Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

THE LIFE CHANGING MAGIC OF

DIJKSTRA AND A*

Friday, March 10, 2017
Reading: Programming Abstractions in C++, Chapter 18.6

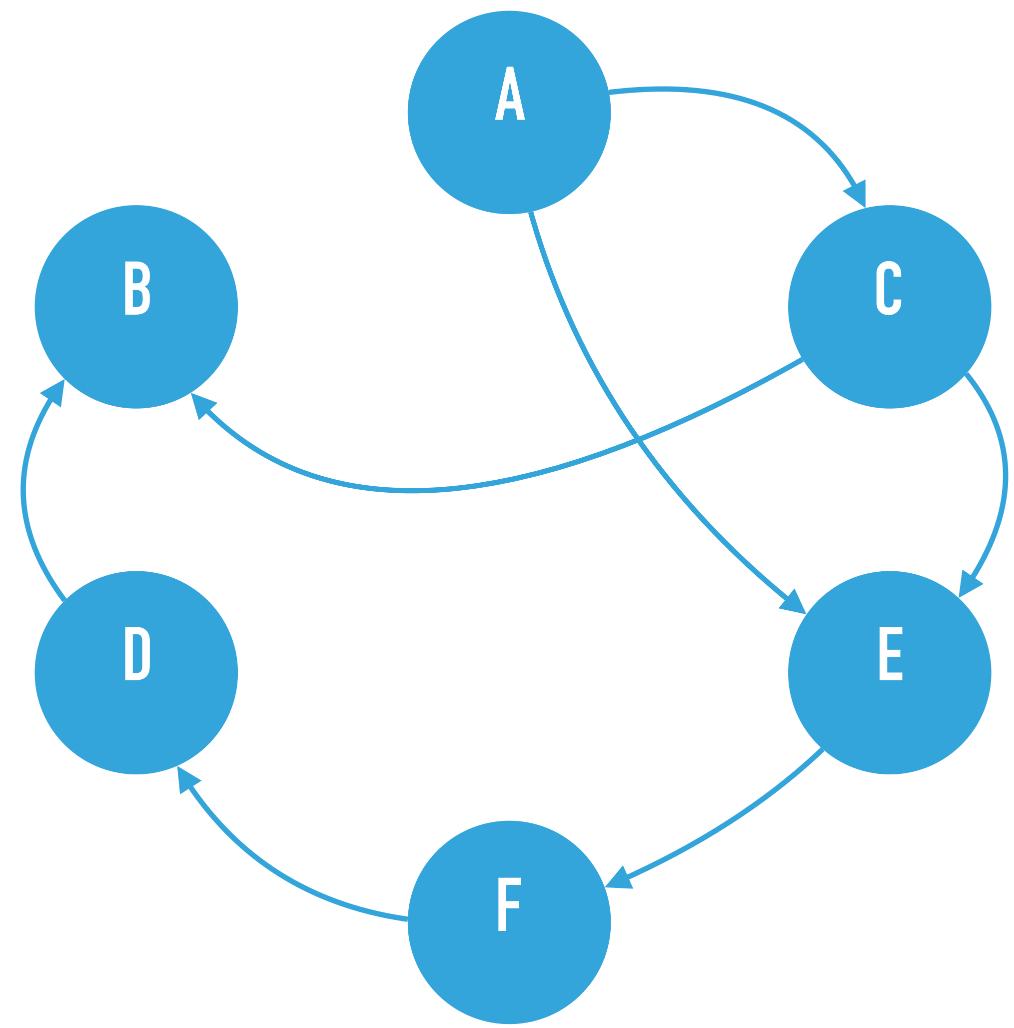
TODAY'S TOPICS - MORE GRAPHS!

- Reviewing DFS and BFS
- Comparing DFS and BFS
- Making weighty decisions using Dijkstra's algorithm
- Looking into the future with A*
- Google Maps

REVIEWING DFS AND BFS

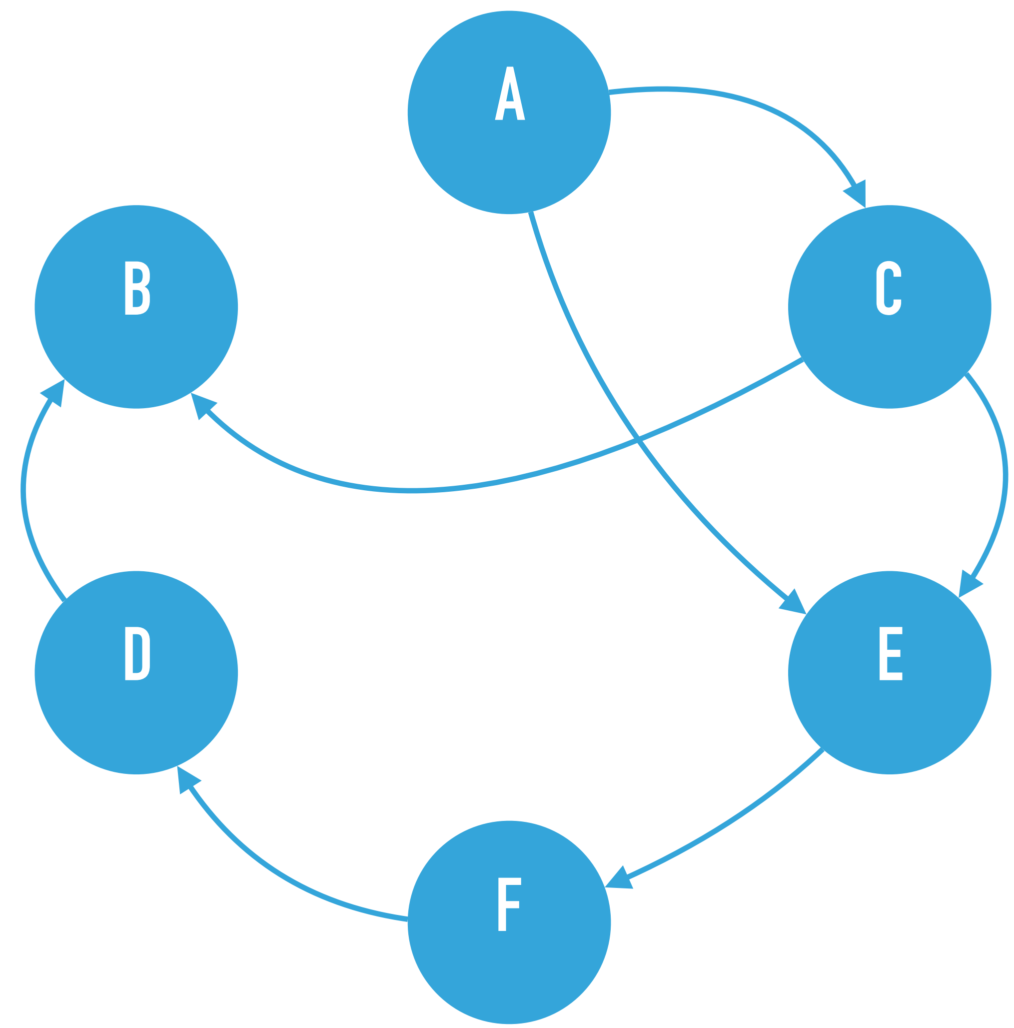
DEPTH FIRST SEARCH

- Find a path from A to B using *iterative* depth first search
  - (Assume that nodes are pushed onto the stack in *alphabetic order*)



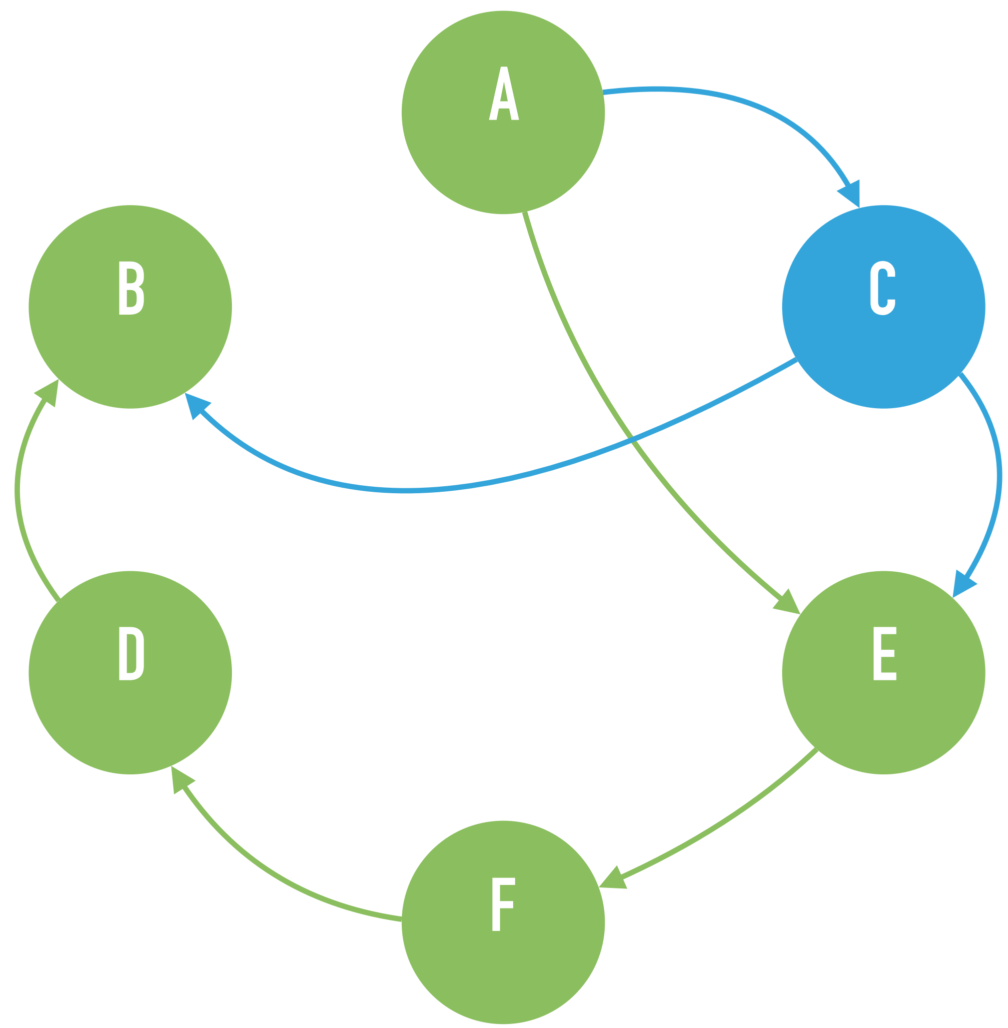
DEPTH FIRST SEARCH (ITERATIVE PSEUDOCODE)

- create a path with just start node and push onto stack s
- while s is not empty
  - p = s.pop()
  - v = last node of p
  - if v is end, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto s



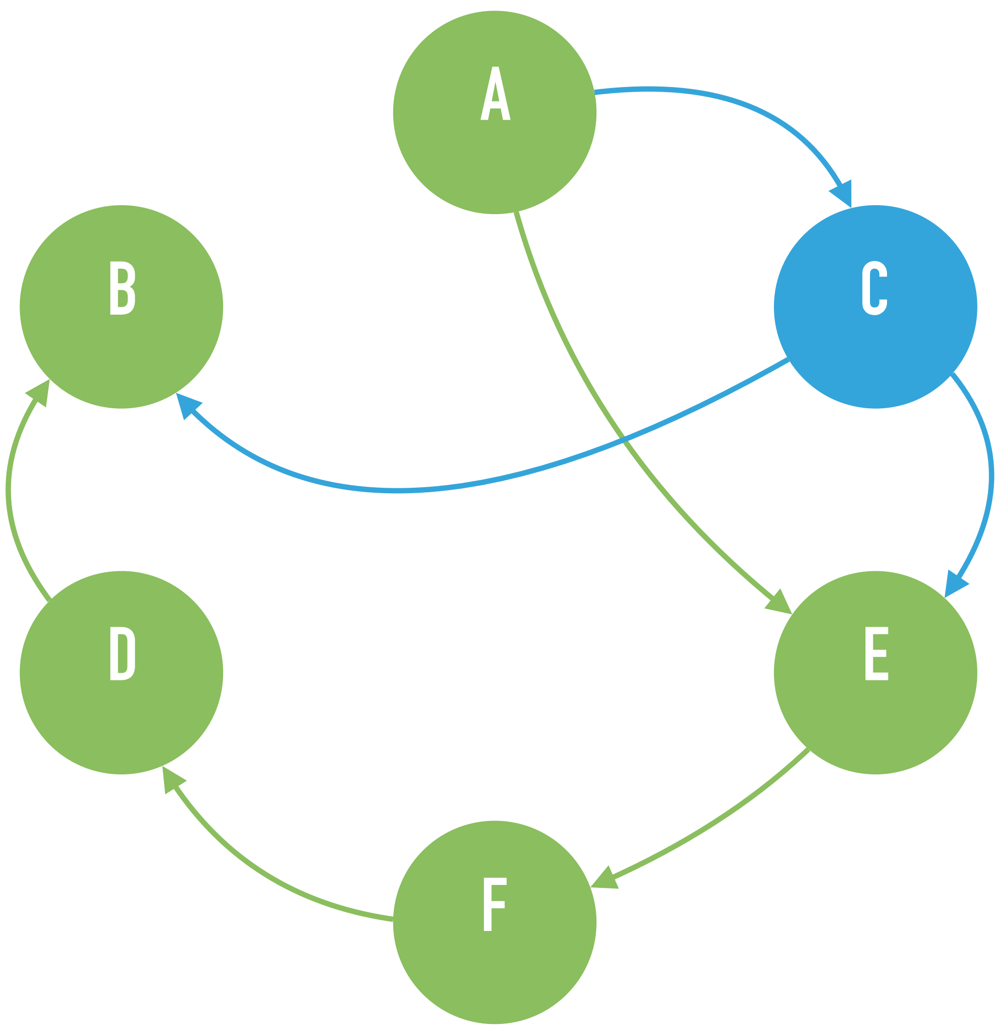
DEPTH FIRST SEARCH

- Find a path from A to B using *iterative* depth first search
  - (Assume that nodes are pushed onto the stack in *alphabetic order*)
- A ➔ E ➔ F ➔ D ➔ B

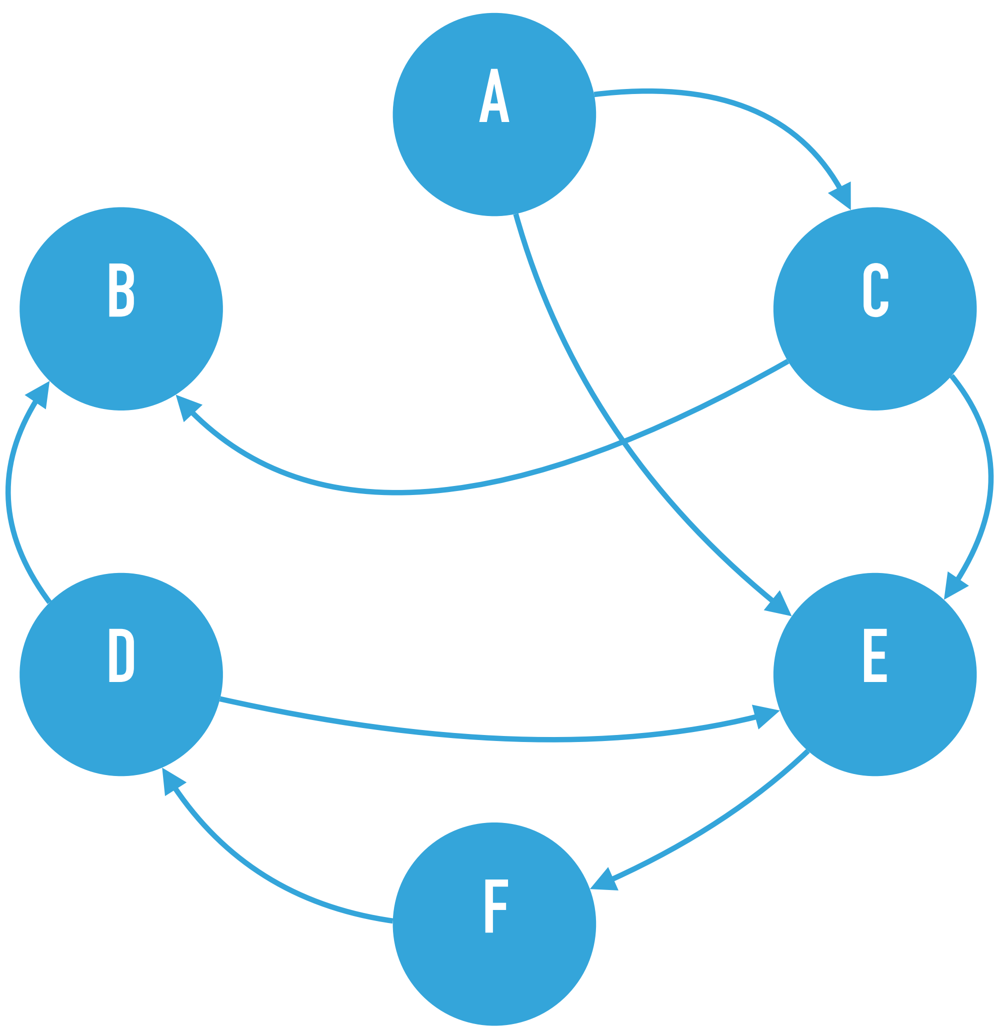


DEPTH FIRST SEARCH

- Find a path from A to B using *iterative* depth first search
  - (Assume that nodes are pushed onto the stack in *alphabetic order*)
- A ➔ E ➔ F ➔ D ➔ B
- Is this the shortest path?

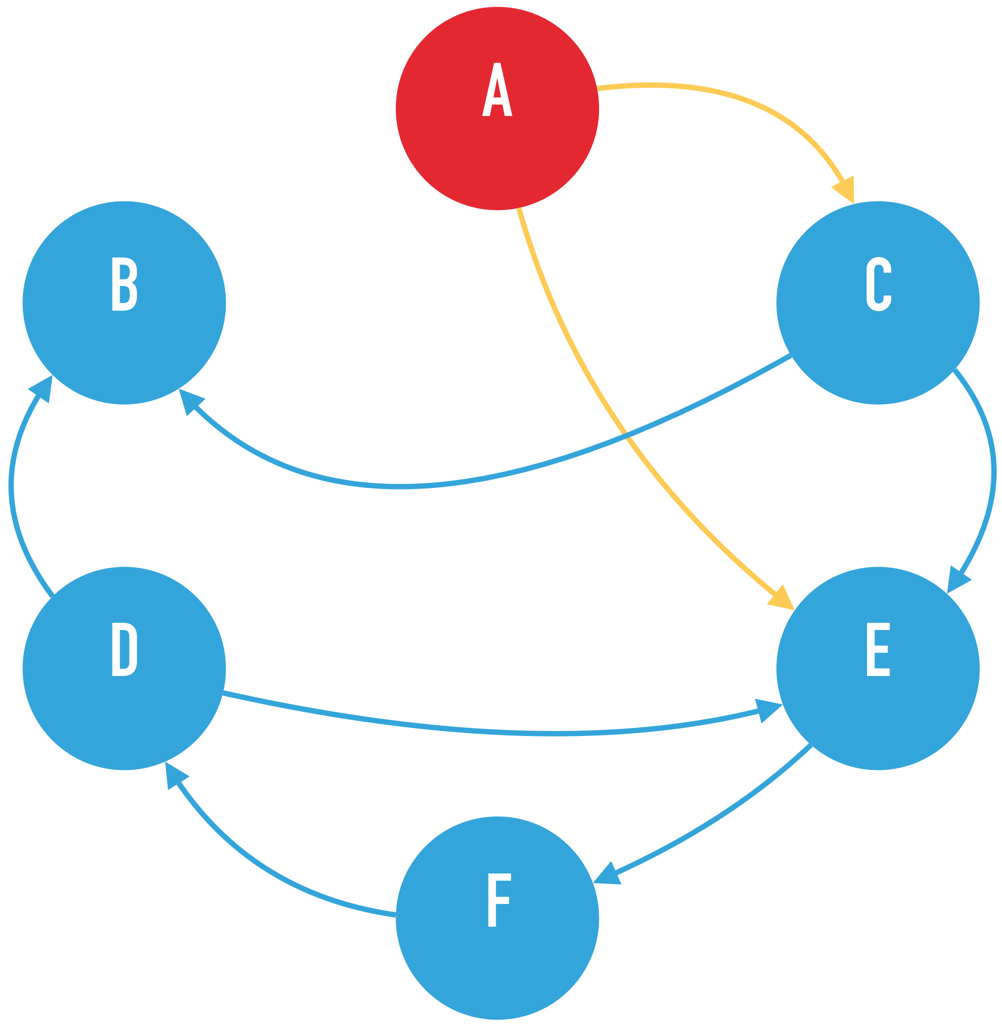


DEPTH FIRST SEARCH



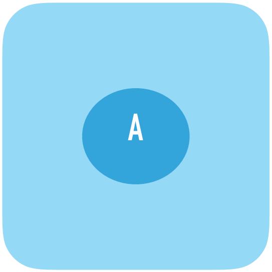
Paths to Consider (Stack)

DEPTH FIRST SEARCH

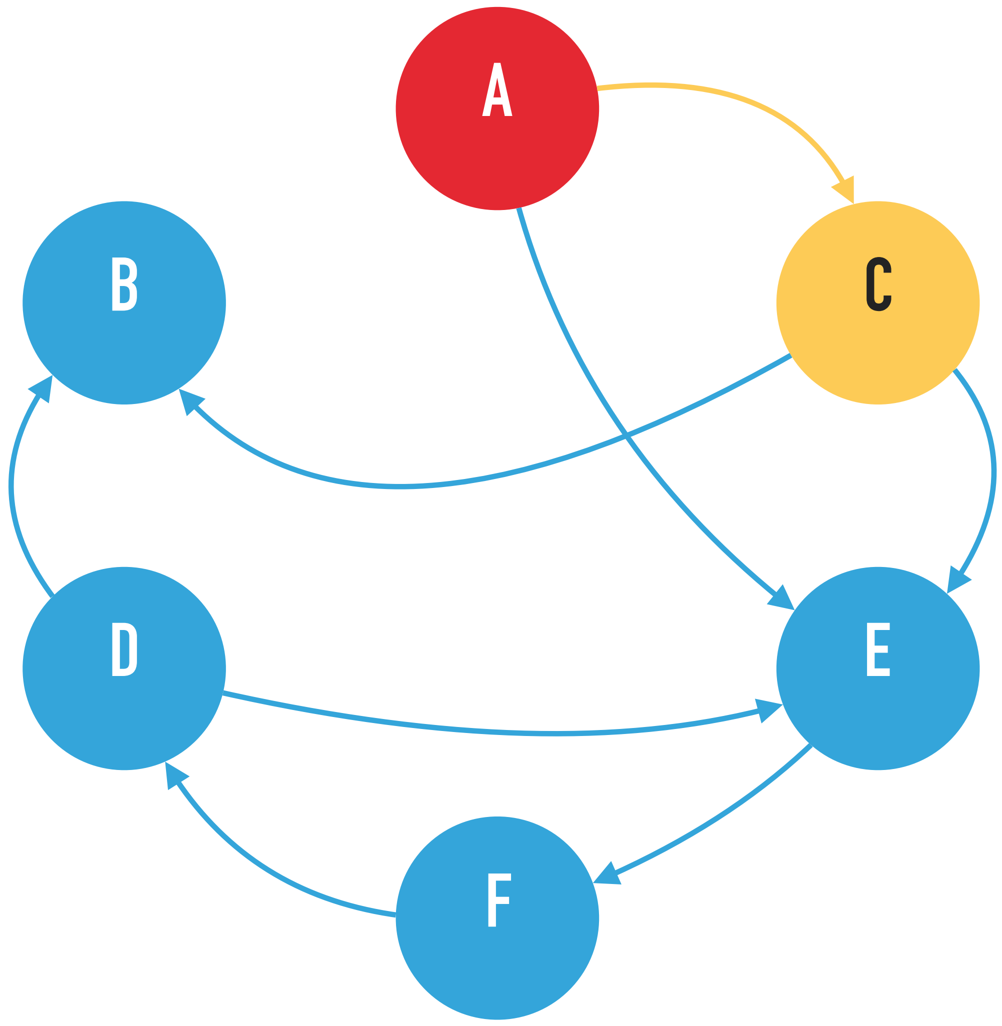


Paths to Consider (Stack)

Current Path

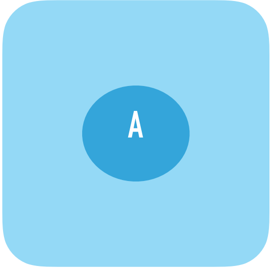


DEPTH FIRST SEARCH

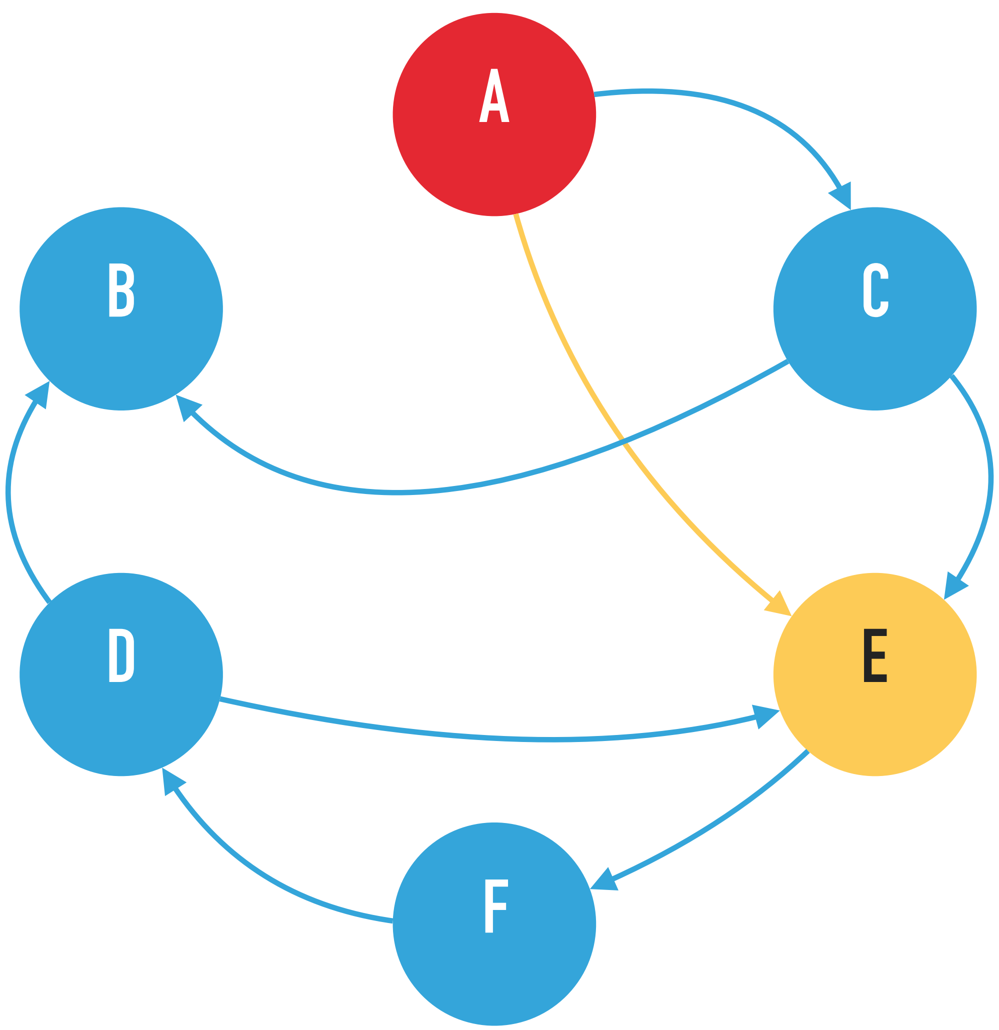


Paths to Consider (Stack)

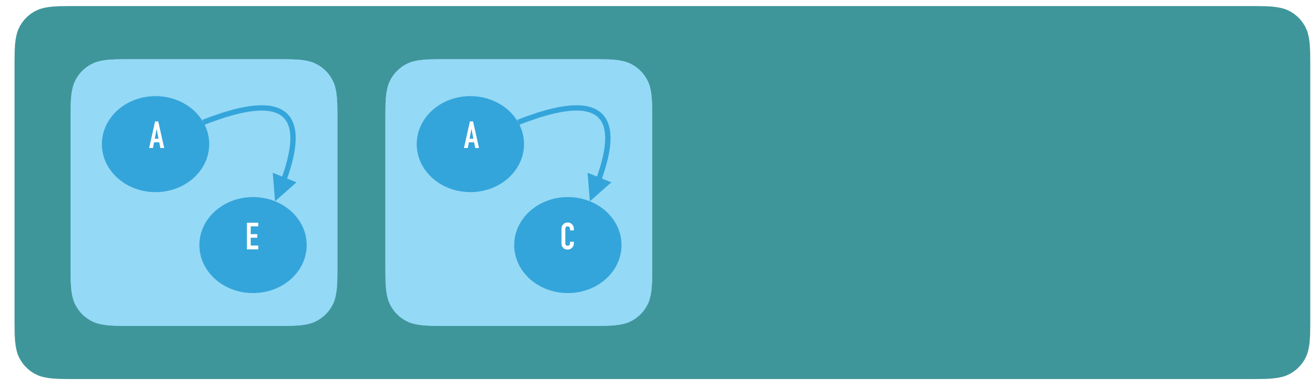
Current Path



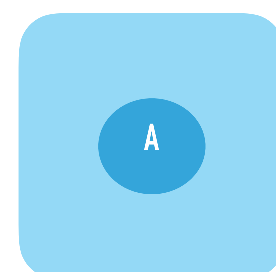
DEPTH FIRST SEARCH



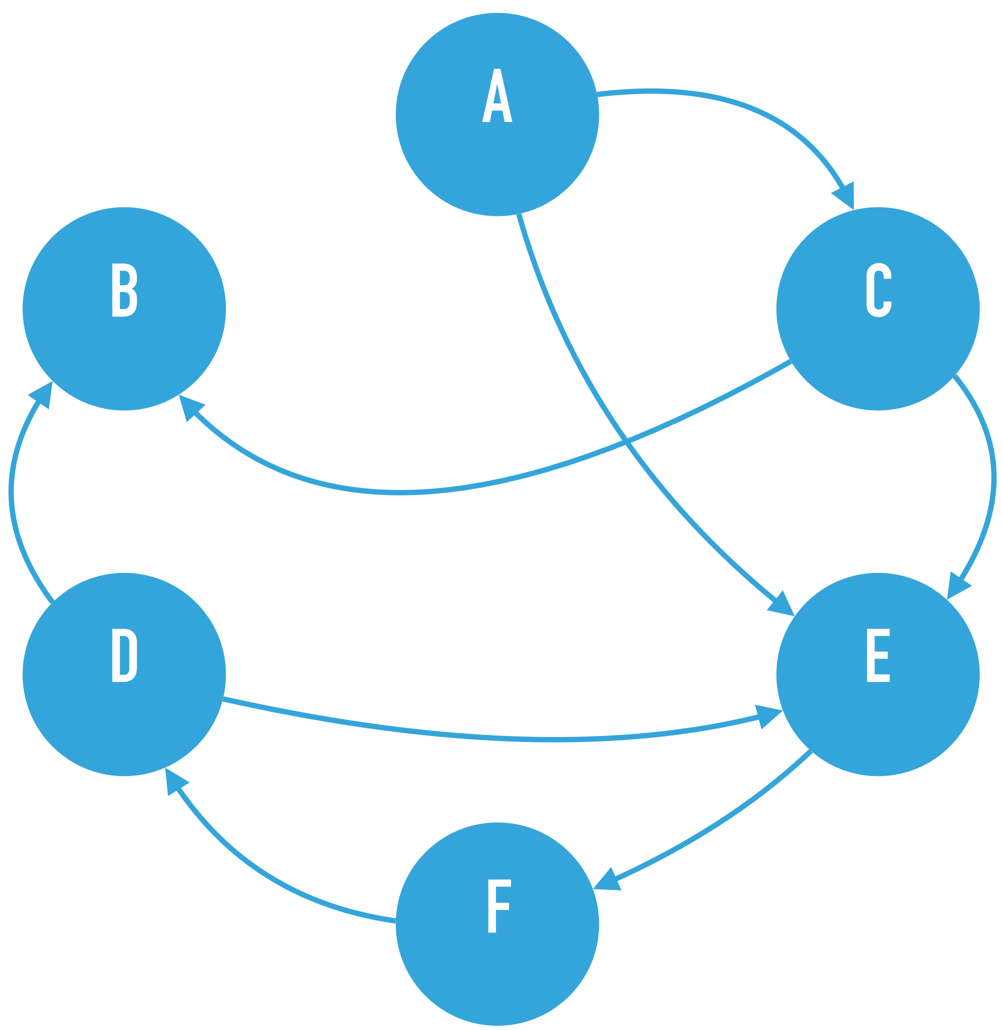
Paths to Consider (Stack)



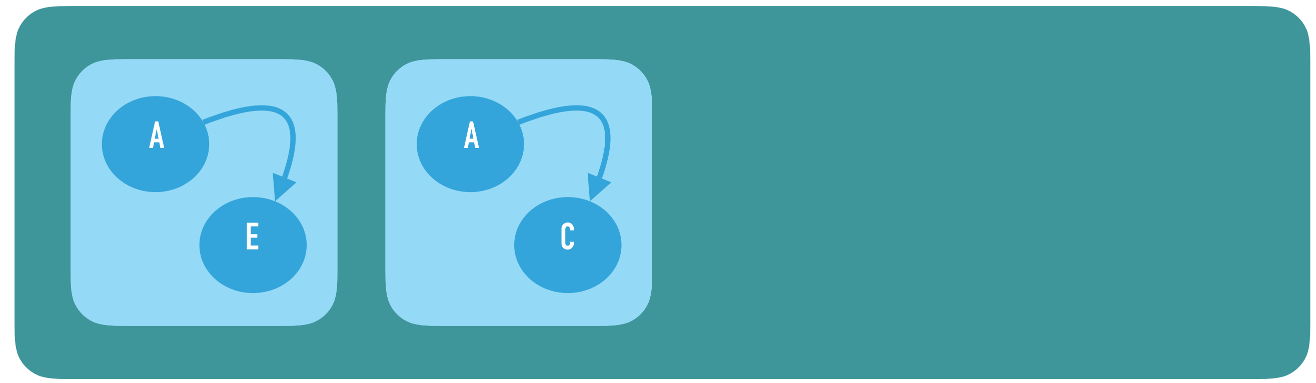
Current Path



DEPTH FIRST SEARCH

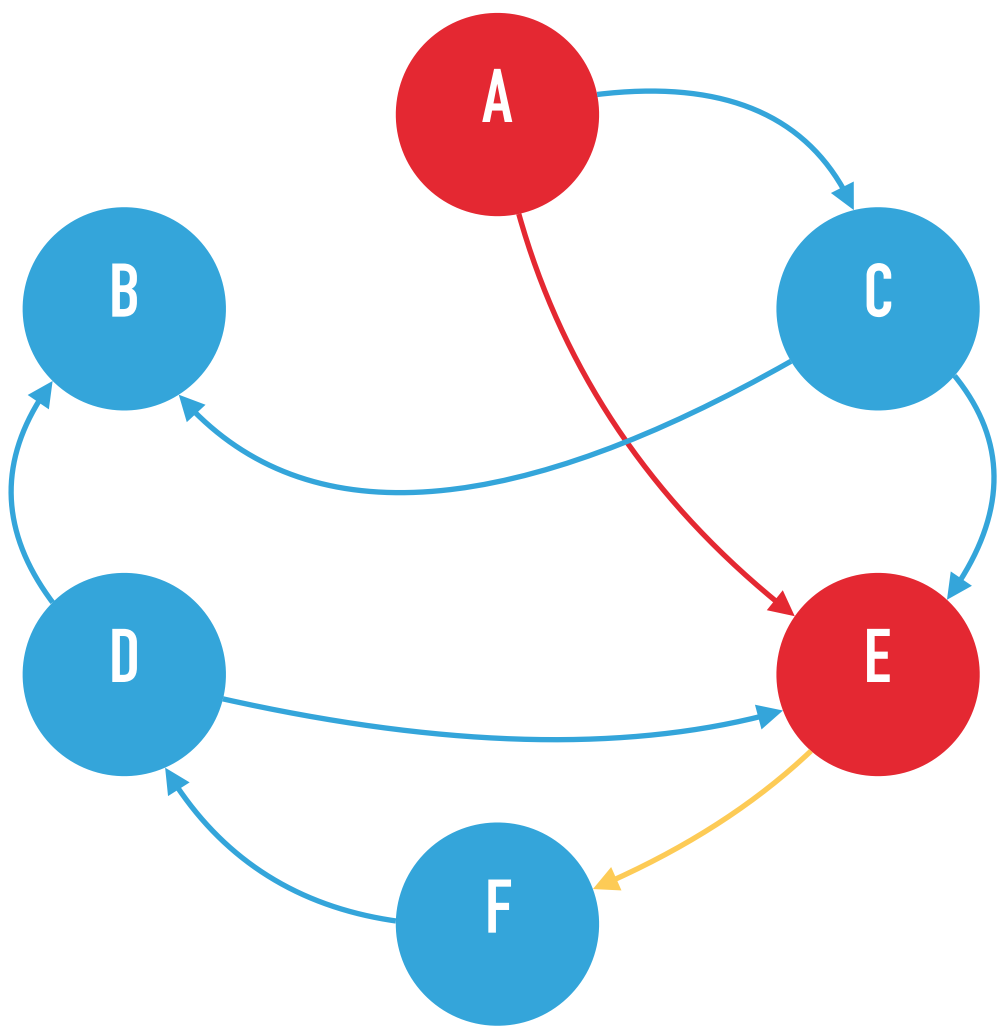


Paths to Consider (Stack)



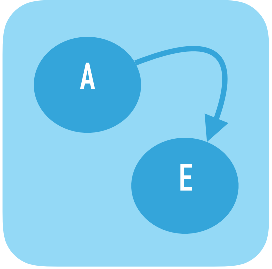
Current Path

DEPTH FIRST SEARCH

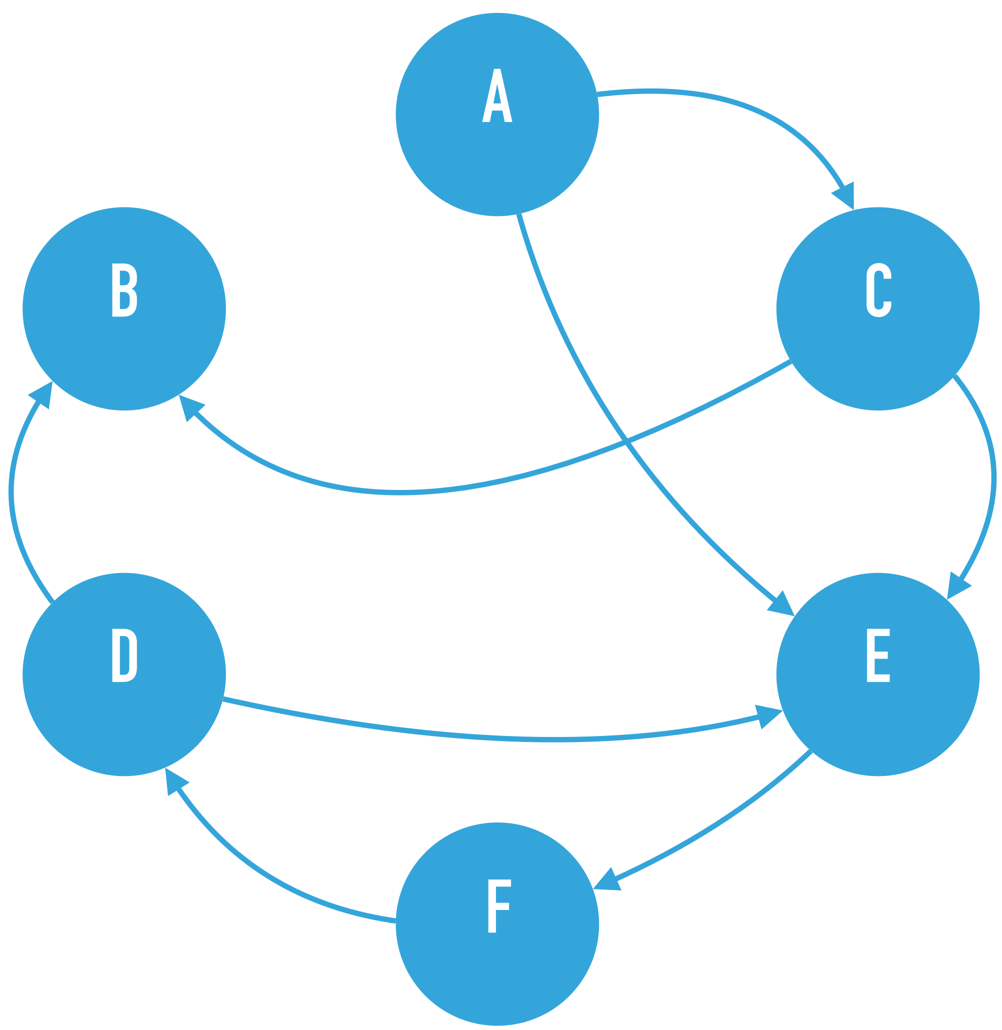


Paths to Consider (Stack)

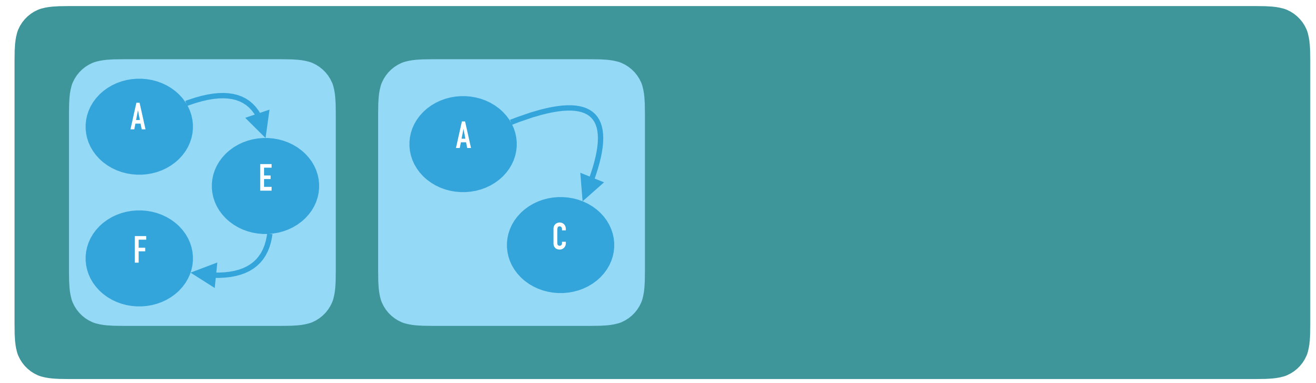
Current Path



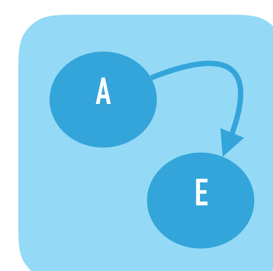
DEPTH FIRST SEARCH



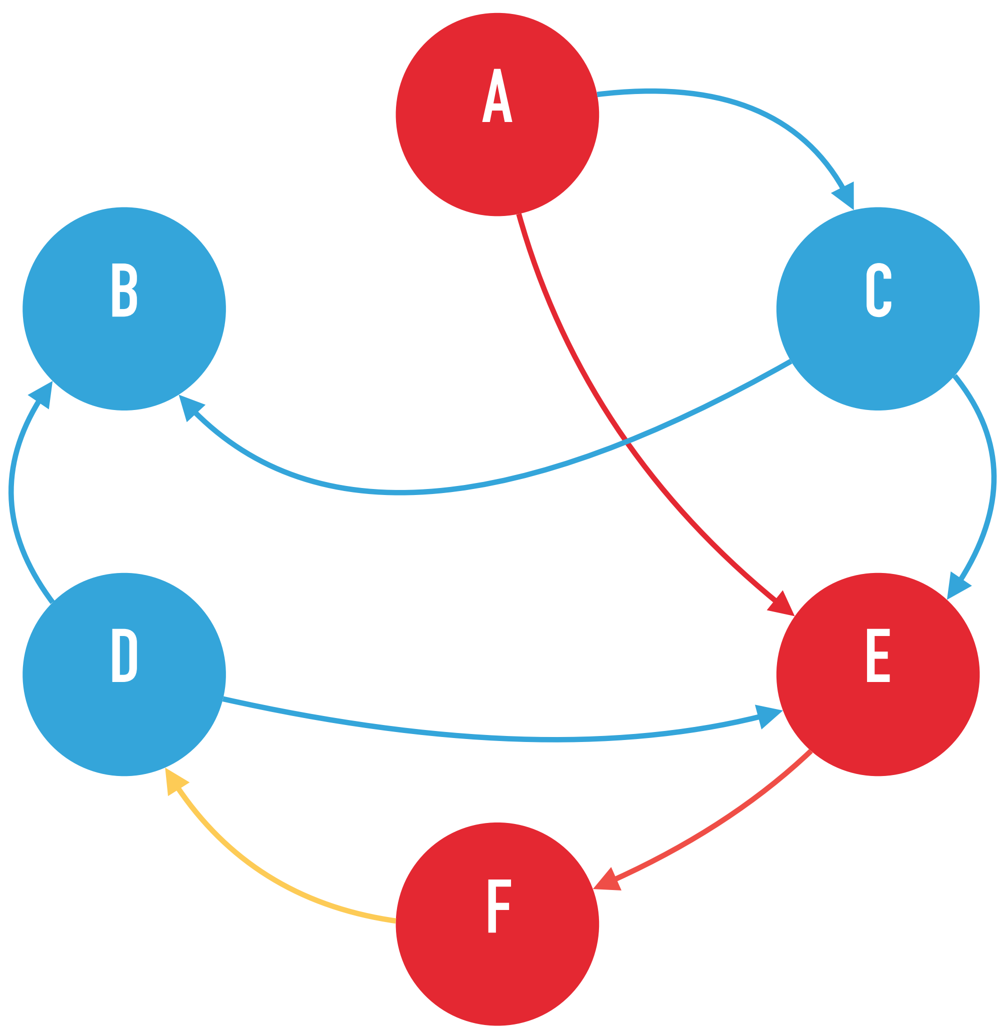
Paths to Consider (Stack)



Current Path

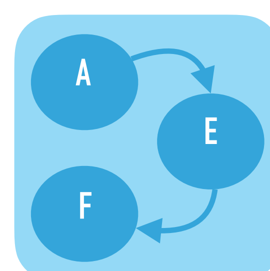


DEPTH FIRST SEARCH



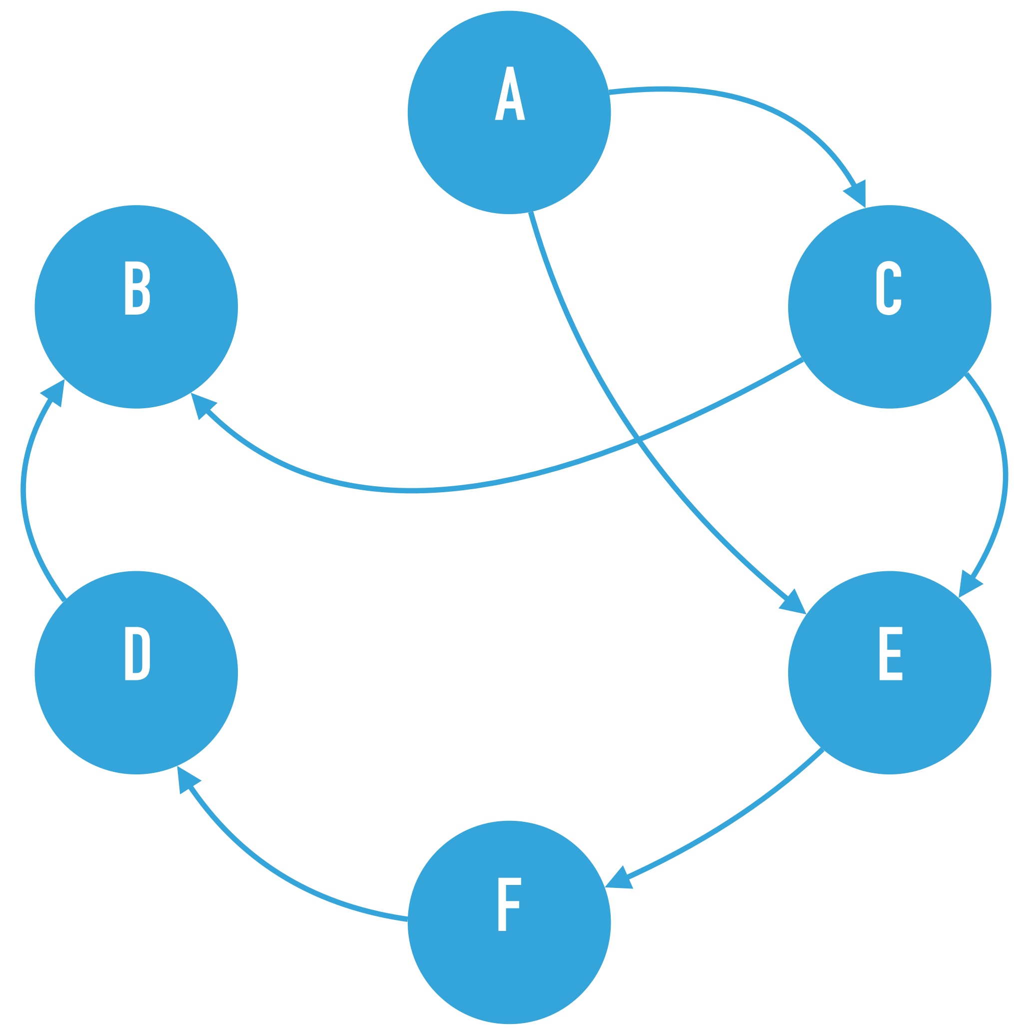
Paths to Consider (Stack)

Current Path



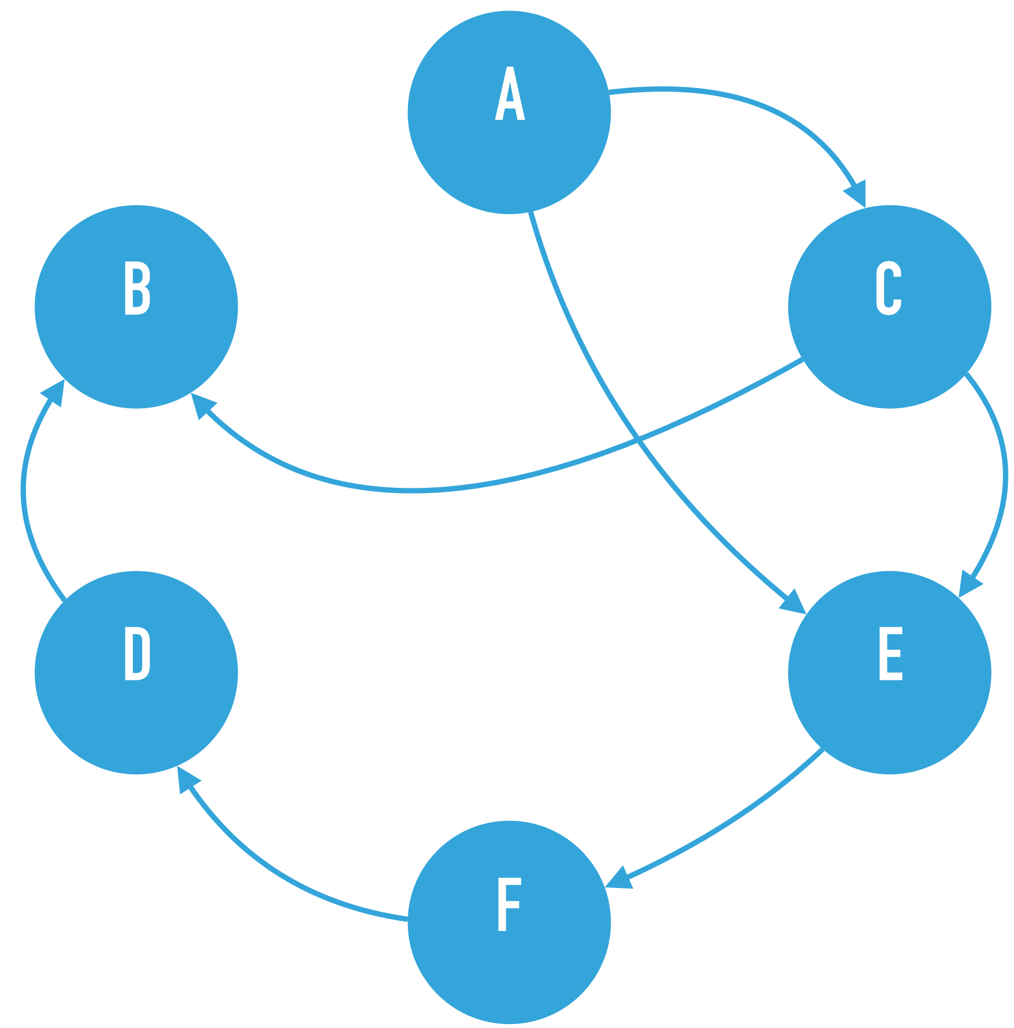
BREADTH FIRST SEARCH

- Find a path from A to B using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)



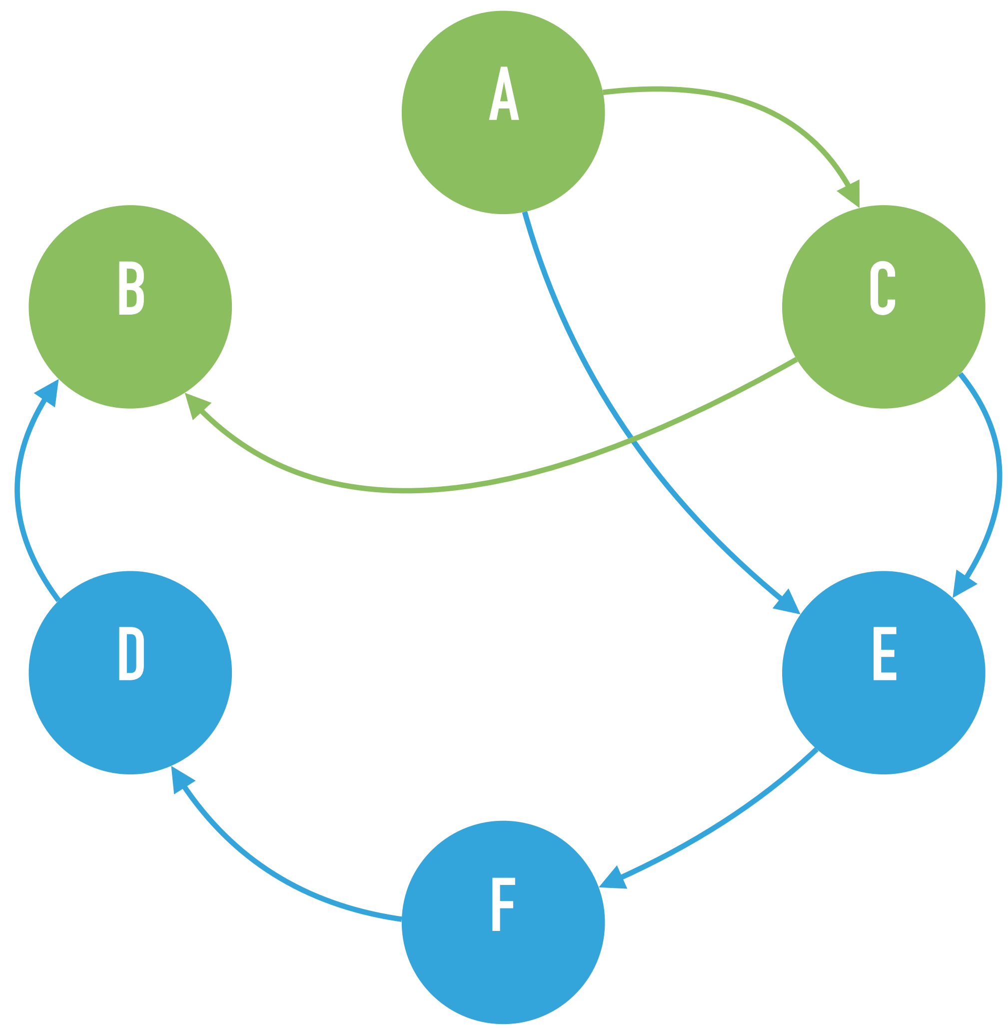
BREADTH FIRST SEARCH (PSEUDOCODE)

- create a path with just start node and enqueue into queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q



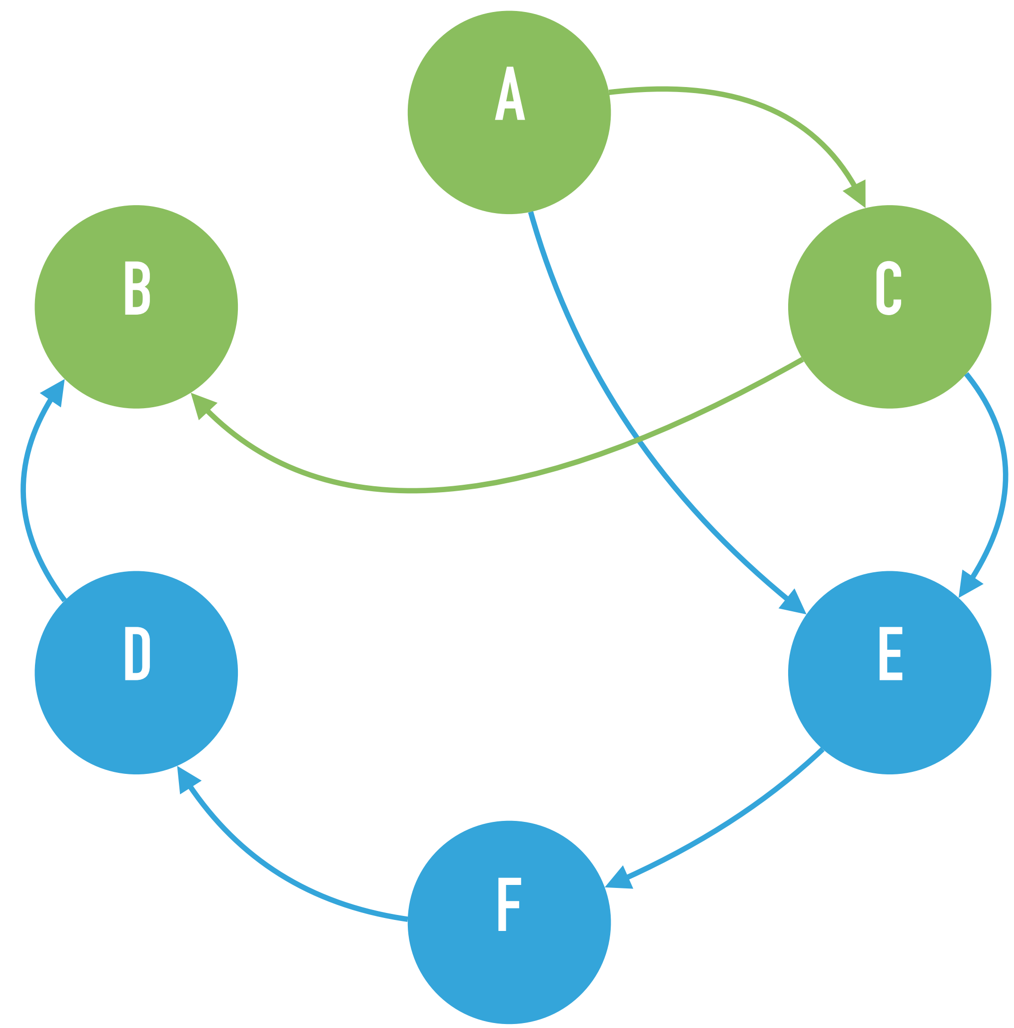
BREADTH FIRST SEARCH

- Find a path from A to F using breadth first search
    - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A ➔ C ➔ B



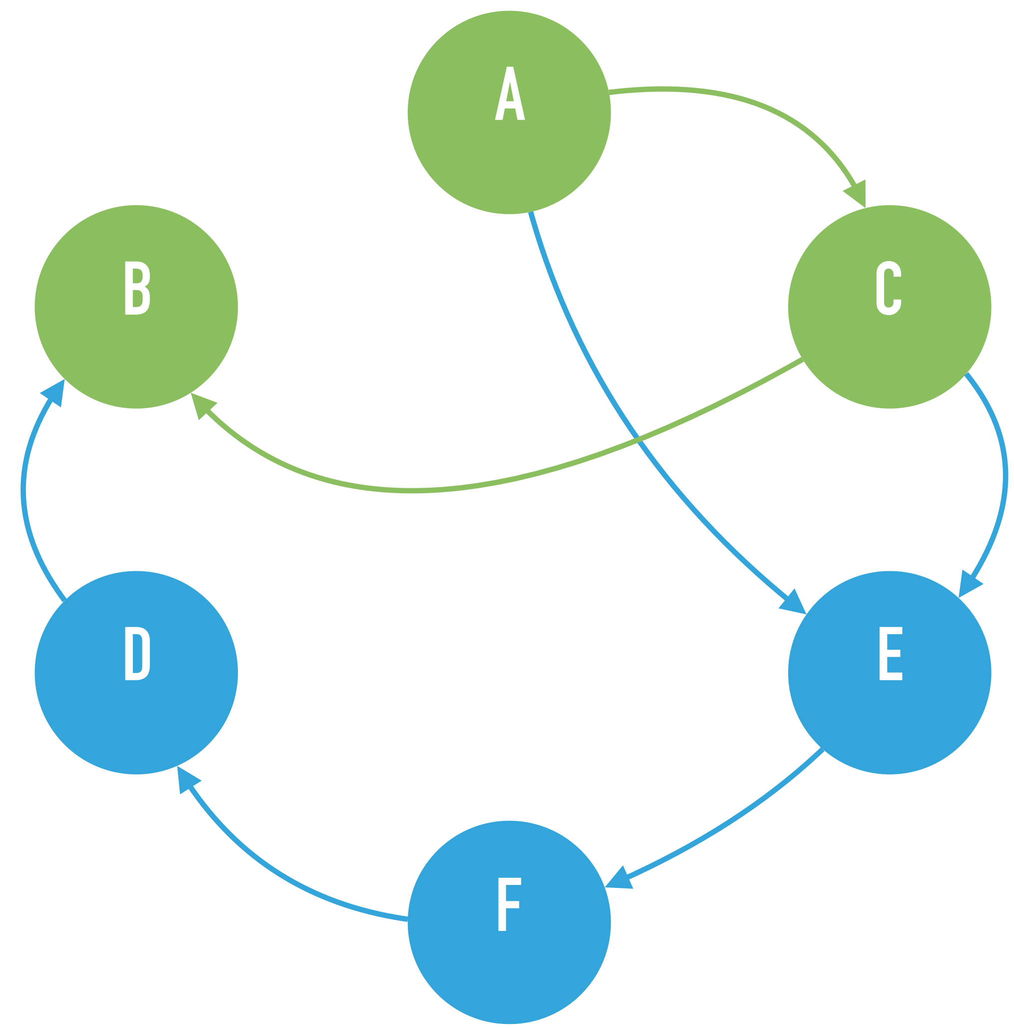
BREADTH FIRST SEARCH

- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A ➔ C ➔ B
- Is *this* the shortest path?

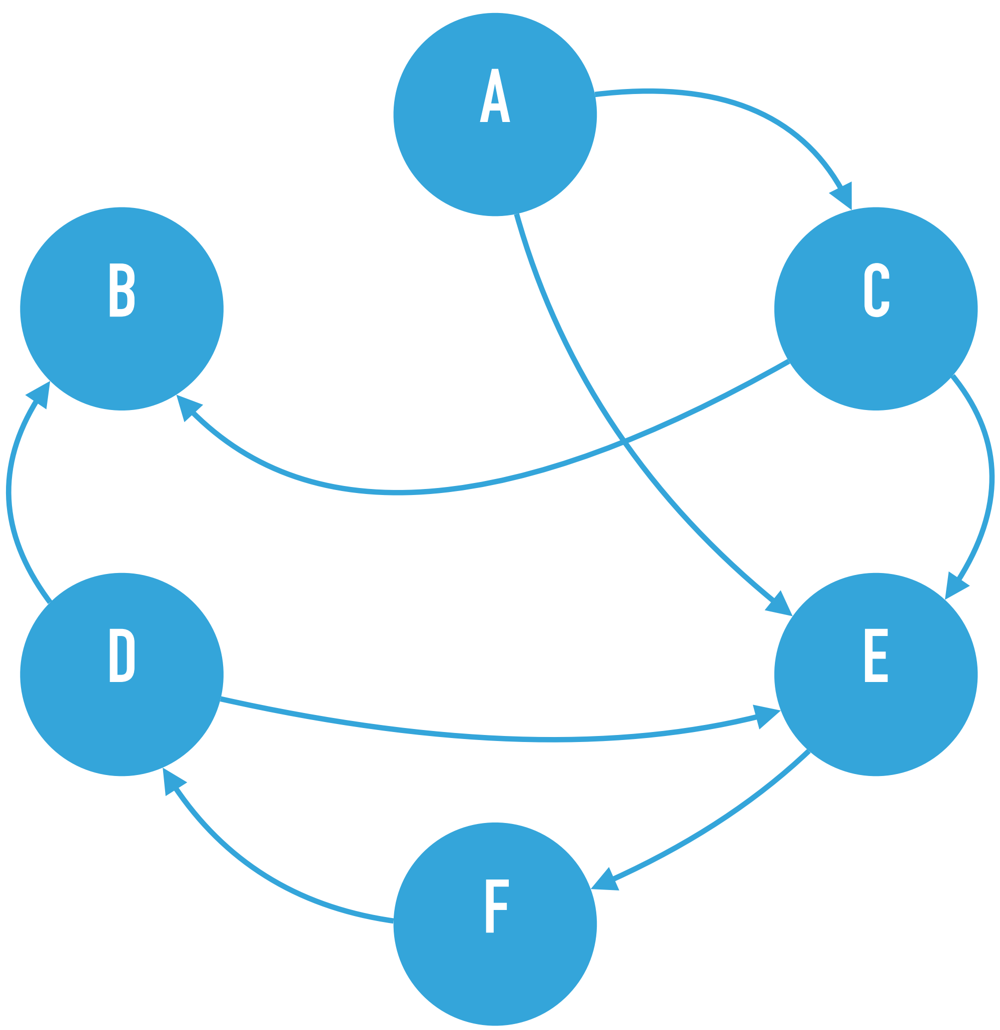


BREADTH FIRST SEARCH

- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A ➔ C ➔ B
- Is *this* the shortest path?
  - Yes



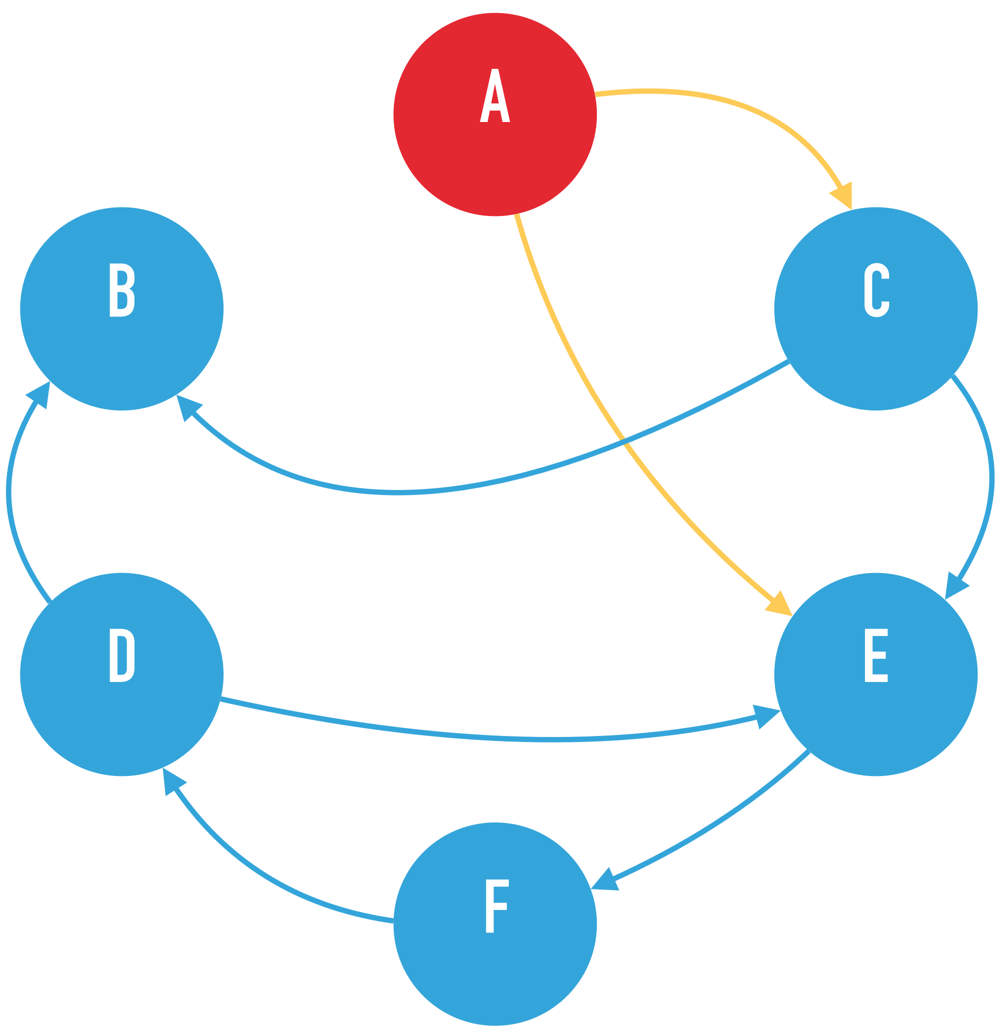
BREADTH FIRST SEARCH



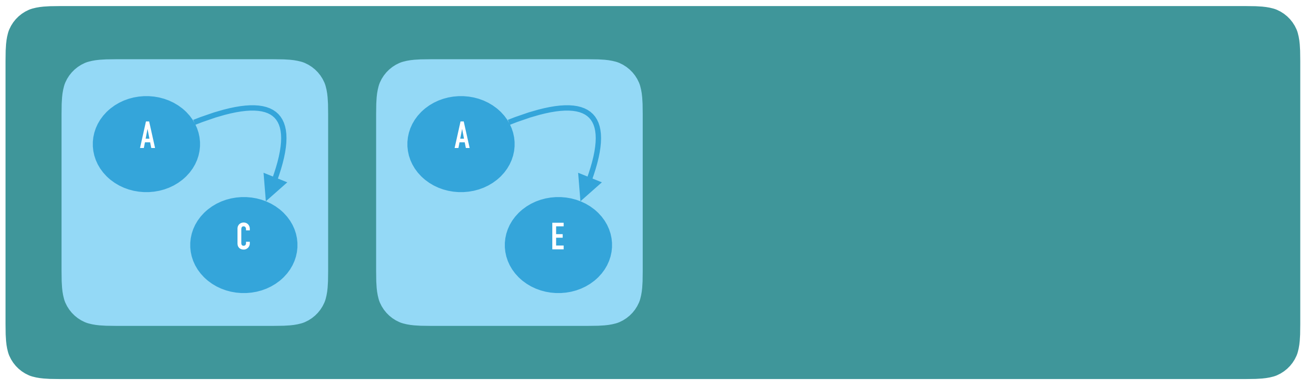
Paths to Consider (Queue)

Current Path

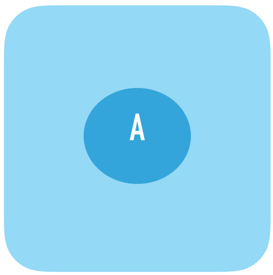
BREADTH FIRST SEARCH



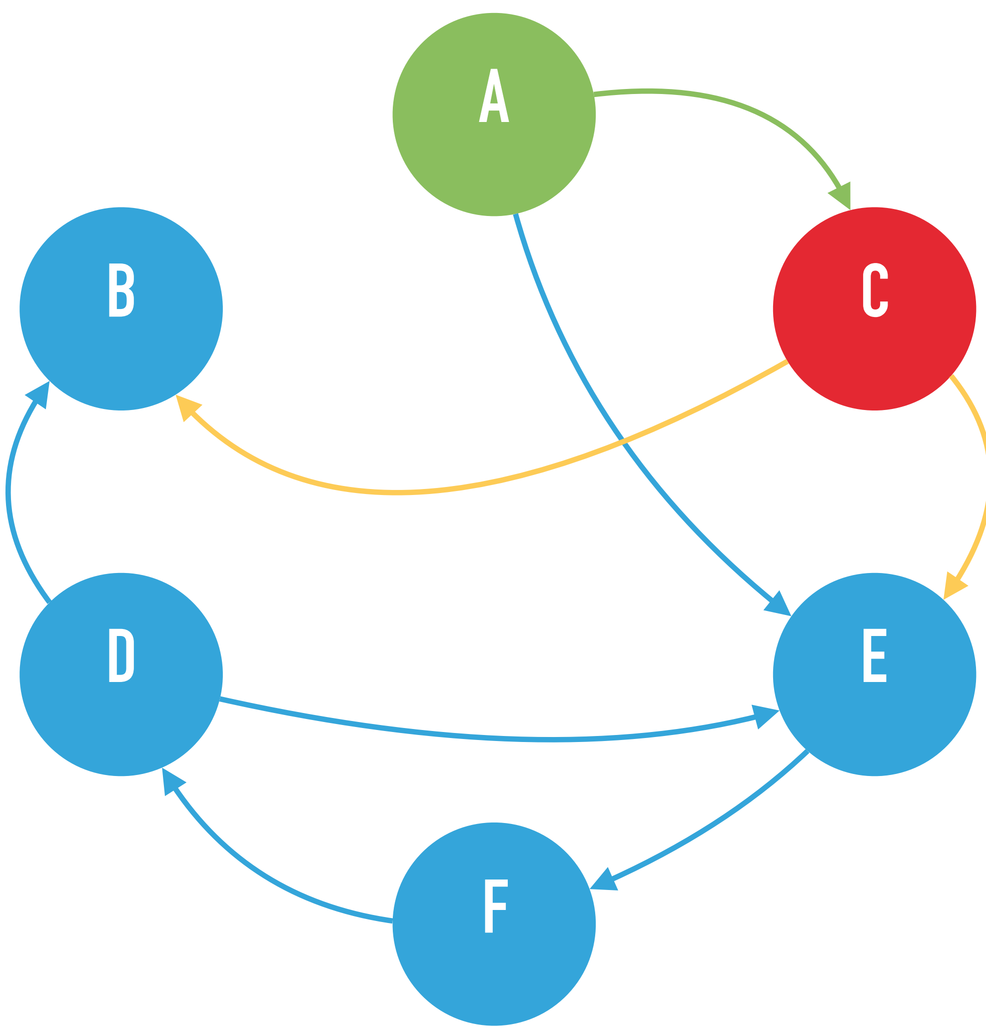
Paths to Consider (Queue)



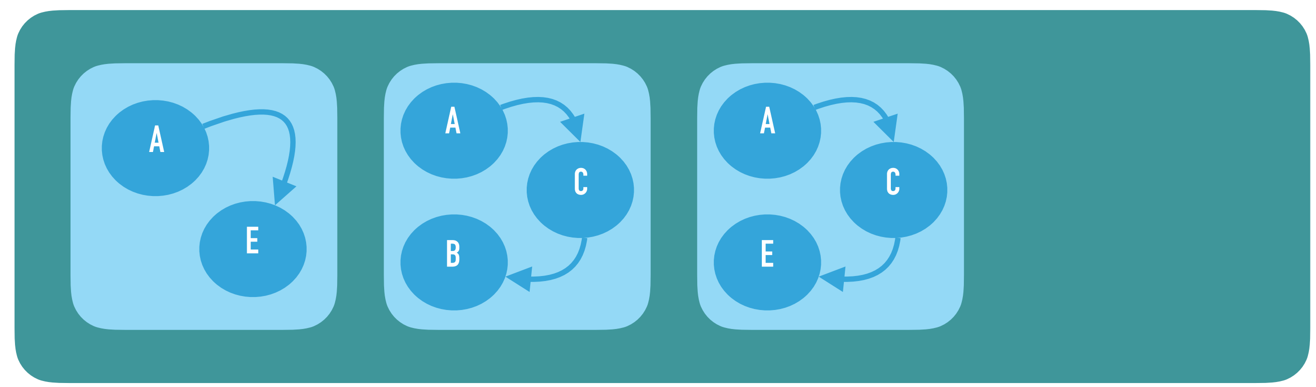
Current Path



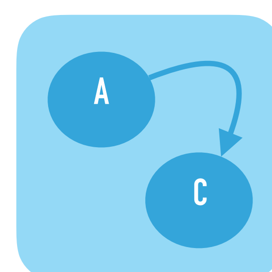
BREADTH FIRST SEARCH



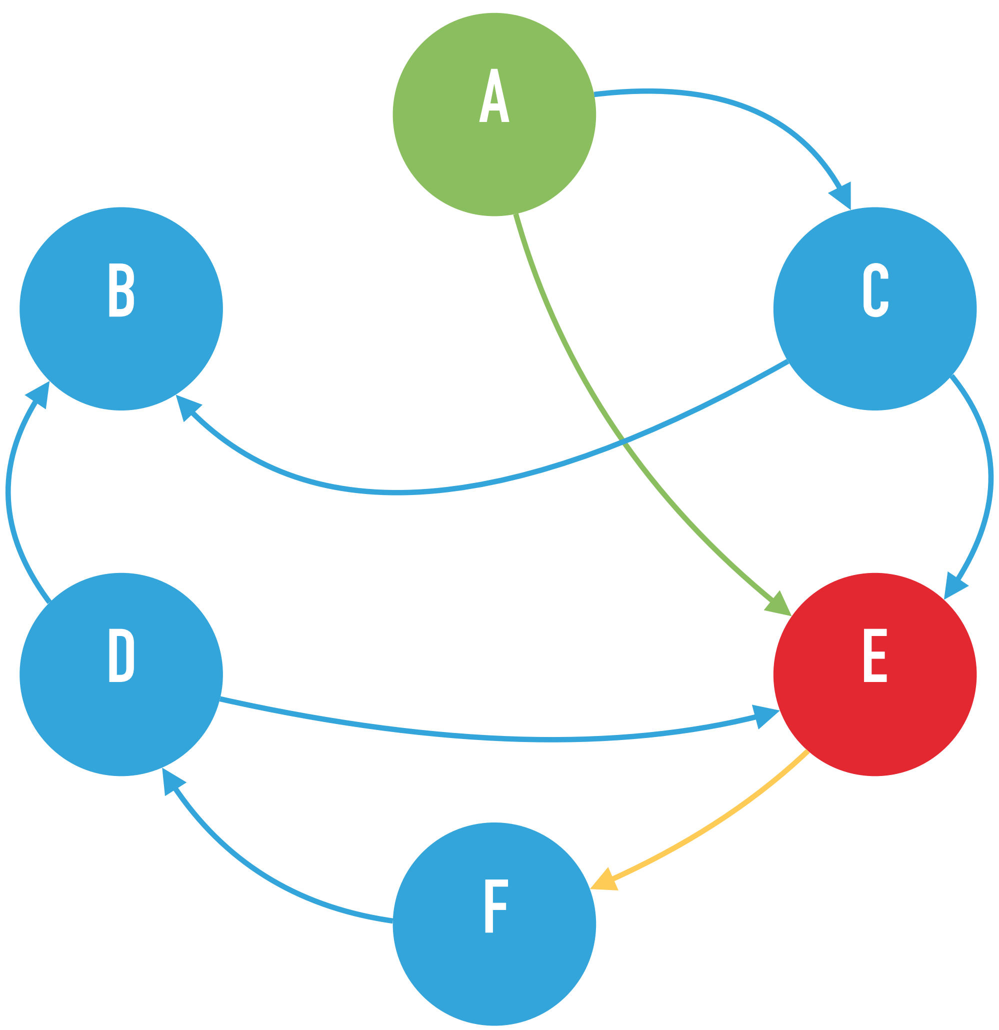
Paths to Consider (Queue)



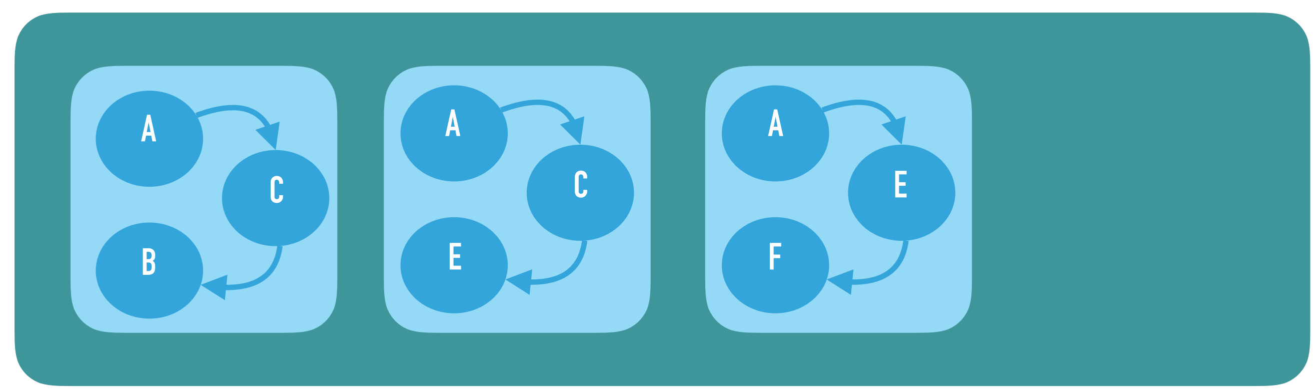
Current Path



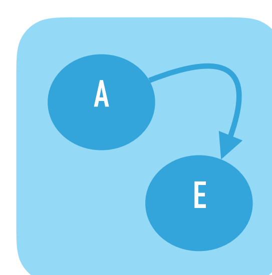
BREADTH FIRST SEARCH



Paths to Consider (Queue)



Current Path



YOU NEVER CONSIDER A PATH OF LENGTH K $+ 1$

UNTIL YOU'VE CONSIDERED ALL PATHS OF LENGTH K OR SHORTER

COMPARING DFS AND BFS

COMPARING DFS AND BFS

DFS

- create a path with just start node and push onto stack s
- while s is not empty:
  - p = s.pop()
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto s

BFS

- create a path with just start node and enqueue into queue q
- while q is not empty:
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q

COMPARING DFS AND BFS

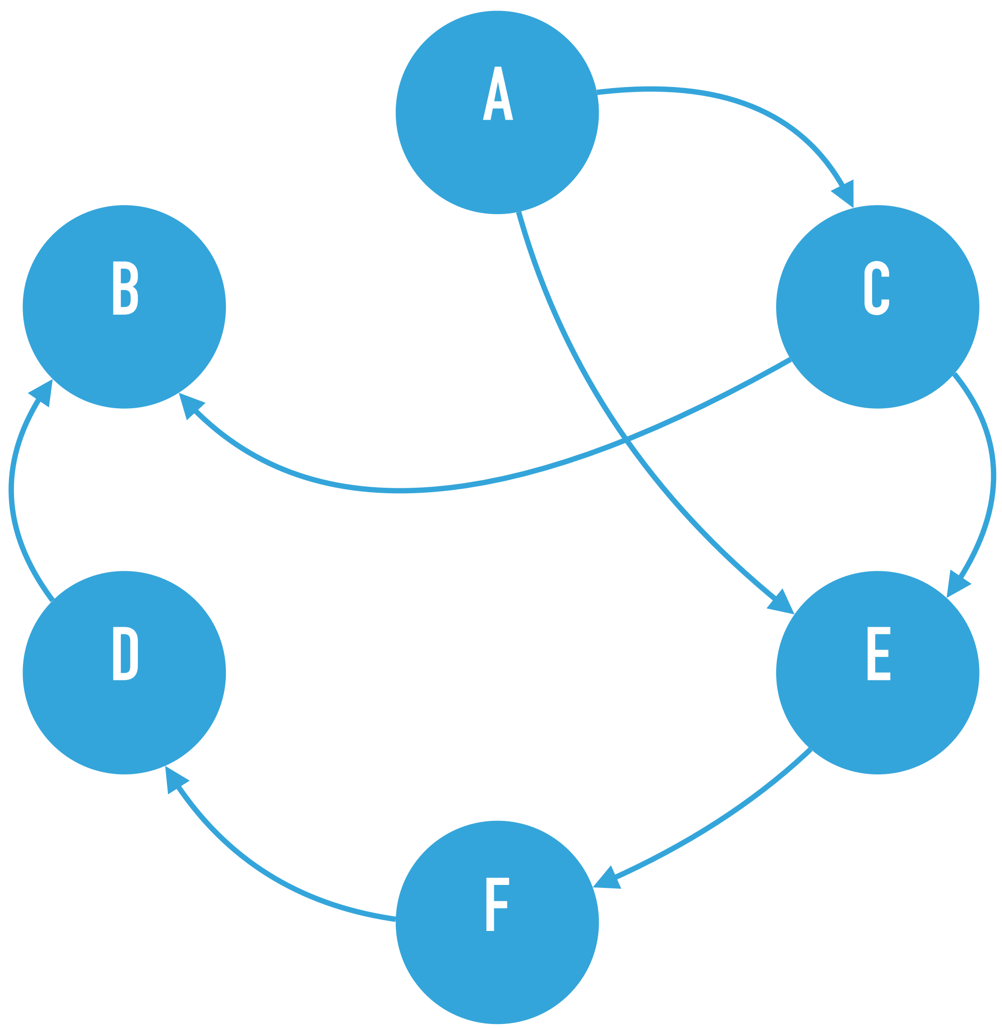
DFS

- create a path with just start node and push onto **stack s**
- while **s** is not empty:
  - p = **s.pop()**
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - **push new path onto s**

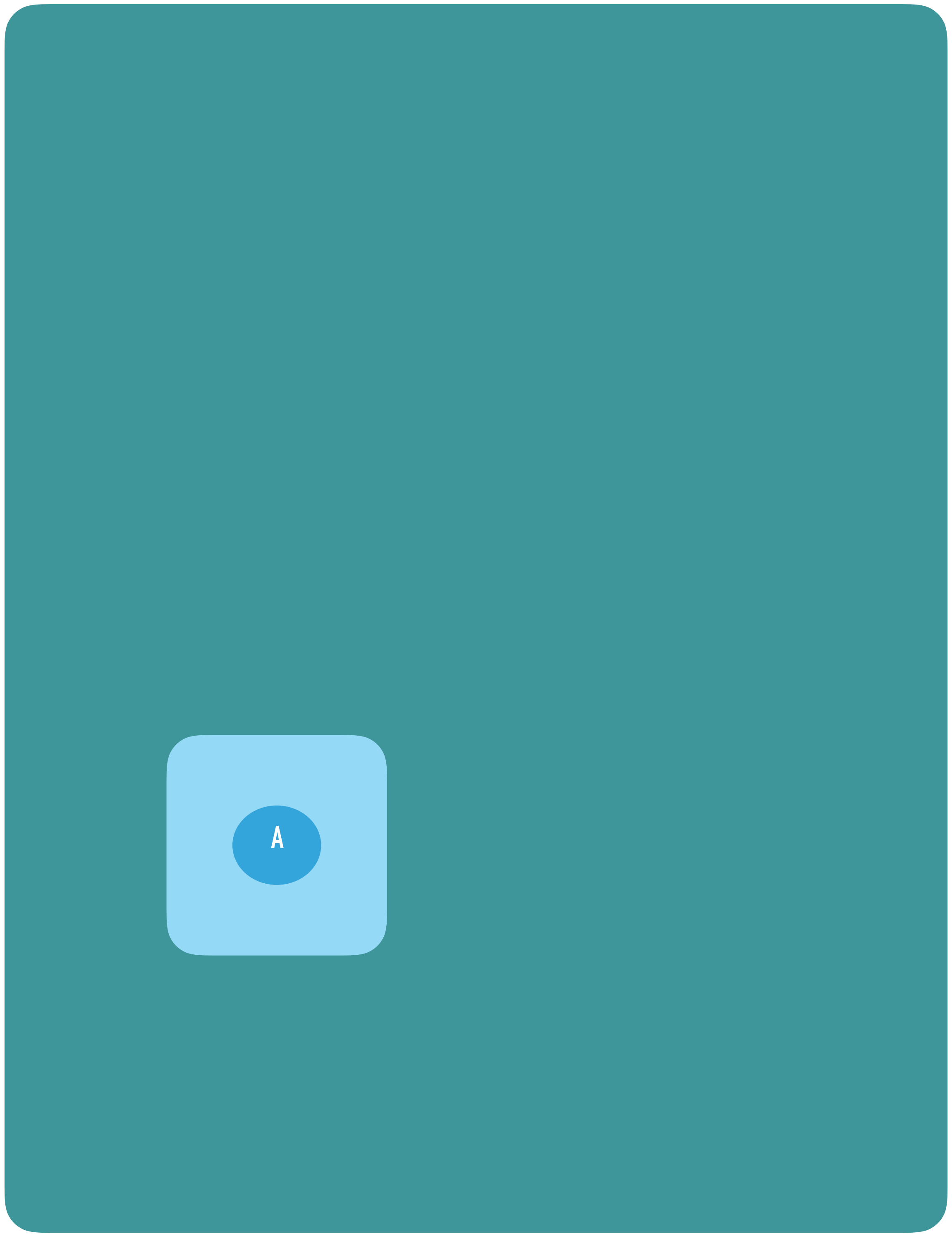
BFS

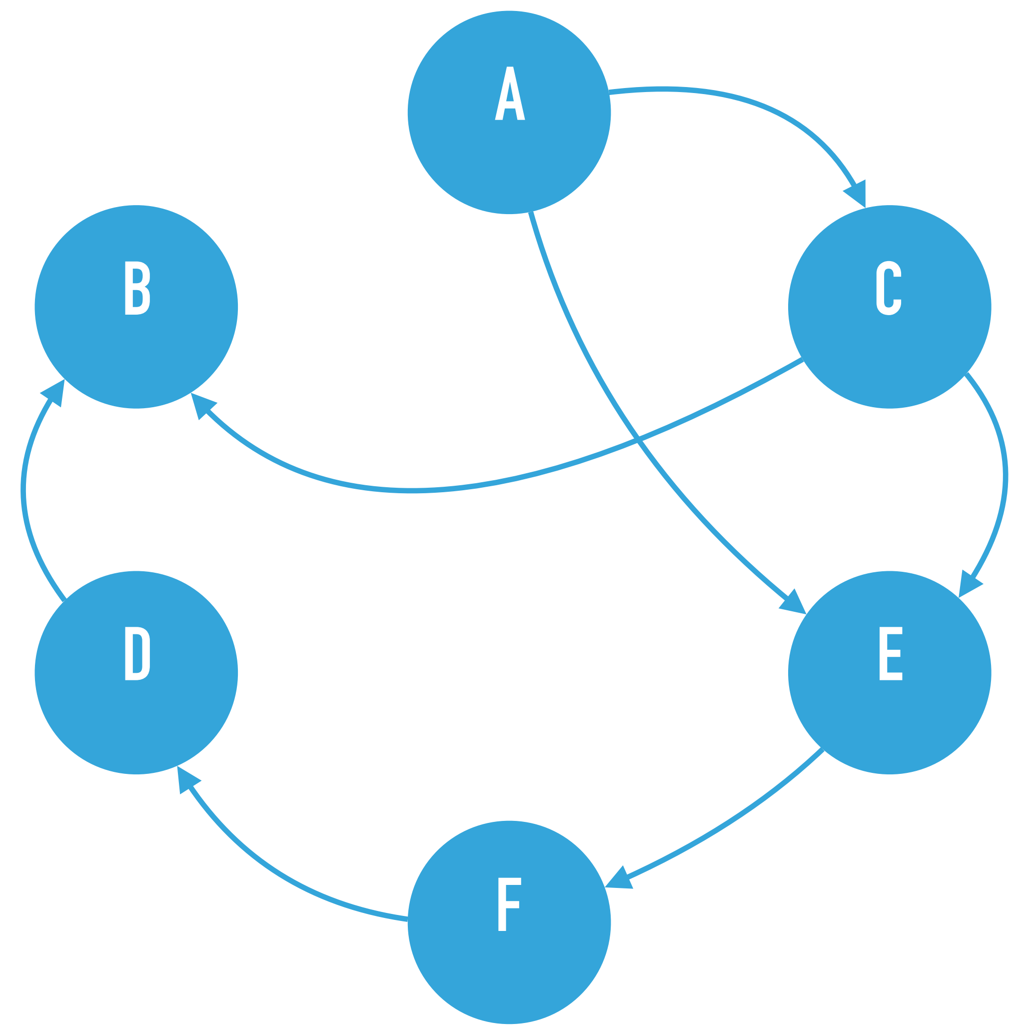
- create a path with just start node and enqueue into **queue q**
- while **q** is not empty:
  - p = **q.dequeue()**
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - **enqueue new path into q**

THE GRAPH SEARCH TO-DO LIST

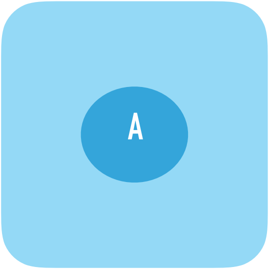


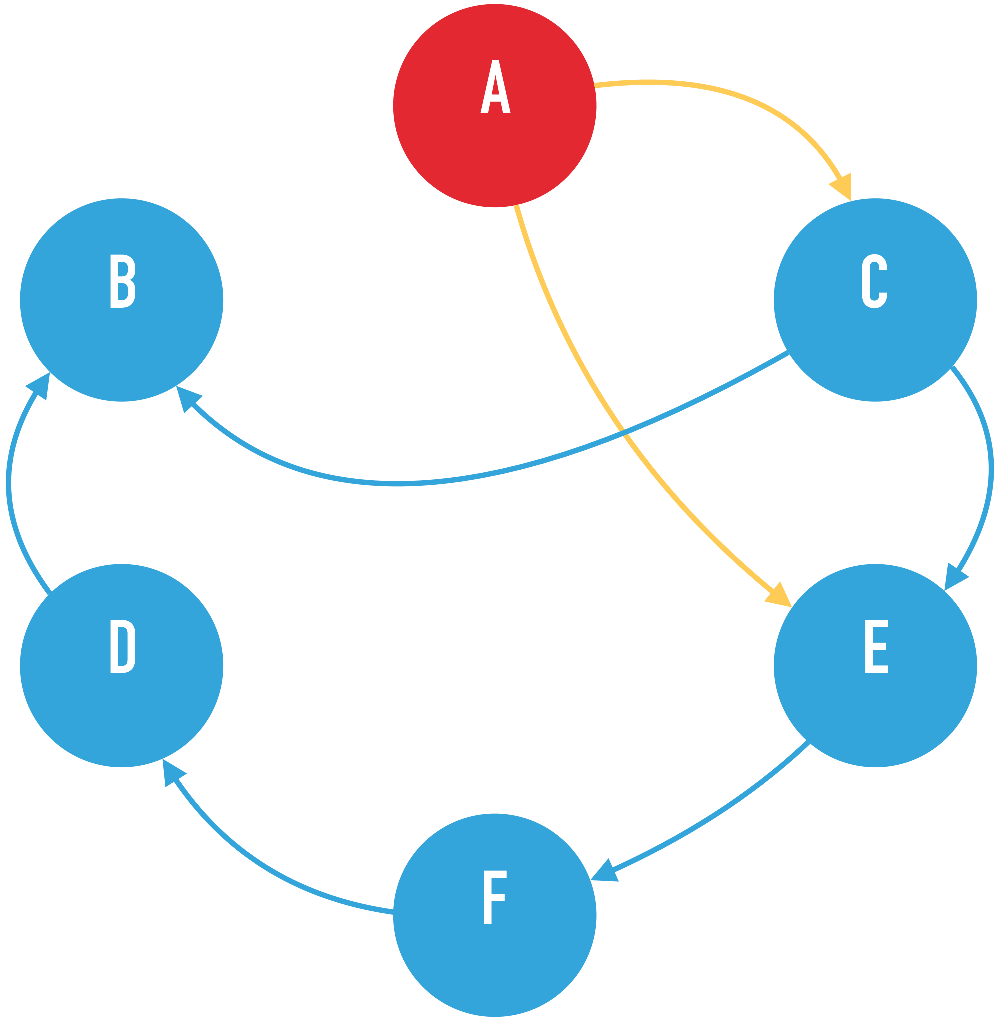
THE GRAPH SEARCH TO-DO LIST



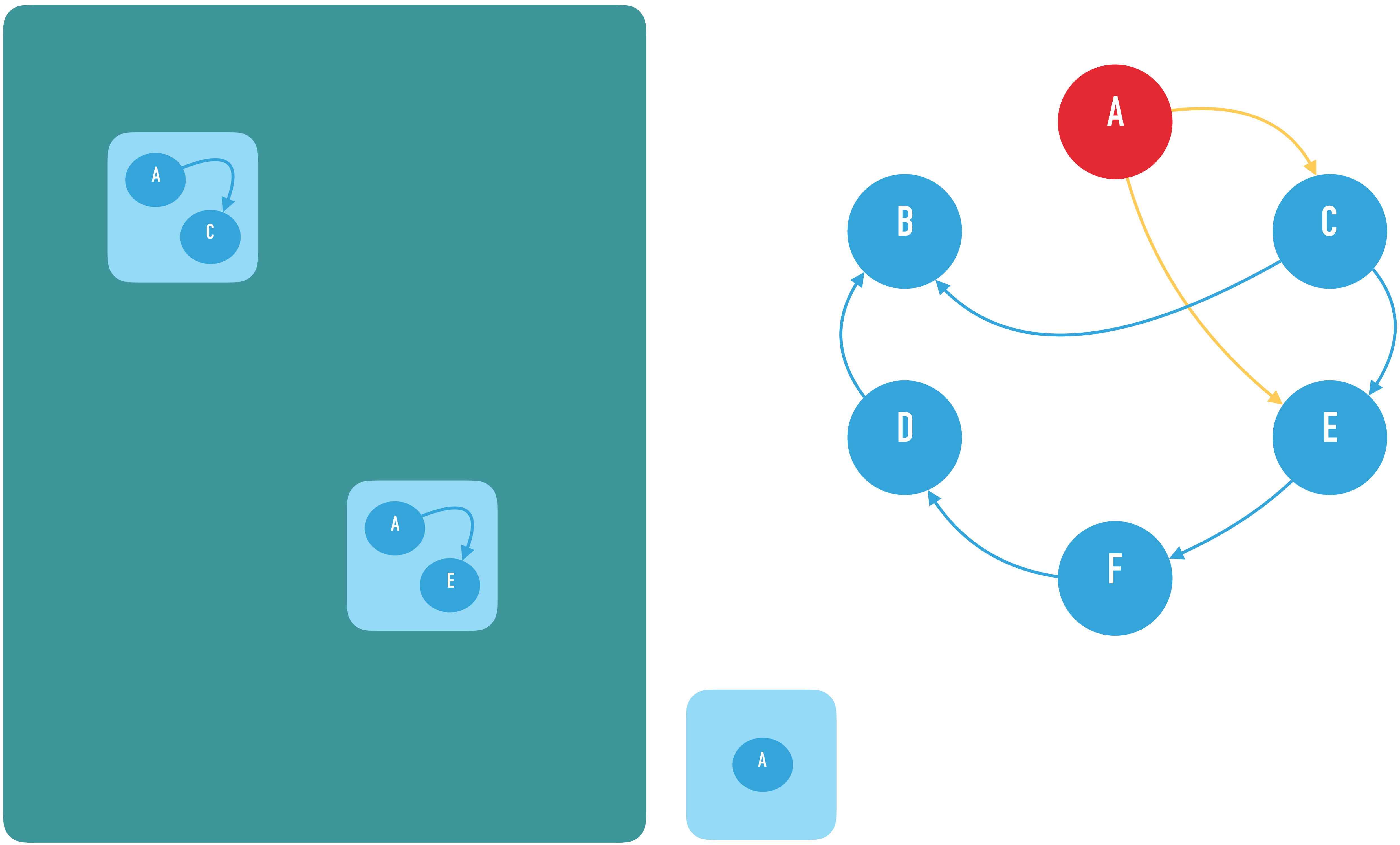


THE GRAPH SEARCH TO-DO LIST

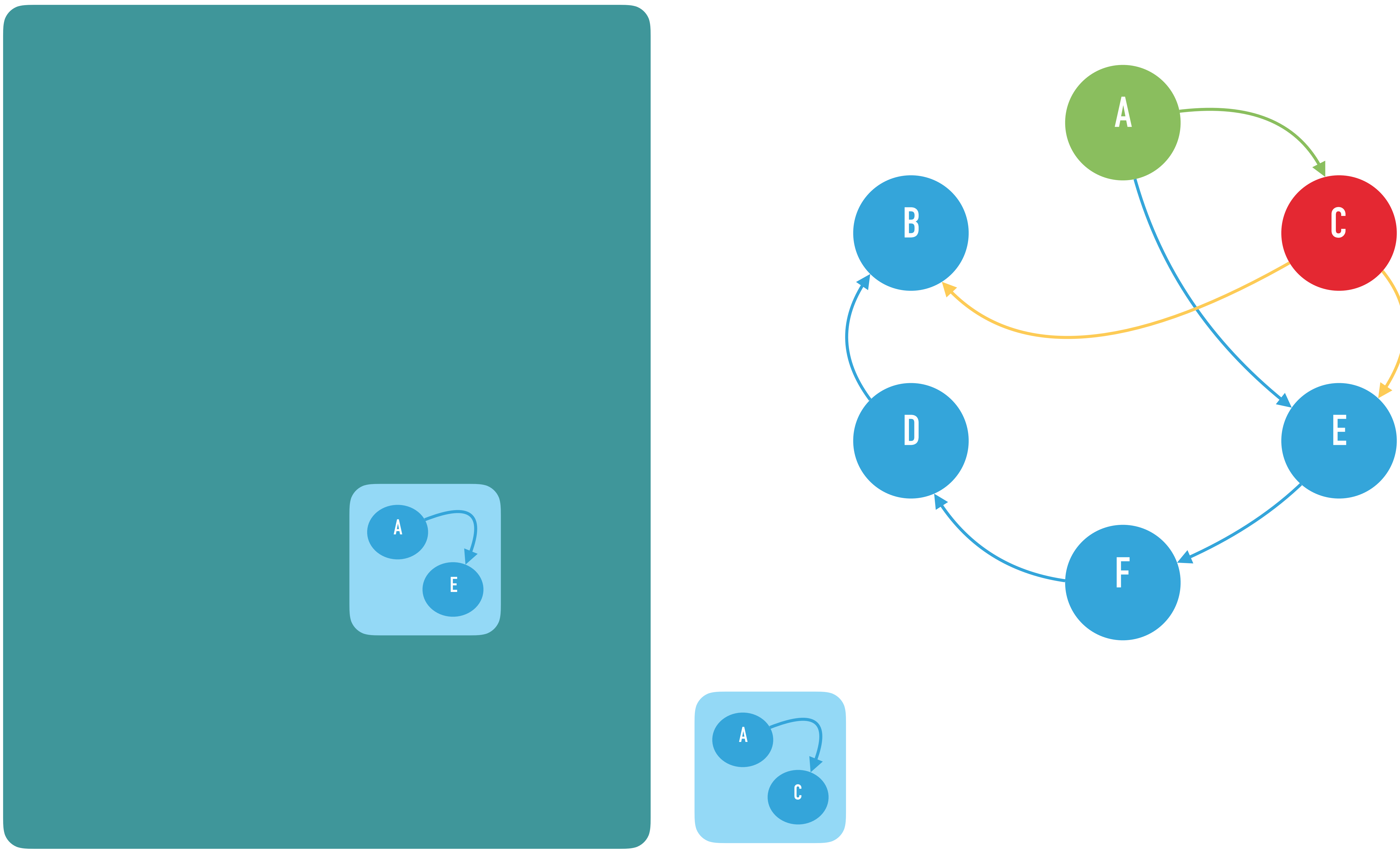




THE GRAPH SEARCH TO-DO LIST

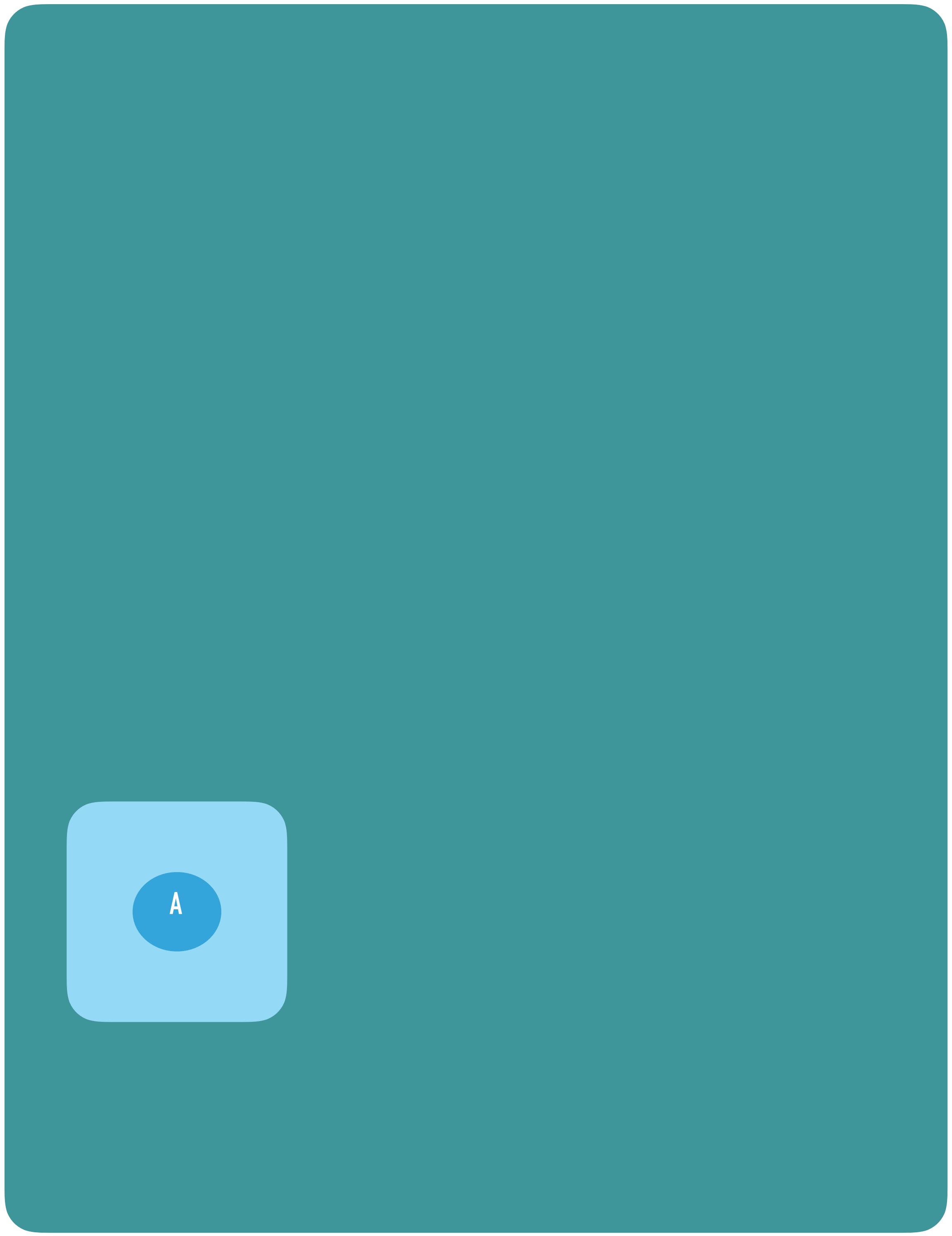


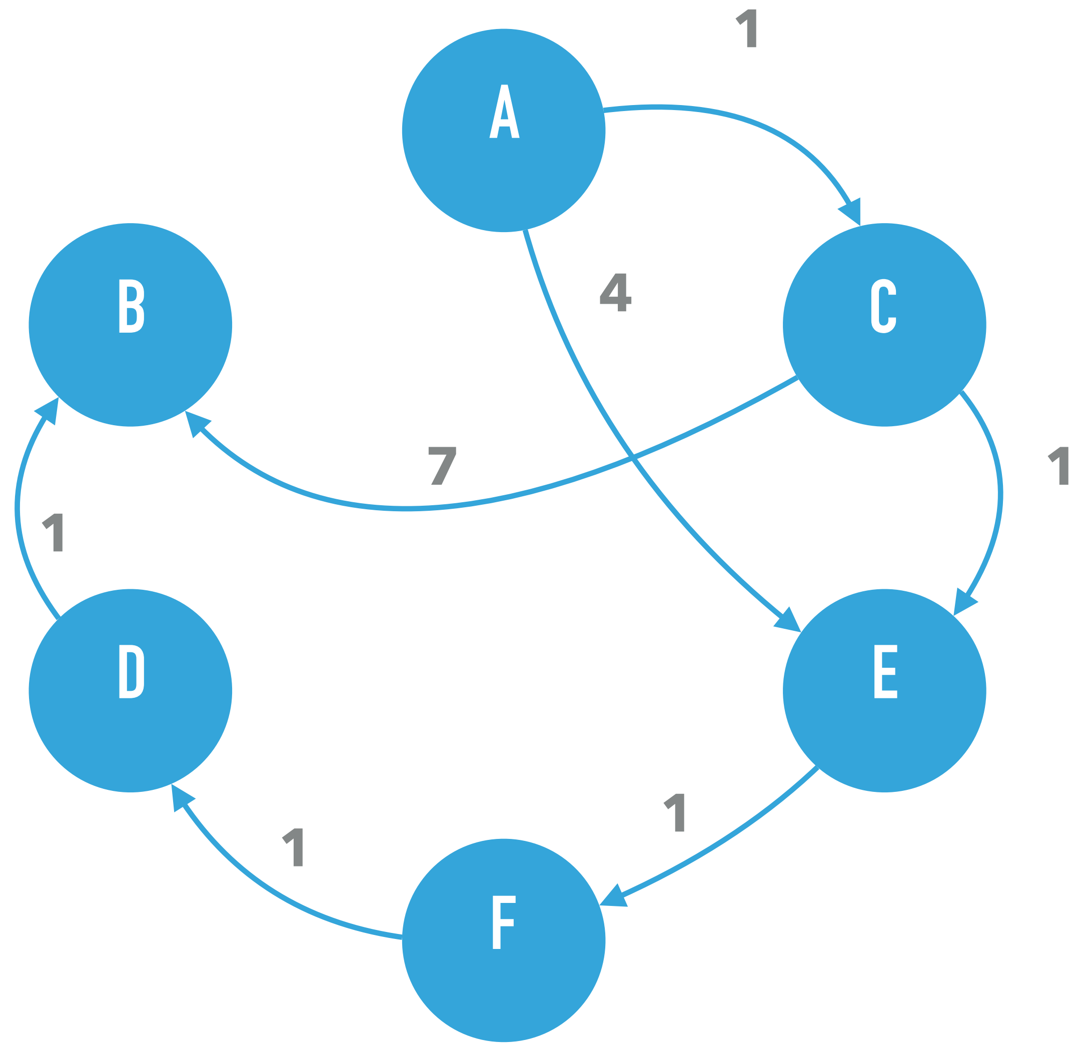
THE GRAPH SEARCH TO-DO LIST



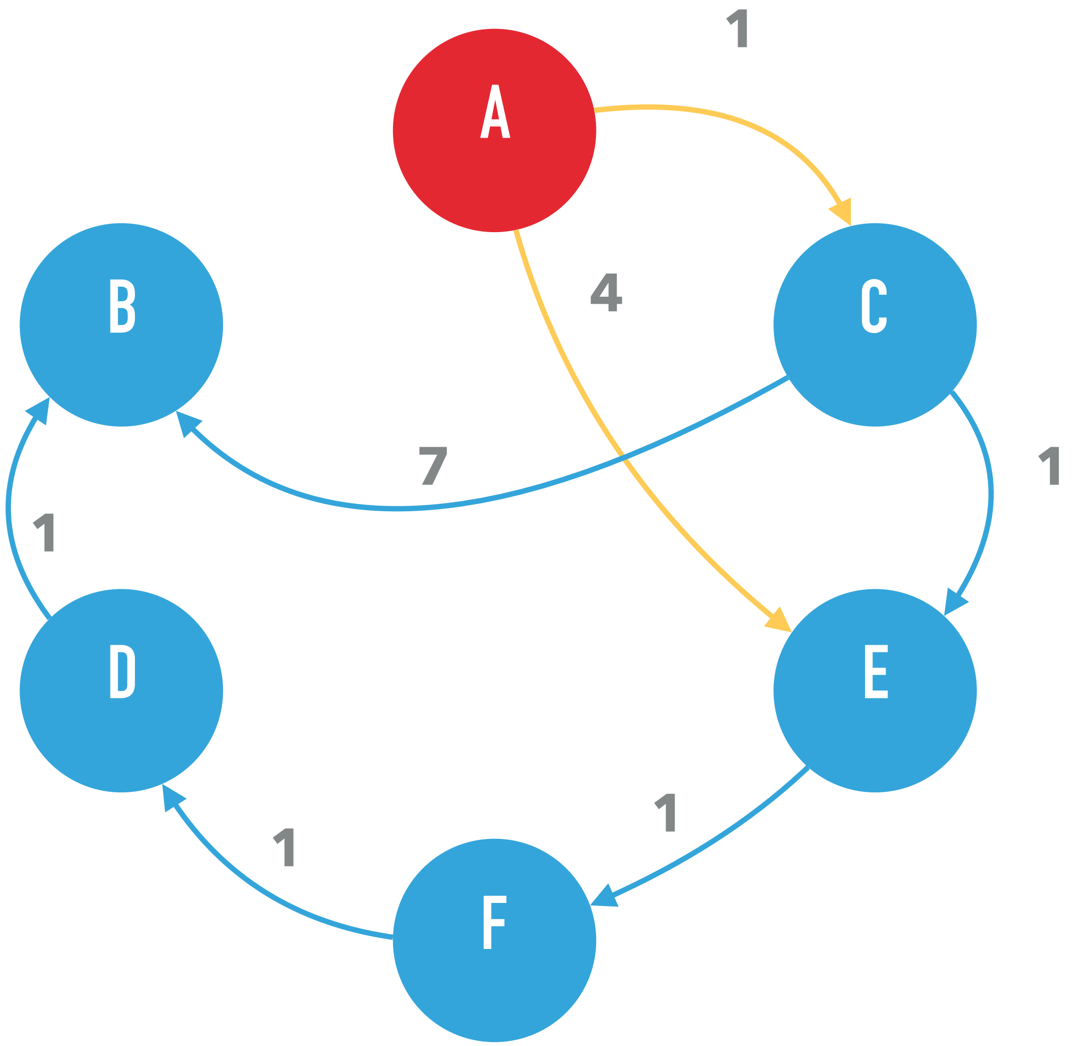
WEIGHTY DECISIONS

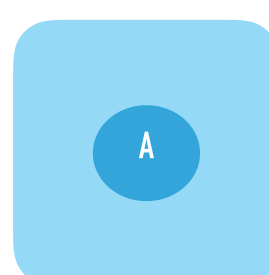
DEALING WITH WEIGHTY TOPICS



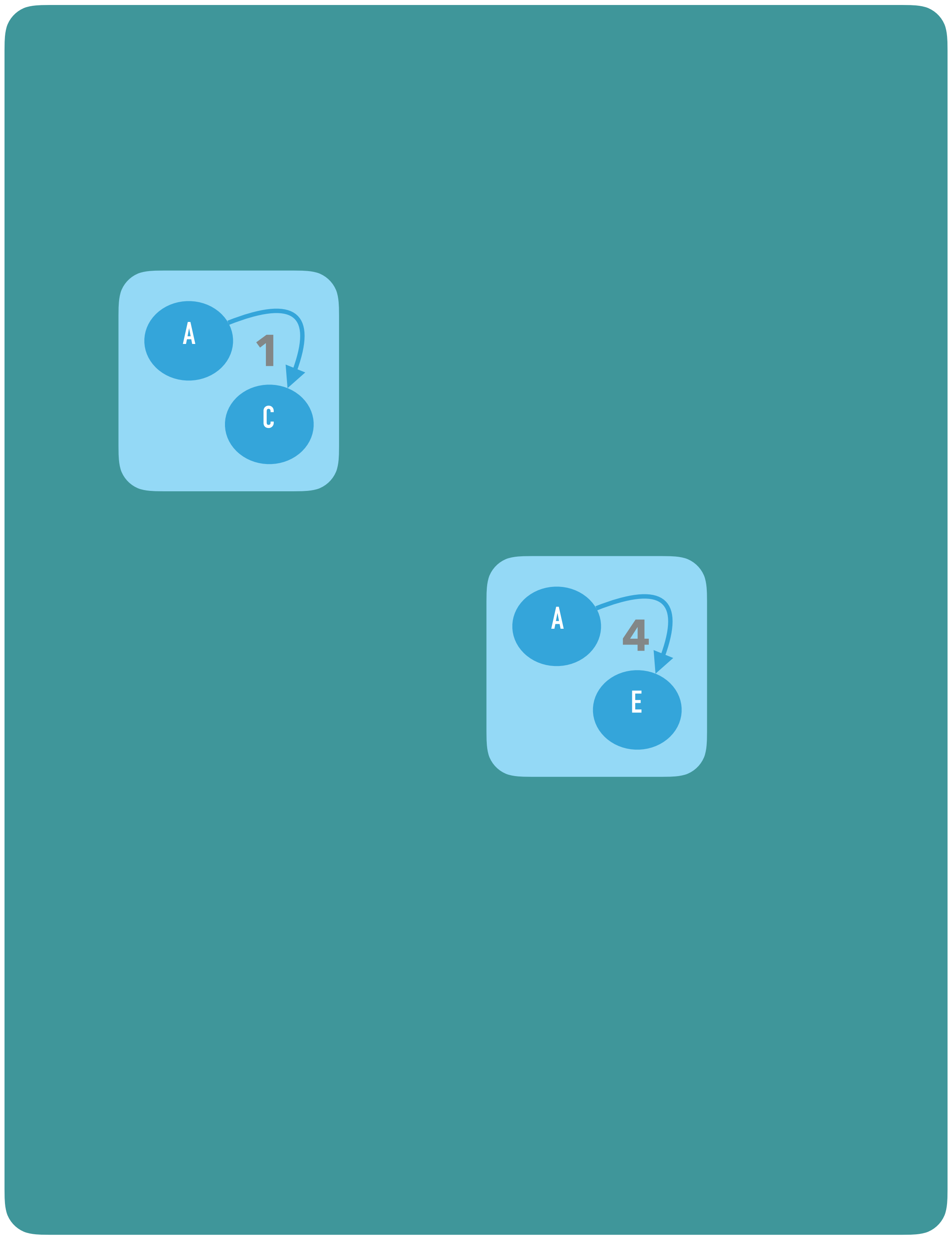


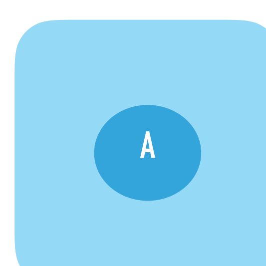
DEALING WITH WEIGHTY TOPICS

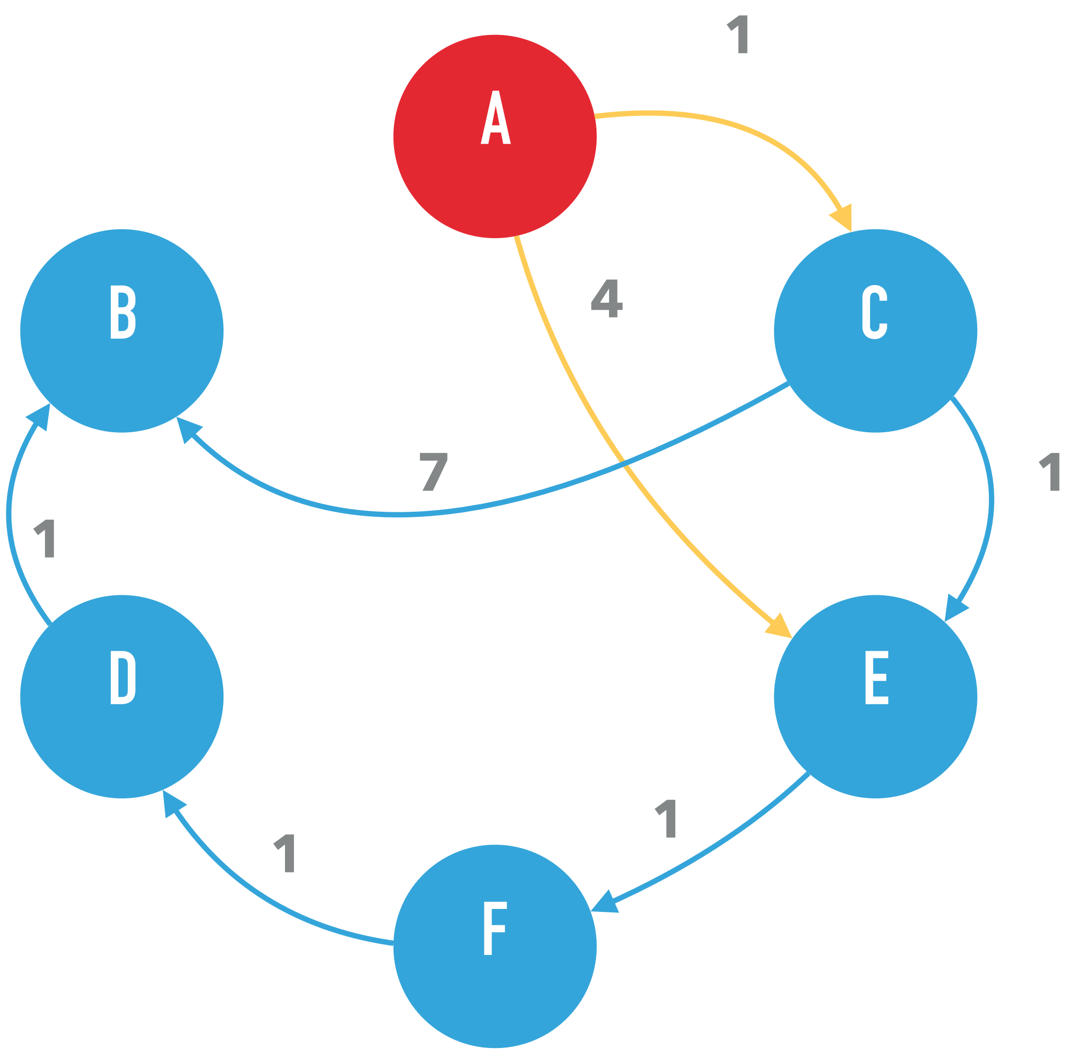




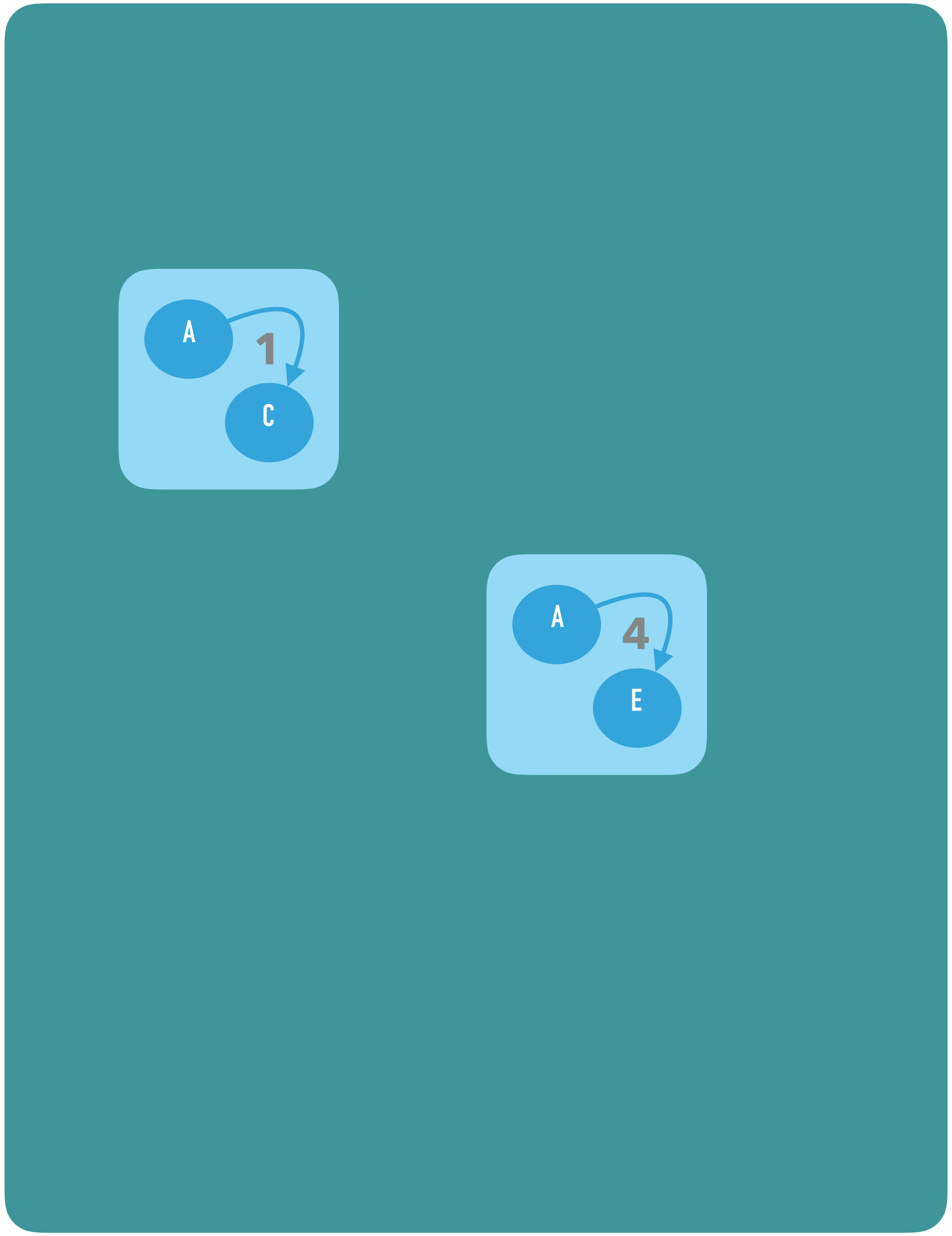
DEALING WITH WEIGHTY TOPICS

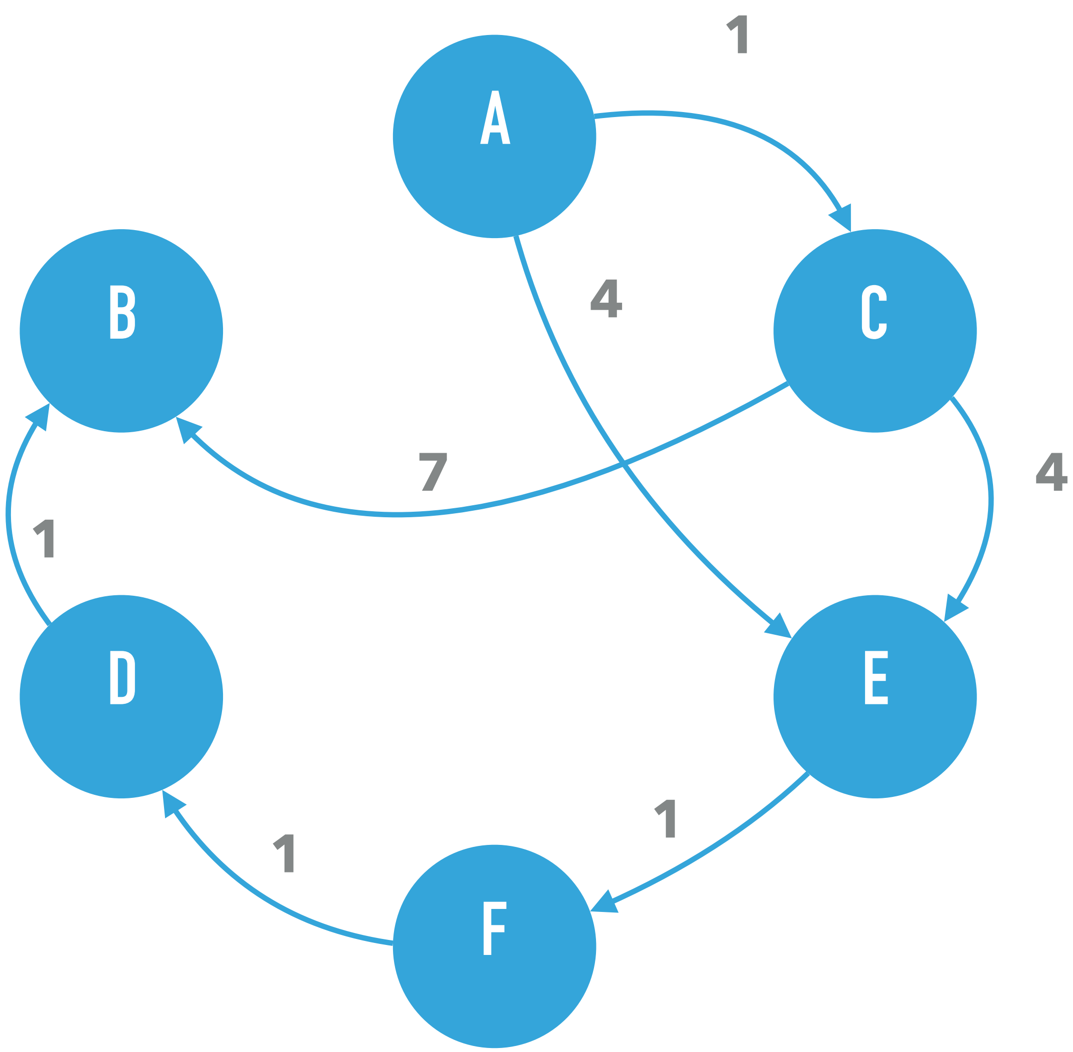




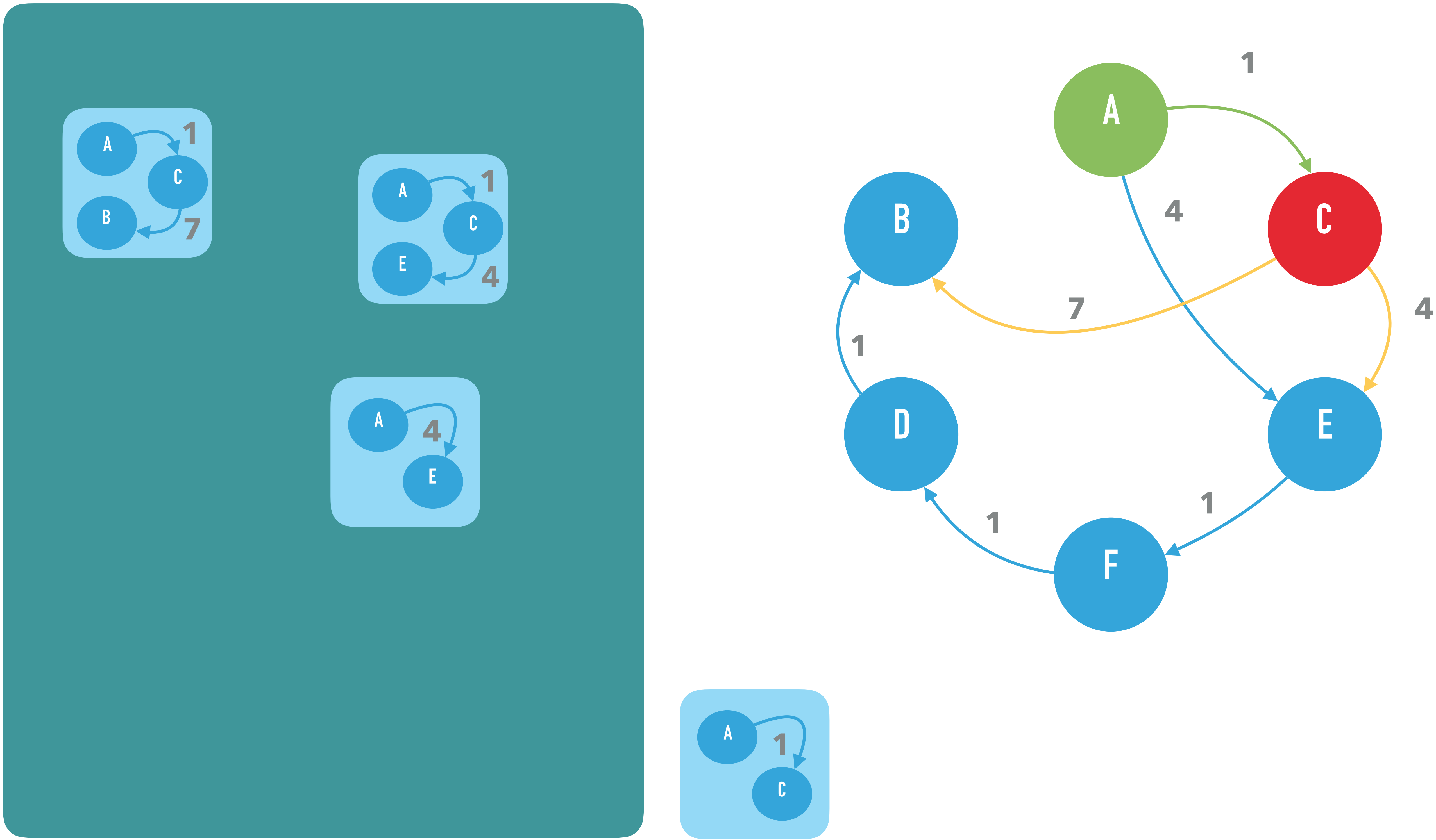


DEALING WITH WEIGHTY TOPICS

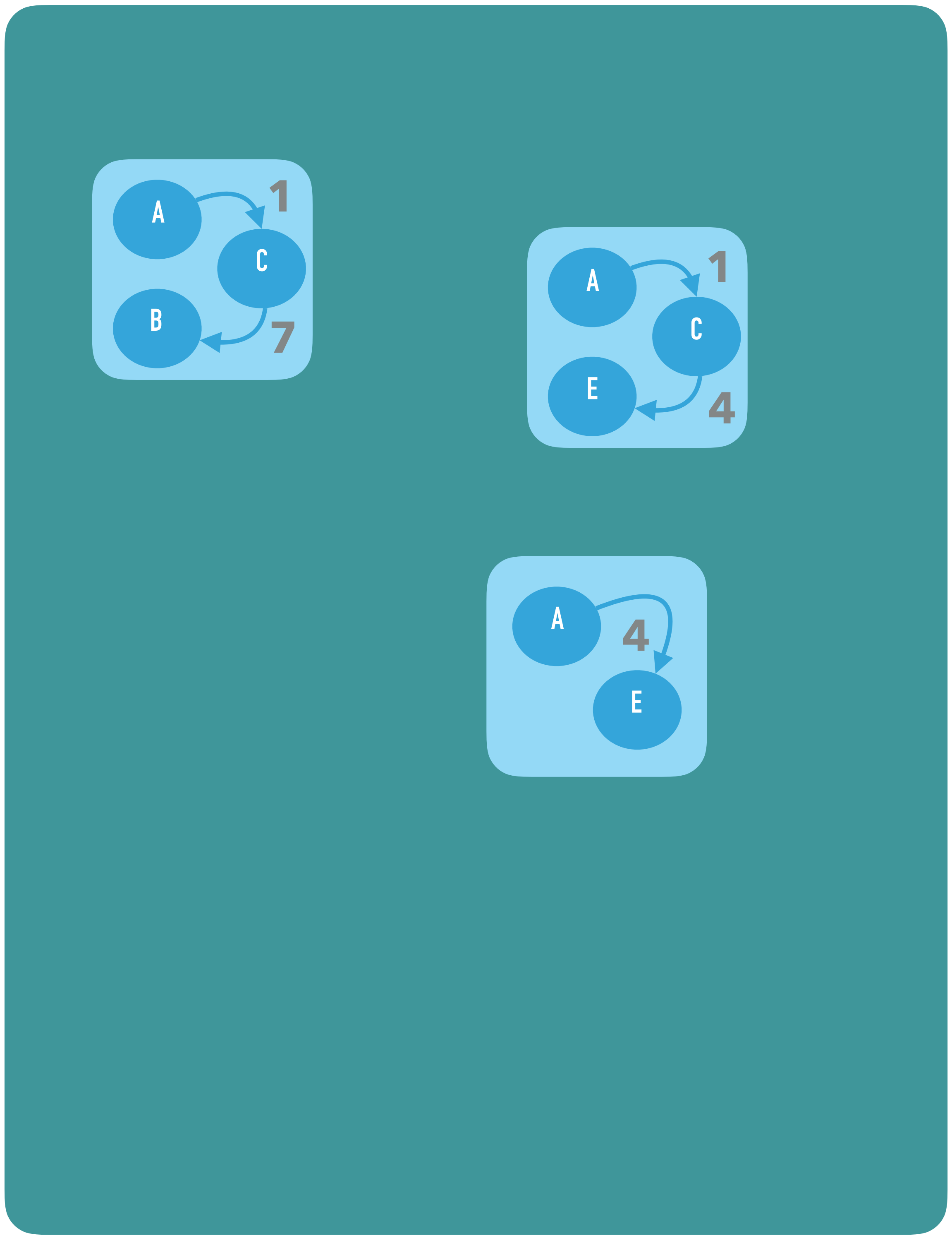


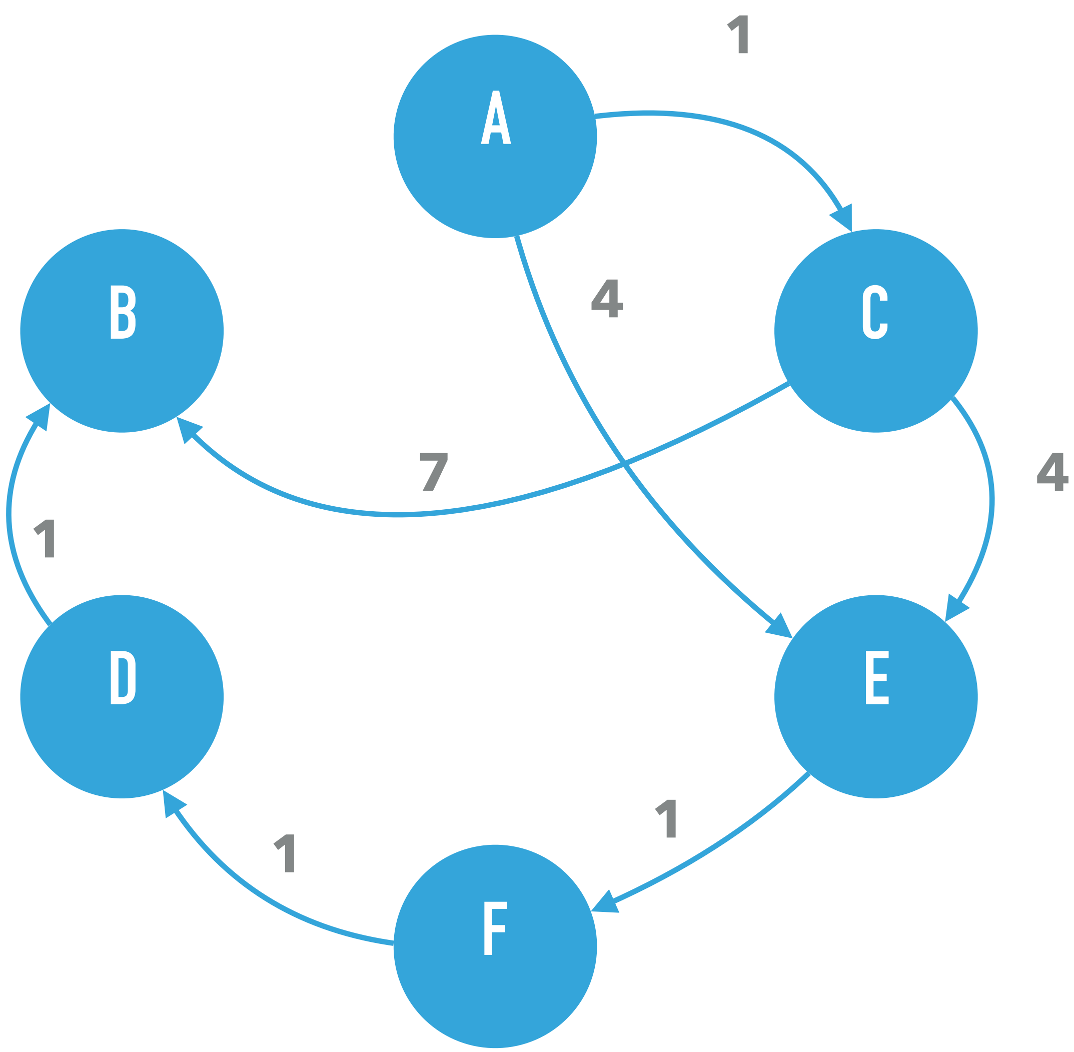


DEALING WITH WEIGHTY TOPICS

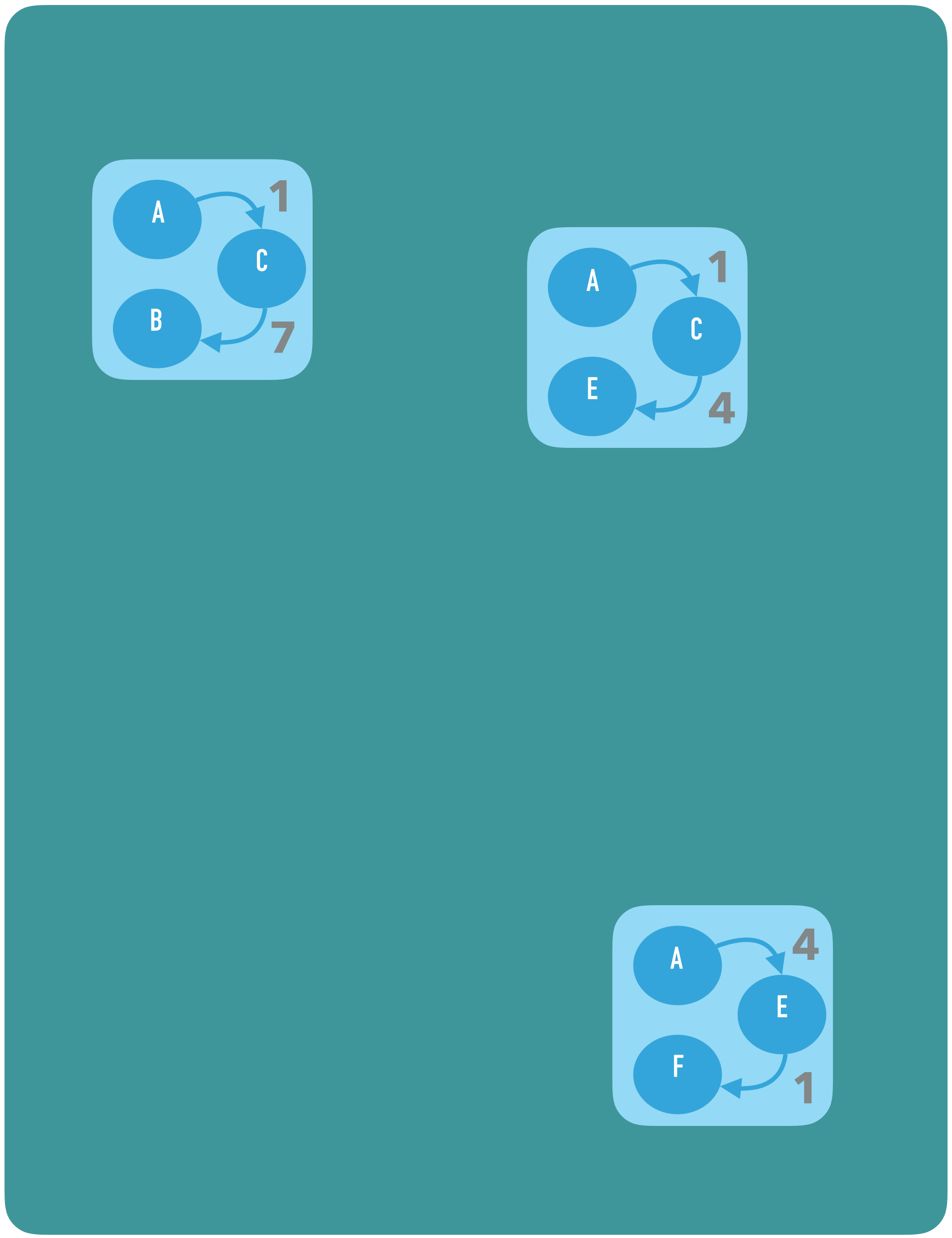


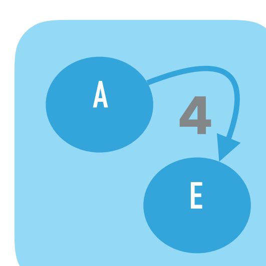
DEALING WITH WEIGHTY TOPICS

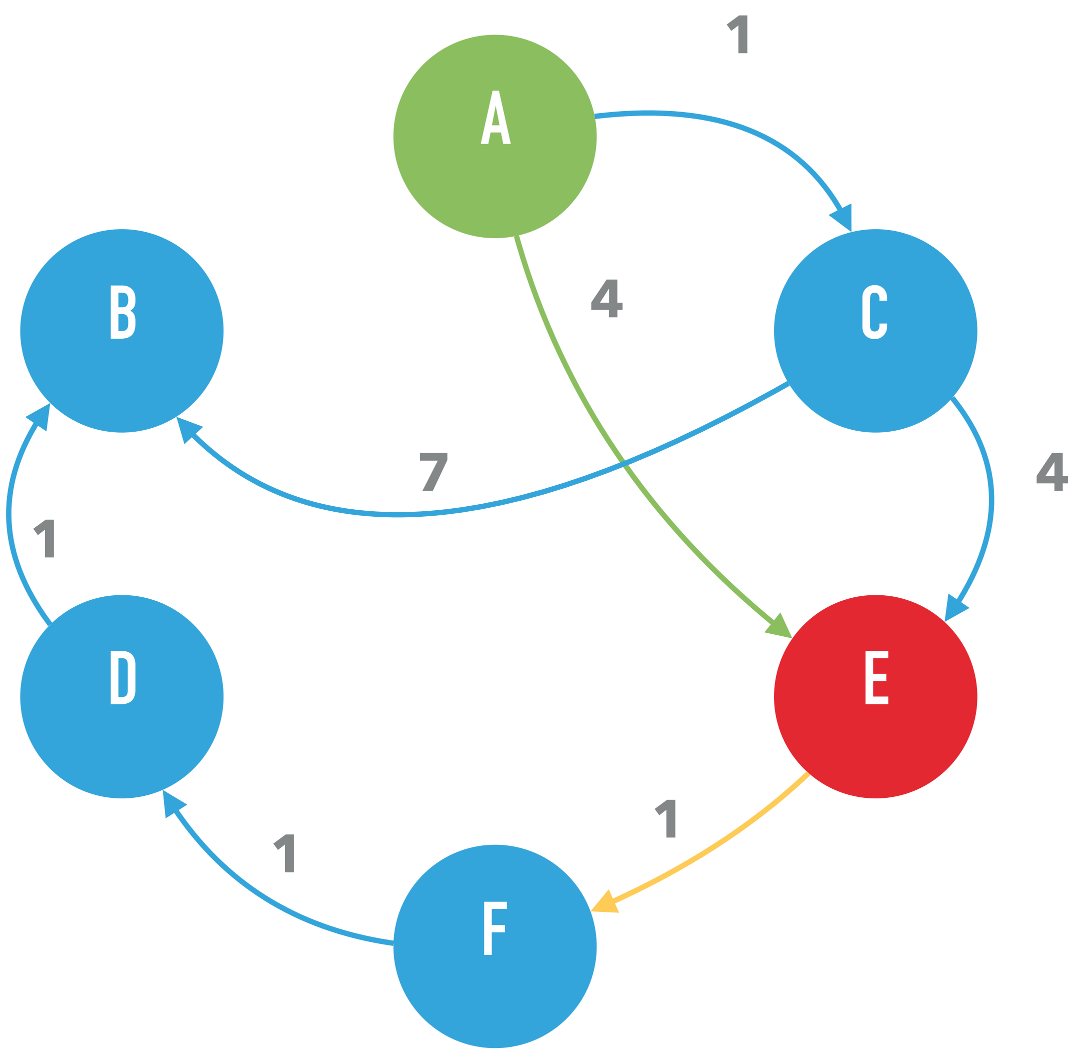




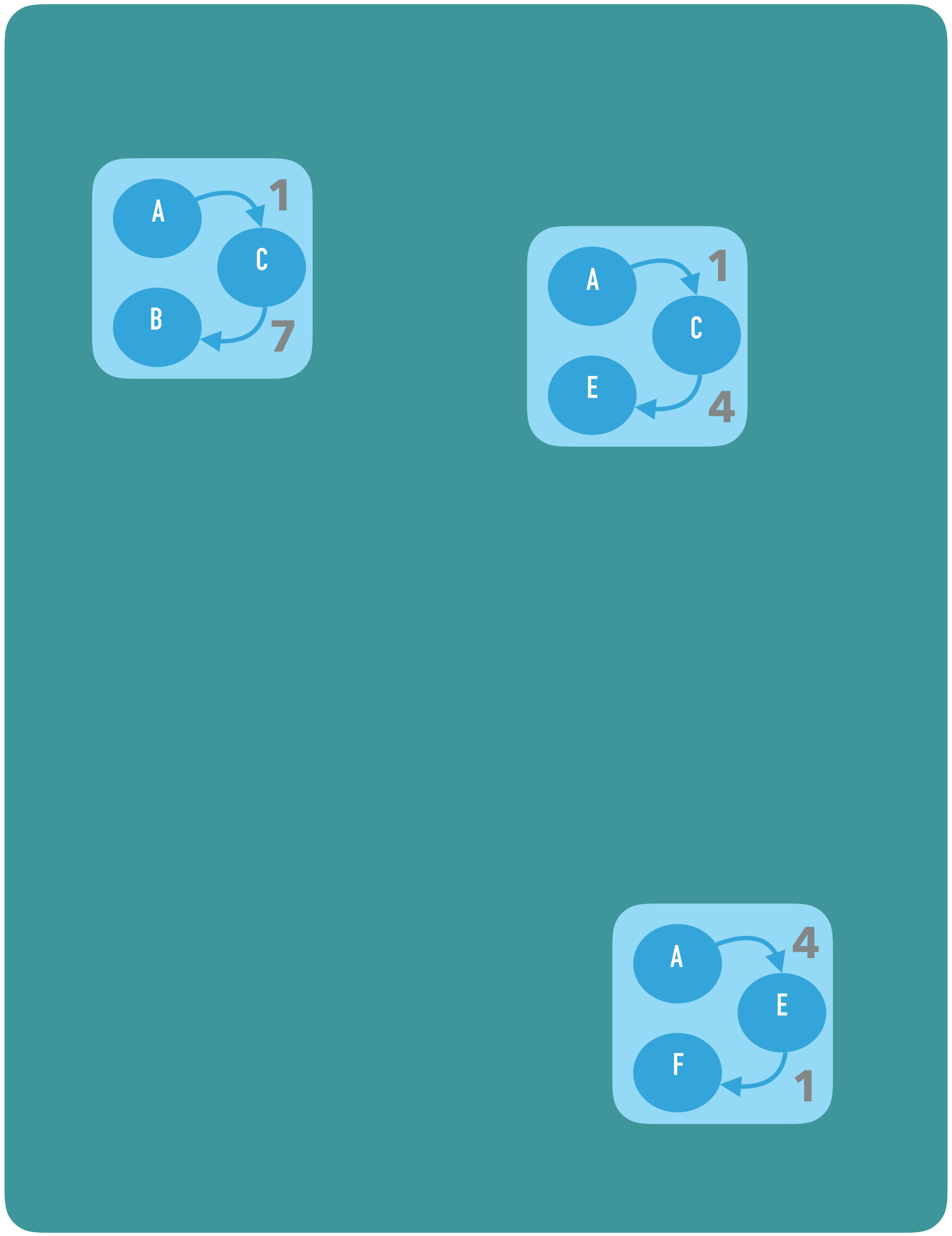
DEALING WITH WEIGHTY TOPICS

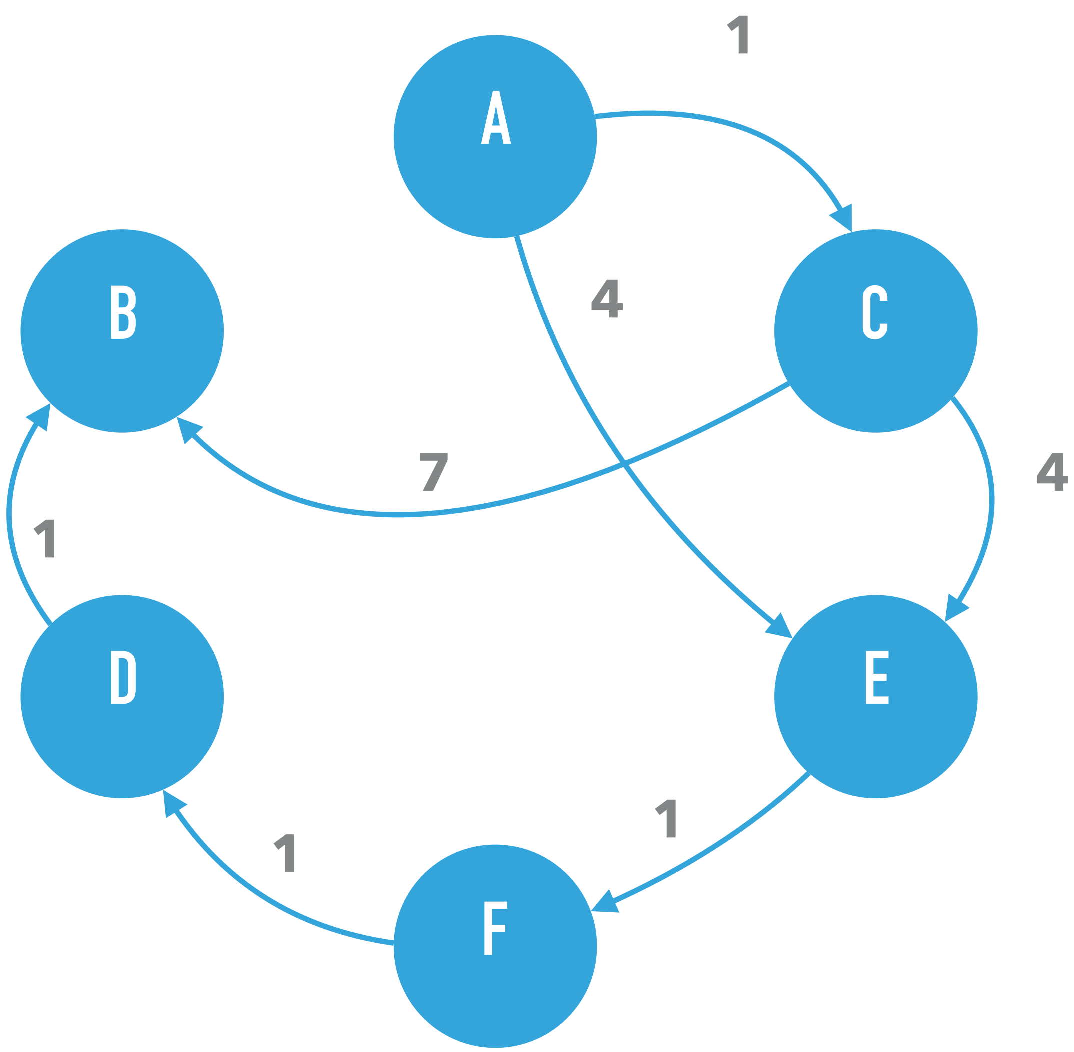




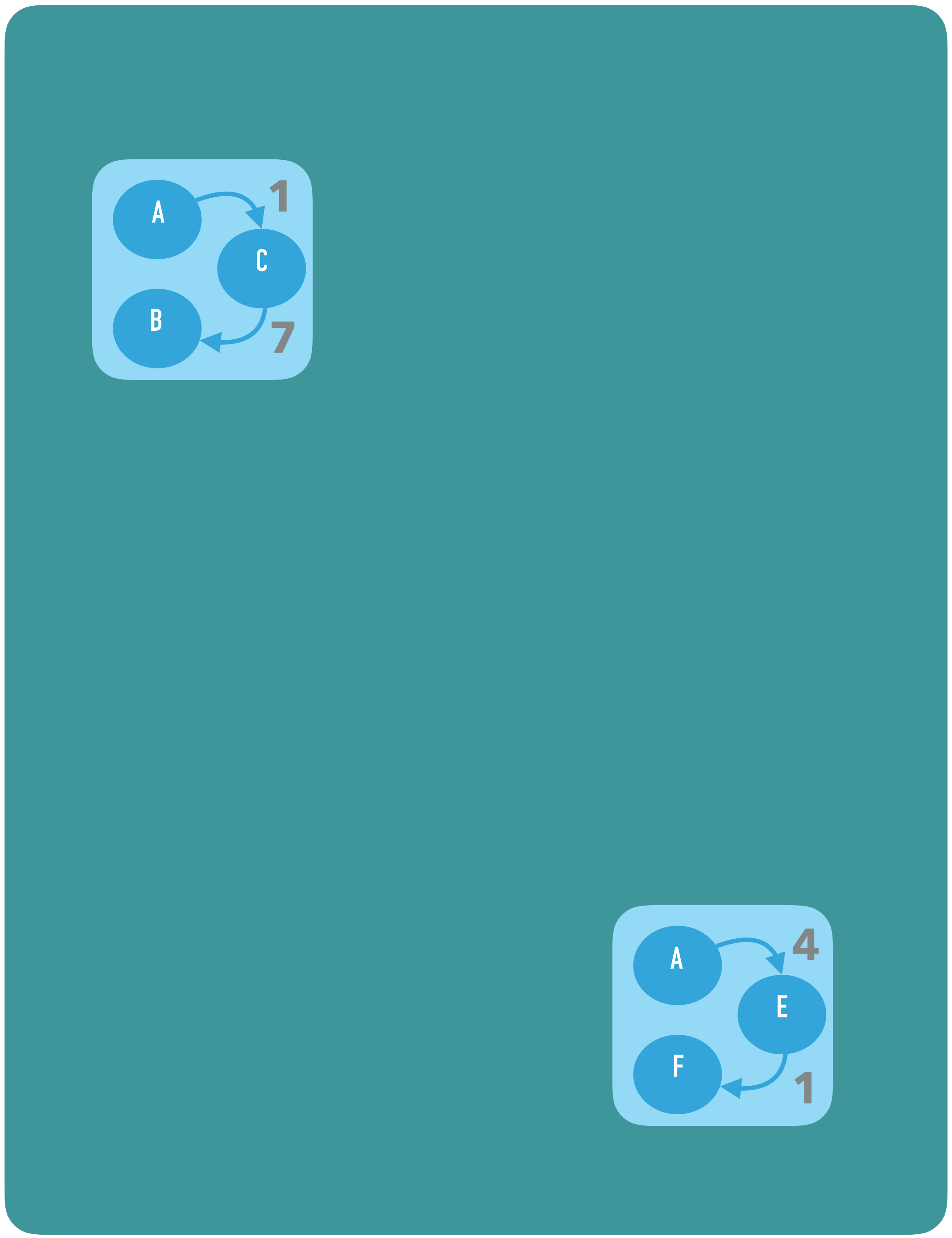


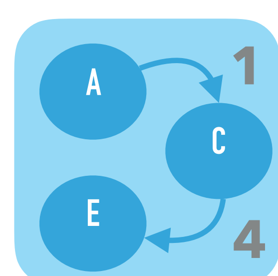
DEALING WITH WEIGHTY TOPICS

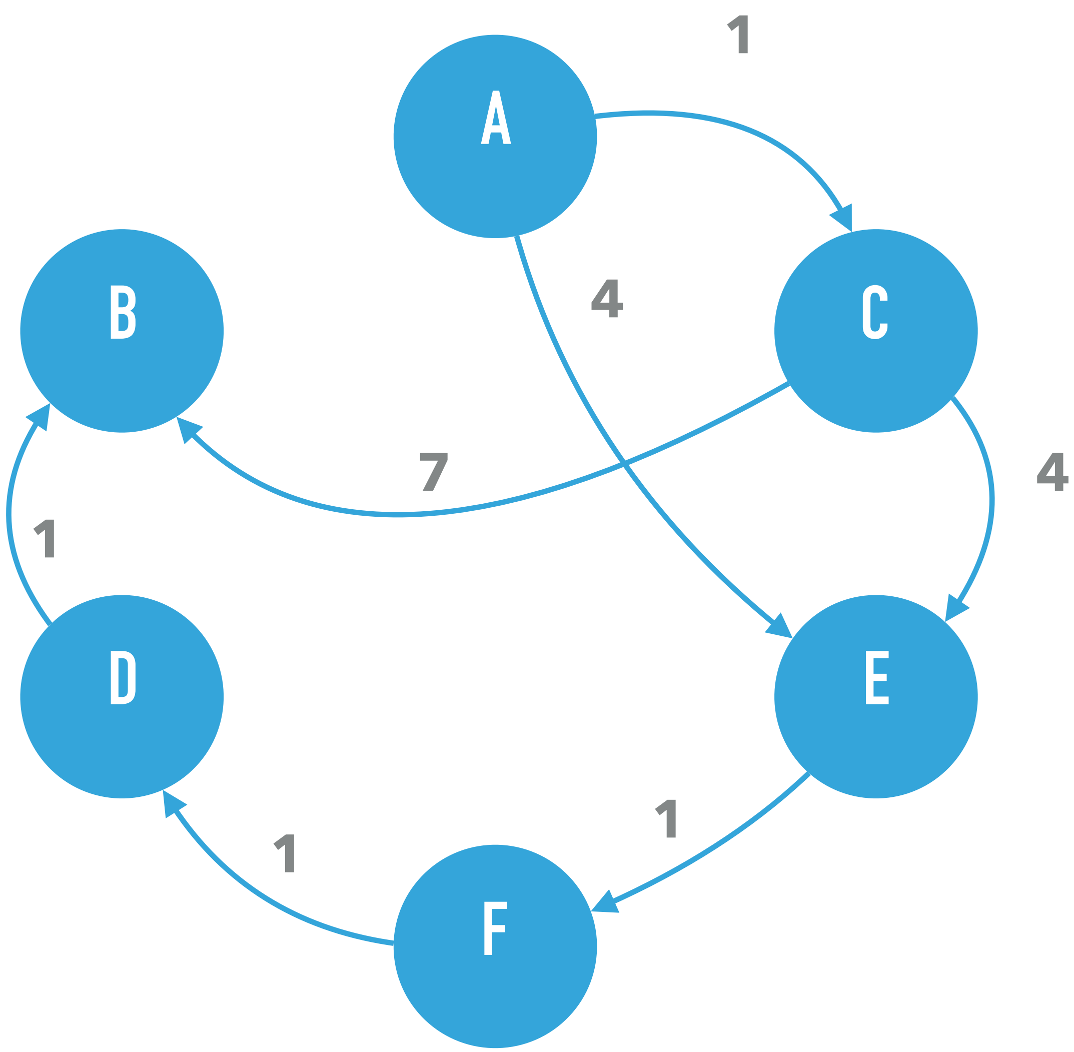




DEALING WITH WEIGHTY TOPICS







IN DIJKSTRA'S ALGORITHM,

THE TODO LIST IS A PRIORITY QUEUE

DIJKSTRA'S ALGORITHM (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.deque()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q

DIJKSTRA'S ALGORITHM (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

DIJKSTRA'S ODDS AND ENDS

- **create a path with just start node and enqueue into priority queue q**
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- What do you initialize the weight of the path to?

DIJKSTRA'S ODDS AND ENDS

- **create a path with just start node and enqueue into priority queue q**
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- What do you initialize the weight of the path to?
  - Zero should be fine

DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - **if v is end node, you're done**
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- Can't I just return the path as soon as I find the end node? Why wait until I dequeue?

DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - **if v is end node, you're done**
  - if you've seen v before, skip it
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

- Can't I just return the path as soon as I find the end node? Why wait until I dequeue?
  - This is one of the most common mistakes people make with Dijkstra's!
  - It's possible a path with a lower priority gets enqueued in the meantime.

DIJKSTRA'S ODDS AND ENDS

- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - **if you've seen v before, skip it**
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

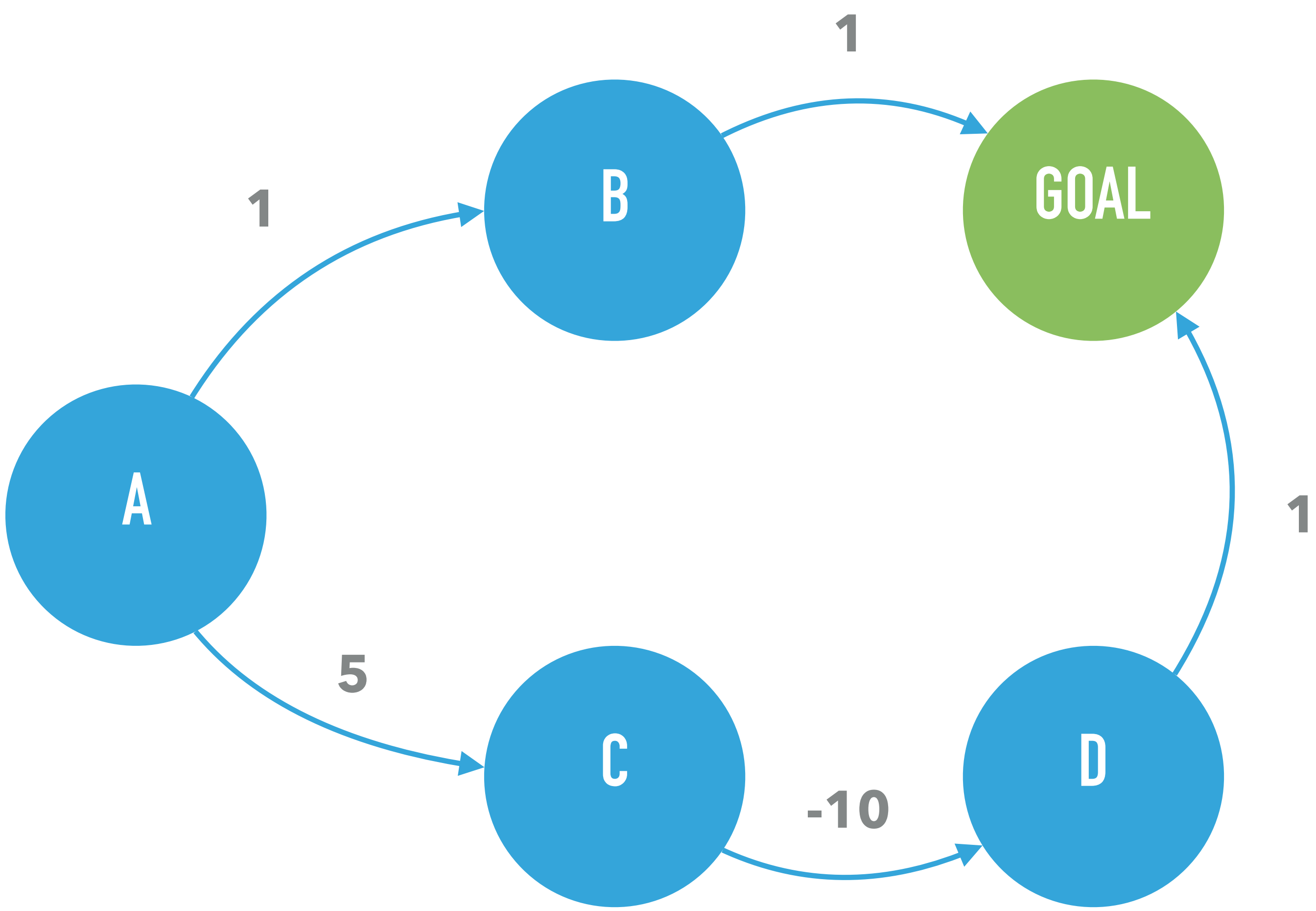
- Why would you skip the node just because you've seen it before?

DIJKSTRA'S ODDS AND ENDS

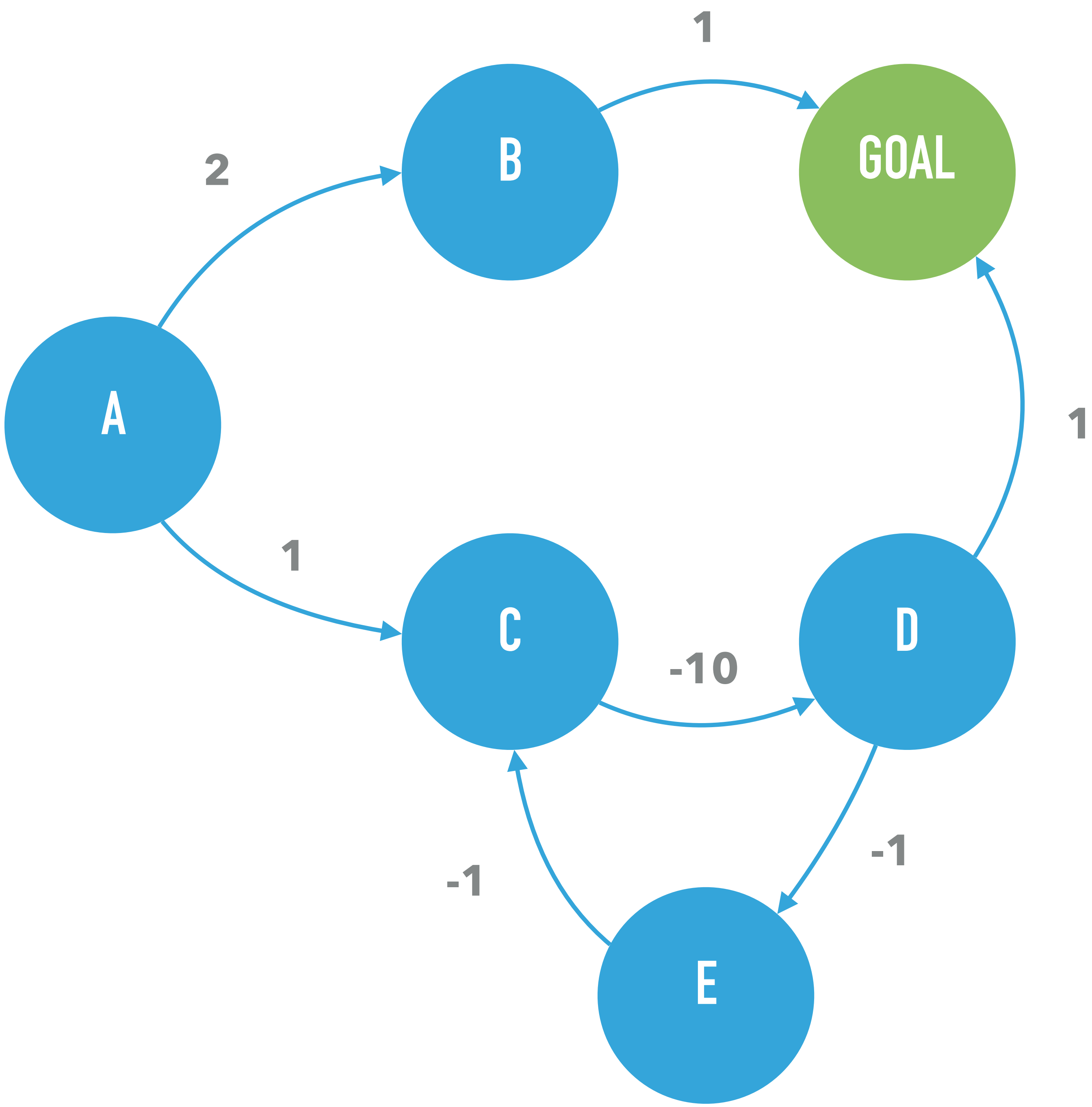
- create a path with just start node and enqueue into priority queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end node, you're done
  - **if you've seen v before, skip it**
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

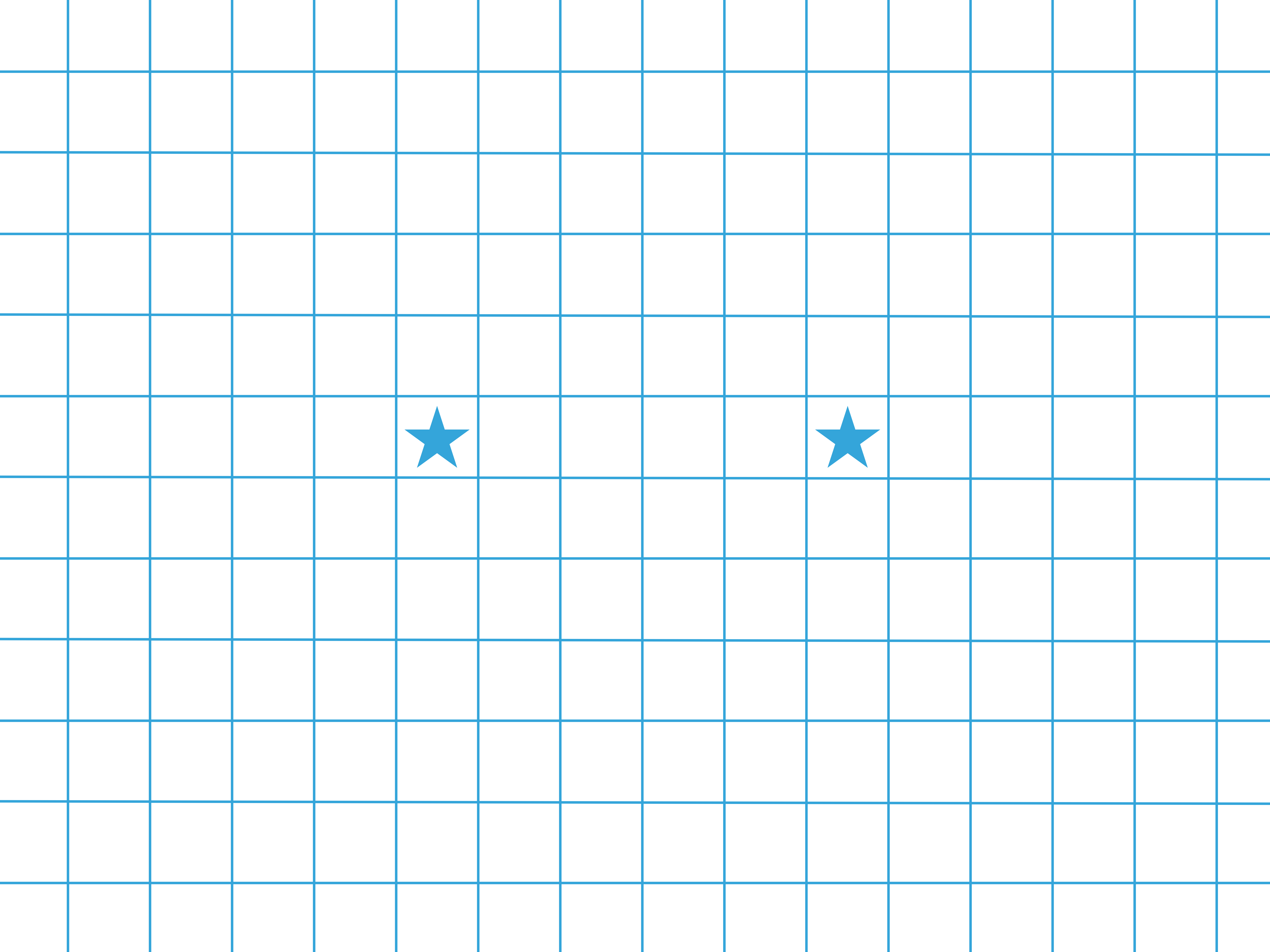
- Why would you skip the node just because you've seen it before?
  - If you've seen the node before, that means you've already found a shorter path to it.
  - Any path that follows from this one already has a shorter equivalent
  - **The first path you find to v will be the shortest path to v**

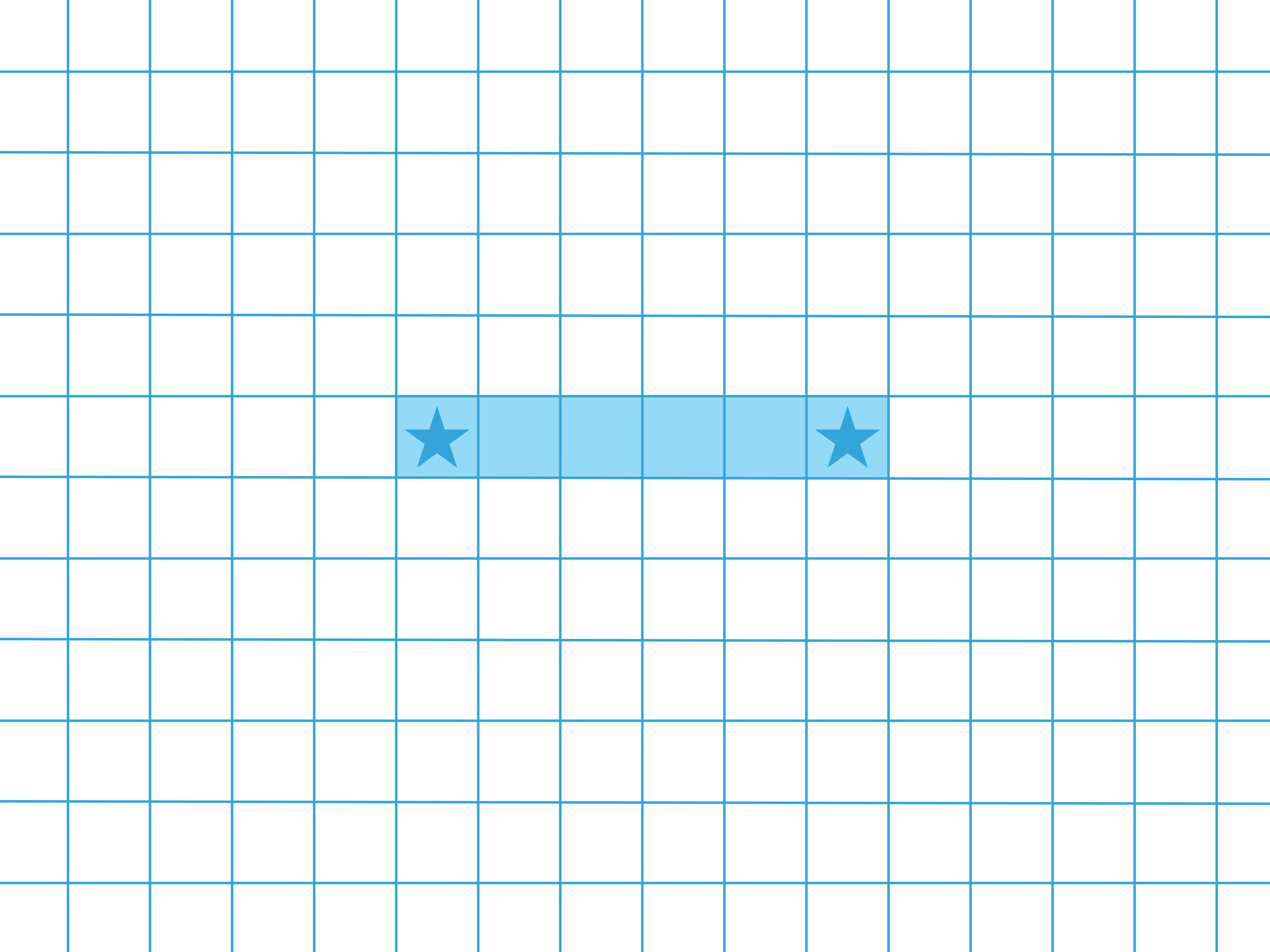
NEGATIVE EDGES

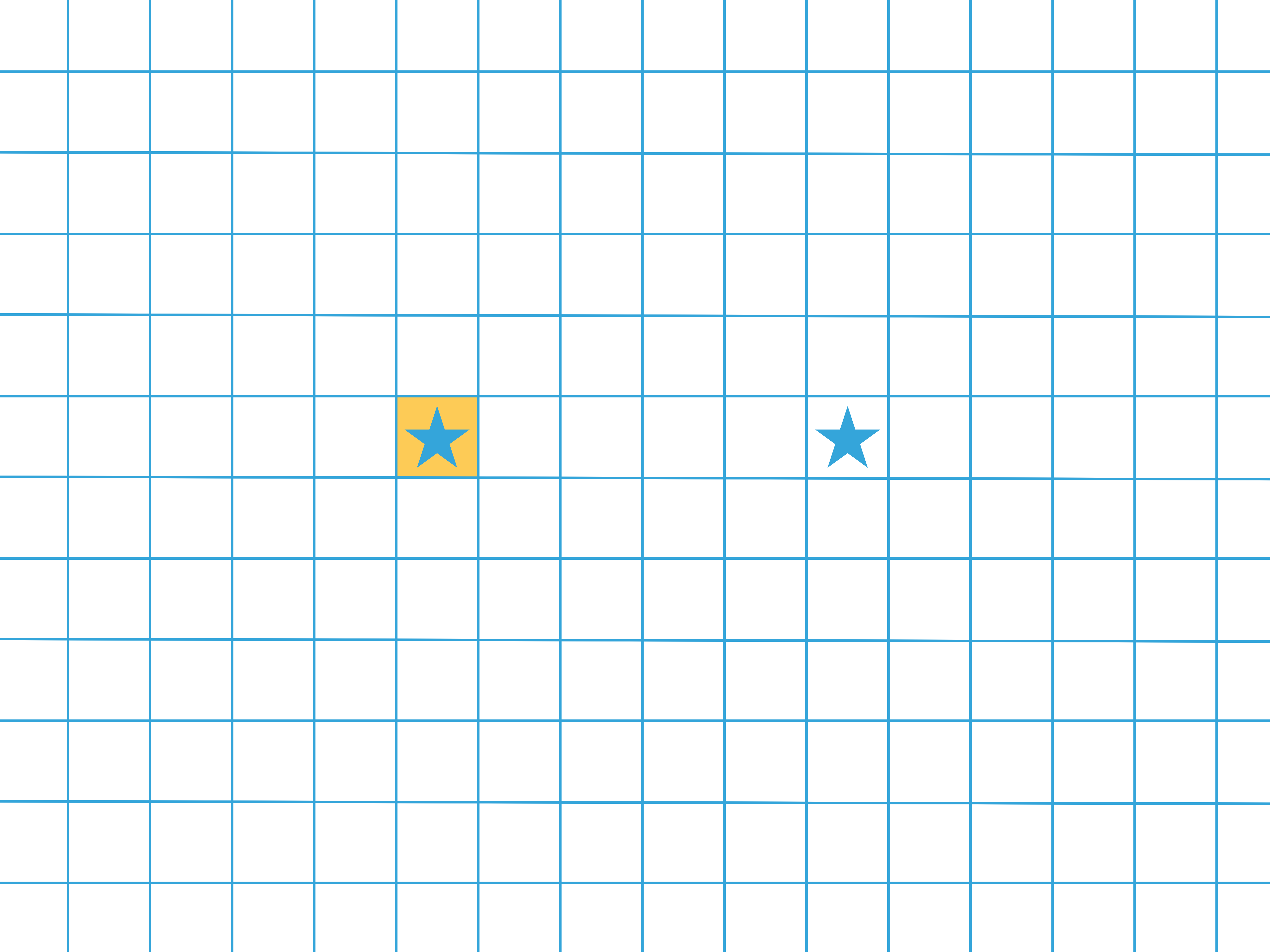


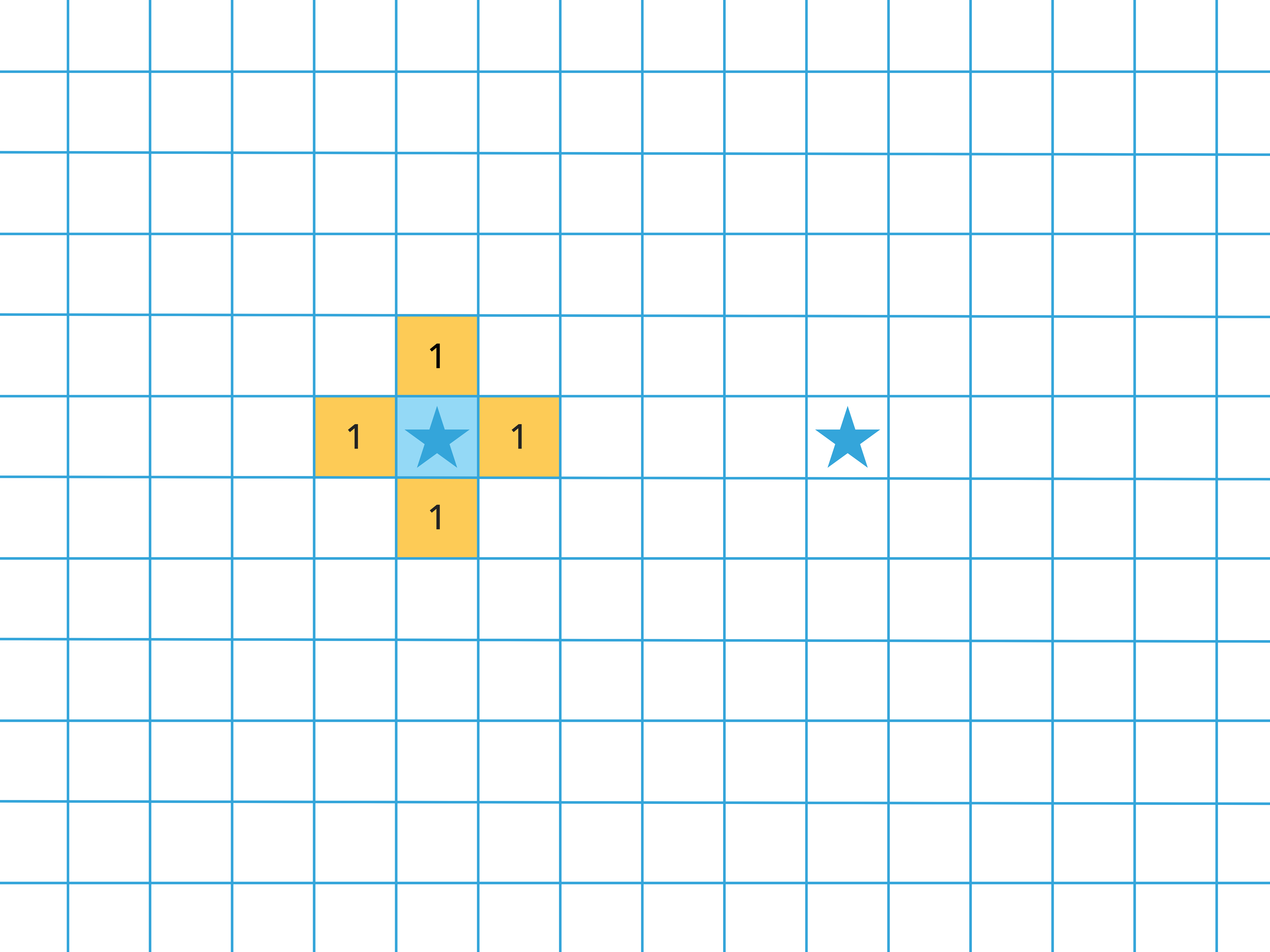
NEGATIVE CYCLES

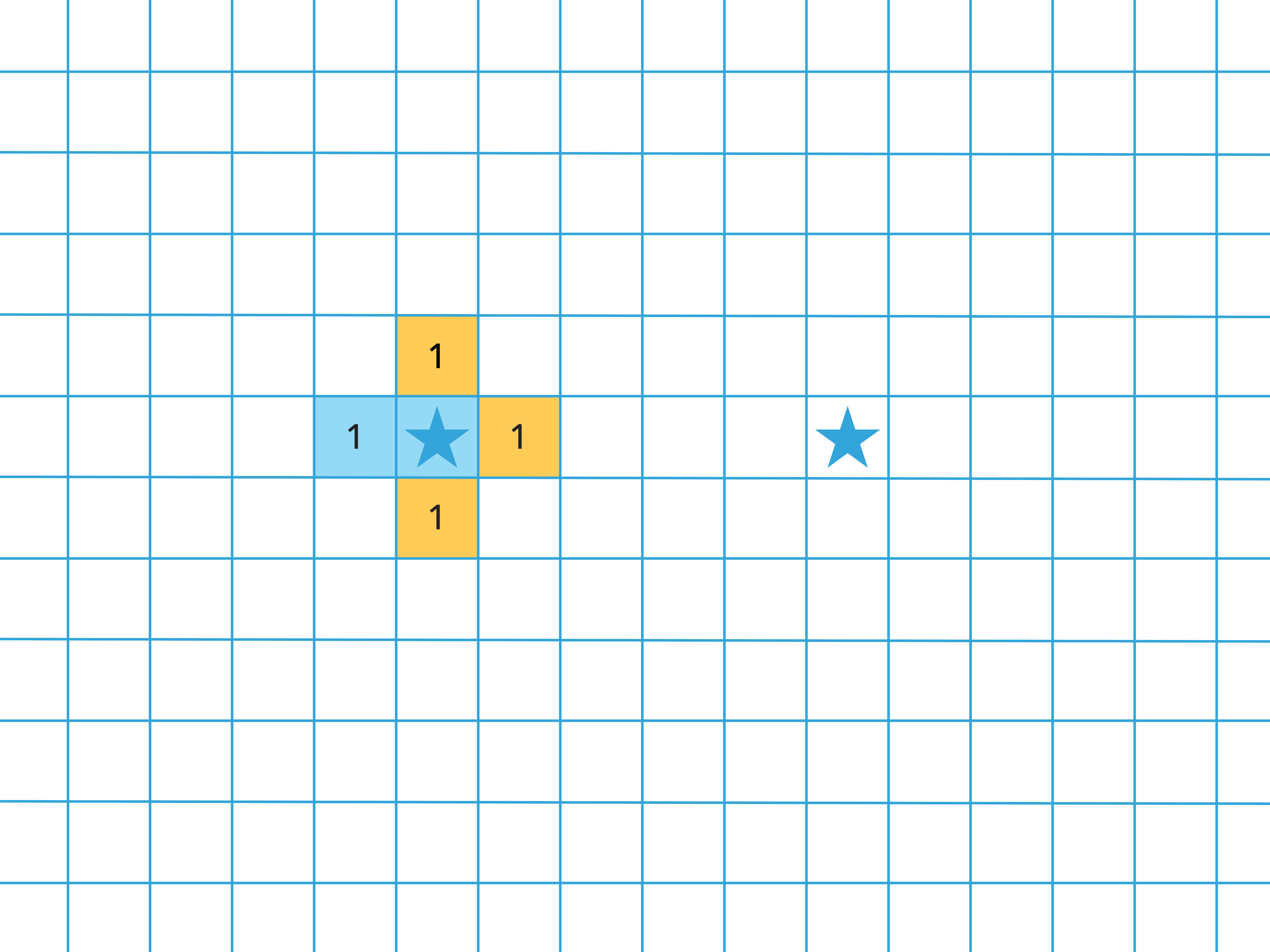




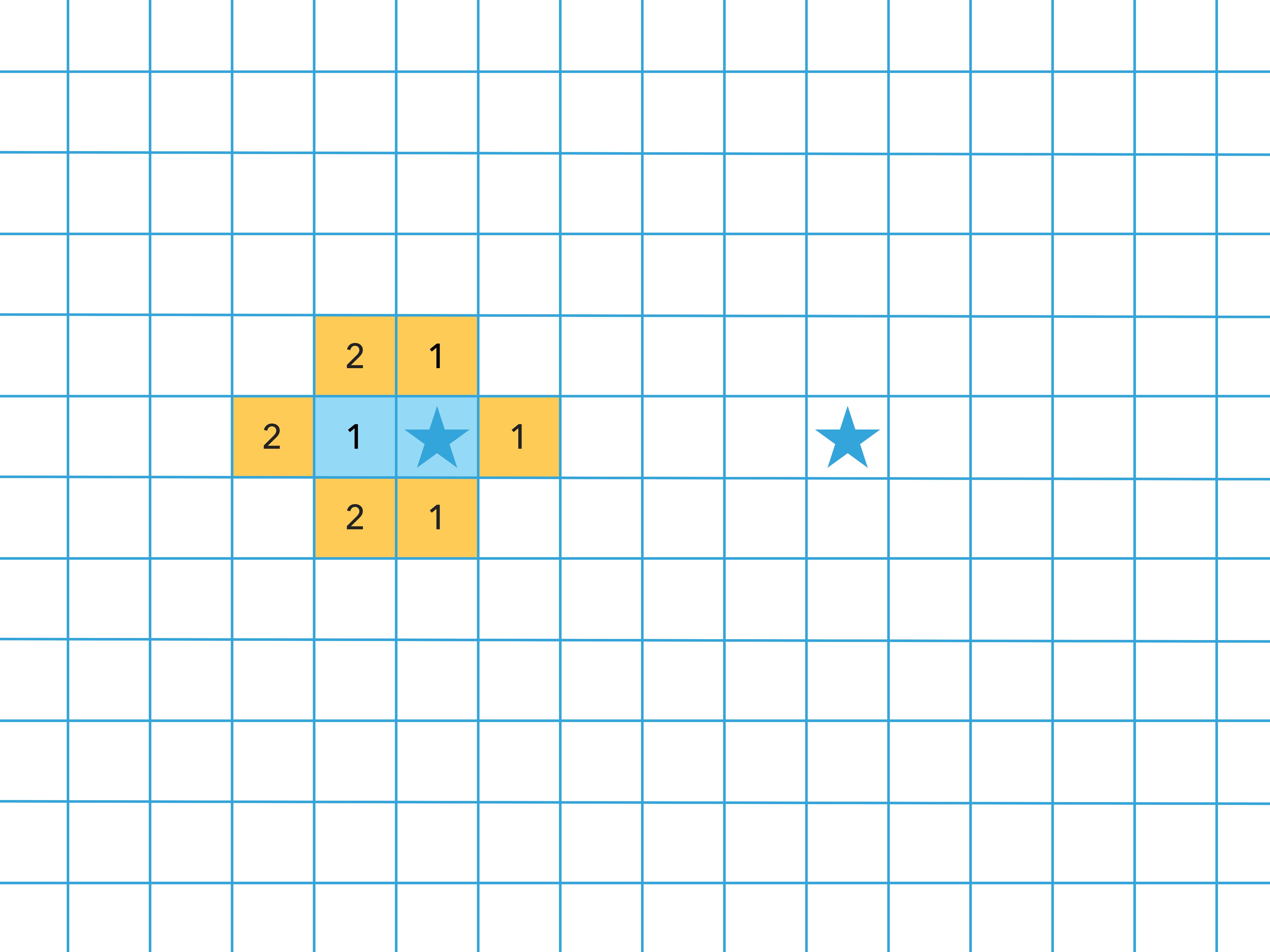


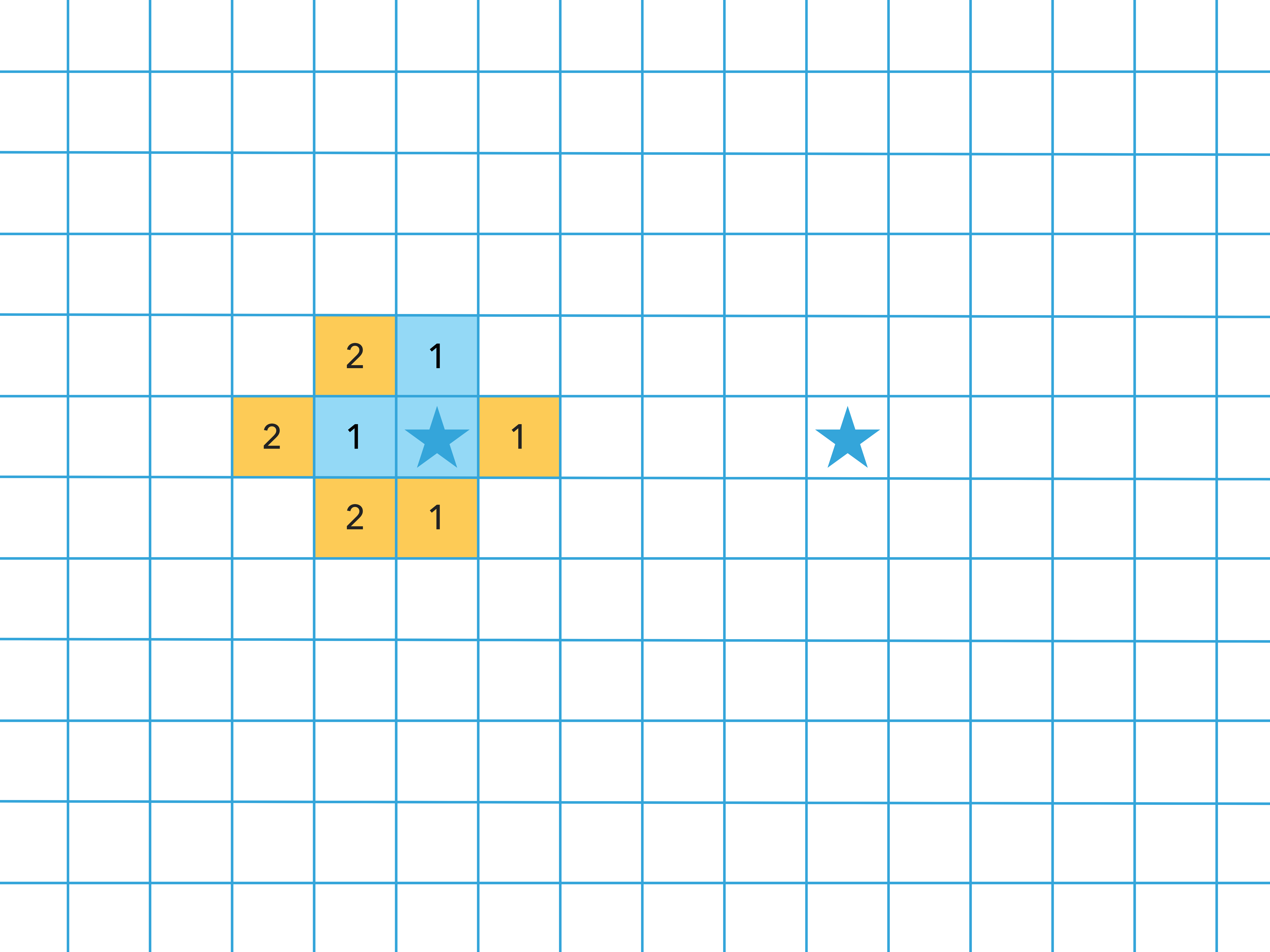


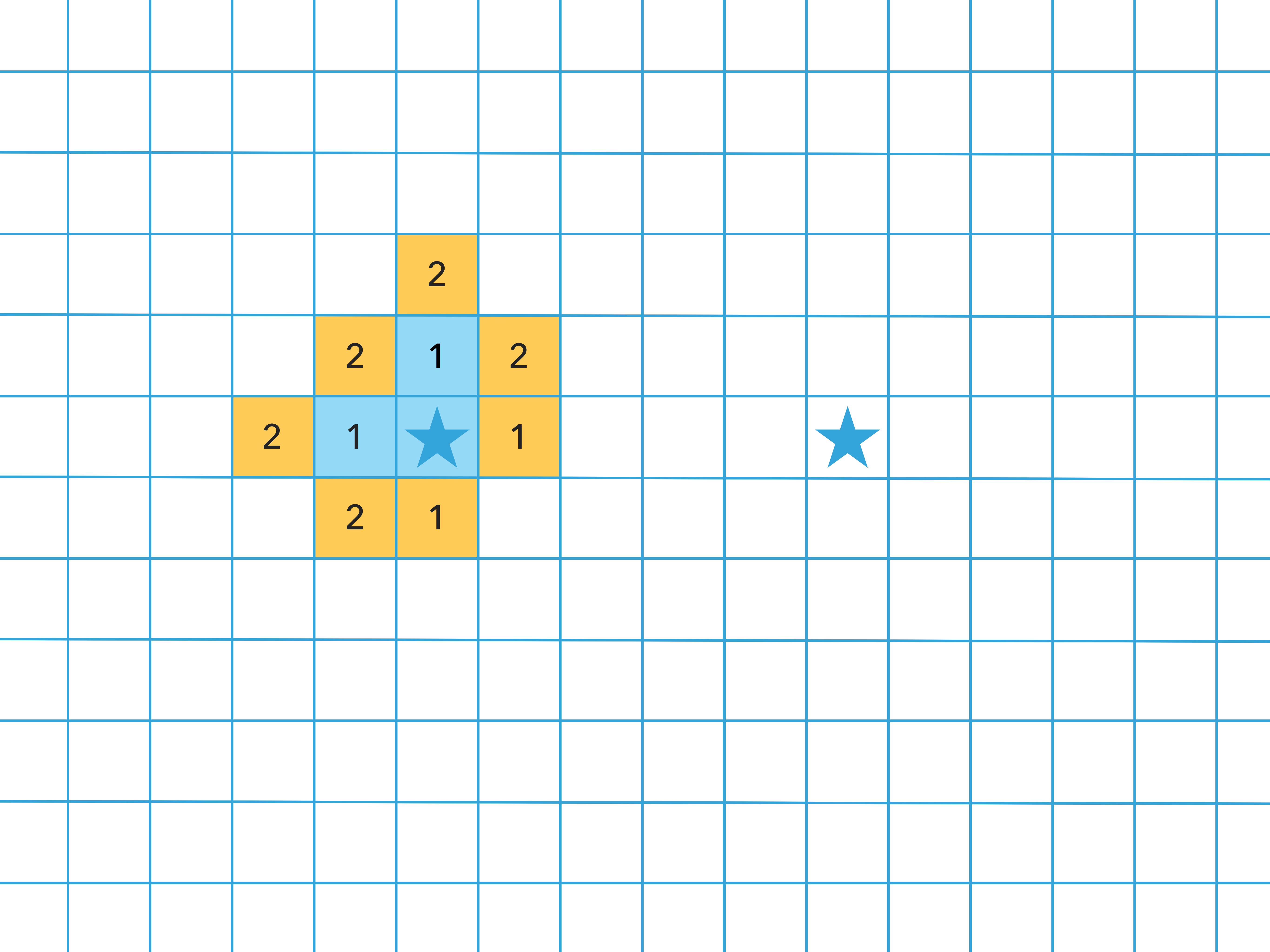

1
1
1
1

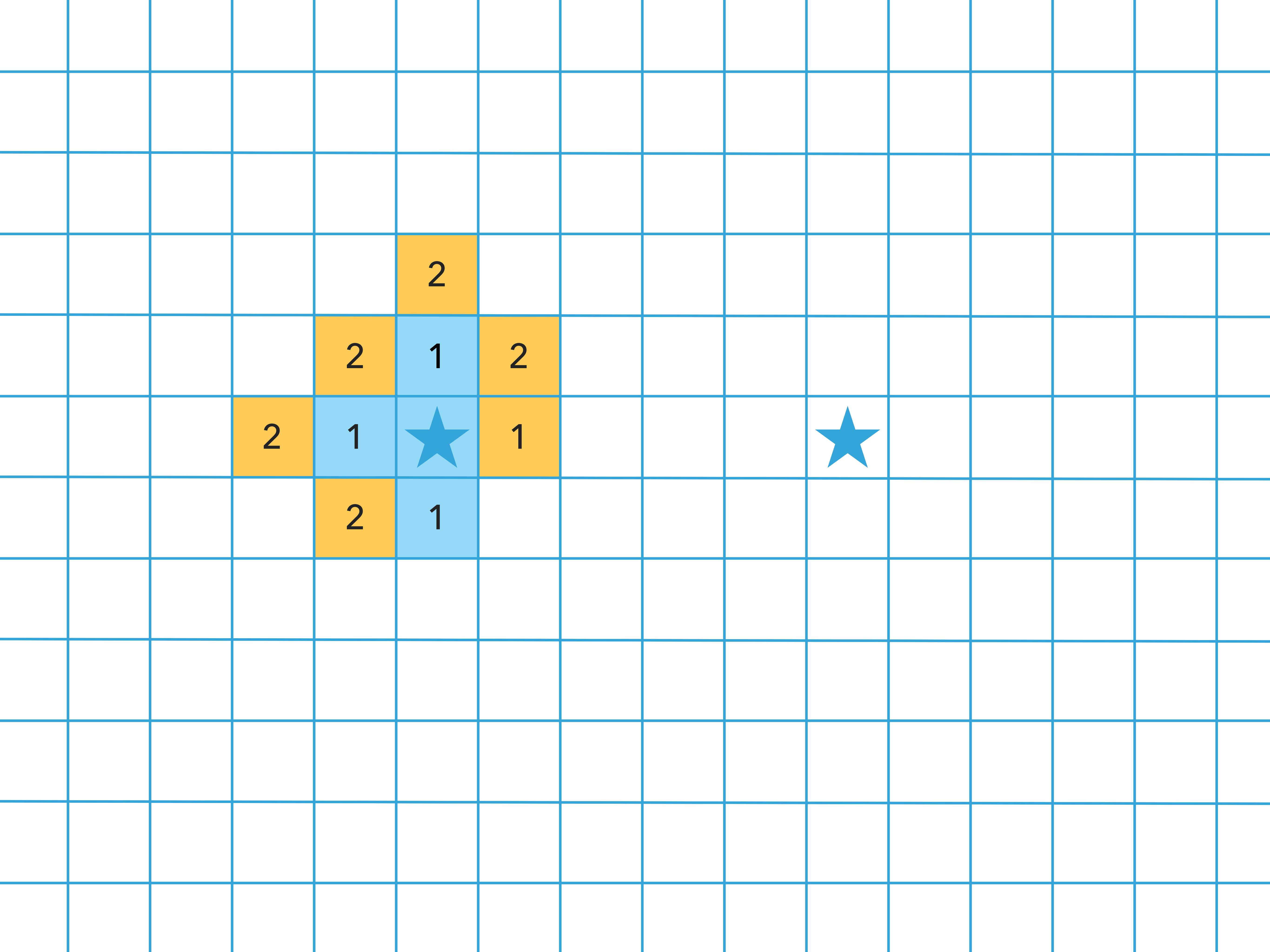


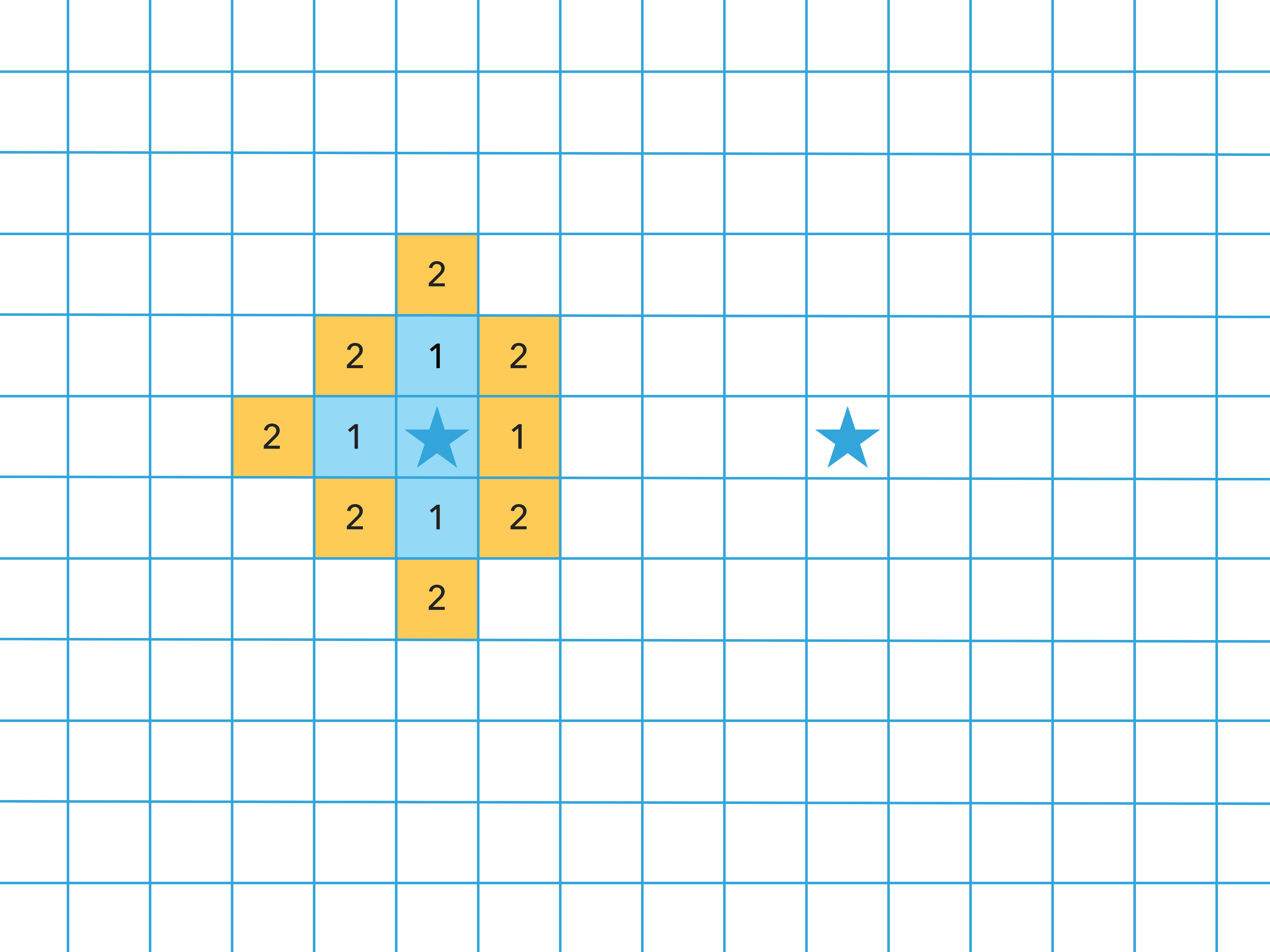
1
1
1
1


2
1
2
1
1
2
1

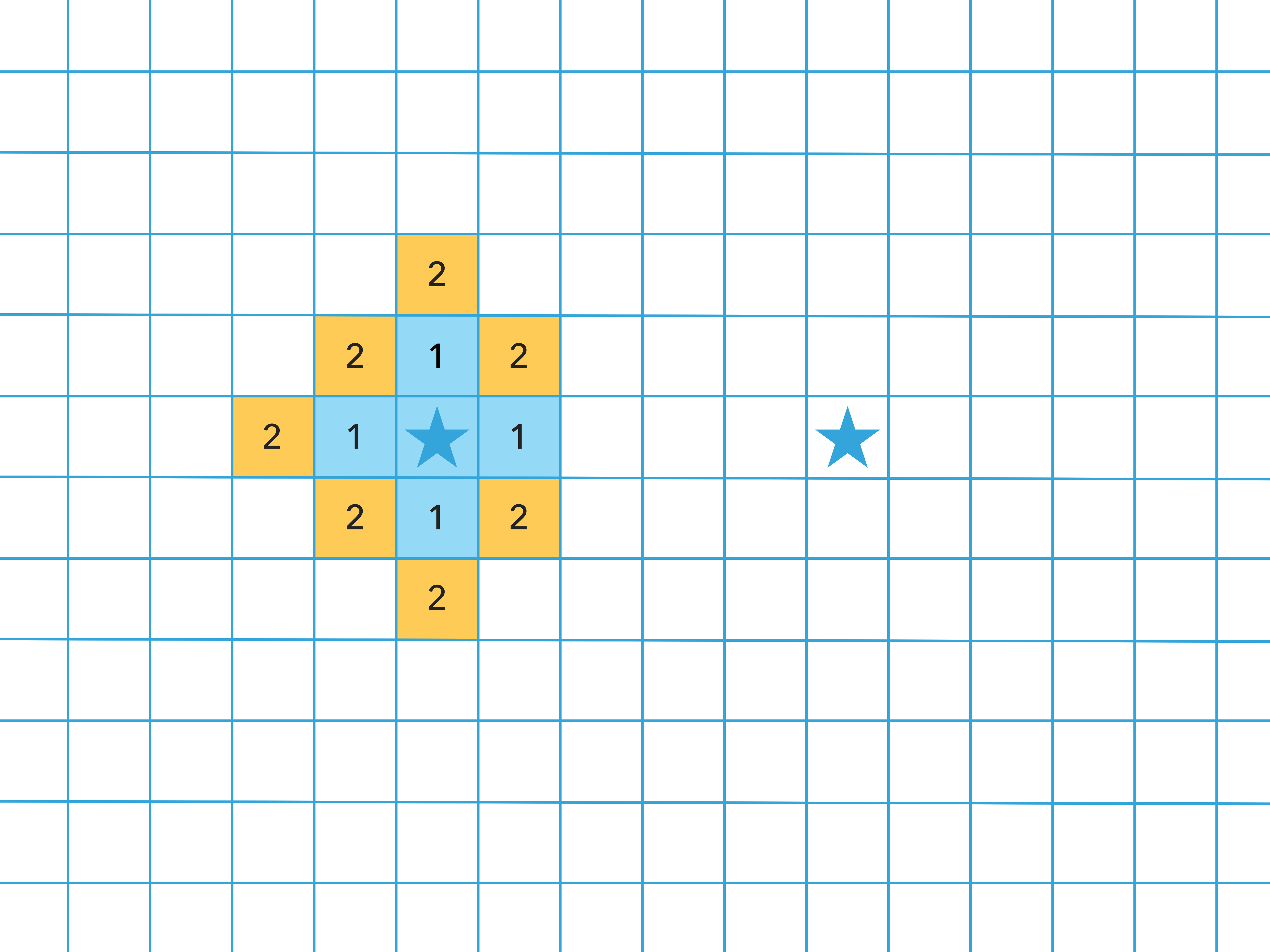

2
1
2
1
1
2
1

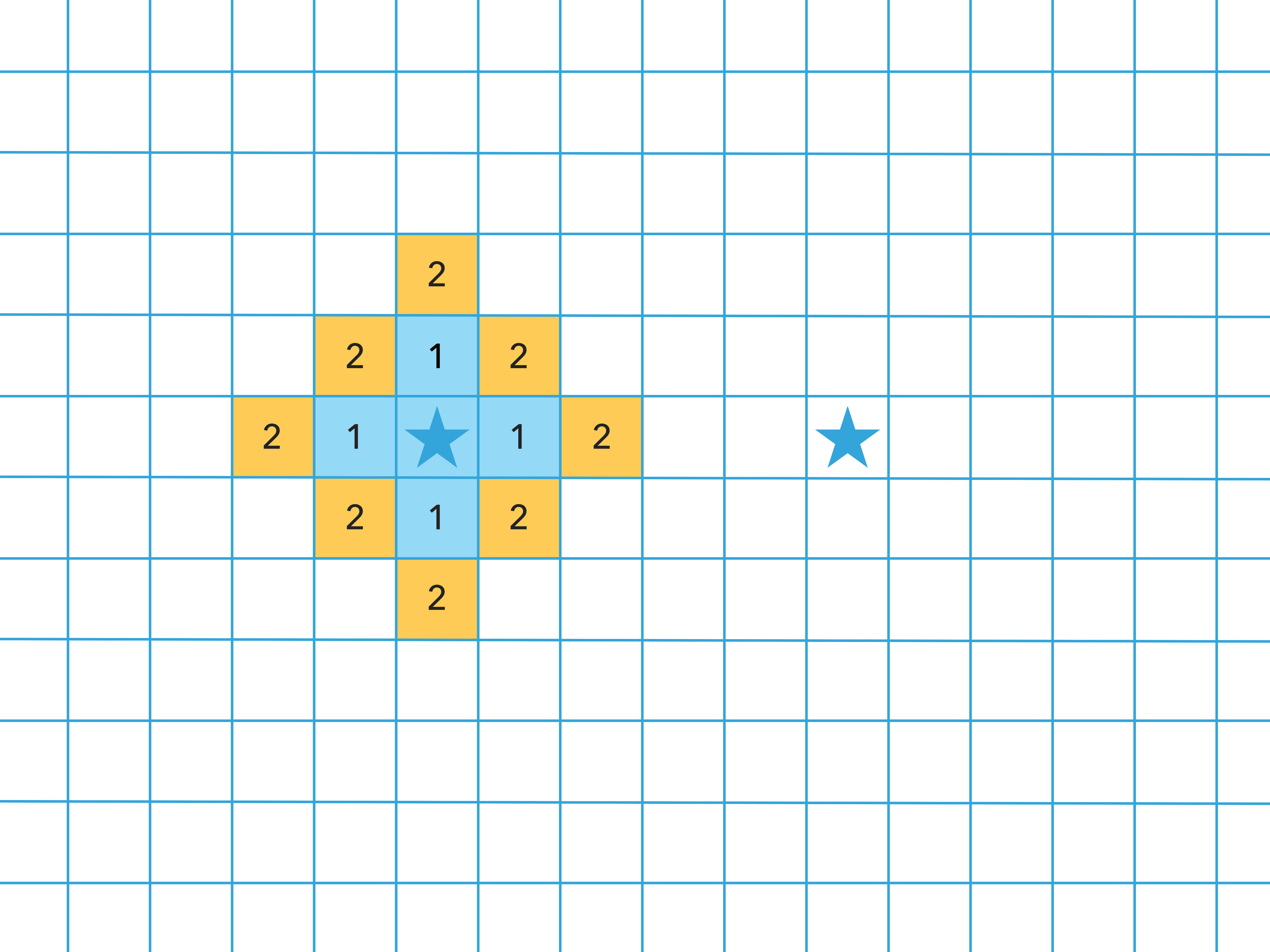

2
2
1
2
2
1
1
2
1

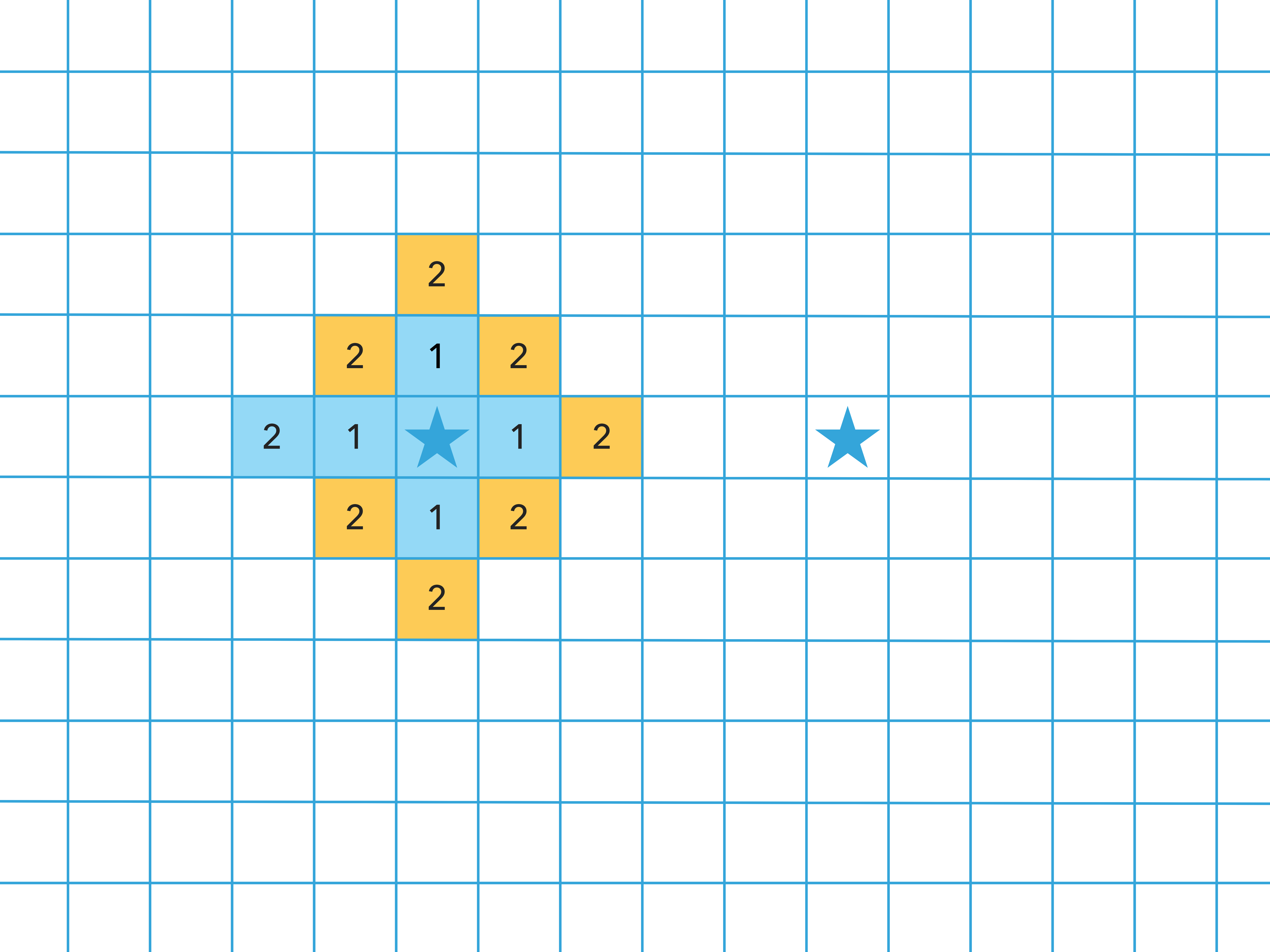

2
2
1
2
2
1
1
2
1



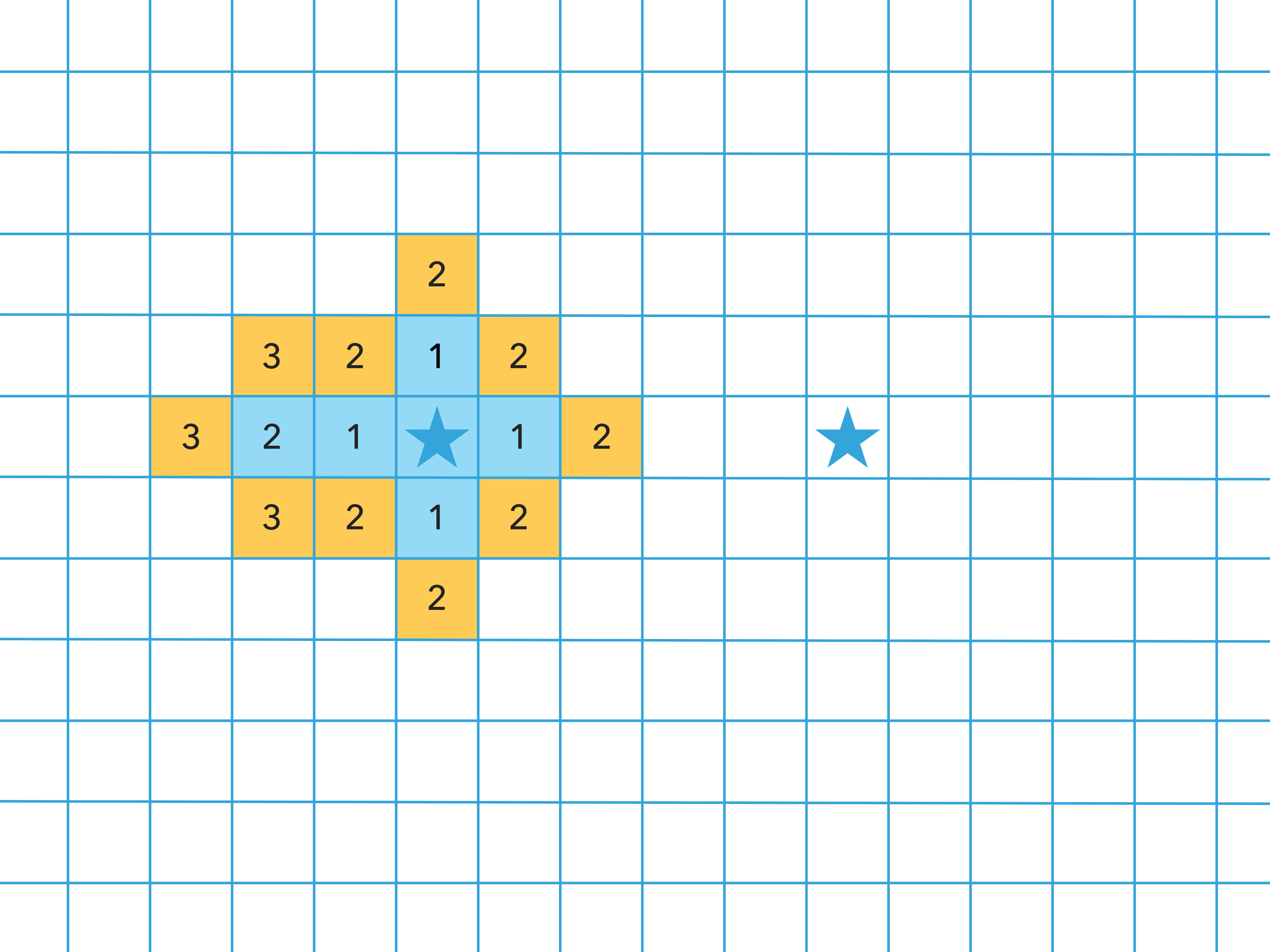
2
2
1
2
2
1
1
2
1
2
2

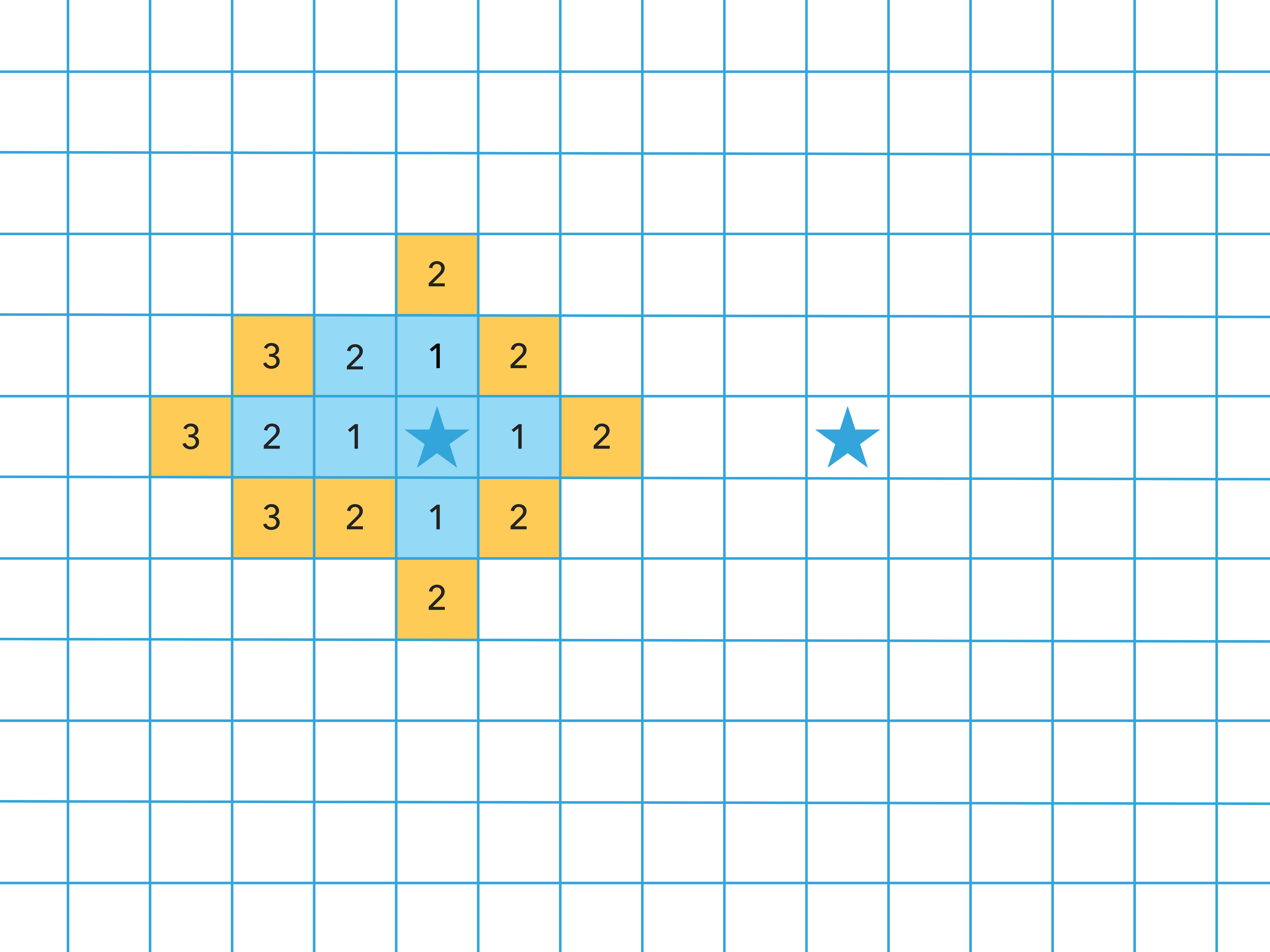

2
2
1
2
2
1
1
2
1
2
2

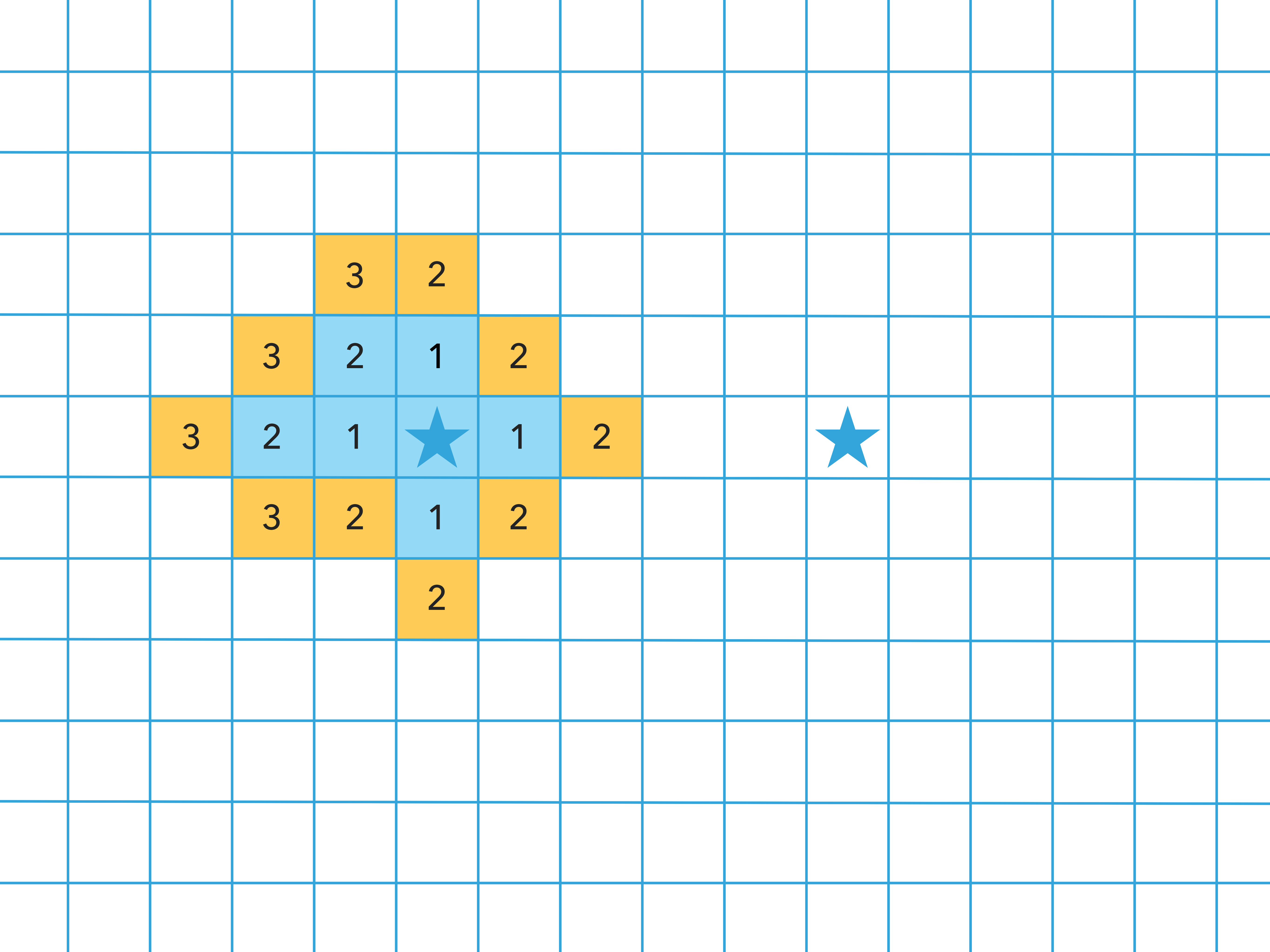

2
2
1
2
2
1
1
2
2
1
2
2

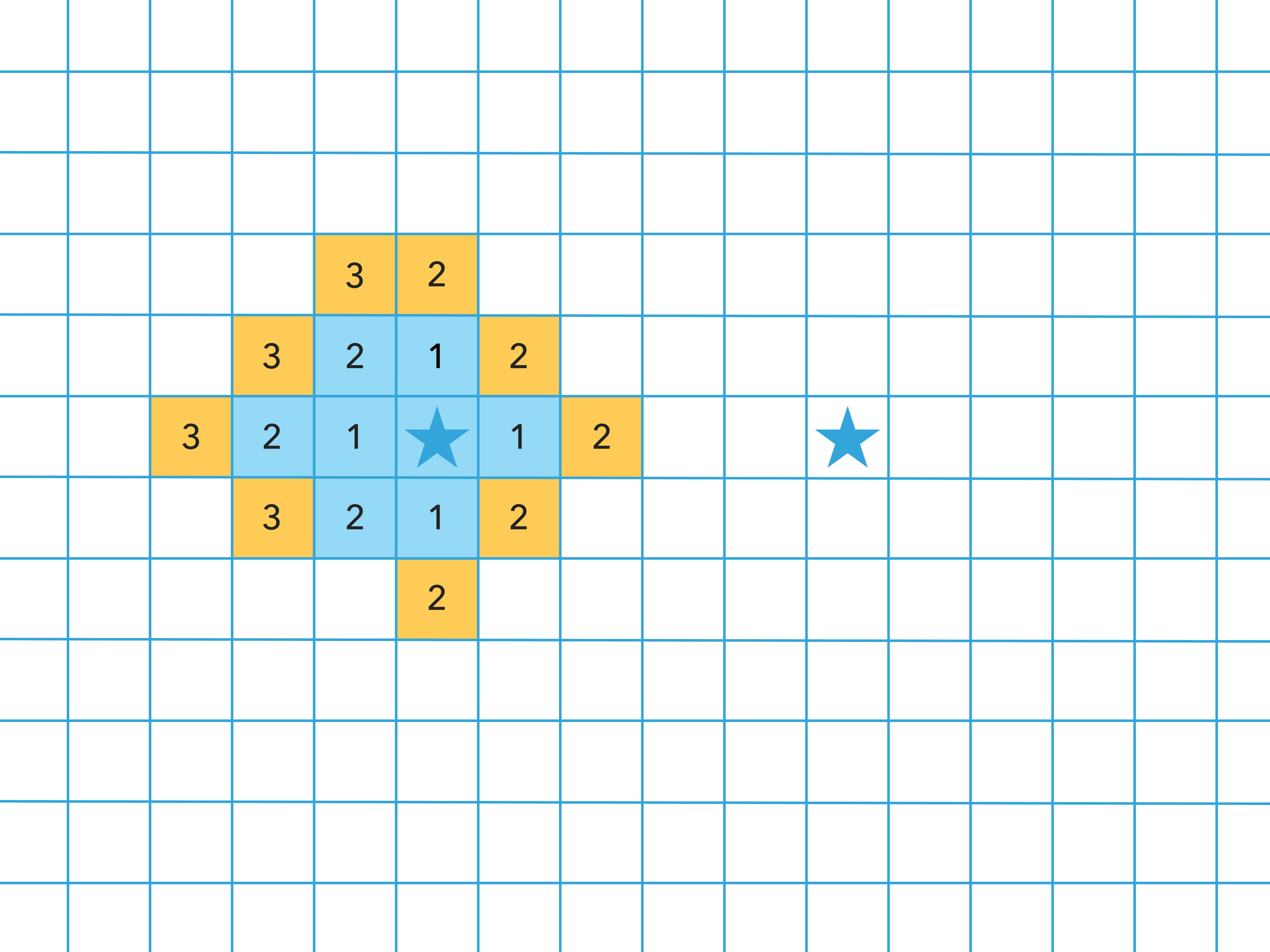


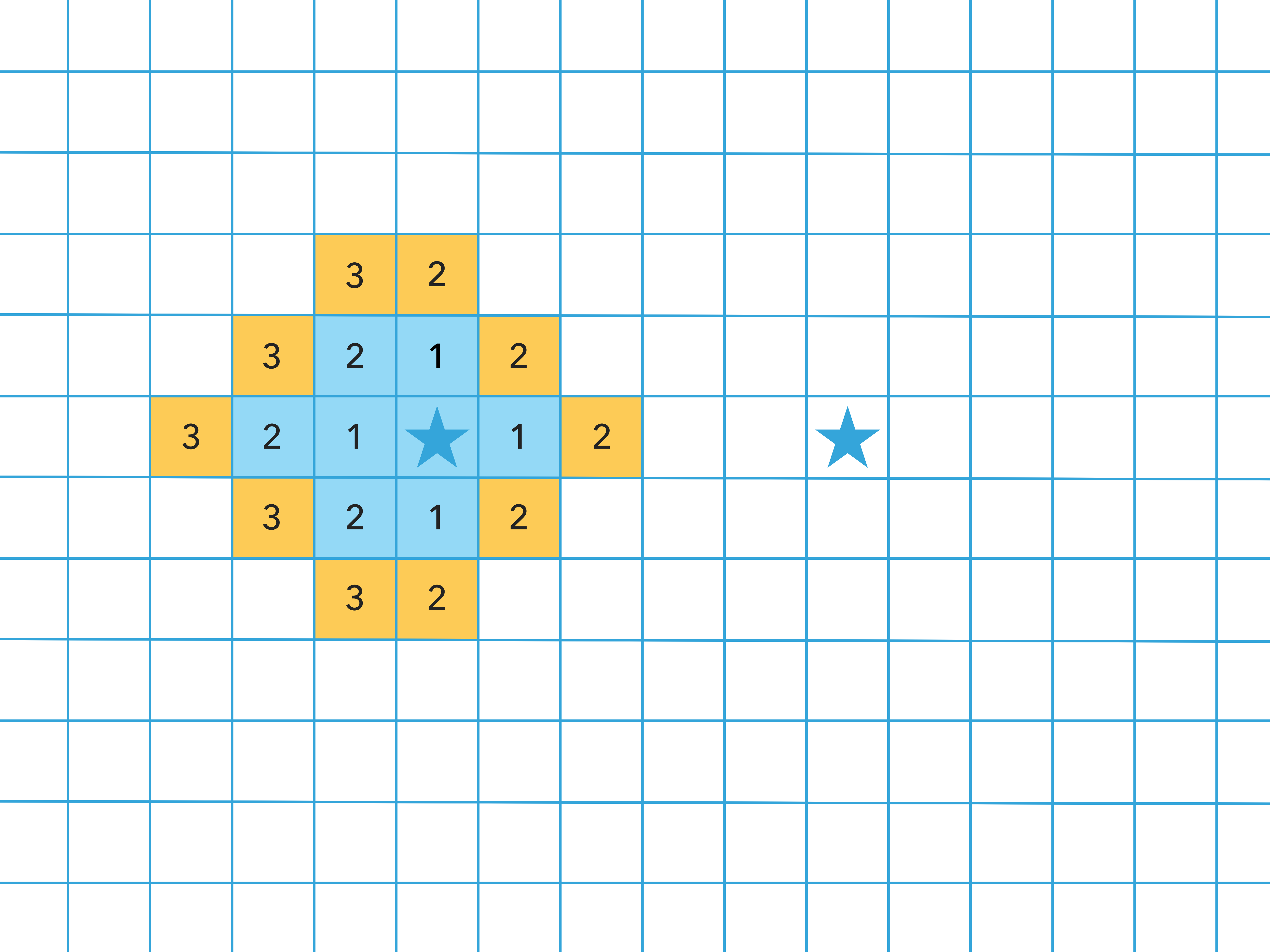
2
2
1
2
2
1
1
2
2
1
2
2

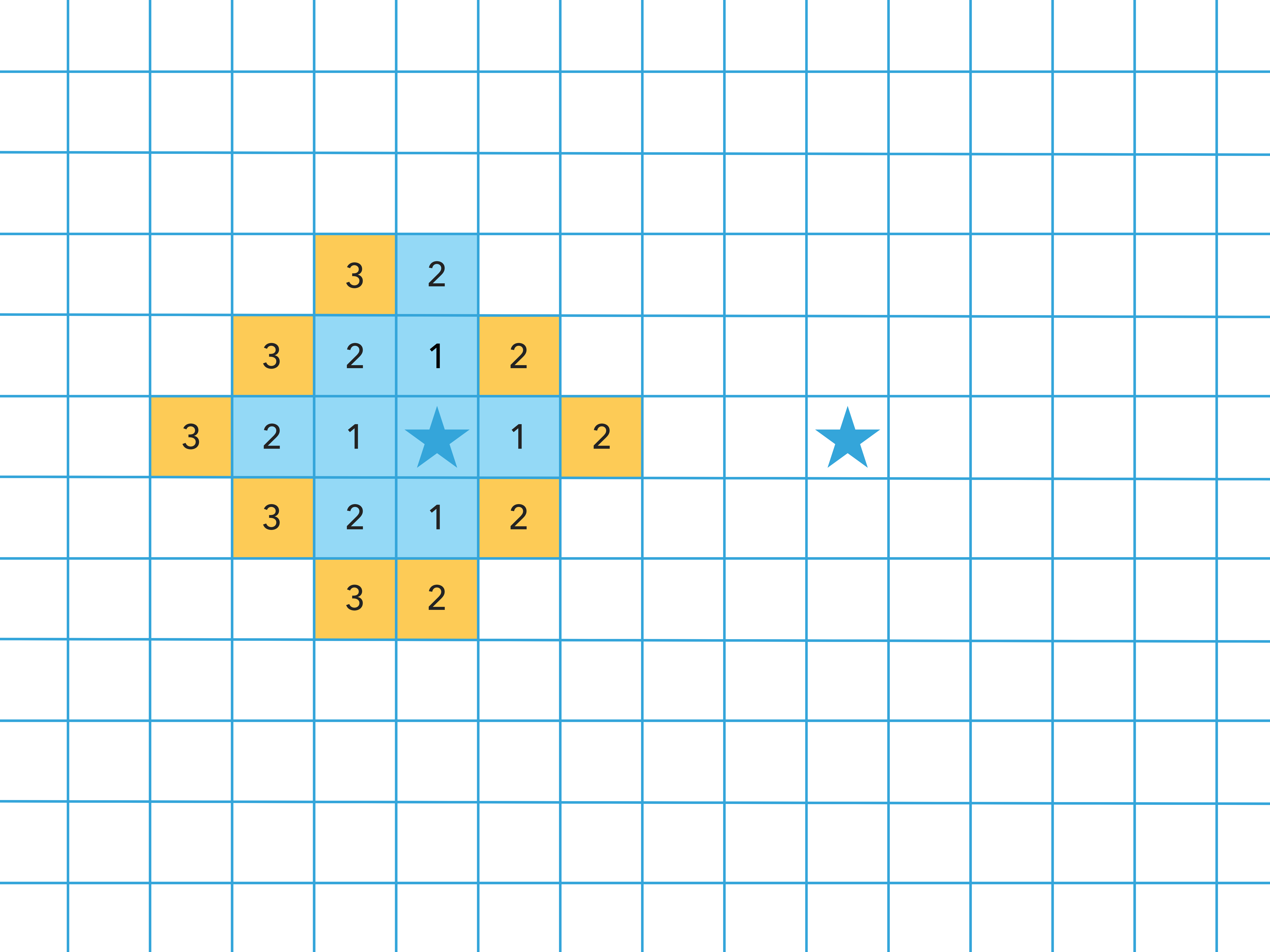

2
3 2 1 2
3 2 1 1 2
3 2 1 2
2

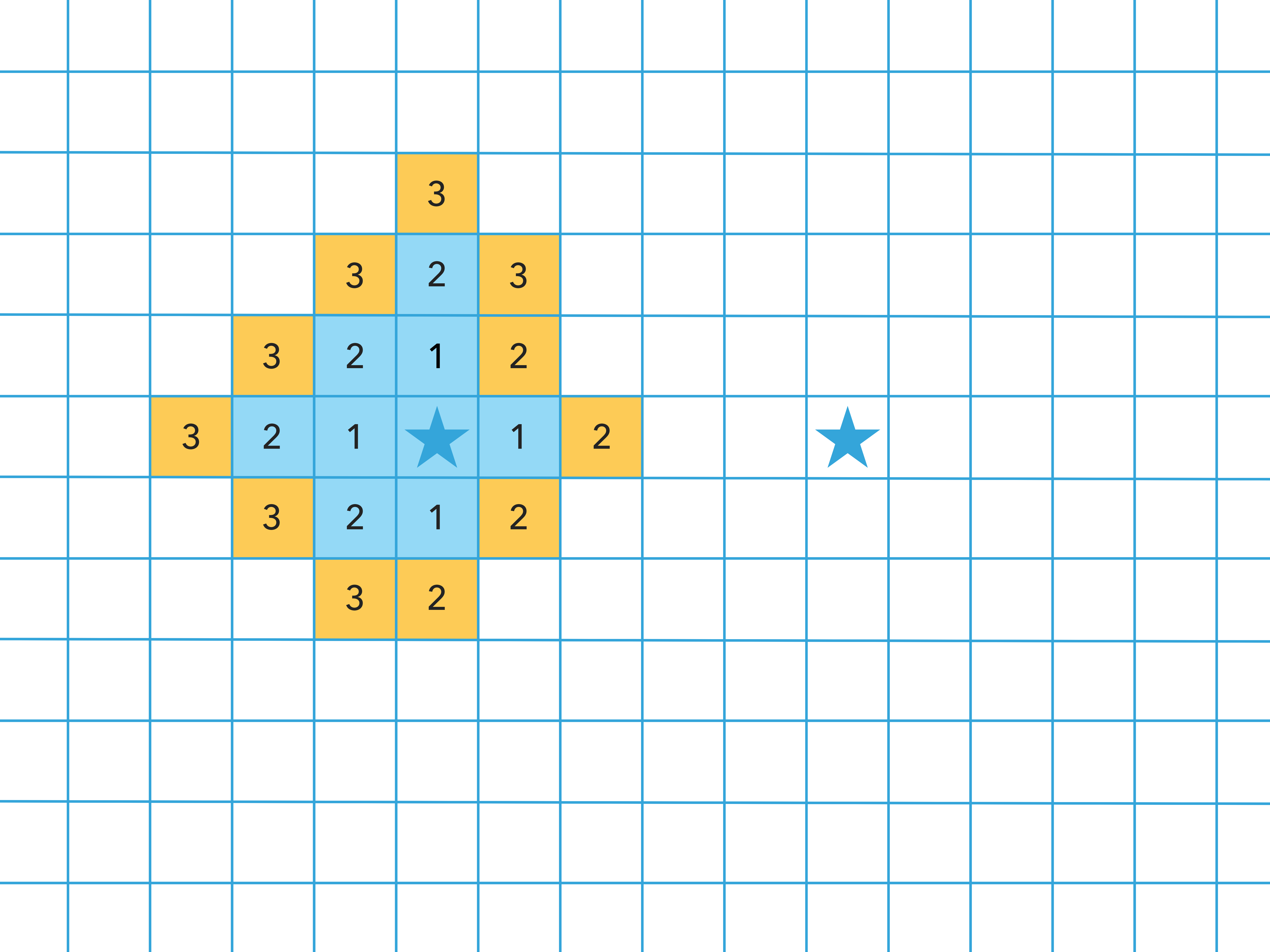

2
3 2 1 2
3 2 1 1 2
3 2 1 2
2

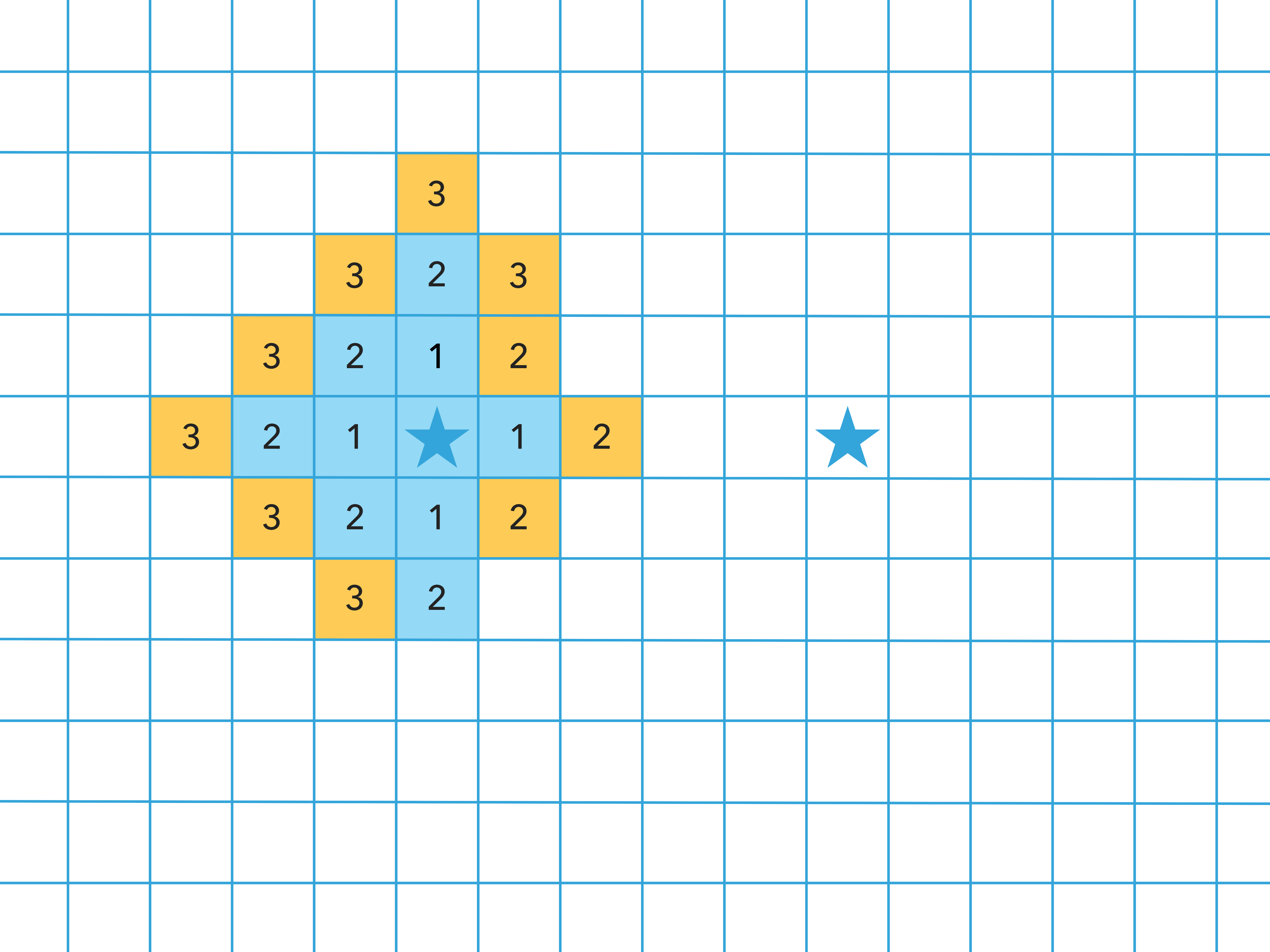

3 2
3 2 1 2
3 2 1 1 2
3 2 1 2
2

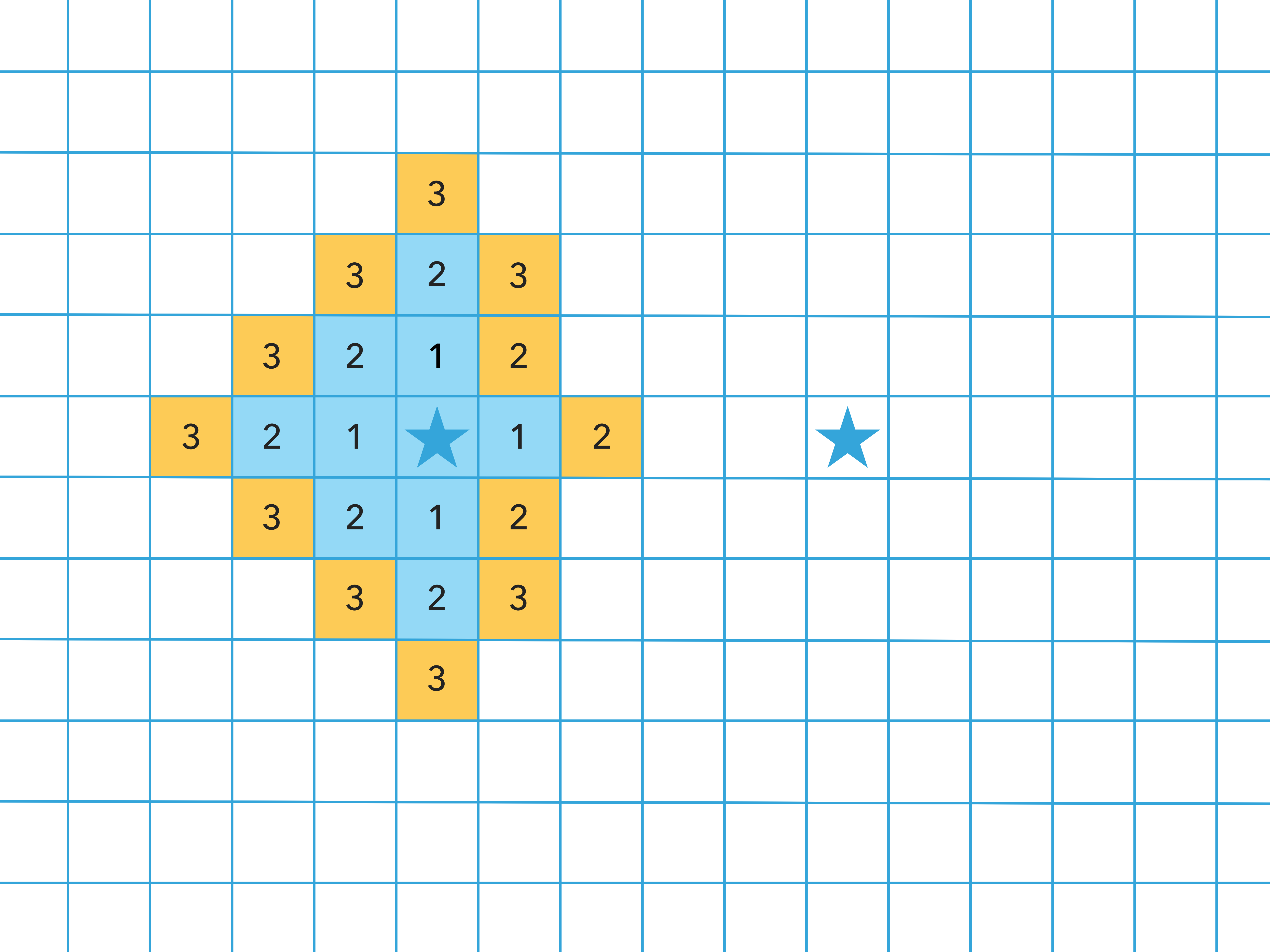

3 2
3 2 1 2
3 2 1 1 2
3 2 1 2
2

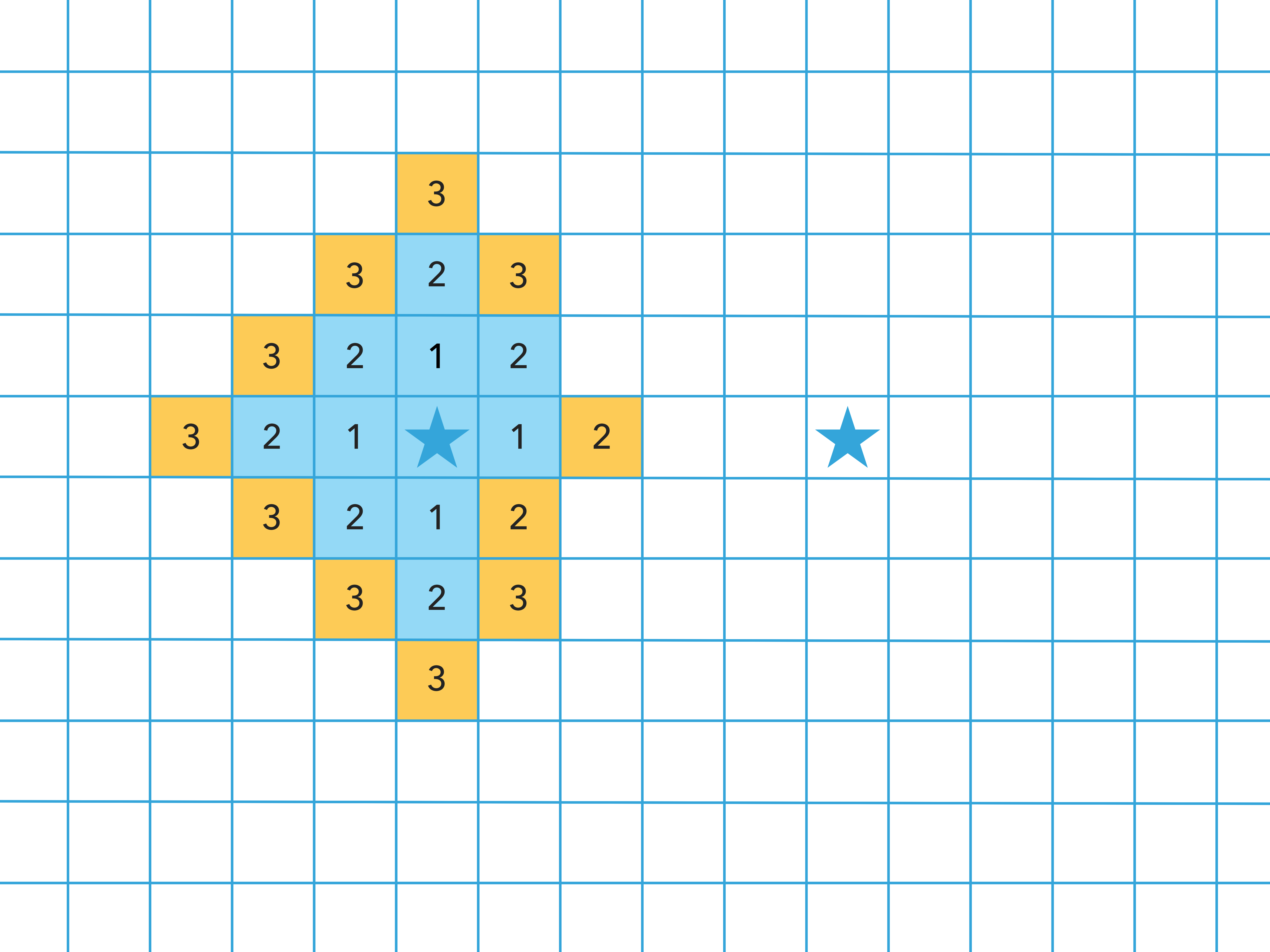

3 2
3 2 1 2
3 2 1 1 2
3 2 1 2
3 2

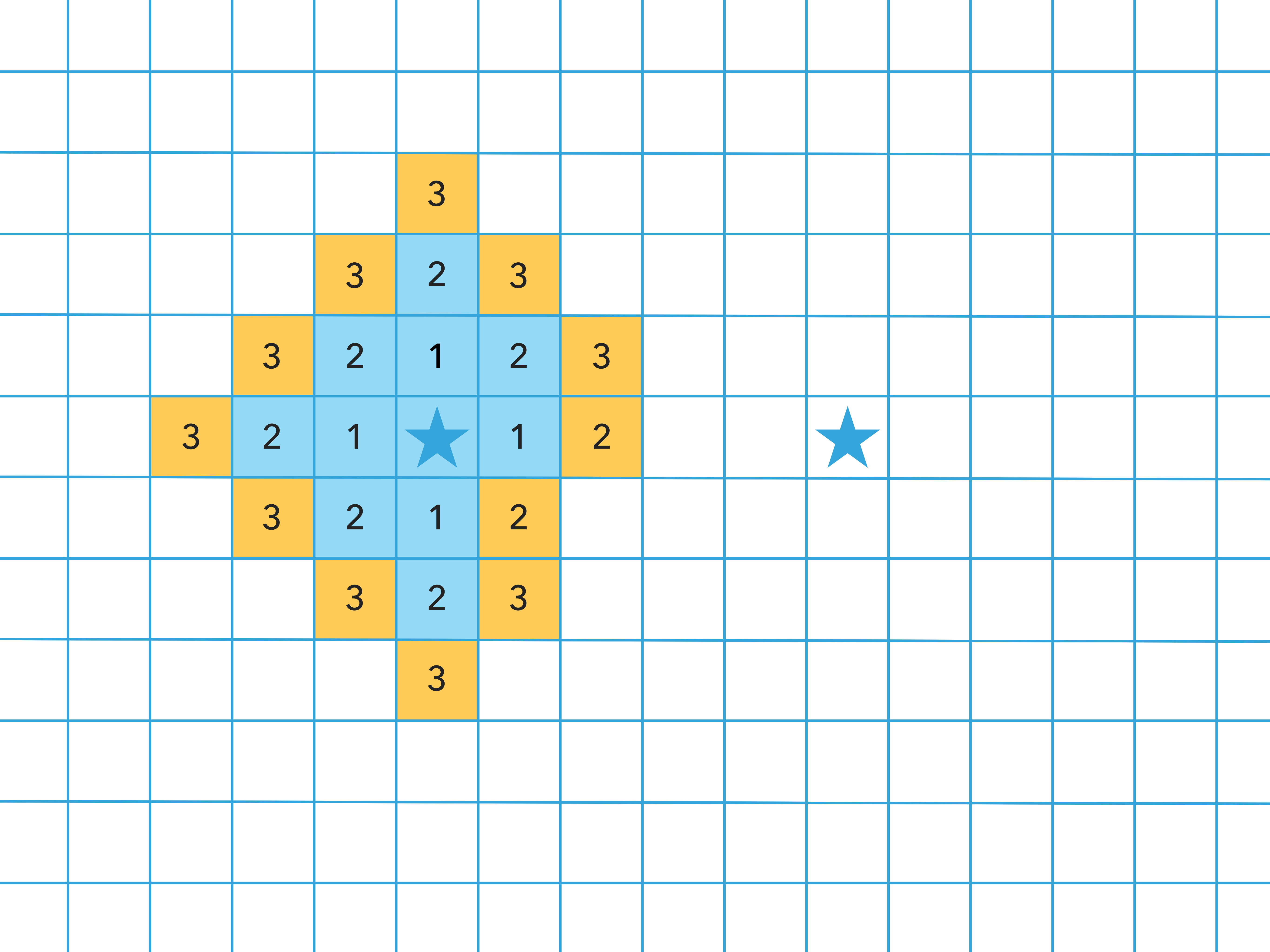

3 2
3 2 1 2
3 2 1 1 2
3 2 1 2
3 2


3
3 2 3
3 2 1 2
3 2 1 1 2
3 2 1 2
3 2

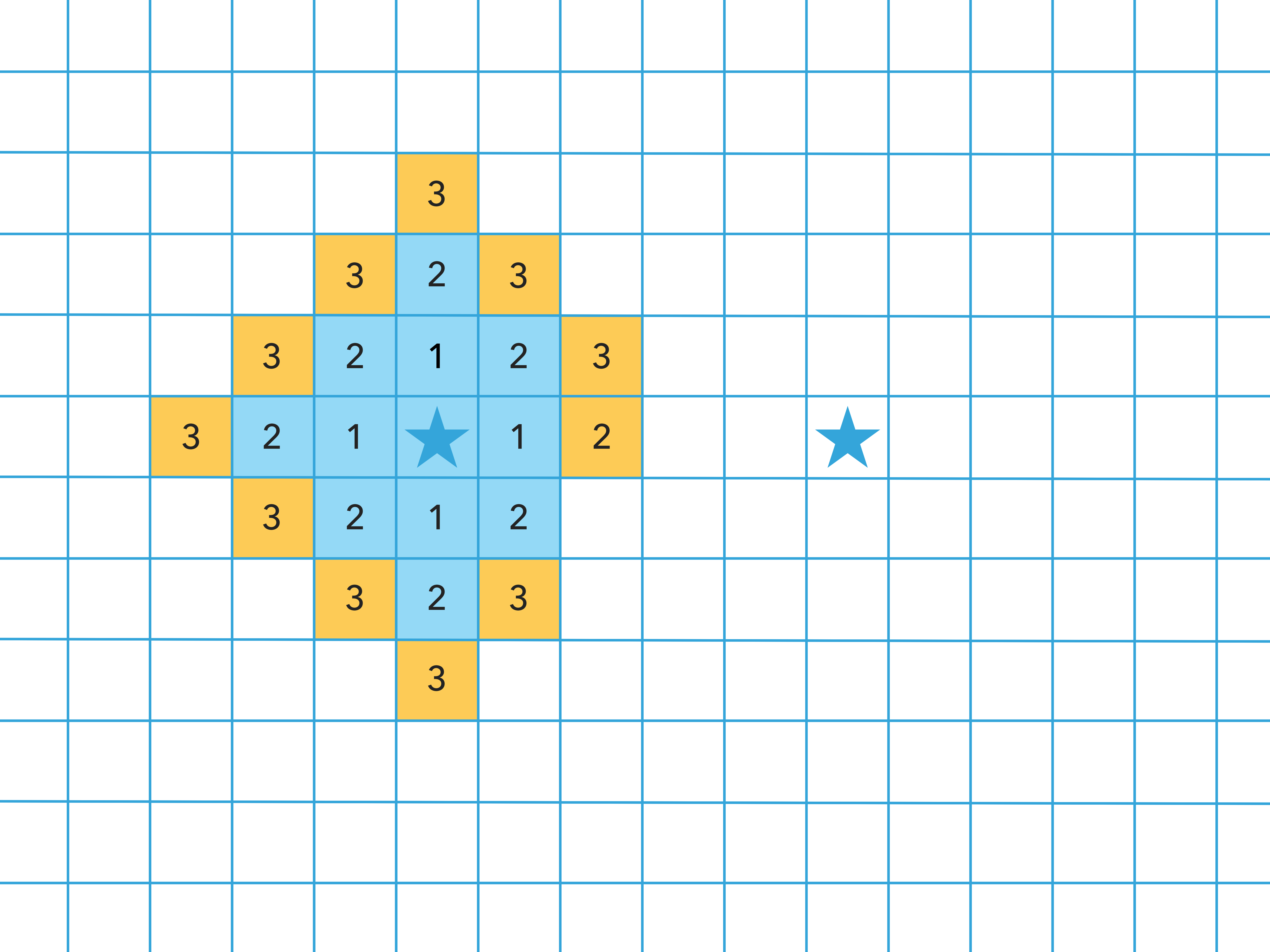

3
3
2
3
3
2
1
2
3
2
1
1
2
3
2
1
2
3
2

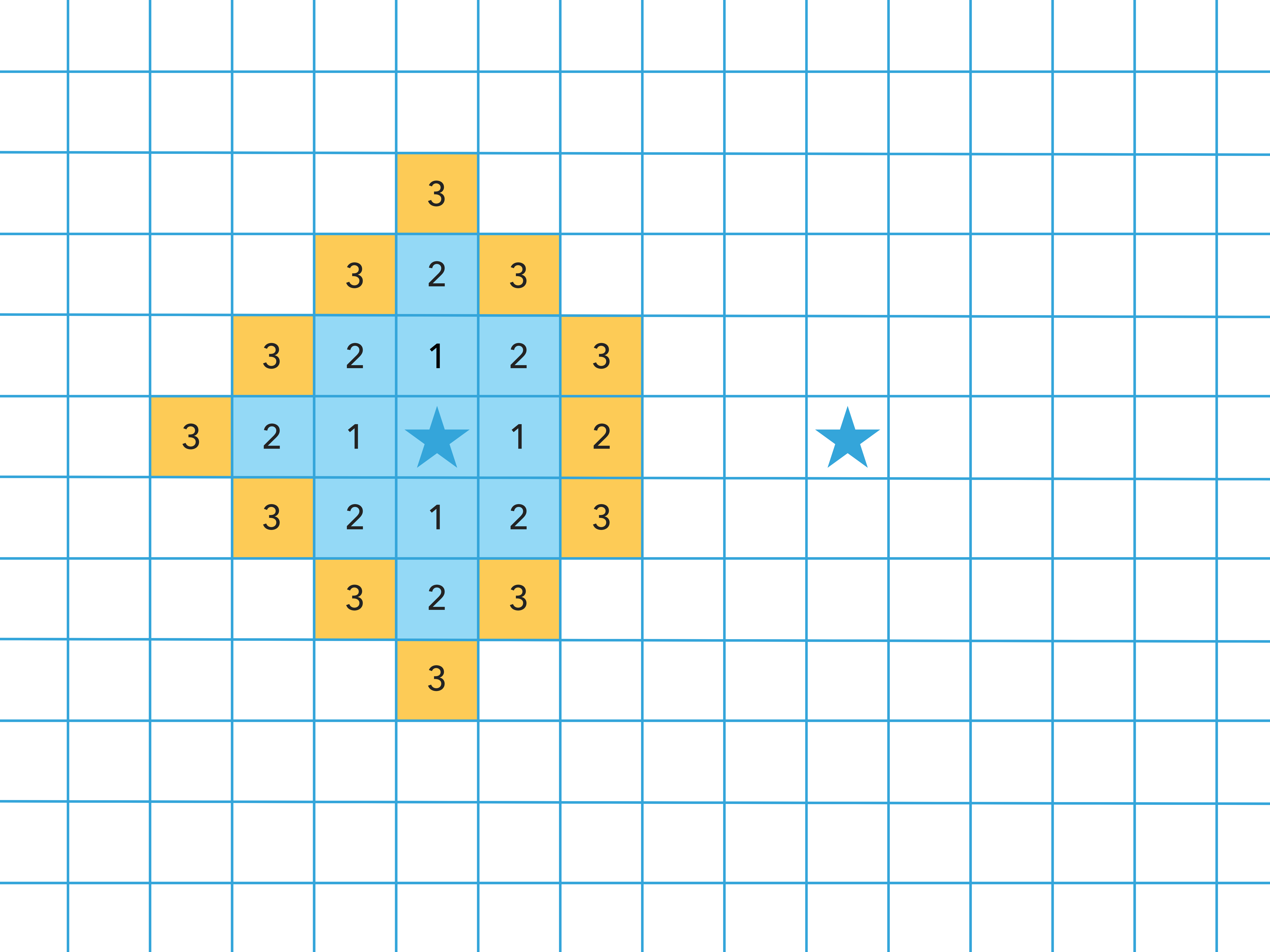

3
3
2
3
3
2
1
2
3
2
1
1
2
3
2
1
2
3
2
3
3

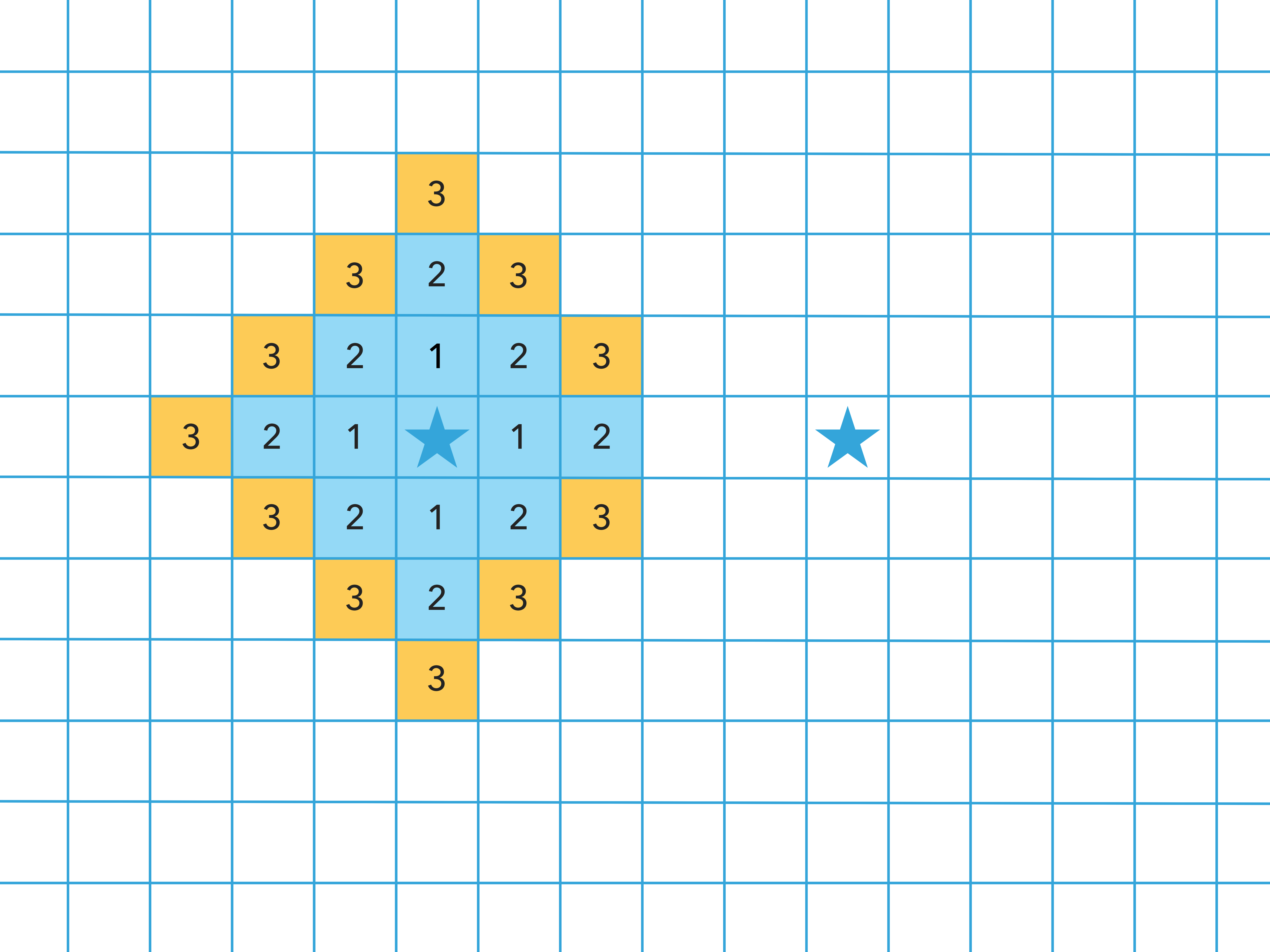

3
3
2
3
3
2
1
2
3
2
1
1
2
3
2
1
2
3
2
3
3

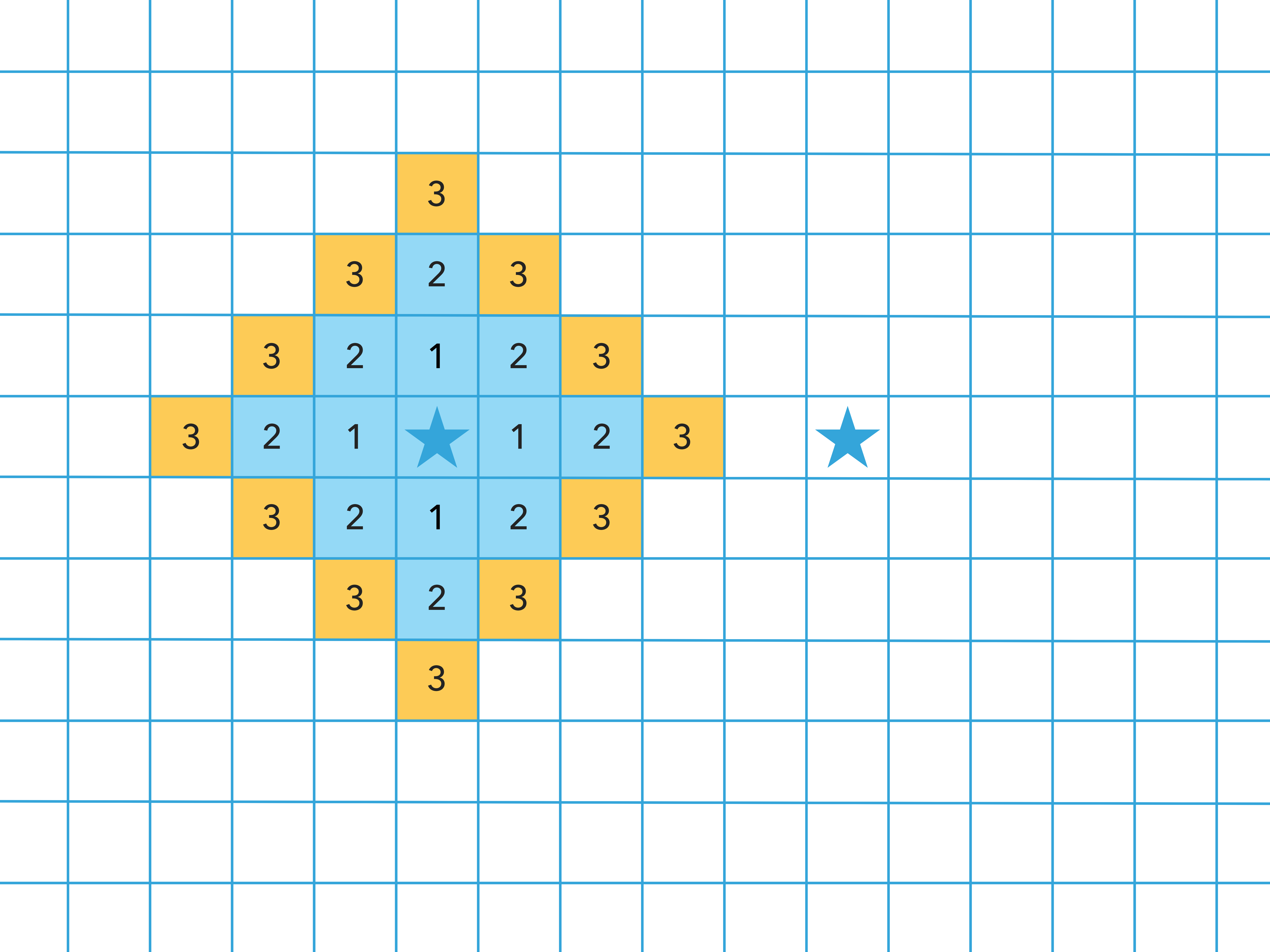


3
3
2
3
3
2
1
2
3
3
2
1
1
2
3
2
1
2
3
2
3
3

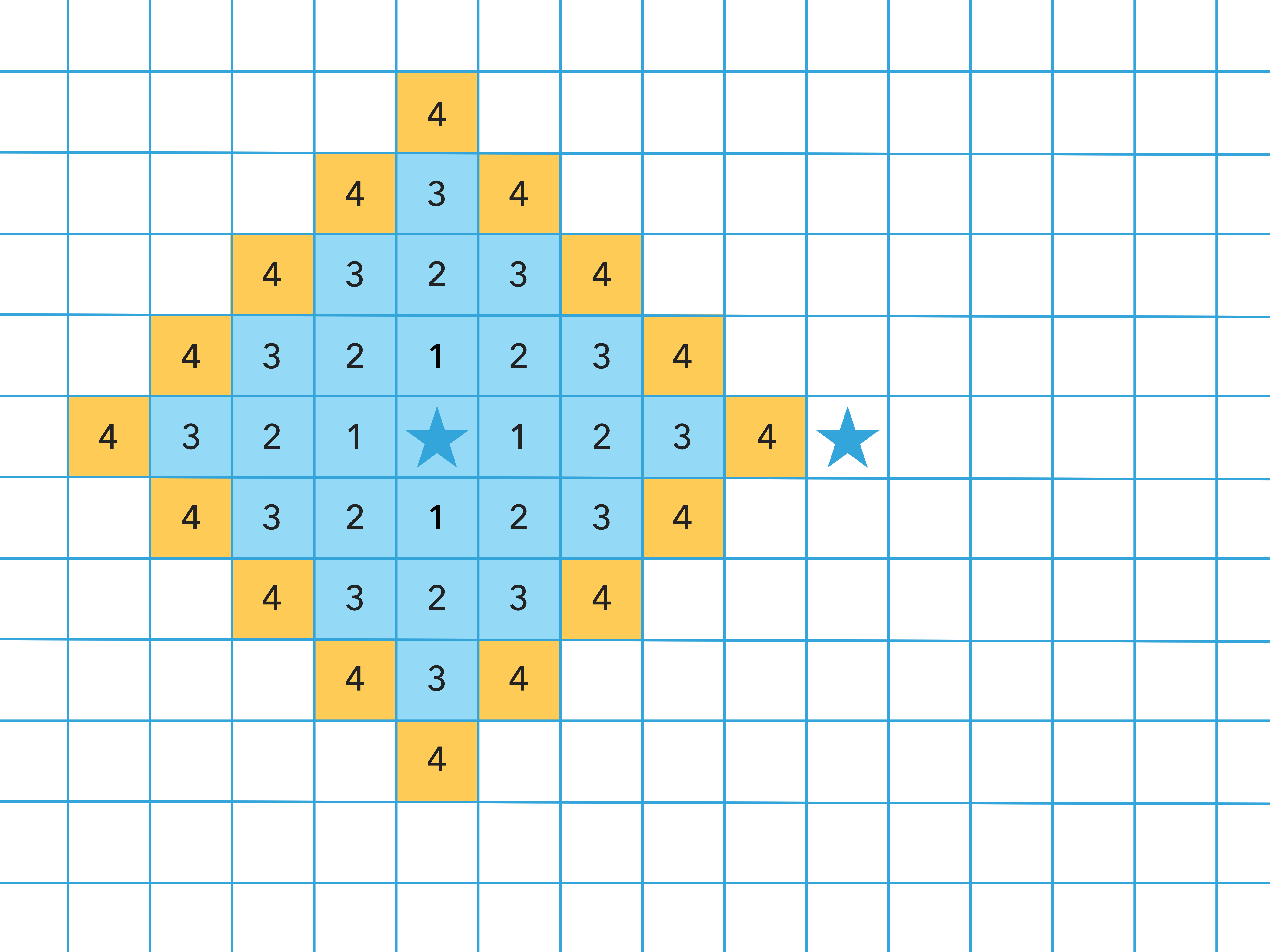

3
3 2 3
3 2 1 2 3
3 2 1 1 2
3 2 1 2
3 2 3
3

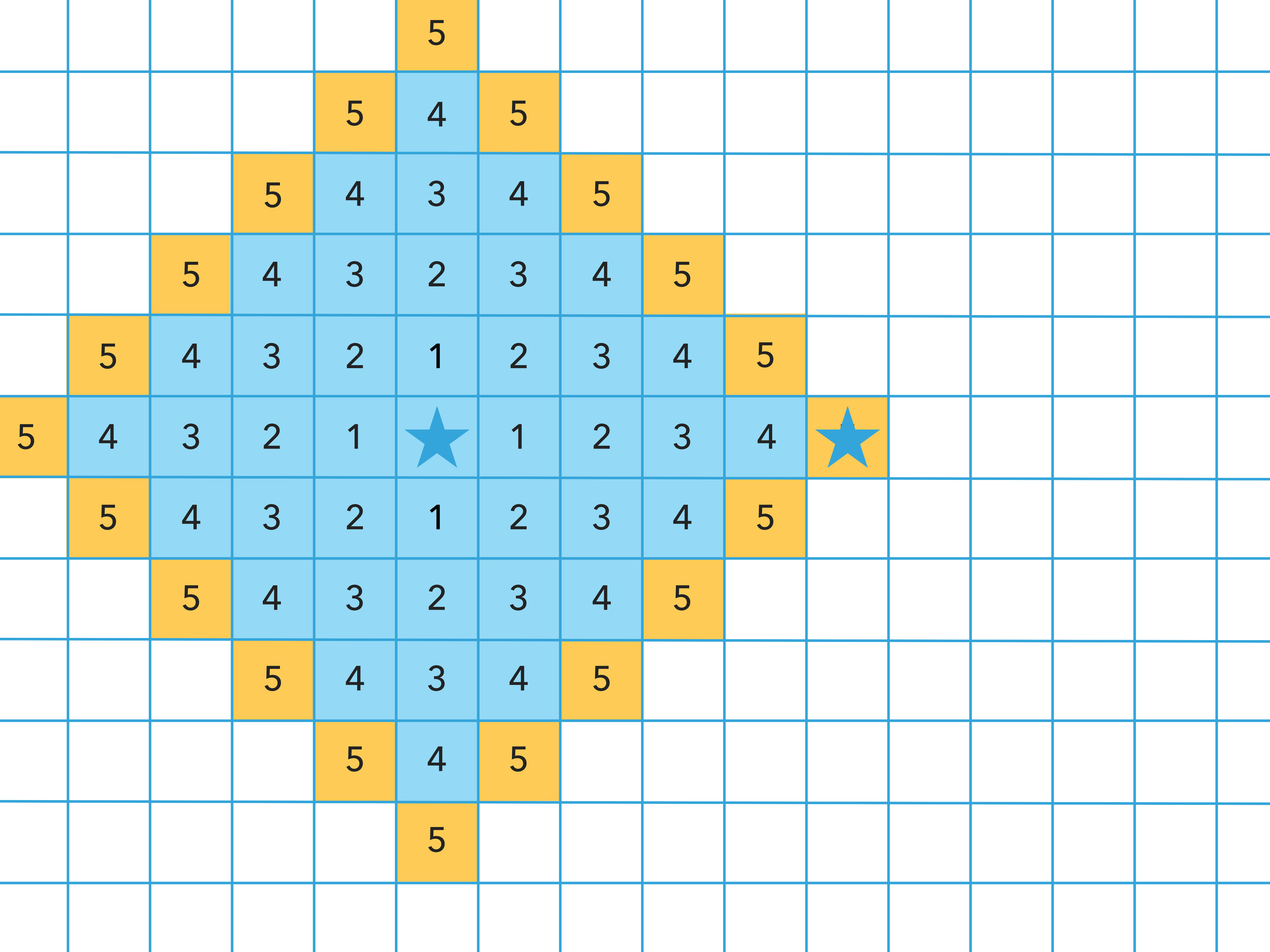

3
3 2 3
3 2 1 2 3
3 2 1 1 2
3 2 1 2 3
3 2 3
3

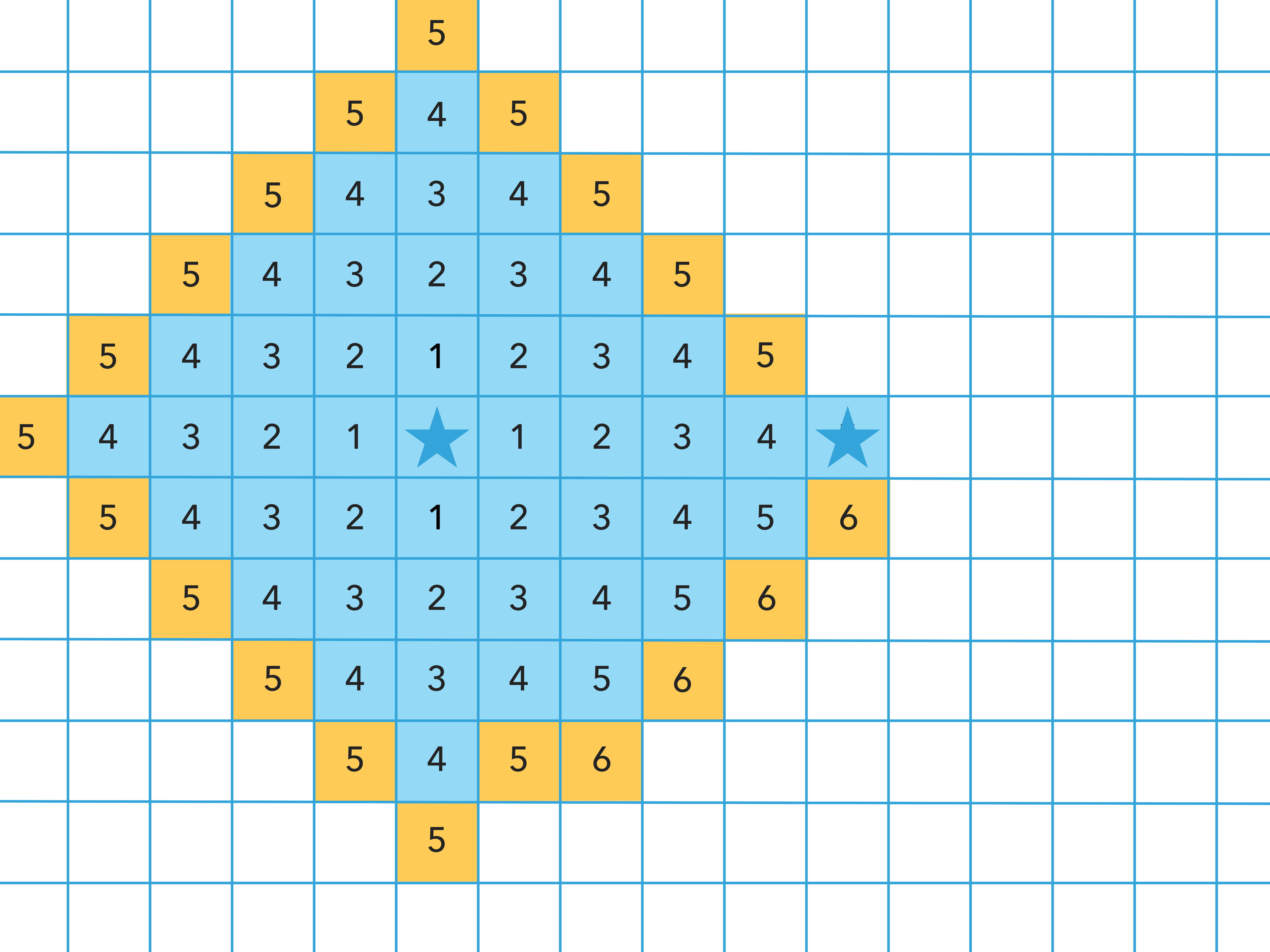

3
3
2
3
3
2
1
2
3
3
2
1
1
2
3
2
1
2
3
3
2
3
3



3
3
2
3
3
2
1
2
3
3
2
1
1
2
3
3
2
1
2
3
3
2
3
3


4
4 3 4
4 3 2 3 4
4 3 2 1 2 3 4
4 3 2 1 1 2 3 4
4 3 2 1 2 3 4
4 3 2 3 4
4 3 4
4


5
5 4 5
5 4 3 4 5
5 4 3 2 3 4 5
5 4 3 2 1 2 3 4 5
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
5 4 3 4 5
5 4 5
5

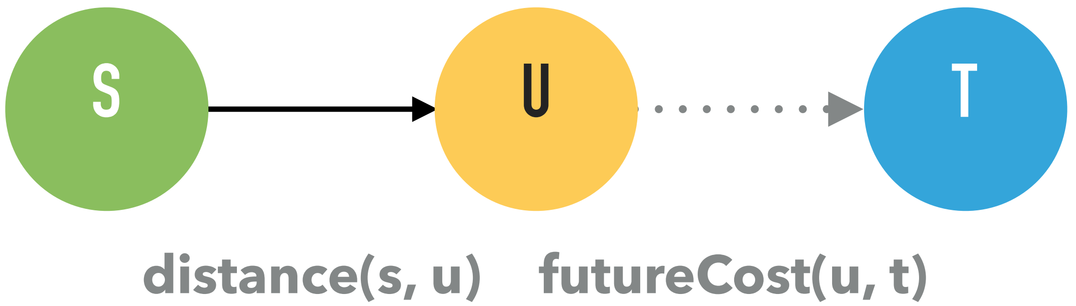


DIJKSTRA'S MEASURES THE DISTANCE FROM THE START NODE TO THE CURRENT NODE.

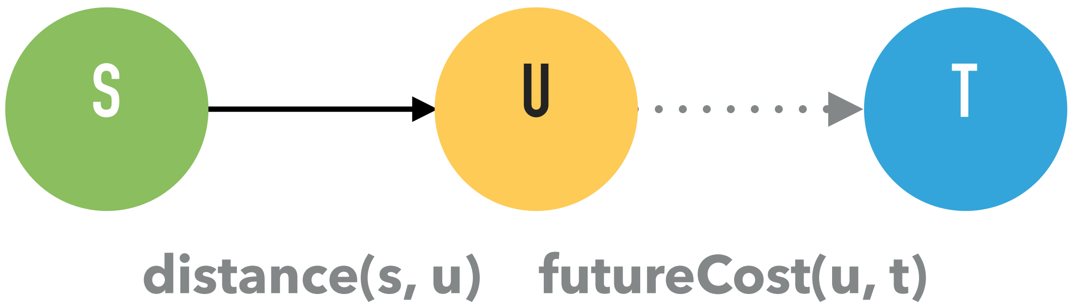
WE WANT THE DISTANCE FROM THE CURRENT NODE TO THE DESTINATION.

SEEING THE FUTURE

FORMAL DEFINITIONS



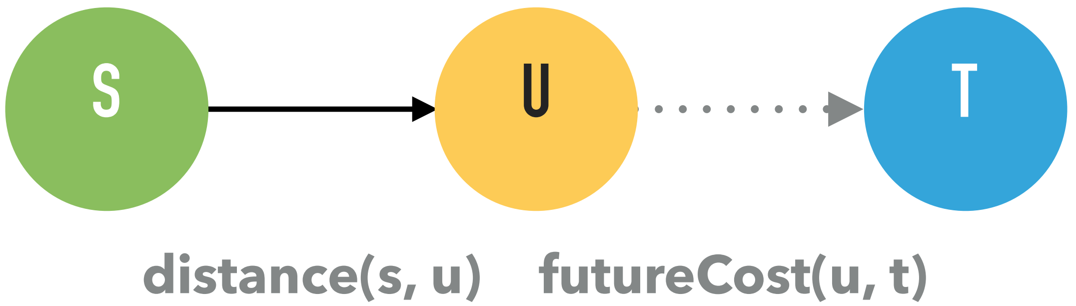
FORMAL DEFINITIONS



DIJKSTRA'S

priority(u) = distance(s, u)

FORMAL DEFINITIONS

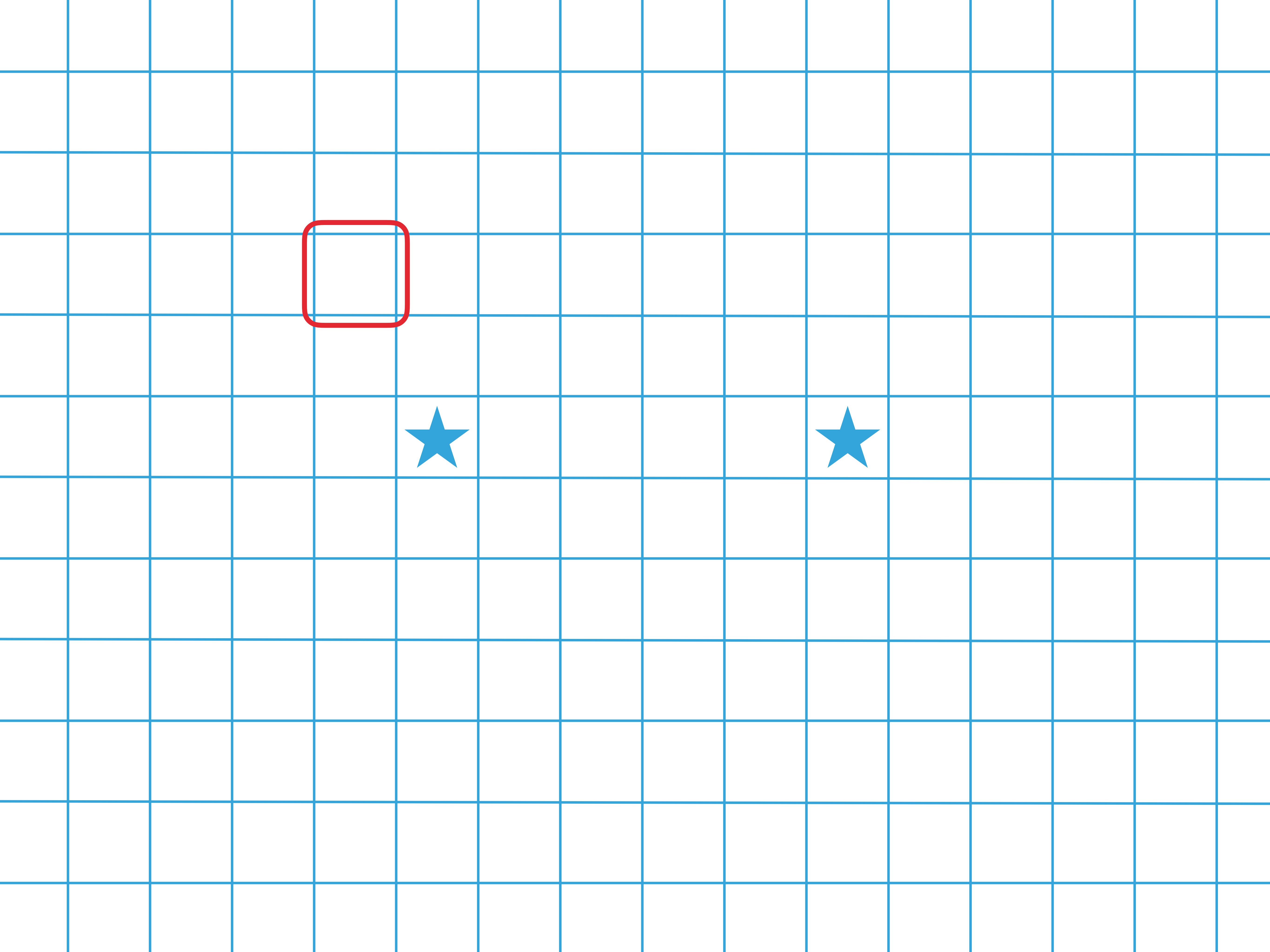


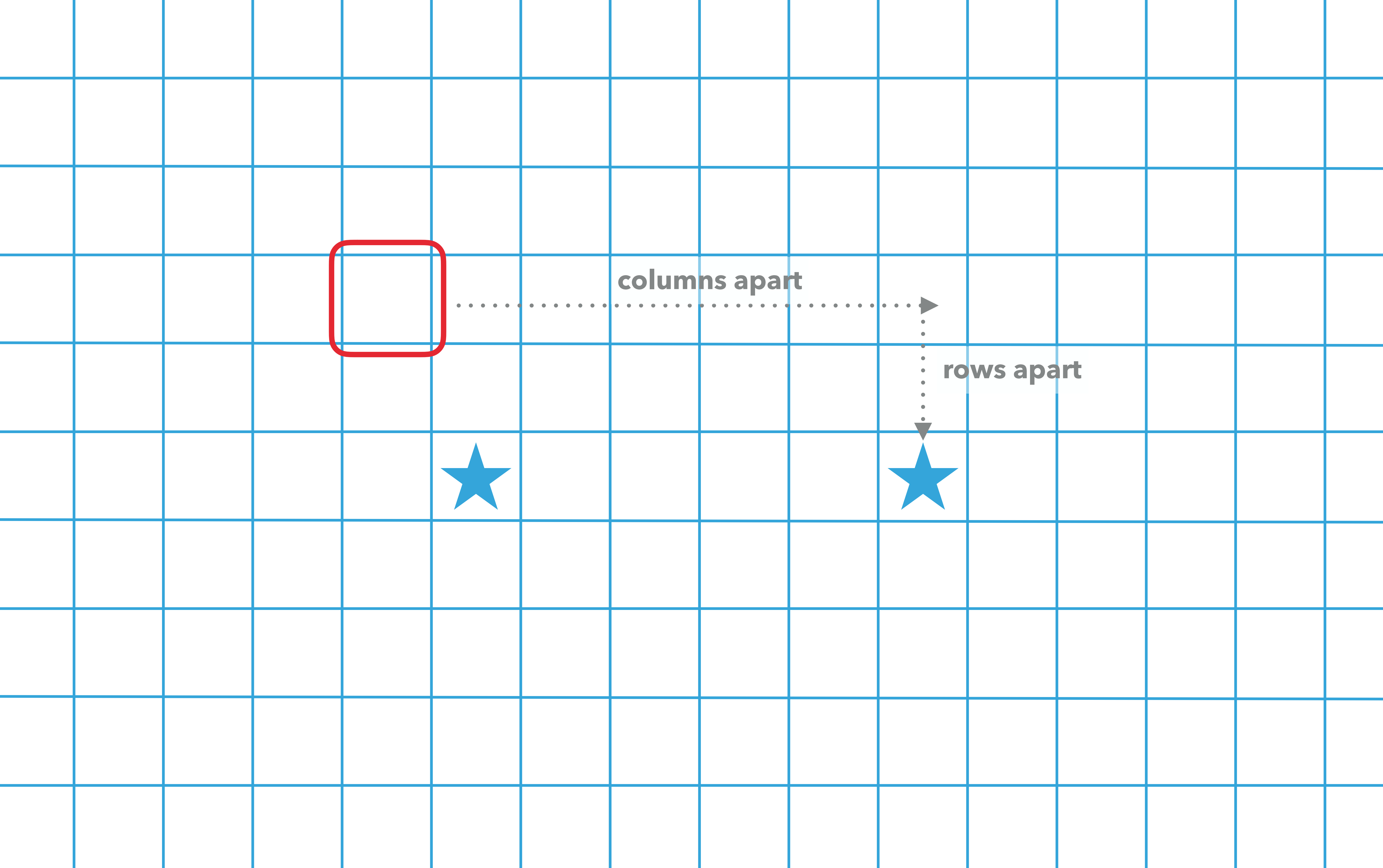
DIJKSTRA'S

$$priority(u) = distance(s, u)$$

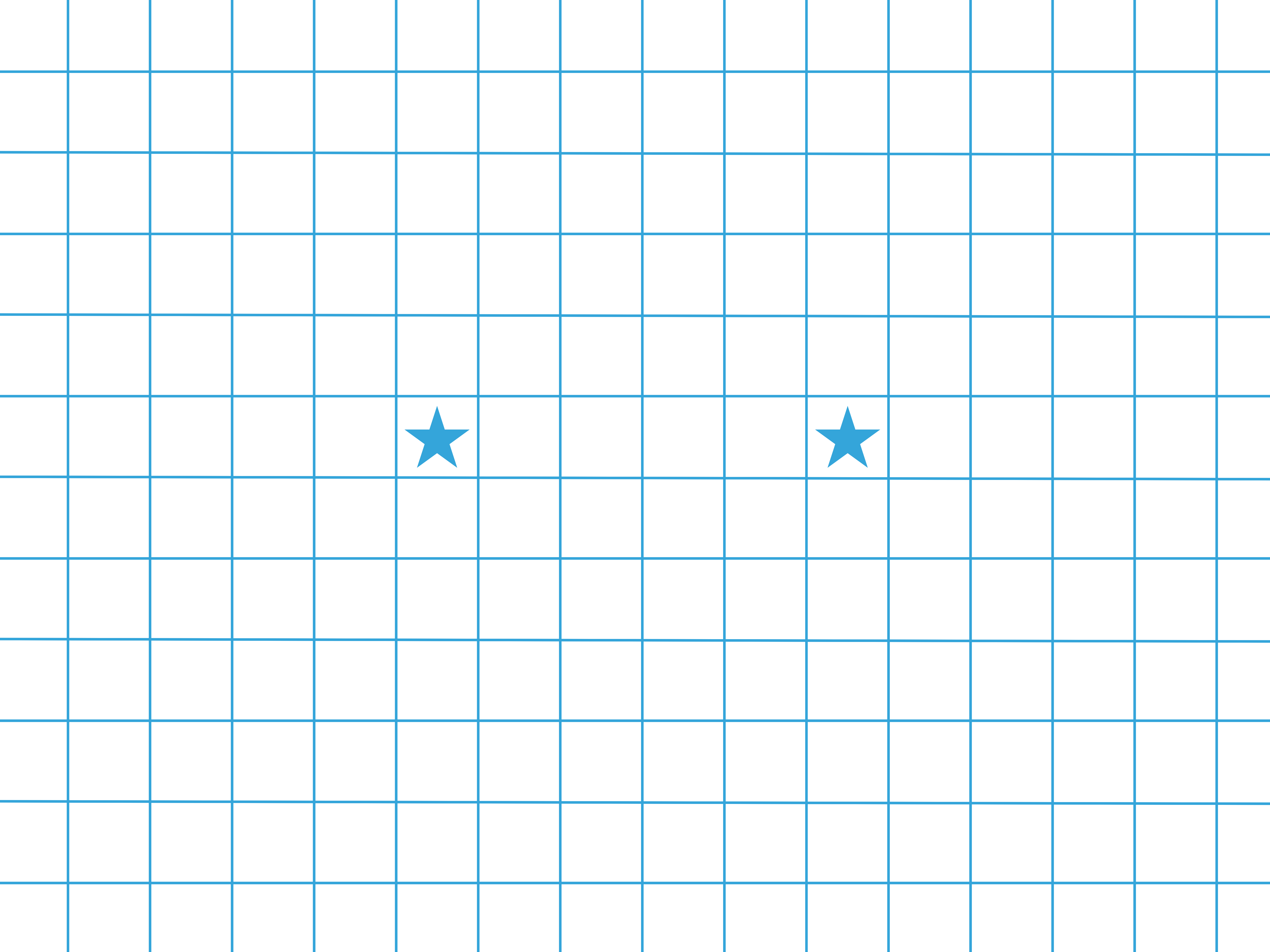
IDEAL

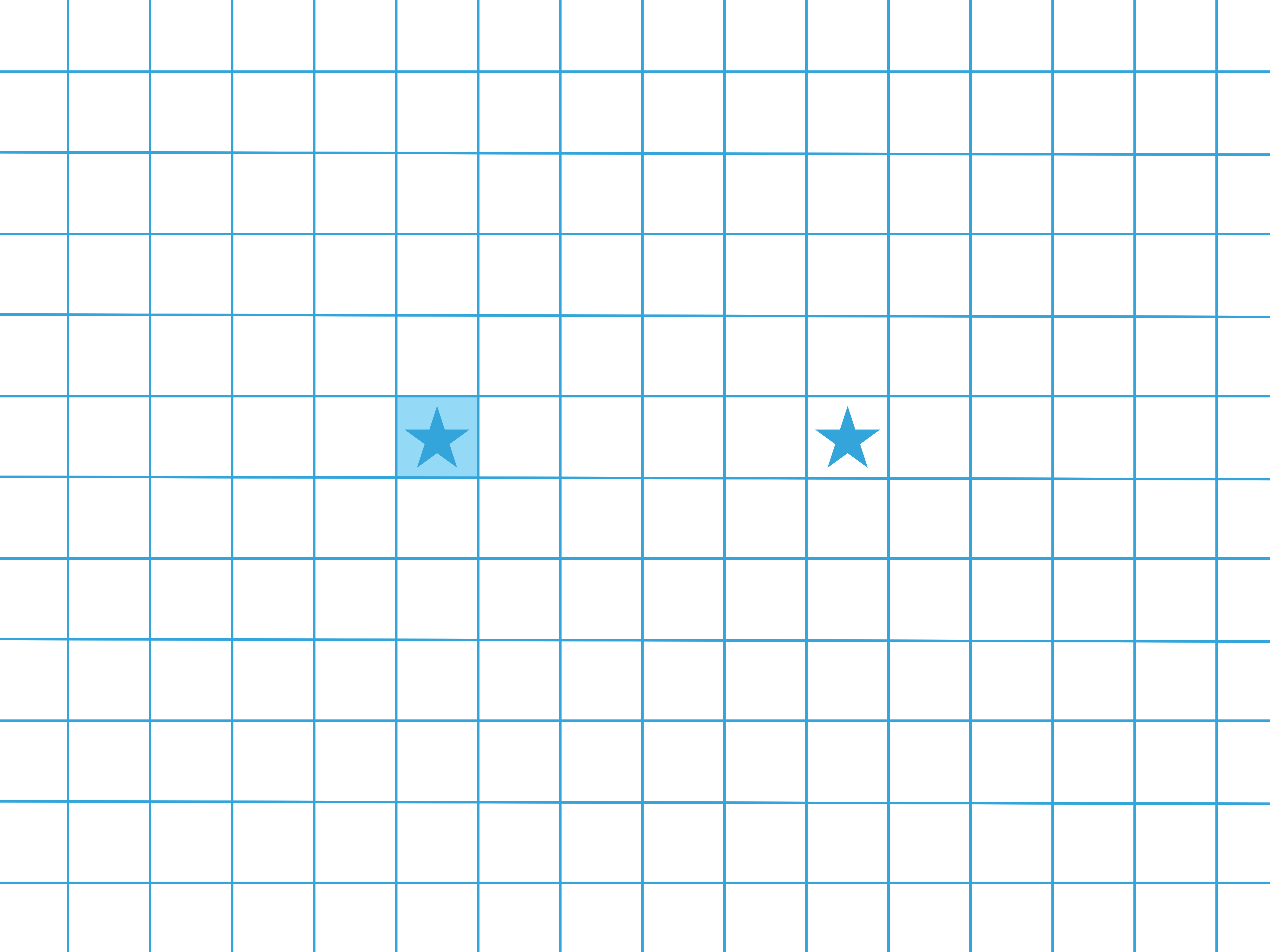
$$priority(u) = distance(s, u) + futureCost(u, t)$$

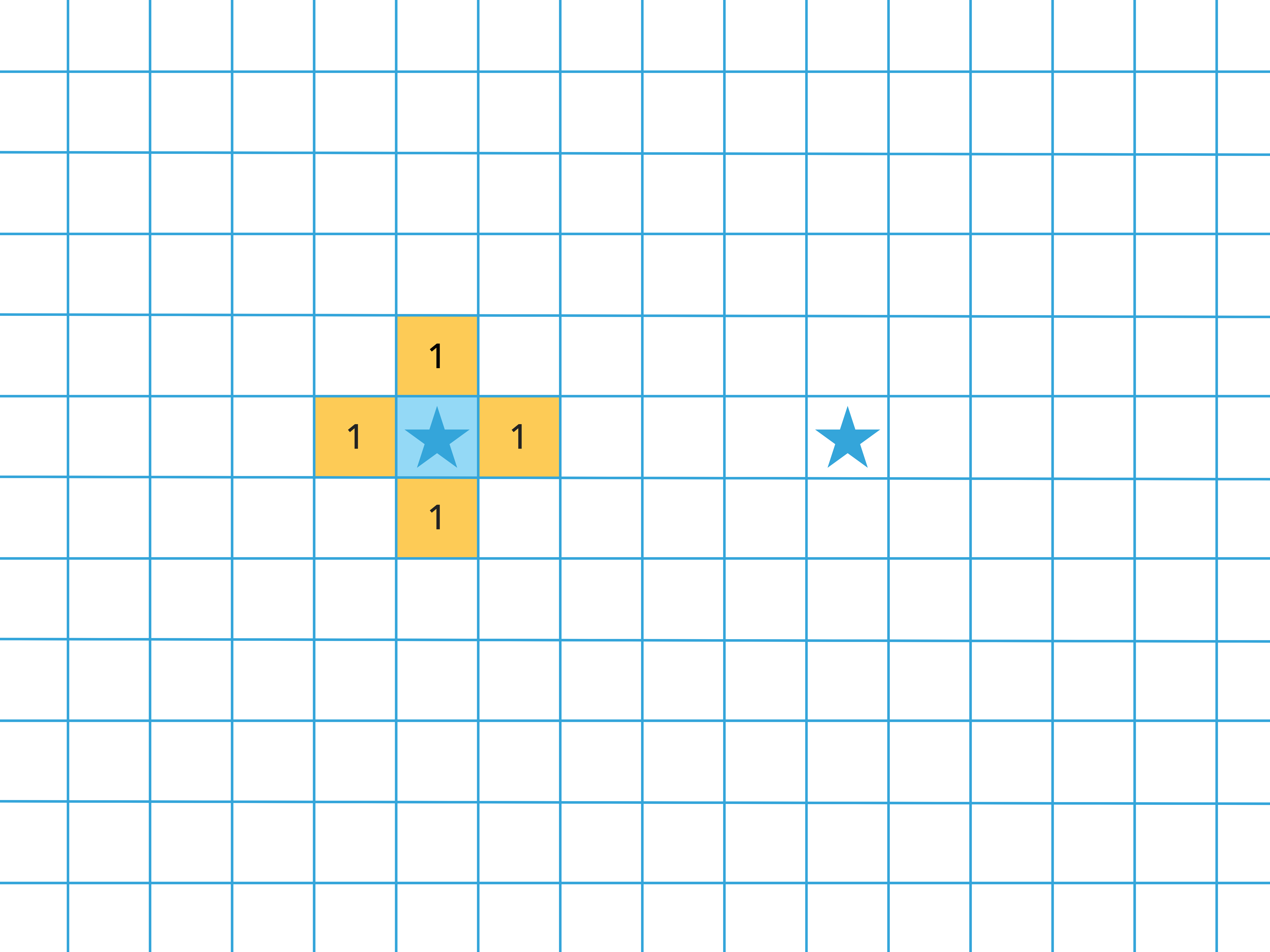


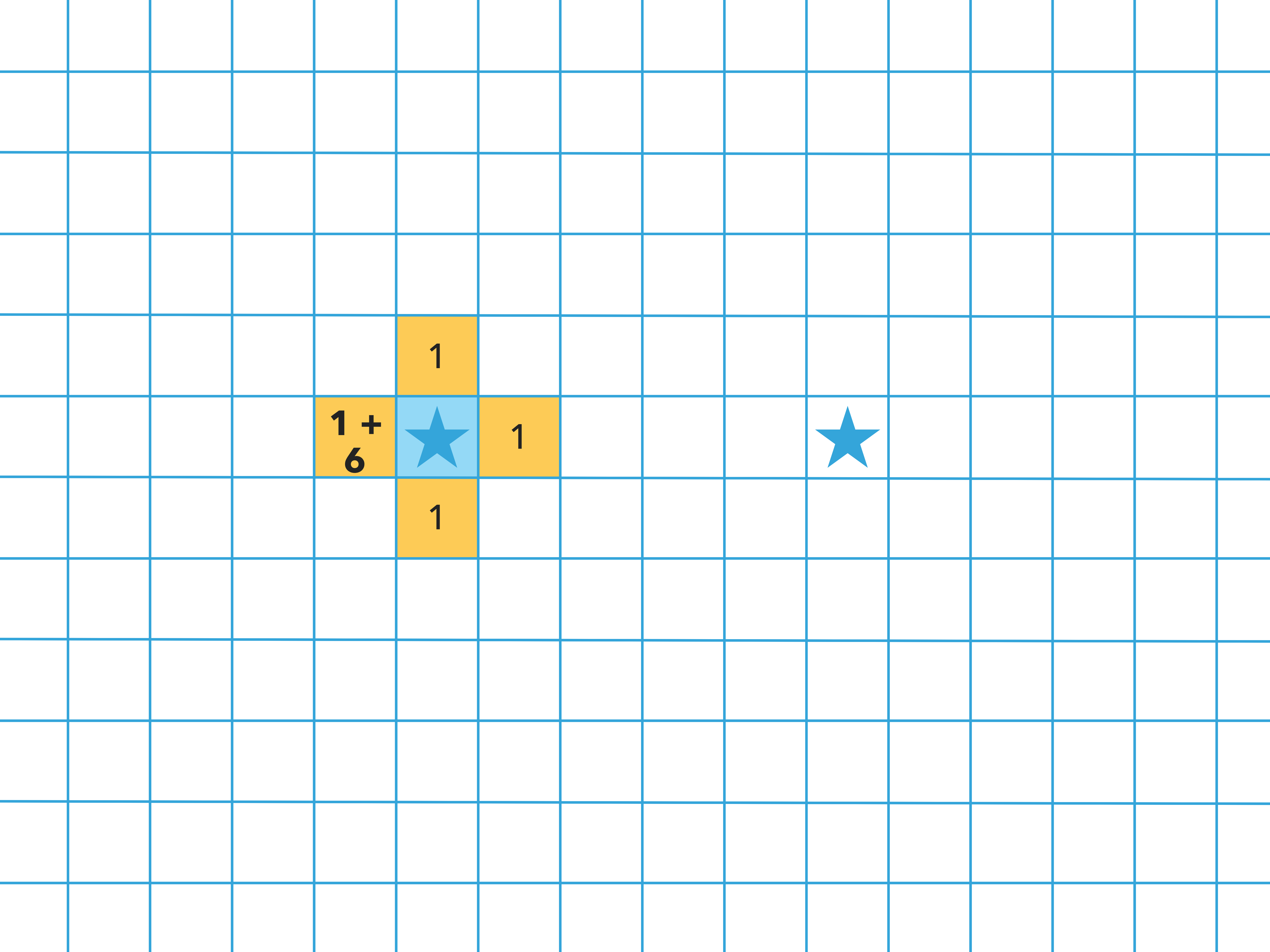


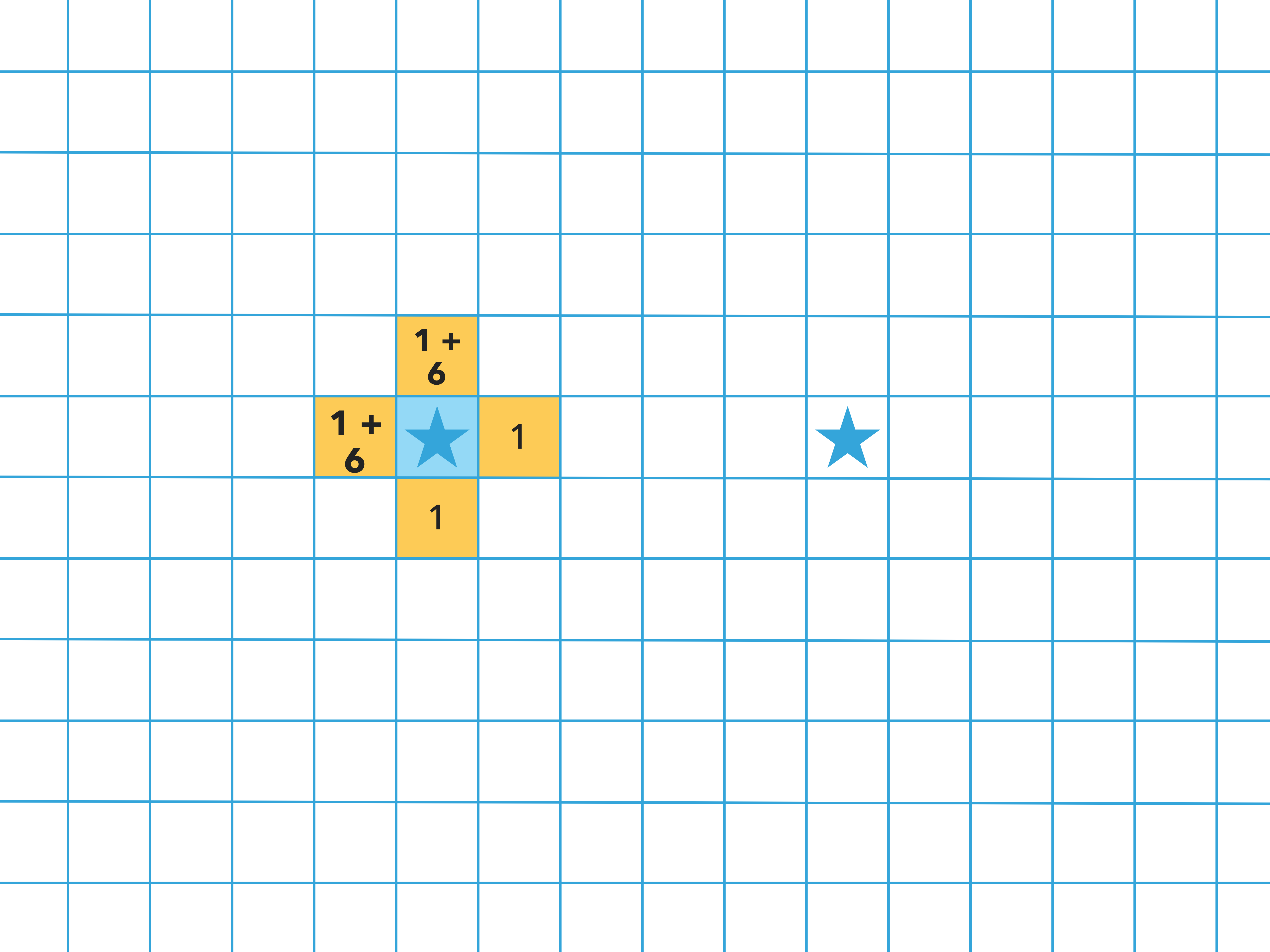

```
function futureCost(u, t)
  return abs(u.row - t.row) + abs(u.col - t.col)
```

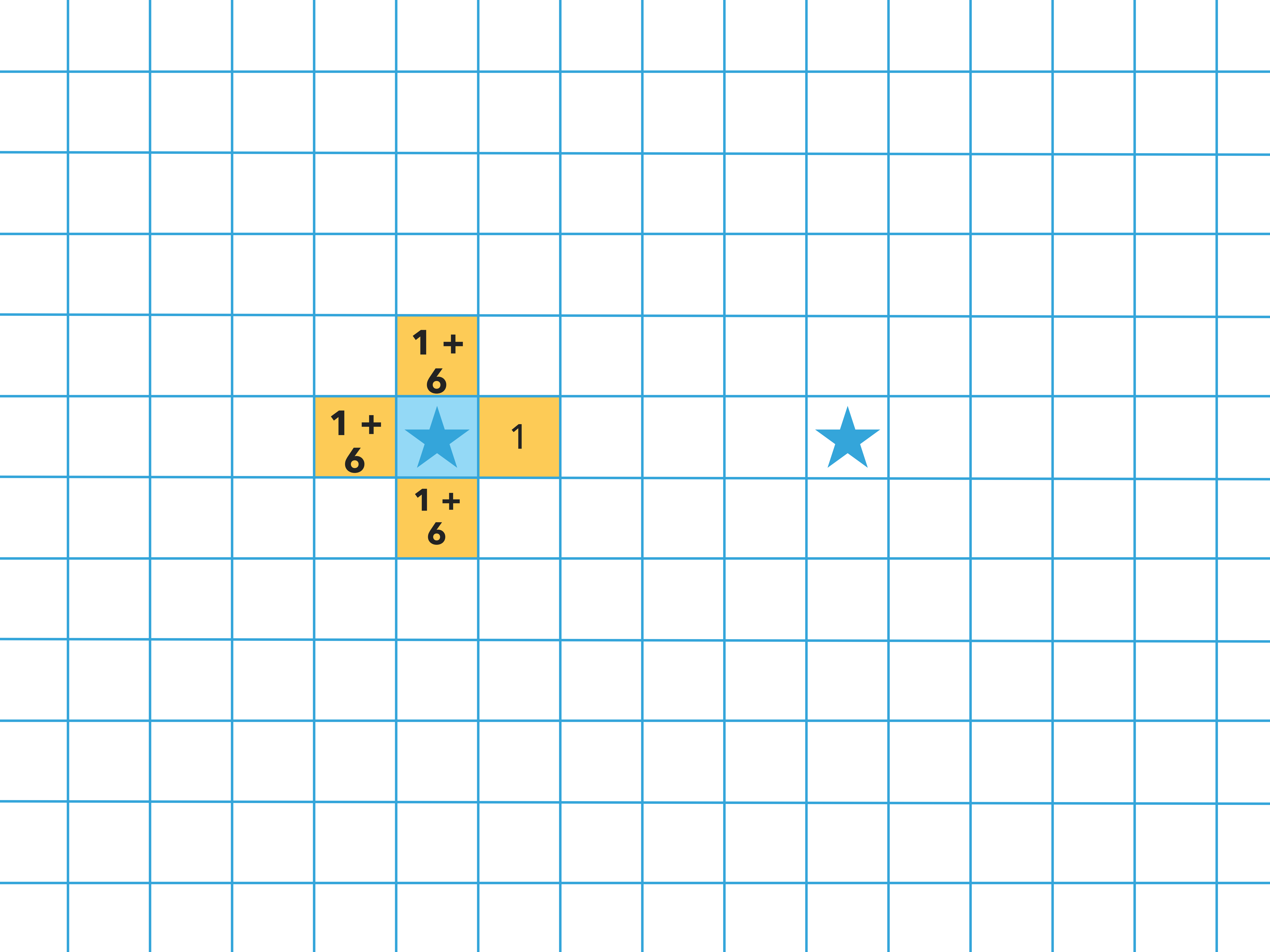


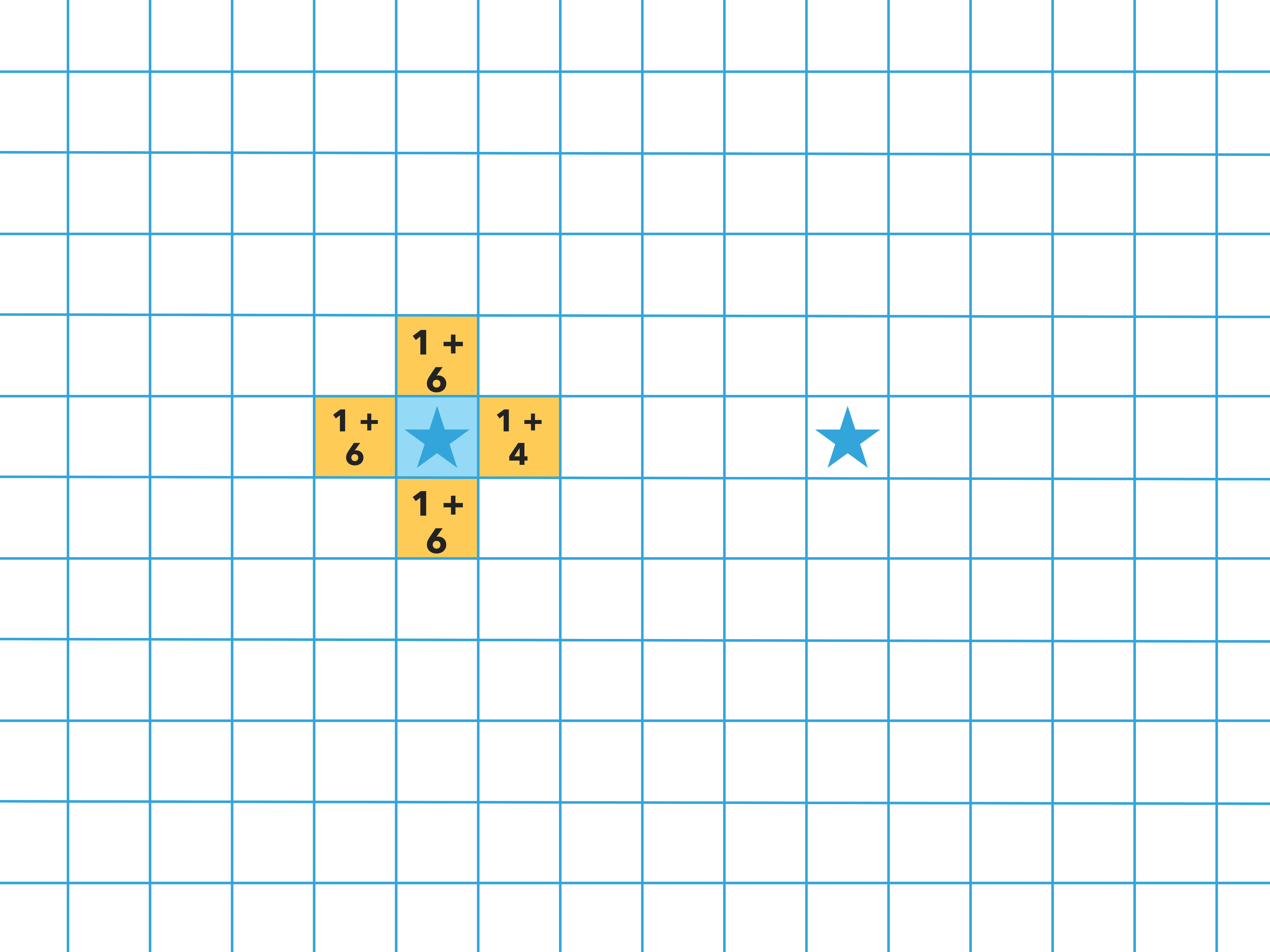



1
1
1
1

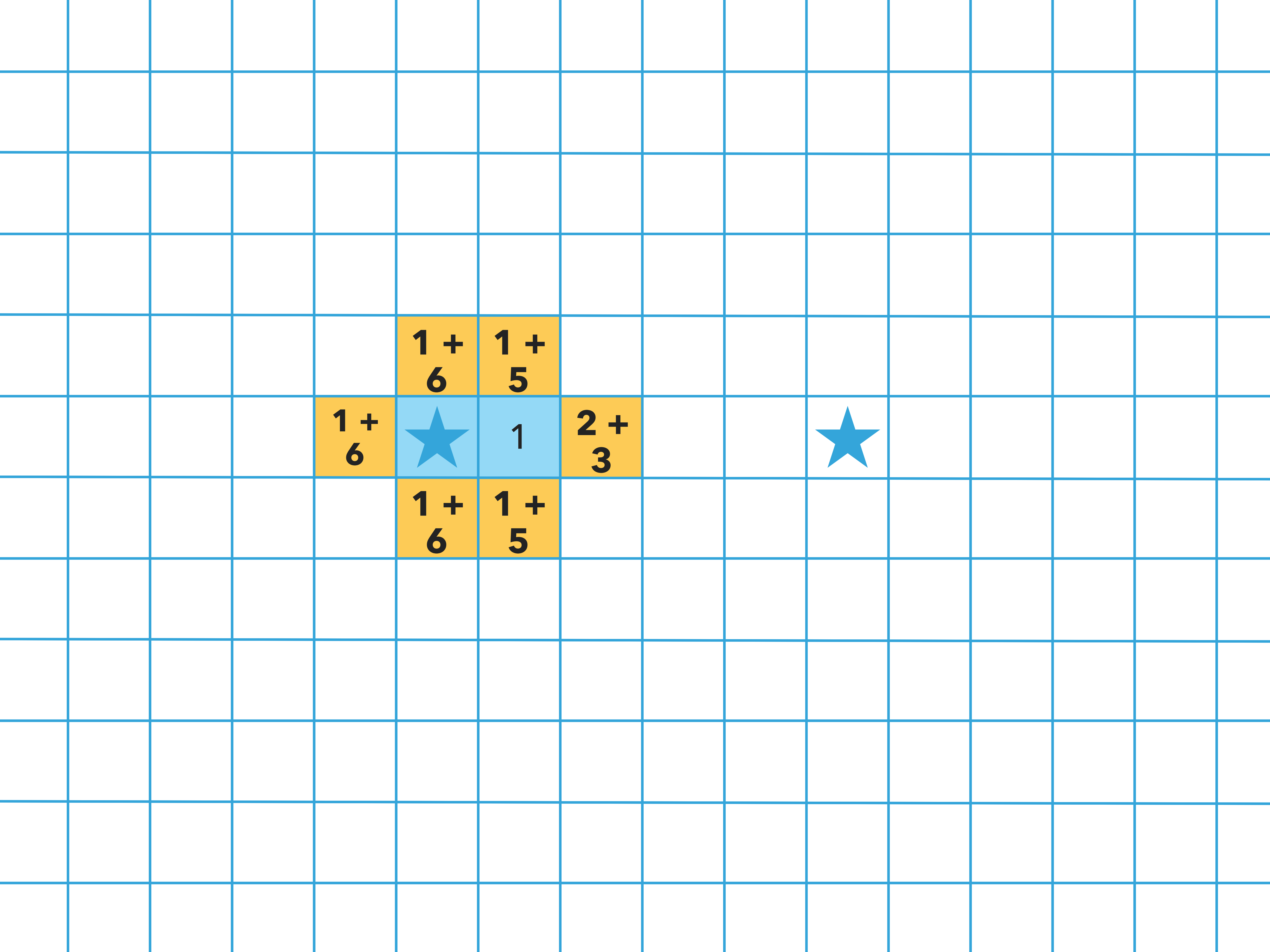

1
1 + 6
1
1


1 +
6
1 +
6
1
1

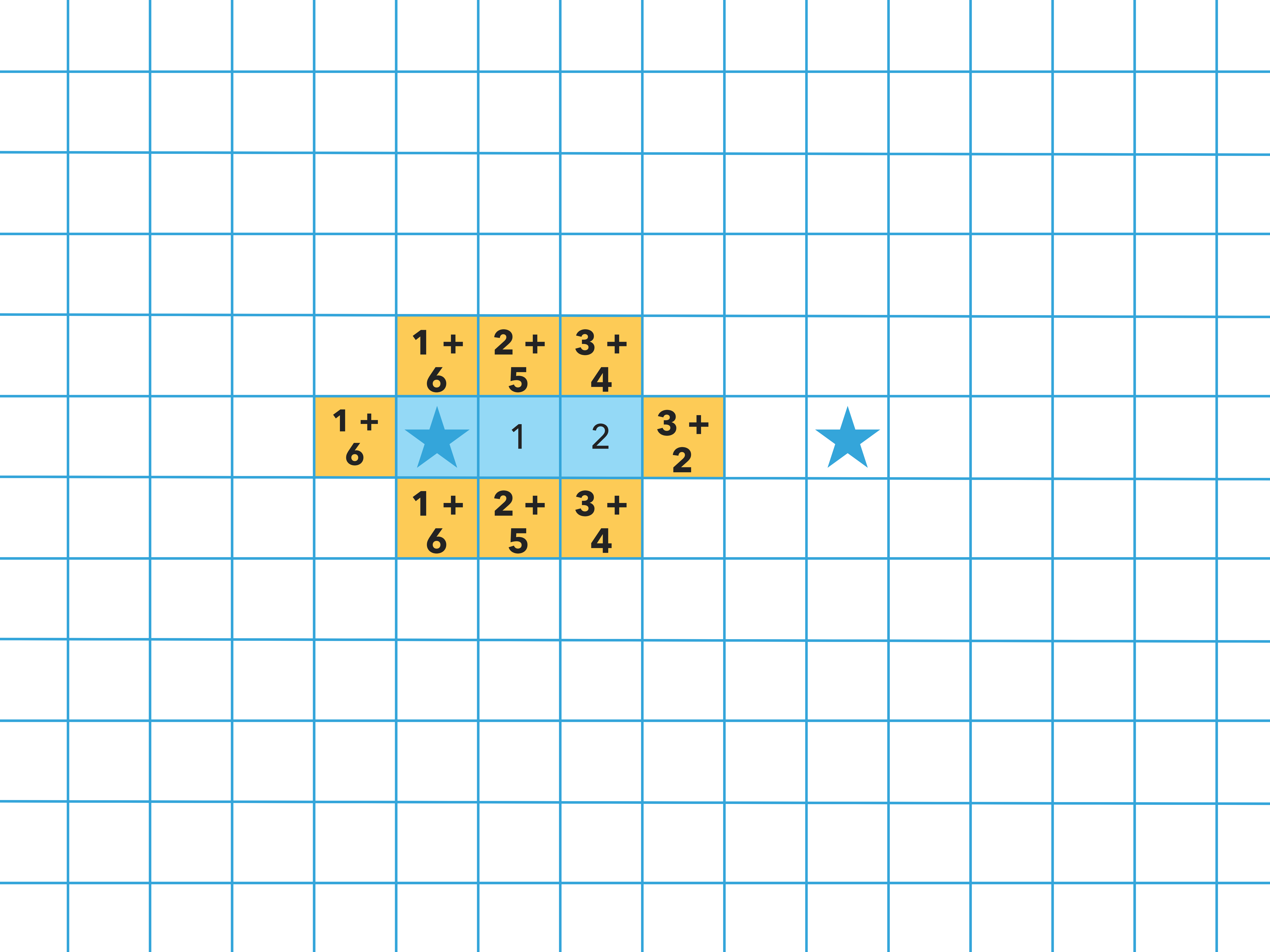

1 + 6
1 + 6
1
1 + 6

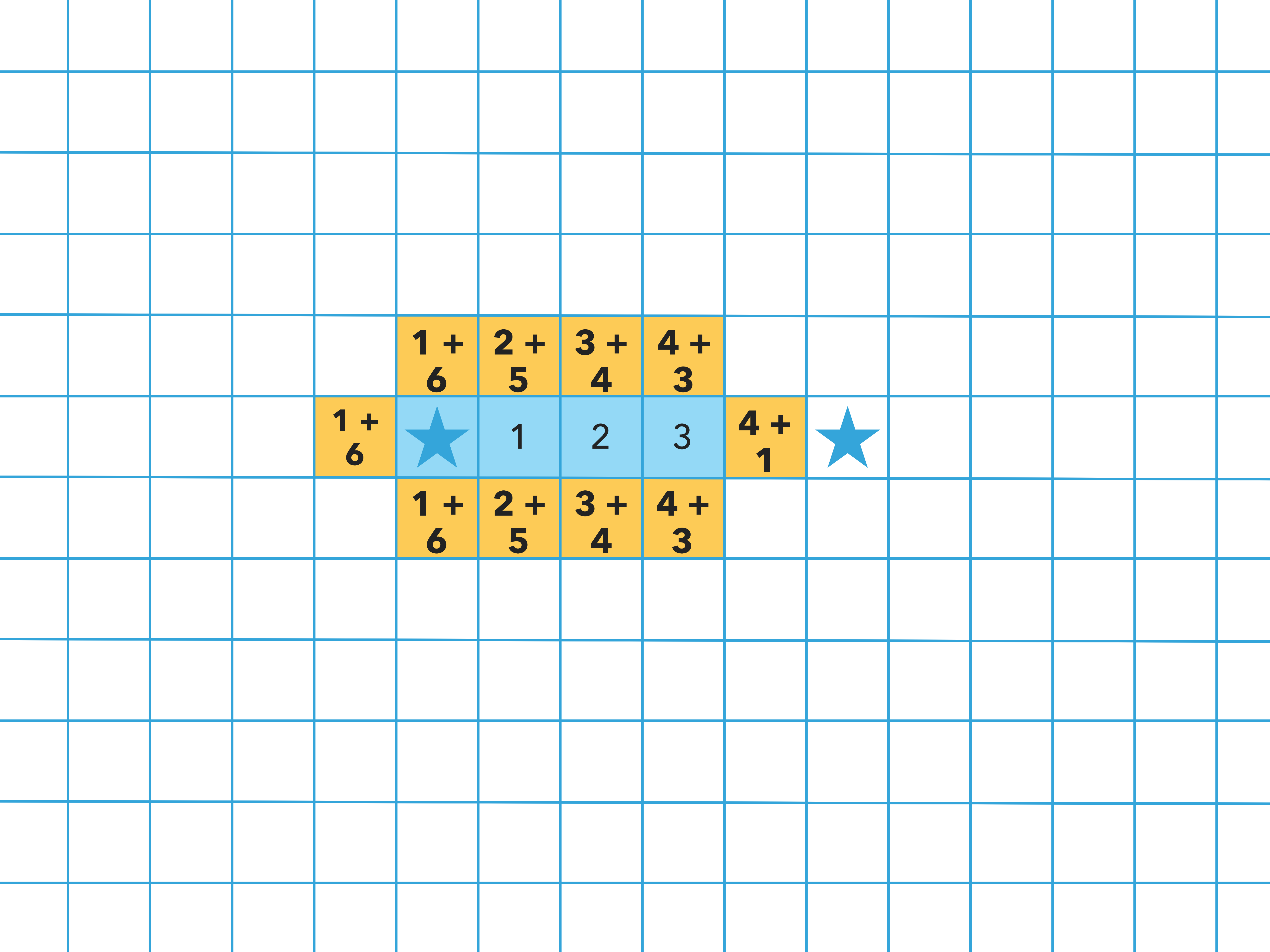


1 + 6
1 + 6
1 + 4
1 + 6

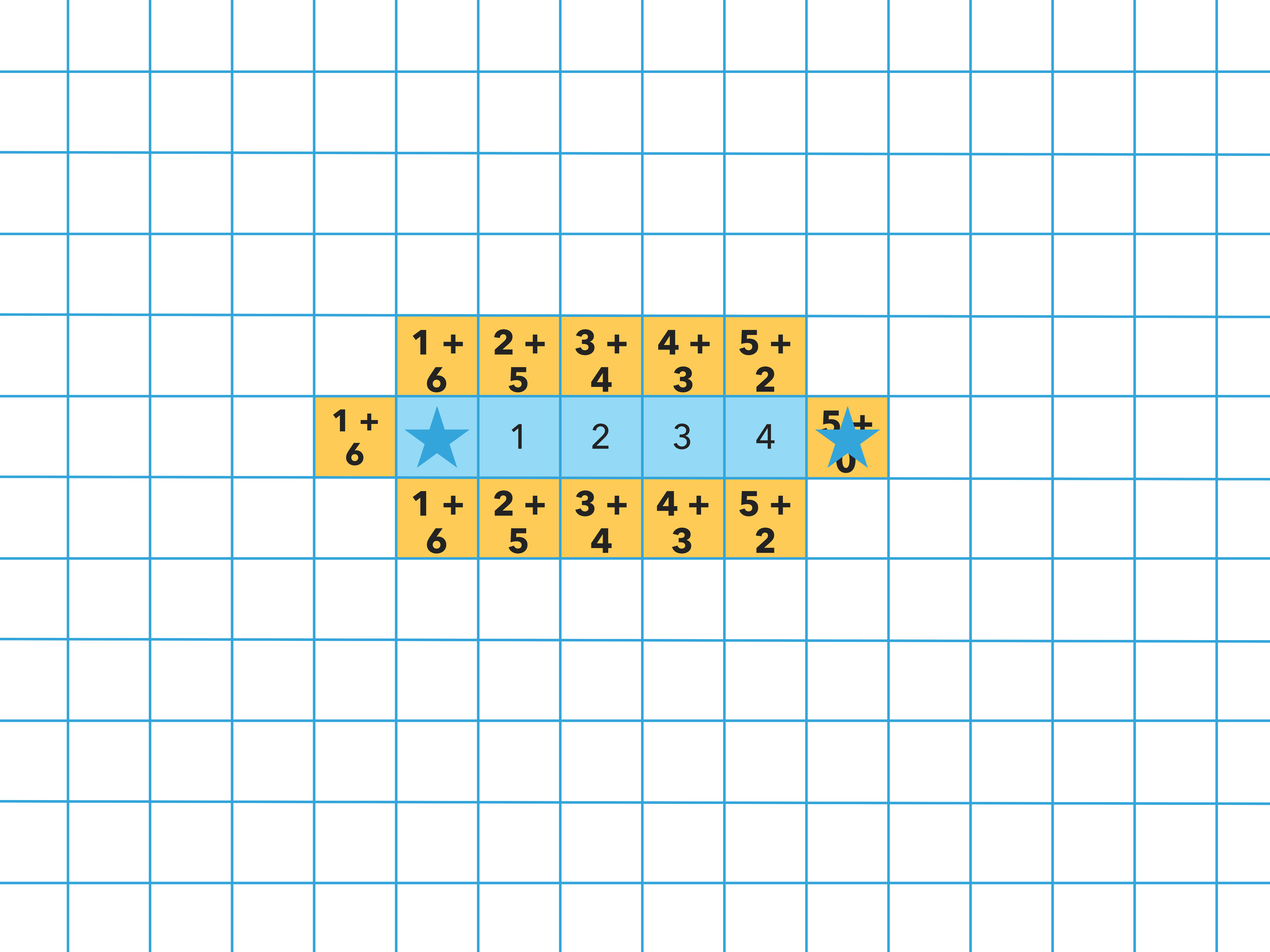


1 + 6
1 + 5
1 + 6
1
2 + 3
1 + 6
1 + 5

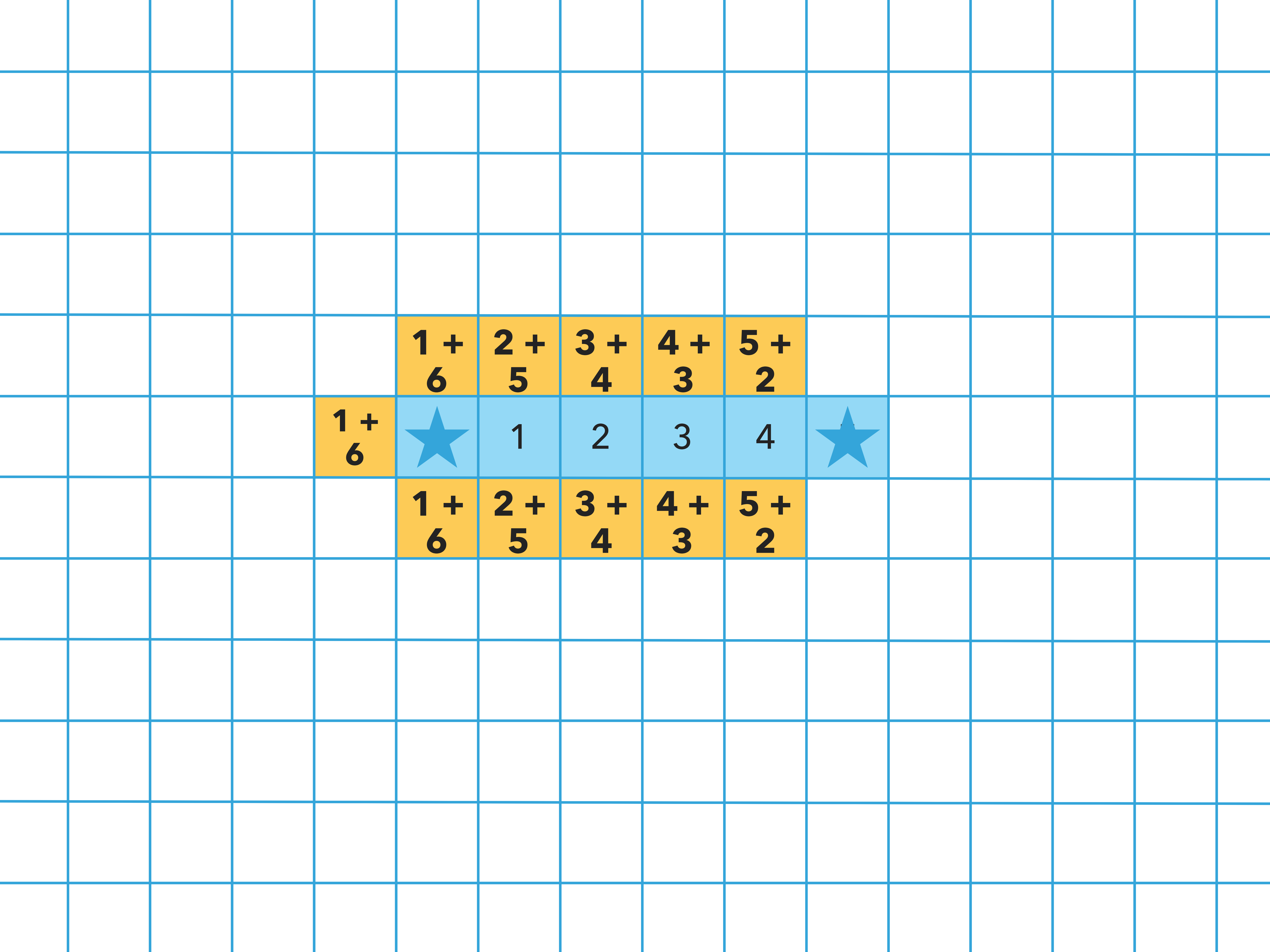

1 + 6
2 + 5
3 + 4
1 + 6
1
2
3 + 2
1 + 6
2 + 5
3 + 4



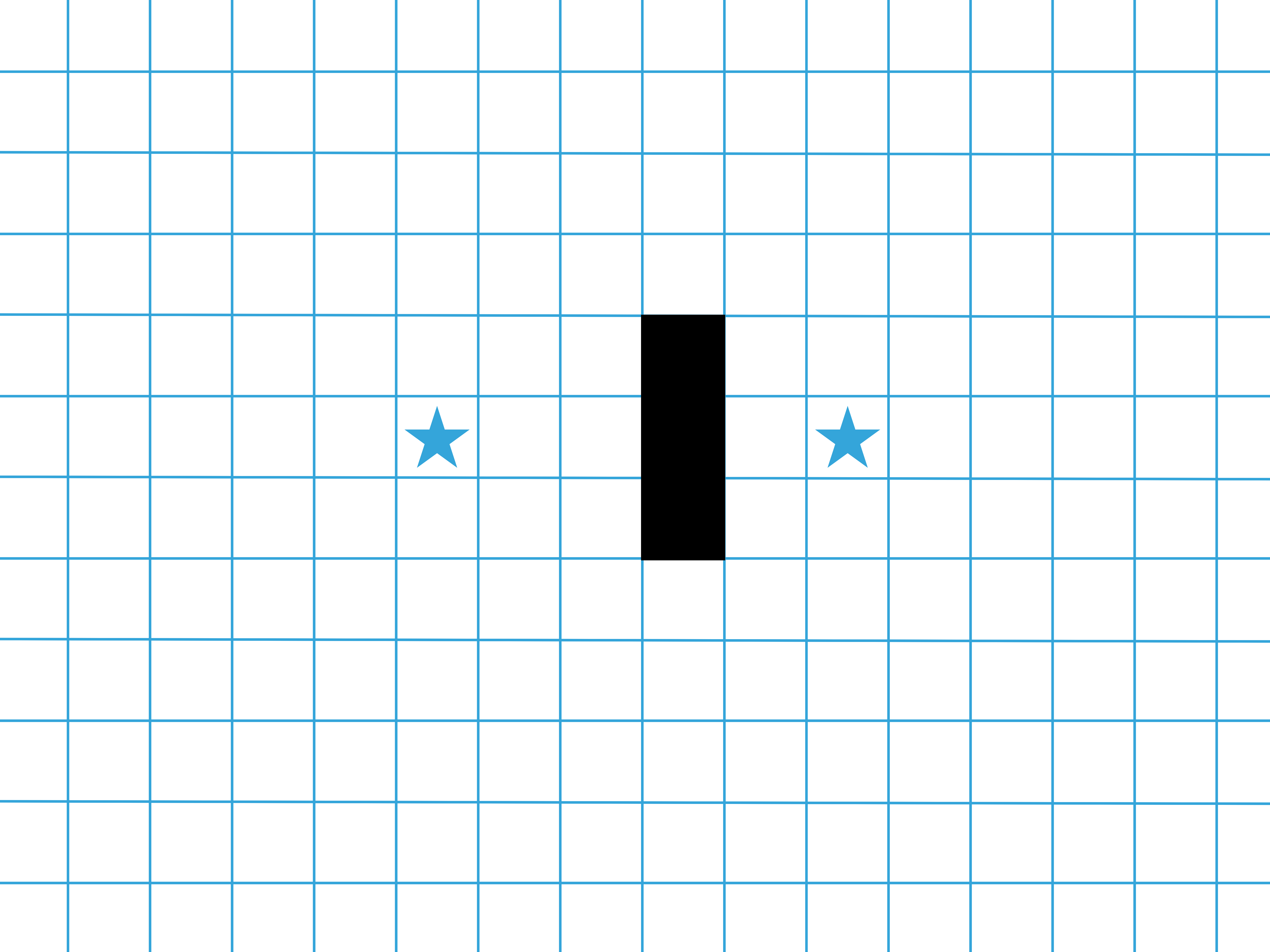
1 + 6
2 + 5
3 + 4
4 + 3
1 + 6
1
2
3
4 + 1
1 + 6
2 + 5
3 + 4
4 + 3



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

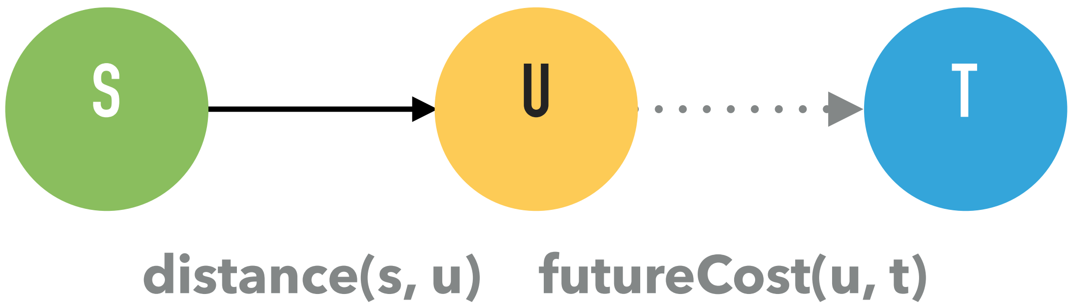


1 + 6
2 + 5
3 + 4
4 + 3
5 + 2
1 + 6
1
2
3
4
1 + 6
2 + 5
3 + 4
4 + 3
5 + 2



MAKING GOOD LIFE DECISIONS

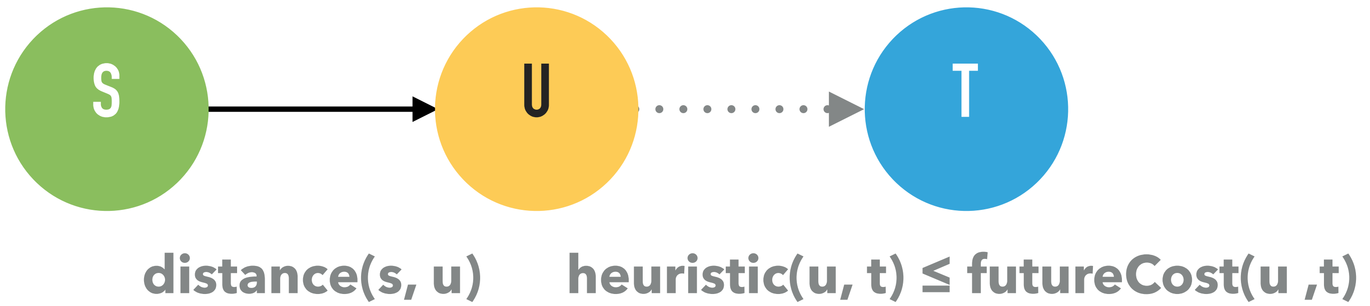
FORMAL DEFINITIONS



IDEAL

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

FORMAL DEFINITIONS



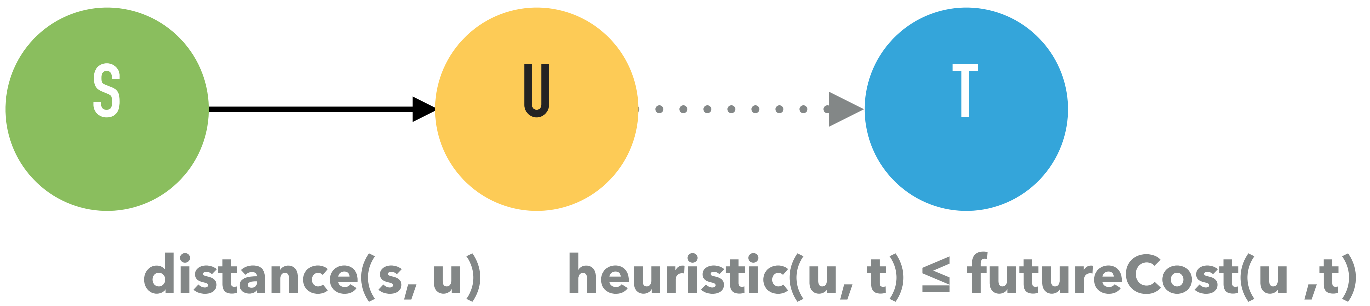
IDEAL

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

A*

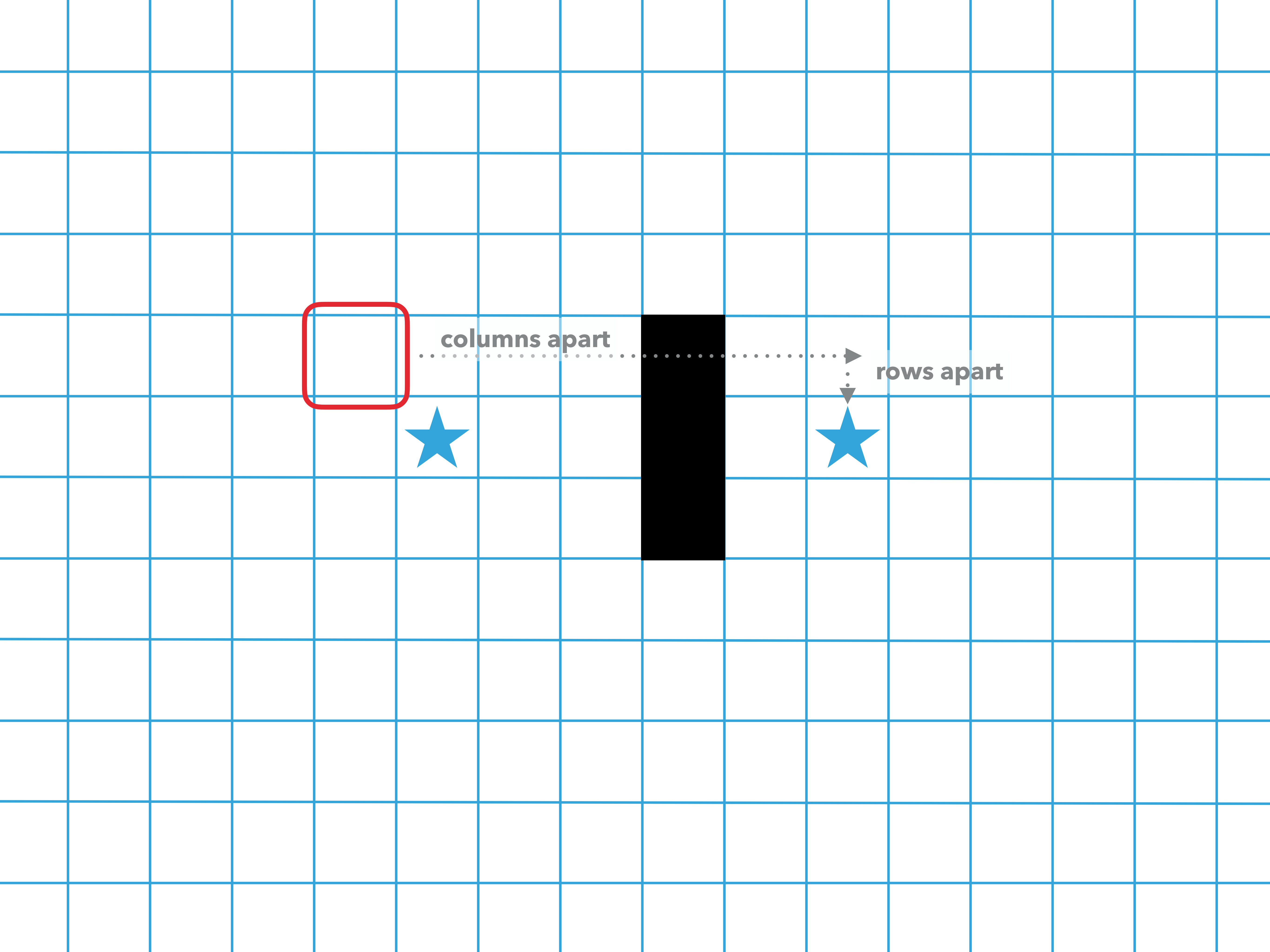
priority(u) = distance(s, u)
+ ***heuristic(u, t)***

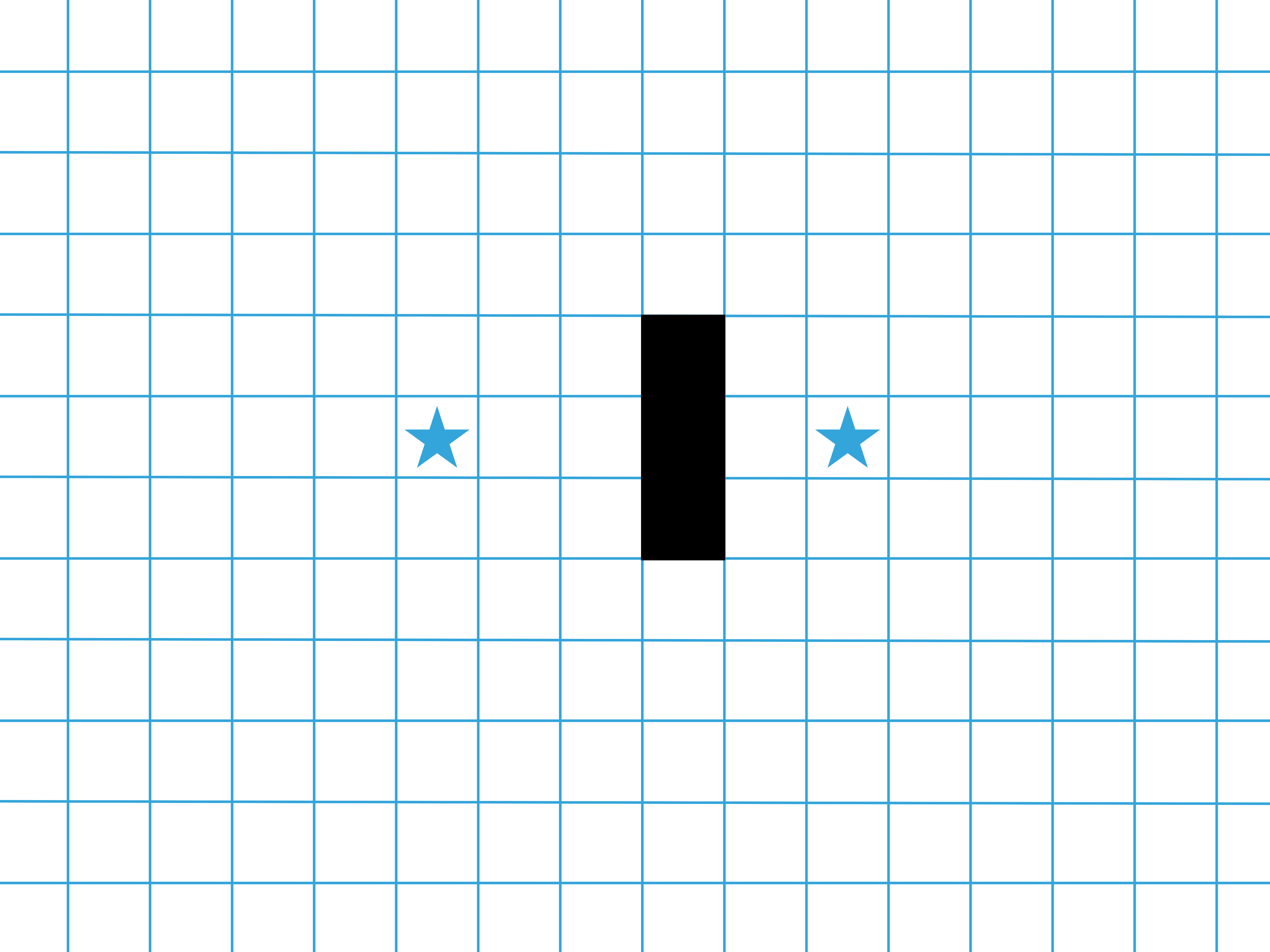
HEURISTICS

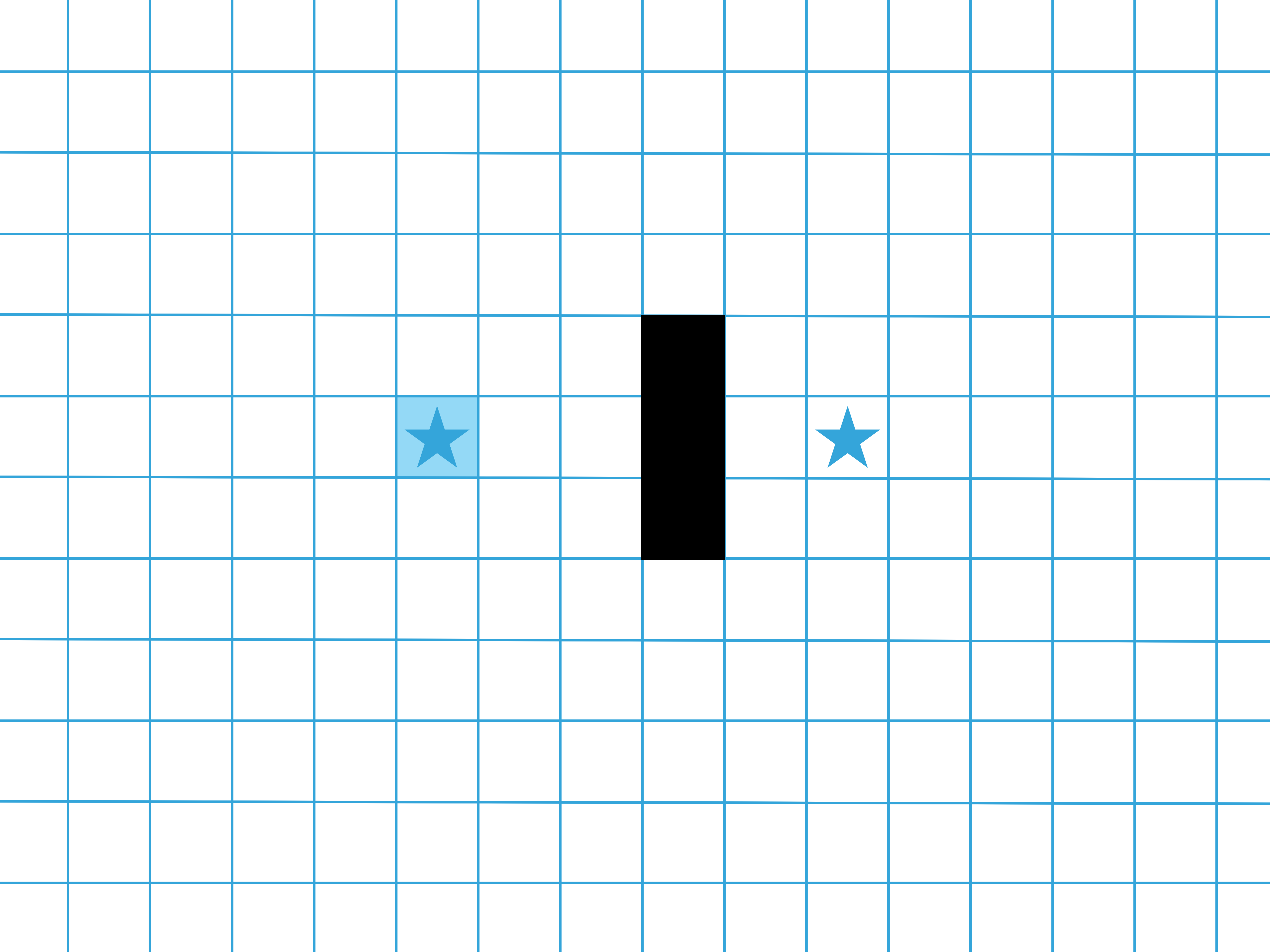


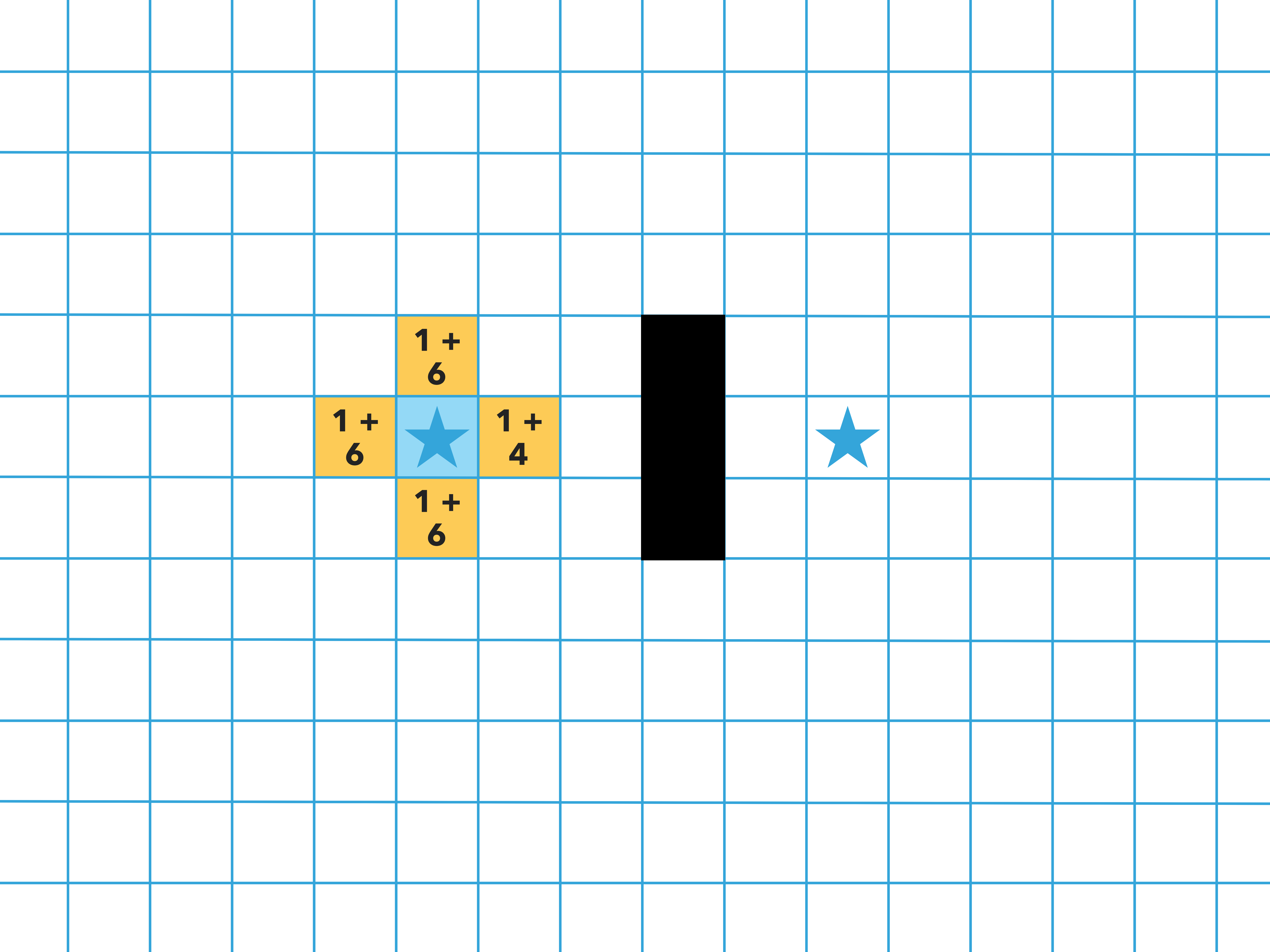
A heuristic is a function that **underestimates** the cost of of traveling from u to t.

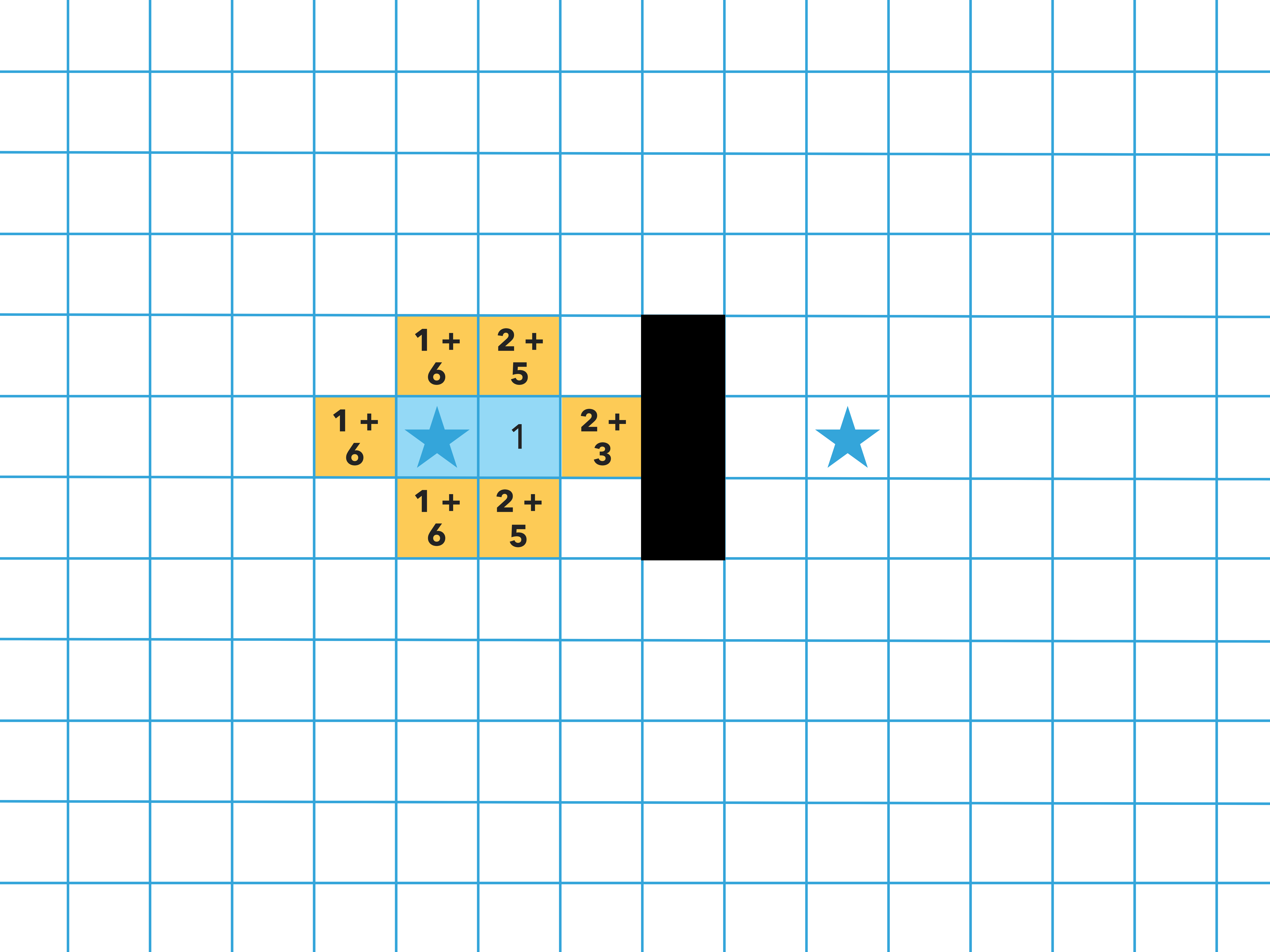
It's a "relaxation" heuristic.


columns apart
rows apart

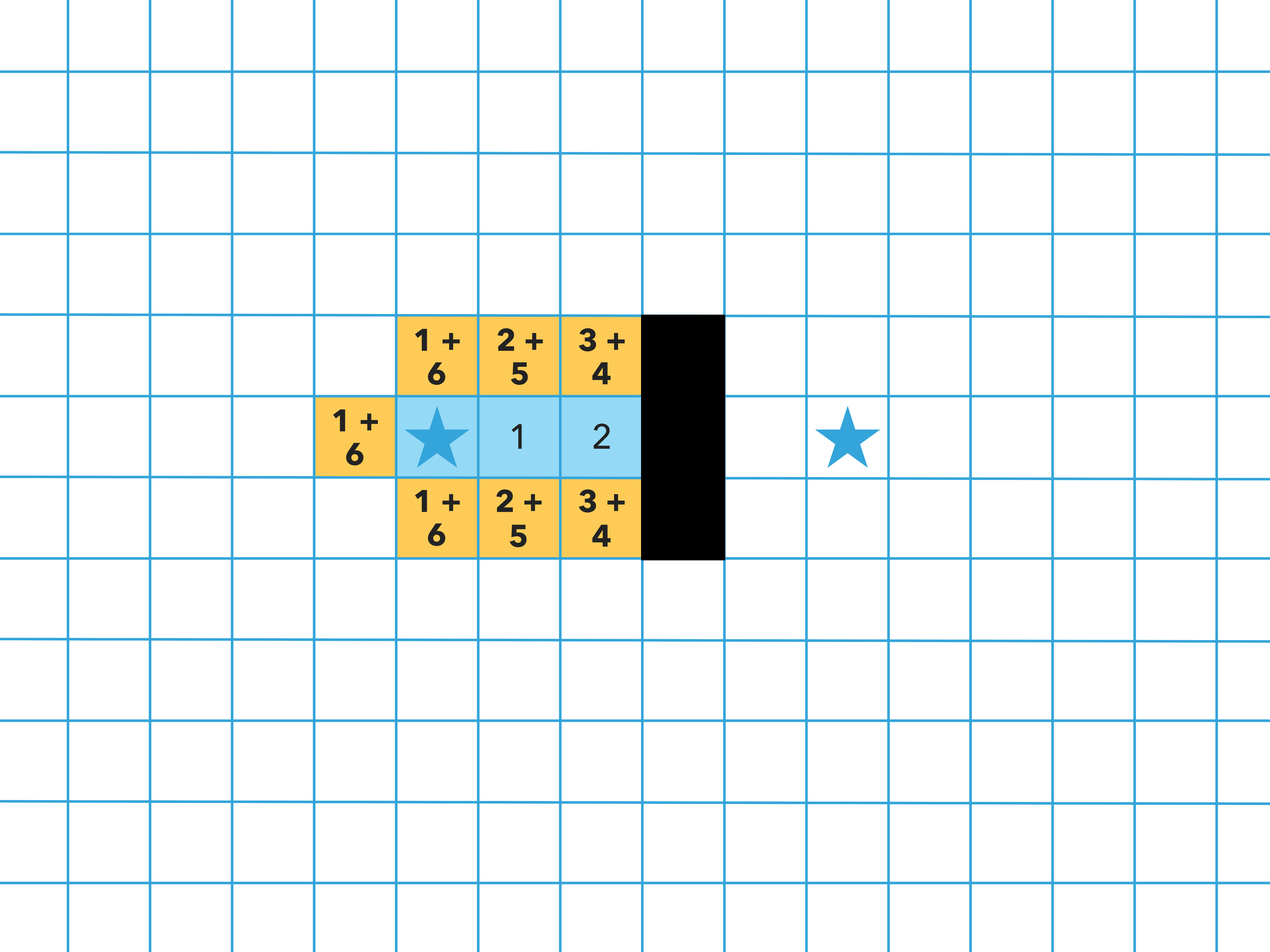


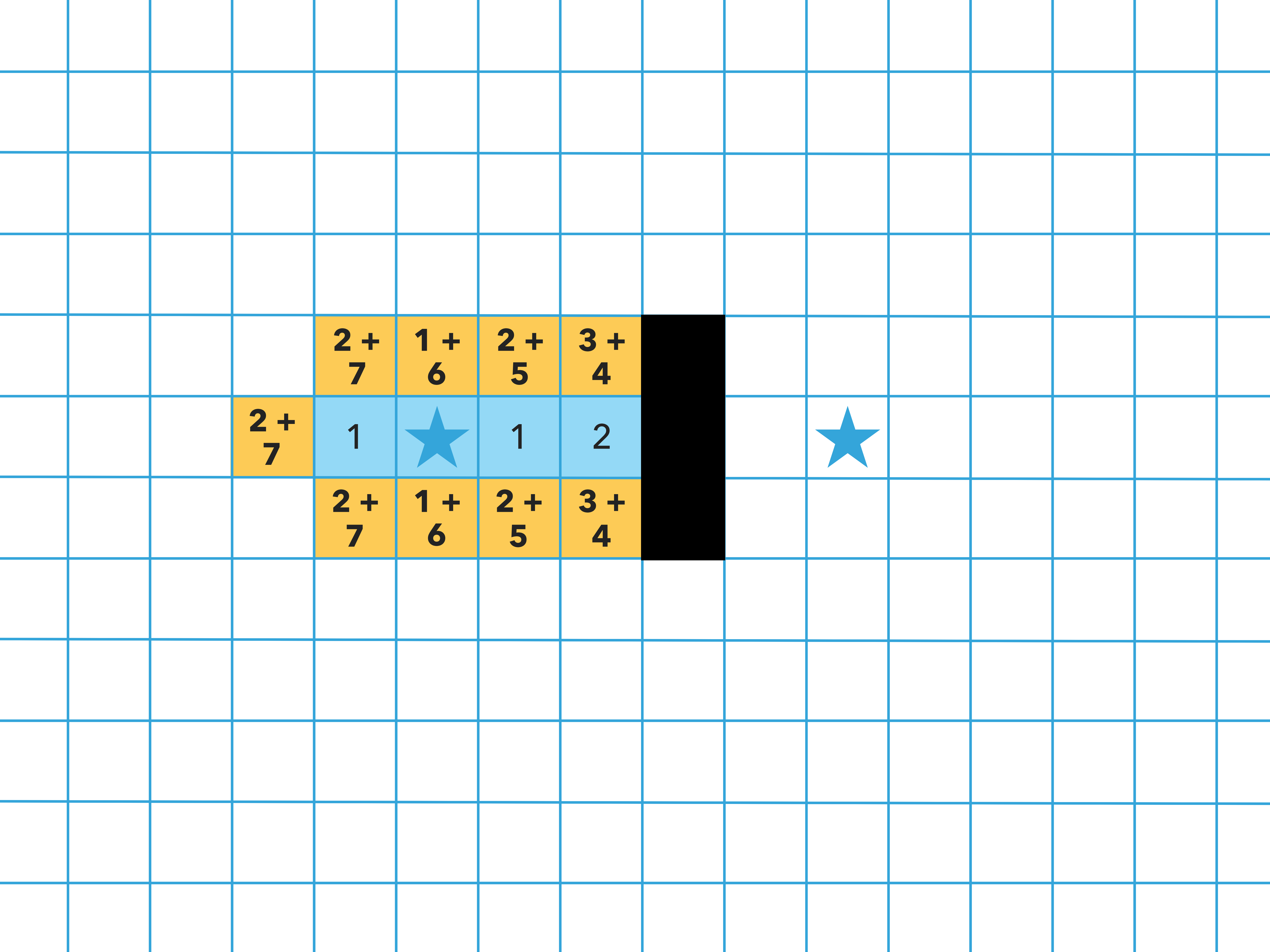


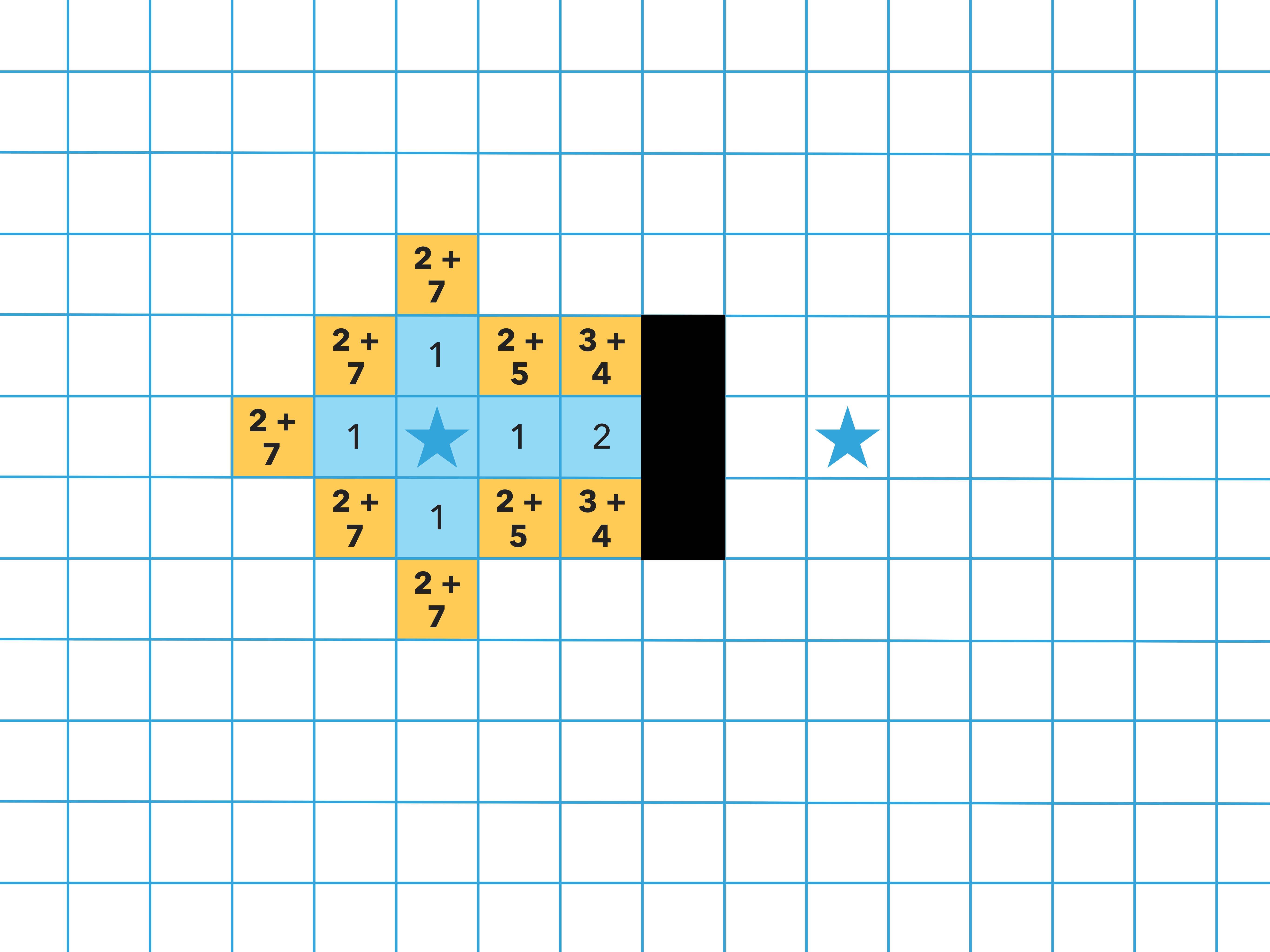

1 + 6
1 + 6
1 + 4
1 + 6

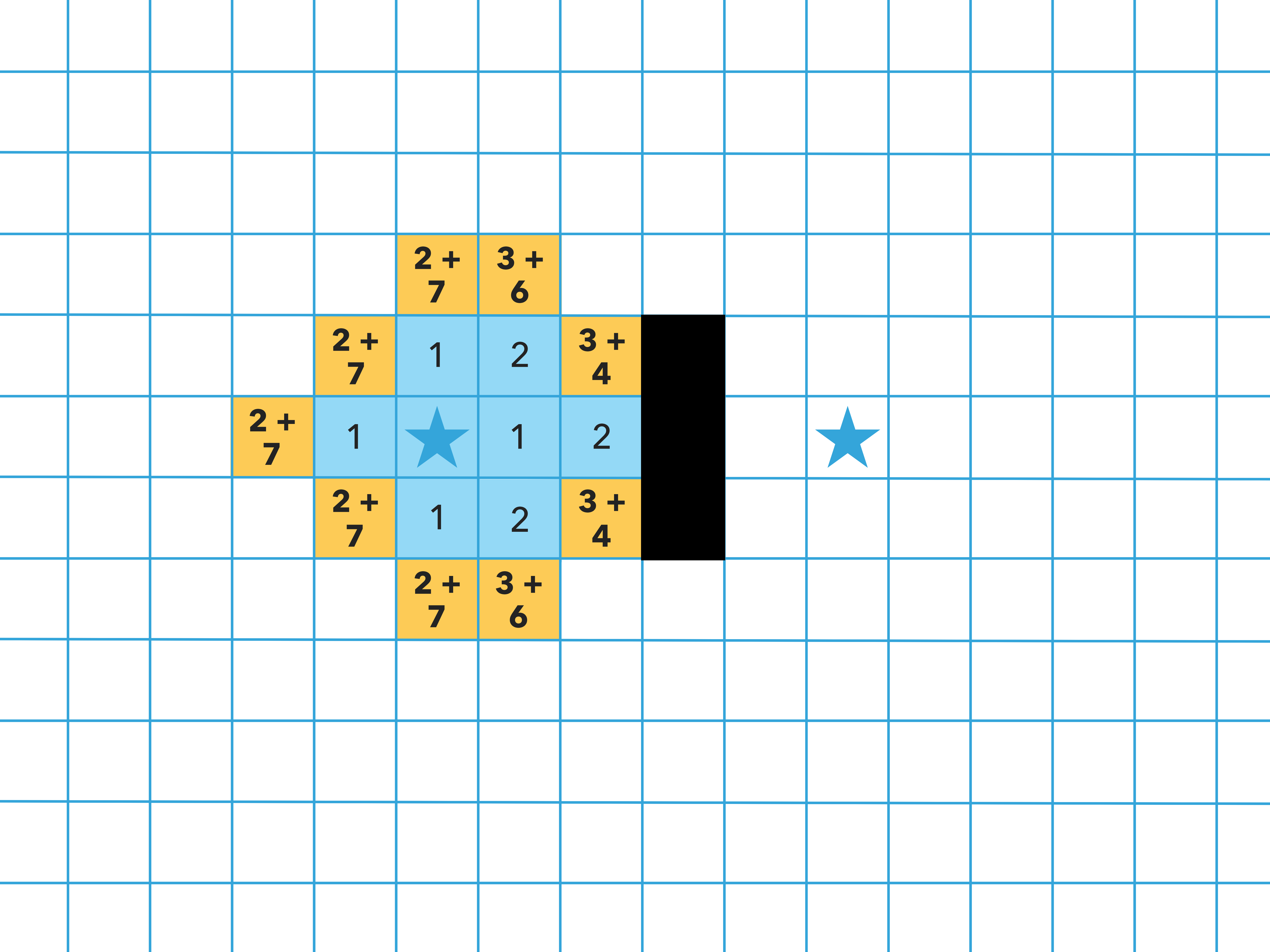


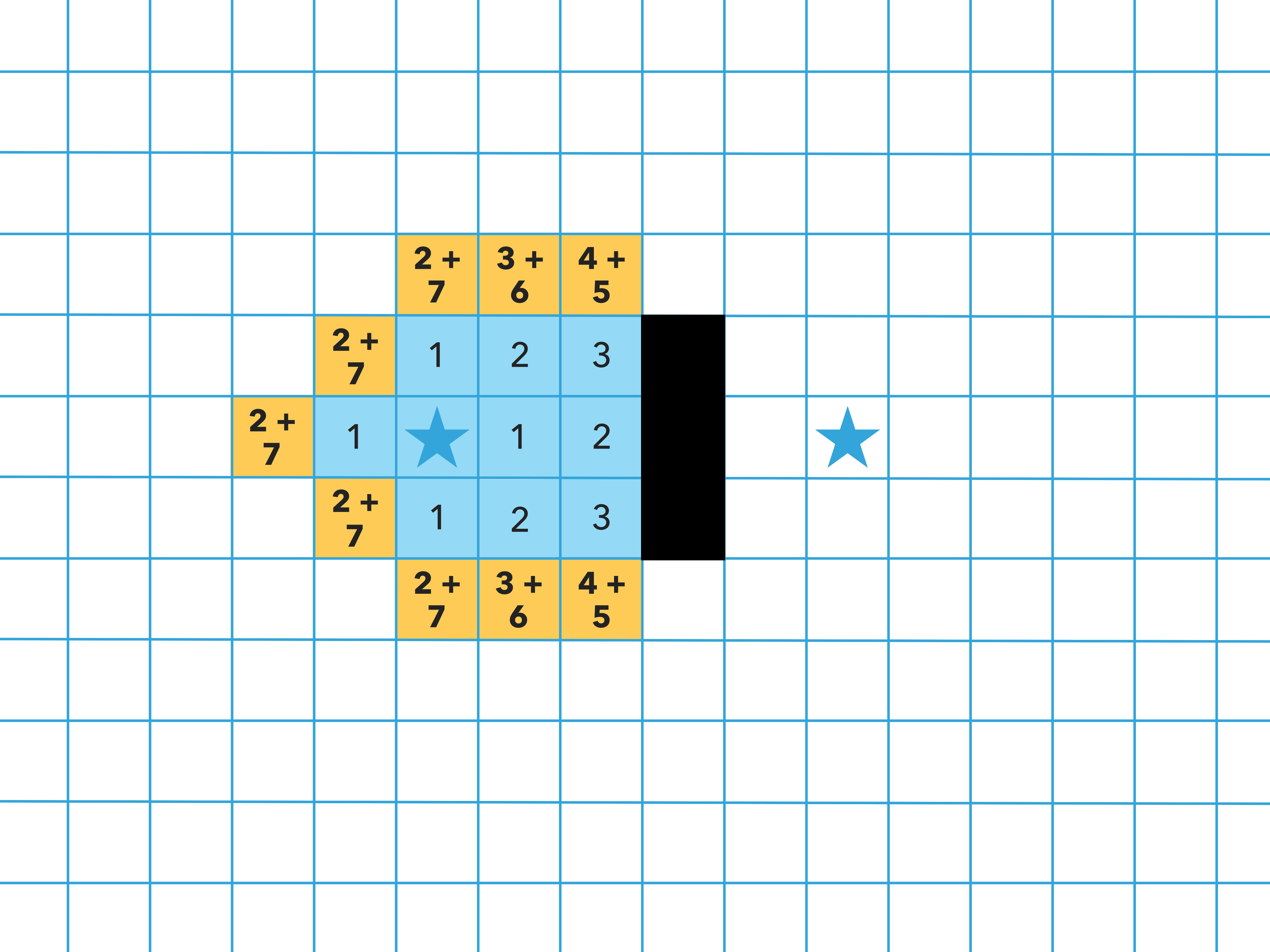
1 + 6
2 + 5
1 + 6
1
2 + 3
1 + 6
2 + 5


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

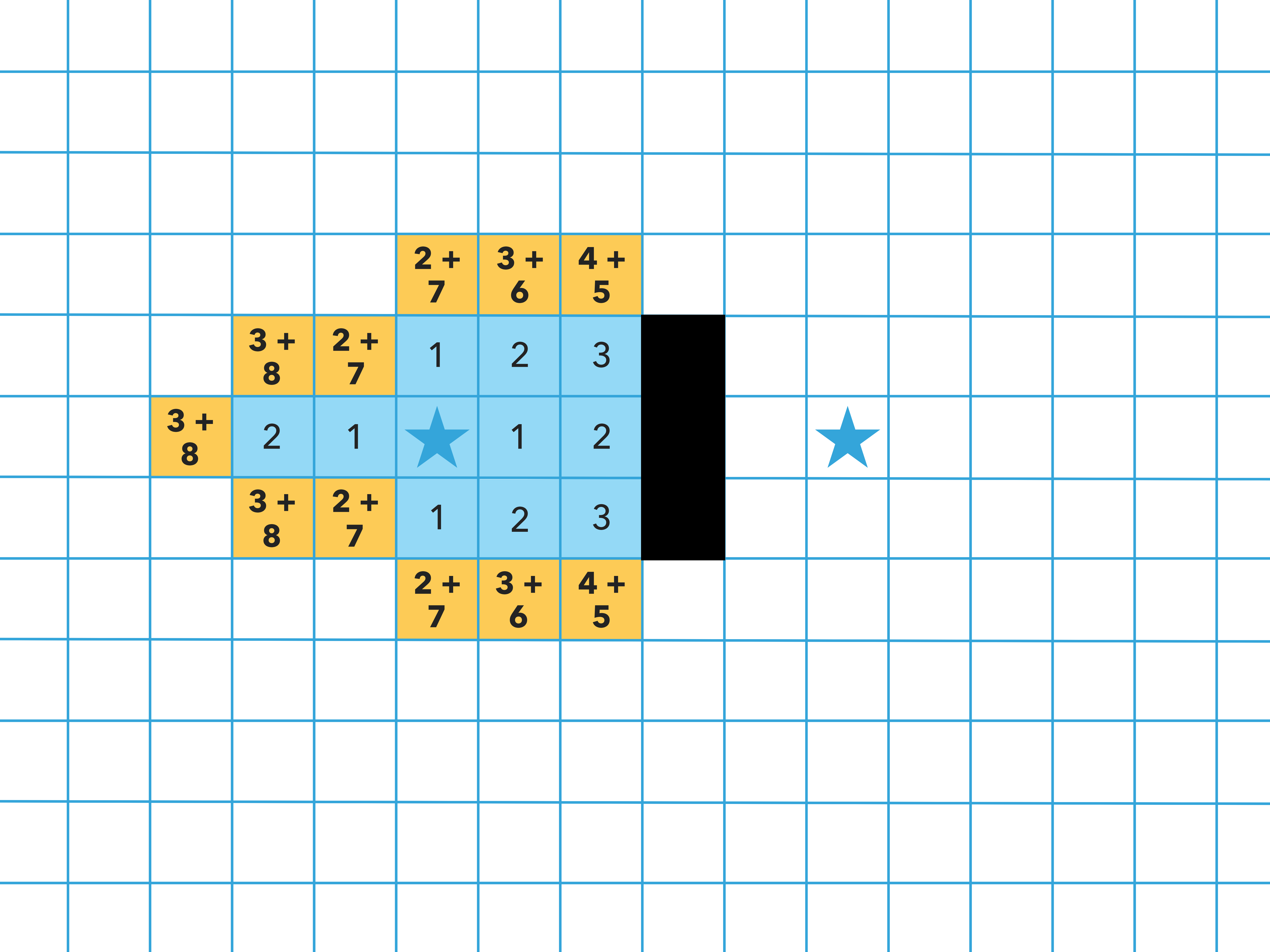

2 + 7
1 + 6
2 + 5
3 + 4
2 + 7
1
1
2
2 + 7
1 + 6
2 + 5
3 + 4


2 + 7
2 + 7
1
2 + 5
3 + 4
2 + 7
1
1
2
2 + 7
1
2 + 5
3 + 4
2 + 7

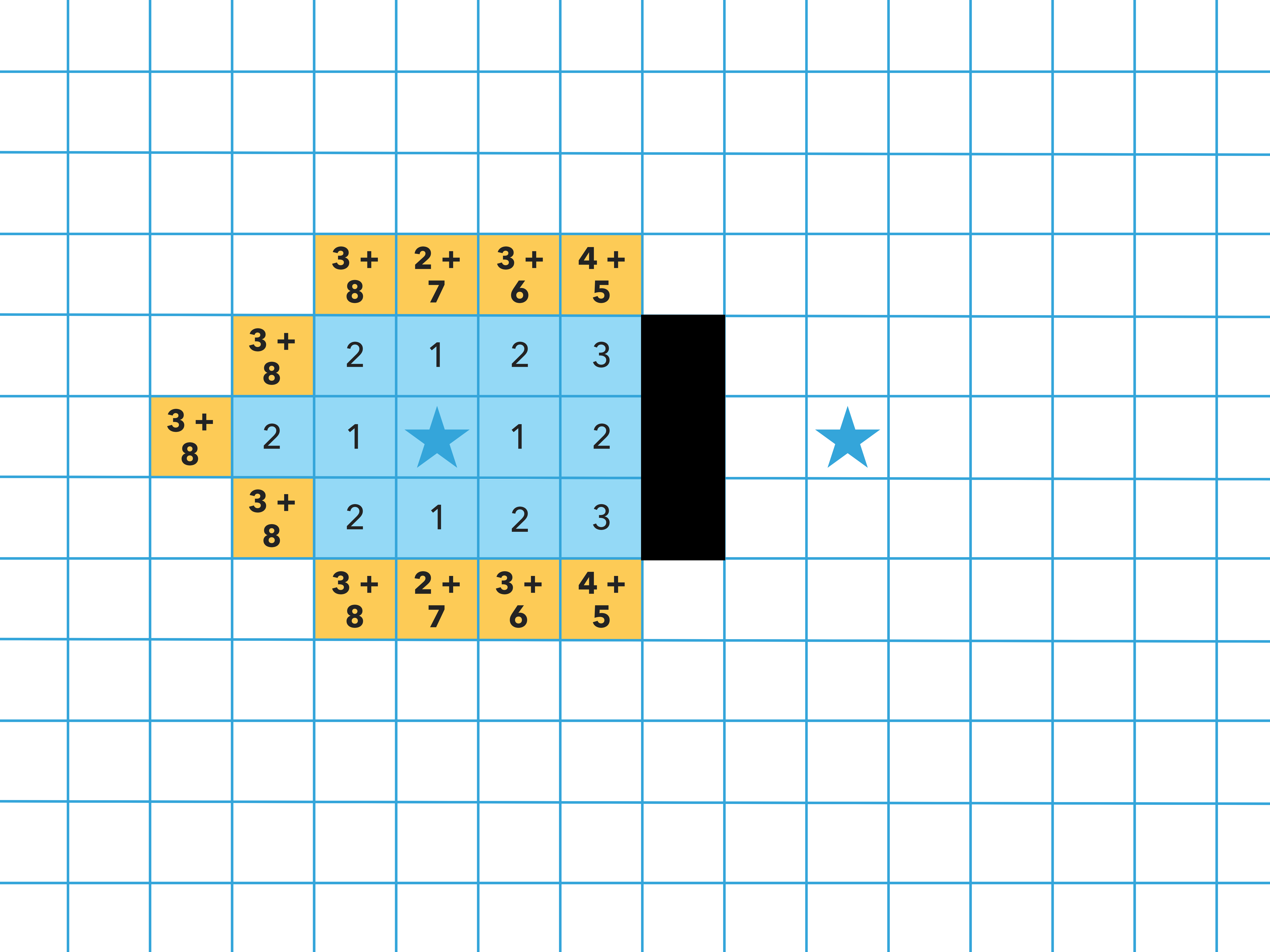

2 + 7
3 + 6
2 + 7
1
2
3 + 4
2 + 7
1
1
2
2 + 7
1
2
3 + 4
2 + 7
3 + 6

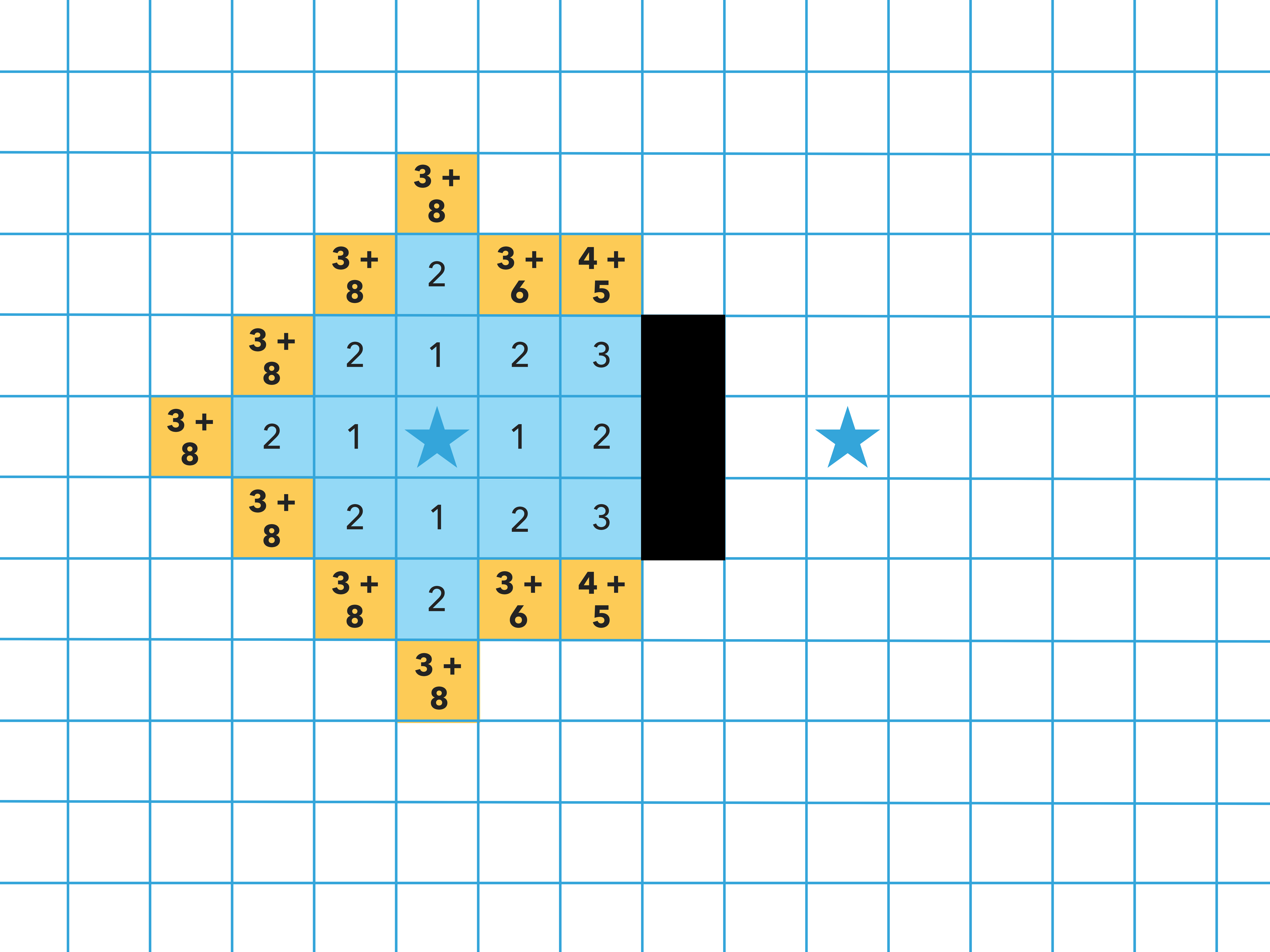


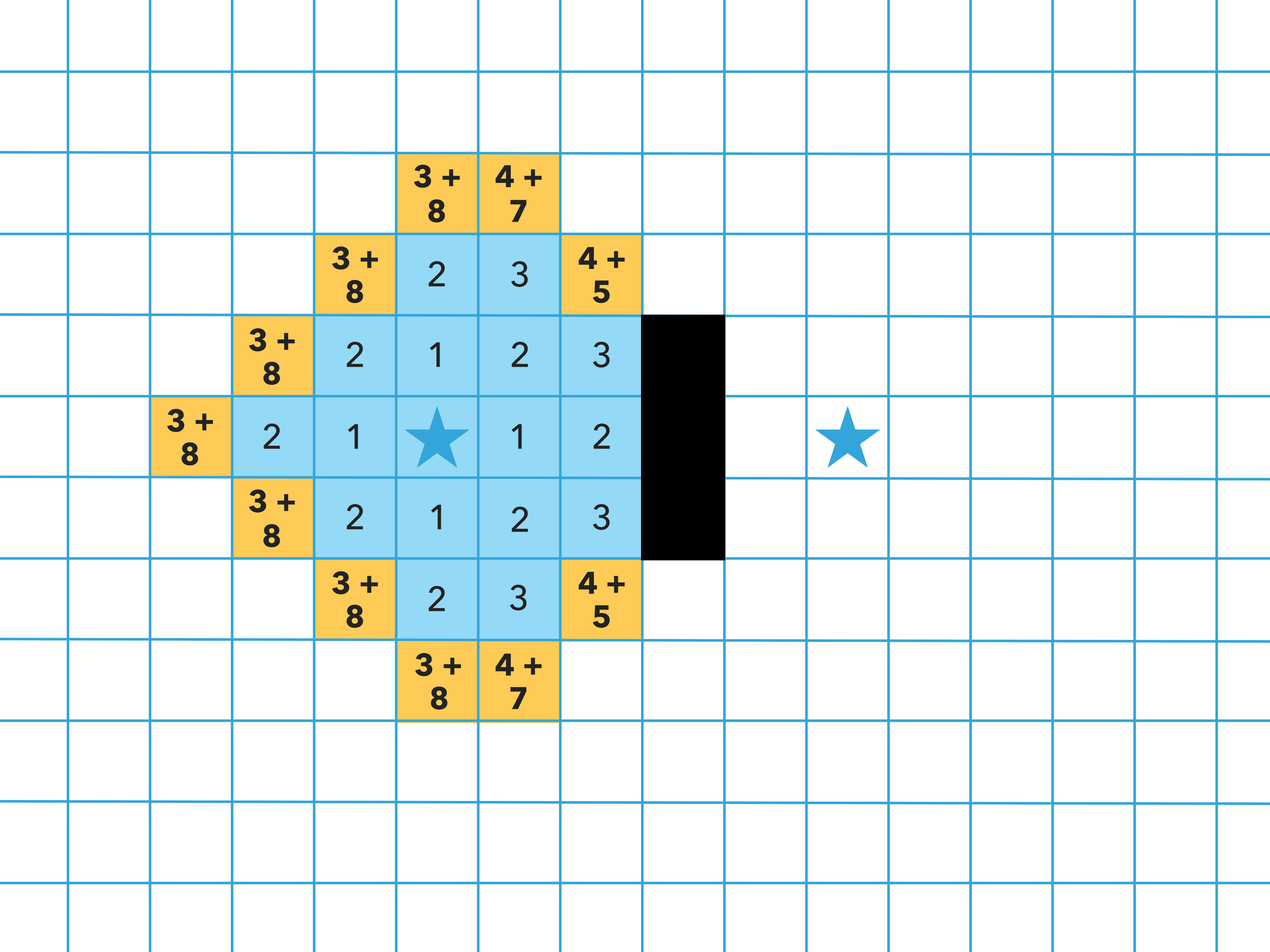
2 + 7
3 + 6
4 + 5
2 + 7
1
2
3
2 + 7
1
1
2
2 + 7
1
2
3
2 + 7
3 + 6
4 + 5

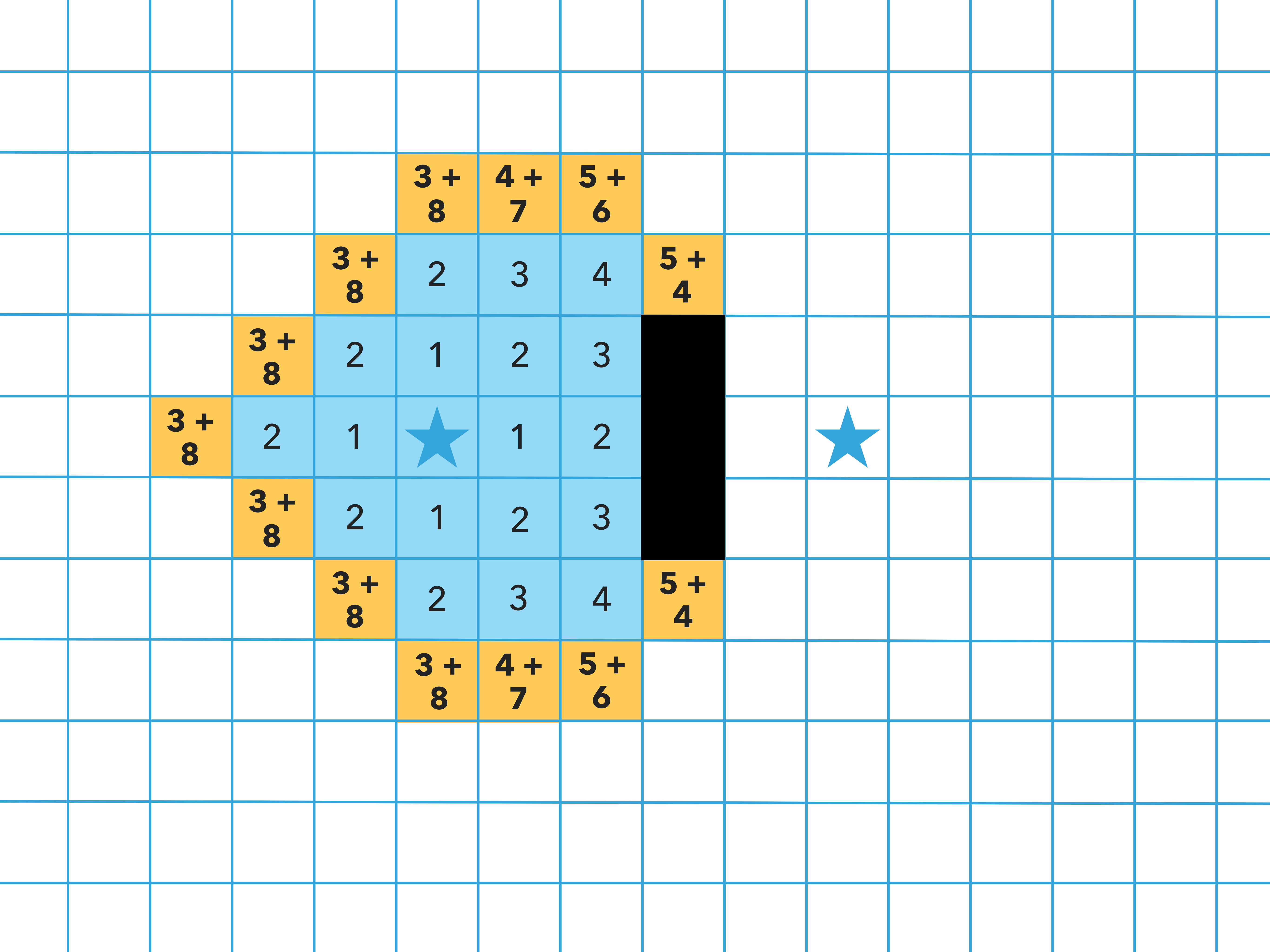


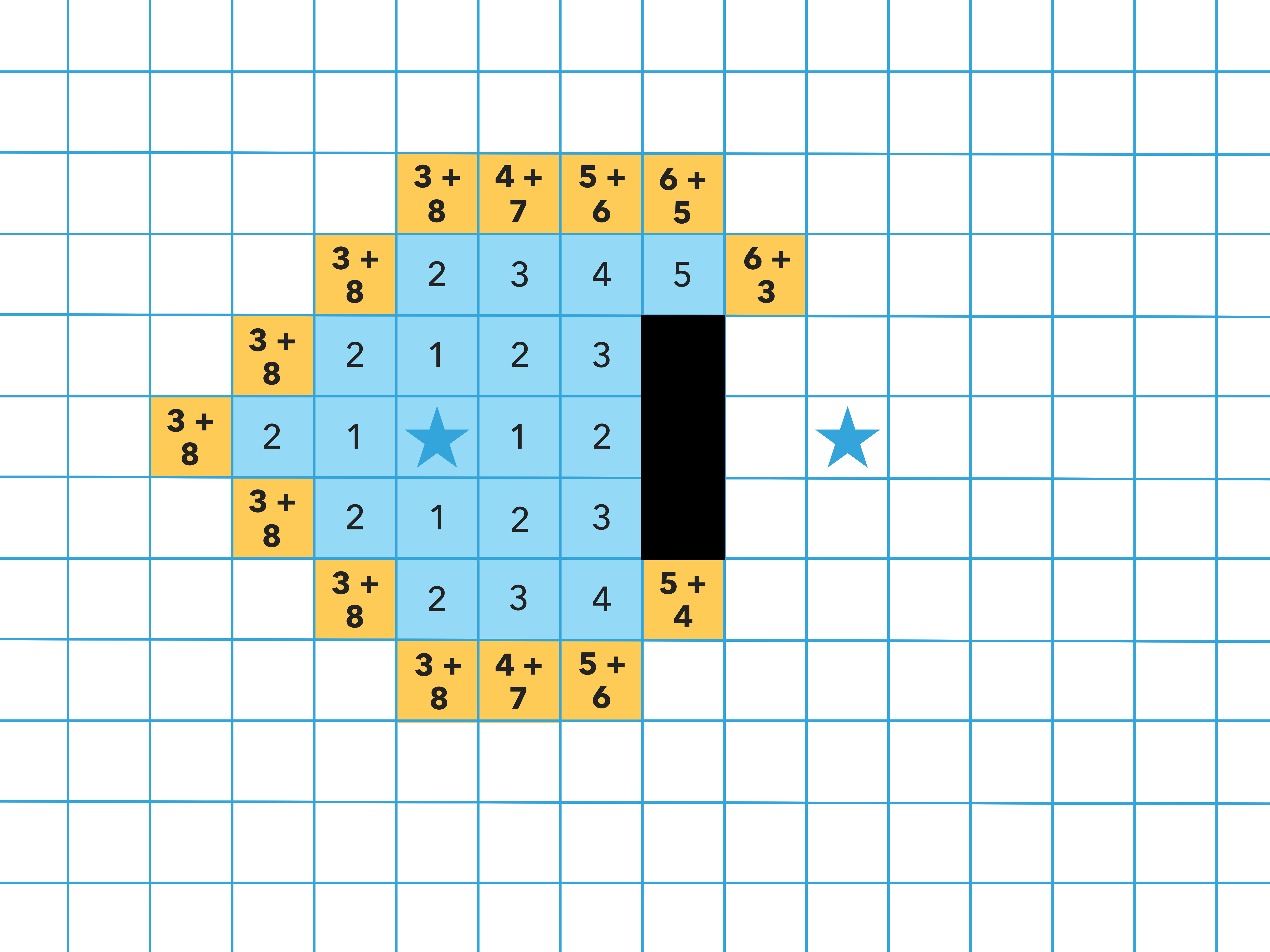
2 + 7
3 + 6
4 + 5
3 + 8
2 + 7
1
2
3
3 + 8
2
1
1
2
3 + 8
2 + 7
1
2
3
2 + 7
3 + 6
4 + 5

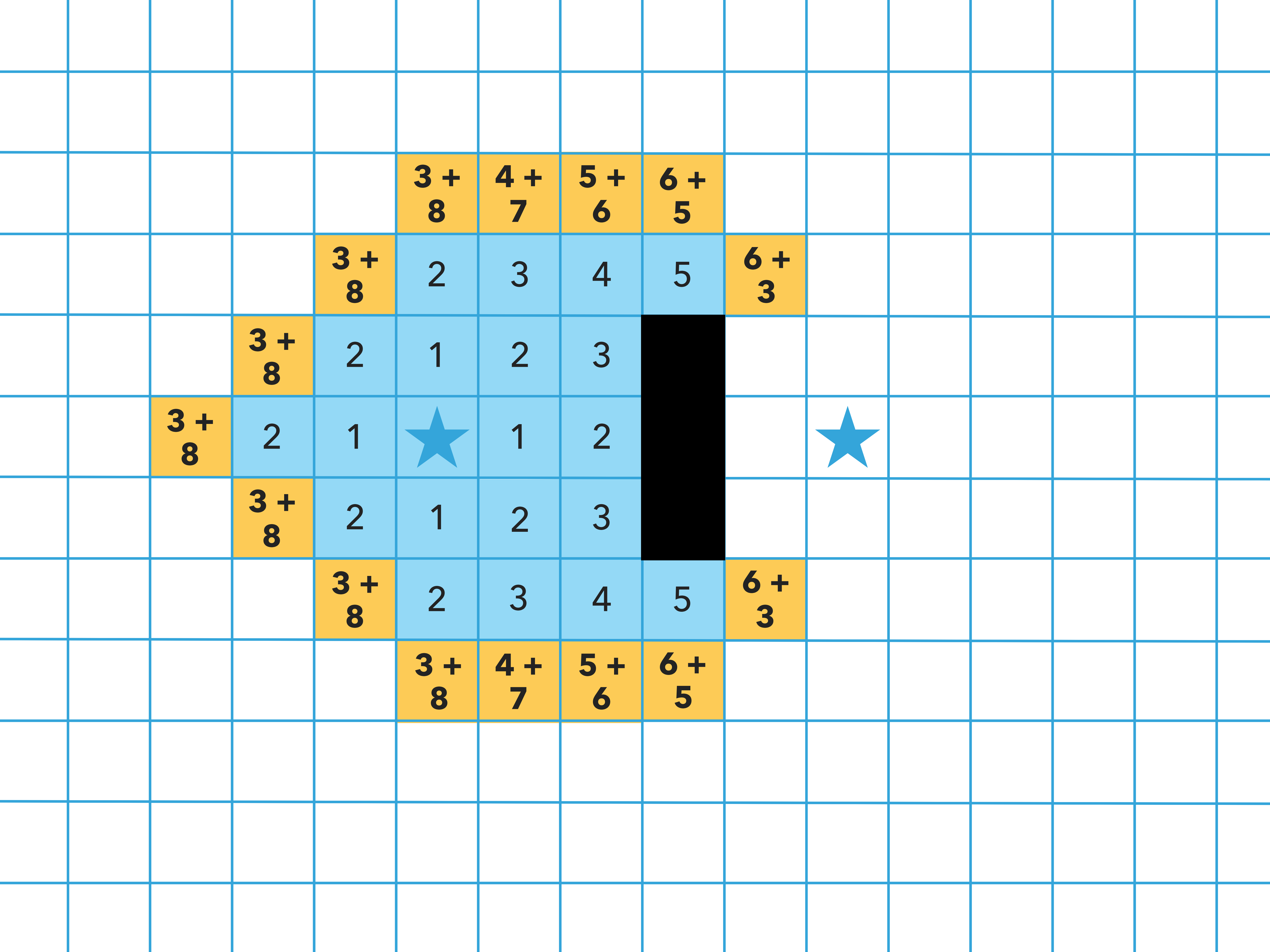

3 + 8
2 + 7
3 + 6
4 + 5
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2 + 7
3 + 6
4 + 5

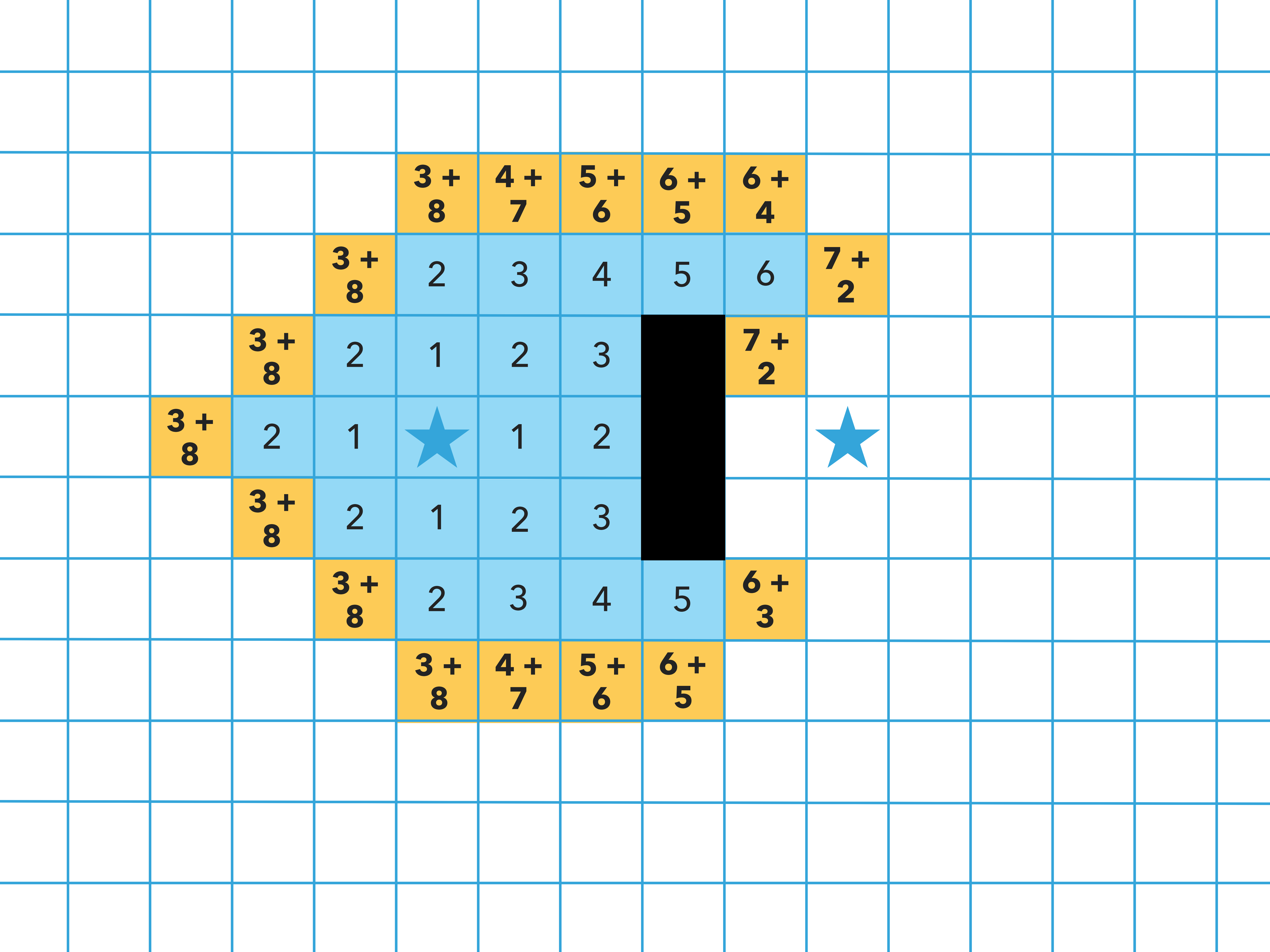

3 + 8
3 + 8
2
3 + 6
4 + 5
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3 + 6
4 + 5
3 + 8

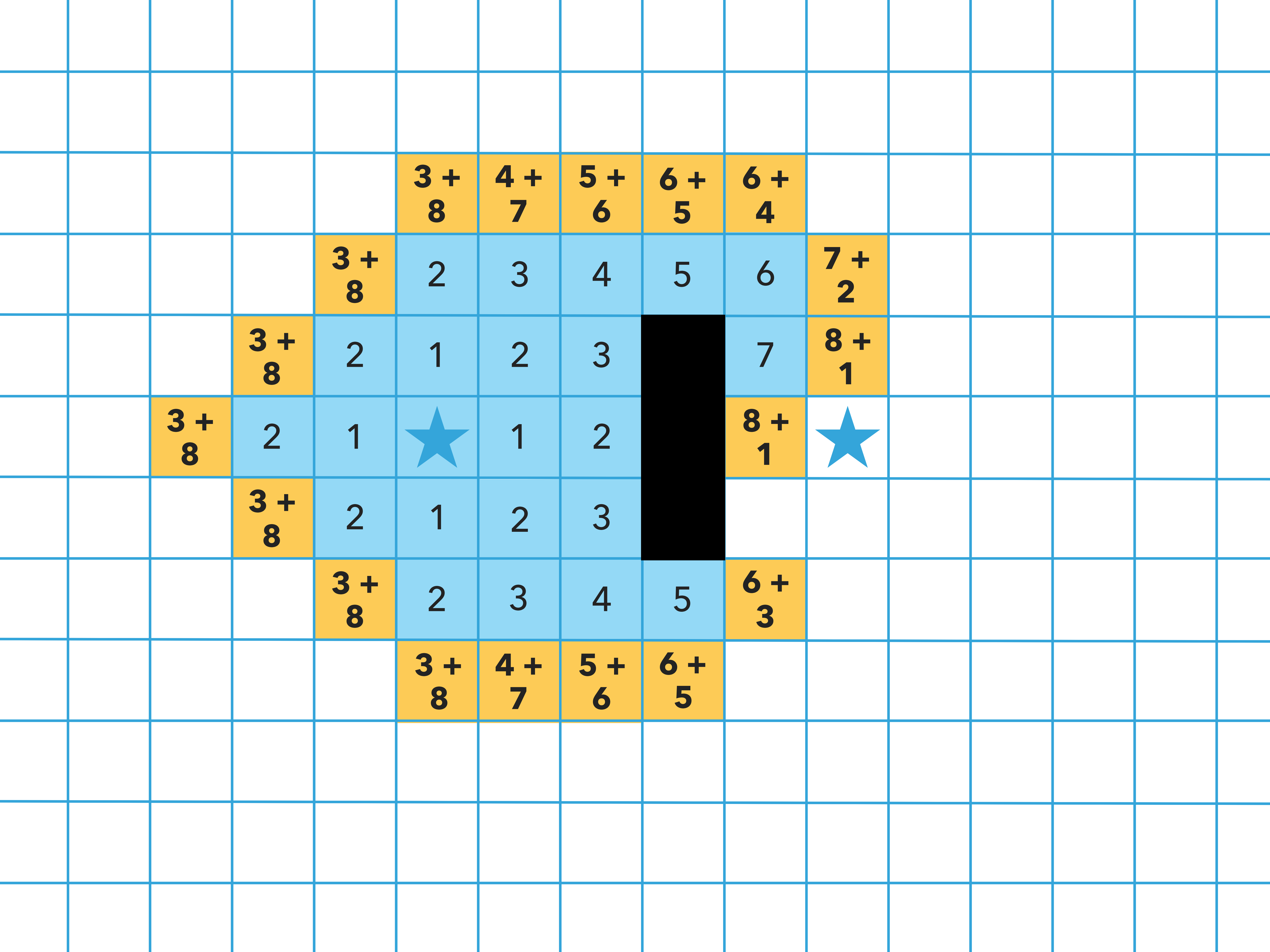

3 + 8 4 + 7
3 + 8 2 3 4 + 5
3 + 8 2 1 2 3
3 + 8 2 1 1 2
3 + 8 2 1 2 3
3 + 8 2 3 4 + 5
3 + 8 4 + 7

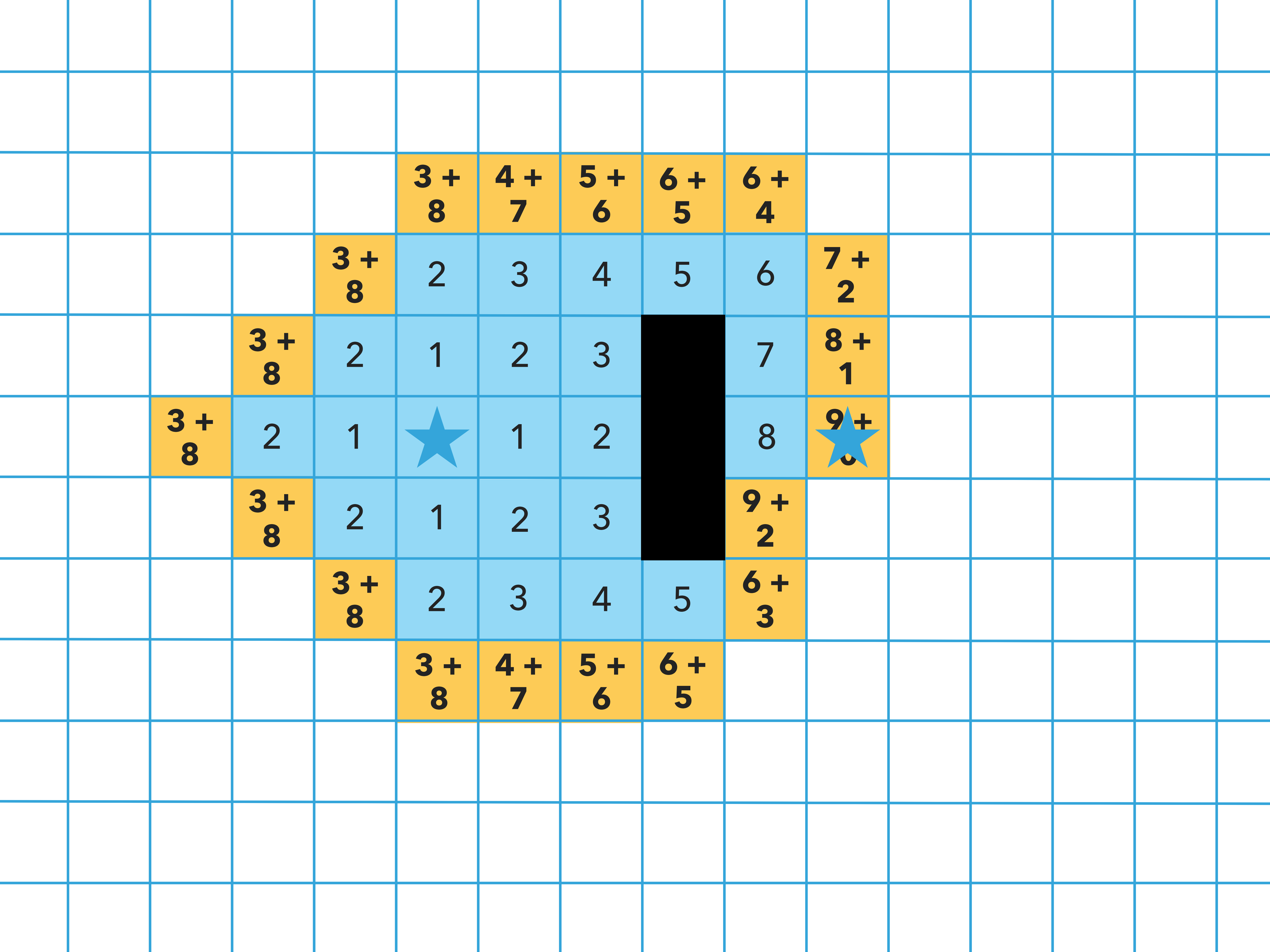

3 + 8
4 + 7
5 + 6
3 + 8
2
3
4
5 + 4
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5 + 4
3 + 8
4 + 7
5 + 6


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

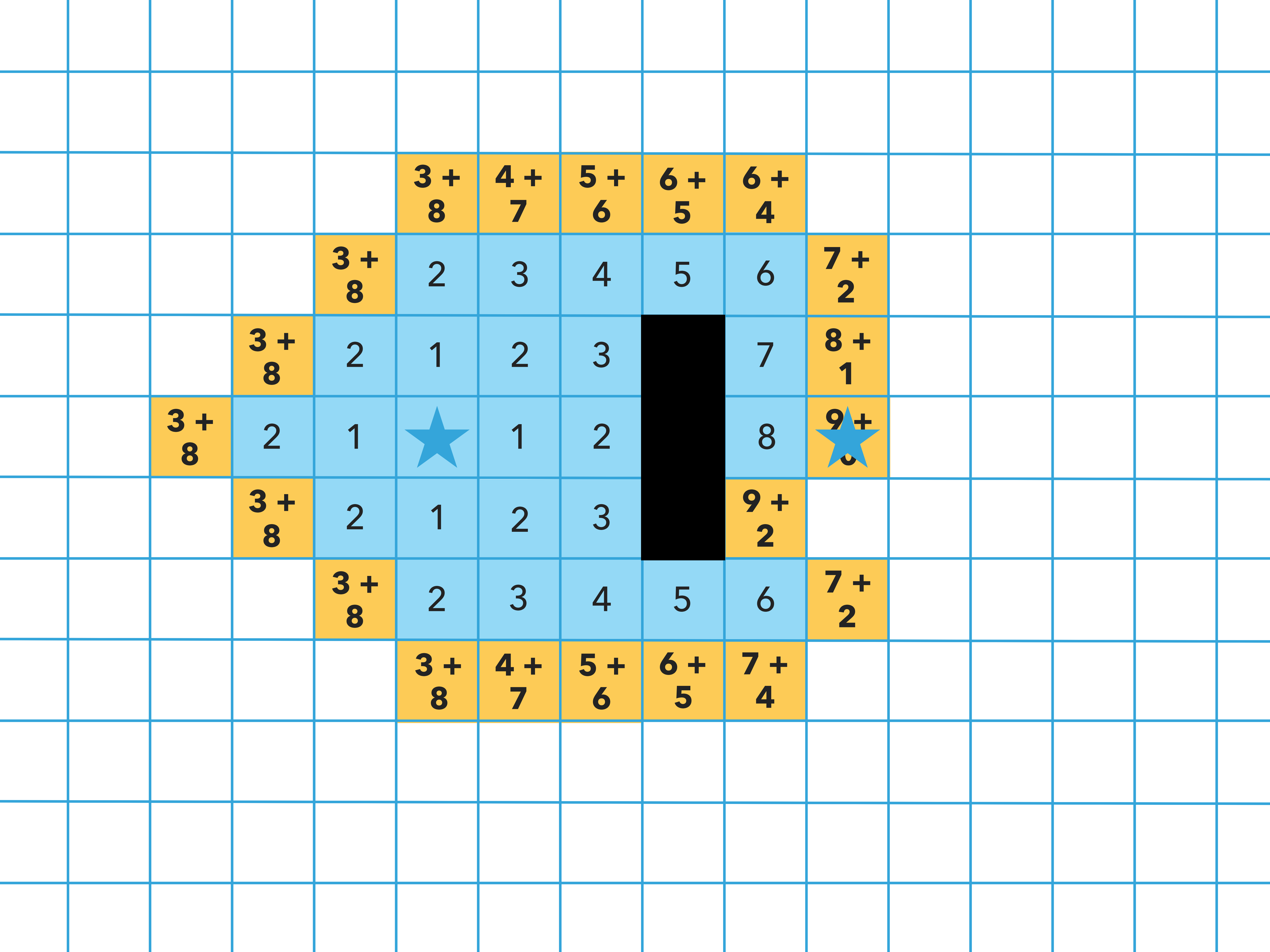

3 + 8
4 + 7
5 + 6
6 + 5
3 + 8
2
3
4
5
6 + 3
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5

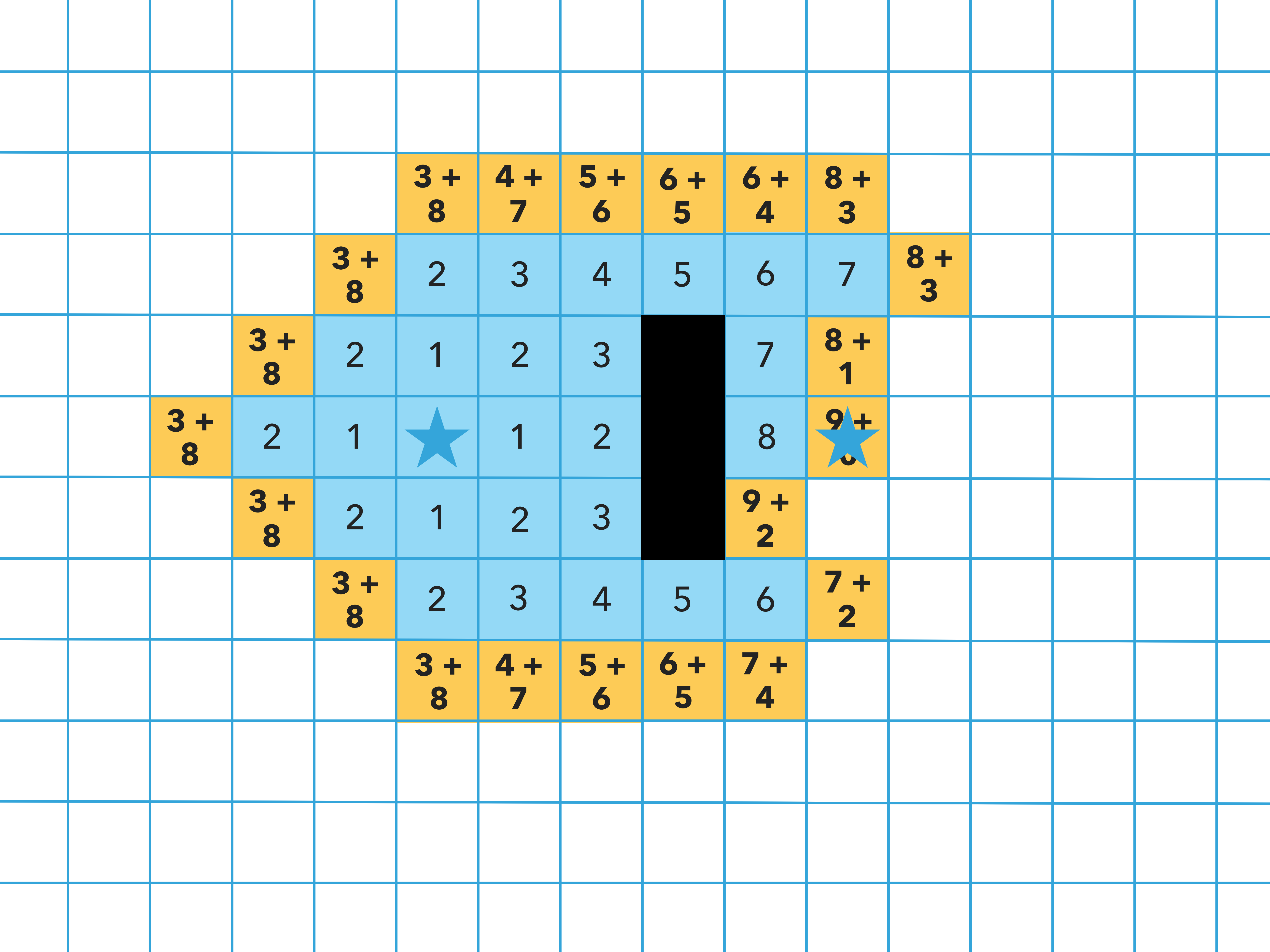

3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
3 + 8
2
3
4
5
6
7 + 2
3 + 8
2
1
2
3
7 + 2
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5


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

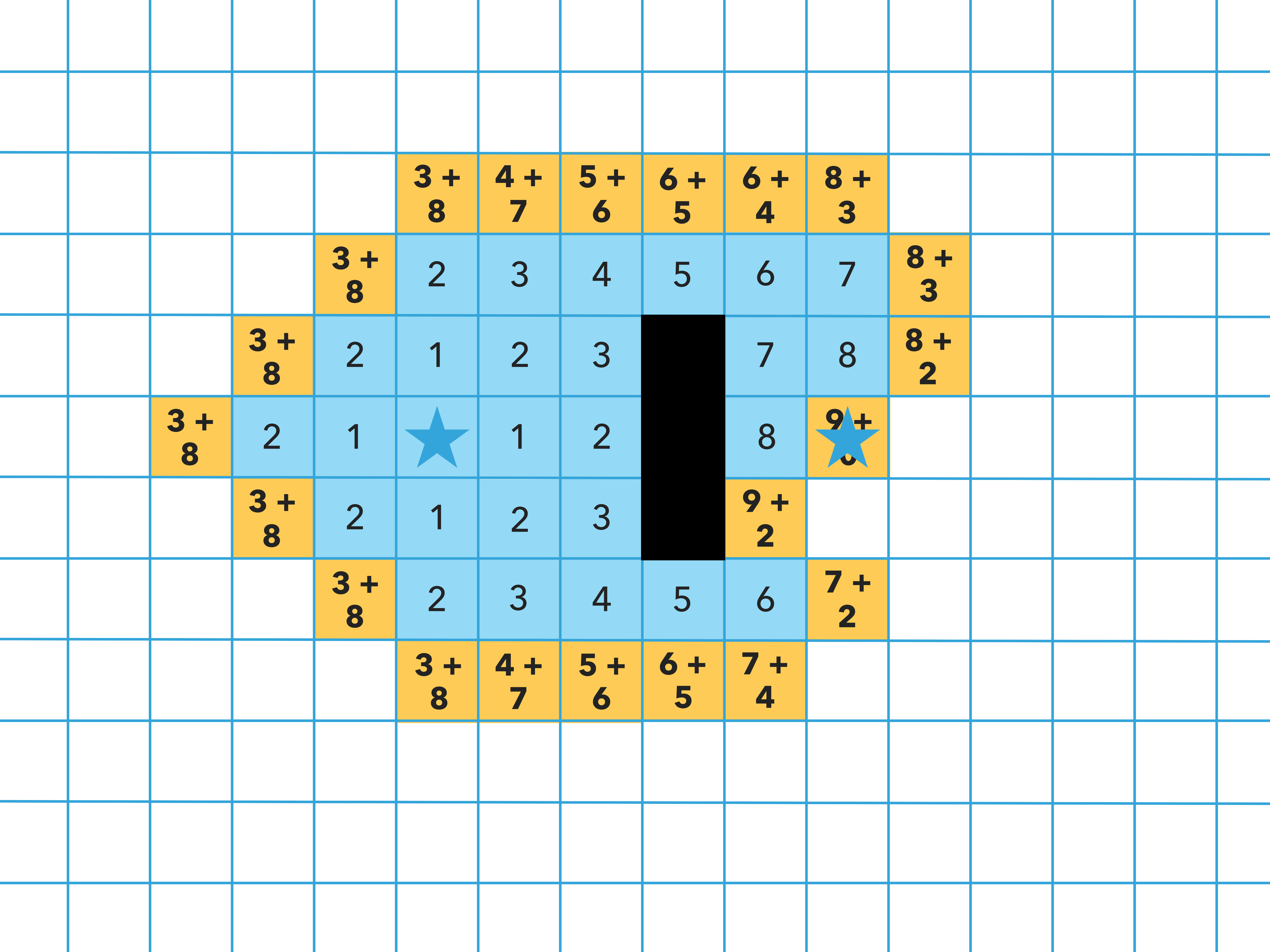


3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
3 + 8
2
3
4
5
6
7 + 2
3 + 8
2
1
2
3
7
8 + 1
3 + 8
2
1
1
2
8
9 +
3 + 8
2
1
2
3
9 + 2
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5

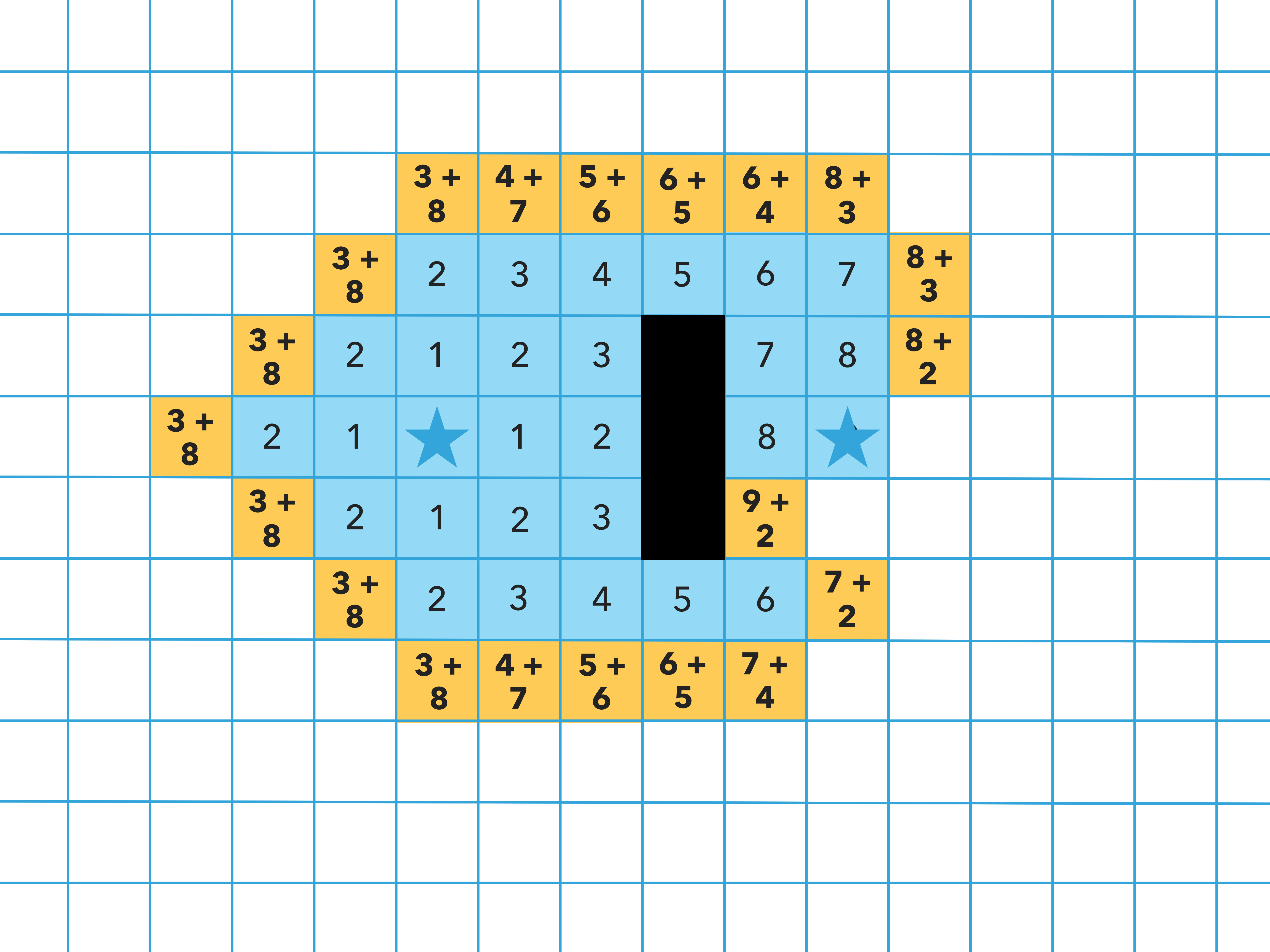

3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
3 + 8
2
3
4
5
6
7 + 2
3 + 8
2
1
2
3
7
8 + 1
3 + 8
2
1
1
2
8
9 +
3 + 8
2
1
2
3
9 + 2
3 + 8
2
3
4
5
6
7 + 2
3 + 8
4 + 7
5 + 6
6 + 5
7 + 4

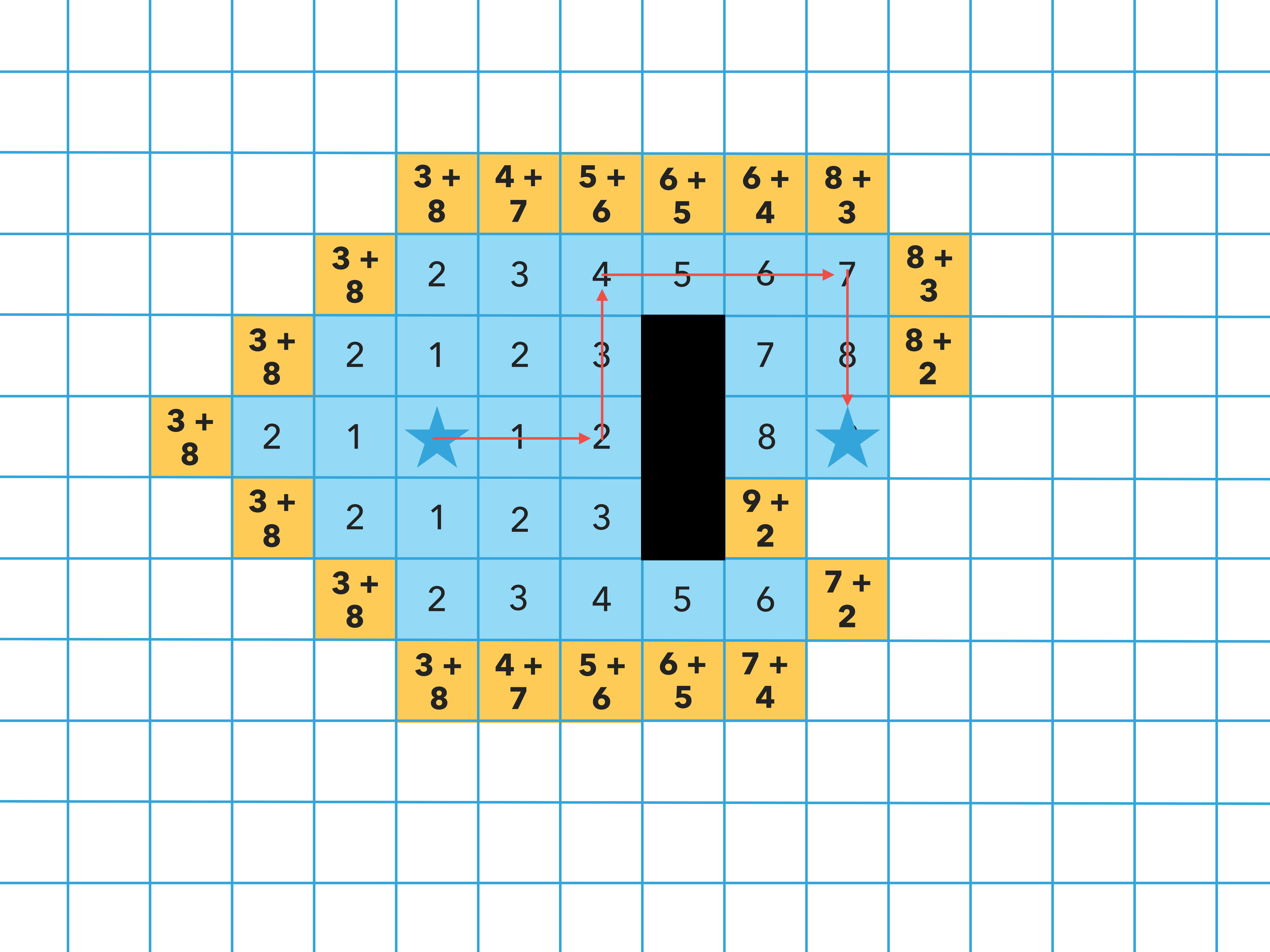


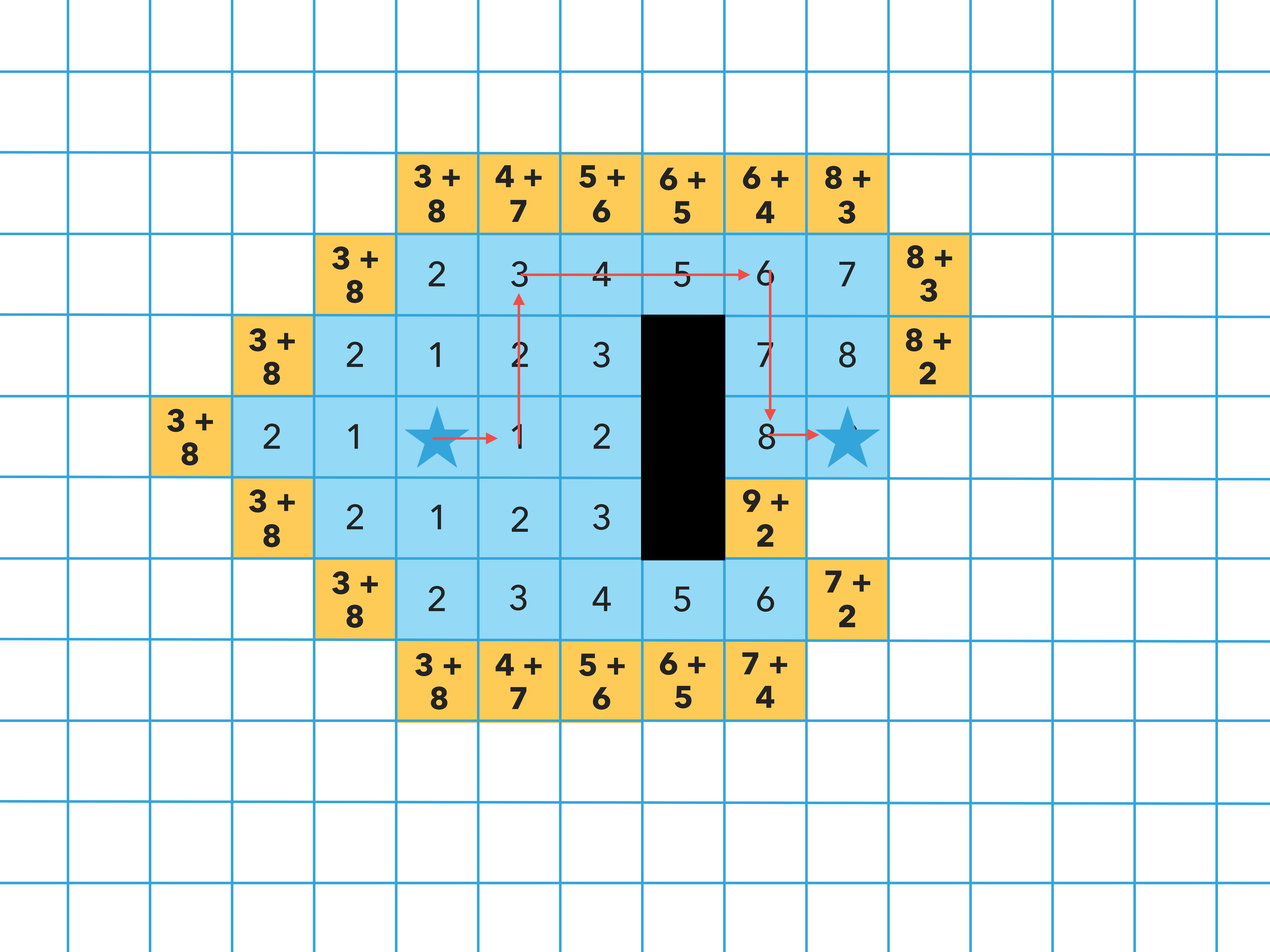
3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
8 + 3
3 + 8
2
3
4
5
6
7
8 + 3
3 + 8
2
1
2
3
7
8 + 1
3 + 8
2
1
1
2
8
9 +
3 + 8
2
1
2
3
9 + 2
3 + 8
2
3
4
5
6
7 + 2
3 + 8
4 + 7
5 + 6
6 + 5
7 + 4

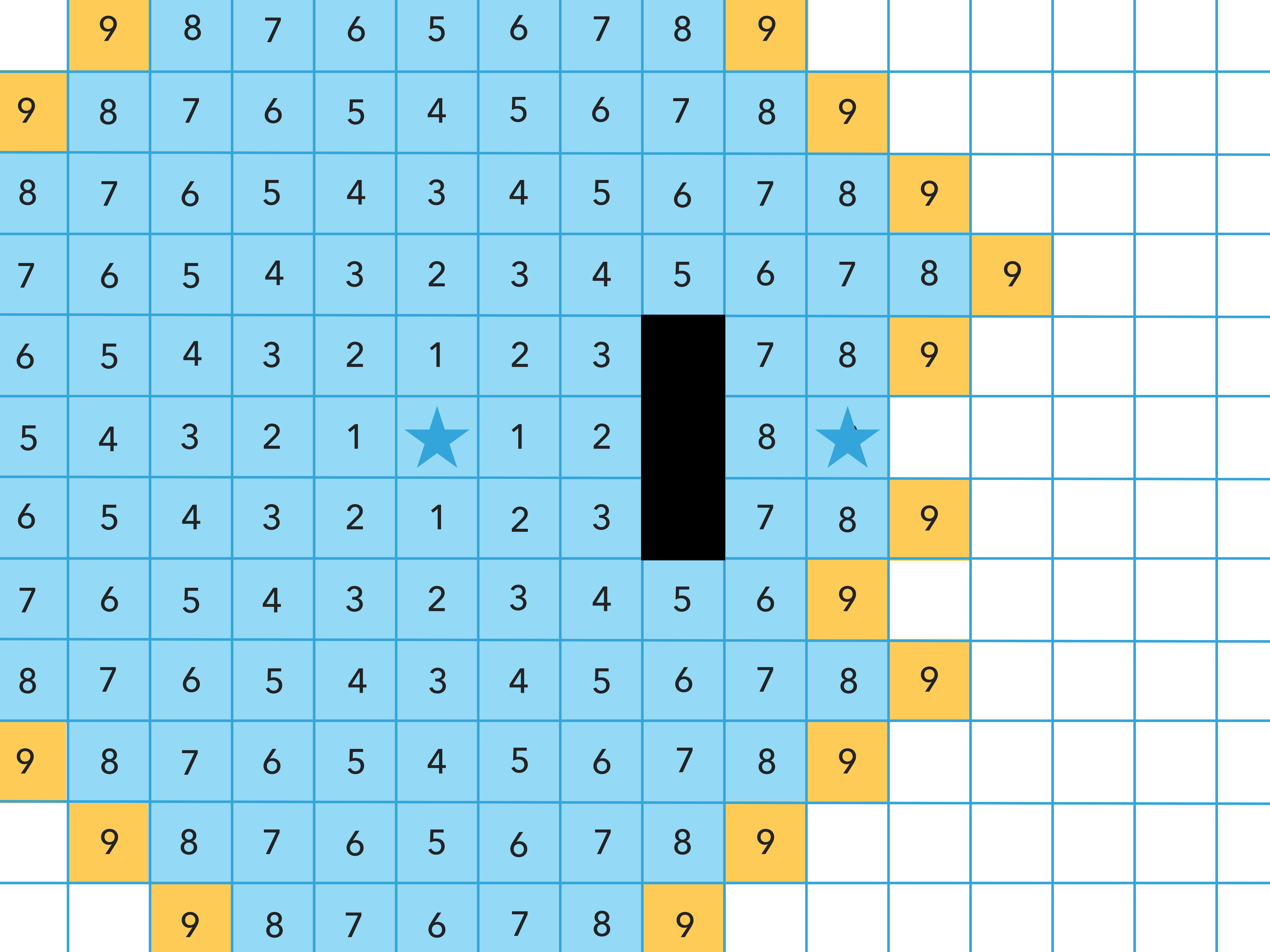


3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
8 + 3
3 + 8
2
3
4
5
6
7
8 + 3
3 + 8
2
1
2
3
7
8
8 + 2
3 + 8
2
1
1
2
8
9 +
3 + 8
2
1
2
3
9 + 2
3 + 8
2
3
4
5
6
7 + 2
3 + 8
4 + 7
5 + 6
6 + 5
7 + 4


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


9 8 7 6 5 6 7 8 9
9 8 7 6 5 4 5 6 7 8 9
8 7 6 5 4 3 4 5 6 7 8 9
7 6 5 4 3 2 3 4 5 6 7 8 9
6 5 4 3 2 1 2 3 7 8 9
5 4 3 2 1 1 2 8
6 5 4 3 2 1 2 3 7 8 9
7 6 5 4 3 2 3 4 5 6 9
8 7 6 5 4 3 4 5 6 7 8 9
9 8 7 6 5 4 5 6 7 8 9
9 8 7 6 5 6 7 8 9
9 8 7 6 7 8 9

A* (PSEUDOCODE)

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + heuristic

COMPARING DIJKSTRA AND A*

DIJKSTRA

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

A*

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + heuristic

COMPARING DIJKSTRA AND A*

DIJKSTRA

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

A*

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + **heuristic**

COMPARING DIJKSTRA AND A*

DIJKSTRA

- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength

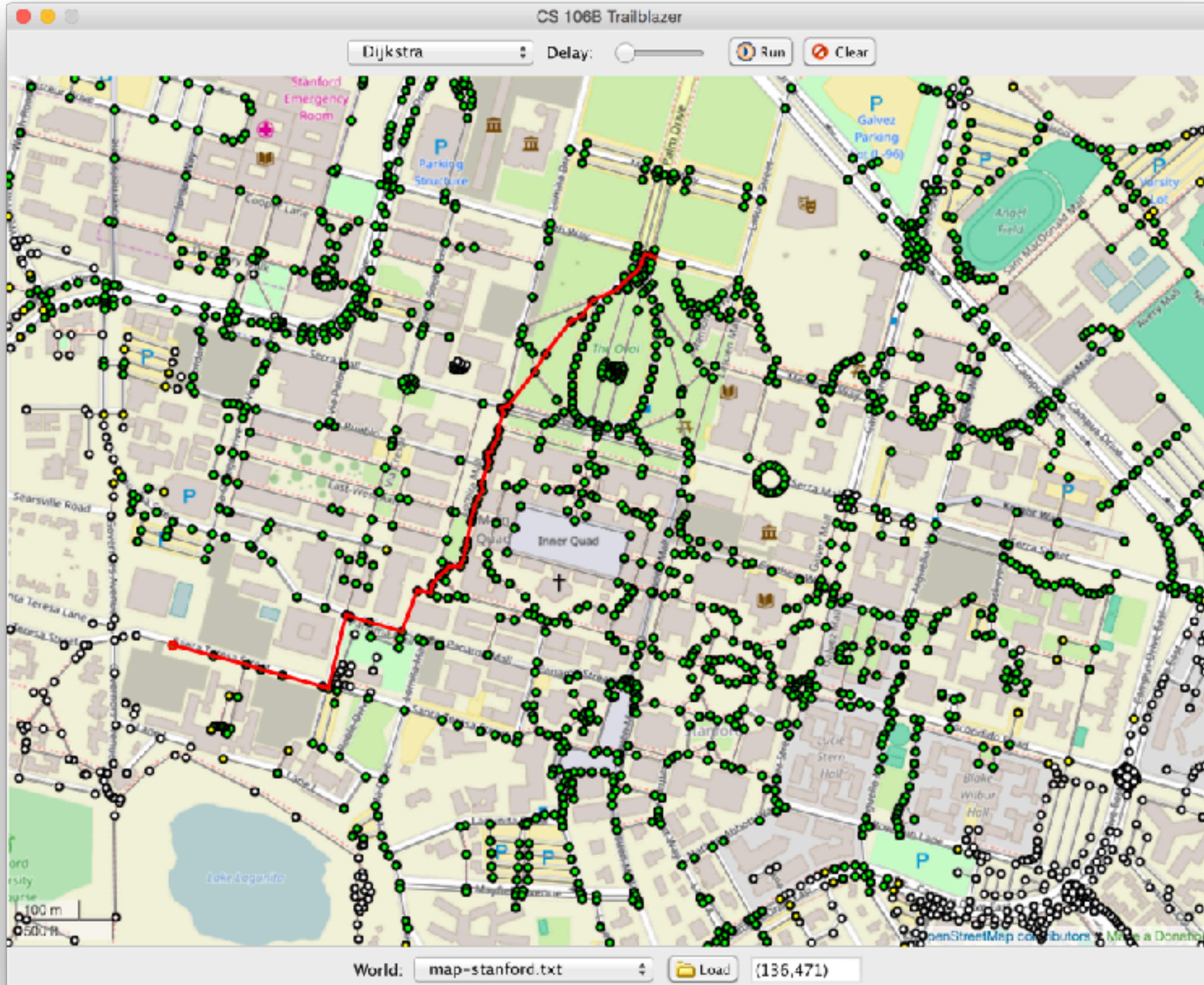
A*

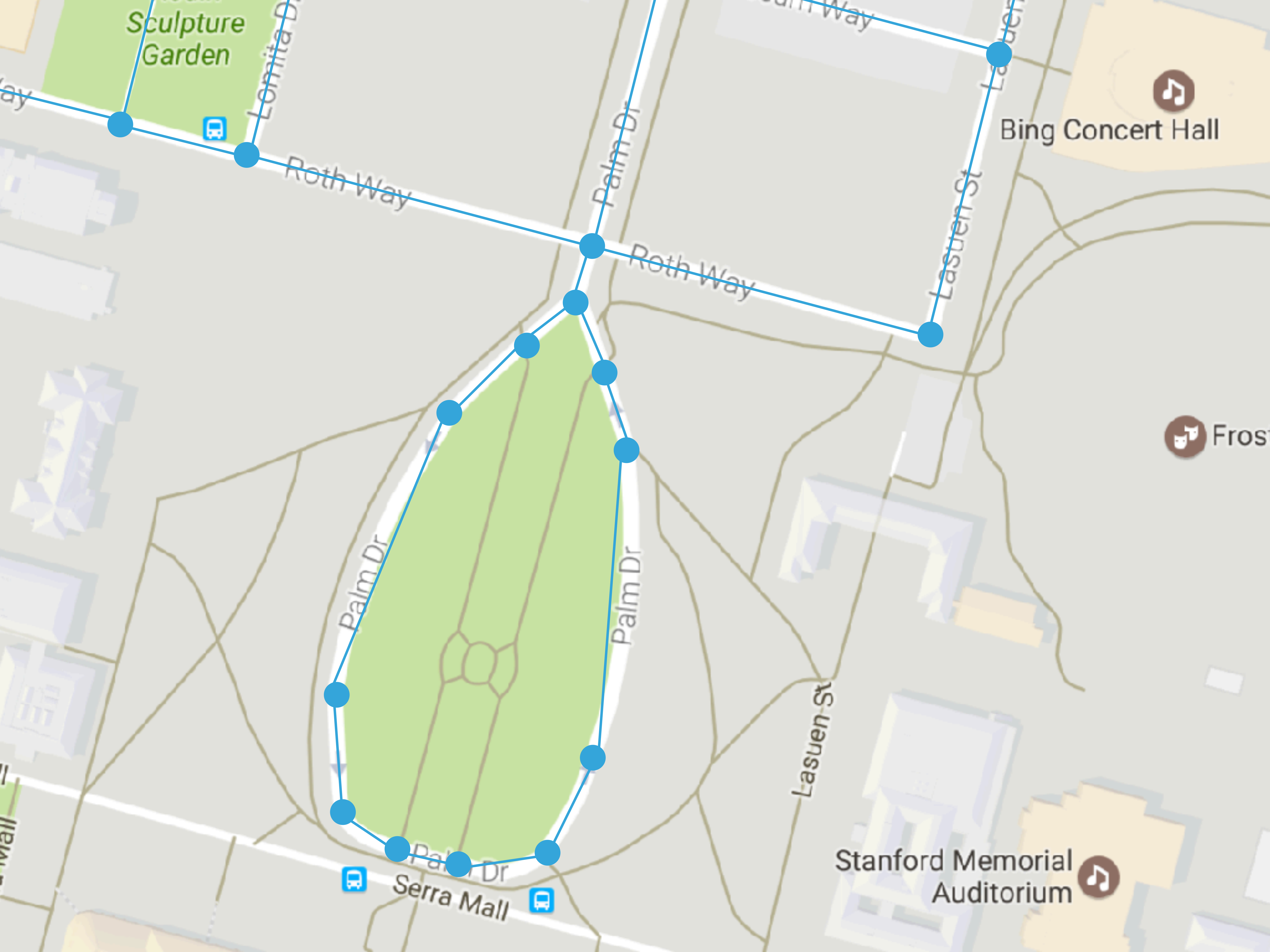
- create a path with just start node and enqueue into priority queue q
- while q is not empty and end node isn't visited:
  - p = q.dequeue()
  - v = last node of p
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q with priority pathLength + **0**

YOU WANT YOUR HEURISTIC TO BE AS LARGE AS POSSIBLE

BUT YOU NEVER WANT IT TO BE LARGER THAN THE ACTUAL COST.

GOOGLE MAPS


CS 106B Trailblazer
Dijkstra
Delay:
Run
Clear
Stanford Emergency Room
Parking Structure
Galvez Parking
Angell Field
Inner Quad
Lake Lagunita
100 m
500 ft
World:
map-stanford.txt
Load
(136,471)



Sculpture Garden
Lomita Dr
Roth Way
Palm Dr
Bing Concert Hall
Lasuen St
Frost
Palm Dr
Palm Dr
Lasuen St
Serra Mall
Stanford Memorial Auditorium

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million
- How many sets of directions would they need to generate?

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million
- How many sets of directions would they need to generate?
  - (roughly) N^2

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million
- How many sets of directions would they need to generate?
  - (roughly) N^2
- How long would that take?

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million
- How many sets of directions would they need to generate?
  - (roughly) N^2
- How long would that take?
  - 6×10^{15} seconds

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?
  - About 75 million
- How many sets of directions would they need to generate?
  - (roughly) N^2
- How long would that take?
  - 6×10^{15} seconds
  - Or… 190 million years

WHAT HEURISTICS COULD GOOGLE USE?

WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway

WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway
- Landmark heuristic
  - Find the distance from A and B to a landmark, calculate the difference (distance < abs(A - B))

WHAT HEURISTICS COULD GOOGLE USE?

- As the crow flies
  - Calculate the straight-line distance from A to B, and divide by the speed on the fastest highway
- Landmark heuristic
  - Find the distance from A and B to a landmark, calculate the difference (distance < abs(A - B))
- All of these and more?
  - You can use multiple heuristics and choose the max