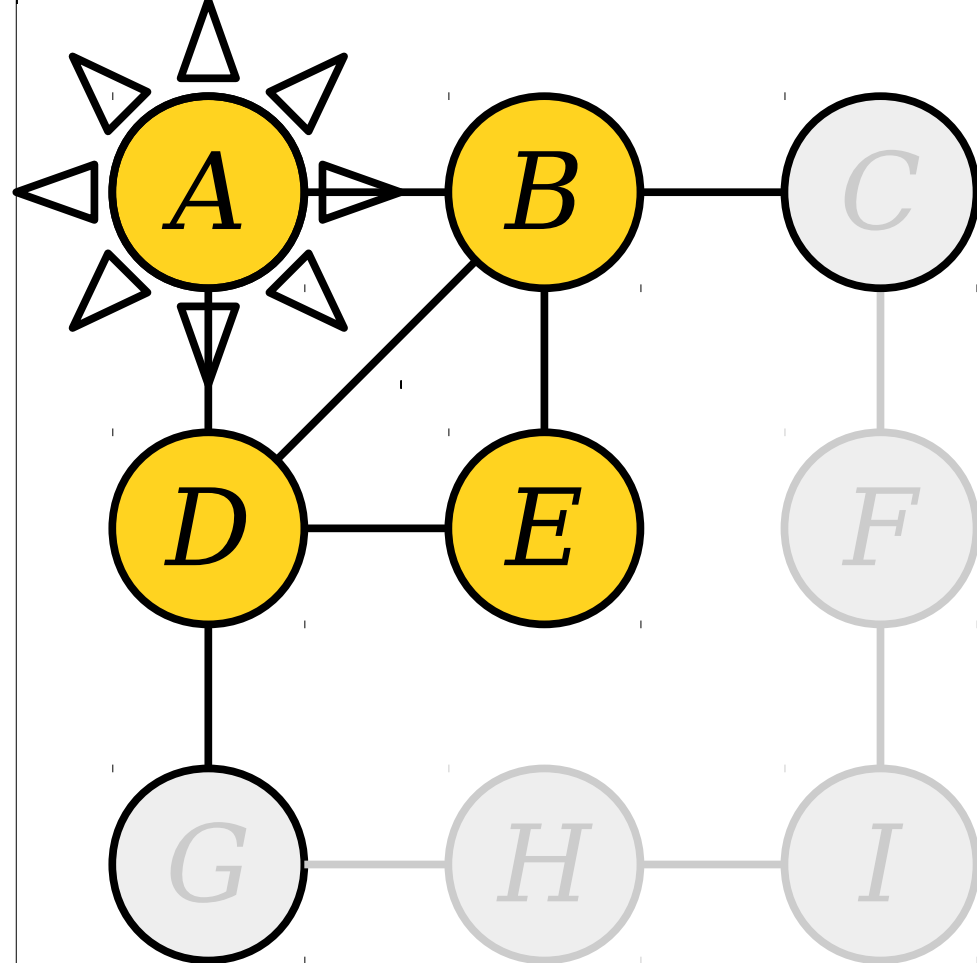


Queue: **G** **C**

Load newly-discovered nodes into a queue.



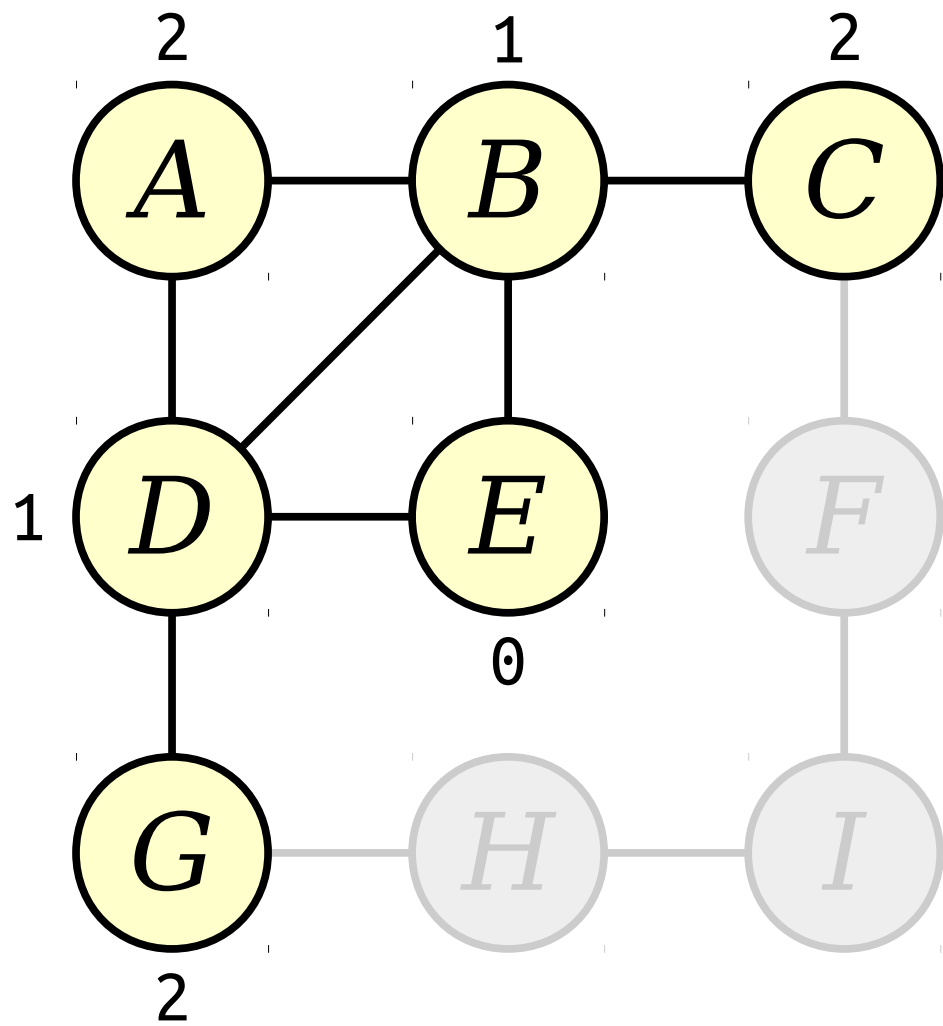
Visit nodes in ascending order of distance from the start node *E*.



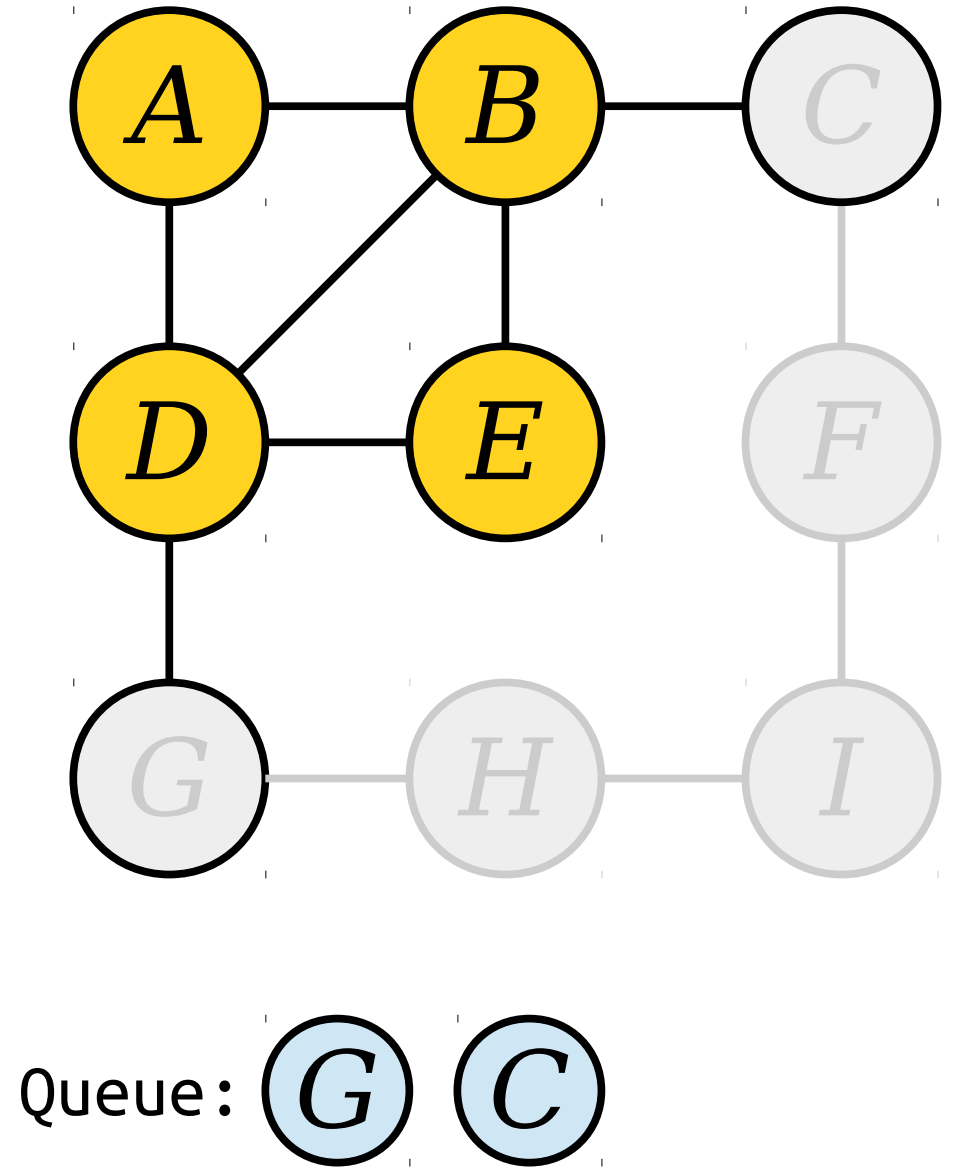
Queue: *G* *C*

Load newly-discovered nodes into a queue.

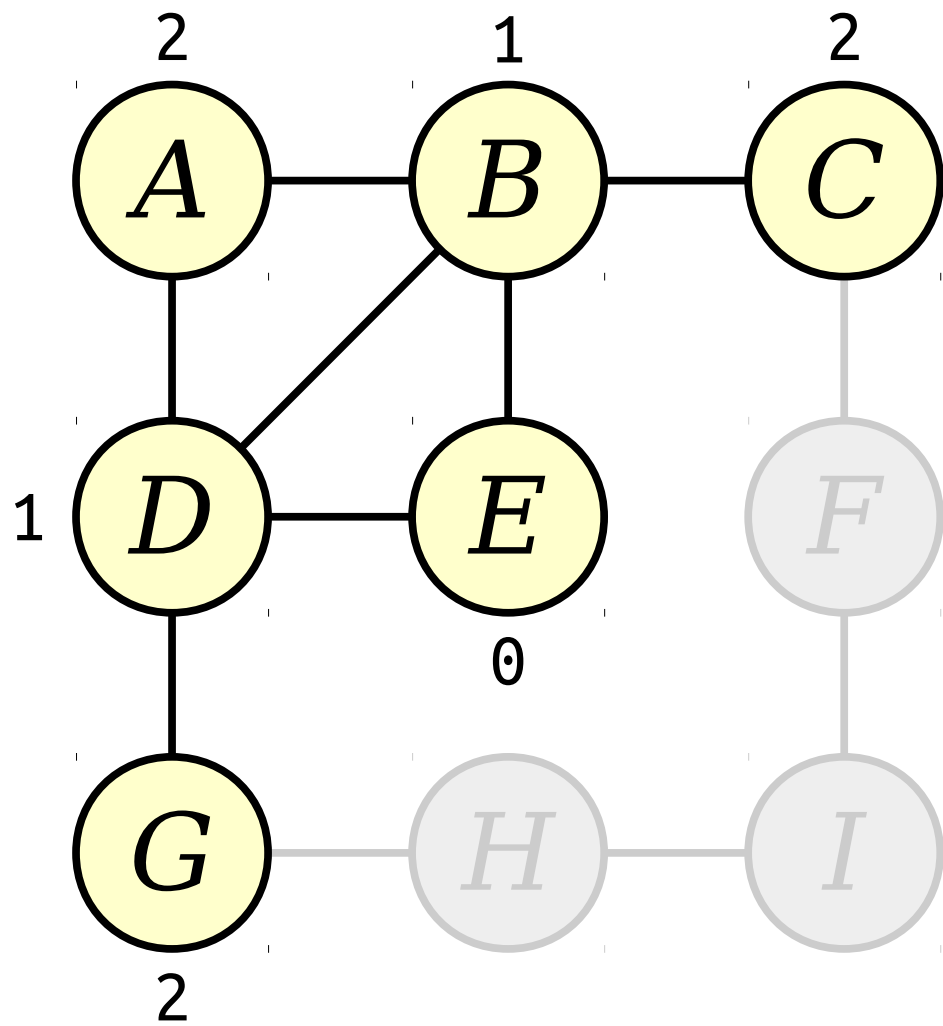




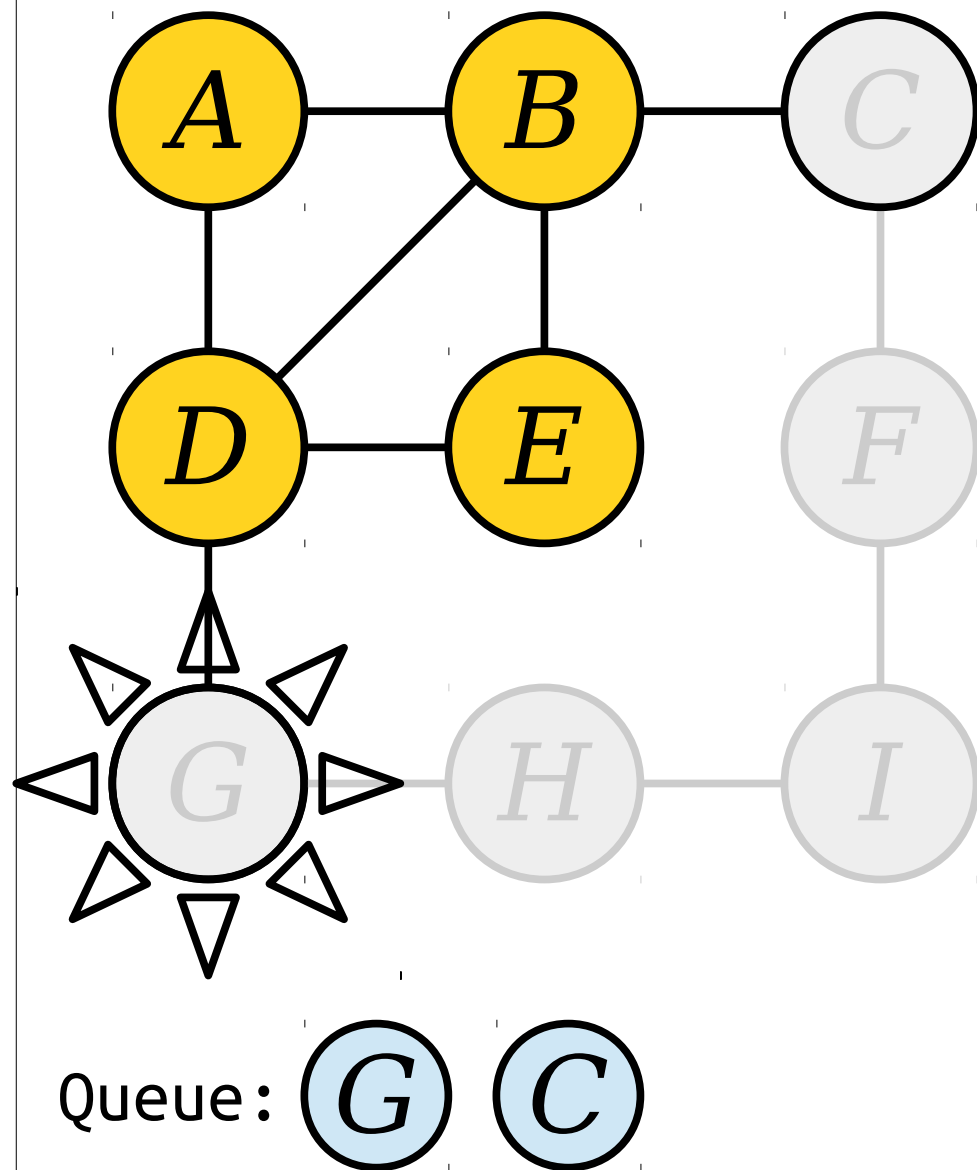
Visit nodes in ascending order of distance from the start node *E*.



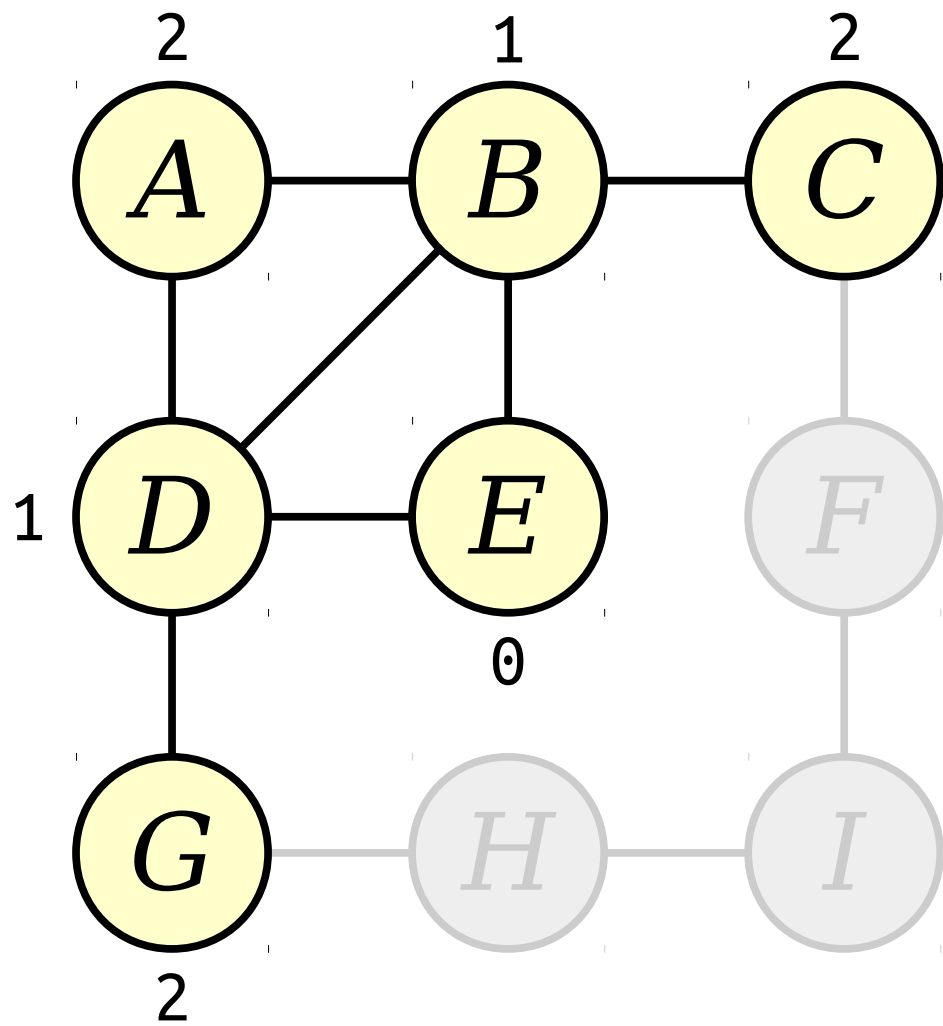
Load newly-discovered nodes into a queue.



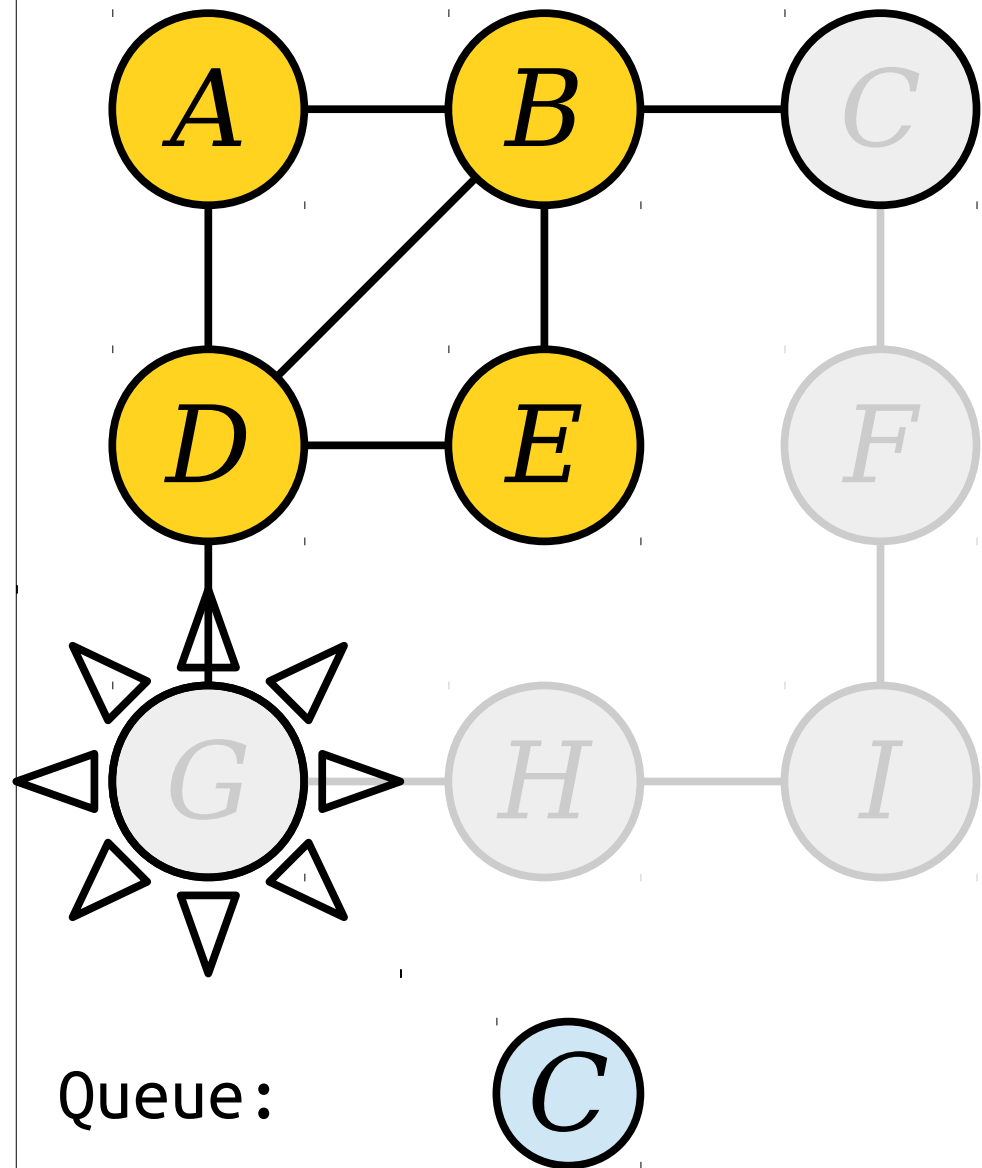
Visit nodes in ascending order of distance from the start node *E*.



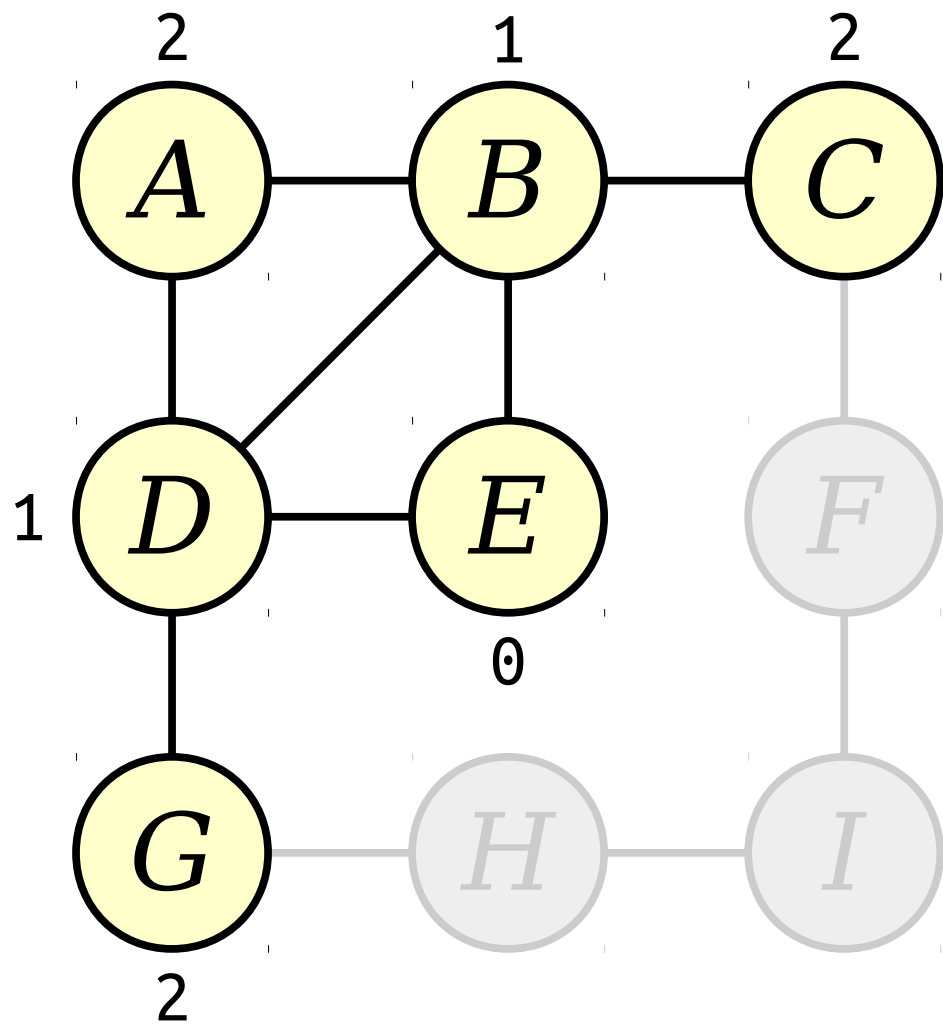
Load newly-discovered nodes into a queue.



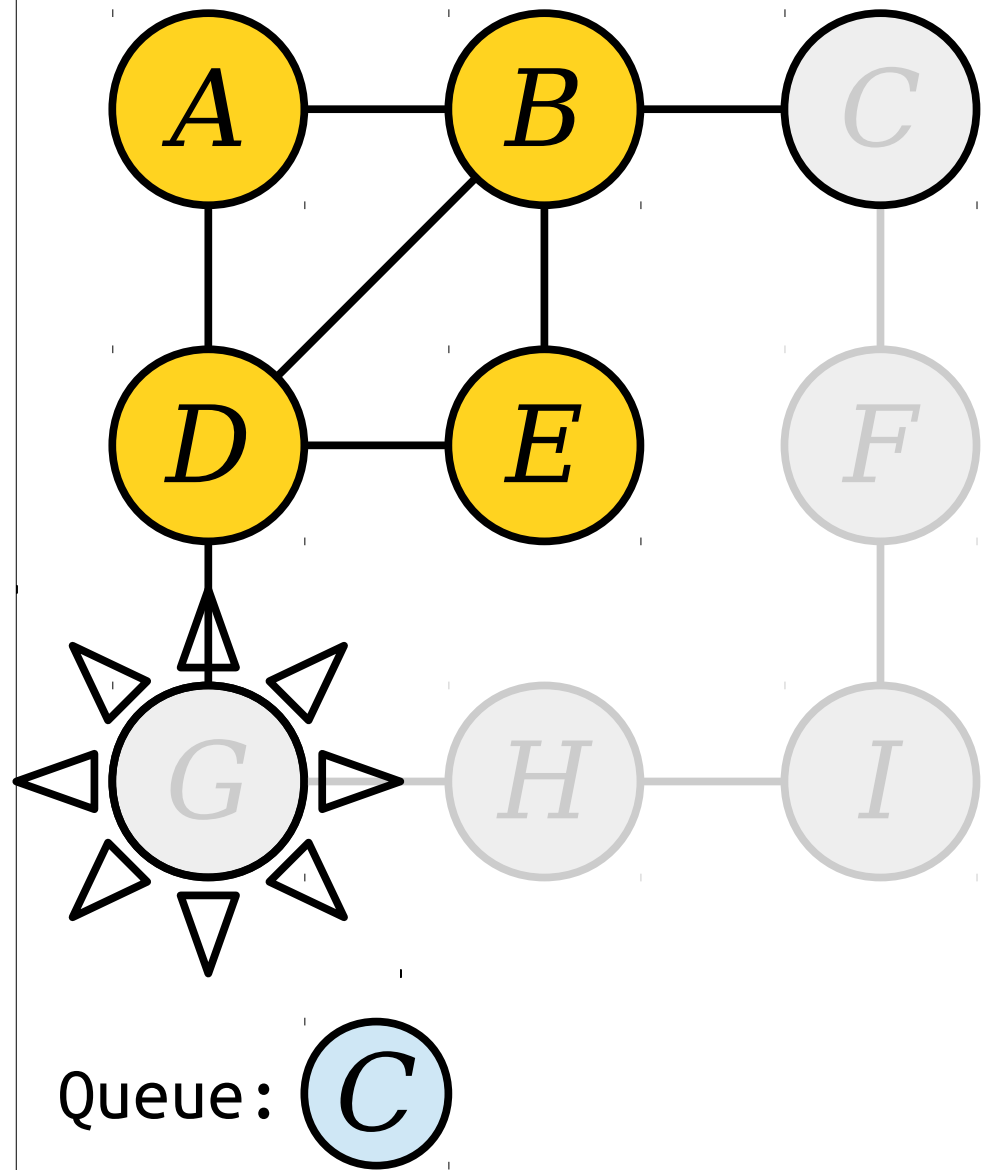
Visit nodes in ascending order of distance from the start node  $E$ .



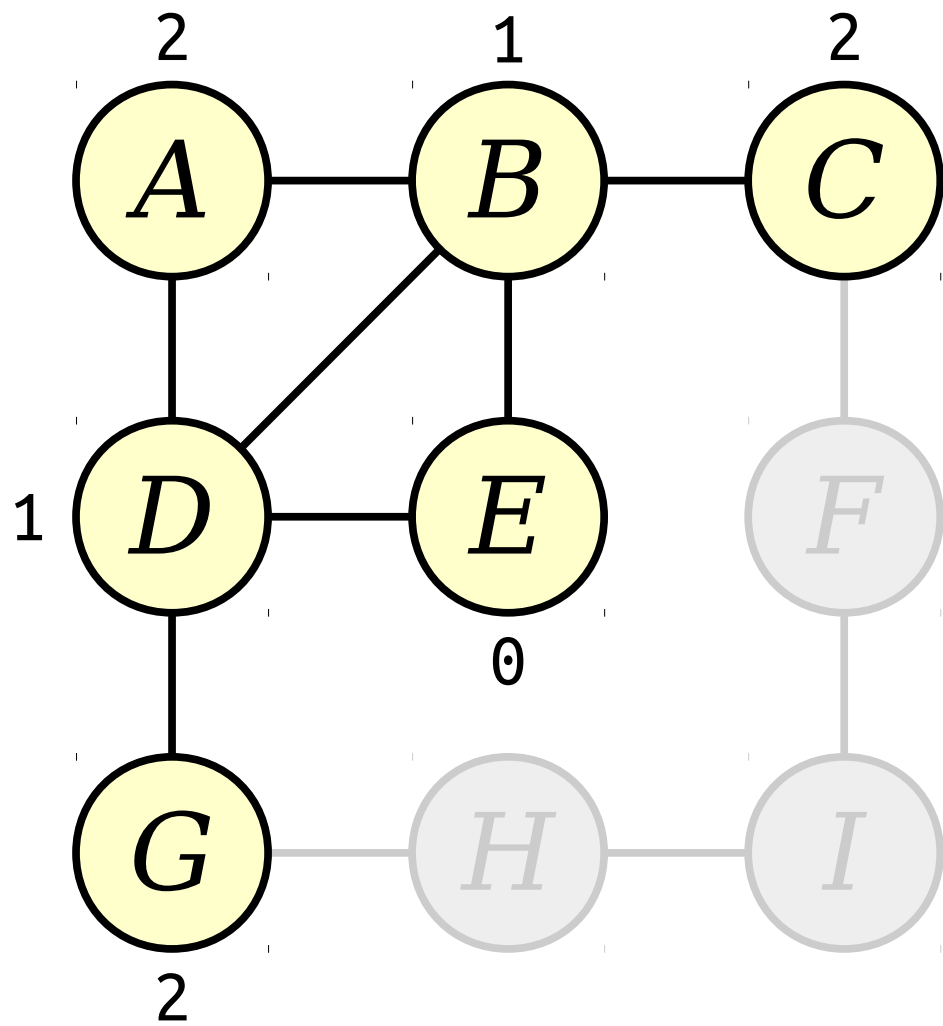
Load newly-discovered nodes into a queue.



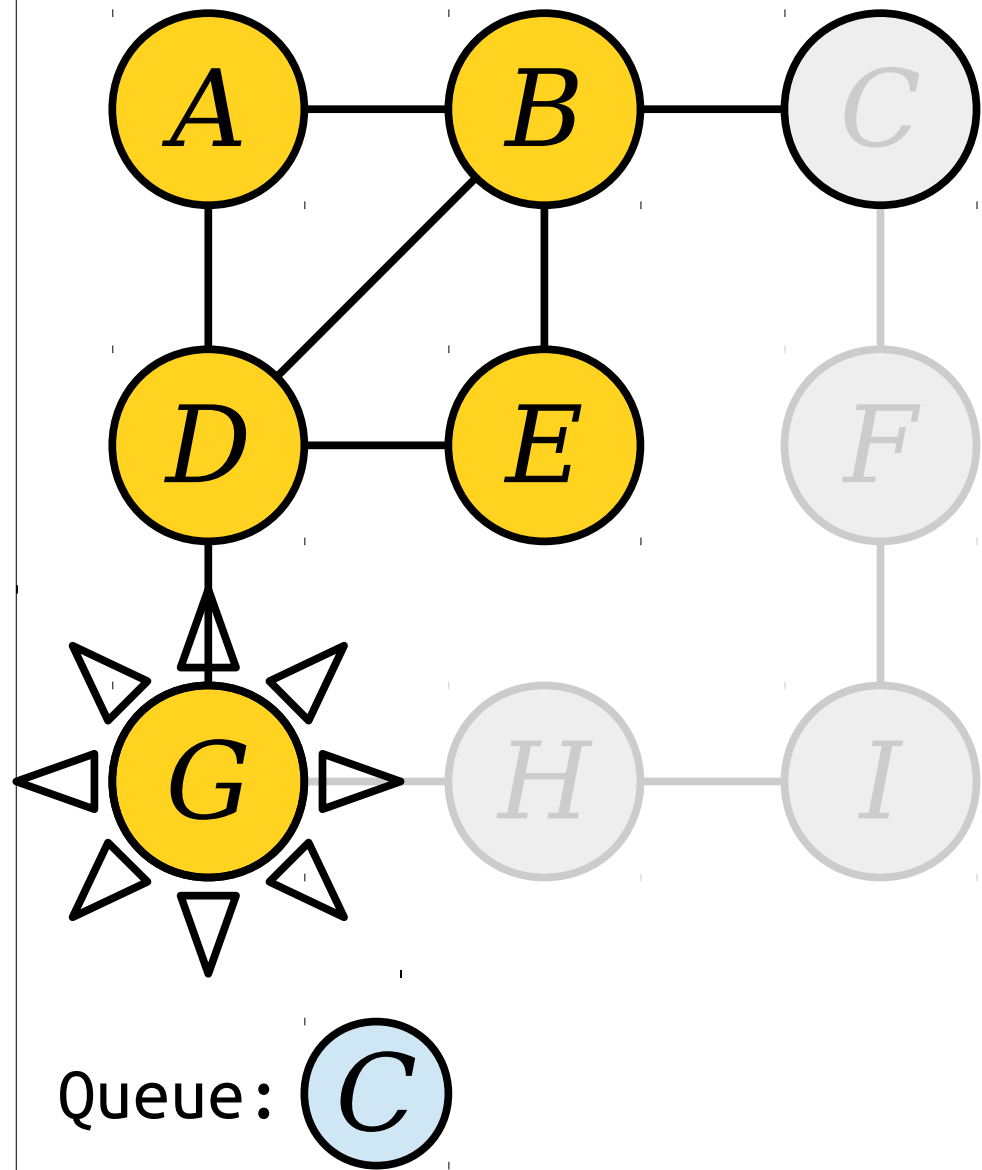
Visit nodes in ascending order of distance from the start node *E*.



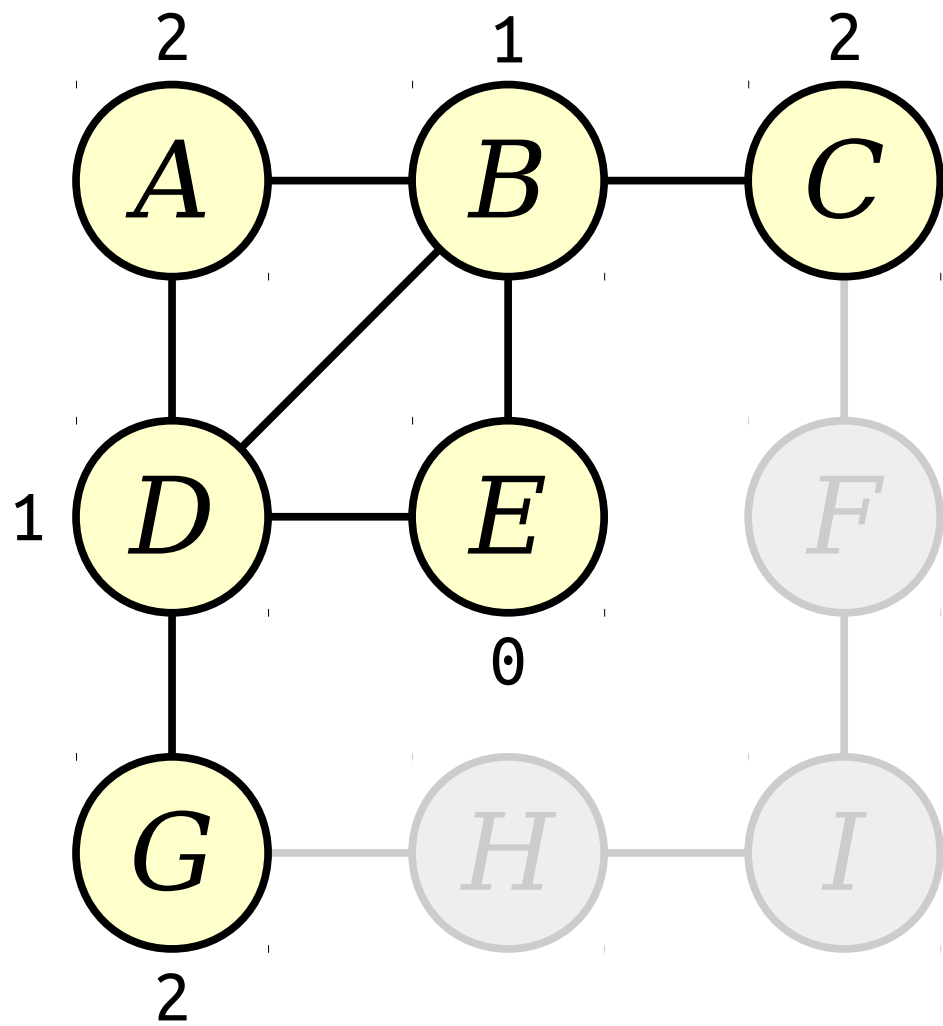
Load newly-discovered nodes into a queue.



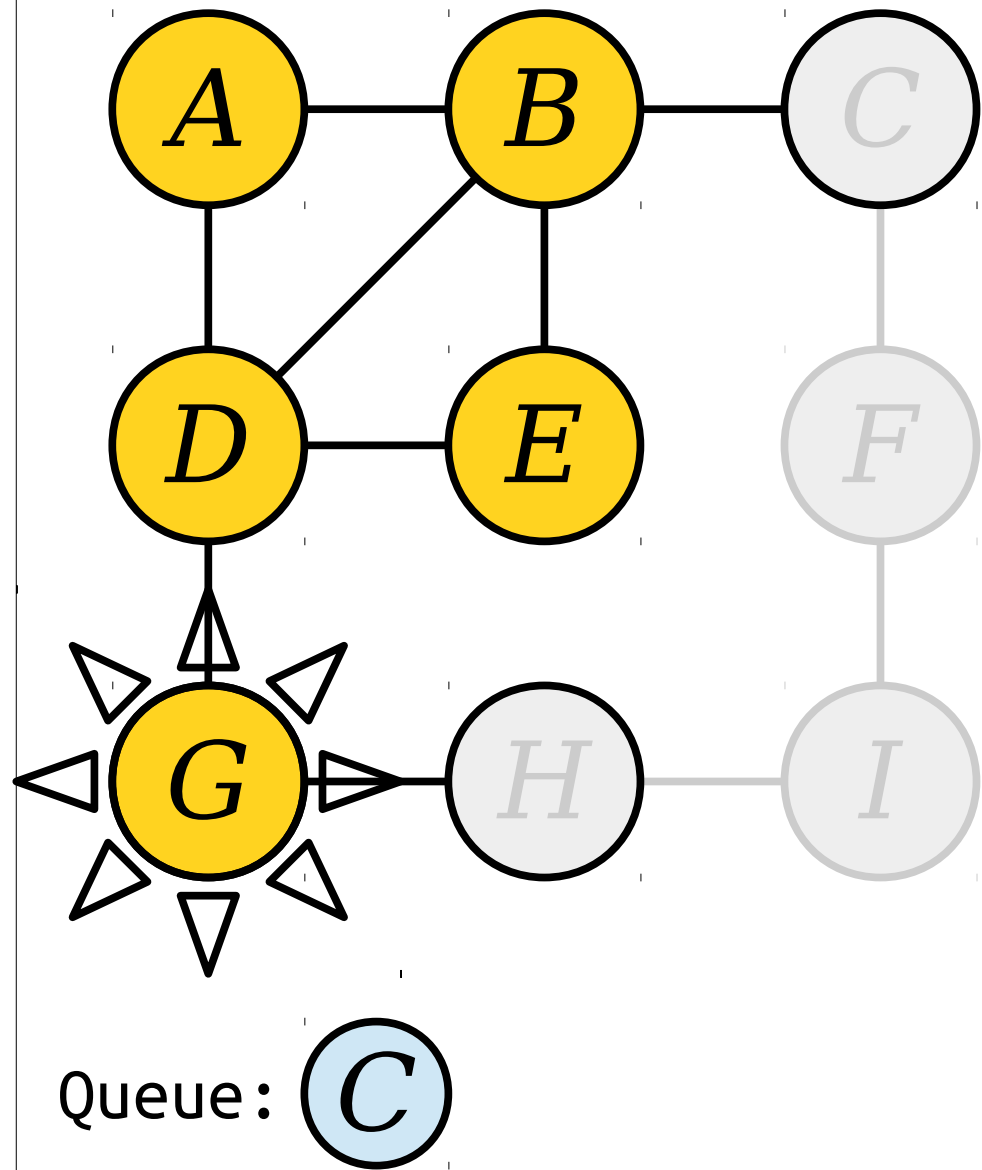
Visit nodes in ascending order of distance from the start node *E*.



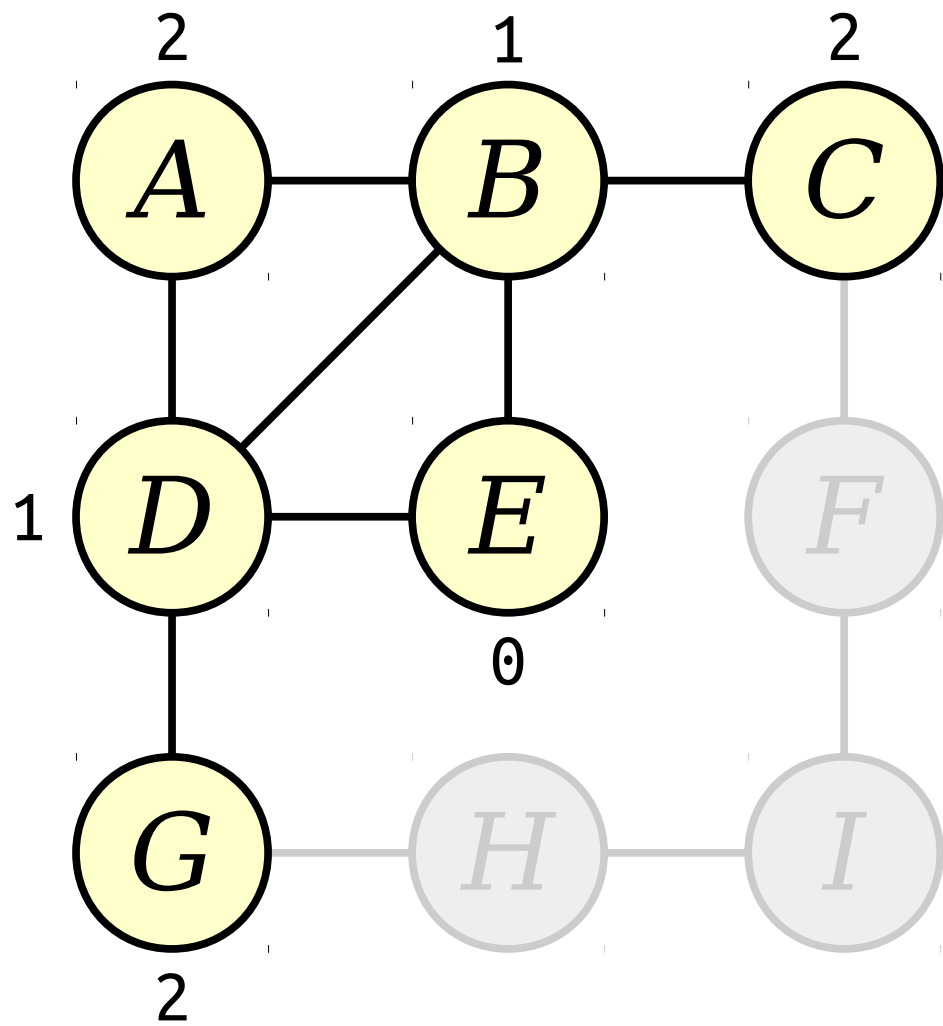
Load newly-discovered nodes into a queue.



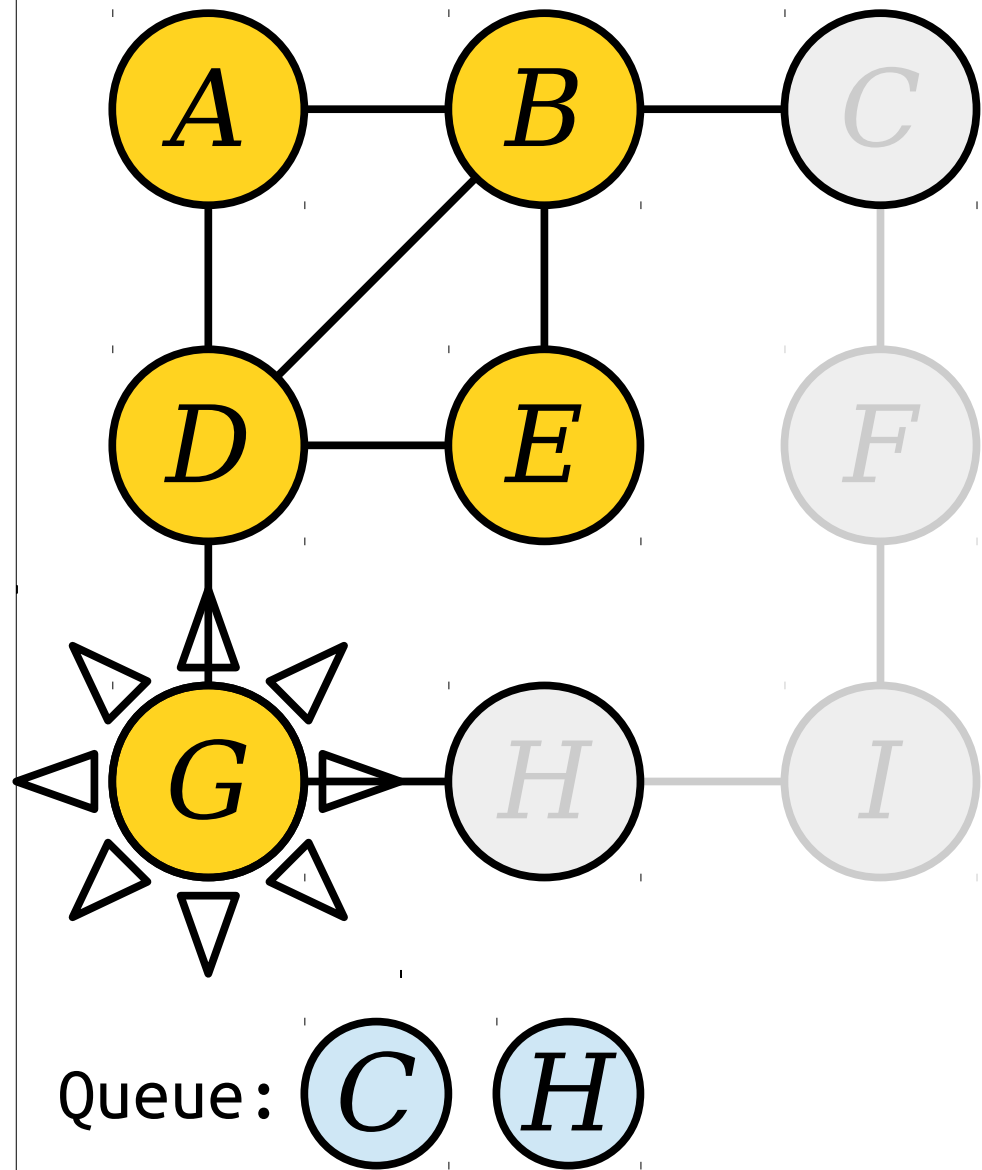
Visit nodes in ascending order of distance from the start node *E*.



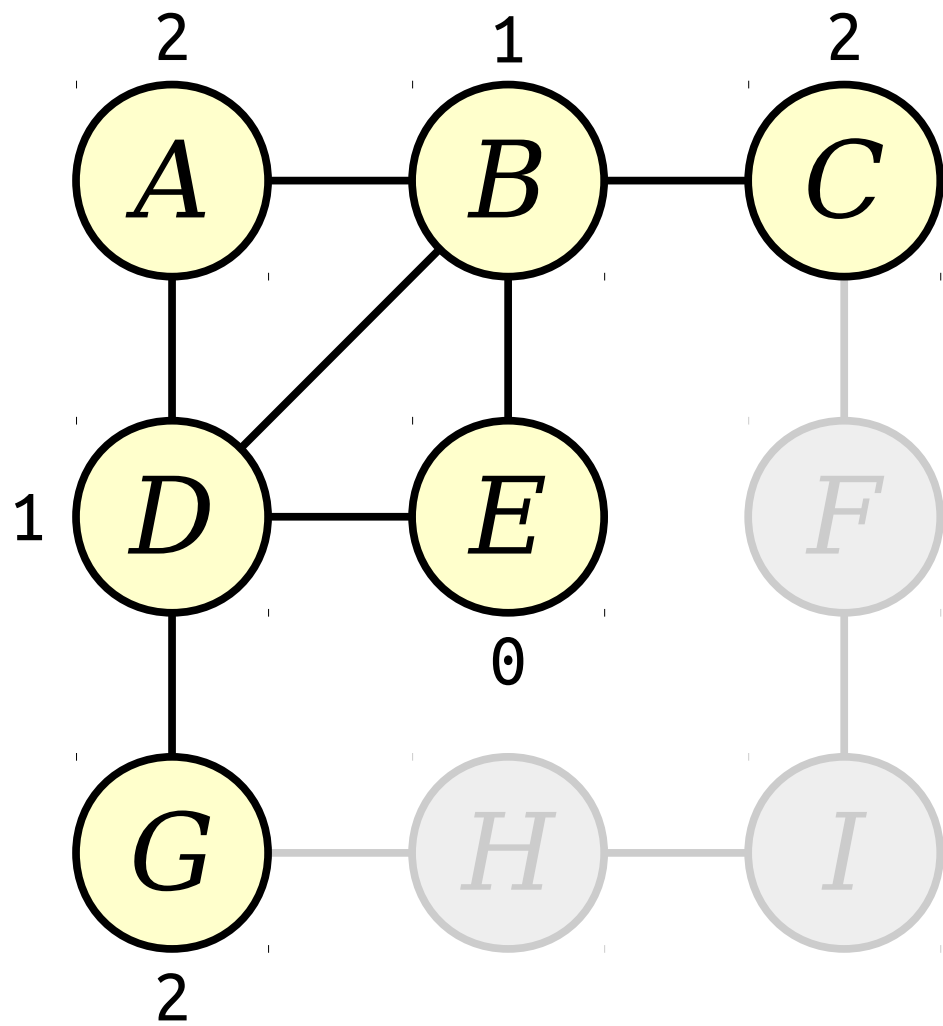
Load newly-discovered nodes into a queue.



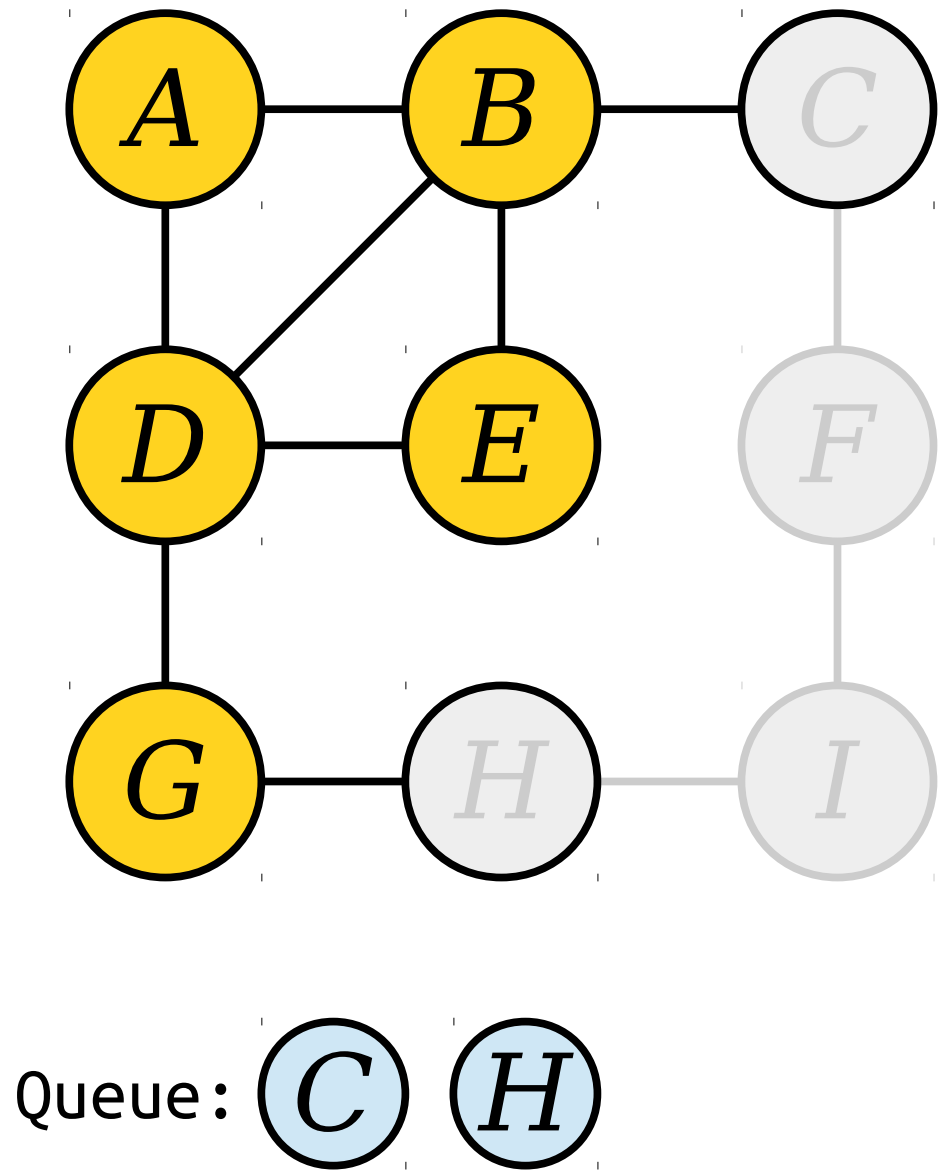
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

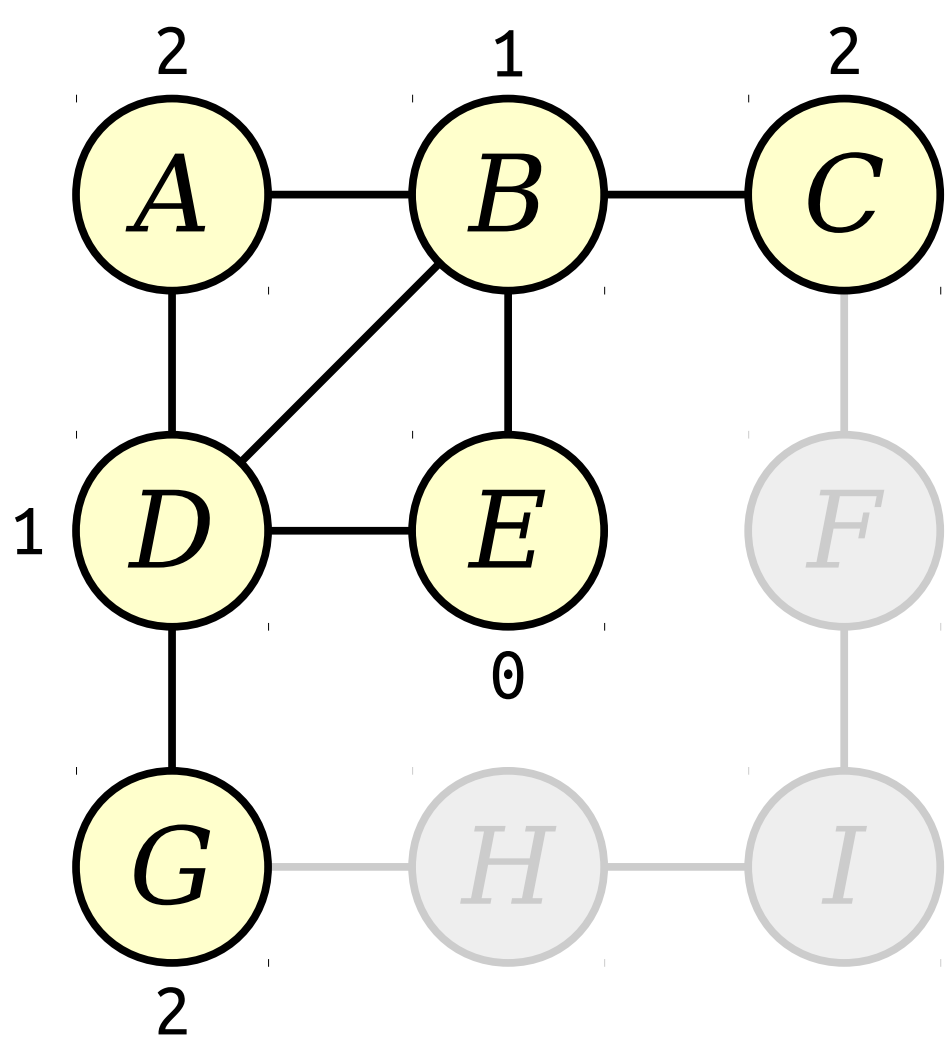


Visit nodes in ascending order of distance from the start node *E*.

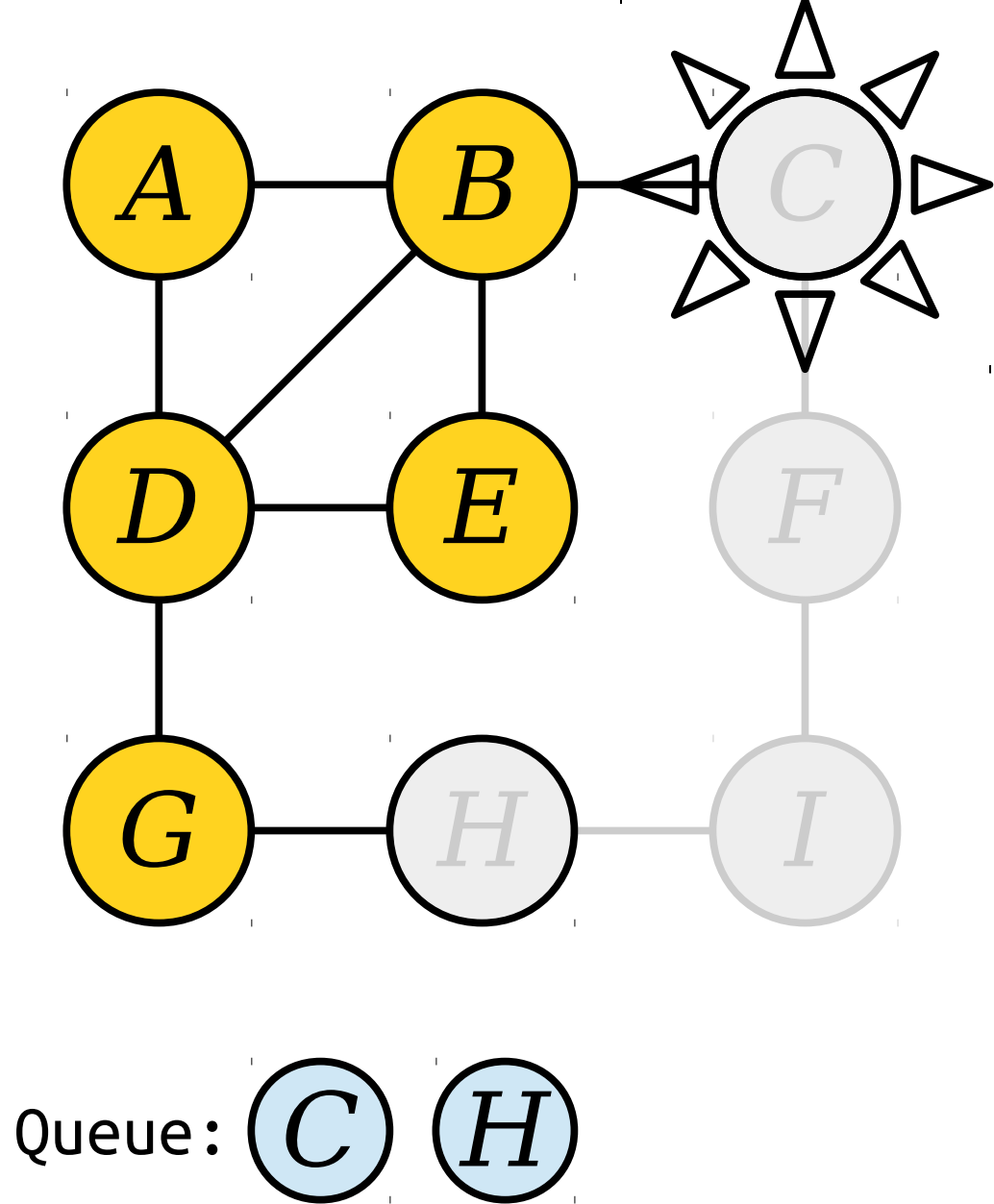


Load newly-discovered nodes into a queue.

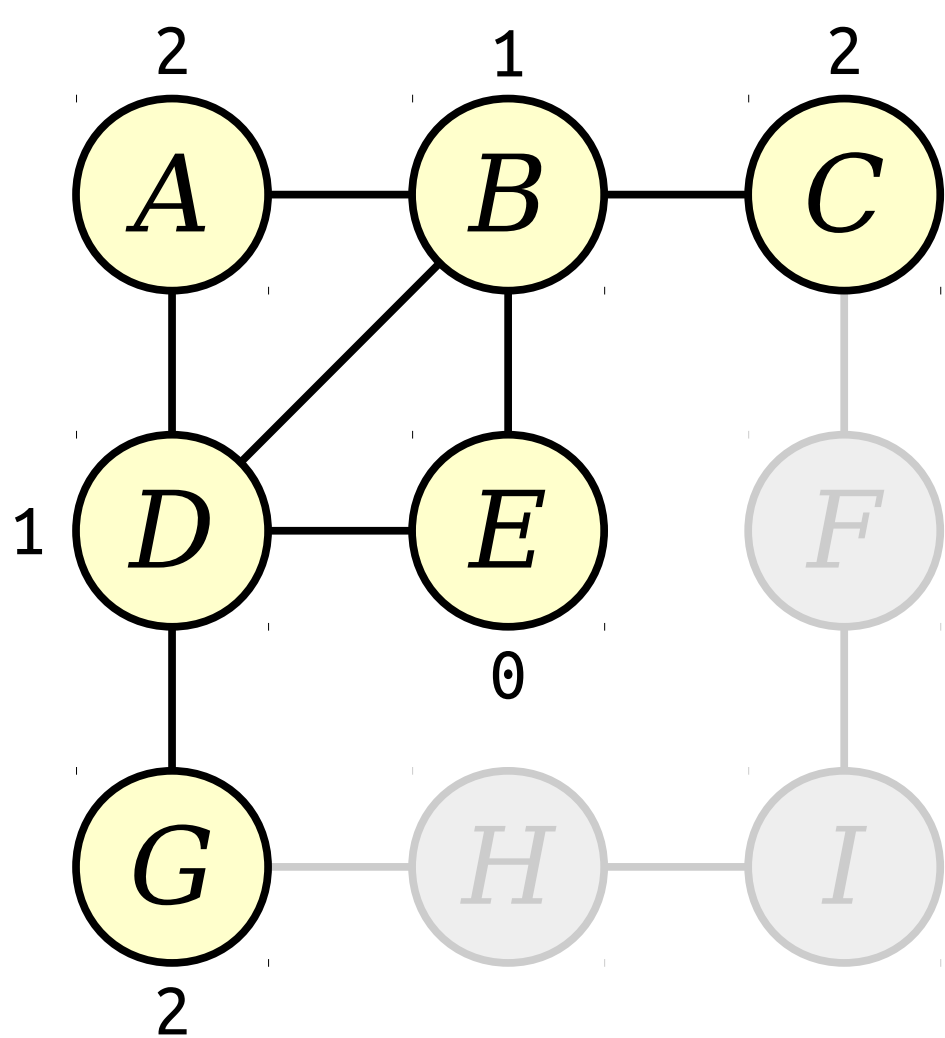




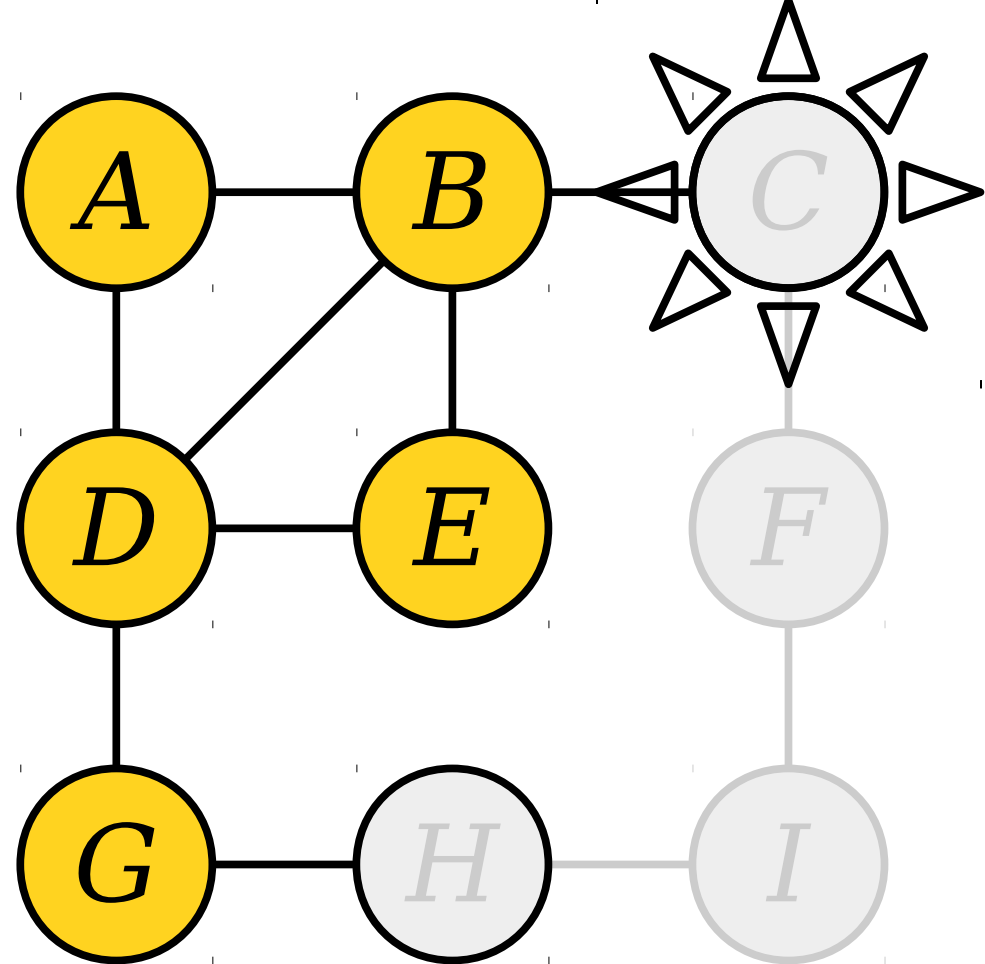
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.



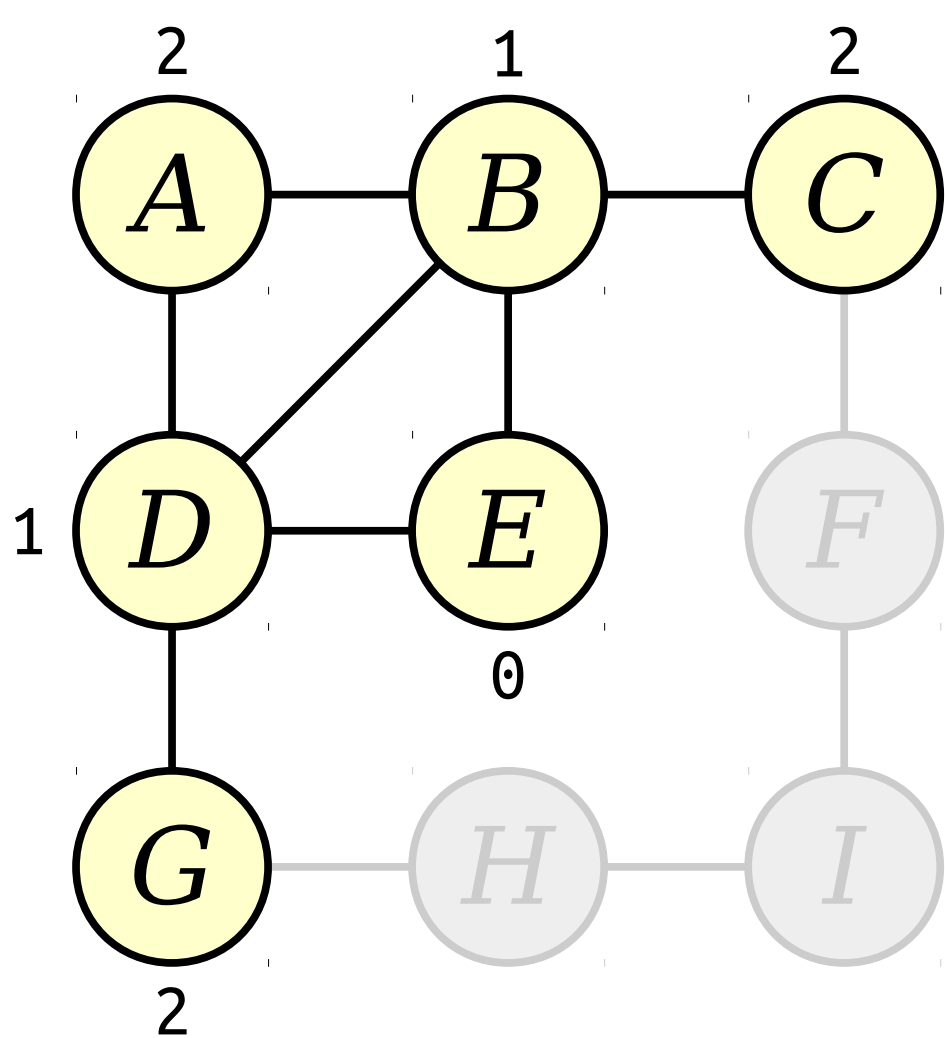
Visit nodes in ascending order of distance from the start node *E*.



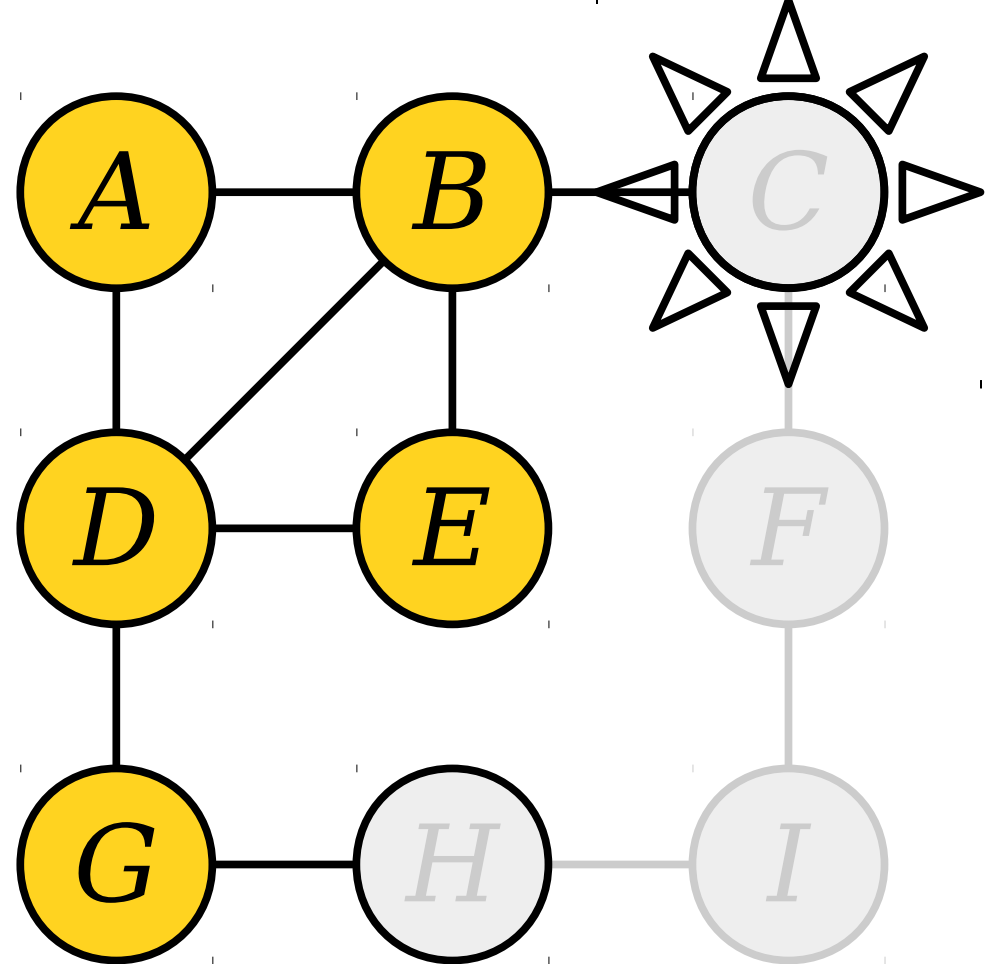
Queue:



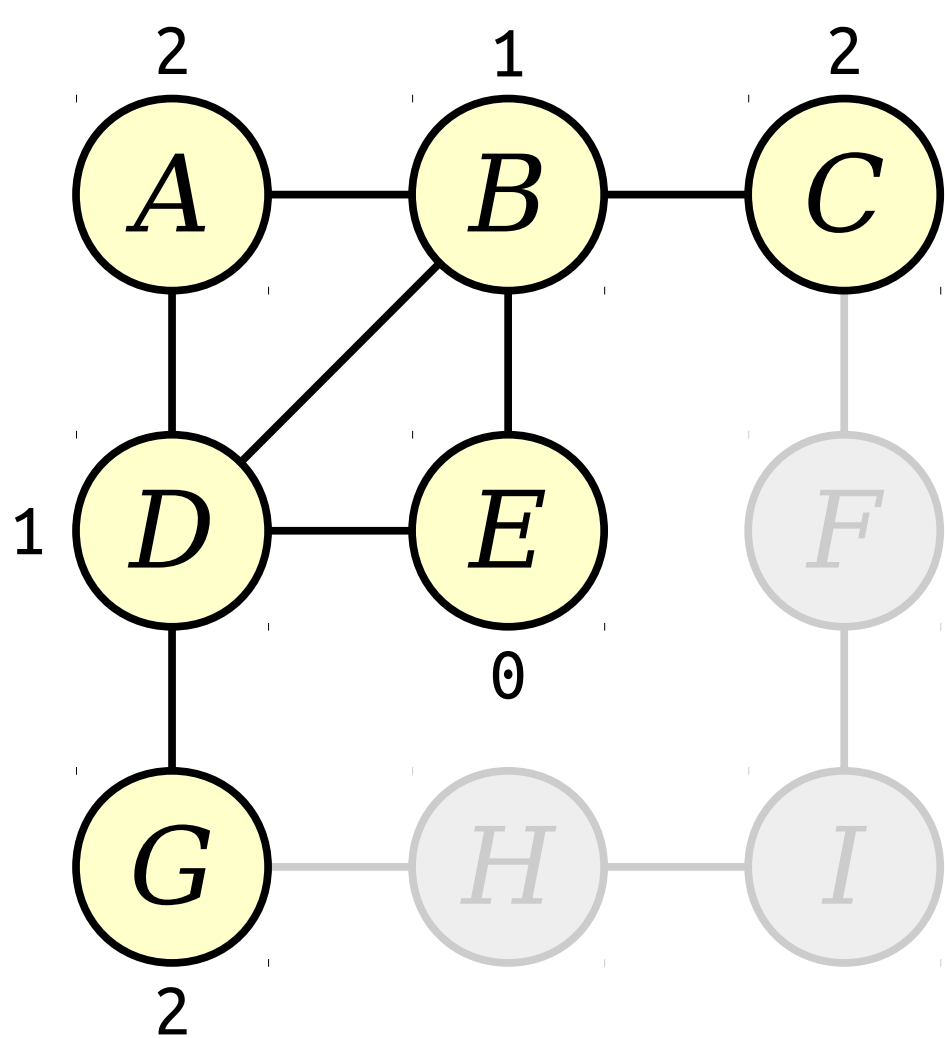
Load newly-discovered nodes into a queue.



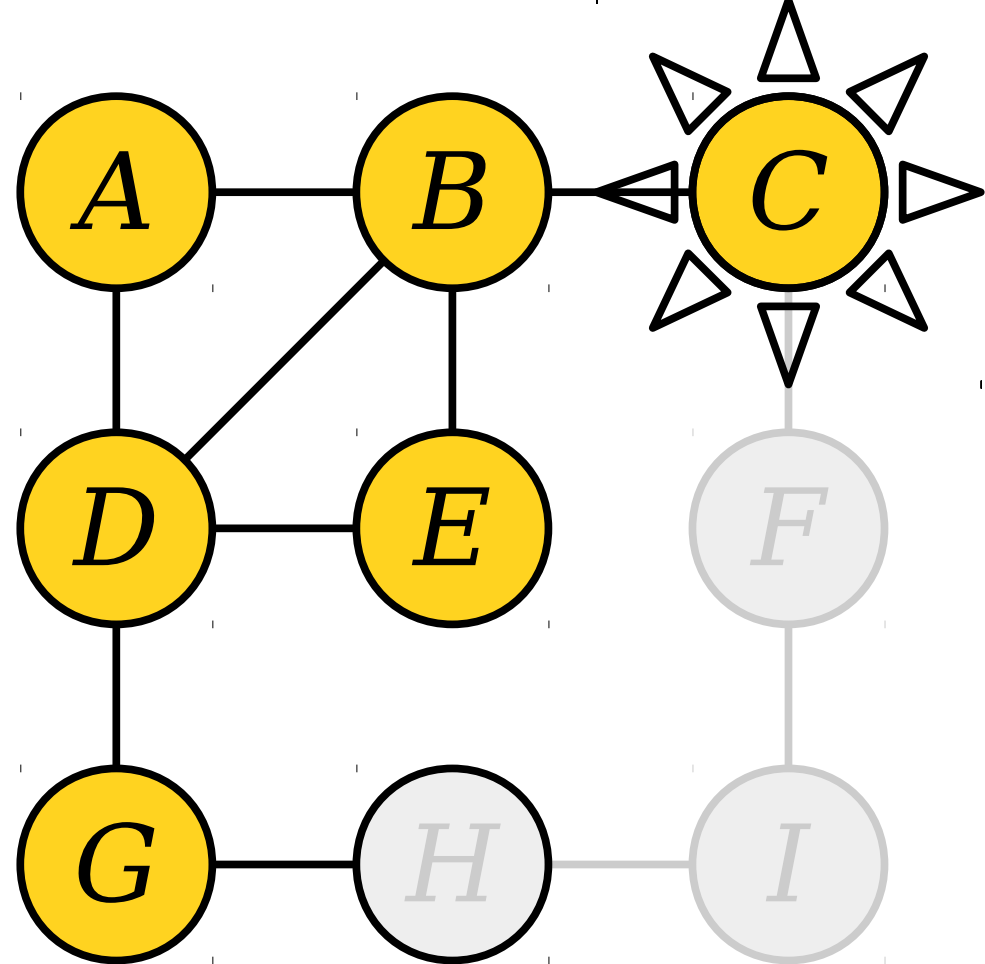
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

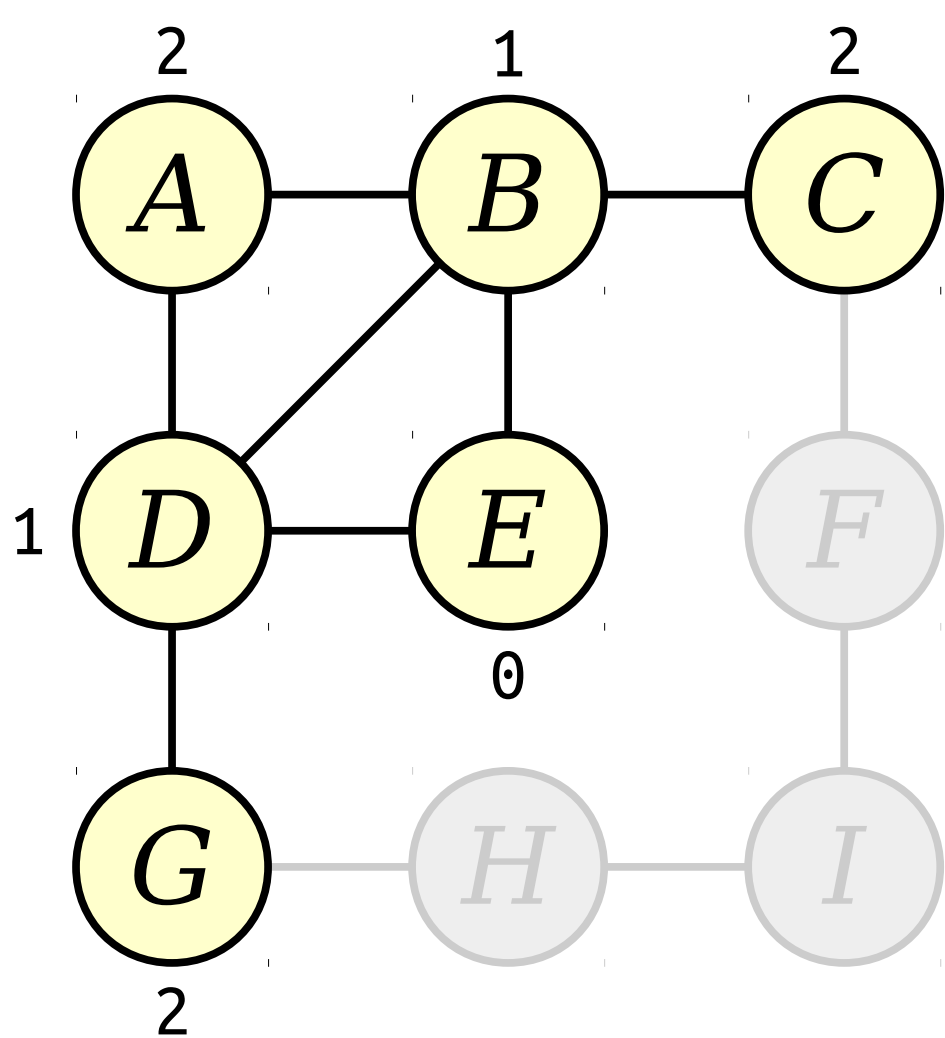


Visit nodes in ascending order of distance from the start node *E*.

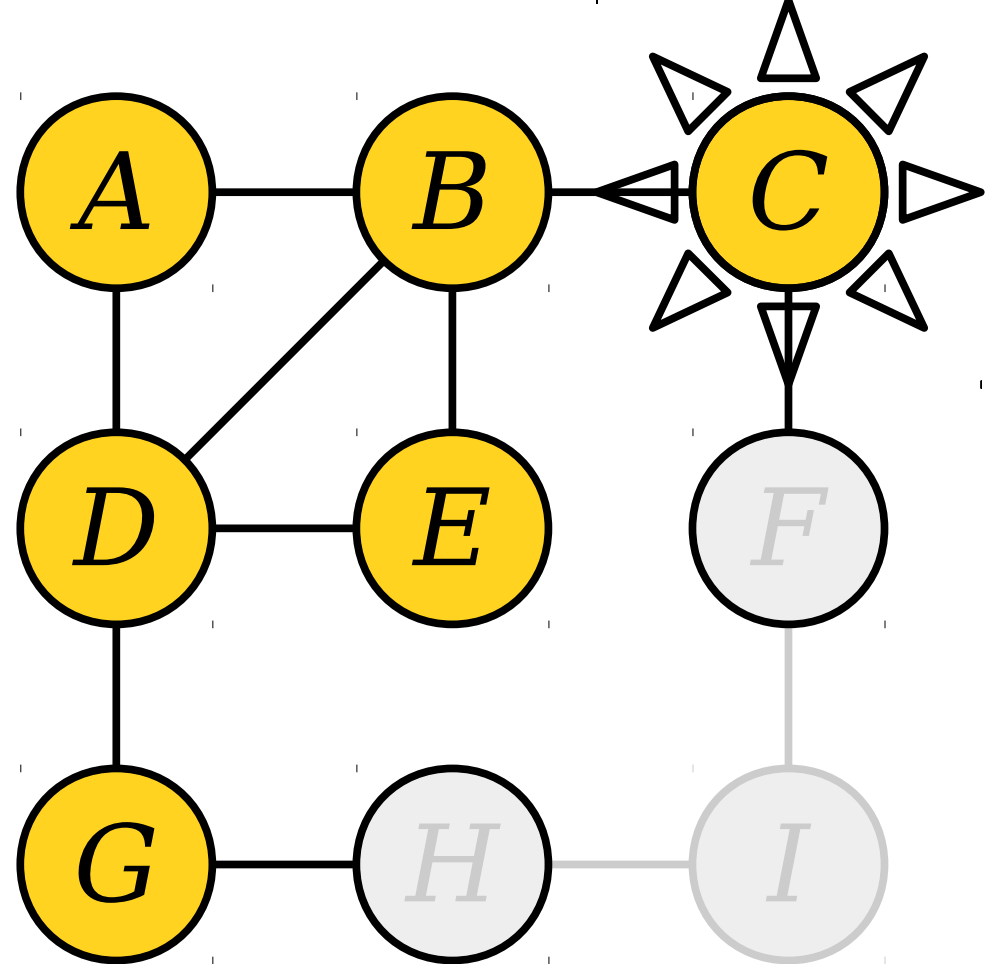


Queue: *H*

Load newly-discovered nodes into a queue.

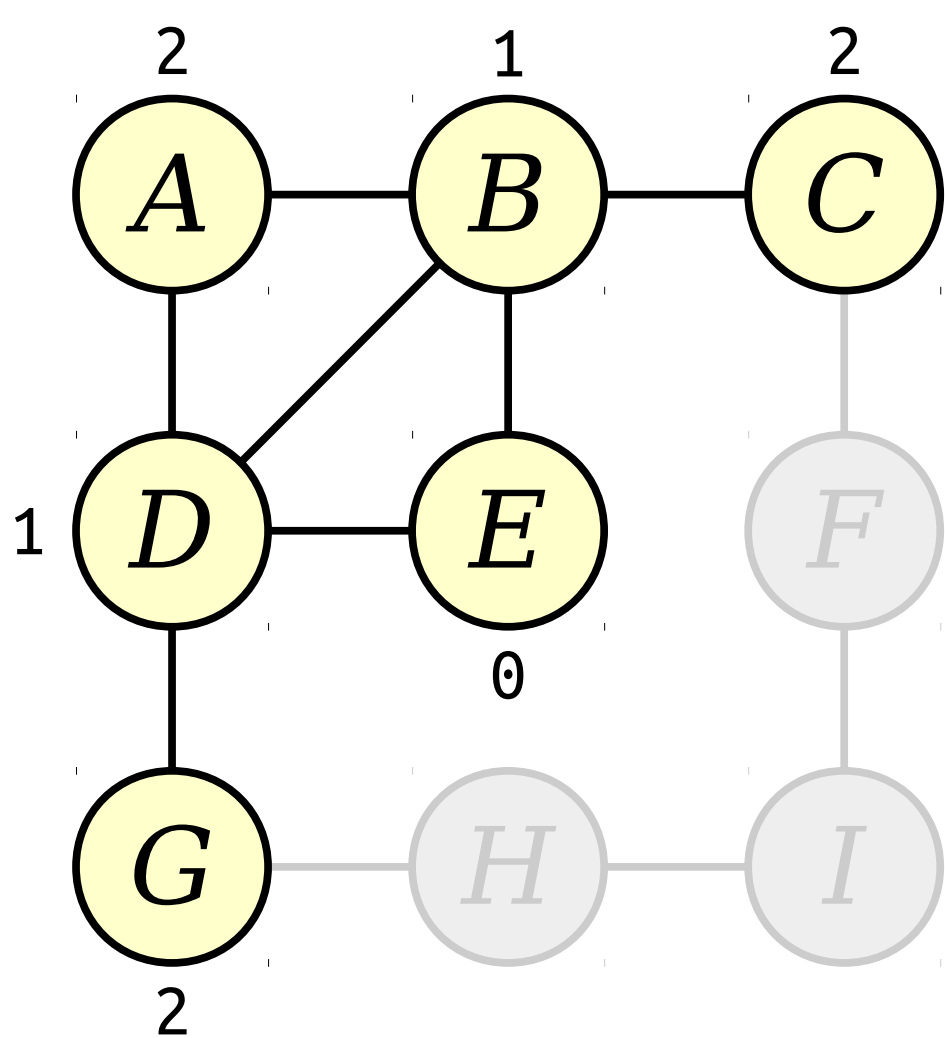


Visit nodes in ascending order of distance from the start node *E*.

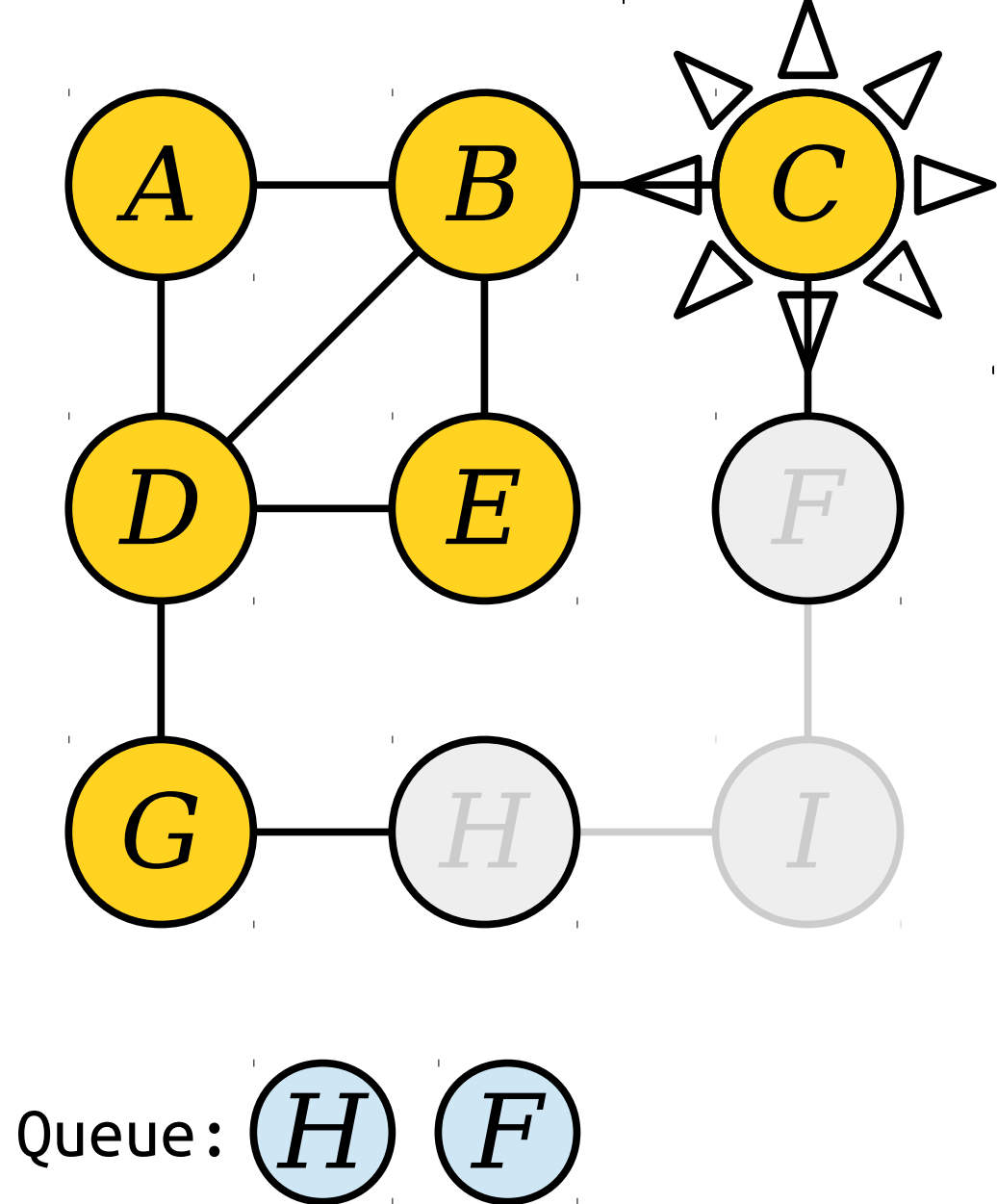


Queue: *H*

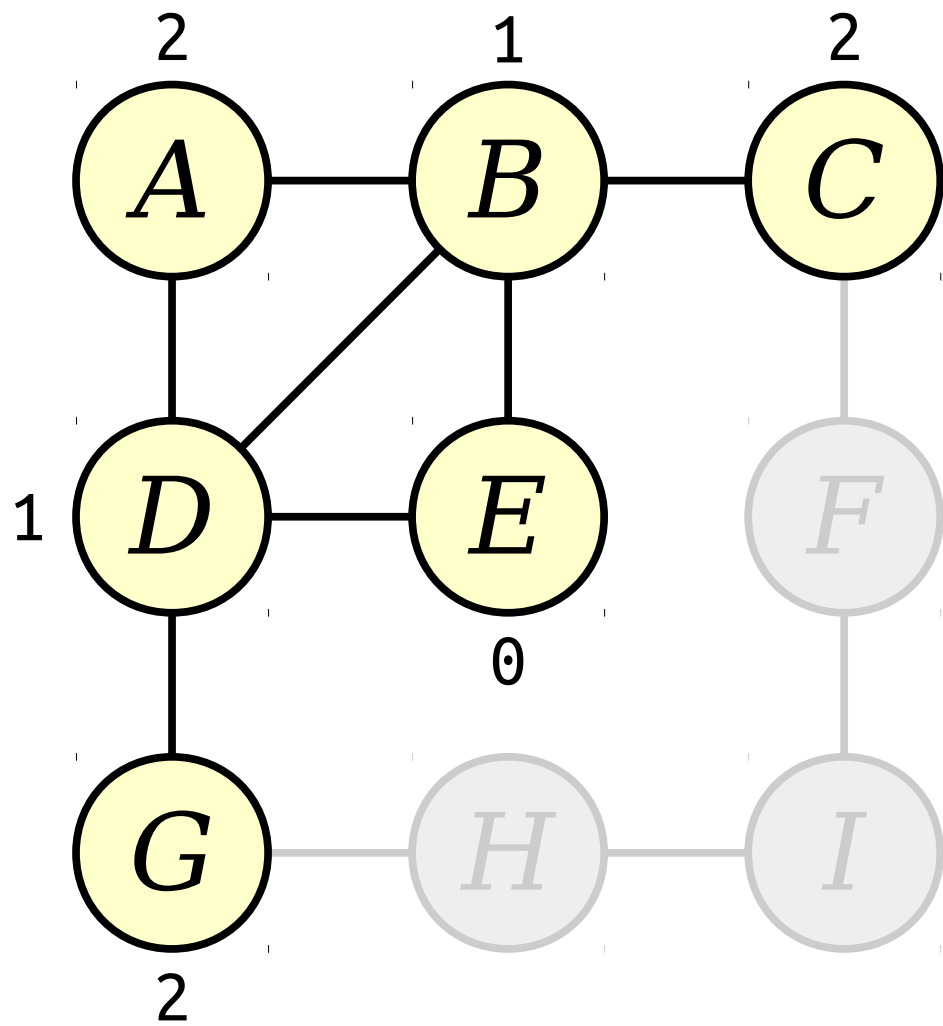
Load newly-discovered nodes into a queue.



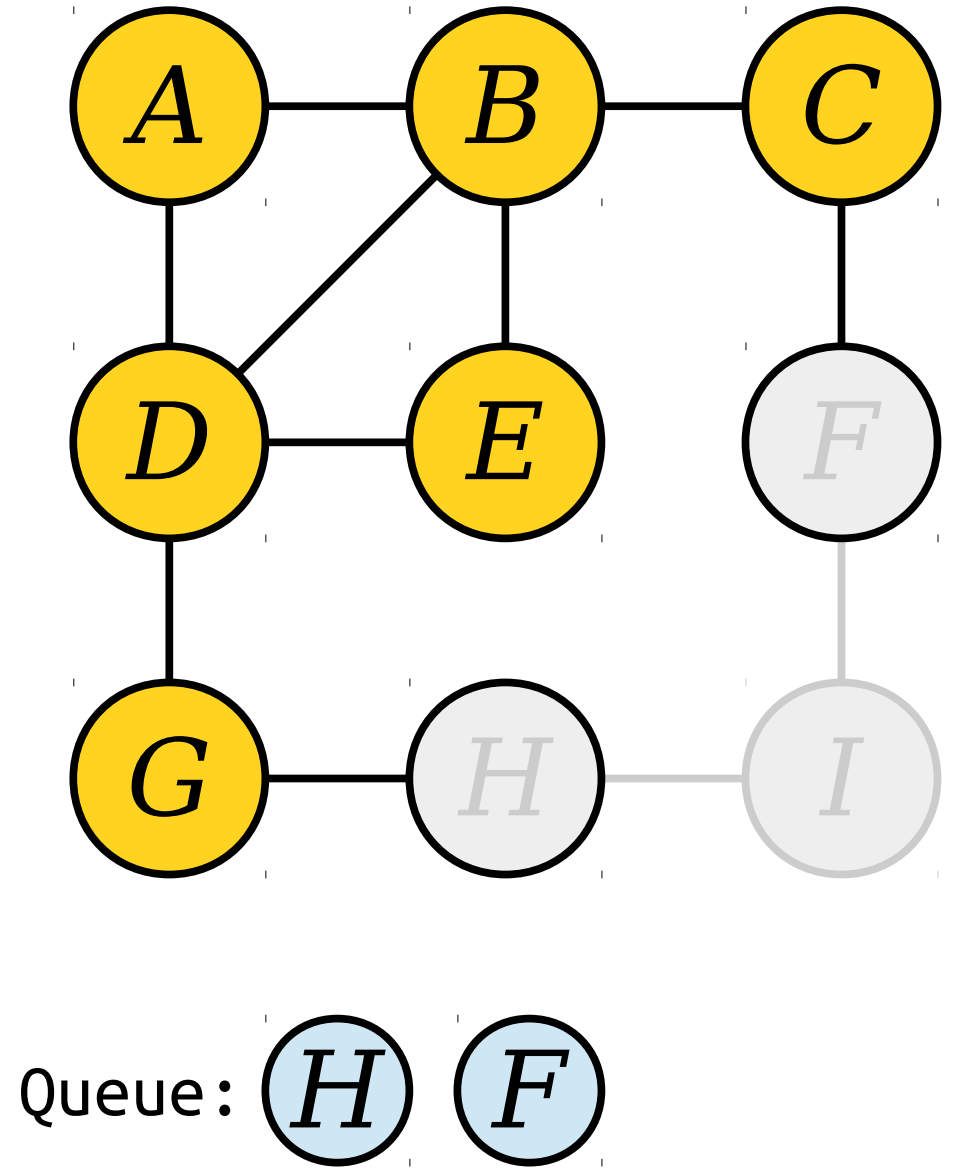
Visit nodes in ascending order of distance from the start node *E*.



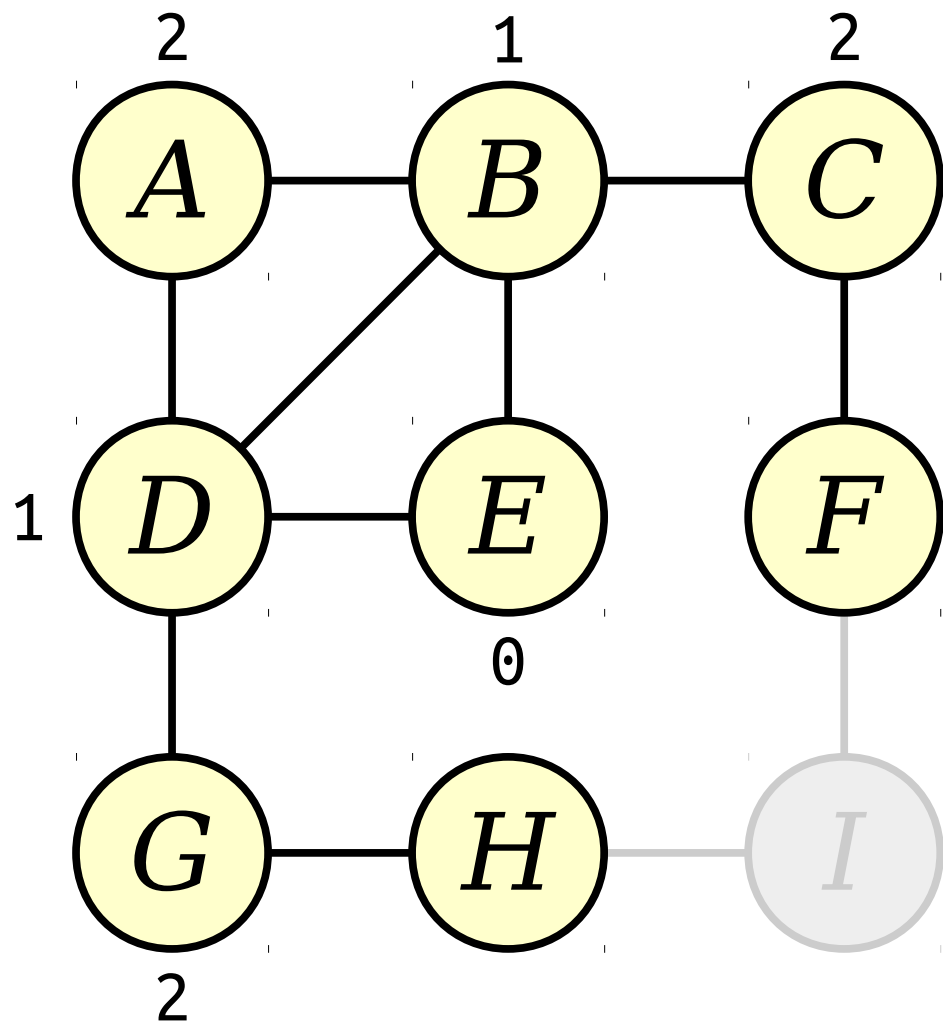
Load newly-discovered nodes into a queue.



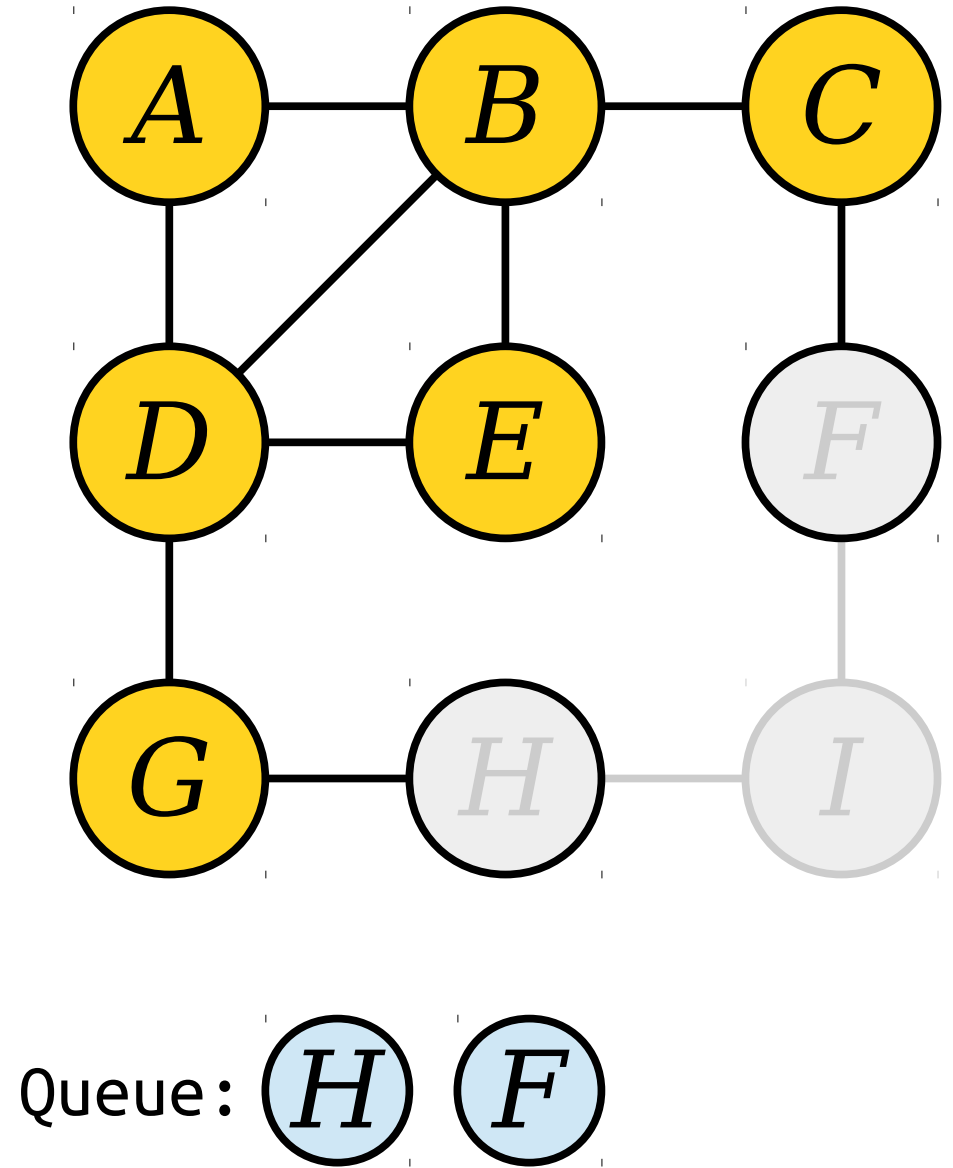
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

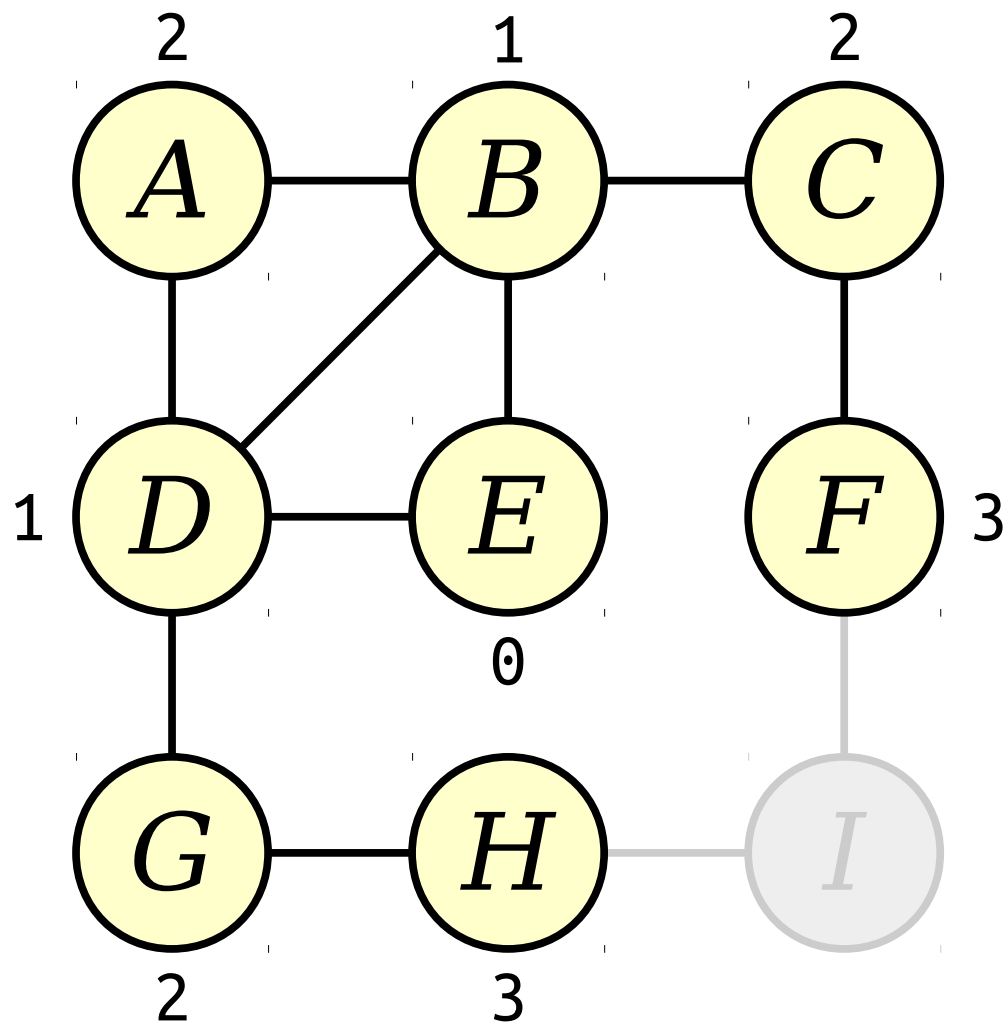


Visit nodes in ascending order of distance from the start node *E*.

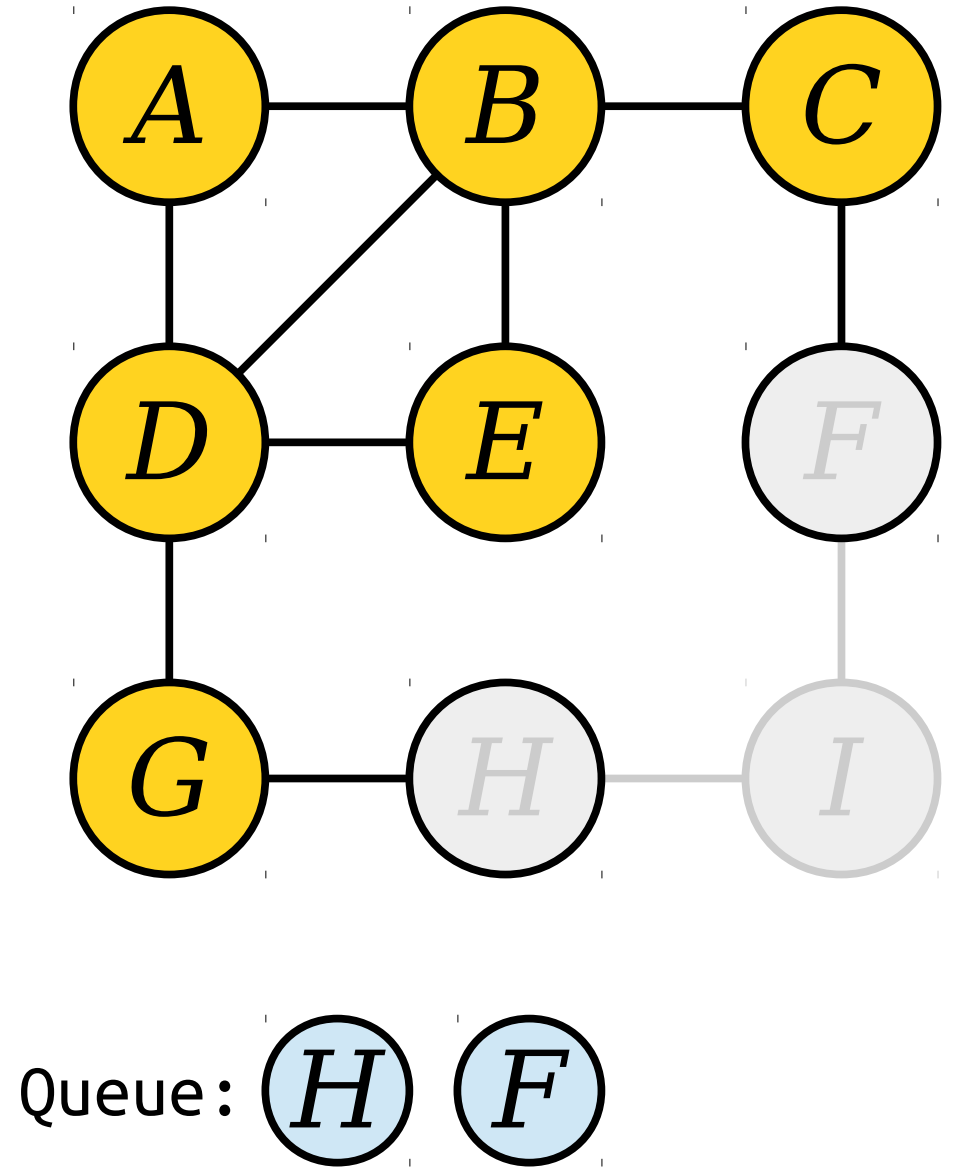


Load newly-discovered nodes into a queue.

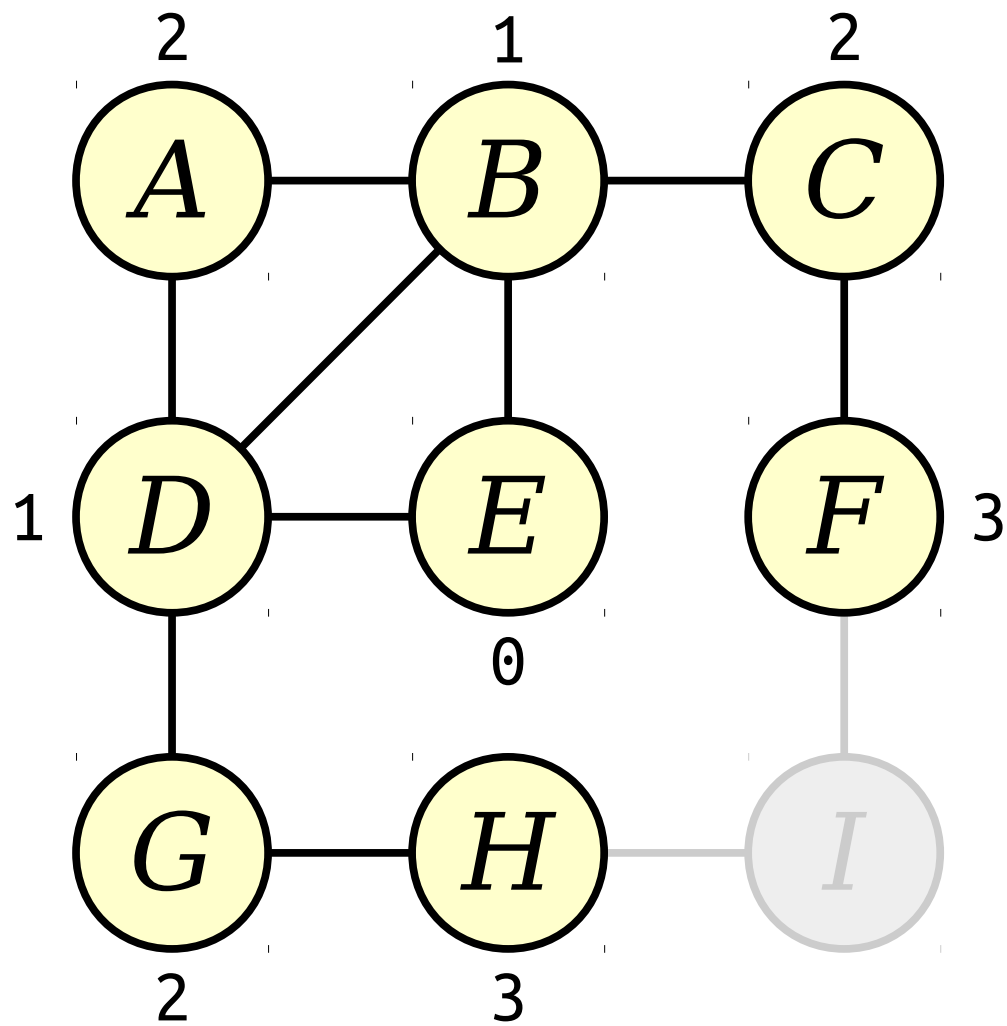




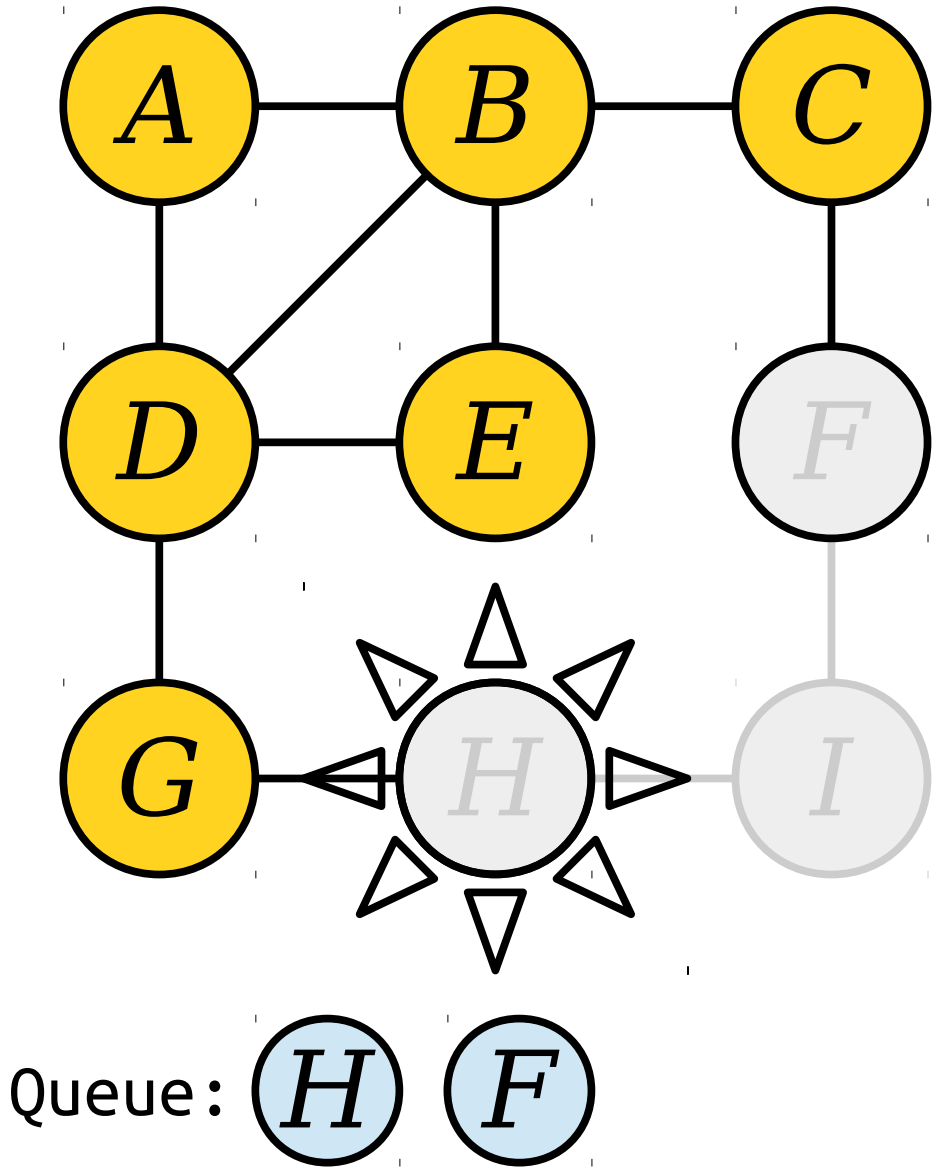
Visit nodes in ascending order of distance from the start node *E*.



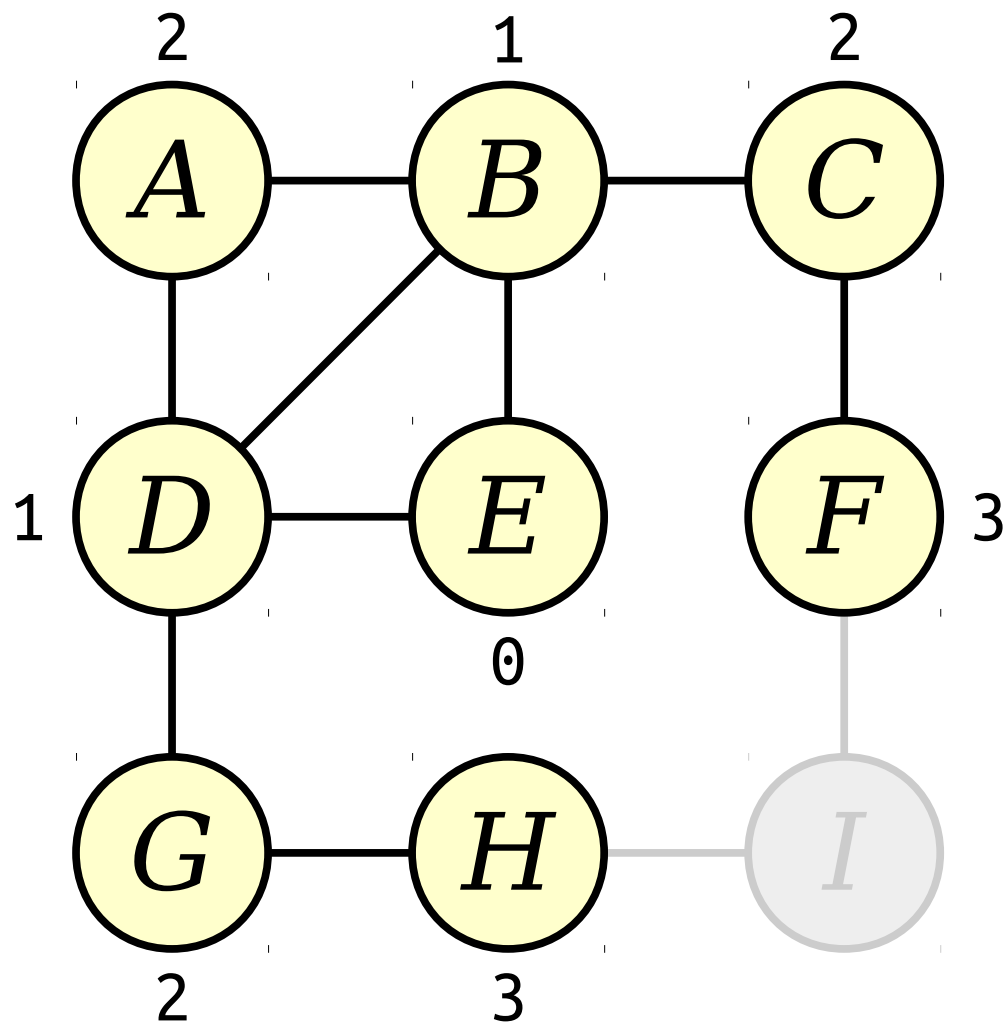
Load newly-discovered nodes into a queue.



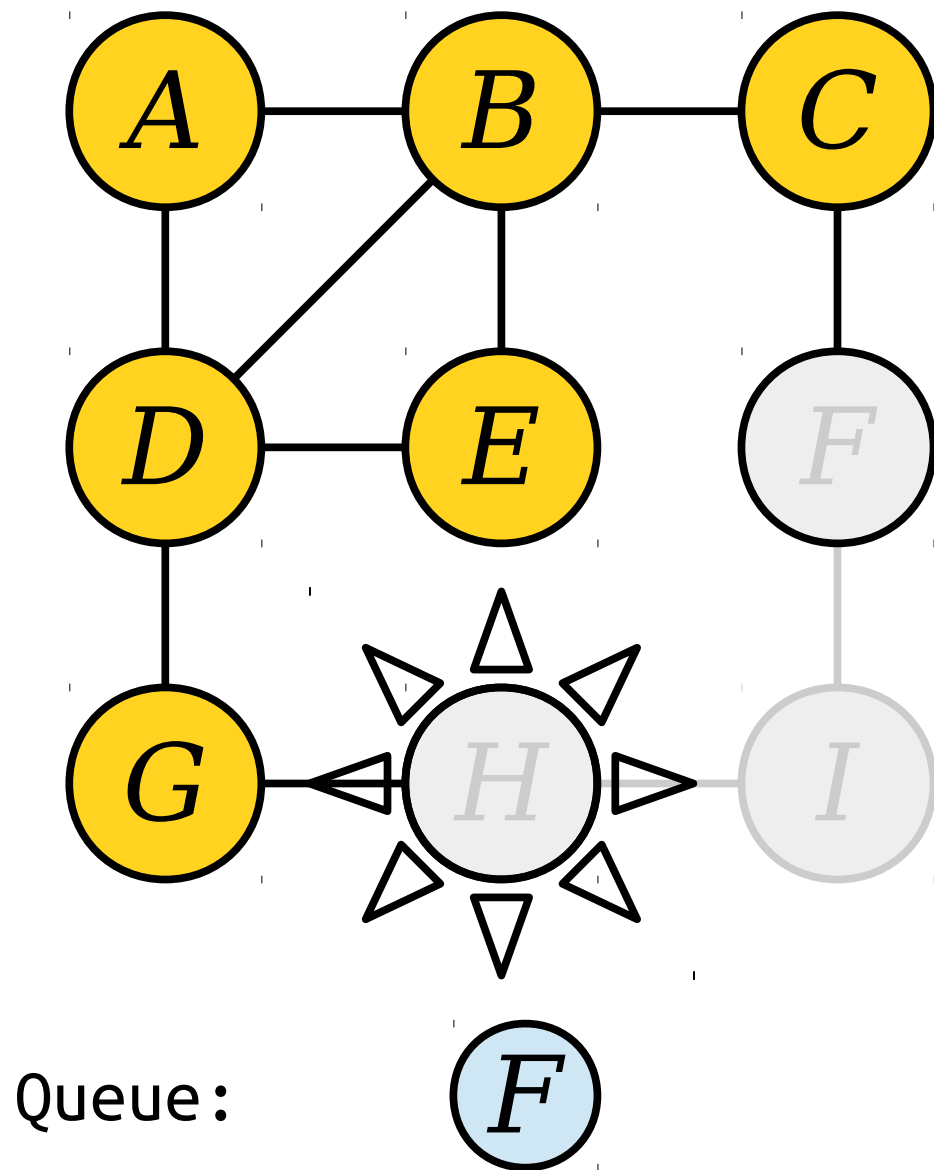
Visit nodes in ascending order of distance from the start node *E*.



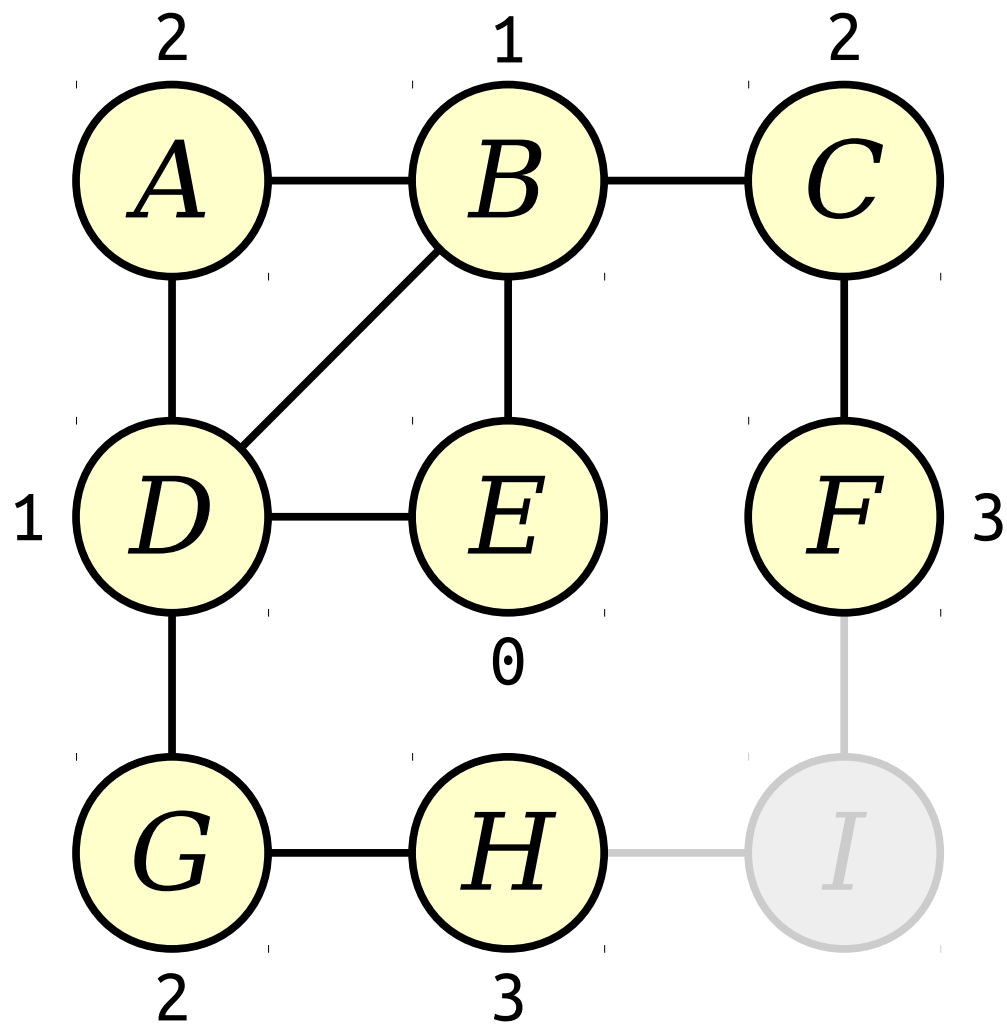
Load newly-discovered nodes into a queue.



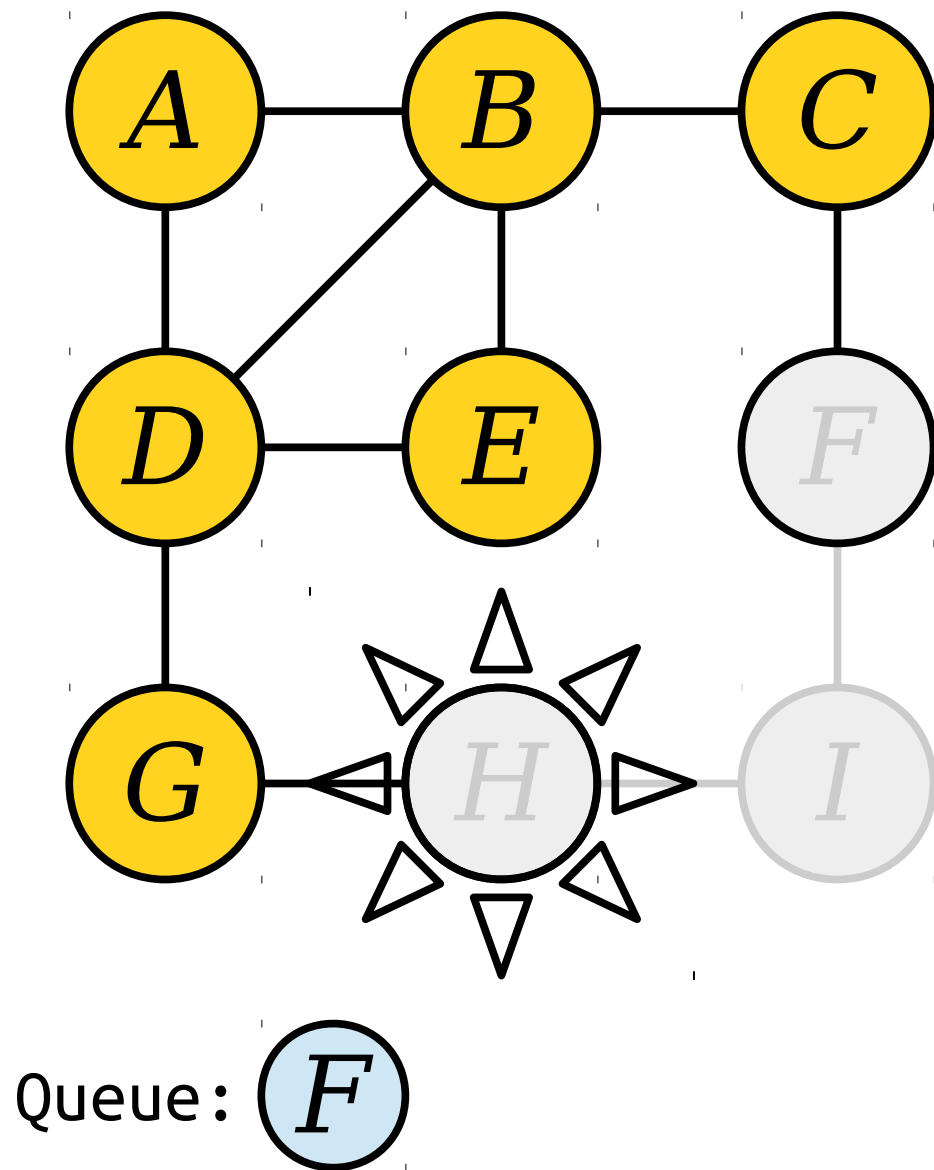
Visit nodes in ascending order of distance from the start node *E*.



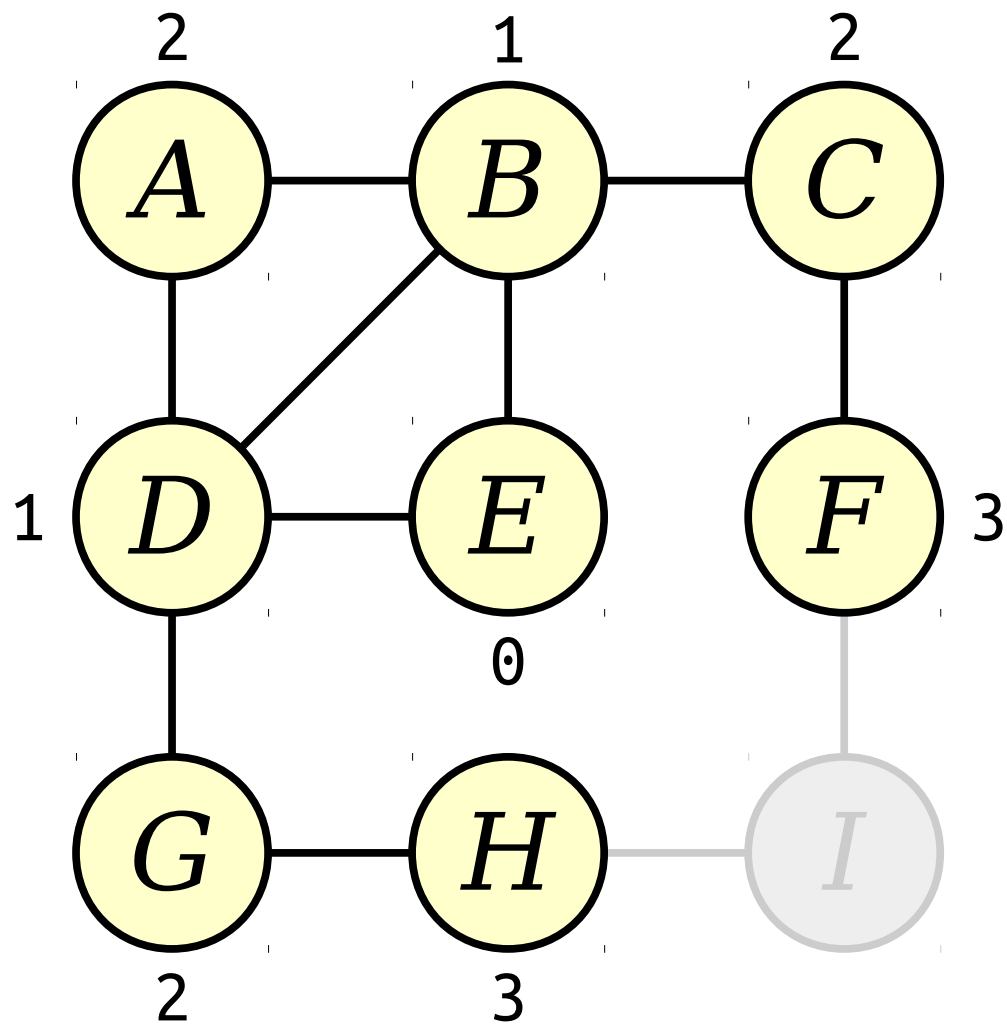
Load newly-discovered nodes into a queue.



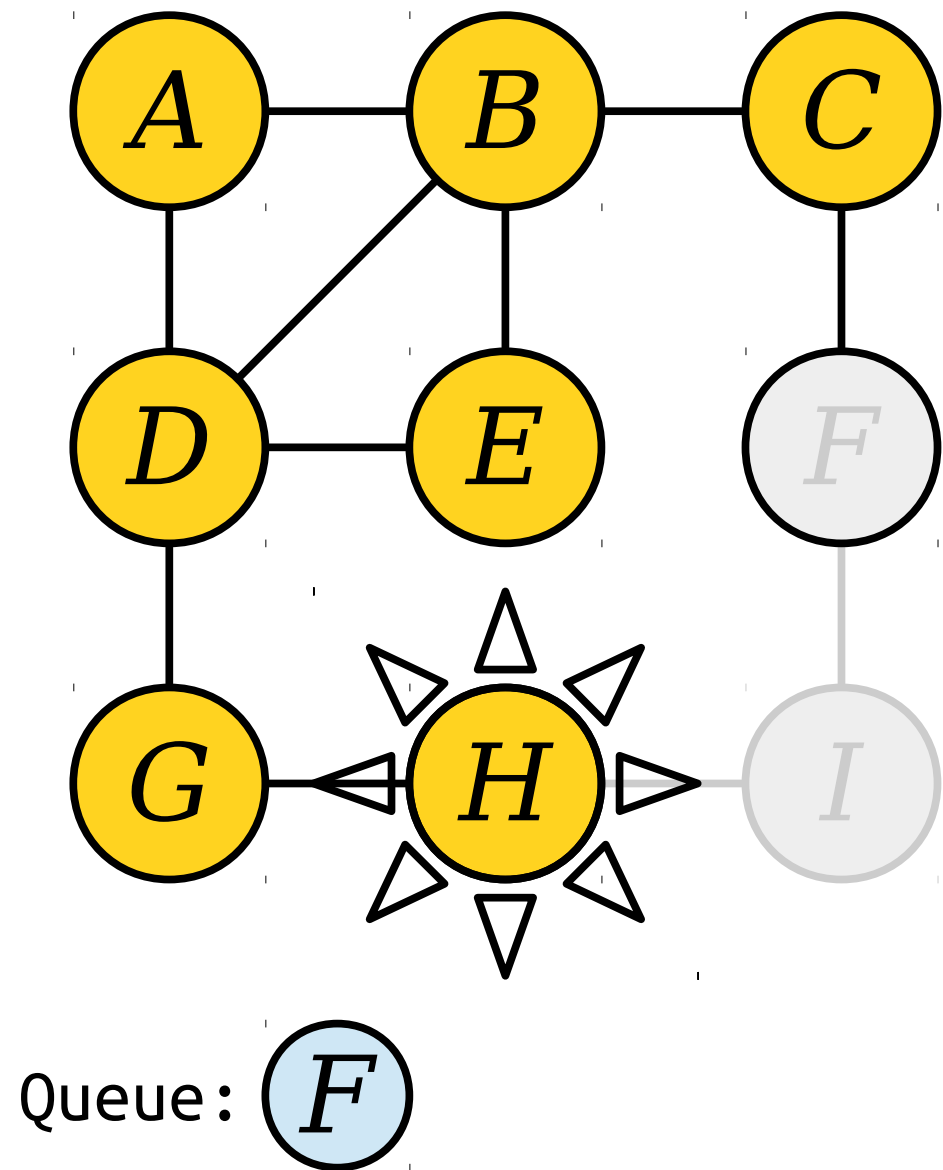
Visit nodes in ascending order of distance from the start node *E*.



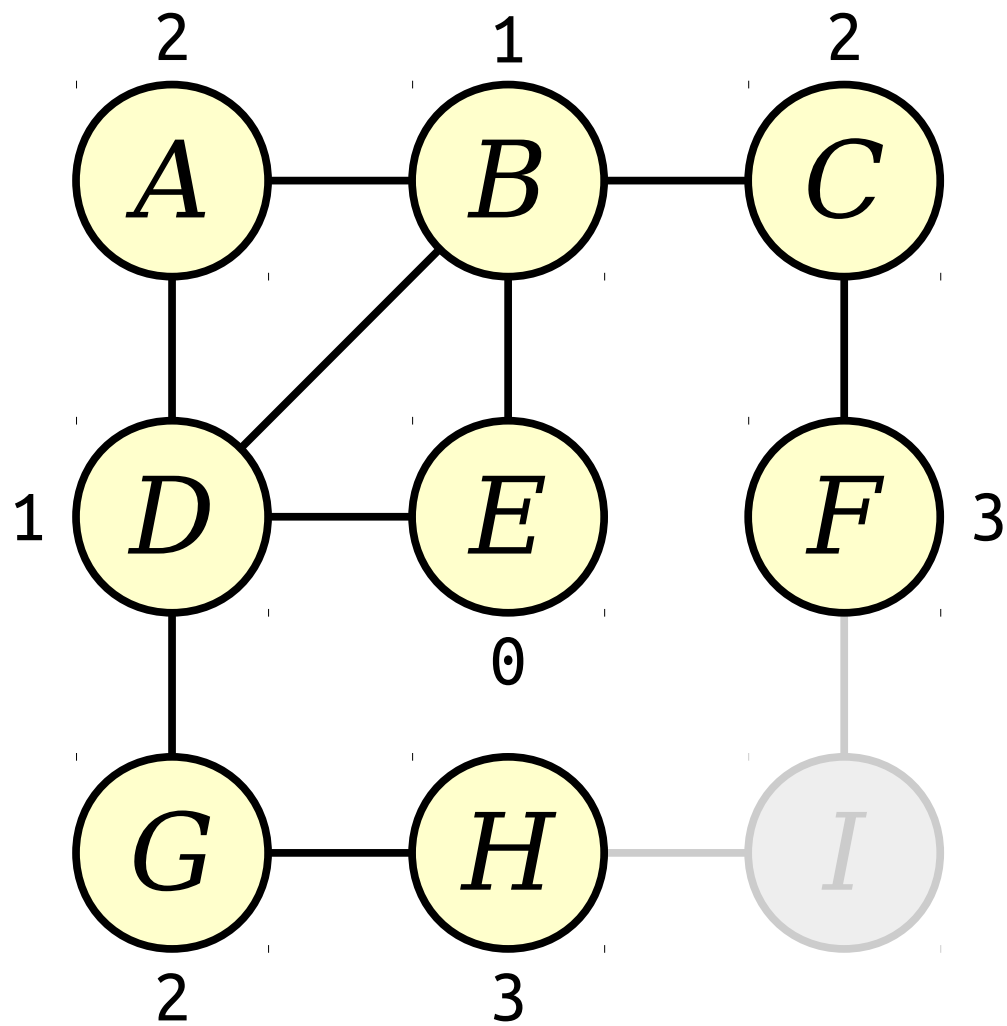
Load newly-discovered nodes into a queue.



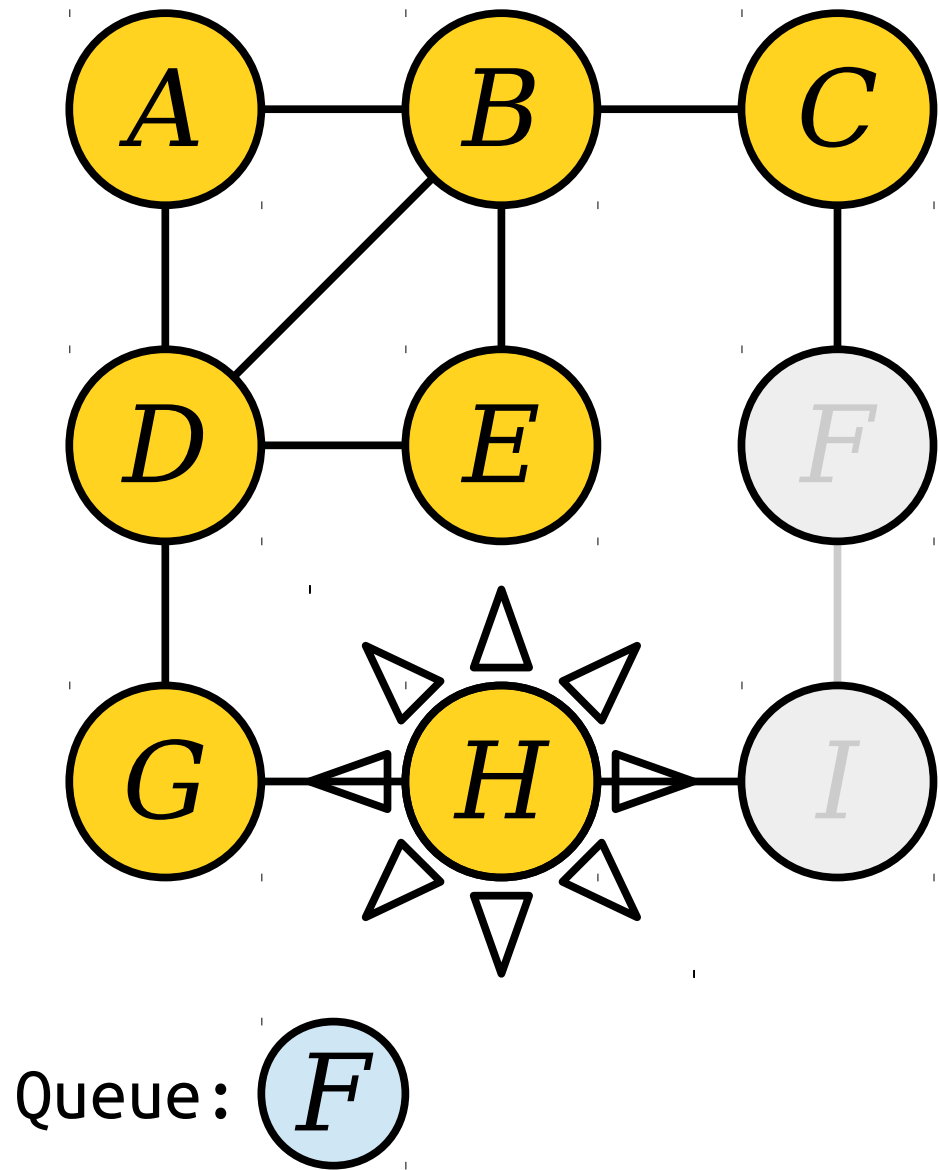
Visit nodes in ascending order of distance from the start node *E*.



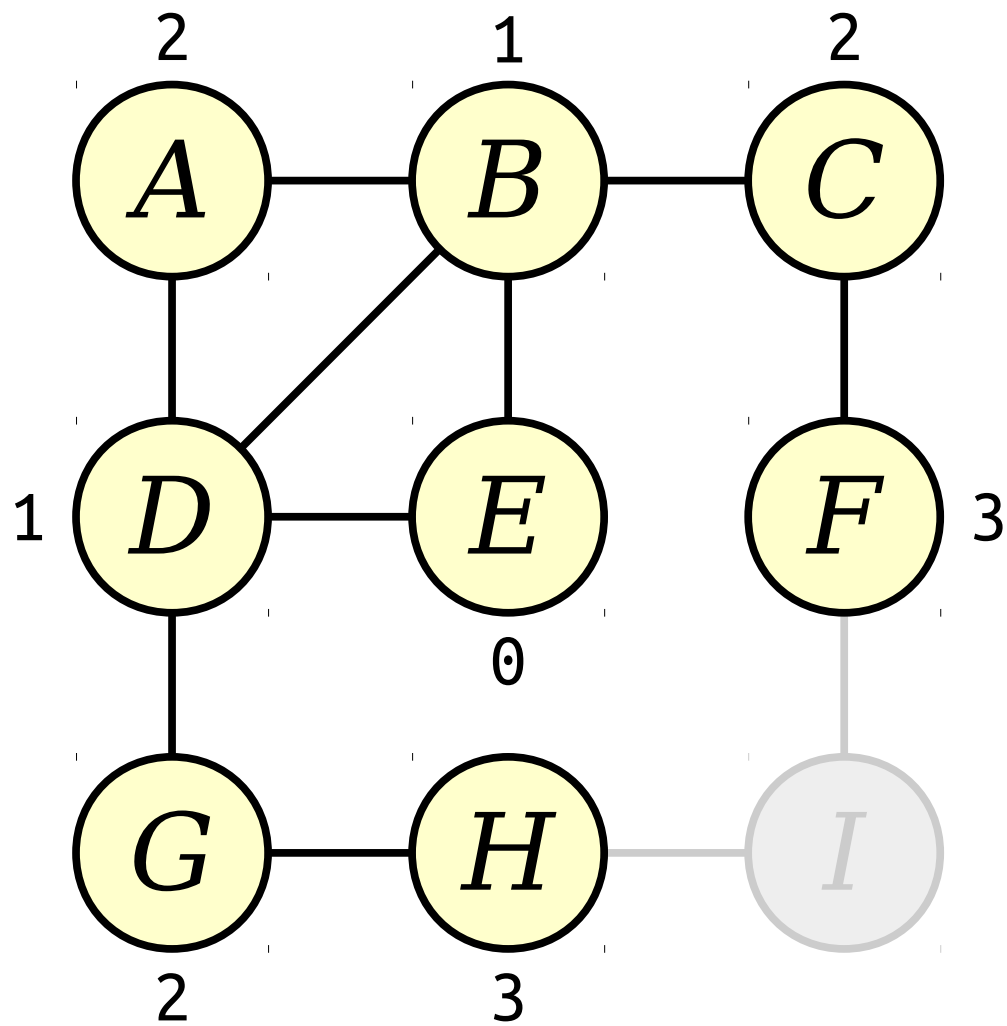
Load newly-discovered nodes into a queue.



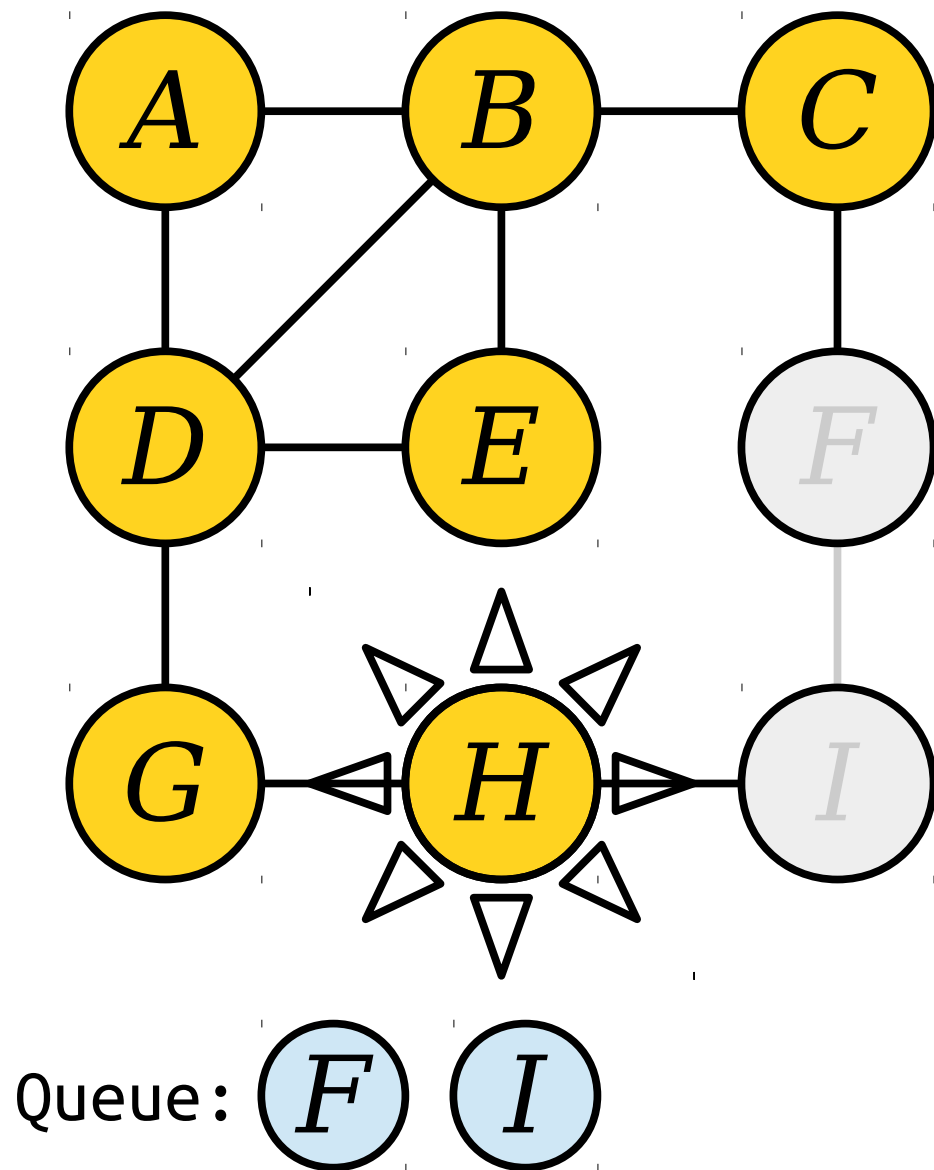
Visit nodes in ascending order of distance from the start node *E*.



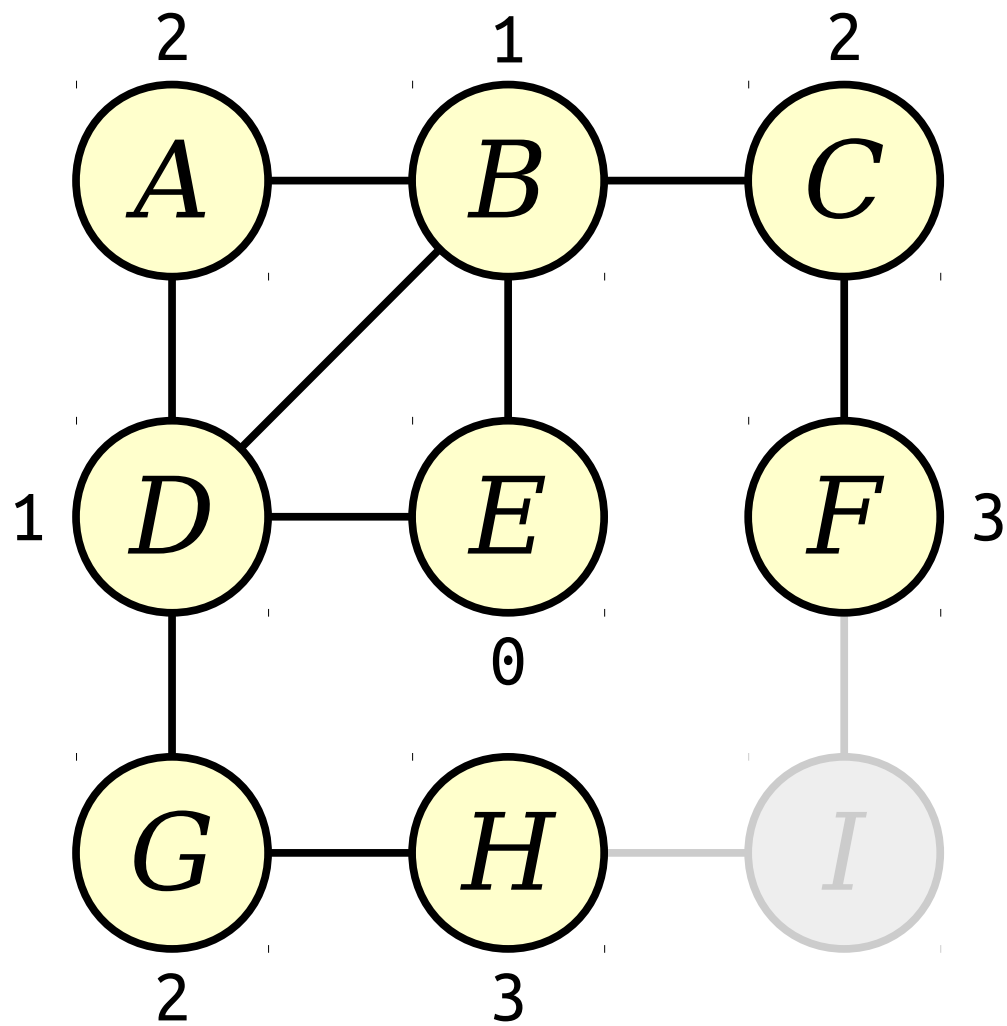
Load newly-discovered nodes into a queue.



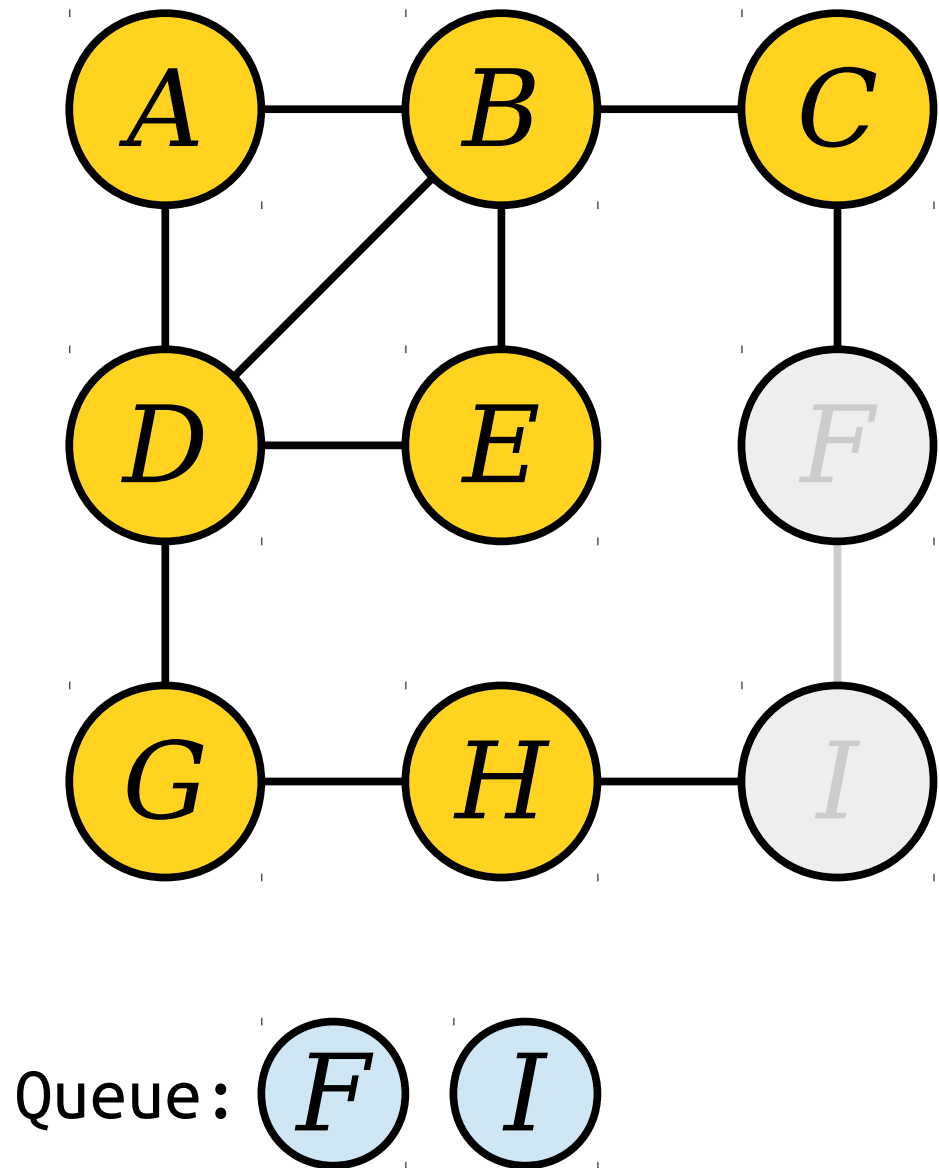
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

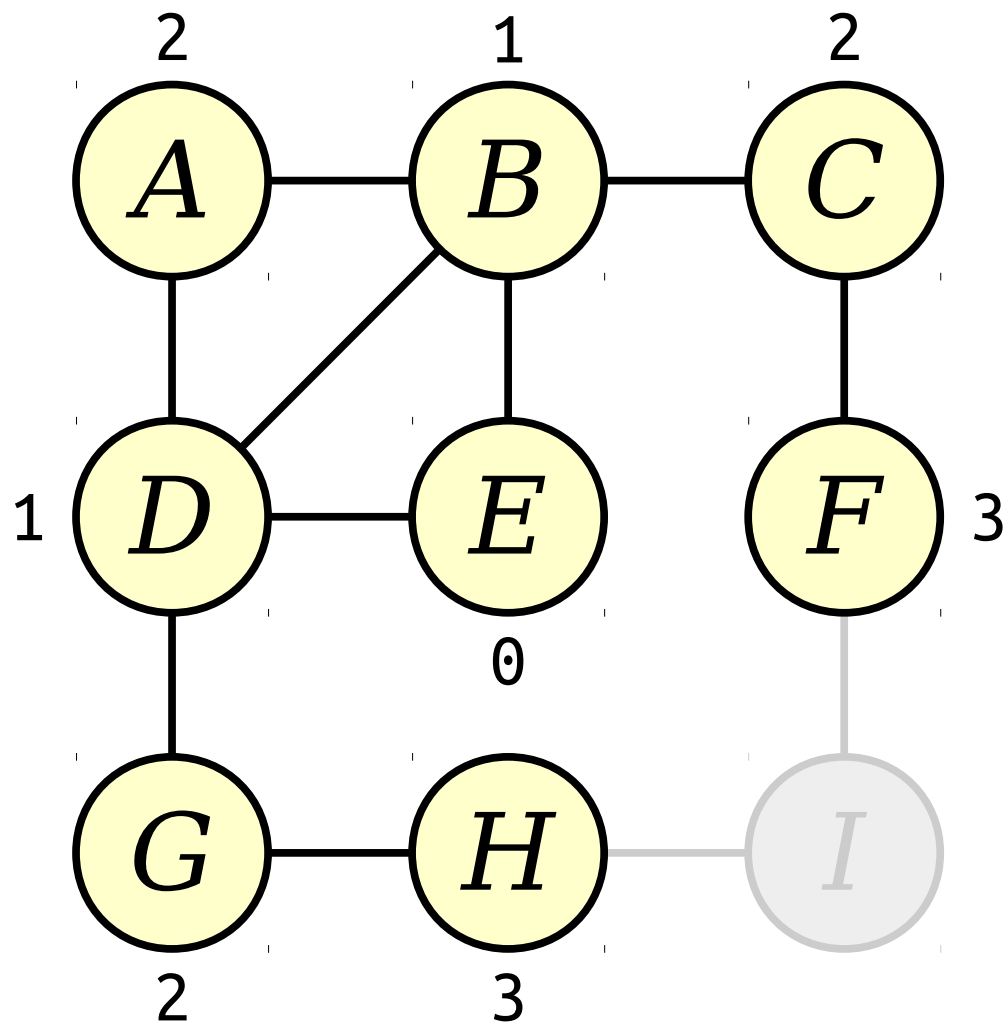


Visit nodes in ascending order of distance from the start node *E*.

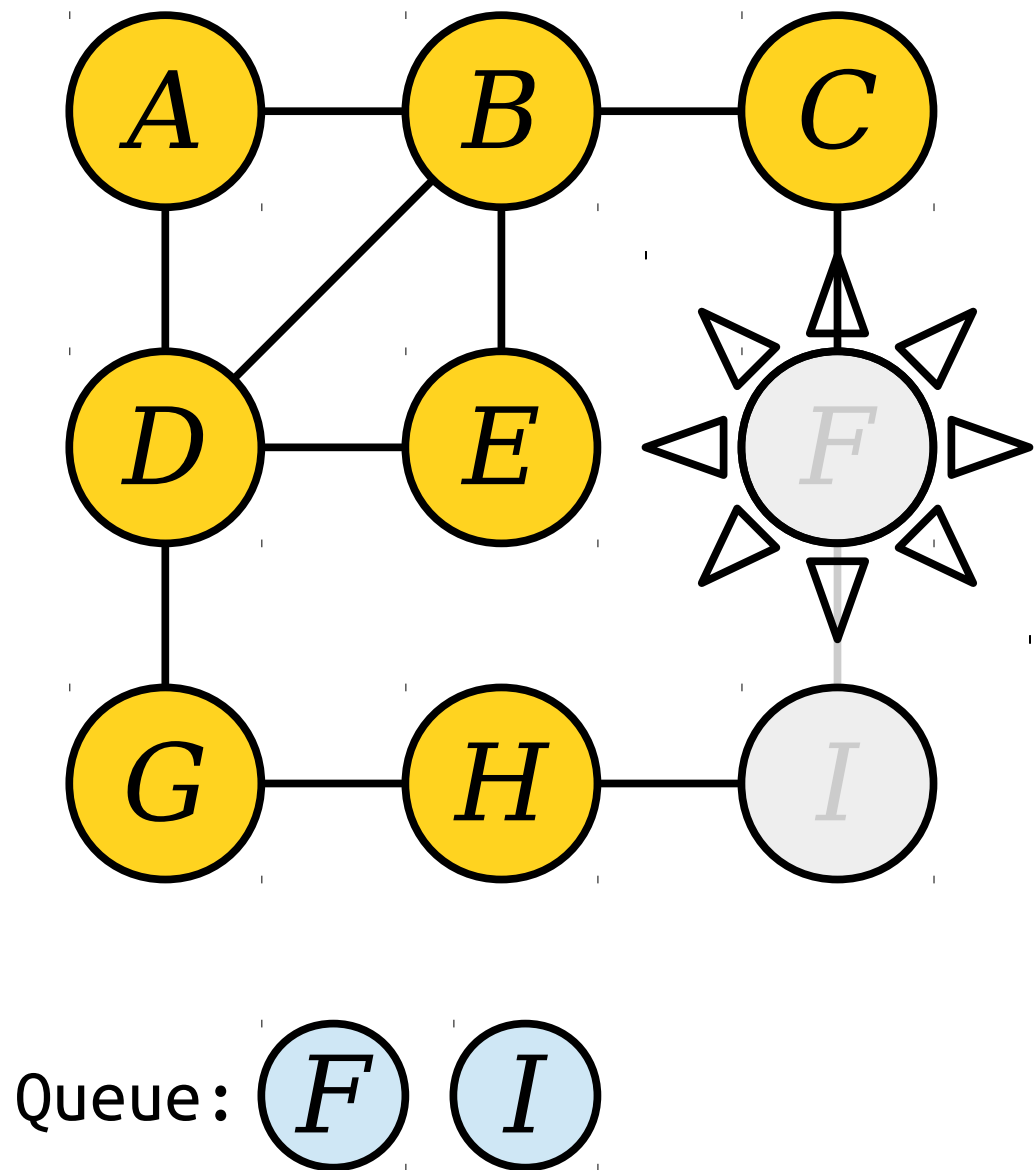


Load newly-discovered nodes into a queue.

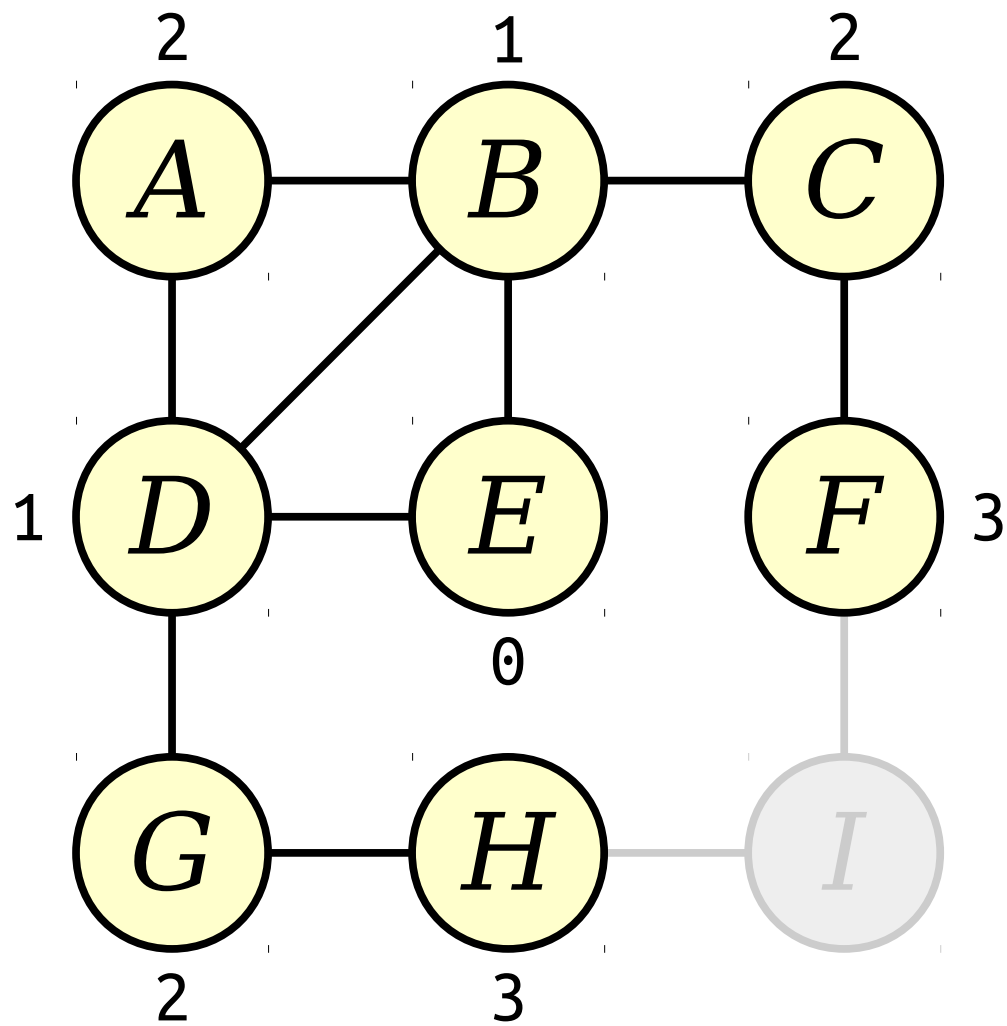




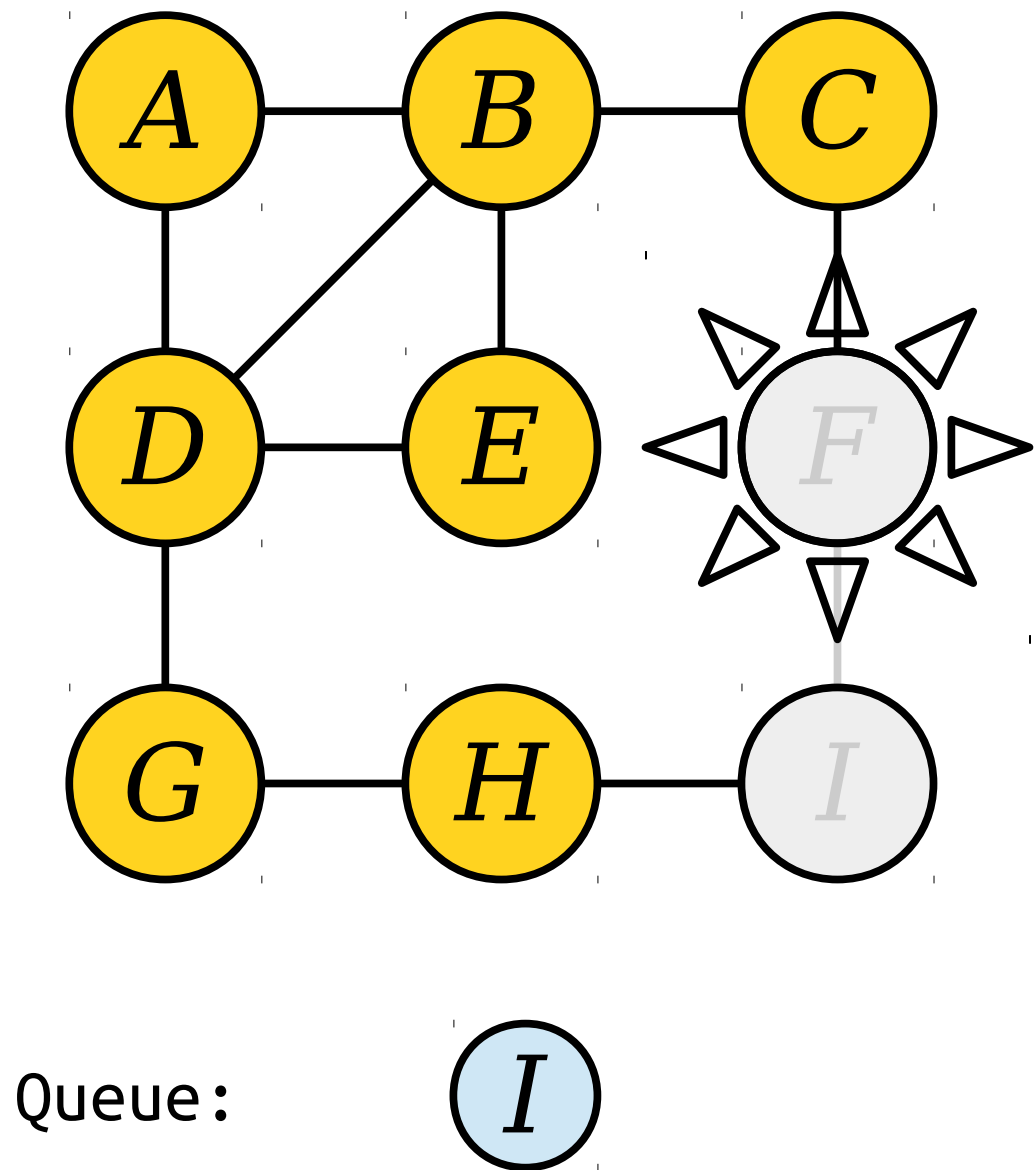
Visit nodes in ascending order of distance from the start node *E*.



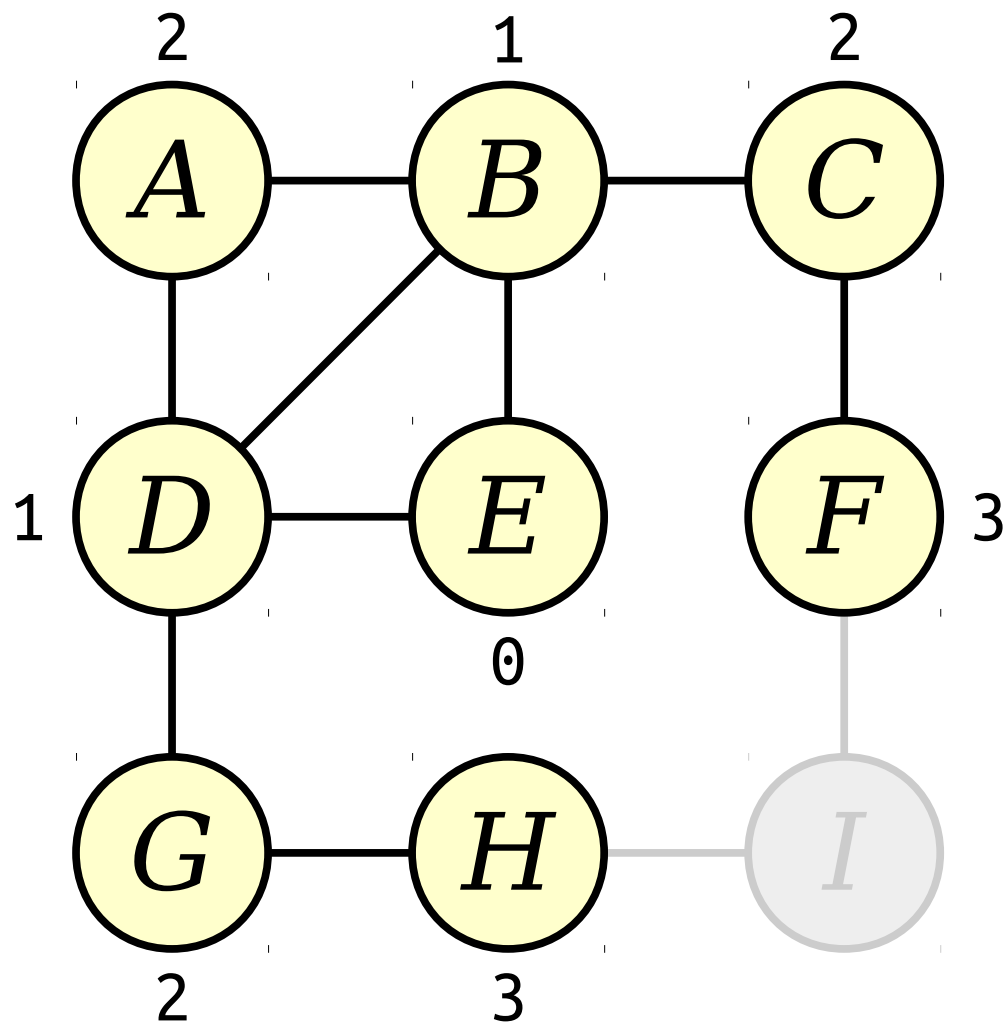
Load newly-discovered nodes into a queue.



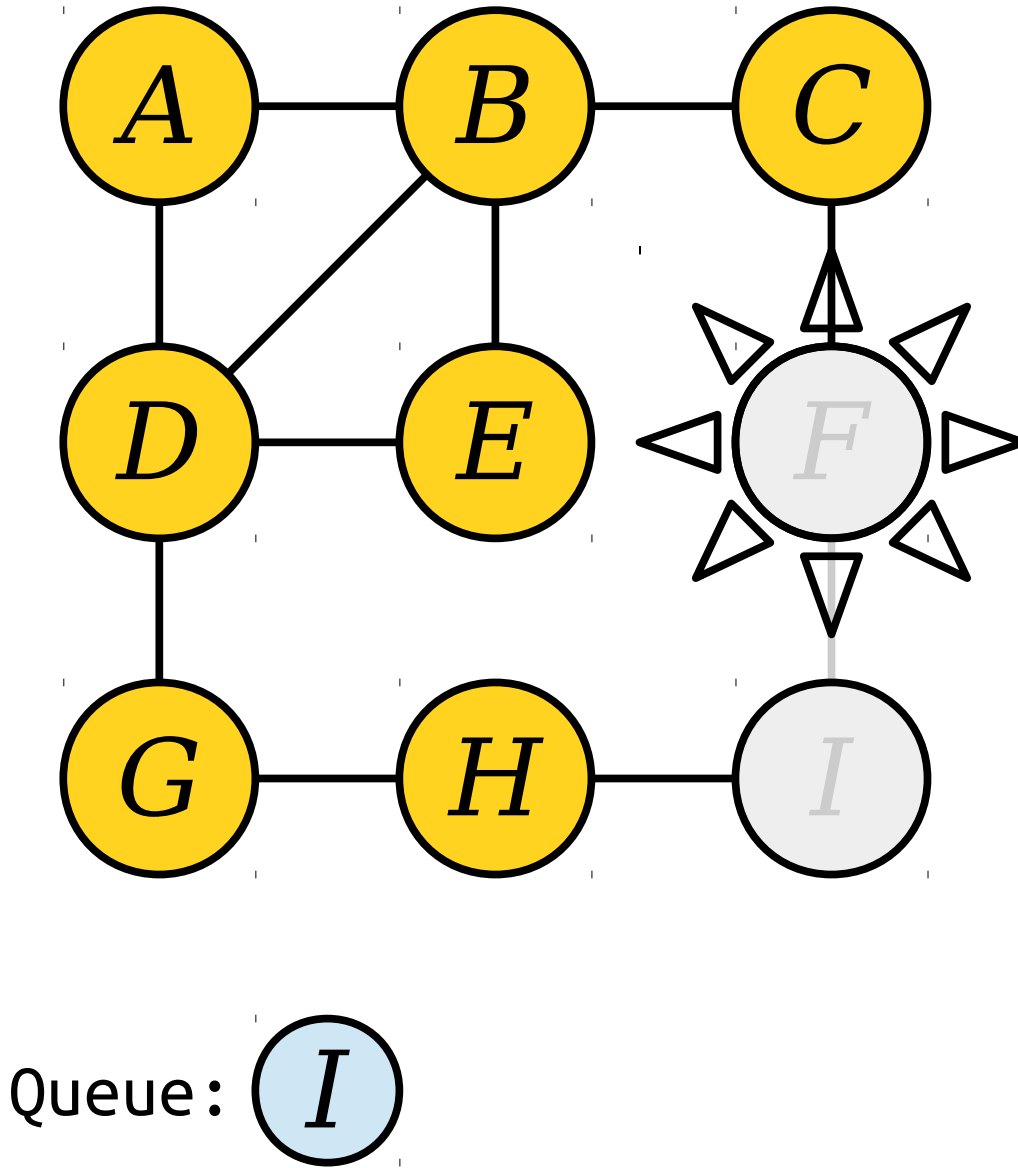
Visit nodes in ascending order of distance from the start node *E*.



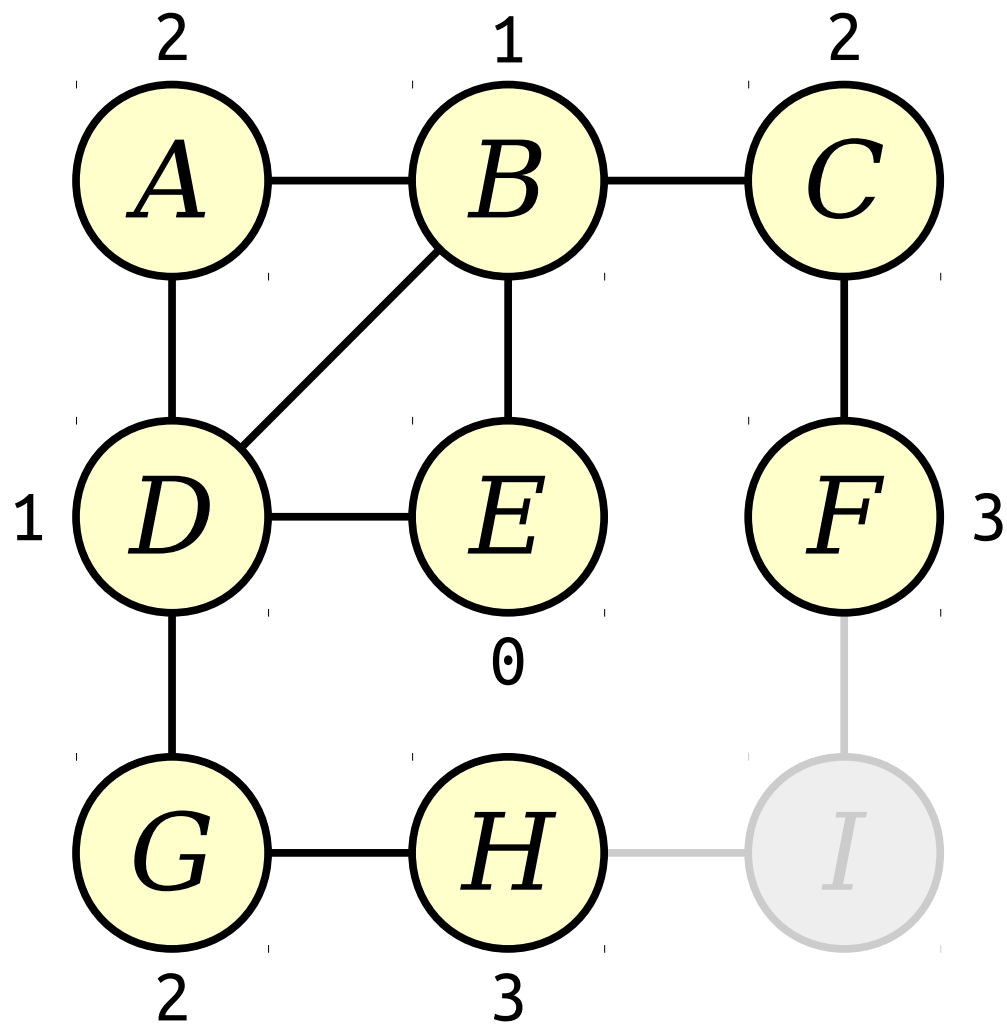
Load newly-discovered nodes into a queue.



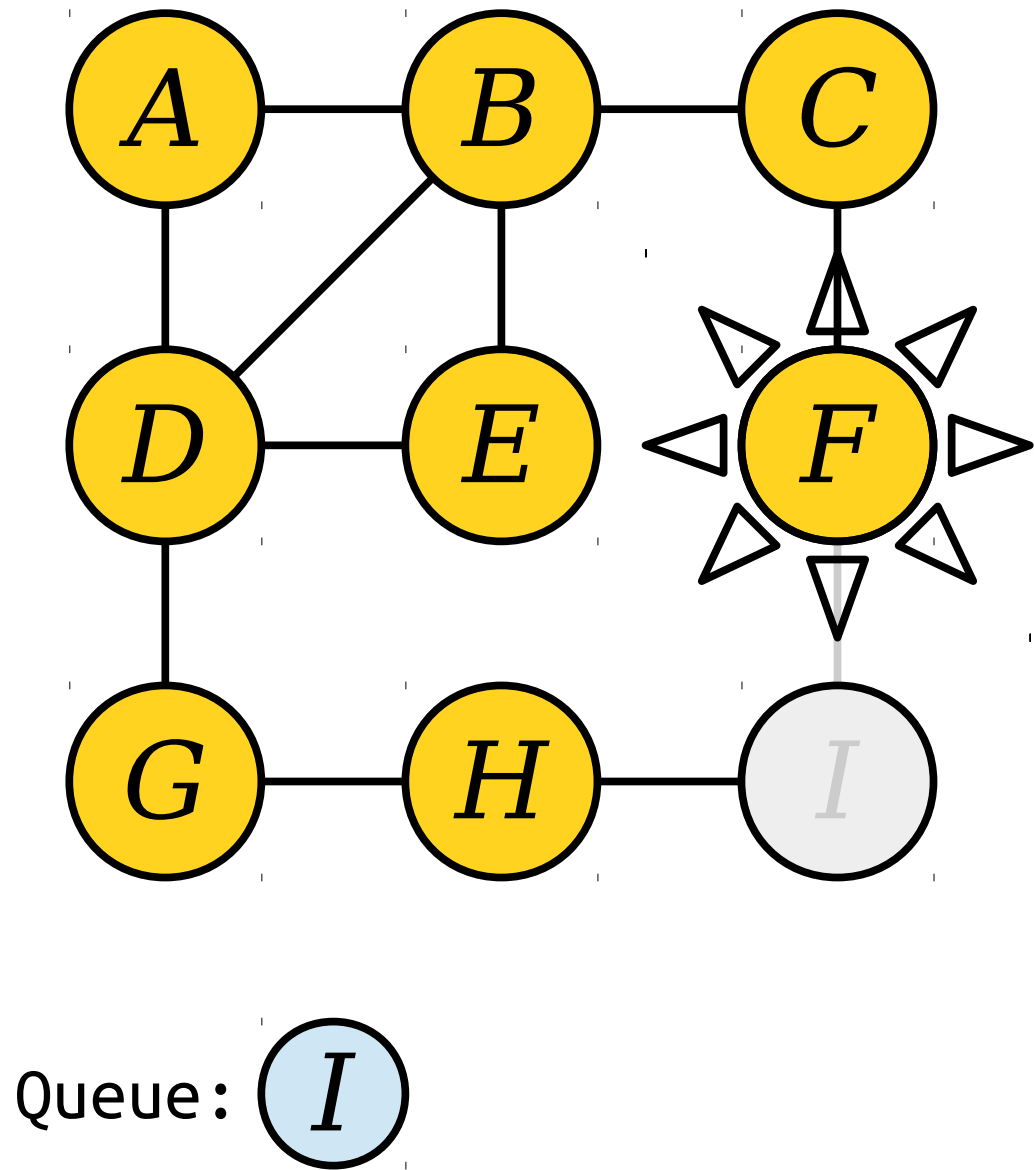
Visit nodes in ascending order of distance from the start node *E*.



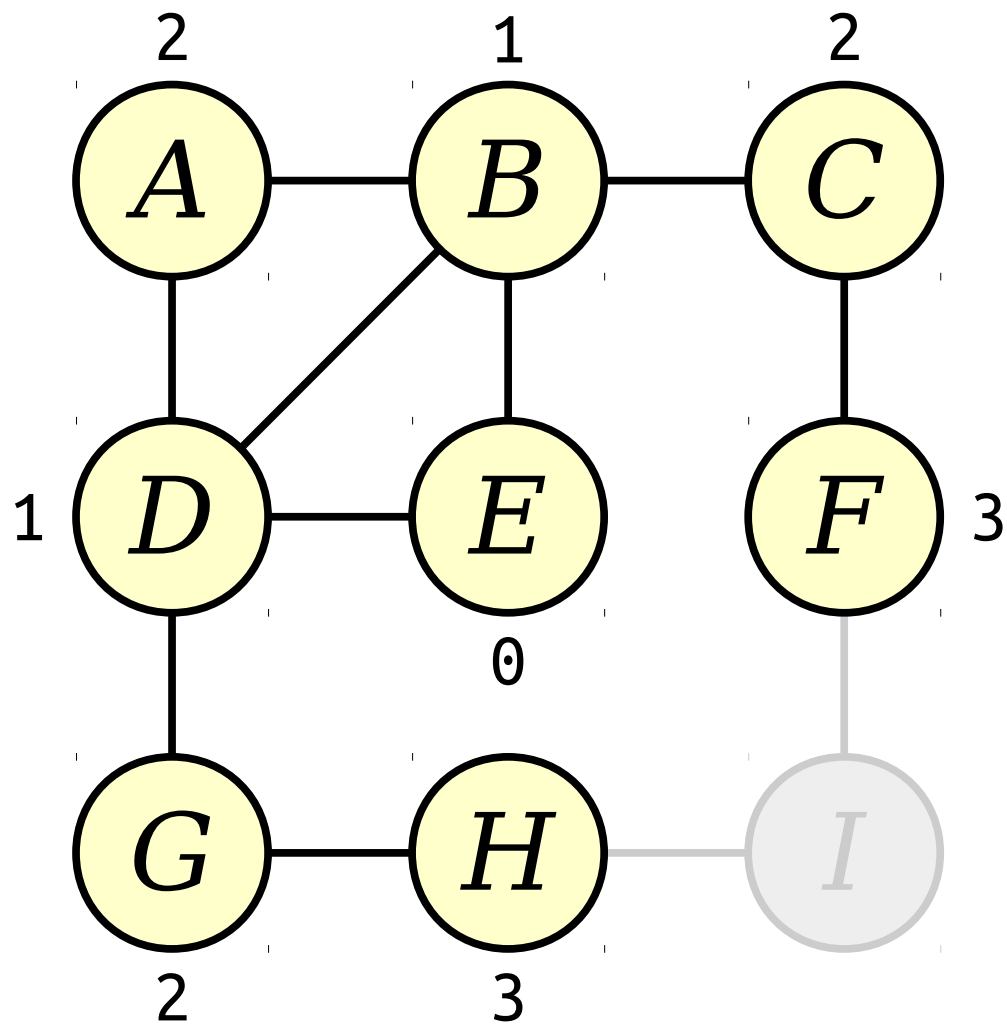
Load newly-discovered nodes into a queue.



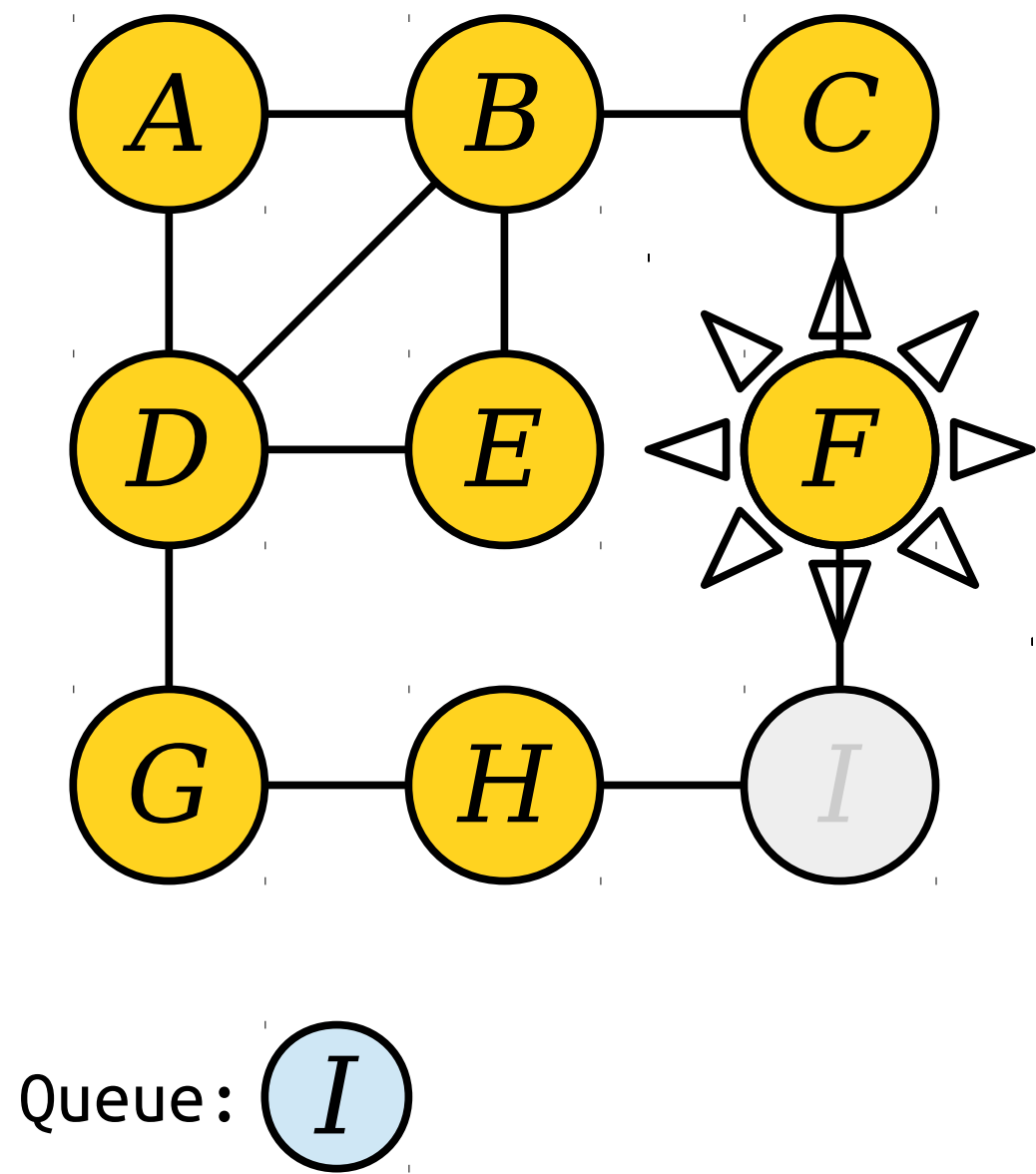
Visit nodes in ascending order of distance from the start node *E*.



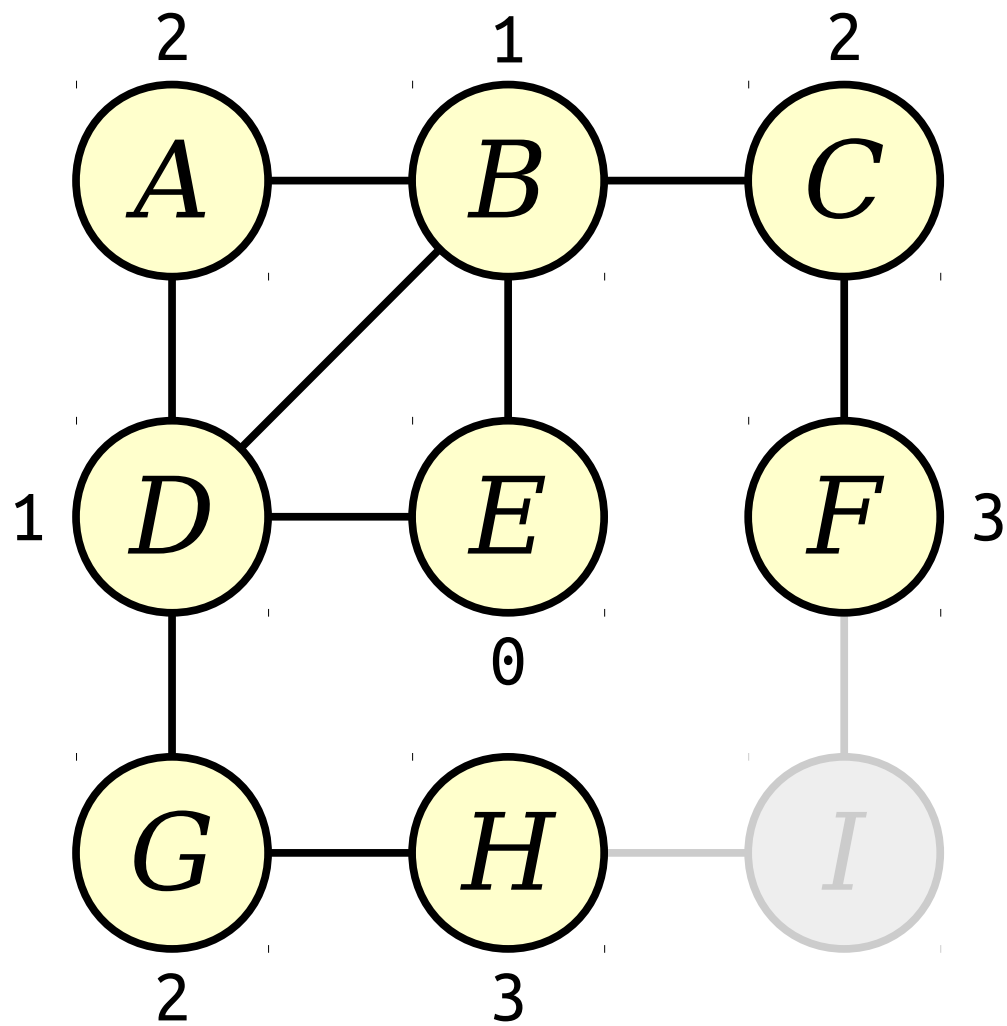
Load newly-discovered nodes into a queue.



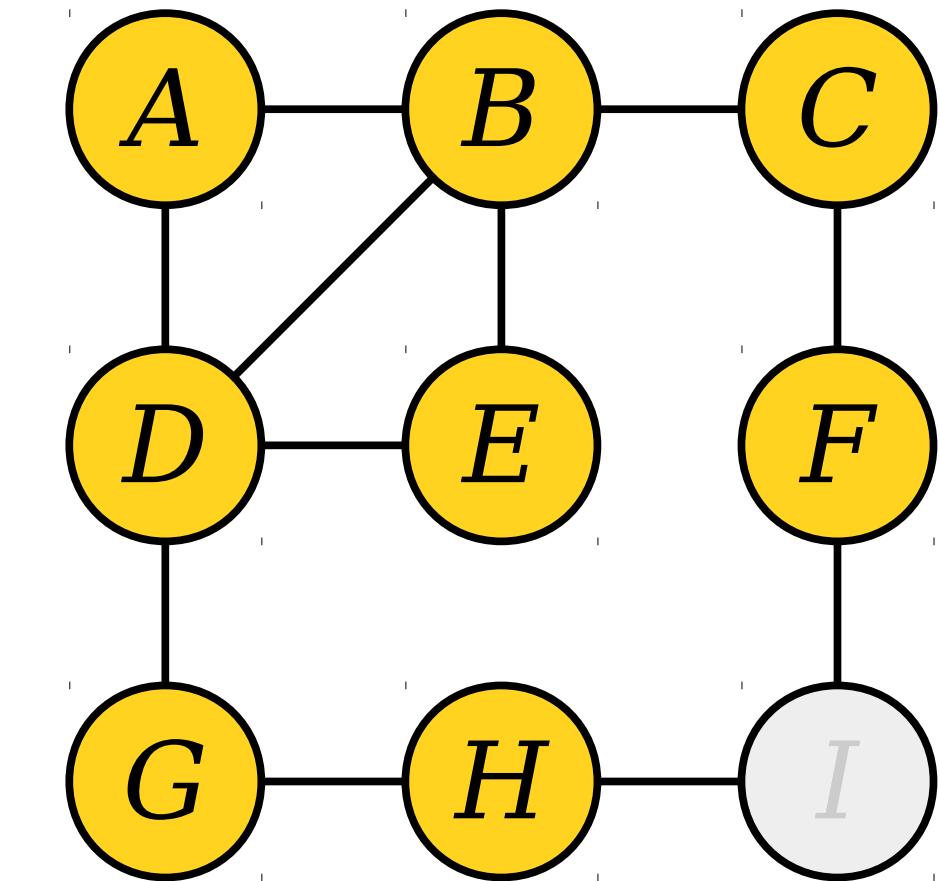
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.

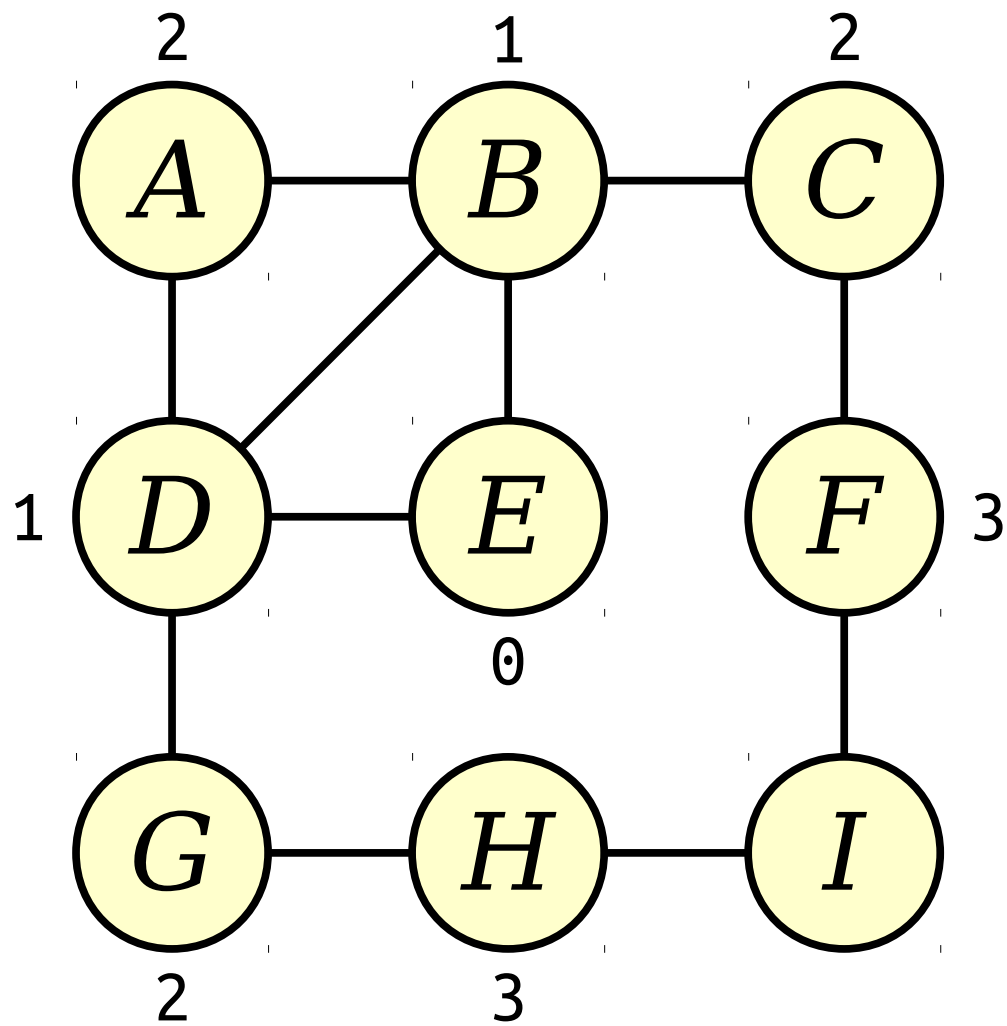


Visit nodes in ascending order of distance from the start node *E*.

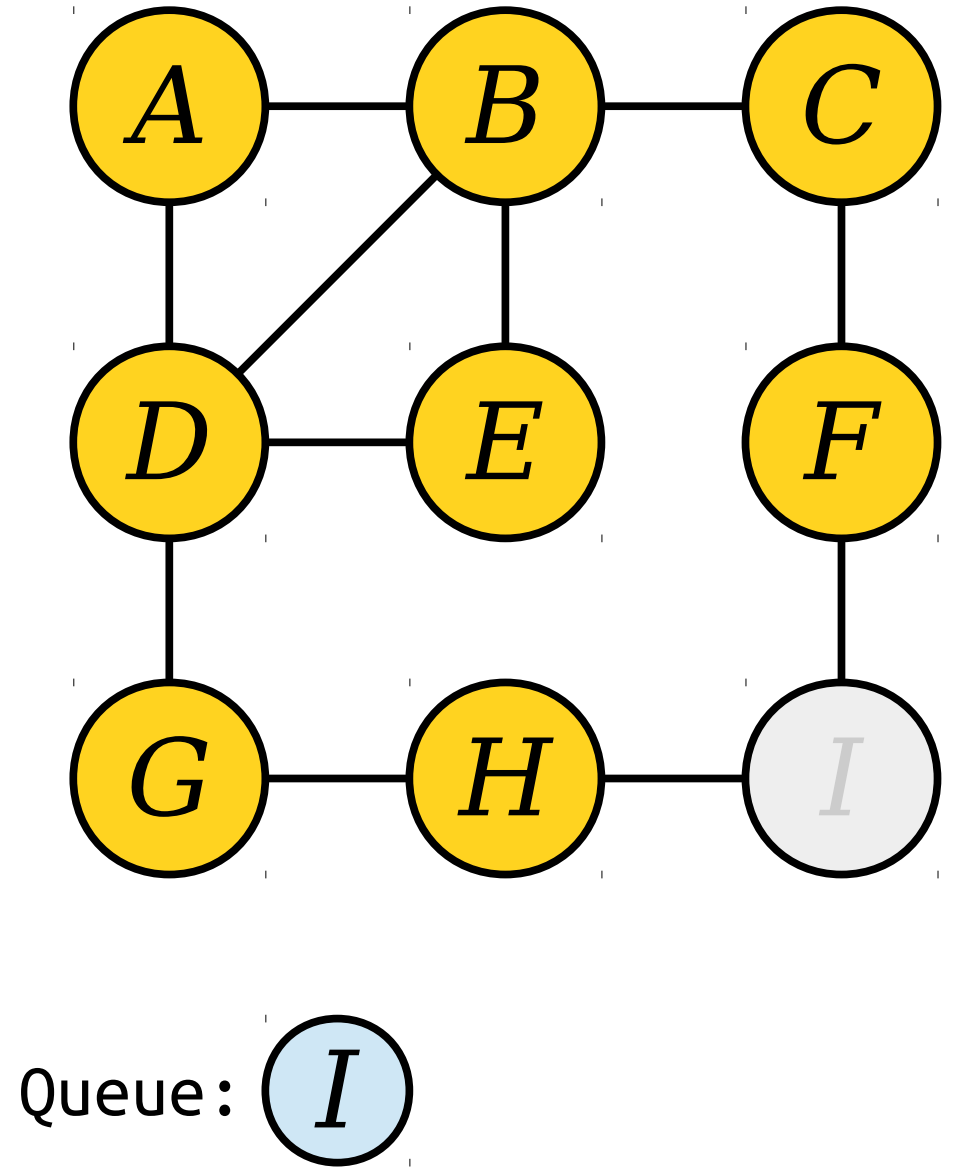


Queue: *I*

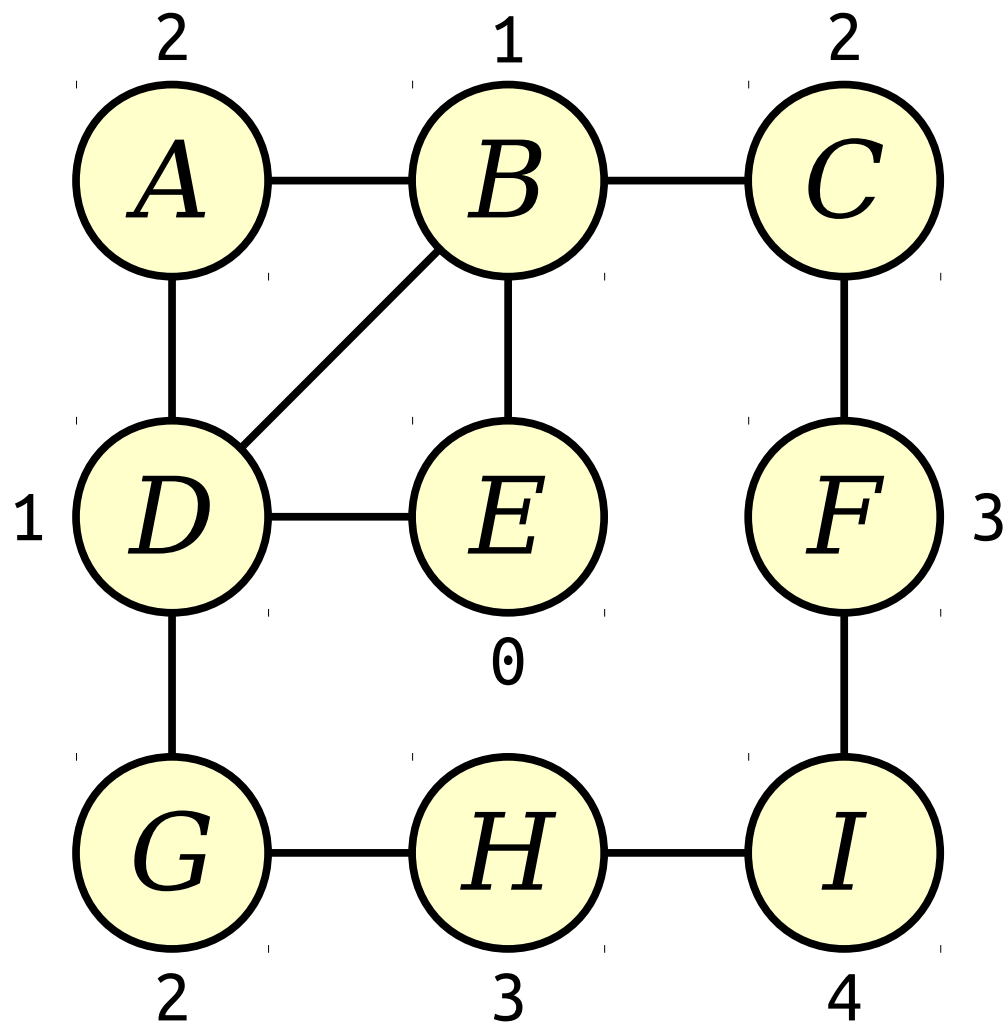
Load newly-discovered nodes into a queue.



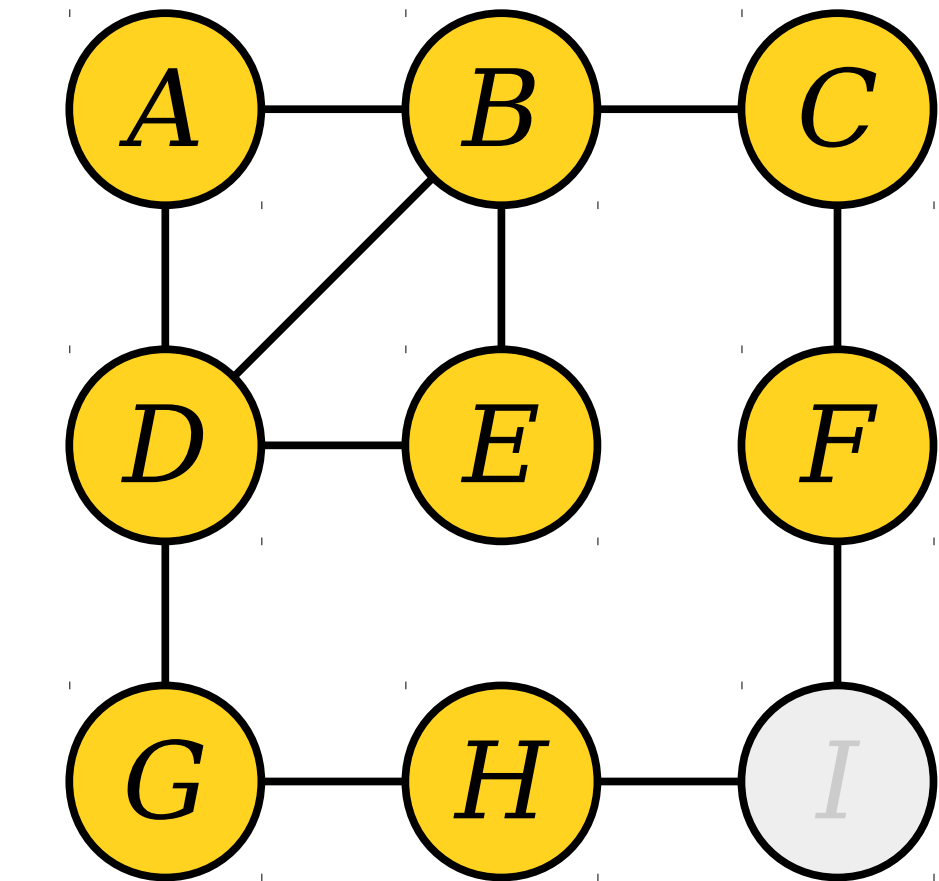
Visit nodes in ascending order of distance from the start node *E*.



Load newly-discovered nodes into a queue.



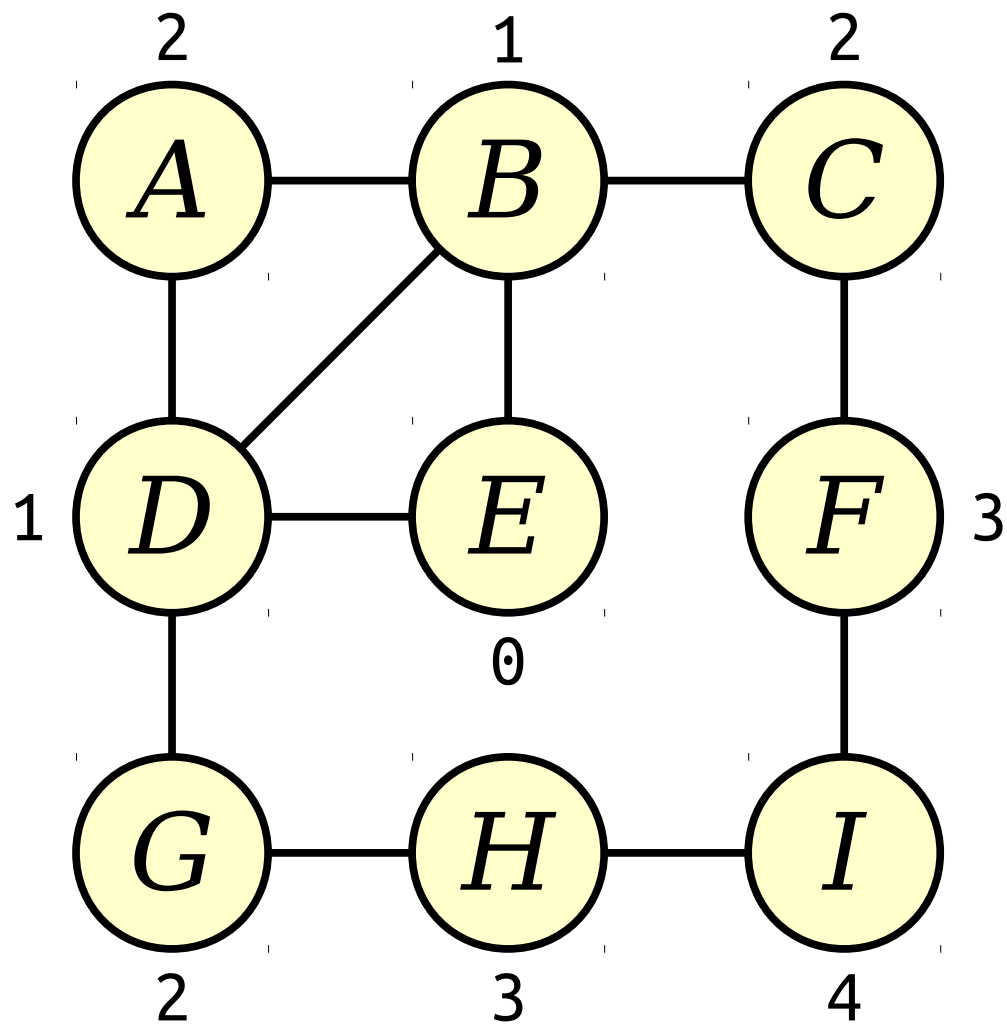
Visit nodes in ascending order of distance from the start node *E*.



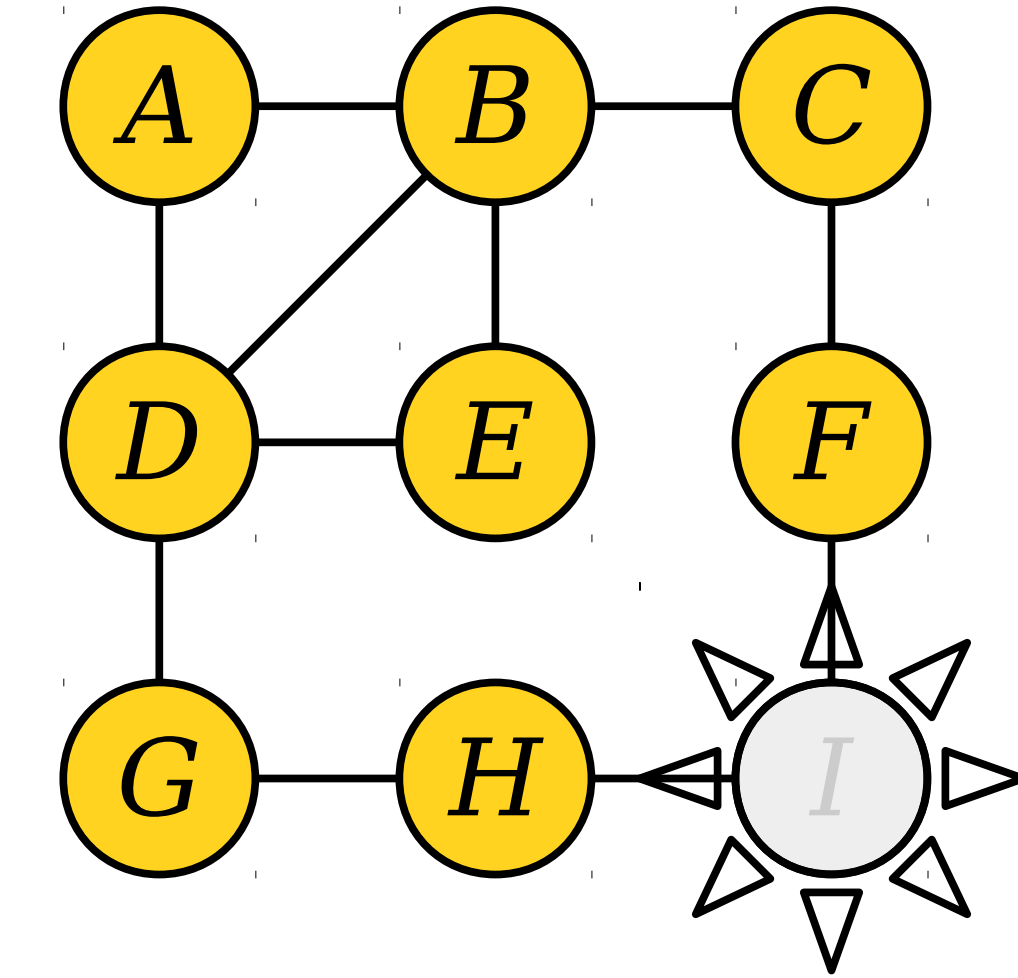
Queue: *I*

Load newly-discovered nodes into a queue.



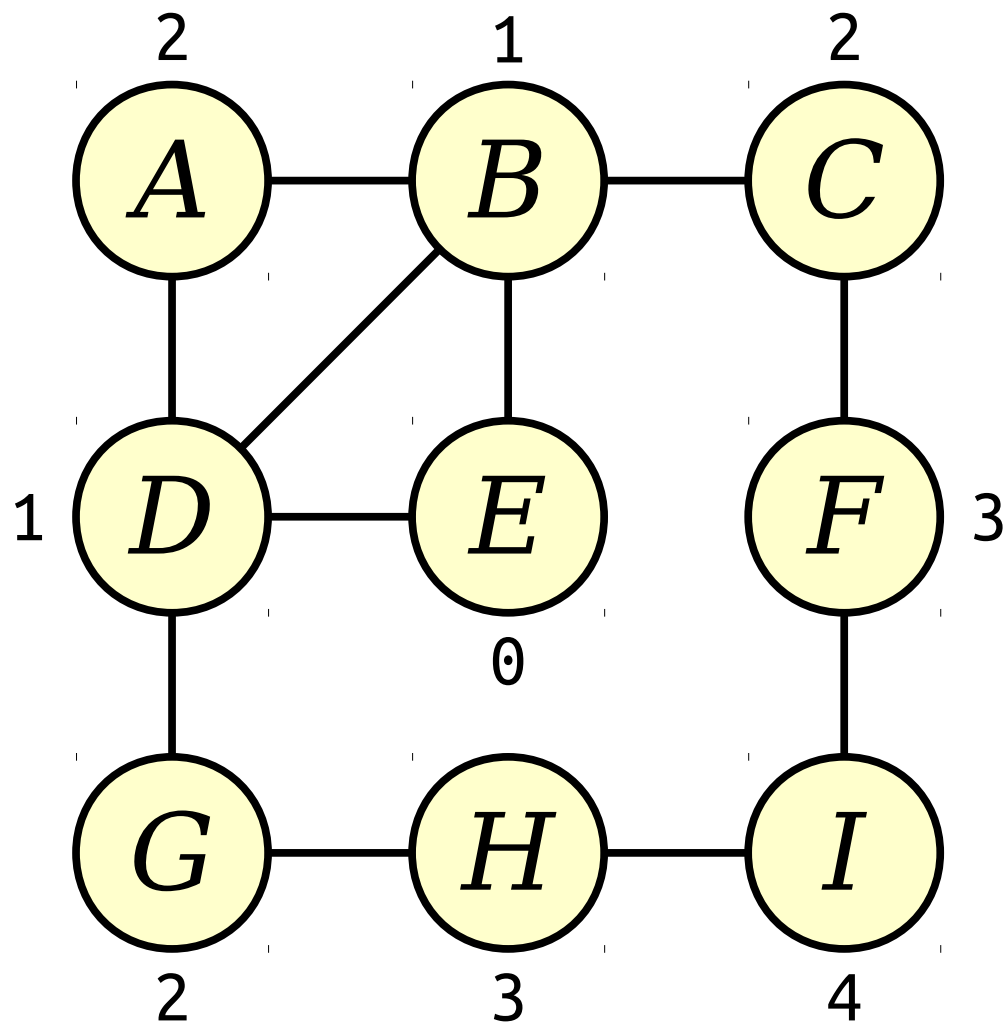


Visit nodes in ascending order of distance from the start node *E*.

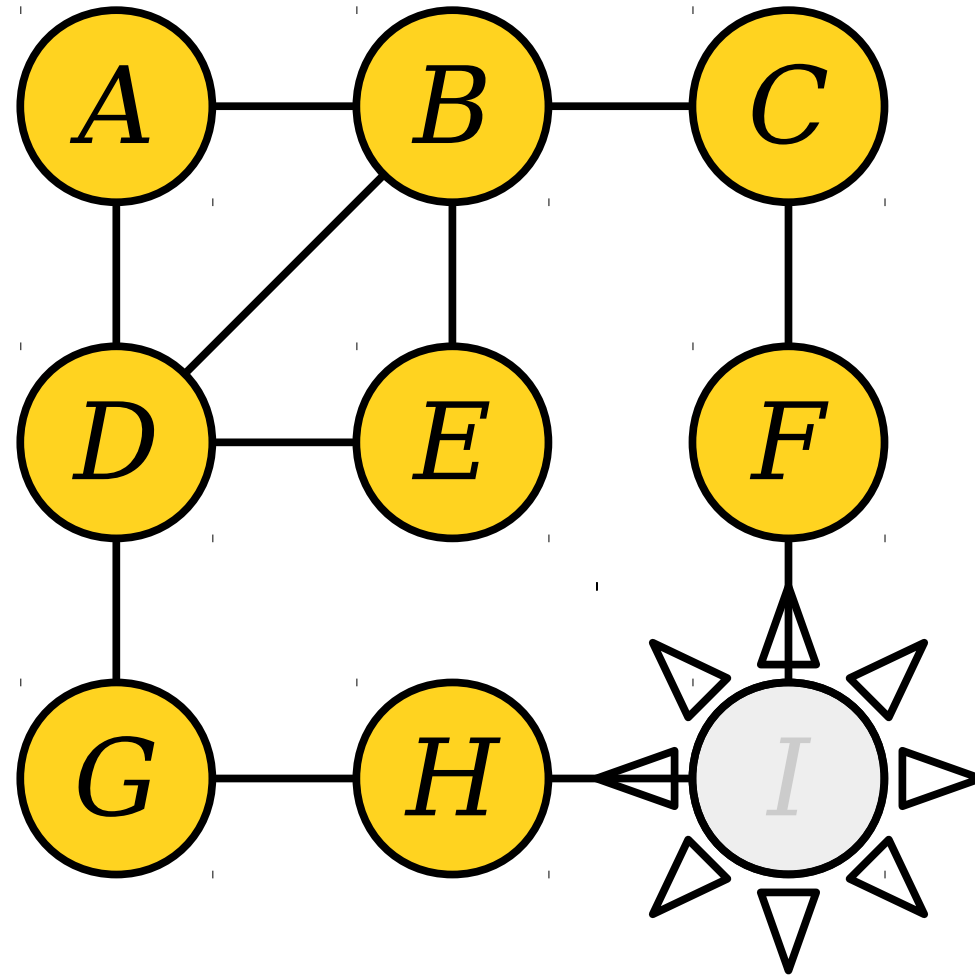


Queue: *I*

Load newly-discovered nodes into a queue.

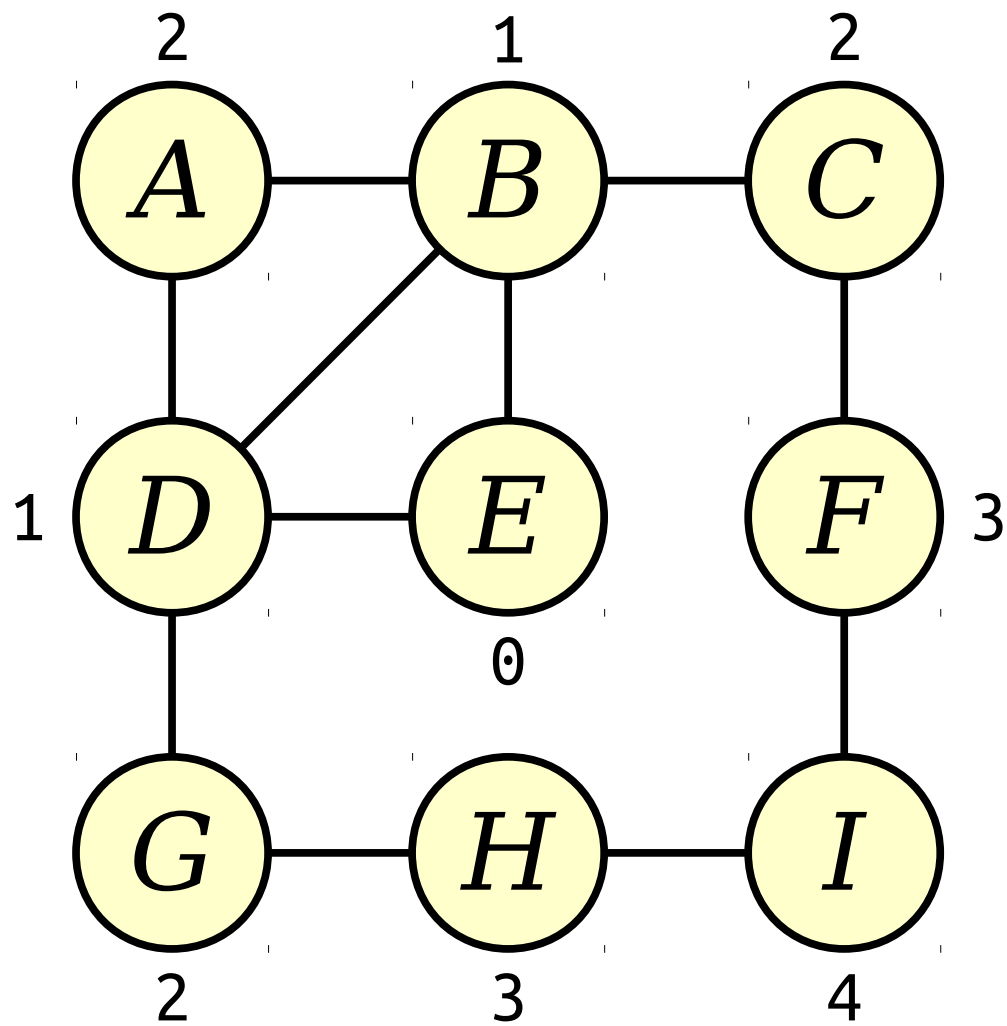


Visit nodes in ascending order of distance from the start node *E*.

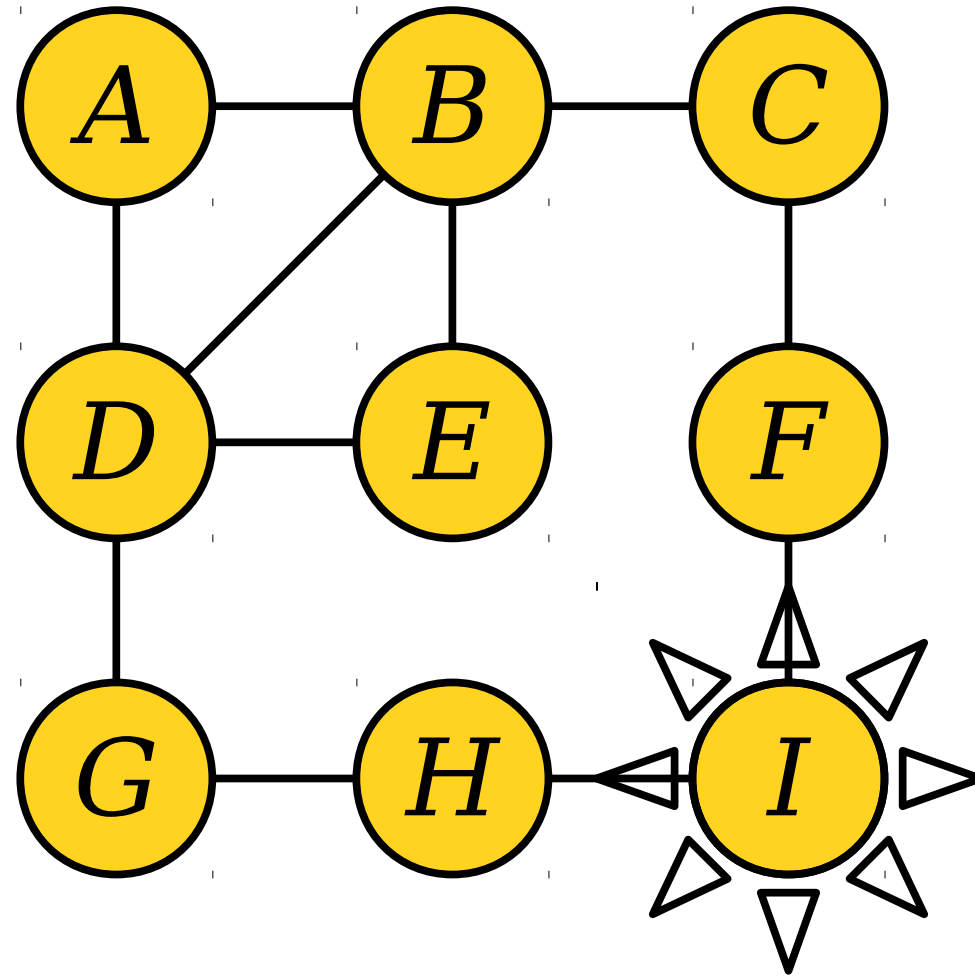


Queue:

Load newly-discovered nodes into a queue.

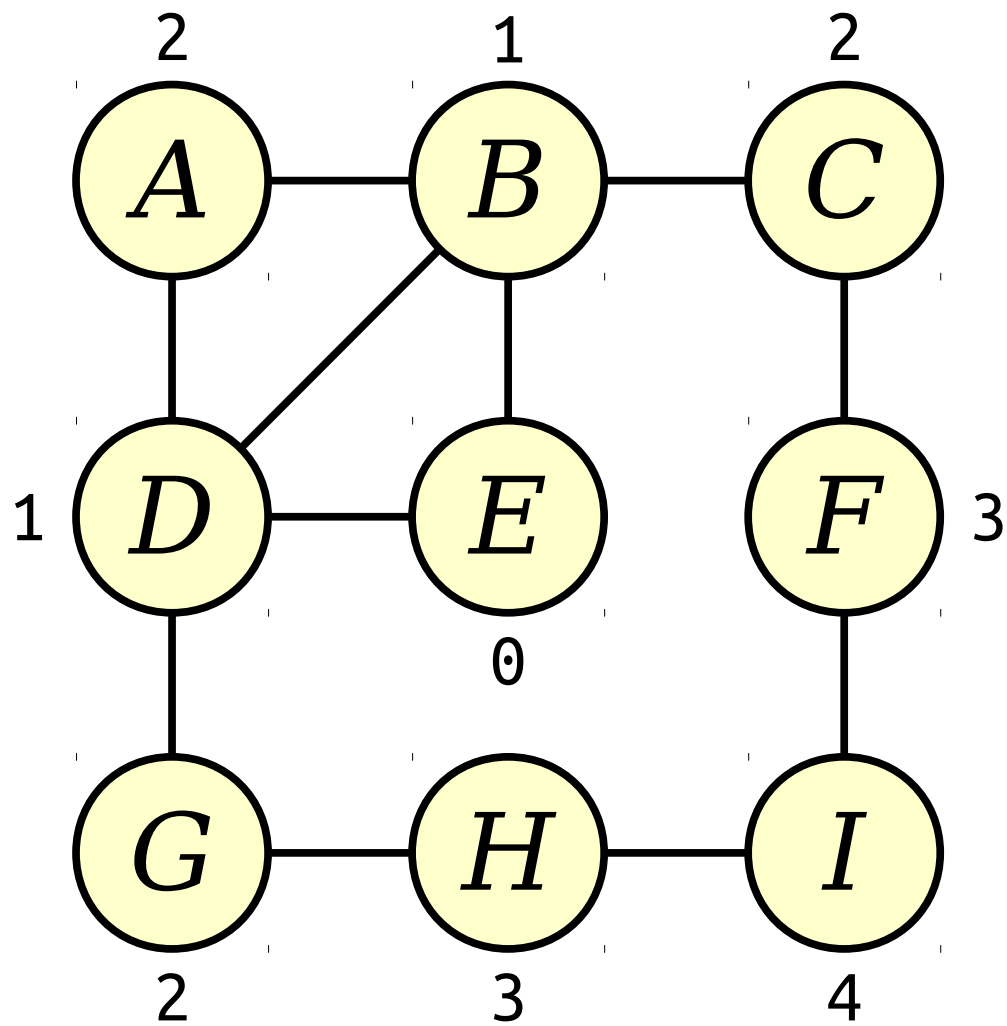


Visit nodes in ascending order of distance from the start node *E*.

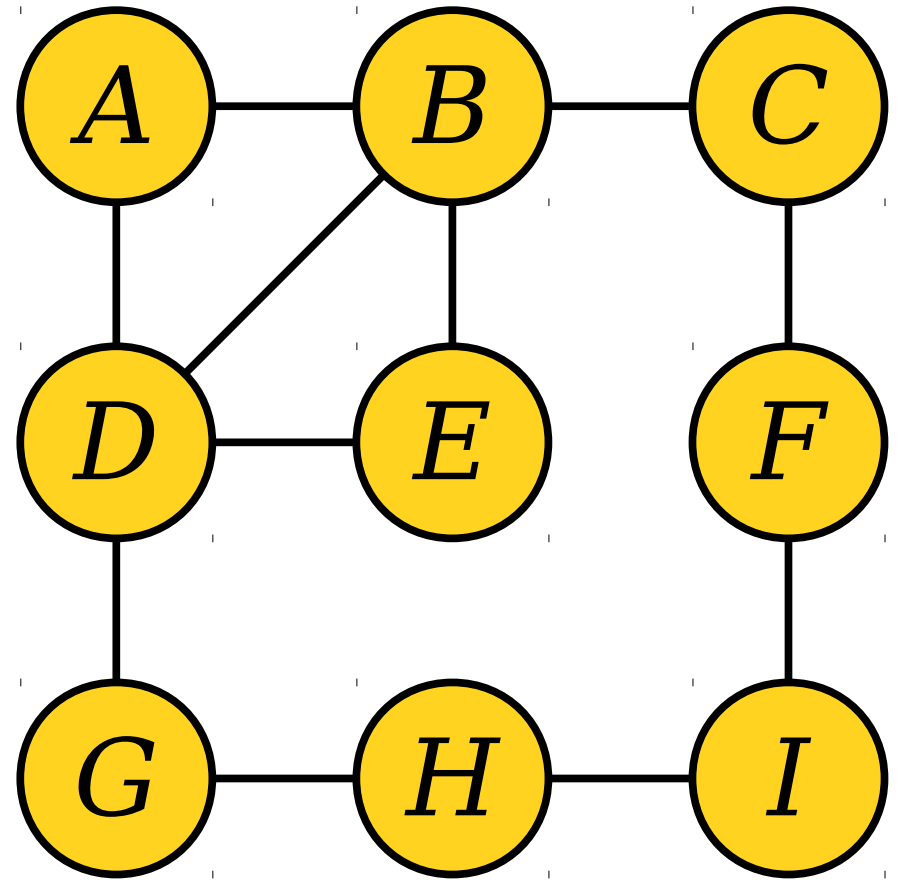


Queue:

Load newly-discovered nodes into a queue.



Visit nodes in ascending order of distance from the start node *E*.



Queue:

Load newly-discovered nodes into a queue.

# Breadth-First Search

- The Queue-based search strategy we just saw is called ***breadth-first search*** (or just ***BFS*** for short).
- In pseudocode:

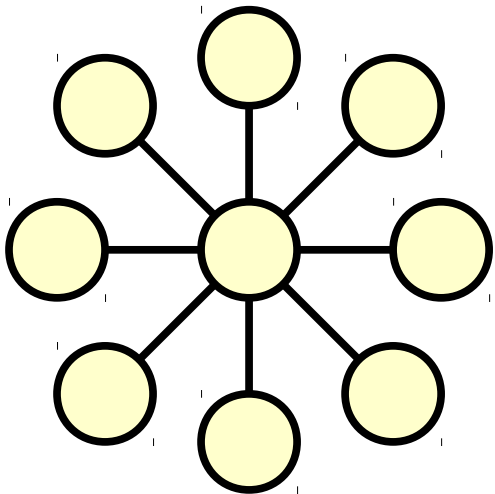
```
bfs-from(node v) {  
    make a queue of nodes, initially seeded with v.  
    while the queue isn't empty:  
        dequeue a node curr.  
        process the node curr.  
        for each node adjacent to curr:  
            if that node has never been enqueued:  
                enqueue that node.  
}
```

# BFS Efficiency

```
bfs-from(node v) {  
    make a queue of nodes, initially seeded with v.  
    while the queue isn't empty:  
        dequeue a node curr.  
        process the node curr.  
        for each node adjacent to curr:  
            if that node has never been enqueued:  
                enqueue that node.  
}
```

# BFS Efficiency

```
bfs-from(node v) {  
  make a queue of nodes, initially seeded with v.  
  while the queue isn't empty:  
    dequeue a node curr.  
    process the node curr.  
    for each node adjacent to curr:  
      if that node has never been enqueued:  
        enqueue that node.  
}
```

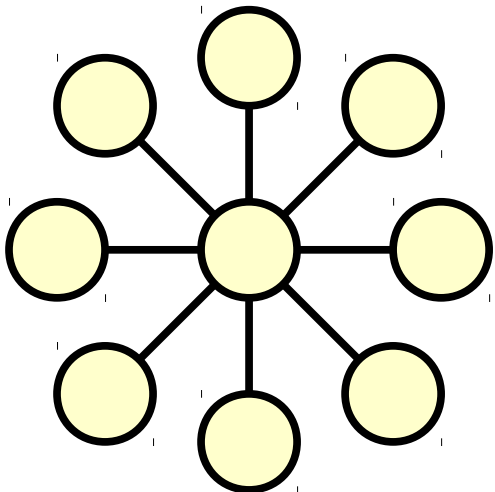


Whenever we process a node, we do

1. some fixed amount of work to process the node, then
2. some amount of work proportional to the number of edges touching that node.

# BFS Efficiency

- Suppose our graph has  $n$  nodes and  $m$  edges.
  - These letters are the standard conventions for talking about graphs.
- Average work done per node:  $O(1)$  baseline work, plus  $O(m/n)$  work processing edges.
- Number of nodes:  $n$ .
- Total work done:  $n \cdot (O(m/n) + O(1)) = \mathbf{O(m + n)}$ .



Whenever we process a node, we do

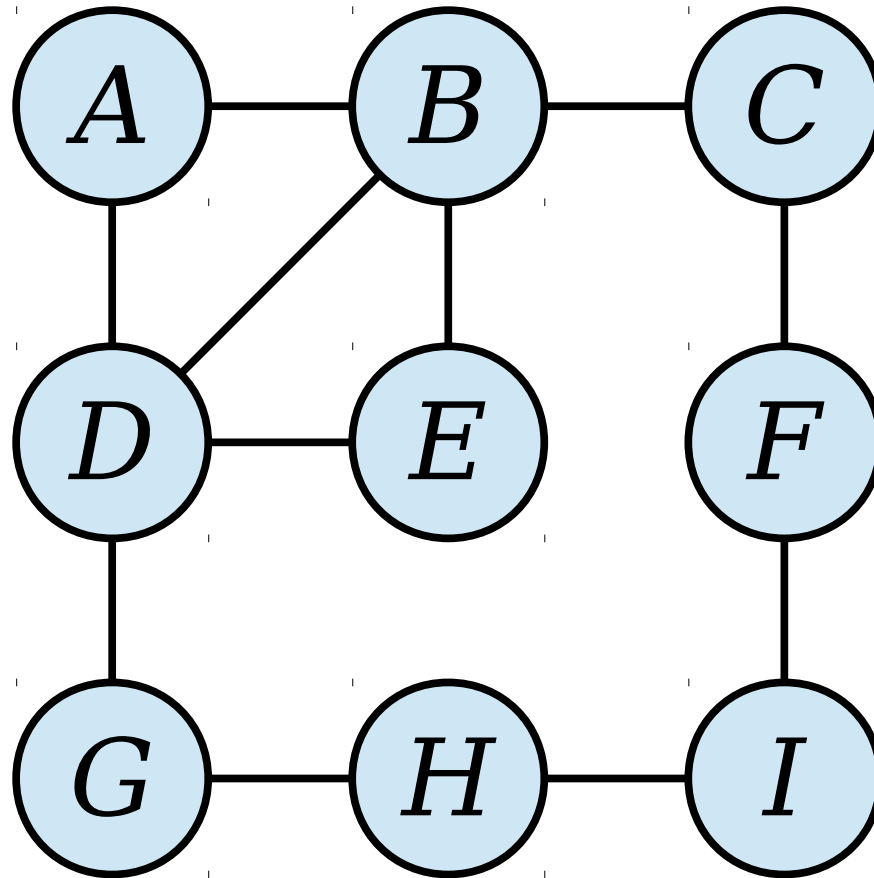
1. some fixed amount of work to process the node, then
2. some amount of work proportional to the number of edges touching that node.



# BFS Efficiency

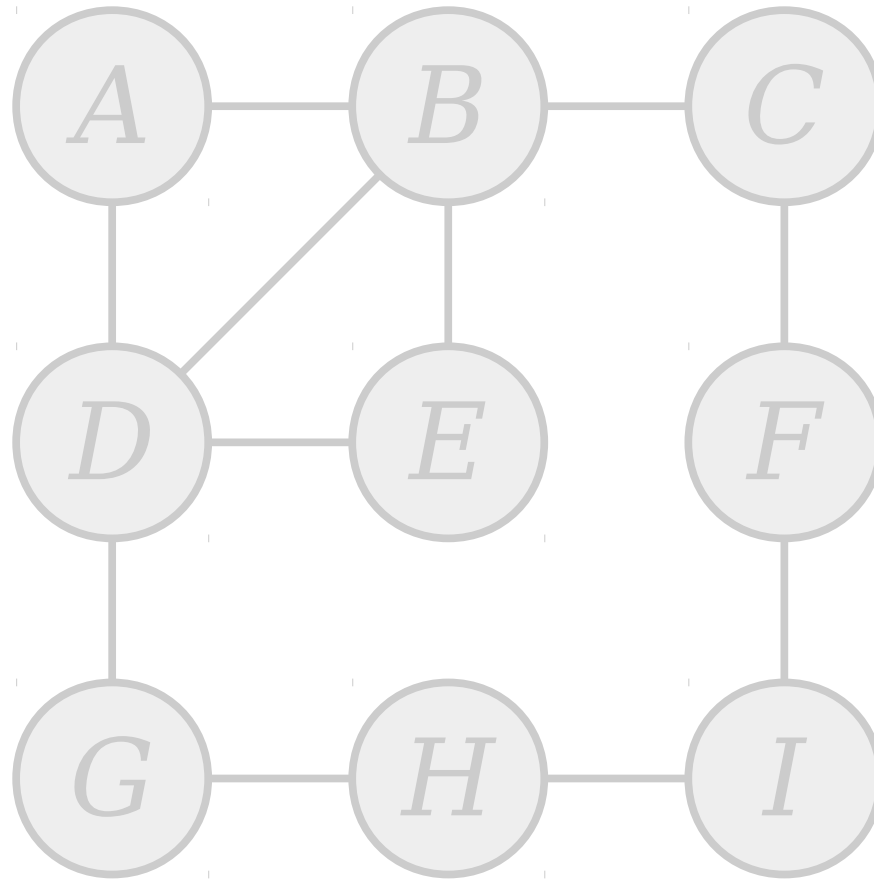
- The amount of work done to run breadth-first search is  $O(m + n)$ , assuming the graph is represented as an adjacency list.
  - Great question to ponder: how fast is breadth-first search if we use an adjacency matrix?
- The work done is proportional to the number of objects (nodes and edges) that make up the graph.
- We say that BFS is a ***linear-time*** graph algorithm.

# A Nifty BFS Trick



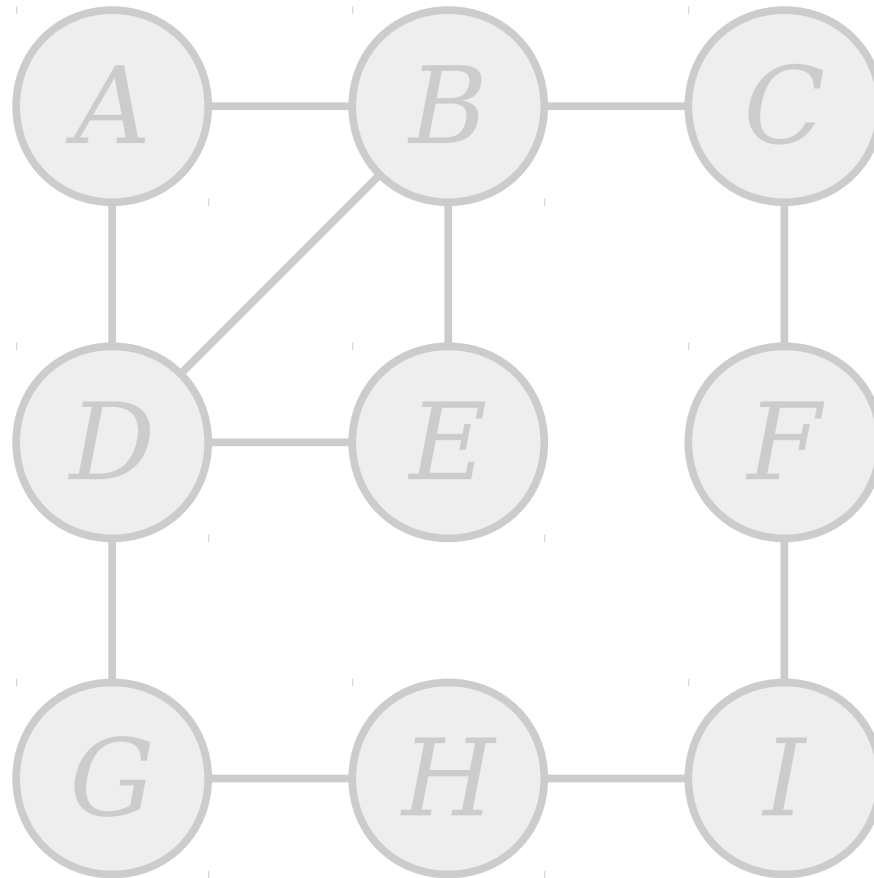
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:



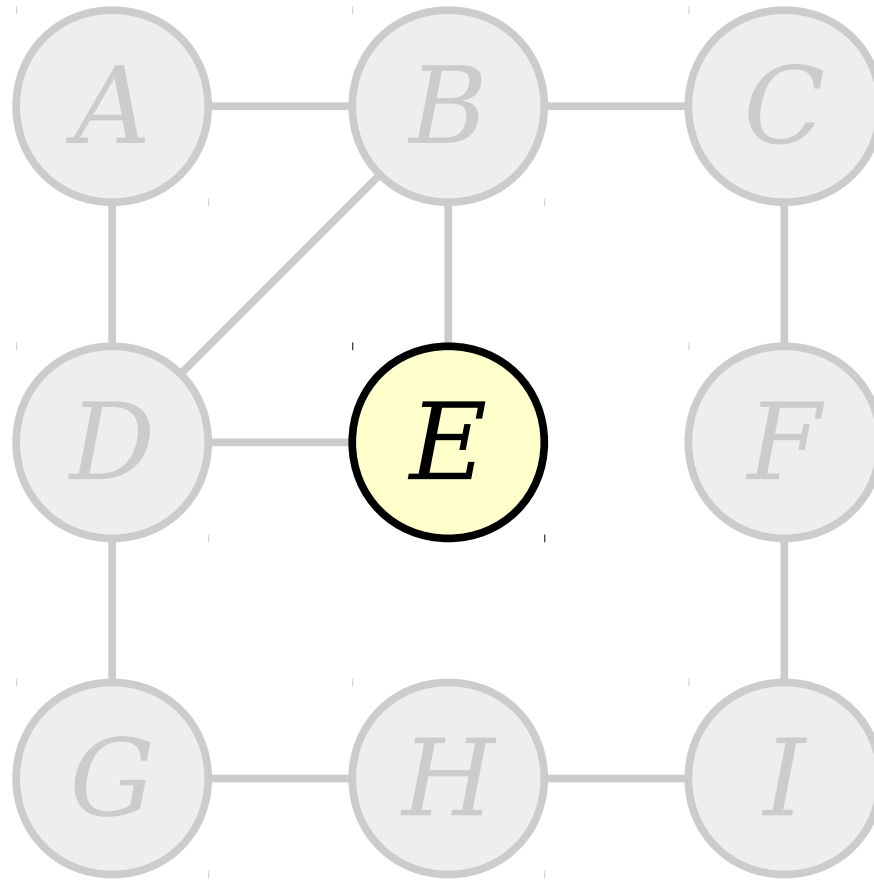
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:



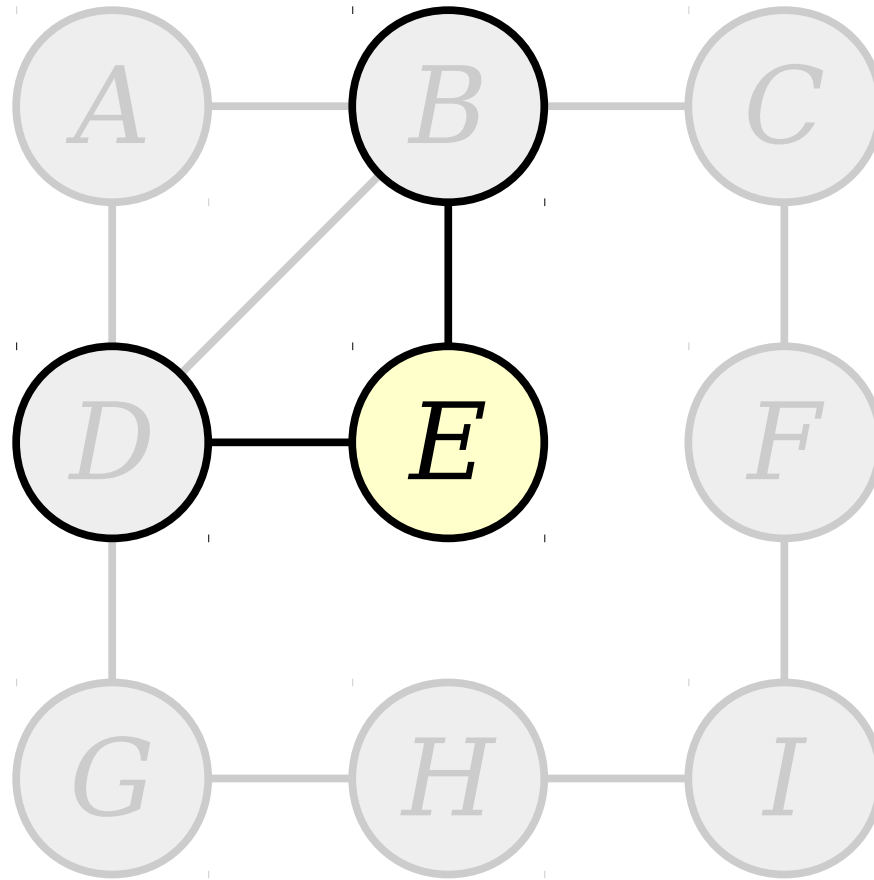
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **E**



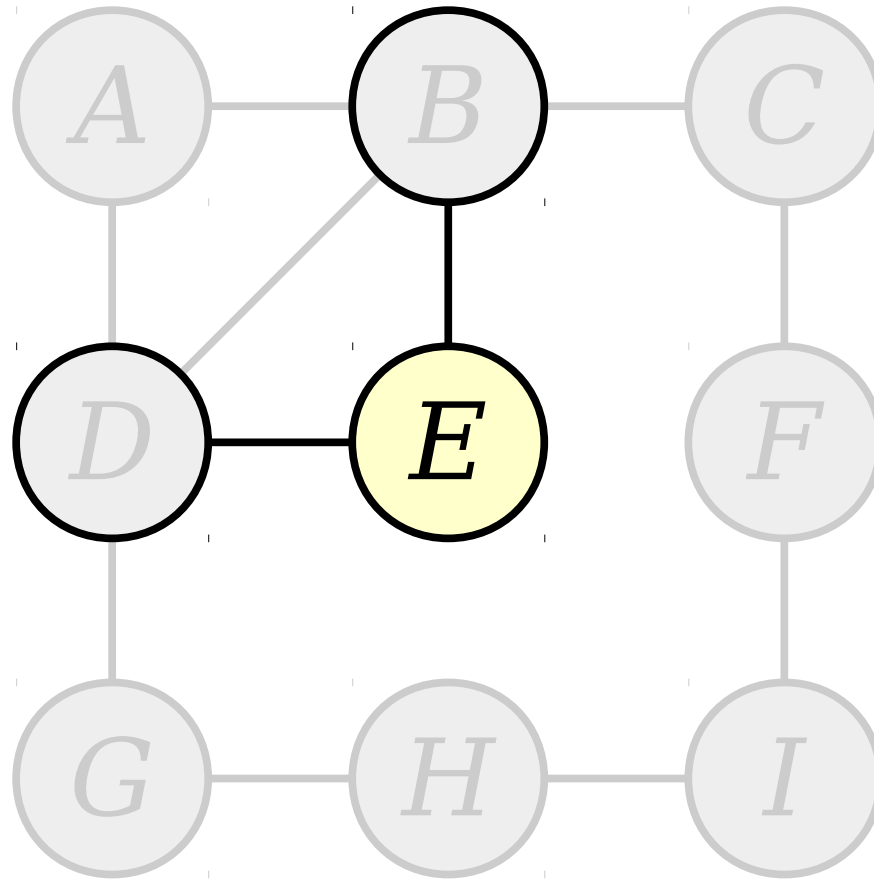
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue:



Run BFS, but have each node store a pointer back to the node that first discovered it.

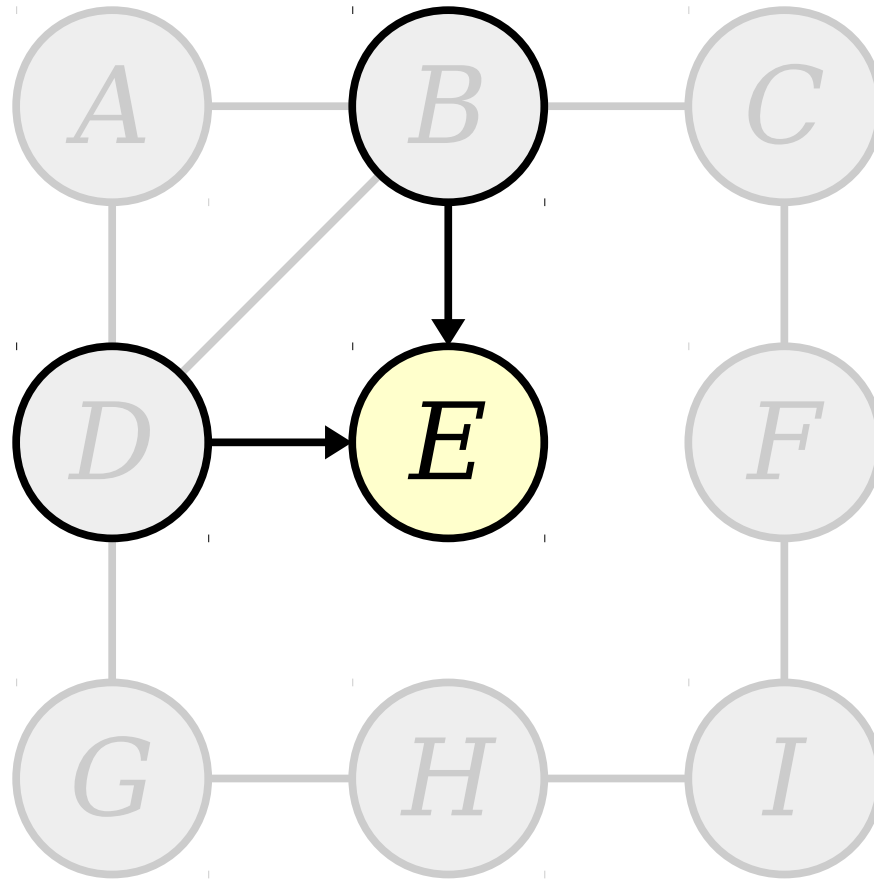
Queue:



Run BFS, but have each node store a pointer back to the node that first discovered it.

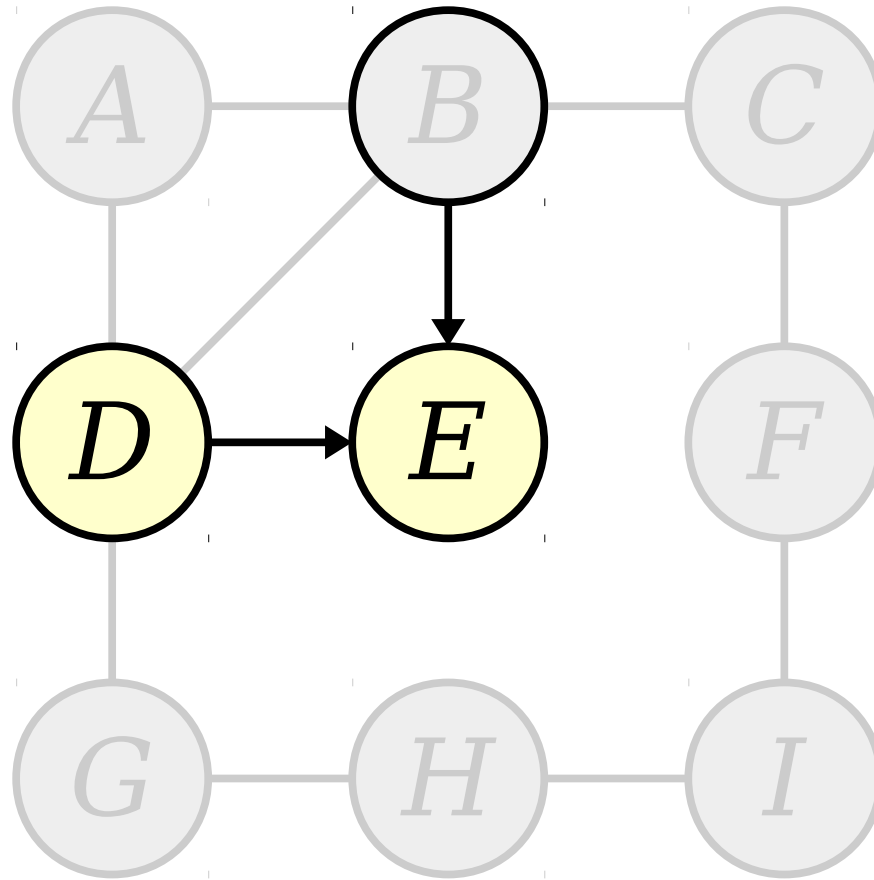
Queue: **D** **B**





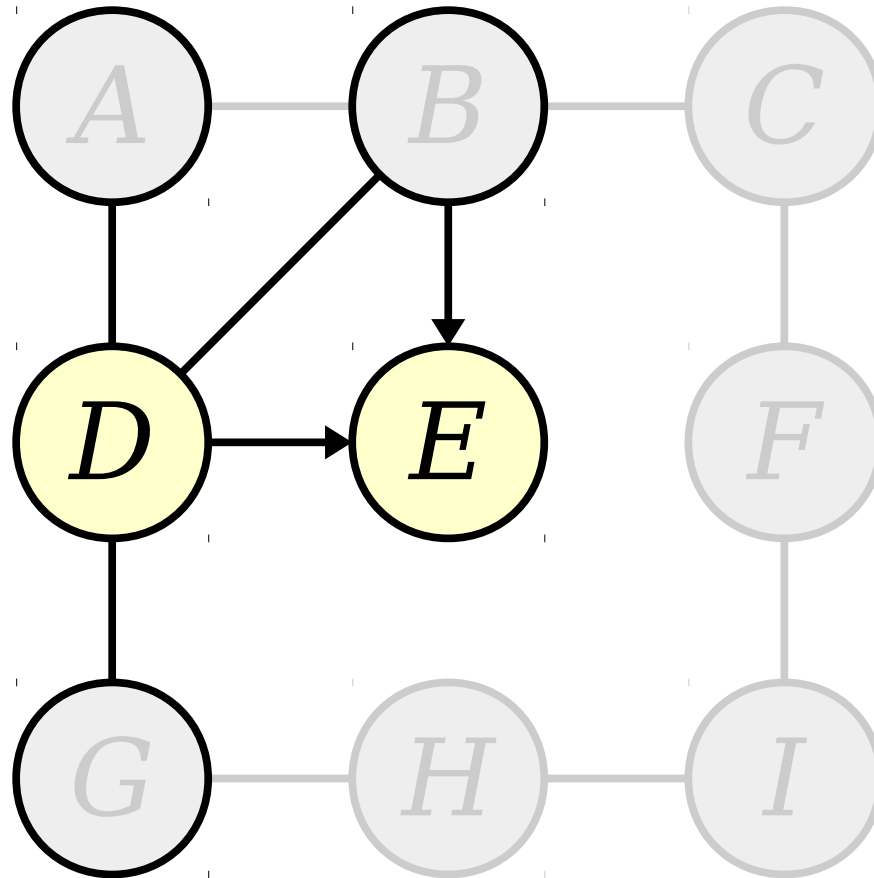
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **D** **B**



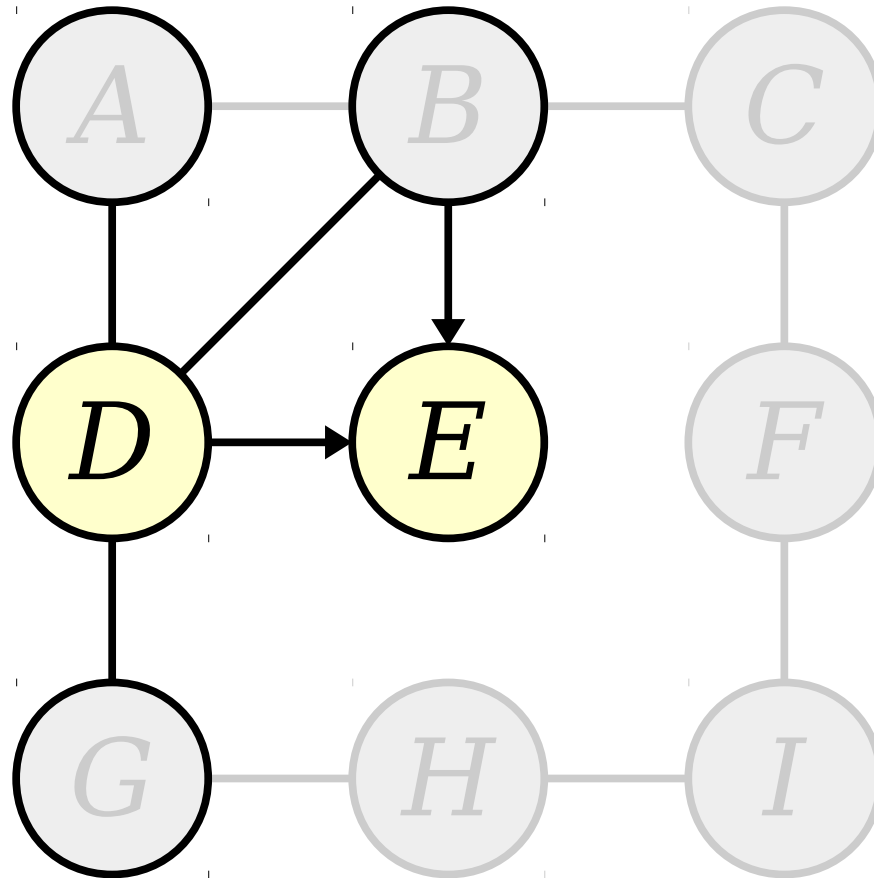
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **B**



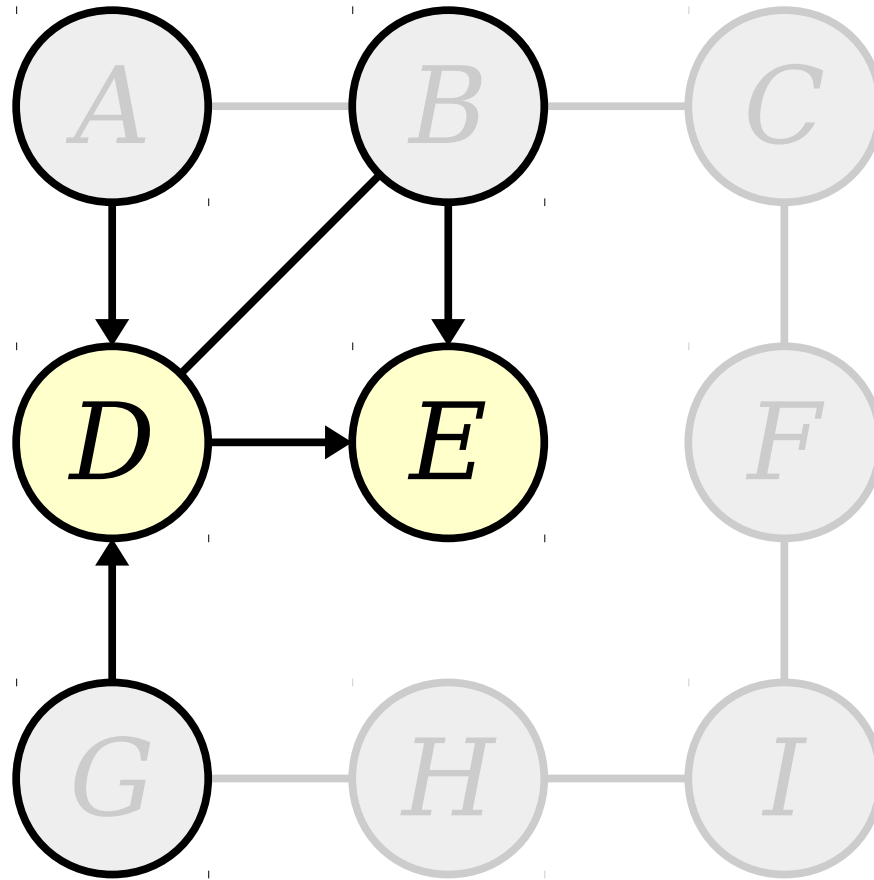
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **B**



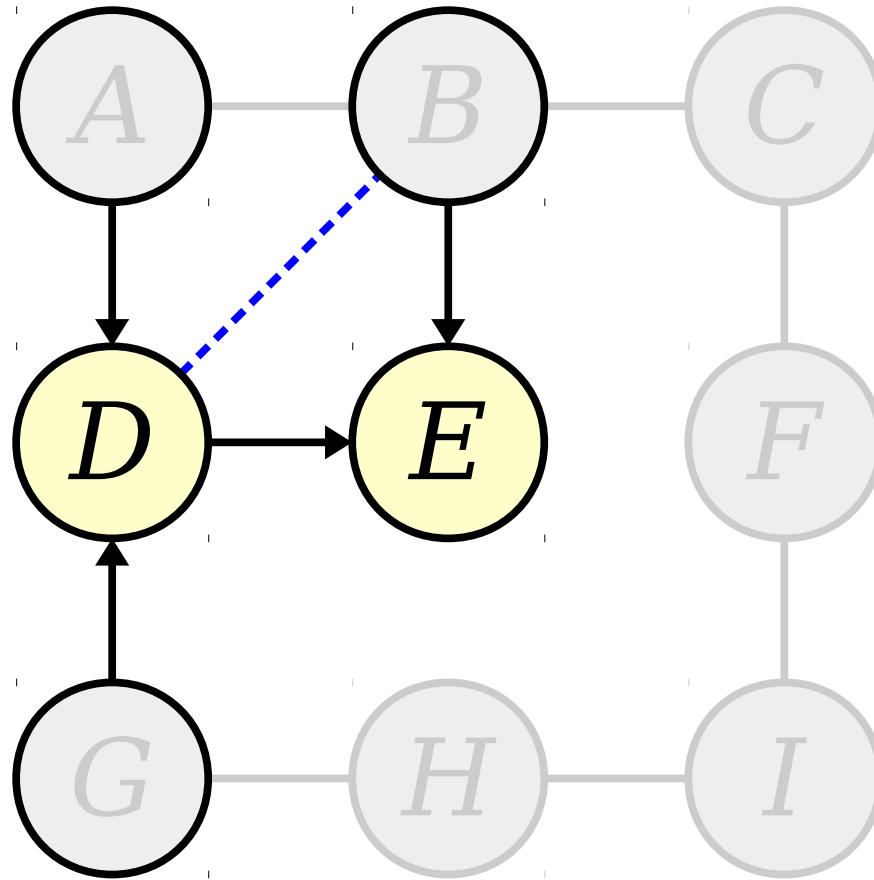
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **B** **A** **G**



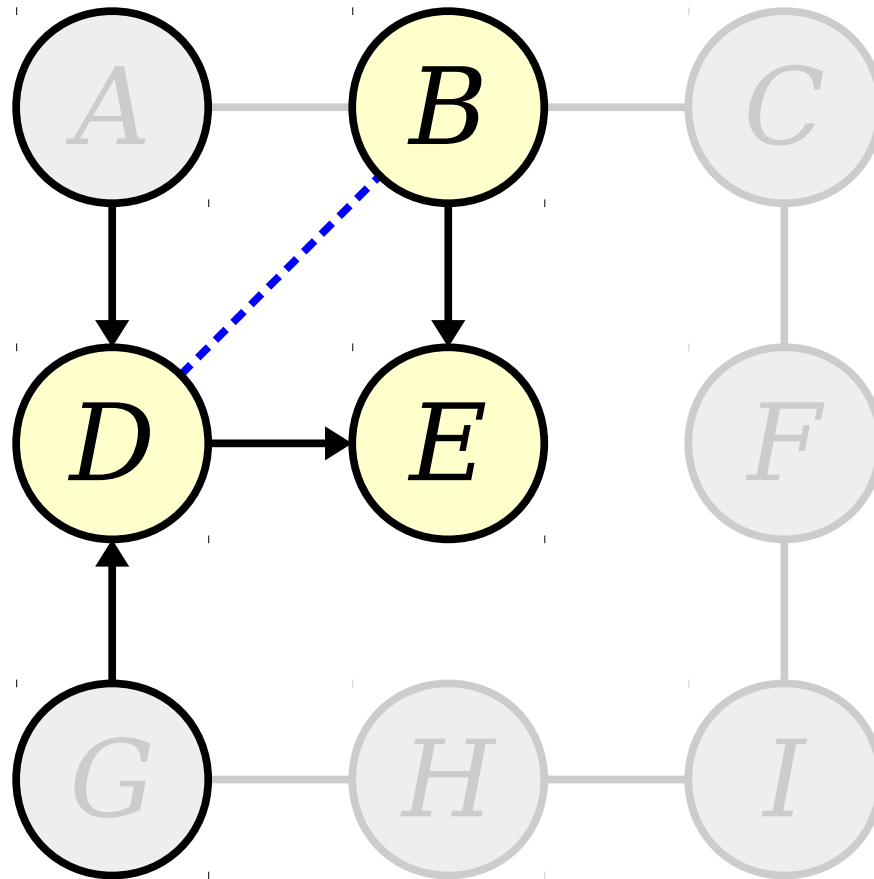
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **B** **A** **G**



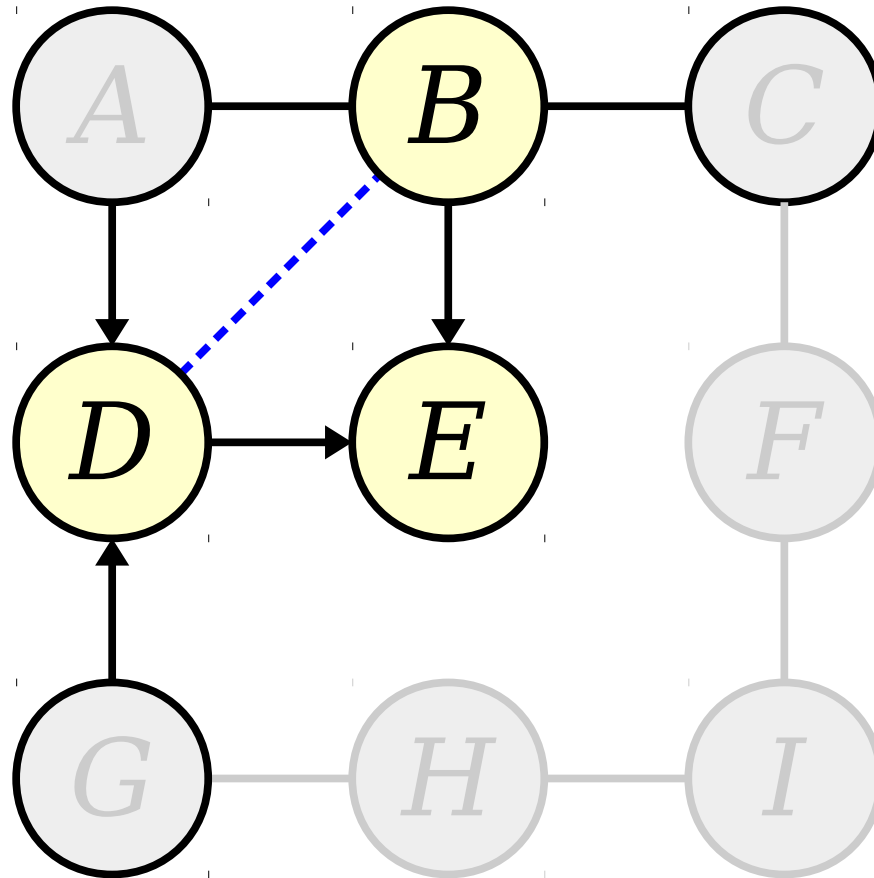
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **B** **A** **G**



Run BFS, but have each node store a pointer back to the node that first discovered it.

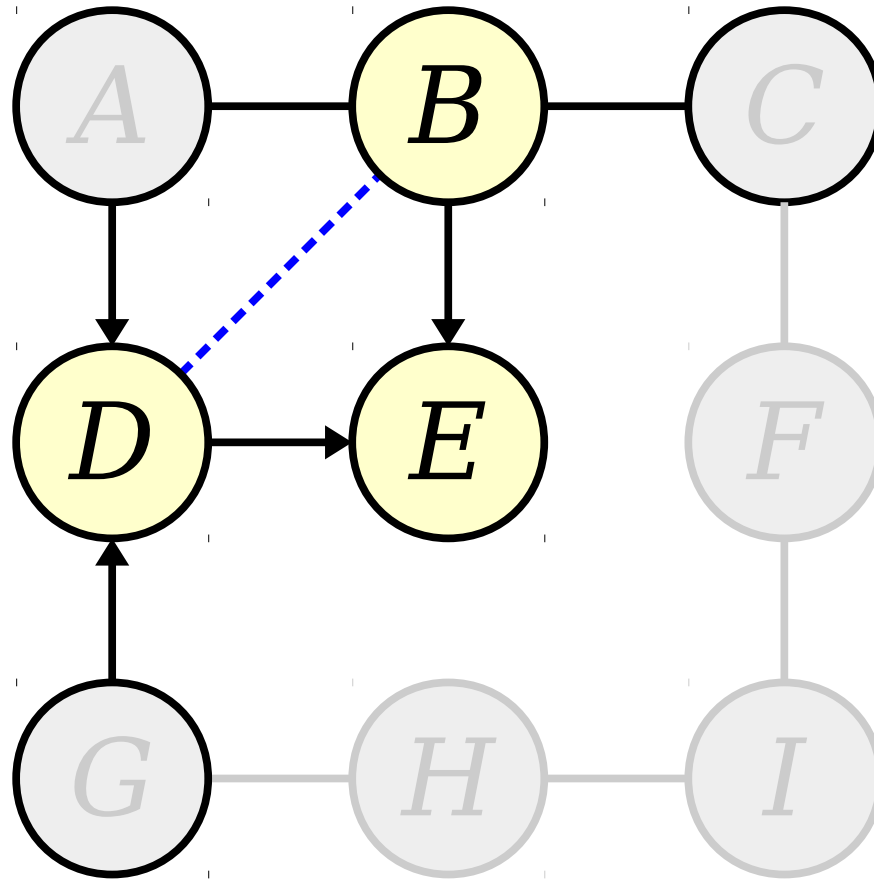
Queue: **A** **G**



Run BFS, but have each node store a pointer back to the node that first discovered it.

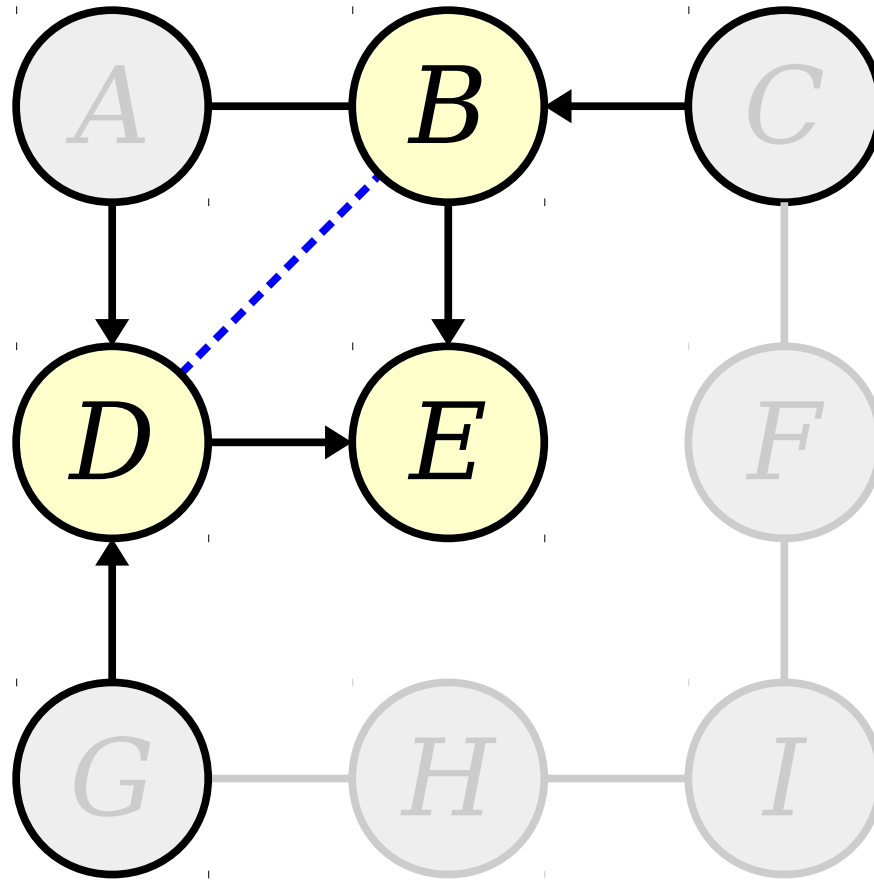
Queue: **A** **G**





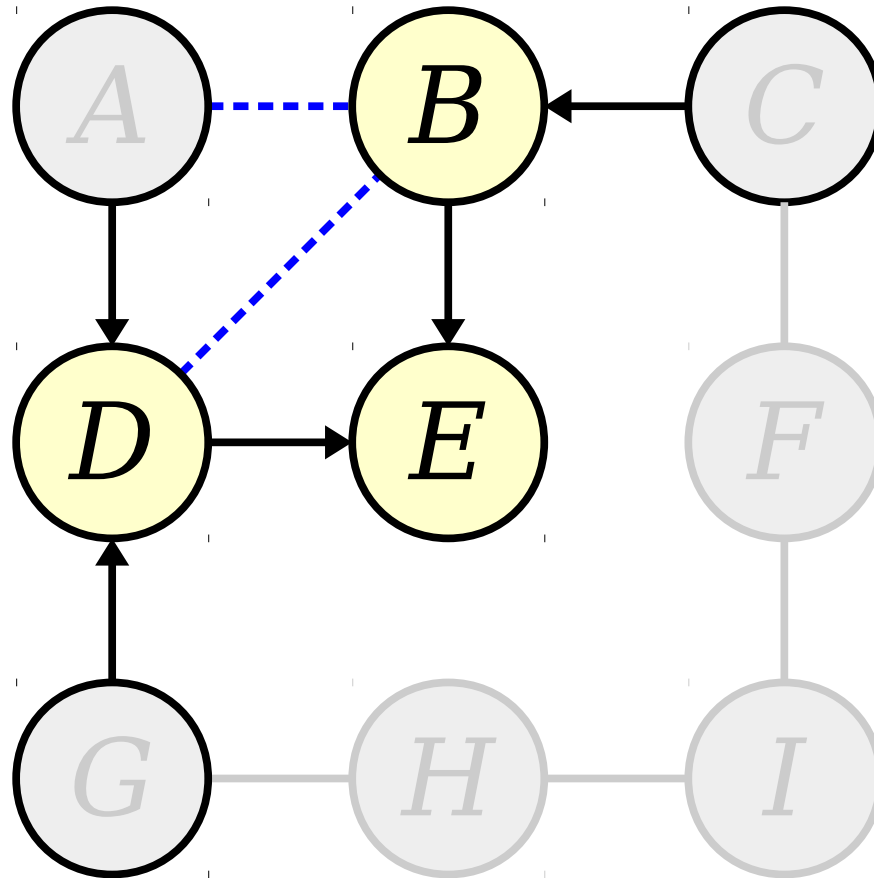
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **A** **G** **C**



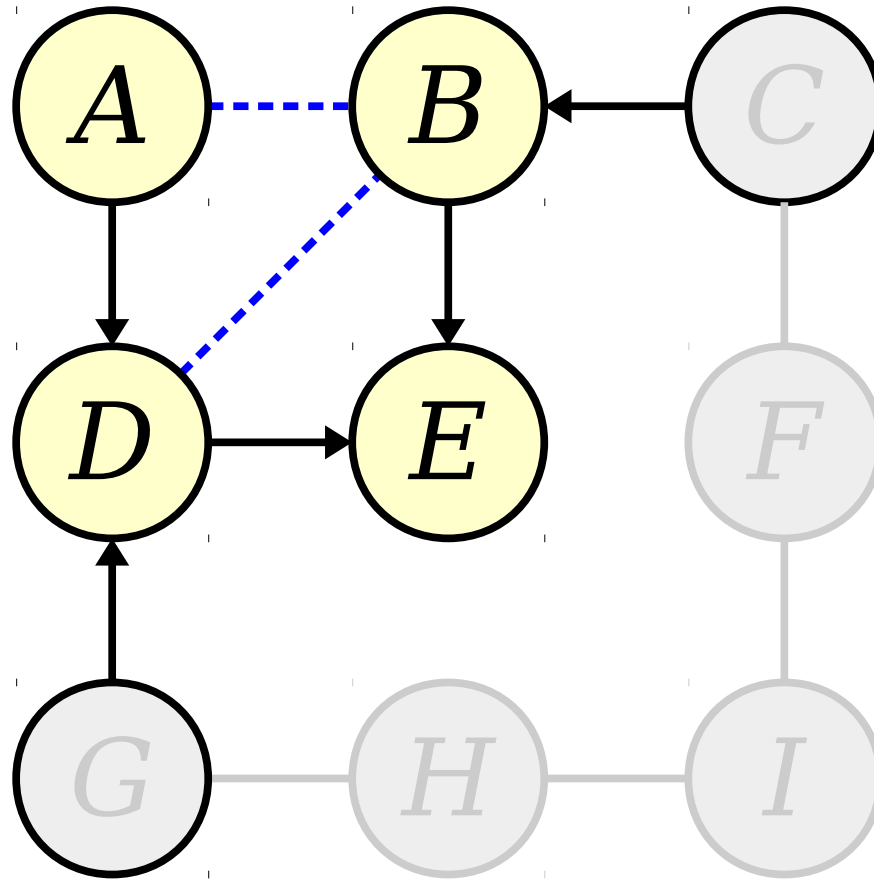
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **A** **G** **C**



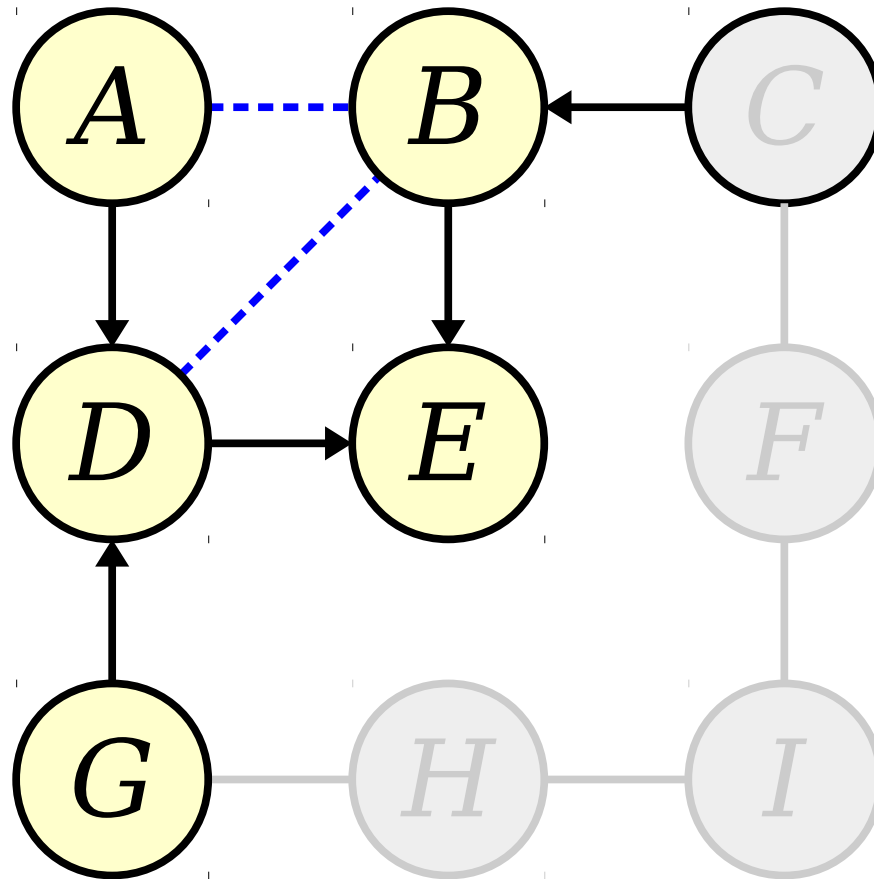
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **A** **G** **C**



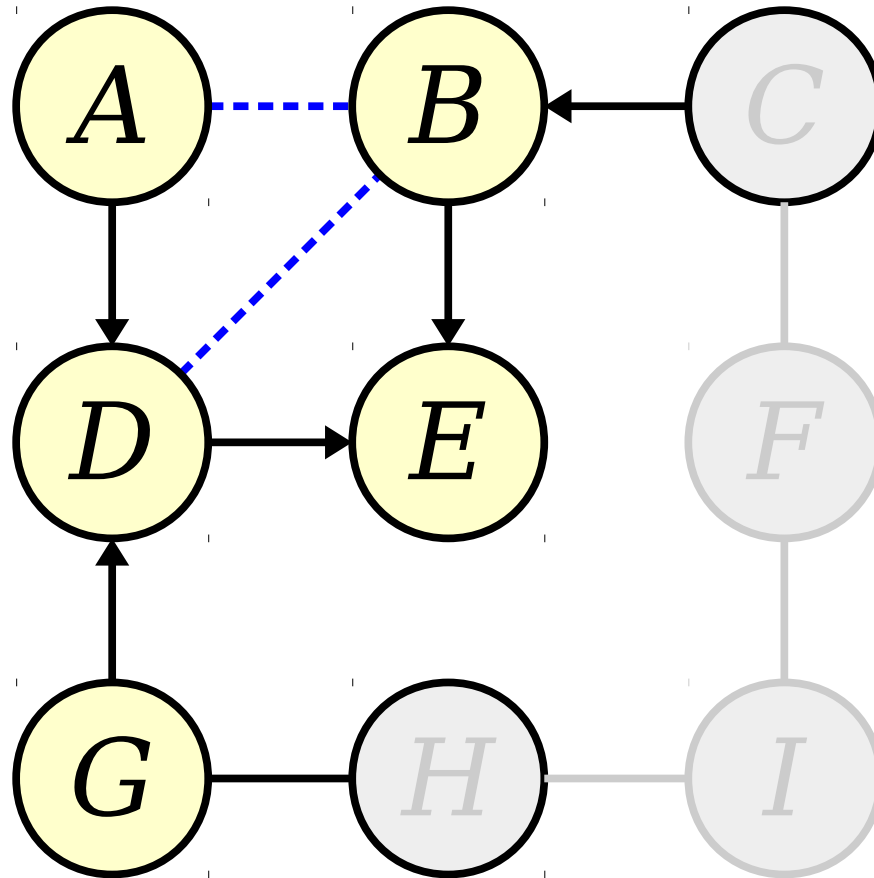
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **G** **C**



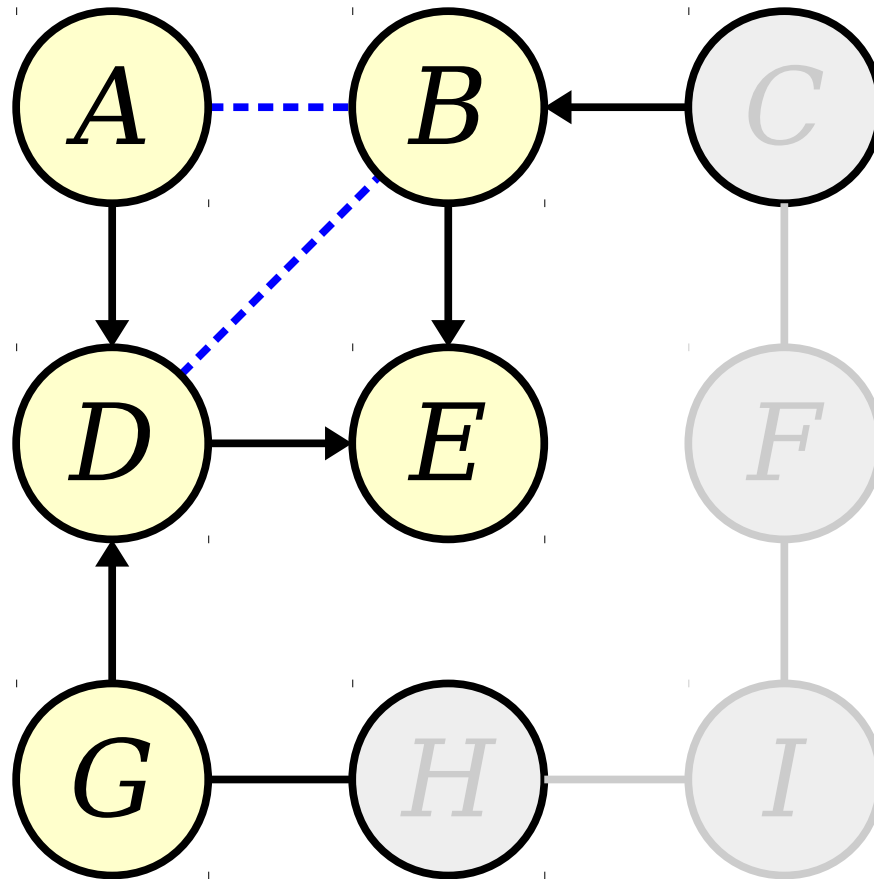
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **G** **C**



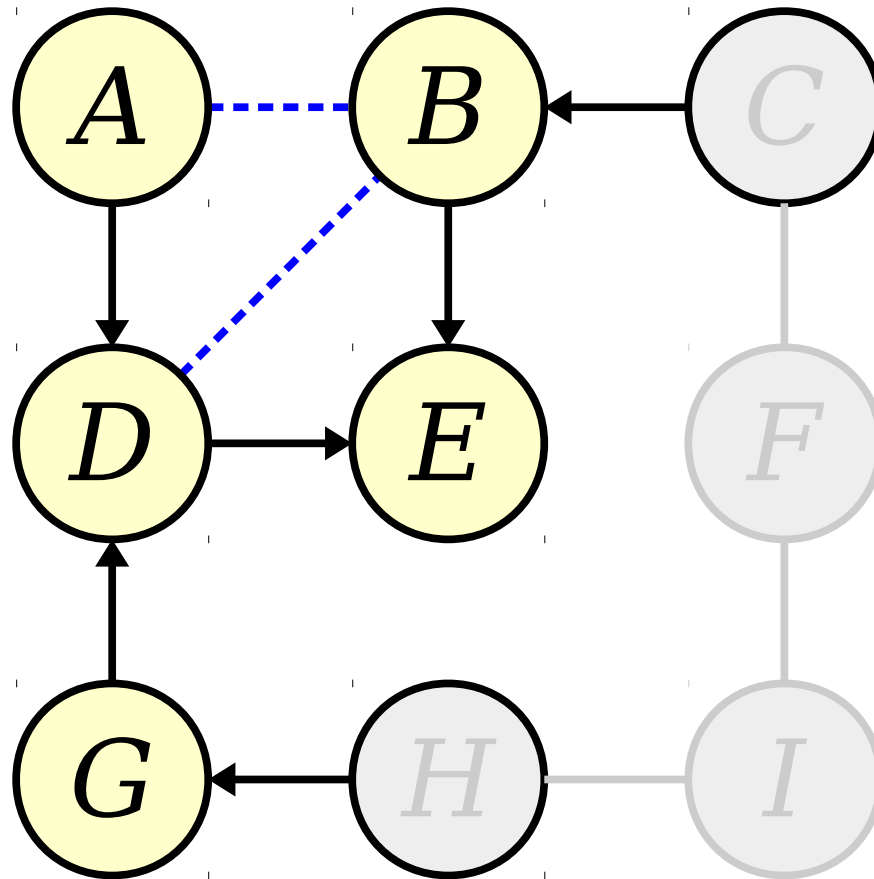
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **C**



Run BFS, but have each node store a pointer back to the node that first discovered it.

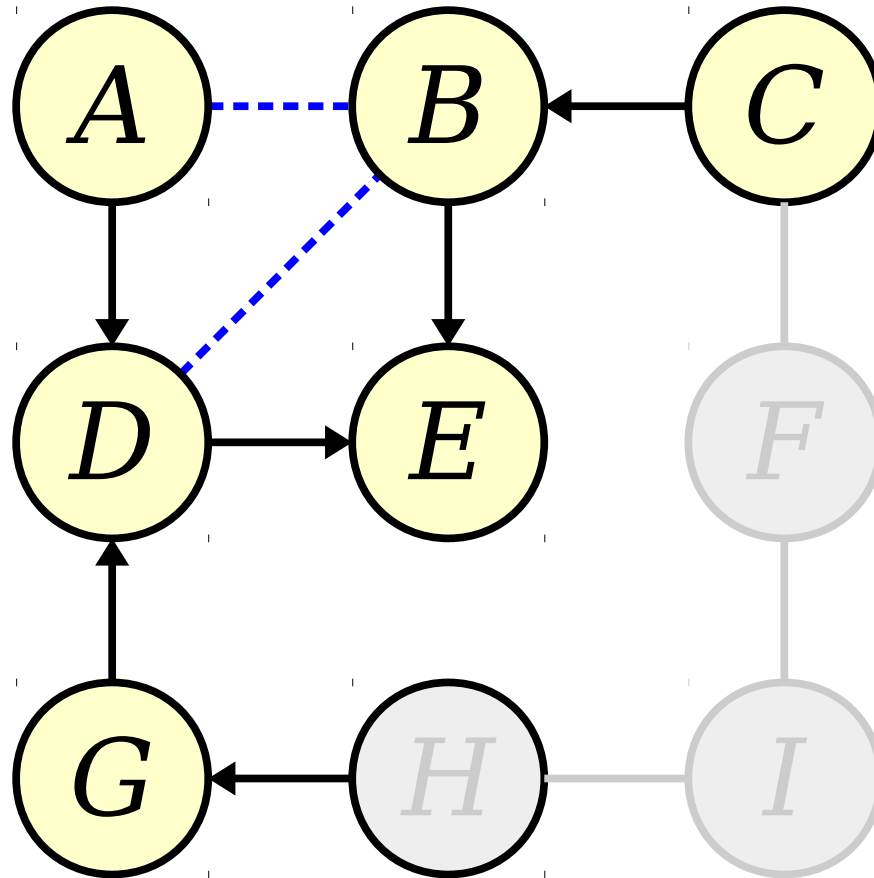
Queue: **C** **H**




Run BFS, but have each node store a pointer back to the node that first discovered it.

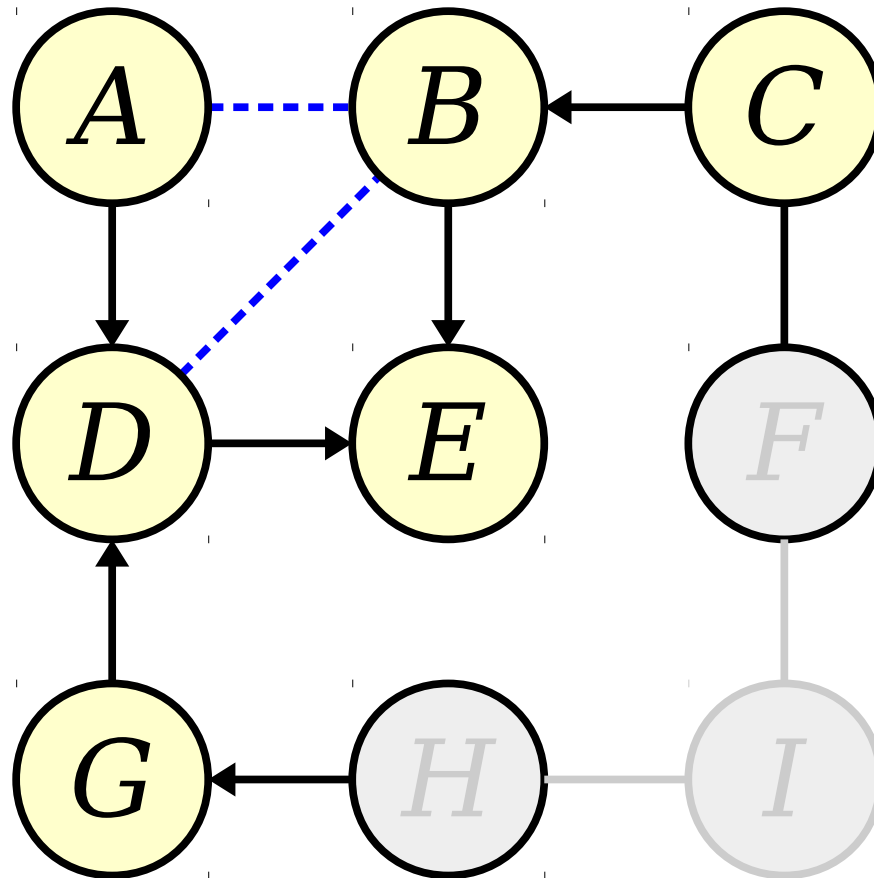
Queue: **C** **H**





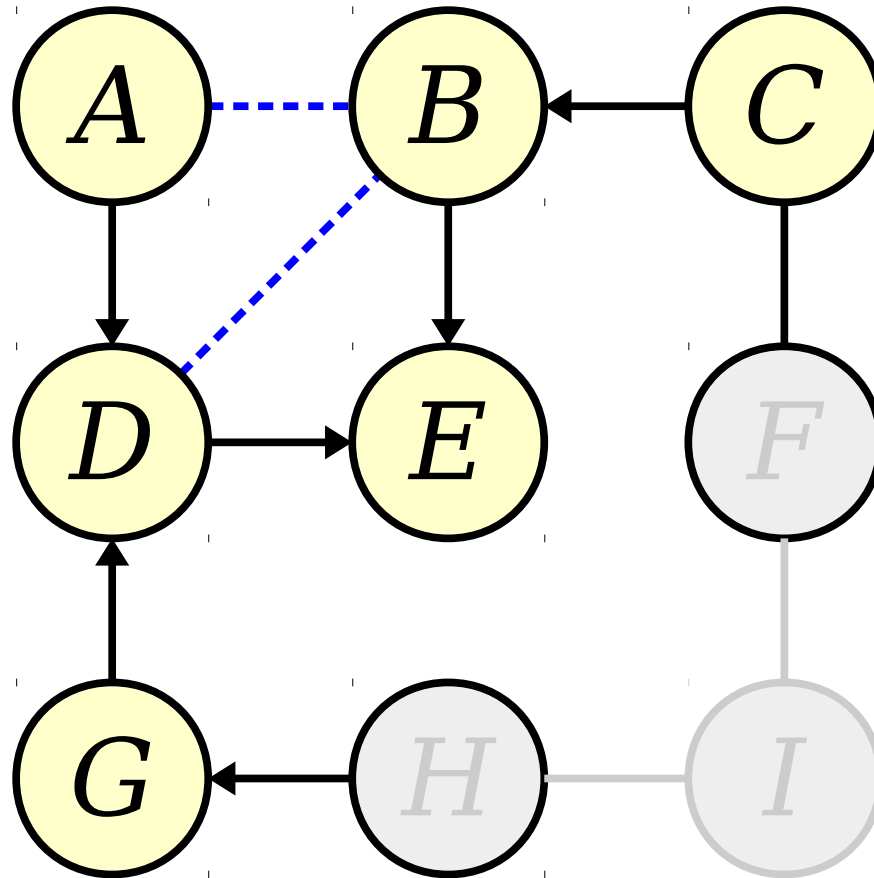
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 



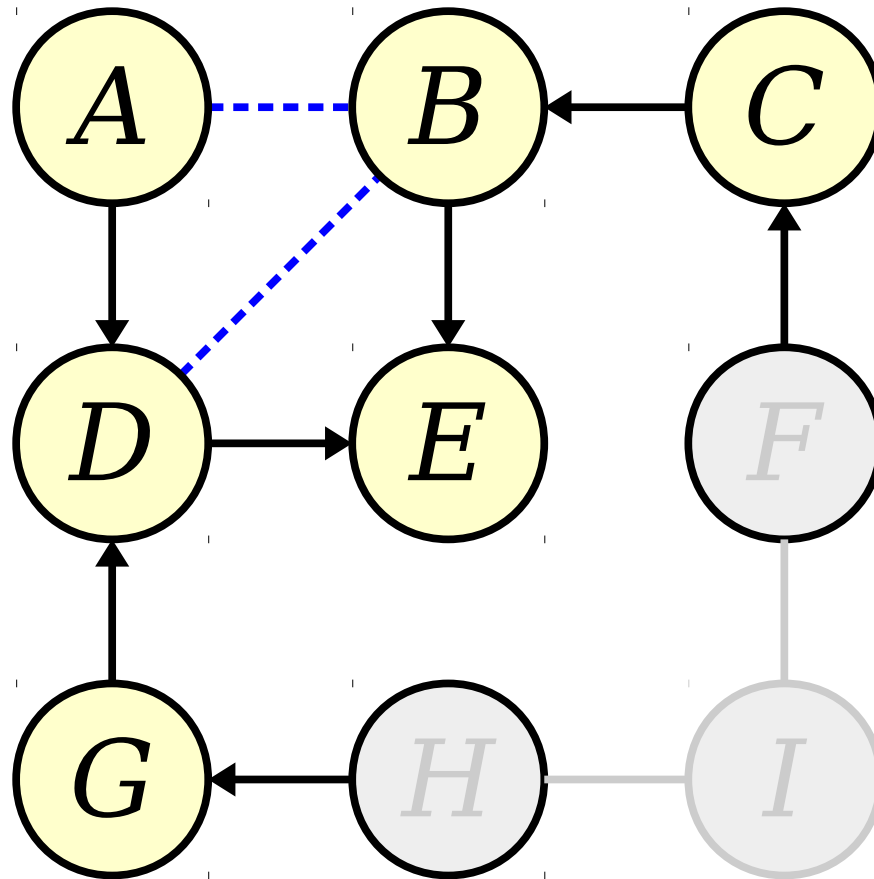
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: **H**



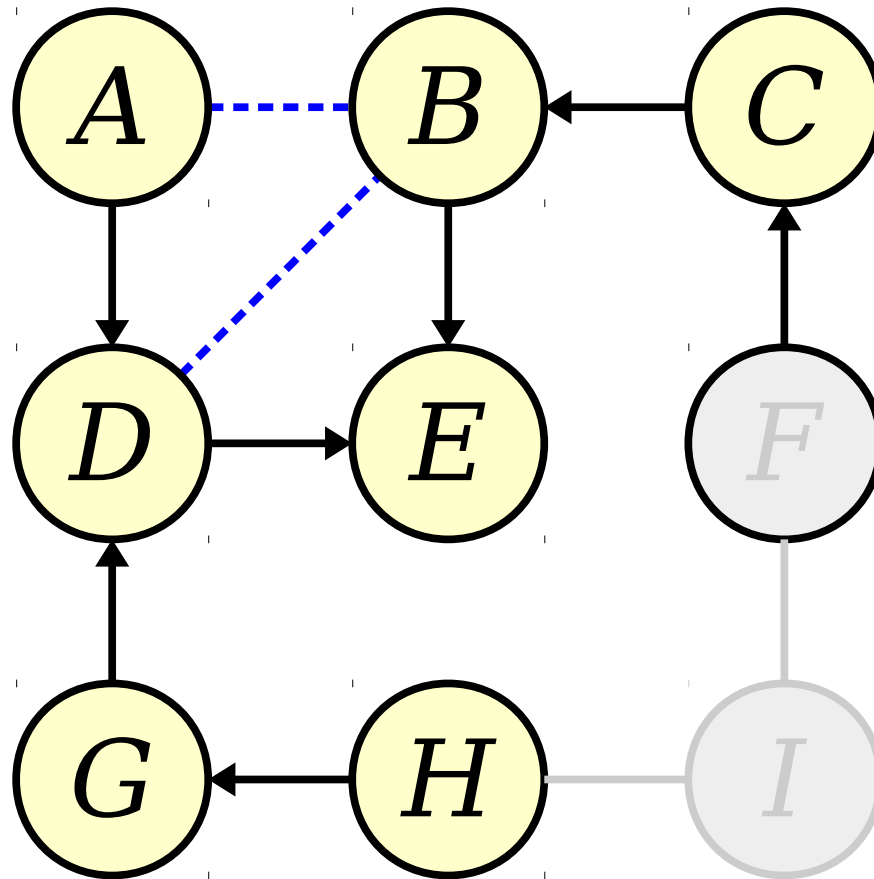
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: *H* *F*




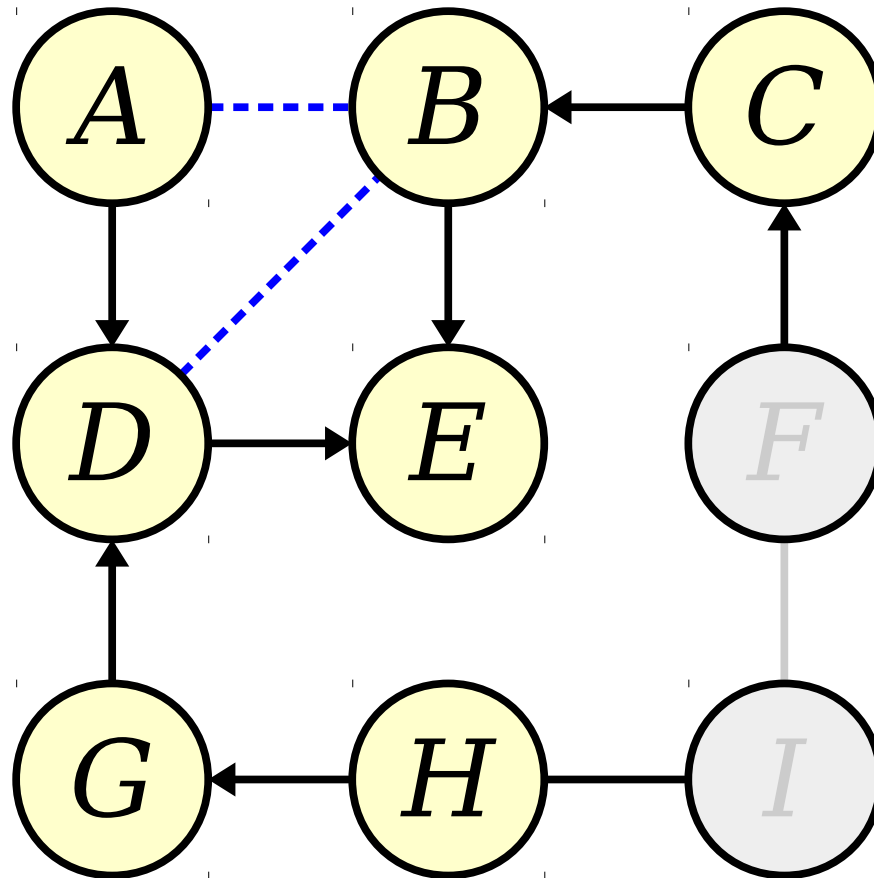
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: H F




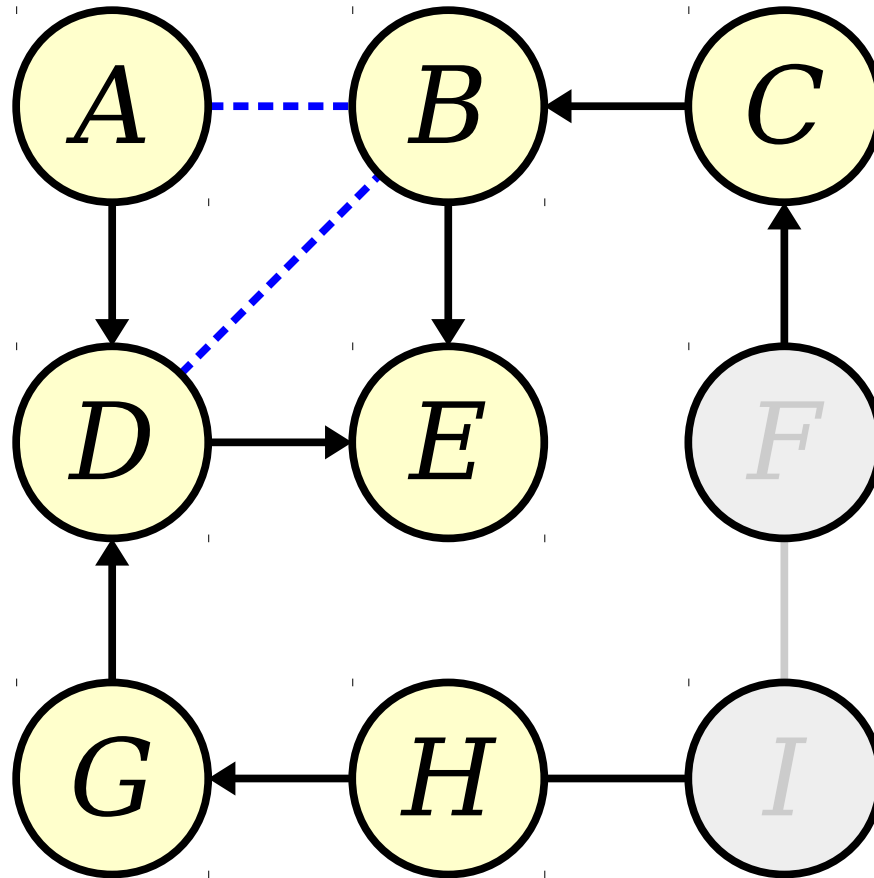
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 



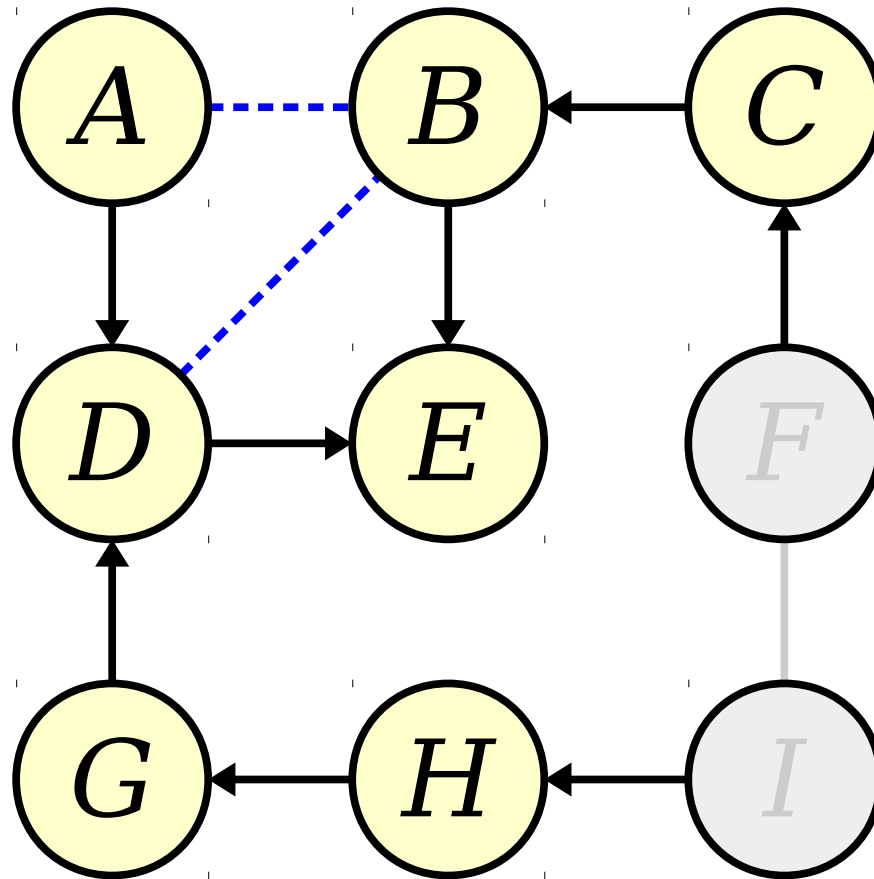
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 





Run BFS, but have each node store a pointer back to the node that first discovered it.

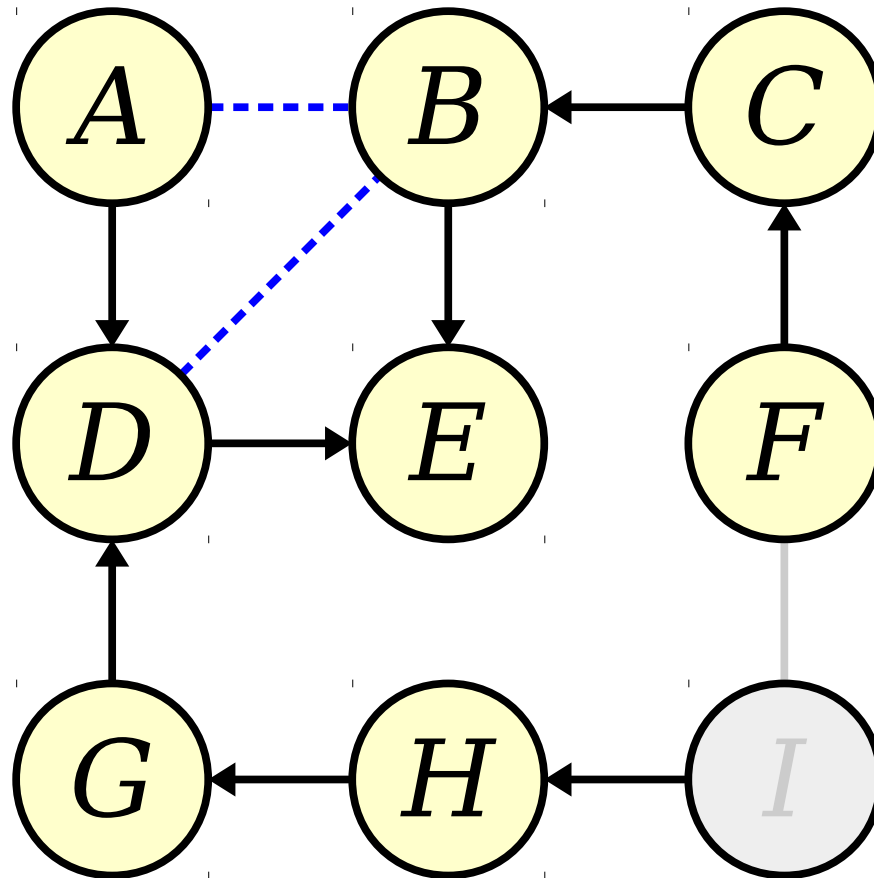
Queue: *F* *I*




Run BFS, but have each node store a pointer back to the node that first discovered it.

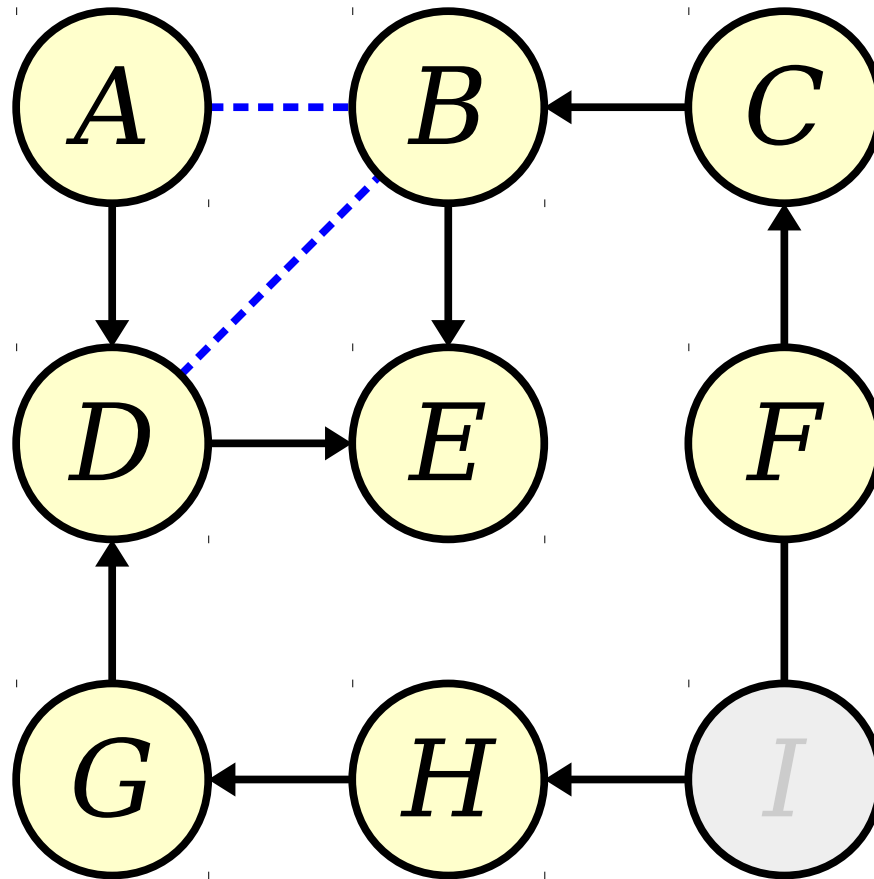
Queue:  






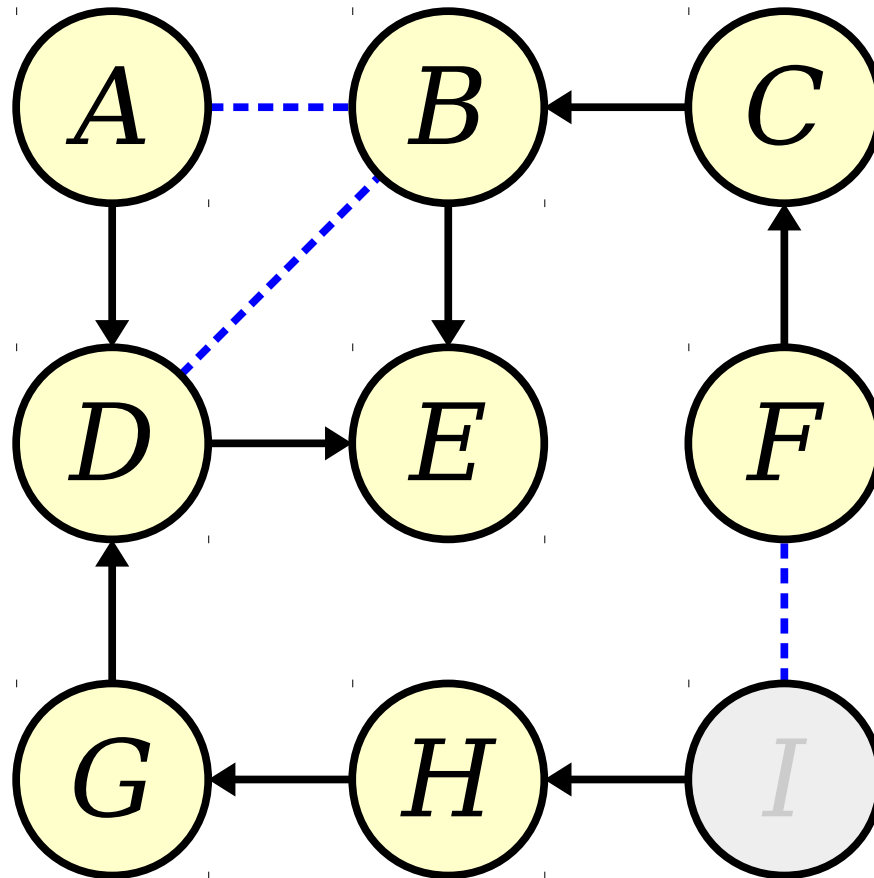
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 




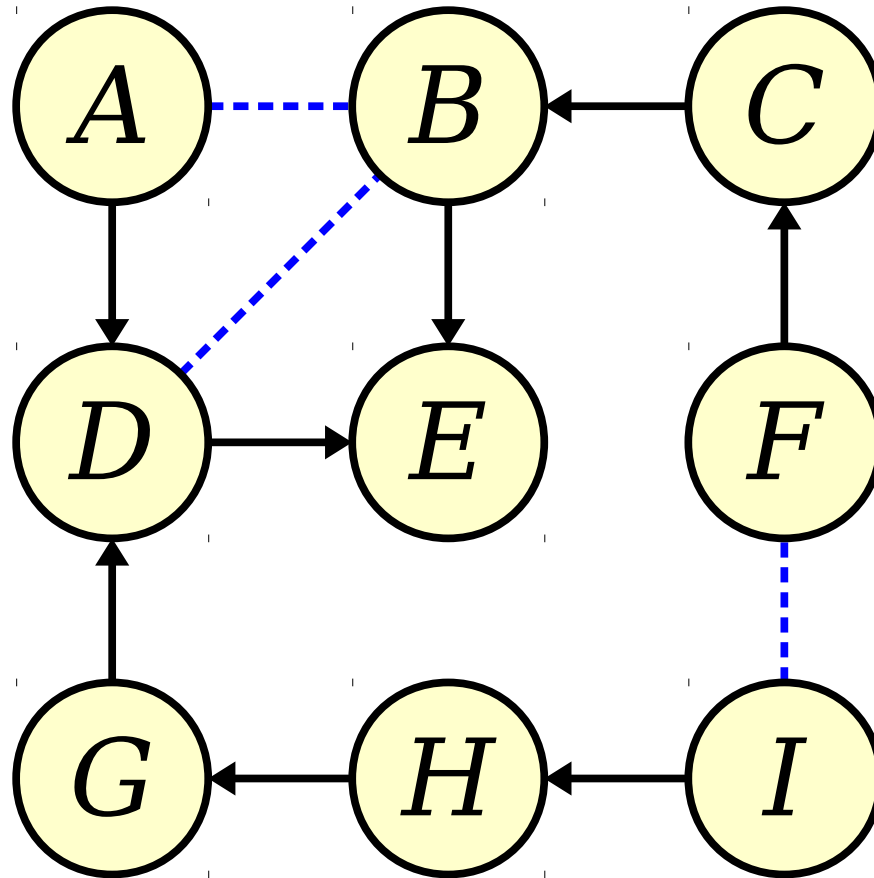
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 



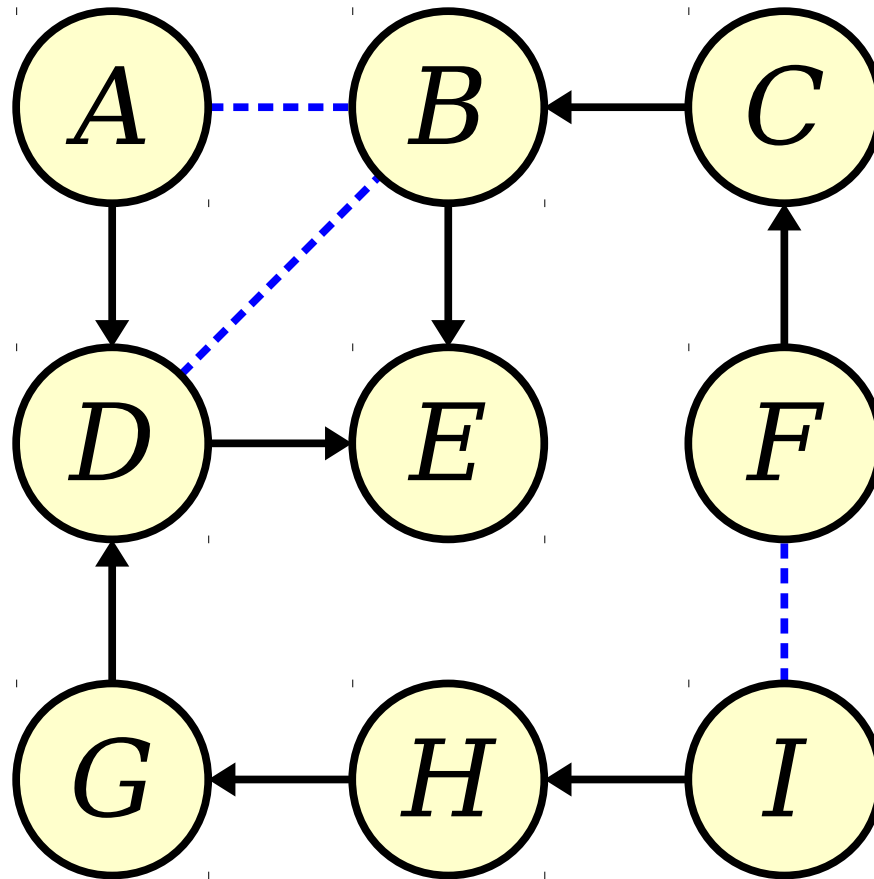
Run BFS, but have each node store a pointer back to the node that first discovered it.

Queue: 

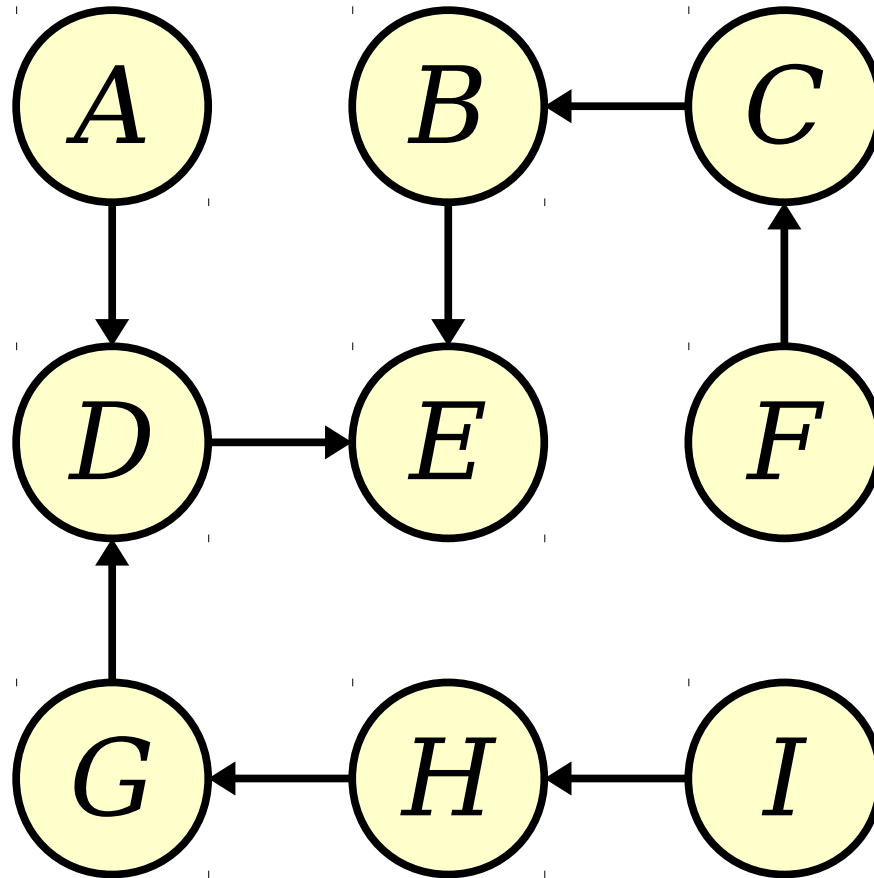


Run BFS, but have each node store a pointer back to the node that first discovered it.

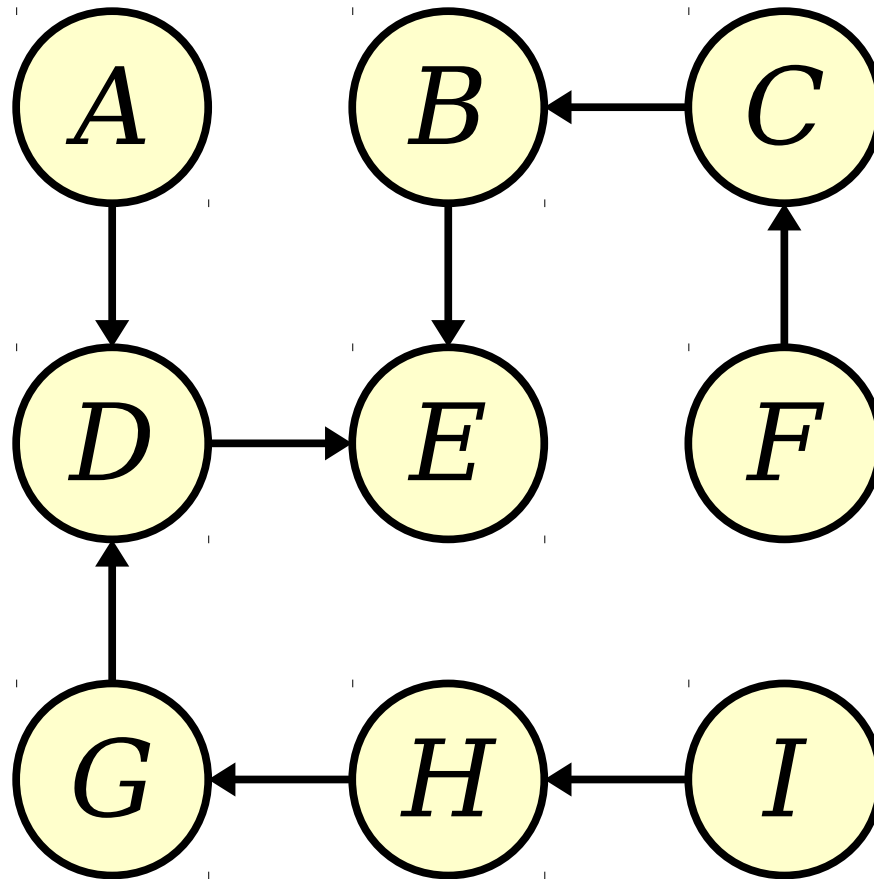
Queue:



Run BFS, but have each node store a pointer back to the node that first discovered it.



Run BFS, but have each node store a pointer back to the node that first discovered it.



Run BFS, but have each node store a pointer back to the node that first discovered it.

Start at any node and follow pointers until you reach E. What path are you tracing out?

# Shortest Path Routing

- Breadth-first search can be used to find a shortest path from each node back to the start node.
- The tree you get when you do this is called a ***breadth-first search tree*** and has lots of fun properties and cool applications.
- Want to learn more? Take CS161!



# Next Time

- ***Depth-First Search***
  - Another graph search algorithm.
- ***Directed Acyclic Graphs***
  - Representing prerequisites.
- ***Topological Sorting***
  - Ordering your “to do” list with constraints.