Search

Chris Piech

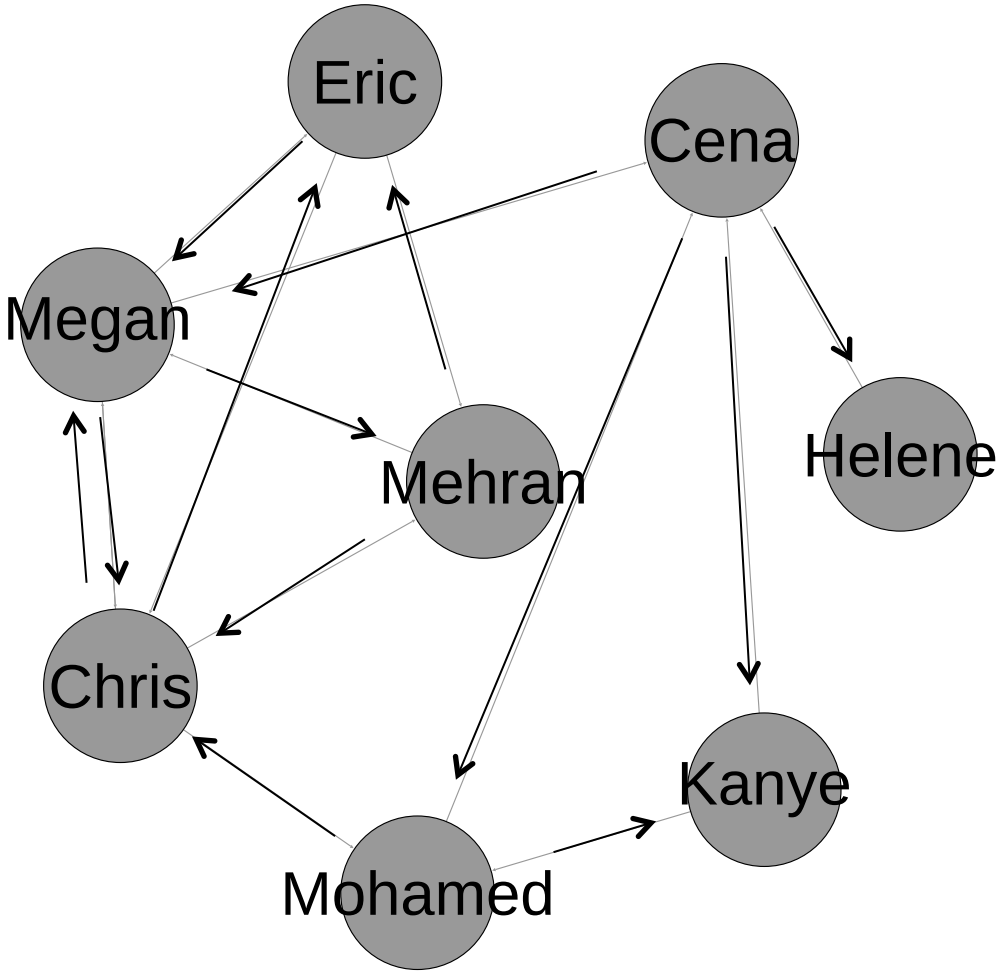
CS 106B
Lecture 20
Feb 29, 2016

Graph Definition

A **graph** is a mathematical structure for representing relationships using nodes and edges.

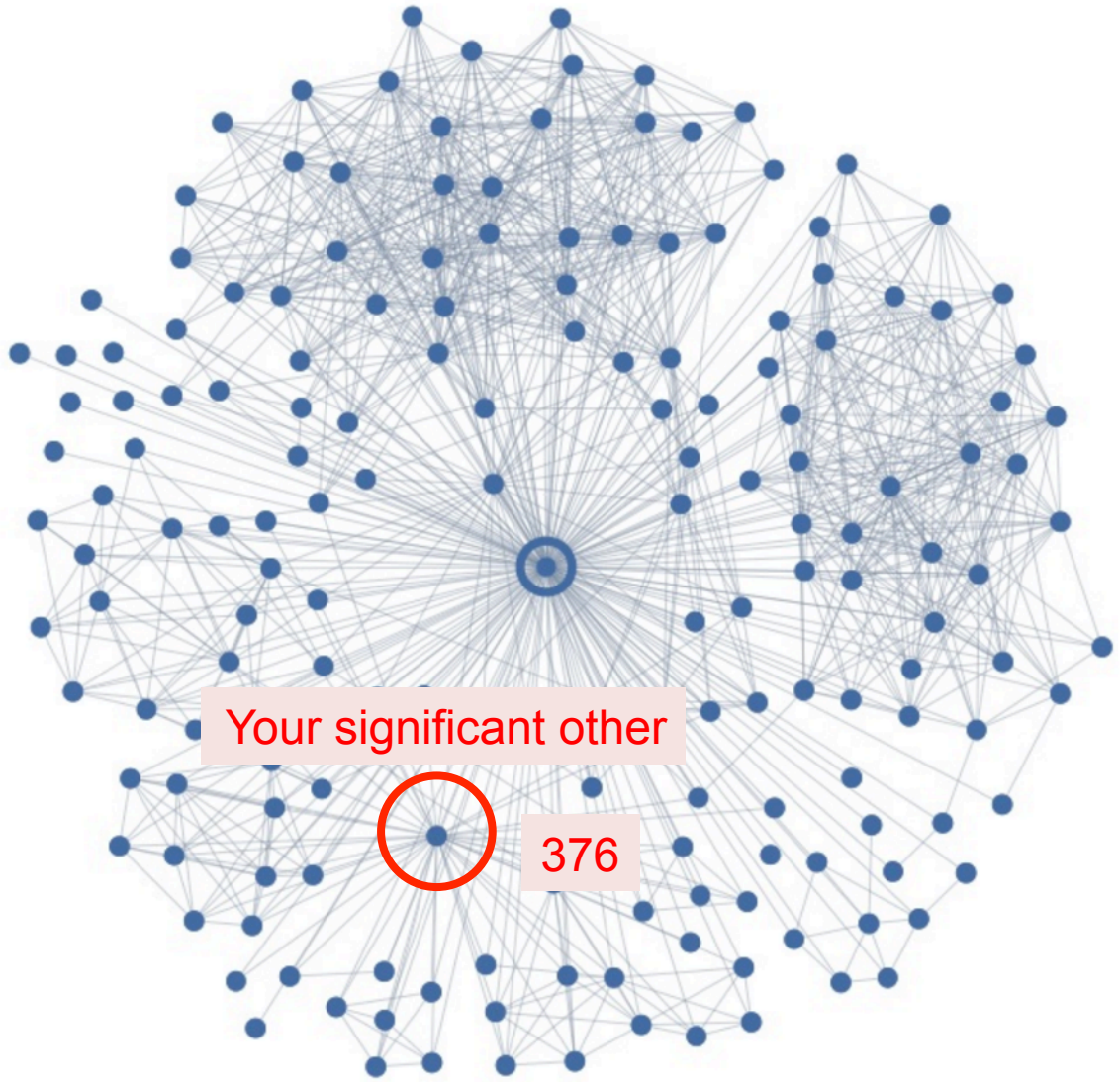
*Just like a tree without the rules

Twitter Graph



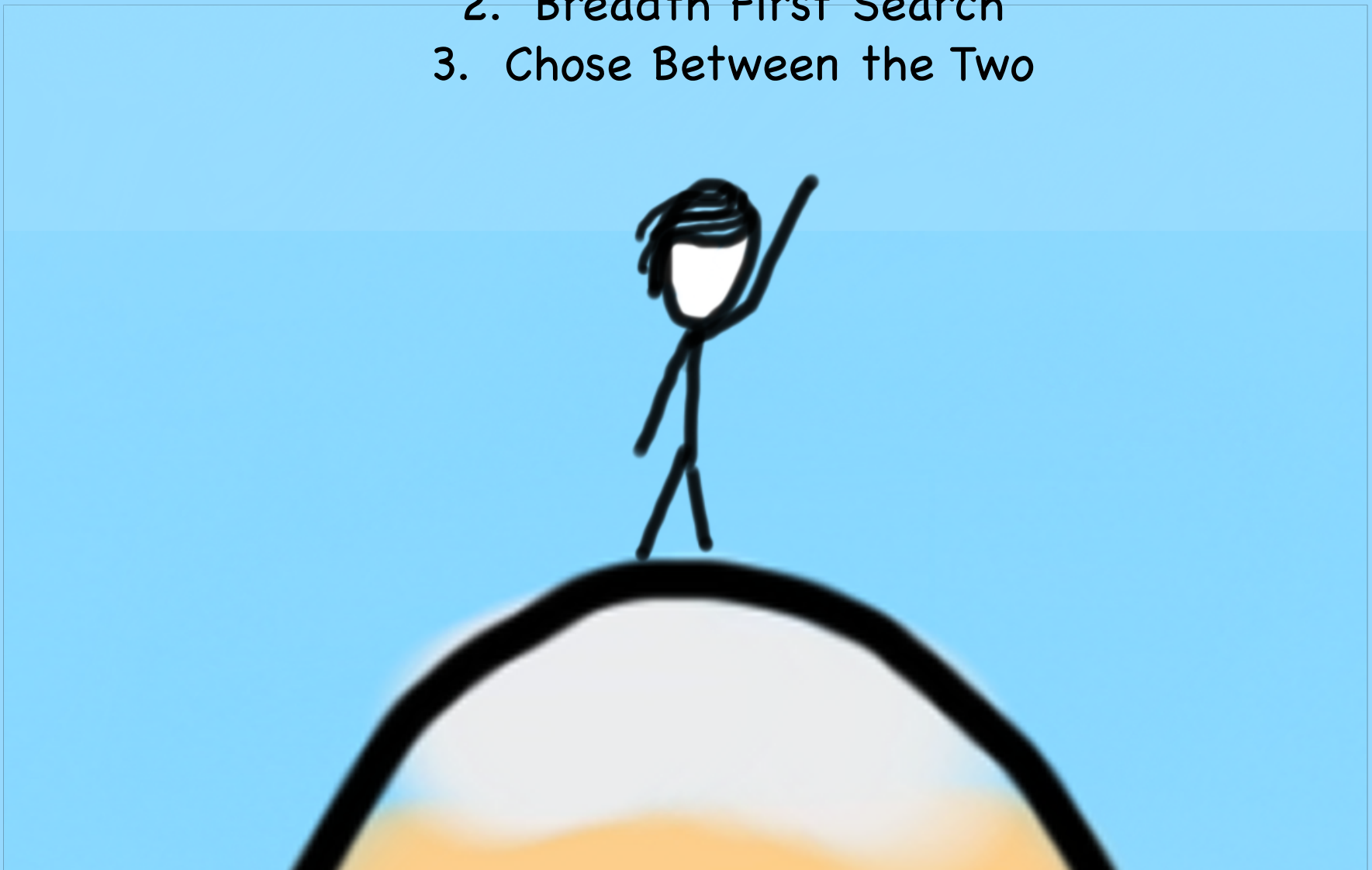
* Finish follows start

Who Do You Love?

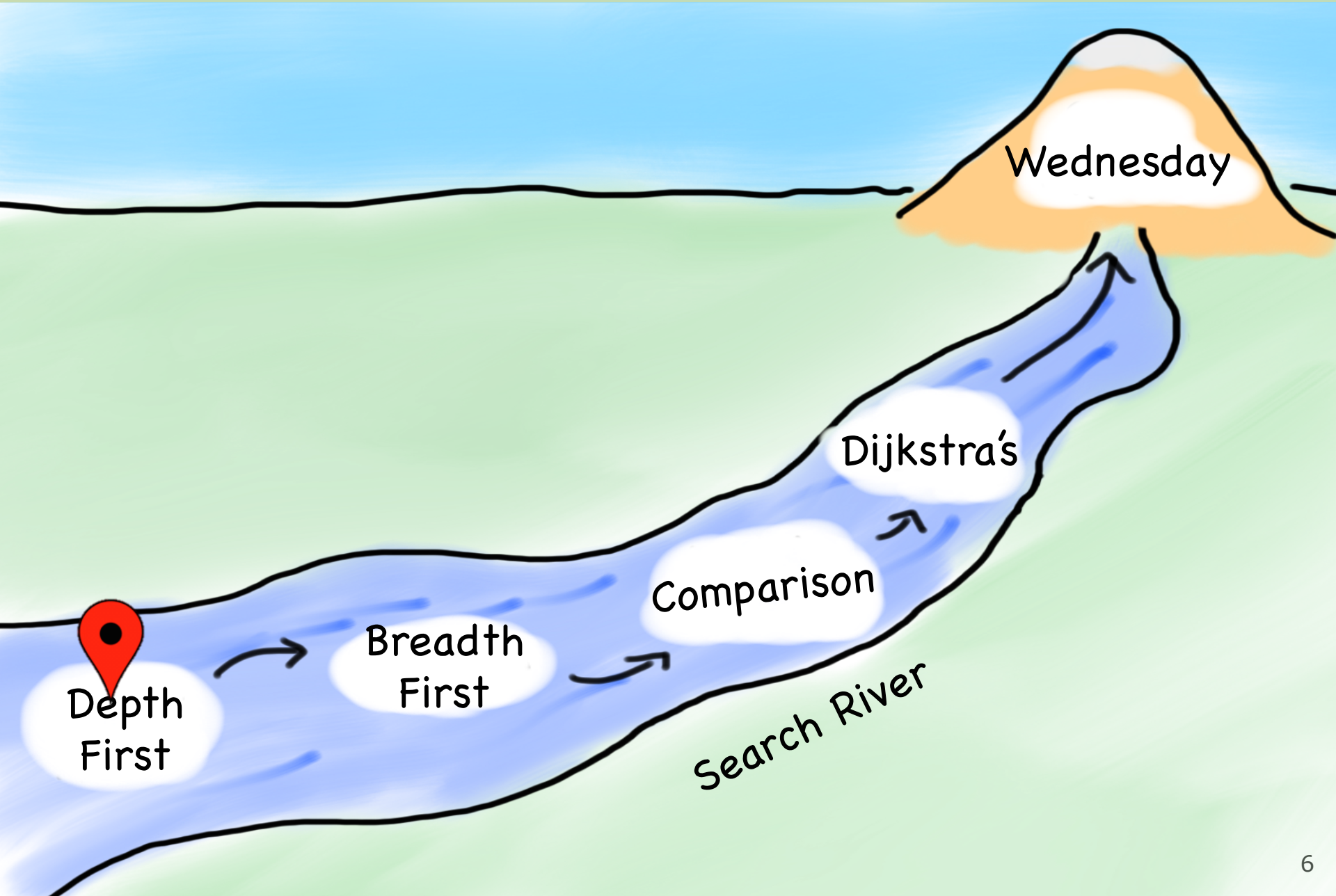


Today's Goal

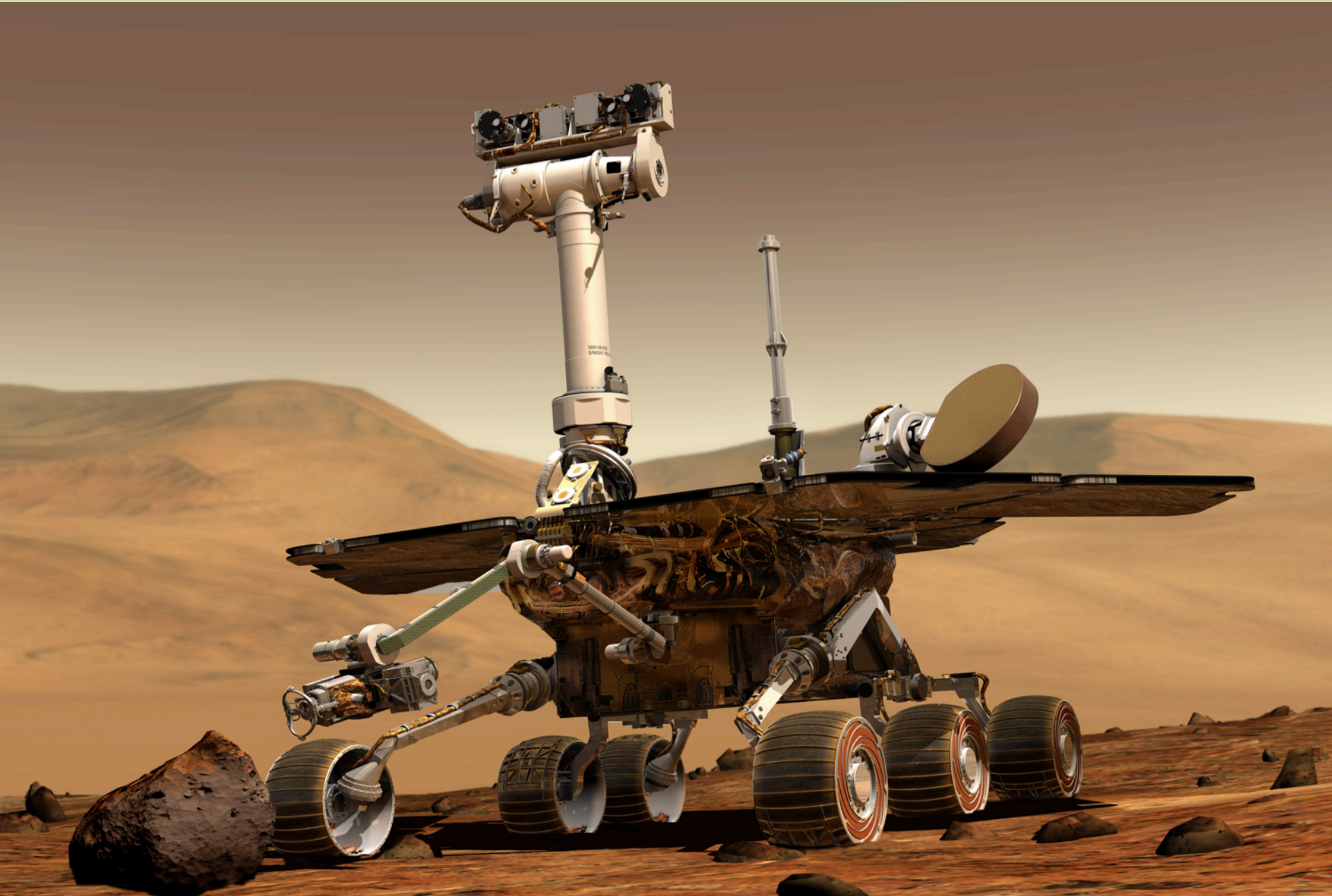
1. Depth First Search
2. Breadth First Search
3. Chose Between the Two



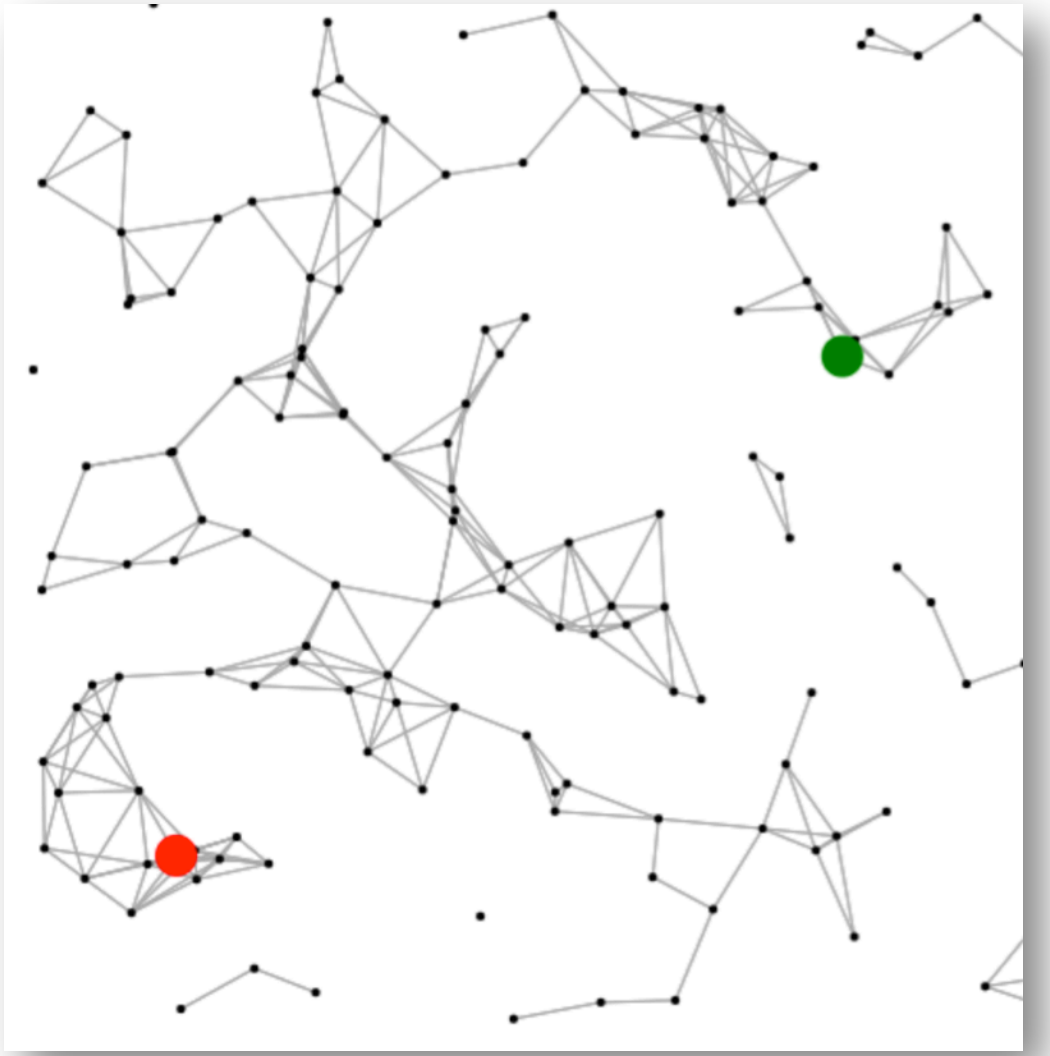
Today's Route



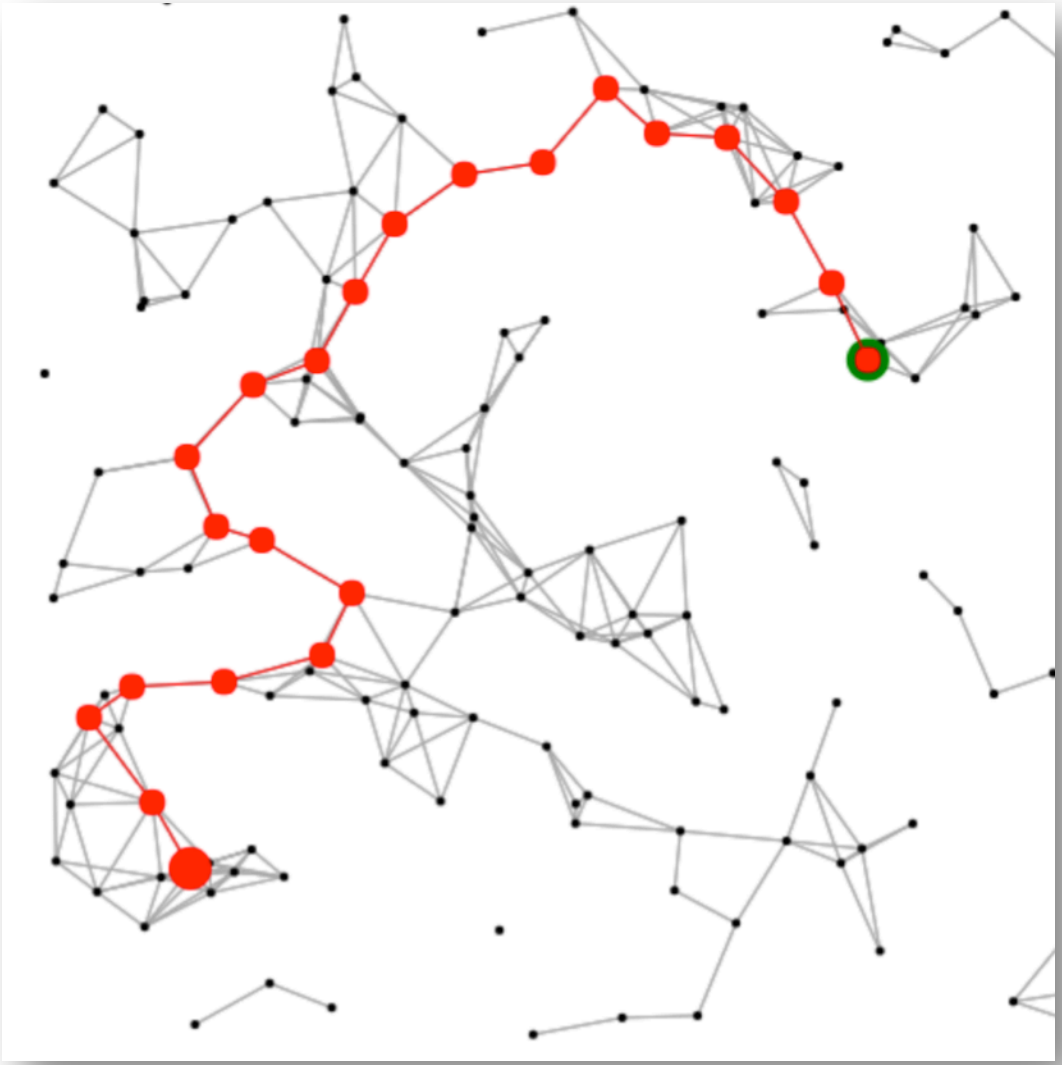
Robot Pathfinding



Graph Search



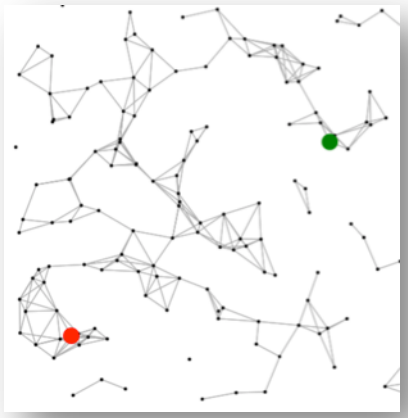
Graph Search

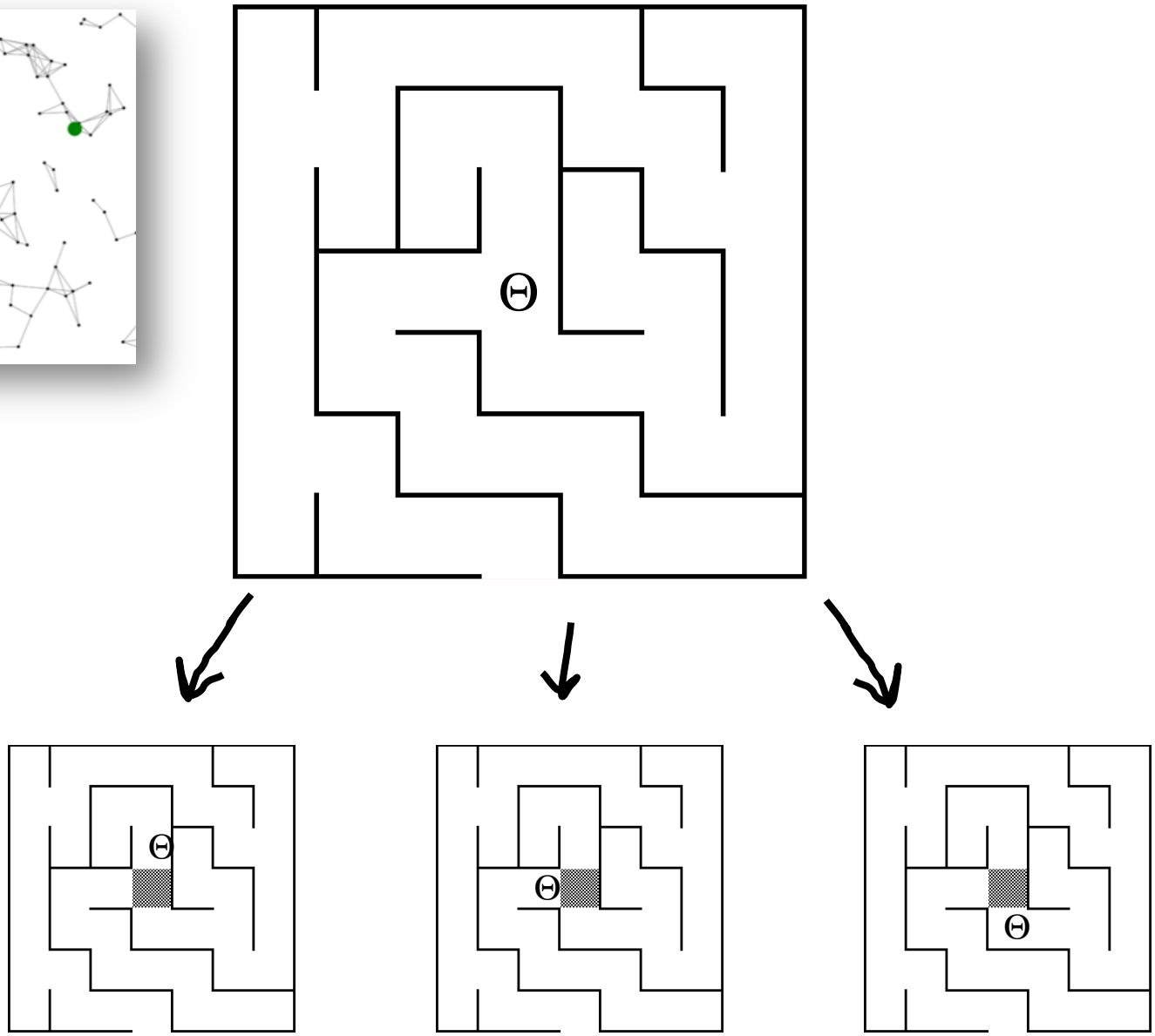


Search is Important

Old friend, Recursive Exploration

Maze Decision Tree





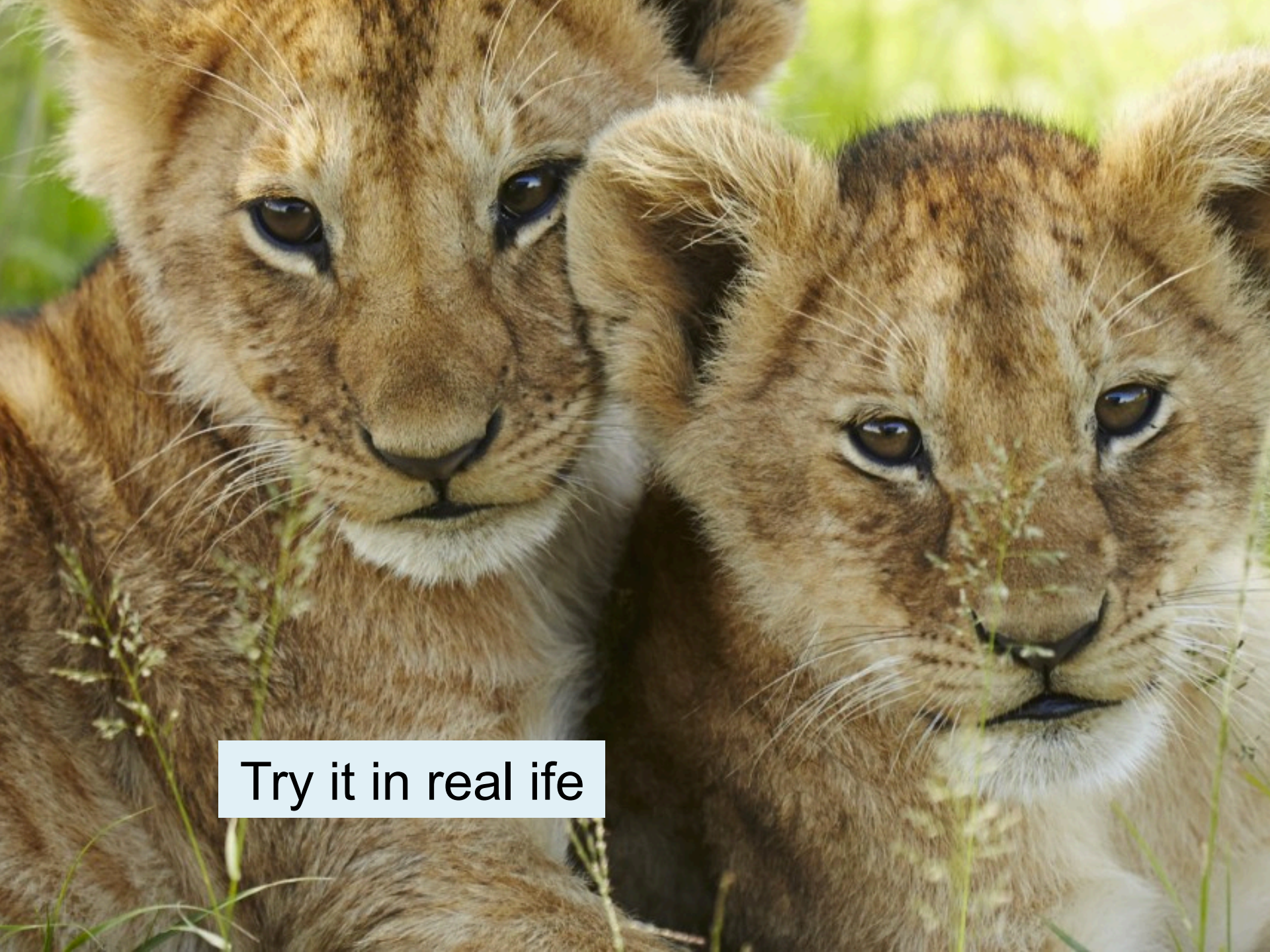
Recursive Exploration Solution

```
bool findPath(BasicGraph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

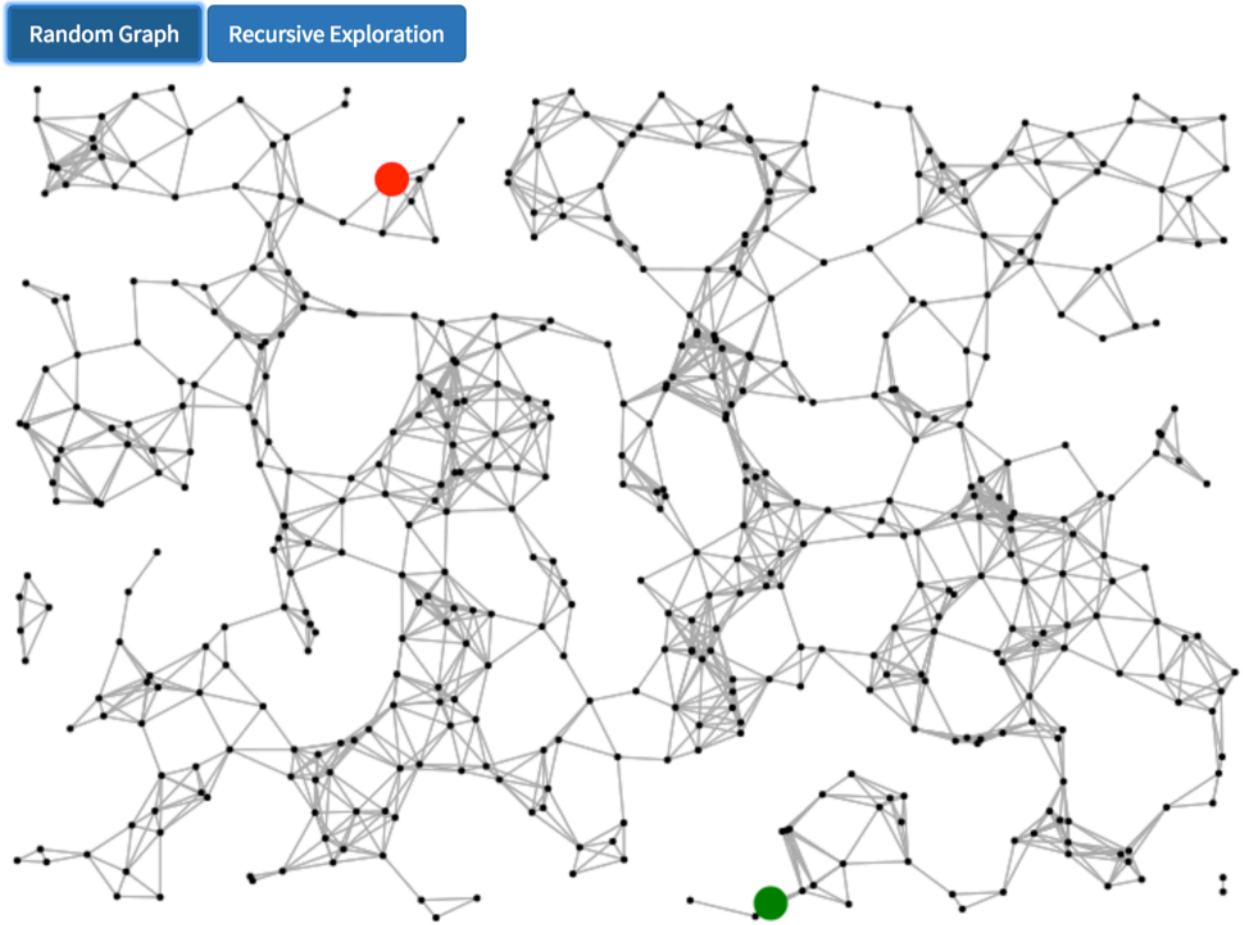
Recursive Exploration Solution

```
Vector<Vertex *> findPath(BasicGraph & g, Set<Vertex *> seen,
                          Vector<Vertex *> path, Vertex * goal) {
  Vertex * node = path.get(path.size() – 1);
  if(node == goal) return path;
  if(seen.contains(node)) return emptyPath();
  seen.add(node);
  for(Vertex * next : g->getNeighbors(node)) {
    Vector<Vertex *> newPath = path;
    newPath.add(next);
    if(!findPath(g, seen, newPath, goal).isEmpty()) {
      return newPath;
    }
  }
  return emptyPath();
}
```

Can be changed to return the path


Try it in real ife

Lets Try it Out!

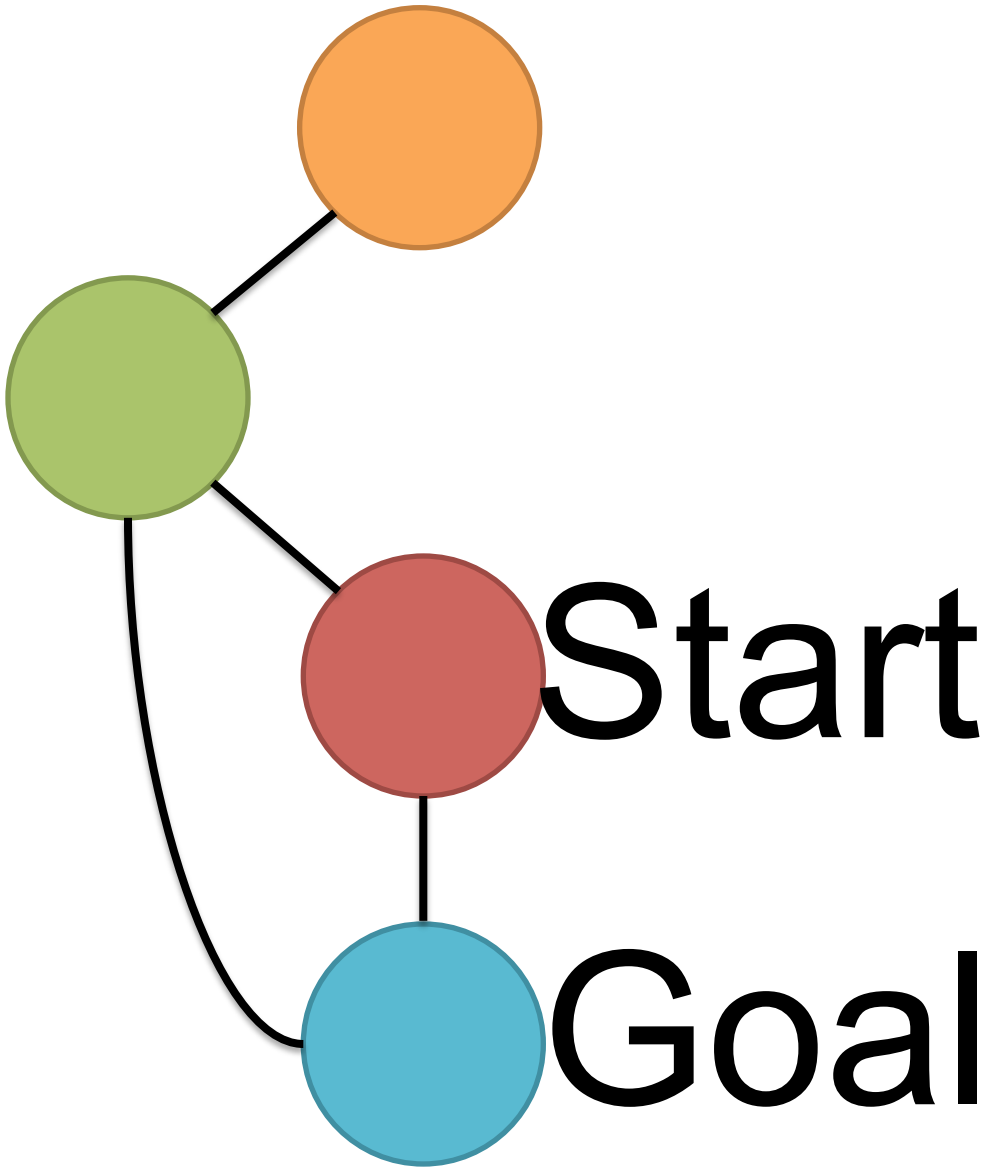


cs106b.stanford.edu/handouts/search.html

#Rekt

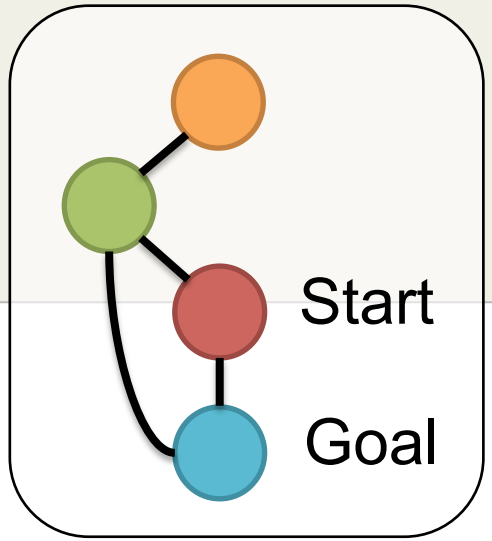
What happened?

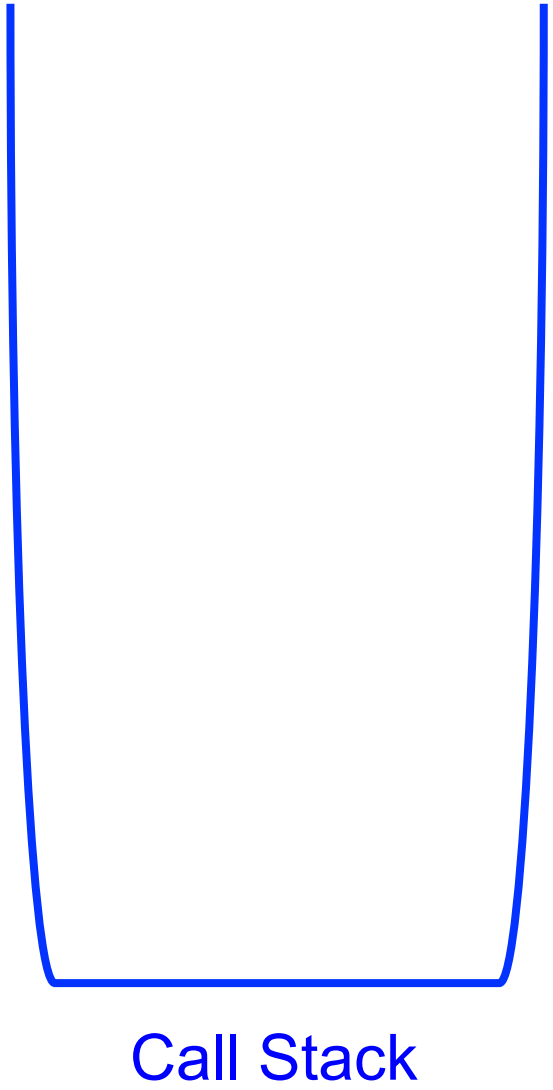
Most Simple Graph



Recursive Exploration Solution

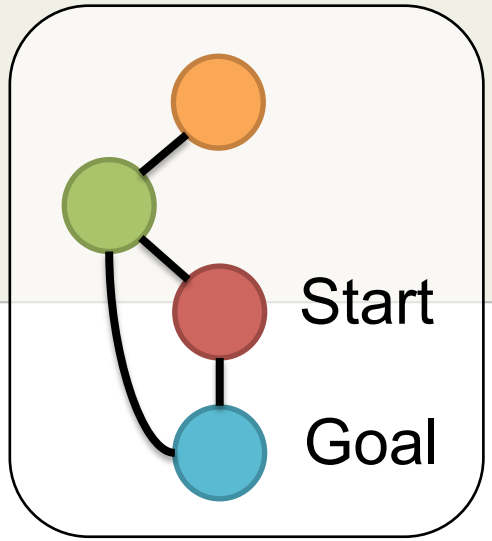
```
bool findPath(BasicGraph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

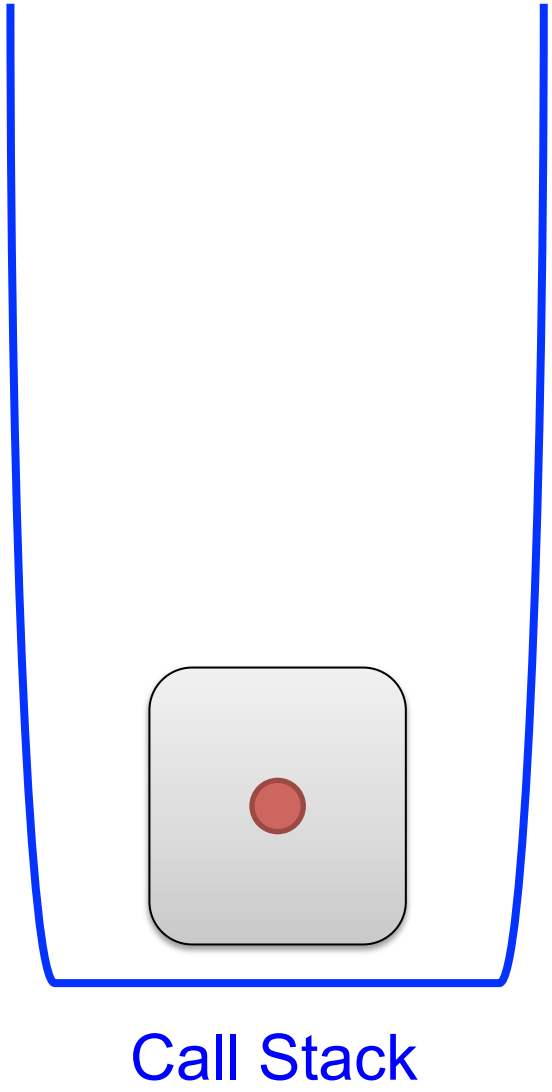




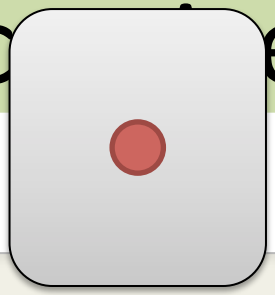
Recursive Exploration Solution

```
bool findPath(BasicGraph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

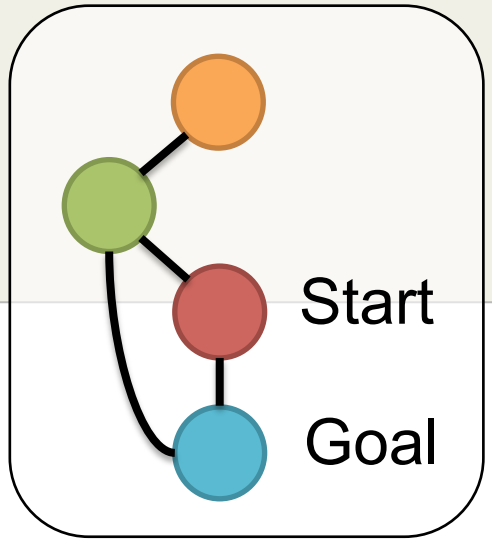


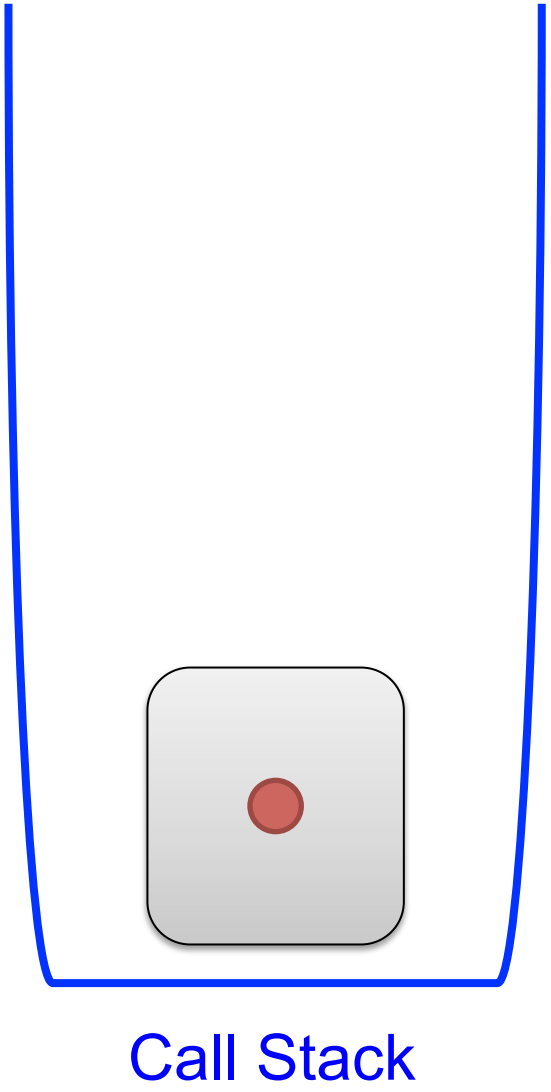


Recursive Exploration Solution

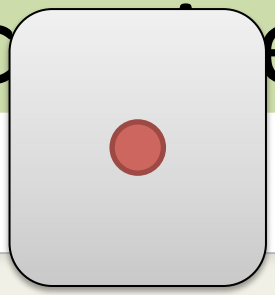


```
bool findPath(BasicGraph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

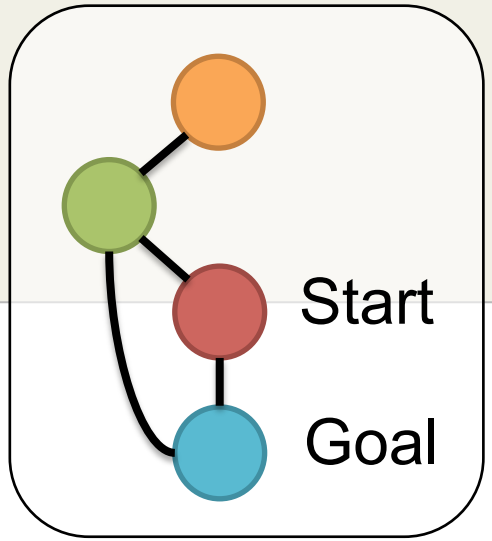


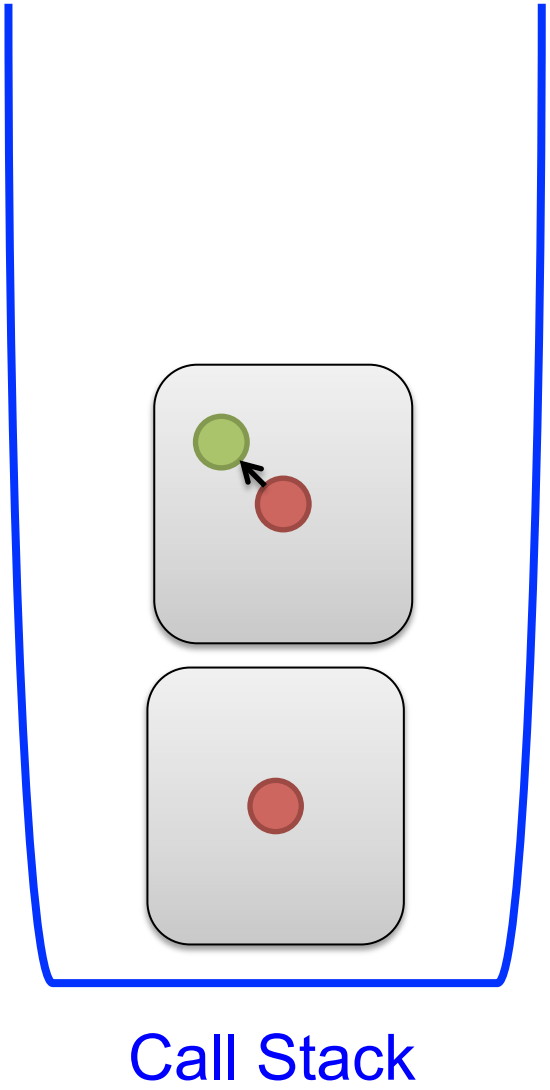


Recursive Exploration Solution

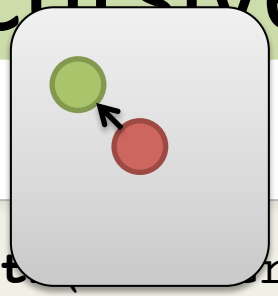


```
bool findPath(BasicGraph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

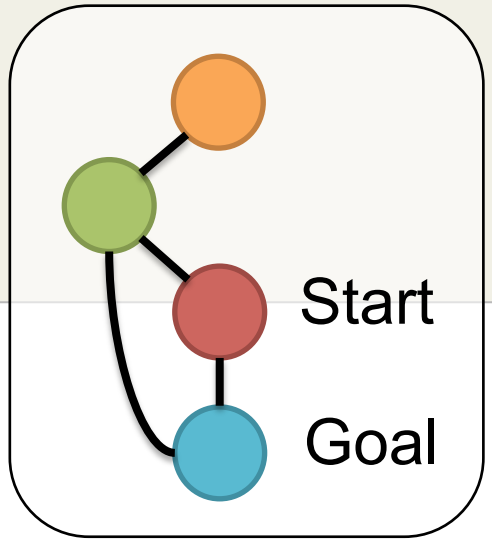


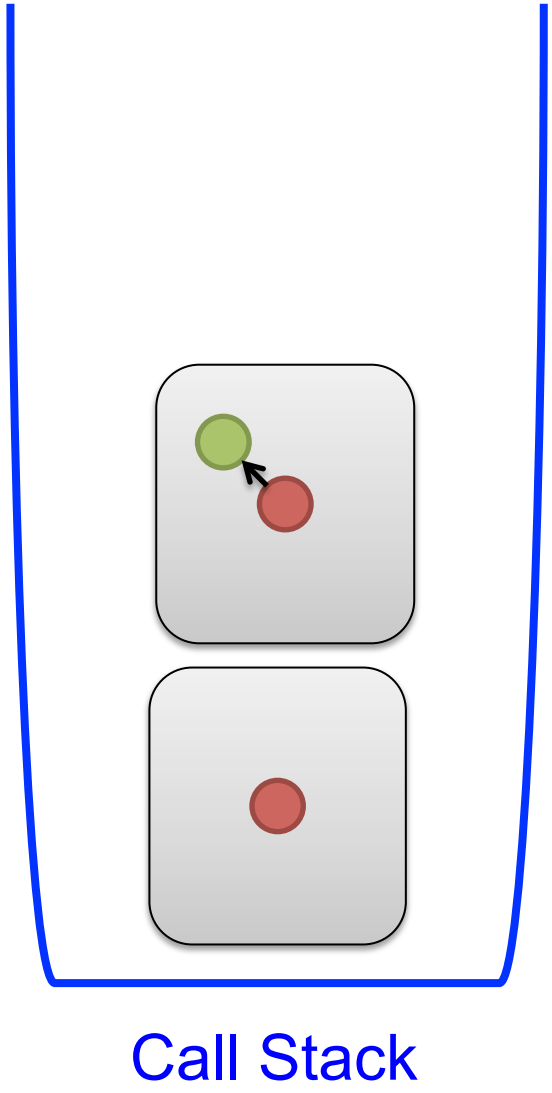


Recursive Exploration Solution

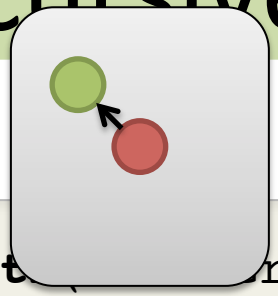


```
bool findPath(Graph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

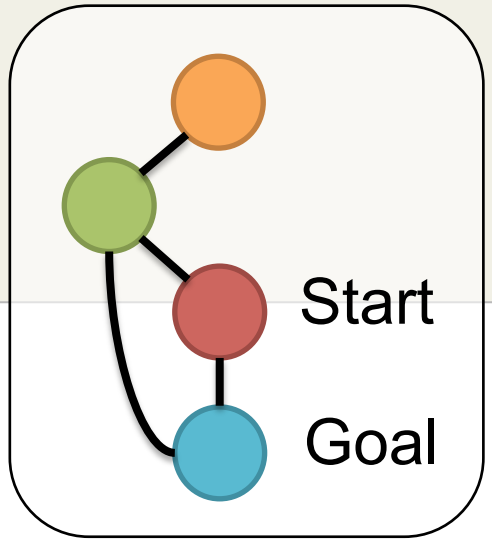


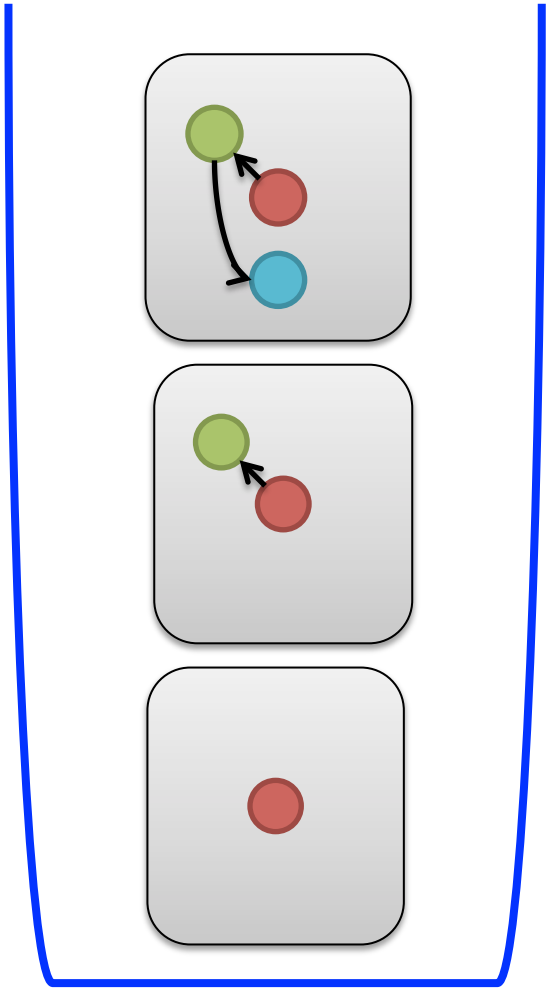


Recursive Exploration Solution

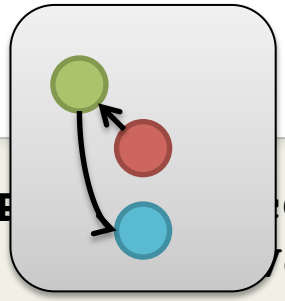


```
bool findPath(Graph & g,
              Set<Vertex *> seen,
              Vertex * node,
              Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```

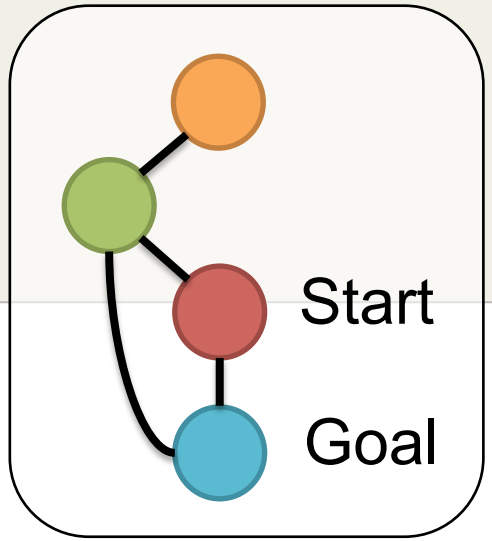


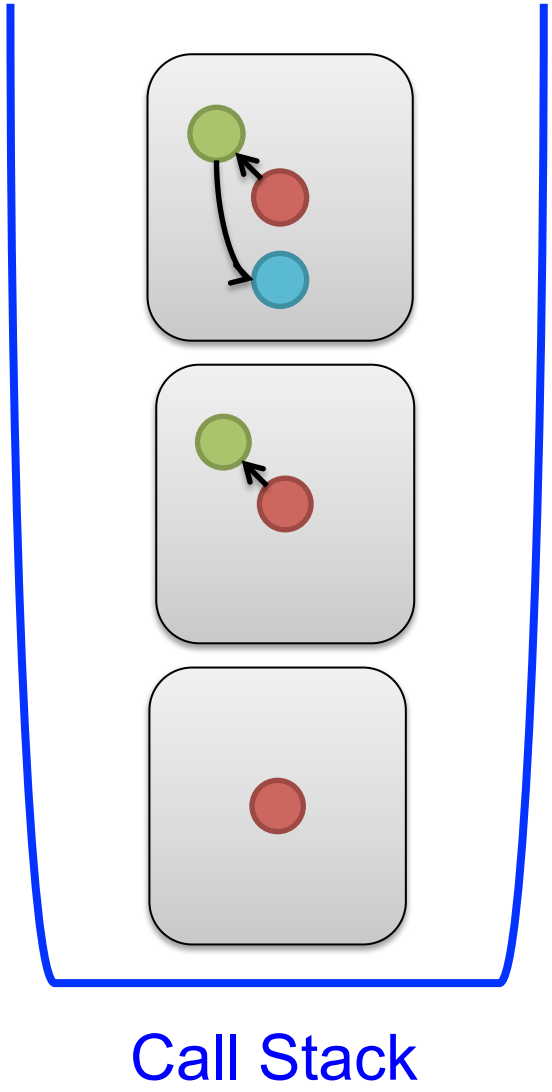


Recursive Exploration Solution



```
bool findP        Graph & g,
                  Vertex *> seen,
                  Vertex * node,
                  Vertex * goal) {
  if(node == goal) return true;
  if(seen.contains(node)) return false;
  seen.add(node);
  for(Vertex* next:g->getNeighbors(node)){
    if(findPath(g, seen, next, goal)) {
      return true;
    }
  }
  return false;
}
```





How to think about search

How to Think About Search

How is the todo list stored?

Call Stack!

How to Think About Search

This code is equivalent*:

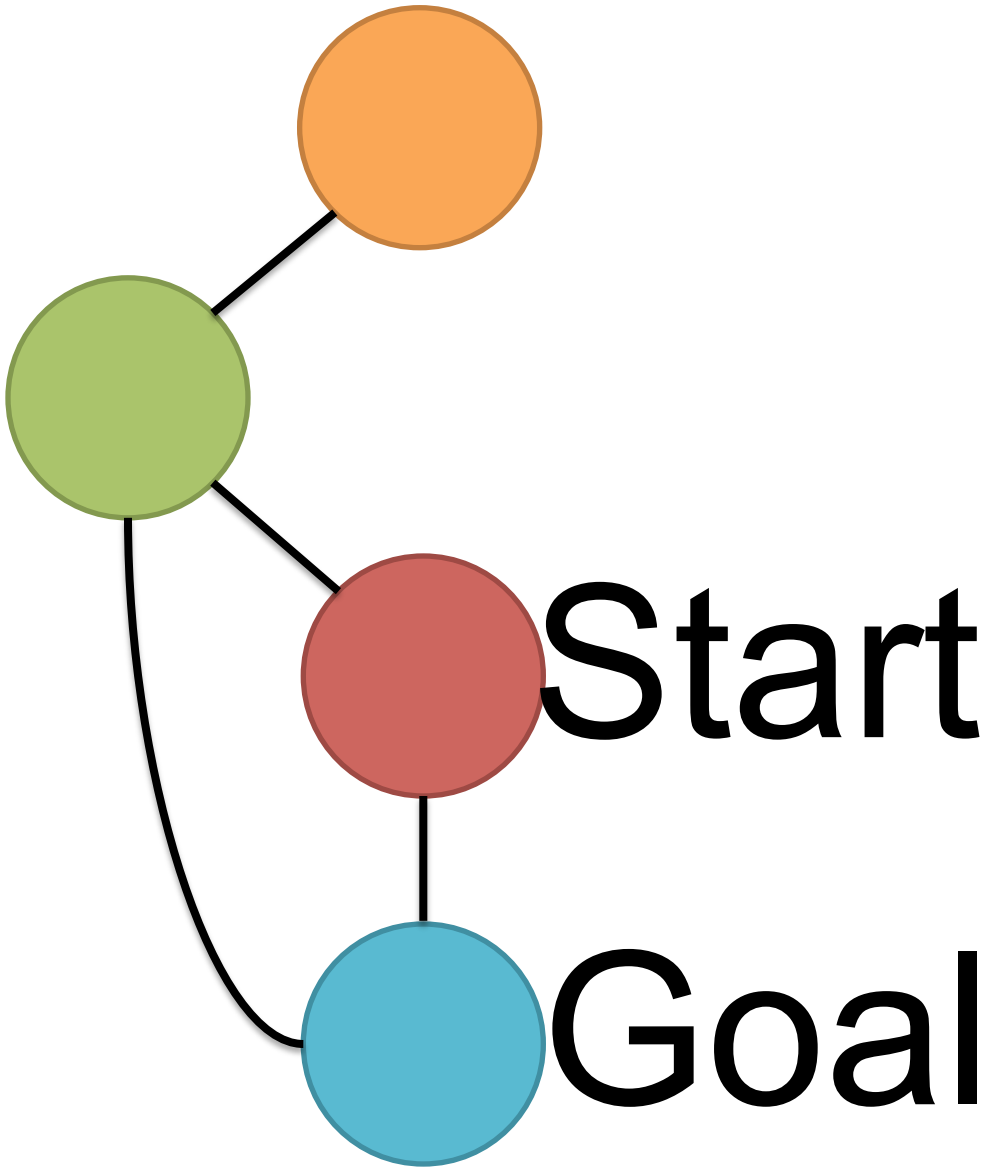
*actually it uses more memory

Depth First Search

```
stack = Stack()
stack.push(startNode)
seen = Set();
visited = Set([])
while !stack.isEmpty():
    currState = stack.pop()
    if(currState is goal) return currState;
    if(seen contains currState) continue;
    seen.add(currState);
    if not currState in visited:
        for nextState in getNextStates(currState)
          stack.push(nextState)
        }
    }
}
```

This code is equivalent:

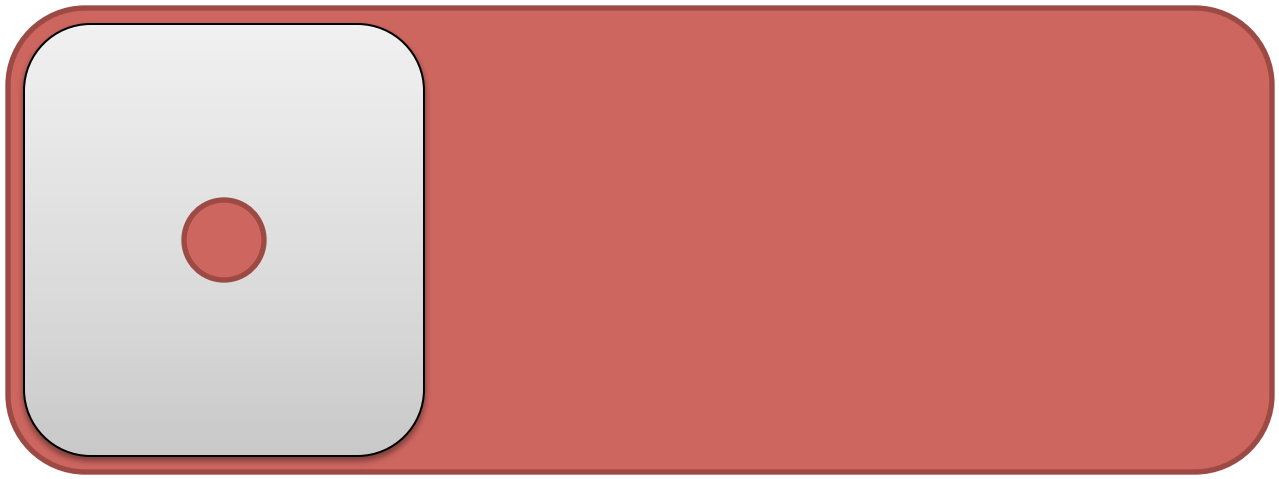
Most Simple Graph



Depth First Search

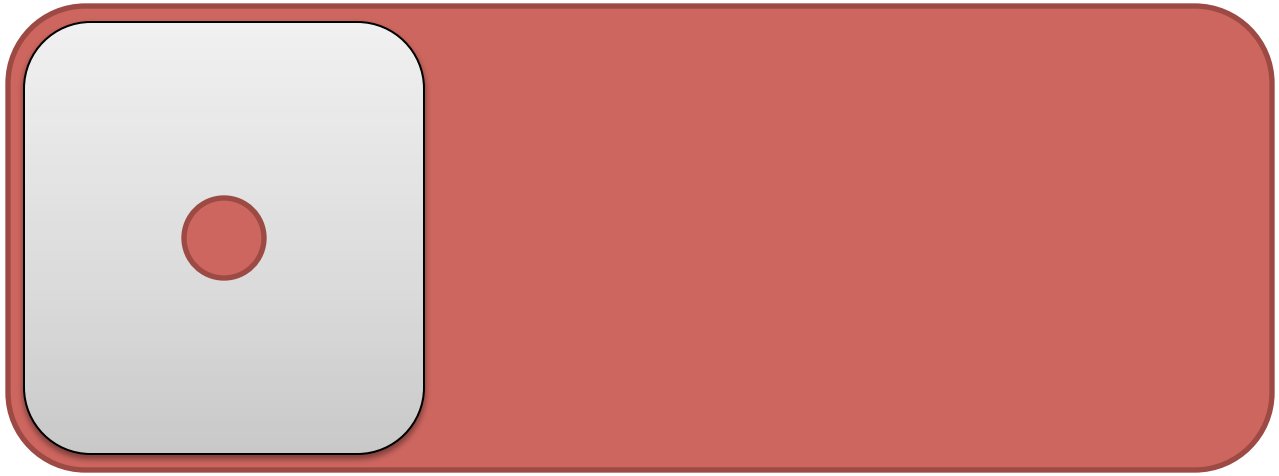
Create an empty stack

Depth First Search



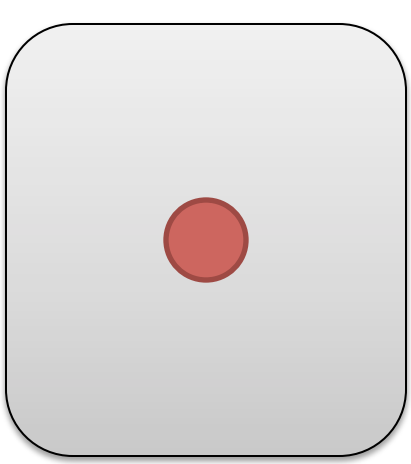
Push a path with just the start node

Depth First Search



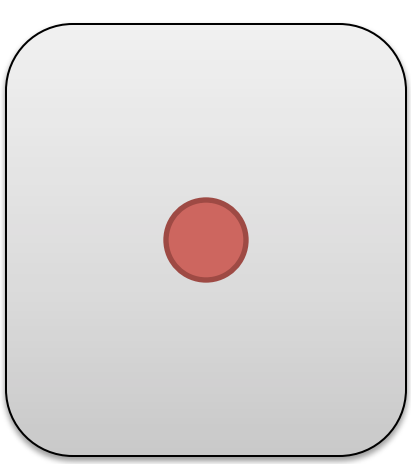
Enter the while loop

Depth First Search



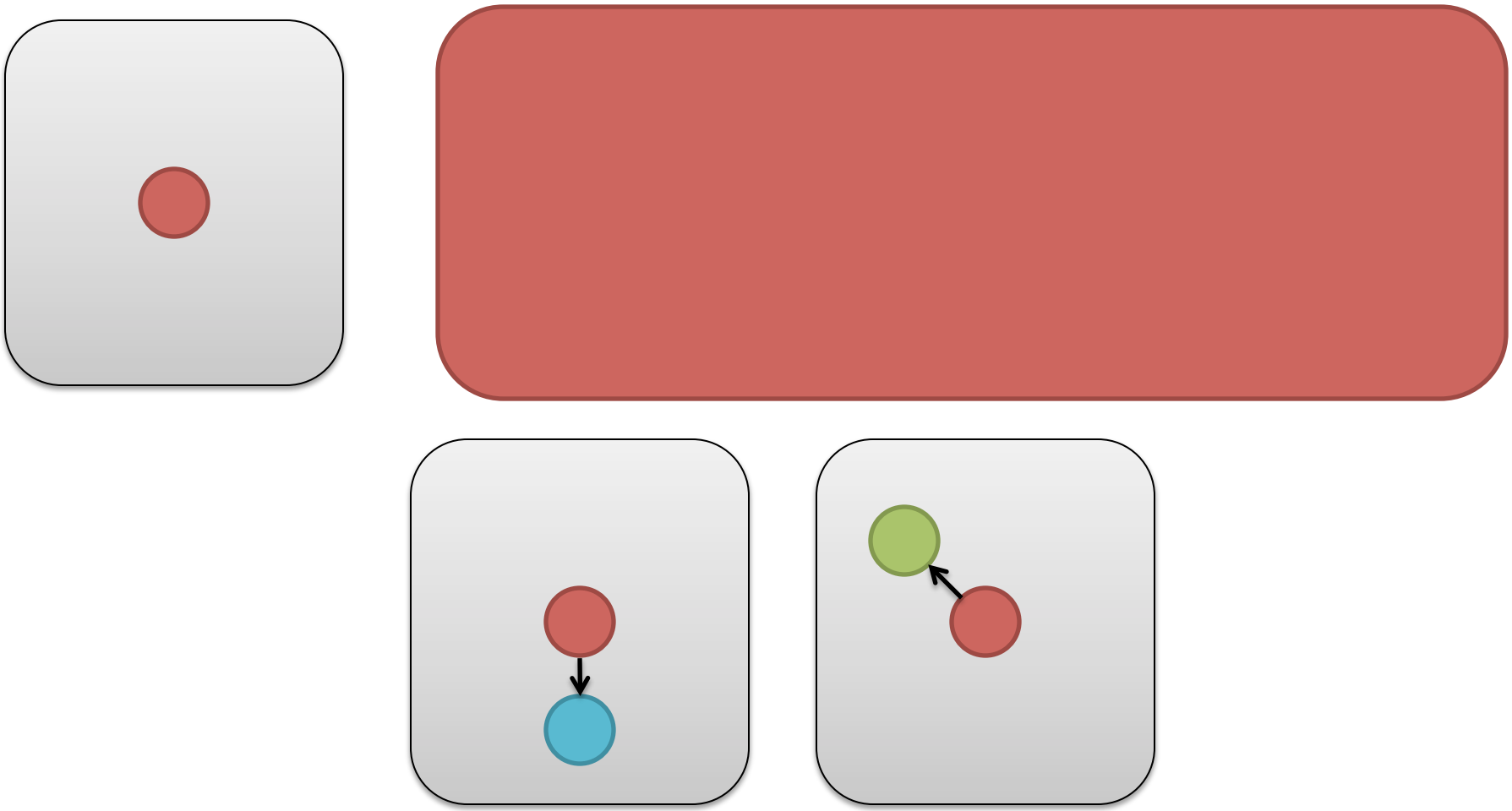
Pop a path

Depth First Search



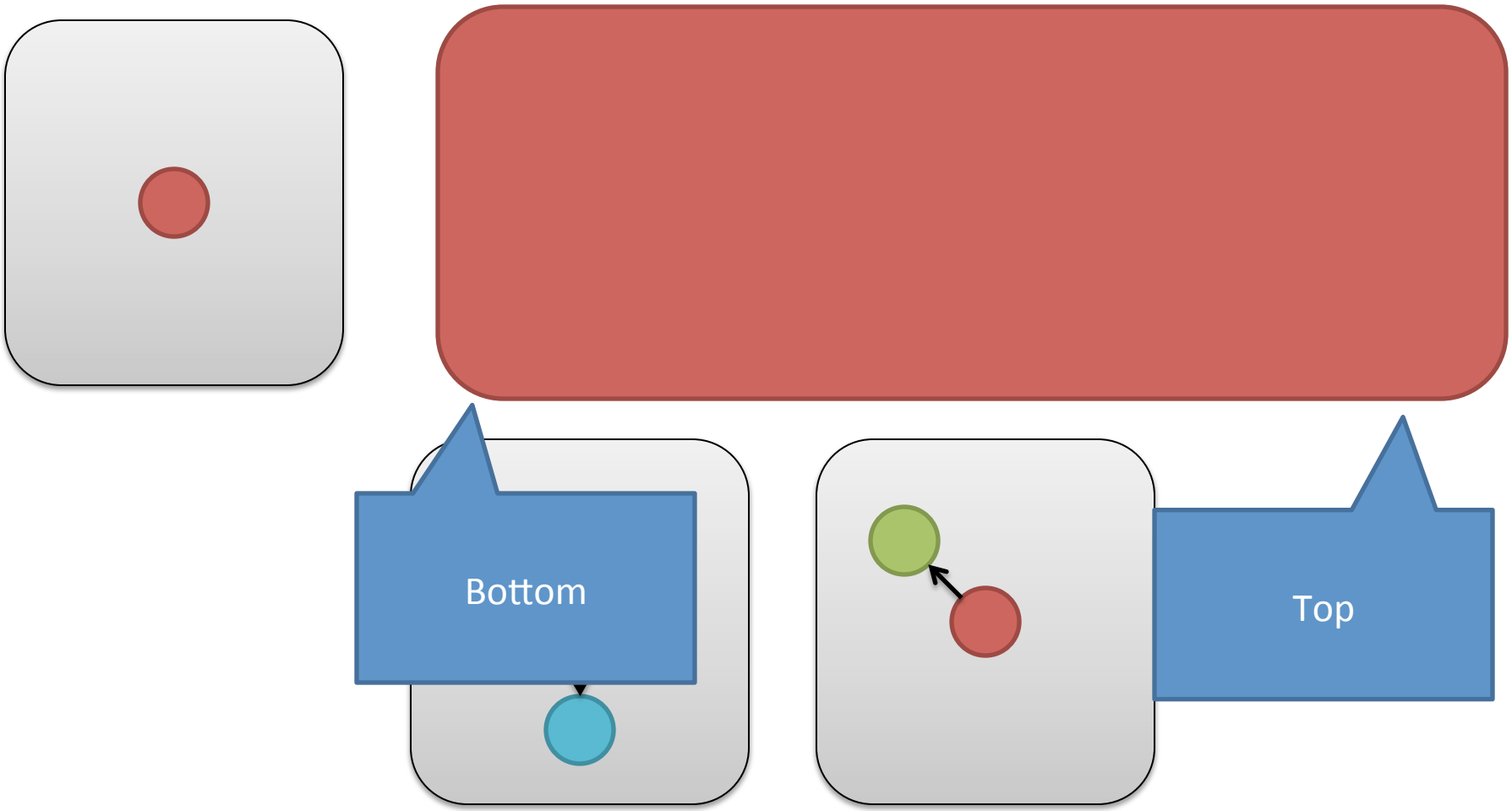
Check if its been visited or if it's the goal

Depth First Search



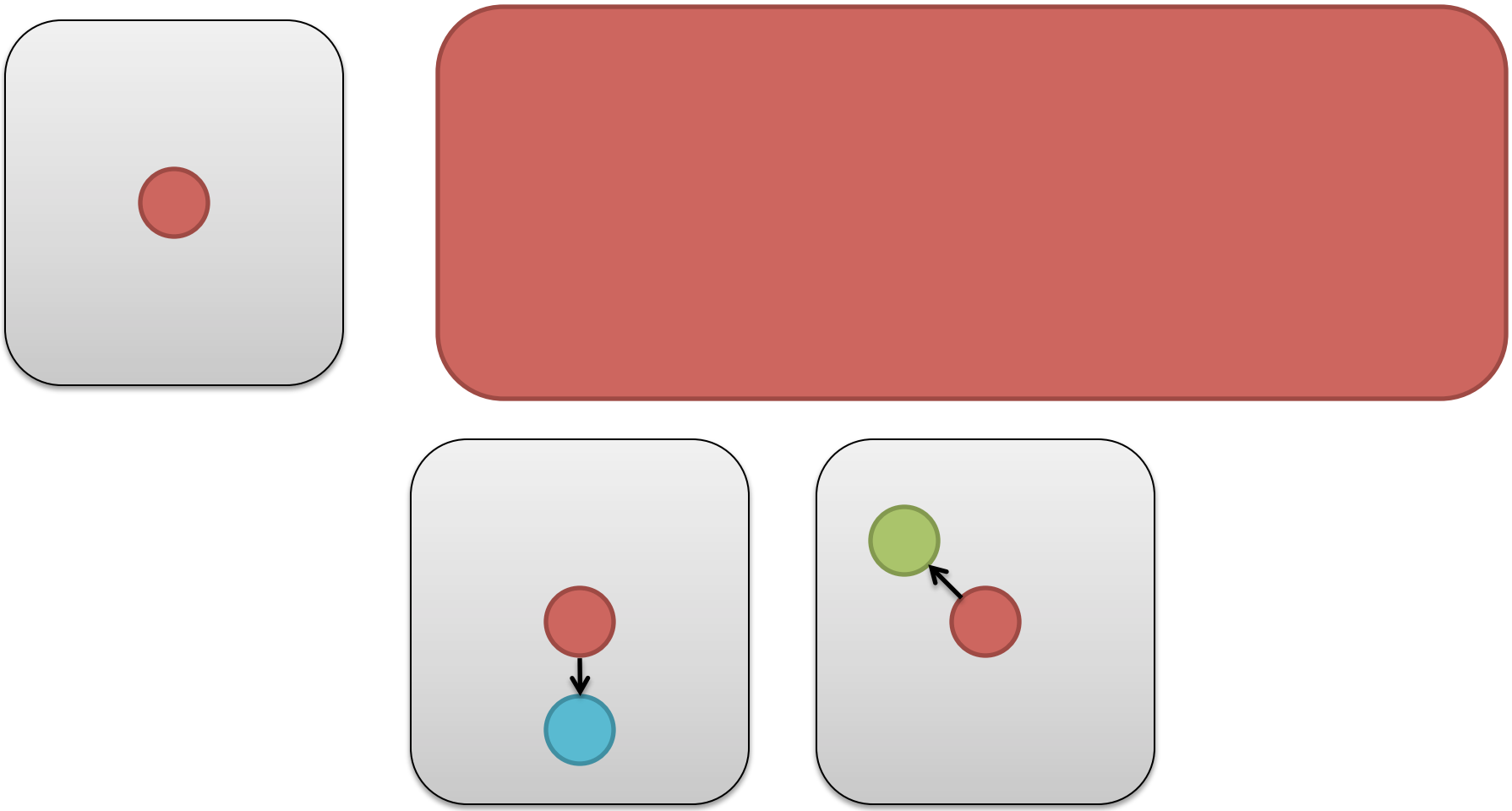
Create a path for each of its neighbors

Depth First Search



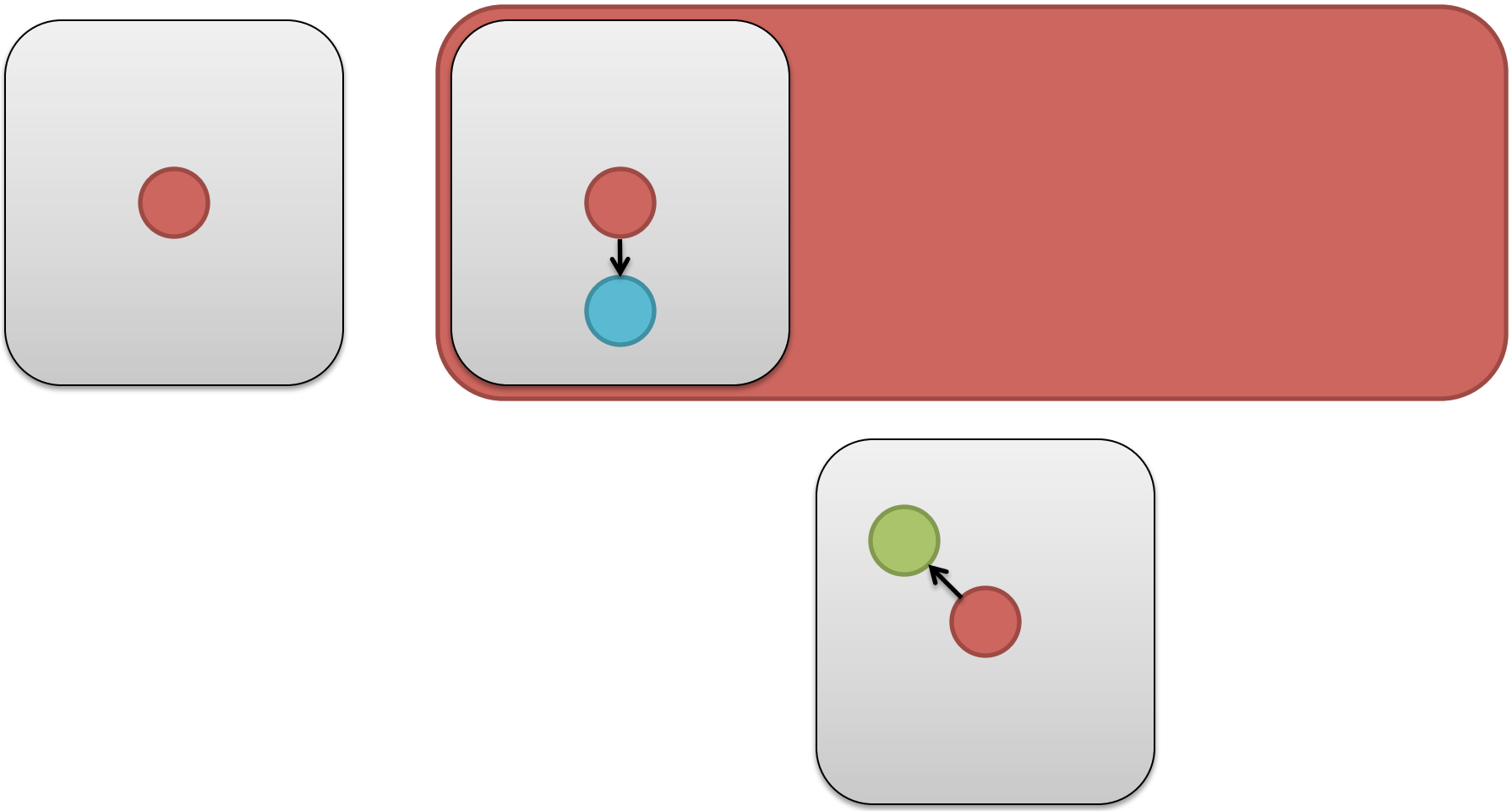
Create a path for each of its neighbors

Depth First Search



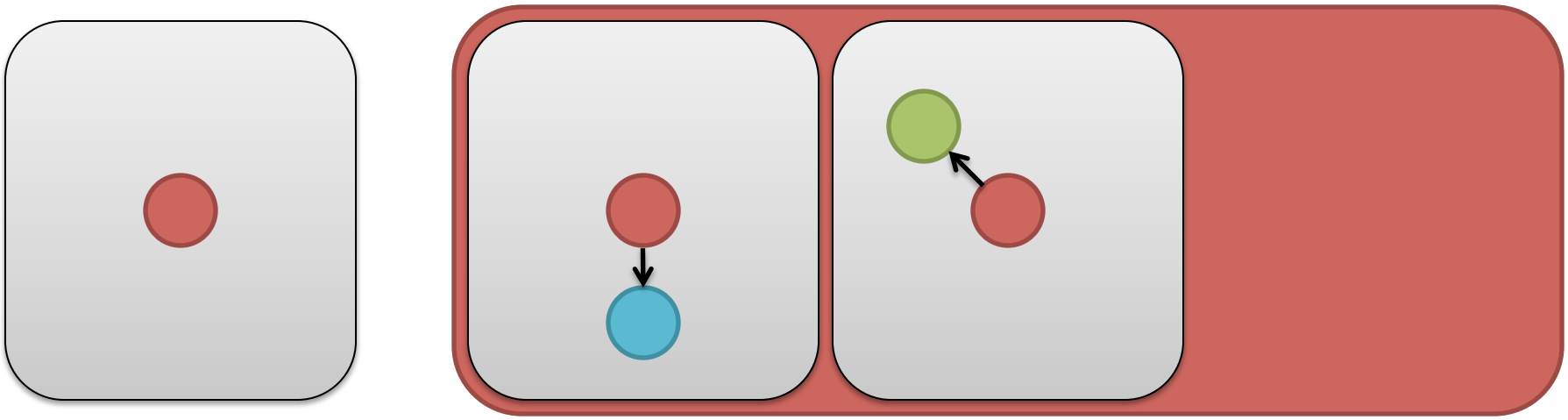
Create a path for each of its neighbors

Depth First Search



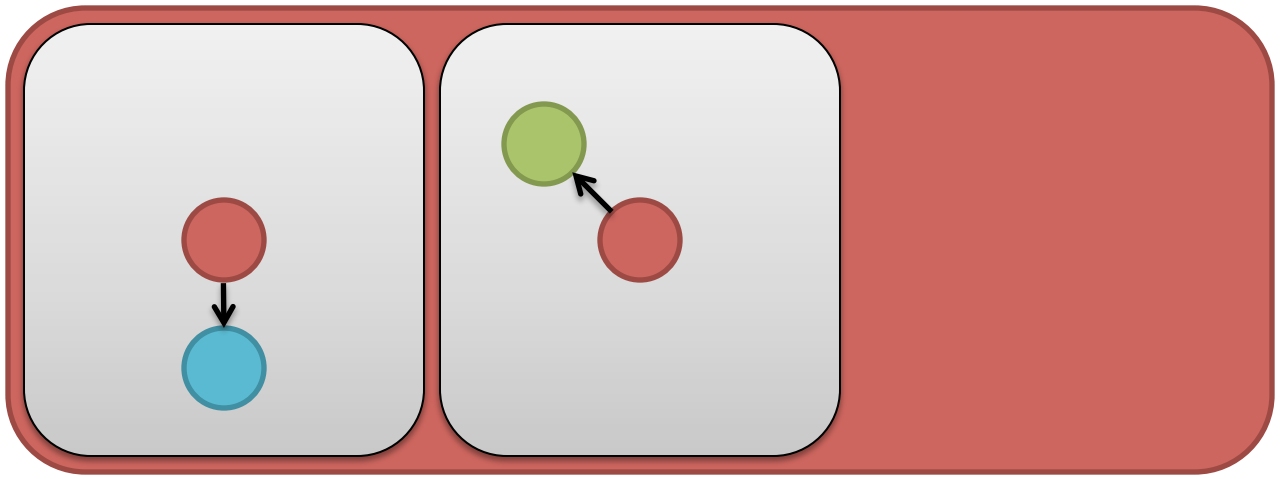
Push the new paths

Depth First Search



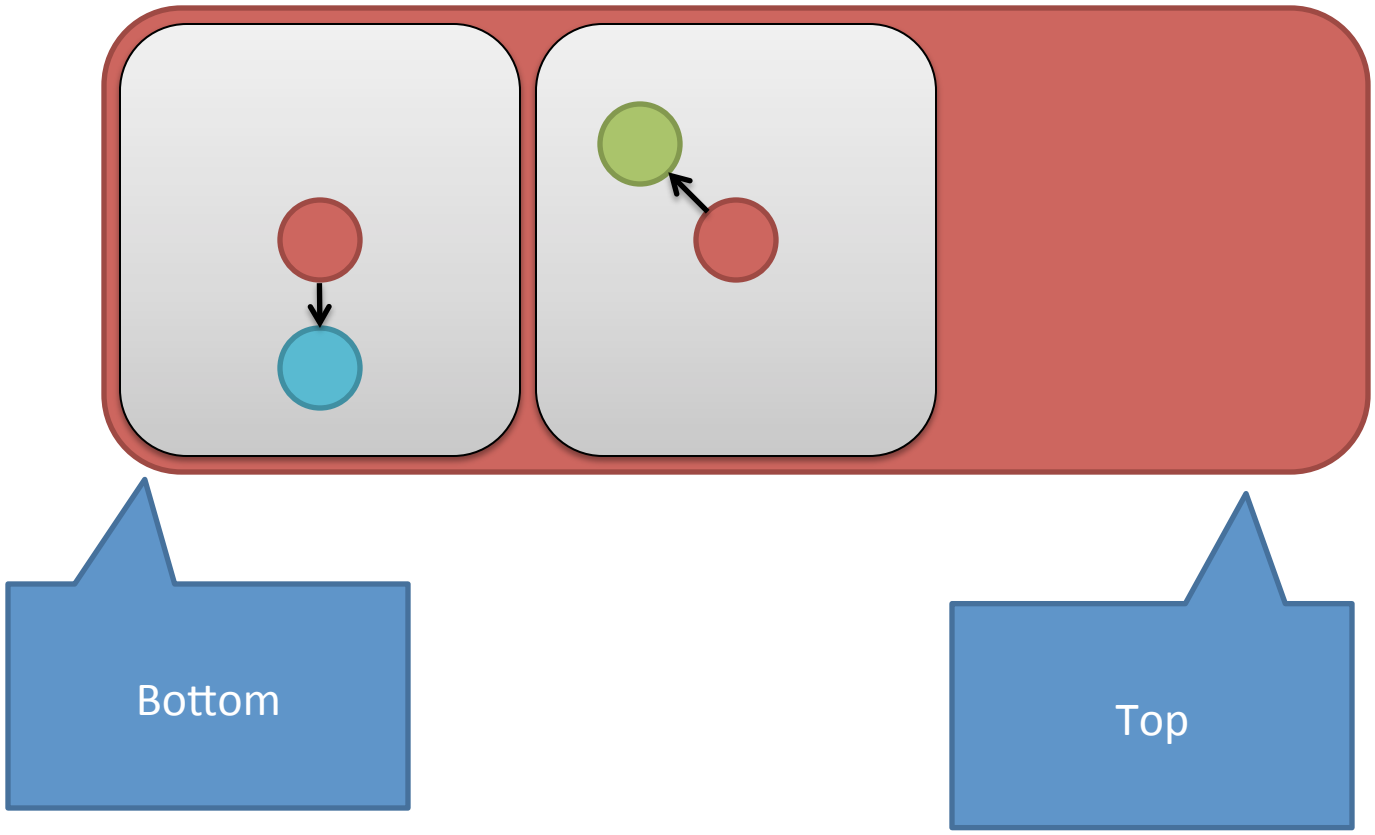
Push the new paths

Depth First Search



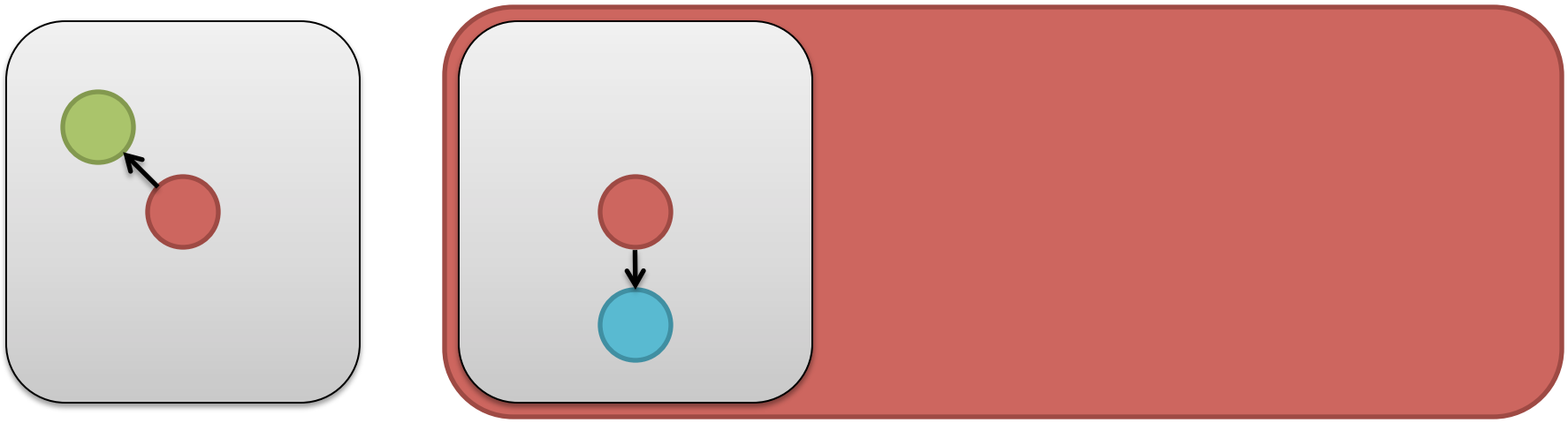
Back to the start of the while loop

Depth First Search



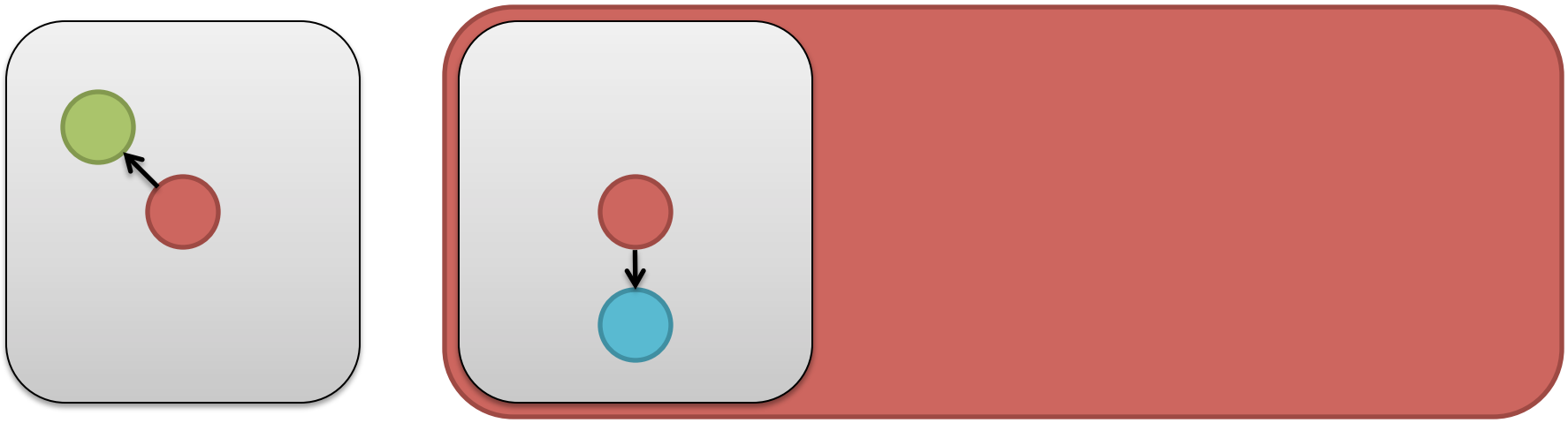
Back to the start of the while loop

Depth First Search



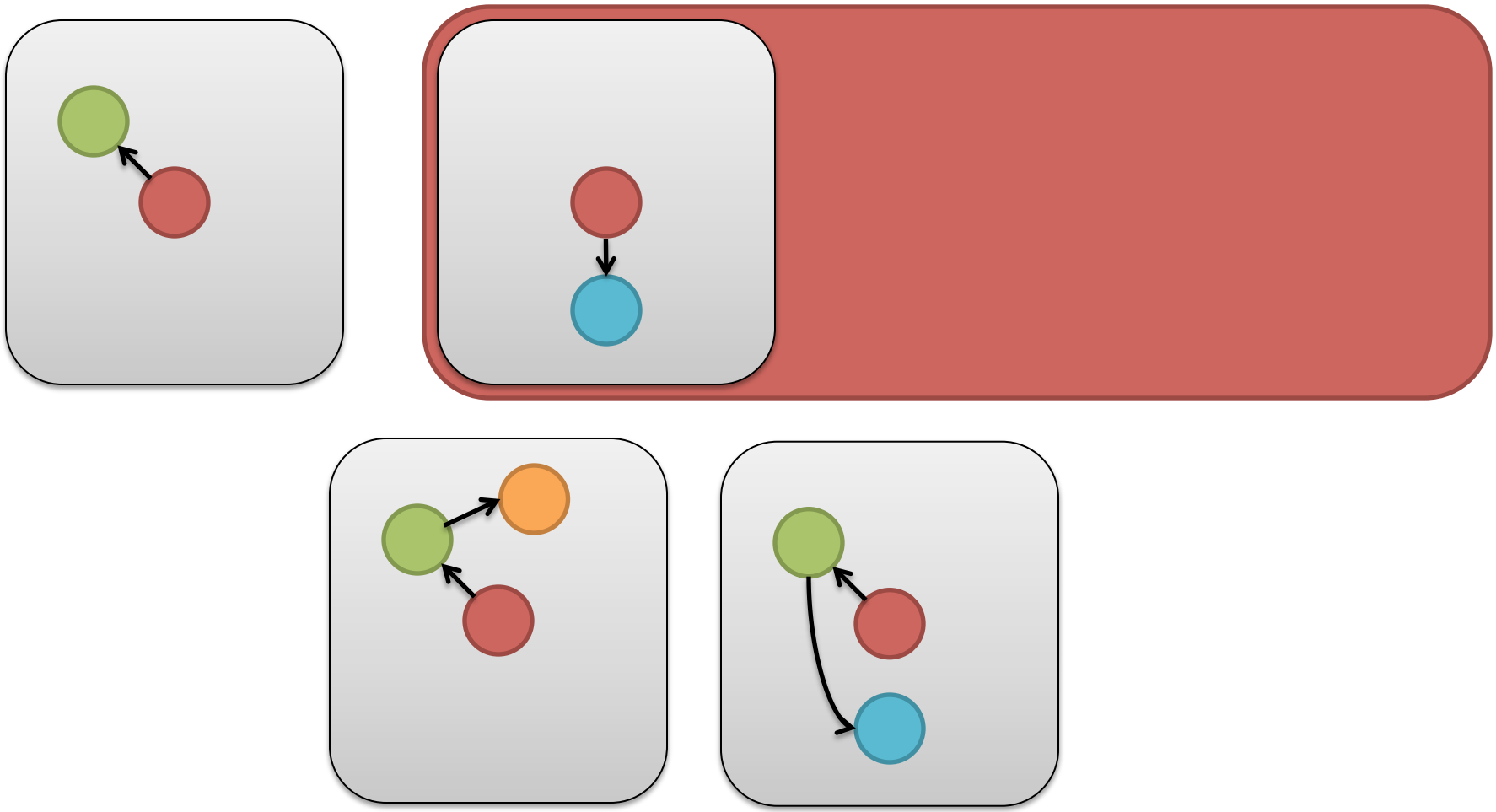
Pop a path

Depth First Search



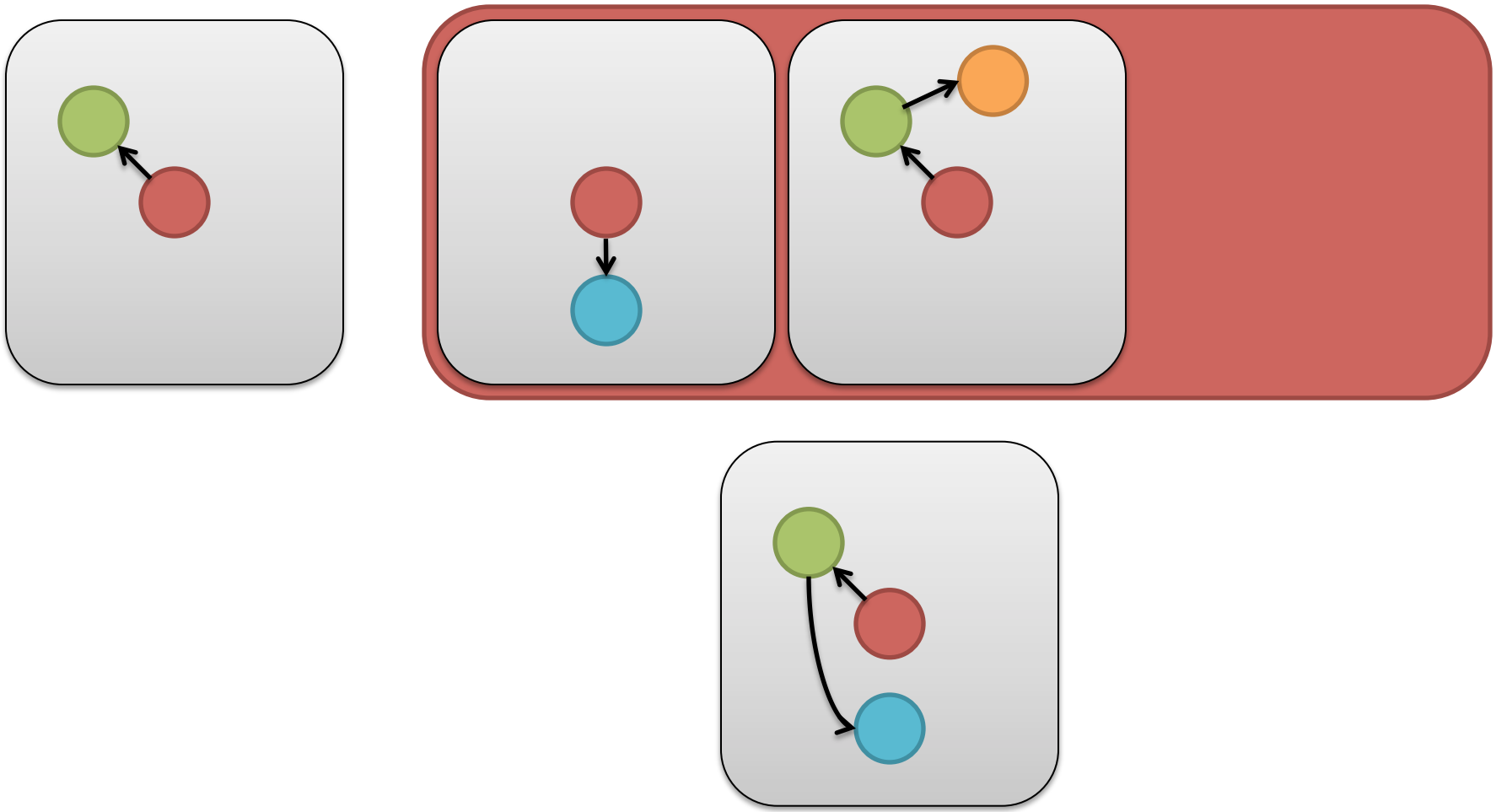
Check if the last node is a goal or if its been visited

Depth First Search



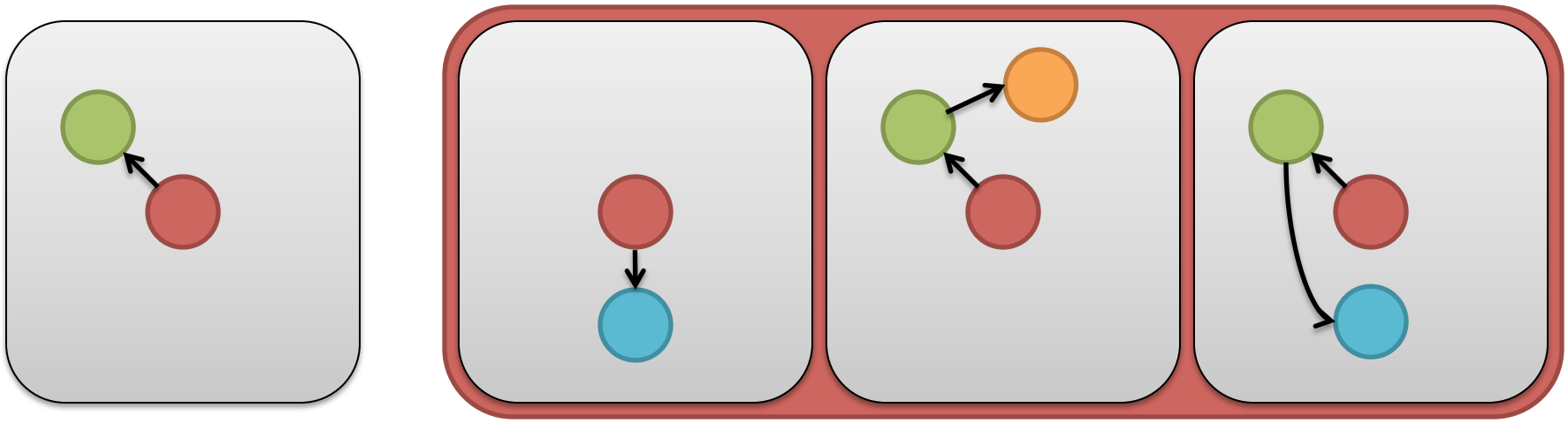
Create a new path for each neighbor of the last node

Depth First Search



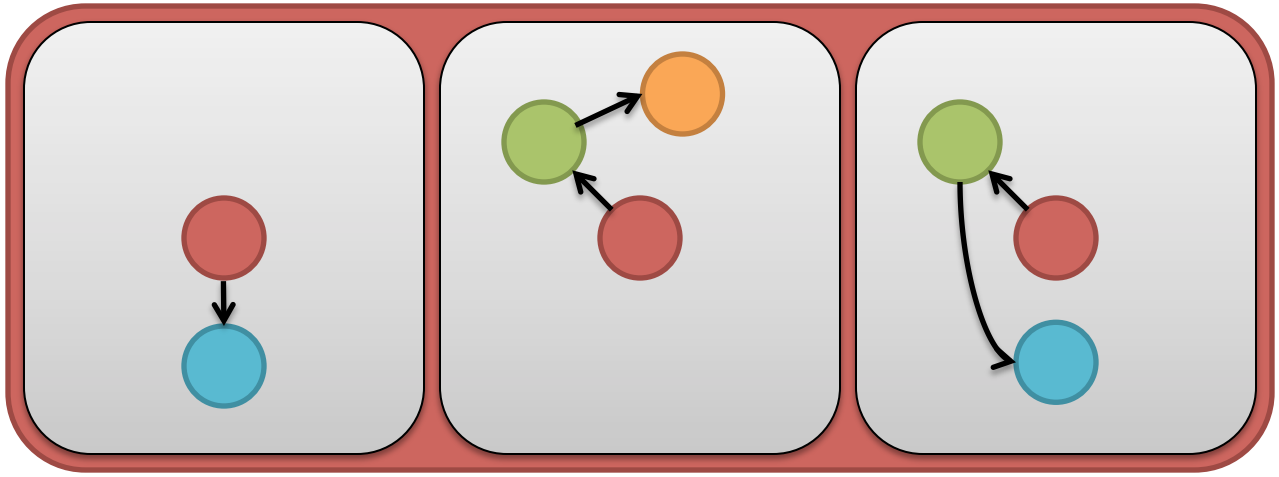
Push the new paths

Depth First Search



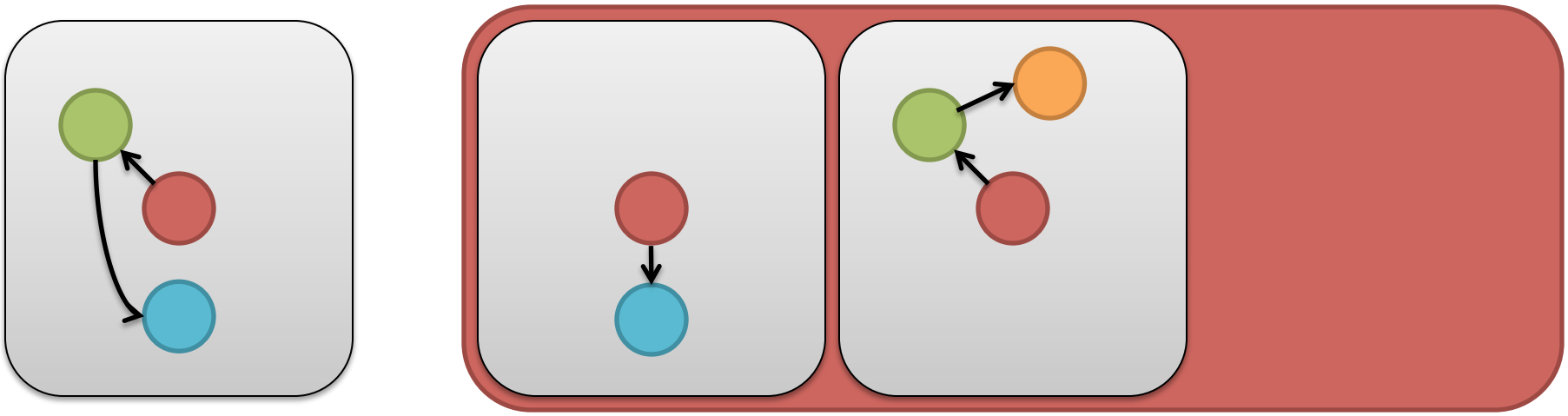
Push the new paths

Depth First Search



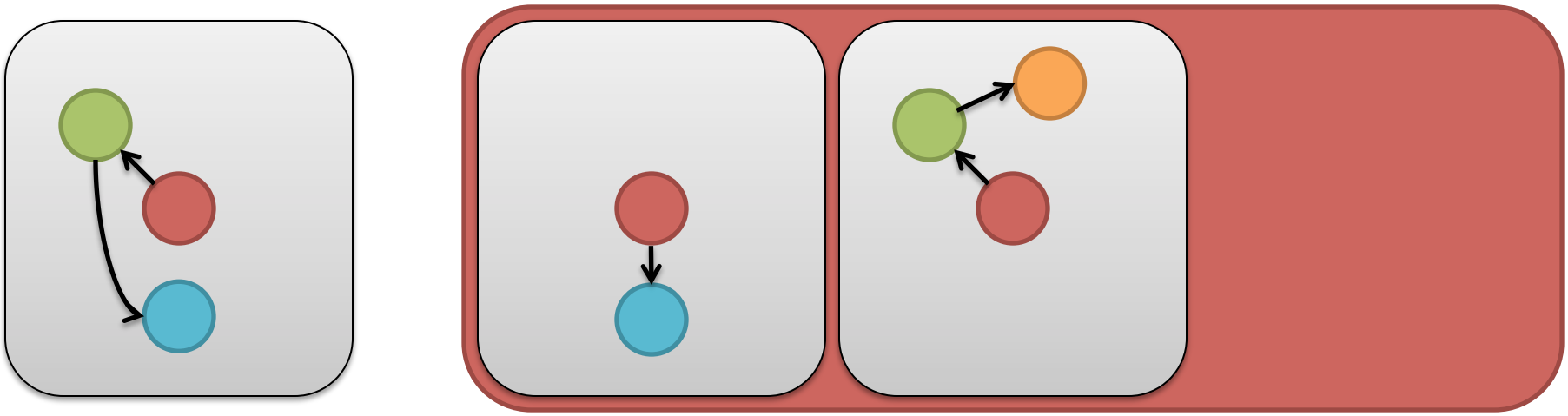
Back to the start of the while loop

Depth First Search



Pop a path

Depth First Search



Check if the last node is a goal or if its been visited


Try it live

#pwned

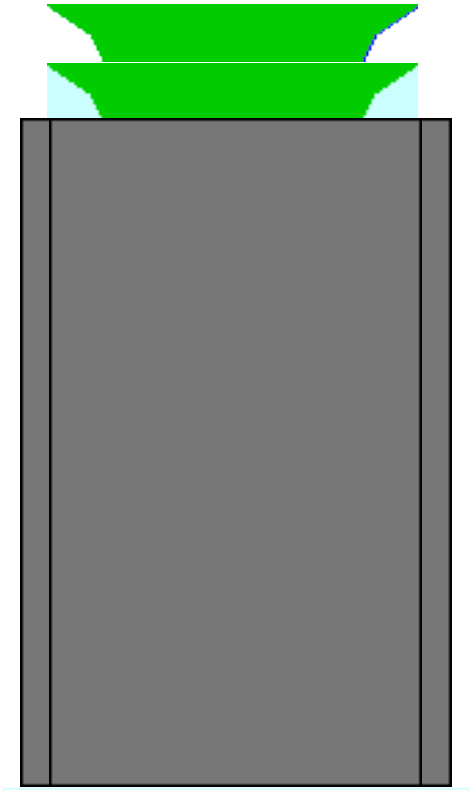

amazon.co.jp
amazon.co.jp

Stacks are LIFO

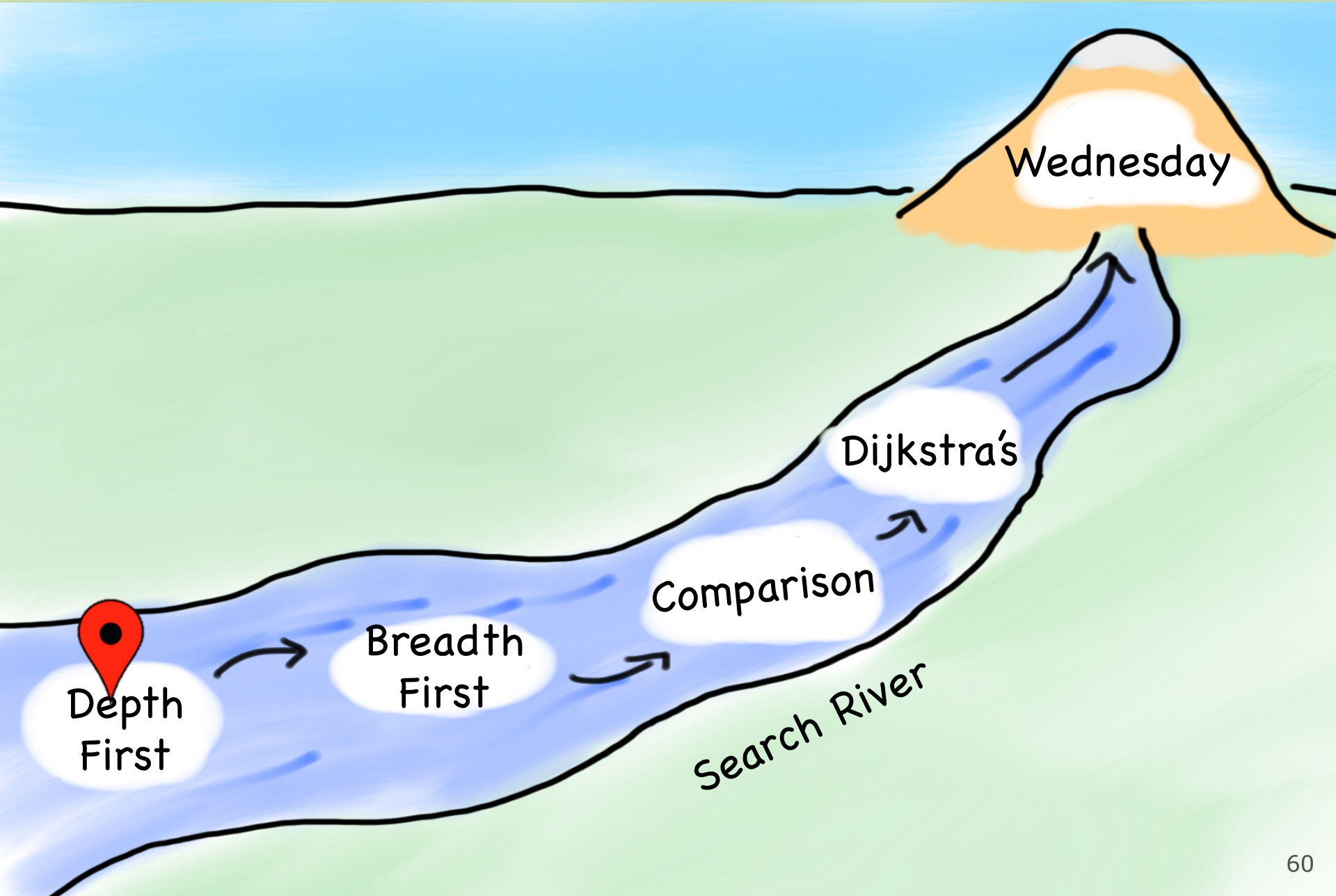
LIFO

`push` and `pop`

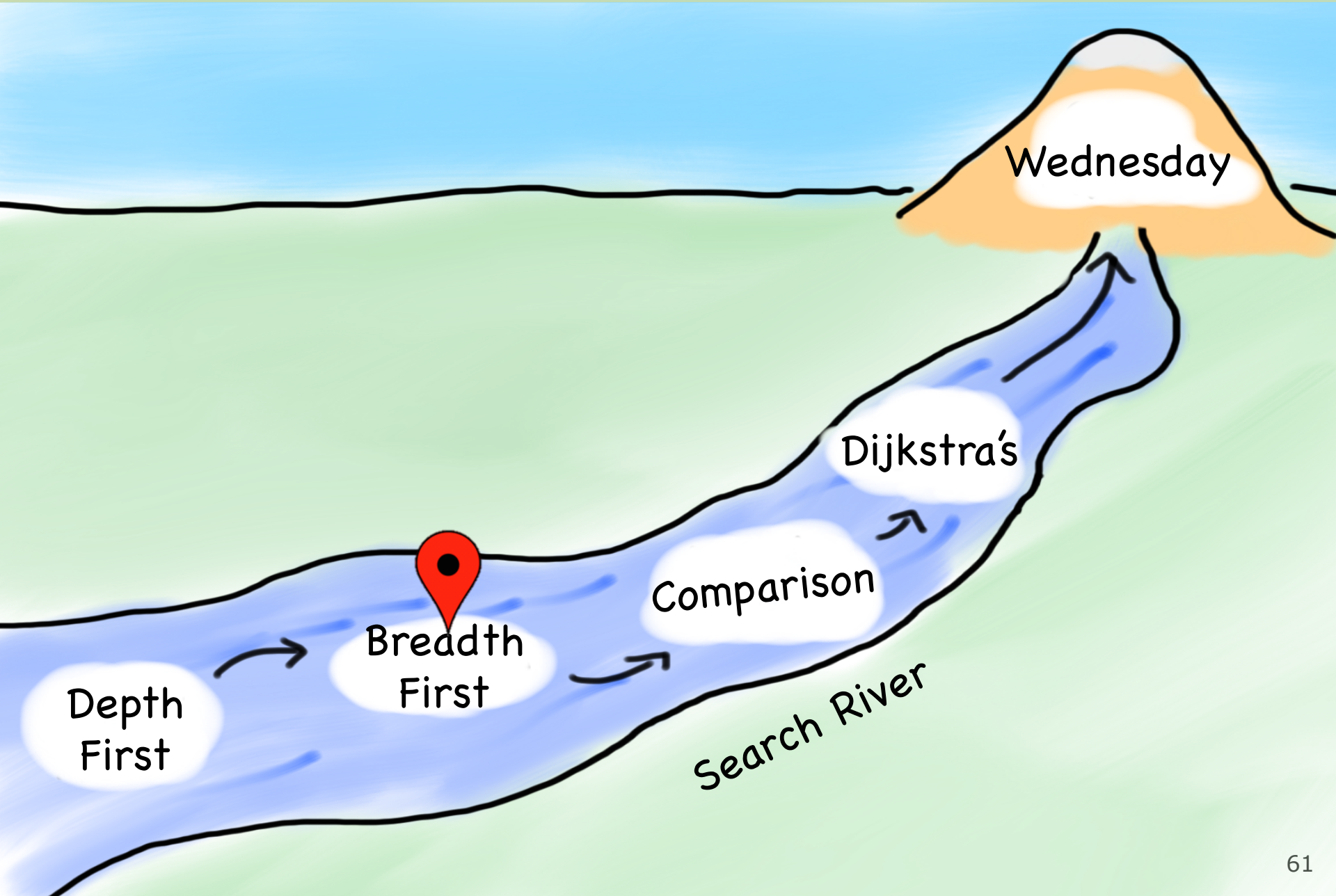
Dish metaphor



Today's Route



Today's Route



Better Data Structure

What about a Queue?

Breadth First Search Collection

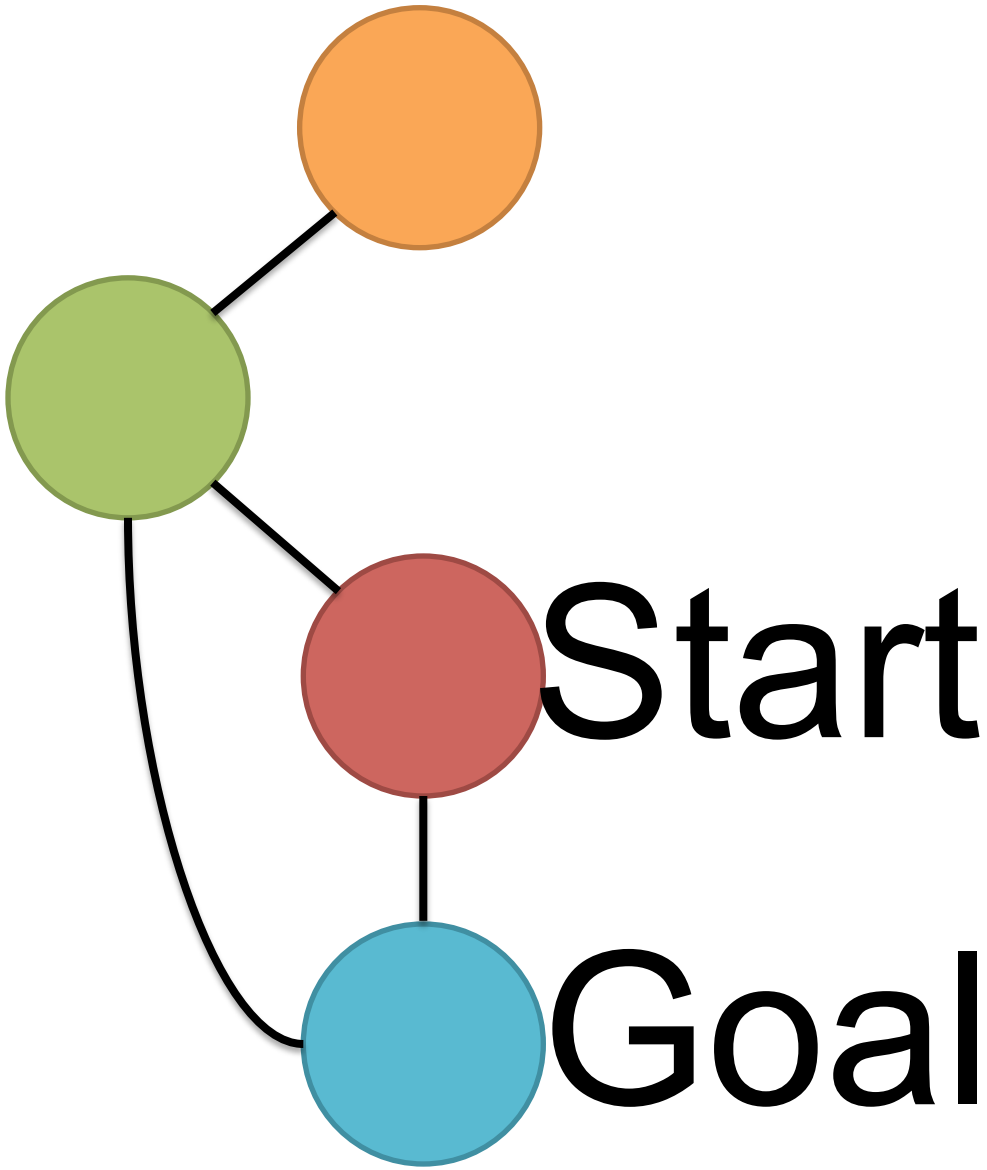
Depth First Search

```
stack = Stack()
stack.push(newPath(startNode))
seen = Set();
while !stack.isEmpty():
    currPath = stack.pop()
    currState = last(currPath);
    if(currState is goal) return currState;
    if(seen contains currState) continue;
    seen.add(currState);
    for nextState in getNextStates(currState)
        path = newPath(currPath, nextState);
        stack.push(path);
    }
}
```

Breadth First Search

```
queue = Queue()
queue.enqueue(newPath(startNode))
seen = Set();
while !queue.isEmpty():
    currPath = queue.dequeue()
    currState = last(currPath);
    if(currState is goal) return currState;
    if(seen contains currState) continue;
    seen.add(currState);
    for nextState in getNextStates(currState)
        path = newPath(currPath, nextState);
        queue.enqueue(path);
    }
}
```

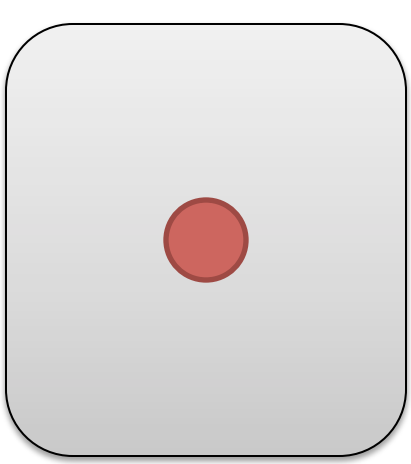
Most Simple Graph



Breadth First Search

Enqueue the start state

Breadth First Search

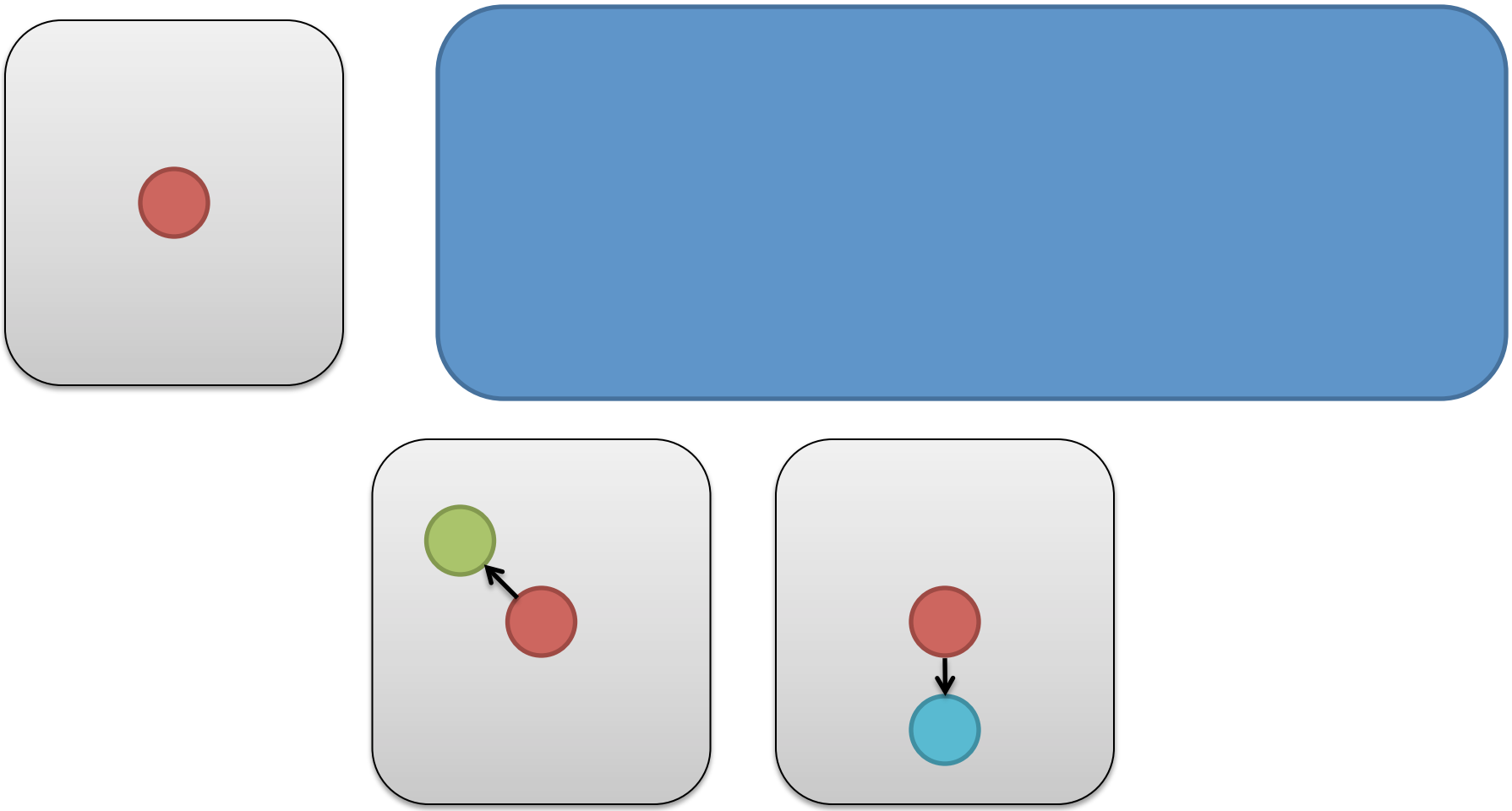


Dequeue the first state

Breadth First Search

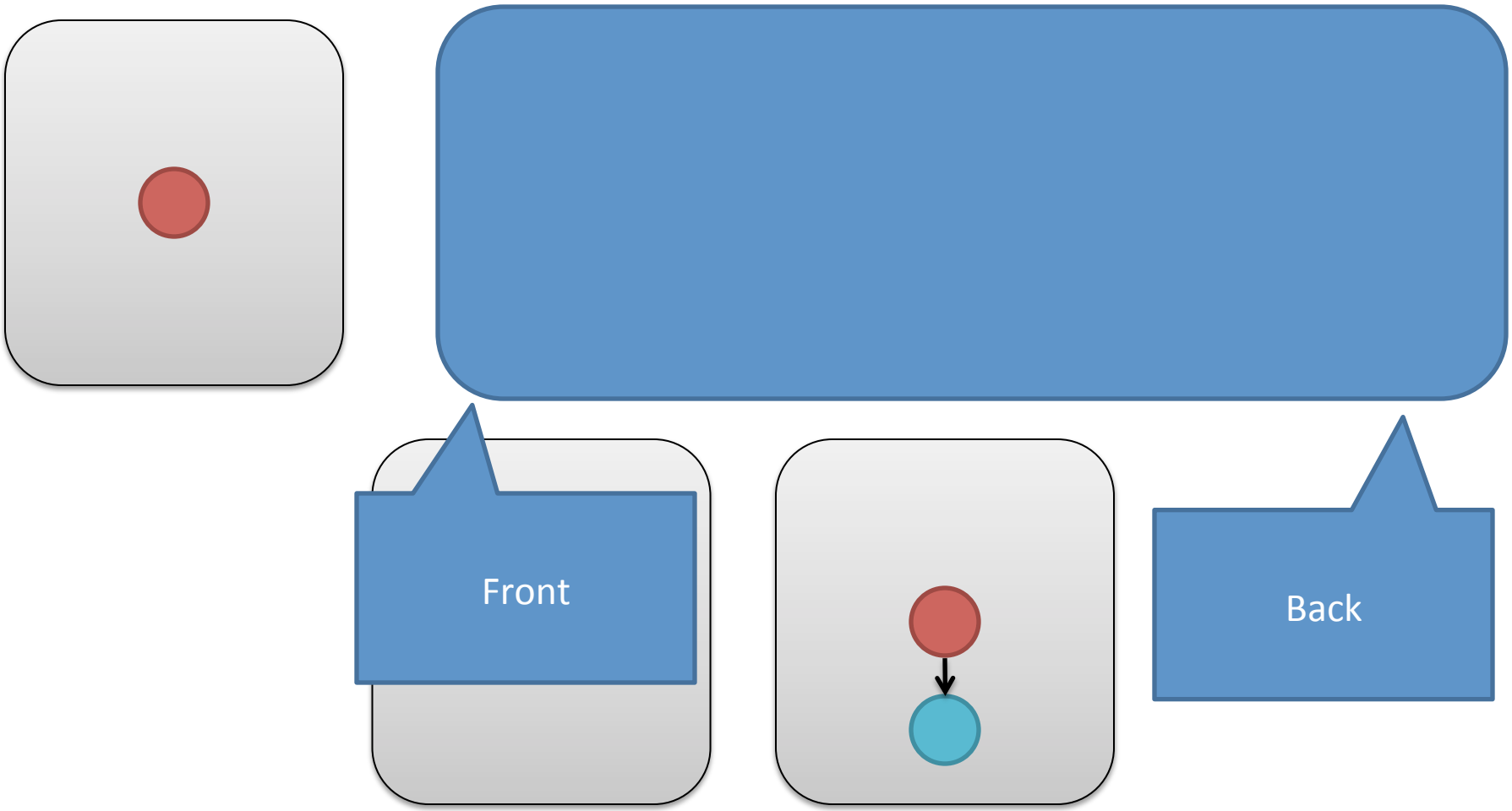
Check if it’s the goal or if it has been visited

Breadth First Search



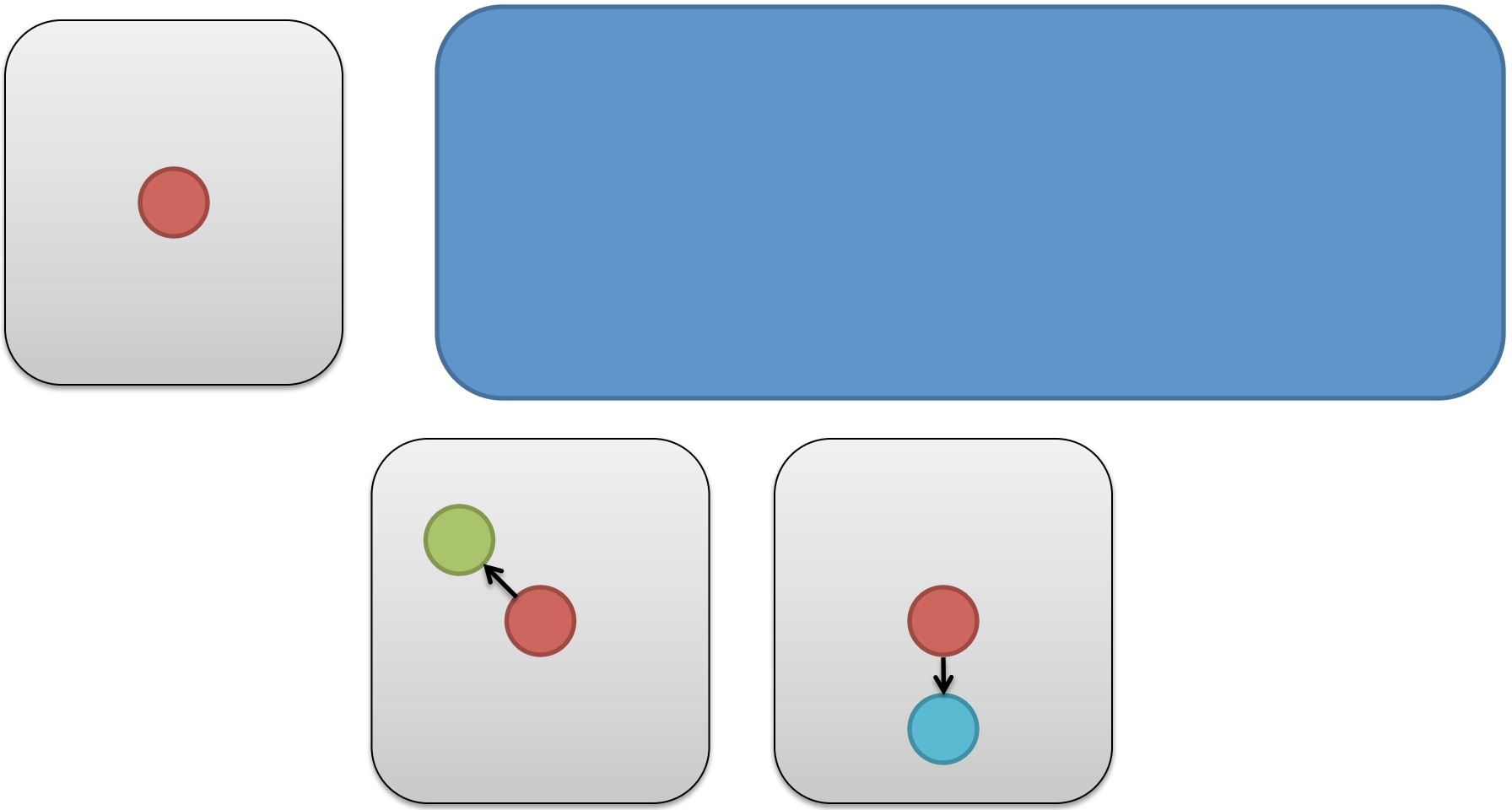
Find each successor state

Breadth First Search



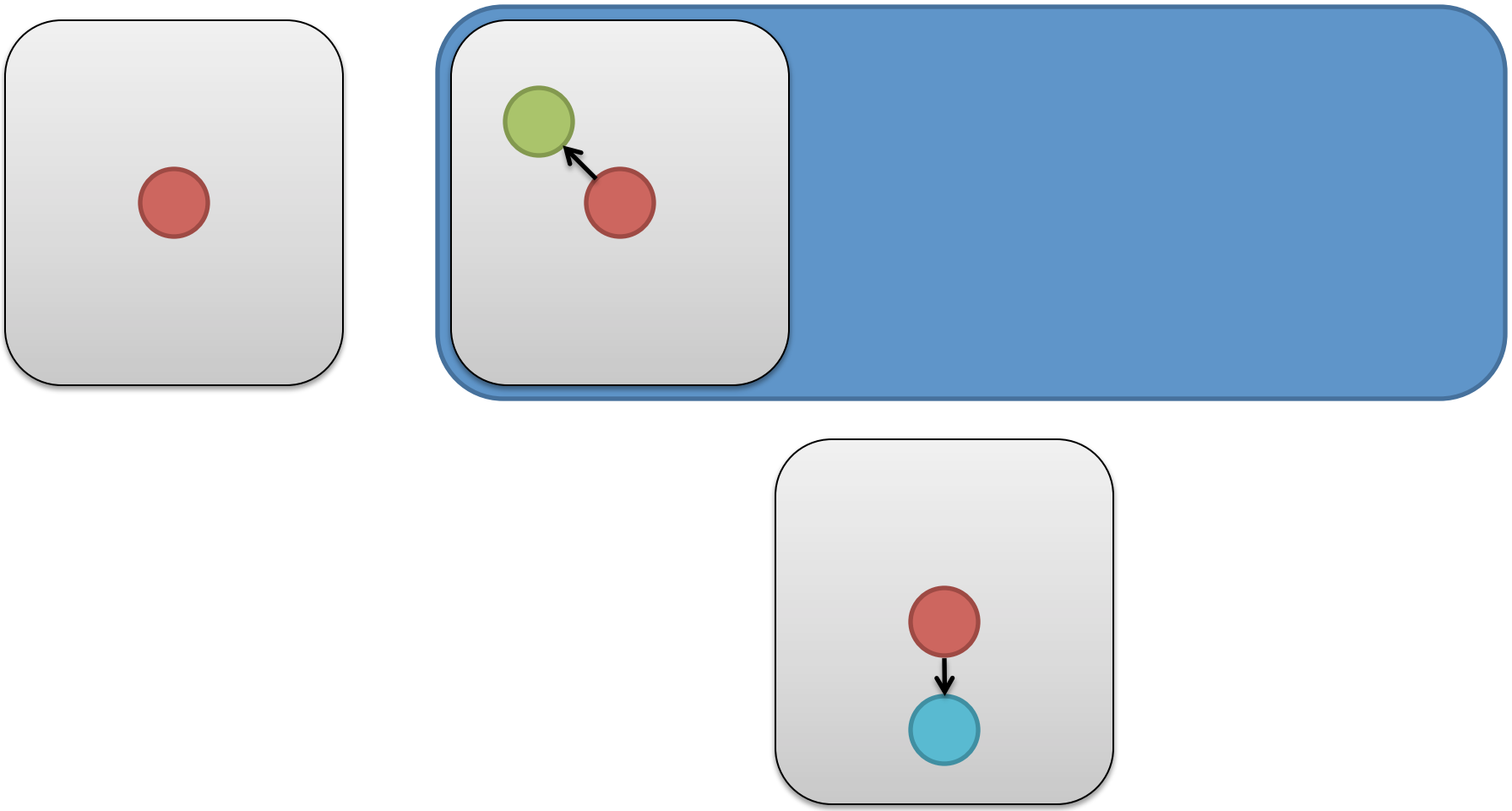
Find each successor state

Breadth First Search



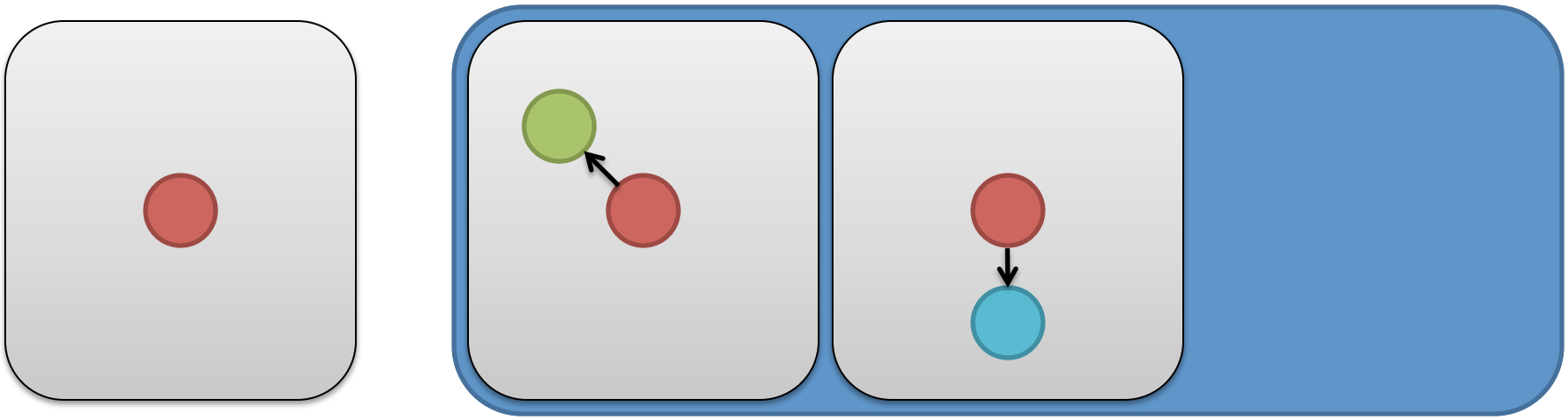
Find each successor state

Breadth First Search



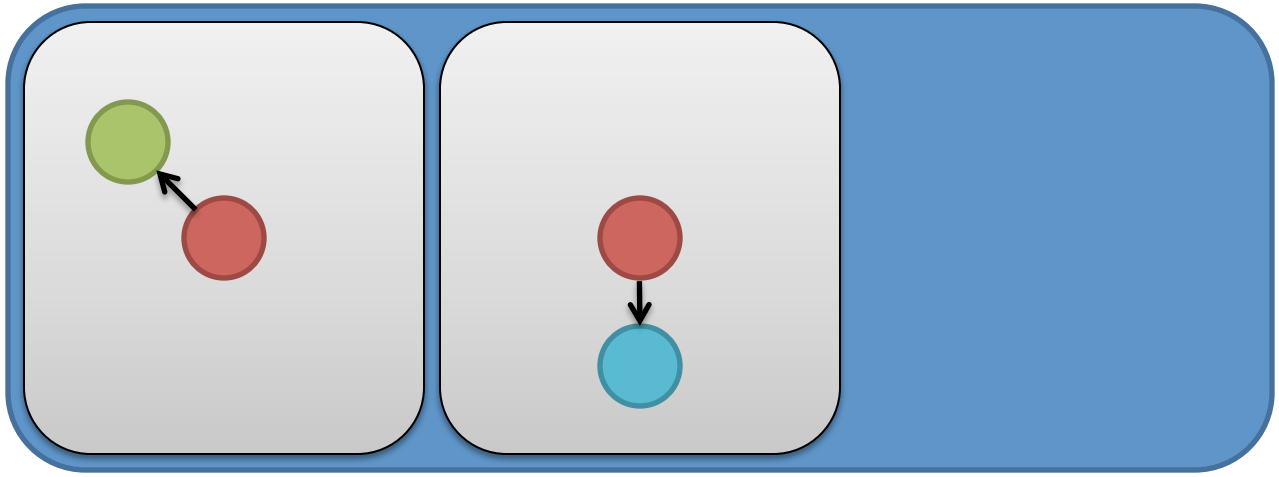
Enqueue the successor states

Breadth First Search



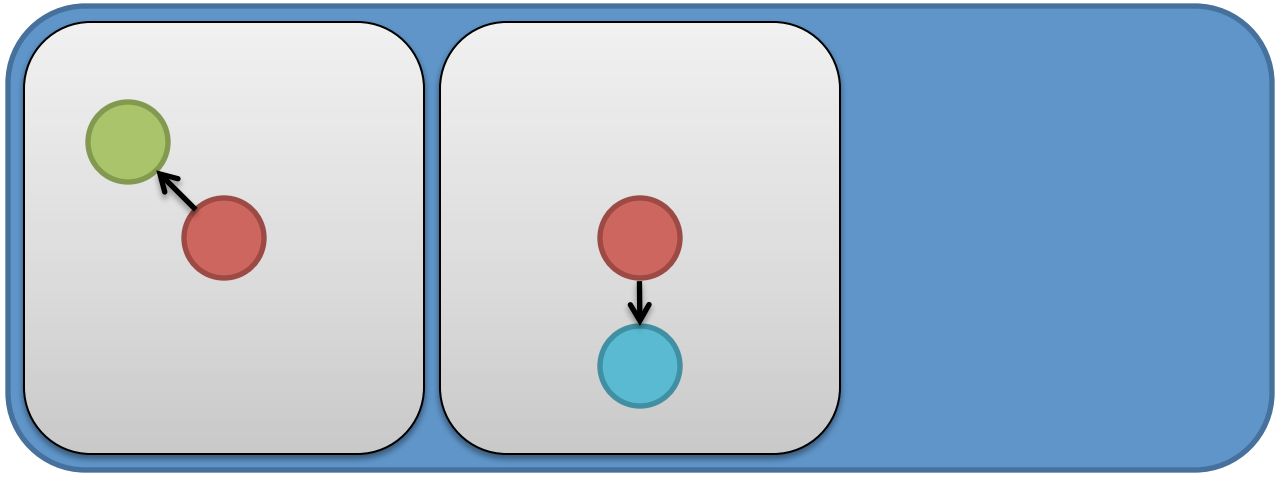
Enqueue the successor states

Breadth First Search



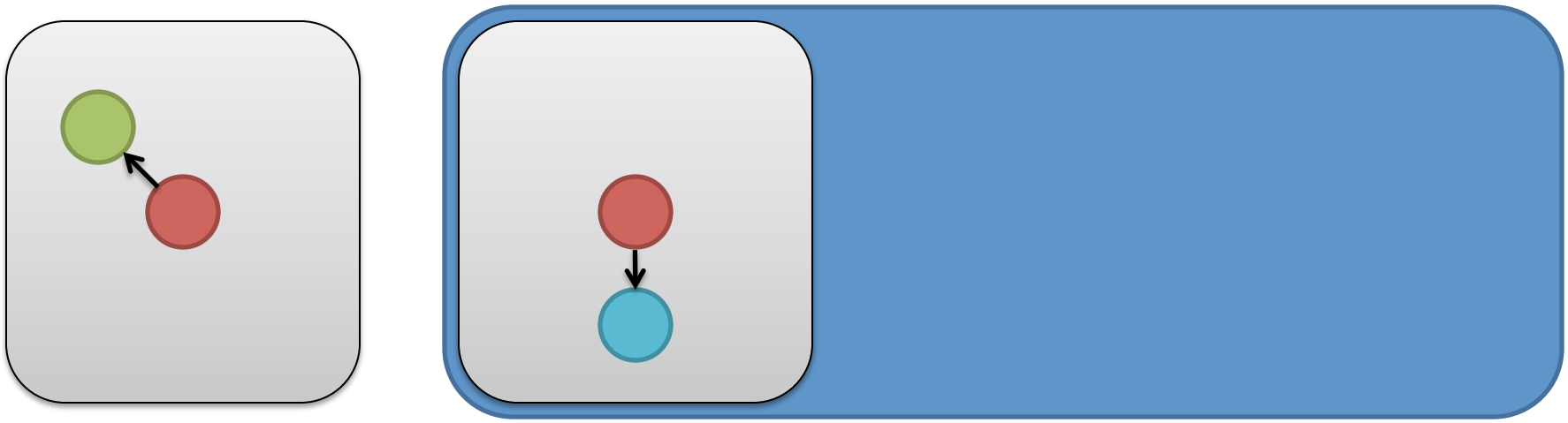
Forget about the dequeued state

Breadth First Search



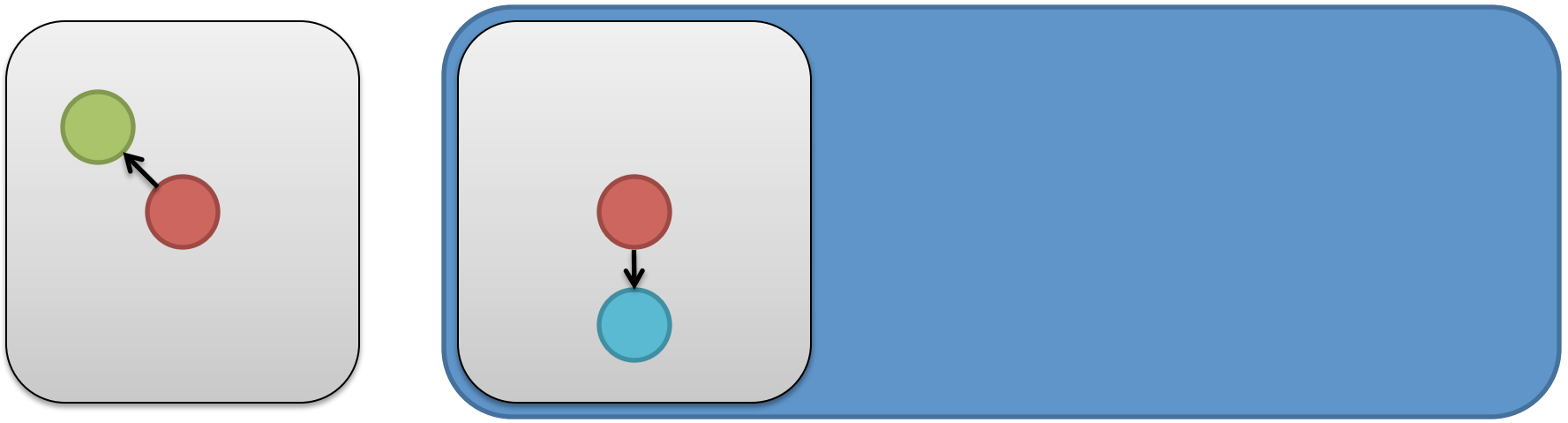
Back to the start of the while loop

Breadth First Search



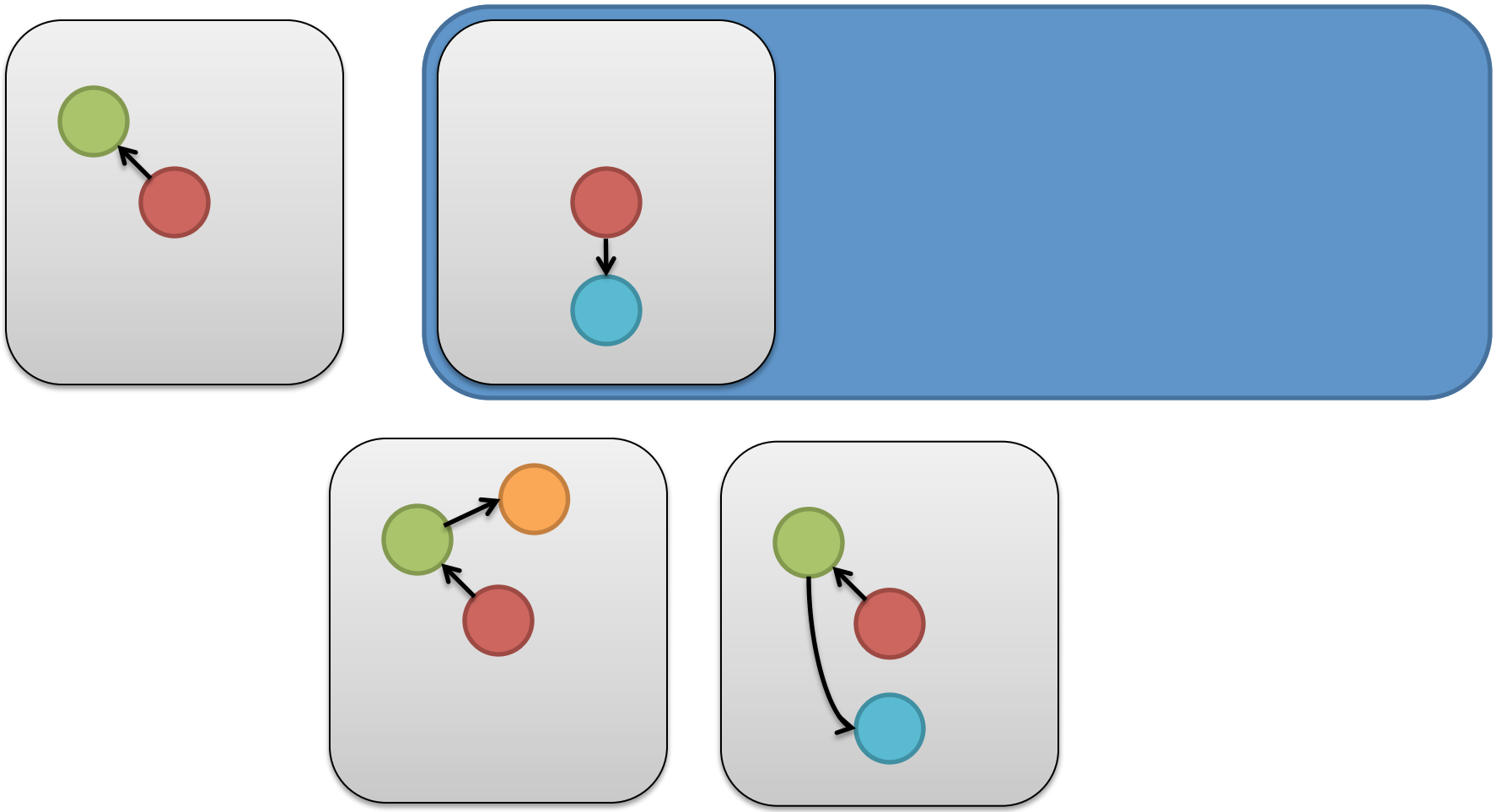
Dequeue a state

Breadth First Search



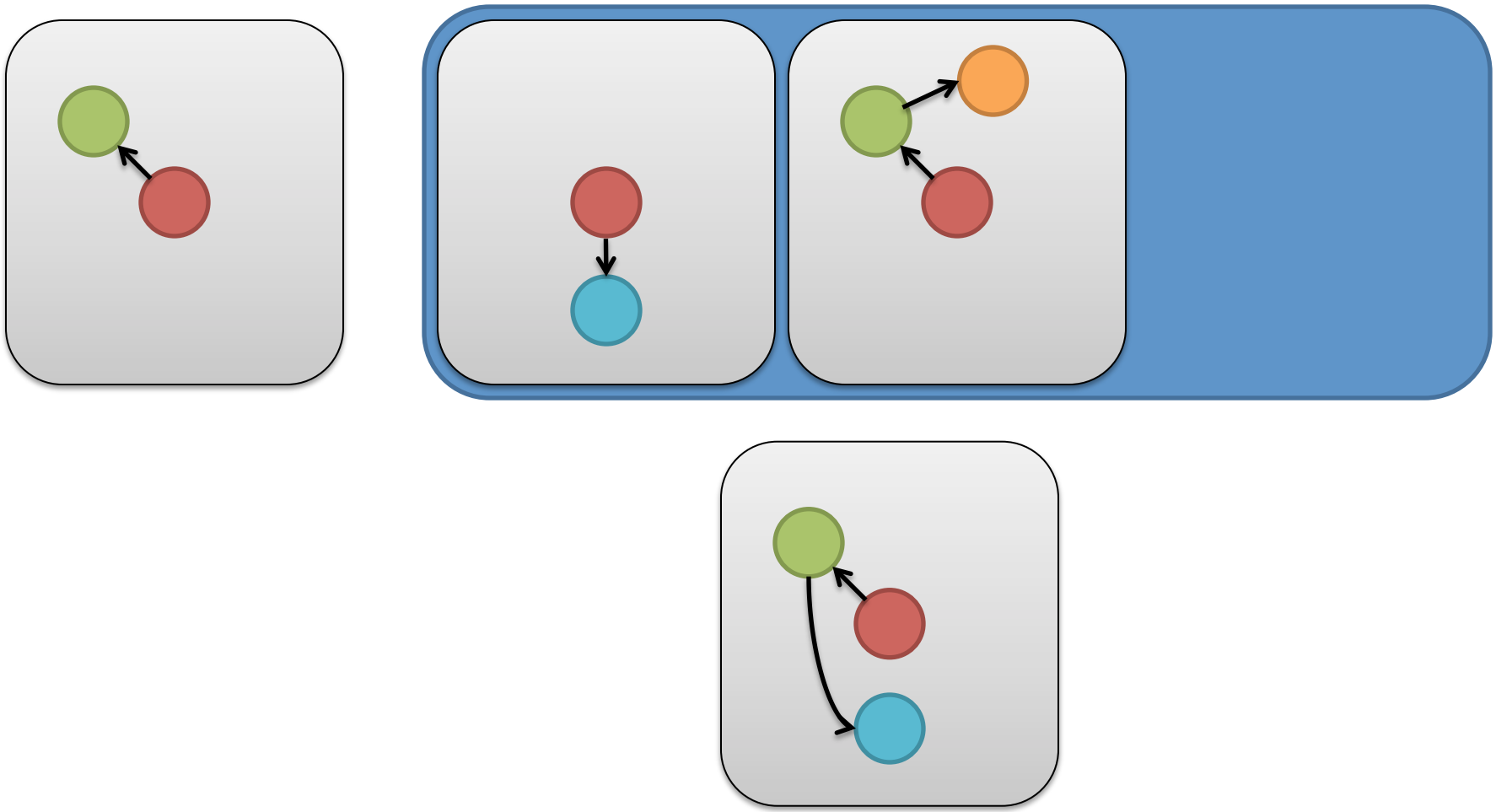
Check if it's a goal or if the state has been visited

Breadth First Search



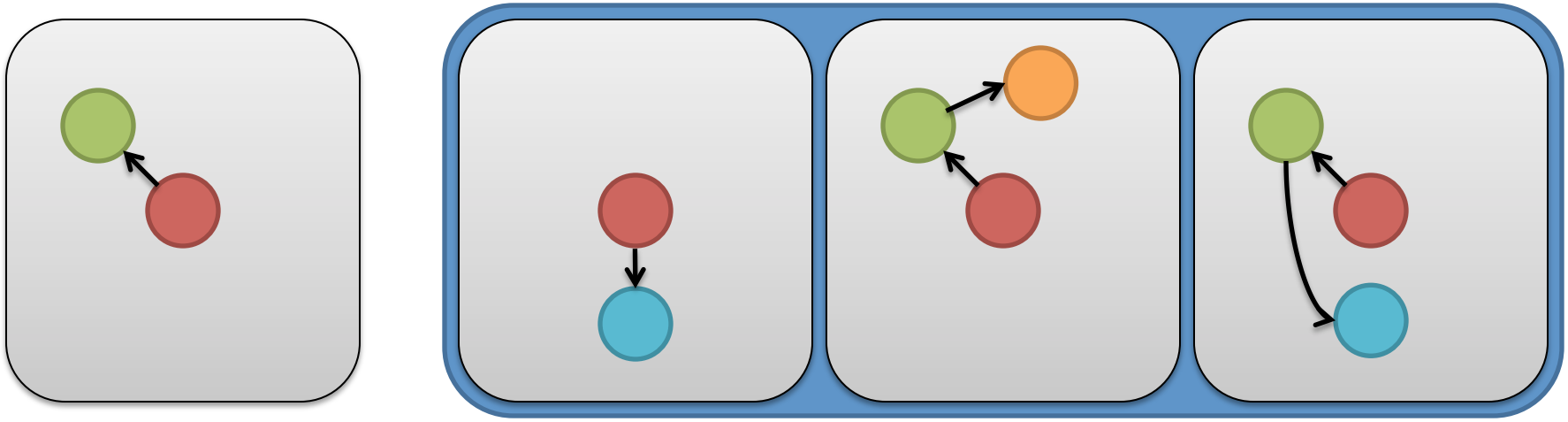
Find each successor state

Breadth First Search



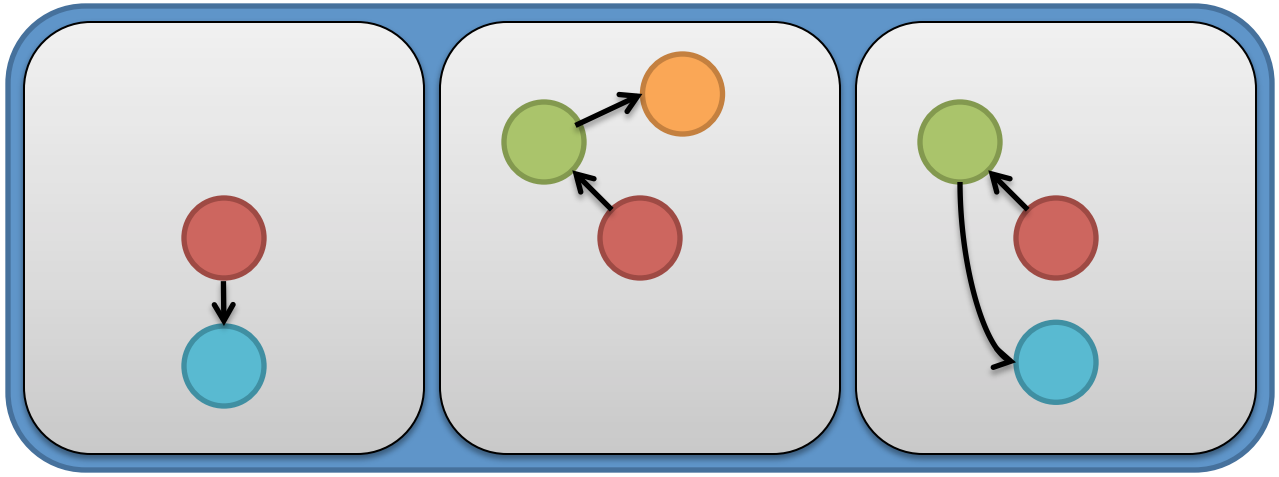
Enqueue each successor

Breadth First Search



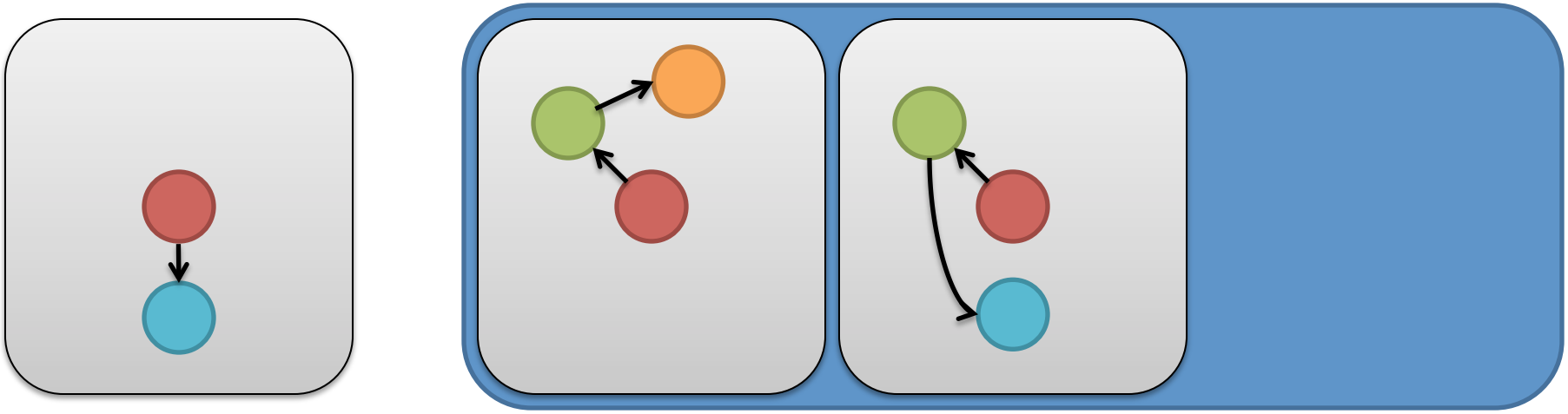
Enqueue each successor

Breadth First Search



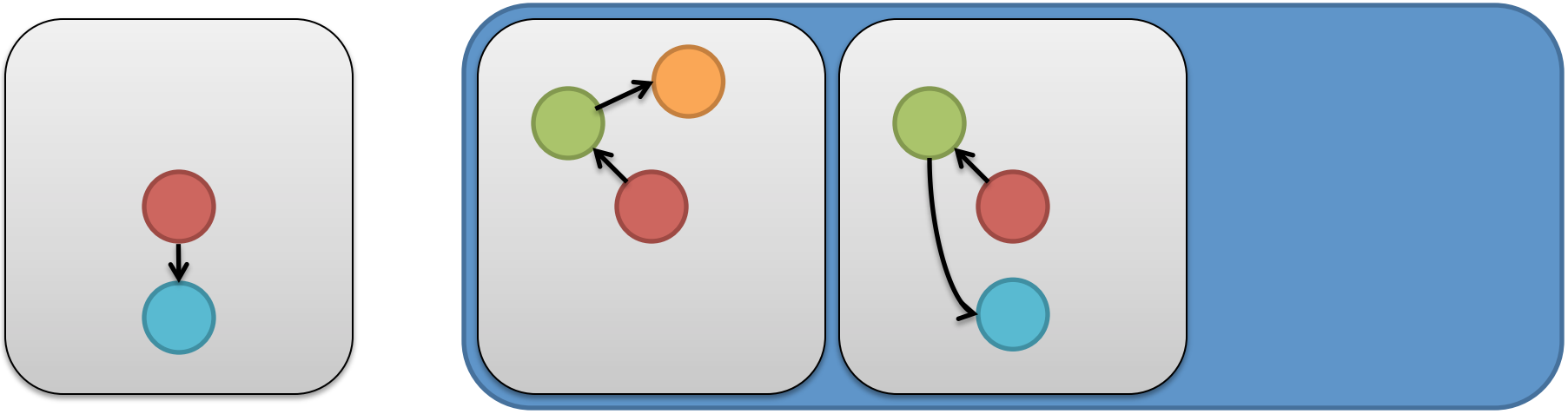
Back to the start of the while loop

Breadth First Search



Dequeue a path

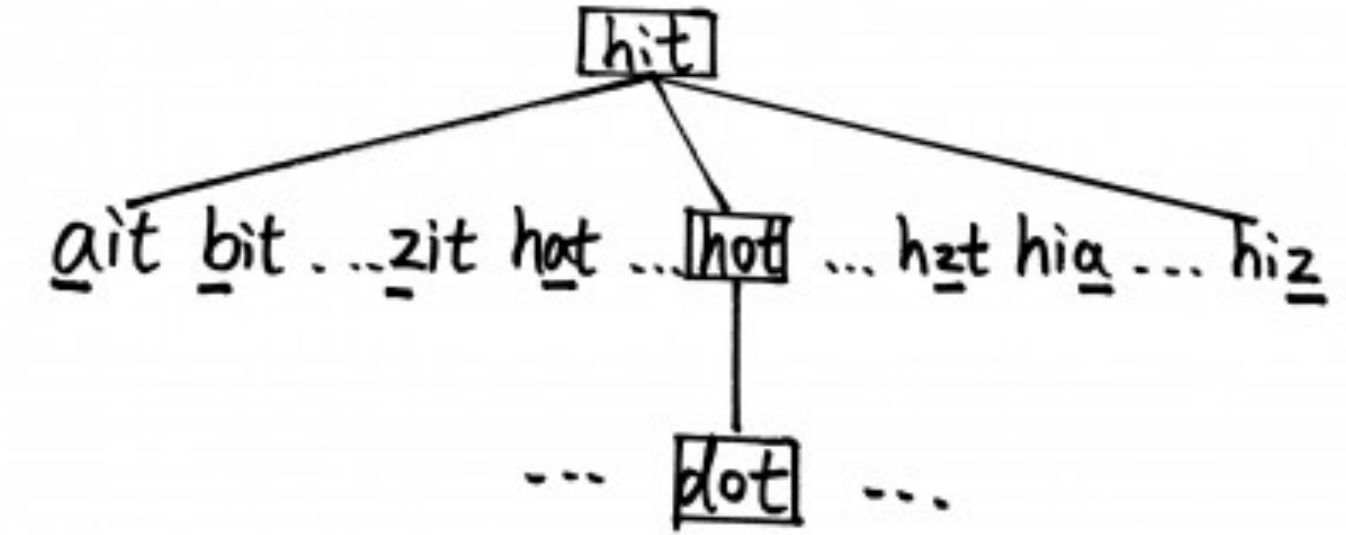
Breadth First Search

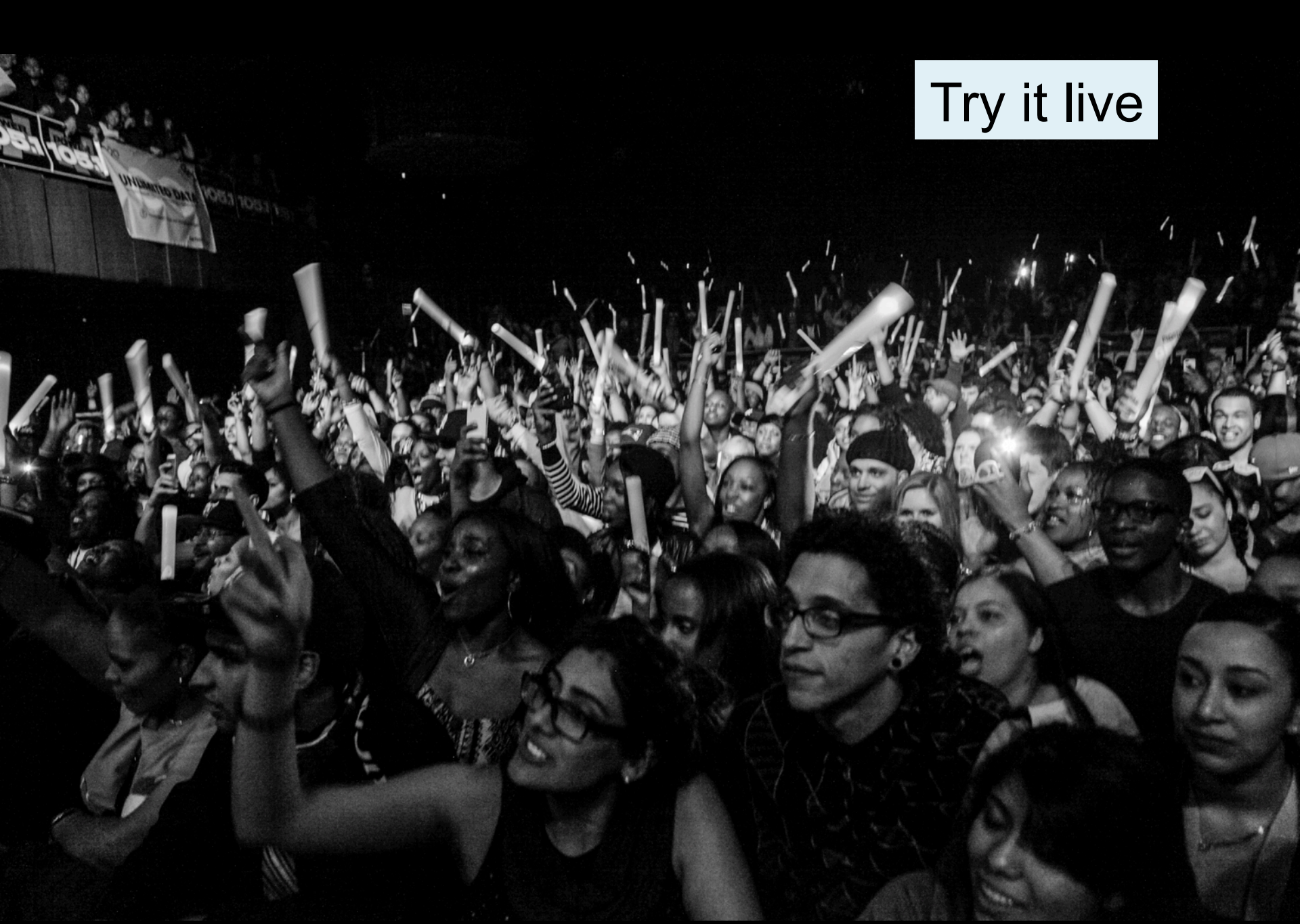


Check if it's a goal

You have seen this before…

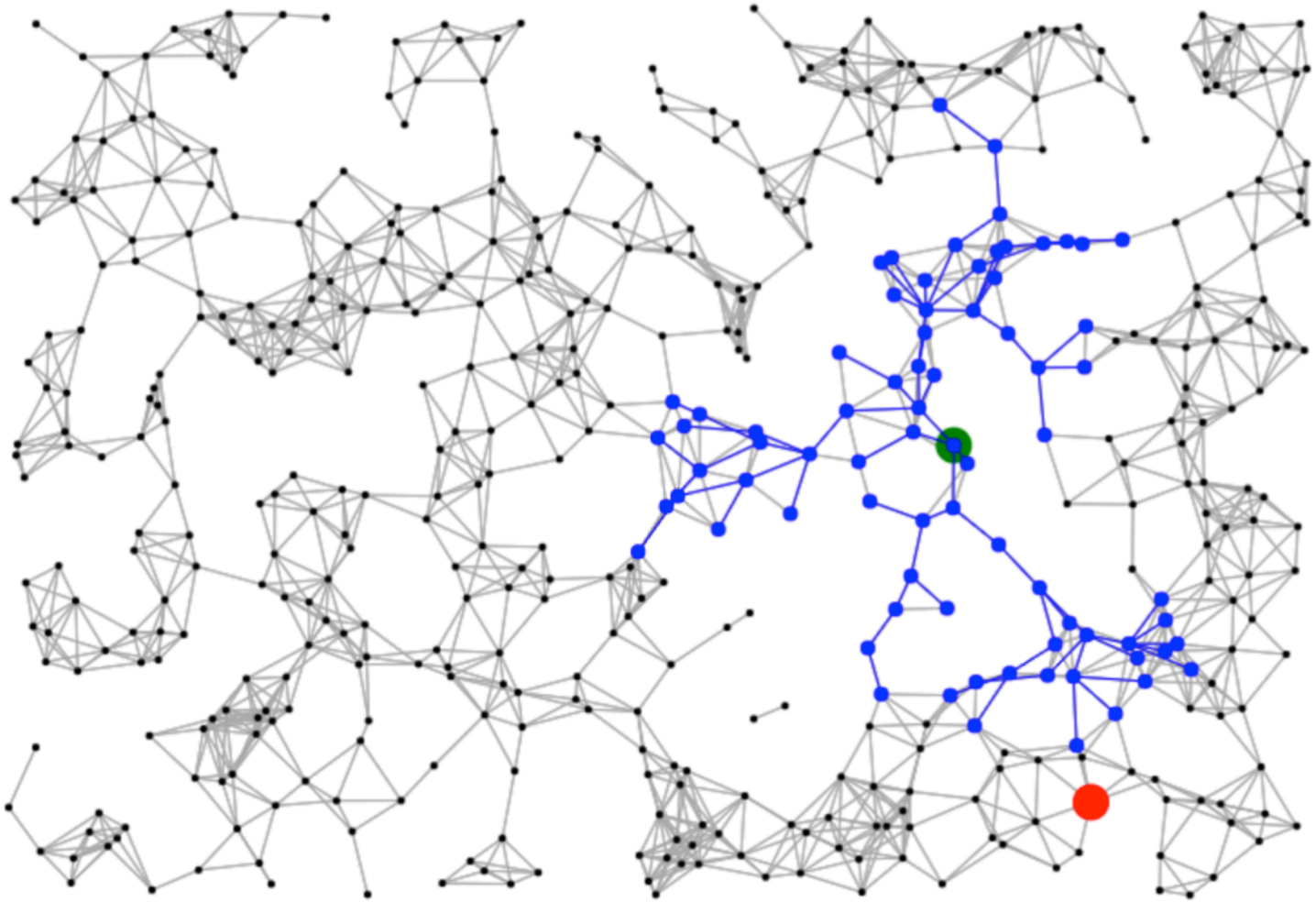
Word Ladder



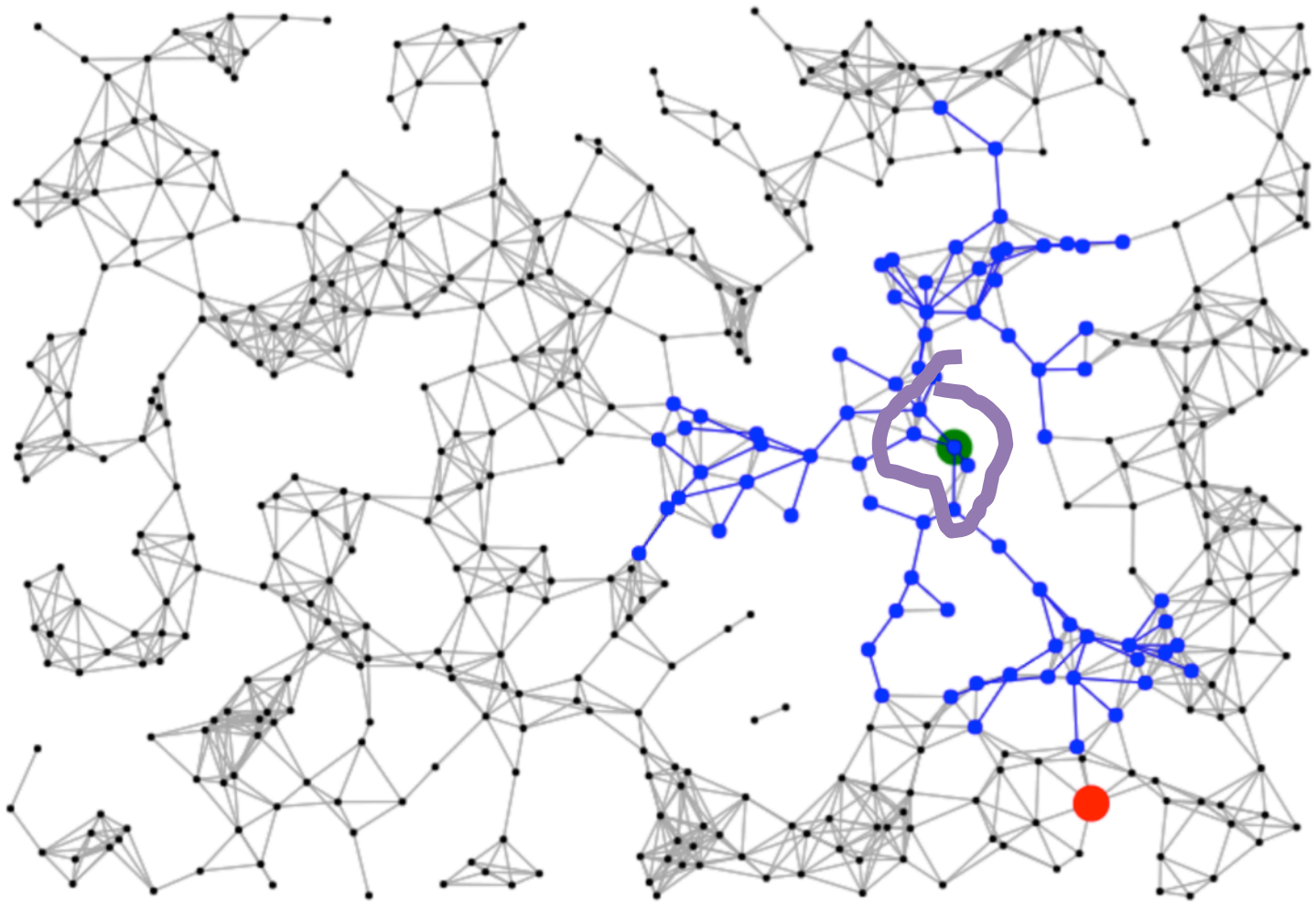

Try it live

Boom!

Guaranteed Shortest

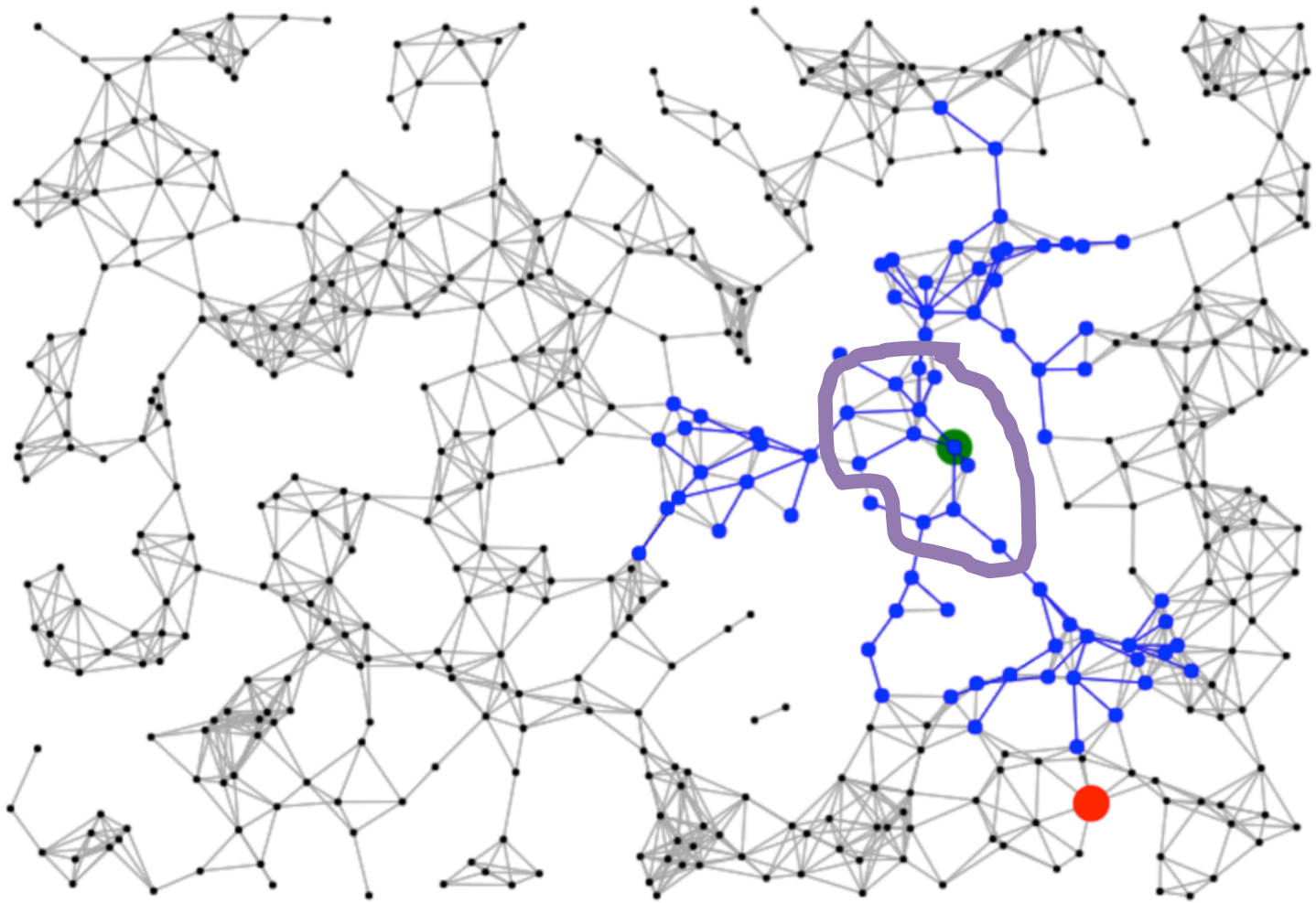


Guaranteed Shortest



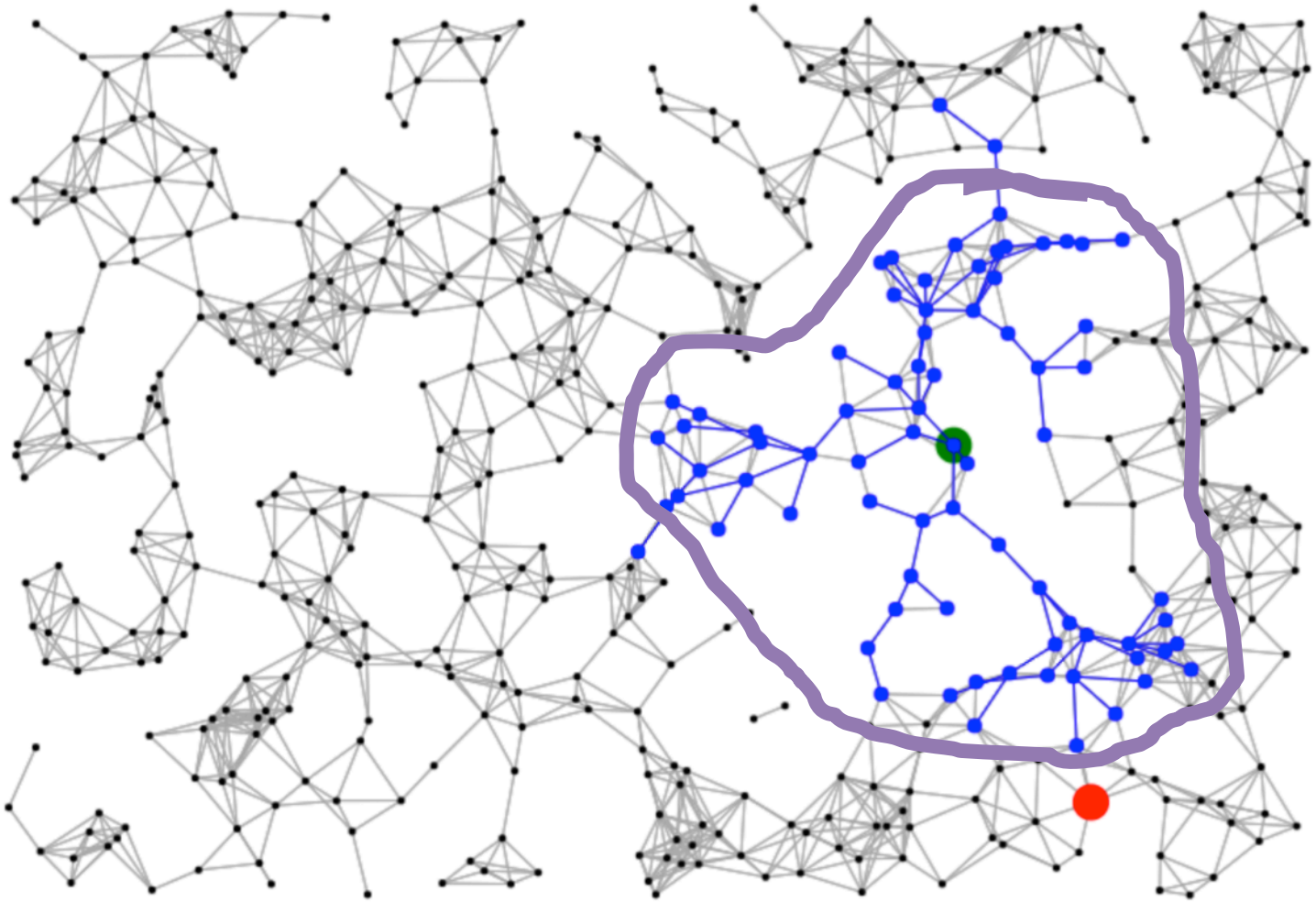
Must explore all paths of length 1, before any paths of length 2

Guaranteed Shortest



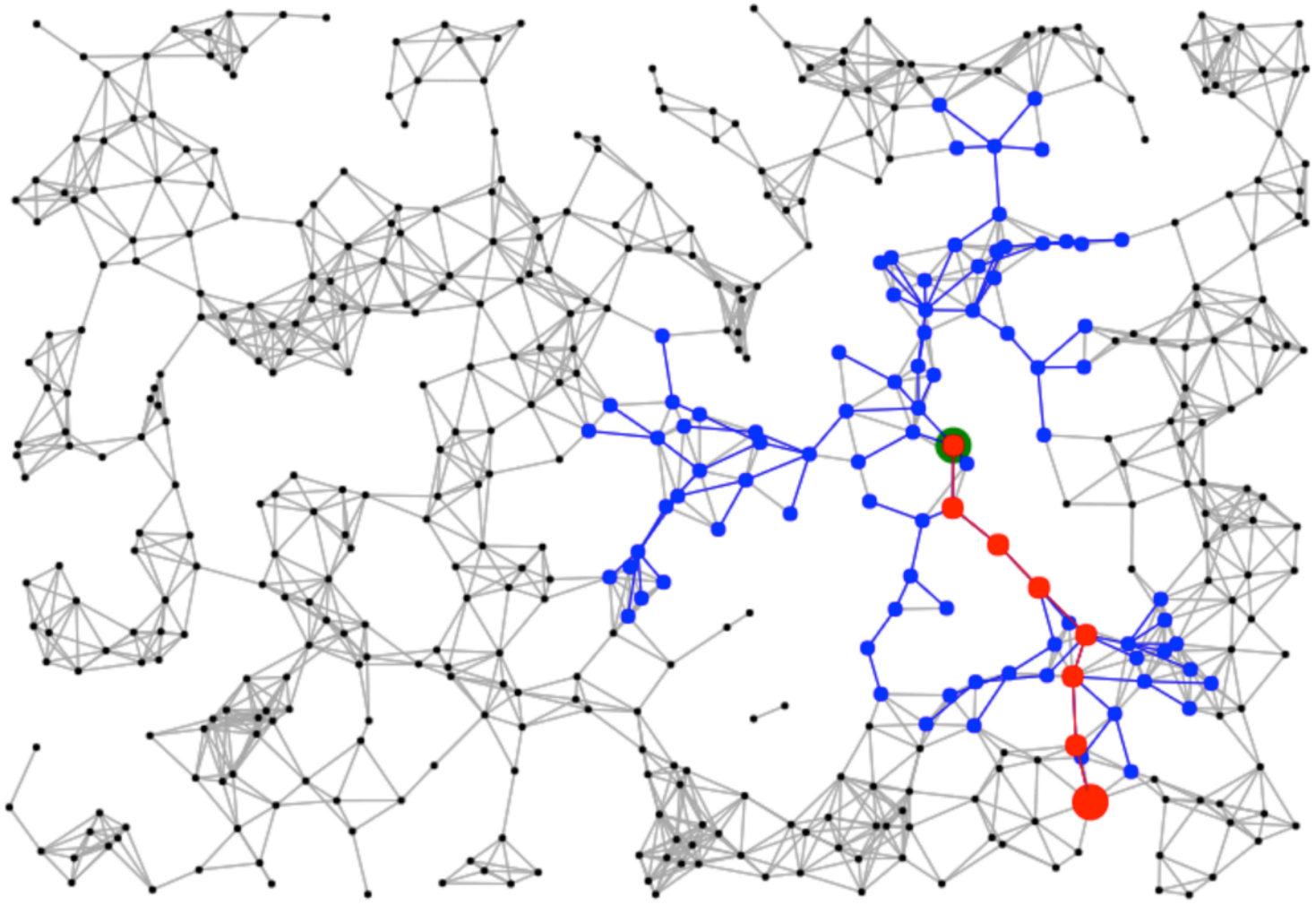
Must explore all paths of length 2, before any paths of length 3

Guaranteed Shortest



Must explore all paths of length *k*, before any paths of length *k+1*

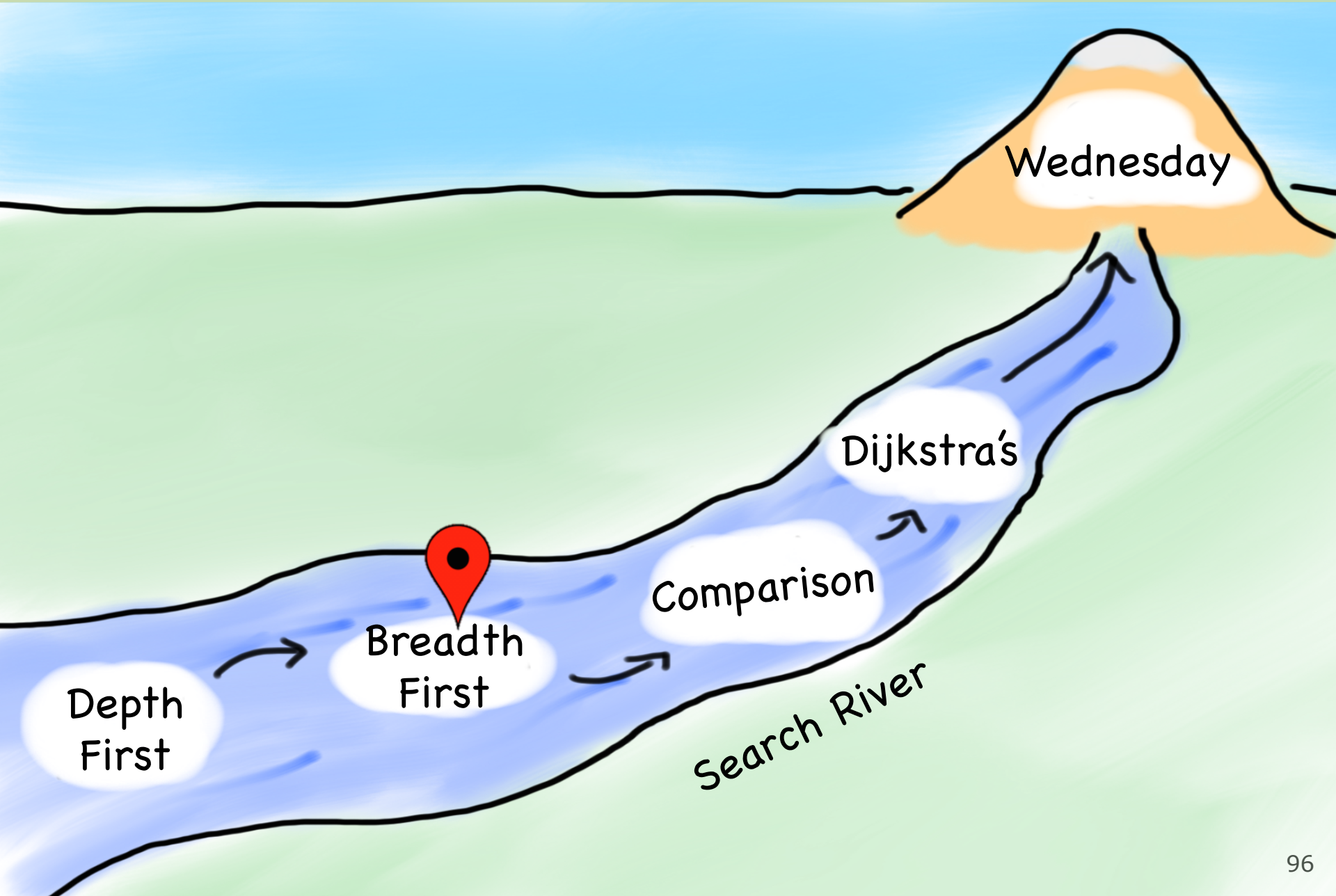
Guaranteed Shortest



When you find a path to an unseen node, it must be the shortest

Slimemold BFS

Today's Route



Today's Route

BFS Pros

It is guaranteed to find the path with the shortest number of hops

BFS Con

It uses a ton of memory (the queue can get very large)

DFS Pros

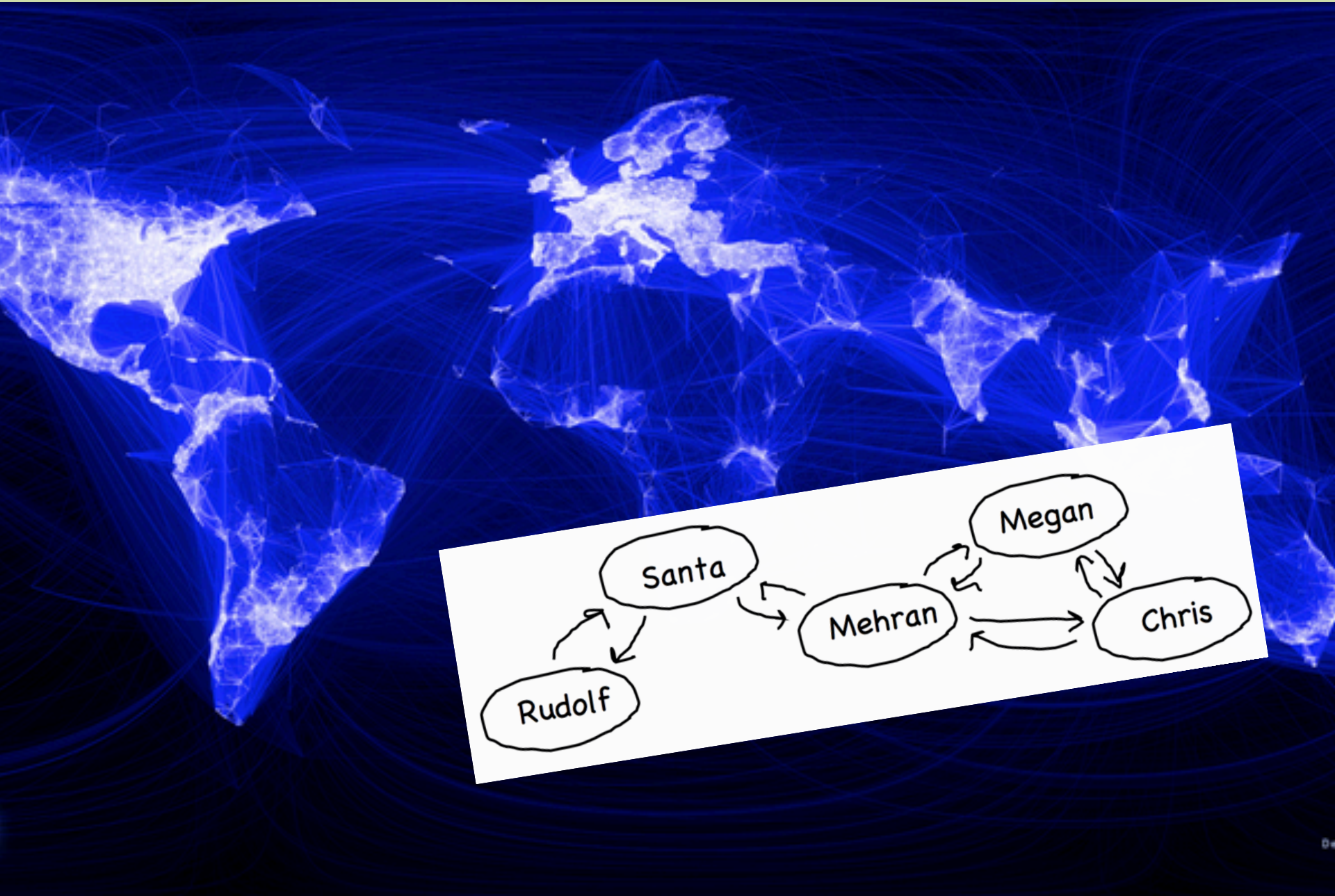
If you implement depth first search recursively it only has to store a single path in memory! BFS must store all states in the fringe.

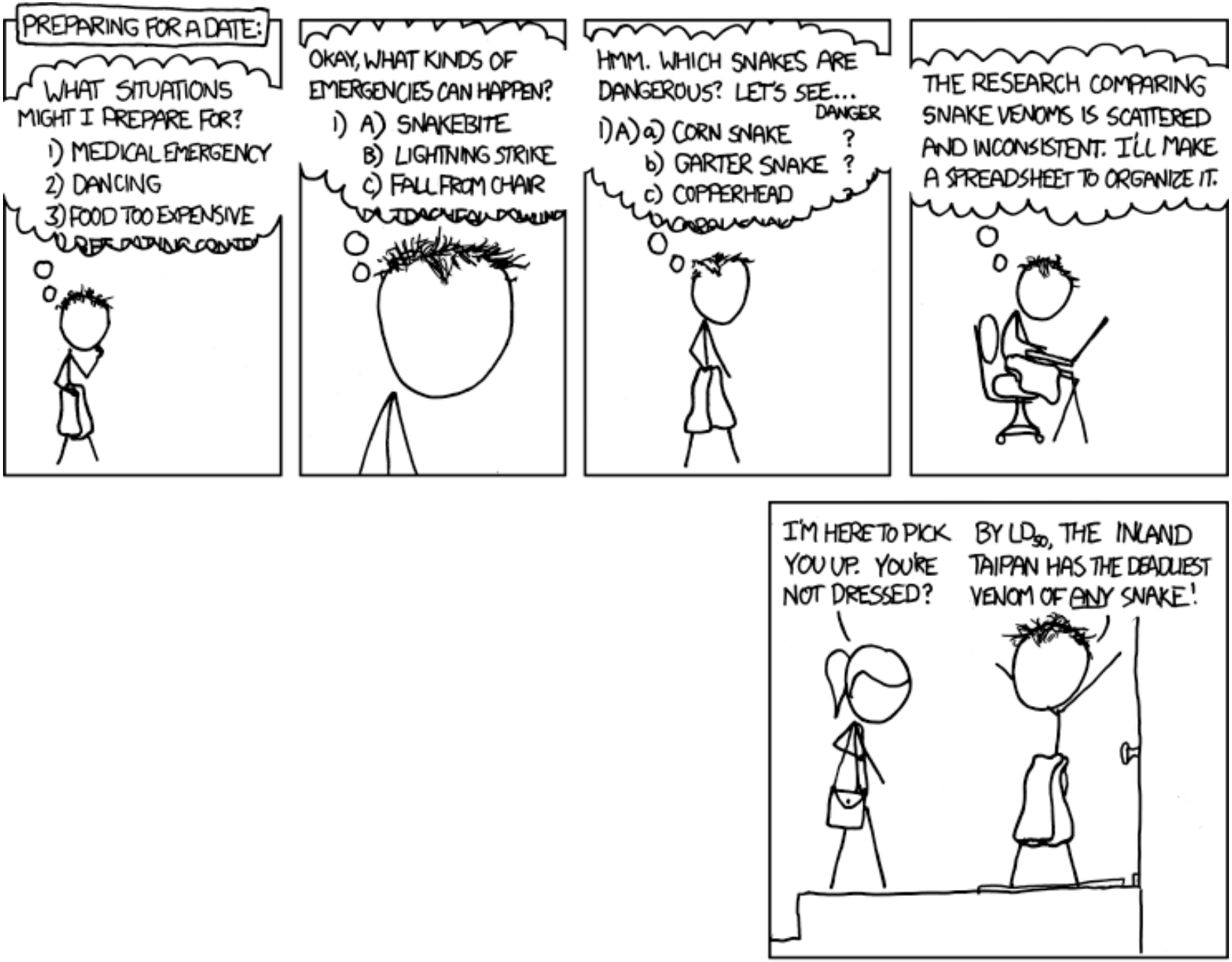
DFS Cons

Not guaranteed to find the lowest cost solution for all problems.

Can get lost.

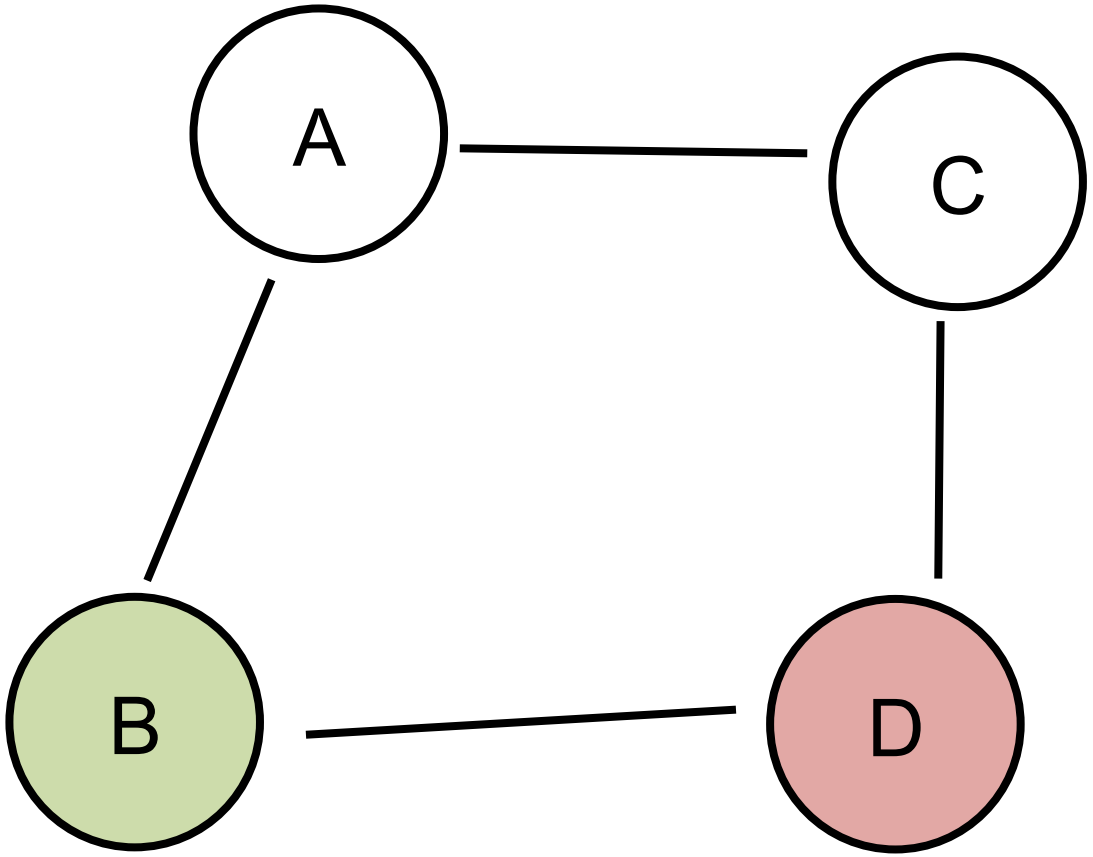
Best of Both Worlds?



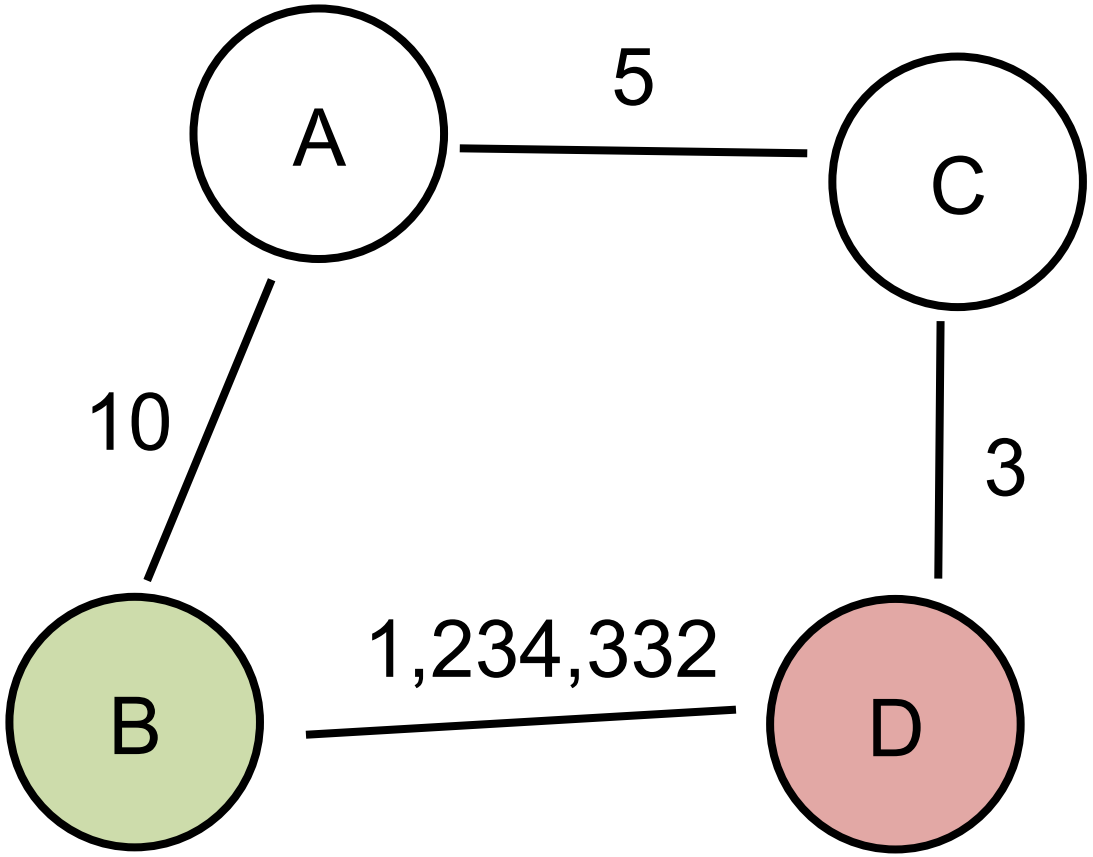

PREPARING FOR A DATE:
WHAT SITUATIONS MIGHT I PREPARE FOR?
1) MEDICAL EMERGENCY
2) DANCING
3) FOOD TOO EXPENSIVE
OKAY, WHAT KINDS OF EMERGENCIES CAN HAPPEN?
1) A) SNAKEBITE
B) LIGHTNING STRIKE
C) FALL FROM CHAIR
HMM. WHICH SNAKES ARE DANGEROUS? LET'S SEE...
1) A) a) CORN SNAKE ?
b) GARTER SNAKE ?
c) COPPERHEAD
DANGER
THE RESEARCH COMPARING SNAKE VENOMS IS SCATTERED AND INCONSISTENT. I'LL MAKE A SPREADSHEET TO ORGANIZE IT.
I'M HERE TO PICK YOU UP. YOU'RE NOT DRESSED?
BY LD_{50}, THE INLAND TAIPAN HAS THE DEADLIEST VENOM OF ANY SNAKE!
I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

Weights?

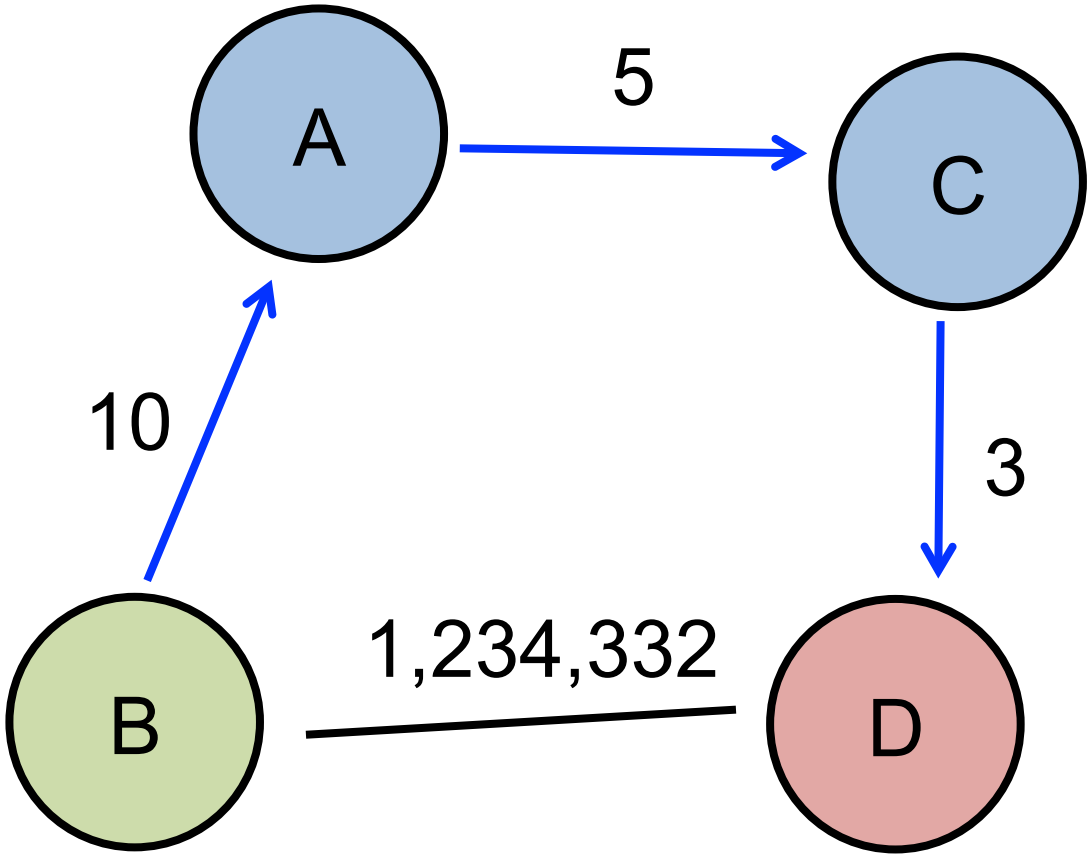
Shortest Path



With Weights



With Weights



Realistic

Realistic

Shortest Path Problem

- Suppose that you have a graph representing different locations.
- Each edge has an associated *cost* (or *weight*, *length*, etc.). We'll assume costs are nonnegative.
- Goal: Find the least-cost (or lowest-weight, lowest-length, etc.) path from some node *u* to a node *v*.

Today's Route

Today's Route

Dijkstra's Collection

Breadth First Search

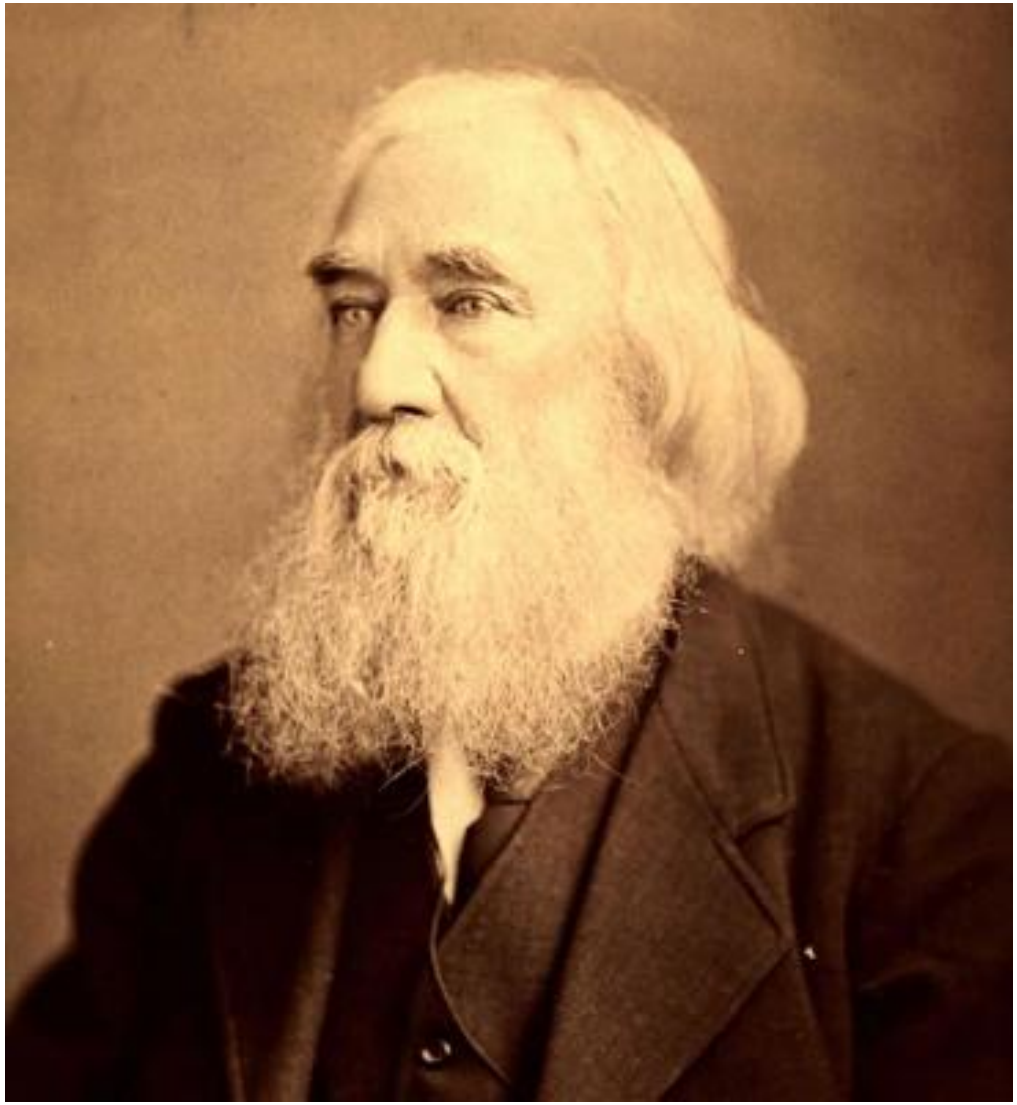
```
queue = Queue()
queue.enqueue(newPath(startNode))
seen = Set();
while !queue.isEmpty():
    currPath = queue.dequeue()
    currState = last(currPath);
    if(currState is goal) return currState;
    if(seen contains currState) continue;
    seen.add(currState);
    for nextState in getNextStates(currState)
        path = newPath(currPath, nextState);
        queue.enqueue(path);
    }
}
```

Dijkstra's Algorithm

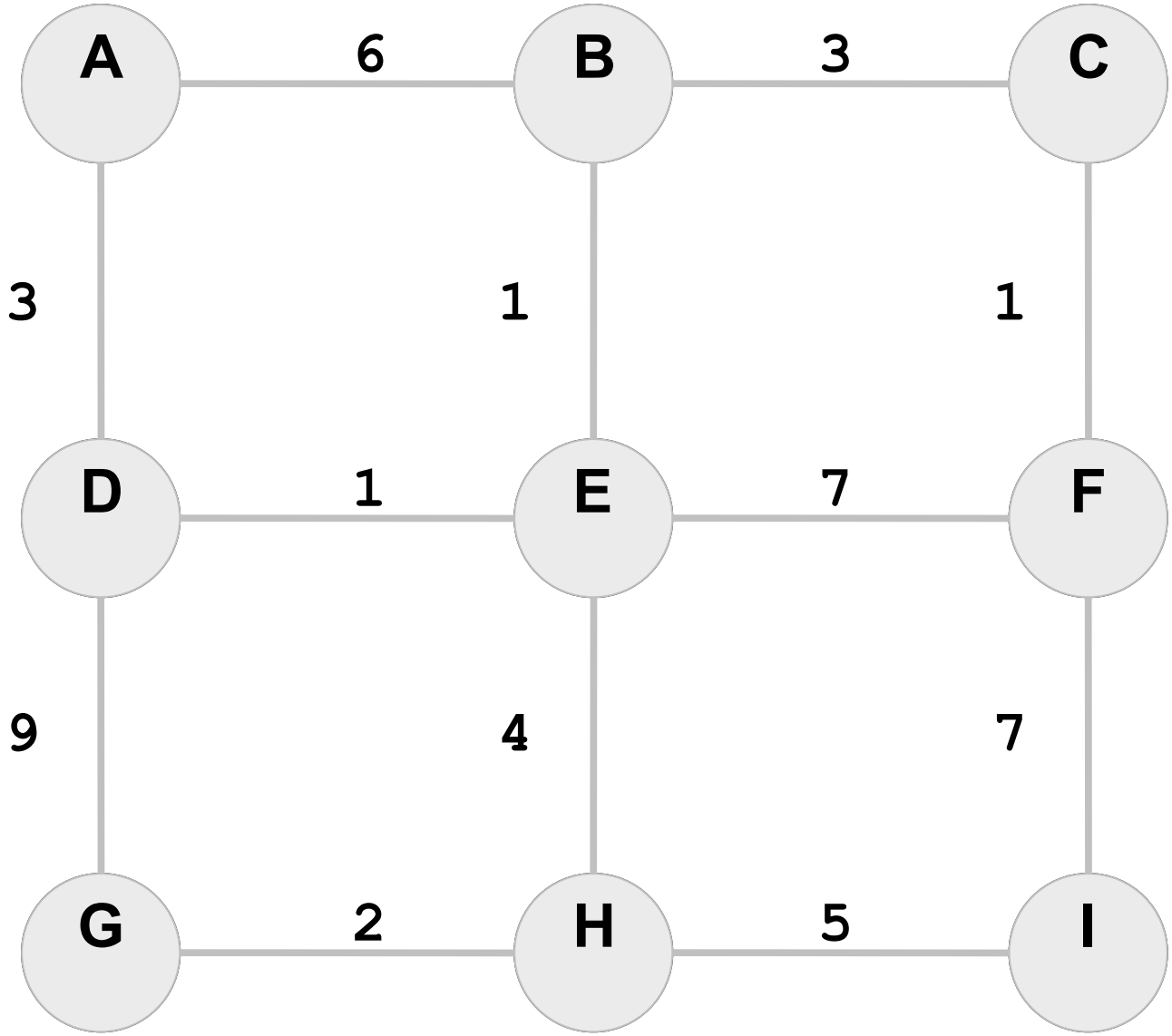
```
pQueue = PriorityQueue()
pQueue.enqueue(newPath(startNode), 0)
seen = Set();
while !pQueue.isEmpty():
    currPath = pQueue.dequeue()
    currState = last(currPath);
    if(currState is goal) return currState;
    if(seen contains currState) continue;
    seen.add(currState);
    for nextState in getNextStates(currState)
        path = newPath(currPath, nextState);
        pQueue.enqueue(path, getCost(path));
    }
}
```

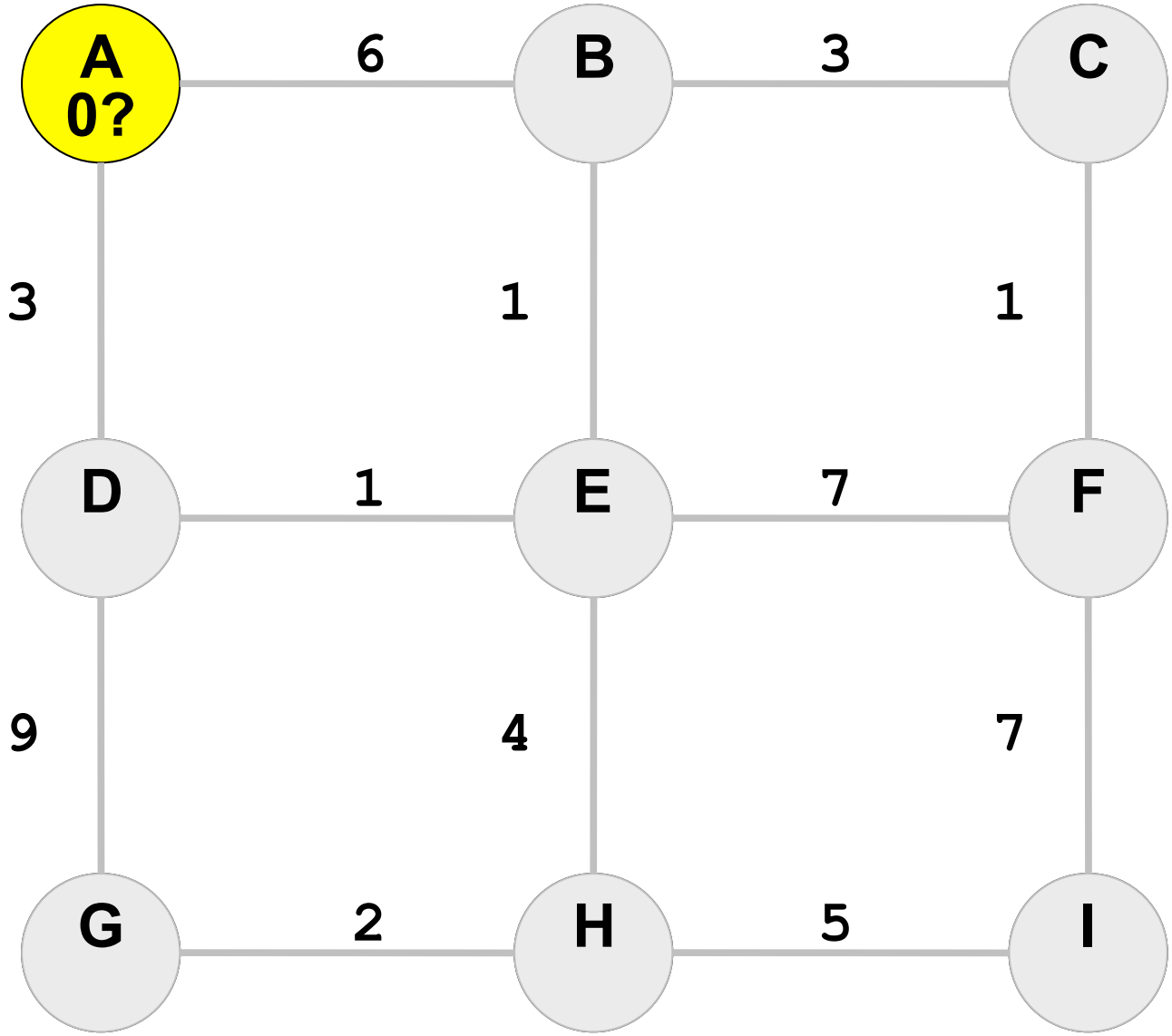

Try it live

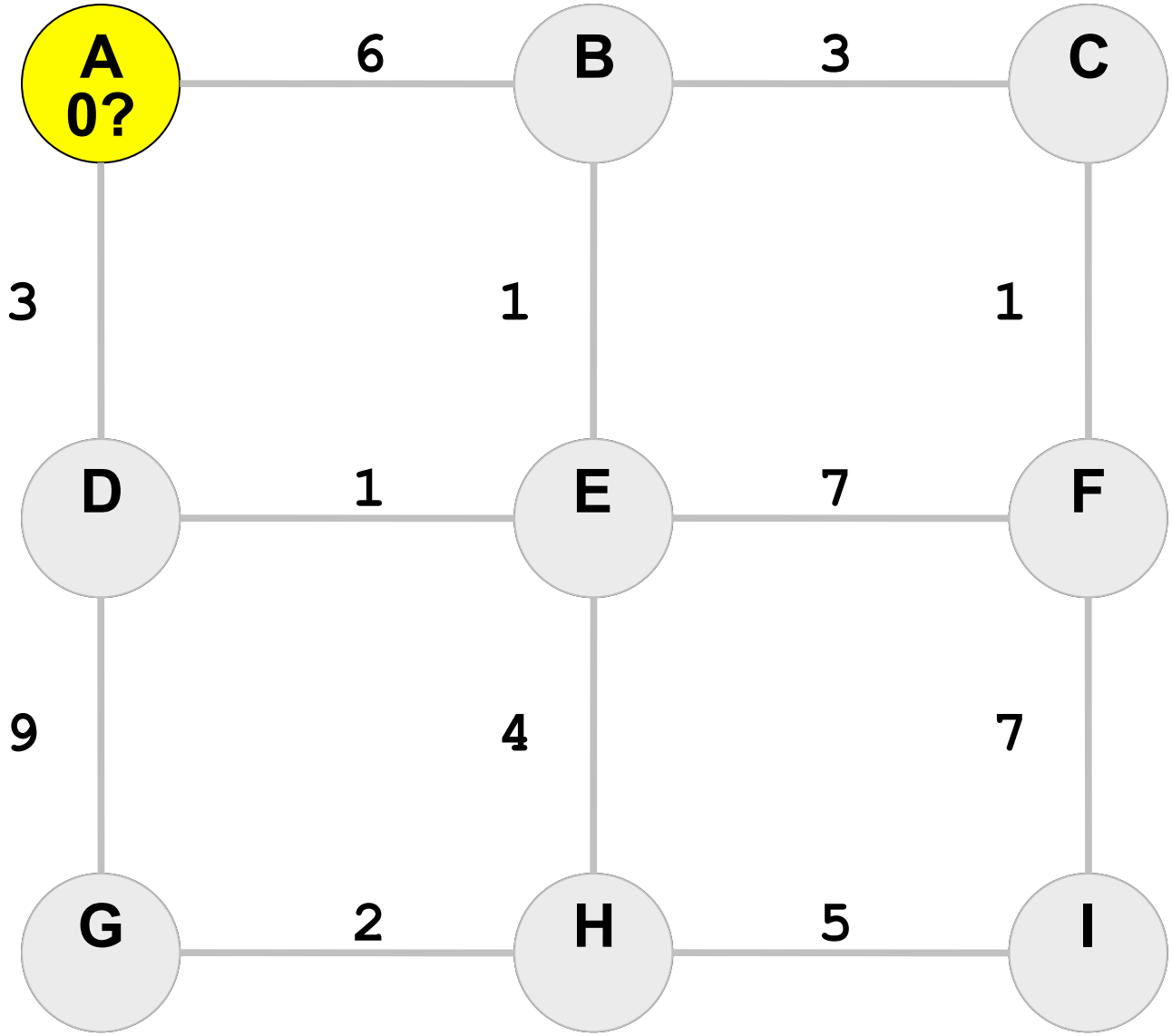
How You Imagine Edsger Dikstra

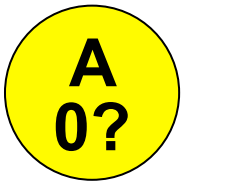


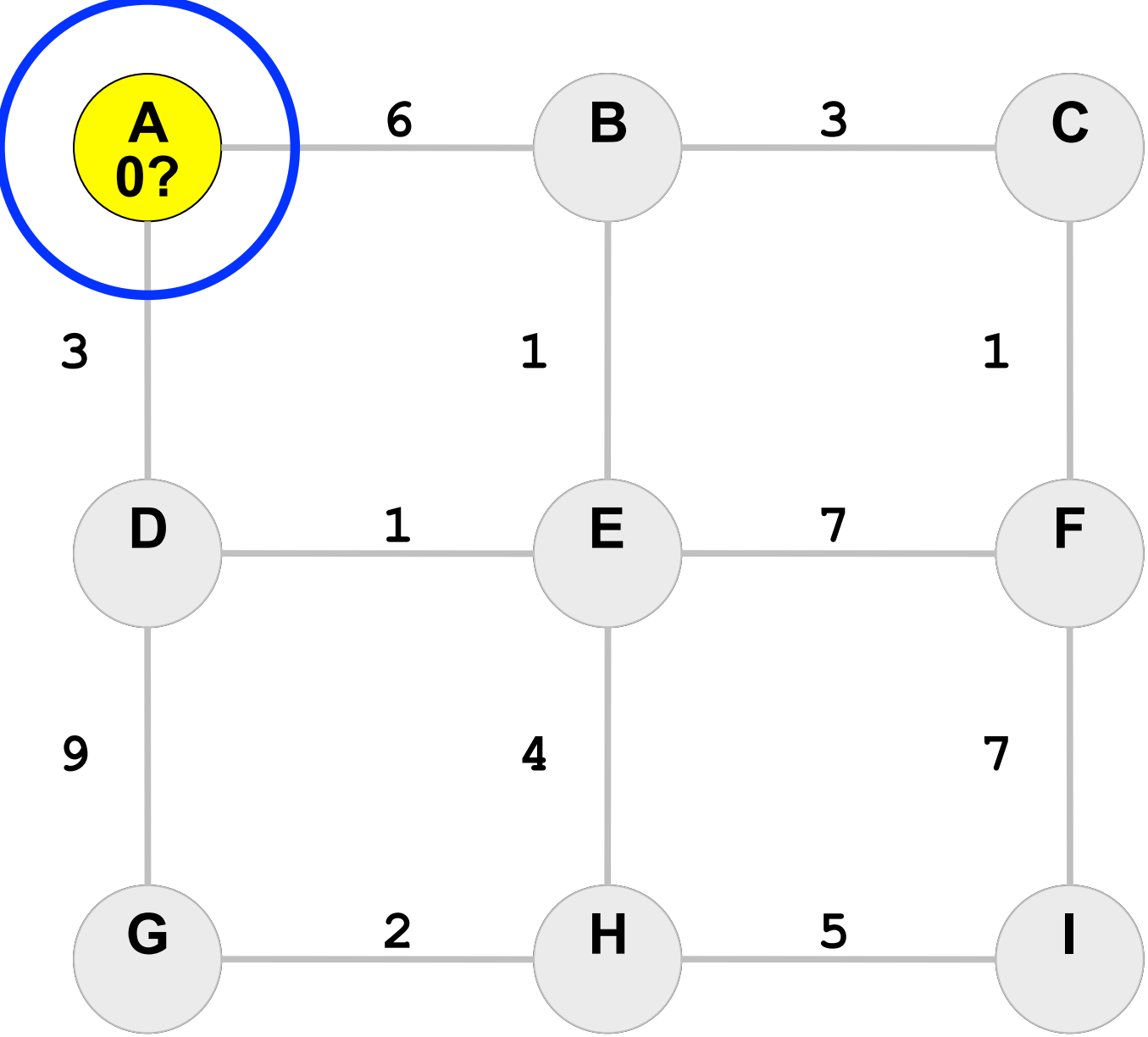
Edsger Dikstra

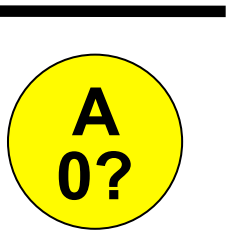

A
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I

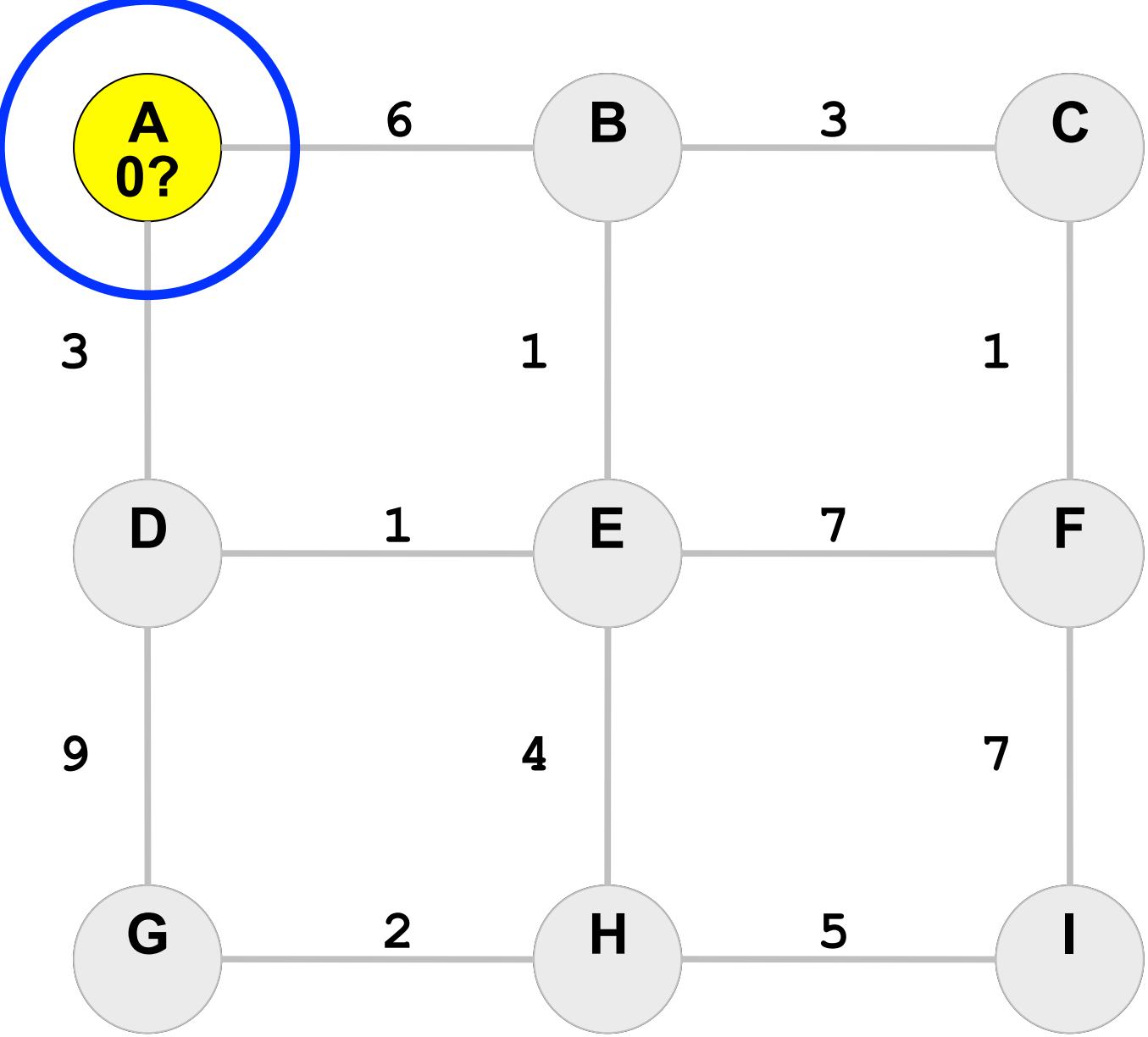

A
0?
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I

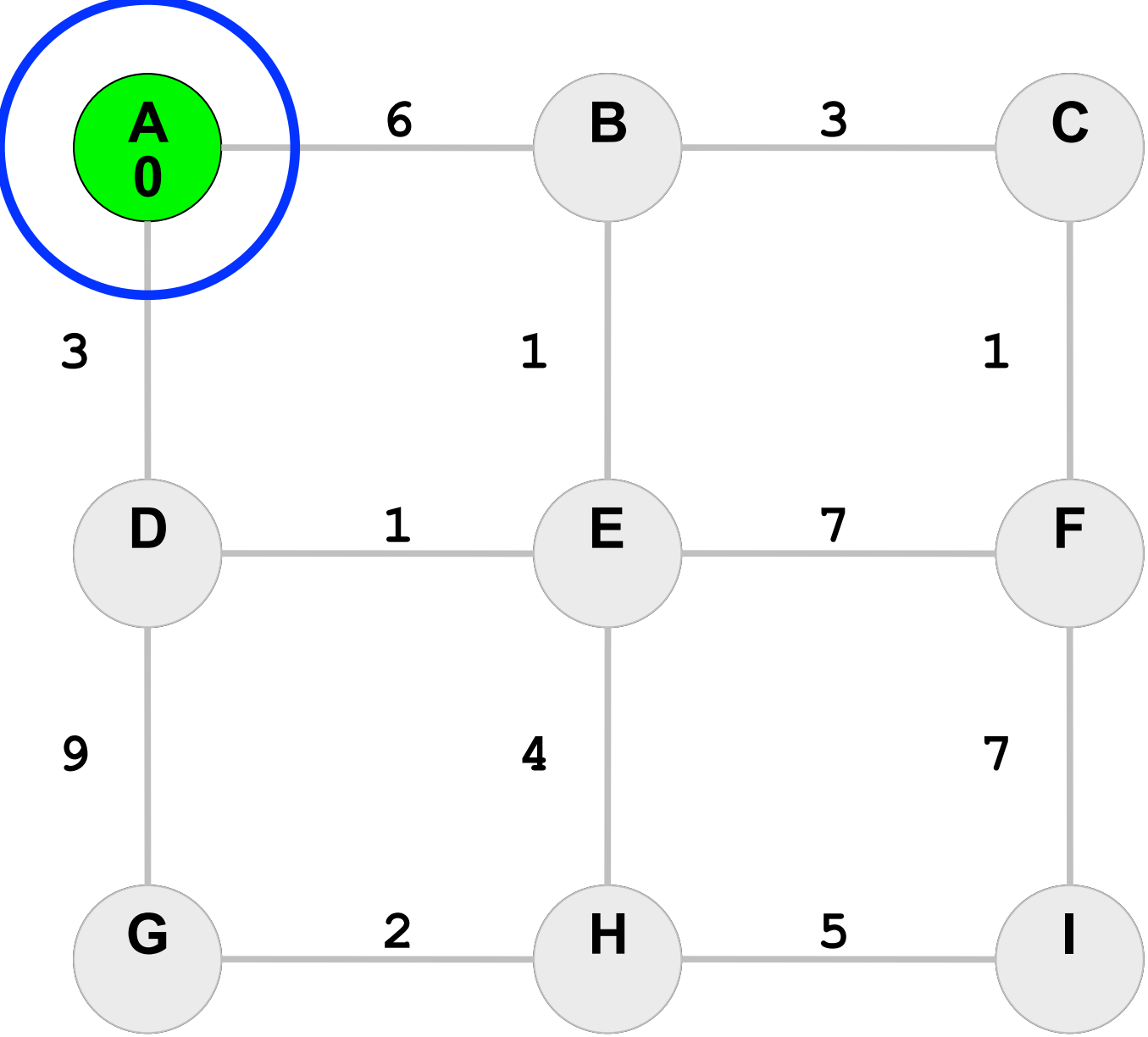

A
0?
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I

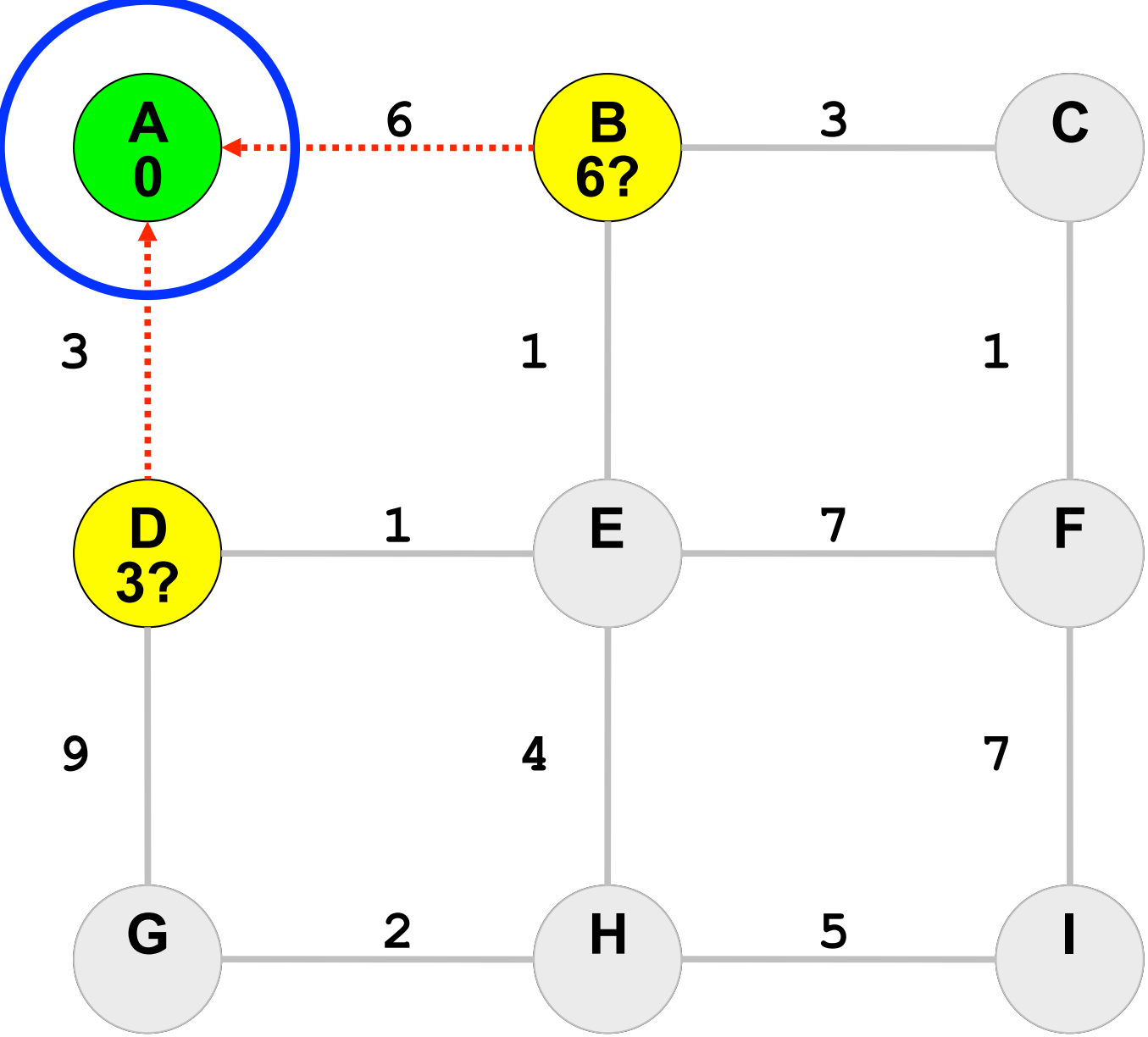

A
0?

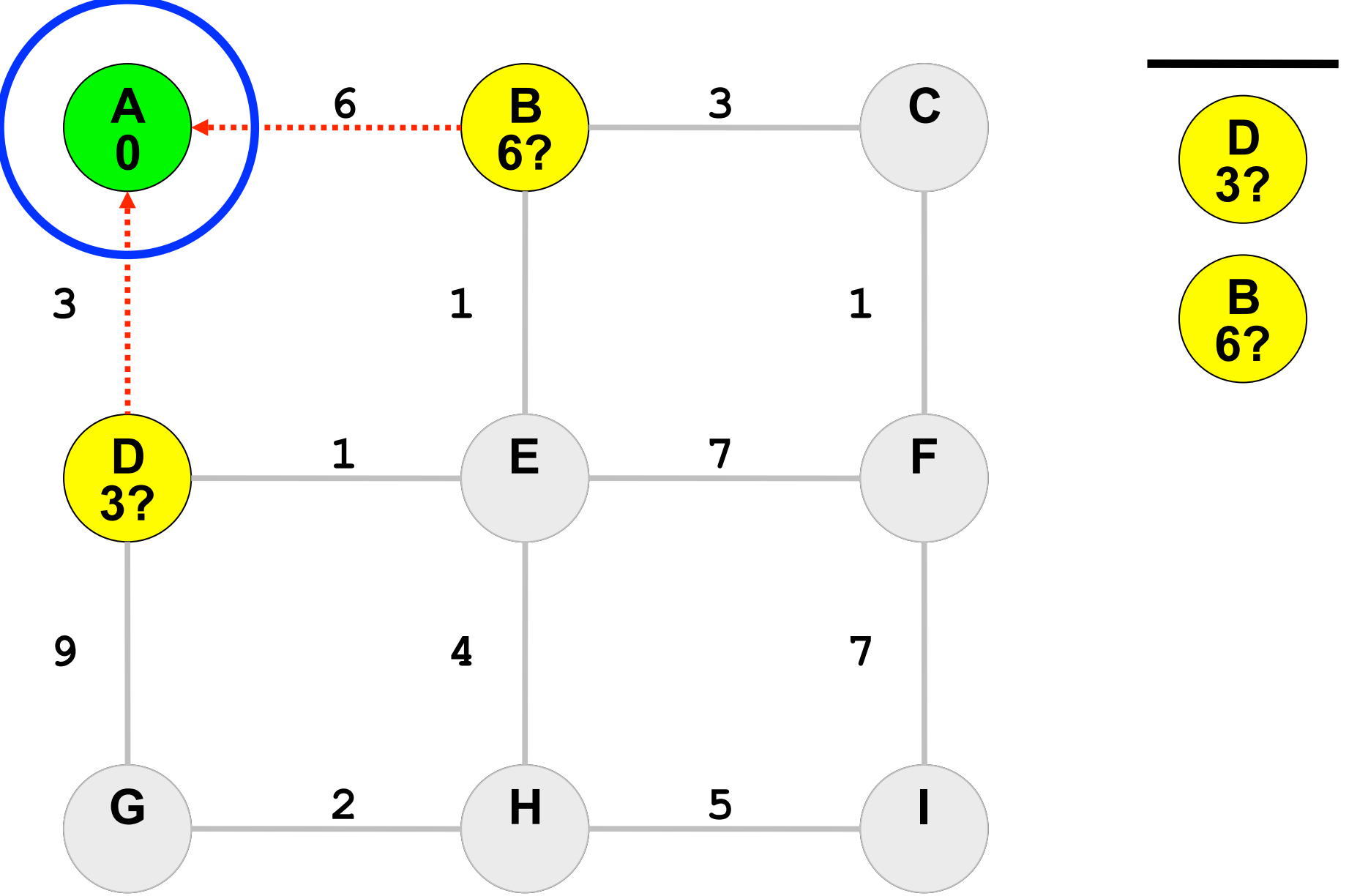

A
0?
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I

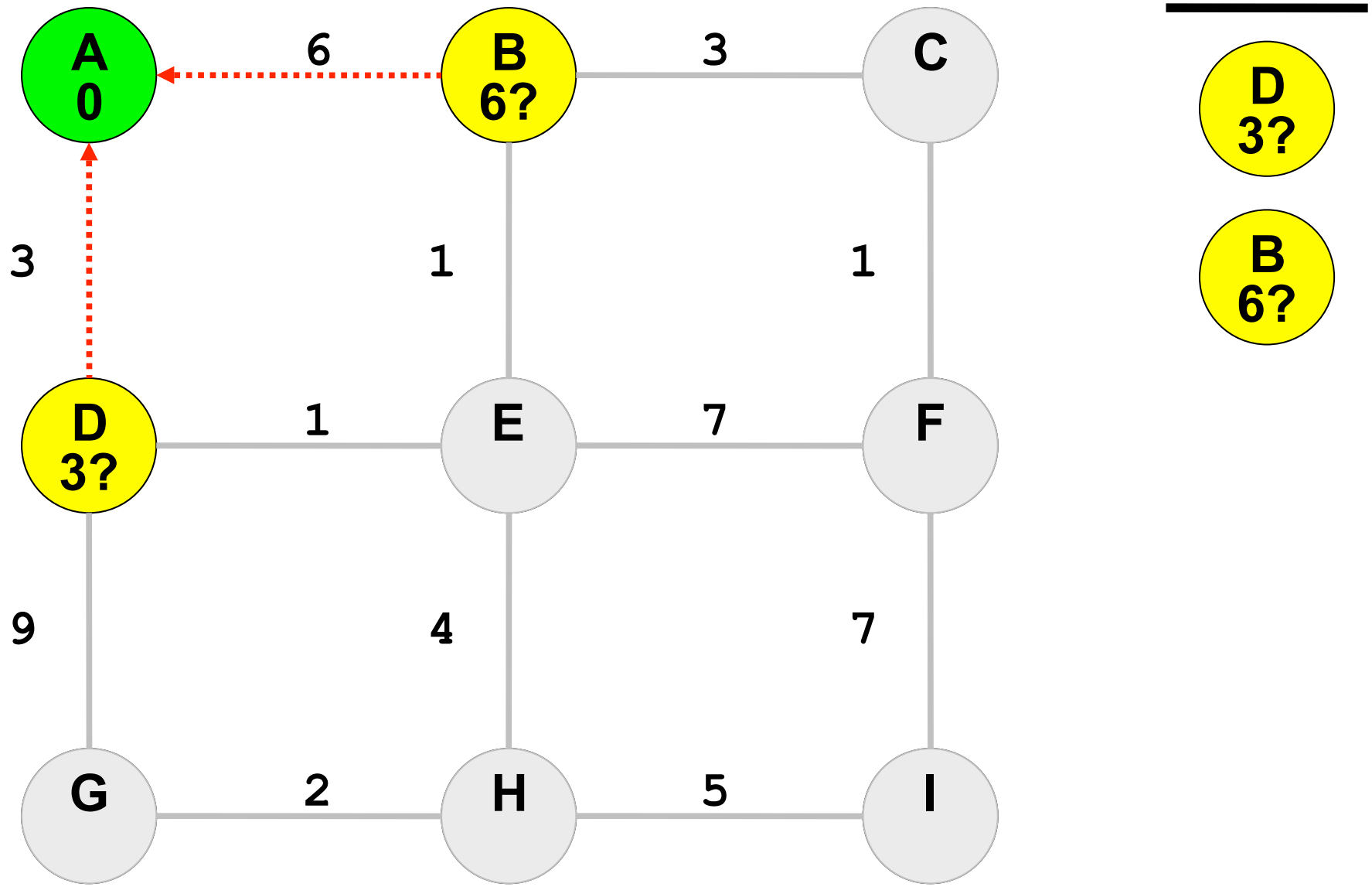

A
0?

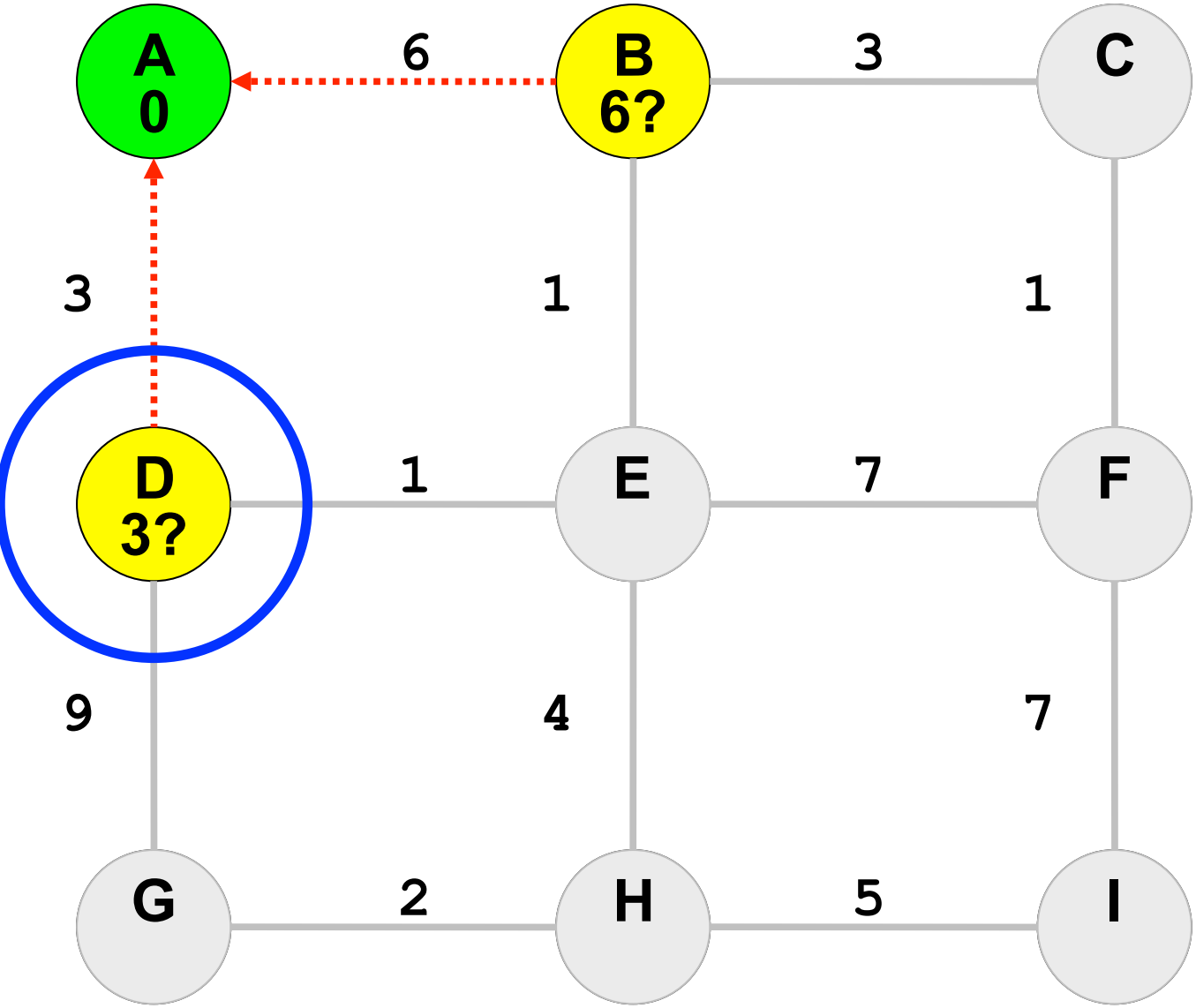

A
0?
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I


A
0
6
B
3
C
3
1
1
D
1
E
7
F
9
4
7
G
2
H
5
I


A
0
6
B
6?
3
C
3
1
1
D
3?
1
E
7
F
9
4
7
G
2
H
5
I

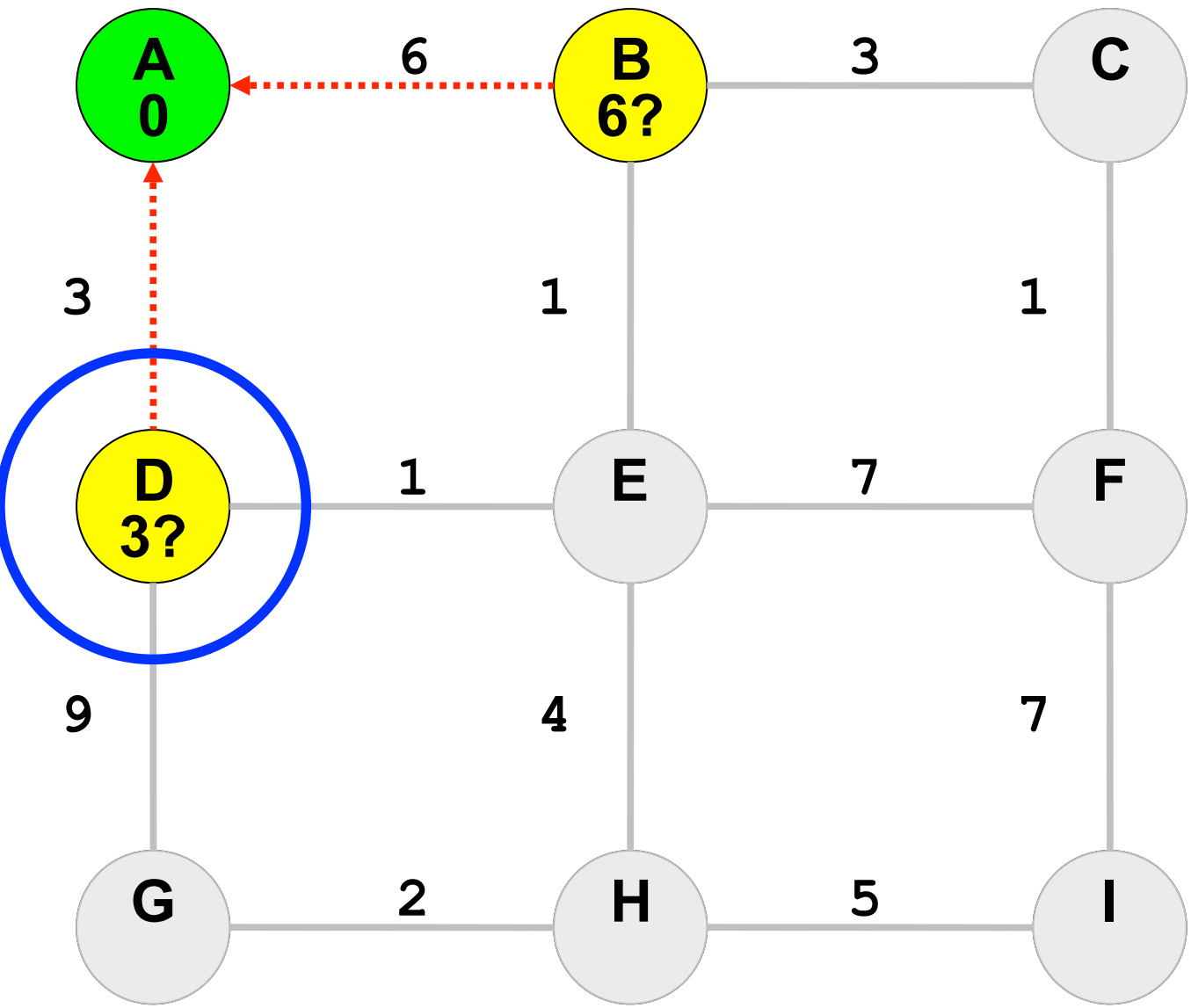

A
0
6
B
6?
3
C
3
1
1
D
3?
1
E
7
F
9
4
7
G
2
H
5
I
D
3?
B
6?

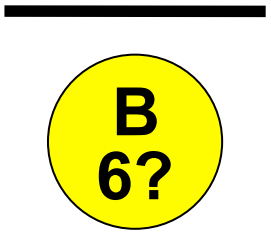

A
0
6
B
6?
3
C
3
1
1
D
3?
1
E
7
F
9
4
7
G
2
H
5
I
D
3?
B
6?

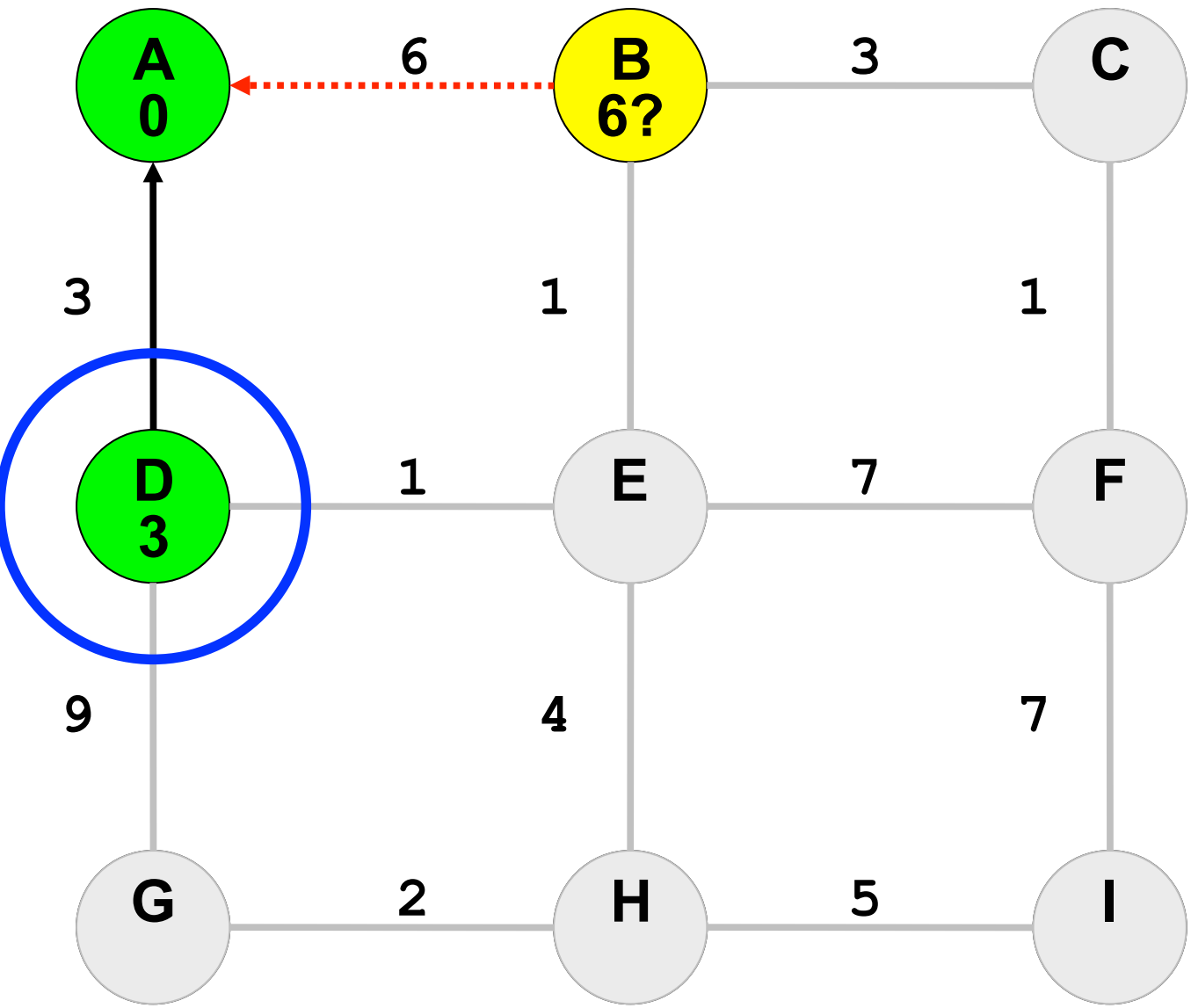


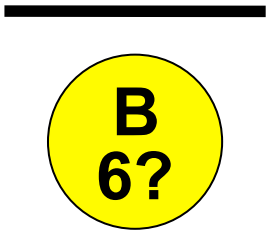
D
3?

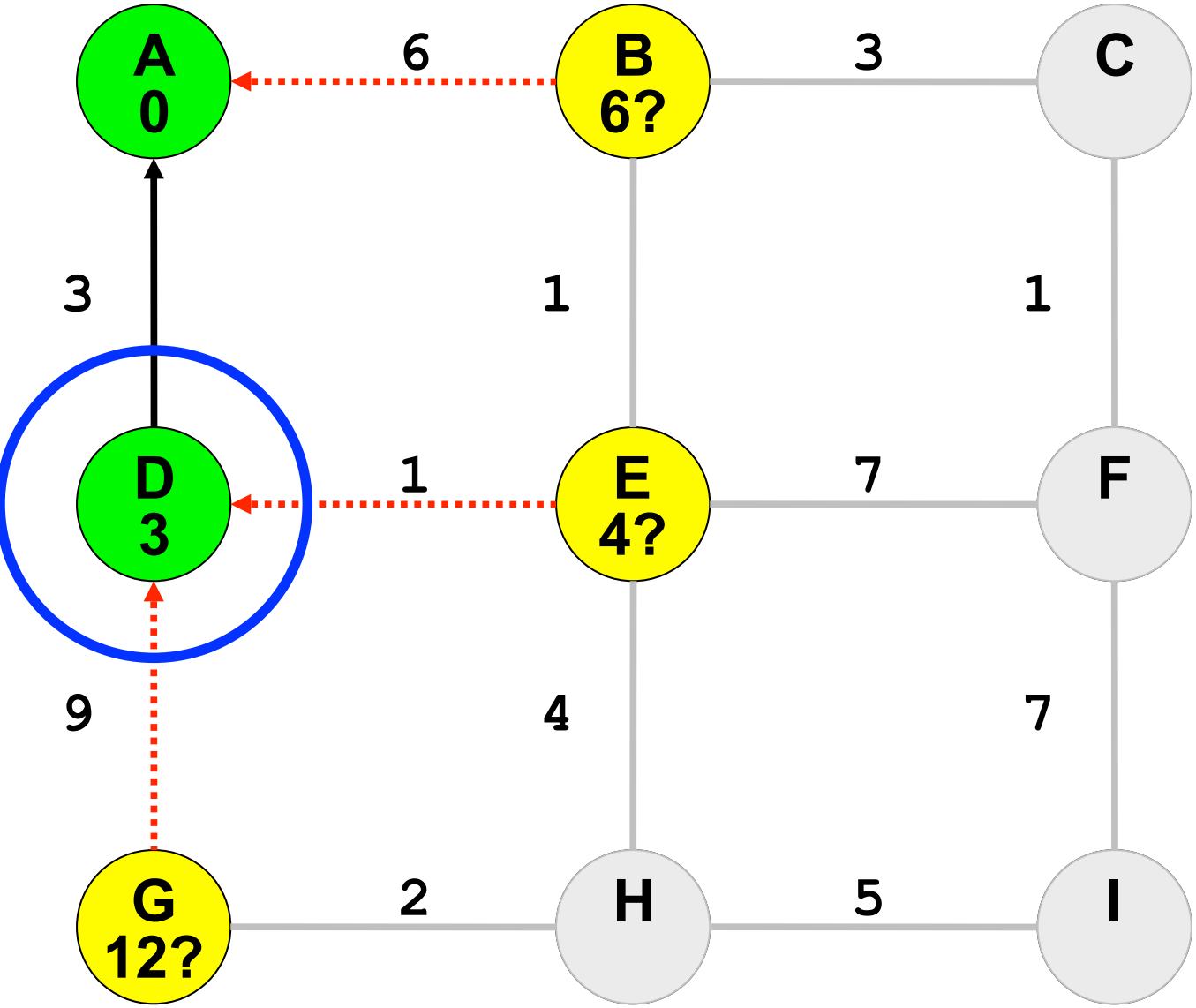
B
6?


A
0
6
B
6?
3
C
3
1
1
D
3?
1
E
7
F
9
4
7
G
2
H
5
I

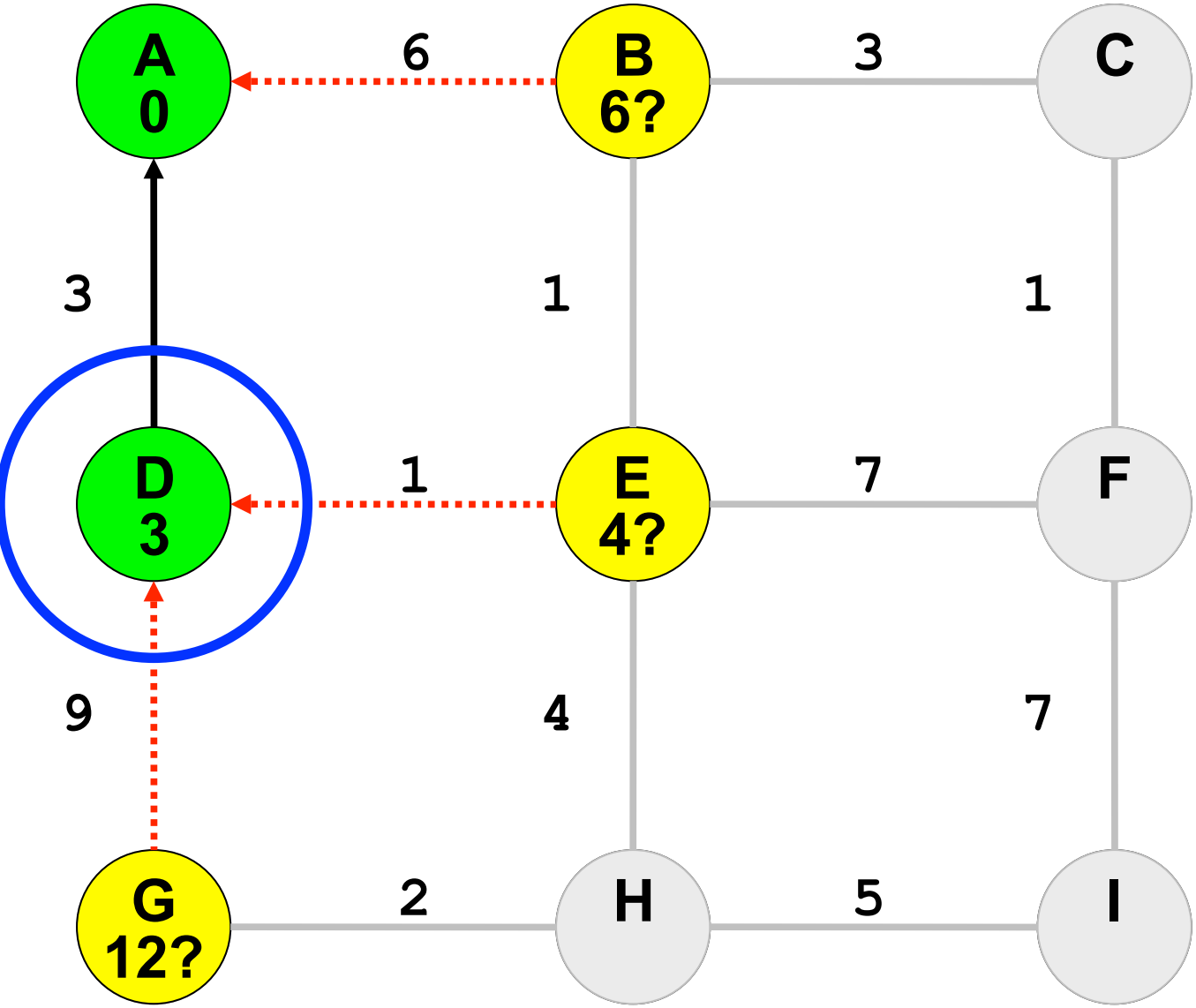

B
6?


A
0
6
B
6?
3
C
3
1
1
D
3
1
E
7
F
9
4
7
G
2
H
5
I

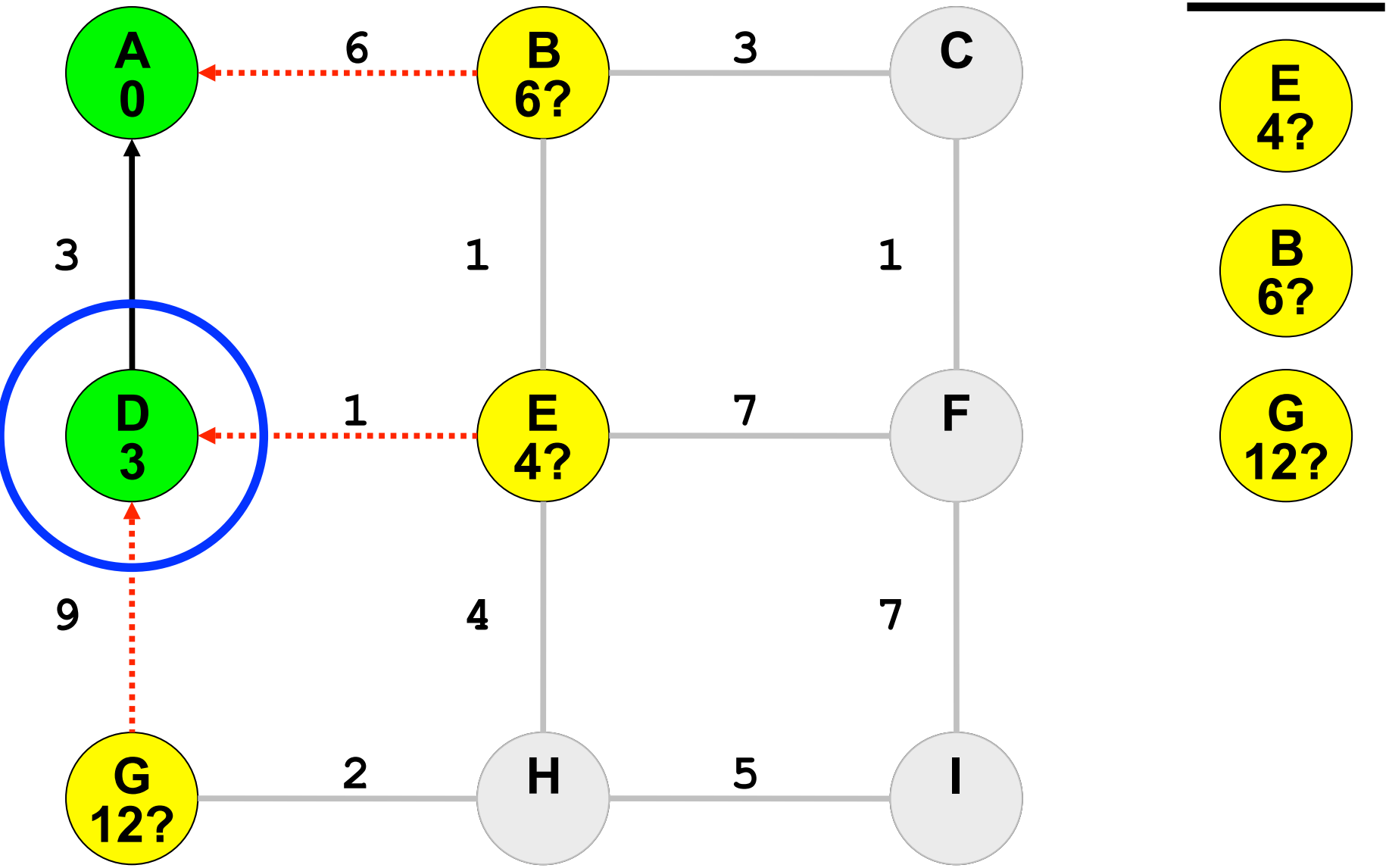

B
6?

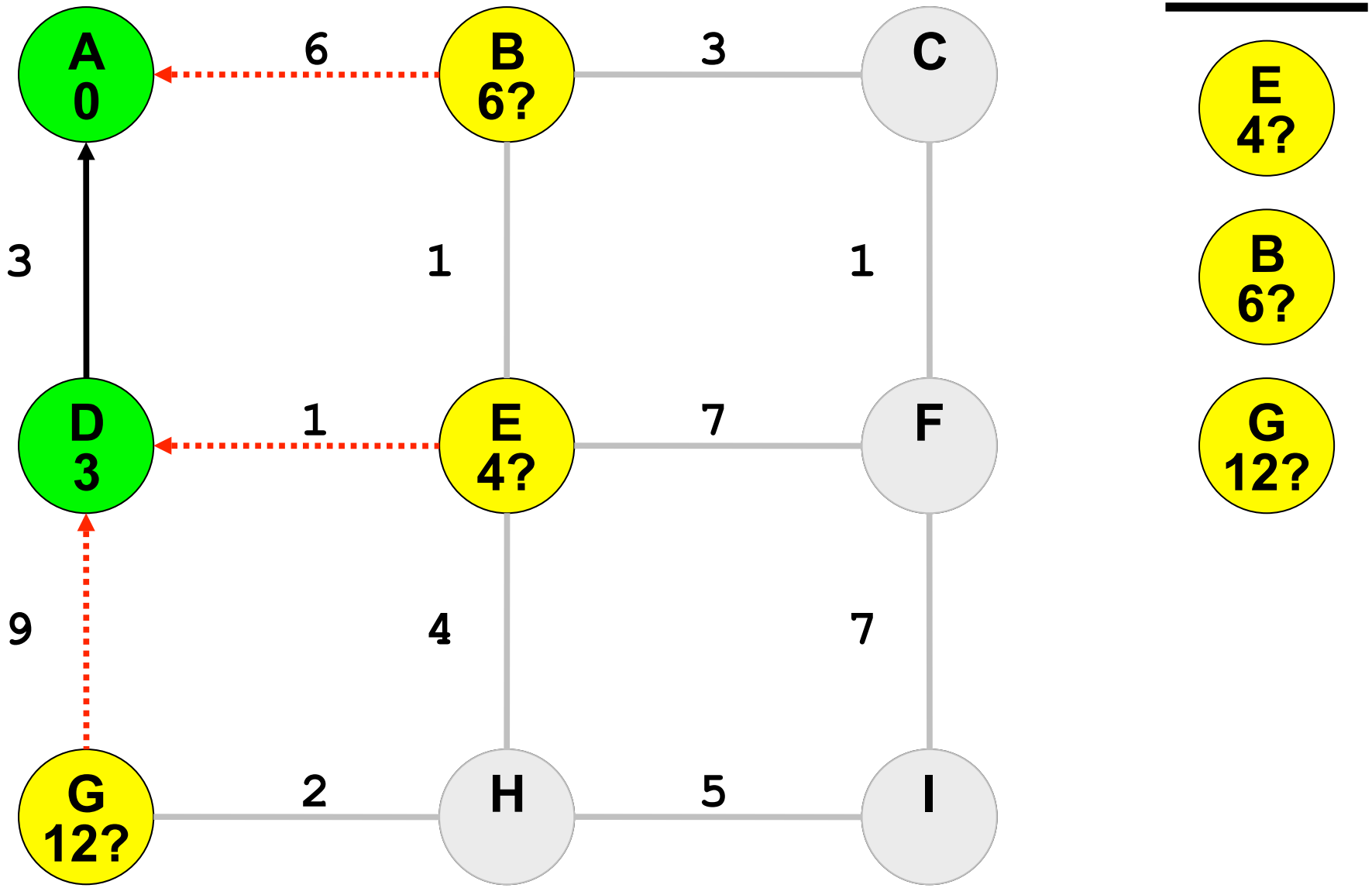

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4?
7
F
9
4
7
G
12?
2
H
5
I

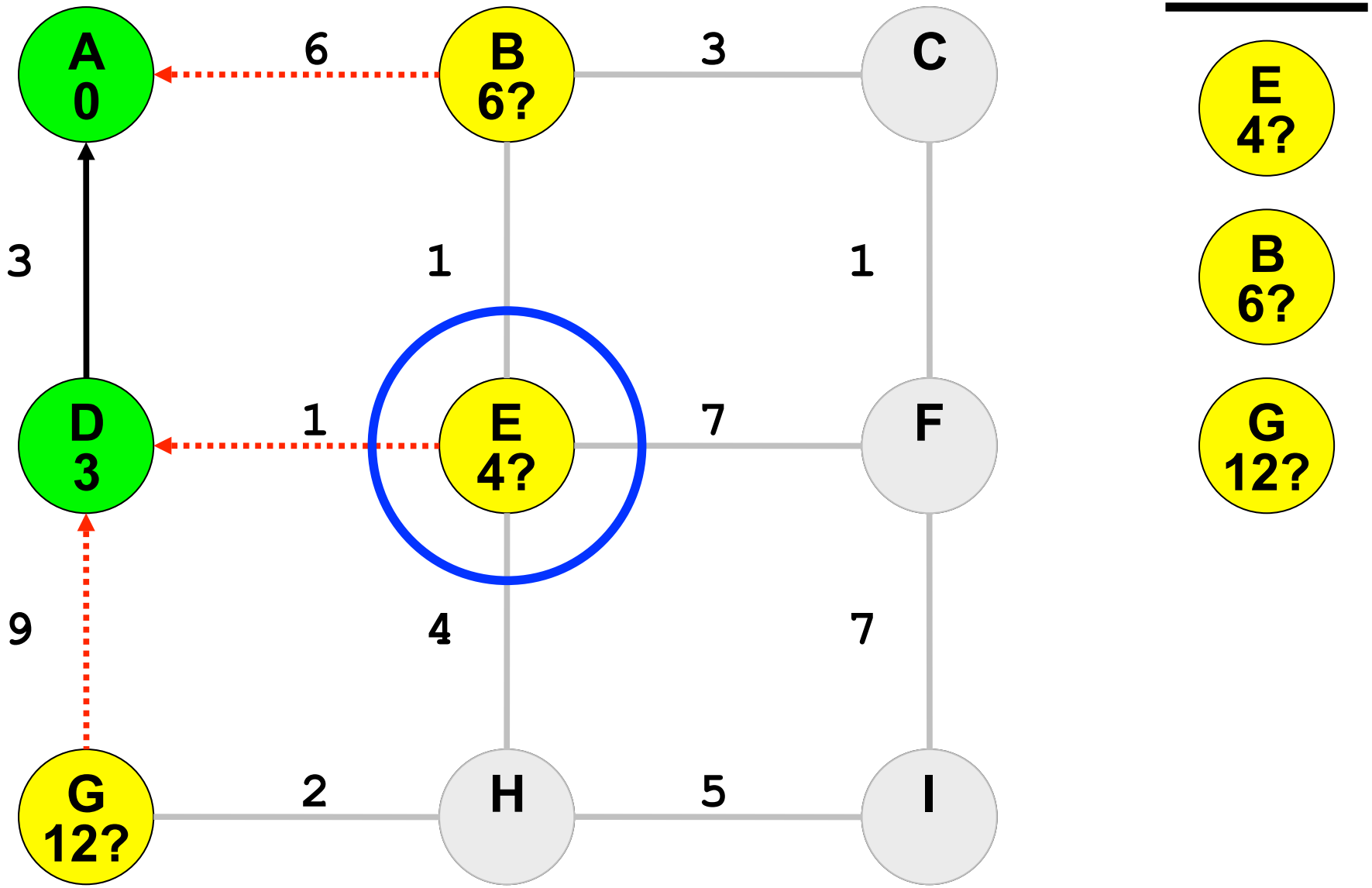

B
6?

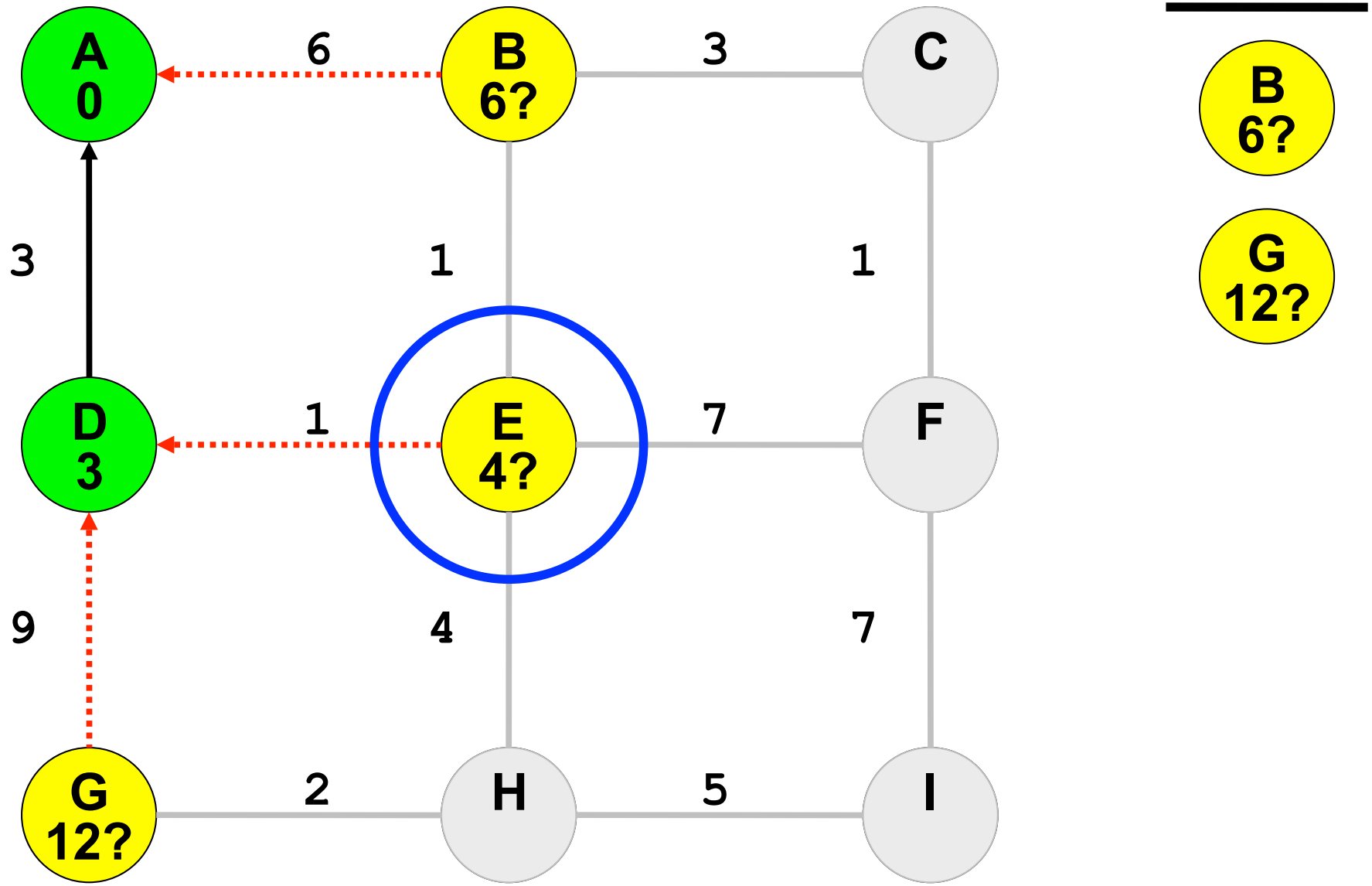


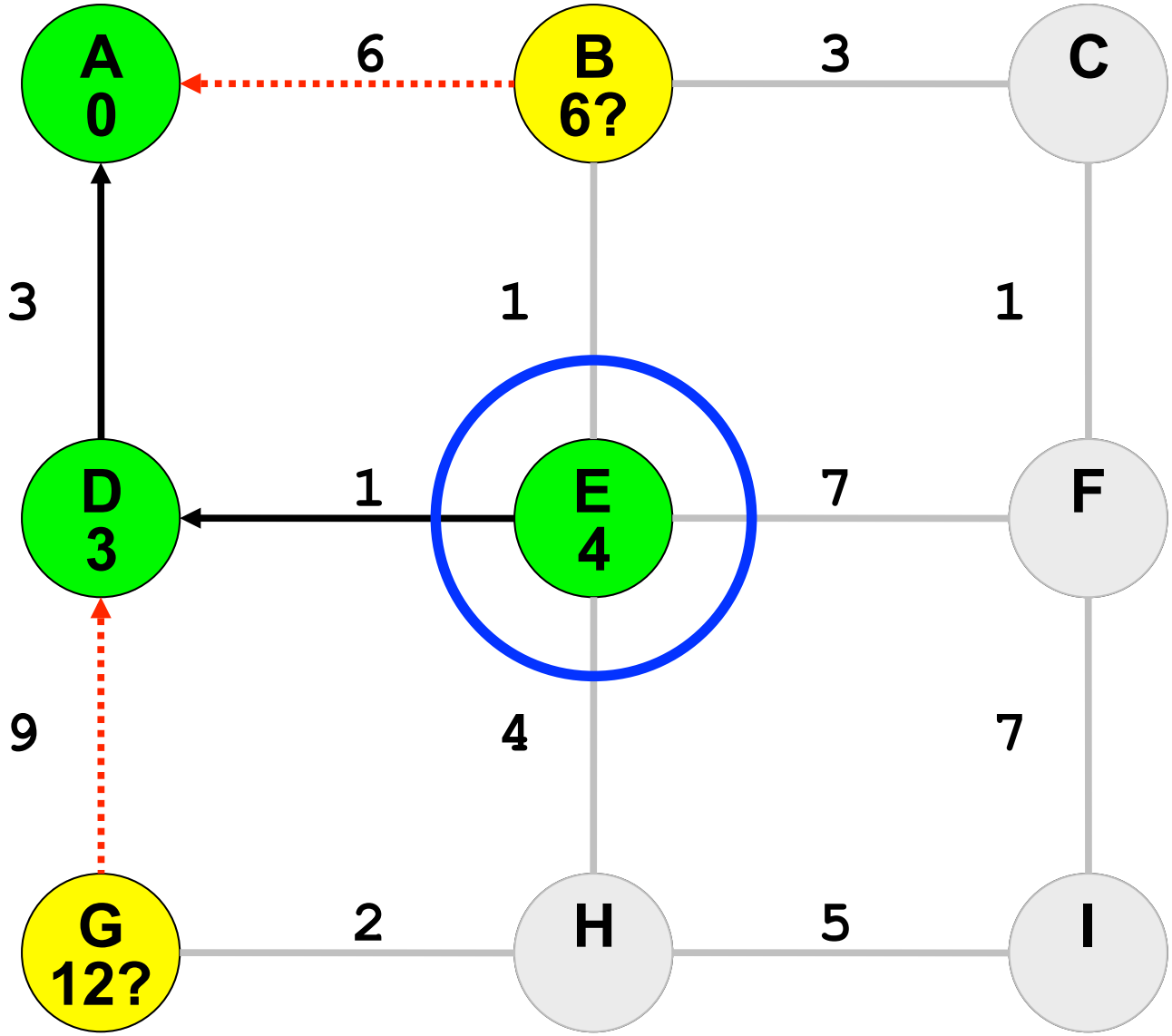
B
6?


A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4?
7
F
9
4
7
G
12?
2
H
5
I
E
4?
B
6?
G
12?


A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4?
7
F
9
4
7
G
12?
2
H
5
I
E
4?
B
6?
G
12?

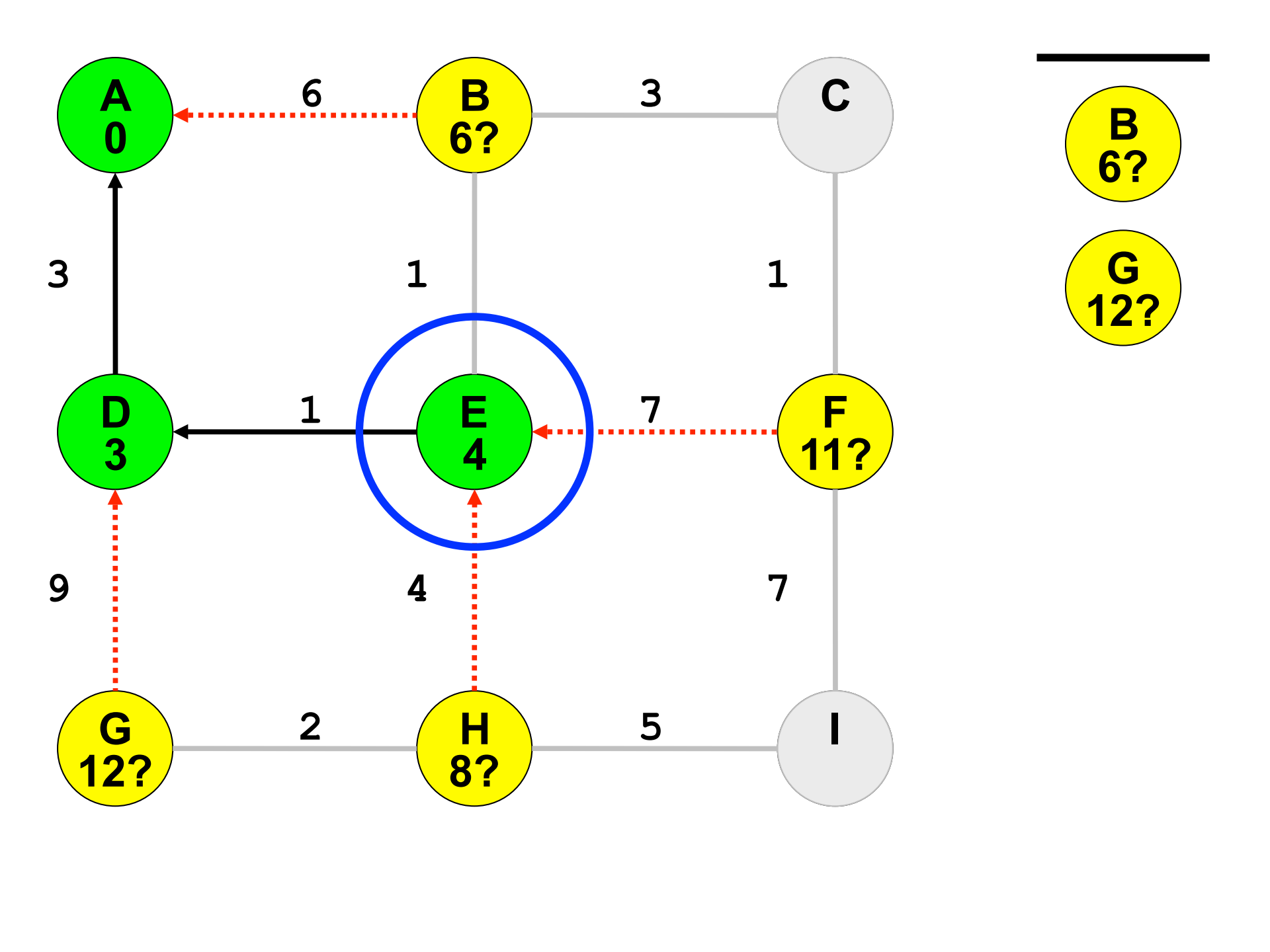

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4?
7
F
9
4
7
G
12?
2
H
5
I
E
4?
B
6?
G
12?

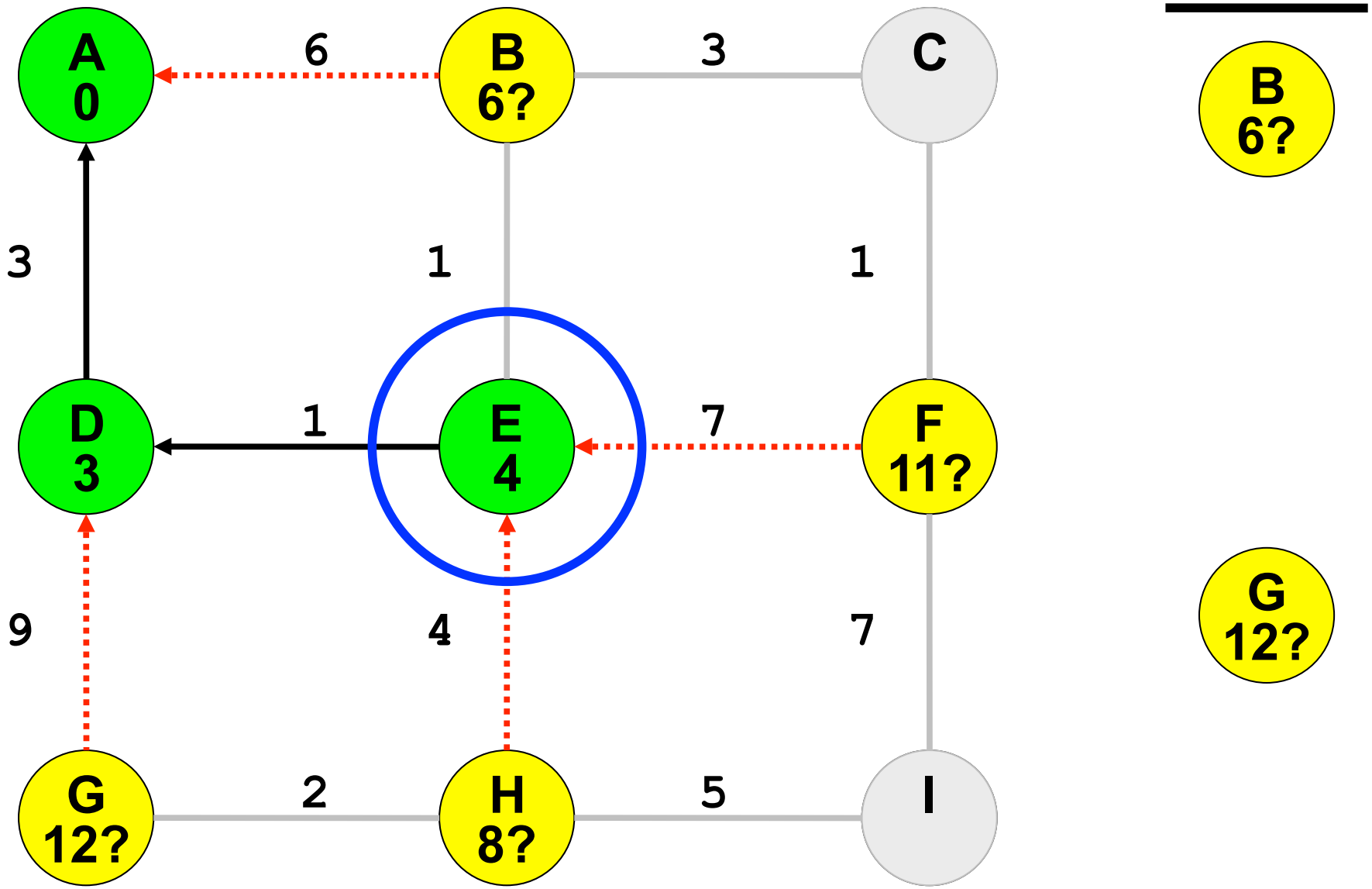

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4?
7
F
9
4
7
G
12?
2
H
5
I
B
6?
G
12?

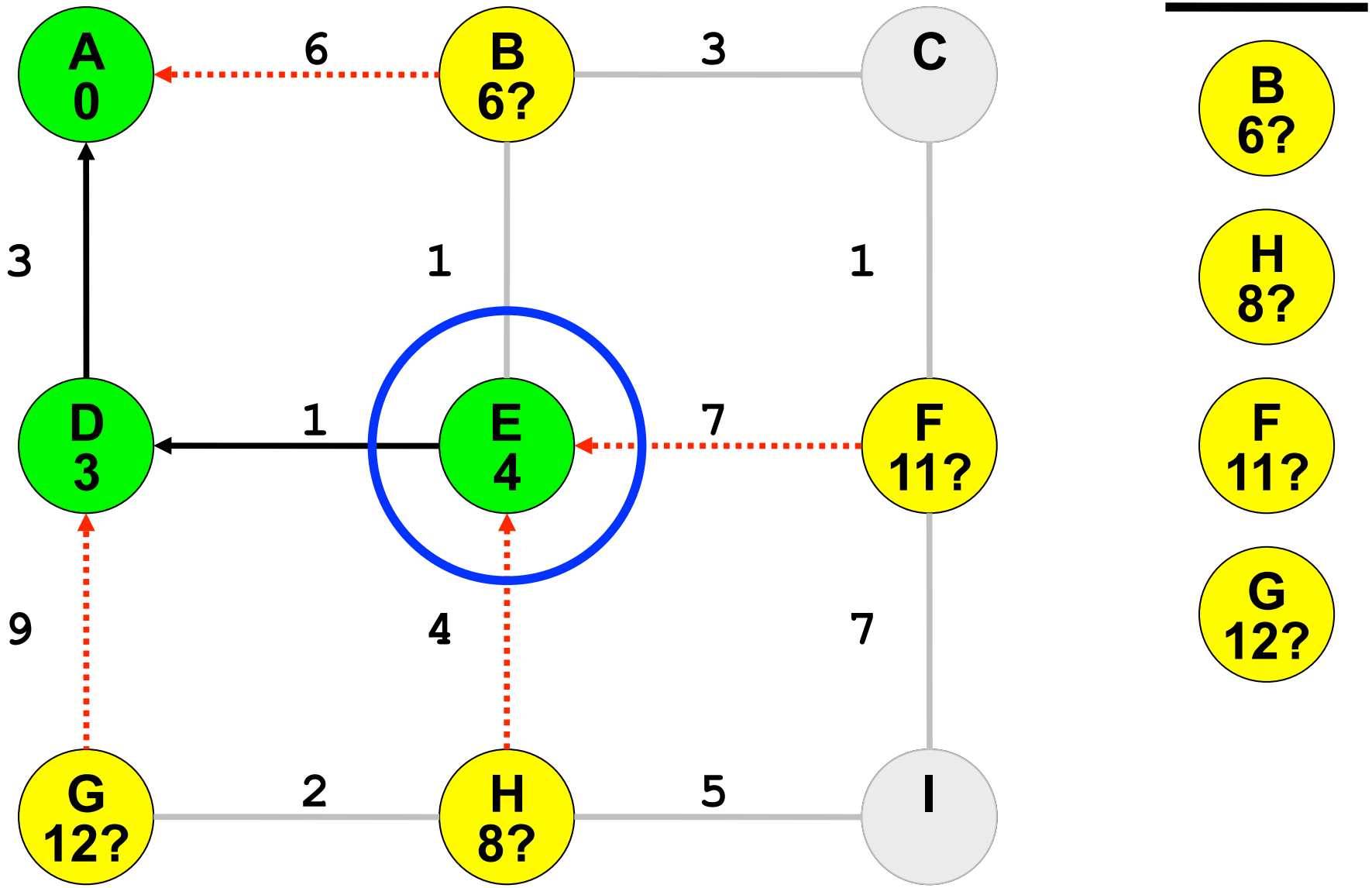


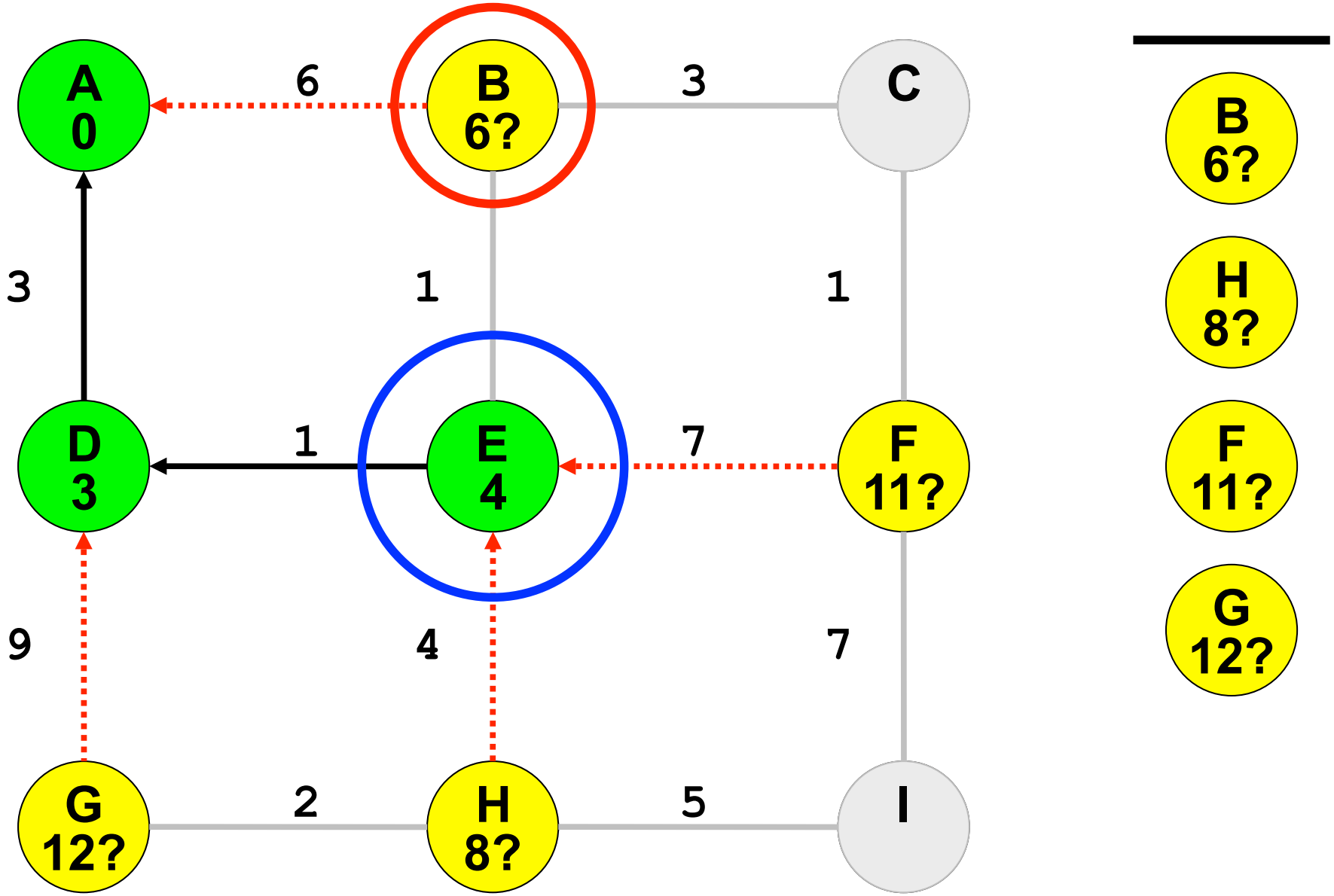
B
6?

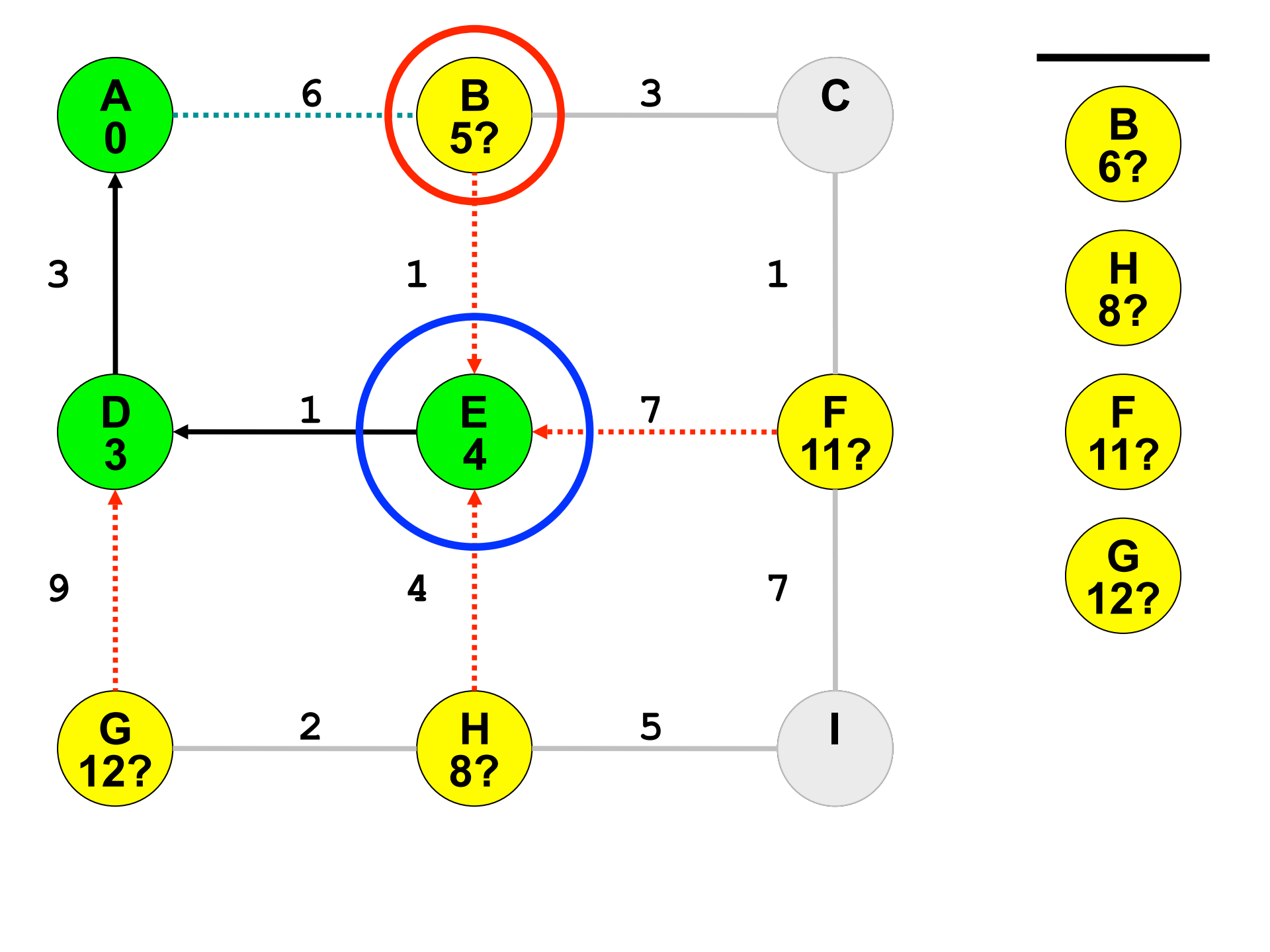
G
12?

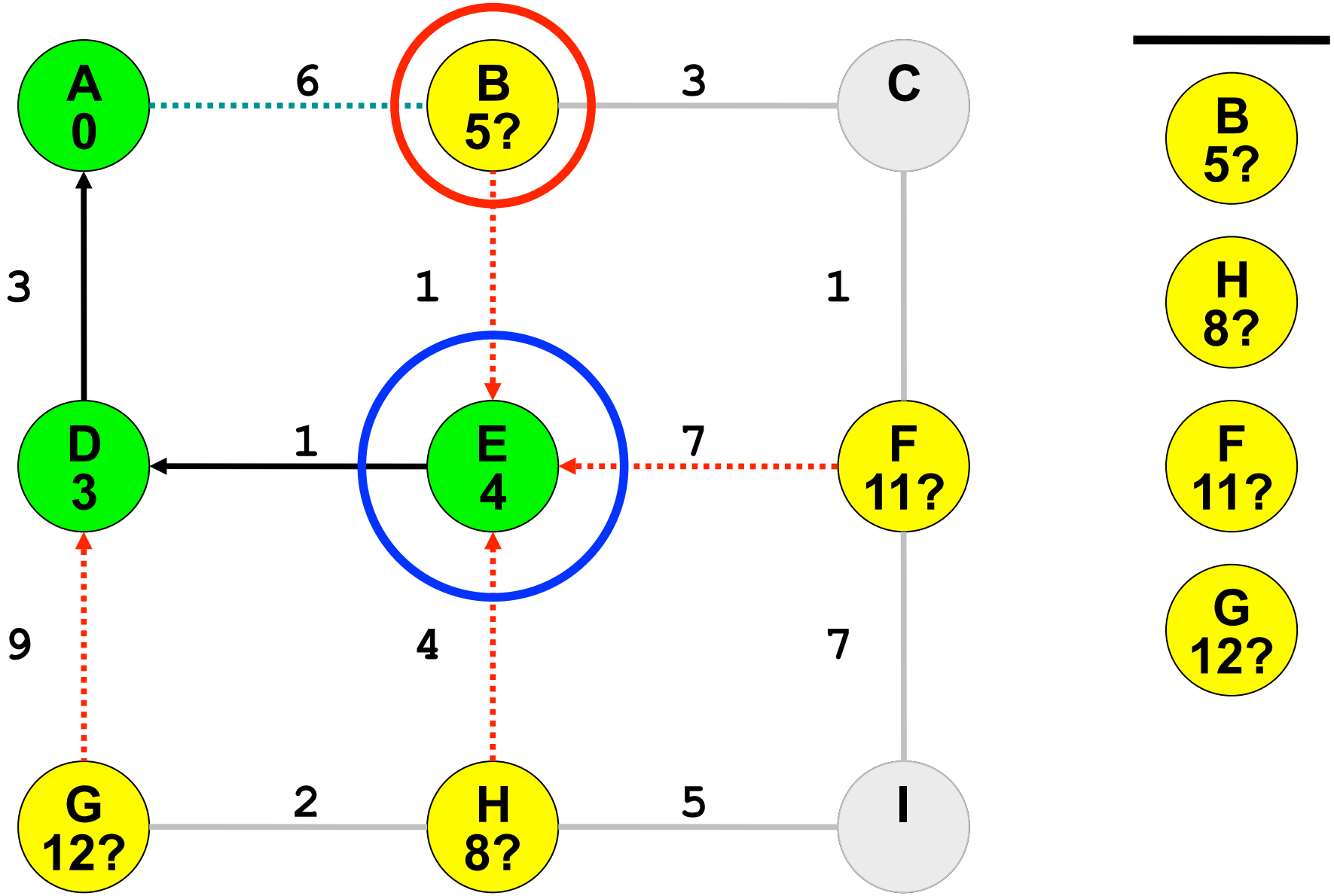

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
6?
G
12?

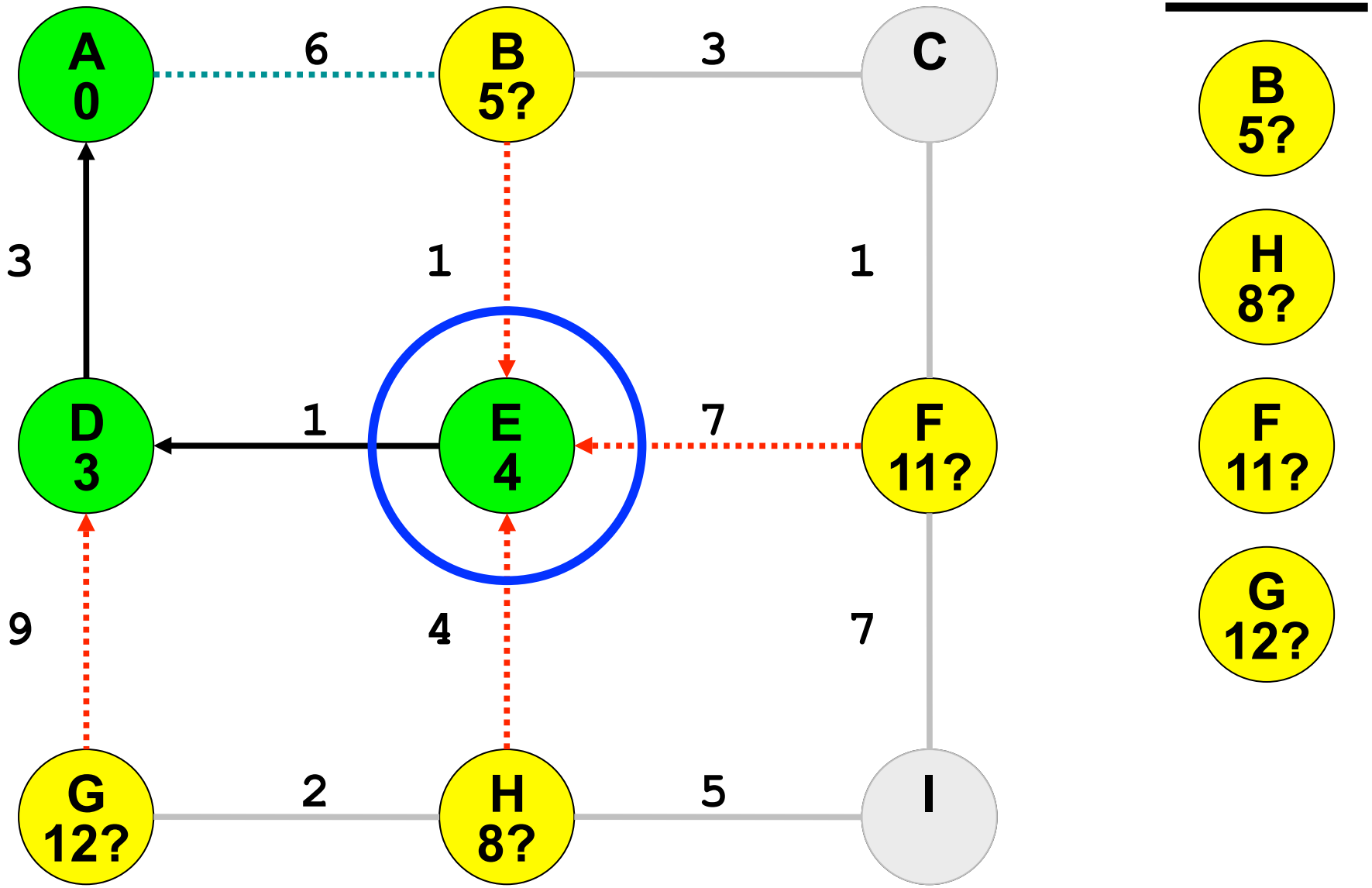

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
6?
G
12?

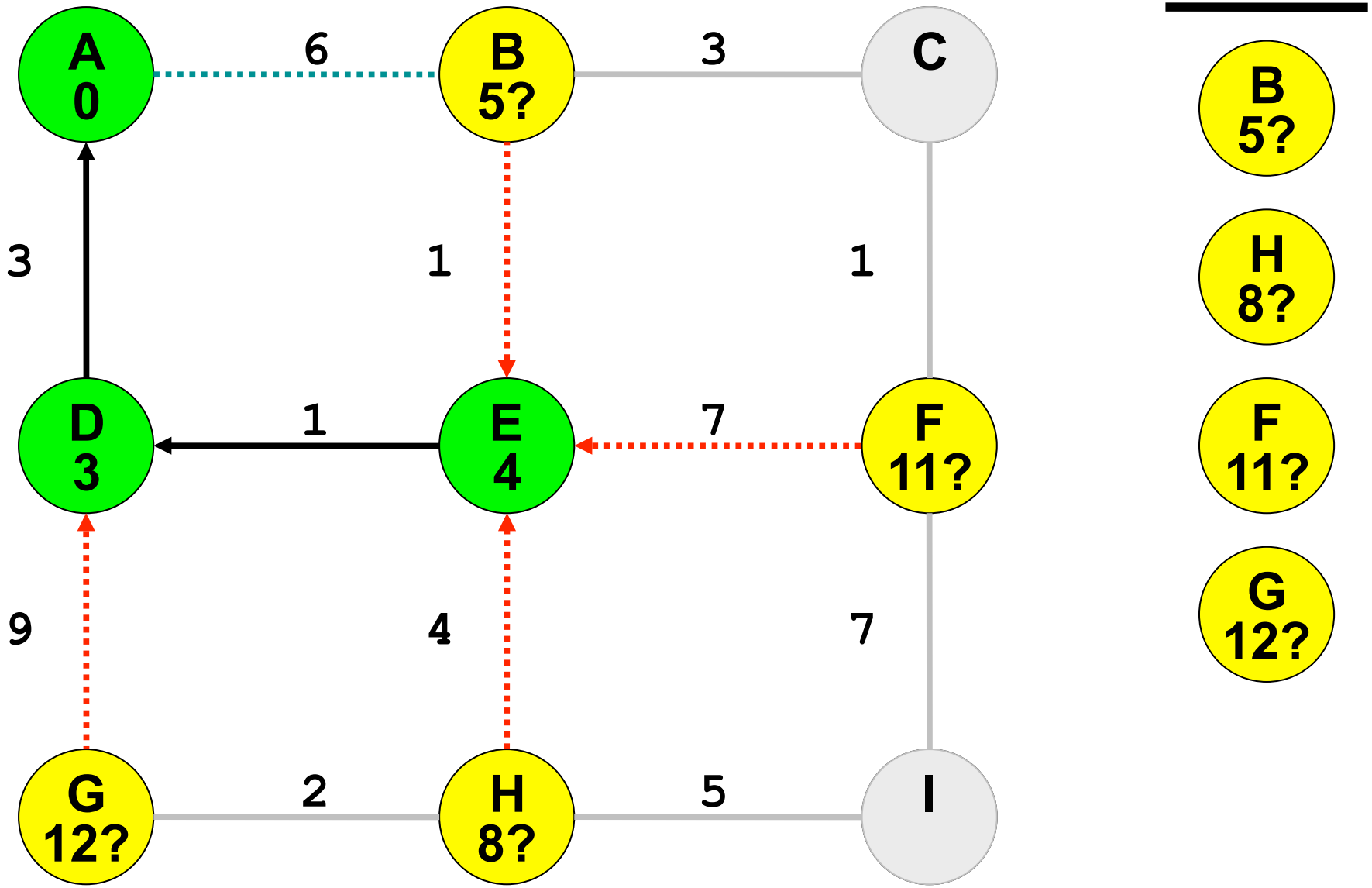

A
0
6
B
6?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
6?
H
8?
F
11?
G
12?

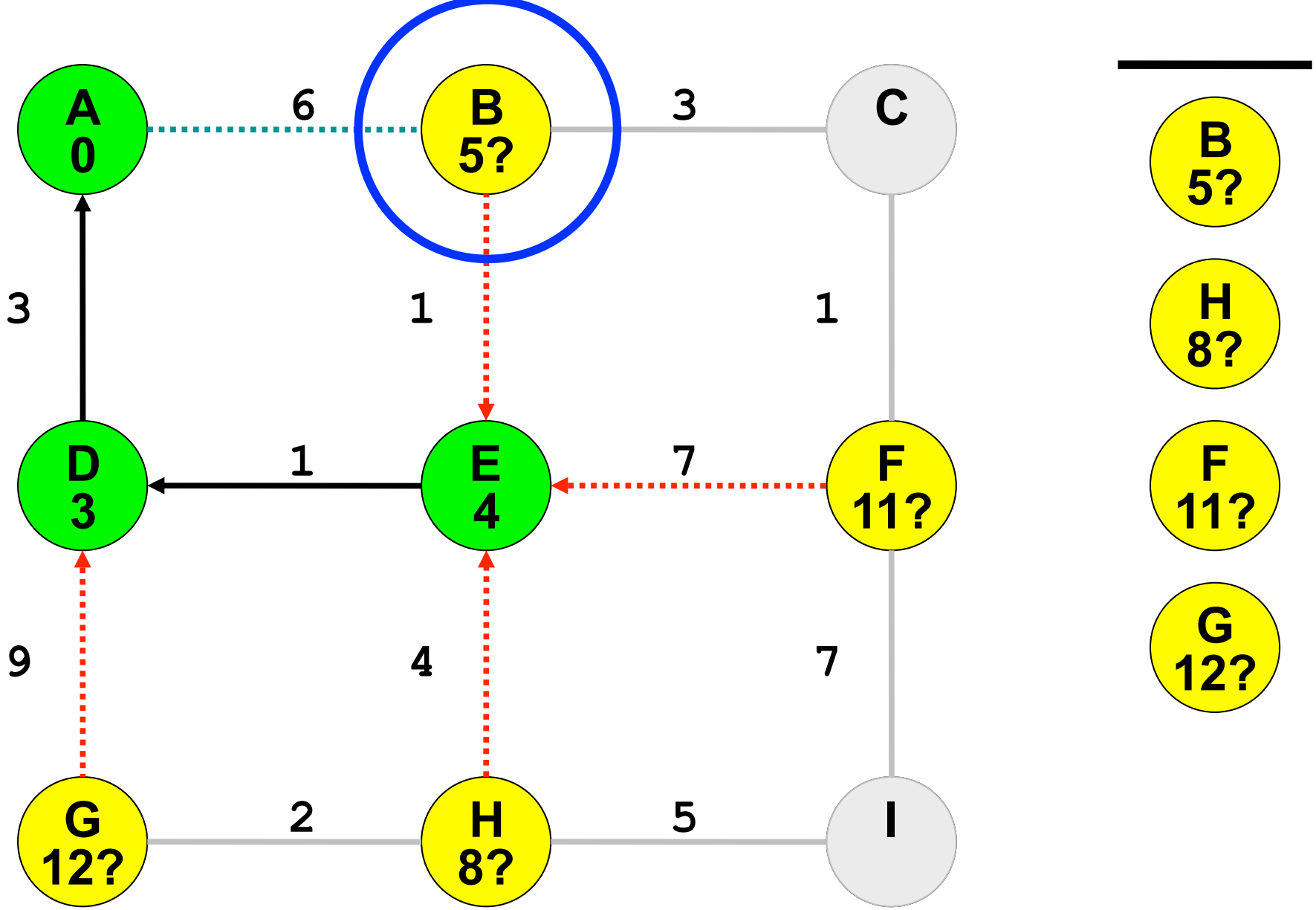

A
0
B
6?
C
D
3
E
4
F
11?
G
12?
H
8?
I
6
3
3
1
1
1
7
9
4
7
2
5
B
6?
H
8?
F
11?
G
12?

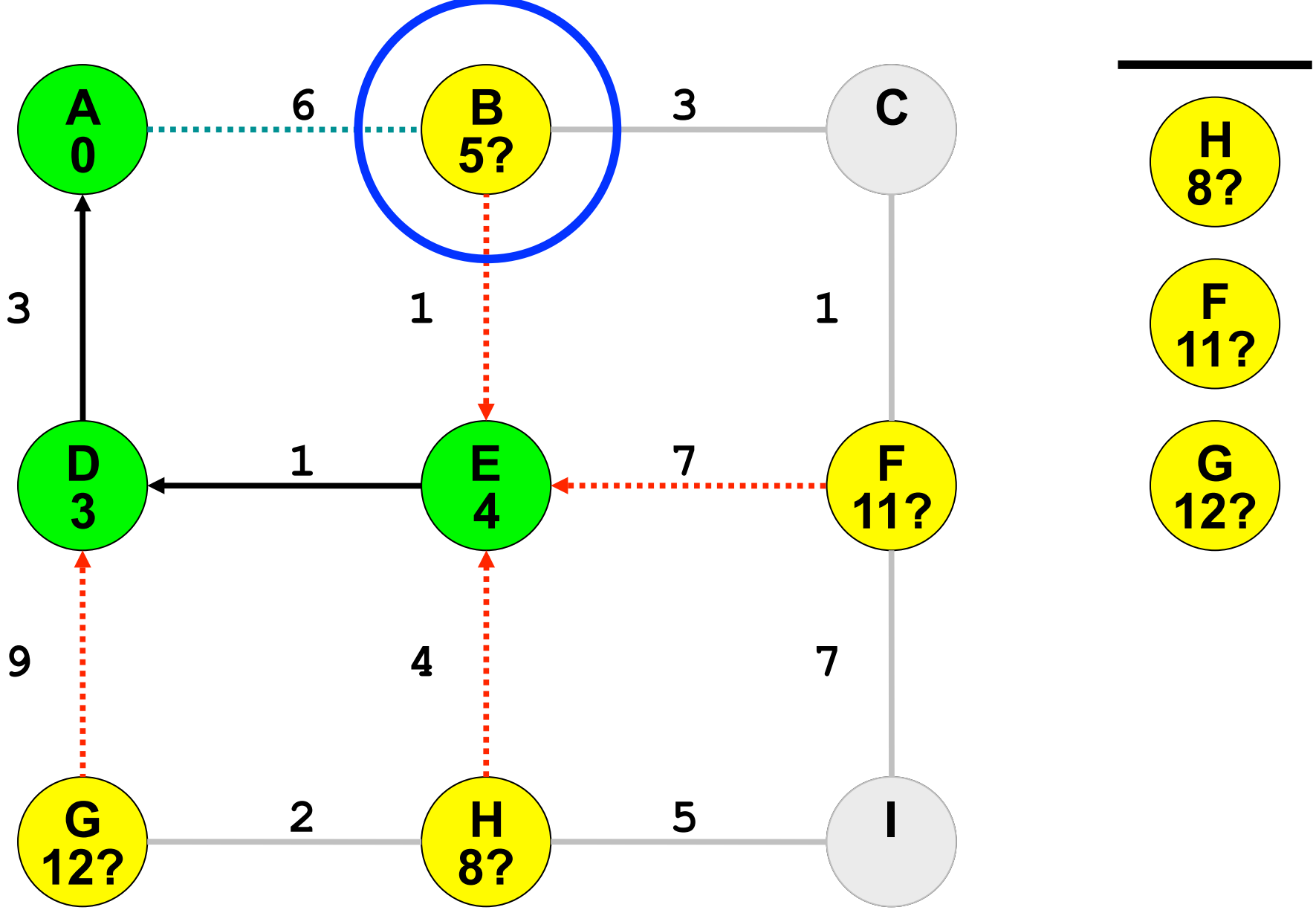

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
6?
H
8?
F
11?
G
12?

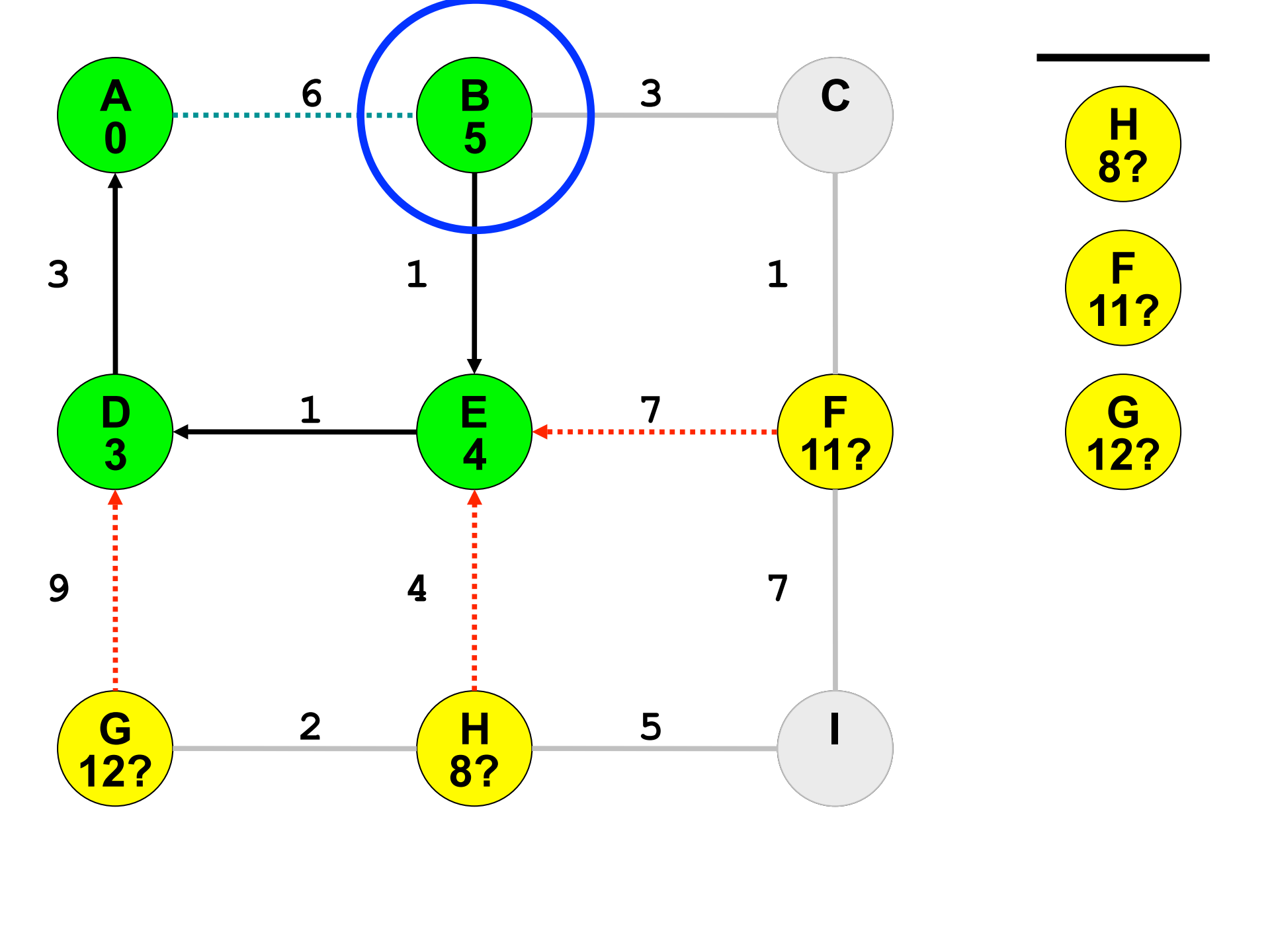

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
5?
H
8?
F
11?
G
12?

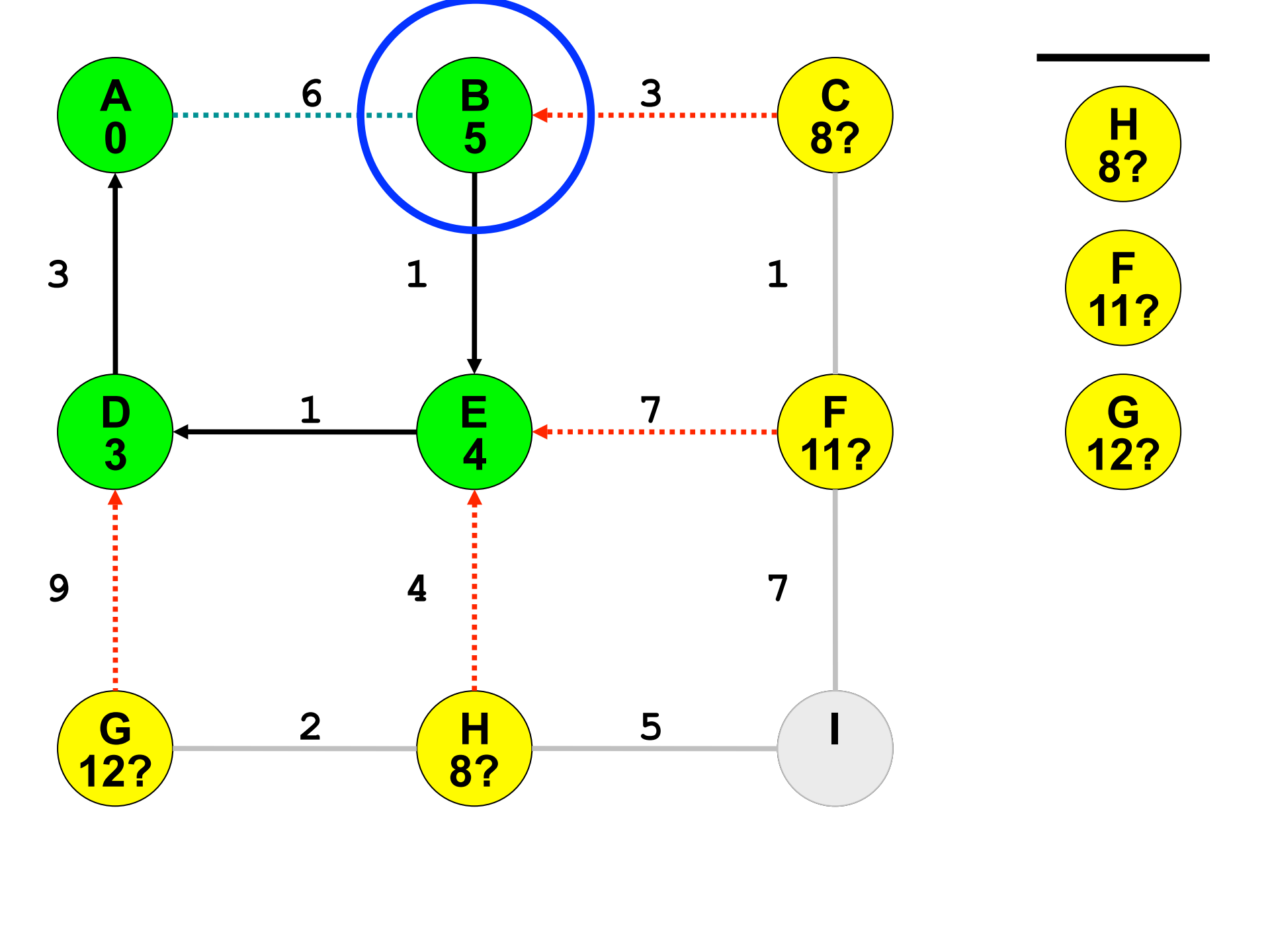

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
5?
H
8?
F
11?
G
12?

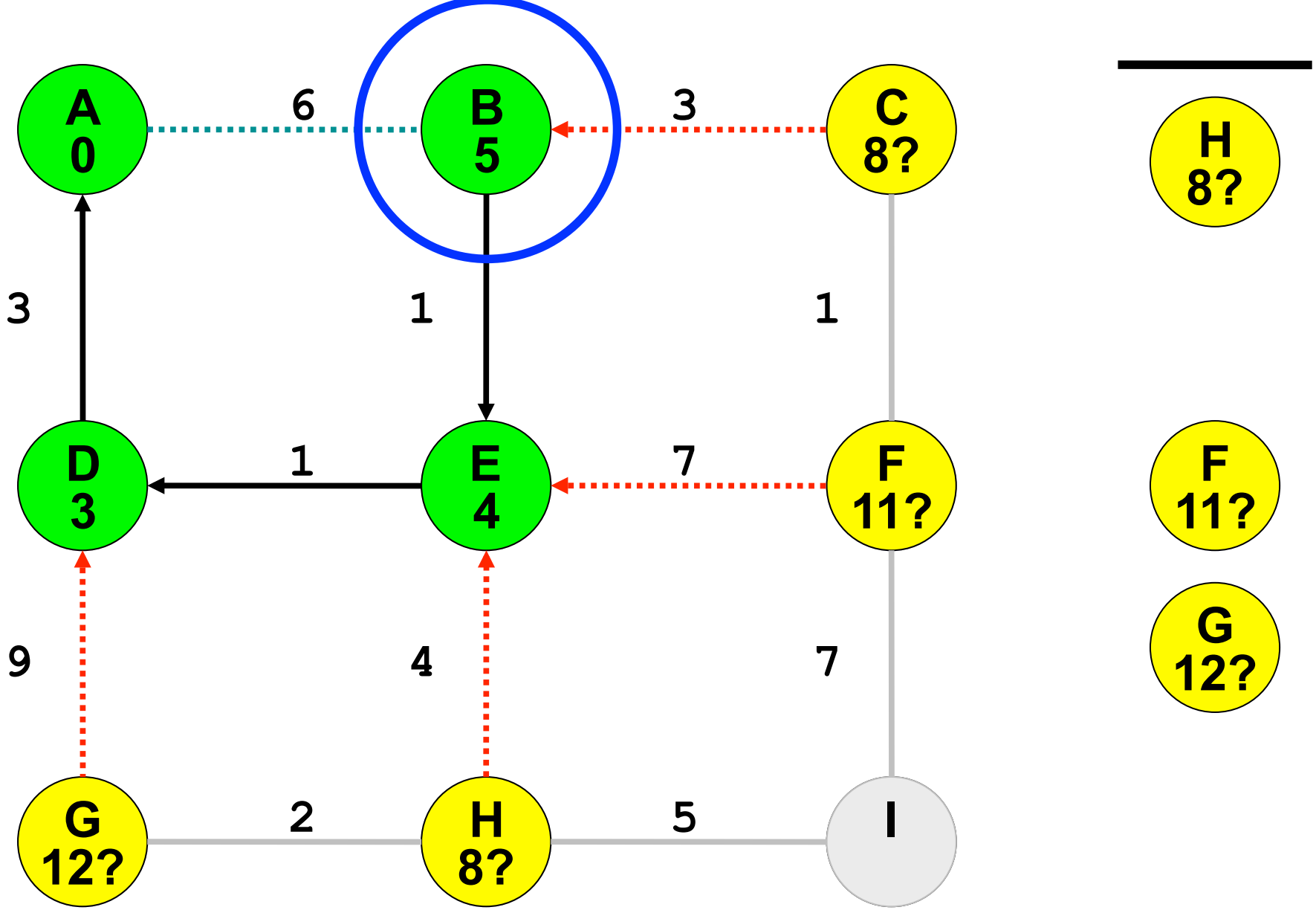

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
5?
H
8?
F
11?
G
12?

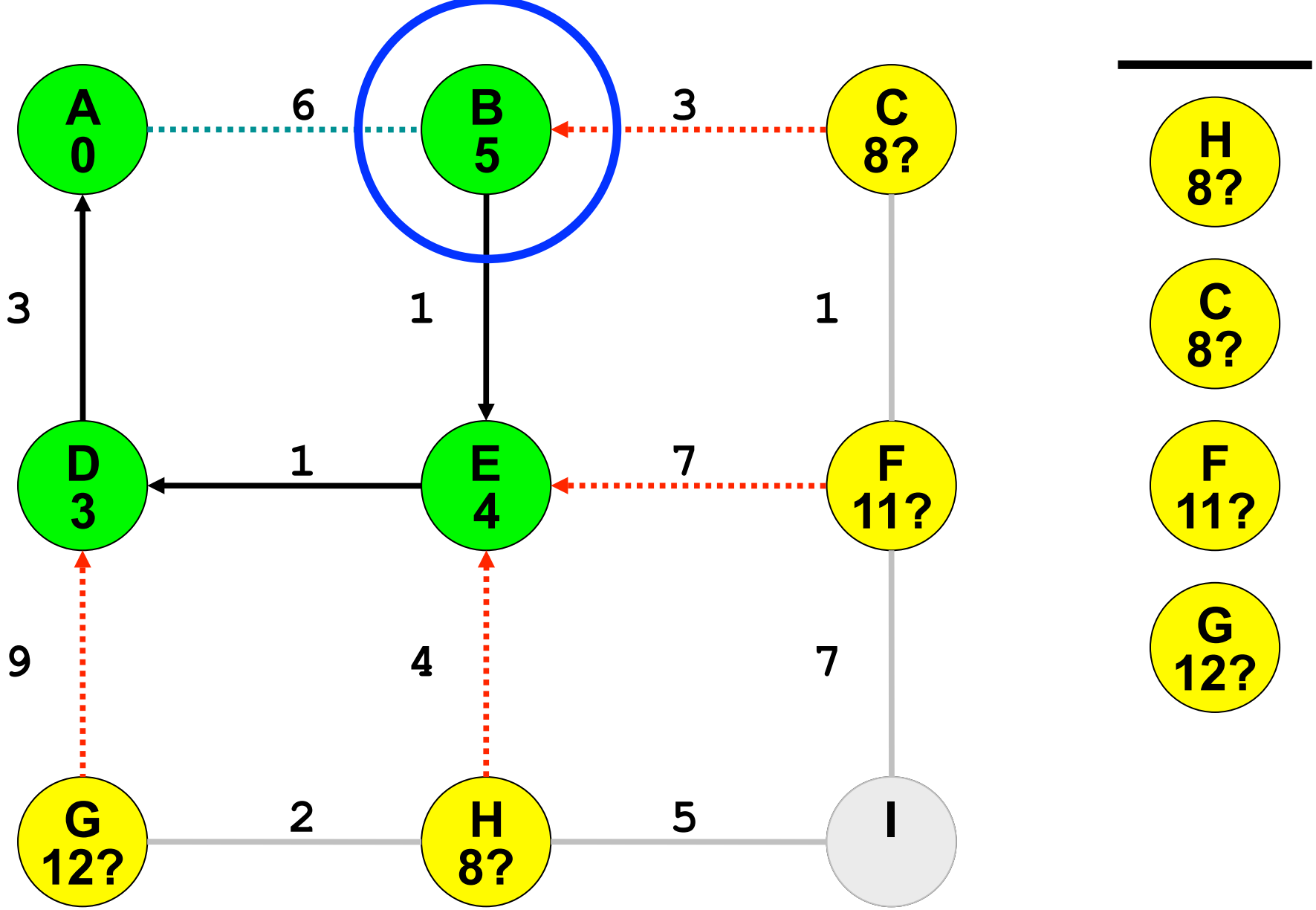

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
B
5?
H
8?
F
11?
G
12?

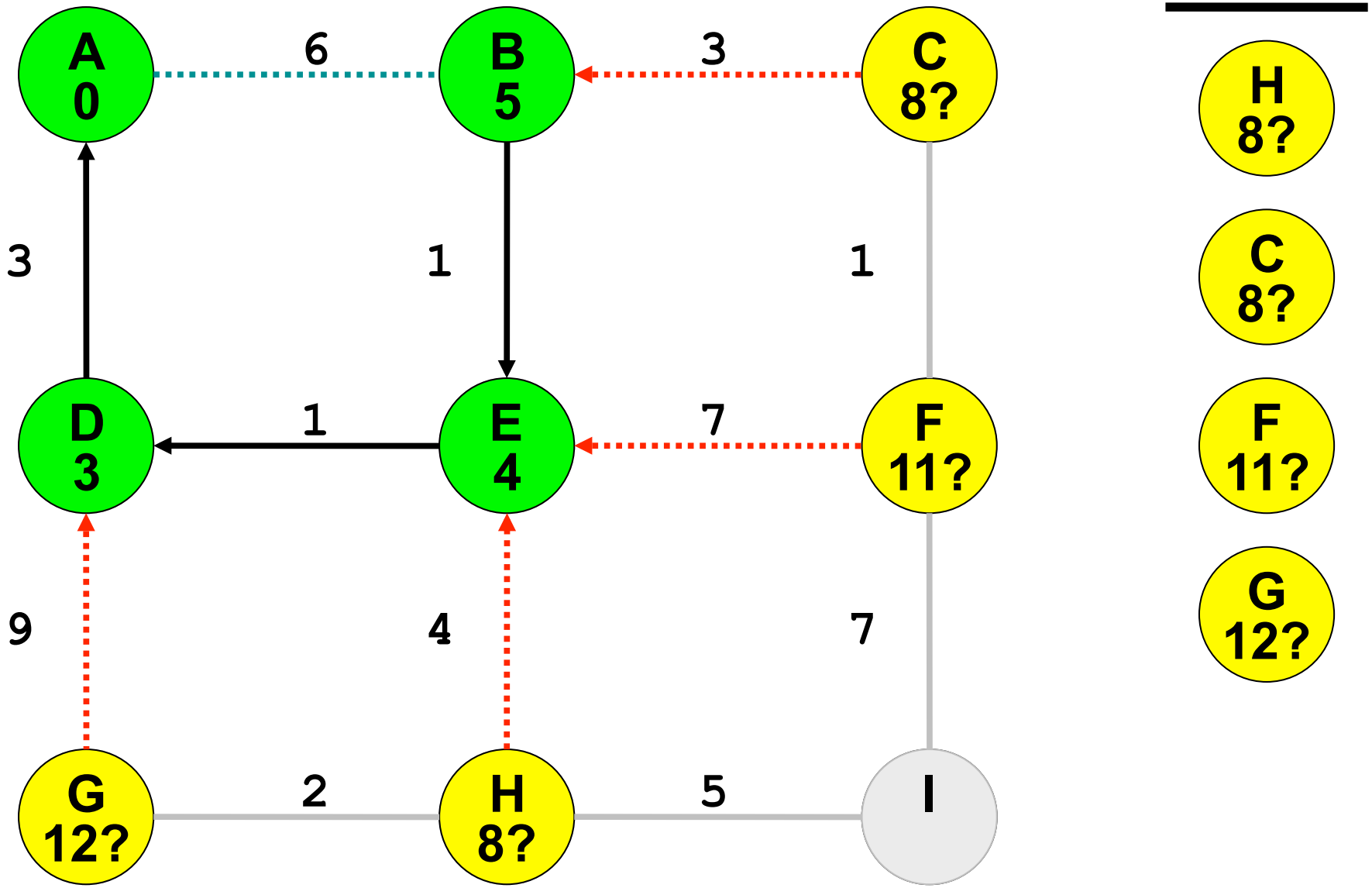

A
0
6
B
5?
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
F
11?
G
12?

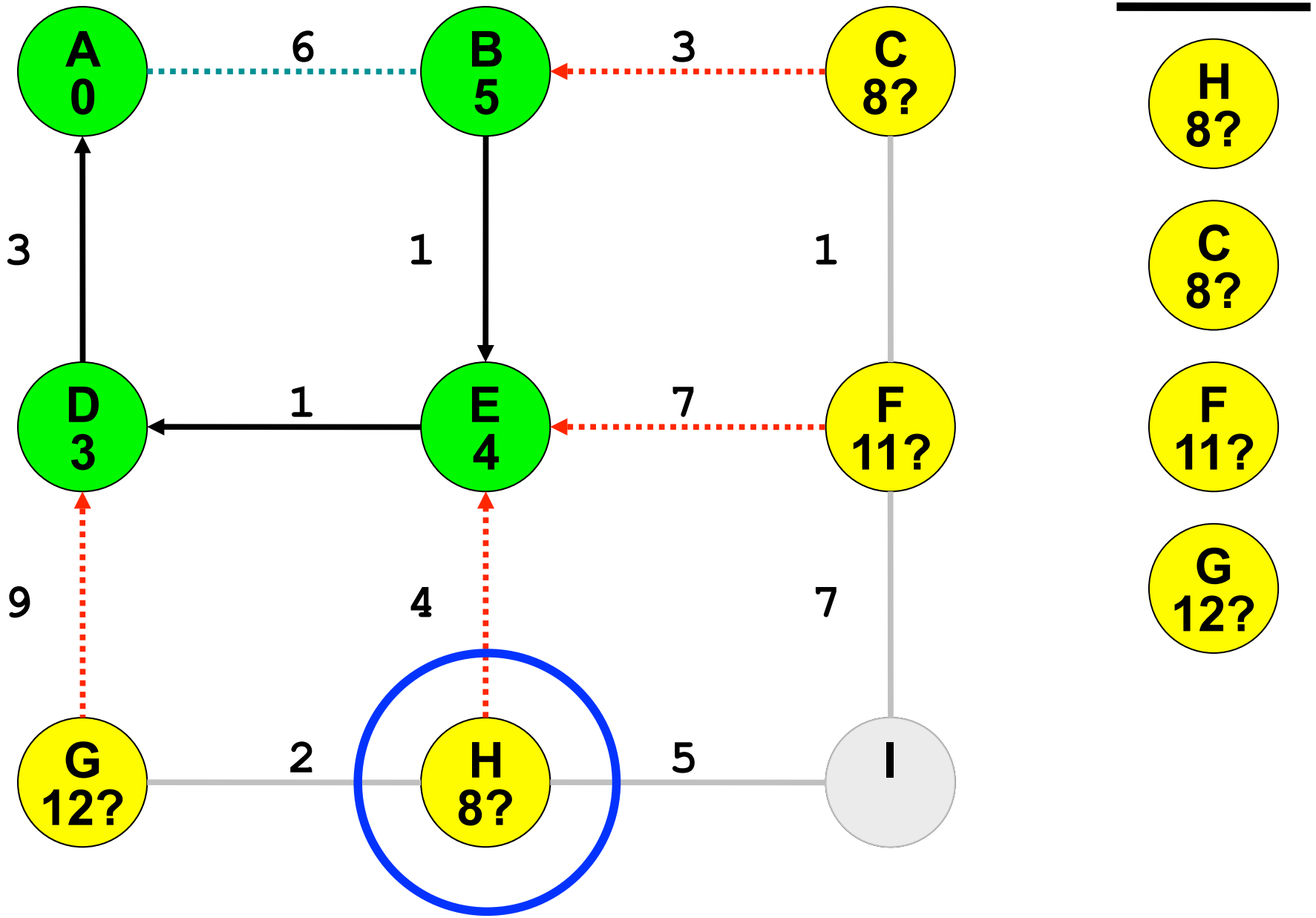

A
0
6
B
5
3
C
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
F
11?
G
12?

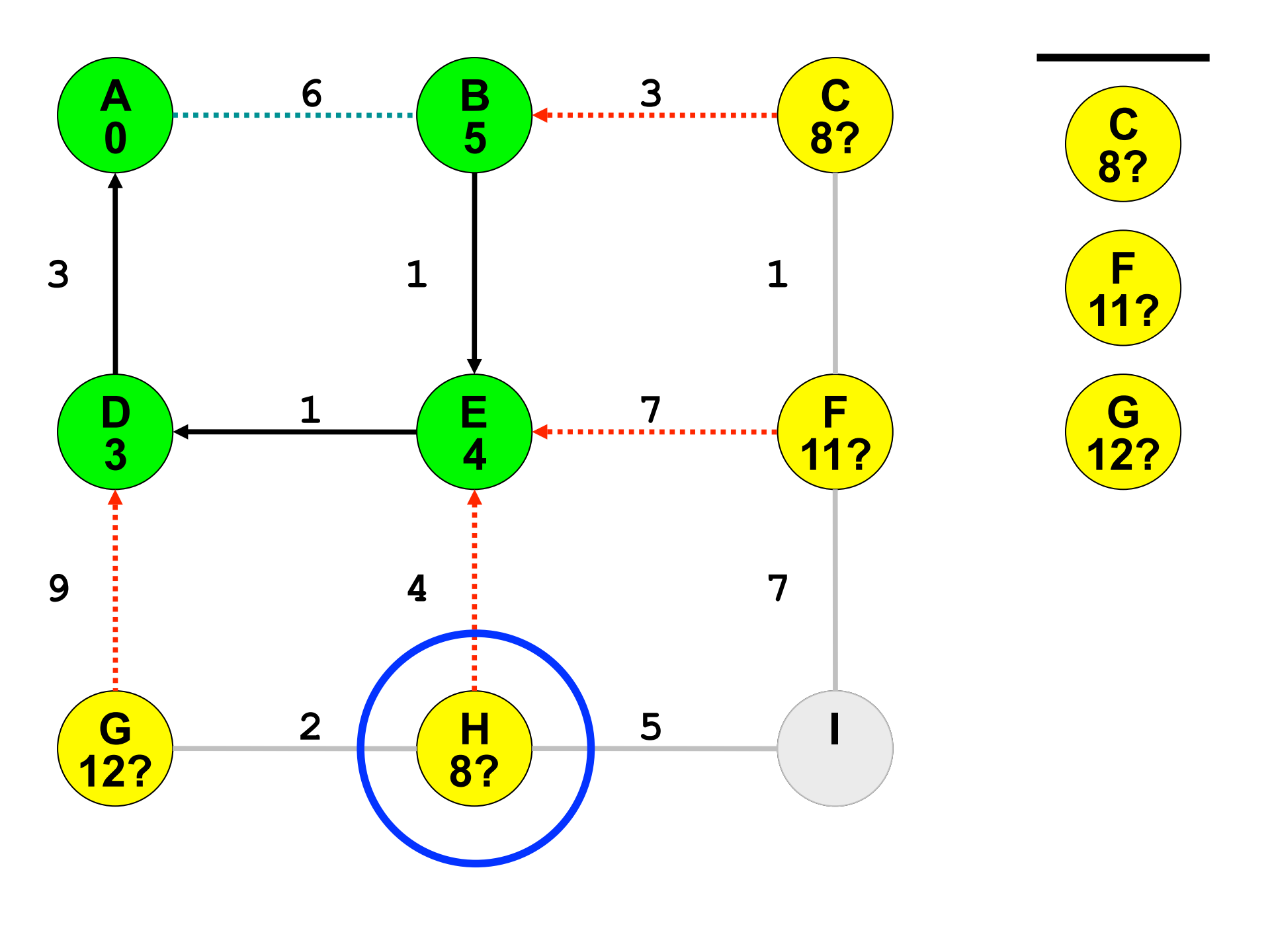

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
F
11?
G
12?

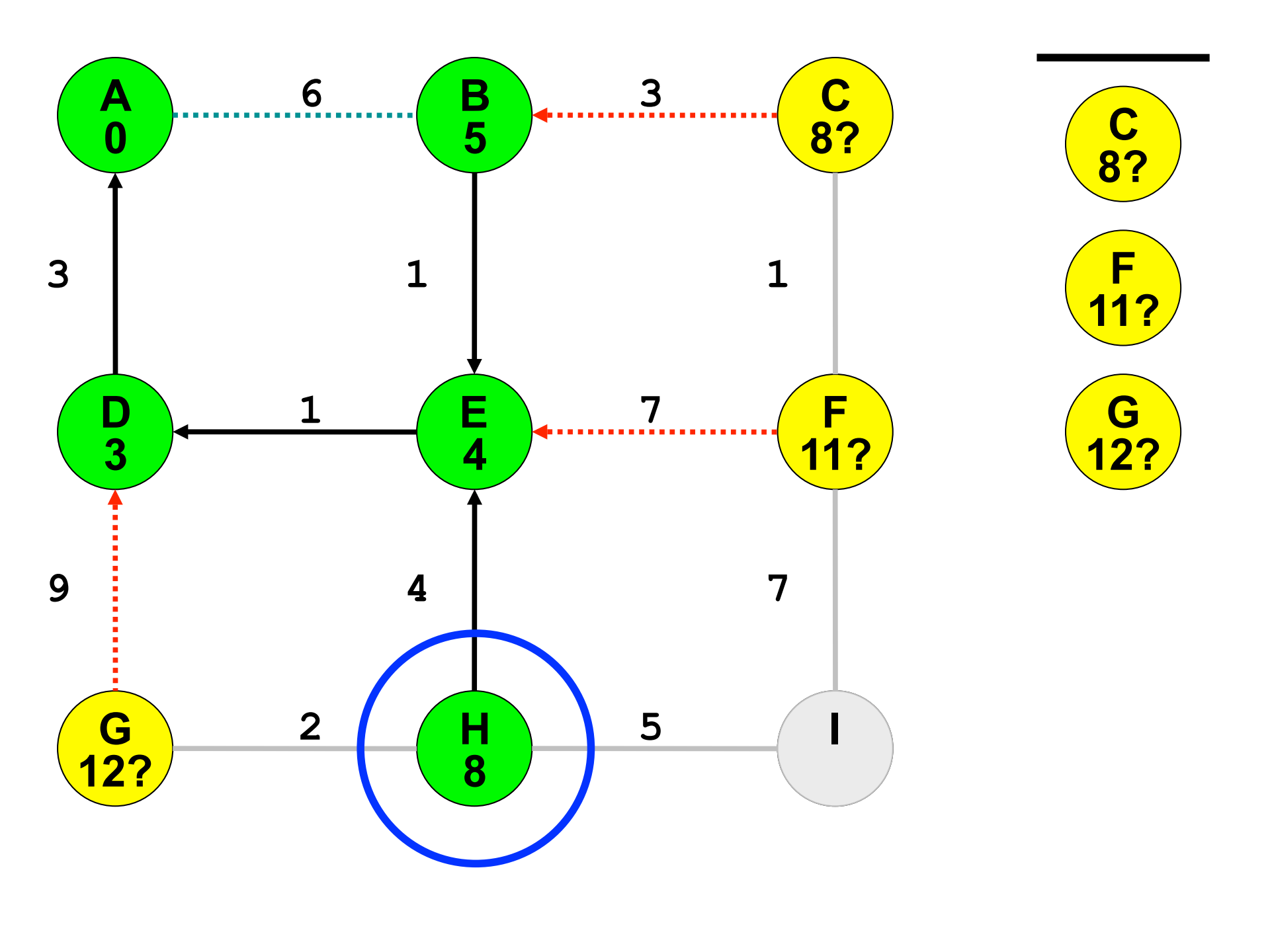

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
F
11?
G
12?

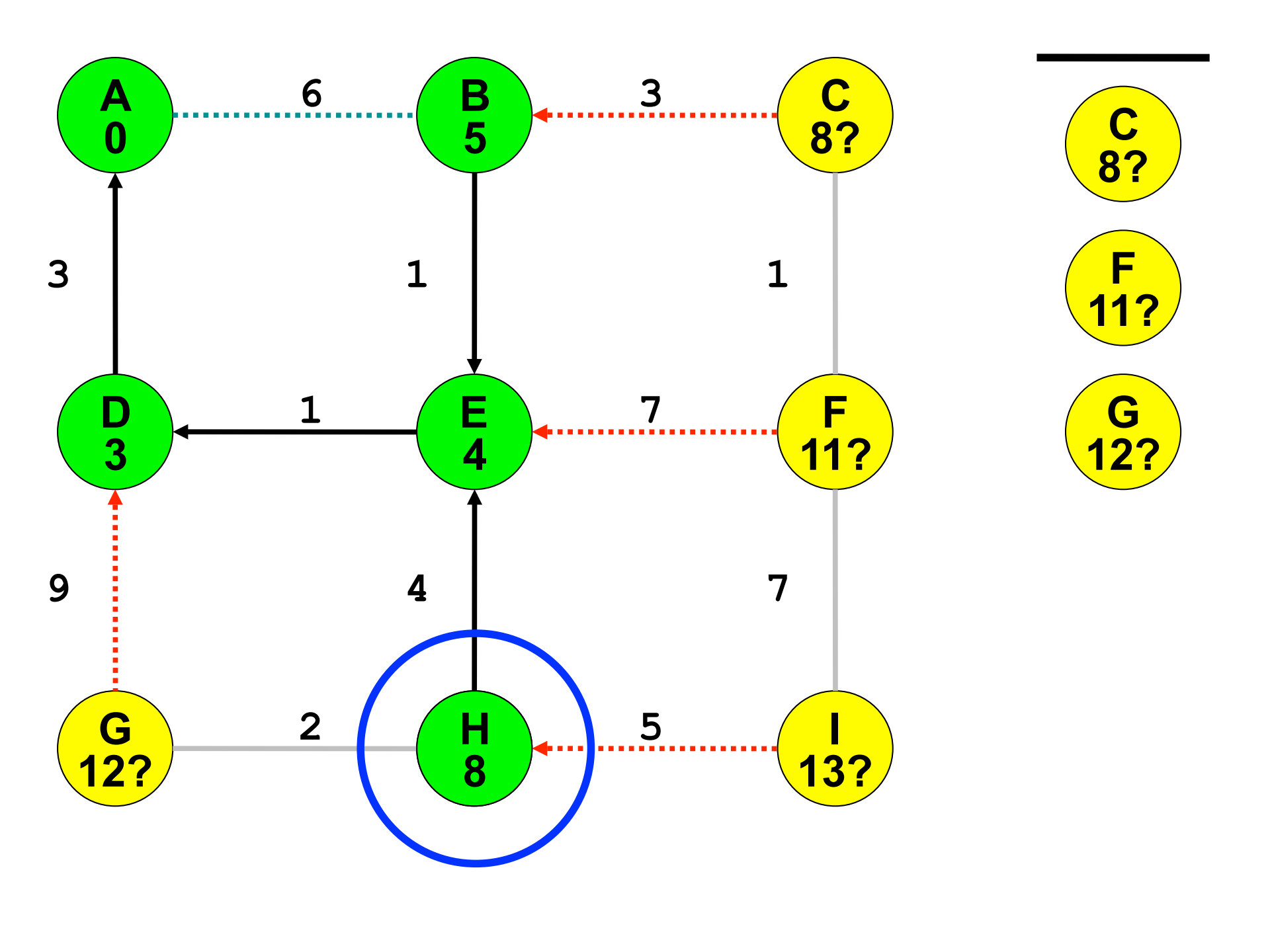

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
C
8?
F
11?
G
12?

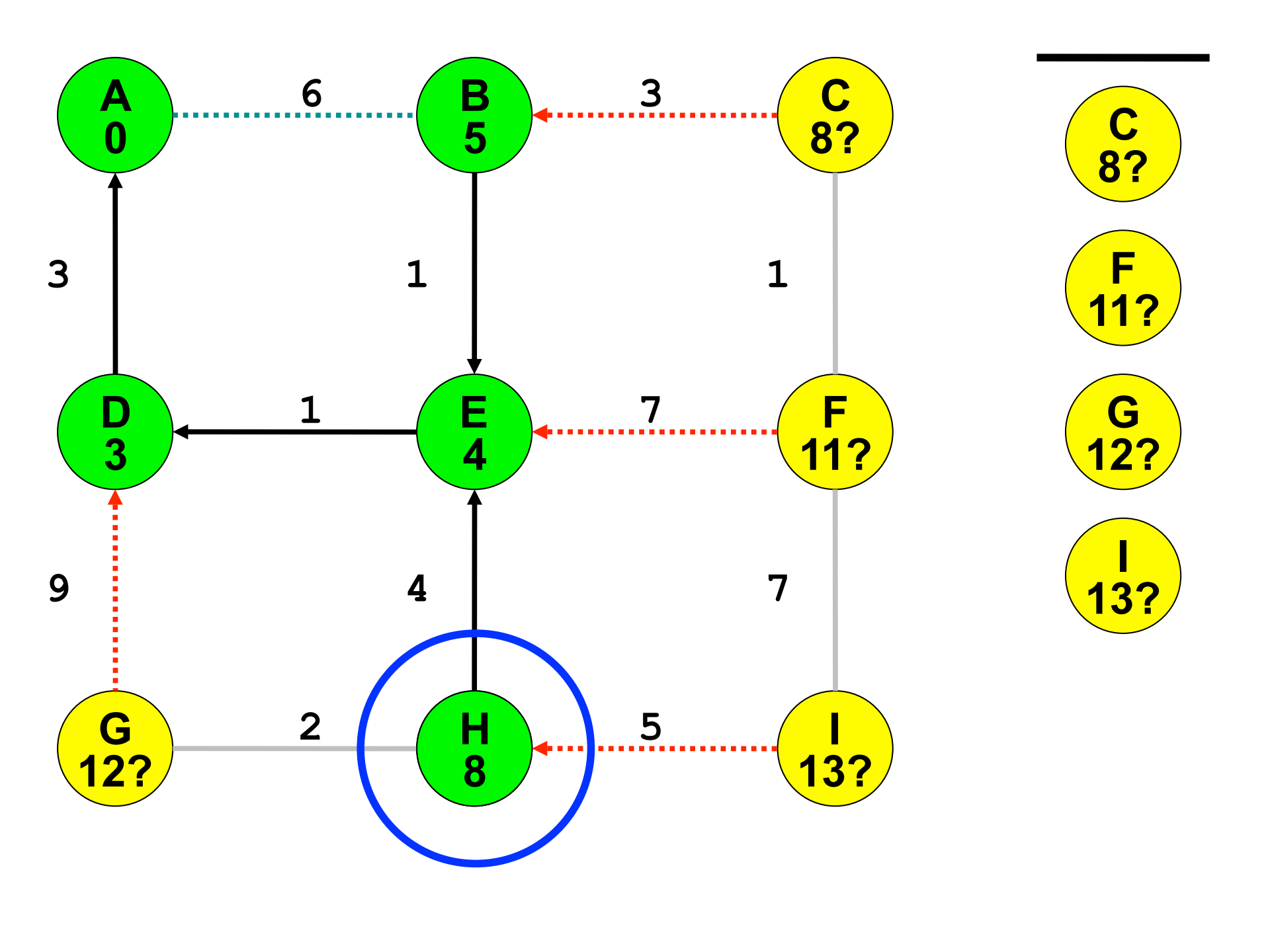

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
C
8?
F
11?
G
12?

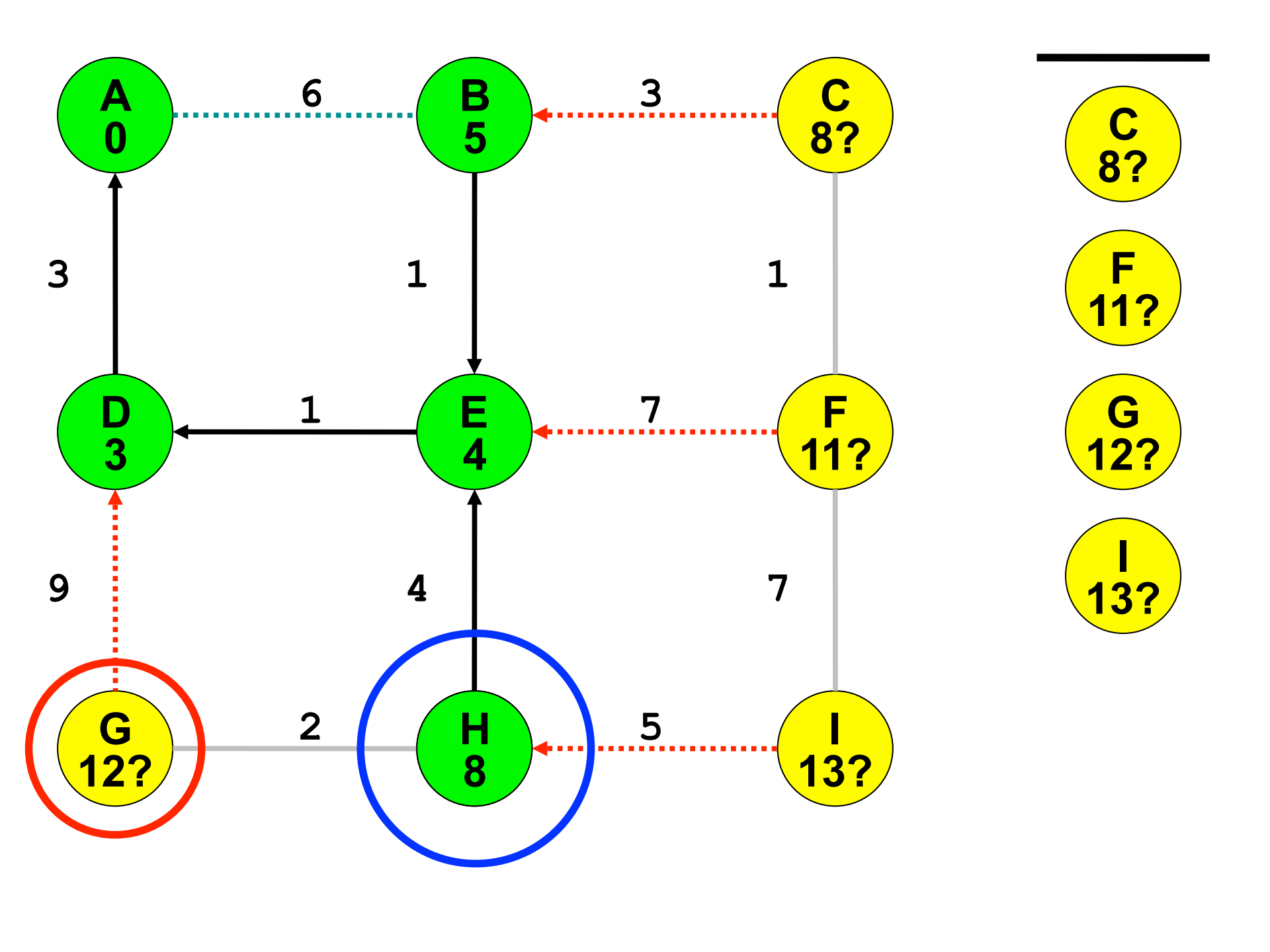

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
H
8?
C
8?
F
11?
G
12?

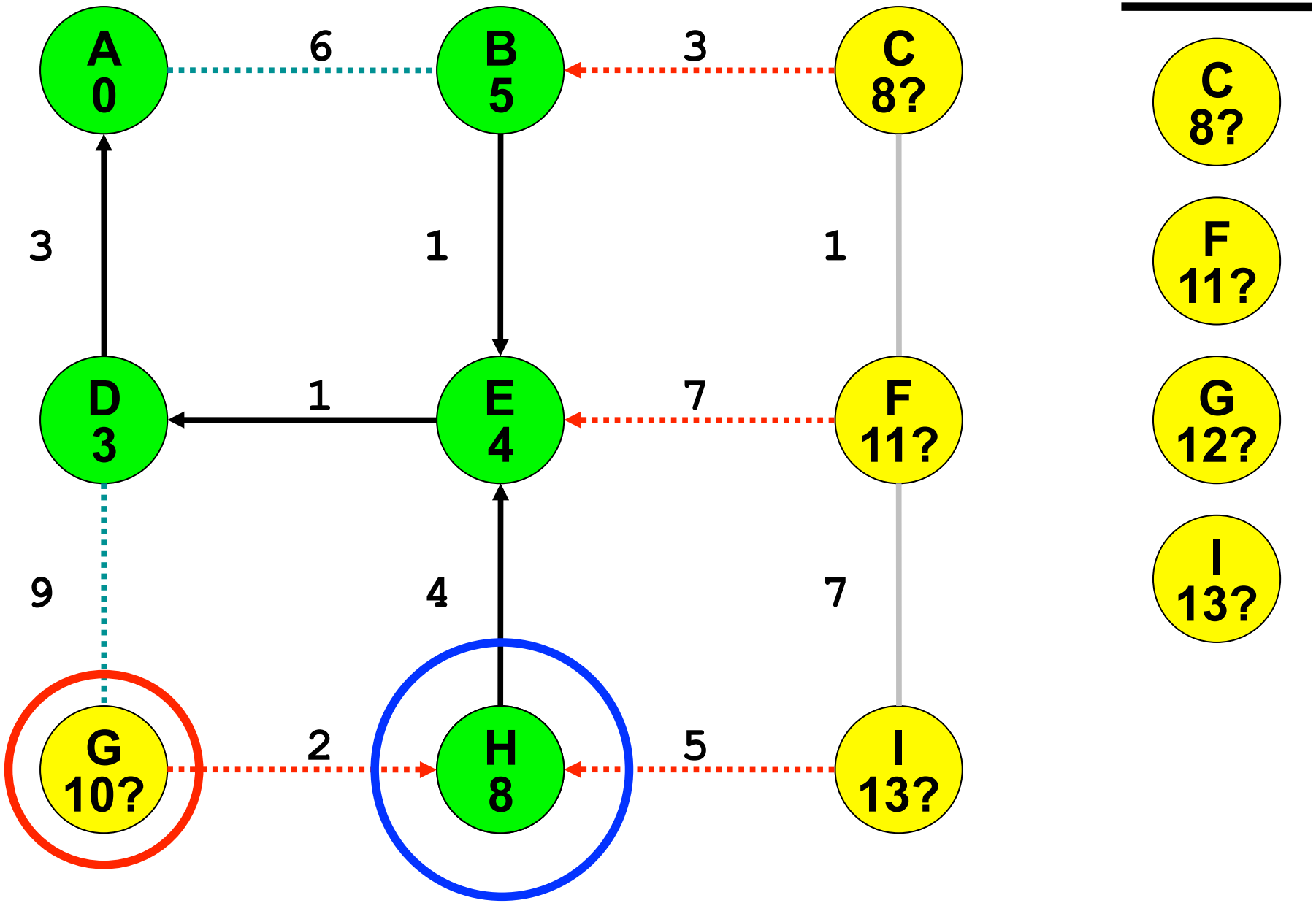

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8?
5
I
C
8?
F
11?
G
12?

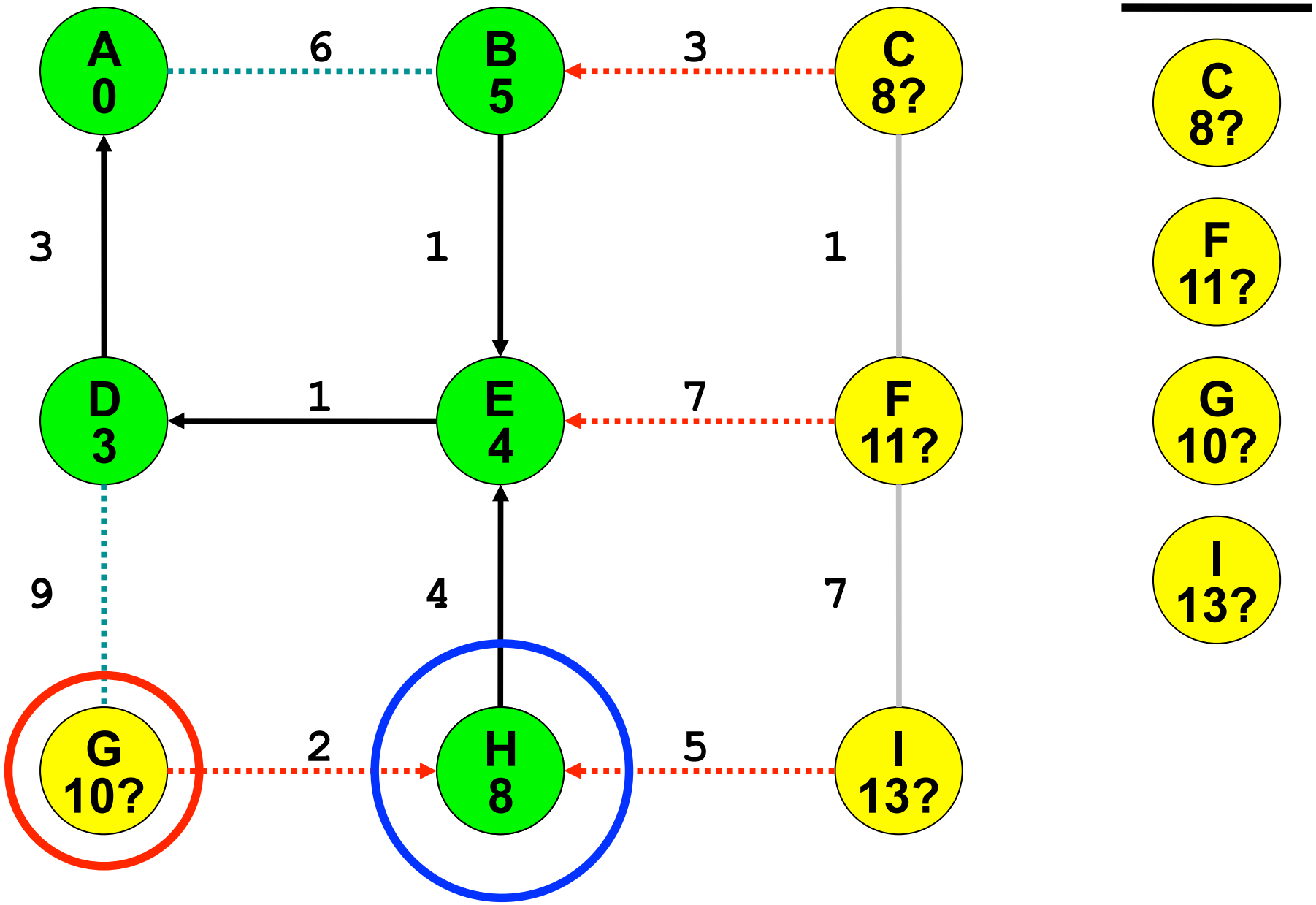

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8
5
I
C
8?
F
11?
G
12?

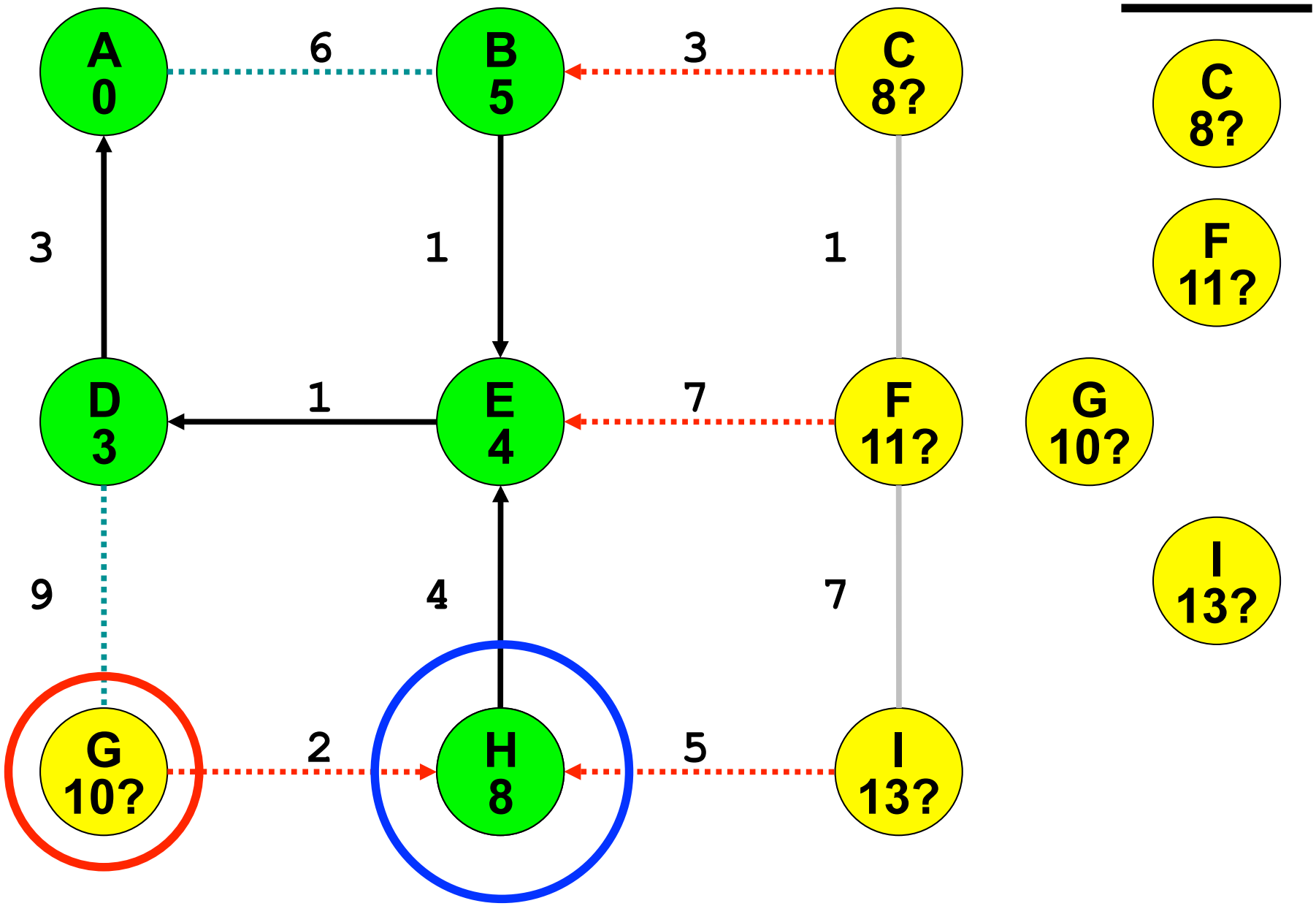

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8
5
I
13?
C
8?
F
11?
G
12?

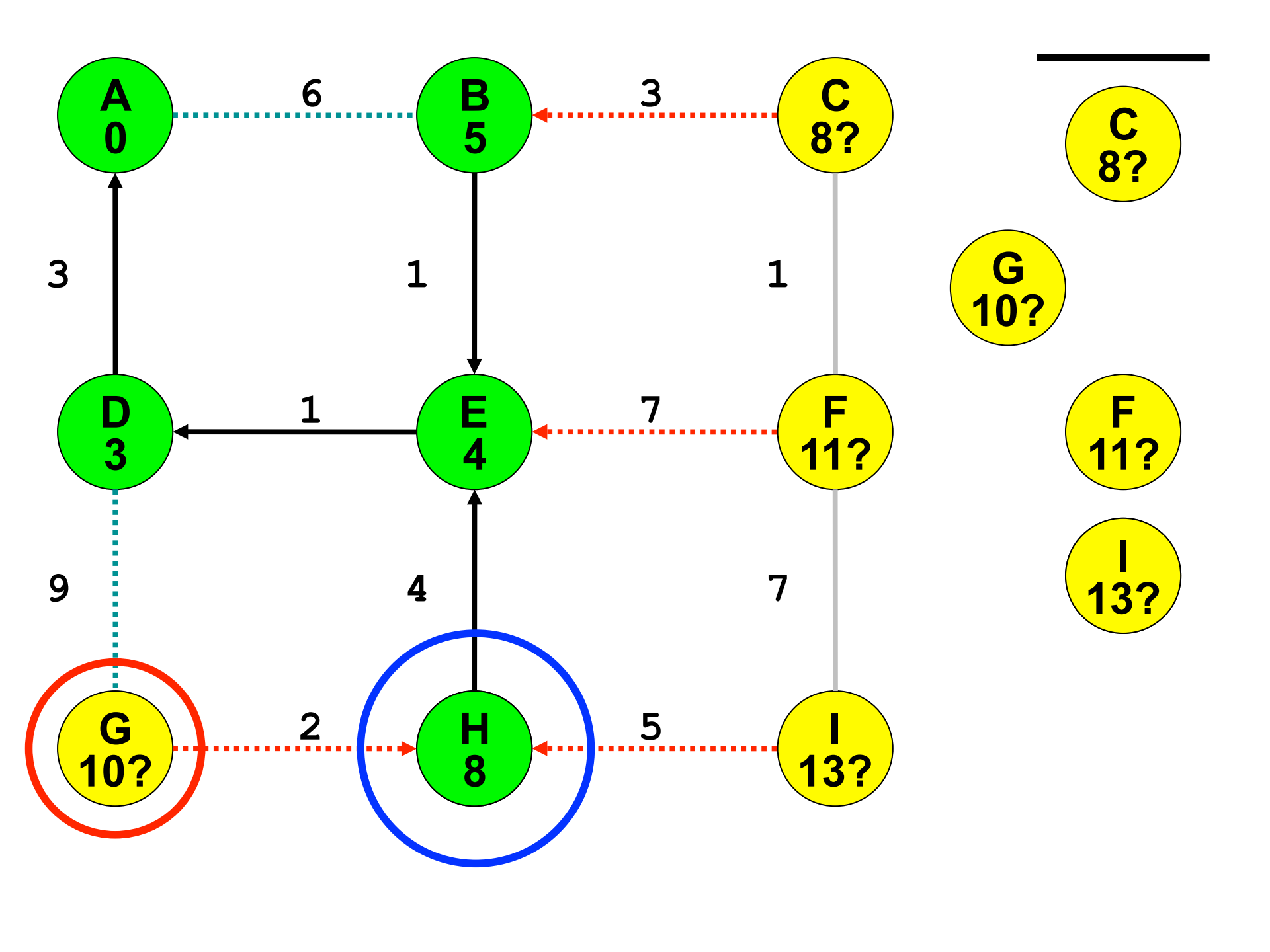

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8
5
I
13?
C
8?
F
11?
G
12?
I
13?

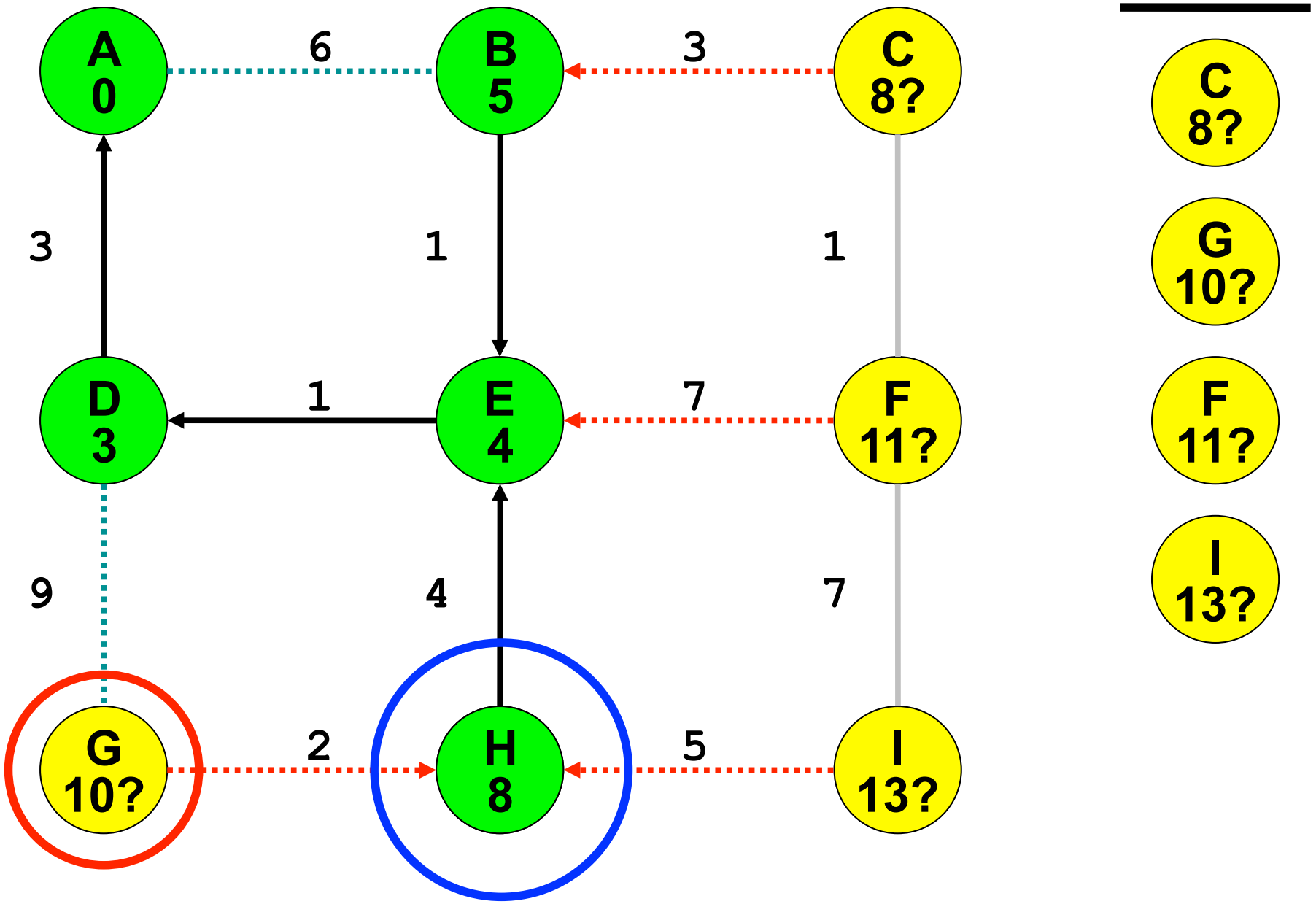

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
12?
2
H
8
5
I
13?
C
8?
F
11?
G
12?
I
13?

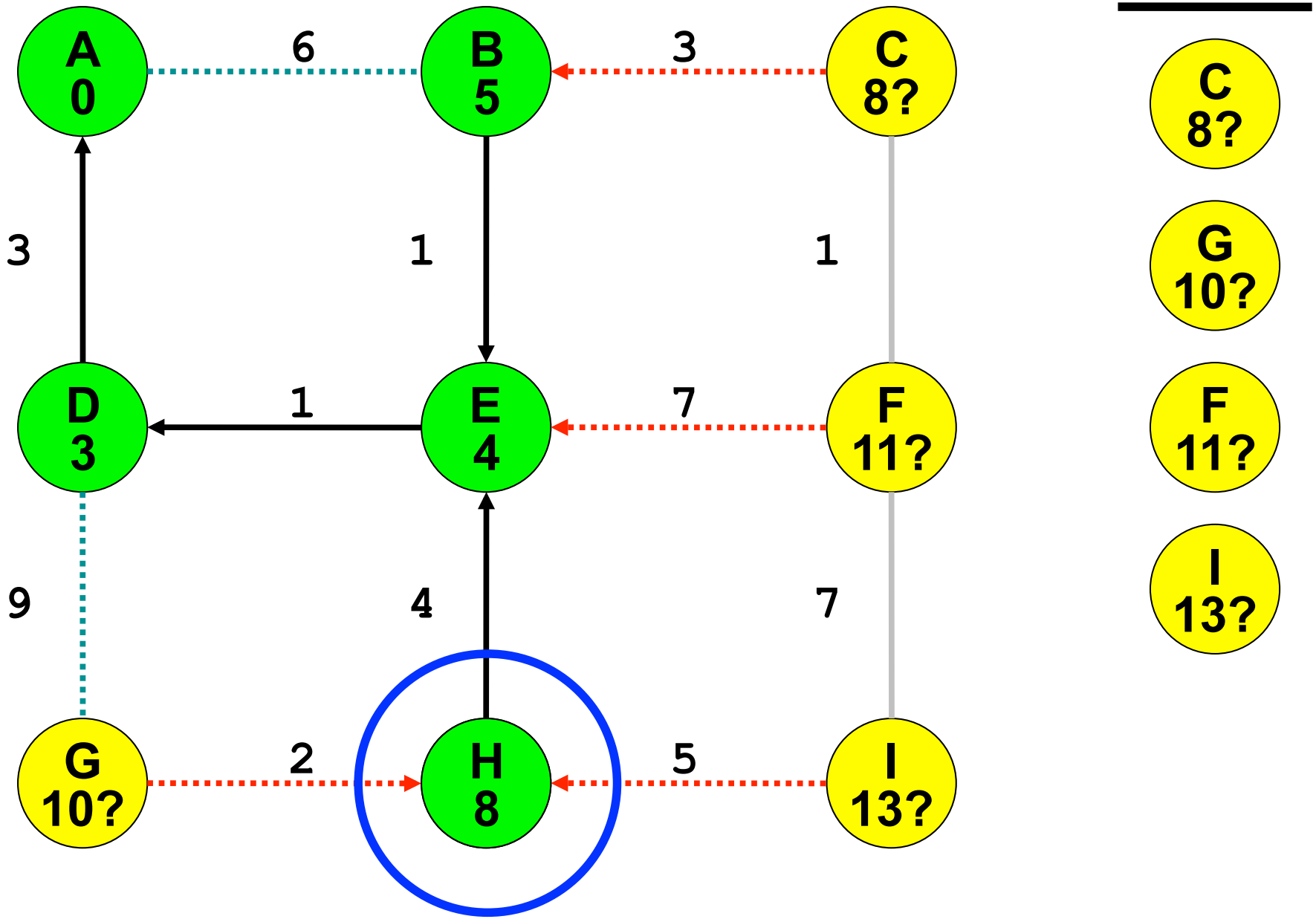

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
F
11?
G
12?
I
13?

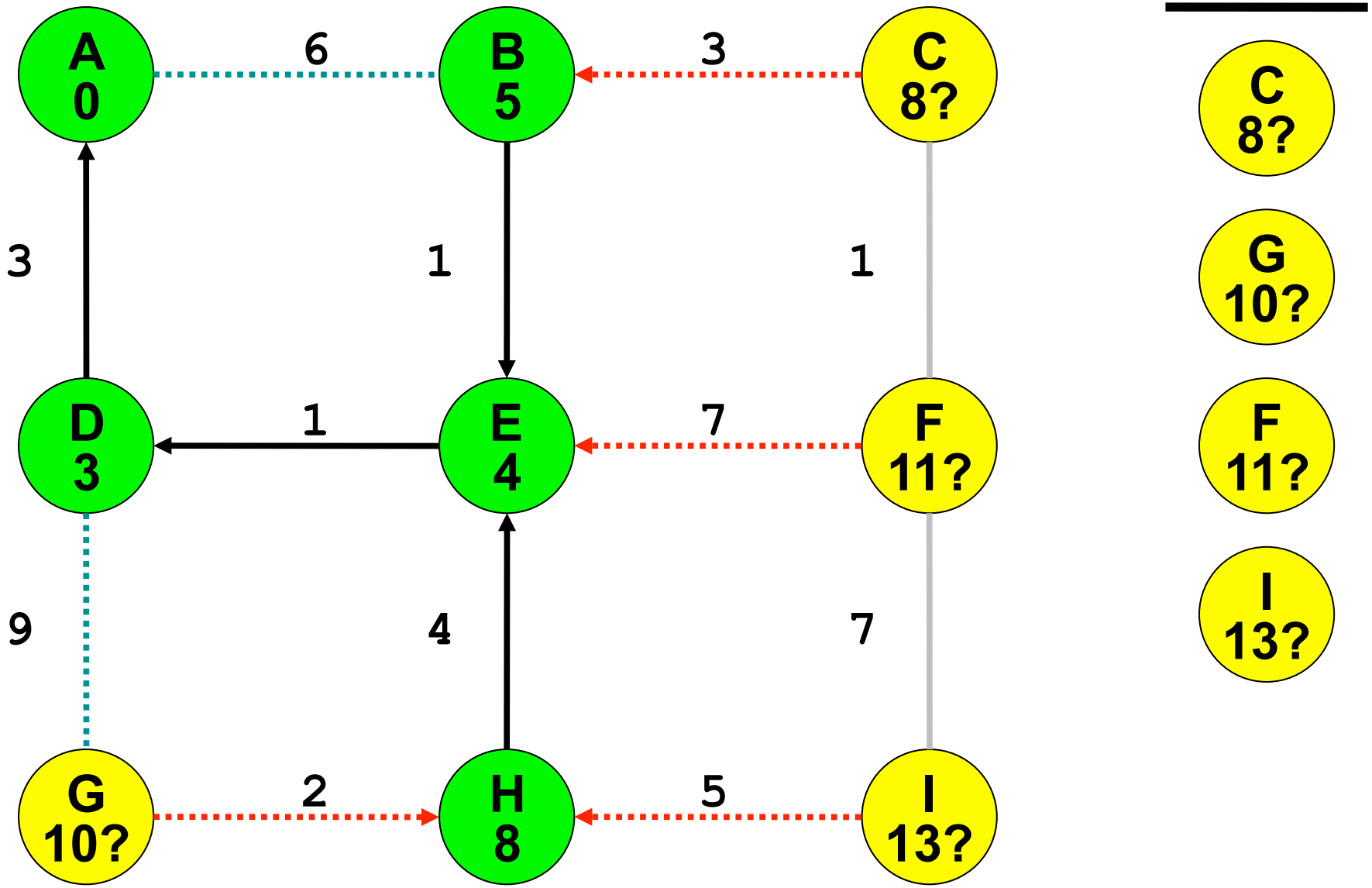

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
F
11?
G
10?
I
13?

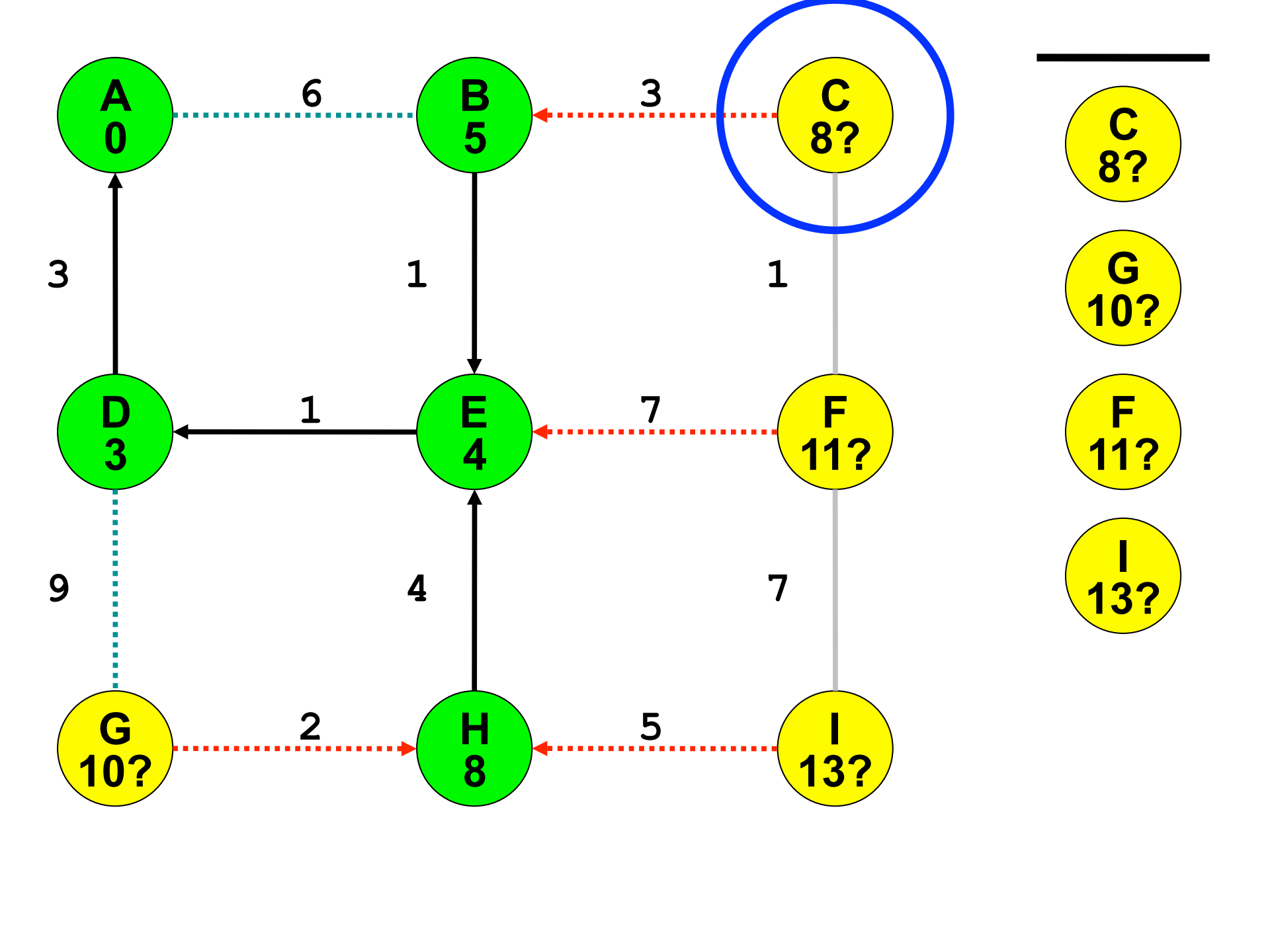

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
F
11?
G
10?
I
13?

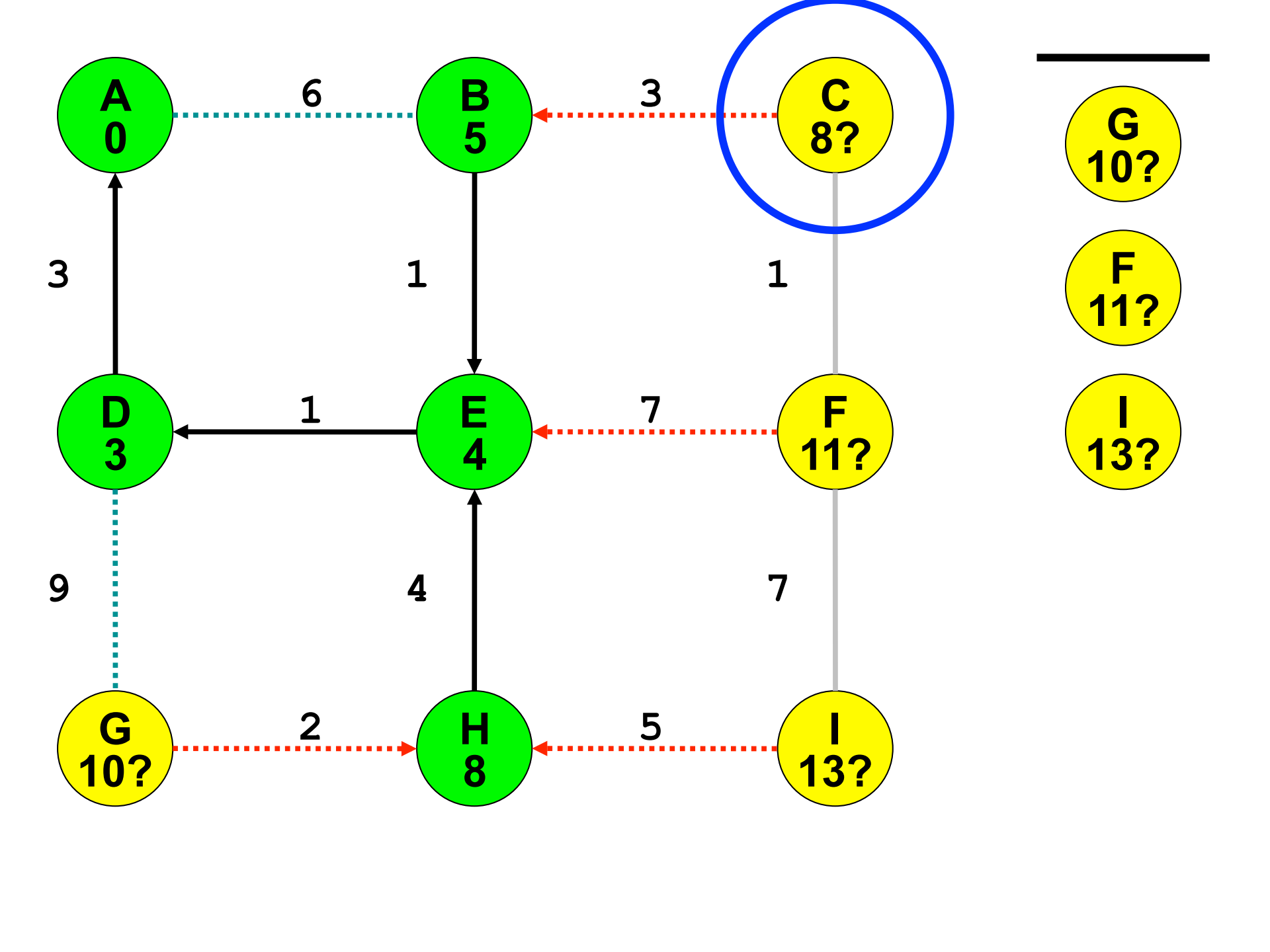

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
G
10?
F
11?
I
13?

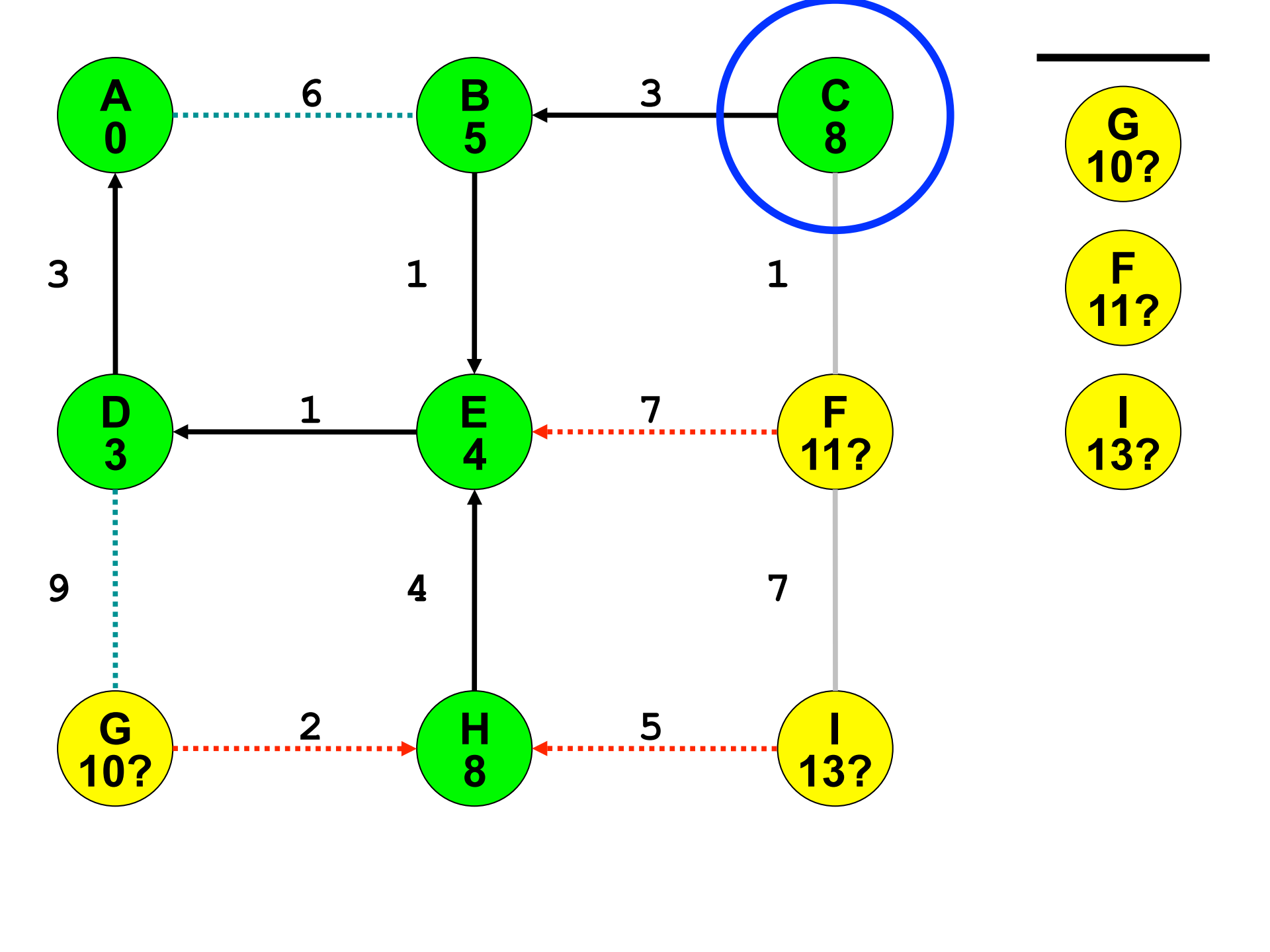

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
G
10?
F
11?
I
13?

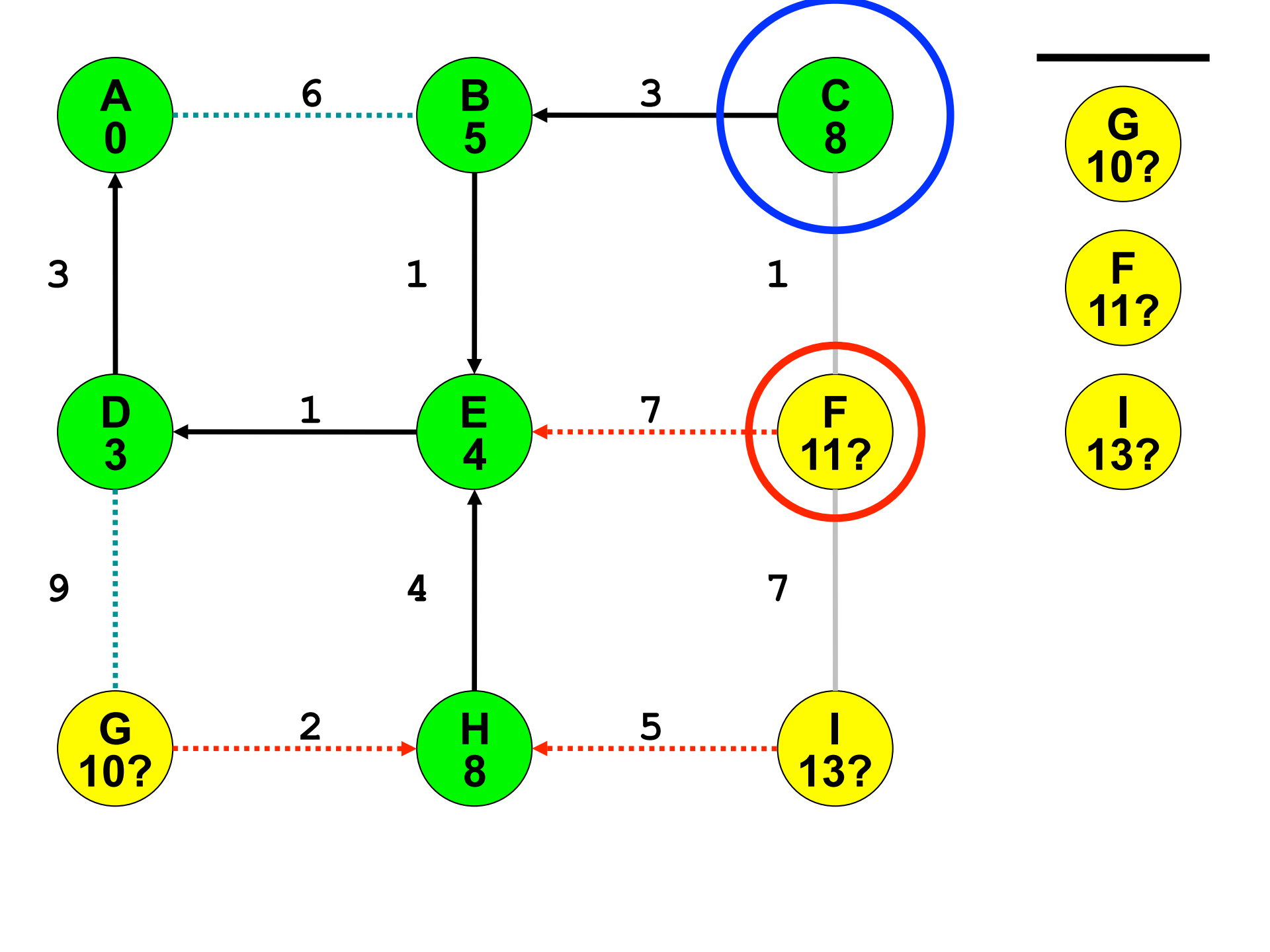

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
G
10?
F
11?
I
13?

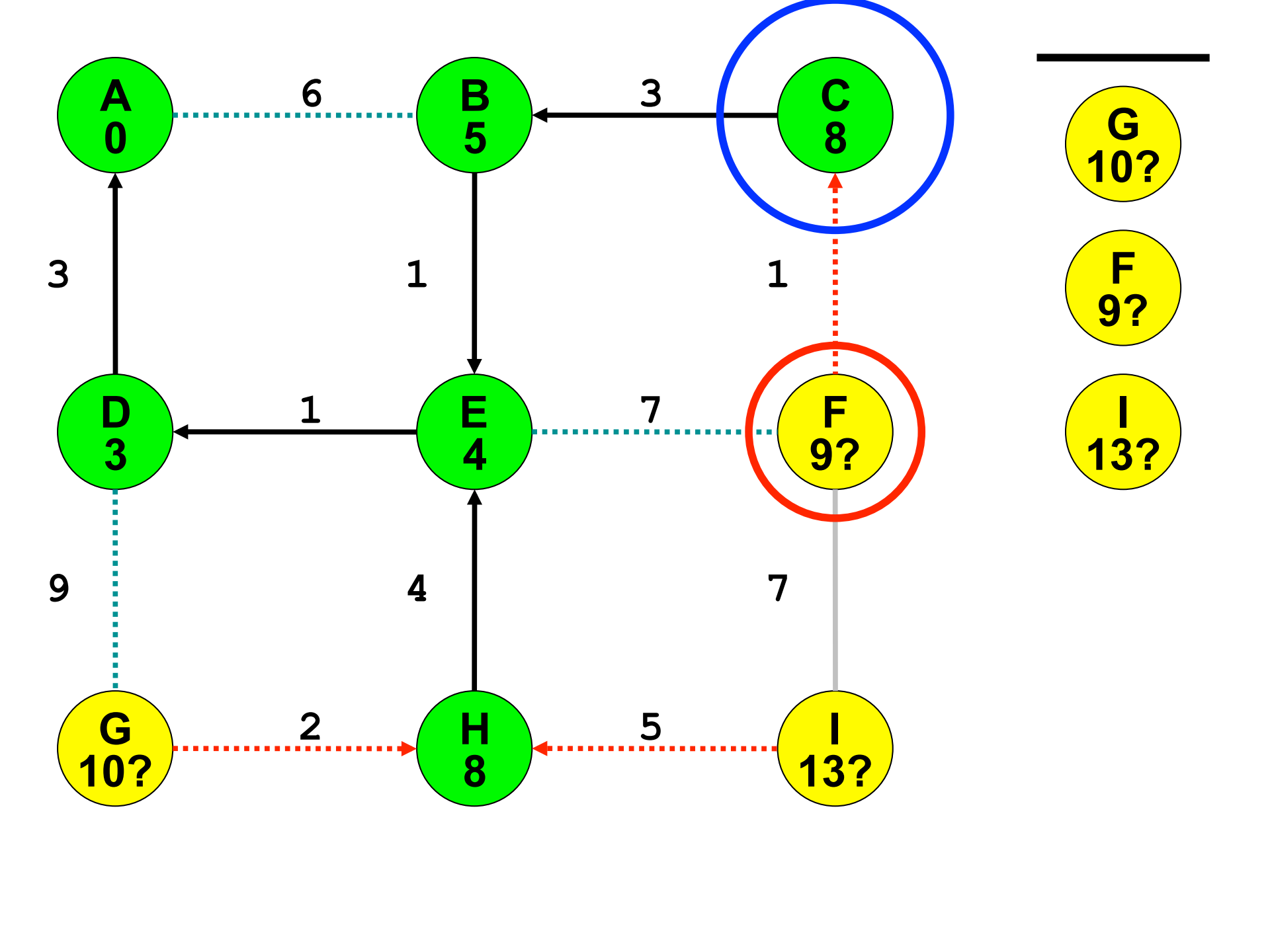

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
G
10?
F
11?
I
13?

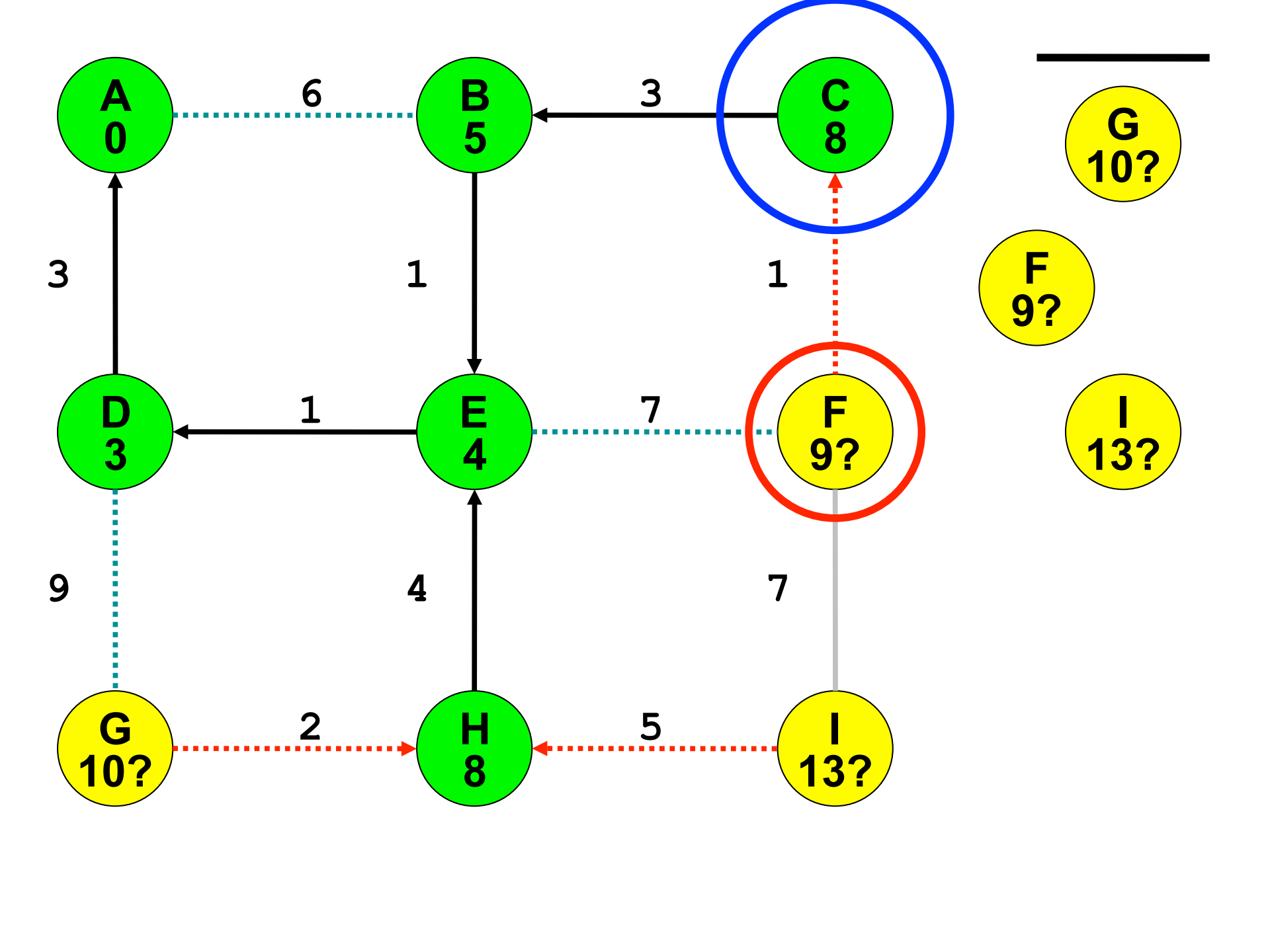

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
C
8?
G
10?
F
11?
I
13?

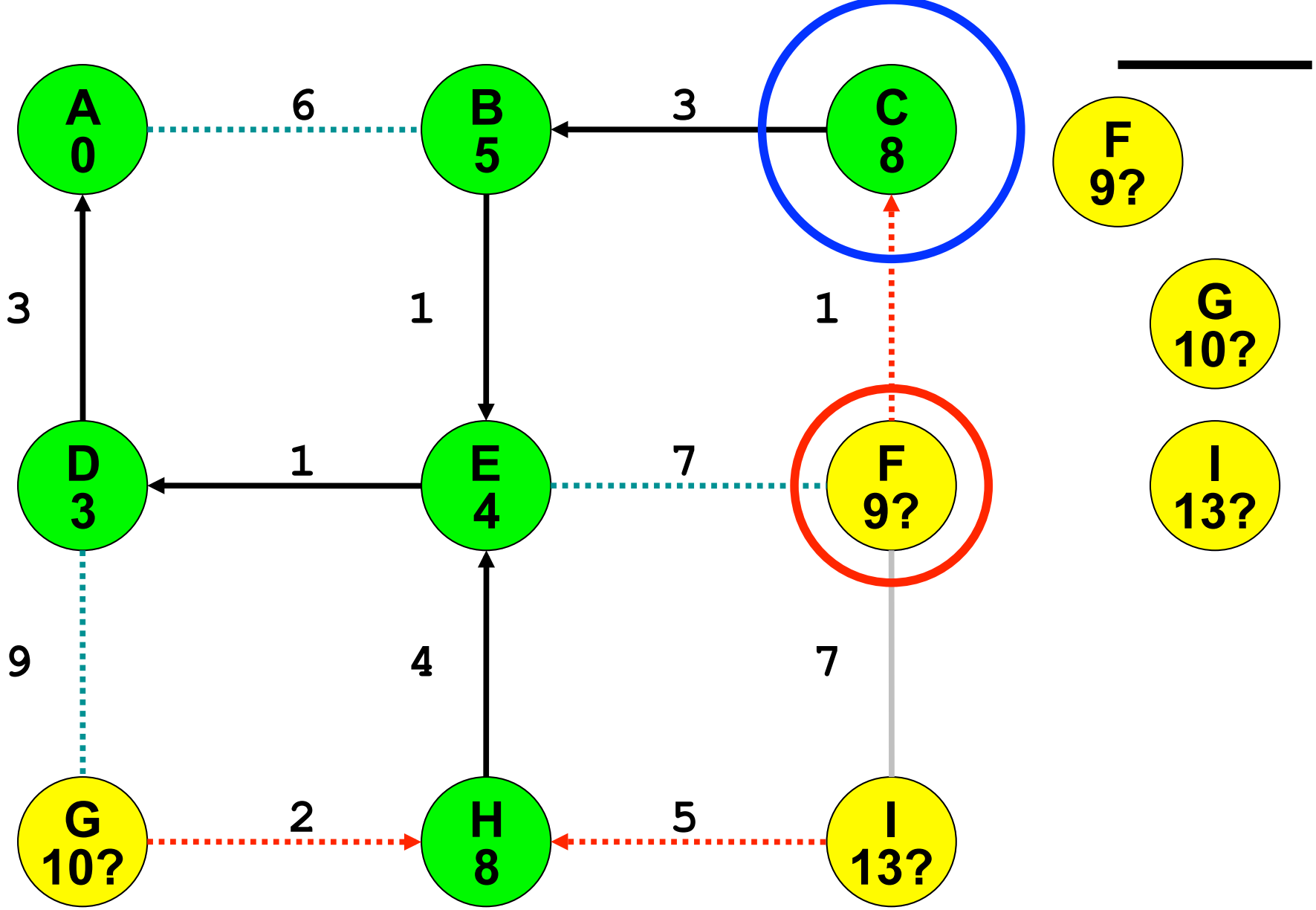

A
0
6
B
5
3
C
8?
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
F
11?
I
13?

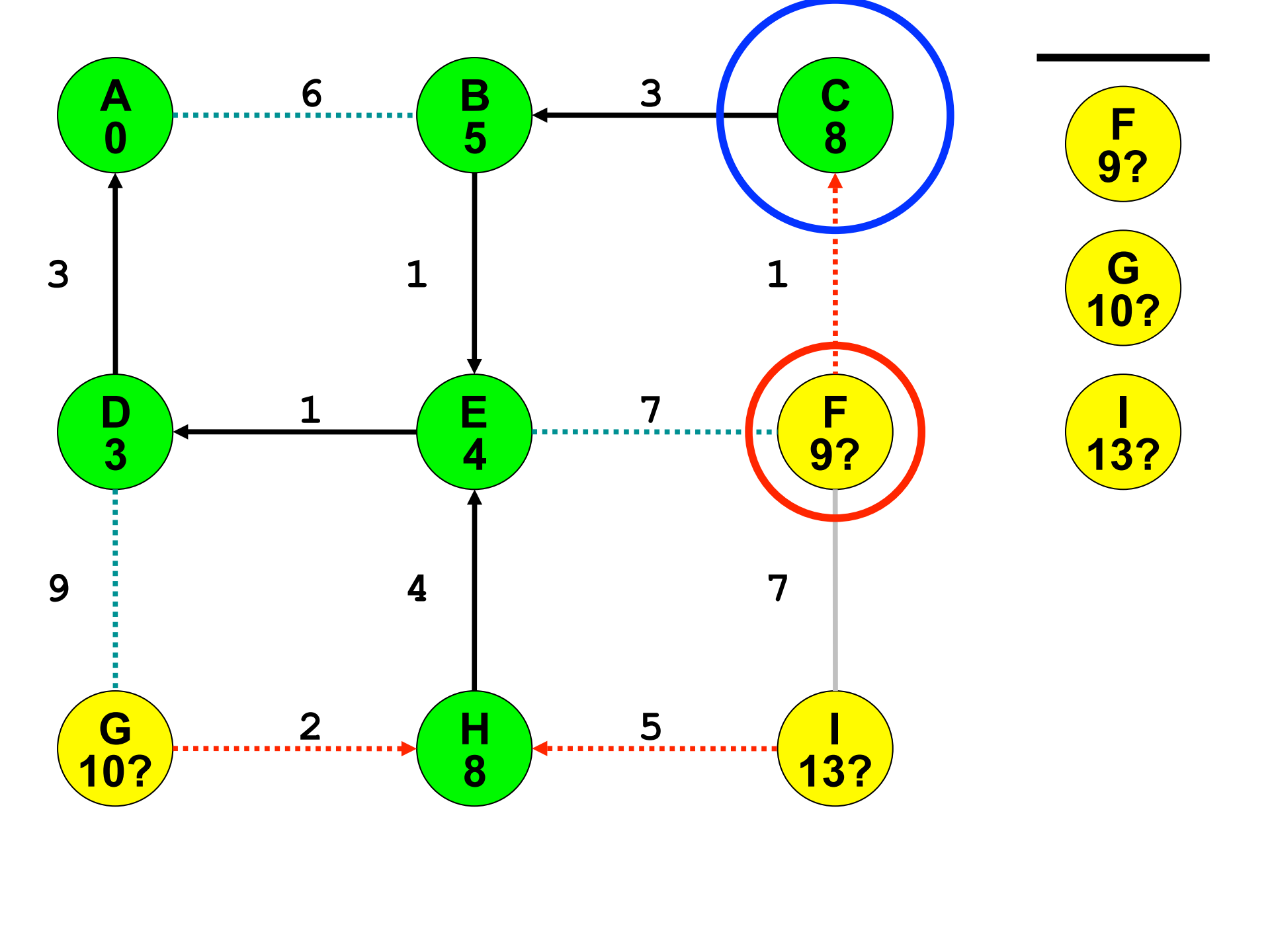

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
F
11?
I
13?

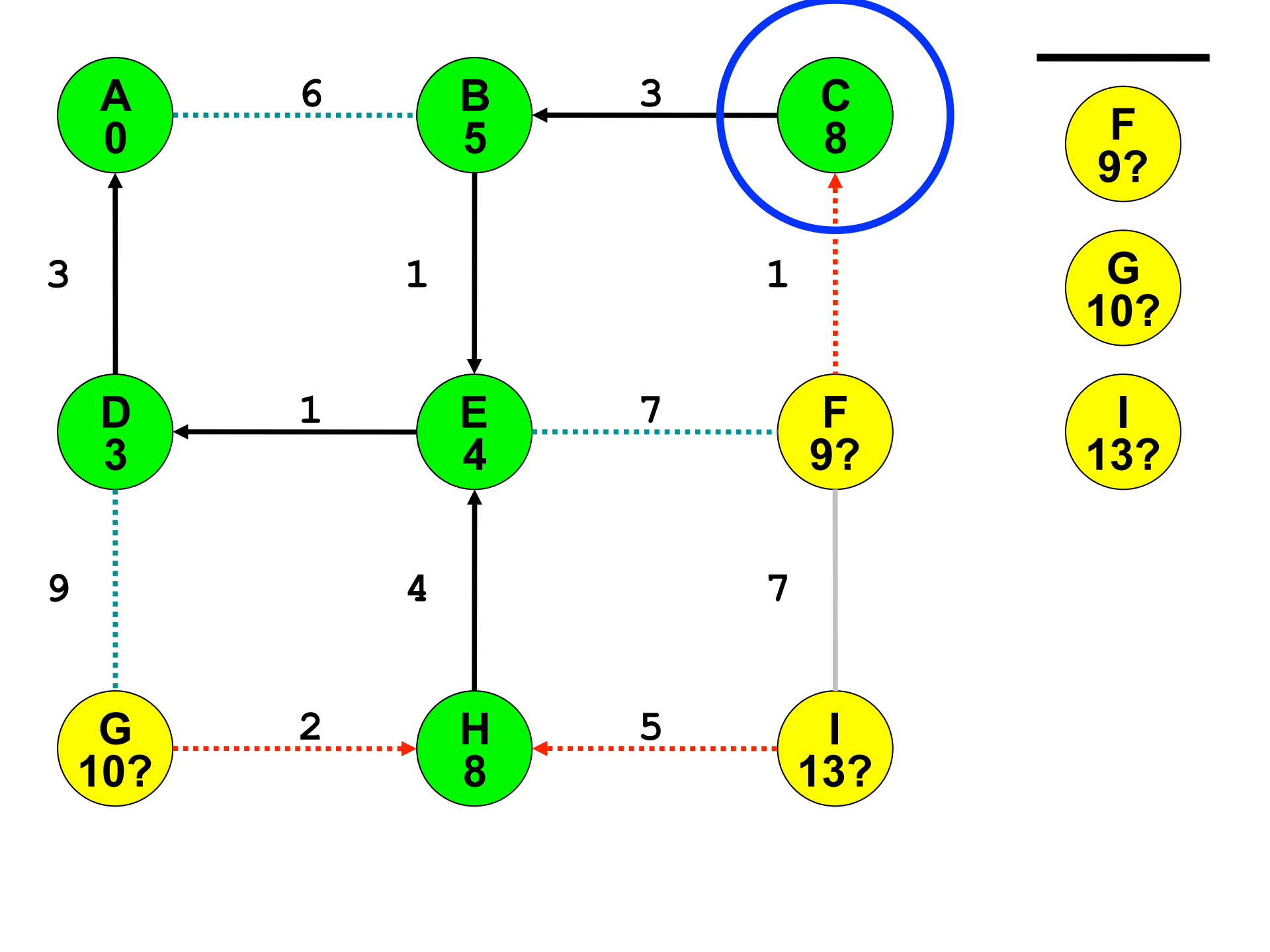

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
11?
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
F
11?
I
13?

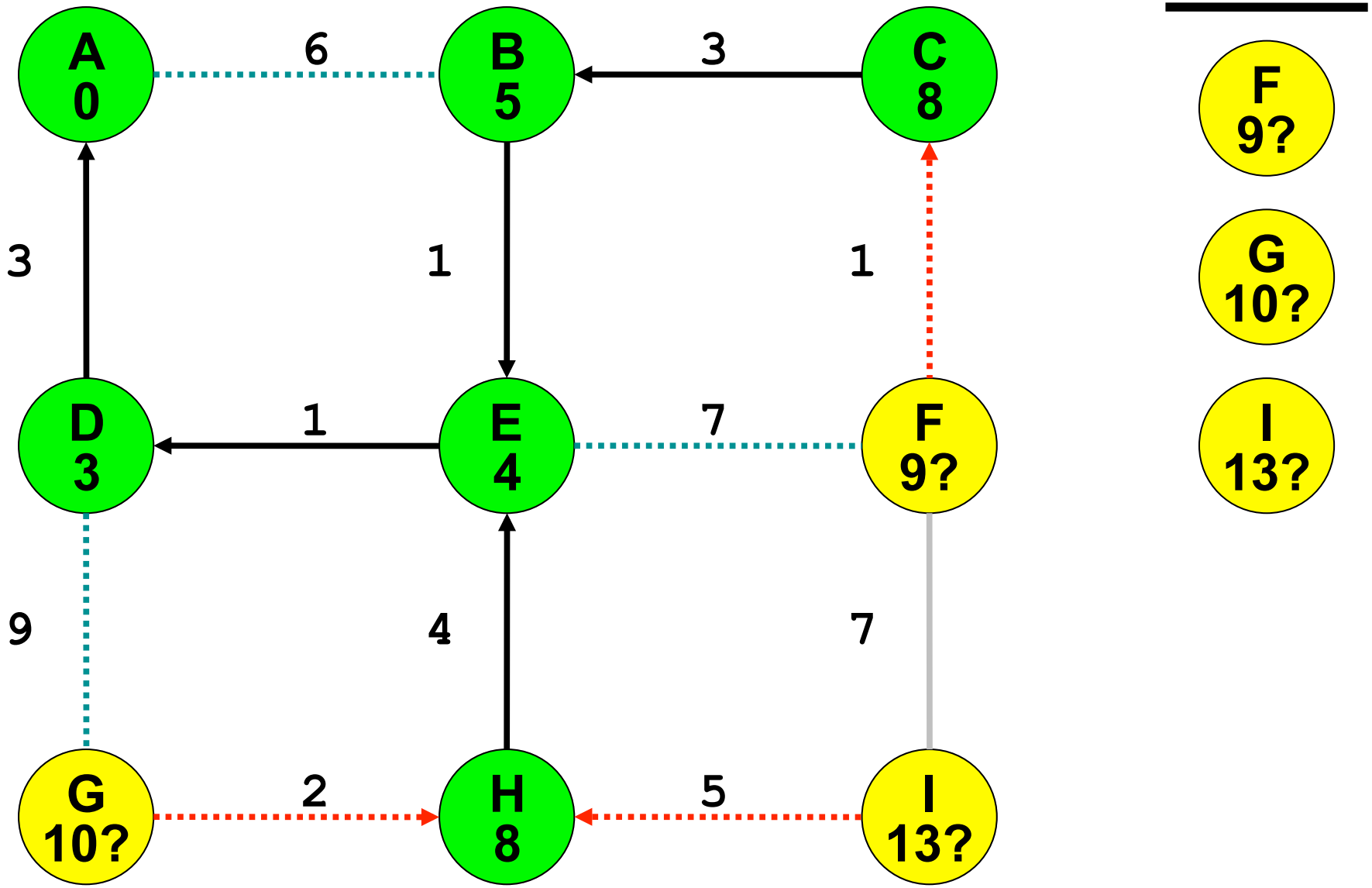

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
F
9?
I
13?

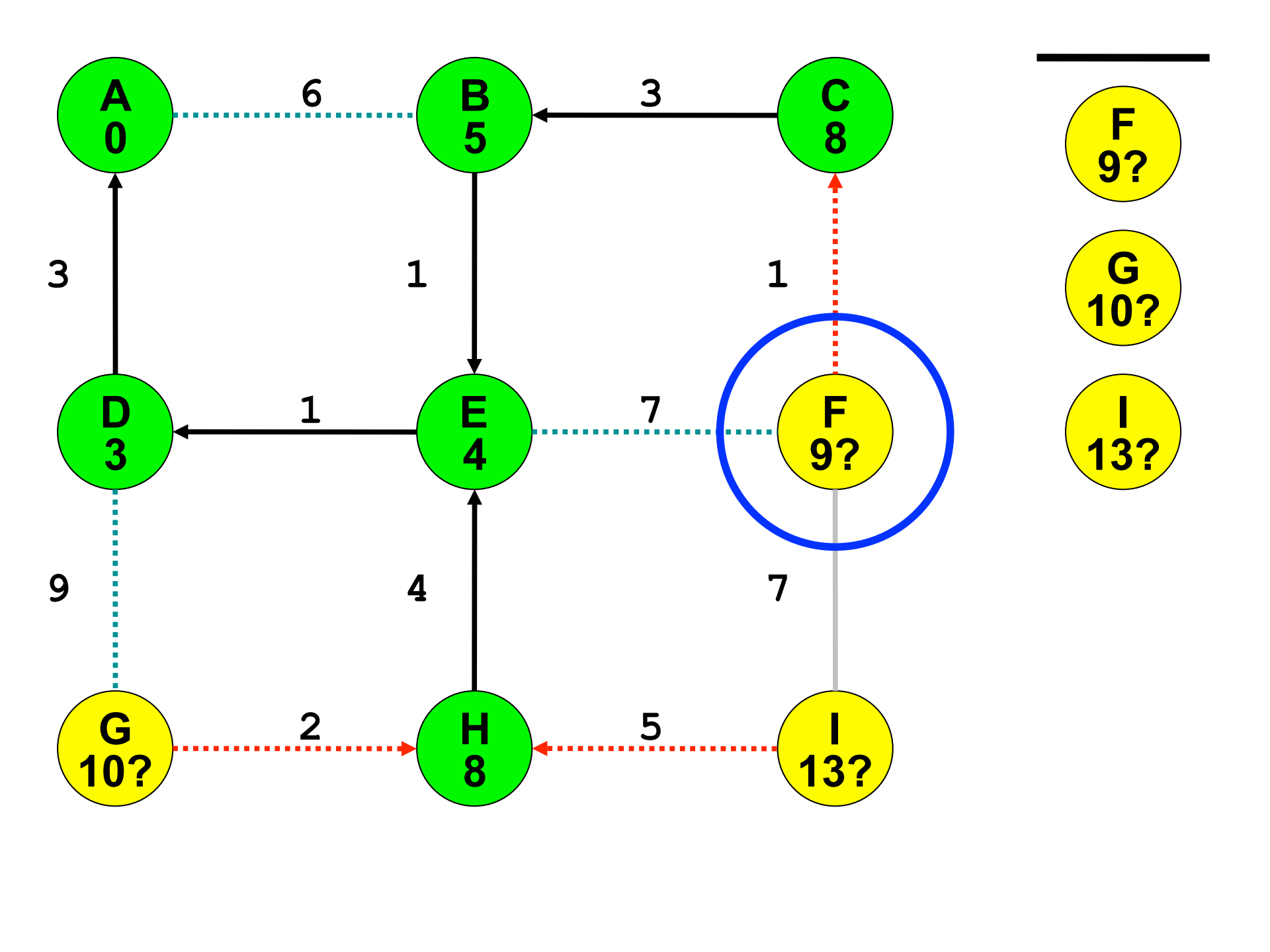

A
0
6
B
5
3
C
8
G
10?
3
1
1
F
9?
D
3
1
E
4
7
F
9?
I
13?
9
4
7
G
10?
2
H
8
5
I
13?

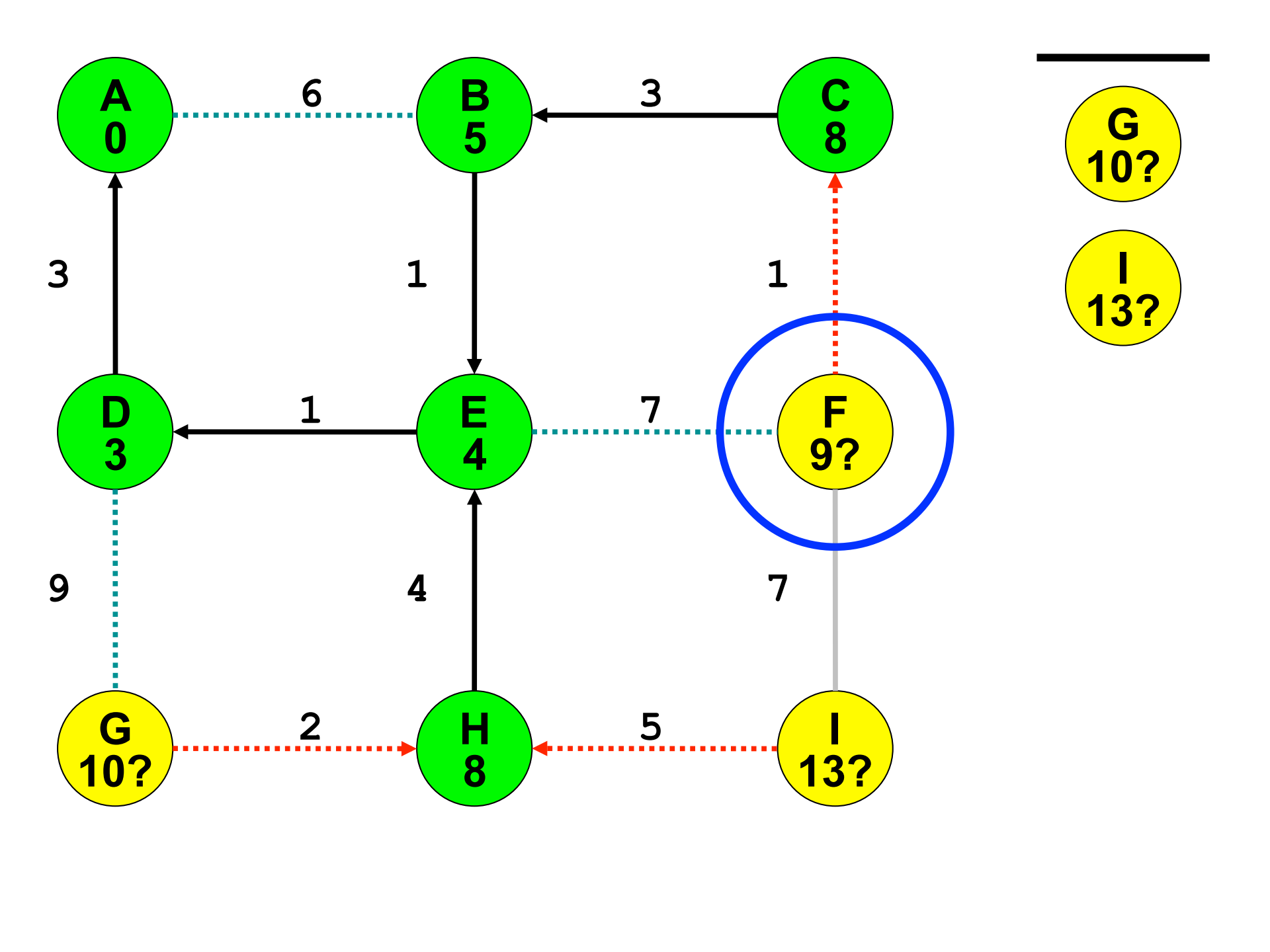

A
0
6
B
5
3
C
8
F
9?
3
1
1
G
10?
D
3
1
E
4
7
F
9?
I
13?
9
4
7
G
10?
2
H
8
5
I
13?

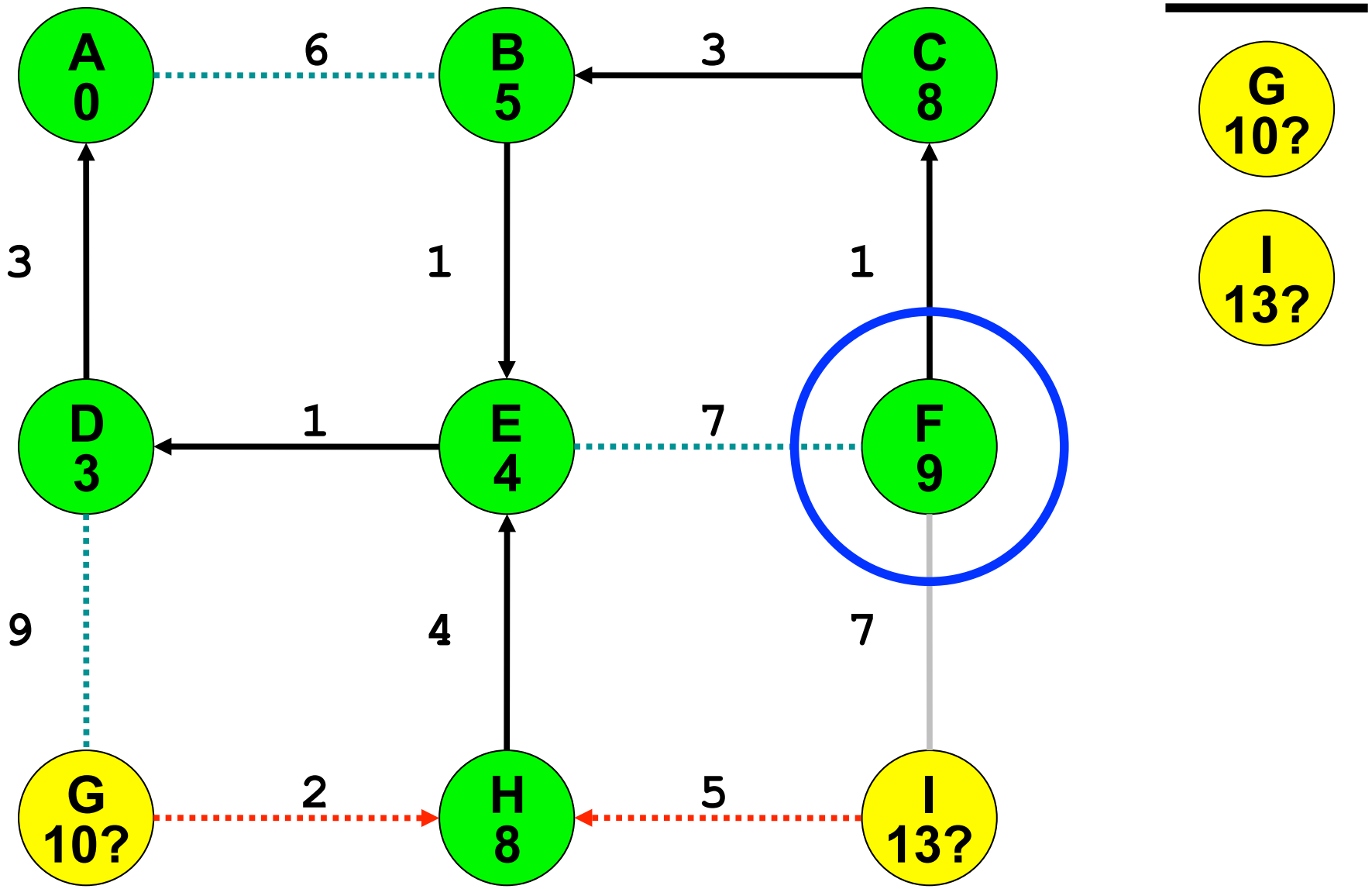

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?
F
9?
G
10?
I
13?

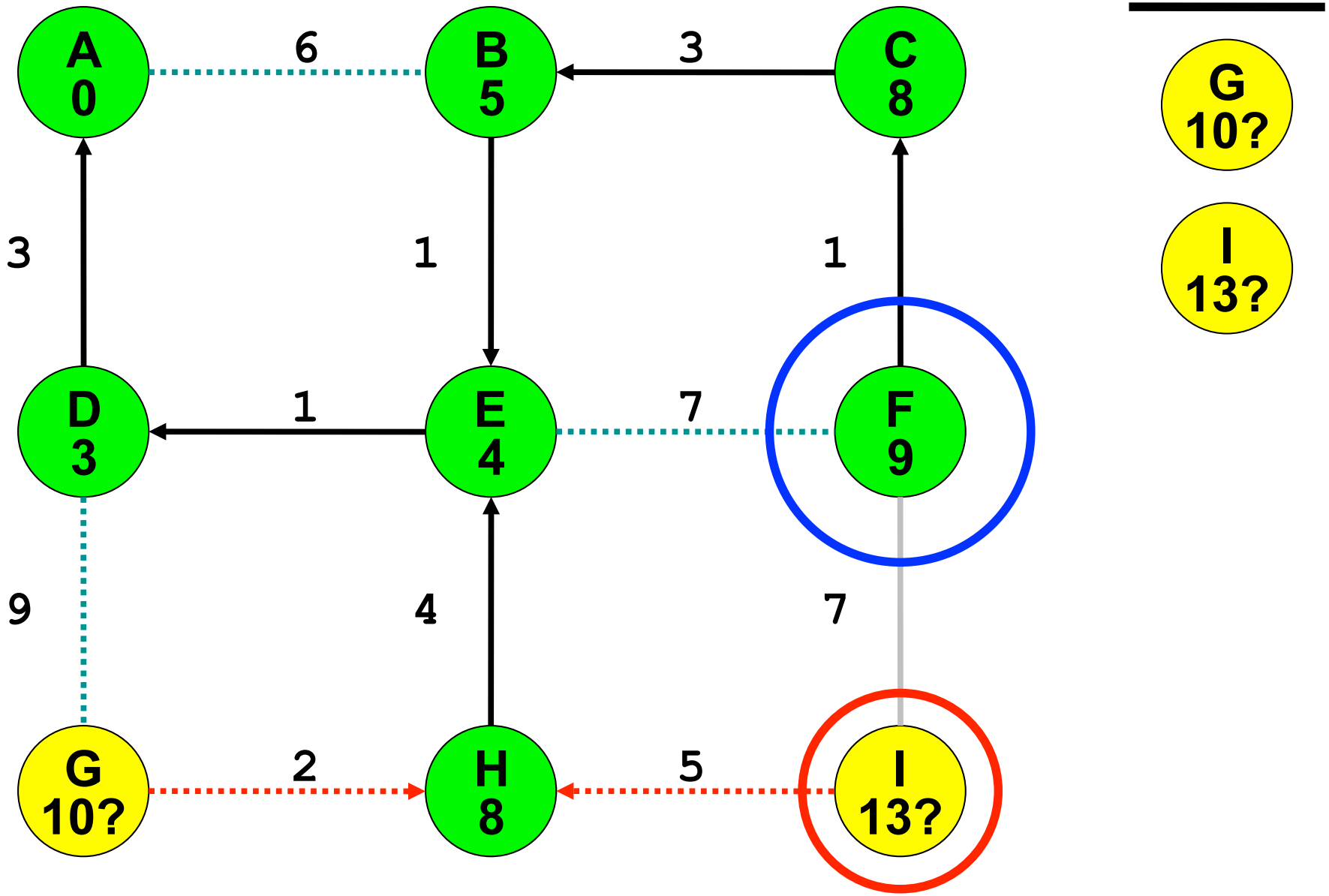

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?
F
9?
G
10?
I
13?

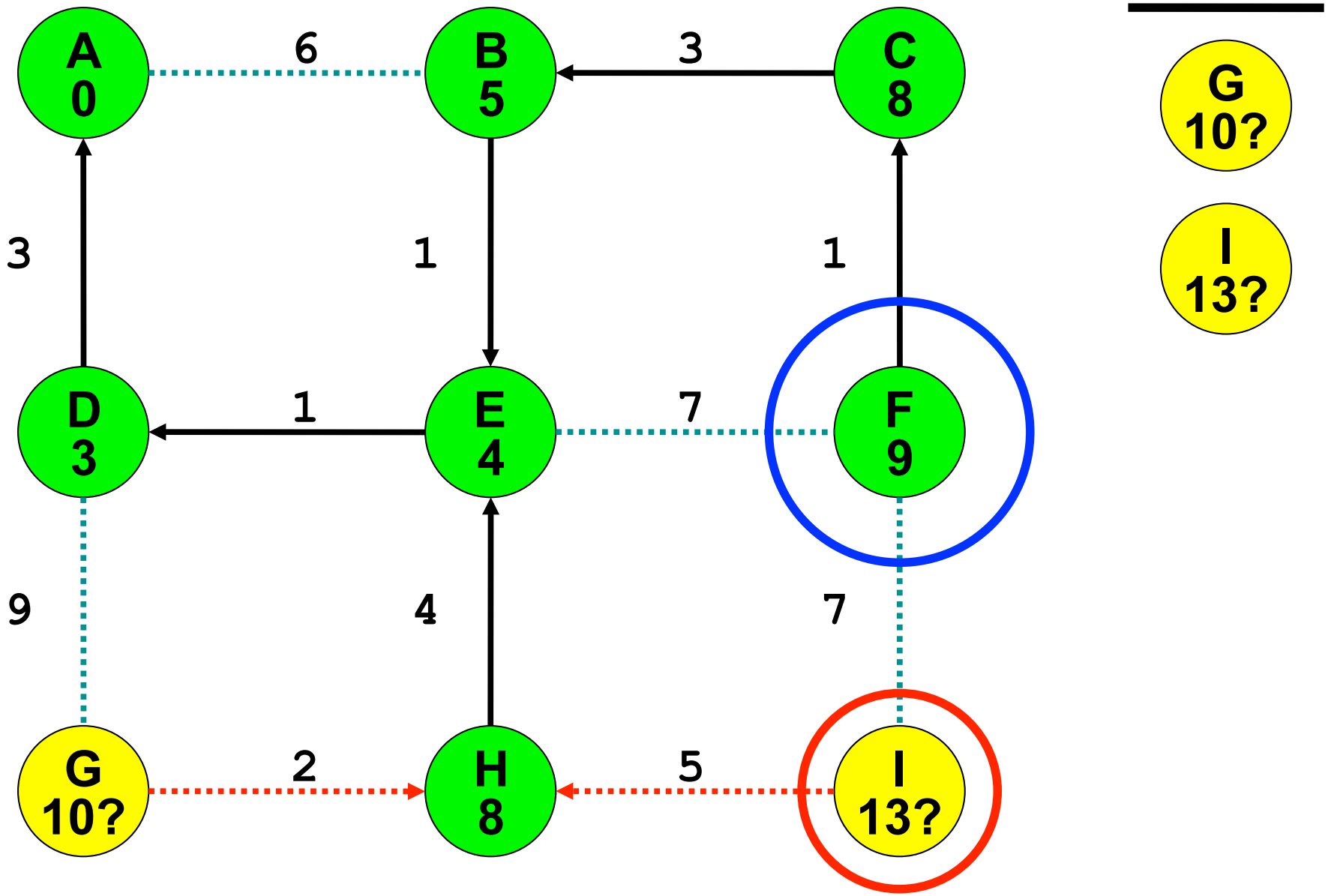

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?
F
9?
G
10?
I
13?

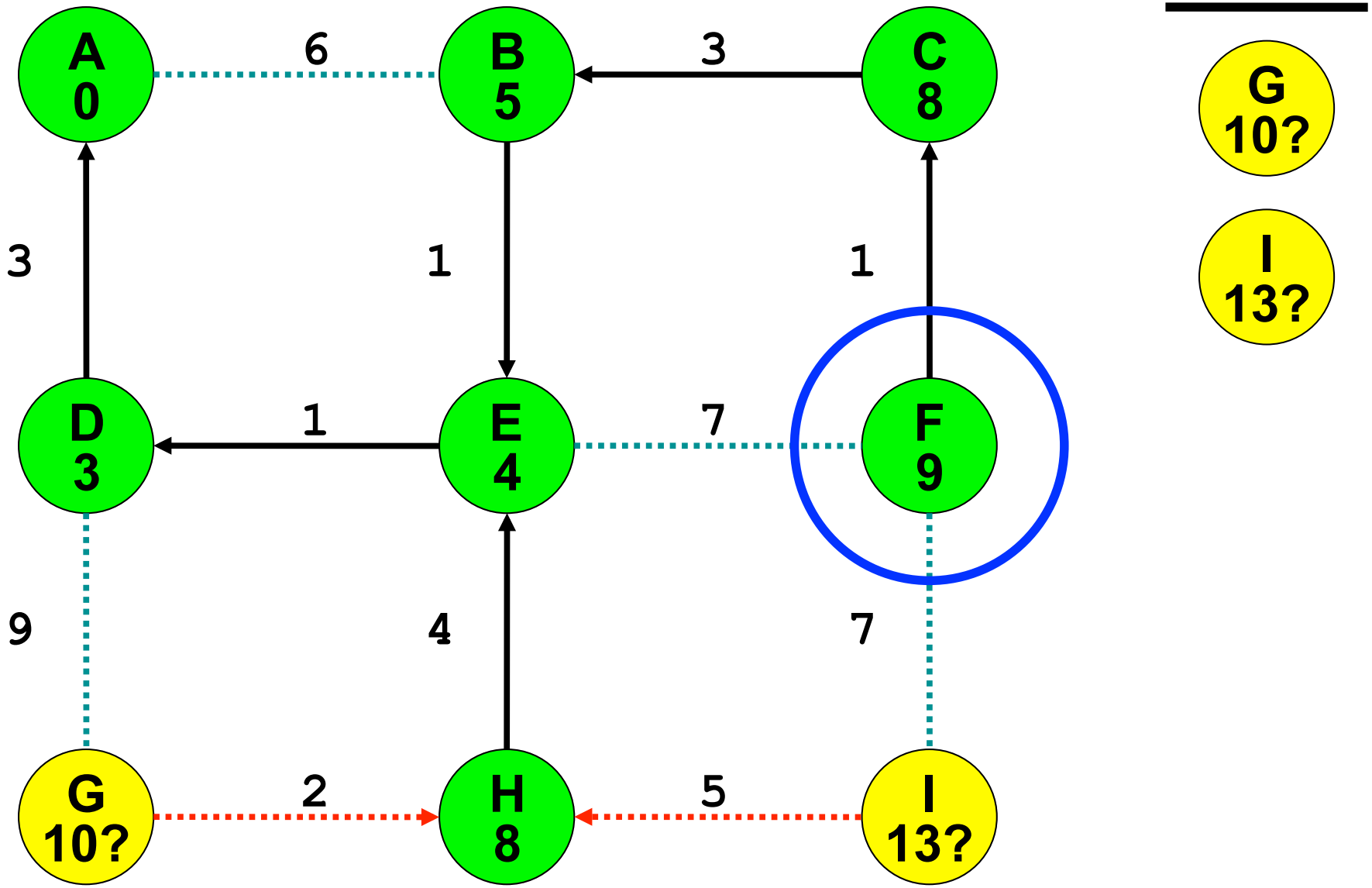

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?
F
9?
G
10?
I
13?

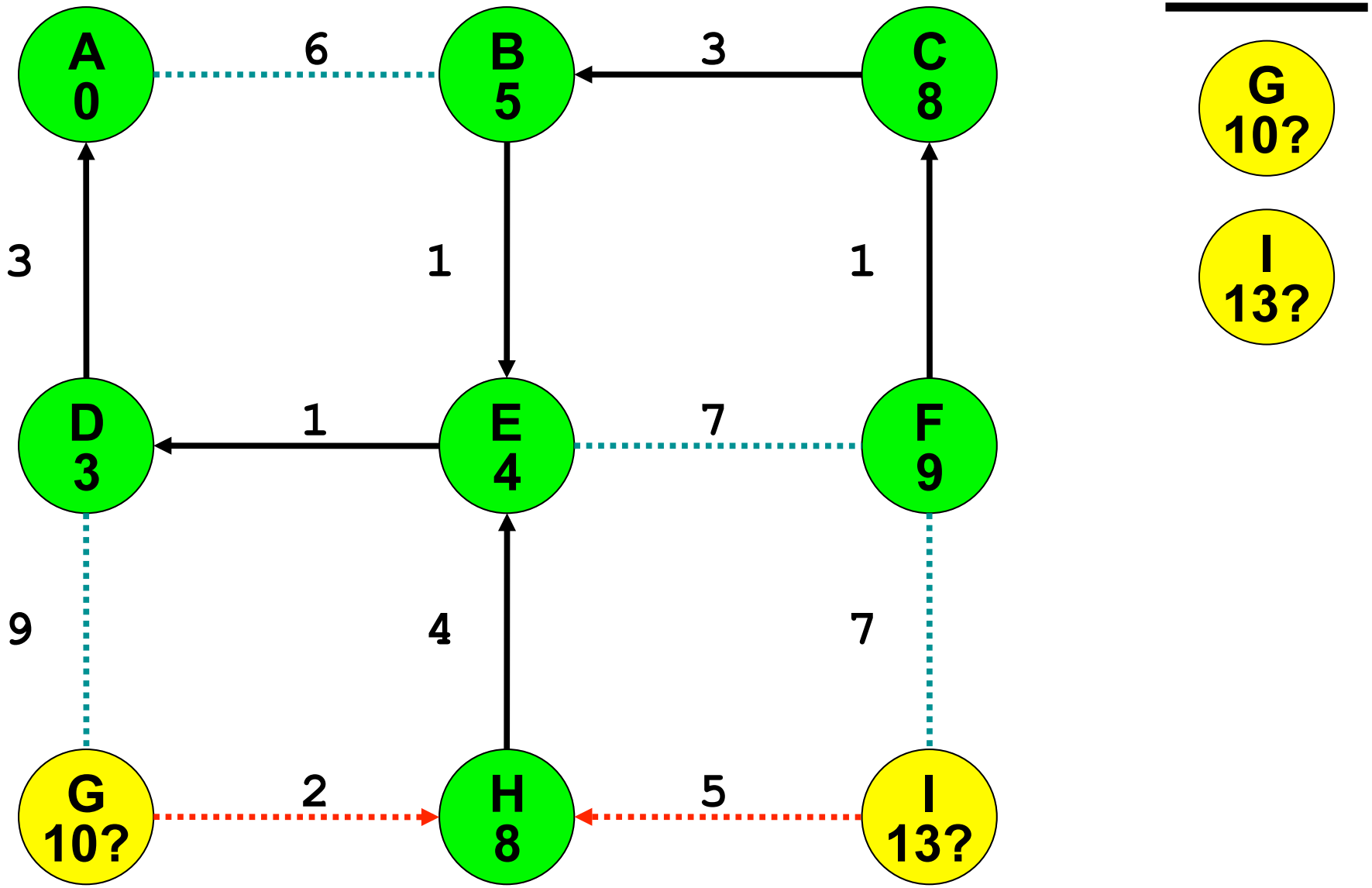

A
0
6
B
5
3
C
8
G
10?
I
13?
3
1
1
D
3
1
E
4
7
F
9?
9
4
7
G
10?
2
H
8
5
I
13?

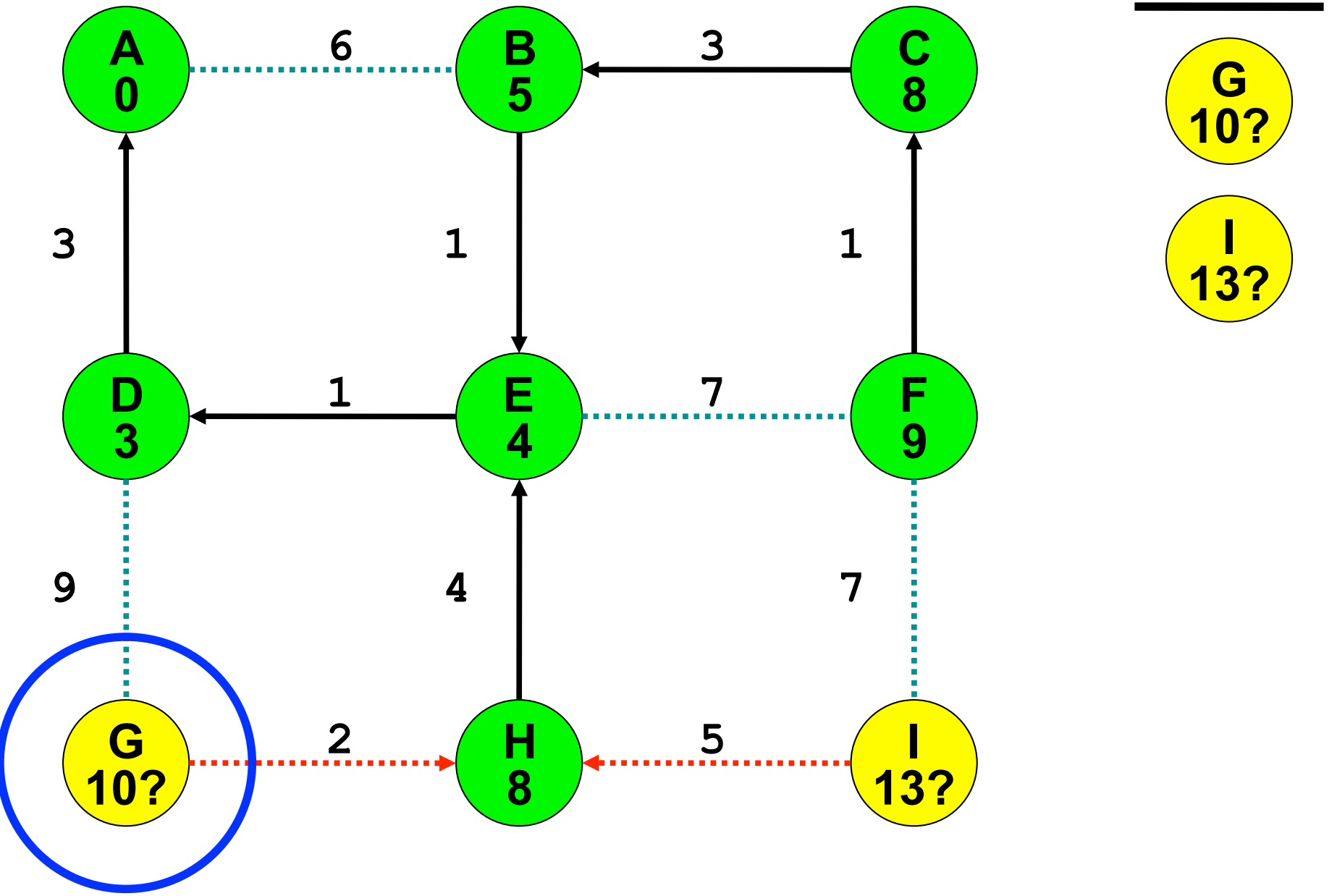

A
0
6
B
5
3
C
8
G
10?
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?

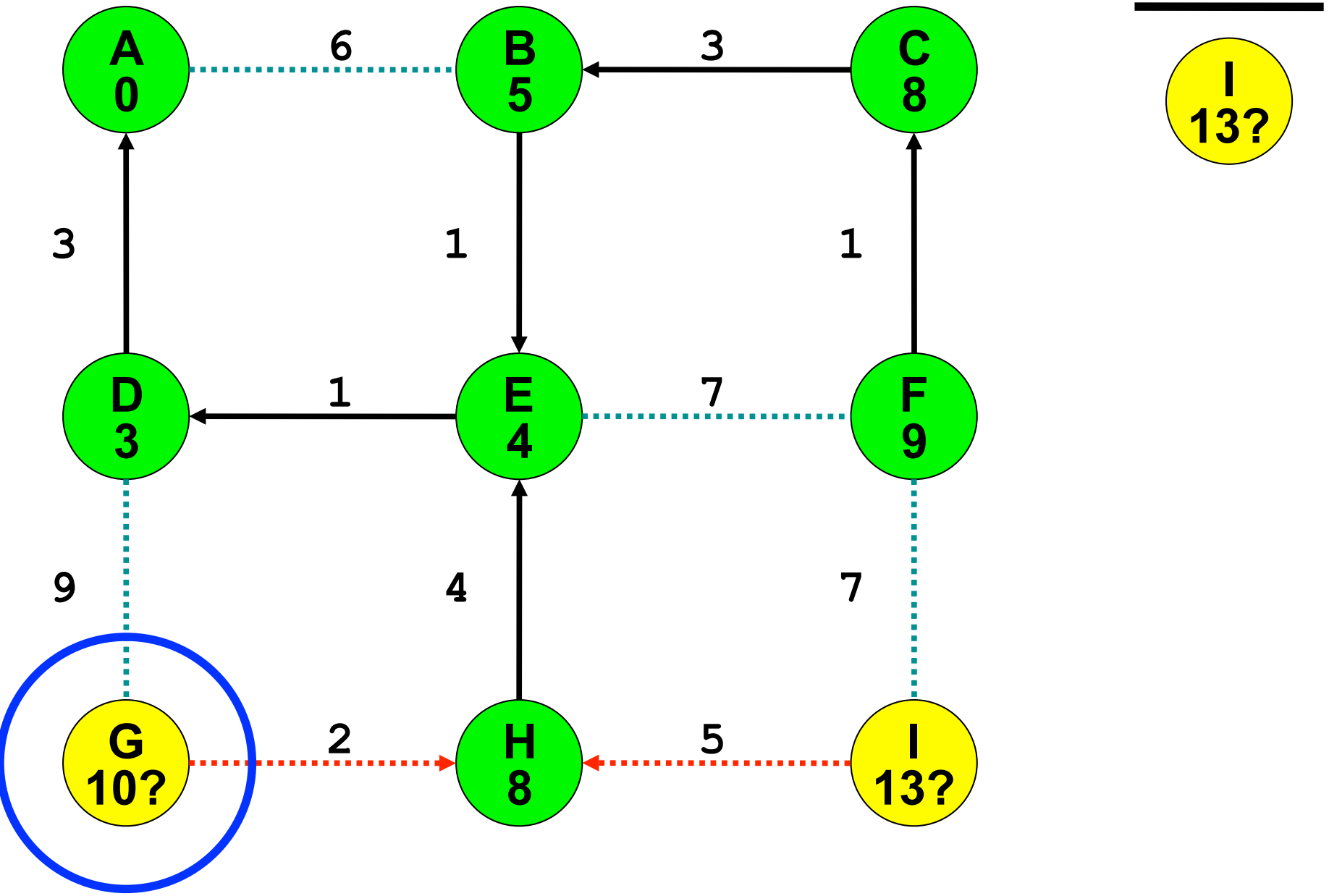

A
0
6
B
5
3
C
8
G
10?
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?

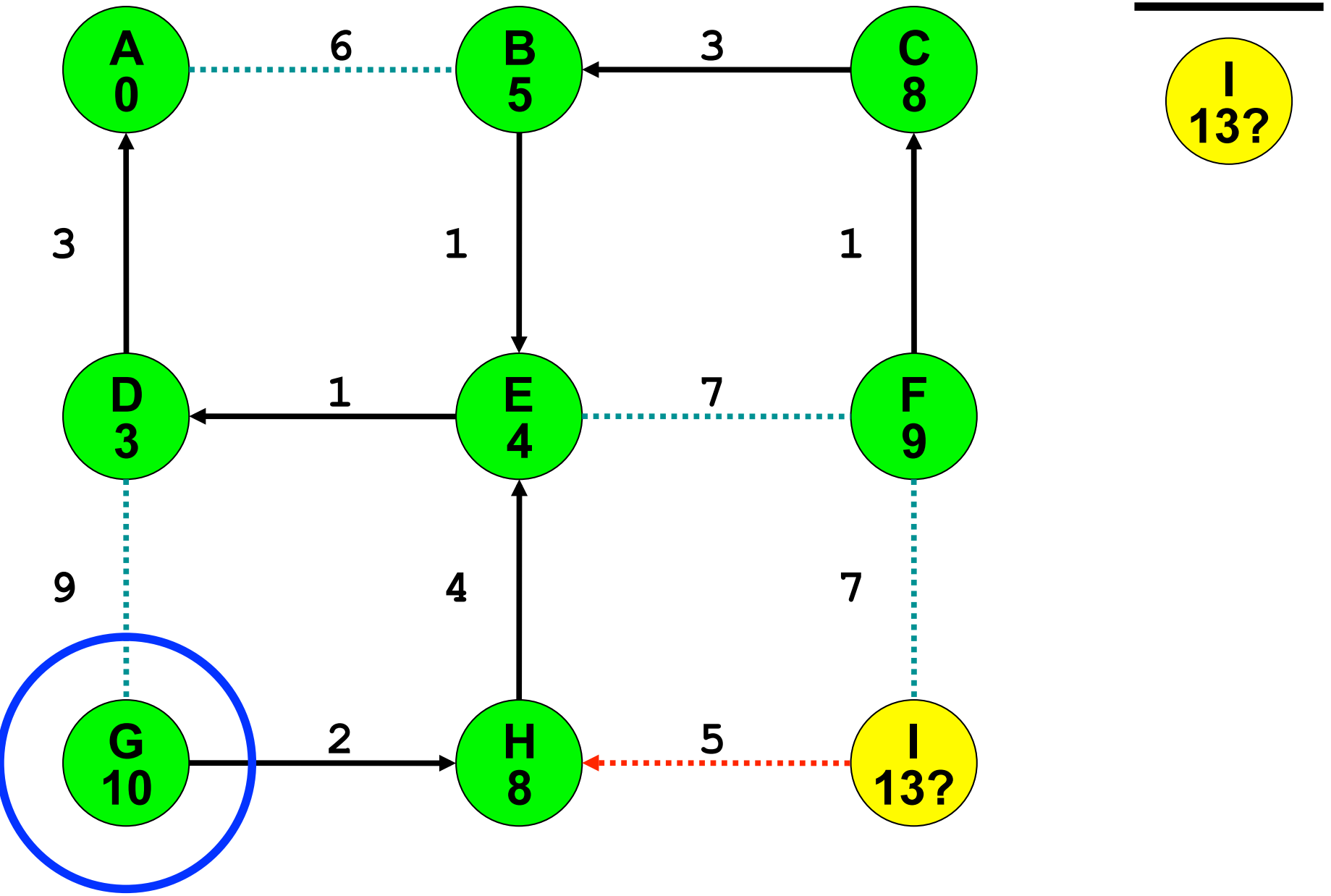

A
0
6
B
5
3
C
8
G
10?
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?

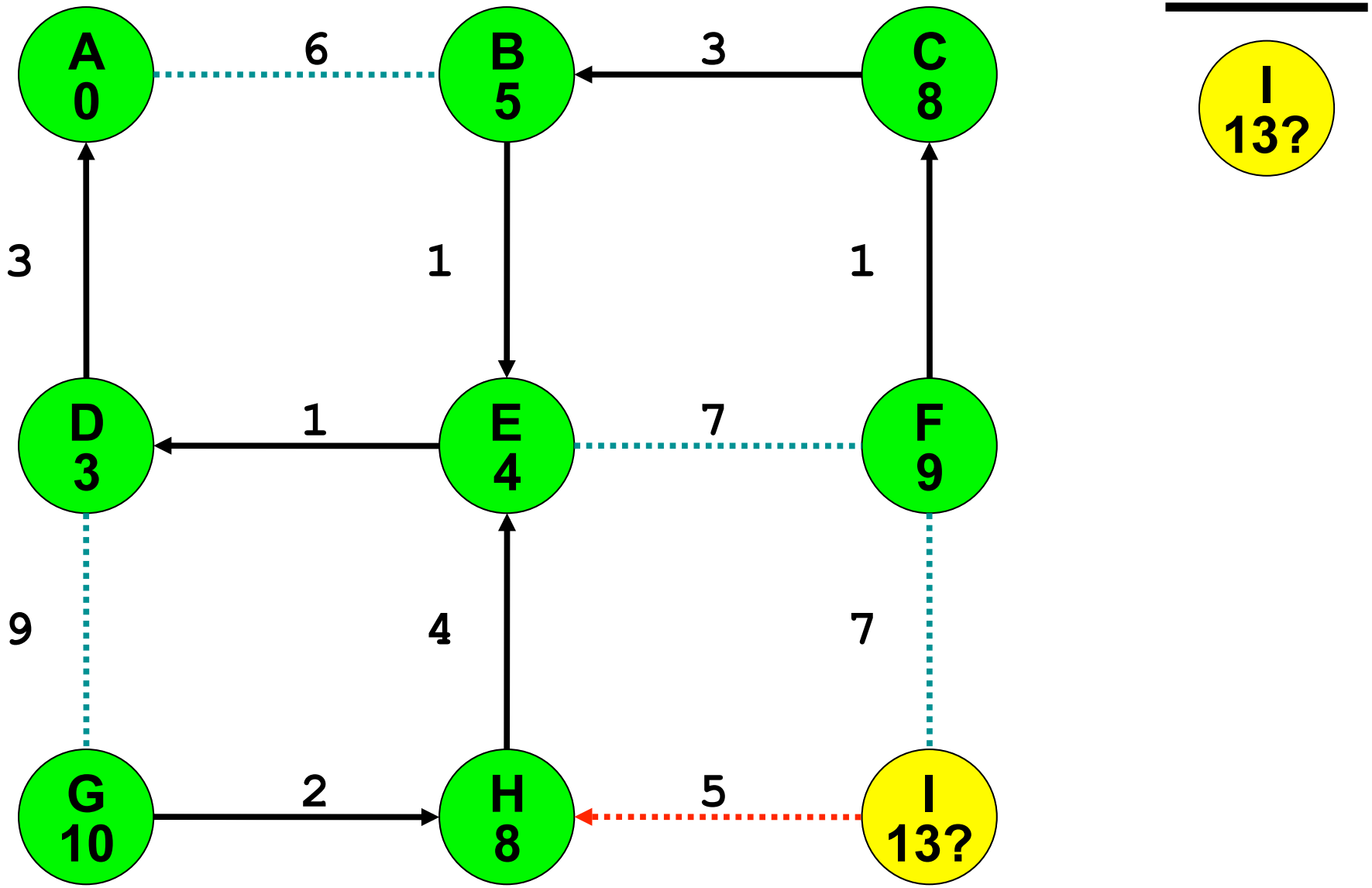

A
0
6
B
5
3
C
8
G
10?
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?

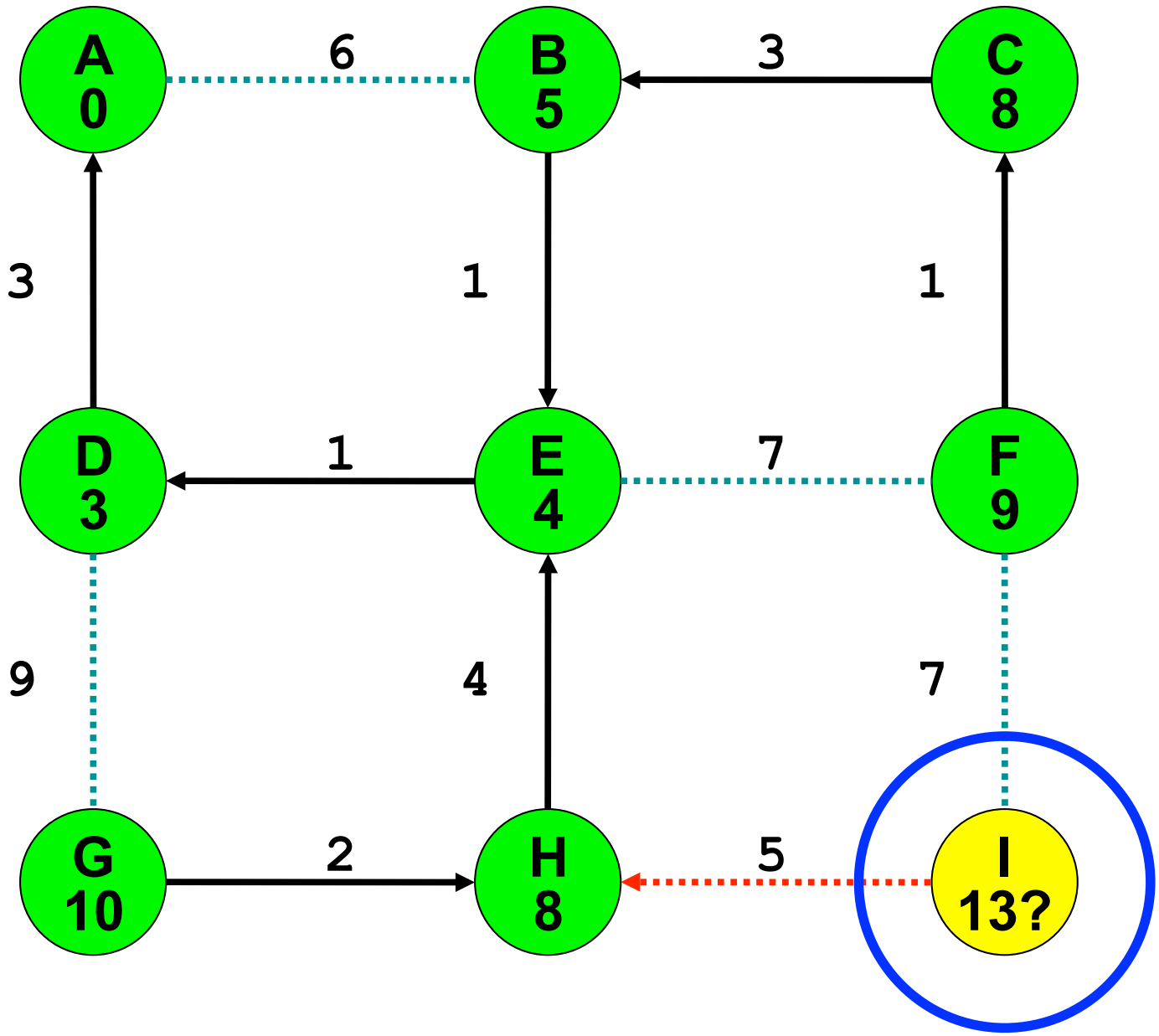

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
I
13?

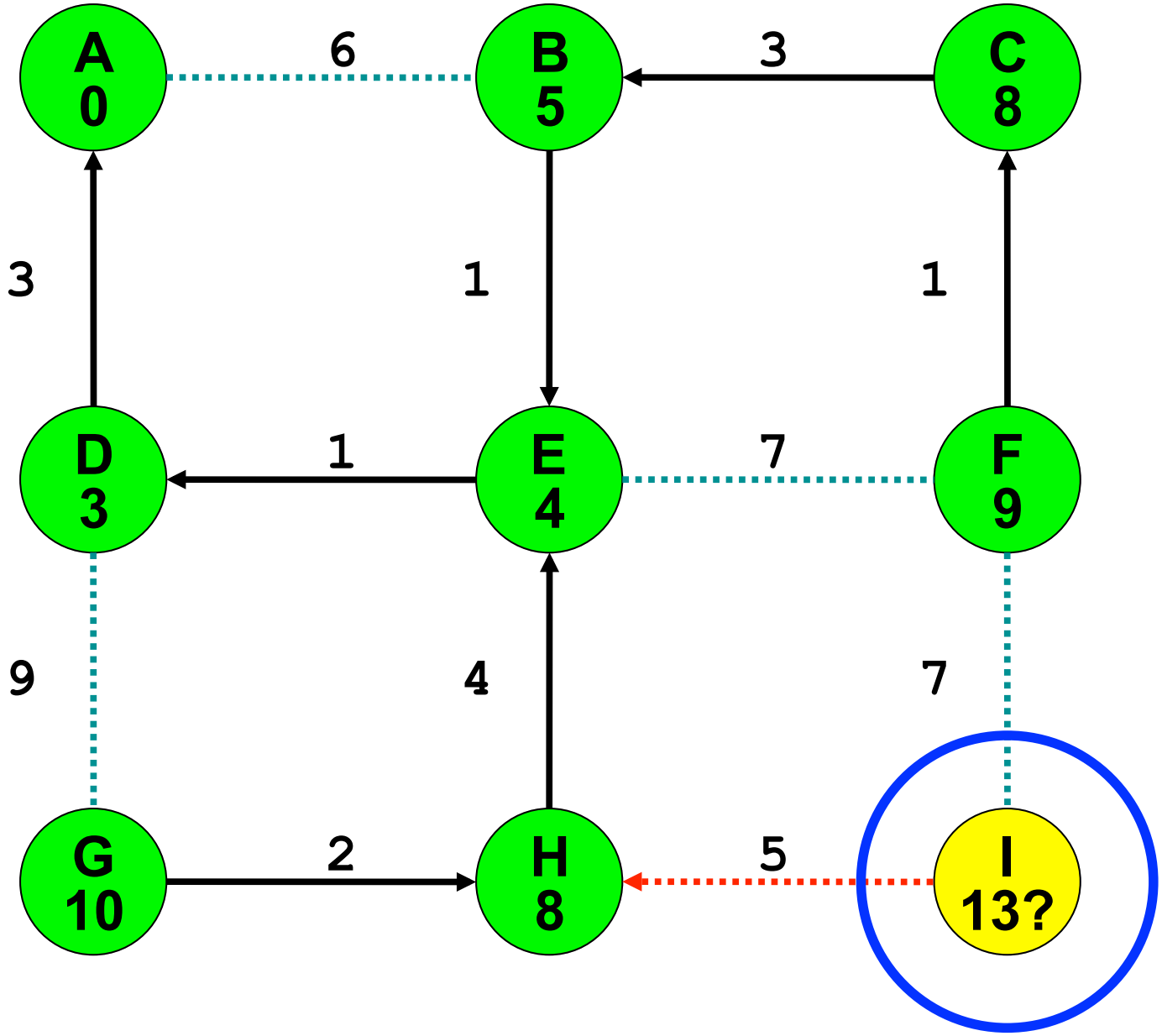

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?
G
10?
I
13?

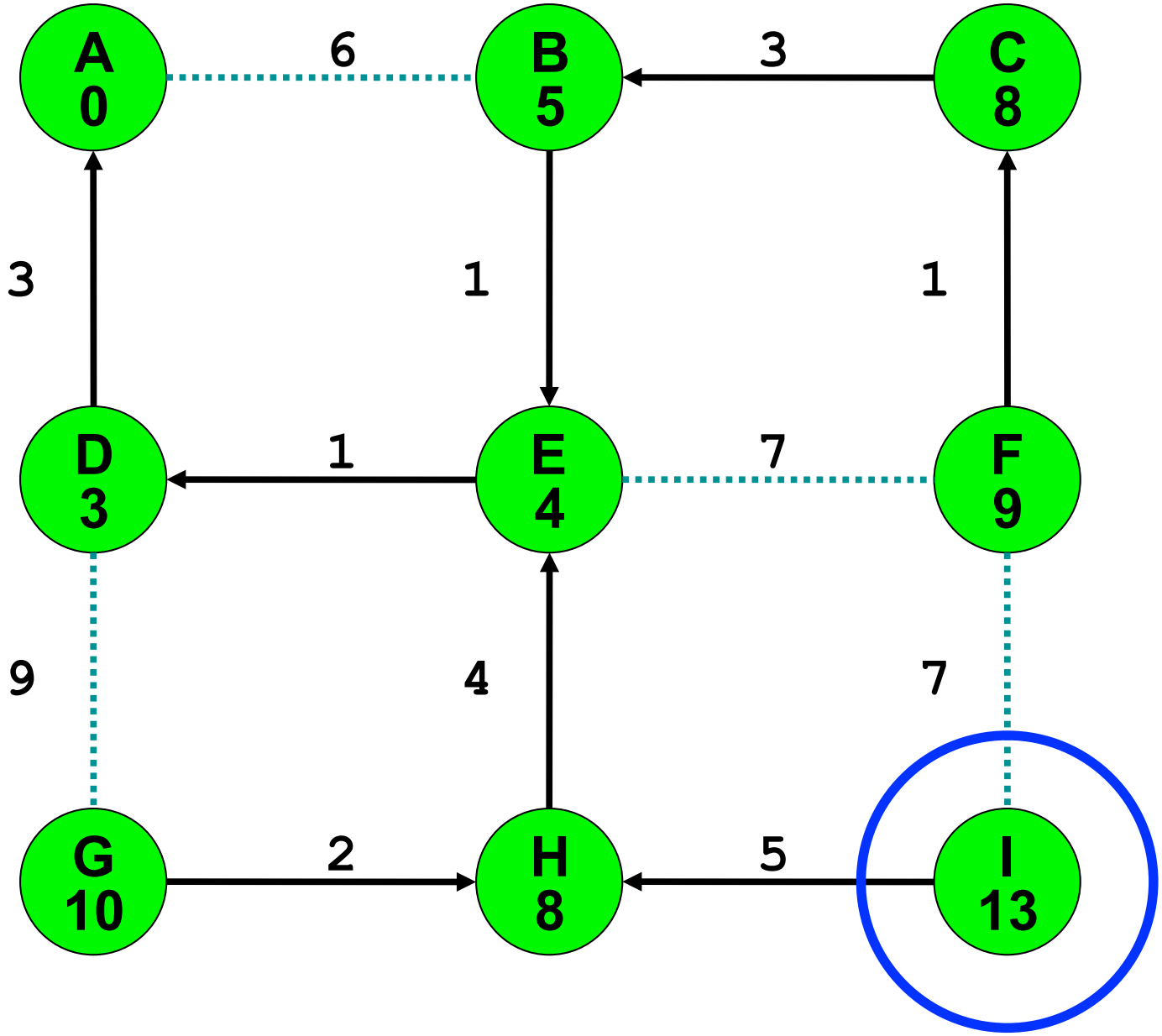

A
0
6
B
5
3
C
8
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10?
2
H
8
5
I
13?

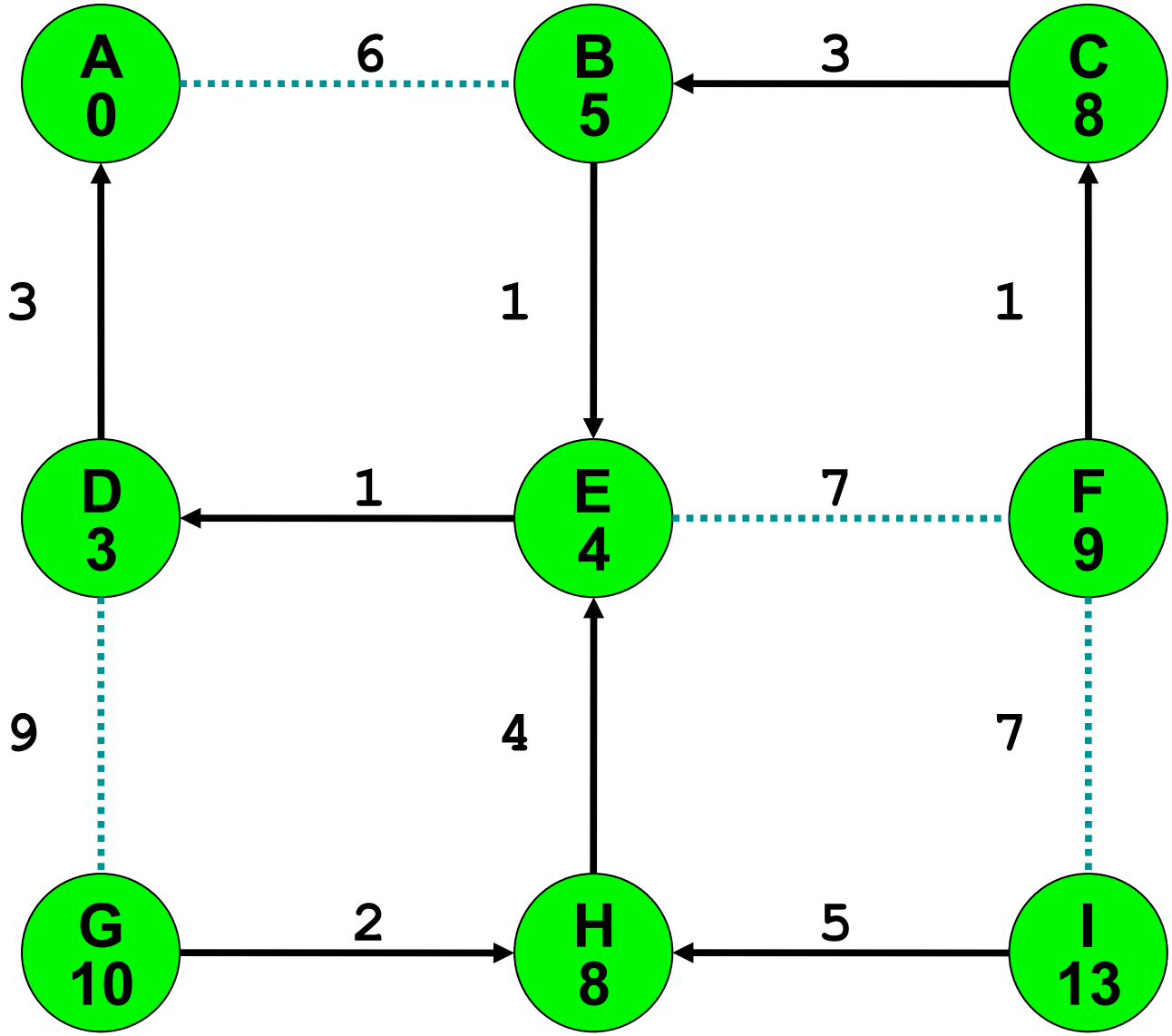

A
0
6
B
5
3
C
8
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13?


A
0
6
B
5
3
C
8
I
13?
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13?

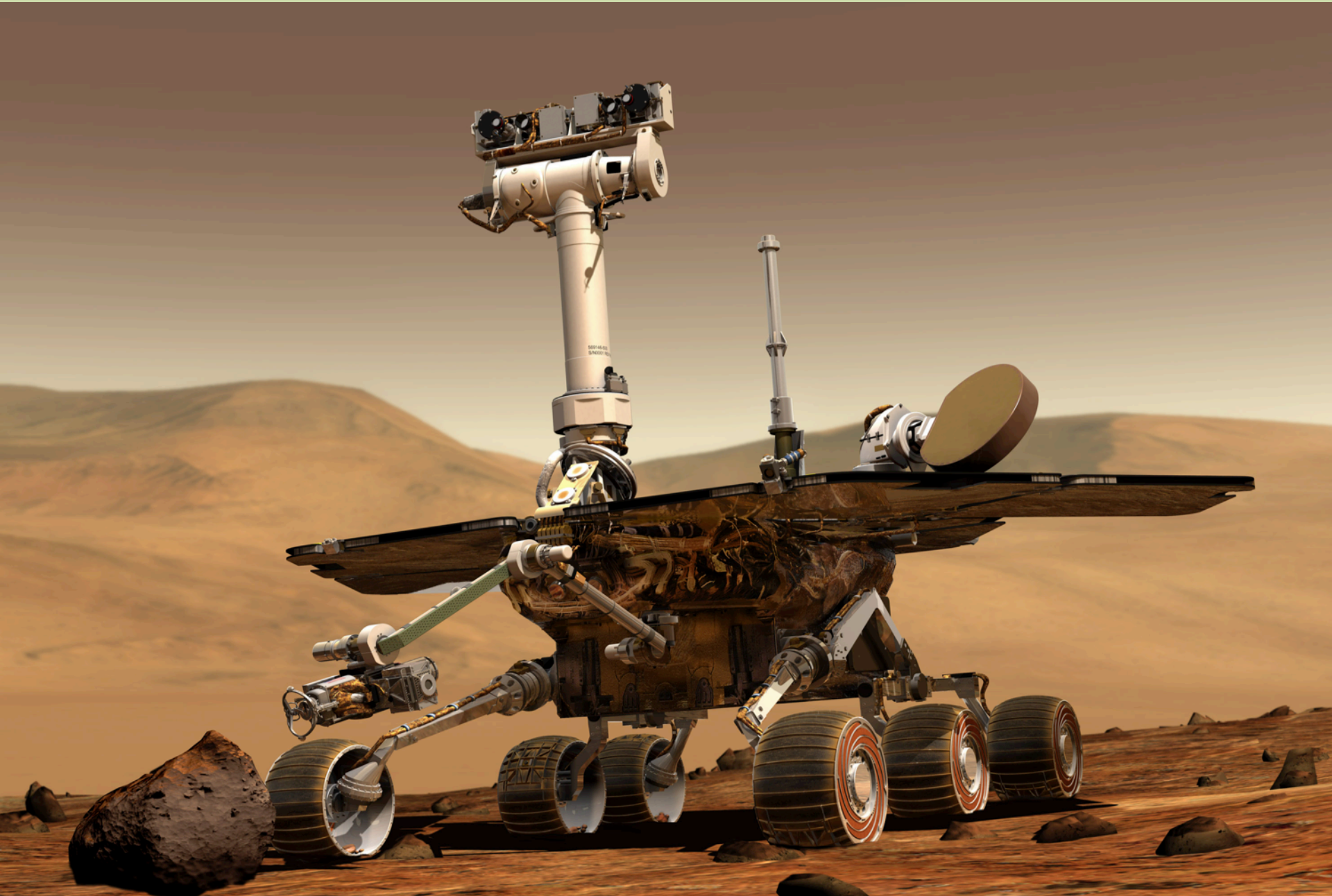

A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13?
I
13?


A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13?


A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13


A
0
6
B
5
3
C
8
3
1
1
D
3
1
E
4
7
F
9
9
4
7
G
10
2
H
8
5
I
13

Robot Pathfinding

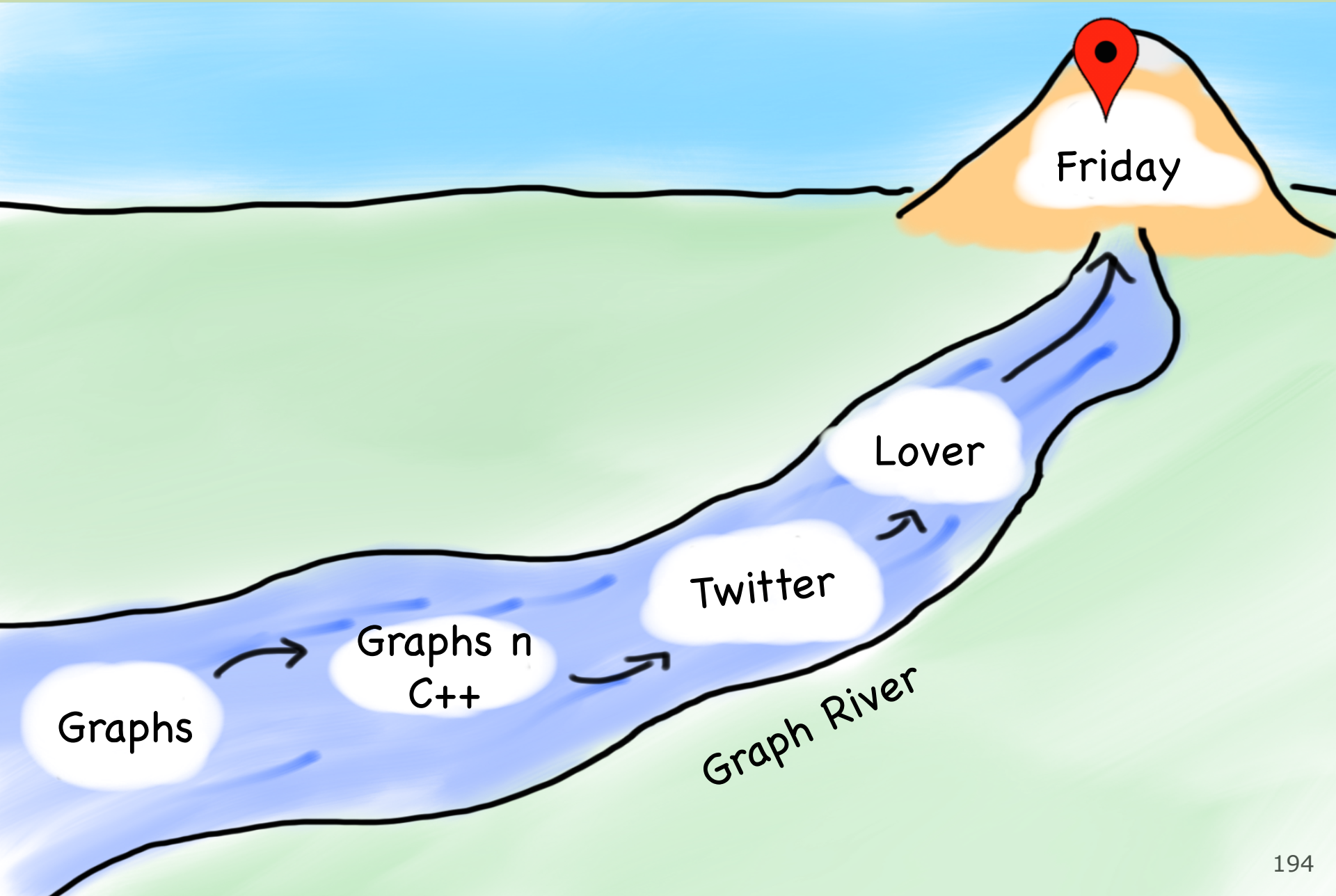


Search is Important

Summary

Algorithm	Collection	Benefit
Depth First Search	Stack	Uses little memory
Breadth First Search	Queue	Finds shortest hops
Dijkstra's	Priority Queue	Finds shortest path

Today's Route



Today's Goal

1. Depth First Search
2. Breadth First Search
3. Chose Between the Two