CS 106B

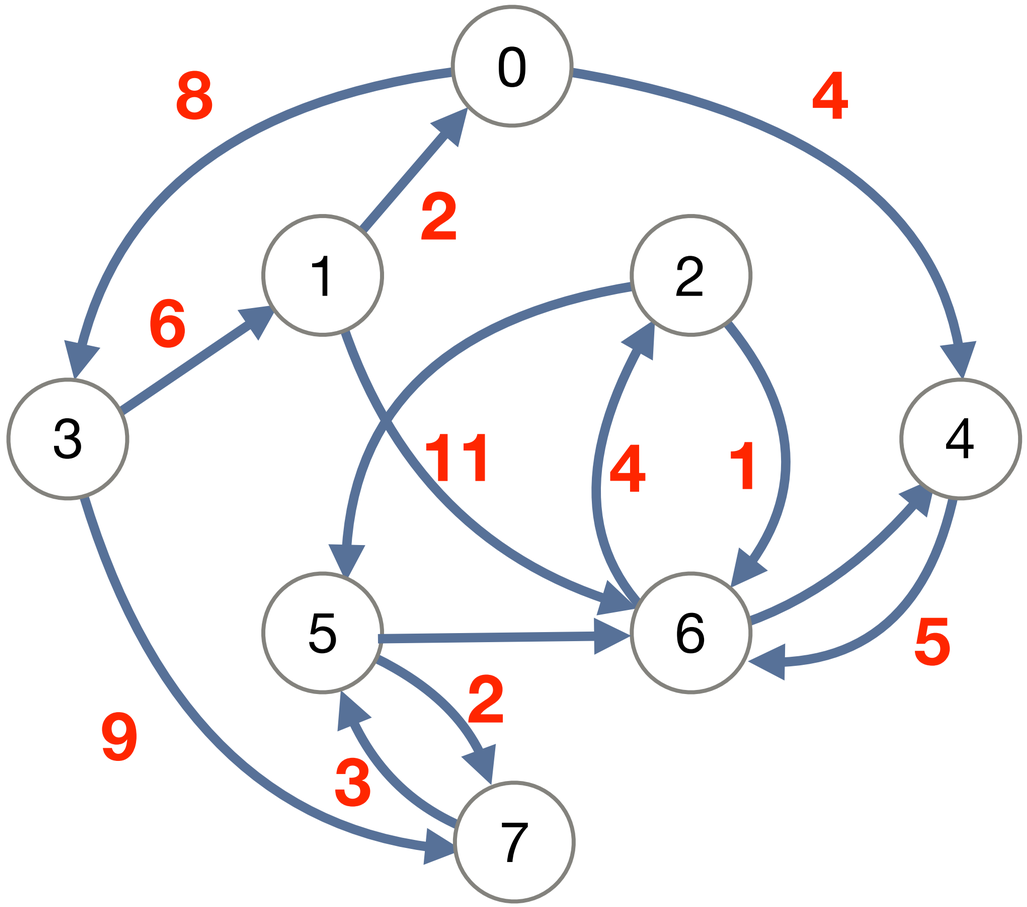
Lecture 25: Dijkstra's Algorithm and the A* Algorithm

Thursday, August 10, 2017

Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 18.6



Today's Topics

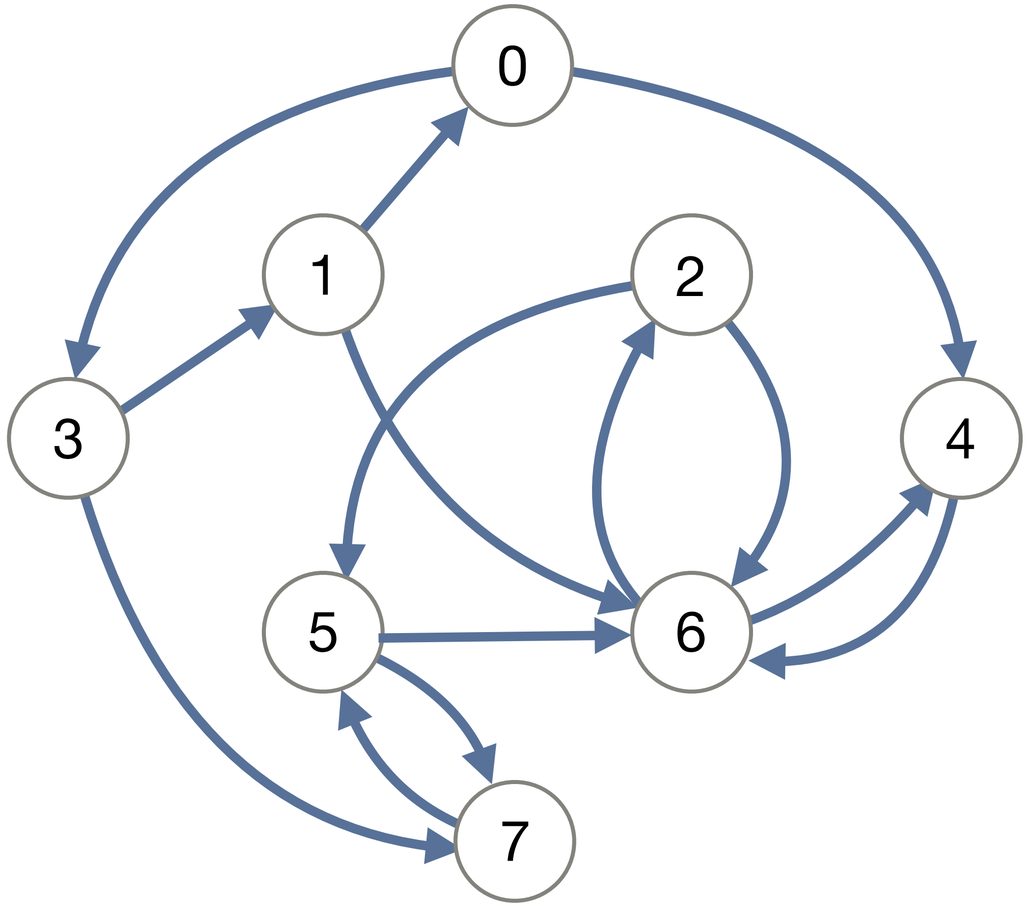
- Logistics
  - YEAH hours slides for Trailblazer: https://piazza.com/class/j44l3eyaz006uz?cid=921

- More on Graphs:
  - Dijkstra's Algorithm
  - A* Algorithm

Yesterday: DFS and BFS

Depth First Search: Keep searching along a path until we need to backtrack: not guaranteed shortest path.

Breadth First Search: Look at paths containing neighbor of distance 1, then neighbors of distance 2, etc., until a path is found: guaranteed shortest path.

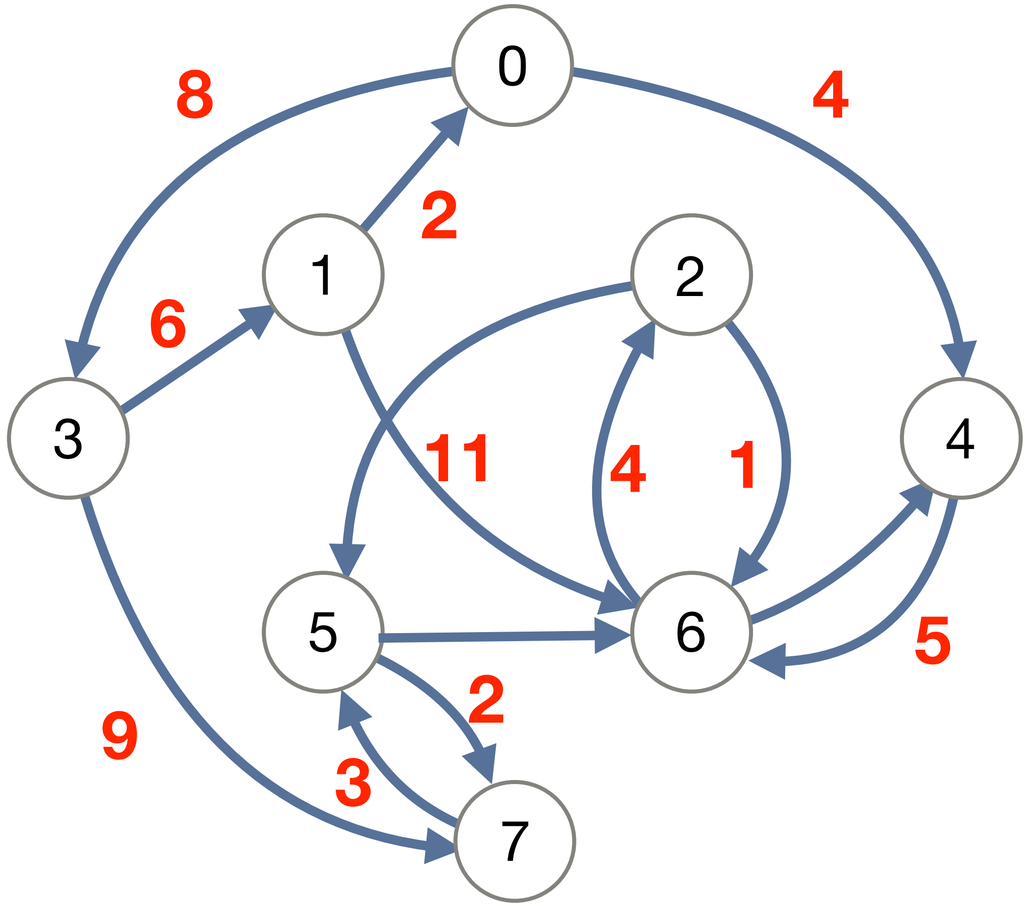


No Weights!

Depth First Search: Keep searching along a path until we need to backtrack: not guaranteed shortest path.

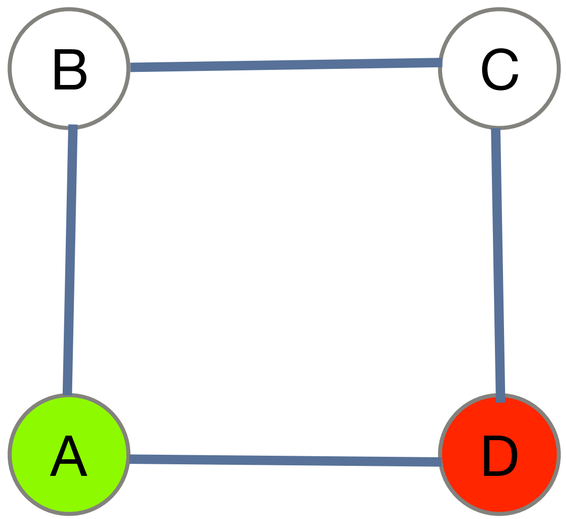
Breadth First Search: Look at paths containing neighbor of distance 1, then neighbors of distance 2, etc., until a path is found: guaranteed shortest path.

Neither DFS or BFS dealt with weights!



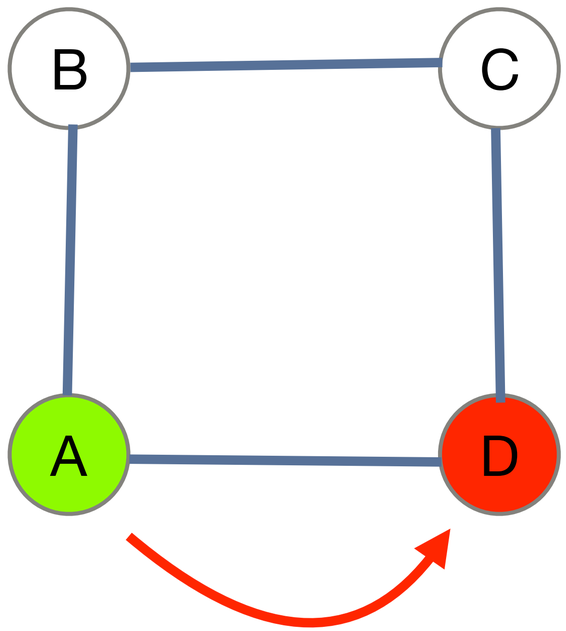
Search Without Weights

Search **without** weights: What is the shortest path from A to D?



Search With Weights

Search **without** weights: What is the shortest path from A to D?

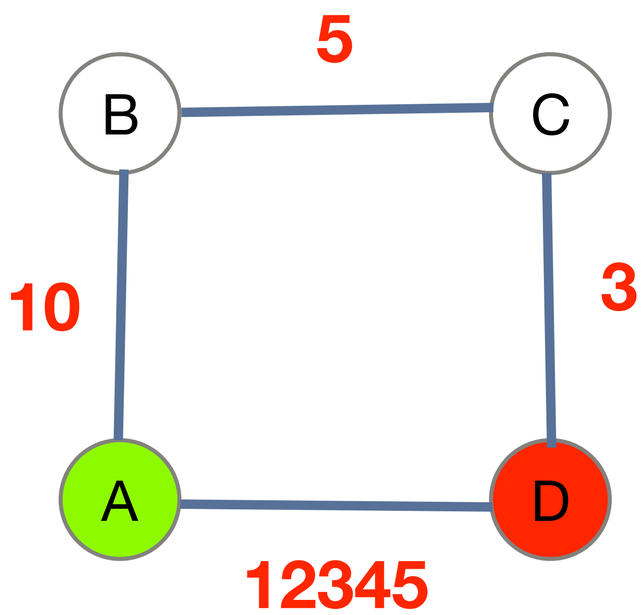


Shortest Path: A-D

Search With Weights

Search **with** weights: What is the shortest path from A to D?

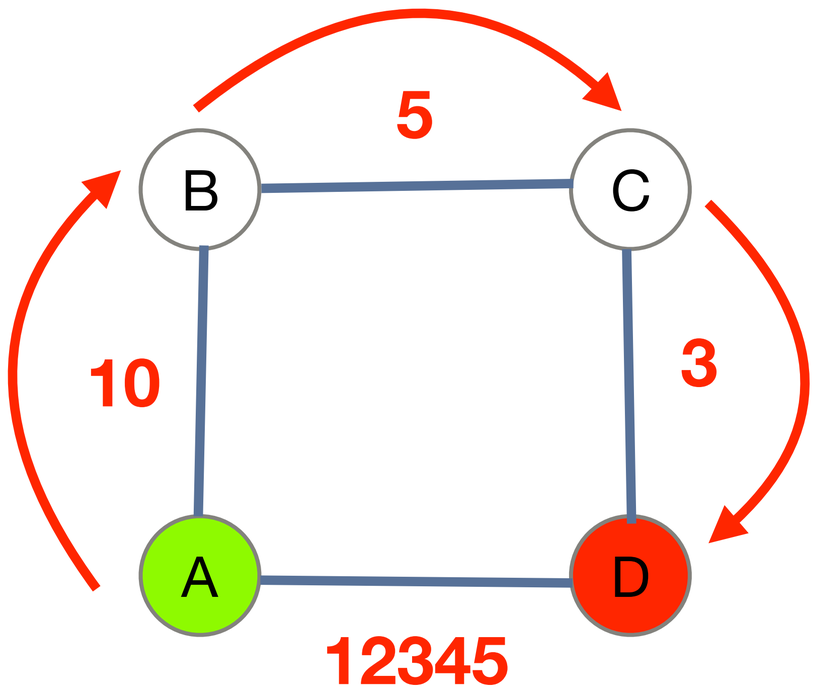
(Assume the numbers are distances, and we want to minimize the overall path distance)



Search With Weights

Search **with** weights: What is the shortest path from A to D?

(Assume the numbers are distances, and we want to minimize the overall path distance)

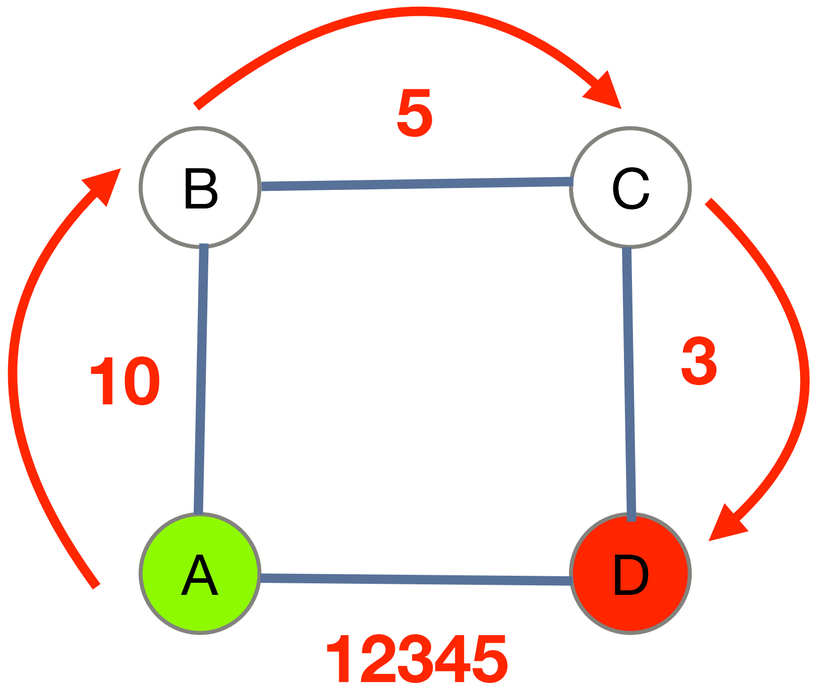


Shortest Path: A-B-C-D

Search With Weights

Search **with** weights: What is the shortest path from A to D?

(Assume the numbers are distances, and we want to minimize the overall path distance)

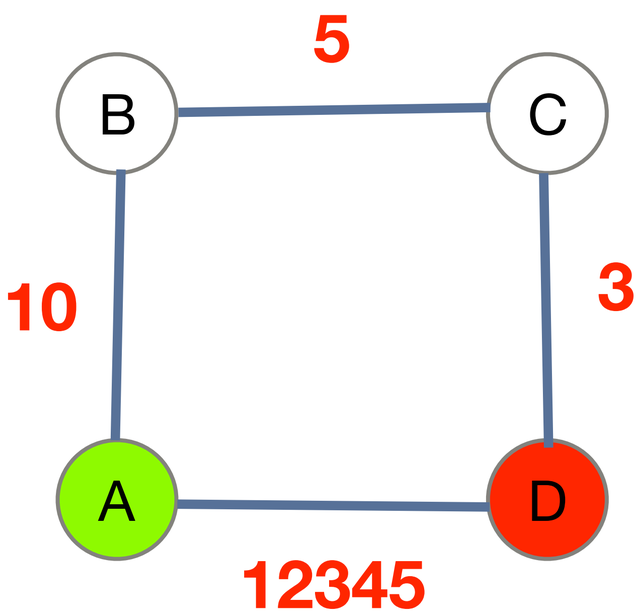


Shortest Path: A-B-C-D

Our BFS would break! The "shortest" path with weights depends on the weight!

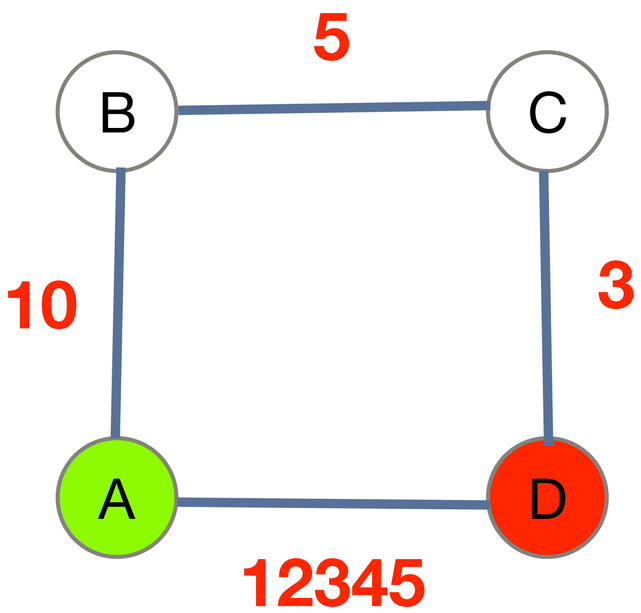
BFS without weights...

If we use BFS to find the path (disregarding weights), we would use a **queue** to enqueue each path.



Dijkstra's Algorithm

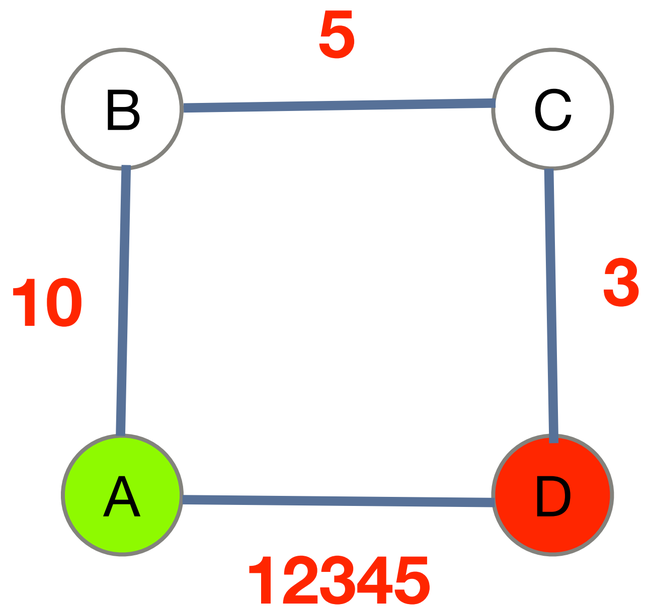
A different algorithm, called "Dijstra's Algorithm" (after the computer scientist Edsger Dijkstra) uses a **priority queue** to enqueue each path.



Breadth First Search

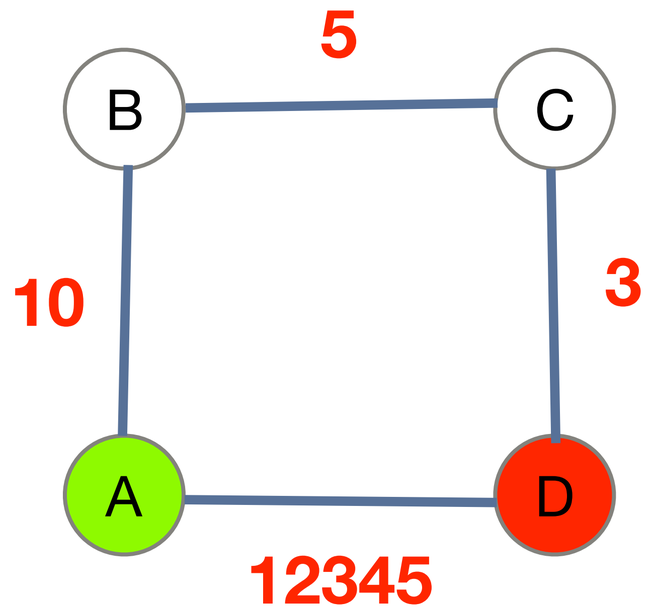
bfs from v_1 to v_2:

create a **queue** of paths (a vector), q
q.enqueue(v_1 path)
while q is not empty and v_2 is not yet visited:
path = q.dequeue()
v = last element in path
mark v as visited
if v is the end vertex, we can stop.
for each unvisited neighbor of v:
make new path with v's neighbor as last element
enqueue new path onto q



Dijkstra's Algorithm

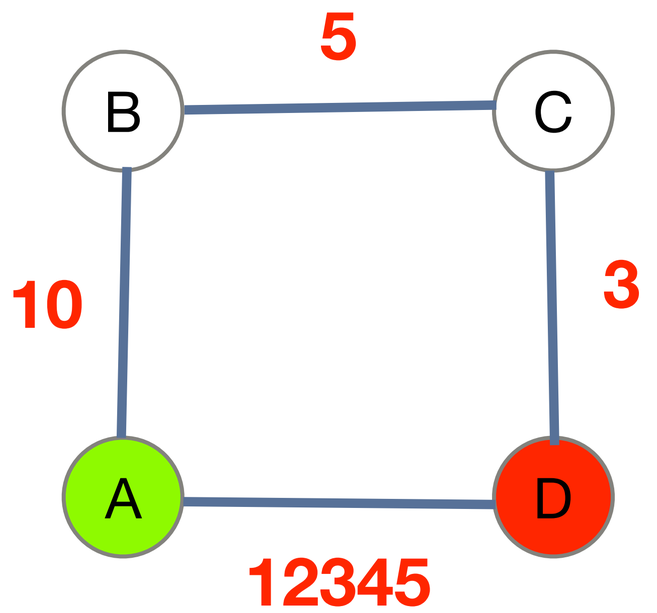
bfs from v_1 to v_2:
    create a **priority queue** of paths (a vector), q
    q.enqueue(v_1 path)
    while q is not empty and v_2 is not yet visited:
        path = q.dequeue()
        v = last element in path
        mark v as visited
        if v is the end vertex, we can stop.
       for each unvisited neighbor of v:
            make new path with v's neighbor as last element
            enqueue new path onto q



Dijkstra's Algorithm

Dijkstra's algorithm is what we call a "greedy" algorithm.

This means that the algorithm always takes the path that is best at the given time -- e.g., starting from A, you would prioritize the path from A-B (10) over the path from A-D (12345). This is why we use a priority queue, because the prioritization is handled with a priority queue.



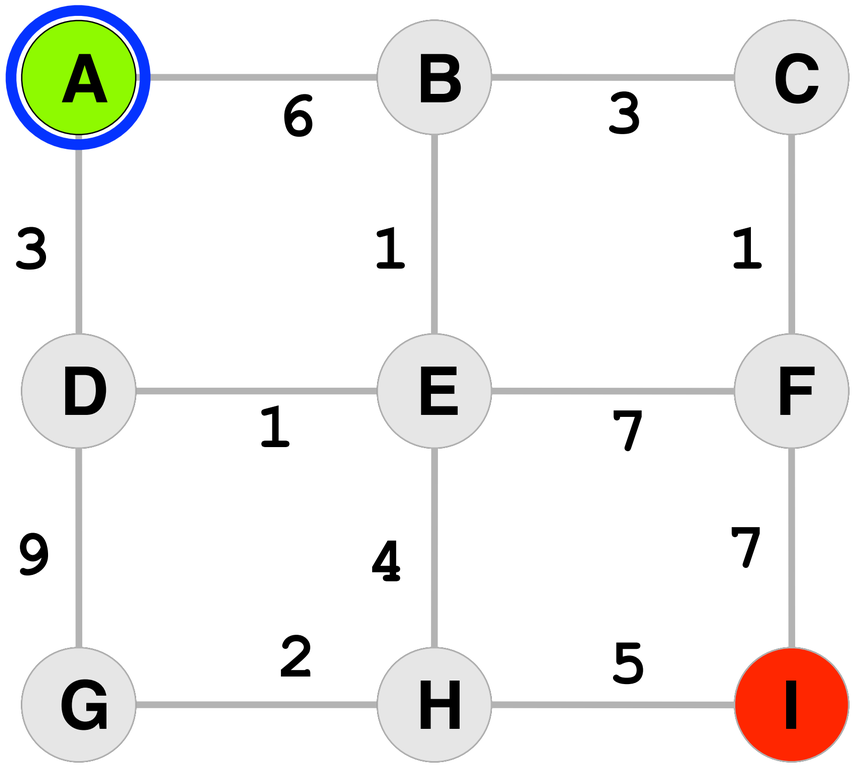
Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:							A
Total Cost:							0




```
Vector<Vertex  *> startPath
startPath.add(A,0)
pq.enqueue(startPath)
```

Visited Set: (empty)

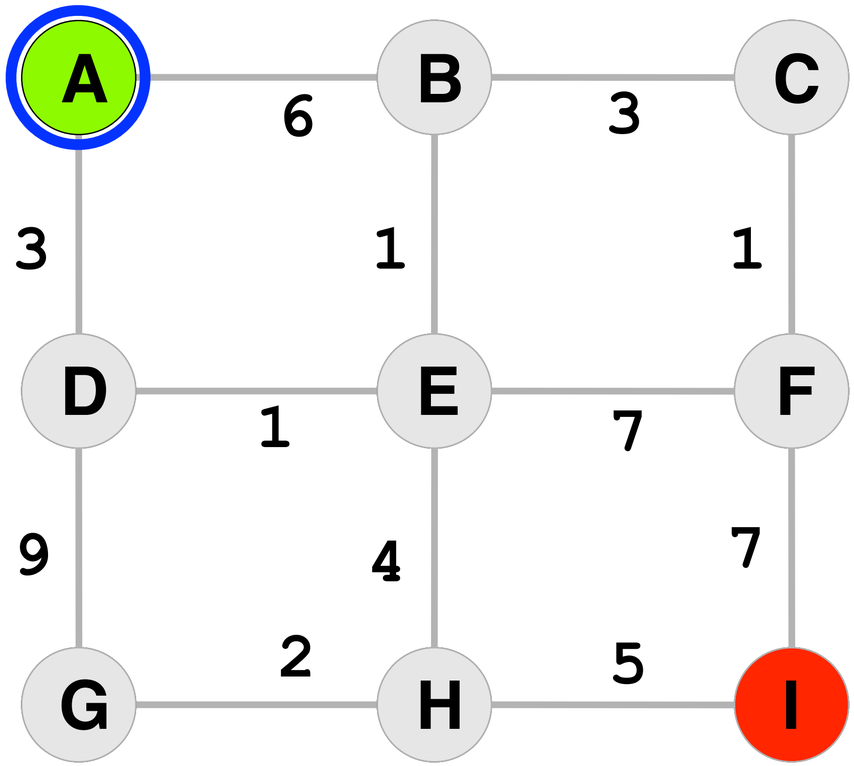
Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:						AB	AD
Total Cost:						6	3



in while loop:
curPath = pq.dequeue() (path is A, priority is 0)
v = last element in curPath (v is A)
mark v as visited
enqueue all unvisited neighbor paths onto q,
with updated priorities based on new edge length

Visited Set: A

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:					ADG	AB	ADE
Total Cost:					12	6	4

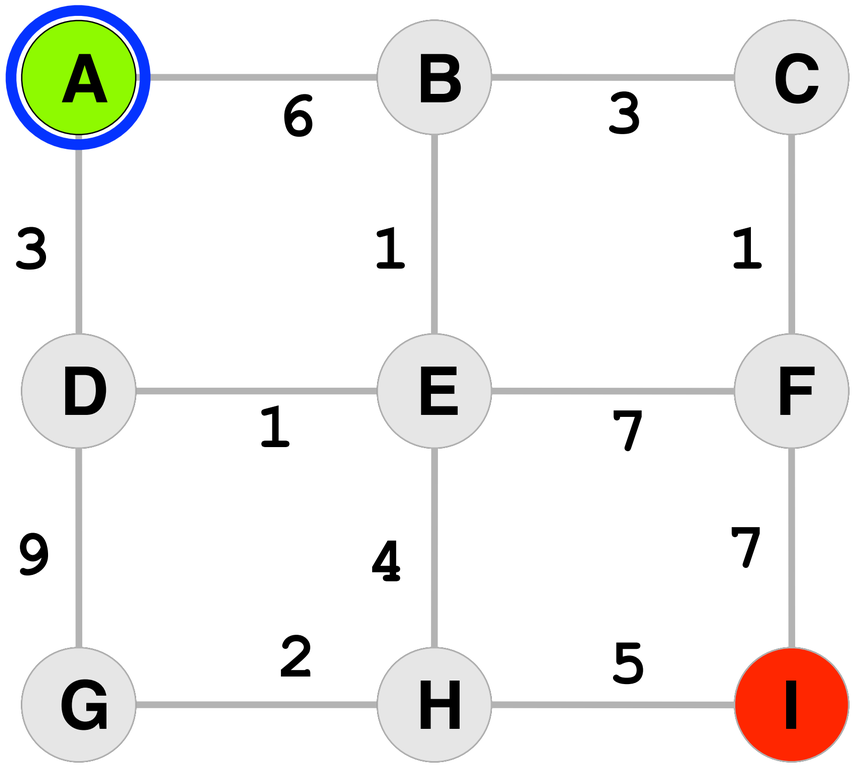
in while loop:

curPath = pq.dequeue() (path is AD, priority is 3)

v = last element in curPath (v is D)

mark v as visited

enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length



Visited Set: A, D

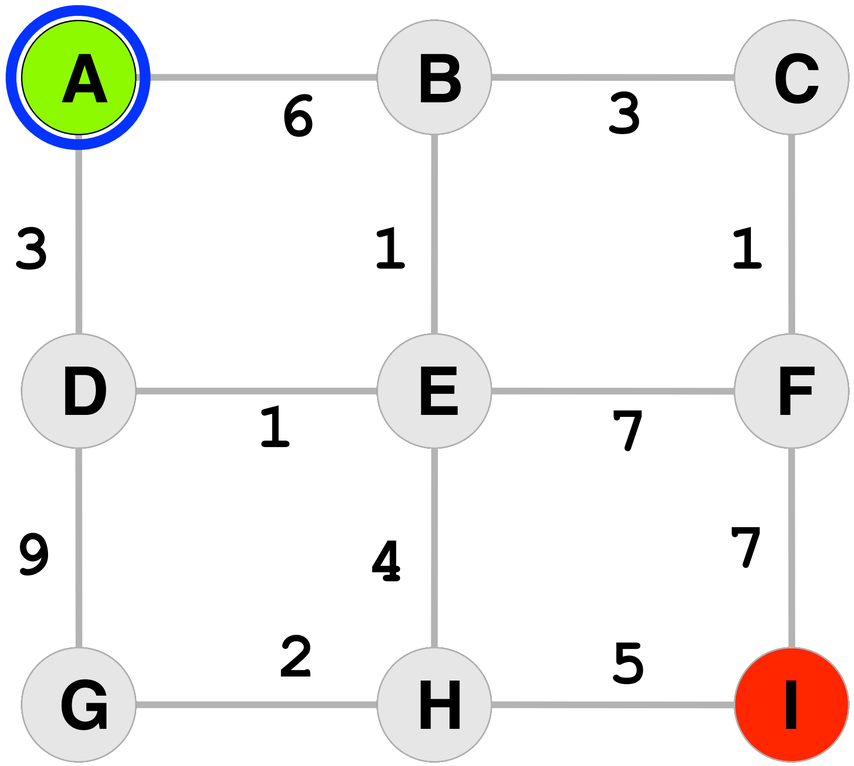
Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i: priority queue:

							front
Path:			ADG	ADEF	ADEH	AB	ADEB
Total Cost:			12	11	8	6	5

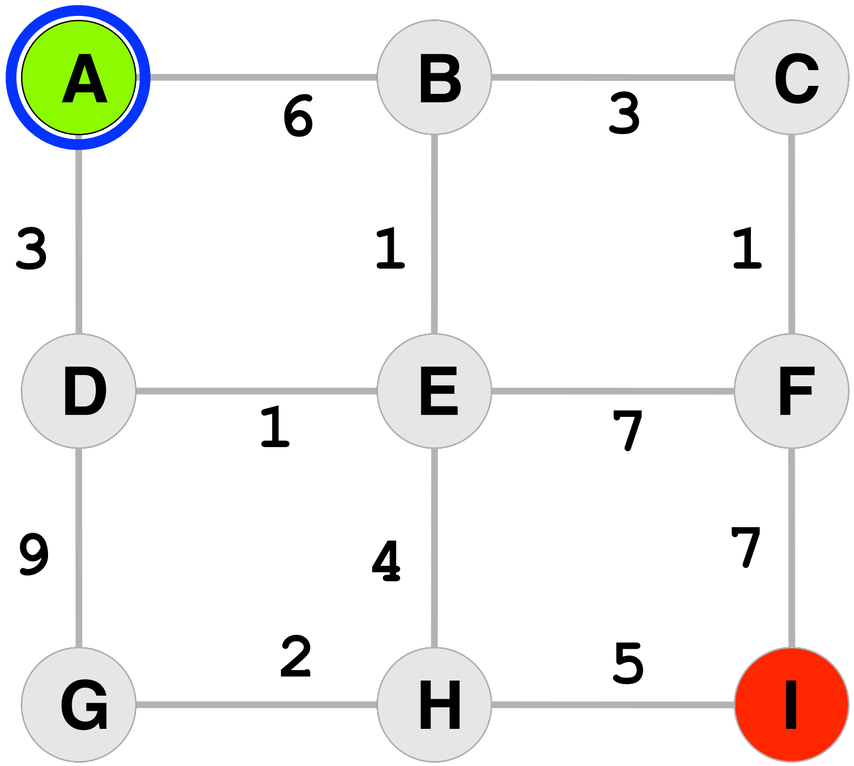
in while loop:
curPath = pq.dequeue() (path is ADE, priority is 4)
v = last element in curPath (v is E)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length



Visited Set: A, D, E

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i: priority queue:

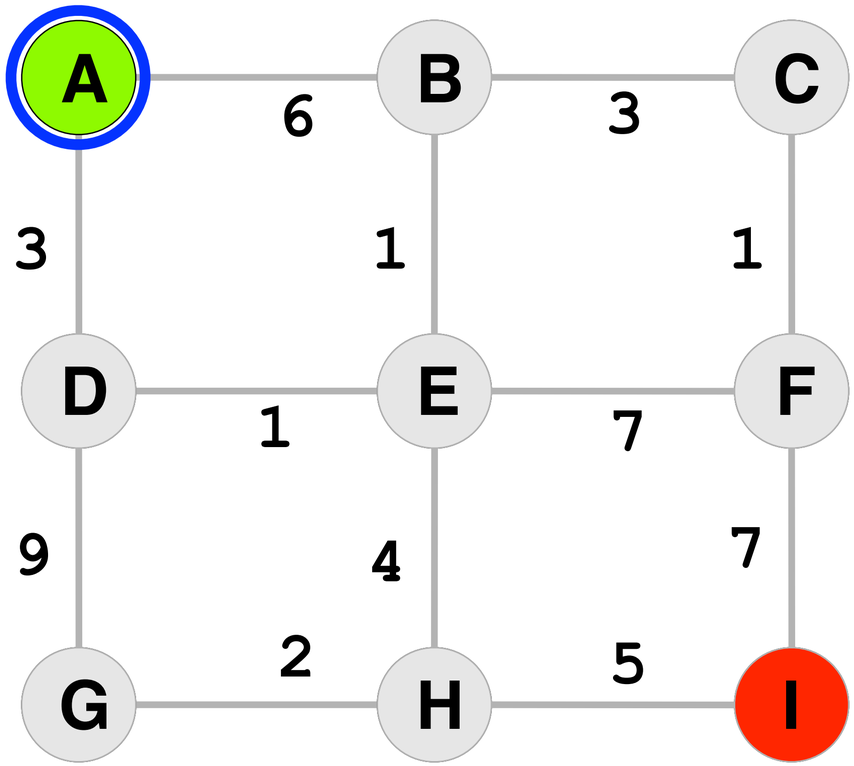
							front
Path:			ADG	ADEF	ADEBC	ADEH	AB
Total Cost:			12	11	8	8	6

in while loop:
curPath = pq.dequeue() (path is ADEB, priority is 5)
v = last element in curPath (v is B)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i: priority queue:

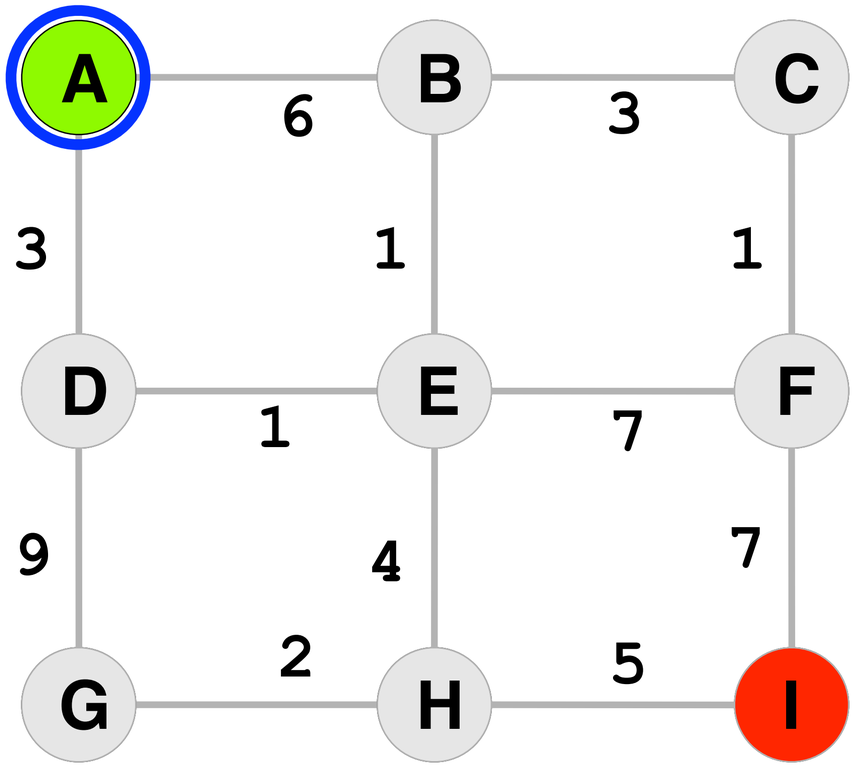
							front
Path:			ADG	ADEF	ABC	ADEBC	ADEH
Total Cost:			12	11	9	8	8

in while loop:
curPath = pq.dequeue() (path is AB, priority is 6)
v = last element in curPath (v is B)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:		ADEHI	ADG	ADEF	ADEHG	ABC	ADEBC
Total Cost:		13	12	11	10	9	8

in while loop:

curPath = pq.dequeue() (path is ADEH, priority is 8)

v = last element in curPath (v is H)

mark v as visited

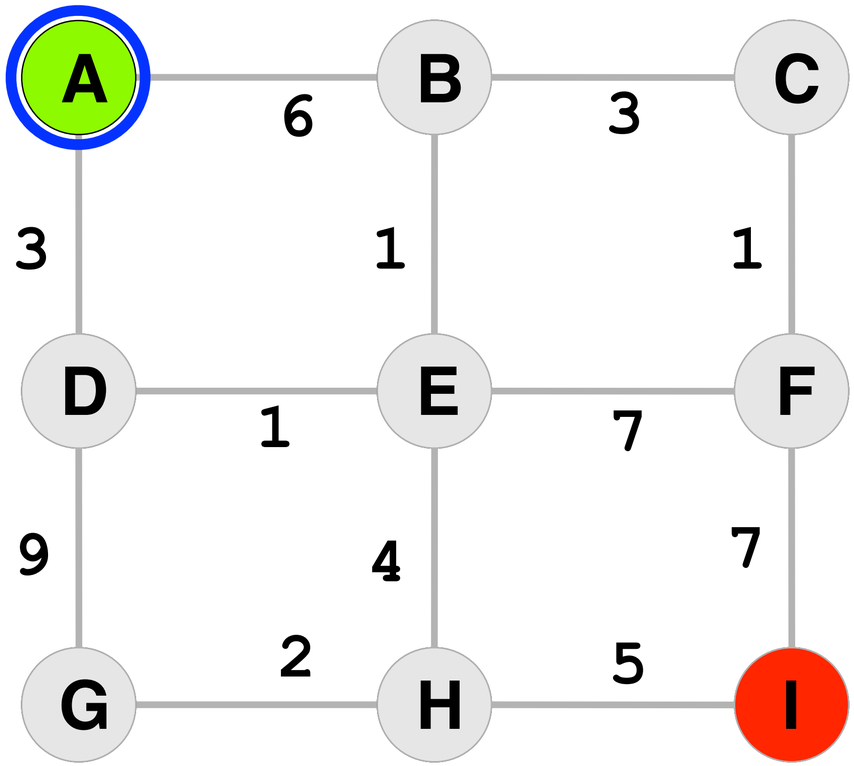
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B, H

Note: cannot stop yet! ADEHI might not be shortest!

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i: priority queue:

							front
Path:		ADEHI	ADG	ADEF	ADEHG	ADEBCF	ABC
Total Cost:		13	12	11	10	9	9

in while loop:
curPath = pq.dequeue() (path is ADEBC, priority is 8)
v = last element in curPath (v is C)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B, H, C

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:		ADEHI	ADG	ADEF	ABCF	ADEHG	ADEBCF
Total Cost:		13	12	11	10	10	9

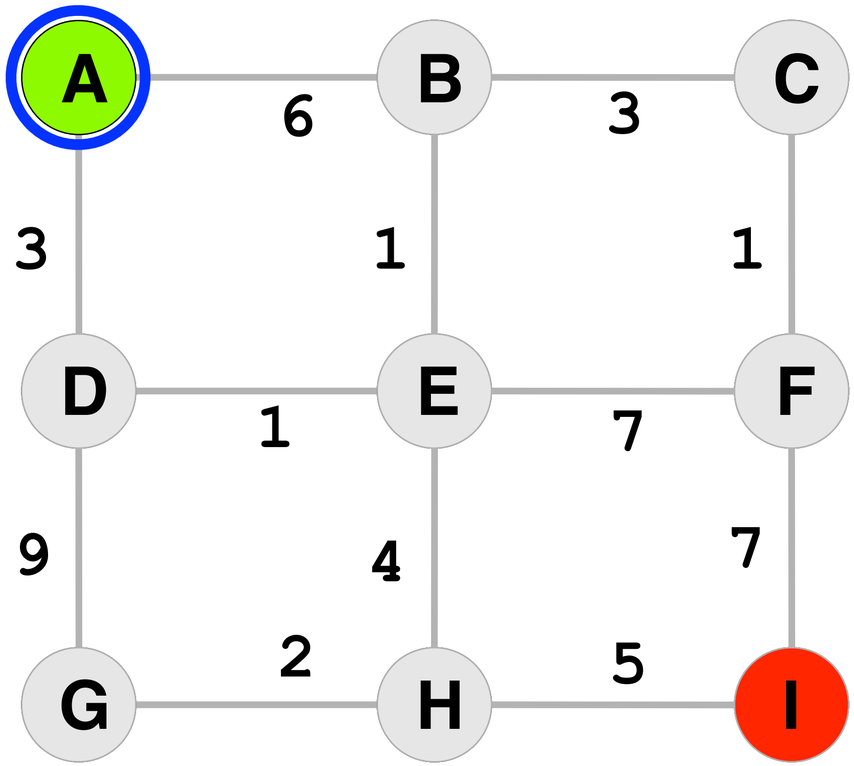
in while loop:

curPath = pq.dequeue() (path is ABC, priority is 9)

v = last element in curPath (v is C)

mark v as visited

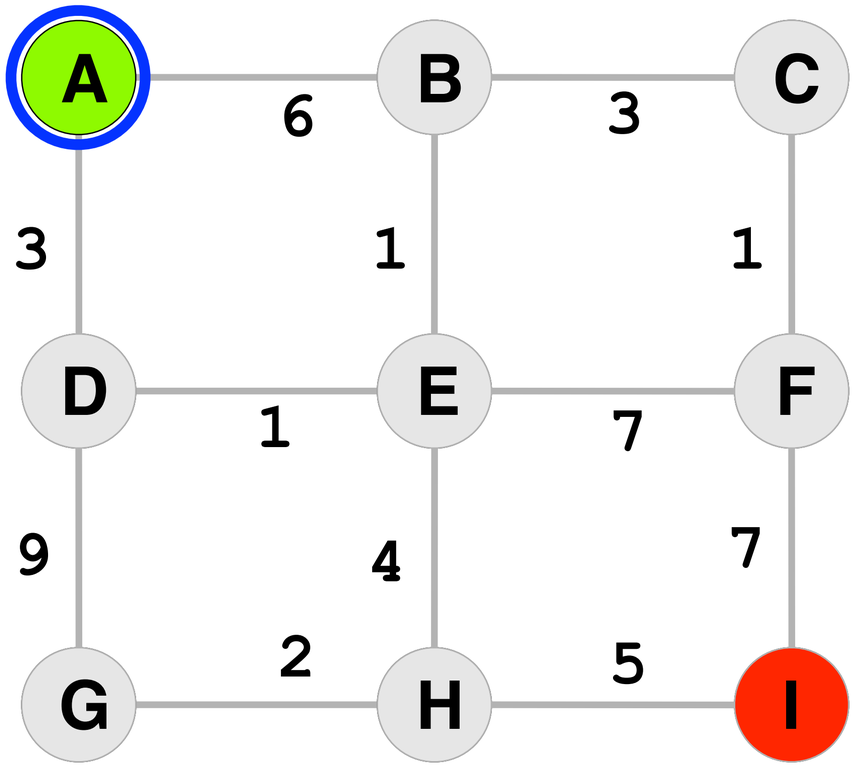
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length



Visited Set: A, D, E, B, H, C

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:		ADEBCFI	ADEHI	ADG	ADEF	ABCF	ADEHG
Total Cost:		16	13	12	11	10	10

in while loop:

curPath = pq.dequeue() (path is ADEBCF, priority is 9)

v = last element in curPath (v is F)

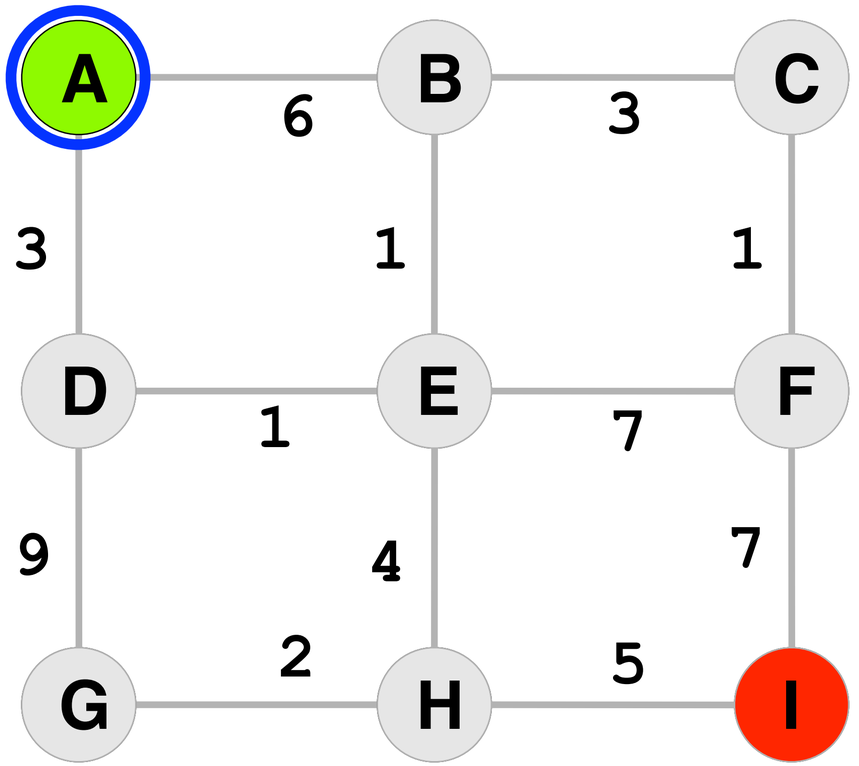
mark v as visited

enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B, H, C

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:			ADEBCFI	ADEHI	ADG	ADEF	ABCF
Total Cost:			16	13	12	11	10

in while loop:

curPath = pq.dequeue() (path is ADEHG, priority is 10)

v = last element in curPath (v is G)

mark v as visited

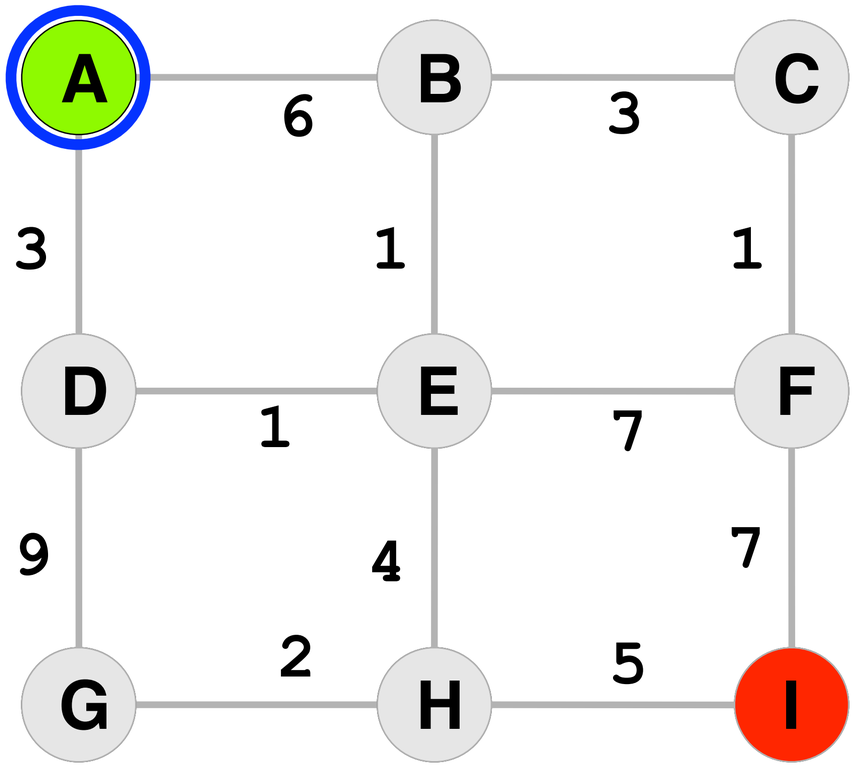
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

(nothing to enqueue, as all neighbors were visited)

Visited Set: A, D, E, B, H, C, F, G

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i: priority queue:

							front
Path:			ABCFI	ADEBCFI	ADEHI	ADG	ADEF
Total Cost:			17	16	13	12	11

in while loop:
curPath = pq.dequeue() (path is ABCF, priority is 10)
v = last element in curPath (v is F)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

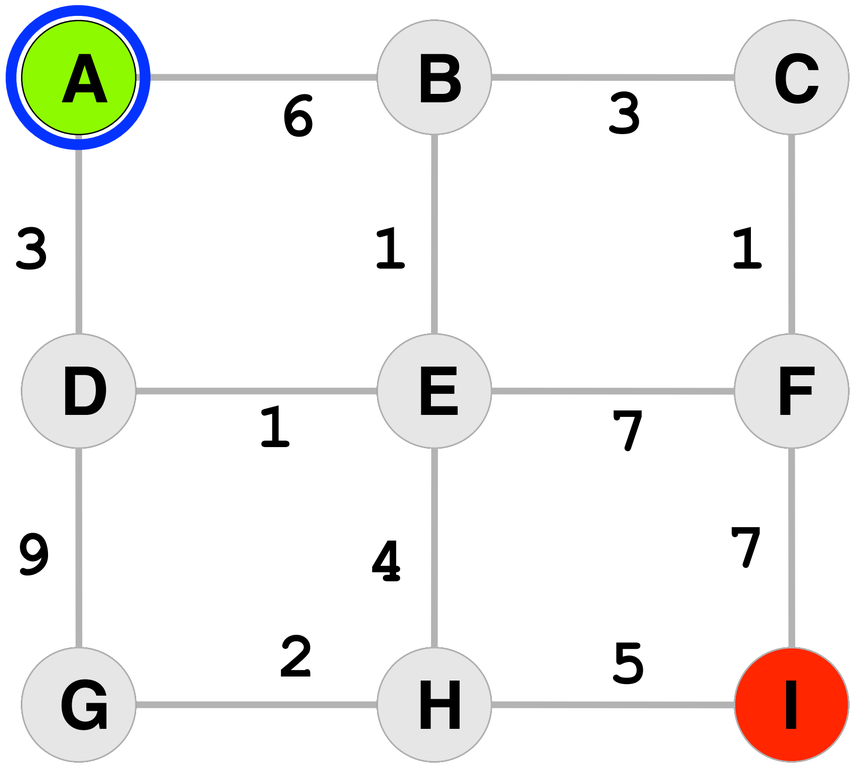
Visited Set: A, D, E, B, H, C, F, G

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:

Let's look at **Dijkstra** from a to i: priority queue:

							front
Path:			ADEFI	ABCFI	ADEBCFI	ADEHI	ADG
Total Cost:			18	17	16	13	12

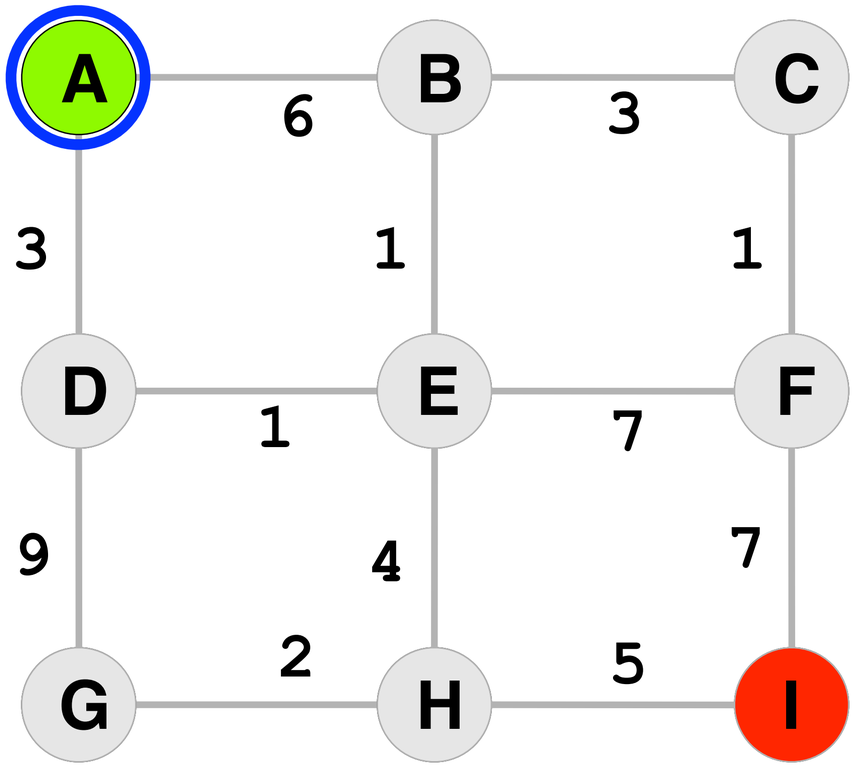


in while loop:
curPath = pq.dequeue() (path is ADEF, priority is 11)
v = last element in curPath (v is F)
mark v as visited
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

Visited Set: A, D, E, B, H, C, F, G

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:				ADEFI	ABCFI	ADEBCFI	ADEHI
Total Cost:				18	17	16	13

in while loop:

curPath = pq.dequeue() (path is ADG, priority is 12)

v = last element in curPath (v is F)

mark v as visited

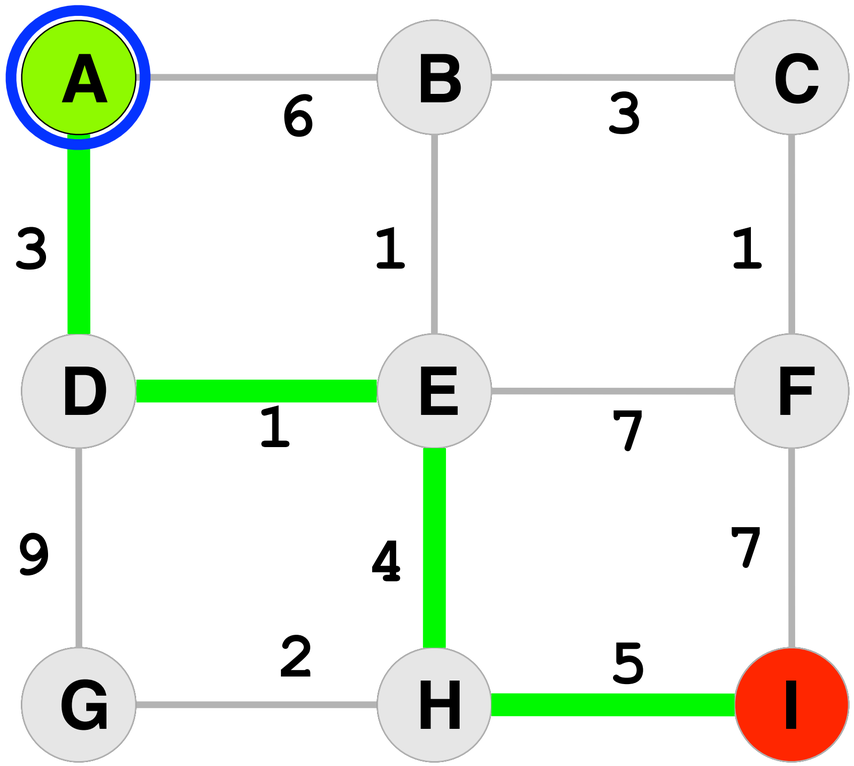
enqueue all unvisited neighbor paths onto q, with updated priorities based on new edge length

(nothing to enqueue, as all neighbors were visited)

Visited Set: A, D, E, B, H, C, F, G

Dijkstra's Algorithm in Practice

- From the start vertex, explore the neighbor nodes first, before moving to the next level neighbors, in priority order. From A to I:



Let's look at **Dijkstra** from a to i:

priority queue:

							front
Path:					ADEFI	ABCF	ADG
Total Cost:					18	16	12

in while loop:

curPath = pq.dequeue() (path is ADEHI, priority is 13)

v = last element in curPath (v is I)

Stop! We've found the shortest path!

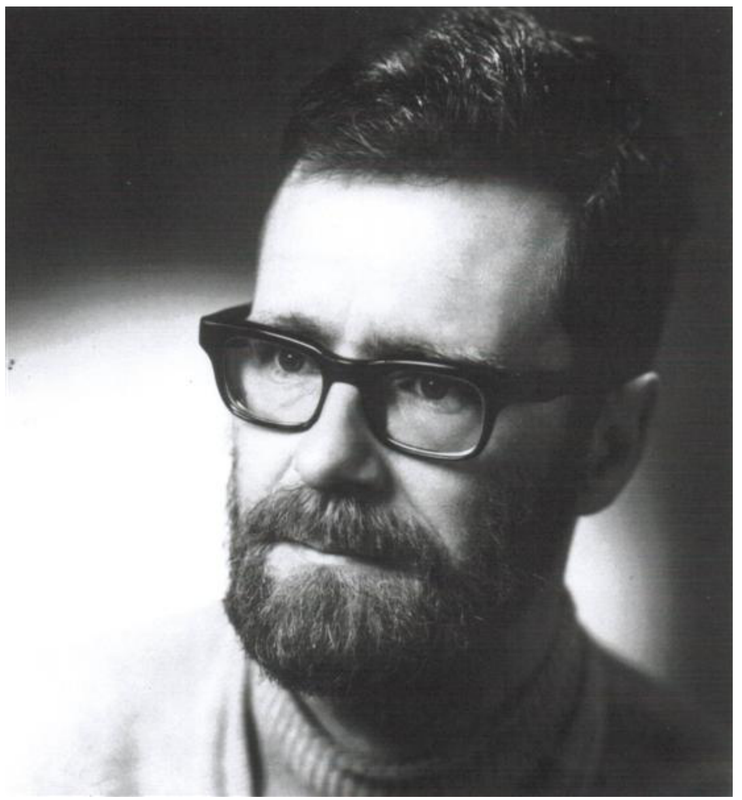
ADEHI

Visited Set: A, D, E, B, H, C, F

Who Was Edsgar Dijkstra?

History of Computing Tidbit: Edsger Dijkstra

- The Dutch academic Edsger Dijkstra was another giant in the field of computer science.
- He was one of the first scientists to call himself a "programmer" (and he almost couldn't get married because of it!)
- He started out with a degree in Theoretical Physics, but became enthralled with computers in the early 1950s.



Who Was Edsgar Dijkstra?

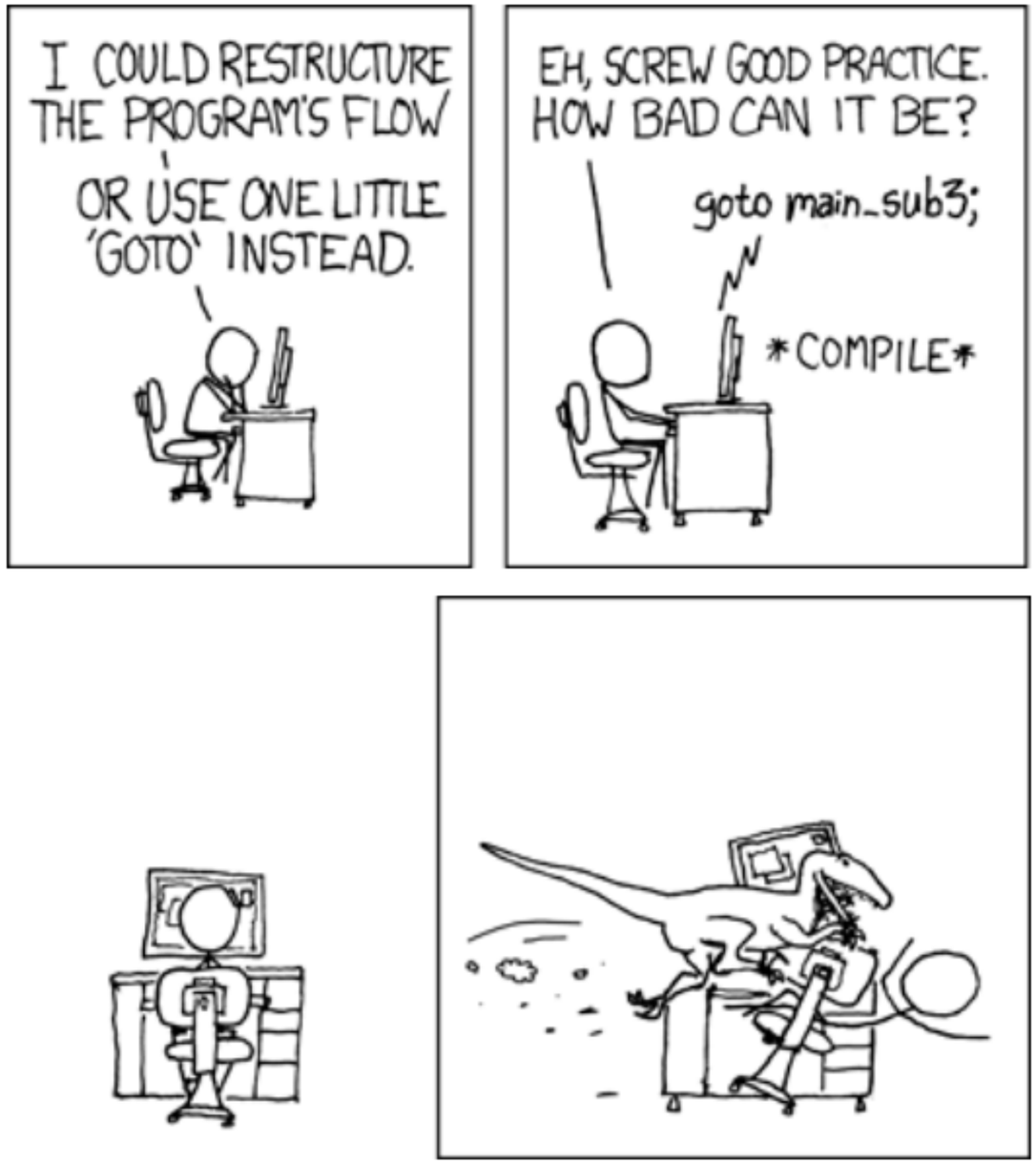
Edsger Dijkstra

- Dijkstra was immensely influential in many fields of computing: compilers, operating systems, concurrent programming, software engineering, programming languages, algorithm design, and teaching (among others!)
- It would be hard to pin down what he is most famous for because he has influenced so much CS.

Goto Considered Harmful

Edsger Dijkstra

- Dijkstra was also influential in making programming more structured -- he wrote a seminal paper titled, "Goto Considered Harmful" where he lambasted the idea of the "goto" statement (which exists in C++ -- you will rarely, if ever, use it!)



Other Cool Dijkstra Facts

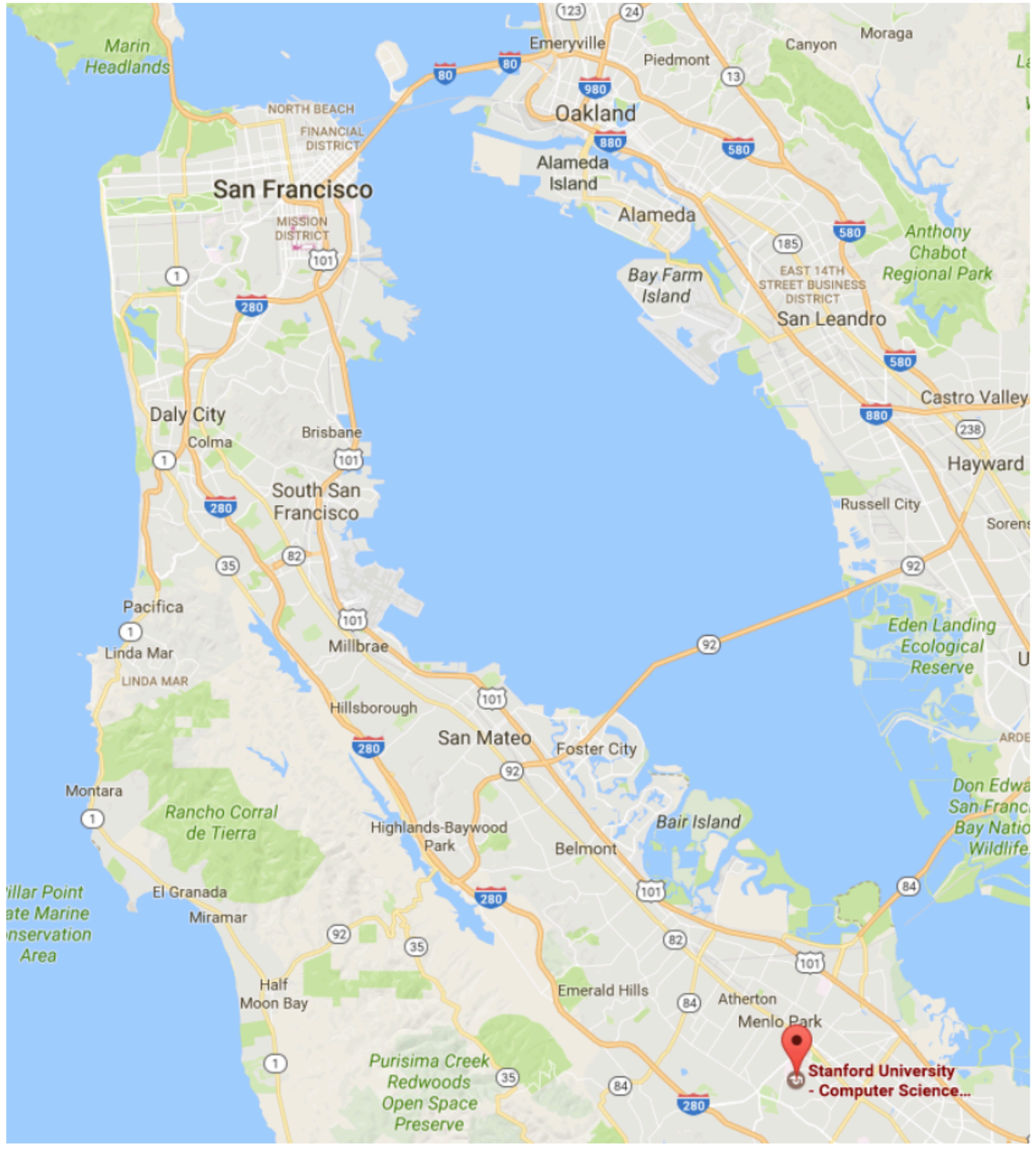
Other Reasons Dijkstra is cool:

- Might actually be Walter White
- Has the letters "ijk" adjacent in his name (is that why we use i,j,k in our loops??)
- The Edsgar Dijkstra font! His early papers were hand-written, and he had beautiful handwriting. This font is the "Edsger Dijkstra" font!

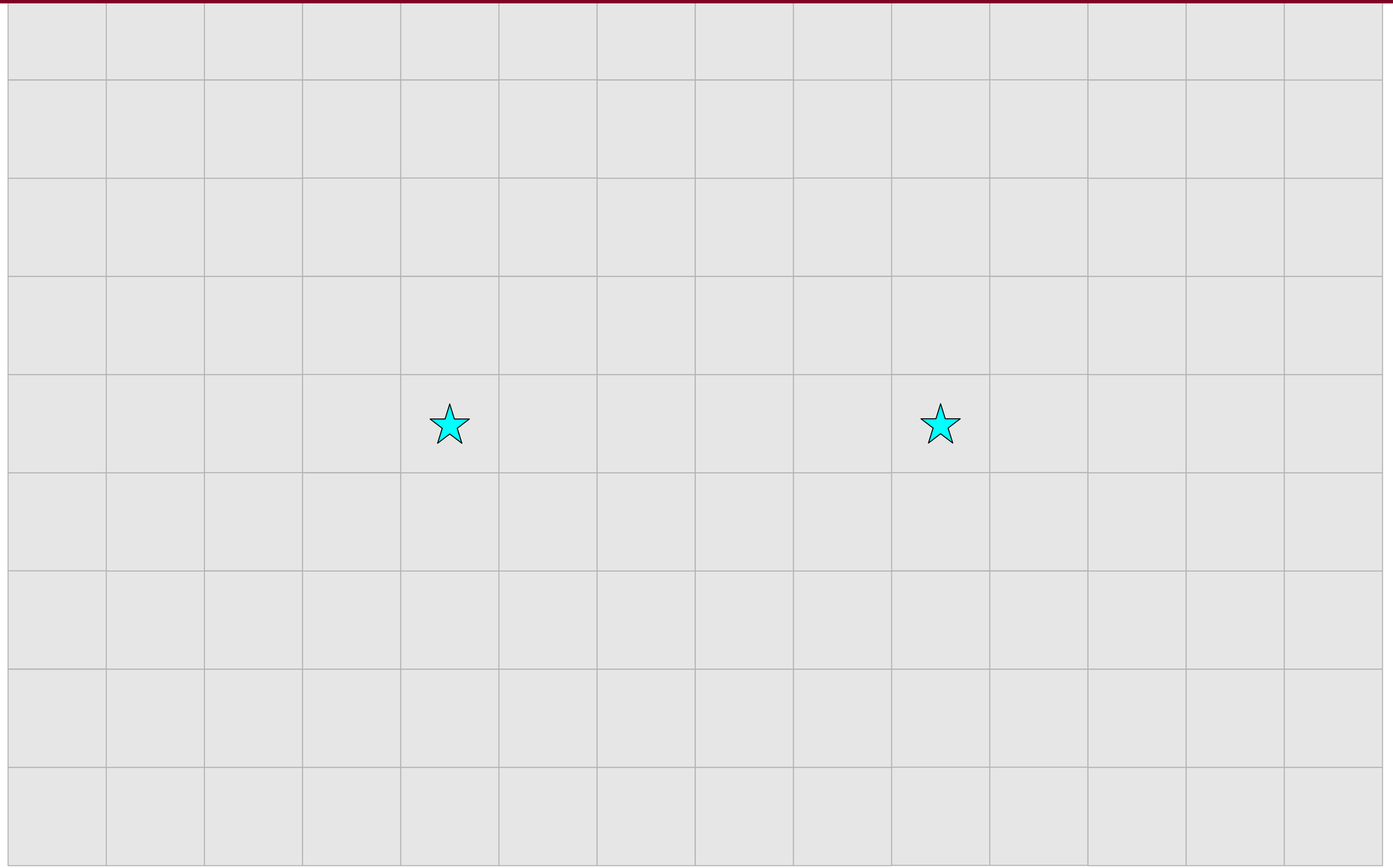
Dijkstra's is great, but we can do better!

If we want to travel from Stanford to San Francisco, Dijkstra's algorithm will look at path distances around Stanford. But, we know something about how to get to San Francisco -- we know that we generally need to go Northwest from Stanford.

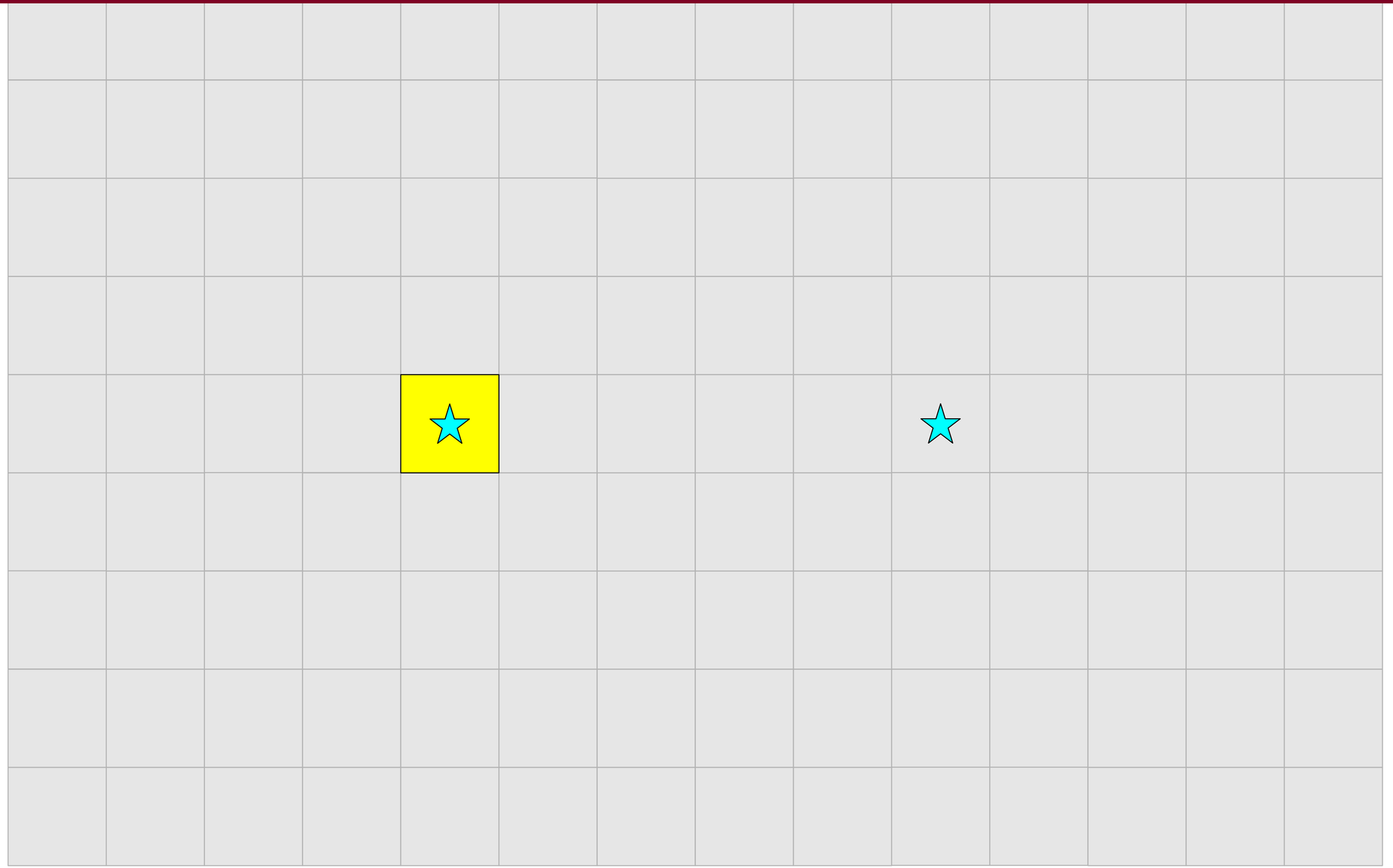
This is more information! Let's not only prioritize by weights, but also give some priority to the **direction** we want to go. E.g., we will **add more information based on a *heuristic*, which could be direction in the case of a street map.**



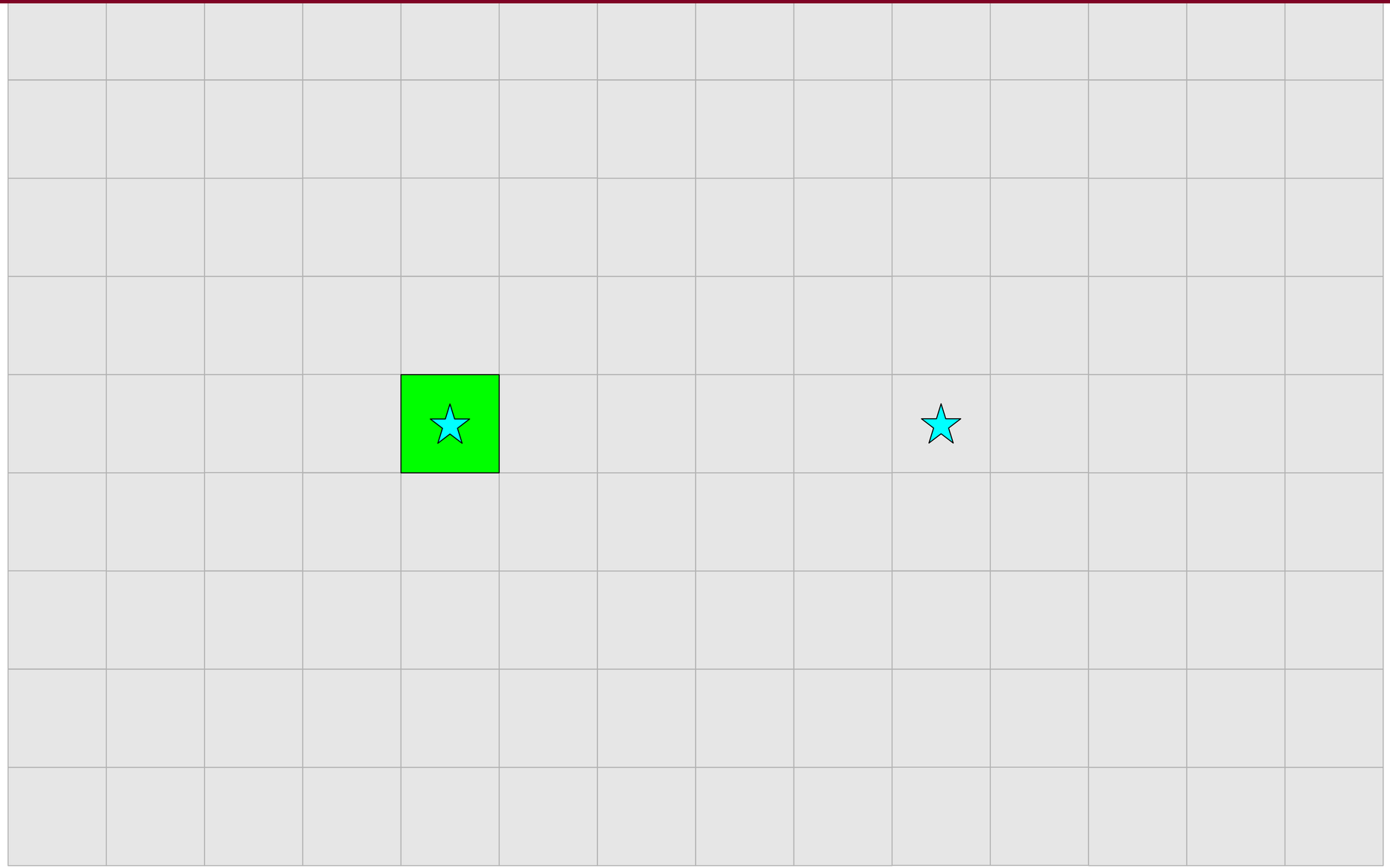
Let's look at Dijkstra where each edge has cost 1



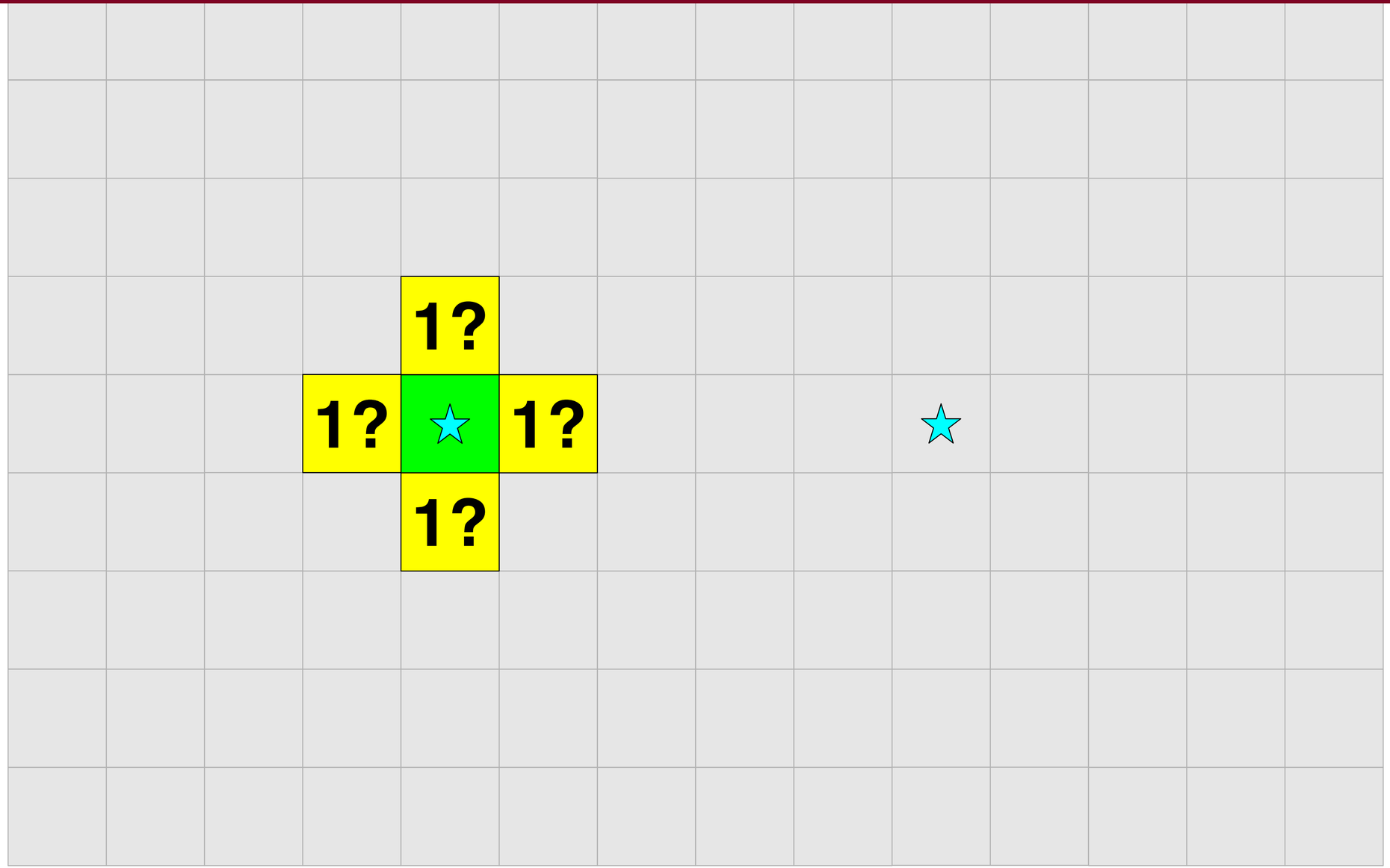
Dijkstra where each edge has cost 1



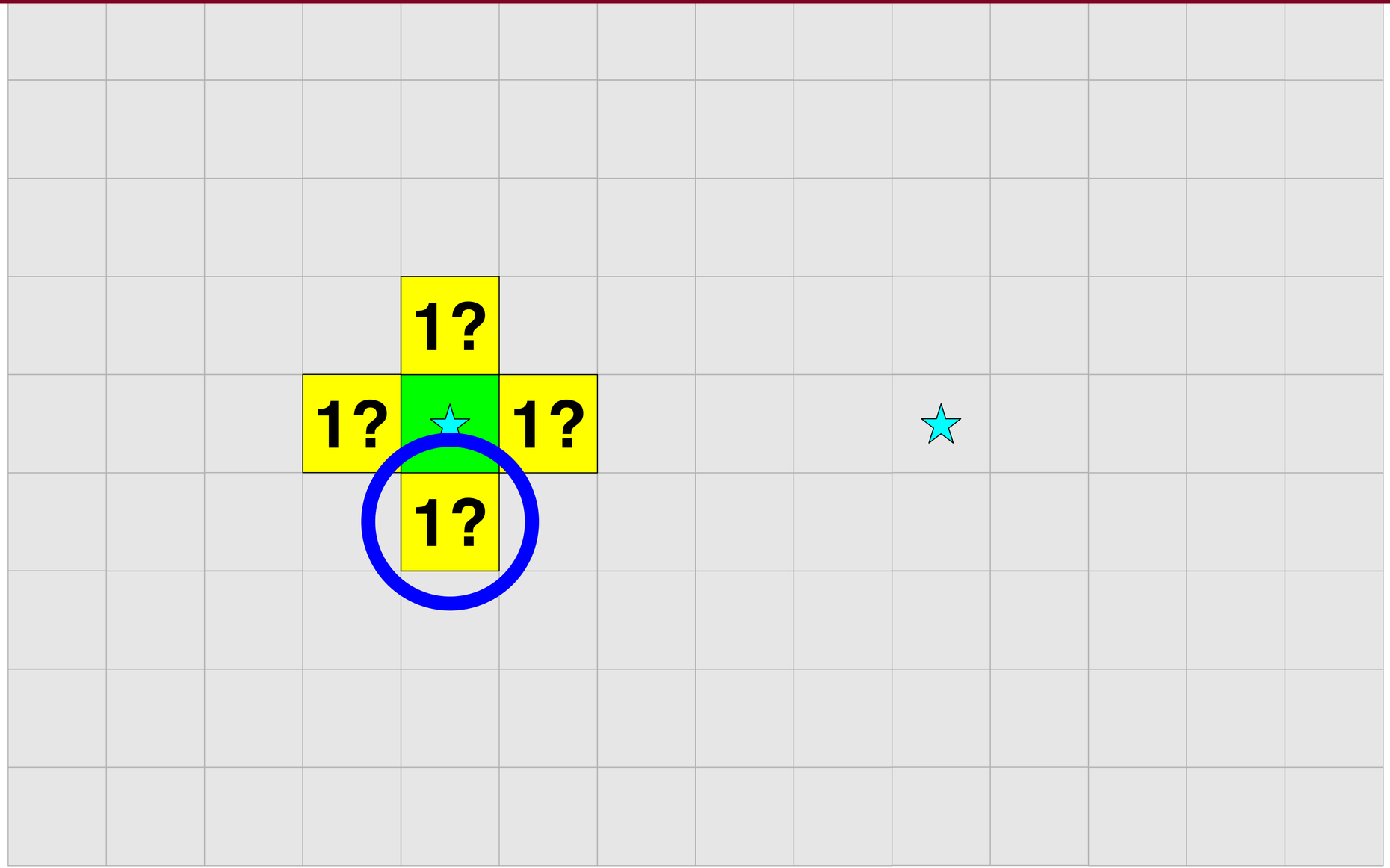
Dijkstra where each edge has cost 1



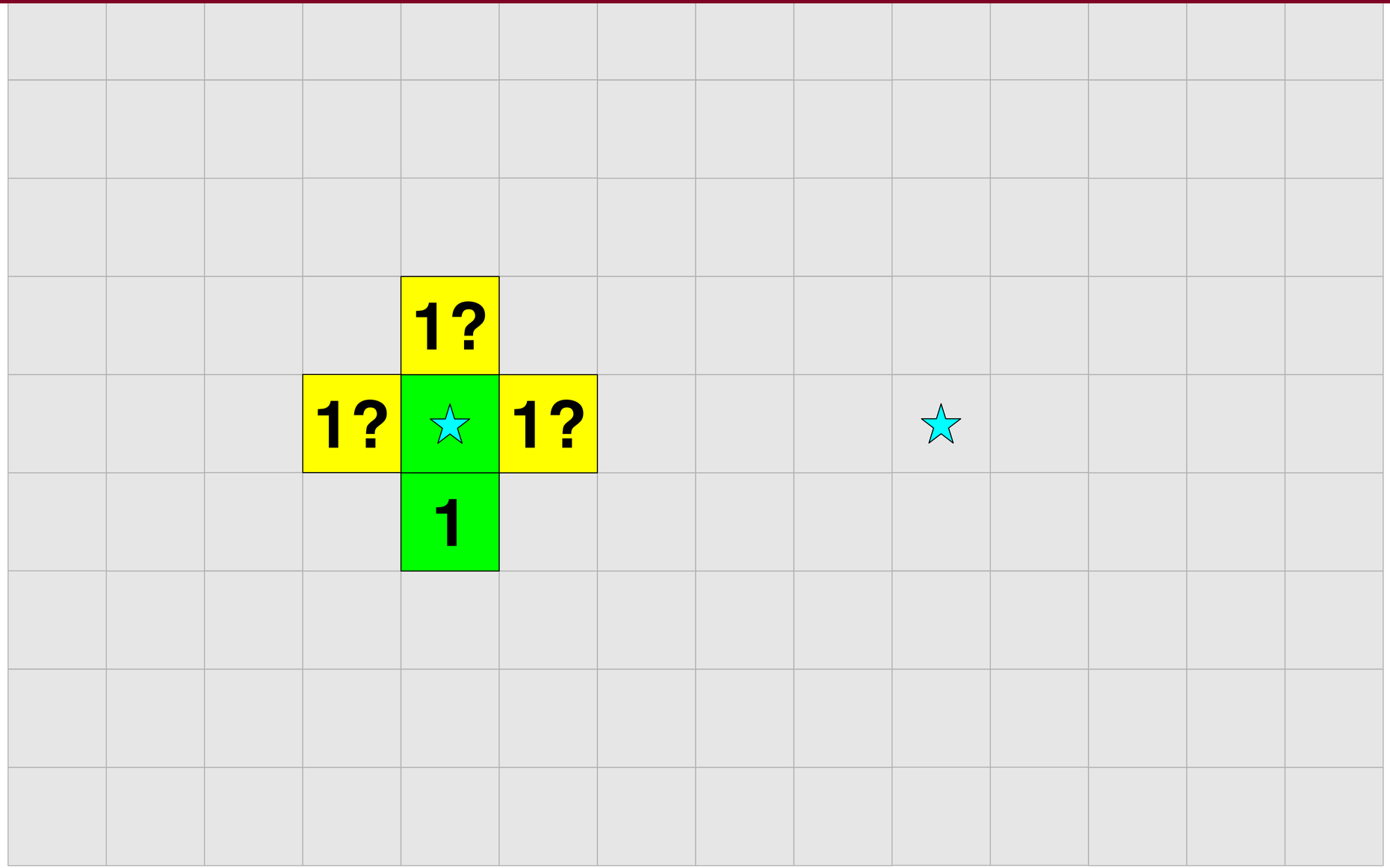
Dijkstra where each edge has cost 1



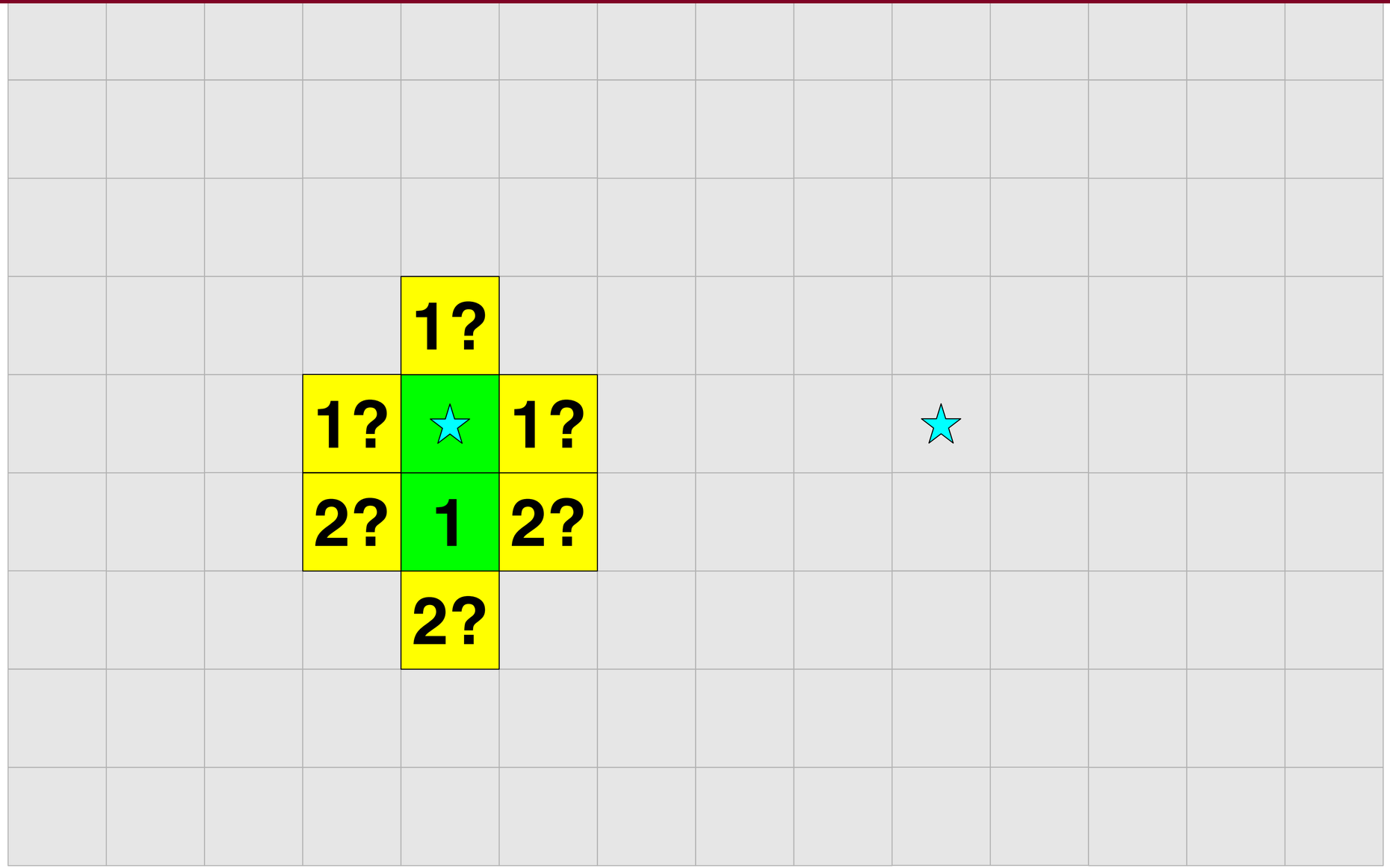
Dijkstra where each edge has cost 1



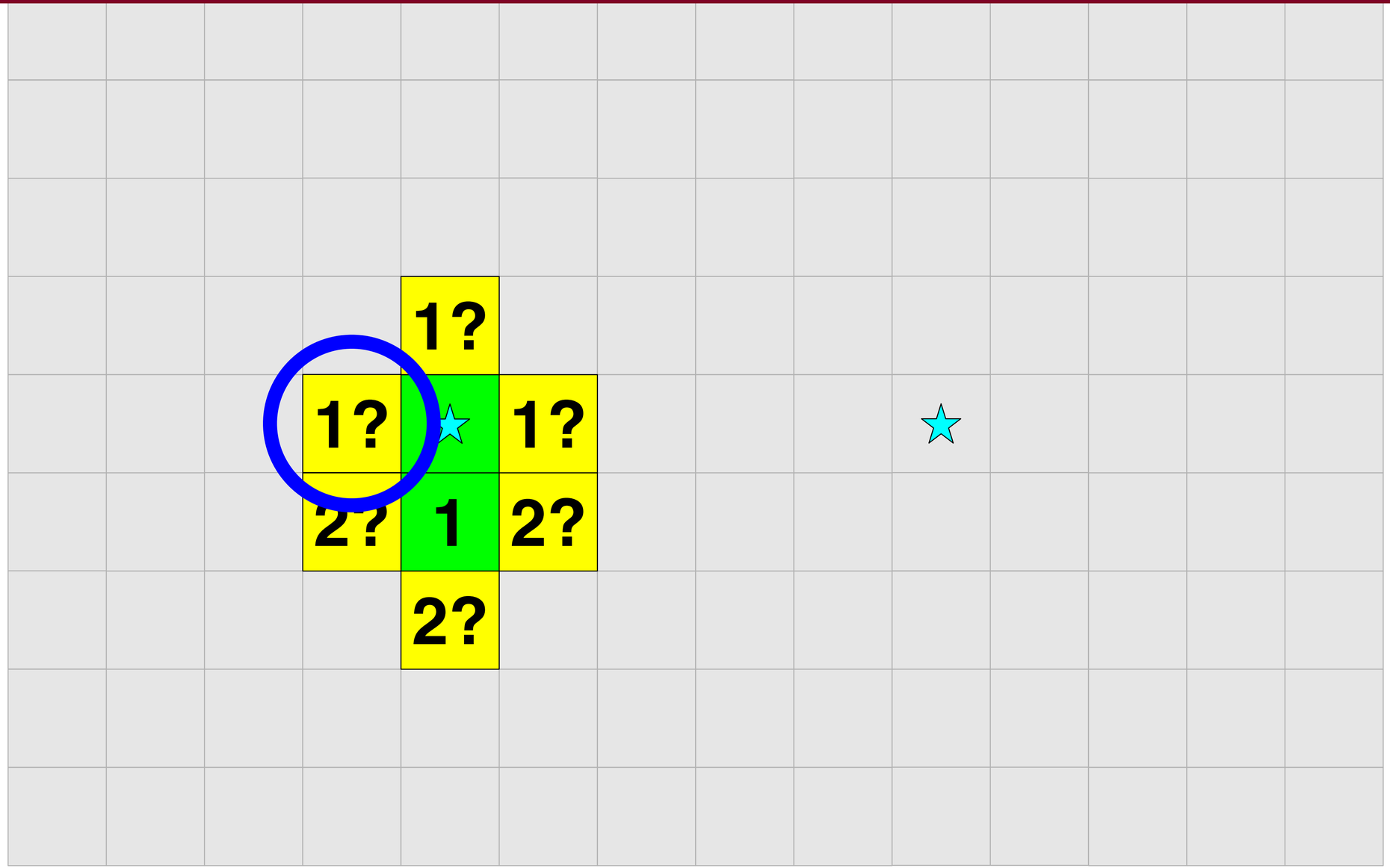
Dijkstra where each edge has cost 1



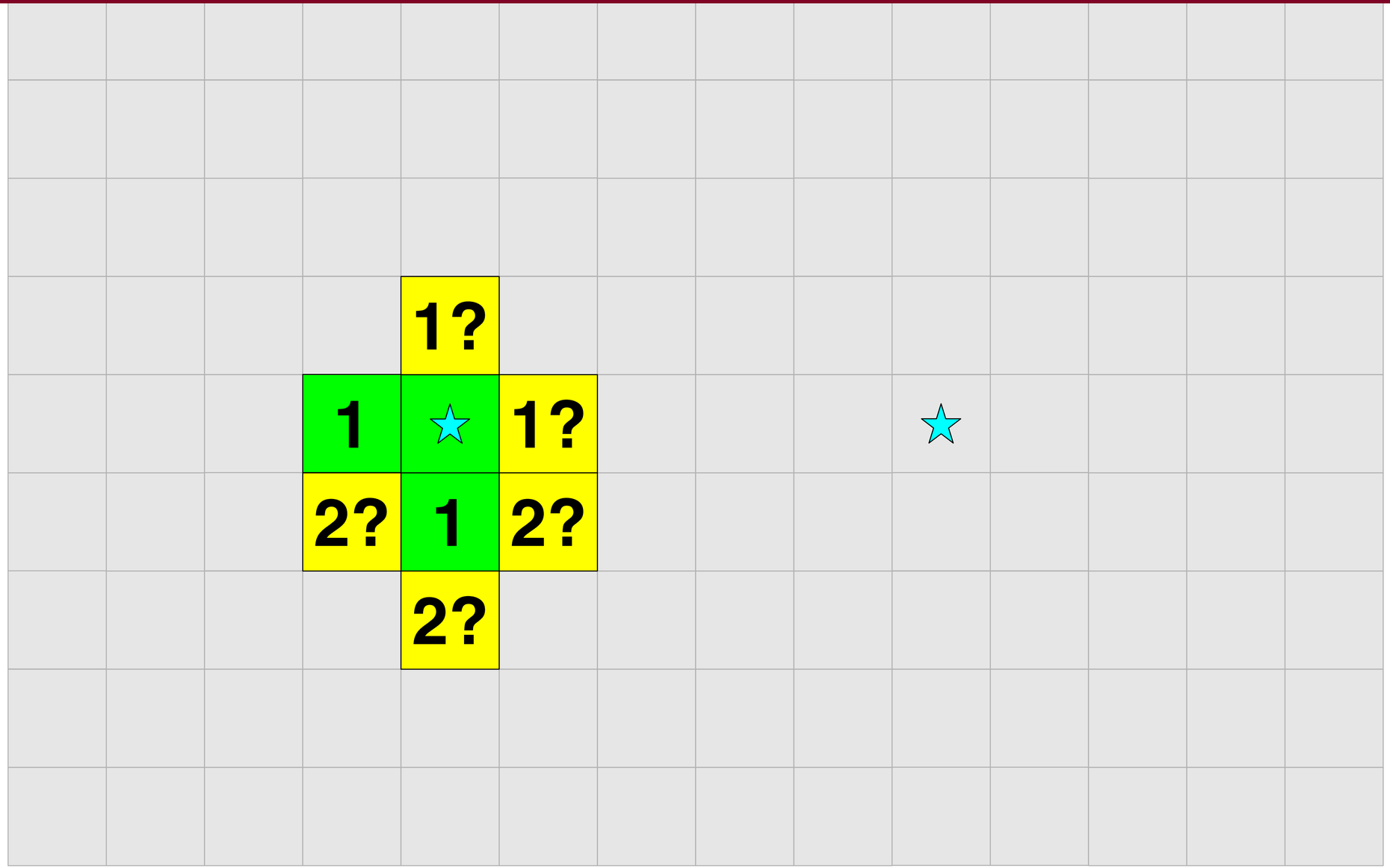
Dijkstra where each edge has cost 1



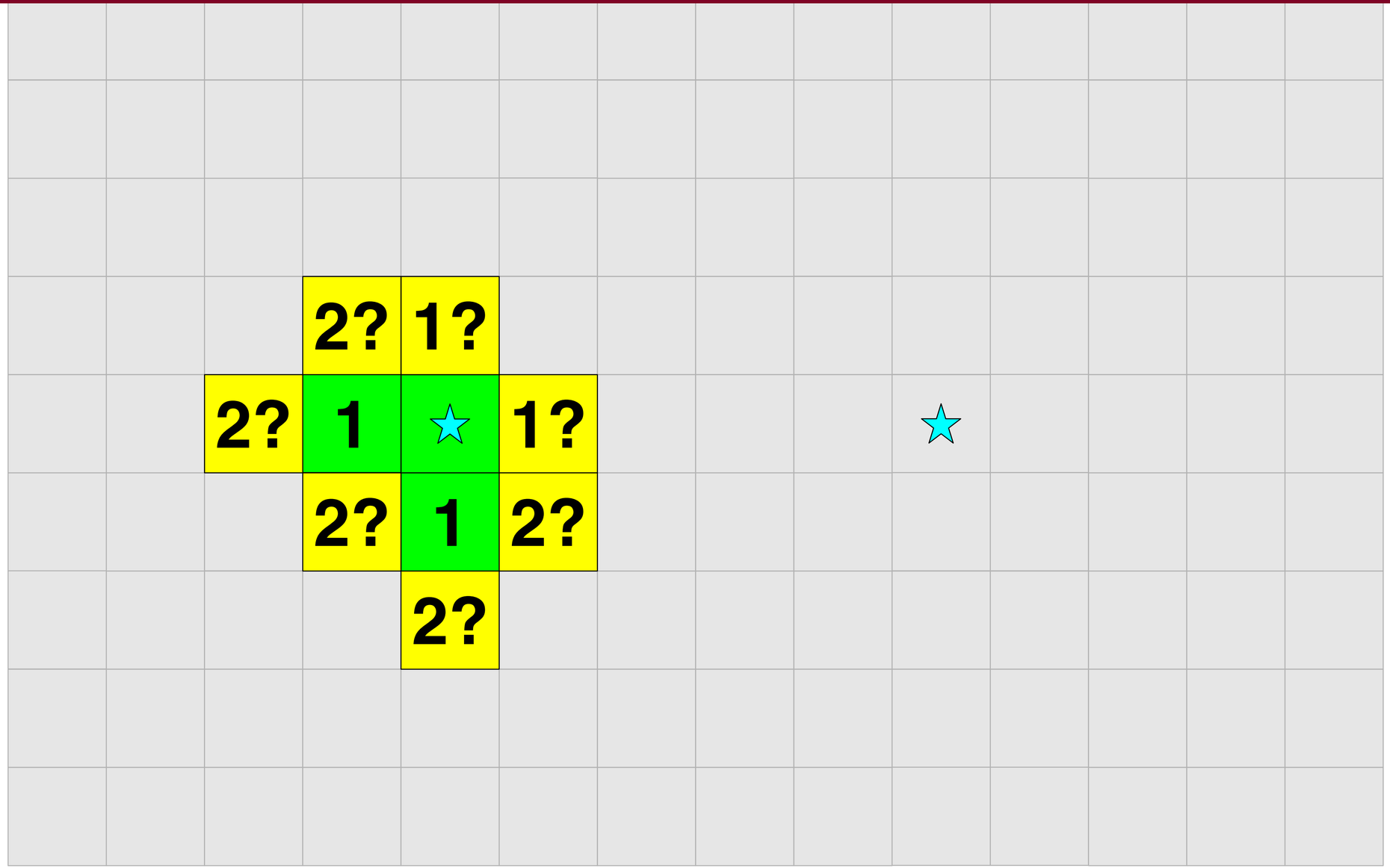
Dijkstra where each edge has cost 1



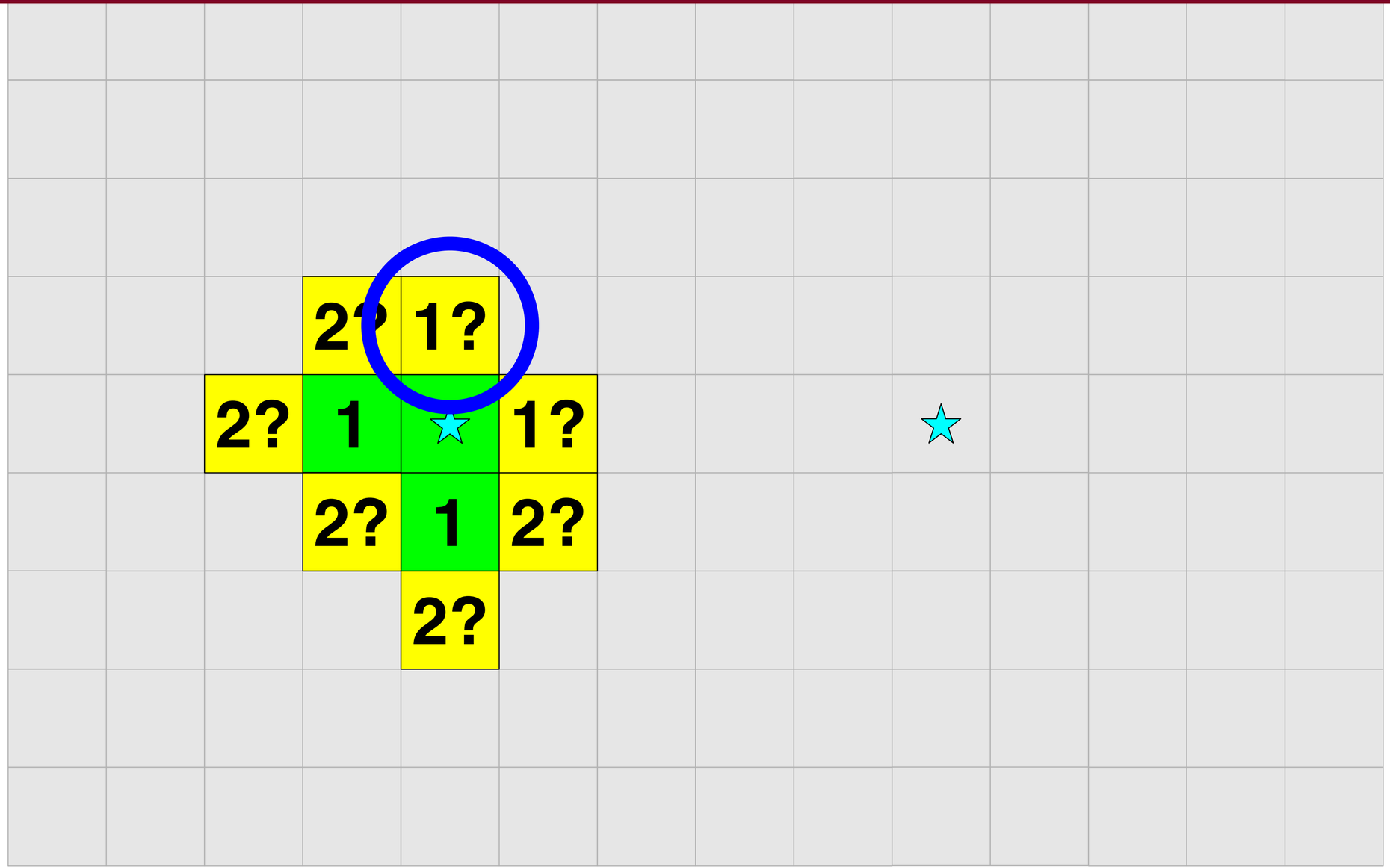
Dijkstra where each edge has cost 1



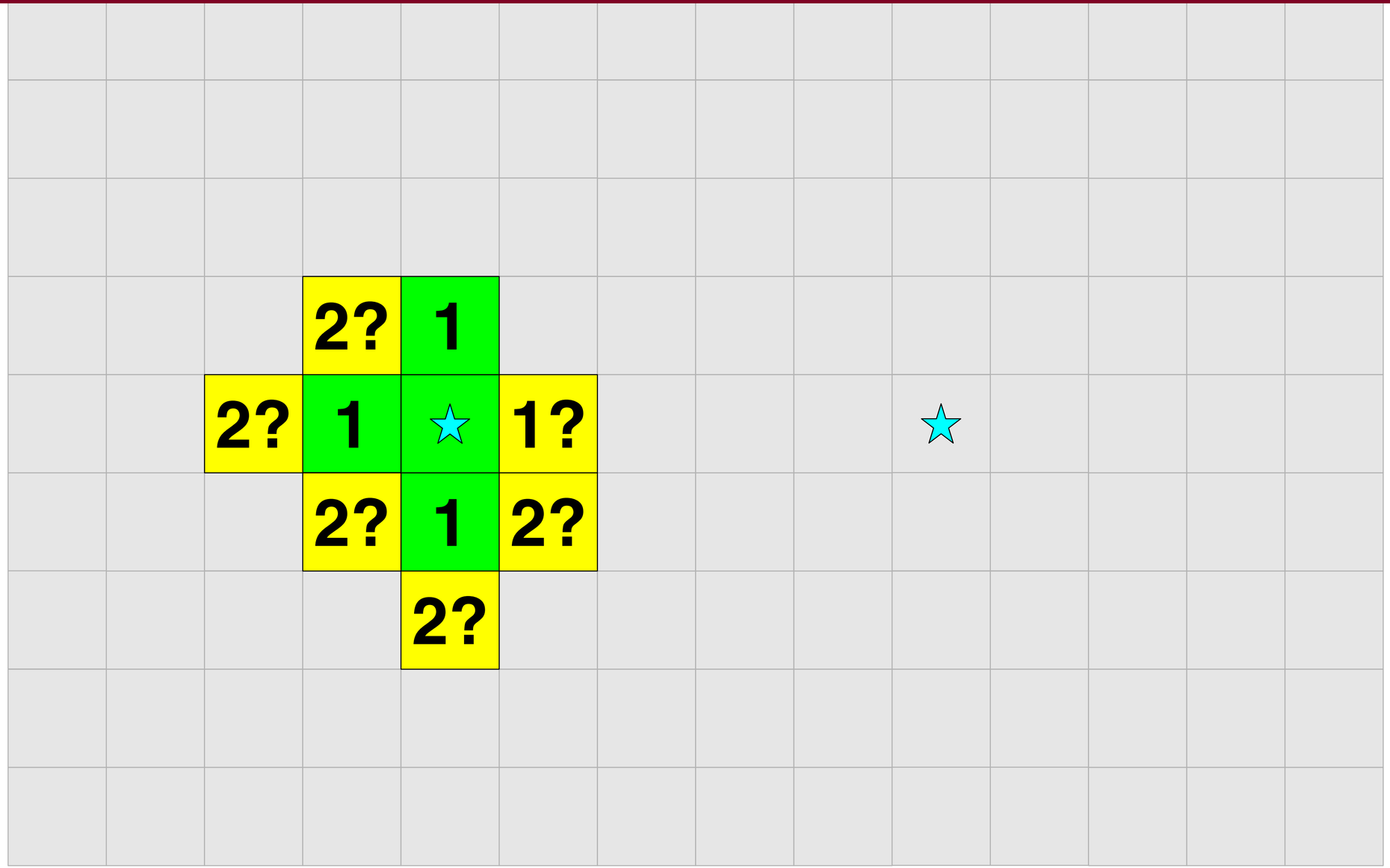
Dijkstra where each edge has cost 1



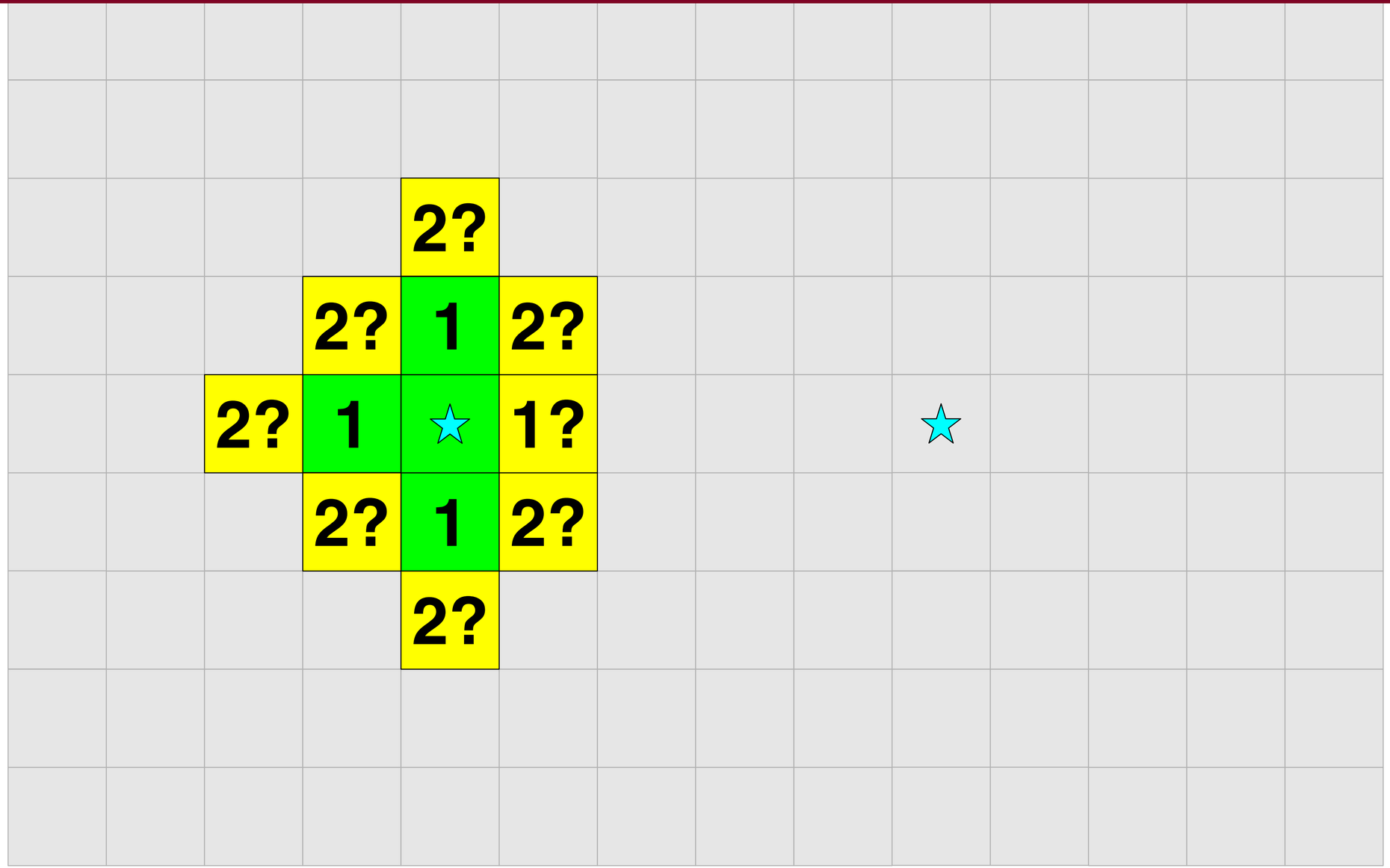
Dijkstra where each edge has cost 1



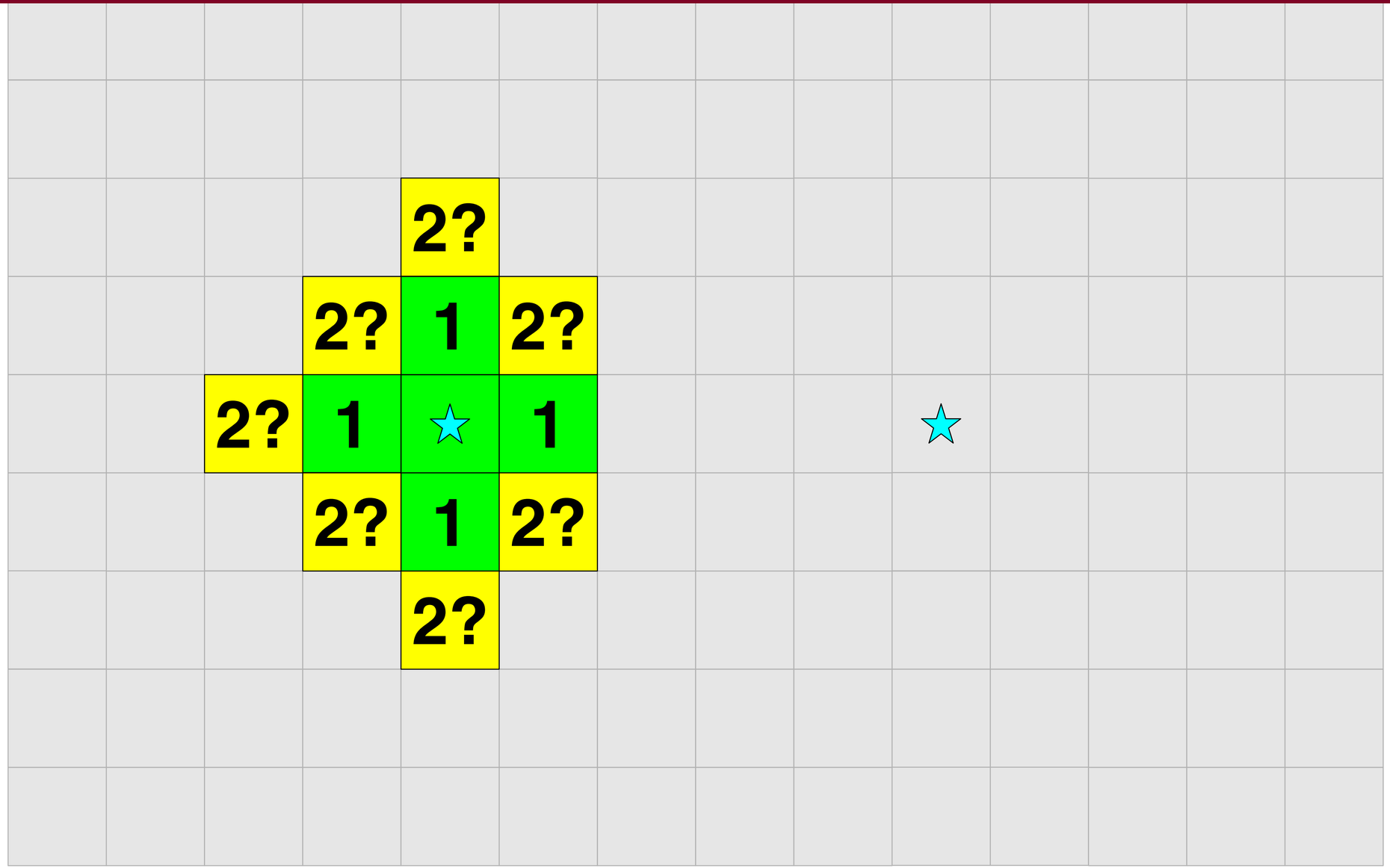
Dijkstra where each edge has cost 1



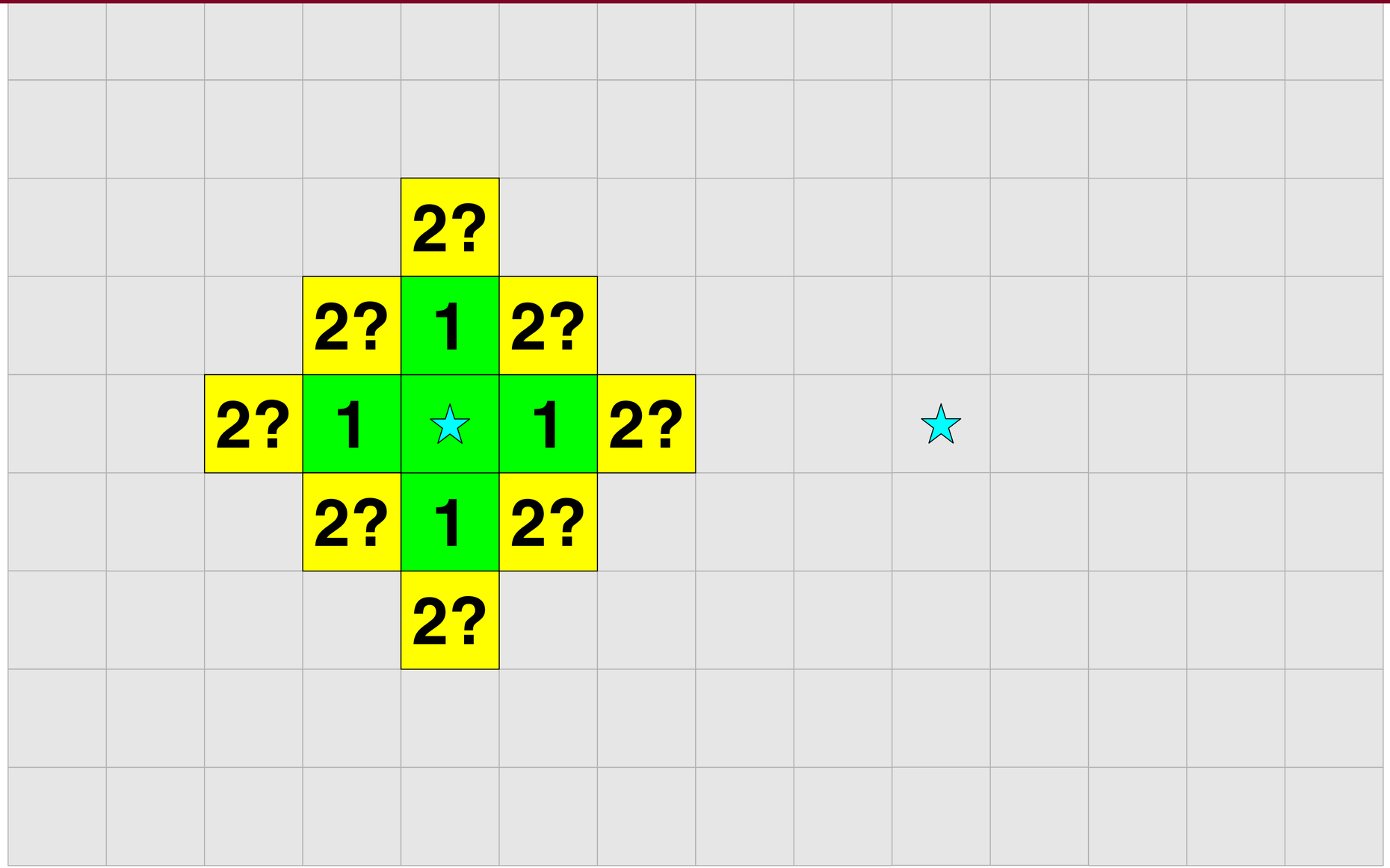
Dijkstra where each edge has cost 1



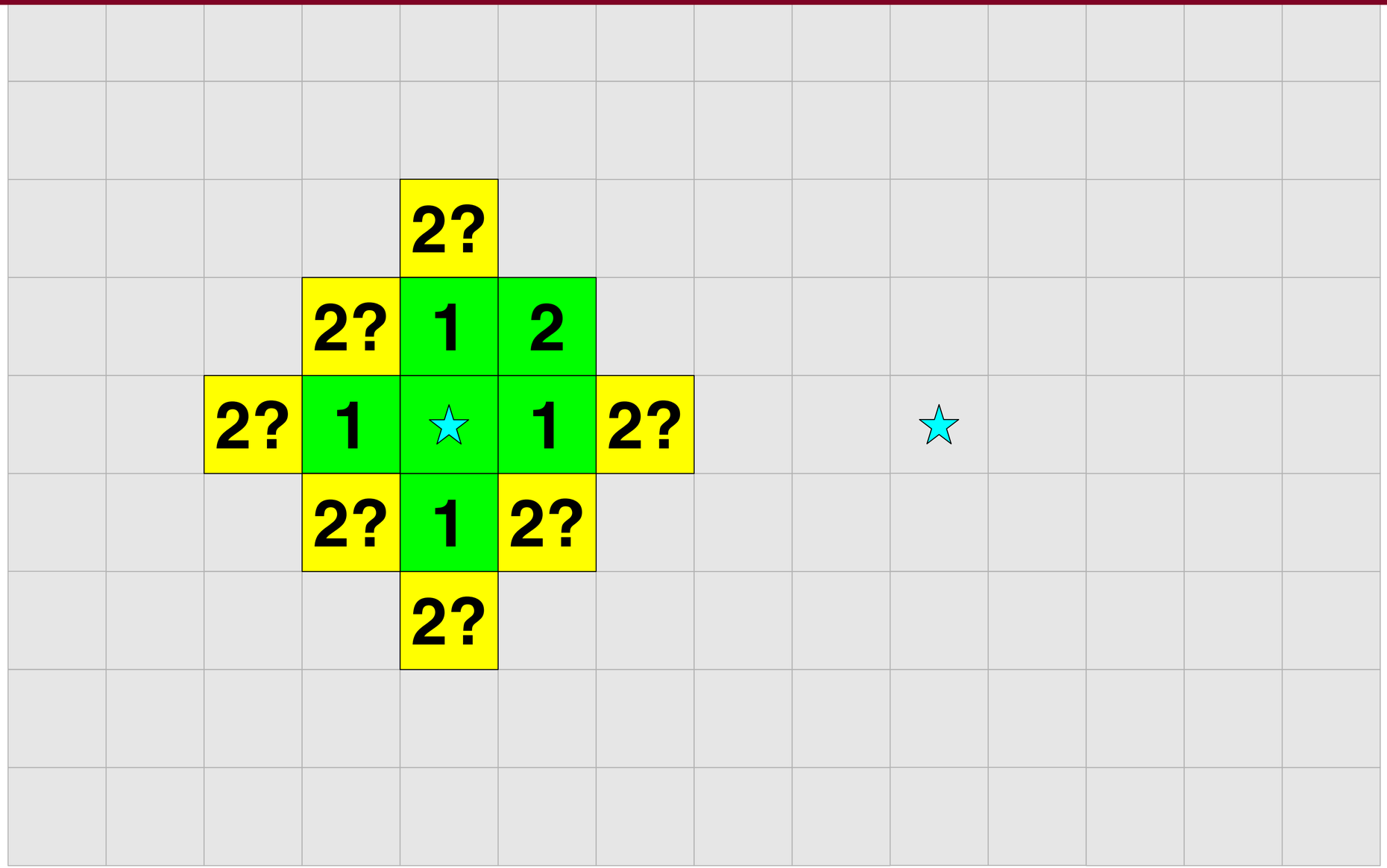
Dijkstra where each edge has cost 1



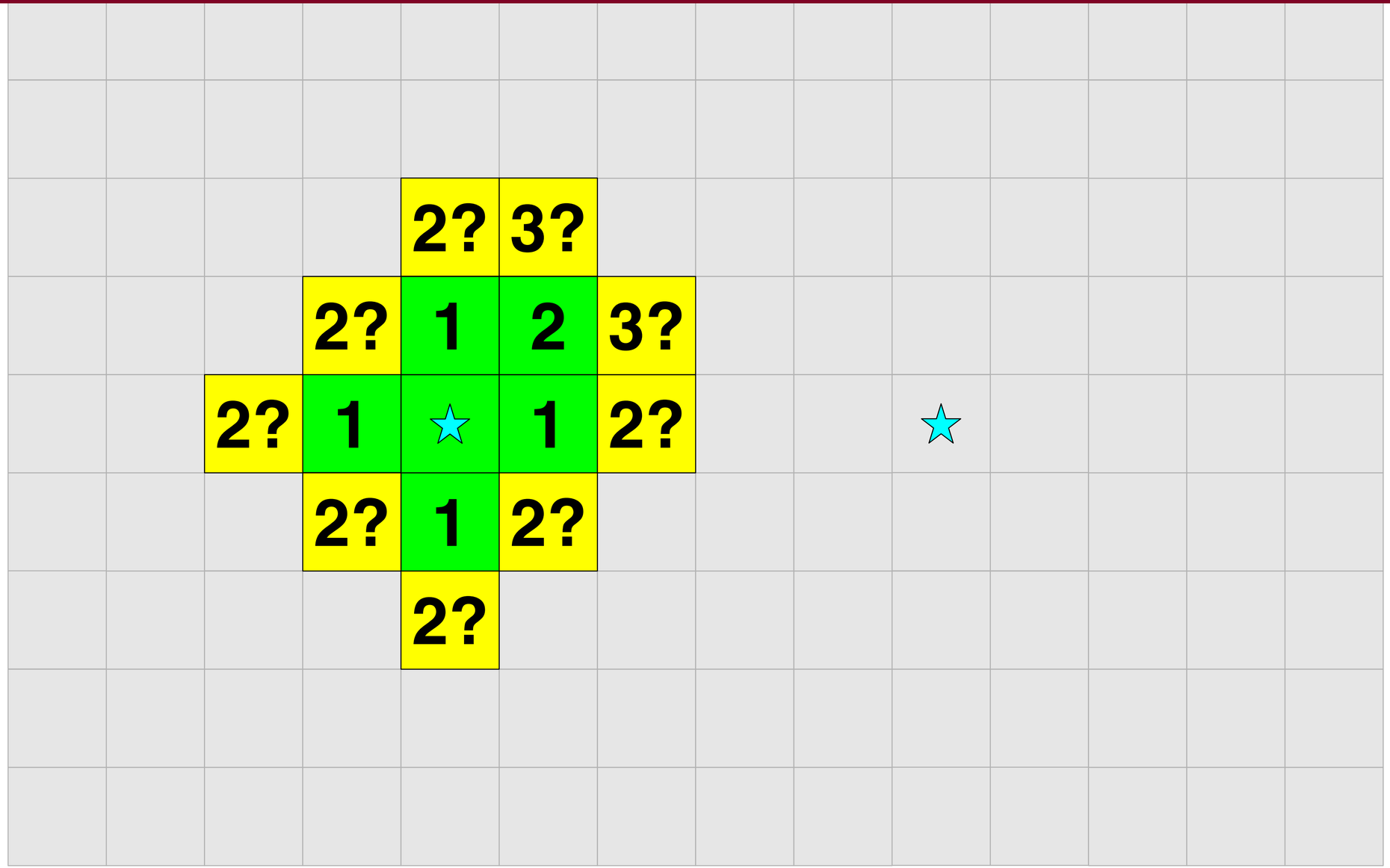
Dijkstra where each edge has cost 1



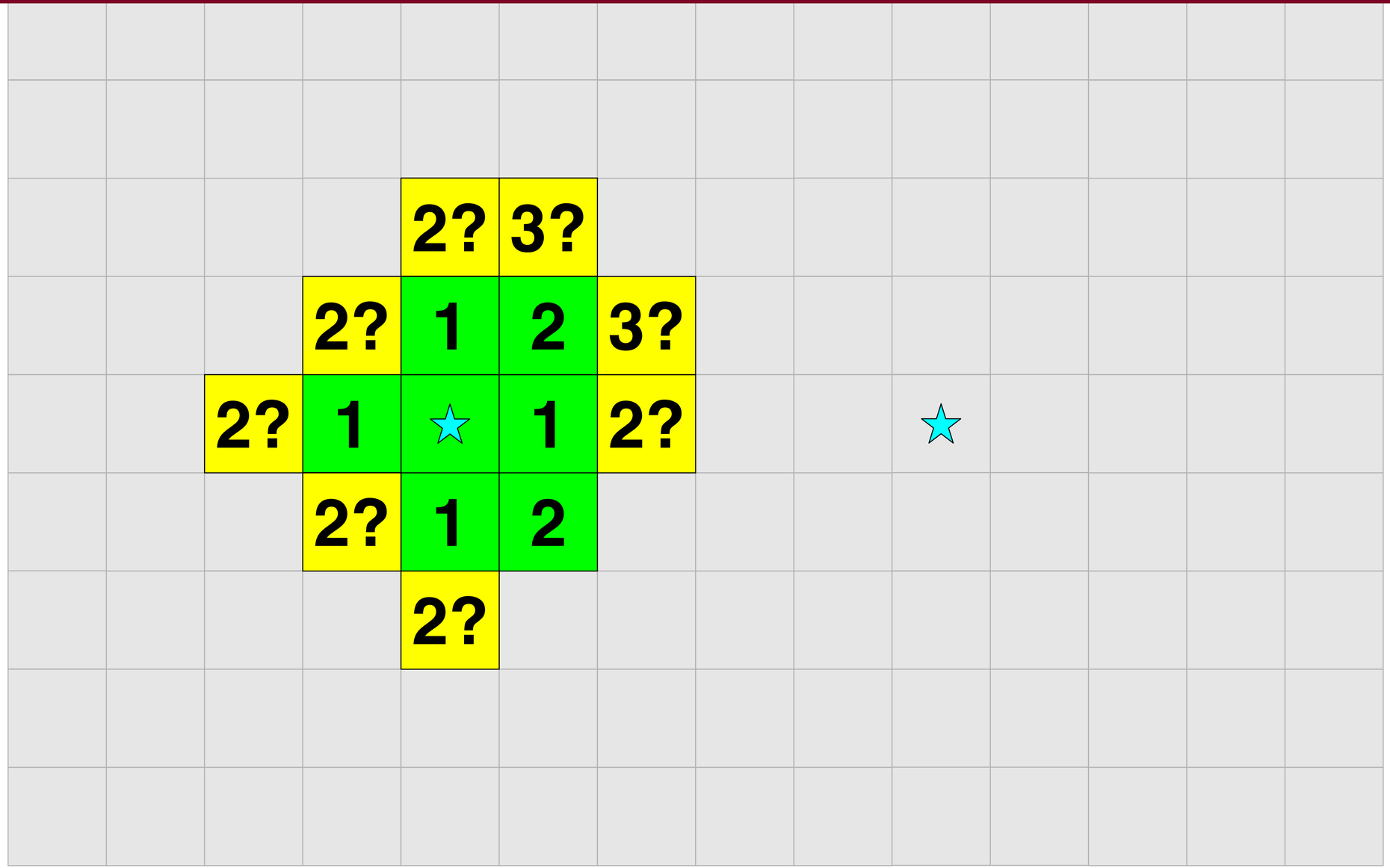
Dijkstra where each edge has cost 1



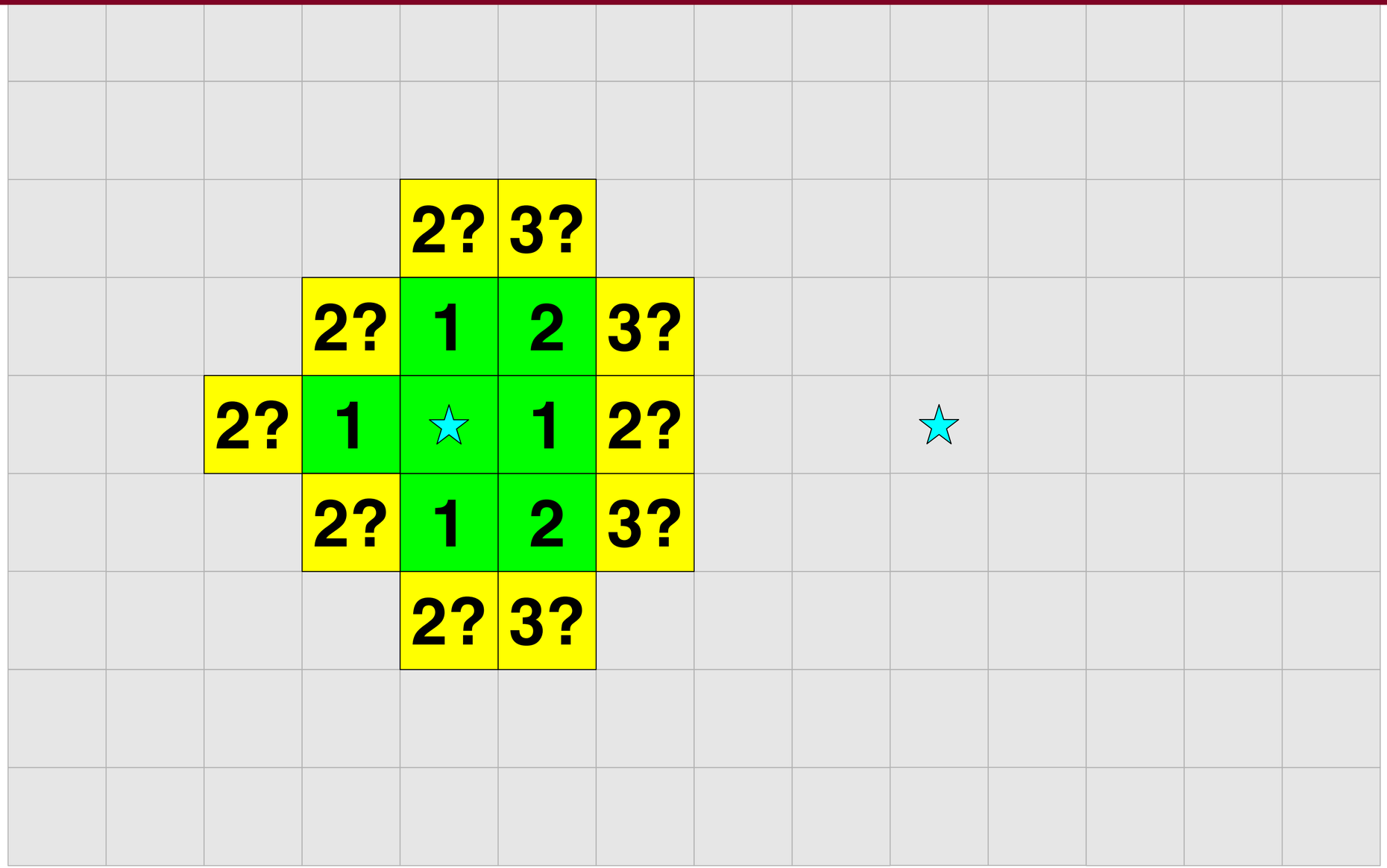
Dijkstra where each edge has cost 1



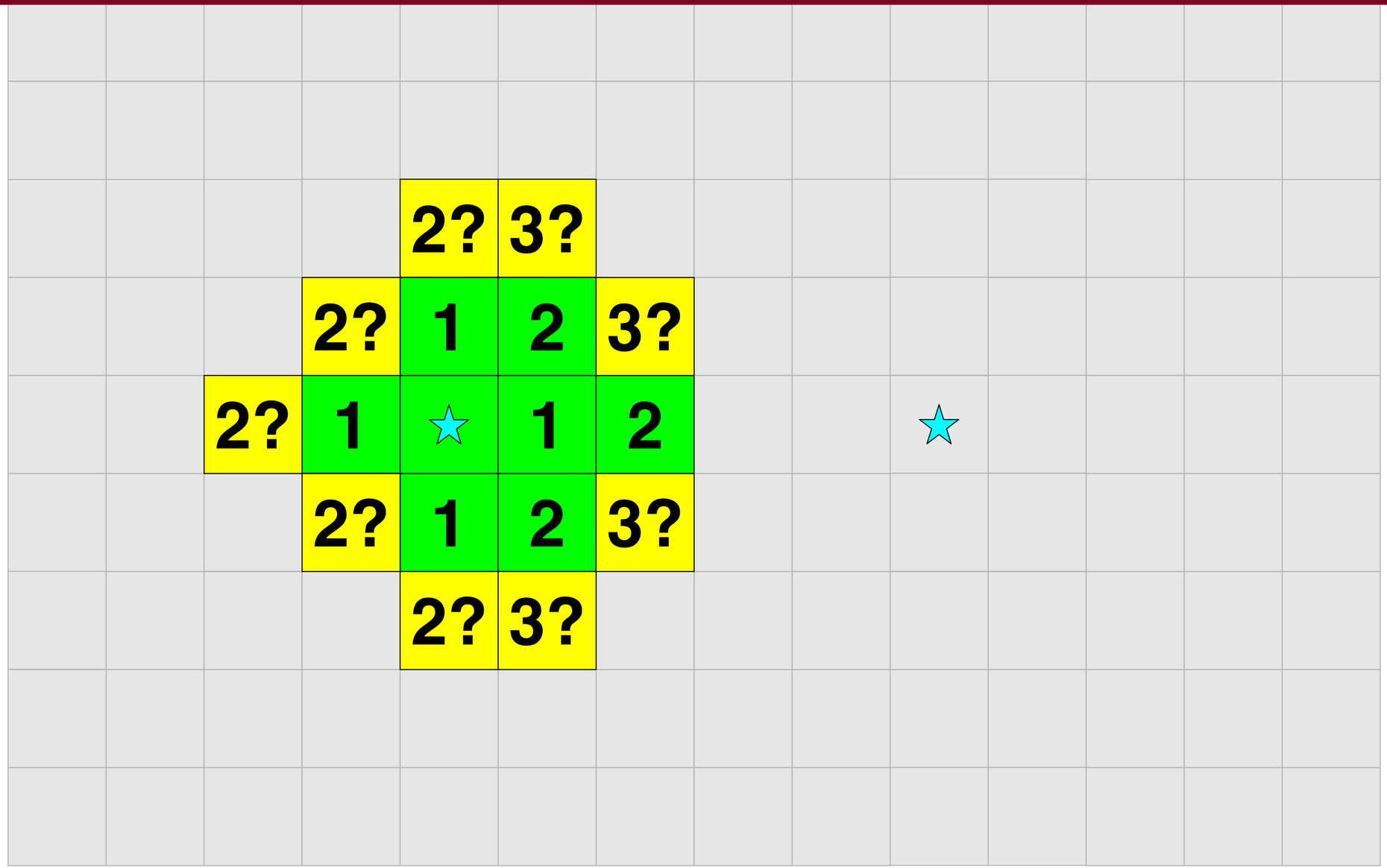
Dijkstra where each edge has cost 1



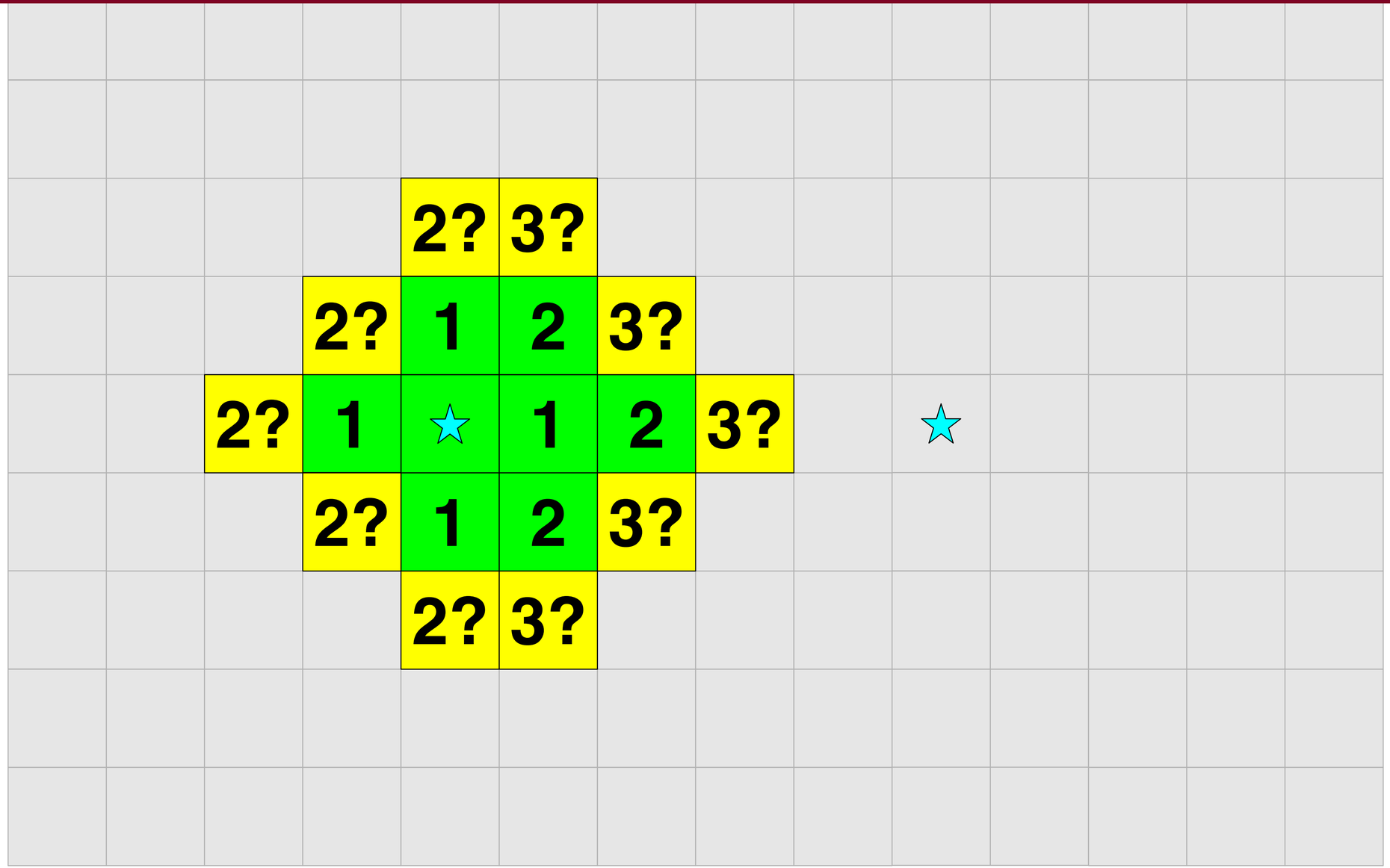
Dijkstra where each edge has cost 1



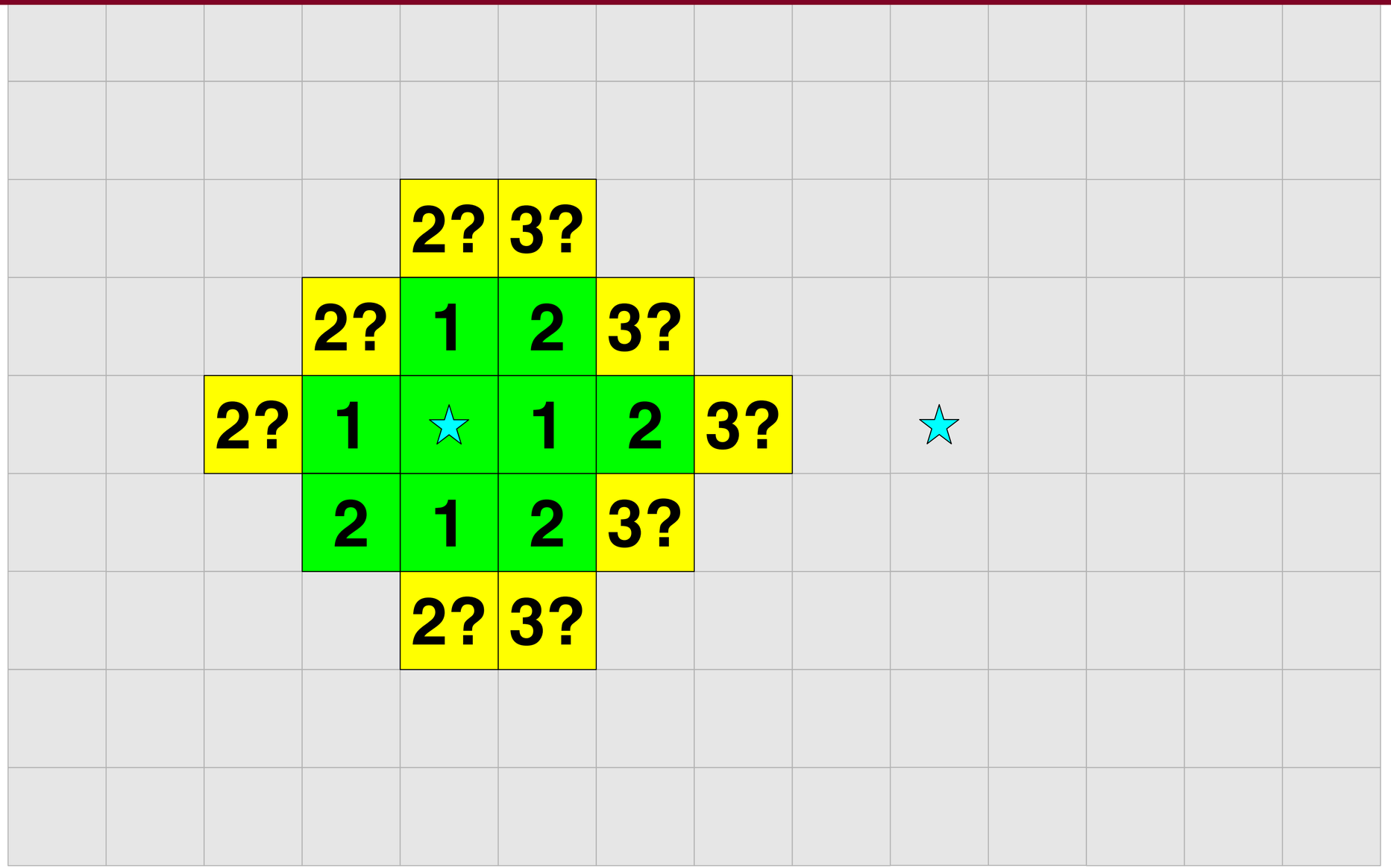
Dijkstra where each edge has cost 1



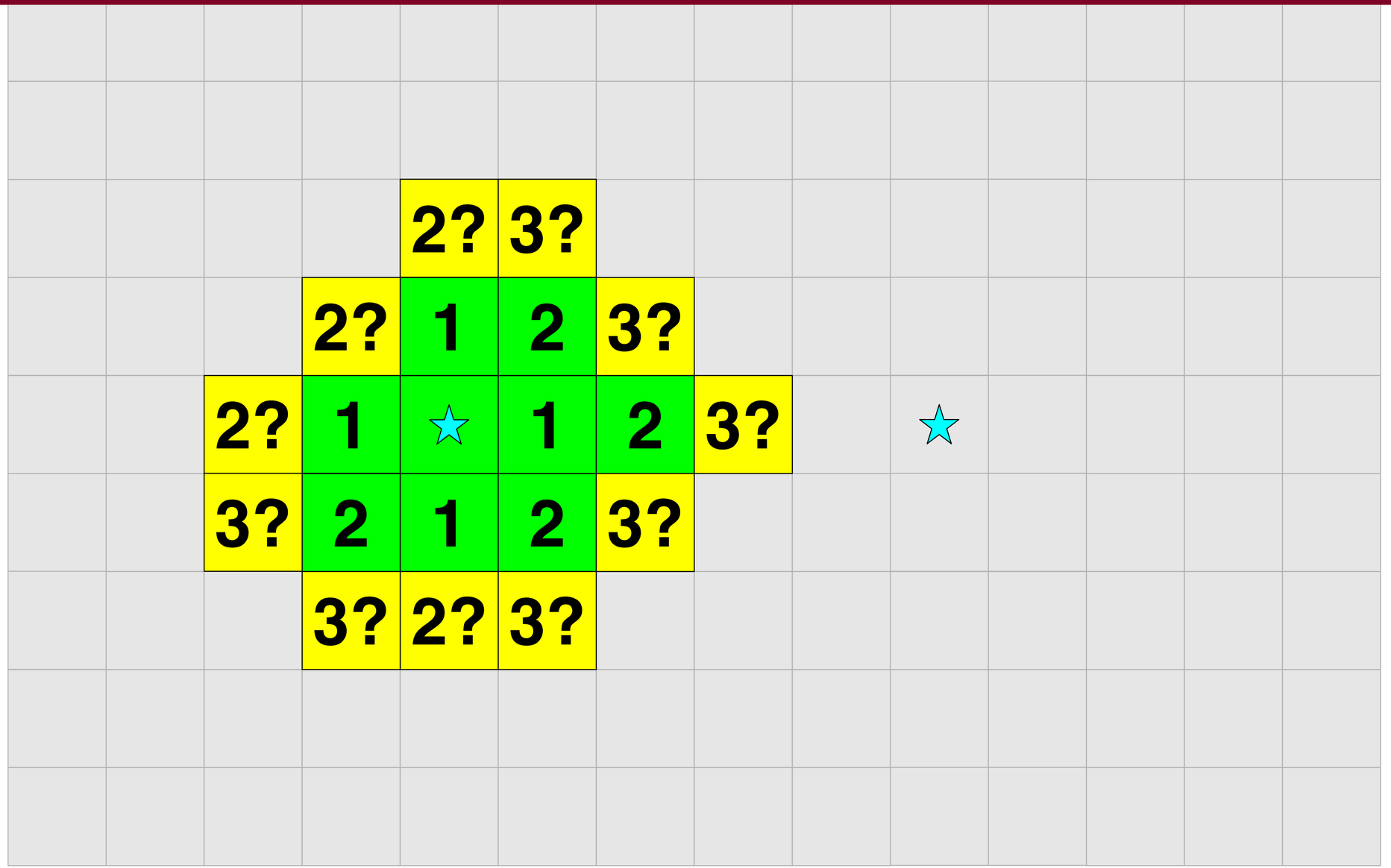
Dijkstra where each edge has cost 1



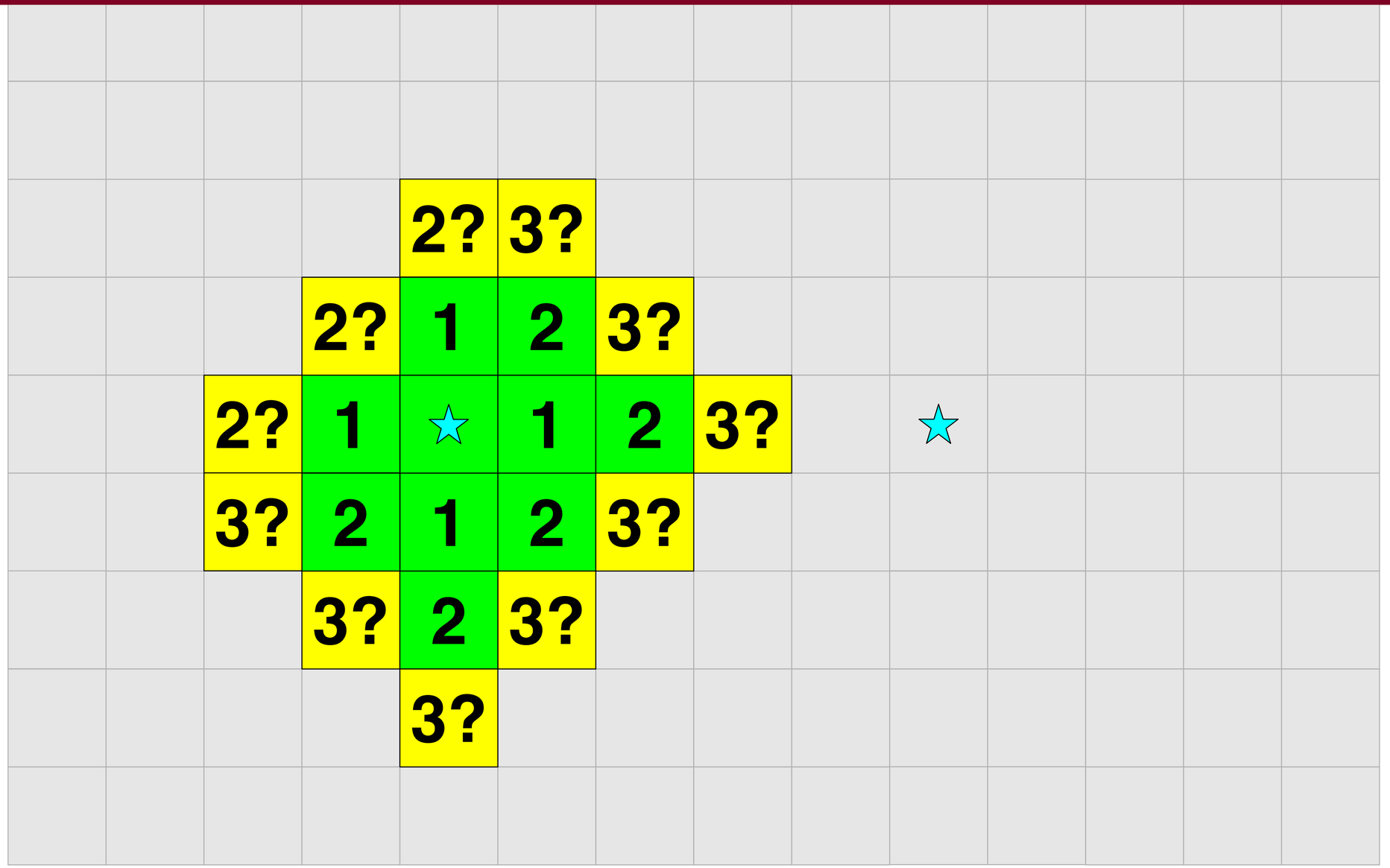
Dijkstra where each edge has cost 1



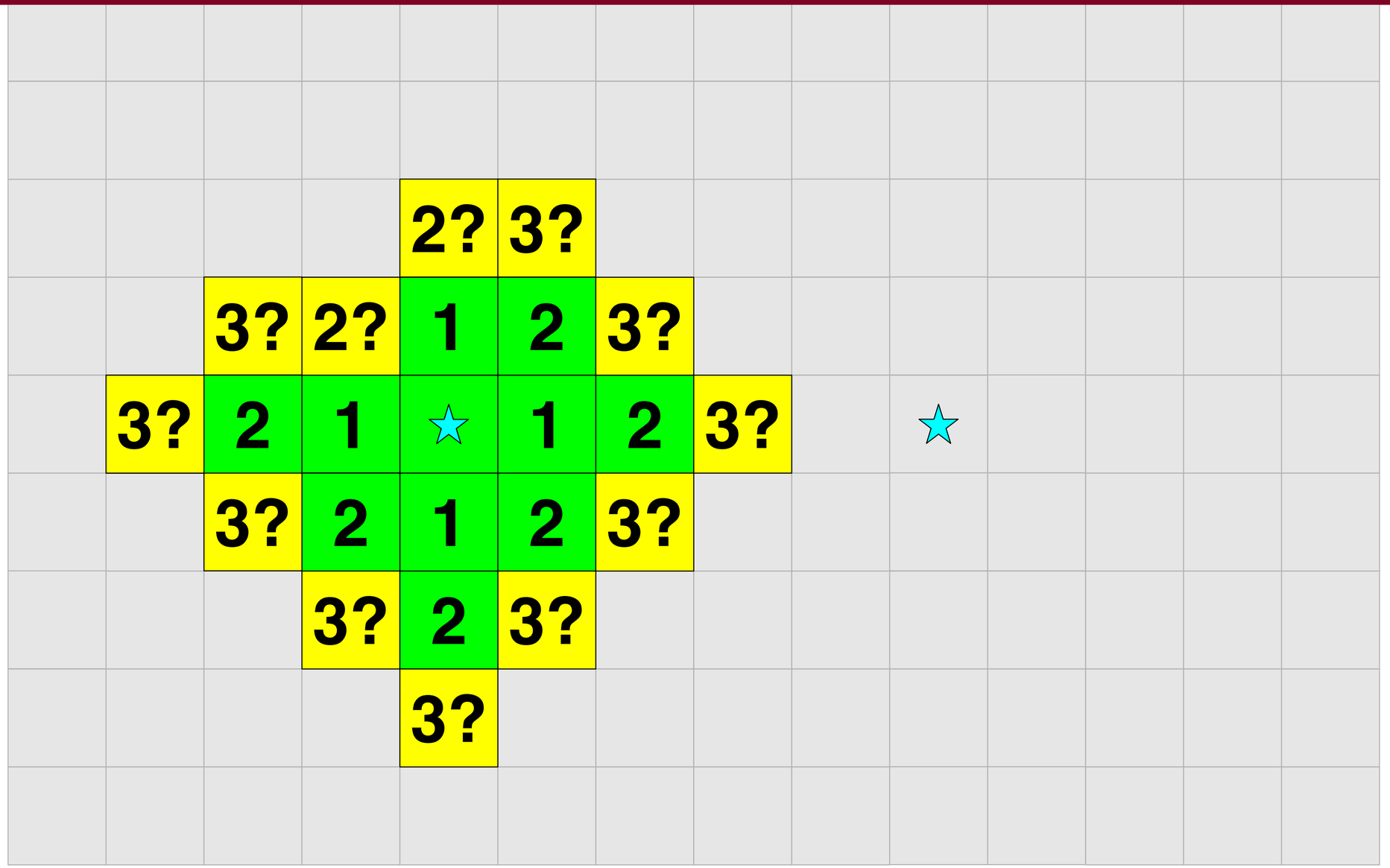
Dijkstra where each edge has cost 1



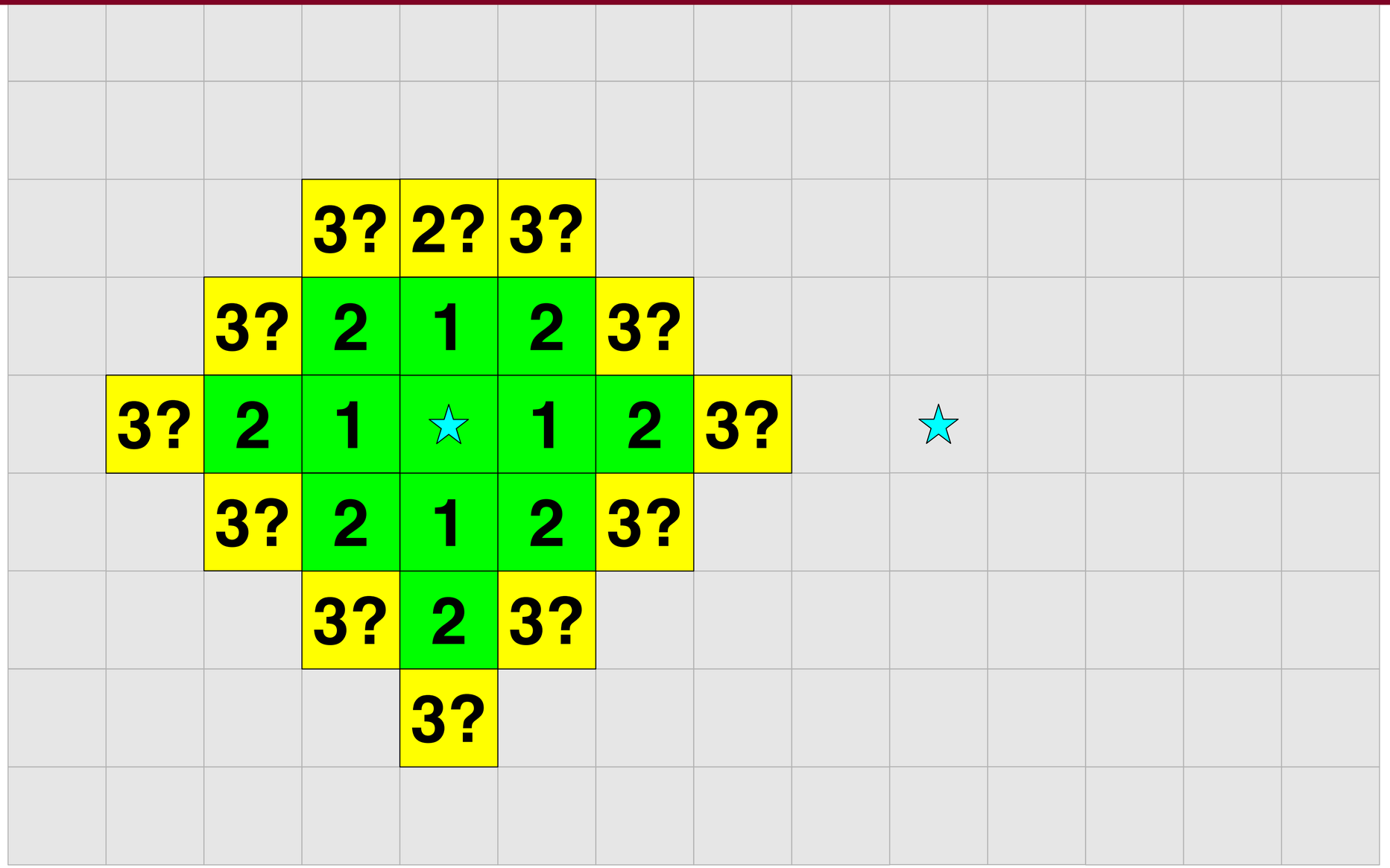
Dijkstra where each edge has cost 1



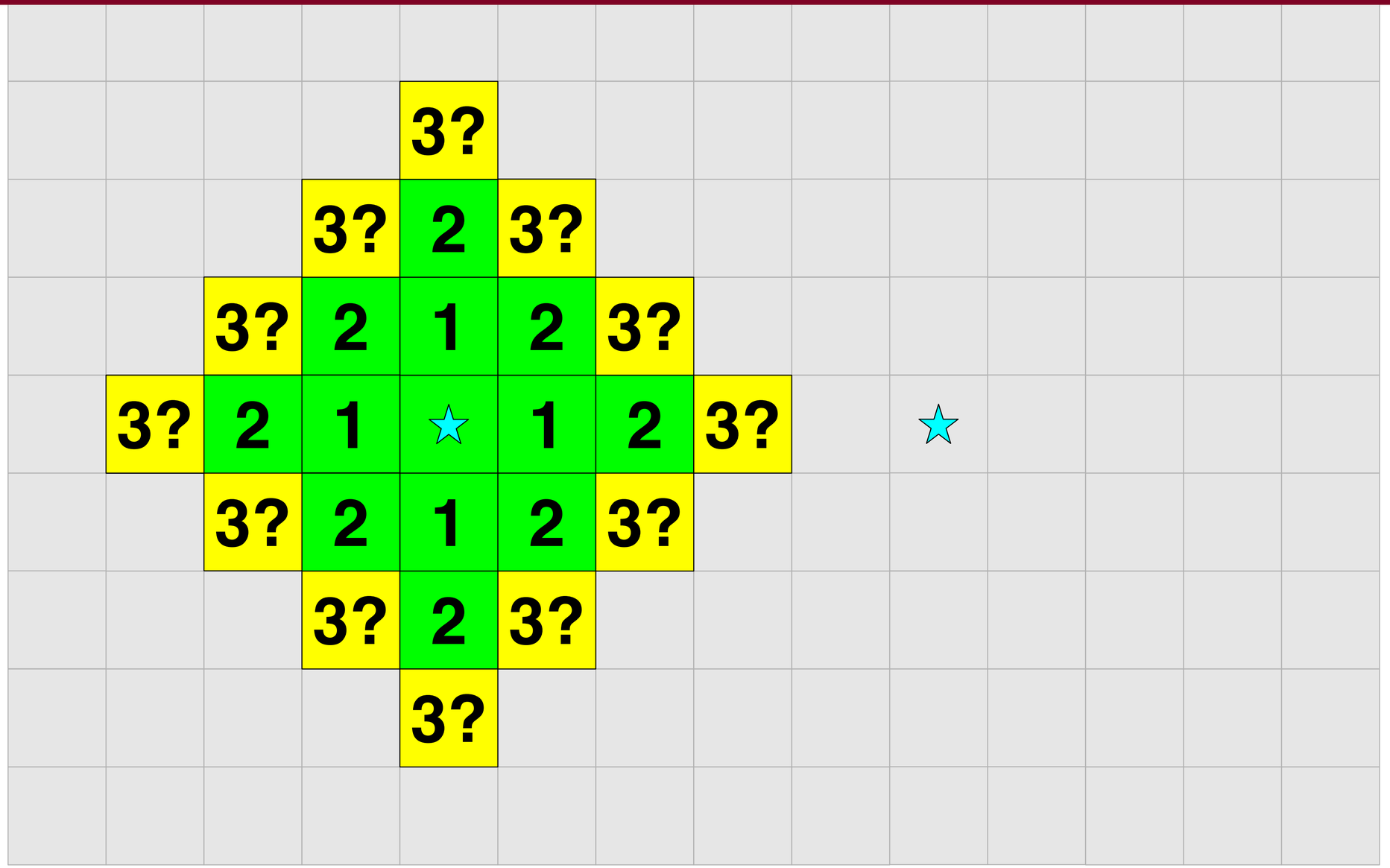
Dijkstra where each edge has cost 1



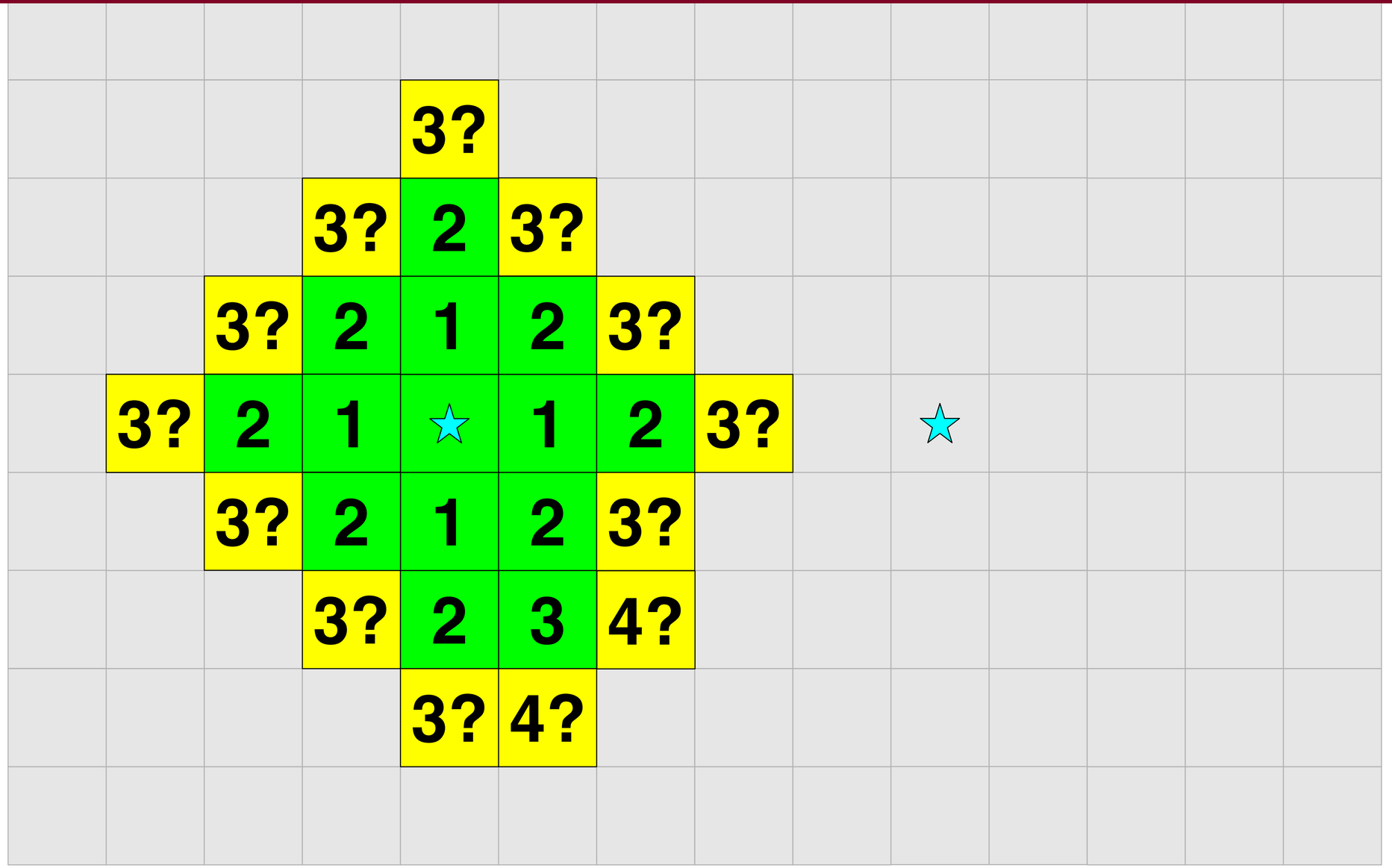
Dijkstra where each edge has cost 1



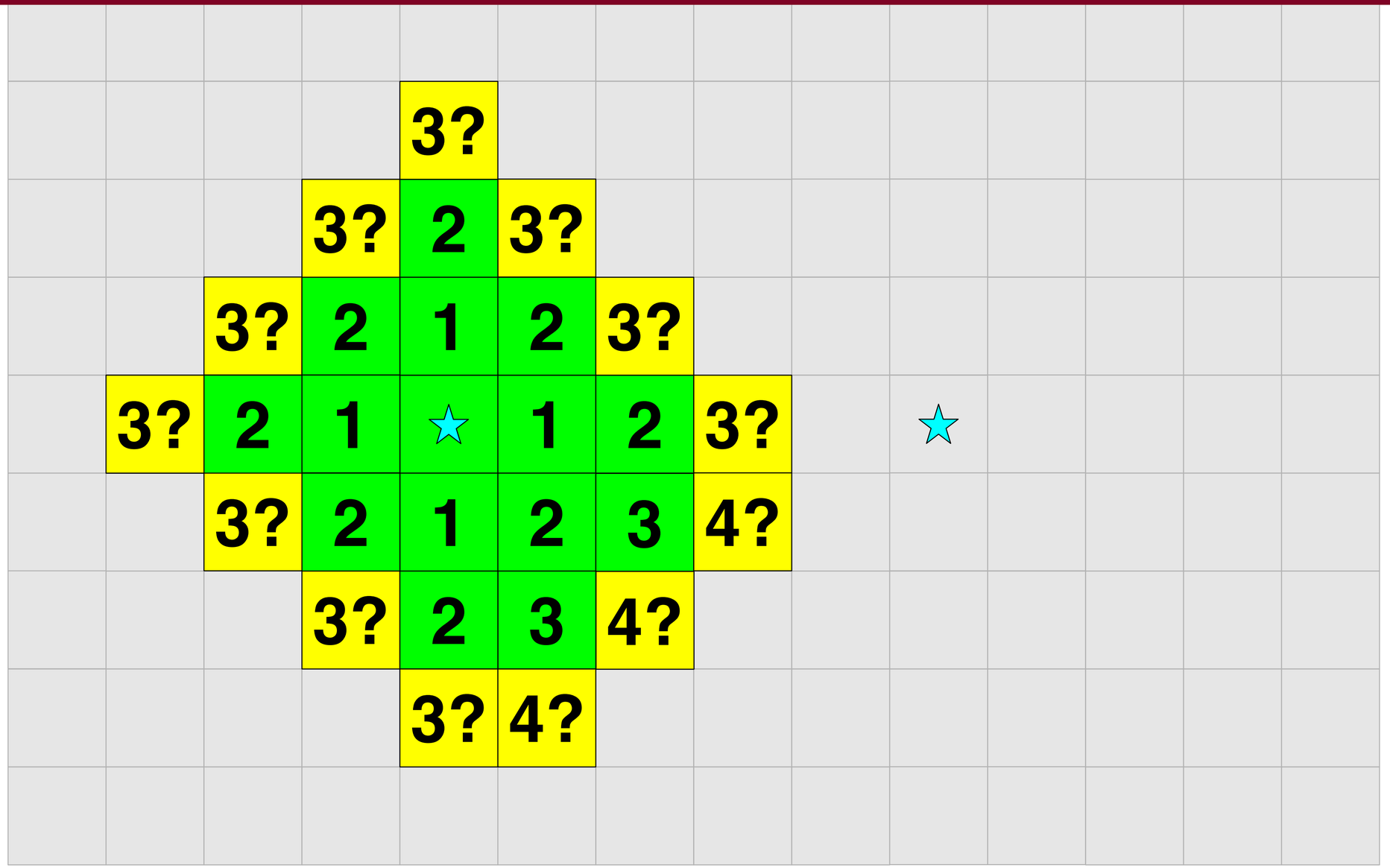
Dijkstra where each edge has cost 1



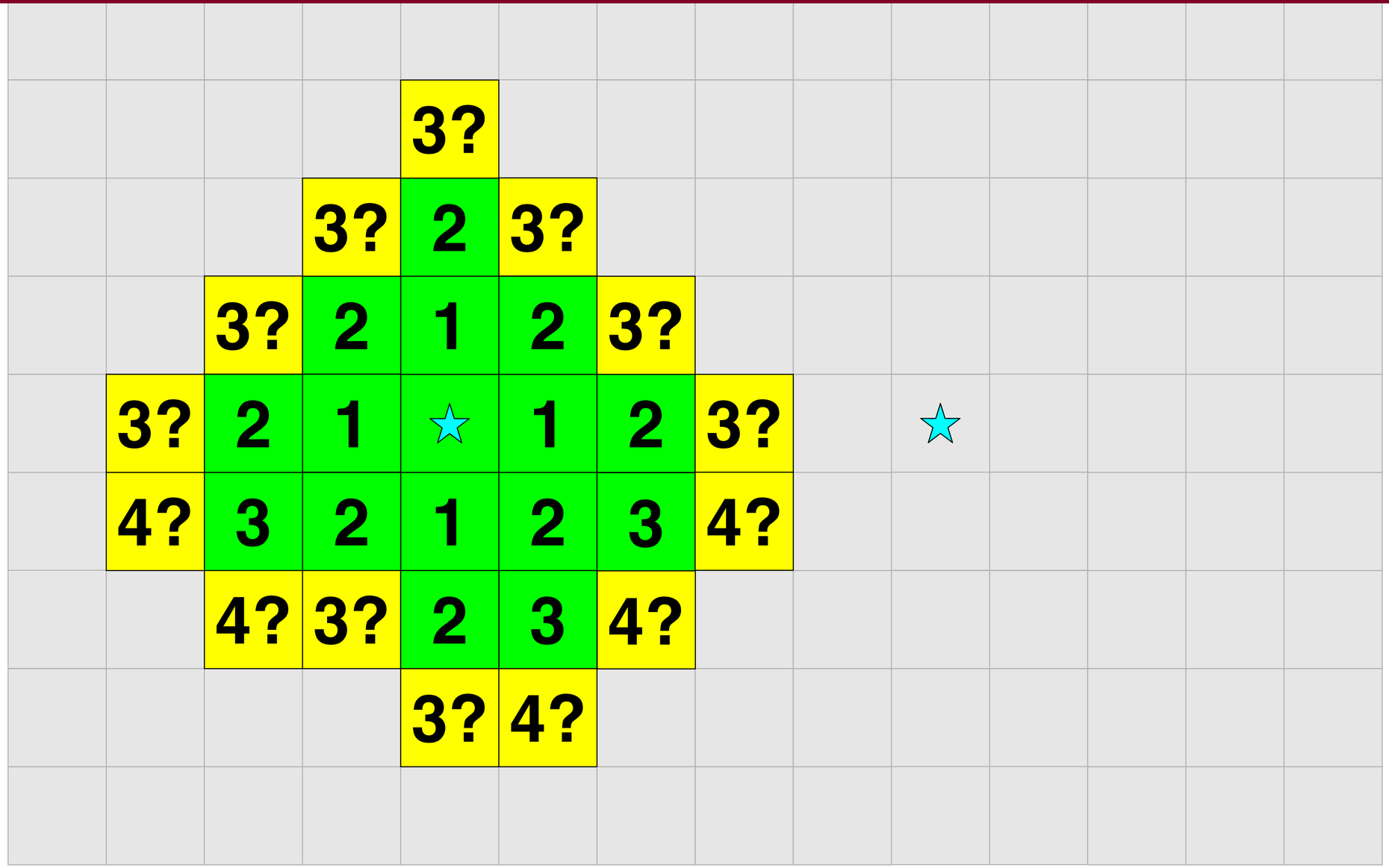
Dijkstra where each edge has cost 1



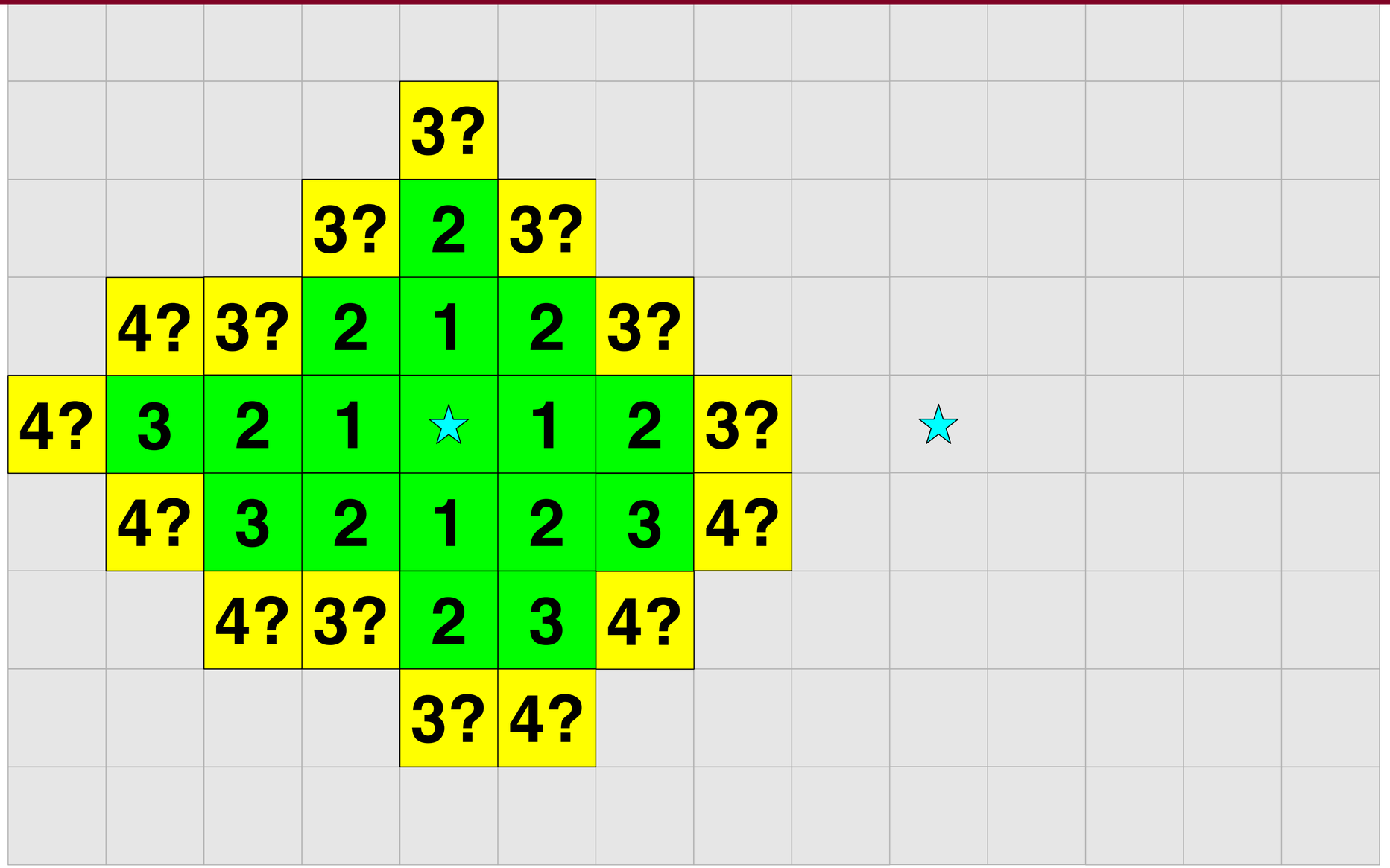
Dijkstra where each edge has cost 1



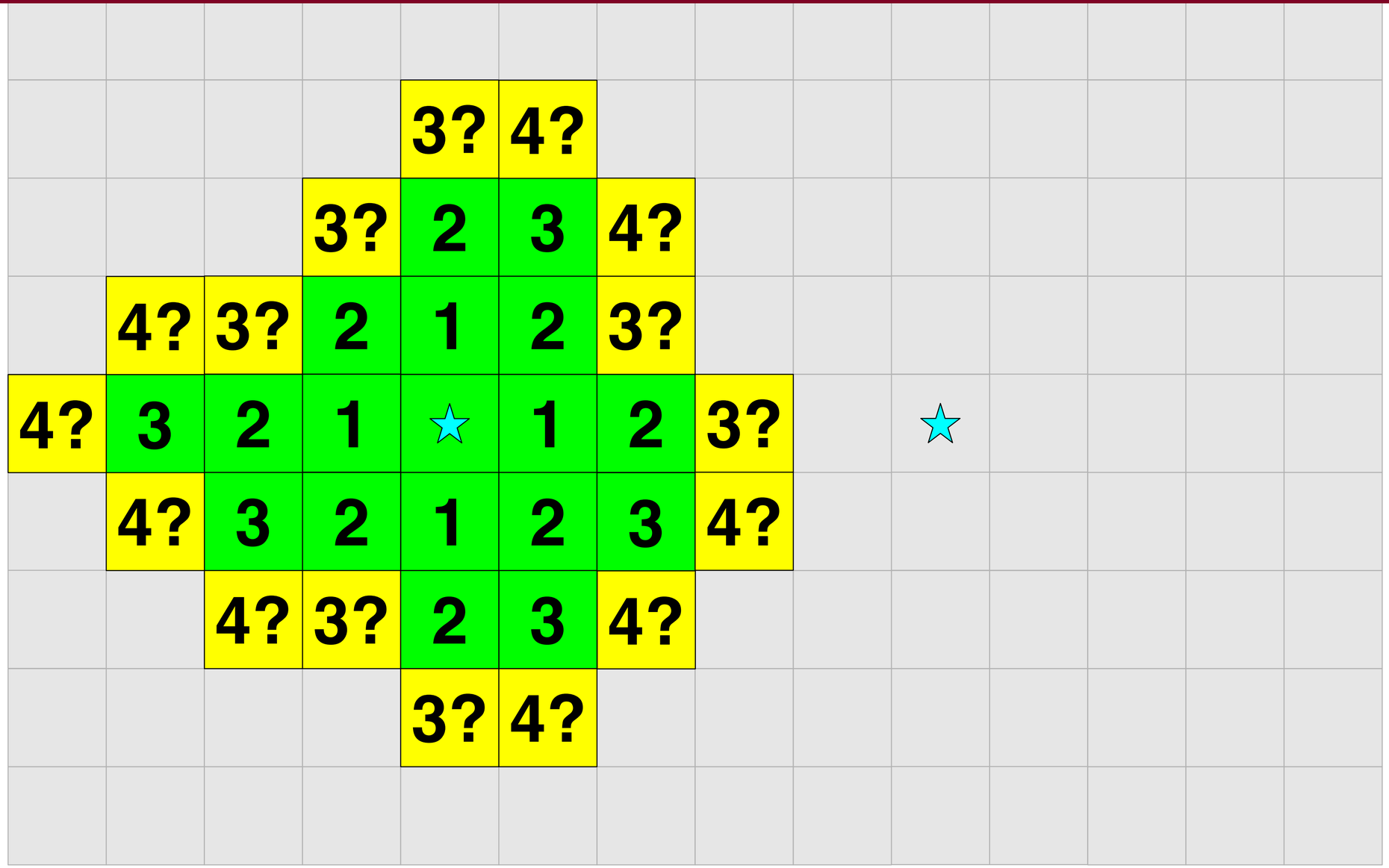
Dijkstra where each edge has cost 1



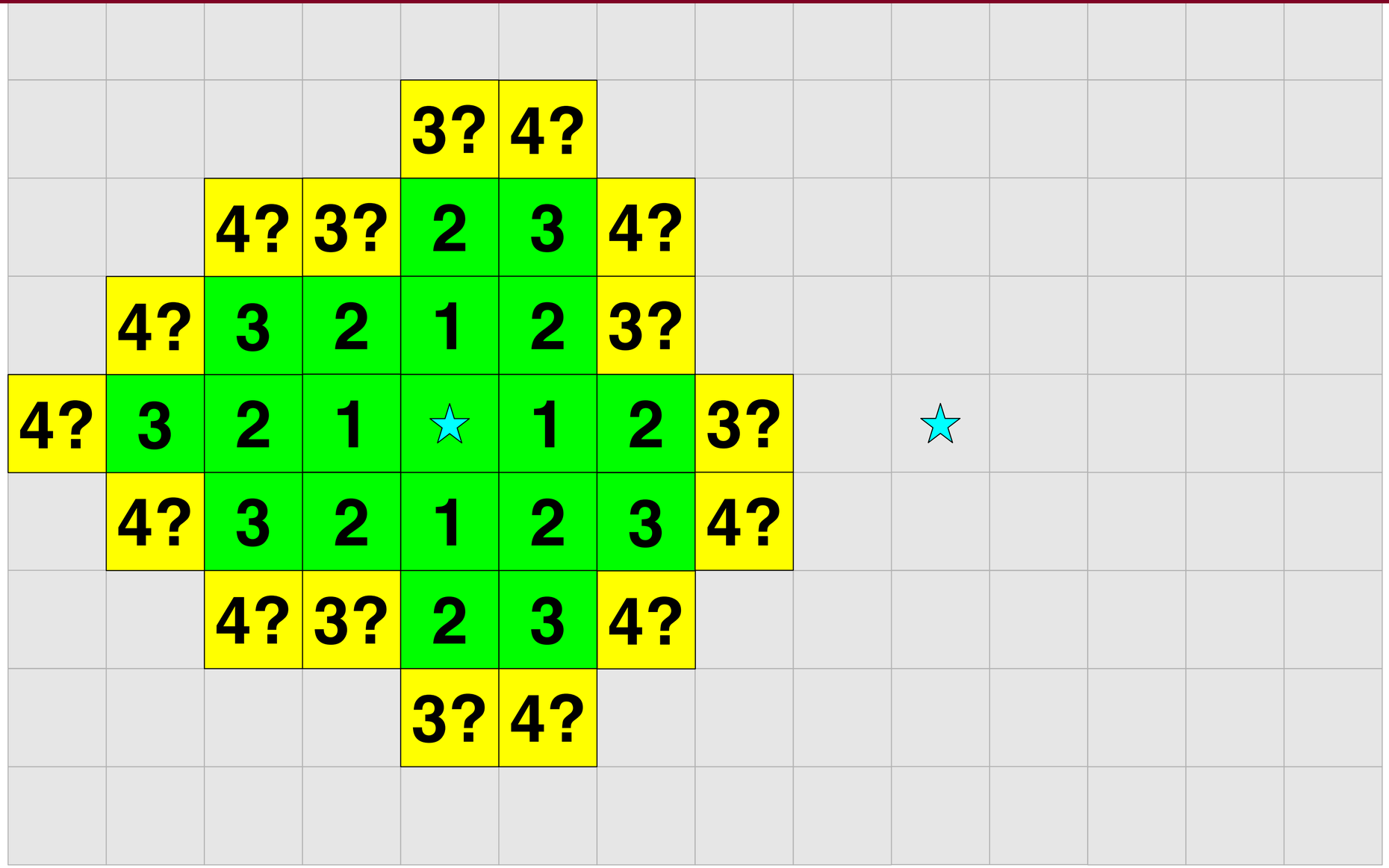
Dijkstra where each edge has cost 1



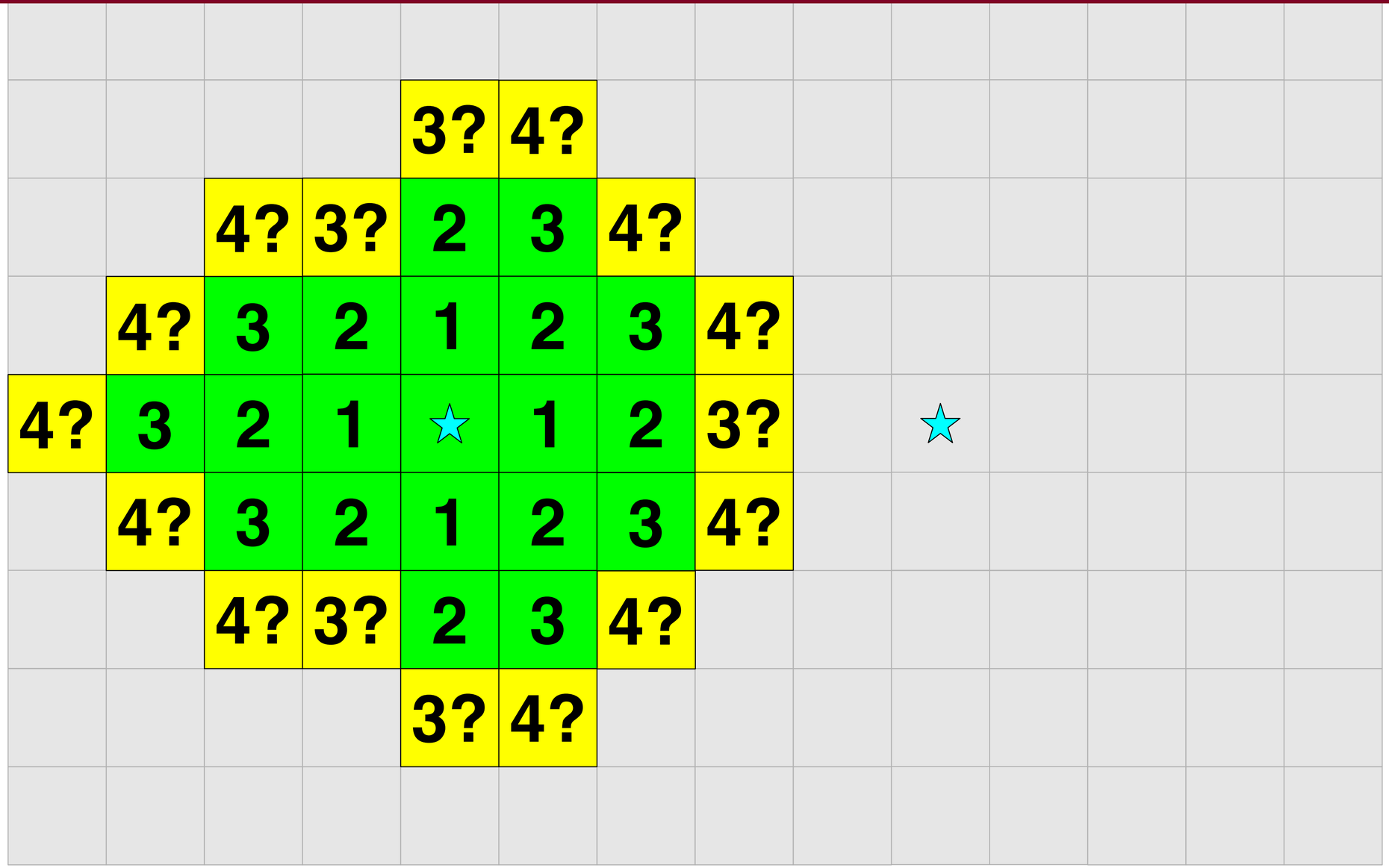
Dijkstra where each edge has cost 1



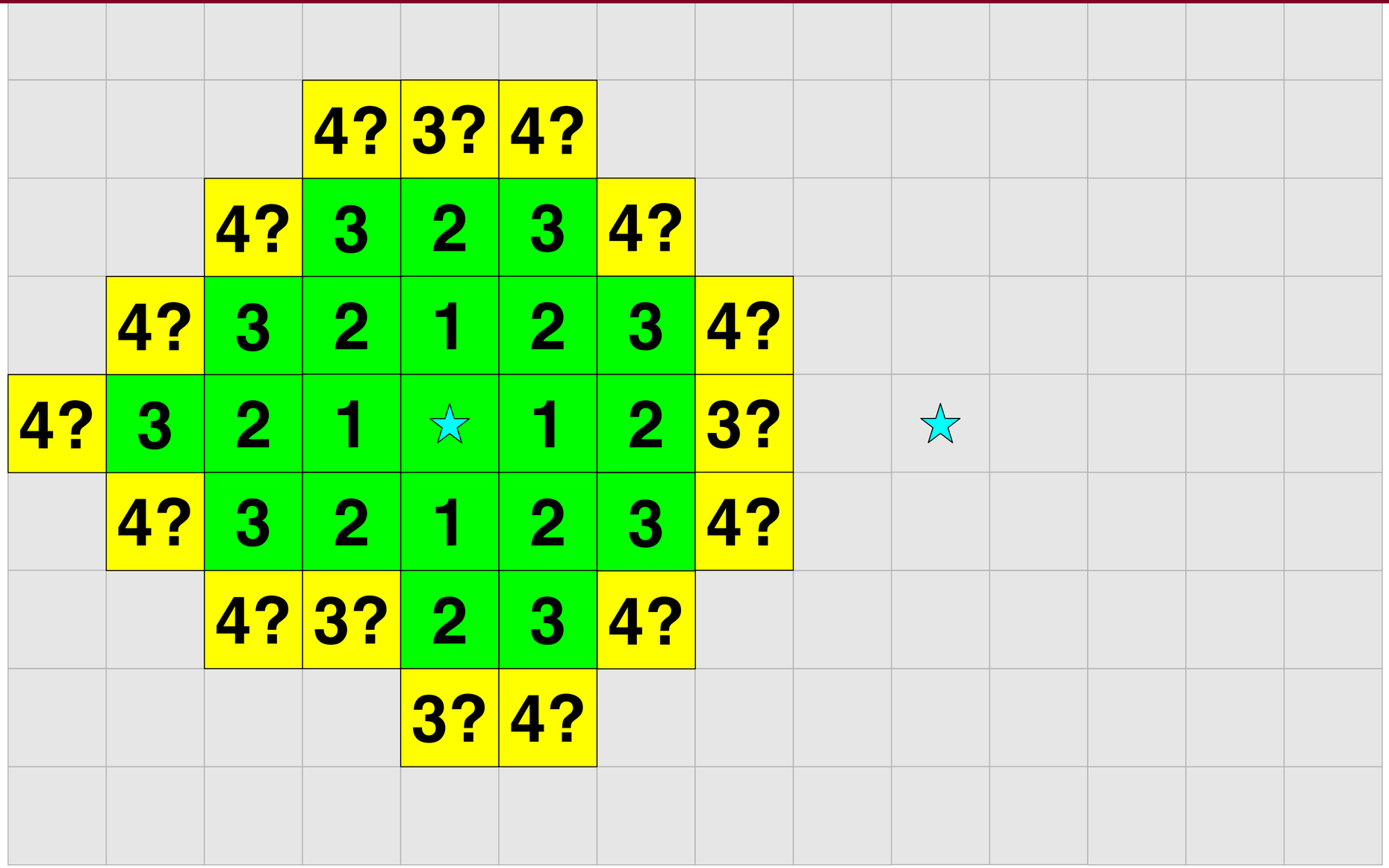
Dijkstra where each edge has cost 1



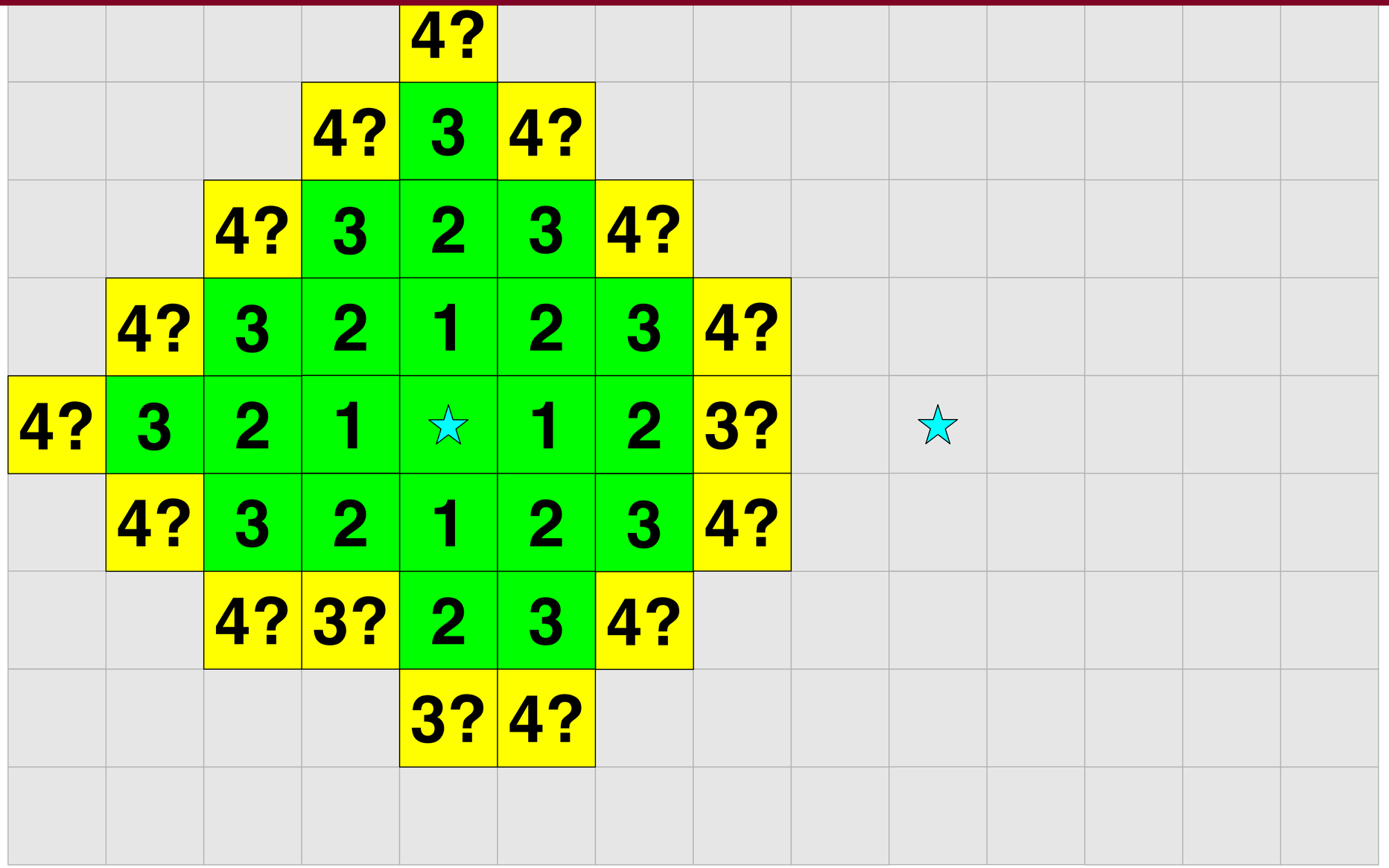
Dijkstra where each edge has cost 1



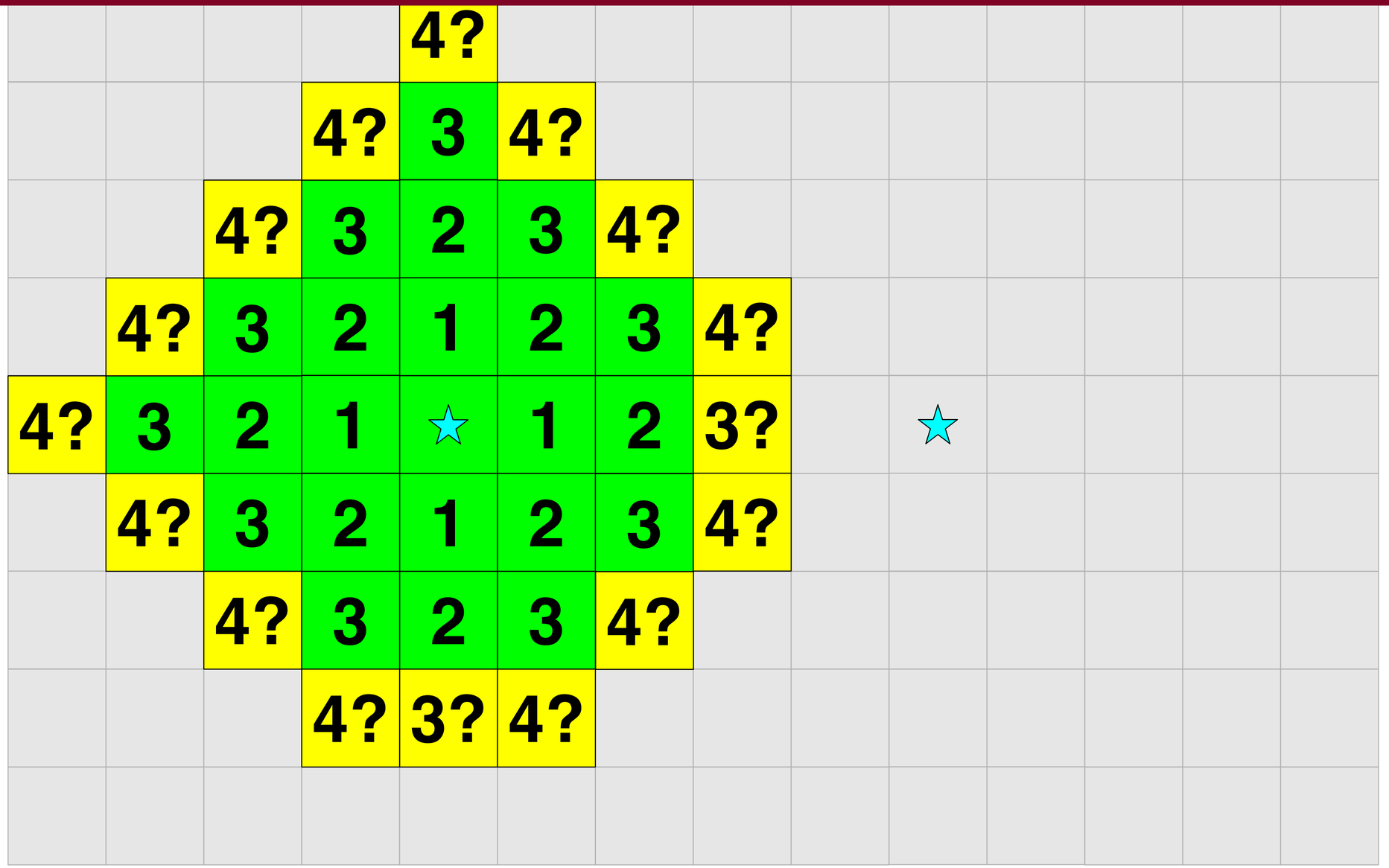
Dijkstra where each edge has cost 1



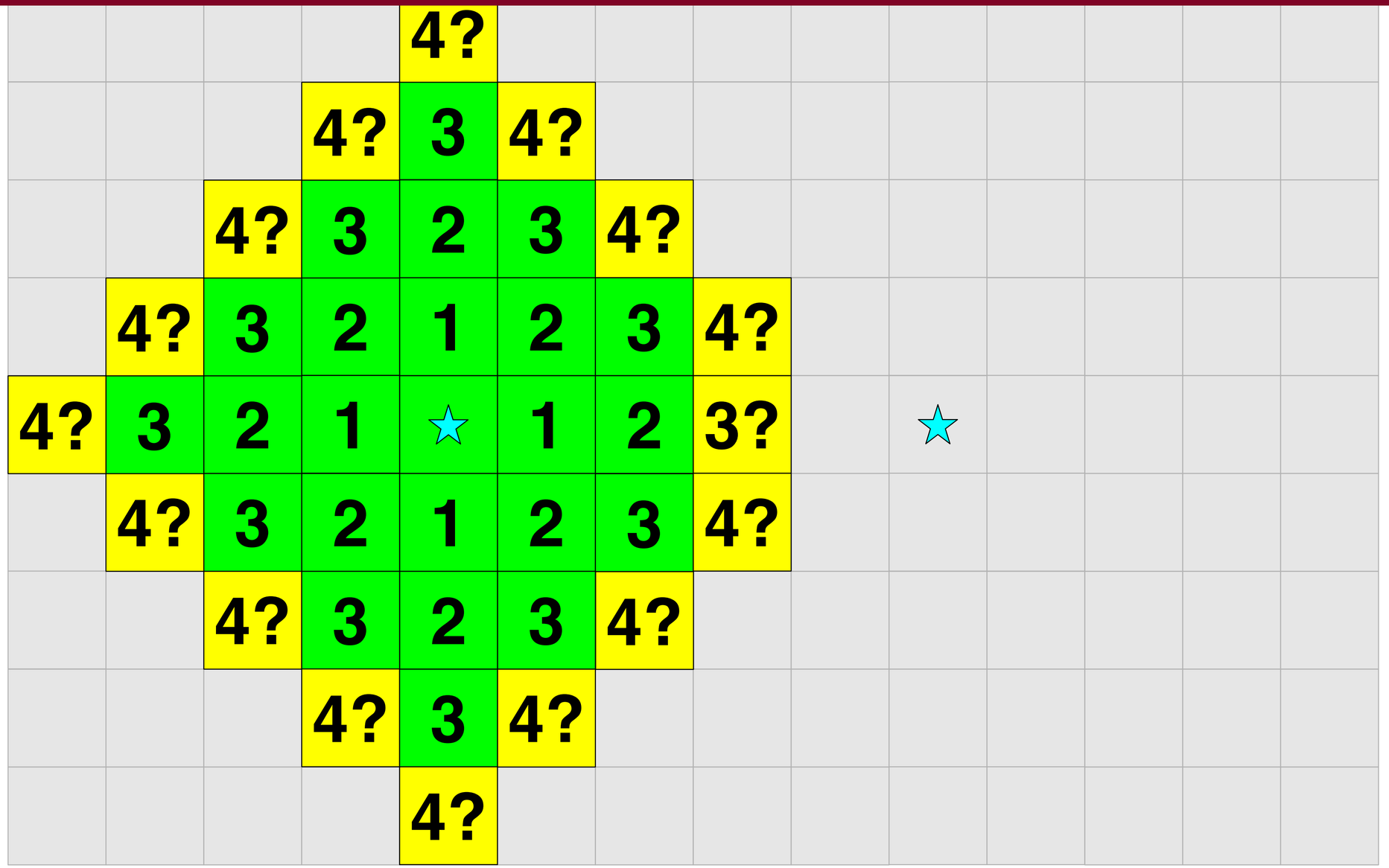
Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

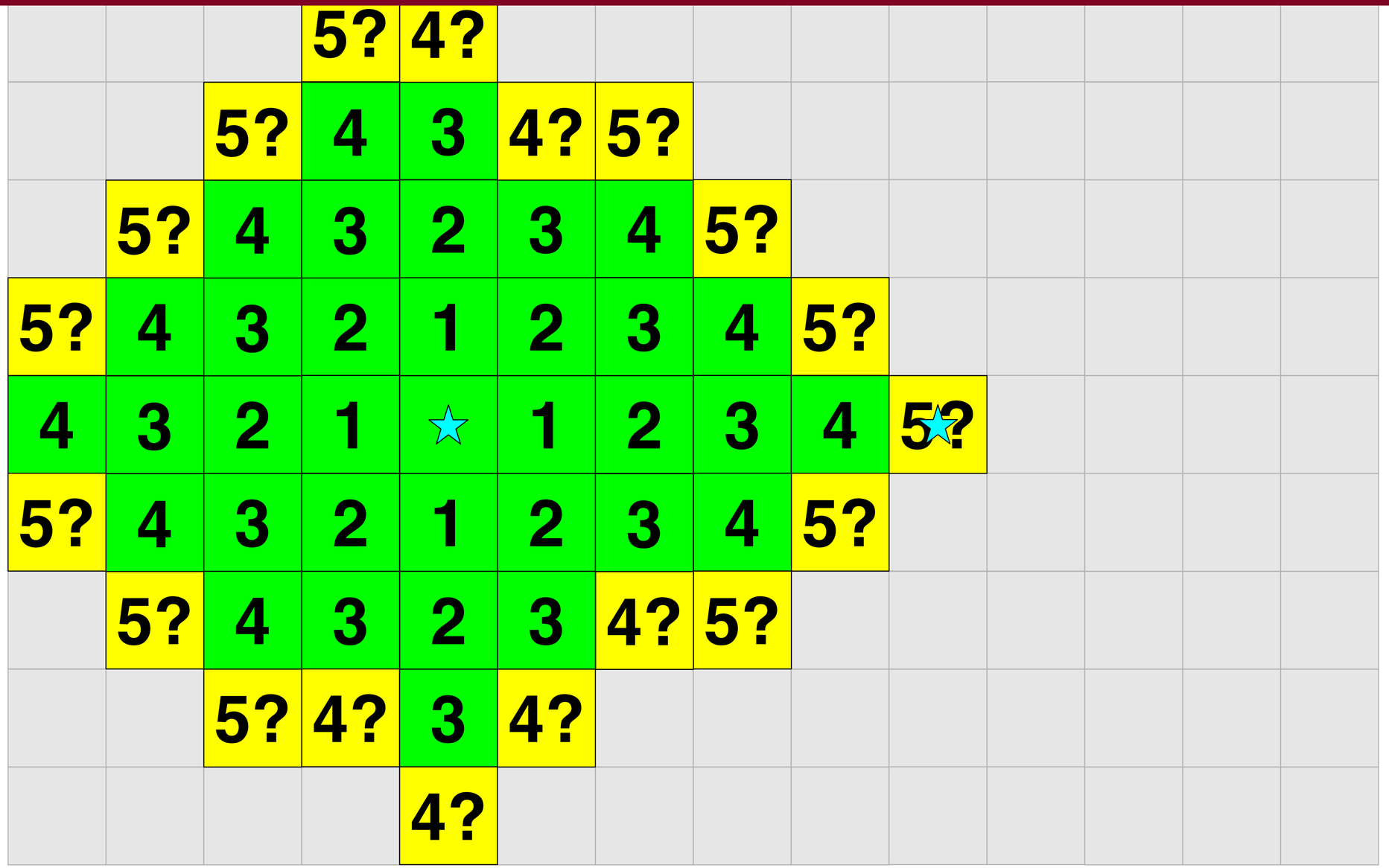
Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

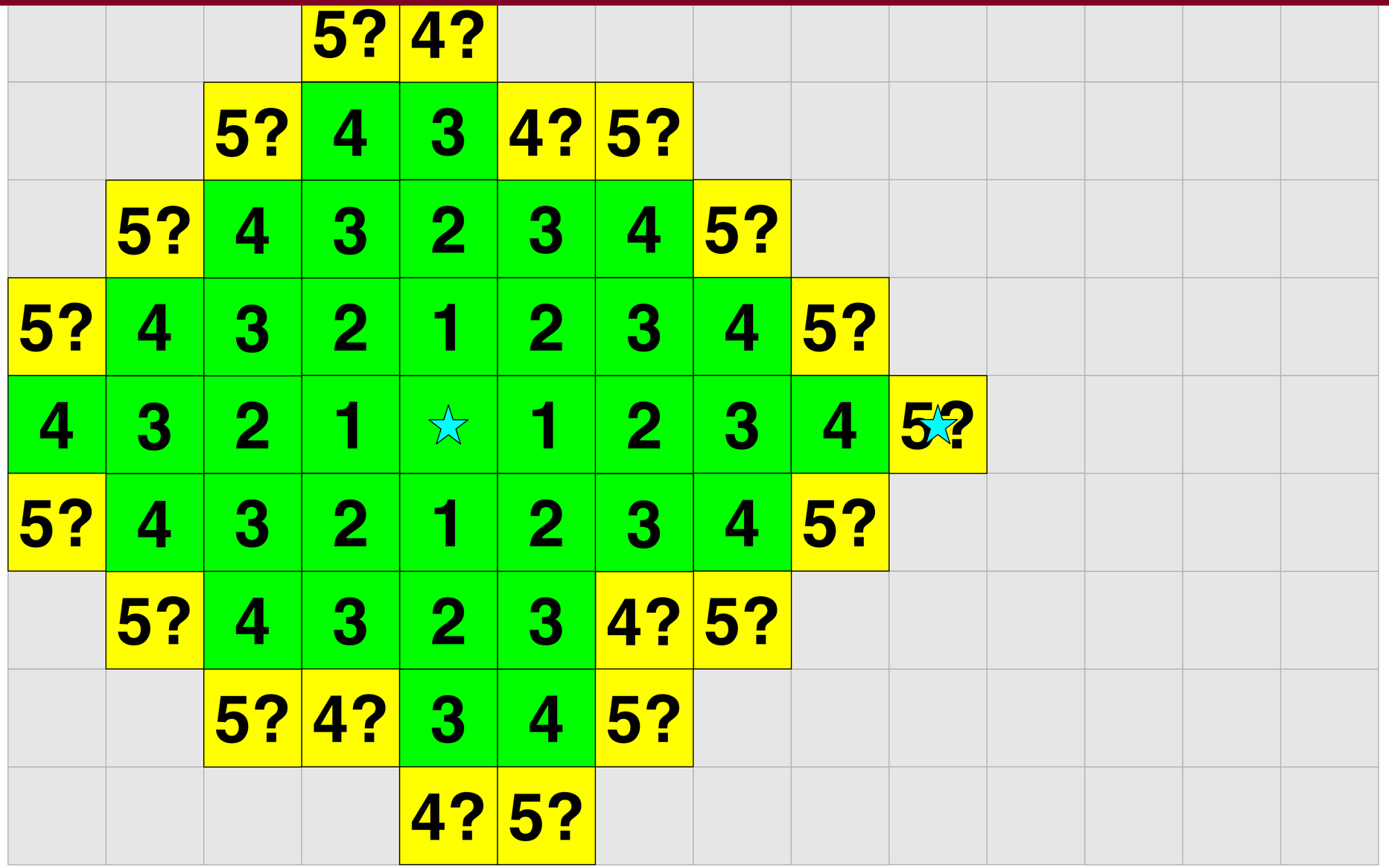
Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

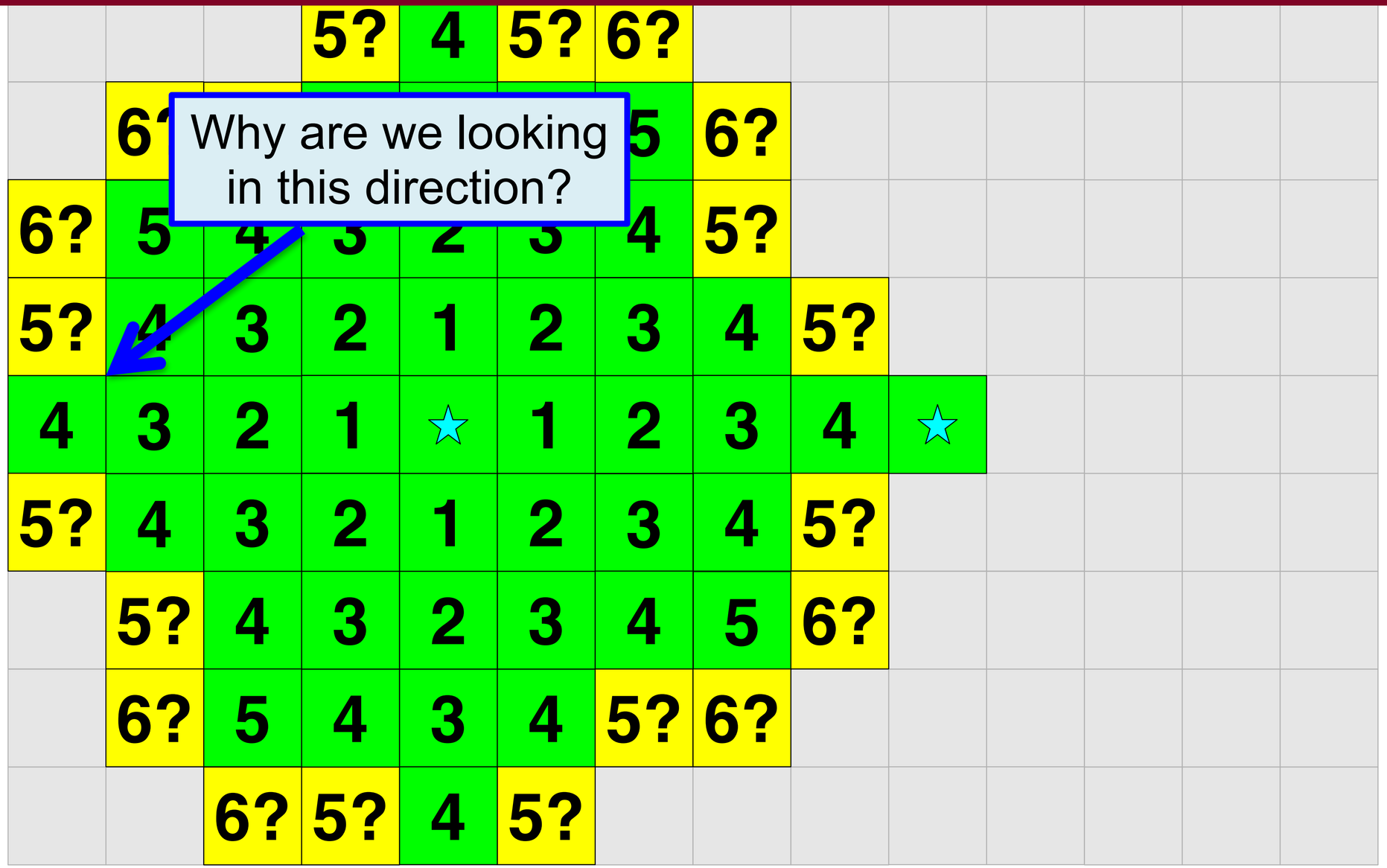
Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

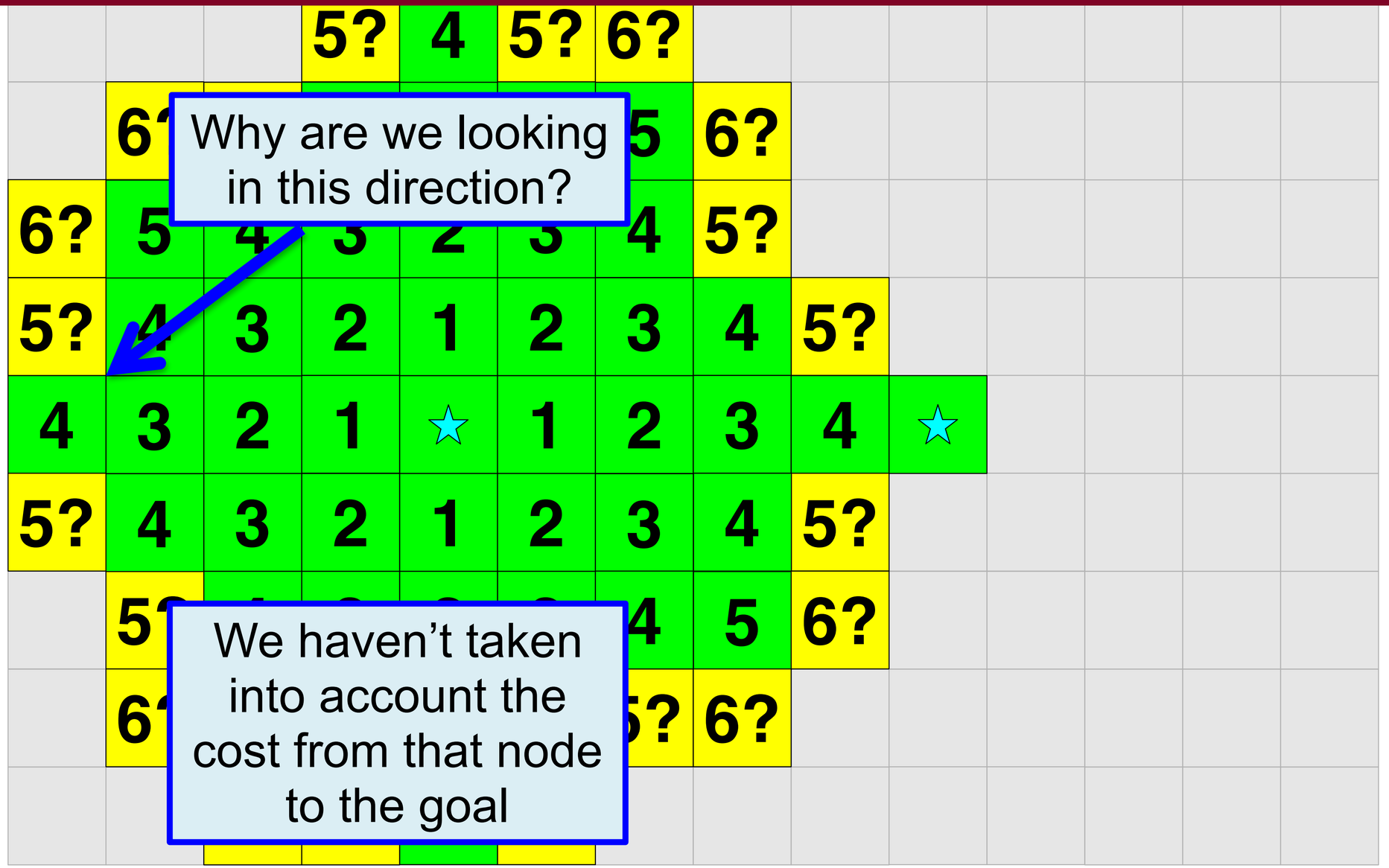
Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

Dijkstra where each edge has cost 1

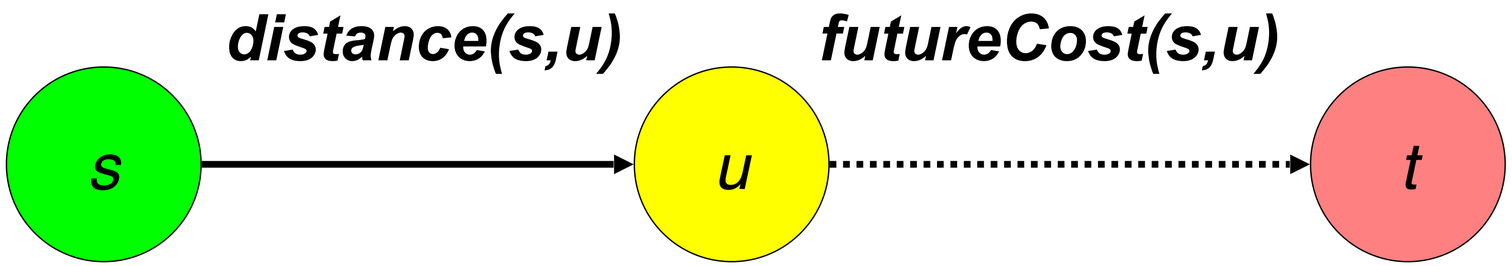


Dijkstra where each edge has cost 1



Dijkstra Priority

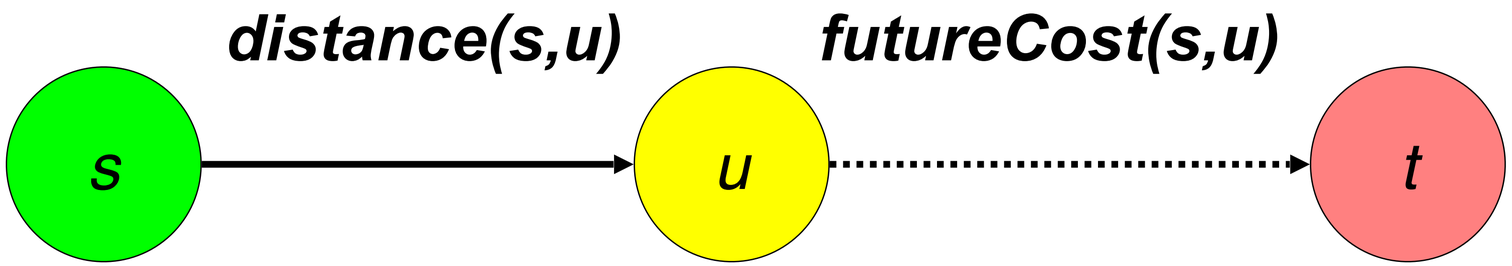
priority(u) = distance(s, u)



Priority of the path that ends in u

Ideal Priority

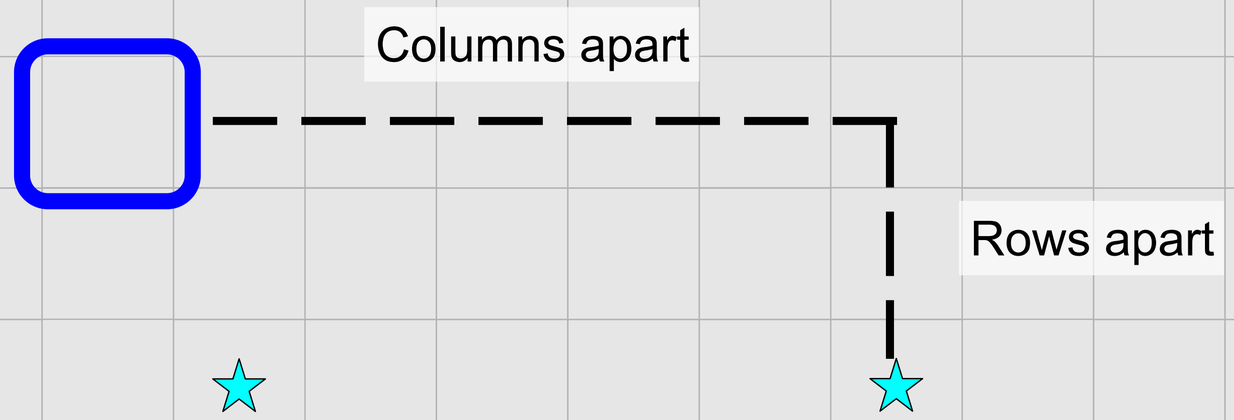
priority(u) = distance(s, u) + futureCost(u, t)



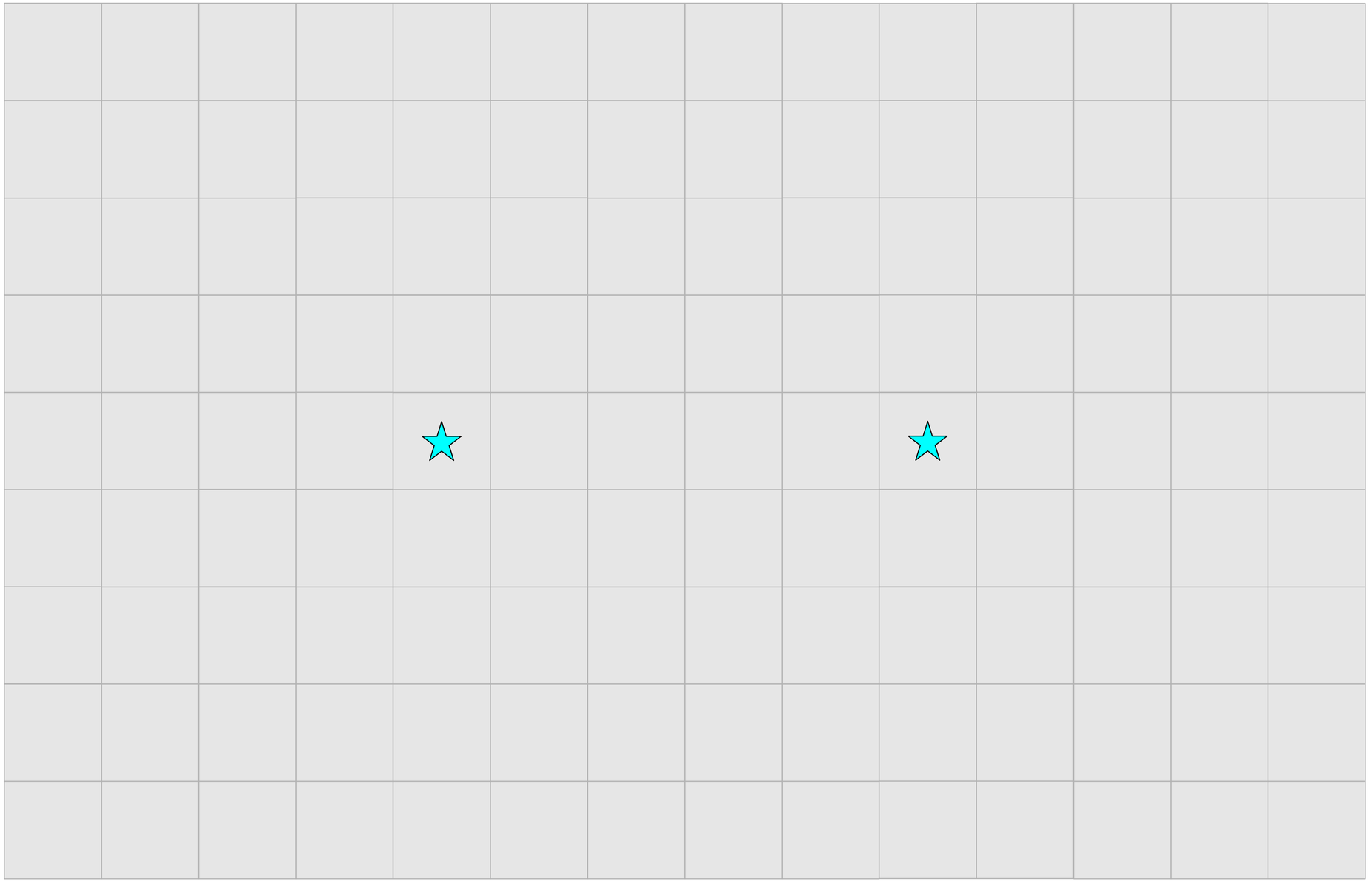
Priority of the path that ends in u

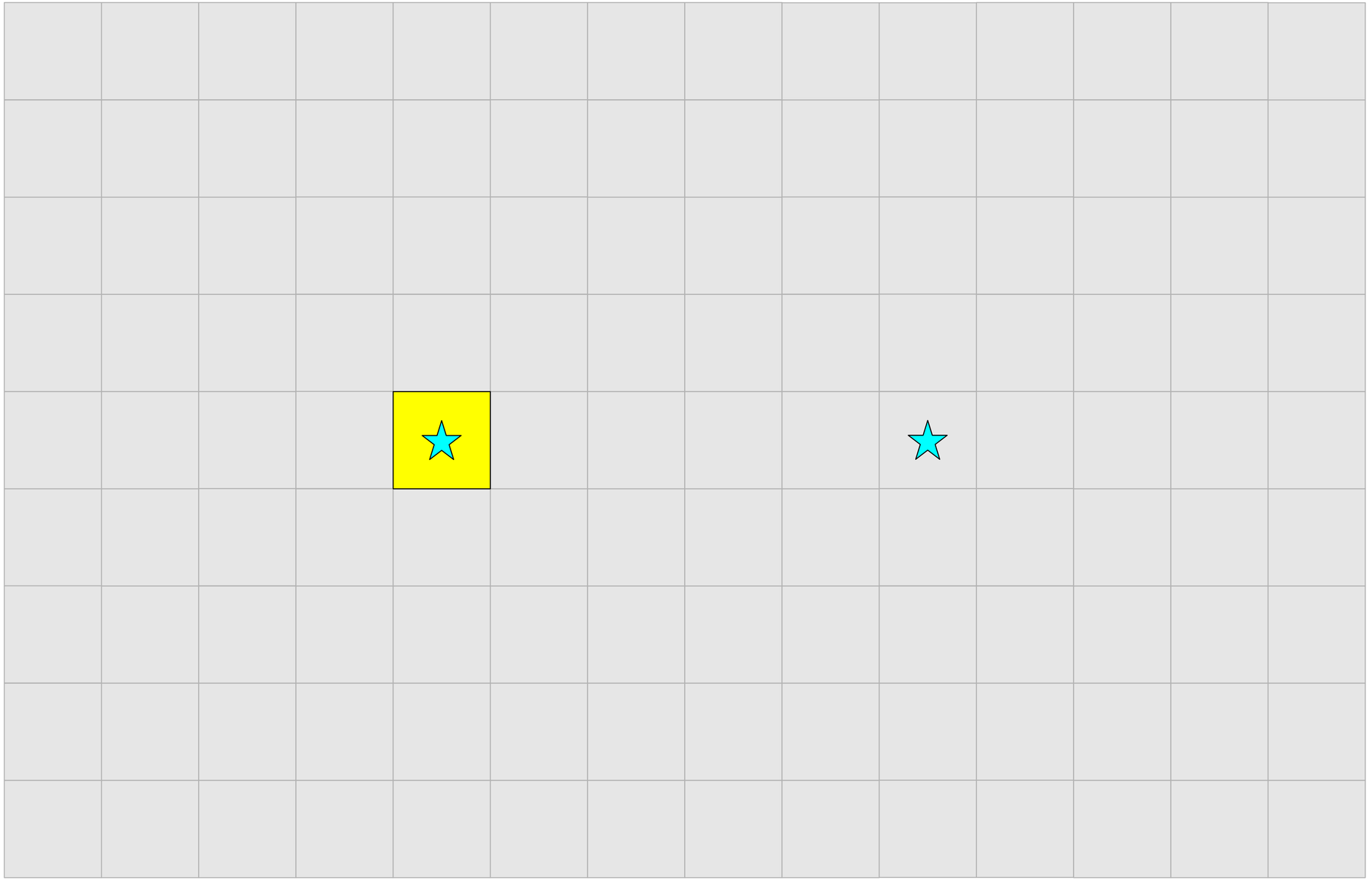
*note we will revise this slightly

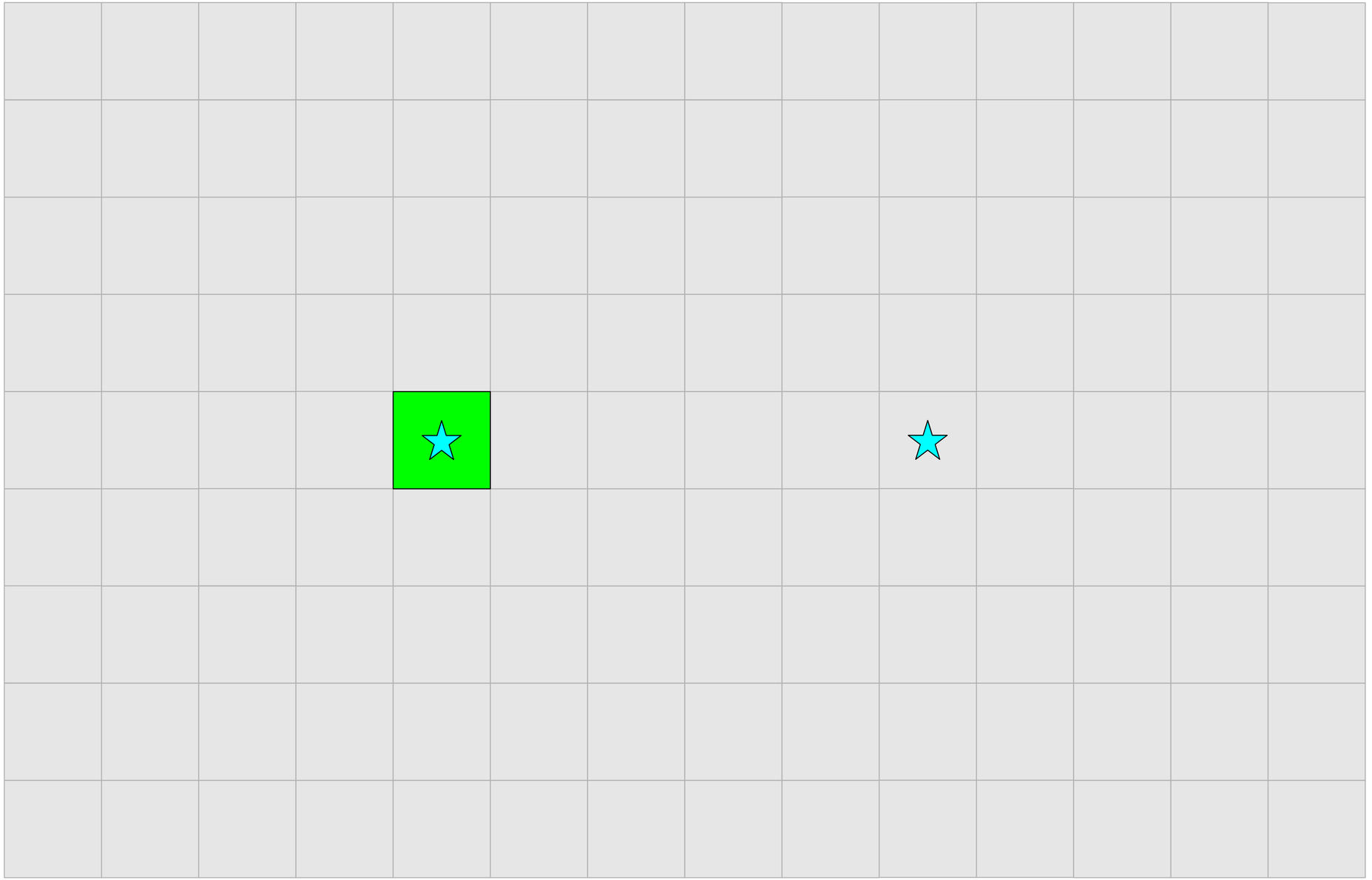
Future Cost?

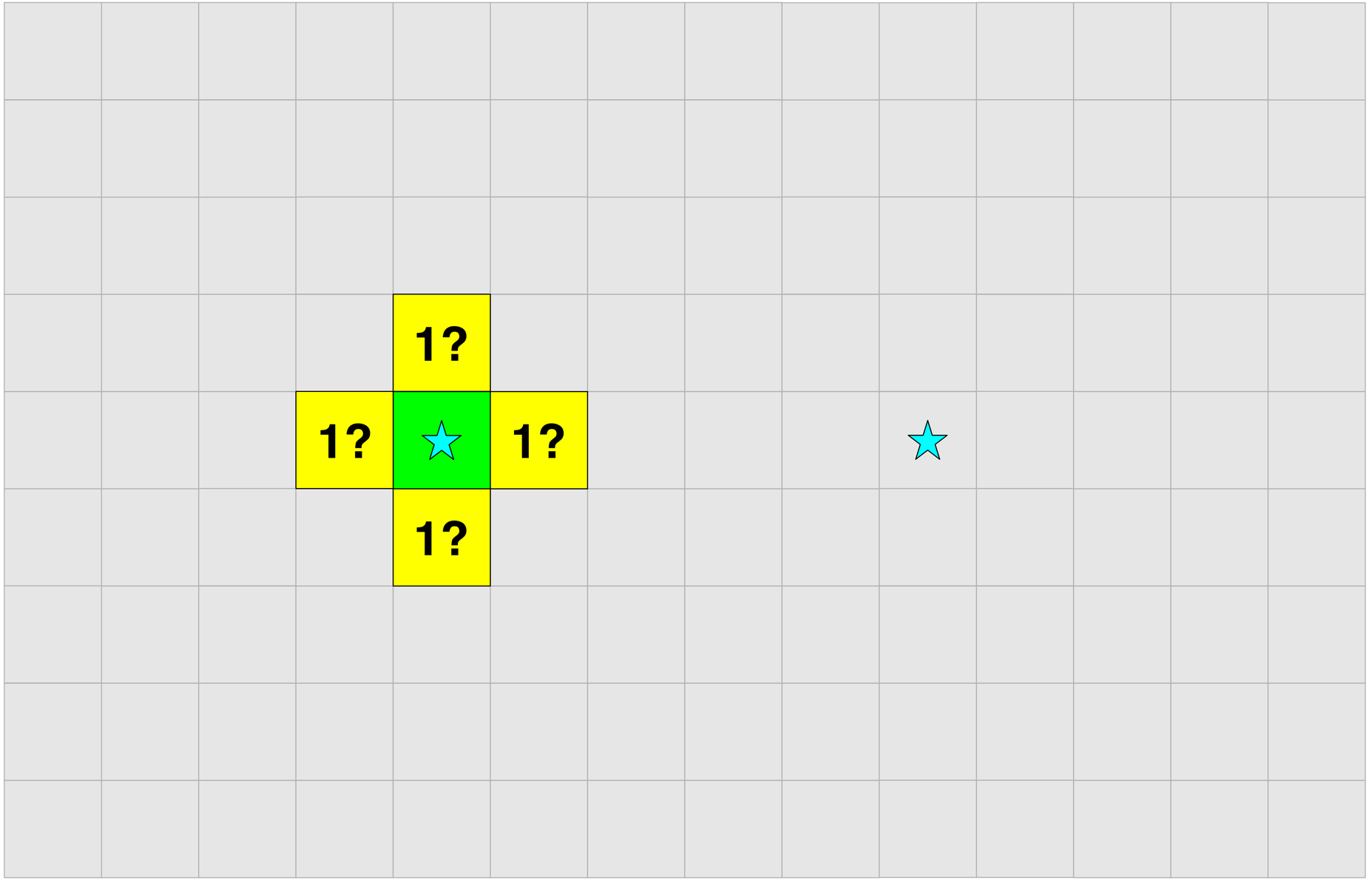


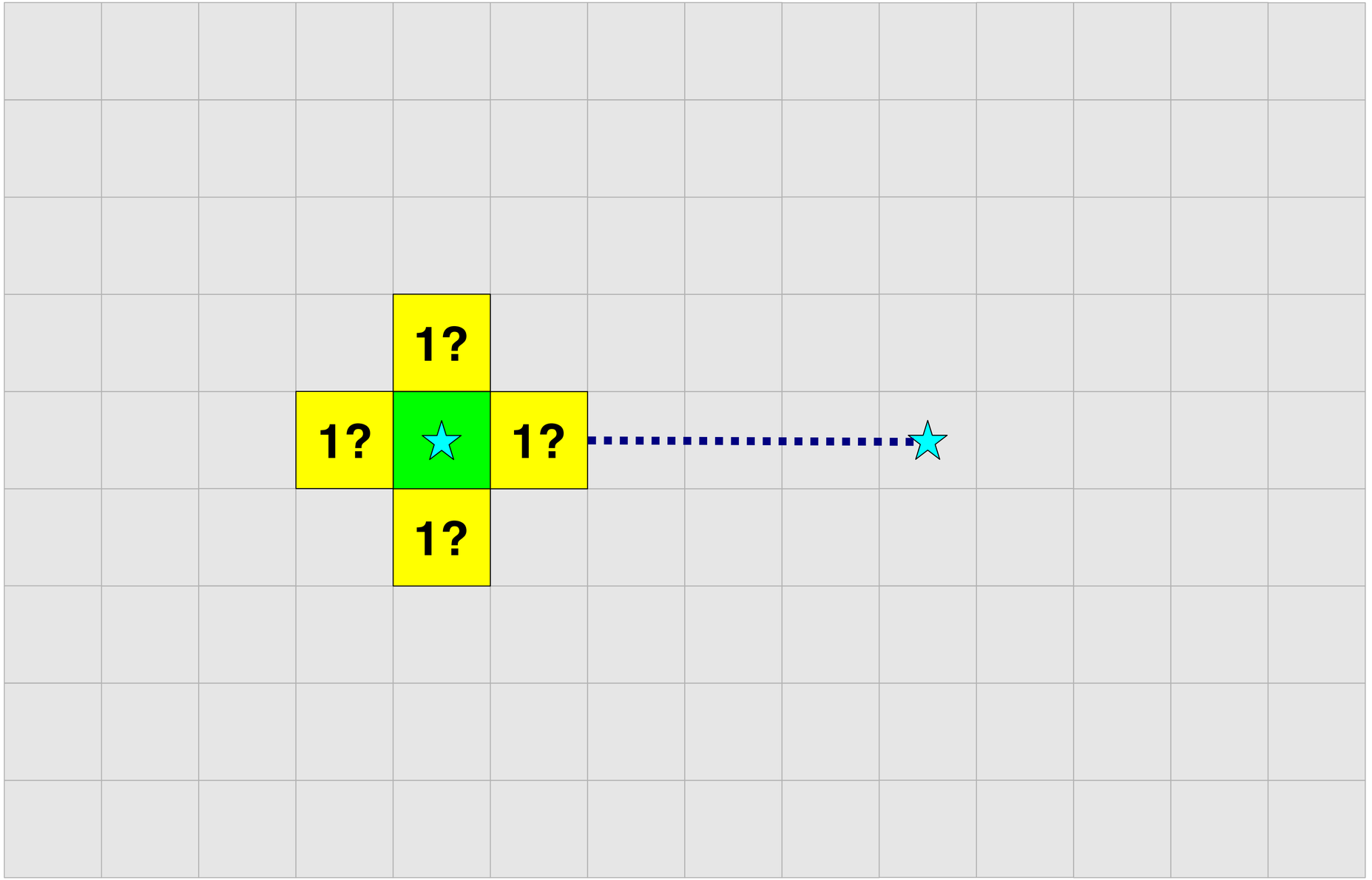

```
function h(node,goal) {
  dRows = abs(node.row - goal.row);
  dCols = abs(node.col - goal.col);
  return dRows + dCols
}
```

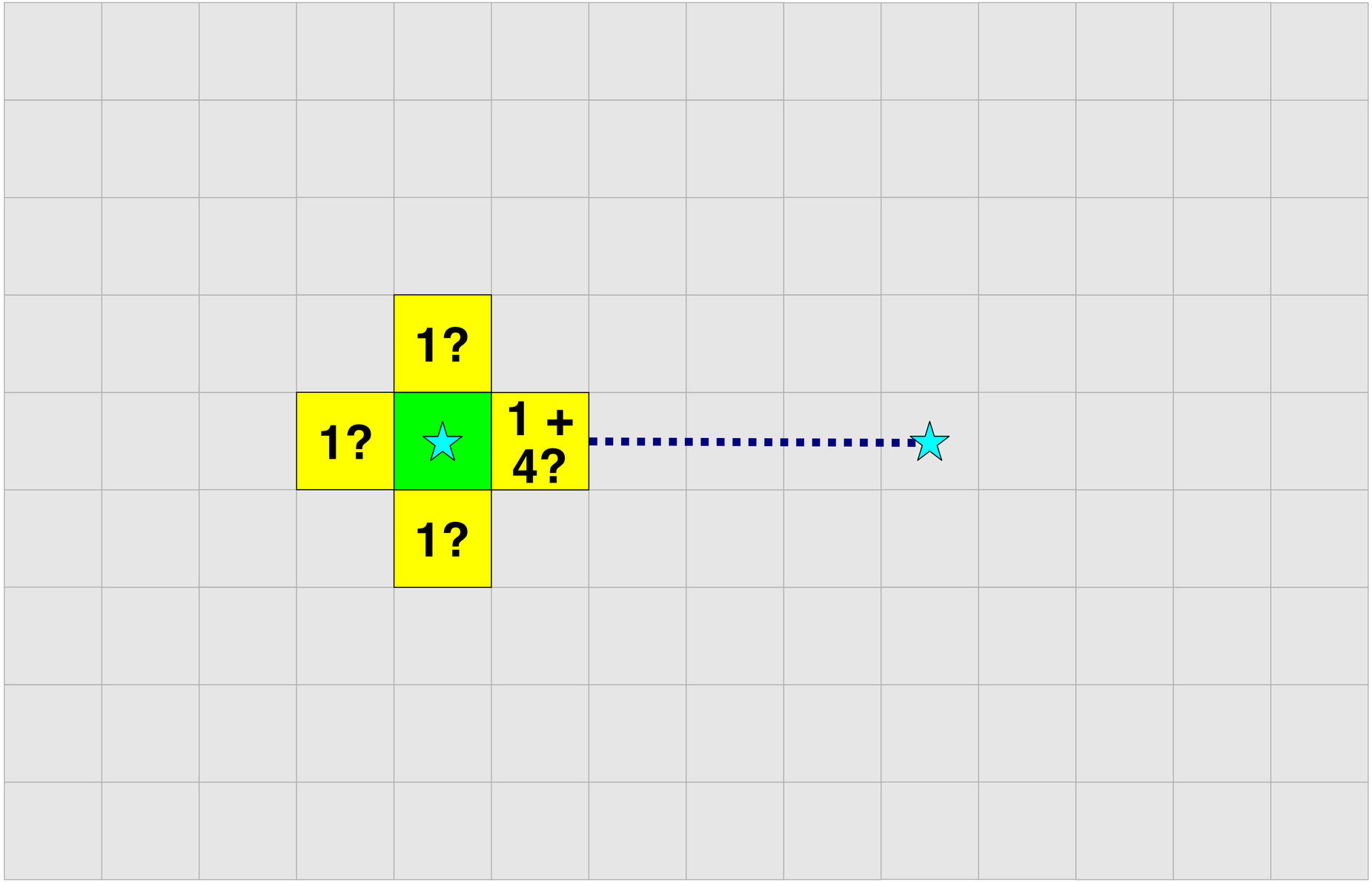


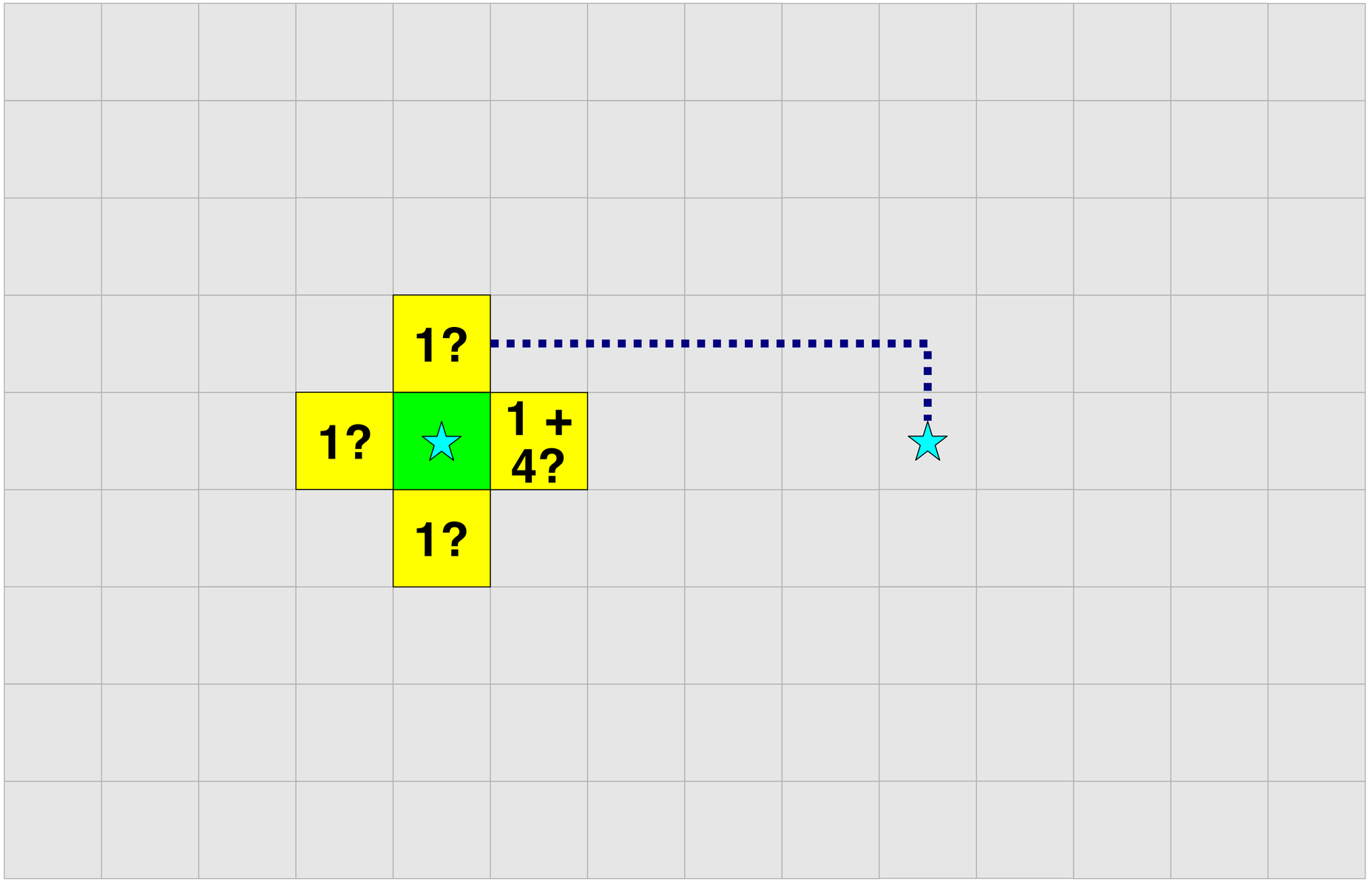


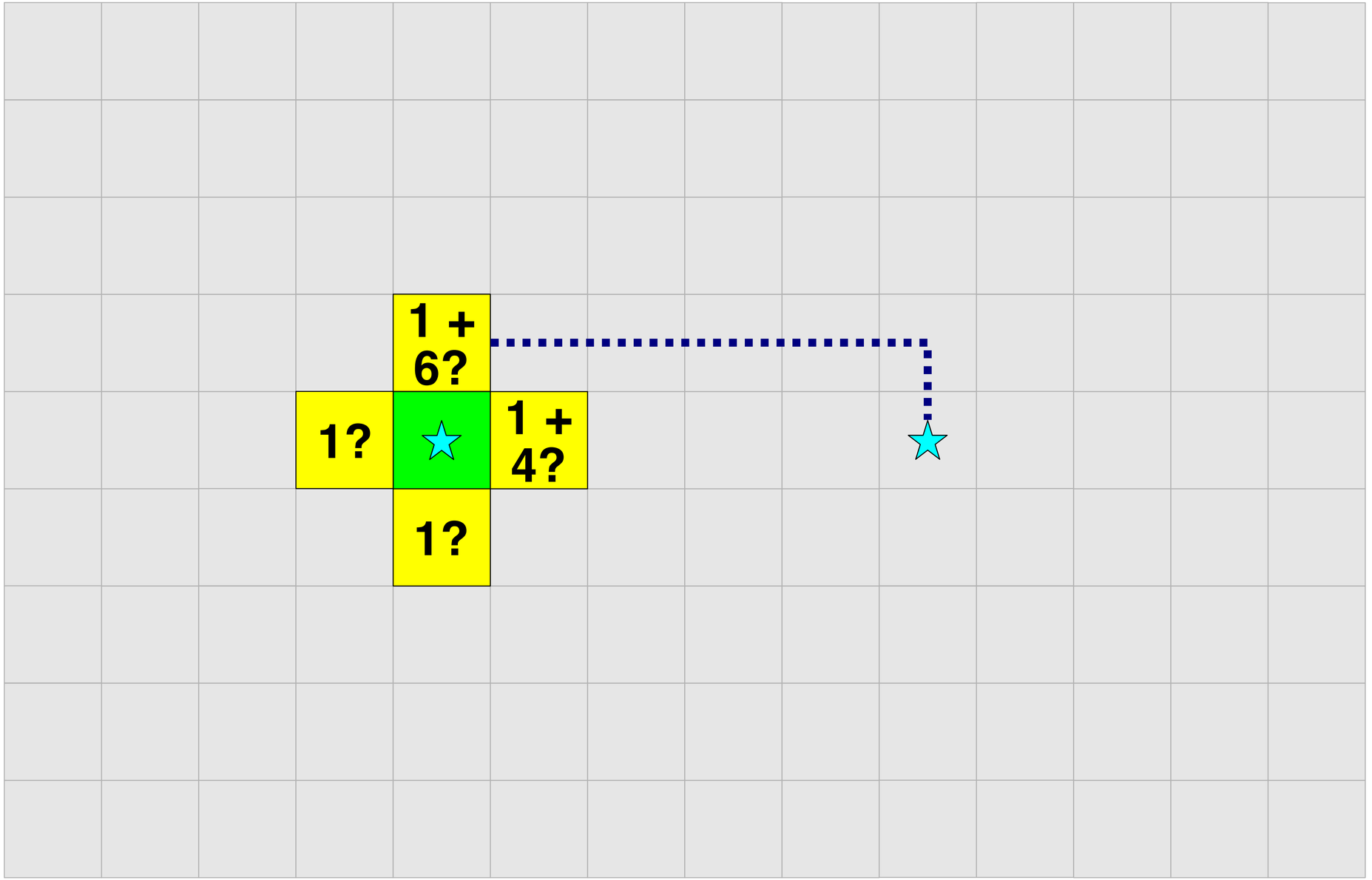


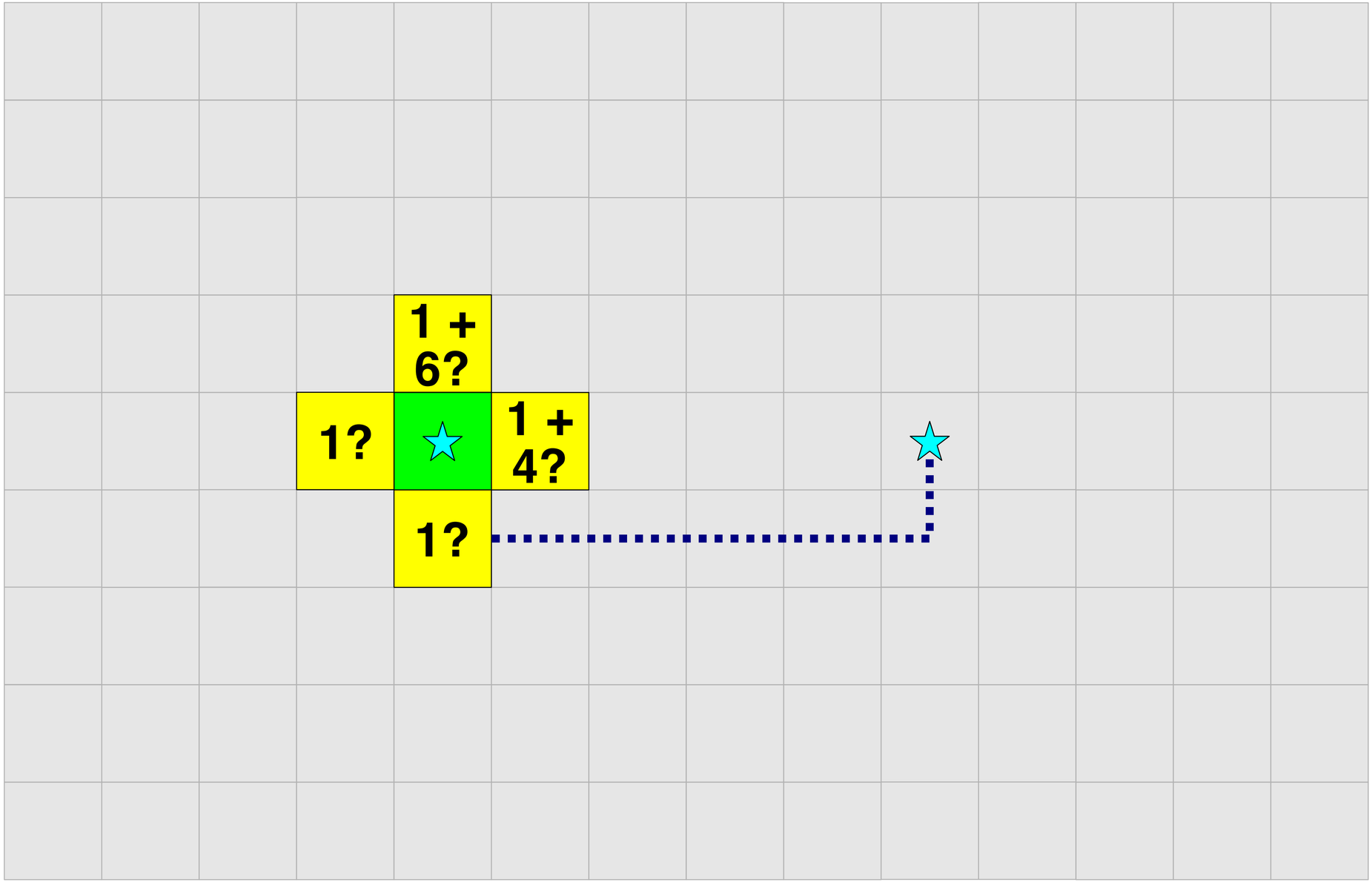

1?
1?
1?
1?

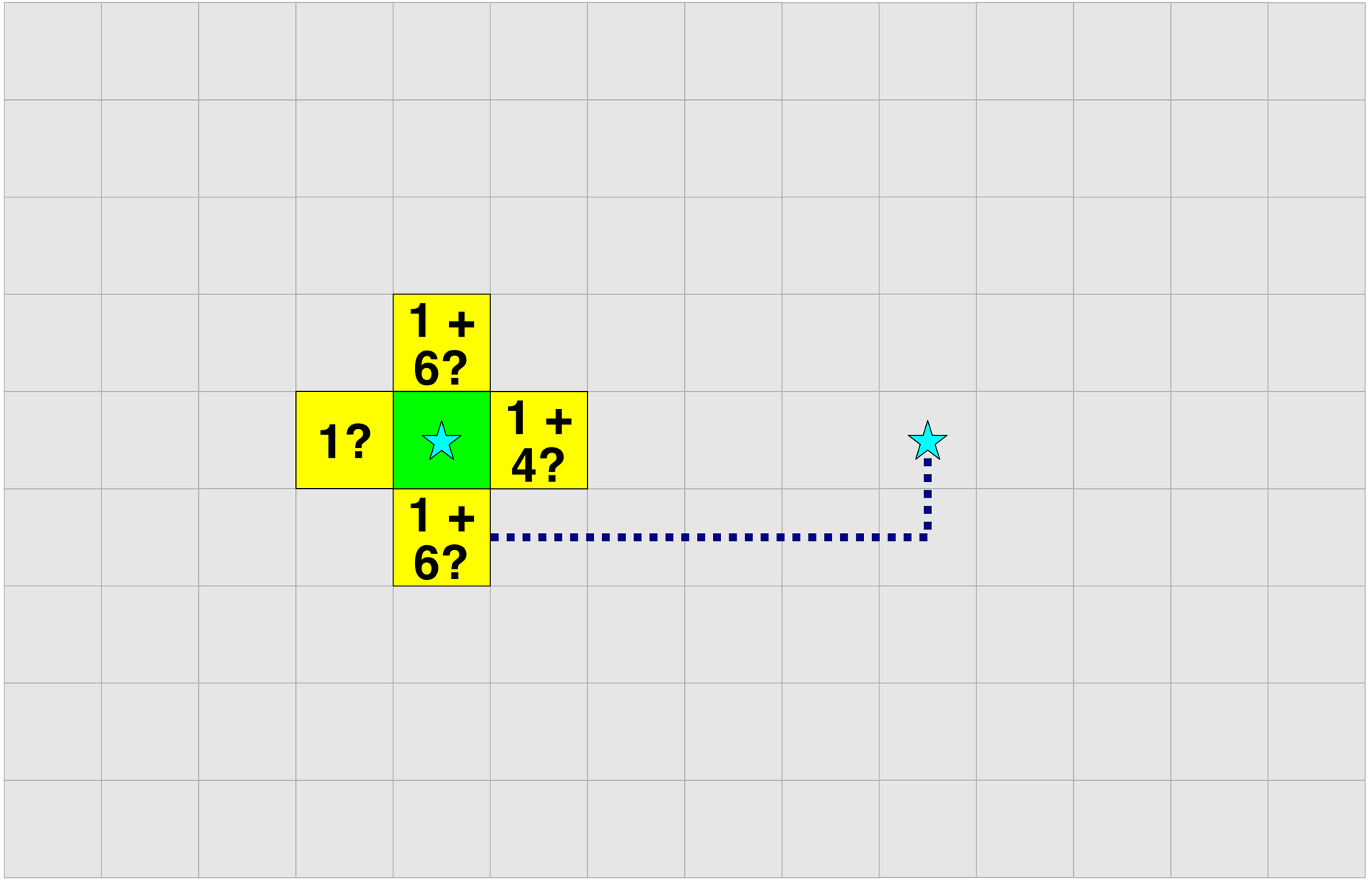

1?
1?
1?
1?

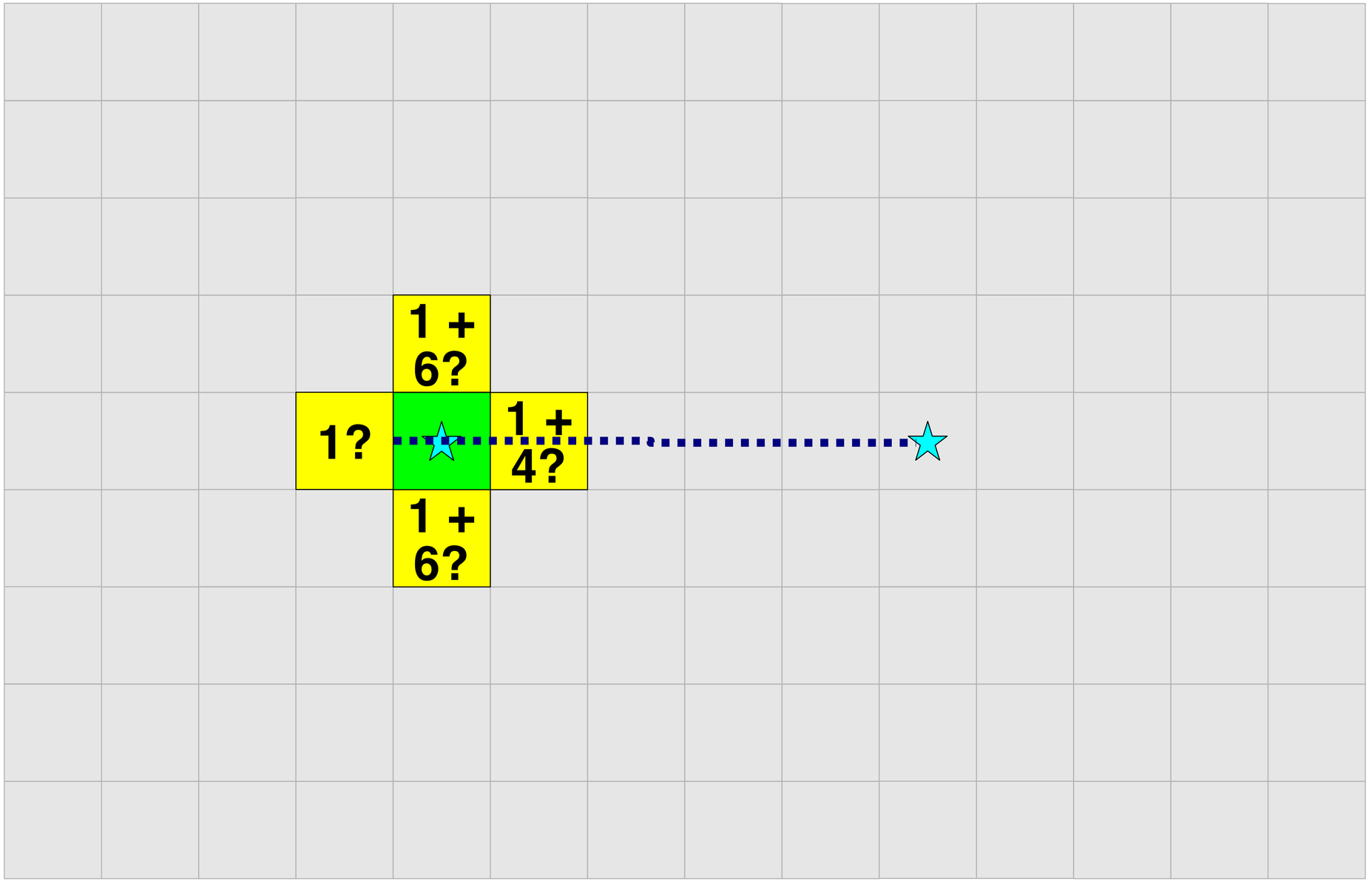

1?
1?
1 +
4?
1?

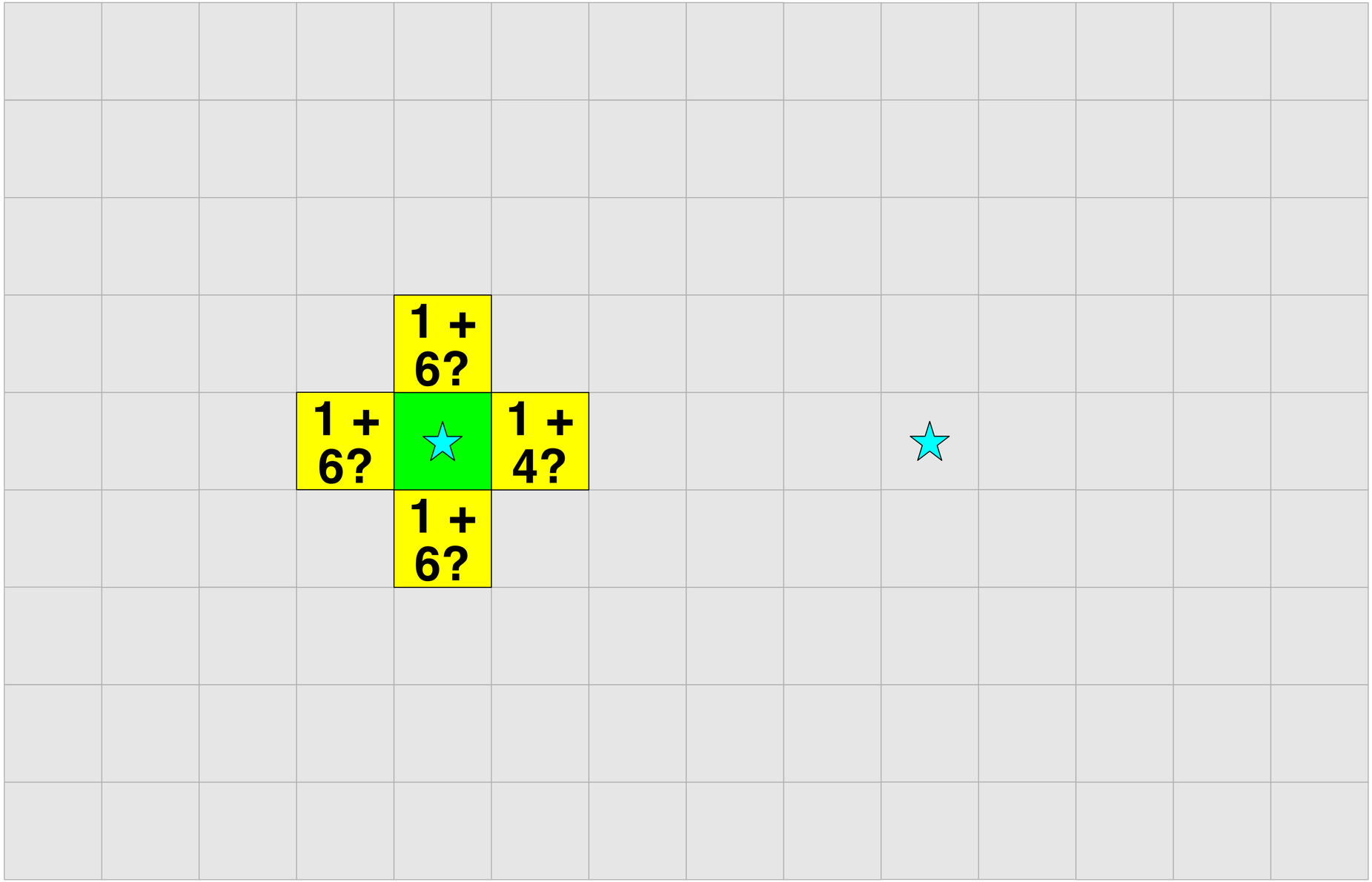

1?
1?
1 +
4?
1?

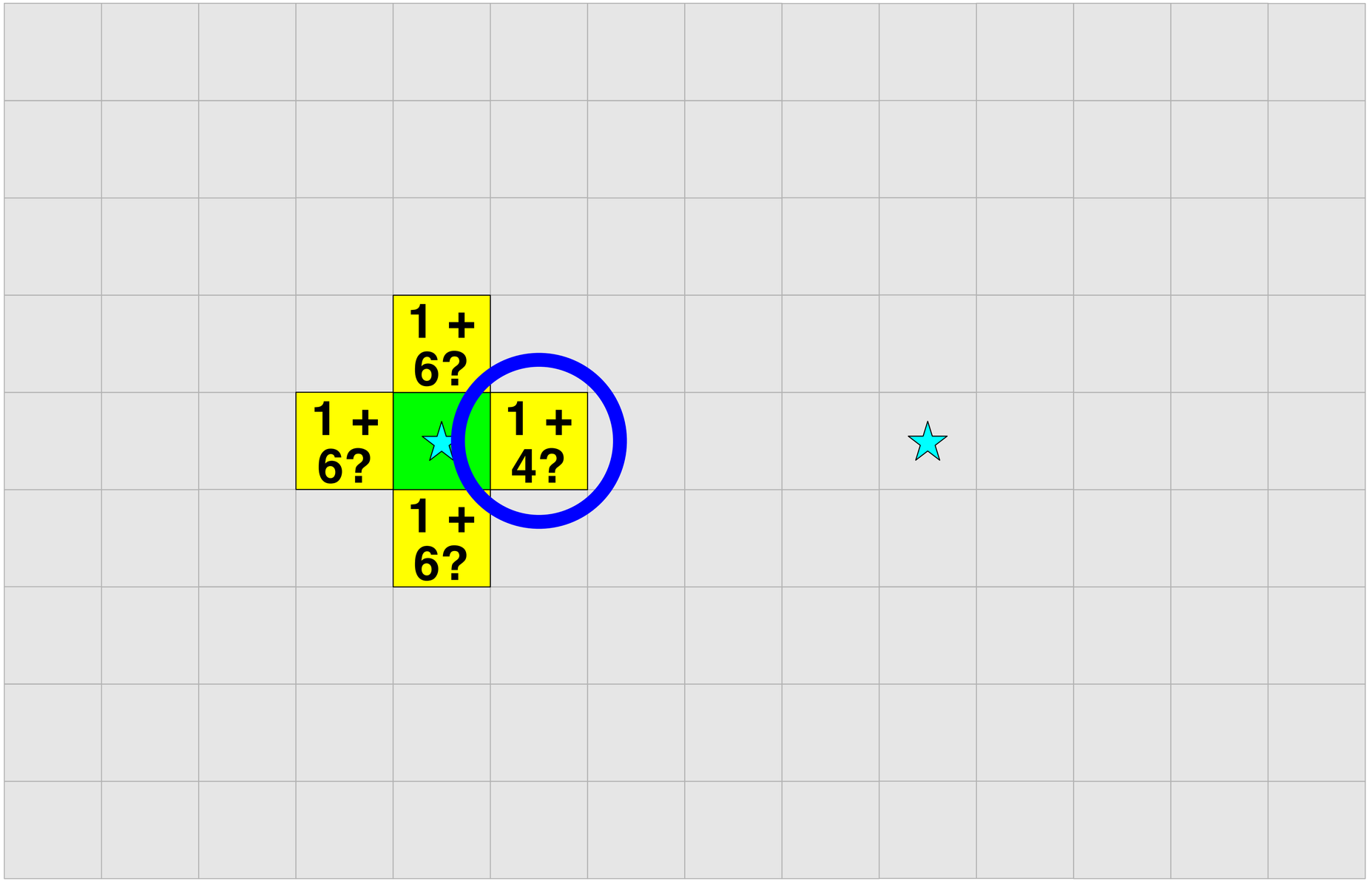

1 +
6?
1?
1 +
4?
1?


1 +
6?
1?
1 +
4?
1?

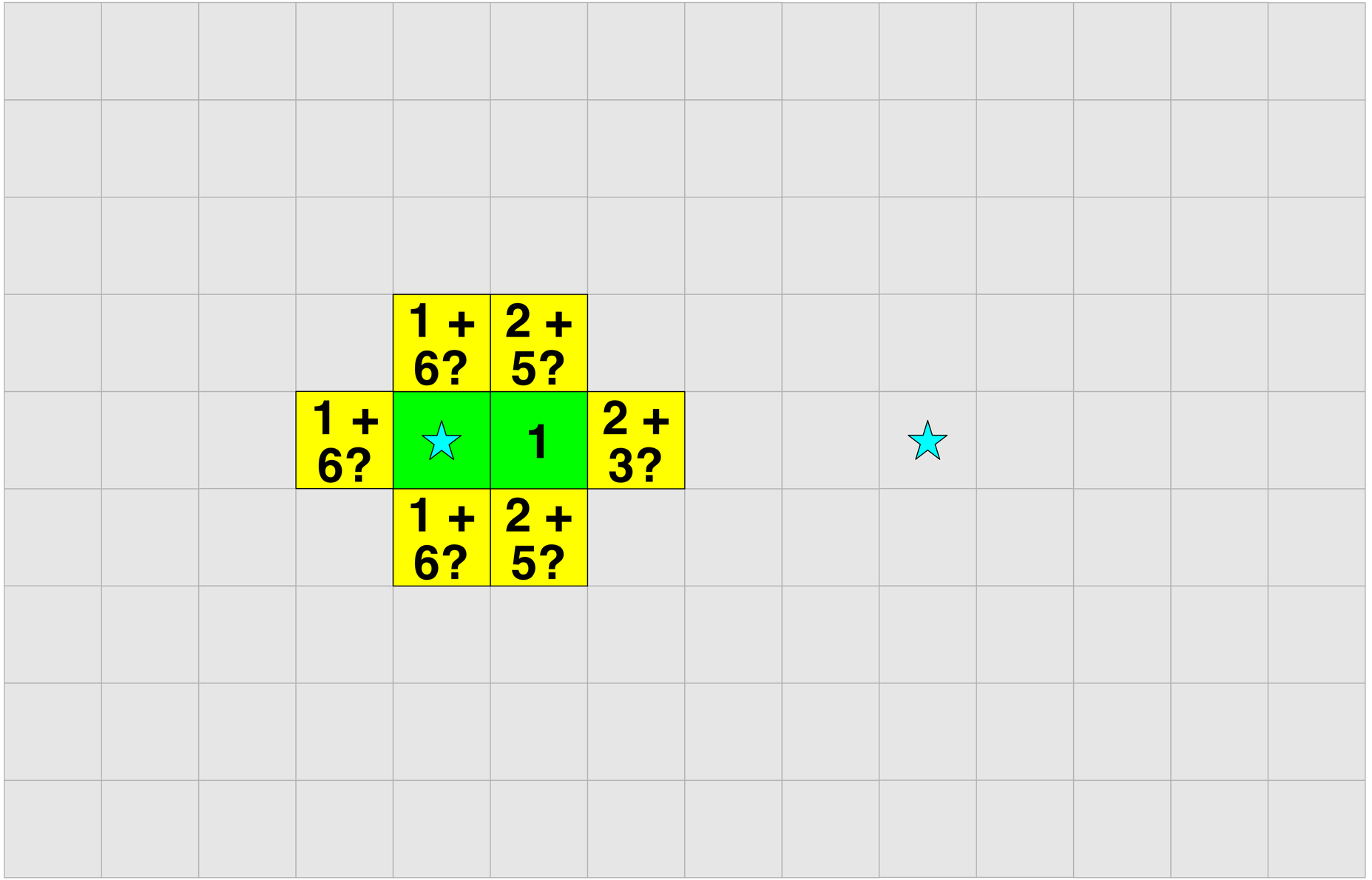

1 +
6?
1?
1 +
4?
1 +
6?

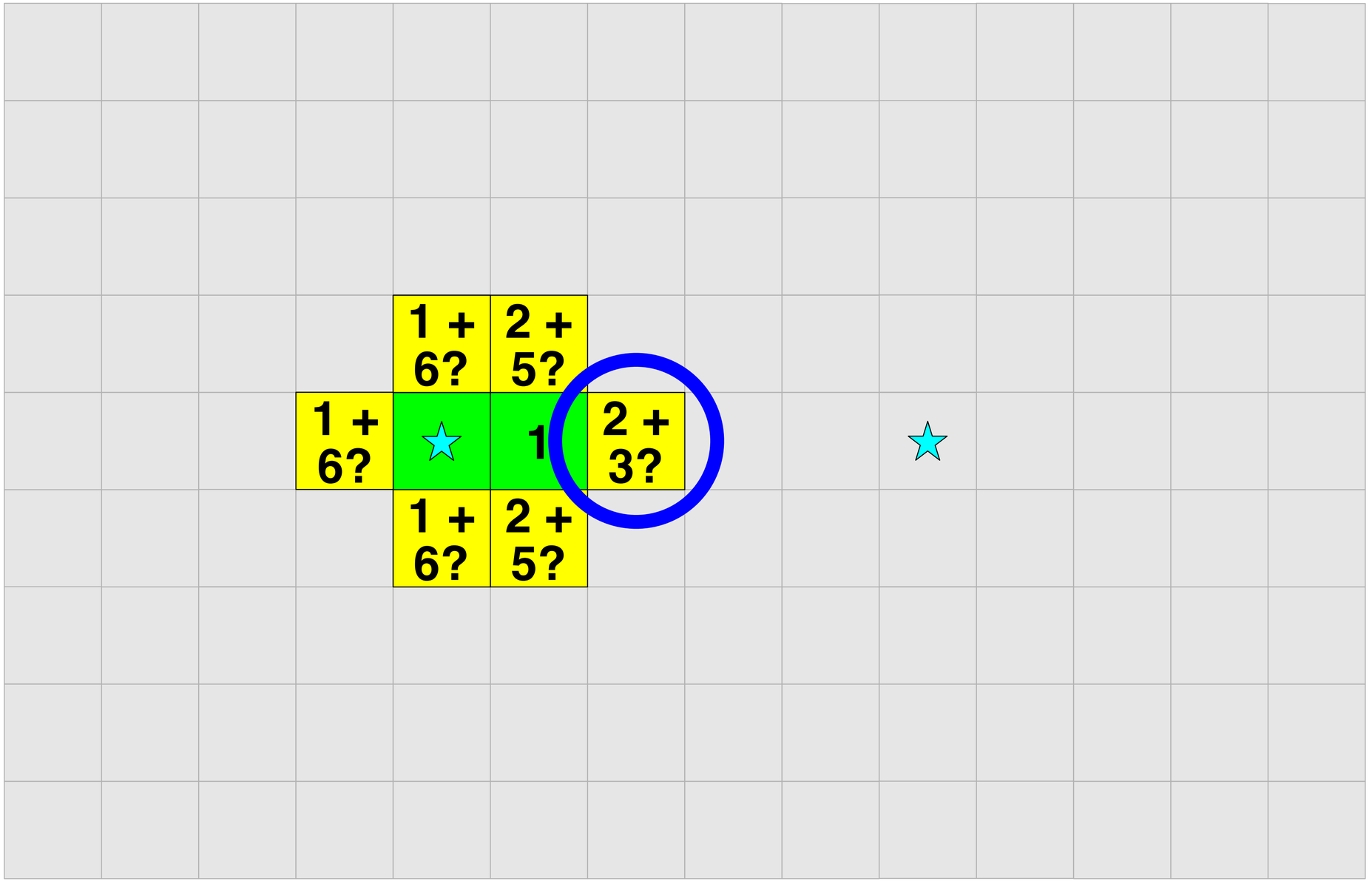

1 +
6?
1?
1 +
4?
1 +
6?

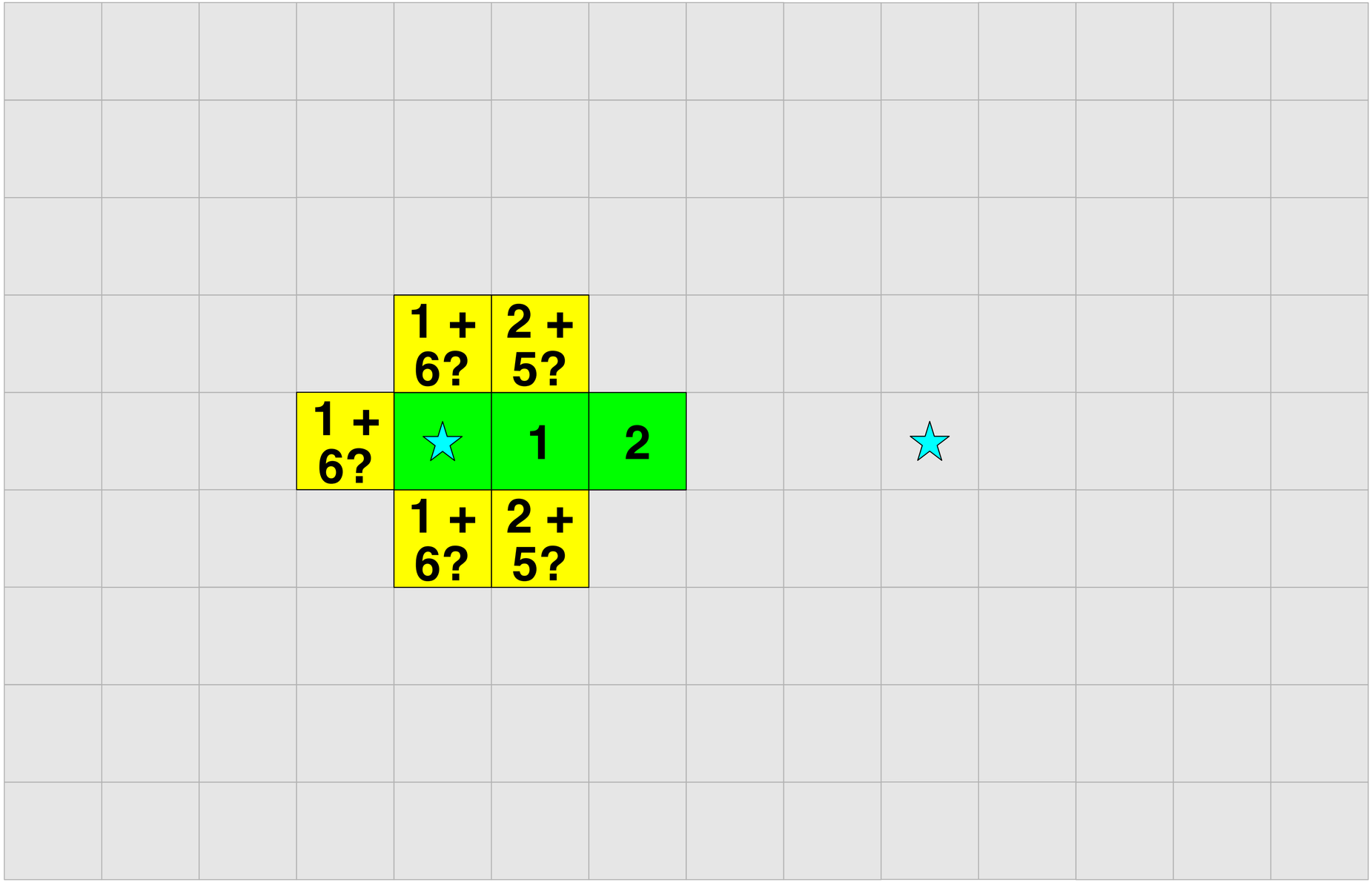

1 +
6?
1 +
6?
1 +
4?
1 +
6?

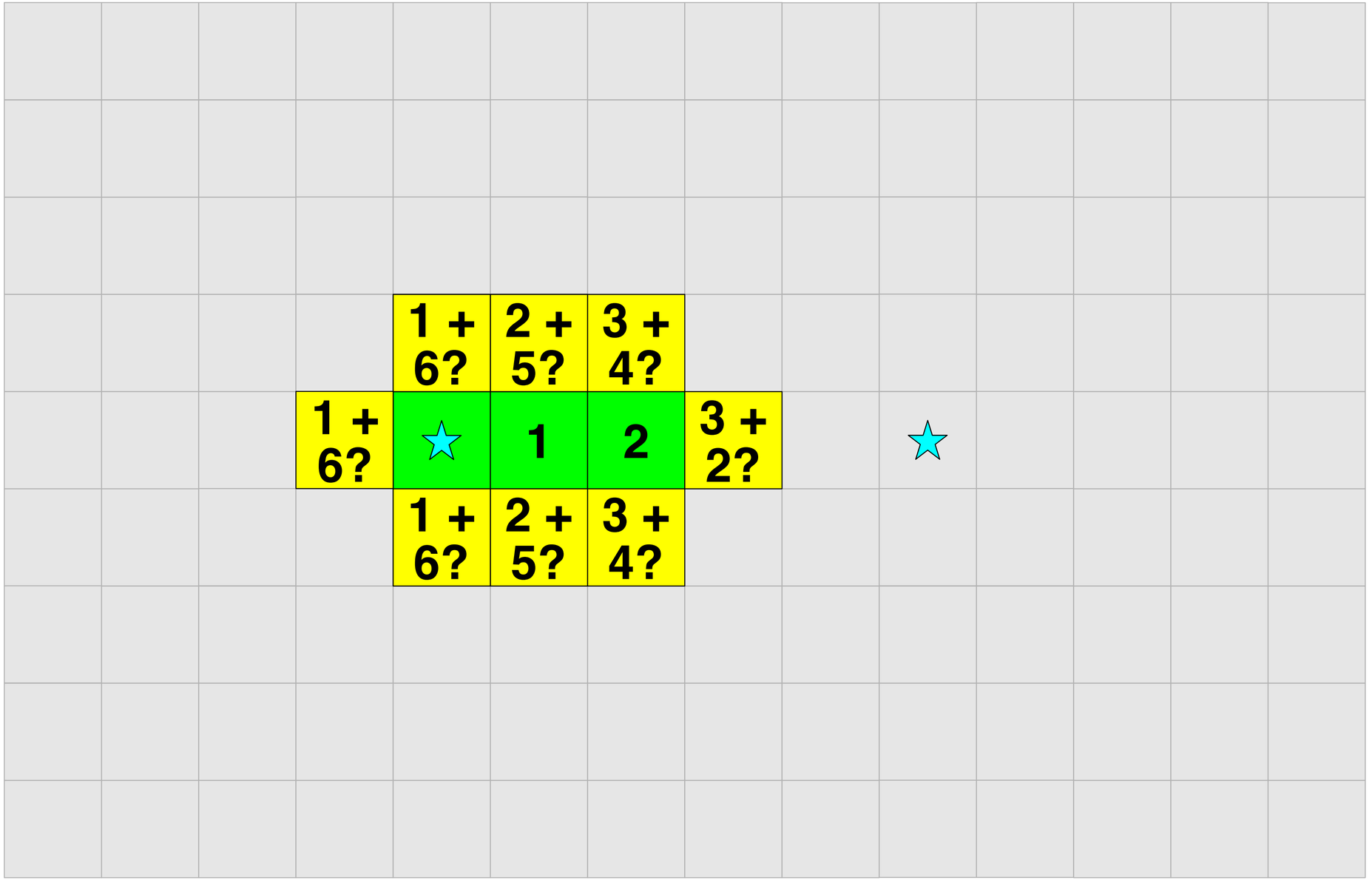

1 +
6?
1 +
6?
1 +
4?
1 +
6?

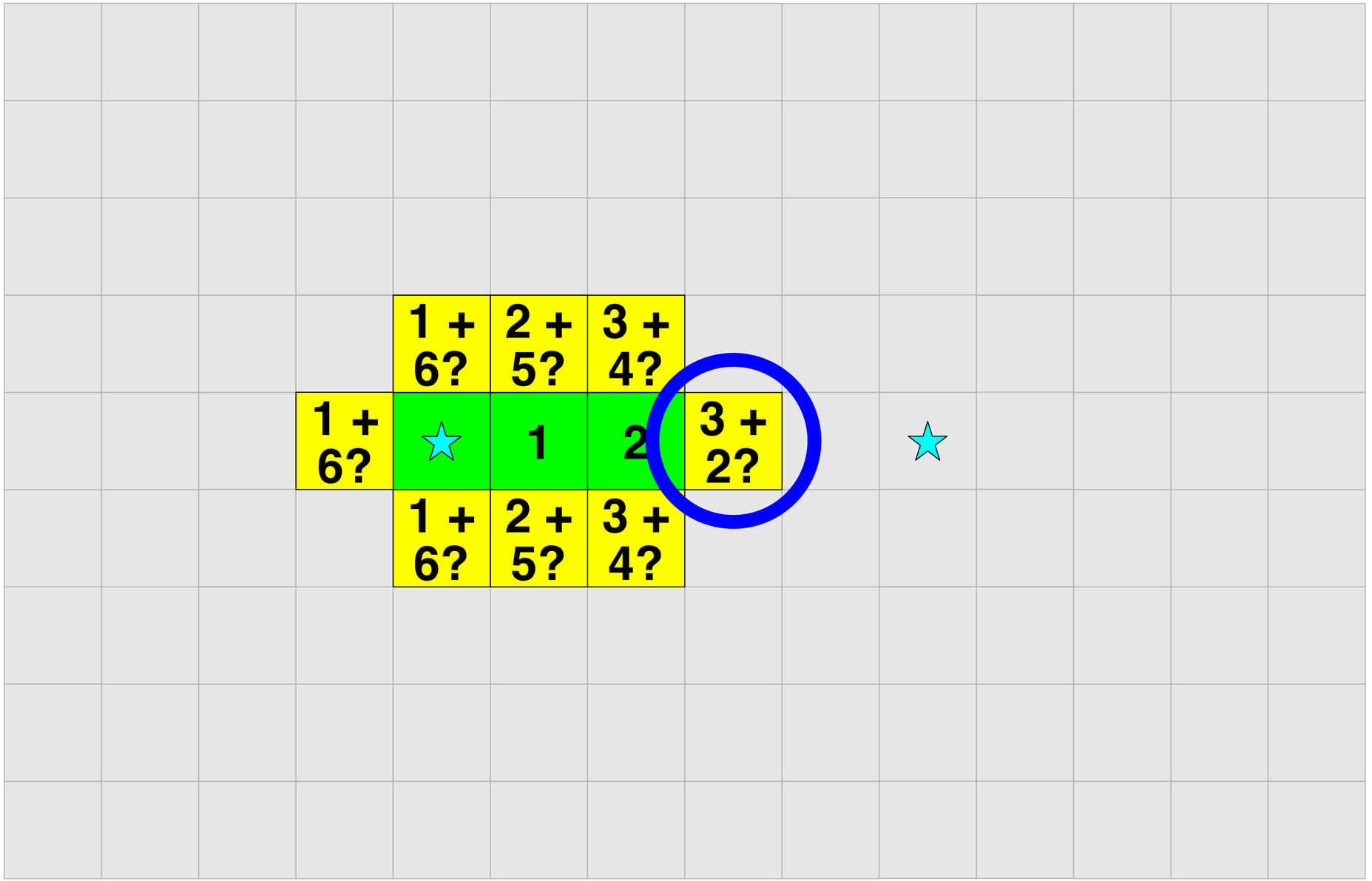

1 +
6?
1 +
6?
1
1 +
6?

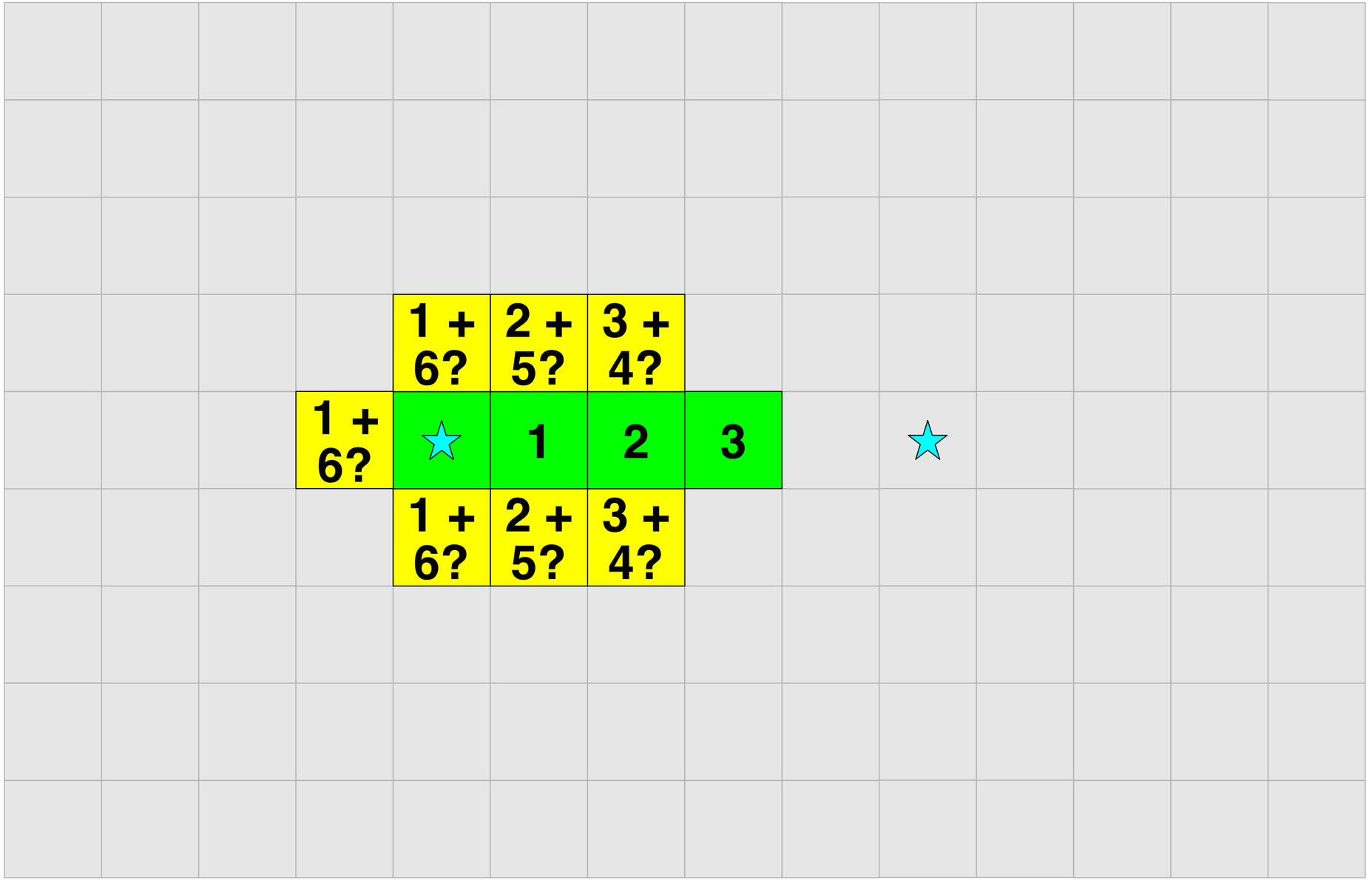

1 + 6?
2 + 5?
1 + 6?
1
2 + 3?
1 + 6?
2 + 5?

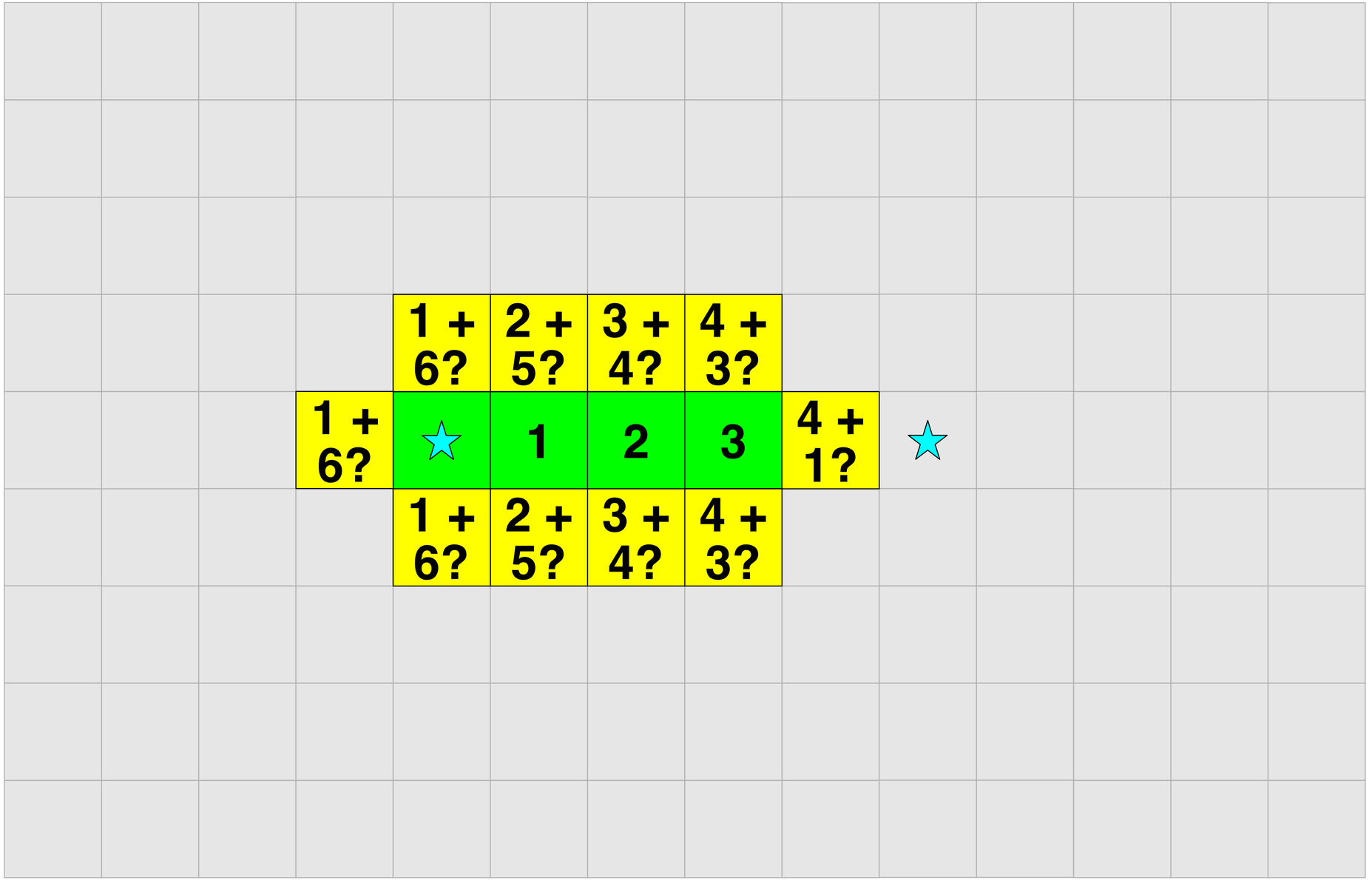

1 +
6?
2 +
5?
1 +
6?
1
2 +
3?
1 +
6?
2 +
5?

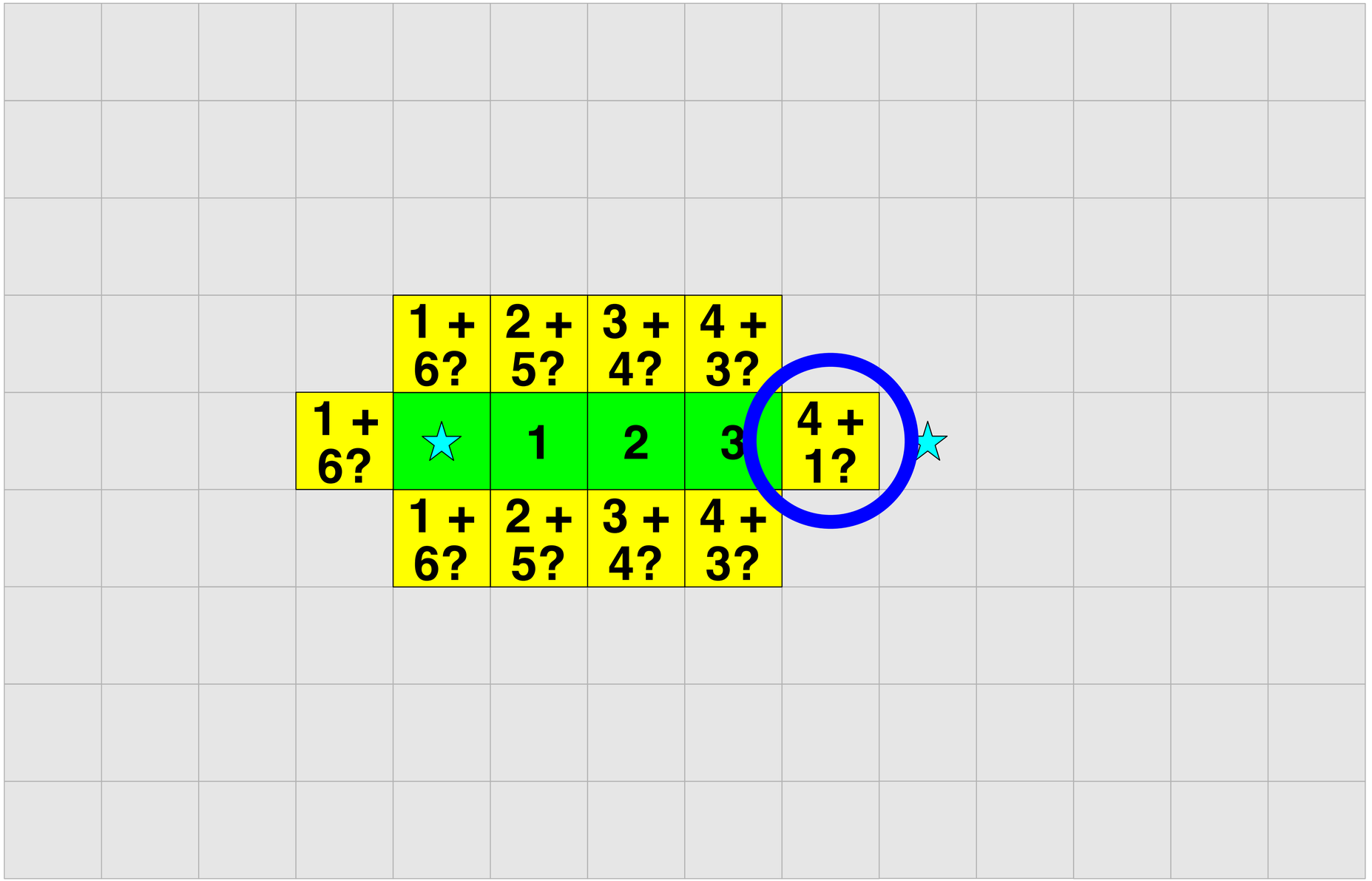

1 + 6?
2 + 5?
1 + 6?
1
2
1 + 6?
2 + 5?

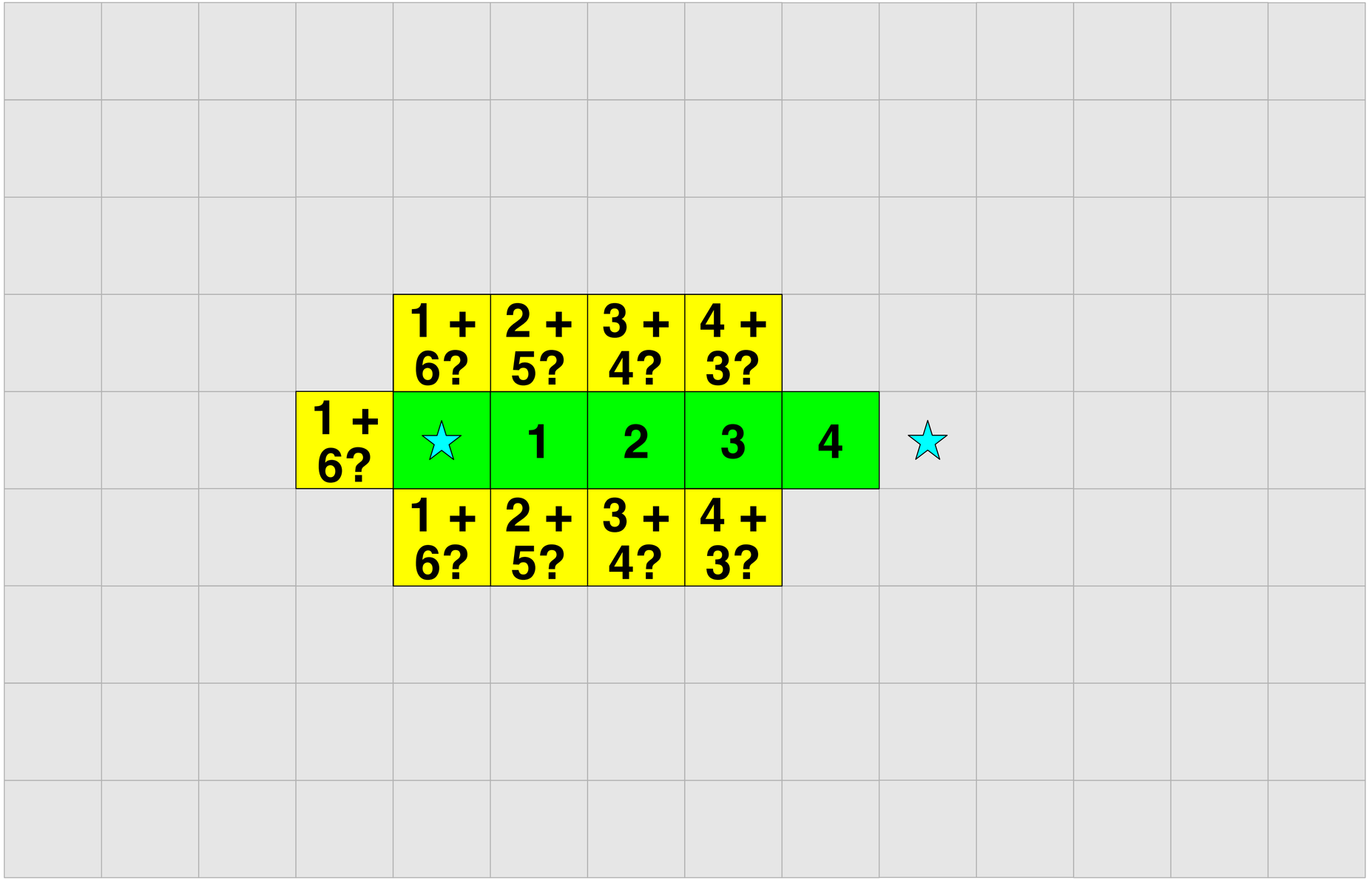

1 + 6?
2 + 5?
3 + 4?
1 + 6?
1
2
3 + 2?
1 + 6?
2 + 5?
3 + 4?

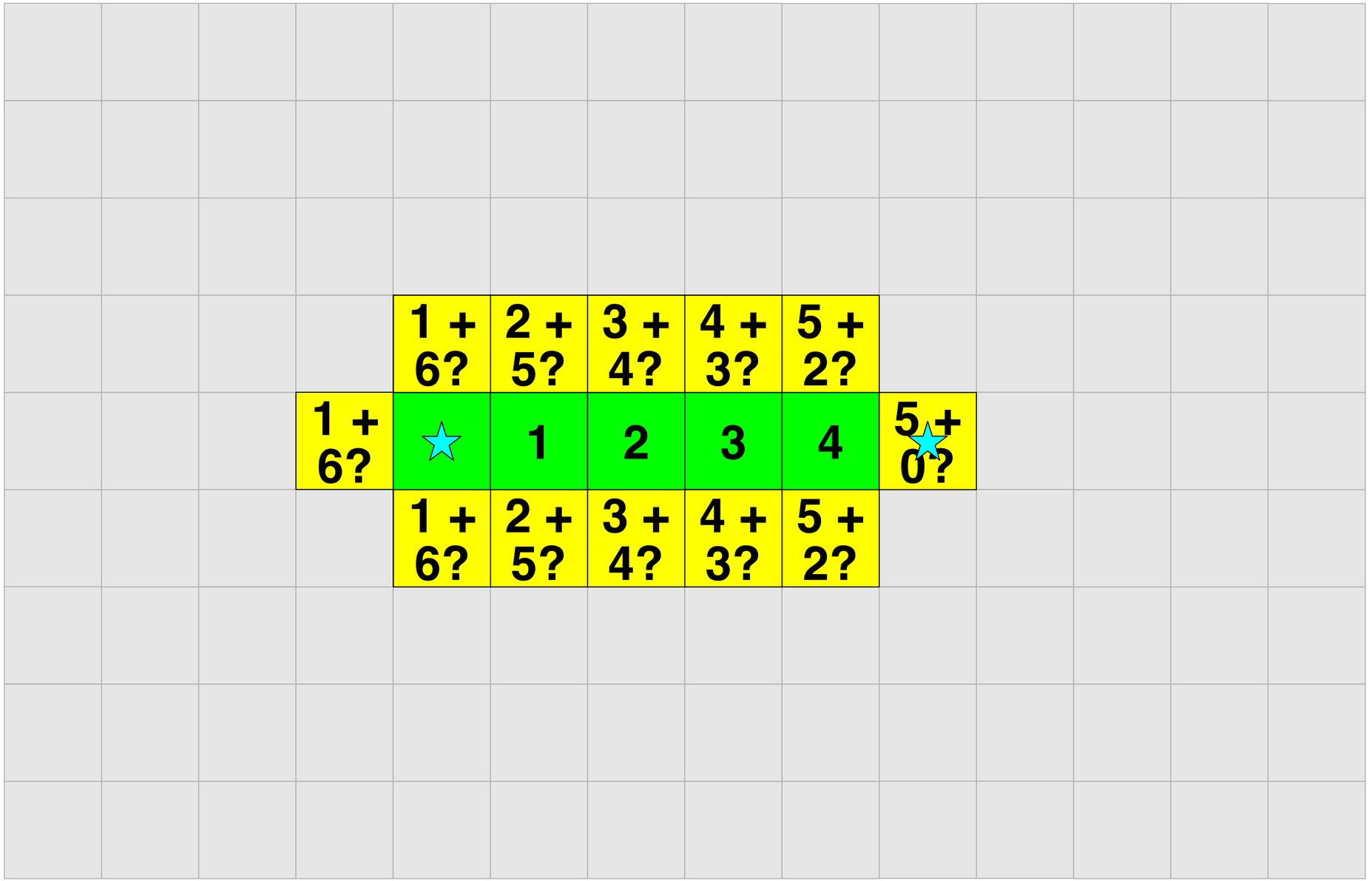

1 + 6?
2 + 5?
3 + 4?
1 + 6?
1
2
3 + 2?
1 + 6?
2 + 5?
3 + 4?

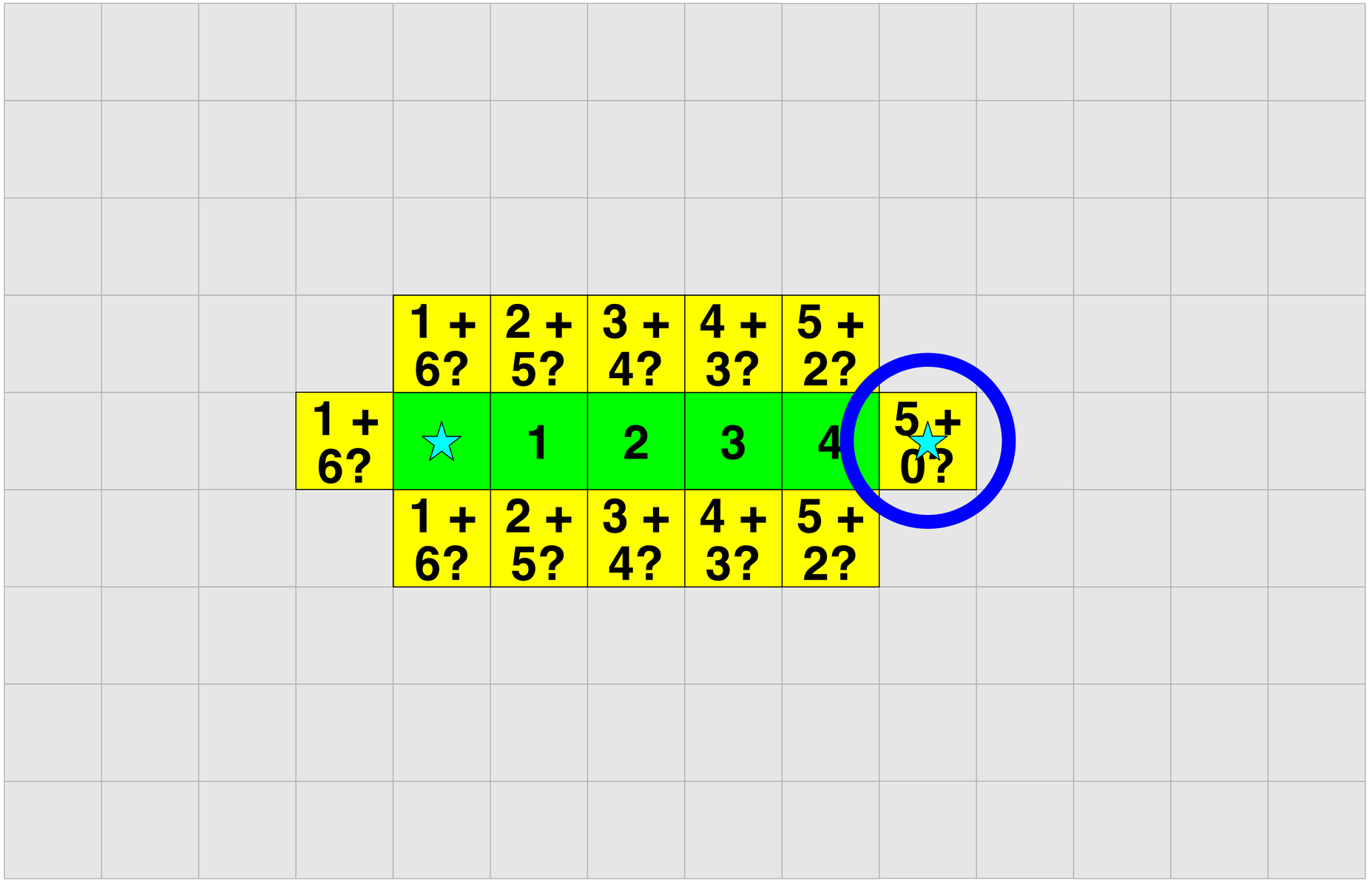

1 + 6?
2 + 5?
3 + 4?
1 + 6?
1
2
3
1 + 6?
2 + 5?
3 + 4?

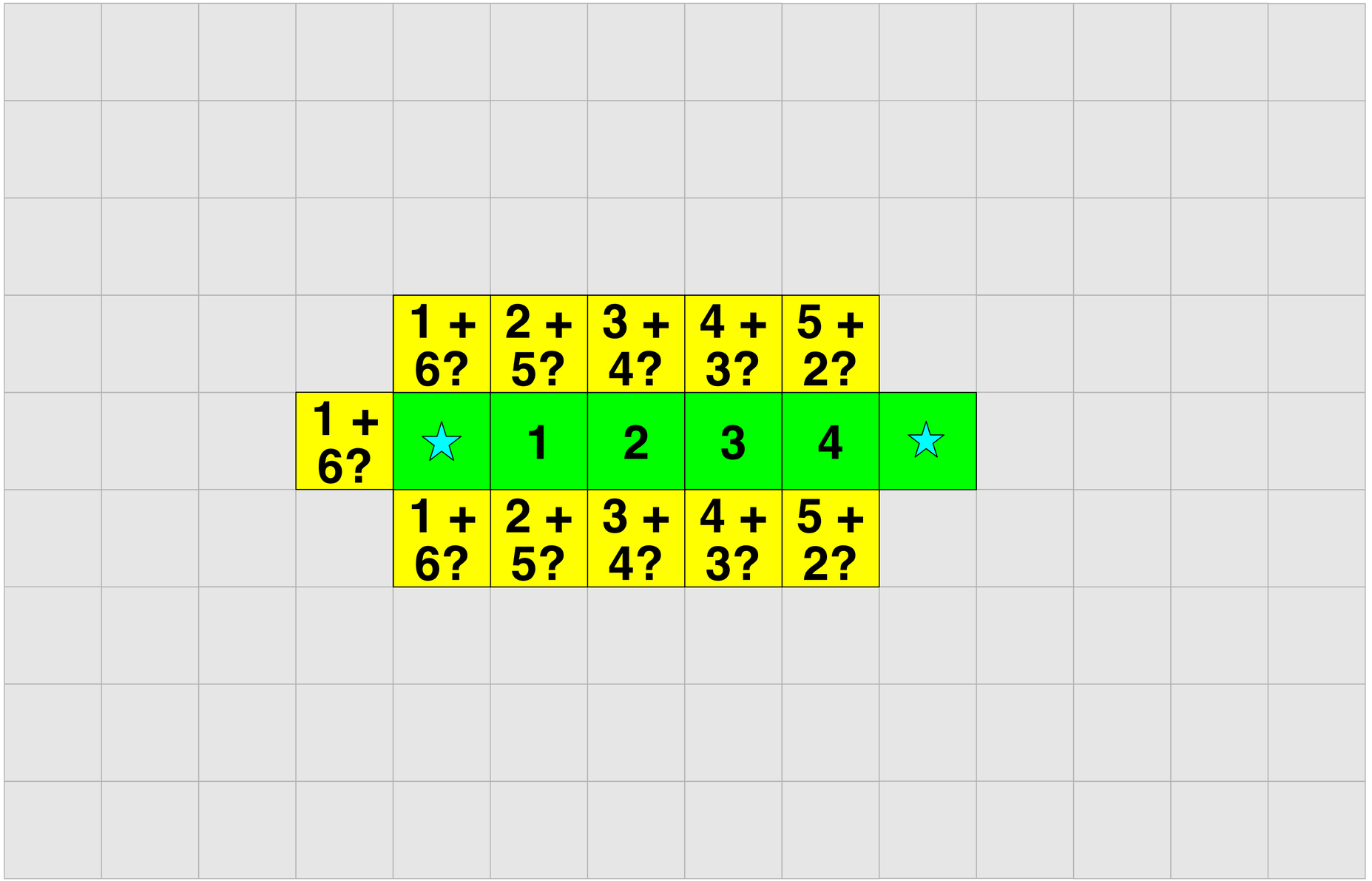

1 + 6?
2 + 5?
3 + 4?
4 + 3?
1 + 6?
1
2
3
4 + 1?
1 + 6?
2 + 5?
3 + 4?
4 + 3?


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

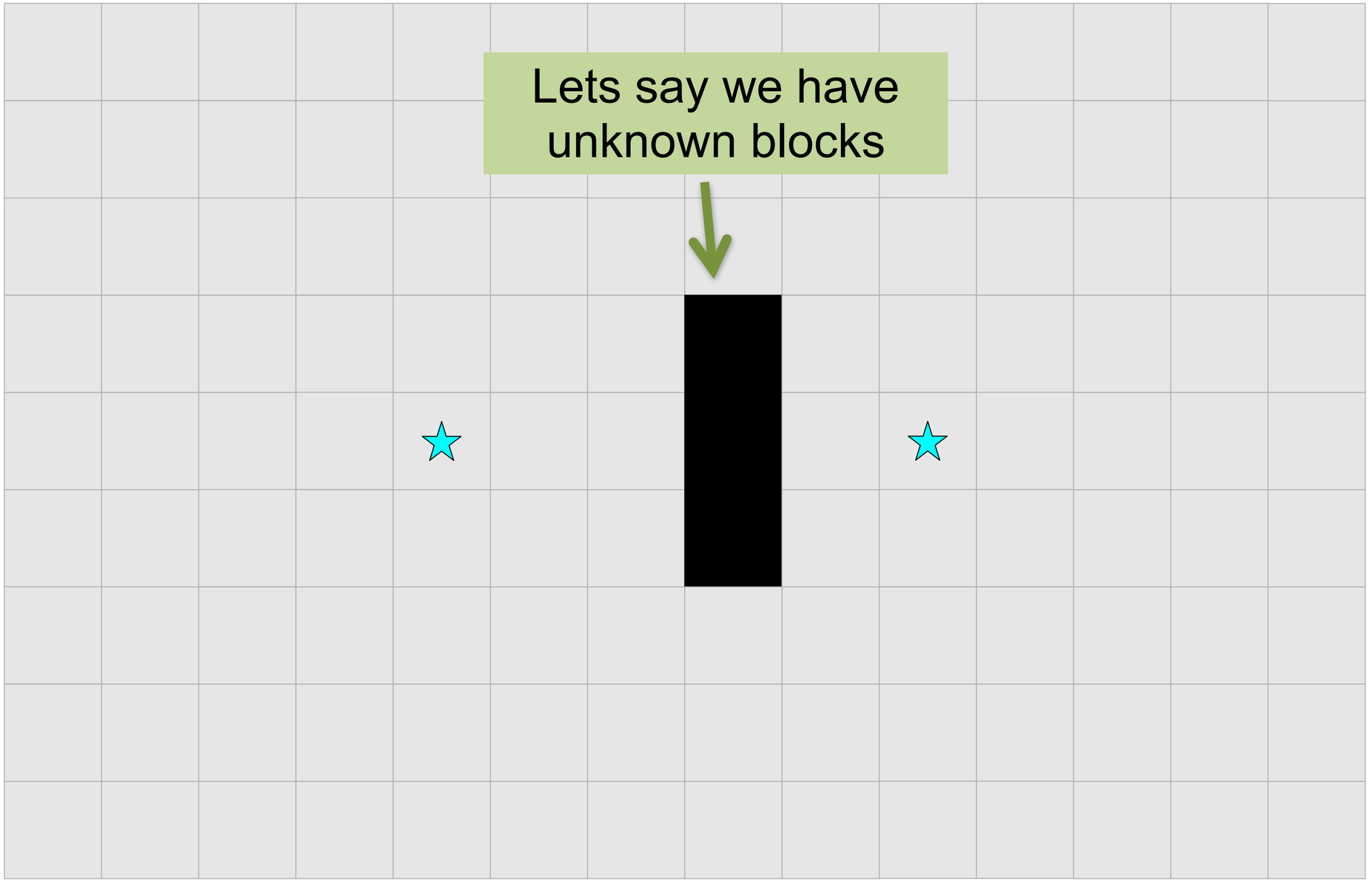

1 + 6?
2 + 5?
3 + 4?
4 + 3?
5 + 2?
1 + 6?
1
2
3
4
5 + 0?
1 + 6?
2 + 5?
3 + 4?
4 + 3?
5 + 2?


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

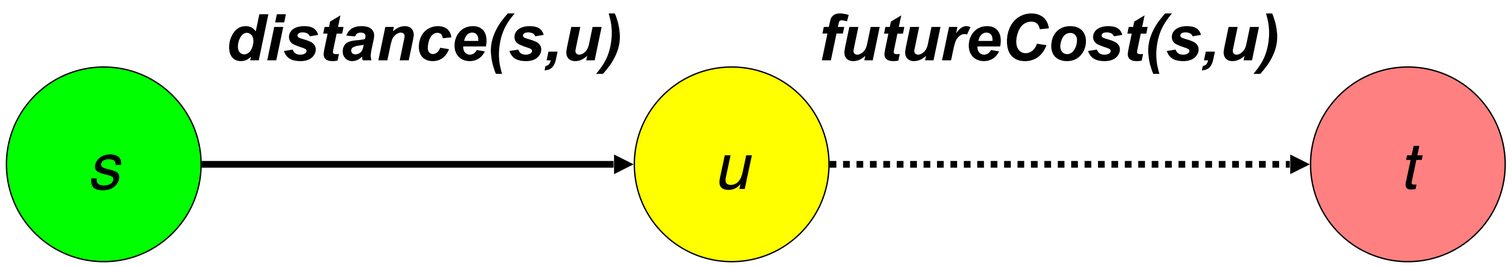
That was easy…

… a little too easy


Lets say we have unknown blocks

Ideal Priority

priority(u) = distance(s, u) + futureCost(u, t)

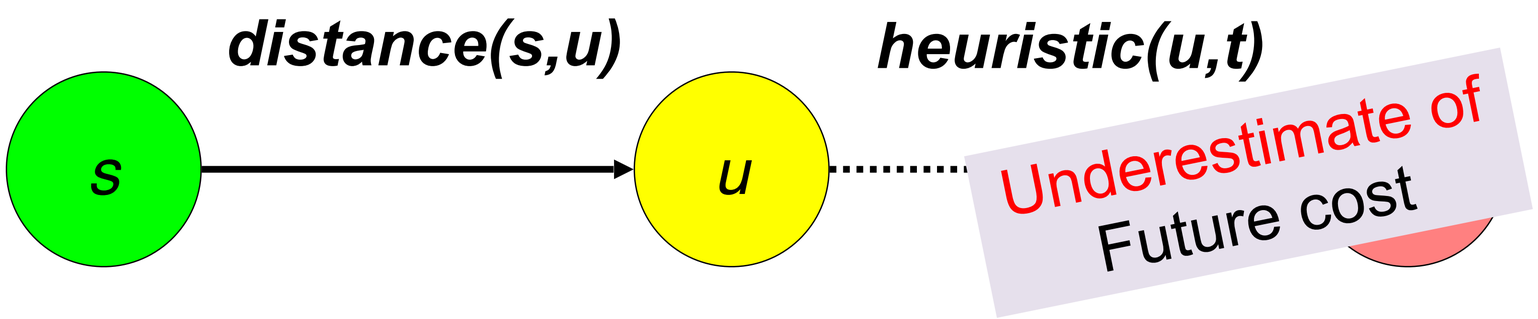


Priority of the path that ends in u

*note we will revise this slightly

A* Priority

priority(u) = distance(s, u) + heuristic(u, t)



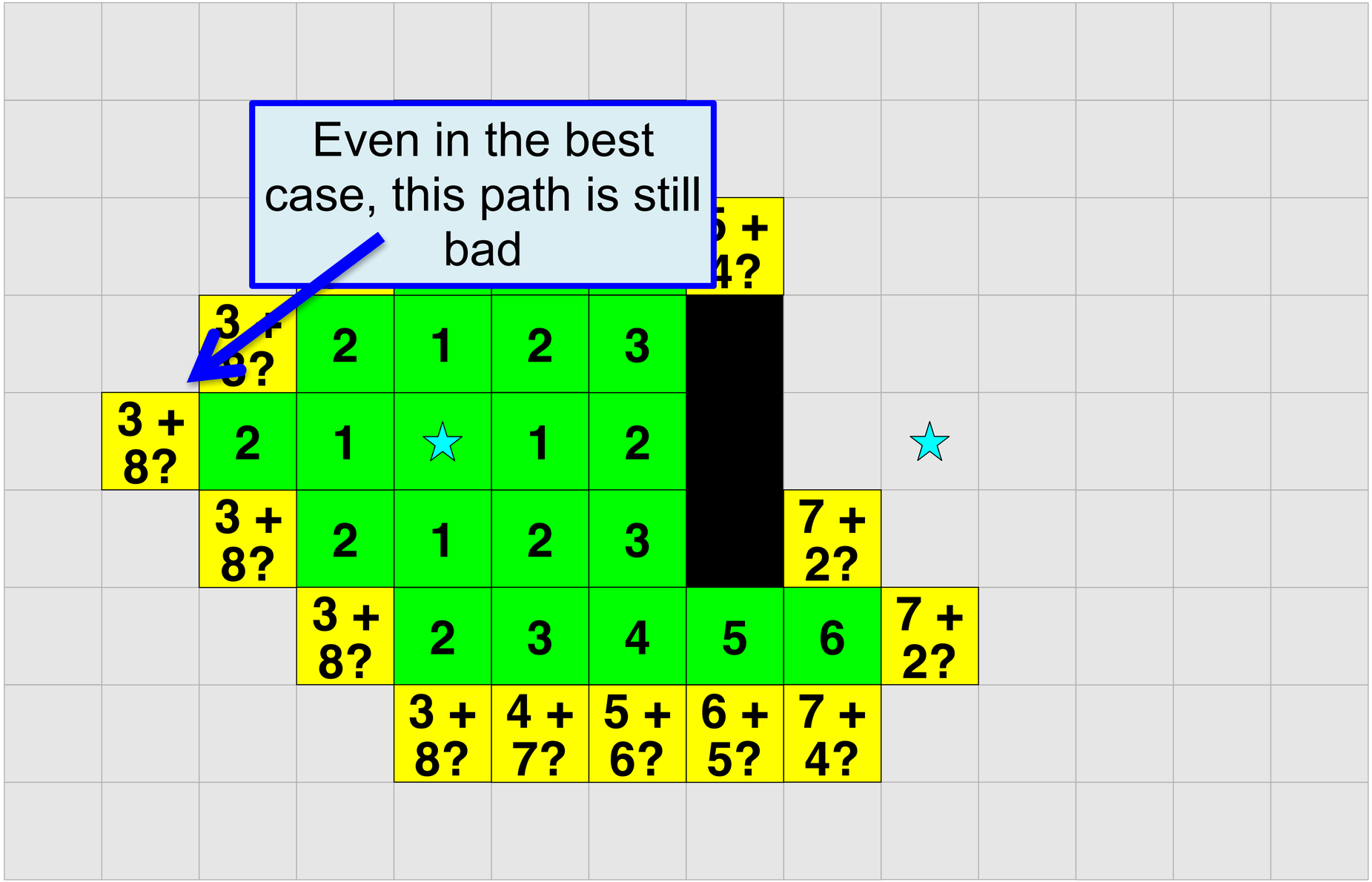
Priority of the path that ends in u

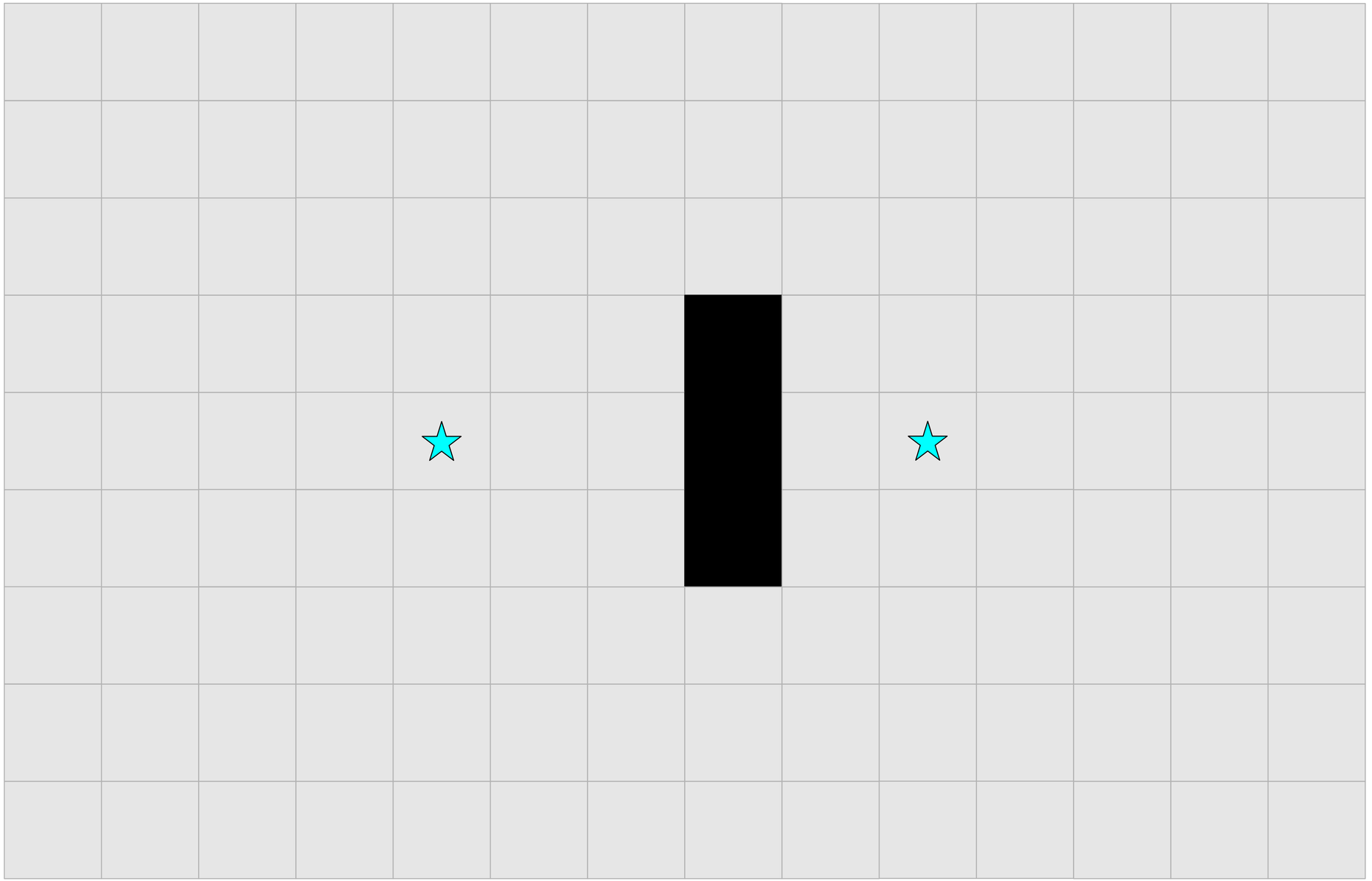
Admissible Heuristic

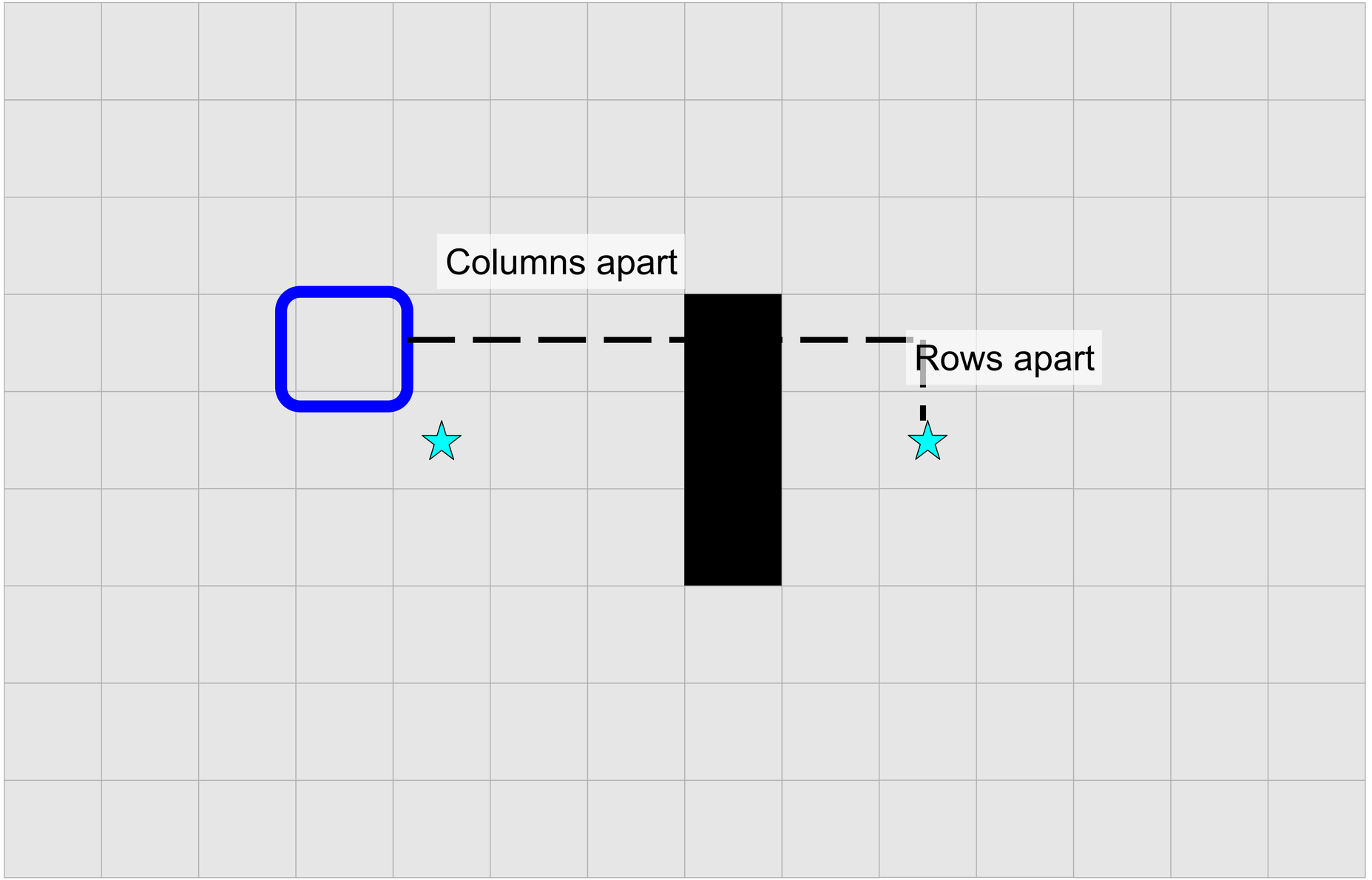
Definition: An admissible heuristic always underestimates the true cost.

Thus: "even in the best case scenario, this path is still terrible…"

Admissible Heuristic






Columns apart
Rows apart

"Manhattan" distance

```
function h(start,goal) {
  dRows = abs(start.row - goal.row);
  dCols = abs(start.col - goal.col);
  return dRows + dCols
}
```

Recall Dijkstra...

```
Make a PriorityQueue todo-list of paths
Put a path with just the start in the todo-list
While the todo-list isn't empty
 1. Take a path out of the todo-list
 2. Call the last node in the path "currNode"
 3. If "currNode" is the goal, you are done.
 4. If you have seen currNode before, skip it.
  5. for all neighbors of currNode
        Make a newPath = path + neighbor
        Add the new path to the todo-list
        Priority = pathLength
```

A Star

```
Make a PriorityQueue todo-list of paths
Put a path with just the start in the todo-list
While the todo-list isn't empty
 1. Take a path out of the todo-list
 2. Call the last node in the path "currNode"
 3. If "currNode" is the goal, you are done.
 4. If you have seen currNode before, skip it.
  5. for all neighbors of currNode
        Make a newPath = path + neighbor
        Add the new path to the todo-list
        Priority = pathLength + h(neighbor, goal)
```

A Star

```
Make a PriorityQueue todo-list of paths
Put a path with just the start in the todo-list
While the todo-list isn't empty
  1. Take a path out of the todo-list
  2. Call the last node in the path "currNode"
  3. If "c                                    e done.
  4. If yo                                    , skip it.
   5. for all neighbors of currNode
        Make a newPath = path + neighbor
        Add the new path to the todo-list
        Priority = pathLength + h(neighbor, goal)
```

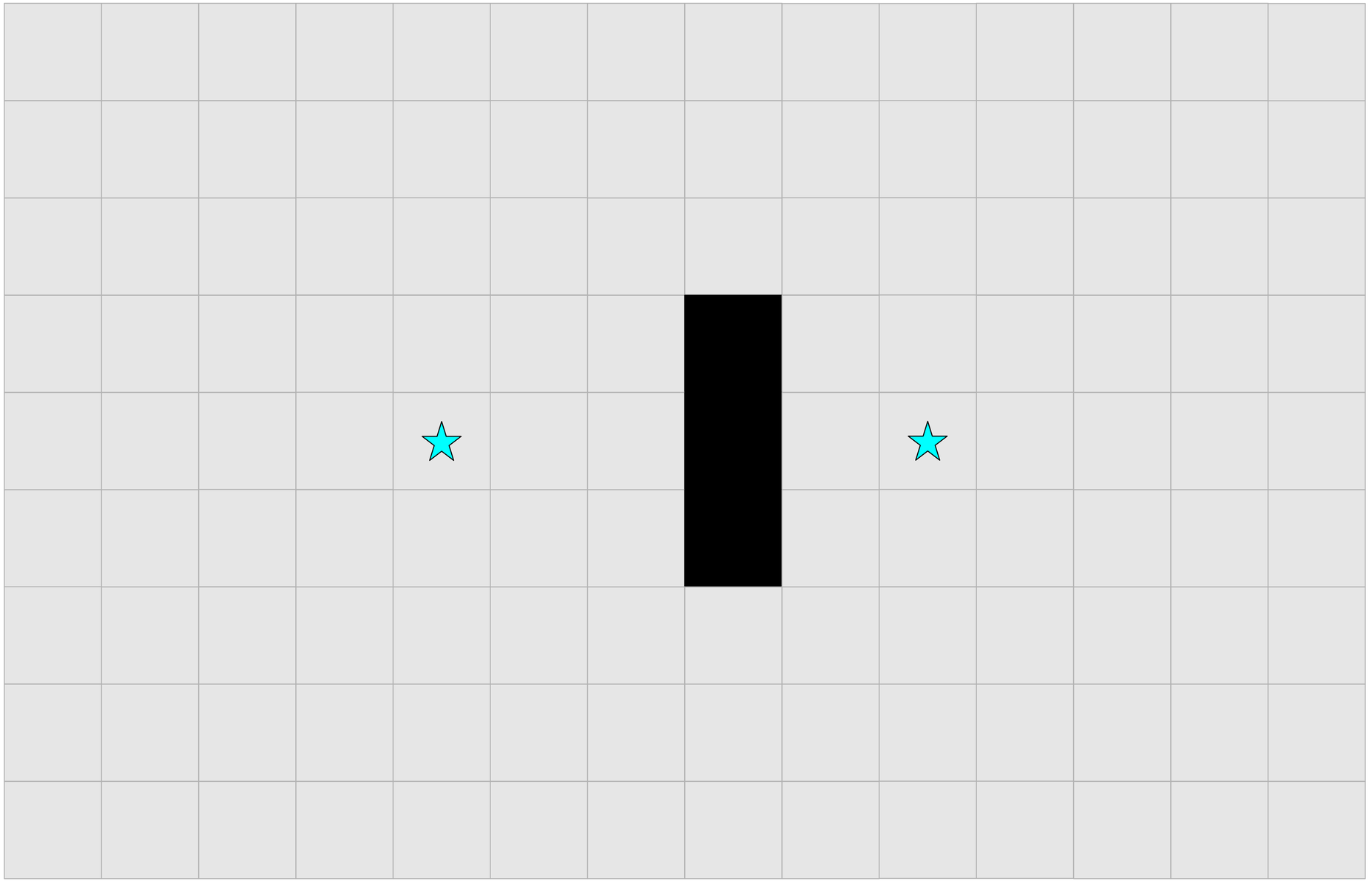
What is the priority of the start path?

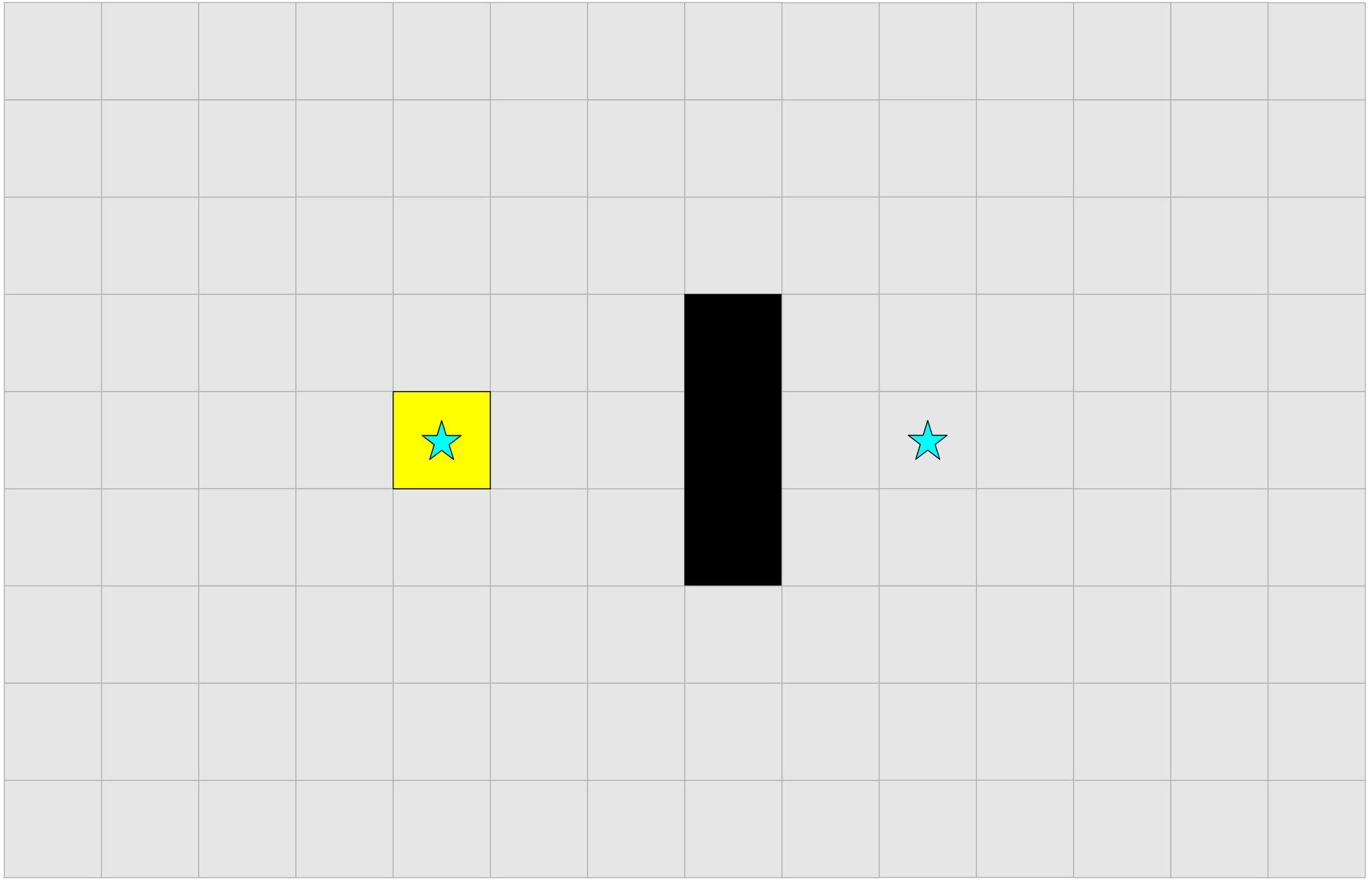
A Star

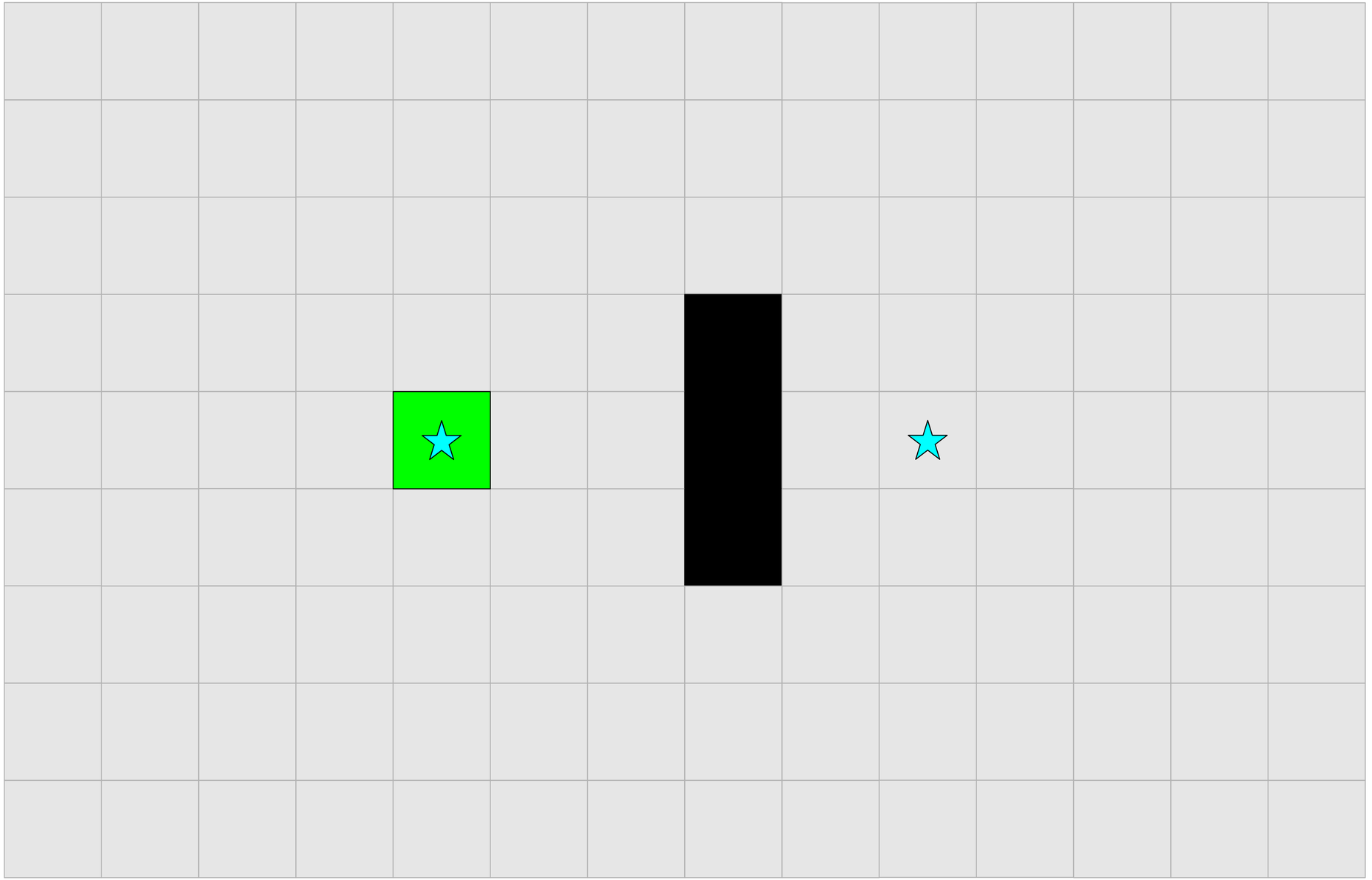
```
Make a PriorityQueue todo-list of paths
Put a path with just the start in the todo-list
While the todo-list isn't empty
  1. Take a path out of the todo-list
  2. Call the last node in the path "currNode"
  3. If "c                                    done.
  4. If yo                                    skip it.
   5. for all neighbors of currNode
        Make a newPath = path + neighbor
        Add the new path to the todo-list
        Priority = pathLength + h(neighbor, goal)
```

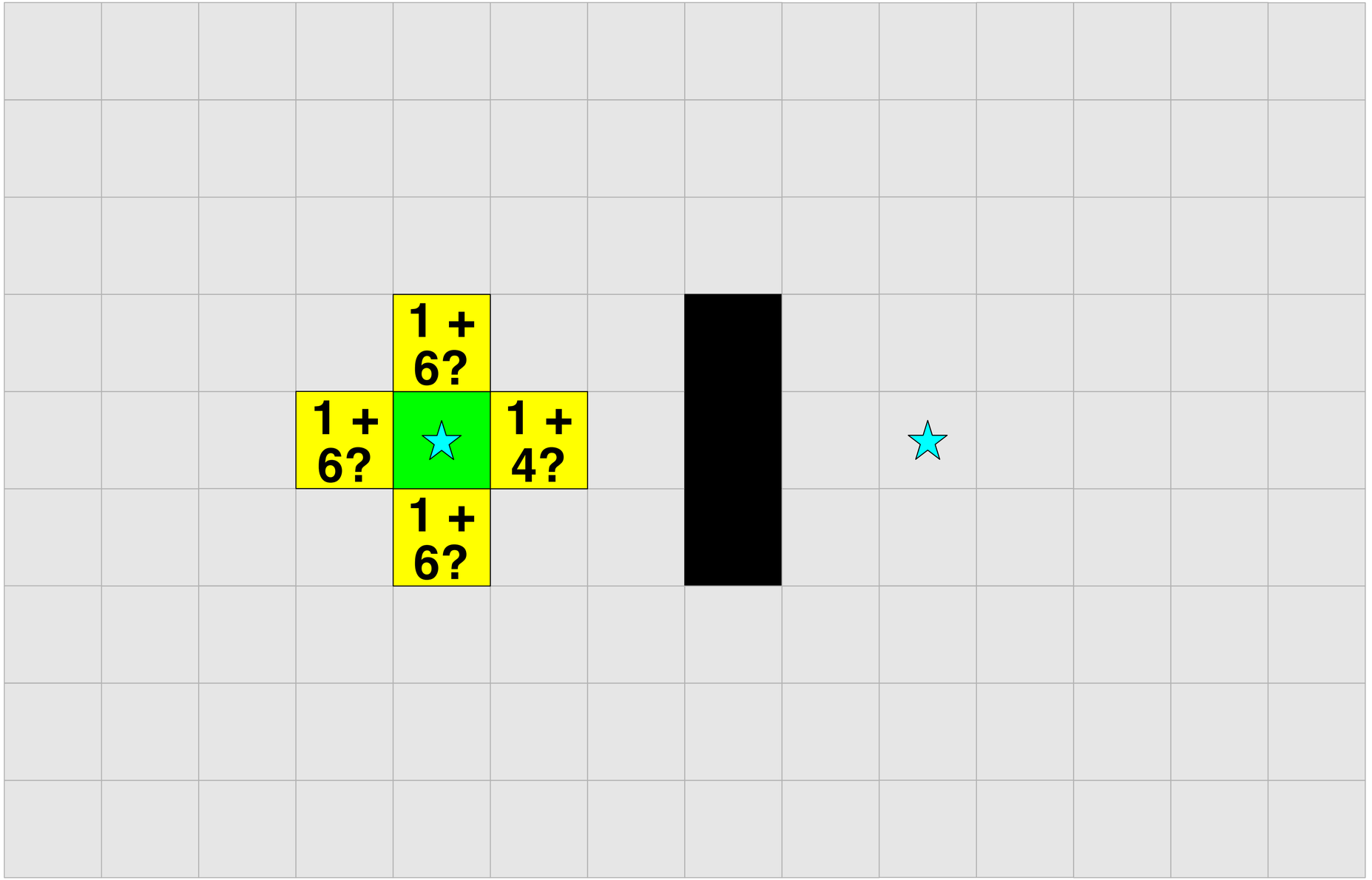
What is the priority of the start path?

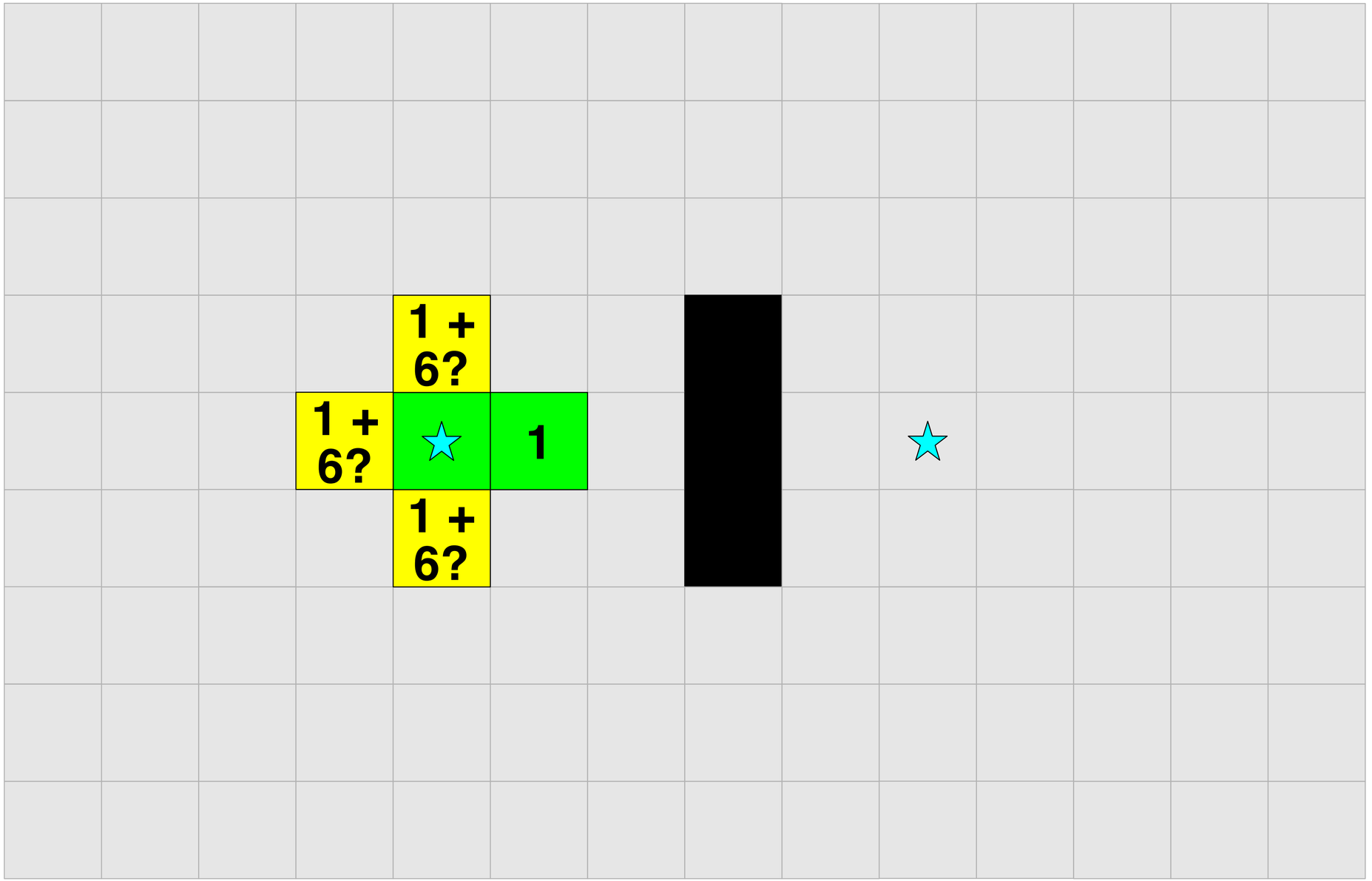
`h(start, goal)`

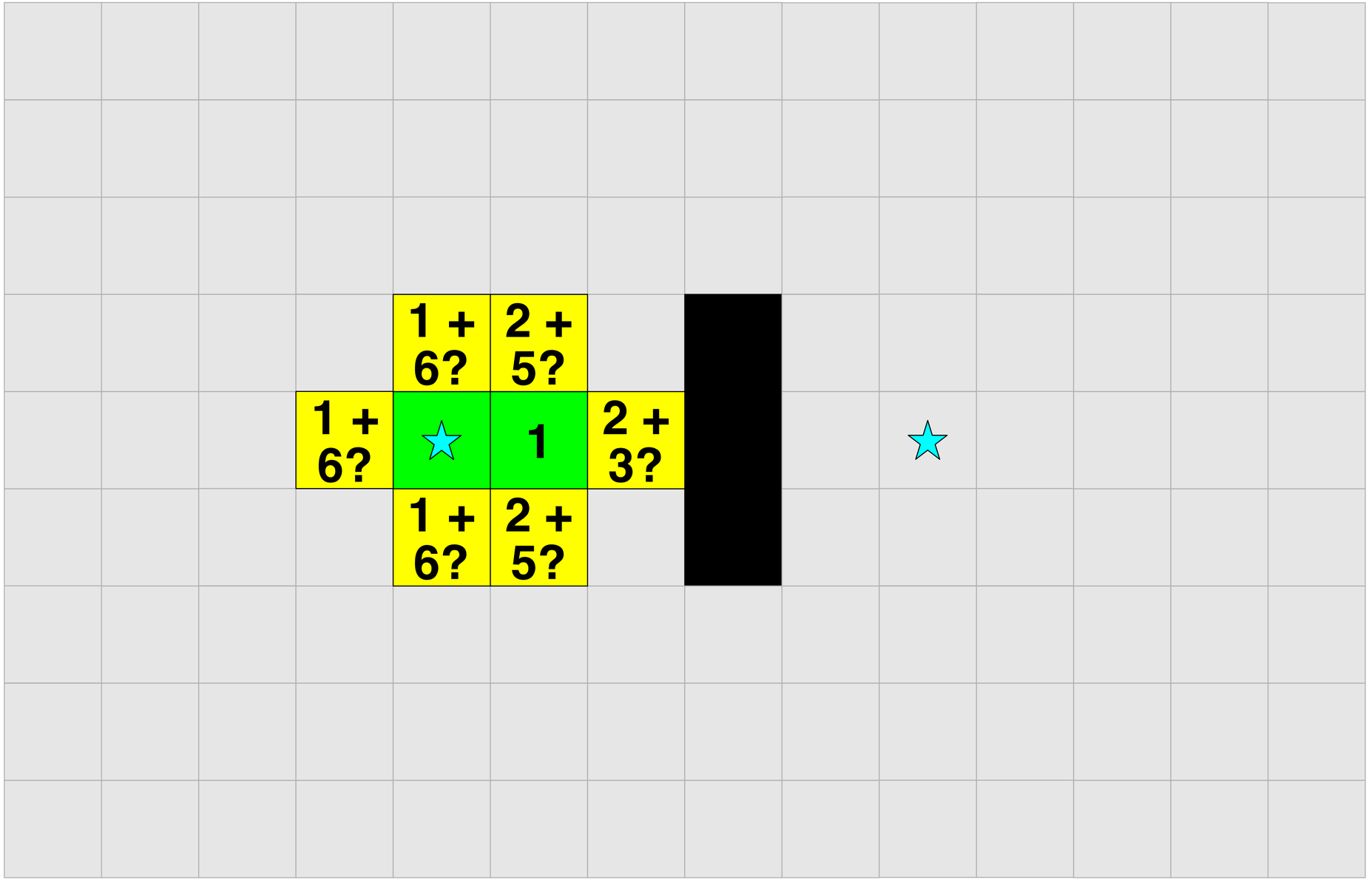


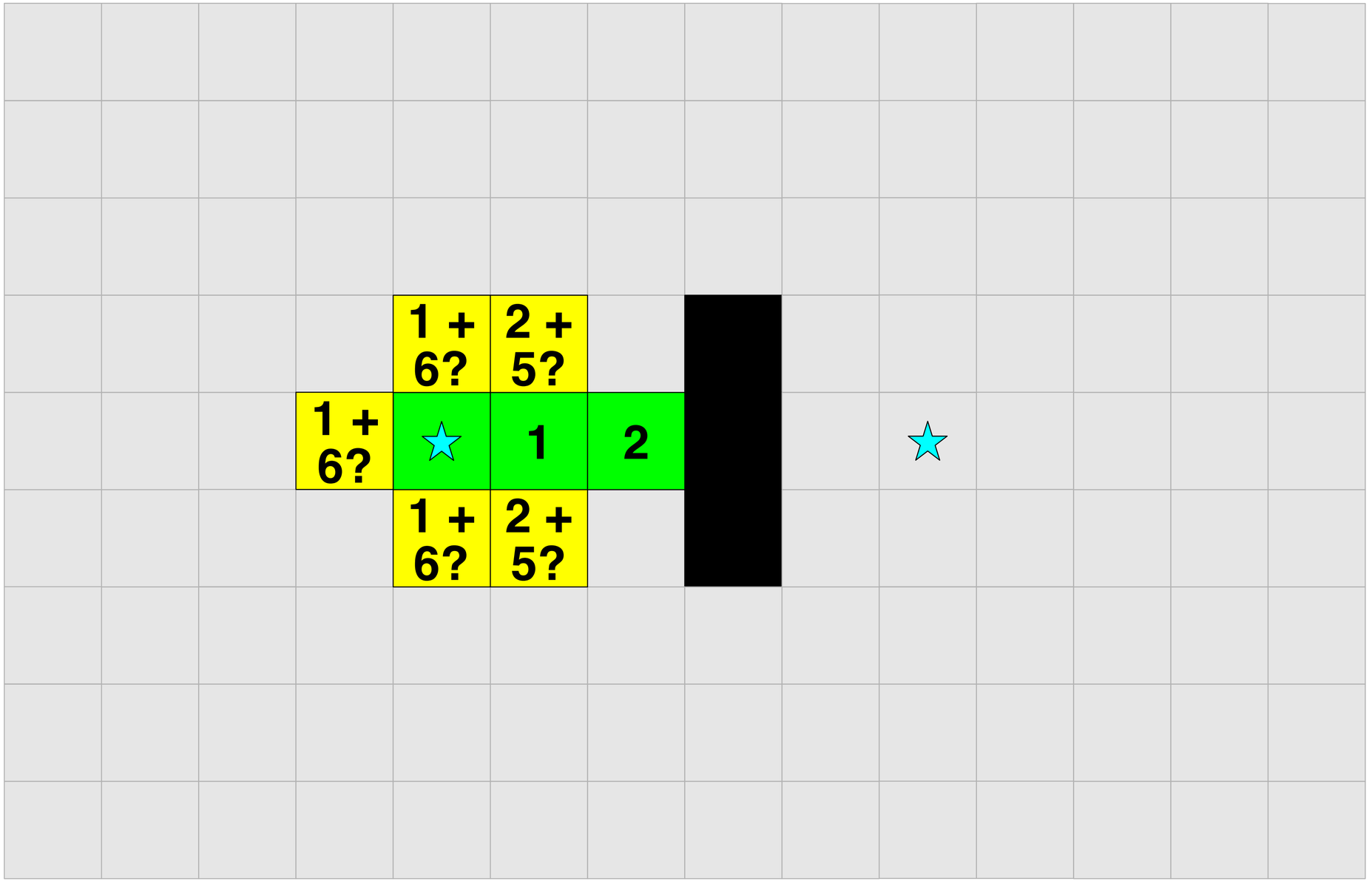


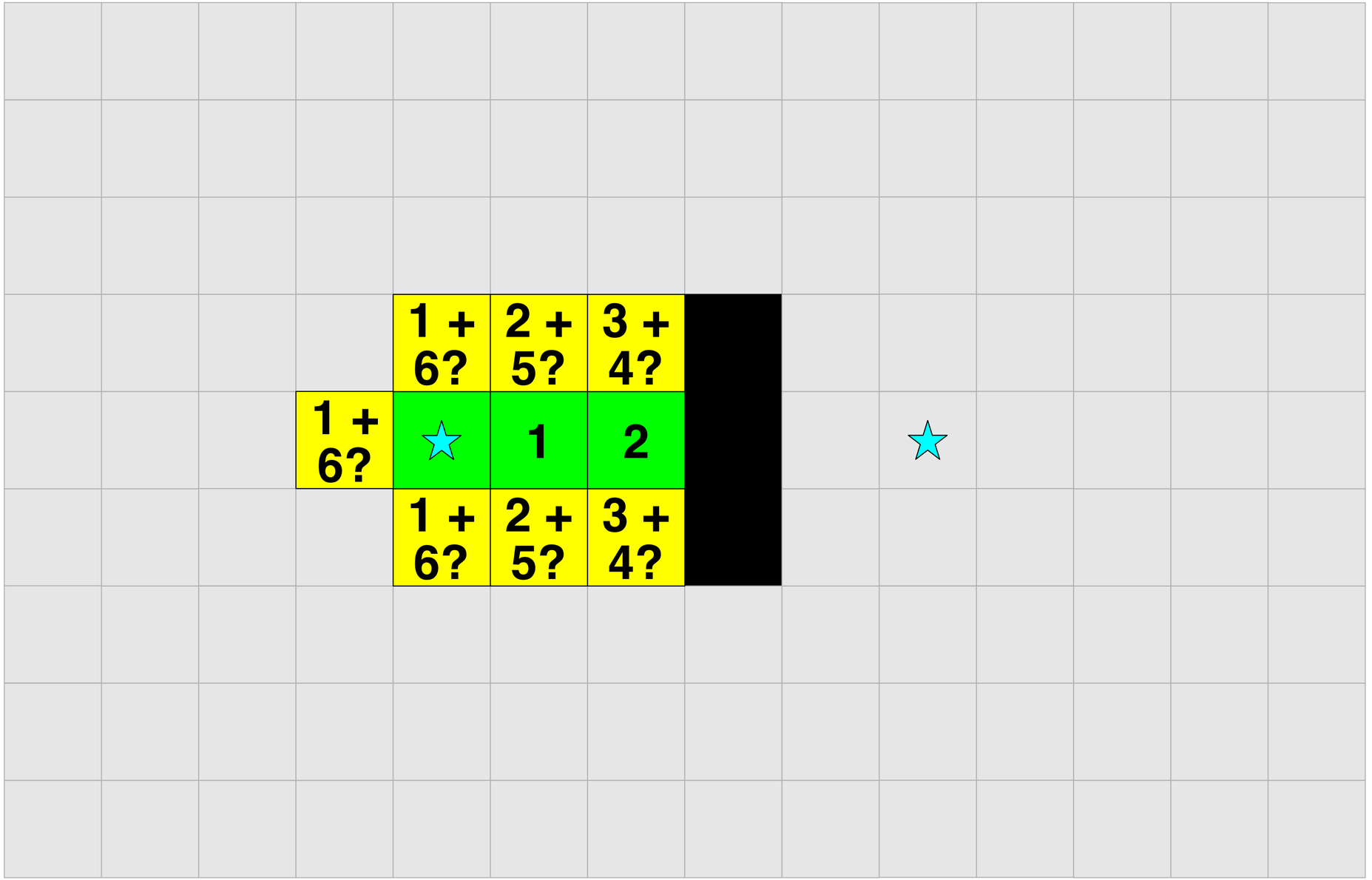


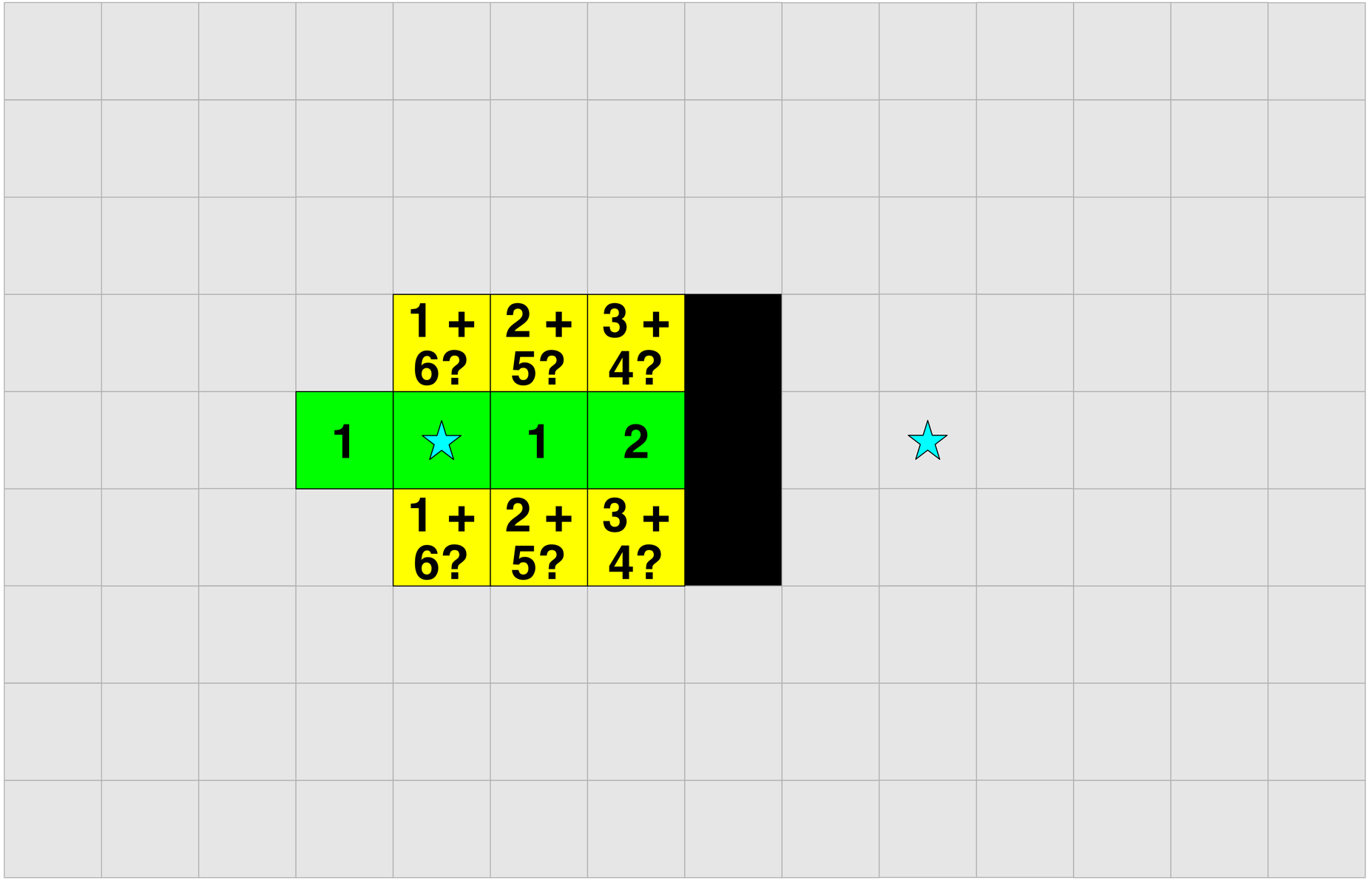

1 +
6?
1 +
6?
1 +
4?
1 +
6?

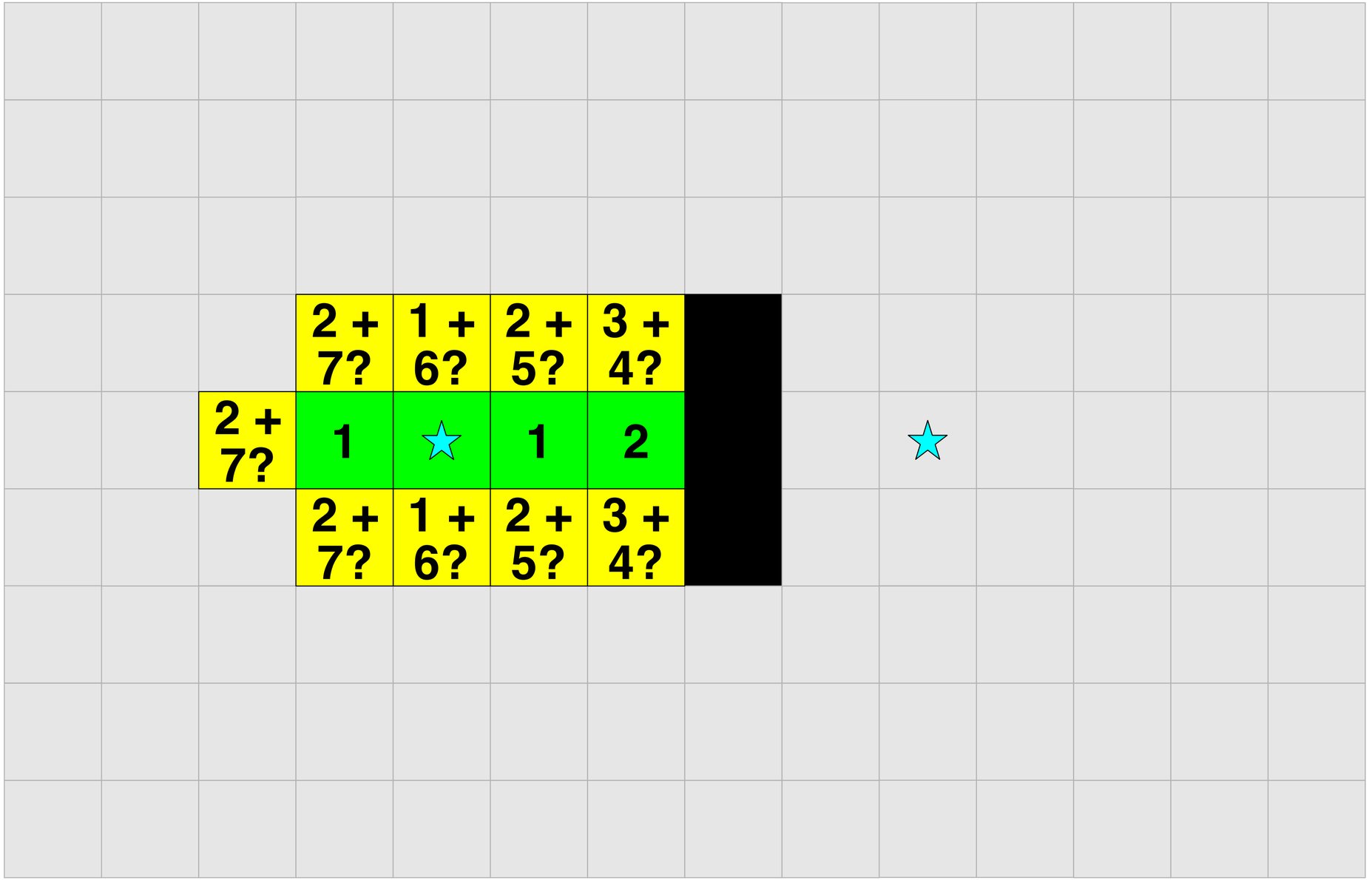

1 +
6?
1 +
6?
1
1 +
6?

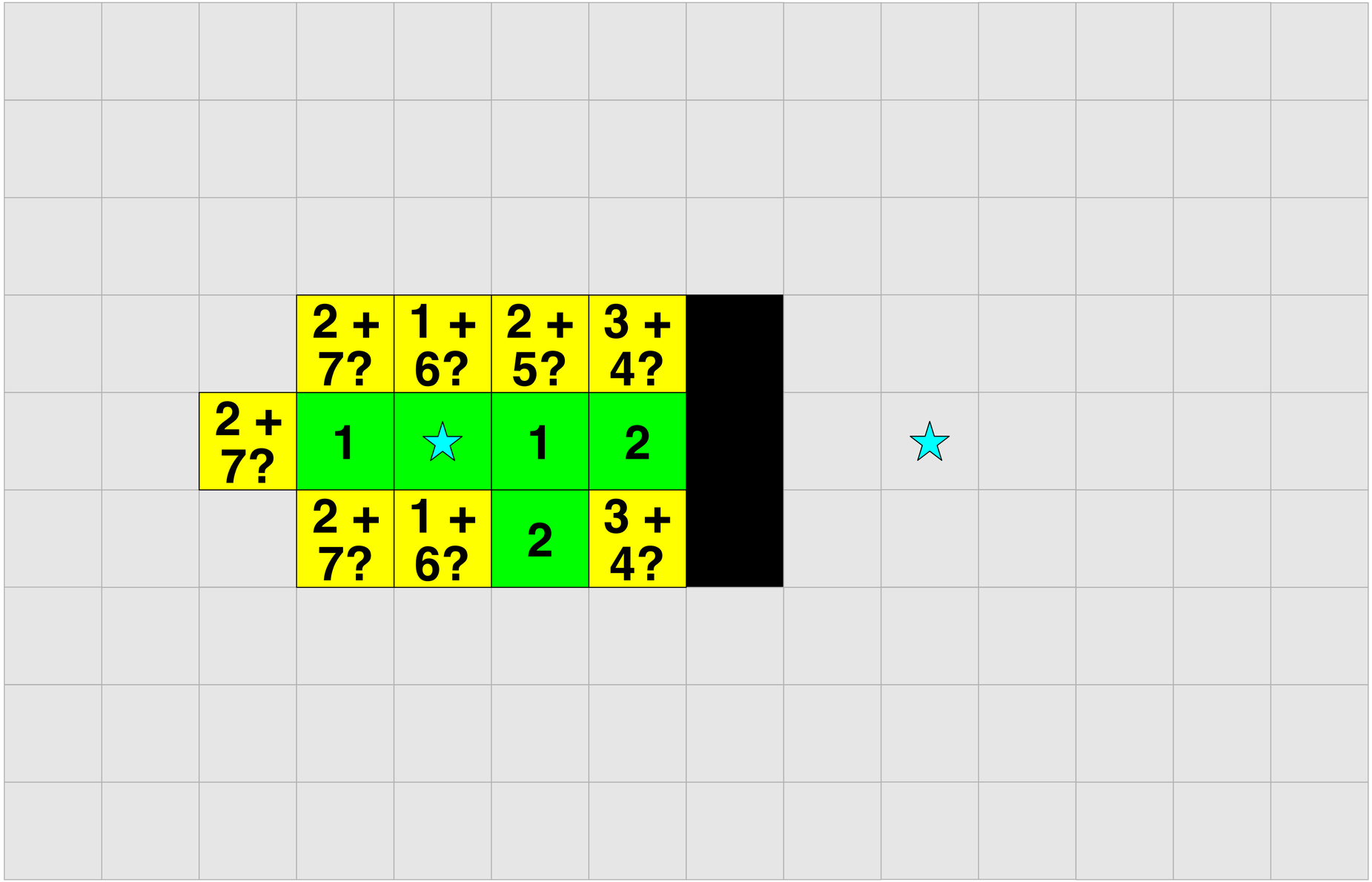

1 + 6?
2 + 5?
1 + 6?
1
2 + 3?
1 + 6?
2 + 5?

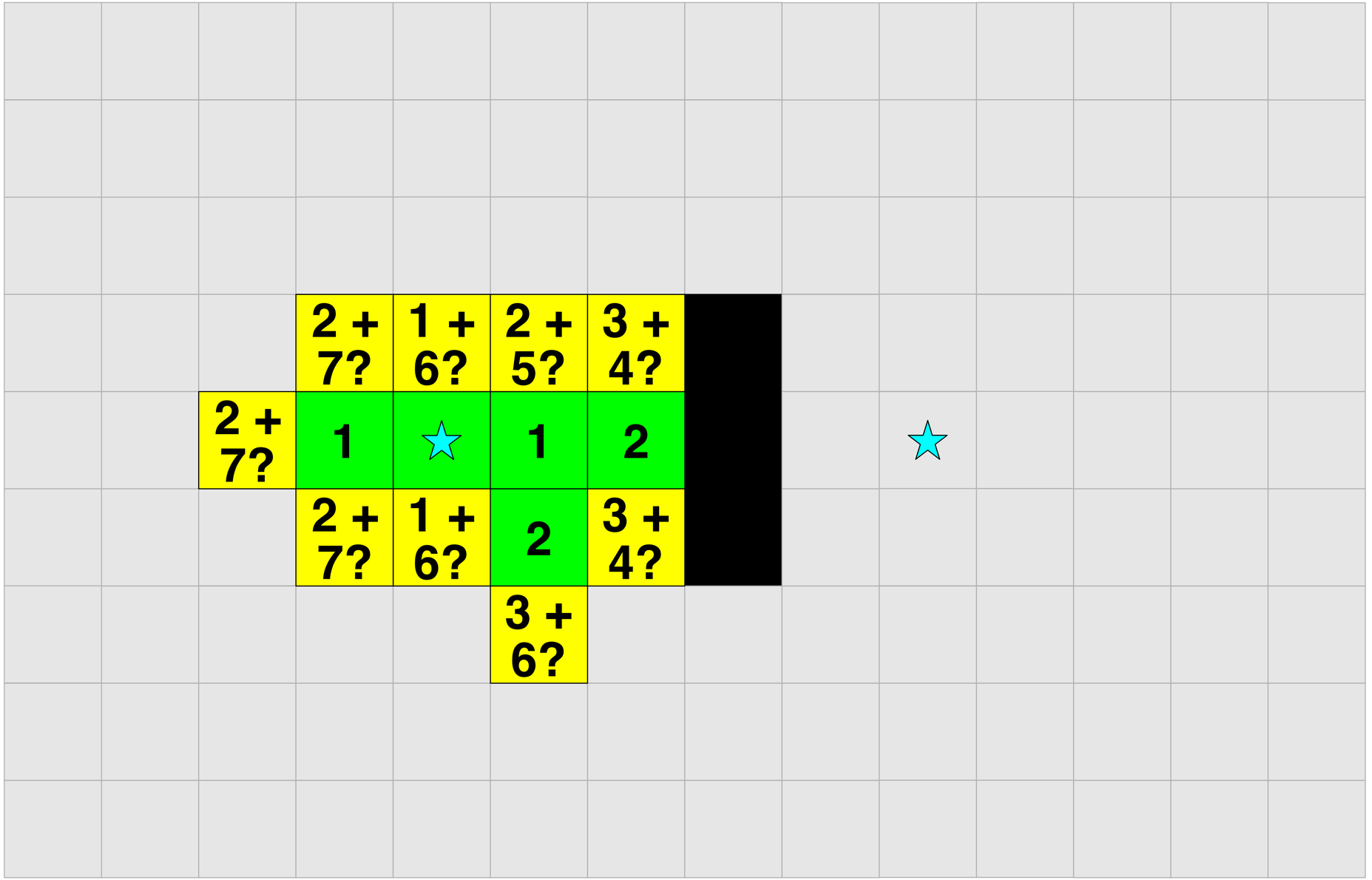

1 + 6?
2 + 5?
1 + 6?
1
2
1 + 6?
2 + 5?

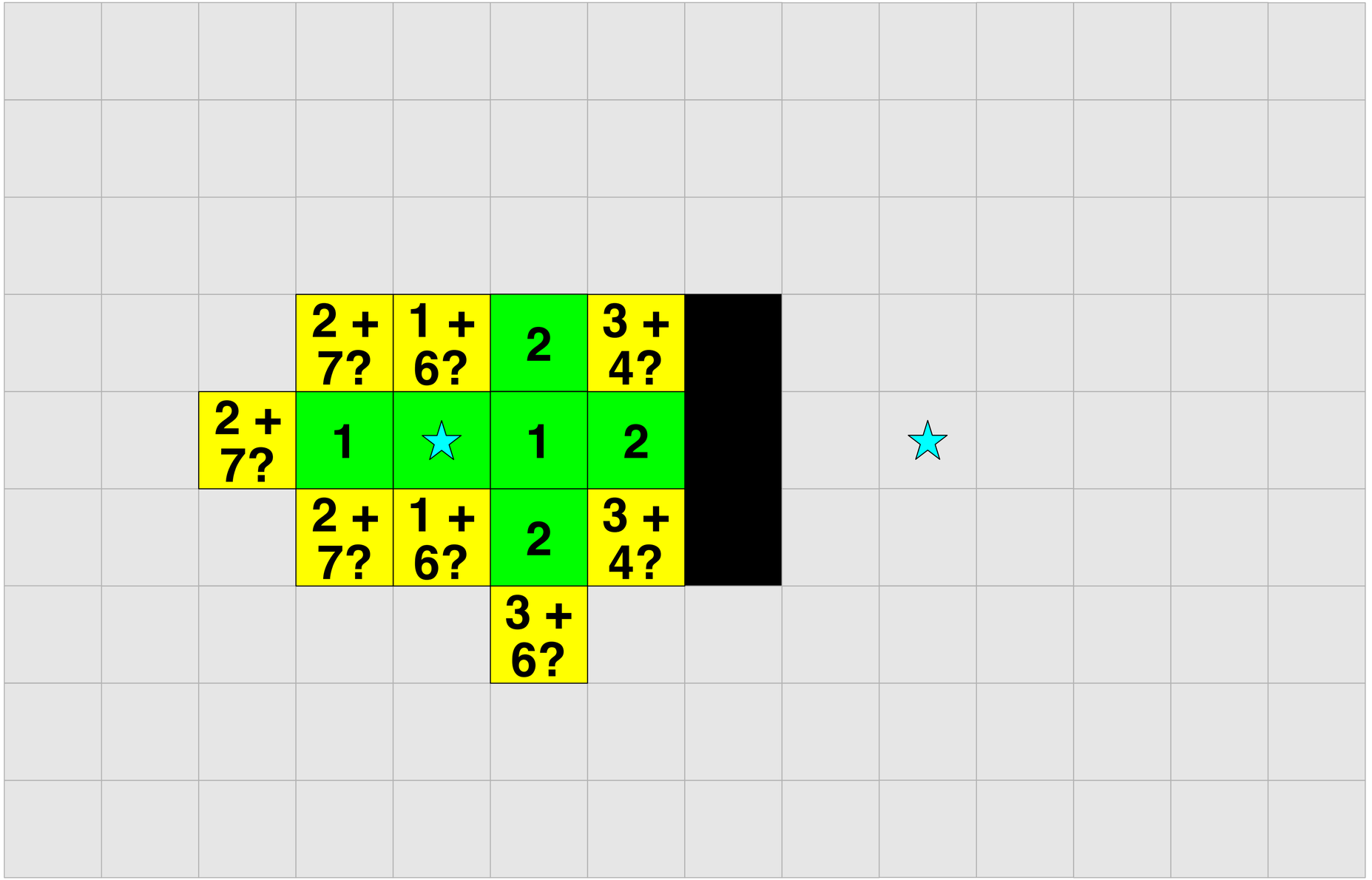

1 + 6?
2 + 5?
3 + 4?
1 + 6?
1
2
1 + 6?
2 + 5?
3 + 4?

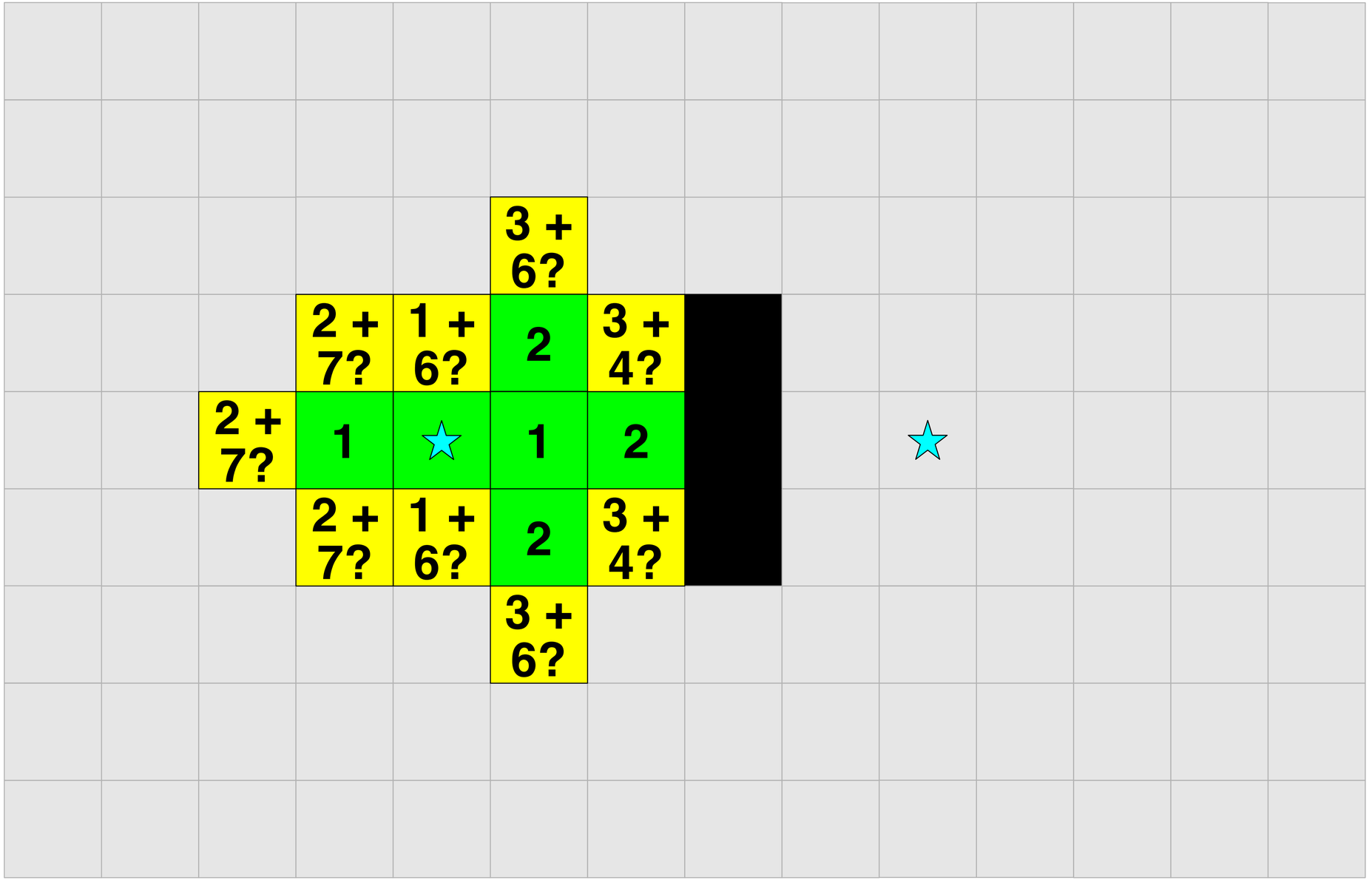

1 + 6?
2 + 5?
3 + 4?
1
1
2
1 + 6?
2 + 5?
3 + 4?

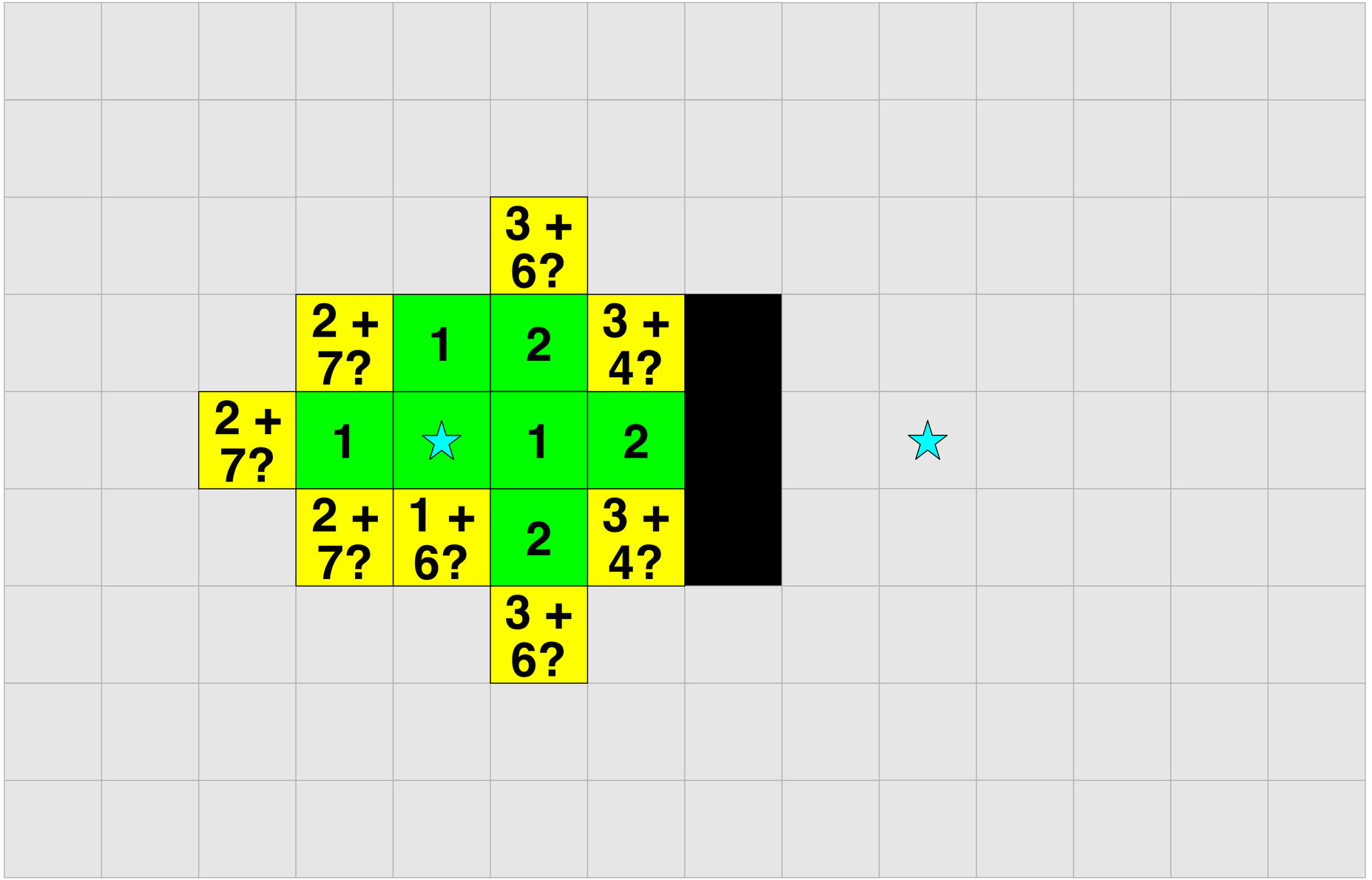

2 + 7?
1 + 6?
2 + 5?
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2 + 5?
3 + 4?

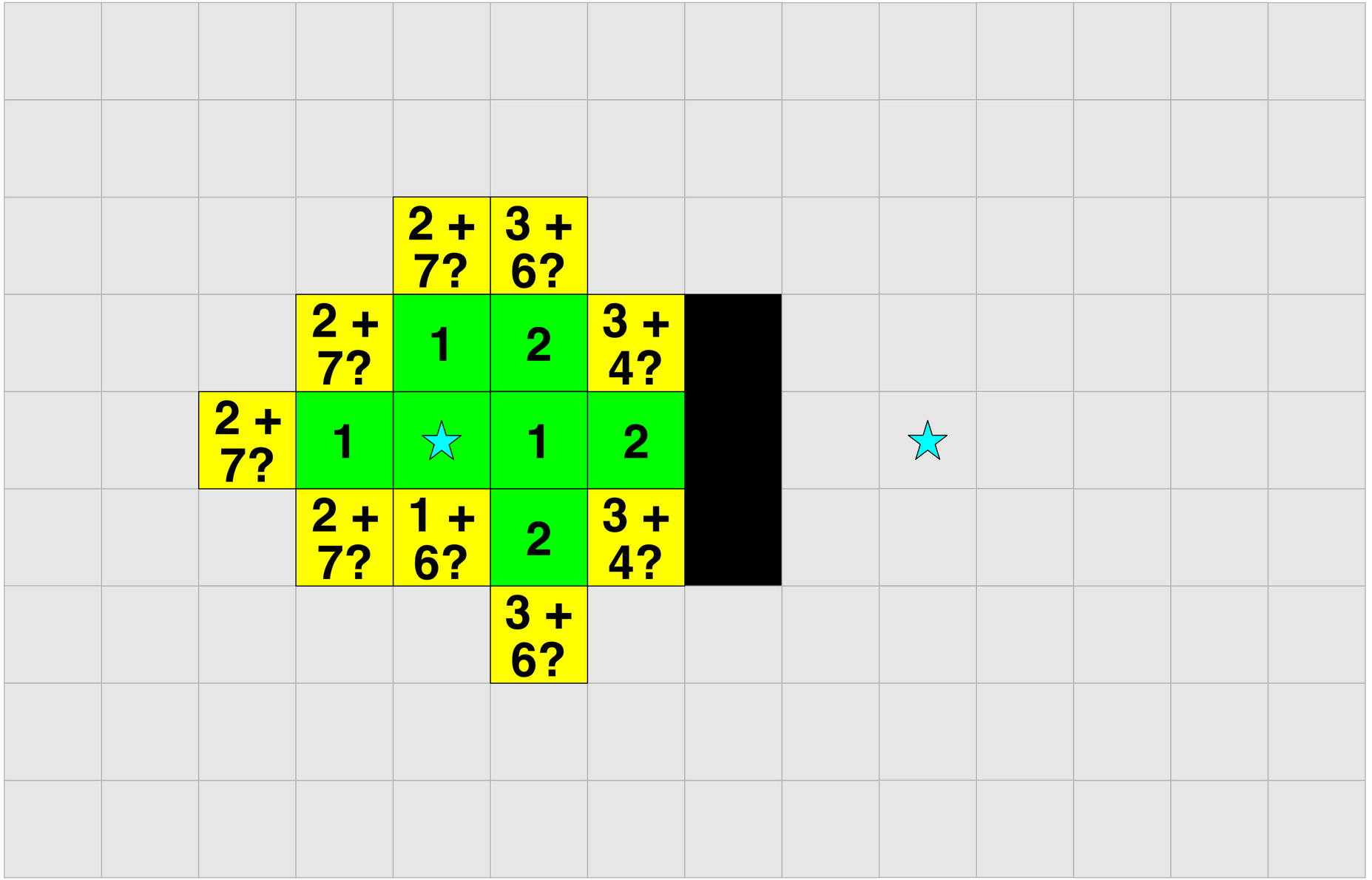

2 + 7?
1 + 6?
2 + 5?
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?

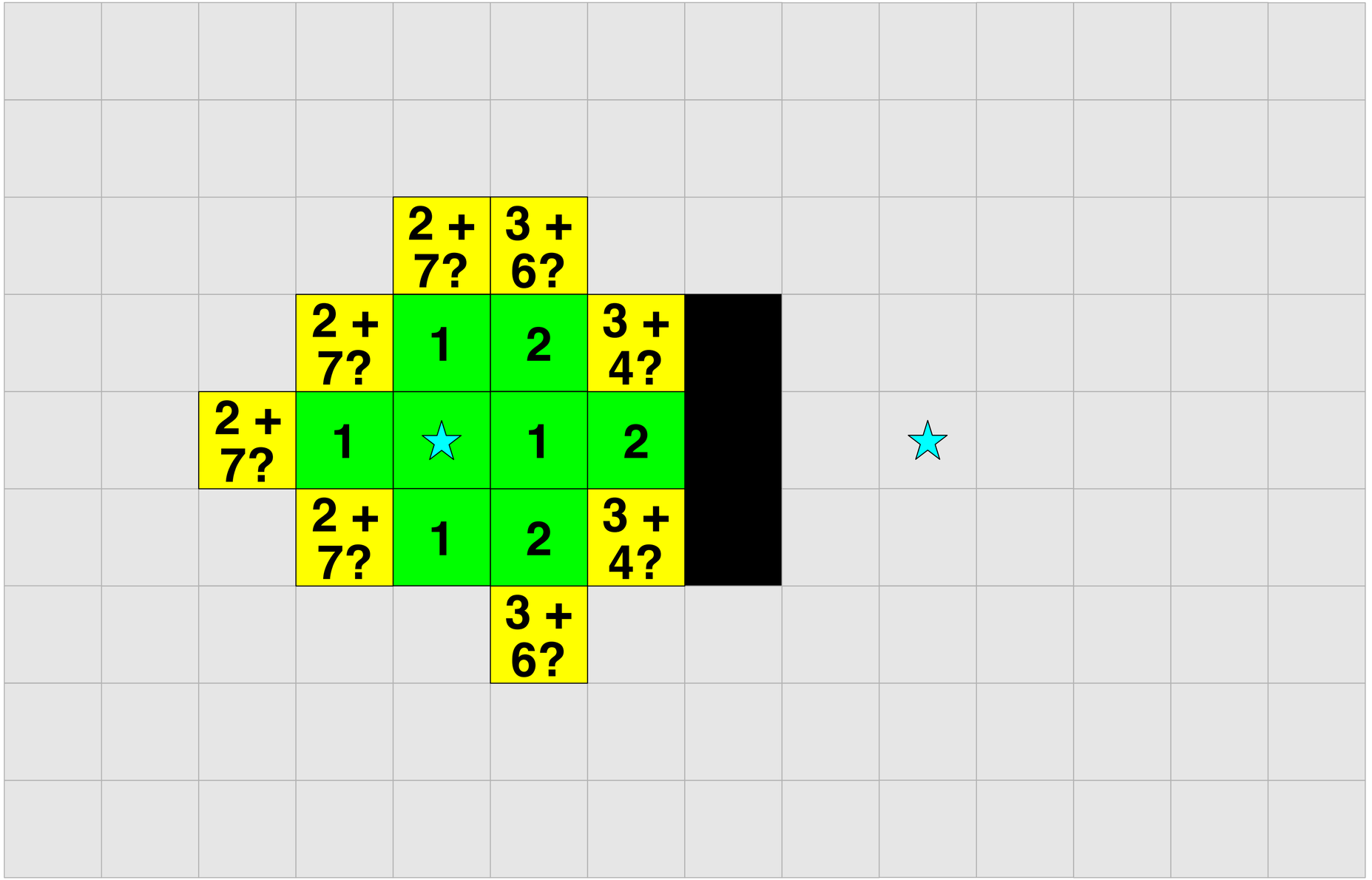

2 + 7?
1 + 6?
2 + 5?
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?
3 + 6?

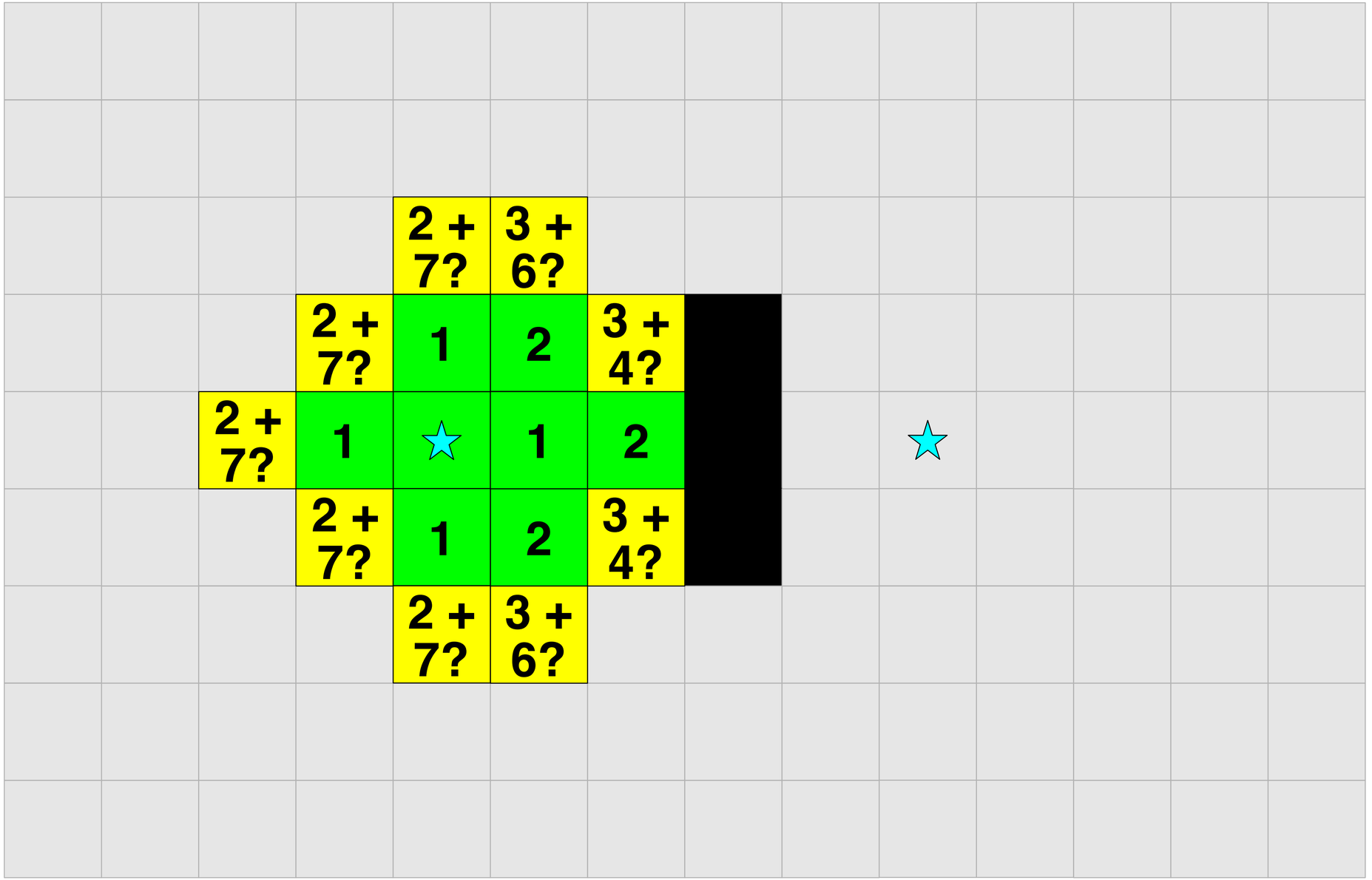

2 + 7?
1 + 6?
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?
3 + 6?

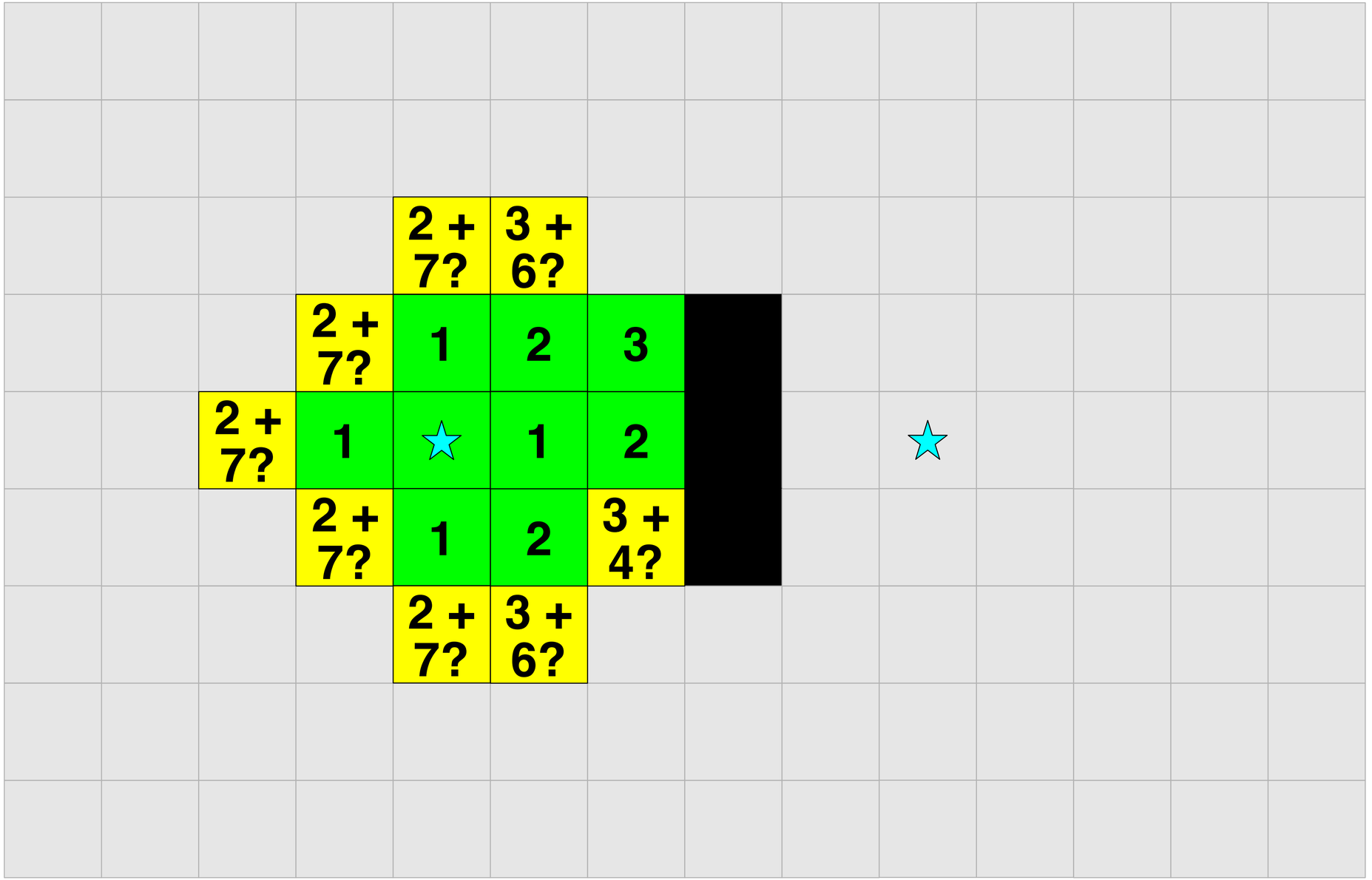

3 + 6?
2 + 7?
1 + 6?
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?
3 + 6?

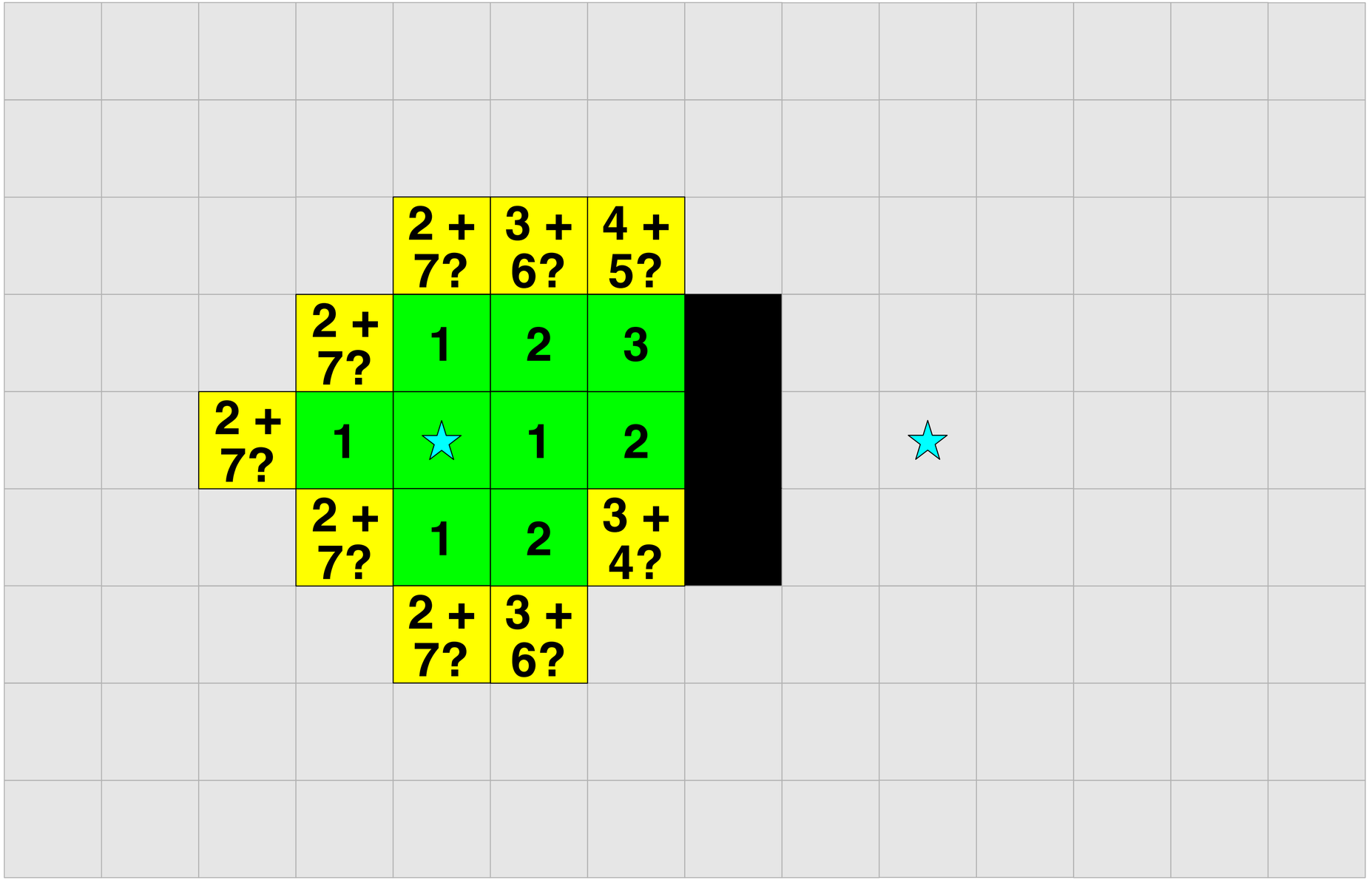

3 + 6?
2 + 7?
1
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?
3 + 6?

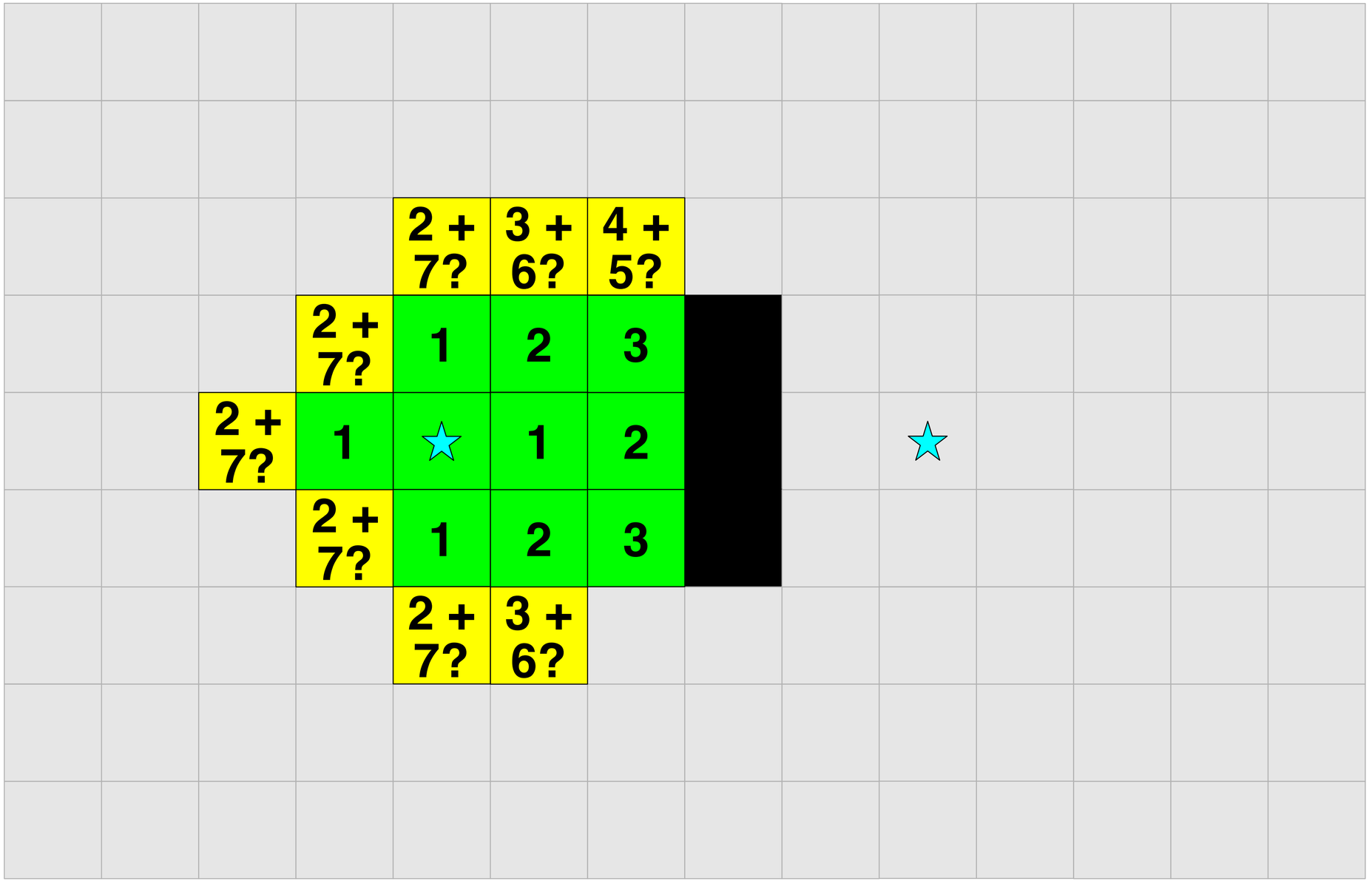

2 + 7?
3 + 6?
2 + 7?
1
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1 + 6?
2
3 + 4?
3 + 6?

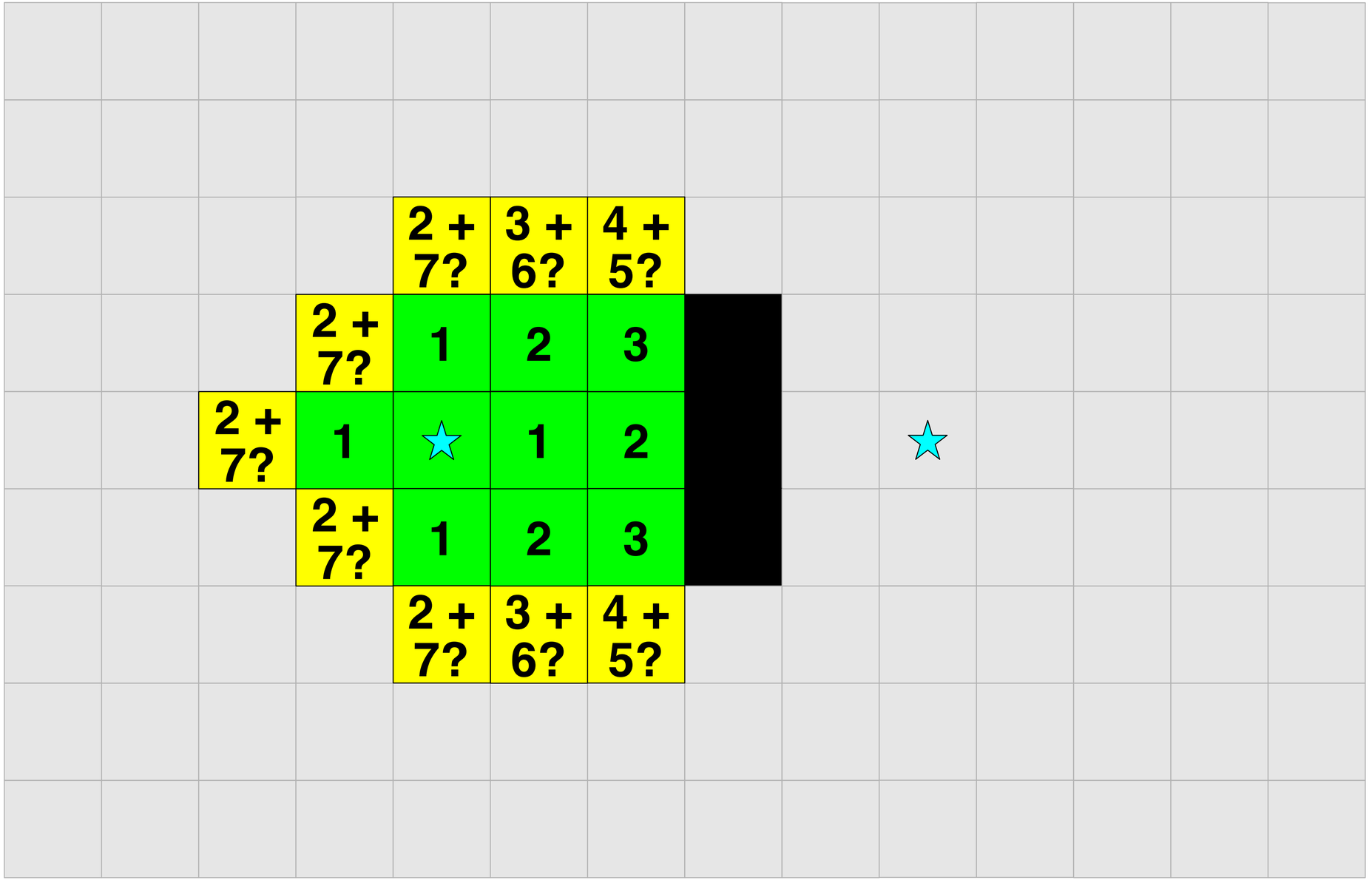

2 + 7?
3 + 6?
2 + 7?
1
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1
2
3 + 4?
3 + 6?

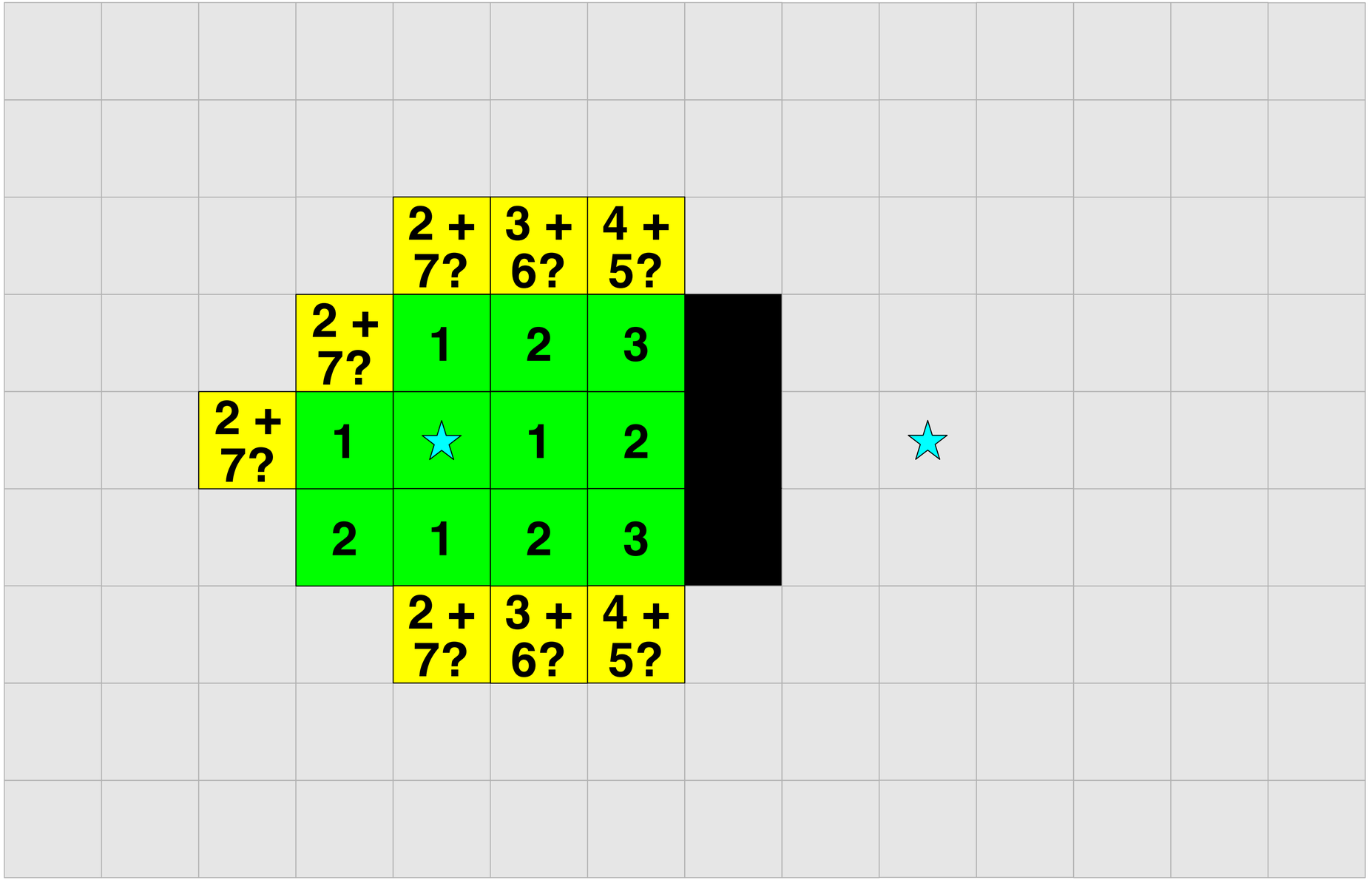

2 + 7?
3 + 6?
2 + 7?
1
2
3 + 4?
2 + 7?
1
1
2
2 + 7?
1
2
3 + 4?
2 + 7?
3 + 6?

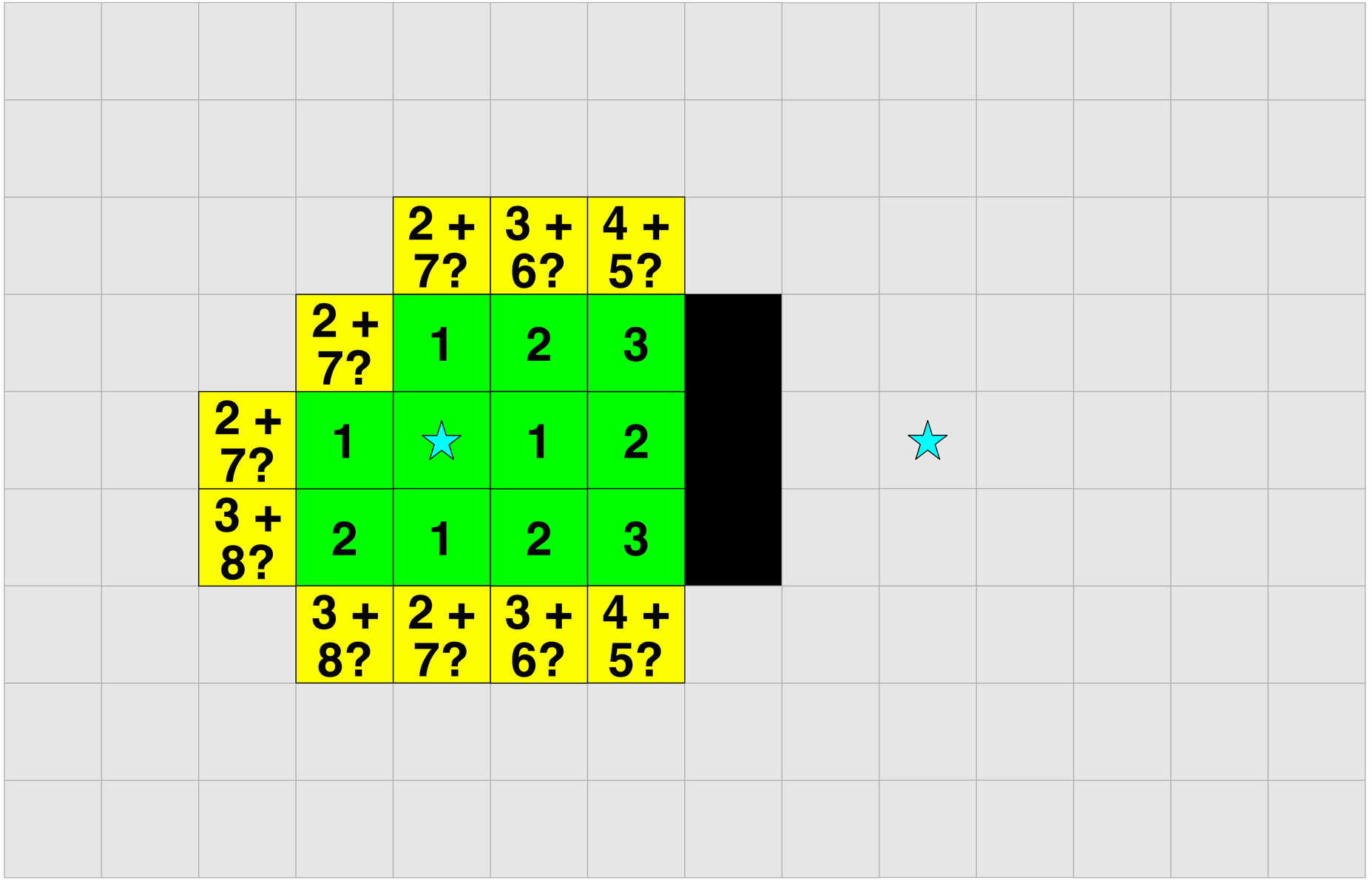

2 + 7?
3 + 6?
2 + 7?
1
2
3
2 + 7?
1
1
2
2 + 7?
1
2
3 + 4?
2 + 7?
3 + 6?

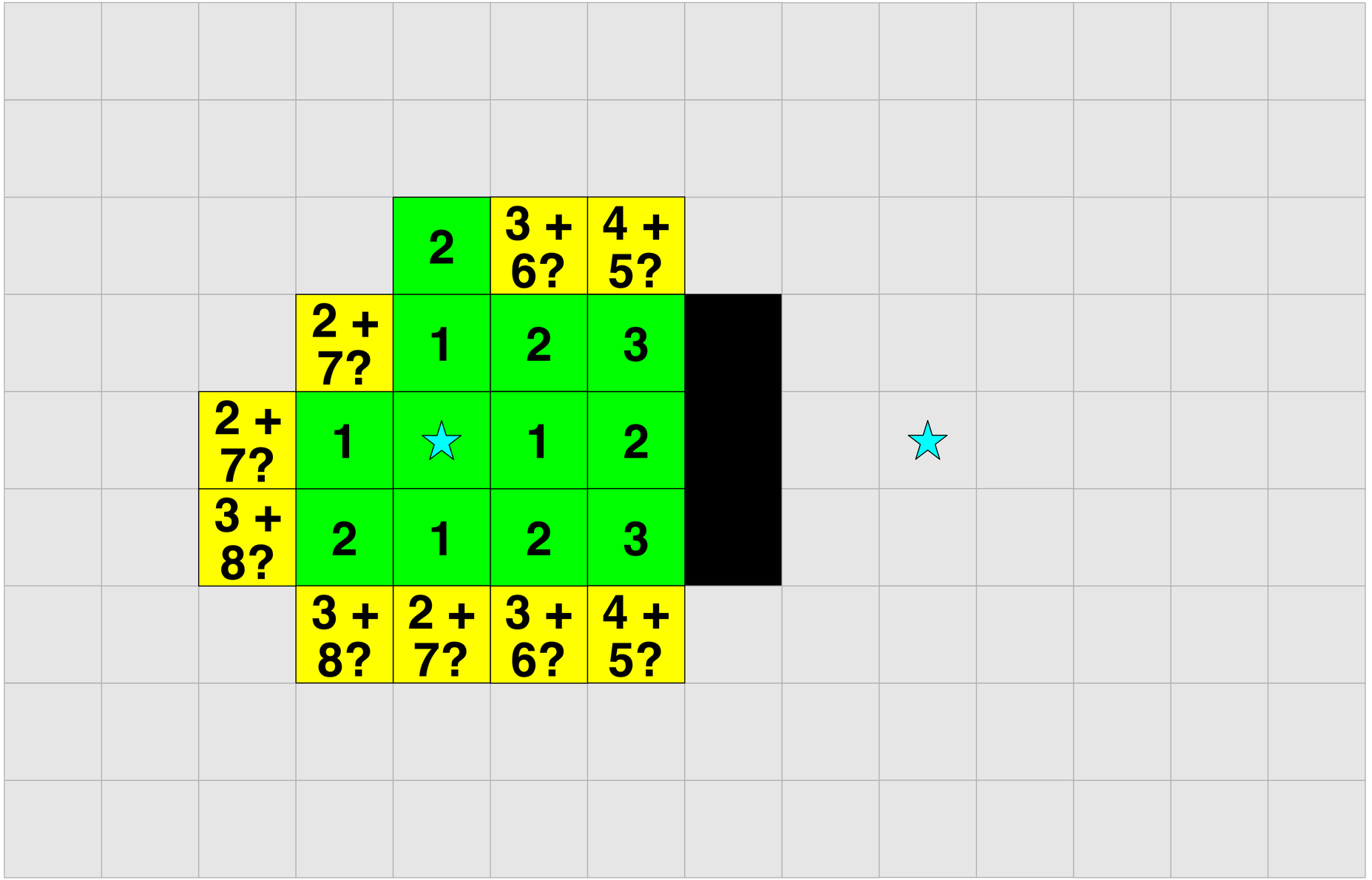

2 + 7?
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
2 + 7?
1
2
3 + 4?
2 + 7?
3 + 6?

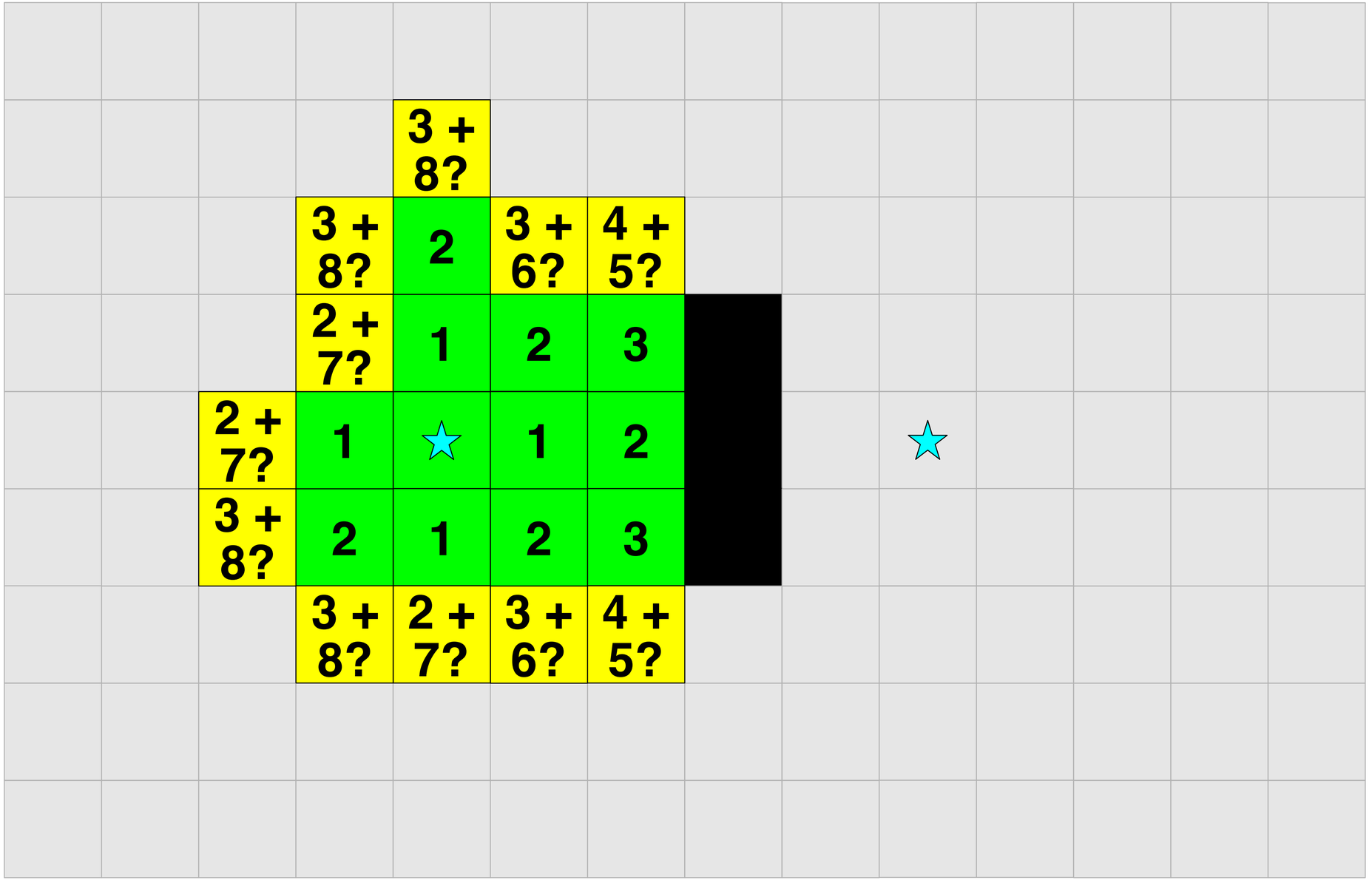

2 + 7?
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
2 + 7?
1
2
3
2 + 7?
3 + 6?

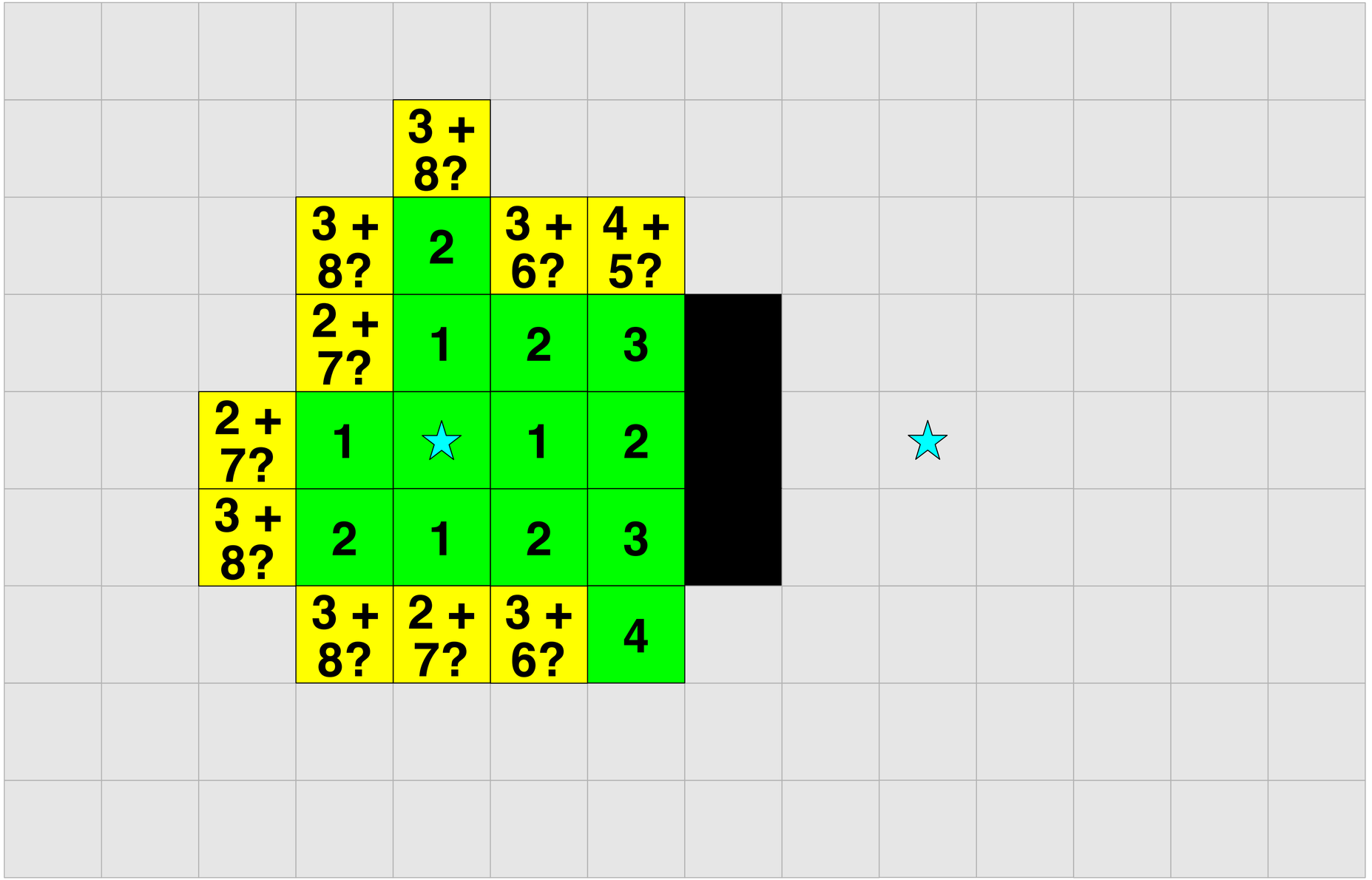

2 + 7?
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
2 + 7?
1
2
3
2 + 7?
3 + 6?
4 + 5?

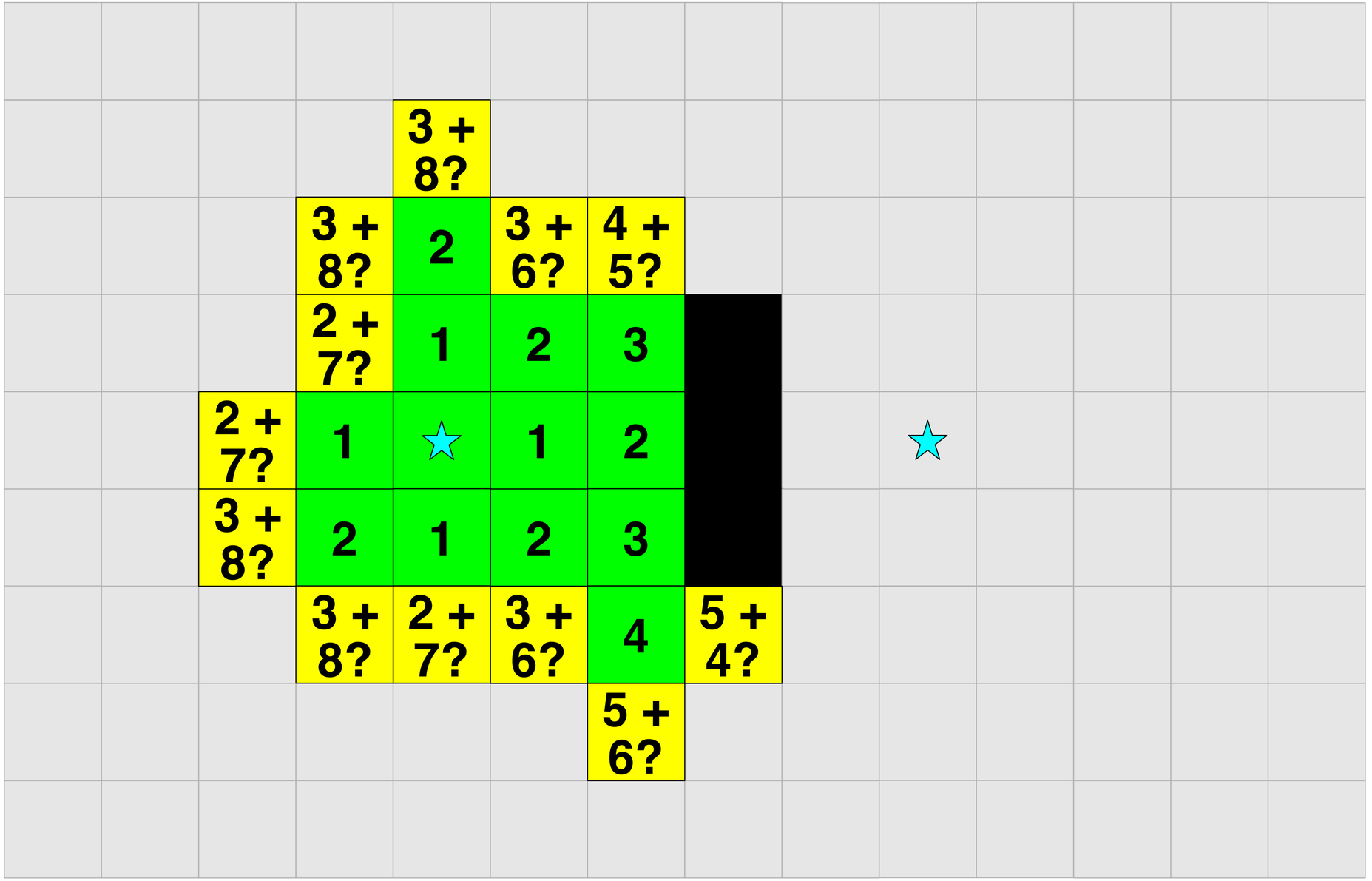

2 + 7?
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
2
1
2
3
2 + 7?
3 + 6?
4 + 5?

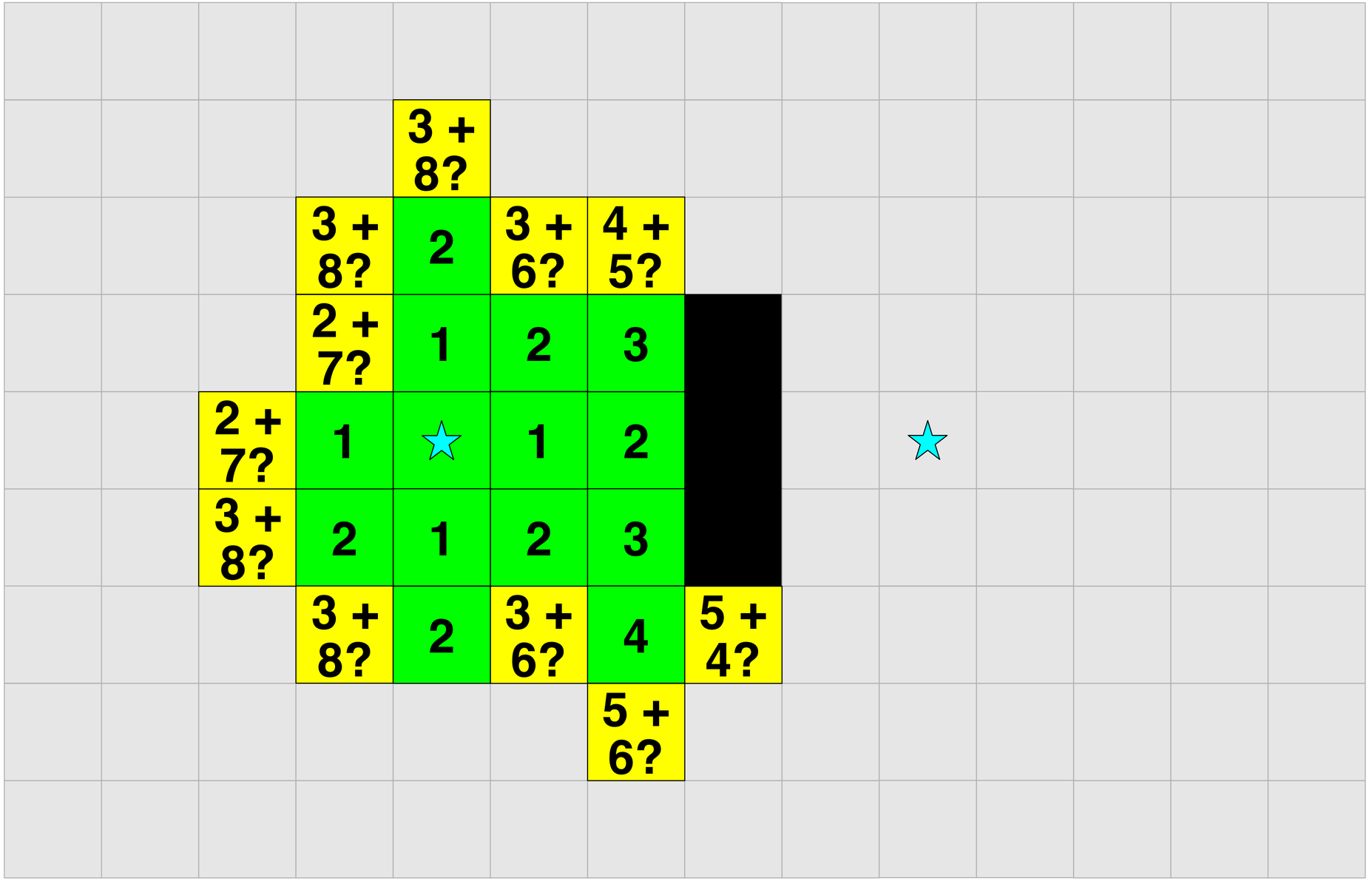

2 + 7?
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2 + 7?
3 + 6?
4 + 5?

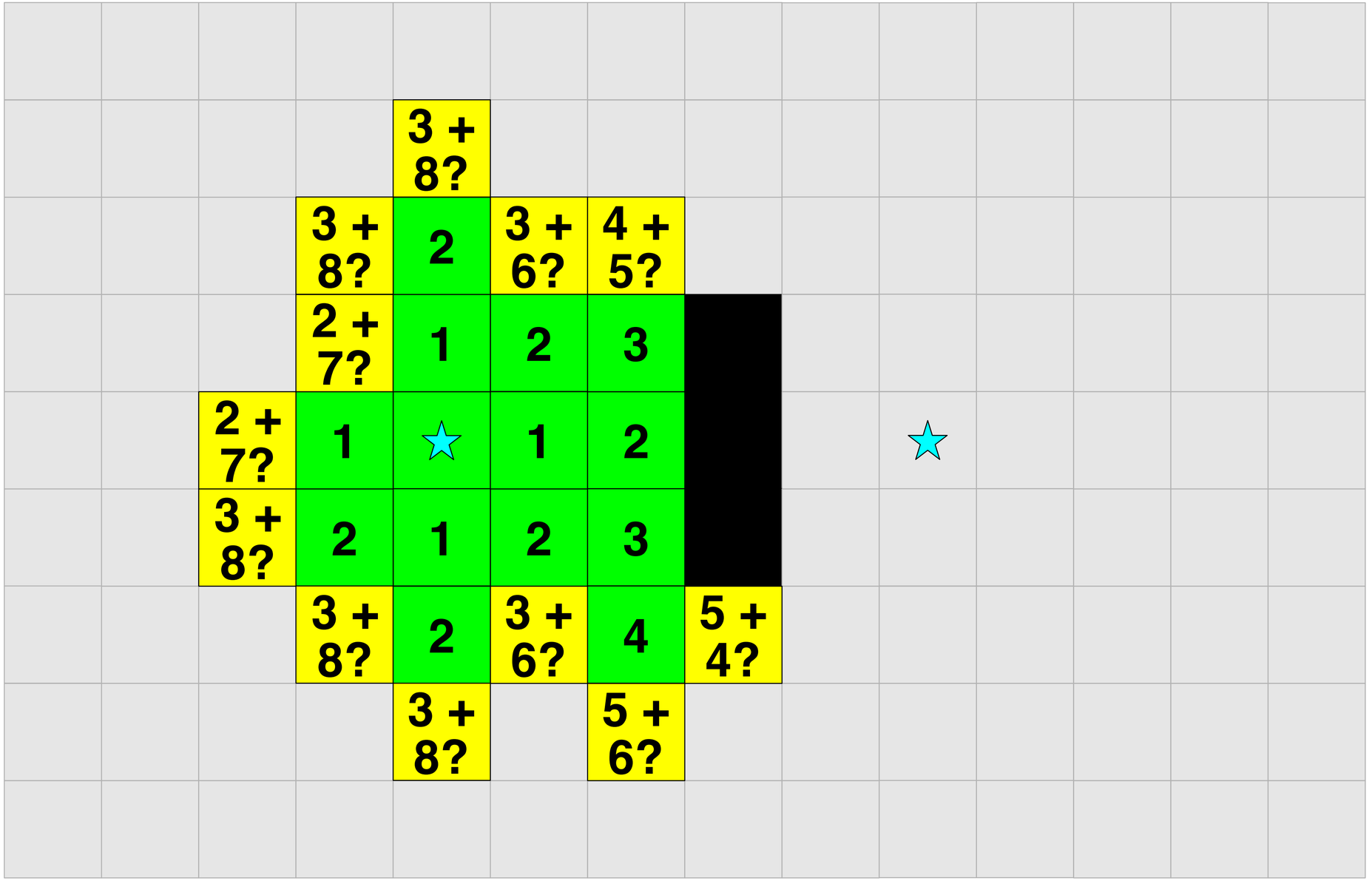

2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2 + 7?
3 + 6?
4 + 5?

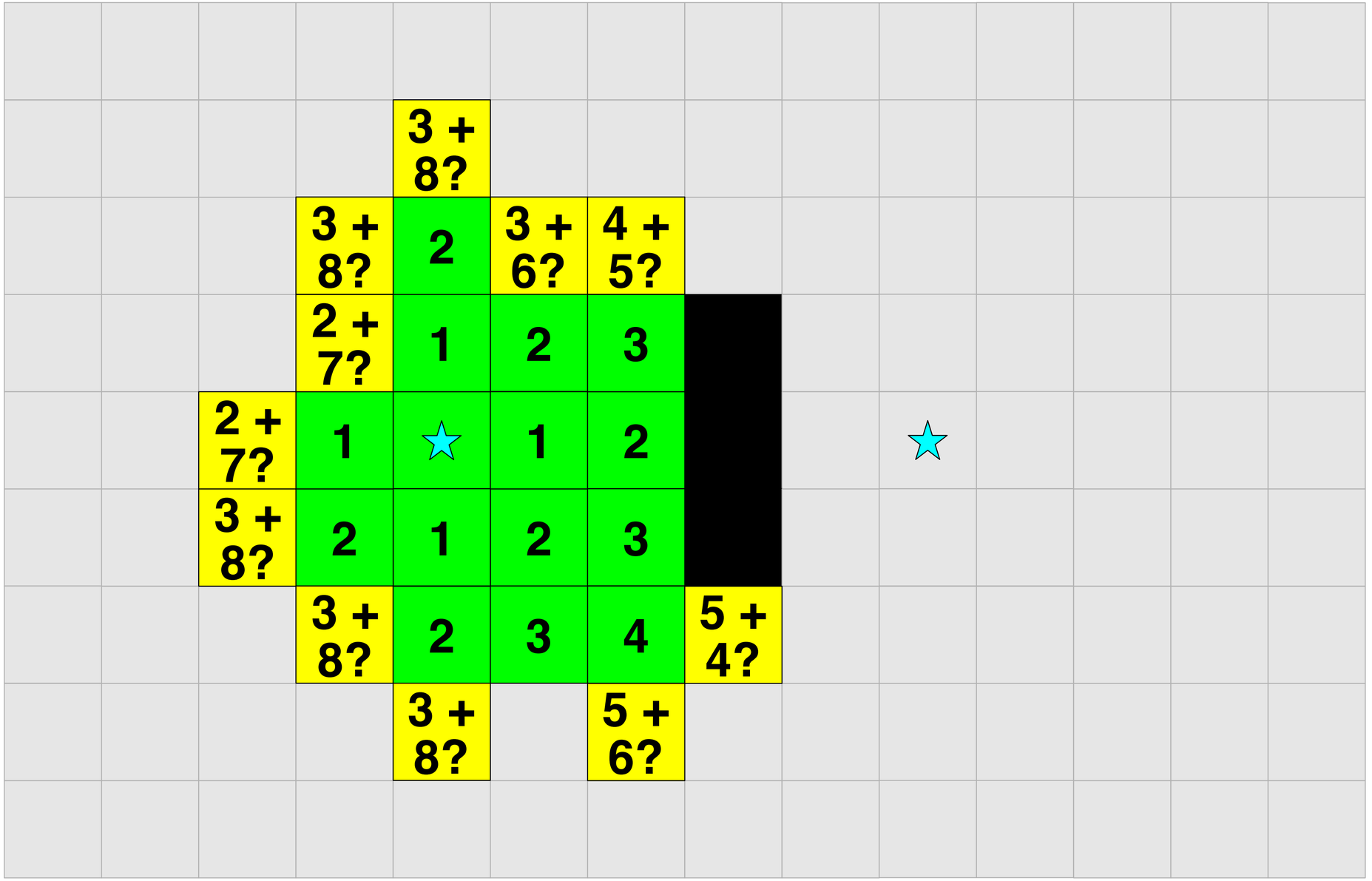

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2 + 7?
3 + 6?
4 + 5?

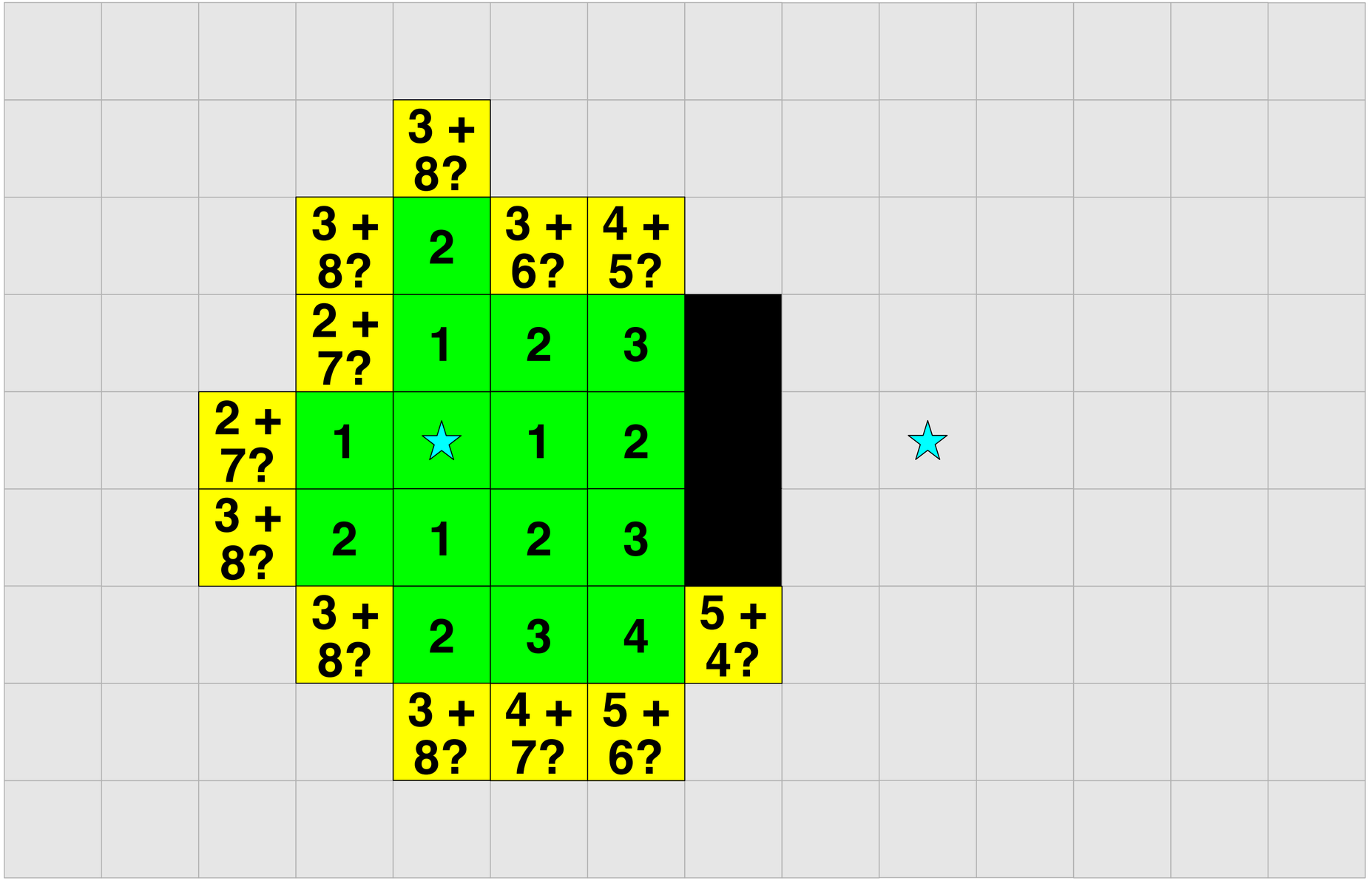

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2 + 7?
3 + 6?
4

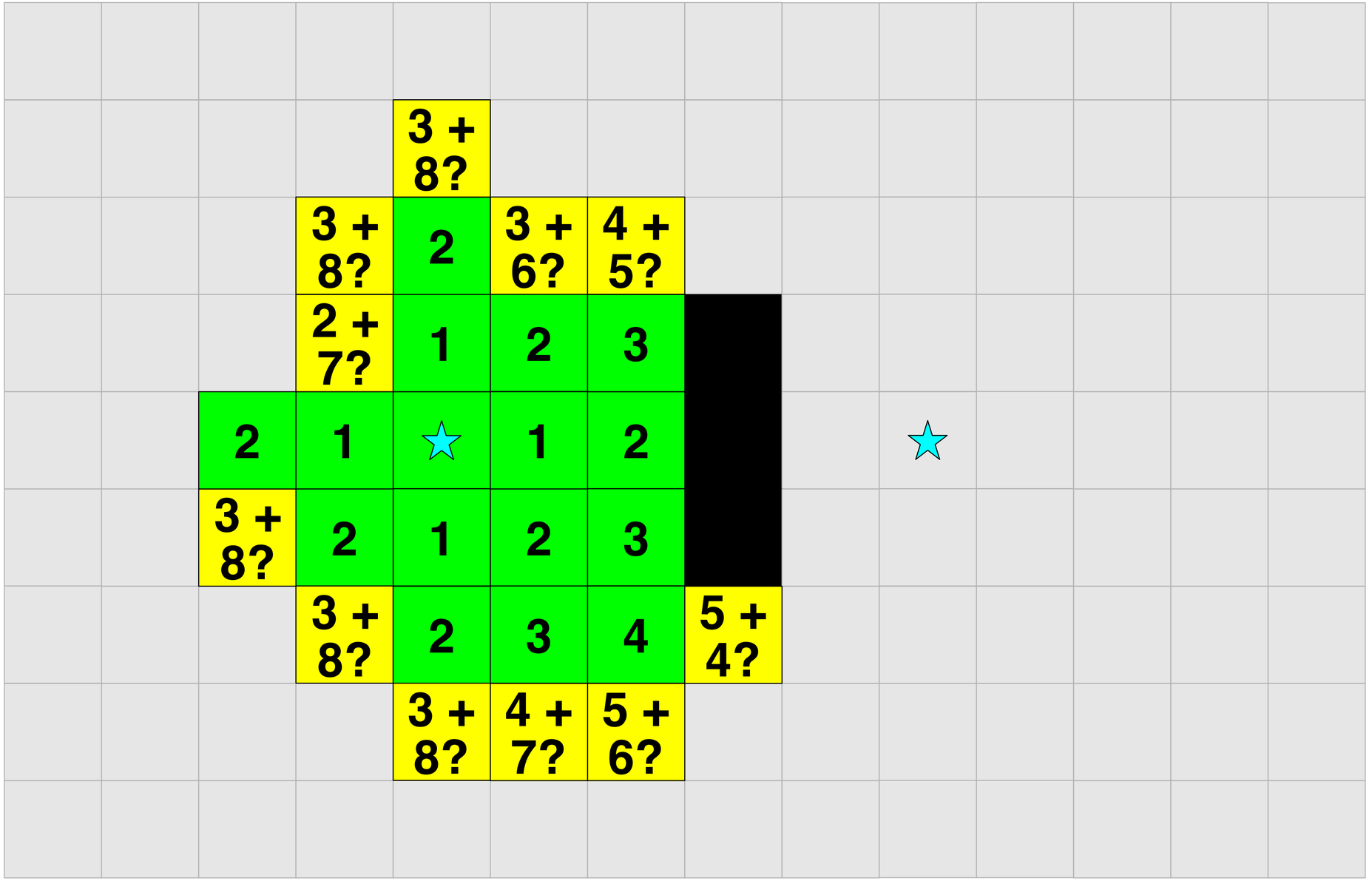

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2 + 7?
3 + 6?
4
5 + 4?
5 + 6?

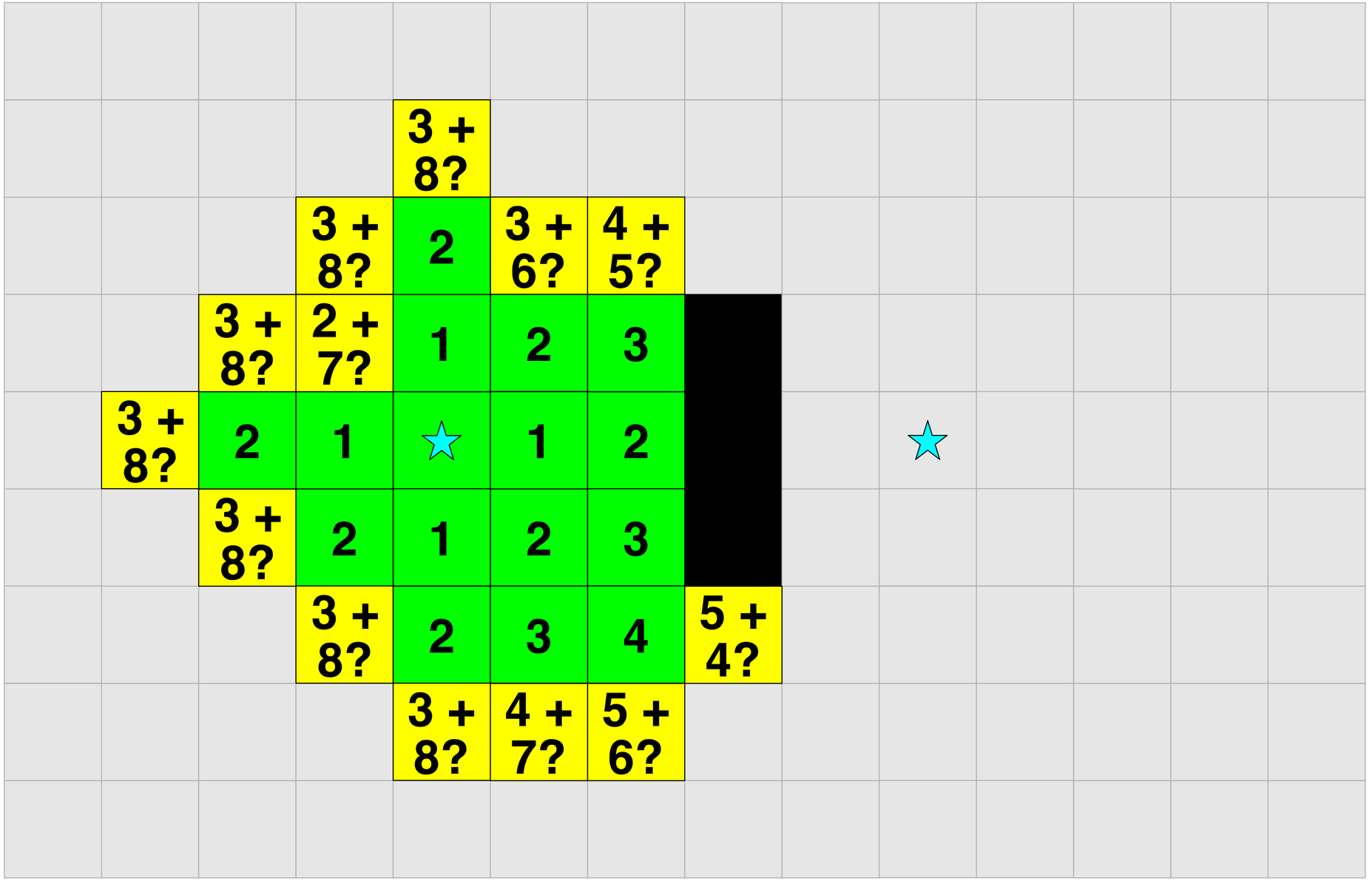

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3 + 6?
4
5 + 4?
5 + 6?

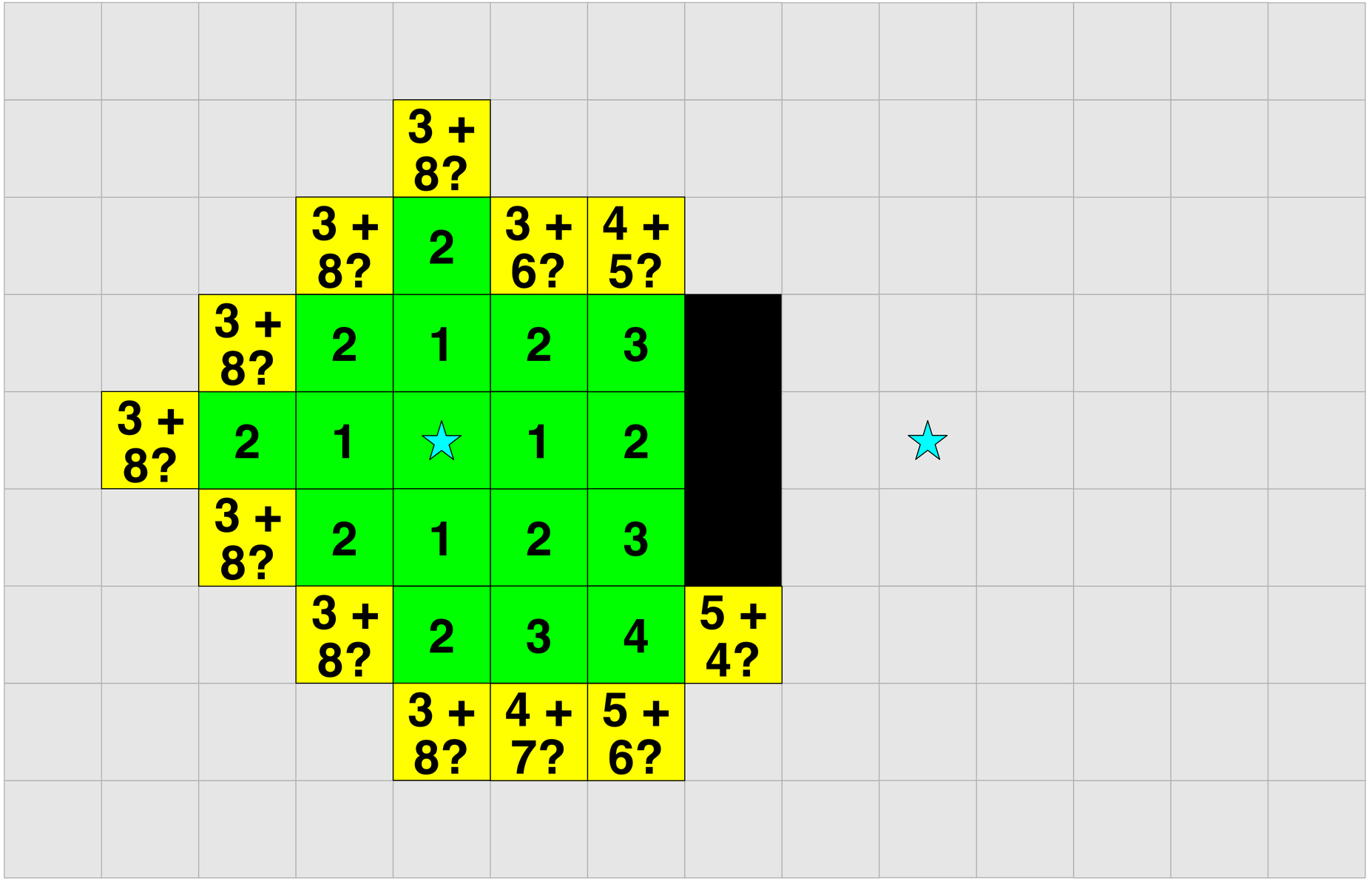

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3 + 6?
4
5 + 4?
3 + 8?
5 + 6?

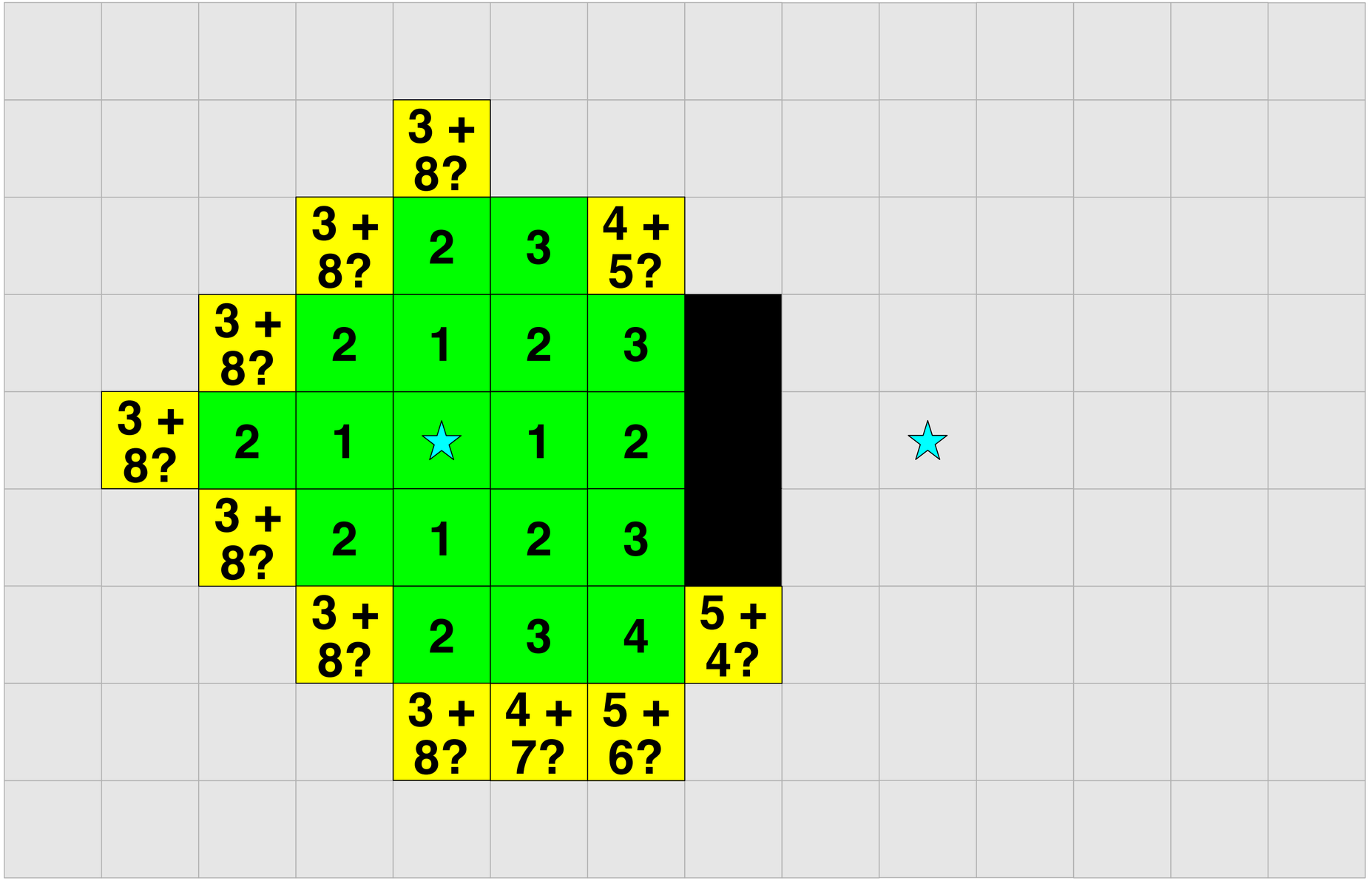

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
5 + 6?

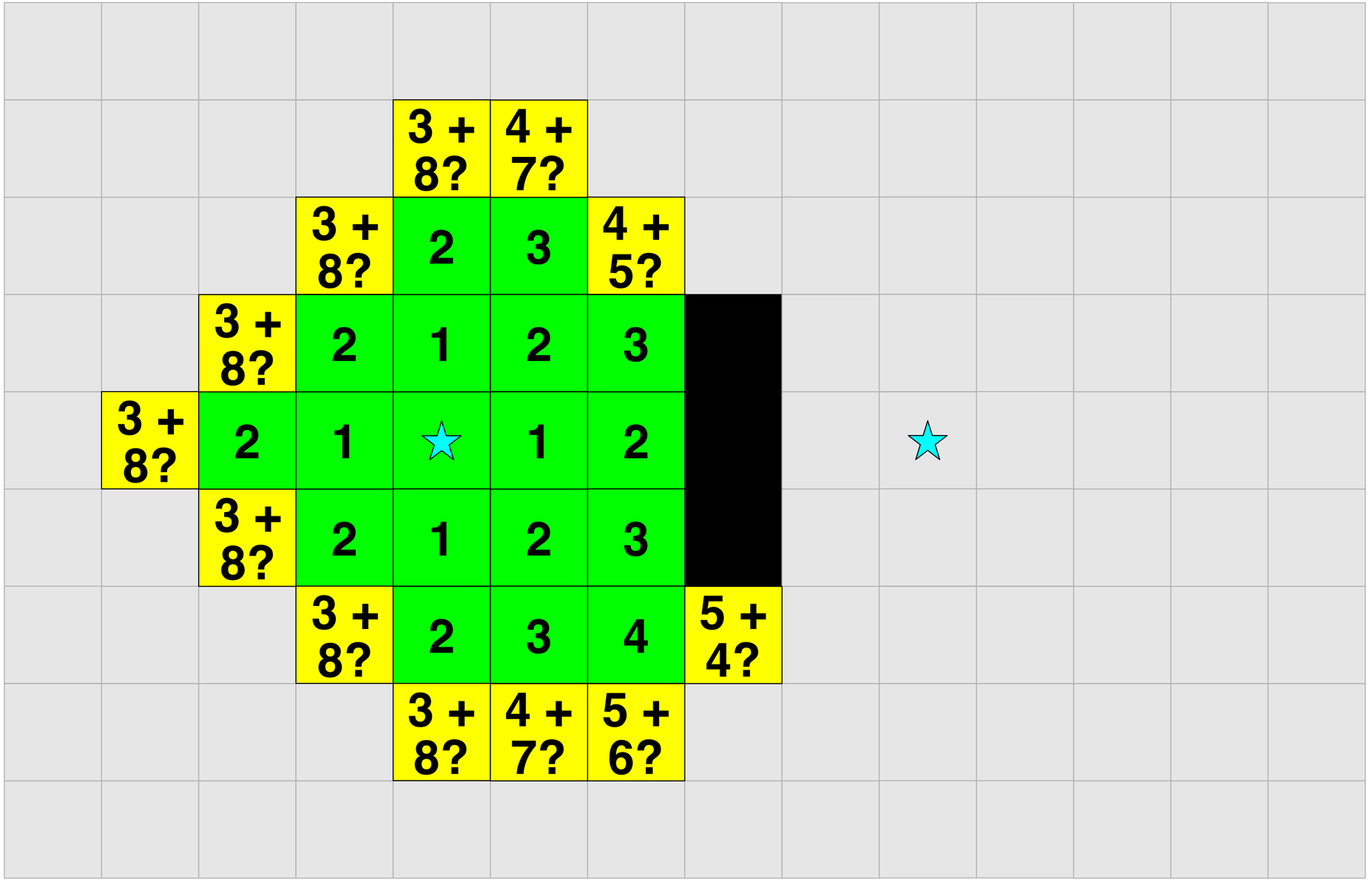

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2 + 7?
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

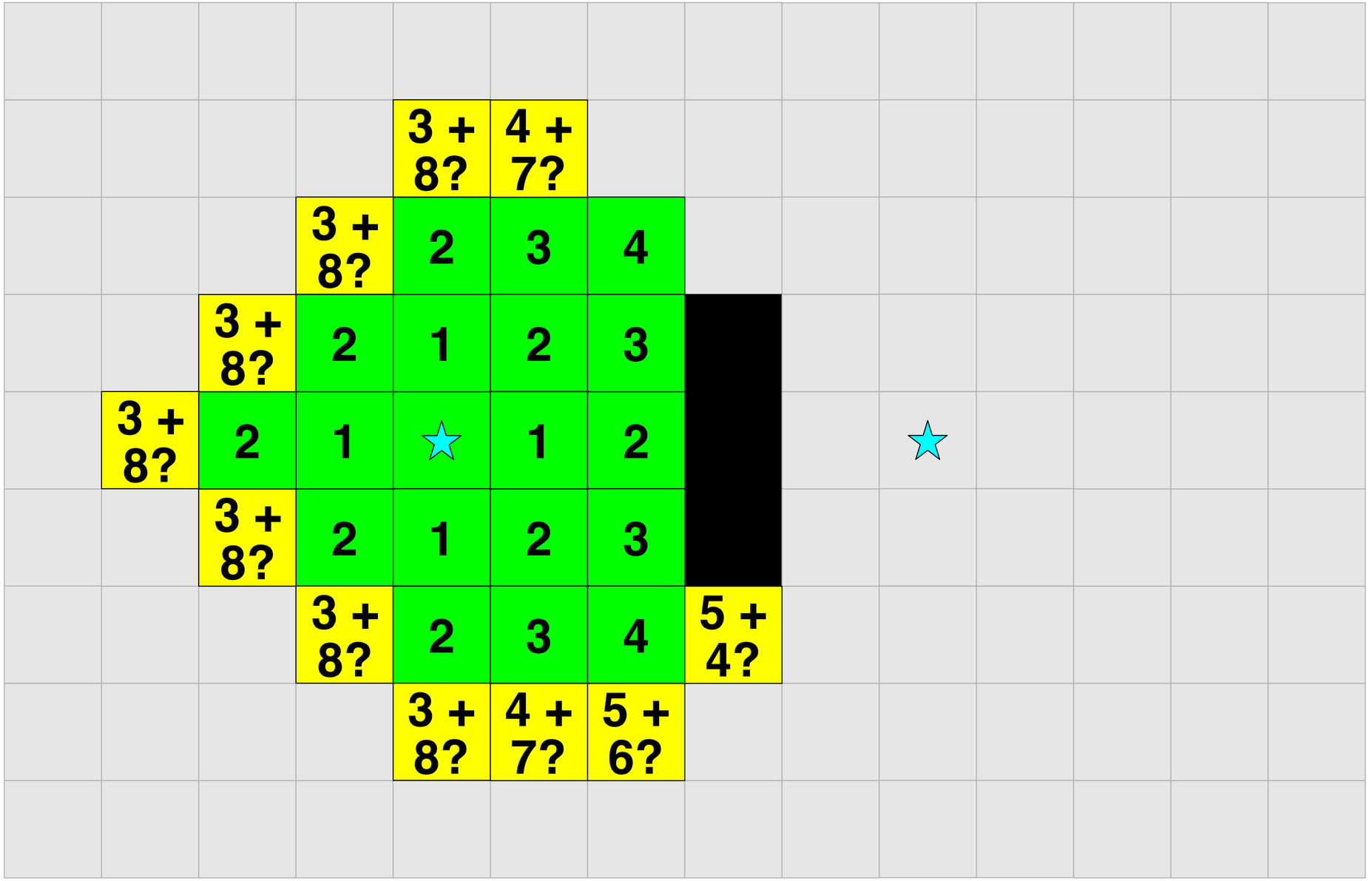

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
2 + 7?
1
2
3
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

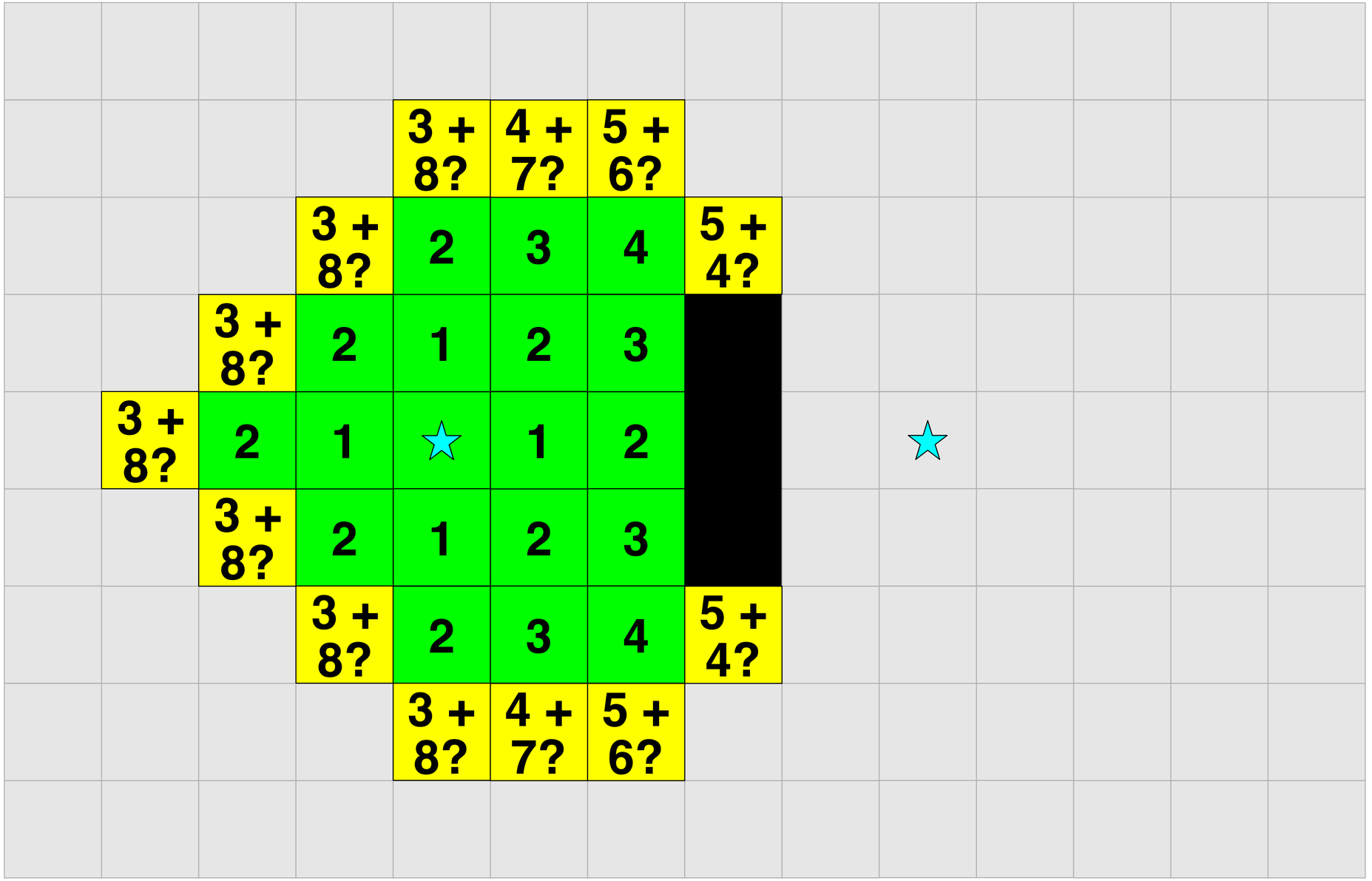

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
3 + 8?
2 + 7?
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

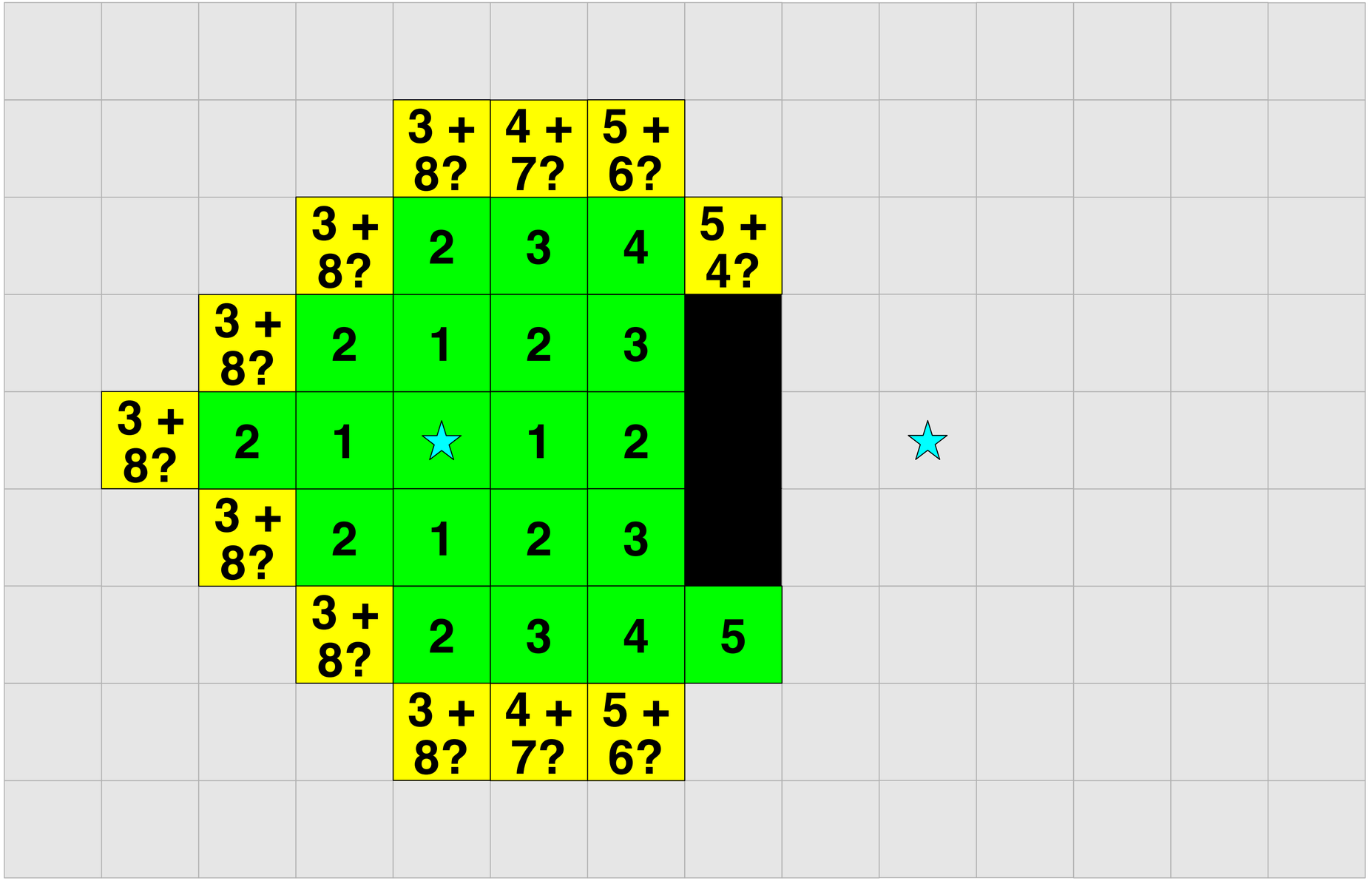

3 + 8?
3 + 8?
2
3 + 6?
4 + 5?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

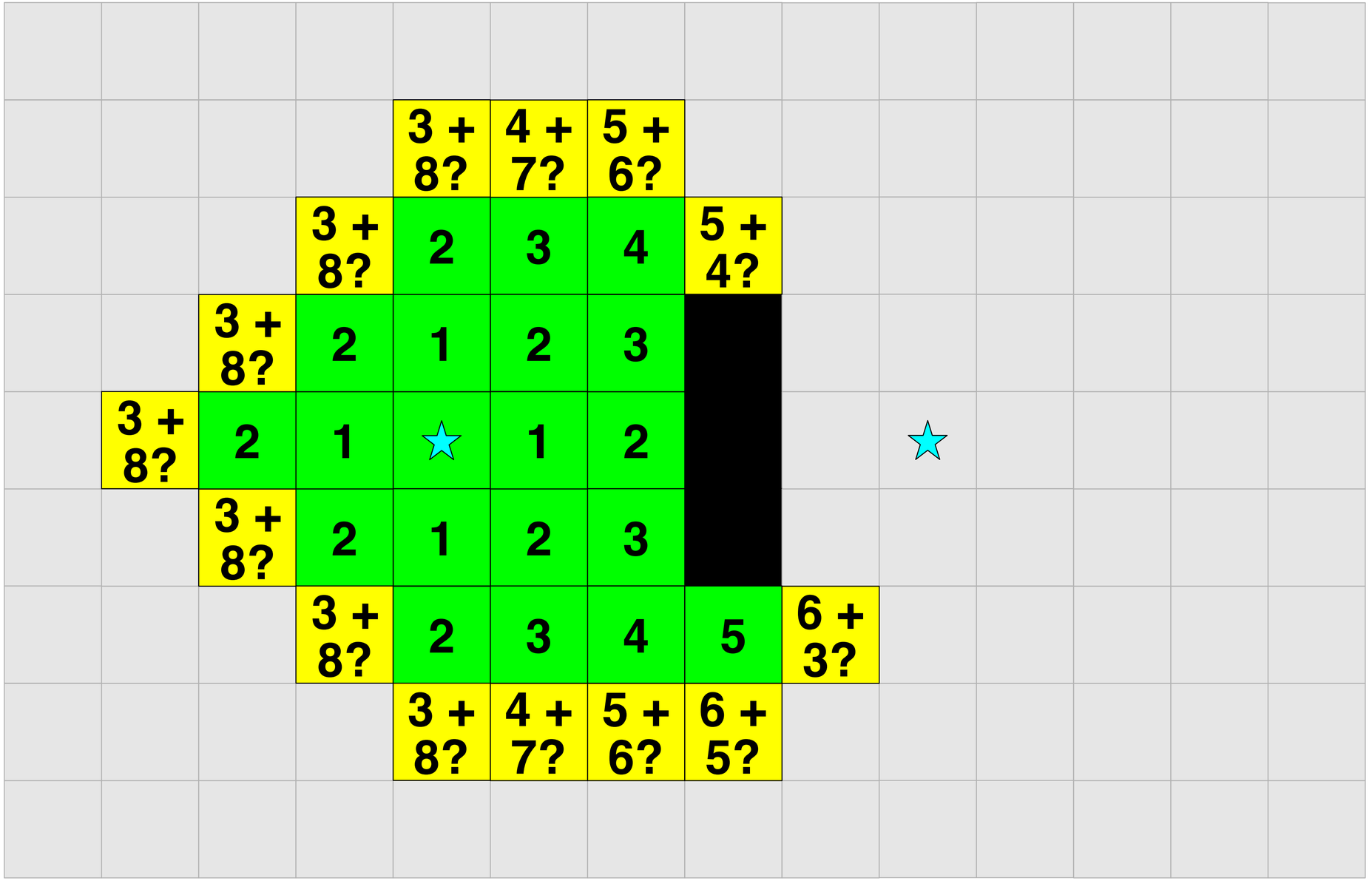

3 + 8?
3 + 8?
2
3
4 + 5?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

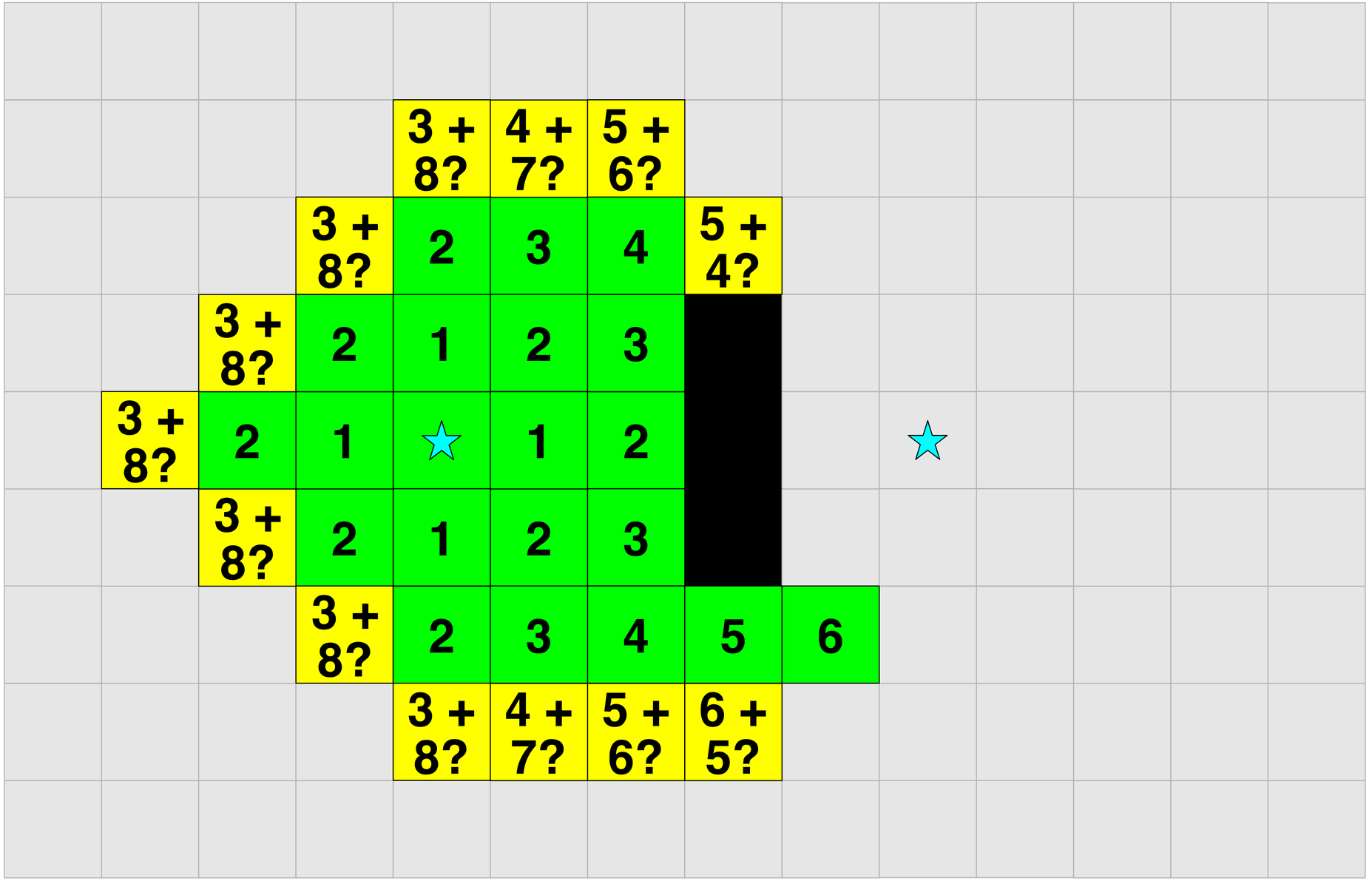

3 + 8?
4 + 7?
3 + 8?
2
3
4 + 5?
3 + 8?
2
1
2
3
3 + 8?
2
1
2
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

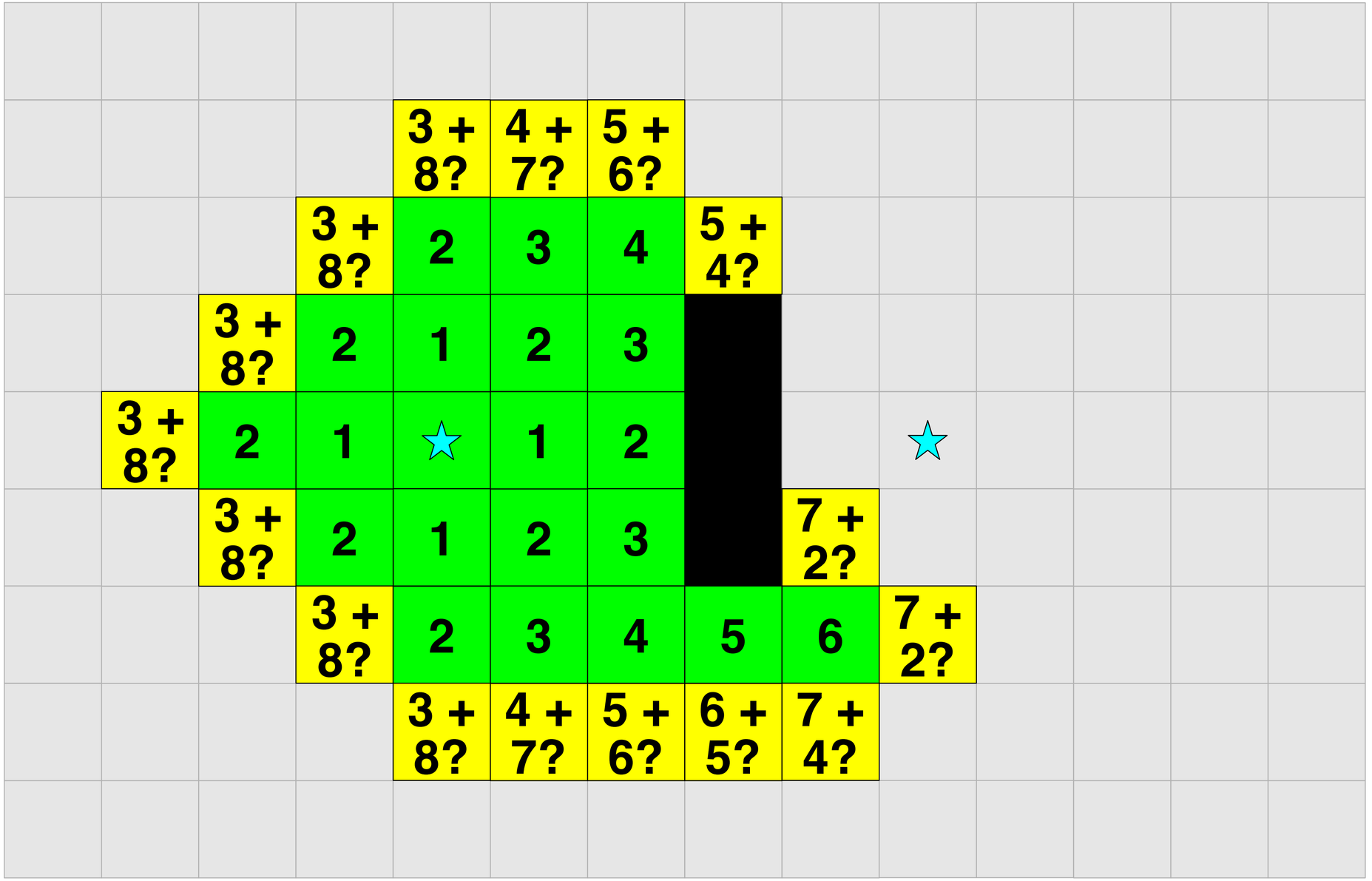

3 + 8?
4 + 7?
3 + 8?
2
3
4
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

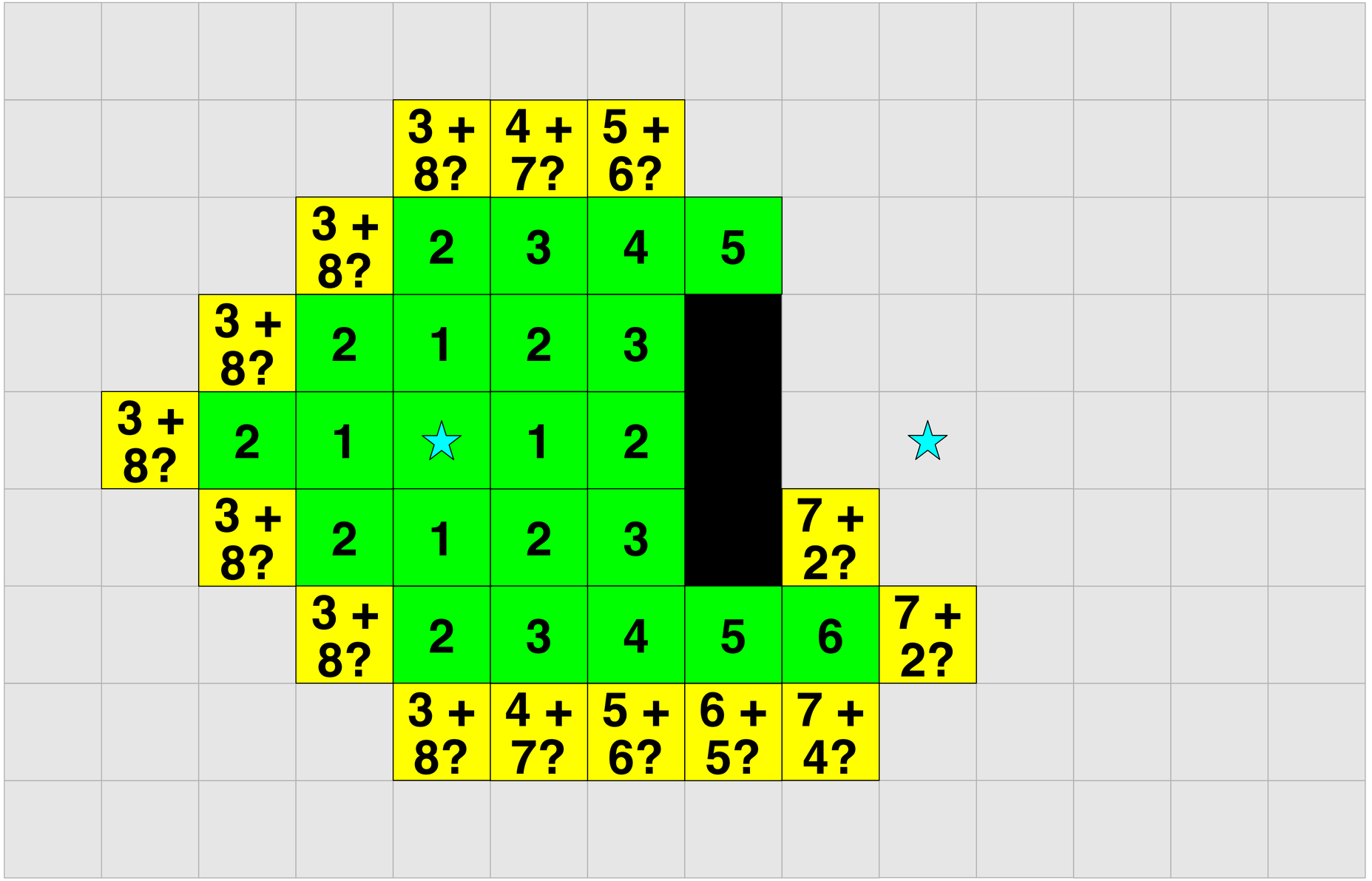

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5 + 4?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5 + 4?
3 + 8?
4 + 7?
5 + 6?

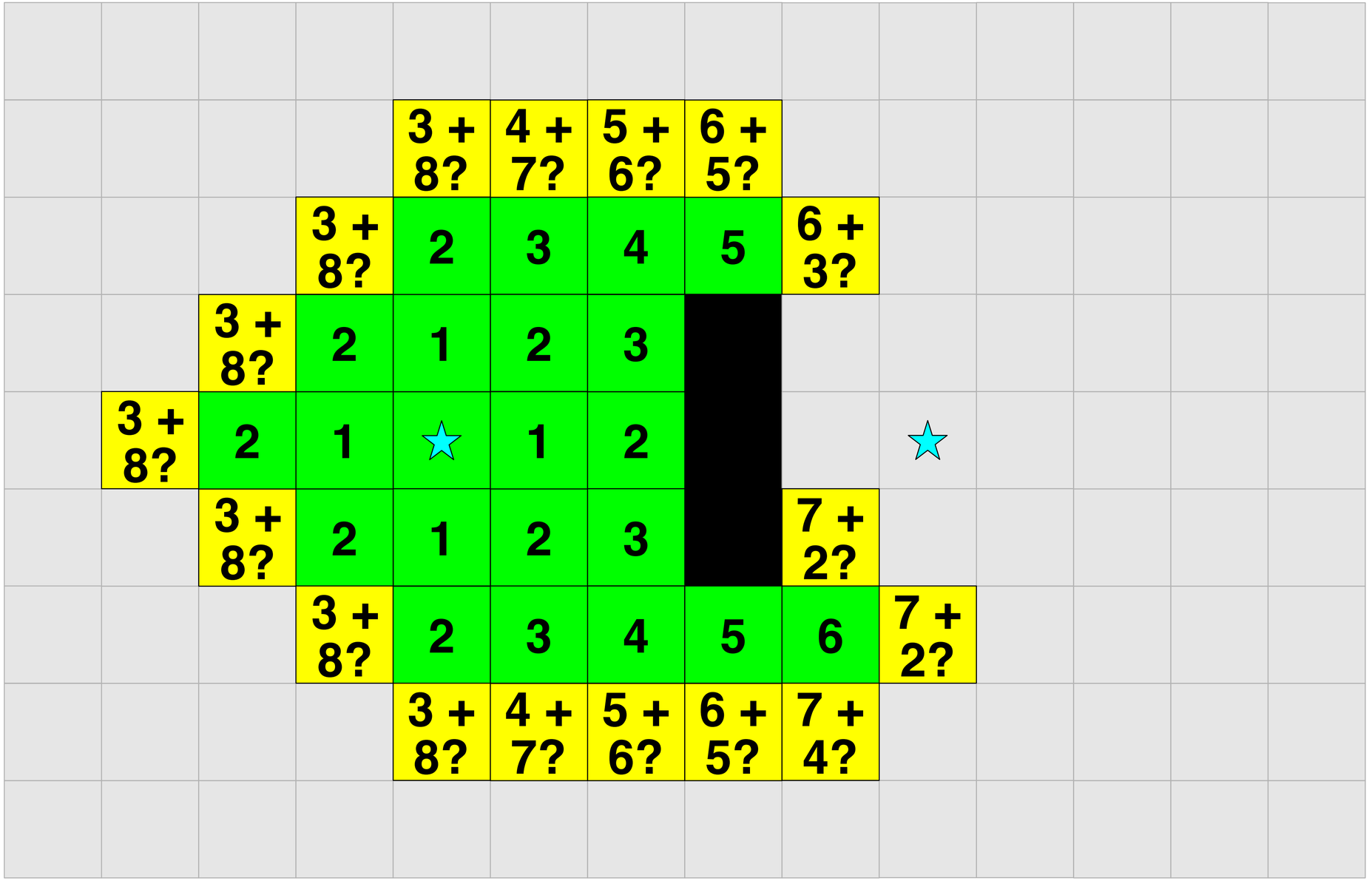

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5 + 4?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5
3 + 8?
4 + 7?
5 + 6?

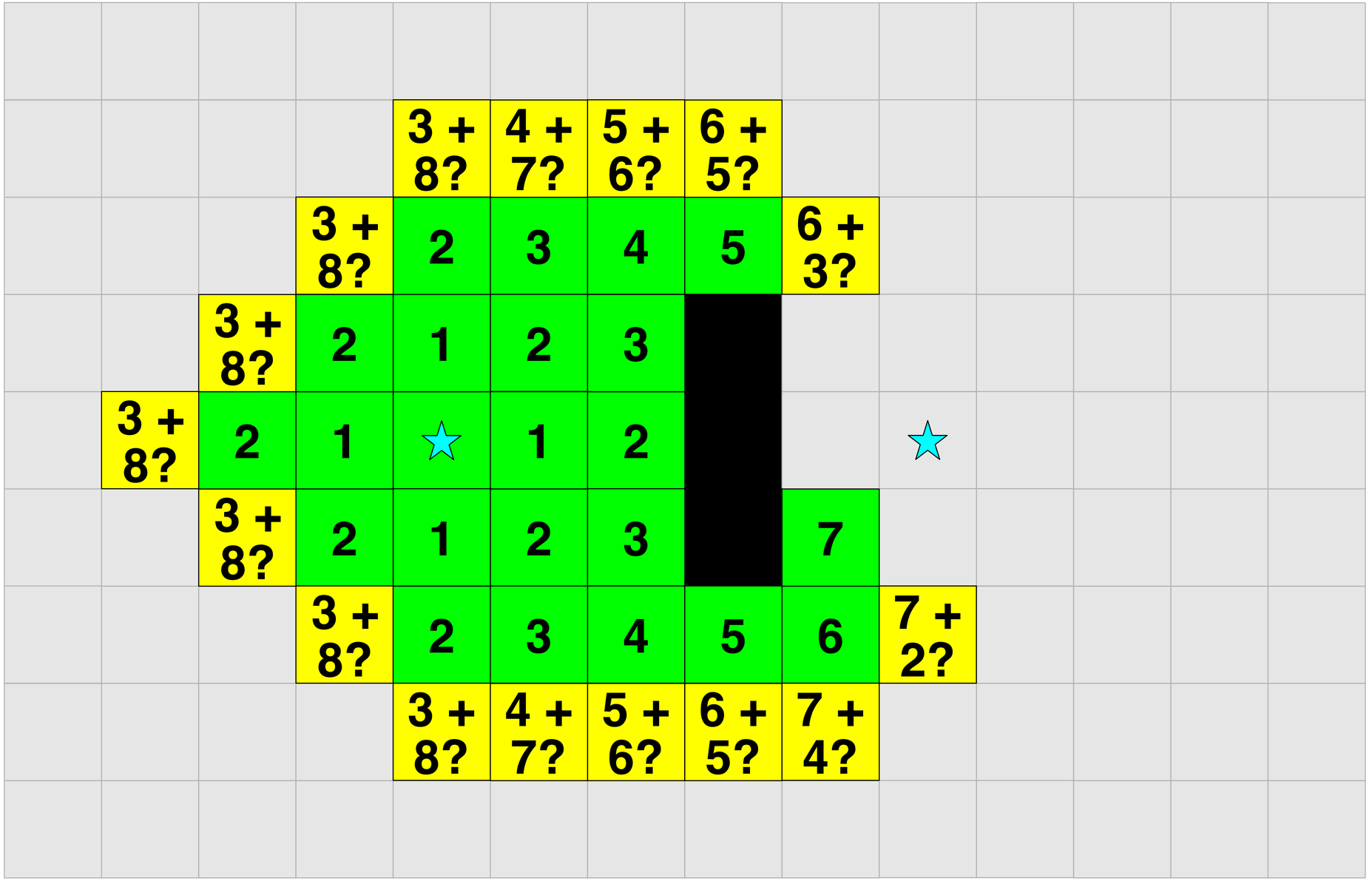

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5 + 4?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
4 + 7?
5 + 6?
6 + 5?

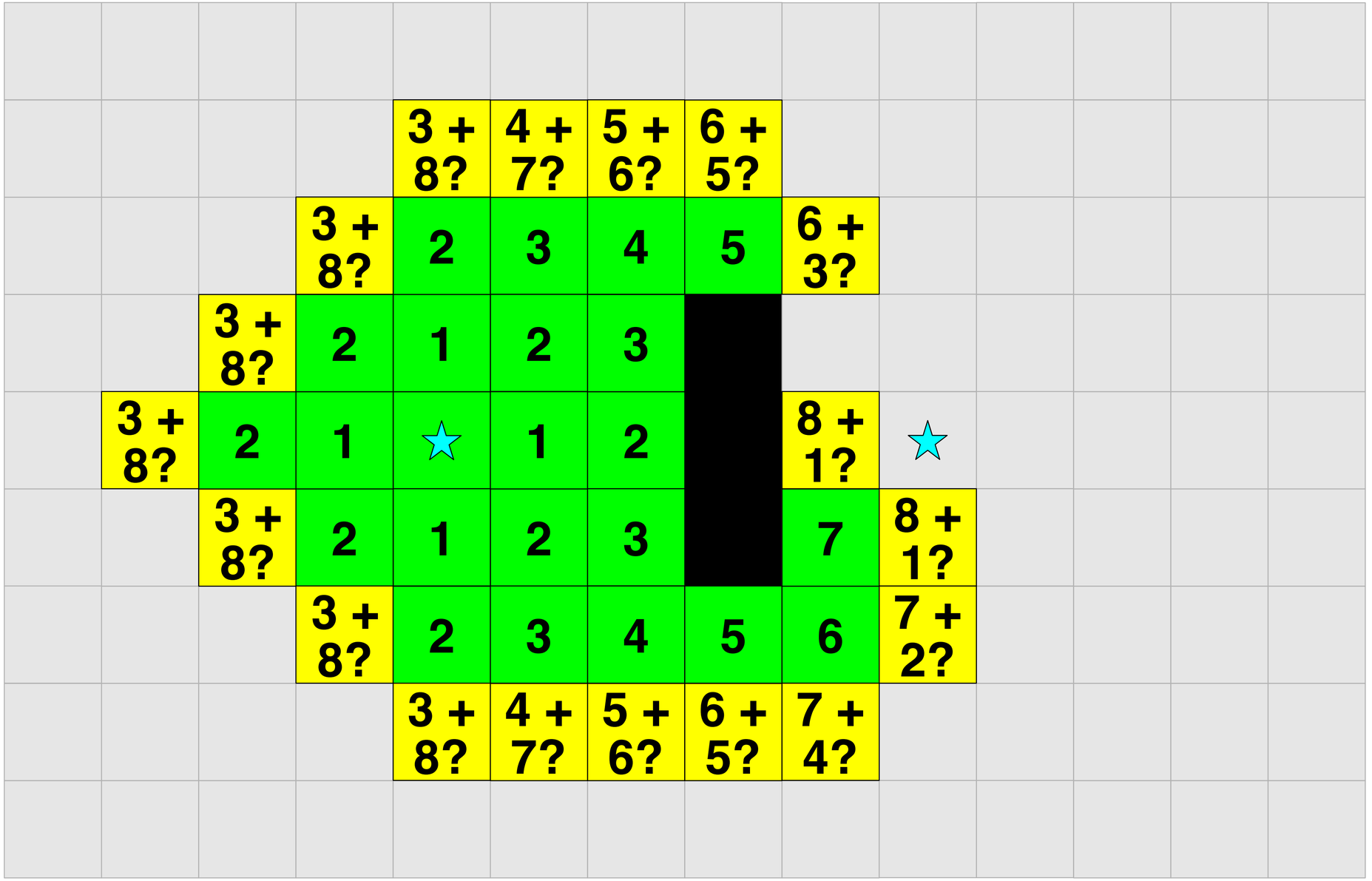

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5 + 4?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
3 + 8?
2
3
4
5
6
3 + 8?
4 + 7?
5 + 6?
6 + 5?

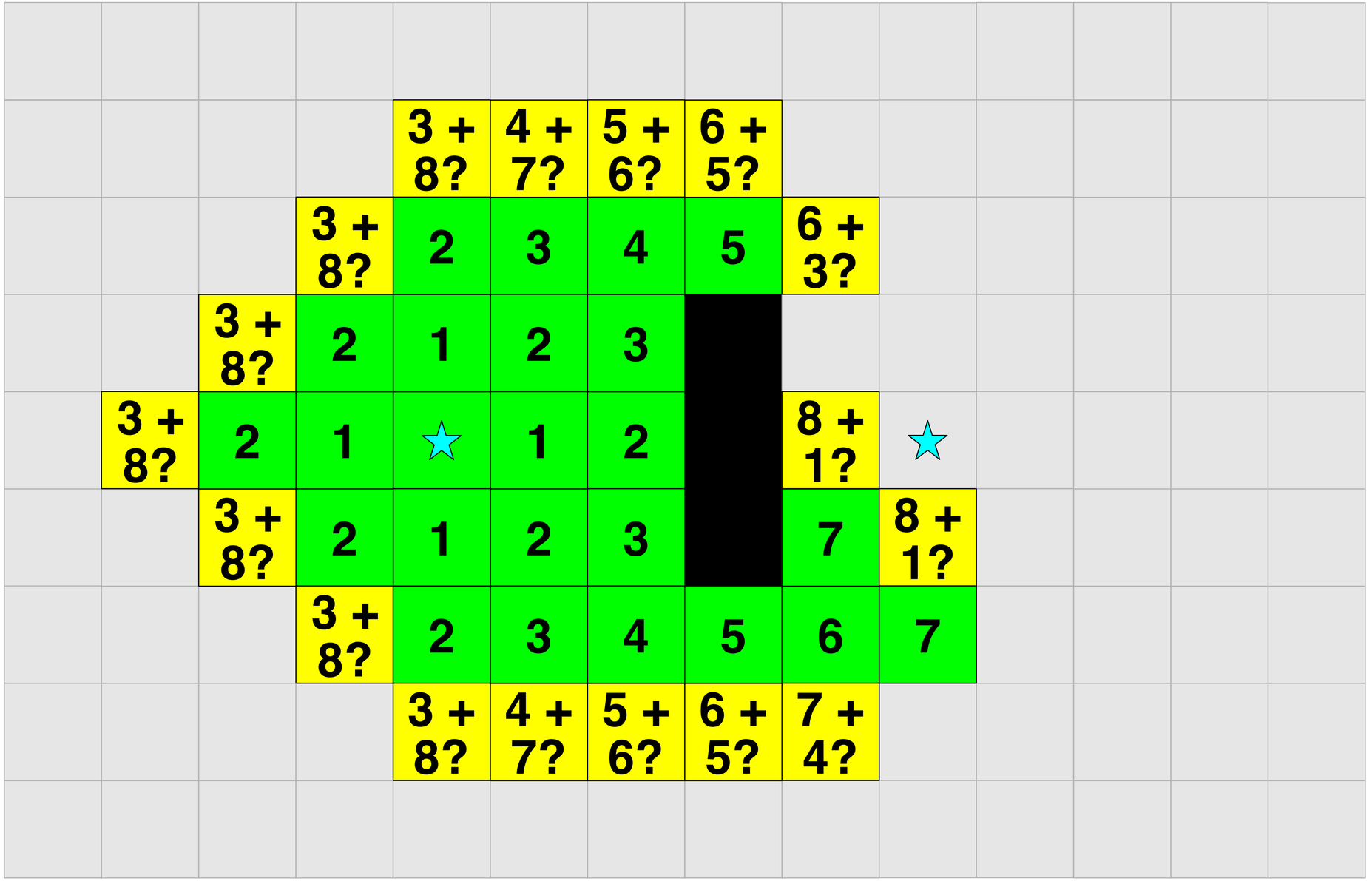

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5 + 4?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
7 + 2?
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

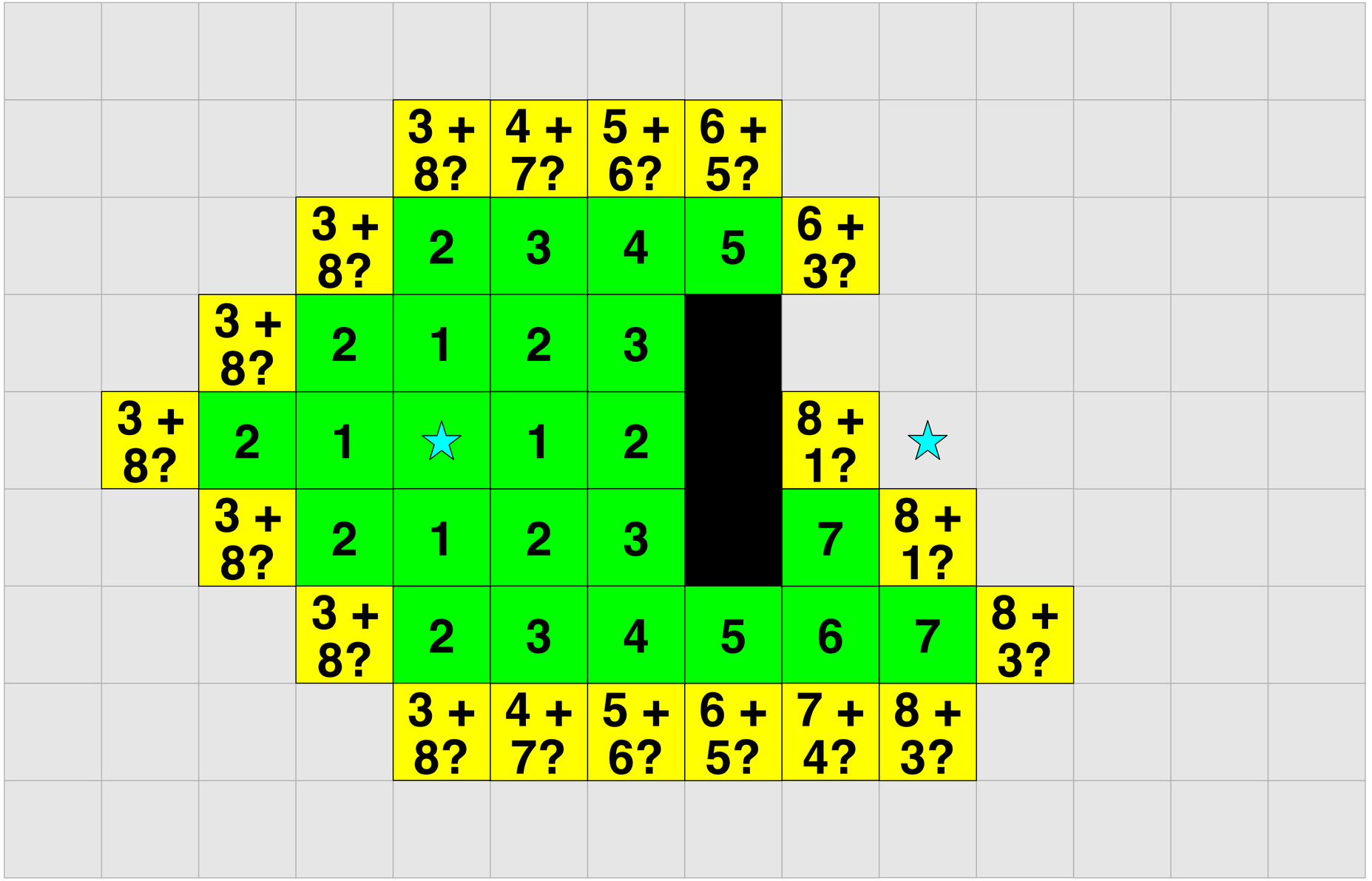

3 + 8?
4 + 7?
5 + 6?
3 + 8?
2
3
4
5
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
7 + 2?
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

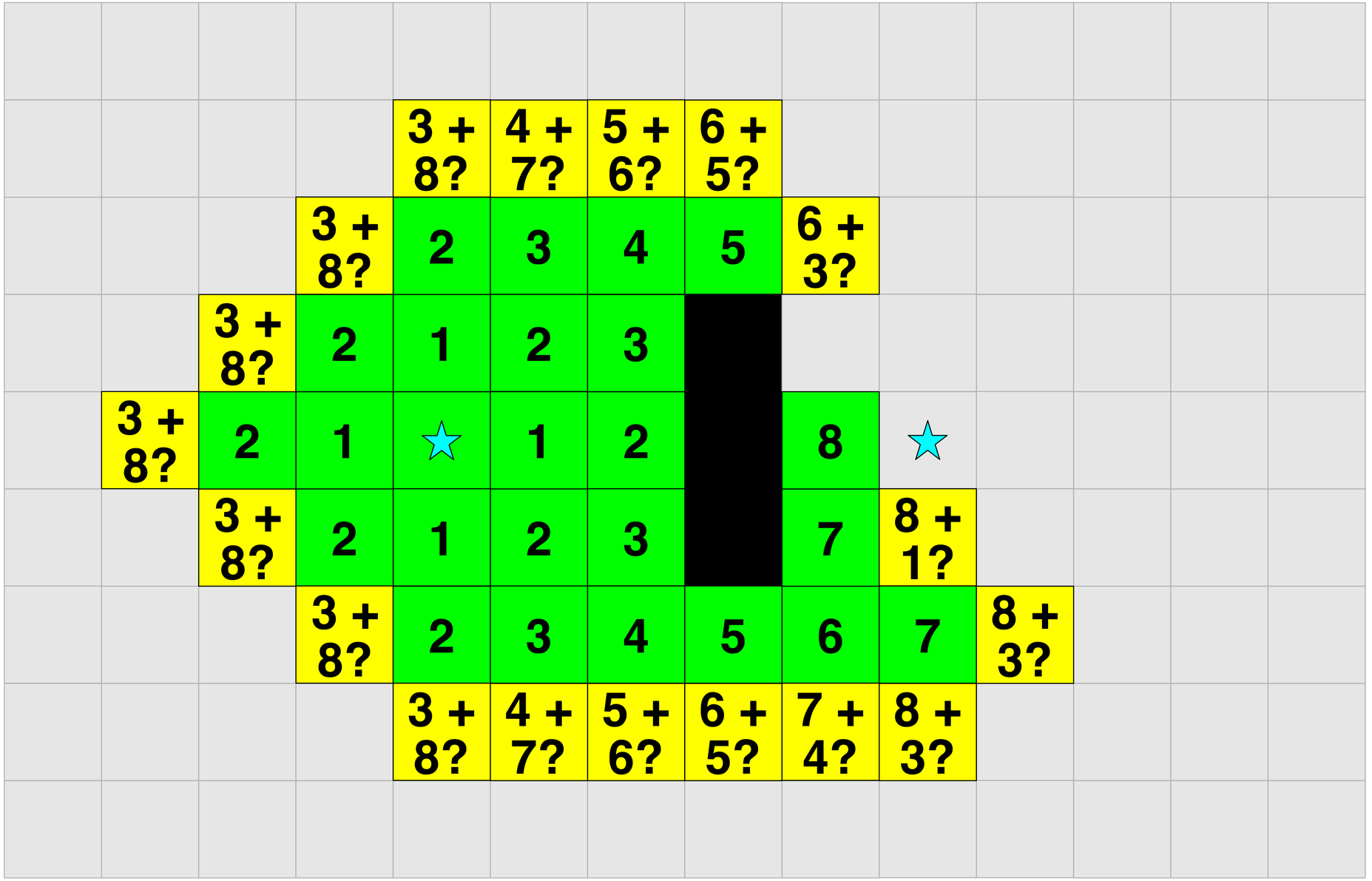

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
7 + 2?
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

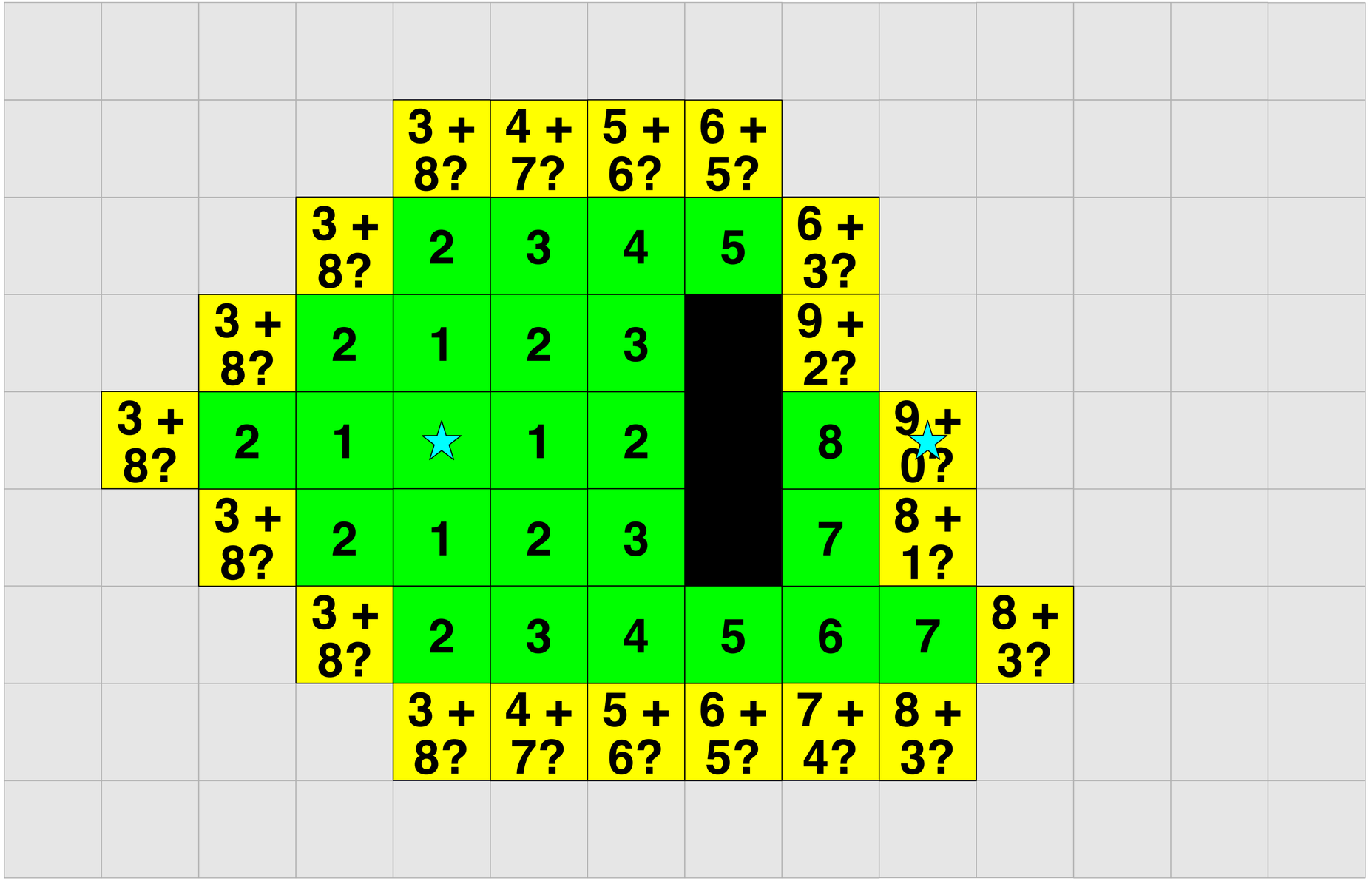

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
3 + 8?
2
1
2
3
7
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

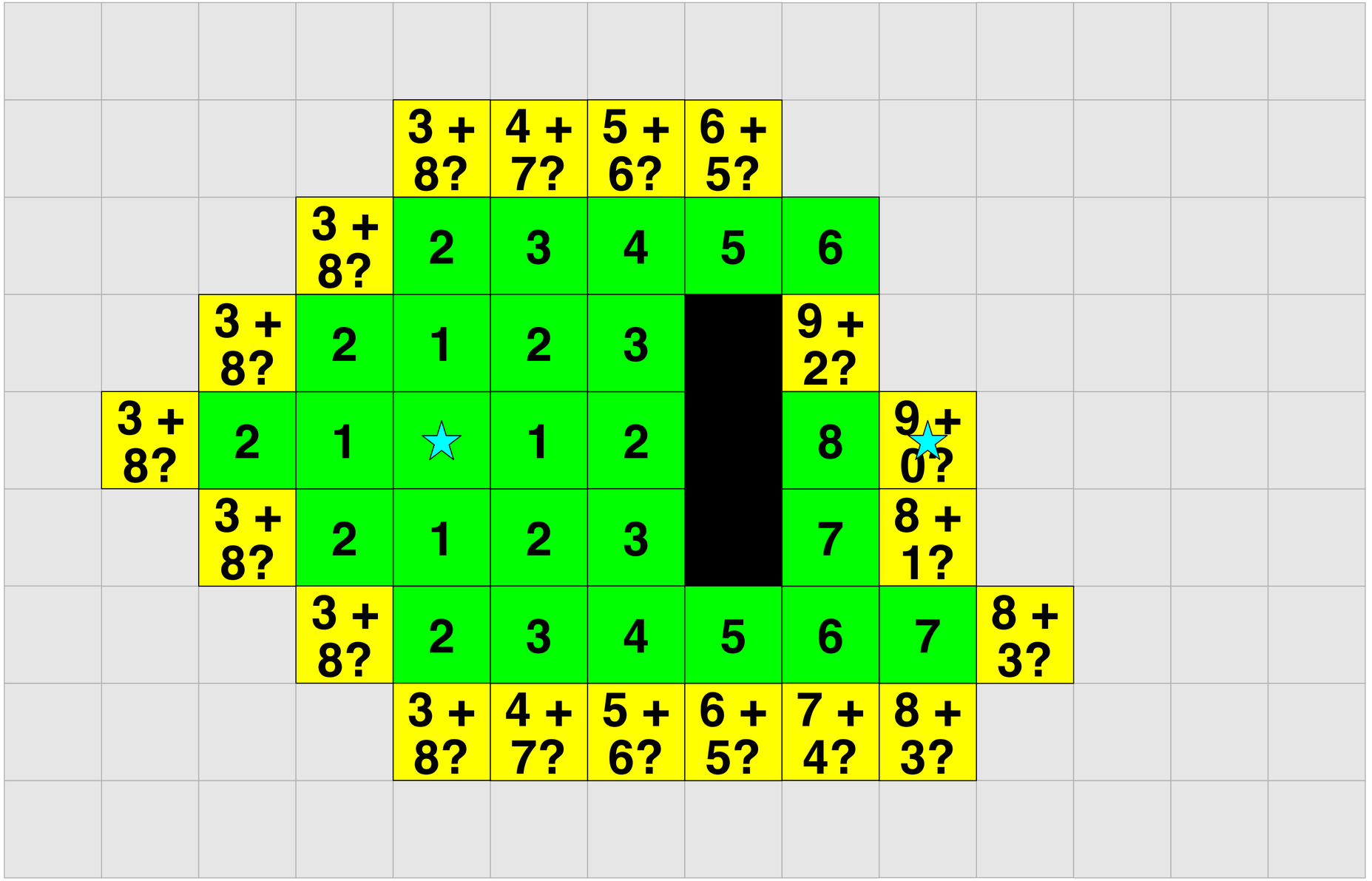

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
8 + 1?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

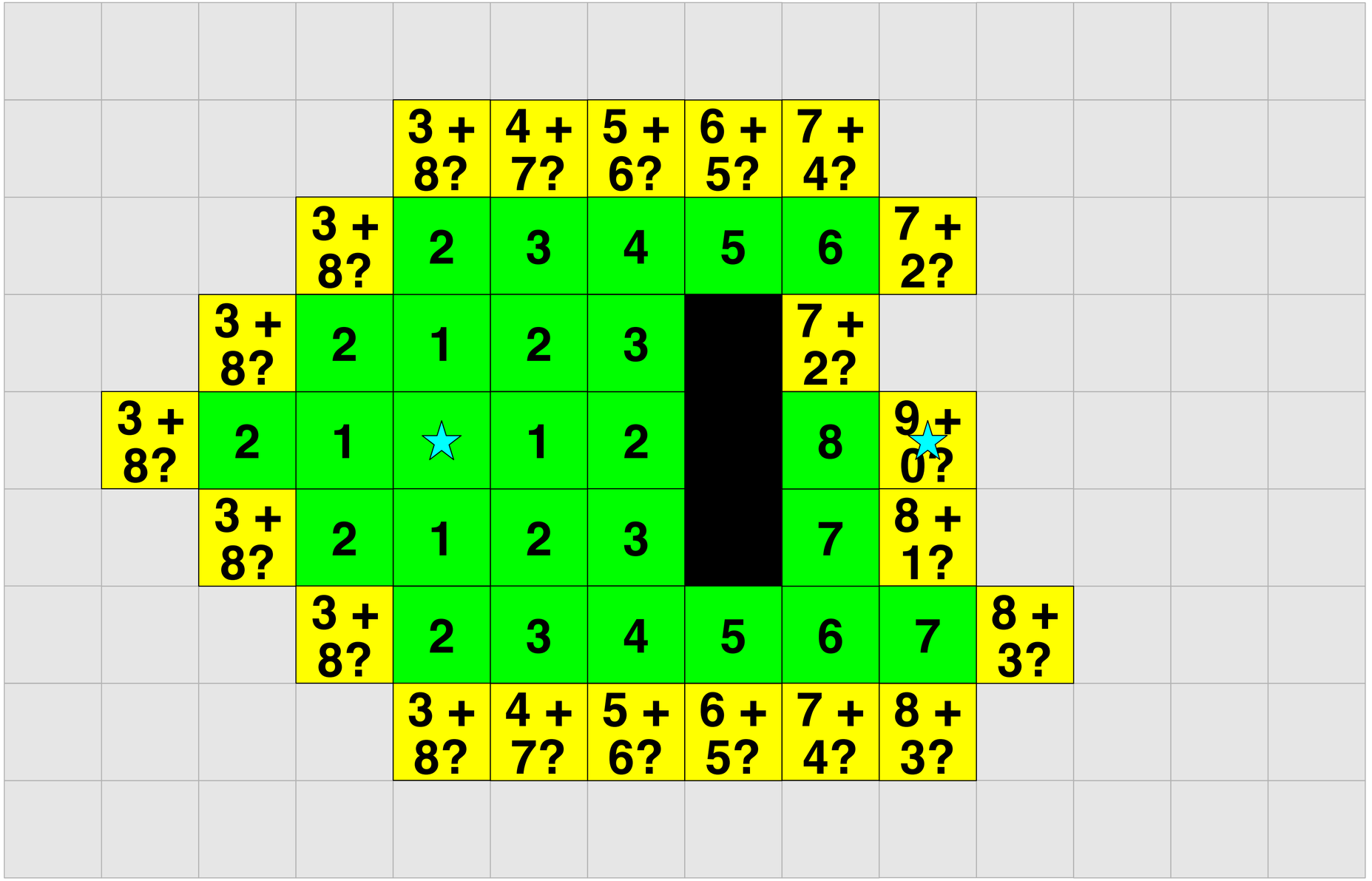

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
8 + 1?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
3
4
5
6
7
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?

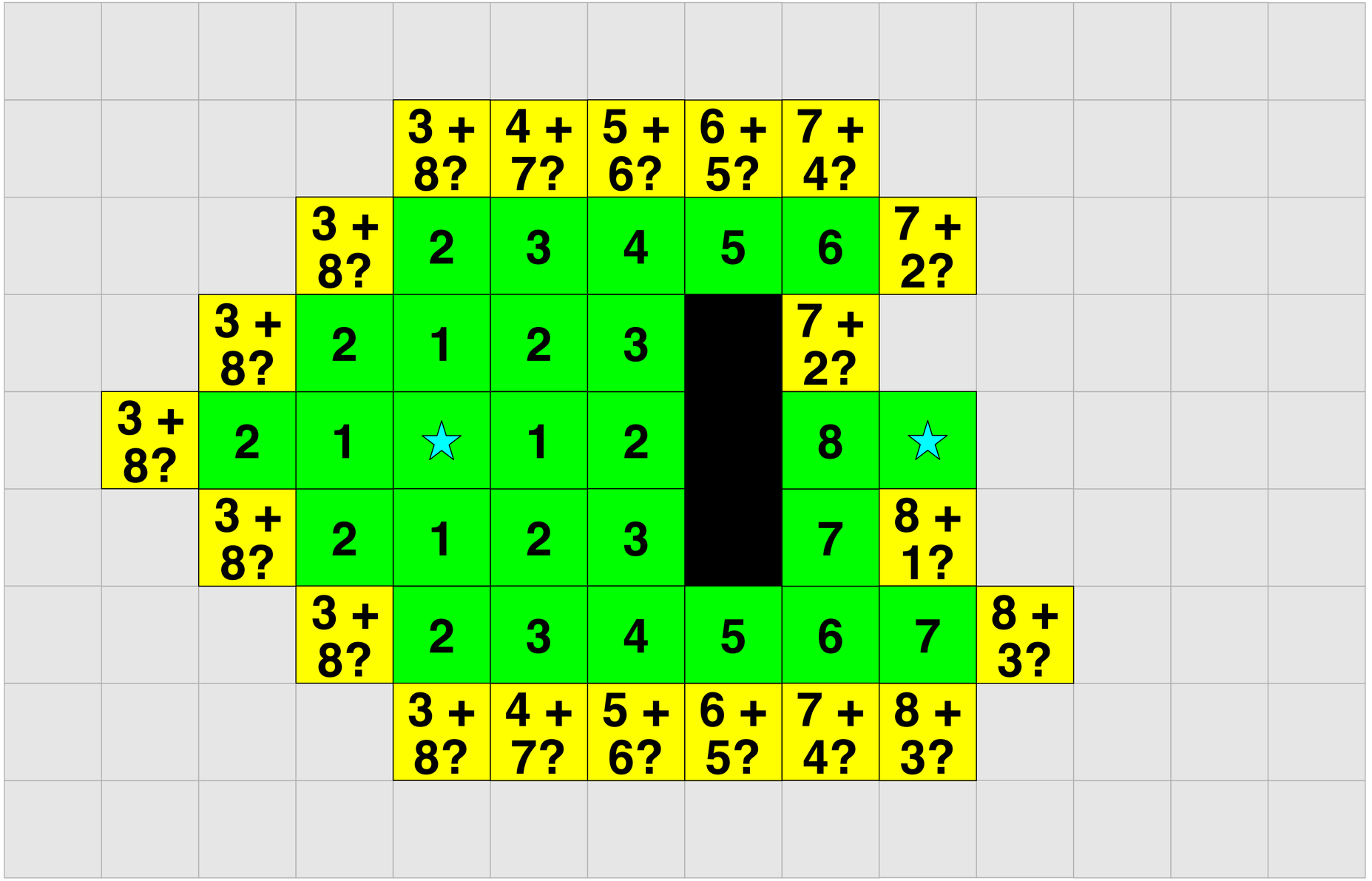

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
3 + 8?
2
1
1
2
8 + 1?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
3
4
5
6
7
8 + 3?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?
8 + 3?


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

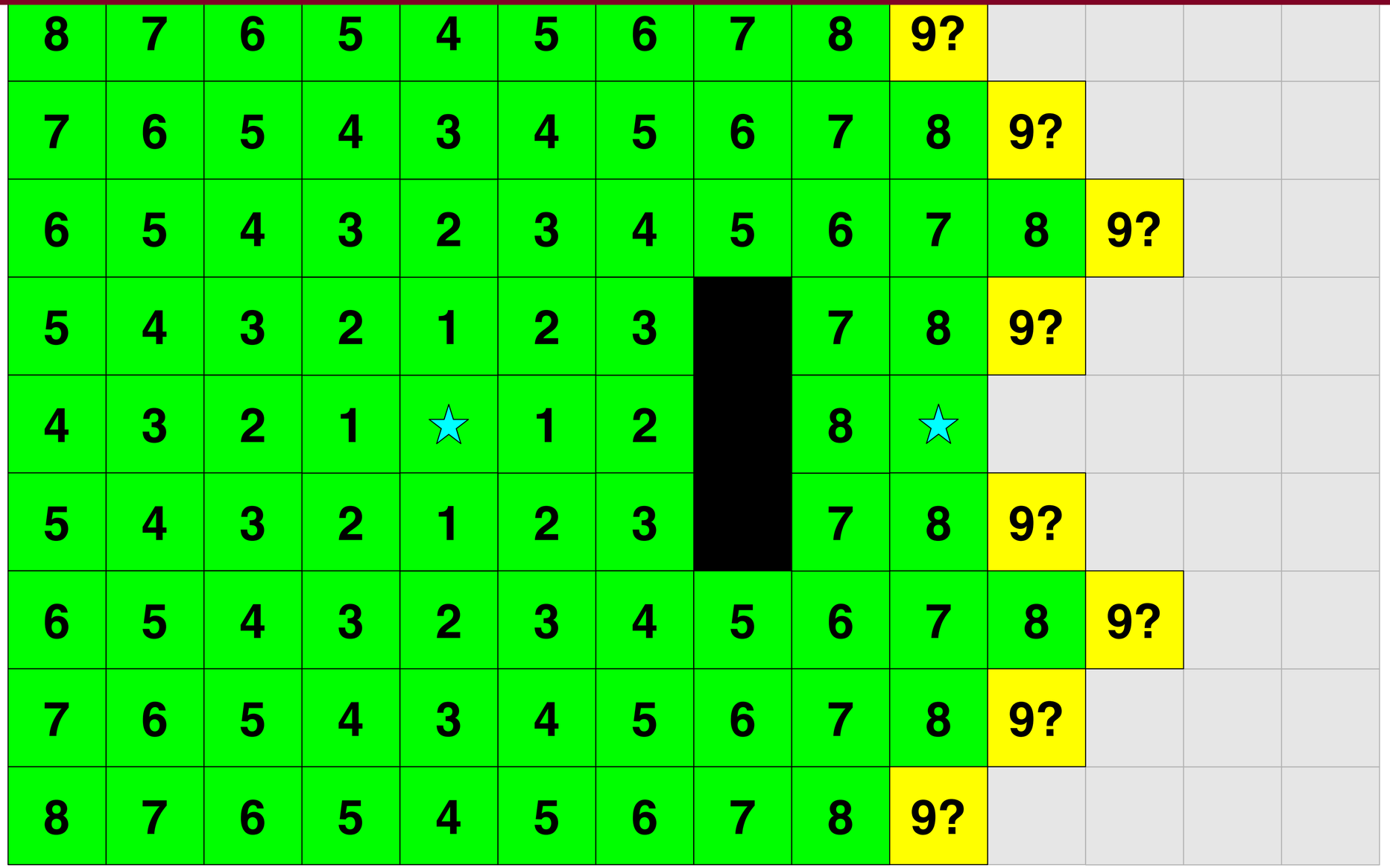

3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6 + 3?
3 + 8?
2
1
2
3
9 + 2?
3 + 8?
2
1
1
2
8
9 + 0?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
3
4
5
6
7
8 + 3?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?
8 + 3?


3 + 8?
4 + 7?
5 + 6?
6 + 5?
3 + 8?
2
3
4
5
6
3 + 8?
2
1
2
3
9 + 2?
3 + 8?
2
1
1
2
8
9 + 0?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
1
2
3
4
5
6
7
8 + 3?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?
8 + 3?


3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?
3 + 8?
2
3
4
5
6
7 + 2?
3 + 8?
2
1
2
3
7 + 2?
3 + 8?
2
1
1
2
8
9 + 0?
3 + 8?
2
1
2
3
7
8 + 1?
3 + 8?
2
3
4
5
6
7
8 + 3?
3 + 8?
4 + 7?
5 + 6?
6 + 5?
7 + 4?
8 + 3?


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

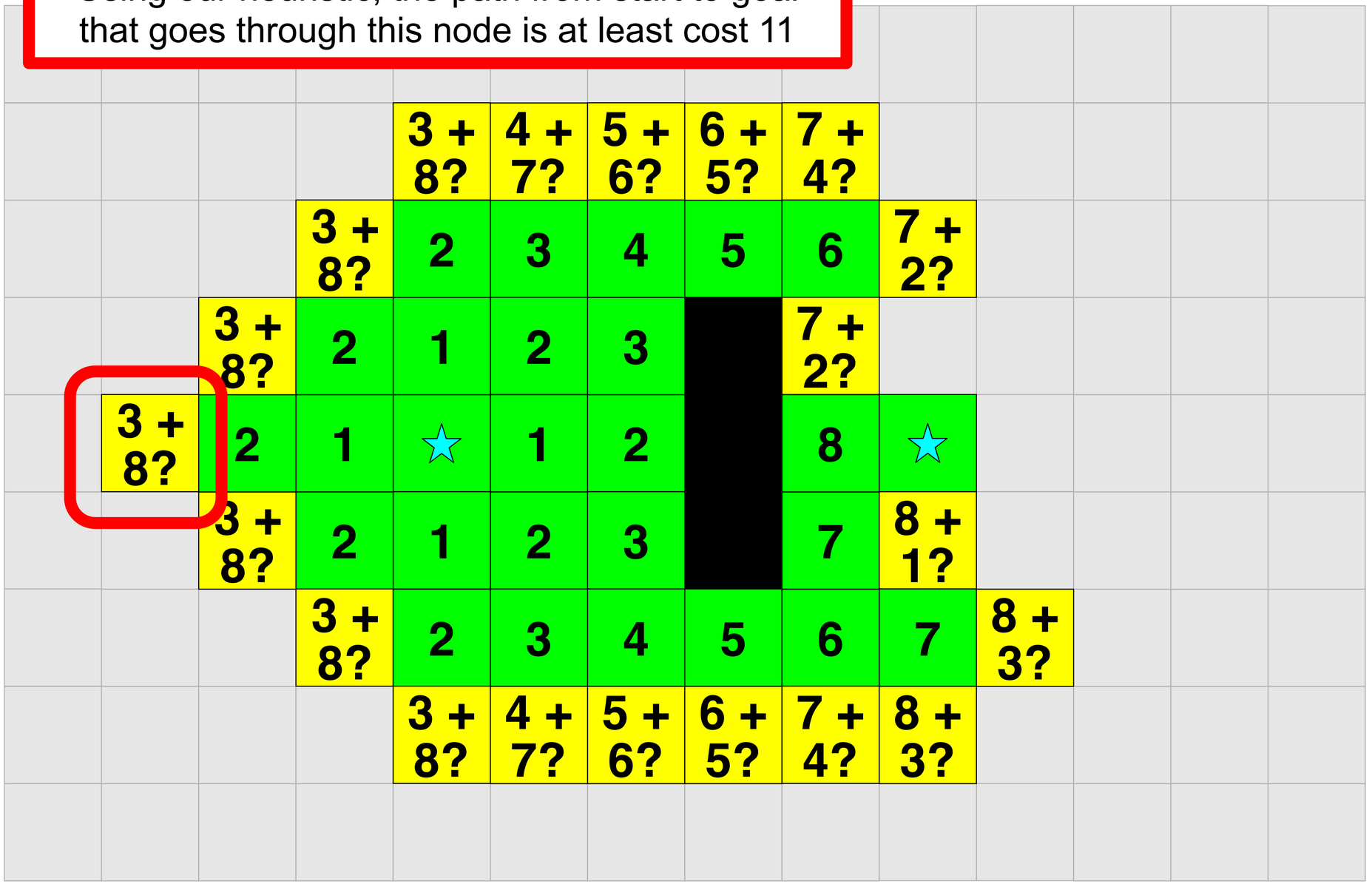
What Dijkstra would have selected!



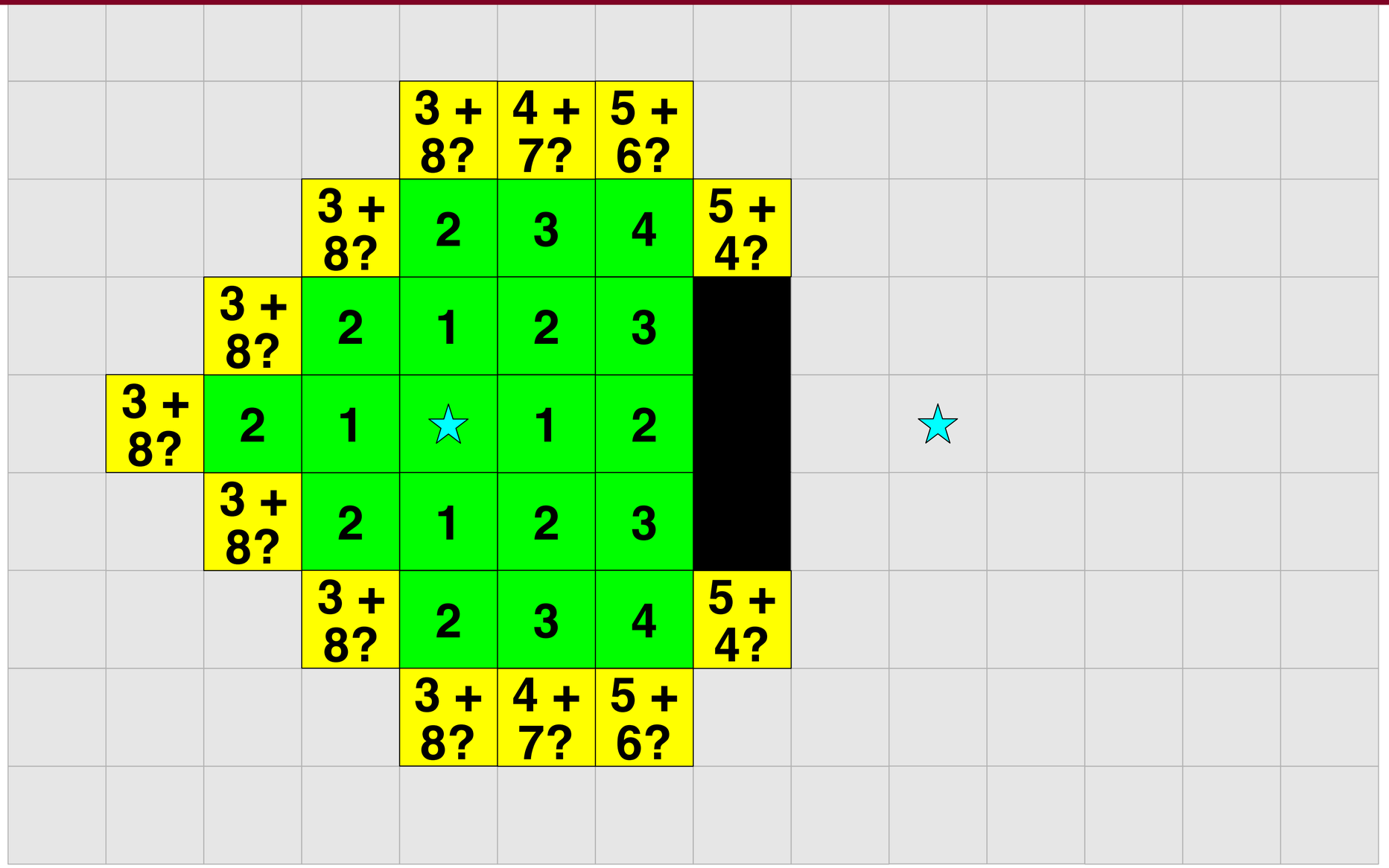
Why underestimate?

You only ignore paths that in the best case are worse than your current path.

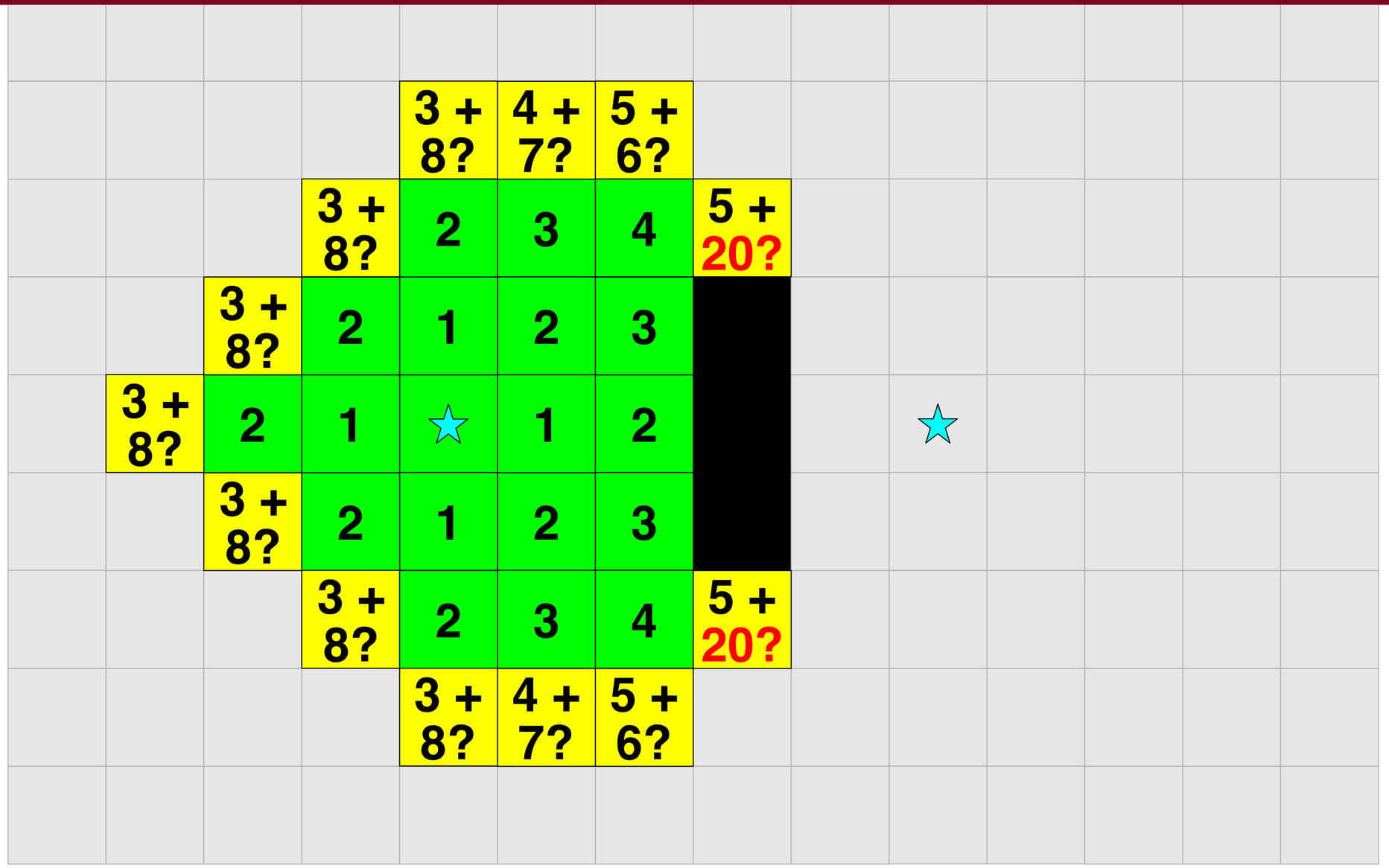
Using our heuristic, the path from start to goal that goes through this node is at least cost 11



Imagine if we overestimate

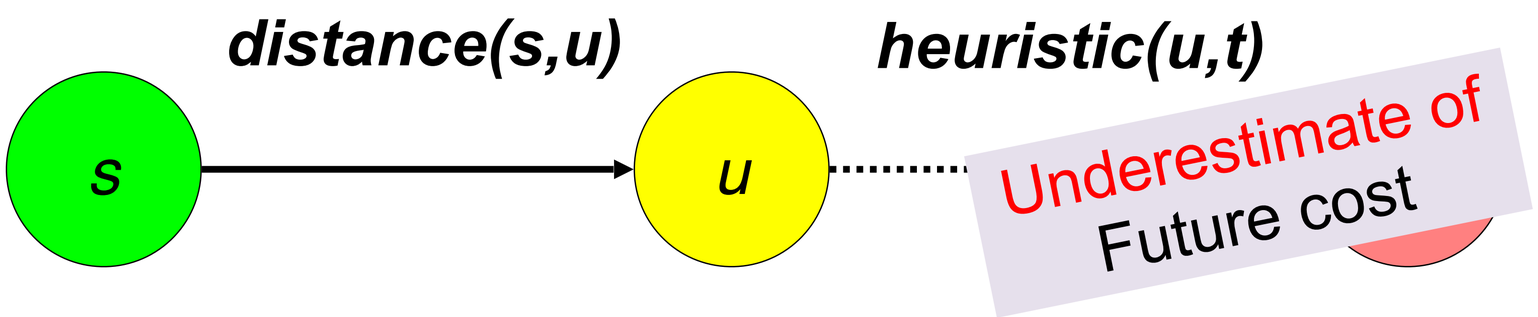


Imagine if we overestimate



More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)

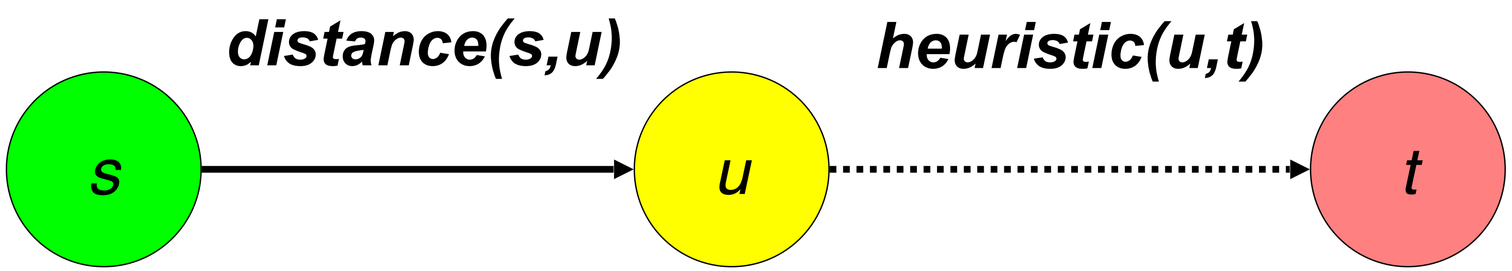


We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



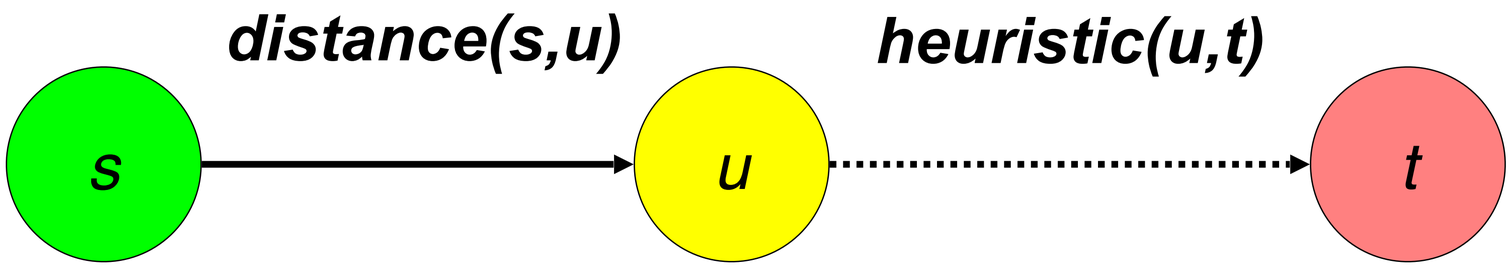
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Same as Dijkstra

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0

heuristic(u,t) = underestimate

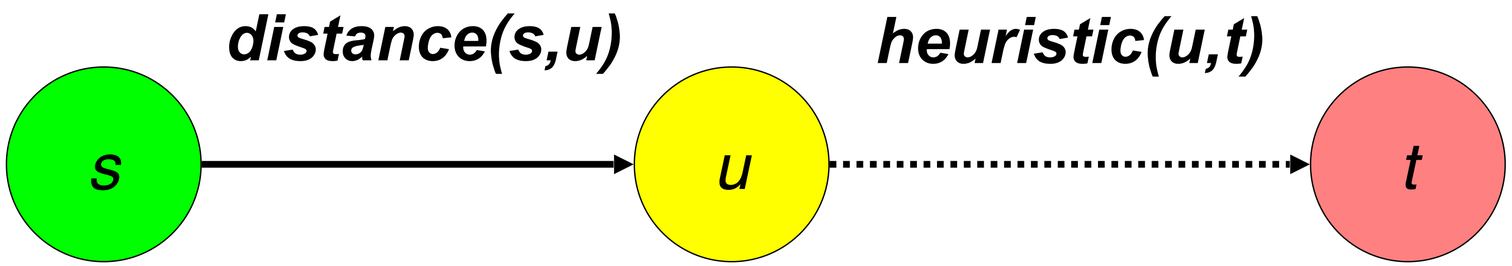
heuristic(u,t) = perfect distance

heuristic(u,t) = overestimate

Will be the same or faster than Dijkstra, and will find the shortest path (this is the only "admissible" heuristic for A*.

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



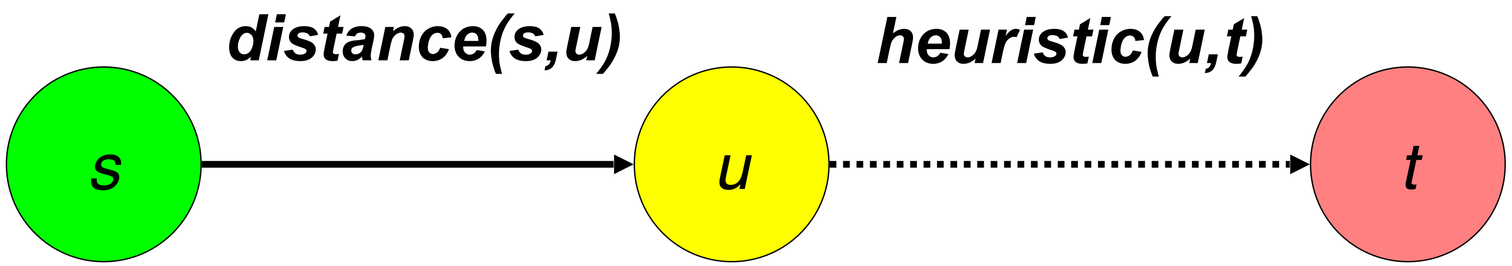
We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Will only follow the best path, and will find the best path fastest (but requires perfect knowledge)

More Detail on A*: Choice of Heuristic

priority(u) = distance(s, u) + heuristic(u, t)



We want to underestimate the cost of our heuristic, by why?
Let's look at the bounds of our choices:

heuristic(u,t) = 0
heuristic(u,t) = underestimate
heuristic(u,t) = perfect distance
heuristic(u,t) = overestimate

Won't necessarily find shortest path (but might run even faster)

Admissible Heuristic

Definition: An admissible heuristic always underestimates the true cost.

Could you precompute this for all your vertices? Yes, but it would not be feasible.

https://media.giphy.com/media/GEPHf81p4svkI/giphy.gif

Extra Slides