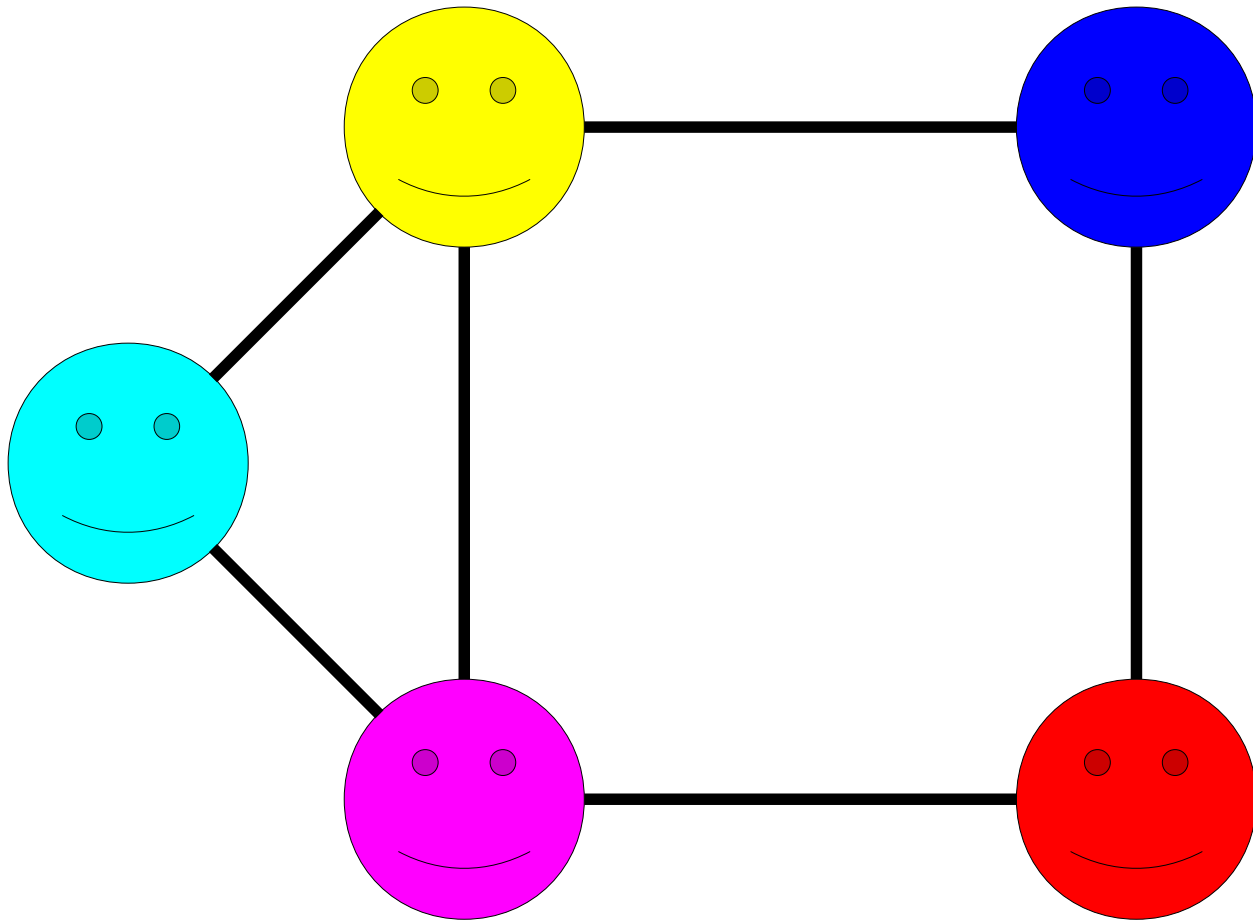
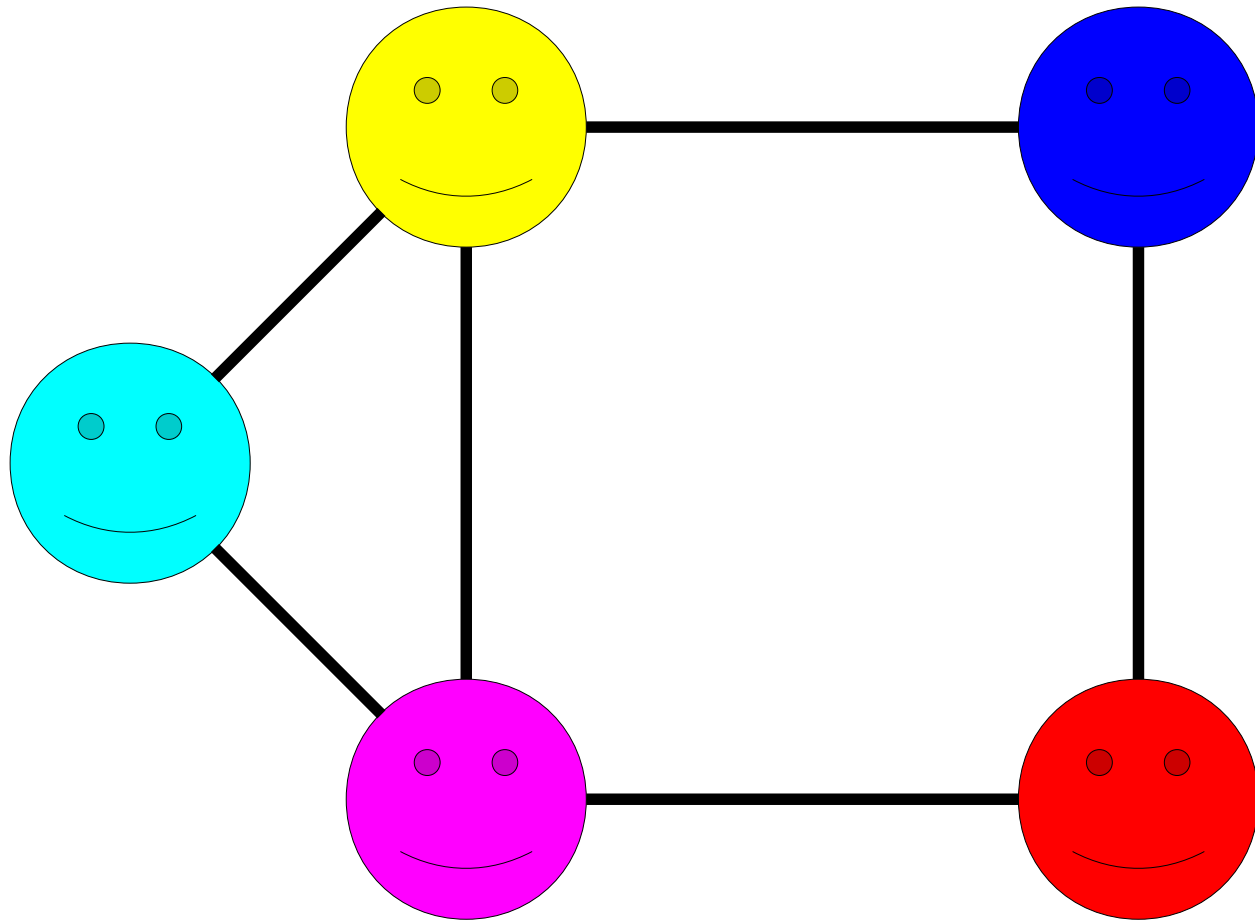


# Shortest Paths

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

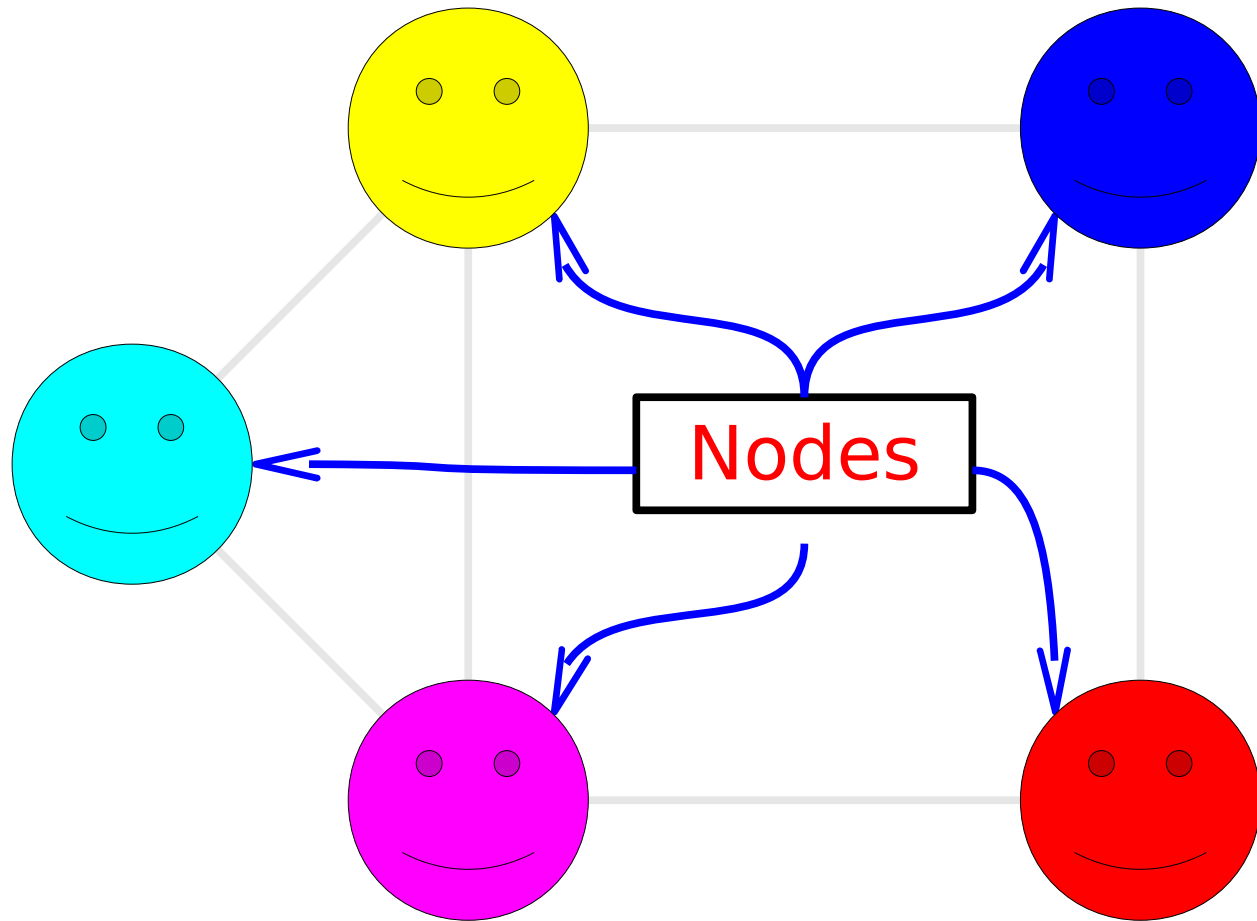


A **graph** is a mathematical structure for representing relationships.



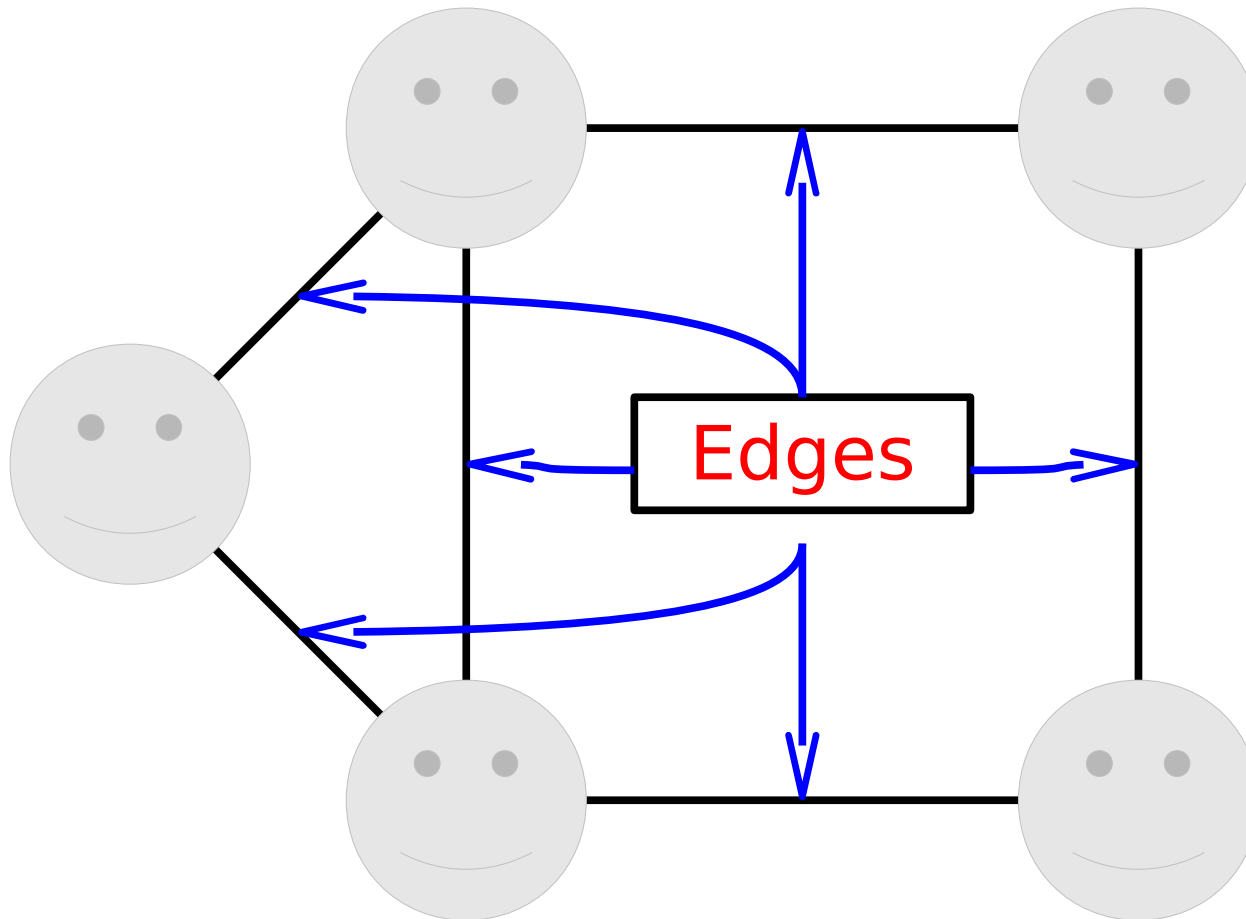
A graph consists of a set of **nodes** connected by **edges**.

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



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

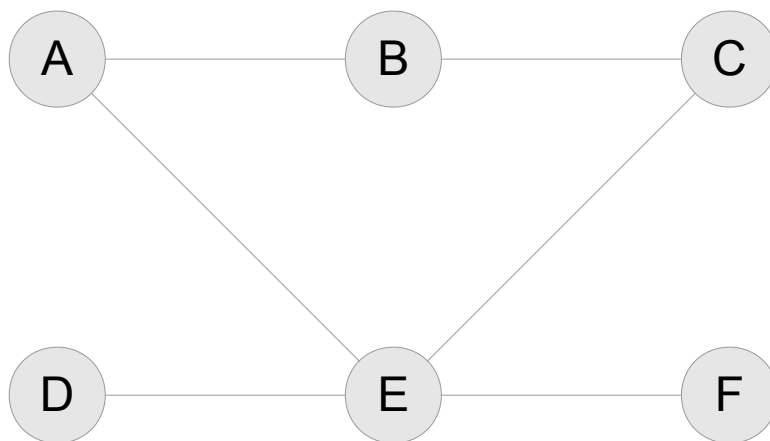
A **graph** is a mathematical structure for representing relationships.



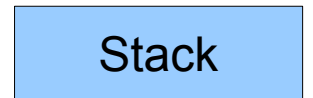
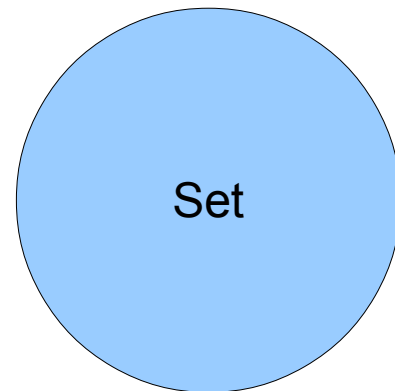
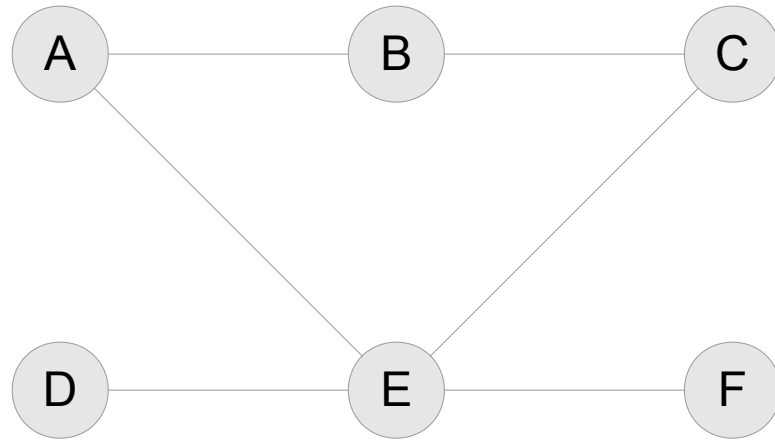
A graph consists of a set of **nodes** connected by **edges**.

# Depth-First Search

# Depth-first search

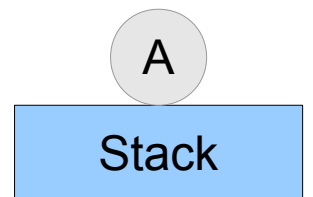
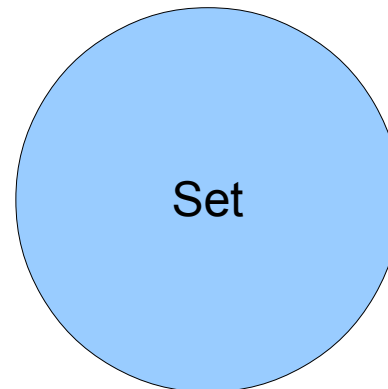
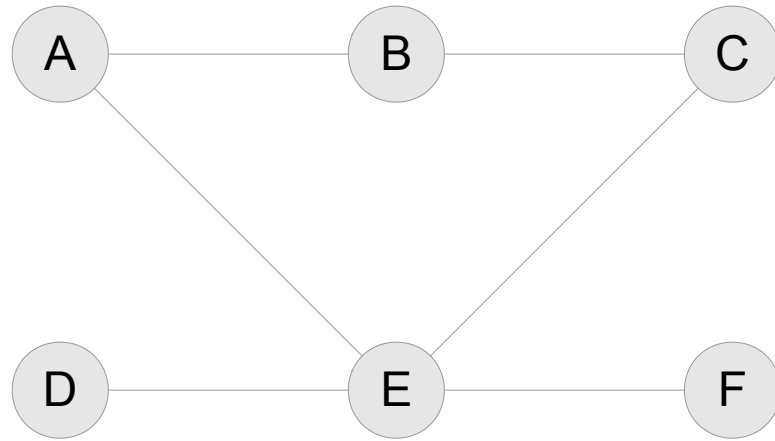


# Depth-first search

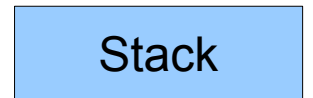
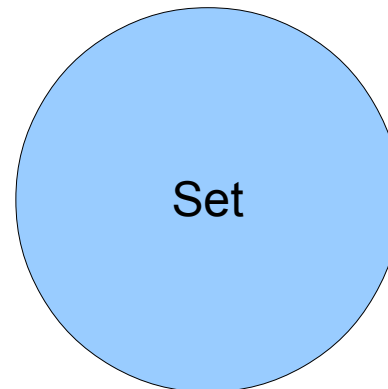
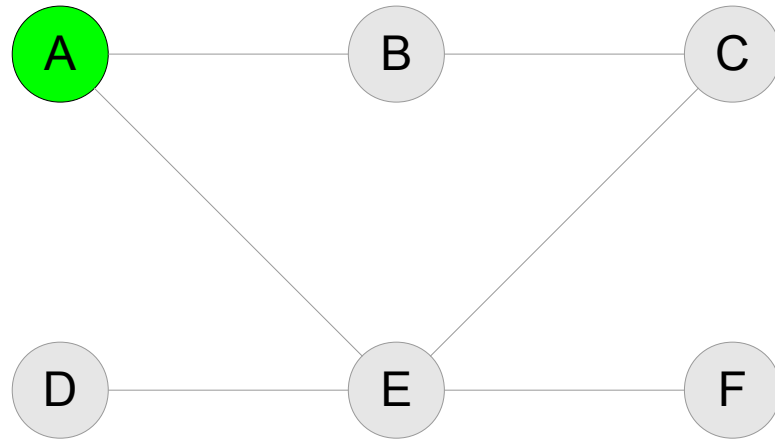




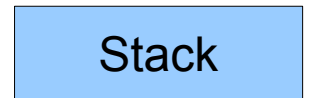
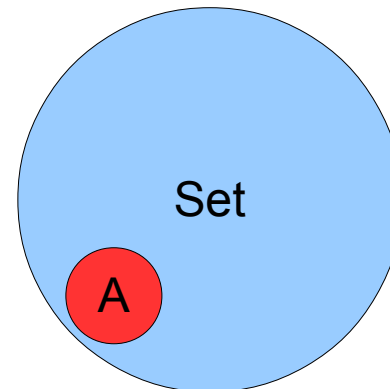
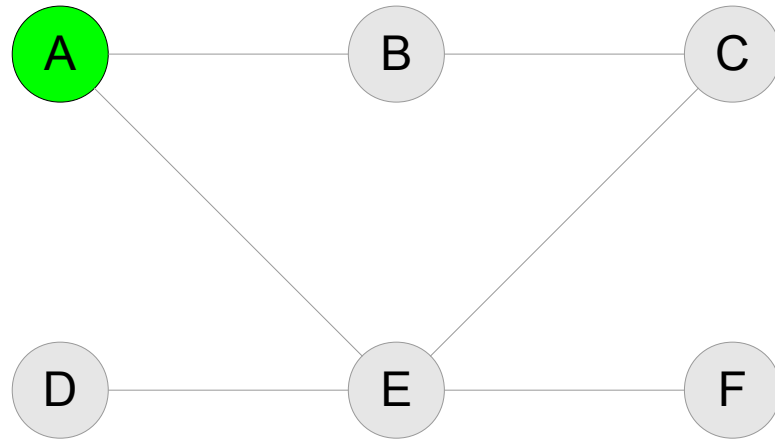
# Depth-first search



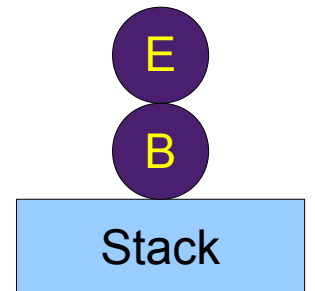
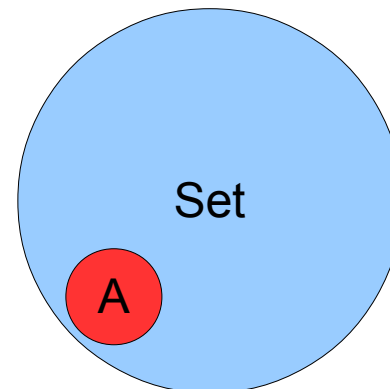
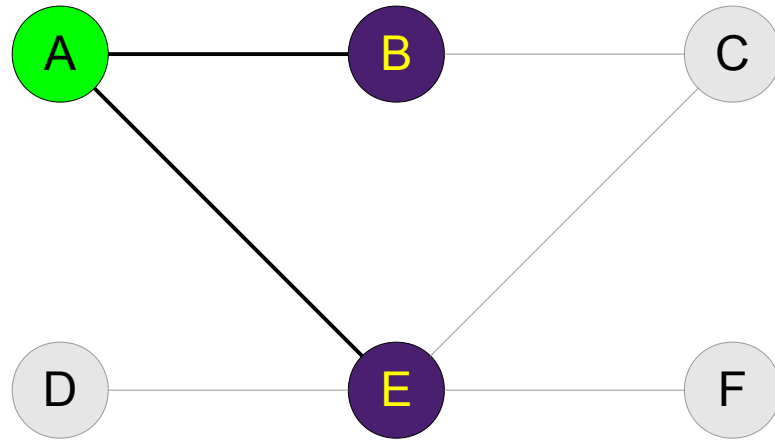
# Depth-first search



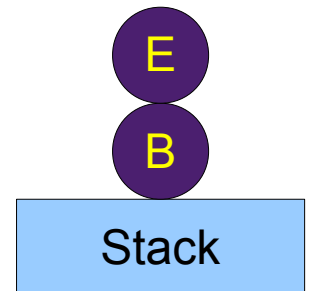
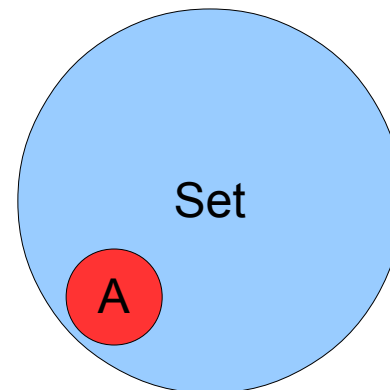
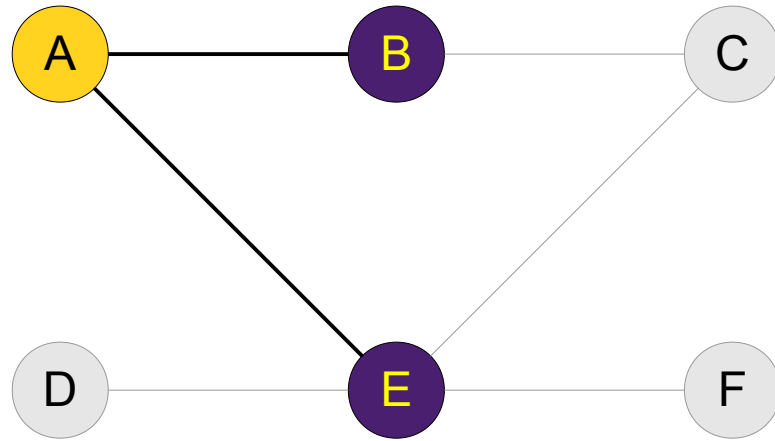
# Depth-first search



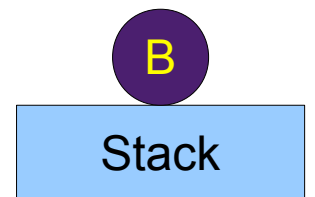
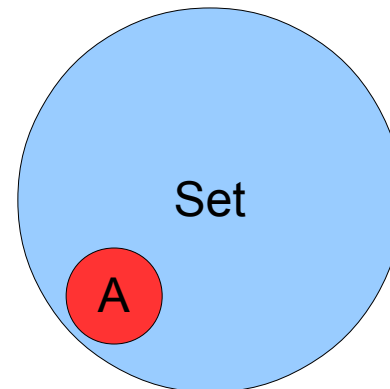
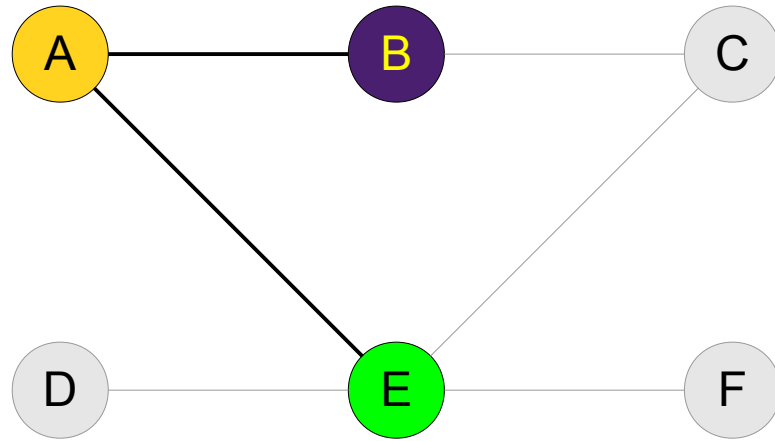
# Depth-first search



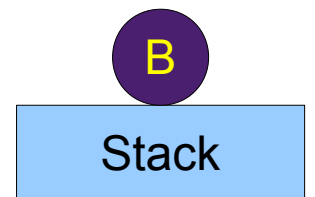
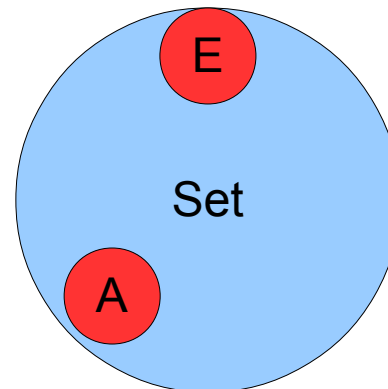
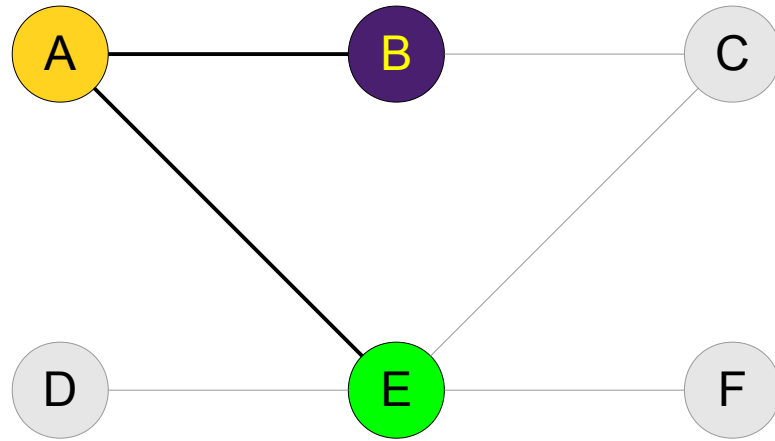
# Depth-first search



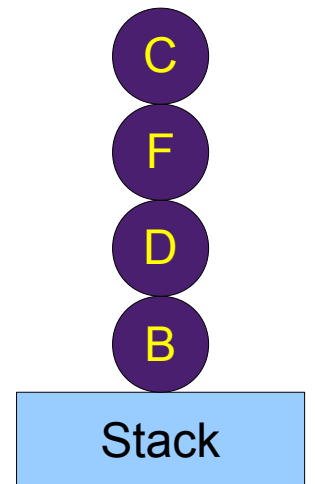
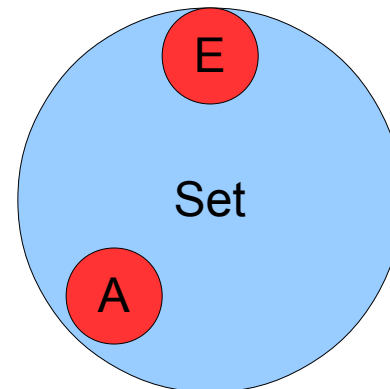
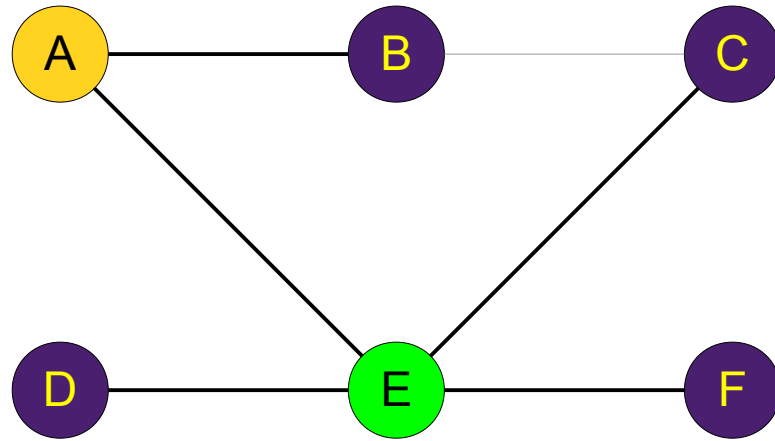
# Depth-first search



# Depth-first search

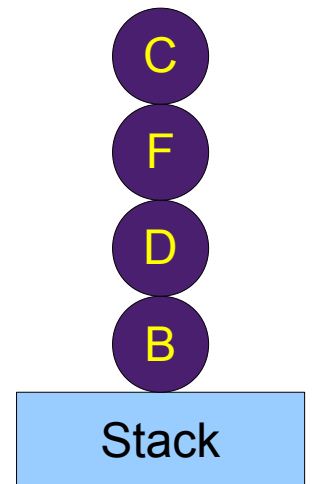
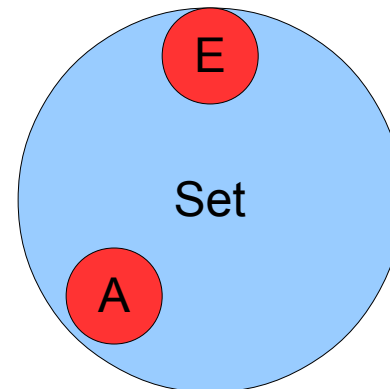
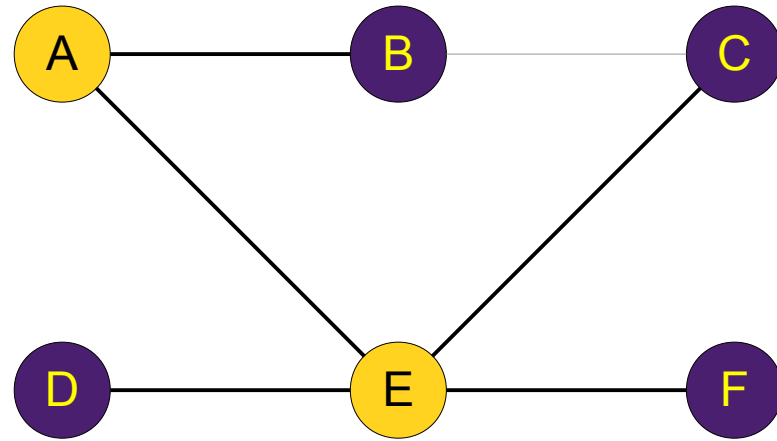


# Depth-first search

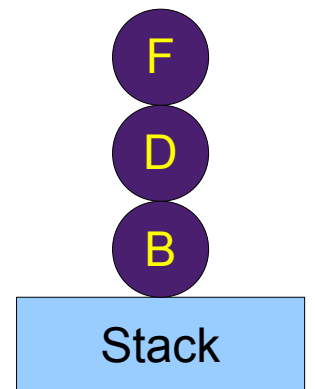
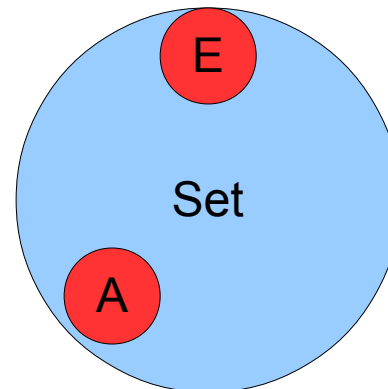
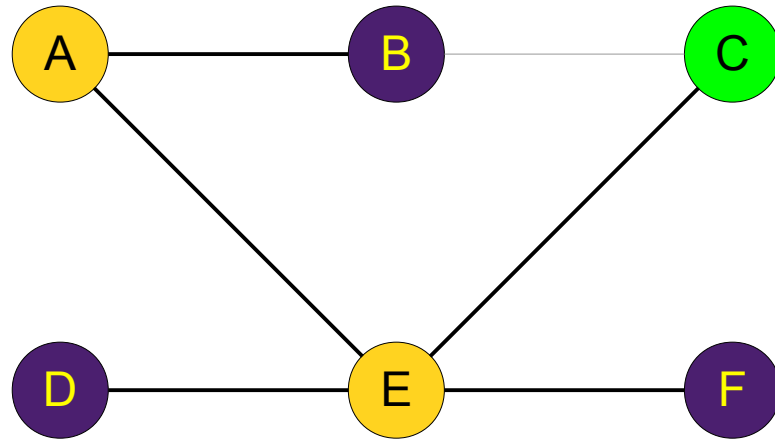




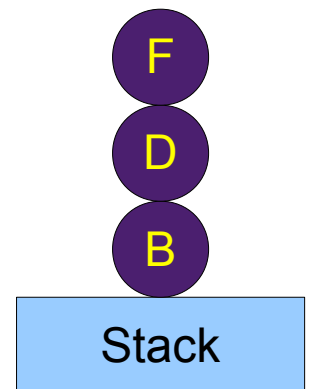
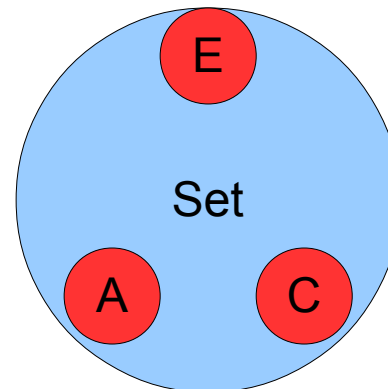
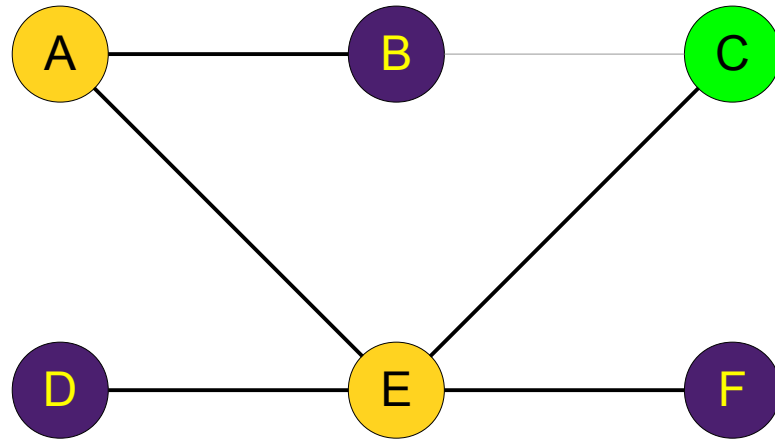
# Depth-first search



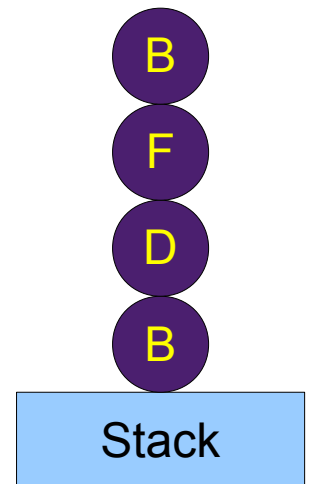
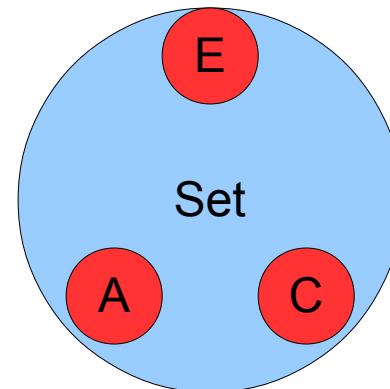
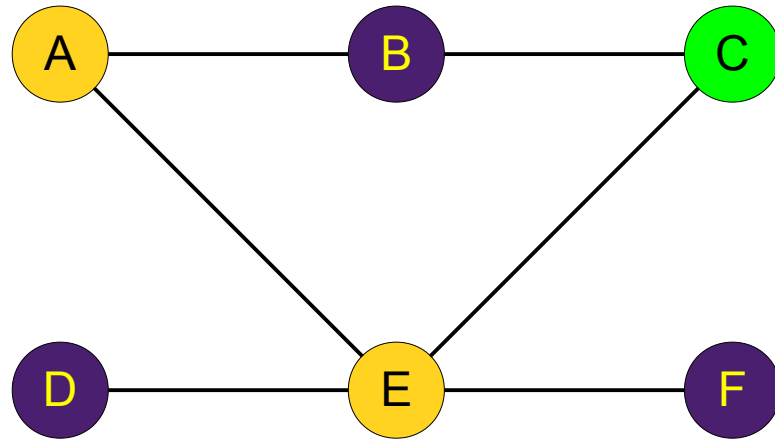
# Depth-first search



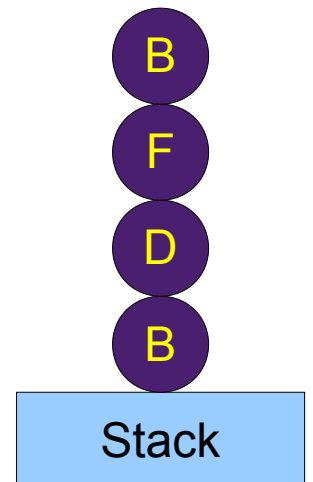
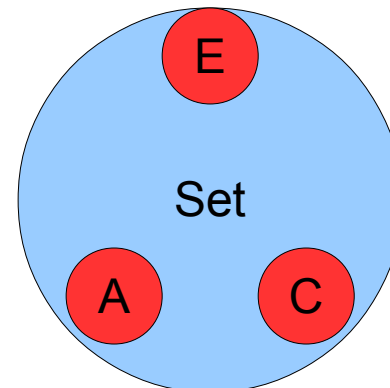
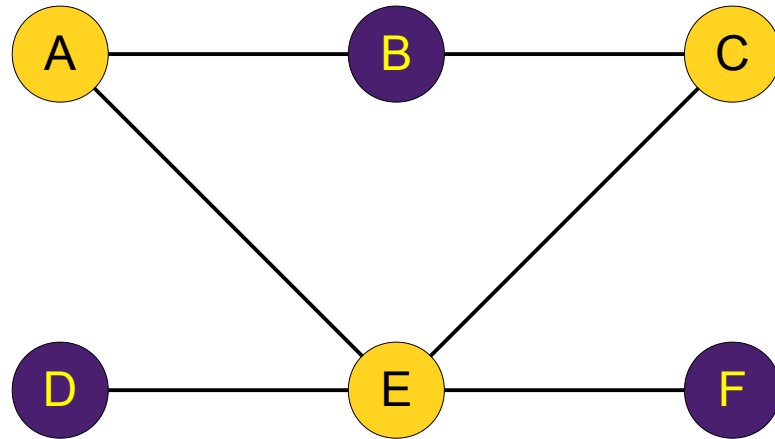
# Depth-first search



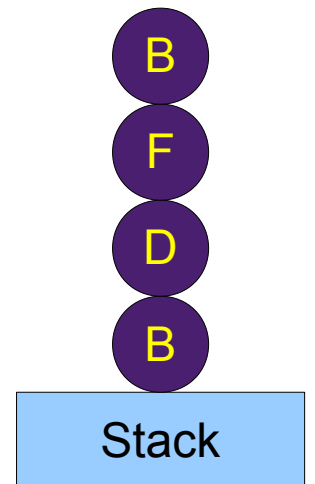
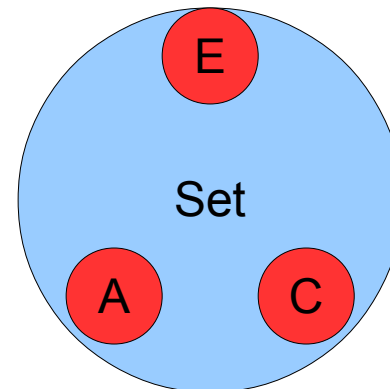
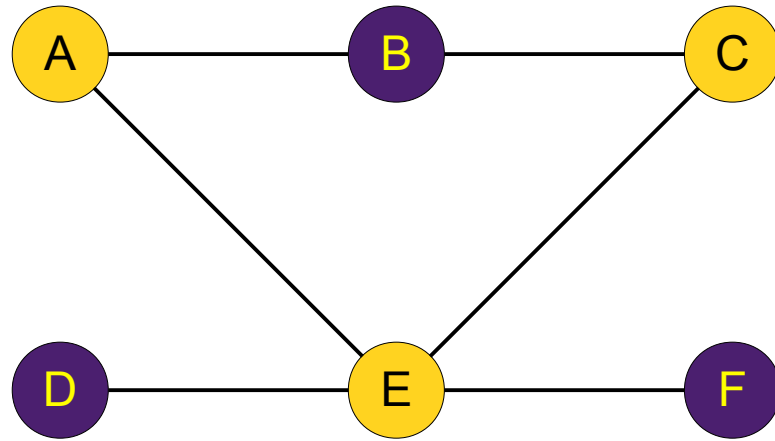
# Depth-first search



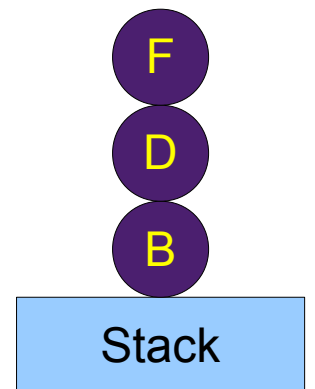
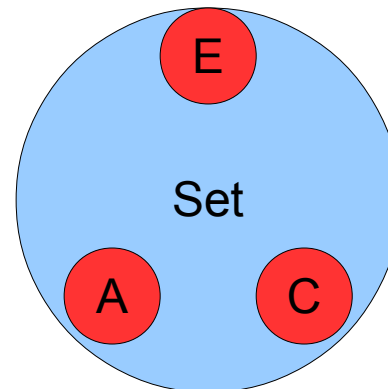
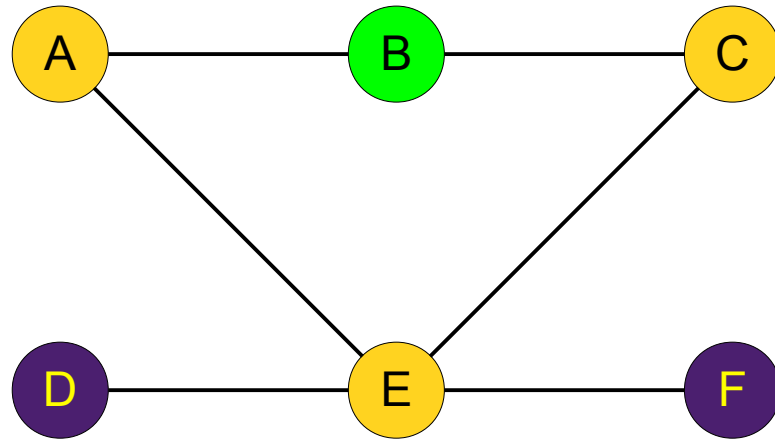
# Depth-first search



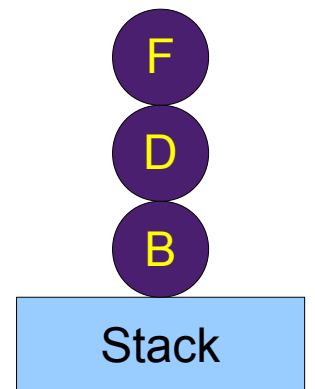
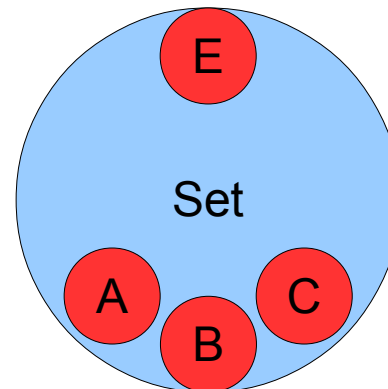
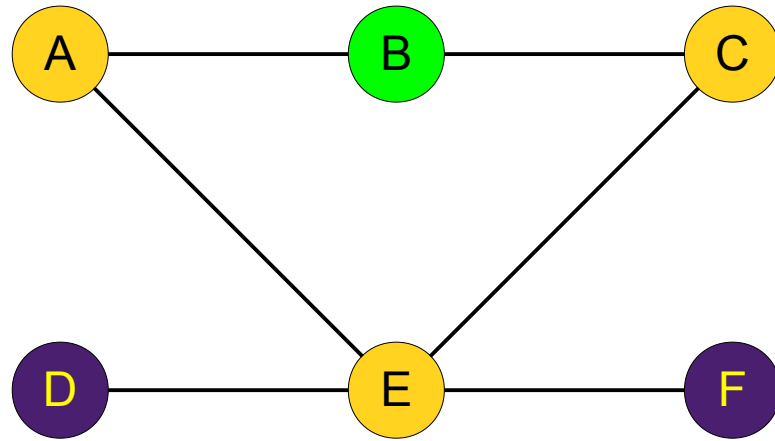
# Depth-first search



# Depth-first search

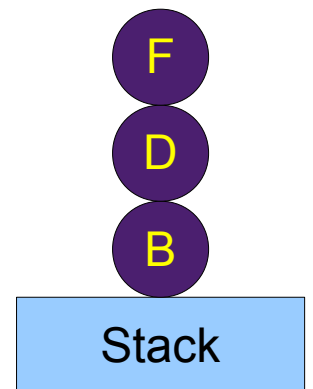
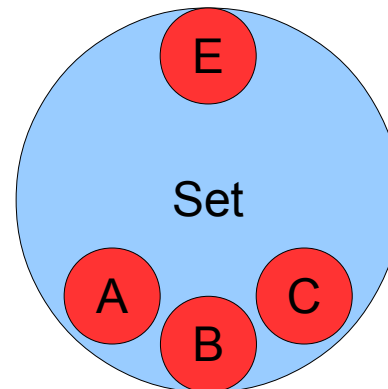
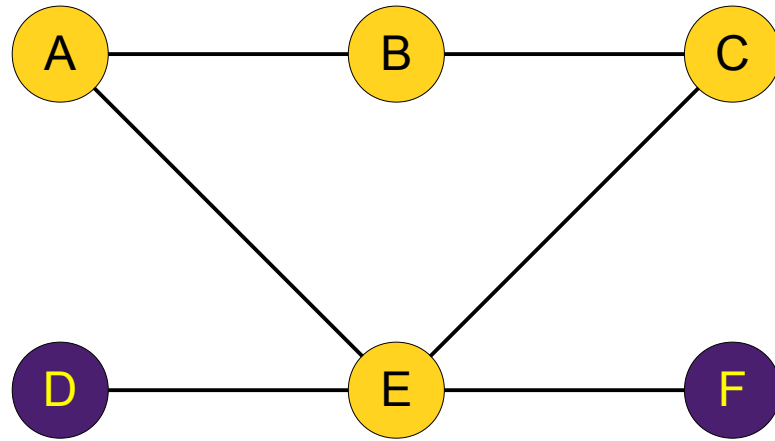


# Depth-first search

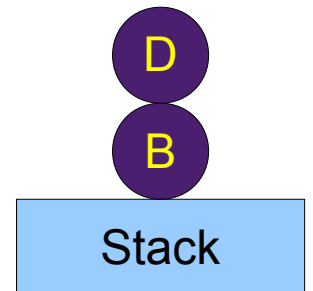
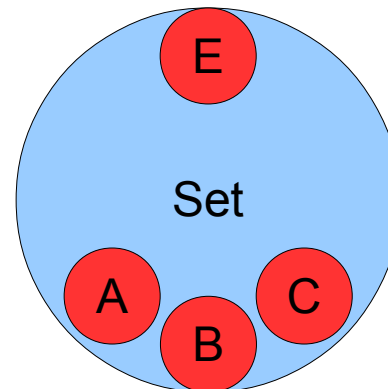
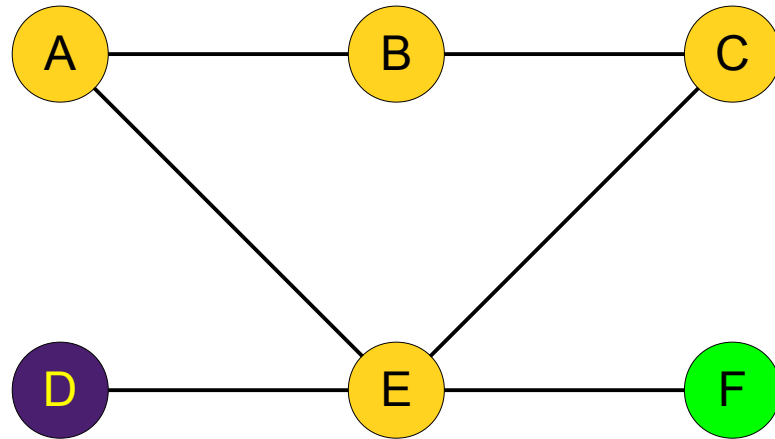




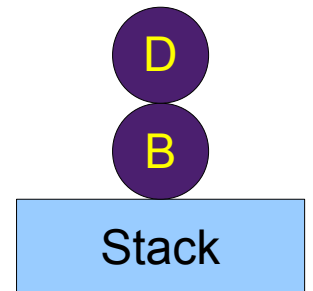
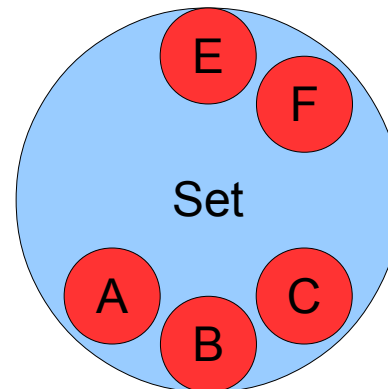
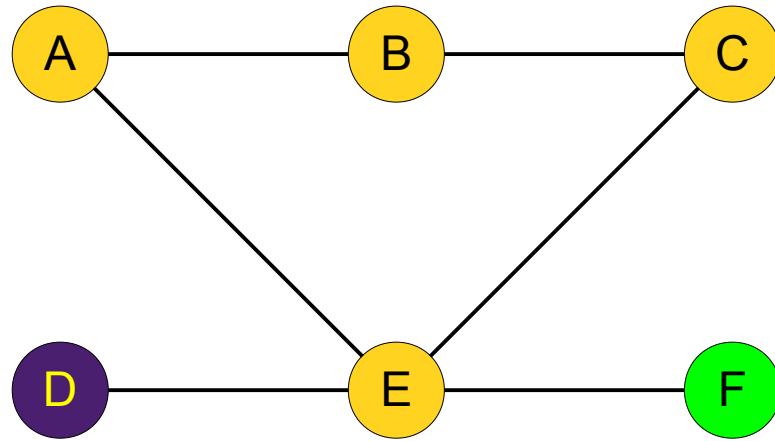
# Depth-first search



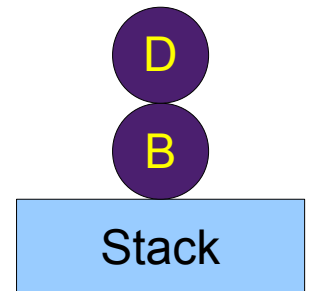
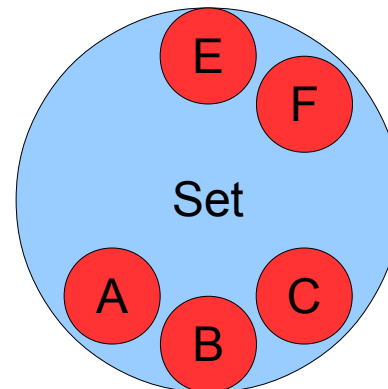
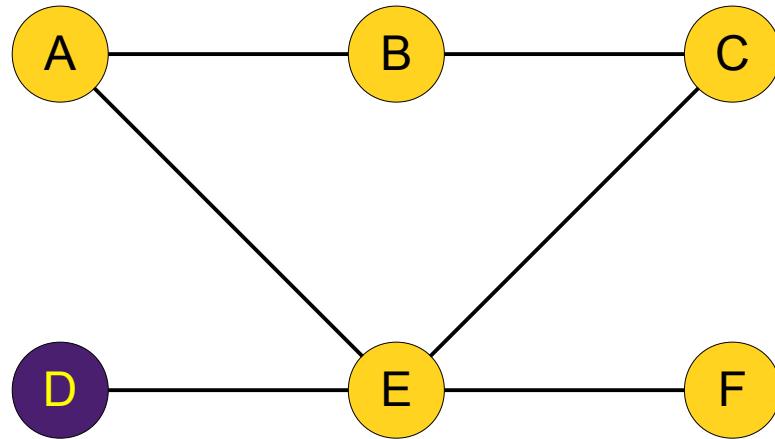
# Depth-first search



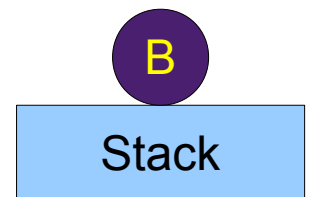
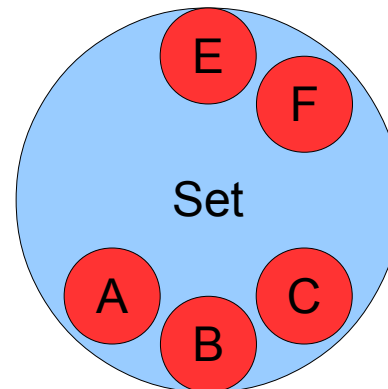
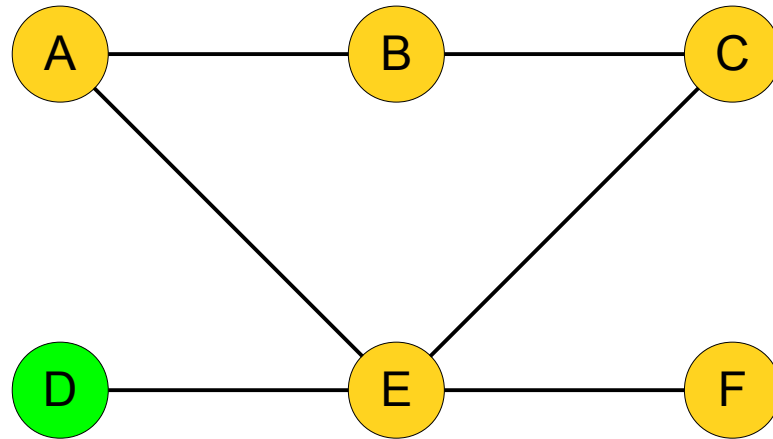
# Depth-first search



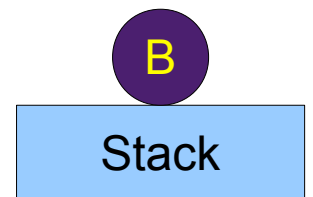
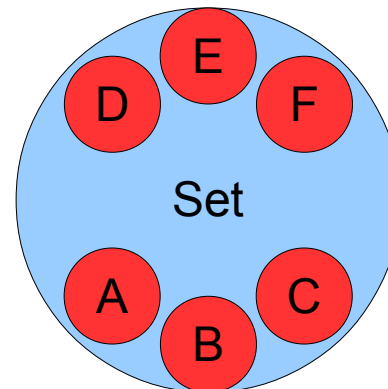
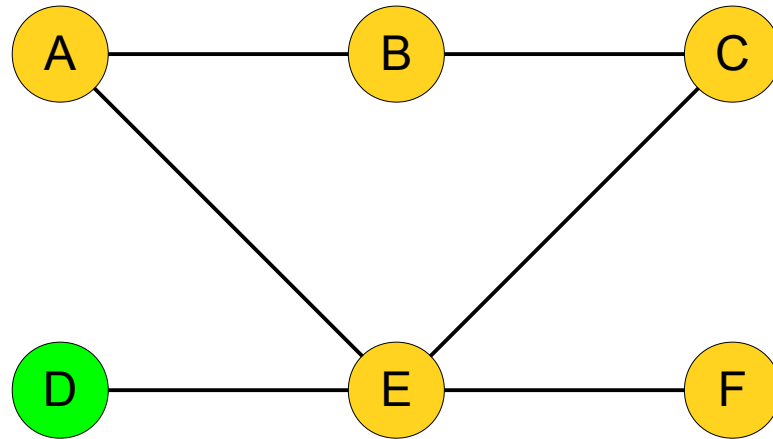
# Depth-first search



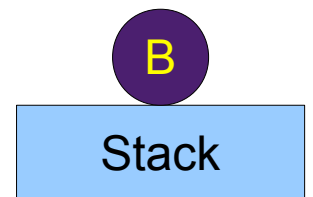
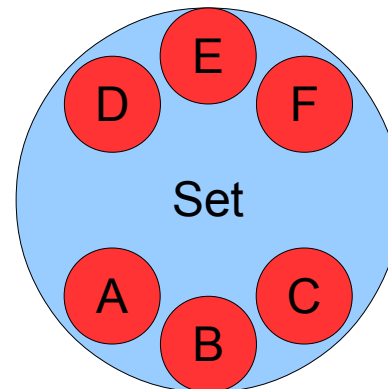
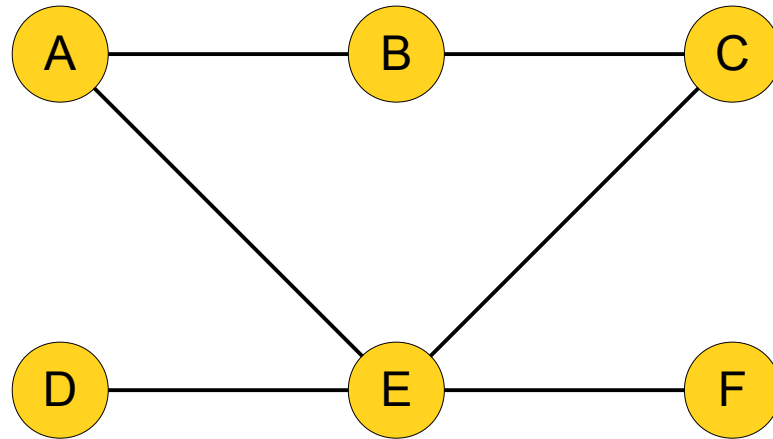
# Depth-first search



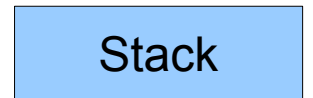
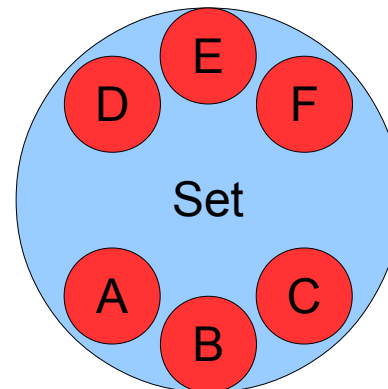
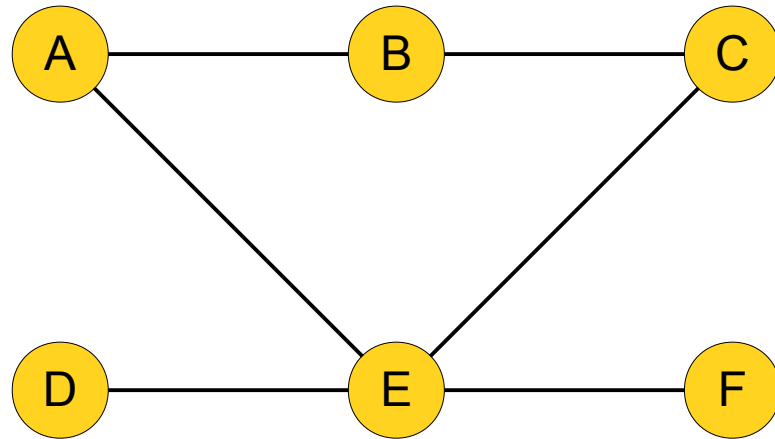
# Depth-first search



# Depth-first search



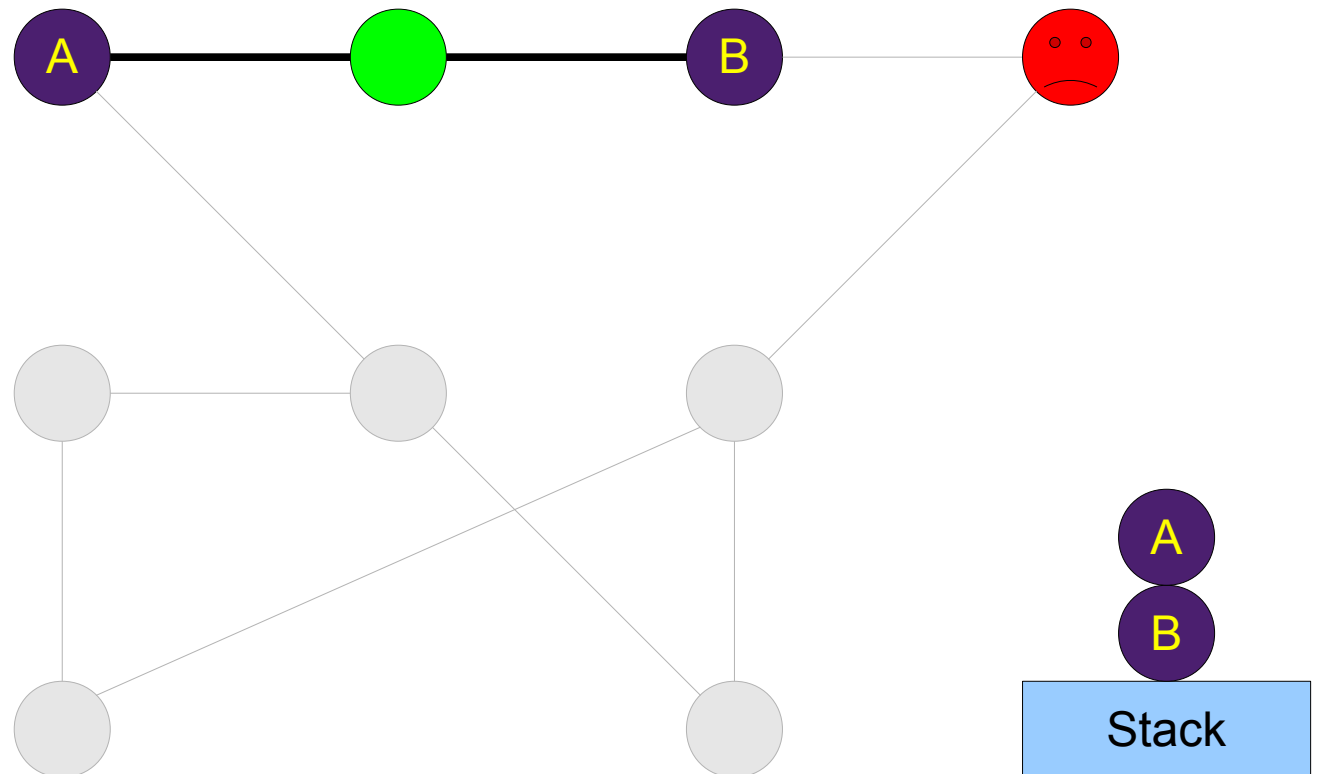
# Depth-first search





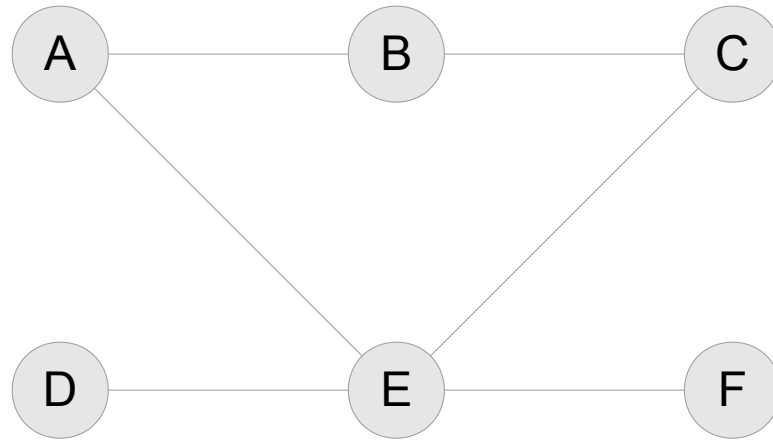
# Problems with DFS

- Useful when trying to explore everything.
- Not good at finding shortest paths

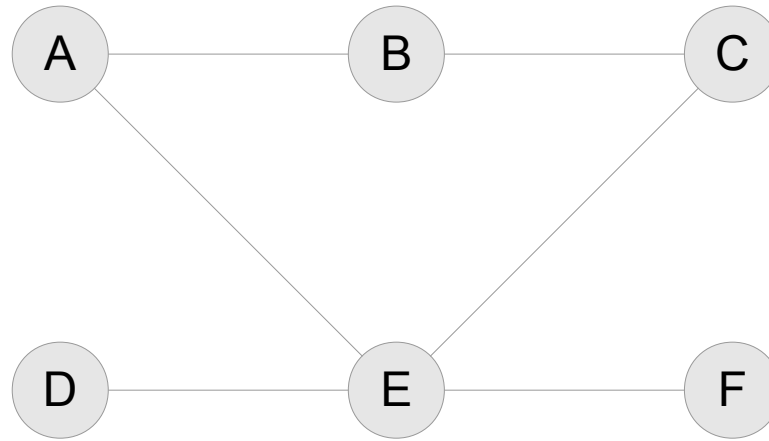


# Breadth-First Search

# Breadth-first search

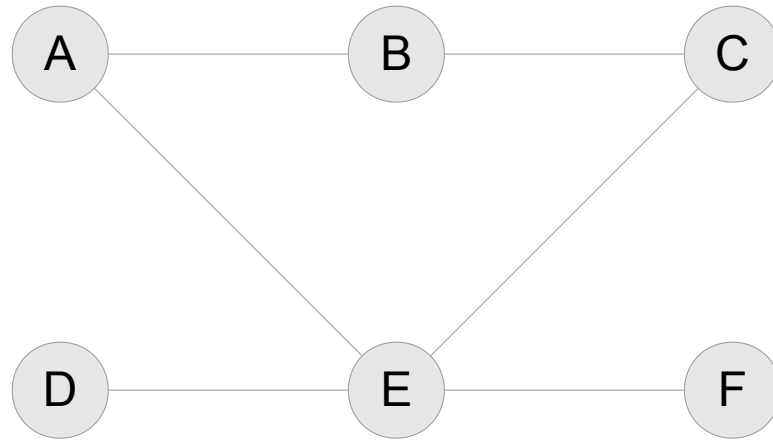


# Breadth-first search



Queue

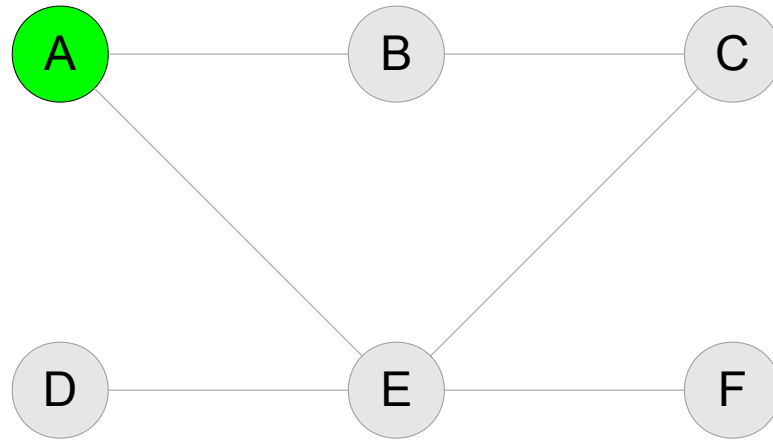
# Breadth-first search



Queue

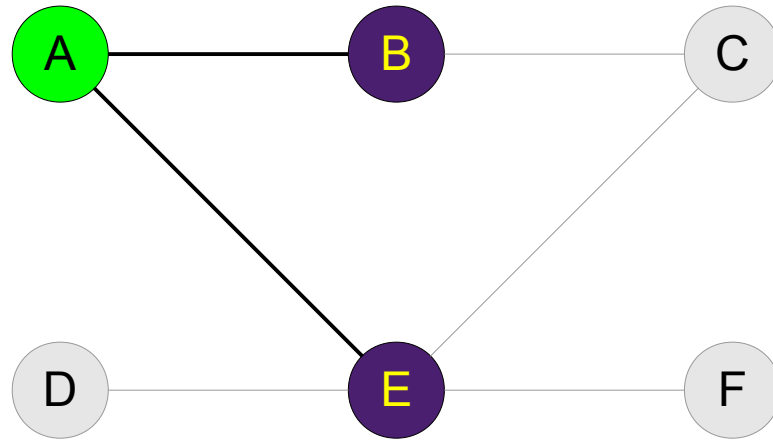
A

# Breadth-first search

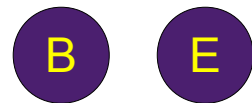


Queue

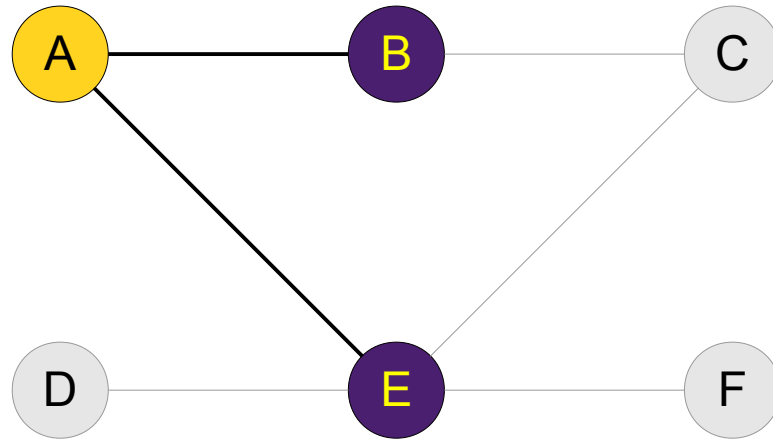
# Breadth-first search



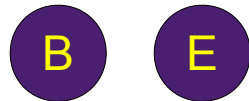
Queue



# Breadth-first search

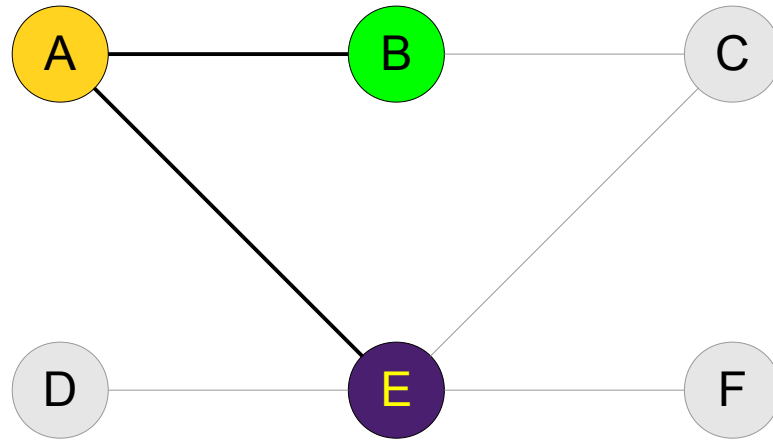


Queue





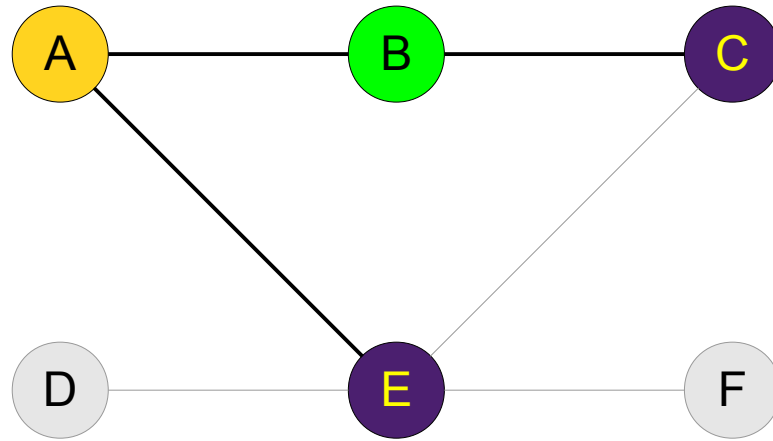
# Breadth-first search



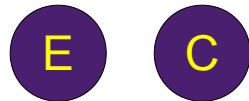
Queue



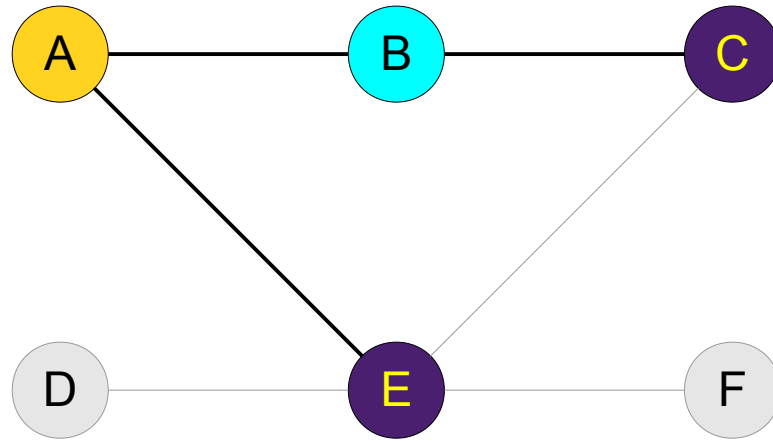
# Breadth-first search



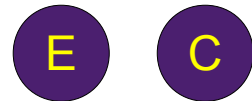
Queue



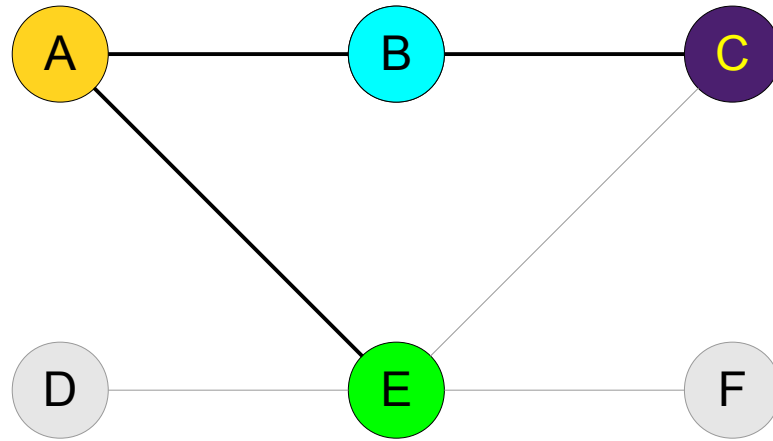
# Breadth-first search



Queue



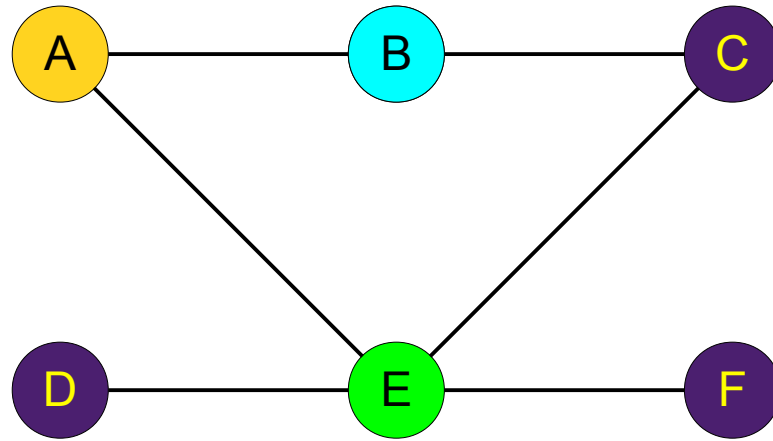
# Breadth-first search



Queue

C

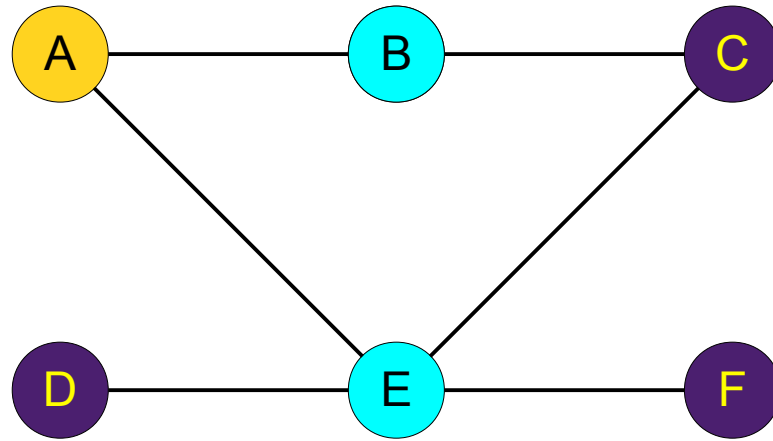
# Breadth-first search



Queue



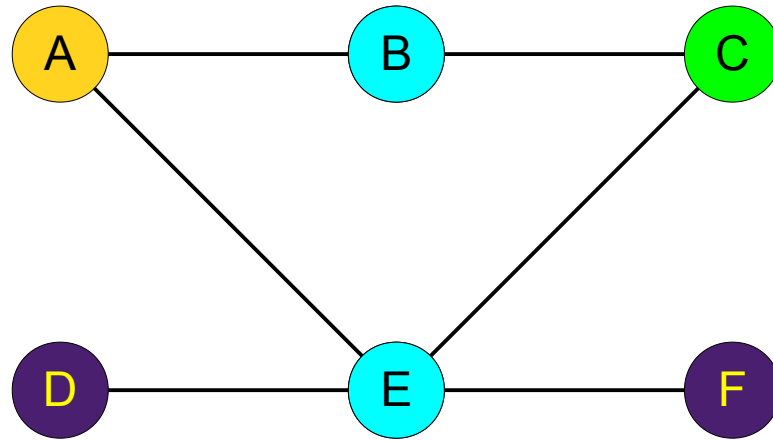
# Breadth-first search



Queue



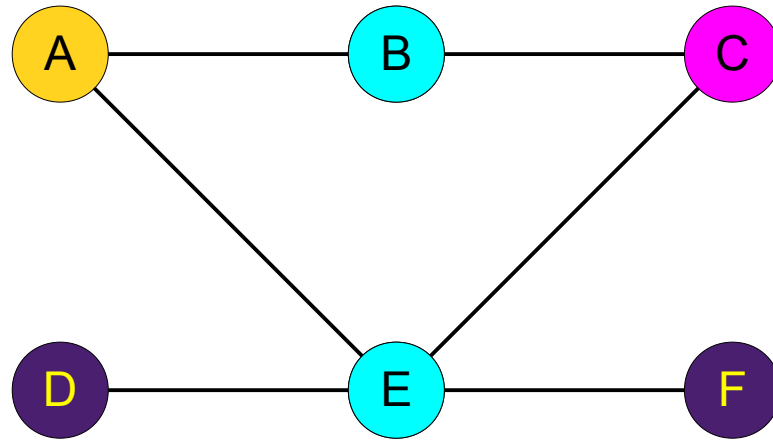
# Breadth-first search



Queue



# Breadth-first search

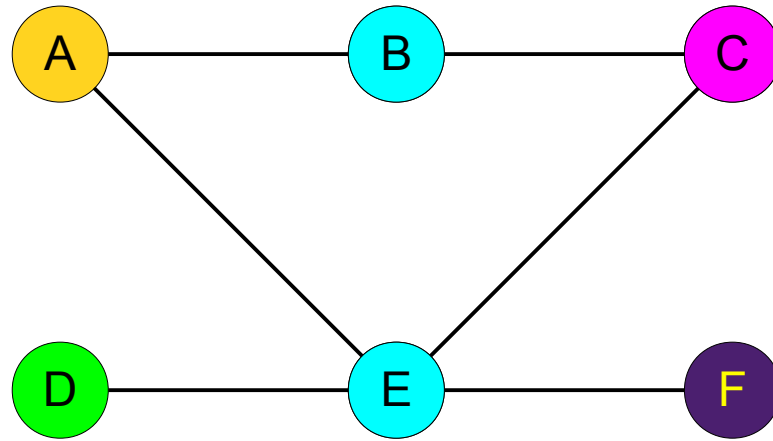


Queue





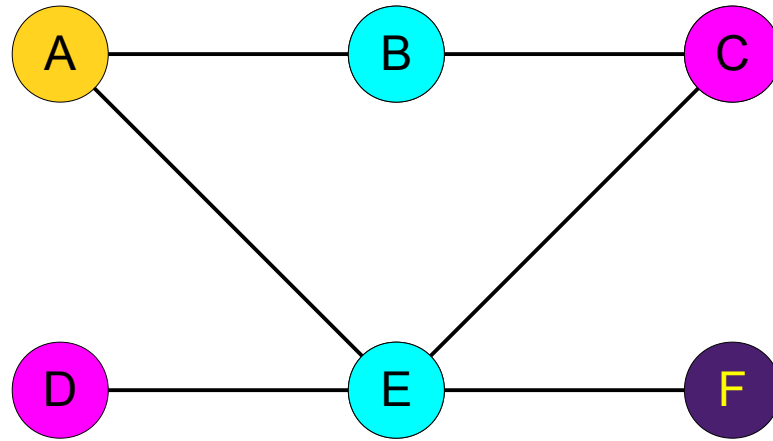
# Breadth-first search



Queue

F

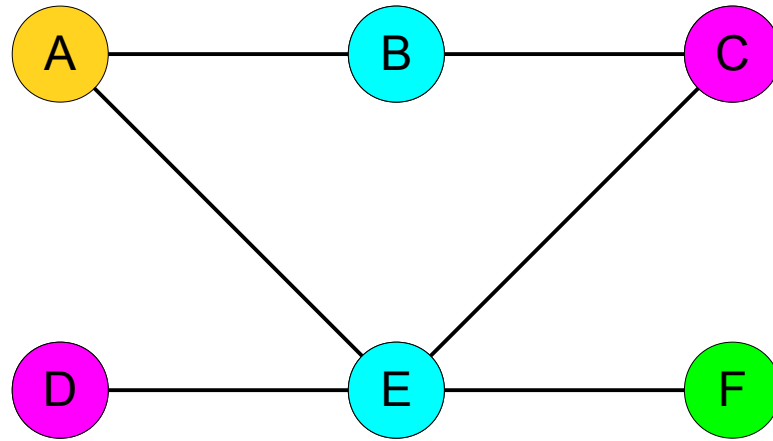
# Breadth-first search



Queue

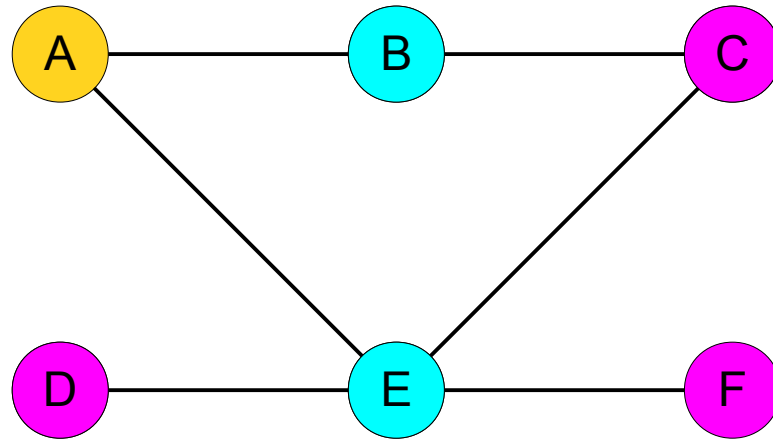
F

# Breadth-first search



Queue

# Breadth-first search



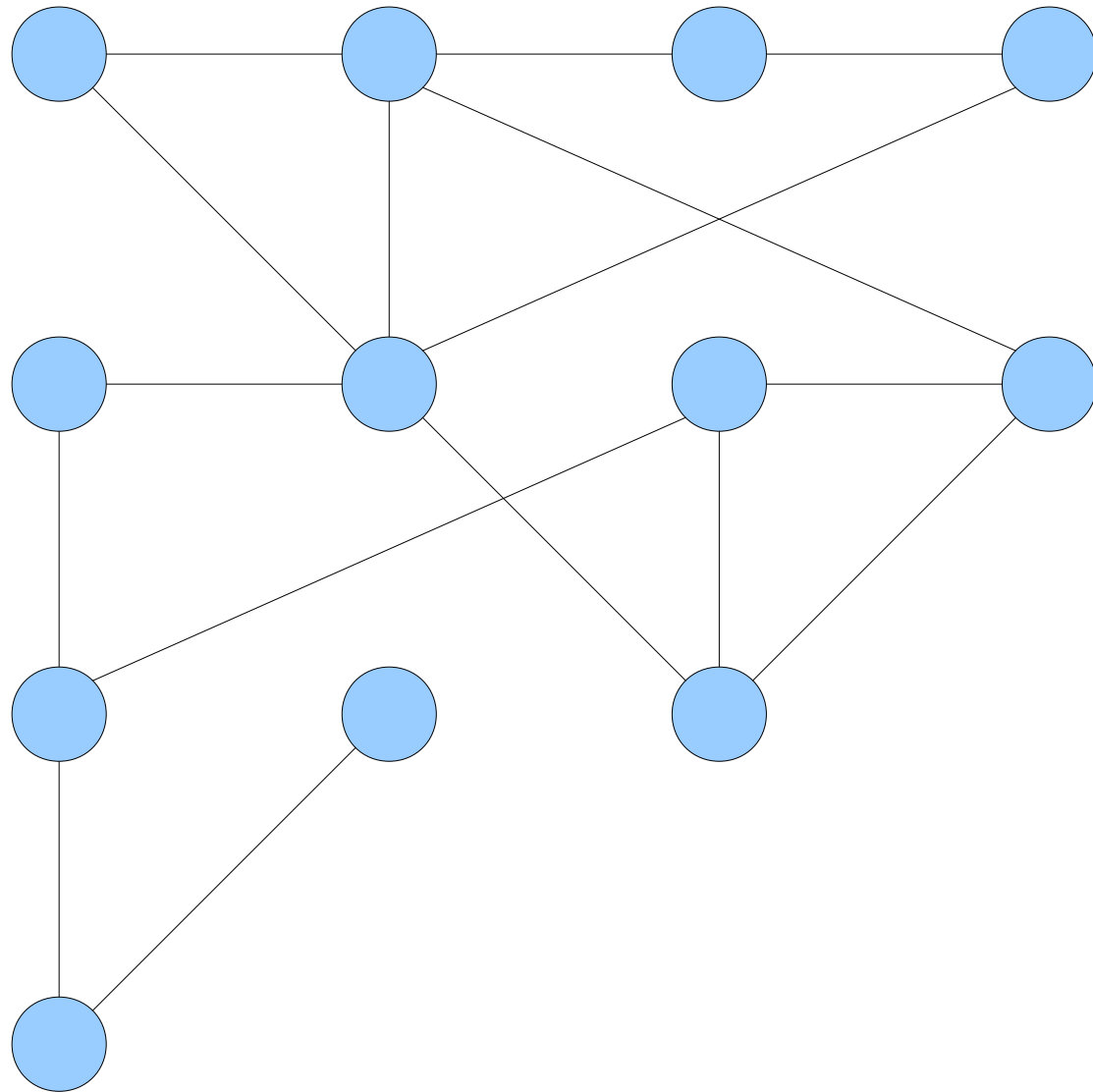
Queue

# Problem with BFS

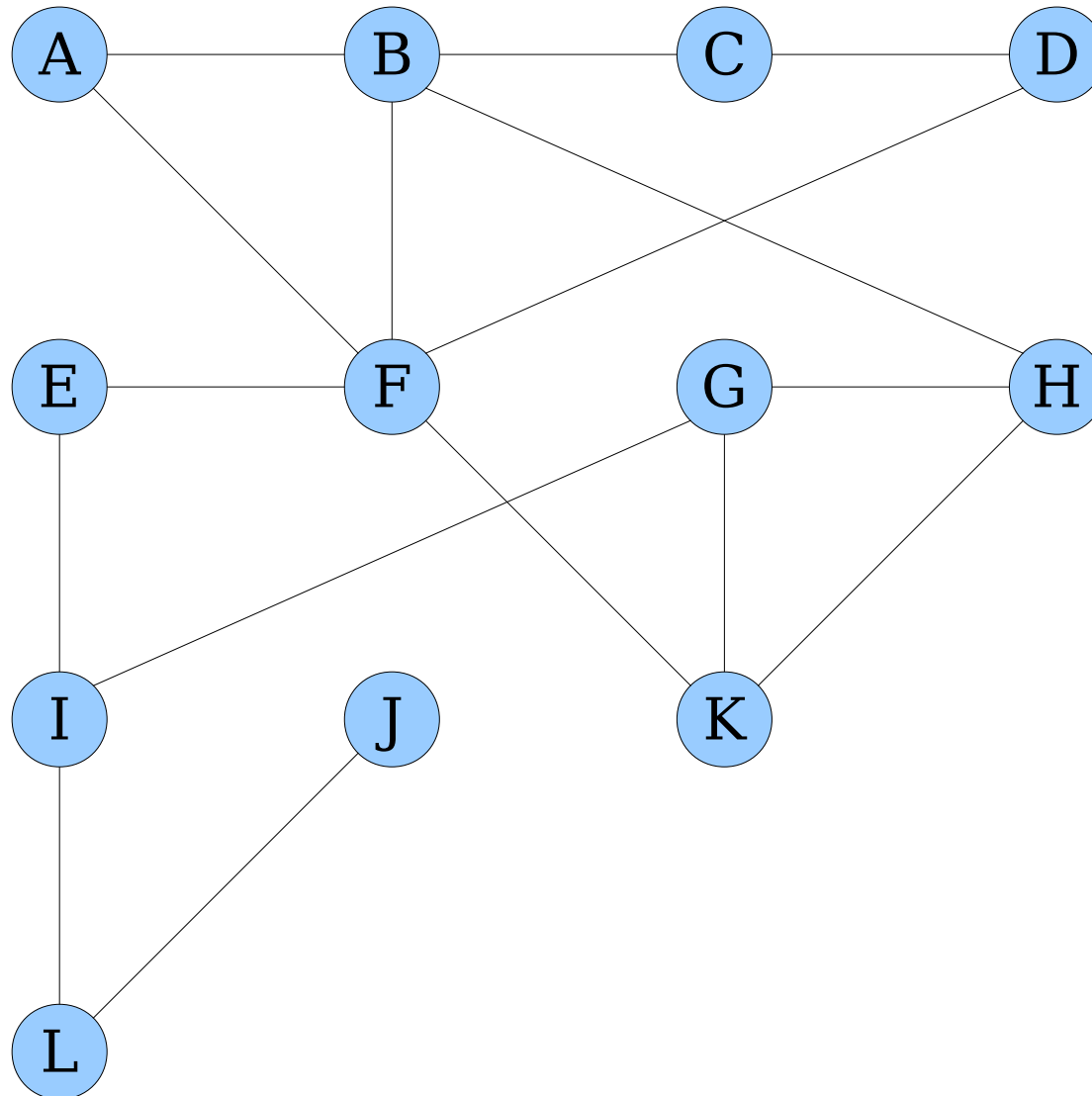
- In the version of BFS we just implemented, we don't keep track of the paths we generate!
- There are a couple ways to fix this, but I'm going to show you the coolest =)

# Shortest Path Tree

# Breadth-First Search

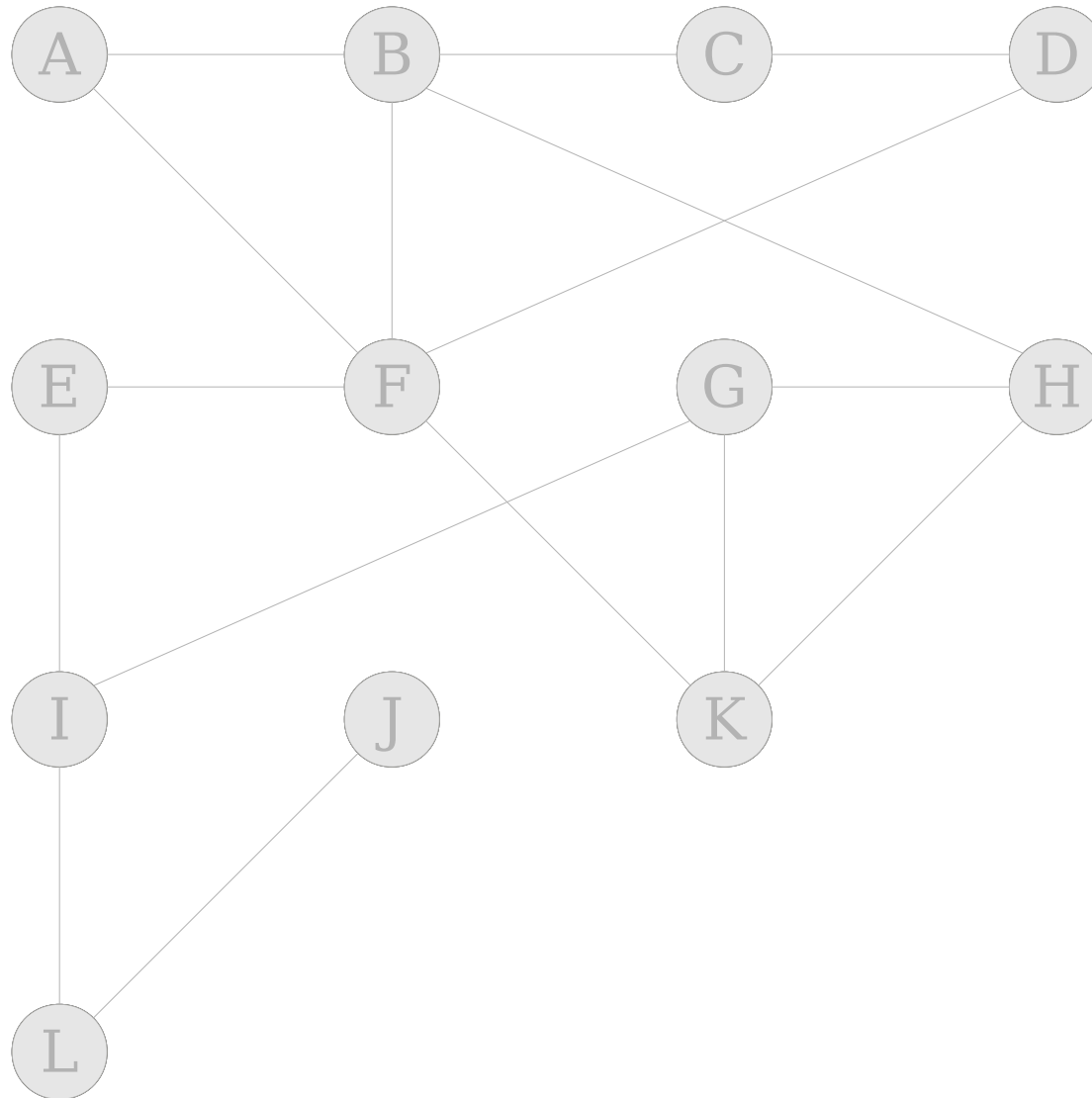


# Breadth-First Search

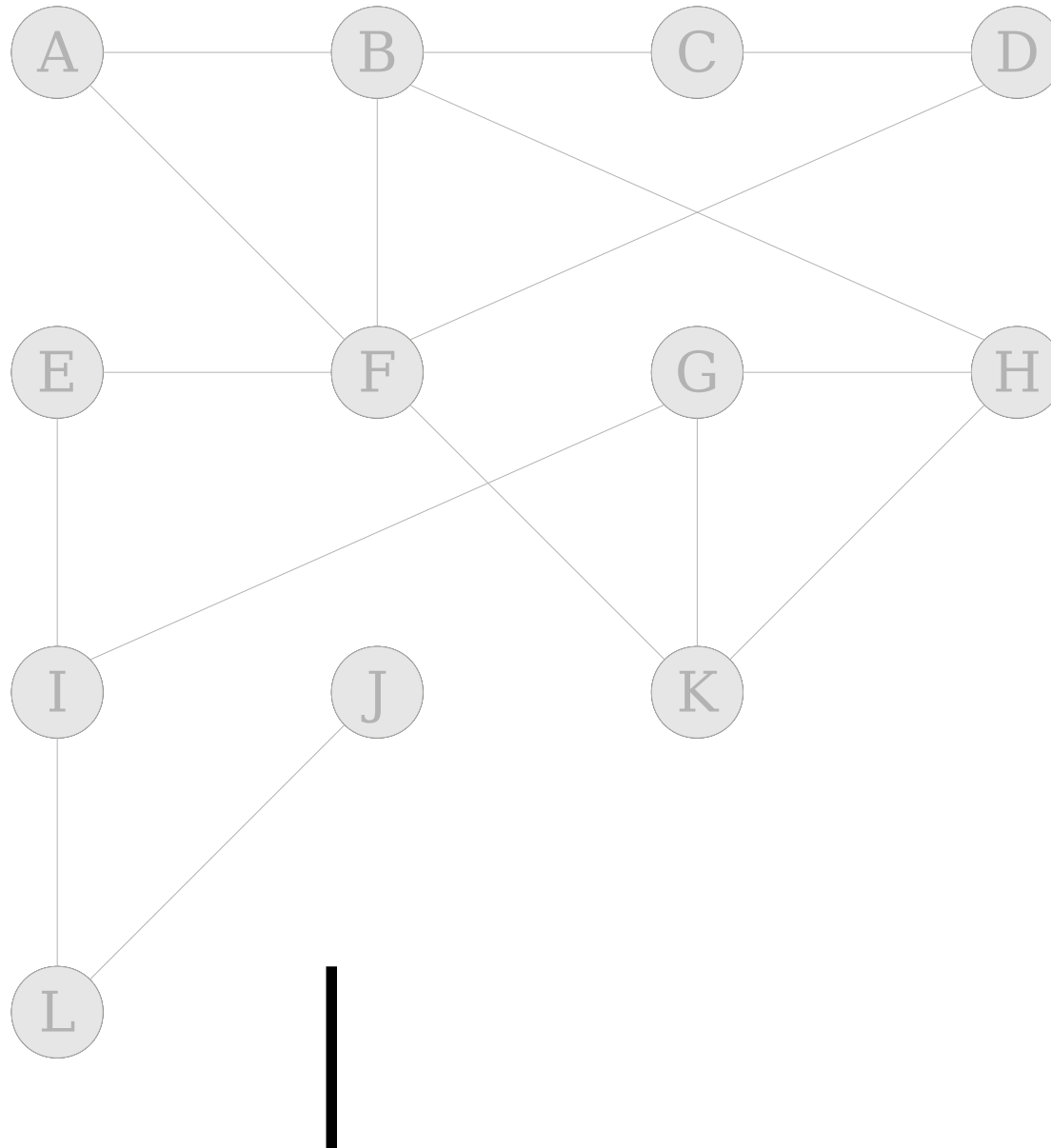




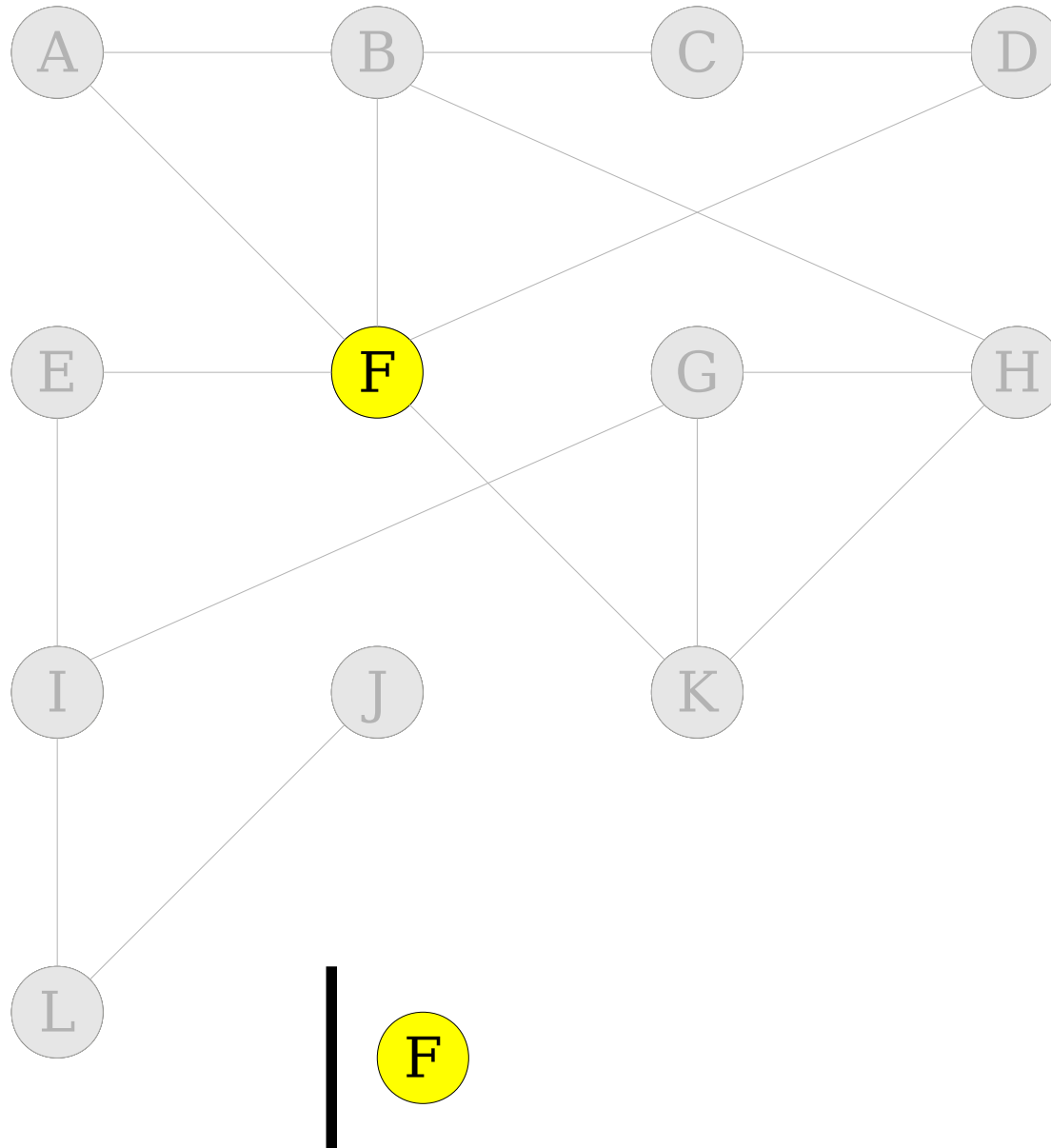
# Breadth-First Search



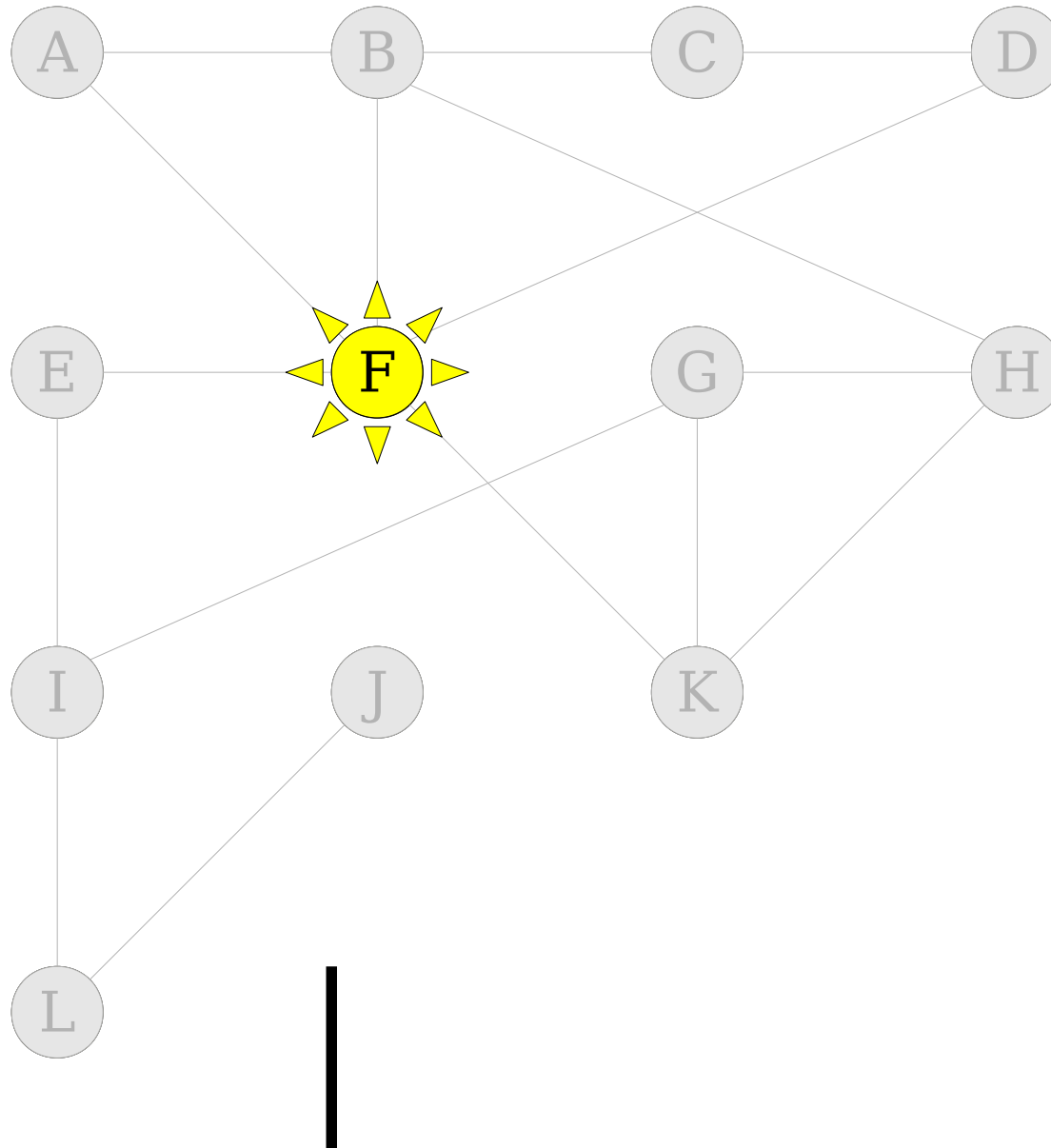
# Breadth-First Search



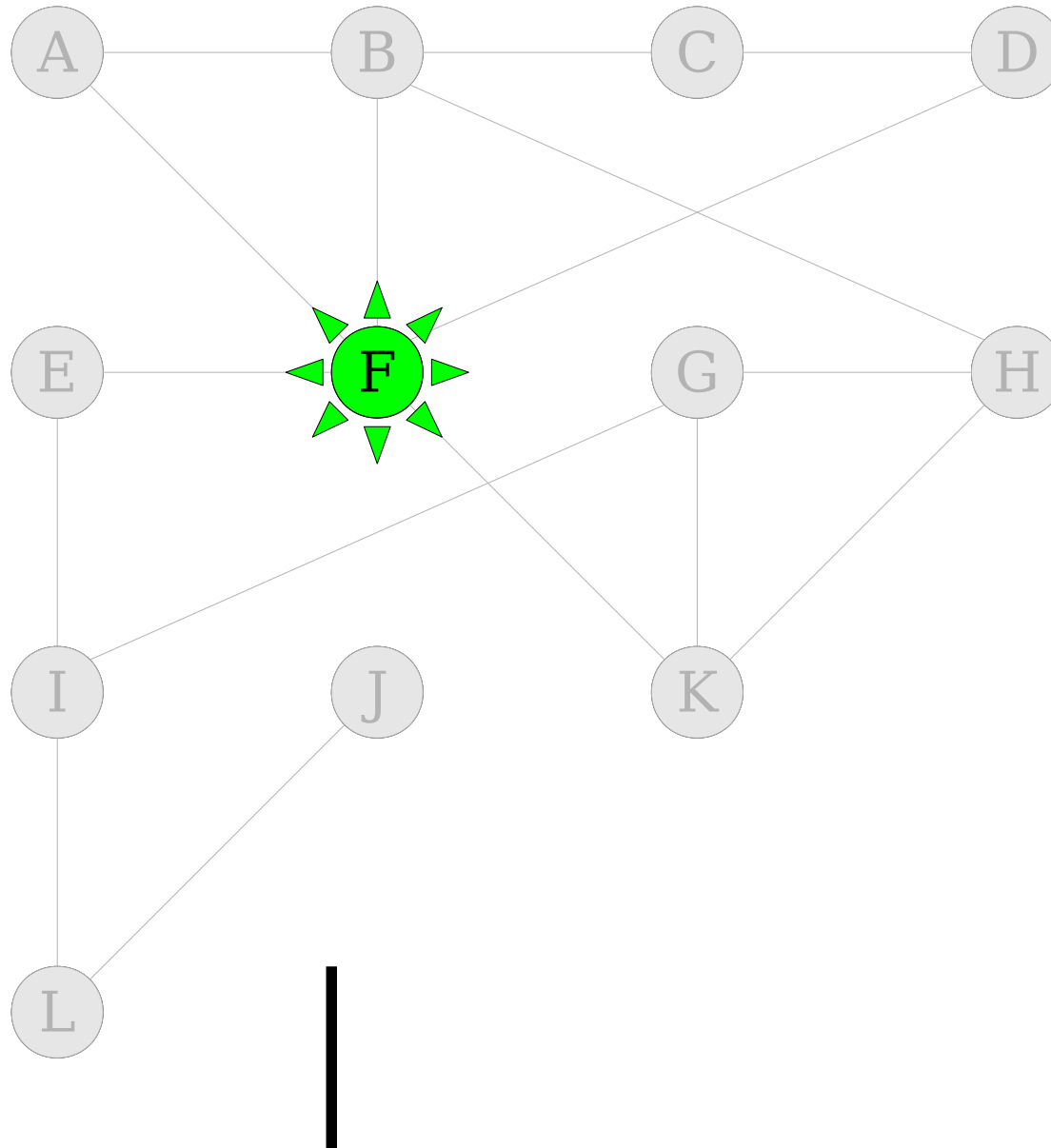
# Breadth-First Search



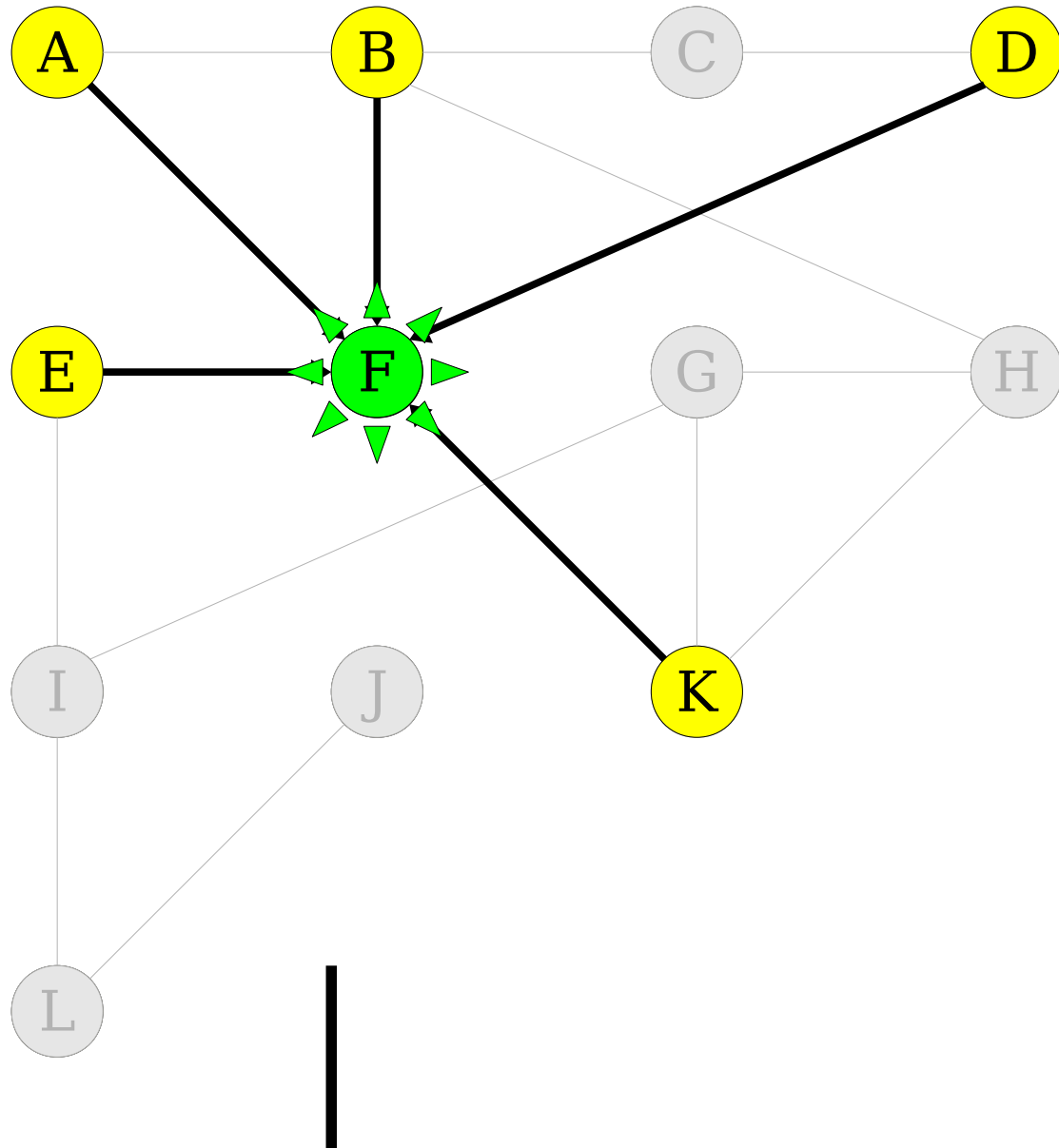
# Breadth-First Search



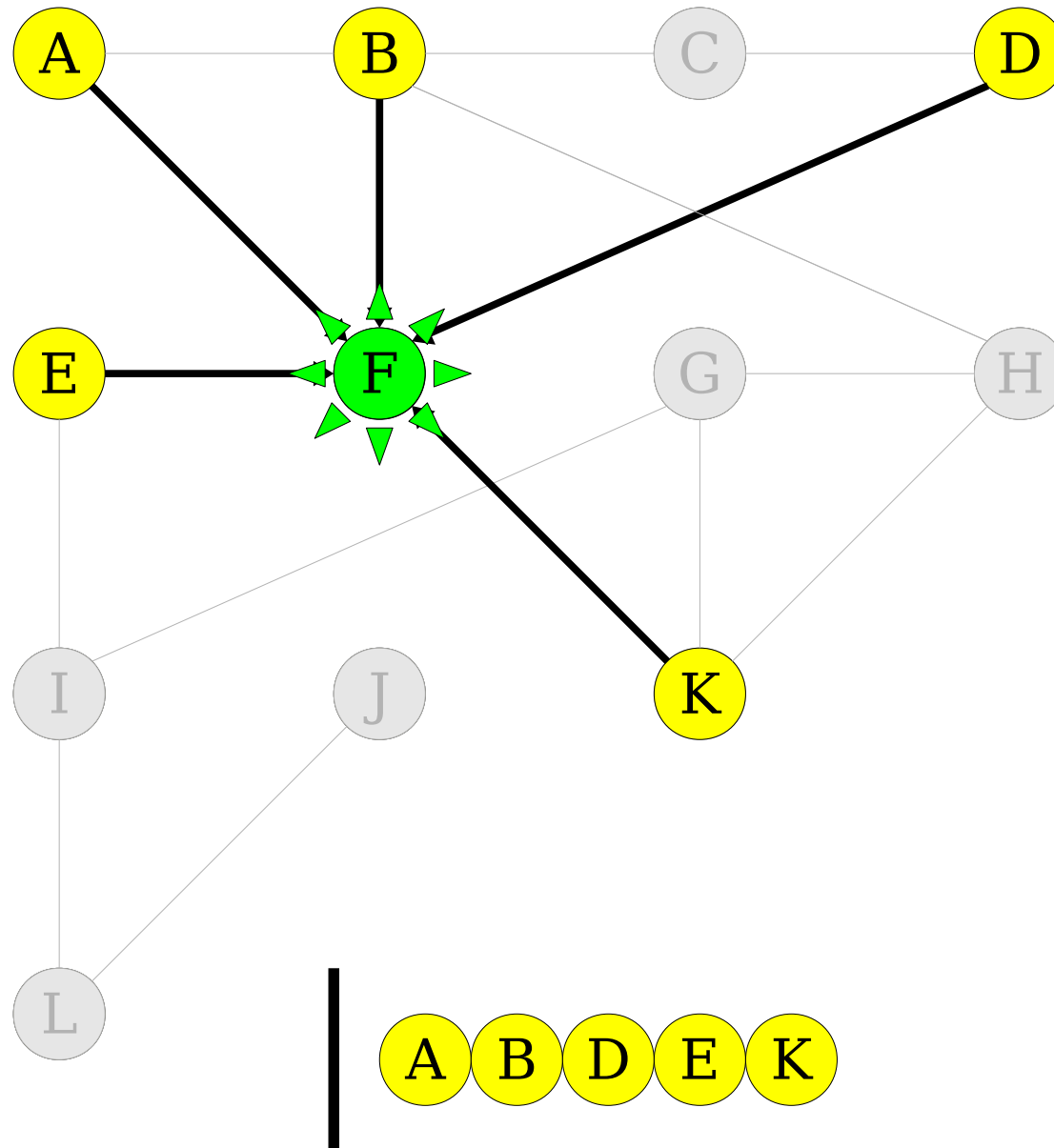
# Breadth-First Search



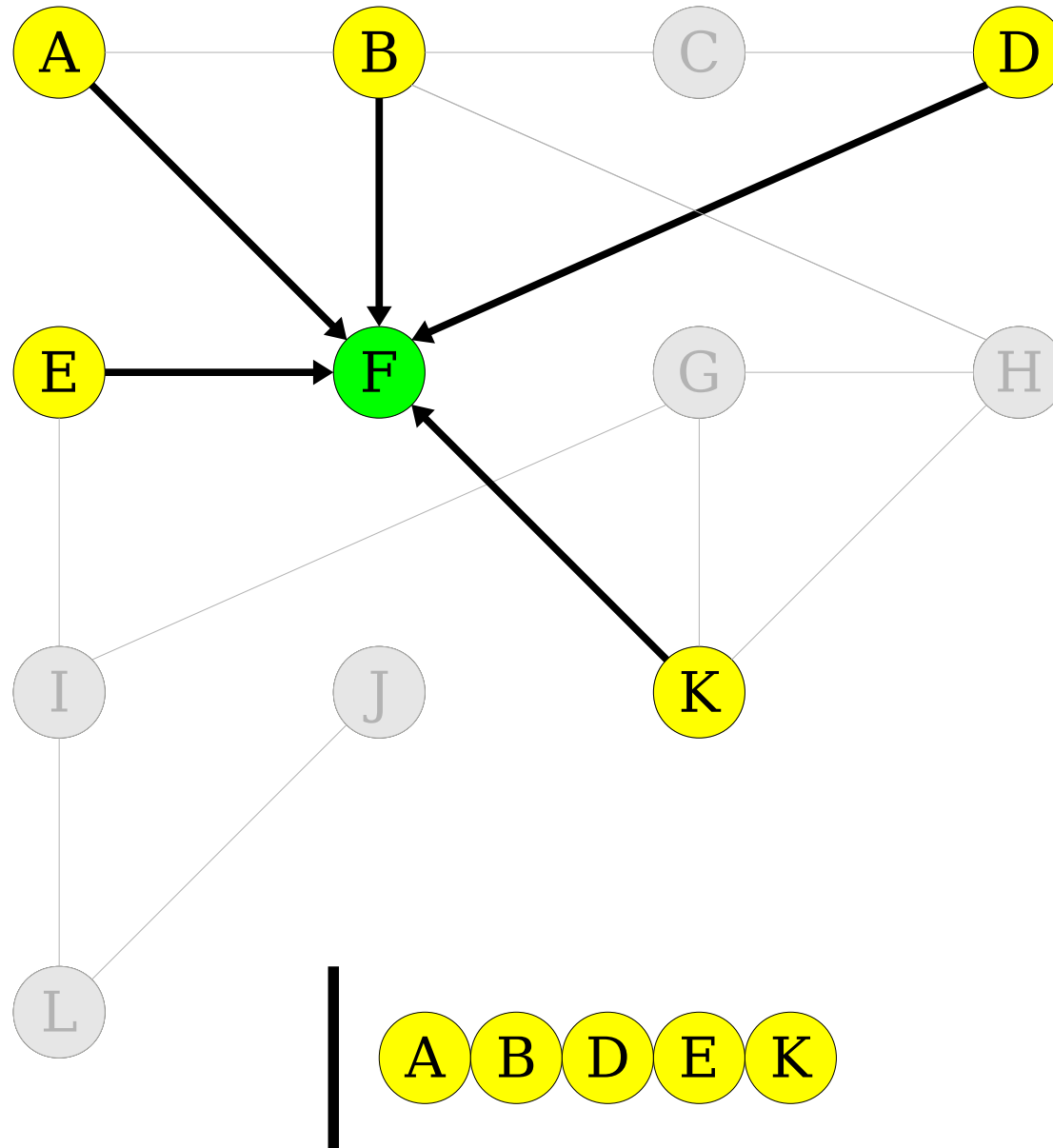
# Breadth-First Search



# Breadth-First Search

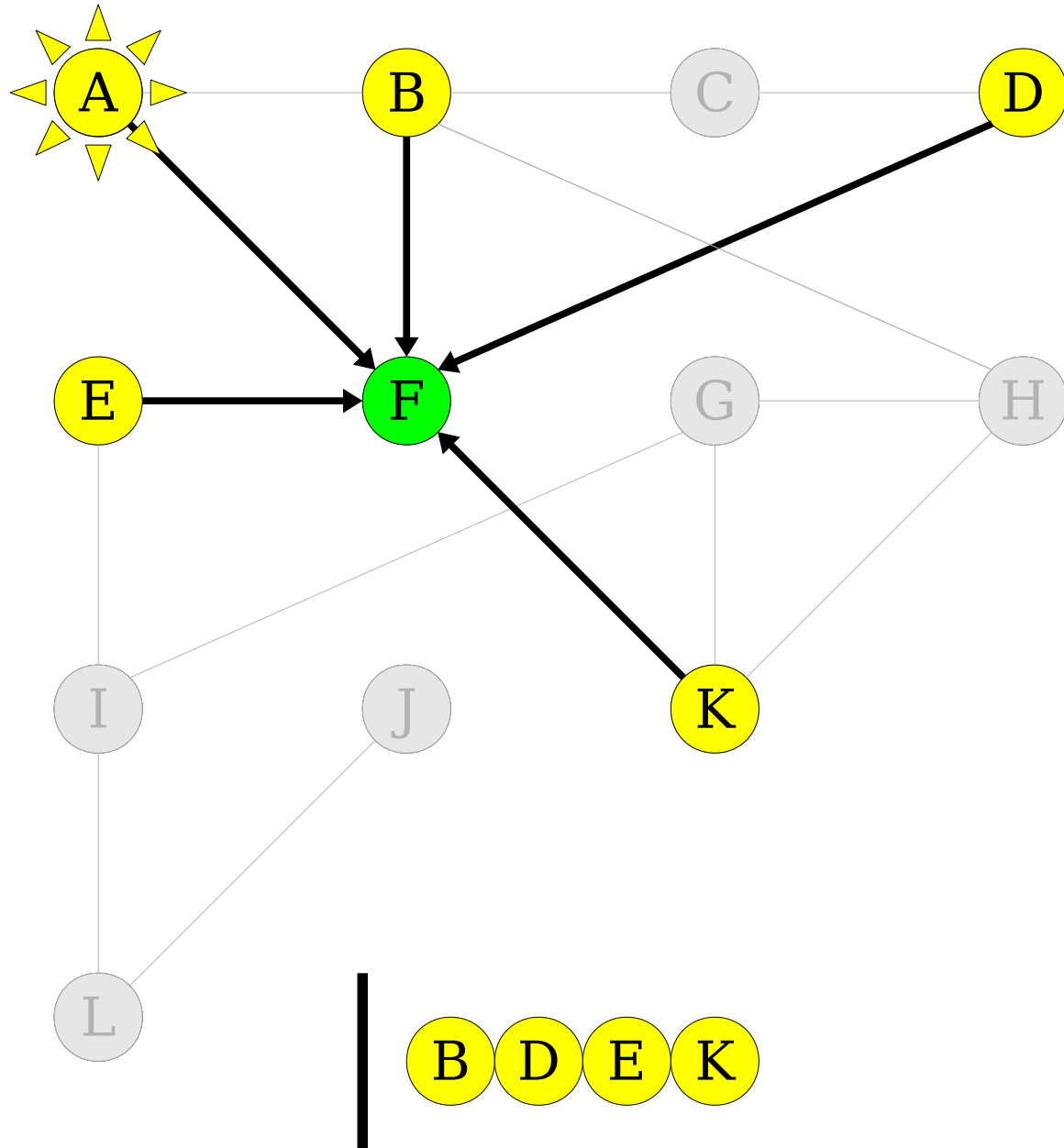


# Breadth-First Search

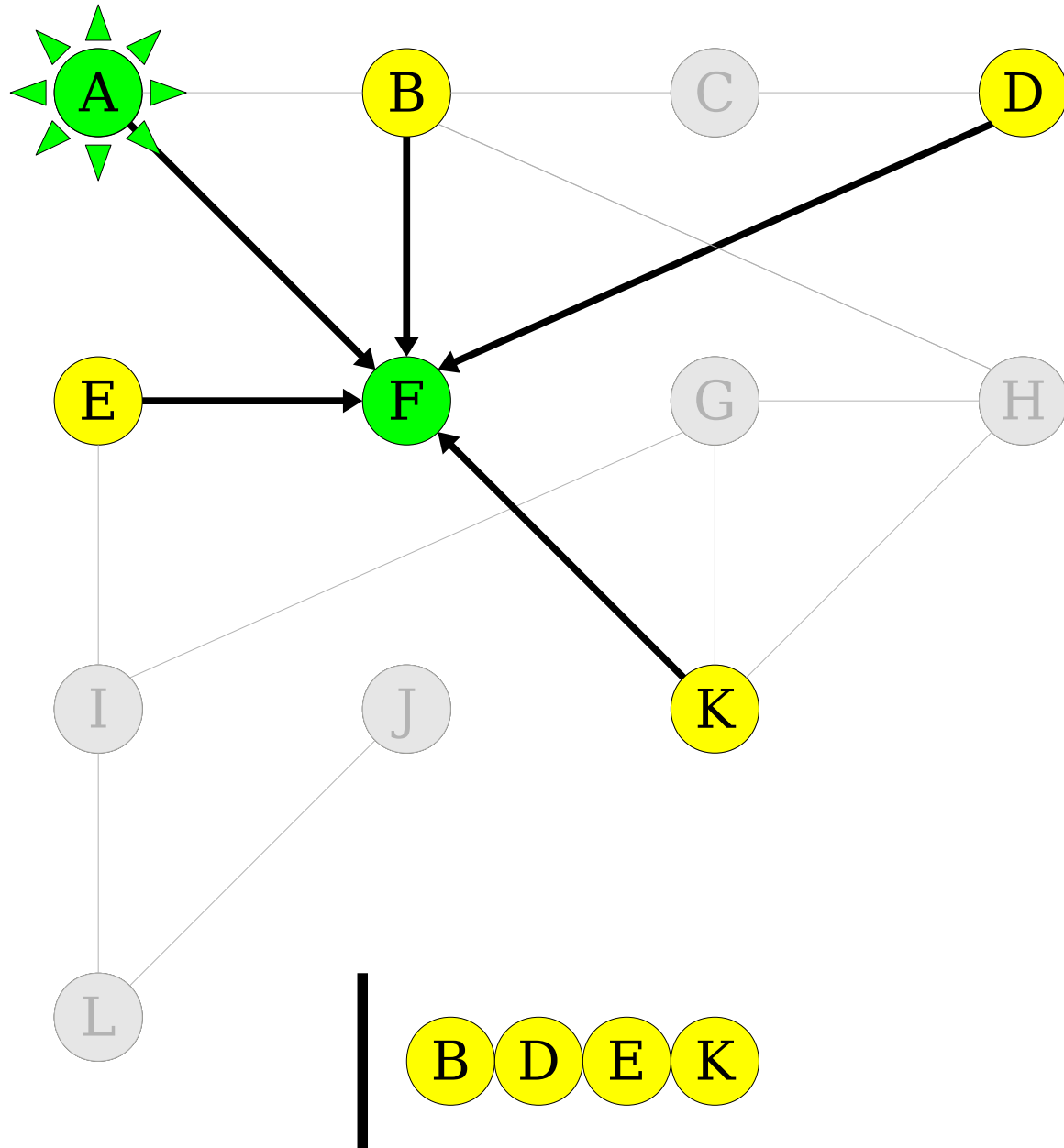




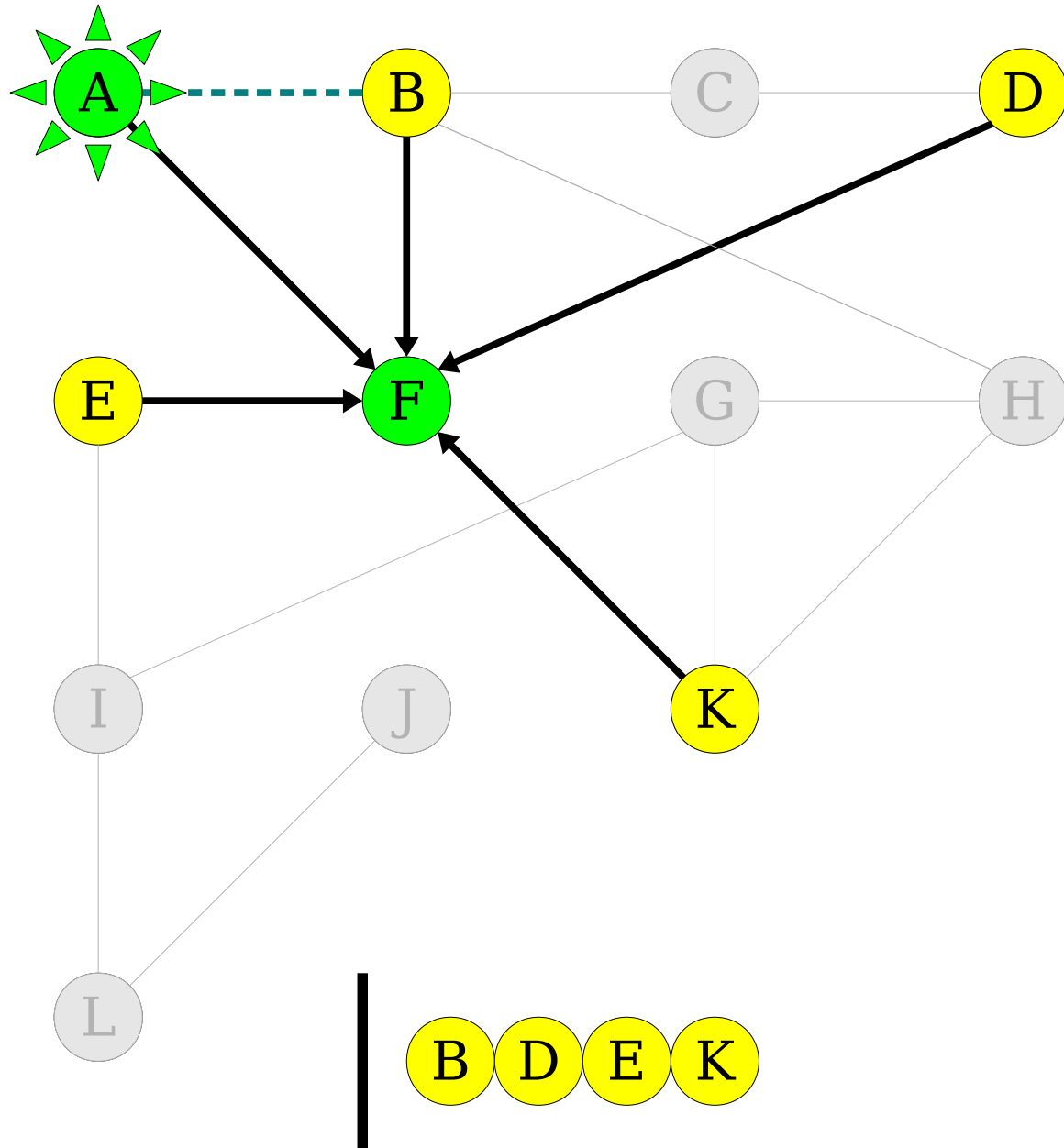
# Breadth-First Search



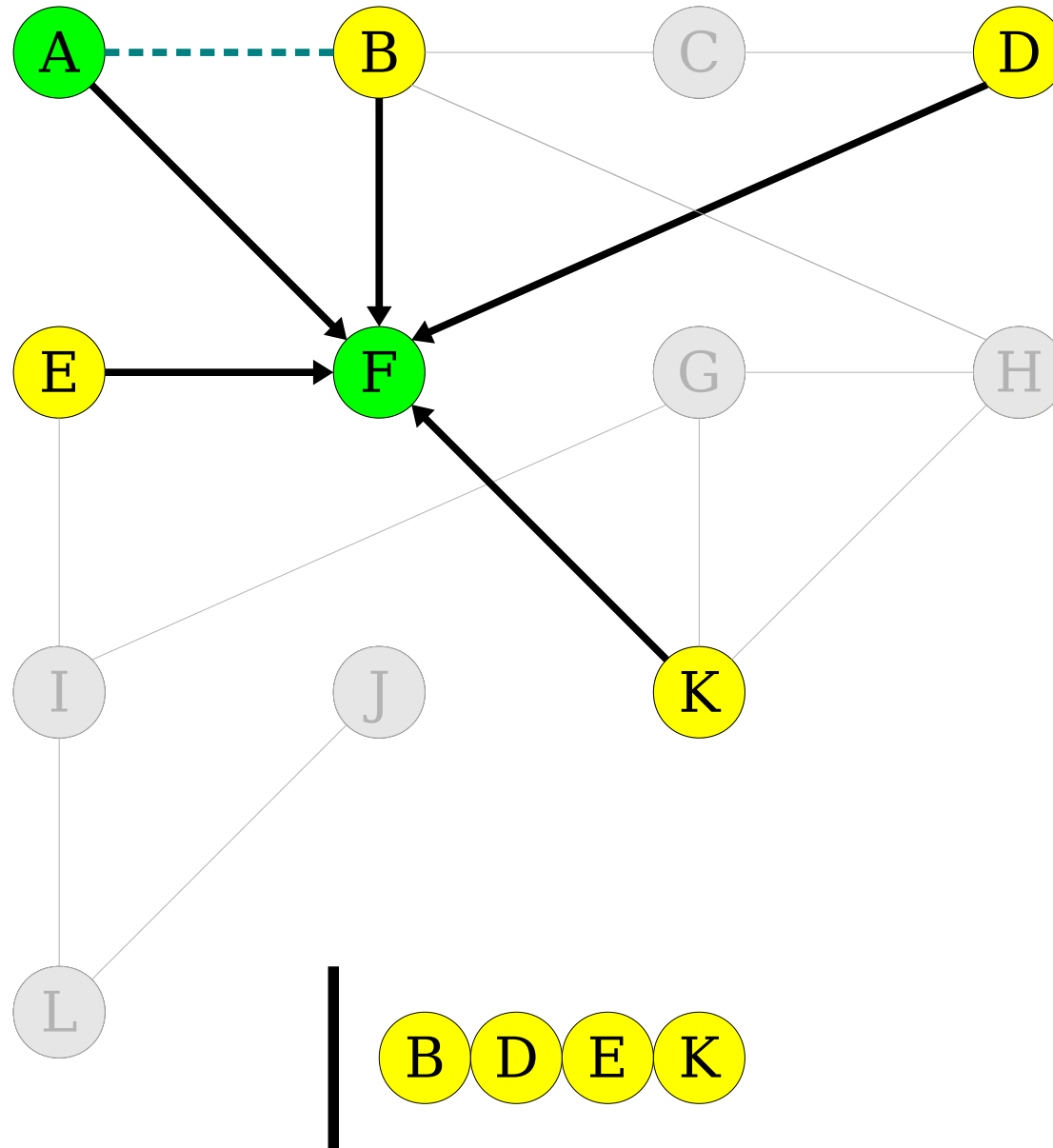
# Breadth-First Search



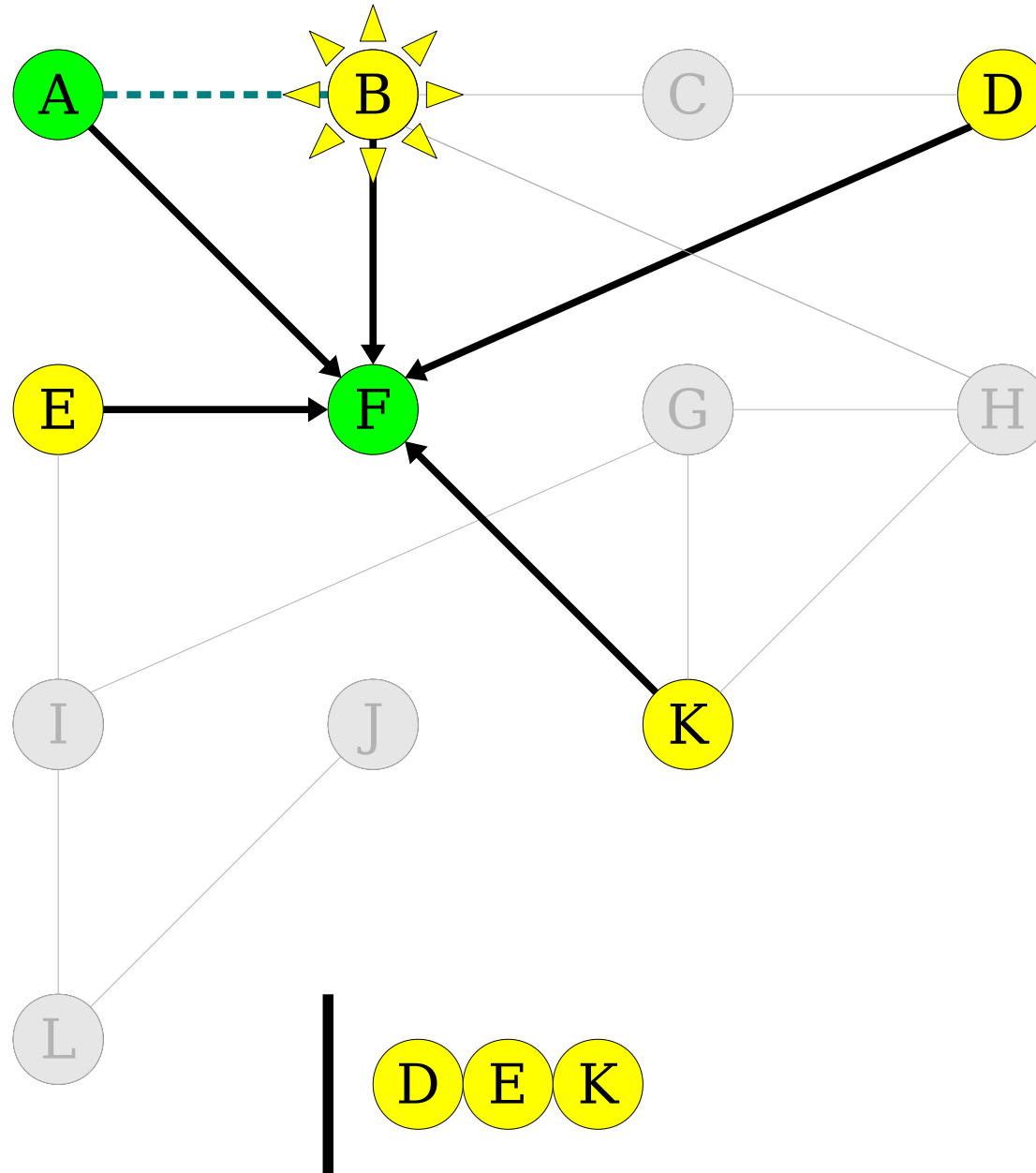
# Breadth-First Search



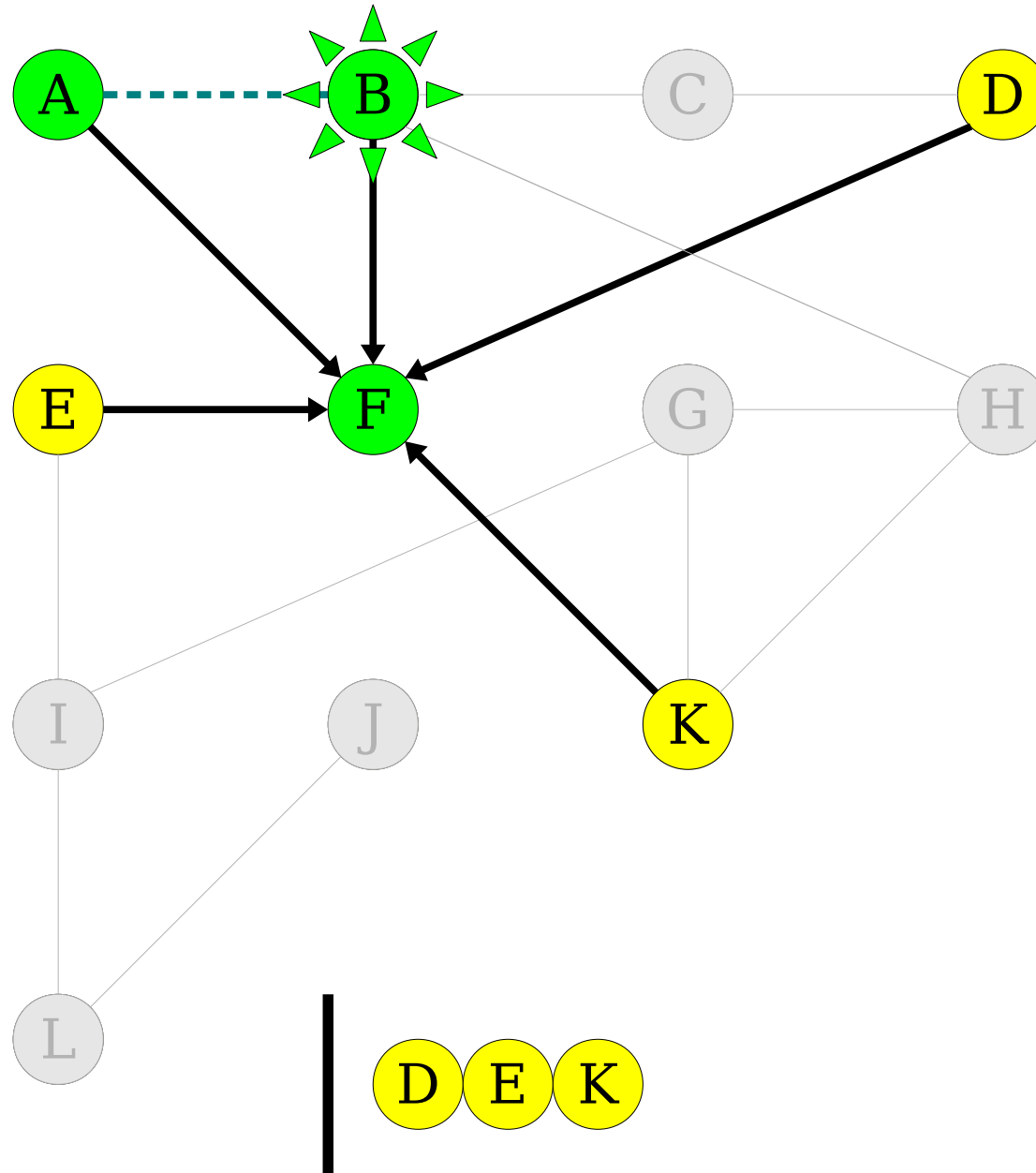
# Breadth-First Search



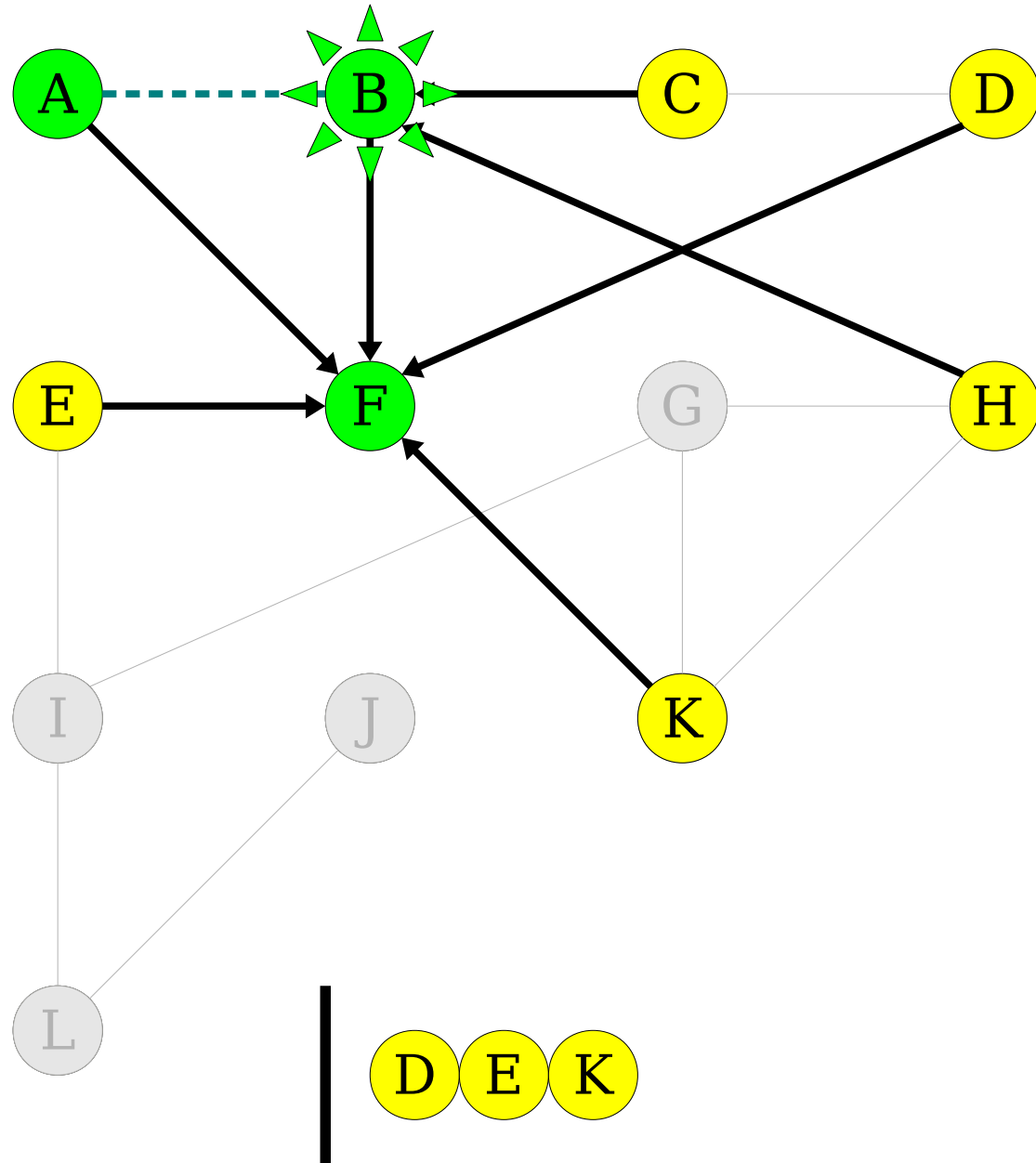
# Breadth-First Search



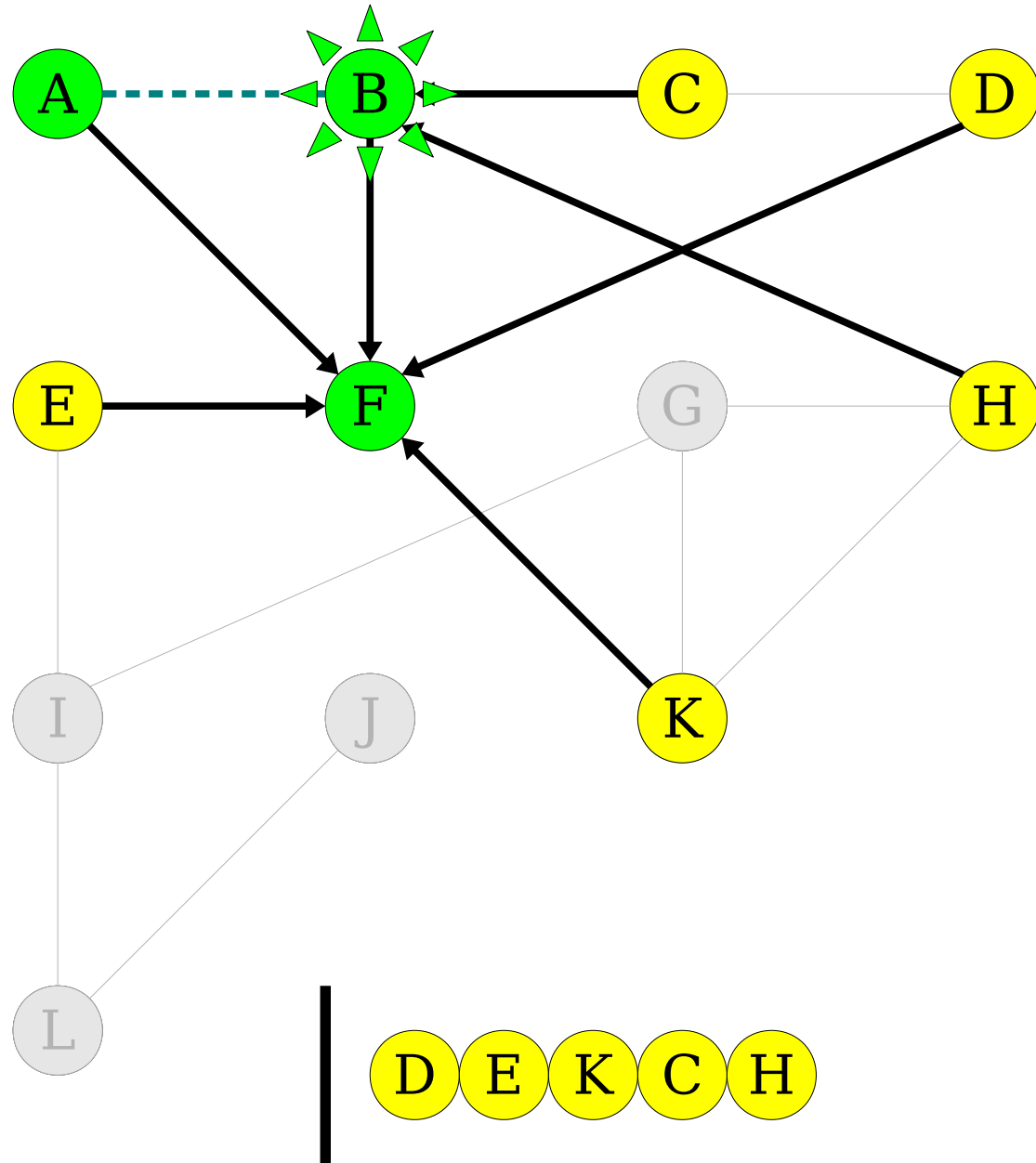
# Breadth-First Search



# Breadth-First Search

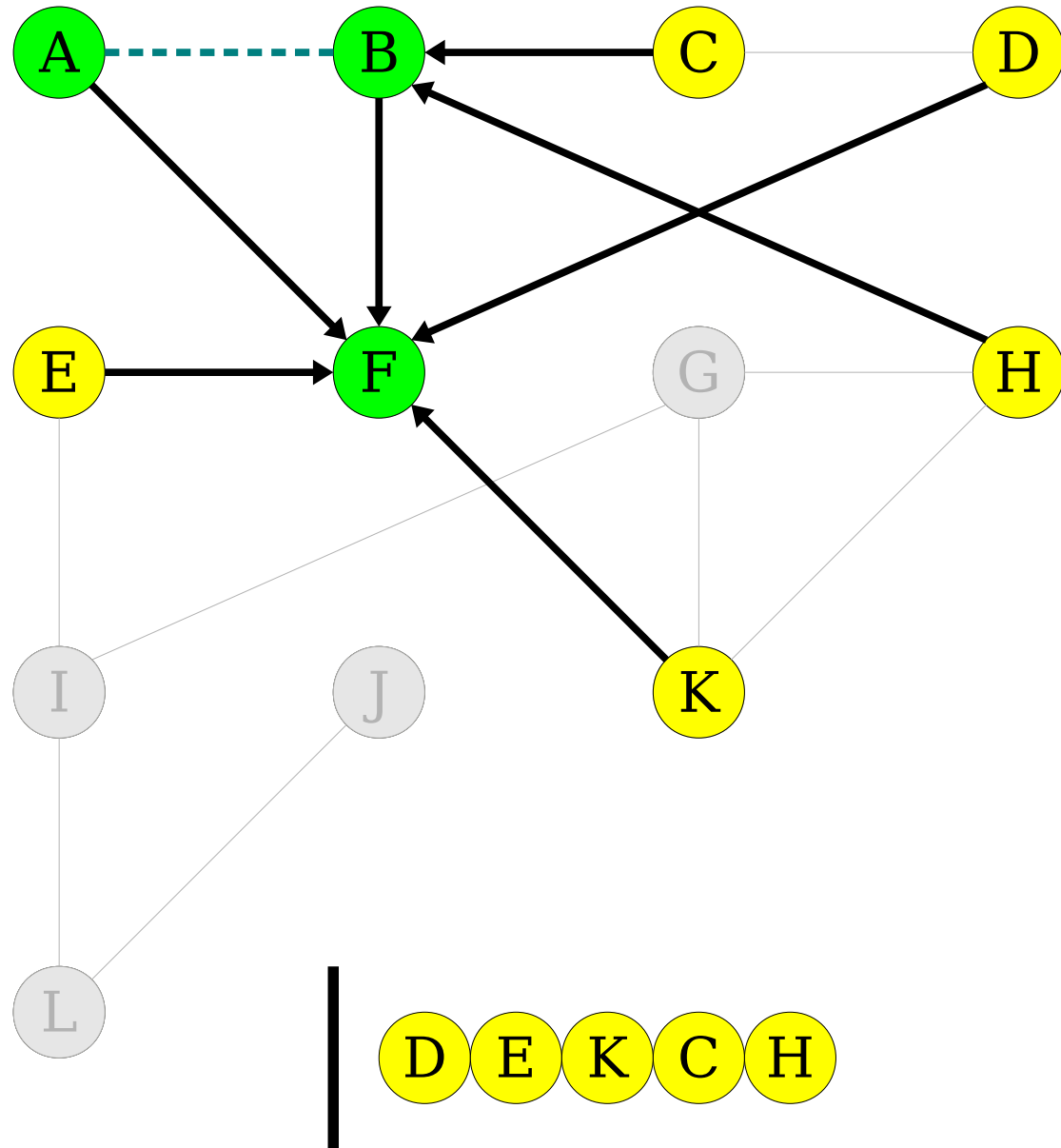


# Breadth-First Search

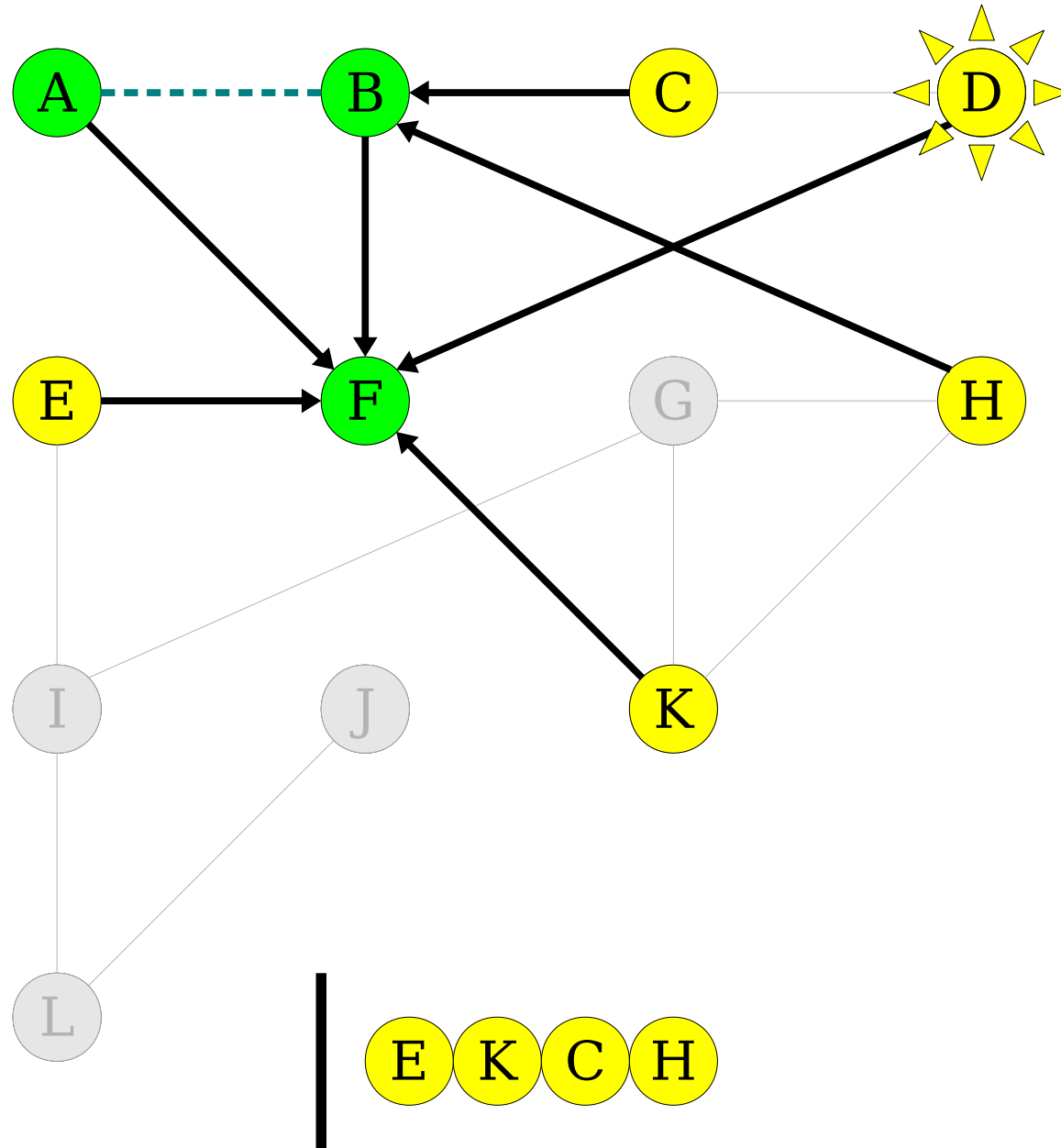




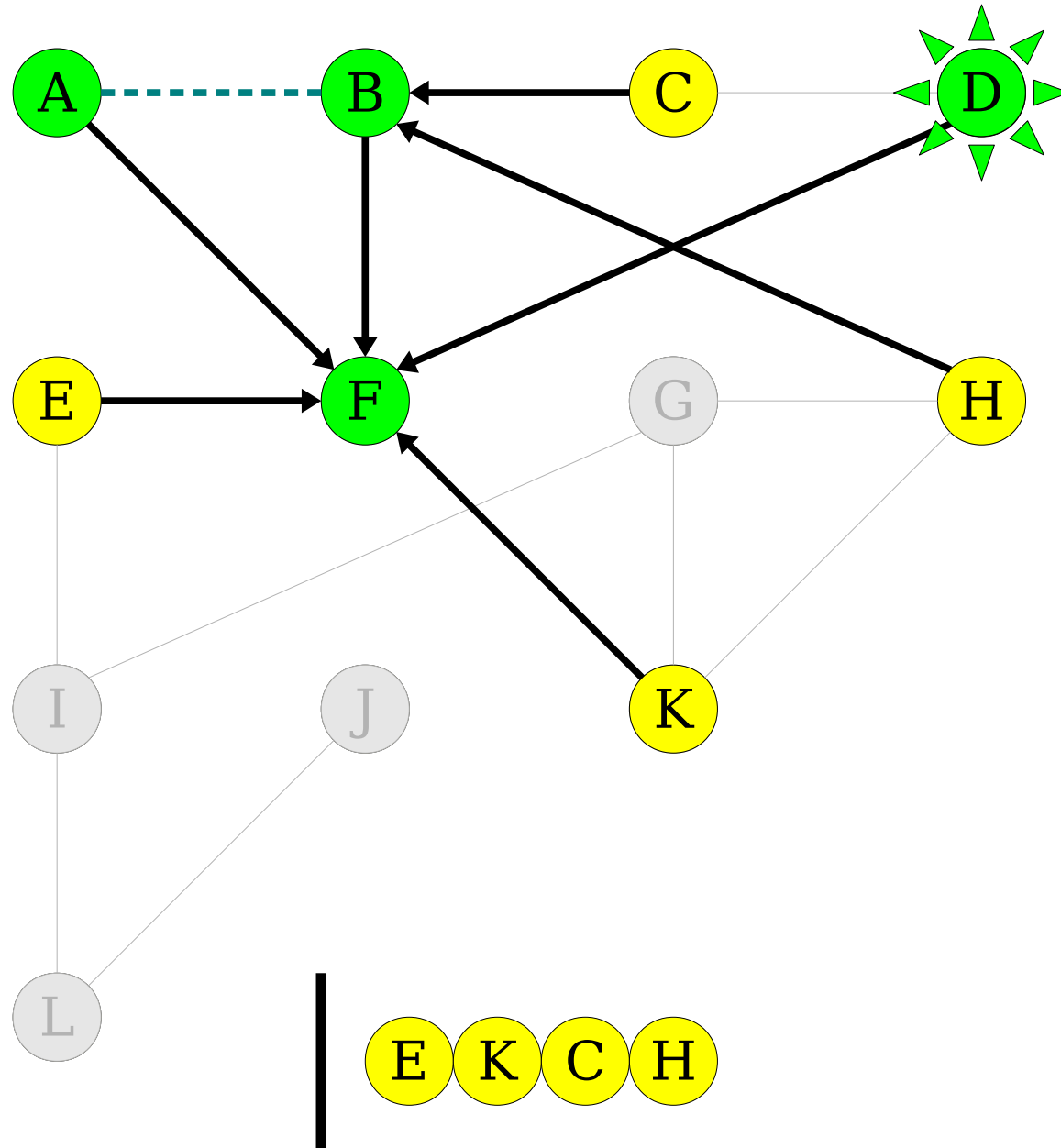
# Breadth-First Search



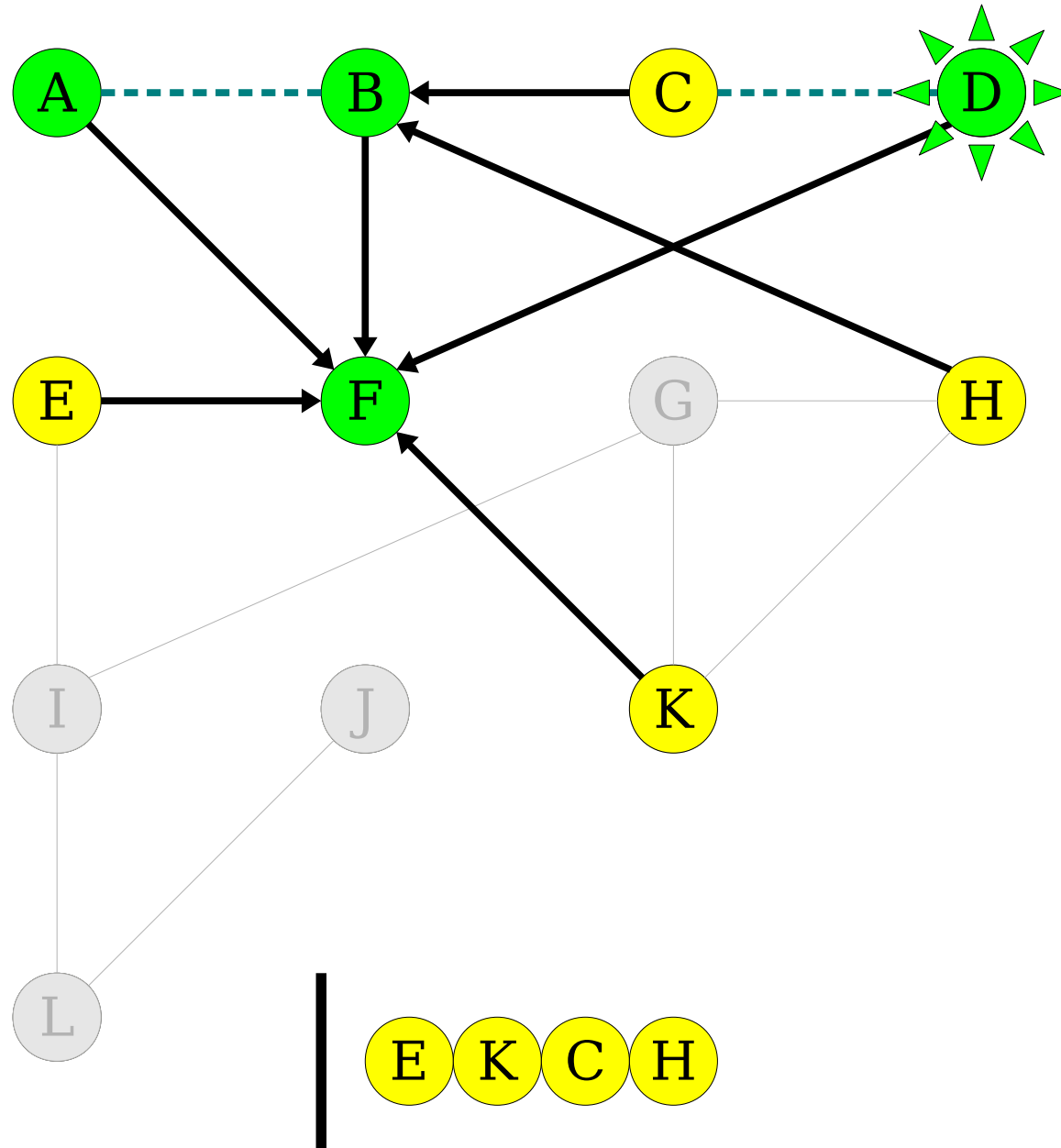
# Breadth-First Search



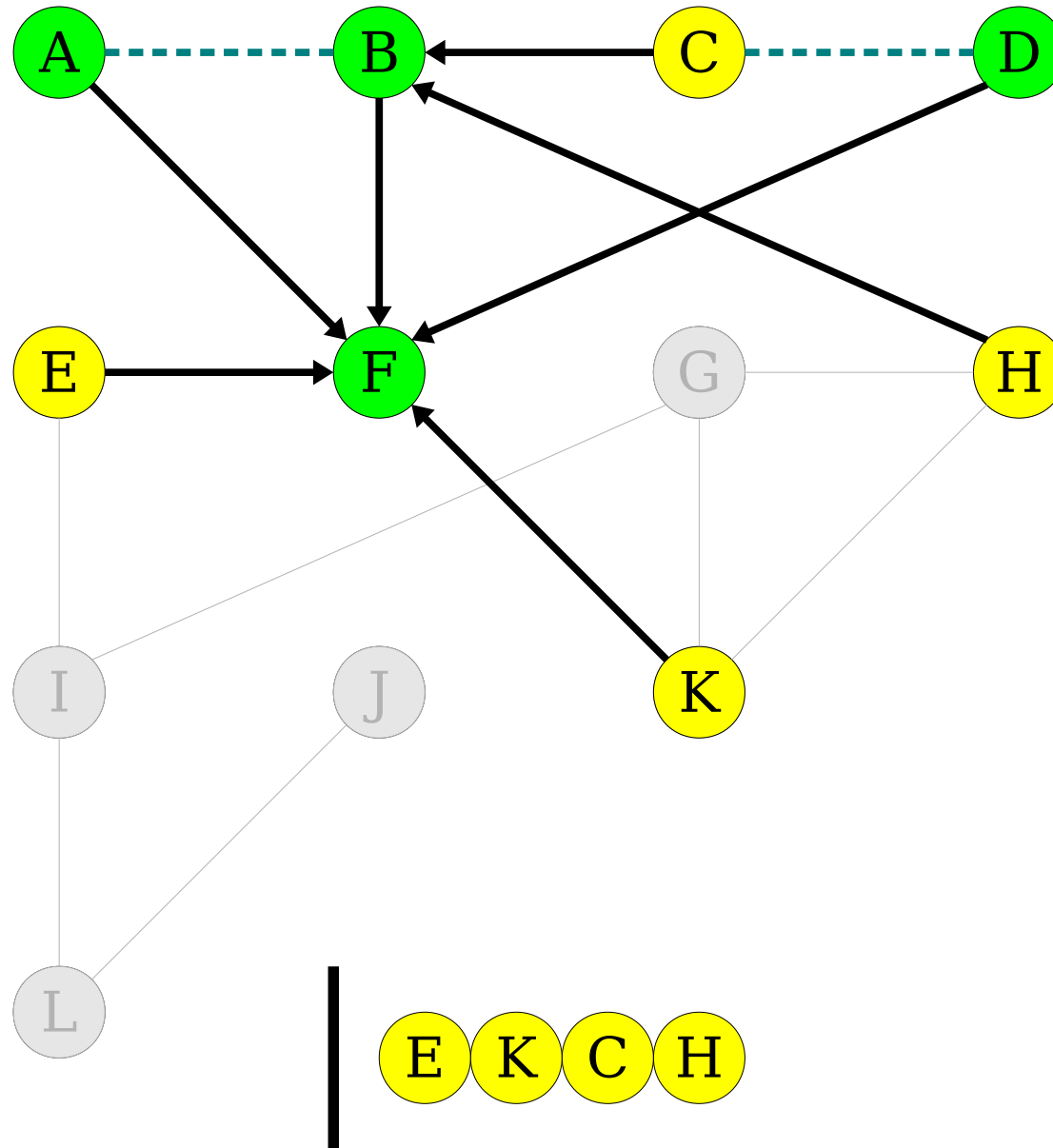
# Breadth-First Search



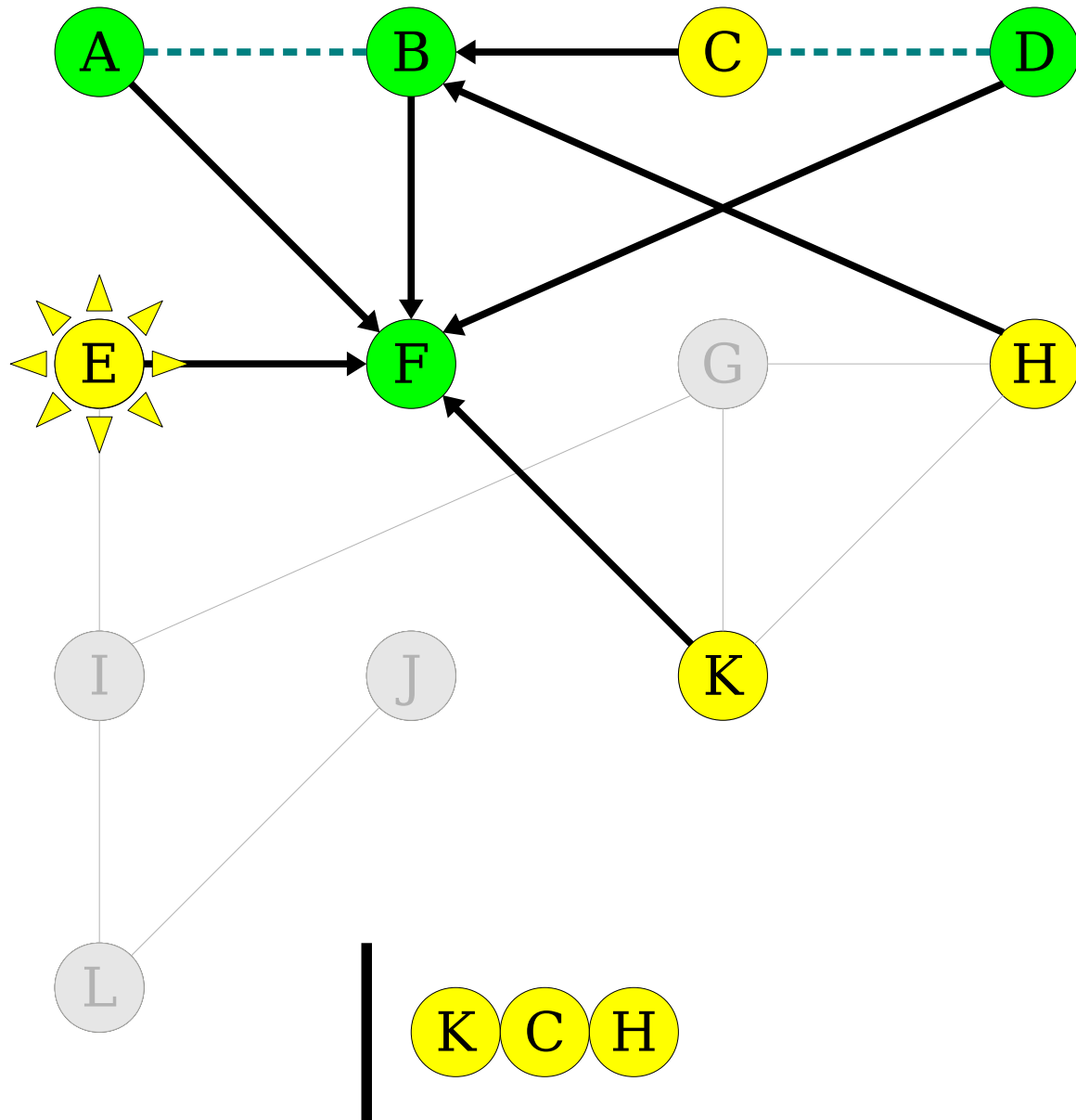
# Breadth-First Search



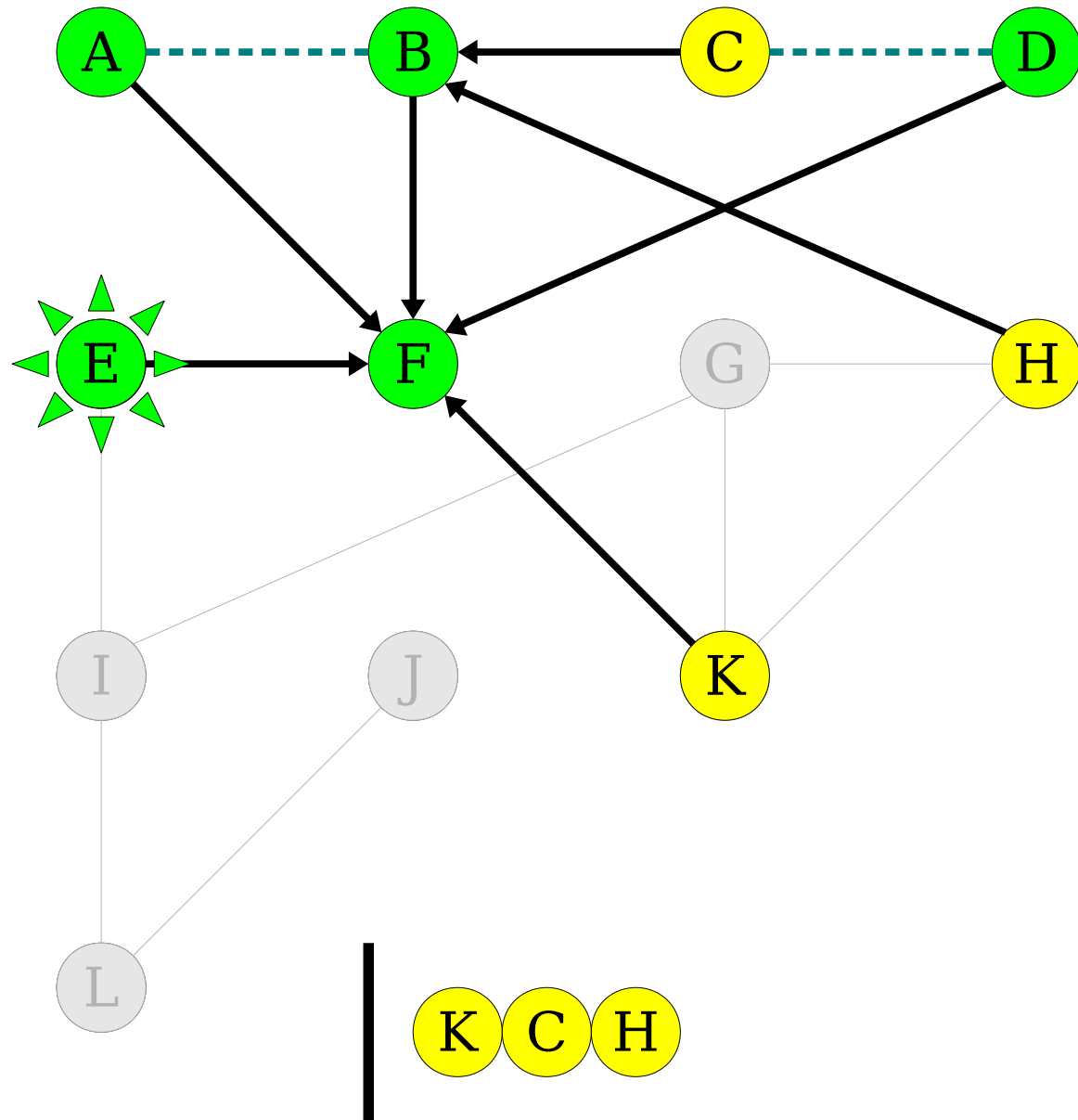
# Breadth-First Search



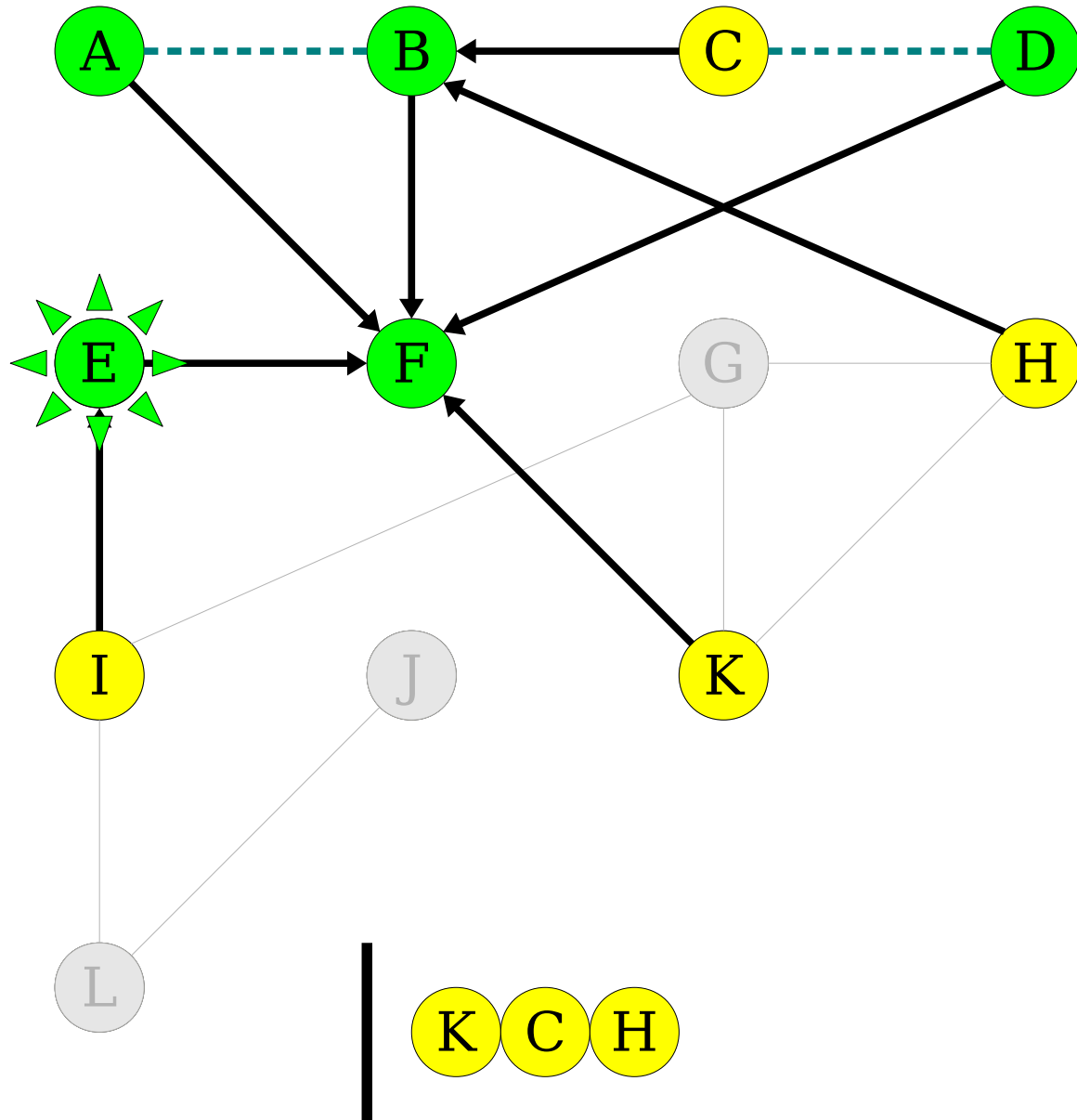
# Breadth-First Search



# Breadth-First Search

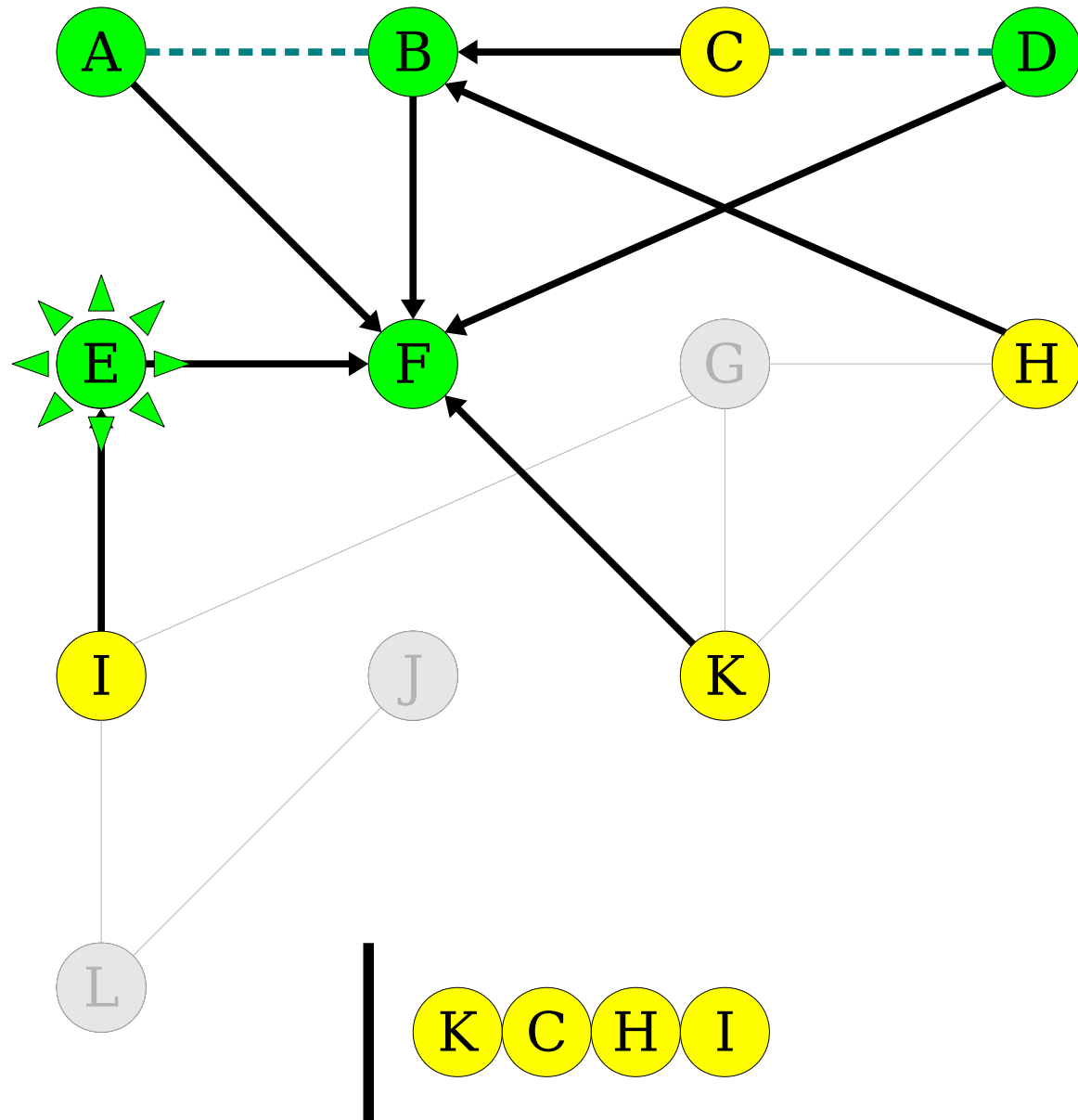


# Breadth-First Search

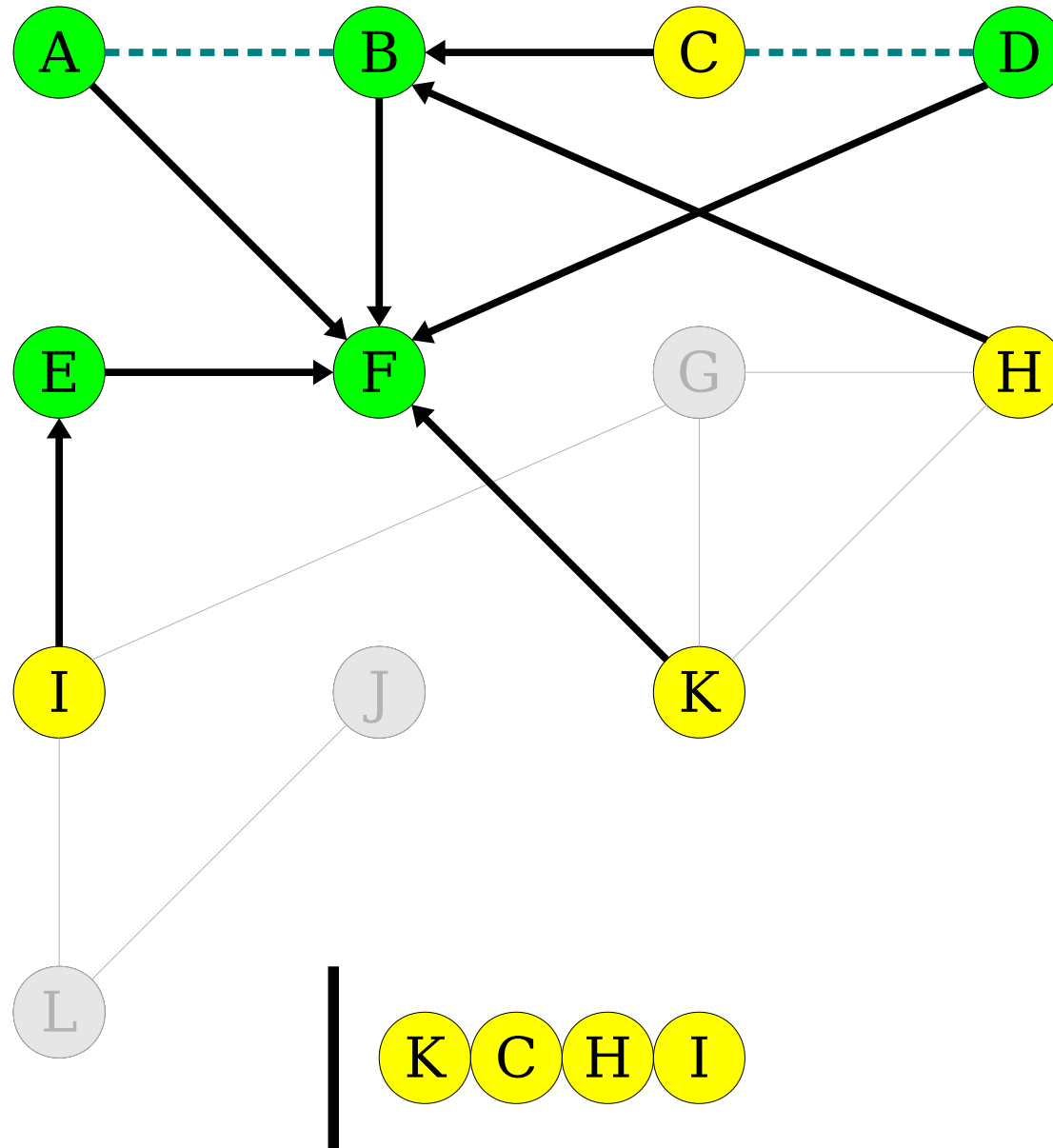




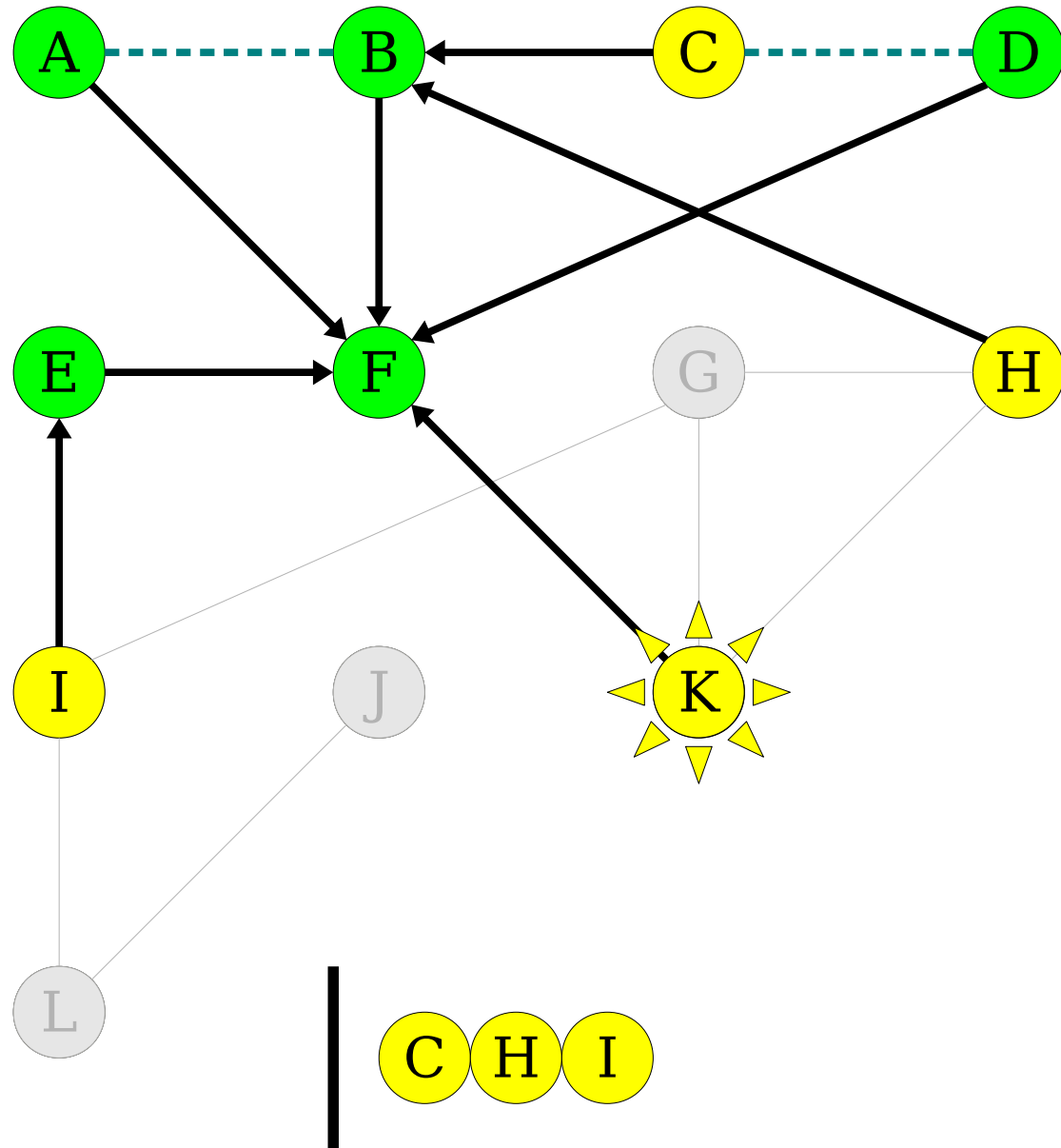
# Breadth-First Search



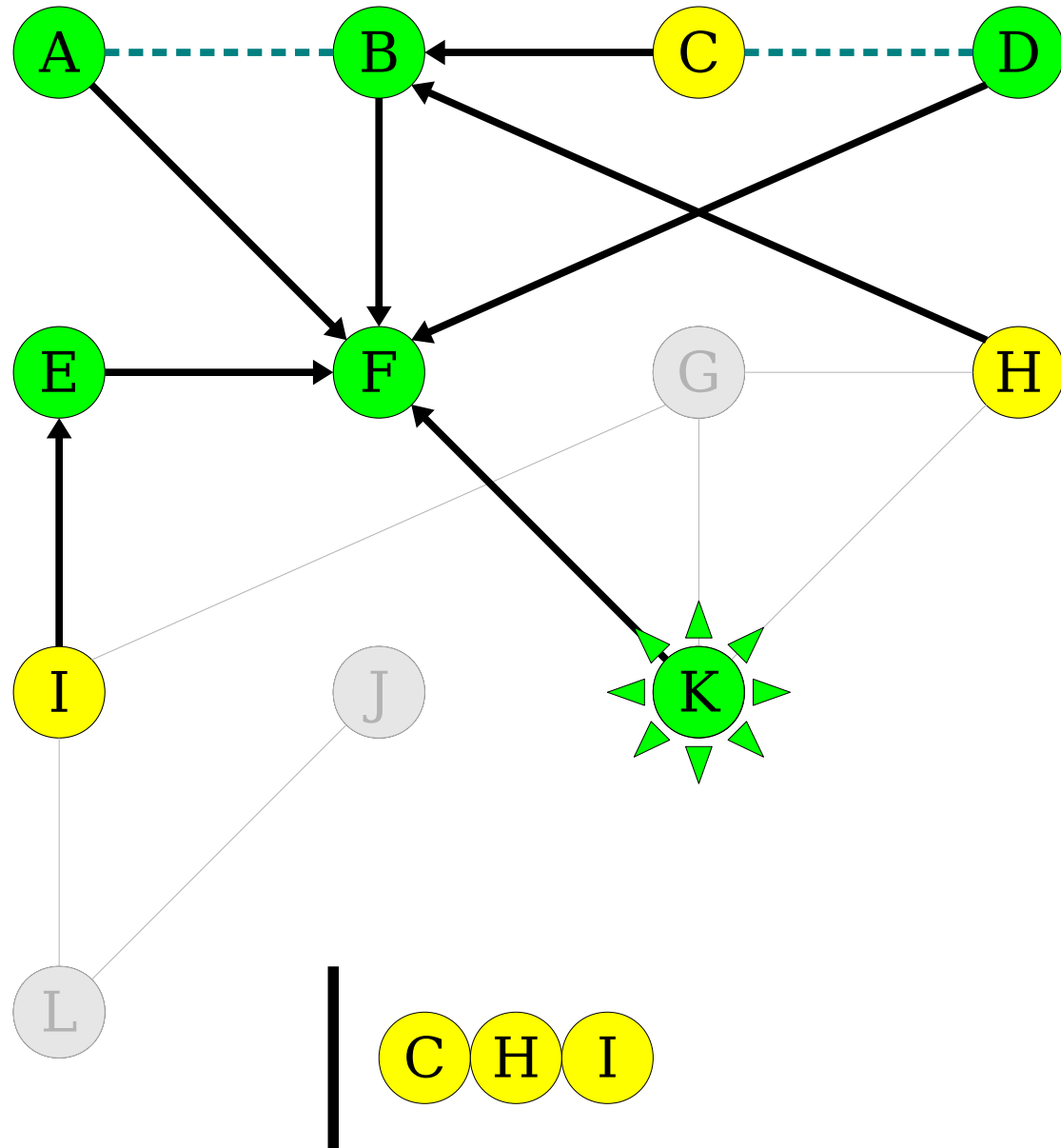
# Breadth-First Search



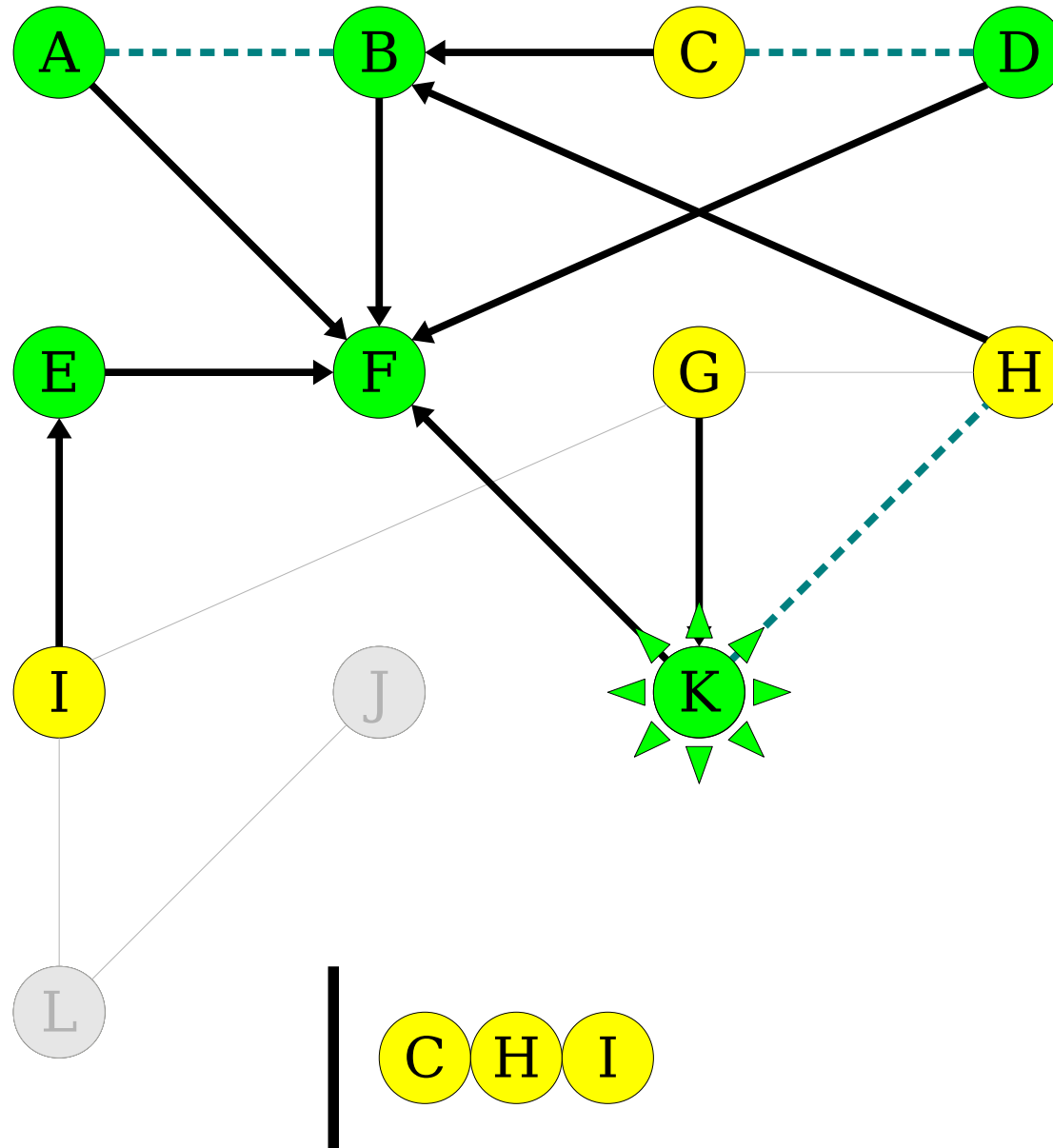
# Breadth-First Search



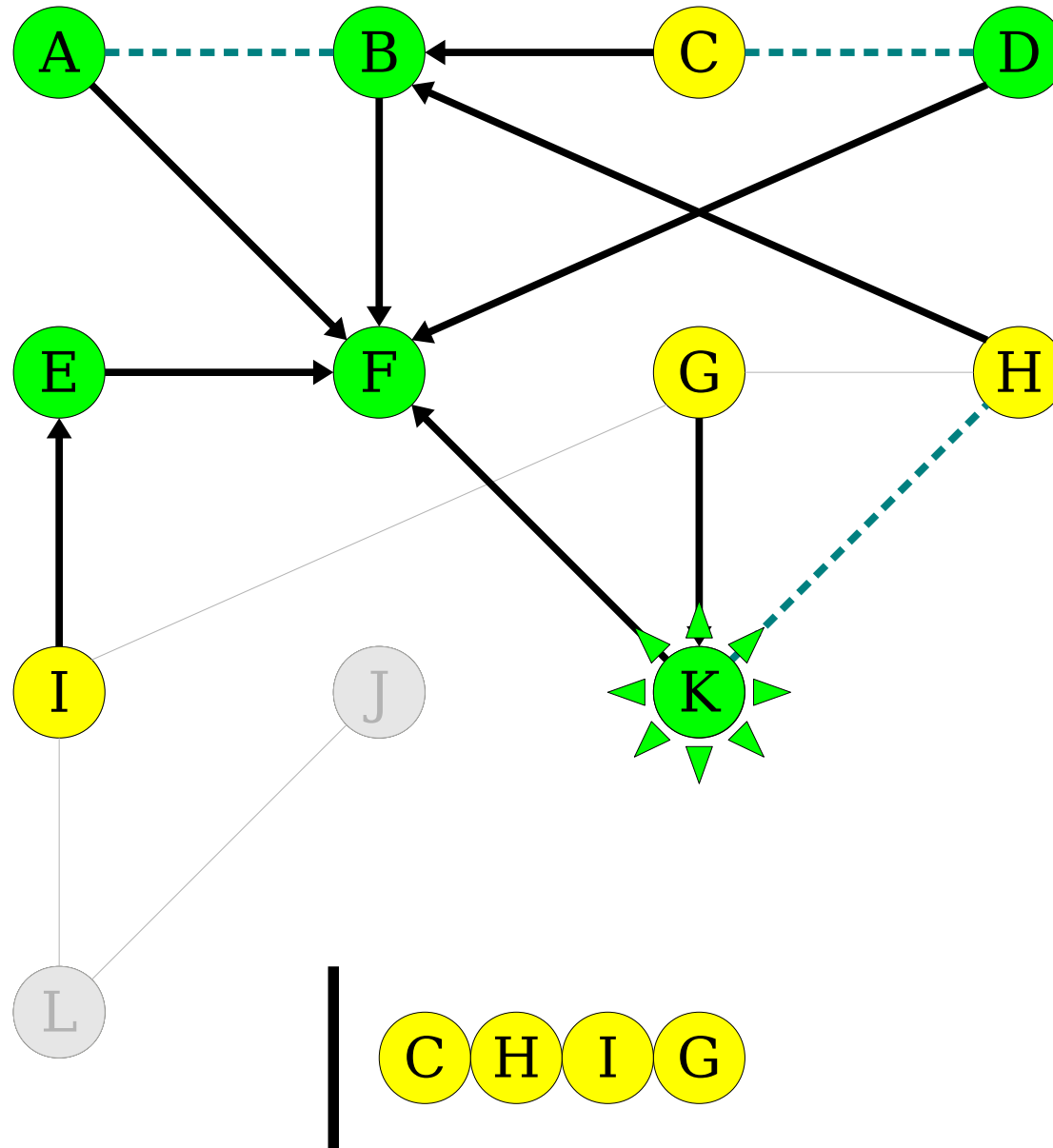
# Breadth-First Search



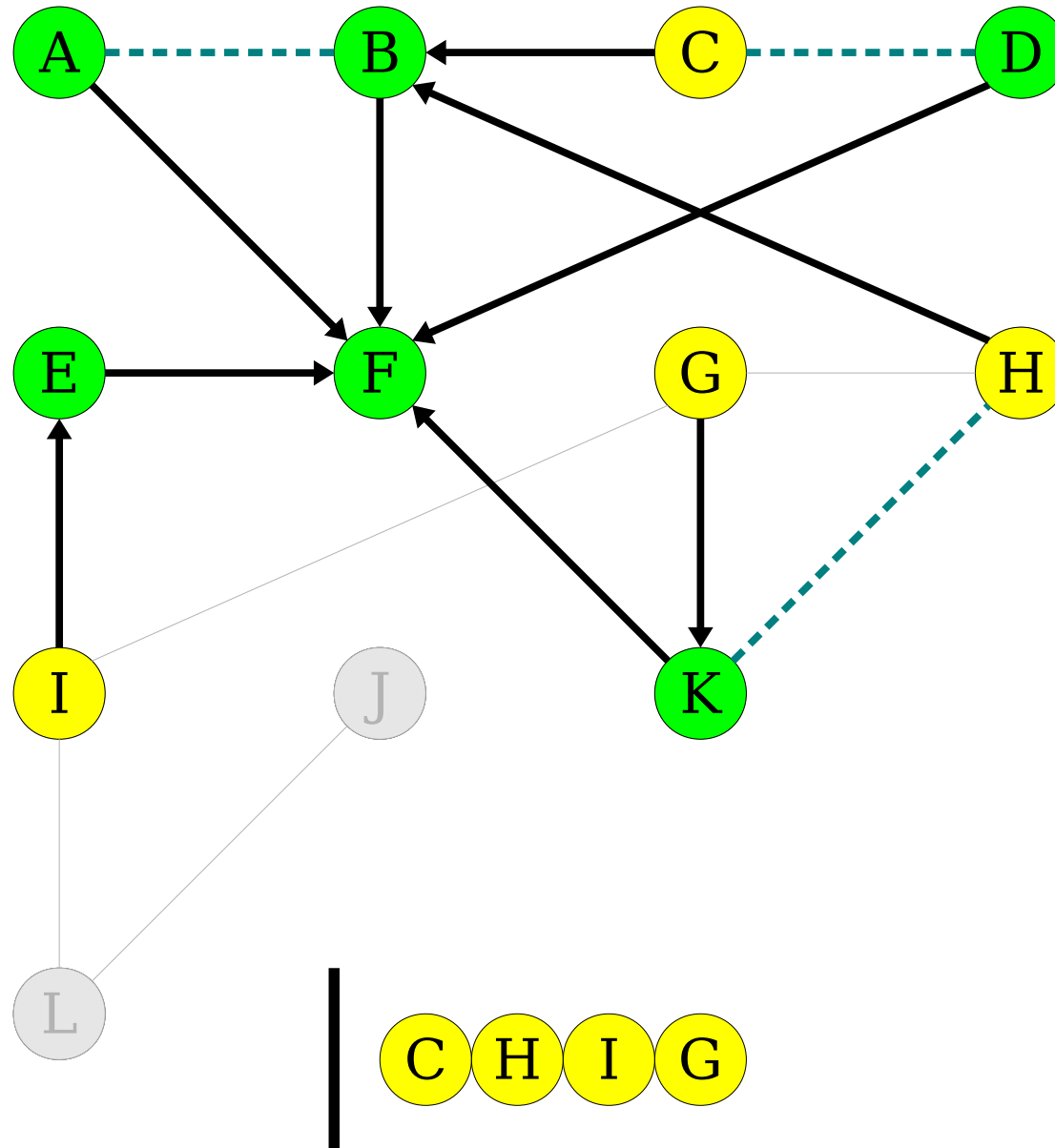
# Breadth-First Search



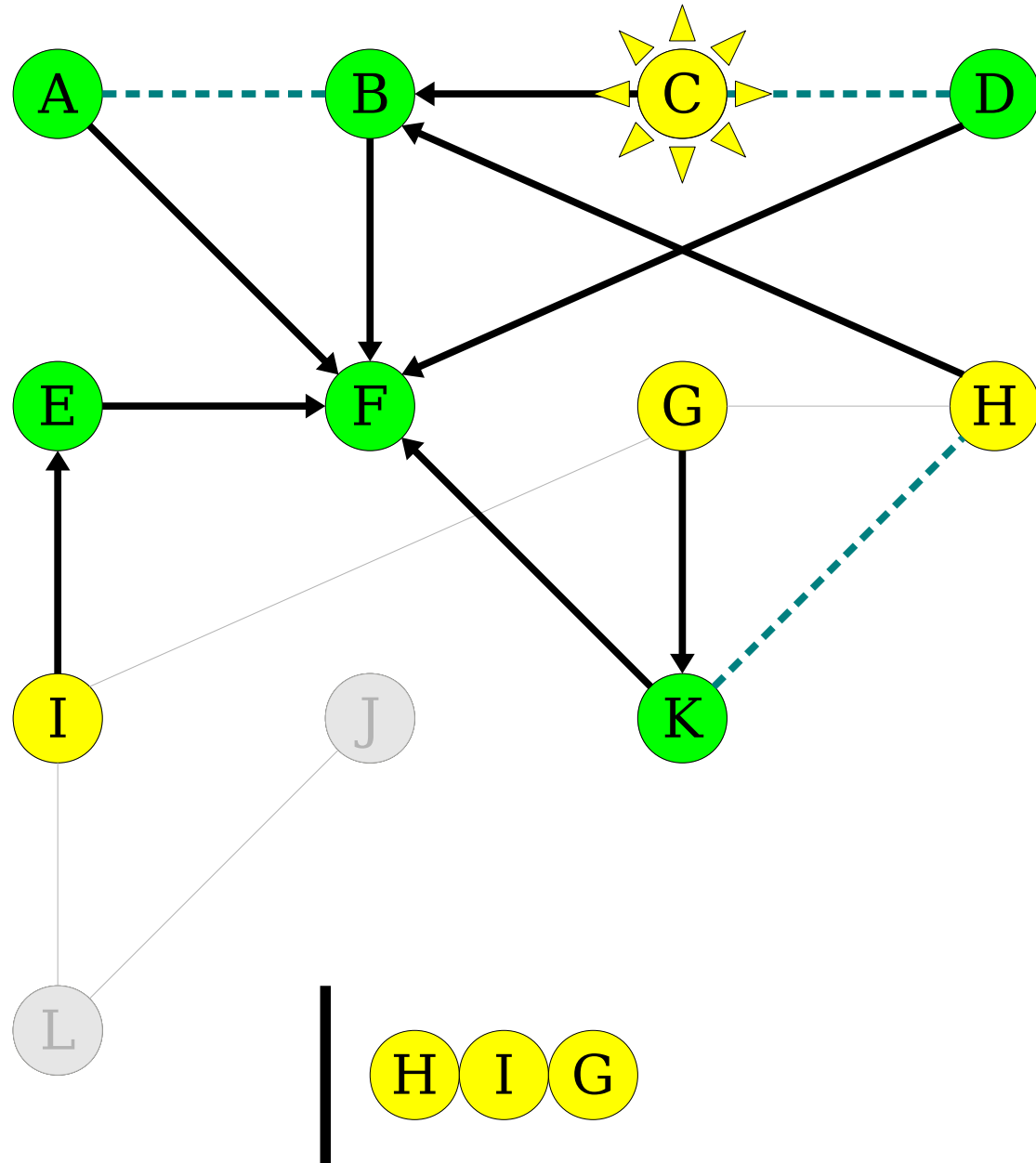
# Breadth-First Search



# Breadth-First Search

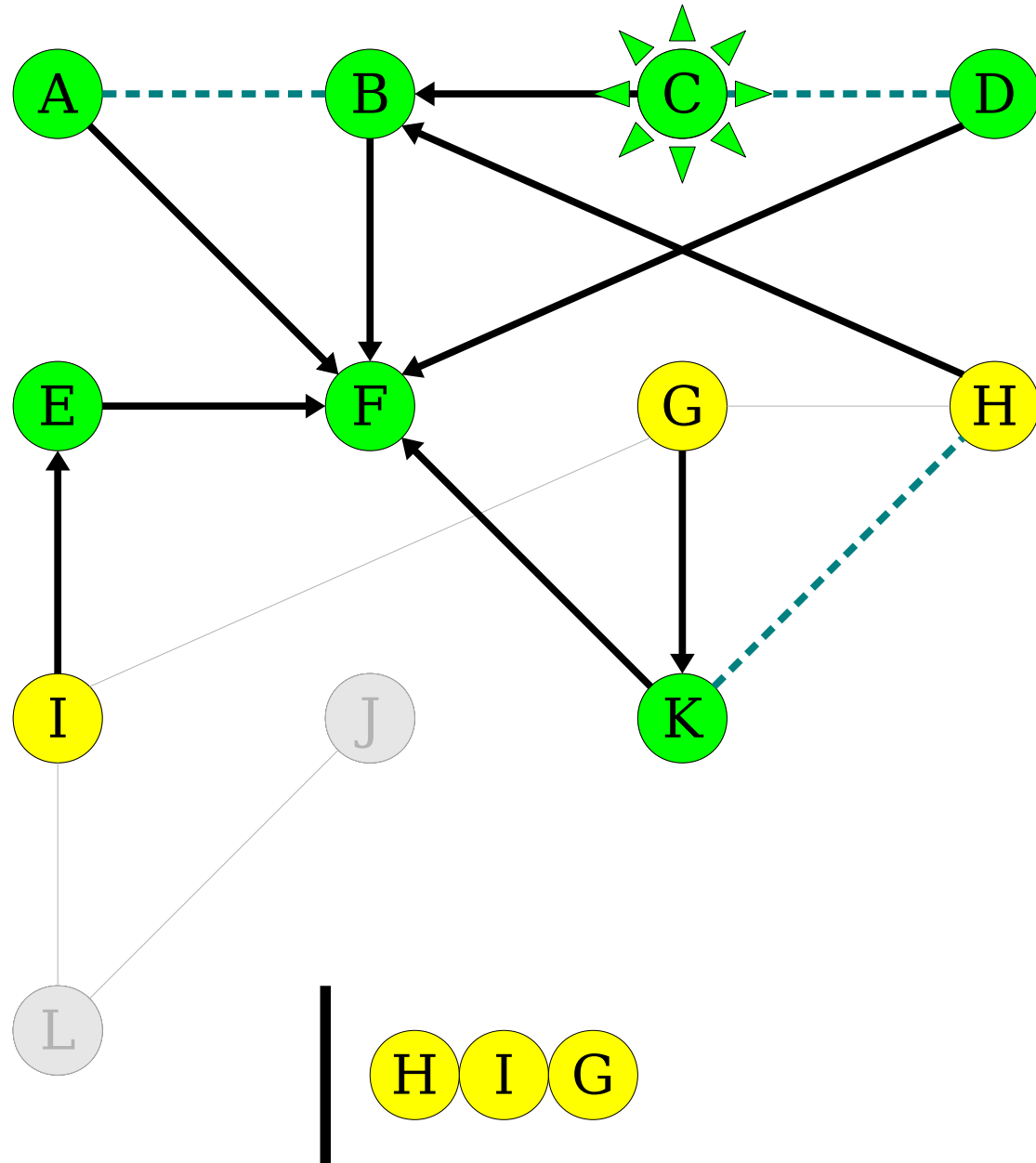


# Breadth-First Search

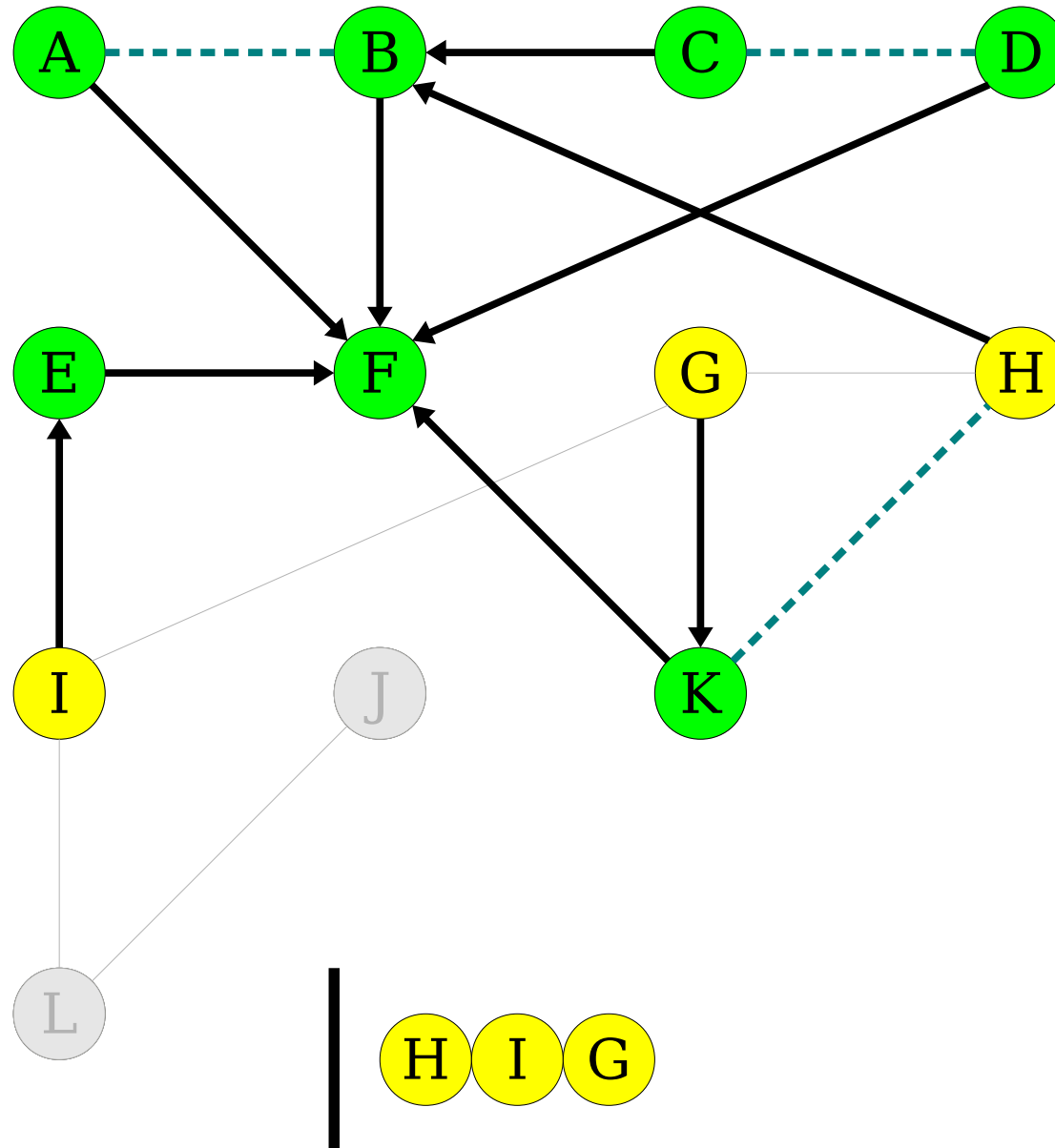




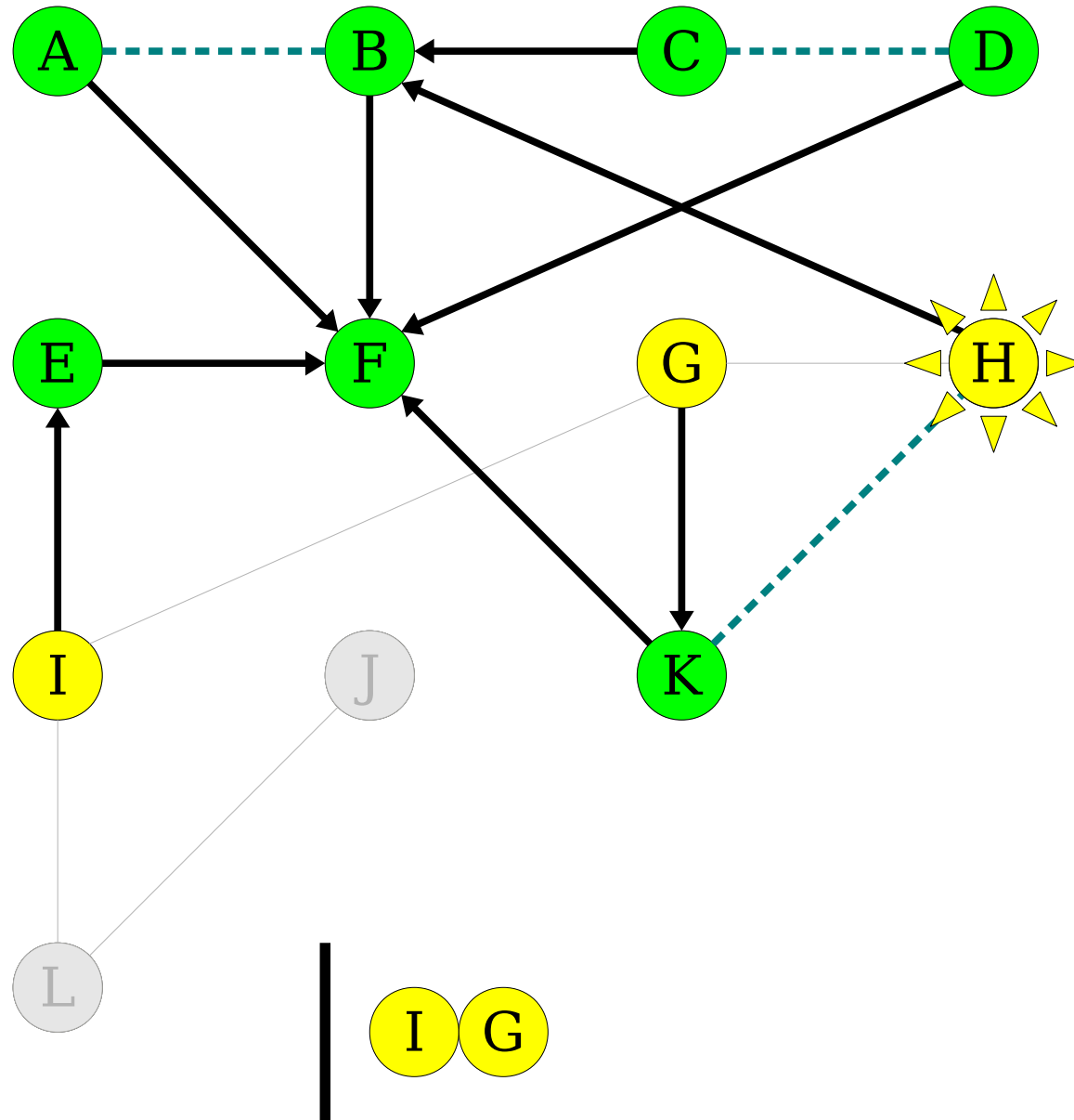
# Breadth-First Search



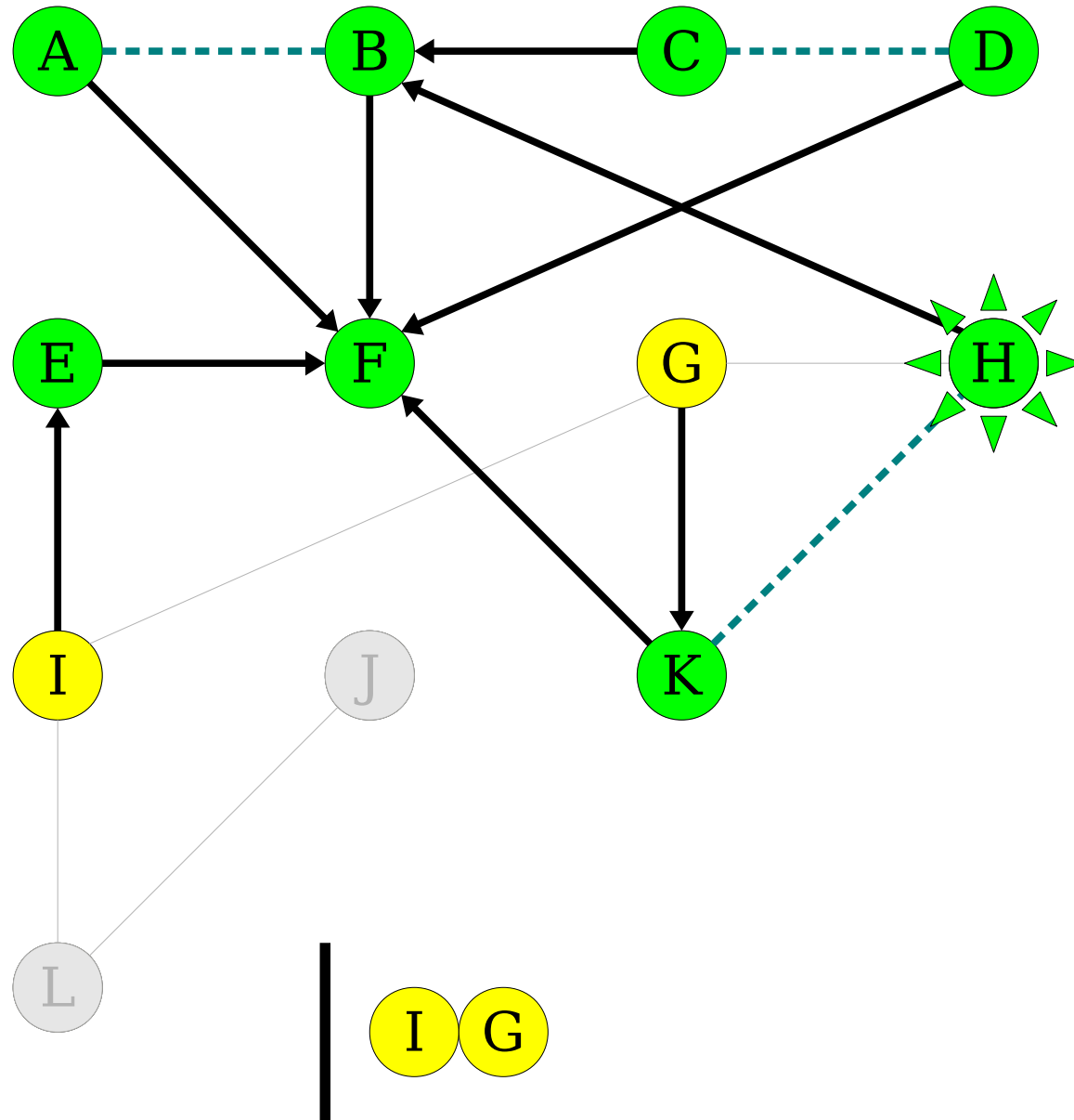
# Breadth-First Search



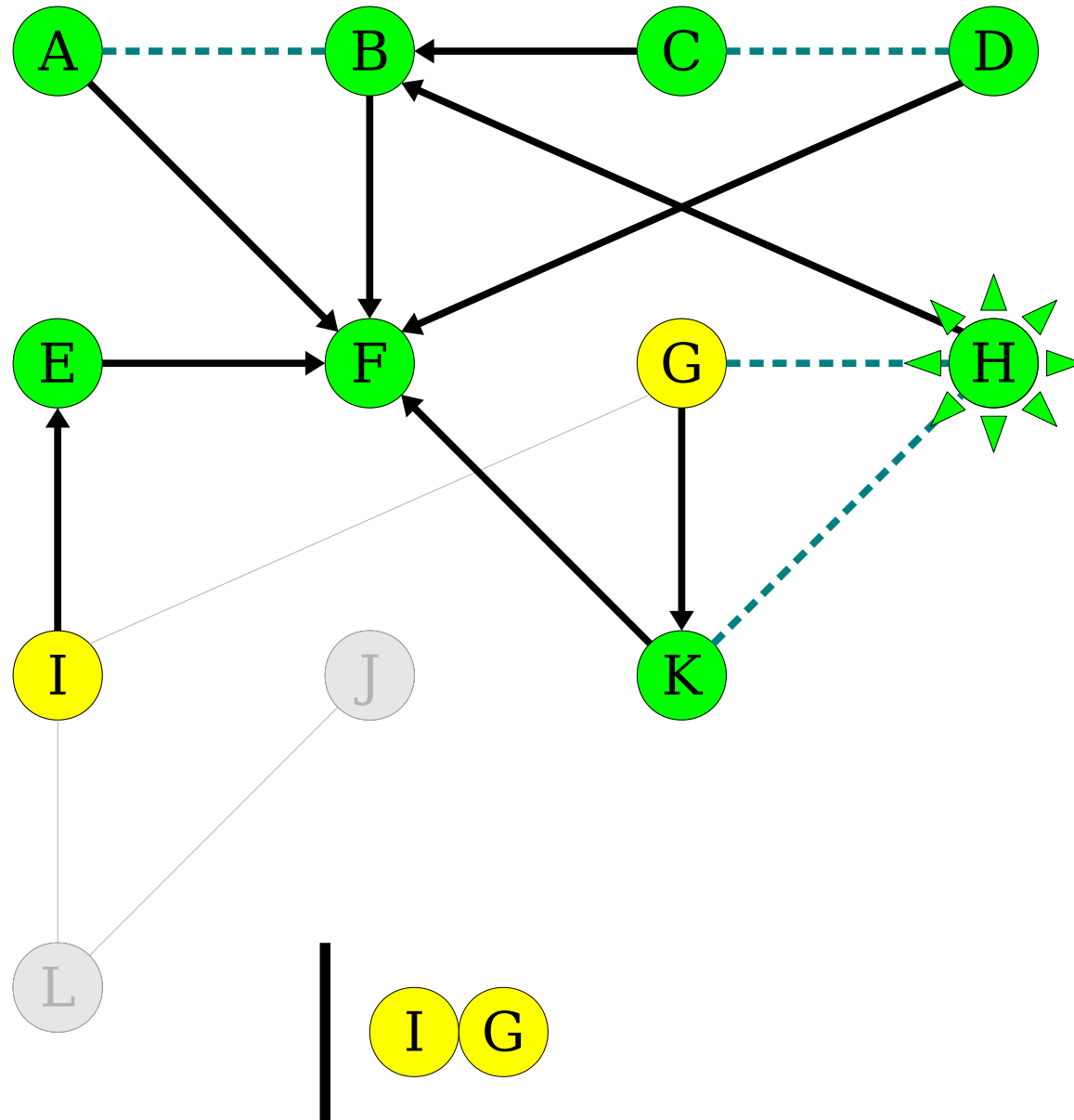
# Breadth-First Search



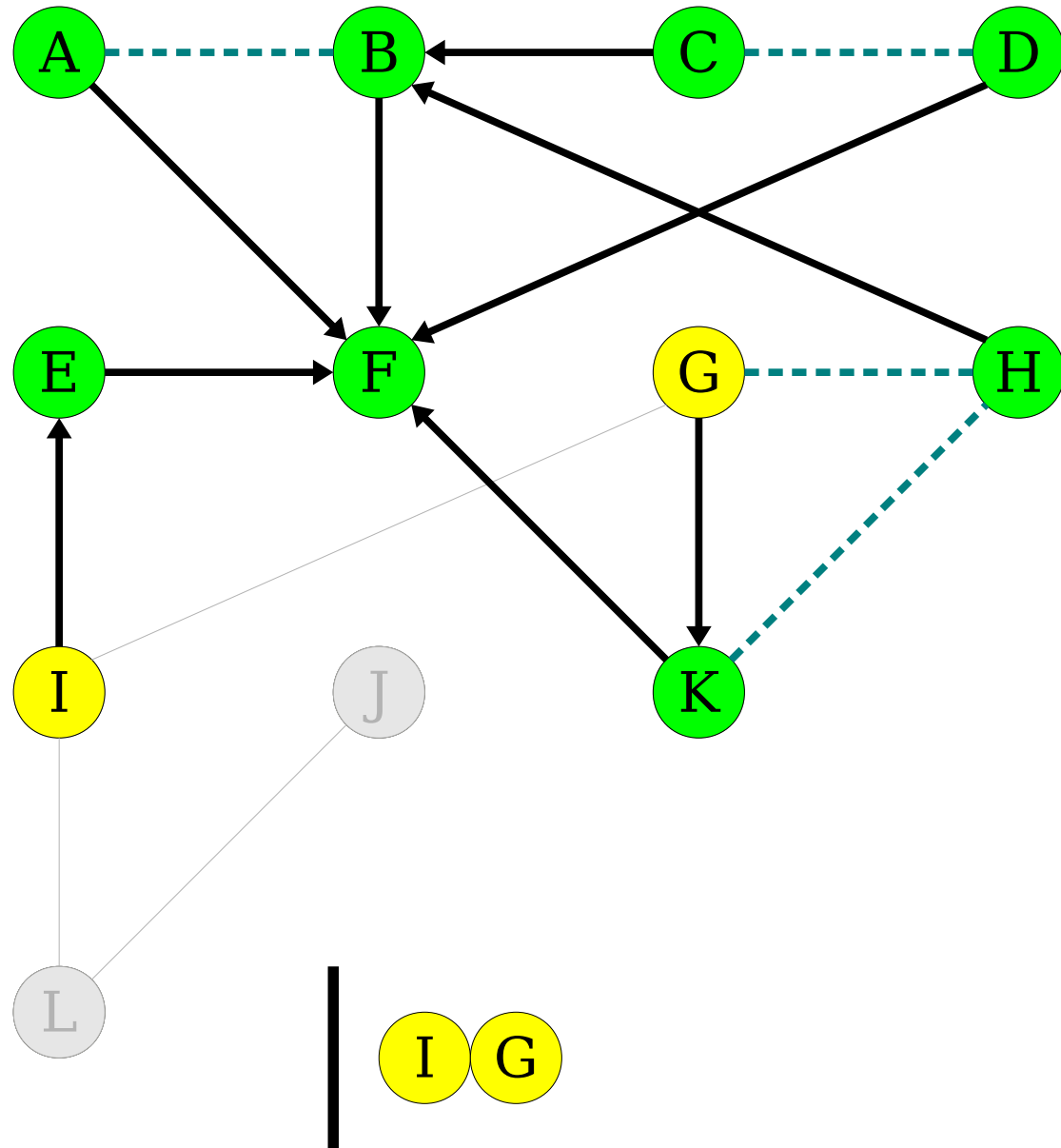
# Breadth-First Search



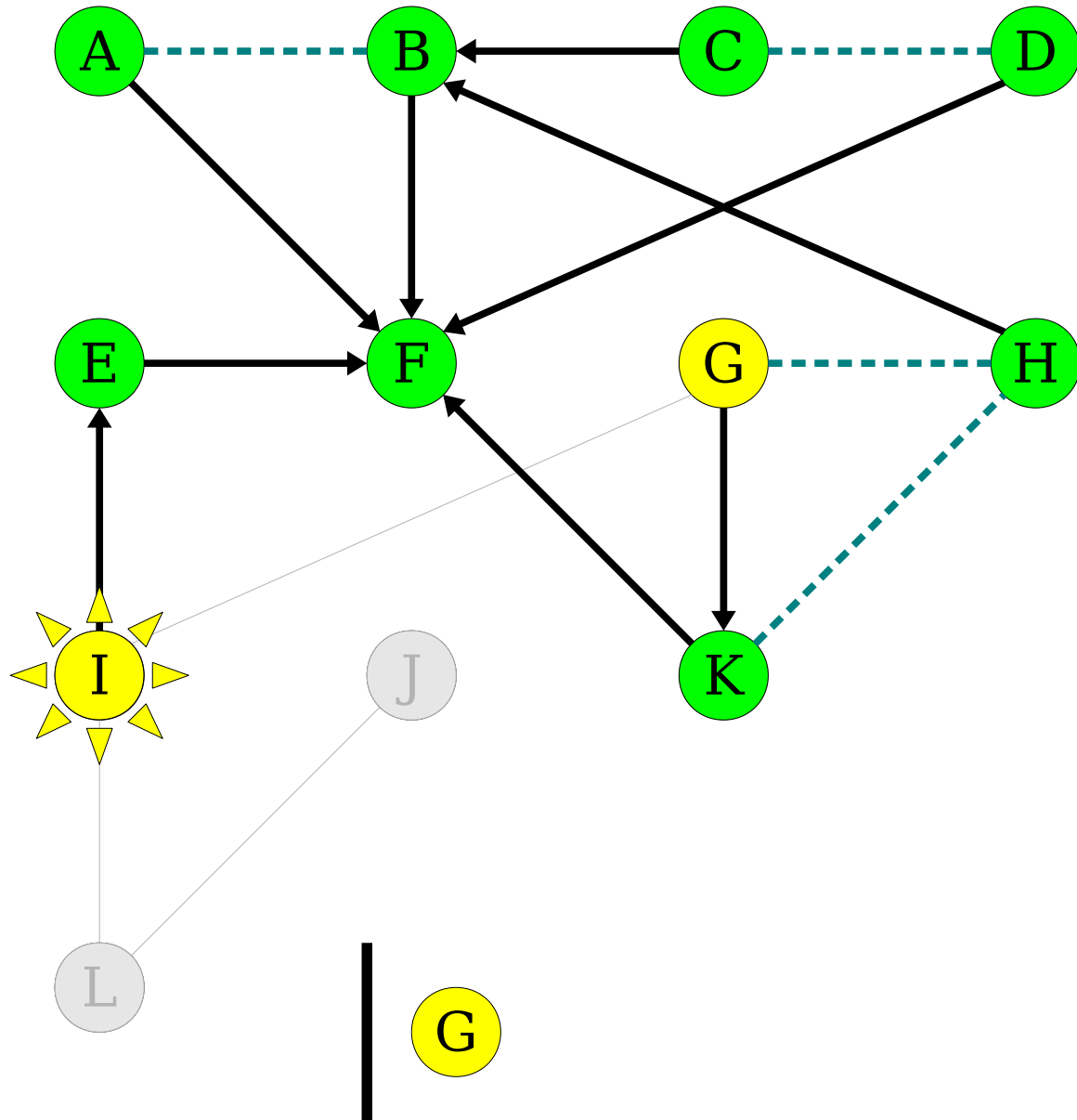
# Breadth-First Search



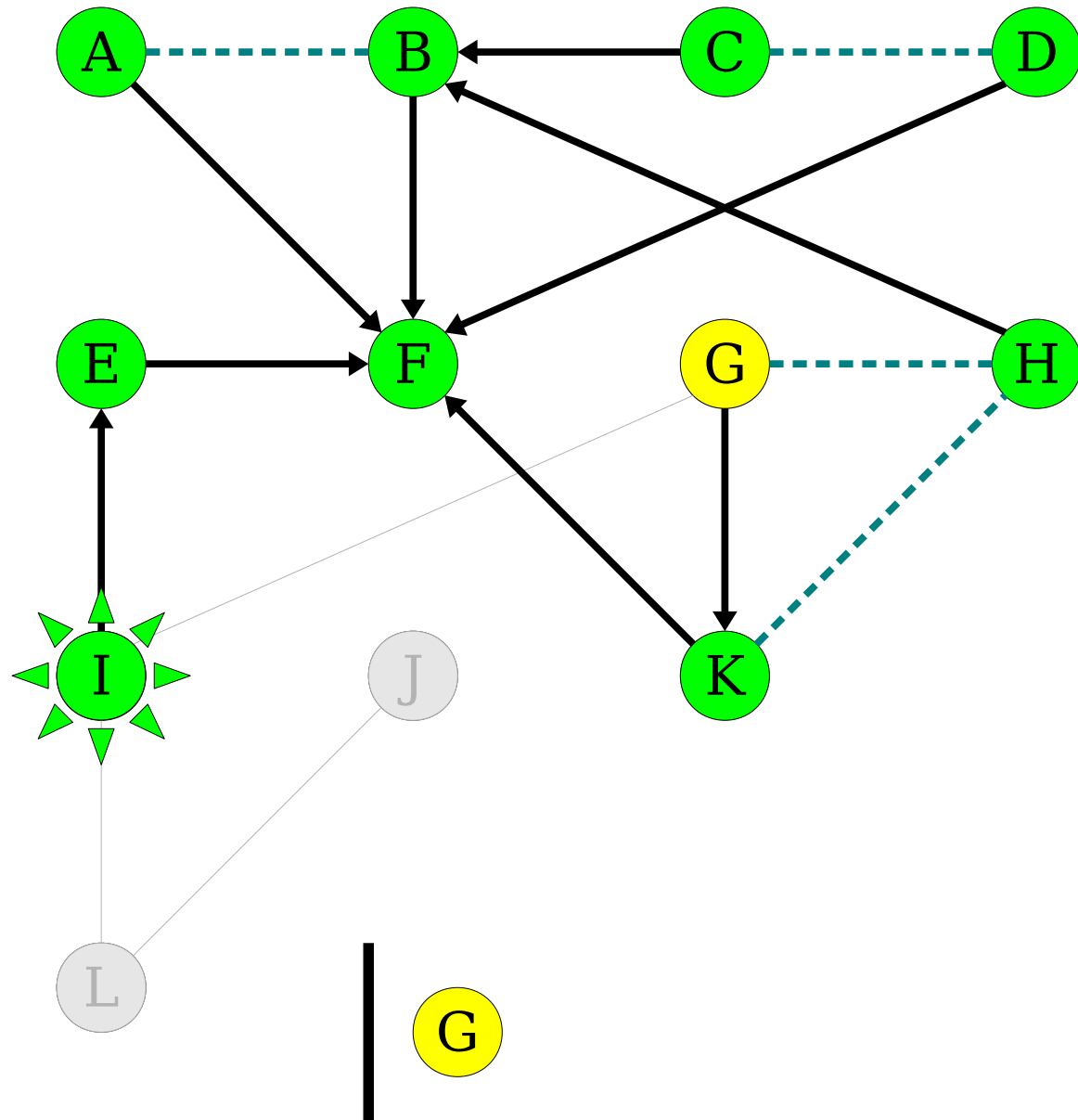
# Breadth-First Search



# Breadth-First Search

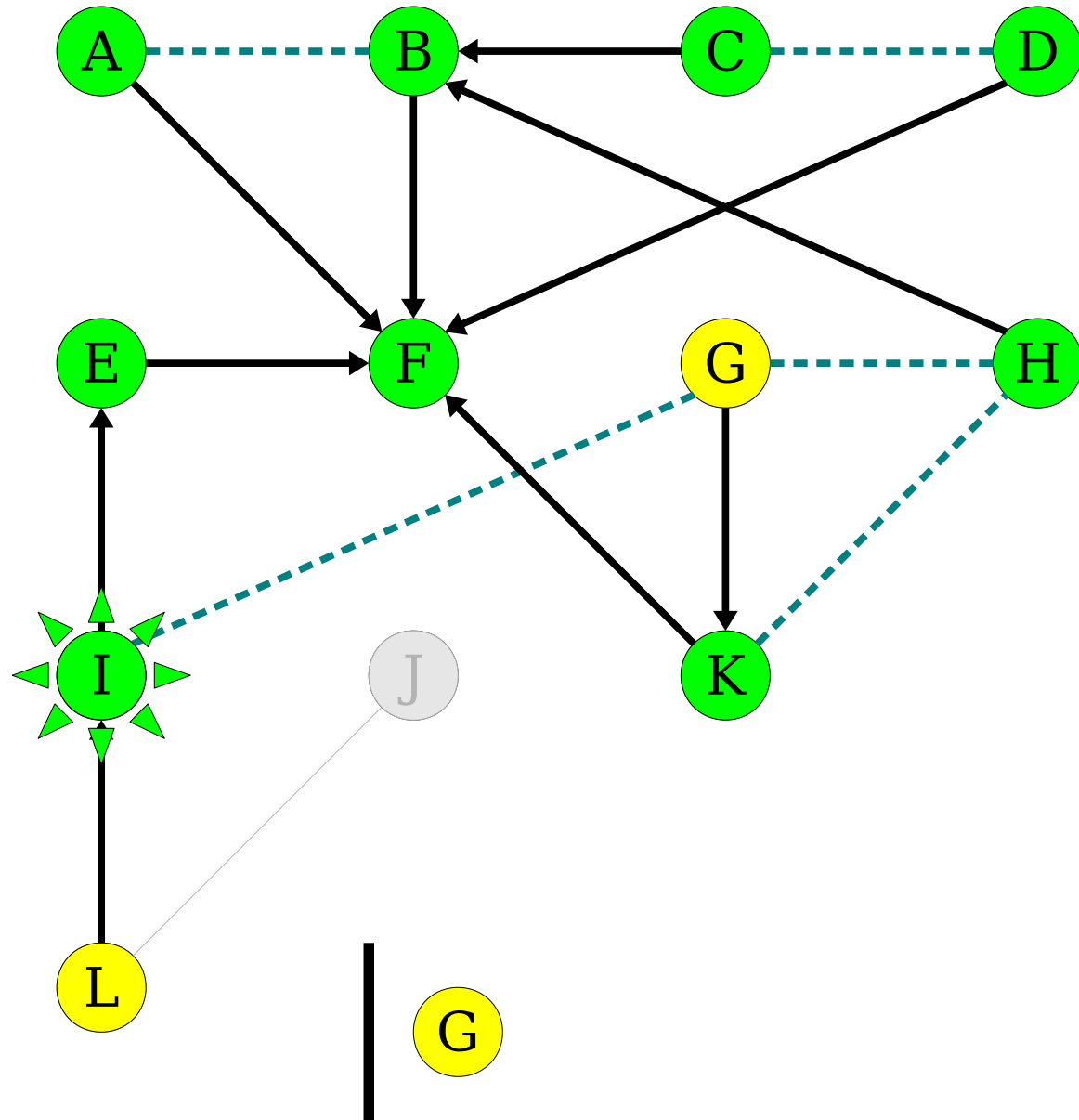


# Breadth-First Search

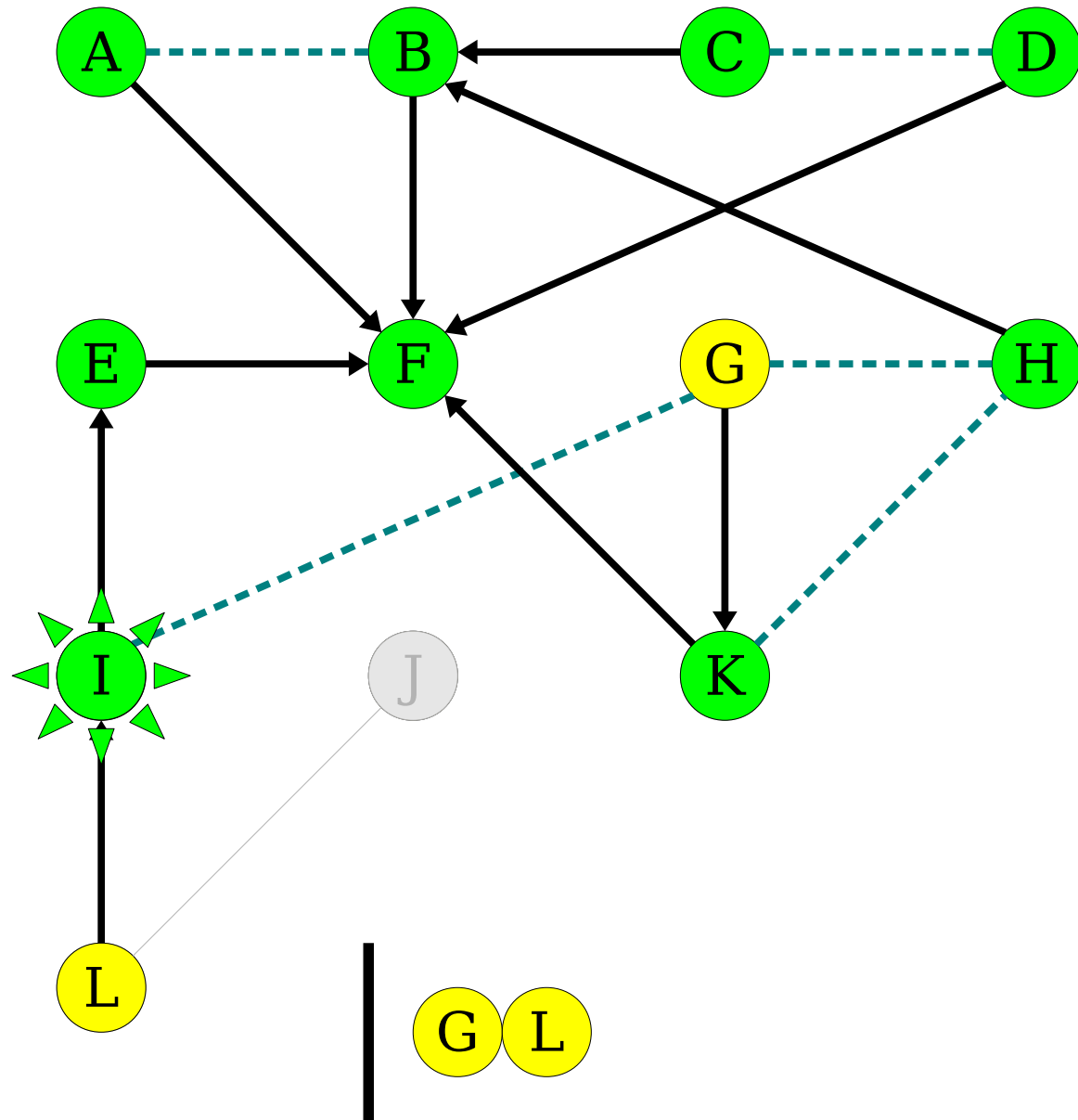




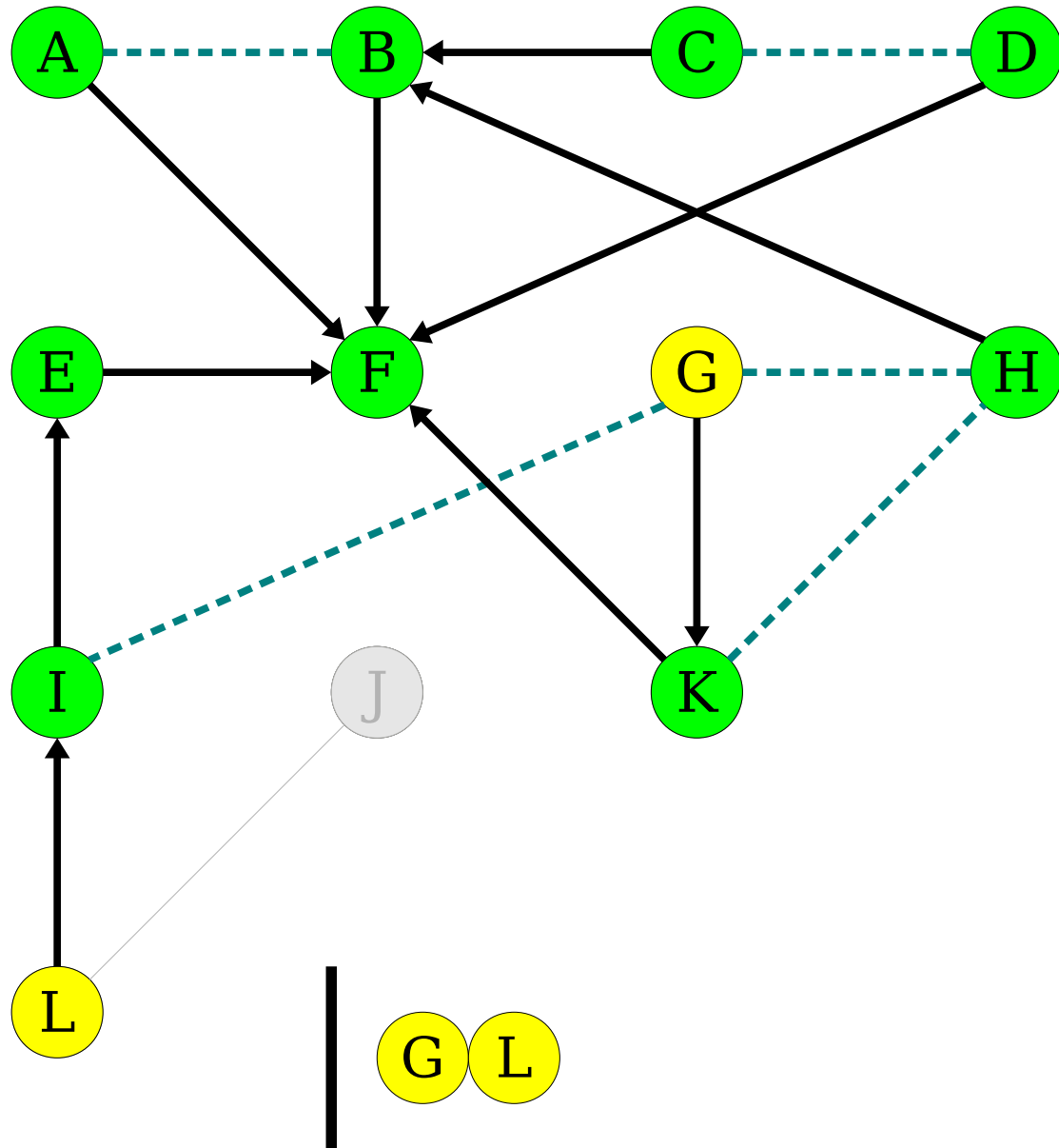
# Breadth-First Search



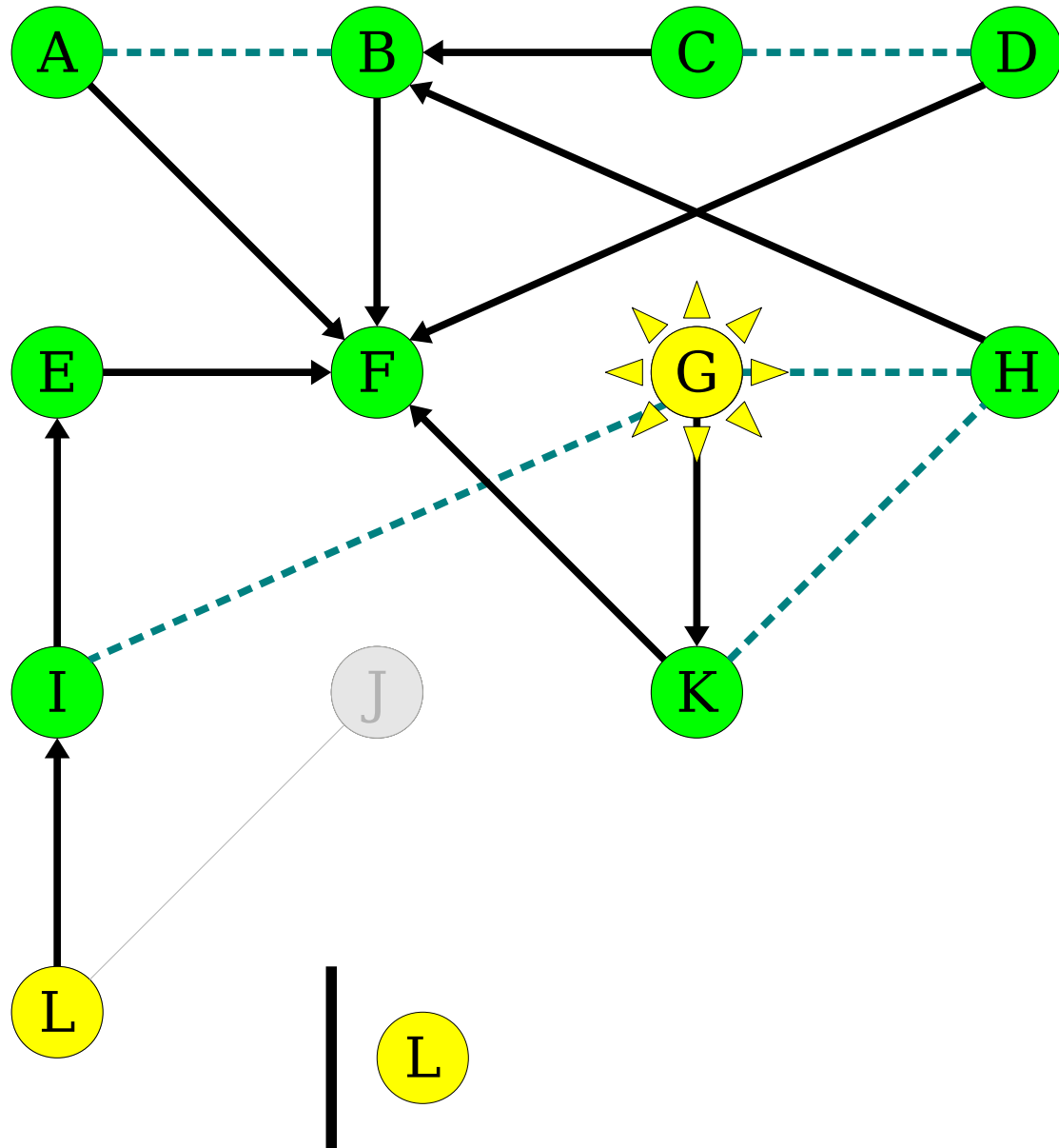
# Breadth-First Search



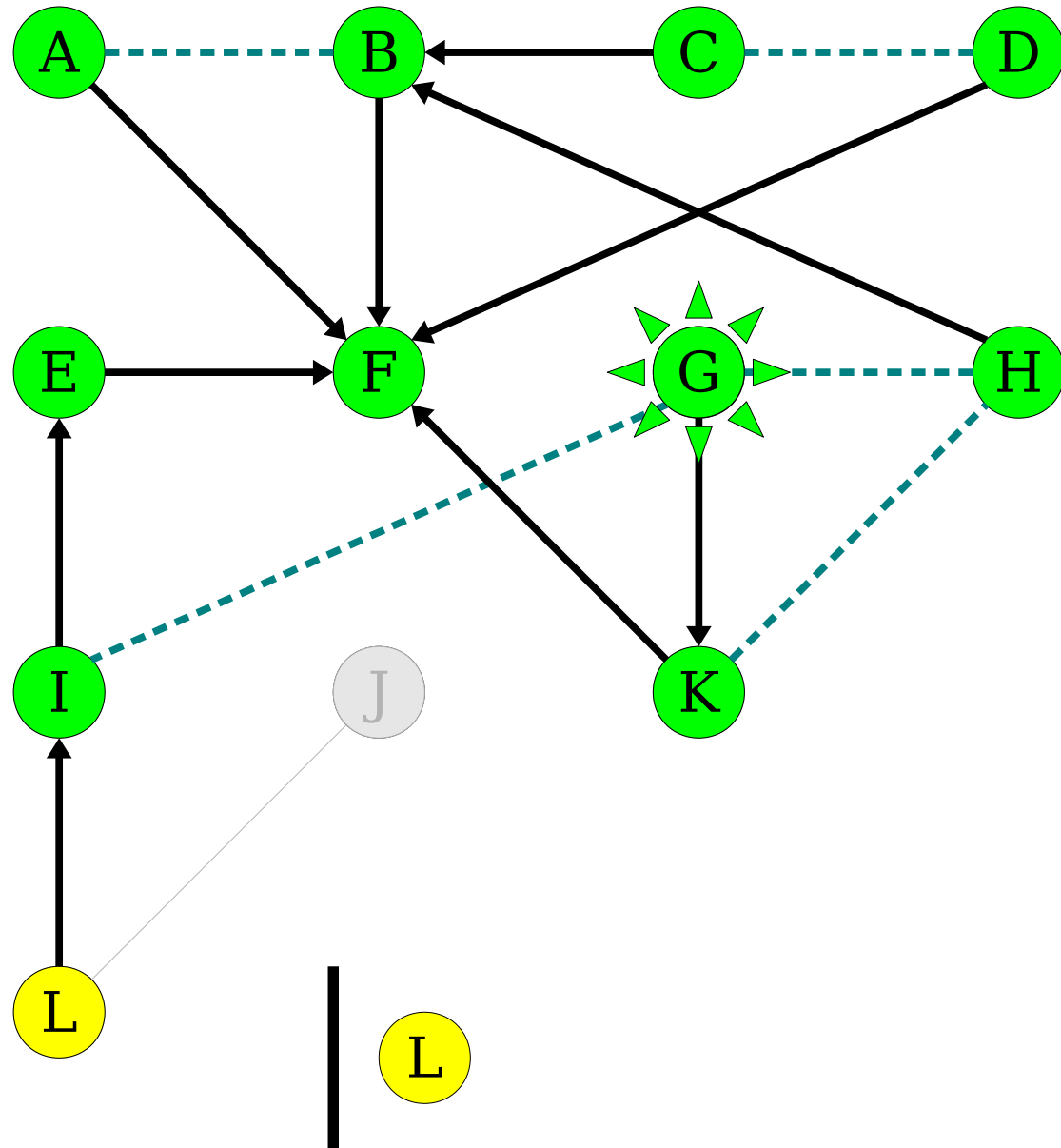
# Breadth-First Search



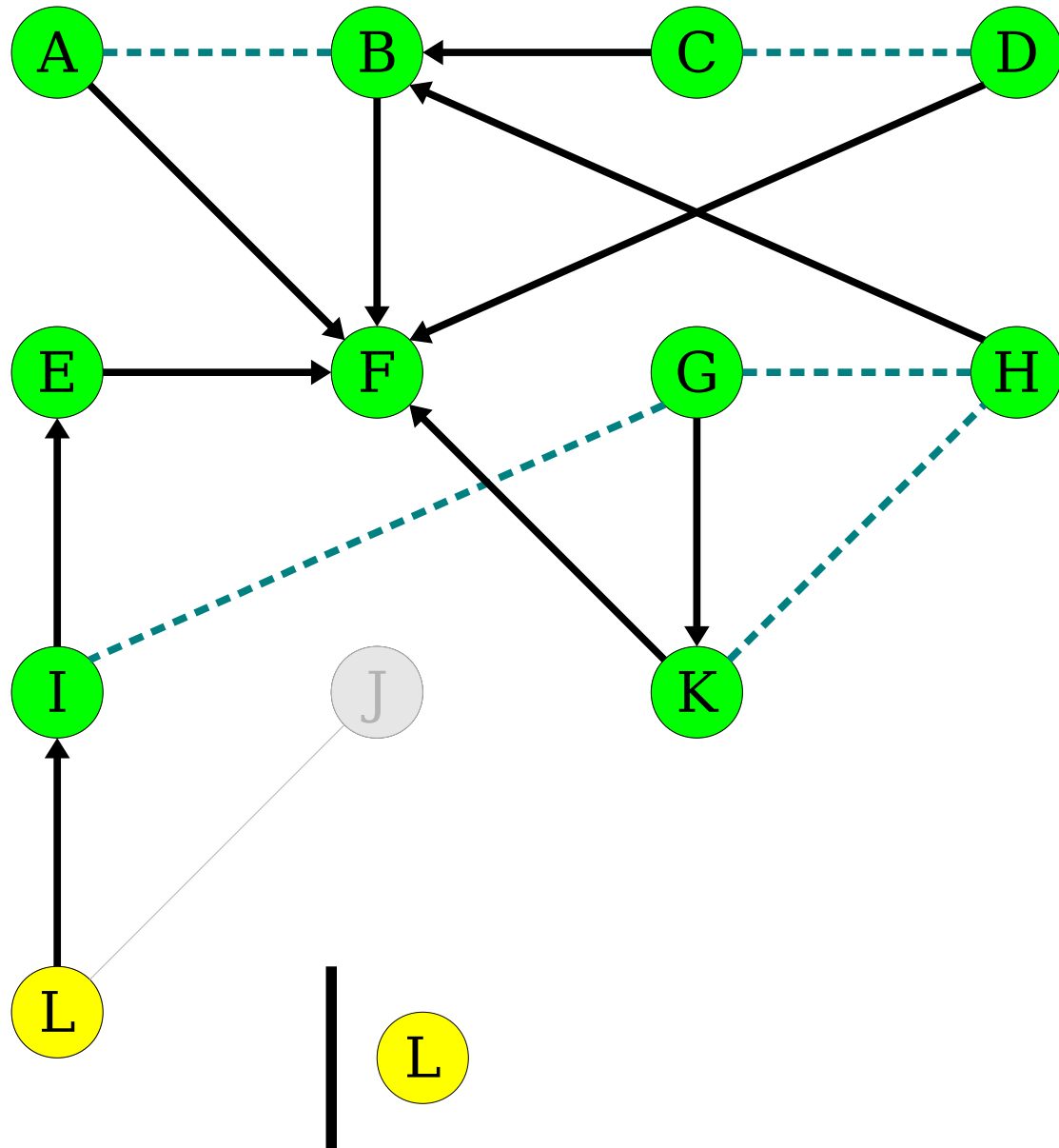
# Breadth-First Search



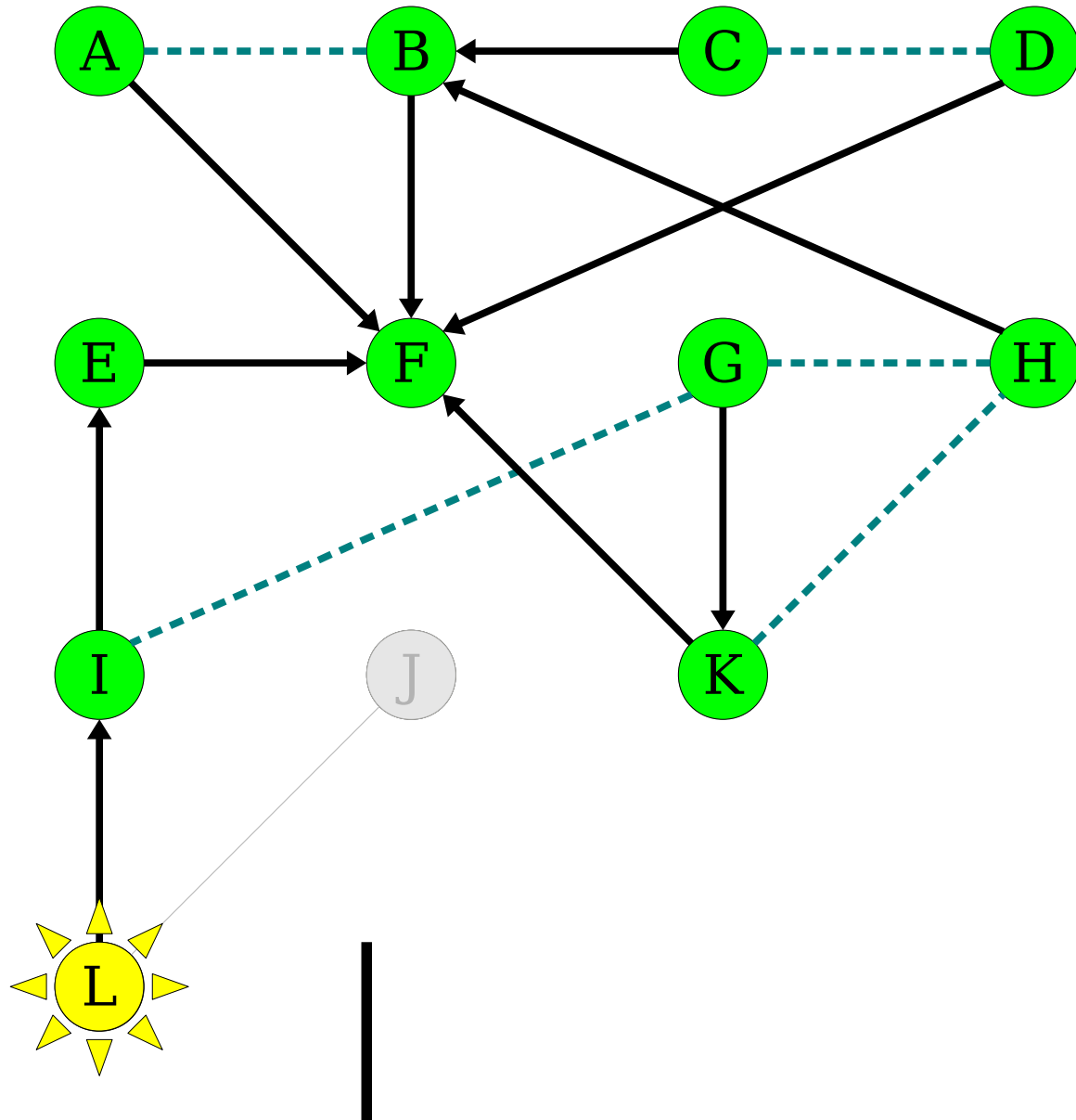
# Breadth-First Search



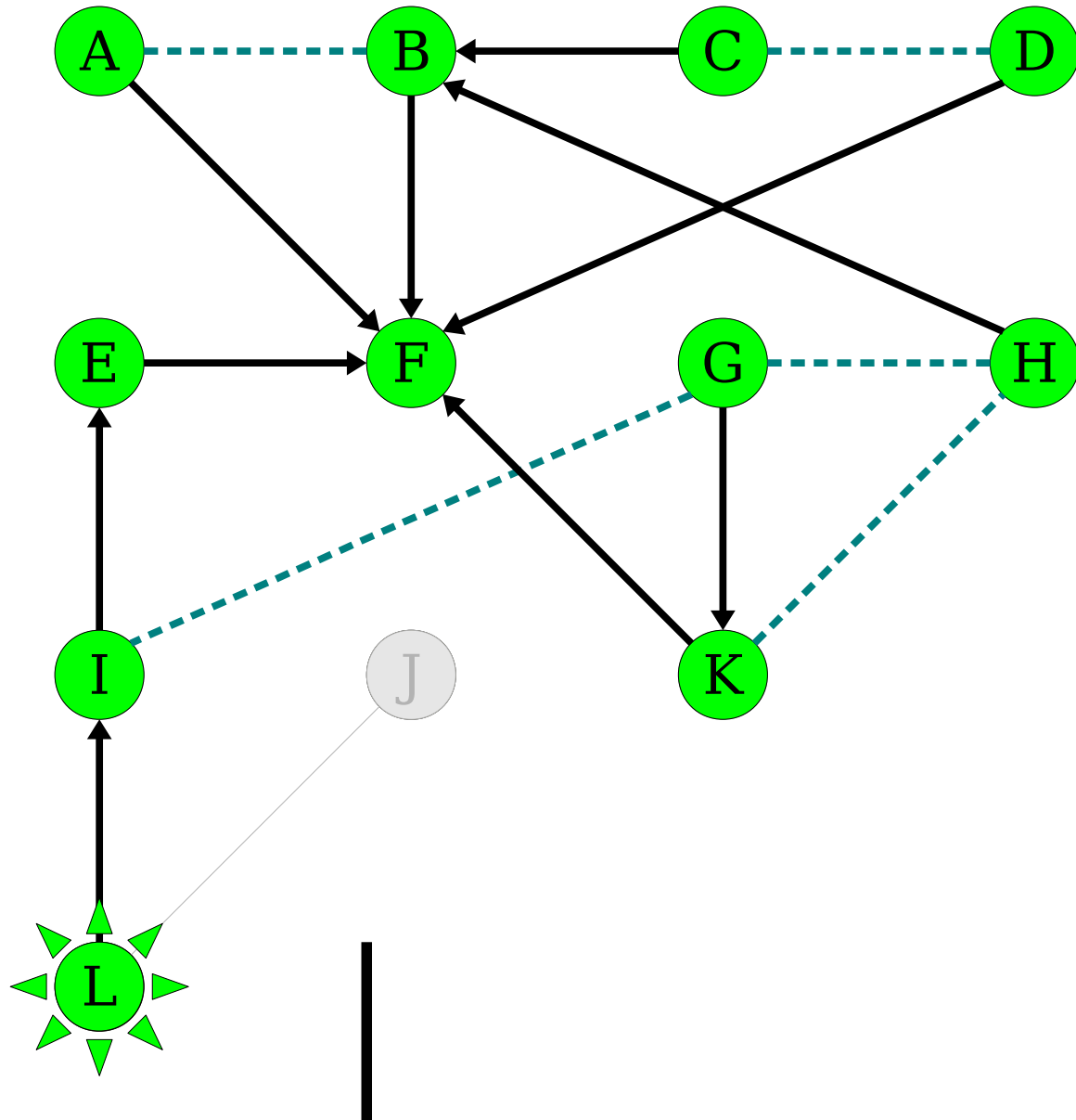
# Breadth-First Search



# Breadth-First Search

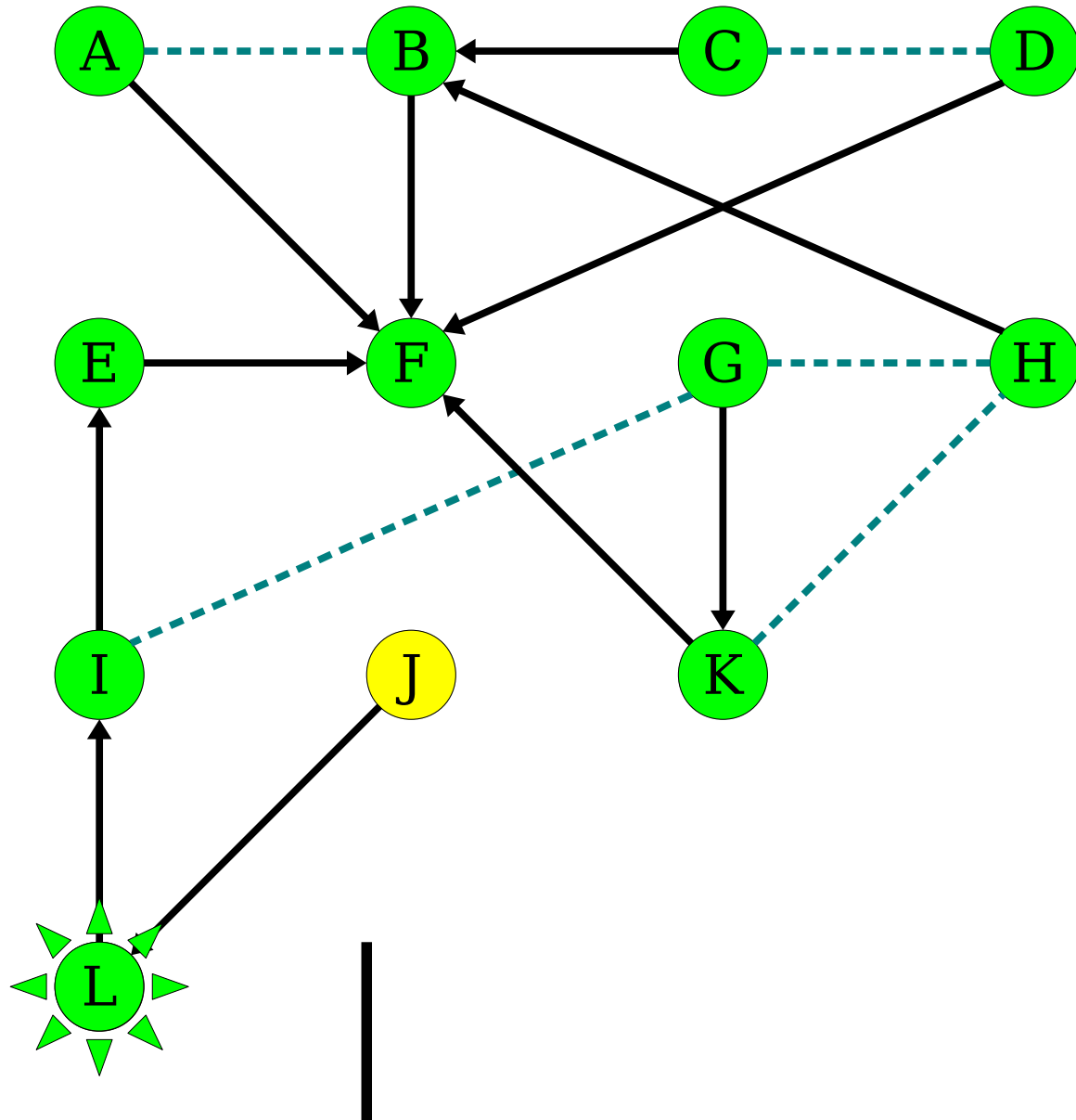


# Breadth-First Search

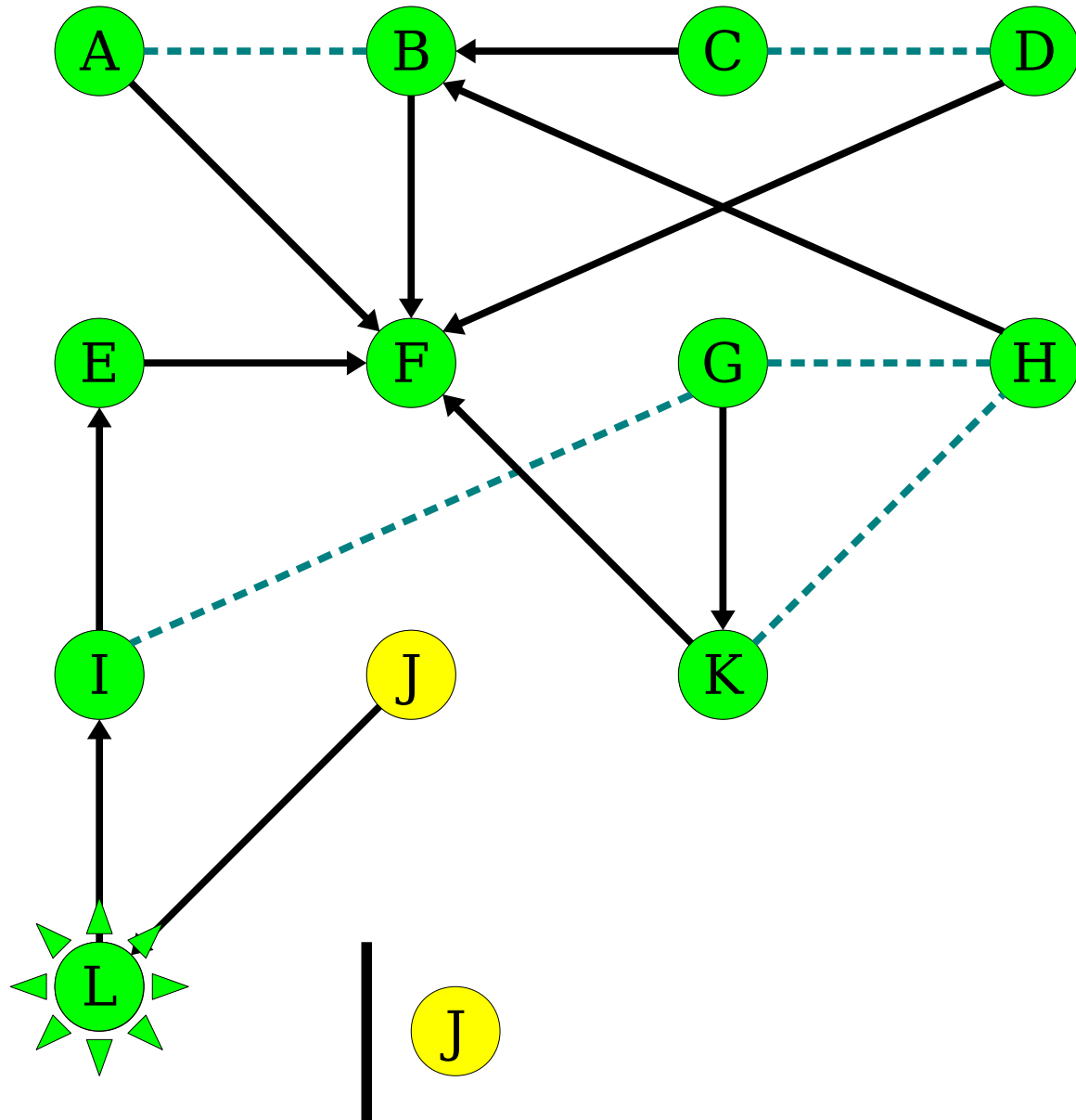




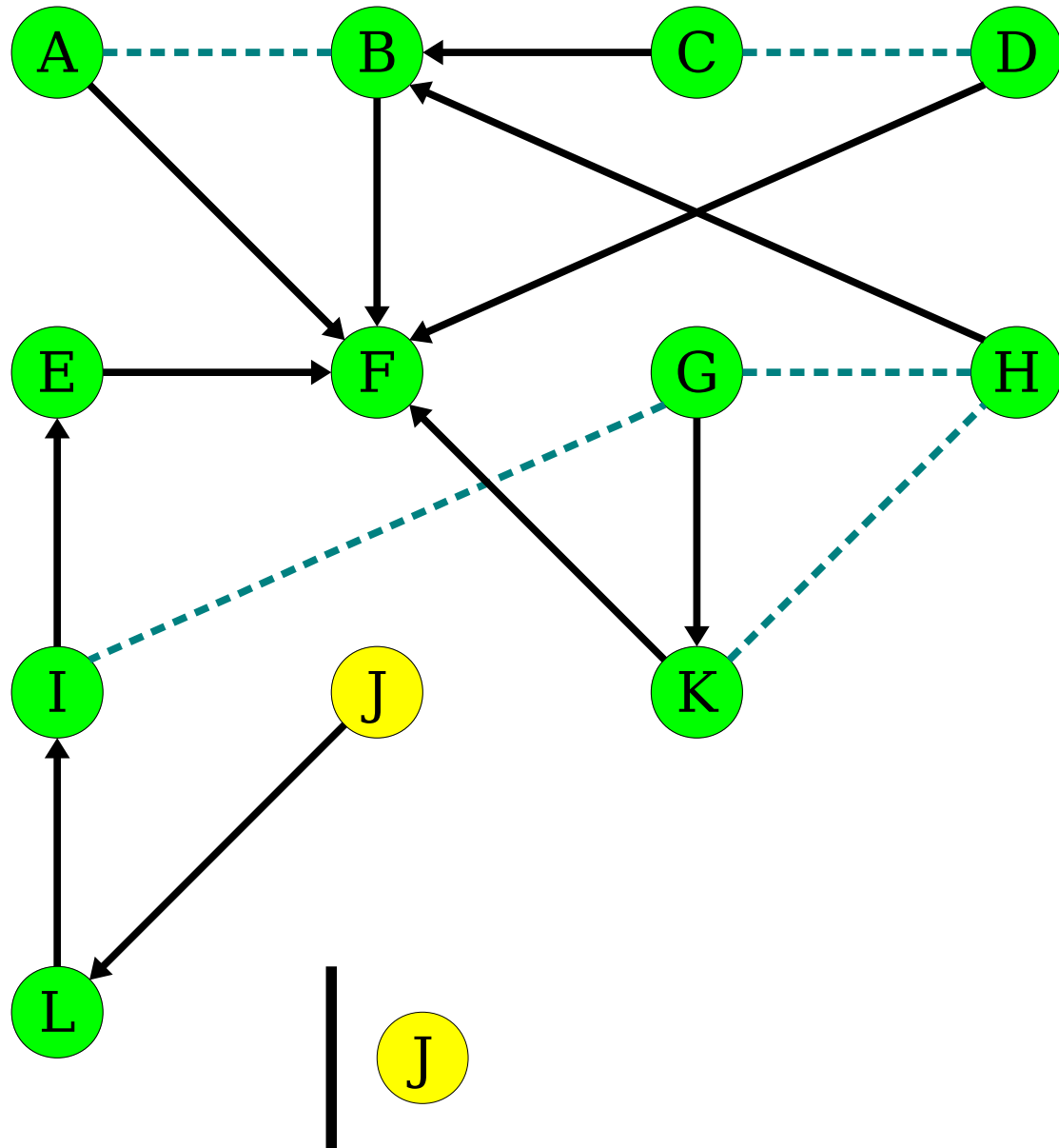
# Breadth-First Search



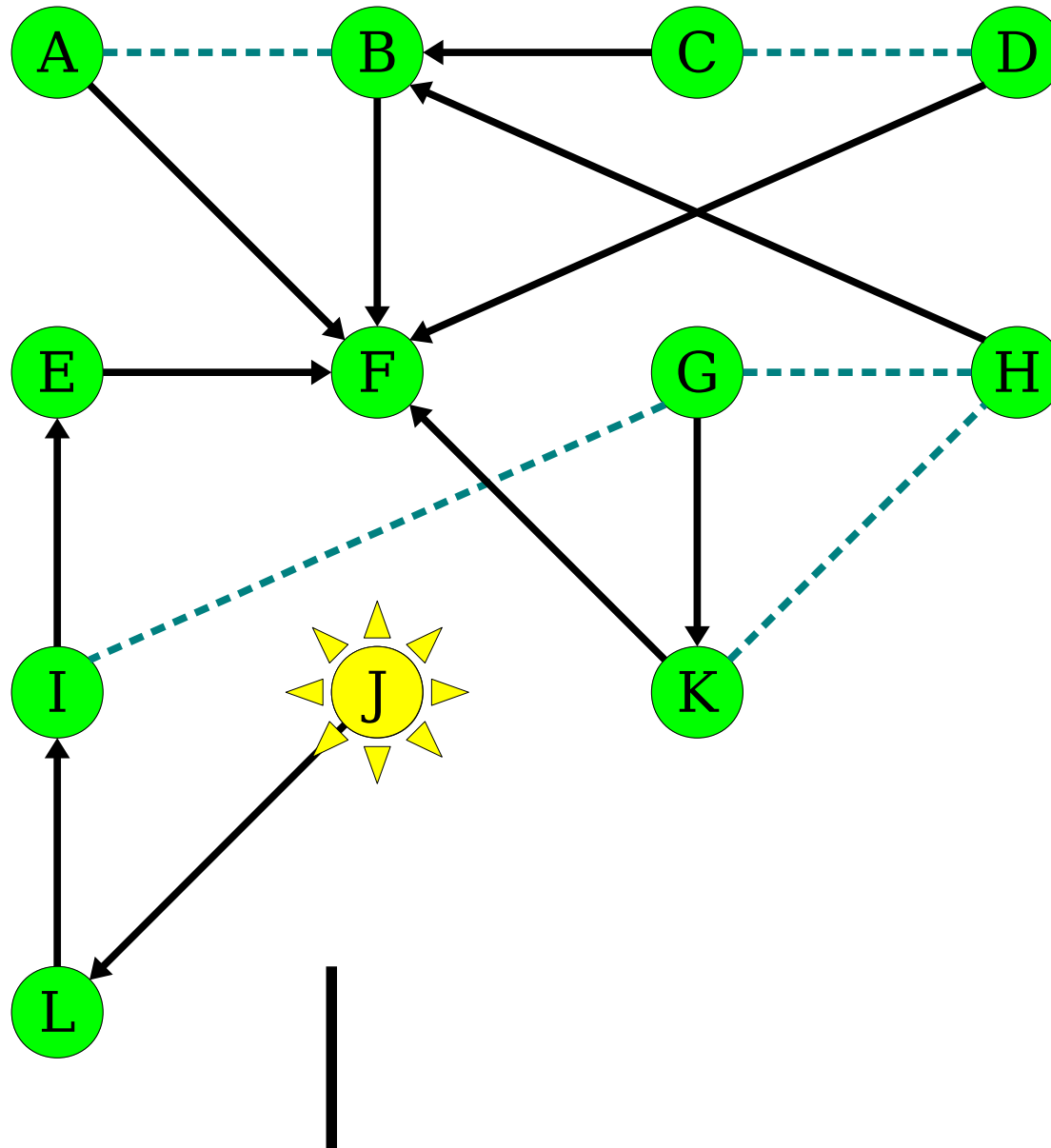
# Breadth-First Search



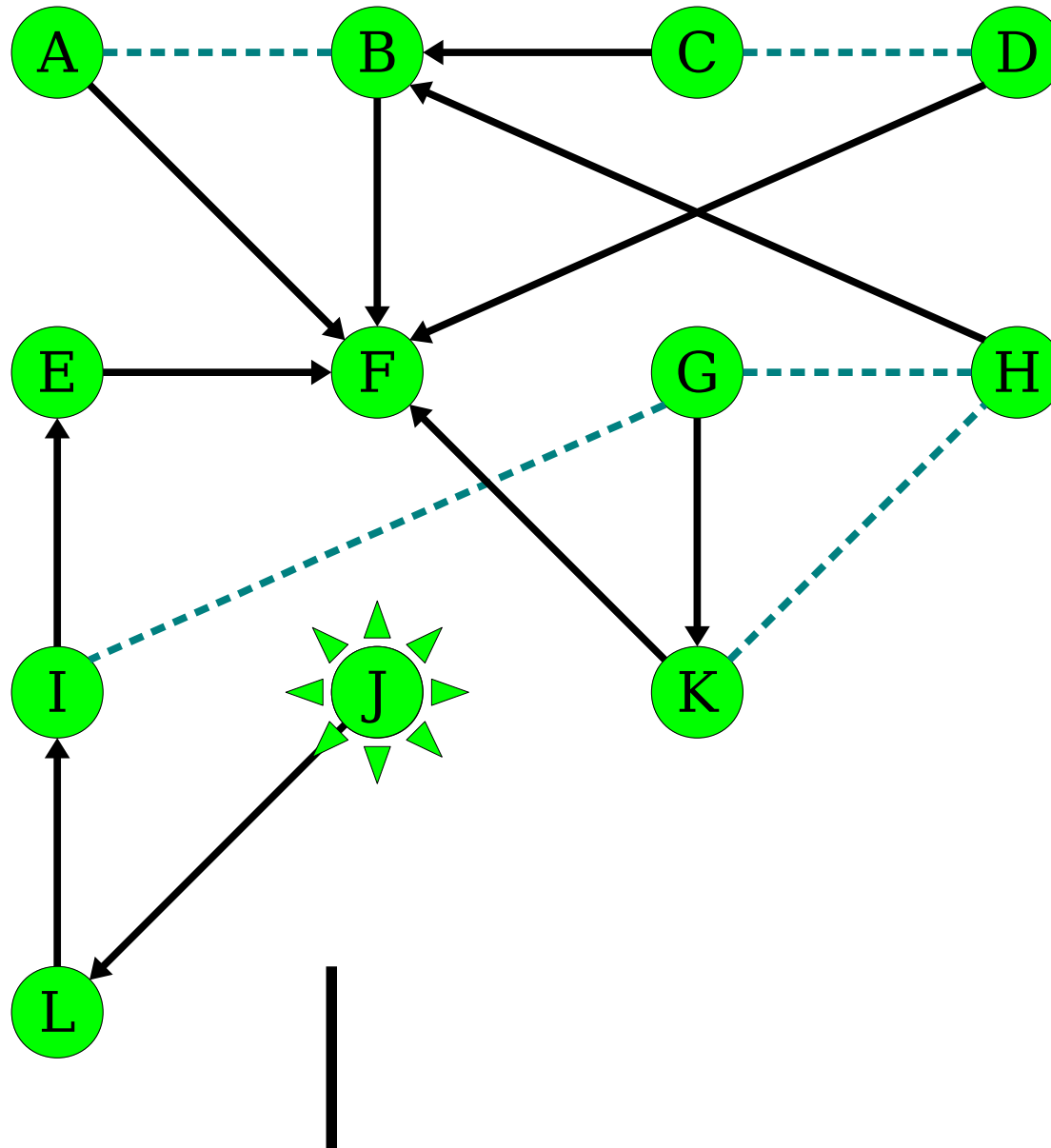
# Breadth-First Search



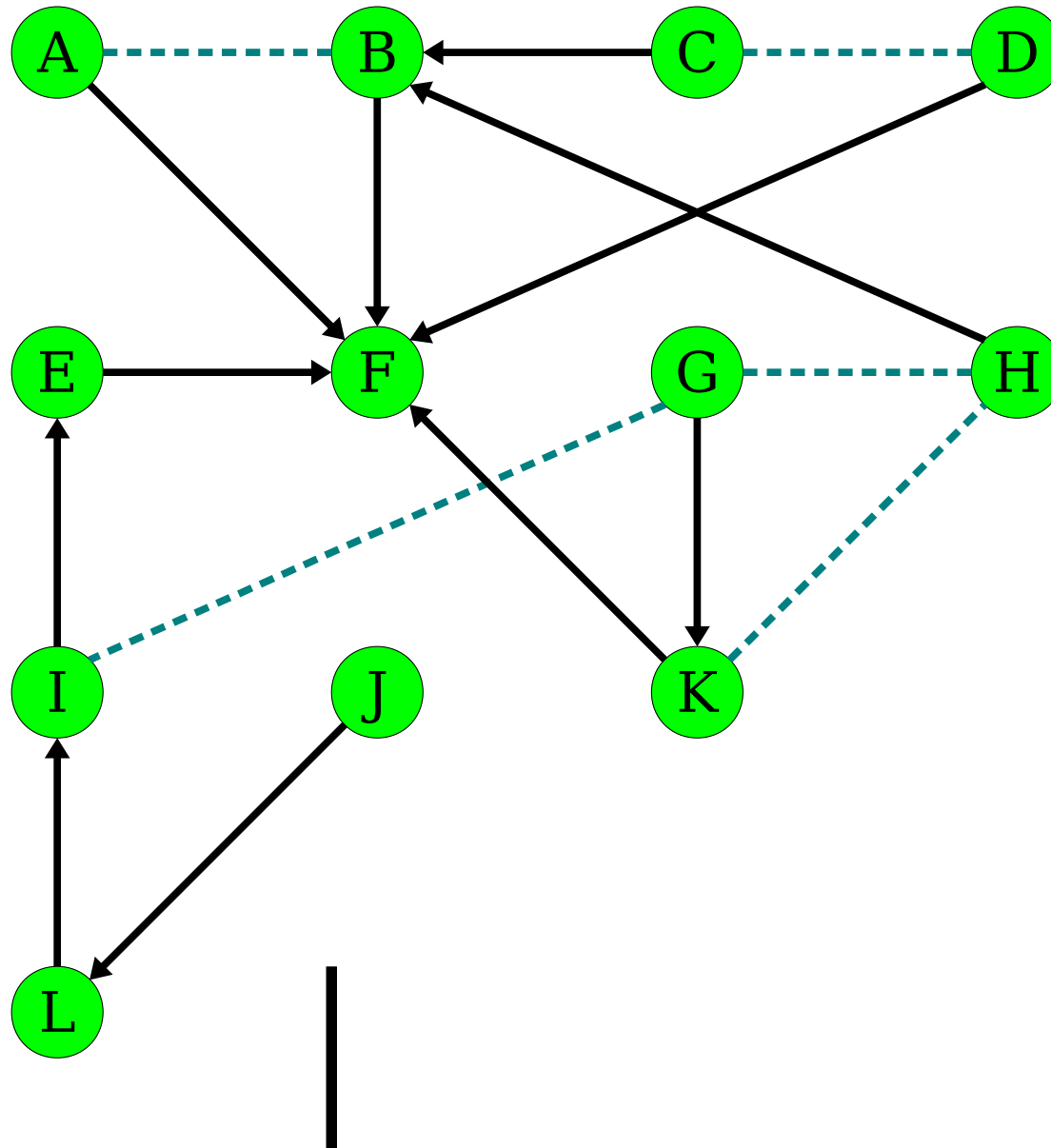
# Breadth-First Search



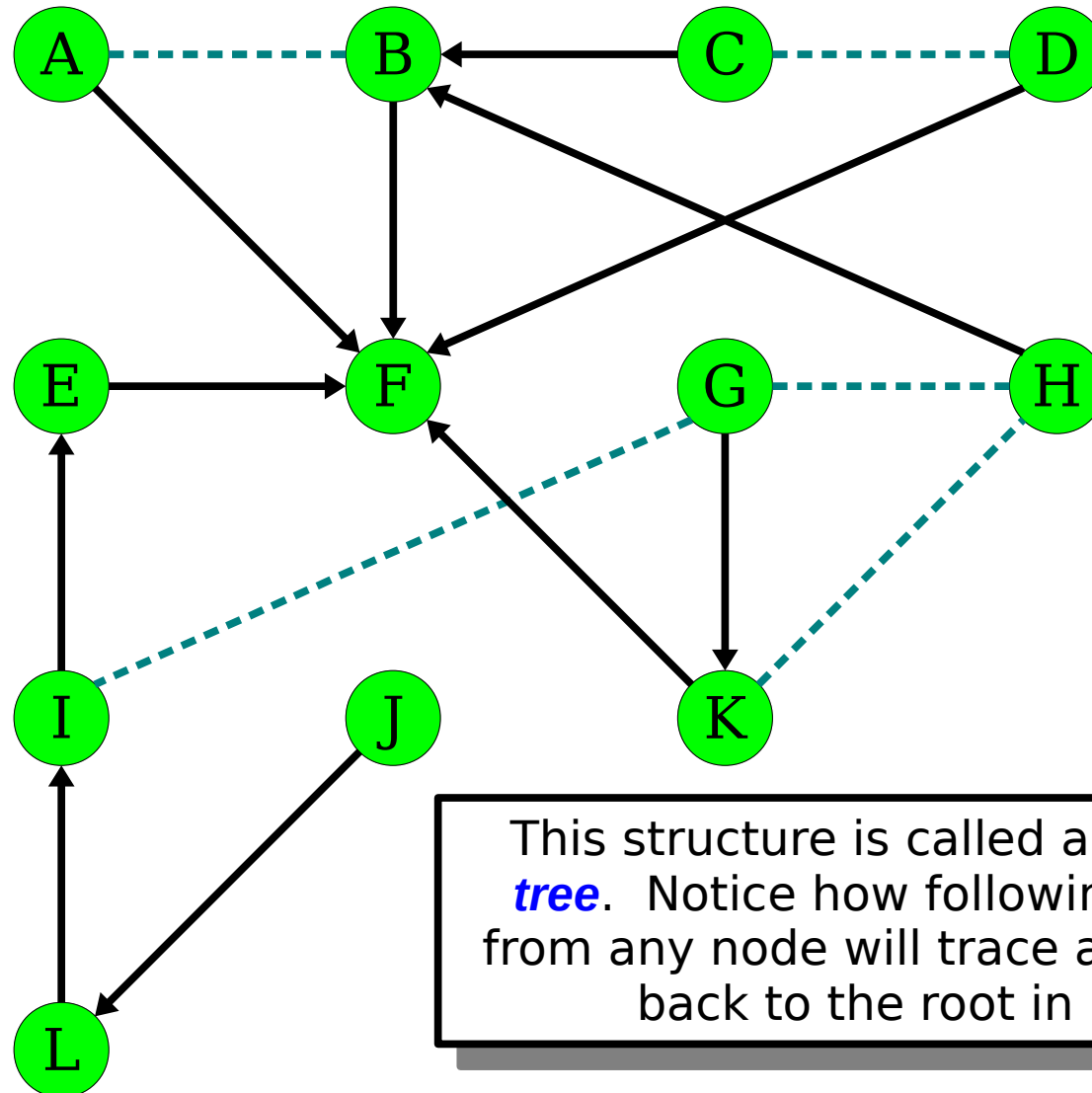
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search

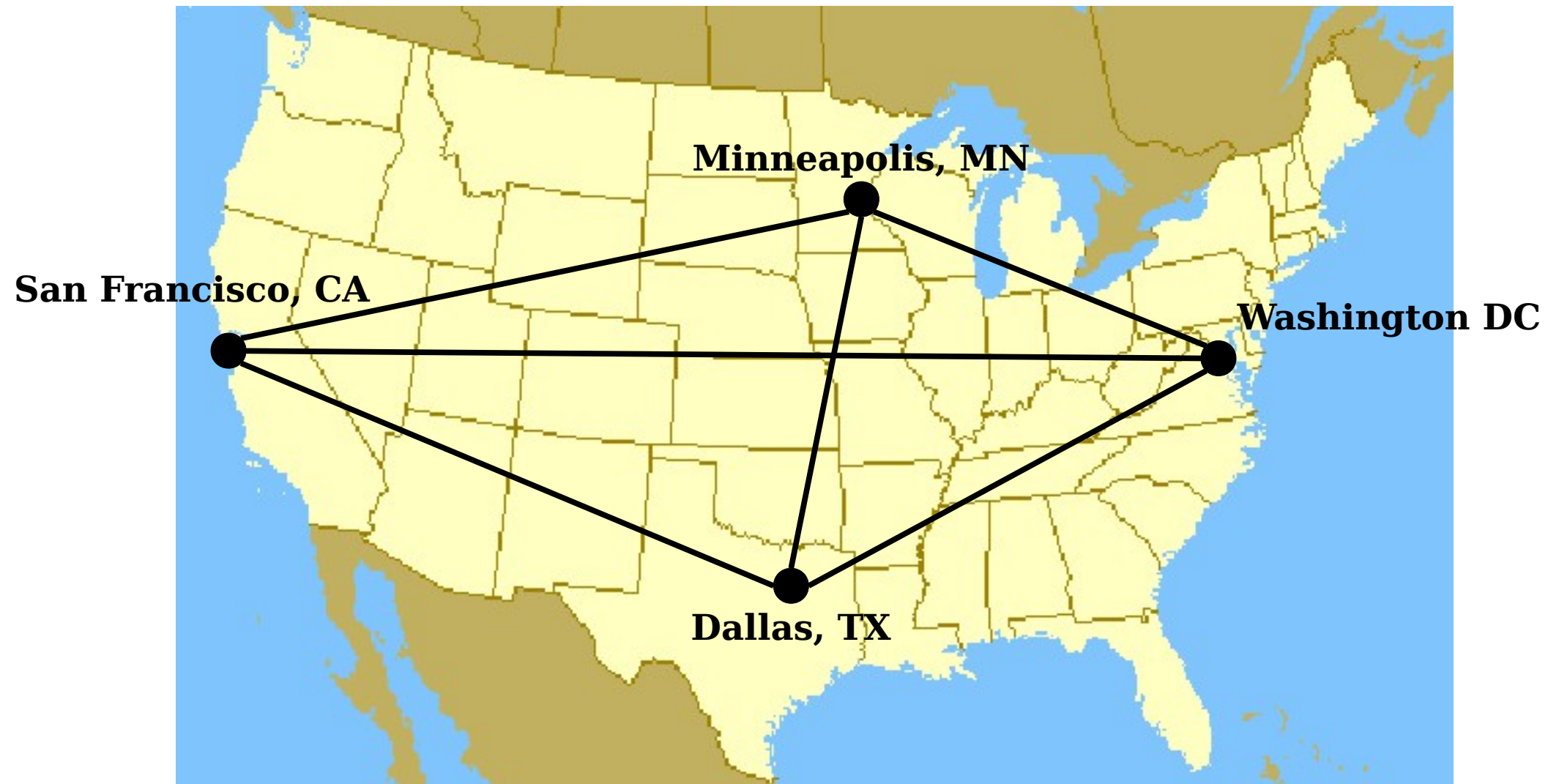


This structure is called a *shortest-path tree*. Notice how following the arrows from any node will trace a shortest path back to the root in reverse.

# Extending Graphs



# Flights



# Flights



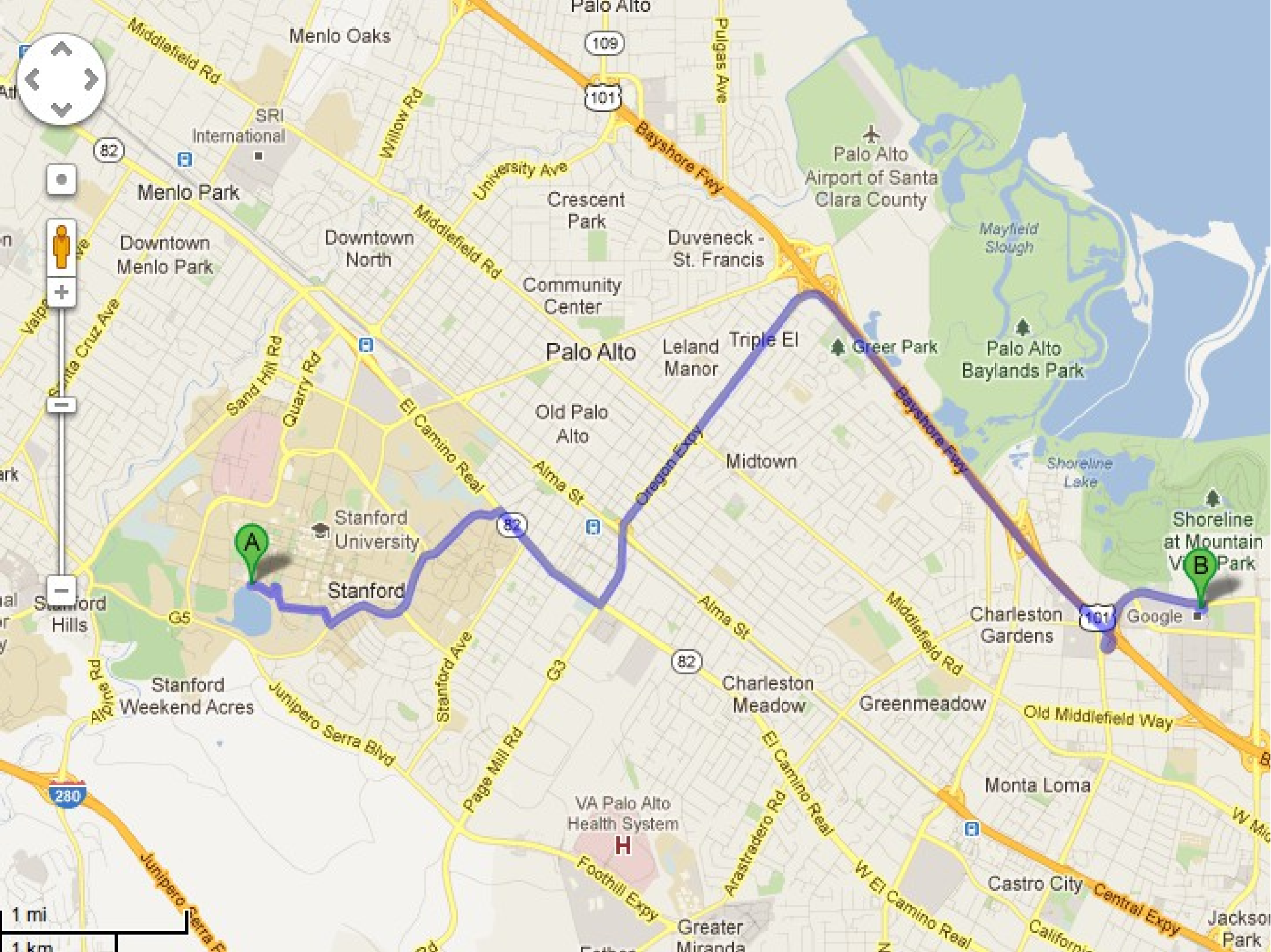
# Pathfinding on Weighted Graphs

# BFS and Pathfinding

- **Breadth-first search** is good for determining the shortest path from  $s$  to  $t$ .
  - Uses a queue.
  - If edges are unweighted, then return optimal path
- What happens if the edges are **weighted**?
  - e.g. distance, cost of airfare

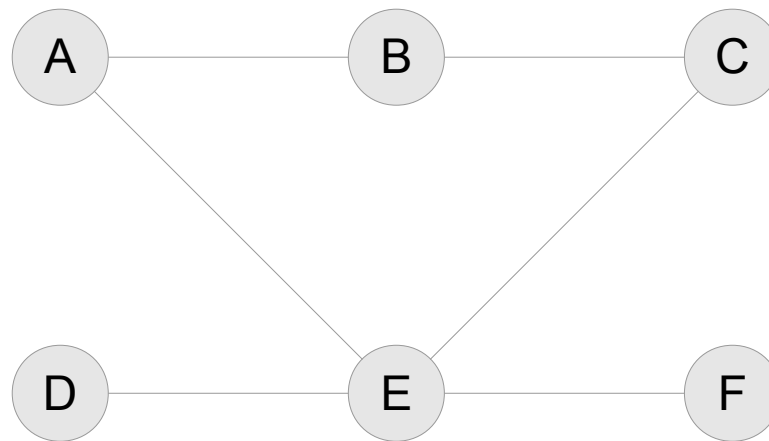
# Shortest Paths

- You are given a graph where each edge has a **nonnegative weight**.
- Given a starting node  $s$ , find the shortest path (in terms of total weight) from  $s$  to each other node  $t$ .



# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



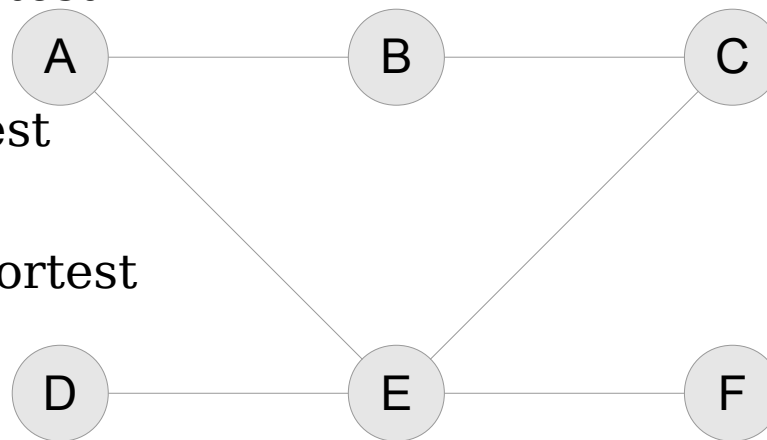
Red: No idea of shortest path



Green: Found shortest path



Yellow: Guess for shortest path





# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



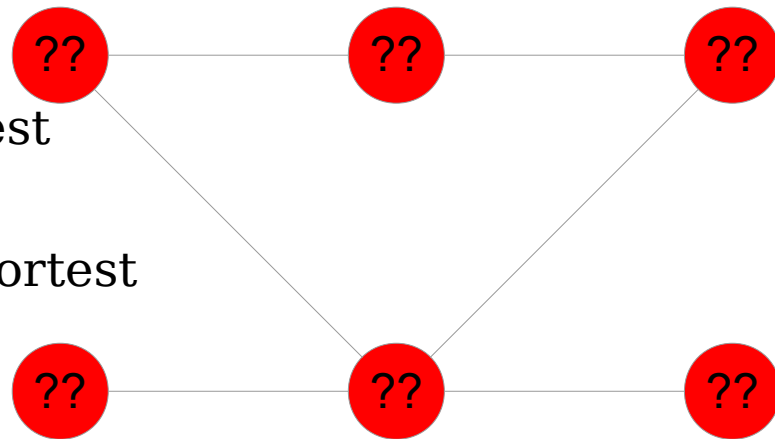
Red: No idea of shortest path



Green: Found shortest path

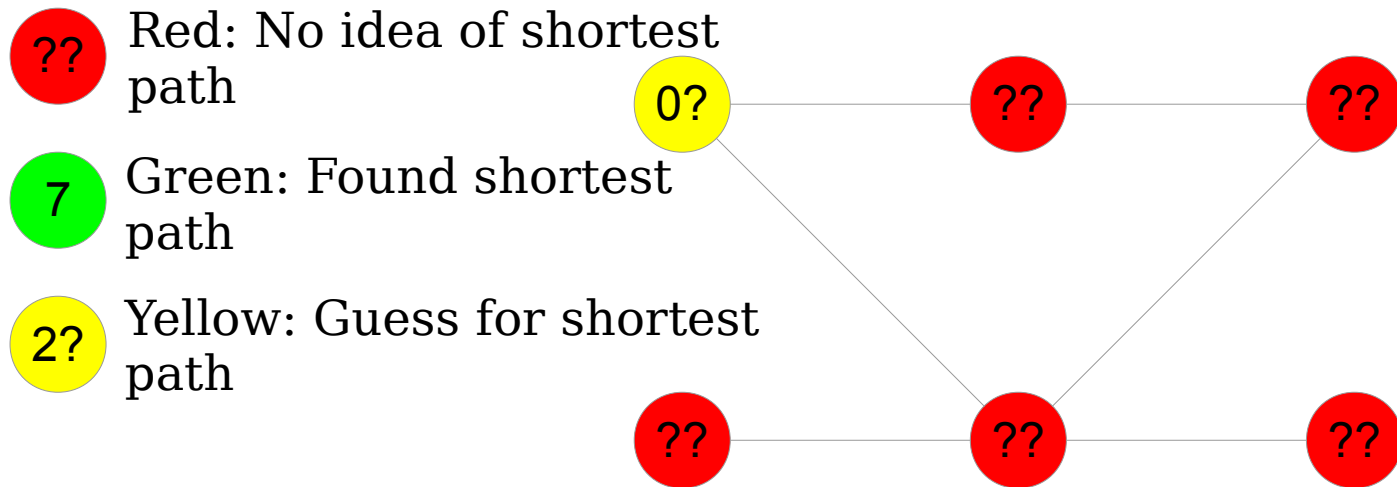


Yellow: Guess for shortest path



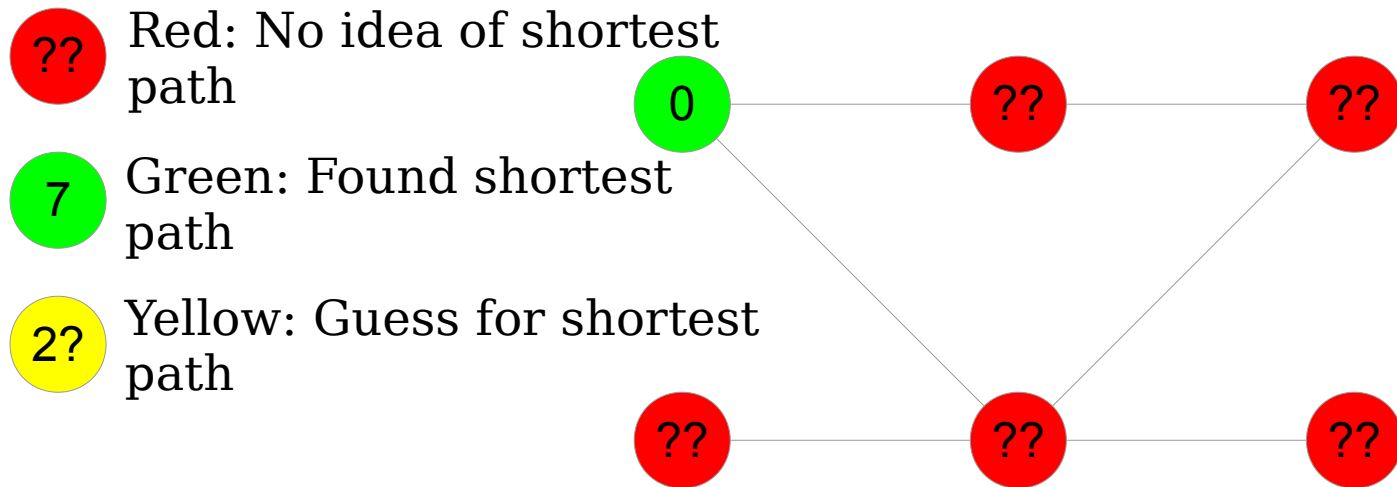
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



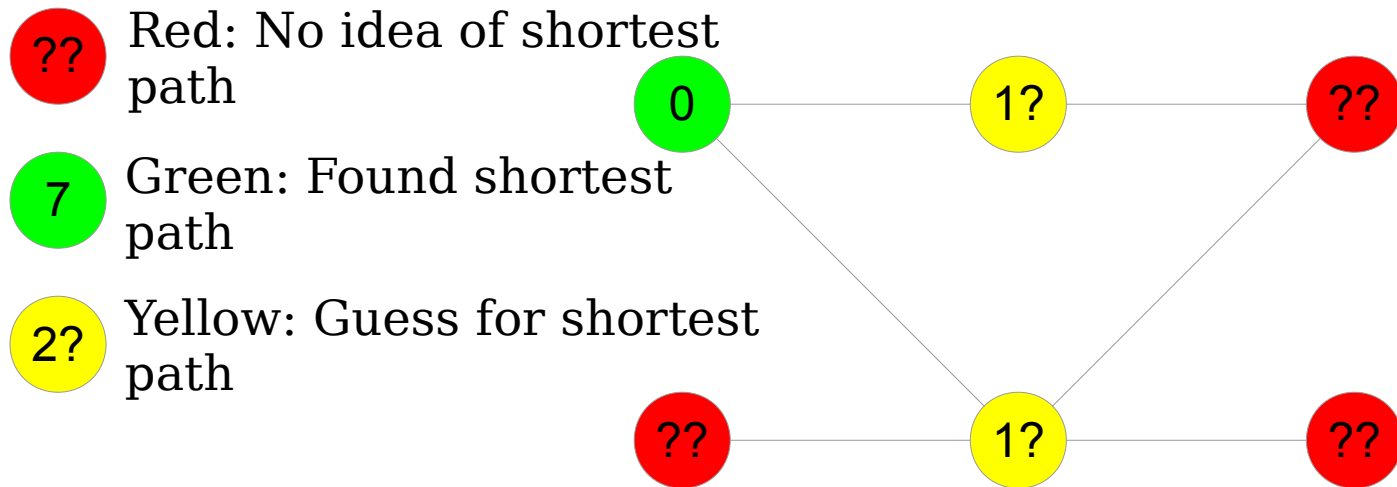
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



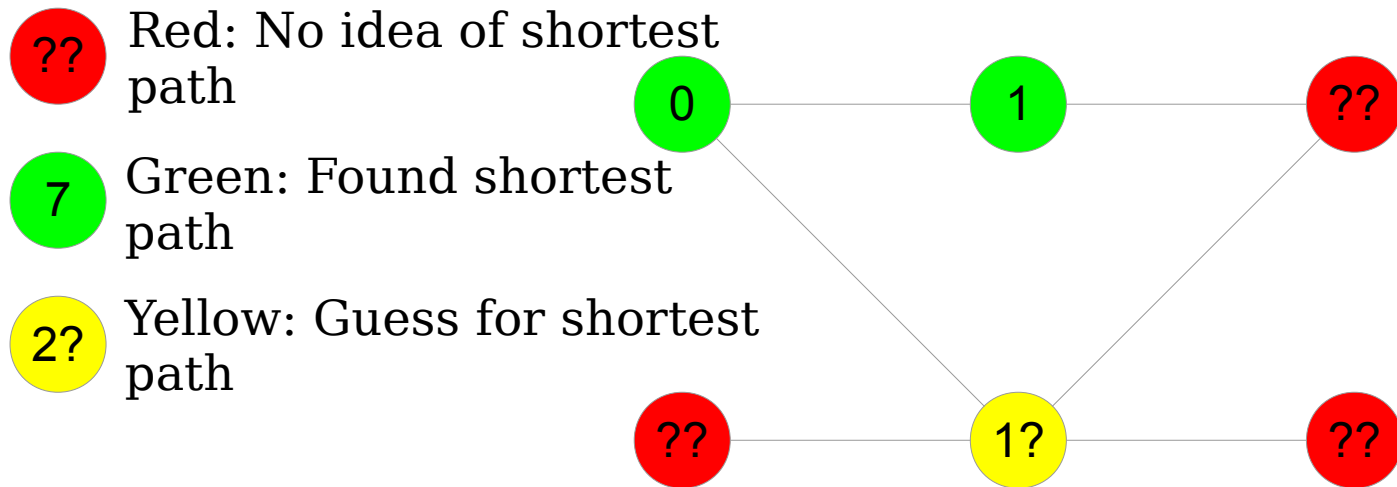
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



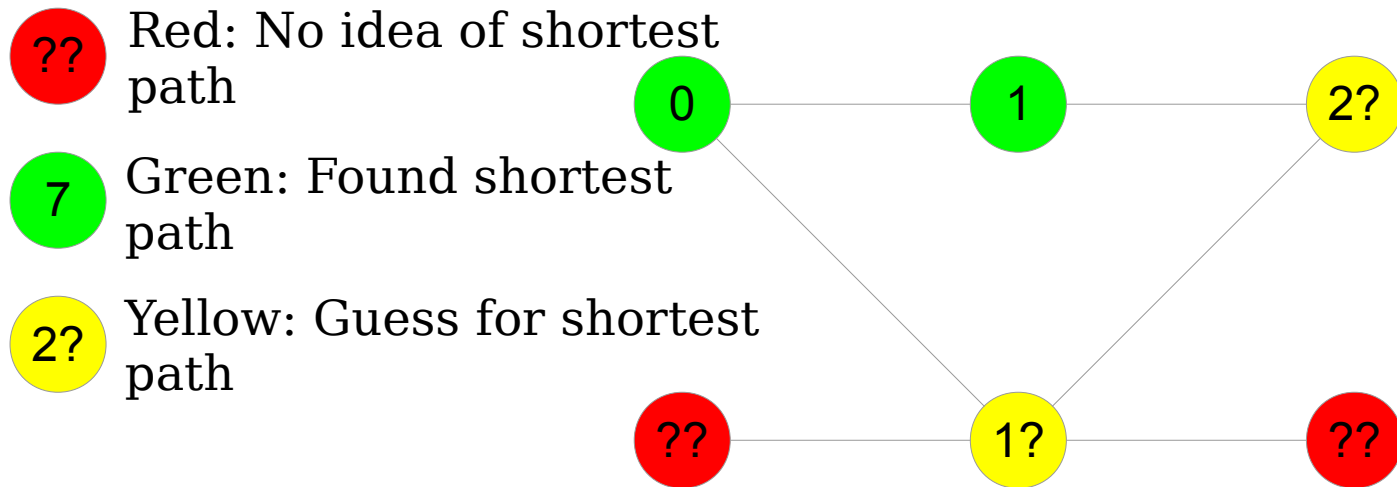
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



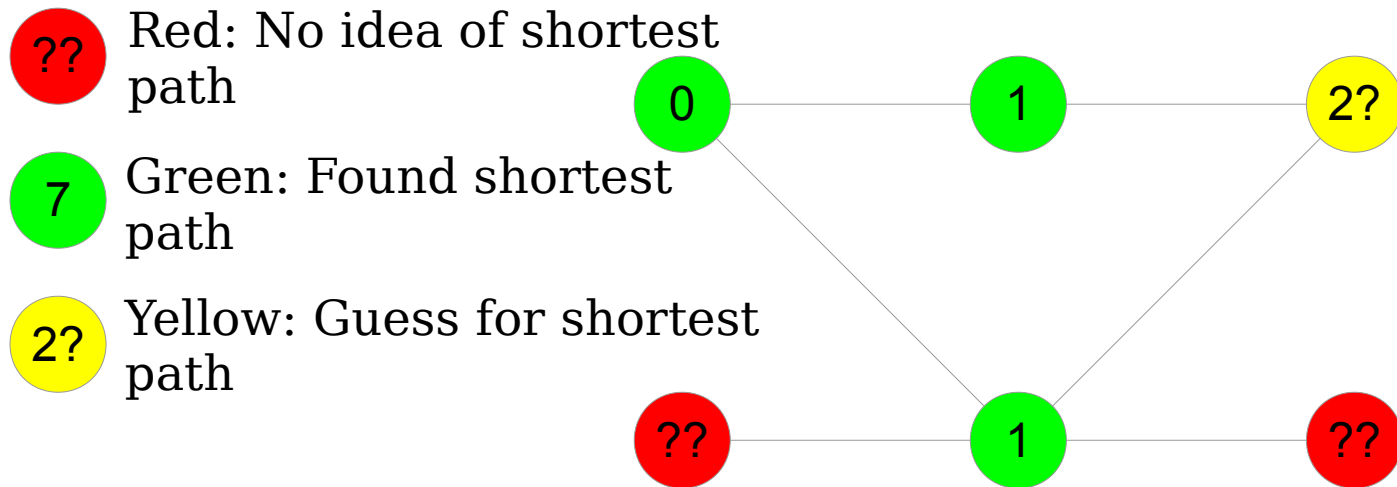
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



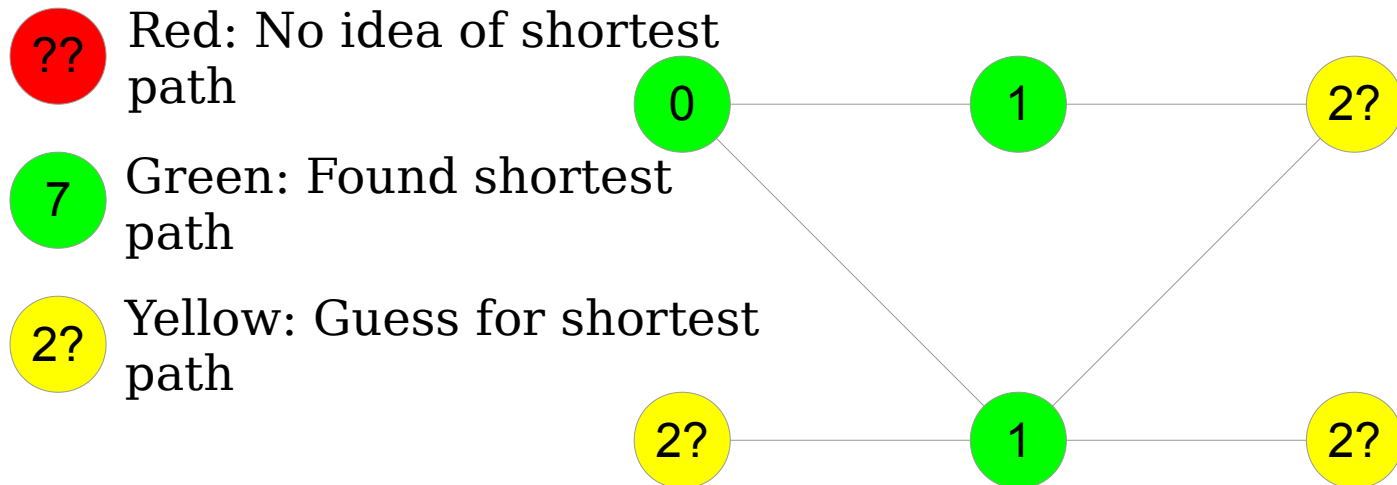
# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start





# Shortest Paths

- Cool fact: We can generalize BFS to work on graphs with weighted edges
  - BFS returns an optimal path because it visits nodes in order of distance from the start



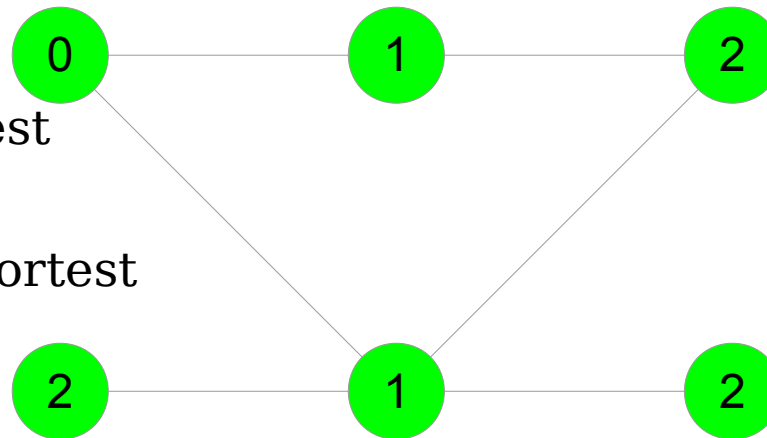
Red: No idea of shortest path



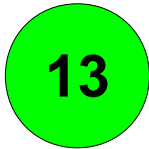

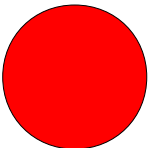
Green: Found shortest path

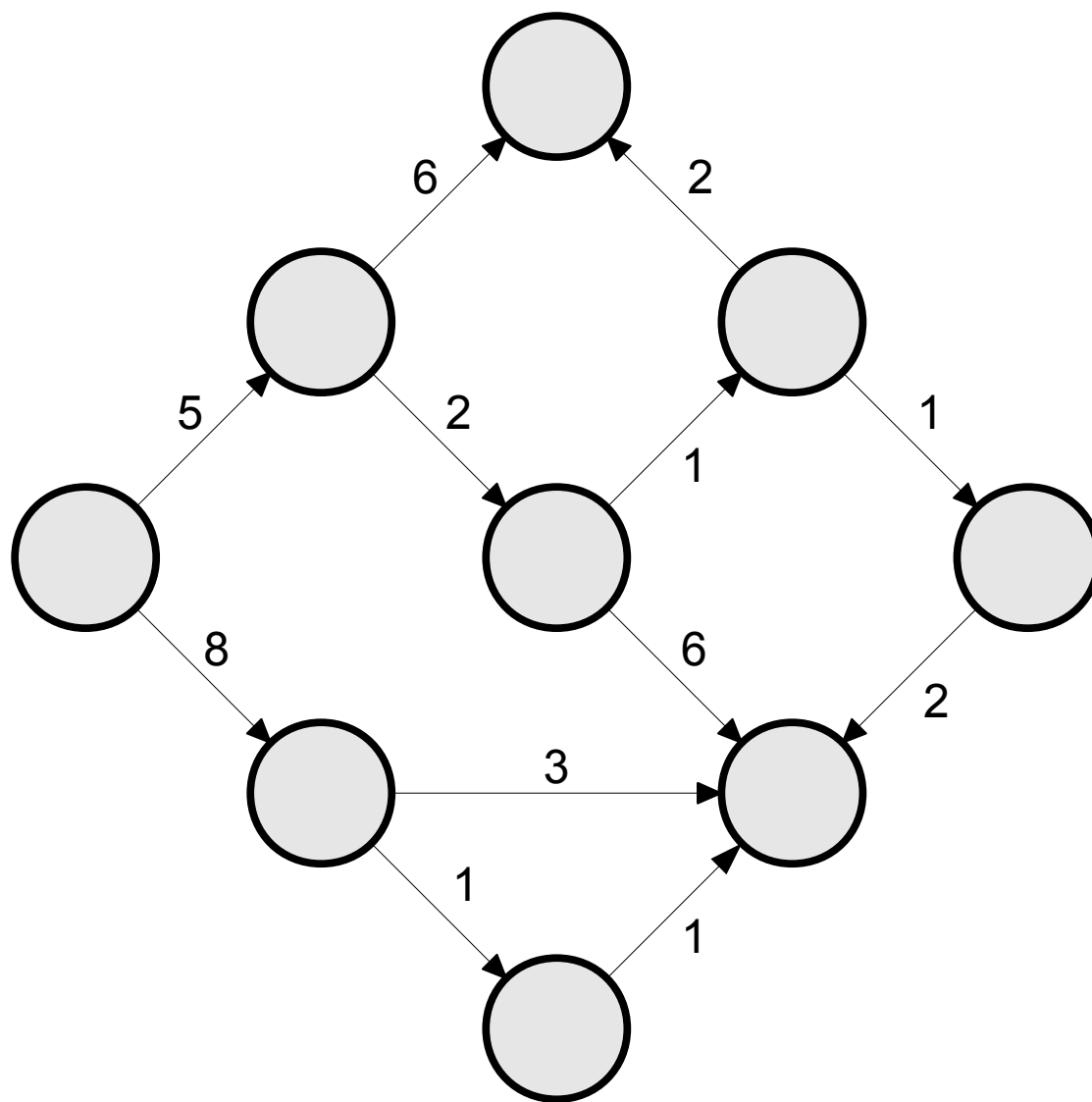


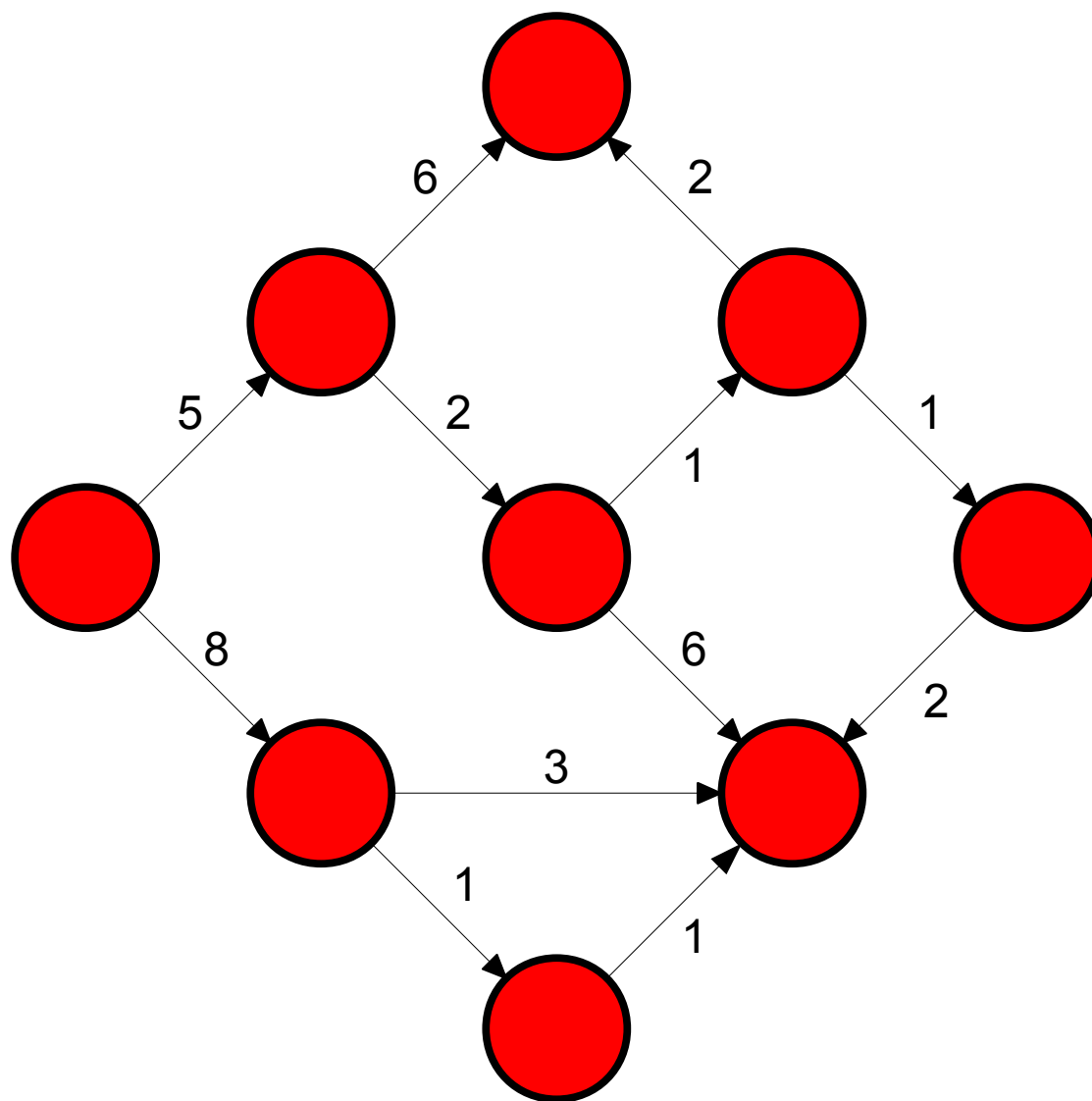
Yellow: Guess for shortest path

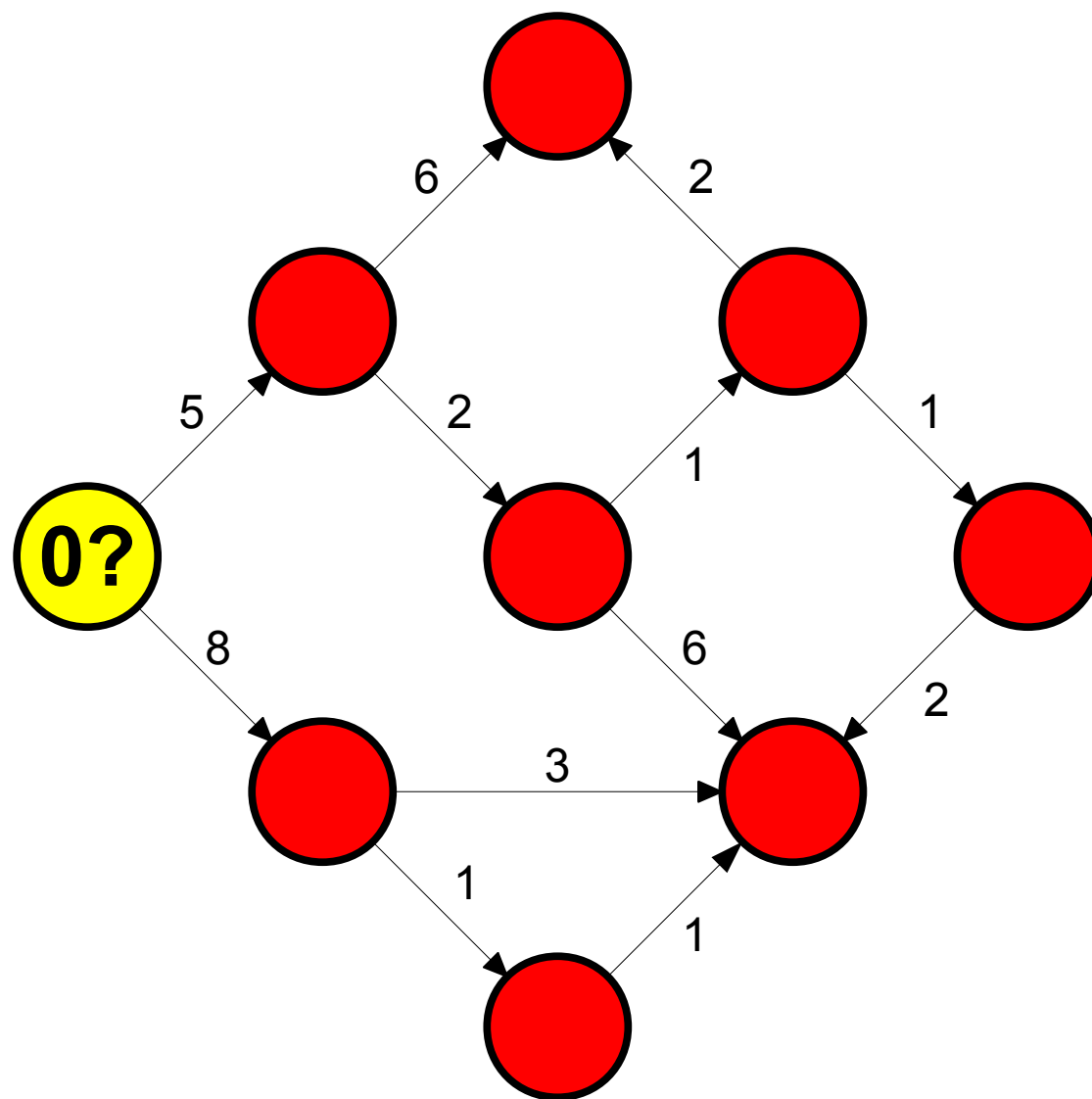


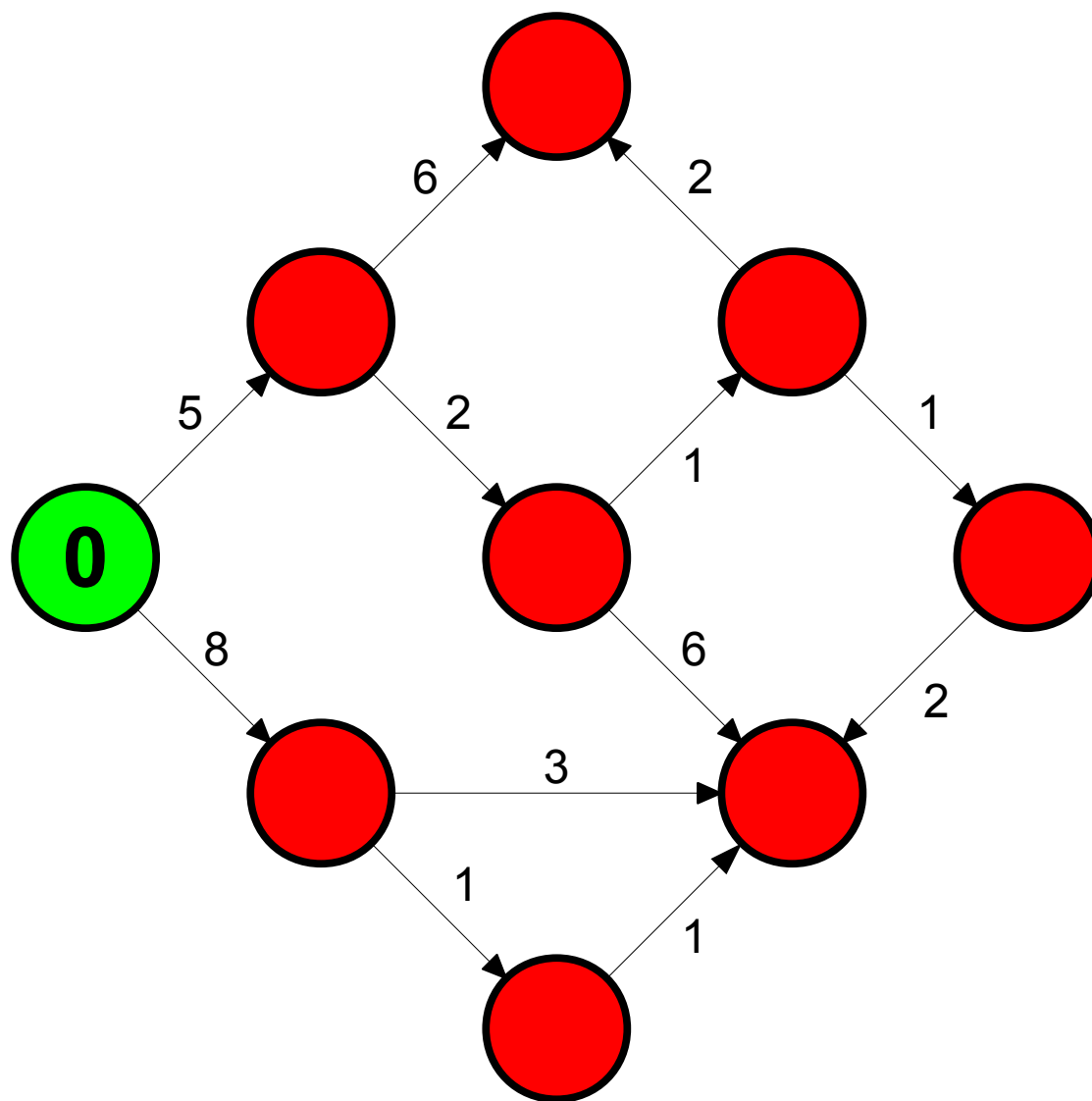
# Our Approach

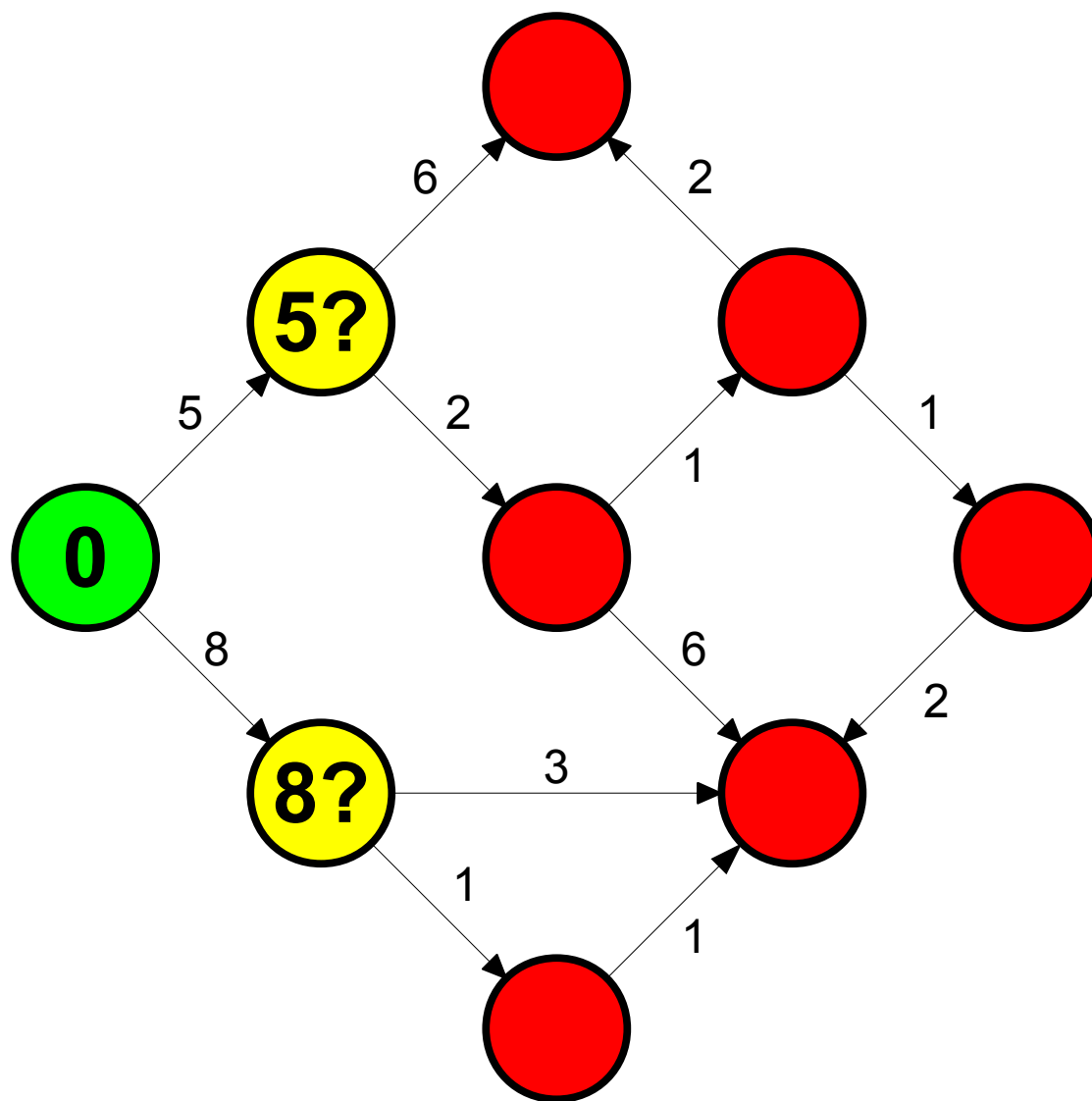
- Split nodes into three groups:
  -  Green nodes, where we know the length of the shortest path,
  -  Yellow nodes, where we have a guess of the length of the shortest path, and
  -  Red nodes, where we have no idea what the path length is.
- Repeatedly remove the **lowest-cost** yellow node, make it green, and update all connected nodes.

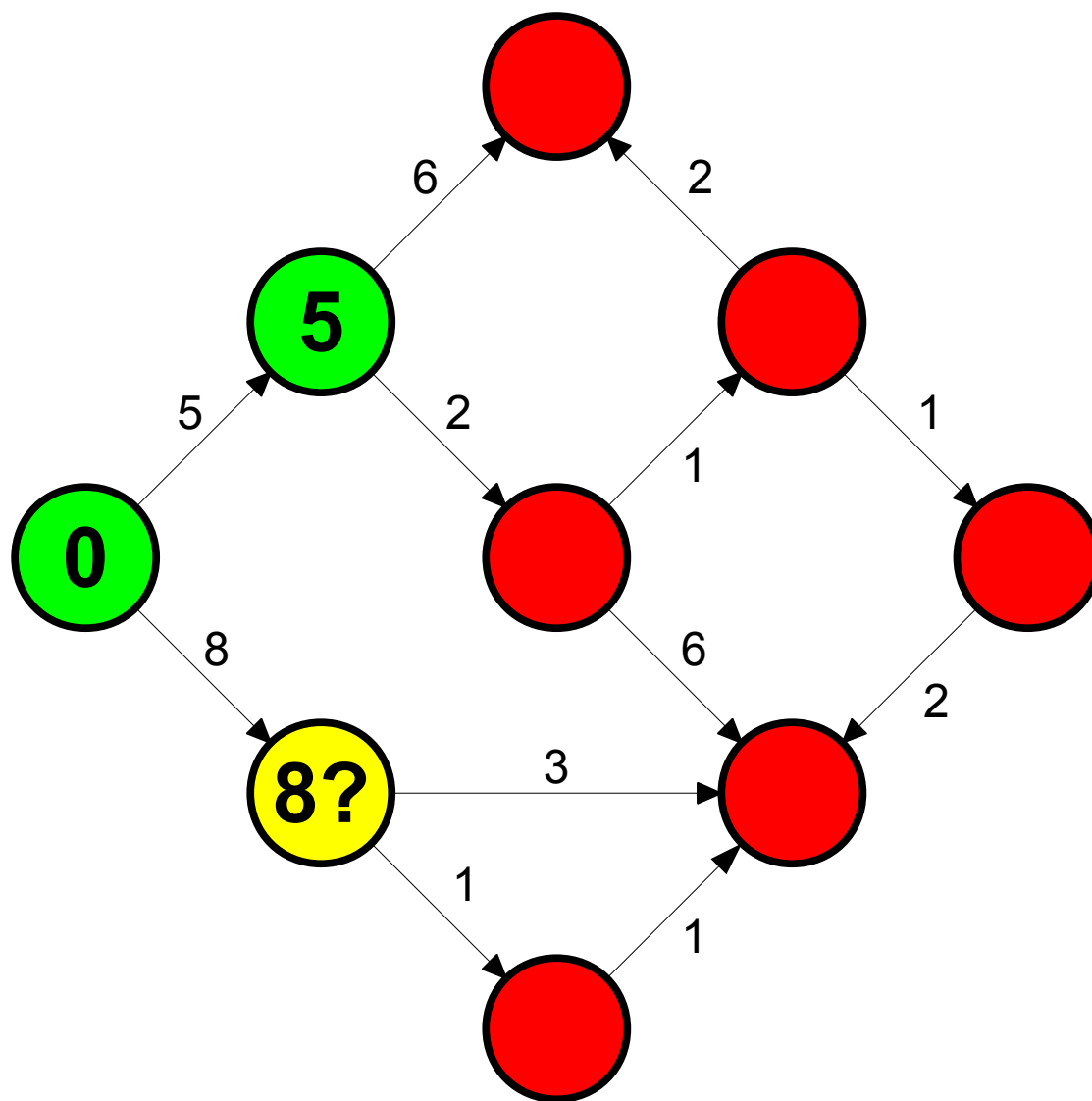




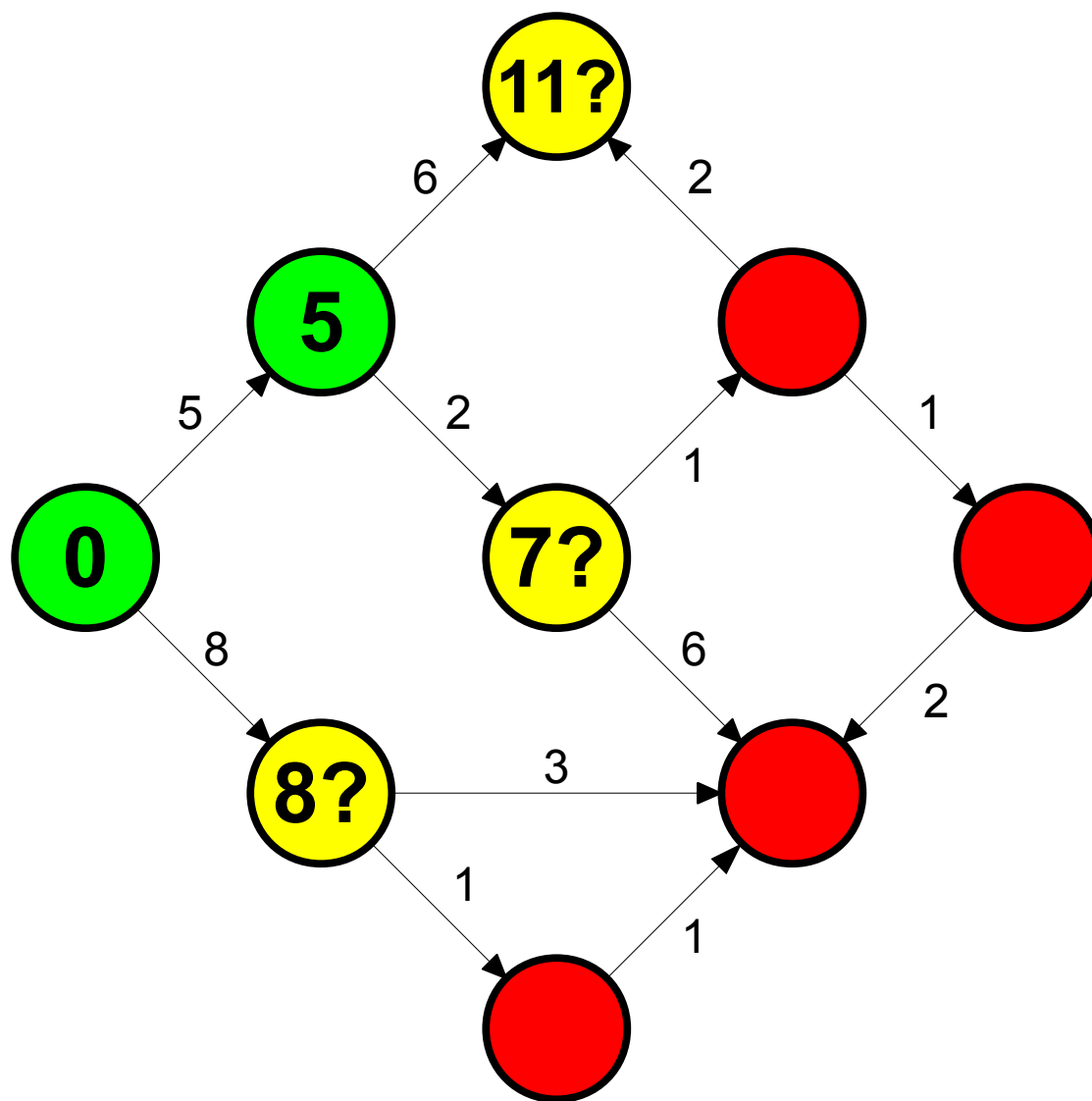


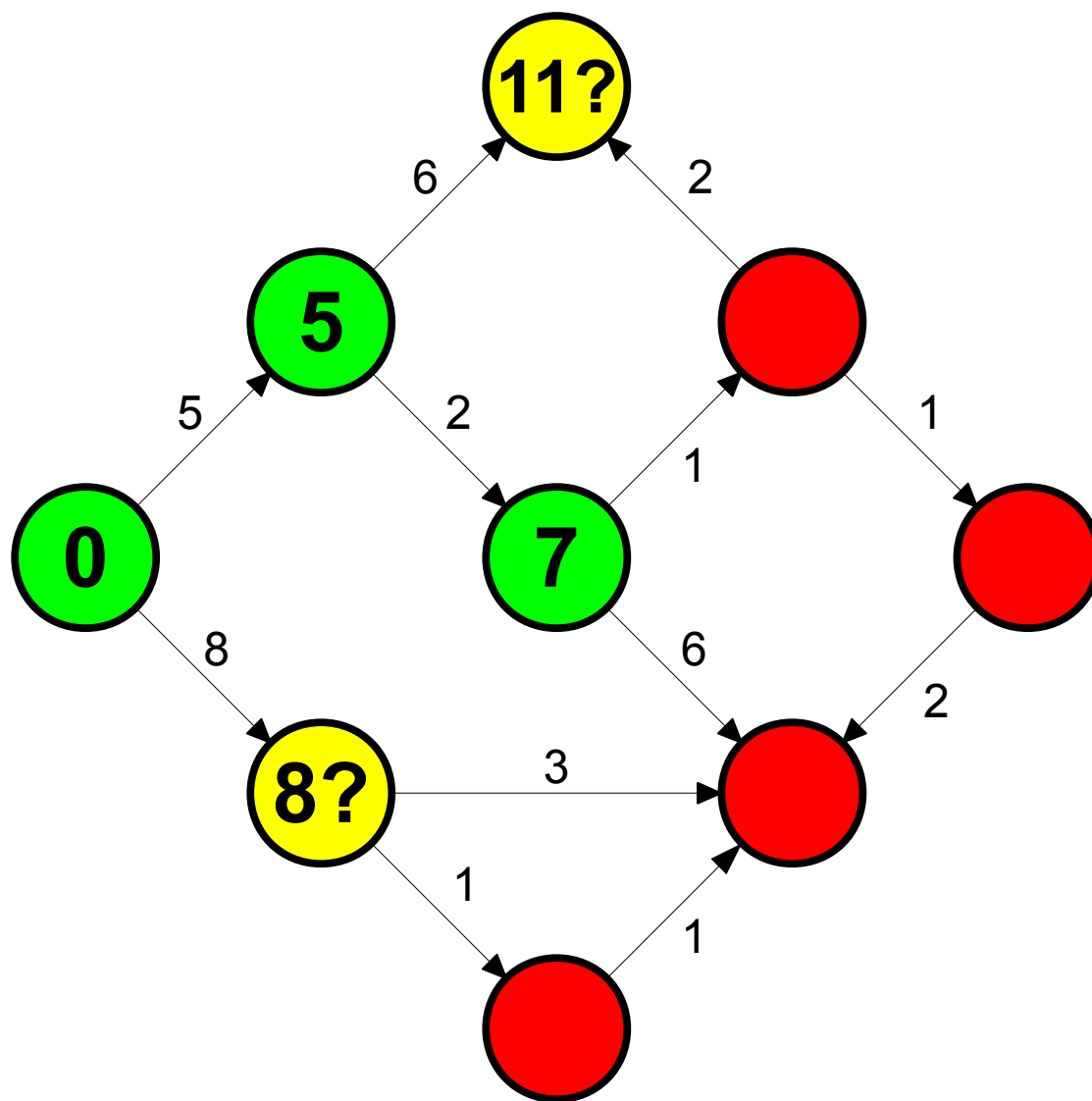


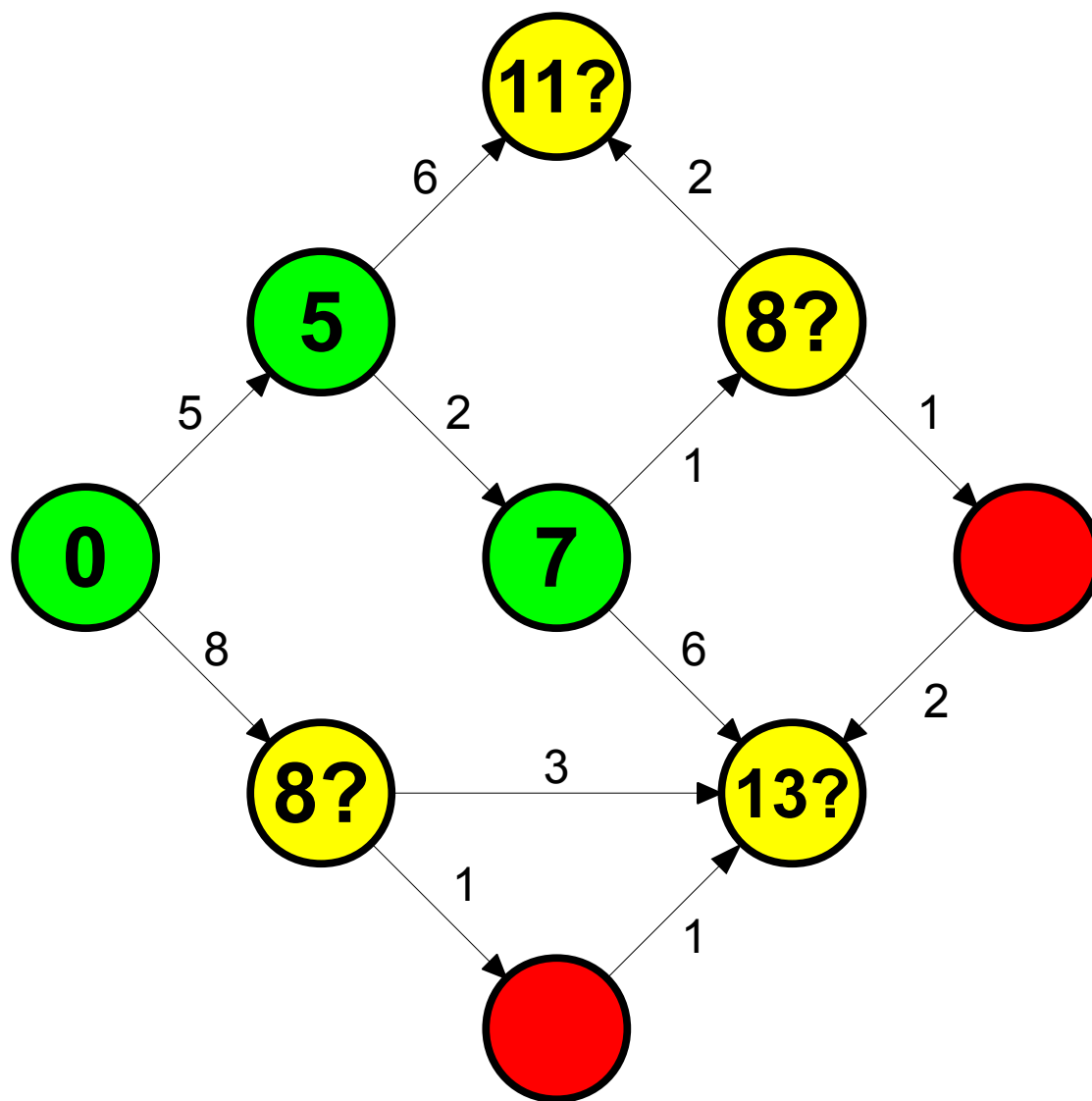


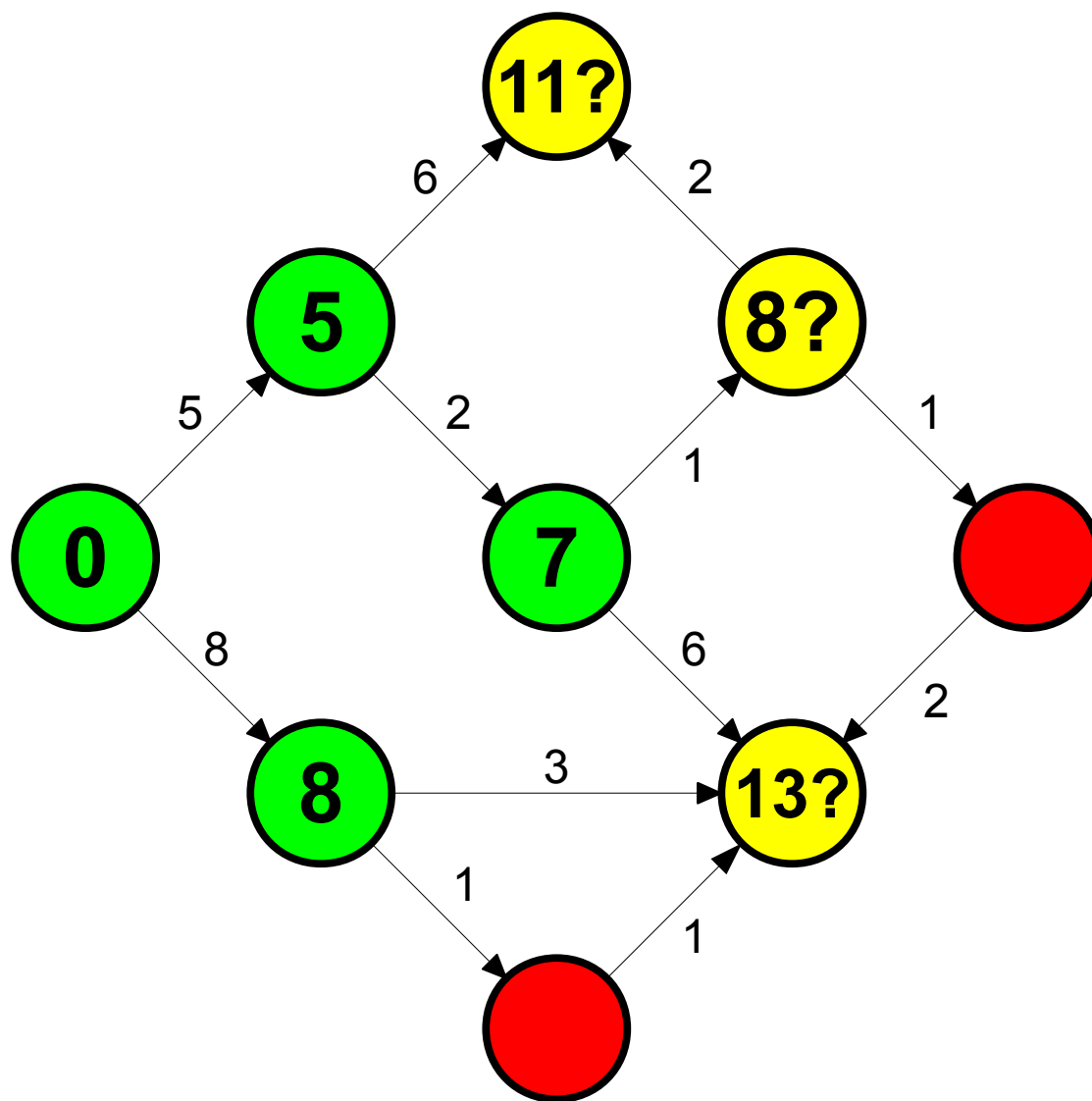


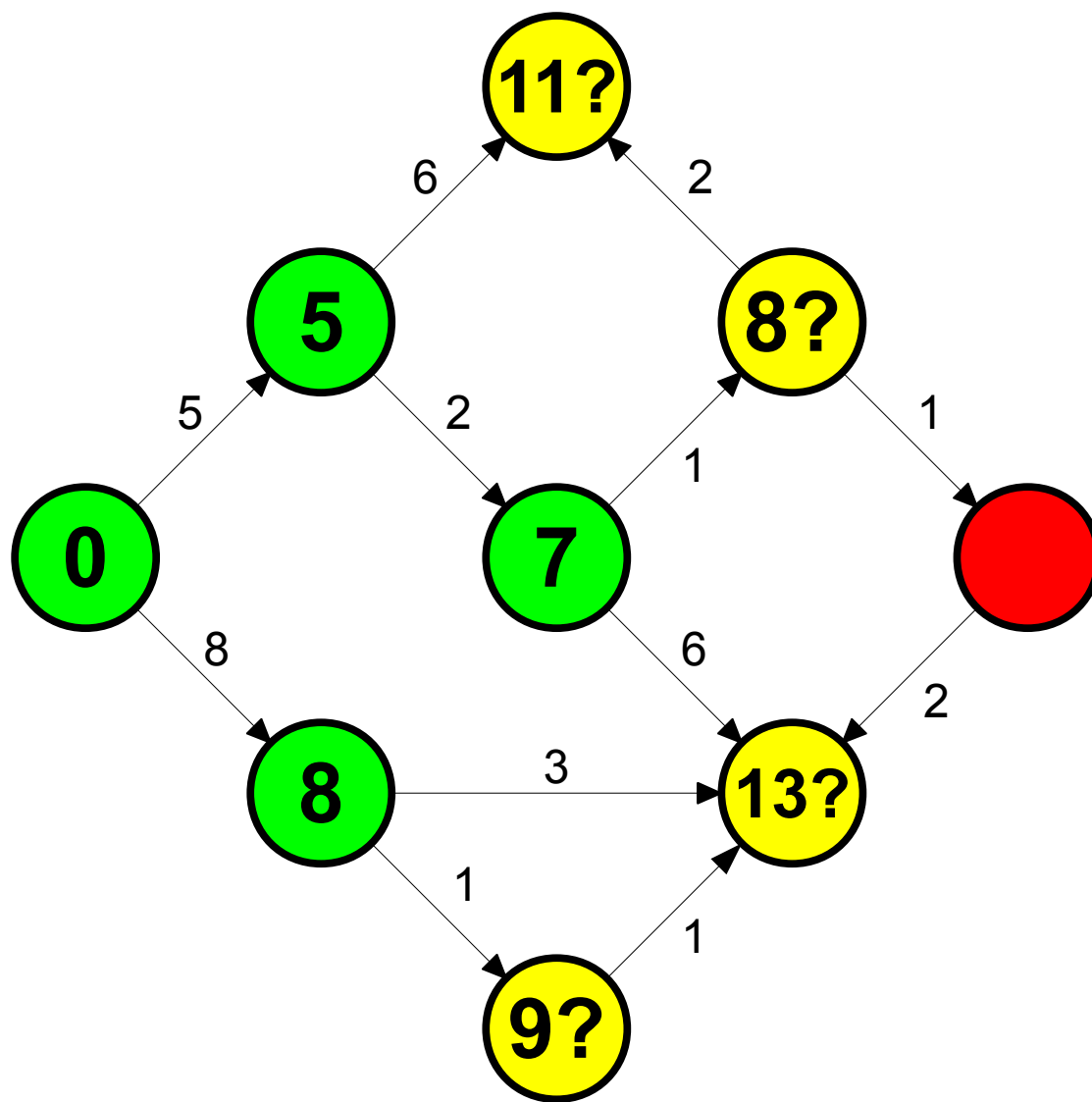


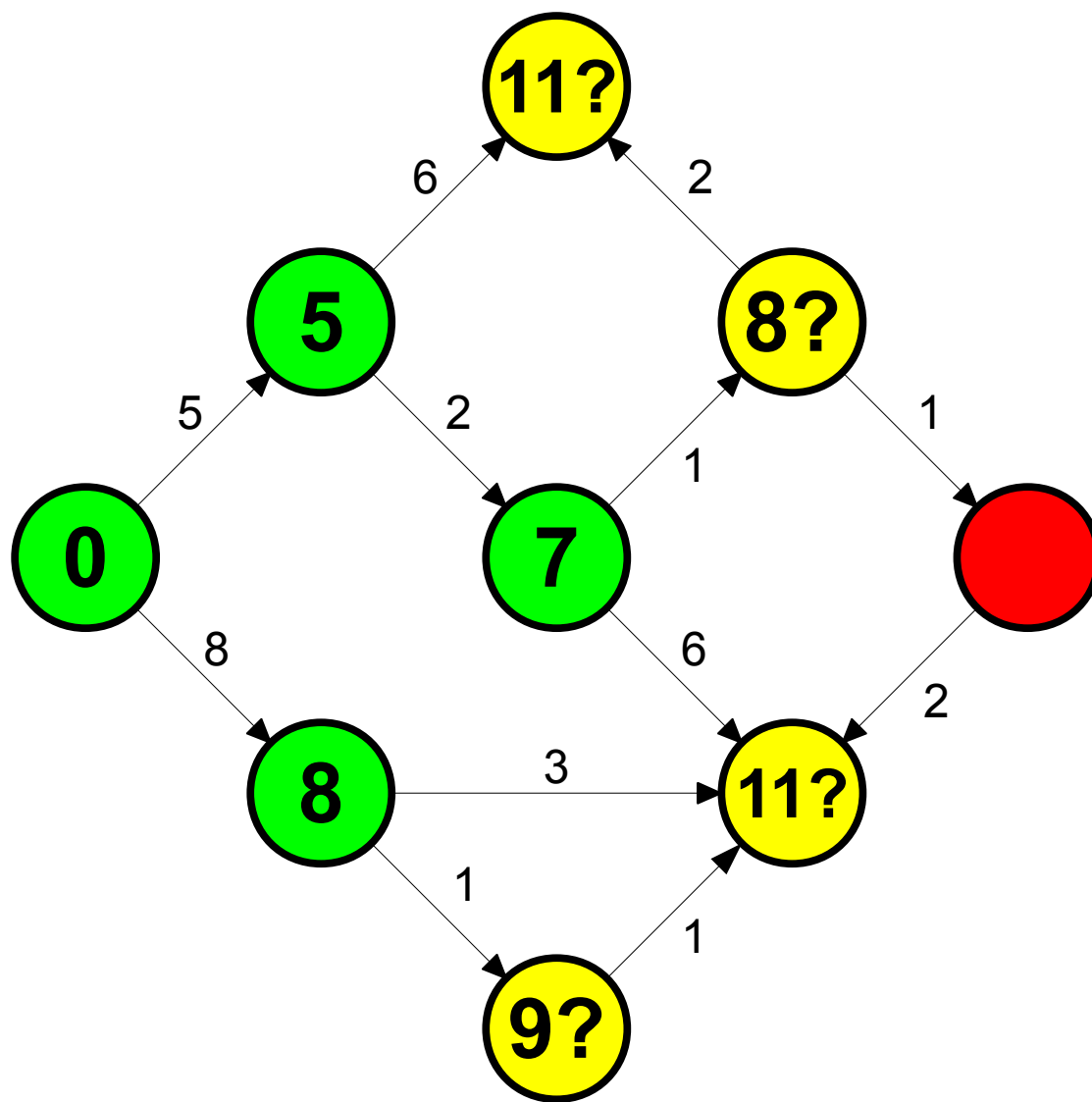


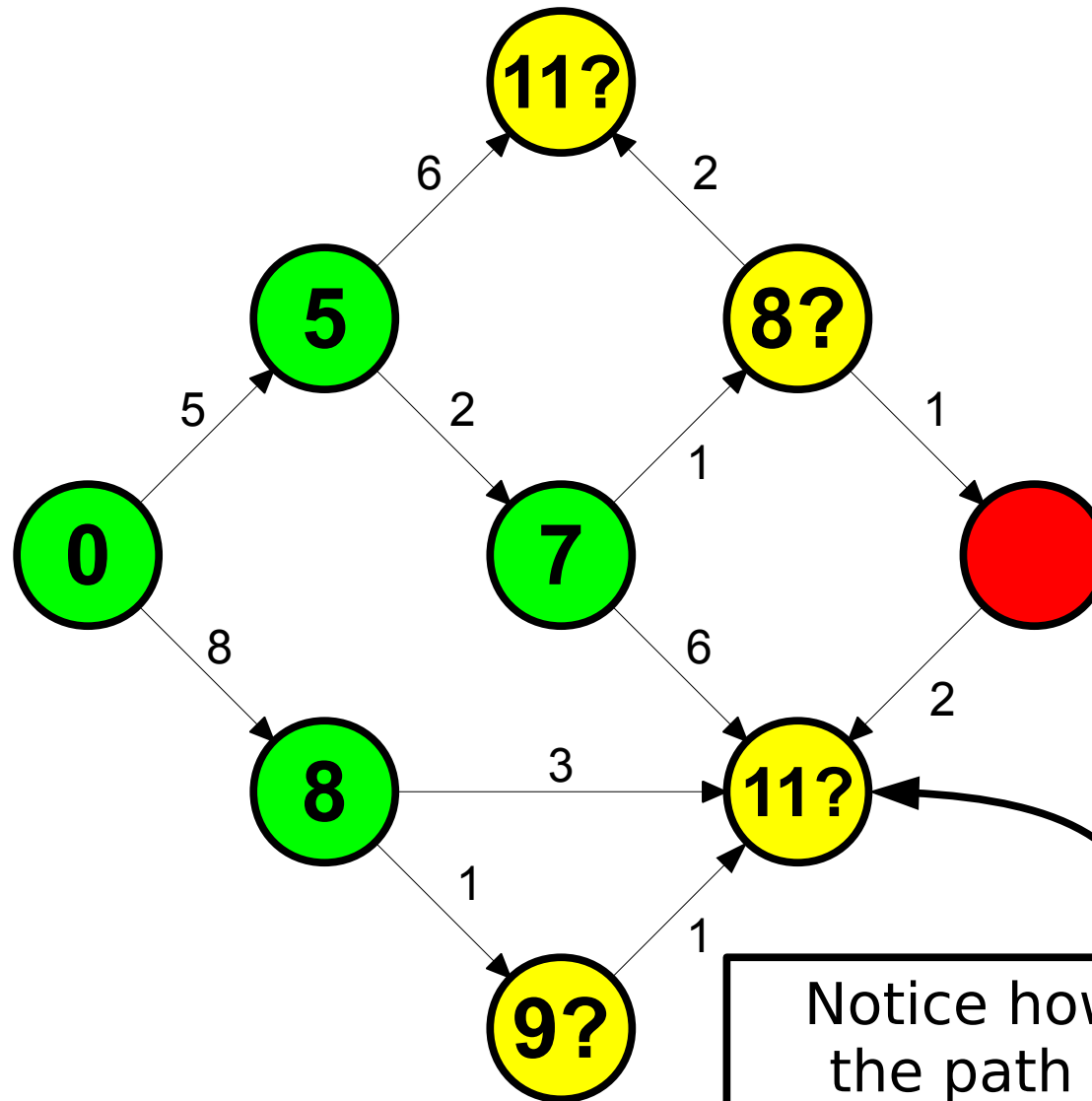




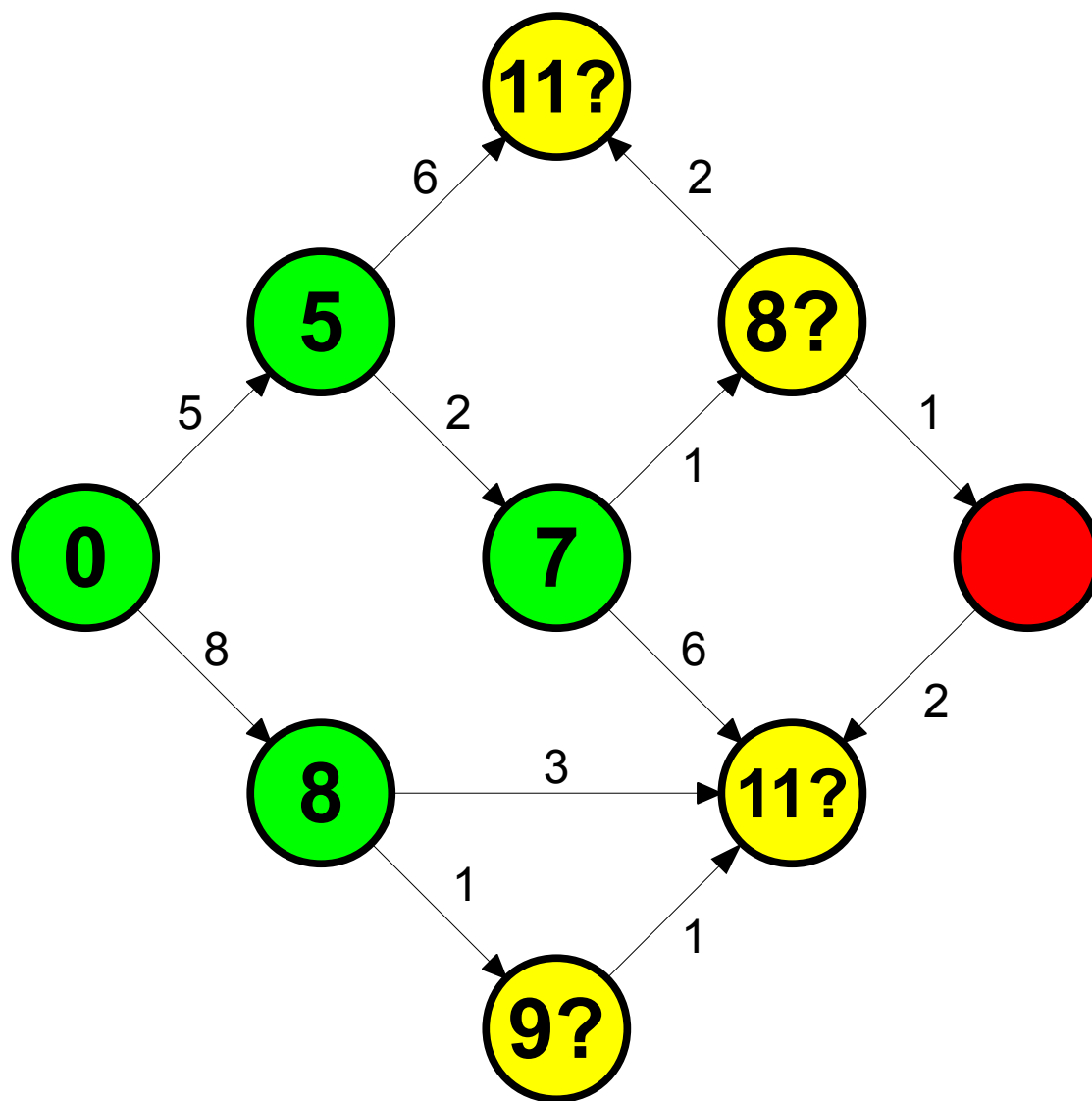




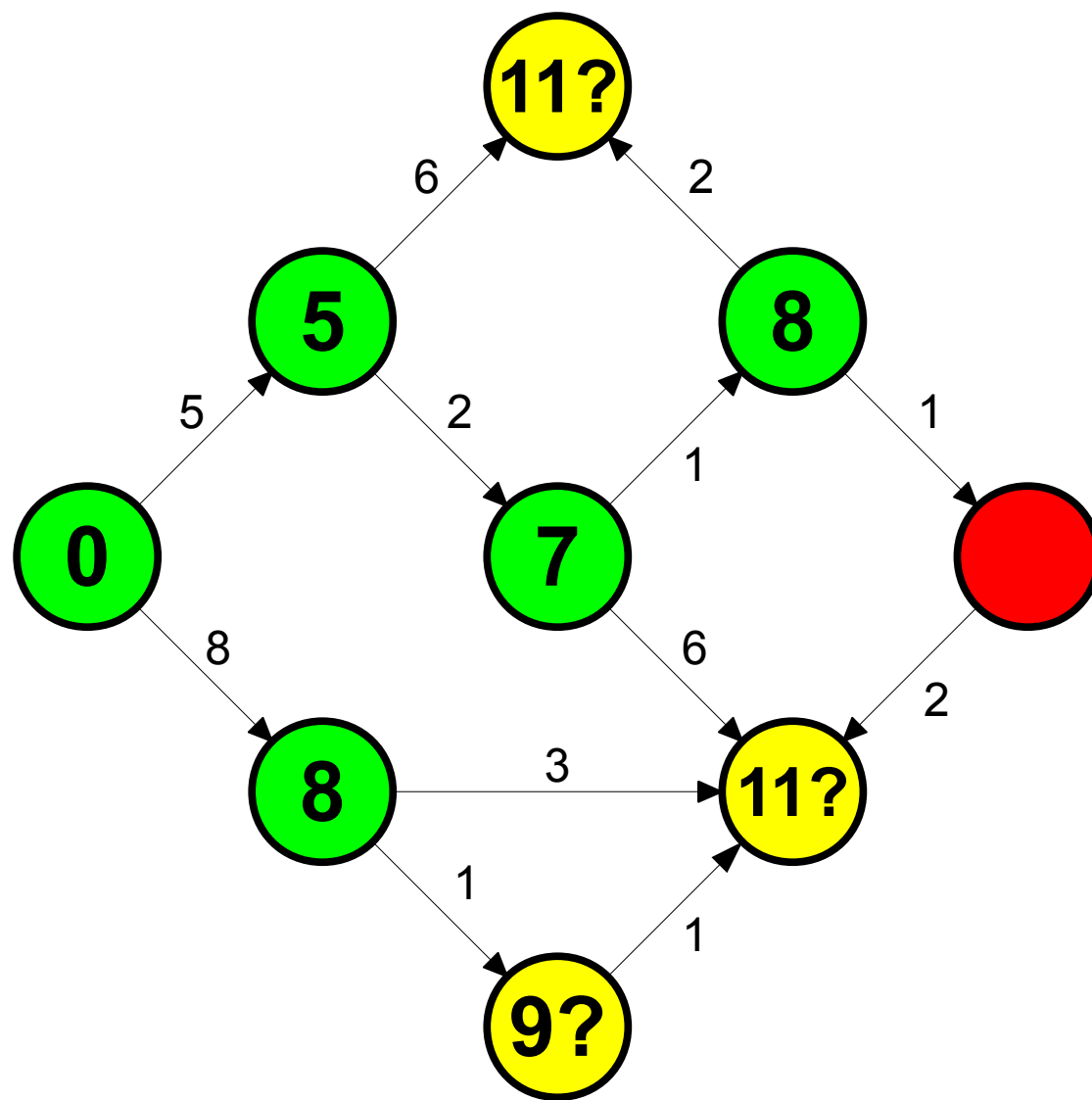


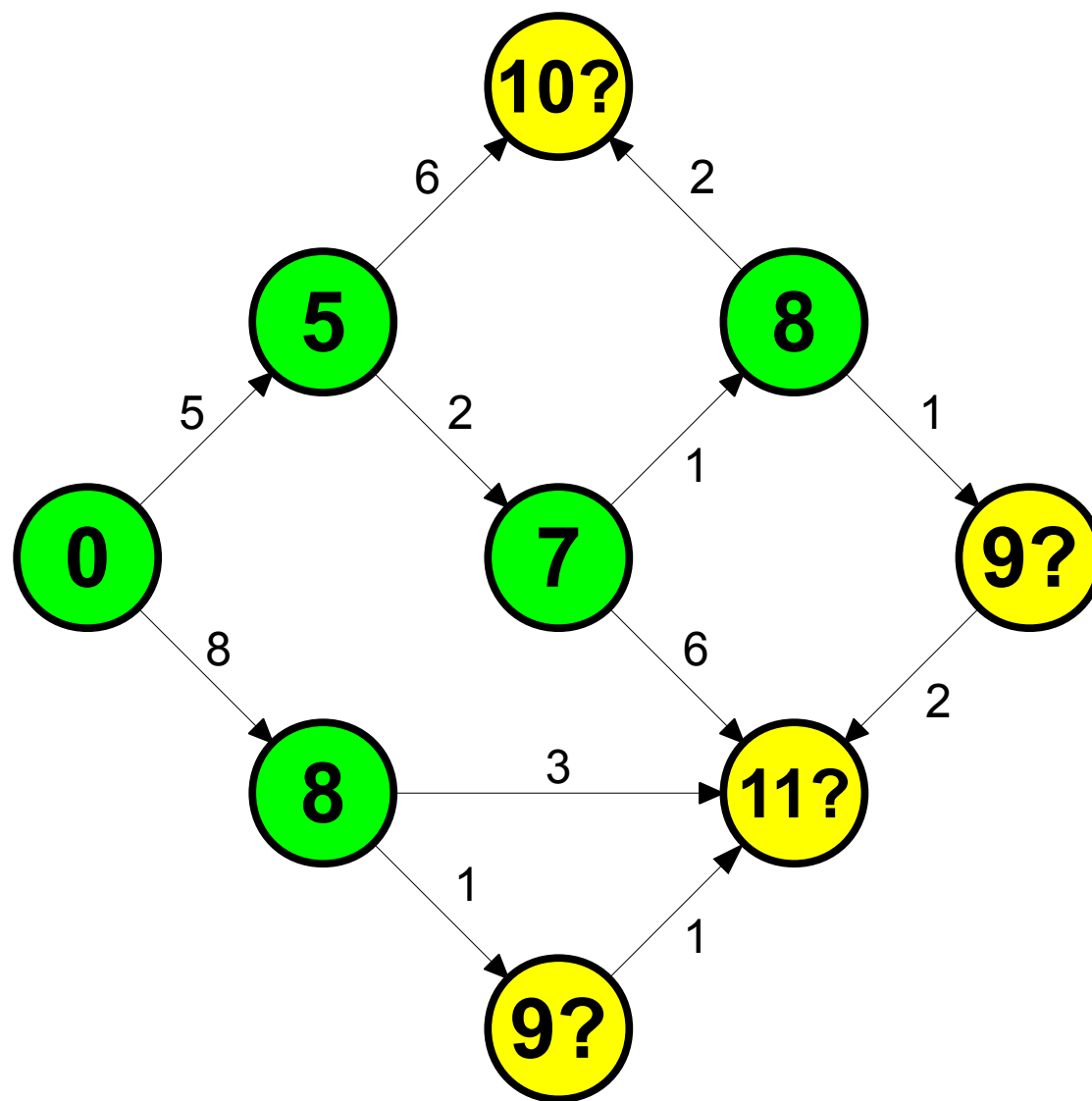


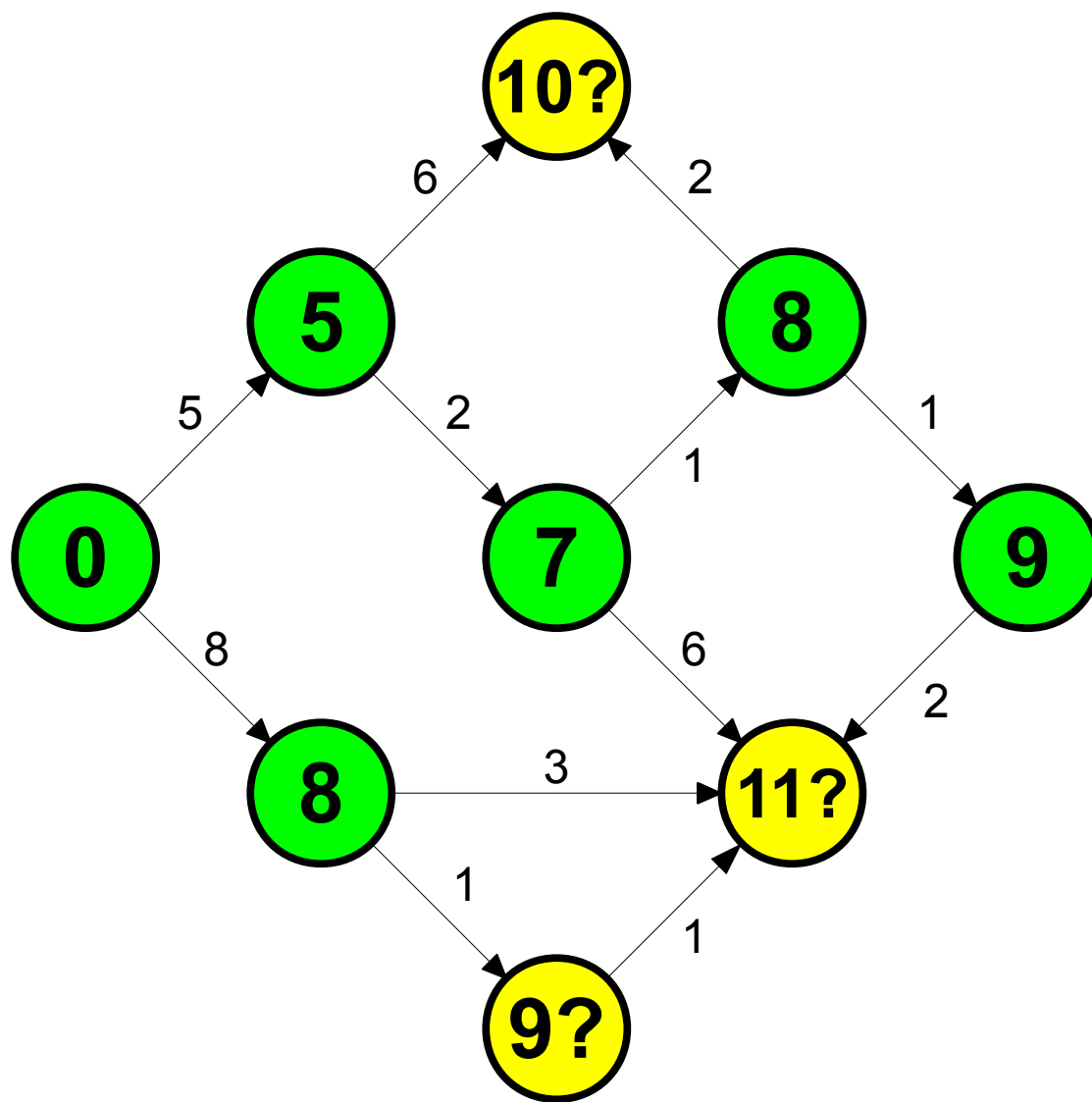
Notice how our guess of the path length to this node just changed.

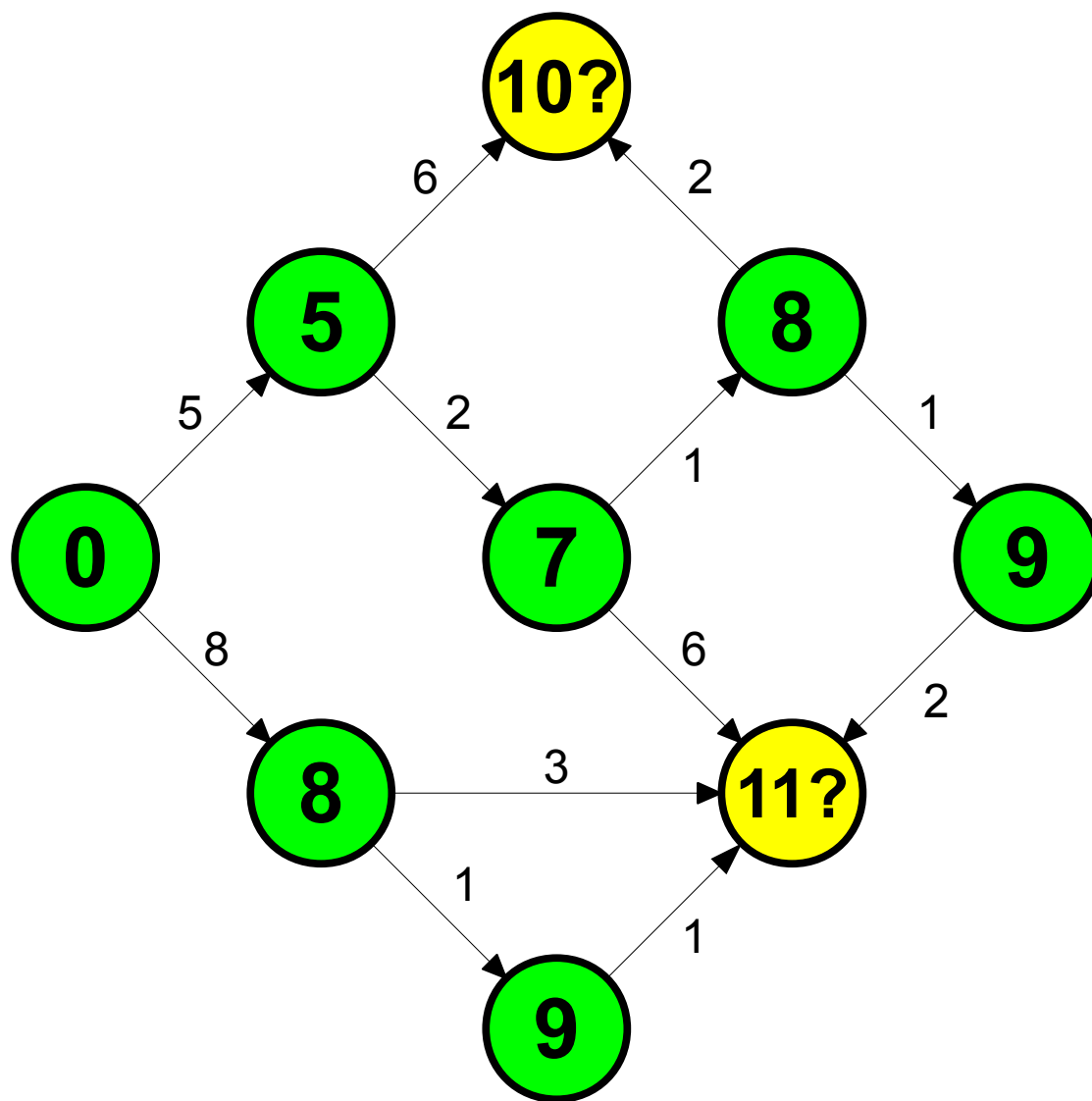


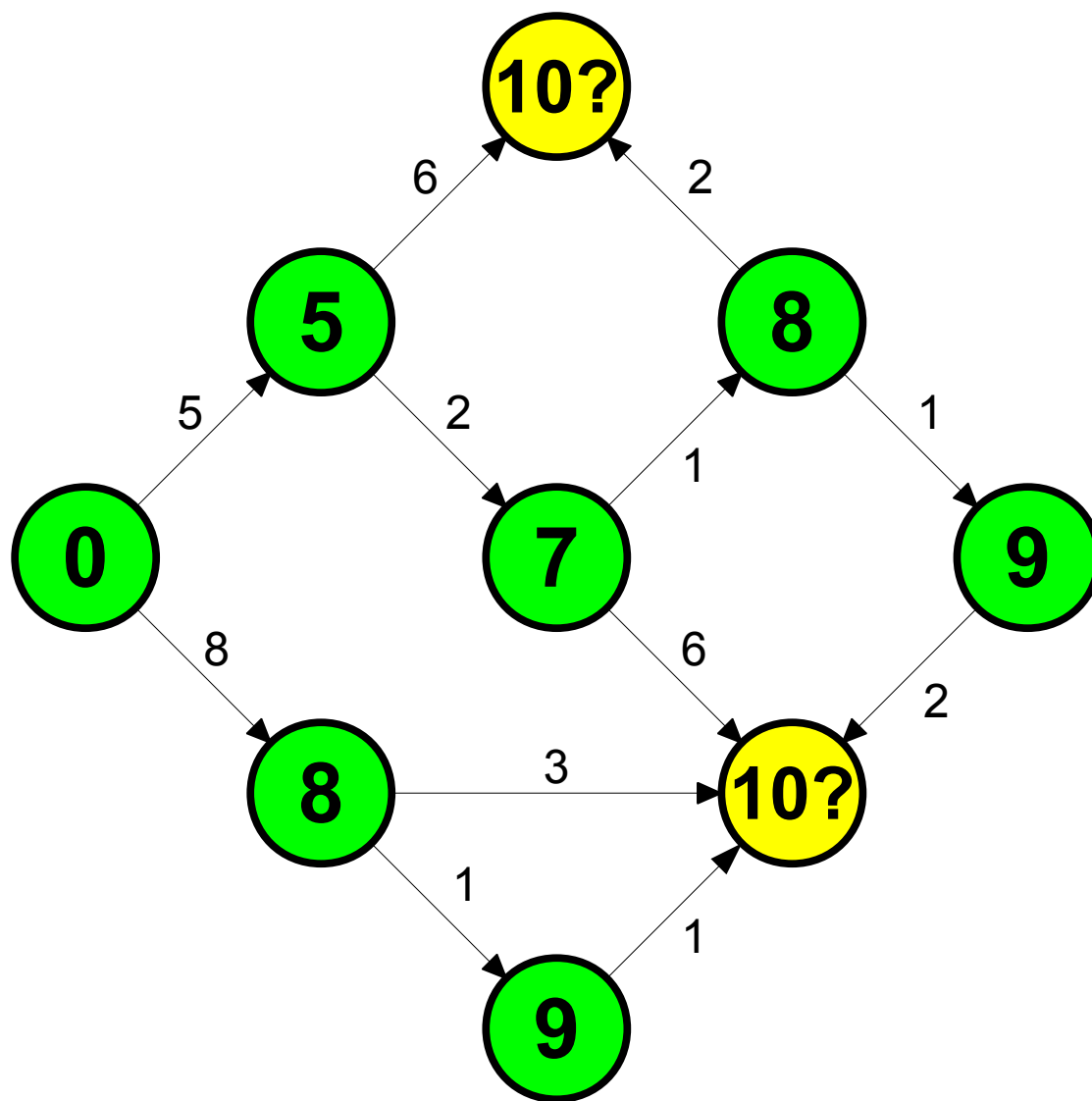


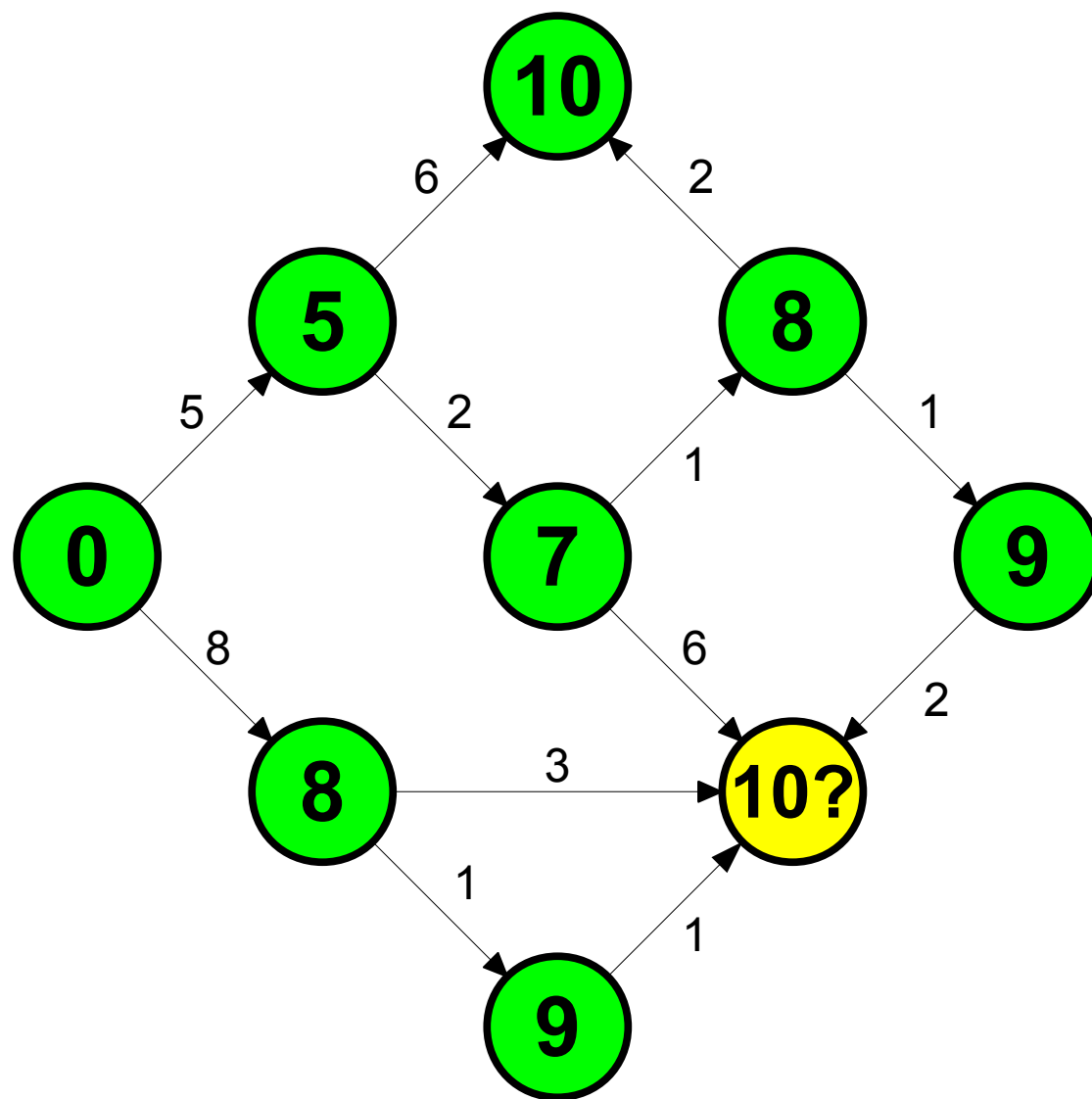


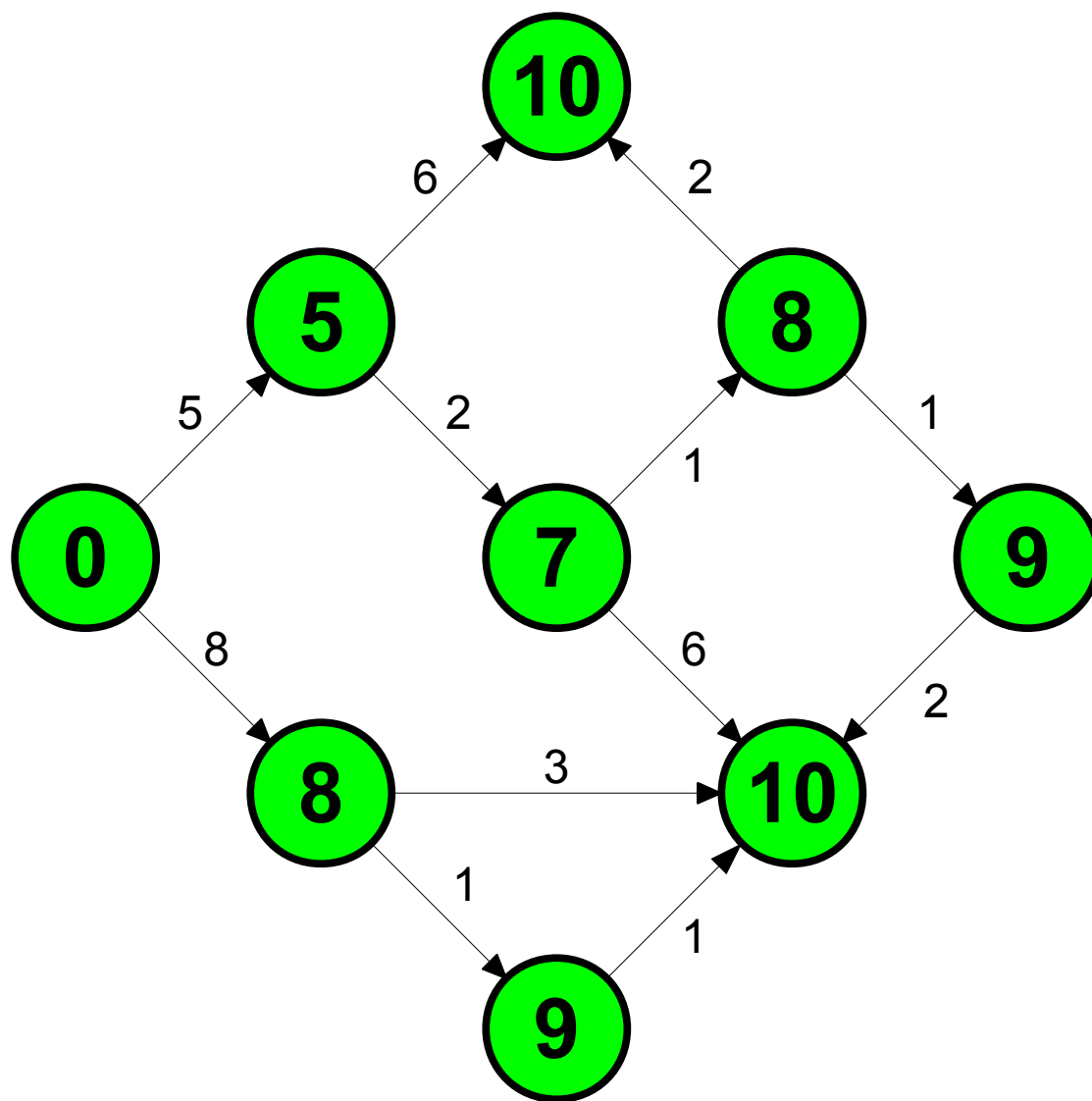












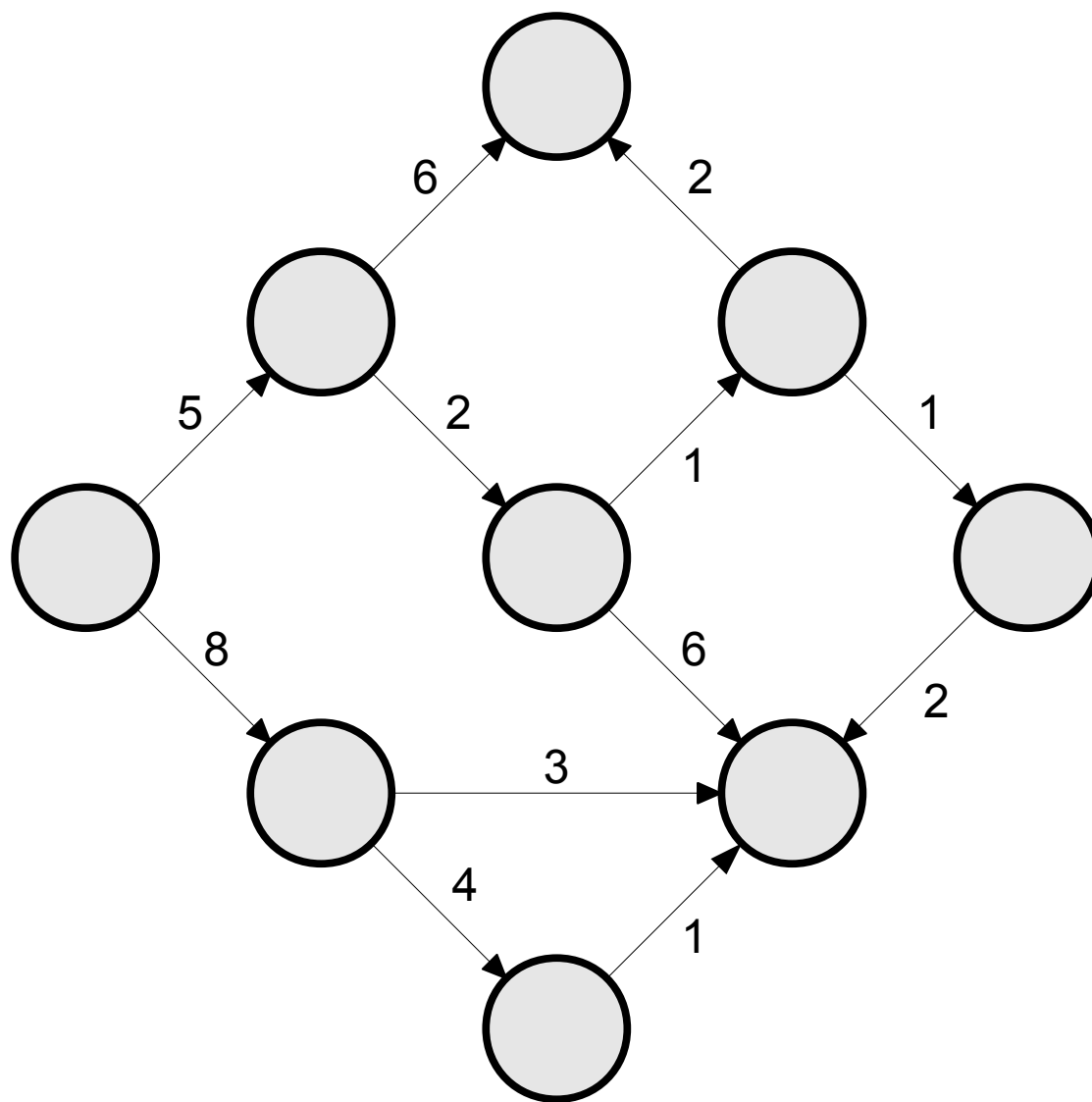
# Dijkstra's Algorithm

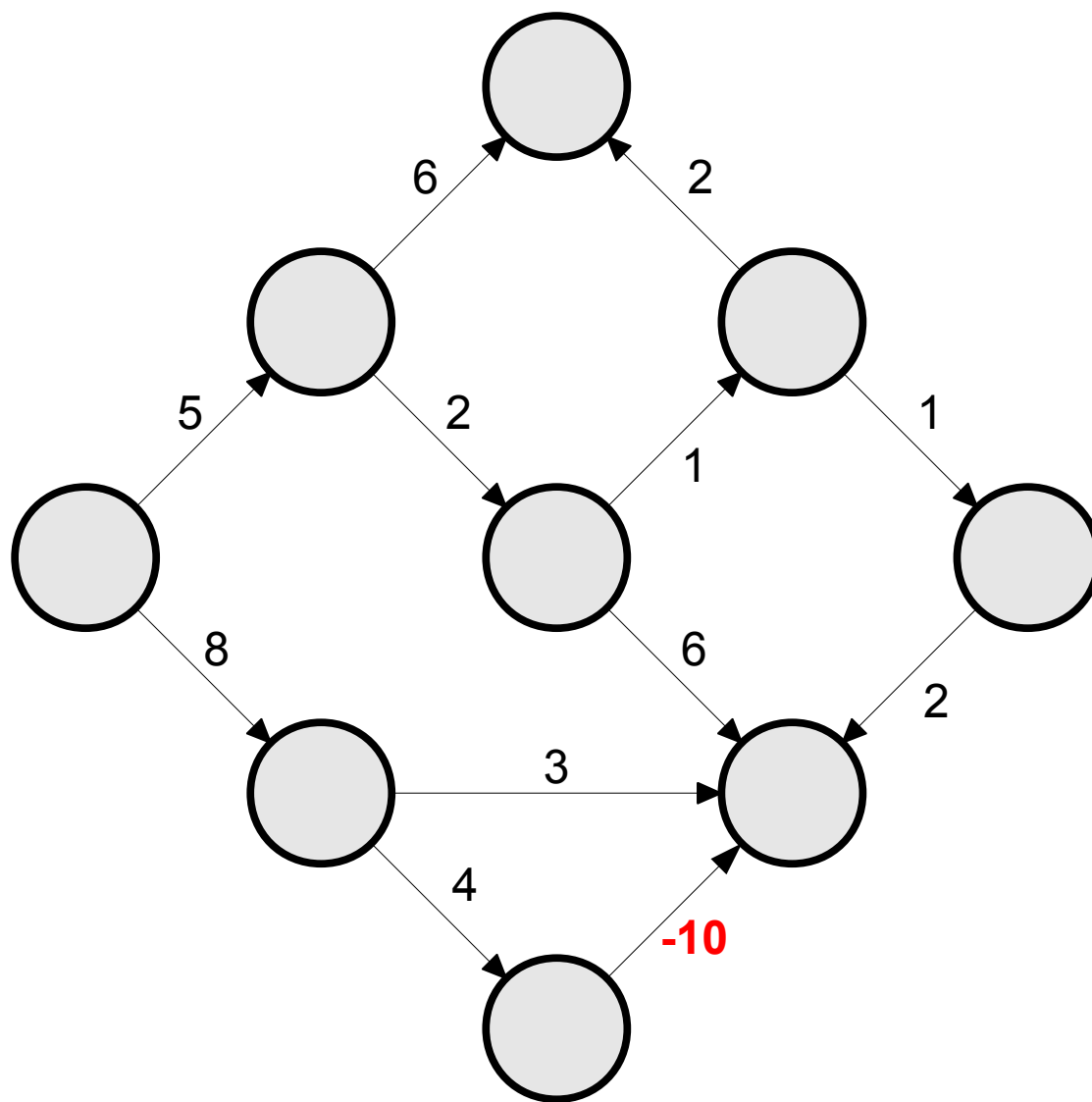
- This algorithm for finding shortest paths is called **Dijkstra's algorithm**.
- One of the fastest algorithms for finding the shortest path from  $s$  to all other nodes in the graph.
  - Can be made to run in  **$O(e + v \log v)$**  where  $v$  is the number of nodes and  $e$  is the number of edges

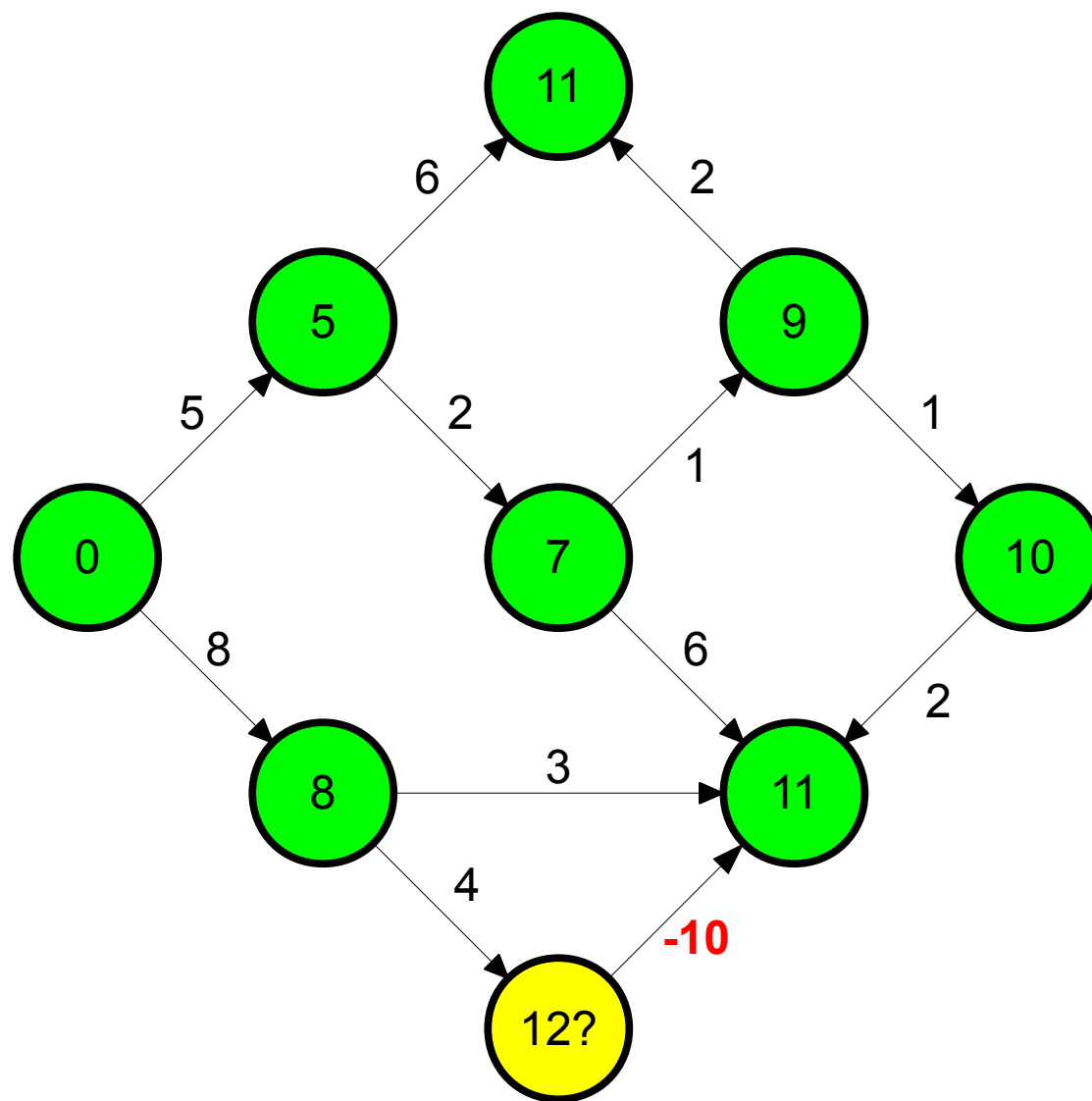


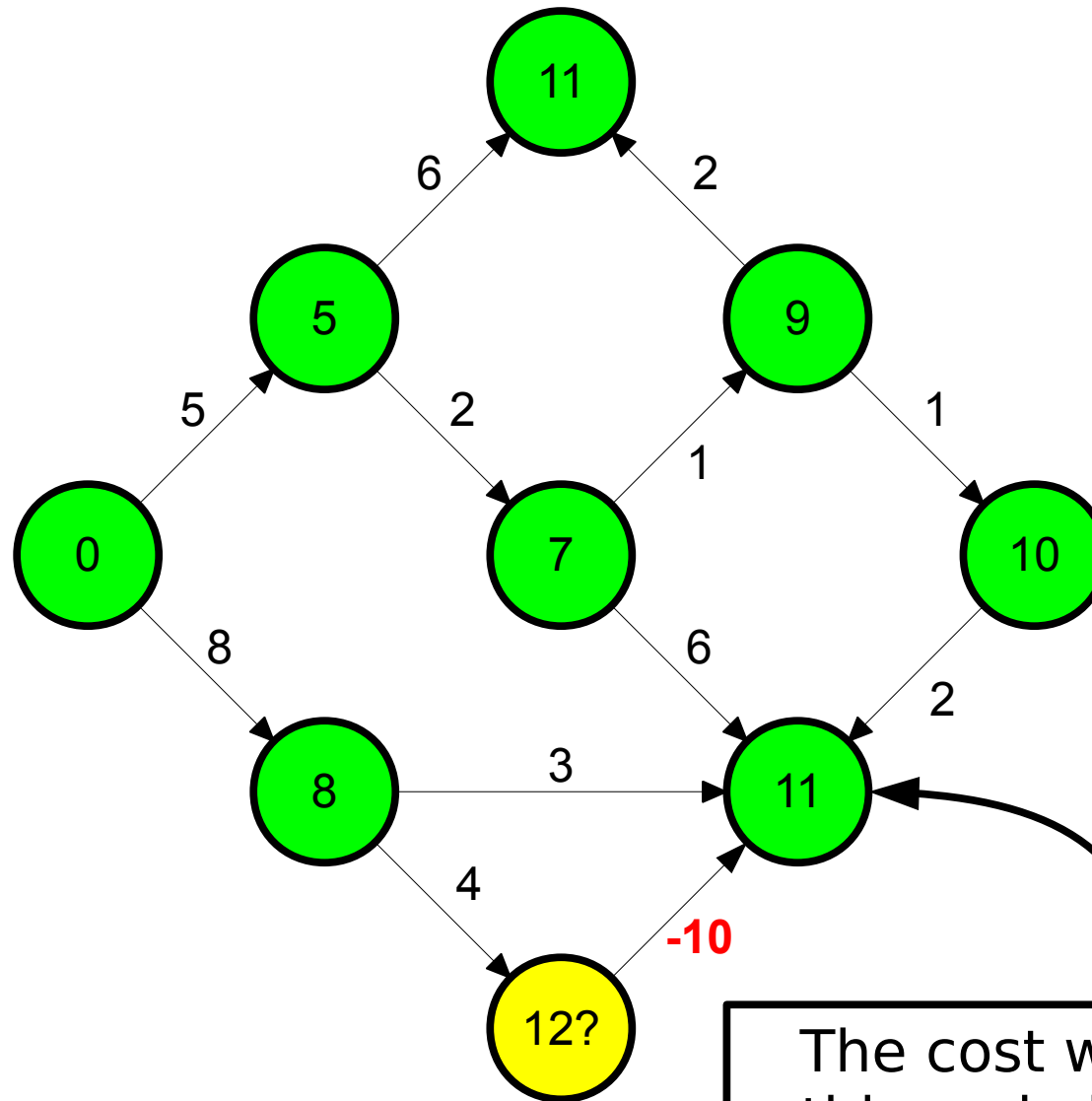
# Dijkstra's Algorithm

- Dijkstra's algorithm is guaranteed to find the shortest paths if **all the edge weights are nonnegative**
- What happens if the graph has edges with negative weights?









The cost we finalized for this node is not optimal!

# Dijkstra's Algorithm

- Earlier I said that the edge weights have to be nonnegative.
- What happens if the graph has edges with negative weights?
  - The algorithm won't calculate the correct answer. More sophisticated algorithms are required to account for negative weights.

# Dijkstra's Algorithm

- Guaranteed to find optimal paths if edges have non-negative weights.
  - Intuition for proof of this:

# Dijkstra's Algorithm

- Guaranteed to find optimal paths if edges have non-negative weights.
  - Intuition for proof of this:
    - 1) We only finalize paths to nodes if that path has the minimum weight.



# Dijkstra's Algorithm

- Guaranteed to find optimal paths if edges have non-negative weights.
  - Intuition for proof of this:
    - 1) We only finalize paths to nodes if that path has the minimum weight.
    - 2) Since all edges have nonnegative weight, any other path to this node will be greater than or equal to the cost of the path built up to it at this point in the algorithm.

# Dijkstra's Algorithm

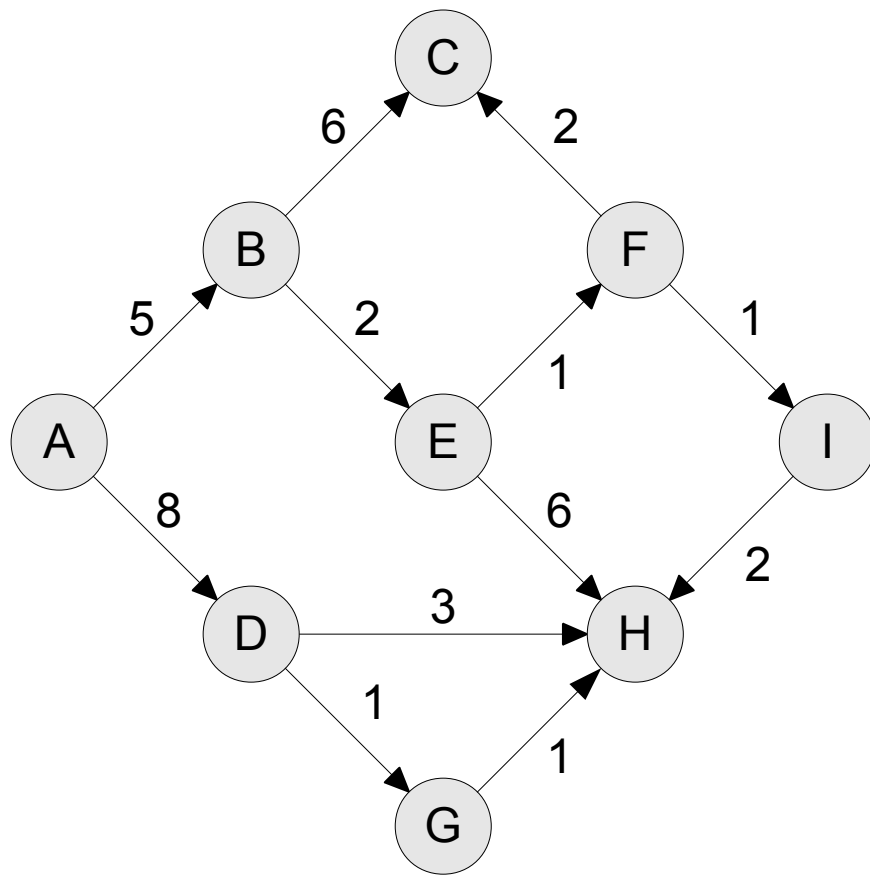
- Guaranteed to find optimal paths if edges have non-negative weights.
  - Intuition for proof of this:
    - 1) We only finalize paths to nodes if that path has the minimum weight.
    - 2) Since all edges have nonnegative weight, any other path to this node will be greater than or equal to the cost of the path built up to it at this point in the algorithm.
    - 3) Therefore, any other path to the node we are finalizing will have greater cost than the path we have built up to it.

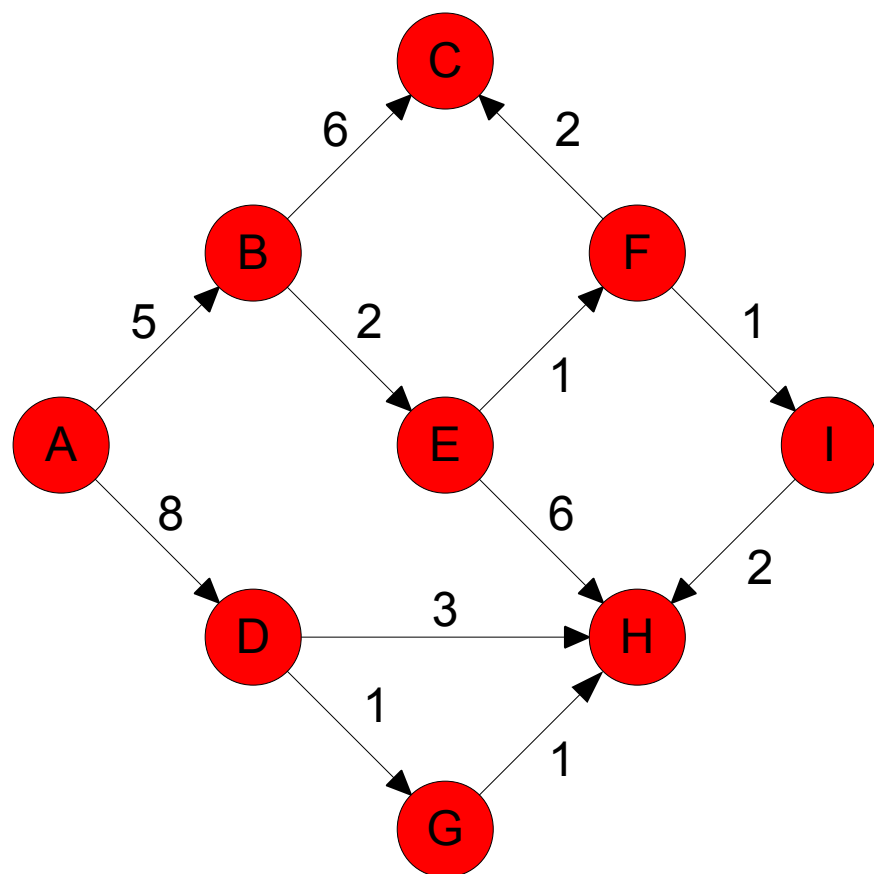
# Dijkstra's Algorithm

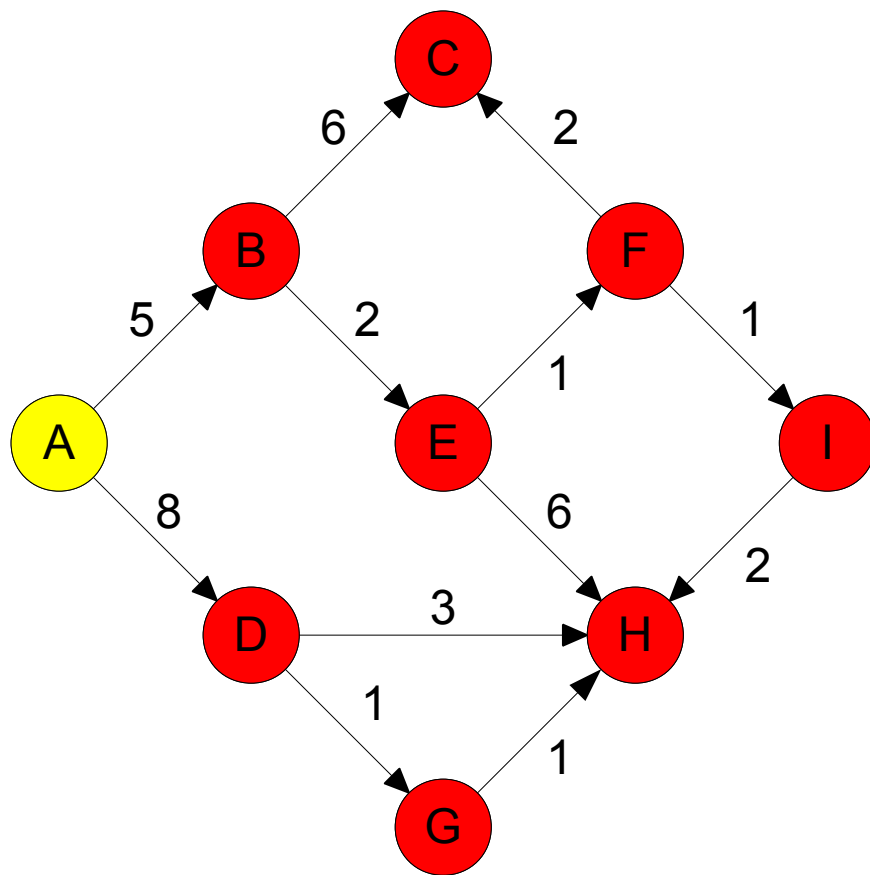
- Guaranteed to find optimal paths if edges have non-negative weights.
  - Intuition for proof of this:
    - 1) We only finalize paths to nodes if that path has the minimum weight.
    - 2) Since all edges have nonnegative weight, any other path to this node will be greater than or equal to the cost of the path built up to it at this point in the algorithm.
    - 3) Therefore, any other path to the node we are finalizing will have greater cost than the path we have built up to it.
    - 4) QED

# Implementing Dijkstra's Algorithm

The Simpler Way of Doing It  
More Intuitive, Much Slower

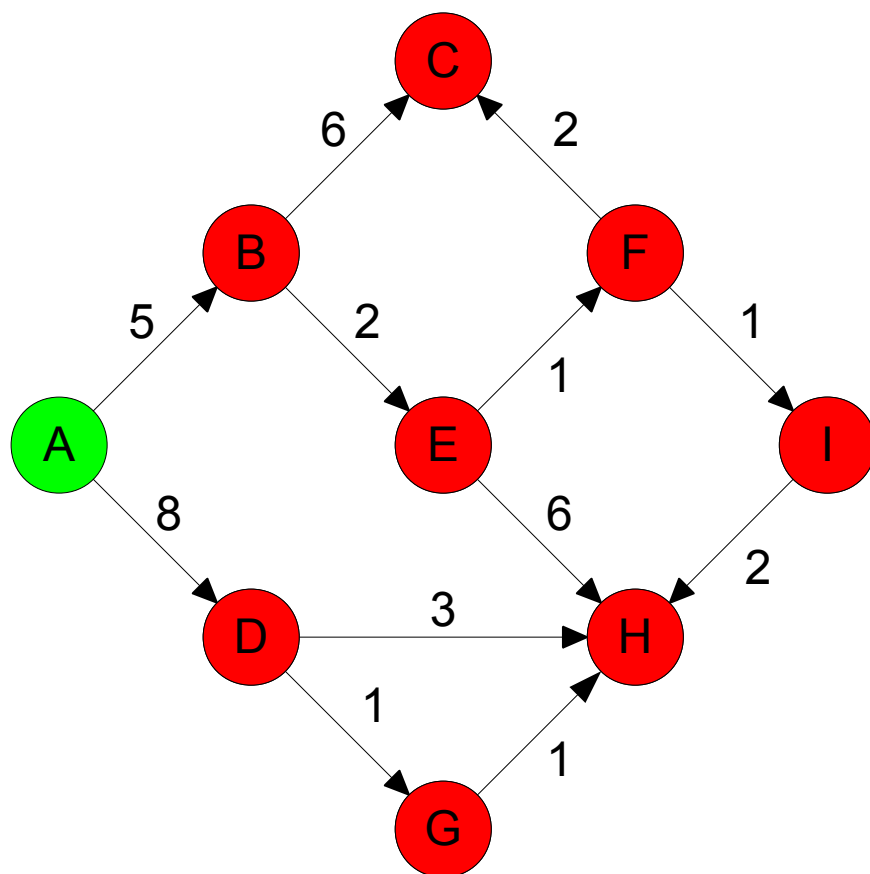






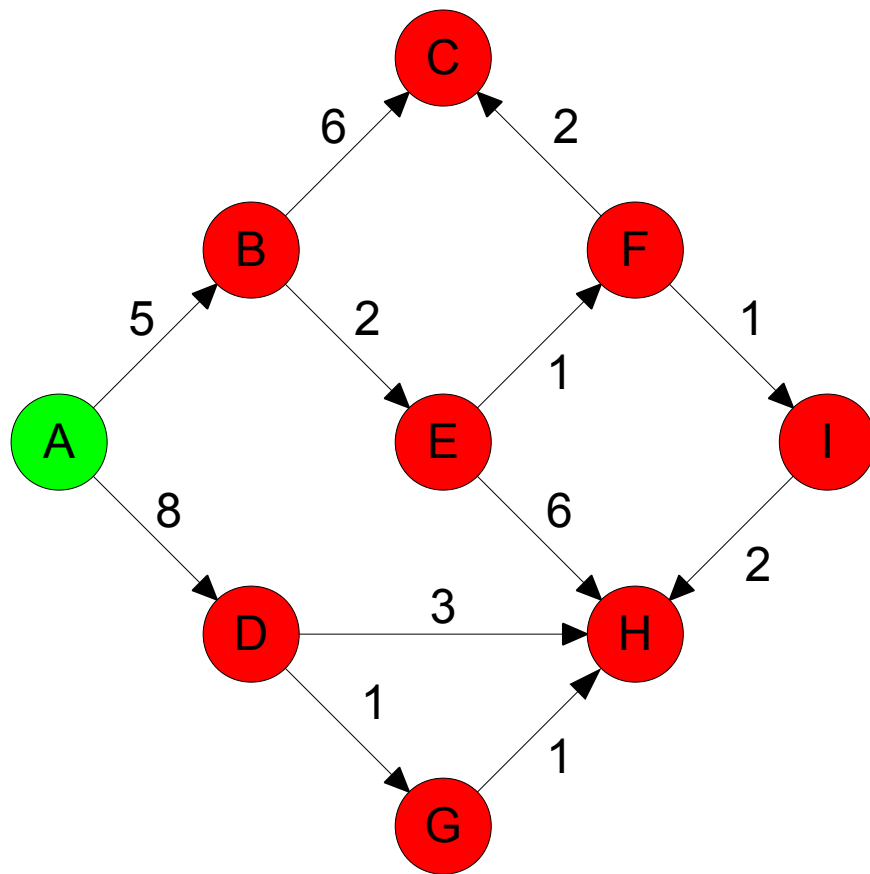
(0) A

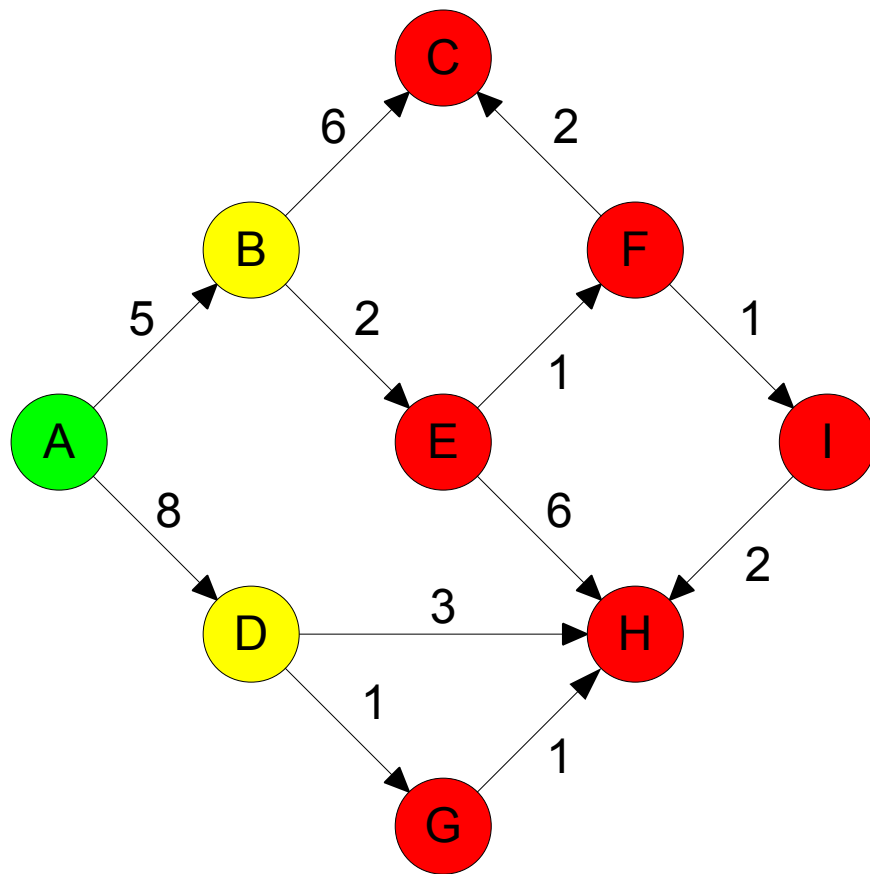




(0) A

(0) A

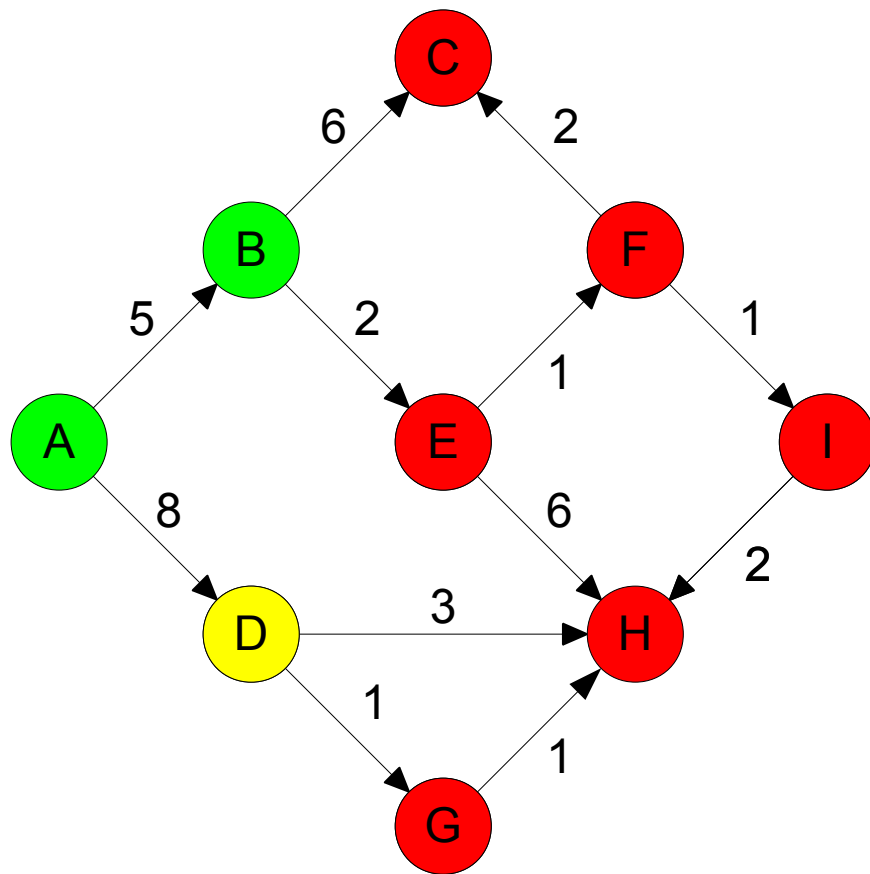




(0) A

(5)  $A \rightarrow B$

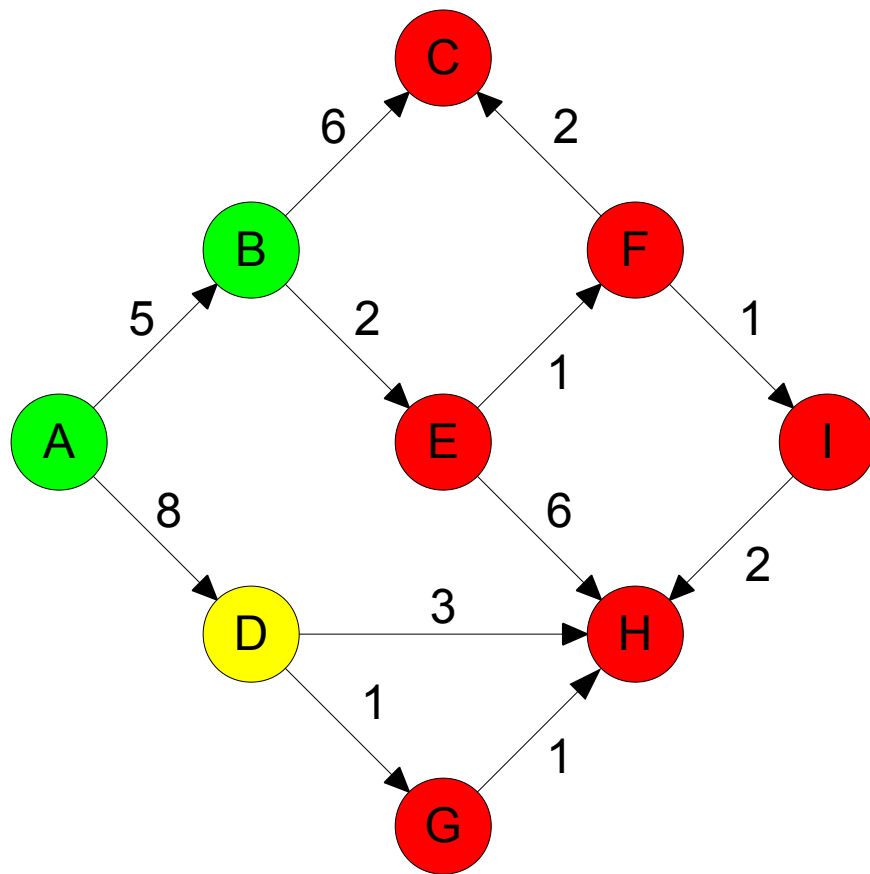
(8)  $A \rightarrow D$



(0) A

(5)  $A \rightarrow B$

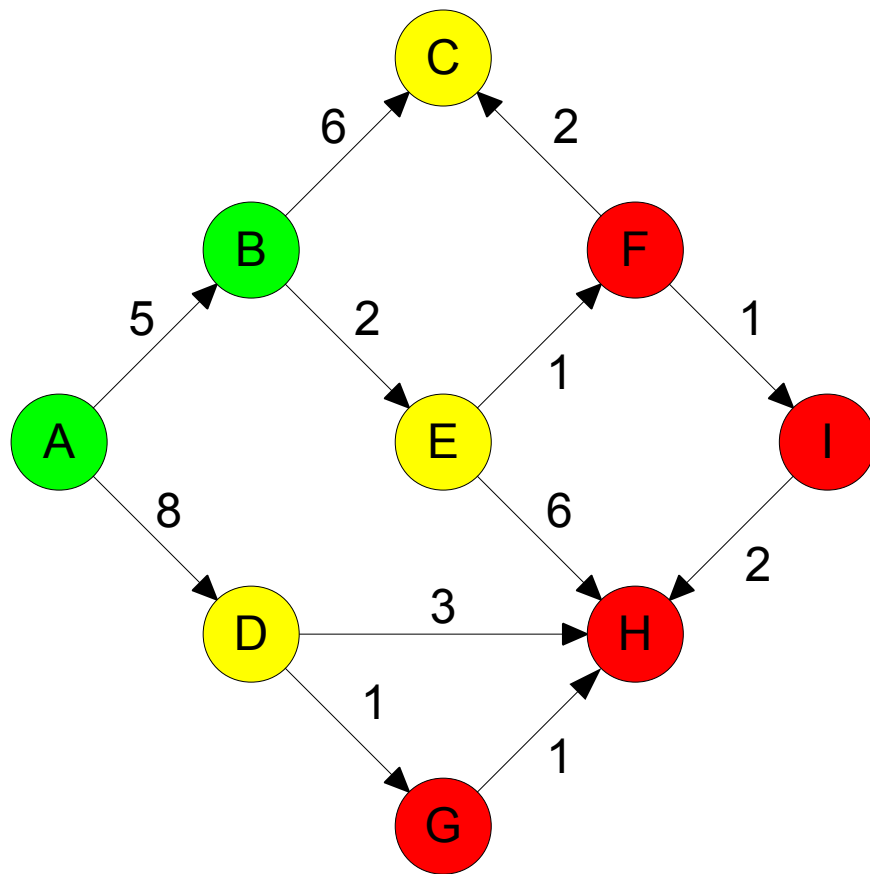
(8)  $A \rightarrow D$



(0) A

(5) A → B

(8) A → D



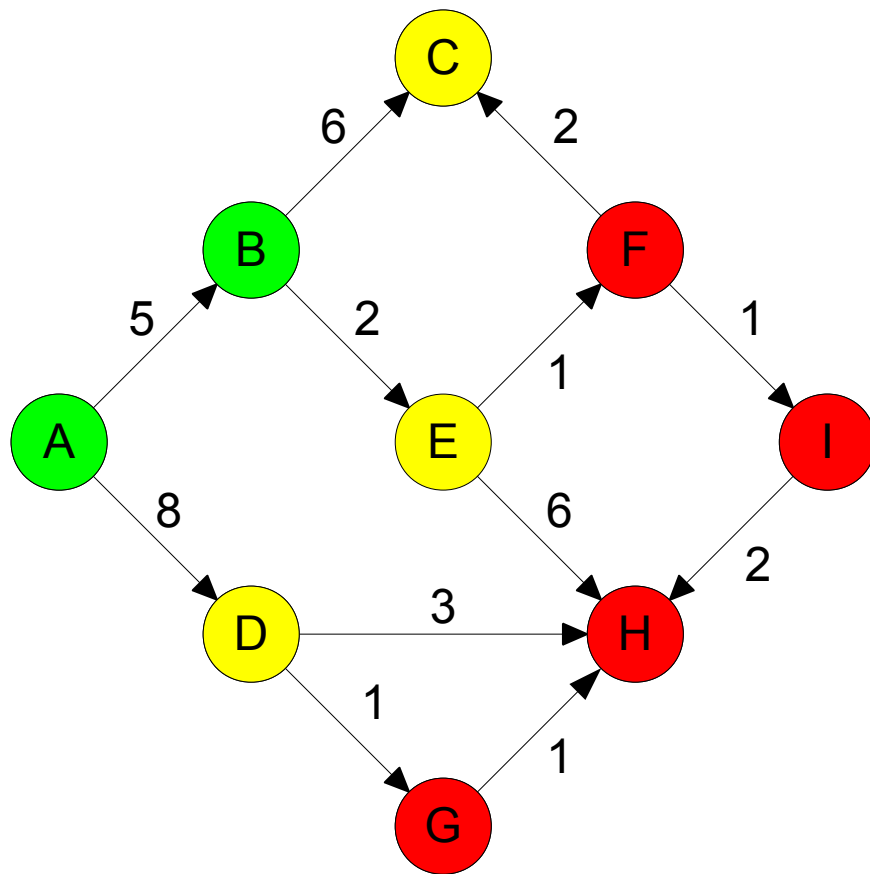
(0) A

(5) A→B

(8) A→D

(11) A→B→C

(7) A→B→E



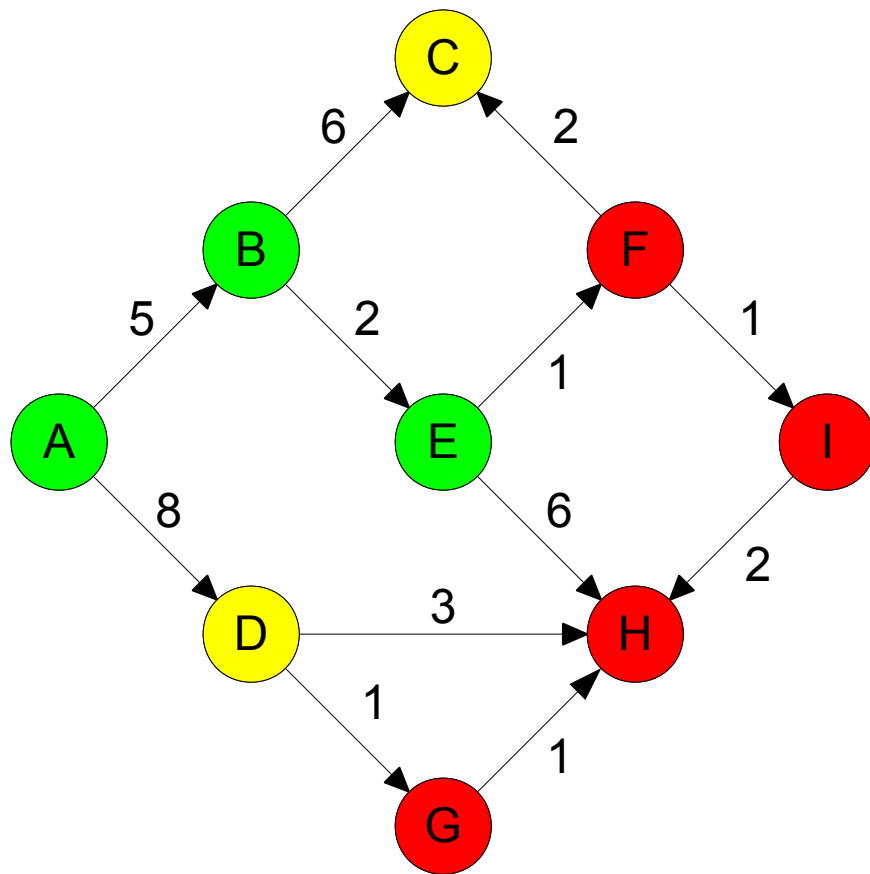
(0) A

(5) A→B

(7) A→B→E

(8) A→D

(11) A→B→C



(0) A

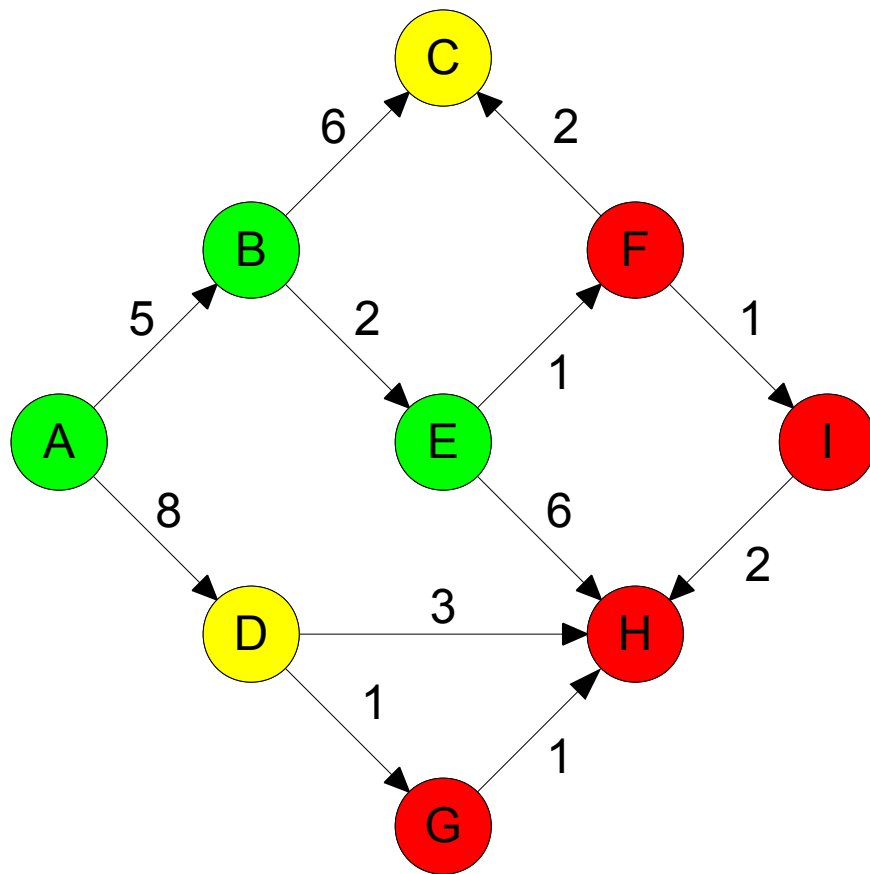
(5) A → B

(7) A → B → E

(8) A → D

(11) A → B → C





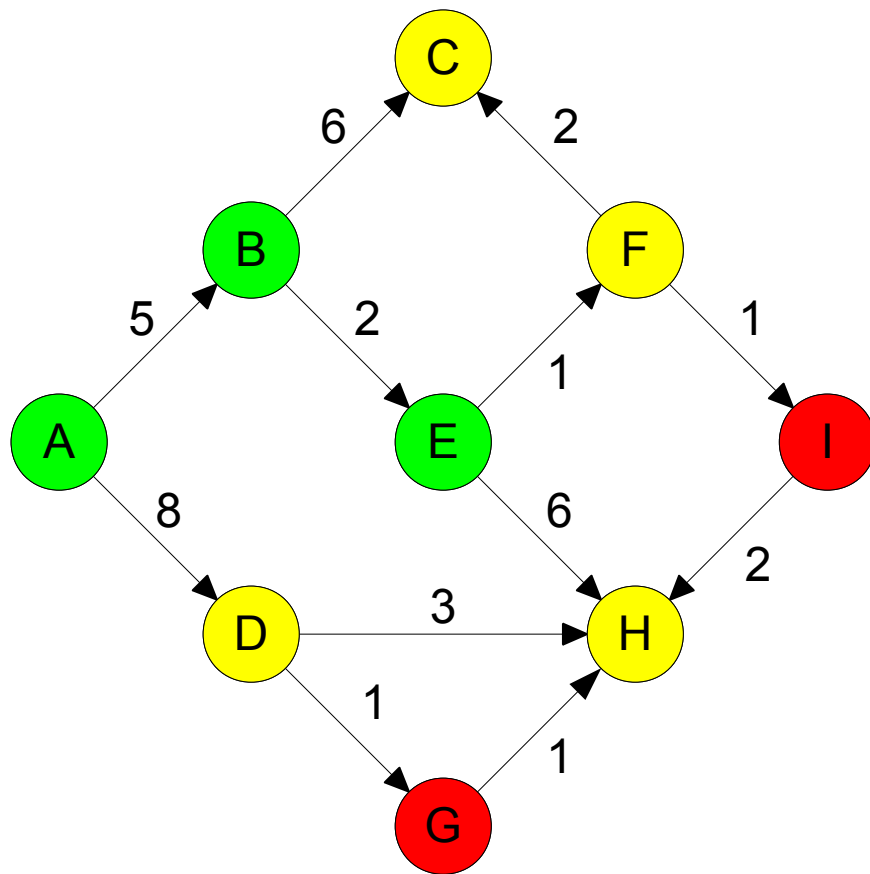
(0) A

(5) A→B

(7) A→B→E

(8) A→D

(11) A→B→C



(0) A

(5) A→B

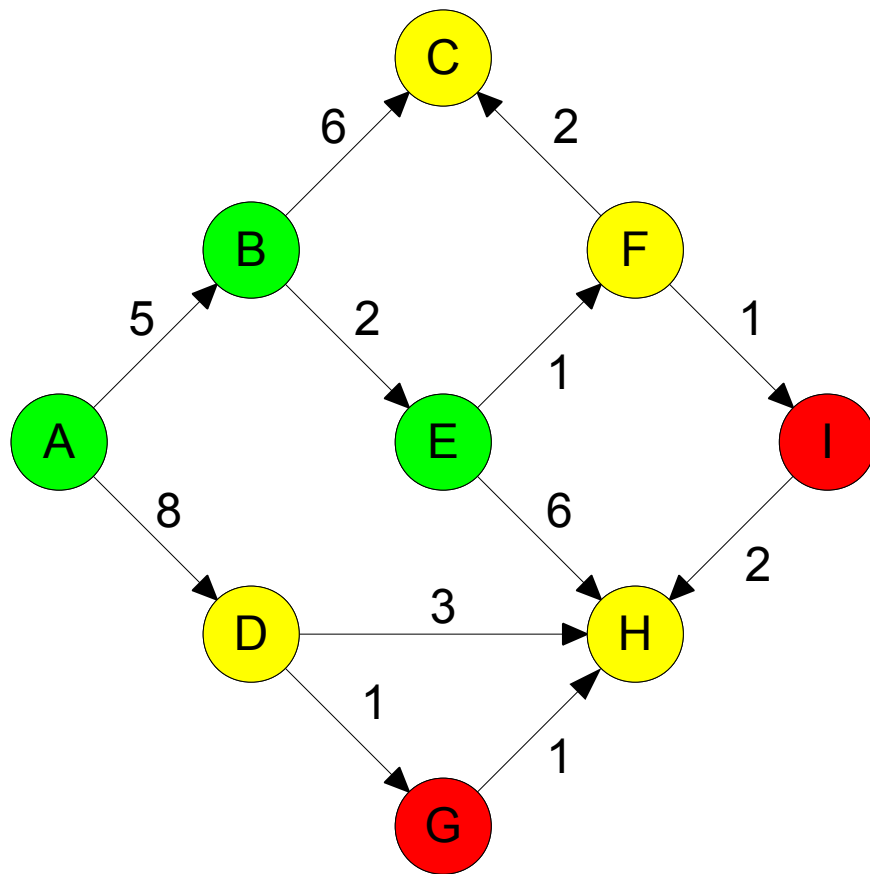
(7) A→B→E

(8) A→D

(11) A→B→C

(8) A→B→E→F

(13) A→B→E→H



(0) A

(5) A→B

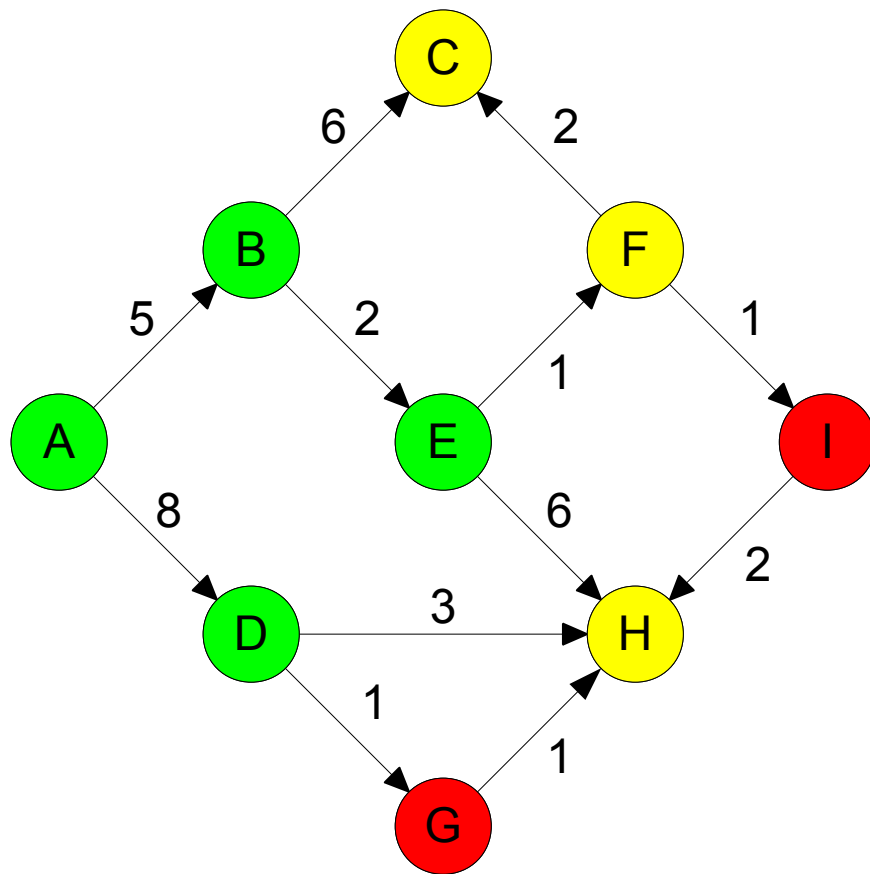
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

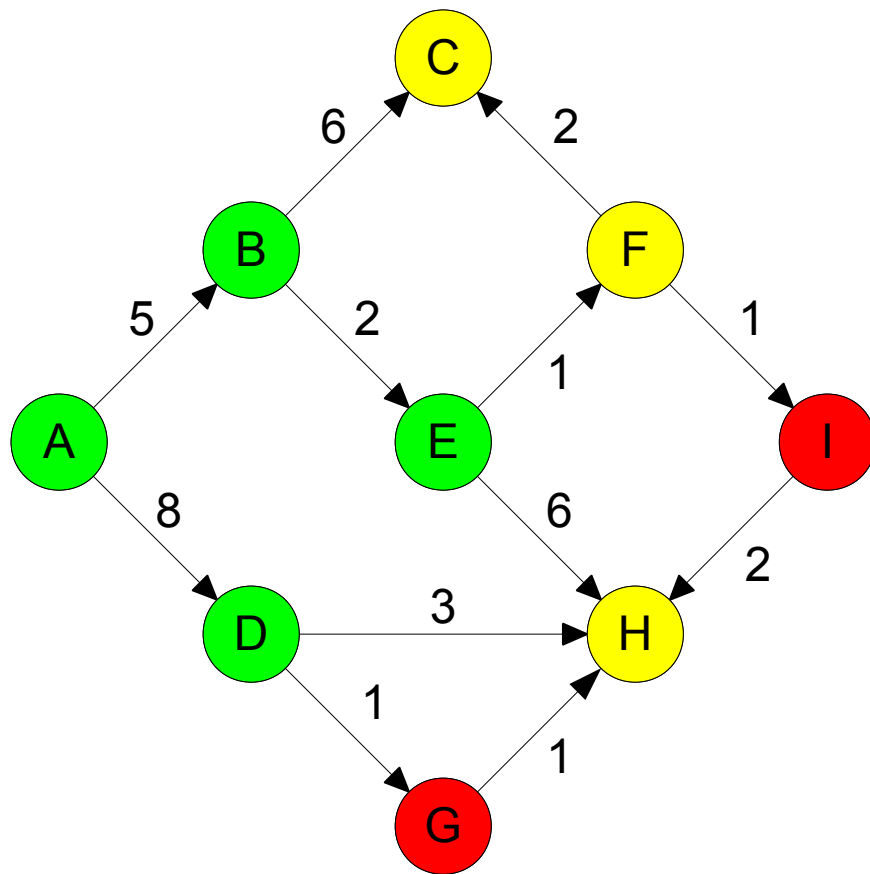
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

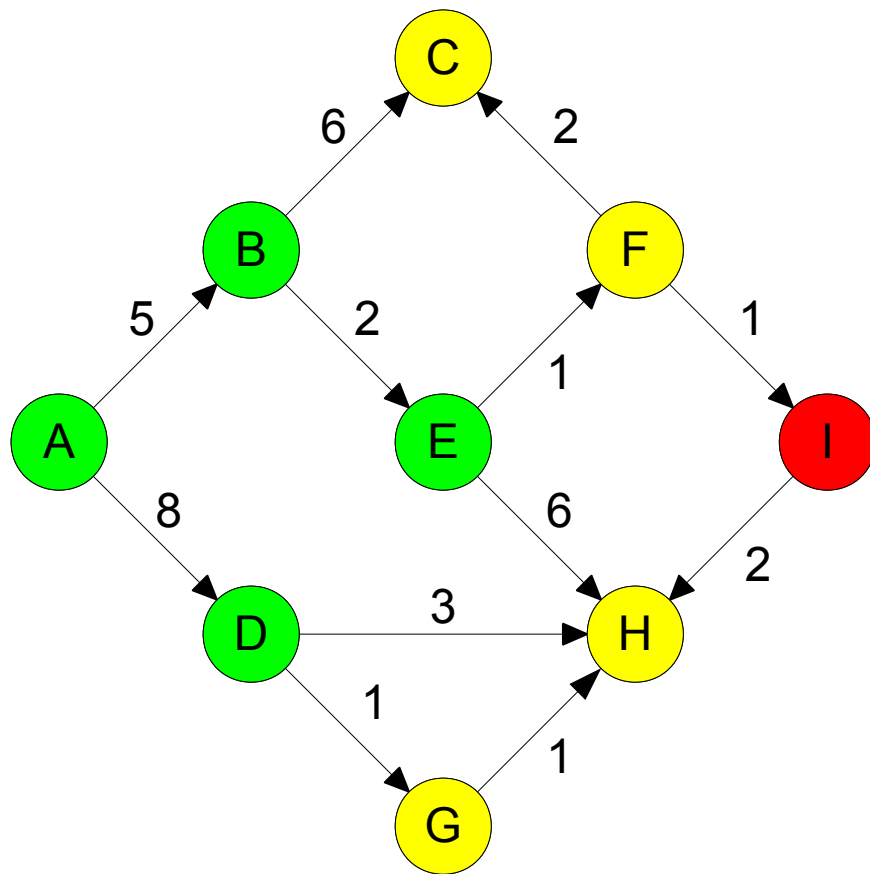
(7) A→B→E

(8) A→D

(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

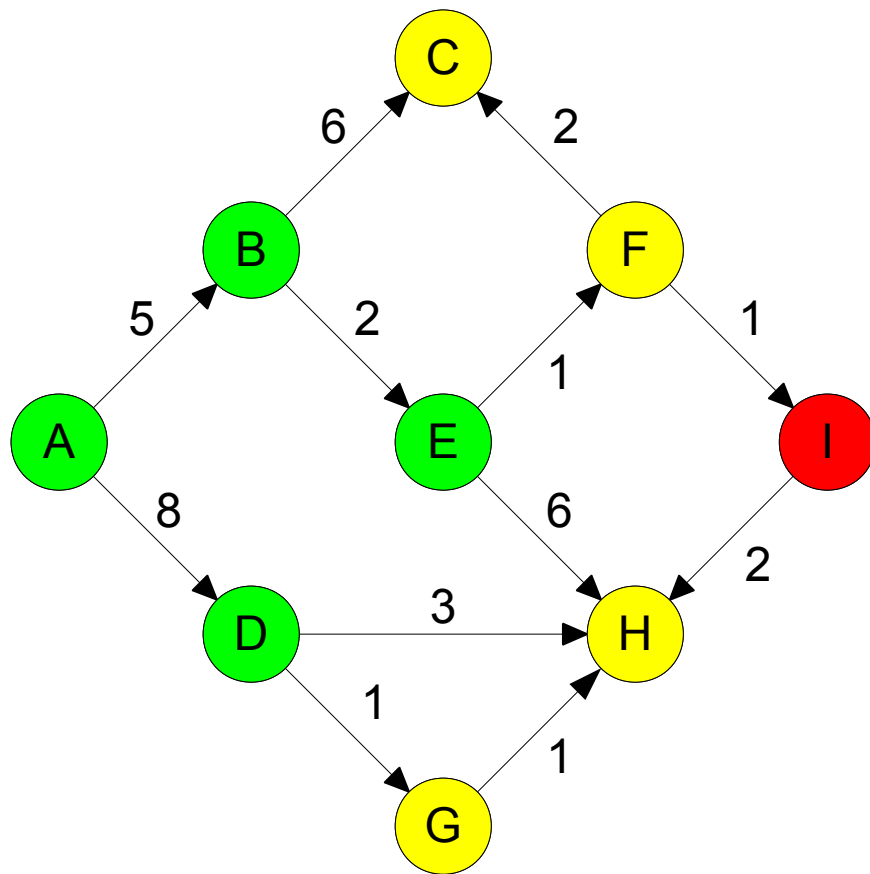
(8) A→B→E→F

(11) A→B→C

(13) A→B→E→H

(9) A→D→G

(11) A→D→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

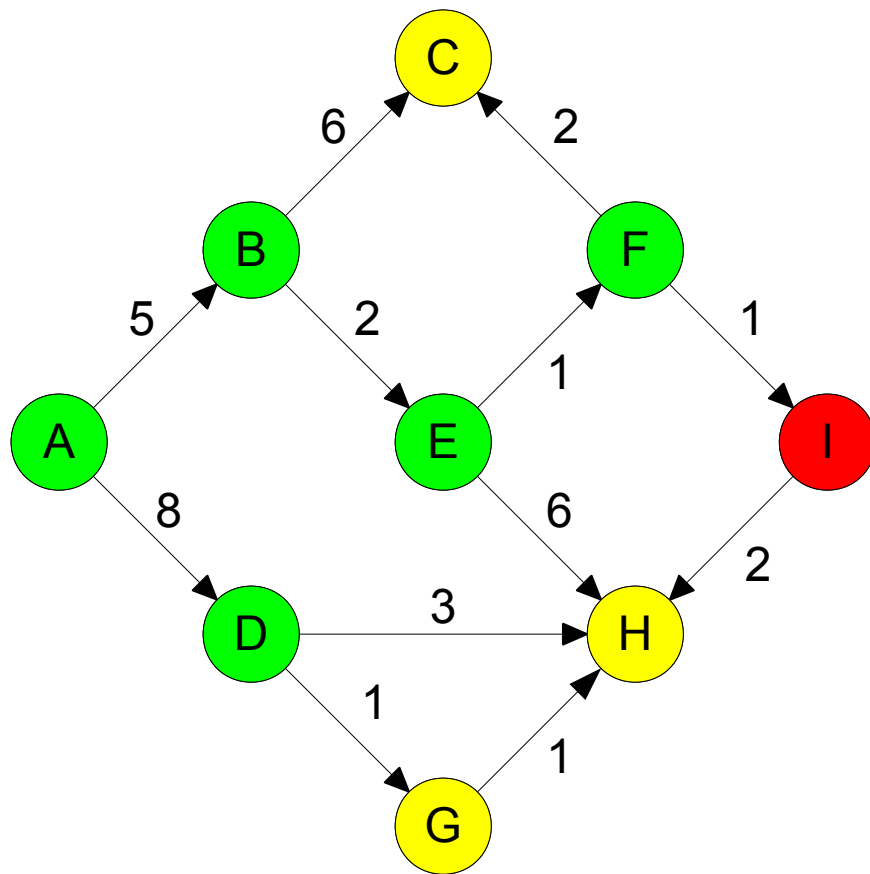
(8) A→B→E→F

(9) A→D→G

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

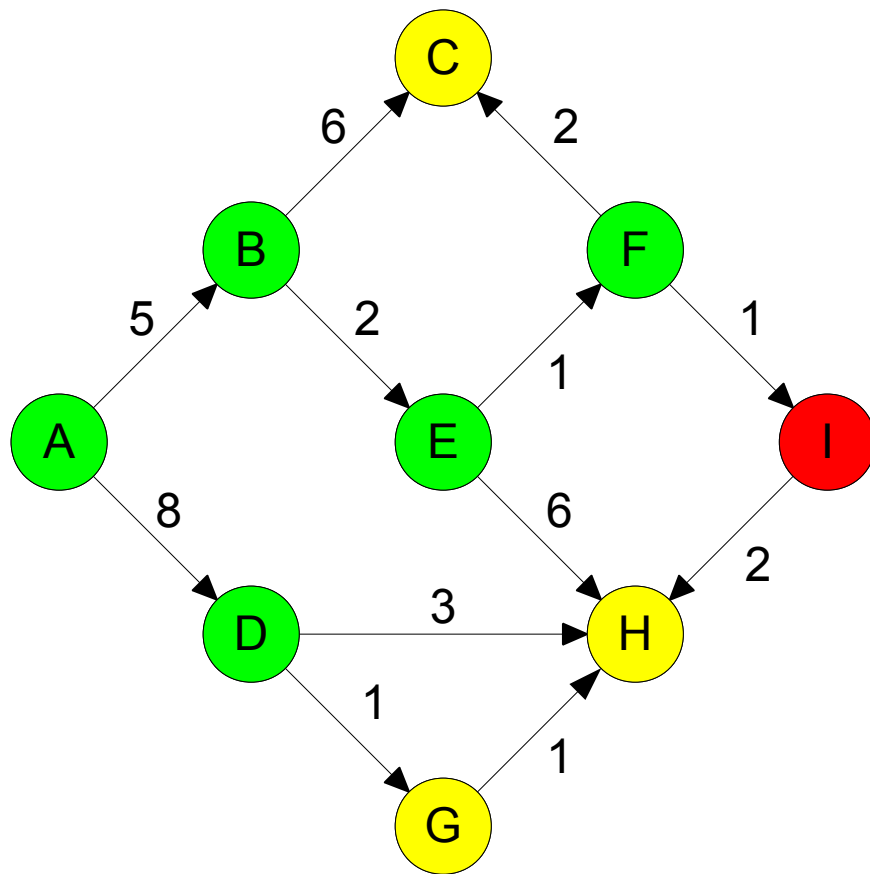
(9) A→D→G

(11) A→B→C

(11) A→D→H

(13) A→B→E→H





(0) A

(5) A→B

(7) A→B→E

(8) A→D

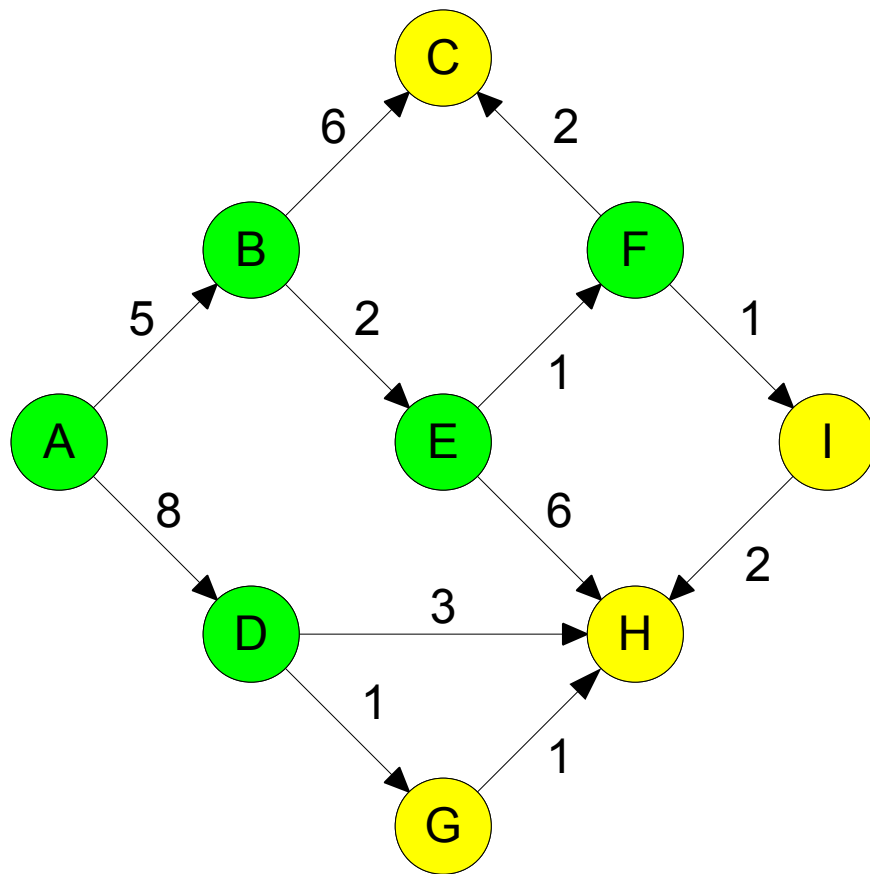
(8) A→B→E→F

(9) A→D→G

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5)  $A \rightarrow B$

(7)  $A \rightarrow B \rightarrow E$

(8)  $A \rightarrow D$

(8)  $A \rightarrow B \rightarrow E \rightarrow F$

(9)  $A \rightarrow D \rightarrow G$

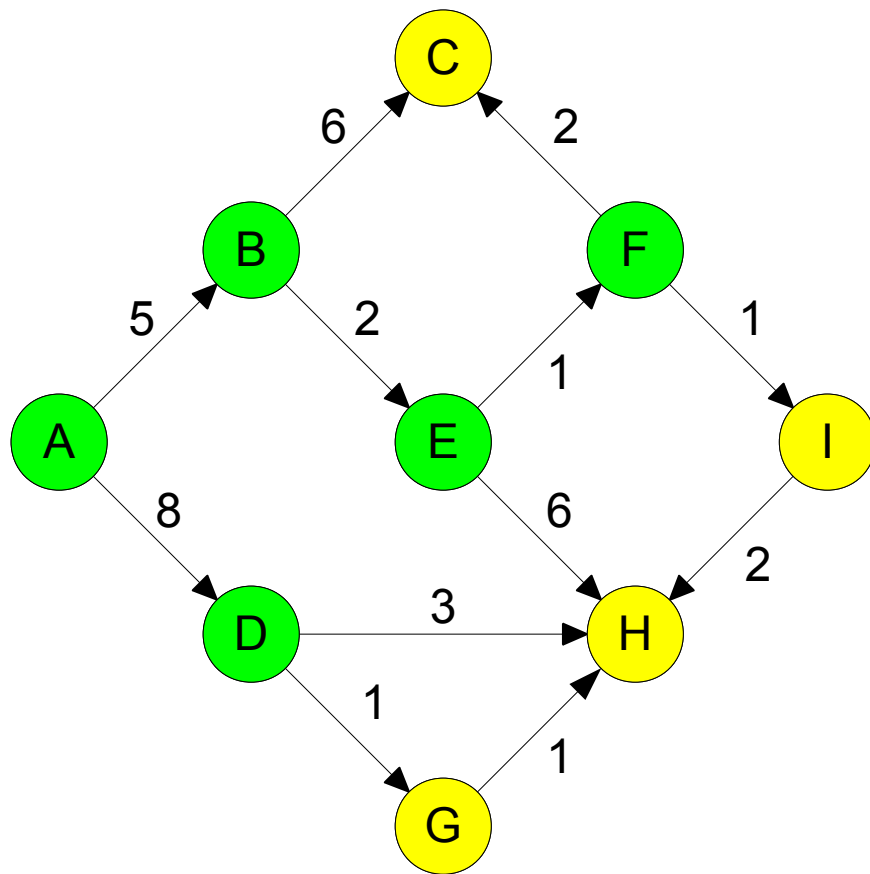
(11)  $A \rightarrow B \rightarrow C$

(11)  $A \rightarrow D \rightarrow H$

(13)  $A \rightarrow B \rightarrow E \rightarrow H$

(9)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow I$

(10)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow C$



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

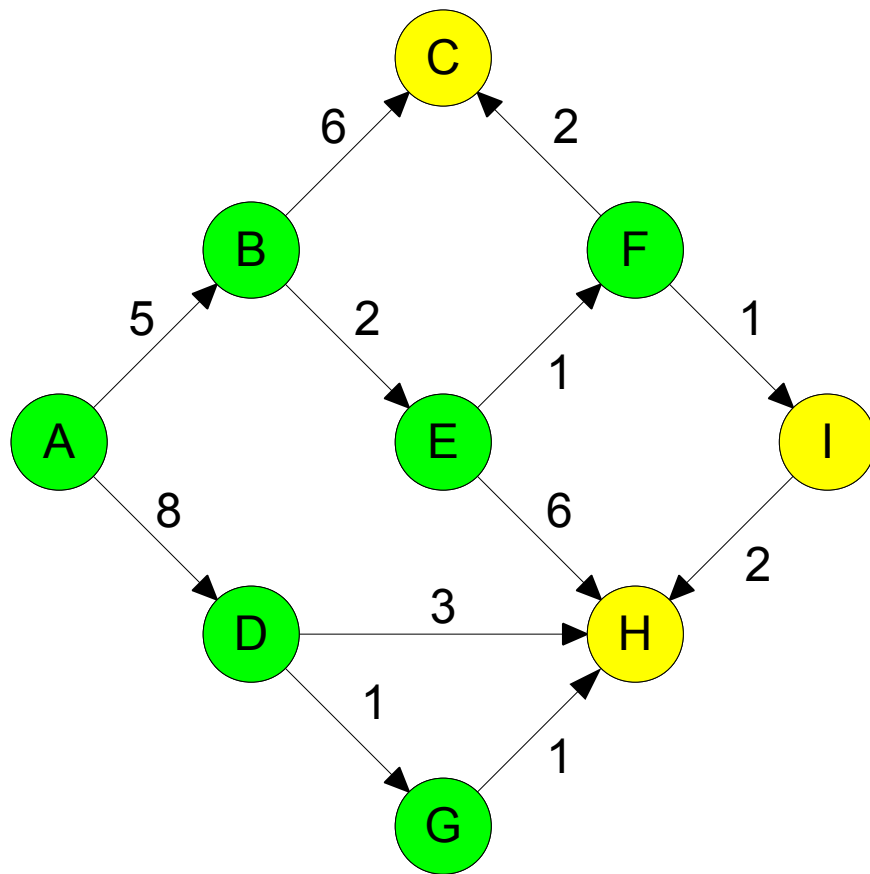
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

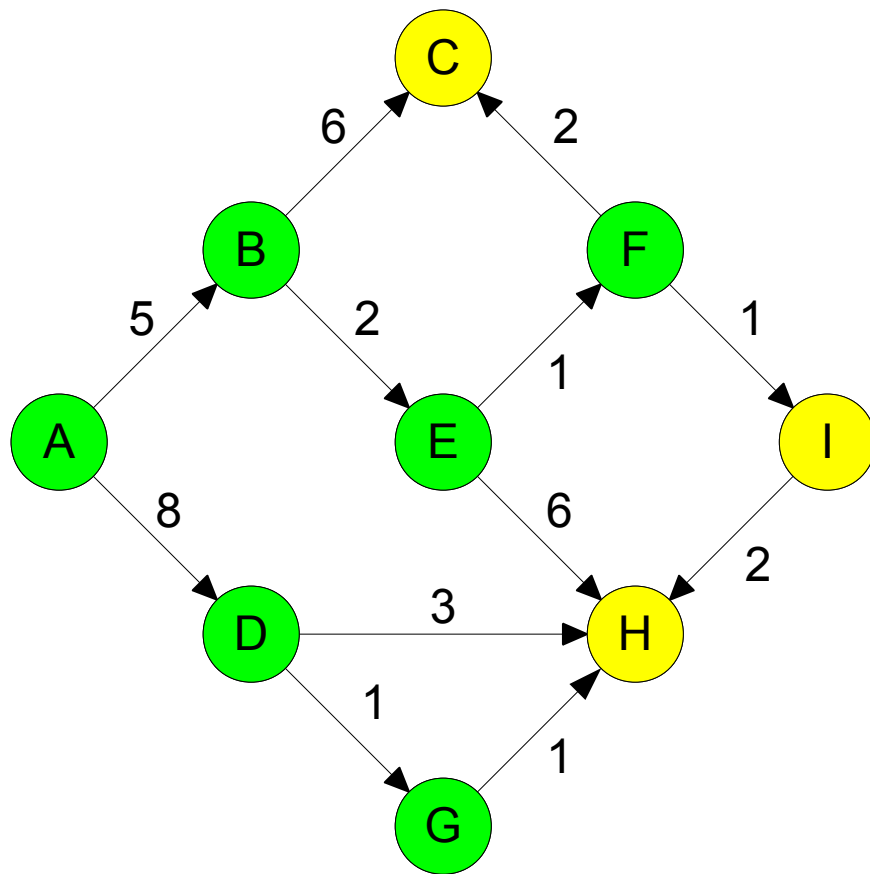
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

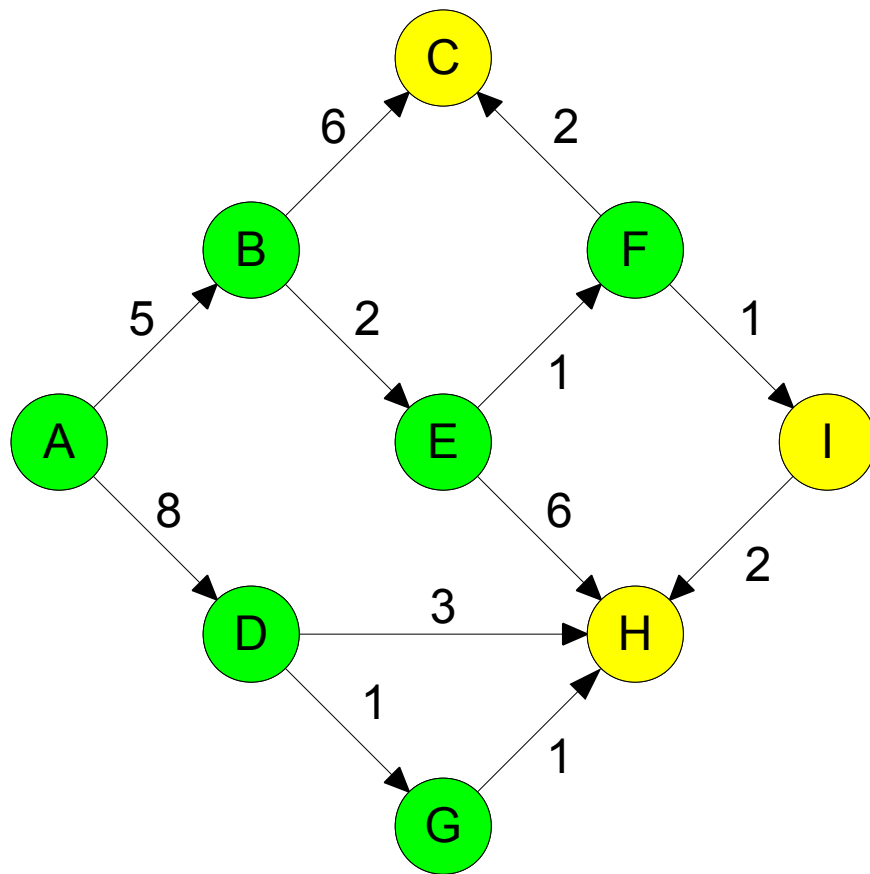
(9) A→B→E→F→I

(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

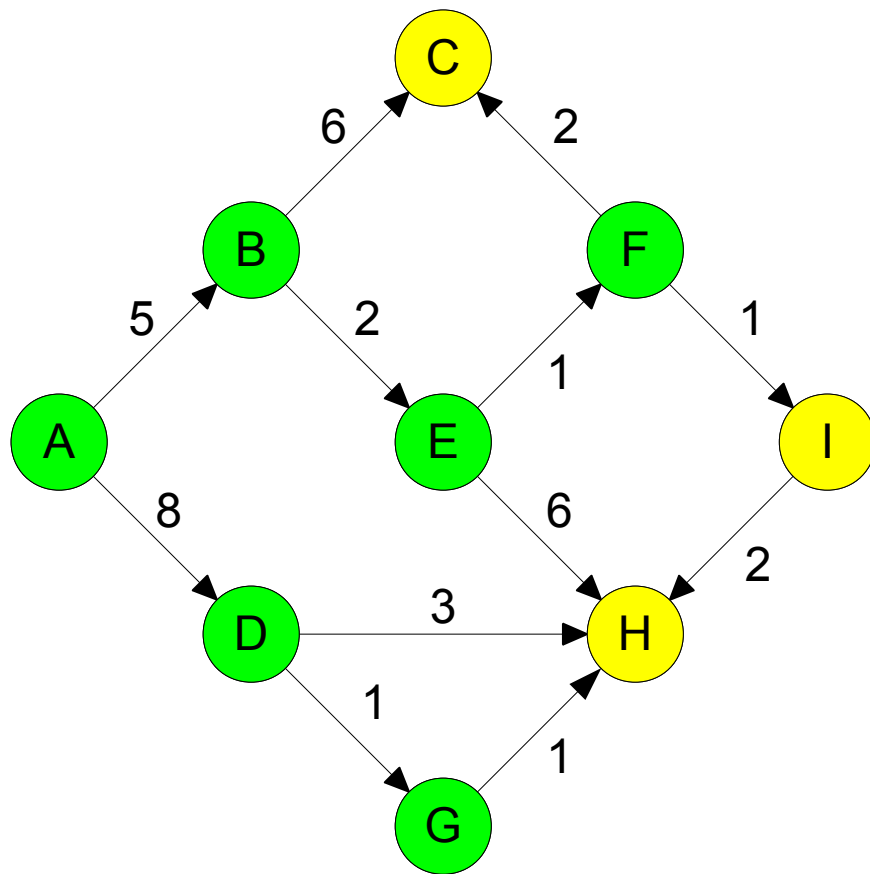
(10) A→B→E→F→C

(11) A→B→C

(11) A→D→H

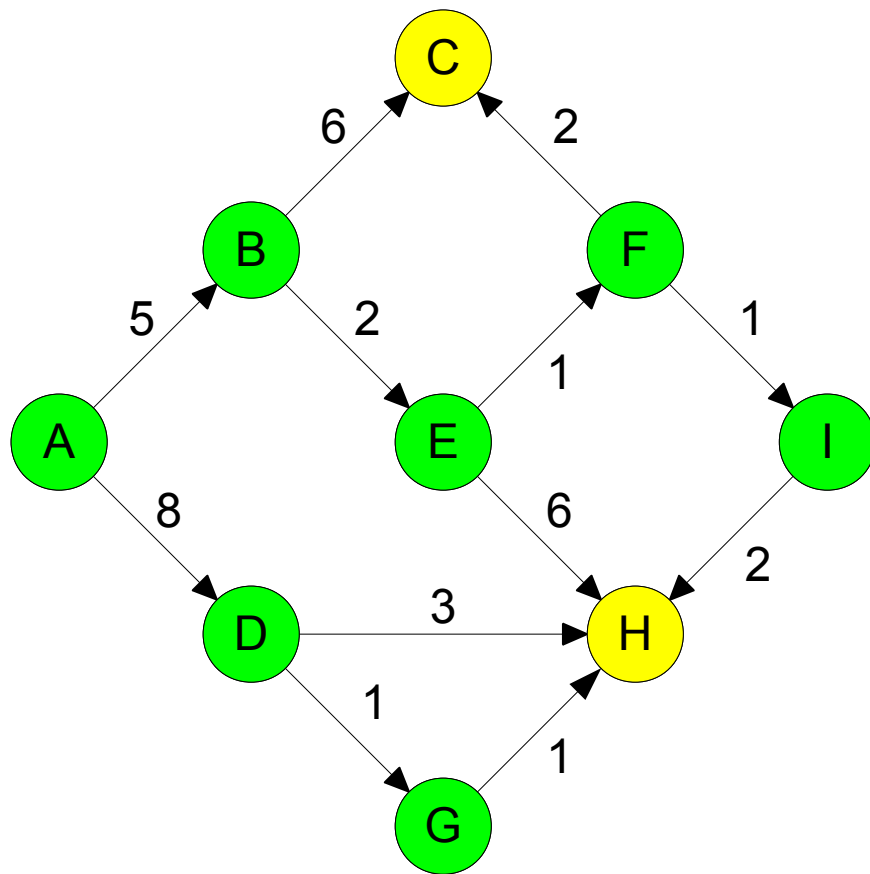
(13) A→B→E→H

(10) A→D→G→H



(0) A
(5) A→B
(7) A→B→E
(8) A→D
(8) A→B→E→F
(9) A→D→G

(9) A→B→E→F→I
(10) A→B→E→F→C
(10) A→D→G→H
(11) A→B→C
(11) A→D→H
(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

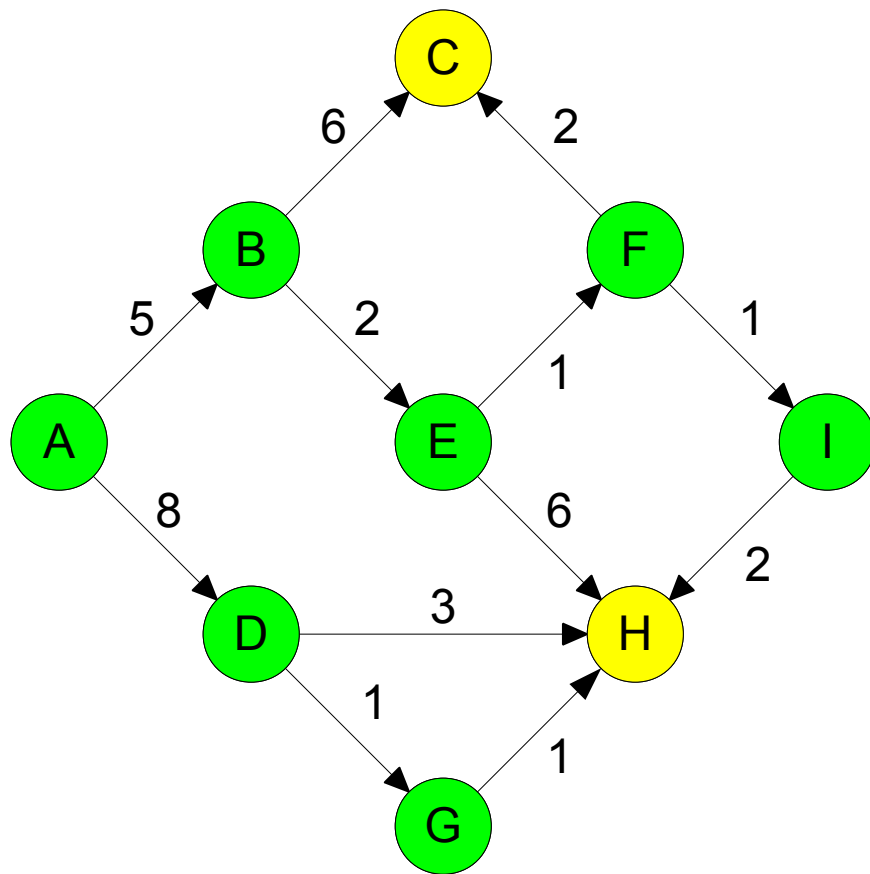
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(13) A→B→E→H





(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

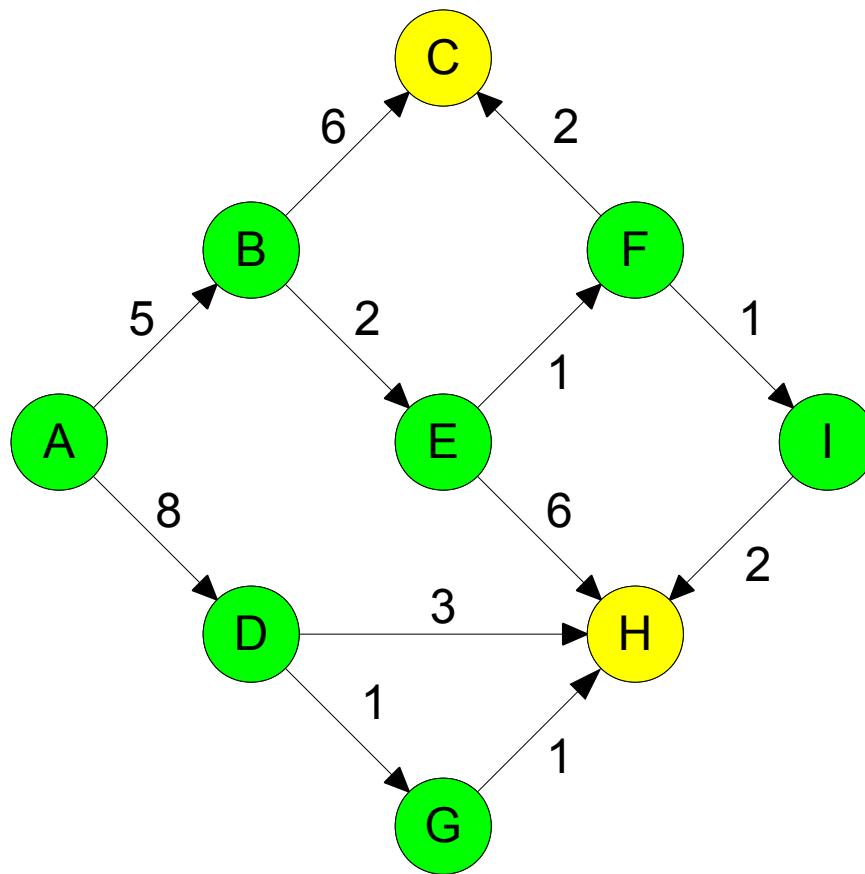
(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

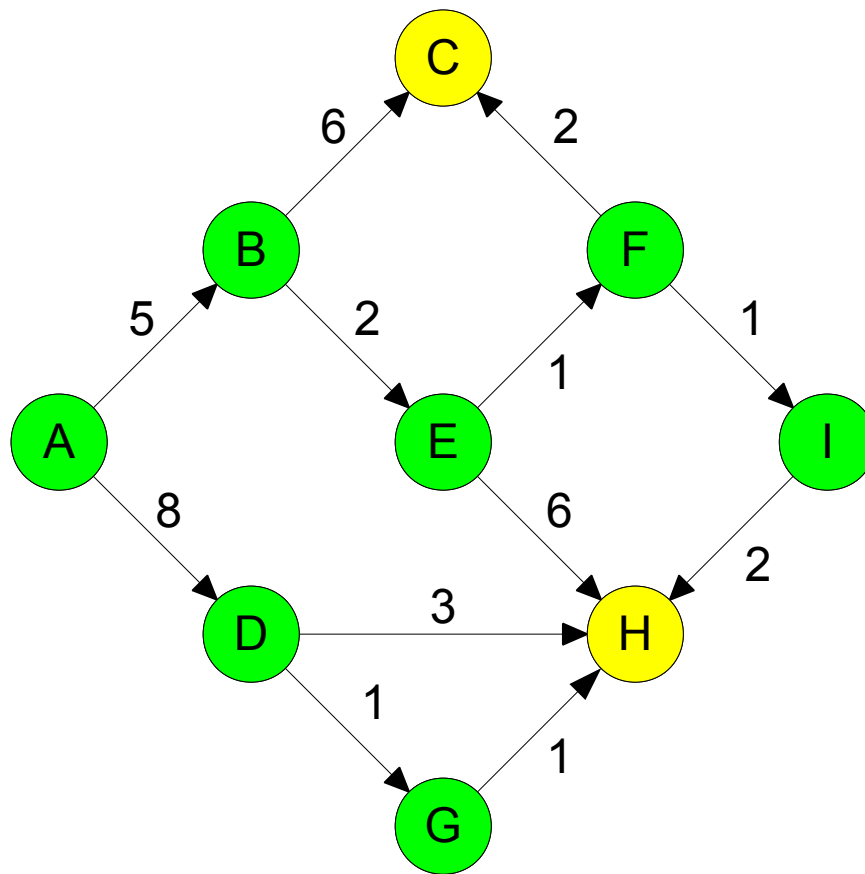
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(13) A→B→E→H

(11) A→B→E→F→I→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

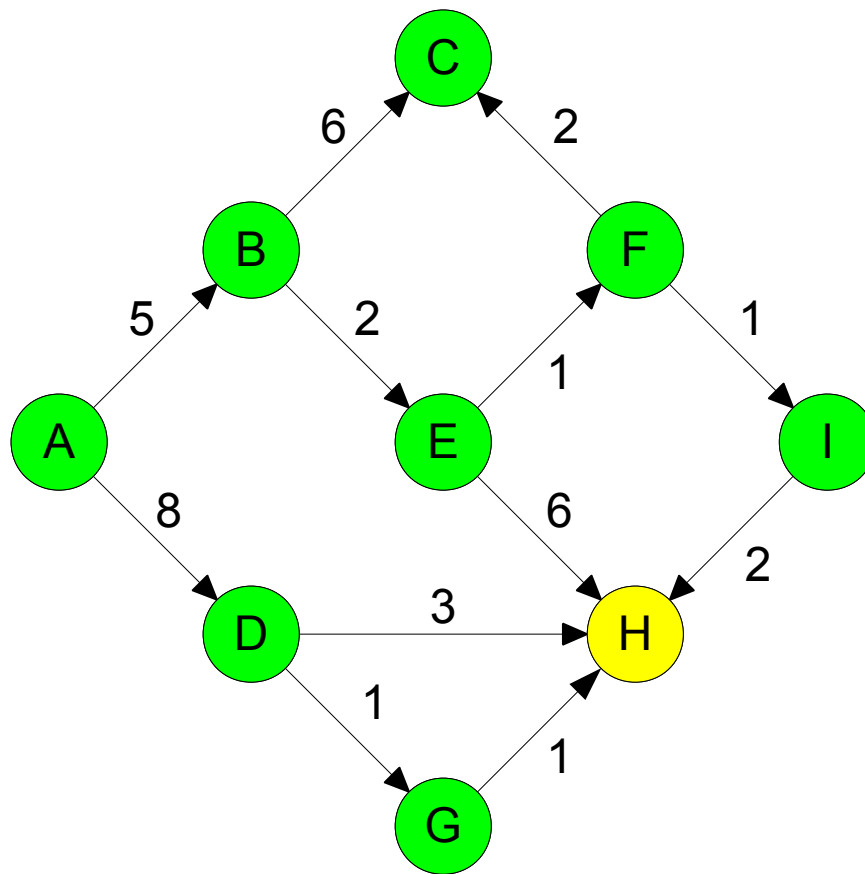
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

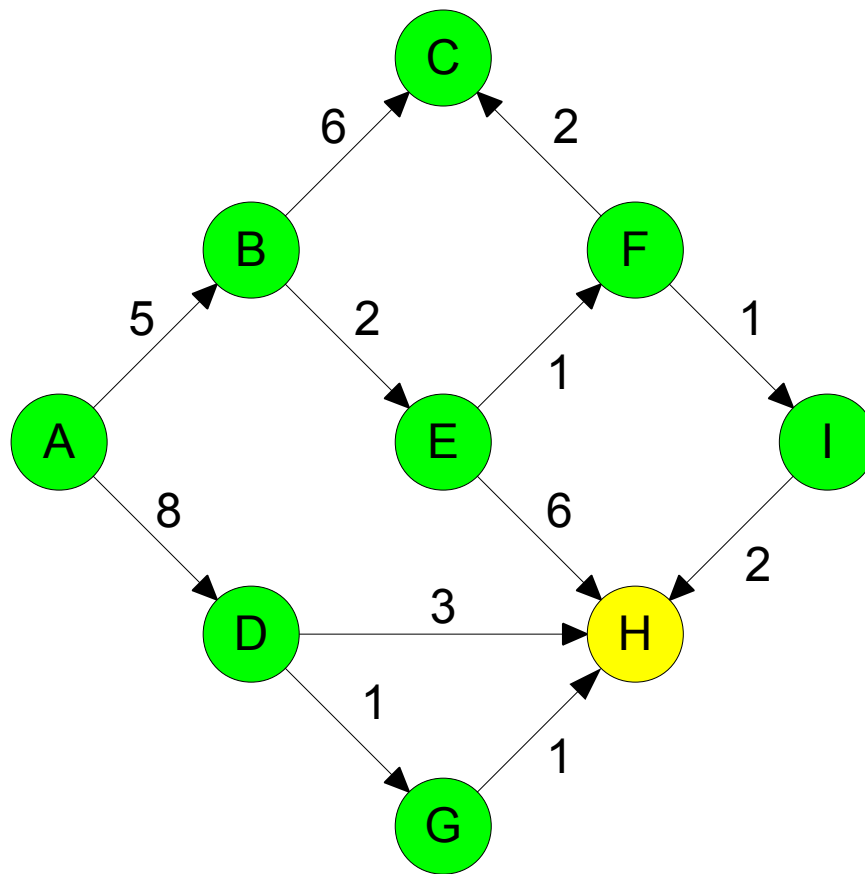
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

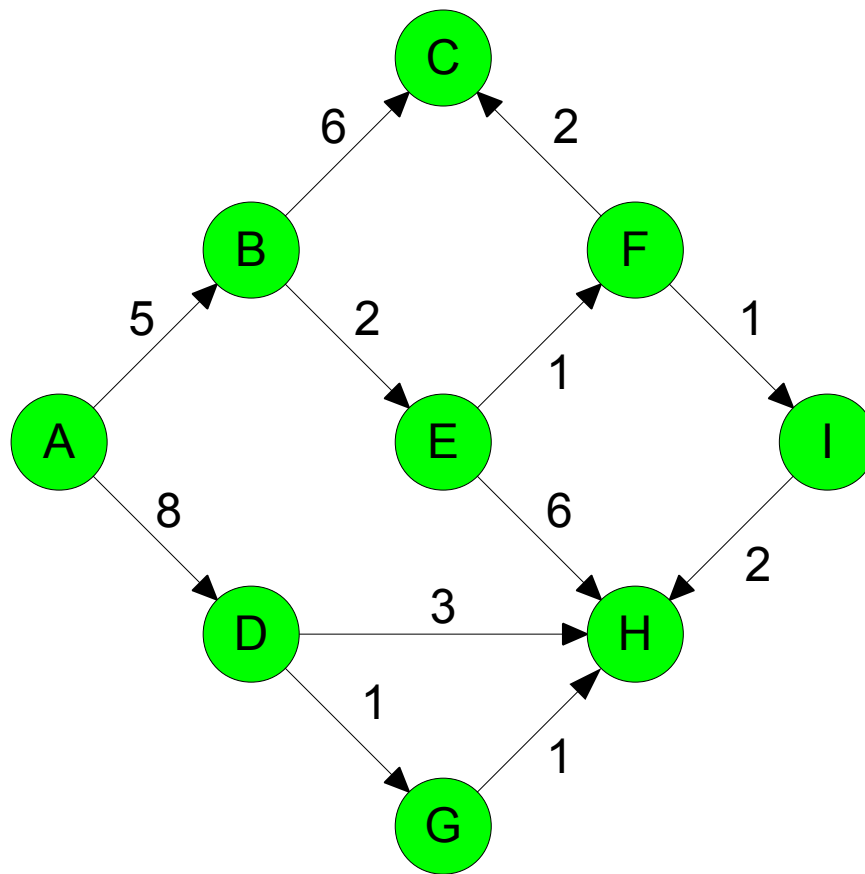
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

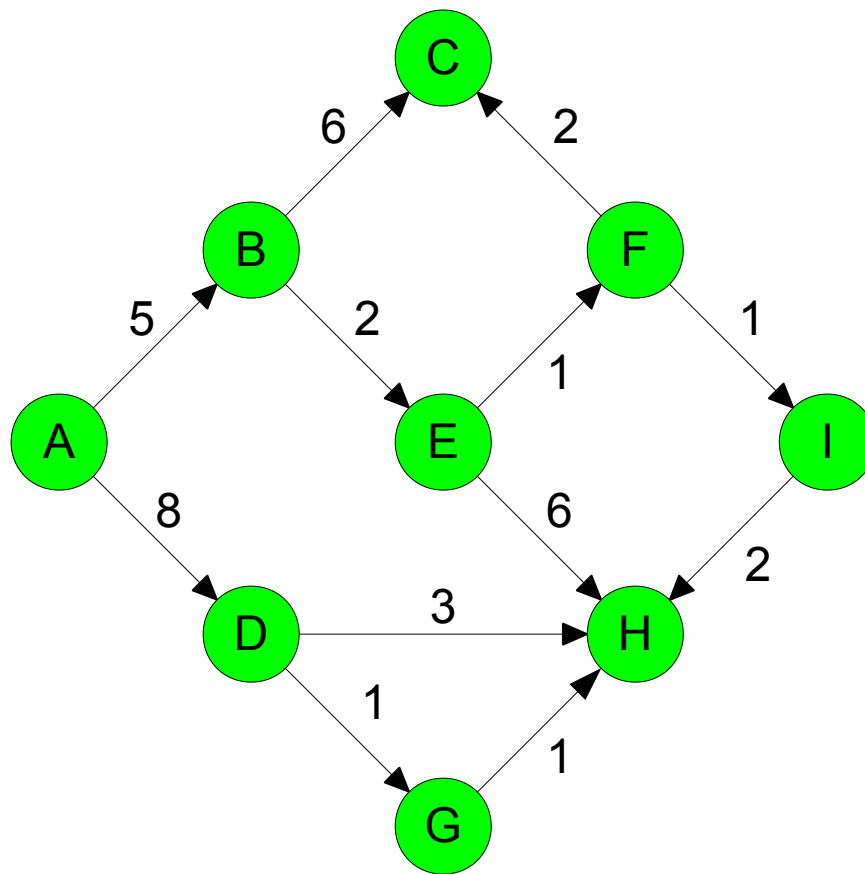
(10) A→D→G→H

(11) A→B→C

(11) A→D→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5)  $A \rightarrow B$

(7)  $A \rightarrow B \rightarrow E$

(8)  $A \rightarrow D$

(8)  $A \rightarrow B \rightarrow E \rightarrow F$

(9)  $A \rightarrow D \rightarrow G$

(9)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow I$

(10)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow C$

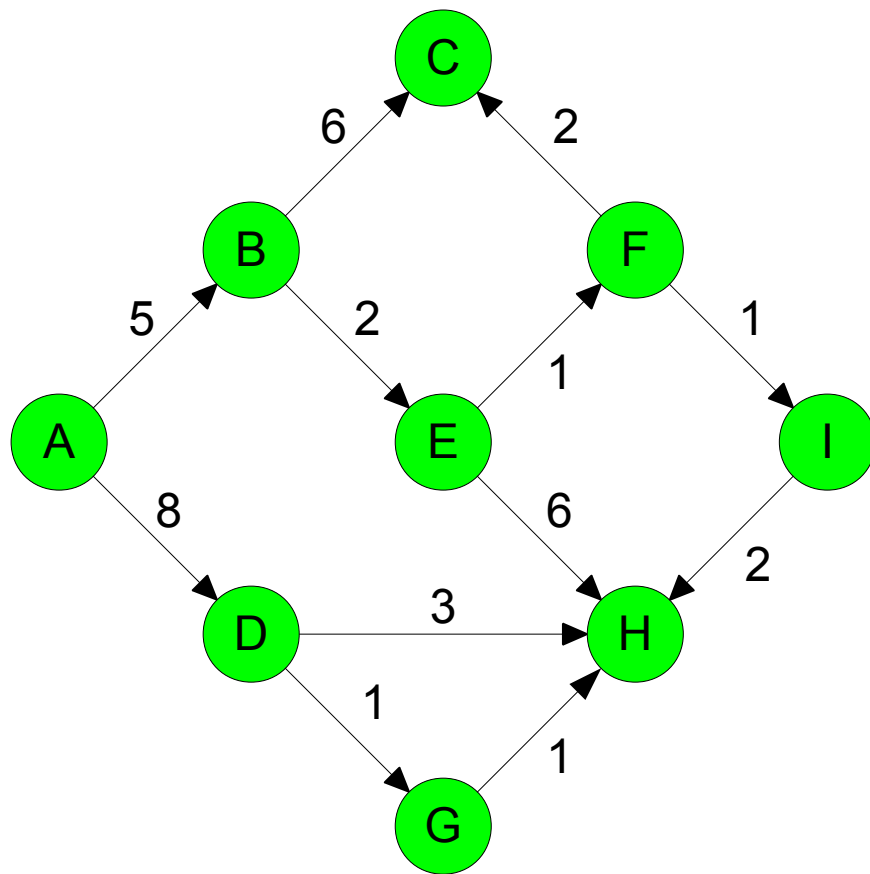
(10)  $A \rightarrow D \rightarrow G \rightarrow H$

(11)  $A \rightarrow B \rightarrow C$

(11)  $A \rightarrow D \rightarrow H$

(11)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow I \rightarrow H$

(13)  $A \rightarrow B \rightarrow E \rightarrow H$



(0) A

(5)  $A \rightarrow B$

(7)  $A \rightarrow B \rightarrow E$

(8)  $A \rightarrow D$

(8)  $A \rightarrow B \rightarrow E \rightarrow F$

(9)  $A \rightarrow D \rightarrow G$

(9)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow I$

(10)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow C$

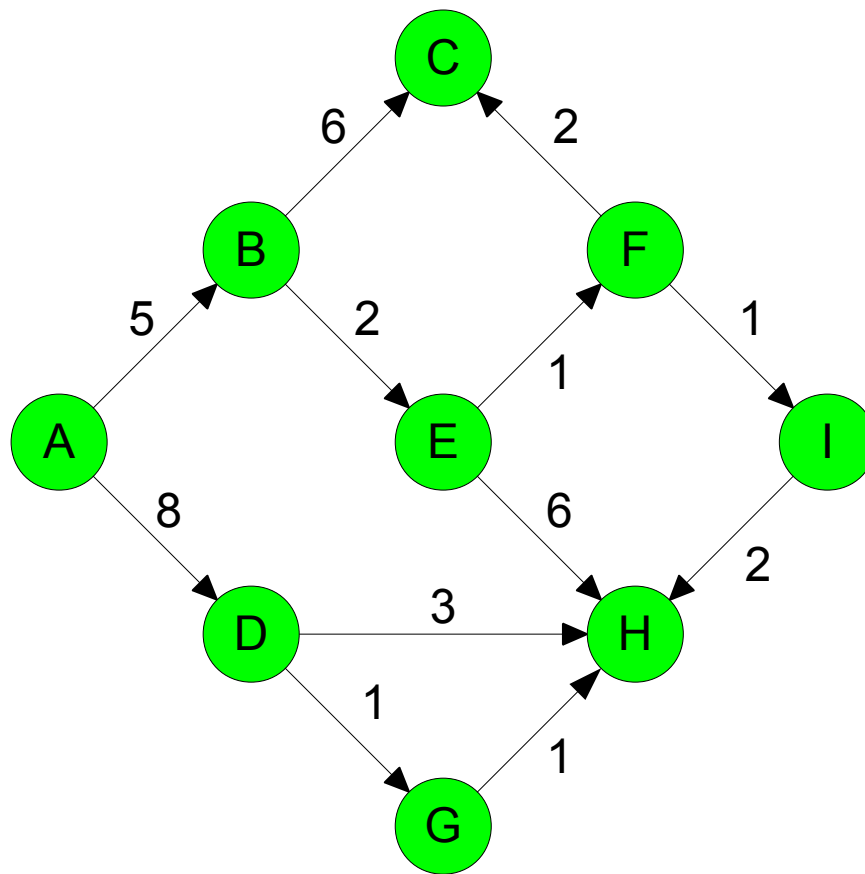
(10)  $A \rightarrow D \rightarrow G \rightarrow H$

(11)  $A \rightarrow D \rightarrow H$

(11)  $A \rightarrow B \rightarrow E \rightarrow F \rightarrow I \rightarrow H$

(13)  $A \rightarrow B \rightarrow E \rightarrow H$





(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

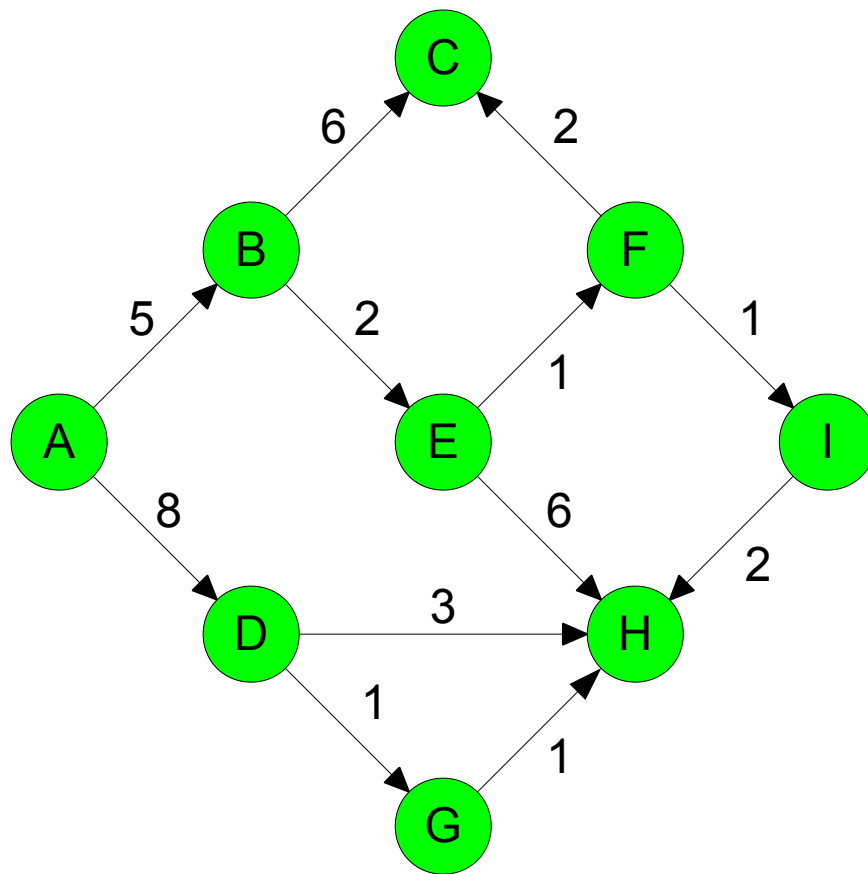
(9) A→B→E→F→I

(10) A→B→E→F→C

(10) A→D→G→H

(11) A→B→E→F→I→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

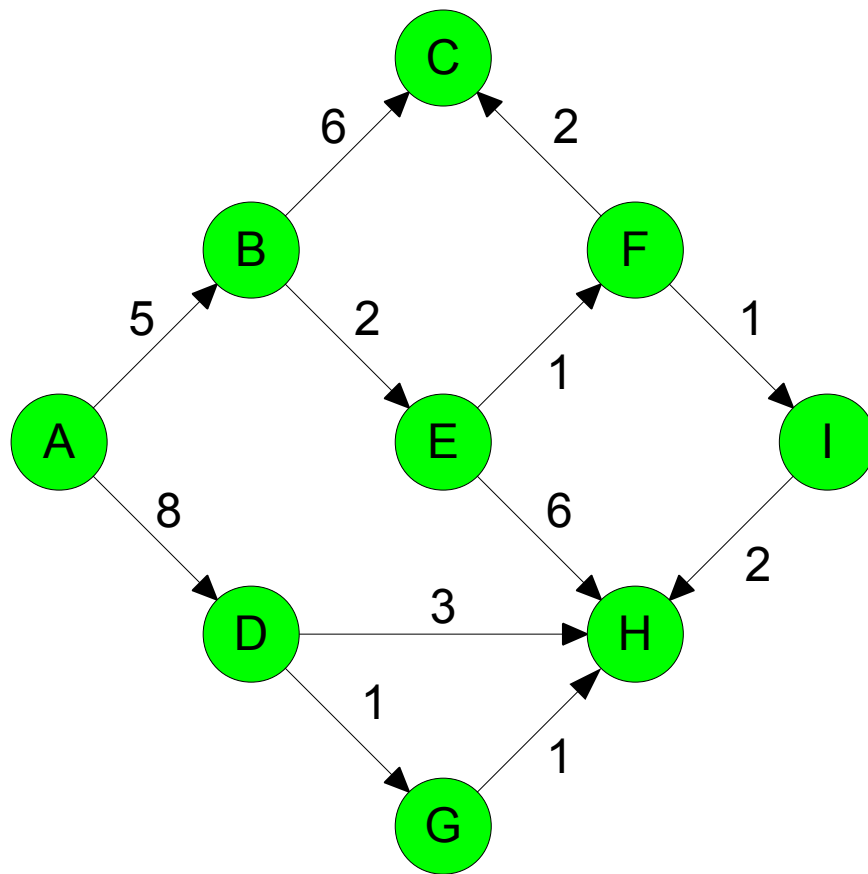
(9) A→D→G

(9) A→B→E→F→I

(10) A→B→E→F→C

(10) A→D→G→H

(13) A→B→E→H



(0) A

(5) A→B

(7) A→B→E

(8) A→D

(8) A→B→E→F

(9) A→D→G

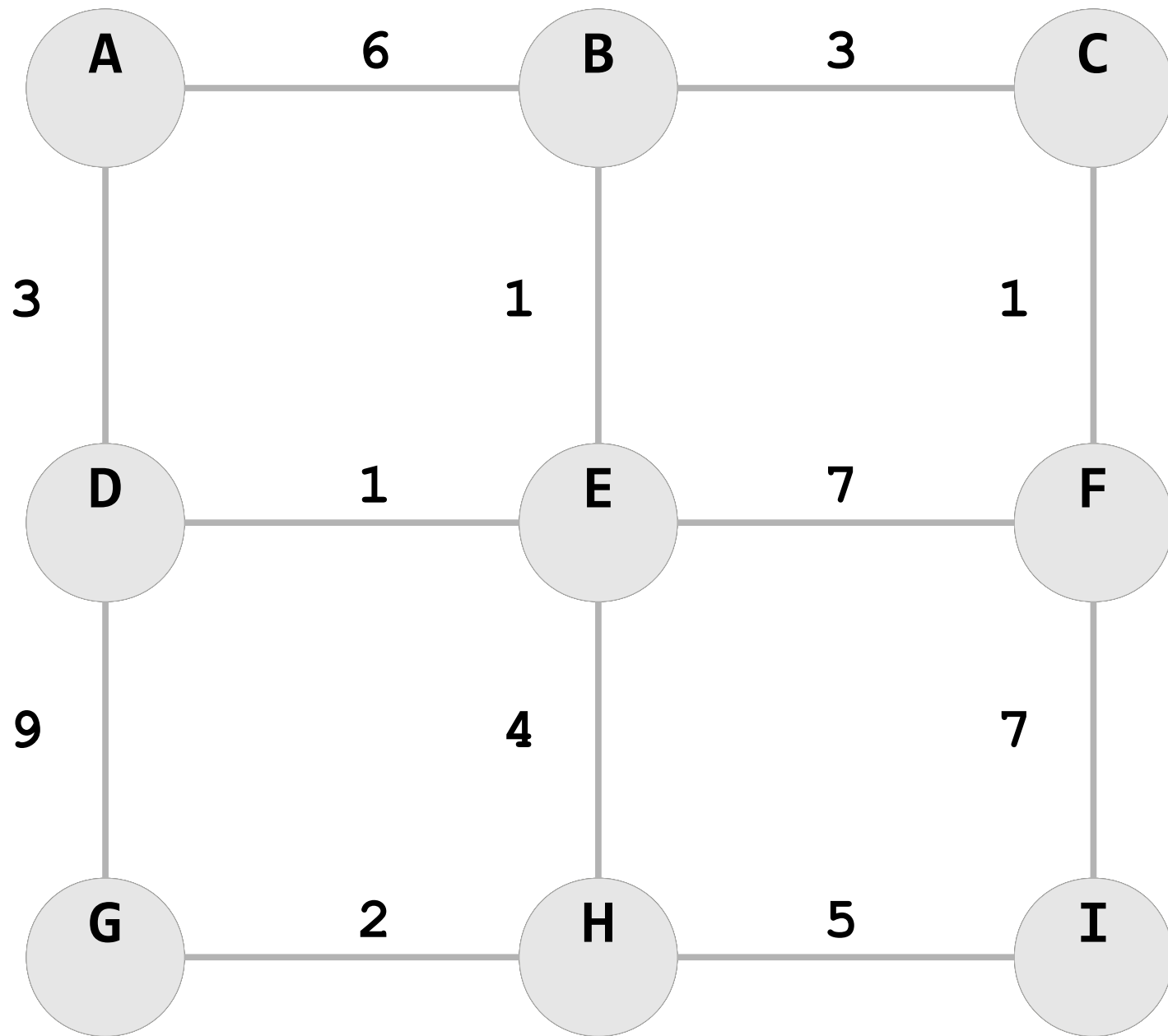
(9) A→B→E→F→I

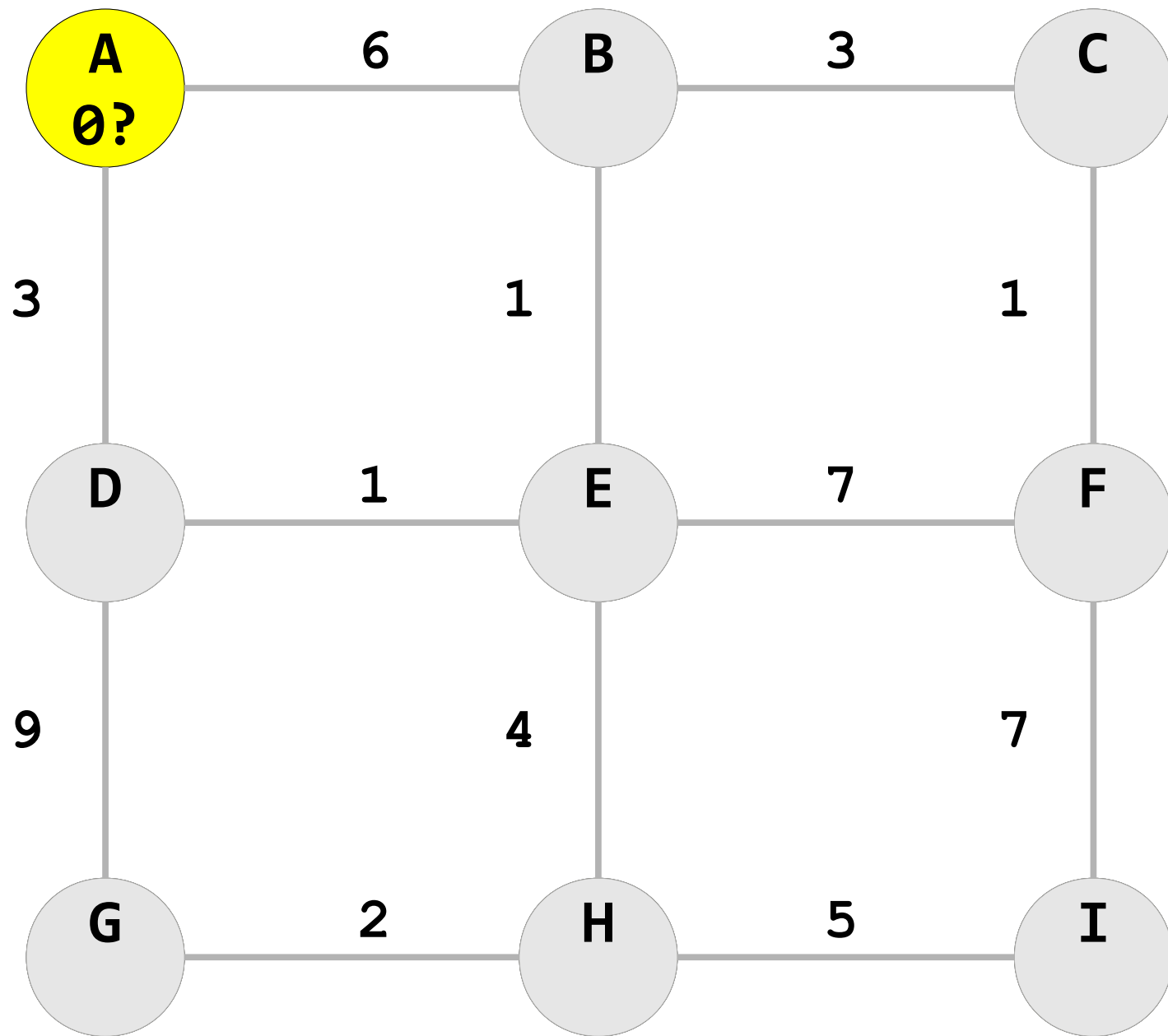
(10) A→B→E→F→C

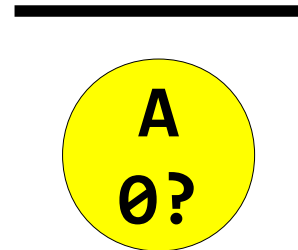
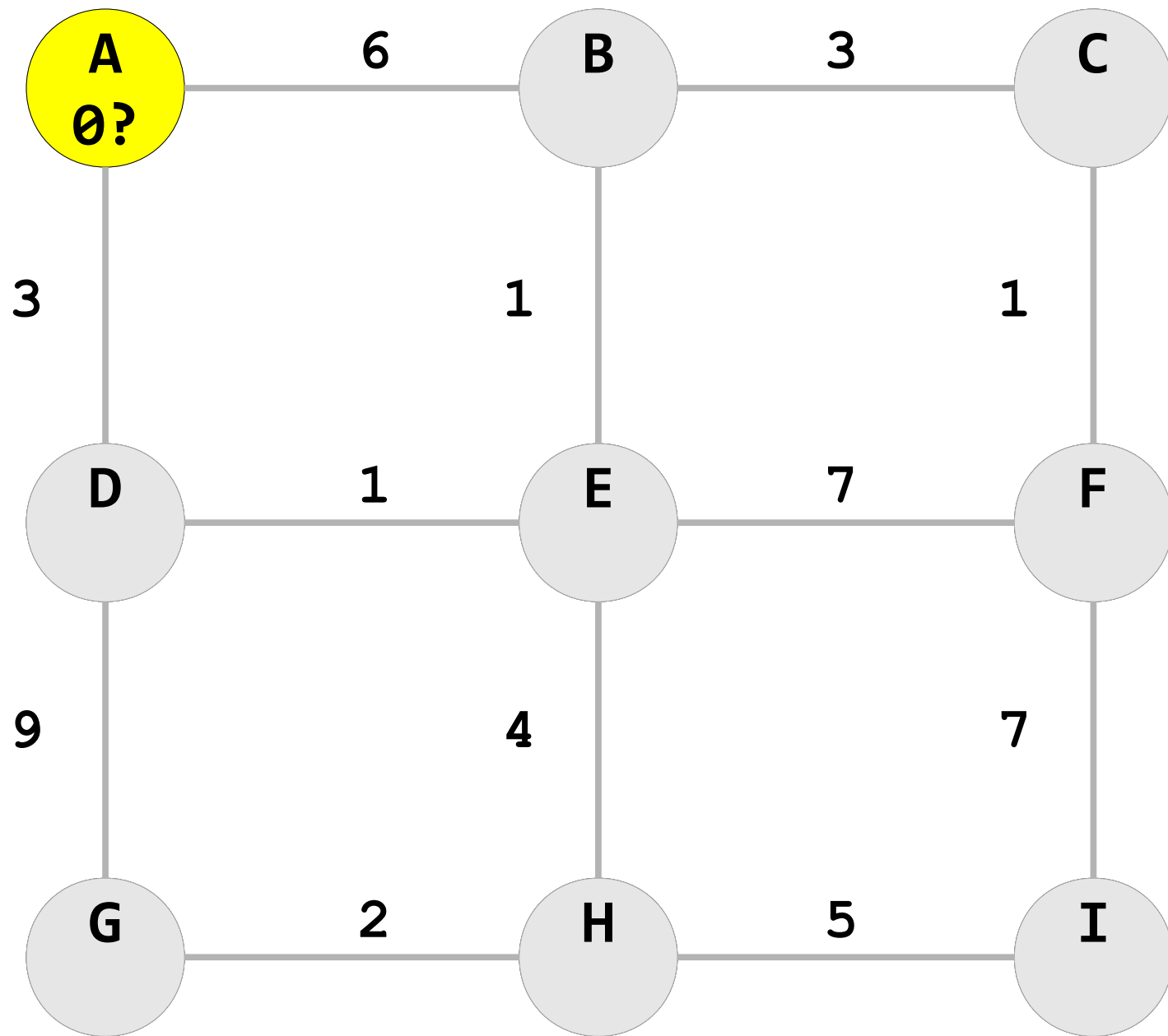
(10) A→D→G→H

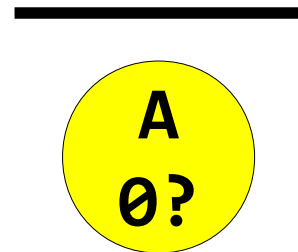
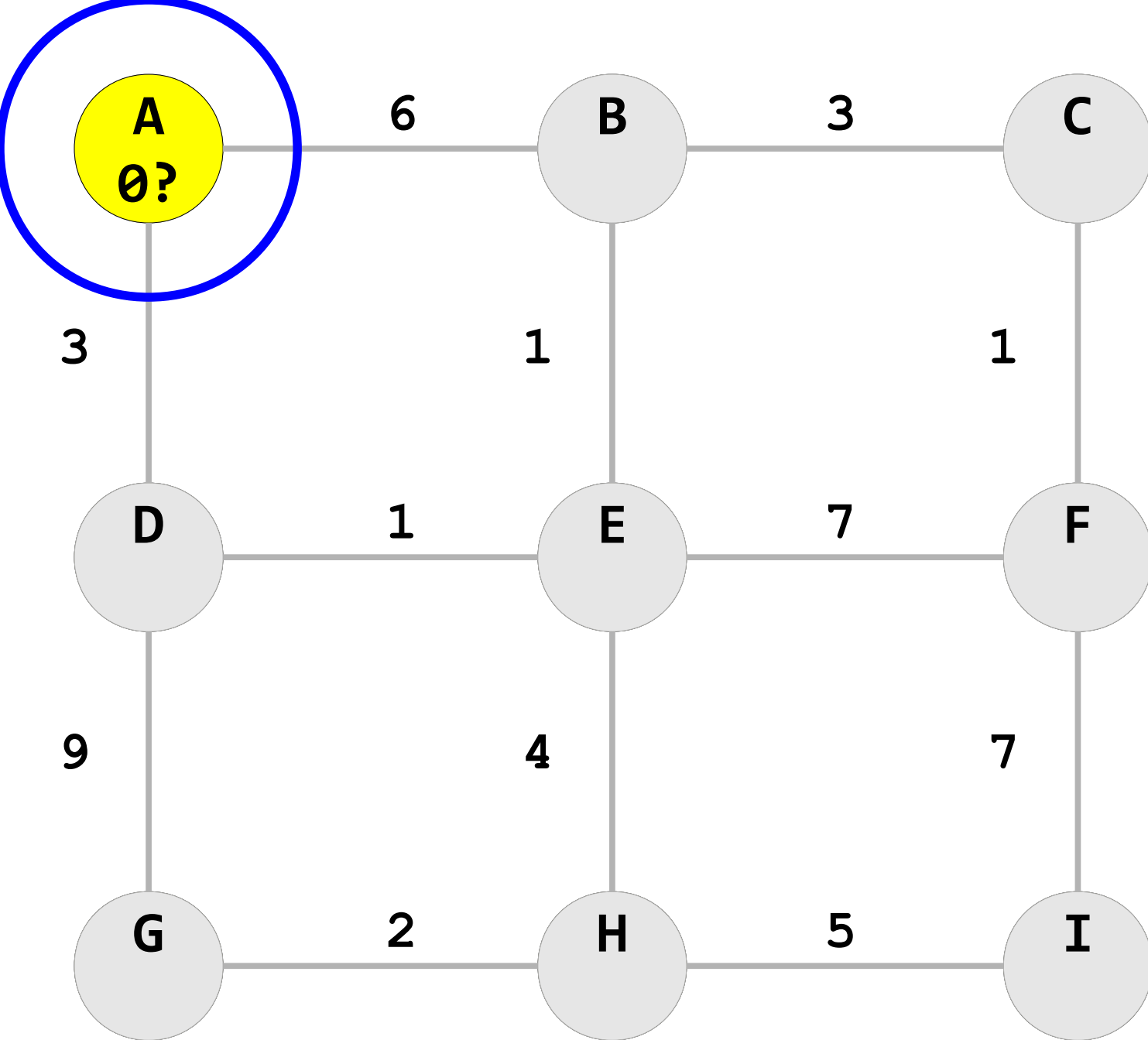
The MUCH Faster Way of Doing It

Also the way you're going to do it for  
Assignment 6 =)

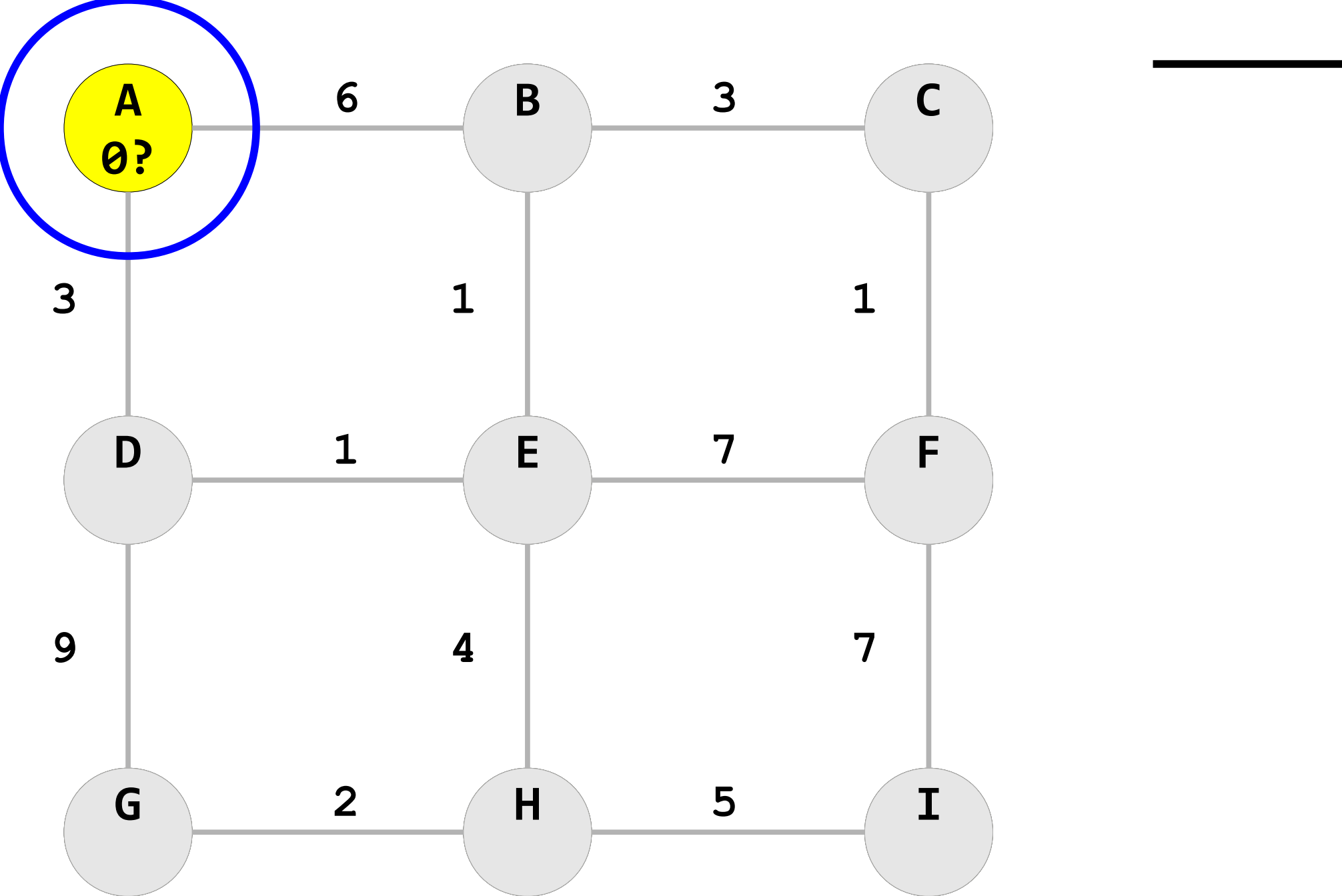


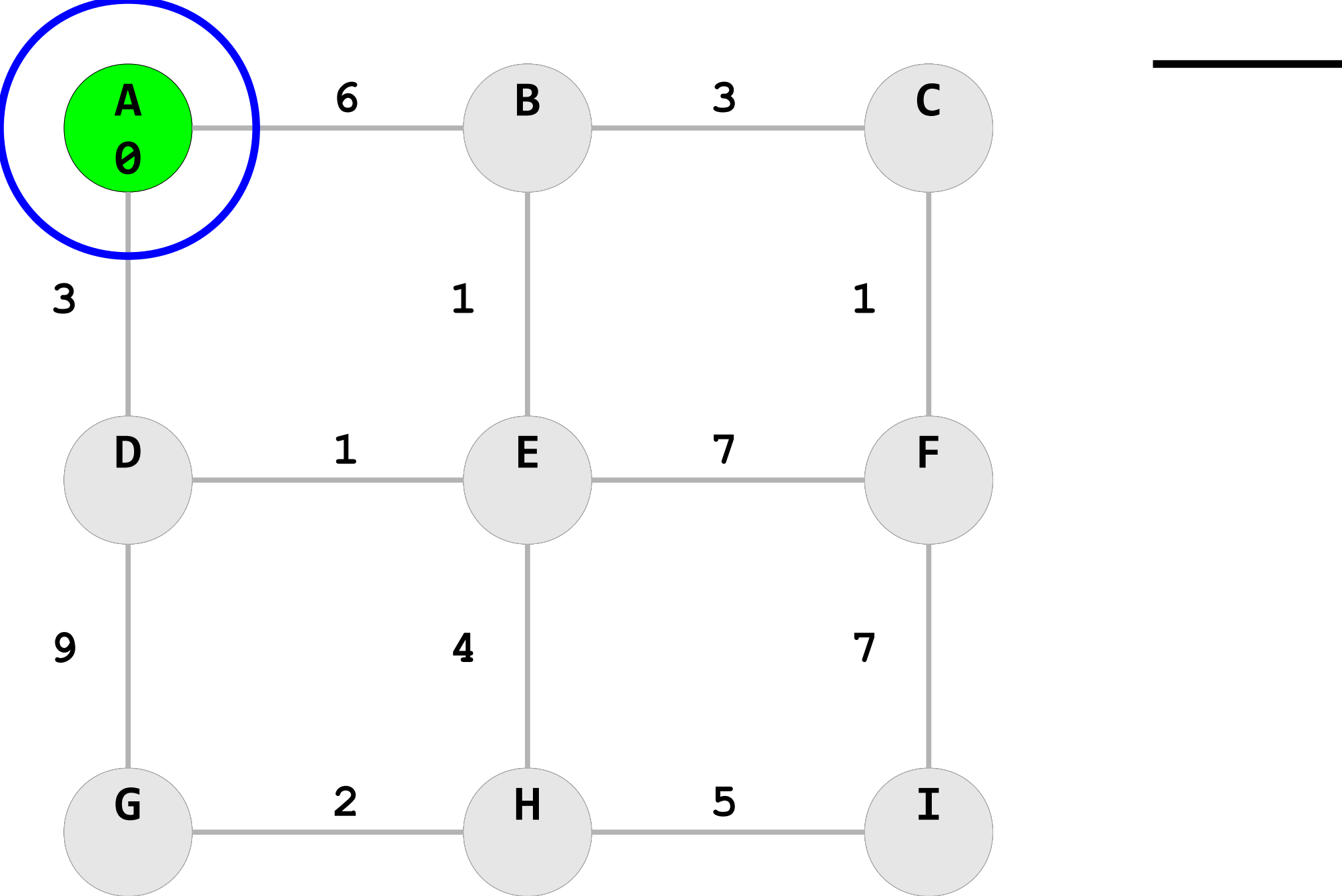


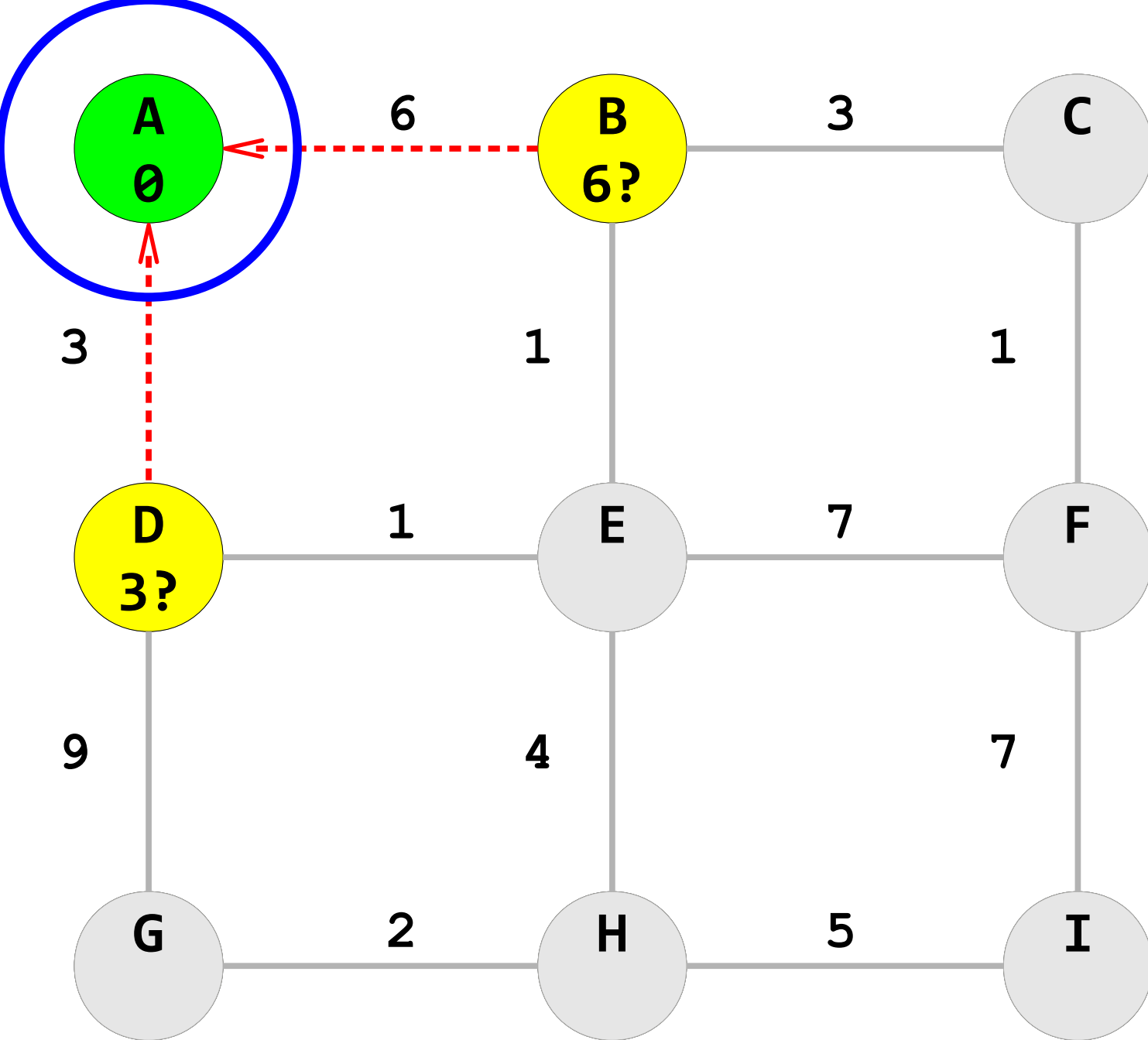


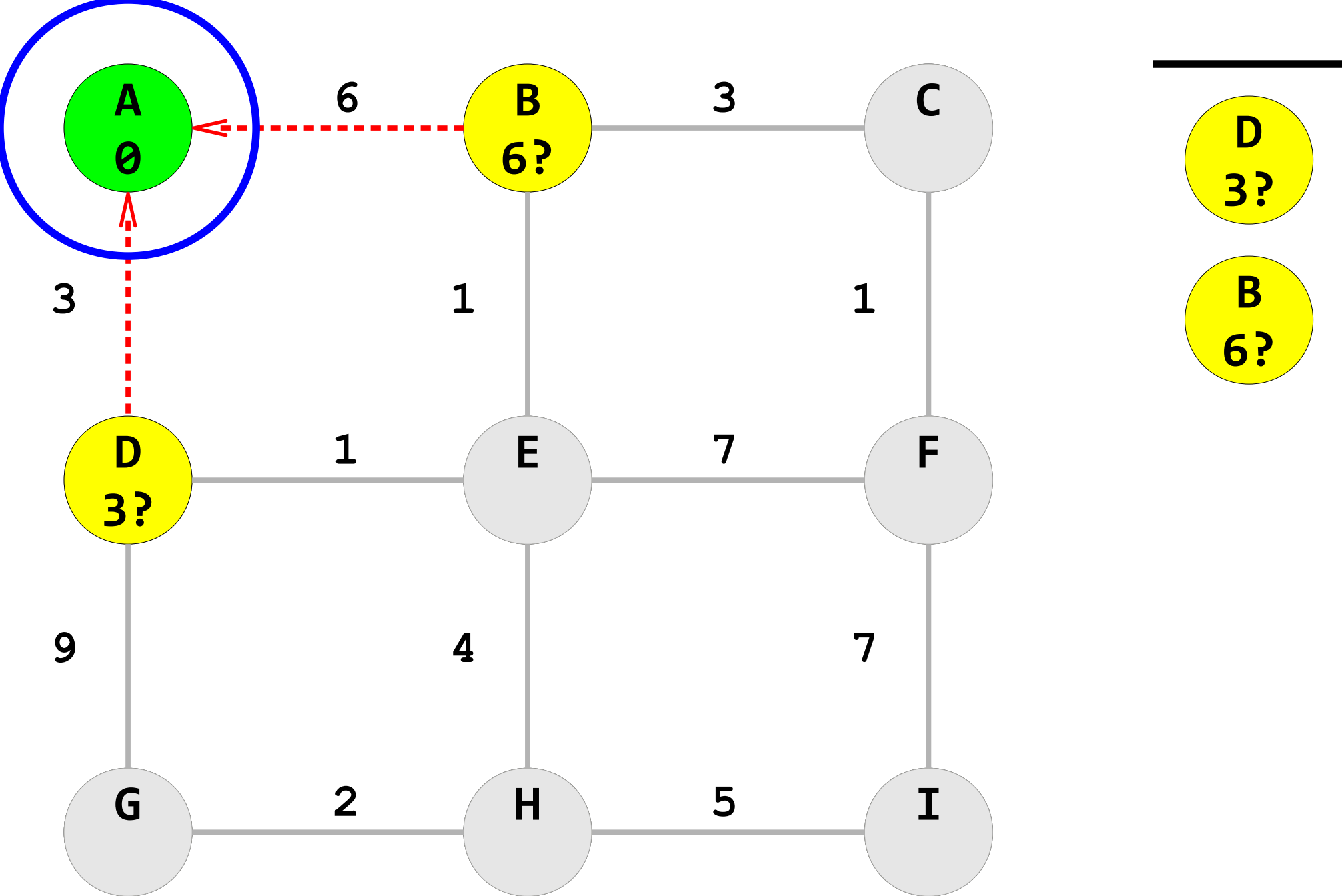


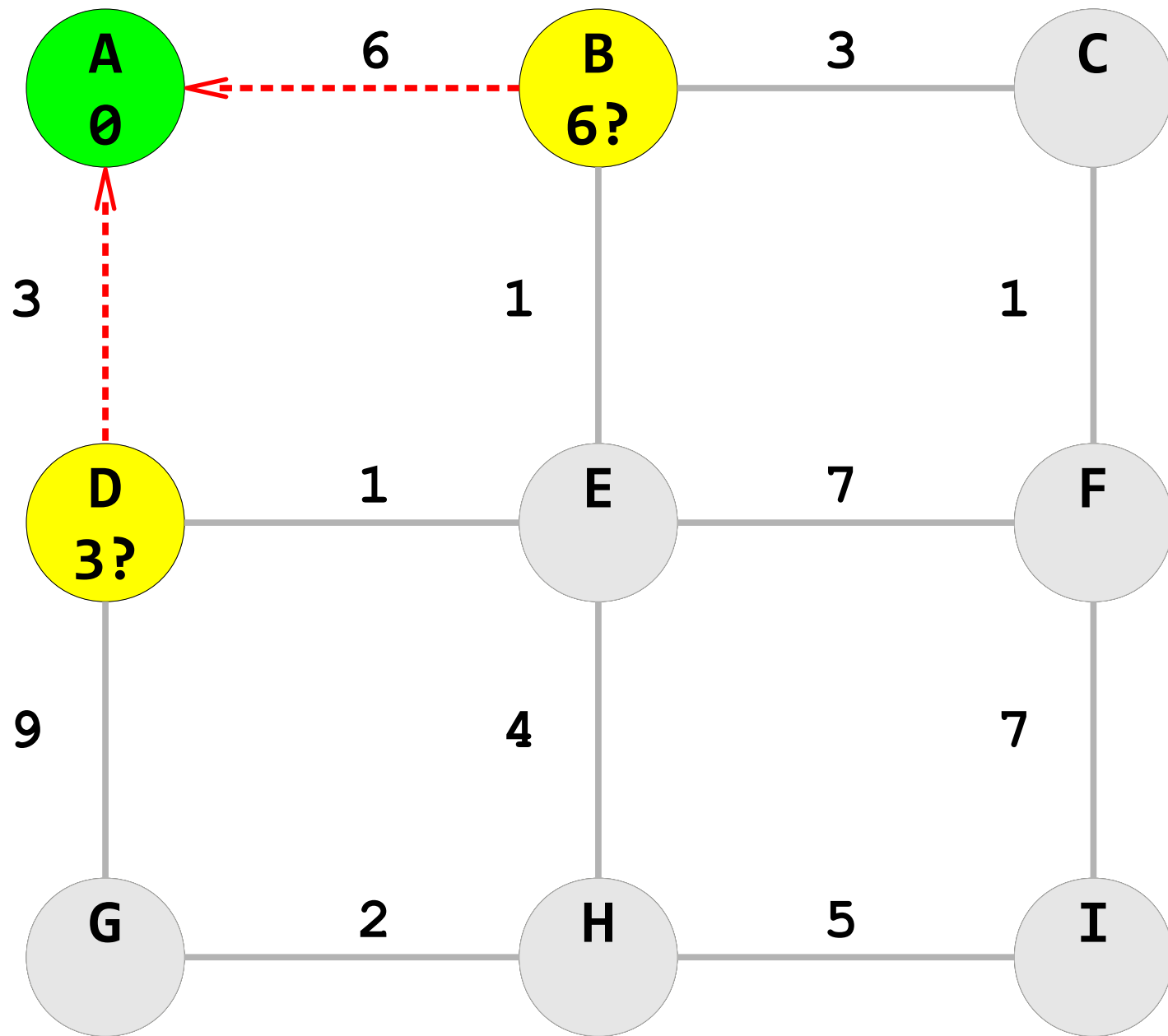








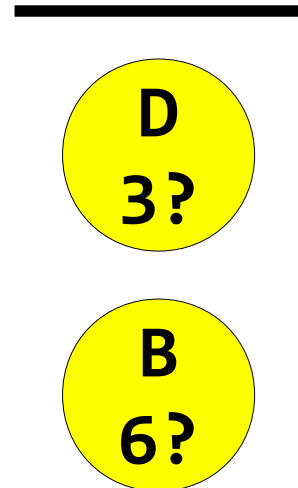
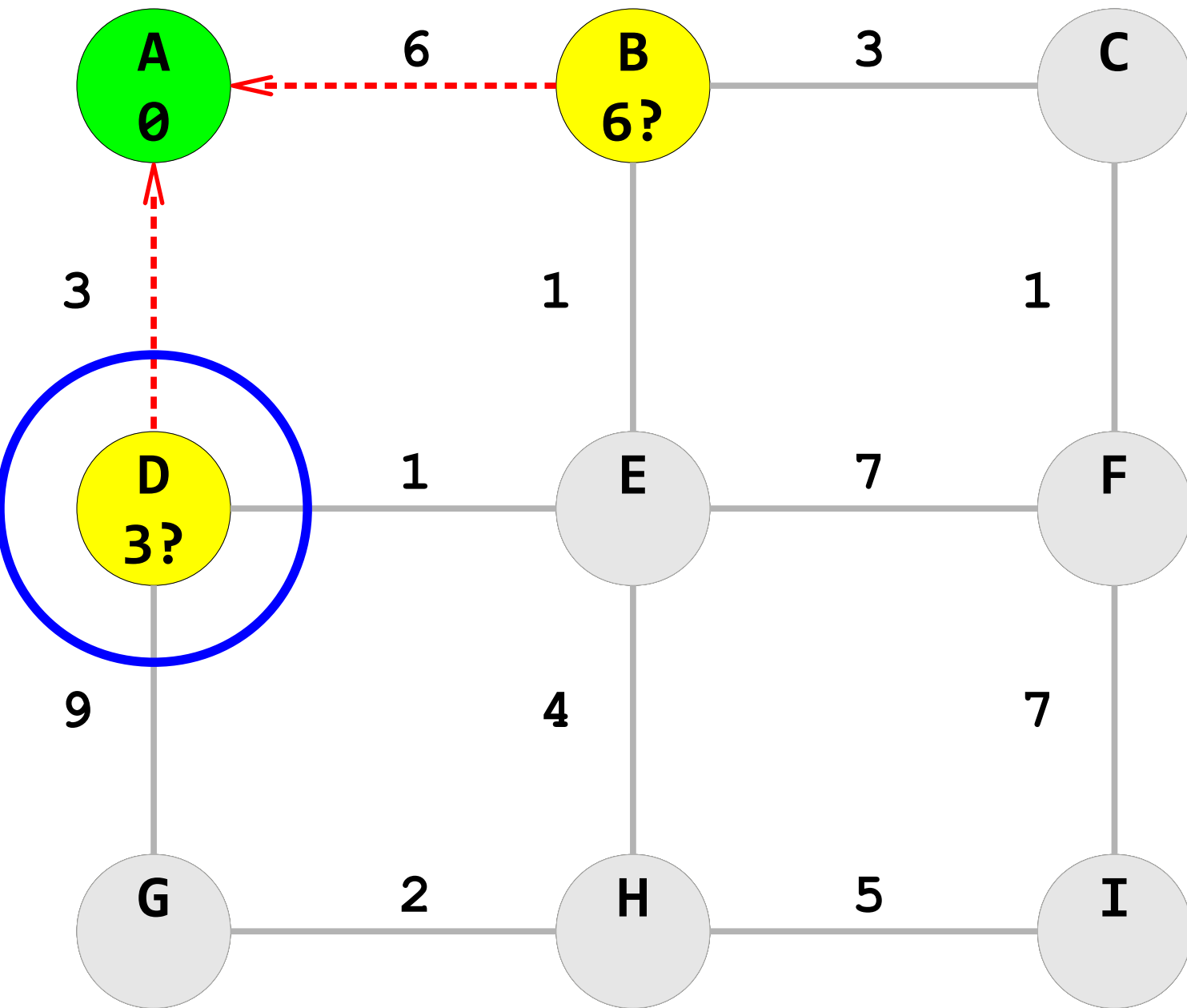


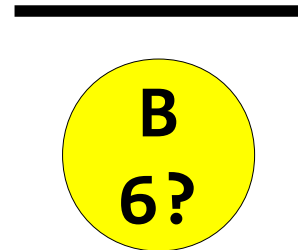
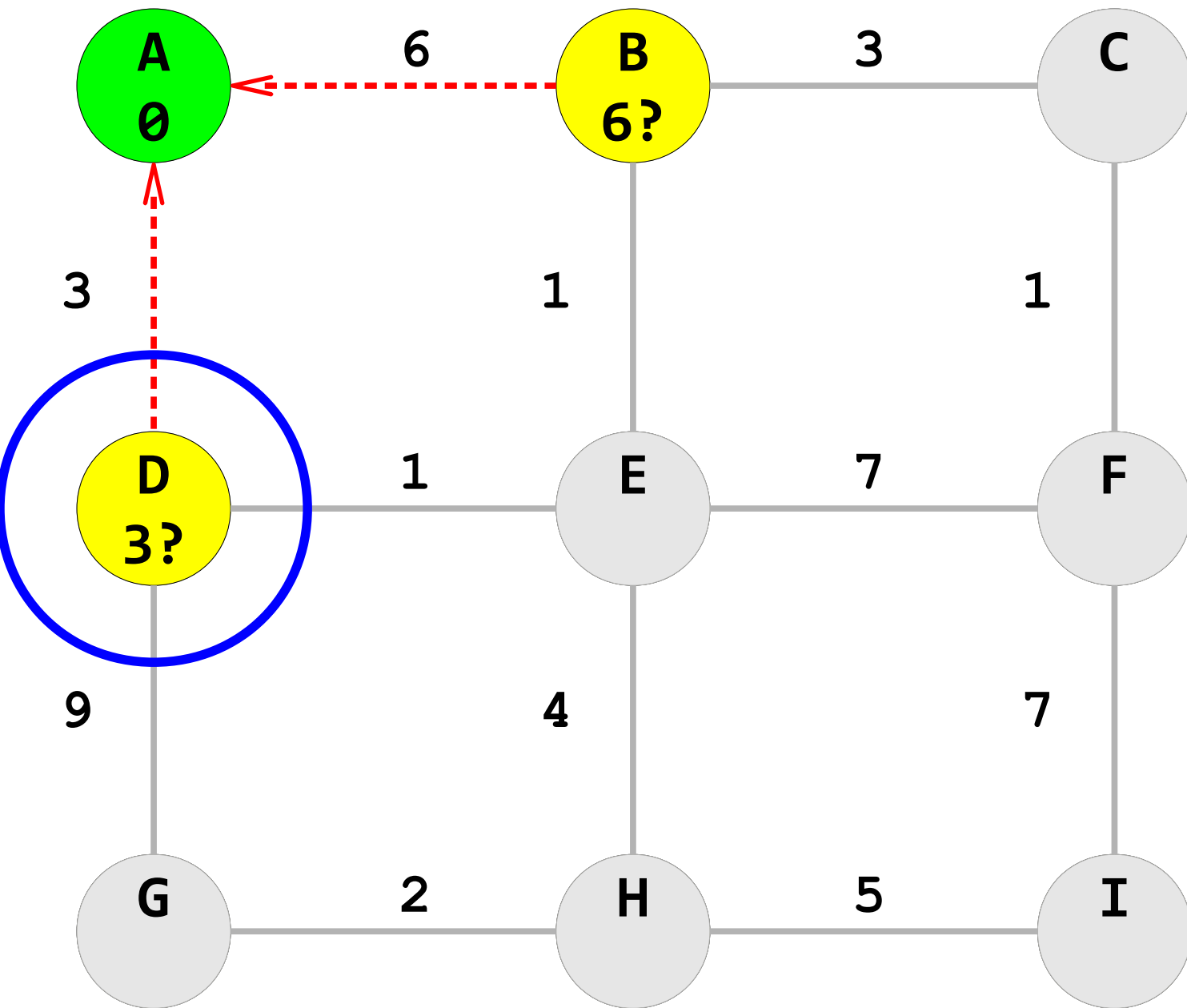


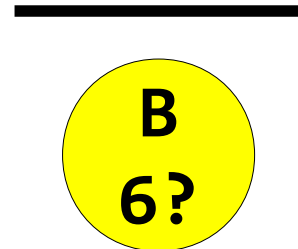
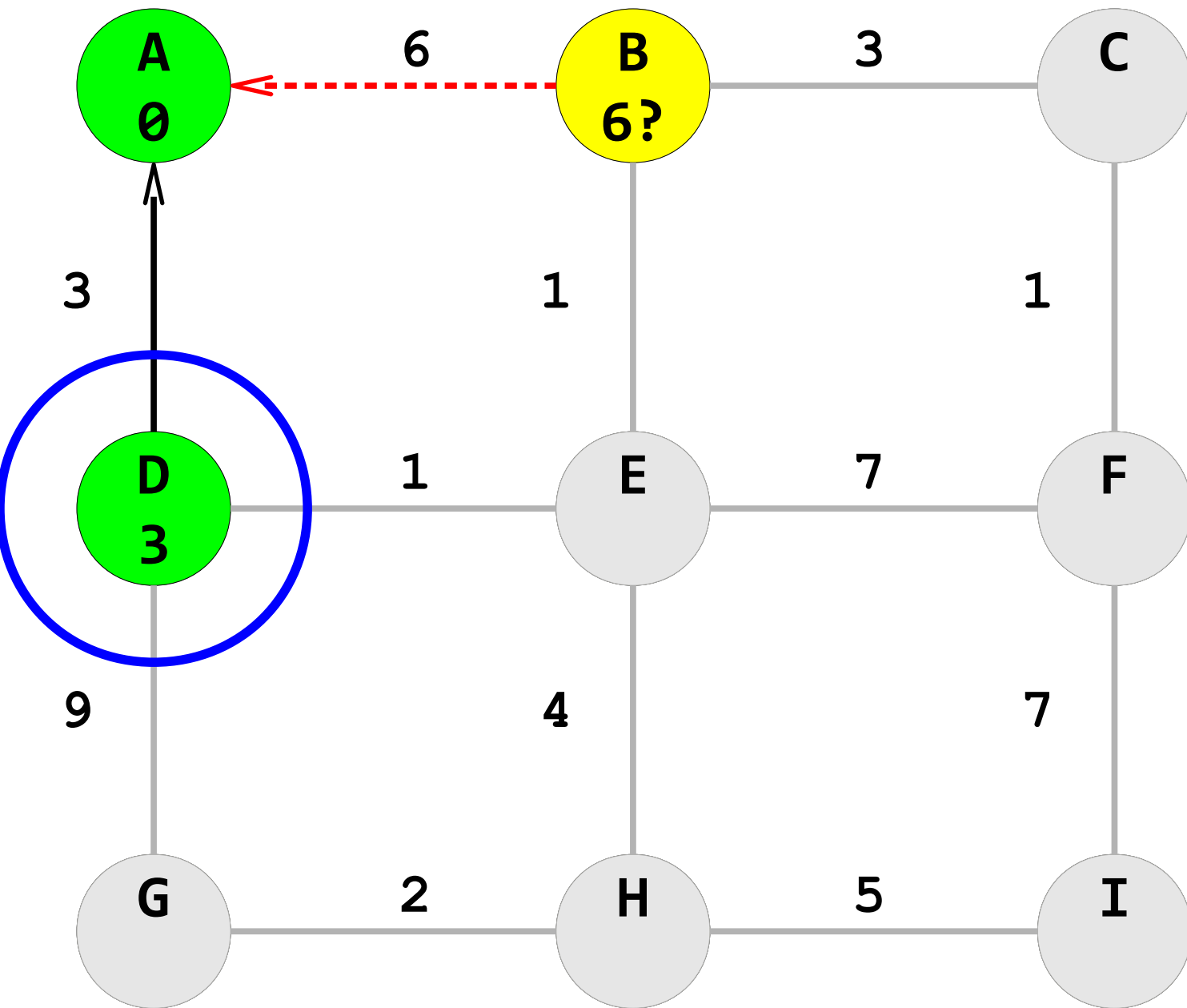
---

D  
3?

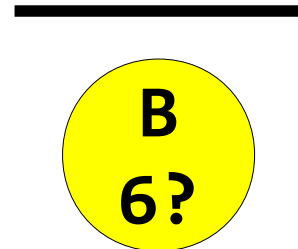
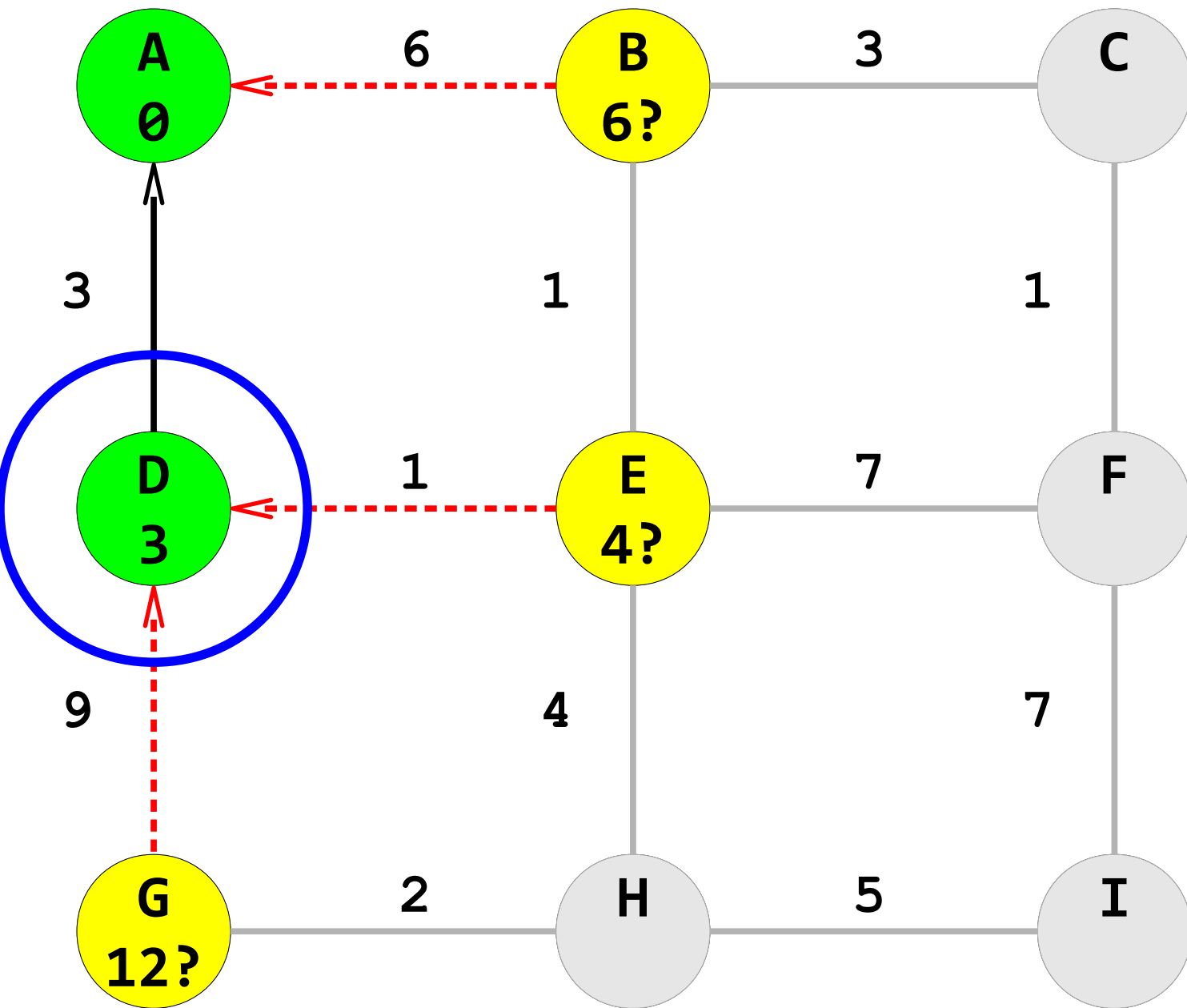
B  
6?

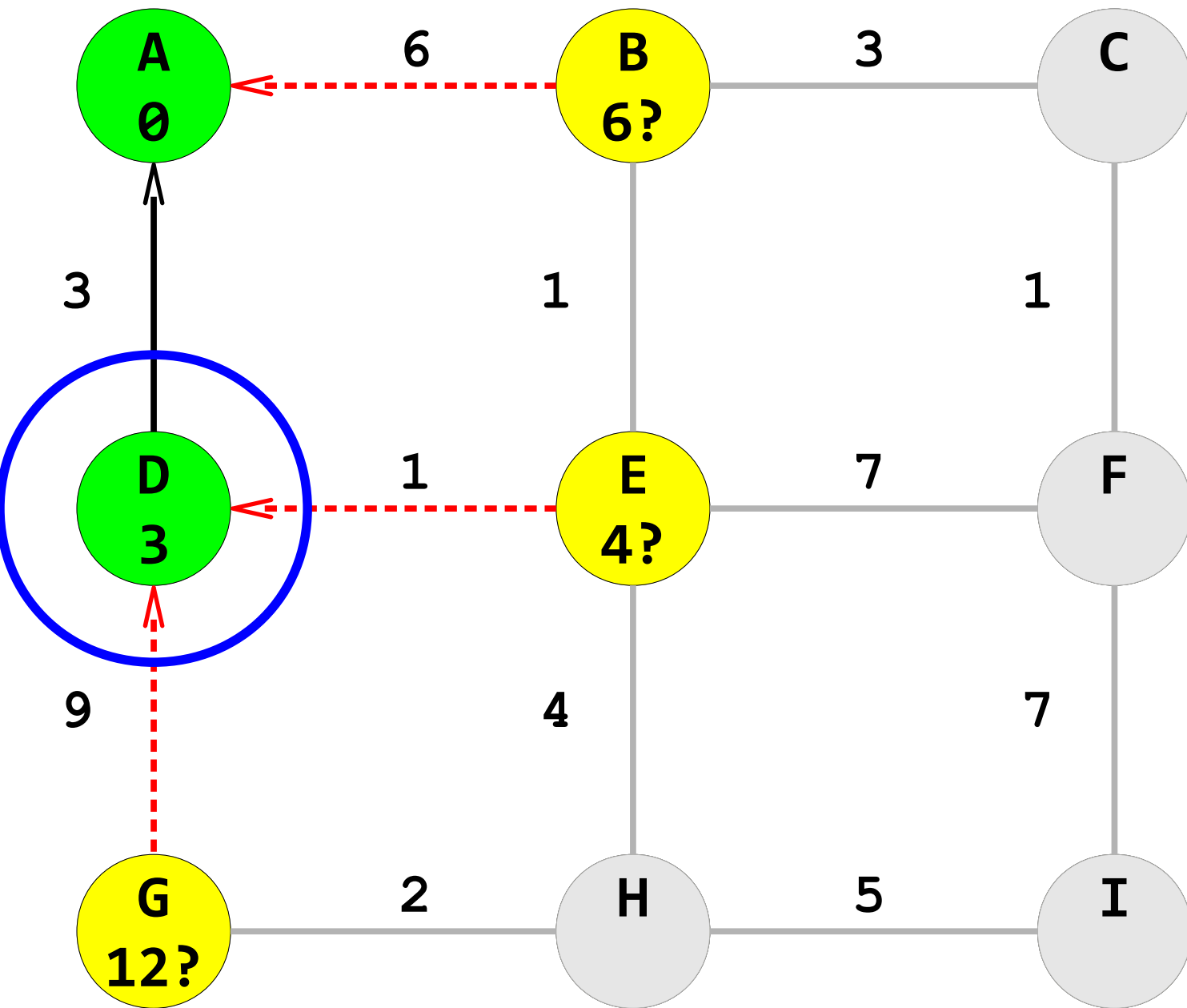




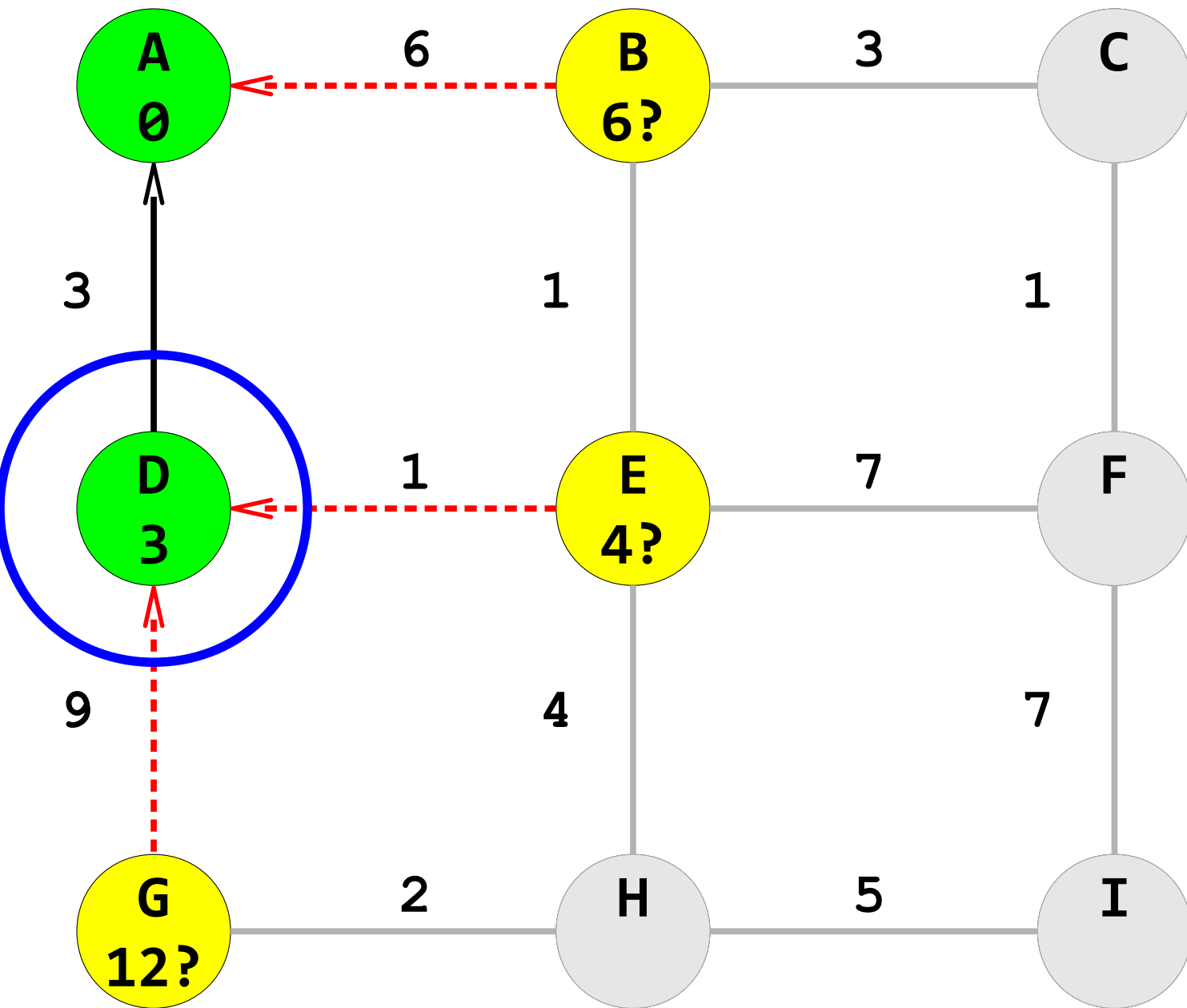








B  
6?

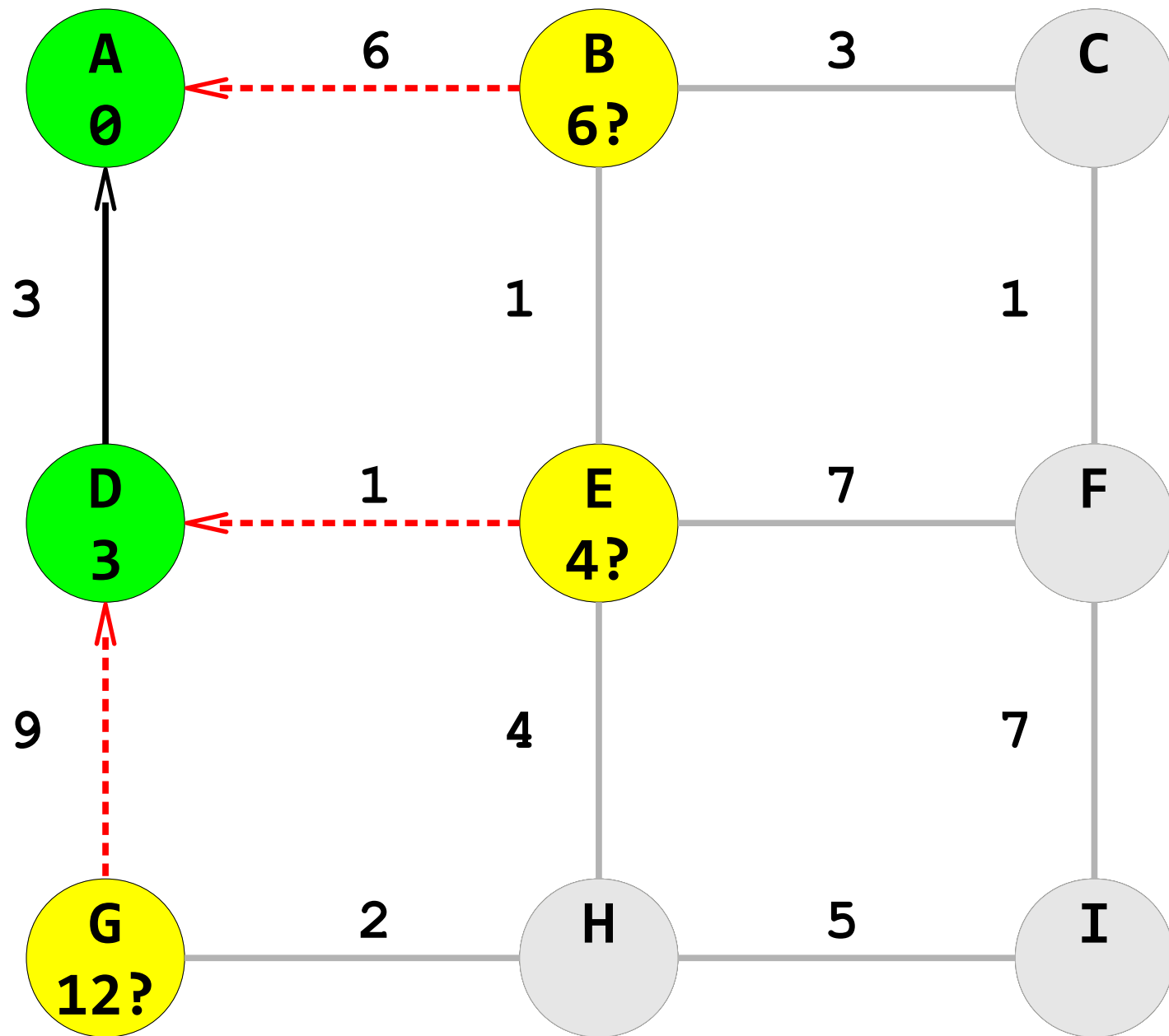


---

E  
4?

B  
6?

G  
12?

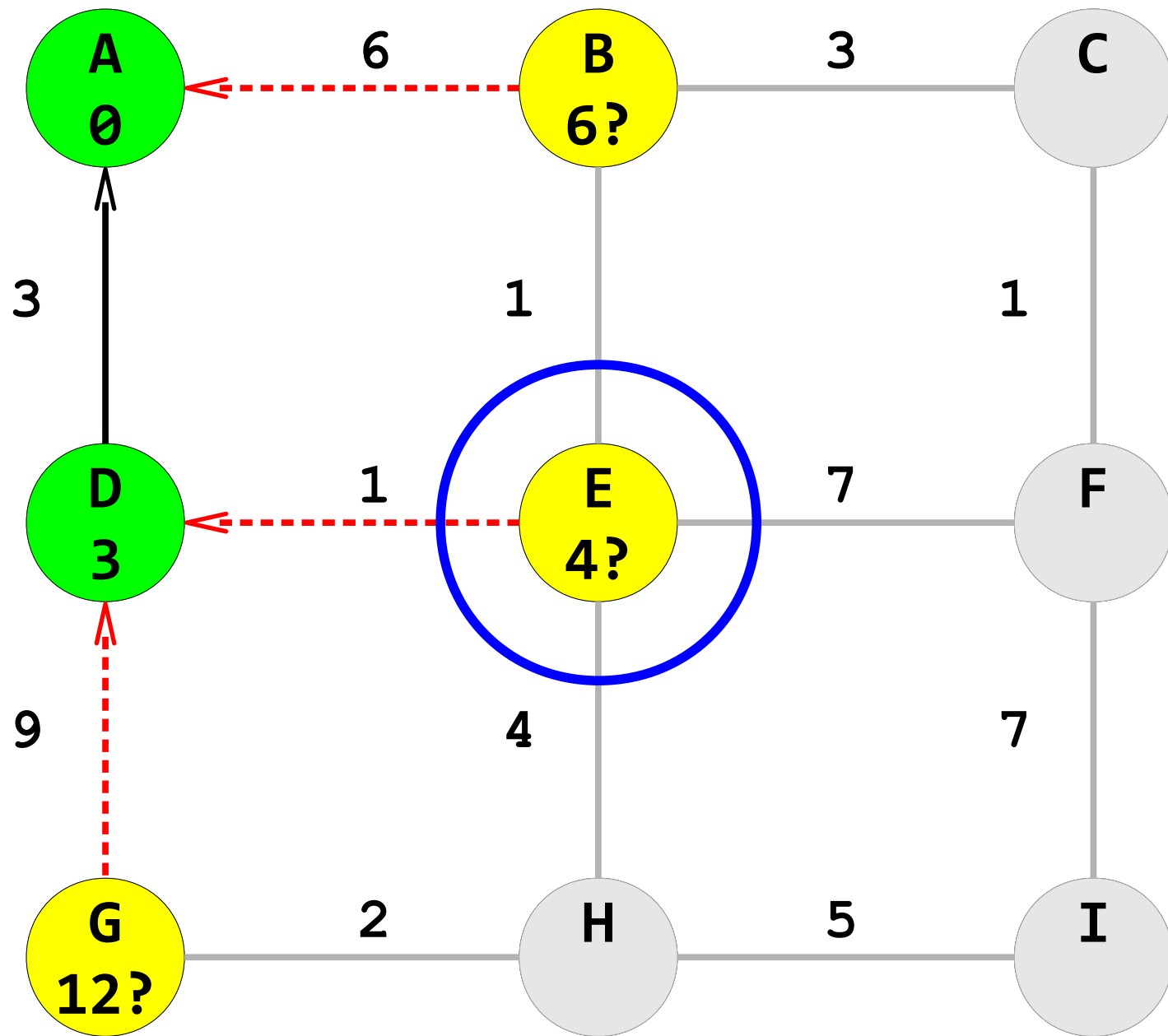


---

E  
4?

B  
6?

G  
12?

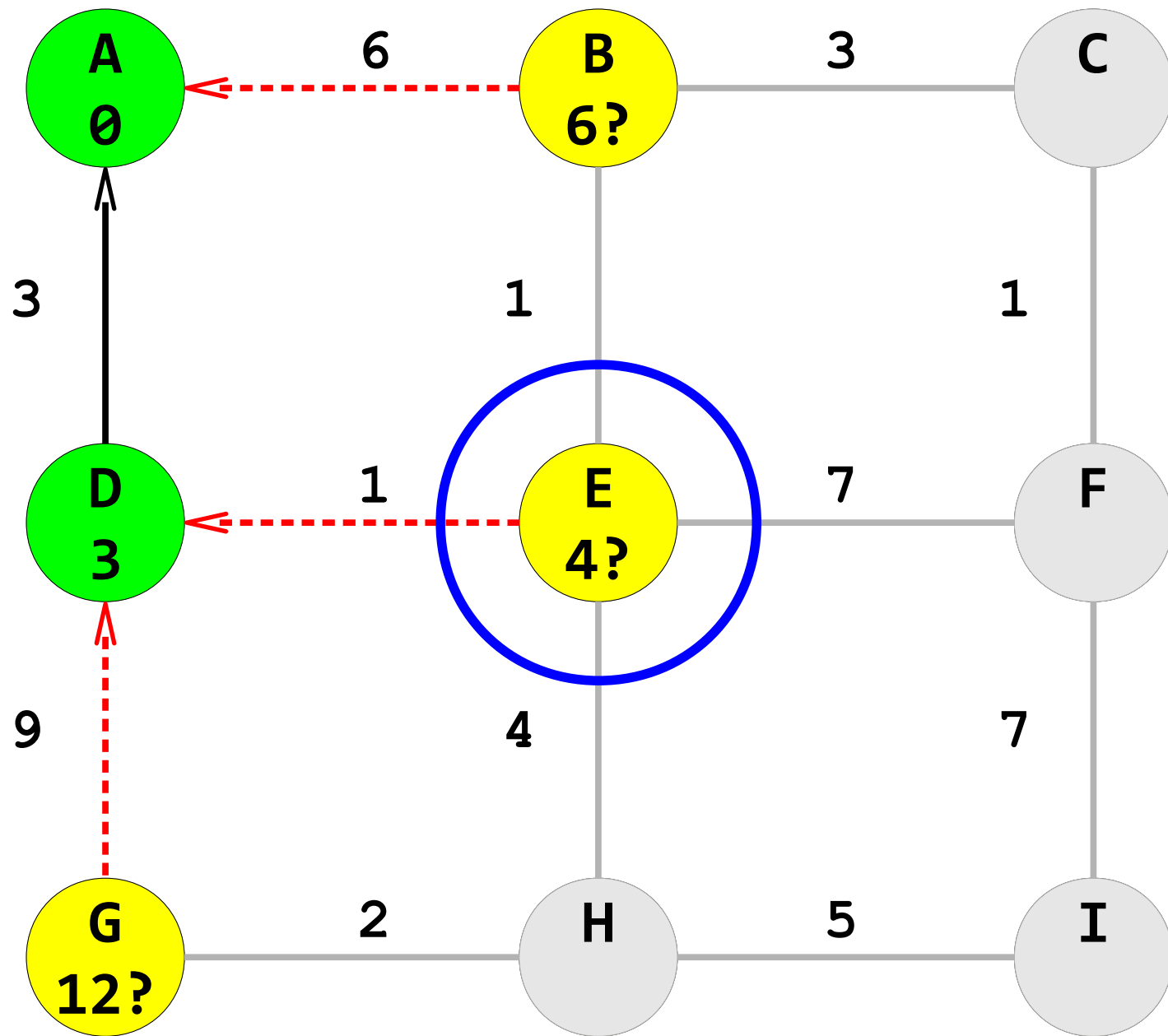


---

E  
4?

B  
6?

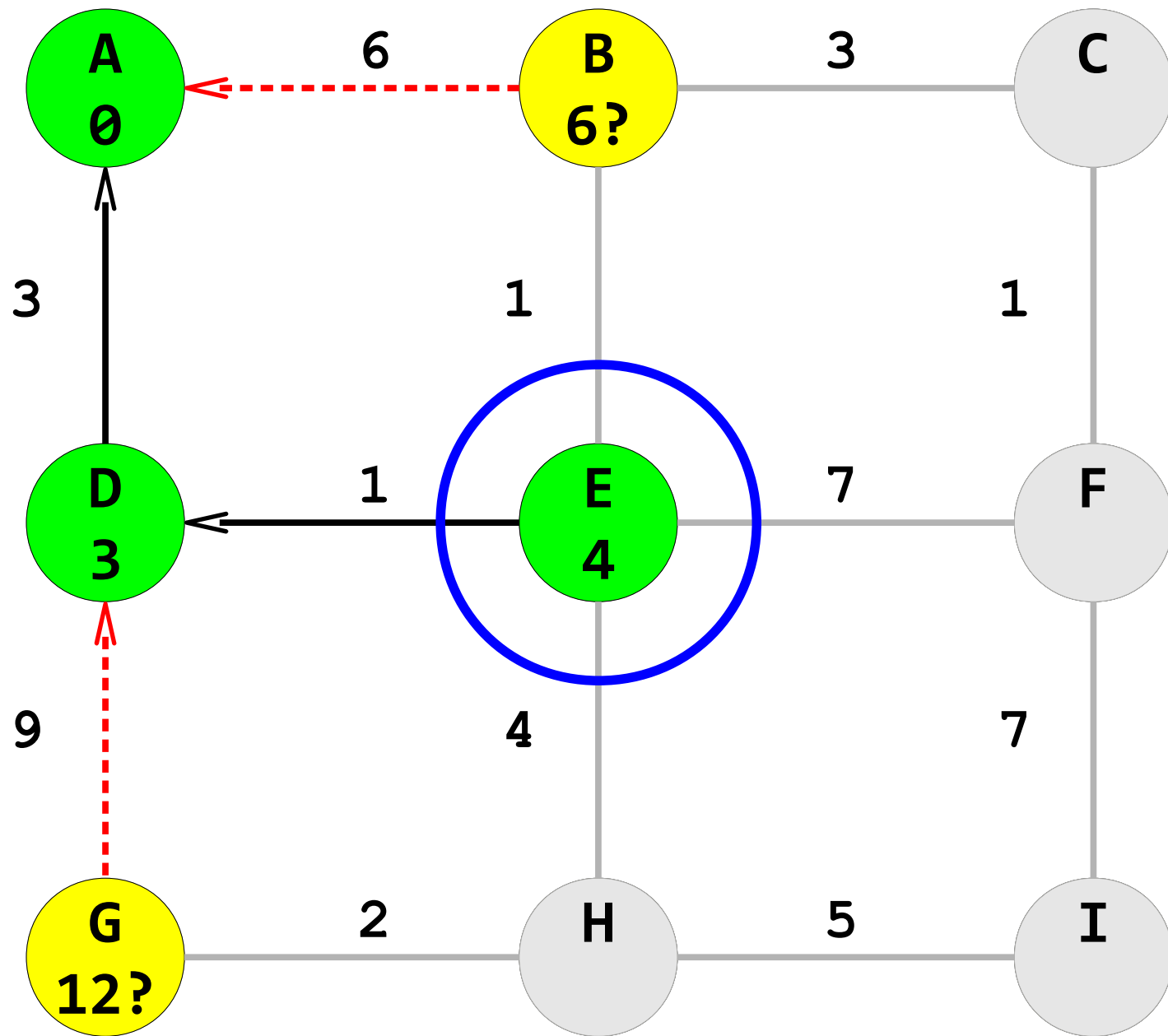
G  
12?



---

**B**  
**6?**

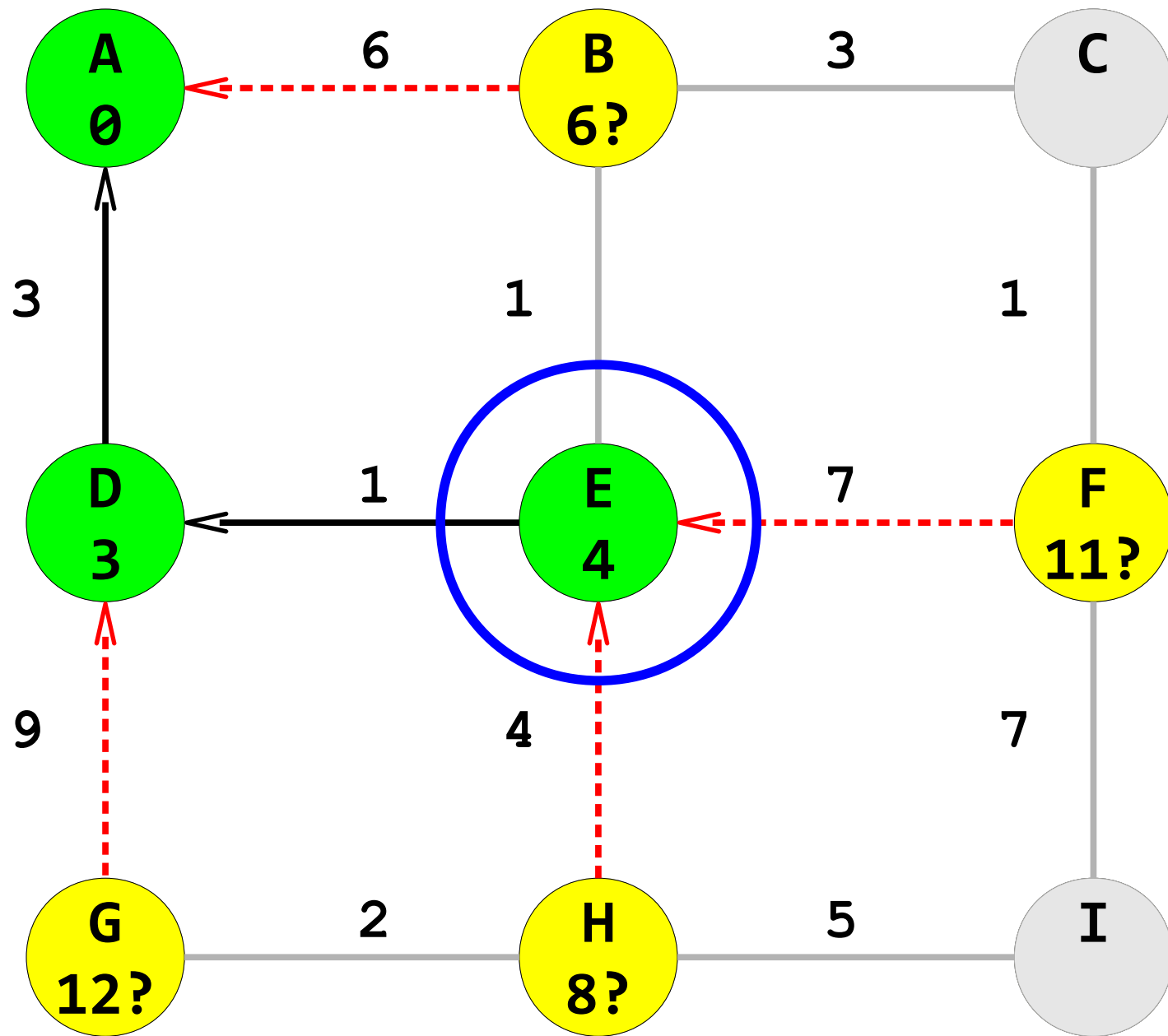
**G**  
**12?**



---

**B**  
**6?**

**G**  
**12?**

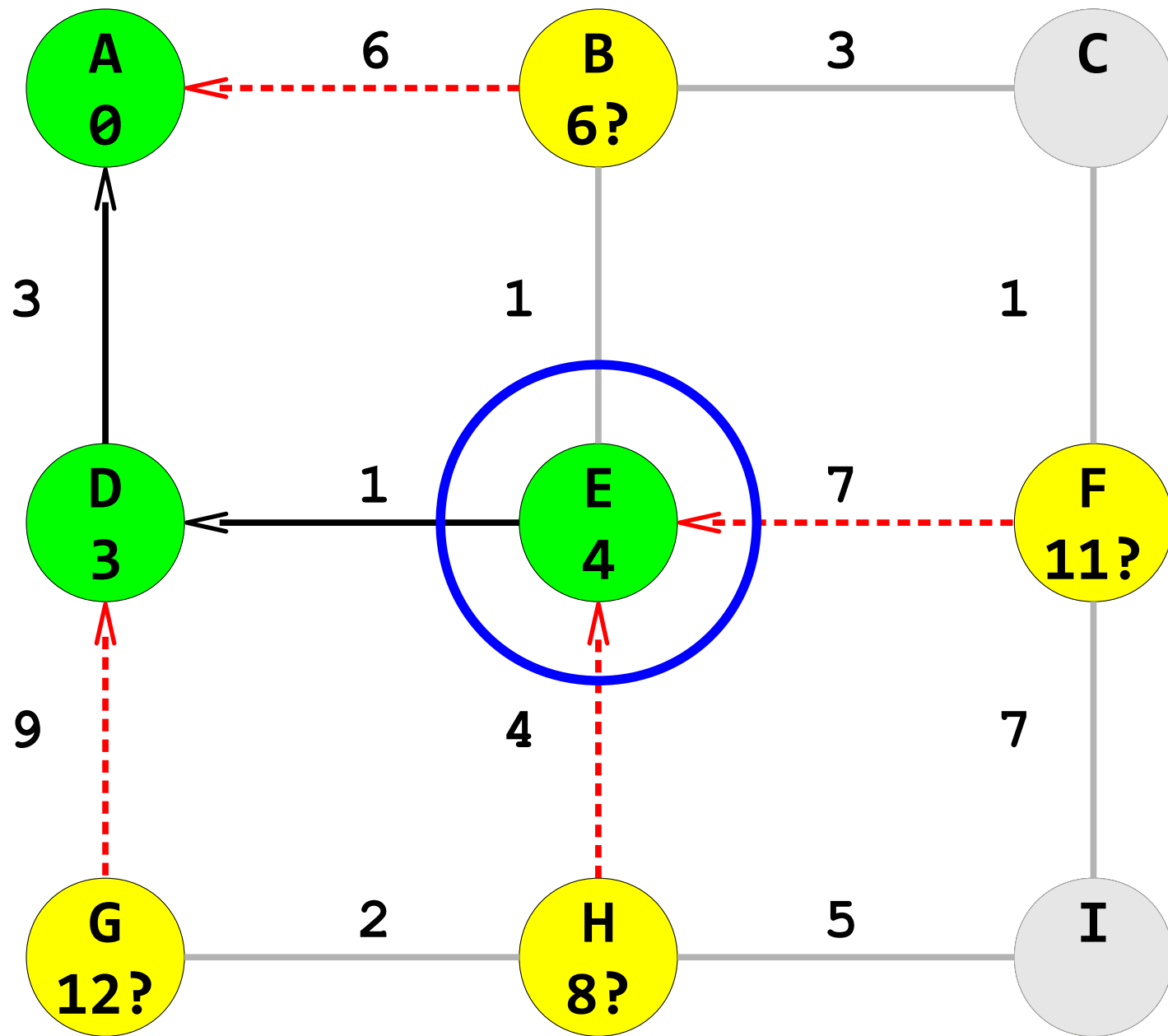


---

B  
6?

G  
12?

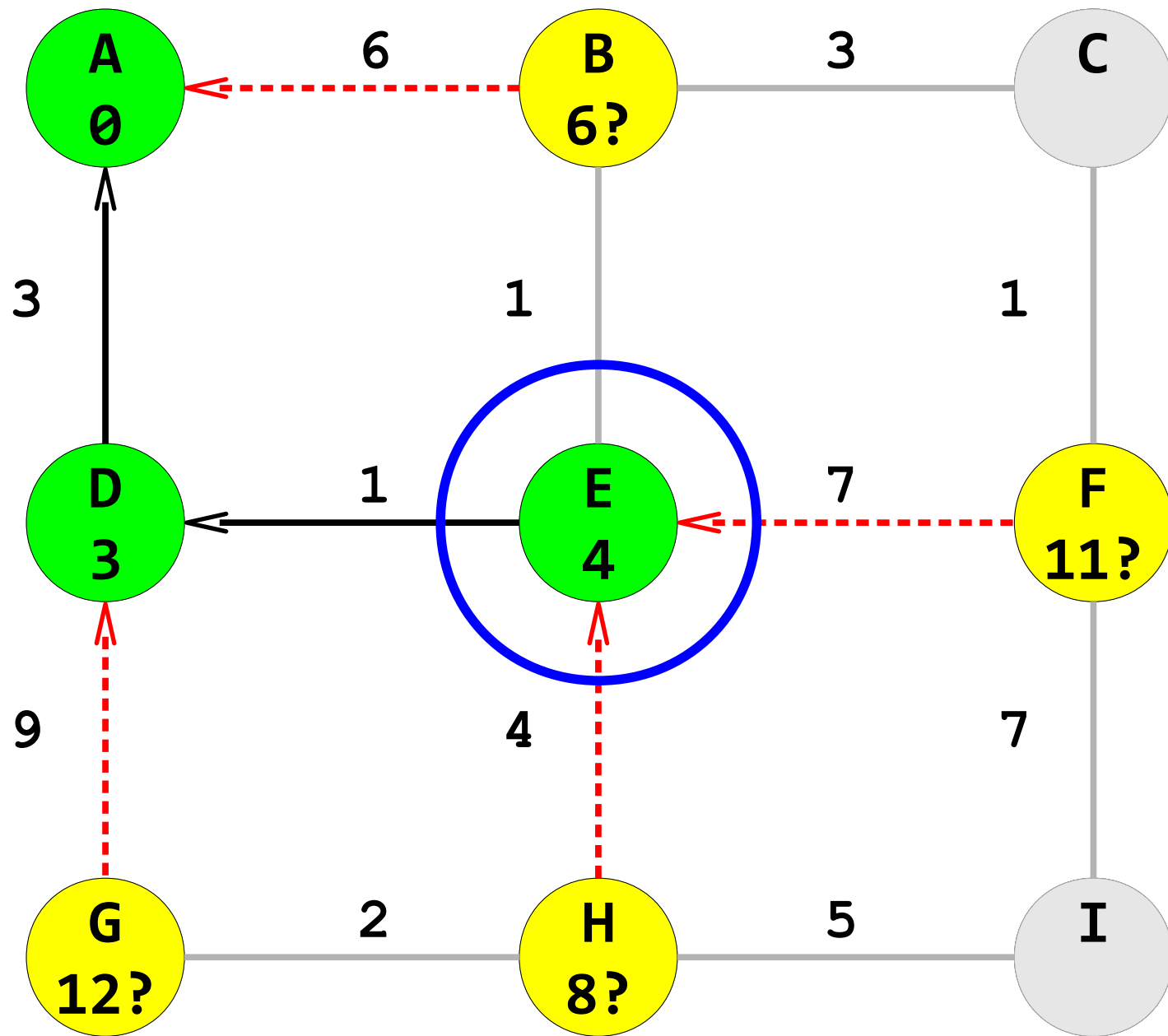




---

**B**  
**6?**

**G**  
**12?**



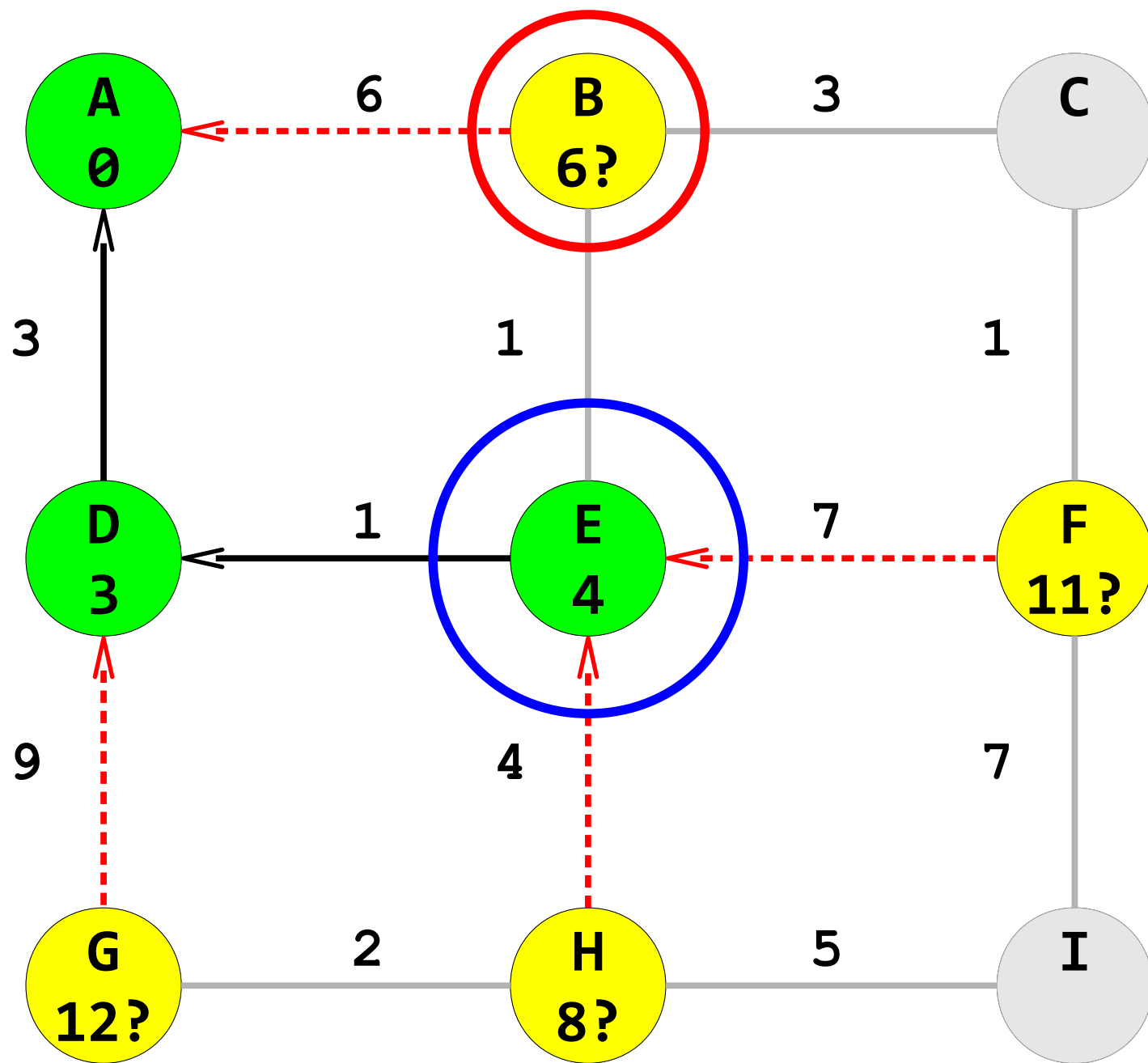
---

B  
6?

H  
8?

F  
11?

G  
12?



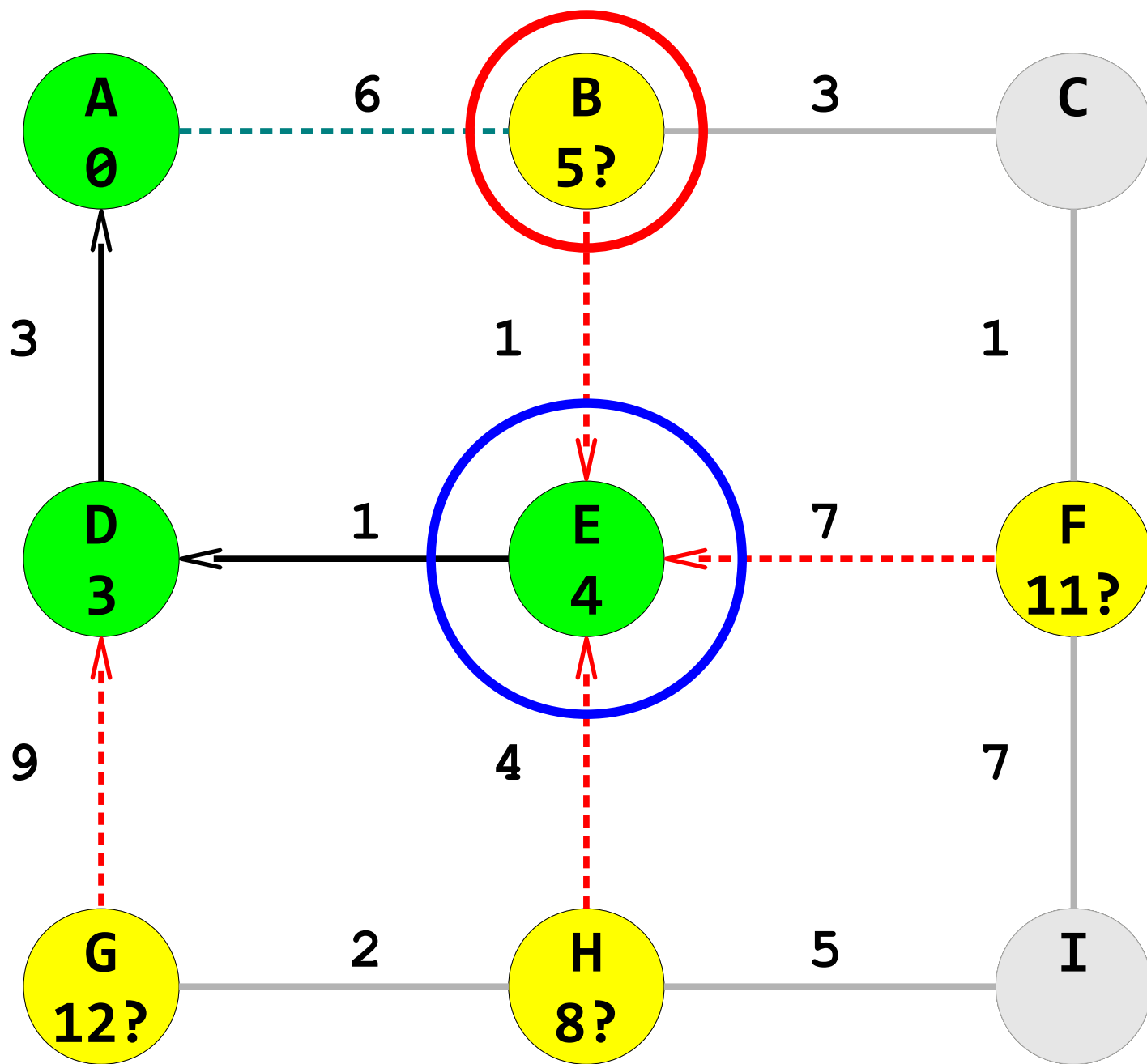
---

B  
6?

H  
8?

F  
11?

G  
12?

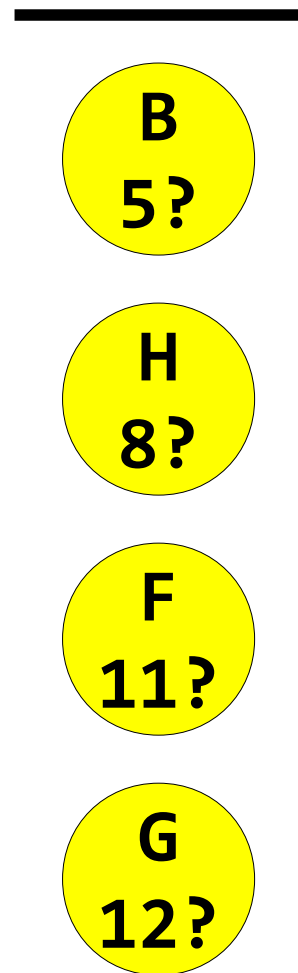
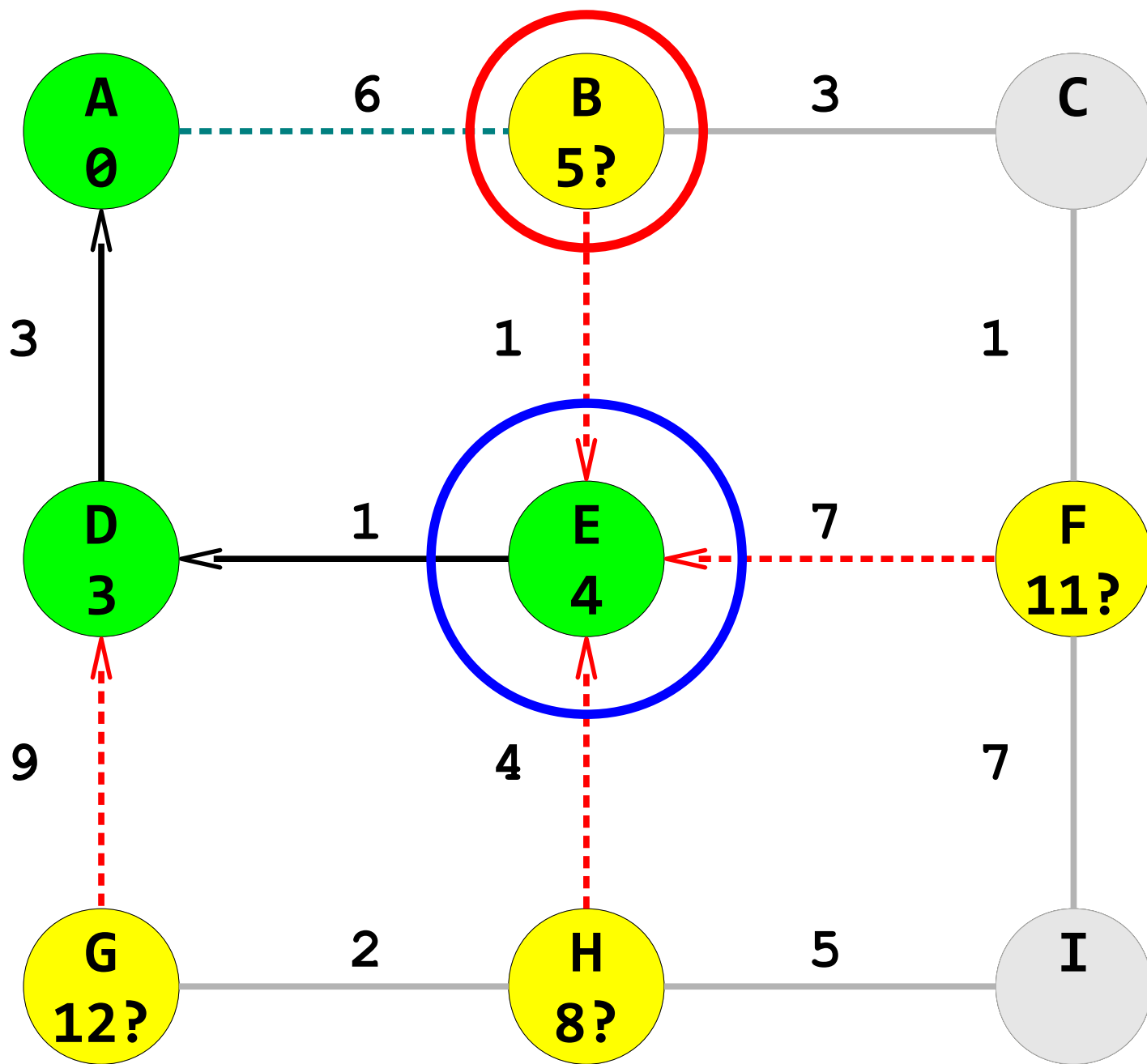


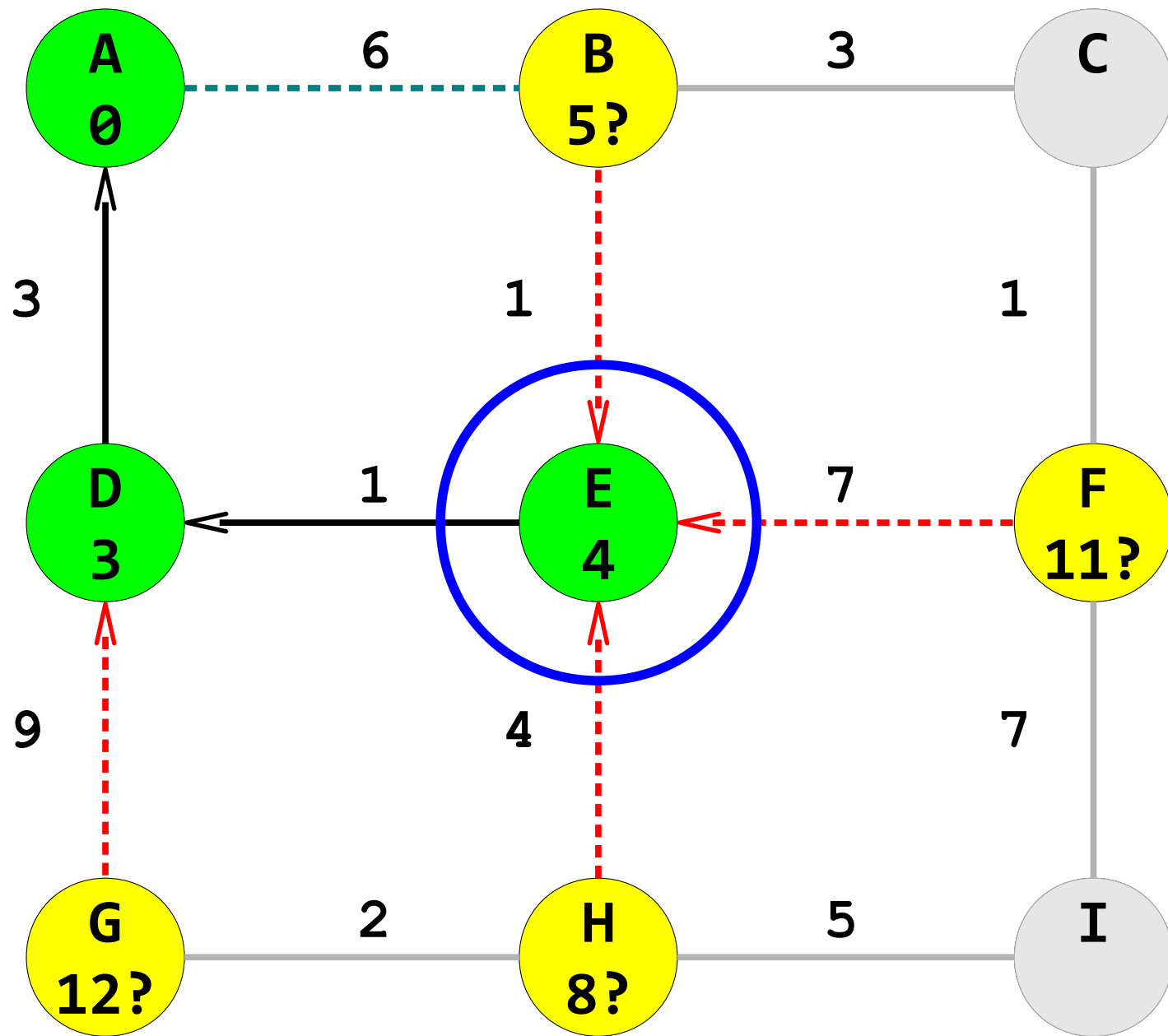
B  
6?

H  
8?

F  
11?

G  
12?





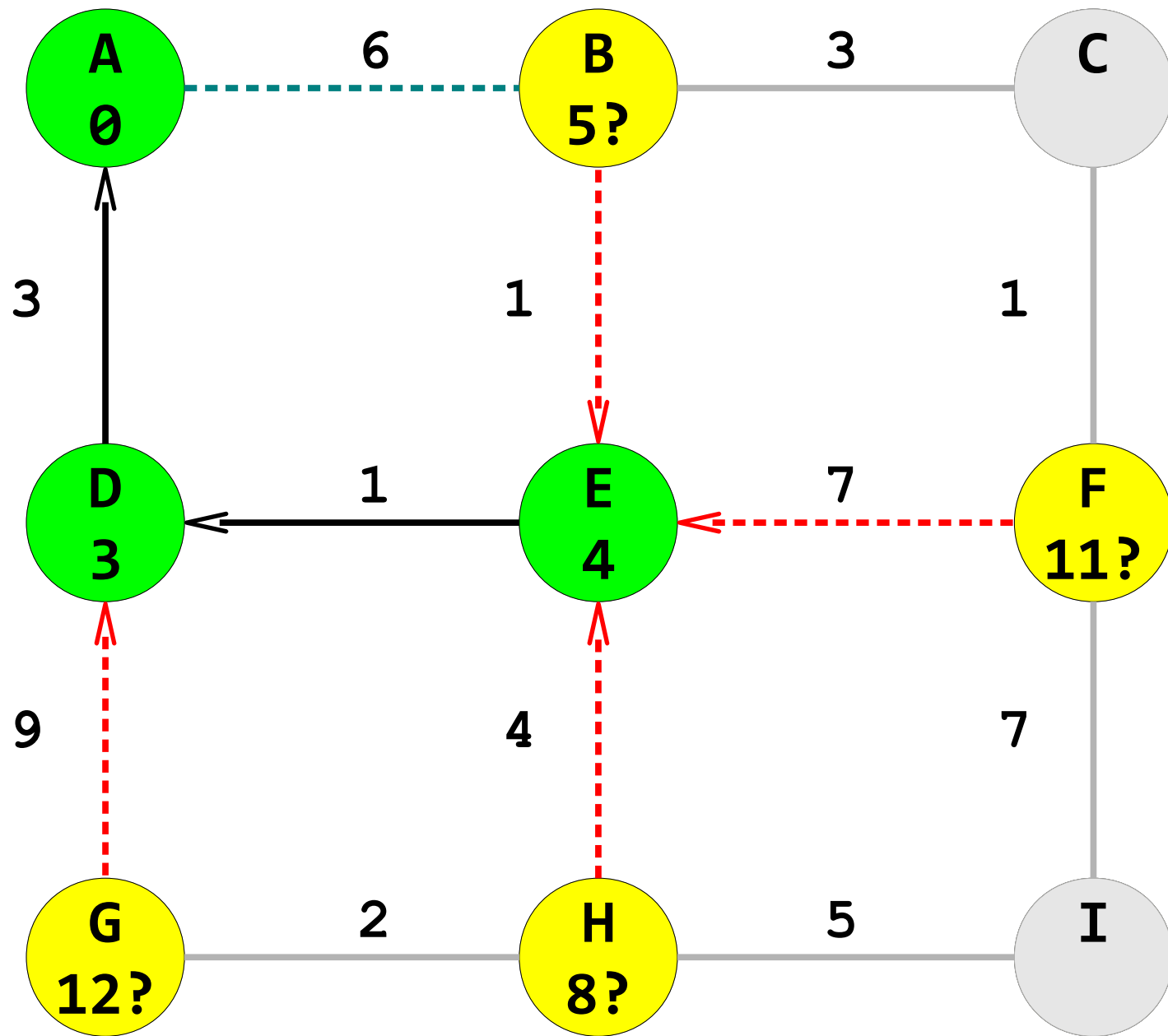
---

B  
5?

H  
8?

F  
11?

G  
12?



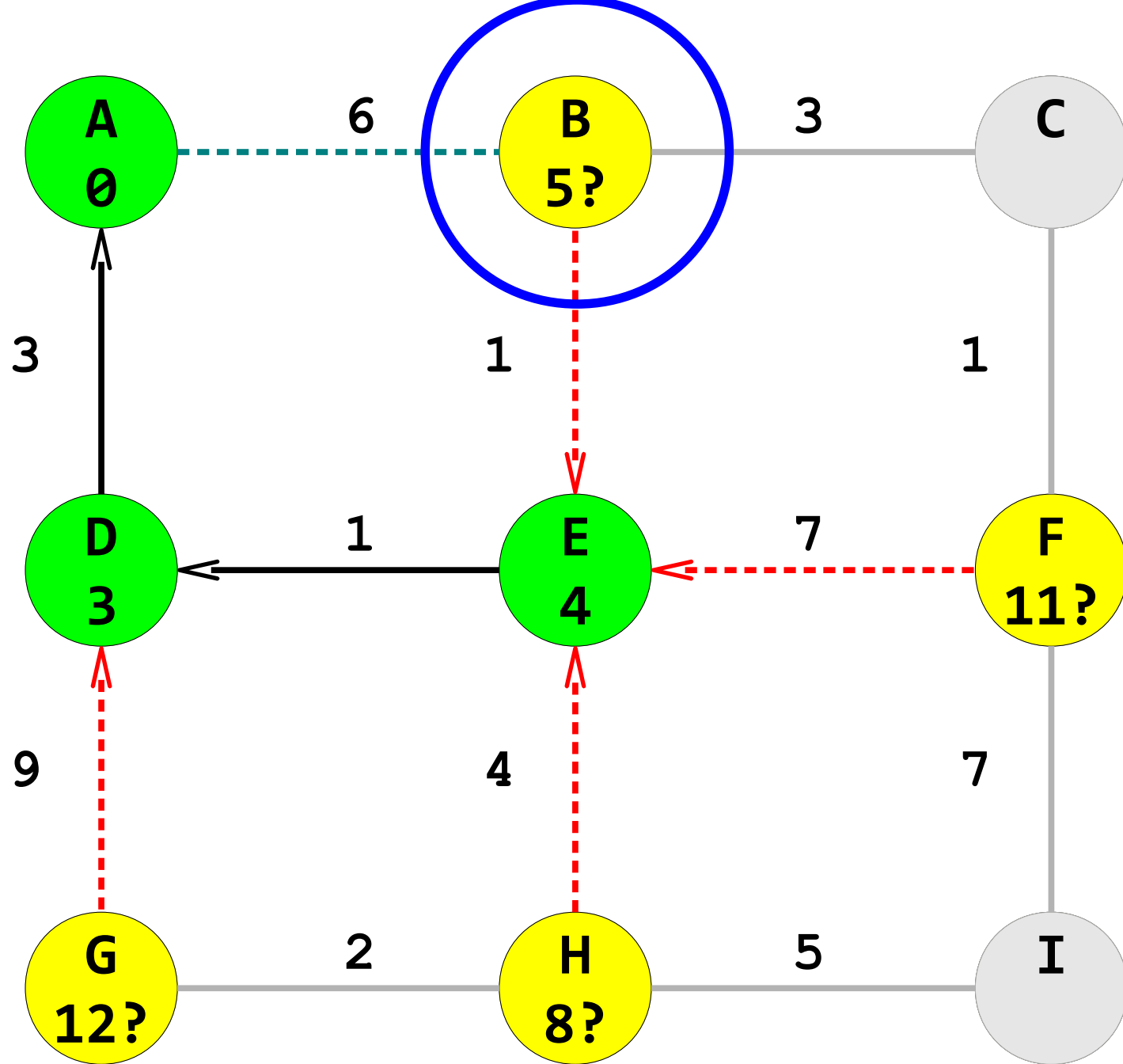
---

B  
5?

H  
8?

F  
11?

G  
12?



---

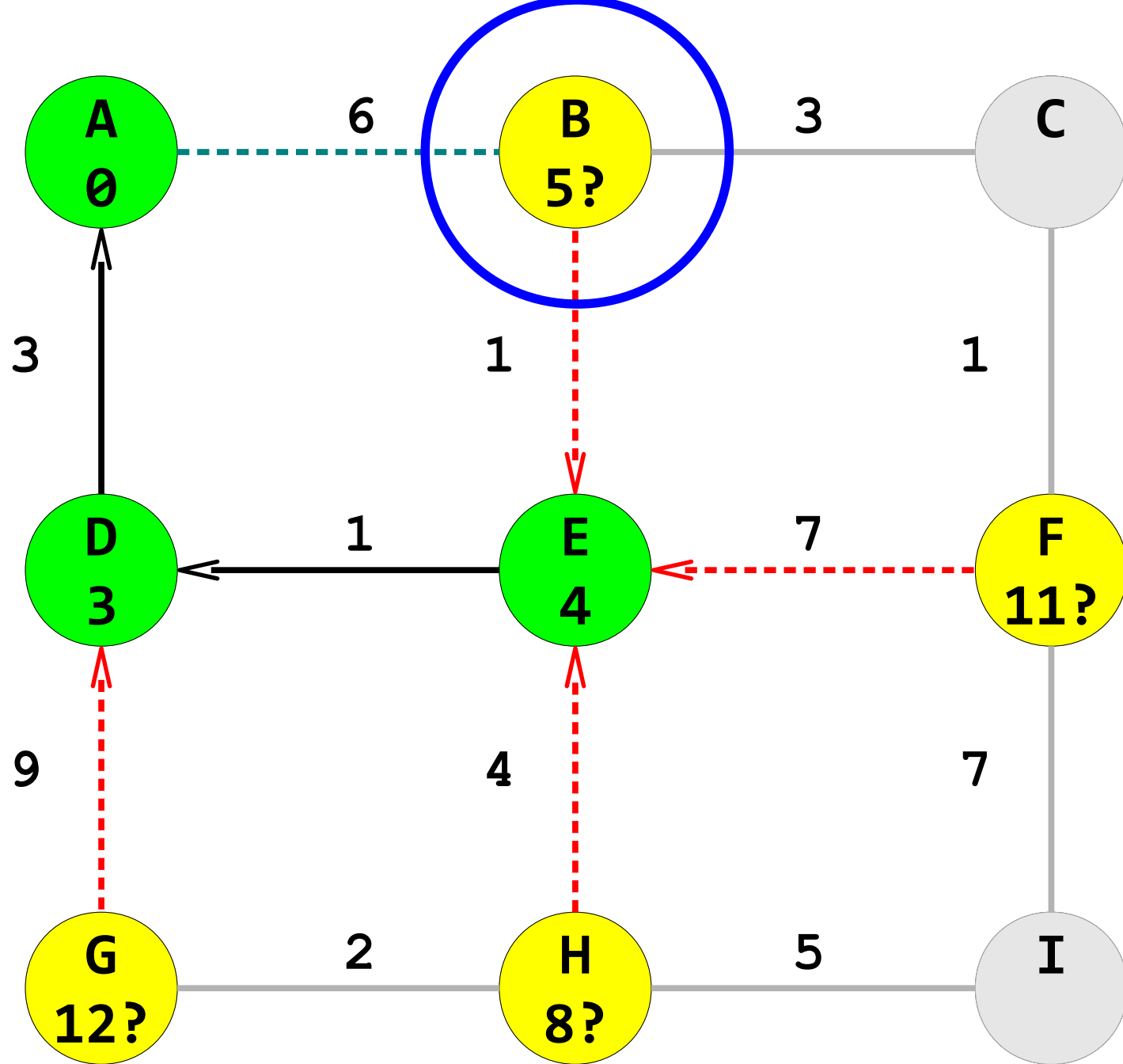
B  
5?

H  
8?

F  
11?

G  
12?



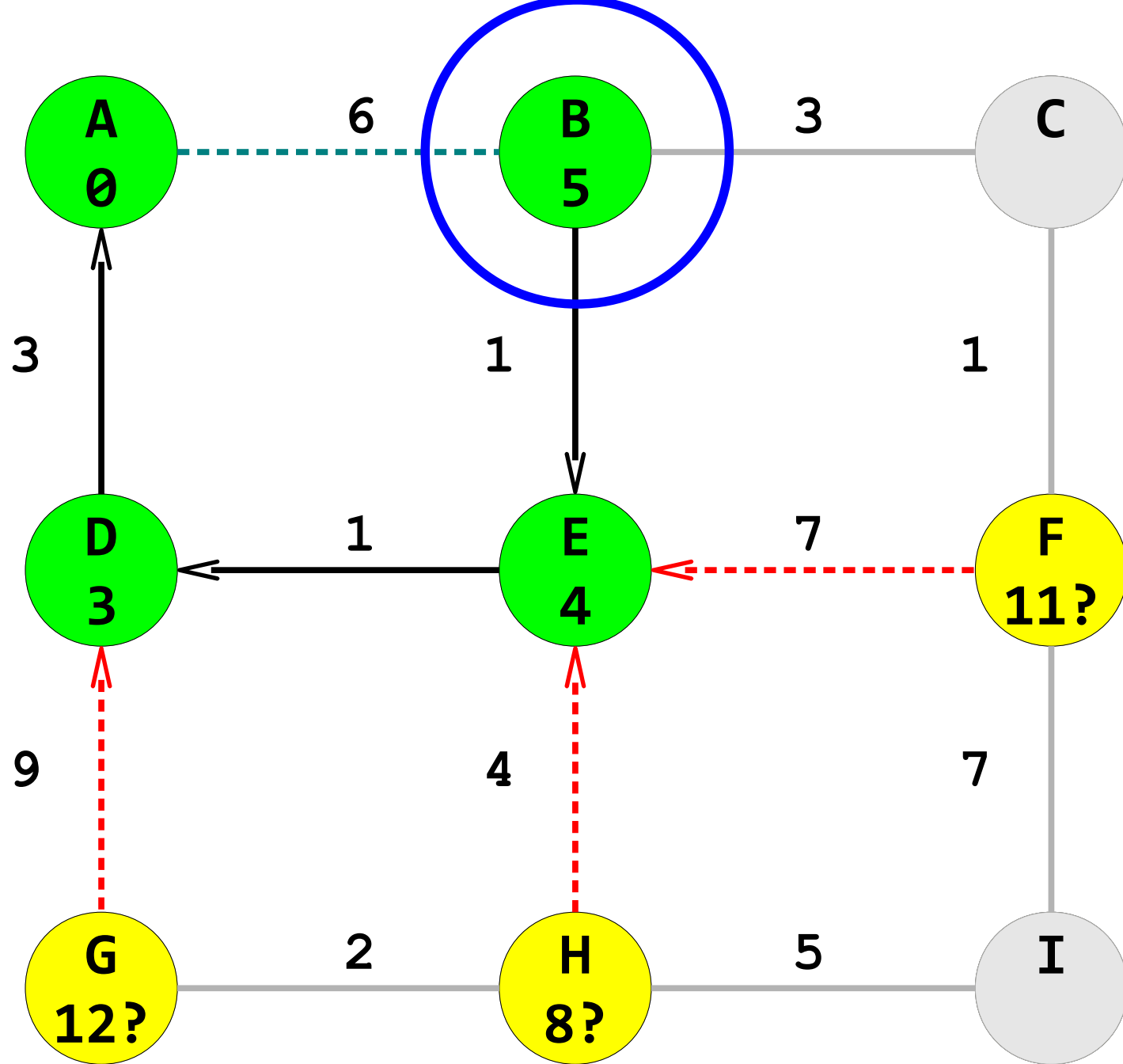


---

H  
8?

F  
11?

G  
12?

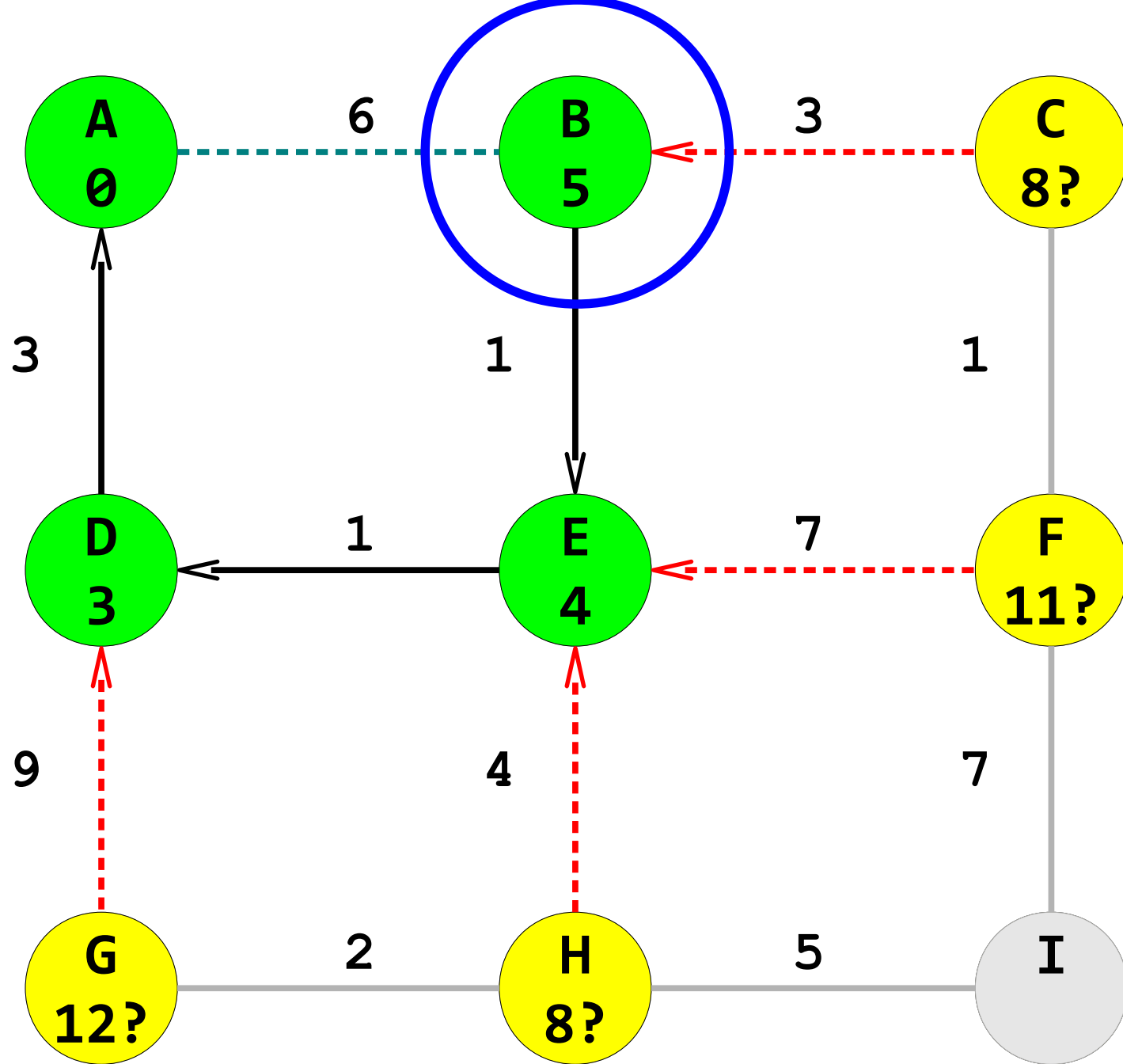


---

H  
8?

F  
11?

G  
12?

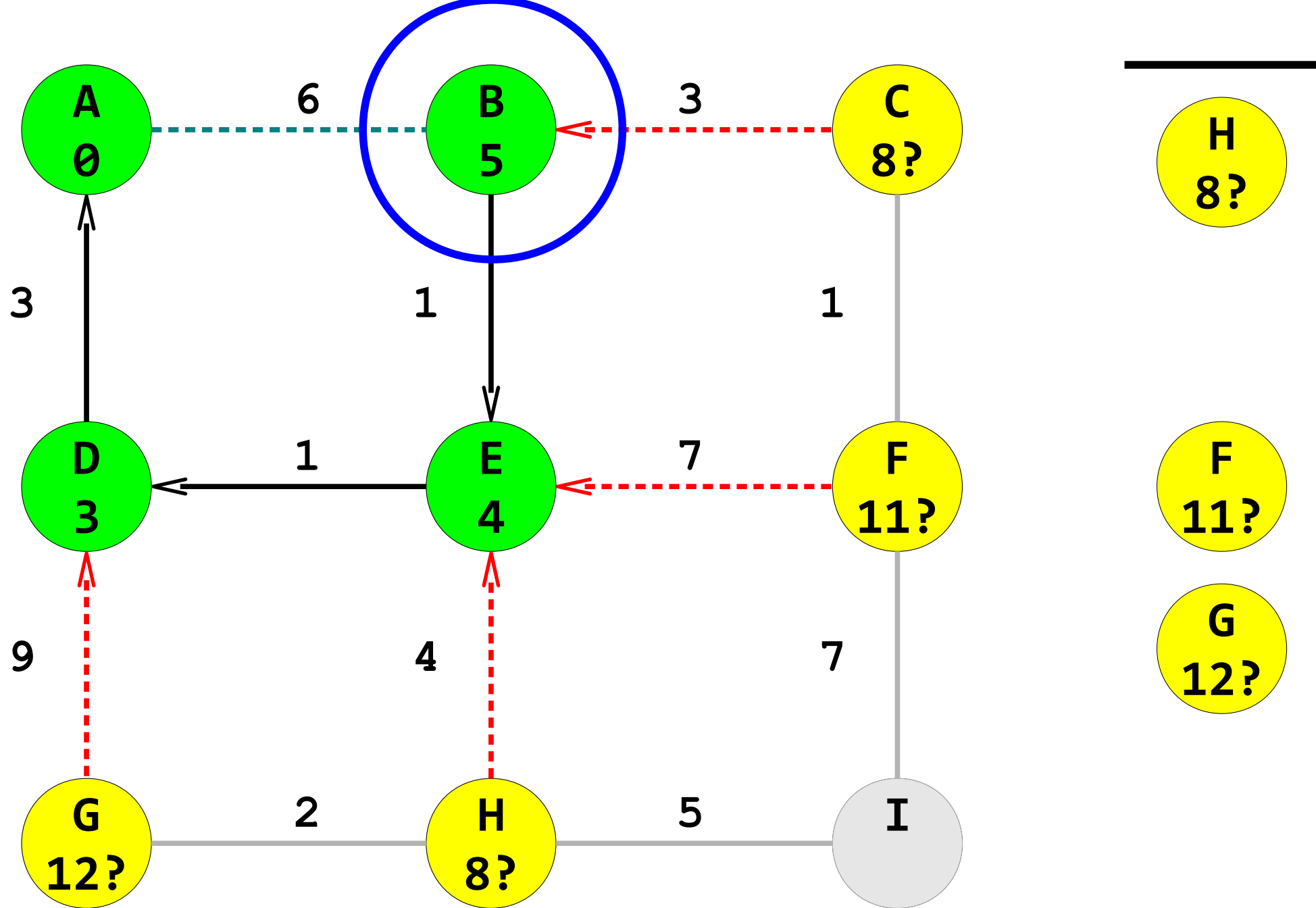


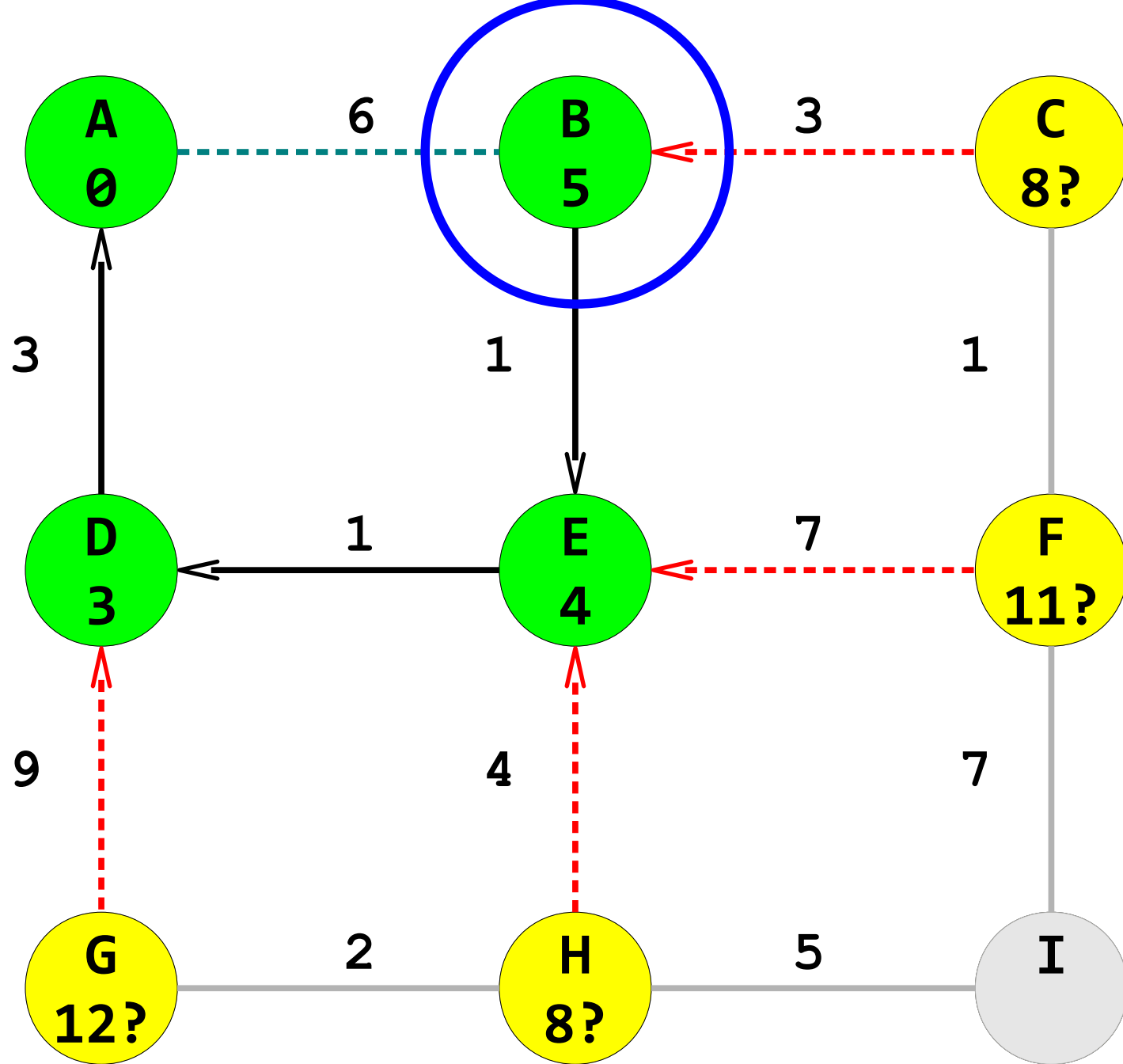
---

H  
8?

F  
11?

G  
12?





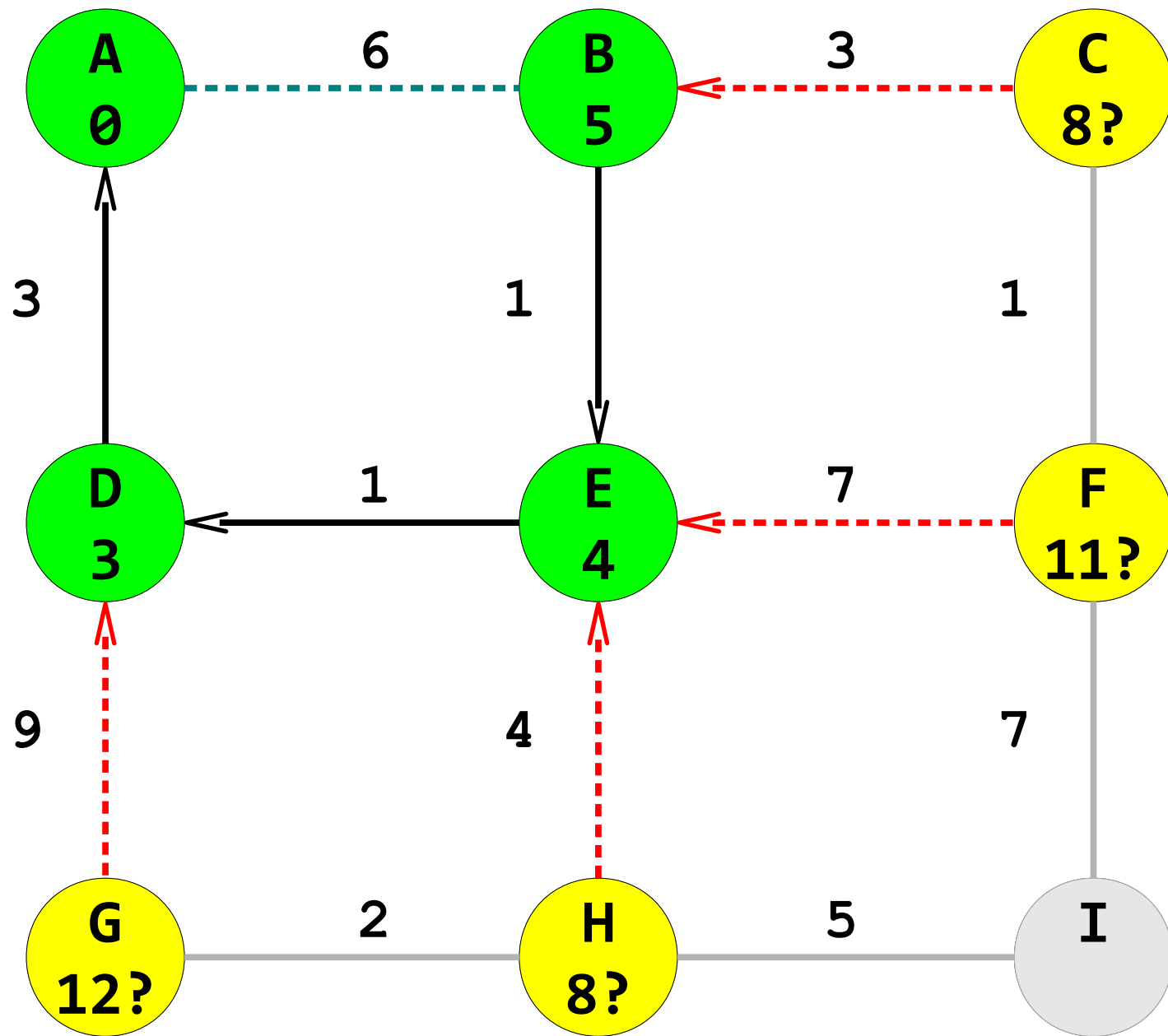
---

H  
8?

C  
8?

F  
11?

G  
12?



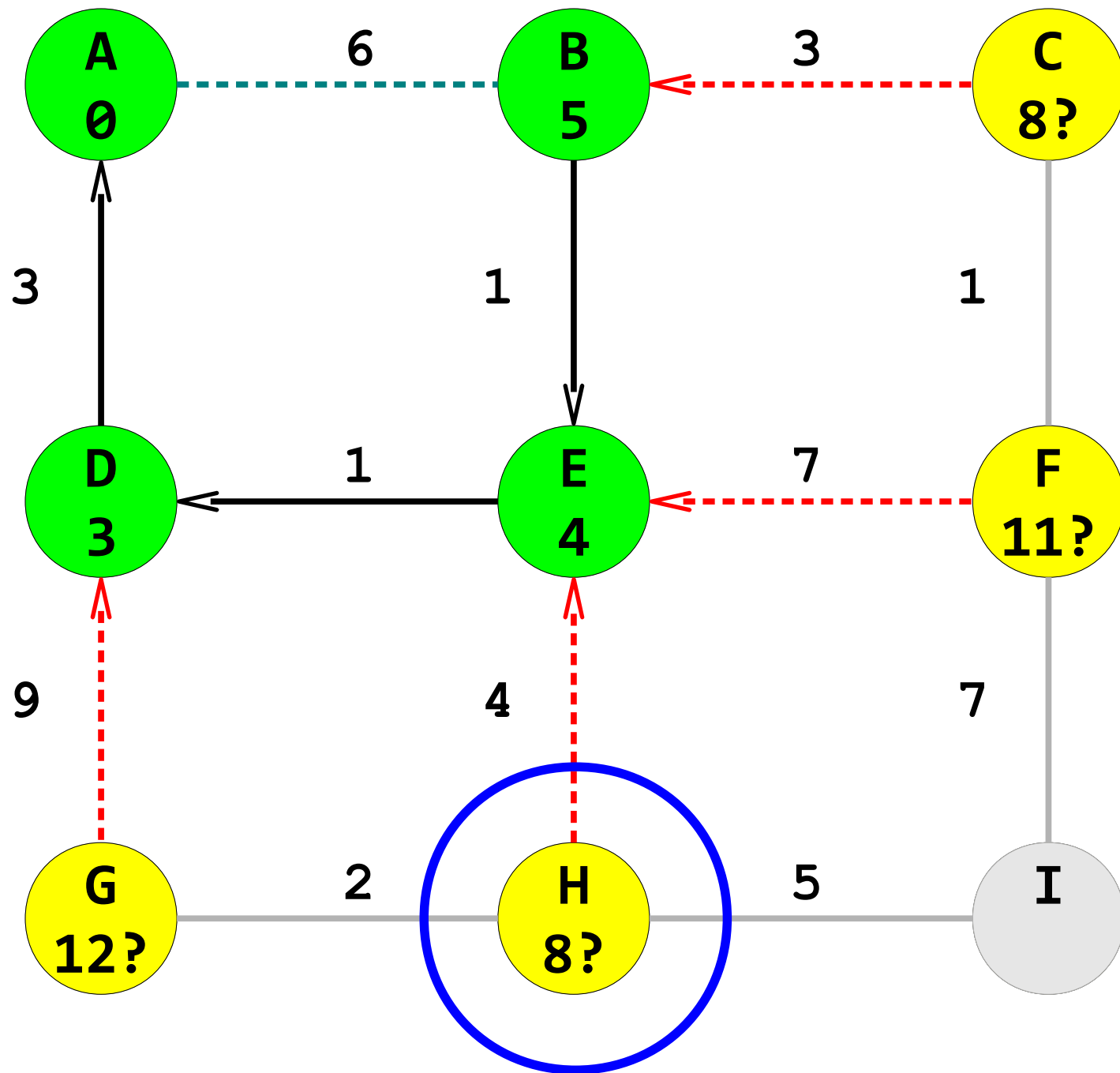
---

H  
8?

C  
8?

F  
11?

G  
12?



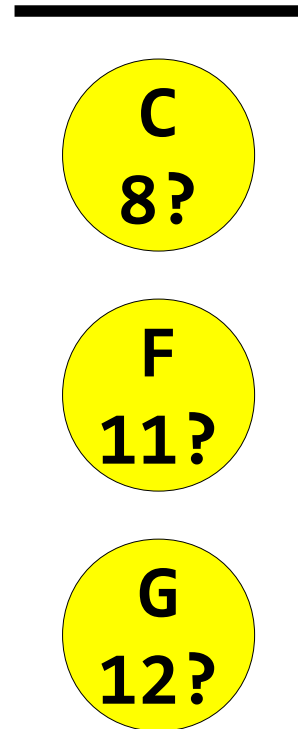
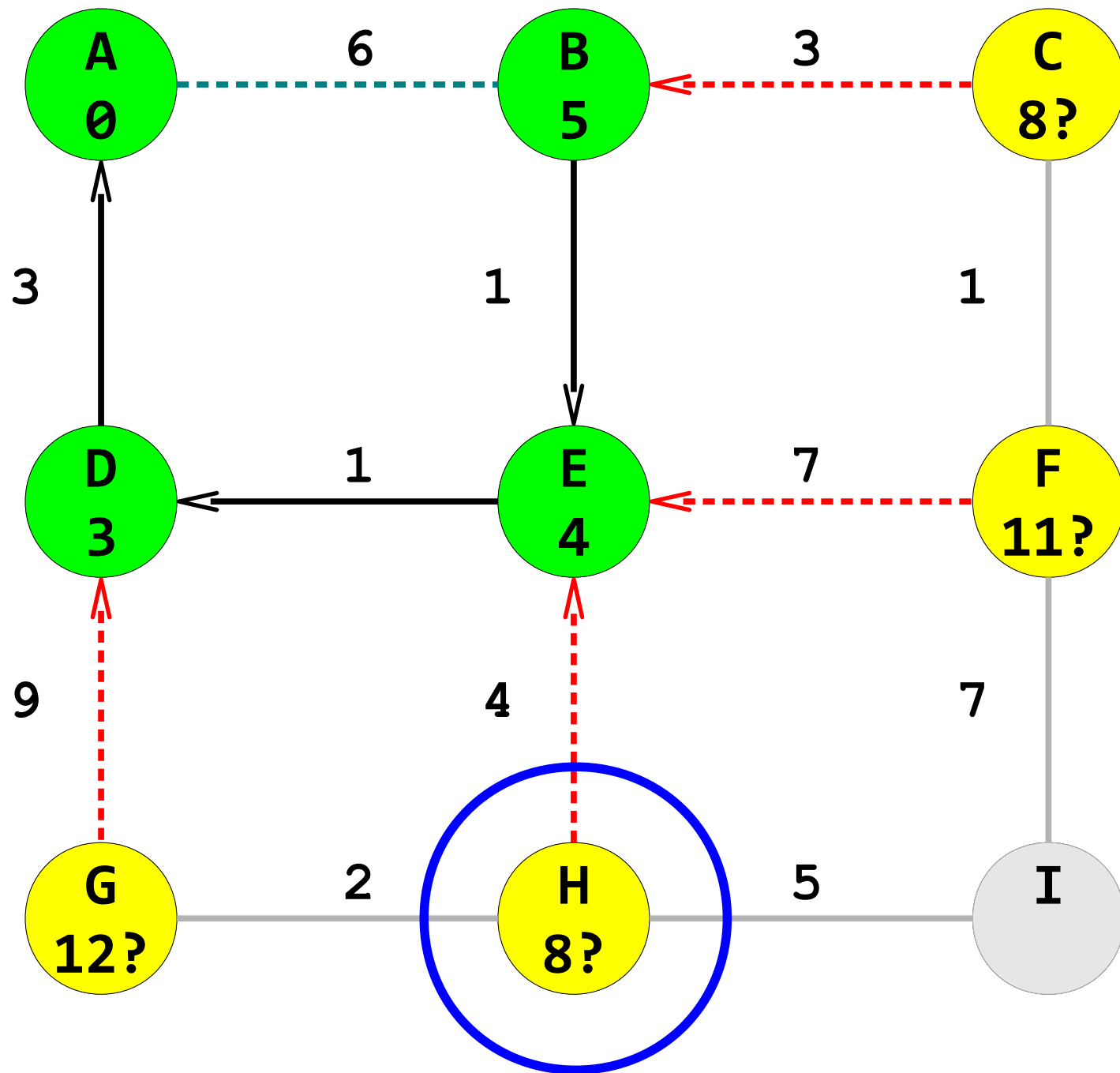
---

H  
8?

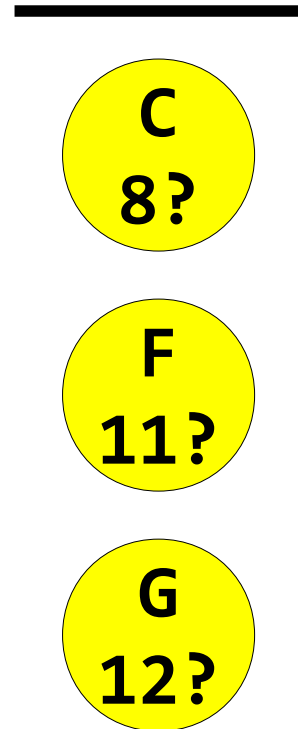
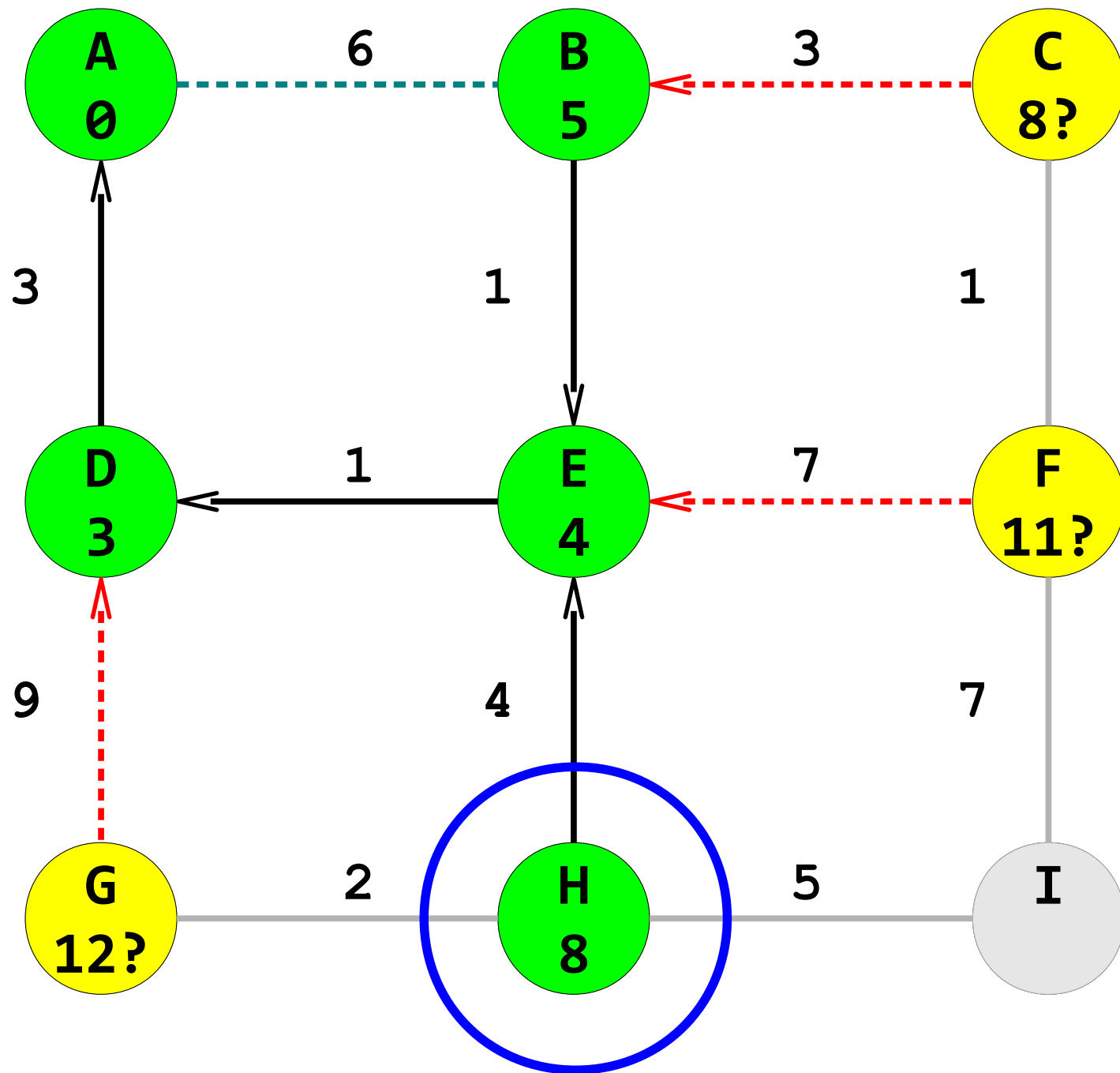
C  
8?

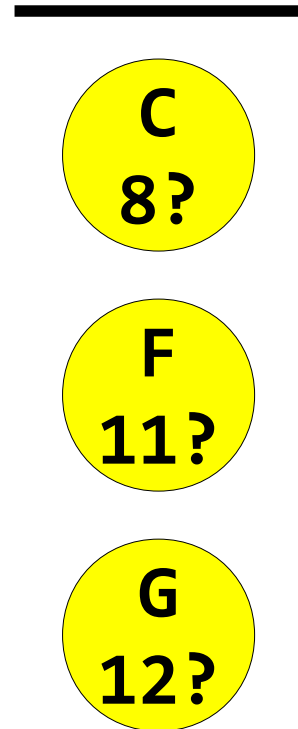
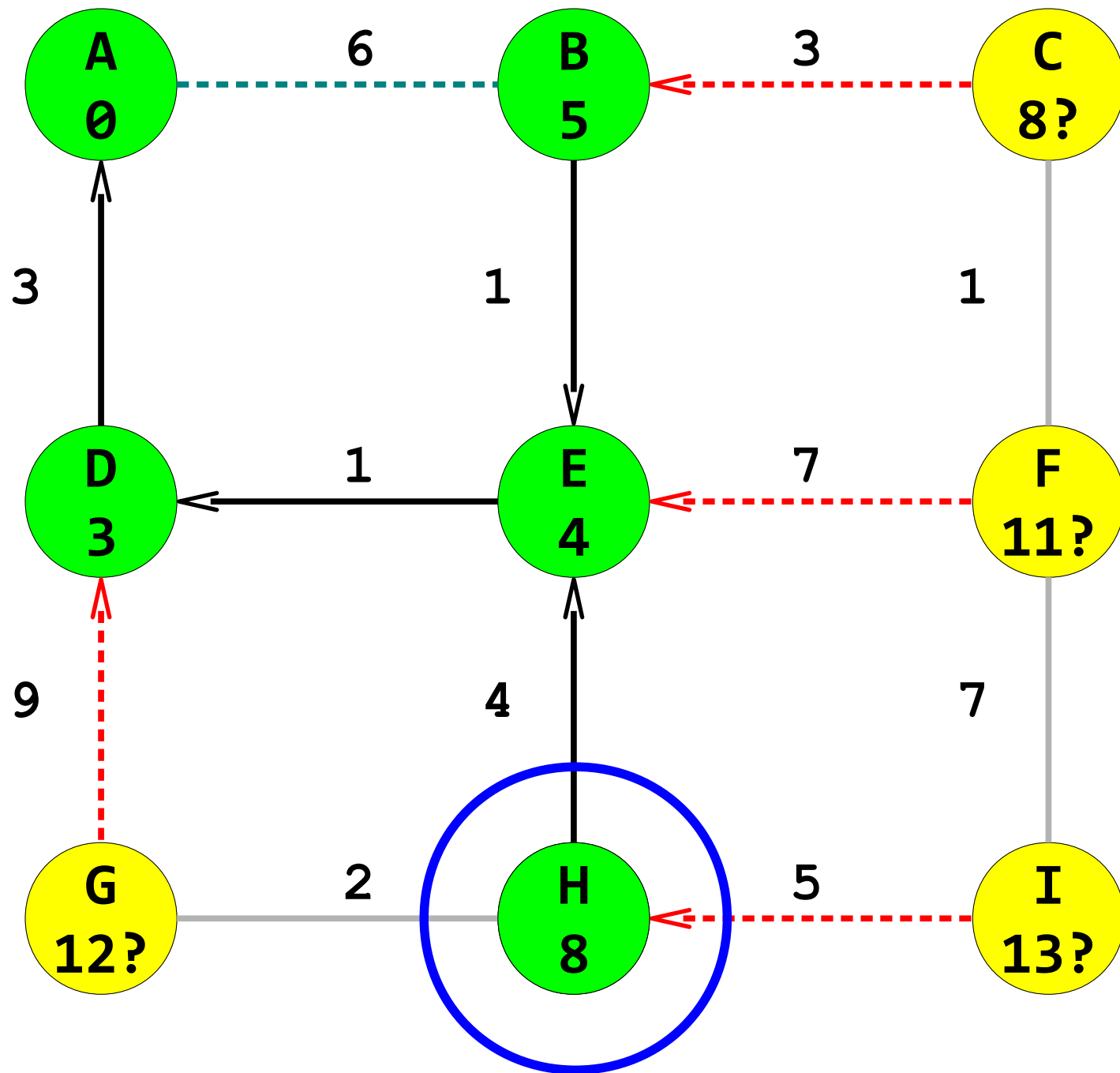
F  
11?

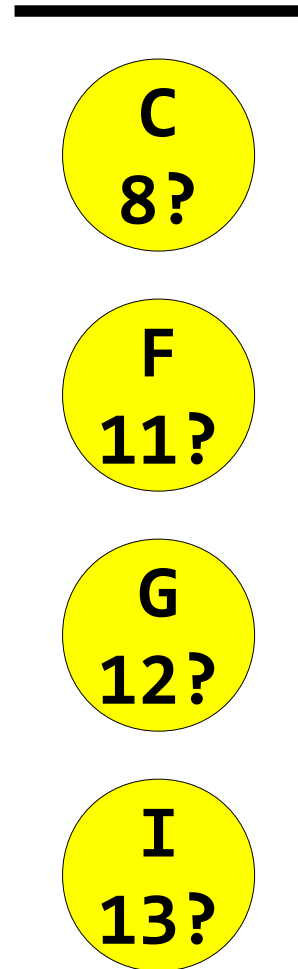
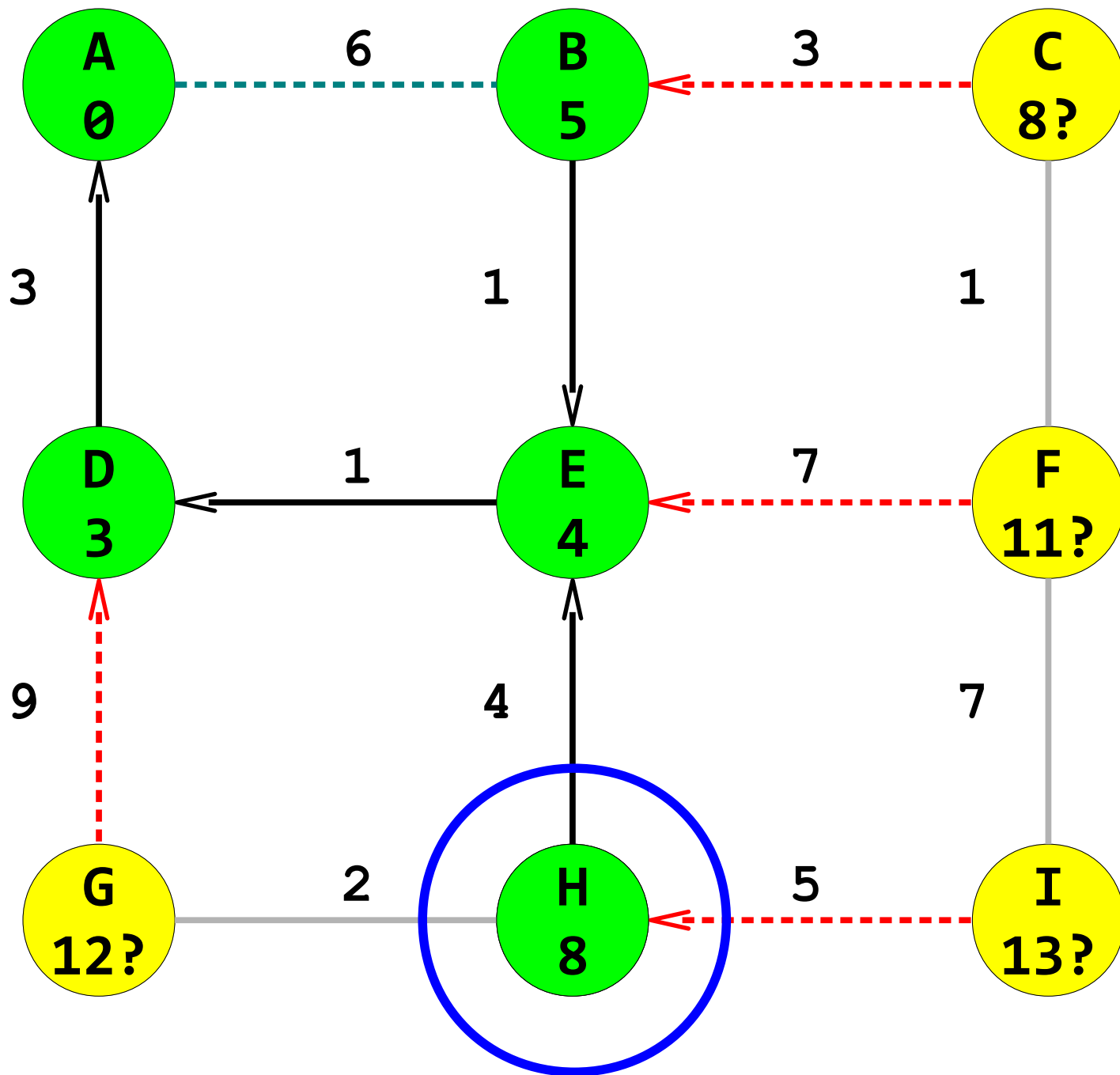
G  
12?

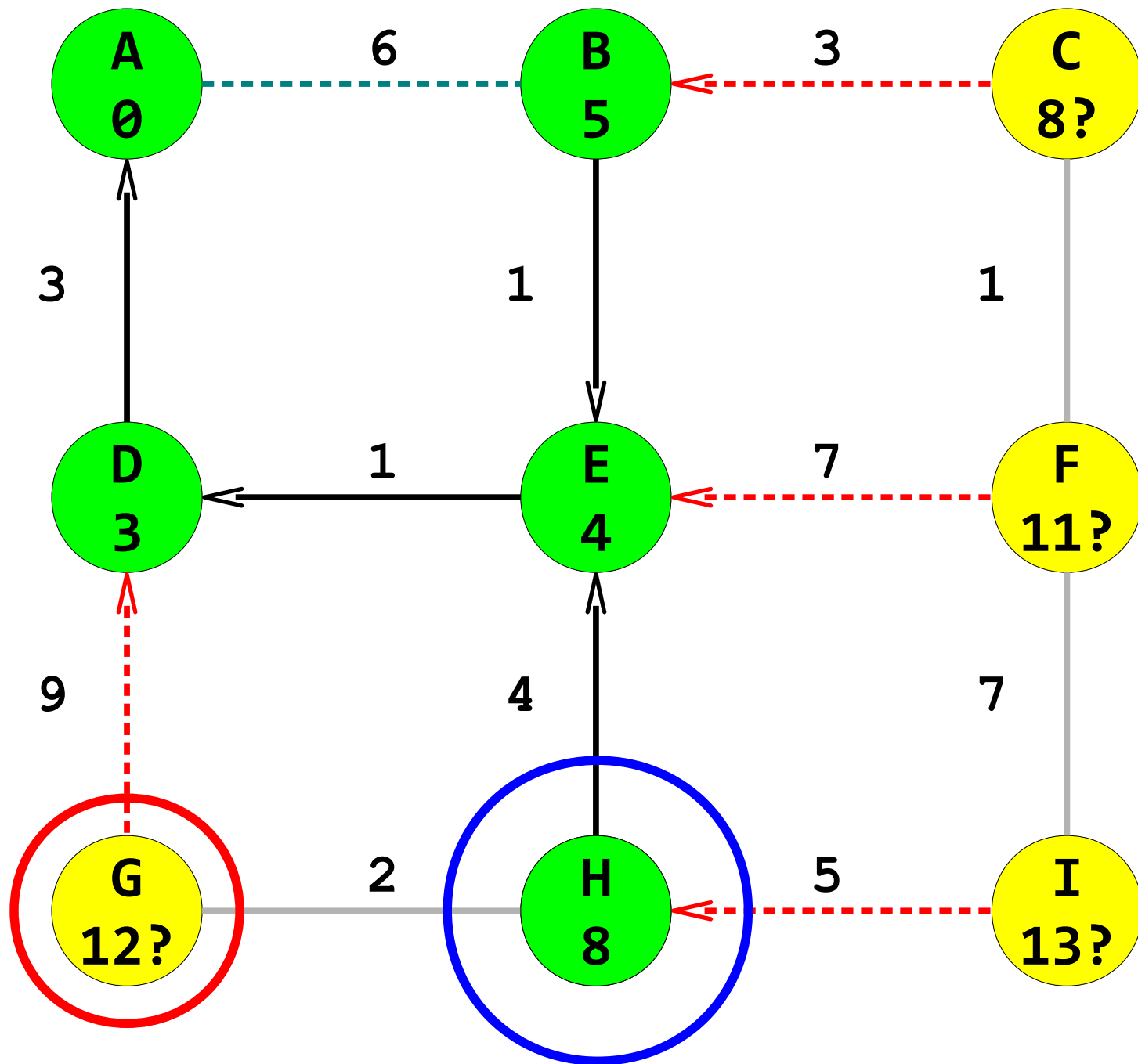




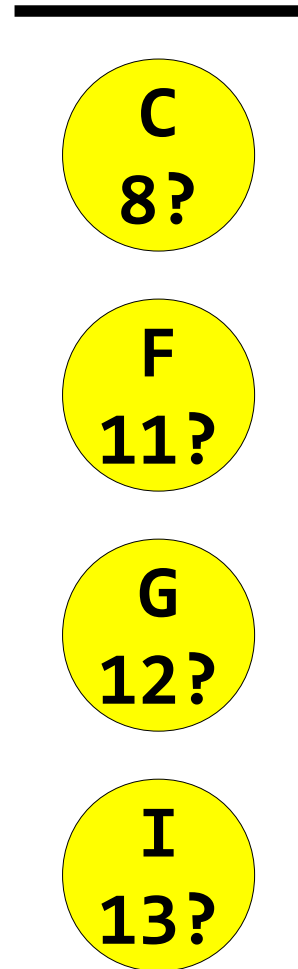
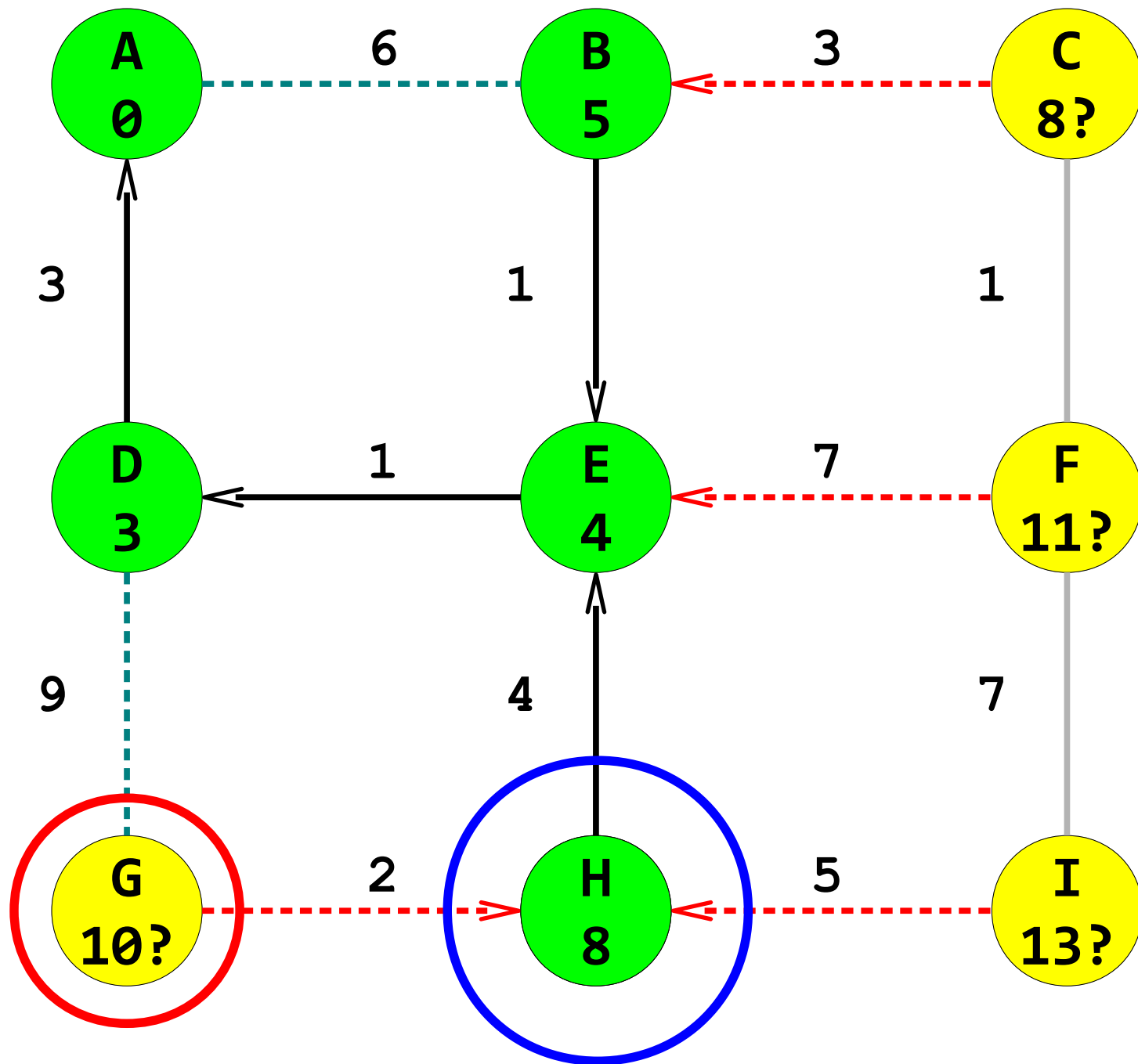


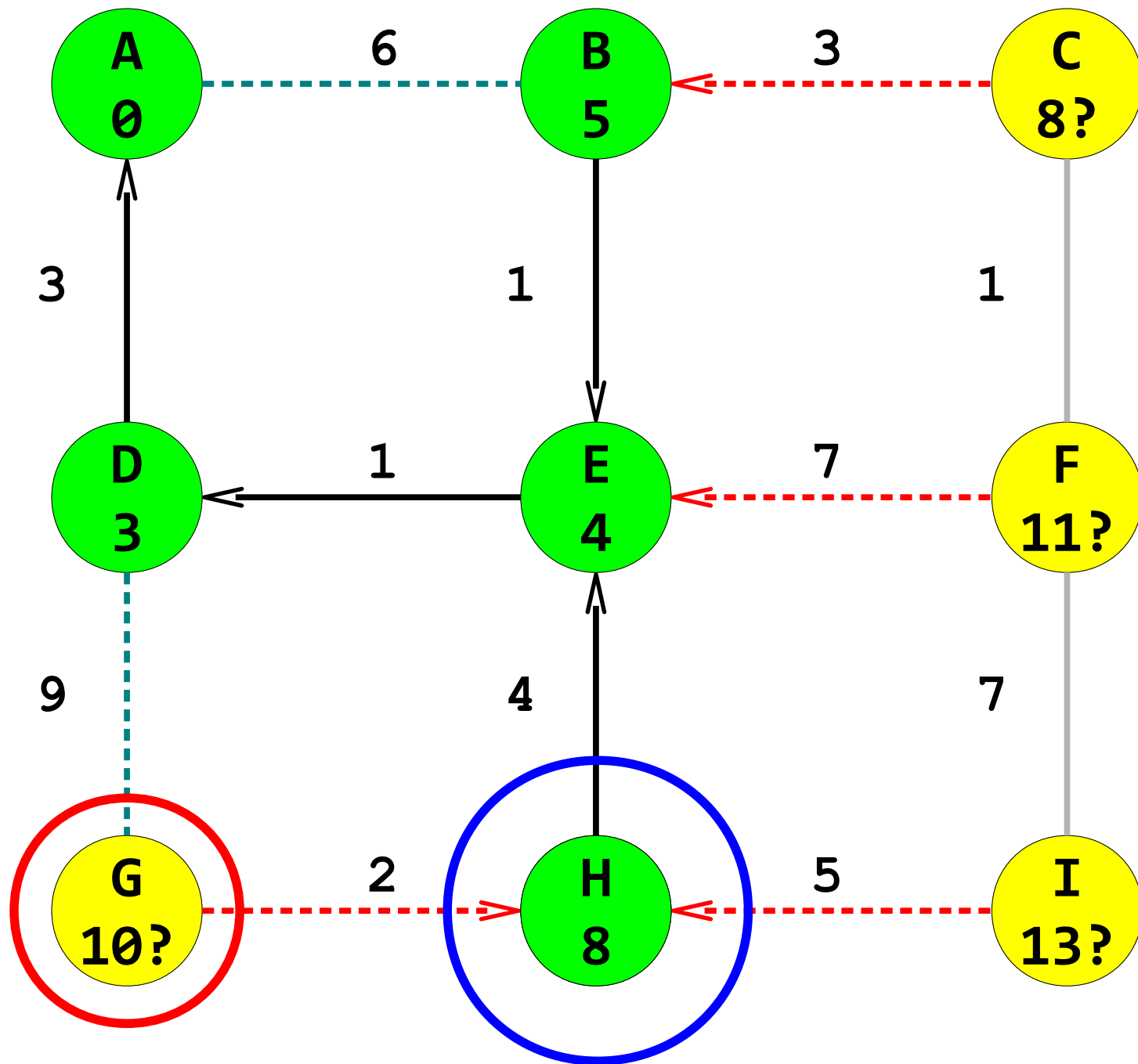




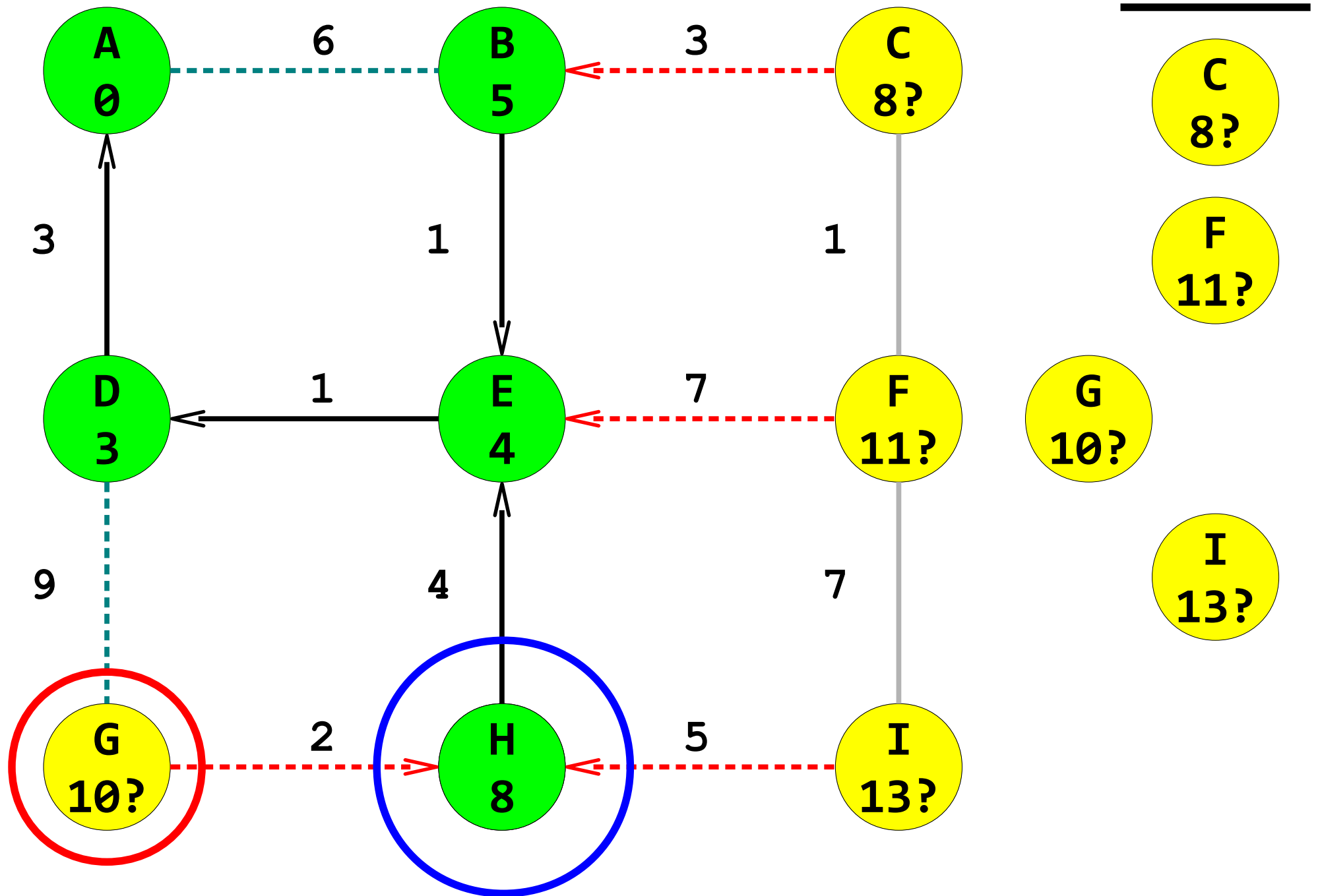


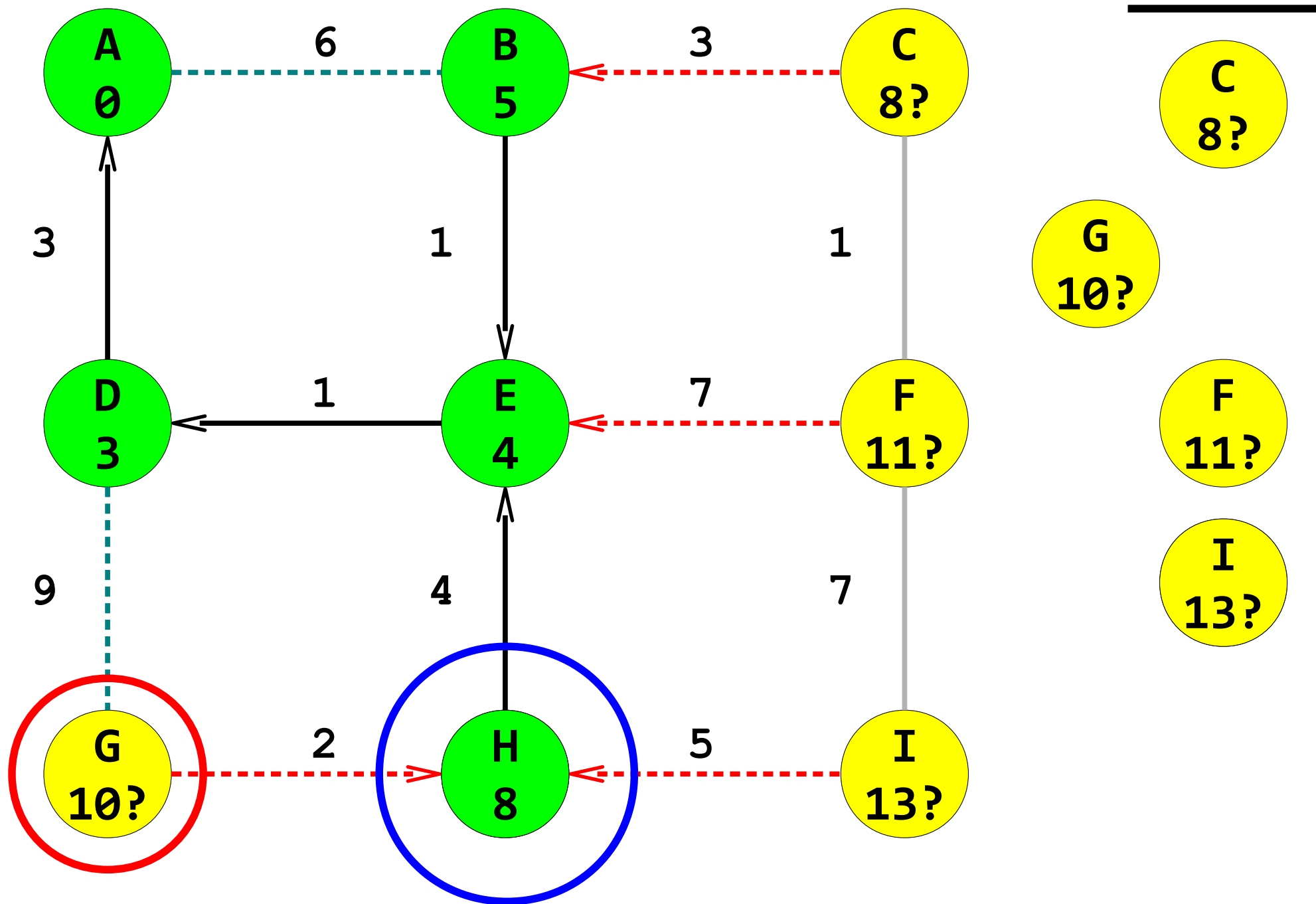
- 
- C  
8?
  - F  
11?
  - G  
12?
  - I  
13?



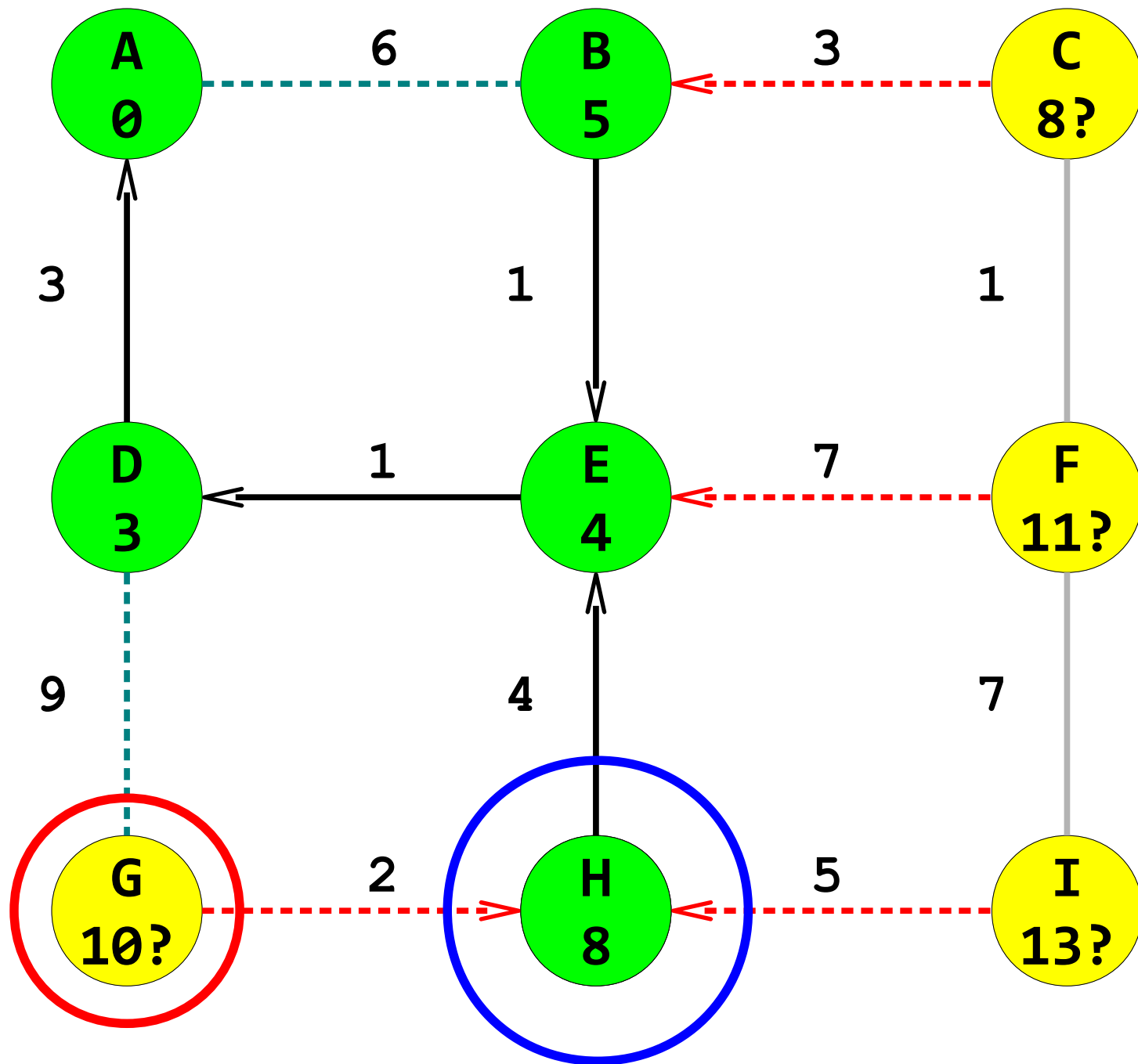


- 
- C 8?
  - F 11?
  - G 10?
  - I 13?

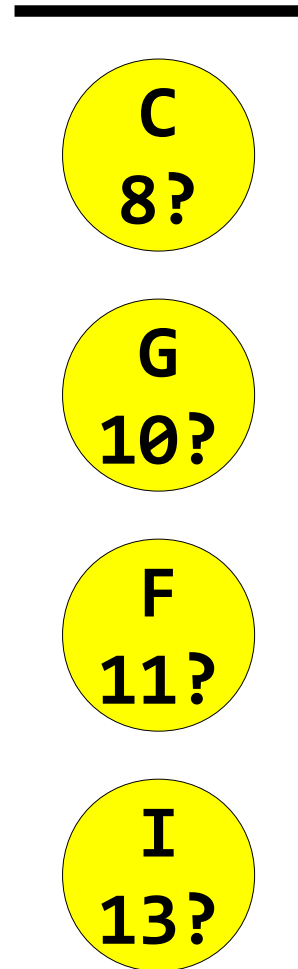
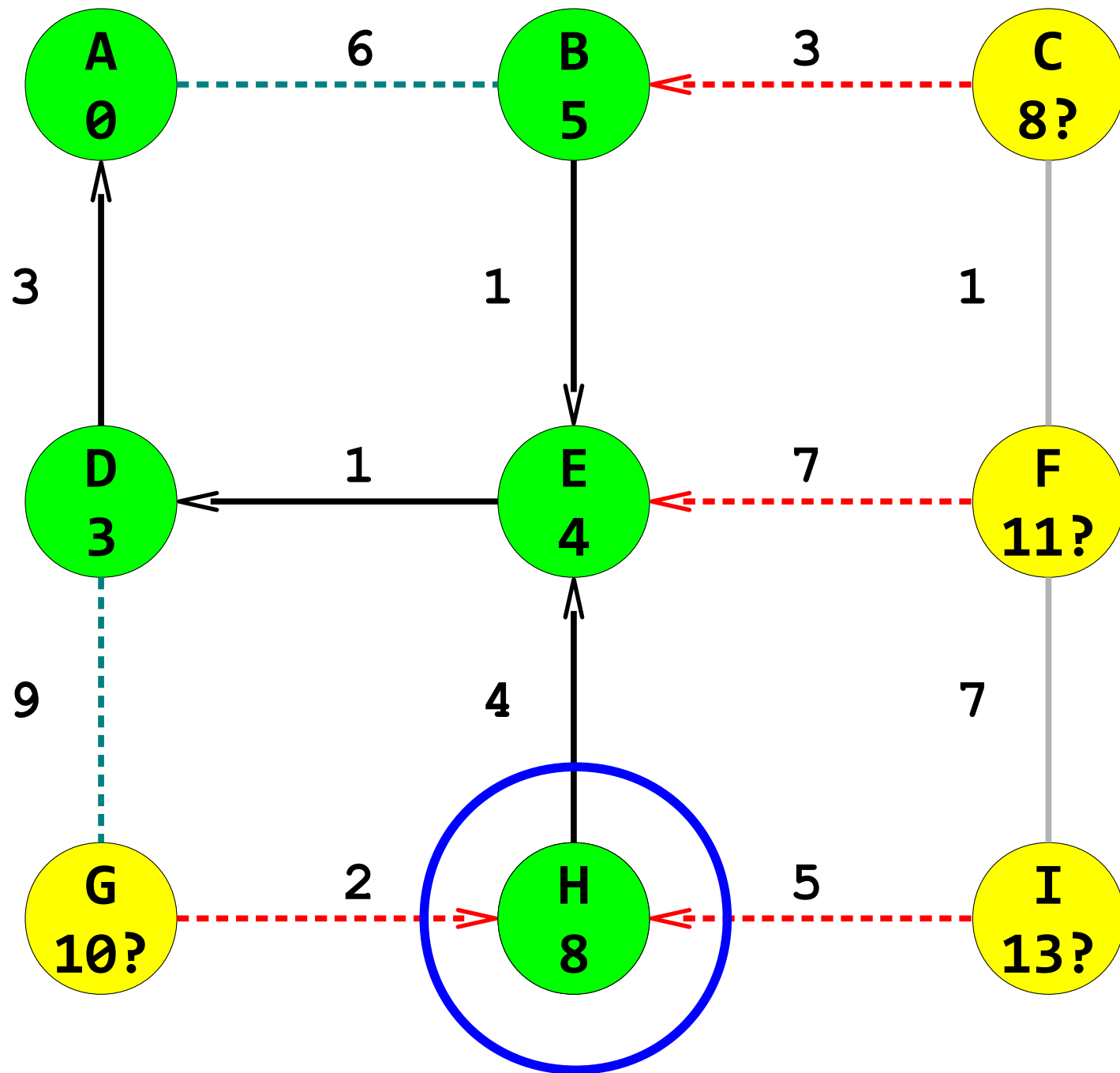


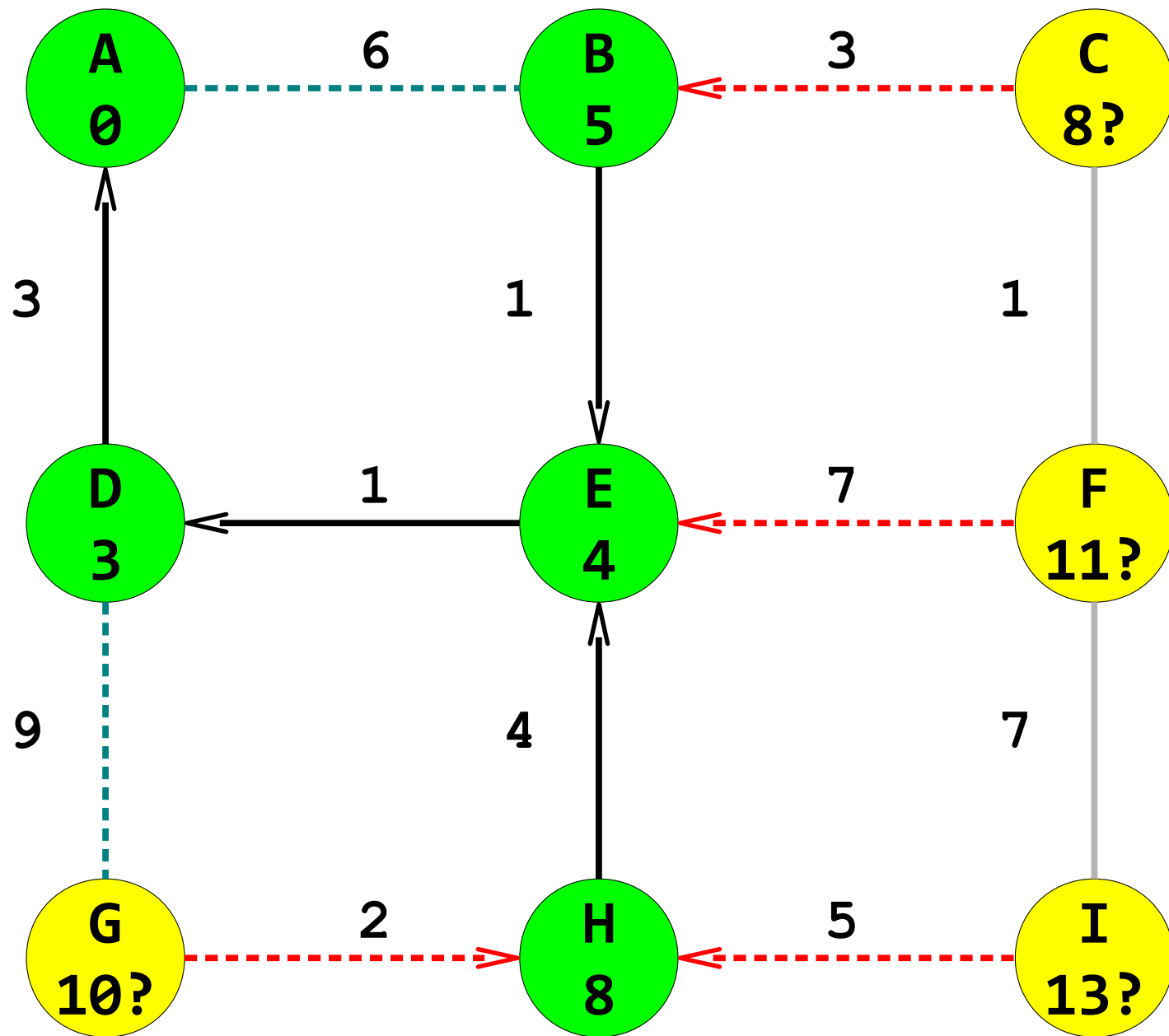






- 
- C  
8?
  - G  
10?
  - F  
11?
  - I  
13?





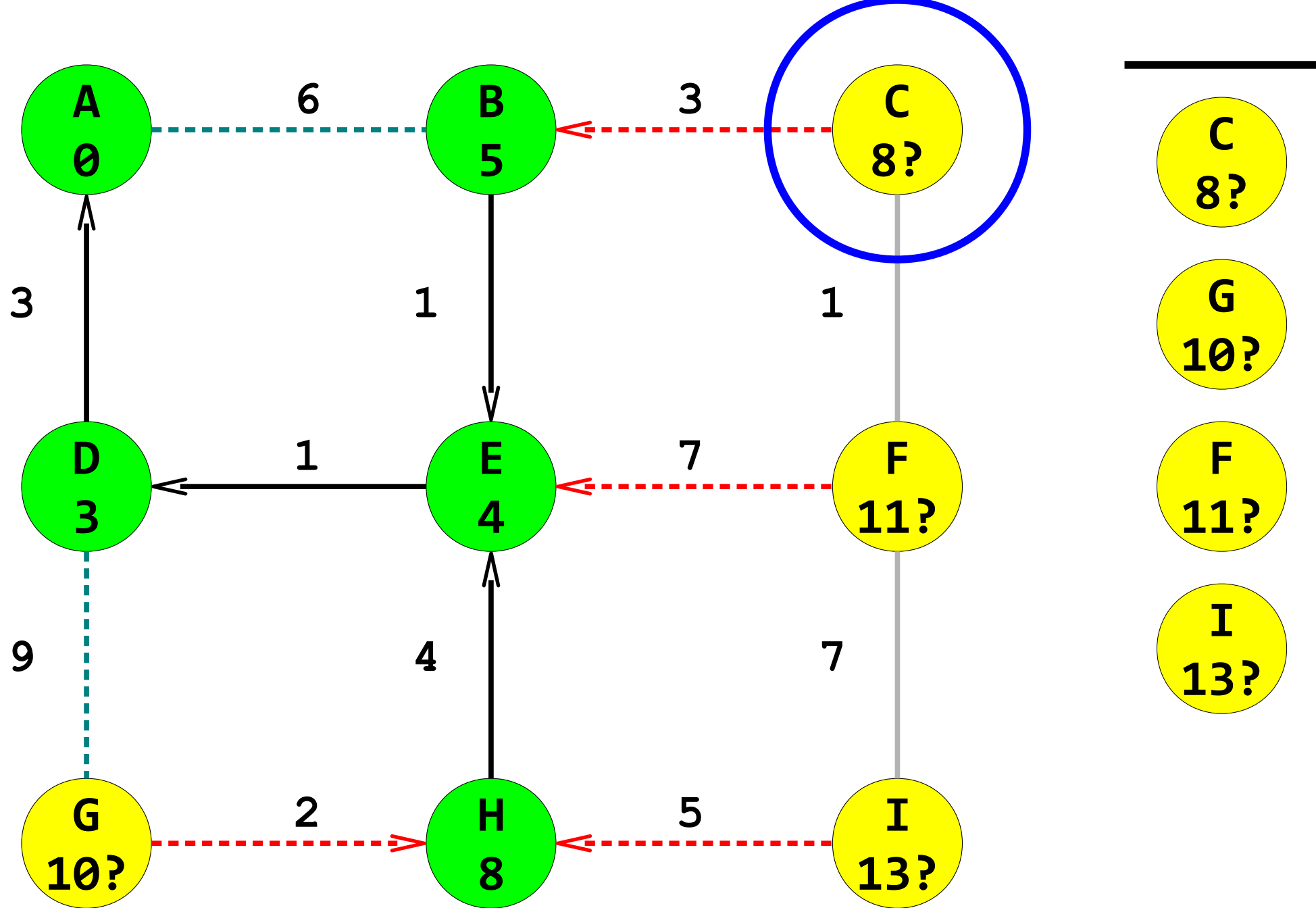
---

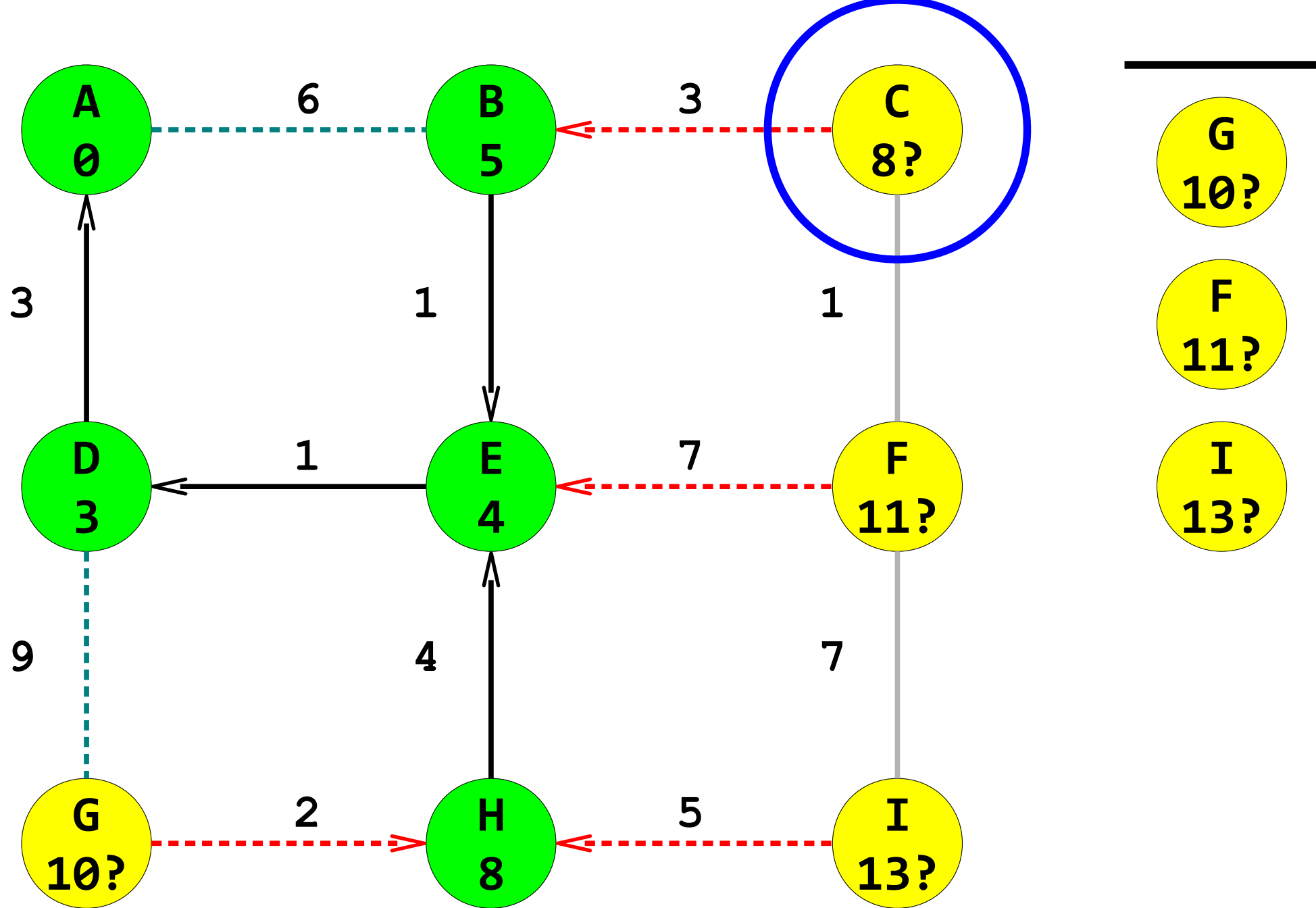
C  
8?

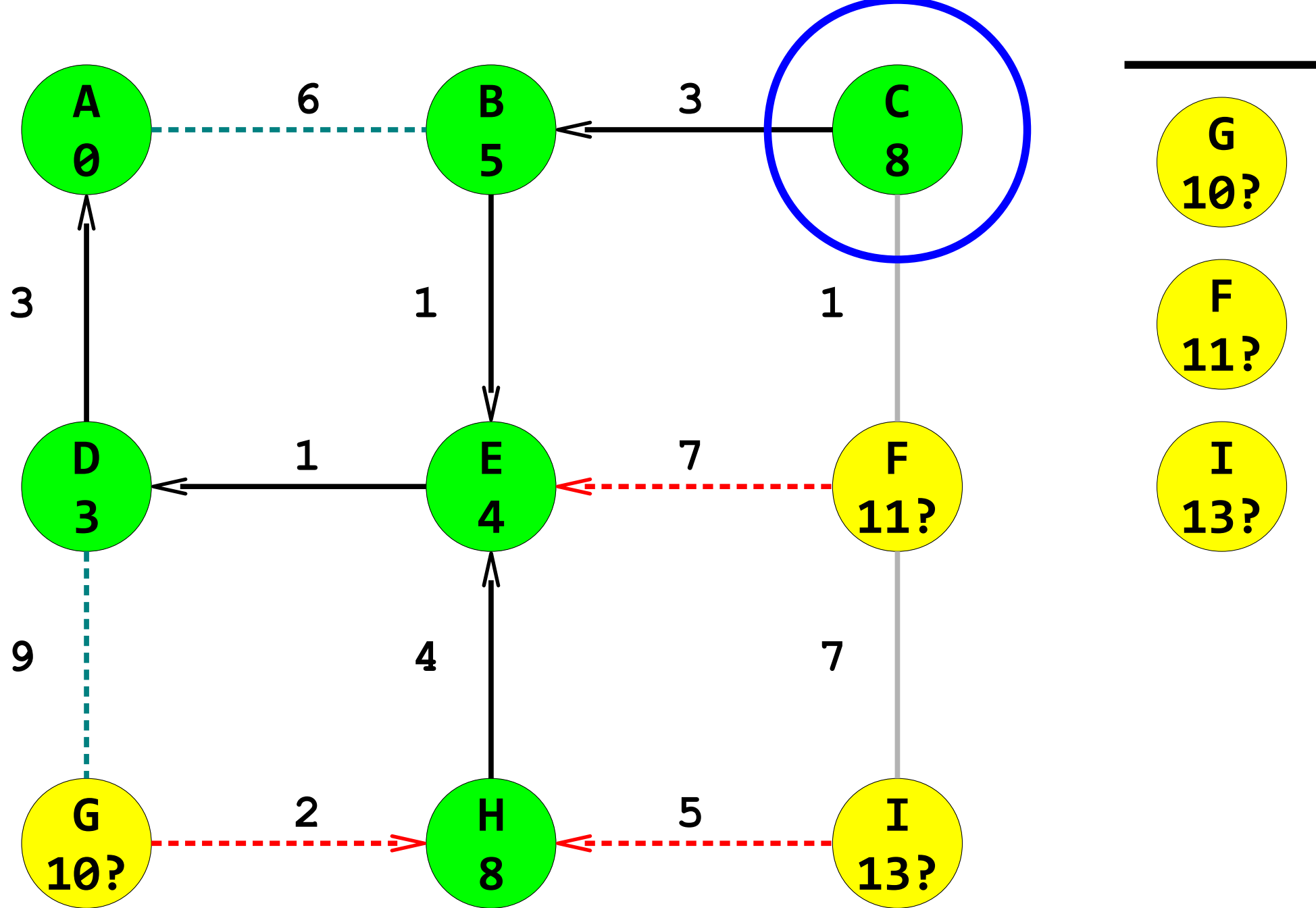
G  
10?

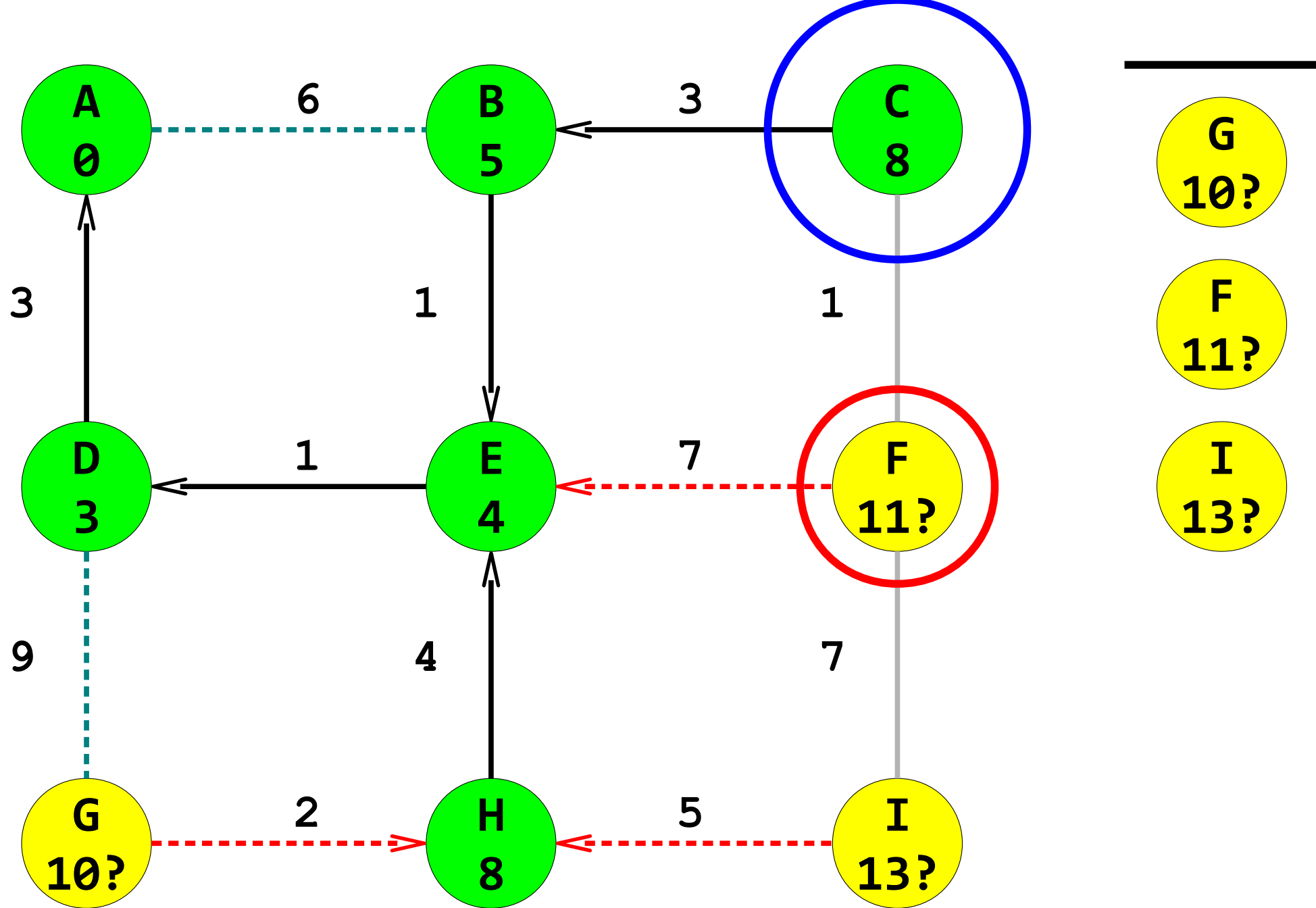
F  
11?

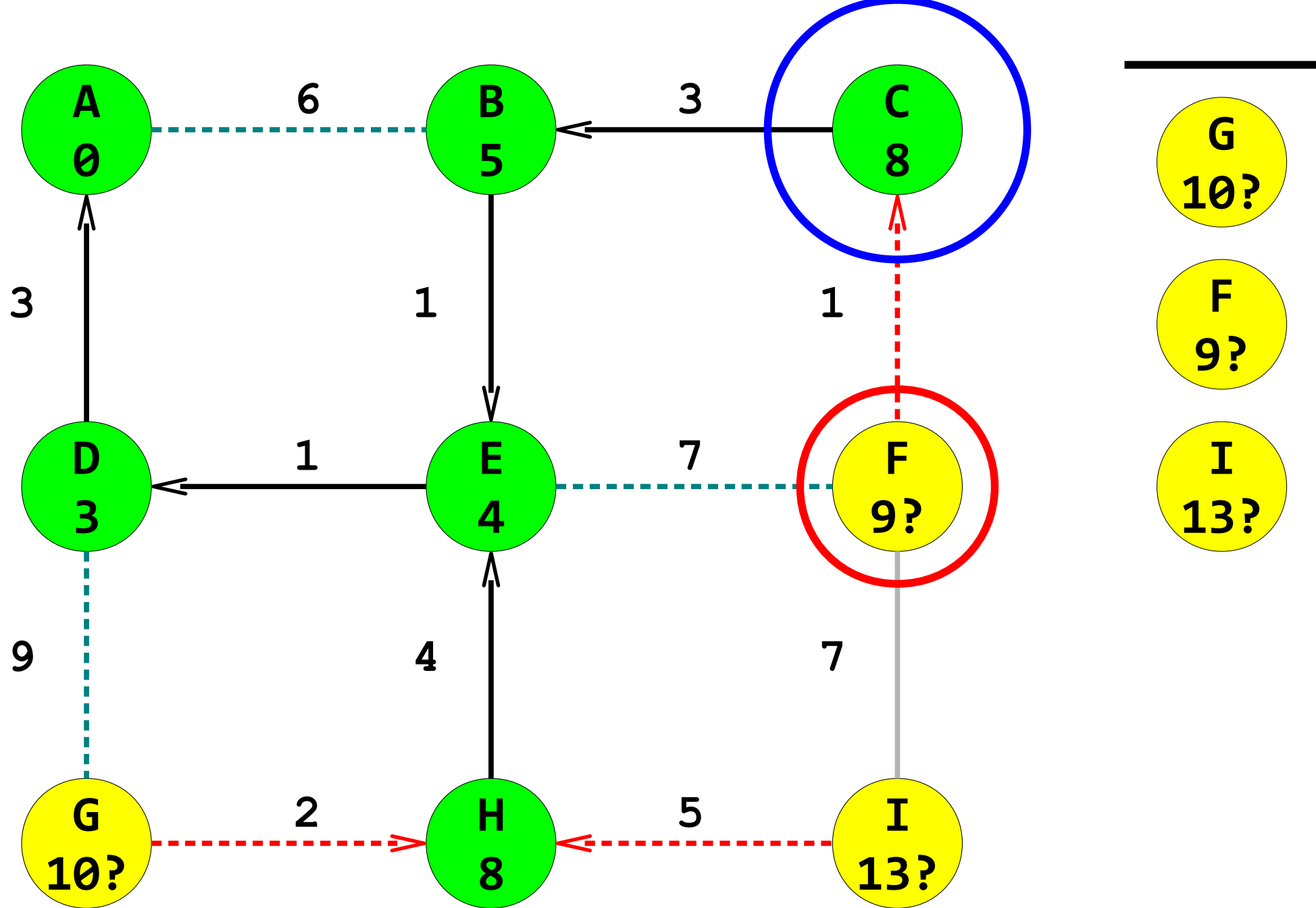
I  
13?



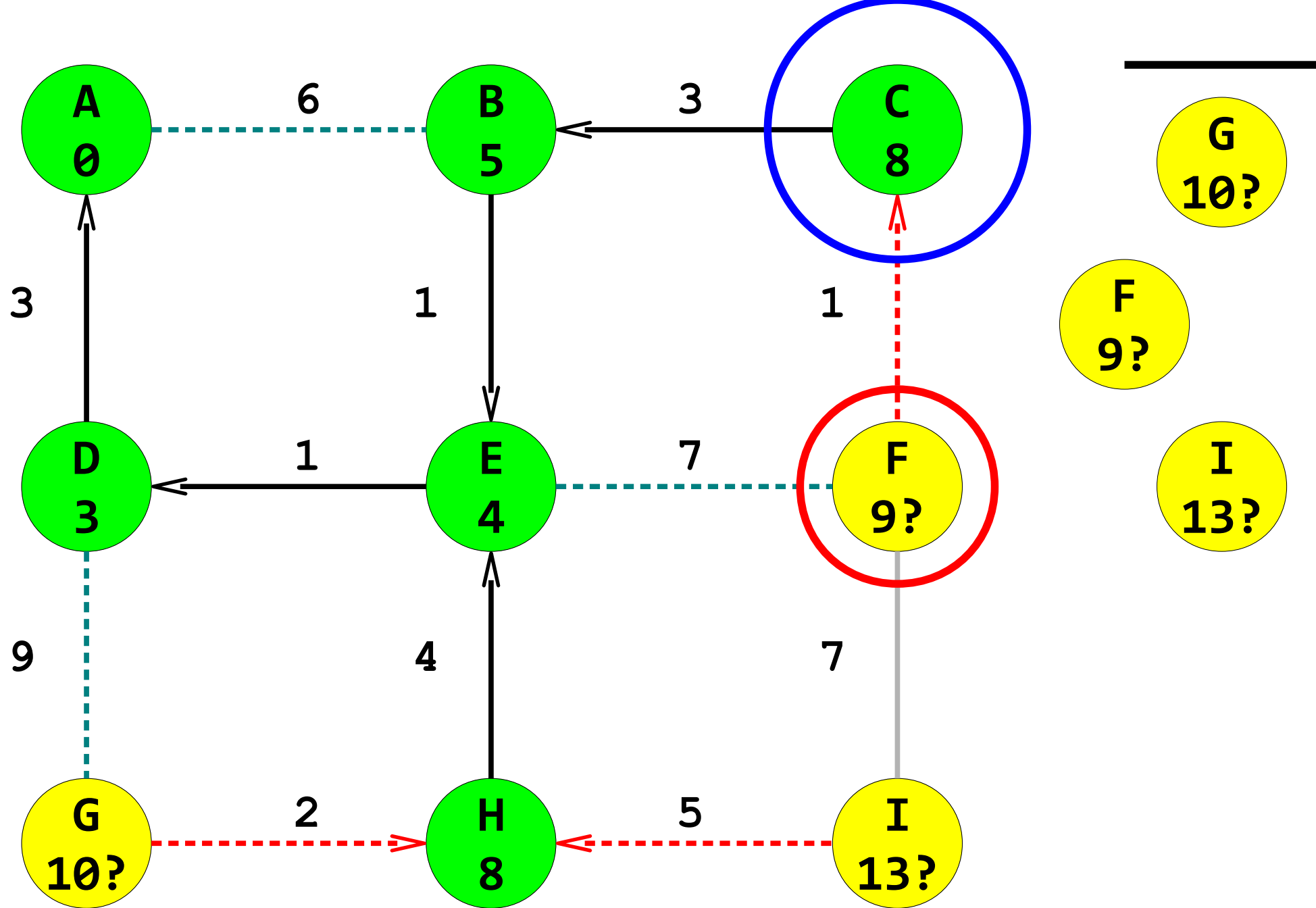


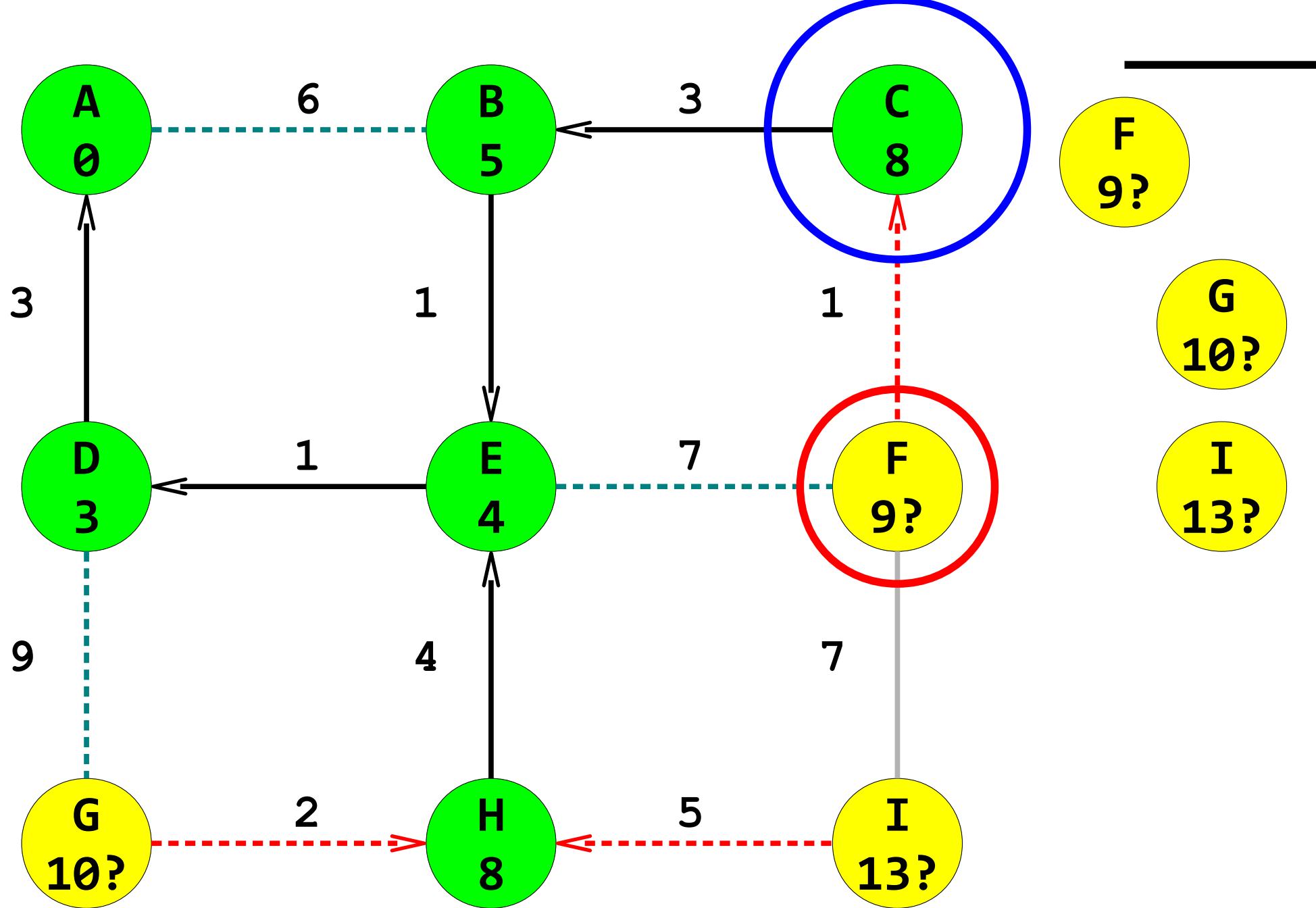


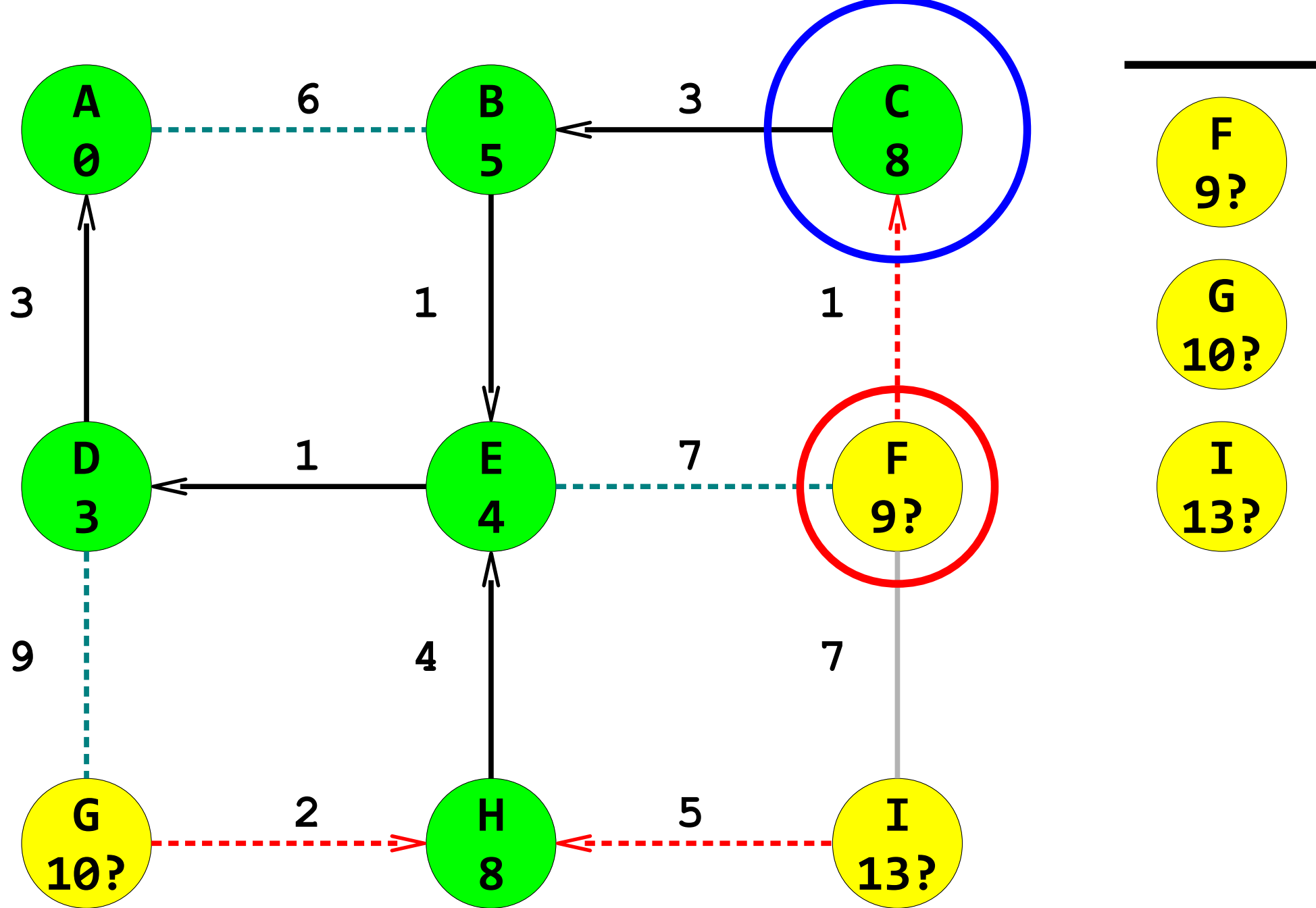


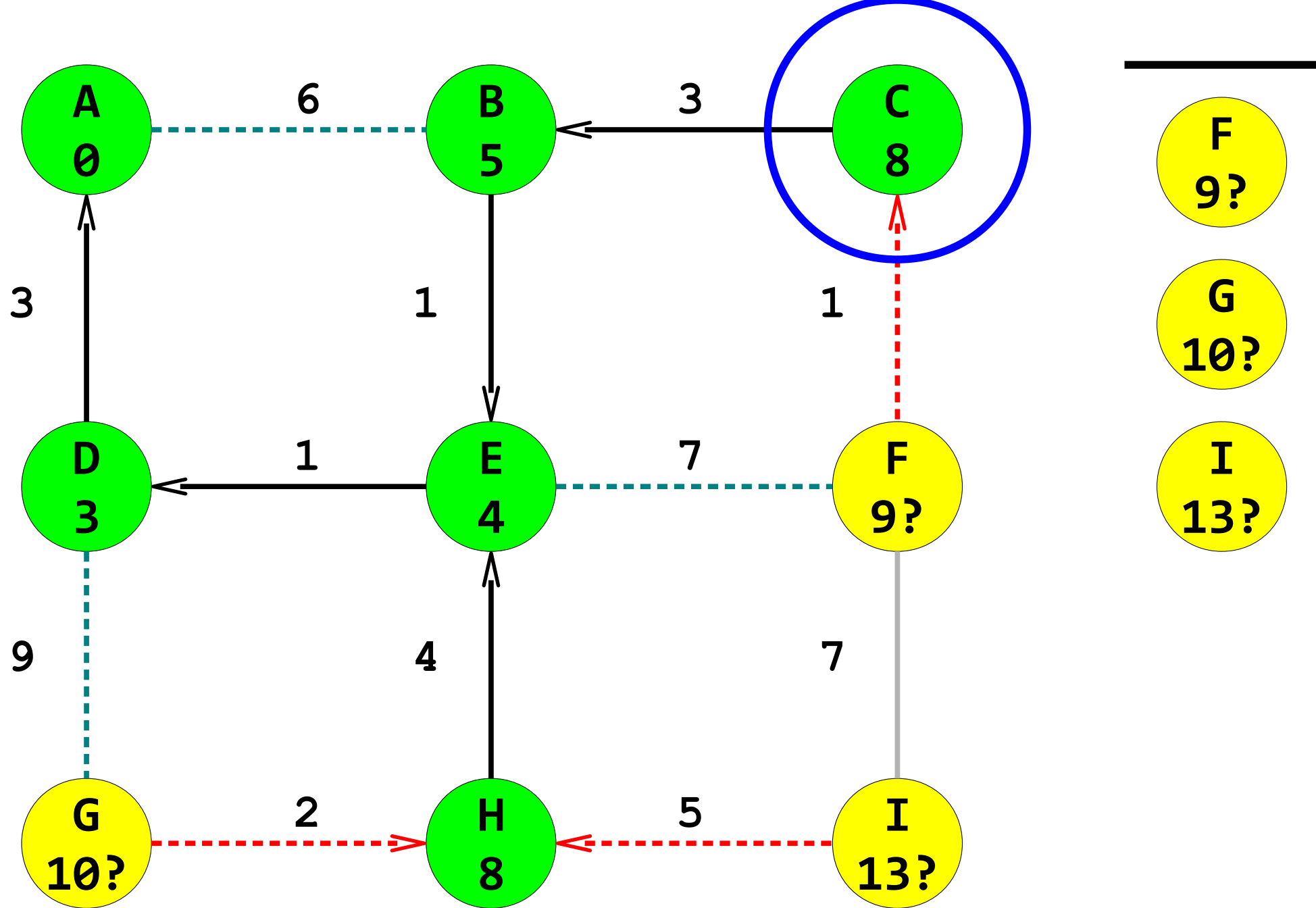


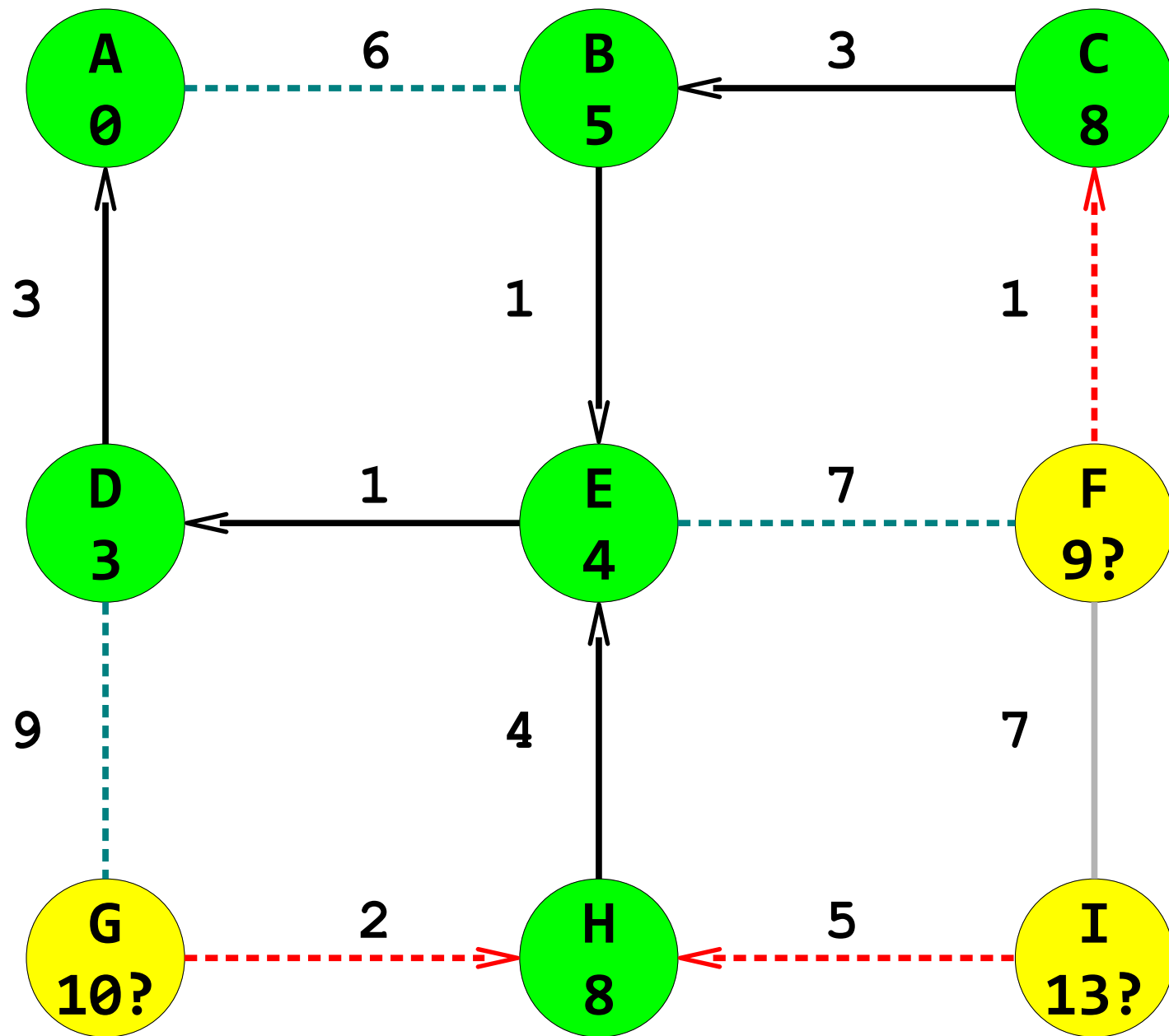










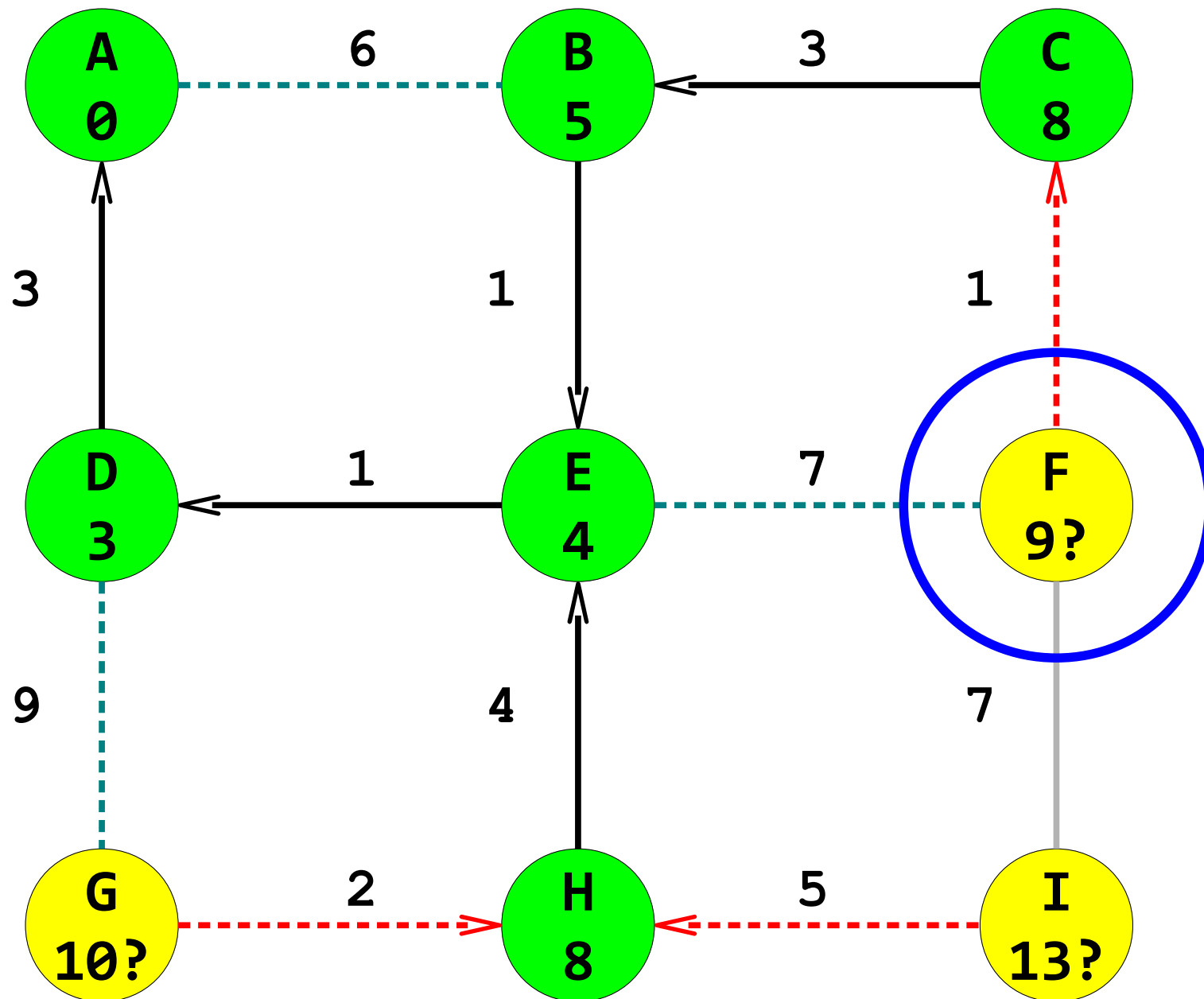


---

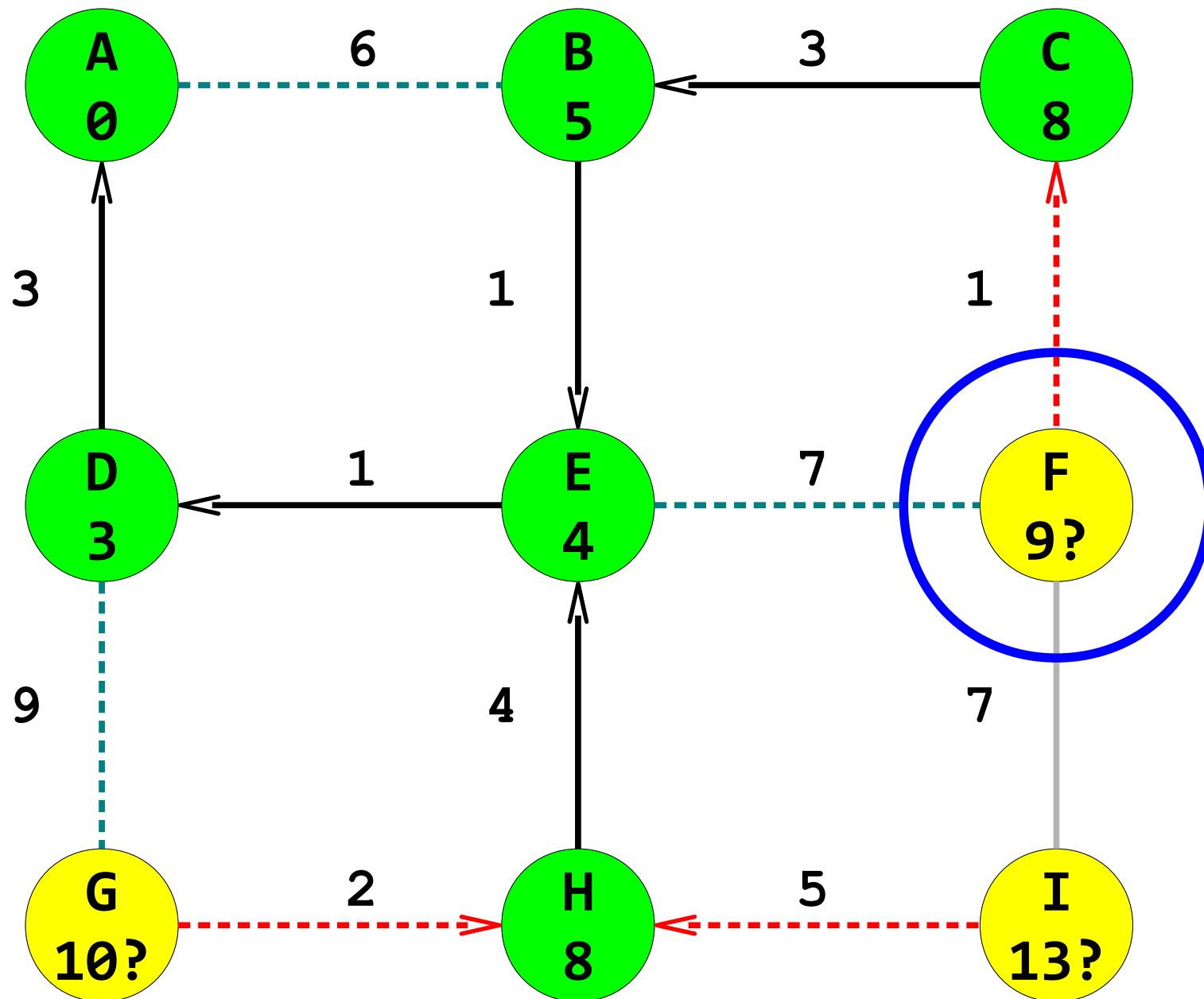
**F**  
**9?**

**G**  
**10?**

**I**  
**13?**



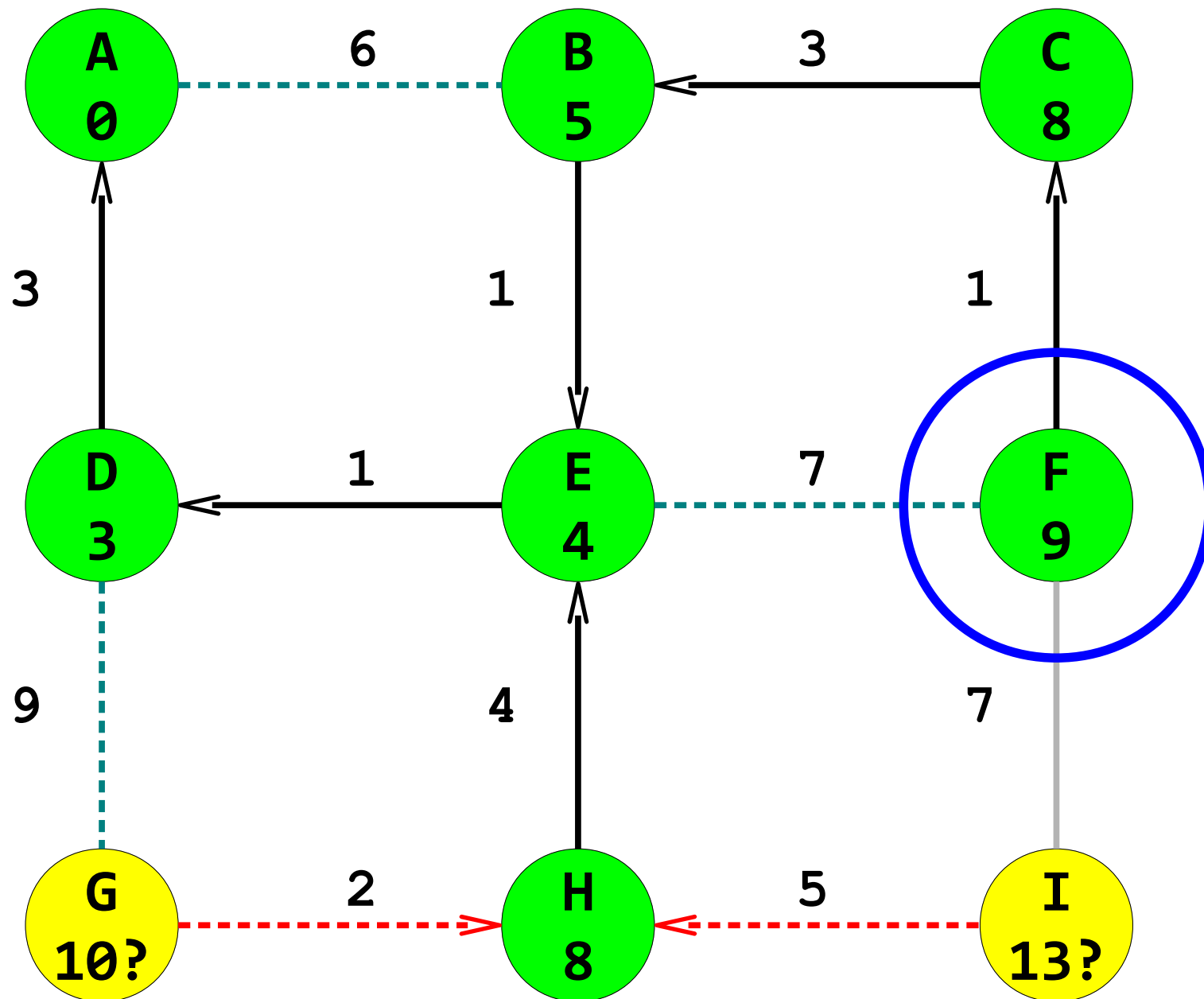
- 
- F 9?
  - G 10?
  - I 13?



---

G  
10?

I  
13?

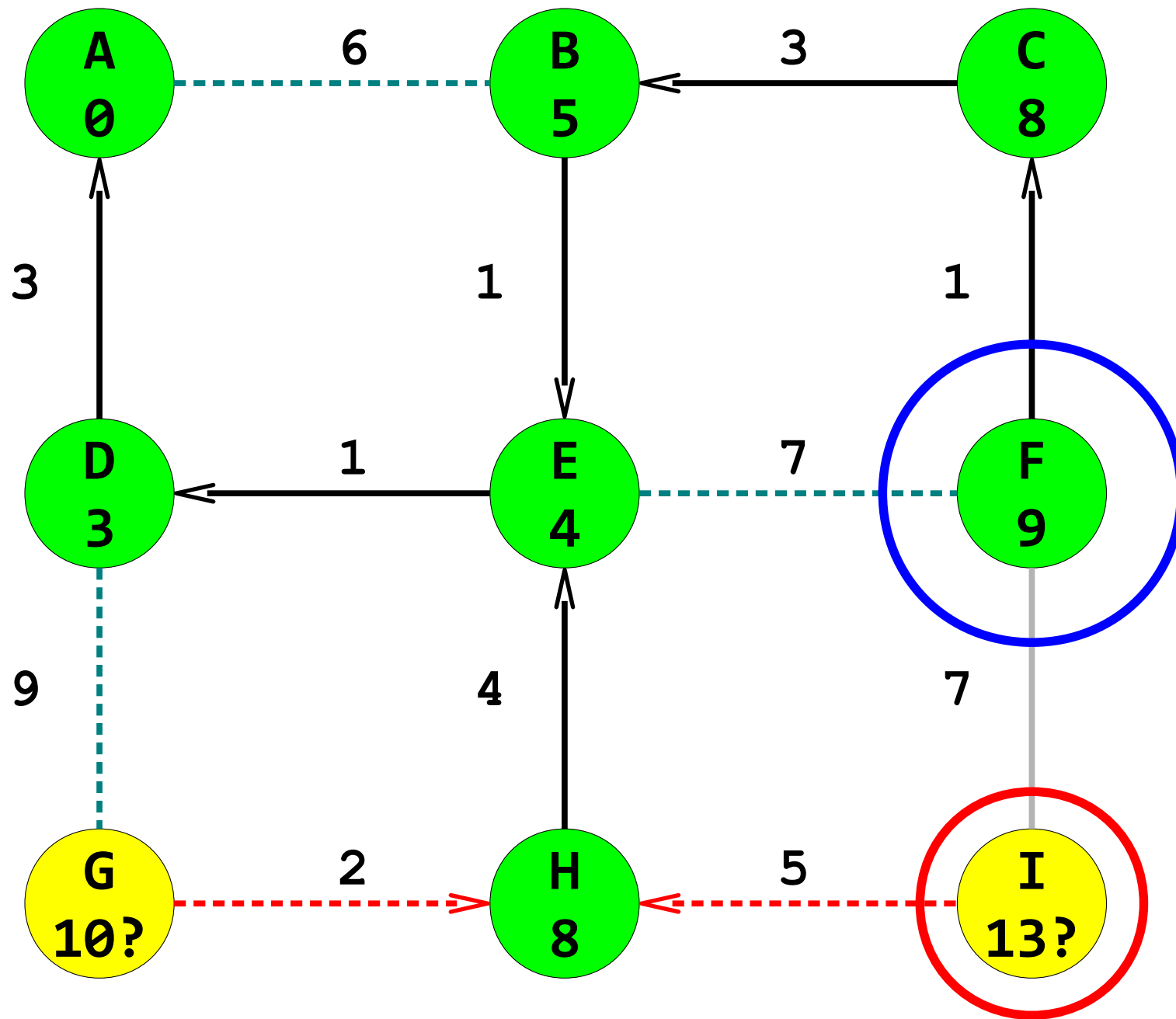


---

**G**  
**10?**

**I**  
**13?**

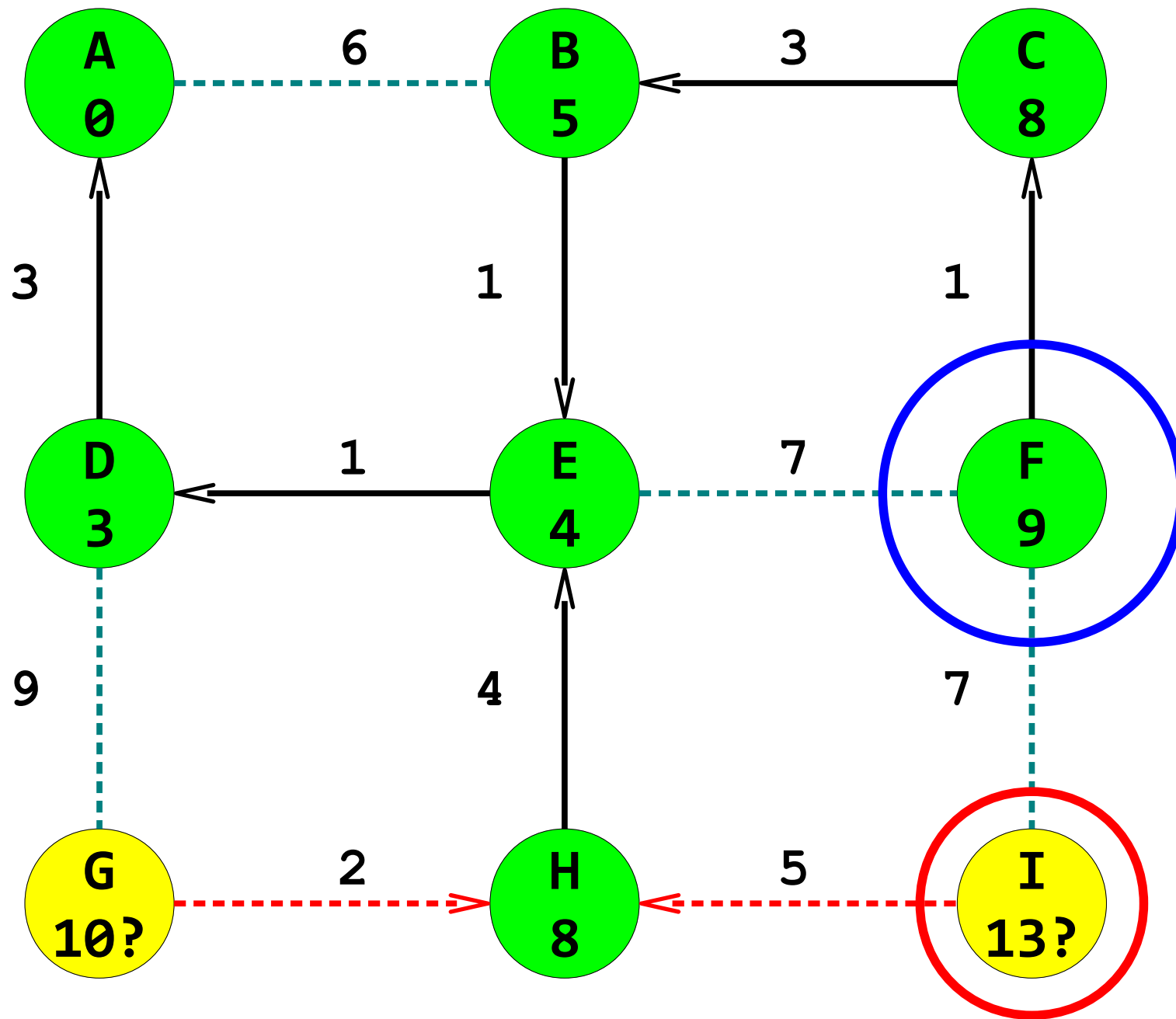




---

G  
10?

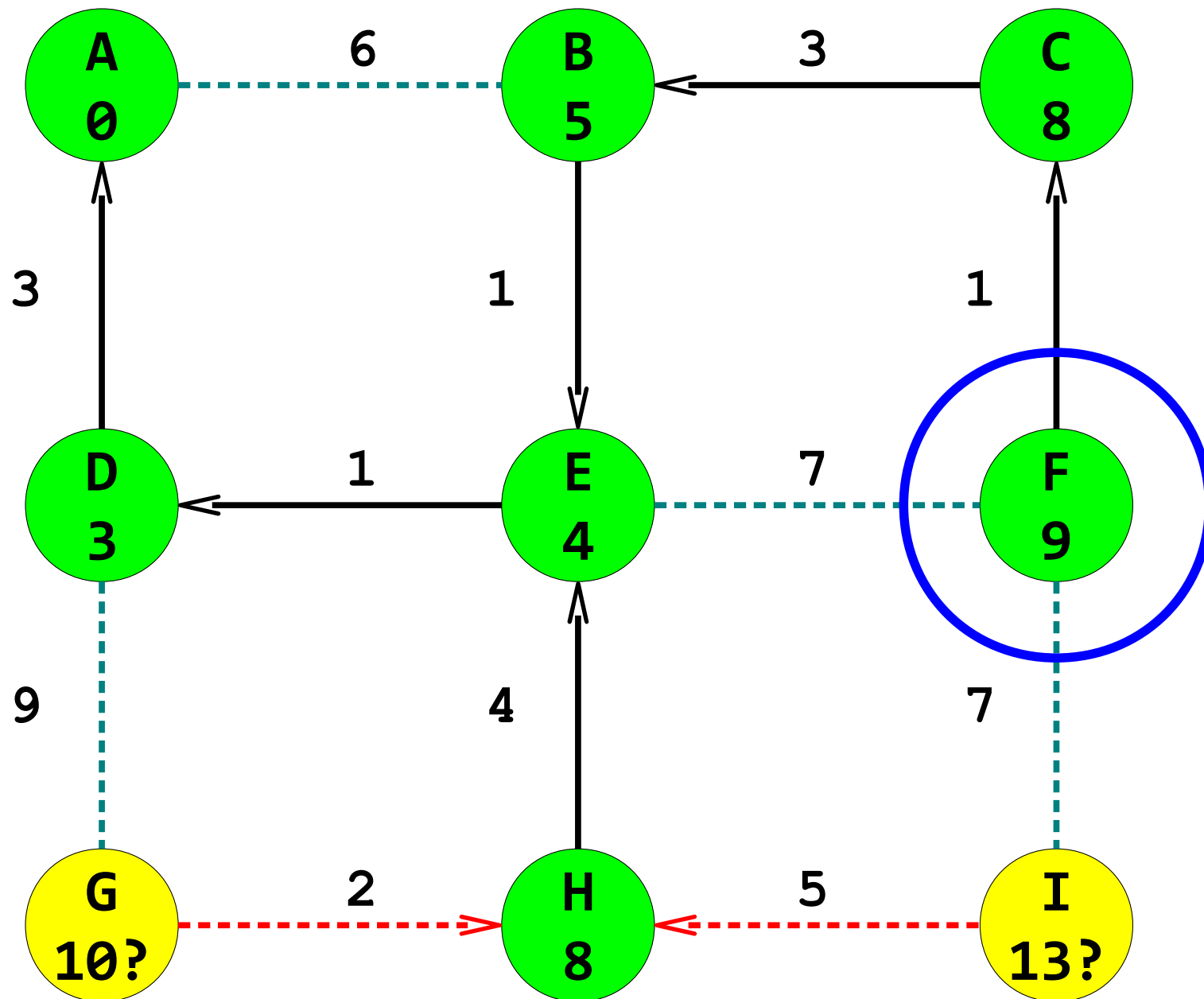
I  
13?



---

**G**  
**10?**

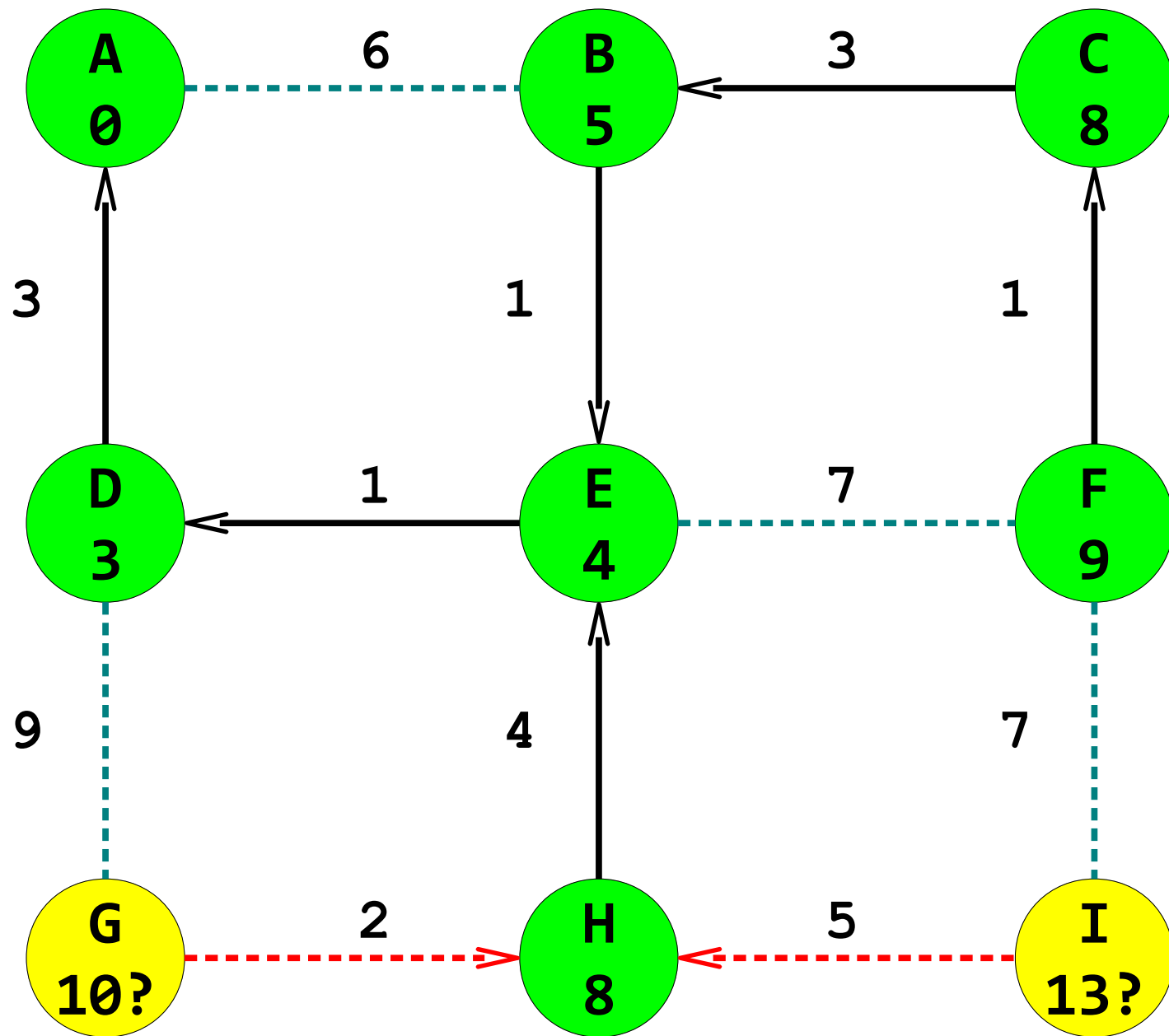
**I**  
**13?**



---

**G**  
**10?**

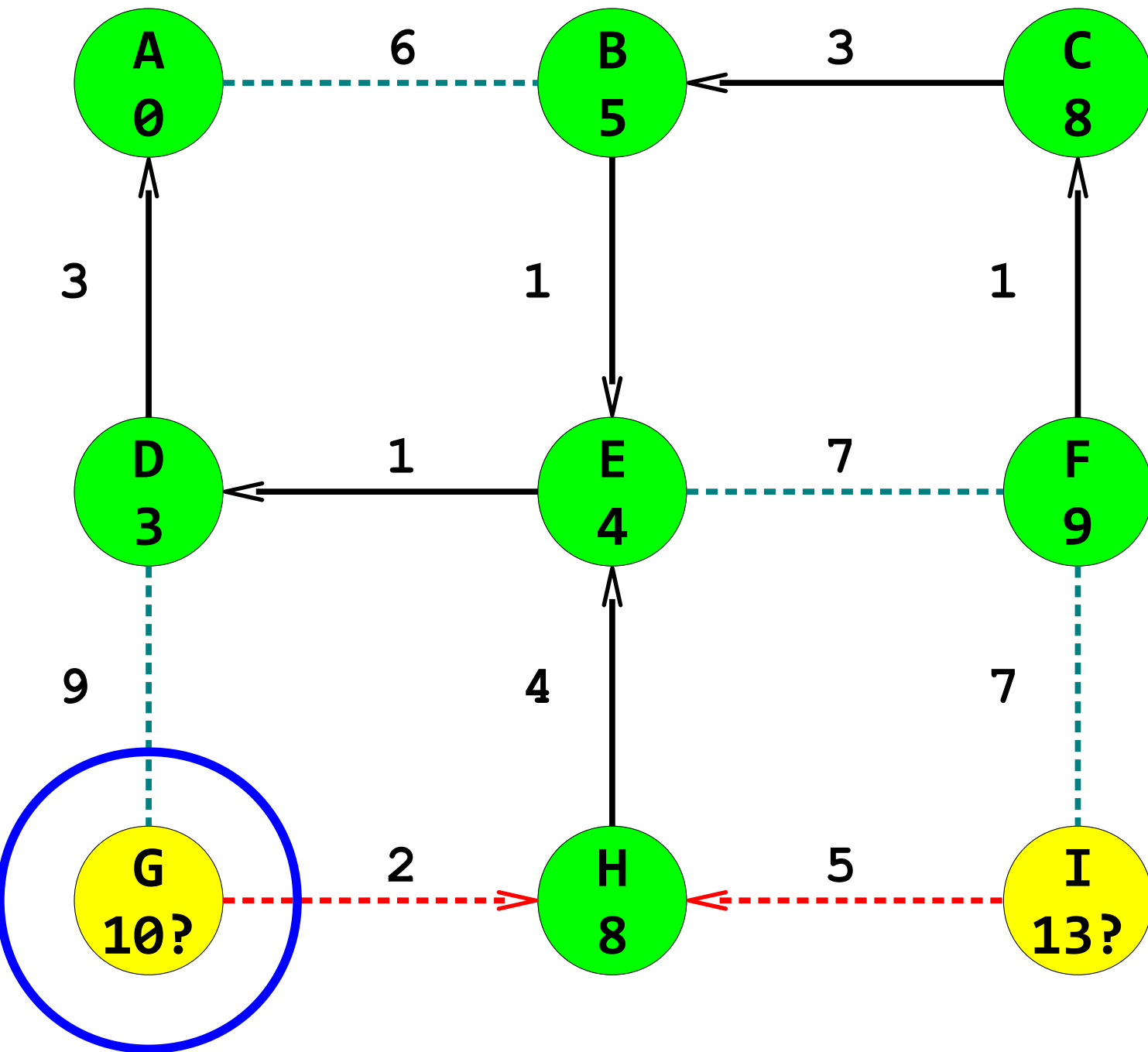
**I**  
**13?**



---

**G**  
**10?**

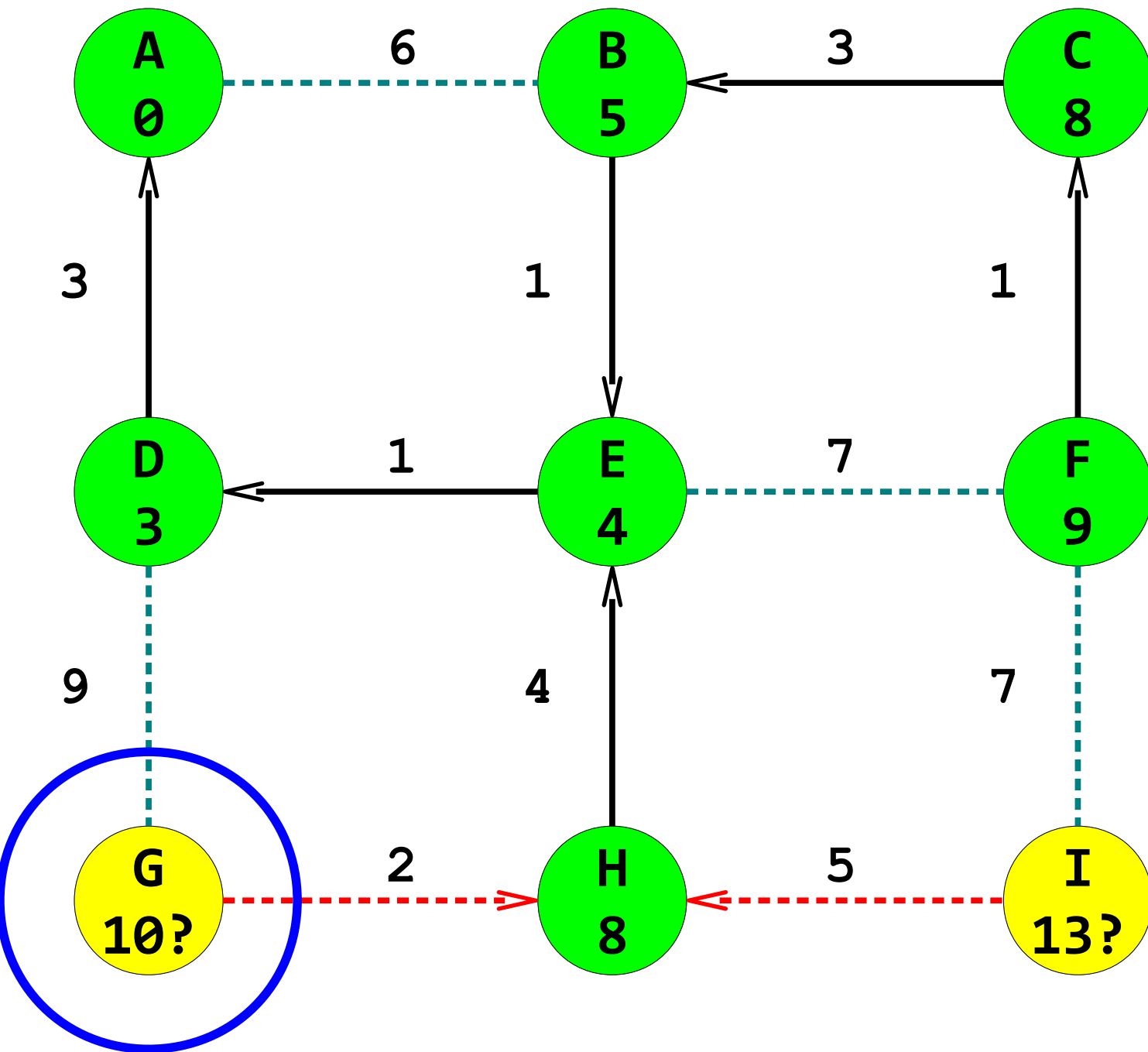
**I**  
**13?**



---

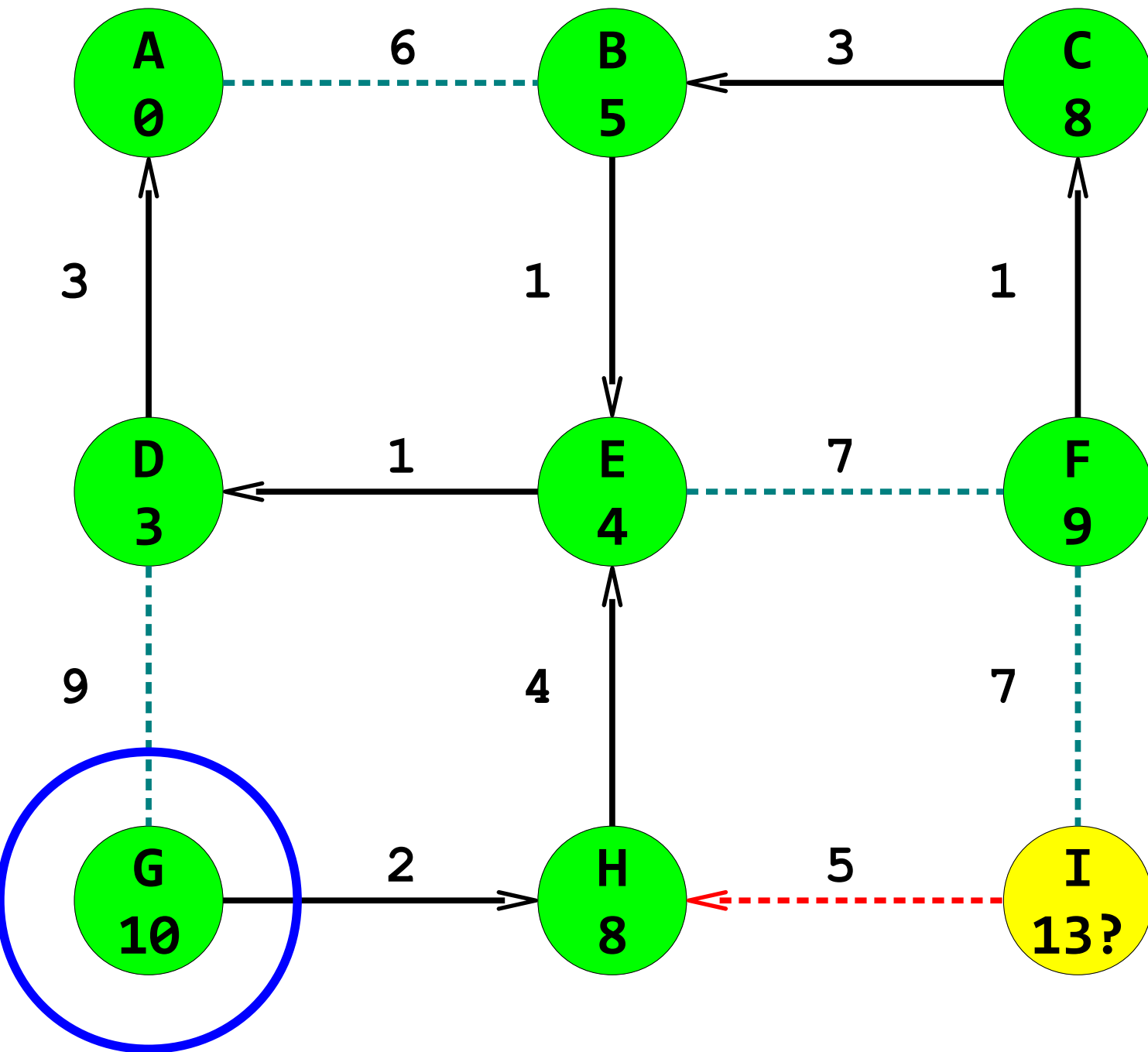
**G**  
**10?**

**I**  
**13?**



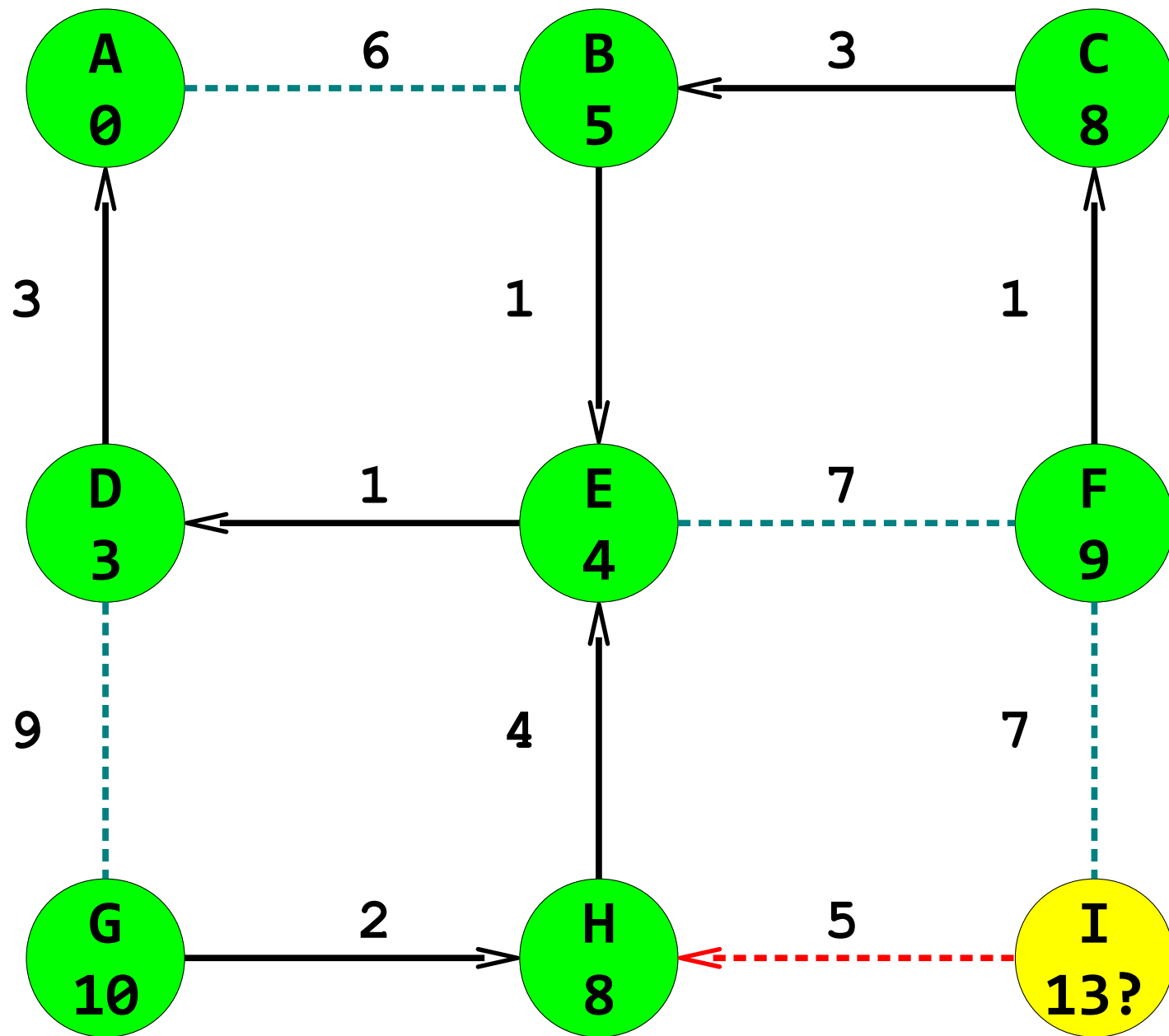
---

**I**  
**13?**



---

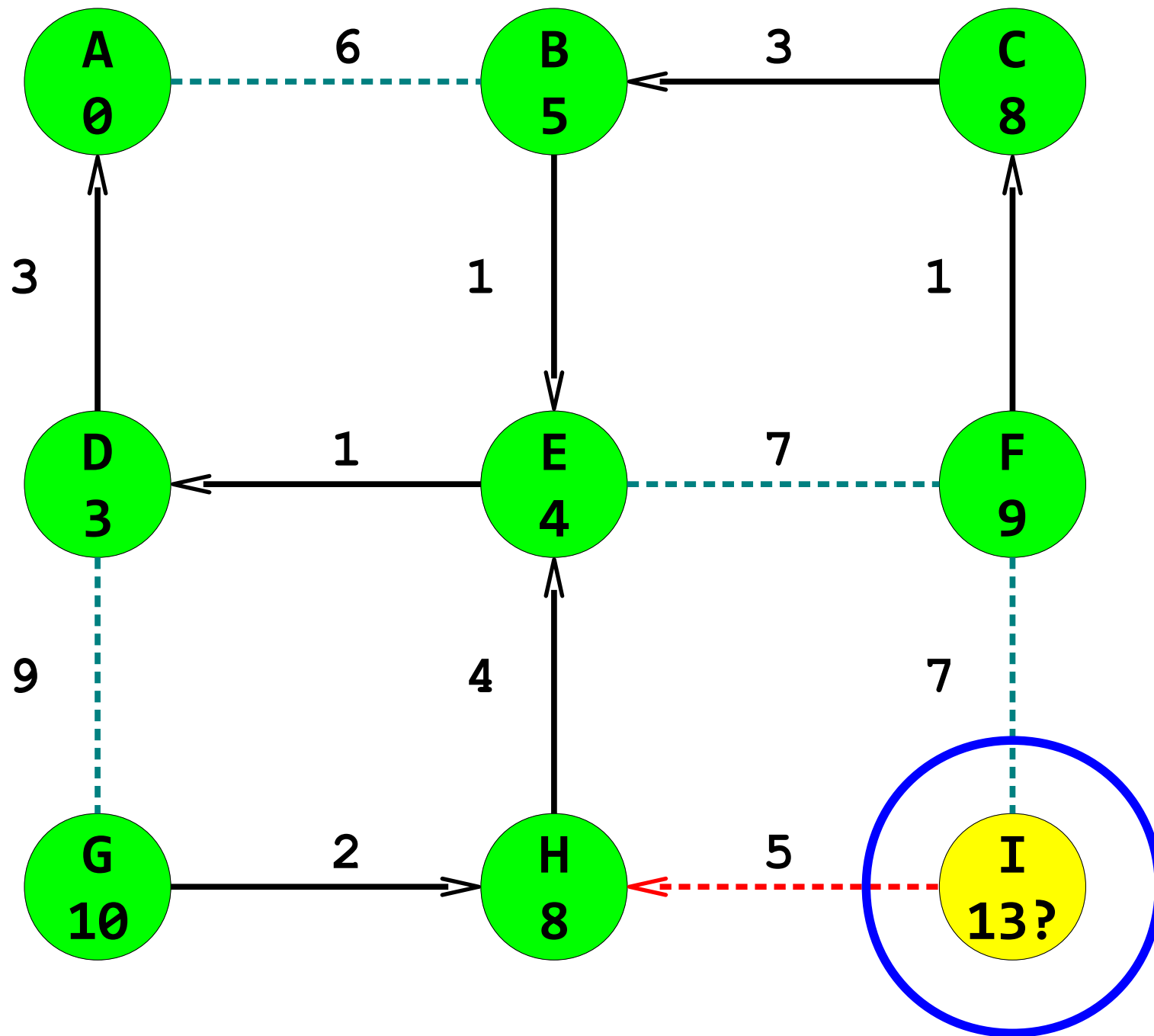
**I**  
**13?**



---

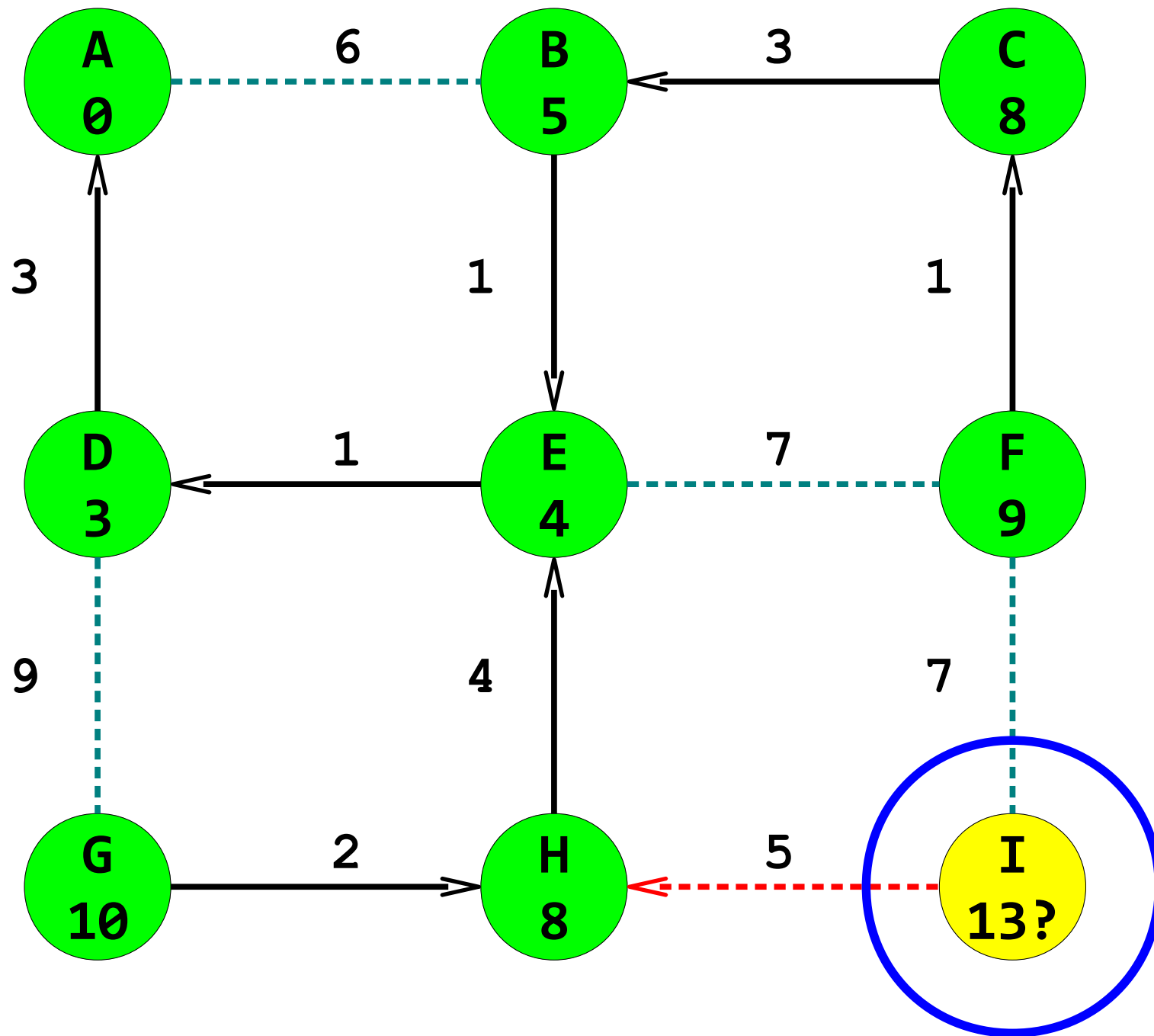
**I**  
**13?**

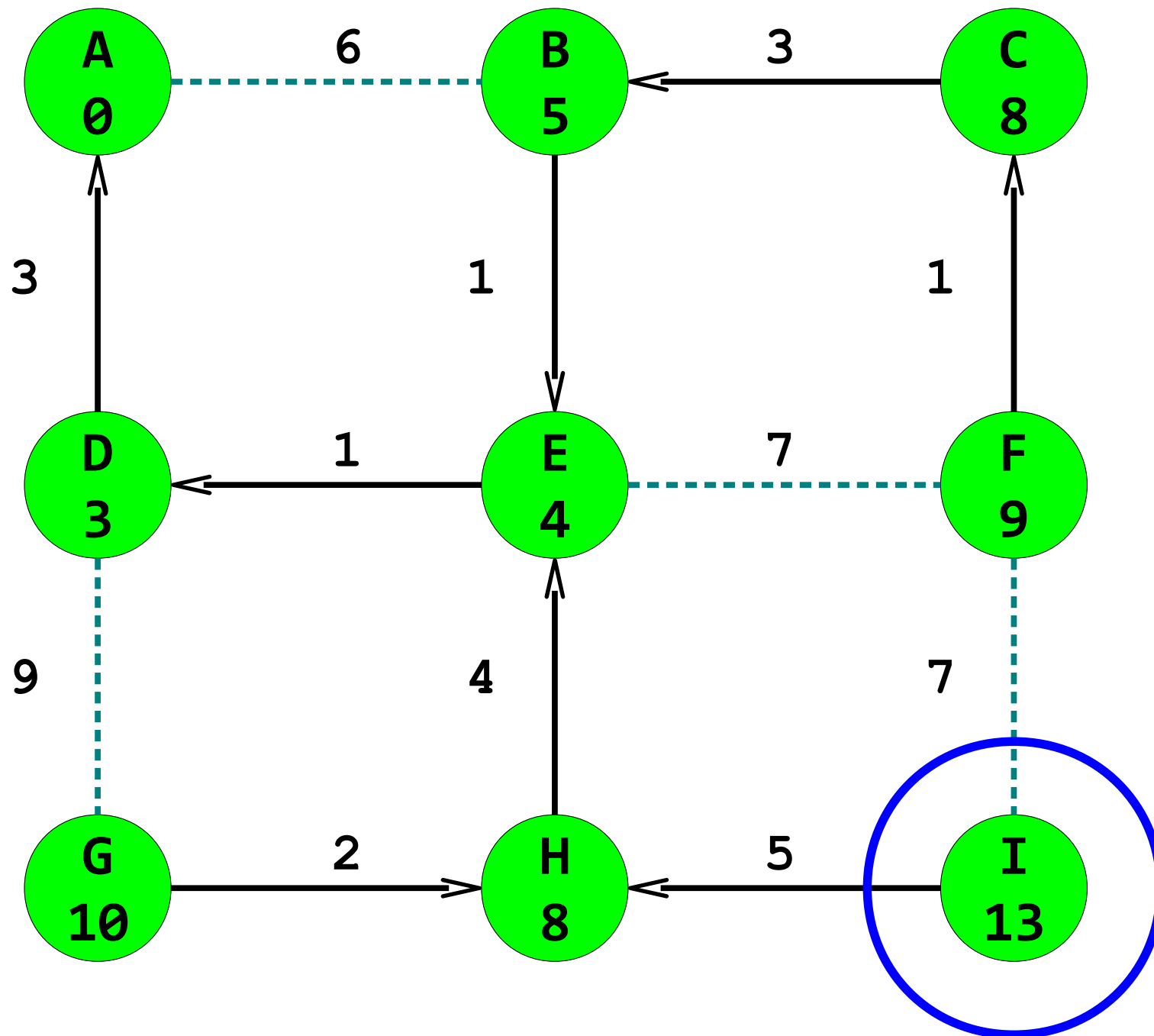


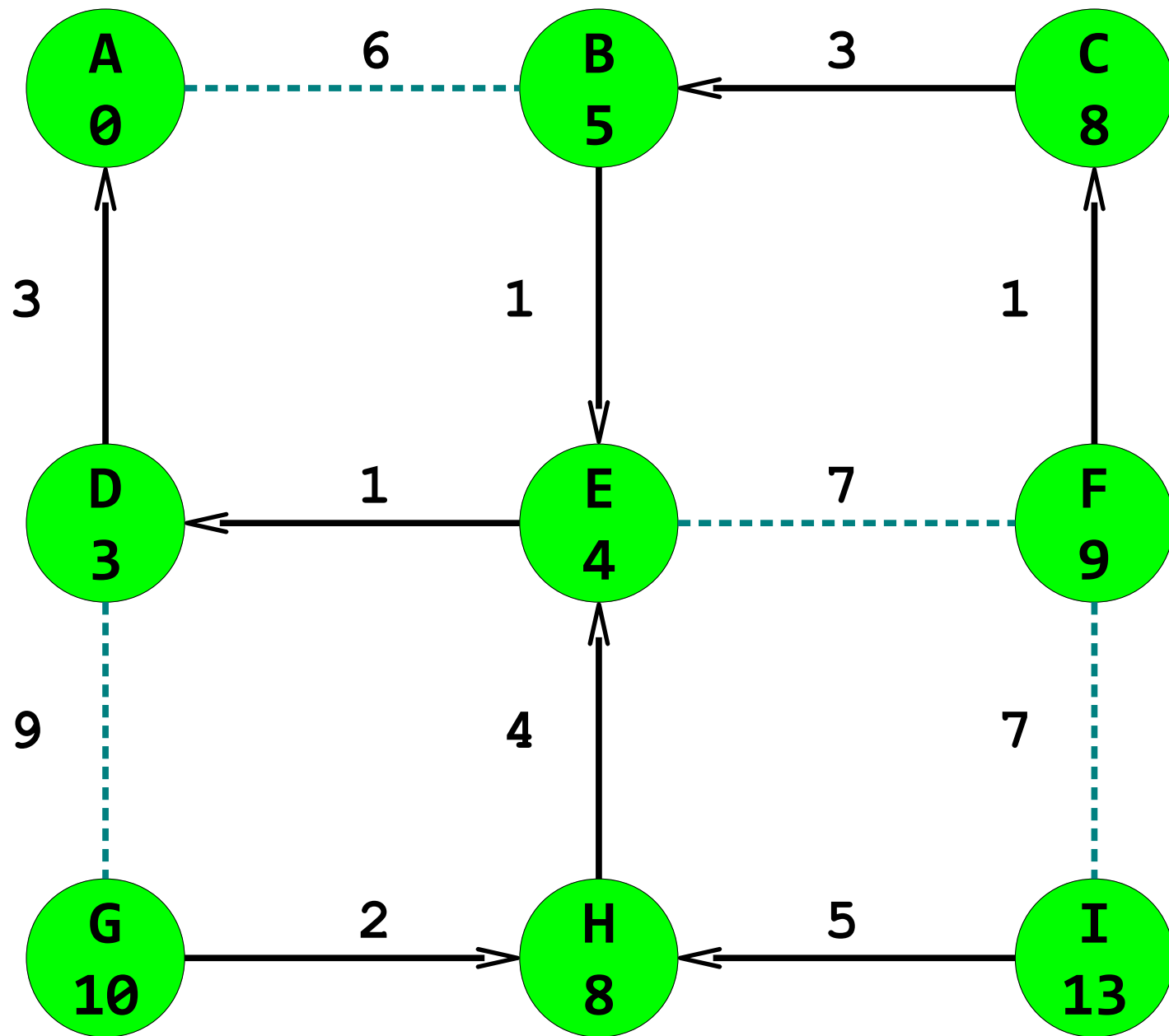


---

**I**  
**13?**







# Dijkstra's Algorithm

- Split nodes apart into three groups:
  - Green nodes, where we already have the shortest path;
  - Gray nodes, which we have never seen; and
  - Yellow nodes that we still need to process.
- Dijkstra's algorithm works as follows:
  - Mark all nodes gray except the start node, which is yellow and has cost 0.
  - Until no yellow nodes remain:
    - Choose the yellow node with the lowest total cost.
    - Mark that node green.
    - Mark all its gray neighbors yellow and with the appropriate cost.
    - Update the costs of all adjacent yellow nodes by considering the path through the current node.

# An Important Note

- The version of Dijkstra's algorithm I have just described is ***not*** the same as the version described in the course reader.
- This version is more complex than the book's version, but is much faster.

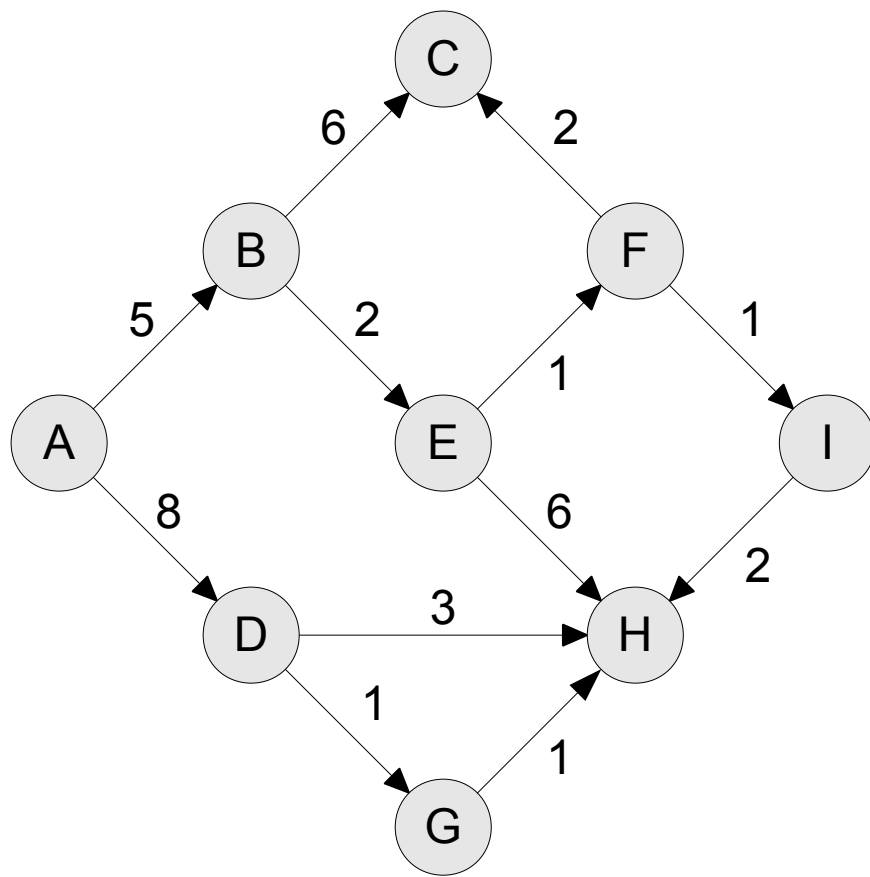
# Differences from BFS

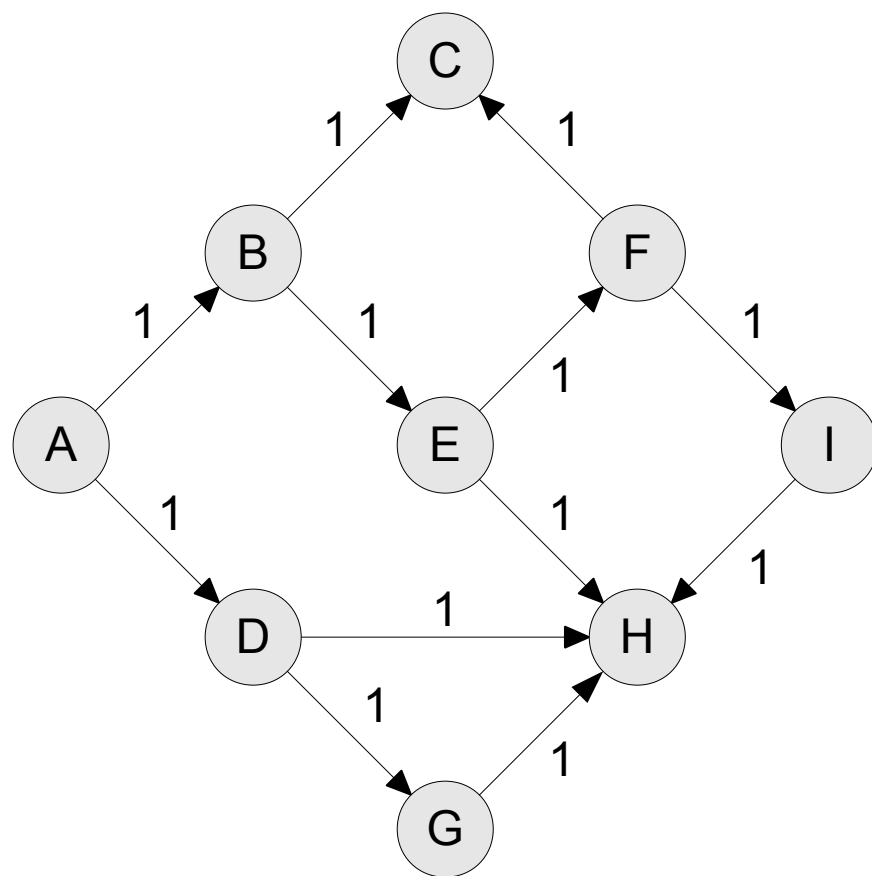
- BFS uses a queue, while Dijkstra's algorithm uses a **priority queue**, where priorities are potential distances to nodes.
  - Need a special operation called **decrease-key**, which lowers the priority of an enqueued element.
- BFS never changes a parent pointer once it is set, while Dijkstra's algorithm might change parent pointers.
  - If a possible path to a node is found to be incorrect, then its parent might change.

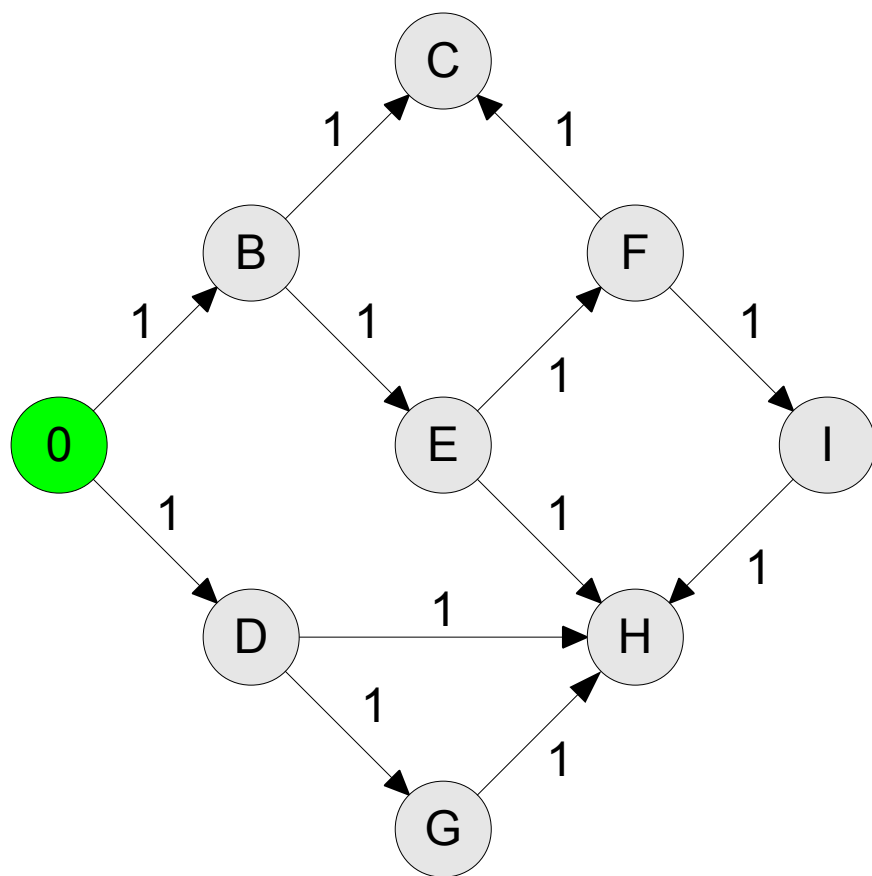
# Dijkstra's Algorithm

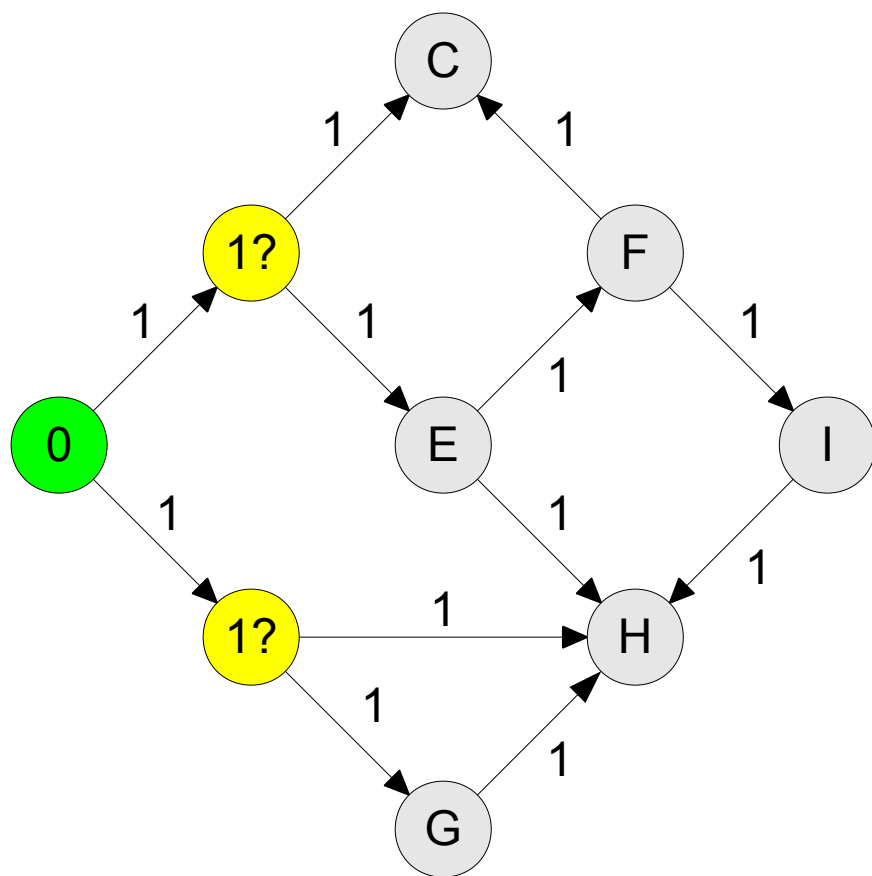
- What happens if we run Dijkstra's Algorithm on a graph where every arc has weight 1?

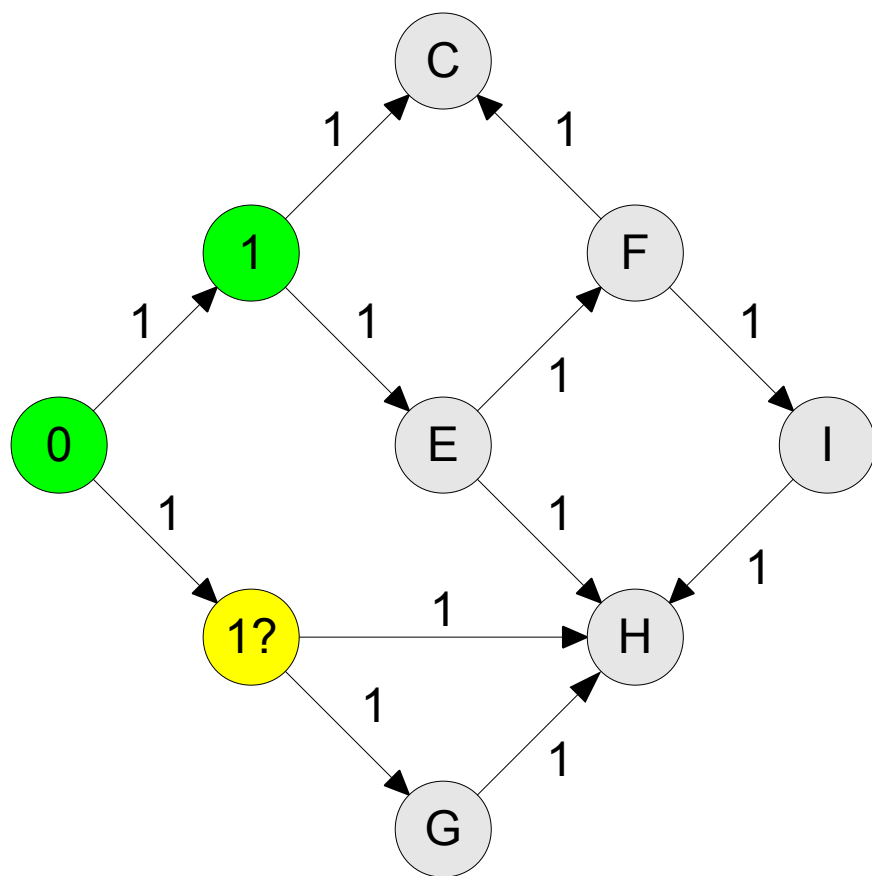


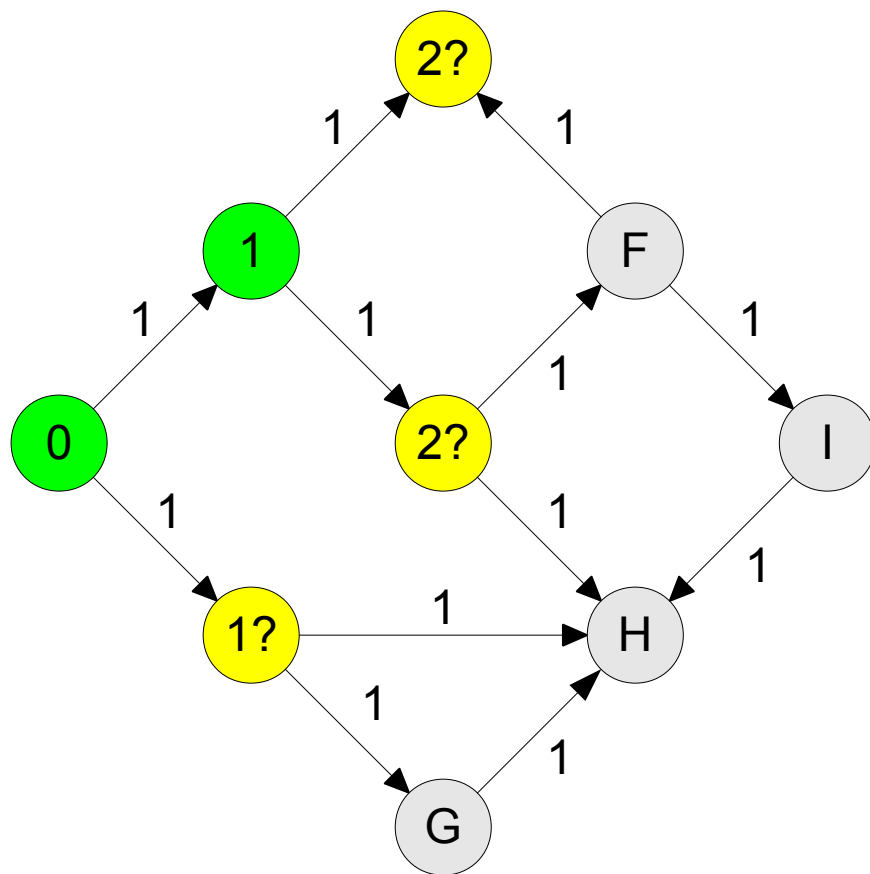


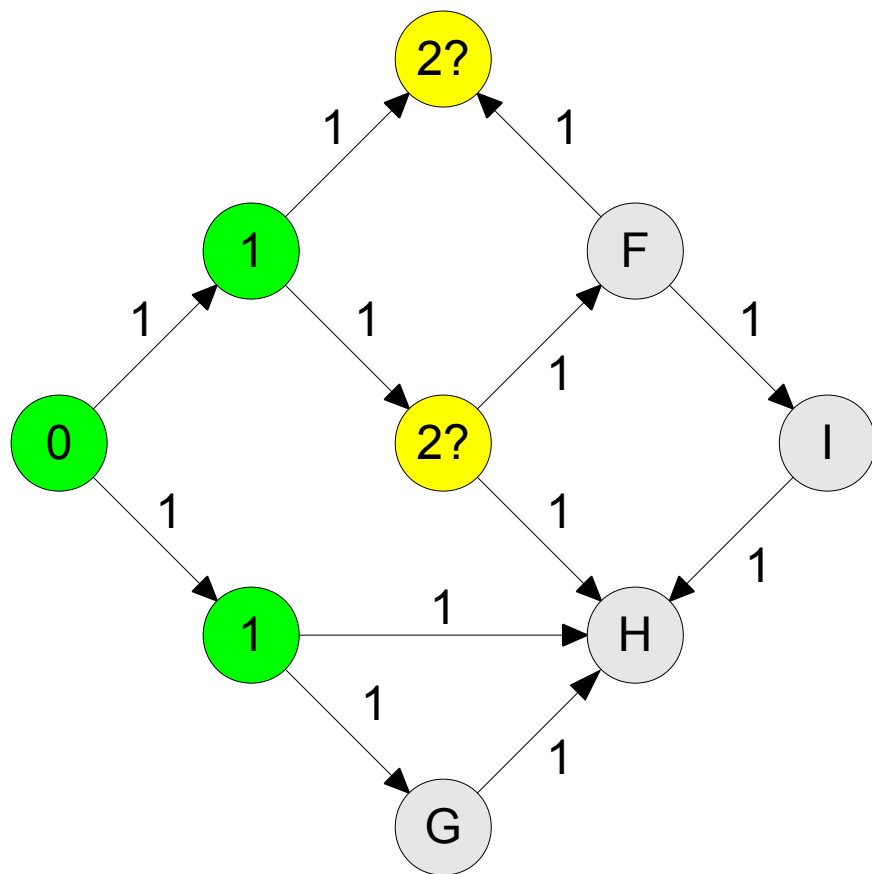


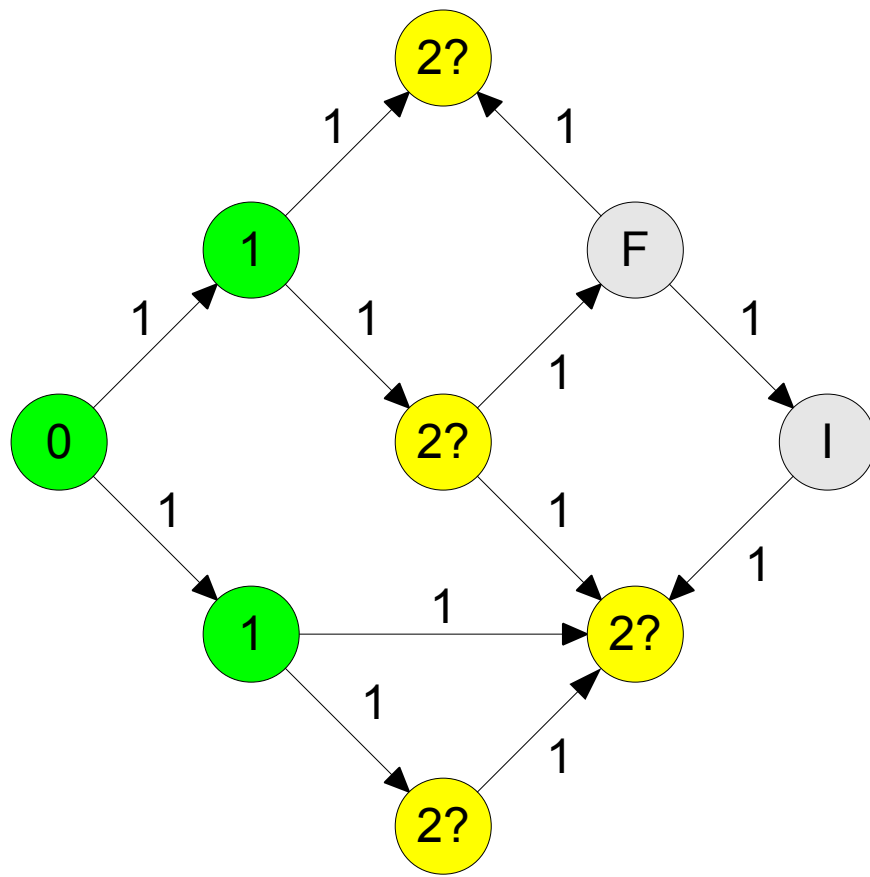




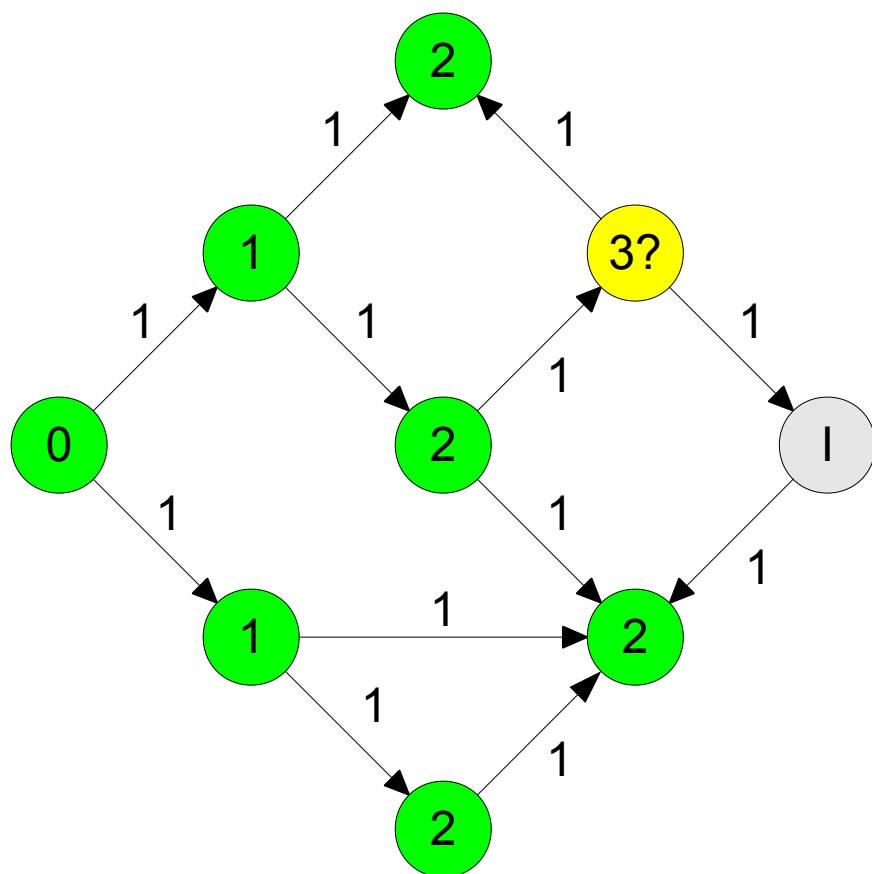


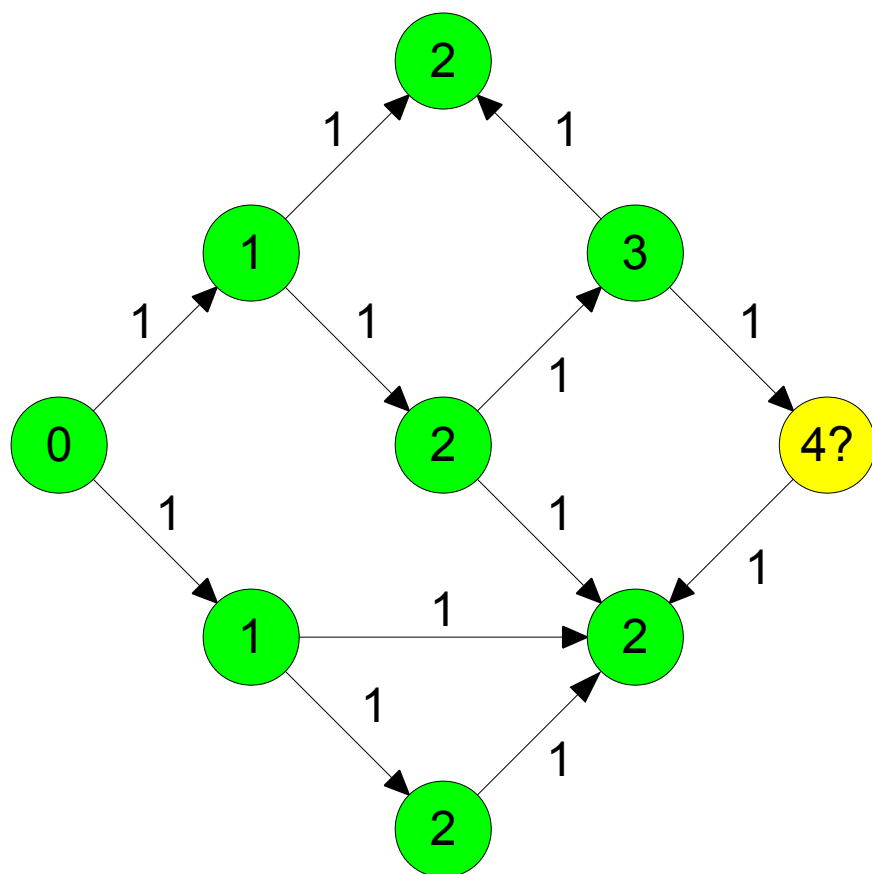


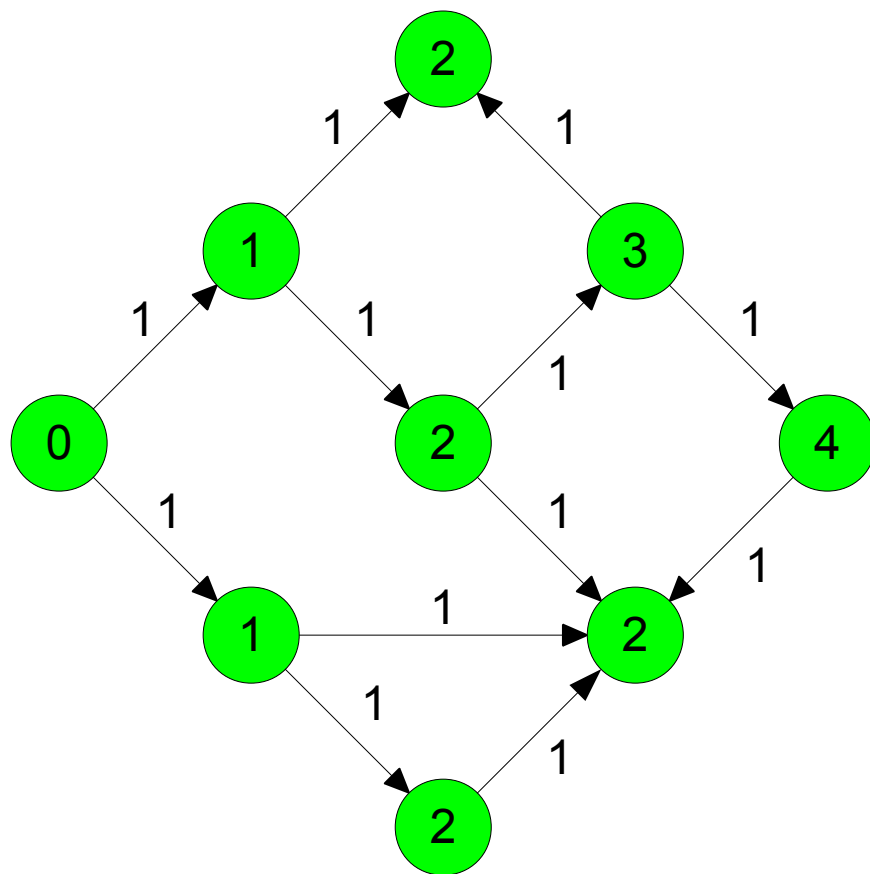












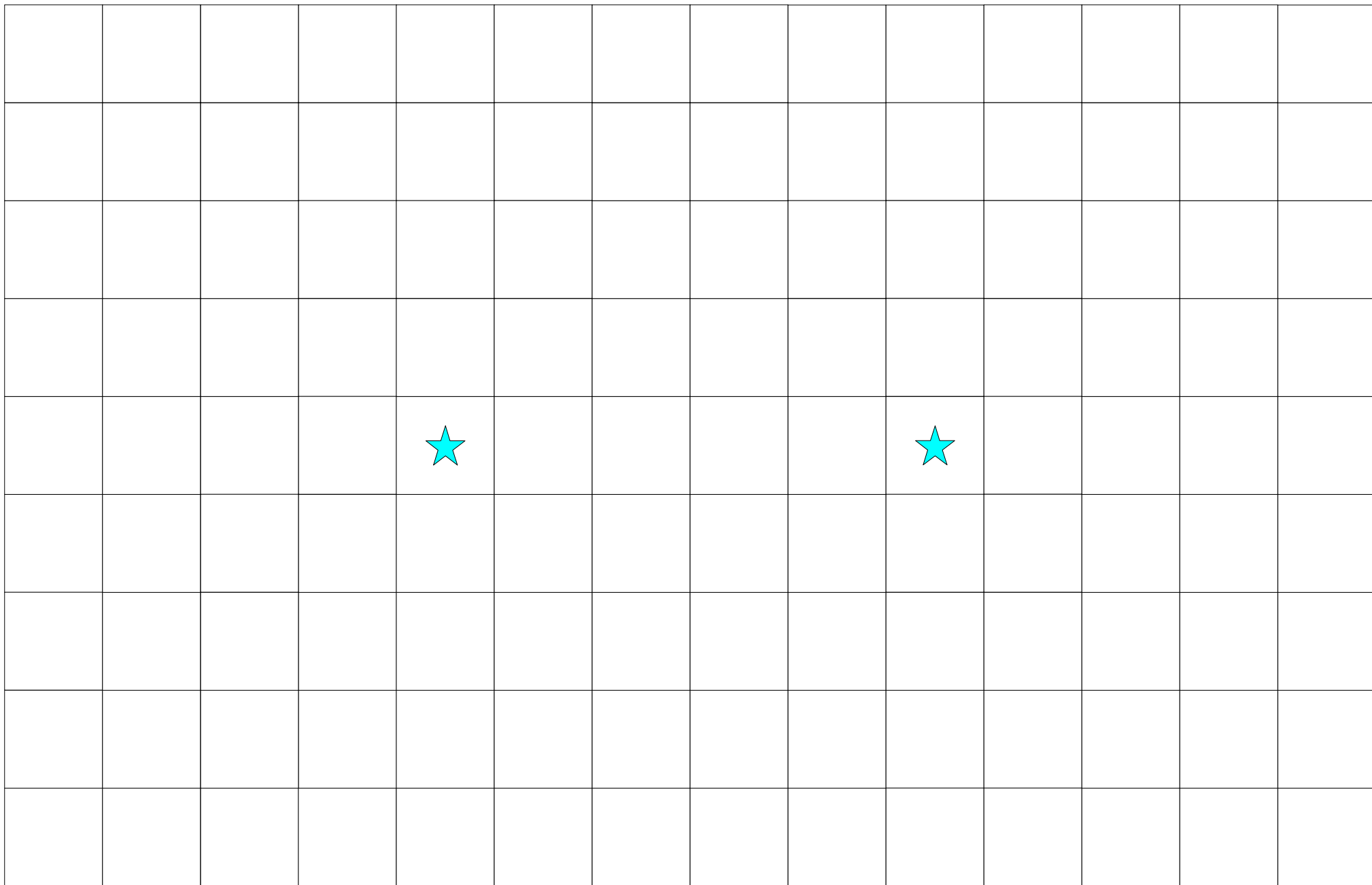
# Dijkstra's Algorithm

- What happens if we run Dijkstra's Algorithm on a graph where every arc has weight 1?
  - It devolves into Breadth First Search

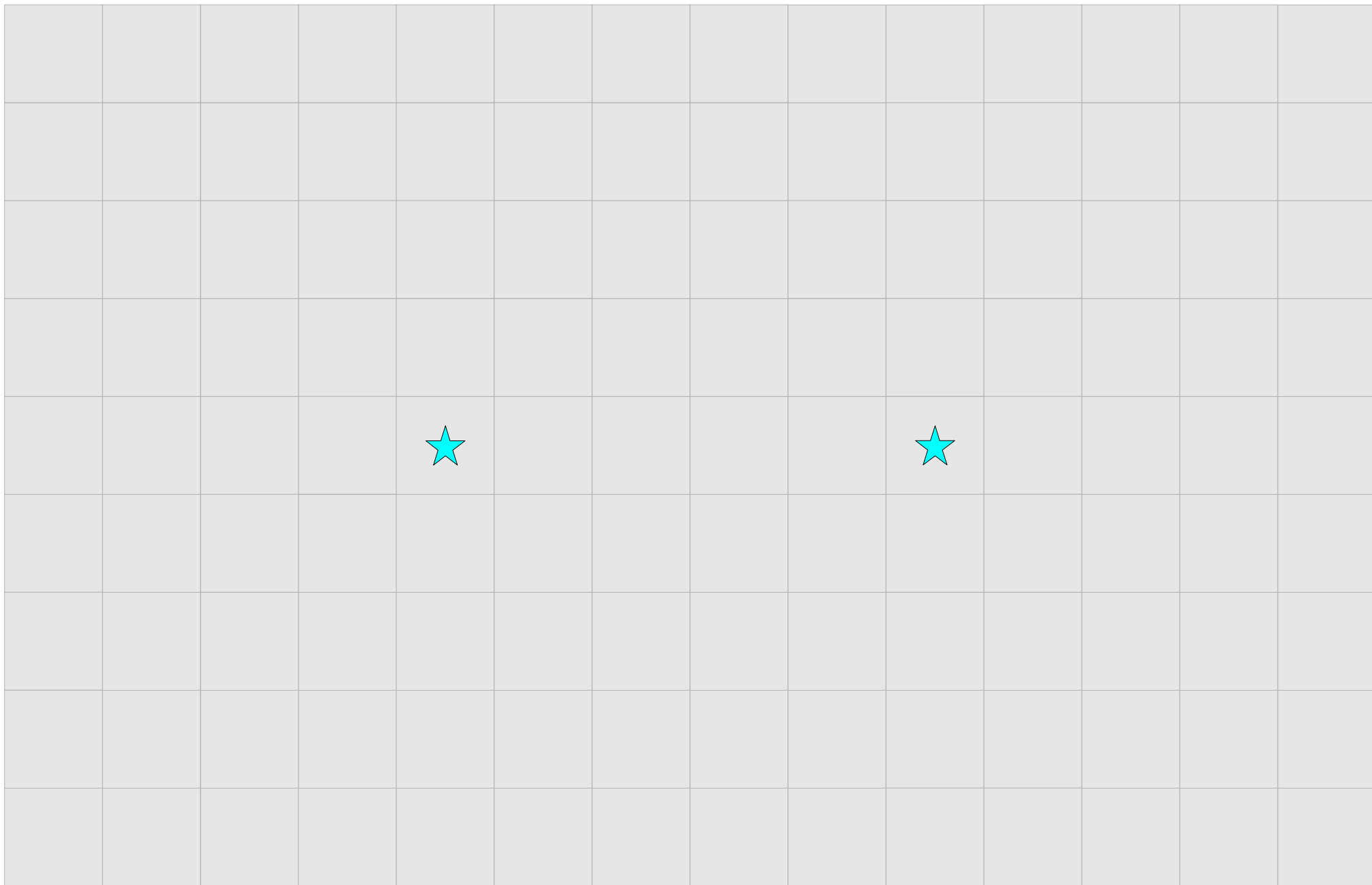
The next slide will be really useful as a reference, so I changed the color scheme so that you can find it more easily.

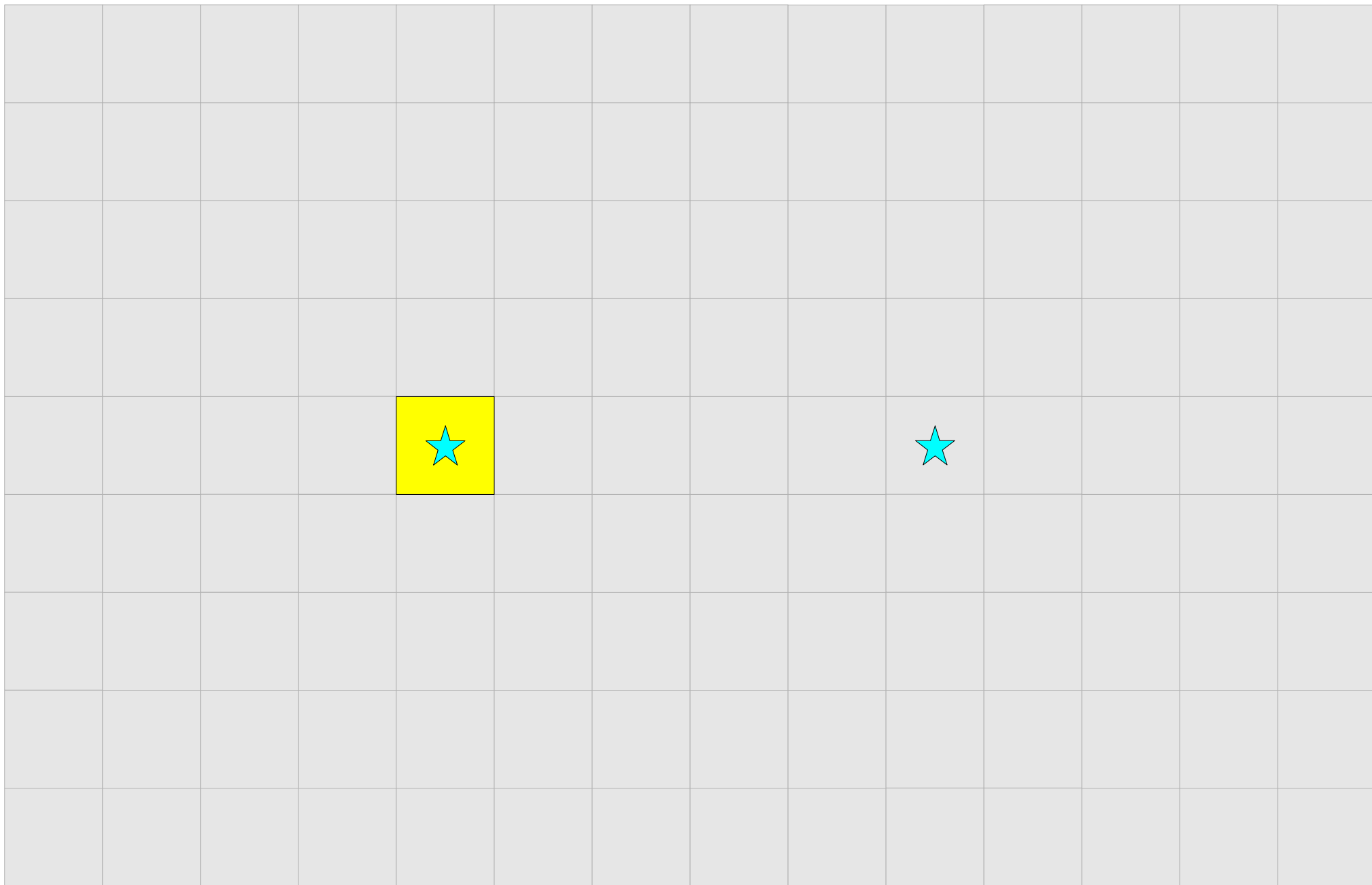
- Mark all nodes as gray.
- Mark the initial node **s** as yellow and at candidate distance **0**.
- Enqueue **s** into the priority queue.
- While not all nodes have been visited:
  - Dequeue the lowest-cost node **u** from the priority queue.
  - Color **u** green. The candidate distance **d** that is currently stored for node **u** is the length of the shortest path from **s** to **u**.
  - If **u** is the destination node **t**, you have found the shortest path from **s** to **t** and are done.
  - For each node **v** connected to **u** by an edge of length **L**:
    - If **v** is gray:
      - Color **v** yellow.
      - Mark **v**'s distance as **d + L**.
      - Set **v**'s parent to be **u**.
      - Enqueue **v** into the priority queue.
    - If **v** is yellow and the candidate distance to **v** is greater than **d + L**:
      - Update **v**'s candidate distance to be **d + L**; the older candidate distance is incorrect.
      - Update **v**'s parent to be **u**.
      - Update **v**'s priority in the priority queue to **d + L**.

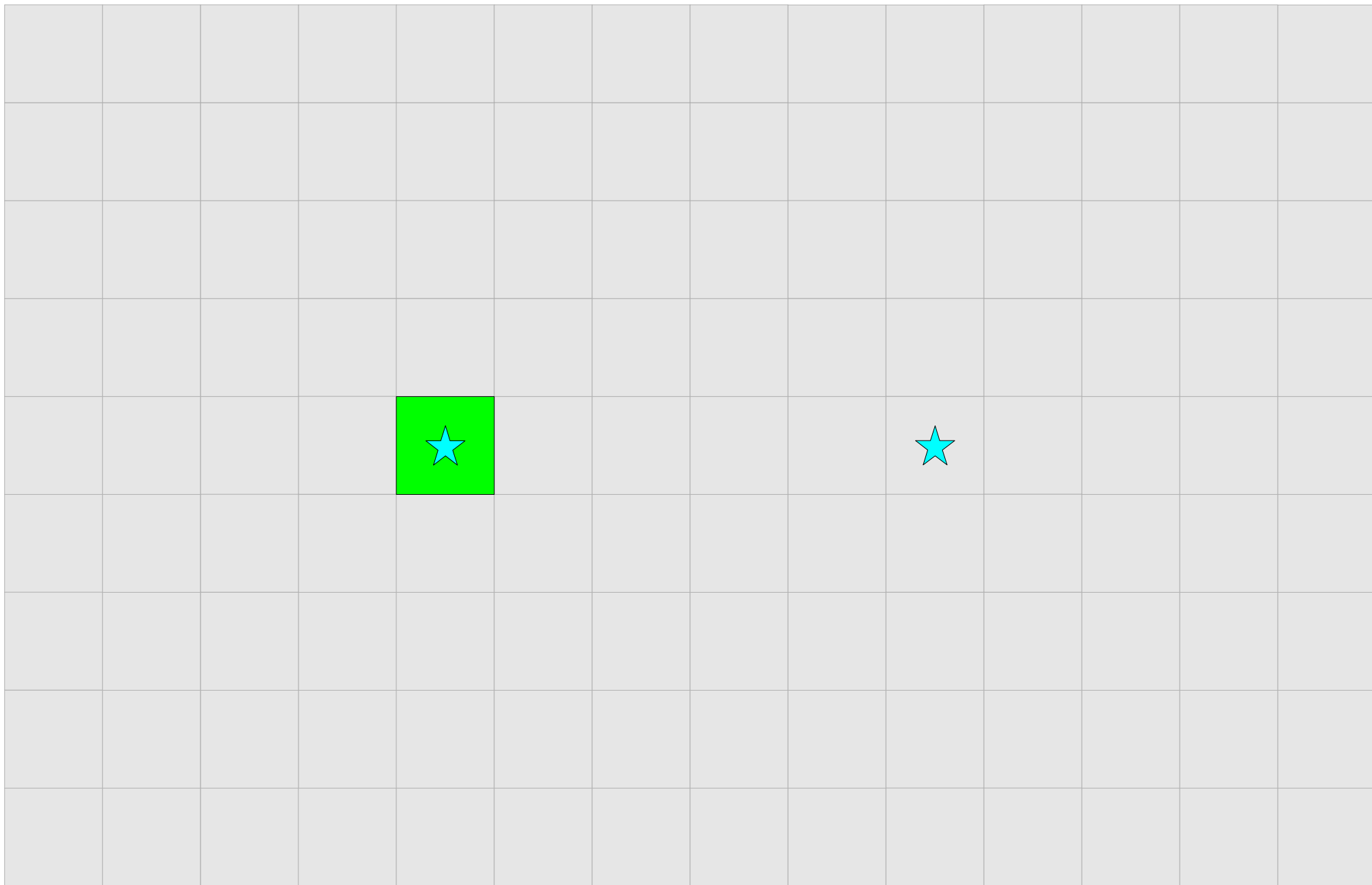
One Detail with Dijkstra's Algorithm

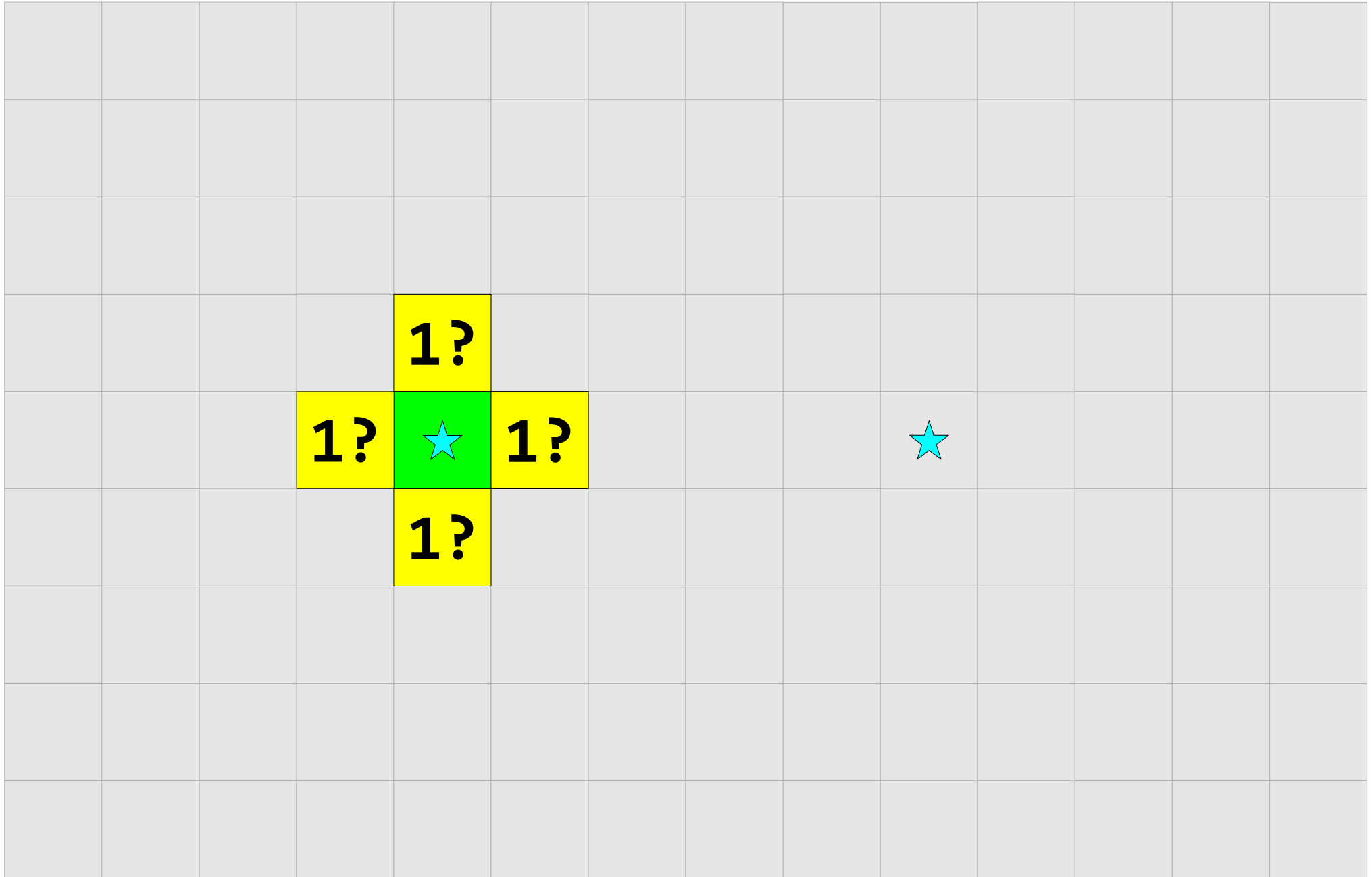




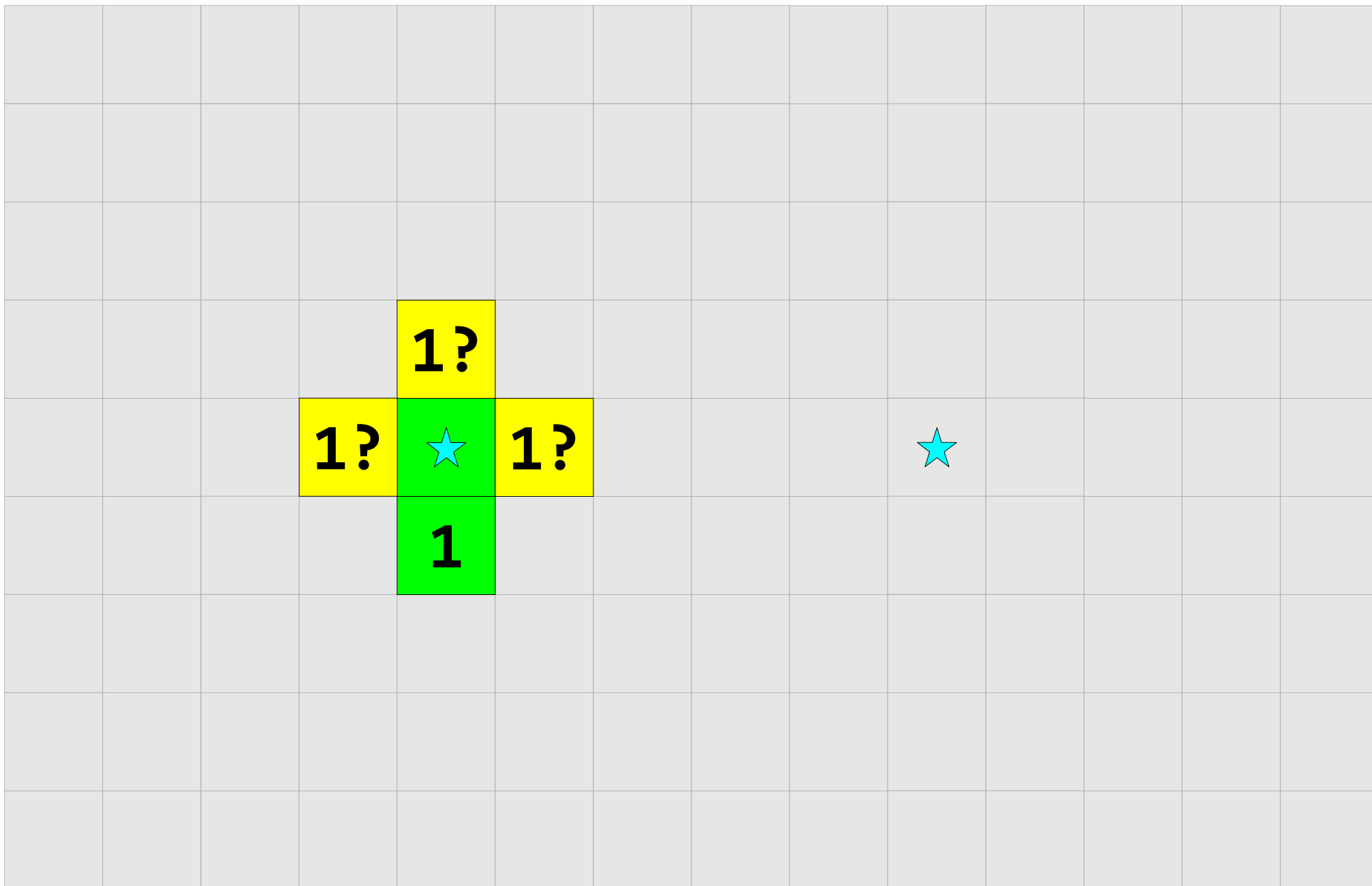


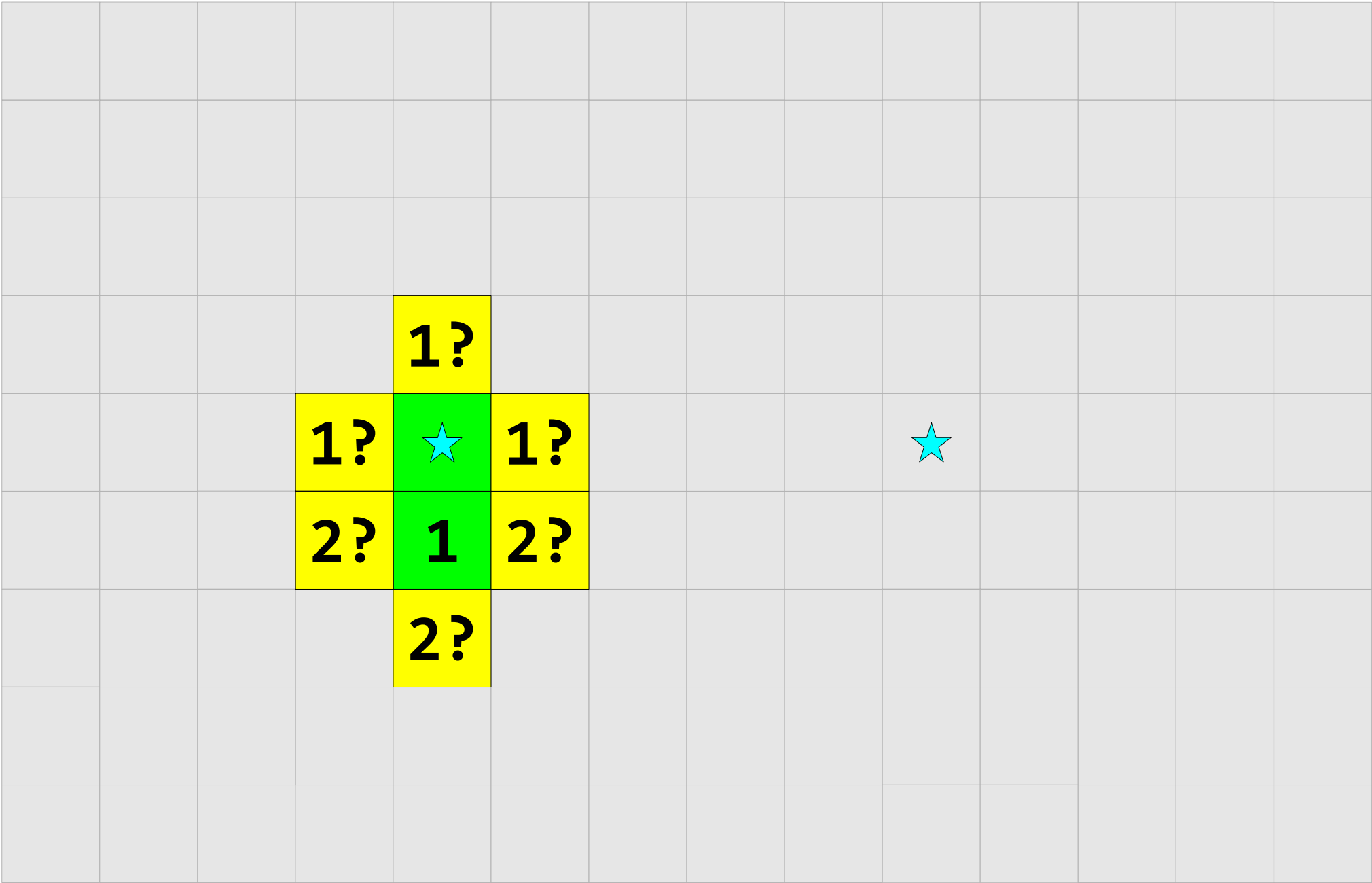


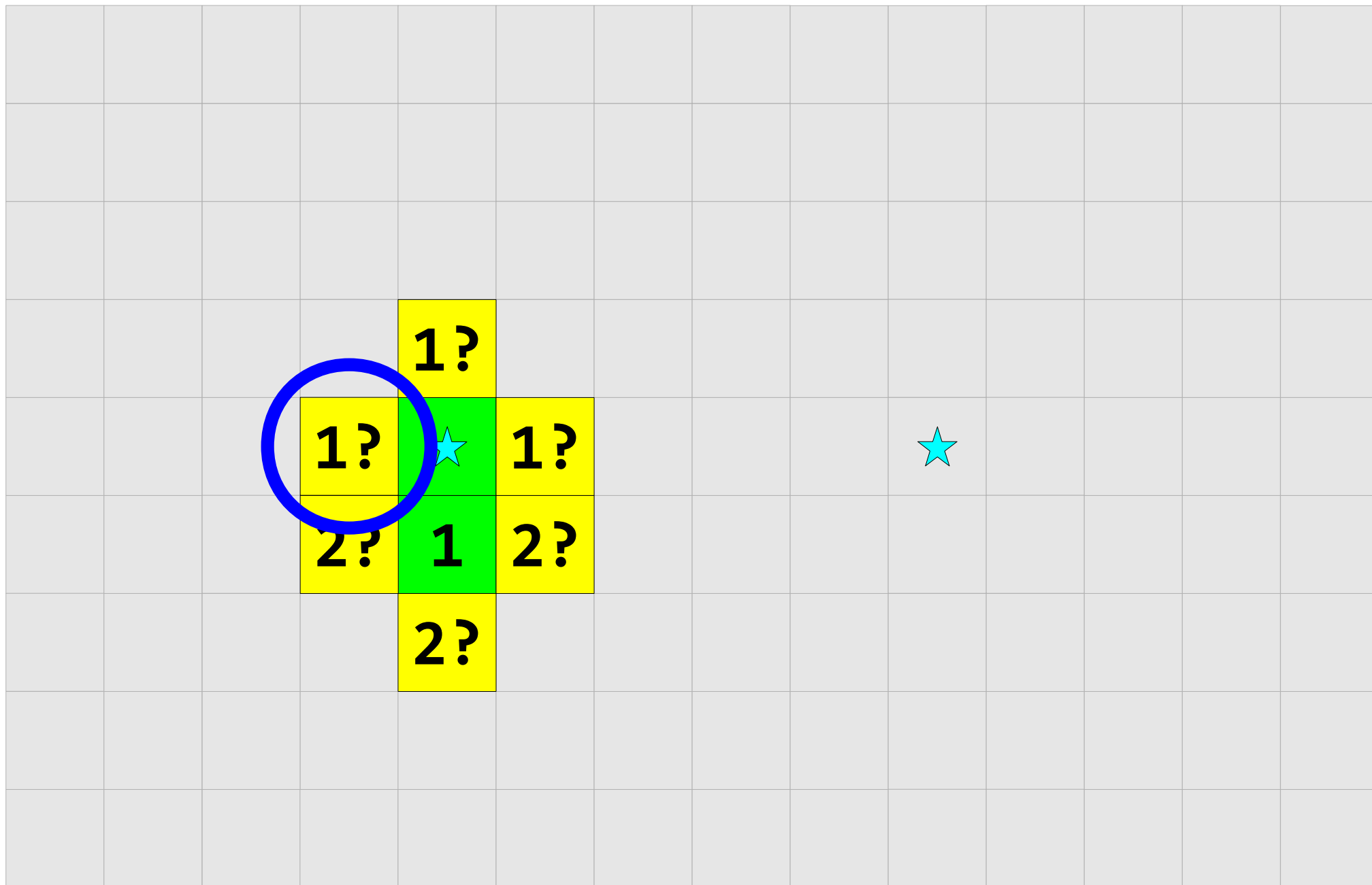




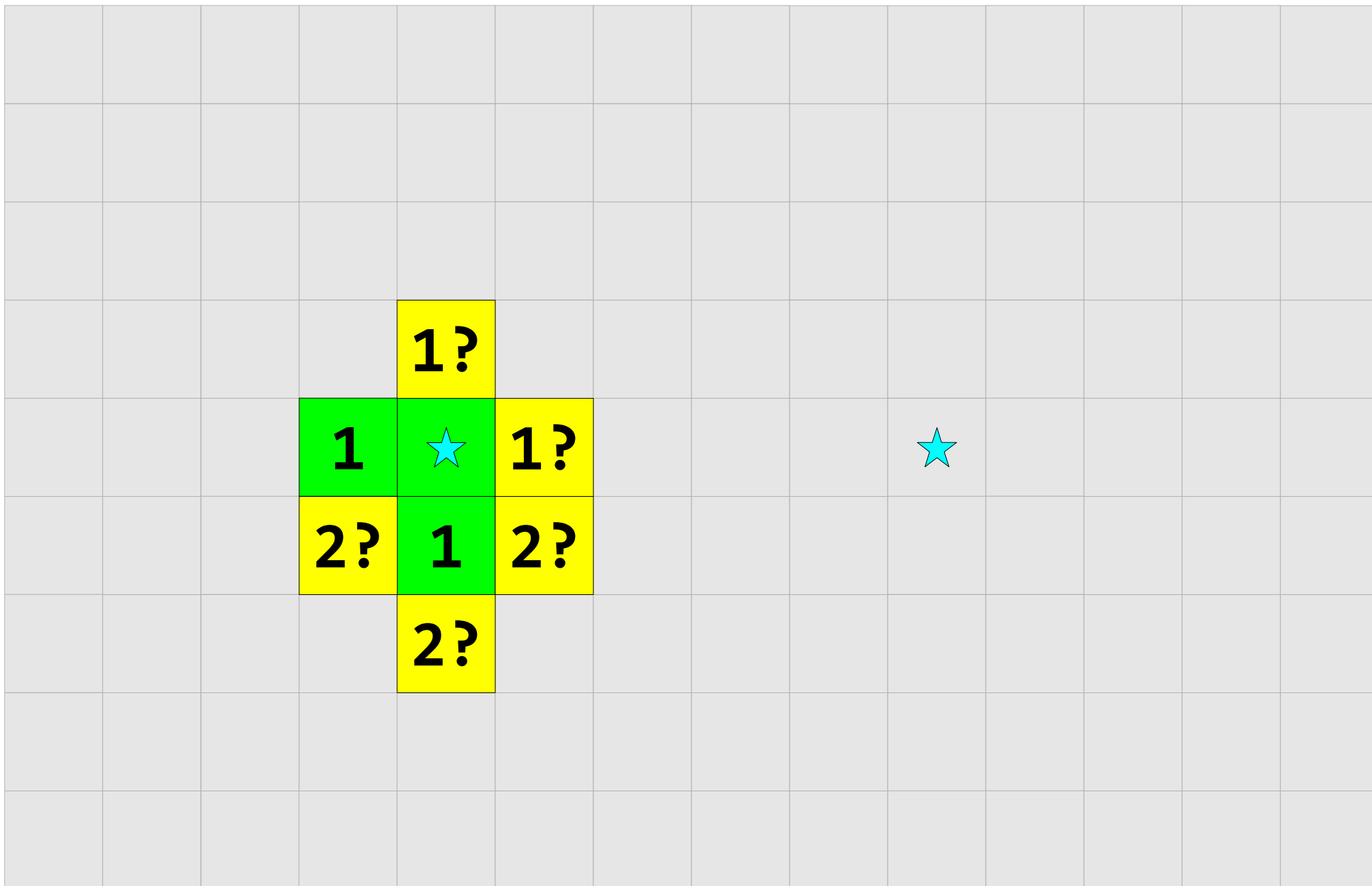


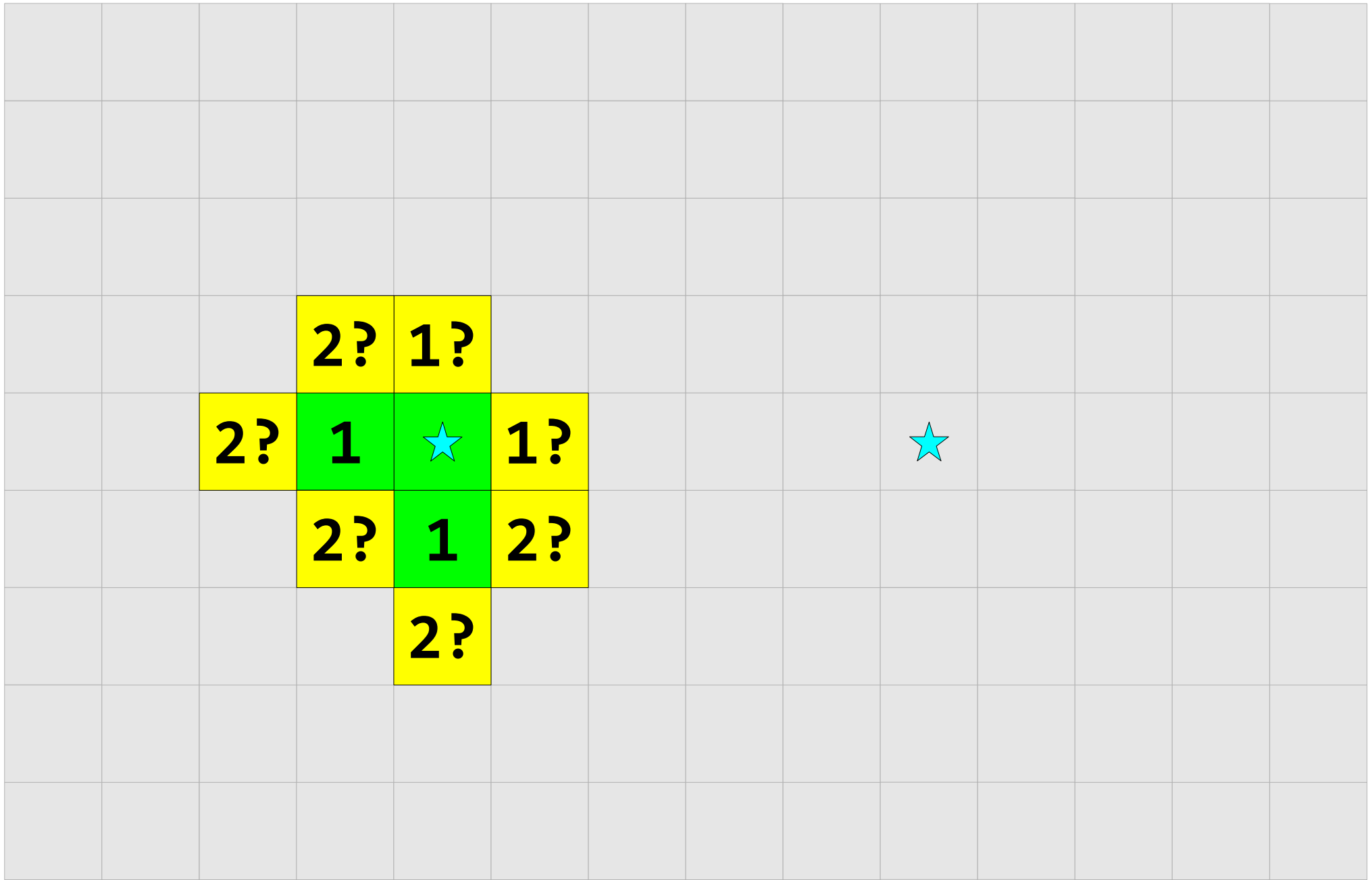


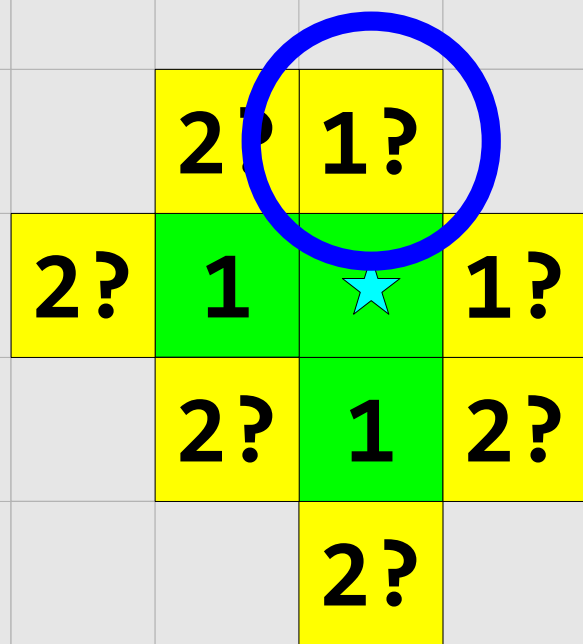


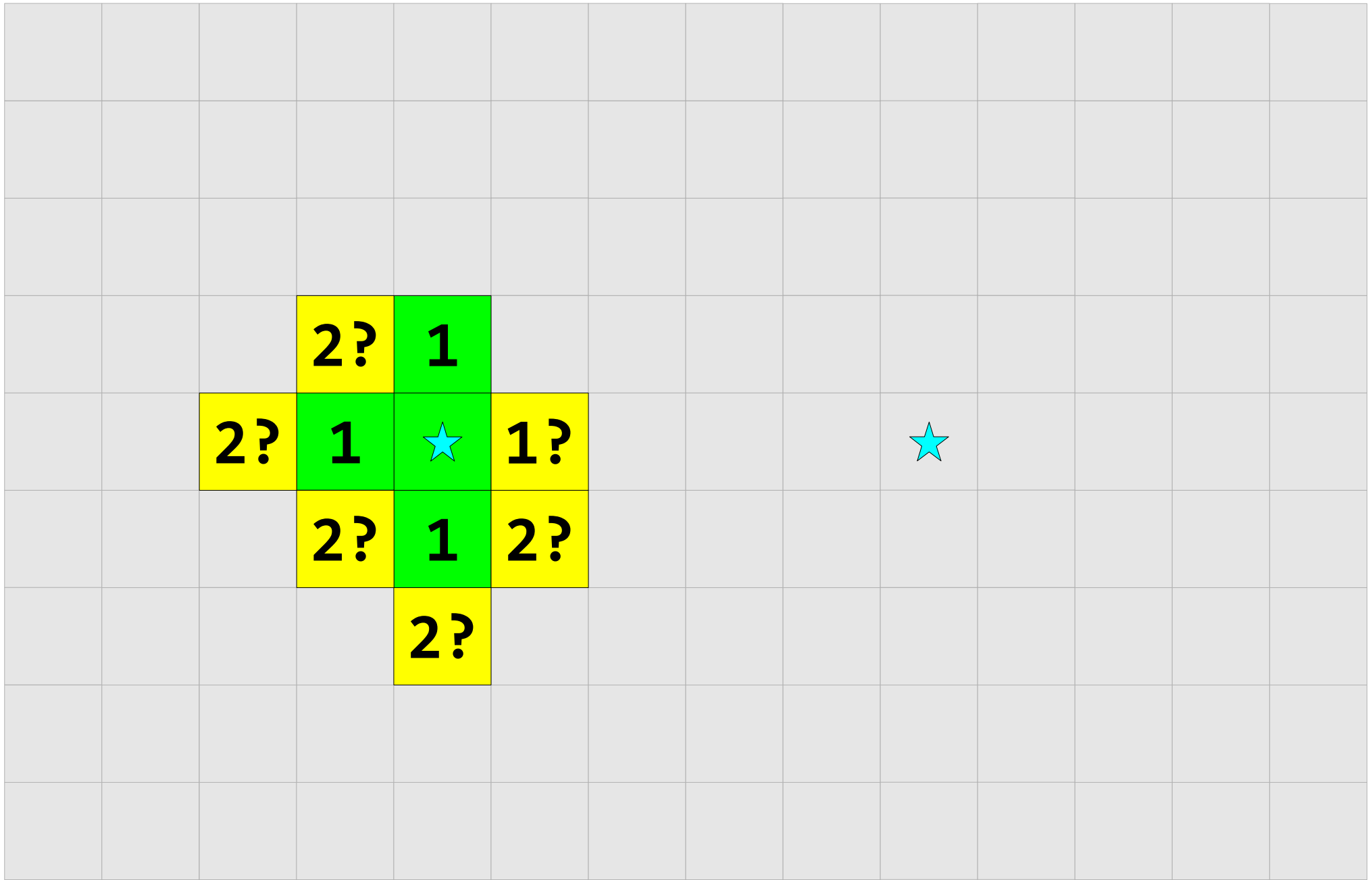


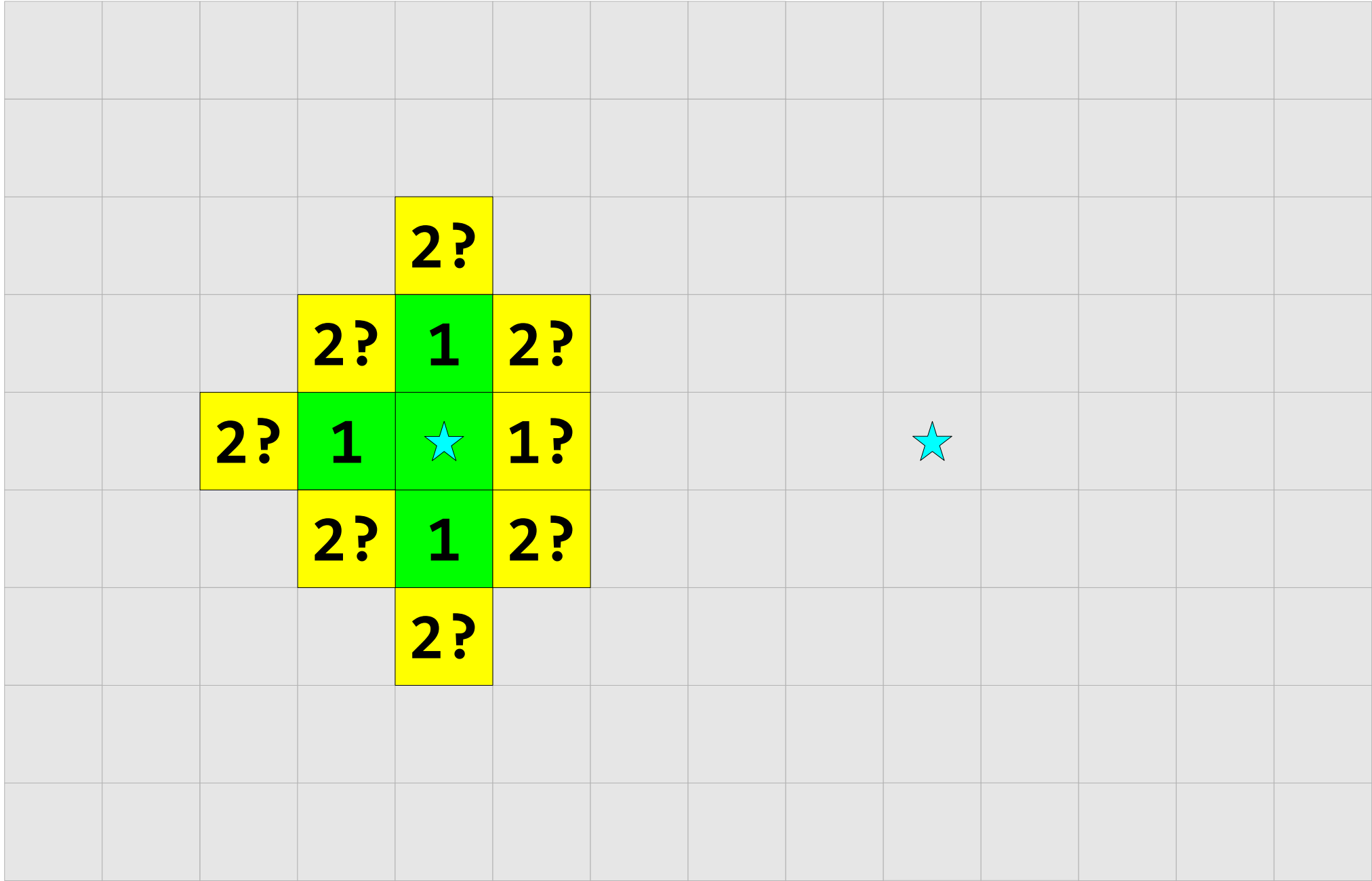


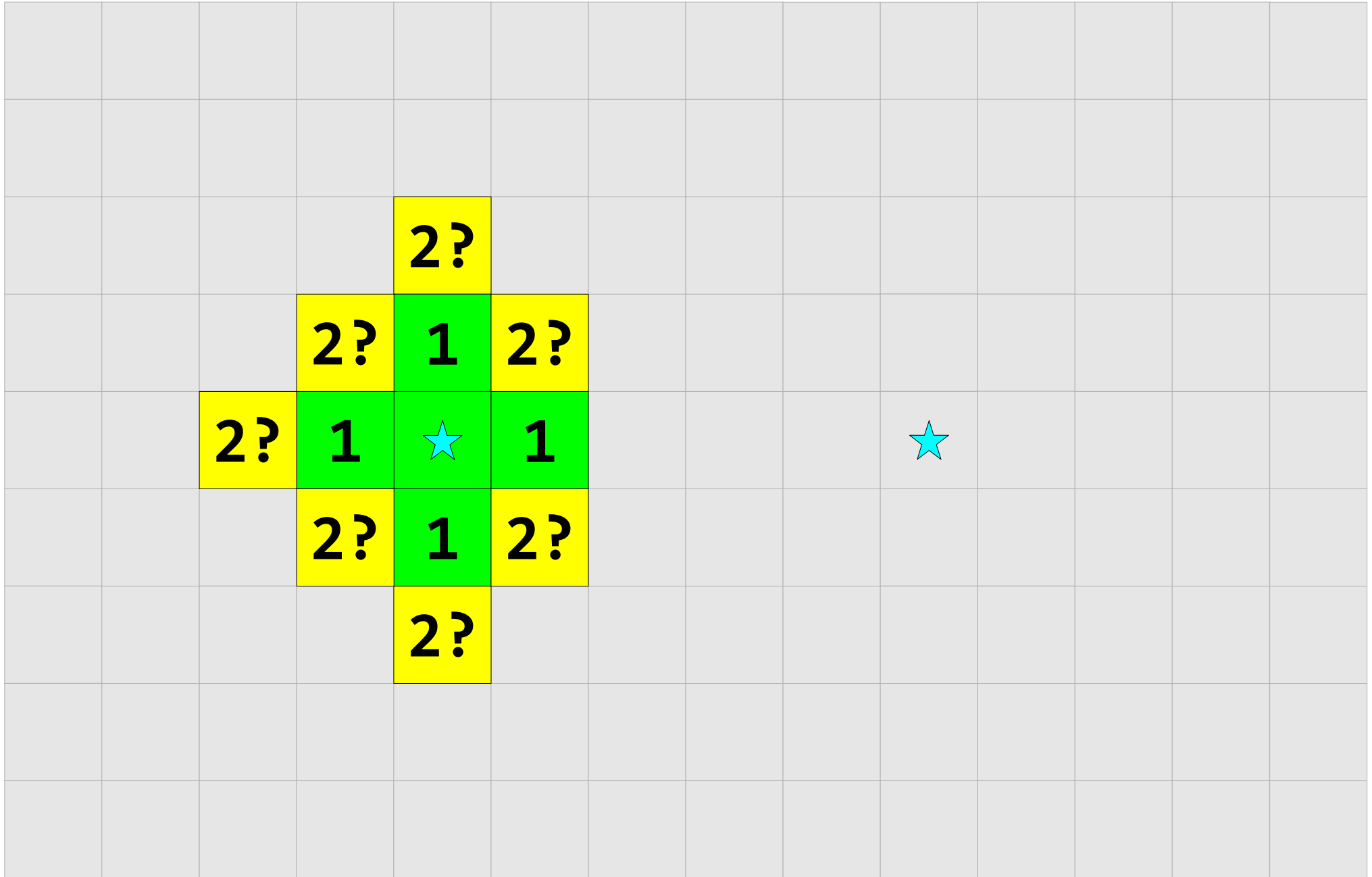




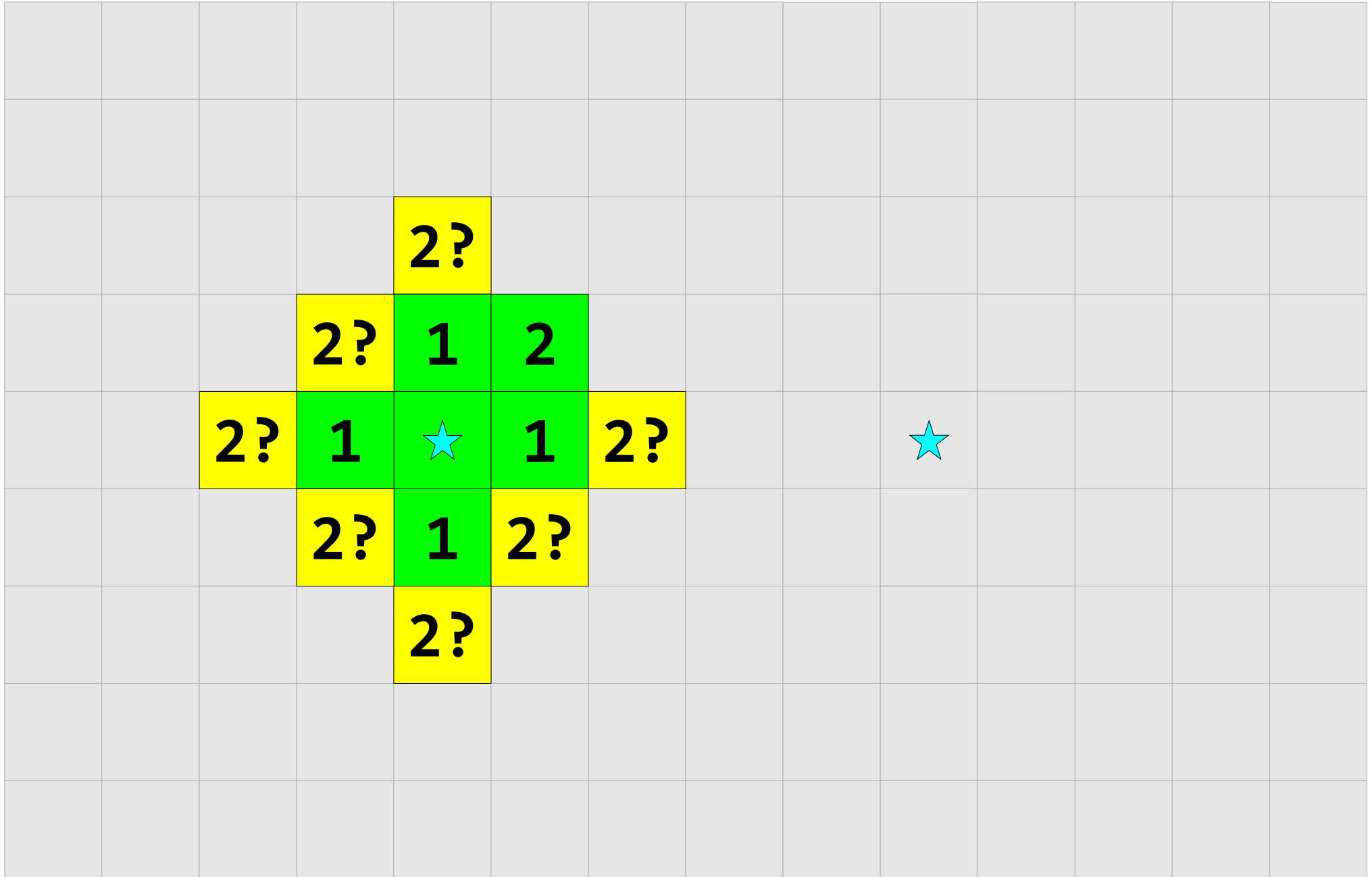




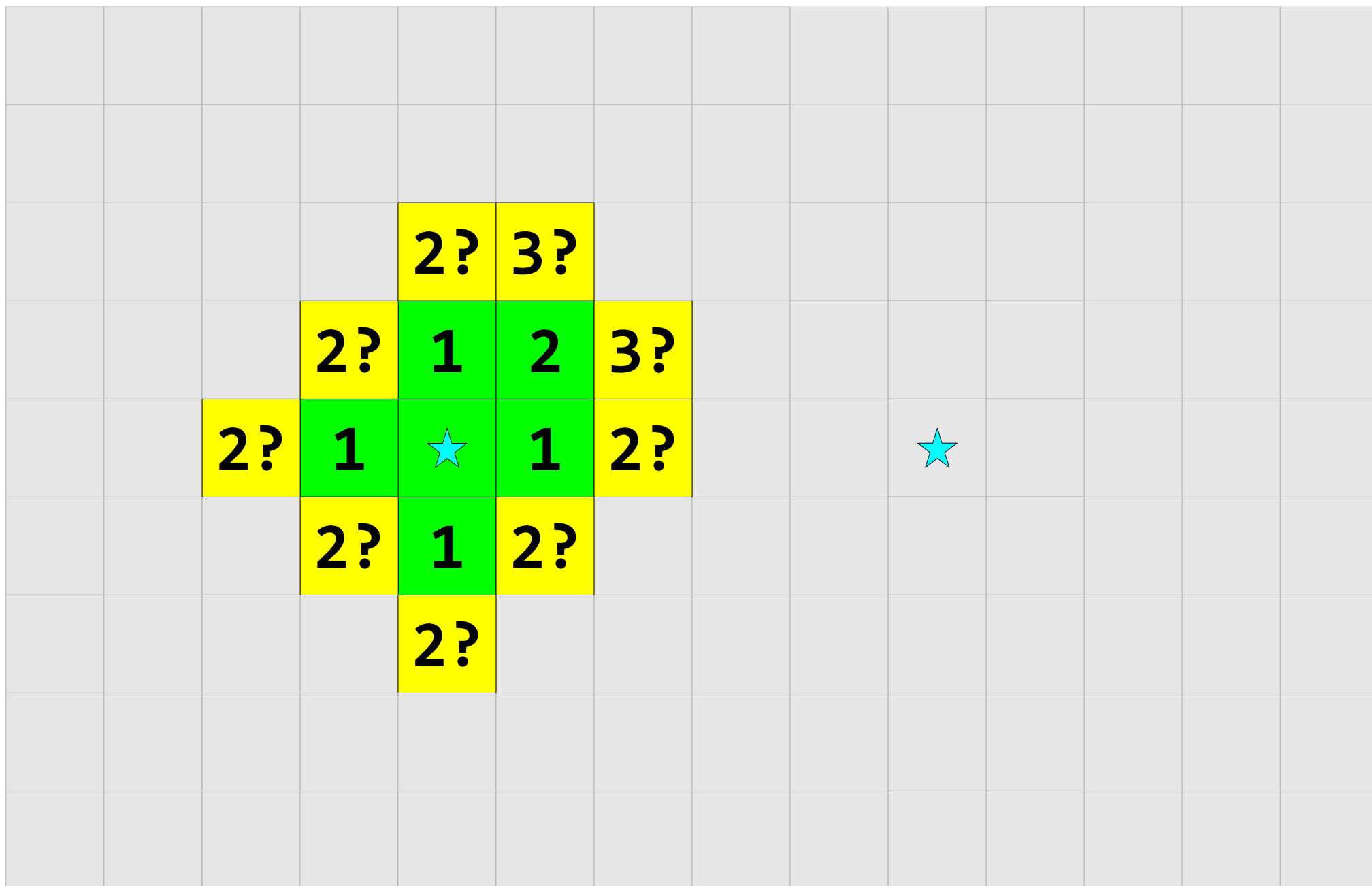


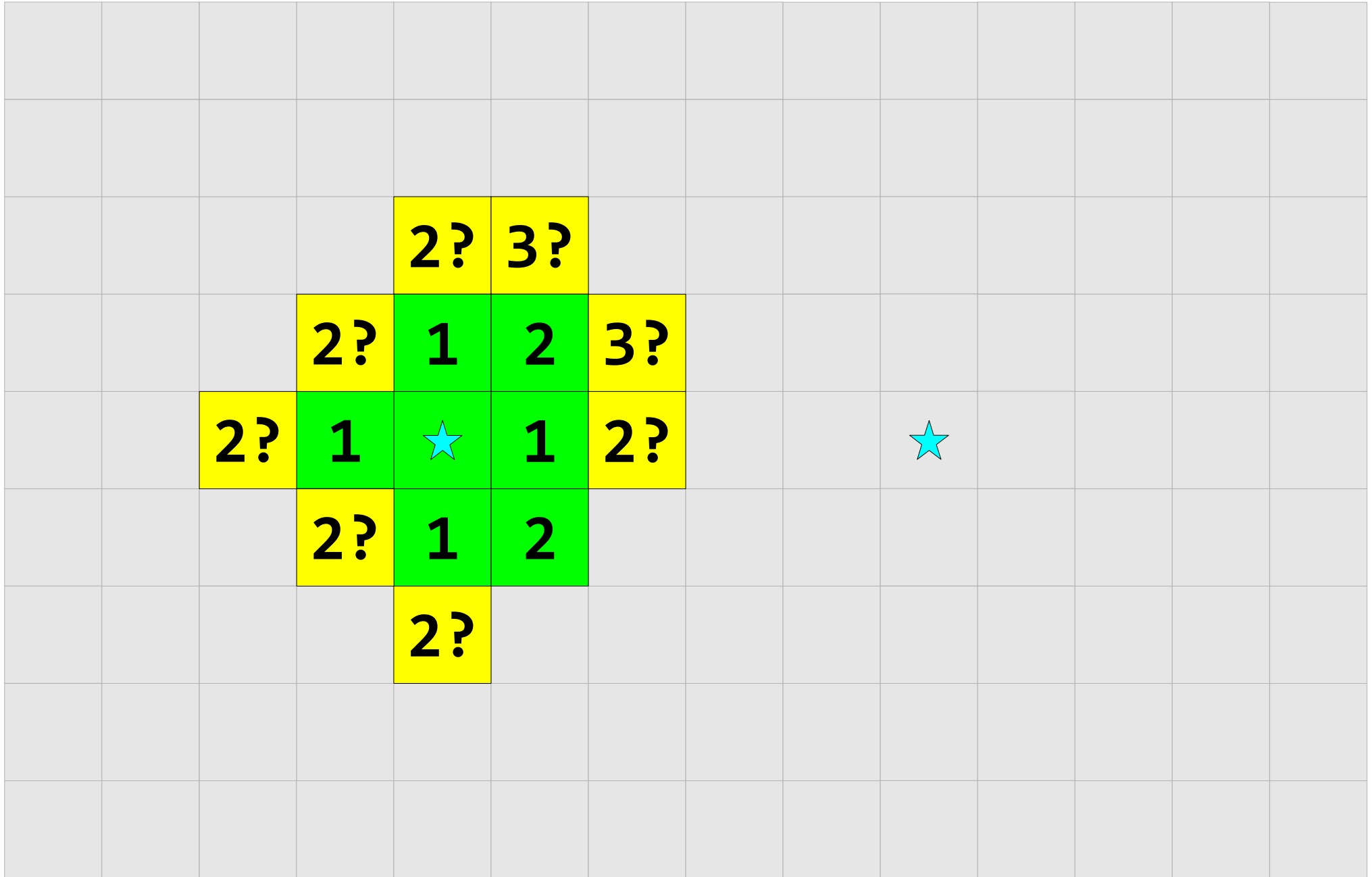












*Skipping a few steps...*

			5?	4	5?	6?							
	6?	5?	4	3	4	5	6?						
6?	5	4	3	2	3	4	5?						
5?	4	3	2	1	2	3	4	5?					
4	3	2	1	★	1	2	3	4	★				
5?	4	3	2	1	2	3	4	5?					
	5?	4	3	2	3	4	5	6?					
	6?	5	4	3	4	5?	6?						
		6?	5?	4	5?								

# How Dijkstra's Works

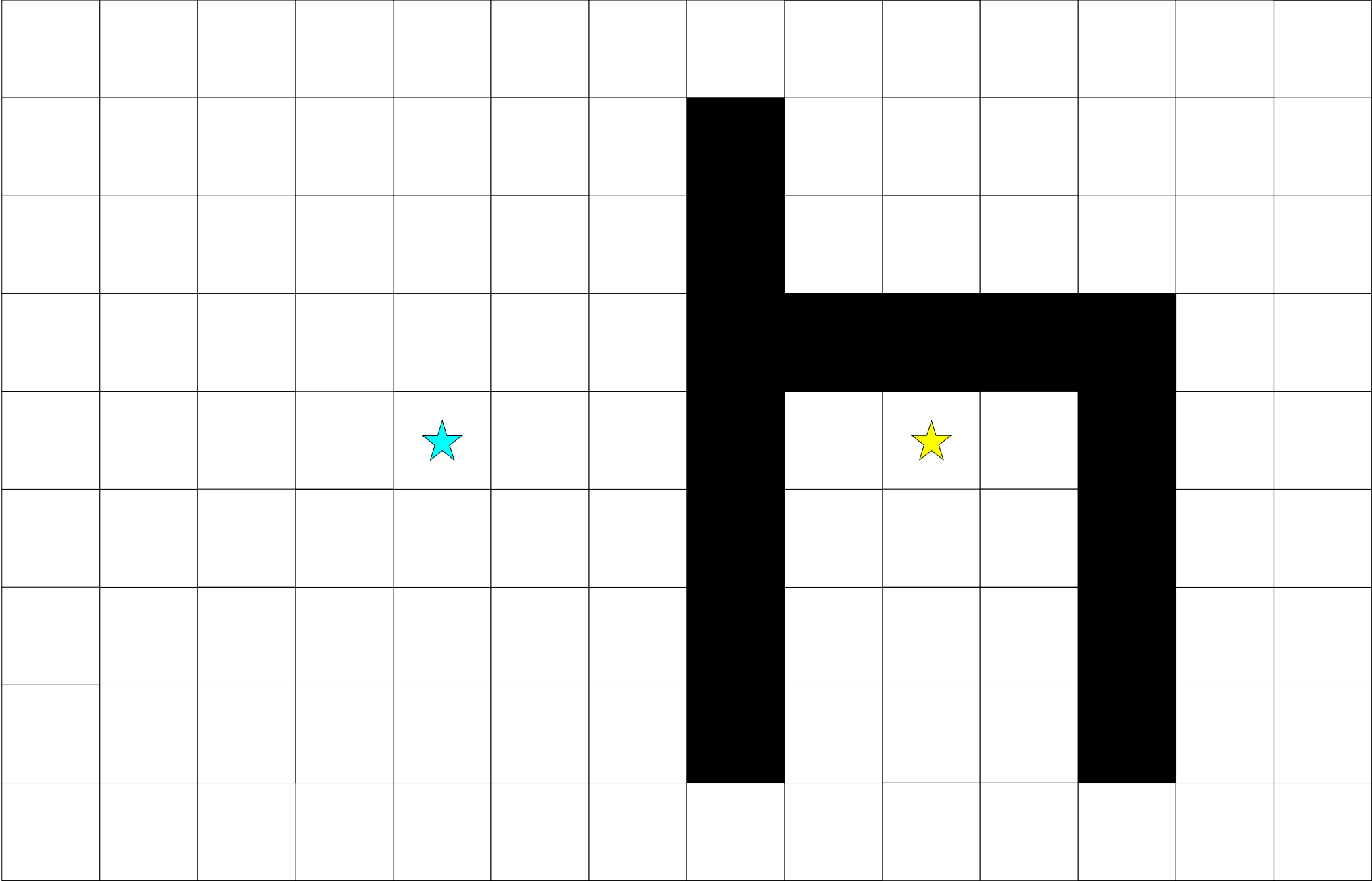
- Dijkstra's algorithm works by incrementally computing the shortest path to intermediary nodes in the graph in case they prove to be useful.
- Most of these nodes are completely in the wrong direction.
- No “big-picture” conception of how to get to the destination – the algorithm explores outward in all directions.
- Could we give the algorithm a hint?

# Heuristics

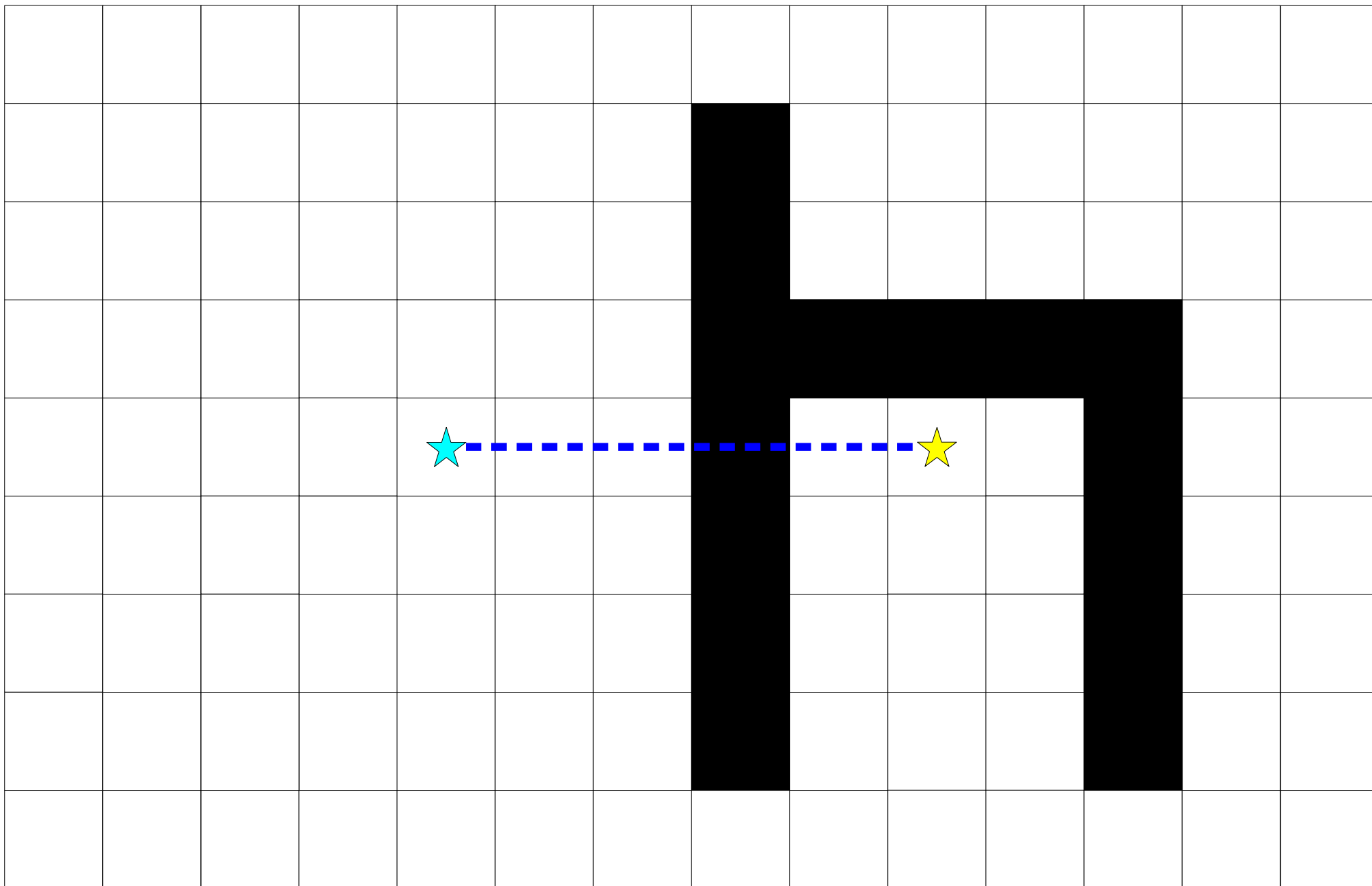
- In the context of graph searches, a **heuristic function** is a function that guesses the distance from some known node to the destination node.
- The guess doesn't have to be correct, but it should try to be as accurate as possible.
- Examples: For Google Maps, a heuristic for estimating distance might be the straight-line distance.

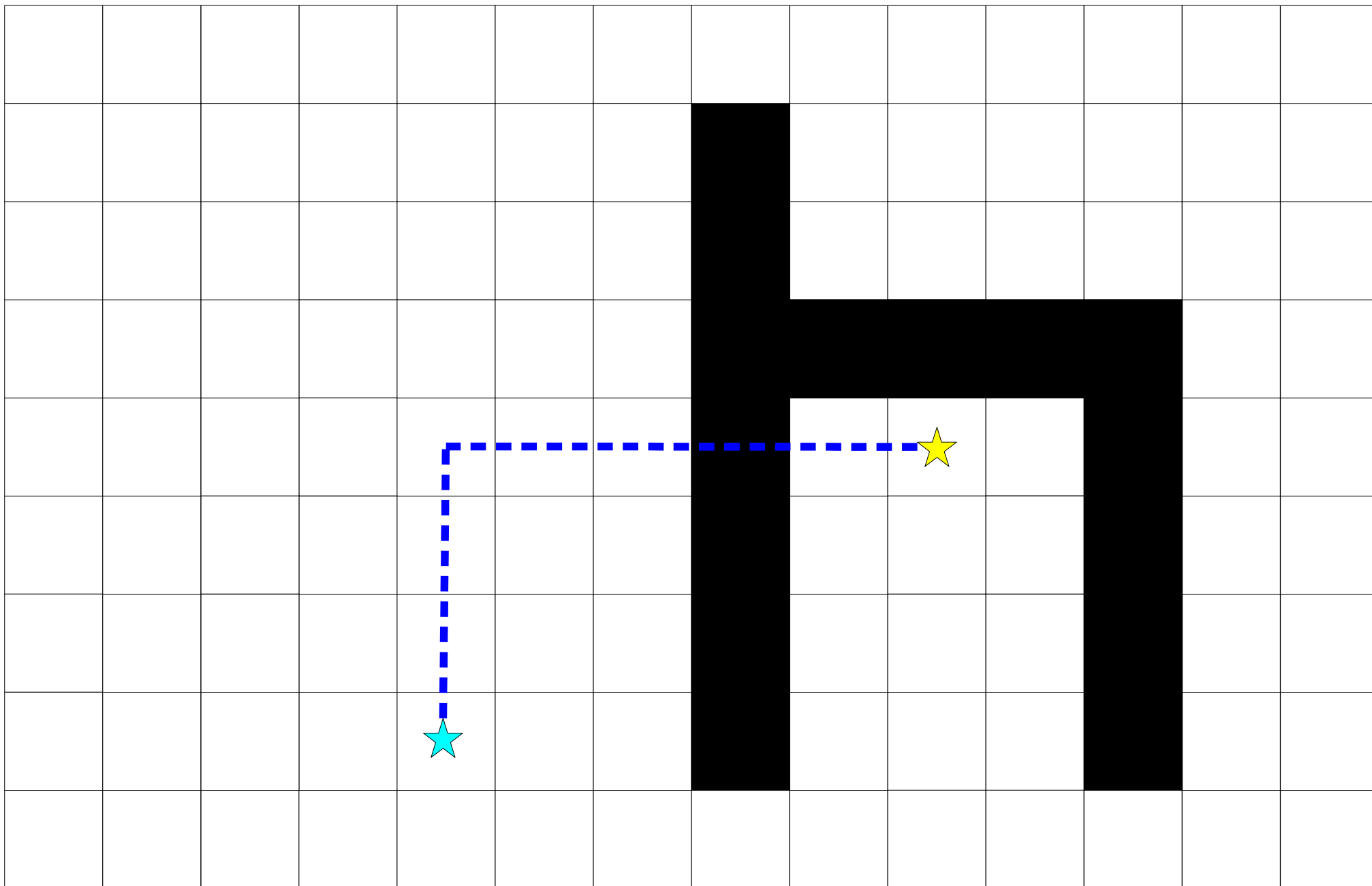
# Admissible Heuristics

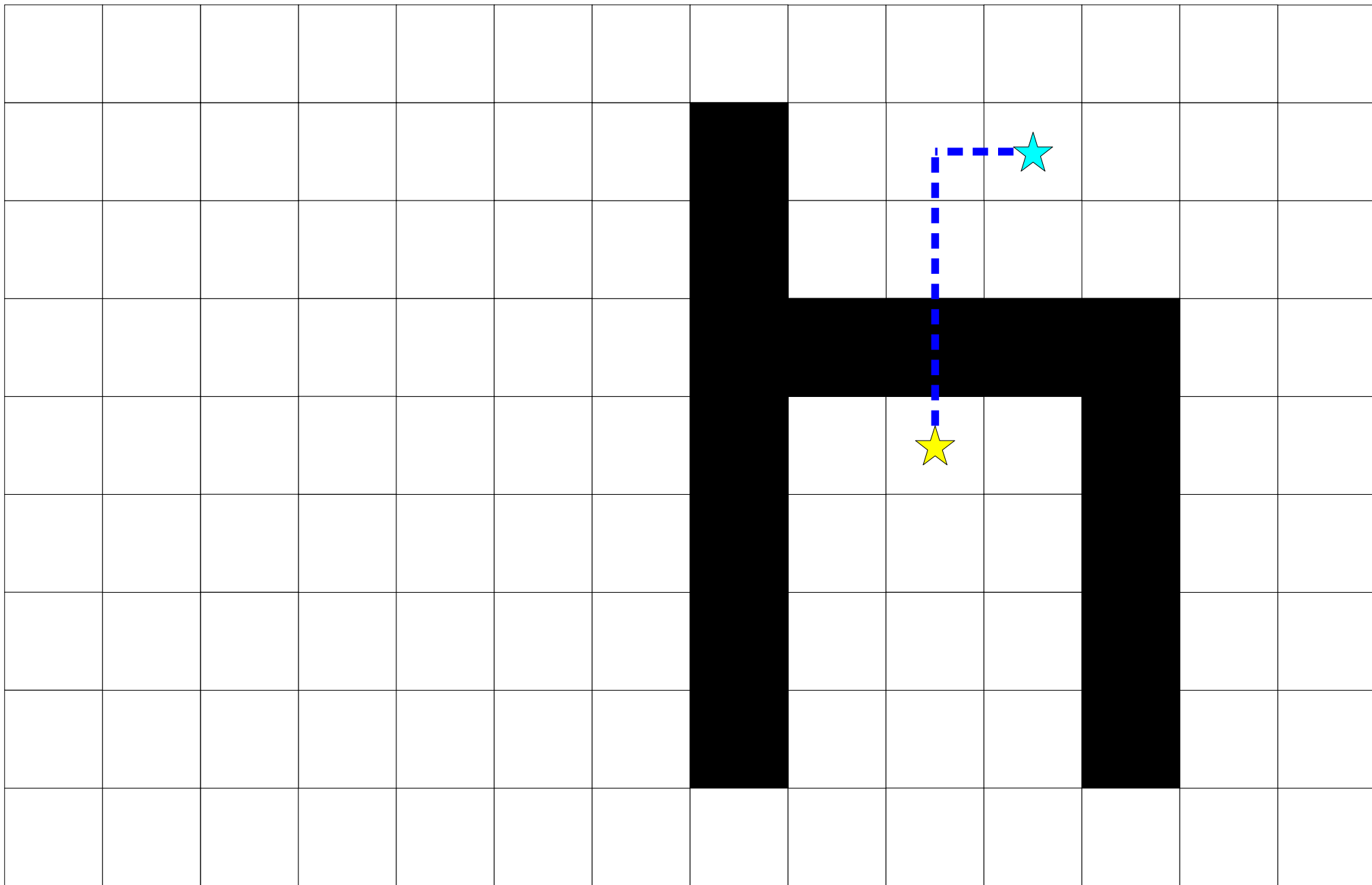
- A heuristic function is called an **admissible heuristic** if it never overestimates the distance from any node to the destination.
- In other words:  
 ***$\text{predicted-distance} \leq \text{actual-distance}$***





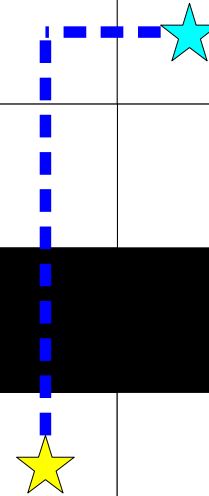






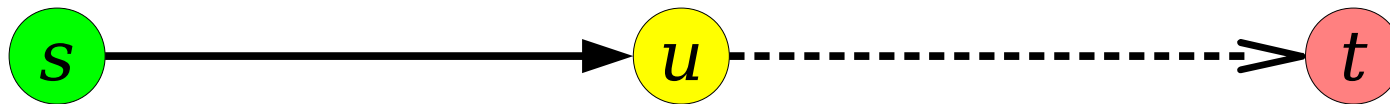
One possible heuristic:

$$|\Delta x| + |\Delta y|$$



# Why Heuristics Matter

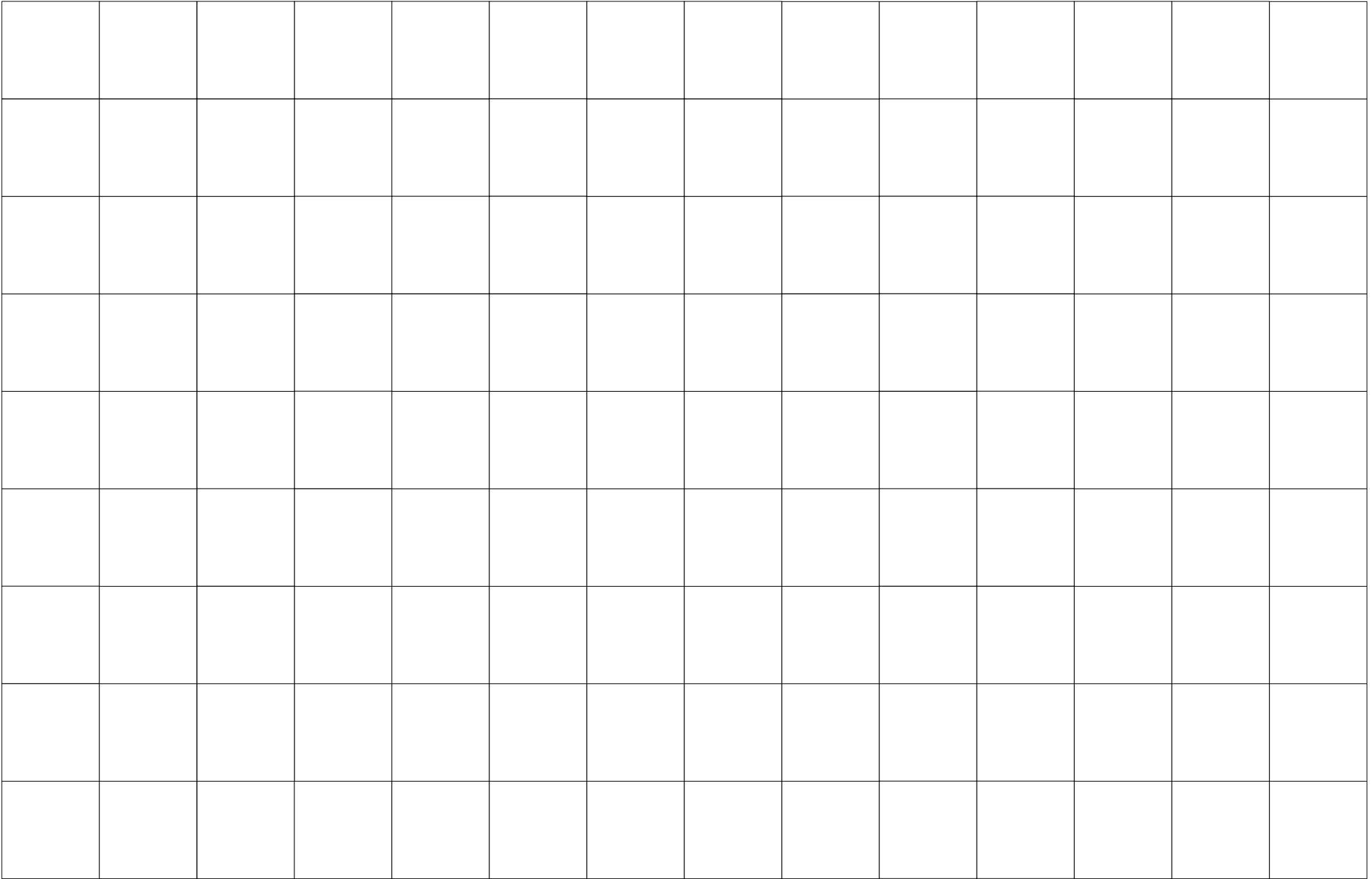
- We can modify Dijkstra's algorithm by introducing heuristic functions.
- Given any node  $u$ , there are two associated costs:



- The actual distance from the start node  $s$ .
- The heuristic distance from  $u$  to the end node  $t$ .
- Key idea: Run Dijkstra's algorithm, but use the following priority in the priority queue:

$$\text{priority}(u) = \text{distance}(s, u) + \text{heuristic}(u, t)$$

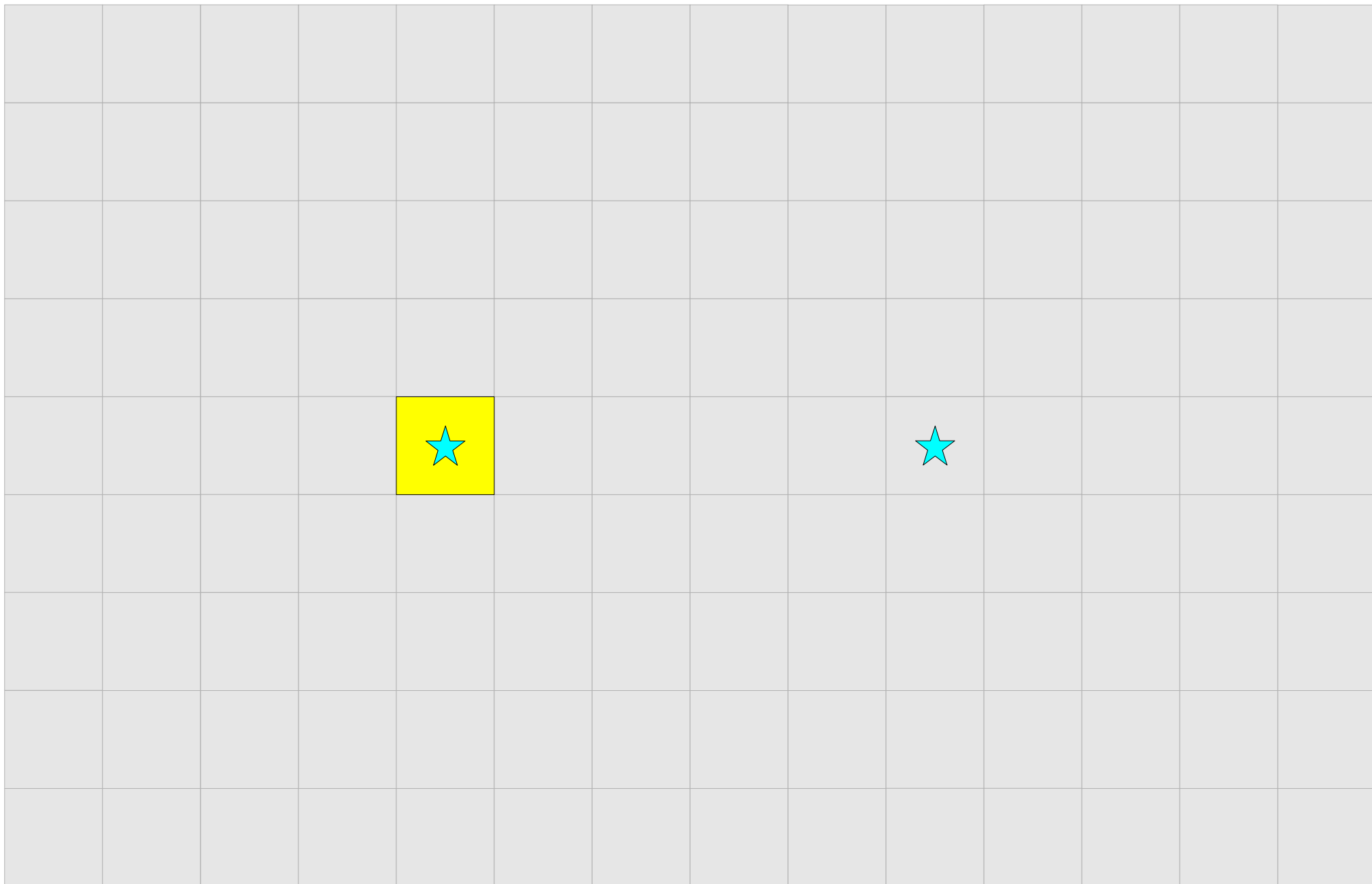
- This modification of Dijkstra's algorithm is called the **A\* search algorithm**.

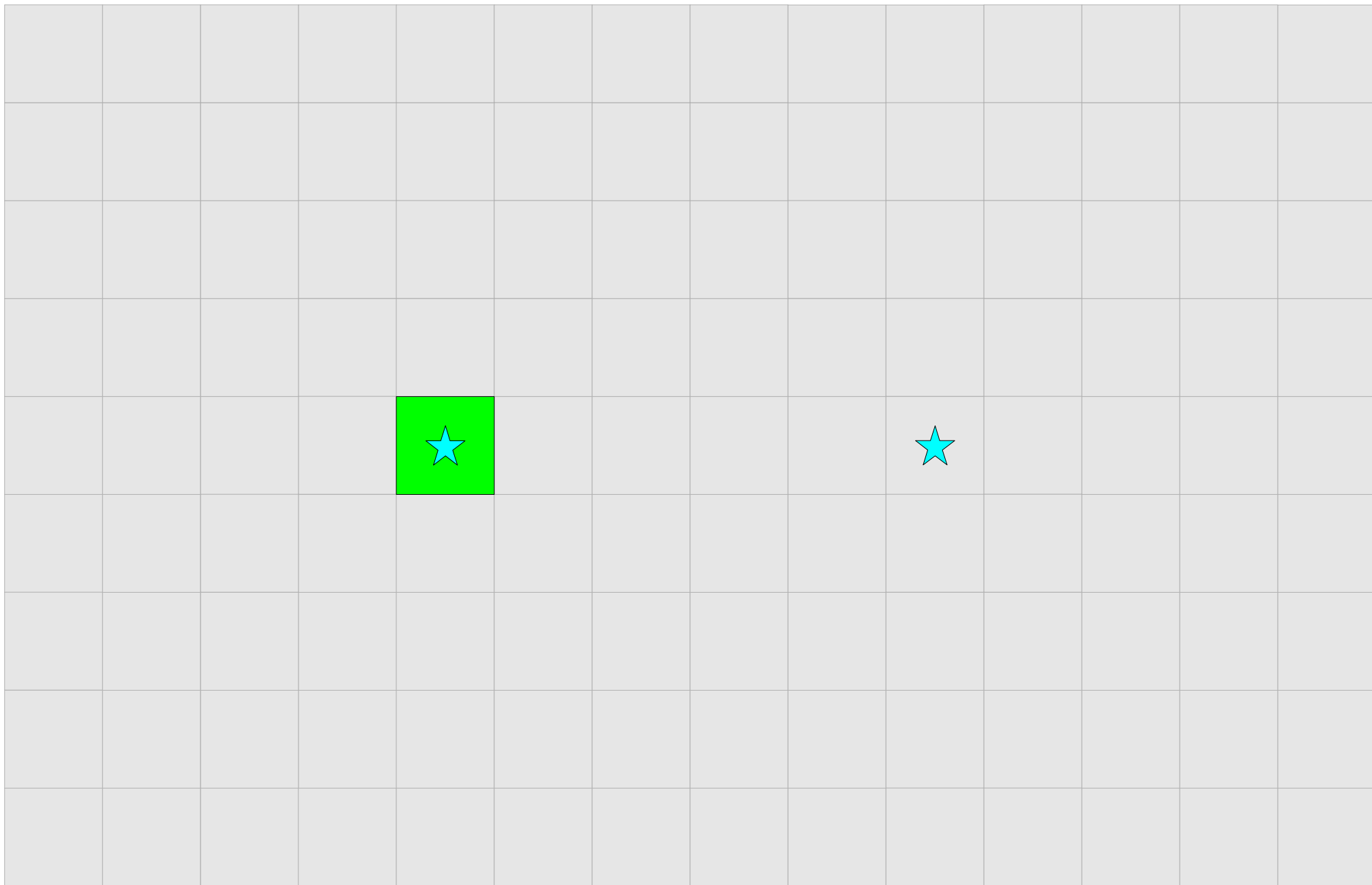


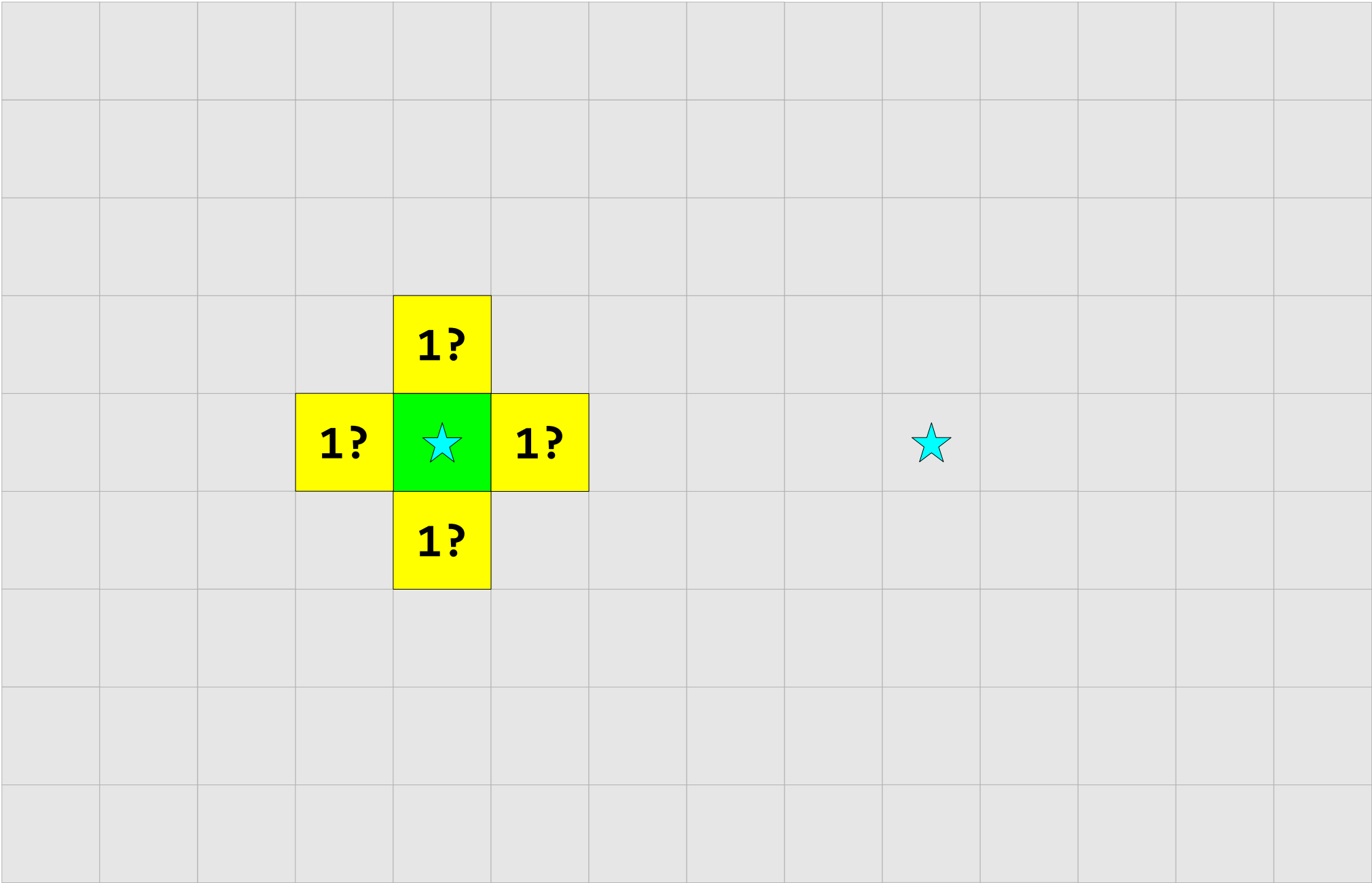


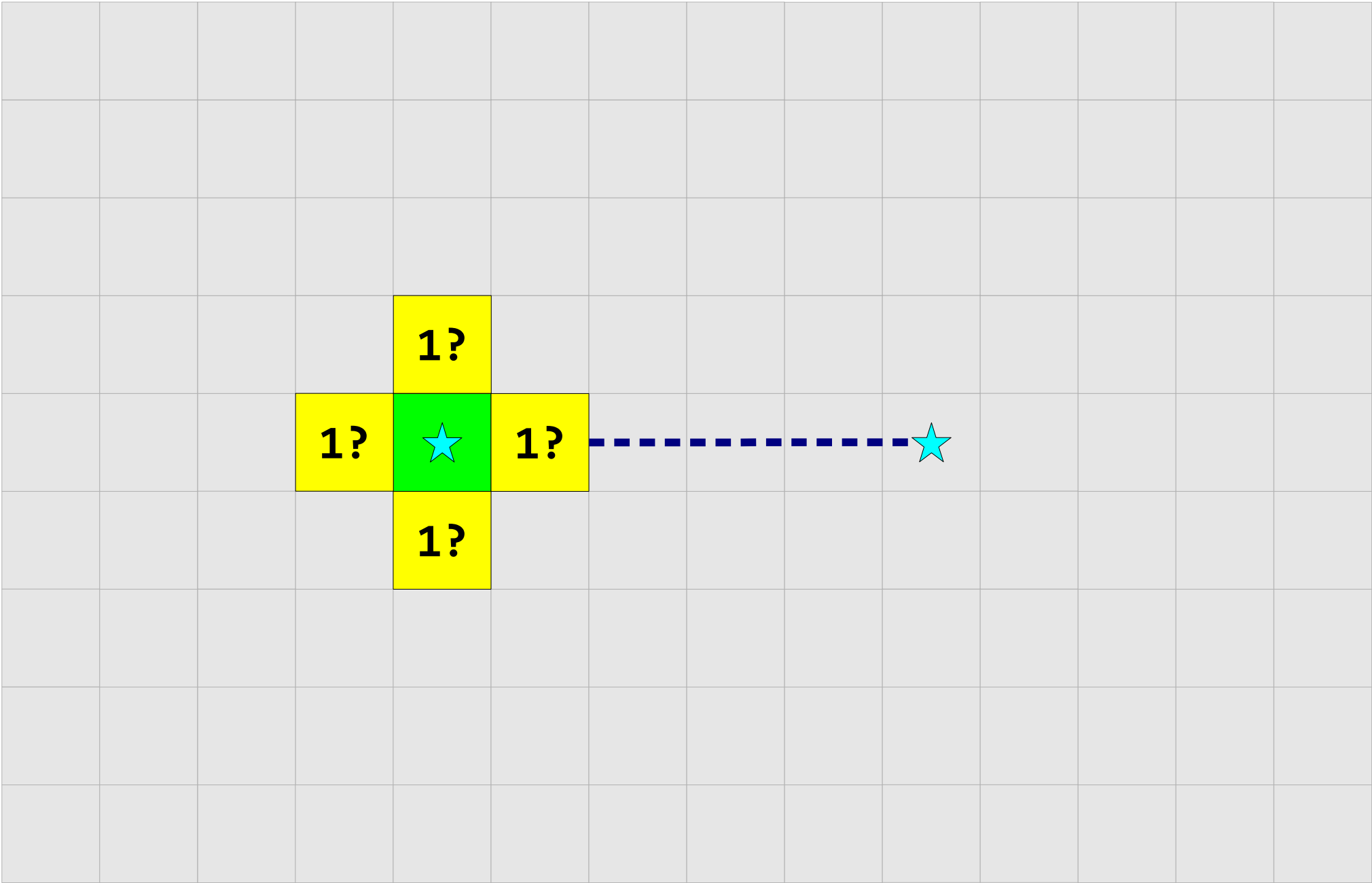


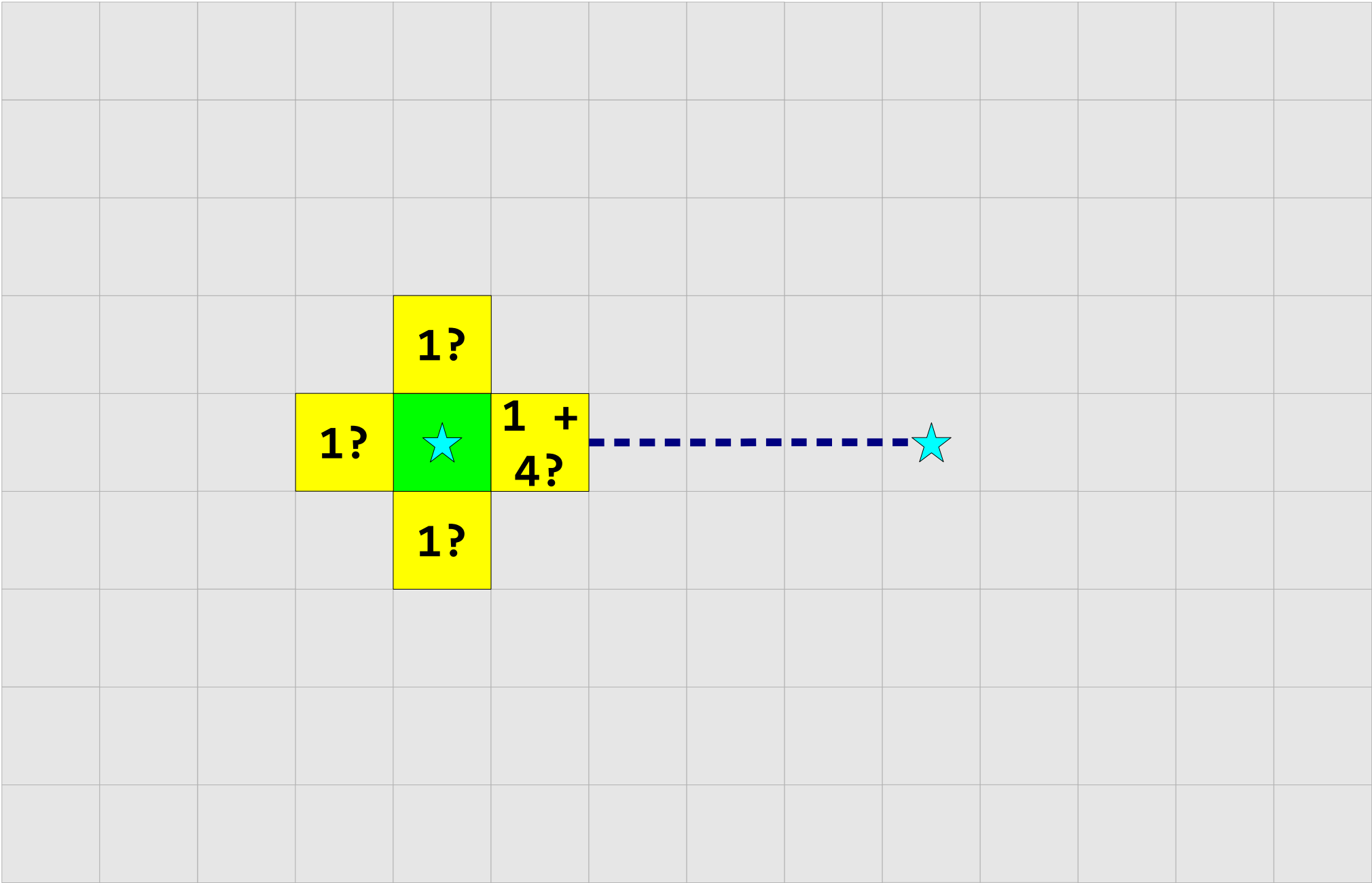


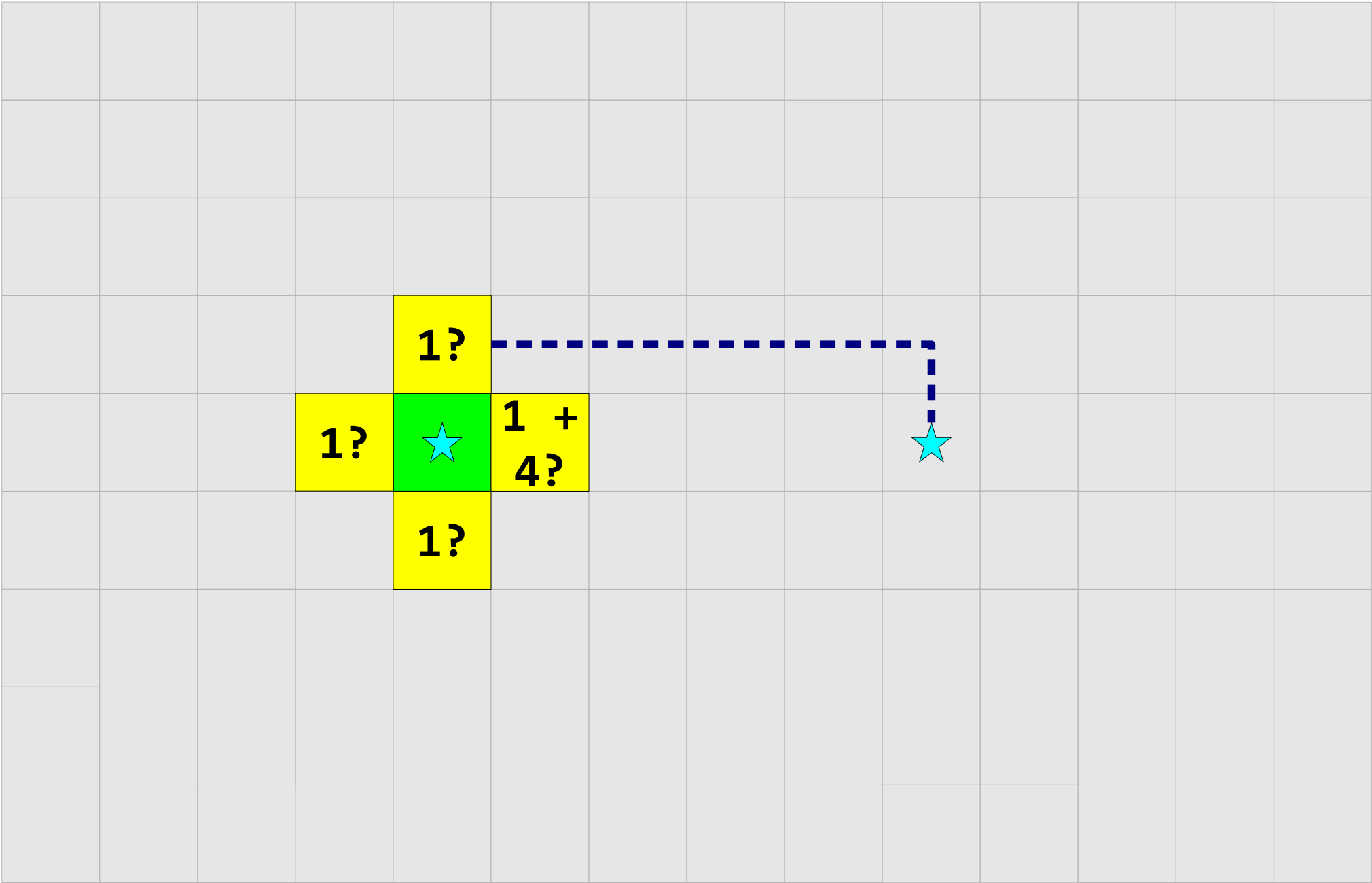


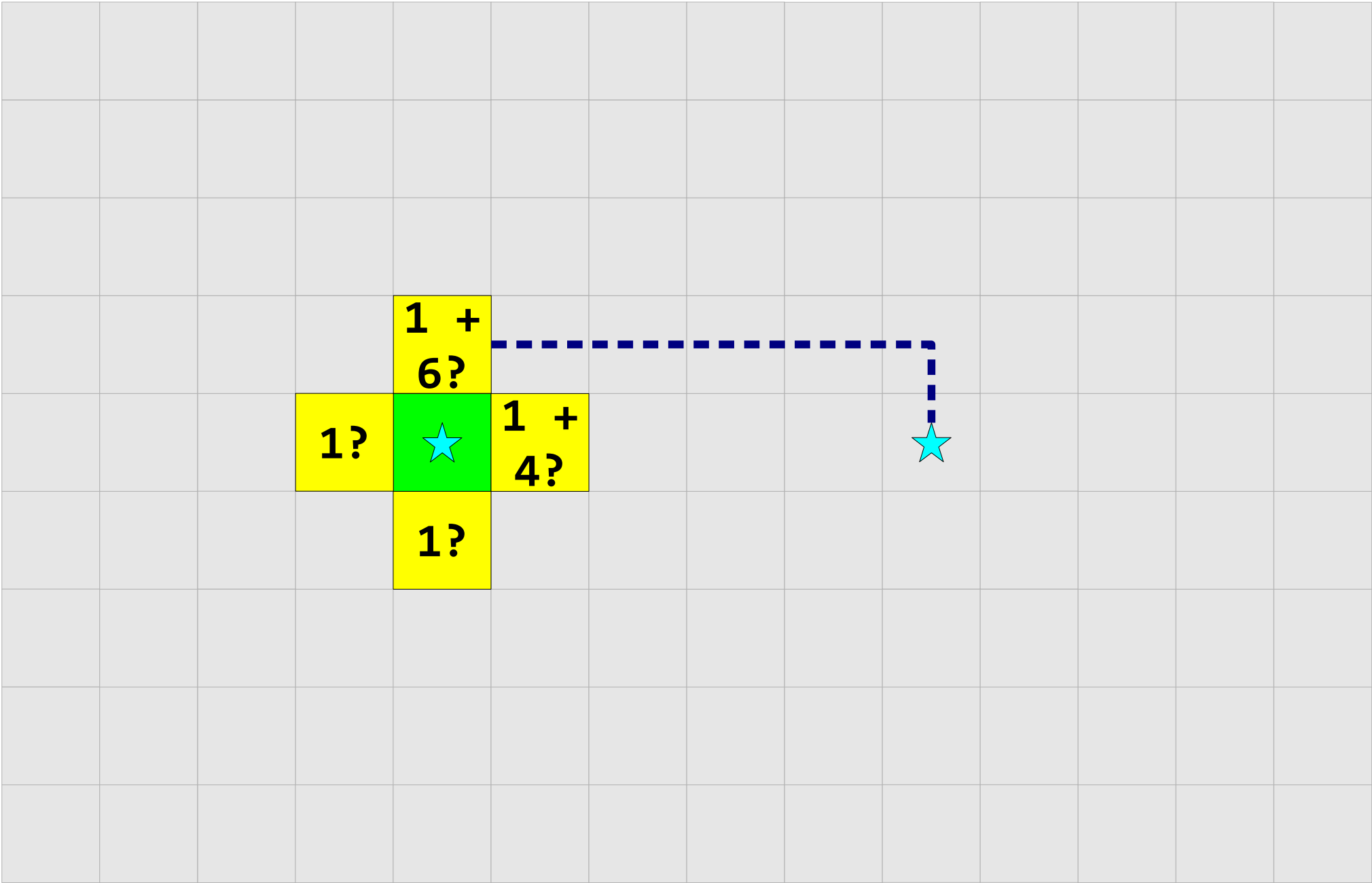


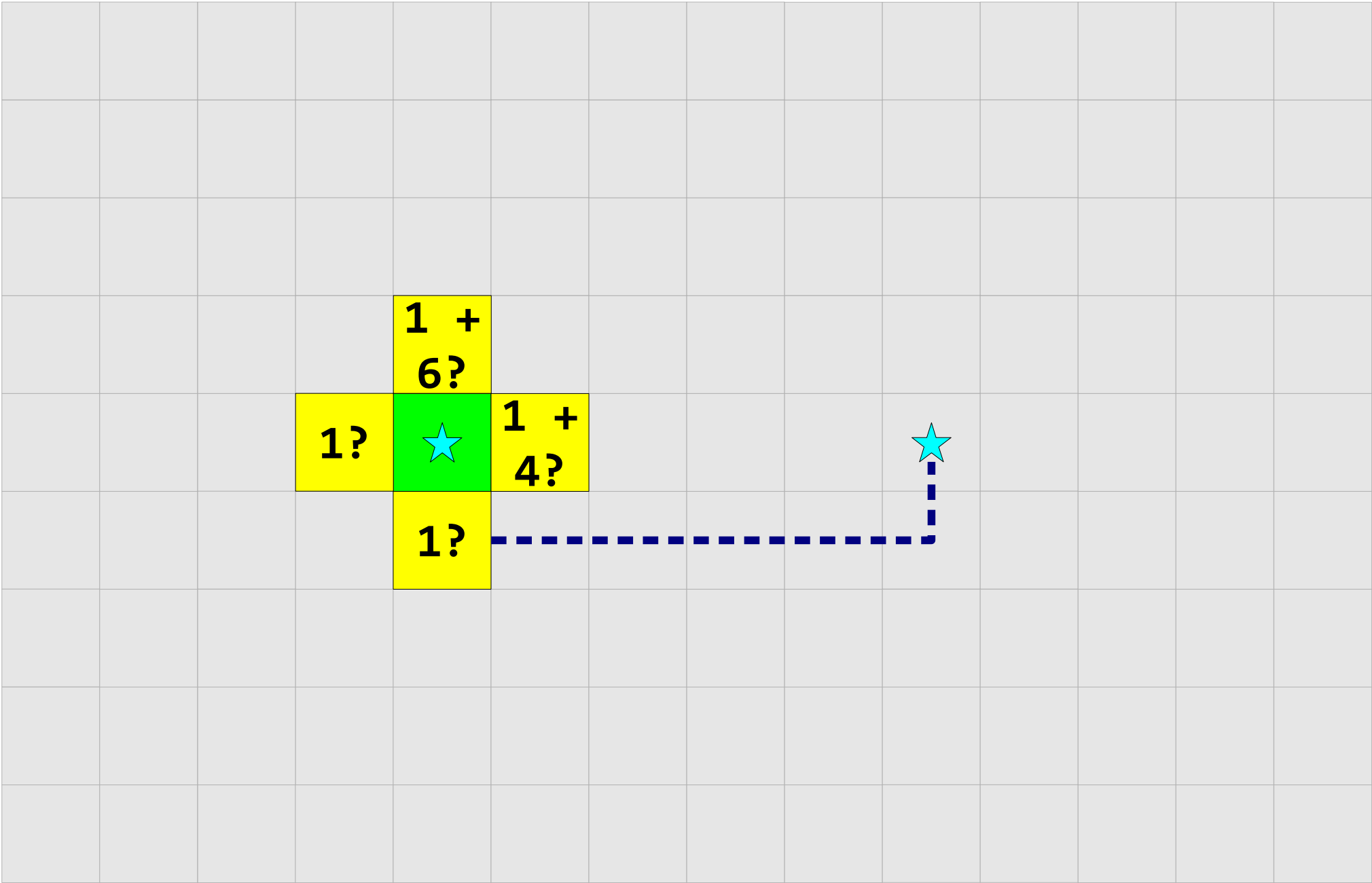




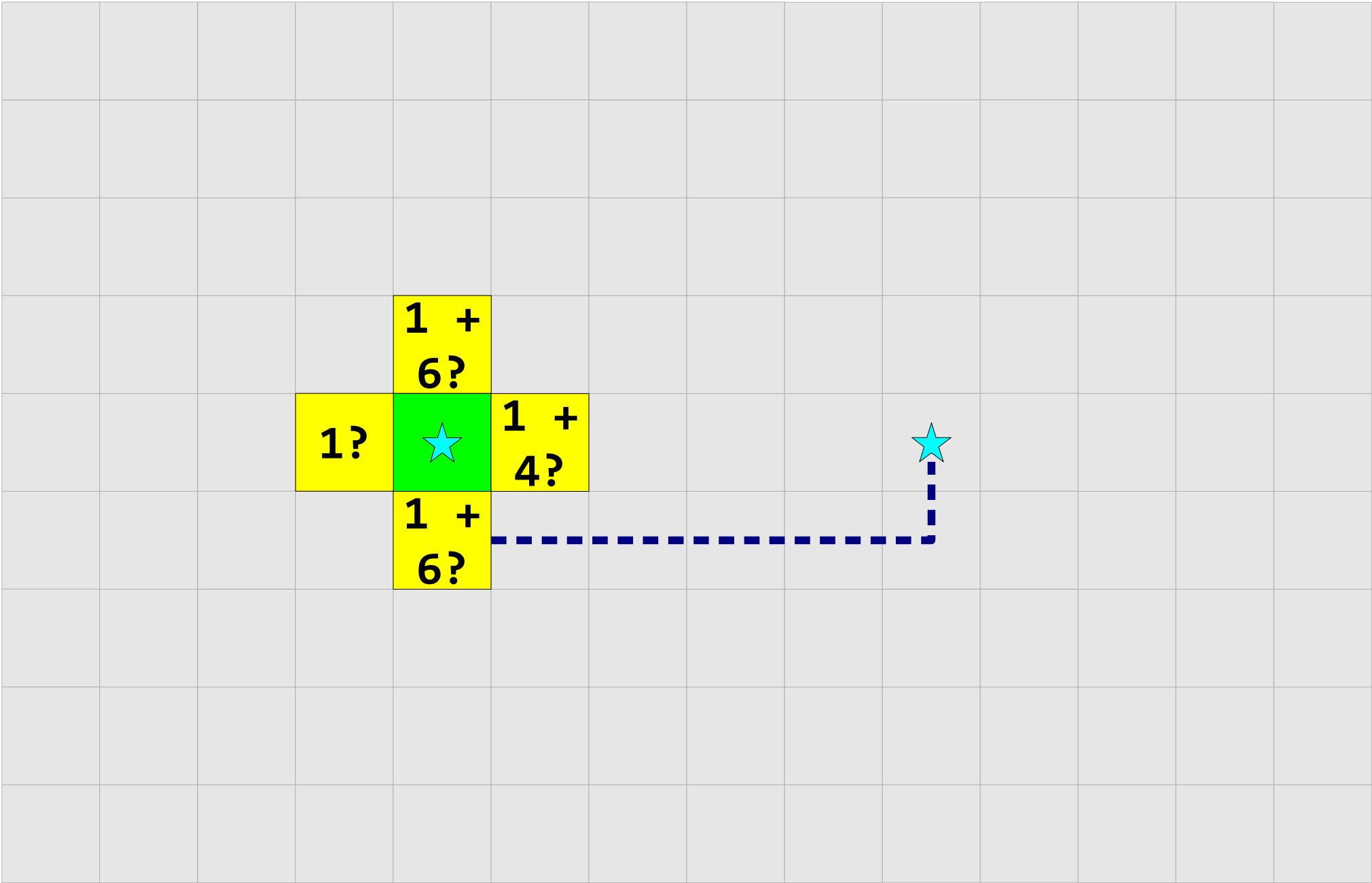


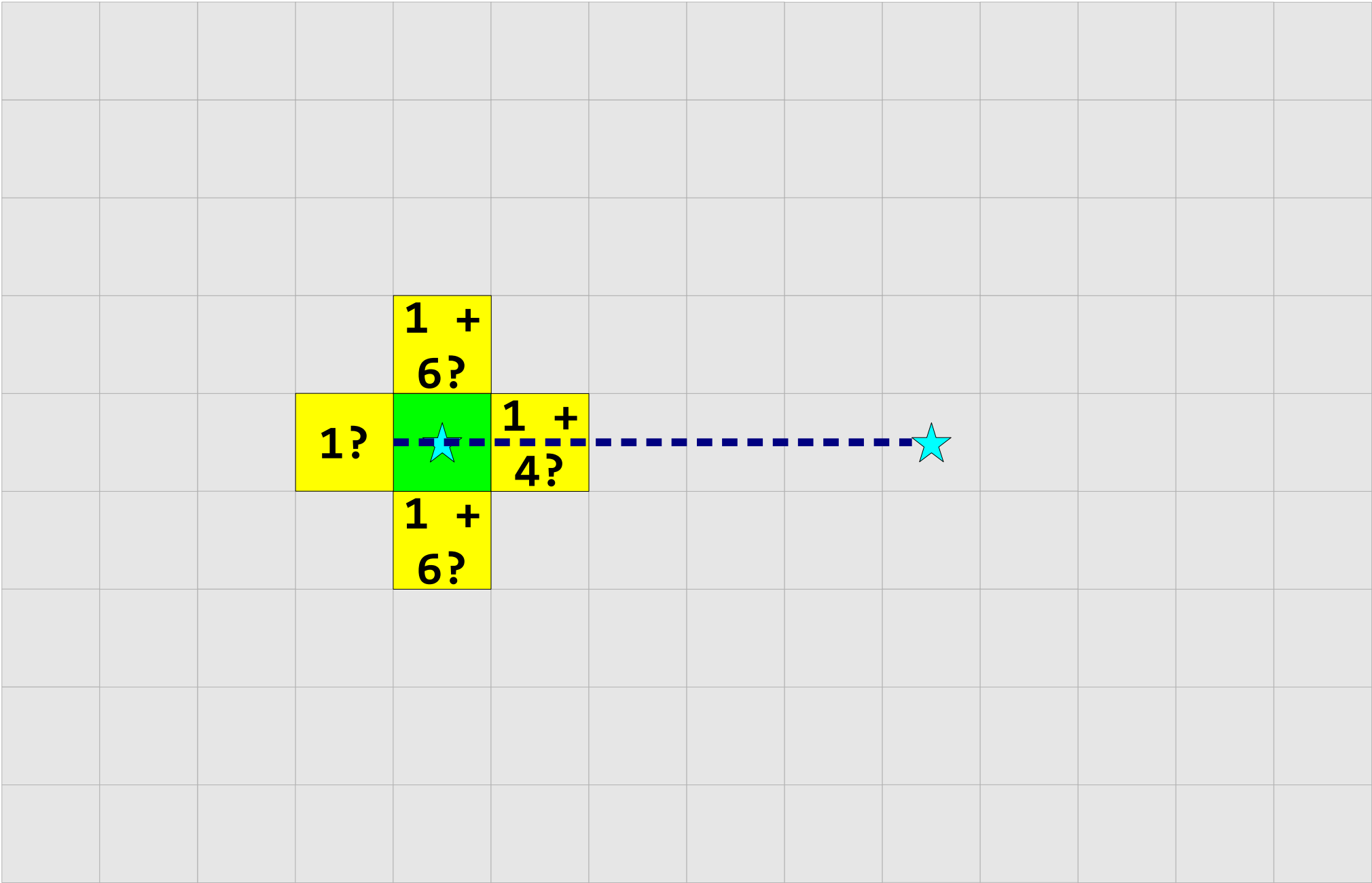


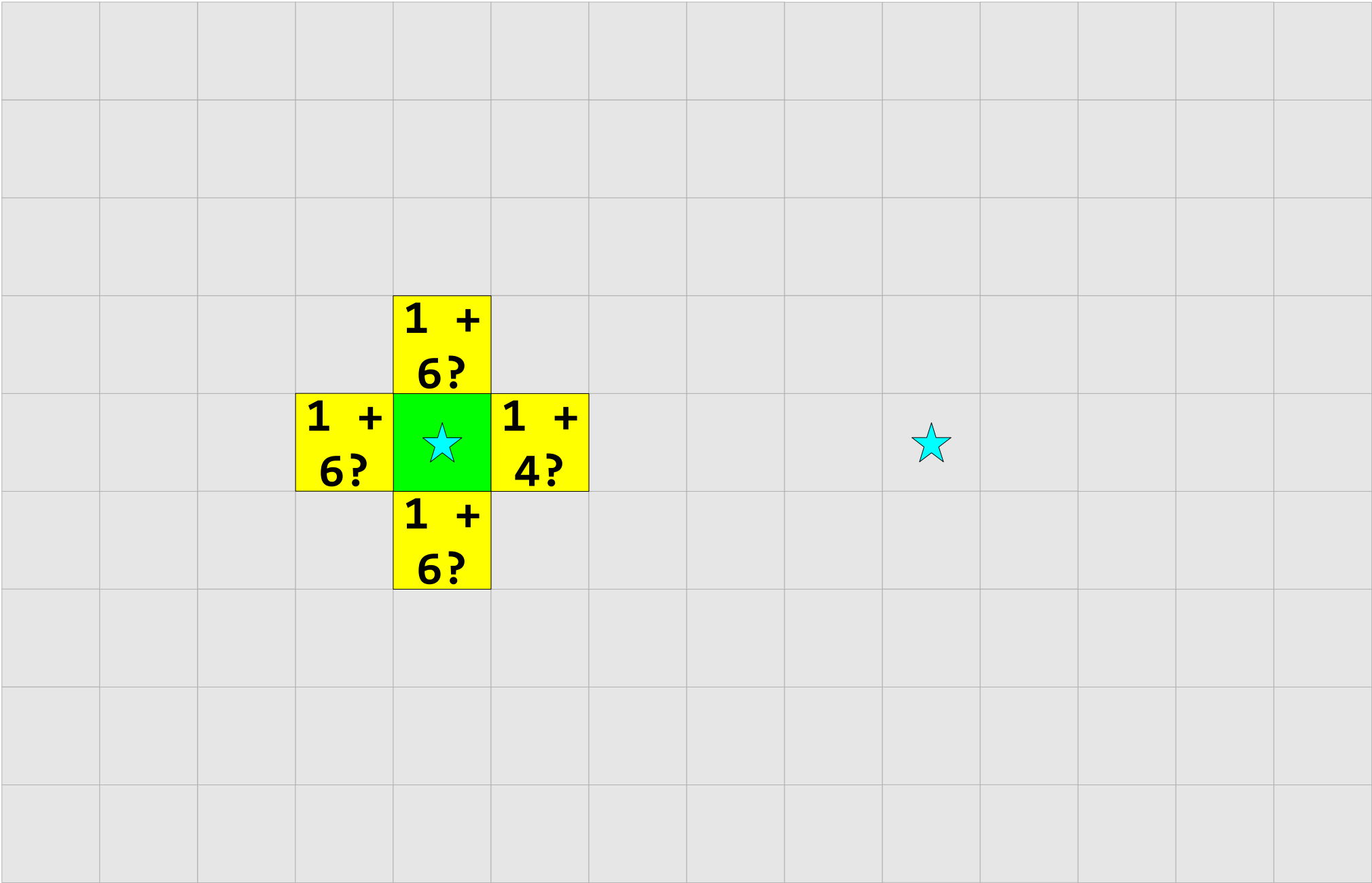


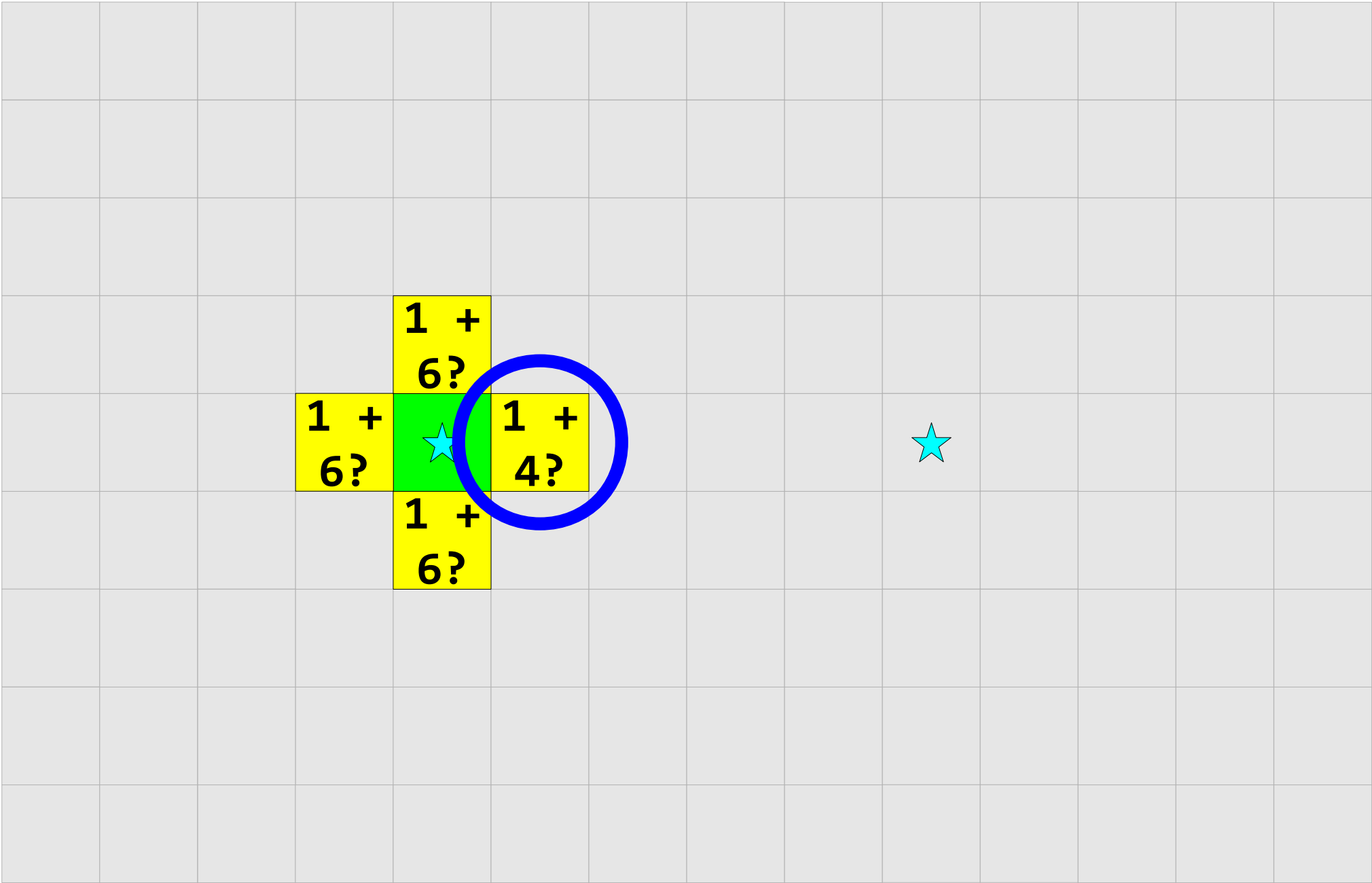






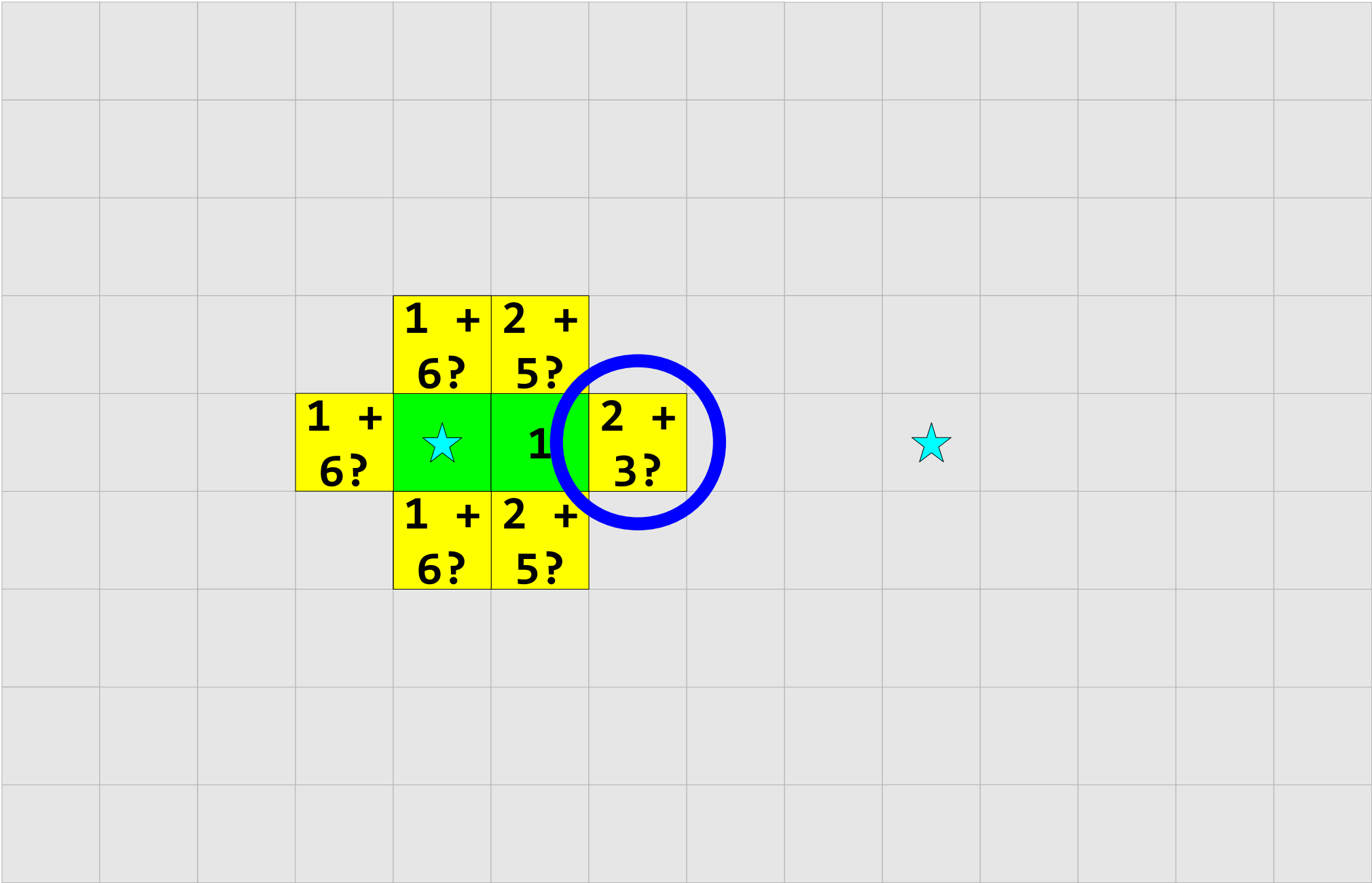


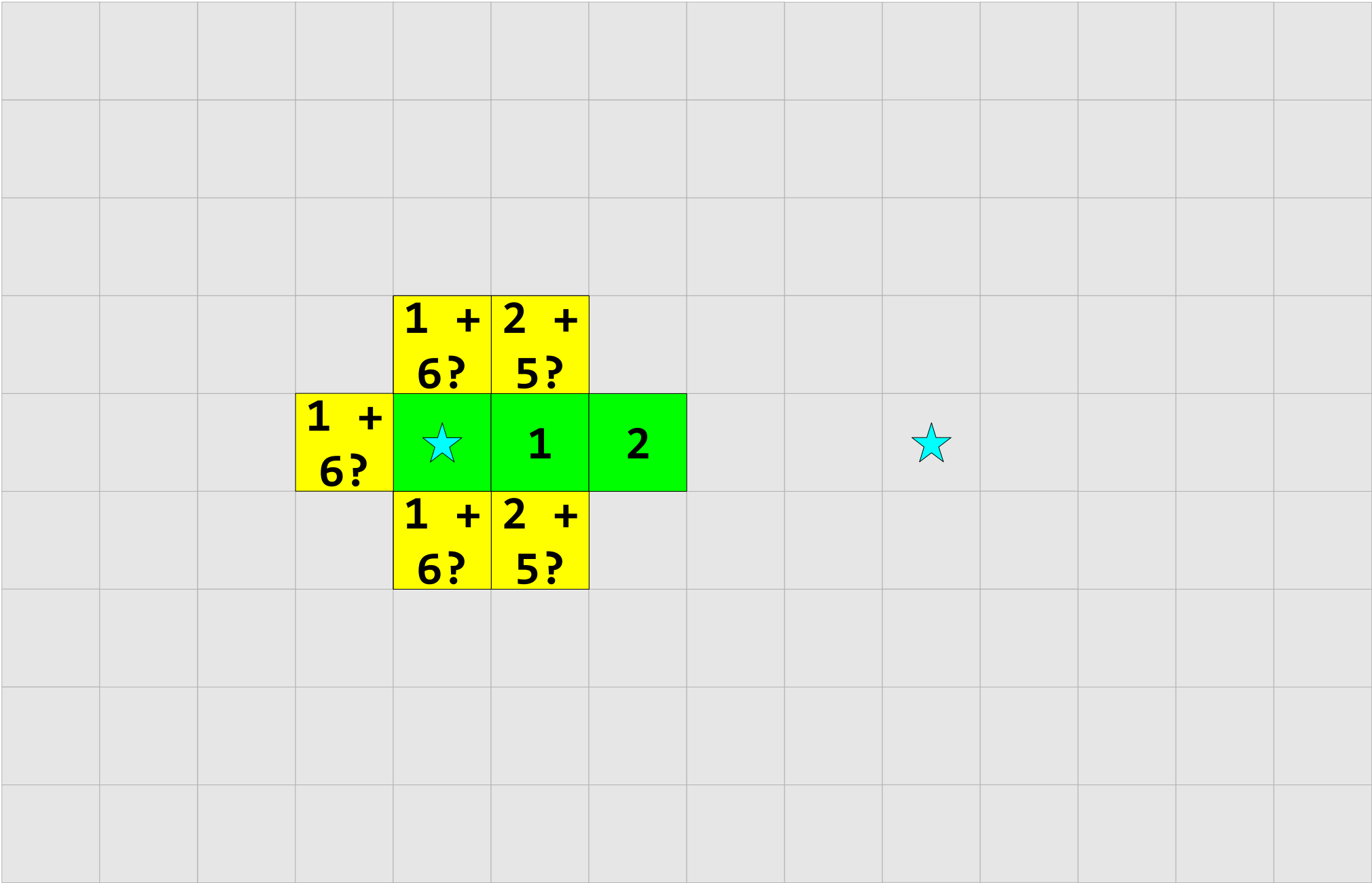






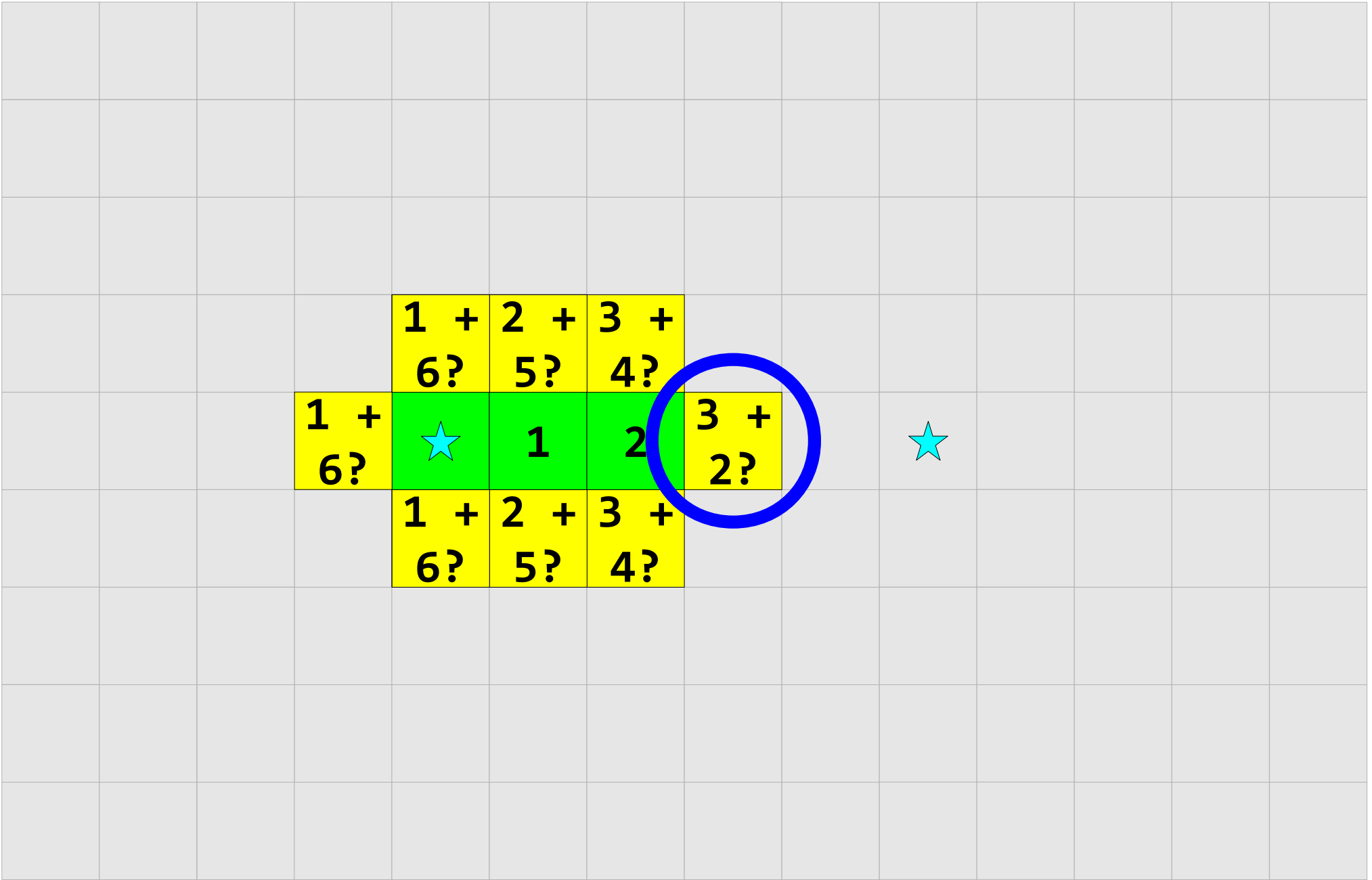




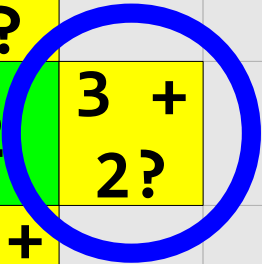






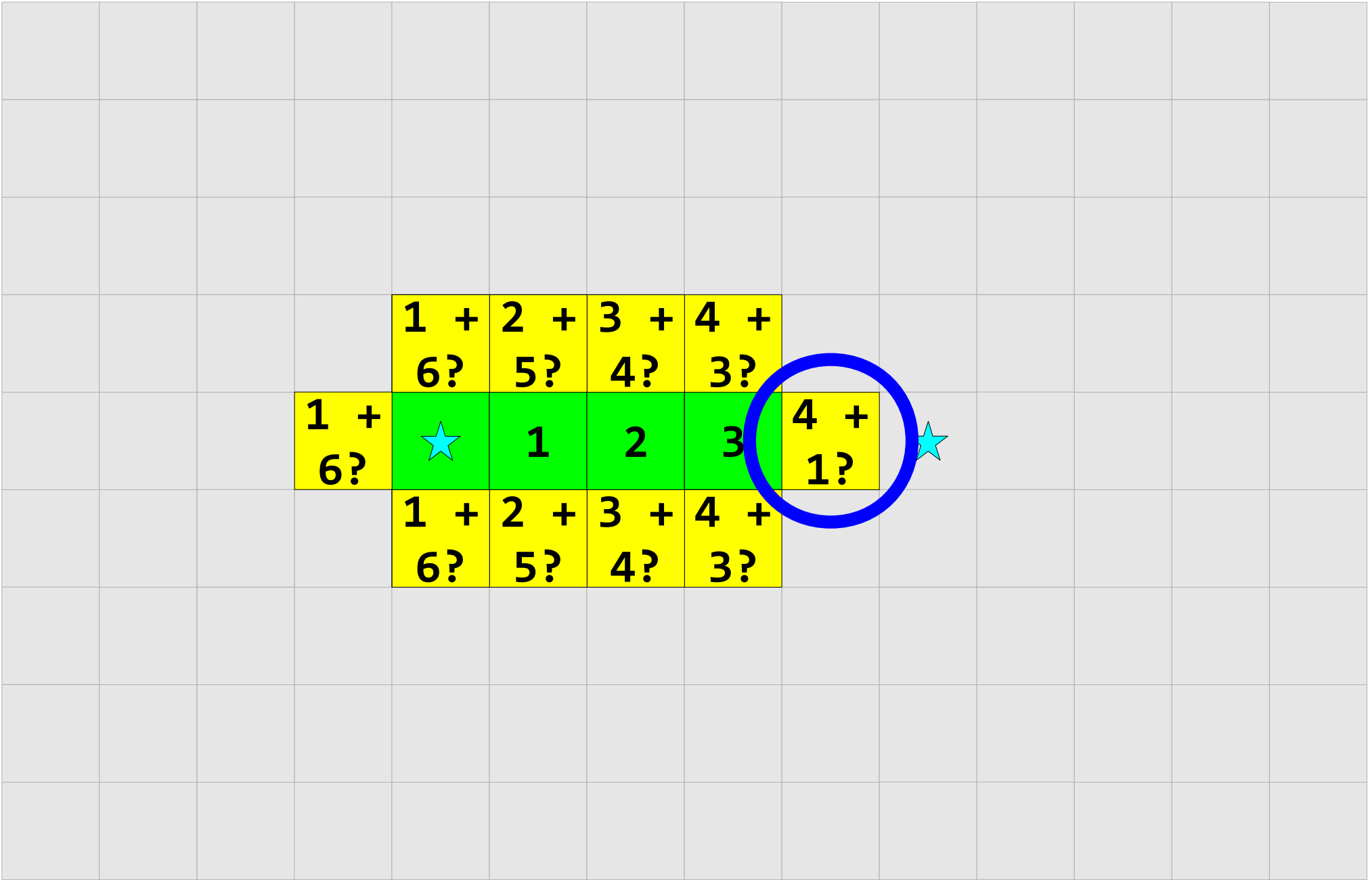


				1 +	2 +	3 +								
				6?	5?	4?								
			1 +	★	1	2	3 +							
			6?				2?							
				1 +	2 +	3 +								
				6?	5?	4?								



A 10x10 grid with a central 3x3 area highlighted in green. The central cell contains a blue star. The four cells adjacent to the center (up, down, left, right) contain the numbers 1, 2, 3, and 4 respectively. The four cells immediately outside this green area (up-up, up-down, down-up, down-down) contain the text "1 + 6?", "2 + 5?", "3 + 4?", and "4 + 3?" respectively. The four cells at the corners of this 3x3 area (up-left, up-right, down-left, down-right) are empty.

	1 + 6?	2 + 5?	3 + 4?	4 + 3?		
1 + 6?	★	1	2	3	4 + 1?	★
	1 + 6?	2 + 5?	3 + 4?	4 + 3?		



	1 + 6?	2 + 5?	3 + 4?	4 + 3?		
1 + 6?	★	1	2	3	4	★
	1 + 6?	2 + 5?	3 + 4?	4 + 3?		

	1 + 6?	2 + 5?	3 + 4?	4 + 3?	5 + 2?	
1 + 6?	★	1	2	3	4	5 + 0? ★
	1 + 6?	2 + 5?	3 + 4?	4 + 3?	5 + 2?	

	1 + 6?	2 + 5?	3 + 4?	4 + 3?	5 + 2?	
1 + 6?	★	1	2	3	4	5 + 0? ★
	1 + 6?	2 + 5?	3 + 4?	4 + 3?	5 + 2?	



1 +	2 +	3 +	4 +	5 +
6?	5?	4?	3?	2?

1 +
6?



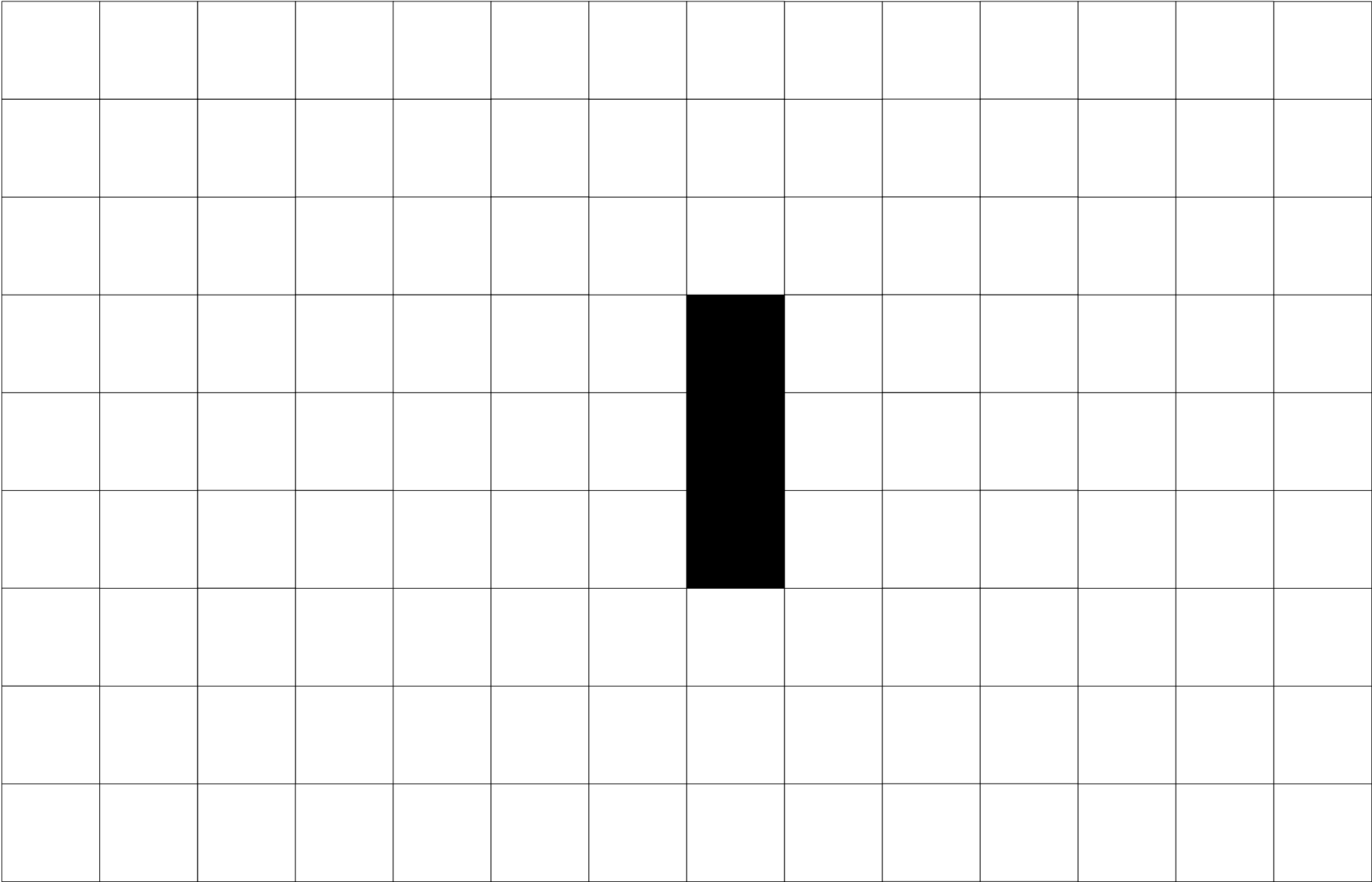
1

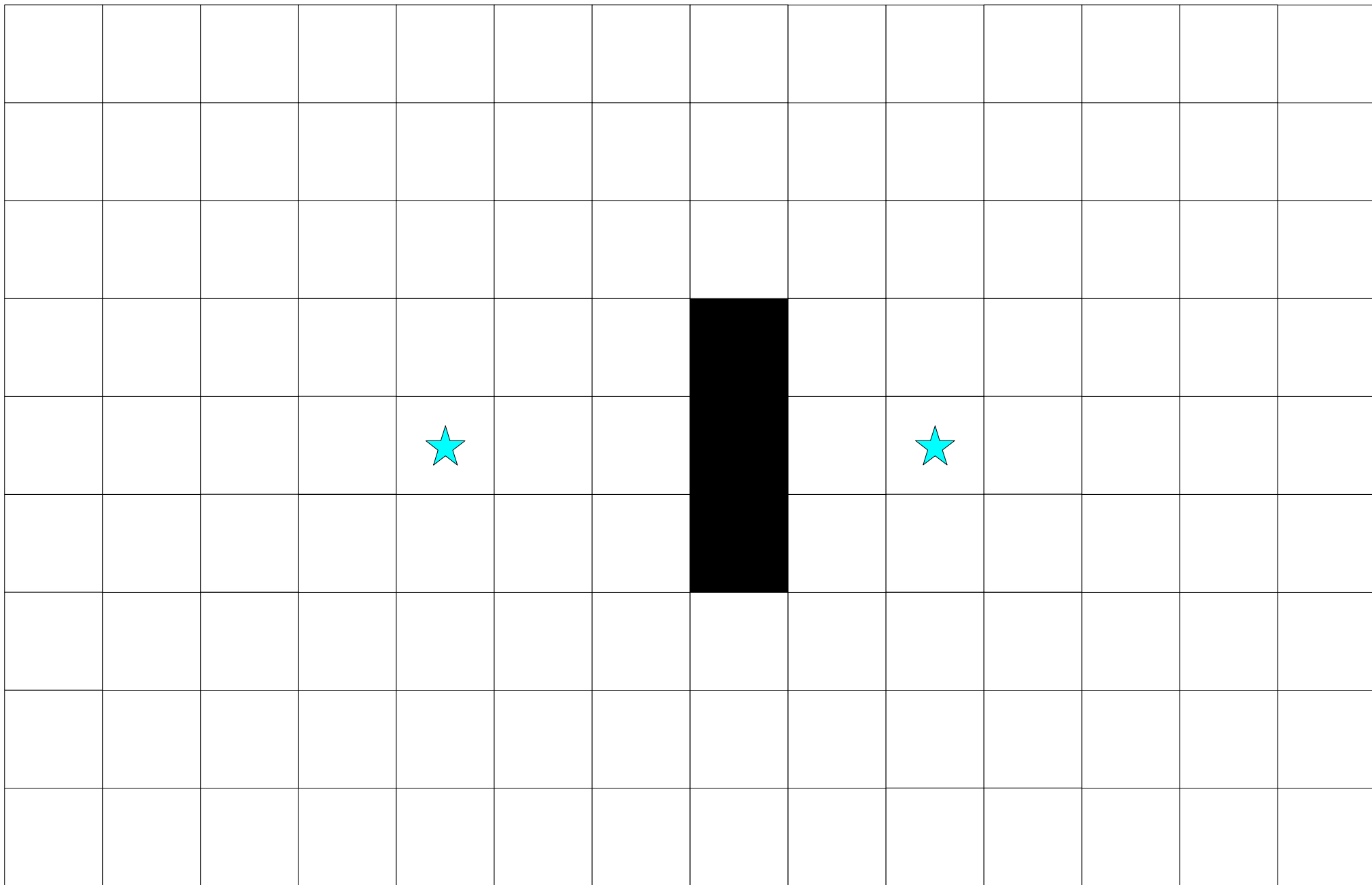
2

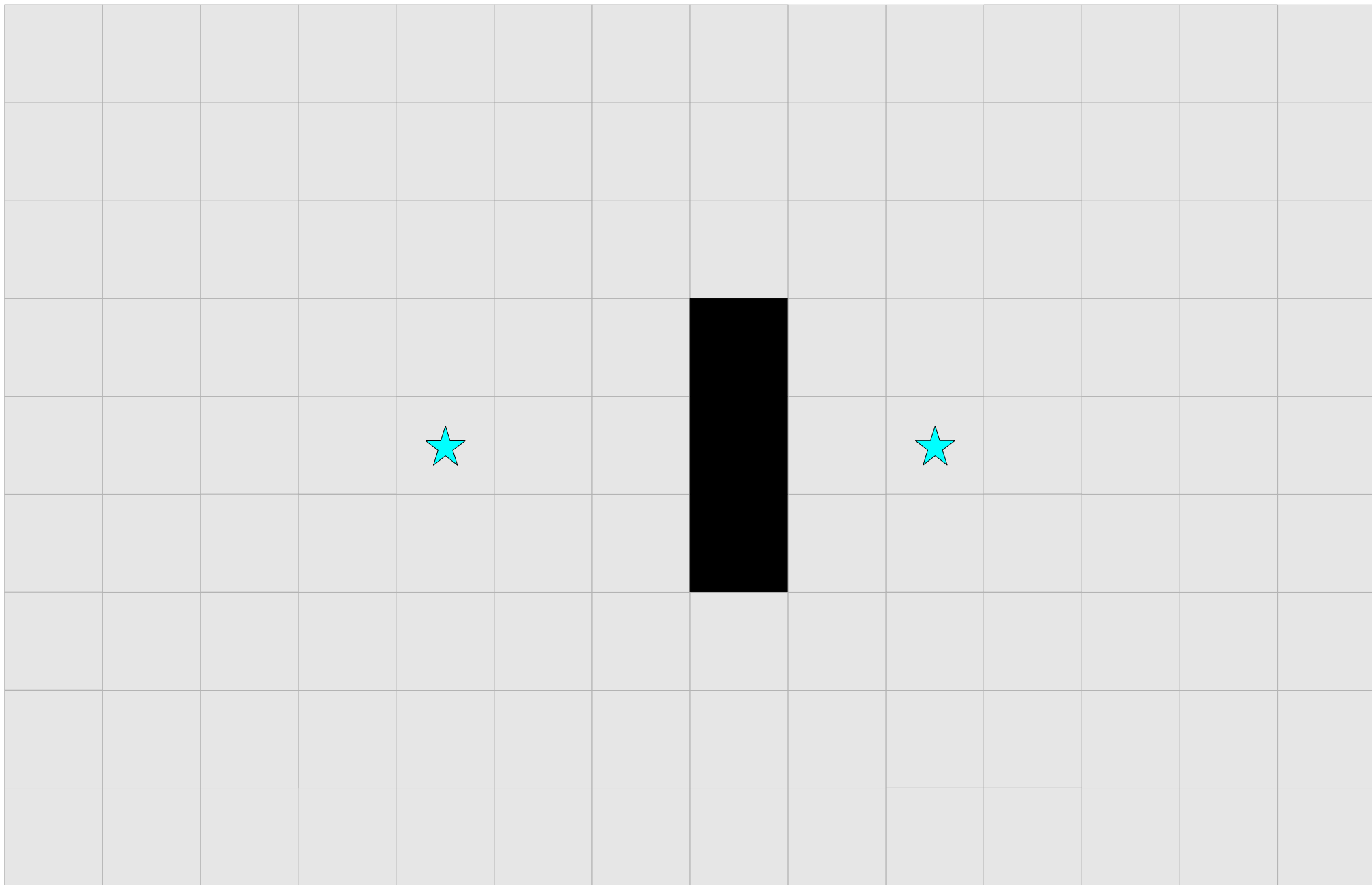
3

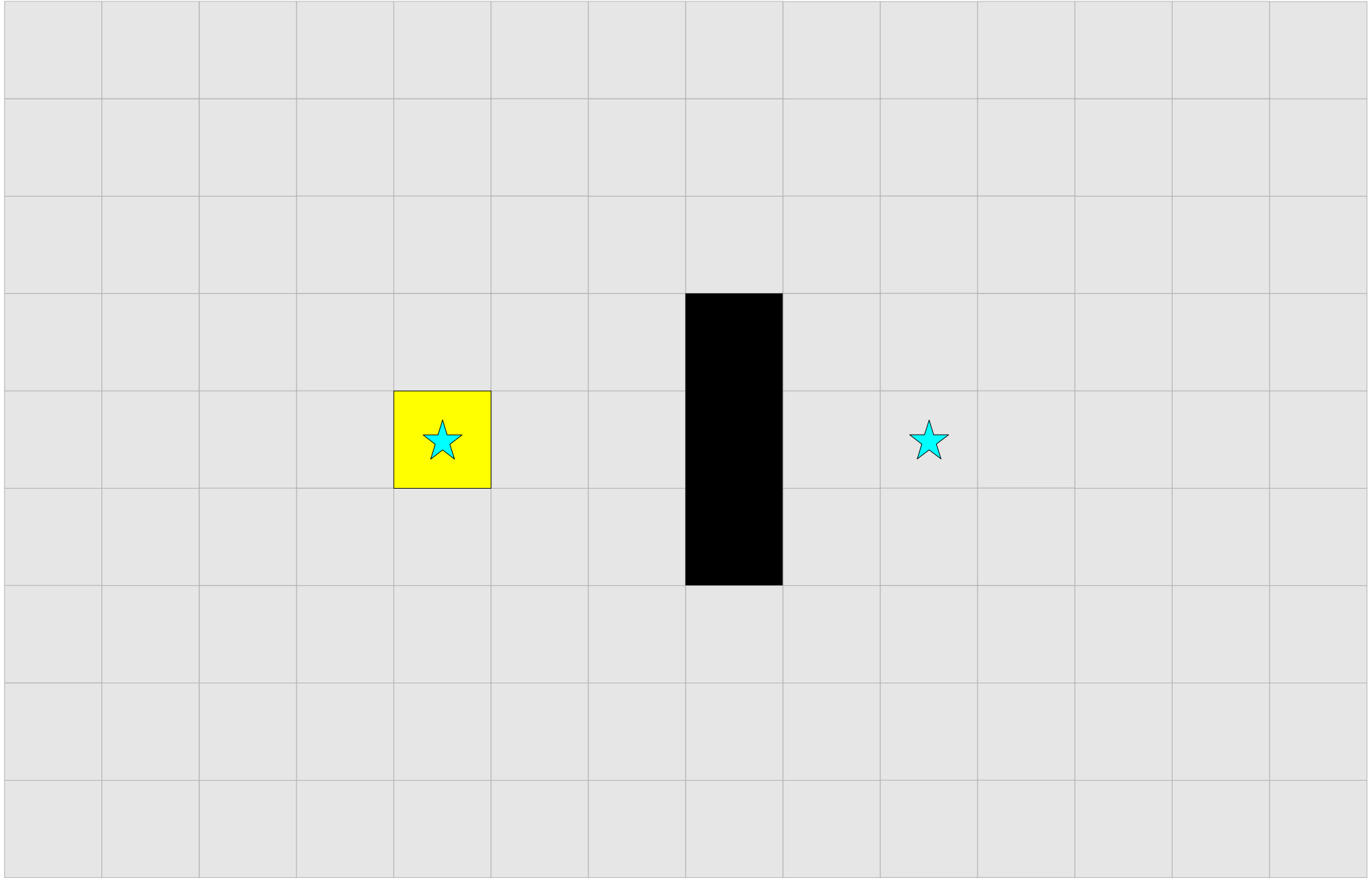
4

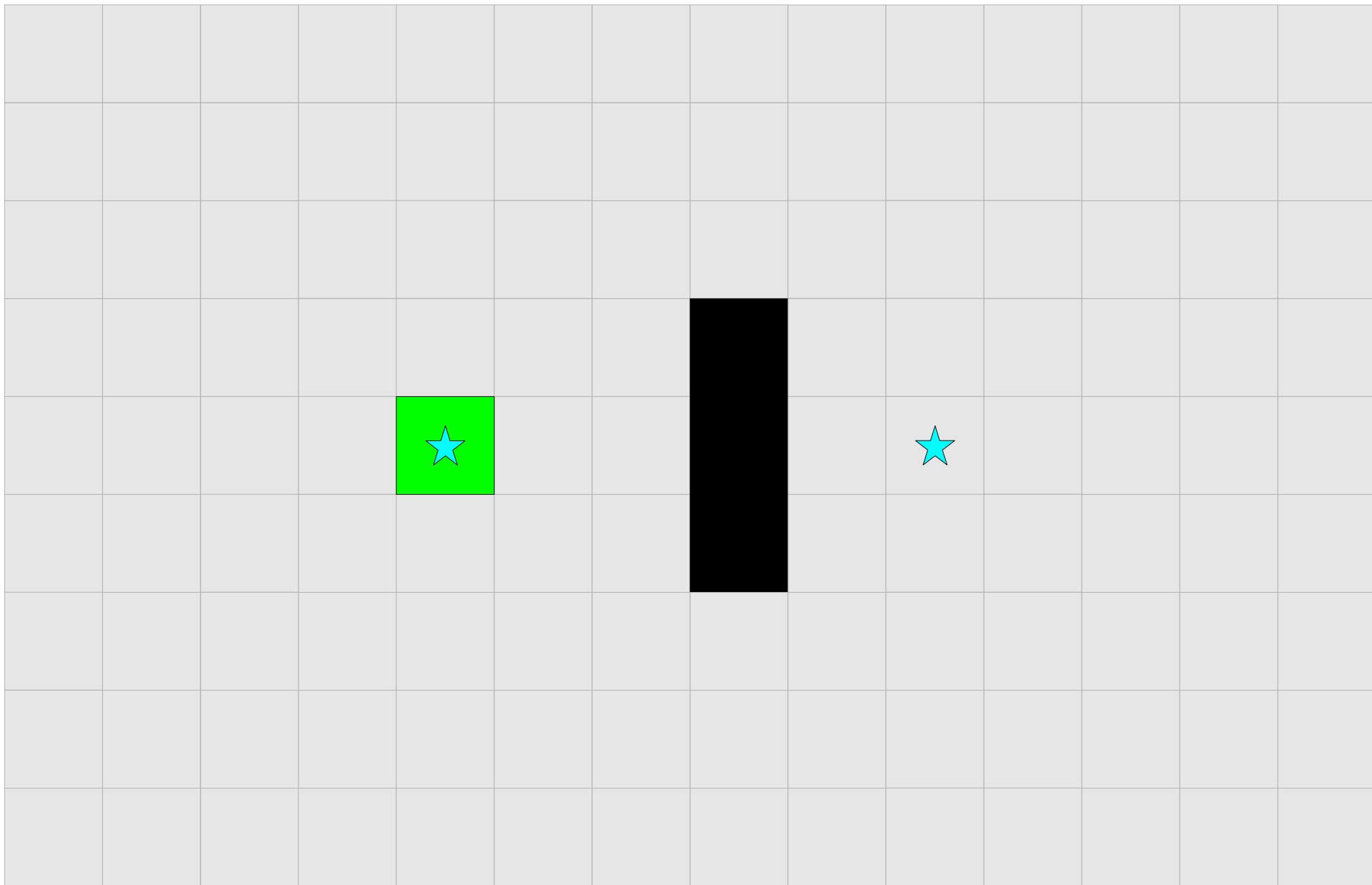


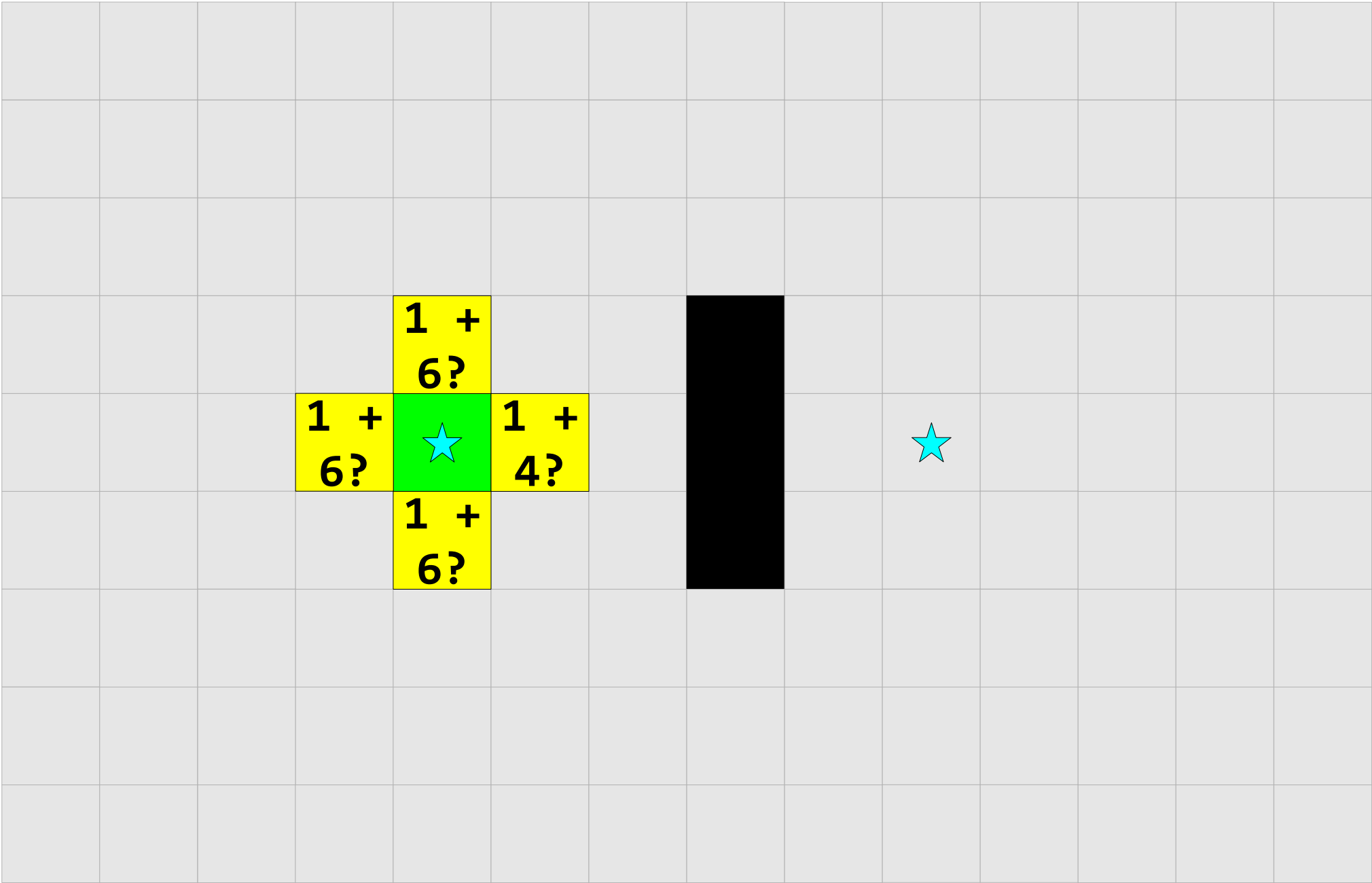


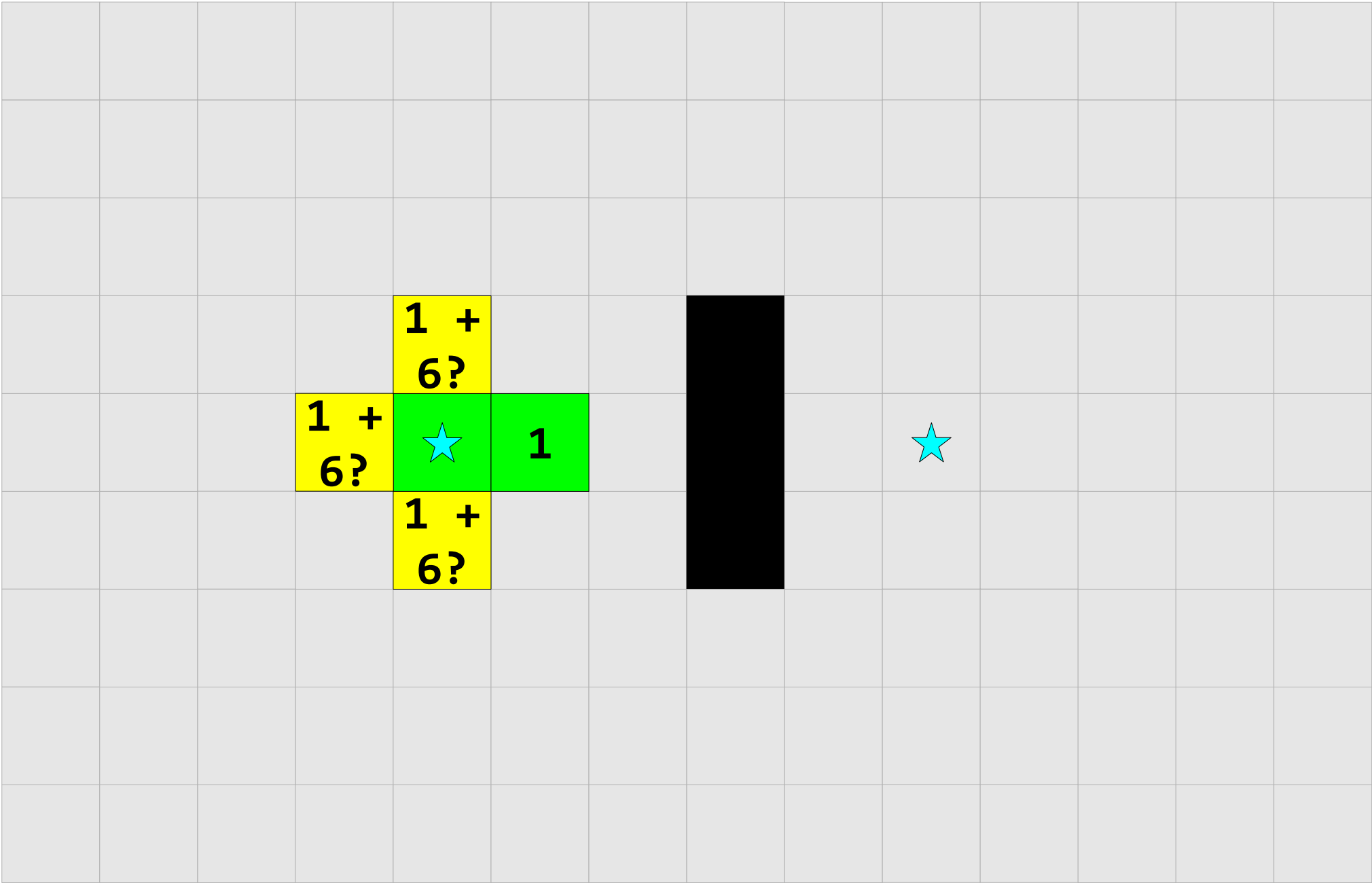




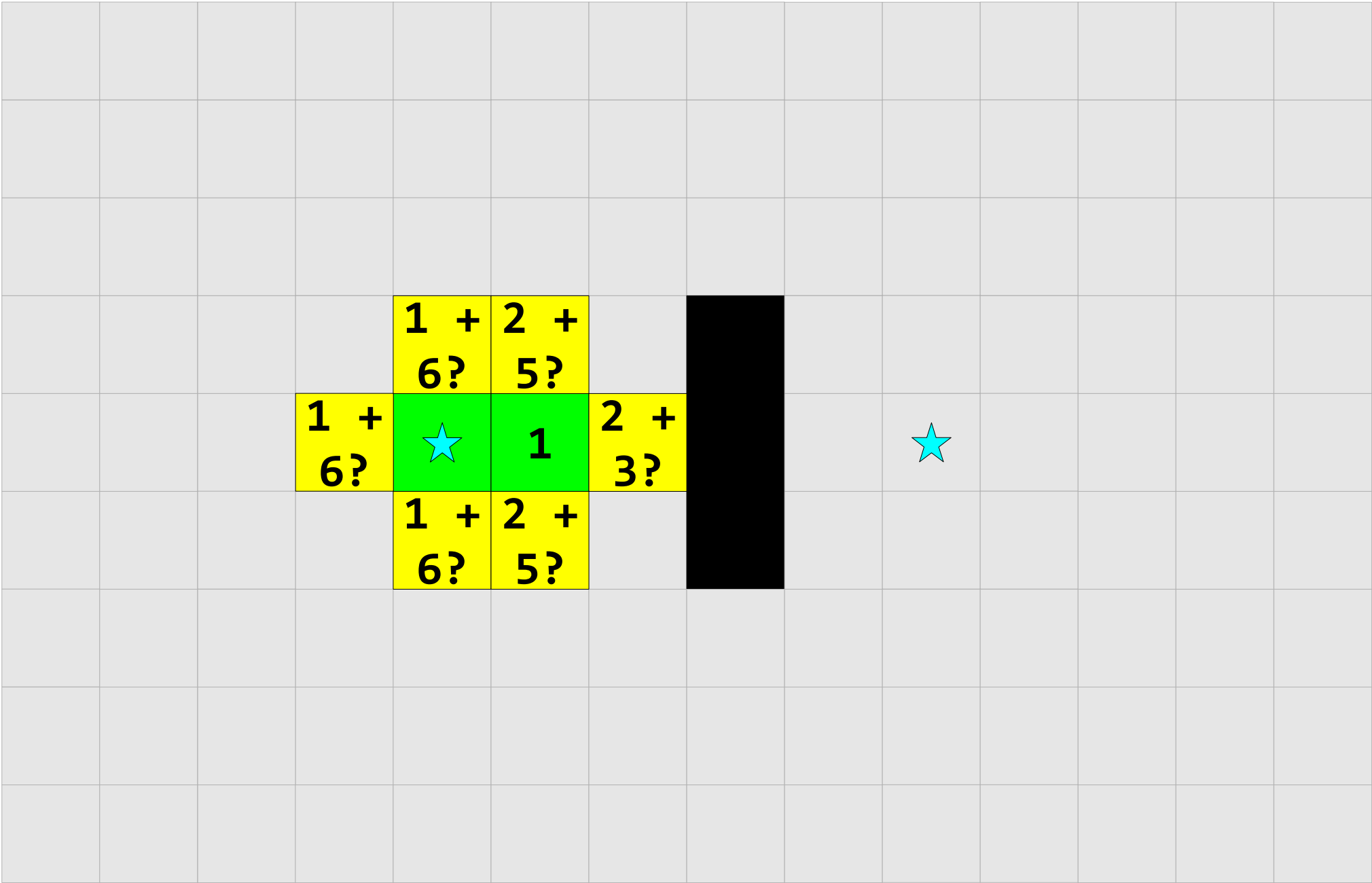


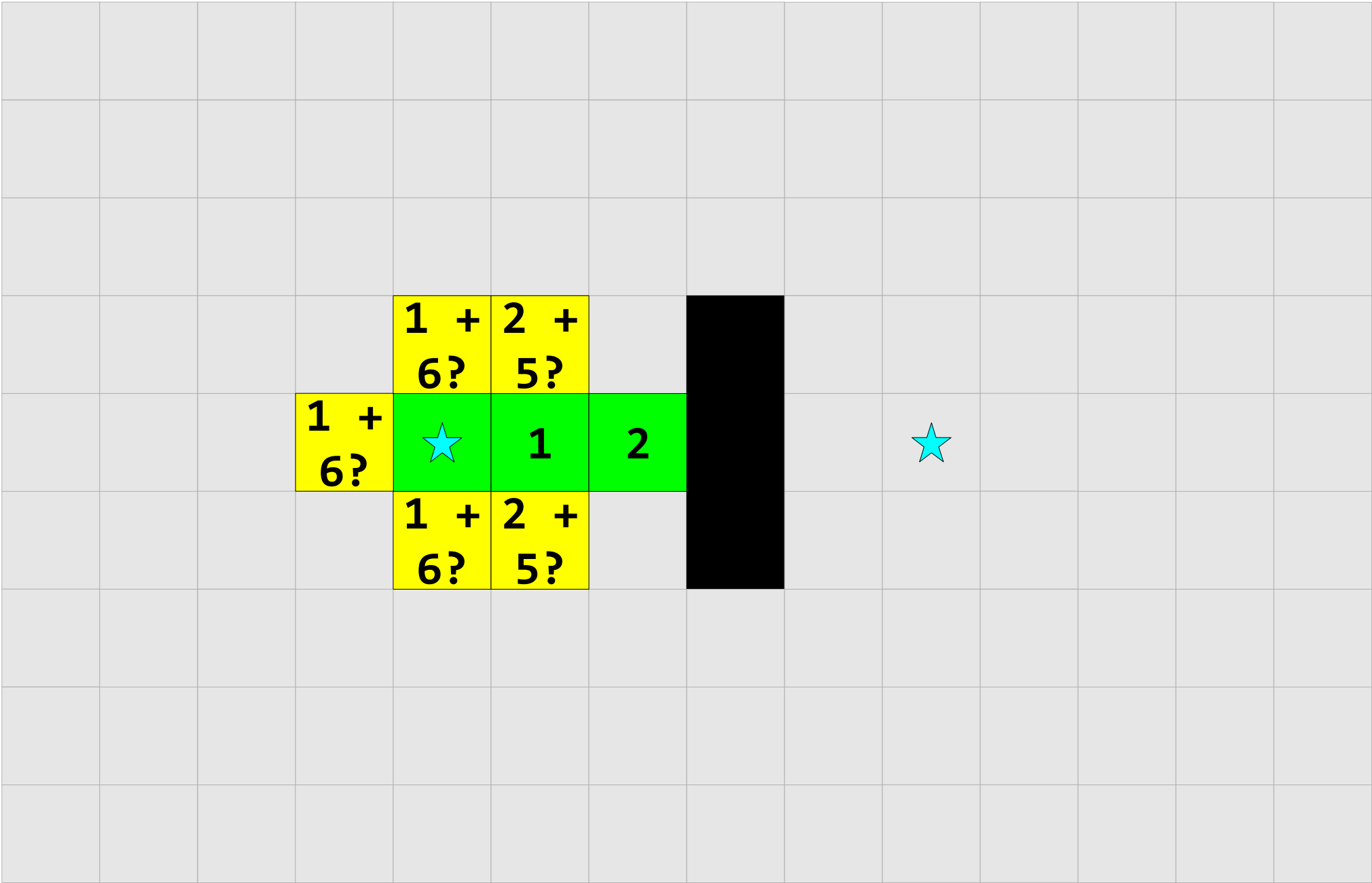


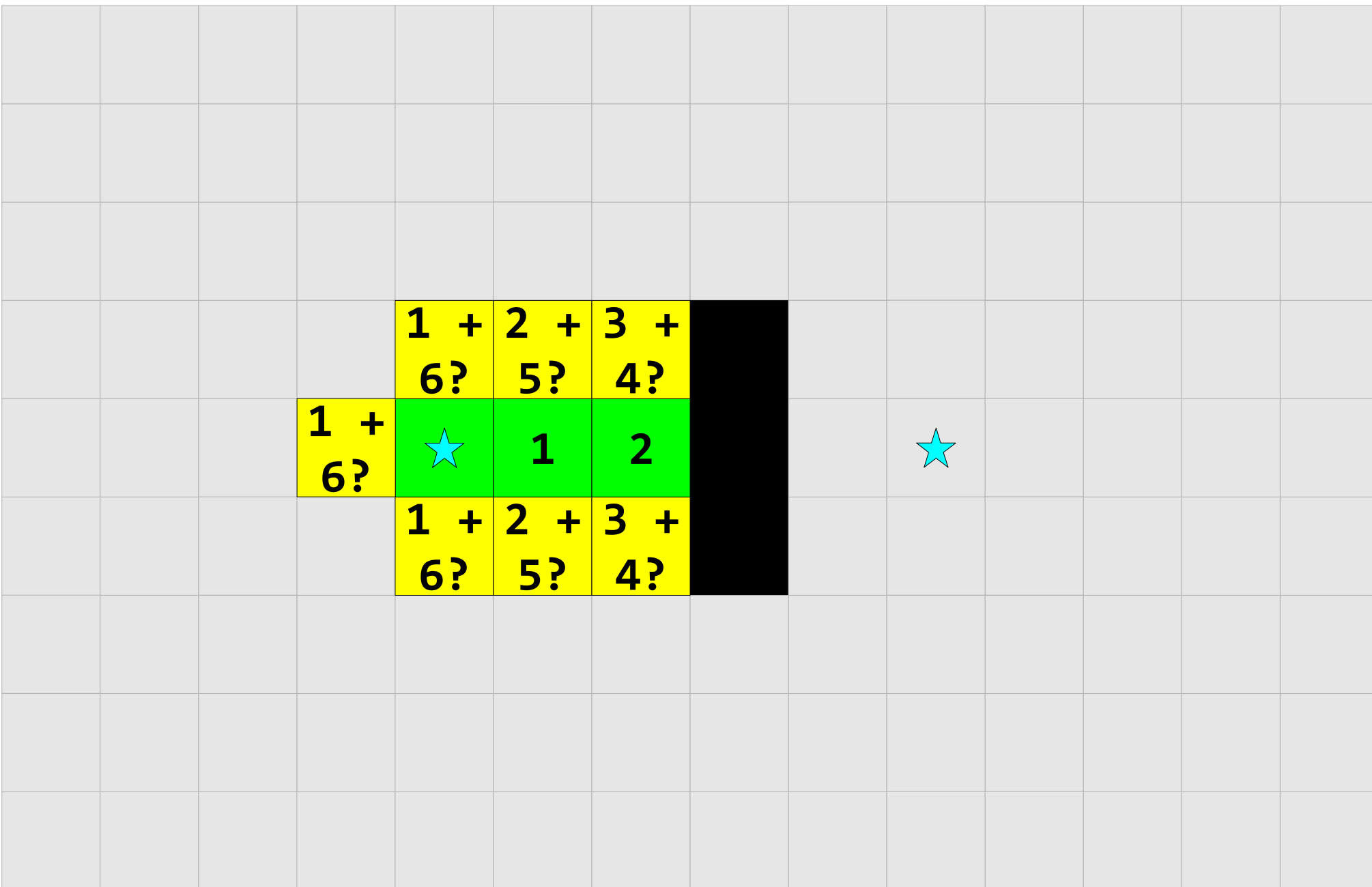


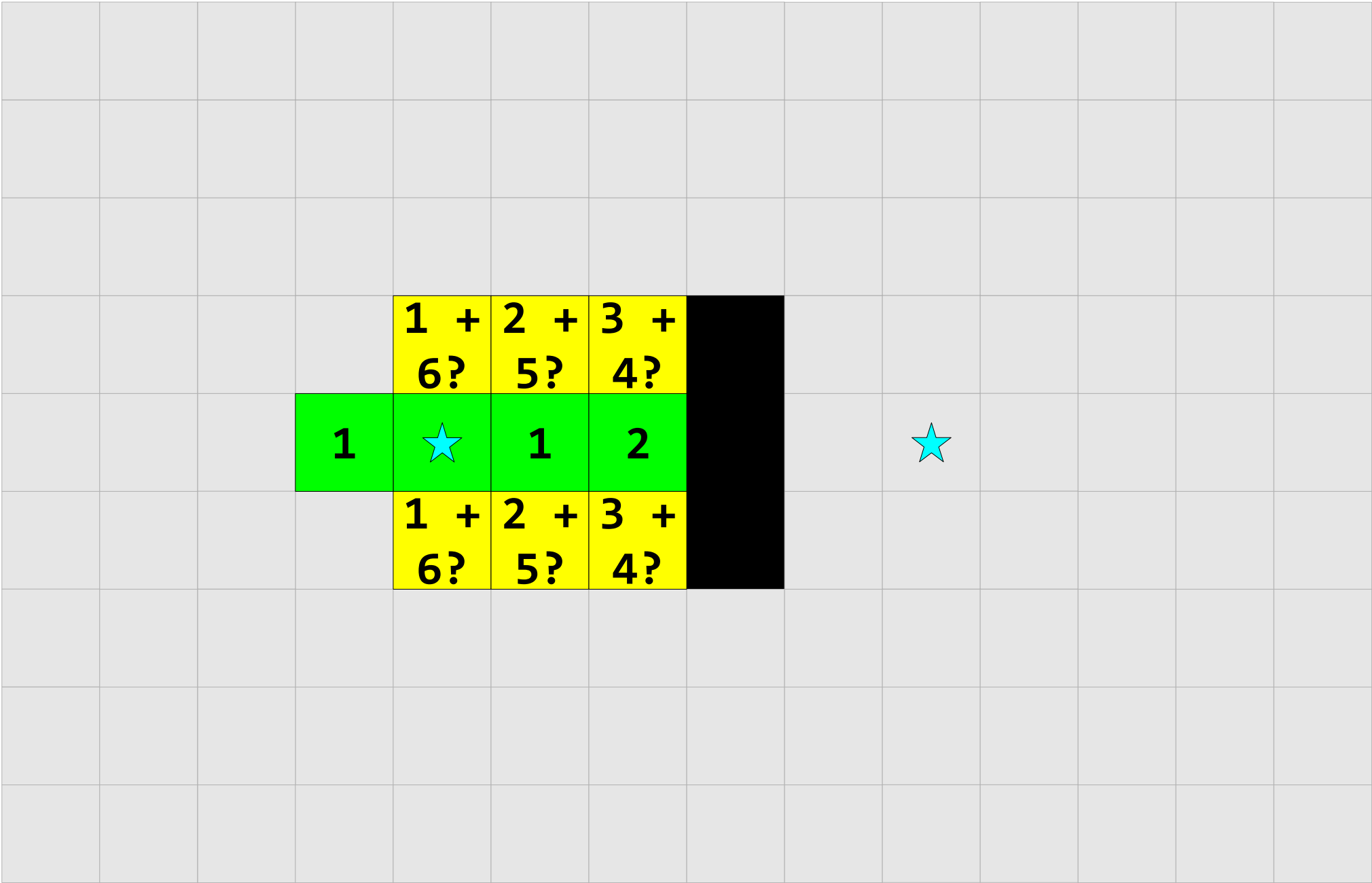










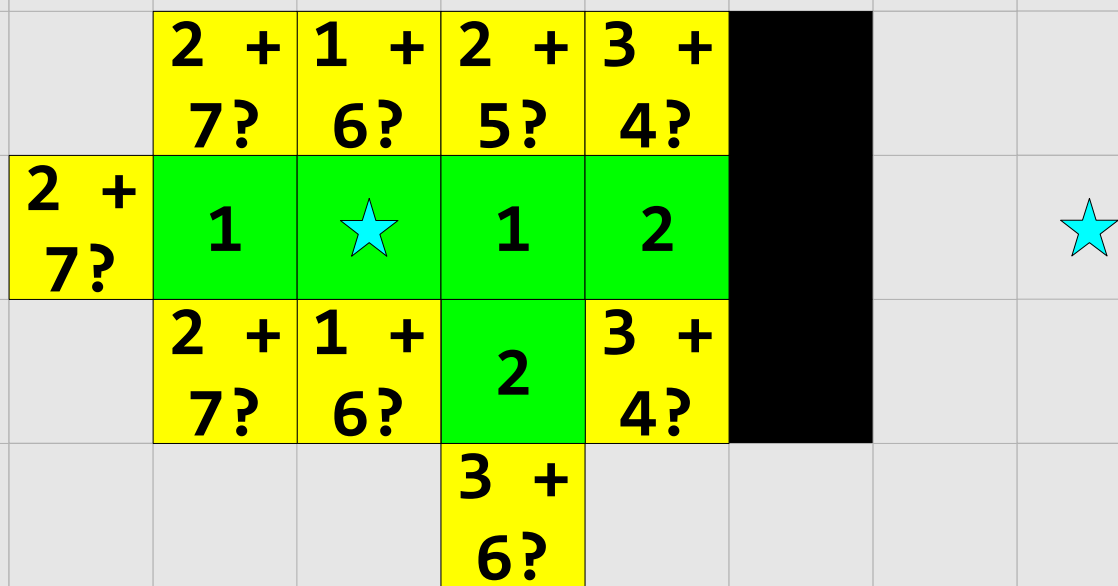


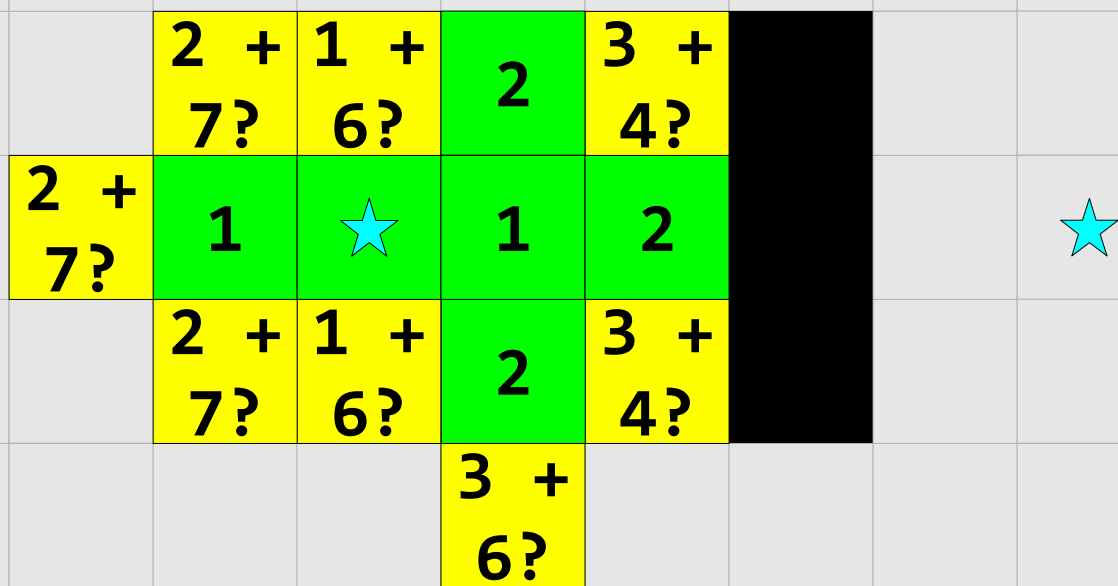
			2 + 7?	1 + 6?	2 + 5?	3 + 4?	
		2 + 7?	1	★	1	2	
			2 + 7?	1 + 6?	2 + 5?	3 + 4?	



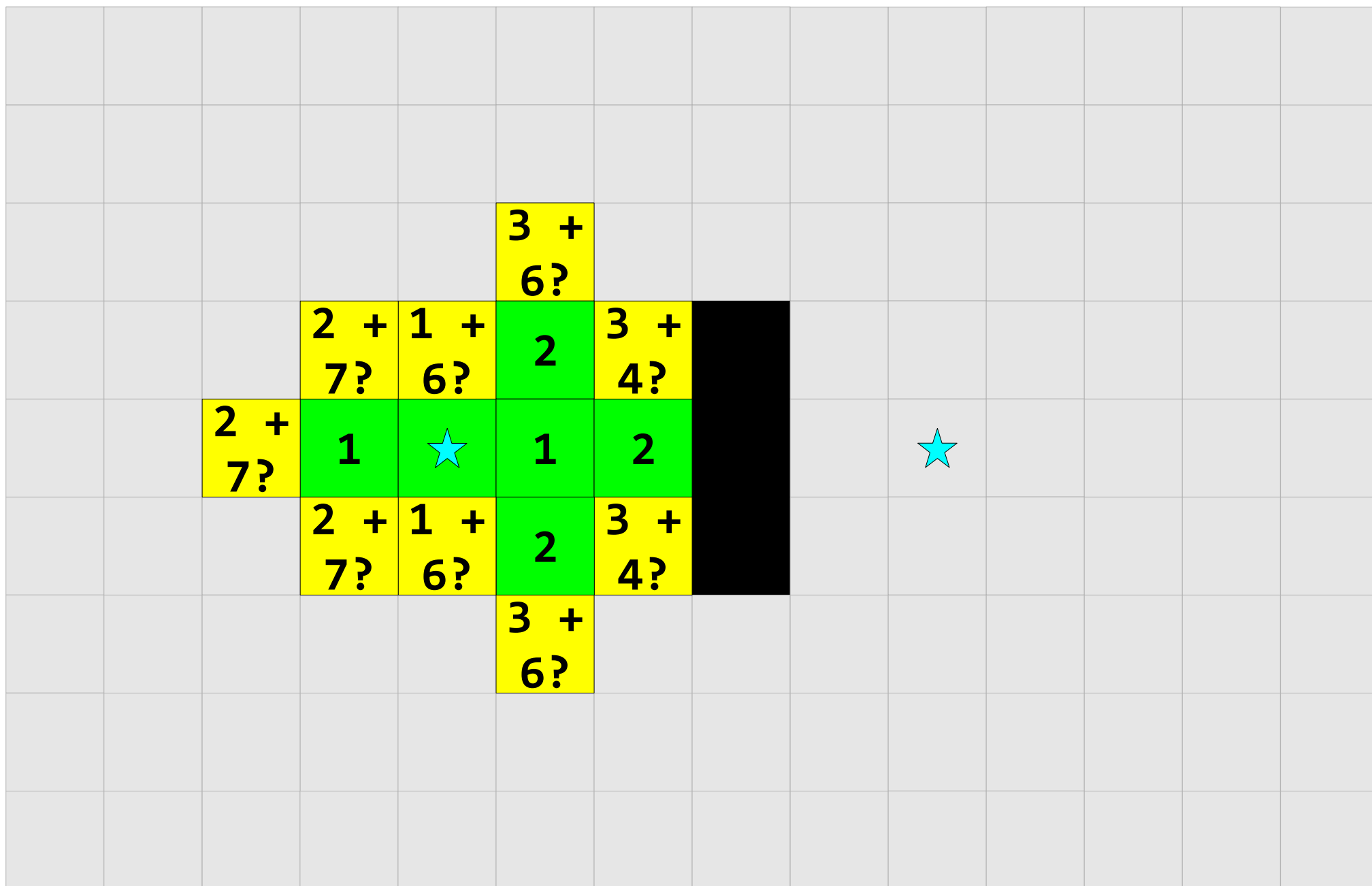
			2 + 7?	1 + 6?	2 + 5?	3 + 4?	
		2 + 7?	1	★	1	2	
			2 + 7?	1 + 6?	2	3 + 4?	

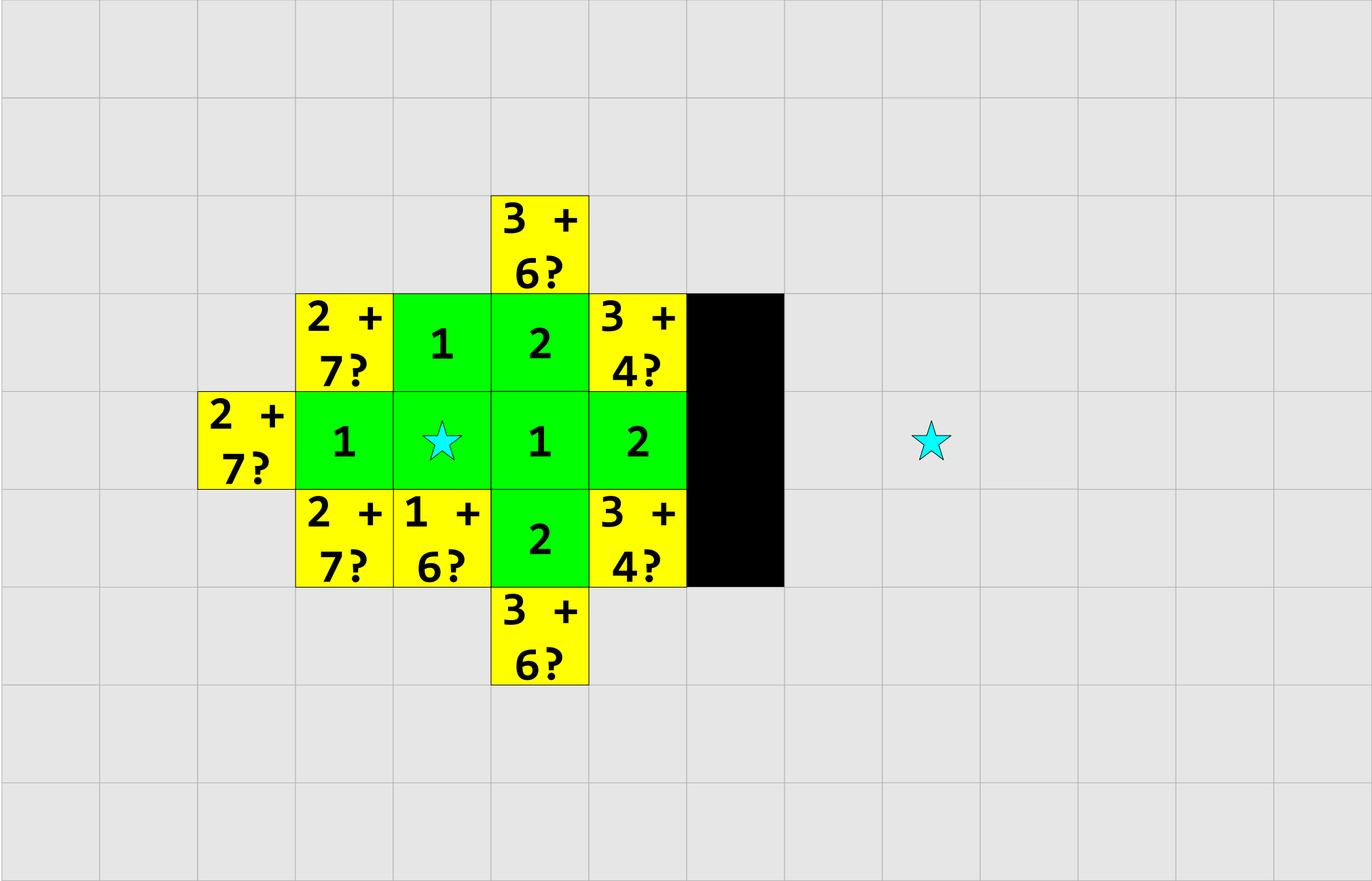


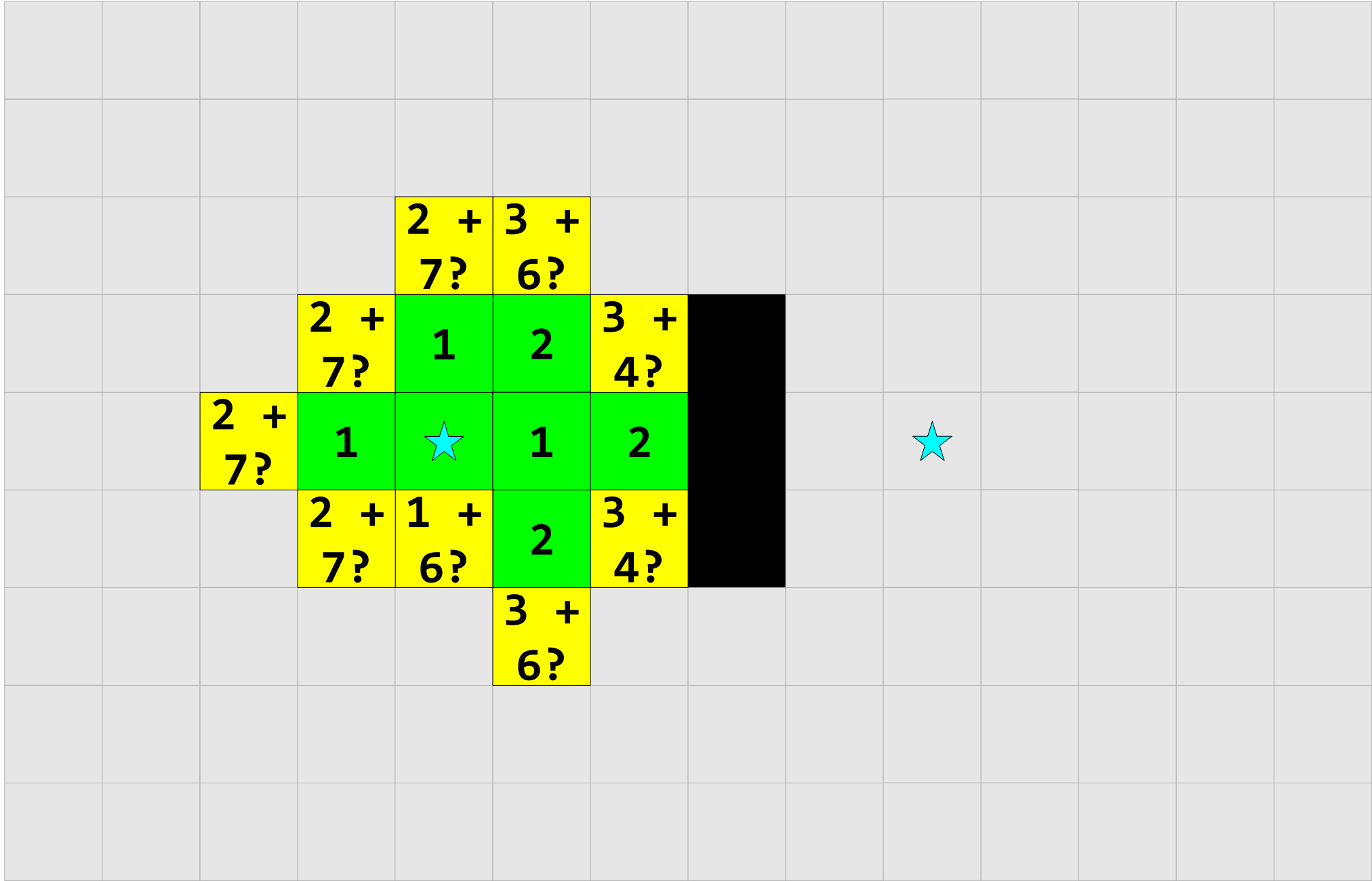


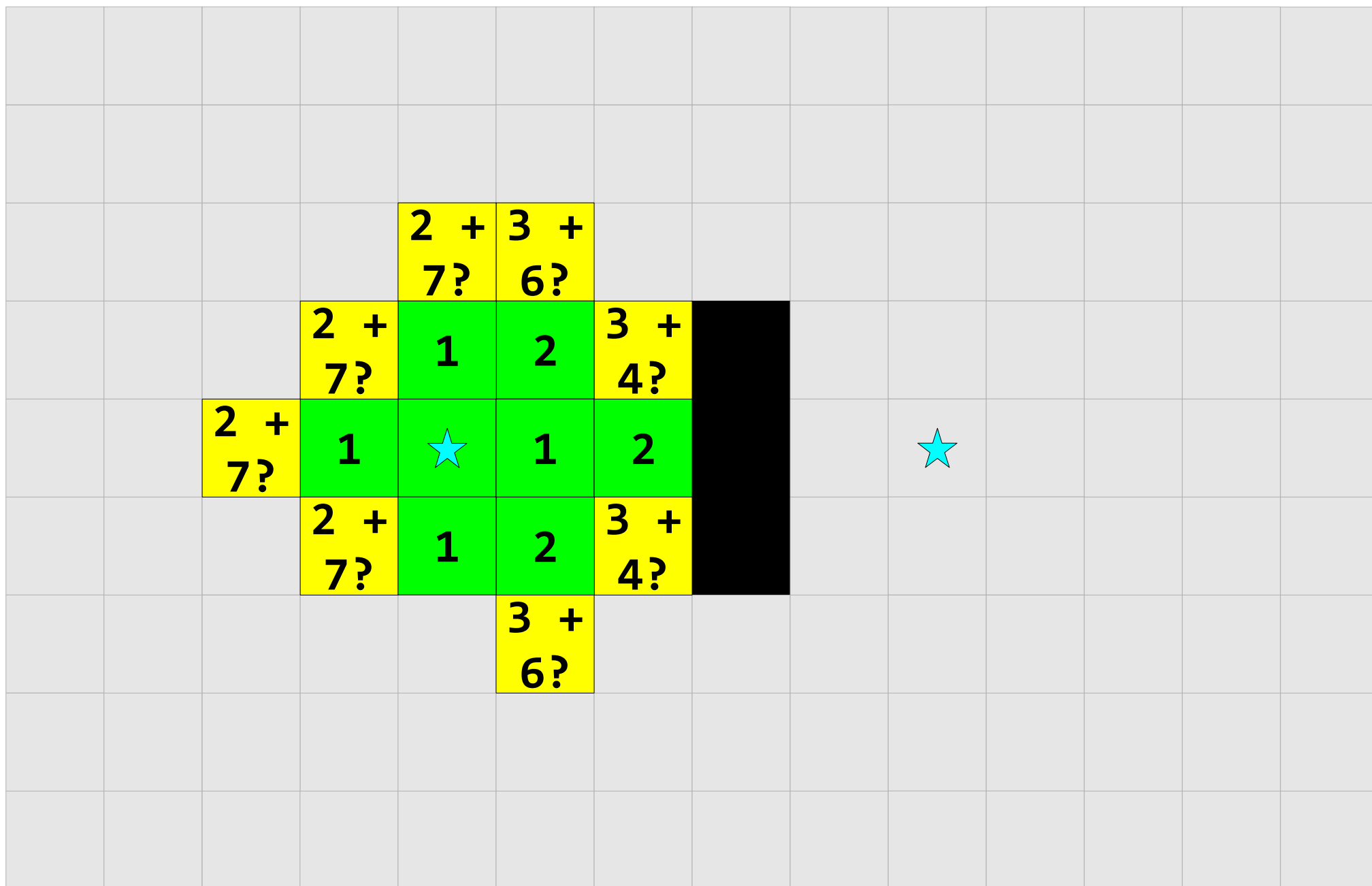


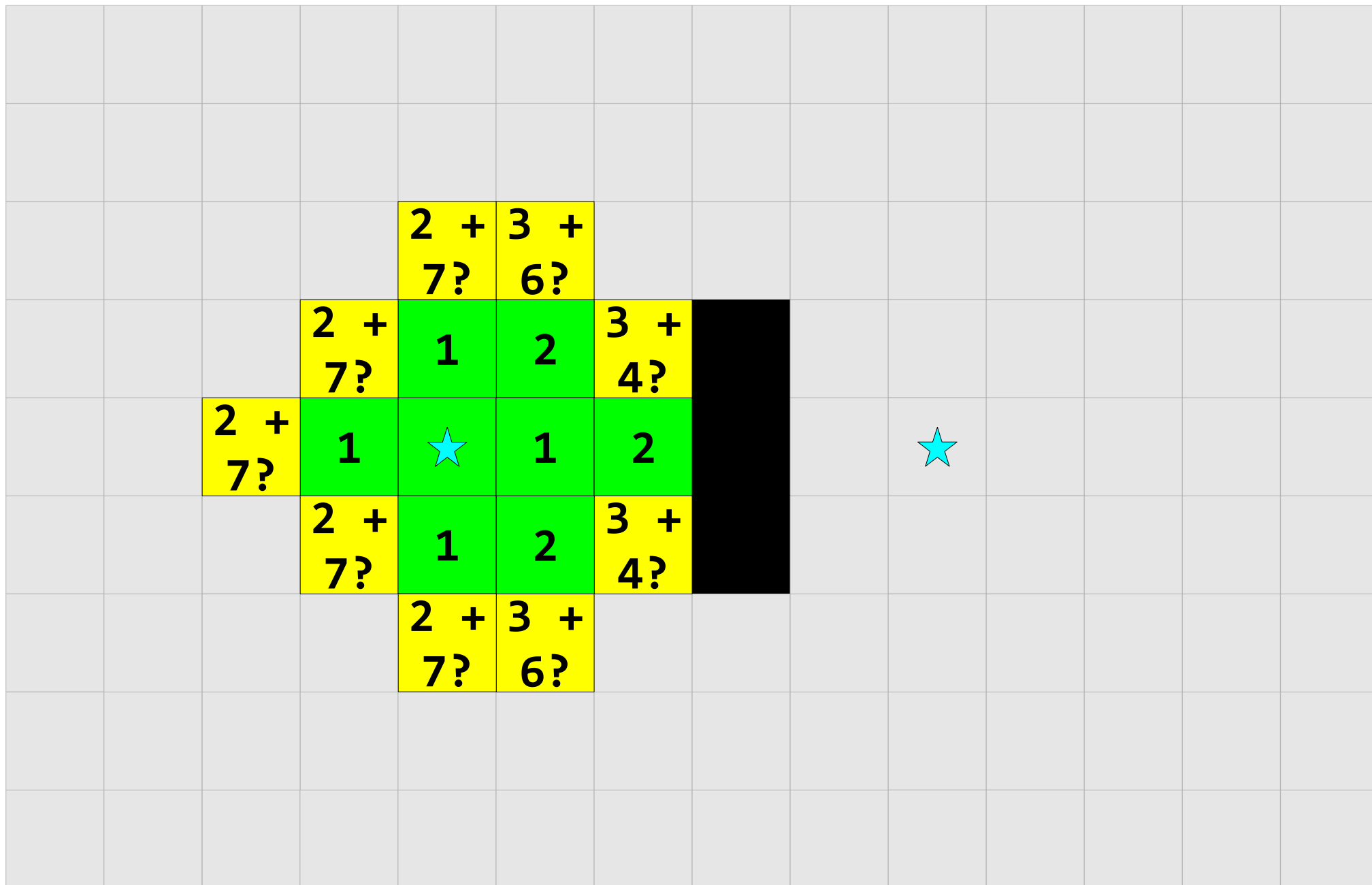


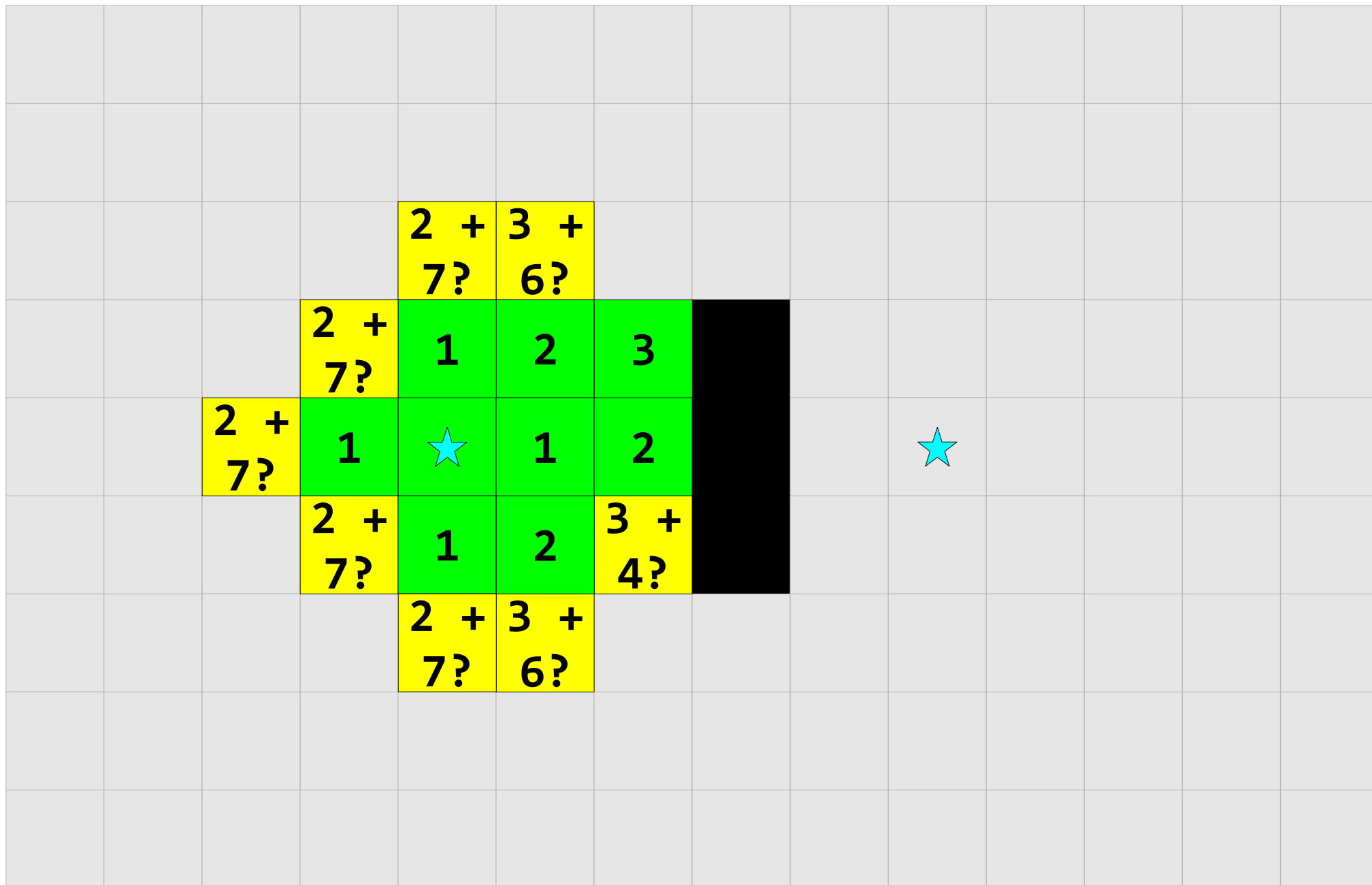


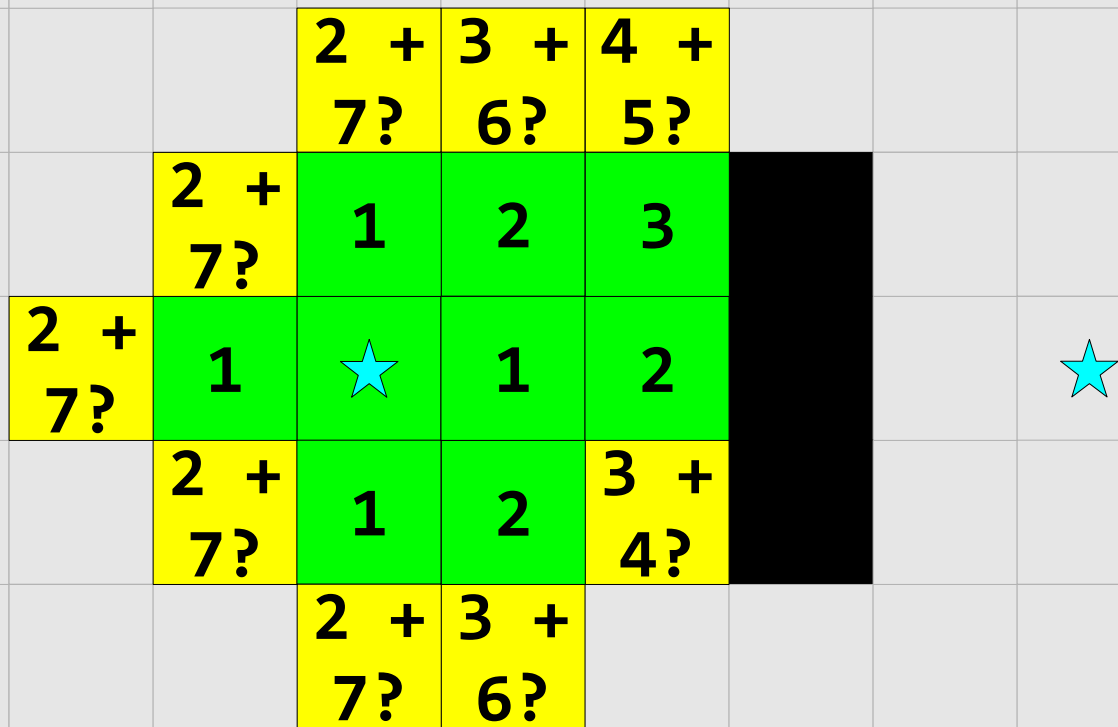


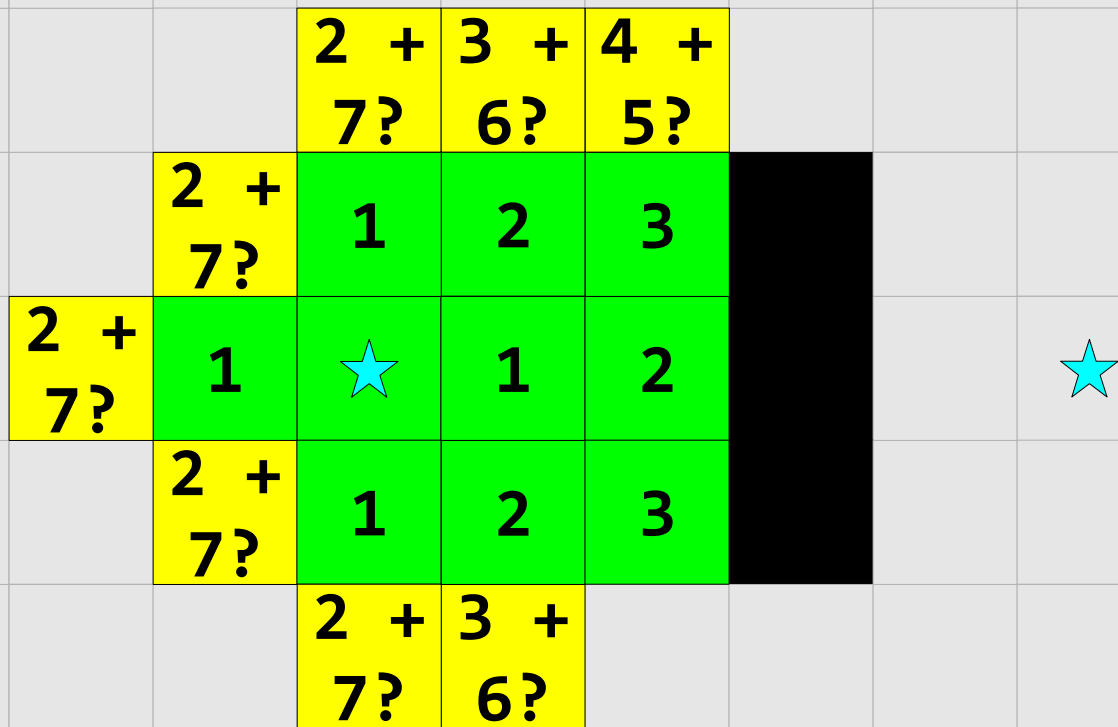




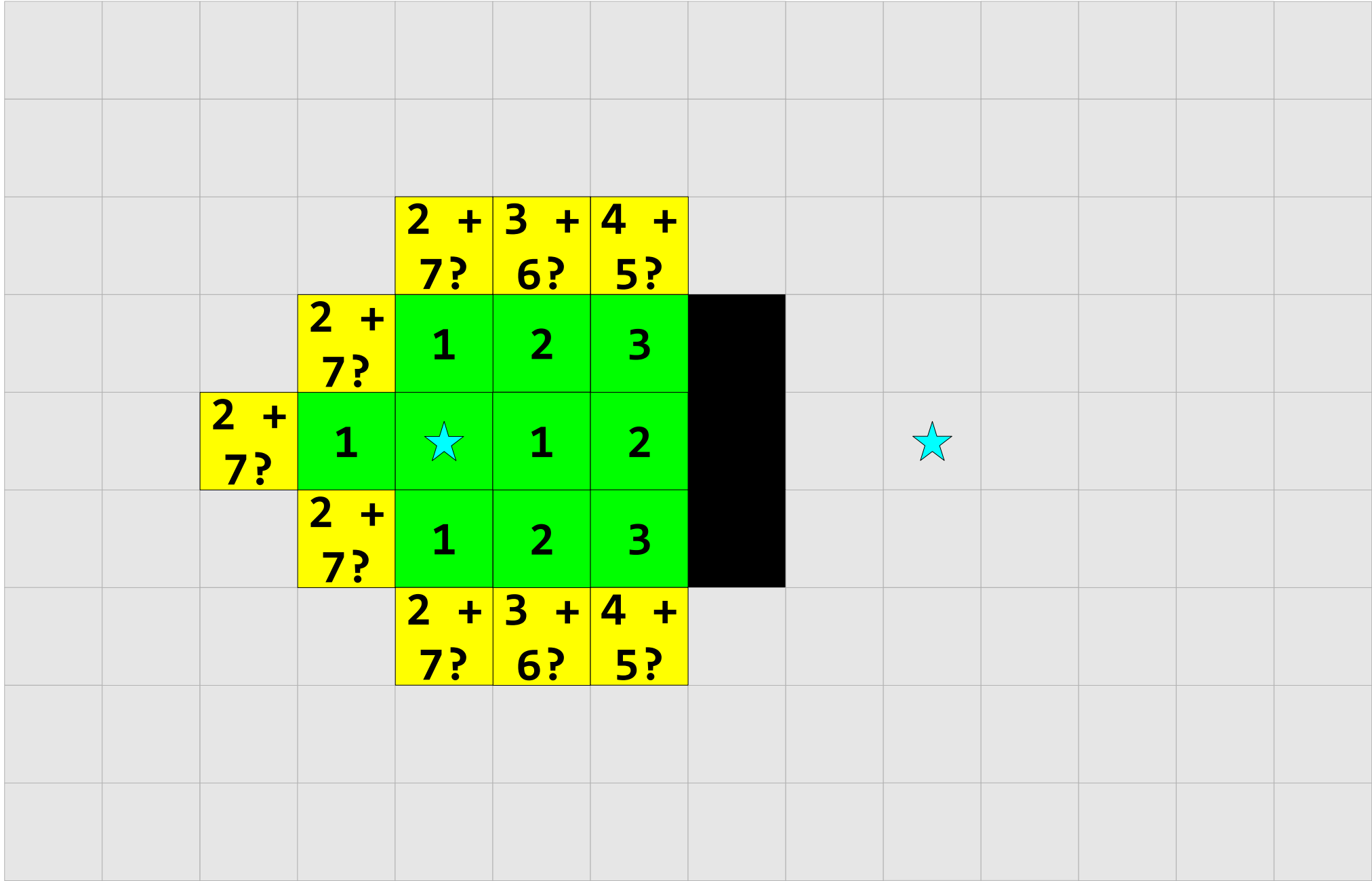


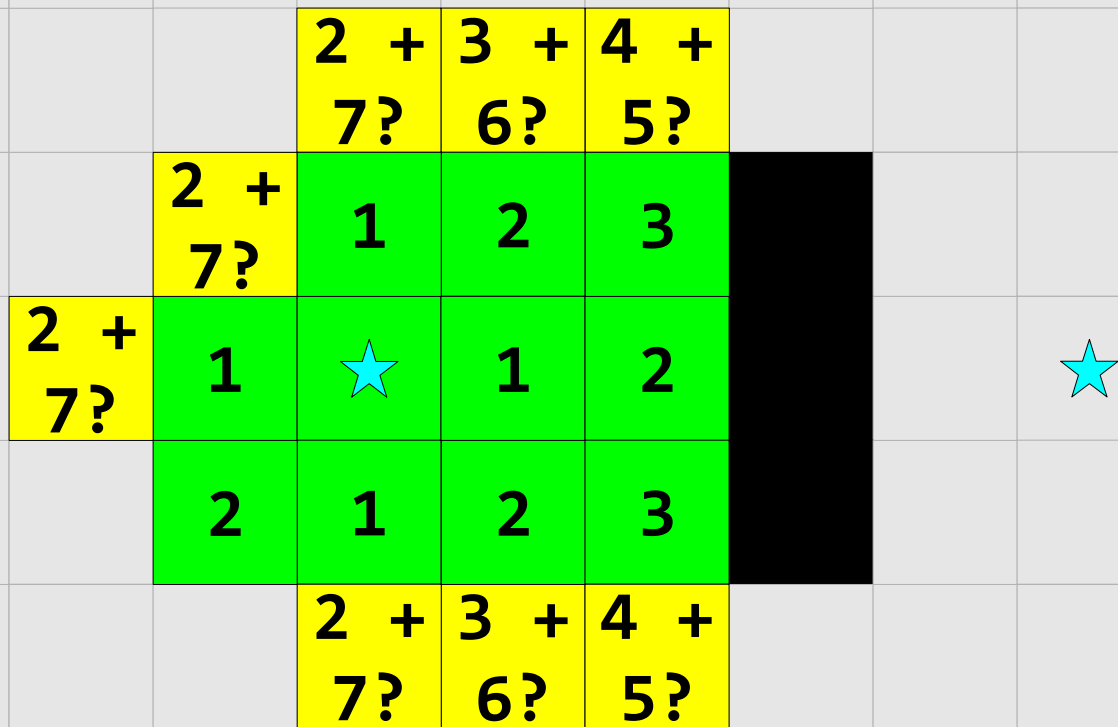




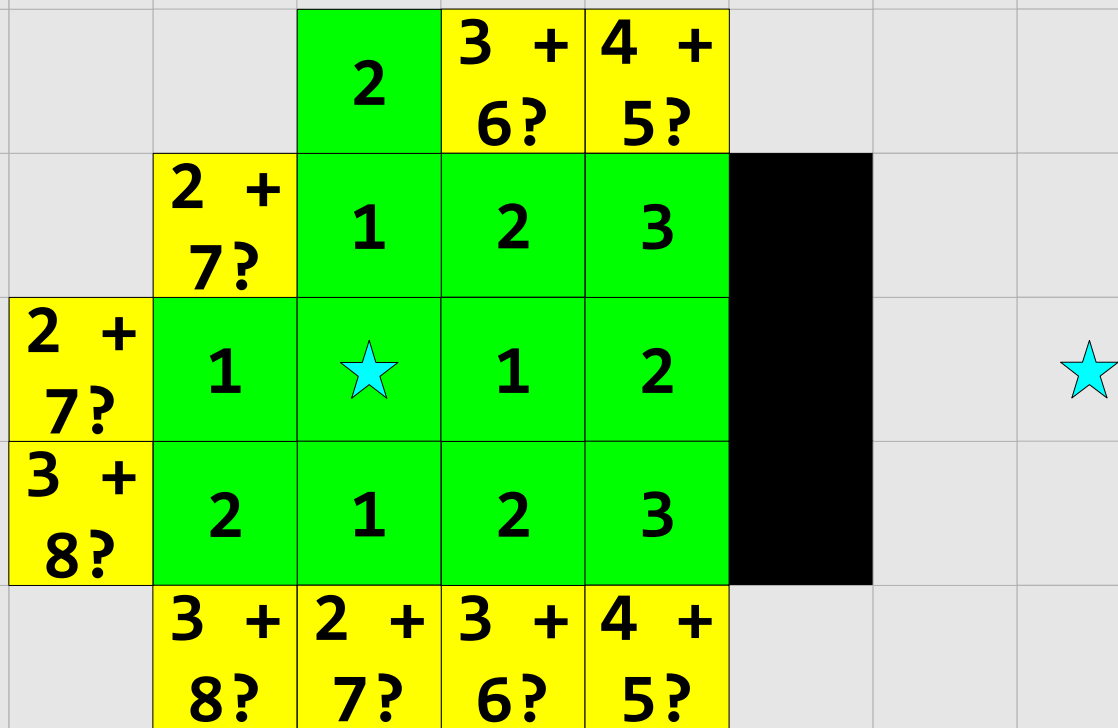


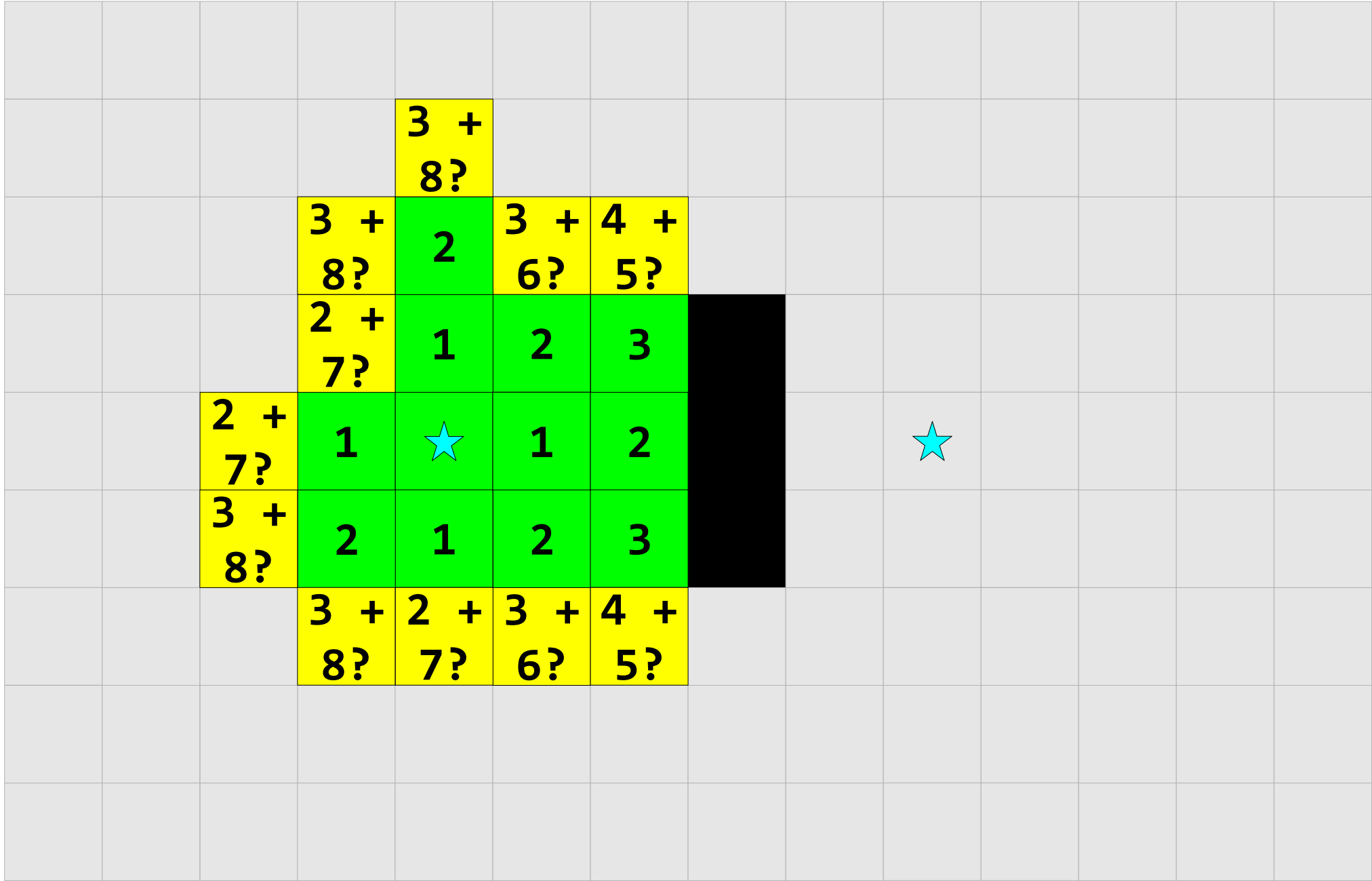


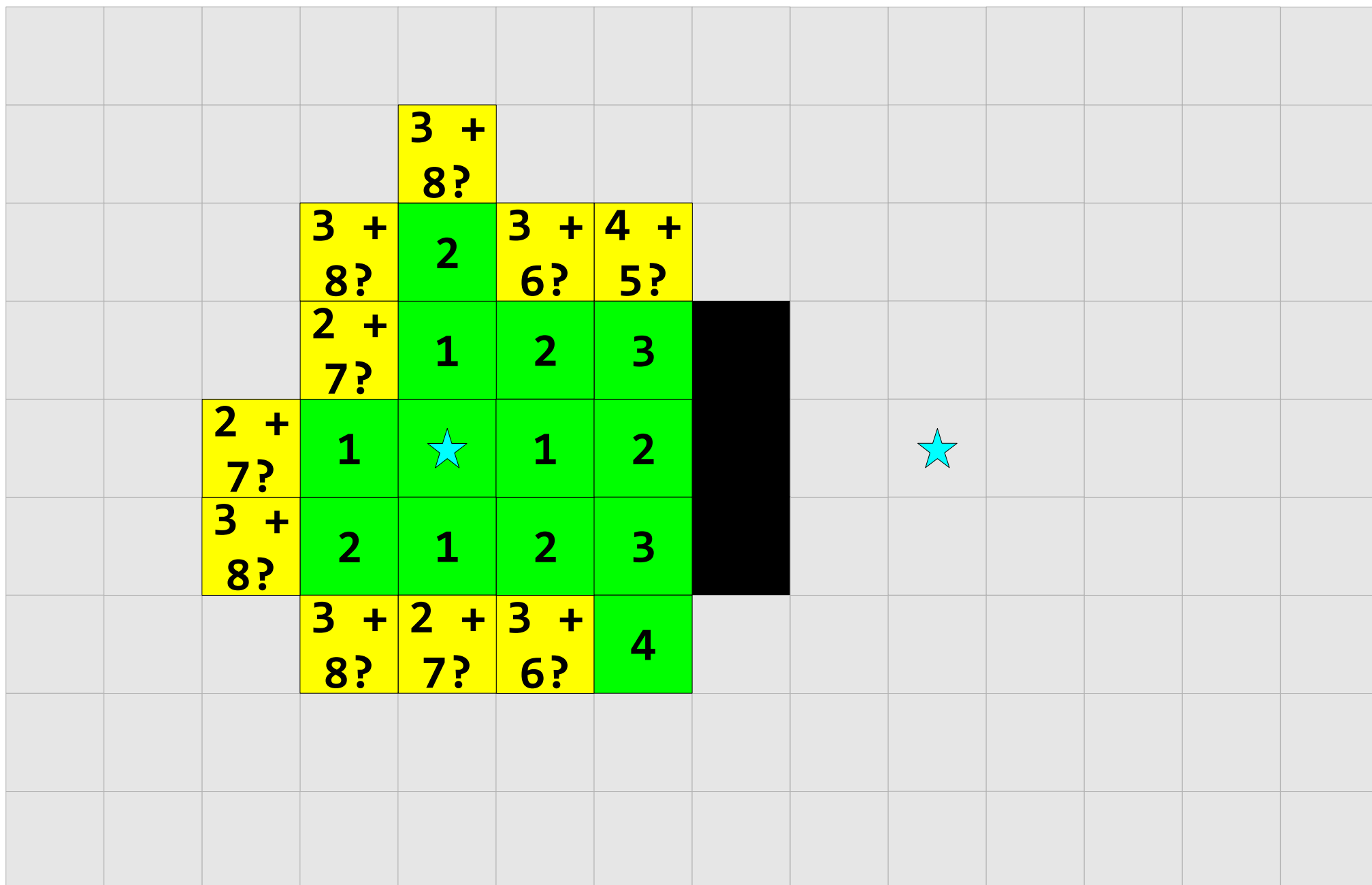


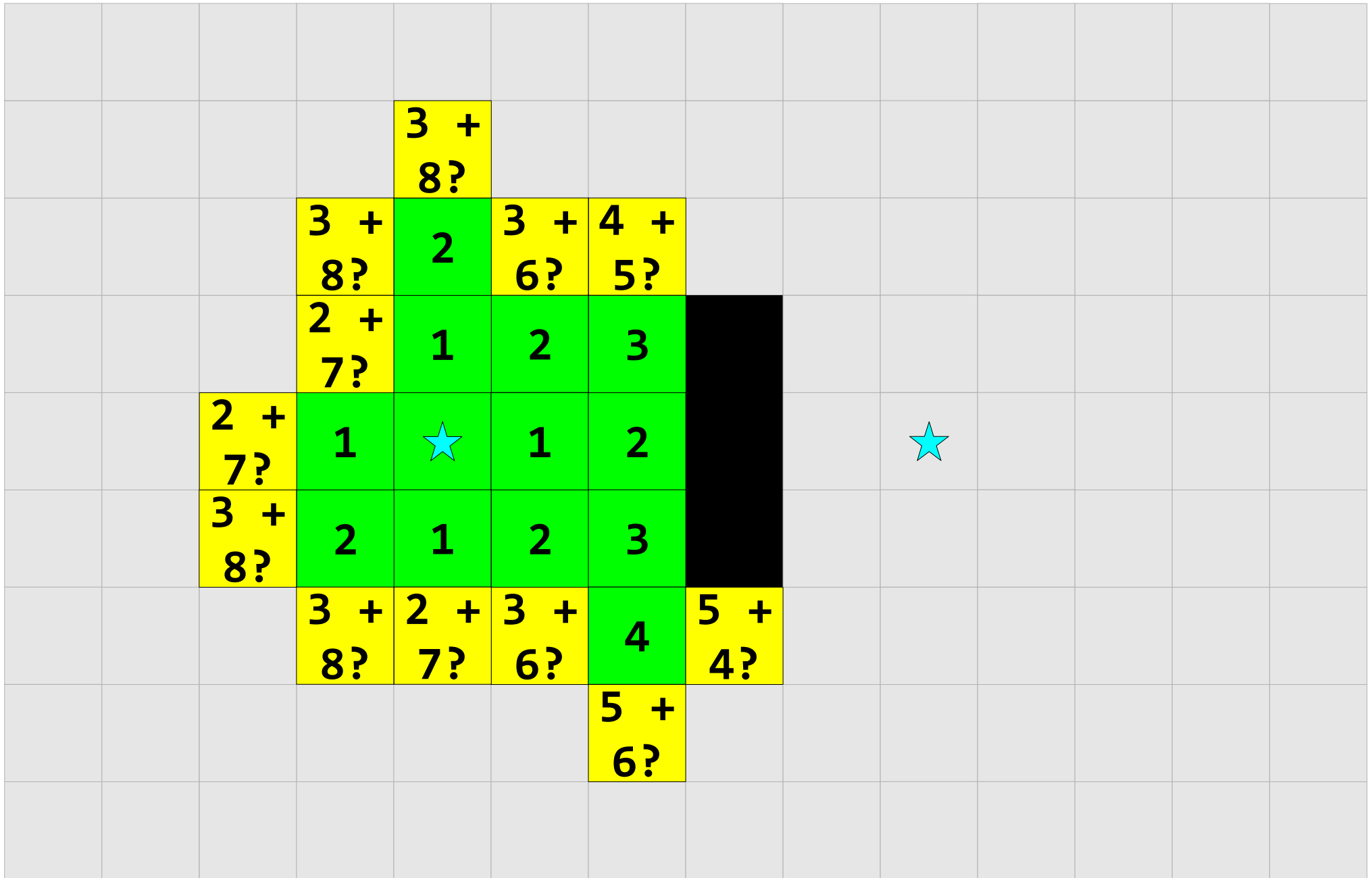


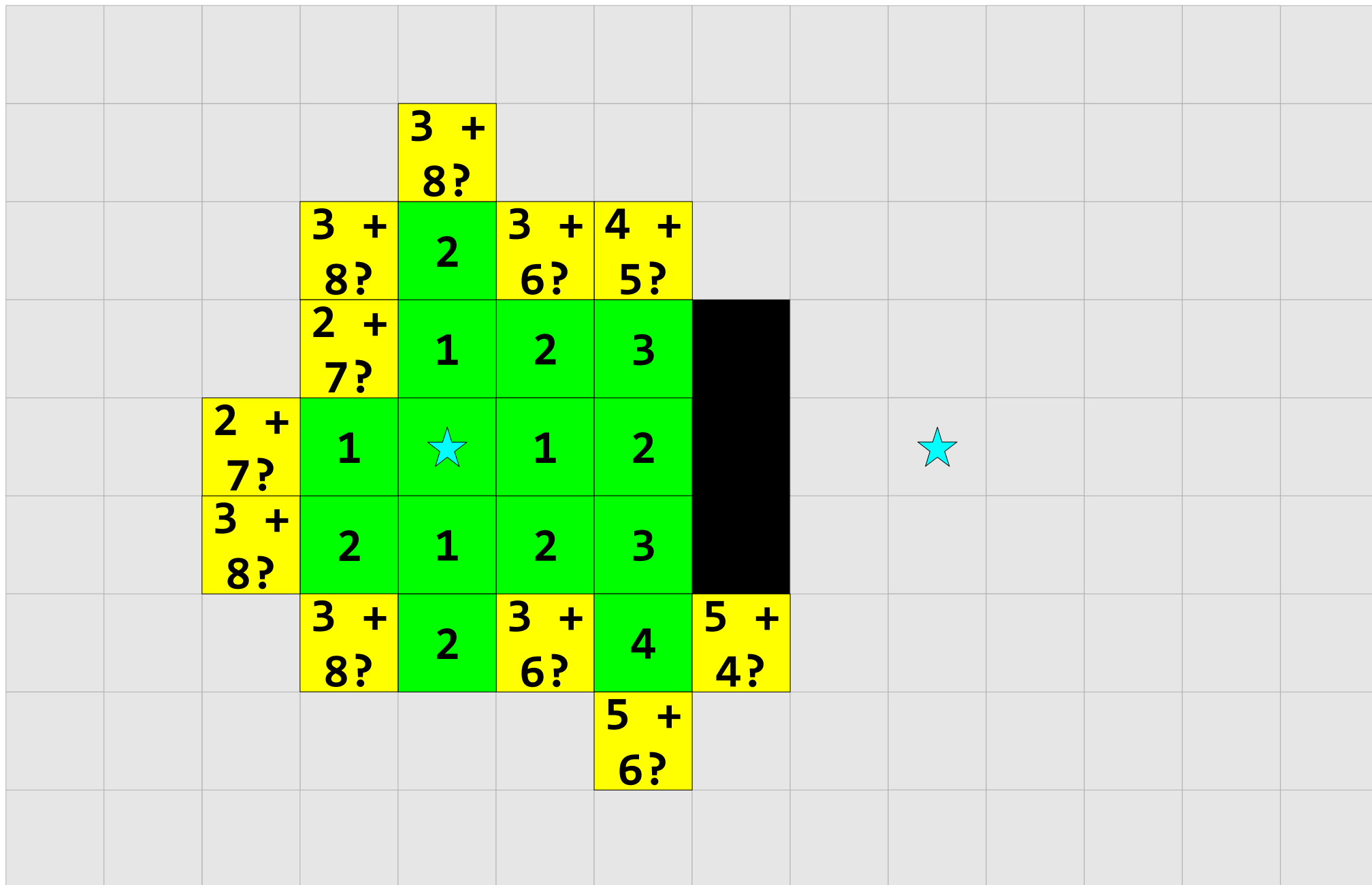
				2 + 7?	3 + 6?	4 + 5?							
			2 + 7?	1	2	3							
		2 + 7?	1	★	1	2				★			
		3 + 8?	2	1	2	3							
			3 + 8?	2 + 7?	3 + 6?	4 + 5?							



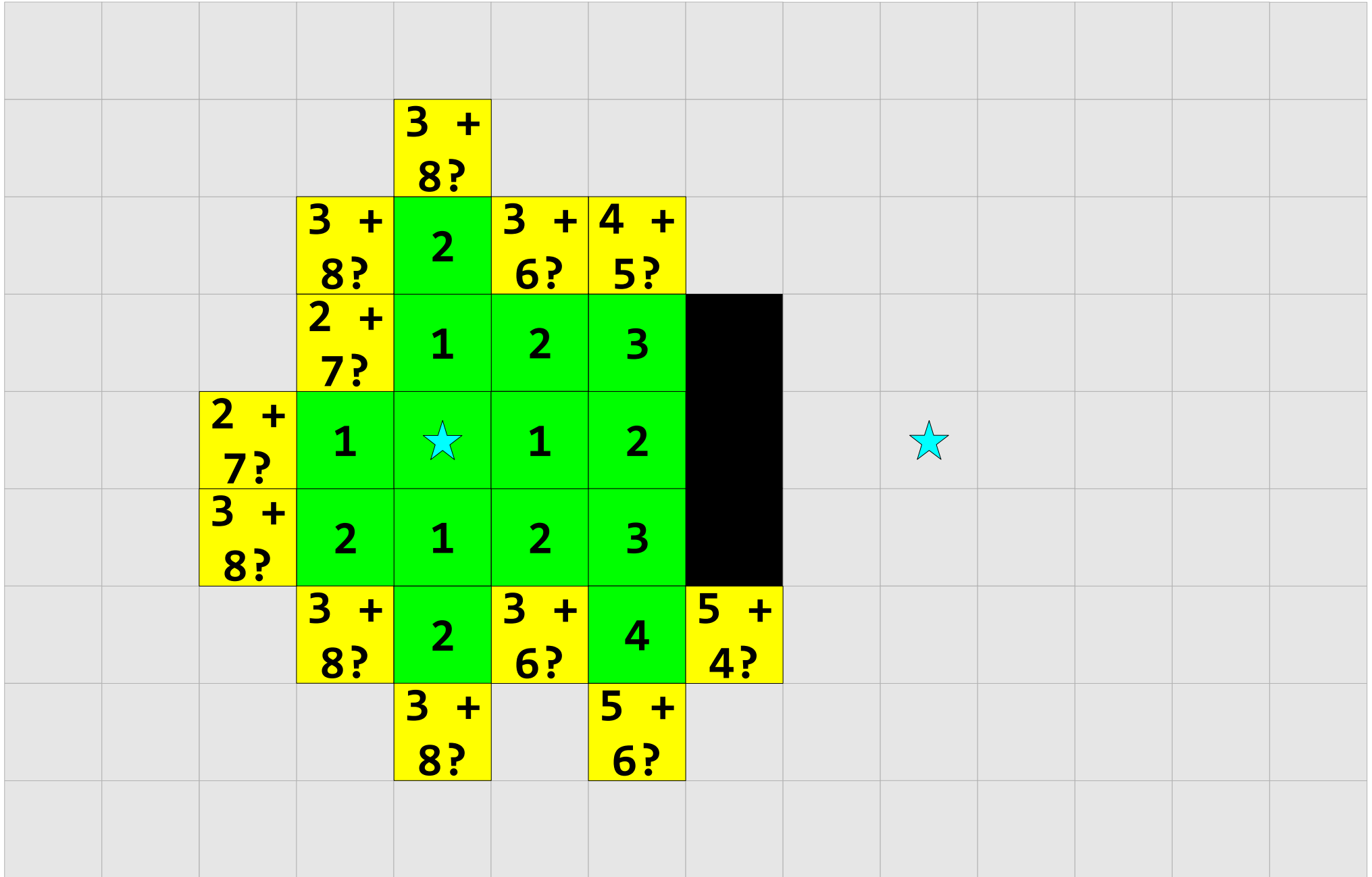




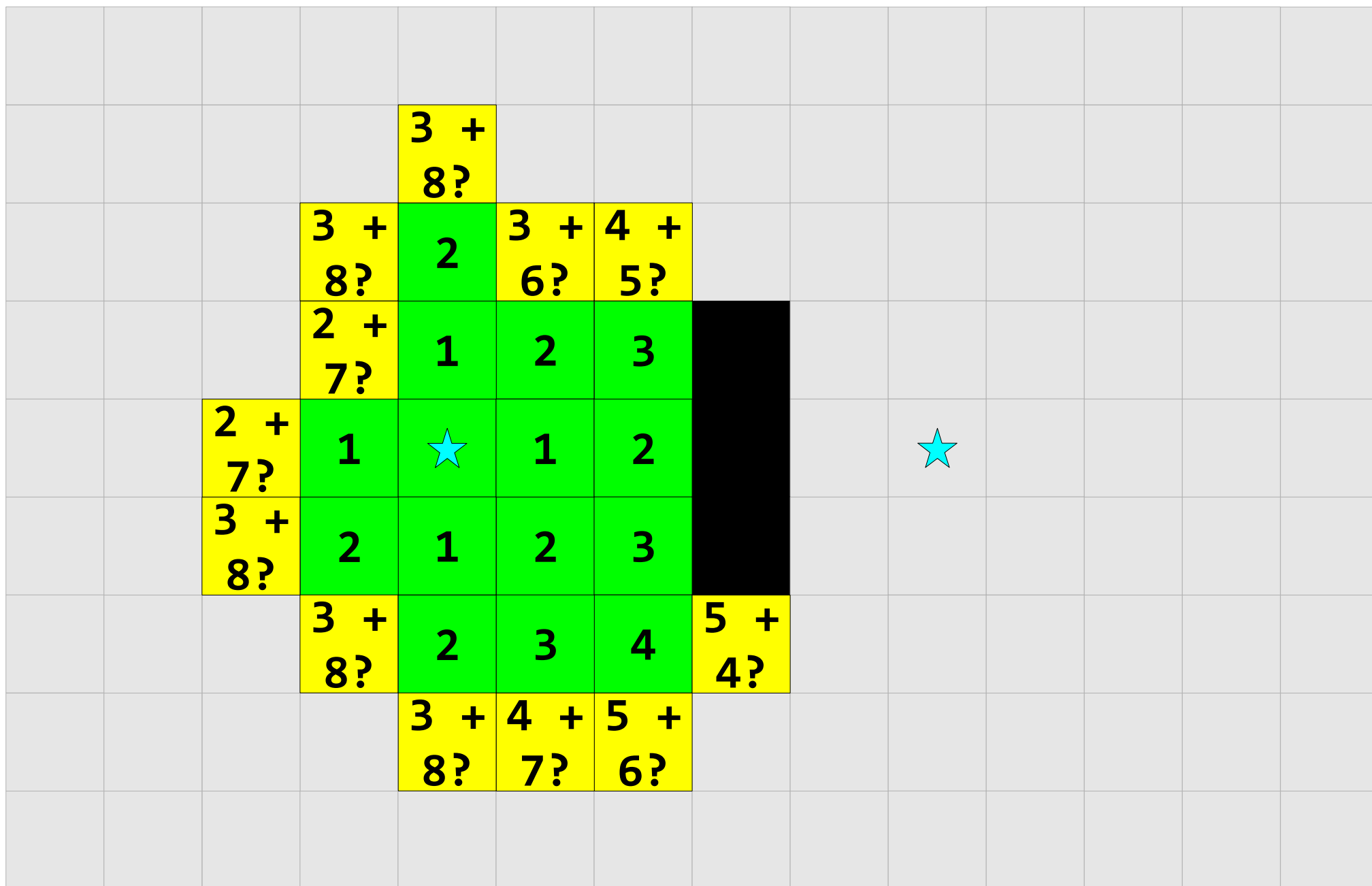


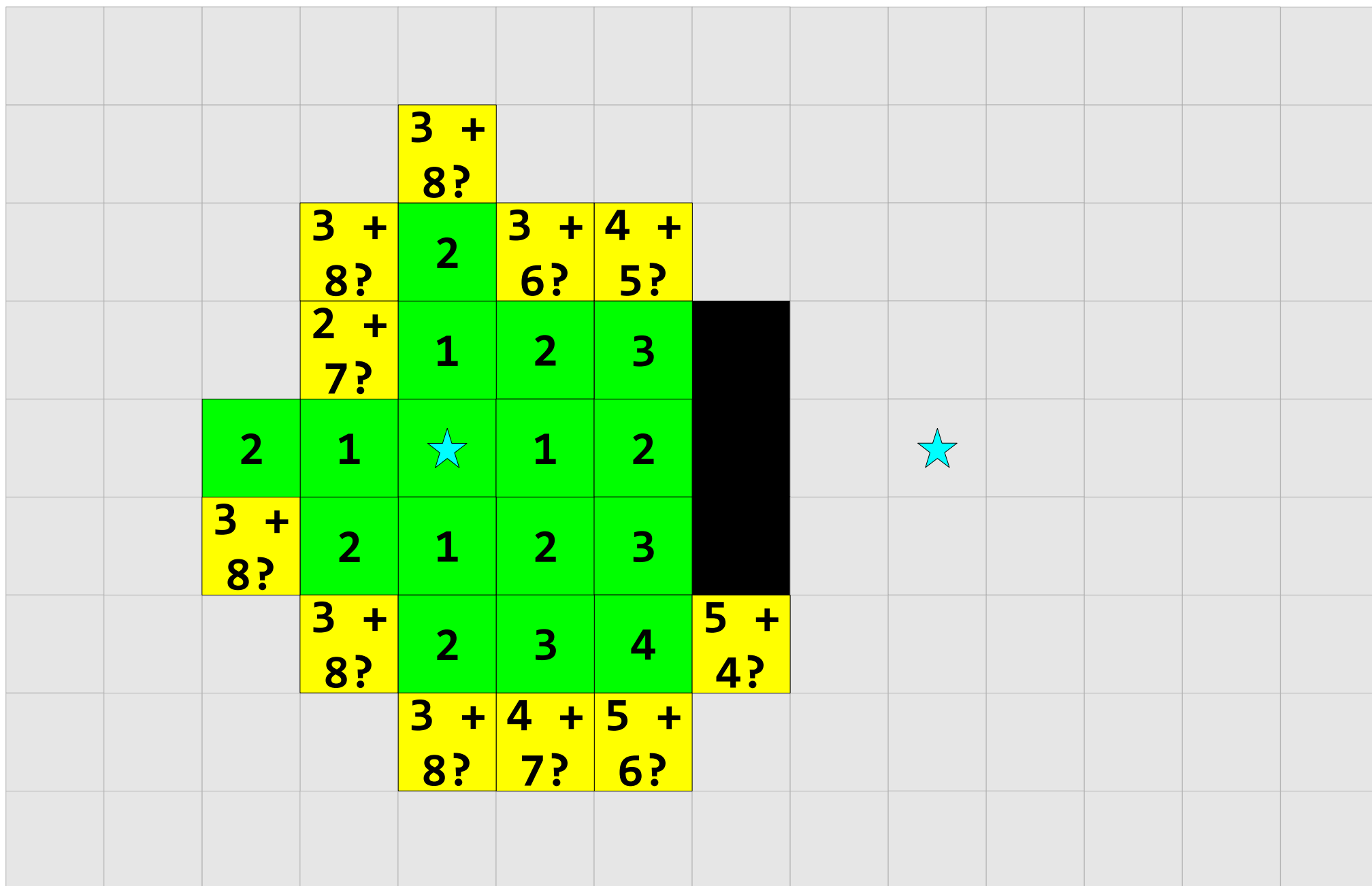


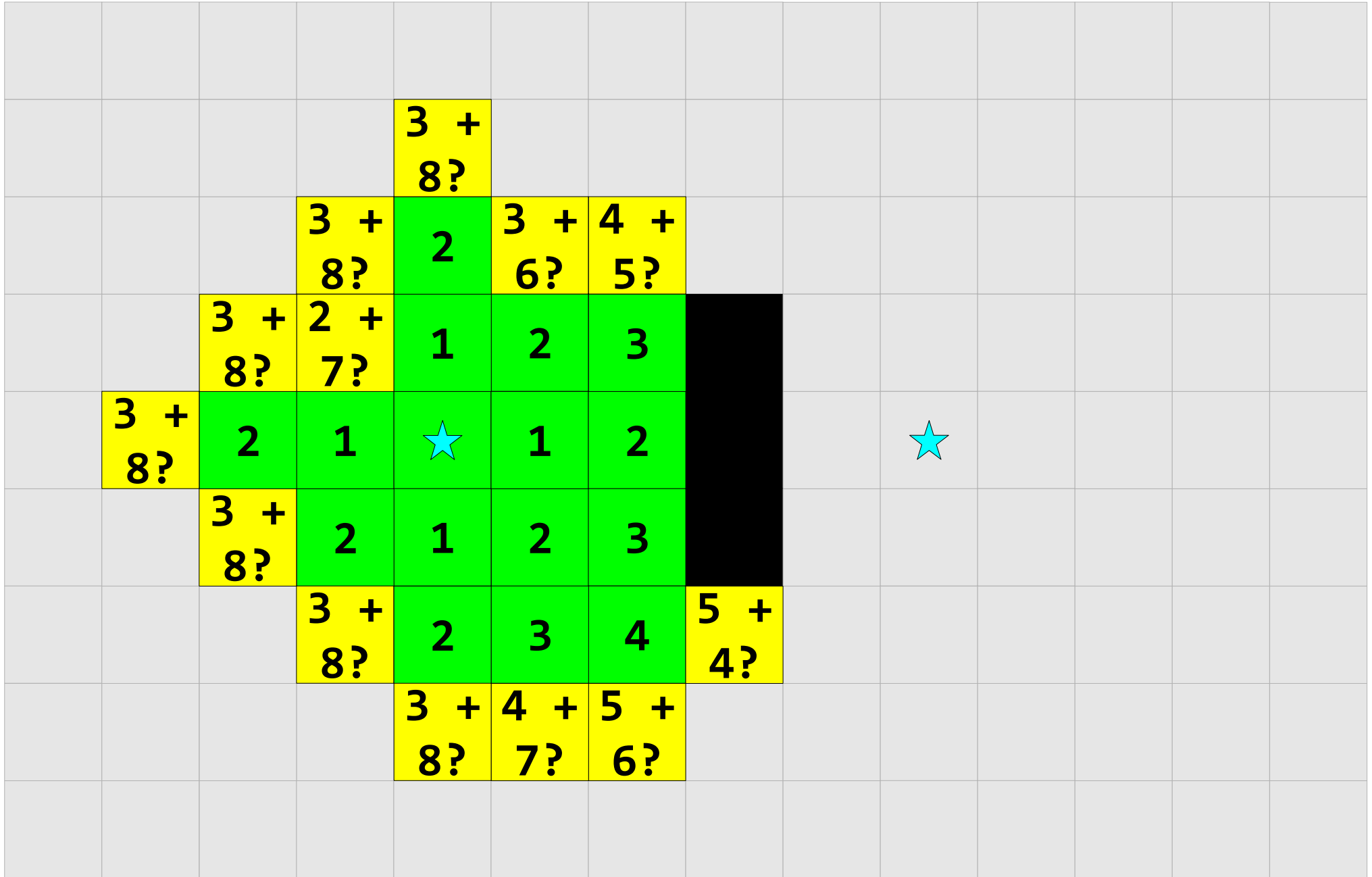


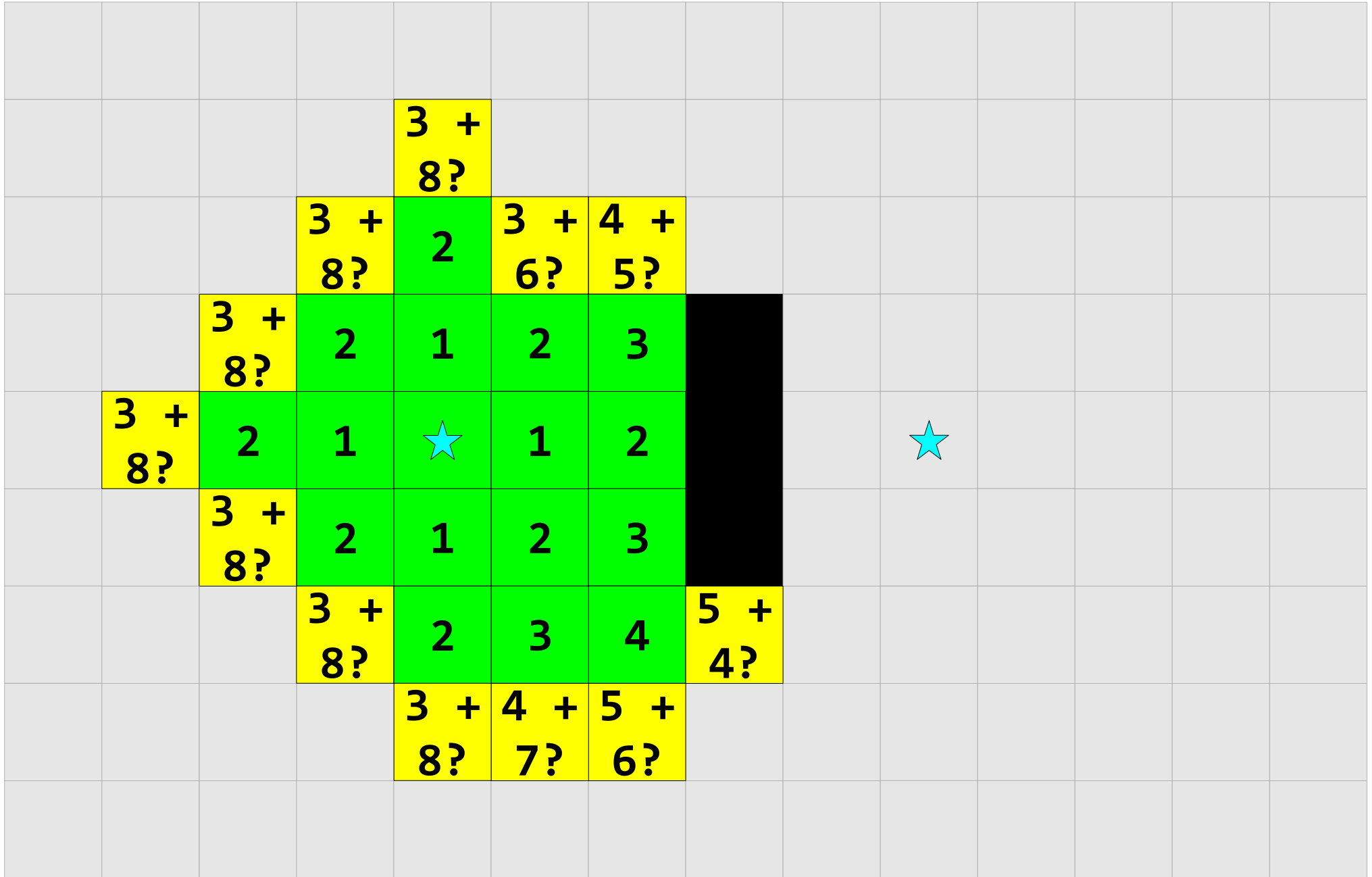


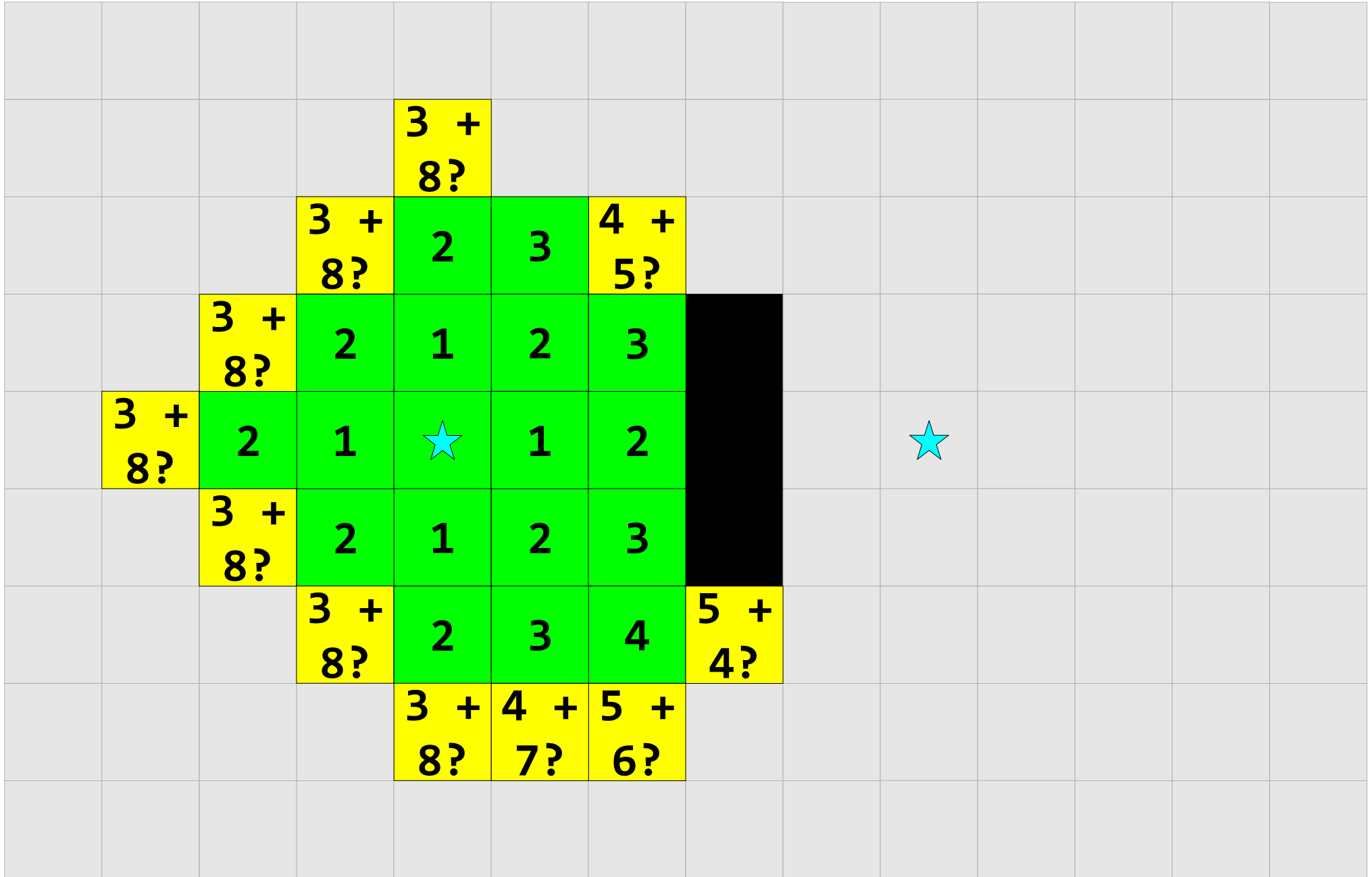


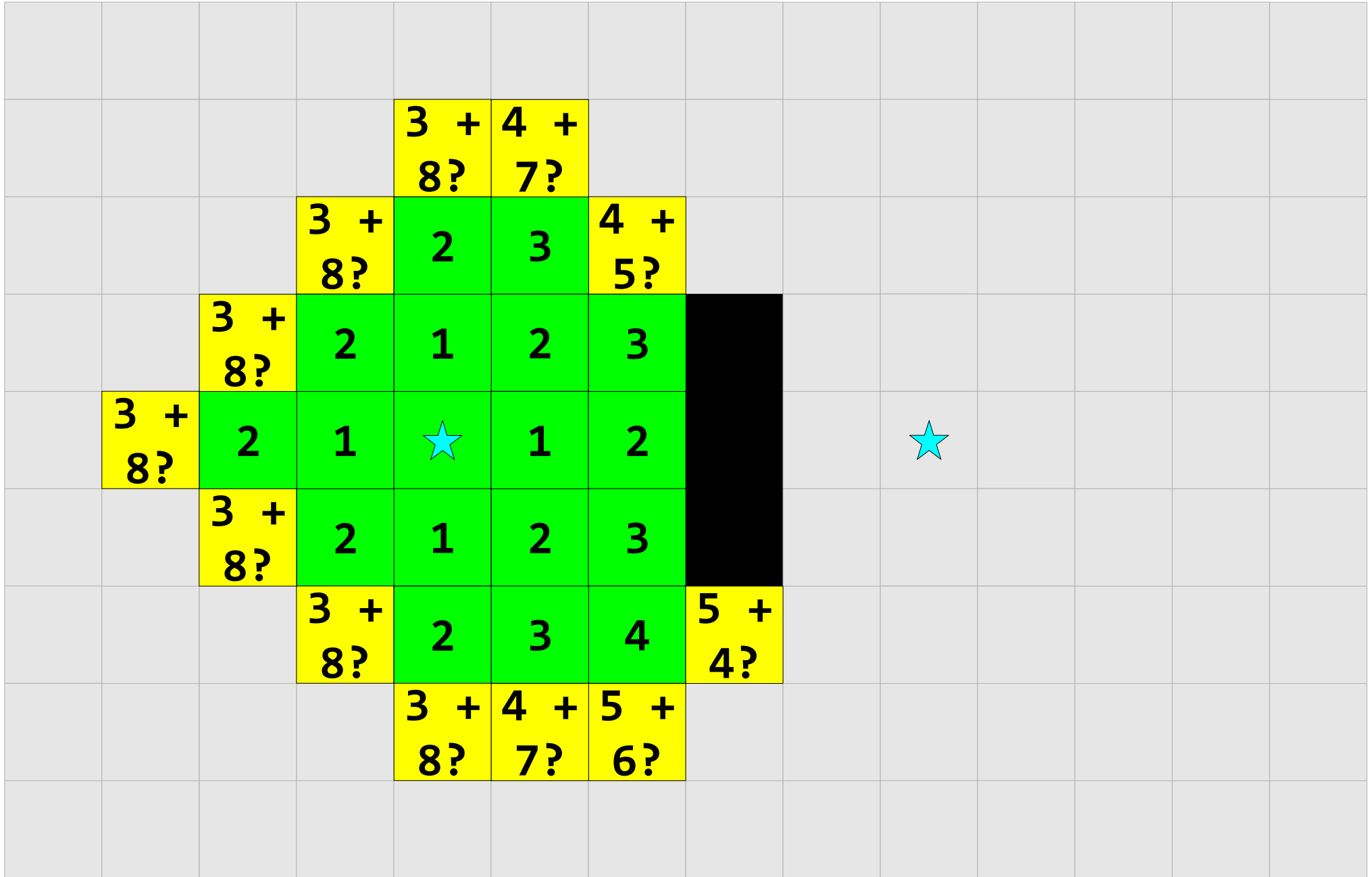




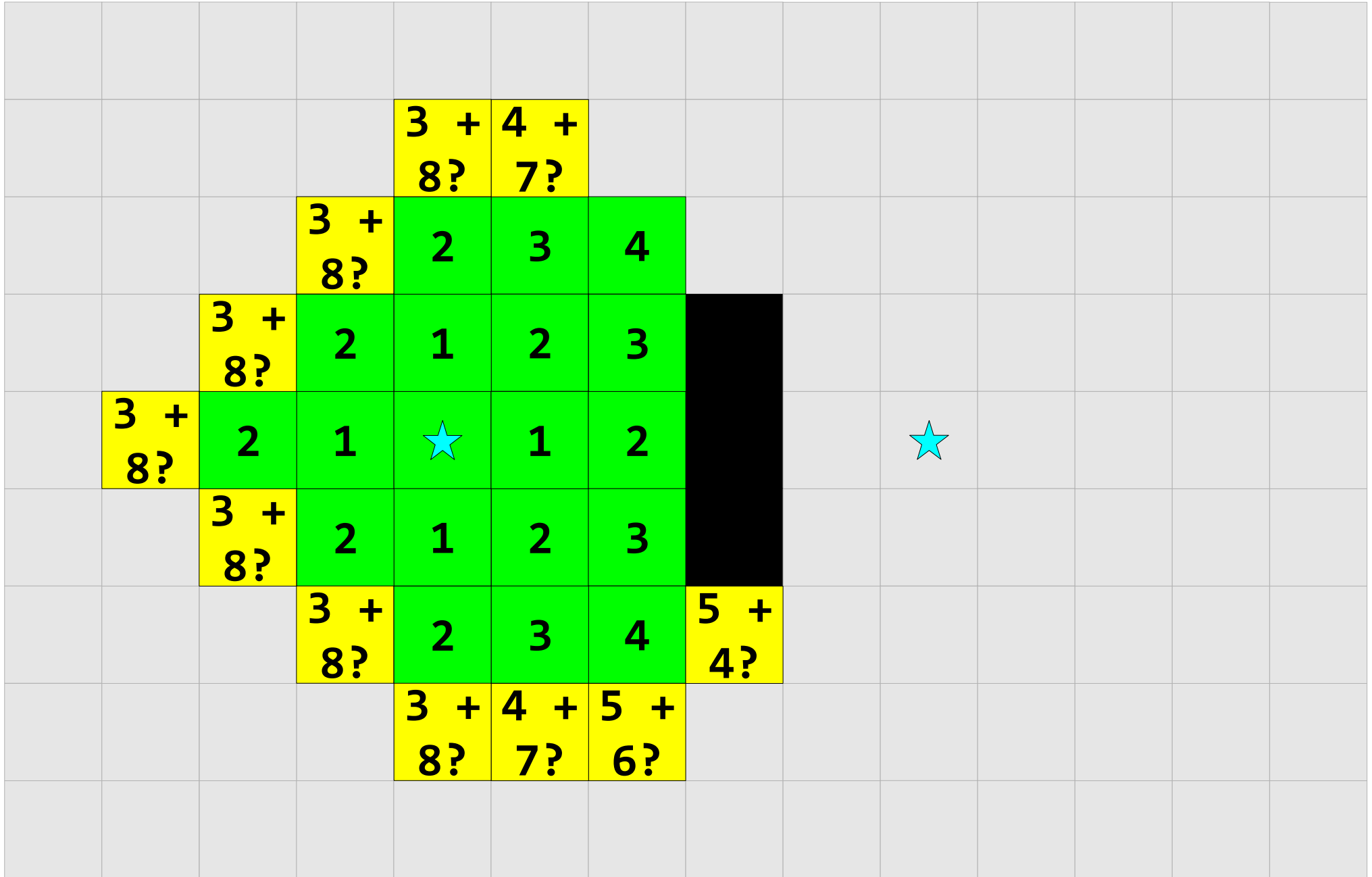




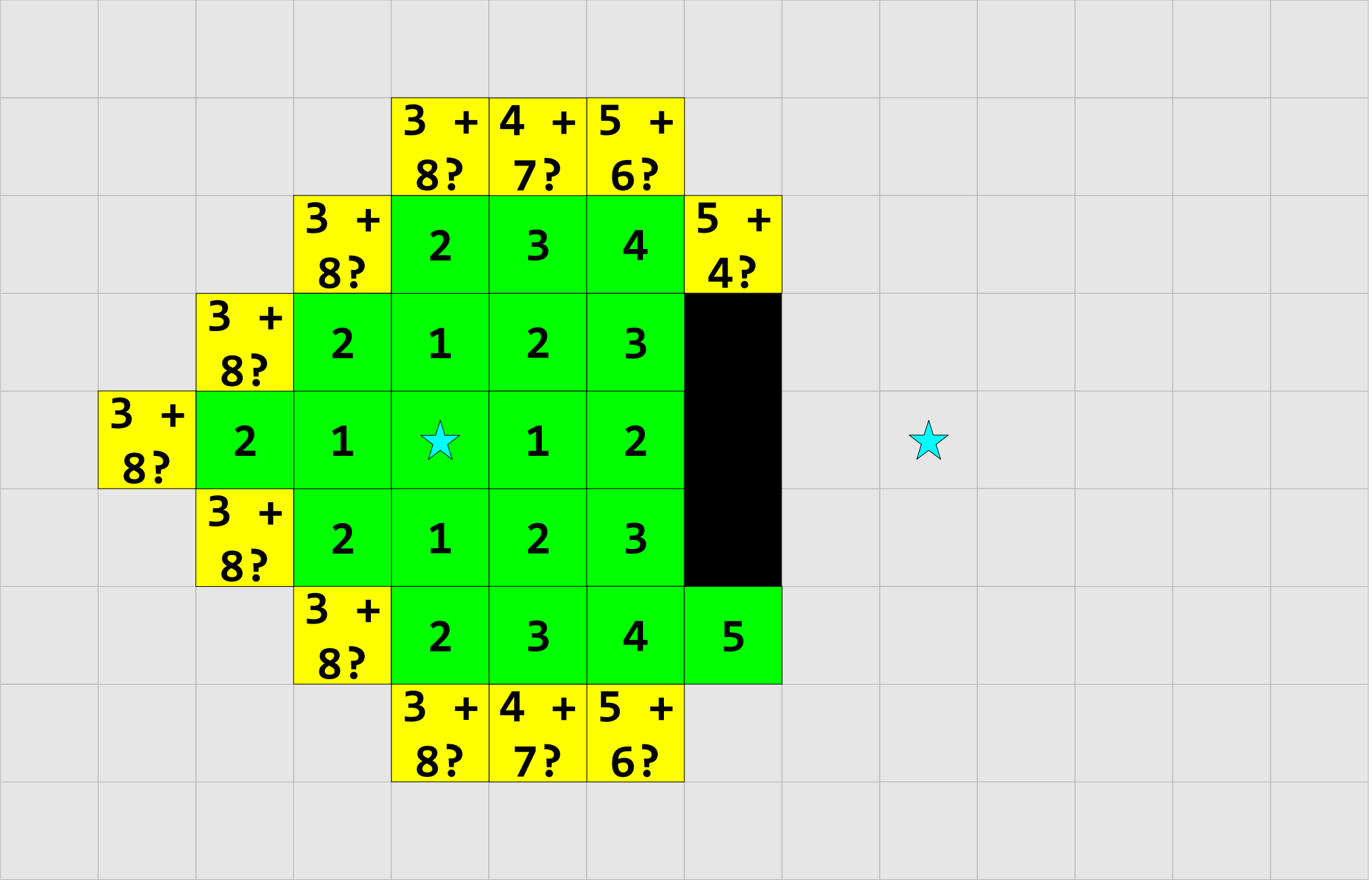


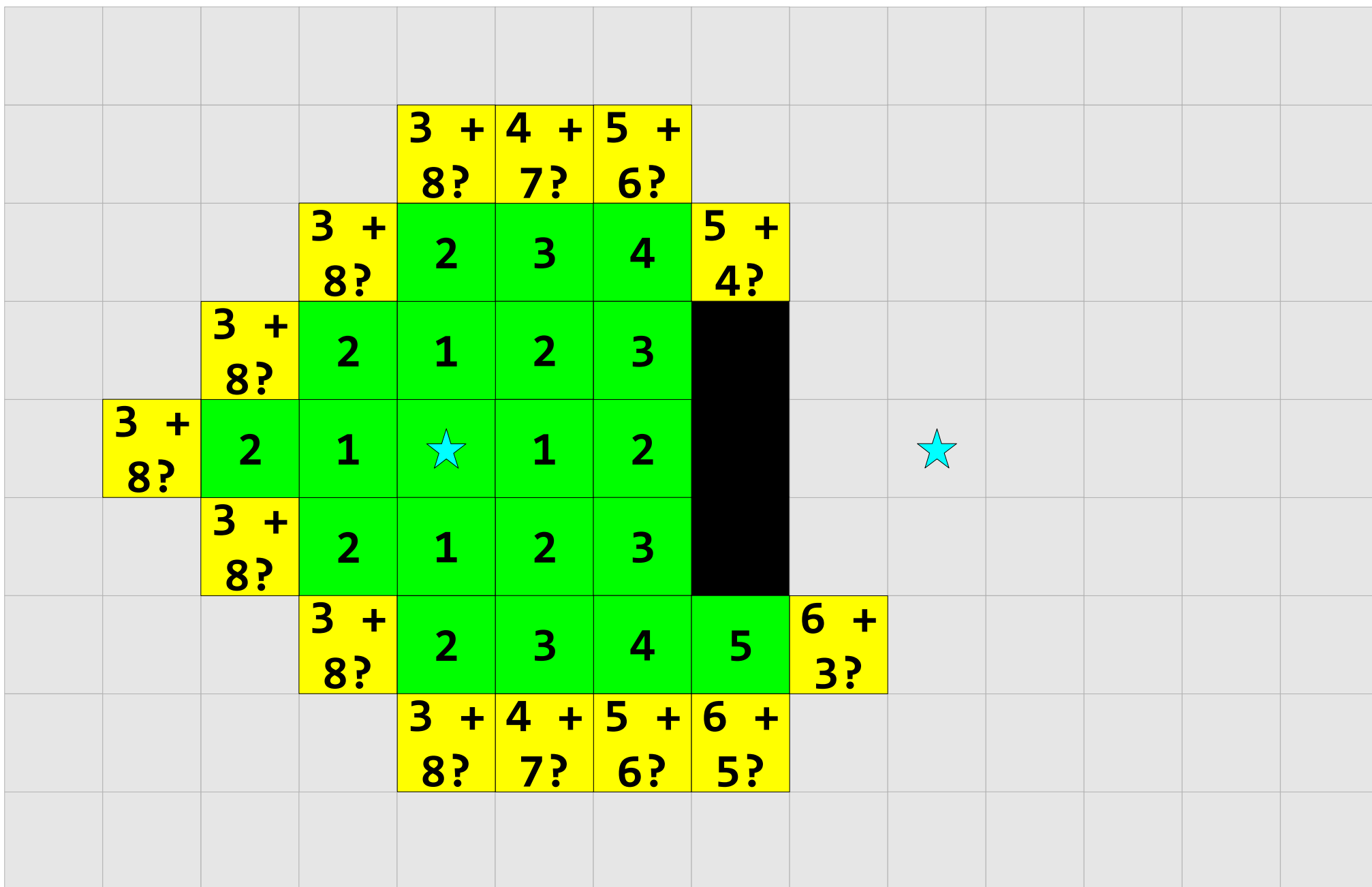


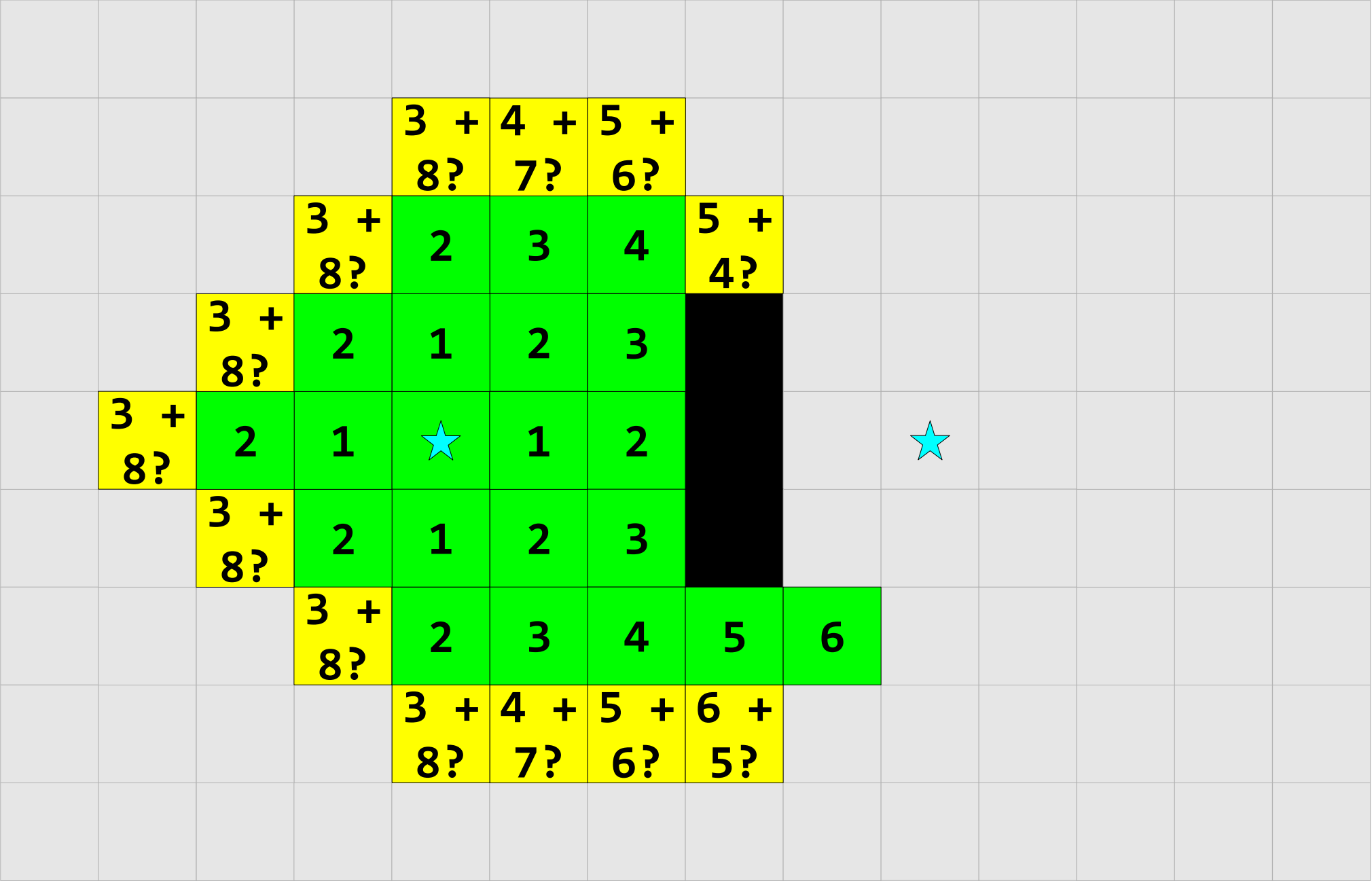




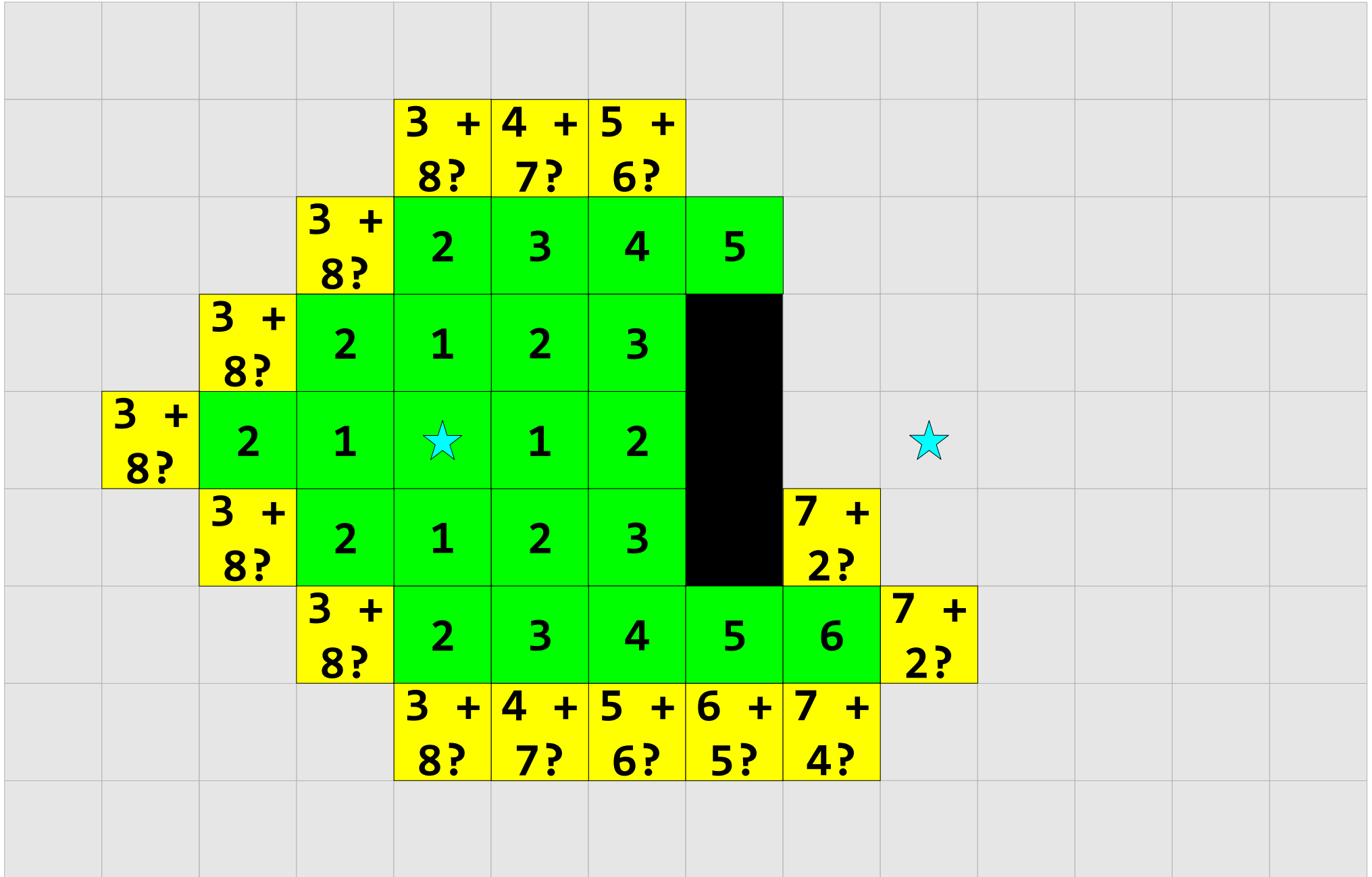


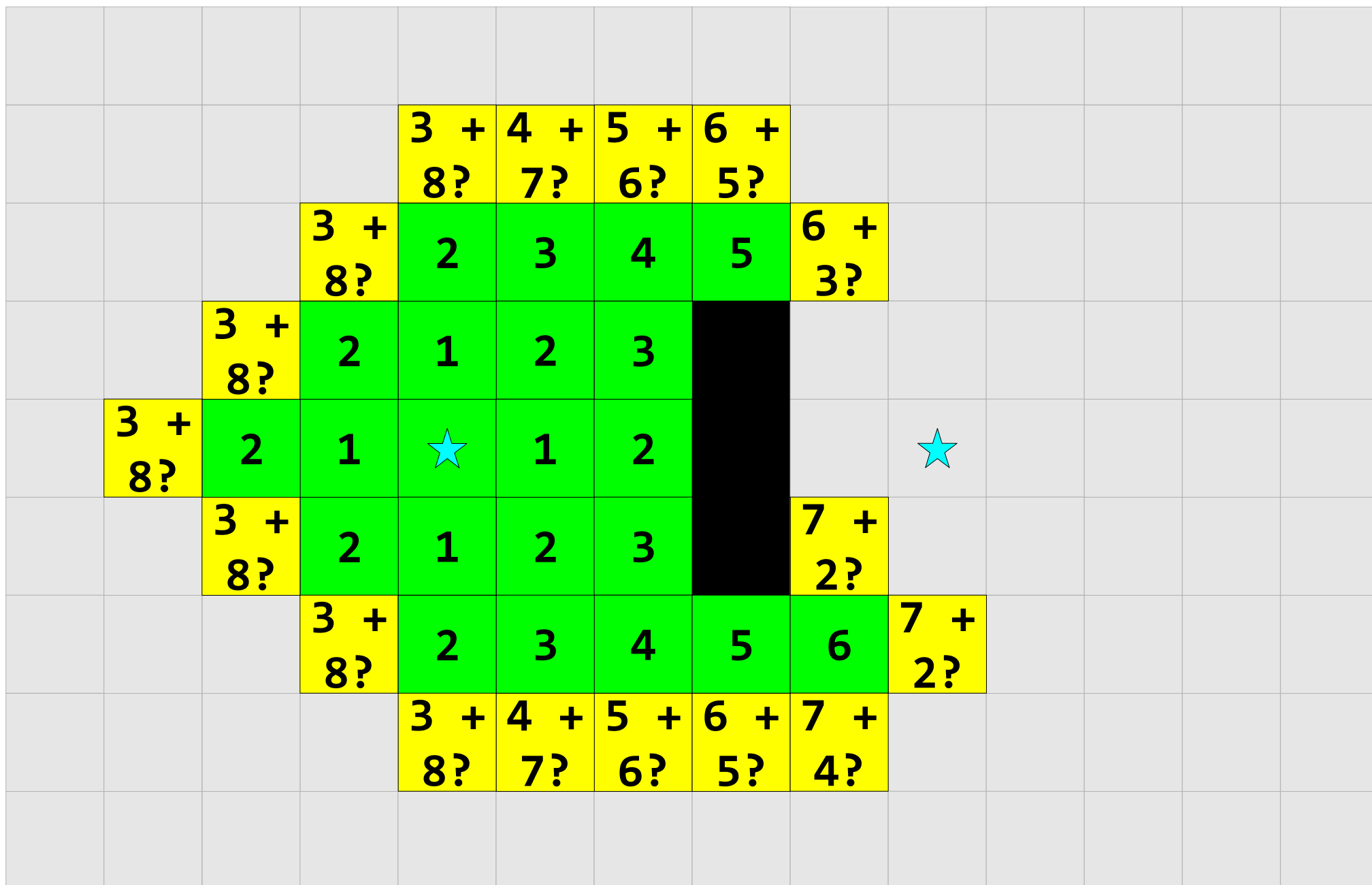




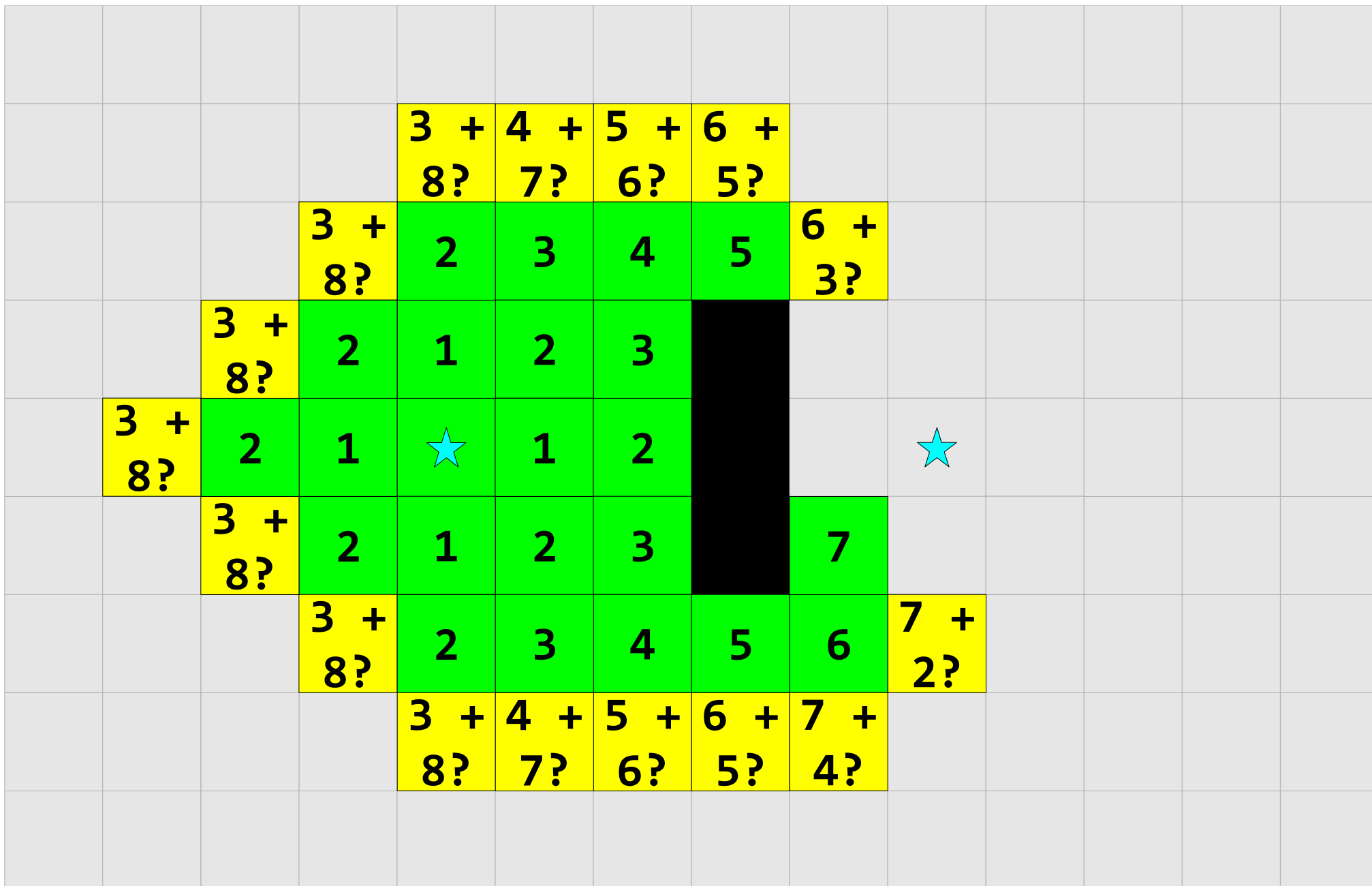


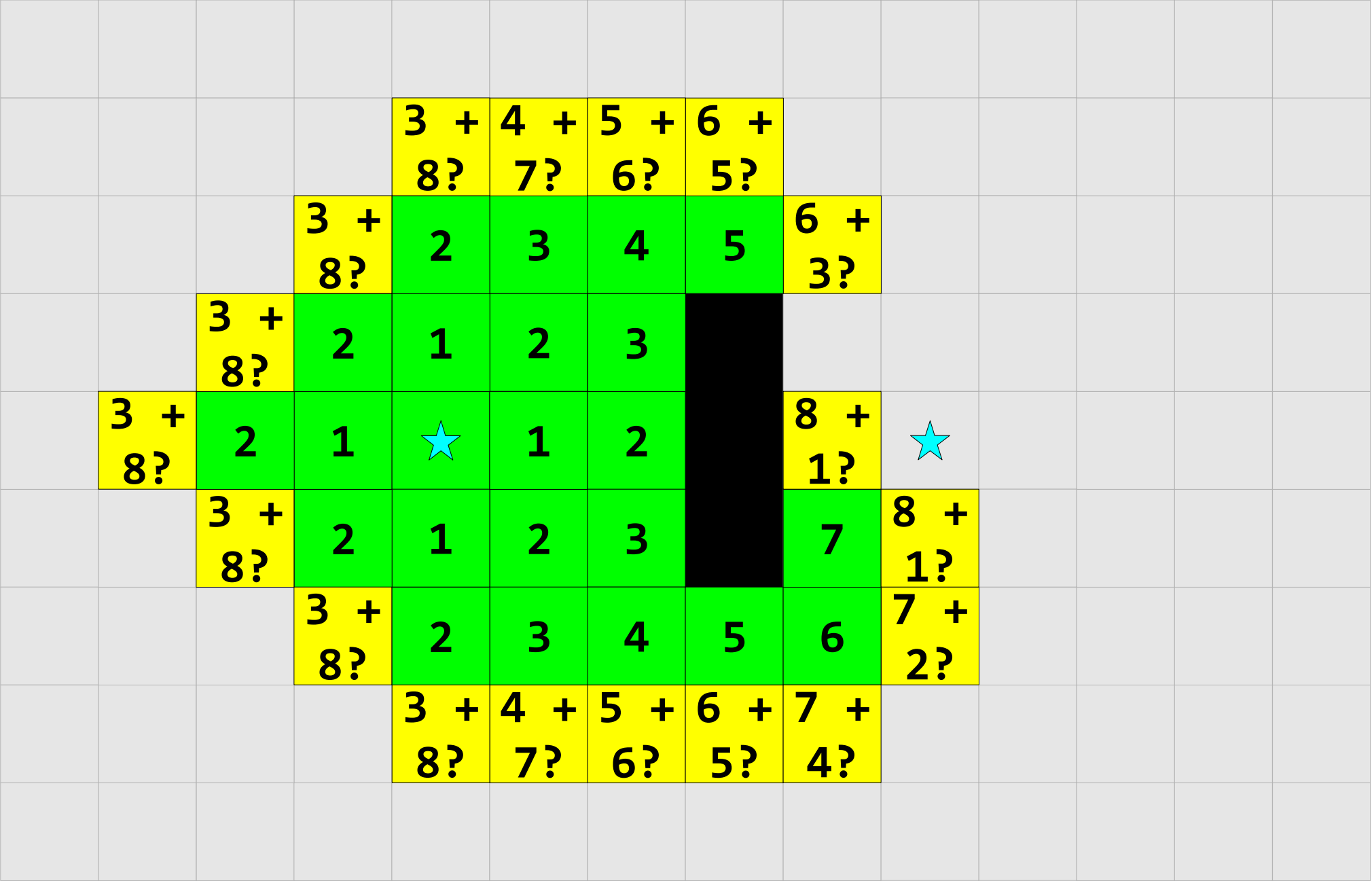


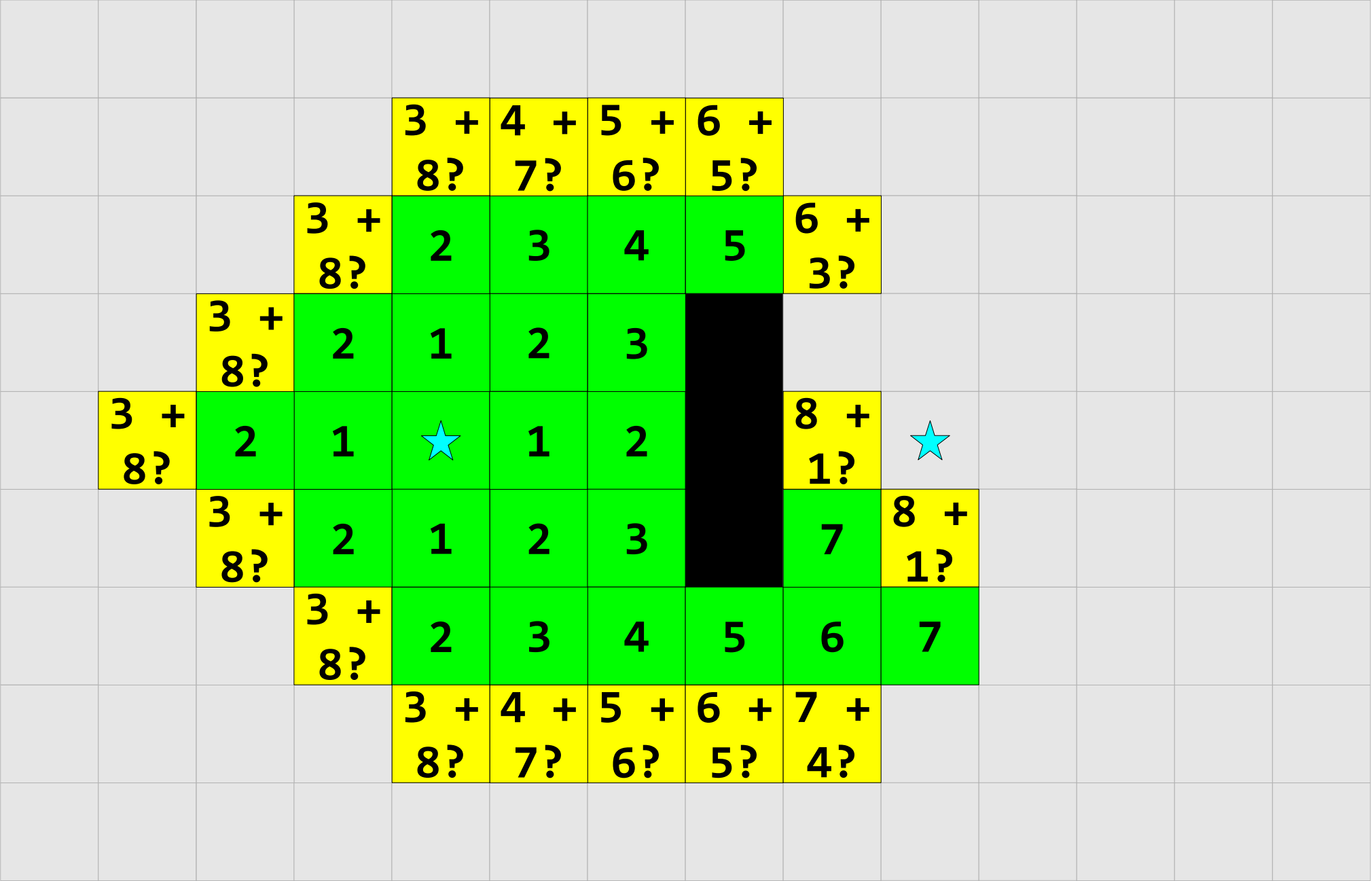


















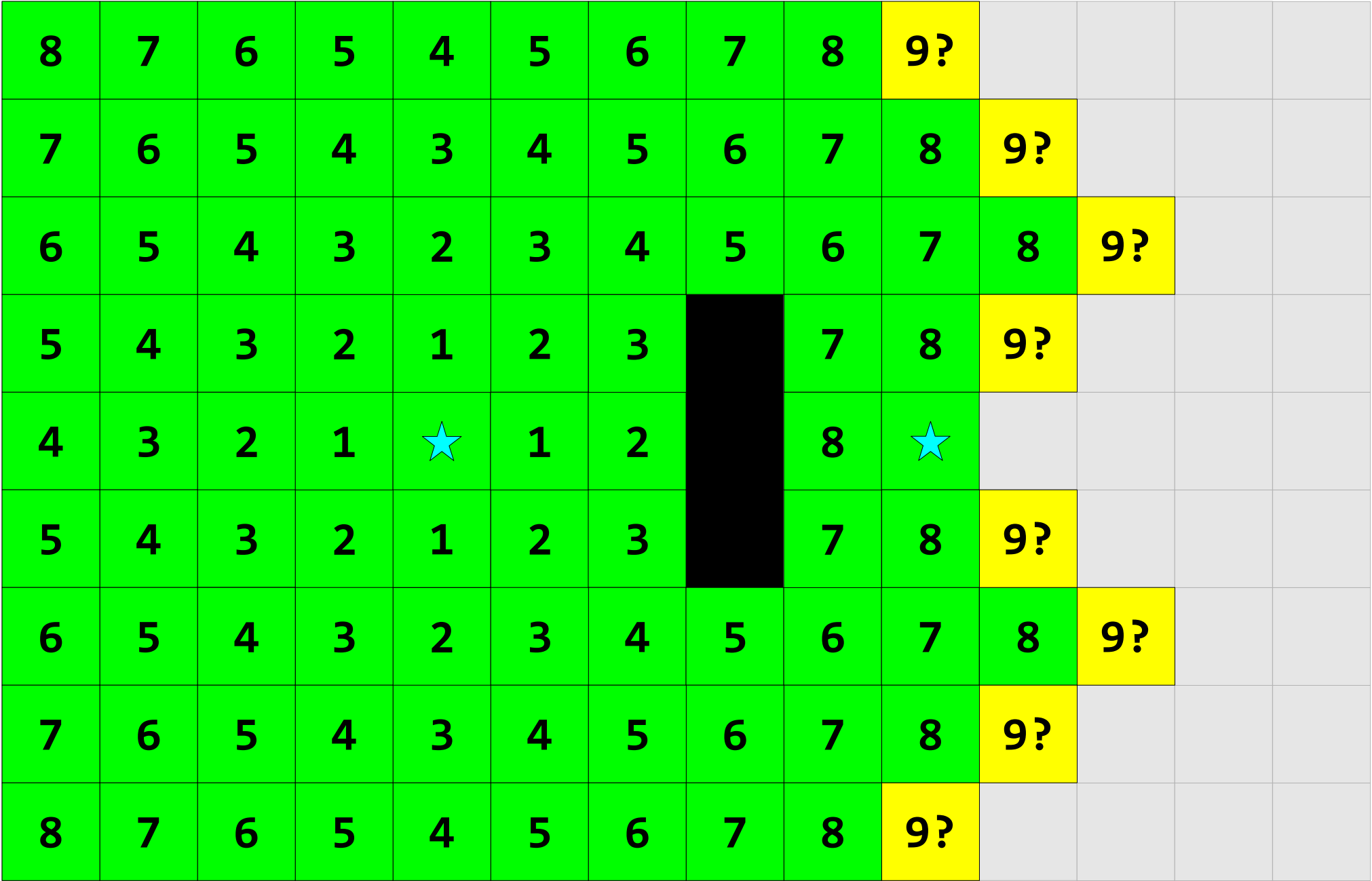








**For Comparison: What Dijkstra's Algorithm Would Have Searched**



# A\* Search

- As long as the heuristic is admissible (and satisfies one other technical condition), A\* will always find the shortest path from the source to the destination node.
- Can be *dramatically* faster than Dijkstra's algorithm.
  - Focuses work in areas likely to be productive.
  - Avoids solutions that appear worse until there is evidence they may be appropriate.

# A\* and Dijkstra's

- Dijkstra's algorithm and A\* search are very closely related:
  - Dijkstra's uses a node's candidate distance as its priority.
  - A\* uses a node's candidate distance **plus a heuristic value** as its priority.
- Interesting fact: If you use the **zero heuristic** (which always predicts a node is at distance 0 from the endpoint), A\* search is completely identical to Dijkstra's algorithm!

## Dijkstra's Algorithm

- Mark all nodes as gray.
- Mark the initial node **s** as yellow and at candidate distance **0**.
- Enqueue **s** into the priority queue with priority **0**.
- While not all nodes have been visited:
  - Dequeue the lowest-cost node **u** from the priority queue.
  - Color **u** green. The candidate distance **d** that is currently stored for node **u** is the length of the shortest path from **s** to **u**.
  - If **u** is the destination node **t**, you have found the shortest path from **s** to **t** and are done.
  - For each node **v** connected to **u** by an edge of length **L**:
    - If **v** is gray:
      - Color **v** yellow.
      - Mark **v**'s distance as **d + L**.
      - Set **v**'s parent to be **u**.
      - Enqueue **v** into the priority queue with priority **d + L**.
    - If **v** is yellow and the candidate distance to **v** is greater than **d + L**:
      - Update **v**'s candidate distance to be **d + L**.
      - Update **v**'s parent to be **u**.
      - Update **v**'s priority in the priority queue to **d + L**.

## A\* Search

- Mark all nodes as gray.
- Mark the initial node **s** as yellow and at candidate distance **0**.
- Enqueue **s** into the priority queue with priority  **$h(s, t)$** .
- While not all nodes have been visited:
  - Dequeue the lowest-cost node **u** from the priority queue.
  - Color **u** green. The candidate distance **d** that is currently stored for node **u** is the length of the shortest path from **s** to **u**.
  - If **u** is the destination node **t**, you have found the shortest path from **s** to **t** and are done.
  - For each node **v** connected to **u** by an edge of length **L**:
    - If **v** is gray:
      - Color **v** yellow.
      - Mark **v**'s distance as  **$d + L$** .
      - Set **v**'s parent to be **u**.
      - Enqueue **v** into the priority queue with priority  **$d + L + h(v, t)$** .
    - If **v** is yellow and the candidate distance to **v** is greater than  **$d + L$** :
      - Update **v**'s candidate distance to be  **$d + L$** .
      - Update **v**'s parent to be **u**.
      - Update **v**'s priority in the priority queue to  **$d + L + h(v, t)$** .